Programa de doctorado en Informática

Escuela de Doctorado de la Universitat Jaume I

Análisis y modelización de los procesos de aprendizaje profundo

---

Memoria presentada por Mar Catalán Carbó para optar al grado de doctora por la Universitat Jaume I.

Director:
Enrique S. Quintana-Ortí

Director:
Manuel F. Dolz Zaragozá

Doctoranda:
Mar Catalán Carbó

Castelló de la Plana, 20 Febrero 2024

Financiación y Licencia de Uso

Financiación La autora de este trabajo ha recibido apoyo económico del Ministerio de Ciencia, Innovación y Universidades a través de una ayuda para contratos predoctorales para la formación de doctores contemplada en el Subprograma Estatal de Formación del Programa Estatal de Promoción del Talento y su Empleabilidad en I+D+i, en el marco del Plan Estatal de Investigación Científica y Técnica y de Innovación 2017-2020. Dicha ayuda, con referencia PRE2018-084865, se durante el periodo comprendido entre el 01/07/2019 y el 30/06/2023.

Resumen

Actualmente, gracias a la inmensa cantidad de datos accesibles, la inteligencia artificial está permitiendo inferir nuevo conocimiento y realizar avances espectaculares. Particularmente, el aprendizaje automático que ofrecen las redes neuronales profundas se encuentra en un momento de grandes progresos. No obstante, estos avances vienen acompañados de la necesidad de mejorar las técnicas, el análisis y el uso de estas redes y de los procesos que conllevan a fin de reducir sus costes.

El objetivo principal sobre el que se ha desarrollado esta tesis consiste en el diseño, implementación y validación experimental de soluciones software paralelas para el aprendizaje automático mediante redes neuronales profundas. Dentro de esta línea, el trabajo de investigación que aquí se presenta aborda tres diferentes vertientes que se han ido derivando de las conclusiones obtenidas a lo largo de los estudios realizados. En primer lugar se desarrolla un entorno de simulación para redes neuronales válido para la implementación de estrategias que mejoren el rendimiento del proceso de entrenamiento, como es el uso de esquemas de paralelismo, y que sea apto para su uso en plataformas distribuidas, como clústeres de computadores equipados con GPU. Seguidamente se trabajan métodos para estimar los tiempos de ejecución de los diferentes procesos relativos a las redes neuronales profundas. En este sentido, se presentan dos modelos que permiten predecir el coste de cada operación involucrada, tanto en el entrenamiento como en la inferencia, y se realiza en base a estos modelos un estudio de escalabilidad del proceso de entrenamiento distribuido. Por último, como consecuencia del cuello de botella observado que

suponen las comunicaciones durante el entrenamiento distribuido, se realiza un análisis profundo de la primitiva *AllReduce* en el que se estudian los diferentes algoritmos de implementación y la idoneidad de estos para cada escenario, proponiendo modelos alternativos que mejoran el ajuste de los propuestos en la literatura.

Abstract

Today, artificial intelligence is able to derive new knowledge and make spectacular progress thanks to the immense amount of data available. In particular, the machine learning offered by deep neural networks is making a great deal of progress. However, this progress is accompanied by the need to improve the techniques, analysis and use of these networks and the processes involved in order to reduce their costs.

The main objective of this thesis is the design, implementation and experimental validation of parallel software solutions for machine learning using deep neural networks. Within this line of research, the work presented here addresses three different aspects that have been derived from the conclusions of the studies carried out. Firstly, a simulation environment for neural networks is developed that is valid for the implementation of strategies that improve the performance of the training process, such as the use of parallelism schemes, and that is suitable for use on distributed platforms, such as clusters of computers equipped with GPUs. Then, methods for the estimation of the execution times of the different processes in the context of deep neural networks will be worked on. In this sense, two models are presented that allow predicting the cost of each operation involved, both in training and in inference, and a scalability study of the distributed training process is carried out on the basis of these models. Finally, as a consequence of the communication bottleneck observed in distributed training, an in-depth analysis of the *AllReduce* primitive is carried out. The different implementation algorithms and their suitability for each scenario are studied, and alternative models are proposed that improve the suitability of those proposed in the literature.

Índice general

Índice de figuras xi

Índice de tablas xv

1. Introducción 1

1.1. Motivación 1
1.2. Objetivos 4
1.3. Estructura de la Tesis 5

1. Introduction 7

1.1. Motivation 7
1.2. Objectives 9
1.3. Structure of the thesis 10

2. Revisión de las Redes Neuronales 13

2.1. Conceptos Básicos 13
2.2. Paso *forward* 19
2.3. Paso *backward* 25
2.4. Entrenamiento por Lotes de Muestras 29
2.5. Aceleración del Procesamiento en las Redes Neuronales 33
2.5.1. Paralelismo de datos 34
2.5.2. Paralelismo de modelo 36
2.5.3. Paralelismo *pipeline* 38
2.5.4. Paralelismo híbrido 39
2.6. Estado del Arte 40

3. Entorno Paralelo de Entrenamiento e Inferencia de Redes Neuronales 43
3.1. Motivación . . . 43
3.2. Una Breve Visión de PyDTNN . . . 44
3.3. Clases y Métodos . . . 46
3.4. Extensibilidad . . . 51
3.4.1. Aceleración mediante tipos de datos no convencionales . . 51
3.4.2. Reducción del coste aritmético utilizando poda . . . 53
3.4.3. Operadores alternativos más eficientes . . . 54
3.5. Paralelismo en PyDTNN . . . 55
3.5.1. Implementación . . . 55
3.5.2. Evaluación . . . 60
3.6. Conclusiones . . . 65

4. Modelado del Entrenamiento 67
4.1. Motivación . . . 67
4.2. Coste del Cómputo . . . 69
4.2.1. Modelo analítico . . . 70
4.2.2. Modelo con redes de tipo MLP . . . 73
4.3. Coste de las Comunicaciones . . . 76
4.4. Modelo Analítico . . . 79
4.4.1. Validación . . . 79
4.5. Modelo con redes de tipo MLP . . . 81
4.5.1. Validación . . . 82
4.6. Análisis de Rendimiento y Escalabilidad del Paralelismo de Datos 88
4.7. Conclusiones . . . 92

5. Análisis de la Primitiva *AllReduce* 95
5.1. Introducción . . . 95
5.2. Análisis del Coste de la Primitiva *AllReduce* . . . 97
5.2.1. Estudio de las posibles causas . . . 99
5.3. Mejora de las Estimaciones . . . 103
5.3.1. Ancho de banda de enlace . . . 104
5.3.2. Ancho de banda de memoria . . . 104

5.3.3. Asincronía 107
5.3.4. Resultados 108
5.4. Elección del Algoritmo 112
5.4.1. Evaluación de los diferentes algoritmos 113
5.4.2. Evaluación de las diferentes bibliotecas 115
5.4.3. Impacto sobre el entrenamiento de RNP 117
5.5. Conclusiones 121

6. Conclusiones **125**
6.1. Conclusiones y Principales Contribuciones 125
6.2. Publicaciones 128
6.2.1. Artículos en revista 128
6.2.2. Artículos en congresos 130
6.3. Líneas de Investigación Abiertas 133

6. Conclusions **137**
6.1. Conclusions and Main Contributions 137
6.2. Publications 140
6.2.1. Journal articles 140
6.2.2. Artículos en congresos 142
6.3. Open Lines of Research 145

A. Algoritmos del *AllReduce* **149**

Bibliografía **155**

Glosario **165**

Índice de figuras

1.1. Evolución del número de parámetros de las redes neuronales profundas. Figura tomada de Bernstein, L., Sludds, A., Hamerly, R. et al. "Freely scalable and reconfigurable optical hardware for deep learning". Sci Rep 11, 3144 (2021). https://doi.org/10.1038/s41598-021-82543-3. Publicado bajo licencia http://creativecommons.org/licenses/by/4.0/. 2

1.1. Evolution of the number of parameters of deep neural networks. Figure taken from Bernstein, L., Sludds, A., Hamerly, R. et al. "Freely scalable and reconfigurable optical hardware for deep learning". Sci Rep 11, 3144 (2021). https://doi.org/10.1038/s41598-021-82543-3. Published under license http://creativecommons.org/licenses/by/4.0/. 8

2.1. Conexiones de una neurona. 14
2.2. Red Neuronal de ejemplo. 15
2.3. Red de la Figura 2.2 con las transiciones del paso *forward*. 20
2.4. Paso *forward*. 22
2.5. Ejemplo de convolución utilizando *im2col*. 24
2.6. Operación convolucional del paso *forward* como un producto de matrices. 24
2.7. Cálculo del gradiente durante el paso *backward*. 27
2.8. Actualización de pesos durante el paso *backward*. 28
2.9. Operaciones convolucionales del paso *backward* como productos de matrices. 28

ÍNDICE DE FIGURAS

2.10. Operaciones del entrenamiento por lotes para capas FC. 32
2.11. Operaciones del entrenamiento por lotes para capas convolucionales utilizando productos de matrices. 32
2.12. Grafo de dependencias en el entrenamiento de una red neuronal. . 33
2.13. Aprovechamiento del paralelismo de datos con dos réplicas en el paso *forward* de una red neuronal. 34
2.14. División entre P réplicas del entrenamiento de una red neuronal usando paralelismo de datos. 36
2.15. División entre P réplicas del entrenamiento de una red neuronal usando paralelismo de modelo. 37
2.16. Distribución entre P réplicas del entrenamiento de una red neuronal usando paralelismo de modelo. 38
2.17. Esquema secuencial y paralelizado mediante *pipelining*. 39

3.1. Esquema de la arquitectura de PyDTNN. 45
3.2. Versiones del paralelismo de datos con solapamiento de comunicaciones y cómputo. 58
3.3. Rendimiento de los entornos PyDTNN y TF+HVD en función del número de nodos/GPU para diferentes redes 61
3.4. Convergencia del entrenamiento y la validación de diferentes redes utilizando TF+HVD y PyDTNN. 64

4.1. Rendimiento alcanzable según el modelo Roofline. 73
4.2. Tiempos estimados y reales de un paso FP+BP de la red VGG11. 81
4.3. Histogramas y gráficos de cajas de los errores relativos cometidos en las estimaciones realizadas para cada núcleo por las redes de tipo MLP, usando las GPU NVIDIA Tesla A100/V100. 84
4.4. Tiempos reales y esperados de cómputo por cada núcleo invocado durante el entrenamiento (paso *forward*, cálculo del gradiente y actualización de pesos) de la red VGG11. 86
4.5. Tiempos reales y esperados para la comunicación colectiva *AllReduce* para 6 y 8 procesos. 88

4.6. Tiempo de ejecución estimado para un paso de entrenamiento *forward-backward* de un lote de la base de datos CIFAR-10, utilizando diferentes redes neuronales (columnas: VGG16, ResNet50 y Densenet-121) y variando los diferentes parámetros que influyen (filas: rendimiento, ancho de banda de memoria, ancho de banda de enlace, número de nodos y tamaño del lote). 90
4.7. Tiempo de ejecución estimado para un paso de entrenamiento *forward-backward* de un lote de la base de datos ImageNet, utilizando diferentes redes neuronales (columnas: VGG16, ResNet50 y Densenet-121) y variando los diferentes parámetros que influyen. 91

5.1. Rendimientos teóricos de la primitiva *AllReduce*. 97
5.2. Rendimiento real y teórico (-T) de la primitiva *AllReduce* en Open-MPI utilizando 7 y 8 nodos (izquierda y derecha respectivamente). 98
5.3. Rendimiento punto a punto de una red medido mediante una prueba *ping-pong*. 100
5.4. Tiempos de ejecución y estimaciones suponiendo comunicaciones síncronas y asíncronas para los algoritmos RDB, RSA y RNG de la primitiva *AllReduce*. 102
5.5. Ancho de banda de enlace alcanzado en experimentos *ping-pong*. . 105
5.6. Ancho de banda de memoria alcanzado en experimentos de suma de vectores. 105
5.7. Tiempos de ejecución y errores relativos para el algoritmo LIN con 8 y 15 procesos. 109
5.8. Tiempos de ejecución y errores relativos para el algoritmo RDB con 8 y 15 procesos. 109
5.9. Tiempos de ejecución y errores relativos para el algoritmo RNG con 8 y 15 procesos. 110
5.10. Tiempos de ejecución y errores relativos para el algoritmo RSA con 8 y 15 procesos. 110
5.11. Rendimientos de los algoritmos *AllReduce* disponibles en MPICH (arriba), Open-MPI (centro) e Intel-MPI (abajo), utilizando 7 y 8 procesos (izquierda y derecha, respectivamente). 114

5.12. Rendimiento de los algoritmos que mejor rendimiento ofrecen para cada caso (BEST) y del escogido por defecto (AUTO) por la biblioteca, utilizando Open-MPI, MPICH e Intel-MPI y comparando el uso de 7 y 8 procesos (izquierda y derecha respectivamente). . . 115
5.13. Número de mensajes y tamaño de estos en el entrenamiento de los escenarios planteados. 118
5.14. Rendimiento en imágenes por segundo del entrenamiento utilizando 8 procesos y tamaño de lote $b = 16$ (izquierda) y $b = 32$ (derecha). 118
5.15. Aceleración (Speed-up) del rendimiento en TF+Horovod con un único proceso MPI por nodo. Utilizando $b = 16$ (arriba), 32 (centro), y 64 (abajo). Los resultados se han normalizado respecto al algoritmo AUTO de cada biblioteca (izquierda) y respecto del algoritmo AUTO escogido por Intel-MPI (derecha). 120

6.1. Ejemplo de reorganización de una matriz (izq.) para tener bloques con mayor dependencia lineal (derecha). 135

6.1. Example of reorganizing a matrix (left) to have blocks with greater linear dependence (right). 147

A.1. Esquema del algoritmo *basic linear* de la primitiva *AllReduce*. . . 150
A.2. Esquema del algoritmo *ring* de la primitiva *AllReduce*. 151
A.3. Esquema del algoritmo *Rabenseifner* de la primitiva *AllReduce*. . . 152
A.4. Esquema del algoritmo *recursive doubling* de la primitiva *AllReduce*. 153

Índice de tablas

4.1. Parámetros del modelo. 69
4.2. Dimensión de los operandos en una capa l FC. 71
4.3. Dimensión de los operandos en una capa l convolucional. 72
4.4. Esquema del cómputo de operaciones por capa. 72
4.5. Núcleos asociados al entrenamiento de cada tipo de capa. 75
4.6. Esquema de las redes de tipo MLP utilizadas para la modelización del coste temporal de los núcleos de la Tabla 4.5. 76
4.7. Conjunto de datos de entrada para las redes MLP de estimación. . 77
4.8. Desglose de costes de la primitiva Allreduce. 79

5.1. Errores relativos respecto del coste real cuando la tendencia se estabiliza. 112

Capítulo 1

Introducción

1.1. Motivación

La ingente cantidad de datos de diferente naturaleza que se recogen continuamente ha dado lugar recientemente a un cuarto paradigma de exploración científica, basado en el análisis de enormes volúmenes de datos (*Big Data*), frente a las tres aproximaciones tradicionales, respectivamente fundamentadas en la formulación teórica, la experimentación práctica y la simulación por computador. Este nuevo paradigma requiere un uso eficiente de técnicas de manipulación, análisis y visualización de los datos, a menudo mediante computadores masivamente paralelos y técnicas de computación de altas prestaciones (*High Performance Computing* o HPC) [35, 63]. La cuestión en este escenario es cómo transformar esta marea de datos en información y conocimiento útiles [23, 28].

La Inteligencia Artificial (IA) proporciona un medio efectivo para inferir nuevo conocimiento a partir de los datos. En consecuencia, resulta natural que actualmente la IA se esté aplicando de manera intensiva para procesar los inmensos volúmenes de datos obtenidos de las redes sociales, así como aquellos menos numerosos pero igualmente relevantes que provienen de aplicaciones científicas e industriales [10, 67]. En particular, en el periodo más reciente, el aprendizaje automático (*Machine Learning* o ML) mediante redes neuronales profundas (RNP), también conocido como aprendizaje profundo (*Deep Learning* o DL), ha producido avances espectaculares en aplicaciones de muy distinta naturaleza. Así, un buen número de casos de éxito para el aprendizaje profundo sigue estando presente

en nichos de aplicación tradicionales para estas tecnologías, como en la clasificación de imágenes, el reconocimiento del lenguaje y la traducción automática [44]. Además, el ámbito de utilización de estas tecnologías se ha extendido en estos últimos años a un abanico mucho más variado de problemas, por ejemplo, en el estudio de nuevos materiales, simulaciones en astrofísica, detección de fraudes en banca, prevención de fallos en entornos industriales, sistemas de recomendación y predicción de series temporales, entre muchas otros [3, 11, 19, 54, 67, 68, 78].

En gran medida el renovado interés por el aprendizaje profundo se remonta al trabajo de Yann LeCun, Yoshua Bengio y Geoffrey Hinton en la década pasada [49]. En los años siguientes, el interés de la comunidad científica, así como de la industria, ha abierto la puerta al diseño de algoritmos de aprendizaje profundo cada vez más sofisticados, a la par que modelos basados en RNP cada vez más complejos, como se muestra en la evolución reflejada en la Figura 1.1.

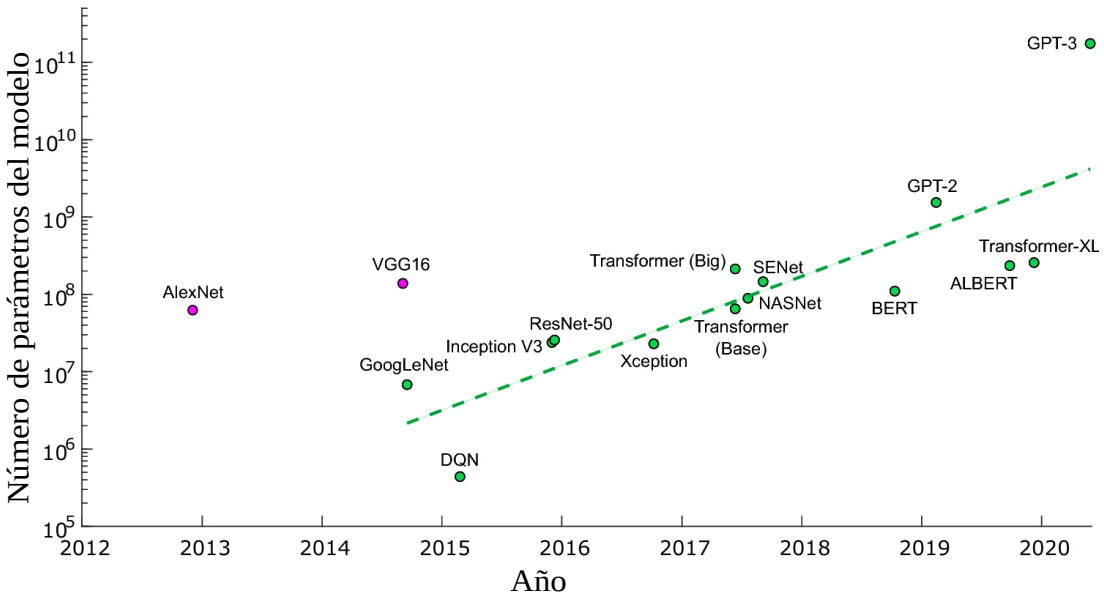

Figura 1.1: Evolución del número de parámetros de las redes neuronales profundas. Figura tomada de Bernstein, L., Sludds, A., Hamerly, R. et al. "Freely scalable and reconfigurable optical hardware for deep learning". Sci Rep 11, 3144 (2021). `https://doi.org/10.1038/s41598-021-82543-3`. Publicado bajo licencia `http://creativecommons.org/licenses/by/4.0/`.

Los modelos más avanzados de RNP requieren un costoso proceso de entrenamiento, que involucra conjuntos de datos enormes, y cuya realización puede requerir del orden de días e incluso semanas. Estos elevados costes computacionales han promovido el diseño de potentes computadores, en algunos casos equipados con aceleradores especializados [34, 62]. Al mismo tiempo, han aparecido un buen número de entornos para aprendizaje profundo amigables (por ejemplo, TensorFlow, PyTorch, Keras, Microsoft Cognitive Toolkit, scikit-learn, etc.), con el propósito de facilitar las tareas tanto en el ámbito científico-investigador como el industrial [68], aumentando la convergencia entre los campos de IA, HPC y las tecnologías *Big Data*.

El aprendizaje automático se clasifica como supervisado o no supervisado según si este cuenta o no con información que defina qué resultados son satisfactorios para dicho aprendizaje. En esta tesis se aborda el aprendizaje profundo supervisado, consistente en dos etapas: una primera etapa de *entrenamiento*, seguida por otra de *inferencia*. Durante la etapa inicial de entrenamiento, el modelo de RNP se ajusta progresivamente para dar una respuesta correcta a las entradas proporcionadas, a través de un proceso computacional basado en el método de gradiente estocástico descendente (*Stochastic Gradient Descent* o SGD) o alguna variante de este. Este proceso tiene como objetivo minimizar las diferencias entre la salida producida por el modelo de RNP y la respuesta esperada [68]. El entrenamiento de un modelo RNP es bastante costoso desde el punto de vista computacional y, por consiguiente, también desde las perspectivas del tiempo y el consumo de energía. En consecuencia, el entrenamiento suele efectuarse en una infraestructura HPC, habitualmente un clúster de computadores equipado con algún tipo de acelerador hardware [46, 75]. Una vez este proceso se ha completado, el modelo se despliega para operar sobre nuevos datos, no "vistos" previamente. Comparada con el entrenamiento, la etapa de inferencia es mucho más liviana desde el punto de vista computacional, pero en muchas ocasiones presenta fuertes restricciones en el tiempo de respuesta. Además, es usual que se ejecute en un dispositivo de bajo consumo y bajo rendimiento.

