Thèse de doctorat

NNT : 2025UPASG002

Distributed and Parallel Computing for very Large Neural Networks

Calcul réparti et parallèle pour les réseaux de neurones de très grandes tailles

Paris-Saclay University doctoral thesis

Doctoral School n° 580, Sciences et Technologies de l'Information et de la Communication (STIC)
Doctoral specialty: Mathematics & Computer Science
Research unit: Maison de la simulation

Thesis defended at Maison de la Simulation, Gif-sur-Yvette, the March 20, 2025, by

Quentin PETIT

Jury composition

Gaétan Hains Professor, Université Paris-Est Créteil	Président
Corinne Ancourt Professor, Les Mines Paris/PSL	Reviewer & Examiner
Michel Daydé Professor, Université de Toulouse	Reviewer & Examiner
Jack Dongarra Professor, University of Tennessee	Examiner
Christophe Calvin Research director, CEA - Université Paris-Saclay	Examiner
Geraud Krawezik Doctor, Flatiron Institute	Examiner

Thesis supervision

Nahid Emad Professor, Université de Versailles Saint-Quentin	Academic Supervisor
Chong Li Doctor, Huawei Technologies France	Industrial Supervisor

université PARIS-SACLAY
ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

Titre: Calcul réparti et parallèle pour les réseaux de neurones de très grandes tailles
Mots clés: Algorithme de recommandation, Réseaux de neurones, Calcul de graphe, Calcul haute performance, Optimisation de performance, Apprentissage profond

Résumé:

Les modèles de très grande taille sont aujourd'hui utilisés dans une multitude de domaines variés et ont permis de généraliser et de populariser l'utilisation du Deep Learning pour de nouvelles applications. Cependant, le traitement de ces tâches toujours plus générales a demandé une augmentation exponentielle de la taille de ces modèles, ce qui a également nécessité une puissance de calcul tout aussi importante pour les entraîner. Des solutions innovantes doivent être trouvées et déployées pour à la fois réduire la complexité des algorithmes existants et améliorer le déploiement de ces derniers dans un environnement massivement distribué avec des données de très grande taille. Le développement de techniques et de méthodes de calcul parallèle et distribué est essentiel pour optimiser l'utilisation des ressources disponibles, maximiser l'efficacité et réduire les coûts de calcul, répondant ainsi aux exigences croissantes de ces modèles.

C'est dans ce contexte que s'inscrit cette thèse. Nous proposons plusieurs contributions pour réduire les coûts associés à l'entraînement des grands réseaux de neurones dans un environnement massivement distribué. Nos travaux se concentrent principalement sur le traitement des données en amont du modèle pour améliorer la qualité des données, qui sont ensuite données en entrée du modèle, dans le but de faciliter son apprentissage. Nous nous sommes concentrés sur le traitement des données creuses, telles que les graphes, dont le traitement pose certains défis en raison de leurs structures complexes et de leurs tailles potentiellement très élevées. Nous proposons également de réduire la taille des données en entrée grâce à une réduction de dimension conservant suffisamment d'informations pour assurer une bonne précision du modèle tout en simplifiant les données d'entrée, réduisant par la même occasion la puissance de calcul nécessaire pour le traitement de ces données.

Title: Distributed and Parallel Computing for very Large Neural Networks
Keywords: Performance optimization, Deep Learning, High performance computing, Neural networks, Graph computing, Recommender system

Abstract:

Very large model sizes are now a very common feature, extending the range of applications for Deep Learning. However, this exponential growth in model size has led to an equally significant increase in computing power requirements. Innovative solutions need to be found and implemented to optimize current algorithms, reduce their complexity and make them easy to use and deploy in a massively distributed environment. The development of parallel and distributed computing techniques and methods to fully exploit available resources is crucial to maximizing efficiency and minimizing computation costs is very important to meet the ever-growing requirements of these models. In this context, we propose several contributions to reduce the costs associated with the training of neural networks in a massively distributed environment. Our contributions focus on the processing of data upstream of the model, in order to improve the quality of the data supplied to the neural network and facilitate its training. We focused on the processing of sparse data, such as graphs, which pose particular challenges due to their complex structures and potentially very large sizes. The processing applied to these data are designed to significantly improve the model's performance. Finally, we propose leveraging this processing to reduce effectively the size of the data, thereby decreasing the number of inputs while retaining sufficient information to ensure good model accuracy.

In loving memory of my grandfather

Acknowledgements

Je tiens à exprimer ma profonde gratitude à ma directrice de thèse, Prof. Nahid Emad, pour m'avoir offert l'opportunité de mener à bien cette thèse. Son soutien constant tout au long de mes recherches a constitué une source précieuse d'orientation et d'inspiration. J'ai eu la chance unique de bénéficier de ses vastes connaissances et de son expertise, qui ont enrichi cette aventure aussi bien sur le plan scientifique que personnel. Je tiens également à remercier chaleureusement mon superviseur industriel, Dr Chong Li, pour son accompagnement précieux tout au long de cette thèse. Son expertise technique, sa disponibilité et ses conseils avisés ont été essentiels dans le cadre de mes travaux. Je suis profondément reconnaissant pour les nombreuses discussions constructives que nous avons eues, qui ont non seulement enrichi mes recherches, mais également renforcé ma compréhension des enjeux industriels liés à mon sujet. Mes sincères remerciements vont également au Prof. Serge Petiton pour m'avoir initialement ouvert la voie de la recherche lors de ma formation d'ingénieur.

Je souhaite également remercier chaleureusement les membres du jury pour l'honneur qu'ils m'ont accordé en acceptant d'évaluer cette thèse. Je suis particulièrement reconnaissant envers Prof. Corinne Ancourt et Prof. Michel Daydé, rapporteurs de ce travail, pour le temps qu'ils ont consacré à la lecture de mon manuscrit ainsi que pour la qualité de leurs commentaires, qui ont permis d'approfondir et d'enrichir les réflexions présentées ici. Je remercie également Prof. Gaétan Hains, président du jury, pour avoir accepté ce rôle et pour sa bienveillance tout au long de la soutenance. J'adresse aussi mes remerciements à Prof. Jack Dongarra, Dr. Géraud Krawezik et au Directeur de recherche Christophe Calvin pour avoir accepté de faire partie du jury et pour la richesse des échanges lors de la soutenance. Leurs remarques pertinentes et leurs questions stimulantes ont grandement contribué à nourrir la réflexion autour de ce travail.

Je suis reconnaissant de l'accueil que m'a offert la Maison de la Simulation durant ces années, et d'avoir eu la possibilité de soutenir ma thèse dans leurs locaux. Un merci tout particulier à Martial Mancip pour le temps et l'énergie qu'il m'a consacrés lors de mes répétitions dans la salle du mur d'images, dont les conseils m'ont été d'une grande aide. Je souhaite également remercier toutes les personnes avec qui j'ai travaillé au centre de R&D de Huawei à Boulogne-Billancourt. Ces années ont été riches en apprentissages, j'ai eu la chance de travailler aux côtés de collègues inspirants dont les échanges enrichissants ont grandement contribué à l'aboutissement de ce travail. Ce fut un véritable plaisir d'évoluer dans un environnement aussi stimulant à vos côtés.

Sur un plan plus personnel, je tiens à remercier du fond du cœur ma famille, à commencer par mes parents, Isabelle et Christophe, pour l'éducation et les valeurs qui m'ont permis de devenir la personne que je suis aujourd'hui. Votre amour inconditionnel et les valeurs que vous m'avez partagées forment une base solide sur laquelle je peux m'appuyer à tout moment. Je remercie aussi ma soeur, Flavie, qui a toujours été là pour m'écouter et m'encourager. Mes remerciements s'étendent également à tous les membres de ma famille élargie, dont le soutien, les encouragements et la bienveillance ont été des moteurs essentiels tout au long de mon parcours. Un merci tout particulier à Annick, Vincent, Sabine et Guillaume, pour leur présence et leurs conseils précieux.

Enfin, je souhaite adresser ma gratitude la plus sincère à Chloé, qui partage ma vie depuis maintenant dix ans. Ton amour et ton soutien m'ont porté à chaque étape de ce voyage, et je ne saurais exprimer à quel point ta présence m'est précieuse. Un immense merci également à Martin, dont la générosité intellectuelle et le temps qu'il me consacre sans réserve ont été une source constante d'enrichissement et d'inspiration. Je tiens tout particulièrement à lui exprimer ma reconnaissance pour son aide précieuse lors de la relecture de mon manuscrit de thèse. Pour conclure, j'adresse toutes mes pensées et mes encouragements à Ruiwen et Lauer pour la dernière ligne droite de leur thèse. Je leur souhaite beaucoup de réussite et d'épanouissement dans cette fin d'aventure.

Contents

List of Figures

List of Tables

Notations

a	A scalar
$\boldsymbol{a}$	A vector
$\boldsymbol{A}$	A matrix
A	A tensor
$\mathcal{P}_n$	A problem of size n
$\mathcal{K}_r$	A order-r Krylov subspace
$\mathbb{A}$	A set
$\mathbb{R}$	The set of real numbers
$\mathbb{C}$	The set of complex numbers
$\{0, 1\}$	The set containing 0 and 1
$\{0, 1, ..., n\}$	The set of all integers between 0 and n
$[a, b]$	The real interval including a and b
$]a, b]$	The real interval excluding a but including b
$\mathcal{G}$	A graph
$\boldsymbol{a}_i$	Element i of vector $\boldsymbol{a}$, with indexing starting at 1
$\boldsymbol{A}_{i,j}$	Element i, j of matrix $\boldsymbol{A}$
$\boldsymbol{A}_{i,:}$	Row i of matrix $\boldsymbol{A}$
$\boldsymbol{A}_{:,j}$	Column j of matrix $\boldsymbol{A}$

Chapter 1

Introduction

1.1 Context and motivations

Artificial intelligence (AI) is currently expanding at an unprecedented rate, marking a revolution in many sectors and redefining the boundaries of what is technologically conceivable. Once considered a futuristic domain reserved for science fiction, AI is now ubiquitous in our daily environment, from intelligent voice assistants to personalized recommendation systems, as well as increasingly sophisticated industrial and medical applications. This evolution is due to major advances in machine learning, neural networks and access to massive data, as well as constant improvements in data processing and computation capacities.

The importance of AI continues to grow, not only because of its potential to transform the global economy [86], but also because of its central role in solving complex problems on a world-scale [103]. AI also poses major societal and scientific issues relating to computing power, energy consumption and scalability. The development and training of AI models, in particular large-scale neural networks, requires considerable computational resources, placing significant demands on computing power. This necessitates the use of massive data centers and performance-intensive infrastructures, resulting in elevated energy consumption and contributing to a substantial ecological impact [63]. Indeed, the carbon footprint of AI, particularly linked to model training, has become a major concern. In addition, the scalability of AI solutions represents another challenge: as models grow in size and data

becomes more complex, the ability of infrastructures to support this growth becomes limited, raising questions about resource efficiency and the long-term viability of current solutions. These challenges call for innovative solutions to make AI more sustainable, eco-efficient and scalable, while balancing its benefits with its environmental and social costs [267].

To meet the challenges posed by AI, solutions need to be implemented at all levels. On the algorithmic level, complexity reduction techniques such as model compression, quantization and dimensionality reduction help lower computational demands while maintaining performance. Additionally, certain model architectures are inherently more hardware-friendly; for example, transformers enable efficient parallelization due to their self-attention mechanism [276], while mixture-of-experts (MoE) [74] models optimize memory usage by activating only a subset of their parameters per inference [249].

On the software side, modern deep learning frameworks such as TensorFlow [210], PyTorch [148] or MindSpore [39] provide powerful tools for developing distributed models, but managing parallelization and inter-machine communications remains a major challenge. On the hardware side, specialized architectures such as GPUs, TPUs and dedicated AI accelerators have considerably improved processing capabilities. However, scaling these models on massive parallel infrastructures, such as supercomputers, requires careful coordination between hardware, algorithms and software [160].

A major challenge lies in developing optimized methods to fully exploit these architectures, ensuring that AI systems make efficient use of available computing power. Despite advances in hardware-aware model designs, such as sparsity-aware models [64] and optimized low-precision computing [110], significant challenges remain. These include increasing model complexity, machine heterogeneity and workload balancing, all of which must be addressed to build AI systems that are not only high-performing but also accessible and sustainable.

The objective of this thesis is to enhance the performance of AI models by proposing innovative solutions to reduce the costs associated with their development and training. Specifically, we present our contributions that aim to minimize the training costs of AI neural networks in massively distributed environments. Our contributions focus on the processing of data upstream of the model, in order to improve the quality of the data supplied to the neural network thereby facilitating and optimizing

the training process. Particular attention is given to the processing of sparse data presented in graph form, which represents important challenges due to their complex structures and potentially large sizes. The preprocessing techniques we propose are designed to significantly enhance neural network performance. Additionally, we leverage these preprocessing methods to effectively reduce the size of the data, thereby decreasing the number of inputs while preserving sufficient information to maintain high model accuracy.

1.2 Timeline of deep learning

The history of Deep Learning (DL) requires a more general interest in the history of machine learning (ML), which will later lead to the emergence of deep learning.

Machine learning is a vast field of study that brings together a set of mathematical and statistical approaches to enable computer systems to solve problems without explicitly programming the solutions to solve them. ML models take data and analyze it to create a model that is then used to make predictions with new data. This is a broad field that has been supported by the desire to create intelligent algorithms, which can generally be grouped under the term of artificial intelligence.

1.2.1 From machine learning to deep learning

Even if the term machine learning has been popularized recently, it was first used in 1959 [239] where ML is defined as an approach to learning from experiments to reduce the workload of the programmer. For the last twenty years, the investigation in the field of artificial intelligence has been very promising. Several projects have generated a lot of excitement around AI during this period [204, 87]. It is also at this era that was introduced the neural system with a weight system for the connections between neurons [119], which contributed to the discovery of the perceptron [232] (Figure 1.1). The perceptron principle is still used today in many neural network architectures.

This period of enthusiasm has been replaced by a disillusionment about AI. The performance of the current systems and the small amount of data were not enough to meet the ambitions in the field. Moreover, the neural network structures did not allow to solve complex problems because of the large

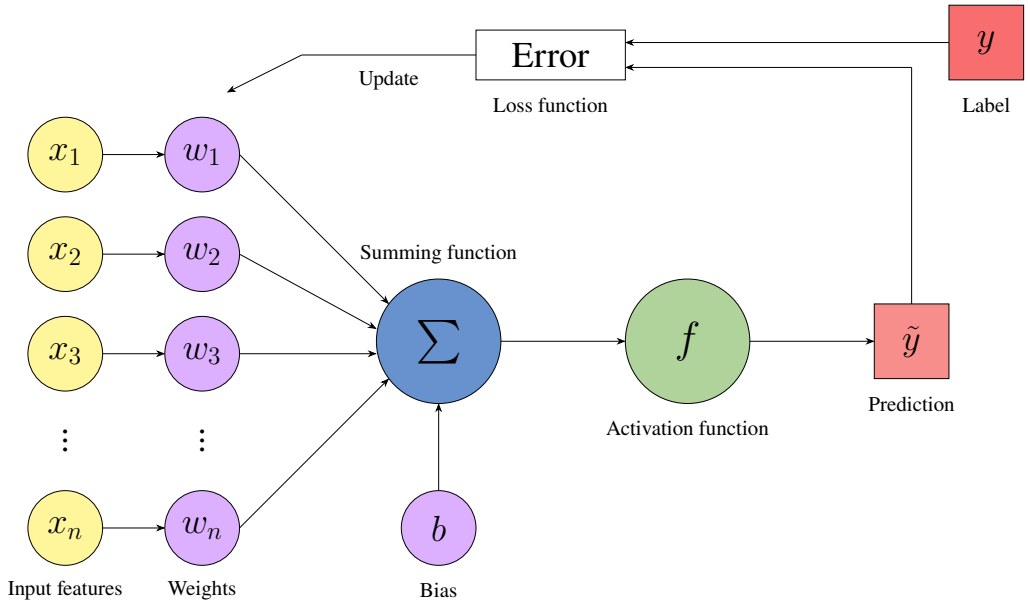


Figure 1.1: Overview of a single layer perceptron, a basis neural networks with only one perceptron.

number of parameters. The gap between the promised results and the reality led to a significant decline in investment in the field of neural networks.

The resurgence of interest in neural networks has been particularly driven by the exponential growth in data availability, often referred as *Big Data*. This vast influx of information, combined with advances in computational power, has enabled the effective processing of large datasets, facilitating robust model training.

In addition to data abundance, machine learning advancements have been fueled by the emergence of more sophisticated algorithms, such as convolutional neural networks. These algorithms have become feasible to implement and train due to significant improvements in computational capabilities, particularly the advent of graphics processing units. These hardware accelerators are ideal for matrix manipulation, a critical operation in neural network architectures since it intervenes in almost all stages of the algorithms.

This period also marked the formal distinction of the subfield of deep learning, whose much deeper architecture gives them the ability to learn complex representations from large-scale data.

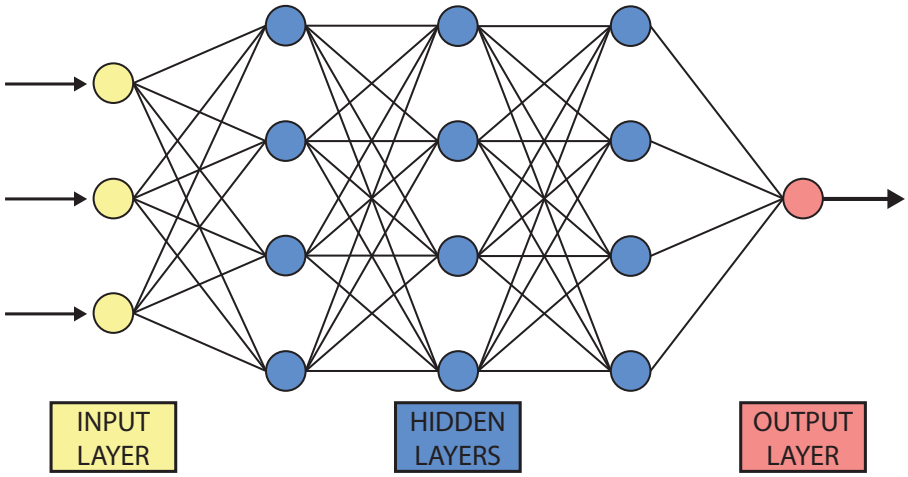


Figure 1.2: Illustration of multilayer perceptron.

1.2.2 The deep learning revolution

In 1986, [233] and [234] introduced back-propagation, a new learning procedure. This method updates the weights of a neural network during training [167]. It allows propagating the error backward through the network layers, starting from the difference between the predicted output and the desired theoretical output. Error minimization is an iterative process that repeatedly updates the network's weights and biases until convergence (purple elements in the figure 1.1). This approach significantly accelerates the convergence of weights and opens the way for training very large neural networks, which were previously untrainable due to essentially computational limitations and inefficient learning algorithms. Back-propagation has been instrumental in making the growth and advancement of deep learning.

Deep learning is a sub-field of machine learning that utilizes artificial neural networks with multiple layers to automatically learn and extract meaningful patterns or representations from data (Figure 1.2). The term deep learning is originally used to designate methods to increase the model sizes [98]. To make deep neural networks allows the computer to learn complicated features or concepts by building them out of simpler ones. Deep learning models have capacity for self-improvement and adaptation, enabling them to converge empirically on a high-performance solution to complex tasks whose solution cannot be put into mathematical form. It is particularly effective for tasks involving large datasets and complex structures, such as image recognition, natural language processing and speech processing.

As the 1990s began, the excitement surrounding neural networks started to fade. The limitations of the era's hardware, combined with the theoretical challenges of training deep networks, pushed deep neural networks to the sidelines of artificial intelligence research. Instead, other machine learning methods, such as support-vector networks [47], began to dominate the landscape due to their superior performance on many practical problems and their much lower computational power requirements for learning.

But at that time, the challenge lays mainly in the problem of overfitting. Overfitting is a phenomenon where a model performs too well on training data but fails to generalize to unseen data. This is a frequent phenomenon [114] that is observed when the model starts to learn the noise that exists among the training values. This problem was amplified by the limited size of the available datasets. In the absence of sufficient data, deep neural networks tended to memorize rather than learn, making them unreliable for real-world tasks.

Another challenge was the vanishing gradient problem [128]. In a neural network, the training process involves adjusting the model's internal parameters using a method called gradient descent [49]. However, when the network becomes deeper, the gradients used to adjusting the model's weight would diminish as they were propagated backward through the layers. By the time they reached the initial layers, these gradients were often so small that they became ineffective, halting meaningful learning in deeper networks.

Despite the challenges, a small group of researchers continued to advance the field of deep learning. Their efforts focused on developing solutions to mitigate challenges issues, in particular to prevent overfitting [94, 222, 295]. In 1998, the convolutional neural networks [168] were introduced for image recognition. This model was designed to process image data by mimicking the way the human visual cortex operates. It introduced concepts like filters and feature maps, allowing the network to focus on different aspects of an image like distinctive patterns or specific textures. Although constrained by the technology of the era, this model established the foundations for breakthroughs that would come years later.

The introduction of AlexNet in 2012 marked a decisive shift in the field of deep learning. AlexNet is a deep CNN that stunned the world by achieving unprecedented accuracy in an image recognition

competition in 2012 [235]. AlexNet's use of hardware accelerators for computation allowed to drastically reduce training times and made deep learning models more practical. The success of AlexNet wasn't just a technical milestone; it was a cultural shift. The AI community, which had largely dismissed neural networks for decades, was compelled to recognize this technical progress. For the first time, deep learning demonstrated its ability to outperform traditional machine learning techniques on a real-world, large-scale task.

The impressive accuracy of AlexNet put the light back on deep learning and led to a veritable craze for pushing back the boundaries of deep neural networks. Deep learning methods have been applied to many different types of data and to a wide variety of applications, greatly extending the limits of what neural networks can do.

Another true revolution came in 2017 with the introduction of the transformer architecture [276]. Transformer introduces the mechanism of self-attention, which enables the capture of dependencies and long-distance relationships in input data. This mechanism has completely disrupted the state of the art of natural language processing, which is the field that focuses on enabling computers to understand, interpret and generate human language [303]. Indeed, the most accurate previous models were recurrent neural networks, which were able to keep the few previous elements in memory, but struggled with long-term dependencies. This innovation formed the backbone of large language models like BERT (Bidirectional Encoder Representations from Transformers) [147] and GPT (Generative Pre-trained Transformer) [225], which redefined tasks like machine translation, text generation or question answering.

The Transformer architecture not only revolutionized natural language processing but also paved the way for a new paradigm in AI: models capable of generalizing across multiple tasks. With the rapid scaling of deep learning, researchers began exploring architectures that could perform well beyond specialized domains, leading to the emergence of generalist AI models.

1.3 Generalist AI models emerging trends

Deep learning has been a field of research for a long time, but its popularity has exploded in recent years. Thanks to the convergence of high-performance hardware systems, data availability and advancements in AI, the rapid growth of deep neural networks has revolutionized many fields, from image recognition to machine translation. Powerful language model and image generator tools have been made available to the general public, facilitating access to technologies once reserved for experts. It is precisely the versatility of these tools that makes them so easy to use and the results quite convincing.

However, to be able to respond to a wide variety of tasks without having been specifically trained for them, AI models have grown exponentially in size and complexity [246]. In a brutal approach, the model is fed with as much data as possible, and its size is expanded to the maximum so that it can be generalized. Today, there is a race to develop ever more powerful models for ever larger tasks. But this poses a number of problems to be solved and challenges to be met. These include problems related to the computational power required and the scalability of algorithms and architectures. In terms of methods, the development of very large models requires vast amounts of data for training, which poses problems of data availability and data quality. The carbon footprint and the development cost of training and operating large AI models are also growing challenges, posing problems of sustainability, equitable access to technology and concentration of power among a few large corporations.

The emergence of DeepSeek models in this last time have confirmed this landscape. The models DeepSeek-R1 [107] and DeepSeek-V3 [182] demonstrated that highly capable, generalist AI systems can be developed with a focus on algorithmic efficiency. By offering a competitive performance at a fraction of the computational cost, this model challenges the dominant paradigm of scaling models indefinitely and highlights algorithmic strategies to build generalist AI. Its open-source and cost-effective approach has sparked debates on the future of AI accessibility and global AI competition. This development underscores a shift where efficiency and feasibility become as crucial as raw model power.

1.4 Socio-economic and ethical impact

The design and the training of generalist AI models are extremely resource-intensive due to their size. The number of parameters is so large and the amount of data to be ingested so vast, that training an AI model today requires hundreds or even thousands of hardware units working together in parallel for several months at a time. According to some estimates based on available information [67, 245], Gemini 1.0 Ultra [269] presented in December 2023 required the use of 55 000 Google TPU v4 for around 100 days to train this model. For GPT-4 [2], estimates suggest 25 000 A100 GPUs for approximately 3 months. This puts the training costs for these two models at between 30 and 40 million dollars. This cost does not take into account the training of intermediate models that were trained but did not meet performance requirements. The cost of such training is counted in millions of dollars and requires large amounts of electricity [214]. In addition to training, using a large model requires a significant amount of computation to traverse the network and obtain results, which in turn leads to higher electricity consumption [263]. As a result, the creation of such a very large model is only possible and conceivable for a small number of major international corporations and governments [137], limiting access to these technologies to only those with the necessary resources. This concentration of strategic technological power raises important questions about the equity and accessibility of AI. Indeed, small businesses, academic institutions and developing countries may find themselves at a disadvantage, unable to compete with the technology giants. The models proposed by DeepSeek show that it is conceivable for these smaller participants to build and train accurate models with more reasonable costs thanks to algorithmic strategies for reducing costs.

Moreover, the environmental impact of training and using these models is a growing source of concern. Massive energy consumption contributes to the emission of greenhouse gases, exacerbating global climate challenges. This increase is unsustainable in the long-term, because if it continues at this current speed, the predicted cost of training for the largest AI model would be higher than the estimated GDP of the United States over the same period [120]. It is therefore crucial to develop more efficient and sustainable methods for training and operating AI models.

To meet these challenges, solutions must be found at all levels to reduce computing costs and the ecological footprint, while improving accessibility to a wider range of actors. One of the key avenues

for addressing these challenges lies in advances in high-performance computing. Understanding how HPC has developed over time, and the impact of these advancements on AI, provides essential context for exploring potential solutions to the growing computational and environmental demands of modern AI systems.

1.5 The evolution and impact of high-performance computing

Over the decades, technological advancements have significantly increased the speed and capacity of supercomputers. The term *supercomputer* is used to describe computers designed to achieve the highest possible performance using the technologies available at the time of their design. The first massively parallel computers emerged in the 1970s, with the ILLIAC IV being the first computer of this type with 64 floating-point units [130]. The 1980s saw further advancements in massively parallel computing, exemplified by the Connection Machine [124], which had more than 65,000 1-bit processors [192]. These early supercomputers were capable of several million operations per second [240, 296]. In the 1980s, the parallelization of computations made it possible to process tasks simultaneously, thereby increasing processing speed. In the early 1990s, the concept of computer clustering began to take shape as an effective approach to achieve high-performance computing. There has been an increasing trend to move away from expensive, specialized, proprietary parallel supercomputers, towards computer networks that are more affordable [252]. Numerous tools were developed during this period, such as Parallel Virtual Machine (PVM) [18] or Message Passing Interface (MPI) [58], which have become key tools for enabling communication between nodes in the distributed systems. These tools allowed programmers to write distributed applications more easily, thanks to an abstraction of the hardware resources for the programmer. It was also during this period that job schedulers [75] were democratized to manage and optimize the execution of computational tasks in distributed or parallel computing environments. For the last decades, the most powerful machines are parallel and distributed architectures: they are equipped with a large number of cores on a node. Each node is connected to the others by a network that will give them the capacity to communicate. These are complex architectures, whose characteristics must be taken into account

to develop algorithms that exploit their maximum power. High-performance computing is used to design adapted algorithms capable of solving complex problems using all the computing power of these supercomputers.

High-performance computing (HPC) can be considered as a set of algorithmic concepts and programming techniques for processing large among of data and running complex calculations on advanced computing architectures. In [257], the authors define HPC as a field of endeavor that relates to all facets of technology, methodology and application associated with achieving the greatest computing capability possible at any point in time and technology. It can also be be considered representing the set of technologies, hardware and software that are used to improve computing performance.

Initially, HPC was used mainly in scientific research and engineering for complex simulations and the processing of large amounts of data. With time, the use of HPC has been extended to various sectors such as finance, medicine and entertainment, where intensive computing power has become crucial for data processing and modeling. HPC systems operate at much higher speeds than traditional desktops or servers, making them capable of performing complex tasks in real time, such as DNA sequencing, automating stock market transactions, or running advanced artificial intelligence algorithms and simulations.

Cloud-based HPC solutions have emerged as a significant trend [109], enabling organizations to access supercomputing power on demand without the need for significant capital investment in physical infrastructure and without knowledge of the underlying architecture. This shift has democratized access to HPC, allowing small and medium-sized enterprises to leverage powerful computing resources for innovation in various fields [186]. However, cloud computing raises a few concerns such as confidentiality, data security and the reliance on third-party providers, which can affect control over sensitive information and compliance with privacy regulations [38].

As the field continues to advance, HPC is expected to play a central role in addressing a large number of the world's most pressing challenges, including climate modeling, drug discovery and real-time language translation. By harnessing the power of cutting-edge hardware and innovative algorithms, HPC stands at the forefront of technological progress, driving breakthroughs across a

wide array of disciplines.

The recent convergence of HPC and AI has opened up an even wider range of applications [59]. HPC's ability to process vast amounts of data and perform complex computations at extraordinary speeds has complemented AI's capacity for learning patterns, making predictions and optimizing systems. The huge increase in computing power has made it possible to train ever larger models, resulting in the development of large language models.

This thesis is firmly rooted in the context of HPC, leveraging its capabilities to address the challenges posed by training and optimizing very large neural networks in massively distributed environments. The computational demands of modern AI models align closely with the objectives of HPC: maximizing efficiency and scalability in processing large and complex datasets. By focusing on preprocessing techniques and distributed algorithms tailored for HPC architectures, this research aims to reduce training costs while keep the model performance or even improve it.

1.6 Accessibility and cost reduction

As generalist AI models offer extraordinary possibilities, it is essential to continue exploring ways of making their development and use more accurate, equitable and sustainable. Ecological impact is directly related to the amount of computation, storage and communications required to train and use an AI model. This poses a considerable challenge, given that the computing power required to train large models tends to be multiplied by 4.2 each year [244]. Modern high-performance system possess immense computing power, but it is often not the primary factor limiting their performance. Instead, performance is frequently constrained by memory access and communication between nodes.

Innovative solutions must, therefore, be implemented to optimize existing algorithms and propose new ones that meet the criteria of precision and computational efficiency. These algorithms and models must present an intrinsic parallelism, making their deployment on parallel and distributed systems easy and efficient. Optimizing efficiency can have a significant impact, especially since models loop a very large number of times. In addition, the development of parallel and distributed computing techniques to fully exploit available resources is crucial to maximize efficiency and minimize energy

costs. By fully exploiting available resources, such as high-performance computing clusters and cloud infrastructures, it is possible to distribute workloads more evenly and reduce overall model costs. This thesis focuses on advancing algorithmic performance to address these challenges ensuring the development of models and methods that are not only computationally efficient but also aligned with the principles of sustainability.

1.7 Research areas and scope

The efficiency of large models depends mainly on the algorithm that describes them, as well as on the quality of the data used to train them. Before deploying these large models on high-performance systems, it is essential to optimize them from a conceptual point of view. This means to design an optimal model delivering accurate and relevant results, all stages of the corresponding algorithms must be optimized, such as reducing dimensionality, considering data sparsity and introducing intrinsic parallelism. Ensuring high-quality data is also crucial for achieving reliable and relevant outcomes. In addition, to effectively deploy these models on high-performance architectures and fully exploit the capabilities of these systems, it is essential to consider efficient parallel and distributed programming paradigms to implement these models.

The scope of this thesis is centered on addressing both of these aspects: model optimization and its effective implementation for high-performance systems. In other words, we aim to optimize training processes through both algorithmic efficiency and practical deployment.

This thesis primarily focuses on the optimization of the training processes of very large neural networks. The focus is on methods and applications at the preprocessing stage of the models, although dimensionality reduction methods discussed in this thesis can be applied to all levels of deep learning models.

A key aspect of our investigation involves designing and optimizing algorithms which take into account constraints that can be both application-specific, such as data sparsity and numerical stability, and high-performance infrastructure-specific such as high-performance programming constraints and potential resource limitations. In addition to the merely adapting existing algorithms, we propose

novel algorithmic strategies tailored for large-scale distributed deep learning environments. These algorithms are designed to optimize the partitioning of workloads and dynamically adjust communication patterns. This ensuring robustness and scalability in high-performance computing environment.

Furthermore, this thesis places particular emphasis on the challenges posed by sparse data structures, such as graphs, which introduce specific computational complexities. Unlike dense datasets, sparse data requires specialized preprocessing techniques to ensure that relevant information is retained while reducing unnecessary overhead. Efficient data representation, transformation and selection are key factors in mitigating the computational burden of training large neural networks in distributed settings.

To reach this objective, we investigate preprocessing strategies that enhance data quality by effectively reducing noise and redundancy while preserving essential structural properties. In particular, we explore feature selection and dimensionality reduction techniques specially designed for sparse data. By optimizing these processes before model training, we aim to enhance the efficiency of deep learning models while minimizing resource consumption.

Another core contribution of this work is proposing advanced data partitioning and workload distribution strategies that facilitate efficient parallel processing. Given the complexity of graph-structured data, traditional partitioning methods may lead to imbalanced workloads and excessive inter-node communication. Our research explores novel partitioning techniques that take into account both computational load and communication overhead, thereby improving the overall efficiency of distributed training.

Additionally, we utilize data preprocessing to reduce the volume of input data without compromising the integrity of the information required for effective learning. This is particularly beneficial in high-performance computing environments, where memory and computational resources must be allocated efficiently. Therefore, we employ dimensionality reduction techniques by innovative numerical methods that effectively compress the data while preserving its essential characteristics. Through careful design and evaluation, we demonstrate that our methods lead to substantial improvements in training efficiency and model performance.

1.8 Structure of the thesis and contributions

The document is organized according to the following structure.

Chapter 2 provides an overview of state-of-the-art techniques for reducing computational complexity in deep learning models. Therefore, we will look at the different methods and approaches used to reduce model size and answer the challenges associated with training these very large models in a parallel and distributed environment.

Chapter 3 presents graph neural networks, where we propose a solution to improve the performance of these networks by extracting topological information from the graph. We introduce an improved dropping technique, incorporating numerical ranking methods to ensure better consistency in graph generation. The next chapter, proposes solutions to address some sparse programming challenges.

Chapter 4 investigates the advantages of data sparsity from both algorithmic and programming perspectives. We examine key operations involving sparse matrices, which are frequently encountered in deep learning models, and propose solutions to optimize their processing in a massively distributed environment.

Extending the discussion on preprocessing dimensionality reduction, the chapter 5 addresses the challenge of high dimensional input data. We propose a general solution for building an embedding of input data that preserves maximum information while significantly reducing the dimension of the model's input data.

After having presented and discussed the proposals we have made to reduce the size of the models, we conclude our investigation and present potential future research directions in chapter 6, including an overview of possible applications of numerical methods for improve algorithms representing the models.

Chapter 2

State of the Art

2.1 Introduction

In the previous chapter, we examined how the exponential growth of deep learning models has led to significant computational challenges in both training and deployment. As model sizes continue to expand, the associated resource demands—ranging from memory consumption to computational power—have escalated dramatically, making scalability and efficiency critical concerns. The main objective of this thesis is to address these challenges by proposing techniques to reduce both the size of large neural network models and their computational cost, thereby improving their feasibility in large-scale applications.

Thus the objective of this chapter is to present an overview of the current state of the art in the algorithmic methodologies of these models. It also provides an overview of parallel and distributed programming paradigms, as well as high performance systems that enable their efficient implementation.

2.2 Fundamentals of matrix computations in deep learning

2.2.1 Introduction to matrix computations

Matrix calculations are fundamental operations in computational sciences. Matrices can be represented by rectangular arrays, which are particularly interesting for representing a set of data in an organized and ordered way in rows and columns. Key matrix operations include addition, multiplication, transposition and inversion. They allow to manipulate these data structures efficiently, enabling transformations, optimizations and solutions to complex mathematical problems.

Matrices play a crucial role in various scientific and engineering disciplines, including physics [195], computer science [259] and machine learning. In linear algebra, they provide a framework for solving systems of linear equations and eigenproblems [96], two fundamental tasks in numerical analysis. In computer pics, matrices facilitate geometric transformations such as rotations, scaling and translations. Moreover, eigenvalues and eigenvectors, key concepts in matrix theory, are widely used in stability analysis, quantum mechanics and for dimensionality reduction. With the evolution of high performance computing, matrix computations have become increasingly efficient, allowing high performance large-scale simulations, high performance deep learning applications, etc.

2.2.2 Role of matrix computations in neural networks

Matrices are used primarily to represent certain data and neural network components. This is generally the case for input data. A batch of input samples is typically stored as a matrix, where each row corresponds to a single data sample and each column represents a feature. Matrices are also used to represent the values of each layer of the neural network. A matrix is used to store the weights that will transform the layer's input values (purple elements in figure 1.1). The layer's biases are also stored in the form of a vector.

Matrix operations are ubiquitous in neural networks. Matrix multiplication is the primary operation in forward and backward propagation, where input matrices are multiplied by the layer weight matrices to progress in the neural network [169]. For a simple feedforward network, given an input vector $\boldsymbol{x}$, the bias vector $\boldsymbol{b}$ and the weight matrix $\boldsymbol{W}$, the pre-activation function output vector $\boldsymbol{y}$ is expressed

as:

$$\boldsymbol{y} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b} \tag{2.1}$$

After that, activation functions is applied to break the linearity [248]. This is an element-wise operation. Gradient calculations also involve matrix multiplication and transposition [5, 14].

Beyond fully connected layers, matrices are essential for more complex neural network structures. In convolutional neural networks, weight matrices are replaced by small filters that convolve over input feature 2D or 3D maps, reducing the number of parameters while preserving spatial relationships [168]. In a similar way, recurrent neural networks rely on matrix multiplications to maintain and update hidden states over time [127]. Transformer models use attention mechanisms based on matrix multiplications to compute relationships between input tokens [276].

2.2.3 Computational challenges in large-scale matrix operations

Notation	***Name***	***Example of Applications***
$\mathcal{O}(1)$	Constant complexity	Access to one element for a matrix $\boldsymbol{A}_{i,j}$
$\mathcal{O}(\log n)$	Logarithmic complexity	Binary search algorithm
$\mathcal{O}(n)$	Linear complexity	Find a specific value in a vector
$\mathcal{O}(n \log n)$	Quasi-linear complexity	Heapsort
$\mathcal{O}(n^2)$	Quadratic complexity	Matrix-vector multiplication
$\mathcal{O}(n^3)$	Cubic complexity	Matrix-matrix multiplication
$\mathcal{O}(2^n)$	Exponential complexity	Knapsack problem with naive approach
$\mathcal{O}(n!)$	Factorial complexity	Find the best solution of Traveling salesman problem with naive approach

Table 2.1: Table of common computation complexities.

Efficient implementation of matrix operations is crucial for scaling neural networks. However, the high complexity of matrix operations poses a challenge for processing very large matrices. As shown in table 2.1, the scalability of processing large-scale matrices is hindered by the quadratic and cubic complexities of these operations. Large-scale models, such as deep convolutional networks and

transformer-based architectures, involve billions of parameters that require extensive matrix multiplications and transformations. These operations pose significant computational difficulties due to their high memory requirements and execution time.

Firstly, there are memory-related problems. Matrix operations in deep learning consume a large amount of memory, particularly in models that process high-dimensional input data or have large-scale layers. Storing weight matrices, intermediate activations and gradients for backpropagation can quickly exceed the available memory on the hardware. Techniques such as mixed-precision training [196] and memory-efficient optimizers [53] help mitigate these challenges, but memory limitations remain a major concern in large-scale training.

Modern deep learning relies on parallel computation using specialized accelerators, which optimize matrix operations through highly parallelized and/or vectorized computations [73]. However, achieving efficient parallelization is non-trivial due to communication overhead, memory bandwidth constraints and load balancing across multiple devices [19]. The limitation of matrix computation can also result from data access issues. Efficient utilization of cache memory plays a crucial role in the performance of large-scale matrix operations. Due to the hierarchical nature of memory in modern computing architectures, frequent data transfers between different levels of cache, RAM and storage introduce latency that can significantly slow down computations [122]. Additionally, accessing large datasets from storage devices or remote locations creates bottlenecks that reduce performance. Optimizations such as tiling [12] or prefetching can be implemented to reduce the risk of blocking and ensure sufficient data flow to load the computation units. However, efforts should be made to minimize communications in order to prevent overloading the network

The computation complexity represents also a major challenge for scaling deep learning applications. The fundamental level 3 BLAS matrix multiplication operation, which is prevalent in both forward and backward passes of neural networks, has a computational complexity of $\mathcal{O}(n^3)$. Although optimized approaches like Strassen's algorithm [261] reduce the complexity to approximately $\mathcal{O}(n^{2.8})$, these methods often introduce trade-offs in numerical stability. In addition, these optimized algorithms generally require more memory to store intermediate results [135], making them unsuitable for use with very large models due to existing memory constraints. Additionally, specialized techniques such

as sparse matrix multiplications help reduce computational demands by leveraging sparsity patterns in weight matrices [201], but these are complex operations to perform.

2.2.4 Optimizations for efficient matrix computations

We have seen that matrix operations are very widespread and represent a significant portion of all computations in many application, in particular in deep learning. Consequently, the hardware architectures of accelerators have adapted to this reality.

Although GPUs (Graphics Processing Units) were initially designed for video games to accelerate computer graphics and rendering [84], in the mid-2000s they began to be used to speed up calculations for training neural network models. Deep learning tasks require the processing of large quantities of data in parallel, and it is this capacity offered by GPUs that has made them a relevant choice for accelerating calculations. The processing speed of matrix operations is much faster on GPUs. This is due to differences in architecture compared to a CPU. As can be seen in figure 2.1, the GPU is made up of hundreds or even thousands of small independent computation cores with very small shared caches, unlike CPUs, which have a small number of complex cores and very large caches. GPUs excel at tasks involving parallel calculations and simple independent arithmetic operations. As we detailed in 2.2.2, this is the type of computation we find when training a neural network. We will discuss the optimizations at the hardware accelerators level in more detail in section 2.4.4.

Efficient matrix computations in deep learning depend not only on hardware but also on highly optimized software libraries and frameworks. General-public deep learning frameworks (which we will look at in more detail in section 2.4.5) are based on libraries specialized in the implementation of linear algebra operations. These high performance libraries ensure efficient implementation for matrix operations.

BLAS (Basic Linear Algebra Subprograms) are a collection of standardized routines used to perform basic operations in linear algebra, such as vector and matrix operations [164]. They are used as fundamental building blocks for more complex algorithms and methods. BLAS are essential for HPC applications, as they enable elementary operations to be defined and optimized to make the most of the hardware's computing power.