1.2. Objetivos

En el contexto que se expone en la sección anterior, se plantea como finalidad principal de la tesis doctoral el diseño, implementación y validación experimental de soluciones software paralelas para el aprendizaje automático mediante RNP. Esta meta debe plasmarse sobre plataformas aceleradoras, tomando en consideración criterios no solo de rendimiento sino también de coste y fiabilidad. Estos tres últimos aspectos, rendimiento, coste y fiabilidad, se evaluarán mediante el desarrollo de un entorno simple de entrenamiento e inferencia, que servirá como base para el análisis de las soluciones software propuestas.

Con la finalidad de alcanzar dicho propósito principal, se plantean los siguientes objetivos generales:

- Desarrollo de un entorno de simulación para RNP válido para la implementación de estrategias que mejoren el rendimiento del proceso de entrenamiento, como es el uso de esquemas de paralelismo, y que sea apto para su uso en plataformas distribuidas.

- Estudio de métodos para estimar los tiempos de ejecución de los diferentes procesos que componen el entrenamiento de las RNP. Búsqueda de modelos analíticos del coste de los núcleos y de las comunicaciones e investigación de nuevas técnicas alternativas.

- Análisis de la escalabilidad del proceso de entrenamiento distribuido, estudiando cómo el sistema responde a un incremento progresivo en la carga de trabajo, así como examinando la variación de su rendimiento y eficiencia a medida que se amplía la escala de la plataforma de entrenamiento.

Así pues, el trabajo realizado que aquí se presenta toma estos tres objetivos como punto inicial sobre el que se traza el camino para alcanzar el propósito principal planteado: soluciones innovadoras y contribuciones significativas en el ámbito de las RNP distribuidas. Esta investigación aborda dichos objetivos, adaptándose y planteando nuevos intereses en función de los resultados obtenidos durante su elaboración.

1.3. Estructura de la Tesis

El presente trabajo se estructura en un total de seis capítulos. El presente capítulo, Capítulo 1, introduce los desafíos actuales relativos al aprendizaje profundo que motivan la investigación desarrollada en esta tesis y describe los objetivos principales que se plantea desarrollar en esta.

A lo largo del Capítulo 2 se presenta el estado del arte y se definen las nociones básicas iniciales sobre las que se fundamenta la investigación realizada: las redes neuronales, su proceso de entrenamiento mediante los pasos *forward* y *backward*, así como los diferentes métodos de aceleración de este proceso.

Los siguientes tres capítulos exponen los trabajos realizados a lo largo del periodo de investigación y los resultados que de ellos se han derivado. El primero de ellos, Capítulo 3, presenta la herramienta PyDTNN diseñada para ofrecer un entorno de entrenamiento distribuido de las RNP versátil. En él se muestra cómo esta característica de PyDTNN hace posible su uso para la implementación de prototipos, facilitando la comprensión de nuevas técnicas y estrategias sobre los procesos de entrenamiento e inferencia.

En el Capítulo 4 se estudian diferentes métodos que permitan modelizar los tiempos de ejecución del entrenamiento. En él se definen diversos modelos que posibilitan la aproximación de este coste temporal en función de la red, los datos y la plataforma sobre la que se realiza el proceso de entrenamiento y se muestra cómo estas estimaciones hacen posible realizar análisis previos sobre la escalabilidad que anticipen las diferentes limitaciones y cuellos de botella y permitan evitarlos.

Como último campo de estudio, el Capítulo 5 analiza la primitiva de comunicación *AllReduce*, utilizada durante el entrenamiento distribuido. La investigación realizada engloba estudios experimentales sobre el tiempo de ejecución de esta comunicación y los algoritmos que la implementan. Se incluye como parte del estudio la indagación sobre las diferentes causas por las que difiere este coste del estimado, propuestas para mejorar dichas estimaciones, análisis sobre los algoritmos que son elegidos por defecto por las bibliotecas de comunicación y el alcance de mejora que derivaría de una elección más cuidadosa.

Para finalizar, en el Capítulo 6 se presentan las conclusiones y aportaciones principales de la investigación, se enumeran las contribuciones científicas que han derivado del trabajo realizado y se expone la actual línea de trabajo enfocada en la compresión de las matrices de pesos de las redes neuronales. Asimismo, se describen brevemente los diferentes campos abiertos que se plantean en esta línea como trabajo futuro.

Chapter 1

Introduction

1.1. Motivation

The enormous amount of data of all kinds that is being collected continuously has recently given rise to a fourth paradigm of scientific research, based on the analysis of large volumes of data (Big Data), as opposed to the three traditional approaches based respectively on theoretical formulation, practical experimentation and computer simulation. This new paradigm requires efficient use of data manipulation, analysis and visualization techniques, often using massively parallel computing and High Performance Computing (HPC) techniques [35, 63]. The question in this scenario is how to transform this flood of data into useful information and knowledge [23, 28].

Artificial Intelligence (AI) provides an effective means of deriving new knowledge from data. It is therefore natural that AI is currently being applied intensively to process the immense volumes of data from social networks, as well as the less numerous but equally relevant data from scientific and industrial applications [10, 67]. In particular, Machine Learning (ML) using Deep Neural Networks (DNNs), also known as Deep Learning (DL), has produced spectacular advances in applications of a very different nature.

Thus, a good number of success stories for deep learning are still present in traditional application niches for these technologies, such as image classification, language recognition and machine translation [44]. Moreover, the scope of use of these technologies has been extended in recent years to a much wider

range of problems, for example, in the study of new materials, simulations in astrophysics, fraud detection in banking, failure prevention in industrial environments, recommendation systems and time series prediction, among many others [3, 11, 19, 54, 67, 68, 78].

To a large extent, the renewed interest in deep learning can be traced back to the work of Yann LeCun, Yoshua Bengio and Geoffrey Hinton in the last decade [49]. In the following years, interest from the scientific community, as well as from industry, has opened the door to the design of increasingly sophisticated deep learning algorithms, along with increasingly complex DNN-based models, as shown in the evolution reflected in Figure 1.1.

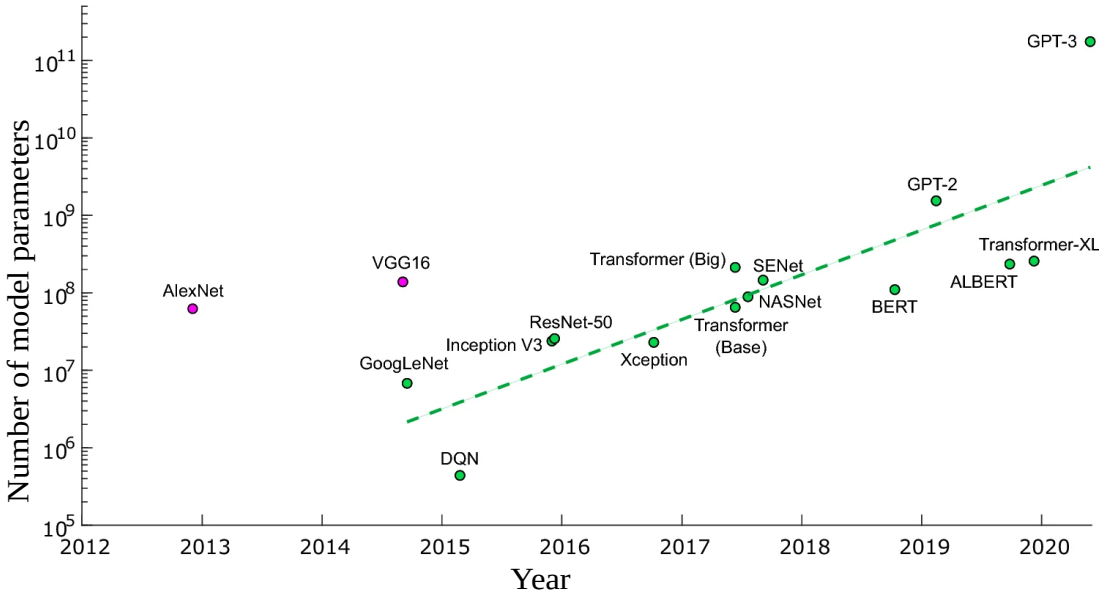

Figure 1.1: Evolution of the number of parameters of deep neural networks. Figure taken from Bernstein, L., Sludds, A., Hamerly, R. et al. "Freely scalable and reconfigurable optical hardware for deep learning". Sci Rep 11, 3144 (2021). `https://doi.org/10.1038/s41598-021-82543-3`. Published under license `http://creativecommons.org/licenses/by/4.0/`.

The most advanced DNN models require a costly training process, involving huge datasets, which can take days or even weeks to complete. These high

computational costs have led to the design of powerful computers, in some cases equipped with specialized accelerators [34, 62]. At the same time, a number of user-friendly deep learning environments have emerged (e.g. TensorFlow, PyTorch, Keras, Microsoft Cognitive Toolkit, scikit-learn, etc.), with the aim of facilitating tasks in both the scientific-research and industrial domains [68], increasing convergence between the fields of AI, HPC and Big Data technologies.

Machine learning is classified as supervised or unsupervised depending on whether or not it has information that defines which results are satisfactory for such learning. This thesis deals with supervised deep learning, which consists of two stages: a first *training* stage, followed by an *inference* stage. During the initial training stage, the DNN model is progressively adjusted to give a correct response to the provided inputs, through a computational process based on the Stochastic Gradient Descent method (SGD) or some variant thereof. This process aims to minimize the differences between the output produced by the DNN model and the expected response [68]. Training a DNN model is quite expensive from a computational point of view, and therefore also from a time and energy consumption point of view. As a result, training is usually performed on an HPC infrastructure, usually a cluster of computers equipped with some kind of hardware accelerator [46, 75]. Once this process is complete, the model is deployed to operate on new, previously unseen data. Compared to training, the inference stage is much lighter from a computational point of view, but it often has strong response time constraints. In addition, it is usually run on a low-power, low-performance device.

1.2. Objectives

In the context described in the previous section, the main objective of the thesis is to design, implement and experimentally validate parallel software solutions for machine learning using DNNs. This goal has to be achieved on accelerator platforms, taking into account not only performance criteria, but also cost and reliability. These last three aspects, performance, cost and reliability, will be evaluated through the development of a simple training and inference environment, which will serve as a basis for the analysis of the proposed software solutions.

In order to achieve this main objective, the following general objectives are proposed:

- Develop a simulation environment for DNNs that is valid for the implementation of strategies that improve the performance of the training process, such as the use of parallelism schemes, and that is suitable for use on distributed platforms.
- Study methods for estimating the execution times of the different processes that make up DNNs training. Search for analytical models of core and communication costs and investigate new alternative techniques.
- Analyzing the scalability of the distributed training process, investigating how the system responds to a progressive increase in workload, and examining the variation in its performance and efficiency as the training platform scales.

Thus, the work presented here takes these three objectives as a starting point from which to trace the path to achieve the main objective: innovative solutions and significant contributions in the field of distributed DNNs. This research will address these objectives, adapting and raising new interests according to the results obtained during its elaboration.

1.3. Structure of the thesis

This thesis is divided into six parts. The present chapter, Chapter 1, introduces the current challenges related to deep learning that motivate the research developed in this thesis and describes the main objectives to be developed in this thesis.

Throughout the chapter, the state of the art is presented and the first basic concepts on which the research is based are defined: neural networks, their training process through the steps *forward* and *backward*, as well as the different methods to accelerate this process.

The following three chapters present the work carried out during the research period and the results derived from it. The first of these, Chapter 3, introduces the

PyDTNN tool, which is designed to provide a versatile distributed DNN training environment. It is shown how this feature of PyDTNN allows it to be used for implementing prototypes, facilitating the understanding of new techniques and strategies for training and inference processes.

Chapter 4 examines different methods for modeling training execution times. It defines different models that make it possible to approximate this time cost as a function of the network, the data and the platform on which the training process is carried out, and shows how these estimates make it possible to carry out prior analyzes of scalability, anticipating the various limitations and bottlenecks and allowing them to be avoided.

As a last field of study, Chapter 5 analyzes the communication primitive *AllReduce*, used during distributed training. The research conducted encompasses experimental studies on the execution time of this communication and the algorithms that implement it. Part of the study includes an investigation into the different reasons why this cost differs from the estimated cost, proposals for improving these estimates, analysis of the algorithms that are chosen by default by the communication libraries and the scope for improvement that would result from a more careful choice.

Finally, Chapter 6 presents the main conclusions and contributions of the research, lists the scientific contributions derived from the work carried out and presents the current line of work focused on the compression of the weight matrices of neural networks. It also briefly describes the different open fields that are proposed as future work in this line of research.

Capítulo 2

Revisión de las Redes Neuronales

A lo largo de este capítulo se introducen los conceptos básicos sobre redes neuronales que se utilizan posteriormente en la memoria, y se proporciona una breve visión del estado del arte de las partes relacionadas con la tesis. El campo de estudio de las redes neuronales es muy amplio. El contenido de este capítulo solo trata de contextualizar al lector en este campo, ofreciendo una visión global y detallando únicamente aquellas partes que resultan clave para comprender mejor el trabajo desarrollado en el marco de la tesis doctoral.

2.1. Conceptos Básicos

Esta primera sección introduce las redes neuronales: qué son, cómo se estructuran y en qué consiste su funcionamiento. Para ello, se explican estas cuestiones con el objetivo de dar una visión general sobre las redes neuronales. Las materias y elementos más específicos que requieran un análisis más detallado se tratarán posteriormente con más detalle.

Las redes neuronales son algoritmos de aprendizaje autónomos que efectúan un reconocimiento de patrones sobre un conjunto de muestras de entrada, con la finalidad de extraer características y realizar una determinada tarea –por ejemplo, clasificar imágenes, reconocer sonidos, o transcribir voz a texto–, tratando de predecir una respuesta correcta sobre muestras de entrada desconocida con anterioridad.

Las redes neuronales deben su nombre a las "neuronas" artificiales que simulan el proceso de aprendizaje de las propias neuronas biológicas. Estas se conectan entre sí mediante una serie de extensiones de entrada, llamadas dendritas, y una extensión de salida, llamada axón, tal y como se muestra en la Figura 2.1. Durante la transmisión de información, una neurona acepta las señales que recibe a través de las dendritas, realiza un cálculo con las mismas y genera una señal en el axón. Estas señales de entrada y salida se denominan activaciones. El axón de una neurona se ramifica y está conectado a las dendritas de muchas otras neuronas. La conexión entre una rama del axón y una dendrita se denomina sinapsis y una característica clave de esta es que puede escalar la señal que la cruza, lo que permite que el cerebro humano aprenda a medida que ajusta correctamente ese factor de escala. Esta característica del cerebro sirve de inspiración para los algoritmos de aprendizaje automático basados en redes neuronales. En particular, una red neuronal también consiste de neuronas conectadas entre sí, recibiendo y enviando información, de forma que las modificaciones que las neuronas hacen con esta información posibilitan el proceso de auto-aprendizaje [69].

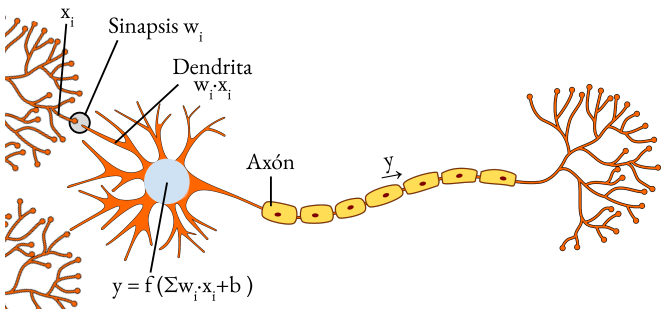

Figura 2.1: Conexiones de una neurona.

La forma en la que las neuronas se conectan entre sí define la topología de la red neuronal. Normalmente estas suelen organizarse por capas ordenadas, de manera que las neuronas de una capa reciben información de las neuronas de la capa anterior y envían información a las de la siguiente. La red que se muestra como ejemplo en la Figura 2.2 está compuesta por 5 capas, con 4, 3, 2, 3 y 2 neuronas respectivamente. En adelante, al número de neuronas de cierta capa l lo denotaremos como n_l y a la información que contienen las neuronas de la

capa l las denominaremos, igual que en las neuronas biológicas, activaciones, y las denotaremos como $a^{(l)} = \{a_j^{(l)}\}_{j=1}^{n^{(l)}}$.

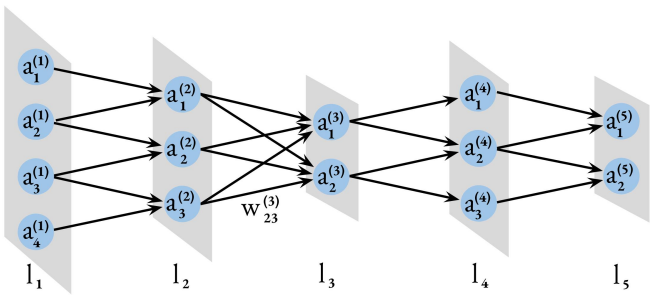

Figura 2.2: Red Neuronal de ejemplo.

En esta tesis solo se abordarán principalmente redes con este tipo de estructura puesto que son las más utilizadas. Existen redes neuronales, sin embargo, que se estructuran de forma diferente: en las redes residuales, por ejemplo, las neuronas toman como entrada información de varias capas previas, rompiendo el patrón secuencial de interconexión.

Otro aspecto a considerar, y que caracteriza la topología de la red, son las conexiones que existen entre neuronas; esto es, cuántas neuronas aportan información a otra. Las capas de una red se clasifican según qué neuronas aportan información (todas, solo las vecinas, etc.) y qué se obtiene de dicha información. En esta tesis se distinguen los siguientes tipos de capas:

- **Completamente conectadas** (*Fully-Connected* o FC): En una capa de este tipo cada neurona recibe información de todas las neuronas de la capa anterior, como ocurre en la capa l_3 de la Figura 2.2, donde cada neurona $a_i^{(3)}$ recibe información de todas las neuronas de la capa l_2, esto es, $a_i^{(3)} = f(a_1^{(2)}, a_2^{(2)}, a_3^{(2)})$. La contribución de una activación $a_i^{(l-1)}$ al cálculo de una activación $a_j^{(l)}$ se denomina *peso*, y se denota como $W_{ij}^{(l)}$. Este peso hace referencia al factor de escala que aplican las neuronas biológicas durante la sinapsis. En la Figura 2.2, por ejemplo, el peso que tiene la activación $a_3^{(2)}$ al calcular $a_2^{(3)}$ es $W_{23}^{(3)}$.

Al combinar información de todas las entradas en el resultado, las características que cada neurona de la capa anterior haya podido detectar también se combinan. De esta forma se obtienen resultados que engloban todas las propiedades conocidas previamente.

- **Convolucionales**: En este tipo de capas las neuronas de una capa solo se conectan con un subconjunto de neuronas vecinas de la capa anterior. Estas capas son capaces de detectar características espaciales tales como patrones. Además pueden tener diferentes dimensiones, si bien en este trabajo solo se tratarán las 2D. Estas capas están compuestas por una o varias matrices bidimensionales de activaciones denominadas *canales*. La notación $k^{(l)}$ indicará el número de canales que forman una capa l.
  Las capas convolucionales son estructuralmente más complejas que las FC, y utilizan tres parámetros adicionales:

  - **Filtros**. Tienen tres dimensiones: alto, ancho y número de canales*, que definen cuántas neuronas vecinas se toman a la vez para obtener cierto resultado y el peso que se da a cada una de ellas durante el cálculo. Por cada filtro que se aplica se obtiene un resultado diferente que forma un canal de activaciones. Por lo tanto, el número de filtros utilizados en cierta capa define el número de canales que forman las activaciones de dicha capa.
  - **Relleno (*padding*)**. Es un parámetro de relleno que se añade a los datos de entrada; un valor de *padding* p viene a ser un marco de p neuronas de ancho, que rodean las neuronas de la capa previa, y que suelen inicializarse a cero.
  - **Paso (*stride*)**. Define la longitud del salto en el uso del filtro, es decir, cada cuántas neuronas se vuelve a aplicar la operación de convolución. El *stride* y el *padding* pueden tomar dos valores diferentes según si se

*El alto y el ancho de los filtros son parámetros configurables; en cambio, el número de canales de los filtros realmente no debe considerarse como parámetro en sí, pues debe coincidir con el número de canales de salida de la capa previa y, por lo tanto, su valor viene fijado por la red.

aplican a lo alto o a lo ancho de las activaciones. En la práctica, en esta tesis se aplica siempre el mismo valor en ambas dimensiones.

La capa l_2 de la Figura 2.2, por ejemplo, puede ilustrar una capa convolucional sin *padding*, con *stride* 1, y un filtro $2 \times 1 \times 1$ (dos filas, una columna y un único canal) de forma que de cada dos activaciones seguidas de la capa l_1, se obtiene un solo resultado.

- **Pooling:** Este tipo de capas también utilizan los parámetros *padding* y *stride*. Sin embargo, a diferencia del caso de las capas convolucionales, las capas *pooling* no tienen *pesos* sino que simplemente aplican una sencilla función, como el máximo o la media sobre la ventana fijada. Una capa *pooling* sirve para agrupar la información creando versiones resumidas de la información a su entrada.

- **Dropout:** Son capas que aplican filtros aleatoriamente para disminuir la cantidad de datos con la que se trabaja. Estas capas se utilizan para evitar un sobreajuste (*overfitting*) de la red y mejorar así el proceso de aprendizaje. Su aplicación reduce el número de conexiones entre neuronas de la capa anterior.

- **Flatten:** Es un tipo de capa muy sencillo cuya única función es aplanar las dimensiones de los datos que recibe. En muchas ocasiones, como suele ocurrir en las capas convolucionales, nos encontramos con capas de hasta 3 dimensiones. Estas capas suelen utilizarse principalmente como último paso de una red neuronal, para obtener el vector de una dimensión de salidas.

Las redes neuronales tradicionales se conocen como *perceptrones multicapa* (*Multilayer Perceptrons* o MLP) y están formadas únicamente por capas FC. En la práctica, sin embargo, las capas FC suelen combinarse con capas convolucionales para mejorar la eficiencia de las redes. Cuando una red contiene capas convolucionales, se dice que es una red neuronal convolucional (*Convolutional Neural Networks* o CNNs). Las redes neuronales que se estudiarán en este trabajo son las CNNs, centrándonos más profundamente en los tipos de capa FC

y convolucional, ya que estas concentran una parte mayoritaria de los cómputos aritméticos en los procesos tanto de entrenamiento como de inferencia.

Como se ha mencionado previamente, la información de salida de una neurona o activación, se calcula como una función de las activaciones de las neuronas de la capa anterior ponderadas, es decir,

$$a_i^{(l)} = \sum_{j=1}^{n^{(l-1)}} W_{ij}^{(l)} a_j^{(l-1)} + b_i^{(l)}, \tag{2.1}$$

Si se tiene en cuenta el conjunto total de activaciones de cada capa, $a^{(l)}$, se utiliza la expresión matricial:

$$a^{(l)} = W^{(l)} a^{(l-1)} + b^{(l)}. \tag{2.2}$$

La forma en que las activaciones $a^{(l-1)}$ contribuyen al valor de la nueva activación viene determinada por los *pesos*. Para una capa l estos pesos suelen agruparse en una matriz $W^{(l)} \in \mathbb{R}^{n_l \times n^{(l-1)}}$, donde $W_{ij}^{(l)}$ es el peso con el que la activación $a_j^{(l-1)}$ contribuye a la activación $a_i^{(l)}$ de la neurona i. De esta forma cada activación de la capa $l-1$ tiene una cierta relevancia a la hora de calcular una activación de la capa l. Además de los pesos, para cada capa l las expresiones anteriores definen también los *sesgos*. Estos están expresados como un vector $b^{(l)} \in \mathbb{R}^{n^{(l)}}$, donde cada valor $b_i^{(l)}$ especifica una constante para la función que contribuye, junto con los pesos, a determinar el valor de la neurona i de dicha capa.

La elección de los valores para los pesos y sesgos es la cuestión que nos queda por resolver. En una red neuronal, los valores de las muestras de entrada se propagan a través de la red, capa tras capa, experimentando las correspondientes transformaciones hasta obtener unos resultados. Dado que estos resultados se obtienen tras aplicar en cada capa los pesos y los sesgos a las activaciones, el hecho de que los resultados sean buenos depende de que dichos pesos y sesgos se elijan correctamente.

Para el propósito de afinado, inicialmente se realiza una elección pseudo-aleatoria que posteriormente se va ajustando mediante un procedimiento de *prueba-error-mejora* iterativo. En otras palabras, se prueba la red obteniendo unos resultados, se calcula el error de dichos resultados respecto a lo esperado,

se ajustan los pesos y sesgos elegidos en función de dicho error. Este procedimiento de tres etapas se repite en bucle hasta obtener un error aceptable o hasta completar un número de épocas* preestablecidas.

Los pasos del entrenamiento de una red neuronal se dividen en dos, *forward* y *backward*, que se describen en detalle a continuación. Una primera definición orientativa de estos pasos, sin embargo, podría basarse en la aproximación *prueba-error-mejora* que se ha mencionado antes: el paso *forward* correspondería a la componente de *prueba* mediante la cual obtenemos el *error*; y el paso *backward* sería la componente de *mejora*, que ajusta los pesos y sesgos.

Se definen, además, dos procesos básicos para una red neuronal: el entrenamiento o *training*, que se basa en el aprendizaje autónomo de la red y engloba, por tanto, *forward* y *backward*; y la inferencia, que consiste en utilizar la red una vez esta ya ha sido entrenada, aplicando el conocimiento adquirido durante el entrenamiento, para generar nuevos resultados, *forward*, sin modificar la red ni sus parámetros.

Cuanto mejor sea el entrenamiento de la red, mejor precisión se obtendrá en la inferencia y, por lo tanto, tendremos más posibilidades de conseguir que la red genere resultados correctos. El proceso de entrenamiento es, por consiguiente, la pieza clave para el funcionamiento de una red neuronal y será el objeto de estudio de este trabajo.

2.2. Paso *forward*

El paso hacia delante o paso *forward* procesa las muestras introducidas en la capa de entrada l_1 de la red obteniendo los resultados correspondientes. En el paso *forward*, cada capa de la red recibe unos datos, los procesa y los envía a la siguiente capa de forma que, para esta última, dichos datos procesados serán sus datos de entrada; excepto para la última capa l_L, en la que el resultado obtenido al procesar los datos serán la salida, es decir, los resultados de la red neuronal.

Cuando una capa l procesa los datos recibidos, sus neuronas aplican en primer lugar los pesos $W^{(l)}$ y sesgos $b^{(l)}$, obteniendo de esta forma una salida ponderada

*Una época (*epoch*) es una etapa que comprende el procesamiento, una única vez, del conjunto completo de muestras de entrenamiento.

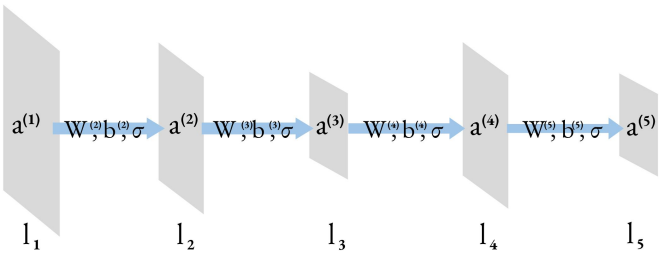

Figura 2.3: Red de la Figura 2.2 con las transiciones del paso *forward*.

de las entradas:

$$z^{(l)} = W^{(l)} \cdot a^{(l-1)} + b^{(l)}.$$

Una vez ponderadas las entradas, se aplica una función de activación no lineal σ, para forzar un comportamiento no lineal de la red:

$$a^{(l)} = \sigma(z^{(l)}).$$

En este trabajo no se profundiza en la elección de la función σ de activación. Sin embargo, es importante mencionar que la función σ debe ser derivable para poder aplicarse, como veremos más adelante, de manera combinada con el método del descenso de gradiente estocástico. Aunque existen muchas funciones no lineales que cumplen esta propiedad, las más comúnmente utilizadas son *sigmoide*, *tanh*, *ReLU* y algunas variaciones de esta última.

Veamos un ejemplo de lo que hemos tratado hasta ahora. Si nos fijamos en la capa l_3 de la Figura 2.2, que corresponde a una capa FC, las activaciones se calcularían de la siguiente manera:

$$\begin{aligned} a^{(3)} &= \sigma\left(\begin{pmatrix} W_{11}^{(3)} & W_{12}^{(3)} & W_{13}^{(3)} \\ W_{21}^{(3)} & W_{22}^{(3)} & W_{23}^{(3)} \end{pmatrix} \cdot \begin{pmatrix} a_1^{(2)} \\ a_2^{(2)} \\ a_3^{(2)} \end{pmatrix} + \begin{pmatrix} b_1^{(3)} \\ b_2^{(3)} \end{pmatrix}\right) \\ &= \sigma\left(\begin{pmatrix} z_1^{(3)} \\ z_2^{(3)} \end{pmatrix}\right) = \begin{pmatrix} a_1^{(3)} \\ a_2^{(3)} \end{pmatrix}. \end{aligned} \tag{2.3}$$

Así, las activaciones $a^{(l)}$ serán los datos de entrada de la siguiente capa $l+1$, con los que calculará sus activaciones $a^{(l+1)}$. Nótese que, debido a esta definición, las activaciones de las diferentes capas se deben calcular secuencialmente.

Durante el paso *forward*, las capas deben calcular además, la matriz diagonal $D^{(l)} \in \mathbb{R}^{n^{(l)} \times n^{(l)}}$ evaluando la derivada de la función de activación, σ', en los anteriores valores calculados $z^{(l)}$:

$$D^{(l)} = \text{diag}(\sigma'(z^{(l)})). \tag{2.4}$$

Esta matriz se calcula durante el paso *forward* para poder ser utilizada en el paso *backward*. En el ejemplo expuesto anteriormente, esta matriz sería:

$$D^{(3)} = \begin{pmatrix} \sigma'(z_1^{(3)}) & 0 \\ 0 & \sigma'(z_2^{(3)}) \end{pmatrix}. \tag{2.5}$$

Los elementos que intervienen en las operaciones anteriores, así como las dependencias, son las mismas para cualquier tipo de capa. En cambio, las operaciones se aplican de diferentes maneras según el tipo de capa del que se trate. Así, en las capas FC se aplican como un producto matriz-vector, de modo que cada neurona aporta información a todas las neuronas de la capa siguiente con ponderaciones diferentes. En cambio, las capas convolucionales y *pooling* realizan la operación de forma diferente para reutilizar los pesos y la función (máximo, media, etc.) respectivamente a lo largo de todas las activaciones de la capa anterior. Este detalle se ilustra gráficamente en la Figura 2.4, donde se muestra la aplicación del paso *forward* y los tamaños de los elementos involucrados en las capas convolucionales, *pooling* y FC, que son las que involucran operaciones más complejas.

En mayor grado de detalle, podemos destacar las siguientes diferencias en función del tipo de capa:

- La primera diferencia de las capas convolucionales y *pooling* respecto a las capas FC son las dimensiones de las activaciones de entrada $a^{(l-1)}$. Mientras que en las capas FC son un vector con los $n^{(l-1)}$ elementos, en las capas convolucionales y *pooling* las entradas son tridimensionales.

- Otra diferencia es el parámetro *padding* p que aparece en las capas convolucionales y *pooling*, el cual permite ampliar las activaciones de entrada añadiéndoles un marco de p elementos de relleno, antes de calcular las activaciones de salida. Esto se puede realizar para ajustar los tamaños de salida

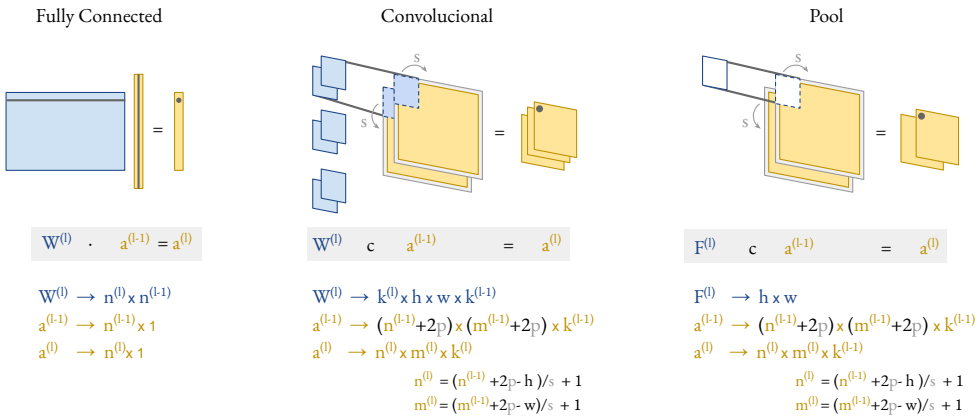

Figura 2.4: Paso *forward*.

a los deseados, o para no restarle importancia a la hora de aplicar los pesos a las activaciones ubicadas, por ejemplo, en los bordes de una imagen.

- Las dimensiones de los pesos utilizados también son diferentes según el tipo de capa. Las capas FC tienen una matriz de pesos con tantas columnas como activaciones tiene la capa previa, $n^{(l-1)}$, y tantas filas como activaciones tiene la siguiente capa, $n^{(l)}$. Los pesos de las capas convolucionales, sin embargo, se dividen en filtros tridimensionales, todos ellos del mismo tamaño y cuyo número de canales debe coincidir con el número de canales de las activaciones de entrada. El tamaño de los sesgos también varía en las capas convolucionales. Mientras que en una capa FC hay tantos sesgos como activaciones, i.e., $b \in \mathbb{R}^{n^{(l)}}$; una capa convolucional tiene sólo un sesgo por filtro, i.e, $b \in \mathbb{R}^{k^{(l)}}$.

- Las capas *pooling* no tienen pesos ni sesgos, sino una ventana bidimensional que aplica una función en concreto a los datos de entrada, como la media o el elemento máximo. La diferencia respecto a los filtros de una capa convolucional es que esta puede aplicar pesos diferentes para cada canal de las activaciones de la capa anterior, mientras que las capas *pooling* siempre aplican la misma función en todos los canales de la capa anterior.

- Entre las capas convolucionales y *pooling* también existe una diferencia en las dimensiones de las activaciones de salida $a^{(l)}$. En las capas *pooling* el

número de canales de las activaciones se mantiene ($k^{(l-1)} = k^{(l)}$), mientras que en las capas convolucionales el número de canales de las activaciones $a^{(l)}$ viene determinado por el número de filtros empleados en la convolución.

- Por último, una gran diferencia con respecto a cómo evaluar el paso *forward* según el tipo de capa, reside en la forma de aplicar los pesos. Las capas FC, como hemos dicho, aplican una multiplicación matriz-vector. Las capas convolucionales aplican los pesos tal como se muestra en la Figura 2.4, y la operación que realizan es una convolución, es decir, la suma de todos los elementos de la multiplicación punto a punto. En el ejemplo de la Figura 2.4, suponiendo que las activaciones tienen incluido el *padding*, la primera activación, $a^{(l)}_{(1,1,1)}$, se calcula como

$$a^{(l)}_{(1,1,1)} = \sigma\left(z^{(l)}_{(1,1,1)}\right), \qquad z^{(l)}_{(1,1,1)} = \sum_{k=1}^{k^{(l-1)}} \sum_{i=1}^{h} \sum_{j=1}^{w} W^{(l)}_{(1,i,j,k)} \cdot a^{(l-1)}_{(i,j,k)} + b^{(l)}_1. \quad (2.6)$$

  Para calcular el siguiente elemento $a^{(l)}_{(1,1,2)}$ se desplazaría el filtro hacia la derecha sobre las activaciones s unidades, siendo s el paso, y se volvería a aplicar la misma operación a las nuevas activaciones. Para calcular las activaciones del siguiente canal se utilizarán los pesos $W^{(l)}_{(2,i,j,k)}$. De forma general, para calcular un elemento de la salida, suponiendo que las activaciones ya tienen incluido el *padding*, se utiliza la siguiente operación:

$$a^{(l)}_{(x,y,z)} = \sigma\left(z^{(l)}_{(x,y,z)}\right), \qquad z^{(l)}_{(x,y,z)} = \sum_{k=1}^{k^{(l-1)}} \sum_{i=y\cdot s}^{y\cdot s+h} \sum_{j=z\cdot s}^{z\cdot s+w} W^{(l)}_{(x,i,j,k)} \cdot a^{(l-1)}_{(i,j,k)} + b^{(l)}_x, \quad (2.7)$$

  siendo k, h y w el número de canales, alto y ancho de cada filtro respectivamente. Las capas de *pooling* no tienen pesos en sí, por lo tanto solo aplican la función a las activaciones recorriéndolas de igual modo que las capas convolucionales.

Con el objetivo de aplicar los pesos de las capas convolucionales como un producto de matrices, a menudo se utiliza la transformación *im2col* [21, 72], que reorganiza las activaciones de entrada en un conjunto de columnas. La Figura 2.5 muestra un ejemplo sencillo de cómo aplicar esta transformación. Así mismo,

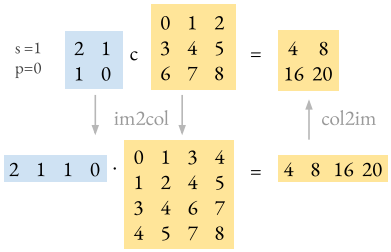

Figura 2.5: Ejemplo de convolución utilizando *im2col*.

existe la variante *im2row*, cuya transformación resulta análoga a la de *im2col* salvo por que esta reorganiza las activaciones en filas en vez de columnas. La elección de una u otra variante depende de la aplicación que se les quiera dar. Así, en el caso que nos concierne en este trabajo, por ejemplo, la conversión de las activaciones para aplicar una convolución se realizará con *im2col* o *im2row* en función de si el orden de los operandos en el producto que se desea realizar es $W \cdot A$ o $A \cdot W$ respectivamente.

Generalizando esta aplicación, la operación de convolución $W^{(l)}A^{(l-1)}$ utilizando la transformada *im2col* se convierte en el producto de matrices $W'^{(l)}A'^{(l-1)}$, tal y como se muestra en la Figura 2.6. Los pesos se reorganizan en una matriz con tantas filas como filtros. Las activaciones, en cambio, se convierten en una matriz con tantas columnas como elementos tendrá cada canal de las activaciones de salida.

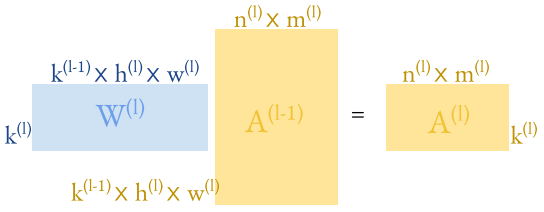

Figura 2.6: Operación convolucional del paso *forward* como un producto de matrices.

2.3. Paso *backward*

La clave del proceso de aprendizaje de una red neuronal reside en este segundo paso. Durante la etapa *forward*, la red produce unos resultados al aplicar a las muestras de entrada ciertos pesos y sesgos. Dichos resultados pueden acercarse en mayor o menor medida a los esperados. En el paso *backward* la red ajusta los pesos y sesgos según el error obtenido en el paso *forward*, de forma que se minimice el error en la siguiente iteración del entrenamiento.

El error cometido por la red neuronal se calcula mediante una función de coste con el propósito de minimizarlo durante el entrenamiento. Esta función puede tomar múltiples formas según el tipo de datos con los que se está trabajando y en función de cómo se desea tratar estos. Una de las funciones más conocidas para calcular esta pérdida es la entropía cruzada que se define como:

$$\text{Coste} = -\sum_{i=1}^{N} p_r(i) \log p_p(i), \tag{2.8}$$

donde $p_r(i)$ y $p_p(i)$ denotan respectivamente la probabilidad real y la probabilidad estimada por la red para la clase i. De esta forma, el objetivo del entrenamiento es obtener los pesos y sesgos apropiados que disminuyan el valor de la función de coste.

El método utilizado para ajustar los pesos y sesgos es el descenso de gradiente estocástico. Este método utiliza los valores de los gradientes con respecto a las salidas esperadas para los pesos y sesgos, para obtener la dirección óptima en la que deben modificarse los parámetros $W^{(l)}$ y $b^{(l)}$ para minimizar el error cometido.

Junto a esta dirección, se utiliza otro parámetro, llamado ratio de aprendizaje (*learning rate* o LR y representado como η), que representa cuánto queremos avanzar en esa dirección. Aunque en este trabajo no se va a profundizar en el valor de η, cabe destacar que dicho parámetro debe escogerse con cuidado. Por ejemplo, un valor de η bajo hace que el aprendizaje sea muy lento, ya que avanzamos hacia el punto óptimo muy lentamente. Por contra, si tomamos η demasiado grande podría avanzarse muy rápido en un solo paso, pero podríamos hacerlo en una dirección poco óptima.

Mientras que en el paso *forward* los cálculos se realizan desde la primera capa (datos de entrada) hacia la última (resultados), en el paso *backward* los cálculos

se llevan a cabo desde la última capa hasta la primera. Esto se debe a que el ajuste de los pesos se realiza en función del error cometido, o coste, y este solo se conoce en la última capa, cuando obtenemos los resultados.

Dado que en el paso *forward* las activaciones de las neuronas se ponderan por su peso, cuando se quieren ajustar dichos pesos, se debe tener en cuenta el factor de influencia de cada activación en el resultado y, por lo tanto, en el valor obtenido de la función *coste*. Para este propósito, se define para cada capa l el vector $\delta^{(l)} \in \mathbb{R}^{n^{(l)}}$ como la derivada de la función coste respecto de cada variable $z_j^{(l)}$. De este modo, el vector $\delta^{(l)}$ representa la sensibilidad de la función coste a las modificaciones sobre las entradas ponderadas; en otras palabras, cómo cambia el coste al ajustar los pesos y sesgos. Se denomina *gradientes* a estos vectores $\delta^{(l)}$ que definen cómo deben cambiar los pesos y sesgos, y se cumple que:

$$\begin{aligned} \delta^{(L)} &= \sigma'(z^{(l)}) \circ (a^{(L)} - y) = D^{(L)}(a^{(L)} - y), \\ \delta^{(l)} &= \sigma'(z^{(l)}) \circ (W^{(l+1)})^T \delta^{(l+1)} = D^{(l)}(W^{(l+1)})^T \delta^{(l+1)}, \qquad 2 \leq l < L, \end{aligned} \tag{2.9}$$

donde y define el valor esperado y $\circ$ denota el producto de Hadamard.

Del mismo modo que en el paso *forward*, las operaciones del paso *backward* se implementan de forma distinta según el tipo de capa. A continuación se muestra la implementación de las capas FC y convolucionales, que son las más interesantes en la parte del paso *backward*, pues son las únicas que utilizan pesos y, por lo tanto, las únicas que deben modificar sus parámetros para mejorar el aprendizaje. Esto, sin embargo, no significa que el resto de capas no realicen el paso *backward*. Existen dependencias que hacen necesario el cálculo de los gradientes asociados a todas las capas, pues para calcular el gradiente de una capa l, se utiliza el gradiente de la capa siguiente $l + 1$ y, en consecuencia, el de todas las capas posteriores.