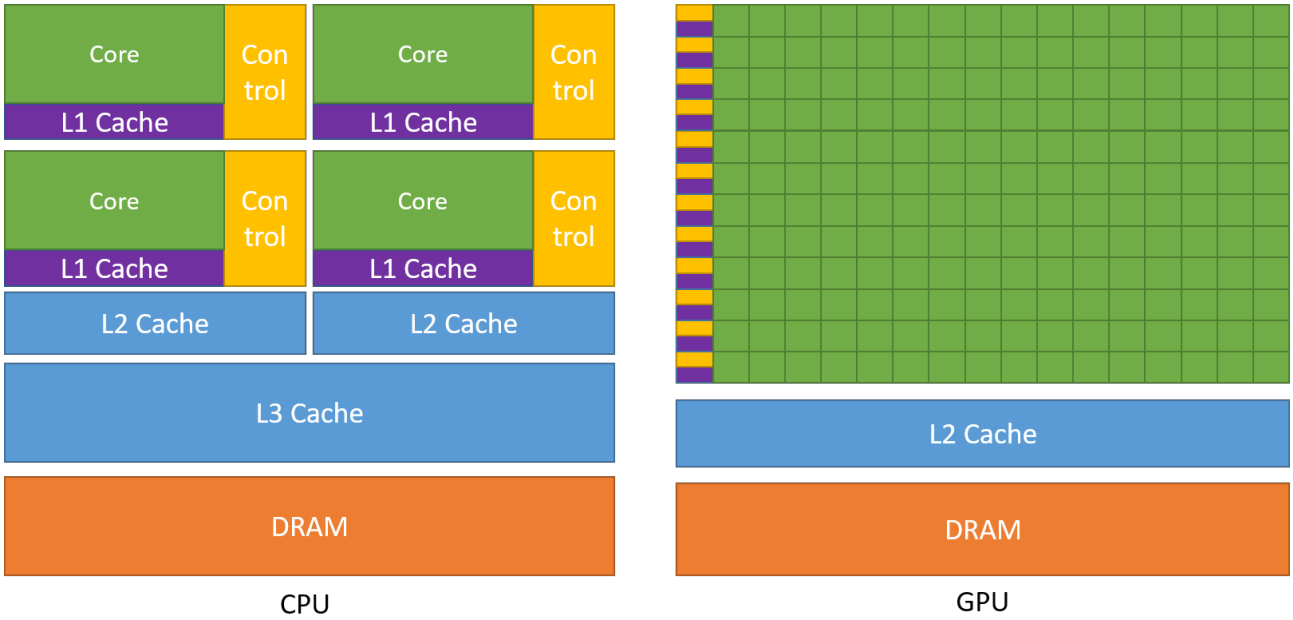


Figure 2.1: CPU and GPU architecture comparison [106].

The BLAS routines are classified into three levels, the first covering all vectorial operations. The routines here only manipulate scalars or vectors. This level includes for example the dot product or the multiplication of a vector by a scalar. Level 2 [56] covers all operations that manipulate one and only one matrix. The most popular Level 2 operation is matrix-vector multiplication. There are also used for solving a system of linear triangular equations. Finally, BLAS Level 3 covers all matrix-matrix operations [57]. Level 3 contains the multiplication operation between two general matrices [44].

Built on BLAS, LAPACK [7] (Linear algebra package) provides more advanced matrix computations such as solving linear equations, eigenvalue decompositions and singular value decomposition. LAPACK's algorithms have been extensively optimized for performance on a variety of hardware architectures, making it a reliable and indispensable tool in the field of numerical linear algebra.

Hardware manufacturers are developing libraries to perform these matrix operations on their accelerators efficiently. For example, NVIDIA proposes its own BLAS implementation for its GPU accelerators, called cuBLAS[1]. The cuBLAS library can be called by programmers as an API for performing matrix operations on GPUs. In this way, the user ensures that the implementation of the matrix operation will be specially optimized for this type of hardware architecture.

[1]https://docs.nvidia.com/cuda/cublas/index.html

2.2.5 Reduction of model size

Optimizing matrix computations is a very interesting way of increasing computational efficiency and reducing the execution time required to perform operations. However, with very large models, this is not enough to ensure that training can be processed within an acceptable time. In this case, reducing the size of the models is the main lever available to reduce the computational power needed to train these models. Indeed, the number of parameters in the model has a proportional impact on the required computational power [43]. It has been shown that implementing methods to train a model with fewer parameters more efficiently has a significantly greater impact on model accuracy compared to merely increasing the number of parameters [15].

The first solution to reducing model size is to modify the model architecture. By removing a layer or reducing the number of perceptrons in that layer, we reduce the number of weights to be trained in the model, which has an immediate impact on the computational power required for training. However, finding the right compromise is not an easy task, both in terms of minimizing size and ensuring that the model is large enough to be trained and generalized.

A second approach to reduce the model size involves modifying the size of the input data. Efficient data reduction enables a significant decrease in neural network size without compromising accuracy [100]. One widely used strategy is dimensionality reduction, which refines input patterns by identifying and preserving only the most relevant features while filtering out redundant or less informative data [268]. By capturing correlations between input variables, dimensionality reduction not only reduces complexity but also enhances interpretability. Reducing the dimensionality of data can help to understand hidden structures and pattern [205].

Beyond improving data representation, dimensionality reduction techniques also contribute to practical advantages in data processing. High-dimensional data can be computationally expensive and challenging to work with, often leading to increased processing time and resource consumption. By reducing dimensionality, these methods minimize the effort required to extract valuable insights, making data analysis more scalable and efficient [9]. This scalability is especially important for handling large datasets where excessive data complexity can prevent performance [68].

Furthermore, dimensionality reduction plays a vital role in improving predictive accuracy. Re-

moving the noise in data or irrelevant features, dimensionality reduction techniques improve data prediction and analysis [227]. Additionally, by simplifying the input space, these techniques enable models to process information more effectively, reducing computational burden while maintaining or even enhancing accuracy. Implementing such methods ensures that the DL models work efficiently, making predictions with greater clarity and reliability.

Another complementary approach to reducing model size is network sparsification. The network sparsification involves pruning less important connections within a neural network while preserving its predictive capabilities [129]. By identifying and eliminating redundant or low-impact parameters, sparsification reduces memory and computational requirements, making models more lightweight and efficient. These techniques help maintain model performance while significantly decreasing the number of active parameters, thereby reducing the model complexity and its size [112]. For sparsification solutions to significantly enhance both model accuracy and size reduction, they must be implemented efficiently [20]. We will explore in more detail the different methods for reducing model sizes in the following section.

2.3 Algorithmic methodologies for reducing model size

As neural network models continue to grow in complexity and size, their computational demands increase significantly, making them challenging to deploy on resource-constrained devices. To address this issue, various algorithmic methodologies have been developed to reduce model size while maintaining performance. This section explores key techniques such as dimensionality reduction and model compression techniques. By optimizing model architecture and improving efficiency, these approaches enable scalable and cost-effective deployment of machine learning models across diverse applications.

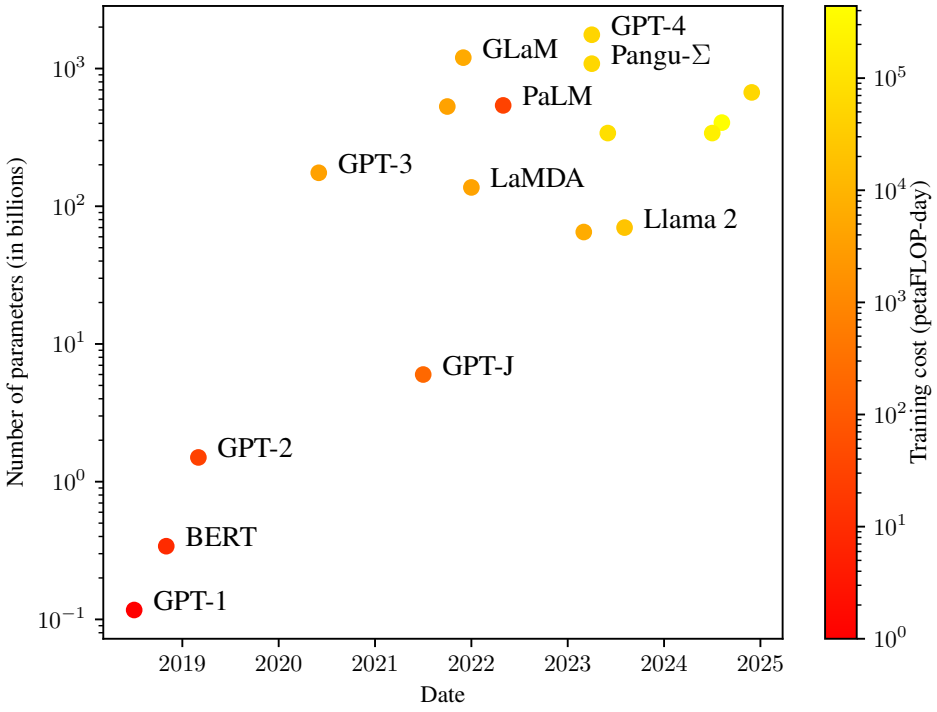


Figure 2.2: Evolution of model parameters and the computational power required for training over time.

2.3.1 Context model size reduction

Exponential increase of model sizes

The link between model complexity and model size is significant. It is easy enough to understand the relationship between these two notions. Model size is often measured by the number of parameters (weights and biases). The more parameters a model contains, the more computing power it will require to run through it. More computing power will also be needed for the training phase, as all these additional parameters will have to be refined at each epoch, adding even more complexity.

However, the success of neural networks has diversified their applications. The complexity of these new tasks has necessitated the design and use of more complex models. The trend has been to increase the size of new models. The search for models to improve performance on existing tasks has also pushed up the number of model parameters. For example, the first simple multi-layers perceptron models in the early 2010s achieved an accuracy of around 97% for the classification of handwritten numbers (MNIST dataset), with a trainable parameter count close to 100k. Today,

the most powerful model achieves an accuracy of 99.87%, but the number of parameters in this model exceeds 1.5 millions. [32]. There are many advantages to using larger models. Large models, particularly in the realm of machine learning and artificial intelligence, have demonstrated the ability to learn sophisticated features from vast and complex datasets [165]. This capability is crucial for various applications such as natural language processing (NLP) [31]. Scaling up language models can lead to emergent abilities that are unattainable with smaller models. These abilities appear in larger models, enhancing the range of the model's capabilities [282]. Staying in the realm of natural language processing, the GPT-4 model, with a parameter count of around 2 trillion, can be used for a wide range of tasks, including text writing, translation, question answering and content personalization. The versatility offered by these large models means that we can use the same model for very different tasks without major modifications.

Large models are also more robust when dealing with noise and variations in input data [299]. This makes them more attractive for real-world applications where data can be unpredictable and noisy.

All the factors mentioned so far have encouraged the development of larger and larger models. Although large models existed before, with AlexNet in 2012 and VGG in 2014, it was around 2018 that the size of deep learning models began to follow an exponential trajectory. The BERT model proposed by Google in 2018 is made up of 340 million parameters, double the number of VGG parameters introduced 4 years earlier. The following year, OpenAI presented a large-scale language model with 1.5 billion parameters, multiplying the number of parameters by 4 in just 1 year. As can be seen in figure 2.2, this exponential trajectory has continued until today, when the most advanced deep learning models exceed a trillion parameters. These include OpenAI's GPT-4, Google's Switch Transformers or Huawei's Pangu-Σ. This has enabled the emergence of models that are trained with very large scales of data, enabling them to generalize and serve as a common base and be adapted to a wide range of downstreams tasks. This type of model is defined in [26] by the term foundation model.

A stagnation of DL models size

Despite all the solutions that have been put in place to meet the growing demand for computational power, the computational power estimates required to build much more powerful models are far too

high to continue to be sustained by increasing parallelism solutions alone. The cost of training models is growing exponentially. The environmental and financial costs of supporting training are more than a factor too great to contemplate [271]. Each additional dollar invested in a model increases performance less and less [185]. For example, increasing the budget from 10 million to 100 million will improve the model's accuracy rate by 10 percentage points, from 65% to 75%. However, multiplying the training budget by a further 10 times to 1 billion will only train a model with an accuracy of 80% [185].

There is a high probability that we will soon see the end of this escalation in size and training costs for the very large foundation models [265]. This is why solutions and techniques that aim to reduce model size while maintaining good performance are very interesting for absorbing part of this problem.

2.3.2 Model compression techniques

A large number of solutions have been proposed to reduce the size of models and the number of parameters. This section describes the main techniques used.

Quantization

Quantization is a process of reducing the accuracy of model weights by replacing original values with approximations using fewer bits [141] while maintaining the model's accuracy. This both reduces the memory space required to store parameters and speeds up calculations thanks to mixed-precision calculations [54] that are efficient on modern accelerators [166]. This solution is mainly used when deploying large language models (LLMs), as they are costly to deploy and maintain [288]. Quantization is limited to the inference part, since model training requires precision that cannot be achieved with a smaller number of bits. Quantization performance is subject to strong perturbations due to outliers [283].

Knowledge distillation

Knowledge distillation is a technique used to reproduce the prediction of a very large model by a smaller model, called a student model [126]. The small model will learn to mimic the behavior of

the very large model. It is this student model that will be used for inference. This student model is generally less efficient than the model it imitates, but it is much faster. This technique is useful for large models that are too cumbersome to deploy, or for saving time in real-time applications where low latency is required [99]. However, this technique has its limits, since the performance of the student model can be altered by a variety of factors, such as the architecture of the student network model, the dataset scale and the data domain [256]. All this can produce a significant gap in fidelity between teacher and student model. Also, this technique is only intended to improve performance during inference, not during training.

Low-rank factorization and optimization

Low-rank approximation is a mathematical technique aimed at approximating a high-dimensional data matrix with a matrix of lower rank [153]. To realize this operation, we build at least two matrices of small size, so that multiplication of these matrices gives a correct approximation of the original matrix. This makes it possible to compress parameter matrices [238] and deploy LLMs on machines with limited memory capacity. The low rank approximation of matrices also plays a very important role in tensor decompositions [69].

The reference method for decomposition is the singular value decomposition (SVD) [258]. This method gives the best approximation of a matrix [153]. The Eckart-Young theorem [62] demonstrates that the k-rank approximation of the matrix with SVD is the closest, both with the Frobenius norm and the spectral norm. SVD decomposes the matrix into three distinct matrices, at least two of which are square, unitary and orthogonal. SVD is applicable to all types of matrices, not just square matrices. SVD is an application to arbitrary matrices of the spectral theorem, which finds an orthonormal basis of eigenvectors for diagonalizing a matrix.

Pruning

Pruning (or sparsification [242]) is the principle of removing a certain percentage of nodes in the neural network. Pruning can be used during the training phase, but mainly to induce sparsity, improve model generalization [91, 20] or prevent overfitting [16]. The pruning technique, designed to reduce

model size, is generally applied after training. We apply pruning to reduce the size of the model while maintaining the same accuracy. By removing non-essential weights, we speed up model inference and reduce memory consumption. But pruning requires a retraining cost that is not attractive when dealing with very large neural networks. Modern solutions for applying pruning to very large models have been proposed [264, 131]. The methods allow networks to be sparsified after training and are compatible with a reduction in the size of the input dimension.

2.3.3 Dimensionality reduction strategies and embedding

Model input data can be very high-dimensional. However, a large proportion of the dimensions may be redundant or not particularly informative, which will not be useful for the prediction or will complicate the analysis and processing of the data. One solution is to get rid of this uninteresting data, and give the input model only data that is of interest for its prediction. This is called dimension reduction. The aim is to simplify the data by reducing the dimension in which it is represented, while preserving as much essential information as possible. This reduces the need for computation and, if done correctly, can improve model performance [278].

Principal component analysis (PCA) [143, 102] is one of the most widely used linear techniques for dimension reduction. It is the reference method for projecting large data into a low-dimensional space [207]. It works by maximizing the variance of the data in the lower-dimensional representation, which often reveals the underlying structure of the data. It is widely used to propose clustering in unsupervised tasks [55]. PCA is also used in others application as features extraction [178], face recognition [291] or fault-detection [88].

There are other strategies for dimensionality reduction. The independent component analysis (ICA) [46] is a technique similar to PCA but focuses on finding statistically independent components in the data, rather than the components that maximize variance [138]. ICA is particularly useful for applications where the goal is to separate mixed signals, such as in the case of blind source separation, where the objective is to recover individual source signals from a mixture of signals [260]. It is commonly used in signal processing, neuroscience and finance.

Linear discriminant analysis (LDA) [78] is another linear technique for dimensionality reduction,

but unlike PCA, it is supervised. LDA aims to find a lower-dimensional space that maximizes the separation between multiple classes of data. It works by maximizing the between-class variance while minimizing the within-class variance, making it especially useful for classification tasks. While LDA is effective for supervised classification problems, its applicability may be limited when dealing with nonlinear data or when the assumptions of normality and homogeneity of variance are not met [76]. High-dimensional classification processing also remains a major problem [293].

Dimensionality reduction strategies are essential tools for managing and interpreting complex datasets. The choice of technique depends on the nature of the data, the problem at hand and the goals of the analysis. Some methods such as t-Distributed Stochastic Neighbor Embedding (t-SNE) [274] will be specially designed for data visualization, while other methods such as Isomap [270] will be used to capture the underlying manifold structure of the data, making them more suitable for datasets with nonlinear relationships.

Even if embedding is not technically a dimension-reduction solution, its role in representing high-dimensional data regularly leads it to modify data dimensions. Embedding is the process of representing complex objects such as words, images, categories or graphs in a low-dimensional vector space [290]. The aim is to build a vector that best represents the initial data. The role of embedding is to build a representation that preserves semantic or structural relationships between objects while reducing their complexity. More concretely, we need to find a good embedding that will project initial objects that are close to each other in relatively the same place. If we take word embedding as an example, a good embedding will produce vectors which are close in term of distance between two words with similar meanings or used in the same context (e.g. the words *dog* and *cat*, are relatively close and are often used in similar contexts since they are both pets, so we expect the vector representations of these two words to be relatively close) and will distance words that have distant meanings (the vectors representing the words *dog* and *cat* will have to be very different from the word *plane*).

Dimension reduction and indirectly embedding both reducing the apparent complexity of the data, while preserving the underlying information and relationships. Both are particularly useful in contexts where the initial data is very large. It is the type of input data that differs. For dimension reduction,

the input data is already numerical, whereas embedding focuses on non-numerical or discrete data, such as categorical data encoded in the one-hot [115] format, or unstructured data, and we will look at the different data structures in more detail in the next section.

2.3.4 Challenges in model size reduction

Reducing the size of DL models often leads to a trade-off between model accuracy and efficiency [42]. We have seen above that techniques such as quantization, pruning and knowledge distillation can reduce the number of parameters and computations, but they may also degrade the model's predictive performance if not carefully implemented. Maintaining a balance between compression and accuracy remains a major challenge.

Moreover, many model size reduction techniques require significant computational resources. For example, many pruning solutions have outperformed random pruning [82, 118, 298]. However, in order to obtain results that are efficient, these techniques necessitate extensive fine-tuning and iterative processes. This significantly increases the computational power required and can have a major impact on training time. There is a real challenge in finding a balance between the gain that such techniques will bring and the additional computational cost they represent.

While there has been progress in automated machine learning [136] and neural architecture search [304], automating the compression process while preserving performance remains difficult. Many current approaches still require manual tuning and domain-specific expertise [292]. More research is needed to develop adaptive, generalist and automated compression frameworks that work across different applications.

2.4 Computational challenges in large-scale neural networks

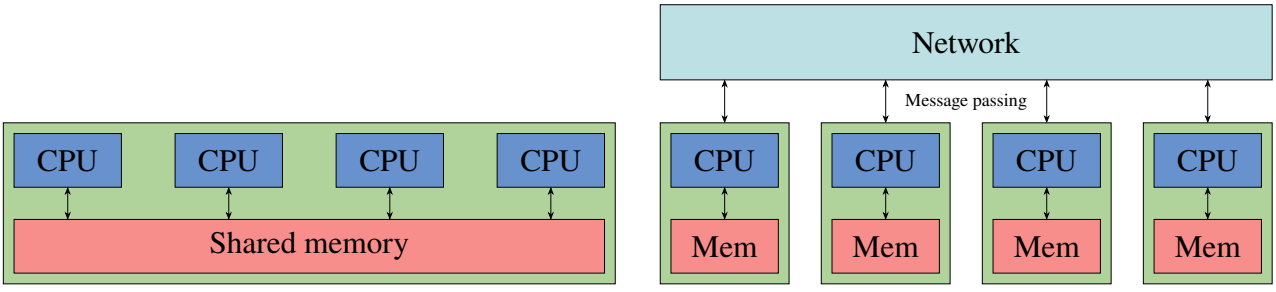
As neural networks grow in scale, both in terms of model parameters and dataset size, they present significant computational challenges. Training and deploying these models require vast amounts of memory, processing power and efficient optimization techniques. Issues such as hardware limitations, distributed training complexities and energy consumption become critical concerns. This section

explores the key computational bottlenecks in large-scale neural networks and discusses strategies to mitigate these challenges, enabling efficient training and deployment.

2.4.1 Overview of distributed and parallel computing paradigms

Distributed and parallel computing paradigms are fundamental to modern computational systems, enabling efficient processing of large-scale tasks by leveraging multiple computing resources. These paradigms address challenges related to performance, scalability and fault tolerance in various applications. This section provides an overview of the key paradigms in distributed and parallel computing, highlighting their characteristics, advantages and use cases.

Parallel computing is a type of computation in which many calculations or processes are carried out simultaneously. Large problems are divided into smaller ones, which are then solved concurrently, often utilizing multiple processors or computers working together. There are two main types of parallelism model: shared memory model and distributed memory model. A representation of these two architectures is shown in figure 2.3. In the shared memory model, multiple processors share a common memory space and communicate through read/write operations. In the distributed memory model, each processor has its own local memory and communication occurs through message passing. There are also hybrid models which combines both shared and distributed memory approaches with multi-level architecture [83].



(a) Shared memory architecture (b) Distributed memory architecture

Figure 2.3: Parallel and distributed memory architecture.

There are many different types of computer architecture. In order to better understand the different ways parallelism can be achieved in computer systems, the Flynn taxonomy [79] classifies architectures into four categories: Single Instruction, Single Data (SISD); Single Instruction, Multiple Data (SIMD);

Multiple Instruction, Single Data (MISD); and Multiple Instruction, Multiple Data (MIMD). An example of SISD is a traditional sequential computer that processes a single instruction and single data at a time. A SIMD is a system where a single instruction operates on multiple data points simultaneously [198]. It is this architecture that is commonly used in modern accelerators. MISD is a rare architecture where multiple instructions operate on a single data stream, often used in fault-tolerant systems [48, 286]. Finally, MIMD is a system where multiple processors execute different instructions on different data streams, which is the basis for modern multiprocessor and distributed systems [11].

The term *distributed computing* refers to a system in which multiple autonomous computing nodes communicate over a network to achieve a common computational goal. Unlike parallel computing, distributed computing focuses on coordinating loosely coupled nodes, often with high fault tolerance and scalability.

The main characteristics of a distributed system are the ability to easily add new computing nodes to ensure scalability, a network connecting different nodes to communicate, share data and synchronize. Additionally, there is generally decentralization with no specific node controlling the entire system.

Communications are slow overall. Even if performance improvements on networks are very important, communications remain a time-consuming issue. Numerous low-latency network architectures connect today's largest supercomputers, such as Fat Trees [172, 220] or the hypercube networks [209]. This requires limiting communications between nodes. With all these new factors, it is complex to compare different distributed implementations of an algorithm to determine which is the most efficient. Especially as the performance of each implementation also depends on the characteristics of the machine on which it will be deployed. Consider two implementations A and B to solve a problem. Let's assume that the A implementation requires more communication than the B implementation, but in exchange requires fewer calculations. It will probably be more interesting to deploy the A implementation on a machine where the computation nodes are connected by a high-performance network, but it will be unsuitable for machines with a mediocre network. On such machines, it will be preferable to use the B implementation.

These new computer architectures have necessitated a rethinking of programming models to in-

corporate the new costs associated with them. The Bulk-Synchronous Parallel model [273] (BSP) is a programming model specially designed to represent parallel and distributed computing environments. It is a model that takes into account the communication and synchronization costs of computation nodes, unlike popular PRAM models [81]. A wide range of actual distributed architectures can be seen as BSP computers [177]. It consists of a set of processor-memory pairs that are considered homogeneous, i.e. all processors have the same computing capacity. These processors are interconnected with a network that enables them to communicate with other processors. It is also assumed that there is a global synchronization unit to perform a synchronization barrier.

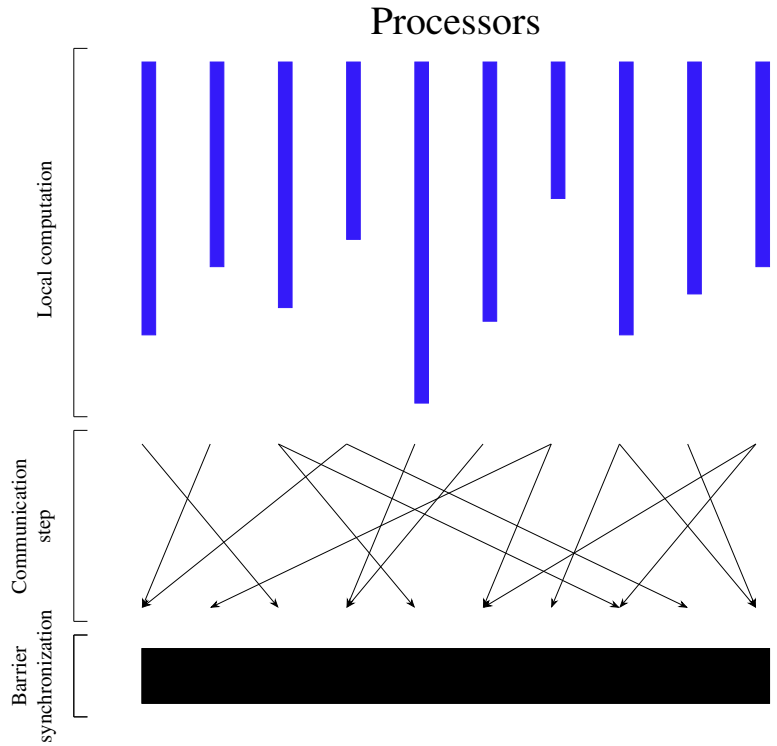


Figure 2.4: Illustration of a Bulk-Synchronous Parallel superstep.

A BSP program is structured as a succession of supersteps. A superstep is illustrated in figure 2.4. These three main phases form the basis for calculating the algorithmic cost of an s superstep. The cost of the first phase is the cost of the largest workload among all the processors. The cost of the second phase is determined by the maximum quantity that can be send and receive from the processors. Finally, the cost of the last phase is the cost of global synchronization. Noting w_i the local workload of processors i and h_i the amount of communication associated with this same node, the algorithmic

cost of a superstep s is defined by:

$$\text{Cost}(s) = \max_{0 \leq i \leq p}(w_i^s) + \max_{0 \leq i \leq p}(h_i^s) \times g + L$$

with p, g et L the BSP machine parameters fixed according to the machine's technical specifications.

The total cost of the BSP program is the sum of the costs of each superstep Cost(s). Let S be the number of supersteps in the program, the total algorithmic cost of the program will be:

$$Cost(S) = \sum_{s=1}^{S} \text{Cost}(s) \tag{2.2}$$

and

$$Cost(S) = W + H \times g + S \times L \tag{2.3}$$

with $W = \sum_{s=1}^{S} \max_{0 \leq i \leq p}(w_i^s)$ and $H = \sum_{s=1}^{S} \max_{0 \leq i \leq p}(h_i^s)$. BSP provides a clear and visual understanding of the different costs that are specific to parallel and distributed architectures. It allows to compare different algorithms for a given machine, thanks to an *a priori* analysis.

2.4.2 Scalability issues in a massively distributed environment

Scalability is a critical concern in massively distributed environments, where systems must efficiently manage increasing workloads, data volumes and network complexities. There are a number of challenges to take into account to ensure scalability in a massively distributed environment.

The first limitation to take into account is the decrease in efficiency as the number of compute nodes increases. In the majority of computer programs, part of the calculations cannot be split up to be parallelized on several calculation nodes. Amdahl's law [6] is a formula that describes the potential speedup in performance of a computational process when using multiple processors. By taking into account that a percentage of calculations must be performed sequentially, Amdahl's law highlights the diminishing returns of adding more processors to a system when there are parts of the computation that cannot be parallelized. This also highlights the importance of optimizing not just parallelizable

tasks but also minimizing sequential dependencies [70].

As distributed systems grow, communication overhead increases due to the important number of interconnected nodes [180]. High network latency and bandwidth constraints can lead to bottlenecks, affecting response times and overall system performance [28]. Increasing the number of nodes also increases the quantity of communications that must circulate over the network [241]. Network congestion must be taken into account during the design phase to prevent it from becoming a bottleneck that would hamper performance [224].

An increase of the number of compute nodes in distributed systems also increases the chance of node failures [250]. Ensuring high availability and fault tolerance requires robust recovery mechanisms and redundancy [156].

Ensuring an even distribution of workload across multiple nodes is essential for scalability. Improper load balancing can lead to resource underutilization in some nodes while others become overloaded [159]. Dynamic load balancing algorithms [4, 121, 162] or adaptive resource allocation [97, 189] are commonly used to address these challenges.

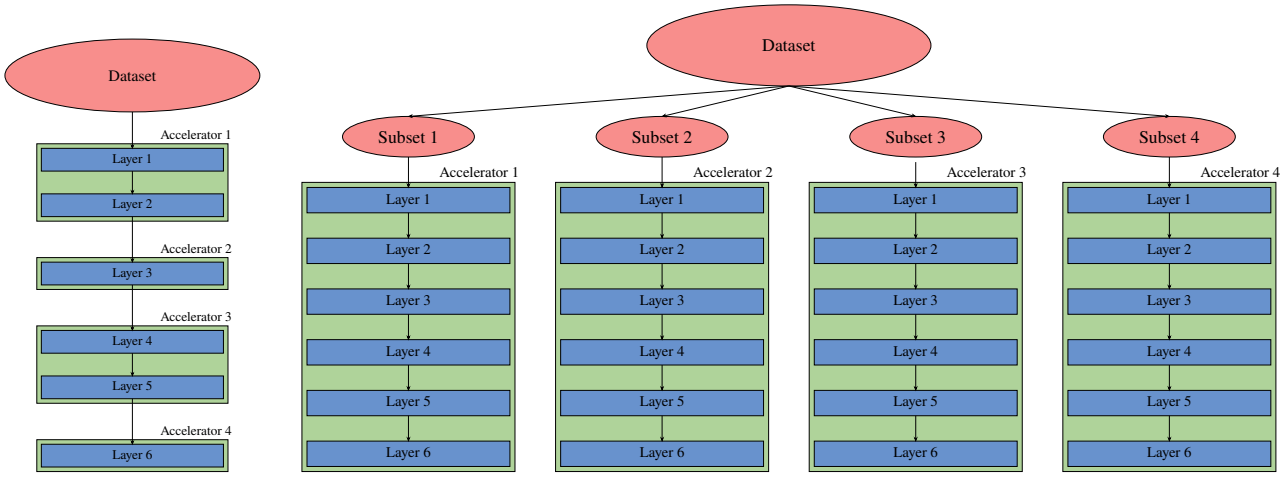
2.4.3 Parallel training: data parallelism and model parallelism

To use these parallel and distributed environments to train deep learning models, it was necessary to define parallelism strategies before correctly distributing the workload over the different compute nodes. The two dominant method are data parallelism [125] and model parallelism [51]. A schematic representation of these two approaches is available in figure 2.5.

Data parallelism approach involves replicating the entire model across multiple devices while dividing the dataset into smaller batches [158]. Each device processes a different batch of data and computes gradients independently, which are then aggregated to update the model parameters synchronously or asynchronously. Data parallelism is effective when the model fits within a single device's memory but requires faster computation.

The second partitioning strategy for deep learning model training is model parallelism. When a model is too large to fit within a single device's memory, it is split across multiple devices [41]. Different layers or components of the model reside on separate computation units and data flows

sequentially or in parallel between them. Model parallelism is crucial for training extremely large models, but comes with increased communication overhead [163].



(a) Model parallelism (b) Data parallelism

Figure 2.5: Model parallelism and data parallelism comparison with 4 computation nodes.

Instead of dividing data or model components, task parallelism assigns different operations or tasks to separate devices [262]. This strategy is useful when different tasks have varying computational requirements and can be executed independently. Task parallelism is often combined with data or model parallelism in complex training pipelines to optimize performance further [71]. However, this requires a good definition of the dependencies between tasks to ensure that this type of parallelism works efficiently. For this, it is recommended to represent the model execution as a directed graph of tasks, where edges denote dependencies between different tasks.

By leveraging these parallelization techniques, deep learning models can be trained more efficiently, enabling faster experimentation and scaling to larger architectures. Depending on the model size, hardware availability and computational needs, a combination of these methods is often the most effective approach.

2.4.4 AI dedicated hardware architectures

The exponential increase of model sizes was possible because there was enough computing power to support the cost. HPC has evolved over the last few years to support the ever-increasing needs of deep learning models.

As we've already discussed in section 2.2.4, GPUs were initially used to accelerate the matrix calculations of deep learning models. Since then, many companies have begun offering accelerators specifically designed for Deep Learning tasks [229]. Nvidia was one of the first to introduce an accelerator designed for deep learning and scientific computing workloads, with the launch of the Tesla [33] series of cards. Google then entered the market with TPUs in 2015 [144], Huawei with its Ascend [181] chips and Intel with its Gaudi [145] chips. Today, accelerators have become much more specific, to meet different needs depending on the task in question. For example, Huawei's Ascend 910 accelerators are designed for model training. This is a time-consuming and intensive processes, requiring enormous computing power and the ability to process very large quantities of data. Ascend 910s are therefore designed to perform massively parallel matrix operations and are linked by high-capacity buses to enable fast, massive communication [228]. The Ascend 310 accelerators, on the other hand, are designed for inference. This is a process which is less intensive in terms of computational power, but often has to be carried out in near-real time. These processors are therefore optimized for low latency and high efficiency. Finally, there are accelerators like the Kirin 990 which are also designed for inference, but on mobile devices. They are capable of processing small models locally, to process photos for example. The focus on this type of accelerator is on energy efficiency and relatively low manufacturing costs, as they are aimed at the general public.

2.4.5 Frameworks for distributed DL

Although the power of accelerators has increased significantly, other factors have helped to meet the need for computations for deep learning tasks. AI frameworks have evolved significantly, providing more efficient algorithms and better optimization techniques. Among the most well-known frameworks, we can mention Tensorflow [93], Caffe [142], PyTorch [213] and MindSpore [133]. The solutions proposed by accelerator manufacturers make the most of the hardware's computational power. In addition, the integration of these frameworks with advanced software libraries such as CUDA [89] has facilitated the implementation and optimization of Deep Learning models, making training and deployment processes faster and more efficient. Today's AI frameworks are both easy to learn and to use, with high-level method calls, and highly optimized. Consequently, the frameworks

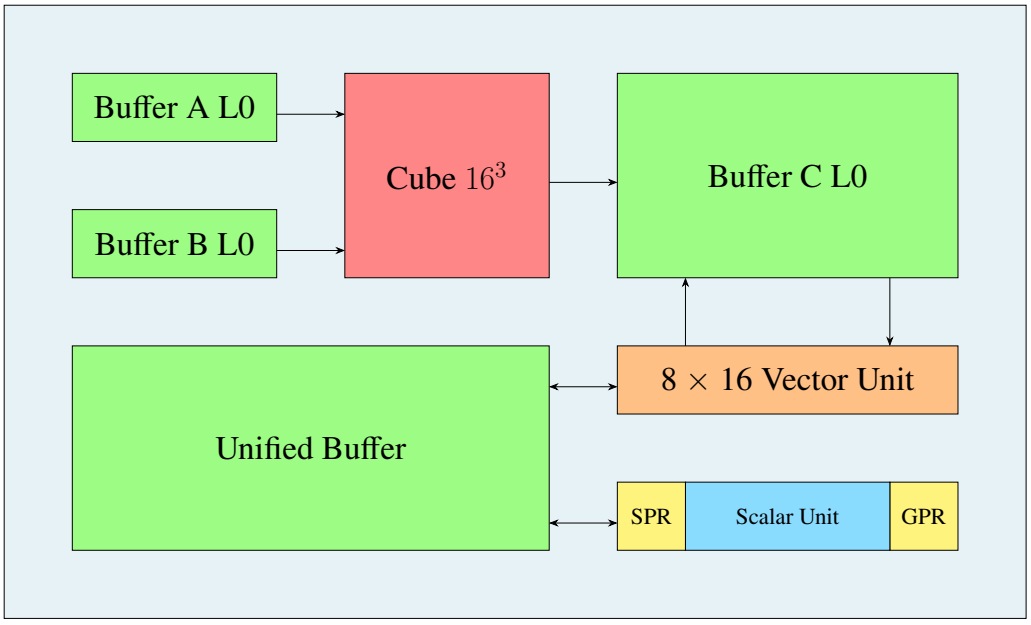


Figure 2.6: Simplified Davinci core architecture [139]. This is the chip architecture used in Ascend boards. We can see a cube used to multiply two matrix blocks of size 16×16 stored in buffers A and B respectively. The result of the multiplication is 4096 arithmetic operations performed in a single clock cycle. The cube massively boosts the performance of matrix operations, which account for a very large proportion of Deep Learning operations.

are written in Python, providing a simple, flexible interface for the user [226]. These methods call low-level methods written in C or C++, which are more suitable since they are several dozen times more efficient. This integration is transparent to the user, offering powerful and efficient solutions for the development of deep learning models without the constraints of less accessible low-level languages.

Additionally, the growing ecosystem of open-source AI frameworks has fostered collaboration and innovation within the community. Researchers can easily share their work, reproduce experiments and build upon existing models, accelerating the pace of advancements in the field. These frameworks often come with extensive documentation, tutorials and community support, making it easier for newcomers to get started and for experienced practitioners to deepen their expertise.

Sparse linear algebra can thus be efficiently processed by known scientific tools like PETSc [13] and Trilinos [123]. These frameworks are very powerful tools and provide solutions for sparse linear algebra in a distributed environment. However, the storage format of these libraries is very rigid and quite heavy for re-implementation, and very expensive to adapt to a DL framework. Solutions like cuSPARSE [203] can take advantage of hardware architecture and allow to perform sparse matrix

operations on accelerators in a very efficient way.

The main AI Frameworks such as TensorFlow GNN[2] and PyTorch Geometric[3] rely on libraries supplied by accelerator manufacturers to perform sparse matrix operations on this hardware. The most popular is NVIDIA's cuSPARSE [203] library. This library is highly optimized for performing matrix operations with sparse matrices on GPUs. It provides a suite of basic linear algebra subroutines specifically designed for sparse matrices.AI frameworks use the functionalities offered by this library through an API. This ensures that operations are correctly optimized for GPUs, but limits the scope for frameworks to implement more operations or support more formats. They are limited to storage formats that are implemented and supported by cuSPARSE. It has been shown that the ELLPACK format is the most efficient for performing matrix operations on GPU [134] when the number of non-zero elements per row is sensible and evenly distributed. However, none of the AI frameworks supported this sparse matrix storage format, as it was not implemented in cuSPARSE. Furthermore, hardware manufacturers are not necessarily developing methods to make the library suitable for distributed use with multiple accelerators. Managing data distribution between accelerators and keeping the advantages is still a research topic.

As for other specific AI accelerators, they generally deliver optimized solutions for dealing with sparse matrices and operations. They come with specific frameworks designed to make the most of hardware accelerators. However, these accelerators are primarily designed for highly efficient, dense matrix multiplication. Sparse matrix storage formats do not offer the data parallelism required to take full advantage of these accelerators, which limits performance. The irregular memory access patterns are the main cause of the performance limitation [10]. Solutions and specific storage formats have been proposed to adapt to the specific characteristics of Multiply-and-Accumulate accelerators, such as Google's TPU [117, 170]. Generally speaking, the accessibility and visibility of methods deployed within these frameworks is limited, and manufacturers restrict the dissemination of implemented solutions and optimizations.

[2]https://github.com/tensorflow/gnn
[3]https://github.com/pyg-team/pytorch_geometric

2.5 Conclusion

The field of parallel and distributed computing for deep learning is becoming increasingly important due to the ever-growing demands of very large neural networks. Despite the significant advancements that have been mentioned, major challenges remain, such as reducing model size and computational costs.

Matrix computation plays a fundamental role in deep learning, serving as the backbone for efficient neural network operations. As deep learning models continue to scale in size and complexity, innovative algorithmic methodologies are essential for reducing model size without compromising performance. Techniques such as matrix factorization, pruning, quantization and dimensionality reduction have demonstrated significant potential in improving computational efficiency while maintaining accuracy. Dimensionality reduction methods such as PCA help to eliminate redundant features, reducing memory usage and speeding up training and inference. However, large-scale neural networks still pose considerable computational challenges, including memory constraints, high energy consumption and latency issues, necessitating further advancements in optimization techniques and hardware acceleration. Future research should focus on developing more adaptive and hardware-aware solutions to enhance scalability and efficiency in deep learning, ensuring that computational advancements match the growing demands of AI applications.

Building on this state of the art, this thesis proposes new methods for optimizing the training of very large models in distributed environments. By focusing on algorithmic efficiency, sparse data optimization and effective parallelization techniques, this thesis aims to contribute to addressing some of the cited challenges regarding scalability, algorithm optimization and computational efficiency. The following chapters will build on these foundations to introduce new approaches for improving model accuracy, optimizing the use of distributed resources and reducing model complexity.

Chapter 3

Innovative Dropping Strategies for Improved Graph Neural Network Accuracy

3.1 Introduction

Graph applications cover a very wide range of domains, from social networks that map out connections between friends, to recommendation systems that suggest products based on user preferences. Graphs are also used in logistics to optimize delivery routes, in biology to understand protein interactions, and in cybersecurity to detect and prevent threats. The versatility and power of graph applications make them an essential tool in various industries, driving innovation and efficiency. Graphs are one of the most common forms of non-Euclidean data. The term non-Euclidean refers to the fact that data cannot be represented in a Euclidean space. In Euclidean space, the shortest path between two points is necessarily a straight line. However, the distance between two nodes in a graph is not necessarily equal to the distance between their coordinates in an Euclidean space [149]. A graph is defined as a triple composed of a set of vertices (also called nodes), a set of edges and a relation table that associates two vertices with each edge [285]. If the edge is directed from x to y, x the tail of the edge and y the head of the edge, x and y together are named the endpoints of the edge. In this document, all the graphs we will be manipulating or discussing are finite. A graph is finite if its vertex set and edge set are finite.

Indeed, the graph is a fairly instinctive way to represent relationships between different entities (figure 3.1). However, it remains a different data structure that needs to be handled accordingly. As graphs are non-Euclidean data, they cannot be represented in Euclidean space without a loss of information. Specific algorithms and methods must therefore be found to handle this type of data. This is also the case with deep learning models, where we find and use models specially designed to handle this type of data.

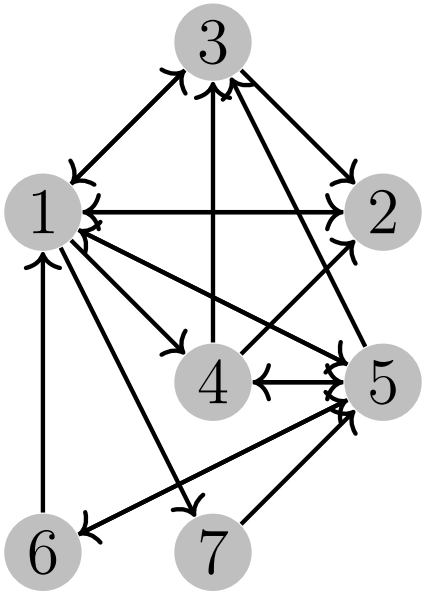


Figure 3.1: Example graph with 7 nodes and 18 edges.

There are many fields and applications where data can be found in graph format. The first area that comes to mind is social networks and all methods around social network analysis. Graphs represent interactions between users, groups or content. Community detection [80], link prediction between individuals [72] or influence maximization [40] are common applications where relationship graphs are at the heart of the problem. Other applications are related to the diffusion of information [230]. This field is concerned with the way in which data propagates within the social network. For example, the structure of the graph representing the diffusion of information in the social network is a very important point in determining the veracity and the viability of the information [95]. Propagation prediction is also used in other fields, such as epidemiology, to analyze the spread of disease or viruses [146, 184].

Recommender systems operate with graph data [108]. The user-item interactions are represented

in the form of a graph. We need to be able to correctly analyze and process this graph to make predictions of items for a user that are relevant relative to the item he or she has already consumed. The same is true in the field of NLP, where words, phrases and documents can be represented in the form of graphs, helping to represent the semantics and similarities of these objects for translation or document classification tasks.

The graphs are also a natural way of abstractly representing different locations and routes from one place to another, by associating weights to represent the distance or time separating these points. Graph neural networks (GNNs) can be used for route prediction delivery task [284]. And last but not least, graphs are also a natural representation of molecules. They are therefore at the heart of numerous methods in chemistry and bioinformatics for predicting molecular properties or protein structures.