Como se aprecia en la Figura 2.7, las capas FC son las que más se ajustan a la descripción matemática, realizando una multiplicación matriz-vector. Para poder calcular el gradiente, las capas convolucionales utilizan el mismo esquema en la operación, pero sirviéndose de una convolución en vez de una multiplicación. Además, durante el paso *backward*, las capas convolucionales a menudo requieren redimensionar alguno(s) de los operandos que intervienen en la convolución. En esta parte, por ejemplo, si el *stride* aplicado durante el *forward* es mayor que

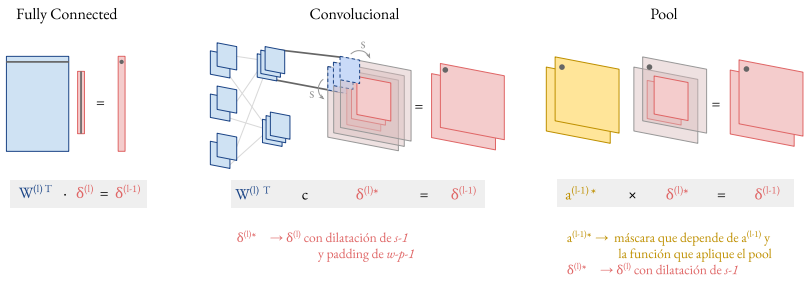

Figura 2.7: Cálculo del gradiente durante el paso *backward*.

uno, para el cálculo del gradiente $g^{(l-1)}$, el gradiente de la capa siguiente $g^{(l)}$ debe dilatarse introduciendo $s-1$ filas y $s-1$ columnas de ceros entre todas las activaciones. Después se le debe añadir, además, un marco de relleno de ceros (*padding*). Estos ajustes son necesarios para conseguir que las dimensiones del gradiente resultante de la convolución se correspondan con las dimensiones de la capa $l-1$, ya que, dicho gradiente es utilizado durante la actualización de pesos mediante una operación "punto a punto" (es decir, que se aplica individualmente a cada uno de los elementos del operador de entrada).

La primera parte del paso *backward* consiste en calcular los anteriores gradientes $\delta^{(l)}$ y en la segunda parte se utilizan estos gradientes, junto al ratio de aprendizaje mencionado antes, para actualizar tanto los pesos como los sesgos:

$$W^{(l)} = W^{(l)} - \eta\delta^{(l)}(a^{(l-1)})^T,$$

$$b^{(l)} = b^{(l)} - \eta\delta^{(l)}.$$

En la Figura 2.8 se diferencia cómo se aplica dicha operación en capas FC y convolucionales donde, de nuevo, las capas FC mantienen la estructura matemática multiplicando ambos vectores, mientras que las convolucionales realizan la operación de convolución.

En las capas convolucionales, las operaciones convolucionales pueden convertirse de nuevo en productos de matrices, tanto en el cálculo de gradientes como en la actualización de pesos, utilizando la transformación *im2col*. Estos productos se representarían como muestra la Figura 2.9, teniendo en cuenta que deben incluir

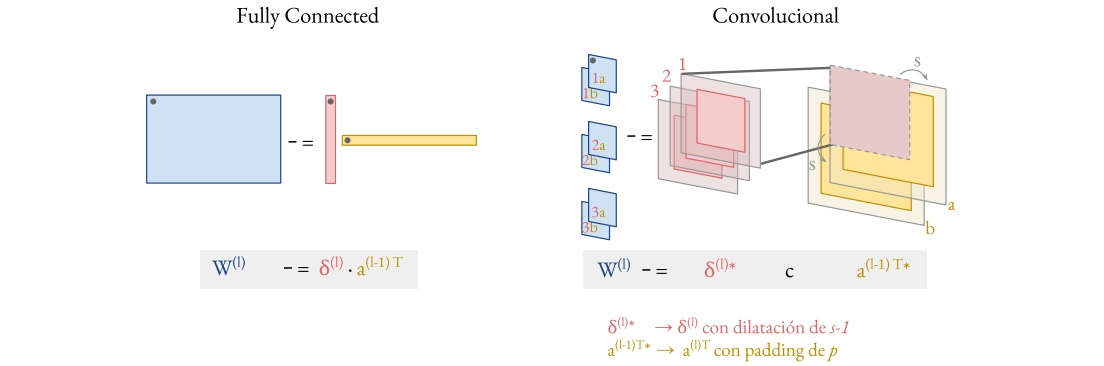

Figura 2.8: Actualización de pesos durante el paso *backward*.

las dilataciones y los *paddings* necesarios, como en las operaciones de convolución.

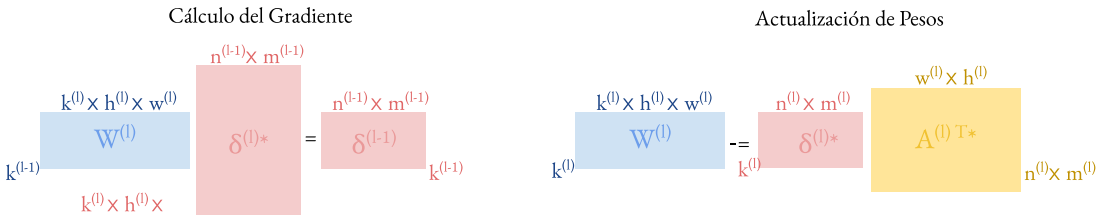

Figura 2.9: Operaciones convolucionales del paso *backward* como productos de matrices.

Entre estas dos partes del paso *backward*, cálculo de gradientes y actualización de pesos, existen dependencias. En primer lugar, para poder actualizar los pesos de cierta capa l necesitaremos haber calculado antes su gradiente $\delta^{(l)}$. Por lo tanto, la segunda parte del paso *backward* para cierta capa depende de la primera parte. También existe una dependencia de la primera parte respecto de la segunda: para poder calcular el gradiente de cierta capa l necesitamos los pesos sin actualizar de la siguiente capa $l+1$, por ello, el primer paso del *backward* en la capa l debe realizarse antes del segundo paso de la capa $l+1$.

Teniendo en cuenta las operaciones mencionadas de los pasos *forward* y *backward*, el entrenamiento de una red formada por L capas para una muestra x cuyo valor esperado es y se esquematiza en el Algoritmo 1.

Paso *forward*
$a^{(1)} = x$ **for** $l = 2$ *hasta* L **do**
  $z^{(l)} = W^{(l)} a^{(l-1)} + b^{(l)}$
  $a^{(l)} = \sigma(z^{(l)})$
  $D^{(l)} = diag(\sigma'(z^{(l)}))$
end

Paso *backward* - cálculo de gradientes
$\delta^{(L)} = D^{(L)}(a^{(L)} - y)$
for $l = L - 1$ *hasta* 2 **do**
  $\delta^{(l)} = D^{(l)}(W^{(l+1)})^T \delta^{(l+1)}$
end

Paso *backward* - actualización de pesos y sesgos
for $l = L$ *hasta* 2 **do**
  $W^{(l)} = W^{(l)} - \eta \delta^{(l)} a^{(l-1)T}$
  $b^{(l)} = b^{(l)} - \eta b^{(l)}$
end

Algorithm 1: Entrenamiento de una red neuronal.

2.4. Entrenamiento por Lotes de Muestras

Hasta ahora se ha explicado cómo funciona el entrenamiento de una red neuronal y los pasos que realiza esta desde que recibe una muestra hasta que actualiza la red en función de la salida obtenida. Esta versión simple aplica los pasos *forward* y *backward* solo a una muestra. Sin embargo, a la hora de poner este entrenamiento en práctica, nos encontramos con un problema computacional: el número de operaciones en coma flotante (*flops*: *floating point operations*) que se realizan respecto al número de accesos a memoria (*memops**) necesarios para llevar a cabo cada operación. La cantidad de accesos a memoria que se deben realizar en esta versión implica, para casi todas las arquitecturas, una limitación en la velocidad del entrenamiento condicionada por la memoria donde residen los datos. Esto se debe a que las operaciones implicadas, tanto en el paso *forward* como en el *backward*, consisten en multiplicaciones matriz-vector, generalmente limitadas por el ancho de banda a memoria.

* *memops* (*memory operations*): número de operaciones de memoria.

Como alternativa a esta versión se define el entrenamiento por lotes de muestras, que soluciona la limitación impuesta por memoria convirtiendo los productos matriz-vector en productos de matrices. Esta versión realiza los pasos *forward* y *backward* de forma simultánea para un conjunto de muestras de entrada, al que llamaremos lote (*batch*). Al realizar los cálculos para varias muestras de entrada a la vez, el número de accesos a memoria necesarios por cada paso disminuye, convirtiendo así la limitación por memoria en una limitación por cómputo cuando el tamaño de los lotes es lo suficientemente grande.

En la versión original, se disponía de un conjunto total de muestras de entrada que se procesaban una a una. De esta forma en cada paso *forward* se obtenía un error con el que se actualizaban los pesos en el paso *backward*. Estos pesos mejorados son los que se utilizaban para la siguiente muestra de entrada. En esta nueva versión de entrenamiento por lotes, dividiremos el conjunto total de muestras de entrada en varios lotes de muestras. Todas las muestras que hay en un mismo lote realizarán el paso *forward* a la vez, calcularán su error y actualizarán conjuntamente los pesos, en función del error, que se utilizarán para todas las muestras de entrada del siguiente lote. Durante el paso *backward* se debe tener en cuenta que se está trabajando de forma conjunta y que el error se debe promediar en consecuencia, no acumular. Así, en la Ecuación (2.8), el valor de N que representaba el número de neuronas de la última capa, $N = n^{(L)}$, en el entrenamiento por lotes representará el número total de neuronas de dicha capa en el lote, esto es, $N = n^{(L)} \times$ tamaño del lote.

En el entrenamiento por lotes las dimensiones de algunas de las variables involucradas varían, incluyendo una nueva dimensión que hará referencia al número de muestras que forman cada lote, al que denotaremos como b. En concreto, tendrán una nueva dimensión aquellas variables que se obtienen a partir de las muestras de entrada: las activaciones $a^{(l)}$ y los gradientes $\delta^{(l)}$, así como los vectores intermedios $z^{(l)}$. Se utilizará la notación $A^{(l)}$, $G^{(l)}$ y $Z^{(l)}$ para denotar respectivamente dichas variables extendidas.

El tamaño de la nueva dimensión, b, se mantendrá constante en todas las capas de la red, a diferencia de las otras dimensiones (alto, ancho y número de canales), que pueden variar según la capa. Por lo tanto, considerando que la dimensión en

el entrenamiento básico de los vectores $a^{(l)}$, $\delta^{(l)}$ y $z^{(l)}$ era $n^{(l)}$, la dimensión en el entrenamiento por lotes de las matrices $A^{(l)}$, $G^{(l)}$ y $Z^{(l)}$ es $n^{(l)} \times b$.

En el caso de las capas FC, cuando el entrenamiento se realiza para un lote compuesto por b muestras de entrada, tendremos b conjuntos de activaciones y de gradientes en cada capa. Por lo tanto, todos los productos matriz-vector involucrados en el paso *forward* y en el paso *backward*, se convierten en productos de matrices; quedando las operaciones como siguen:

- Paso *forward*:

$$A^{(l)} = \sigma(Z^{(l)}) = \sigma\left(W^{(l)} \cdot A^{(l-1)} + B^{(l)}\right) \tag{2.10}$$

- Paso *backward* - Cálculo de gradientes:

$$\begin{aligned} G^{(L)} &= D^{(L)} \circ (A^{(L)} - y), \qquad y \in \mathbb{R}^{n^{(L)} \times b}, \\ G^{(l)} &= D^{(l)} \circ (W^{(l+1)})^T G^{(l+1)}, \qquad 2 \leq l < L. \end{aligned} \tag{2.11}$$

- Paso *backward* - Actualización de pesos:

$$\begin{aligned} W^{(l)} &= W^{(l)} - \eta G^{(l)} A^{(l-1)T}, \\ b^{(l)} &= b^{(l)} - \eta \cdot \uplus G^{(l)}, \end{aligned} \tag{2.12}$$

  donde $B^{(l)} = (b^{(l)}, b^{(l)}, \cdots, b^{(l)}) \in \mathbb{R}^{n^{(l)} \times b}$, $D^{(L)} = \sigma'(Z^{(l)})$ y $\uplus$ denota la suma de todos los elementos sobre la dimensión coincidente con los sesgos, esto es, $\uplus G^{(l)} = \sum_{j=1}^{b} G^{(l)}_{(i,j)} \in \mathbb{R}^{n^{(l)}}$.

En la Figura 2.10 puede observarse cómo se extienden las activaciones y los gradientes, manteniendo la estructura de las operaciones.

Para las capas convolucionales, también se amplían las activaciones y los gradientes en una nueva dimensión de tamaño b en el entrenamiento por lotes de muestras. De esta forma, cada capa tendrá un tamaño $k^{(l)} \times n^{(l)} \times m^{(l)} \times b$. En cambio, al igual que en las capas FC, los pesos se mantienen iguales y se aplican sobre todas las muestras que forman el lote de forma análoga. En cuanto a los sesgos, estos se aplican y se actualizan de manera análoga teniendo en cuenta la diferencia entre las dimensiones de este tipo de capas respecto a las

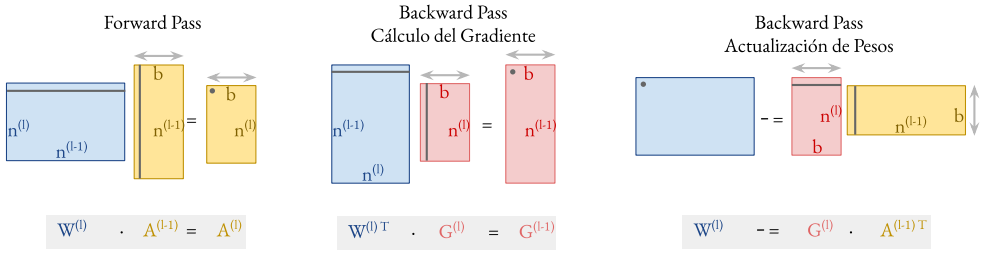

Figura 2.10: Operaciones del entrenamiento por lotes para capas FC.

FC: en el paso *forward* la matriz $B^{(l)}$ replicará los sesgos en todas las dimensiones de forma que $B^{(l)} \in \mathbb{R}^{k^{(l)} \times h^{(l)} \times w^{(l)} \times b}$; y en la actualización de los sesgos $\uplus G^{(l)} = \sum_{i=1}^{h^{(l)}} \sum_{j=1}^{w^{(l)}} \sum_{k=1}^{b} G^{(l)}_{(x,i,j,k)} \in \mathbb{R}^{k^{(l)}}$.

La ampliación en la convolución para b muestras puede representarse también en su variante *im2col*, extendiendo las matrices de activaciones y gradientes con las columnas que representan cada conjunto de muestras del lote.

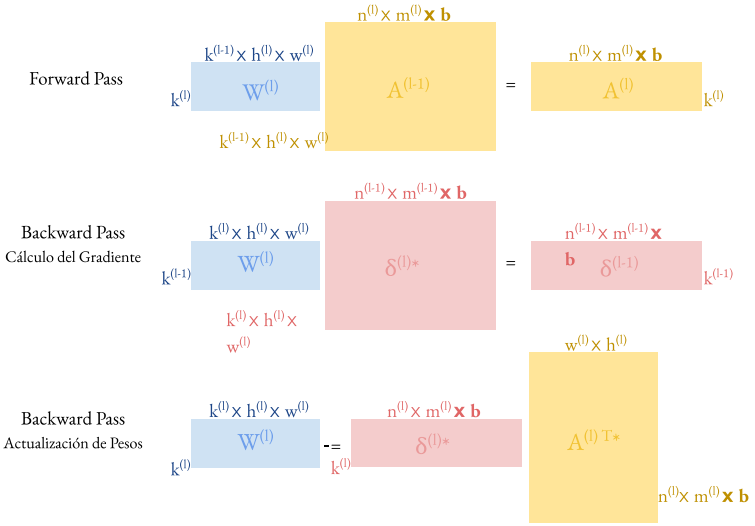

Figura 2.11: Operaciones del entrenamiento por lotes para capas convolucionales utilizando productos de matrices.

Al trabajar por lotes el entrenamiento se convierte en un conjunto de productos de matrices, GEMM (*General Matrix Multiply*), que pueden ejecutarse con llamadas a la biblioteca BLAS (Basic Linear Algebra Subprograms) [27], y en particular a versiones especializadas de la misma para arquitecturas multinúcleo

como OpenBLAS, BLIS, Intel MKL; o como cuBLAS para procesadores gráficos (GPU).

2.5. Aceleración del Procesamiento en las Redes Neuronales

Una de las técnicas más extendidas para acelerar un proceso computacional consiste en explotar el paralelismo. En la sección anterior, se ha expuesto que tanto en el entrenamiento como en la inferencia de las redes neuronales, existen múltiples dependencias a tener en cuenta si se desea aplicar una paralelización. Por ejemplo, durante la etapa *forward* las activaciones se propagan entre las capas adyacentes, requiriendo las activaciones de la capa *l+1* las de la capa l para su cálculo ($A^{(l+1)} = f(A^{(l)})$). Durante el paso *backward* del entrenamiento también ocurre una propagación entre capas contiguas, siendo el gradiente de la capa *l+1* necesarios para el cálculo del gradiente de la capa l ($G^{(l)} = f(G^{(l+1)})$); ver la Figura 2.12. Existen en ambas situaciones, por tanto, unas dependencias estrictas entre capas que impiden la paralelización de los procesos de entrenamiento e inferencia mediante una división por capas de la red.

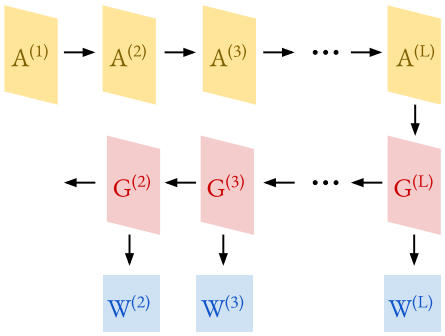

Figura 2.12: Grafo de dependencias en el entrenamiento de una red neuronal.

Así pues, la paralelización de estos procesos queda restringida a dividir únicamente el trabajo internamente intrínseco a cada capa: paralelismo intracapa. En este sentido, tanto la extensión de las activaciones derivada del uso de lotes, como la divisibilidad de las matrices de pesos en función de las activaciones que calculan, permite particionar las operaciones involucradas en el entrenamiento

entre diferentes réplicas. Si tenemos en cuenta que todos los cálculos se pueden realizar cómo una operación entre matrices* de la forma $A = B\dot{C} + D$, la paralelización de los procesos de entrenamiento e inferencia consistirá en la división de esta operación, pudiendo distribuir entre las réplicas la matriz B y/o la C, es decir, distribución de muestras y/o pesos [9].

En las siguientes subsecciones se describen las principales estrategias de partición [9]: 1) aquellas que mantienen el orden estricto de las operaciones realizando una partición intracapa ya sea por muestras de entrada –paralelismo de datos (Sección 2.5.1)– o por pesos de la red –paralelismo de modelo (Sección 2.5.2)–; 2) la estrategia *pipelining* (Sección 2.5.3) que hace una división por capas alterando el método clásico de entrenamiento; y, por último, 3) la combinación de estas diferentes opciones para un paralelismo híbrido (Sección 2.5.4).

2.5.1. Paralelismo de datos

Esta paralelización replica el modelo de la red neuronal y reparte las muestras de entrada entre las réplicas involucradas en el entrenamiento. Es decir, distribuye el trabajo de entrenamiento entre las réplicas dividiendo los lotes de muestras de entrada que se deben procesar en cada iteración; ver la Figura 2.13 [9, 13, 77].

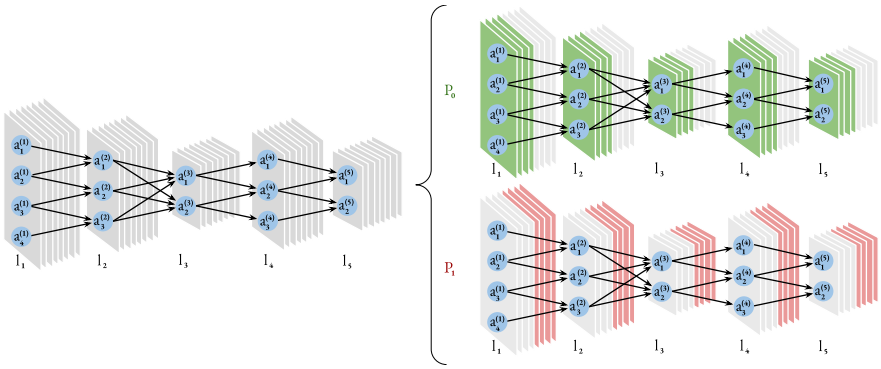

Figura 2.13: Aprovechamiento del paralelismo de datos con dos réplicas en el paso *forward* de una red neuronal.

*Recordar que el cálculo de una convolución puede convertirse, mediante el uso de la transformación *im2col* o *im2row*, en un producto de matrices.

En este esquema, cada réplica aplica los pasos *forward* y *backward* únicamente al subconjunto de muestras del lote que le han sido asignadas, de modo que cada réplica puede realizar las operaciones necesarias para el cálculo de las activaciones y de los gradientes asociados a sus muestras sin requerir comunicación alguna con las restantes réplicas. No obstante, para mantener el mismo procedimiento que se realiza en un entrenamiento secuencial, es necesario coordinar las réplicas durante la actualización de pesos, de modo que esta tenga en cuenta los errores observados en todos los subconjuntos de muestras y todas las réplicas puedan proseguir el entrenamiento del siguiente lote de muestras con los pesos igualmente actualizados.

Centrándonos en las operaciones asociadas al entrenamiento, la implicación que tiene este modelo de paralelismo sobre los tres cálculos que intervienen –activaciones, gradientes y actualización de pesos– resulta en la división de las matrices asociadas a las muestras: activaciones y gradientes. Por lo tanto, las tres operaciones que aparecen durante el entrenamiento, Ecuaciones 2.10, 2.11 y 2.12, quedan particionadas para cada réplica p_i de la siguiente manera:

$$\begin{aligned} A_{p_i}^{(l)} &= \sigma\left(W^{(l)} A_{p_i}^{(l-1)} + B^{(l)}\right), \\ G_{p_i}^{(l)} &= W^{(l+1)T} G_{p_i}^{(l+1)}, \\ W^{(l)} &= W^{(l)} - \eta \sum_{i=1}^{P} G_{p_i}^{(l)} A_{p_i}^{(l-1)T}, \\ b^{(l)} &= b^{(l)} - \eta \sum_{i=1}^{P} \uplus G_{p_i}^{(l)}. \end{aligned}$$

Durante la actualización de pesos y sesgos debe realizarse la comunicación entre las réplicas que se ha mencionado anteriormente, de tipo *AllReduce* [26], para mantener una coherencia global de todas las réplicas.

En la Figura 2.14 se muestra un esquema de este reparto para P réplicas, de forma que la notación p_i dentro de una sección de la matriz indica que dicha fracción de datos y cálculos se asociará a la réplica p_i.

Un factor a tener en cuenta en los modelos de paralelismo es su capacidad de escalar. En el paralelismo de datos es fácil conseguir que la división de muestras entre las réplicas preserve la carga de trabajo por réplica, bastando para ello con

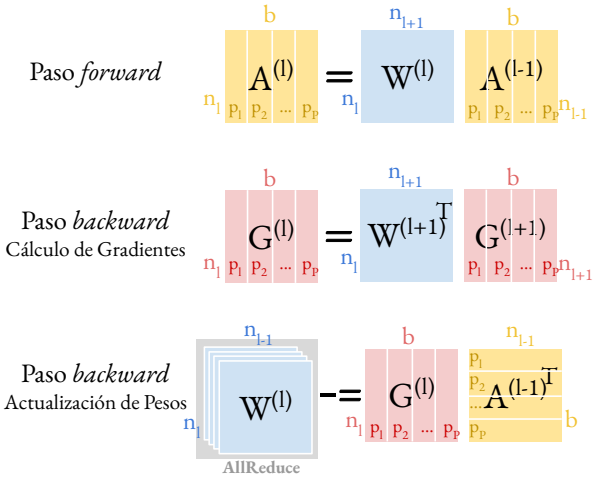

Figura 2.14: División entre P réplicas del entrenamiento de una red neuronal usando paralelismo de datos.

incrementar el tamaño de los lotes que se entrenan proporcionalmente al número de réplicas que participan.

Por último, cabe mencionar que el paralelismo de datos posibilita una superposición de cálculo y comunicaciones que permite mejorar el rendimiento del entrenamiento. Aunque sí se deben considerar las dependencias existentes entre el cálculo de gradientes contiguos ($G_{p_i}^{(l)} \rightarrow G_{p_i}^{(l-1)}$) así como entre el cálculo de los gradientes parciales y la comunicación que permite actualizar los pesos ($G_{p_i}^{(l)} \rightarrow \sum_{i=1}^{P} G_{p_i}^{(l)} A_{p_i}^{(l-1)T}$), ambos son independientes entre sí, por lo que puede superponerse el cálculo de los siguientes gradientes con las comunicaciones; es decir, realizar las comunicaciones de modo asíncrono y así acelerar el entrenamiento.

2.5.2. Paralelismo de modelo

Esta estrategia de paralelismo, al contrario que el paralelismo de datos, replica las muestras mientras divide los parámetros de la red, pesos y sesgos, entre las réplicas [12]. De esta manera, cada réplica aplicará solo una parte de la red neuronal a todas las muestras, tal y cómo se observa en la Figura 2.15. La distribución de estos parámetros se realiza mediante la partición de la matriz que representan.

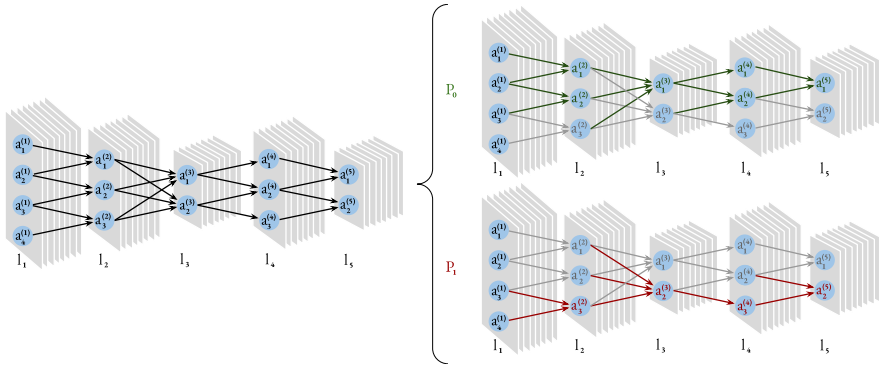

Figura 2.15: División entre P réplicas del entrenamiento de una red neuronal usando paralelismo de modelo.

Obsérvese en este esquema que, al repartir los parámetros de la red, lo que ocurre es que cada réplica obtiene únicamente una sección de las activaciones calculadas, quedando estas repartidas y siendo necesario, por tanto, volver a compartir las activaciones antes de aplicar los pesos de las siguientes capas, pues estos se deben aplicar sobre el conjunto completo.

El resultado de aplicar este modelo de paralelismo sobre los cálculos involucrados –activaciones, gradientes y actualización de pesos–, deriva en las siguientes operaciones:

$$
\begin{aligned}
A_{p_i}^{(l)} &= \sigma\left(W_{p_i}^{(l)} A^{(l-1)} + B_{p_i}^{(l)}\right), \\
G_{p_i}^{(l)} &= Scatter\left(\sum_{i=1}^{P} D^{(l)} W_{p_i}^{(l+1)T} G_{p_i}^{(l+1)}\right), \\
W_{p_i}^{(l)} &= W_{p_i}^{(l)} - \eta G_{p_i}^{(l)} A^{(l-1)T}, \\
b_{p_i}^{(l)} &= b_{p_i}^{(l)} - \eta \biguplus G_{p_i}^{(l)}.
\end{aligned}
$$

Como puede observarse tanto en las ecuaciones como en su representación en el esquema de la Figura 2.16, la dependencia entre capas que se ha mencionado existe en el cálculo de activaciones así como en el cálculo de gradientes. Esto supone una gran limitación de este modelo de paralelismo ya que requiere de una comunicación, de tipo *AllGather*, para reunir las activaciones parciales obtenidas y otra comunicación, de tipo *Reduce-Scatter* [26], para que cada proceso consiga el cálculo completo del gradiente que le corresponde. Otra limitación de esta

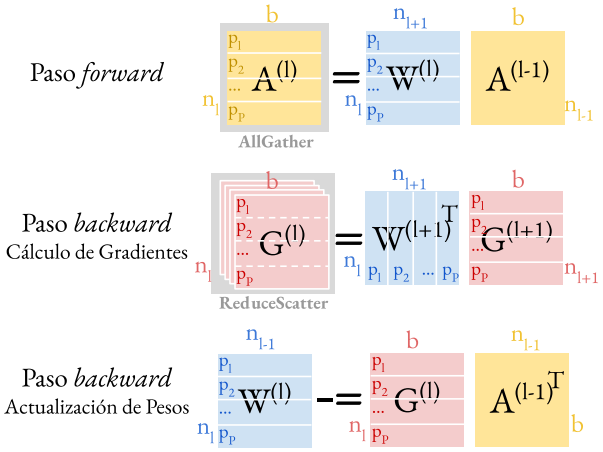

Figura 2.16: Distribución entre P réplicas del entrenamiento de una red neuronal usando paralelismo de modelo.

estrategia de paralelismo es la distribución del modelo en sí, pues no es posible escalar el tamaño global de los datos a repartir respecto a la cantidad de procesos que participan para optimizar el rendimiento de la paralelización.

Finalmente, cabe mencionar que el paralelismo de modelo admite una mejora del rendimiento mediante la conservación de las particiones en aquellas capas que no tienen pesos y sesgos. Dado que no todos los tipos de capas cuentan con estos parámetros entrenables, como es el caso de las capas *Dropout* o *Pool*, existe la opción de mantener la distribución de las activaciones durante estas capas, y únicamente realizar la comunicación antes del cálculo de las siguientes capas con parámetros entrenables.

2.5.3. Paralelismo *pipeline*

La paralelización *pipelining* es una estrategia de aceleración que varía la secuencialidad del proceso de entrenamiento, pues evade las dependencias se han mencionado anteriormente. En este esquema de paralelización se superpone el paso *backward* de cada lote de datos con el paso *forward* de los siguientes lotes, de modo que estos últimos empiezan las operaciones de las activaciones con los pesos desactualizados al no haberse finalizado la actualización de los pesos del lote anterior [31]. Esta modalidad de entrenamiento con pesos no-actualizados se

denomina entrenamiento asíncrono; mientras que en los esquemas de paralelismo estudiados en las secciones anteriores –datos y modelo–, que siempre trabajan con pesos actualizados, el entrenamiento se clasifica como síncrono.

Al relajar de este modo las restricciones por dependencias, se permite realizar la distribución del trabajo particionando las capas de la red entre las réplicas y obteniendo una paralelización que parece una combinación del paralelismo de datos con uno de capas. En la Figura 2.17 se ilustra el concepto de este paralelismo: cada réplica p_i trabaja sobre una sección de las capas de la red en las que le aplica los pasos *forward* y *backward*, F^i y B^i respectivamente, sobre cada lote de datos representados por diferentes colores.

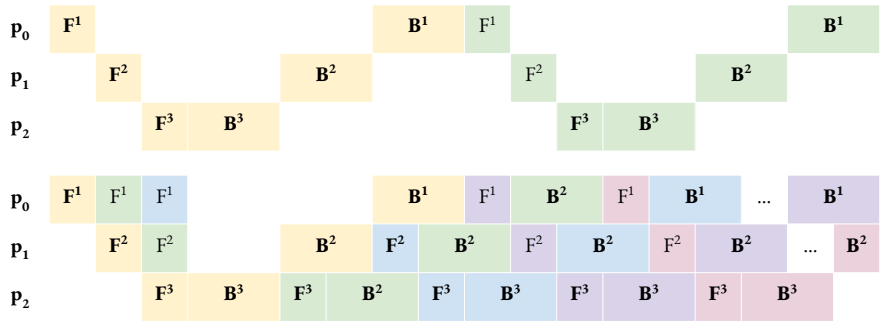

Figura 2.17: Esquema secuencial y paralelizado mediante *pipelining*.

Como inconveniente de este modelo, el hecho de entrenar con pesos parcialmente actualizados conlleva una reducción de la velocidad de convergencia y, en consecuencia, la necesidad de prolongar el entrenamiento hasta obtener la precisión deseada.

2.5.4. Paralelismo híbrido

Como se ha mencionado al inicio de la sección, otra alternativa interesante para paralelizar el entrenamiento de las redes neuronales proviene de la combinación de diferentes esquemas de paralelismo, denotando a este paralelismo híbrido [30].

A modo de ejemplo, uno de los esquemas mixtos más populares consiste en combinar el paralelismo de datos y de modelo, de modo que el primero es aplicado a nivel global distribuyendo el conjunto de muestras, mientras que el paralelismo de modelo se aplica para cada uno de estos lotes de muestras, siendo ahora

distribuidos los pesos del modelo. De esta forma, el hecho de combinar diferentes esquemas de paralelismo permite combinar también sus características y obtener un mayor rendimiento de los recursos disponibles. En particular, el sistema de memoria utilizado es una de las propiedades que pueden ser aprovechadas en modo híbrido, combinando el paradigma de programación de memoria distribuida (paralelismo de datos) y de memoria compartida (paralelismo de modelo).

Este esquema de paralelismo, por lo tanto, ofrece la posibilidad de explotar los diferentes modelos de paralelismo en conjunto, aprovechando así los beneficios de cada uno, aumentando la capacidad de los sistemas computacionales.

2.6. Estado del Arte

Como se ha comentado al inicio de este capítulo, el entrenamiento de modelos de RNP es un proceso muy costoso desde el punto de vista computacional que habitualmente se realiza en supercomputadores a fin de reducir el tiempo de procesamiento. Existe una razón subyacente económica: Tras esta fase de aprendizaje inicial, las grandes compañías (Google, Facebook, Baidu, etc.) despliegan sus modelos entrenados sobre sus centros de datos así como en las "Apps" instaladas en los dispositivos personales de los usuarios [60, 74]: El propósito es proporcionar una mejor experiencia a los usuarios, pero también incrementar el retorno para la empresa. ¡El tiempo es dinero! En este sentido, estamos habituados a que las Apps de nuestros dispositivos reciban actualizaciones continuas, y todos nos sorprendemos y a veces nos alegramos de cómo de "bien" los móviles aprenden nuestros gustos personales a través del uso diario, o cómo han mejorado las técnicas de reconocimiento del habla. Detrás de este proceso de aprendizaje hay sistemas de recomendación personalizados, y algoritmos de traducción/transcripción automáticos afinados para nuestra voz, basados en RNP.

Desde el punto de vista empresarial, las compañías están fuertemente interesadas en reducir el tiempo de entrenamiento para evaluar el resultado de posibles mejoras en sus modelos [60, 74]. La utilización de tecnologías de aprendizaje profundo en las aplicaciones científicas se puede beneficiar, igualmente, de la aceleración del proceso de entrenamiento.

El proceso de entrenamiento se suele realizar sobre un sistema HPC, habitualmente un clúster de computadores con nodos equipados con algún tipo de acelerador *hardware*. El ejemplo más habitual es emplear uno o más procesadores gráficos (*Graphics Processing Units* o GPU) por cada nodo de cómputo, habitualmente de la compañía NVIDIA [57]. Cabe hacer aquí mención especial a los sistemas de aceleración propios de Google, basados en la arquitectura *Tensor Processing Unit* (TPU), que la empresa emplea en sus centros de datos (y ofrece a través de la nube) para acelerar el procesamiento y reducir el consumo energético de procesos de IA [40].

Los *entornos de entrenamiento distribuido* –esto es, entornos de entrenamiento sobre clústers– más difundidos, como TensorFlow (TF), Keras o PyTorch, explotan el paralelismo de datos cuando se ejecutan, procesando lotes de entradas independientes de manera simultánea [9, 64, 68]. Concretamente, en el paralelismo de datos se aprovecha la independencia entre las muestras que componen un lote, asignando el procesamiento de un grupo de estas a cada nodo del clúster. Horovod, por otro lado, proporciona un entorno eficiente para la paralelización distribuida utilizando el esquema paralelismo de datos para el entrenamiento de modelos a escala, diseñado para integrarse de manera óptima con TensorFlow, Keras y PyTorch [41, 64]. Este tipo de procesamiento no es totalmente independiente de modo que, durante el mismo, los nodos deben intercambiar sus resultados parciales, a fin de construir un conocimiento global. Las ventajas principales del paralelismo de datos son su sencillez de programación, su eficiencia –siempre que el lote comprenda un número suficiente de muestras–, el moderado volumen de comunicaciones, el bajo número de puntos de sincronismo, y la posibilidad de solapar cálculos y comunicaciones. En contra tiene la necesidad de replicar el modelo en cada uno de los nodos del clúster, lo que requiere una alta capacidad de almacenamiento en los nodos (y aceleradores), o el uso de técnicas de programación de almacenamiento en disco (*out-of-core computing*). Para concluir esta breve revisión del paralelismo de datos, cabe mencionar que, puesto que la "dimensión" que se paraleliza en este esquema es el lote, teóricamente es posible aumentar el número de muestras del mismo de manera proporcional al número de nodos para mantener la escalabilidad. Sin embargo, aumentar el tamaño del lote tiene consecuencias sobre la convergencia del proceso de entrenamiento, y suele

requerir un ajuste muy meticuloso de otros parámetros del proceso (en particular, el ratio de aprendizaje) [68, 76, 77].

Como alternativa, en el paralelismo de modelo el trabajo se reparte entre los nodos en alguna de las "otras dimensiones" del problema. En el caso de capas densas FC, estas dimensiones alternativas corresponden a las activaciones de entrada o salida a la propia capa. En el caso de capas convolucionales, el paralelismo suele explotar a nivel de núcleo de convolución (*kernel*) [9]. El paralelismo de modelo presenta fuertes restricciones en cuanto a su escalabilidad: Dado un modelo de RNP, el número de neuronas en las capas densas así como la cantidad de kernels en las capas convolucionales quedan fijados. Además, sus valores oscilan habitualmente entre unos pocos cientos y miles de neuronas/kernels. Por tanto, al aumentar el número de nodos de una instalación, cabe esperar que aparezcan fuertes cuellos de botella en la escalabilidad de este esquema de paralelización. En consecuencia, salvo unas pocas excepciones como Mesh TensorFlow, los entornos de entrenamiento distribuidos explotan el paralelismo de modelo internamente a los nodos, pero raramente lo hacen entre nodos [65].

Existen varios estudios del grado de paralelismo de distintos esquemas de paralelización, principalmente orientados a la ejecución de los procesos de entrenamiento y/o inferencia sobre aceleradores hardware o nodos de computación compuestos por varios de estos aceleradores (tipo GPU o TPU); ver, por ejemplo, [59, 68] para una lista de referencias completa y, en el caso del primer trabajo, actualizada. En cambio, los estudios de escalabilidad paralela a nivel de clúster son mucho más escasos.

Capítulo 3

Entorno Paralelo de Entrenamiento e Inferencia de Redes Neuronales

En este capítulo se presenta PyDTNN, acrónimo que corresponde a *Python Distributed Training of Neural Networks*, y que aporta un entorno desarrollado en el marco de esta tesis para trabajar con redes neuronales [4, 5]. Las secciones del capítulo describen las características principales de este entorno, en particular, su funcionalidad, estructura y extensiones. Así mismo, también se muestra cómo se explota el paralelismo en PyDTNN y las prestaciones que ofrece el entorno como herramienta de entrenamiento de redes neuronales.

3.1. Motivación

Durante los últimos años se ha producido un vertiginoso crecimiento en el uso y desarrollo de las redes neuronales. No sólo resultan cada vez más habituales en nuestro día a día y en una gran variedad de dispositivos, sino que también es mayor su complejidad y más abundante la cantidad de datos a procesar. Esto ha derivado en un incremento masivo de la carga de trabajo que conlleva trabajar con redes neuronales, y en una creciente necesidad de mejorar la eficiencia de la inferencia y del propio proceso de entrenamiento. Bajo estas condiciones, se han desarrollado entornos bien conocidos, como TensorFlow, PyTorch y Keras

entre otros, que permiten el entrenamiento y la inferencia de redes neuronales de manera eficiente y con una interfaz amigable [61, 69]. Sin embargo, las altas prestaciones que estas herramientas ofrecen llevan consigo una complejidad de los códigos subyacentes que dificulta su comprensión, hasta el punto de que realizar pequeñas modificaciones resulta una tarea únicamente al alcance de expertos. Dificultando, por ejemplo, la modificación del código con el fin de explorar técnicas alternativas durante el proceso de entrenamiento.

En este sentido, en el ámbito de la investigación a menudo resulta fructífero poder de indagar, probar y entender qué hay detrás de una técnica o un algoritmo, así como estudiar el efecto que se observa al introducir modificaciones basadas en una idea. Con esa perspectiva se creó el entorno PyDTNN, es decir, con el propósito de entender y mejorar los procesos que hay detrás del aprendizaje profundo.

El entorno PyDTNN que aquí se presenta, no se crea con el propósito de competir con otros entornos profesionales en producción, sino como ejercicio para entender los detalles inherentes al entrenamiento y la inferencia de las redes neuronales. De este modo, PyDTNN se ha desarrollado de un modo más accesible para, además de facilitar los procesos de inferencia y entrenamiento, permitir implementar de forma asequible múltiples modificaciones y ampliaciones como, por ejemplo, nuevas técnicas de aceleración o compresión.

3.2. Una Breve Visión de PyDTNN

El entorno PyDTNN está diseñado para proporcionar las funcionalidades básicas que permiten crear, inferir y entrenar tanto redes neuronales básicas de tipo MLP como redes convolucionales, abarcando así una gran variedad de modelos, tales como DenseNet, ResNet y VGG [33, 36, 66]. Así mismo, también facilita la modificación del entorno para su personalización y extensión, de modo que sea posible incluir nuevas funcionalidades. Bajo estas premisas, el entorno PyDTNN se diseña como una herramienta para el prototipado de nuevas ideas aplicadas al entrenamiento e inferencia de RNP.

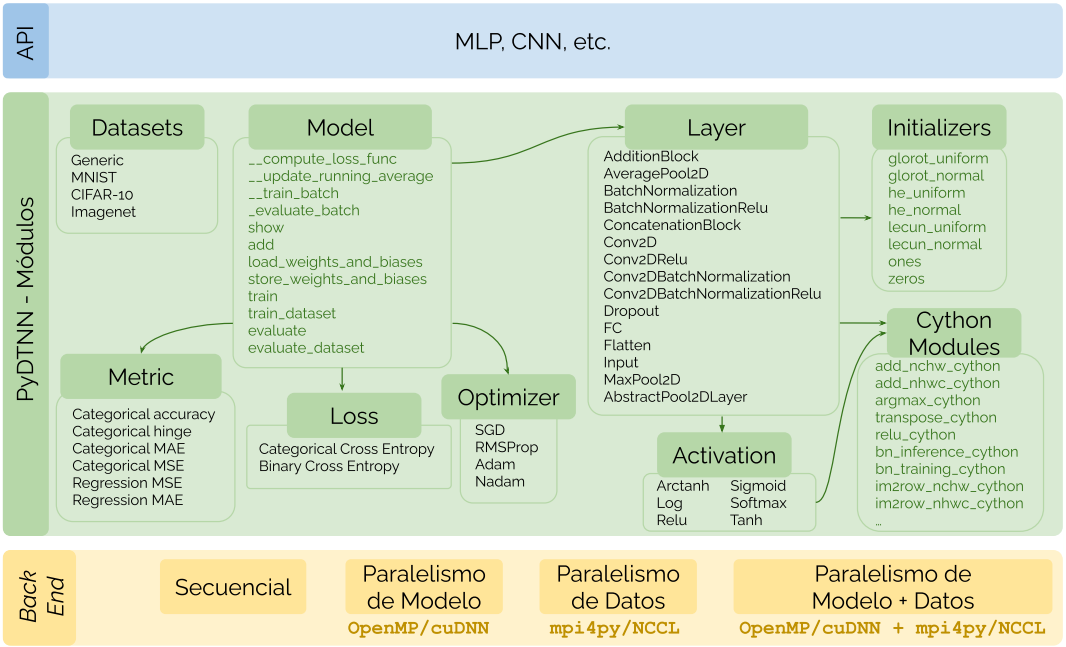

Figura 3.1: Esquema de la arquitectura de PyDTNN.