GNNs play a very important role today. It does not only analyze the graph data itself, but also the data connectivity of the graph. The quality of a GNN is thus altered by the result of extracted graph structure information. The extraction could be enhanced by GNN model design or directly from the training dataset with a GNN-decoupled method.

In this chapter, we propose RankedDrop, a new sampling method to improve the extraction of graph structure information. This method is based on dropping-out technique, and it adopts a spatial-aware selection of edges to drop. It takes into account structure information of the graph to control the dropping-out, and its random selection of edges to be dropped is under the control of a probability generated with respect to graph's topological importance. Our experiments point out that RankedDrop provides high-quality and robust training results compared to the leading solutions. Furthermore, RankedDrop could be a framework plugin and combined with other GNN model improvements to maximize GNN quality.

3.2 Graph neural networks

Graph neural networks are deep learning models specifically designed to manipulate non-Euclidean data structures represented by graphs. There are several main categories of GNNs, with different approaches and strategies for manipulating graphs.

According to [287], the different types of GNNs can be classified into four main categories: the recurrent graph neural networks, the convolutional graph neural networks, the graph autoencoders and the spatial-temporal graph neural networks.

3.2.1 Recurrent graph neural networks

Recurrent graph neural networks (RGNNs) [140] extends the application of recurrent neural networks to graphs. Recurrent neural networks are models used to process sequential data. They are used in applications with temporality or a succession of states, such as the traffic prediction [35]. This was the first approach explored for dealing with graphs. These methods assume that graph nodes propagate information with their neighbors until they reach a stable equilibrium [287].

3.2.2 Graph convolutional neural networks

Graph convolutional networks (GCNs) [151] are neural networks that apply convolution operations to graphs. GCNs exploit graph structure to aggregate information from neighboring nodes and update node representations based on received and its own information in an interactive way. Convolution on graph-structured data is defined by the application of a convolution filter on the data received by neighboring nodes. This is a very popular type of model. This is a very large family of neural networks. They are very efficient for node classification. The attention mechanism has also been generalized to graphs with the graph attention network (GAT) [277]. The attention operator has been incorporated into the information propagation phase.

3.2.3 Graph autoencoders

Graph autoencoders (GAEs) [279] are models that learn to build representations of data structured as a graph in a low-dimensional vector or matrix space. GAEs are used to learn how to embed network information and network content. Architecture varies according to methods and models, but in general, there are two main components. The first is the encoder, which takes the graph and tries to capture the topological information of the graph's features to build a representation in a smaller dimension.

The second is the decoder, which uses the representation to try to reconstruct the graph as closely as possible to the original, while minimizing error. The objective when training this type of model is to minimize the difference between the original graph and the graph reconstructed from the embedding data. These models are used in graph generation to construct graphs with defined properties.

3.2.4 Spatial-temporal graph neural networks

Spatial-temporal graph neural networks (STGNNs) [297] are models specially designed to deal with dynamic graphs, i.e. graphs in which the attributes are dynamically modified over time. This is a specific type of graph that we will not be dealing with in the rest of this chapter.

Our investigation focus more specially on GCNs because of their widespread success in various applications.

3.3 Challenges in deep learning graph applications

The GNN backbones used today are built around a layer-wise propagation rule with similar baseline between the methods. They can by synthesized by

$$\boldsymbol{H}_{l+1} = \phi(\widetilde{\boldsymbol{A}}\boldsymbol{H}_l\boldsymbol{W}_l) \tag{3.1}$$

where $\boldsymbol{H}_l$ is the embedding matrix representing the l-th graph layer output, $\boldsymbol{W}_l$ is the weight matrix for the l-th layer and $\tilde{\mathbf{A}}$ is the modified graph adjacency matrix. These backbones offer interesting performances and have already shown their efficiency. However, this approach, called spectral, is really subject to the quality of the graph due to the fact that $\tilde{\mathbf{A}}$ is at the core of the method and is included in each layer. It is notably what limits the use of these networks with a large number of layers. Beside that, the attention-based spatial approaches allowed to obtain good results by focusing on an approach to reduce the noise of the graph by manipulating the weights of the neighborhood nodes to control the diffusion of the information in the graph. The sampling phase that can be done on the GNNs allows to influence this point as well. By modifying the adjacency matrix in this

way, the propagation of the data is less important and the performances can be improved when these modifications are done randomly [23, 231]. It is possible to use this improvement to go further by exploring the structural information of the graph to guide the selection phase. The dropping approach to modifying the adjacency matrix also represents an advantage in terms of execution time. Dropping reduces the number of non-zero elements nnz, which reduces the cost of matrix operations since the complexity of matrix multiplication is $\mathcal{O}(n \times nnz)$ with the main sparse storage formats.

The graph's structure contains a lot of information, which is why graphs are not reducible to a set of input perceptrons as with more classical data like images. Because of the irregularity of the data, the possibilities with non-Euclidean data are limited compared to other domains of deep learning that focus on more classical data. The main goal of GNNs study is to make an inference by capturing the network structure as well as node attributes [132]. We saw earlier that the GNNs place the structure at the heart of the methods, which requires a high-quality representation of the relationships between the nodes. The problem is that the quality of the graph structure is not easy to control. The structure of the graphs chosen to represent the data is not absolute. This quality can be altered by many factors: lack of information during data recovery, partially erroneous data, a preprocessing that removes information from the graph structure, etc. This imperfection can be represented as noise that interferes with the propagation of information. The problems caused by this noise become more important as the number of GNN layers increases. Indeed, the larger the depth of the GNN, the higher the noise propagation. This limits the performance of current implementations [302]. The sampling modules aim to improve the quality of the graph structure to enhance the propagation of information.

3.3.1 Scalability issues

Like many deep learning applications, the scalability of graph neural networks is a major challenge. The very large size of datasets means that the adjacency matrix associated with the graph has to be stored and manipulated as a sparse matrix. Otherwise, the memory costs for storing the adjacency matrix would be inordinately high. This requires matrix multiplication between sparse and dense matrices in a massively distributed environment. We will explore matrix multiplication sequences with sparse and dense matrices in more detail in section 4.4.

Load balancing is also a major challenge. To reduce communications, it is possible to visualize the processing of a neural network graph as sequential operations to update each node of the graph and communications phases to propagate information with neighboring nodes. The load-balancing of nodes between machines requires complex algorithms that are able to slice the graph into sub-graphs to minimize inter-sub-graph communications so as to limit communications between computing machines.

To meet scalability requirements, we have to adapt and propose new methods and algorithms well suited to distributed environments.

3.3.2 Oversmoothing

Oversmoothing refers to the phenomenon of smoothing predictions and representations of graph nodes when the number of network layers starts to increase [36]. Node features become similar to each other, which has a significant negative impact on model performance. With most GCNs, the best performance is obtained when the number of layers is low, 1, 2 or 4. The performance of these types of models dramatically decreases when more layers are added.

GCN model is actually a special form of Laplacian smoothing [179]. This phenomenon is directly linked to the propagation of information. The graph is used as a pattern to propagate information between nodes. At each layer, the model propagates this information to its neighbors and then updates these features with the information it has received. But at the next layer, the information distributed by the node is the result of aggregating the information from its neighbors, so it will broadcast information that is already a mixture of all its neighbors, and will update itself to take into account the mixture of its neighbors that has just been sent to it.

For a number of applications, the maximum distance between two nodes is very small. Milgram showed in an experiment that, in a very large population, the number of intermediaries between two random people is very low [272]. This work has shown that in the majority of cases, the number of intermediaries is less than 7 [197]. It is easy to understand why oversmoothing occurs. If the maximum value of the shortest path problem is less than the number of nodes [208], each node has received information from all the nodes in the graph. All values will therefore tend to converge towards

similar values.

3.3.3 Overfitting

Overfitting refers to excessively specific training of the model on the training data. The model captures noise and details too specific to the training dataset, so that when the model is used once trained, it is unable to generalize to new data.

Graph neural networks are very prone to overfitting. Since graphs are used as propagation structures, the same pattern is always used for propagation. This can hinder generalization with data other than the training data.

That's why we soon suggested varying the graph during training, so that propagation wouldn't always follow the same pattern. Changing the structure of the graph is an excellent way to vary the data during training. By creating a different subgraph at each epoch, the GNN can train with different graphs at each iteration. Many solutions have been proposed, and we will look at them in the next section.

3.4 Method sampling: overview of different strategies

Sampling methods with adding information are very rare because in general, it is complicated to create information in the graph without taking a great chance on the consistency of the information. [92] is one of these rare solutions to be effective in this field. It adds extra nodes to the graph to allow long range interactions between nodes. The experiment results show that not limiting propagation to direct neighbors is efficient. Taking an interest in the global structure therefore seems to be an approach with potential.

Sample and aggregate methods like GraphSage [111] were introduced in an early stage of the evolution of GNN to leverage node's neighbor information by an aggregator. The aggregator collects the topological structure of each node's neighborhood and the distribution of node features during the learning process. These methods use predefined aggregators to decide node dropping. The choice of aggregator used is a hyper-parameter that makes the training a difficult task. They also suffer

quickly from the growing layer size. Moreover, the coupling between sampling and learning limits the evolution of GNN.

One possible approach is to design a method that uses the output of the GNNs during training to control dropping, which would allow dynamic control of sampling. NeuralSparse [301] separates sampling and learning by using a dedicated sparsification deep neural network for the selection of dropping. Therefore, NeuralSparse samples subgraphs before applying GNNs can be coupled with different backbones to improve the performance. However, designing a sparsification network significantly increases the GNN learning complexity. The robustness and realisticness of such a method are still understudied.

Relying on the topology of the graph to direct the sampling is a good way to obtain good results in terms of accuracy. In the paper [300], Zhao *et al.* introduce GAUG, a method which uses data augmentation techniques to eliminate graph noises before or during a GNN training. GAUG includes an edge predictor to compute edge probabilities for all possible edges in the input graph. Similar to NeuralSparse, the edge predictor is a GNN. Significant computation power and learning complexity are required.

DropEdge [231] is a dropping method that generates a different subgraph at each epoch during GNN training. This approach of data preprocessing prevents oversmoothing, the backbones can train on many different subgraphs which would be the case with validating. This method, as well as other sampling approaches like DropGNN [211], use randomness. It is a simple but efficient way, to sample the input graph which requires less learning complexity and computation power than NeuralSparse and GAUG but have higher flexibility and performance than GraphSage and PinSage. However, these random-dropping methods do not take into account the topological information of the graph and drop different edges/nodes in a similar random way. A low-complexity, efficient and not just based on randomness high-performance solution is desirable for making GNN industrially applicable.

Methods have been proposed to take into account the graph topology in GNNs. Personalized PageRank has been used to manage the propagation phase of the information from the nodes in the neighborhood [25]. These methods provide solutions for using the personalized PageRank to manage the propagation in the graph. This approach tends to separate prediction from propagation,

which permits good scaling of the method but does not allow the use of graph structure information for training the model. Yang *et al.* [294] proposed another approach to control message-passing propagation with masks that filter the propagation strength to label-scale instead of edge-scale. By comparing the proximity of the labels of two neighboring nodes, a propagation strength is calculated for each label thus decreasing the level at which the propagation is managed. However, the accuracy obtained with this method and the additional complexity added to the model do not allow today to consider going down to the attribute level for propagation control.

3.5 Extract graph topology information

Many applications such as web search, recommendation systems or academic paper ranking need to classify the nodes that compose the graph according to their importance. In this context, we need to find a way of building a ranking from the information available in the graph. The graph's topology is an important source of information, but we still need to be able to extract information about the topology. The vast majority of metrics rely on the edges between nodes to calculate their structural significance and rank them in relation to one another. The graph link structure is used to classify graph nodes. We call these algorithms Link Analysis Ranking algorithms.

A Link Analysis Ranking algorithm is defined in [27] as a function that maps a graph $\mathcal{G}$ to a real vector of weight. In other words, this algorithm assigns one and only one numerical value to each node in the graph, representing the node's importance. The different nodes are compared according to the assigned values, so that the importance of one node is greater than another if the numerical value associated with it is higher.

Various centrality measures quantify a node's importance: the degree centrality classifies nodes according to the number of node connections, the betweenness centrality quantifies how often a node lies on the shortest path between other nodes and the closeness centrality reflects the average distance from a node to all other nodes.

Centrality measures provide a highly local perspective, ignoring global network properties. In order to extract and rank nodes from their global importance, we will use algorithms able to uncover

hidden structures and prioritize relevant information.

3.5.1 PageRank

In 1998, PageRank is initially an algorithm that was developed for ranking web pages [30]. Internet becoming larger and larger, the goal was to propose a solution to present the most relevant results to the user according to their searches. This method has revolutionized the search engines that were until then not very relevant while having a limited automation. It is the efficiency of this algorithm that will permit Google to quickly impose itself in the world of search engines and to become the giant that it is today. The idea behind this algorithm is to use the hypertext links of the Web pages to build a graph representing the links between the various Web pages. The idea is to use robots, called `web crawler`, which will continuously explore the Web to update the graph and collect information about the web pages content. Each node of the graph represents a web page and the edges, which are directed, represent the pages towards which point each of the pages. Already in 1996, the Internet was composed of more than 250,000 websites, and therefore even more web pages. But how to exploit the graph and the associated information to determine the relevance of each page?

This is where the PageRank will be used, it will allow to associate a score to each node of the graph. This score shows the relative importance of each page. The score will propagate in the graph and the score of a web page is directly influenced by the score of the pages that point to this page. The more the set of scores of the pages that point to a page is important, the more the score of this page will be important. Other parameters are also taken into account, such as the fact that pages that are composed of a large number of hyperlinks will give less importance to the pages to which it points.

One of the great strengths of PageRank is that the ranking of the relevance of a page is pre-calculated independently of the query, which is not the case with HITS and SALSA, two methods that we will see later. This allows the search engine to simply select the pages related to the user's query and display them according to their scores, thus reducing the response time.

The sequential version of the PageRank algorithm is presented in the algorithm 1. We only focus on the implementation from the power iteration approach, an algebraic method to compute the PageRank exist but will not be discussed in this document [280].

Algorithm 1 Serial PageRank algorithm implementation.

Input: $\boldsymbol{A}$ the adjacency (sparse) matrix, δ precision, β coefficient
Output: $\boldsymbol{v}$ vector of
PageRank score of size n
Initialisation :
1: $sum \leftarrow 0$
2: $err \leftarrow$ INF
3: new vector $\boldsymbol{t}$ of size n
4: assign $\frac{1}{n}$ to each element in v
START
5: **while** $err > \delta$ **do**
6: reset all element of $\boldsymbol{t}$ to 0
7: $\boldsymbol{t} \leftarrow$ SpMV between $\boldsymbol{A}$ and $\boldsymbol{v}$
8: **for** each $elem$ in $\boldsymbol{t}$ **do**
9: $elem \leftarrow \beta * elem + (1 - \beta) * \frac{1}{n}$
10: **end for**
11: $err \leftarrow$ norm between $\boldsymbol{t}$ and $\boldsymbol{v}$
12: $\boldsymbol{v} \leftarrow \boldsymbol{t}$
13: **end while**
14: **return** $\boldsymbol{v}$
END

3.5.2 HITS

Hypertext Induced Topic Search (HITS) have been proposed by Kleinberg [154]. It is an iterative algorithm that is used to rank web pages, like PageRank. This method is based on the following postulate. There are two different types of web pages: Some of them have the purpose of giving access to content (authoritative) while the other pages have the purpose of grouping a set of links to content pages (hubs). HITS will try to give each page an authoritative score and a hub score, in order to be able to rank them later on. Each page will therefore have two scores, unlike PageRank where one score was associated with each node. A page will have an important hub score if it links many pages with a high authority, on the contrary, a page with authority will be more important if it is referenced by many hub pages. The authority scores are therefore calculated from the hub scores and vice versa.

Unlike PageRank, HITS applies to a set of web pages that are related to the query. HITS first requires you to build a subgraph with only those pages that are related to the user's query. This allows to have a calculation cost much less important than the PageRank but does not allow to make these

calculations in advance. This makes HITS slower than PageRank to obtain page rankings. To reduce the computational load for each query, optimizations are often used, such as considering all web pages of a site as a single element. Thus reducing the size of the graph.

Algorithm 2 represents a sequential version of the HITS algorithm. We can clearly see the two loops that update the authority and hub scores.

Algorithm 2 Serial version of HITS.

Input: $\boldsymbol{A}$ the adjacency (sparse) matrix of size n,
Output: $\boldsymbol{h_A}$ vector of hub score of size n, $\boldsymbol{a_A}$ vector of authority score of size n

Initialisation :
1: initialize all $\boldsymbol{h_A}$ and $\boldsymbol{a_A}$ values to 1
START LOOP
2: **while** the weights did not converge **do**
3: **for** every hub $i \in \boldsymbol{a_A}$ **do**
4: $\boldsymbol{a_A}(i) \leftarrow \sum_{j \in B(i)} \boldsymbol{h_A}(j)$
5: **end for**
6: Normalize $\boldsymbol{a_A}$
7: **for** every hub $i \in \boldsymbol{h_A}$ **do**
8: $\boldsymbol{h_A}(i) \leftarrow \sum_{j \in F(i)} \boldsymbol{a_A}(j)$
9: **end for**
10: Normalize $\boldsymbol{h_A}$
11: **end while**
12: **return** $\boldsymbol{h_A}$, $\boldsymbol{a_A}$
END

3.5.3 SALSA

The SALSA algorithm was introduced in 2001 [173]. It is another page ranking algorithm which is a mix between PageRank and HITS.

As Hits, SALSA is a method which focus on a sub-graph which is topic dependent. This sub-graph is obtained by selecting only the pages related to the query. SALSA is therefore used on smaller graphs. Salsa can extract for each page a score of authority and hub

With the HITS method, the computation of the authority and hub scores was linked to each other. SALSA allows to compute both score vectors independently. The graph is transformed into a bi-partite graph. Concretely, the method starts by constructing two distinct matrices and then random walks

are computed on these two matrices to obtain the authority and hub scores [27]. The goal here is to find the dominant eigenvector of each matrix to calculate the importance of each page in the two categories. It is this step that refers to PageRank when Salsa is defined as a method that mixes both PageRank and HITS.

The sequential approach to compute the SALSA algorithm is given in the algorithm 3.

Algorithm 3 Algorithm to get the SALSA authority and hub score.

Input: $\boldsymbol{A}$ the adjacency (sparse) matrix of size n, δ precision
Output: $\boldsymbol{h_A}$ vector of hub score of size n, $\boldsymbol{a_A}$ vector of auth score of size n
Initialization :
START LOOP
1: $\boldsymbol{A}_r \leftarrow$ the matrix $\boldsymbol{A}$ where each nonzero element of $\boldsymbol{A}$ is divided by the number of non-zero values in the row
2: $\boldsymbol{A}_c \leftarrow$ the matrix $\boldsymbol{A}$ where each nonzero element of $\boldsymbol{A}$ is divided by the number of non-zero values in the column
3: $\widetilde{\boldsymbol{A}} \leftarrow \boldsymbol{A}_c^T \boldsymbol{A}_r$
4: $\widetilde{\boldsymbol{H}} \leftarrow \boldsymbol{A}_r \boldsymbol{A}_c^T$
5: $\boldsymbol{a_A} \leftarrow$ random walk on $\widetilde{\boldsymbol{A}}$ matrix
6: $\boldsymbol{h_A} \leftarrow$ random walk on $\widetilde{\boldsymbol{H}}$ matrix
7: **return** $\boldsymbol{h_A}, \boldsymbol{a_A}$
END

3.6 Proposed improvements by topology sampling

We introduce in this section RankedDrop, a sampling selection method based on extracted graph data. This allows to build subgraphs from the original graph to improve the training of GNNs on these datasets. This method and the experiments results has been published in an international conference [218]. It is an advanced sampling solution to take into account the topology of the graph during sampling. We propose a solution to control and frame the randomness in dropping. Based on topological information extracted from the graph, RankedDrop allows to obtain subgraphs that are topologically close to the original graph. Randomness is still inherent in this sampling approach, but the selection probability that RankedDrop assigns to the nodes ensures that it is used as a control framework to maintain consistency between the structure of the original graph and the generated

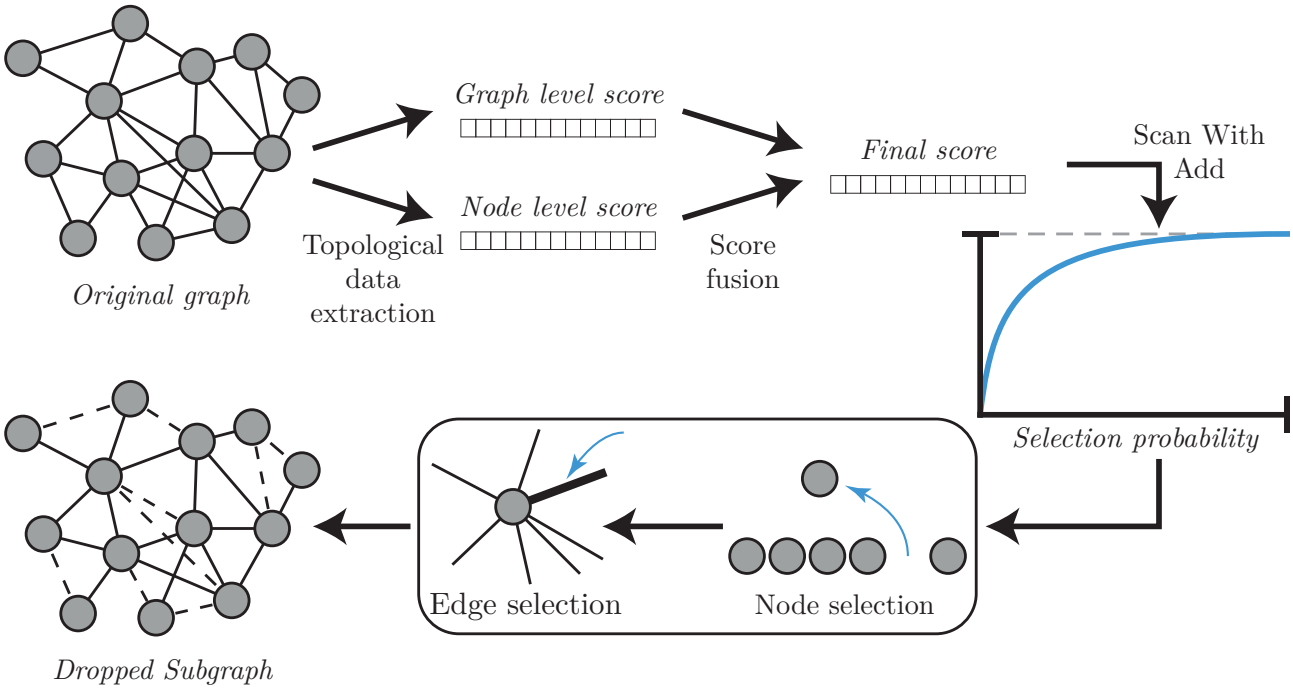


Figure 3.2: Overview of the RankedDrop sampling method to pass from the original graph to the dropped subgraph.

subgraph. At each epoch, a new sub-graph is used to vary the input data to train the model on similar graphs. An overview of the method is presented in figure 3.2, the different steps are presented in this section. The algorithms 4 and 5 summarizes the different steps that we will see next.

3.6.1 Information extraction

The first step is the extraction of data from the original graph. In order to make it easier to understand how the method works, we will separate the data extraction phase from the sampling phase, and we will discuss the data extraction in more detail in the section 3.7.1. For the moment, we assume that we have extracted an important set of data about the structure of the graph.

The data that is extracted from the graph should reflect the importance of the nodes in the graph structure. Data describes around which nodes the graph is structured. Not all nodes are equal with respect to the place they occupy in the graph. The approach proposed here is to merge the different features of the structure into a final score vector which will attribute to each node a smaller or larger importance and will make it possible to reflect the disparities between the nodes. Thus, the more

important a node is, the more important its score in the score vector will be.

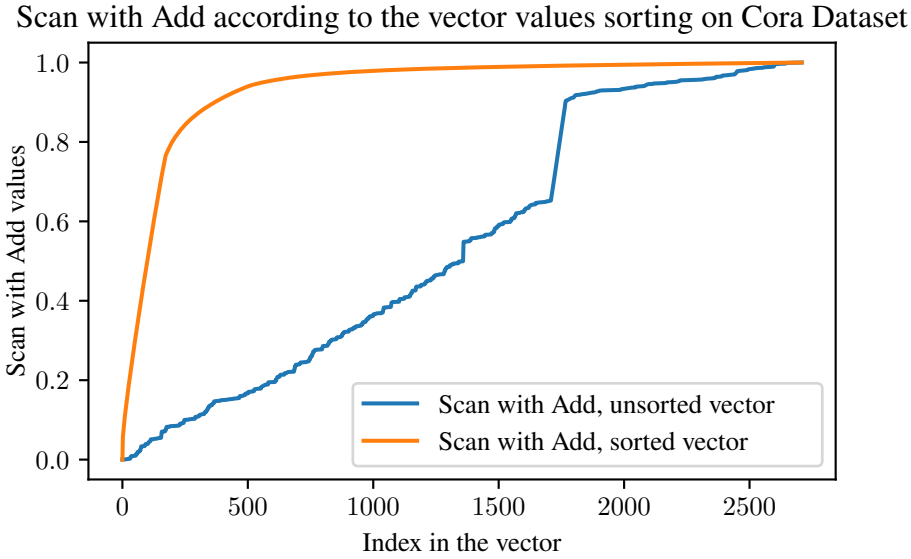


Figure 3.3: Scan with Add vector values for the Cora dataset. The orange curve represents the values when the final score values are sorted in descending order before applying the Scan with Add. In practice, there is no need to sort the values. However, for the sake of understanding and comparison, we will present the curves in this configuration.

The goal is to generate coherent subgraphs with respect to the original graph but which are different at each epoch. The use of randomness controlled by the topological information of the graph in the method has two advantages (1) the input graph varies and perturbations created at each epoch help to reduce overfitting. (2) There is no loss of information due to sampling, the graph is exploited in its entirety due to the fact that each node will have a non-zero probability of being selected. We opted to use probabilities to introduce randomness to generate different subgraphs from the extracted graph data. Randomness will direct the sampling and have different subgraphs but close to the original one. We have chosen to use the Scan with Add (a.k.a prefix sum, [22]) in our application to transform the score vector of the nodes into a probability distribution. The Scan with Add operation creates a new vector such as that each element corresponds to the successive partial sum of the elements of the original vector, taken in a decreasing order : i.e. from an array $\boldsymbol{v}$ composed of n elements, it builds an array $\boldsymbol{s}$ also of size n such that

$$\forall i \in [0, n-1], \boldsymbol{s}_i = \sum_{j=0}^{i} \boldsymbol{v}_j \tag{3.2}$$

In our application, the vector $\boldsymbol{v}$ corresponds to the score vector of graph nodes, which was computed from the extracted graph topological information. An application example is shown in Figure 3.3. This method is widely documented and there are powerful distributed algorithms [125]. The selection probability result of this operation continues to illustrate the relative importance of the node which was described by the final vector of scores. But it also controls the selection of nodes: The greater is the difference between elements i and $i+1$, for $i \in [0, n-2]$, the higher is the probability of the node $i+1$ to be selected.

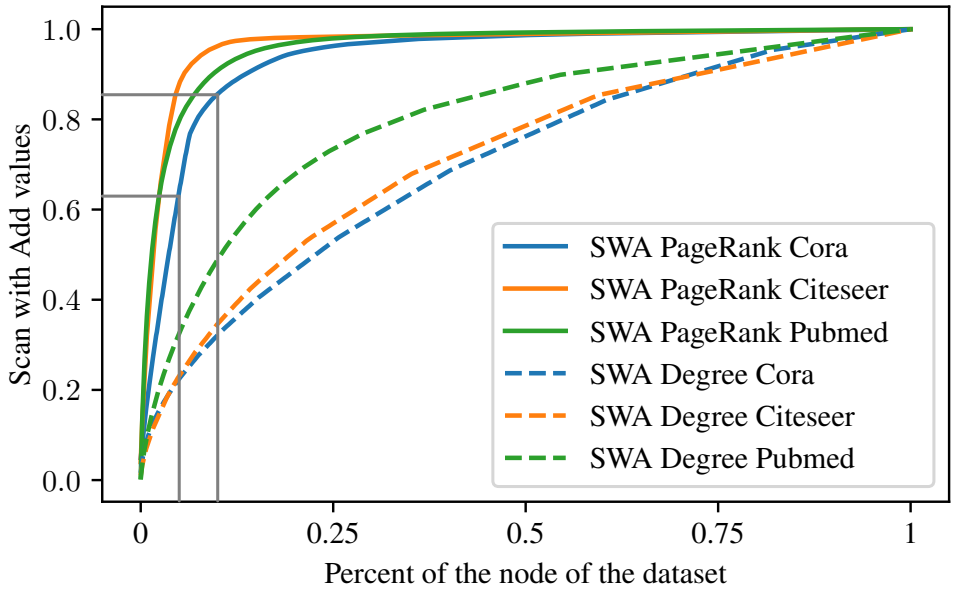


Figure 3.4: Variation of the Scan With Add (SWA) vectors for Cora, Citeseer and Pubmed datasets for local structure with degrees and global structure with PageRank. All the values of the array are presented in percentage on the x-axis. The y-axis represents the value corresponding to the index in the Scan with Add array.

In concrete terms, the selection probability is a vector. It is composed of a set of positive values such that $\forall i \in [0, n-2]$, the element i of the vector is less than or equal to the element $i+1$ and the last value of the vector is equal to 1. We thus obtain a probability interval which will be exploited to select the nodes for the sampling phase. Figure 3.4 represents graphically the Scan with Add vectors

that are obtained for different datasets in degrees and PageRank topological data, we will detail in the section 3.7. These curves represent the distribution of the values in the array. For comprehension reasons, the score values have been sorted in a decreasing order to better visualize the differences in score distributions. The distribution curves that are obtained during application look more like the unsorted curve in figure 3.3. The slope of the curve informs us about the distribution of the selection probabilities. Two annotations have been added around the curve of the Scan with Add of the PageRank for the Cora dataset. On the x-axis these annotations are located at 5 and 10 percent. What we can see from these annotations is that the top 5 percent of the nodes share 63 percent of the score, and the bottom 10 percent of the score shares 85 percent of the total score. This means that the selection of the node on the whole graph has an 85% chance of finding one of the nodes which happens to be among the 10% of the best rated. We can clearly understand on this figure that the nodes that are the best rated have a higher chance of being selected, and this probability will be stronger as the slope of the curve varies along the value. However, we can see that the less well rated nodes keep a probability of being selected. This probability will increase when some nodes are completely explored and the probabilities will be redistributed according to the remaining nodes.

3.6.2 Dropping selection

We based the proposed method on the idea of building a sampling phase that takes into account the information about the structure of the graph and also that allows to build different subgraphs with a similar structure to the graph. For the second part, we have to select the edges that will be part of the subgraph. The selection of the edges will go through the selection of the nodes based on the probability control that will be calculated so far. We will detail this selection in this part. Although our approach is a dropping method, our implementation leads us to build the subgraph. This means that we select the data that will be added to the subgraph. The dropped data is therefore the data that was not selected during the generation of the subgraph. The algorithm 4 summarizes the different steps that we will describe below.

Algorithm 4 Subgraph generation algorithm.

Input: $\boldsymbol{s}$ the Scan with Add vector, p the desired edges ratio compared to the graph $\mathcal{G}$, l list of edge-node association, m is the number of edges composing the original graph $\mathcal{G}$
Output: Subgraph $\widetilde{\mathcal{G}}$ generation

$\widetilde{\mathcal{G}} \leftarrow$ empty
while $\widetilde{\mathcal{G}}$ length $\leq p \times m$ **do**
  $r \leftarrow$ uniform random pull in $]0, 1[$
  $n \leftarrow$ select the node associated to r in s such as $s_{k-1} \leq r \leq s_k$
  $e \leftarrow$ select randomly one edge associated to node n in l
  Add e in $\widetilde{\mathcal{G}}$
end while
return $\widetilde{\mathcal{G}}$

Node selection

The selection of nodes is guided by the probability distribution we constructed earlier. To select a node, we will first draw a random number in a uniform way between 0 and 1 noted r. Let $\boldsymbol{s}$ be the output vector of the scan with Add and $\boldsymbol{s}_k$ and $\boldsymbol{s}_k$ the k^{th} element of this vector, the next step is to determine k such that

$$\boldsymbol{s}_{k-1} \leq r \leq \boldsymbol{s}_k \tag{3.3}$$

Thus, it is the node k that is selected for the next step of the operation. Note that the array $\boldsymbol{s}$ is increasing, so it is possible to use linear research algorithms to improve the performance of this phase (for example, the binary search can reduce the complexity to $\mathcal{O}(\log n)$). By selecting nodes in this way, we ensure that nodes with higher scores have a greater chance of being selected because they represent a larger range over the set of possible values of r. At the same time, the less important nodes still have a chance to be selected but with a lower probability, ensuring that the generated subgraphs will be different from one epoch to another.

Edge selection

The choice of the edge to be sampled from the selected node requires to associate each edge of the graph to a node to have a list of edges associated to each node. The creation of this list can be done in advance, as the same list will be used during all the training. This list is called edge-node association

in the algorithm 4. Edges can either be associated with the tail or the head of the edge. An edge is only associated with one node. The length of this list is therefore equal to the number of edges in the graph. If it is an undirected graph, it must be considered as directed, and each directed edge must be associated with the endpoints of the chosen edge. The goal is to build the subgraph's adjacency matrix so that it can be used in the next training epoch. We have to focus on the selection of the non-zero values that will be kept in this matrix. This is equivalent to selecting the edges that will be used to build the submatrix and this list of edges will be used to build it.

3.6.3 The RankedDrop algorithm

Algorithm 5 RankedDrop general algorithm.

Input: $\mathcal{G}$ the original graph, nn the GNN model, the number of epoch e
Output: The trained GNN model
1: $e \leftarrow$ Data extracted from $\mathcal{G}$
2: $v \leftarrow$ compute final score vector (e)
3: $s \leftarrow$ Scan with Add of v
4: **for** i in 1 to epoch **do**
5: $\quad \mathcal{G}_i \leftarrow$ Algorithm 4 //Subgraph generation
6: $\quad$ train nn on the graph $\mathcal{G}_i$
7: **end for**
8: **return** nn

We summarize the proposed method in the current section. For this purpose, we rely on the algorithm 5 which synthesizes the whole method. The first step is to extract data from the original graph at line 1. The aim is to extract data to represent the topology of the graph. One or more data based on different approaches can be extracted and used to build a final score vector. This normalized vector stores a unique value for each nodes in the original graph. The value associated with each node in this vector is intended to represent its importance in the graph. This final score, compute at line 2 in the algorithm, is used to build the selection probability thanks to the Scan with Add at line 3. All the previous steps are to be done only once per graph and we have explained more in detail these steps in the section 3.6.1. From the line 4, we will construct a subgraph for each epoch. Therefore, we run the algorithm 4 which return a subgraph of the original graph according to the topological importance of nodes. This importance is represented by the selection probabilities that we have calculated. In the

algorithm 4, we use a random draw, in the line 3, to select a node in the Scan with Add result vector and then, we select one edge associated with this node and we append this edge to the subgraph (line 6 in the algorithm 4). The selection of a node and an associated edge is repeated until the subgraph has the required size. The generated subgraph will be used for this epoch when training the GNN. The subgraph generation steps were explained in section 3.6.2.

The proposed algorithm is a generic algorithm that can be applied to most GNN architectures. Parallelizable methods which are applicable to this type of architecture can still be applied because the modification is limited to the adjacency matrix used in the algorithm [202].

Now that we have seen in detail how RankedDrop method works, we will see and compare its performance in the next section.

3.7 Case studies and experimental results

Three standard citation datasets were used in our experiments: Cora, Citeseer and Pubmed. These datasets represent collections of scientific articles that are classified according to the main research topic. Detail information of these datasets are described in the table 3.1.

	Cora	*Citeseer*	*Pubmed*
Nb of nodes	2 707	3 327	19 717
Nb of edges	5 429	4 732	44 338
edges/node (mean)	2.01	1.42	2.25
Max node degree	168	99	171
Max val PR	0.0492	0.0401	0.00610

Table 3.1: Datasets global information.

To compare our results, we use the accuracies of the version without sampling and with sampling performed by DropEdge. The dropping phase that is performed with DropEdge is completely random. Comparing the results between these methods shows the influence of the randomness control on the accuracies of the GNN models.

The extraction of graph data, the score computation until edge selection and dropping were done before the GCN training on Intel Xeon Processor E5-2690 with 8 cores. We used the C++ language to

perform the above mentioned operations. Several high performance computing techniques were used, in particular the computations on the adjacency matrices were done in parallel. To gain computational efficiency and memory, all the adjacency matrices were compressed and stored in the COO format, a sparse storage format. The training of GNNs were done on Nvidia Tesla V100 PCIe 16GB GPUs with a TensorFlow implementation. Since our solution can generate subgraphs independently of the training, we first generated the subgraphs before using them with different hyperparameters and GNN models. We used the Protobuf library to transfer the data between the two programming languages.

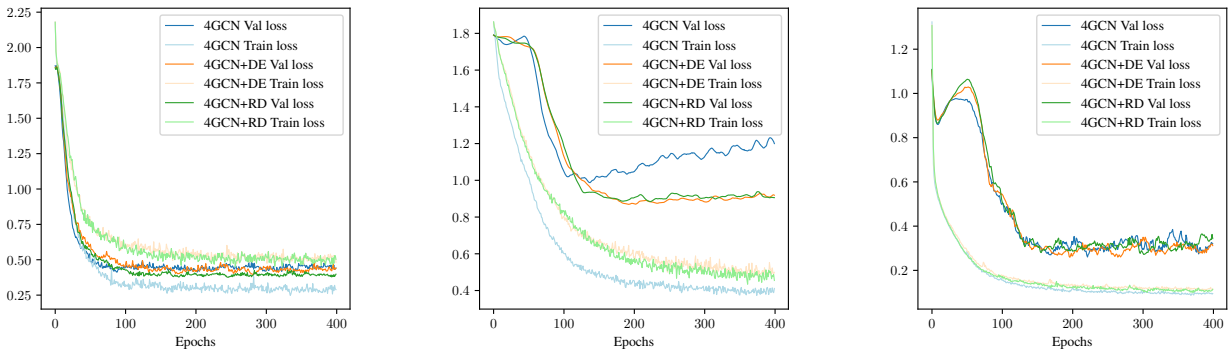
3.7.1 Score computation in presented experiments

As indicated in the section 3.6, we will here discuss the extraction of data from the graph structure to build the score vector. The final score vector should reflect the importance of a node in the graph and the probability to be selected in function of the ranking. As a reminder, the RankedDrop method relies on the fact that the more important the score associated with the node is, the more it will be selected when building the subgraphs. For our experiments, we used two main data on the graph structure: PageRank and degrees. This choice allows us to have both an extraction of the importance of the node in the local structure and in the global structure of the graph.

Global structure Different graph node ranking algorithms could be used here to judge importance of each node on the global graph. In RankedDrop, we decide to use the PageRank algorithm to generate the score of importance, because (1) PageRank is a well-known and studied algorithm of the last decades, by our knowledge it can be easily implemented in a distributed way to accelerate its computation [275]; (2) It was already used in GNNs to reduce the oversmoothing [24]. From the adjacency matrix $\boldsymbol{A}$, a vector of size n will be returned and will contain the score of each of the n nodes. The main operation of each iteration of the PageRank algorithm is a sequence of matrix-vector multiplication where the output vector is used to perform the next iteration multiplication. By considering $\boldsymbol{A}$ as sparse, the cost of this sequence of sparse matrix-vector multiplications is reduced and can be executed efficiently in a distributed way [134], which allows to optimize the extra computations that PageRank requires. This iterative method stops when the convergence has reached

the expected precision. The result vector of the last iteration contains then the scores of each node of the graph and all elements are between 0 and 1. The higher the score, the more important the node is in the global graph. A β coefficient is also introduced during the PageRank. It is an optimization allowing to redistribute a part of the scores of each node among all the other nodes. In this way, the convergence of the result vector is faster and avoids that all the score is distributed only within the strongly connected component. Conventionally, the β coefficient is fixed around 0.85, this value was used for the following experiments.

Local structure The node-level information reflect the local impact that a node will have on its neighborhood. We will use the degrees of each node as information to determine a score for the local structure. Indeed, it is by its neighborhood that the node will propagate information during the execution of the GNN. If a node has many neighbors, it will have an impact at each layer on them and thus the information it contains will be strongly taken into account at the local level. The degrees are extracted from the adjacency matrix $\boldsymbol{A}$ of the graph.

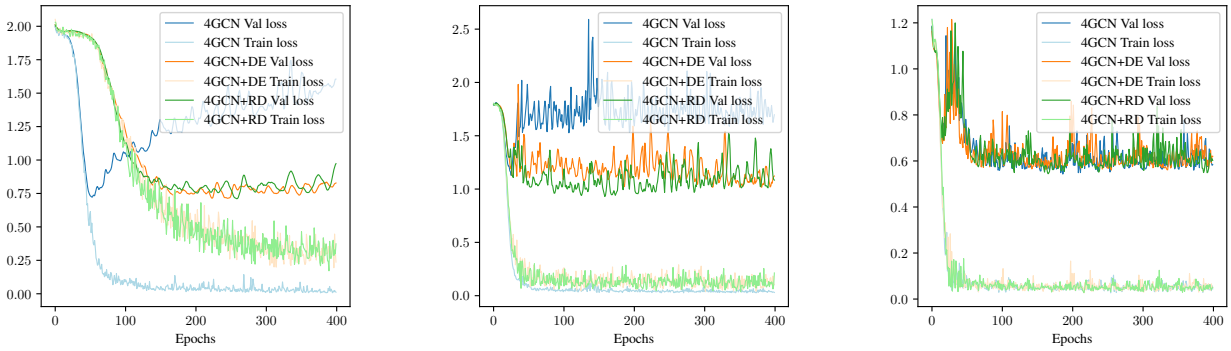


(a) Cora (b) Citeseer (c) Pubmed

Figure 3.5: Loss payout curve for datasets with the GCN 4 layers architecture in full-supervised learning. Comparison between the original method, DropEdge (DE) and RankedDrop (RD).

3.7.2 Scalability and impact on overfitting

In the following subsections, the RankedDrop (RD) method was compared to both an original GNN method without dropping and with the DropEdge [231] (DE) method discussed in the section 3.2.2. GCN used is the original version.



(a) Cora (b) Citeseer (c) Pubmed

Figure 3.6: Loss payout curve for datasets with the GCN 4 layers architecture in semi-supervised learning.

Dataset	Layers	Original	DE	RD
Cora	2	81.10	82.80	82.90
	4	78.50	78.80	82.00
	8	31.10	53.10	63.90
Citeseer	2	70.80	72.30	73.20
	4	61.20	68.80	71.30
	8	30.20	33.20	45.50
Pubmed	2	79.00	79.60	79.90
	4	78.30	77.70	79.40
	8	61.20	54.50	77.10

Table 3.2: Accuracy comparison for semi-supervised learning methods for GCN architecture.

We analyze in this section the performance of the method with the GCN architecture as initially introduced in [151]. The accuracy obtained from the original GCN, GCN with DropEdge and GCN with RankedDrop are summarized in the table 3.2 with varied network depths. The hyper-parameters to train the 2-layer GCN were directly from the paper of DropEdge, and the ones for 4 and 8 layers are the best configurations we found. The table 3.2 shows us that both dropping methods perform better performance than the no-dropping original method. This observation validates our claim that higher quality graph structure information improves the performance of GNN. More importantly, we can observe that RankedDrop helps GNN always obtain better accuracy than DropEdge. It confirmed our claim that higher-quality graph structure information requires better analysis of the graph's topology.

Moreover, the deeper the GCN is, the better accuracy improvement RankedDrop offers compared to DropEdge. The accuracy obtained with RankedDrop for the 8-layer GCN could be up to 20% better than the one with DropEdge. Even for the 2-layer GCN, the accuracies of RankedDrop are equivalent or superior to those well-tuned by DropEdge.