En la Figura 3.1 se ofrece un esquema que presenta y relaciona los principales módulos de PyDTNN, los cuales se detallan más en profundidad en la siguiente sección.

Bajo la premisa de aportar un entorno que facilite al usuario trabajar con redes neuronales, PyDTNN se ha diseñado de modo que la curva de aprendizaje inicial sea bastante plana, tanto para los aprendices, como para los expertos cuyo objetivo sea una interacción más profunda con el entorno. Para ello, PyDTNN mantiene una interfaz similar a la de Keras, uno de los paquetes más populares de aprendizaje profundo, y se ha desarrollado en el lenguaje de alto nivel Python [22]. Como muestra de la interacción con PyDTNN, el Código 3.1 ilustra cómo se implementa y entrena la red ResNet-32 para el conjunto de datos CIFAR-10 [43]. Este ejemplo expone la interacción habitual que el usuario tiene con el entorno. Tal y como se observa, se siguen para ello tres pasos:

1. Crear la propia red neuronal añadiendo al modelo cada una de las capas que lo componen;
2. Cargar los datos necesarios para el aprendizaje; e

3. Invocar la función de entrenamiento de la red con los parámetros que se deseen.

```
# 1) Definir el modelo agregando las capas necesarias
resNet32 = Model()
resNet32.add( Input(shape=(32, 32, 3)) )
resNet32.add( Conv2D(nfilters=16, filter_shape=(3, 3), stride=1, padding=1) )
resNet32.add( BatchNormalization() )
layout = [ [16, 5, 1], [32, 5, 2], [64, 5, 2] ] #Bloques de capas
for n, b, s in layout:
  for r in range(res_blocks):
    if r > 0: s = 1
    resNet32.add( AdditionBlock(
      [ Conv2D(nfilters=n, filter_shape=(3, 3), stride=s, padding=1),
        BatchNormalization(), Relu(),
        Conv2D(nfilters=n, filter_shape=(3, 3), stride=1, padding=1),
        BatchNormalization()
      ], [
        Conv2D(nfilters=n, filter_shape=(1, 1), stride=s),
        BatchNormalization()
      ] if s != 1 else [] ) )
    resNet32.add( Relu() )
resNet32.add( AveragePool2D(pool_shape=(0,0)) ) # Global average pooling 2D
resNet32.add( Flatten() )
resNet32.add( FC(shape=(64,)) ); r32.add( BatchNormalization() ); r32.add( Relu() )
resNet32.add( FC(shape=(10,), activation="softmax") )

# 2) Cargar CIFAR-10 como conjunto de datos para el entrenamiento
dataset = get_dataset("cifar10")

# 3) Definir los parametros de entrenamiento y entrenar la red
lr, n_epochs, batch_size = 0.1, 100, 64

resNet32.train_datset(dataset, n_epochs, batch_size, loss="categorical_cross_entropy",
          metrics="accuracy", optimizer=optimers.SGD(learning_rate=lr))
```

Código 3.1: Código para crear y entrenar el modelo ResNet-32 con CIFAR-10 en el entorno PyDTNN.

3.3. Clases y Métodos

El entorno PyDTNN, al igual que las redes neuronales en sí, tiene dos partes principales: el modelo que define la red neuronal como conjunto y las capas que lo componen. Del mismo modo, existen dos clases esenciales en PyDTNN: `Model` y `Layer`. A continuación se describen las propiedades más representativas de cada una de estas clases.

`Model`. Esta podría considerarse la clase principal, pues define las características y los métodos más relevantes de cara al usuario. Mediante ella, el usuario crea y entrena la red neuronal, especificando las características que desea utilizar: ratio

de aprendizaje, número de épocas, tamaño de muestras en el lote, etc. Además de las características básicas que permiten realizar lo anterior, explicadas a continuación, se han ido añadiendo nuevos métodos para ampliar la funcionalidad de PyDTNN.

Para definir una red, el usuario solo debe crear un objeto de la clase `Model` y después añadir, de forma ordenada, las capas que se desea que compongan la red. Para añadir una capa al modelo, esta clase cuenta con una función `add` (Código 3.2), a la que debe indicarse qué tipo de capa quiere añadirse (FC, convolucional, etc.) y sus especificaciones si así lo requiere. El objeto `Model` mantiene en uno de sus atributos (`layers`) la relación de capas que se le añaden.

```
def add(self, layer):
        # Se indica que la capa pertenece al modelo
        layer.set_model(self)
        # Se inicializa la capa con los parametros necesarios como el tam. de entrada
        layer.initialize(...)
        #Se incrementa el numero de parametros del modelo
        self.nparams += layer.nparams
        #Se agrega la capa a la lista de capas del modelo
        self.layers.append(layer)
        #Si la capa tiene activaciones, estas se agregan como otra capa
        if layer.act:
            self.add(layer.act())
```

Código 3.2: Esquema del método `add` de la clase `Model`.

Así mismo, de esta clase cabe destacar los métodos `train_dataset` y `train_batch` (Códigos 3.3 y 3.4), que realizan el entrenamiento de la red neuronal. Se definen también de manera análoga los métodos `evaluate_dataset` y `evaluate_batch` para realizar el proceso de inferencia de una red sobre un conjunto de datos.

```
def train_dataset(self, dataSet, n_epochs, batch_size, ...):
    # En cada epoca...
    for epoch in range(n_epochs):

        # Generar los los lotes de entrenamiento y validacion
        tr_b_generator, val_b_generator = dataSet.get_train_val_generator(batch_size, ...)
        # Entrenar cada lote de entrenamiento y actualizar la perdida total
        for x, y, b in tr_b_generator:
            train_batch_loss = self.__train_batch(x, y, b, ...)
            train_total_loss = self.__update_running_average(train_batch_loss, ...)

        # Para la validacion, inferir cada lote de validacion y actualizar la perdida total
        for x, y, b in val_b_generator:
            val_batch_loss = self.__evaluate_batch(x, y, b, ...)
            val_total_loss = self.__update_running_average(val_batch_loss, ...)
```

Código 3.3: Método `train_dataset` de la clase `Model`.

```
def __train_batch(self, x_lote, y_lote, ...):
  # Paso Forward
  x = x_lote
  for l in range(1, len(self.layers)):
    x = self.layers[l].forward(x)

  # Paso Backward - Calculo del gradiente
  loss, dx = loss_func(x, y_lote)
  for l in range(len(self.layers)-1, 0, -1):
    dx = self.layers[l].backward(dx)

  # Paso Backward - Actualizacion de pesos
  for l in range(len(self.layers)-1, 0, -1):
    self.layers[l].reduce_weights_sync(self.comm)
    self.layers[l].update_weights(optimizer)
```

Código 3.4: Método `train_batch` de la clase `Model`.

Para llevar a cabo el entrenamiento, el conjunto de muestras que debe procesar la red se divide previamente en lotes, de forma que el subconjunto de muestras que forman cada lote pasan por el ciclo completo de entrenamiento (pasos *forward* y *backward*). La función `train_dataset` es la que realiza la división de muestras e invoca, para cada lote de muestras, a la función `train_batch` que se encarga de realizar el ciclo de entrenamiento. Como se ha mostrado en el Código 3.1, los pasos que se siguen son básicamente los que se han mencionado: crear el modelo, añadir las capas, cargar el conjunto de muestras con los que se entrenará la red, definir los parámetros de entrenamiento (ratio de aprendizaje, número de épocas y tamaño del lote) y, finalmente, invocar al método de entrenamiento `train_dataset`. La inferencia, sería semejante: una vez se tiene un modelo entrenado, se cargan las muestras que se desean inferir y se llama al método `evaluate_dataset` con los parámetros de inferencia deseados. Ver Código 3.5.

```
#Crear el modelo
model = Model(...)
#Cargar sus pesos y sesgos
model.load_weights_and_bias(model.weights_and_bias_filename)
#Cargar el conjunto de datos a evaluar
datasest = get_dataset("MNIST")
#Inferir la red con los parametros deseados
model.evaluate_dataset(dataset, batch_size, loss_func, ...)
```

Código 3.5: Esquema ejemplo de inferencia de una red con PyDTNN.

`Layer`. Aunque existen ciertos parámetros comunes a todos los tipos de capas y funciones que pueden ser utilizadas, independientemente de la capa con la que se esté trabajando, cada tipo de capa realiza los procesos de entrenamiento e inferencia de forma distinta. Así pues, existen parámetros específicos solo para

ciertas capas. Por ejemplo, las capas convolucionales y las de tipo *pooling* utilizan *padding* y *stride*, mientras que las capas FC o las de tipo *flatten* no. La forma en la que se realizan las operaciones del *forward* y *backward*, como se ha visto anteriormente, también es diferente entre cada tipo de capa. Por ello, es necesario definir ciertas características específicas para cada uno de ellas y esto se realiza en la clase `Layer`.

`Layer` tiene una clase heredada por cada tipo de capa. Como puede observarse en la Figura 3.1, entre los distintos tipos de capas que se han definido, hay capas individuales que permiten componer redes MLP, CNN, etc., y capas bloque que permiten agrupar en una única salida los resultados de diferentes capas en paralelo para crear redes residuales.

Todos los tipos de capas tienen definidas las funciones `forward`, `backward` y `update_weights` para implementar el paso *forward*, el cálculo del gradiente y la actualización de pesos respectivamente. Así pues, mientras que la clase `Layer` contiene la parte genérica, las clases heredadas implementan las particularidades necesarias según el tipo de capa. Por ejemplo, la función `update_weights` que se utiliza en la función `train_batch` (véase el Código 3.4) está implementada para todas las capas, pero solo aquellas capas con pesos (convolucionales y FC) realizan operaciones dentro de dicha función.

Además de estas dos clases principales, PyDTNN define otras clases y módulos auxiliares que sirven para esquematizar el entorno y favorecer la simplificación de todas las clases con el fin de dar una visión más comprensiva de las mismas. A continuación, se describen brevemente las características de los módulos que se muestran, junto a sus relaciones en la Figura 3.1, y que constituyen las clases, métodos y funciones más significativos y representativos del entorno:

- `Datasets`: El objetivo de este módulo y sus diversas subclases es el de englobar los principales aspectos de los conjuntos de datos con los que se entrenan, validan e infieren las redes neuronales. Consta de una clase genérica para abarcar cualquier conjunto de datos, y de otras tres clases herederas de esta, cada una de las cuales engloba los conjuntos de datos: MNIST, CIFAR-10, e ImageNet [25, 43, 48]. Algunos métodos asociados a estas clases llevan a cabo la normalización de las muestras, generación de nuevos

datos de entrenamiento aplicando transformaciones a los datos iniciales (*data augmentation*), particionado de las muestras en datos de entrenamiento y validación, etc.

- `Metric`: Mediante una clase genérica y clases herederas que implementan cada una de las métricas más conocidas, este módulo, auxiliar a la clase `Model`, proporciona las métricas que permiten tener una o diversas valoraciones sobre el rendimiento del modelo durante el paso *forward*, sin que este juicio afecte al resto del proceso de entrenamiento (paso *backward*). Las métricas actualmente incorporadas en PyDTNN son, para clasificación, precisión, *hinge*, MAE (Mean Absolute Error) y MSE (Mean Squared Error); y para regresión MAE y MSE.

- `Loss`: Análogamente al módulo `Metric`, este módulo, también complementario a la clase `Model`, calcula la pérdida o el coste que se ha generado al finalizar el paso *forward* de cada lote, así como el primer gradiente que se retropropagará durante el paso *backward*. Para calcular dicho coste, como se explicaba en el Capítulo 2, una de las funciones más conocidas es la entropía cruzada (Ec. (2.8)). En este caso, se incluyen como clases herederas las funciones de pérdida entropía cruzada *categorical cross entropy* y *binary cross entropy*.

- `Optimizer`: Este módulo incluye los métodos empleados por la clase `Model` para optimizar la función coste. Al igual que en las clases anteriores, se cuenta con una clase genérica y las clases herederas que se corresponden con algunos de los optimizadores más populares: SGD (Stochastic Gradient Descent), RMSProp (Root Mean Square Propagation), ADAM (ADaptive Moment Estimation) y NADAM (Nesterov-accelerated ADAM).

- `Initializers`: Este módulo es utilizado por la clase `Layer` para la inicialización de los pesos de las capas. Se incluyen como posibles opciones de inicialización: inicializar a ceros, a unos y mediante el uso de las distribuciones *glorot normal/uniform*, *he normal/uniform* y *lecun normal/uniform*.

- `Activation`: Sirve de apoyo a los objetos de la clase `Layer` para indicar la función de activación que dicha capa aplicará. De esta forma, dicha clase generaliza las instrucciones necesarias que deben añadirse al tipo de capa con el que se esté trabajando durante el proceso de entrenamiento o inferencia. Este módulo consta también de una clase genérica y de clases herederas para las posibles funciones a aplicar como activación: *Sigmoid*, *ReLU* (Rectified Linear Unit), *Softmax*, *arctanh*, log y tanh.

Asimismo, PyDTNN admite que el usuario agregue nuevos módulos personalizados para ampliar las funcionalidades ya definidas en base a sus necesidades; pudiendo crear, así, nuevas funciones coste, optimizadores, métricas, etc.

3.4. Extensibilidad

En este apartado se explican diferentes extensiones que se han realizado sobre PyDTNN, o que podrían ser implementadas sobre el mismo, para ilustrar la facilidad de su personalización y las posibilidades que ello abre para incrementar sus funcionalidades y rendimiento.

3.4.1. Aceleración mediante tipos de datos no convencionales

Una posible extensión de PyDTNN consiste en incluir soporte para tipos de datos de precisión reducida, que requieran menos espacio en memoria y, en consecuencia, aceleren la ejecución tanto de cómputo como de comunicaciones. Esta aceleración puede ser significativa en el entrenamiento de las redes neuronales, dado que son estos dos factores, el cómputo y las comunicaciones, los que generan cuellos de botella. El uso de precisión mixta es un modo de afrontar estas limitaciones pero, por contra, conlleva una pérdida de precisión sobre los datos cuyo efecto debe ser valorado. La flexibilidad de PyDTNN permite adaptar el entorno para incluir el uso de estos tipos de datos en aquellas secciones localizadas de código dónde pueda resultar interesante su uso para acelerar ciertas funciones a costa de una reducción de precisión asumible. Así, por ejemplo, resulta interesante emplear esta técnica en las comunicaciones, por ejemplo, comprimiendo (con

pérdida) los datos transferidos, mediante una conversión previa a un tipo de datos menos pesado y su posterior descompresión una vez la transmisión se ha completado. Aplicada de este modo, esta técnica consigue reducir la cantidad de bytes intercambiados y acelerar, así, las comunicaciones, mientras que los cálculos mantienen la precisión original. Esta mejora podría integrarse en PyDTNN creando una versión de la comunicación que integre las transformaciones mencionadas. Teniendo en cuenta los esquemas de paralelización mencionados en el Capítulo 2, su implementación requería de comunicaciones colectivas. En concreto, y tal como se retomará más adelante, el paralelismo de datos requiere de la primitiva *AllReduce*, la cual puede constituir un coste significativo durante el proceso de entrenamiento. Una opción de acelerar su ejecución, sería implementando un algoritmo *AllReduce* personalizado tal y como se ejemplifica en el Código 3.6. En él se observa la implementación del algoritmo RING, un algoritmo que realiza la reducción y distribución de los datos mediante envíos que siguen una topología de anillo, realizando transmisiones encadenadas. Este algoritmo se detalla más profundamente en futuras secciones (Sección 4.3 y Apéndice A).

```
def MPI_Allreduce_RNG(sbuf,comm_dtype,op_reduce,comm):
  rank, comm_size = comm.Get_rank(), comm.Get_size() # Datos de los procesos
  bsize = sbuf.size / comm_size # Tam. del bloque local
  comp_dtype = sbuf.dtype # Tipo de datos original.

  # Datos de las comunicaciones entre procesos
  send_to = (rank + 1) % comm_size
  recv_from = (rank - 1) % comm_size
  inbuf = [ np.empty((bsize), dtype=comm_dtype), np.empty((bsize), dtype=comm_dtype) ]
  req = [None, None]
  inbi = True
  ini, end = block_offset(rank, sbuf.size, 1, bsize)

  # Recepcion de un bloque del mensaje
  req[inbi] = comm.Irecv([inbuf[inbi], inbuf[inbi].size*inbuf[inbi].itemsize,  MPI.CHAR],
                         source=recv_from, tag=RED)

  # Conversion del bloque al nuevo tipo de datos
  sbuf_ = sbuf[ini:end].astype(comm_dtype)

  #Envio del bloque del mensaje
  comm.Send([sbuf_, sbuf_.size*sbuf_.itemsize, MPI.CHAR], dest=send_to, tag=RED)

  # Recepcion, reduccion, conversion y envio del resto de bloques
  for k in range(2, comm_size):
    inbi = not inbi
    # Post irecv for the current block
    req[inbi] = comm.Irecv([inbuf[inbi], inbuf[inbi].size*inbuf[inbi].itemsize, MPI.CHAR],
                           source=recv_from, tag=RED)
    req[not inbi].Wait()
    ini, end = block_offset(rank, sbuf.size, k, bsize)
    sbuf[ini:end]= op_reduce(sbuf[ini:end], inbuf[not inbi].astype(comm_dtype))
    sbuf_ = sbuf[ini:end].astype(comm_dtype)
    comm.Send([sbuf_, sbuf_.size*sbuf_.itemsize, MPI.CHAR], dest=send_to, tag=RED)
```

```
35  req[inbi].Wait()
36  ini, end = block_offset(rank, sbuf.size, comm_size, bsize)
37  sbuf[ini:end] = op_reduce(sbuf[ini:end], inbuf[inbi].astype(comp_dtype))
38
39  # ... Distribucion analoga
```

Código 3.6: Versión del algoritmo RING para la implementación de la primitiva *AllReduce* utilizando precisión mixta.

En este código, tal y como se ha mencionado antes, los datos son transformados previamente a su envío a un tipo de datos de menor precisión (Líneas 19 y 33); y para la realización de la reducción vuelven a ser reconvertidos al tipo de datos original (Líneas 32 y 37), de modo que no se pierde precisión en la componente aritmética de la propia reducción.

3.4.2. Reducción del coste aritmético utilizando poda

Una técnica que posibilita la aceleración de los procesos de entrenamiento e inferencia de las redes neuronales consiste en eliminar las conexiones innecesarias o menos relevantes entre las neuronas. Esta técnica, conocida como poda, suprime aquellos parámetros –pesos y sesgos– de la red que, debido a su bajo valor, en principio aportan poco valor a los cálculos. De este modo, al reducir el número de parámetros que participan en la red, se reduce también el coste computacional; y al haberse excluido únicamente los que proporcionaban escasa contribución a los resultados, la precisión de la red no debería verse significativamente afectada. Sin embargo, es cierto que el efecto sobre los resultados vendrá determinado por la cantidad de conexiones eliminadas: a mayor poda, la red será más ligera pero menos precisa. Un modo habitual de usar esta técnica se basa en su aplicación repetitiva junto al reentrenamiento de la misma red, de tal forma que en cada iteración se eliminan ciertas conexiones para centrar el foco en las restantes, las cuales se vuelven a reentrenar durante varias épocas para recuperar la precisión perdida. Estas repeticiones permitirán reducir el número de parámetros de la red hasta el volumen deseado, afectando de manera más moderada a la capacidad de aprendizaje de la red.

En el entorno PyDTNN se puede implementar esta modalidad de poda, utilizando cada ciertas épocas máscaras binarias que, al multiplicarlas punto a punto por los pesos actualizados, descarten aquellos pesos cuyo valor se considere bajo.

3.4.3. Operadores alternativos más eficientes

Otra extensión que permite reducir el coste computacional es diseñar operadores alternativos. En las redes del tipo convolucional, la aplicación clásica de las convoluciones supone una parte significativa del coste computacional debido al alto coste de cómputo derivado de las operaciones de convolución. Un método alternativo para aplicar la convolución, que puede ofrecer un mayor rendimiento, consiste en realizar esta mediante una multiplicación matriz-matriz. Este método, que se explicó en la Sección 2.2 consiste en aplicar, previamente a la operación, la transformación *im2col* (o *im2row*) sobre el tensor de activaciones, junto con una reorganización de los pesos de la capa, para realizar posteriormente el producto de ambos elementos.

Cambiar el operando clásico de convolución por esta alternativa puede implementarse fácilmente en PyDTNN como se muestra en el Código 3.7, en el cual, a fin de tener una mejor eficiencia, se utiliza un método externo implementado en Cython para obtener la matriz de activaciones resultante de la transformación *im2col* (Línea 2) [8]. Como se observa en la línea 5, además de transformar las activaciones, también es necesario reorganizar los pesos previamente a calcular el producto de ambos operadores (Línea 6).

```
1 def forward(self, x):
2     # ...
3     self.x_cols = im2col_cython(x, self.kh, self.kw,
4                     self.vpadding, self.hpadding, self.vstride, self.hstride)
5     w_cols = self.weights.reshape(self.co, -1)
6     res = self.matmul(w_cols, x_cols)
7     # ...
```

Código 3.7: Fragmento del código para implementar el paso *forward* de las convoluciones utilizando *im2col*.

Este método, por contra, puede agotar la memoria del sistema debido al gran tamaño de la matriz de activaciones que se construye al aplicar esta transformación, pues la matriz resultante puede llegar a ocupar $k_h \times k_w$ veces su tamaño original. Una alternativa para evitar parte de ese incremento del consumo de memoria, consiste en su aplicación por secciones, dividiendo el trabajo a realizar por conjuntos de muestras. El Código 3.8 muestra la implementación para conjuntos de `chunk_size` muestras, de modo que se reduce el tamaño del problema

`batch_size`/`chunk_size` veces respecto a la implementación previa.

```
def forward(self, x):
    # ...
    w_cols = self.weights.reshape(self.co, -1)
    y_cols = np.empty(self.co, self.ho*self.wo*self.batch_size)
    for s in range(0, self.batch_size, self.chunk_size):
        e = min(s+self.chunk_size, self.batch_size)
        self.x_cols = im2col_cython(x[s:e,...], self.kh, self.kw,
                    self.vpadding, self.hpadding, self.vstride, self.hstride)
        s_, e_ = s*self.ho*self.wo, e*self.ho*self.wo
        y_cols[:,s_:e_] = self.matmul(w_cols, x_cols)
    # ...
```

Código 3.8: Fragmento del código para implementar el paso *forward* de las convoluciones utilizando *im2col* por secciones.

3.5. Paralelismo en PyDTNN

En la Sección 2.5 se comentaron las principales paralelizaciones del entrenamiento de las redes neuronales: paralelismo de datos, de modelo y *pipelining*. En el entorno PyDTNN se deseó realizar una paralelización del entrenamiento que permitiera acelerar su ejecución. Dado que la pretensión de este entorno es implementar nuevas ideas, se consideró que el modelo de paralelización debía mantener el mismo comportamiento numérico que la versión secuencial del entorno, para poder realizar comparaciones. Esta elección, junto con el potencial de escalabilidad de las estrategias de paralelismo descritas anteriormente, fueron las razones por las que se decidió integrar el modelo de paralelismo de datos, tal y como se detallará durante este capítulo, y cuya aplicación se apoya en el paquete `mpi4py` de Python como capa de comunicación entre nodos y en la biblioteca NCCL de NVIDIA para clústeres equipados con GPU [56].

3.5.1. Implementación

En el paralelismo de datos el modelo de la red neuronal, es decir, los pesos y sesgos que la definen, se replican en todos los procesos participantes, mientras que las muestras del lote se distribuyen entre estos. Para la primera parte de esta tarea, replicar el modelo, basta con establecer la misma semilla en todos los nodos antes de iniciar el entrenamiento, de modo que la generación inicial de los pesos y sesgos sea idéntica en todos ellos. En cuanto a la segunda parte de

este primer paso, la distribución de las muestras, en la implementación esta se ha programado de modo que el usuario indica el tamaño del lote que entrenará cada proceso, es decir, el número de muestras del lote local. De este modo, el lote global se considera como la unión de los locales y, por tanto, su tamaño vendrá determinado por el producto del tamaño de los lotes locales y el número de procesos que intervienen. En el Código 3.9 puede observarse cómo se realiza la distribución de acuerdo a lo explicado: en primer lugar se calcula cuál es el tamaño del lote global, y después se asigna a cada proceso la sección de dicho lote que le corresponde en función de su identificador.

```
1  def batch_generator(generator, local_batch_size=64, rank=0, nprocs=1, shuffle=True):
2    #Calcular tam. lote global
3    batch_size = local_batch_size * nprocs
4
5    for X_data, Y_data in BackgroundGenerator(generator):
6      # Generar indices
7      nsamples = X_data.shape[0]
8      if shuffle:
9        s = memoryview(np.arange(nspamples))
10     np.random.shuffle(s)
11     # ...
12     # Generar de lotes
13     for batch_num in range(0, end_for, batch_size):
14       # Seleccionar indices
15       start = batch_num +  rank    * local_batch_size # inicio
16       end   = batch_num + (rank+1) * local_batch_size # fin
17       indices = s[start:end]
18       # Seleccionar muestras
19       X_local_batch = X_data[indices,...]
20       Y_local_batch = Y_data[indices,...]
21       yield (X_local_batch, Y_local_batch, batch_size)
22   # ...
```

Código 3.9: Esquema del método `batch_generator` para generar los lotes de entrenamiento.

Otra alternativa que se consideró consistía en que el usuario pudiera introducir el tamaño del lote global; sin embargo, este modo puede producir divisiones de trabajo desiguales entre los procesos (por ejemplo, 10 muestras y 3 procesos). Además, tal y como se mencionó en la Sección 2.5.1, una de las características positivas del paralelismo de datos es su capacidad de incrementar la carga de trabajo local para mejorar el rendimiento; teniendo esto en cuenta, es más conveniente indicar la carga local que se desea que tenga cada proceso, pues ofrece una visión más fidedigna del trabajo asignado a cada proceso.

Una vez ha sido inicializado el modelo en cada nodo, queda adaptar el entrenamiento a las necesidades que derivan de la distribución. Para ello, como se

ha explicado en la Sección 2.5.1, basta con que los procesos se comuniquen entre sí los gradientes locales de modo que todos terminen contando con el gradiente global necesario para actualizar los pesos. De este modo, es posible mantener su coherencia y la coordinación del propio entrenamiento. La versión paralela más elemental del proceso de entrenamiento puede observarse en el Código 3.4, e incluye en la línea 14 la comunicación *AllReduce* necesaria.

Cabe hacer constar que el grafo de dependencias del proceso de entrenamiento permite modificar el orden de ejecución de algunas instrucciones sin afectar a los resultados de los cálculos. Esto posibilita solapar las comunicaciones con tareas no dependientes del dato recibido y, de este modo, acelerar más el entrenamiento. La clave para este solapamiento radica en reemplazar el uso de las comunicaciones síncronas `MPI_Allreduce()`, por su versión asíncrona `MPI_Iallreduce()`, junto con las llamadas de sincronización `MPI_Wait()` necesarias. Para la implementación de este entrenamiento con comunicaciones asíncronas se contemplaron tres versiones, cada una de las cuales se establece teniendo en cuenta las siguientes modificaciones:

- **Versión A)** Realizar las actualizaciones de pesos tras completar el producto de las activaciones y los gradientes locales de todas capas, necesario para la actualización, de tal modo que las comunicaciones se ejecuten simultáneamente con estas operaciones: mientras se completa la comunicación *AllReduce* de la capa l, se pueden ir calculando el resto de gradientes locales $G^{(i)}$ y sus productos $G^{(i)} \cdot A^{(i-1)}, \forall i < l$. Véase la imagen A de la Figura 3.2.

- **Versión B)** Adicionalmente a la superposición de las comunicaciones con los cálculos de los gradientes mencionados en la versión A, esta modificación elimina la dependencia entre el orden de ejecución de las comunicaciones de capas contiguas y el orden de su finalización mediante el uso de un comunicador por capa. De este modo, la dependencia pasa a estar solo entre la finalización de la comunicación de cierta capa y la actualización de sus pesos, la cual se realiza durante el siguiente lote previamente al cálculo de las activaciones. Véase la imagen B de la Figura 3.2, dónde los subíndices referencian el lote sobre el que se efectúa dicha tarea.

- **Versión C)** Conjuntamente con la superposición de lotes mencionada en la versión B, en este caso se realizan los productos $G^{(l)} \cdot A^{(l-1)}$ y las llamadas de las comunicaciones en el orden en que se requieren posteriormente los resultados de estas, es decir, desde la primera hasta la última capa, como se observa en la imagen C de la Figura 3.2. Esta versión permite eliminar el uso de múltiples comunicadores de la versión previa al llamar a las comunicaciones en el mismo orden en el que se precisan.

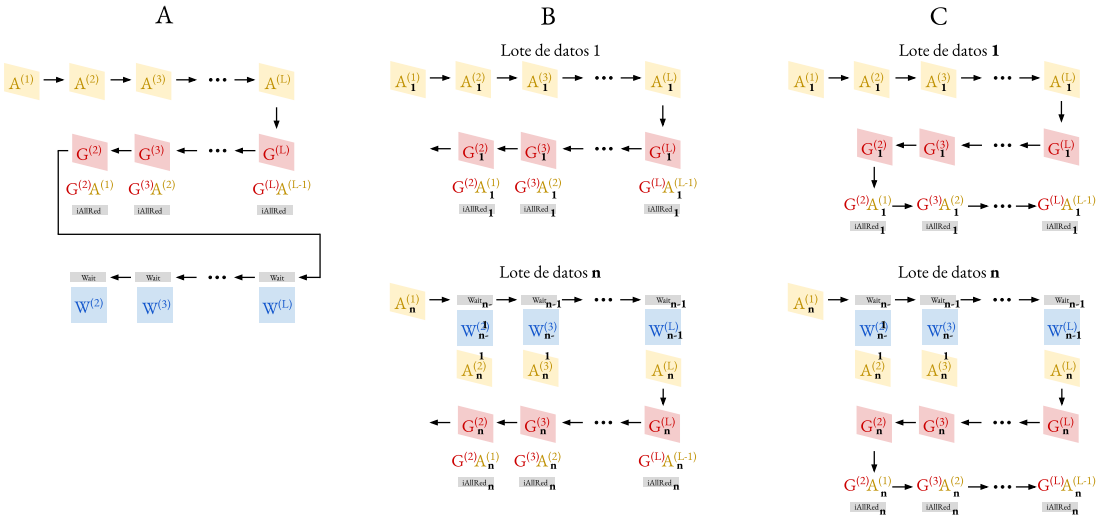

Figura 3.2: Versiones del paralelismo de datos con solapamiento de comunicaciones y cómputo.

En la primera versión todas las capas deben completar el proceso *forward-backward* del lote actual antes de iniciar el siguiente, mientras que en las versiones restantes se superponen tareas de diferentes lotes. Pese a que esto supone una limitación en la capacidad de solapar diferentes etapas para la versión A, se decidió implementar esta primera en PyDTNN por mantener la sencillez del entorno y su característica amigable. En comparación, la adaptación necesaria para combinar tareas de diferentes lotes con la estructura actual de PyDTNN supondría añadir una gran complejidad a este.

En el Código 3.10 se muestra la implementación realizada en PyDTNN del entrenamiento con comunicaciones asíncronas (versión A). Con respecto a su versión síncrona (Código 3.4), se diferencia un nuevo método, `wait_allreduce_async`

(Línea 15), previo a la actualización de pesos, `update_weights`. Este método invoca internamente a la función `MPI_Wait()` para forzar la finalización de la comunicación `MPI_Iallreduce()` iniciada por el método de suma de pesos, el cual ahora se denomina `reduce_weights_async` (Línea 11), y mantiene, así, la sincronización necesaria entre los procesos para que el entrenamiento sea análogo a sus versiones secuencial y síncrona. Este método, del mismo modo que otros métodos relacionados con los pesos como `reduce_weights_sync`, `reduce_weights_async` y `update_weights`, pertenece a la clase `Layer` y es ejecutado por todas las capas, tengan o no pesos, pese a que solo aquellas con pesos realizarán una sincronización en la invocación de este método.

```
1  def __train_batch(self, x_lote, y_lote, ...):
2    # Paso Forward
3    x = x_lote
4    for l in range(1, len(self.layers)):
5      x = self.layers[l].forward(x)
6
7    # Paso Backward - Calculo del gradiente
8    loss, dx = loss_func(x, y_lote)
9    for l in range(len(self.layers)-1, 0, -1):
10     dx = self.layers[l].backward(dx)
11     self.layers[l].reduce_weights_async()
12
13   # Paso Backward - Actualizacion de pesos
14   for l in range(len(self.layers)-1, 0, -1):
15     self.layers[l].wait_allreduce_async(self.comm)
16     self.layers[l].update_weights(optimizer)
```

Código 3.10: Método `train_batch` de la clase `Model`.

Antes de finalizar esta sección, cabe destacar que la paralelización que se ha realizado respeta, de cara al usuario, el principio de entorno amigable. Así, para ejecutar un entrenamiento paralelizado según el paralelismo de datos equivale a invocar `mpirun`, tal y como se muestra en el siguiente comando:

```
mpirun -np 4 python benchmark.py --model resnet --dataset mnist
```

donde el usuario sólo debe indicar la cantidad de procesos que participarán, el código de referencia `benchmark`, e indicarle a este los parámetros con los que se desea trabajar. En este caso se ha especificado únicamente el modelo y el conjunto de datos, pero pueden incluirse otros parámetros como el ratio de aprendizaje, el tamaño del lote, si se desea realizar sólo inferencia, etc.

3.5.2. Evaluación

En esta sección se muestra el rendimiento y la precisión que ofrece PyDTNN en el entrenamiento de redes, cuando se ejecuta en múltiples nodos (paralelismo iter-nodo), así como explotando el paralelismo intra-nodo que se obtiene del uso de rutinas multihilo externas como Cython a través de OpenMP [58].

Dado que el comportamiento del paralelismo implementado es semejante al funcionamiento de TensorFlow combinado con Horovod (HVD), en esta evaluación se comparan, a modo de referencia, los resultados de PyDTNN con los mismos casos de estudio sobre TF+HVD [1, 64]. El entrenamiento distribuido con esta última combinación crea un hilo auxiliar para llevar a cabo la comunicación colectiva *AllReduce* simultáneamente con el cálculo de los gradientes de las capas previas que realiza el hilo principal. Este funcionamiento es análogo al comportamiento asíncrono resultante de emplear la versión `Iallreduce()` que se ha aplicado en PyDTNN y mantiene igualmente un entrenamiento con comunicaciones asíncronas restringido a un único paso *forward-backward*.

Antes de analizar los datos, cabe destacar que el propósito de crear este entorno era el de conseguir un marco de trabajo en el que implementar fácilmente nuevos prototipos de ideas, y que, por tanto, no se buscaba competir con el rendimiento que presentan populares entornos de entrenamiento actuales como el mencionado TensorFlow, PyTorch, Keras, etc. Por ello, el objetivo de esta comparativa es únicamente ilustrar el desarrollo alcanzado por PyDTNN en relación a TensorFlow.

Para la realización de los experimentos, las características de la plataforma utilizada son las siguientes: 9 nodos con 2 CPU Intel Xeon Gold 5120 de 14 núcleos cada uno (28 núcleos por nodo) y 187 GB de RAM DDR4; una GPU NVIDIA Tesla V100-PCIe con 32 GB de memoria HBM2; y una red de conexión Infiniband EDR de 100 Gbps. Respecto al software, las versiones utilizadas han sido: TF 2.1.0, HVD 0.20.3, cuDNN (NVIDIA) 7.6.5, OpenMPI 4.0.3 y NVIDIA NCCL 2.7.8. Finalmente, en cuanto a las redes y su configuración, se ha trabajado con las características referenciadas en TF para cada modelo [37].

En esta evaluación se ha estudiado el comportamiento del entrenamiento sobre diversos conjuntos de datos y modelos de redes neuronales popularmente cono-

cidos, con diferentes características y particularidades, de modo que el estudio proporcione un análisis amplio y representativo, y por consiguiente, se logre una evaluación global y completa de PyDTNN. Asimismo, también se han considerado diferentes bibliotecas de comunicación a la hora de ejecutar las pruebas. En concreto, se han extraído resultados de los modelos de redes neuronales AlexNet, VGG16 y ResNet-101, para los conjuntos de datos CIFAR-10 e ImageNet [45]. Además, se han utilizado las dos bibliotecas de comunicación de las que se disponía en las GPU de la plataforma utilizada para estos experimentos: OpenMPI y NCCL [29].

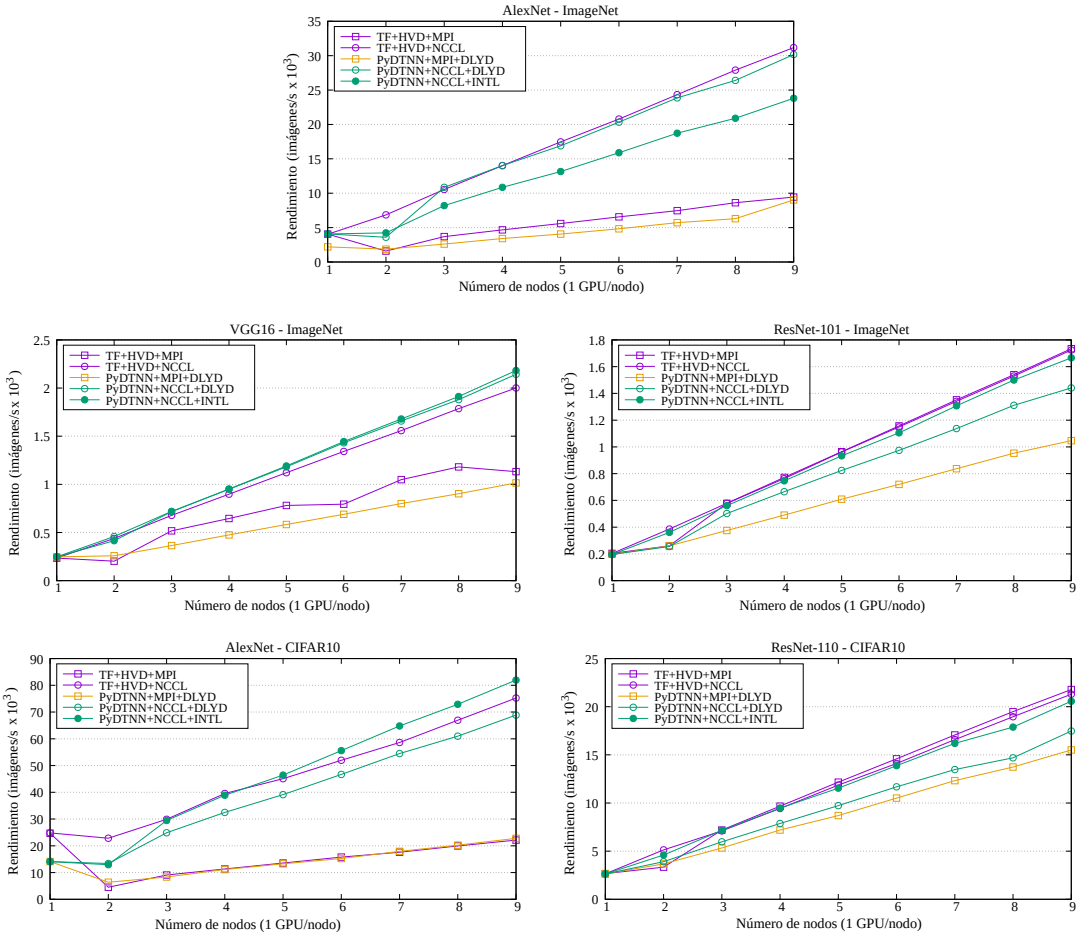

Figura 3.3: Rendimiento de los entornos PyDTNN y TF+HVD en función del número de nodos/GPU para diferentes redes

Los datos recogidos en las gráficas de la Figura 3.3 permiten observar la capacidad de escalado conseguida en diferentes escenarios al analizar el rendimiento medido en imágenes por segundo en función del número de nodos que intervienen. En cada una de ellas se observa los resultados obtenidos tanto en TF+HVD como en PyDTNN, utilizando las dos bibliotecas de comunicación mencionadas. Además, en el caso de PyDTNN se diferencia el entrenamiento con el uso de comunicaciones síncronas (denotadas como DLYD –*delayed*– y que se corresponden con la versión de entrenamiento que utiliza el Código 3.4) y asíncronas (denotadas INTL –*interleaved*– y correspondientes a la versión de entrenamiento del Código 3.10). Nótese que no se recogen datos para la combinación PyDTNN+MPI+INTL. Esto se debe a que en OpenMPI la versión asíncrona de la primitiva *AllReduce* no es compatible con CUDA.

Pueden advertirse un par de observaciones generales en un primer análisis de estas gráficas, referentes a la escalabilidad que ambos entornos ofrecen al incrementar los nodos participantes en el entrenamiento y la diferencia entre el rendimiento ofrecido según la biblioteca de comunicaciones empleada. Examinando más detenidamente estos factores pueden argumentarse dichos comportamientos.

Con respecto a la capacidad de escalado observada en ambos entornos de entrenamiento, es notable la linealidad de esta, que únicamente se ve alterada en los casos en los que participan menos de tres nodos. La justificación de esta irregularidad en el rendimiento se debe al sobrecoste derivado de las comunicaciones que se generan al pasar del modo secuencial con un único nodo al entrenamiento paralelo con dos nodos. A partir de ese punto, y conforme a lo mencionado en capítulos previos sobre la escalabilidad del paralelismo de datos, el rendimiento aumenta linealmente a medida que se incrementa la cantidad de nodos participantes.

Por otro lado, en cuanto a la disparidad entre las bibliotecas de comunicación, se aprecia en las gráficas una frecuente ventaja en favor de NCCL (líneas representadas con circunferencias) sobre MPI (líneas representadas con cuadrados). Esta predominancia se debe a las transferencias que se realizan entre el dispositivo y el *host* cuando la comunicación se lleva a cabo utilizando la biblioteca MPI, las cuales no ocurren en el caso de emplear NCCL. De hecho, puede observarse que los casos en los que esta disparidad no es notoria son los que involucran a la red ResNet-101, debido a que la carga que suponen las comunicaciones en el

entrenamiento de esta es una proporción ínfima frente a la importancia del coste aritmético, lo que hace que el coste de dichas transferencias no afecte significativamente al rendimiento global. Teniendo esto en consideración, en los siguientes análisis se examinan los resultados de PyDTNN frente a los de TF+HVD en función de la biblioteca utilizada; sin comparar los datos de las bibliotecas de comunicación entre sí.

En cuanto a los resultados obtenidos utilizando la biblioteca NCCL (simbolizada mediante circunferencias), el rendimiento que puede alcanzar PyDTNN (líneas de trazo verde) es similar al de TF+HVD (líneas de trazo morado), siendo incluso superior en algunos casos. Además, nótese que, en función de la red analizada, puede ser más competitivo el modo de entrenamiento DLYD, es decir, sin solapamiento de comunicación y cómputo, que el modo INTL, mediante comunicaciones asíncronas. Resulta interesante, de hecho, que esta ganancia ocurra en aquellas redes limitadas por las comunicaciones como es el caso de AlexNet; por el contrario, el rendimiento de INTL resulta mayor en las redes con mayor cómputo, como ResNet.