The figures 3.5 and 3.6 show the training and validation loss curves in full-supervised and semi-supervised learning, respectively. All curves for the same dataset were obtained with the same hyper-parameters. Only the percentage of dropping edges is different between DropEdge and RankedDrop. We can observe that the two dropping methods generally have similar behavior and better loss convergence than the original GCN. However, in some cases, the validation loss of RankedDrop converges again better than the one of DropEdge. These experiments show that RankedDrop is an interesting method to reduce the overfitting phenomenon and stabilize the loss. RankedDrop also has the same behavior on oversmoothing reduction as DropEdge then we did not study this part.

3.7.3 Portability and efficiency on different GNN models

We evaluate here, for each dataset, with three different GCN backbones: Vanilla network (GCN) [151] , Inception network (IncepGCN) [266] and dense network (JK) [289].

DATASET	TYPE	ORIGINAL	DE	RD
CORA	4 GCN	88.70	90.50	90.70
	4 INCEPGCN	89.90	91.10	91.80
	16 JK	90.40	91.40	88.30
CITESEER	4 GCN	76.70	79.20	79.90
	8 INCEPGCN	79.20	80.50	80.30
	8 JK	79.60	80.20	79.80
PUBMED	4 GCN	88.70	90.50	90.70
	4 INCEPGCN	89.90	91.10	91.80
	16 JK	90.40	91.40	88.30

Table 3.3: Accuracy comparison for full-supervised learning with GCN, IncepGCN and JK architectures based on the most efficient dropping architectures with the DropEdge method.

The accuracy results in full-supervised learning obtained without a dropping method (Original), with DropEdge (DE) and with RankedDrop (RD) are summarized in the table 3.3. We applied

to RankedDrop the same hyper-parameters of the best accuracy one from DropEdge to study the portability and efficiency of RankedDrop, but not the best-trained ones to explore the accuracy limit. The only parameter changed from DropEdge for RankedDrop is the percentage of dropping edges, which is not a hyper-parameter for learning but a parameter used during the sampling to determine the size of the subgraphs to build. The results of accuracy are very similar between RankedDrop and DropEdge. In particular, RankedDrop achieved better accuracy results than DropEdge on Cora the smallest dataset, where we control the overfitting better. Meanwhile, we can see that the deeper the network is (for example, with 16-layer JK), the more significant different results are obtained between DropEdge and RankedDrop. According to our analysis, deeper networks are more sensitive to hyper-parameters. If we apply RankedDrop tuned hyper-parameters, we could obtain better accuracy results too. Nevertheless, RankedDrop can still benefit from the hyper-parameters tuned from other methods with acceptable performance.

The additional computation power required for RankedDrop can be summarized as the extraction of the data and the creation of the subgraphs. The complexity of PageRank is $\mathcal{O}(2nnz - n)$ and the complexity of subgraph creation is $\mathcal{O}(nnz)$ where nnz is the number of graph edge and n the number of graph nodes. The PageRank, like any other data extraction is only done once per graph and the generation of the subgraph is done at each epoch. This generation complexity is negligible compared to the complexity required to train the graph neural network model.

3.7.4 Experiments reproducibility

The parameters used to generate the accuracy that are presented in the table 3.2 and 3.3 are explained in the table 3.4. There are both the hyper-parameters of the models that are used for the execution of the backbones. As our implementation is based on the DropEdge [231] solution, its model hyper-parameters such as learning rate, random seed or number of epochs are also hyper-parameters for our RankedDrop method. All hyper-parameters that have not been modified from the DropEdge version have not been added to the table 3.4. We have also added a number of additional hyperparameters to control the selection of the edges to drop. We have implemented three ways to take into account the information from the structure of the graph. This is the parameter which is named `score`. Either we

have used only the degree information or the PageRank information, which is respectively indicated by `Deg` and `PR`, or we have used both at the same time to build the score vector and it is noted `PRxD`. In addition to this parameter, we have influenced the choice of edges to remove from the graph with the following parameters:

- `dd`: It is a boolean that removes the edge in the opposite direction of the selected edge when the dataset is symmetric. Vertices are removed in pairs and this allows to keep a undirected graph.
- `reverse`: It is a boolean that allows to reverse the adjacency matrix. By doing this, each edge is no longer associated with the tail node but with the head node, and if the scores of the two nodes associated with that edge are not the same, it changes the probability of selecting that particular edge.
- `lowest`: It is a boolean that reverses the ranking of the nodes of the graph using the reciprocal of the score associated to each node.

Ref	Backbone	Dataset	nlayers	Hyper-parameters
Table 3.2	GCN	Cora	2	lr:0.001, weight-decay:1e-4, sampling-percent:0.7, score:PRxD, dd:false, reverse:true, lowest:true, niter:400
Table 3.2	GCN	Citeseer	2	lr:0.007, weight-decay:1e-4, sampling-percent:0.6, score:PR, dd:false, reverse:false, lowest:true, niter:400
Table 3.2	GCN	Pubmed	2	lr:0.009, weight-decay:1e-2, sampling-percent:0.8, score:PR, dd:true, reverse:false, lowest:true, niter:400
Table 3.2	GCN	Cora	4	lr:0.004, weight-decay:1e-4, sampling-percent:0.3, score:PRxD, dd:false, reverse:true, lowest:true, niter:400
Table 3.2	GCN	Citeseer	4	lr:0.008, weight-decay:1e-3, sampling-percent:0.1, score:Deg, dd:false, reverse:true, lowest:false, niter:400
Table 3.2	GCN	Pubmed	4	lr:0.008, weight-decay:1e-2, sampling-percent:0.9, score:Deg, dd:false, reverse:true, lowest:false, niter:400
Table 3.2	GCN	Cora	8	lr:0.003, weight-decay:1e-5, sampling-percent:0.7, score:PR, dd:true, reverse:true, lowest:false, niter:1000
Table 3.2	GCN	Citeseer	8	lr:0.001, weight-decay:1e-5, sampling-percent:0.5, score:PRxD, dd:false, reverse:true, lowest:false, niter:1000
Table 3.2	GCN	Pubmed	8	lr:0.006, weight-decay:1e-4, sampling-percent:0.5, score:PRxD, dd:true, reverse:true, lowest:true, niter:1000
Table 3.3	GCN	Cora	4	lr:0.01, weight-decay:0.005, sampling-percent:0.6, score:Deg, dd:true, reverse:true, lowest:true, niter:400
Table 3.3	GCN	Citeseer	4	lr:0.009, weight-decay:1e-3, sampling-percent:0.1, score:Deg, dd:true, reverse:true, lowest:false, niter:400
Table 3.3	GCN	Pubmed	4	lr:0.01, weight-decay:1e-3, sampling-percent:0.2, score:PRxD, dd:true, reverse:true, lowest:false, niter:400
Table 3.3	IncepGCN	Cora	8	lr:0.01, weight-decay:1e-3, sampling-percent:0.1, score:PR, dd:true, reverse:false, lowest:true, niter:400
Table 3.3	IncepGCN	Citeseer	8	lr:0.002, weight-decay:0.005, sampling-percent:0.1, score:Deg, dd:true, reverse:true, lowest:false, niter:400
Table 3.3	IncepGCN	Pubmed	4	lr:0.002, weight-decay:1e-5, sampling-percent:0.3, score:PRxD, dd:false, reverse:true, lowest:true, niter:400
Table 3.3	JK	Cora	16	lr:0.008, weight-decay:5e-4, sampling-percent:0.1, score:PR, dd:true, reverse:true, lowest:true, niter:400
Table 3.3	JK	Citeseer	8	lr:0.004, weight-decay:5e-5, sampling-percent:0.8, score:PR, dd:true, reverse:true, lowest:true, niter:400
Table 3.3	JK	Pubmed	64	lr:0.005, weight-decay:1e-4, sampling-percent:0.9, score:Deg, dd:false, reverse:false, lowest:false, niter:400

Table 3.4: Hyper-parameters used to obtain the accuracy presented in the chapter 3 with the RankedDrop method.

3.8 Conclusion

The RankedDrop method presented in this section is a flexible and easy-to-use solution for graph sampling with topology-aware controls for selecting dropping edges. RankedDrop could be effectively introduced into Mindspore [39] as a data augmentation solution for the various GNN models currently proposed in this open-source framework. RankedDrop appears as a mixed-use solution for sampling and retaining the essential information about the structure in generated subgraphs. RankedDrop maintains the advantages of the SOTA dropping methods, including low complexity in computation and a sensible reduction of oversmoothing and overfitting, as well as the possibility to combine with improvements on the GNN architecture design. The results generated by RankedDrop are very encouraging and promising.

In the future, RankedDrop may be extended to help the development of deeper GNNs. Moreover, thanks to the low computation complexity, we could develop a parallel and distributed computing version of RankedDrop with fewer obstacles. An important opening of this study is the possibility of using RankedDrop to improve the training of neural networks on denser graphs.

Chapter 4

Optimizing Sparse Matrix Operations for Deep Learning in Distributed Systems

4.1 Introduction

The increasing complexity of deep learning models has necessitated more efficient computational techniques, particularly in the context of large-scale distributed environments. One of the most prominent challenges in this domain is the efficient handling of sparse data structures, which, as we discussed in the previous chapter, frequently arise in applications such as graph neural networks and recommendation systems. Sparse matrices, which contain a significant proportion of zero values, introduce unique computational challenges, including memory inefficiency and suboptimal workload distribution. Addressing these challenges is crucial to ensuring the scalability and performance of deep learning models in high-performance computing environments.

In this chapter, we investigate the role of data sparsity in deep learning, exploring both its algorithmic advantages and its computational implications. Our focus is on key sparse matrix operations that are commonly encountered in deep learning applications, particularly in distributed settings. We propose optimized methods for storing, processing, and distributing sparse data, aiming to enhance computational efficiency while maintaining model accuracy. Additionally, we introduce novel techniques for data distribution and workload balancing. These contributions provide a foundation for

improving the scalability of deep learning models, making them more viable for large-scale applications.

After introducing sparse matrices and the challenges associated with these matrix operations in sections 4.2 and 4.3, we will present our data distribution strategy for successive sparse matrix operations in section 4.4. Then, we will introduce our approach for efficiently constructing the co-occurrence matrix in the section 4.5. Following an *a priori* analysis in 4.6, we will conclude with experimental results in section 4.7.

4.2 Fundamentals of sparse matrix computation

4.2.1 Definition of sparse matrix

A sparse matrix is a matrix composed mainly of zeros and containing relatively few non-zero elements. There is no precise threshold below which a matrix is considered sparse. A good definition would be that a matrix is sparse when it is advantageous to consider it as such in computations. When you want to manipulate or store sparse matrices with a computer, it is advantageous and often necessary to use algorithms and data structures that take into account the sparse structure of the matrix. These data structures will only store the non-zero elements of the matrix, as well as the position of these non-zero elements. Based on this information, it is possible to rebuild the matrix overall, considering all elements for which we have no information as null elements.

By convention, the number of non-zero elements of a sparse matrix is denoted nnz. The *density* of a sparse matrix is a measure of the proportion of non-zero elements to the total number of elements in the matrix. If we consider the matrix $\boldsymbol{A}$ of size $(m \times n)$ with nnz non-zero elements, the density d is calculated as the ratio

$$d = \frac{nnz}{m \times n} \tag{4.1}$$

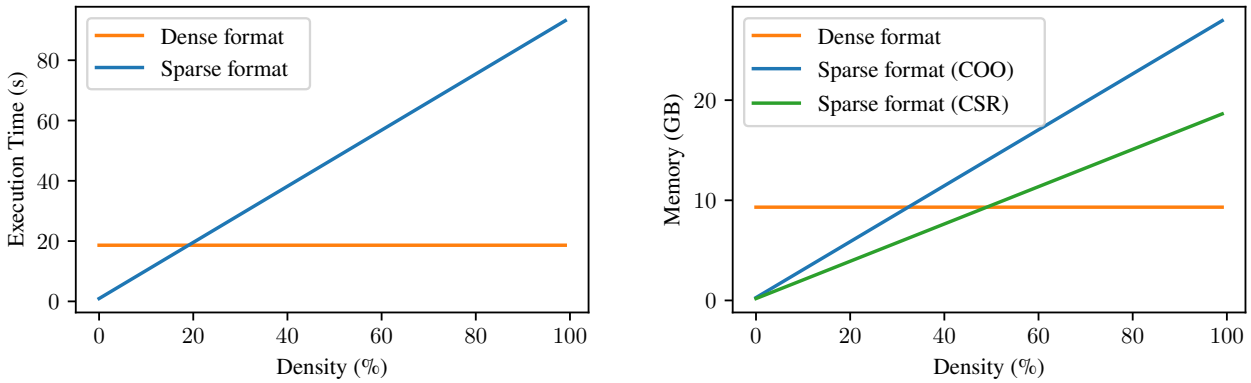
The proportion of zero elements to the total number of elements in the matrix is called *sparsity*.

The sparsity s and density d are directly correlated because s is defined as

$$s = 1 - d \tag{4.2}$$

By definition, the density of a sparse matrix is low and its sparsity is high. The lower the density of a matrix, the more advantageous it is to process and manipulate it in a sparse format. There are a number of sparse matrix storage formats, as we will see in the next section.

In figure 4.1, I've calculated the theoretical execution times for a matrix-vector product with a $100,000$ square matrix, as well as an analysis of the memory required to store this same matrix. We can see that the performance gains and memory savings are very significant when the matrix density is low. I've only taken into account the fact that pipelining is not possible with sparse formats because of the unstructured data, unlike dense formats which store the matrix elements in a 2D array to build the figure 4.1a. In practice, other factors have a significant impact on the execution time of sparse matrix operations: the storage format, memory efficiency for accessing indices or the distribution of non-zero elements in the matrix. In application, the use of matrix compression formats is interesting at even lower density levels. In terms of memory, however, the gains are interesting from a density of 30% to 50% depending on the storage format, as can be seen in figure 4.1b.



(a) Theoretical execution time in seconds of a SpMV.

(b) Memory comparison in GB.

Figure 4.1: Theoretical memory space needed to store the matrix and execution time to realize a matrix-vector multiplication in function of data density and compression method. Execution time is calculated on the assumption that pipelining is not possible with the sparse format due to unstructured data.

4.2.2 Sparse storage formats

There are many different types of sparse storage format. The choice of format depends on several factors. First of all, it depends on the special properties of the matrix. Take, for example, a tridiagonal matrix. The non-zero elements follow a structure and it is therefore advantageous to store the matrix using storage formats specially designed for this type of matrix, such as diagonal storage [3]. Another factor is the distribution of non-zero elements in the matrix. Some formats can only be used when the number of non-zero elements is evenly distributed per row or column. A final factor is the evolution of the matrix content. Some storage formats will not be appropriate if the number of non-zero elements changes during processing. Indeed, some formats will require extra computation to add non-zero elements to the matrix. Research has been performed to automatically determine the optimal storage format based on the matrix structure and the computing environment [193, 194]. In the remainder of this section, we present the most popular sparse storage formats.

To get a better idea of how the information for the different storage formats is stored, the following square matrix with $nnz = 8$ will be used as an example:

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 15 \\ 21 & 22 & 0 & 0 & 0 \\ 0 & 0 & 33 & 0 & 35 \\ 41 & 42 & 0 & 0 & 0 \\ 0 & 0 & 0 & 54 & 0 \end{pmatrix} \tag{4.3}$$

Coordinate format (COO)

This storage format is certainly the simplest of all : it consists to store data in three different vectors. The size of vectors is the number of nonzero values nnz in the matrix. These three vectors will store one of these data for each nonzero value : the row index, the column index and the value. With these three vector, it is very simple to have information about a value : simply read the i^{th} value of each vector to find the information for that point. This format has the advantage of being easy to understand and it is very quick to add new nonzero values because just need to increase the size of the vectors and add these new values at the end. A representation of the matrix 4.3 in the COO format is:

$$
\begin{aligned}
row &= \begin{pmatrix} 1 & 2 & 2 & 3 & 3 & 4 & 4 & 5 \end{pmatrix} \\
col &= \begin{pmatrix} 5 & 1 & 2 & 3 & 5 & 1 & 2 & 4 \end{pmatrix} \\
val &= \begin{pmatrix} 15 & 21 & 22 & 33 & 35 & 41 & 42 & 54 \end{pmatrix}
\end{aligned}
$$

Compressed sparse row (CSR)

This storage format, also known as the Yale format, stores the sparse matrix in three vectors like the COO format. A first vector, with a size of nnz will be used to store the nonzero values in the order of appearance in the matrix, line by line. A second vector with the same size will store for each value of the first vector the index of the column in the matrix. Finally, the third and last vector, with a size of (number of rows + 1) will be used to store the index of the first value of each row. This format allows to take less memory space than the COO format, but it is more complicated to modify the matrix because it requires many modifications in the vectors, where the COO format does not require a specific order of data storage in the vectors. The storage of the matrix 4.3 in the CSR format is:

$$
\begin{aligned}
ptr &= \begin{pmatrix} 0 & 1 & 3 & 5 & 7 & 8 \end{pmatrix} \\
ind &= \begin{pmatrix} 5 & 1 & 2 & 3 & 5 & 1 & 2 & 4 \end{pmatrix} \\
val &= \begin{pmatrix} 15 & 21 & 22 & 33 & 35 & 41 & 42 & 54 \end{pmatrix}
\end{aligned}
$$

Compressed sparse column (CSC)

In the same way as the CSR format, the CSC format stores matrix elements column by column. This time, we store the row indices of non-zero elements. This gives the matrix 4.3 the following vectors:

$$
\begin{aligned}
ptr &= \begin{pmatrix} 0 & 2 & 4 & 5 & 6 & 8 \end{pmatrix} \\
ind &= \begin{pmatrix} 2 & 4 & 2 & 4 & 3 & 5 & 1 & 3 \end{pmatrix} \\
val &= \begin{pmatrix} 21 & 41 & 22 & 42 & 33 & 54 & 15 & 35 \end{pmatrix}
\end{aligned}
$$

ELLPACK (ELL)

This format implicitly imposes to have approximately the same number of nonzero values on each row of the sparse matrix. The values are stored in two matrices of size (number of rows $\times$ maximum

number of nonzero values in a row). For this, we will add in the first matrix all the nonzero elements on the corresponding line. This is similar to compressing the matrix by removing the nonzero values from each row. The index of the column associated with each of these values is stored in the other matrix. For a nonzero value, its row is obtained by looking at its position in the matrix and its column with the second matrix. This format makes it easy to modify the matrix, on condition that you do not want to add a new value to a row that already contains the maximum number of non-zero values. In this case, it will be necessary to add a column to each of the two matrices. It is a format that tends to be more demanding in terms of memory than the first two other formats, but it has the particularity of being quite intuitive. The ELLPACK representation of the matrix 4.3 is

$$colInd = \begin{pmatrix} 5 & \_ \\ 1 & 2 \\ 3 & 4 \\ 2 & 1 \\ 3 & \_ \end{pmatrix} \qquad val = \begin{pmatrix} 15 & \_ \\ 21 & 22 \\ 33 & 35 \\ 42 & 41 \\ 54 & \_ \end{pmatrix}$$

4.2.3 Sparse matrix applications

Sparse matrices are at the heart of a wide range of applications in many different fields, including scientific computing [206], mathematical modeling [150], bioinformatics [243], engineering and economics [50].

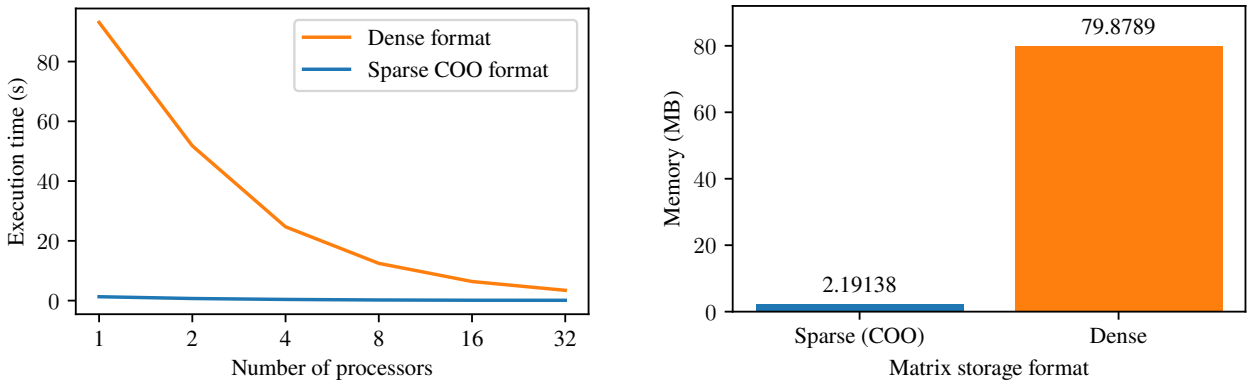
One common approach is to store the structure of a graph by storing the adjacency matrix. The adjacency matrix size is $(n \times n)$ if the graph is composed of n nodes. Each element $\boldsymbol{A}_{i,j}$ of the adjacency matrix indicate of there is an edges from the node i to the node j. The adjacency matrix of the example graph in figure 3.1 is the following matrix:

$$\begin{pmatrix} 0 & 1 & 1 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix} \tag{4.4}$$

This matrix is a fundamental tools in graph theory. The spectral graph theory study the relationship between the eigenvalues and the eigenvectors of the adjacency matrix and the graph. The adjacency

matrix of a graph is sparse, specially when the graph dimension is high. Storing this matrix in memory in a sparse storage format will reduces the memory space needed.

Sparse linear algebra methods for multiplication with a sparse matrix are essential to take full advantage of the matrix's sparsity. Figure 4.2 shows the performance differences with the FIDAPM37 [65, 66] matrix, a square matrix of size 9152 from the FIDAP set, composed of $765,944$ non-zero elements, representing a density of 0.914%. We can see from sub-figure 4.2b that using the sparse format to compress the matrix saves on the memory needed to store it. The COO format requires 36 times less memory to store the matrix. We can also observe a reduction in execution time for matrix-vector multiplication on figure 4.2a. Execution time is divided by more than 50 thanks to data compression. We can see that with a density close to 1%, the gains made by using sparse matrices are very significant. This is explained by the fact that operations involving sparse matrices have complexities that depend on the number of non-zero elements nnz [90], unlike matrix operations whose complexity depends on the size of the matrix n^2.



(a) Execution time in seconds of a SpMV.

(b) Memory comparison in MB.

Figure 4.2: Comparison of the memory required to store the matrix FIDAPM37 [65, 66] and the execution time to perform matrix-vector multiplication as a function of how the matrix is stored.

4.2.4 Overview of sparse matrices in deep learning

Sparse matrices are present at different levels and in different areas of deep learning, specially when dealing with large-scale data and models. First of all, at data level. Many applications use sparse data as input. For example, the adjacency matrix of a graph or the user-item interaction matrices are

typically sparse because users interact with only a small subset of items. In some neural networks, sparse matrices are used to represent convolutional filters [37], especially when dealing with large and high-dimensional data. Finally, sparse matrices can also be the result of network regularization techniques such as dropout [255].

4.2.5 Data distribution and model distribution

Data distribution and model distribution are two types of pattern that define how parallel calculations are to be performed on a cluster of machines interconnected by a network. Data distribution, on one hand, refers to the partitioning of data and its distribution among the different computation nodes. Each node performs the computations linked to the data it has been given, and communicates with the other nodes to merge the partial results and obtain the complete information. Model distribution, on the other hand, refers to the distribution of the various model elements between nodes. In this way, data is moved from node to node until it has passed through the entire model. A simple example would be to associate each layer of a neural network to a computation node. The data will be presented to the first node, which will then communicate the results to the second node, and so on. This serial approach can be represented as a pipeline. It is a division of computational tasks between machines.

On the one hand, data distribution ensures that the system will be able to scale horizontally and handle very large datasets. However, this distribution requires considerable focus on the way data is distributed between nodes, to ensure optimal data access and avoid massive data redistribution between nodes, which would have a significant negative impact on system performance. This requires complex algorithms that can take into account the current distribution of data to ensure continuity of computations throughout the process. Model distribution, on the other hand, involves taking into account the way in which the various components interact with each other, ensuring that each node has all the data it needs to perform its calculations. We also need to take into account the distribution of workloads between nodes and communications between nodes, to avoid having a bottleneck caused by a specific component that limits the entire performance of the system.

Both distributions are crucial and must be taken into account and implemented simultaneously in massively distributed environments, to enable the manipulation of very large data sets and models. In

this case, the result is multi-level distributed computing. This is an advanced technique for meeting the needs of both data and model sizes. First, our model is divided into several separate components, which are then distributed to sub-clusters of machines. Then, within each sub-cluster, we will set up data parallelism between the computation nodes to accelerate the processing of this component. Multi-level parallelism with data and model distribution requires careful planning and a deep understanding of distributed computing tools and environments. It also requires a good comprehension of task and operation training in the model, since the choice of optimal operations to implement depends on the distribution of data between nodes, and consequently depends on previous operations.

4.3 Challenges in sparse matrix operations

Sparse matrix operations in deep learning present several challenges that can impact performance and efficiency. Beyond the basic challenges associated with the use of sparse matrix formats (load balancing, fill-in effect, not well-structured data, etc.), there are a number of additional context-specific challenges that make it quite complex to process sparse matrix operations.

4.3.1 Hardware utilization

Most deep learning models are deployed on accelerators. However, accelerators such as GPUs and Tensor Cores are initially designed for dense matrix operations. Efficiently utilizing specialized hardware for sparse matrix operation is challenging due to the structuring of data, which often leads to non-contiguous memory accesses [113]. Moreover, the SIMD architecture of GPUs is not easy to fully exploit with sparse matrix operations and makes the execution of these operations relatively slow [191].

4.3.2 Algorithmic complexity

Developing efficient algorithms that take maximum advantage of data sparsity is very important to popularize their use. Designing algorithms that manipulate sparse matrices can be difficult in a parallel computing environment. Indeed, the distribution of data and non-zero elements between

nodes to enable efficient load-balancing can greatly complicate data management and greatly increase the number of communications, thus reducing algorithm performance.

In a similar spirit of BLAS detailed in section 2.2.4, a collection of computational routines for sparse matrix operations has been proposed in Sparse BLAS [60]. It supports all elementary sparse linear algebra operations around sparse matrices. This standardized format supports multiple sparse matrix storage formats and is designed to be easily adapted to use with other sparse matrix storage formats. All three levels of BLAS are included, with sparse vector and matrix manipulation within the operation.

4.3.3 Succession of matrix-matrix multiplication

Succession of matrix-matrix multiplication is a very common operation in deep learning models. This is because the matrix resulting from the output of each layer is immediately used as input for the next layer. Moving forward in the model is a succession of matrix-matrix multiplications interspersed with the application of functions on the matrix elements, such as activation functions. It is therefore very important to think about data distribution and load balancing so as to be able to carry out all these multiplications with the least possible redistribution of data. Communications between each matrix multiplication can have a significant impact on the execution time of the operation.

4.4 Matrix-matrix multiplication succession in distributed environment

Our objective is to propose solutions for the correct treatment of large neural network for graphs. GNNs need algorithmic solutions to efficiently assign and process graph data on modern distributed and parallel machines, which are considered with mixed arithmetic and various types of tensor/matrix accelerators.

As we saw earlier in this document in sections 2 and 3, graph is an unstructured data and it is not possible to restructure it without an important lost of information. This is why GNNs are efficient to deal with this type of data. They take into account the structure of graph during the learning process

as well as the features themselves. To integrate the structure in the computations, the GNN models often use the adjacency matrix to integrate the topological information in the model, and are managed in a way that the information is propagated in the graph. GNNs for industrial applications use very large graphs. Storing and manipulating adjacency matrices in the form of a sparse matrix is desirable to have good performance and to limit memory footprint. Handling and exploiting these very large GNNs and their data require heavy computing power, where parallel and distributed machines are involved.

In this context, we propose an automatic solution for the distribution of matrix data, specially designed to minimize communication and to limit the memory footprint. To present this solution, we will first describe the problem and the context in section 4.4.1, then we will detail the functioning of the solution in section 4.4.2 and how the redistribution of data is done in section 4.4.3, and compare our solution with other distribution models in section 4.4.4.

4.4.1 Description of the problem

Programmers can perform multiplication between two matrices in a distributed environment in many different ways. It is mainly the distribution of the data between the nodes that determines the way the computations and the communications are performed. Focusing on a succession of matrix-matrix multiplication forces us to consider how the data is distributed at the end of each multiplication and what communications are needed to restart a matrix multiplication using the previous result matrix. The optimal approach is to have as much data as possible already on the right nodes to restart an operation. This limits the number of communications needed to redistribute the data.

A succession of matrix multiplications is relevant for the execution of a Graph Convolutional Network[152]. The equation 3.1 that we will be rewrite just below is a generalization of how GNNs work, and the computations required to move from one layer to the next. This equation is:

$$\boldsymbol{H}_{l+1} = \phi(\widetilde{\boldsymbol{A}}\boldsymbol{H}_l\boldsymbol{W}_l)$$

where $\boldsymbol{H}_l$ is the embedding matrix representing the l-th graph layer output, $\boldsymbol{W}_l$ is the weight matrix

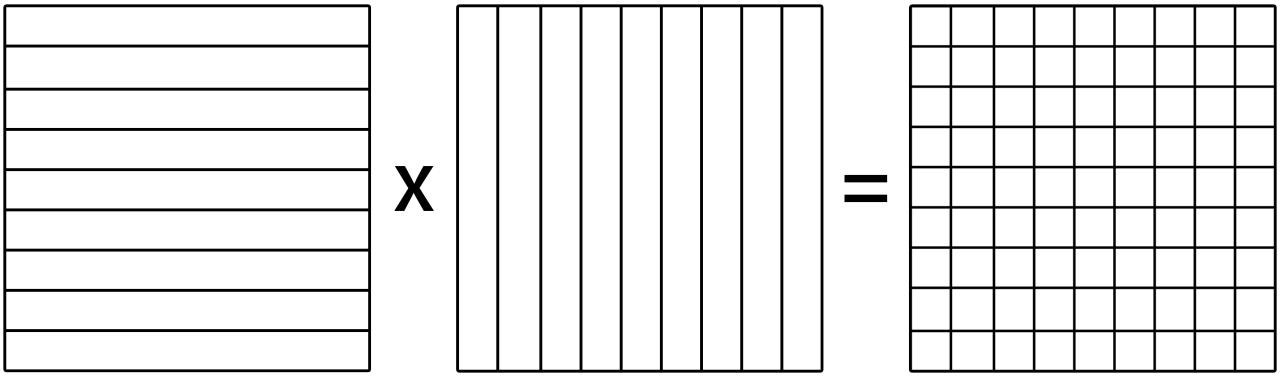


Figure 4.3: Overview of the data distribution with a grid of 9 by 9 nodes.

for the l-th layer and $\widetilde{\boldsymbol{A}}$ is the modified graph adjacency matrix. We have two matrix-matrix multiplications, one between a sparse matrix and a dense matrix at each layer. If the model is deep, the result is a long succession of matrix-matrix multiplications (the activation function is only an operation that must be performed on each element of the result matrix, which can be done independently).

4.4.2 Data distribution

We propose a matrix data distribution model that minimizes communications and requires a relatively small space on each node. To illustrate this, we will consider the following situation: We wish to compute $\boldsymbol{C}$ where $\boldsymbol{C} = \boldsymbol{AB}$ where $\boldsymbol{A}$ and $\boldsymbol{B}$ are square matrices of size n and we have p computation nodes with distributed memory. The first step is to organize the different computational nodes into a 2D grid. The grid that is generated is $\sqrt{p} * \sqrt{p}$ and we denote $p_{x,y}$ with $0 \leq x, y < \sqrt{p}$ the node located on the x row and the y column of the 2D grid.

The distribution of the matrix data will depend on the place of the matrix in the multiplication. For the left term, here $\boldsymbol{A}$, the matrix will be cut in $\sqrt{p}$ block of rows, and each block will be distributed to all the nodes which are on the row associated with this block in the grid. Thus the i-th row block of the matrix $\boldsymbol{A}$ will be distributed on the nodes $p_{i,:}$. The distribution of the right term of the multiplication follows the same logic but with column blocks. The matrix $\boldsymbol{B}$ is split in $\sqrt{p}$ sub-matrix of columns of size $(\frac{n}{\sqrt{p}}, n)$ and each block j is distributed on the nodes $p_{:,j}$. Thus each node $p_{i,j}$ ends up with two blocks: the block i of $\boldsymbol{A}$ and the block j of the matrix $\boldsymbol{B}$. Each node can compute a part of the result matrix from this distribution. The result block will be in our example of size $(\frac{n}{\sqrt{p}}, \frac{n}{\sqrt{p}})$. The

multiplication has been done and the data are now dispatched on the p nodes of the grid. The matrices' splitting is illustrated in the figure 4.3.

Left matrix	*Right matrix*	*Memory space store matrices*	*Memory space store result*	*Communication to get results*	*Communication left*	*Communication right*
Row	Full	$\frac{n^2}{p} + n^2$	$\frac{n^2}{p}$	0	0	$(p-1)n^2$
Col	Row	$\frac{n^2}{p} + \frac{n^2}{p}$	n^2	$p(p-1)n^2$	0	0
Block	Row	$\frac{n^2}{p} + \frac{n^2}{\sqrt{p}}$	$\frac{n^2}{\sqrt{p}}$	$p(\sqrt{p}-1)\frac{n^2}{\sqrt{p}}$	0	0
Grid-Row	Grid-Col	$\frac{n^2}{\sqrt{p}} + \frac{n^2}{\sqrt{p}}$	$\frac{n^2}{p}$	0	$p(\sqrt{p}-1)\frac{n^2}{p}$	$p(\sqrt{p}-1)\frac{n^2}{p}$

Table 4.1: Comparison memory and communication with different distribution formats.

4.4.3 Redistribution for the next multiplication

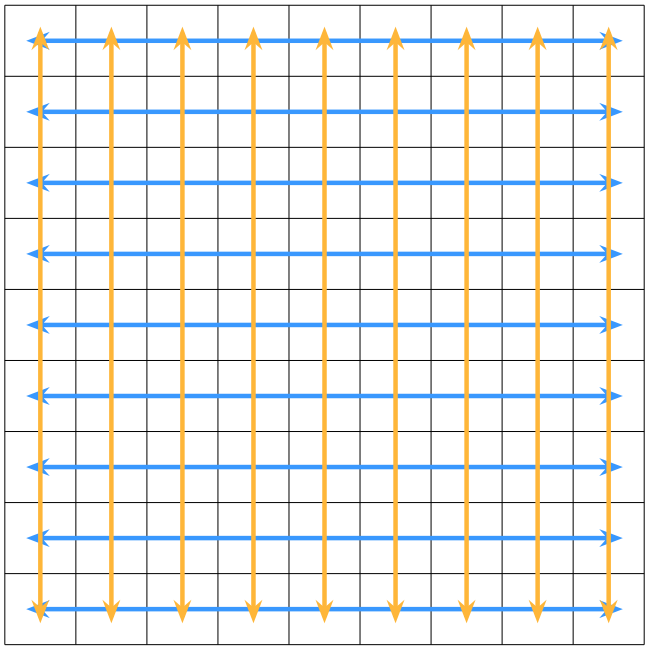


Figure 4.4: Communications between two matrix multiplications in a sequence. The blue and orange arrows indicate respectively the communications if the resulting matrix is used as the left or right term in the next multiplication.

Now that the computation of C has been performed, matrix data are distributed in p blocks on the p nodes of the grid. As illustrated in the figure 4.4, the communications necessary to redistribute the matrix C will depend on its place in the next multiplication. If C will be the left term, each node will send to all the other nodes on the same row of the grid its block of C. By receiving the $\sqrt{p}-1$ other

blocks, each node will be able to build the block row of the $\mathbf{C}$ matrix and perform the next matrix multiplication. In an equivalent way if the matrix $\mathbf{C}$ is the term of the right-hand side of the next multiplication, each node will diffuse its block with all the other nodes which are on the same column. This communication only requires *many-to-many* communications. This communication model is very well suited for 2D grid of cores architecture.

The use of a general pattern to dictate the data distribution in the matrices avoids managing data location. Each node stores the blocks of matrices assigned to it. Moreover, the fact that the matrix is split according to a grid means that sparse formats can be used locally on each matrix block without any repercussions on the communications, which remain of fixed size.

4.4.4 Comparison with other data distribution

In this section, we compare our approach of data distribution as a grid in the table 4.1. We made the comparison with three other distributions. For each method, we have listed the memory space needed to store the matrices for the multiplication and the result matrix, as well as the communication needed to obtain the final values. We also compare the communication needed to distribute the result matrix according to whether it is on the left or the right in the next multiplication. To compare the communications between the methods, we multiplied the number of communications by the size of the communications. A summary of the total memory space and the necessary communications can be found in the table 4.2. Our block approach allows us to limit the memory space to distribute the data while ensuring that the maximum communications are the lowest. Although the Block-Row approach also requires only *many-to-many* communications, our approach requires $\sqrt{p}$ times less communications.

Method	*Memory space*	*Max comm.*	*Type of comm.*
Row-Full	$n^2 + 2\frac{n^2}{p}$	$(p-1)n^2$	All-to-All
Col-Row	$n^2 + 2\frac{n^2}{p}$	$p(p-1)n^2$	All-to-All
Block-Row	$\frac{n^2}{p} + 2\frac{n^2}{\sqrt{p}}$	$p(\sqrt{p}-1)\frac{n^2}{\sqrt{p}}$	Many-to-Many
Grid-Block	$\frac{n^2}{p} + 2\frac{n^2}{\sqrt{p}}$	$p(\sqrt{p}-1)\frac{n^2}{p}$	Many-to-Many

Table 4.2: Comparison memory and communication distribution formats.

4.5 Efficient and scalable approach to build co-occurrence matrix for DNN's embedding layer

4.5.1 How to build co-occurrence matrix?

Let's recall in this section how a co-occurrence matrix is basically built, in order to prepare a smooth understanding on our design of Sparse-Pairwise co-occurrence matrix construction presented in section 4.5.3. We will first provide here the notations with the basic symmetrical dot product approach (a.k.a. matrix product) in both sequential and distributed environments. We will then extend it to sparse matrices with a discussion on storage format and space complexity.

Symmetrical dot product from incidence matrix

Let's first define an incidence matrix before going into the entire symmetrical dot product approach. An incidence matrix, noted $\boldsymbol{V}$, is a representation used to show the connections between two sets of data. In our example in figure 4.5, the incidence matrix is used to show the connections between instances and features in our dataset. Each row of the matrix represents an instance, and each column a feature. We can quickly see from this matrix which data are linked to each other.

Therefore, the co-occurrence matrix $\boldsymbol{C}$ is constructed from this incidence matrix $\boldsymbol{V}$. Based on the associations between instances and individuals, this can be used to determine how often each feature is associated with another feature.

More generally, the construction of the co-occurrence matrix between the n features of a dataset composed of k instances is a level-3 BLAS matrix multiplication. We can build with $\boldsymbol{V}^T \times \boldsymbol{V}$ that

co-occurrence matrix $\boldsymbol{C}$, which represents the *true together* frequencies of elements. The result of this operation is a symmetrical matrix. This operation corresponds to a multiplication between a $n \times k$ and a $k \times n$ matrix, which corresponds to $k \times n^2$ multiplication and $(k-1) \times n^2$ addition. The complexity of this operation as a function of k and n is $\mathcal{O}(k \times n^2)$.

The proportion of non-zero elements in the matrix over the total number of elements in the matrix is called the density of the matrix. The inverse of the density is called the sparsity of the matrix. When the density of non-zero elements in a matrix is sufficiently low, storing only the positions and values of non-zero elements can save both memory and computing power. Low-density matrices are called sparse matrices [90].

Sparse matrices can be used to build the co-occurrence matrix. When the proportion of non-zero values is very low in the k vectors of the dataset, it is possible to consider the incidence matrix $\boldsymbol{V}$ as a sparse matrix to speed up calculations. Exploiting matrix sparsity considerably reduces the computational costs associated with matrix multiplication. However, performing a multiplication between two sparse matrices is a complex and costly operation. Multiplication operations between several sparse matrices are generally avoided and not supported by most sparse matrix libraries. The main reason is the complexity required. Multiplying several sparse matrices in fact involves calculating a prediction of the structure of the non-zero elements of the result matrix, before calculating the non-zero elements of this matrix [45]. Determining the fill-in of an operation between several sparse matrices is beyond the scope of a low-level computational kernel because it requires complicated graph analysis [61]. In general, multiplying two sparse matrices tends to produce a matrix with a higher density. This undermines the benefits of using sparse matrices. The complexity of supporting sparse-sparse operations lies in the number of methods required to cover all possible multiplications between different storage formats of sparse matrices. As the number of storage formats increases, the number of subroutines to be developed also increases. This requires a high workload and makes the process of adding new storage formats a complicated one.

One way of avoiding multiplication between several sparse matrices is to change the order in which matrix operations are performed. For example, suppose we need to compute the following matrix multiplication $\boldsymbol{A} \cdot \boldsymbol{B} \cdot \boldsymbol{C}$ with $\boldsymbol{A}$ and $\boldsymbol{B}$ sparse matrices and $\boldsymbol{C}$ a dense matrix whose dimensions

are consistent for realizing the multiplications. It is simpler to first compute the result of the sparse matrix-dense matrix multiplication $\boldsymbol{B} \cdot \boldsymbol{C}$ to obtain a new intermediate dense matrix $\boldsymbol{T}$. We obtain the final result by once again performing a sparse matrix-dense matrix multiplication $\boldsymbol{A} \cdot \boldsymbol{T}$.

Therefore, we will not discuss the SpGEMM approach in this paper and will consider the sparse approach as being the approach where one of the two matrices is considered to be stored in a sparse storage format. Multiplying a sparse matrix with a dense matrix is a very popular and well-referenced operation. The great advantage of this approach is that the computational complexity depends on the density of the sparse matrix. So, the use of this approach is optimal when the density is close to 0.

In the rest of this paper, we will refer to the dense symmetrical dot product approach when both multiplication matrices are stored in memory in dense storage format. The approach where one of the two matrices is stored in memory and manipulated in a sparse storage format will be called Sparse symmetrical dot product. We will compare both the dense and sparse symmetrical dot product (SDP) approaches in section 4.7.

Distributed dot product

Multiplying two matrices in a distributed environment is well studied. A comparison of different data distributions in terms of computational power, memory and communications costs can be found in the paper [217].

By distributing the left matrix in $\sqrt{p}$ row blocks and the right matrix in $\sqrt{p}$ column blocks, we maximize the load balancing while minimizing the memory space required on each node and limiting communications. This distribution of data and calculations ensures optimal performance efficiency. The computational complexity of such Dense symmetrical dot product approach is $O(k \times \frac{n^2}{p})$ and we need two blocks of size $\frac{k \times n}{\sqrt{p}}$ on each processor. Each processor calculates partial values of the result matrix block. A communication phase is required to obtain the final values of the result matrix elements. Many-to-many communications are needed to process the reduction of these partial results.

Sparse storage format

To facilitate a better understanding of approaches that deal with sparse matrices, we will be using ELLPACK [237] as the sparse storage format in our examples. This format is easy to visualize and comprehend while also effectively demonstrating the benefits of compressing matrix data. It should be noted that depending on the characteristics and requirements of the dataset, other sparse matrix storage formats can be employed in place of ELLPACK. The choice of format is completely free and flexible.

Storing low-density matrices in a sparse storage format saves a lot of memory space. If the matrix can be stored in memory on each node, then it is very interesting to consider duplicating the sparse matrix on each node. In fact, one of the data distribution options allows you to obtain blocks of the result matrix on each node without any additional communication. The result matrix will then be distributed to the different nodes. Duplicating the sparse matrix on each node and splitting the other dense matrix into p blocks avoids the communication phase involved in the reduction of partial results with the $\sqrt{p}$ block approach described previously. This data distribution is more memory-intensive on each node but eliminates any need for communication to obtain the final results.

I like you.	$x_1 = \text{I}$
You like dogs.	$x_2 = \text{you}$
I don't like dogs.	$x_3 = \text{don't}$
———————	$x_4 = \text{like}$
	$x_5 = \text{dogs}$

(a) Example of corpus $\boldsymbol{D}$

$$\begin{matrix} x_1 & x_2 & x_3 & x_4 & x_5 \end{matrix}$$
$$\begin{bmatrix} 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 \end{bmatrix}$$

(b) Incidence matrix $\boldsymbol{V}$

$$\begin{bmatrix} 2 & 1 & 1 & 2 & 1 \\ 1 & 2 & 0 & 2 & 1 \\ 1 & 0 & 1 & 1 & 1 \\ 2 & 2 & 1 & 3 & 2 \\ 1 & 1 & 1 & 2 & 2 \end{bmatrix}$$

(c) Co-occurrence matrix $\boldsymbol{C}$

Figure 4.5: Example of corpus of words with (b) the incidence matrix and (c) the co-occurrence matrix associated with the (a) distribution.

4.5.2 Pairwise approach

The pairwise approach is based on the following idea: the $\boldsymbol{C}_{i,j}$ element of the co-occurrence matrix represents the number of times that features i and j have been simultaneously active for instance. In other words, in a dataset composed of k elements, the co-occurrence matrix allows us to visualize the number of times the features were simultaneously present on an instance. When $i = j$, the co-

occurrence matrix tells us how many times the feature has been associated together on one instance. Therefore, it is possible to construct the co-occurrence matrix by forming the set of feature pairs (i, j) among all dataset instances. In concrete terms, it consists in finding all combinations of pairs of non-zero values within each vector of the dataset.

Let's take as an example the dataset proposed in figure 4.5a. The first instance (e.g., "I like you") is composed of the features x_1, x_2 and x_4. We should add 1 to the three elements on the diagonal of the co-occurrence matrix $\boldsymbol{C}_{x_1,x_1}, \boldsymbol{C}_{x_2,x_2}, \boldsymbol{C}_{x_4,x_4}$, then add 1 for each possible pair with $i \neq j$. We have 6 possible pairs which are as follows: (x_1, x_2), (x_1, x_4), (x_2, x_4), (x_2, x_1), (x_4, x_1), (x_4, x_2). We, therefore, add 1 to all the elements of the co-occurrence matrix with these indices. Do the same with the other sentences in the dataset to obtain the co-occurrence matrix $\boldsymbol{C}$. A visual example of the Pairwise approach is shown in figure 4.6. In this example, a color is associated with each element of each vector, and underneath are the possible pairs of elements. We have highlighted the pairs that can be constructed with the non-zero elements of each vector. The result is a matrix for each vector, which is then reduced to produce the final co-occurrence matrix.

Algorithm 6 Build the co-occurrence matrix from the Pairwise approach.

Input: $\boldsymbol{D}$ the dataset (list of the k input boolean vectors of size n)
Output: The co-occurrence matrix $\boldsymbol{M}$ of size $n \times n$

```
initialize all elements of M to 0
for each non-zero element i of D do
    for each non-zero element j in the same vector than i do
        M_{i,j} ← M_{i,j} + 1
    end for
end for
return M
```

Note here that it is possible to limit the search for pairs with $i \leq j$. This makes it possible to construct only the upper triangle of the co-occurrence matrix. If we name the resulting triangular matrix $\boldsymbol{T_C}$, we obtain $\boldsymbol{C} = \boldsymbol{T_C} + \boldsymbol{T_C}^T - \text{diag}(\boldsymbol{T_C})$.

Algorithm 6 represents the pairwise method. Although on theory this is a very interesting approach, since it takes advantage of the fact that the data set is Boolean, it generally gives less interesting performance. Finding all possible pairs of elements in a vector means finding the non-zero elements in the vector, then for each of these values, finding the other non-zero elements in the same

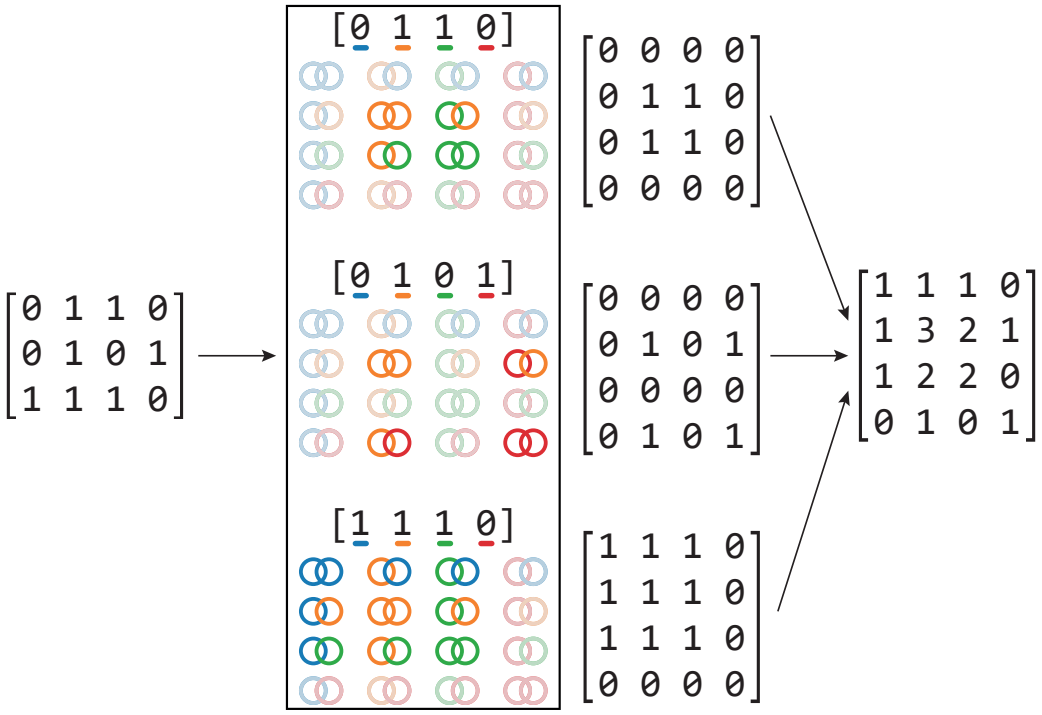


Figure 4.6: Example of processing with the Pairwise method to construct the co-occurrence matrix.

vector. Still in the algorithm 6, the `for` loop in the lines 2 and 3 are actually two nested loops whose execution depends on the result of a condition. For each element in the vector, test the element value. If the result is yes, continue in the next loop; otherwise, test the next element. The problem is that `if` statements tend to break the pipeline that runs within CPUs on modern architectures [212]. We will look at this in more detail in section 4.7.

The sparsity of the dataset has an impact on the performance of this method: it will define the number of times we enter the first loop `for` (line 2). The second loop, `for`, will run through all elements, regardless of sparsity. In the next section, we will take a look at an approach derived from the pairwise approach that takes greater advantage of data sparsity.

4.5.3 Sparse-Pairwise approach

We have seen in the previous sub-section that the pairwise approach takes advantage of the fact that the dataset is composed only of boolean elements, and the symmetrical dot product approach takes advantage of the fact that sparsity is high to speed up computations thanks to sparse linear algebra.

Algorithm 7 Sequential algorithm of Sparse-Pairwise approach.

Input: $\boldsymbol{D}$ the dataset (list of the k input boolean vectors of size n)
Output: The co-occurrence matrix $\boldsymbol{M}$ of size $n \times n$

initialize all elements of $\boldsymbol{M}$ to 0
$\boldsymbol{A} \leftarrow$ build the ELLPACK sparse matrix index from $\boldsymbol{D}$
for each of the k vectors in $\boldsymbol{D}$ **do**
  for each non-zero element i in k **do**
    for each element j in $\boldsymbol{A}_{k,:}$ **do**
      $\boldsymbol{M}_{i,j} \leftarrow \boldsymbol{M}_{i,j} + 1$
    end for
  end for
end for
return $\boldsymbol{M}$

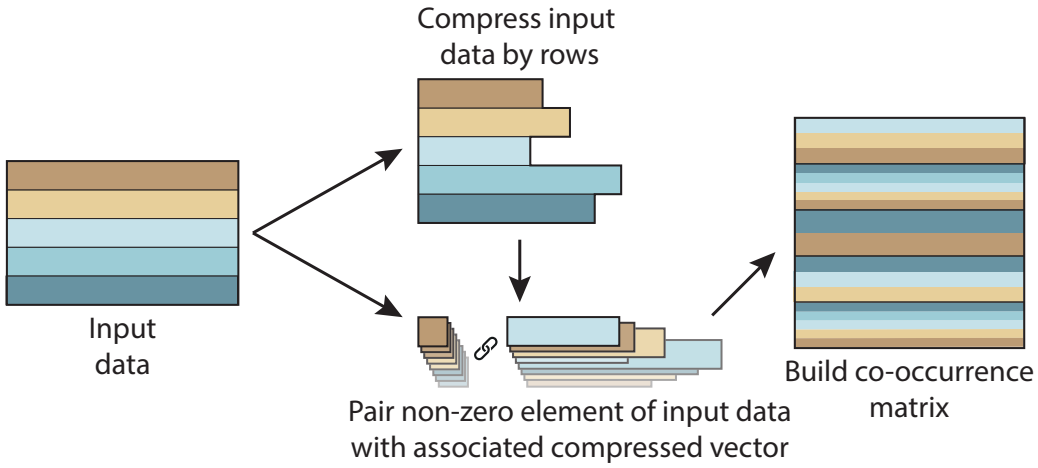


Figure 4.7: Overview of the Sparse-Pairwise approach.

In this part, we propose an approach that combines this approach with the dot product approach to speed up the construction of the co-occurrence matrix with both sparse linear algebra and boolean arithmetic. Figure 4.7 illustrates the main points of this approach to build co-occurrence matrix.

The limitation of the pairwise approach is that each time a non-zero element is found, the set of other non-zero elements in the feature vector must be found. Instead of traversing the entire vector when a non-zero value is found in the dataset, the Sparse-Pairwise approach consists of an initial scan the dataset to prepare the index list of non-zero values. By doing this, each time a non-zero value is found, we can immediately refer to the index list to find the pairs in which this non-zero element will be found. This quickly completes the list of pairs, without having to go through the rest of the vector.

Taking as an example the dataset in figure 4.5, compressing the incidence matrix in ELLPACK

format gives the following index matrix:

$$\boldsymbol{E}_V = \begin{bmatrix} 1 & 2 & 4 & - \\ 2 & 4 & 5 & - \\ 1 & 3 & 4 & 5 \end{bmatrix}$$

Then, for each vector i of dataset instances, we will increment all the elements of the co-occurrence matrix whose coordinates are the index pairs stored in row i of the above matrix.

For the first dataset instance, the indices in line 1 above are $s_1 = \{x_1, x_2, x_4\}$, so the set of pairs is (x_1, x_1), (x_1, x_2), (x_1, x_4), (x_2, x_1), (x_2, x_2), (x_2, x_4), (x_4, x_1), (x_4, x_2), (x_4, x_4). We then add 1 to all the elements of the co-occurrence matrix with these coordinates. This operation is repeated with the other vectors in the matrix to obtain the co-occurrence matrix for the dataset.

With this approach, we take advantage both of the pairwise search made possible by the fact that dataset elements are binary values, and of the sparse storage format made possible by the data's sparsity. The algorithm 7 represents the sequential version of this approach, and we will discuss its deployment in a massively distributed environment in the following part.

4.5.4 Deploying in a massively distributed environment

Datasets are generally very large, and to be able to build the co-occurrence matrix on very large datasets, it is essential to have an algorithm adapted to a distributed computing environment. In this section, we will compare the two previous implementations and see what possible optimizations we can take advantage of with distribution. Let's note p the number of processors on which calculations will be distributed. For communications purposes, we assume that these p nodes are linked by a network and have distributed memory.

Given the general size and density of large DNN datasets, it has been assumed that every node possesses ample memory space to replicate the sparse matrix, as elaborated in section 4.5.1. Duplicating the data to avoid communication seems to be the most advantageous approach for data distribution while dealing with sparse matrices. In the case where the sparse matrix is too large to be stored as such on each node, dividing the sparse matrix into several blocks of size $\sqrt{p}$ is also a plausible method

for data distribution.

Pairwise approach

The naive pairwise search approach distribution is to distribute `for` loops between the nodes. This approach is not very efficient because building the final co-occurrence matrix will create a lot of communication for the reduction. A more interesting approach is to construct the co-occurrence matrix by blocks of rows. This approach allows us to play with the intervals covered by the `for` loops. Let b be the number of blocks into which you want to divide the matrix $\boldsymbol{C}$. The i block of the matrix represents the rows $[\frac{n}{b} \times i, \frac{n}{b} \times (i+1)[$. Since the matrix is symmetrical, adding data to the $\boldsymbol{C}_{i,j}$ element will also add data to the $\boldsymbol{C}_{j,i}$ element. So we can limit the range of loops by checking that either $i \in [\frac{n}{b} \times i, \frac{n}{b} \times (i+1)[$ or $j \in [\frac{n}{b} \times i, \frac{n}{b} \times (i+1)[$. In addition, since $i \leq j$, we can also limit the interval of the first loop `for` to $[0, \frac{n}{b} \times (i+1)]$. When the element visited by the first loop is non-zero, it checks whether the element is in the interval. If yes, the second loop must traverse the rest of the vector. If not, then the interval of the second loop will be limited to the interval $[\frac{n}{b} \times i, \frac{n}{b} \times (i+1)[$.

Implementing the concept of the Sparse-Pairwise approach in a distributed environment is a challenging task. Indeed, distributing the different index lists of non-zero values of each instance will effectively distribute the computational power need, but each node will build a partial result of the entire co-occurrence matrix. Allreduce communications must be made with a length of n^2 values. This scenario is unthinkable with very large datasets, given the communications size and the associated cost.

To be able to eliminate communications, each node must build a block of the final result of the co-occurrence matrix independently of the other nodes. This would result in the co-occurrence matrix being distributed across the different nodes, with blocks of similar size.

Sparse-Pairwise adapted from the sparse symmetrical dot product

The first approach is to use the same data and computation distribution of the sparse symmetrical dot product approach. The $\boldsymbol{E}$ matrix representing the list of indices is duplicated on each calculation node. Each node then calculates a block of $\frac{n}{p}$ rows of the co-occurrence matrix, by scanning each vector in

Approach	*Computation complexity*	*Memory space to store input*
Dense symmetrical dot product	$\mathcal{O}(k \times \frac{n^2}{p})$	$2 \times \frac{k \times n}{\sqrt{p}}$
Sparse symmetrical dot product	$\mathcal{O}(d \times k \times \frac{n^2}{p})$	$d \times n \times k + \frac{k \times n}{p}$
Pairwise	$\mathcal{O}(d \times k \times \frac{n^2}{p})$	$k \times n$
Sparse-Pairwise	$\mathcal{O}(d^2 \times k \times \frac{n^2}{p})$	$d \times n \times k + \frac{k \times n}{p}$
Sparse-Pairwise (Save memory)	$\mathcal{O}(d^2 \times k \times \frac{n^2}{p})$	$(1 + \frac{1}{p}) \times d \times k \times n$

Table 4.3: Comparison of required computation power and memory for approach in a distributed environment.

the dataset for non-zero elements. When a non-zero value is found, we update the matrix as explained in section 4.5.3 with the indices of the $\boldsymbol{E}$ matrix. This approach requires no additional computation. This is the approach we will be deploying when memory constraints are not the priority. We will refer to this approach as the standard Sparse-Pairwise approach in the remainder of this paper.

Sparse-Pairwise approach to save memory

Storing dense blocks of vectors for scanning may require sparing memory to store the entire dataset when the dataset is large. This is why we propose an approach that uses the Sparse-Pairwise principle to limit the memory space required. The aim is to transform all input data into sparse formats. This reduces the amount of memory required to store the input data and adds to the cost of transforming the data.

The principle of this approach is similar to the first, except that instead of dispatching the vectors to the different nodes, we first calculate the columns' compressed matrix, then dispatch this compressed matrix and use the indices in this matrix. It is impossible here to use the already calculated rows' compressed matrix, as it gives no information on the position of the indices to be taken into account when creating a block of the co-occurrence matrix. Consequently, searching for the values included in the processing interval requires going through the entire compressed matrix, reducing the interest in this approach in a distributed environment. Scattering the matrix ensures that each node immediately has the set of non-zero values it needs to find in order to update its result block in the matrix.

However, using only compressed matrices requires more computational power than the standard Sparse-Pairwise approach. To build the columns' compressed matrix, we need to go through the blocks of vectors in the dataset and then build the matrix. The columns' compressed matrix requires more computational power to build than the standard Sparse-Pairwise approach for simply traversing the dense blocks.

This approach is very interesting for processing very large datasets on machines with limited RAM. The saving in terms of memory space will depend on the sparsity of the data. All the input data used to build the co-occurrence matrix is compressed. This approach will only be used when RAM memory cannot store all the information required to build the co-occurrence matrix with the standard Sparse-Pairwise approach. In the next section, we will examine the theoretical comparison of the different approaches with a cost analysis.

4.6 A priori analysis

This section will discuss the theoretical performance of the approaches we discussed in the previous section. To obtain a theoretical approximation of costs, we have used the BSP approach [34].

We have used approaches that minimize communication. Since this is a very costly step, we have favored approaches that allow us to obtain a block of my result matrix on each node without communication. The only communication we have is the distribution of data among the calculations. The cost of this communication phase depends on where and how the data is stored, and on the implementation. We will not be considering the communication costs of initial data distribution and splitting.

The table 4.3 compares the different approaches regarding computational complexity, memory and communication. Complexities are given as a function of n and k, the dimensions of the dataset, the number of processors p and the density of the dataset noted d. d is between 0 and 1 and represents the ratio between the number of non-zero values and the total number of elements in the matrix.

When sparsity starts to become significant, the most interesting approach from a memory perspective is the save memory Sparse-Pairwise approach. This is the only approach where the total memory

space required for input values is directly related to the matrix density. This means that if the density is very low, the storage space required to store the data will be low. However, from a computational point of view, compressing data by both rows and columns is computationally more demanding than the Sparse-Pairwise approach adapted from the classical matrix approach. If there is a need to save even more memory, it is possible to compress the incidence matrix in SGP format [219], a compression pattern that lets you quickly toggle between row and column compression in exchange for a certain additional computation.

Regarding the computational complexity required to build the co-occurrence matrix, the two Sparse-Pairwise approaches are equivalent in complexity. The dense symmetrical dot product approach is the only one where the complexity does not depend on the density d. Sparse-Pairwise approaches have smaller complexities than the sparse symmetrical dot product and Pairwise approaches. This is due to the fact that $d \in [0, 1]$ and therefore $d^2 \leq d$. While $d < 1$, our proposal Sparse-Pairwise approach is the most interesting in terms of computational complexity. In the next section, we will verify these complexities in practice, which presents our experiments.

Name	*instances*	*Features*	*nnz*	*Sparsity*
Anonymous MS Web [29]	37 711	294	999 974	99.11%
Criteo	200 000	206	10 555 469	74.3799%
Kasandr [251]	2 158 860	291 486	15 844 718	99,9974%

Table 4.4: Datasets overview.

4.7 Experimental results

To validate our cost analysis and check the performance of our Sparse-Pairwise approach, we experimented with implementing the 4 approaches described above in C++ and with MPI. In section 4.7.1, we will be describing our experimental environment. In 4.7.2 section, we will use a dataset generator to independently vary different parameters to see how the different approaches perform. Finally, in 4.7.3 section, we will look at the performance of the different approaches with various datasets from real-world applications.

4.7.1 Experimentation environment and datasets

Our working environment is as follows: we have at our disposal 25 nodes comprising 2 Intel Xeon Gold 6230 20 cores @ 2.1 GHz (Cascade Lake). This enabled us to distribute calculations over a maximum of 1000 cores. Each compute node has a RAM capacity of 192GB. The Operating System is CentOS 7.9.2009 and the network technology is an Intel Omni-Path Architecture network 100 Gbit/s. Our disk storage capacity is 500 GB. It is a Spectrum Scale GPFS parallel file system that allows 9 GB/s input/output rate.

To be able to test co-occurrence matrix construction approaches accurately and under different conditions, we have developed a Boolean dataset generator. The algorithm 8 shows how we can build a dataset with a defined size and sparsity. Parameters k and n are respectively the number of instances and the number of features we want in the dataset. After generating an empty dataset of the desired size on line 1, we use the parameter d to fill our dataset according to the expected density. By drawing a random number between 0 and 1 for each element in the dataset, we add non-zero elements to the dataset with a probability of d (line 3-6). The value of parameter d is included in the interval $[0, 1]$.

Algorithm 8 Dataset generator.

Input: k the number of elements in the dataset, n the numbers of dataset features, d the expected density of the dataset
Output: A dataset $\boldsymbol{D}$

```
1: D ← create k vectors of size n and initialize all elements to 0
2: for each element e in D do
3:     r ← Random number in [0, 1]
4:     if r > d then
5:         Change the value of e to 1
6:     end if
7: end for
8: return D
```

We also chose to use three datasets for our experiments. An overview of the characteristics of these datasets is available in table 4.4. We selected the Anonymous MS Web dataset for its low density. In contrast, Criteo is a relatively high-density dataset. Finally, the last dataset, Kasandr, will enable us to see the scalability of the approaches thanks to its large size.

4.7.2 Efficiency and scalability

In order to test co-occurrence methods, we used the generator introduced in 4.7.1 to vary the parameters one by one and observe the resulting variations in execution time to construct the co-occurrence matrix. This will also enable us to progressively verify that the results are consistent with the complexity analysis performed in section 4 and to see the prevalence of Sparse-Pairwise approach relative to other approaches.

Memory complexity analysis

Approach	*Theoretical memory requirements*	*implementation memory allocation*
Dense SDP	12.649×10^5	12.652×10^5
Sparse SDP	2.20×10^5	2.57×10^5
Pairwise	200.0×10^5	200.0×10^5
Sparse-Pairwise	2.20×10^5	2.57×10^5

Table 4.5: Memory complexity for each approach implementation.

The memory complexity of our implementation is shown in the table 4.5. The theoretical values according to the table 4.3 are also given for comparison. In this example, the environment parameters have been set as follows: $n = 100000$, $k = 200$, $p = 1000$ and $d = 1\%$. As the memory required to store the co-occurrence matrix is the same for all approaches, only the memory required to store the input data has been taken into account in this table.

For the Sparse SDP and Sparse-Pairwise approaches, we observe a fairly large difference with the theoretical value. This is due to our sparse matrix storage format. Using the ELLPACK format, we initialize an array larger than the number of non-zero values when the distribution of non-zero values is not perfectly distributed between the rows. The slight difference in complexity of the Dense SDP approach is due to the fact that $\sqrt{n}$ is rounded up to the nearest integer during load balancing.

For each method implementation, we used a vector of size p to store the index of the first row of the block associated with each matrix. This buffer vector is used to distribute the data to ensure good

load balancing. However, the additional memory cost required to store this information is very low. We can see that approaches using sparsity require the least memory space. We also observe that the pairwise approach is very memory-intensive, making it difficult to use with very large datasets.

Density d

Figure 4.8 shows the execution times of the different approaches as a function of dataset density. The figure shows that all the approaches vary as a function of density except the dense symmetrical dot product approach. The results have been deliberately zoomed in on the lowest curves, removing the Pairwise approach, whose results are very high when the density exceeds 0.2.

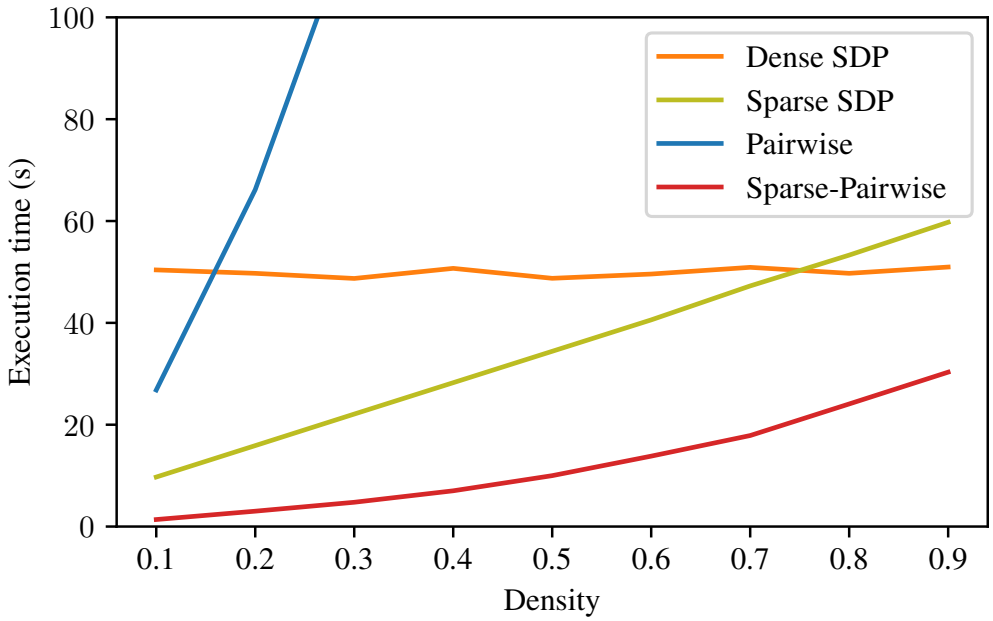


Figure 4.8: Execution time comparison between the different pairwise and the matrix approaches to build the co-occurrence matrix in the function of the sparsity.

The Sparse symmetrical dot product approach performs better than the dense one when the density is less than 0.7. Similarly, the Pairwise approach performs better when the density is less than 0.2. We observe that execution times follow a curve in a similar way to the cost analysis predictions. We observe that execution times increase linearly as a function of density with the sparse symmetrical dot product approach. The execution times for the Sparse-Pairwise approach follow a parabolic pattern, confirming the squared complexity according to density. The Sparse-Pairwise approach achieves the

fastest execution times regardless of density in the $[0.1, 0.9]$ range.

To take a more detailed study of the performance of the approaches at low density, we experimented as function of density in the interval $[0, 0.1]$. The results are given in figure 4.9. The results in this figure show that even with a low density, the Sparse-Pairwise approach is the most interesting in terms of execution time. The Pairwise approach has about the same performance as the sparse symmetrical dot product approach when the density is 1%.

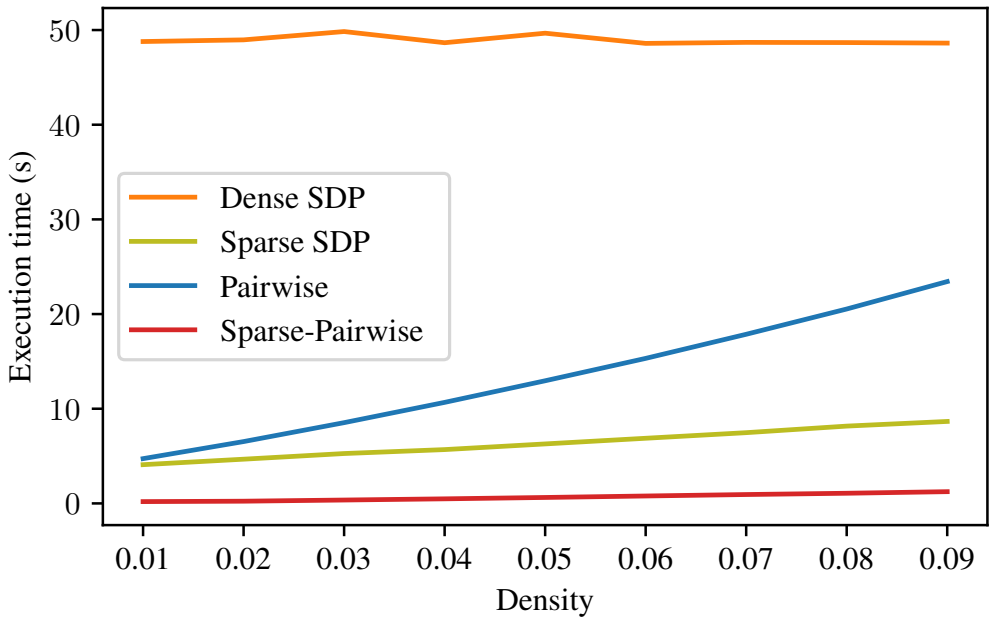


Figure 4.9: Execution time comparison between the different approaches to build the co-occurrence matrix in function of the density. Zoom in the interval $[0, 0.1]$.

The time required to build the co-occurrence matrix becomes negligible with the Sparse-Pairwise approach when the density is very low. With a density of 1%, the execution time to build the matrix is 0.19 seconds, while building the sparse matrix from the dataset takes 3.09 seconds.

Number of instances k

For the k parameter, which corresponds to the number of individuals in the dataset, experiments have shown that the impact on execution time is linear. This fully verifies the cost analysis carried out in section 4.6. Doubling k means doubling the execution time. The difference between the two approaches is the value of the linearity coefficient. In table 4.6, we have calculated the coefficient of

Approach	$k = 200$	$k = 2000$	*Coefficient*
Dense SDP	4.79802	48.7221	2.440×10^{-2}
Sparse SDP	2.21438	22.09	1.104×10^{-2}
Pairwise	12.2988	119.868	5.976×10^{-2}
Sparse-Pairwise	0.486258	4.76849	2.379×10^{-3}

Table 4.6: Execution time to build the co-occurrence matrix with different approaches for two values of k. The coefficient represents the coefficients of the linear functions of execution time.

linearity for each method between two measurements with $k = 200$ and $k = 2000$. For each approach, this coefficient represents the additional time required when k is incremented by 1. The results were obtained by setting the parameters $p = 1000$, $n = 50000$ and density at 30 %. It can be seen that k has no impact on the differences in performance between the approaches. Whatever the value of k, we observe that given the experimental conditions of the table 4.6, the Sparse-Pairwise approach is 25 times faster than the Pairwise approach, 10 times faster than the Dense symmetrical dot product approach and also 4.6 times faster than the Sparse symmetrical dot product approach.

Number of features n

In figure 4.10, we have scaled the parameter n by setting the other variables to $k = 500$, $p = 1000$ and fixing the density at 10 %. The figure shows execution times for n between 10000 and 100000. We can see that all the different approaches have execution times that follow a curved trajectory with an increase of the value of n increases. The differences are in the second-degree coefficients associated to each curve. We can see that the slope of the curve is very slight for the Sparse-Pairwise approach compared to the other approaches. In this configuration, the Sparse-Pairwise approach offers the best performance, whatever the value of n.

We have shown that the performance of the Sparse-Pairwise approach is the most interesting whatever the values of k, n and matrix density. The Sparse-Pairwise approach is scalable and well suited over a wide range of n and k values. The efficiency of the Sparse-Pairwise approach is improved even further with very sparse datasets, but it is still worth using regardless of density. In addition, we have verified that the experiments match the theoretical performance in terms of computational

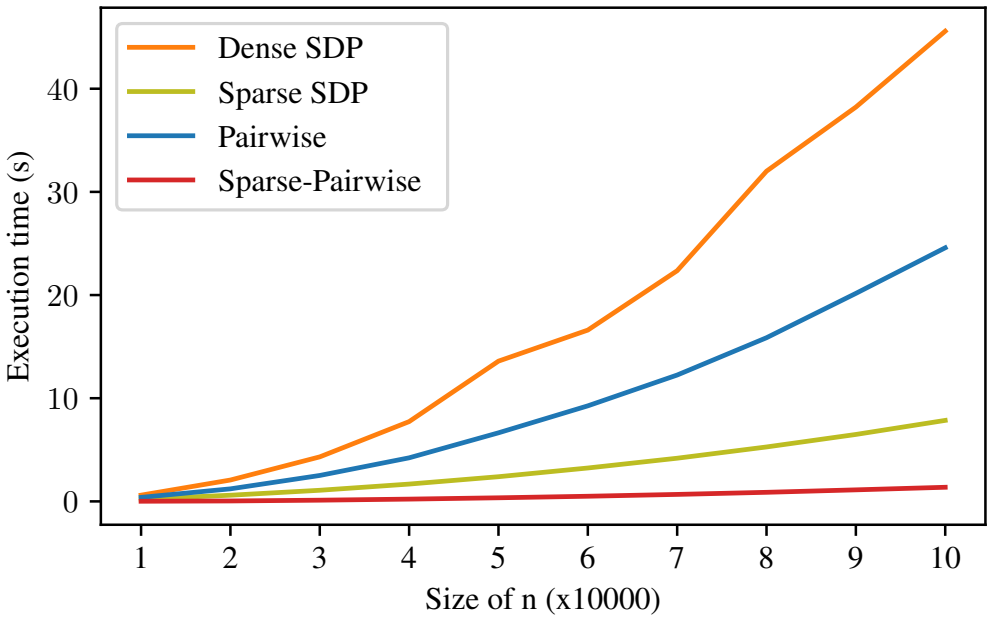


Figure 4.10: Execution time for different co-occurrence matrix building approaches in the function of the size of n.

complexity obtained in the previous section.

Number of processors p

Figure 4.11 shows execution times as a function of the number of processors p. The matrix size is set to $k = 100$ and $n = 100000$ and the sparsity is fixed to 20%. We can see in this figure that the different methods for building the co-occurrence matrix all have excellent scalability. The Sparse-Pairwise approach has an efficiency of 96.9% with 1000 nodes compared with the execution time with 100 nodes, which is very good scalability. The efficiencies of the other methods are quite similar, although the sparse SDP approach achieves 87.8%. Which makes this approach the least interesting in terms of scalability.

The results for the study of weak scalability are shown in figure 4.12. In this figure, we varied n and p linearly, so that each processor always has a block of the input dataset of the same size. In other words, the problem size is fixed for each processor. In the experiments shown in figure 4.12, we set $k = 100$ and $n = 100 \times p$, so the size of the block distributed to each compute node is 100×100.

That execution time increases linearly as a function of p and n. When p (and n) are doubled,

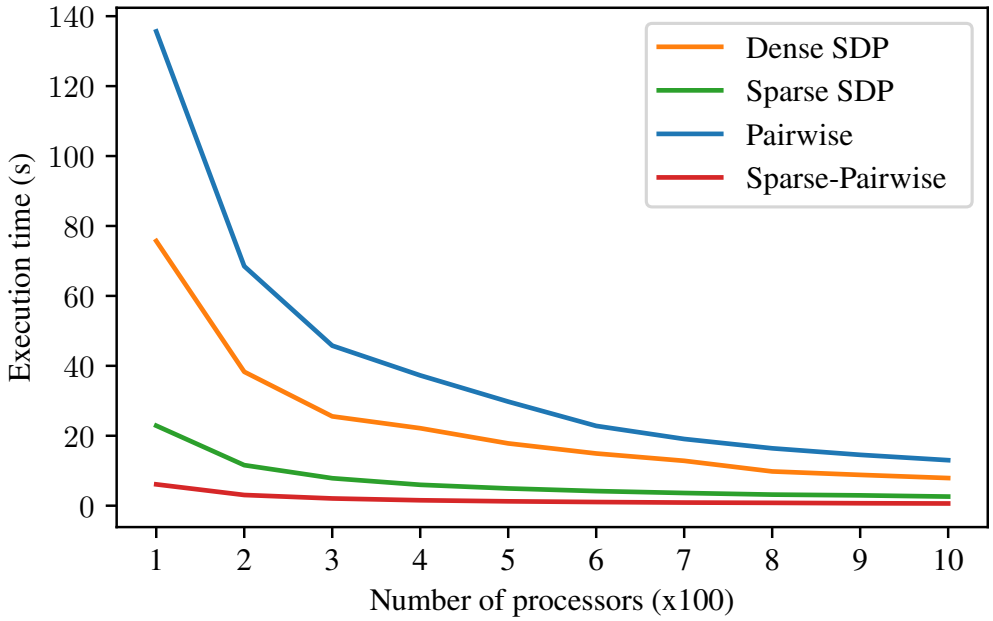


Figure 4.11: Strong scalability: Execution time for different co-occurrence matrix building approaches in the function of the number of processors p.

execution time is also doubled. This verifies the computation complexity given in table 4.3. If the complexity of $\frac{n^2}{p} = \alpha$, then $\frac{(2n)^2}{2p} = 2\frac{n^2}{p} = 2\alpha$. All else being equal, we efficiently expect execution times to double when n and p are doubled. The experiments in figure 4.12 were performed with a density of 5%. We have the same performances associated with this density as in figure 4.9.

4.7.3 Validation with real-case datasets

Now that we have demonstrated the efficiency of our method, we need to show that it is also working on real-world datasets. Consequently, we will use the three datasets presented in table 4.4. We will apply the different co-occurrence matrix building approaches to these datasets and compare performance.

The execution times for building the co-occurrence matrix for each dataset with each approach are printed in table 4.7. The performances obtained highlight that the Sparse-Pairwise approach builds the co-occurrence matrix fastest with all datasets. We can see that the performance of the Sparse-Pairwise approach is just over 4 times better than the Sparse symmetrical dot product approach with the Criteo dataset and up to over 34 times faster with the Kasandr dataset. This shows that the lower the density of

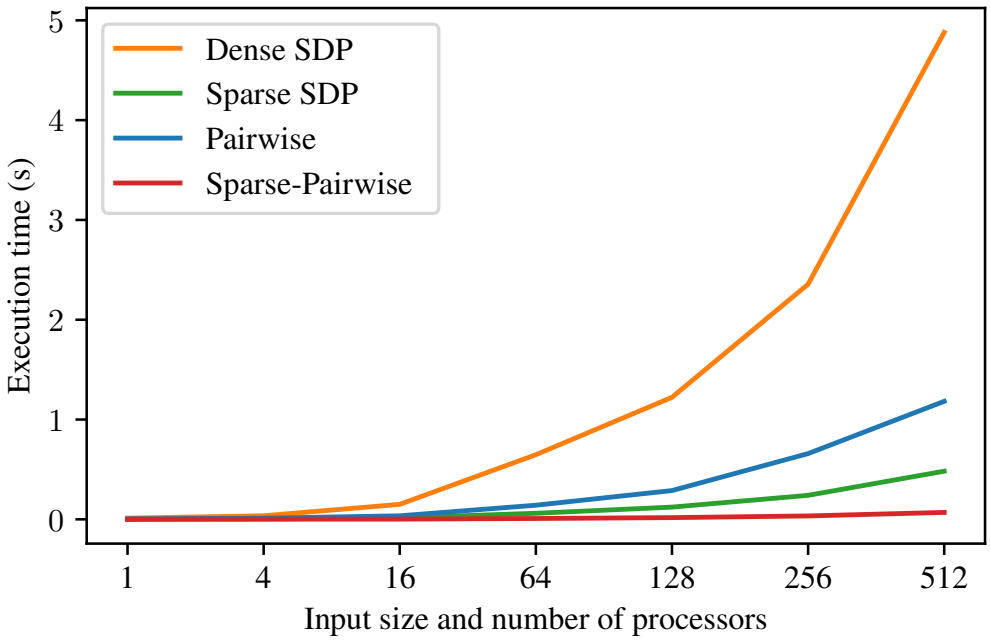


Figure 4.12: Weak scalability: Execution time for different co-occurrence matrix building approaches with a linear modification of n and p.

the dataset, the more effective the Sparse-Pairwise approach. Thanks to the sparse storage formats, our approach also takes advantage of the limited memory required to store matrices. It makes it possible to work with large datasets like Kasandr, where memory space is insufficient to store matrices densely.

	Anonymous MS Web	*Criteo*	*Kasandr*
Dense SDP	3.10211	80.9875	OOM
Sparse SDP	0.349944	28.7367	80.5154
Pairwise	0.460221	170.988	OOM
Sparse-Pairwise	0.0218032	6.77173	2.33458

Table 4.7: Execution time in seconds to build the co-occurrence matrix with different approaches. These results are obtained with $p = 1000$. The execution times take into account the time required to build sparse matrices from dataset data, if necessary.

The results obtained correspond to the performance observed with the dataset generator. The Sparse-Pairwise approach significantly reduces the execution time required to build the co-occurrence matrix. The greater the sparsity of the dataset, the greater the performance gains. The results obtained with Kasandr allow us to justify the scalability of the Sparse-Pairwise approach with very large

datasets.

4.8 Conclusion

We have proposed in this chapter solutions to improve the manipulation of sparse matrices in deep learning applications in a distributed environment. Our study, published in an international conference [217], investigates the successive multiplications between multiple sparse and dense matrices. This research enhances the ability to manage such operations for deep learning tasks, particularly in GNNs. Our complexity study ensures a fair distribution of both workload and communications between the different compute nodes. By avoiding all-to-all communications for the redistribution of data for the next multiplication, we ensure to limit network overload and favor local communications, which is particularly well suited to 2D node grids. Our method for building the co-occurrence matrix by taking advantage of both the data sparsity and the data structure enables us to obtain this matrix more quickly, taking full advantage of the distributed environment. Using a dataset generator, we demonstrated the benefits of our method as a function of machine and dataset parameters, and verified the cost analysis we had performed previously. Experiments on real-world datasets validated the results obtained with the generator and the performance of our method. This study, method and experiments results were published in an international conference [216].

Chapter 5

Spectral Based Embedding Generalization

5.1 Introduction

Dimensionality reduction is a central problem in large deep learning models, as it helps to simplify complex datasets while preserving their essential features. Many dimensionality reduction techniques concern spectral computation based on dominant eigenpairs computation. The dominant eigenvectors allow extracting the most relevant information of the matrix, which represents large dataset. These eigenvectors provide the dataset's principal axes, representing the maximum variance and minimum error. Therefore, it is imperative that the methods used to compute the dominant eigenpairs are both robust and efficient, ensuring reliable and accurate results even with large, sparse and complex datasets.

In this chapter, we propose a high-performance implementation of an innovative technique to reduce the dimensionality of the embedding layer of a deep neural network. This technique is based on a variant of the most robust existing methods for computing a few numbers of dominant eigenpairs of a large sparse matrix. This variant allows the extraction of eigen-information of interest from a dataset even when the eigenvalues are highly clustered. By focusing on the dominant eigenvectors, we ensure that the most meaningful information from the data is preserved, facilitating more accurate and efficient training in deep neural networks. Moreover, the proposed technique exhibits intrinsic parallel characteristics, making it particularly suitable for implementation for high-performance computing systems.

We show the effectiveness of the proposed dimensionality reduction technique through its parallel implementation on a high-performance cluster architecture.

Furthermore, we will see that using the proposed dimensionality reduction technique in the embedding layer training provides results with almost the same level of accuracy as without dimensionality reduction. Moreover, this technique significantly reduces the time and space complexity required to reach a solution. Our experience with different types of datasets shows that the proposed dimensionality reduction technique is both fast and reliable while maintaining good data quality.

5.2 Importance of embedding initialization

We have already proposed a definition of embedding in section 2.3.3. Here we refer to the challenge of reducing the complexity of non-numerical or unstructured data or objects by providing a dense numerical representation suitable for deep learning models. An embedding table is used to move from the initial data representation to the dense representation. This is a table of parameters that can be tuned. Mathematically, the embedding table can take the form of a matrix which, by means of a matrix-vector multiplication from the original representation, yields the dense representation in smaller dimensions. Figure 5.1 simplifies the use of the embedding table.

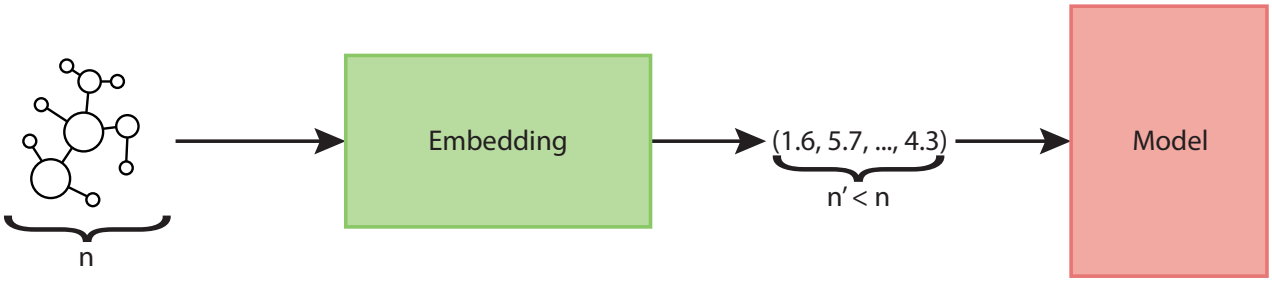


Figure 5.1: Example of graph embedding.

The way in which the embedding table is initialized has a number of influences. It accelerates convergence. By initializing the parameters of the embedding table, taking into account the information at our disposal, we can approach the convergence value. Also, a good initialization can make the learning process faster by guiding the model toward meaningful gradients early on while reducing the

risk of tending toward a local minimum, i.e. a value that is lower than at any nearby points, but not the lowest overall.

A number of pretrained embedding layers exist, but they are often very generic and widespread, which means they cannot capture the specific information in our dataset, and they generally need fine-tuning to adapt to the problem, which slows down the deep learning model. Another solution is to initialize embedding table values from scratch, by initializing all values to zero or randomly. While this has the advantage of being simple to implement, it does not limit the risk of minimum locality and can take a long time to train. In fact, the first results returned by these embedding tables will be random representations that make no sense in relation to the original data. And it is this data that will be supplied to the model so that it can make its predictions. In this way, we increase the number of epochs needed to start getting results. A final initialization strategy, which we might call data-specific one, consists in taking into account the data we have available before the model training to propose an initialization that corresponds to the working environment. This will be the subject of this chapter. We will look at how to create a coherent embedding from the information obtained following an *a priori* analysis of the dataset. Thus, we begin by describing how spectral methods work, and more specifically the MIRAMns method, in section 5.3. Then, in section 5.4, we will propose using MIRAMns to initialize our embedding table. Finally, we will analyze the performance and impact of this method on the performance of different deep learning models in section 5.5.

5.3 Spectral methods to compute dominant eigenvectors

We saw in the previous sections that the eigenproblem is the basis of many dimensionality reduction techniques. We recall that for a square n-size matrix $\boldsymbol{A}$, λ_i and the vector u_i are the i-th eigenvalue and its associated eigenvector if they satisfy

$$\boldsymbol{A} \cdot \boldsymbol{u}_i = \lambda_i \cdot \boldsymbol{u}_i \qquad (\mathcal{P})$$

for $i = 1, \ldots, n$. In many applications, including those requiring dimensionality reduction, we need to compute only a small number k of dominant eigenvalues and their associated eigenvectors. This

number represents the reduced dimension of the n-dimensional space in which the original problem $\mathcal{P}_n$ is defined. In this section, we present IRAM, the most robust numerical method for the computation of dominants eigenpairs when $\boldsymbol{A}$ is a very large size matrix of general structure, and its variant MIRAMns which allows to extract the dominant eigenvalues of $\boldsymbol{A}$ even when they are strongly clustered.

5.3.1 Restarted projection methods

The restarted projection methods are commonly used techniques for solving very large and sparse linear algebra problems. The general idea behind these methods is to project the very high-dimensional problem into a reduced-dimensional space. We project the problem in a reduced dimension and use its solution to define approximations for our original problem. This first approximation is then refined through restarting cycles until a sufficiently accurate approximation is obtained to find a solution to the initial very large problem. We will focus here on restarted projection methods for calculating eigenelements.

Projection into a subspace

To project a problem of large size n into a subspace, we start by choosing an initial subspace of dimension m with $m \ll n$. Let's project our n-dimensional problem $\mathcal{P}_n$ into the m-size subspace. This projected problem $\mathcal{P}_m$ is in a space of more reasonable dimension. It is now possible to use methods to solve the $\mathcal{P}_m$ problem that could not be applied to $\mathcal{P}_n$ due to its very large size. Once the $\mathcal{P}_m$ problem has been solved, we return to dimension$n$ to obtain an approximation of the $\mathcal{P}_n$ problem.

Restart process for iteration

The restart process is the core of the iteration process. Starting from the initial subspace, projection into the subspace provides an approximation to the n-dimensional problem. However, this approximation is highly dependent on the initial conditions. If the accuracy of the approximation is insufficient, we will redefine a new projection subspace to replace the previous one. Therefore, we use the results obtained to find a subspace which increases condition number. Once this new subspace has been defined, we start a new iteration. The process is repeated iteratively until sufficient precision is achieved.

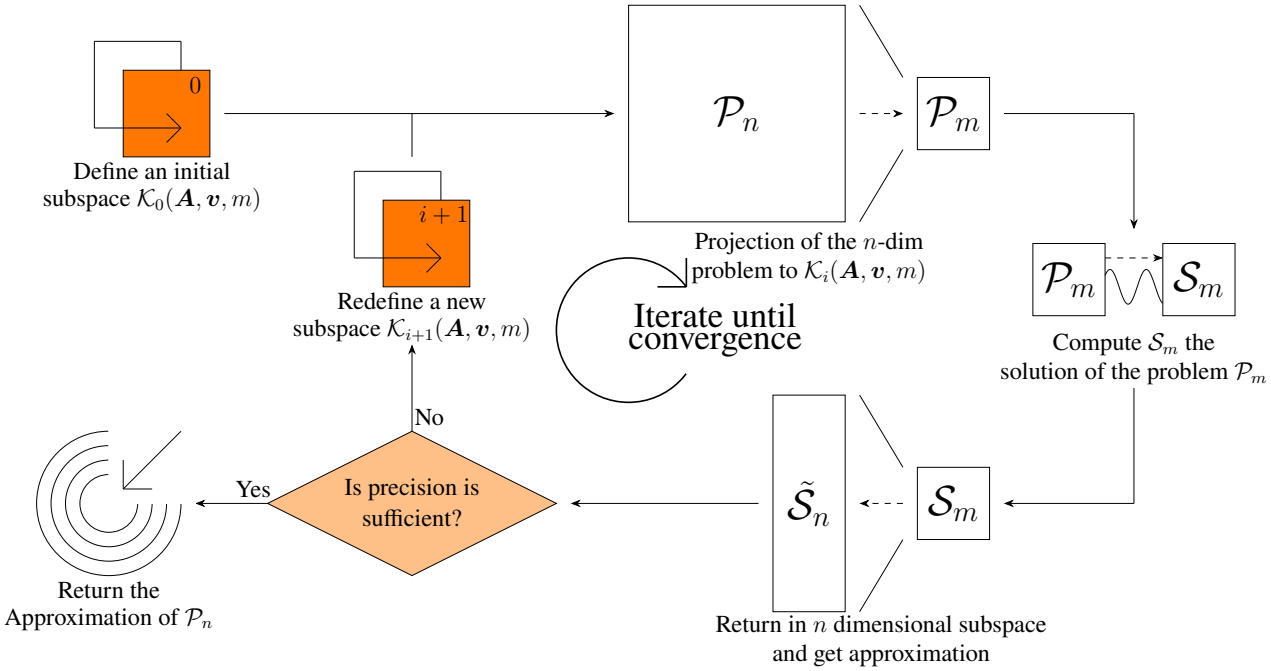


Figure 5.2: Overview of restarted projection methods.

Even if the number of operations is virtually unknown and potentially infinite, iterative methods can in most cases deliver results quickly with large matrices. The desired error ϵ that stops the iterative loop gives some control over the time needed to achieve the desired accuracy. Iterative algorithms are generally built around a succession of matrix operations, which are simple to implement and well-suited to parallel architectures. As matrix operations are very well adapted to a parallel and distributed environment, parallelization of these iterative methods generally enables good efficiency to be achieved in a distributed environment.

Iterative methods are, at the same time, subject to a number of limitations that need to be taken into account. Convergence is one possible problem. In some cases, the matrix may converge very slowly, or even not at all. This risk must be taken into consideration and minimized. The results may lack precision. Depending on how the method converges, the accuracy may be lower, and the approximation obtained with the iterative method will be relatively distant from the desired solution. Finally, the last constraint of iterative methods is the high sensitivity of the results to initial conditions. The speed of convergence and the accuracy of the approximation can be strongly influenced by the properties of the matrix and the initial conditions of the method. The choice of search subspace and initial vector values, for example, are factors to be modified and adjusted to ensure fast, accurate

convergence.

5.3.2 Krylov subspaces

A Krylov subspace is defined as the vector space generated by the successive powers of a matrix $\boldsymbol{A}$ multiplied by a normalized initial vector $\boldsymbol{v}$. Let

$$\mathcal{K}_m(\boldsymbol{A}, \boldsymbol{v}_1) = \text{span}(\boldsymbol{v}_1, \boldsymbol{A} \cdot \boldsymbol{v}_1, \ldots, \boldsymbol{A}^{m-1} \cdot \boldsymbol{v}_1) \tag{5.1}$$

be the m-th Krylov subspace with $\boldsymbol{v}_1 = \frac{\boldsymbol{v}}{\|\boldsymbol{v}\|_2}$. This set of m vectors forms a basis for the Krylov subspace.

Krylov subspaces are particularly well suited to the very large $\mathcal{P}_n$ problem. This is because calculating the basis vectors requires no matrix-matrix multiplication, just a succession of matrix-vector multiplications. Given the vector $\boldsymbol{v}_1$ as the first vector of the Krylov subspace basis, we multiply the last vector by $\boldsymbol{A}$ to obtain the next vector and so on.

The matrix powers quickly become numerically unstable, as the vectors $\boldsymbol{A}^i \cdot \boldsymbol{v}$ and $\boldsymbol{A}^{i+1} \cdot \boldsymbol{v}$ tend to be parallel as i increases, making the basis vectors linearly dependent. This is why methods based on Krylov subspace are generally based on an orthogonalization process. Extracting an orthogonal basis by the Gram-Schmidt process is the basis of Arnoldi's projection method.

Arnoldi method The Arnoldi method is an efficient technique which permits to compute an approximation of desired dominant eigenvalues of an n-size matrix $\boldsymbol{A}$. Arnoldi's projection is a simple sequence of matrix operations used to obtain the projection of a $\mathcal{P}_n$ problem into the Krylov subspace $\mathcal{K}_m$ [8]. This projection is represented by

$$\boldsymbol{A} \cdot \boldsymbol{V}_m = \boldsymbol{V}_m \cdot \boldsymbol{H}_m + \boldsymbol{f}_m \cdot \boldsymbol{e}_m^T \tag{5.2}$$

where $\boldsymbol{H}_m$ is the problem matrix projected into the subspace of size m with $\boldsymbol{H}_m \cdot \boldsymbol{y}_i = \lambda_i \cdot \boldsymbol{y}_i$, $\boldsymbol{V}_m = \{\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_m\}$ is an orthonormal basis of the Krylov subspace $\mathcal{K}_m$, $\boldsymbol{f}_m$ is the error vector and $\boldsymbol{e}_m$ is the m-th vector of the canonical basis of $\mathbb{C}^m$.

Arnoldi's method are highly dependent on the initialization of the initial vector and the size of the search subspace. Many techniques have been proposed to improve the approximation solutions. Krylov subspace eigenvalue algorithms for large matrices can be improved by restarting strategies [17].

Solutions for restarting an Arnoldi projection with a better subspace have clearly improved the performance of the Arnoldi method. This restart consists in defining a new initial vector from the information available to improve convergence. Among these solutions are those with explicit restart, noted ERAM for Explicitly restarted Arnoldi method. Saad [236] was the first to propose a restart by building the initial vector as an explicit linear combination of the Ritz vectors calculated during the previous iterations. The Ritz vectors are the approximate vectors that are used as the eigenvector solution of the initial $\mathcal{P}_n$ problem. Despite being easy to understand and use, explicit restarting is not always effective. Finding the right linear combination of previously calculated vectors is not an easy task. The more complex the combination, the greater the cost of restarting. What's more, explicit restarting is limited by the rounding errors associated with the previously calculated Ritz vectors. The new initial vector is an increasingly large sum of errors. These limits can be bypassed with an implicit restart.

5.3.3 IRAM

The first implicitly restart Arnoldi method has been proposed by Sorensen [253]. The algorithm is detailed in Algorithm 9. The difference with other restart methods is that we do not explicitly redefine a new vector to start a new iteration. Instead, the algorithm uses an implicitly shifted QR iteration to keep only the part of the spectrum of interest. This gives a good approximation of the desired eigenvalues of $\boldsymbol{A}$ without having to recalculate the whole spectrum at each iteration.

It uses the QR shifted algorithm on a small matrix which is a representation of $\boldsymbol{A}$ in the projection subspace. This lets us restart a new iteration of Arnoldi's method with an efficient and numerically stable formulation [247].

IRAM's implicit restarts make this method robust, as they maintain numerical stability iteration after iteration. Since IRAM does not recalculate the entire spectrum at each iteration, the IRAM approach significantly reduces the computational costs involved in solving eigenvalue problems.

Although the implicit restart approach has been clearly confirmed to be more efficient than the explicit restart approach [200], the IRAM approach remains complex to implement due to the choice of subspace size.

Algorithm 9 Implicitly restarted Arnoldi method (IRAM) algorithm.

Input: $\boldsymbol{A}$ the initial matrix, $\boldsymbol{V}_m$ the matrix of orthogonal basis issued from the Arnoldi factorization, $\boldsymbol{f}_m$ the residual vector with $\boldsymbol{A}\boldsymbol{V}_m = \boldsymbol{V}_m\boldsymbol{H}_m + \boldsymbol{f}_m\boldsymbol{e}_m^T$ the m-steps Arnoldi factorization
Output: The approximate k dominant eigen-elements of $\boldsymbol{A}$

1: **for** iterate until convergence **do**
2: $\boldsymbol{H}_m \leftarrow$ the projected matrix of $\boldsymbol{A}$ issued from the Arnoldi factorization
3: Compute the eigenvalues and their associated eigenvectors of $\boldsymbol{H}_m$
4: Compute residual norms
5: **if** convergence **then** `stop`
6: Compute Shifted QR to push important information in the top left-hand corner of the matrix
7: Truncate this new Arnoldi basis of size k
8: Begging Arnoldi factorization with the k-step and add additional steps of the Arnoldi process to obtain the new m-size basis $\boldsymbol{V}_m$
9: **end for**

5.3.4 MIRAMns

MIRAMns [247], which means Multiple implicitly restarted Arnoldi Method with nested subspaces, proposes an improvement on the IRAM approach by taking into account not only the initial subspace but also a set of nested subspace to improve the quality of the subspace when restarting. Instead of simply projecting into the associated Krylov subspace, MIRAMns will project into several Krylov subspaces of different sizes. The method then selects the subspace noted m_{best} in which the approximation of the Ritz elements is best, i.e. the subspace that provides the most accurate approximations to the $\mathcal{P}_n$ problem.. Then, continue execution as IRAM by applying the shifted QR algorithm to the selected subspace $\mathcal{K}_{m_{\text{best}}}$. Let l be the number of Krylov subspaces explored and m_i each of these subspaces with $i = 1, 2, \ldots, l$ and $m_1 < m_2 < \dot{<} m_l$. The properties of Krylov subspaces give the following nesting relationship:

$$\mathcal{K}_{m_1} \subset \mathcal{K}_{m_2} \subset \cdots \subset \mathcal{K}_{m_{l-1}} \subset \mathcal{K}_{m_l} \tag{5.3}$$

In other words, the computation of the Krylov subspace of size m_l is iteratively built up by passing

Algorithm 10 Multiple implicitly restarted Arnoldi method with nested subspaces (MIRAMns).

Input: $\boldsymbol{A}$ the initial matrix, $\boldsymbol{V}_{m_i}$ the matrix of orthogonal basis issued from the Arnoldi factorization, $\boldsymbol{f}_{m_i}$ the residual vector with $\boldsymbol{A}\boldsymbol{V}_{m_i} = \boldsymbol{V}_{m_i}\boldsymbol{H}_{m_i} + \boldsymbol{f}_{m_i}\boldsymbol{e}_{m_i}^T$ the m_i-steps Arnoldi factorization with $i = 1, 2, ..., l$

Output: The approximate k dominant eigen-elements of $\boldsymbol{A}$

for it until convergence **do**
  for $i \leftarrow 1, 2, ..., l$ **do**
    $\boldsymbol{H}_{m_i} \leftarrow$ the projected matrix of $\boldsymbol{A}$ issued from the Arnoldi factorization in the subspace m_i
    Compute the eigenvalues and their associated eigenvectors of $\boldsymbol{H}_m$
    Compute residual norms
    if convergence **then** `stop`
  end for
  Select m_{best} the best subspace size
  Compute Shifted QR to push important information in the top left-hand corner of the matrix
  Truncate this new Arnoldi basis of size k
  Begging Arnoldi factorization with the k-step and add additional steps of the Arnoldi process to obtain the new m_{best}-size basis $\boldsymbol{V}_{m_{\text{best}}}$
end for

through the smaller subspaces of size m_1, m_2, m_{l-1}. MIRAM is computationally equivalent to the IRAM algorithm for subspace size m_l, with the addition of a few additional steps to check the accuracy of the intermediate subspaces when constructing the Krylov subspace $\mathcal{K}_{m_l}$. In this way, MIRAMns can explore a larger number of spaces at a very low computational overhead, considerably improving the convergence of the iterative method. The algorithm 10 detail MIRAMns.

MIRAMns provides good approximations of the desired eigenvalues and eigenvectors, even with very large dataset. Thanks to its implicit restart system, which optimizes both the subspace size and the initial vector, the application of Arnoldi's method is all the simpler and more generalizable.

MIRAMns is composed of common matrix linear algebra operations. It is therefore a very well-studied operation with very good implementation in a distributed environment [217]. In the following section, we propose an embedding using this method.

5.4 Embedding initialization from dominant eigenvectors

In this section, we propose a solution for building a consistent embedding that can be applied to a wide range of data structures, thereby reducing the size of the input data and, at the same time, the cost of

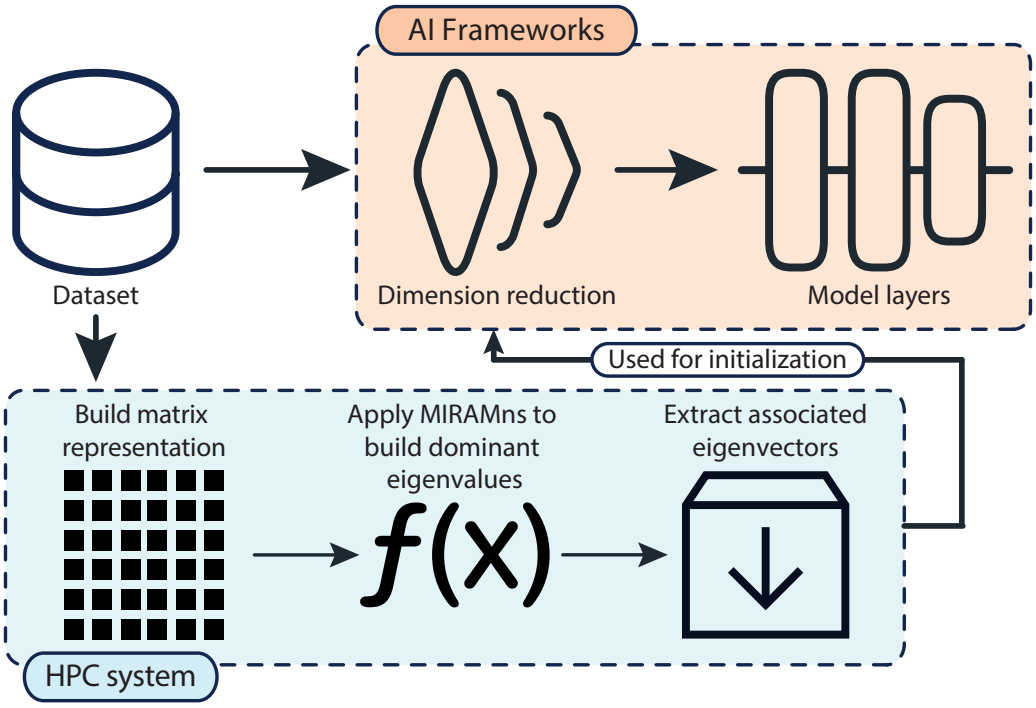


Figure 5.3: Overview of the proposed method to initialize embedding from eigenvectors associated to dominant eigenvalues.

training the model. We propose to extract the eigenvectors associated with the dominant eigenvalues of a matrix representing the dataset and then use these dominant eigenvectors to construct an embedding table. We will also discuss the choice of target embedding size for the reduction of dimension.

The figure 5.3 is an overview of the method.

5.4.1 Embedding strategy from data representation

We need to find a matrix representation of the data which can then be used as the initial matrix in the MIRAMns algorithm. As the dataset cannot generally be used as an initial matrix, it is possible to build a square non-Hermitian matrix which will capture a large part of the information on the data topology. The matrix to be built should be the same size as the number of features in the dataset, since the eigenvectors we extract will reduce the number of features.

Below are a few possible representations that can be used to apply the rest of the method.

Similarity matrix

The similarity matrix is a square matrix representing the similarity coefficient between two features [101]. It is a matrix used in many fields to detect features that are similar to each other. Each index is associated with a feature, and the element $\boldsymbol{A}_{i,j}$ contains the similarity score between the feature i and the feature j. In this way, the similarity matrix is symmetrical and all the elements on its diagonal $\boldsymbol{A}_{i,i} = 1$.

There are different ways of measuring similarity coefficients. Geometrically, we use the distance between features or the angle between feature vectors in high-dimensional space. The similarity matrix has the advantage of being bounded. All non-zero elements of this matrix lie within the interval $[-1, 1]$.

Co-occurrence matrix

The co-occurrence matrix [188] is a matrix that depicts the frequency of co-occurrence of pairs of items in a dataset. This matrix provides information about the relationships and patterns between items in a dataset. Each row and column with the same index represent a unique item, and the cells of the matrix store the frequency or count of how often two items co-occur together in the dataset. Initially used for visualizing co-citations [190], its use has become very popular in information science [176] for tasks like finding associations, identifying patterns, calculating similarity measures and building recommendation systems.

In NLP, a co-occurrence matrix can be used as the basis for numerical analysis of how words or word pairs appear together within a given corpus. For example, the co-occurrence matrix plays a crucial role in the GloVe [215], a neural network-based algorithm used to generate word embedding, by providing the statistical information necessary to learn the word embeddings through the neural network training process. The co-occurrence matrix also has a major role in different topic models like LDA [223, 21] or PLSA [183]. The co-occurrence matrix is thus a good representation to prepare the embedding layer.

Co-variance matrix

Another representation is the covariance matrix. A covariance matrix is a square matrix containing the variances (on the diagonals) and co-variances associated with several variables. It is a positive symmetrical matrix that visualizes the dependencies between different variables. This matrix is used to reduce the dimension of the data with principal component analysis [187].

The variance [281] of a variable x is defined as

$$\text{Var}(x) = \text{E}((x - \text{E}(x)^2)) = \text{E}(x^2) - (\text{E}(x))^2 \tag{5.4}$$

The covariance of two variables x and y is defined by [77]

$$\text{Cov}(x, y) = \text{E}((x - \text{E}(x))(y - \text{E}(y))) \tag{5.5}$$

with $\text{E}(x)$ is the expectation value of the variable x.

5.4.2 Compute and build the embedding table from eigenvectors

Now that we have a matrix representation of our dataset, let's see how to initialize an embedding table from this matrix. Let's call this square matrix $\boldsymbol{A}$. We will apply the MIRAMns algorithm with $\boldsymbol{A}$ as the initial matrix. The fact that we can use several sizes of search subspaces means that we can maximize the chances of convergence quickly and correctly, compared with the traditional approach of setting the size of the search subspace to $2k$ [171, 254]. We also fix the desired eigenelements as k the desired embedding size. We will discuss how to choose the size of k in the section 5.4.3. The choice of subspace sizes was discussed by Shahzadeh *et al.* [247].

MIRAMns gives us an approximation of the k dominant eigenvectors. We will use these k vectors to perform a k-dimensional embedding of our input data. Therefore, we will initialize the elements of the embedding table with the values of the dominant eigenvectors. In this way, each element of the vector output from the dimension reduction will correspond to the value of the input data projected in the direction of the associated eigenvector. The overall result is the projection into the

lower-dimensional subset k.

Once the embedding matrix has been defined, the cost of embedding the input data is equivalent to the cost of multiplication between two matrices, or matrix-vector multiplication if only one element needs to be embedded.

5.4.3 How to choose the right dimension for reduction?

The dimension reduction is a compromise between speed and accuracy. Find a good approximation of embedding size is quite difficult. A smaller dimension reduces the computing power required to process the data. It is therefore in our interest to reduce the dimension as much as possible. However, the quality of the data must be sufficient to represent the situation and enable the model to make a correct prediction. In general, variance is used to assess the quality of dimension reduction. The greater the variance, the more the embedded vector will represent the complexity of the input data.

A frequently used approach is to consider the proportion of variance explained by the k dominant eigenvectors. Given that the objective of dimension reduction is to retain a very large proportion of the dataset's information. We can therefore set a threshold of $0 \leq \alpha \leq 1$ and determine k in such a way that the k dominant eigenvectors explain at least a variance equivalence to α. The thresholds generally used are 80%, 90% and 95%. We use the Covariance matrix associated with the dataset to calculate the variance associated with each element. The sum of the elements on the diagonal gives the total variance. The eigenvector associated with the dominant eigenvalue explains the most variance. Simply cumulate the variances associated with the dominant eigenvalues until enough variance has been explained and choose the number of features that at least capture a desired percentage of variance α.

We said in the previous paragraph that the more dominant the eigenvalues, the greater the variance explained by them. Generally speaking, the first eigenvalues explain a lot of variance, then this decreases rapidly. So the difference in variance explained by eigenvalues i and $i + 1$ diminishes rapidly. Let's denote σ_i the variance explained by the dominant eigenvalue i. We can then fix k so that $\sigma_k - \sigma_{k+1} < \epsilon$.

Another technique for deciding the size of the embedding dimension is the permutation test [199].

The idea is to apply random permutations of the data and then compare the variances explained by the dominant eigenvectors before and after the permutations. The permutations will tend to balance the variance explained by each dominant eigenelement. The variance explained by the first dominant eigenvalues will decrease with the permutations, while the variance explained by the following ones will increase. We can choose k to keep only those eigenvalues that explain more variance before the permutation than after. The limitation of this approach is that it depends on randomness. Different permutations will affect the variance explained differently. We usually run several tests with different random permutations. We also need to define the optimal number of permutations, depending on the size of the dataset, which will have an impact on the variance explained.

5.5 Experiments

To see the impact of a dimension reduction based on eigenvectors associated with dominant eigenvalues, we experimented with different datasets. In the part 5.5.1, we will be describing our experimental environment and then observing the impact of dimension reduction on performance. Then, we will evaluate our implementation and test it with real-case datasets and models.

5.5.1 Experimentation environment and datasets

Our working environment is as follows: once again, we used the Ruche CPU cluster to compute the matrix representation of the dataset and to compute MIRAMns algorithm in a distributed environment. We have at our disposition 25 nodes comprising 2 Intel Xeon Gold 6230 20 cores @ 2.1 GHz (Cascade Lake). This enabled us to distribute calculations over a maximum of 1000 cores. Each compute node has a RAM capacity of 192GB. The Operating System is CentOS 7.9.2009 and the network technology is an Intel Omni-Path Architecture network 100 Gbit/s. Our disk storage capacity is 500 GB. It is a Spectrum Scale GPFS parallel file system that allows 9 GB/s input/output rate. To train and evaluate the models of neural networks, we used a computer with a 4-core i7-1165G7 processor running at 2.80GHz and an Intel IRIS Xe GPU running at 1300MHz. The computer is equipped with an SSD disk with read and write speeds of 2.200 GB/s and 1.200 GB/s respectively.

Type	Name	Features	Instances	Classes
Numerical	Date fruit	34	898	7
	Fetal Health	21	2126	3
	Radar	175	325834	7
Images	MNIST	784	70000	10
	CIFAR-10	3072	60000	10
	CIFAR-100	3072	60000	100

Table 5.1: Datasets Overview.

5.5.2 Evaluation of MIRAMns distributed implementation

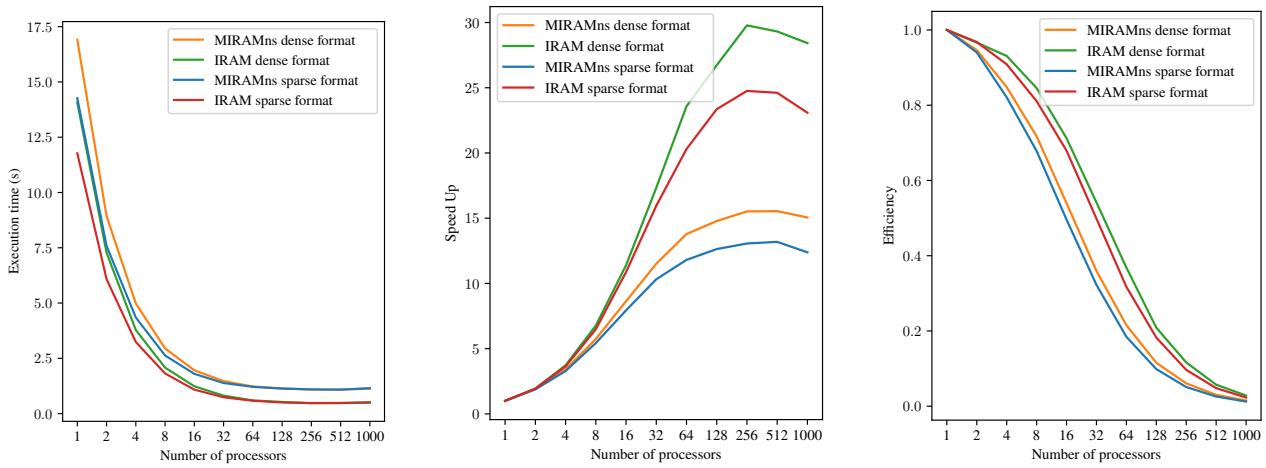
We have implemented MIRAMns for the CPU cluster, using MPI for communication between computation nodes. We used the Eigen [104, 105] library to store matrices and vectors. We also used this library to process elementary matrix operations. We carried out a set of experiments to test the performance of the MIRAMns implementation. We will look at and discuss the results in this section.

Matrix	Size of matrix	nonzero elements	Density
Epinions [175]	75,879	508,837	8.838×10^{-2}
Facebook [174]	4,039	88,234	5.409×10^{-3}

Table 5.2: General information about the test matrices.

We made comparisons between the IRAM method and the MIRAMns method. We have also performed executions by storing the dataset matrix in a sparse or dense format. These comparisons were made with the Facebook (figure 5.7) and Epinions (figure 5.8) datasets, and the covariance matrix of the datasets CIFAR-10 (figure 5.4), CIFAR-100 (figure 5.5), MNIST (figure 5.6). Information on each of these dataset is available in table 5.1 and 5.2.

Execution times are comparable for all methods with covariance matrices (see figures 5.4a, 5.5a and 5.6a). Since the matrix density is 100%, Eigen does not seem to compress the sparse matrix and treats it as dense. If the matrix were compressed into a sparse format, the execution time would be significantly higher than with the dense format, due to the high density (see figure 4.1a). We can see that the speed-up is greater with the IRAM method than with MIRAMns (figures 5.4b, 5.5b and 5.6b). This is due to the fact that the additional computations performed at each operation are not parallel, but are performed by all compute nodes simultaneously in a sequential way. This increases the proportion of work that is not parallelized, which has an impact on speed-up and method efficiency.

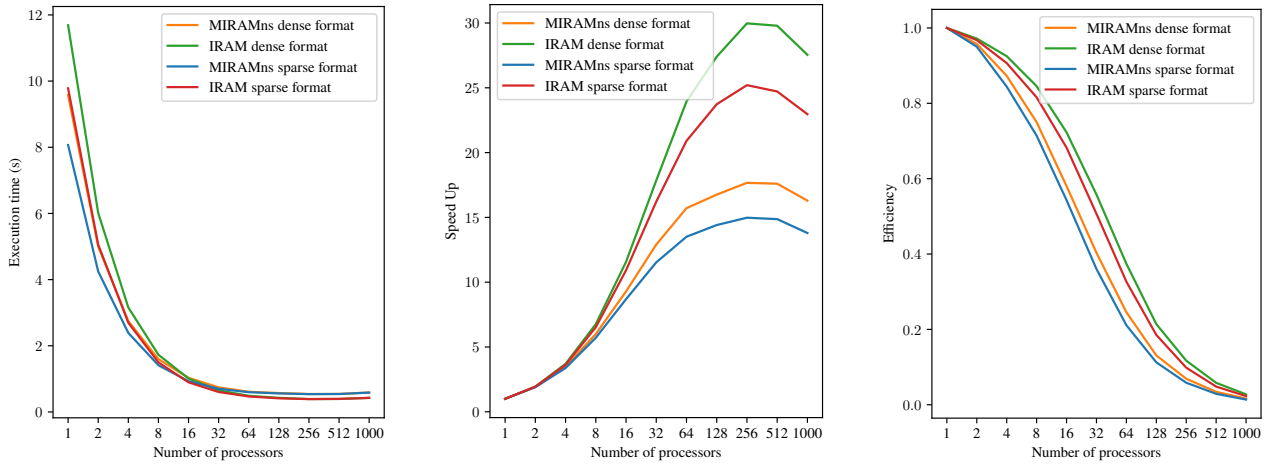


(a) Execution time (b) Speedup (c) Efficiency

Figure 5.4: Comparison of execution time, speedup and efficiency for IRAM and IRAMns executions for the covariance matrix of the CIFAR-10 dataset. The figures illustrate the performance differences between dense and sparse storage matrices.

We can also see on figures 5.4b, 5.5b, 5.6b and 5.7b that we have a speedup peak that comes down with more compute nodes. This is explained by the fact that matrices are relatively small and the workload distributed to each node is not sufficient to compensate for the additional communications induced by the increase in the number of compute nodes. We can see on figure 5.6b that this performance peak is obtained with 32 compute nodes with the MNIST covariance matrix, whereas the peak is observed with 256 nodes with the CIFAR-10 and CIFAR-100 dataset matrices (figure 5.4b and 5.5b). This is consistent with the fact that MNIST has a much smaller covariance matrix than the other two datasets (784 vs. 3072).

With the Facebook dataset, figure 5.7, the matrix to be processed is sparse (density = 5.409×10^{-3}), which has a strong impact on execution times. These are very low compared to execution times with the dense matrix. This also has an impact on speedup and efficiency. Indeed, as the workload is lower, the use of more compute nodes will have a much less significant impact. However, using a sparse compression format to handle the matrix means that larger matrices can be handled and manipulate. This is the case with the Epinions dataset on figure 5.8. The dataset matrix is too large and causes an out-of-memory condition when we try to load it and treat it as dense with few nodes. We have to use 32 compute nodes before the dense blocks of the matrix can be stored in the nodes' memory. Speedup and efficiency are therefore computed in relation to the first value obtained.



(a) Execution time (b) Speedup (c) Efficiency

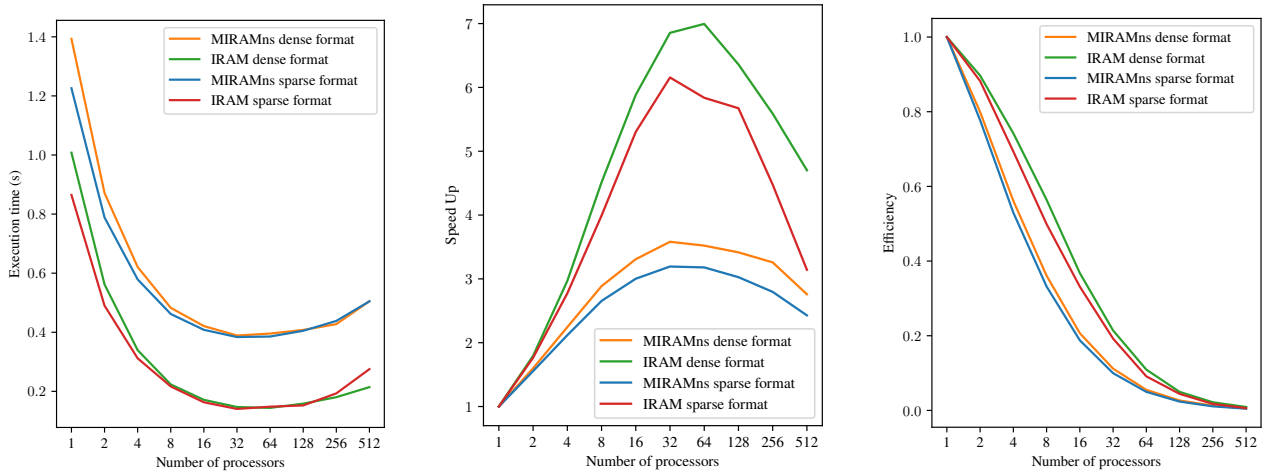
Figure 5.5: Comparison of execution time, speedup and efficiency for IRAM and IRAMns executions for the covariance matrix of the CIFAR-100 dataset. The figures illustrate the performance differences between dense and sparse storage matrices.

Once again, we observe a high speedup with the dense format, again due to the fact that most of the execution time is devoted to compute matrix operations with the dense matrix, and that this operation is easily parallelizable with high efficiency. The advantage of compressing matrices into sparse format is that you can process larger matrices while reducing the costs associated with matrix operations. Overall, MIRAMns requires slightly more computational power, which translates into a relatively low overhead in terms of execution time. This surplus is directly linked to the number of subspaces to be explored, which is a user choice. However, it does have an impact on speedup and efficiency when the number of compute nodes is increased. This is because the extra workload of MIRAMns compared to IRAM is not parallelizable and is part of the proportion of code that is run sequentially on all nodes simultaneously. The workload is too low to consider parallelizing this part of the code, since it is not sufficient to cover the additional communication that will be generated by parallelization.

5.5.3 Results with real-case datasets

Analysis of dimension reduction on performance.

We used three different numerical classification datasets of different sizes: Date fruits [155], Fetal Health [161] and Radar [1]. We also use MNIST [52] and CIFAR-10 [157] image dataset to Information



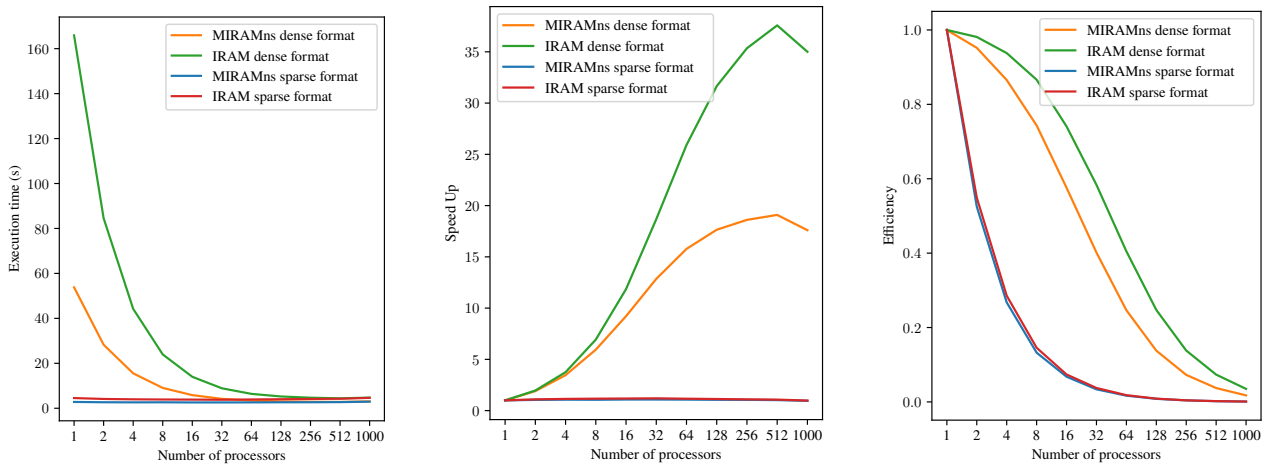
(a) Execution time (b) Speedup (c) Efficiency

Figure 5.6: Comparison of execution time, speedup and efficiency for IRAM and IRAMns executions for the covariance matrix of the MNIST dataset. The figures illustrate the performance differences between dense and sparse storage matrices.

on these datasets is available in Table 5.1. For all experiments, we randomly divided the dataset into two sets for training and testing. The set for model performance validation represents 20% of the total dataset for numerical dataset. For image dataset, we used the Keras data split for training and test subset. The test set is composed of 10000 images. Only the training dataset was used to build the data matrix representation. The format used to represent the data is the feature co-variance matrix. MIRAMns was run on these matrices. The implementation of the algorithm is based on the Eigen [104] library for linear algebra and matrix calculation operations. We used 100 computational cores for each execution of MIRAMns.

For each model associated with each dataset, we chose the hyperparameter combination that gave the highest accuracy following a hyperparameter tuning phase with the model without any dimension reduction. All hyperparameters were fixed and no further hyperparameter tuning was performed with the models with dimension reduction and smaller size.

We used a multi-layer perceptron (MLP) [116] composed of 5 hidden layers for numerical dataset prediction and 2 hidden layers for image classification dataset. The result of this MLP is used to make a prediction about the class of the given input element. The size of the input layer is modified to match the size of the input data. We have built different models with different numbers of perceptrons on each layer, so as to modify the number of parameters making up the neural networks. The composition



(a) Execution time (b) Speedup (c) Efficiency

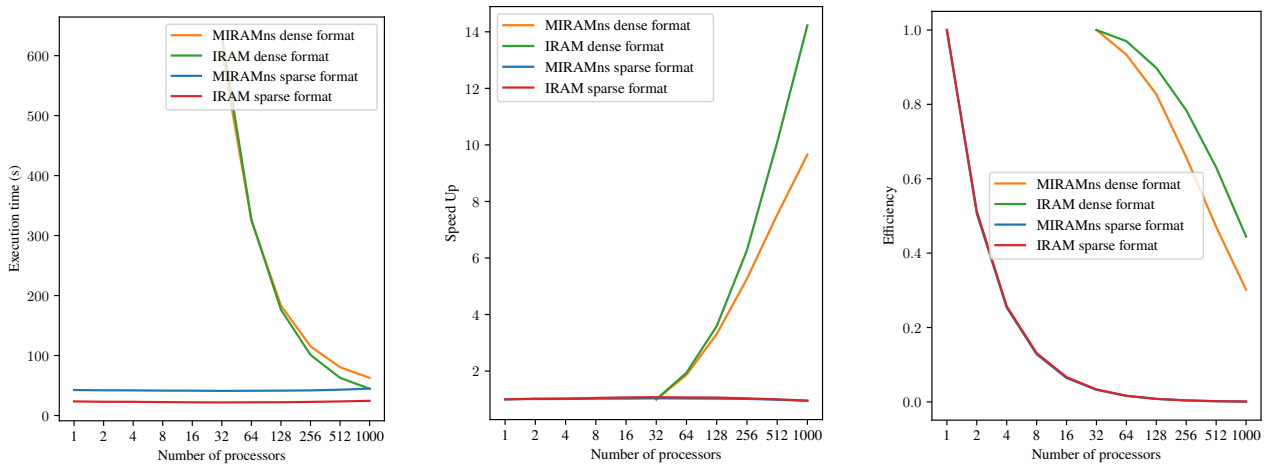
Figure 5.7: Comparison of execution time, speedup and efficiency for IRAM and IRAMns executions for the Facebook dataset. The figures illustrate the performance differences between dense and sparse storage matrices.

Dataset type	**Model**	**Hidden sizes**	**Total params #**
Numerical	Large	512, 512, 512, 512, 256	$\approx 9.39 \times 10^5$
	Medium	256, 256, 256, 256, 128	$\approx 2.40 \times 10^5$
	Low	64, 64, 64, 64, 32	$\approx 1.7 \times 10^4$
	Very low	32, 32, 32, 32, 16	$\approx 4.9 \times 10^3$
Images	Large	512, 512	$\approx 1.841 \times 10^6$
	Medium	256, 256	$\approx 8.55 \times 10^5$
	Low	64, 64	$\approx 2.01 \times 10^5$
	Very low	32, 32	$\approx 9.97 \times 10^4$

Table 5.3: Hidden size and approximation of the number of parameters for each MLP models.

of the number of perceptrons in each layer of each model is available in table 5.3. This variation in size will enable us to see the evolution of accuracy when the model is relatively too small for the task in question. It will also allow us to see the impact of the dimension reduction described in the previous section.

Figures 5.9 and 5.10 plot the accuracy obtained with the test set with different model sizes and different embedding sizes respectively for the Date fruit and Fetal Health datasets. Performances without any dimension reduction have been added in clear on the curves. It can be seen from these two figures that performance as a function of embedding size follows a logarithmic curve. Adding an extra dimension when the number of dimensions is low results in significant performance gains. Above a certain size, adding a new dimension to represent the data no longer significantly increases



(a) Execution time (b) Speedup (c) Efficiency

Figure 5.8: Comparison of execution time, speedup and efficiency for IRAM and IRAMns executions for the Epinions social network dataset. The figures illustrate the performance differences between dense and sparse storage matrices.

performance.

We can also see from these figures that model performance is globally proportional to the number of parameters. The larger the model, the more precise the learning process. We can also see that performance with dimension reduction is relatively good compared to performance without dimension reduction. We can see from the Date fruit dataset in figure 5.9 that dimension reduction improves performance compared to the model without dimension reduction. This is particularly true for small models.

	Large size model		Medium size model	
Input dimension	175	10	175	10
Number of parameters	1,143,815	1,059,847	275,975	233,991
Training Exec. time (s)	1180.81	1146.43	760.51	727.466
Test accuracy (%)	98.772	98.406	98.958	98.390

Table 5.4: Results of Radar dataset with embedding dimension reduction to 10.

Image classification

MLP networks are not the most suitable models for image processing. Because all neurons in each layer are fully connected to all neurons in the previous and next layers, MLPs are very expensive to train with large images. Moreover, they are not optimal for understanding and finding general patterns

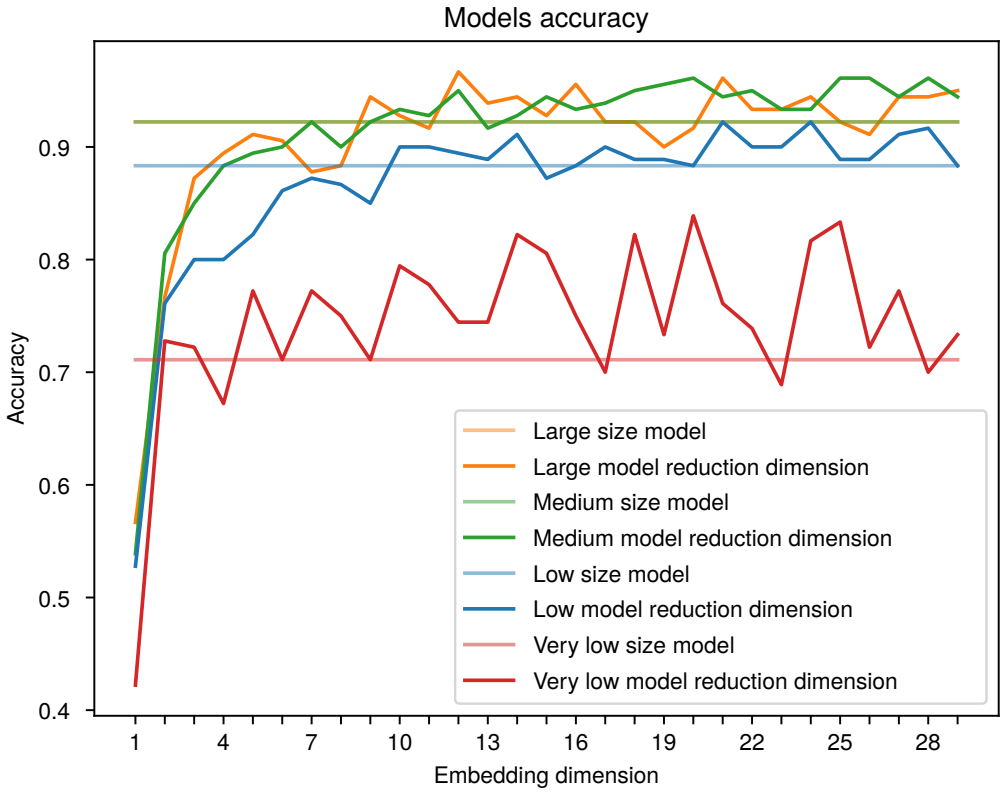


Figure 5.9: Comparison of test accuracy with the Date fruit dataset with different model sizes as a function of embedding size.

in images. To get the best accuracy with this type of data, we prefer to use other types of neural network such as convolution networks, which have a structure that limits the exponential growth in the number of parameters with very large data sizes [221] and are effective for generalizing patterns during the learning phase [85]. The following results are not intended to achieve the best possible accuracy with these datasets, but to compare the impact of dimension reduction with dominant eigenvectors on model performance.

Figure 5.11 compares performance with the MNIST dataset. MNIST is a collection of handwritten black-and-white figures. Each image represents one digit and is normalized and centered, measuring 28 pixels on a side. MNIST was designed to provide a consistent and homogeneous database for the scientific community, and has become a reference dataset in the field of deep learning for evaluating the performance of handwriting recognition models.

The results obtained with our MLP implementation on this dataset are very impressive. As with

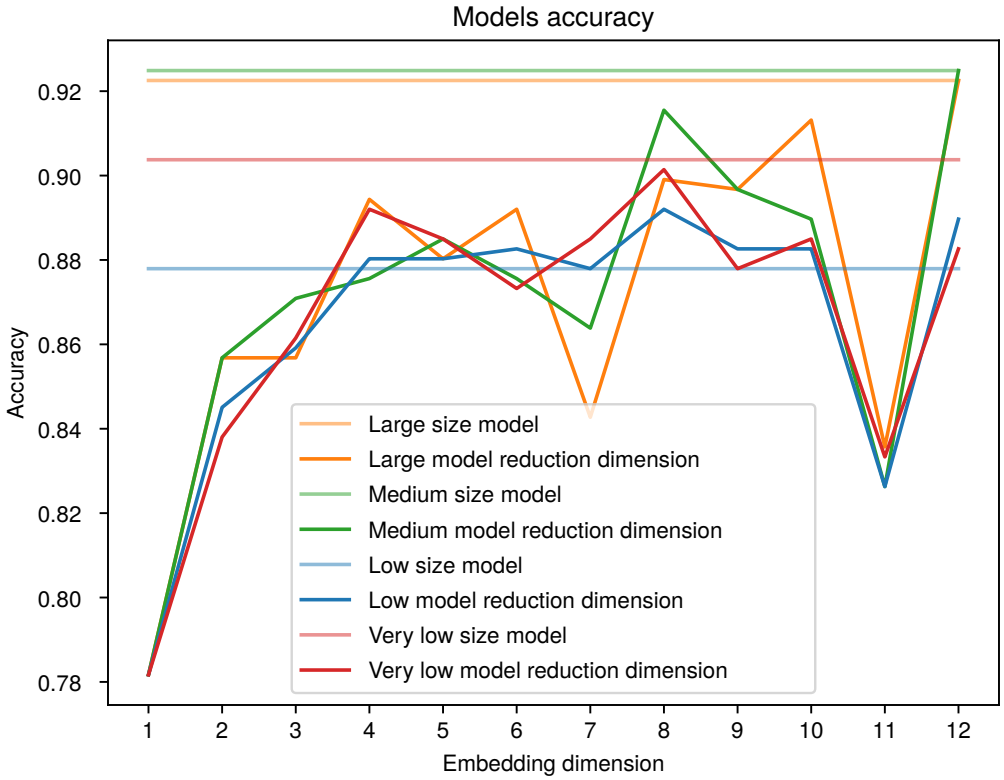


Figure 5.10: Comparison of test accuracy with the Fetal Health dataset with different model sizes as a function of embedding size.

numerical datasets, the larger the model, the greater the accuracy. All models, even the very smallest, achieve an accuracy of over 96%. With larger models, performance as the reduction dimension increases tends towards performance without dimension reduction. It is therefore in our interest to use data embedding with dominant eigenvectors to significantly reduce training time. As a reminder, a smaller input dimension has an impact on model size and training speed. With MNIST, the time to train the large model without dimension reduction is 62.7 seconds. With 28-dimension embedding, which offers the same accuracy, the training time is 52.3 seconds. Dimension reduction reduces model training time by 20%. The very small size of the model means that it cannot match the performance obtained without dimension reduction. .

Figure 5.12 compares performance with different model sizes on the CIFAR-10 dataset. This is a set of 32 by 32 color images divided into 10 classes. The input dimension is therefore 3072. It can be seen that using the dominant eigenvectors to reduce the input dimension improves the accuracy

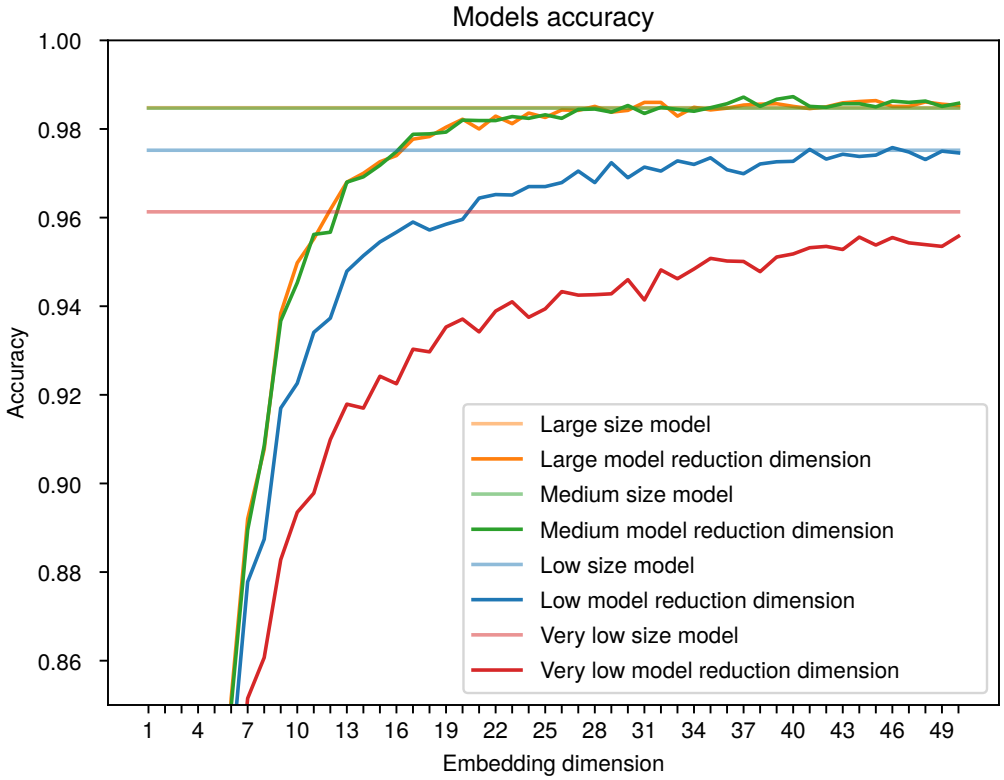


Figure 5.11: Comparison of test accuracy with the MNIST image dataset with different model sizes as a function of embedding size.

of the models with CIFAR-10. While the performance of the very small model is close to random without dimension reduction, we obtain significantly better performance with dimension reduction. The small model achieves 40 times better performance with dimension reduction than with the very large model and no dimension reduction. In this case, dimension reduction considerably decreases the computation time required for training and significantly improves performance.

5.6 Conclusion and discussion

In this section, we have proposed a technique for building a consistent embedding of datasets with different types of data structure with an high precision. By leveraging matrix representations, this method optimizes dimensionality reduction while retaining maximum variance through dominant eigenvectors. The MIRAMns algorithm proved instrumental in efficiently approximating eigenvectors associated with dominant eigenvalues, demonstrating its capability to handle large-scale data

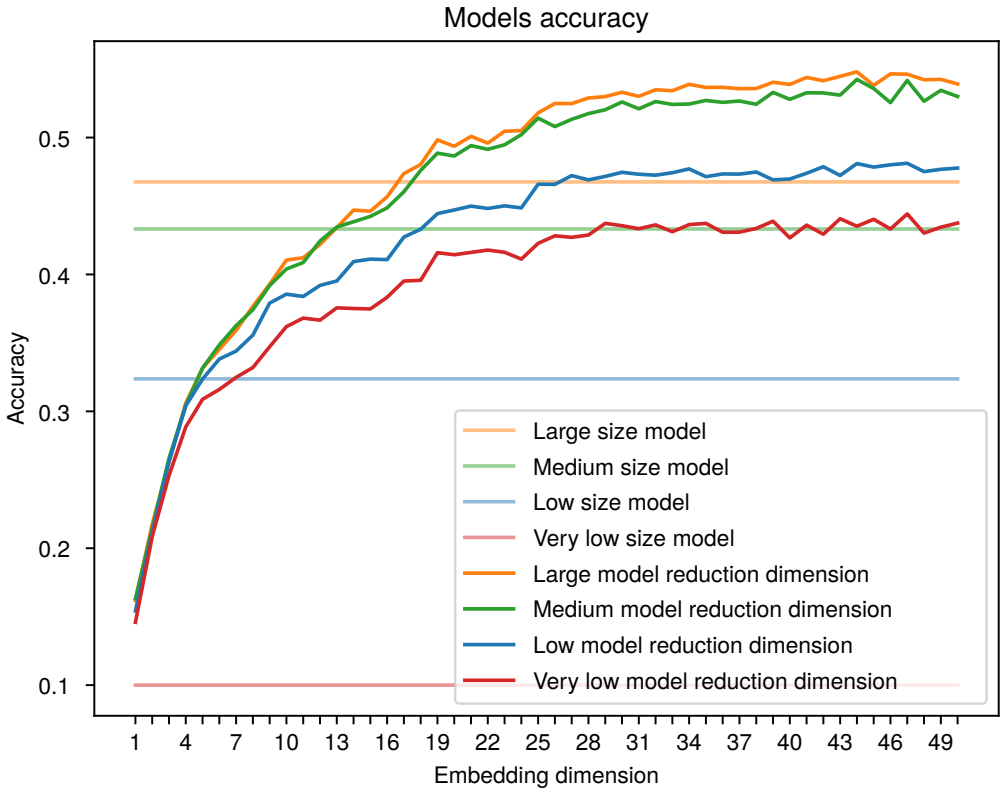


Figure 5.12: Comparison of test accuracy with the CIFAR-10 image dataset with different model sizes as a function of embedding size.

effectively.

Our experimental results underscored the significant potential of this approach to achieve high performance with reduced input dimensions. This reduction not only streamlined the computational process but also facilitated faster training times without compromising model accuracy. Specifically, the experiments highlighted that embedding data using dominant eigenvectors can substantially reduce training costs while maintaining competitive performance levels.

Overall, we have shown in this study that dimension reduction methods are effective in reducing model sizes, enabling a good balance to be struck between model accuracy and size. The use of dimension reduction methods to embed data in a coherent way facilitates data representation while preserving essential variability for effective learning. Future work may explore expanding the application of this method across diverse data types and distributed environments to further enhance its scalability and impact.

Chapter 6

General Conclusion

Throughout this manuscript, we have seen that to meet the challenges posed by large deep learning models, solutions need to be implemented at all levels. On the algorithmic stage, complexity reduction techniques such as model compression, quantization and dimensionality reduction help lower computational demands while maintaining performance. Moreover, certain model architectures are inherently more hardware-friendly; for example, transformers enable efficient parallelization due to their self-attention mechanism, while mixture-of-experts models optimize memory usage by activating only a subset of their parameters per inference. On the software side, modern deep learning frameworks such as TensorFlow, PyTorch or MindSpore provide powerful tools for developing distributed models, but managing parallelization and inter-machine communications remains a significant challenge. On the hardware side, specialized architectures such as GPUs, TPUs and dedicated AI accelerators have considerably improved processing capabilities. However, scaling these models on massive parallel infrastructures, such as supercomputers, requires careful coordination between hardware, algorithms and software.

The main objective of this thesis is to enhance the performance of large DL models by proposing innovative solutions to reduce the costs associated with their development and training. Specifically, by minimize the training costs of AI neural networks in massively distributed environments.

6.1 Summary of findings

As this end, we began by exploring neural networks specially designed to process data in the form of graphs. The irregular structure of graphs requires the use of sophisticated techniques to take into account topological data in addition to features for prediction. The quality of a graph neural networks depends precisely on how the graph structure information is extracted. After demonstrating the value of deploying dropping solutions to reduce overfitting on a specific graph, we proposed RankedDrop, a novel sampling method to improve the extraction of graph structure information. Based on spatial-aware selection, we proposed to take into account information on the importance of graph nodes. This information is then used to control the distribution of the random selection. Thanks to the randomness, this makes it possible to have different graphs for training, but which remain consistent with the structure of the basic graph. We have shown in the experiments in chapter 3 that RankedDrop outperforms both the baseline (without dropout) and standard random dropout methods that disregard graph structure. This is particularly the case in semi-supervised learning when the model is large. RankedDrop allows for improving the accuracy of the 8-layers GCN by up to twice and enhancing performance by more than 50% compared to random dropping.

As sparse BLAS constitute the computational core of large models, we study then the matrix-matrix multiplication sequence with both sparse and dense matrices. We proposed a way of distributing the data between the different compute nodes to ensure a low amount of communications while avoiding all-to-all communications. This distribution meets the computational needs to deal with very large graphs and compute large graph neural networks models.

Understanding the structure and relationships within textual data is crucial for efficiently processing large NLP datasets, and the co-occurrence matrix serves as a fundamental tool in capturing these relationships. We have also proposed a solution for constructing the co-occurrence matrix of an NLP dataset rapidly, taking advantage of both data sparsity and data arithmetic. Using a dataset generator, we were able to study the impact of different parameters on execution time and check that these results corresponded to the cost analysis we carried out to compare methods for constructing the co-occurrence matrix.

As explained in this thesis, dimensionality reduction is one of the most important levers for

reducing data complexity and model size. Building upon this, we proposed a technique based on a numerical method for constructing a coherent representation of an element in a reduced dimension. We have based this technique on MIRAMns, a method perfectly adapted to a distributed environment and specially designed for very large datasets. Joining numerical method of dimension reduction to embedding allows us to guarantee a consistent initialization of the embedding while keeping a maximum of variance, which ensures embedded vector will represent correctly the complexity of the input data.

These contributions have allowed us to tackle the challenge of reducing the cost of large models across various levels and contexts. They align with the broader objective of identifying and extracting the most relevant information from data to streamline and enhance subsequent model training. By introducing scalable solutions, we ensure their adaptability for future applications, enabling them to handle increasingly larger datasets and operate effectively in massively distributed environments.

6.2 Perspectives

There are numerous promising directions for extending this work. In this thesis, we have begun to apply different numerical methods to deep learning to improve performance. They allow us to obtain more information about the data we are dealing with, in order to facilitate the processing. Whether to help choose hyperparameter values or to support the model by improving data quality, the applications of numerical methods are vast and promising. Dominant eigenvalues and analysis of variance can be used to refine the choice of embedding dimensions, while spectral analysis of graphs can provide additional information for prediction. The cost of these methods is generally low compared with the total cost of model training, especially for very large models. It is essential to explore their potential for reducing the model workload, either by reducing the number of iterations through faster convergence, or by reducing the number of models to be trained during the hyper-parameter optimization phase.

Although the proposed methods are efficient in a distributed environment, it is essential to continue improving the scalability of the methods. It would be interesting to think about the distribution of computations for multi-level machines. This would make it possible to take into account all the

specificities of the machine in order to design optimized and efficient methods. In addition, the experiments we have presented in this work are intended to show that the algorithm works in a distributed environment. A study phase to adapt these solutions to the environment actually used today is necessary, with a multi-level analysis to use hardware accelerators on the compute nodes.

Appendix A

Résumé en Français

A.1 Contexte et motivations

L'intelligence artificielle est aujourd'hui au cœur d'une transformation numérique de grande ampleur, bouleversant un large éventail de secteurs, allant de la médecine à la finance, en passant par les industries manufacturières et les technologies de l'information. En particulier, l'apprentissage profond (*Deep Learning*, DL) a permis d'atteindre des performances inédites dans de nombreuses applications, telles que la reconnaissance d'images, la traduction automatique et la modélisation du langage naturel. Cependant, le succès grandissant du DL repose sur des modèles de plus en plus complexes, nécessitant des quantités massives de données et une puissance de calcul exponentielle.

Les modèles d'apprentissage profond modernes et les grands modèles de langage, comme GPT-4, BERT ou encore les réseaux neuronaux convolutionnels utilisés pour l'analyse d'images, possèdent des milliards de paramètres. Leur entraînement repose sur des infrastructures massivement parallélisées, exploitant des milliers d'unités de calcul spécialisées, comme les GPU (*Graphics Processing Units*) ou les TPU (*Tensor Processing Units*).

La croissance rapide de la taille des modèles présente plusieurs défis majeurs. En premier lieu, il y a l'explosion des coûts de calcul liés à l'entraînement de ces modèles. Ils mobilisent des supercalculateurs pendant plusieurs semaines, voire plusieurs mois, impliquant des coûts financiers et énergétiques considérables. Cela pose également des problèmes d'impact écologique, mais aussi

d'accessibilité et de démocratisation. Seules quelques grandes entreprises ou puissants États sont aujourd'hui capables de supporter les coûts liés à l'entraînement de ces modèles, ce qui limite l'accès à ces technologies pour de nombreux laboratoires de recherche, PME et institutions académiques.

La grande taille des modèles pose également des problématiques d'extensibilité. La distribution efficace des tâches sur des architectures massivement parallèles et distribuées est un enjeu crucial pour optimiser le temps d'exécution, ce qui a un impact significatif sur le temps d'entraînement des modèles. Pour cela, il faut s'assurer que les algorithmes utilisés soient adaptés à ce type d'environnement et que leur implémentation limite les communications et assure une répartition optimale de la charge de travail entre les différents nœuds de calcul.

L'efficacité des grands modèles dépend principalement de l'algorithme qui les décrit, ainsi que de la qualité des données utilisées pour leur entraînement. Avant de déployer ces modèles sur des systèmes de calcul haute performance (HPC), il est essentiel de les optimiser d'un point de vue conceptuel. Cela signifie que, pour concevoir un modèle optimal fournissant des résultats précis et pertinents, toutes les étapes des algorithmes correspondants doivent être optimisées, notamment la réduction de la dimensionnalité, la prise en compte des données creuses et l'introduction d'un parallélisme intrinsèque. Il est également crucial de disposer de données de haute qualité pour obtenir des résultats fiables et pertinents.

En outre, pour déployer efficacement ces modèles sur des architectures haute performance et exploiter pleinement les capacités de ces systèmes, il est indispensable d'envisager des paradigmes de programmation parallèle et distribuée performants pour leur mise en œuvre.

Cette thèse porte sur ces deux aspects : l'optimisation du modèle et sa mise en œuvre efficace sur des systèmes haute performance. Autrement dit, nous visons à améliorer les processus d'entraînement à la fois par l'efficacité algorithmique et par une approche optimisée du déploiement.

Nos recherches se concentrent principalement sur l'optimisation des processus d'entraînement de très grands réseaux neuronaux. L'accent est mis sur les méthodes et applications liées au prétraitement des modèles, bien que les techniques de réduction de la dimensionnalité abordées dans cette thèse puissent être appliquées à tous les niveaux des modèles de deep learning.

A.2 Contributions

Dans le but de répondre à ces objectifs, nous avons commencé par explorer les réseaux de neurones de DL spécialement conçus pour manipuler des graphes. En effet, les graphes sont une structure de données largement utilisés pour représenter et modéliser des objets ainsi que les relations entre ces objets. Les graphes ont une structure irrégulière, ce qui nécessite un traitement particulier pour en extraire les informations topologiques. Or, la performance et la précision d'un réseau de neurones de graphes dépendent fortement de la qualité de l'analyse de la structure du graphe.

Dans de nombreuses applications, le graphe est fixe, c'est-à-dire qu'il n'y a qu'un seul graphe et que ce dernier ne change pas. Cela pose de sérieux problèmes de surapprentissage puisque le modèle de réseau de neurones va s'entraîner uniquement sur cette structure et aura des difficultés à généraliser son apprentissage à d'autres graphes lors de la phase d'inférence. Pour remédier à cela, la littérature scientifique propose de générer, à chaque époque d'entraînement, un sous-graphe différent en retirant aléatoirement des parties du graphe original. Cette méthode a permis d'obtenir un gain important en performance, notamment avec les réseaux de neurones de grande taille.

C'est dans ce contexte que nous avons proposé RankedDrop, une nouvelle méthode de construction de sous-graphes basée sur l'analyse topologique du graphe. Nous avons choisi de prendre en compte l'importance des nœuds du graphe pour construire les sous-graphes lors de l'entraînement des modèles de réseaux de neurones. Ces informations topologiques sont utilisées pour contrôler la distribution de la sélection aléatoire employée pour la génération de sous-graphes. Ainsi, il est possible d'obtenir des sous-graphes différents lors de l'entraînement, tout en conservant une cohérence avec la structure du graphe original.

RankedDrop a été spécialement conçu pour être flexible et adaptable : il peut intégrer divers critères afin de mieux comprendre la topologie du graphe. Il est également très général et peut être appliqué à une multitude d'architectures de modèles de réseaux de neurones de graphes. Pour expérimenter notre méthode, nous avons utilisé deux critères pour générer l'importance topologique des nœuds : les scores de PageRank et le degré des nœuds. Ces deux informations complémentaires permettent d'évaluer à la fois l'importance globale des nœuds dans la structure via PageRank et leur importance locale en fonction du nombre de relations.

Nos expérimentations montrent que notre méthode RankedDrop surpasse à la fois l'approche sans modification du graphe original et l'approche basée sur la génération totalement aléatoire de sous-graphes. RankedDrop permet d'améliorer les performances des réseaux de neurones dans toutes les situations testées, indépendamment de la taille du modèle et de l'architecture utilisée. Son application est particulièrement efficace en apprentissage semi-supervisé et sur des modèles de grande taille. RankedDrop permet d'améliorer la précision des modèles à 8 couches de plus de 50 % par rapport à une génération totalement aléatoire de sous-graphes, soit le double de précision par rapport à l'approche naïve des réseaux de neurones.

Comme les matrices creuses sont au cœur de nombreuses méthodes et algorithmes des modèles de grande taille, nous avons exploré l'algèbre linéaire creuse et, plus spécifiquement, son implication dans les modèles de DL. Les matrices creuses, qui contiennent une proportion importante de valeurs nulles et donc une faible densité de valeurs non nulles, posent des problèmes de calcul importants en raison de leur nature non structurée. La répartition de la charge de travail dans un environnement massivement distribué est très compliquée. Il est essentiel de relever ces défis pour garantir de bonnes performances des modèles d'apprentissage profond dans les environnements HPC.

Pour cela, nous avons étudié la succession de multiplications matrice-matrice avec des matrices creuses et denses. Nous avons proposé un découpage et une distribution des données qui assurent une bonne répartition de la charge de travail entre les nœuds de calcul ainsi qu'une réduction des communications, qui sont très gourmandes en temps. Notre distribution permet également d'éviter les communications *all-to-all*, c'est-à-dire les échanges où chaque nœud doit envoyer des messages à tous les autres nœuds de calcul. En limitant les communications localement, nous réduisons la charge de données circulant sur l'ensemble du réseau et minimisons ainsi le risque de goulot d'étranglement au niveau des performances.

Grâce à ces spécificités, notre découpage est particulièrement bien adapté à une grille de nœuds de calcul, un type d'architecture de cluster couramment utilisé dans les environnements HPC. Ce type d'architecture se caractérise par des communications limitées aux nœuds voisins, sans possibilité de communication directe entre tous les nœuds.

Comprendre la structure et les relations entre les données textuelles est une étape essentielle pour traiter efficacement les jeux de données en traitement du langage naturel. La matrice de co-occurrence est un outil fondamental pour saisir ces relations. Il s'agit d'un élément clé utilisé dans de nombreuses méthodes de DL pour l'incorporation de mots, que l'on nommera *embedding* dans la suite. Par exemple, certaines méthodes d'embedding utilisent la matrice de co-occurrence entre les mots du jeu de données pour construire un embedding cohérent et regrouper les mots en fonction des similarités mises en évidence par cette matrice.

La matrice de co-occurrence peut être obtenue à partir de la matrice d'incidence du jeu de données et d'une multiplication matricielle. Cependant, les spécificités des jeux de données en traitement du langage naturel ont la particularité de produire des matrices d'incidence à faible densité, composées exclusivement de données booléennes.

Nous avons proposé Sparse-Pairwise, une nouvelle approche pour construire la matrice de co-occurrence en tirant parti à la fois de la nature booléenne des données et de leur faible densité. Nous avons réalisé une analyse théorique de complexité entre les différentes approches de construction de la matrice de co-occurrence, en termes de temps de calcul et d'occupation mémoire. Cette analyse montre que notre approche est la plus efficace du point de vue de la complexité en temps et qu'elle est d'autant plus rapide que la densité de la matrice est faible, ce qui est généralement le cas dans les jeux de données utilisés en traitement du langage naturel.

Afin de vérifier et valider notre analyse de complexité, nous avons construit un générateur de corpus afin d'étudier précisément l'impact de chaque paramètre sur les performances. Nous avons mené une analyse approfondie de l'extensibilité des méthodes grâce au générateur, puis testé différents ensembles de données issus d'applications réelles pour vérifier que les résultats obtenus avec le générateur étaient représentatifs des performances observées sur de véritables jeux de données. Nos expérimentations ont montré que l'utilisation de notre méthode Sparse-Pairwise permet de construire la matrice de co-occurrence jusqu'à 34 fois plus rapidement que les autres approches.

Un bon embedding doit permettre de construire une représentation capable de capturer le sens et les relations entre les éléments du jeu de données, en les transformant en vecteurs numériques denses, afin qu'ils puissent être utilisés comme entrée pour le reste du modèle. C'est une technique essentielle

pour permettre aux modèles de DL de comprendre et de traiter des données à la structure complexe de manière efficace. Encore faut-il que la représentation vectorielle en sortie de l'embedding soit cohérente par rapport aux données initiales. Or, réussir à capter la complexité sous-jacente de ces données dans un vecteur est une tâche difficile.

Un embedding non pertinent causerait des problèmes de cohérence des données, ce qui limiterait le potentiel du modèle et ralentirait son apprentissage. Pour répondre à cette problématique, nous proposons une solution d'initialisation de l'embedding qui s'appuie sur des informations extraites du jeu de données pour fournir une représentation matricielle cohérente. Nous proposons d'utiliser les vecteurs propres dominants pour projeter les données initiales et obtenir une représentation vectorielle pertinente.

Cependant, l'extraction des vecteurs propres dominants n'est pas directement possible avec un jeu de données rectangulaire classique. C'est pourquoi nous proposons d'utiliser une représentation matricielle carrée du jeu de données, comme la matrice de co-occurrence. Les vecteurs propres dominants permettent de maximiser la variance de la projection des données initiales et donc d'assurer une meilleure explicabilité des données projetées.

Nous avons proposé d'utiliser la méthode MIRAMns pour trouver les vecteurs propres dominants. Cette méthode a la particularité d'explorer différentes dimensions de sous-espaces, ce qui la rend très robuste et permet son utilisation même lorsque les valeurs propres sont regroupées. En effet, la capacité à converger malgré des groupes de valeurs propres permet une bonne généralisation de cette méthode sur un large spectre de jeux de données. De plus, MIRAMns étant une méthode de projection itérative, elle est idéalement adaptée aux problèmes de très grandes tailles. Ces caractéristiques assurent l'extensibilité et la stabilité de cette initialisation d'embedding.

Nous avons montré, grâce à des expérimentations, que la méthode MIRAMns est adaptée aux environnements distribués et tire profit de la faible densité des jeux de données. Enfin, nous avons démontré, en entraînant des modèles de deep learning de différentes tailles, que l'embedding réalisé grâce aux vecteurs propres dominants permet de conserver suffisamment d'informations pour garantir un apprentissage cohérent. De plus, la réduction de dimension des données d'entrée simplifie les données, facilitant ainsi leur traitement par le modèle.

Ainsi, un bon embedding permet d'obtenir les mêmes performances qu'un modèle dix fois plus grand sans embedding. Cela montre que notre approche d'embedding est particulièrement intéressante pour réduire la complexité des modèles tout en assurant leurs performances.

A.3 Conclusion

Les travaux présentés dans cette thèse s'inscrivent dans une démarche d'optimisation des modèles d'apprentissage profond à grande échelle, en traitant des problématiques liées à leur entraînement, leur scalabilité et leur déploiement sur des architectures massivement parallèles. Nous avons proposé des approches innovantes pour améliorer la qualité des données et l'efficacité des modèles de réseaux neuronaux, afin de faciliter l'apprentissage sur des données non structurées ou semi-structurées. Restructurer ou réduire la complexité de ce type de données permet d'en améliorer la qualité et d'optimiser leur traitement par le modèle. Ces avancées ouvrent la voie à une meilleure exploitation des ressources de calcul haute performance, tout en réduisant les coûts et l'empreinte énergétique des entraînements massifs.

Les résultats obtenus démontrent l'impact positif de ces optimisations, tant sur la précision des modèles que sur leur efficacité computationnelle. Ces travaux constituent ainsi une contribution importante à la recherche en deep learning et en calcul distribué, avec des perspectives intéressantes pour la démocratisation de l'accès aux modèles de grande échelle. Des pistes futures pourraient notamment explorer l'intégration de ces méthodes à d'autres types d'architectures de réseaux de neurones, ainsi que l'exploration d'autres méthodes numériques avancées pour approfondir ces travaux à différents niveaux du modèle.

A.4 Publications

Ces travaux ont fait l'objet de publications dans des conférences internationales:

- Petit, Q. R., Li, C., & Emad, N. (2022, December). Distributed and Parallel Sparse Computing for Very Large Graph Neural Networks. In *2022 IEEE International Conference on Big Data*

(Big Data) (pp. 6796-6798). IEEE. [217]

- Petit, Q. R., Li, C., Petiton, S. G., Chai, K., & Emad, N. (2022, December). Enhancing Graph Convolutional Networks by Topology Sampling. In *2022 IEEE International Conference on Big Data (Big Data)* (pp. 1316-1323). IEEE. [218]

- Petit, Q., Li, C., & Emad, N. (2024, May). An Efficient and Scalable Approach to Build Co-occurrence Matrix for DNN's Embedding Layer. In *Proceedings of the 38th ACM International Conference on Supercomputing (ICS'24)* (pp. 286-297). ACM. [216]

Bibliography

[1] Crop mapping using fused optical-radar data set. UCI Machine Learning Repository, 2020. DOI: https://doi.org/10.24432/C5G89D.

[2] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

[3] N. Ahmed, N. Mateev, K. Pingali, and P. Stodghill. Compiling imperfectly-nested sparse matrix codes with dependences. *JAD*, 1(1):3–4, 2000.

[4] M. Alanyali and B. Hajek. Analysis of simple algorithms for dynamic load balancing. *Mathematics of Operations Research*, 22(4):840–871, 1997.

[5] S.-i. Amari. Backpropagation and stochastic gradient descent method. *Neurocomputing*, 5 (4-5):185–196, 1993.

[6] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485, 1967.

[7] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, et al. *LAPACK users' guide*. SIAM, 1999.

[8] W. E. Arnoldi. The principle of minimized iterations in the solution of the matrix eigenvalue problem. *Quarterly of applied mathematics*, 9(1):17–29, 1951.

[9] S. Ayesha, M. K. Hanif, and R. Talib. Overview and comparative study of dimensionality reduction techniques for high dimensional data. *Information Fusion*, 59:44–58, 2020.

[10] D. Baek, S. Hwang, T. Heo, D. Kim, and J. Huh. Innersp: A memory efficient sparse matrix multiplication accelerator with locality-aware inner product processing. In *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 116–128. IEEE, 2021.

[11] Baer. Multiprocessing systems. *IEEE Transactions on Computers*, 100(12):1271–1277, 1976.

[12] R. G. Balagafshe, A. Akoushideh, and A. Shahbahrami. Matrix-matrix multiplication on graphics processing unit platform using tiling technique. *Indonesian Journal of Electrical Engineering and Computer Science*, 28(2):1012–1019, 2022.

[13] S. Balay, S. Abhyankar, M. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, A. Dener, V. Eijkhout, W. Gropp, et al. Petsc users manual. 2019.

[14] P. Baldi. Gradient descent learning algorithm overview: A general dynamical systems perspective. *IEEE Transactions on neural networks*, 6(1):182–195, 1995.

[15] P. Bartlett. The sample complexity of pattern classification with neural networks: The size of the weights is more important than the size of the network. *IEEE Trans. Inf. Theory*, 44: 525–536, 1998. doi: 10.1109/18.661502.

[16] B. Bartoldson, A. Morcos, A. Barbu, and G. Erlebacher. The generalization-stability tradeoff in neural network pruning. *Advances in Neural Information Processing Systems*, 33:20852–20864, 2020.

[17] C. Beattie, M. Embree, and J. Rossi. Convergence of restarted krylov subspaces to invariant subspaces. *SIAM Journal on Matrix Analysis and Applications*, 25(4):1074–1109, 2004.

[18] A. Beguelin, J. Dongarra, A. Geist, R. Manchek, and V. Sunderam. *A Userś Guide to PVM Parallel Virtual Machine*. University of Tennessee, 1991.

[19] T. Ben-Nun and T. Hoefler. Demystifying parallel and distributed deep learning: An in-depth concurrency analysis. *ACM Computing Surveys (CSUR)*, 52(4):1–43, 2019.

[20] D. Blalock, J. J. Gonzalez Ortiz, J. Frankle, and J. Guttag. What is the state of neural network pruning? *Proceedings of machine learning and systems*, 2:129–146, 2020.

[21] D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent dirichlet allocation. *Journal of machine Learning research*, 3(Jan):993–1022, 2003.

[22] G. E. Blelloch. *Vector models for data-parallel computing*, volume 2. MIT press Cambridge, 1990.

[23] A. Bojchevski, O. Shchur, D. Zügner, and S. Günnemann. Netgan: Generating graphs via random walks. In *International conference on machine learning*, pages 610–619. PMLR, 2018.

[24] A. Bojchevski, J. Klicpera, B. Perozzi, M. Blais, A. Kapoor, M. Lukasik, and S. Günnemann. Is pagerank all you need for scalable graph neural networks? In *ACM KDD, MLG Workshop*, 2019.

[25] A. Bojchevski, J. Klicpera, B. Perozzi, A. Kapoor, M. Blais, B. Rózemberczki, M. Lukasik, and S. Günnemann. Scaling graph neural networks with approximate pagerank. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2464–2473, 2020.

[26] R. Bommasani, D. A. Hudson, E. Adeli, R. Altman, S. Arora, S. von Arx, M. S. Bernstein, J. Bohg, A. Bosselut, E. Brunskill, et al. On the opportunities and risks of foundation models. *arXiv preprint arXiv:2108.07258*, 2021.

[27] A. Borodin, G. O. Roberts, J. S. Rosenthal, and P. Tsaparas. Link analysis ranking: algorithms, theory, and experiments. *ACM Transactions on Internet Technology (TOIT)*, 5(1):231–297, 2005.

[28] P. Bose. Communication versus computation. *IEEE Micro*, 24(05):5–5, 2004.

[29] J. S. Breese, D. Heckerman, and C. M. Kadie. Anonymous web data from www. microsoft. com. *Microsoft Research, Redmond WA*, pages 98052–6399, 1998.

[30] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. *Computer networks and ISDN systems*, 30(1-7):107–117, 1998.

[31] N. R. Brodnik, S. Carton, C. Muir, S. Ghosh, D. Downey, M. Echlin, T. Pollock, and S. Daly. Perspective: Large language models in applied mechanics. *Journal of Applied Mechanics*, 2023. doi: 10.1115/1.4062773.

[32] A. Byerly, T. Kalganova, and I. Dear. No routing needed between capsules. *Neurocomputing*, 463:545–553, 2021.

[33] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, and K. Skadron. A performance study of general-purpose applications on graphics processors using cuda. *Journal of parallel and distributed computing*, 68(10):1370–1380, 2008.

[34] T. Cheatham, A. Fahmy, D. Stefanescu, and L. Valiant. Bulk synchronous parallel computing—a paradigm for transportable software. *Tools and Environments for Parallel and Distributed Systems*, pages 61–76, 1996.

[35] C. Chen, K. Li, S. G. Teo, X. Zou, K. Wang, J. Wang, and Z. Zeng. Gated residual recurrent graph neural networks for traffic prediction. In *Proceedings of the AAAI conference on artificial intelligence*, volume 33, pages 485–492, 2019.

[36] D. Chen, Y. Lin, W. Li, P. Li, J. Zhou, and X. Sun. Measuring and relieving the over-smoothing problem for graph neural networks from the topological view. In *Proceedings of the AAAI conference on artificial intelligence*, volume 34, pages 3438–3445, 2020.

[37] J. Chen, C. Wang, Z. Ma, J. Chen, D. He, and S. Ackland. Remote sensing scene classification based on convolutional neural networks pre-trained using attention-guided sparse filters. *Remote Sensing*, 10(2):290, 2018.

[38] K. Chen. Confidential high-performance computing in the public cloud. *IEEE Internet Computing*, 27(1):24–32, 2023.

[39] L. Chen. *Deep learning and practice with mindspore*. Springer Nature, 2021.

[40] W. Chen, Y. Wang, and S. Yang. Efficient influence maximization in social networks. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 199–208, 2009.

[41] H. Choi, B. H. Lee, S. Y. Chun, and J. Lee. Towards accelerating model parallelism in distributed deep learning systems. *Plos one*, 18(11):e0293338, 2023.

[42] T. Choudhary, V. Mishra, A. Goswami, and J. Sarangapani. A comprehensive survey on model compression and acceleration. *Artificial Intelligence Review*, 53:5113–5155, 2020.

[43] A. Chowdhery, S. Narang, J. Devlin, M. Bosma, G. Mishra, A. Roberts, P. Barham, H. W. Chung, C. Sutton, S. Gehrmann, et al. Palm: Scaling language modeling with pathways. *Journal of Machine Learning Research*, 24(240):1–113, 2023.

[44] A. Chtchelkanova, J. Gunnels, G. Morrow, J. Overfelt, and R. A. van de Geijn. Parallel implementation of blas: General techniques for level 3 blas. *Concurrency: Practice and Experience*, 9(9):837–857, 1997.

[45] E. Cohen. Structure prediction and computation of sparse matrix products. *Journal of Combinatorial Optimization*, 2:307–332, 1998.

[46] P. Comon. Independent component analysis, a new concept? *Signal processing*, 36(3):287–314, 1994.

[47] C. Cortes. Support-vector networks. *Machine Learning*, 1995.

[48] F. Cristian. Understanding fault-tolerant distributed systems. *Communications of the ACM*, 34 (2):56–78, 1991.

[49] H. B. Curry. The method of steepest descent for non-linear minimization problems. *Quarterly of Applied Mathematics*, 2(3):258–261, 1944.

[50] T. A. Davis and Y. Hu. The university of florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)*, 38(1):1–25, 2011.

[51] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, et al. Large scale distributed deep networks. *Advances in neural information processing systems*, 25, 2012.

[52] L. Deng. The mnist database of handwritten digit images for machine learning research [best of the web]. *IEEE signal processing magazine*, 29(6):141–142, 2012.

[53] T. Dettmers, M. Lewis, S. Shleifer, and L. Zettlemoyer. 8-bit optimizers via block-wise quantization. In *International Conference on Learning Representations*, 2021.

[54] T. Dettmers, M. Lewis, Y. Belkada, and L. Zettlemoyer. Gpt3. int8 (): 8-bit matrix multiplication for transformers at scale. *Advances in Neural Information Processing Systems*, 35:30318–30332, 2022.

[55] C. Ding and X. He. K-means clustering via principal component analysis. In *Proceedings of the twenty-first international conference on Machine learning*, page 29, 2004.

[56] J. J. Dongarra, J. D. Croz, S. Hammarling, and R. J. Hanson. A proposal for an extended set of fortran basic linear algebra subprograms. *ACM Signum Newsletter*, 20(1):2–18, 1985.

[57] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. S. Duff. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software (TOMS)*, 16(1):1–17, 1990.

[58] J. J. Dongarra, S. W. Otto, M. Snir, D. Walker, et al. An introduction to the mpi standard. *Communications of the ACM*, 18:11, 1995.

[59] N. Dube, D. Roweth, P. Faraboschi, and D. Milojicic. Future of hpc: The internet of workflows. *IEEE Internet Computing*, 25(5):26–34, 2021.

[60] I. S. Duff, M. A. Heroux, and R. Pozo. The sparse blas. Technical report, CM-P00041360, 2001.

[61] I. S. Duff, M. A. Heroux, and R. Pozo. An overview of the sparse basic linear algebra subprograms: The new standard from the blas technical forum. *ACM Transactions on Mathematical Software (TOMS)*, 28(2):239–267, 2002.

[62] C. Eckart and G. Young. The approximation of one matrix by another of lower rank. *Psychometrika*, 1(3):211–218, 1936.

[63] D. Edelman, J. McDonald, D. Bestor, M. Jones, B. Li, D. Tiwari, D. Zhao, S. Samsi, and V. Gadepally. Interventions to reduce ai energy requirements. *HPCA NetZero*, 2023.

[64] E. Elsen, M. Dukhan, T. Gale, and K. Simonyan. Fast sparse convnets. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 14629–14638, 2020.

[65] M. Engelman. Fidap: Examples manual, revision 6.0. *Fluid Dynamics International, Evanston, IL*, 1991.

[66] M. Engelman and I. Hasbani. Matrix-free solution algorithms in a finite element context. In *Technical Report 88-1*. Fluid Dynamics International Evanston, Illinois, 1988.

[67] Epoch AI. Data on notable ai models, 2024. URL `https://epochai.org/data/notable-ai-models`. Accessed: 2024-10-21.

[68] M. Espadoto, R. M. Martins, A. Kerren, N. S. Hirata, and A. C. Telea. Toward a quantitative survey of dimension reduction techniques. *IEEE transactions on visualization and computer graphics*, 27(3):2153–2173, 2019.

[69] M. Espig, K. K. Naraparaju, and J. Schneider. A note on tensor chain approximation. *Computing and Visualization in Science*, 15:331–344, 2012.

[70] S. Eyerman and L. Eeckhout. Modeling critical sections in amdahl's law and its implications for multicore design. In *Proceedings of the 37th annual international symposium on Computer architecture*, pages 362–370, 2010.

[71] S. Fan, Y. Rong, C. Meng, Z. Cao, S. Wang, Z. Zheng, C. Wu, G. Long, J. Yang, L. Xia, et al. Dapple: A pipelined data parallel approach for training large models. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 431–445, 2021.

[72] W. Fan, Y. Ma, Q. Li, Y. He, E. Zhao, J. Tang, and D. Yin. Graph neural networks for social recommendation. In *The world wide web conference*, pages 417–426, 2019.

[73] K. Fatahalian, J. Sugerman, and P. Hanrahan. Understanding the efficiency of gpu algorithms for matrix-matrix multiplication. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 133–137, 2004.

[74] W. Fedus, B. Zoph, and N. Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *Journal of Machine Learning Research*, 23(120):1–39, 2022.

[75] D. G. Feitelson and L. Rudolph. Parallel job scheduling: Issues and approaches. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 1–18. Springer, 1995.

[76] M. R. Feldesman. Classification trees as an alternative to linear discriminant analysis. *American Journal of Physical Anthropology: The Official Publication of the American Association of Physical Anthropologists*, 119(3):257–275, 2002.

[77] W. Feller and P. M. Morse. An introduction to probability theory and its applications, 1958.

[78] R. A. Fisher. The use of multiple measurements in taxonomic problems. *Annals of eugenics*, 7 (2):179–188, 1936.

[79] M. J. Flynn. Some computer organizations and their effectiveness. *IEEE transactions on computers*, 100(9):948–960, 1972.

[80] S. Fortunato. Community detection in graphs. *Physics reports*, 486(3-5):75–174, 2010.

[81] S. Fortune and J. Wyllie. Parallelism in random access machines. In *Proceedings of the tenth annual ACM symposium on Theory of computing*, pages 114–118, 1978.

[82] J. Frankle and M. Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks. In *International Conference on Learning Representations*, 2018.

[83] J. Fresno, D. Barba, A. Gonzalez-Escribano, and D. R. Llanos. Hitflow: A dataflow programming model for hybrid distributed-and shared-memory systems. *International Journal of Parallel Programming*, 47:3–23, 2019.

[84] H. Fuchs, J. Poulton, J. Eyles, T. Greer, J. Goldfeather, D. Ellsworth, S. Molnar, G. Turk, B. Tebbs, and L. Israel. Pixel-planes 5: A heterogeneous multiprocessor graphics system using processor-enhanced memories. *ACM Siggraph Computer Graphics*, 23(3):79–88, 1989.

[85] K. Fukushima. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological cybernetics*, 36(4):193–202, 1980.

[86] J. Furman and R. Seamans. Ai and the economy. *Innovation policy and the economy*, 19(1): 161–191, 2019.

[87] H. L. Gelernter. Realization of a geometry theorem proving machine. In *IFIP congress*, pages 273–281, 1959.

[88] J. Gertler and J. Cao. Pca-based fault diagnosis in the presence of control and dynamics. *AIChE Journal*, 50(2):388–402, 2004.

[89] J. Ghorpade. Gpgpu processing in cuda architecture. *Advanced Computing: An International Journal*, 3(1):105–120, 2012.

[90] J. R. Gilbert, C. Moler, and R. Schreiber. Sparse matrices in matlab: Design and implementation. *SIAM journal on matrix analysis and applications*, 13(1):333–356, 1992.

[91] C. L. Giles and C. W. Omlin. Pruning recurrent neural networks for improved generalization performance. *IEEE transactions on neural networks*, 5(5):848–851, 1994.

[92] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl. Neural message passing for quantum chemistry. In *International conference on machine learning*, pages 1263–1272. PMLR, 2017.

[93] S. S. Girija. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *Software available from tensorflow. org*, 39(9), 2016.

[94] F. Girosi, M. Jones, and T. Poggio. Regularization theory and neural networks architectures. *Neural computation*, 7(2):219–269, 1995.

[95] S. Goel, A. Anderson, J. Hofman, and D. J. Watts. The structural virality of online diffusion. *Management science*, 62(1):180–196, 2016.

[96] D. Goldfarb. Modification methods for inverting matrices and solving systems of linear algebraic equations. *Mathematics of Computation*, 26(120):829–852, 1972.

[97] M. Gong, Z. Tang, H. Li, and J. Zhang. Evolutionary multitasking with dynamic resource allocating strategy. *IEEE Transactions on Evolutionary Computation*, 23(5):858–869, 2019.

[98] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[99] J. Gou, B. Yu, S. J. Maybank, and D. Tao. Knowledge distillation: A survey. *International Journal of Computer Vision*, 129(6):1789–1819, 2021.

[100] B. Gouin-Ferland, R. Coffee, and A. C. Therrien. Data reduction through optimized scalar quantization for more compact neural networks. *Frontiers in Physics*, 10:957128, 2022.

[101] J. C. Gower. A general coefficient of similarity and some of its properties. *Biometrics*, pages 857–871, 1971.

[102] M. Greenacre, P. J. Groenen, T. Hastie, A. I. d'Enza, A. Markos, and E. Tuzhilina. Principal component analysis. *Nature Reviews Methods Primers*, 2(1):100, 2022.

[103] A.-M. Grisogono. How could future ai help tackle global complex problems? *Frontiers in Robotics and AI*, 7:509614, 2020.

[104] G. Guennebaud, B. Jacob, et al. Eigen v3. http://eigen.tuxfamily.org, 2010.

[105] G. Guennebaud, B. Jacob, et al. Eigen. *URl: http://eigen. tuxfamily. org*, 3(1):8, 2010.

[106] D. Guide. Cuda c++ programming guide. *NVIDIA, July*, 2020.

[107] D. Guo, D. Yang, H. Zhang, J. Song, R. Zhang, R. Xu, Q. Zhu, S. Ma, P. Wang, X. Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

[108] Q. Guo, F. Zhuang, C. Qin, H. Zhu, X. Xie, H. Xiong, and Q. He. A survey on knowledge graph-based recommender systems. *IEEE Transactions on Knowledge and Data Engineering*, 34(8):3549–3568, 2020.

[109] A. Gupta, P. Faraboschi, F. Gioachin, L. V. Kale, R. Kaufmann, B.-S. Lee, V. March, D. Milojicic, and C. H. Suen. Evaluating and improving the performance and scheduling of hpc applications in cloud. *IEEE Transactions on Cloud Computing*, 4(3):307–321, 2014.

[110] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan. Deep learning with limited numerical precision. In *International conference on machine learning*, pages 1737–1746. PMLR, 2015.

[111] W. L. Hamilton, R. Ying, and J. Leskovec. Representation learning on graphs: Methods and applications. *IEEE Data Eng. Bull.*, 40(3):52–74, 2017.

[112] S. Han, J. Pool, J. Tran, and W. Dally. Learning both weights and connections for efficient neural network. *Advances in neural information processing systems*, 28, 2015.

[113] S. A. Haque, M. T. Parvez, and S. Hossain. Gpu algorithms for structured sparse matrix multiplication with diagonal storage schemes. *Algorithms*, 17(1):31, 2024.

[114] F. E. Harrell et al. *Regression modeling strategies: with applications to linear models, logistic regression, and survival analysis*, volume 608. Springer, 2001.

[115] S. Harris and D. Harris. *Digital design and computer architecture*. Morgan Kaufmann, 2015.

[116] T. Hastie. The elements of statistical learning: data mining, inference, and prediction, 2009.

[117] X. He, S. Pal, A. Amarnath, S. Feng, D.-H. Park, A. Rovinski, H. Ye, Y. Chen, R. Dreslinski, and T. Mudge. Sparse-tpu: Adapting systolic arrays for sparse matrices. In *Proceedings of the 34th ACM international conference on supercomputing*, pages 1–12, 2020.

[118] Y. He, X. Zhang, and J. Sun. Channel pruning for accelerating very deep neural networks. In *Proceedings of the IEEE international conference on computer vision*, pages 1389–1397, 2017.

[119] D. Hebb. The organization of behavior; a neuropsychological theory. 1949.

[120] L. Heim. This can't go on (?)-ai training compute costs. june 1, 2023. June 2023. URL `https://blog.heim.xyz/this-cant-go-on-compute-training-costs/`. Accessed: 2024-10-21.

[121] B. Hendrickson and K. Devine. Dynamic load balancing in computational mechanics. *Computer methods in applied mechanics and engineering*, 184(2-4):485–500, 2000.

[122] J. L. Hennessy and D. A. Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[123] M. Heroux, R. Bartlett, V. H. R. Hoekstra, J. Hu, T. Kolda, R. Lehoucq, K. Long, R. Pawlowski, E. Phipps, A. Salinger, et al. An overview of trilinos. 2003.

[124] W. D. Hillis. *The connection machine*. MIT press, 1985.

[125] W. D. Hillis and G. L. Steele Jr. Data parallel algorithms. *Communications of the ACM*, 29 (12):1170–1183, 1986.

[126] G. Hinton, O. Vinyals, and J. Dean. Distilling the knowledge in a neural network. *stat*, 1050:9, 2015.

[127] S. Hochreiter. Long short-term memory. *Neural Computation MIT-Press*, 1997.

[128] S. Hochreiter. The vanishing gradient problem during learning recurrent neural nets and problem solutions. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 6 (02):107–116, 1998.

[129] 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(241):1–124, 2021.

[130] R. M. Hord. *The Illiac IV: the first supercomputer*. Springer Science & Business Media, 2013.

[131] H. Huang, H.-J. Song, and H.-K. Pao. Large language model pruning. *arXiv e-prints*, pages arXiv–2406, 2024.

[132] K. Huang, C. Xiao, L. M. Glass, M. Zitnik, and J. Sun. Skipgnn: predicting molecular interactions with skip-graph networks. *Scientific reports*, 10(1):1–16, 2020.

[133] L. Huawei Technologies Co. Huawei mindspore ai development framework. In *Artificial Intelligence Technology*, pages 137–162. Springer, 2022.

[134] M. R. Hugues and S. G. Petiton. Sparse matrix formats evaluation and optimization on a gpu. In *2010 IEEE 12th International Conference on High Performance Computing and Communications (HPCC)*, pages 122–129. IEEE, 2010.

[135] S. Huss-Lederman, E. M. Jacobson, J. R. Johnson, A. Tsao, and T. Turnbull. Strassen's algorithm for matrix multiplication: Modeling, analysis, and implementation. In *Proceedings of Supercomputing*, volume 96, pages 9–6. Citeseer, 1996.

[136] F. Hutter, L. Kotthoff, and J. Vanschoren. *Automated machine learning: methods, systems, challenges*. Springer Nature, 2019.

[137] C. Huyen. *AI Engineering: Building Applications with Foundation Models*. O'Reilly, 2025.

[138] A. Hyvärinen and E. Oja. Independent component analysis: algorithms and applications. *Neural networks*, 13(4-5):411–430, 2000.

[139] A. Ignatov, R. Timofte, A. Kulik, S. Yang, K. Wang, F. Baum, M. Wu, L. Xu, and L. Van Gool. Ai benchmark: All about deep learning on smartphones in 2019. In *2019 IEEE/CVF International Conference on Computer Vision Workshop (ICCVW)*, pages 3617–3635. IEEE, 2019.

[140] V. N. Ioannidis, A. G. Marques, and G. B. Giannakis. A recurrent graph neural network for multi-relational data. In *ICASSP 2019-2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 8157–8161. IEEE, 2019.

[141] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2704–2713, 2018.

[142] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*, pages 675–678, 2014.

[143] I. Jolliffe. Principal component analysis. *Encyclopedia of statistics in behavioral science*, 2005.

[144] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*, pages 1–12, 2017.

[145] R. Kaplan. Intel gaudi 3 ai accelerator: Architected for gen ai training and inference. In *2024 IEEE Hot Chips 36 Symposium (HCS)*, pages 1–16, 2024. doi: 10.1109/HCS61935.2024.10665178.

[146] M. J. Keeling and K. T. Eames. Networks and epidemic models. *Journal of the royal society interface*, 2(4):295–307, 2005.

[147] J. D. M.-W. C. Kenton and L. K. Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of naacL-HLT*, volume 1. Minneapolis, Minnesota, 2019.

[148] N. Ketkar, J. Moolayil, N. Ketkar, and J. Moolayil. Introduction to pytorch. *Deep learning with python: learn best practices of deep learning models with PyTorch*, pages 27–91, 2021.

[149] B. Khemani, S. Patil, K. Kotecha, and S. Tanwar. A review of graph neural networks: concepts, architectures, techniques, challenges, datasets, applications, and future directions. *Journal of Big Data*, 11(1):18, 2024.

[150] O. Khimich, A. Popov, and O. Chystiakov. Effective use of sparse matrices in problems of mathematical modeling. In *UkrPROG*, pages 212–221, 2022.

[151] T. N. Kipf and M. Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

[152] T. N. Kipf and M. Welling. Semi-Supervised Classification with Graph Convolutional Networks. In *Proceedings of the 5th International Conference on Learning Representations*, ICLR '17, 2017.

[153] N. Kishore Kumar and J. Schneider. Literature survey on low rank approximation of matrices. *Linear and Multilinear Algebra*, 65(11):2212–2244, 2017.

[154] J. M. Kleinberg et al. Authoritative sources in a hyperlinked environment. In *SODA*, volume 98, pages 668–677. Citeseer, 1998.

[155] M. Koklu, I. Cinar, and Y. S. Taspinar. Classification of rice varieties with deep learning methods. *Computers and electronics in agriculture*, 187:106285, 2021.

[156] I. Koren and C. M. Krishna. *Fault-tolerant systems*. Morgan Kaufmann, 2020.

[157] A. Krizhevsky, G. Hinton, et al. Learning multiple layers of features from tiny images. *Master's thesis, University of Tront*, 2009.

[158] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25:1097–1105, 2012.

[159] P. Kumar and R. Kumar. Issues and challenges of load balancing techniques in cloud computing: A survey. *ACM computing surveys (CSUR)*, 51(6):1–35, 2019.

[160] V. P. Kumar and A. Gupta. Analyzing scalability of parallel algorithms and architectures. *Journal of parallel and distributed computing*, 22(3):379–391, 1994.

[161] A. Kuzu and Y. Santur. Early diagnosis and classification of fetal health status from a fetal cardiotocography dataset using ensemble learning. *Diagnostics*, 13(15):2471, 2023.

[162] N. D. Lagaros. An efficient dynamic load balancing algorithm. *Computational Mechanics*, 53: 59–76, 2014.

[163] Z. Lai, Y. Hao, S. Li, and D. Li. Communication analysis for multidimensional parallel training of large-scale dnn models. In *2023 IEEE International Conference on High Performance Computing & Communications, Data Science & Systems, Smart City & Dependability in Sensor, Cloud & Big Data Systems & Application (HPCC/DSS/SmartCity/DependSys)*, pages 728–729. IEEE, 2023.

[164] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Transactions on Mathematical Software (TOMS)*, 5(3):308–323, 1979.

[165] Q. V. Le, M. Ranzato, R. Monga, M. Devin, G. Corrado, K. Chen, J. Dean, and A. Ng. Building high-level features using large scale unsupervised learning. *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 8595–8598, 2011. doi: 10.1109/ICASSP.2013.6639343.

[166] S. Le Grand, A. W. Götz, and R. C. Walker. Spfp: Speed without compromise—a mixed precision model for gpu accelerated molecular dynamics simulations. *Computer Physics Communications*, 184(2):374–380, 2013.

[167] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1 (4):541–551, 1989.

[168] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[169] Y. LeCun, L. Bottou, G. B. Orr, and K.-R. Müller. Efficient backprop. In *Neural networks: Tricks of the trade*, pages 9–50. Springer, 2002.

[170] J. H. Lee, B. Park, J. Kong, and A. Munir. Row-wise product-based sparse matrix multiplication hardware accelerator with optimal load balancing. *IEEE Access*, 10:64547–64559, 2022.

[171] R. B. Lehoucq, D. C. Sorensen, and C. Yang. *ARPACK users' guide: solution of large-scale eigenvalue problems with implicitly restarted Arnoldi methods*. SIAM, 1998.

[172] C. E. Leiserson. Fat-trees: Universal networks for hardware-efficient supercomputing. *IEEE transactions on Computers*, 100(10):892–901, 1985.

[173] R. Lempel and S. Moran. Salsa: the stochastic approach for link-structure analysis. *ACM Transactions on Information Systems (TOIS)*, 19(2):131–160, 2001.

[174] J. Leskovec and J. Mcauley. Learning to discover social circles in ego networks. *Advances in neural information processing systems*, 25, 2012.

[175] J. Leskovec, D. Huttenlocher, and J. Kleinberg. Signed networks in social media. In *Proceedings of the SIGCHI conference on human factors in computing systems*, pages 1361–1370, 2010.

[176] L. Leydesdorff and L. Vaughan. Co-occurrence matrices and their applications in information science: Extending aca to the web environment. *Journal of the American Society for Information Science and technology*, 57(12):1616–1628, 2006.

[177] C. Li. A software-hardware bridging model for simplifying parallel programming. *Thesis manuscript*, 07 2013.

[178] F. Li and S. Nath. Scalable data summarization on big data, 2014.

[179] Q. Li, Z. Han, and X.-M. Wu. Deeper insights into graph convolutional networks for semi-supervised learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 32, 2018.

[180] S. Li, M. A. Maddah-Ali, Q. Yu, and A. S. Avestimehr. A fundamental tradeoff between computation and communication in distributed computing. *IEEE Transactions on Information Theory*, 64(1):109–128, 2017.

[181] H. Liao, J. Tu, J. Xia, H. Liu, X. Zhou, H. Yuan, and Y. Hu. Ascend: a scalable and unified architecture for ubiquitous deep neural network computing: Industry track paper. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 789–801. IEEE, 2021.

[182] A. Liu, B. Feng, B. Xue, B. Wang, B. Wu, C. Lu, C. Zhao, C. Deng, C. Zhang, C. Ruan, et al. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*, 2024.

[183] T. Liu, N. L. Zhang, and P. Chen. Hierarchical latent tree analysis for topic detection. In *Machine Learning and Knowledge Discovery in Databases: European Conference, ECML PKDD 2014, Nancy, France, September 15-19, 2014. Proceedings, Part II 14*, pages 256–272. Springer, 2014.

[184] Z. Liu, N. Emad, S. B. Amor, and M. Lamure. Pagerank computation using a multiple implicitly restarted arnoldi method for modeling epidemic spread. *International Journal of Parallel Programming*, 43(6):1028–1053, 2015.

[185] A. Lohn. Scaling ai. Technical report, Technical report, Center for Security and Emerging Technology, 2023.

[186] Q. Lu, N. Damij, and J. Whalley. An exploration of high performance computing cloud contract for smes: a systematic literature review. *Digital Policy, Regulation and Governance*, 24(6): 525–540, 2022.

[187] A. Maćkiewicz and W. Ratajczak. Principal components analysis (pca). *Computers & Geosciences*, 19(3):303–342, 1993.

[188] C. D. Manning. *An introduction to information retrieval*. Cambridge university press, 2009.

[189] A. Maranjyan, E. M. Saad, P. Richtarik, and F. Orabona. Ata: Adaptive task allocation for efficient resource management in distributed machine learning. *arXiv preprint arXiv:2502.00775*, 2025.

[190] I. V. Marshakova. Bibliographic coupling system based on references. *Nauchno-Tekhnicheskaya Informatsiya Seriya, Ser*, 2(6):3–8, 1973.

[191] K. Matam, S. R. K. B. Indarapu, and K. Kothapalli. Sparse matrix-matrix multiplication on modern architectures. In *2012 19th International Conference on High Performance Computing*, pages 1–10. IEEE, 2012.

[192] O. A. McBryan. The connection machine: Pde solution on 65 536 processors. *Parallel Computing*, 9(1):1–24, 1988.

[193] I. Mehrez, O. Hamdi-Larbi, T. Dufaud, and N. Emad. Towards an auto-tuning system design for optimal sparse compression format selection with user expertise. In *2016 IEEE/ACS 13th International Conference of Computer Systems and Applications (AICCSA)*, pages 1–6. IEEE, 2016.

[194] I. Mehrez, O. Hamdi-Larbi, T. Dufaud, and N. Emad. Machine learning for optimal compression format prediction on multiprocessor platform. In *2018 International Conference on High Performance Computing & Simulation (HPCS)*, pages 213–220. IEEE, 2018.

[195] E. Merzbacher. Matrix methods in quantum mechanics. *American Journal of Physics*, 36(9): 814–821, 1968.

[196] P. Micikevicius, S. Narang, J. Alben, G. Diamos, E. Elsen, D. Garcia, B. Ginsburg, M. Houston, O. Kuchaiev, G. Venkatesh, et al. Mixed precision training. In *International Conference on Learning Representations*, 2018.

[197] S. Milgram. The small world problem. *Psychology today*, 2(1):60–67, 1967.

[198] G. Mitra, B. Johnston, A. P. Rendell, E. McCreath, and J. Zhou. Use of simd vector operations to accelerate application code performance on low-powered arm and intel platforms. In *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*, pages 1107–1116. IEEE, 2013.

[199] J. H. Moore. Bootstrapping, permutation testing and the method of surrogate data. *Physics in Medicine & Biology*, 44(6):L11, 1999.

[200] R. Morgan. On restarting the arnoldi method for large nonsymmetric eigenvalue problems. *Mathematics of Computation*, 65(215):1213–1230, 1996.

[201] S. Narang, G. Diamos, S. Sengupta, and E. Elsen. Exploring sparsity in recurrent neural networks. In *International Conference on Learning Representations*, 2022.

[202] D. Narayanan, M. Shoeybi, J. Casper, P. LeGresley, M. Patwary, V. Korthikanti, D. Vainbrand, P. Kashinkunti, J. Bernauer, B. Catanzaro, et al. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2021.

[203] M. Naumov, L. Chien, P. Vandermersch, and U. Kapasi. Cusparse library. In *GPU Technology Conference*, 2010.

[204] A. Newell, J. C. Shaw, and H. A. Simon. Report on a general problem solving program. In *IFIP congress*, volume 256, page 64. Pittsburgh, PA, 1959.

[205] L. H. Nguyen and S. Holmes. Ten quick tips for effective dimensionality reduction. *PLoS computational biology*, 15(6):e1006907, 2019.

[206] A. Nicola and C. Popa. Sparse matrix techniques in scientific computing. *Studies in Informatics and Control*, 18(1):33, 2009.

[207] C. Ordonez, N. Mohanam, and C. Garcia-Alvarado. Pca for large data sets with parallel data summarization. *Distributed and Parallel Databases*, 32:377–403, 2014.

[208] H. Ortega-Arranz, A. Gonzalez-Escribano, and D. R. Llanos. *The shortest-path problem: Analysis and comparison of methods*. Springer Nature, 2022.

[209] G. Ostrouchov. Parallel computing on a hypercube: an overview of the architecture and some applications. In *Computer Science and Statistics, Proceedings of the 19th Symposium on the Interface*, pages 27–32, 1987.

[210] B. Pang, E. Nijkamp, and Y. N. Wu. Deep learning with tensorflow: A review. *Journal of Educational and Behavioral Statistics*, 45(2):227–248, 2020.

[211] P. A. Papp, K. Martinkus, L. Faber, and R. Wattenhofer. Dropgnn: Random dropouts increase the expressiveness of graph neural networks. *Advances in Neural Information Processing Systems*, 34, 2021.

[212] R. Parihar. Branch prediction techniques and optimizations. *University of Rochester, NY, USA*, 2015.

[213] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

[214] 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.

[215] J. Pennington, R. Socher, and C. D. Manning. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 1532–1543, 2014.

[216] Q. Petit, C. Li, and N. Emad. An efficient and scalable approach to build co-occurrence matrix for dnn's embedding layer. In *Proceedings of the 38th ACM International Conference on Supercomputing*, pages 286–297, 2024.

[217] Q. R. Petit, C. Li, and N. Emad. Distributed and parallel sparse computing for very large graph neural networks. In *2022 IEEE International Conference on Big Data (Big Data)*, pages 6796–6798. IEEE, 2022.

[218] Q. R. Petit, C. Li, S. G. Petiton, K. Chai, and N. Emad. Enhancing graph convolutional networks by topology sampling. In *2022 IEEE International Conference on Big Data (Big Data)*, pages 1316–1323. IEEE, 2022.

[219] S. Petiton and C. Weill-Duflos. Massively parallel preconditioners for the sparse conjugate gradient method. In *International Conference on Vector and Parallel Processing*, pages 373–378. Springer, 1992.

[220] F. Petrini and M. Vanneschi. k-ary n-trees: High performance networks for massively parallel architectures. In *Proceedings 11th international parallel processing symposium*, pages 87–93. IEEE, 1997.

[221] E. Poisson, C. V. Gaudin, and P.-M. Lallican. Multi-modular architecture based on convolutional neural networks for online handwritten character recognition. In *Proceedings of the 9th International Conference on Neural Information Processing, 2002. ICONIP'02.*, volume 5, pages 2444–2448. IEEE, 2002.

[222] L. Prechelt. Early stopping-but when? In *Neural Networks: Tricks of the trade*, pages 55–69. Springer, 2002.

[223] J. K. Pritchard, M. Stephens, and P. Donnelly. Inference of population structure using multilocus genotype data. *Genetics*, 155(2):945–959, 2000.

[224] H. Racke. Minimizing congestion in general networks. In *The 43rd Annual IEEE Symposium on Foundations of Computer Science, 2002. Proceedings.*, pages 43–52. IEEE, 2002.

[225] A. Radford, K. Narasimhan, T. Salimans, and I. Sutskever. Improving language understanding by generative pre-training. 2018.

[226] S. Raschka, J. Patterson, and C. Nolet. Machine learning in python: Main developments and technology trends in data science, machine learning, and artificial intelligence. *Information*, 11(4), 2020. ISSN 2078-2489. doi: 10.3390/info11040193. URL https://www.mdpi.com/2078-2489/11/4/193.

[227] P. Ray, S. S. Reddy, and T. Banerjee. Various dimension reduction techniques for high dimensional data analysis: a review. *Artificial Intelligence Review*, 54(5):3473–3515, 2021.

[228] A. Reuther, P. Michaleas, M. Jones, V. Gadepally, S. Samsi, and J. Kepner. Survey of machine learning accelerators. In *2020 IEEE high performance extreme computing conference (HPEC)*, pages 1–12. IEEE, 2020.

[229] A. Reuther, P. Michaleas, M. Jones, V. Gadepally, S. Samsi, and J. Kepner. Ai and ml accelerator survey and trends. In *2022 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–10. IEEE, 2022.

[230] E. M. Rogers, A. Singhal, and M. M. Quinlan. Diffusion of innovations. In *An integrated approach to communication theory and research*, pages 432–448. Routledge, 2014.

[231] Y. Rong, W. Huang, T. Xu, and J. Huang. Dropedge: Towards deep graph convolutional networks on node classification. In *International Conference on Learning Representations*, 2019.

[232] F. Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[233] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by error propagation. Technical report, California Univ San Diego La Jolla Inst for Cognitive Science, 1985.

[234] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

[235] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, et al. Imagenet large scale visual recognition challenge. *International journal of computer vision*, 115:211–252, 2015.

[236] Y. Saad. Variations on arnoldi's method for computing eigenelements of large unsymmetric matrices. *Linear algebra and its applications*, 34:269–295, 1980.

[237] Y. Saad. Sparskit: A basic tool kit for sparse matrix computations. 1990.

[238] T. N. Sainath, B. Kingsbury, V. Sindhwani, E. Arisoy, and B. Ramabhadran. Low-rank matrix factorization for deep neural network training with high-dimensional output targets. In *2013 IEEE international conference on acoustics, speech and signal processing*, pages 6655–6659. IEEE, 2013.

[239] A. L. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of research and development*, 3(3):210–229, 1959.

[240] R. K. Sato and P. N. Swarztrauber. Benchmarking the connection machine 2. In *Supercomputing'88: Proceedings of the 1988 ACM/IEEE Conference on Supercomputing, Vol. I*, pages 304–309. IEEE, 1988.

[241] R. Segal, C. Avin, and G. Scalosub. Constrained in-network computing with low congestion in datacenter networks. In *IEEE INFOCOM 2022-IEEE Conference on Computer Communications*, pages 1639–1648. IEEE, 2022.

[242] F. Seide, G. Li, D. Yu, et al. Conversational speech transcription using context-dependent deep neural networks. In *Interspeech*, pages 437–440, 2011.

[243] O. Selvitopi, S. Ekanayake, G. Guidi, G. A. Pavlopoulos, A. Azad, and A. Buluç. Distributed many-to-many protein sequence alignment using sparse matrices. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14. IEEE, 2020.

[244] J. Sevilla and E. Roldán. Training compute of frontier ai models grows by 4–5x per year. *Epoch AI, May*, 28, 2024.

[245] J. Sevilla, T. Besiroglu, O. Dudney, and A. Ho. The longest training run, 2022. URL `https://epochai.org/blog/the-longest-training-run`. Accessed: 2024-10-21.

[246] J. Sevilla, L. Heim, A. Ho, T. Besiroglu, M. Hobbhahn, and P. Villalobos. Compute trends across three eras of machine learning. In *2022 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 2022.

[247] S. Shahzadeh Fazeli, N. Emad, and Z. Liu. A key to choose subspace size in implicitly restarted arnoldi method. *Numerical Algorithms*, 70:407–426, 2015.

[248] S. Shalev-Shwartz and S. Ben-David. *Understanding machine learning: From theory to algorithms*. Cambridge university press, 2014.

[249] N. Shazeer, A. Mirhoseini, K. Maziarz, A. Davis, Q. Le, G. Hinton, and J. Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *arXiv preprint arXiv:1701.06538*, 2017.

[250] Y. Shen, G. Chen, H. Jagadish, W. Lu, B. C. Ooi, and B. M. Tudor. Fast failure recovery in distributed graph processing systems. *Proceedings of the VLDB Endowment*, 8(4):437–448, 2014.

[251] S. Sidana, C. Laclau, M. R. Amini, G. Vandelle, and A. Bois-Crettez. Kasandr: a large-scale dataset with implicit feedback for recommendation. In *Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 1245–1248, 2017.

[252] L. Silva and R. Buyya. *Parallel Programming Models and Paradigms*, pages 4 – 27. Prentice Hall, Australia, 1999. ISBN 0130137855.

[253] D. C. Sorensen. Implicit application of polynomial filters in ak-step arnoldi method. *Siam journal on matrix analysis and applications*, 13(1):357–385, 1992.

[254] D. C. Sorensen. Implicitly restarted arnoldi/lanczos methods for large scale eigenvalue calculations. In *Parallel Numerical Algorithms*, pages 119–165. Springer, 1997.

[255] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15 (1):1929–1958, 2014.

[256] S. Stanton, P. Izmailov, P. Kirichenko, A. A. Alemi, and A. G. Wilson. Does knowledge distillation really work? *Advances in Neural Information Processing Systems*, 34:6906–6919, 2021.

[257] T. Sterling, M. Brodowicz, and M. Anderson. *High performance computing: modern systems and practices*. Morgan Kaufmann, 2017.

[258] G. W. Stewart. *Matrix algorithms: volume 1: basic decompositions*. SIAM, 1998.

[259] M. Stoll. A literature survey of matrix methods for data science. *GAMM-Mitteilungen*, 43(3): e202000013, 2020.

[260] J. V. Stone. Independent component analysis: an introduction. *Trends in cognitive sciences*, 6 (2):59–64, 2002.

[261] V. Strassen. Gaussian elimination is not optimal. *Numerische mathematik*, 13(4):354–356, 1969.

[262] K. Streit, J. Doerfert, C. Hammacher, A. Zeller, and S. Hack. Generalized task parallelism. *ACM Transactions on Architecture and Code Optimization (TACO)*, 12(1):1–25, 2015.

[263] E. Strubell, A. Ganesh, and A. McCallum. Energy and policy considerations for modern deep learning research. In *Proceedings of the AAAI conference on artificial intelligence*, volume 34, pages 13693–13696, 2020.

[264] M. Sun, Z. Liu, A. Bair, and J. Z. Kolter. A simple and effective pruning approach for large language models. In *Workshop on Efficient Systems for Foundation Models@ ICML2023*.

[265] M. Svanberg, W. Li, M. Fleming, B. Goehring, and N. Thompson. Beyond ai exposure: Which tasks are cost-effective to automate with computer vision? *Available at SSRN 4700751*, 2024.

[266] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2818–2826, 2016.

[267] G. Tamburrini. The ai carbon footprint and responsibilities of ai scientists. *Philosophies*, 7(1): 4, 2022.

[268] S. Tan and M. L. Mayrovouniotis. Reducing data dimensionality through optimizing neural network inputs. *AIChE Journal*, 41(6):1471–1480, 1995.

[269] G. Team, R. Anil, S. Borgeaud, Y. Wu, J.-B. Alayrac, J. Yu, R. Soricut, J. Schalkwyk, A. M. Dai, A. Hauth, et al. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805*, 2023.

[270] J. B. Tenenbaum, V. d. Silva, and J. C. Langford. A global geometric framework for nonlinear dimensionality reduction. *science*, 290(5500):2319–2323, 2000.

[271] N. Thompson, K. Greenewald, K. Lee, and G. F. Manso. The computational limits of deep learning. In *Ninth Computing within Limits 2023*. LIMITS, 2023.

[272] J. Travers and S. Milgram. An experimental study of the small world problem. In *Social networks*, pages 179–197. Elsevier, 1977.

[273] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, Aug. 1990. ISSN 0001-0782. doi: 10.1145/79173.79181. URL `https://doi.org/10.1145/79173.79181`.

[274] L. Van der Maaten and G. Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(11), 2008.

[275] M. Vandromme, J. Gurhem, M. Tsuji, S. Petiton, and M. Sato. Scaling the pagerank algorithm for very large graphs on the fugaku supercomputer. In *International Conference on Computational Science*, pages 389–402. Springer, 2022.

[276] A. Vaswani. Attention is all you need. *Advances in Neural Information Processing Systems*, 2017.

[277] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio. Graph attention networks. *arXiv preprint arXiv:1710.10903*, 2017.

[278] S. Velliangiri, S. Alagumuthukrishnan, et al. A review of dimensionality reduction techniques for efficient computation. *Procedia Computer Science*, 165:104–111, 2019.

[279] P. Vincent, H. Larochelle, I. Lajoie, Y. Bengio, P.-A. Manzagol, and L. Bottou. Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion. *Journal of machine learning research*, 11(12), 2010.

[280] V. Vlasyuk, O. Galchonkov, and A. Nevrev. An algebraic method for calculating pagerank. *Eastern-European Journal of Enterprise Technologies*, 3(2 (93)):6–12, May 2018. doi: 10.15587/1729-4061.2018.131275. URL `http://journals.uran.ua/eejet/article/view/131275`.

[281] L. Wasserman. *All of statistics: a concise course in statistical inference*. Springer Science & Business Media, 2013.

[282] J. Wei, Y. Tay, R. Bommasani, C. Raffel, B. Zoph, S. Borgeaud, D. Yogatama, M. Bosma, D. Zhou, D. Metzler, E. Chi, T. Hashimoto, O. Vinyals, P. Liang, J. Dean, and W. Fedus. Emergent abilities of large language models. *ArXiv*, abs/2206.07682, 2022. doi: 10.48550/arXiv.2206.07682.

[283] X. Wei, Y. Zhang, X. Zhang, R. Gong, S. Zhang, Q. Zhang, F. Yu, and X. Liu. Outlier suppression: Pushing the limit of low-bit transformer language models. *Advances in Neural Information Processing Systems*, 35:17402–17414, 2022.

[284] H. Wen, Y. Lin, X. Mao, F. Wu, Y. Zhao, H. Wang, J. Zheng, L. Wu, H. Hu, and H. Wan. Graph2route: A dynamic spatial-temporal graph neural network for pick-up and delivery route prediction. In *Proceedings of the 28th ACM SIGKDD conference on knowledge discovery and data mining*, pages 4143–4152, 2022.

[285] D. B. West et al. *Introduction to graph theory*, volume 2. Prentice hall Upper Saddle River, 2001.

[286] K. Wiatr. Median and morphological specialized processors for a real-time image data processing. *EURASIP Journal on Advances in Signal Processing*, 2002:1–7, 2002.

[287] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and S. Y. Philip. A comprehensive survey on graph neural networks. *IEEE transactions on neural networks and learning systems*, 32(1): 4–24, 2020.

[288] G. Xiao, J. Lin, M. Seznec, H. Wu, J. Demouth, and S. Han. Smoothquant: Accurate and efficient post-training quantization for large language models. In *International Conference on Machine Learning*, pages 38087–38099. PMLR, 2023.

[289] K. Xu, C. Li, Y. Tian, T. Sonobe, K.-i. Kawarabayashi, and S. Jegelka. Representation learning on graphs with jumping knowledge networks. In *International Conference on Machine Learning*, pages 5453–5462. PMLR, 2018.

[290] M. Xu. Understanding graph embedding methods and their applications. *SIAM Review*, 63(4): 825–853, 2021.

[291] W. S. Yambor, B. A. Draper, and J. R. Beveridge. Analyzing pca-based face recognition algorithms: Eigenvector selection and distance measures. In *Empirical evaluation methods in computer vision*, pages 39–60. World Scientific, 2002.

[292] H. Yang, S. Gui, Y. Zhu, and J. Liu. Automatic neural network compression by sparsity-quantization joint learning: A constrained optimization-based approach. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 2178–2188, 2020.

[293] H. Yang, D. Lin, and Q. Li. An efficient greedy search algorithm for high-dimensional linear discriminant analysis. *Statistica Sinica*, 33(SI):1343, 2023.

[294] L. Yang, F. Wu, Y. Wang, J. Gu, and Y. Guo. Masked graph convolutional network. In *IJCAI*, pages 4070–4077, 2019.

[295] Y. Yao, L. Rosasco, and A. Caponnetto. On early stopping in gradient descent learning. *Constructive Approximation*, 26(2):289–315, 2007.

[296] M. YOUNG and N. R. L. W. DC. Benchmarking the connection machine. Technical report, Technical Report R. A87-4Naval Research Laboratory, 1990.

[297] B. Yu, H. Yin, and Z. Zhu. Spatio-temporal graph convolutional networks: A deep learning framework for traffic forecasting. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence*. International Joint Conferences on Artificial Intelligence Organization, 2018.

[298] R. Yu, A. Li, C.-F. Chen, J.-H. Lai, V. I. Morariu, X. Han, M. Gao, C.-Y. Lin, and L. S. Davis. Nisp: Pruning networks using neuron importance score propagation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 9194–9203, 2018.

[299] C. Zhang, S. Bengio, M. Hardt, B. Recht, and O. Vinyals. Understanding deep learning (still) requires rethinking generalization. *Communications of the ACM*, 64:107 – 115, 2021. doi: 10.1145/3446776.

[300] T. Zhao, Y. Liu, L. Neves, O. Woodford, M. Jiang, and N. Shah. Data augmentation for graph neural networks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 11015–11023, 2021.

[301] C. Zheng, B. Zong, W. Cheng, D. Song, J. Ni, W. Yu, H. Chen, and W. Wang. Robust graph representation learning via neural sparsification. In *International Conference on Machine Learning*, pages 11458–11468. PMLR, 2020.

[302] J. Zhou, G. Cui, S. Hu, Z. Zhang, C. Yang, Z. Liu, L. Wang, C. Li, and M. Sun. Graph neural networks: A review of methods and applications. *AI Open*, 1:57–81, 2020.

[303] M. Zhou, N. Duan, S. Liu, and H.-Y. Shum. Progress in neural nlp: modeling, learning, and reasoning. *Engineering*, 6(3):275–290, 2020.

[304] B. Zoph and Q. Le. Neural architecture search with reinforcement learning. In *International Conference on Learning Representations*, 2016.

université PARIS-SACLAY

ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

Titre: Calcul réparti et parallèle pour les réseaux de neurones de très grandes tailles
Mots clés: Algorithme de recommandation, Réseaux de neurones, Calcul de graphe, Calcul haute performance, Optimisation de performance, Apprentissage profond

Résumé:

Les modèles de très grande taille sont aujourd'hui utilisés dans une multitude de domaines variés et ont permis de généraliser et de populariser l'utilisation du Deep Learning pour de nouvelles applications. Cependant, le traitement de ces tâches toujours plus générales a demandé une augmentation exponentielle de la taille de ces modèles, ce qui a également nécessité une puissance de calcul tout aussi importante pour les entraîner. Des solutions innovantes doivent être trouvées et déployées pour à la fois réduire la complexité des algorithmes existants et améliorer le déploiement de ces derniers dans un environnement massivement distribué avec des données de très grande taille. Le développement de techniques et de méthodes de calcul parallèle et distribué est essentiel pour optimiser l'utilisation des ressources disponibles, maximiser l'efficacité et réduire les coûts de calcul, répondant ainsi aux exigences croissantes de ces modèles.

C'est dans ce contexte que s'inscrit cette thèse. Nous proposons plusieurs contributions pour réduire les coûts associés à l'entraînement des grands réseaux de neurones dans un environnement massivement distribué. Nos travaux se concentrent principalement sur le traitement des données en amont du modèle pour améliorer la qualité des données, qui sont ensuite données en entrée du modèle, dans le but de faciliter son apprentissage. Nous nous sommes concentrés sur le traitement des données creuses, telles que les graphes, dont le traitement pose certains défis en raison de leurs structures complexes et de leurs tailles potentiellement très élevées. Nous proposons également de réduire la taille des données en entrée grâce à une réduction de dimension conservant suffisamment d'informations pour assurer une bonne précision du modèle tout en simplifiant les données d'entrée, réduisant par la même occasion la puissance de calcul nécessaire pour le traitement de ces données.

Title: Distributed and Parallel Computing for very Large Neural Networks
Keywords: Performance optimization, Deep Learning, High performance computing, Neural networks, Graph computing, Recommender system

Abstract:

Very large model sizes are now a very common feature, extending the range of applications for Deep Learning. However, this exponential growth in model size has led to an equally significant increase in computing power requirements. Innovative solutions need to be found and implemented to optimize current algorithms, reduce their complexity and make them easy to use and deploy in a massively distributed environment. The development of parallel and distributed computing techniques and methods to fully exploit available resources is crucial to maximizing efficiency and minimizing computation costs is very important to meet the ever-growing requirements of these models.

In this context, we propose several contributions to reduce the costs associated with the training of neural networks in a massively distributed environment. Our contributions focus on the processing of data upstream of the model, in order to improve the quality of the data supplied to the neural network and facilitate its training. We focused on the processing of sparse data, such as graphs, which pose particular challenges due to their complex structures and potentially very large sizes. The processing applied to these data are designed to significantly improve the model's performance. Finally, we propose leveraging this processing to reduce effectively the size of the data, thereby decreasing the number of inputs while retaining sufficient information to ensure good model accuracy.