Por otro lado, respecto a los datos de la biblioteca OpenMPI (simbolizada mediante cuadrados), dado que PyDTNN sólo se ha ejecutado en el modo DLYD, TF+HVD ofrece en esta ocasión unas prestaciones más competitivas. Recordar que TF+HVD cuenta con el beneficio de la comunicación asíncrona proporcionada por el hilo de HVD. Esta ventaja, además, resulta superior en los casos limitados por cómputo. Esto coincide con lo mencionado en la sección anterior donde, en tales circunstancias, el modo DLYD no tenía tanta ganancia como el modo INTL.

En virtud de estos resultados, puede concluirse que el rendimiento de PyDTNN, pese a que con OpenMPI ofrece unas prestaciones ligeramente inferiores a las de TF+HVD, resulta competitivo con respecto a TF+HVD cuando se utiliza la biblioteca NCCL. Por este motivo, PyDTNN puede ser un entorno para el entrenamiento de las redes tan válido como TF+HVD en cuanto a rendimiento, presentando, además, las facilidades de modificación y ampliación que se han ilustrado en el presente capítulo.

Otro aspecto del entrenamiento de las redes neuronales cuyo análisis resulta fundamental para poder valorar la calidad de un entorno, es su convergencia y

validación, es decir, qué precisión es capaz de alcanzar en ambos casos y a qué velocidad. En la Figura 3.4 pueden observarse los datos obtenidos del entrenamiento y la validación de los modelos AlexNet y ResNet-110 con el conjunto de datos CIFAR 10 mediante el uso de ambos entornos, TF+HVD (morado) y PyDTNN (verde).

Para la obtención de estos resultados se ha realizado el entrenamiento utilizando 9 nodos con GPU, durante un número fijo de épocas establecido a 120 y utilizando los mismos valores para los parámetros de entrenamiento: tamaño de los lotes de 576 imágenes, lo que equivale a procesar 64 por nodo; optimizador SGD con un momentum de 0.9, ratio de caída de peso $1,0e-4$, y un ratio de aprendizaje de 0.01 escalado al número de nodos, es decir, 0,09 ($0,01 \times 9 = 0,09$); además, este ratio de aprendizaje se incrementa durante las cinco primeras épocas y, después, decae cada cuarenta épocas en un factor $10\times$.

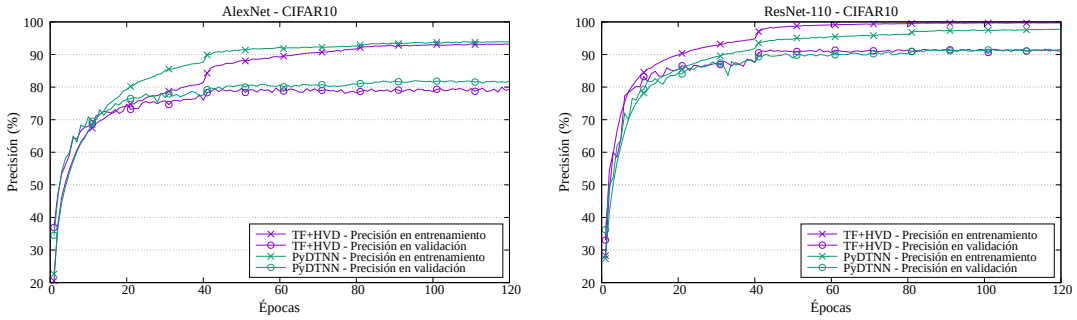

Figura 3.4: Convergencia del entrenamiento y la validación de diferentes redes utilizando TF+HVD y PyDTNN.

Como puede observarse en dicha figura, ambas redes consiguen un incremento logarítmico en la precisión, tanto para el entrenamiento como para la validación, logrando un crecimiento pronunciado durante las primeras épocas para, después, estabilizarse en una precisión máxima alcanzada. Por otro lado, los datos muestran una convergencia similar para ambos entornos. En el caso de AlexNet, la precisión alcanzada en la última época estudiada por PyDTNN es ligeramente mejor a la de TF+HVD, tanto en entrenamiento –95,30 % vs 94,82 %–, como en validación –82,46 % vs 78,42 %–. En ResNet, sin embargo, TF+HVD presenta una modesta ventaja frente a PyDTNN en el entrenamiento –99,81 % vs 98,46 %–; no

obstante, en la validación ambos entornos son igual de precisos –91,71 %–. Asimismo, ambos entornos muestran velocidades de convergencia bastante similares. Por tanto, la calidad del entrenamiento de PyDTNN en cuanto a precisión, puede considerarse similar y, en consecuencia, también comparable a la del entorno TF+HVD.

3.6. Conclusiones

A lo largo de este capítulo se ha presentado la herramienta PyDTNN desarrollada. La creación de este recurso surgió como un modo de poner a prueba la comprensión sobre el entrenamiento de las redes neuronales y mejorar el conocimiento acerca de los detalles que este abarca. Con esta premisa presente, PyDTNN se plantea como una alternativa a los populares entornos de aprendizaje profundo como TensorFlow o PyTorch, presentando una interfaz simple y accesible y una estructura interna cuya versatilidad facilita su modificación para la inclusión de nuevas técnicas o la modificación de las ya existentes. Esta característica se ha demostrado con múltiples ejemplos a lo largo del capítulo, algunos de los cuales ya se encuentran implementados en el entorno, como es el caso del paralelismo de datos. Esta practicidad de cara al usuario se consideró prioritaria frente a la posibilidad de que ello limitara el rendimiento alcanzable en el entrenamiento. No obstante, se ha mostrado experimentalmente que, tanto la precisión, como el rendimiento logrados por PyDTNN, aunque en algunos escenarios no son capaces de alcanzar los ofrecidos por entornos sofisticados como TensorFlow, se acercan a estos considerablemente pudiendo en ocasiones competir con ellos.

Capítulo 4

Modelado del Entrenamiento

En este capítulo se presenta el desarrollo de dos modelos que permiten aproximar el coste temporal que conlleva entrenar una red neuronal y su validación por medio de PyDTNN [7, 16]. Estos modelos tienen en cuenta las condiciones de entrenamiento cuando se explota el paralelismo de datos con unas características concretas de hardware. Asimismo, gracias a este modelos, se expone un estudio en el que se analiza la capacidad de escalado así como las limitaciones que se encuentran en este esquema de paralelización para proporcionar una noción al usuario que le ayude a anticipar qué opciones de entrenamiento pueden resultar más adecuadas.

4.1. Motivación

El coste temporal de entrenar una red neuronal depende de múltiples factores. Así, entre otros, está ligado a las características de la red, el volumen de datos que se procesarán durante el entrenamiento y los recursos de los que se disponga para llevarlo a cabo. El progreso de las redes neuronales y el permanente aumento de datos disponibles para su entrenamiento, llevan implícito un agravamiento del coste de este. En consecuencia, aparece el requisito de disponer de plataformas de gran escala que contrarresten el alto coste producido por la gran carga de trabajo [9, 61, 66]. Así pues, tiene un valor relevante ser capaz de conocer con antelación este coste en función de los recursos que se van a invertir, de modo que puedan ser utilizados con la mayor eficiencia posible, siendo conscientes de

los cuellos de botella que pueden aparecer y de qué modo es más conveniente afrontar los mismos.

Con este fin surge el uso de modelos: esquemas teóricos que generan de forma matemática predicciones del coste del entrenamiento en función de ciertos parámetros; y la oportunidad de realizar, en base a estos pronósticos, un análisis previo al despliegue que sirva como fundamento para decidir cómo destinar nuestros recursos.

Como se ha mencionado en el Capítulo 3, la estrategia de paralelismo de datos como método de aceleración del entrenamiento ofrece una alta escalabilidad en comparación a otros esquemas, sin introducir, en sí misma, una complejidad notable de implementación. Es por ello que su uso está muy extendido y, por tanto, resulta relevante estudiar y analizar su modelado.

En base a lo anterior, se consideró la relevancia de diseñar un modelo que proporcionara una estimación del tiempo de entrenamiento de una red neuronal que explota el paralelismo de datos, y para su elaboración se tomó como entorno de trabajo, la herramienta PyDTNN.

La construcción de los modelos se ha realizado mediante la descomposición del problema inicial –estimar el coste de entrenamiento– en subproblemas más simples. En concreto, nuestro modelo persigue estimar el coste de un paso *forward-backward*, seleccionando para ello únicamente aquellas tareas que aportan un coste significativo. Estas se han diferenciado en tareas cuyo coste se deriva del cálculo computacional frente a aquellas en las que predomina el coste de comunicación. Teniendo en cuenta esto y observando las tareas a ejecutar definidas en el paralelismo de datos, los subproblemas que se abordarán para aproximar su coste serán, por un lado, el cómputo local involucrado en los pasos *forward* y *backward* de cada capa y, por otro lado, se evaluará el coste de la comunicación *AllReduce* necesaria para conseguir la actualización global de los pesos.

Para empezar a modelar el coste computacional y de comunicaciones, en la Tabla 4.1 se muestra una serie de parámetros que serán utilizados para estimar dichos costes. Como se ha mencionado previamente, los parámetros que influyen en el tiempo de entrenamiento son aquellos derivados de las características de la red neuronal, las opciones de entrenamiento escogidas, y los recursos de la plataforma que se van a utilizar para realizar el entrenamiento. Las variables

correspondientes a cada uno de estos aspectos se encuentran en dicha tabla, junto con la nomenclatura que se utilizará para ellas en adelante, agrupadas por secciones según los aspectos a los que corresponden.

Tabla 4.1: Parámetros del modelo.

Notación	Descripción
L	Número de capas de la red neuronal
$n^{(l)}$	Número de neuronas de la capa l (FC)
$h^{(l)}, w^{(l)}, k^{(l)}$	Alto, ancho y núm. de canales de la capa l (Conv o *pooling*)
$h_W^{(l)}, w_W^{(l)}$	Alto y ancho de los filtros de la capa l (Conv o *pooling*)
$s_h^{(l)}, s_w^{(l)}$	*Stride* vertical y horizontal de la capa l (Conv o *pooling*)
$p_h^{(l)}, p_w^{(l)}$	*Padding* vertical y horizontal de la capa l (Conv o *pooling*)
b	Tamaño del lote
λ	Bytes por número en coma flotante
P	Número de procesos en el clúster
γ	Rendimiento teórico en coma flotante (en seg/flop)
μ	Ancho de banda de la memoria en (seg/byte)
α	Latencia de enlace (en seg)
β	Ancho de banda de enlace (en bytes/seg)

4.2. Coste del Cómputo

En esta sección se presentan dos estrategias de modelado que predicen el coste temporal del cómputo mediante diferentes metodologías:

- Un primer modelo analítico, en la Sección 4.2.1, basado en la estimación mediante fórmulas teóricas de las tareas realizadas durante el entrenamiento, en el que se analizan el número de operaciones y accesos a memoria realizados.
- Un modelo alternativo, en la Sección 4.2.2, respaldado parcialmente por datos experimentales, cuya estimación se fundamenta en las predicciones

realizadas por un conjunto de redes neuronales diseñadas y entrenadas expresamente para predecir los costes de los núcleos o *kernels* utilizados durante el entrenamiento.

Cada una de estas opciones de modelado se describe a continuación, junto con las ventajas e inconvenientes que conllevan, así como los escenarios en los cuales resulta más apropiado el uso de cada una.

4.2.1. Modelo analítico

En este modelo se estima el coste computacional considerando los parámetros del computador en el que se realizará el entrenamiento y la velocidad de este para ejecutar las operaciones involucradas durante los pasos *forward* y *backward*. Así, teniendo en cuenta el cómputo a realizar y la velocidad a la que este se realiza, puede calcularse el tiempo que tardará en efectuarse el entrenamiento.

Como primer paso, por tanto, debe precisarse qué operaciones se contabilizarán. Como se ha mencionado previamente, la estimación del coste temporal se aproximará considerando únicamente las tareas que aportan un coste significativo. Por consiguiente, en este modelo se predecirá el coste de entrenamiento de una red en base al coste de las capas cuya complejidad sea mayor y, por ende, su peso en la duración global sea más significativo. Se considerarán, por tanto, aquellas capas con mayor carga de trabajo –FC, convolucionales y *pooling*–, obviando aquellas cuyo coste sea, en comparación con las primeras, relativamente despreciable –*Flatten*, *Dropout*, etc–. Análogamente, de las operaciones asociadas a estas capas solo se incluirán en el modelo aquellas cuya carga sea más relevante, considerándose como tales los productos matriz-matriz (GEMM) y las transformaciones *im2col*.

Teniendo en cuenta que las capas *pooling* no disponen de pesos y, por tanto, sus cálculos durante el entrenamiento no incluyen productos matriz-matriz, en este tipo de capas solo se considerará el coste de la transformación *im2col* necesaria durante el paso *forward*, cuyo coste se calcula únicamente en operaciones de acceso a memoria (o *memops*). Para ello se suma la cantidad de elementos del operando resultante, que en este tipo de capas equivale a

$$k^{(l)} h^{(l)} w^{(l)} \lceil b/P \rceil \times h_W^{(l)} w_W^{(l)} \quad memops. \tag{4.1}$$

Las capas convolucionales, además de esta transformación inicial cuyo coste es

$$k^{(l)}h^{(l)}w^{(l)}\lceil b/P\rceil \times k^{(l-1)}h_W^{(l)}w_W^{(l)} \quad memops, \tag{4.2}$$

requieren posteriormente de las mismas tres operaciones de tipo producto matricial que las capas FC durante los pasos *forward* y *backward*, las cuales, recordemos, son:

$$\begin{cases} \textit{Forward} \text{ (FP)} & A^{(l)} = \mathbf{W^{(l)}} \cdot \mathbf{A^{(l-1)}} + B^{(l)}, \\ \textit{Backward} \text{ - Calcular Gradientes (BP-CG)} & G^{(l-1)} = \mathbf{W^{(l)T}} \cdot \mathbf{G^{(l)}}, \\ \textit{Backward} \text{ - Actualizar Pesos (BP-AP)} & W^{(l)} = W^{(l)} - \mathbf{G^{(l)}} \cdot \mathbf{A^{(l-1)T}}; \end{cases}$$

incluyendo cada una de ellas una operación GEMM.

El coste de cada uno de estos productos en operaciones (*flops*) y accesos a memoria (*memops*) se calcula del mismo modo para ambos tipos de capas: sean $A \in \mathbb{R}^{m\times p}$ y $B \in \mathbb{R}^{p\times n}$ los operandos del producto, cada elemento de la matriz resultante $C = A \cdot B \in \mathbb{R}^{m\times n}$, se obtiene como $C_{i,j} = \sum_{x=0}^{p} A_{i,x} \cdot B_{x,j}$; requiriendo, por tanto, $2p$ operaciones por elemento y un total de $2pmn$ para la matriz completa. Por otra parte, para el número de accesos a memoria, suponiendo por simplicidad el mismo costo para aquellos de lectura y escritura, y asumiendo que solo es preciso acceder a cada elemento una vez, se obtiene un total de $mp + pn + mn$ accesos, uno por cada elemento de las tres matrices implicadas. Así pues, el coste de cada producto queda como

$$2pmn \quad \textit{flops} \qquad y \qquad mp + pn + mn \quad \textit{memops} \tag{4.3}$$

siendo p, m, y n las dimensiones de las matrices a operar y correspondiéndose los valores de estas variables con los dados en las Tablas 4.2 y 4.3 para las capas FC y convolucionales respectivamente.

Tabla 4.2: Dimensión de los operandos en una capa l FC.

	m	p	n
FP	$n^{(l)}$	$n^{(l-1)}$	$\lceil b/P\rceil$
BP-CG	$n^{(l)}$	$n^{(l+1)}$	$\lceil b/P\rceil$
BP-AP	$n^{(l)}$	$n^{(l-1)}$	$\lceil b/P\rceil$

Tabla 4.3: Dimensión de los operandos en una capa l convolucional.

	m	p	n
FP	$k^{(l)}$	$k^{(l-1)}h_W^{(l)}w_W^{(l)}$	$h^{(l)}w^{(l)}\lceil b/P \rceil$
BP-CG	$k^{(l-1)}$	$k^{(l)}h_W^{(l)}w_W^{(l)}$	$h^{(l-1)}w^{(l-1)}\lceil b/P \rceil$
BP-AP	$k^{(l)}$	$h^{(l)}w^{(l)}\lceil b/P \rceil$	$k^{(l-1)}h_W^{(l)}w_W^{(l)}$

De este modo, para calcular la cantidad de operaciones –*flops* y *memops*– que se considerarán por cada una de las capas durante el entrenamiento, deberán sumarse los valores correspondientes según lo expresado anteriormente y que, a modo de resumen, se dejan reflejados en la siguiente tabla esquema:

Tabla 4.4: Esquema del cómputo de operaciones por capa.

	FC	Convolucional	*Pooling*
GEMM	Tabla 4.2+Ec. 4.3	Tabla 4.3+Ec. 4.3	-
im2col	-	Ec. 4.2	Ec. 4.1

Asimismo, otro paso a seguir es la estimación de la velocidad de operación, es decir, la cantidad de operaciones aritméticas capaces de ejecutarse por segundo (FLOPS), teniendo presente las limitaciones en el rendimiento que pueden darse por el ancho de banda de memoria. Para obtener este valor se utiliza el modelo *Roofline* [73]. Este modelo representa, de manera gráfica e intuitiva, el rendimiento real alcanzable estimado en función de la intensidad aritmética, I, siendo esta la relación entre operaciones y bytes accedidos en memoria. Ver Figura 4.1.

Como puede observarse en dicha figura, la velocidad a la que se realiza una operación viene determinada por la intensidad calculada en función del número de *flops* y *memops* como $I = \frac{flops}{memops \cdot \lambda}$ (Medido en *flops*/byte accedido).

En función de esta intensidad, puede calcularse la velocidad real alcanzable en FLOPS, que viene delimitada por la velocidad teórica en coma flotante ($\frac{1}{\gamma}$) o

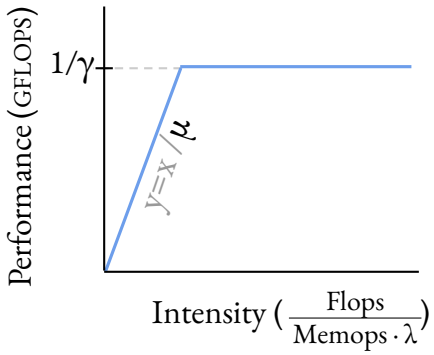

Figura 4.1: Rendimiento alcanzable según el modelo Roofline.

por la limitación del ancho de banda de la memoria ($\frac{1}{\mu}$):

$$FLOPS = \text{mín}\left(\frac{1}{\gamma}, \frac{I}{\mu}\right).$$

Por último, una vez es conocido el número de operaciones y la velocidad a la que estas se ejecutan, bastaría calcular la estimación del tiempo de ejecución como:

$$Tiempo = \frac{flops}{FLOPS}.$$

Nótese que estas ecuaciones están sumamente ligadas al número de *flops*; sin embargo, el coste de la transformación *im2col*, como se ha mencionado previamente, se basa únicamente en *memops*, sin aportar ningún *flop*. Es por ello que, para que las ecuaciones involucradas en el cálculo del coste temporal ofrezcan un valor real para estas transformaciones, en dichos casos se contará el número de *flops* como 1.

4.2.2. Modelo con redes de tipo MLP

Esta modelización del coste computacional se fundamenta en la recogida de evidencias experimentales y en la posterior estimación del coste en base a estas. Concretamente, el objetivo es crear y utilizar redes neuronales de tipo MLP que modelen el coste temporal de los núcleos involucrados en el entrenamiento de la red, así como los costes de copiar las muestras del lote del anfitrión al dispositivo y de la función de optimización utilizada para actualizar los pesos. En adelante, para

abreviar, estos dos últimos conceptos se omitirán del texto, hablando únicamente de núcleos, cuando se haga referencia a los aspectos que estas redes predecirán.

El hecho de usar un enfoque práctico en vez de la anterior versión teórica, aporta al modelo una visión más precisa en cuanto a los detalles inherentes al entrenamiento que, por simplicidad o limitaciones, no se tienen en cuenta en la modelización teórica, pero que las redes MLP son capaces de modelar.

Sin embargo, este beneficio derivado de la capacidad de las redes para abstraerse de ciertas características, también supone que dichas redes están sujetas a las características de las arquitecturas de computadores que componen el clúster y, por ende, su aprendizaje y las estimaciones que pueden realizar están subordinadas a la plataforma para la que han sido entrenadas. Así, el mayor inconveniente de este tipo de modelado es que requiere un entrenamiento individual de las redes MLP para la plataforma utilizada y que el coste de realizar dichos entrenamientos es considerablemente alto, teniendo en cuenta que deben generarse los conjuntos de datos que se utilizarán para ello.

En base a esto, nótese que este modelado ya no depende de los parámetros de la Tabla 4.1 relativos a la plataforma que se va a utilizar; y que, por tanto, los datos a considerar para la estimación y que, en consecuencia, conforman las entradas de los modelos basados en redes de tipo MLP, son los parámetros restantes de dicha tabla: las opciones de entrenamiento elegidas (b, λ y P) y las características de la red asociadas a los núcleos utilizados durante el entrenamiento de cada capa (Véase la Tabla 4.5); así como las opciones de configuración de estos núcleos en caso que hubieran. Así, el número de neuronas de entrada para cada red MLP varía en función de la cantidad de valores que definen la ejecución de cada núcleo. Por ejemplo, para la red MLP encargada de estimar el coste del núcleo `cudnnConvolutionForward` realizado durante el paso *forward* de las capas convolucionales, la entrada se compone de doce entradas correspondientes al tipo de algoritmo utilizado para la convolución (automático, GEMM, GEMM implícita, etc.), las dimensiones de las activaciones de entrada de dicha capa ($k^{(l-1)}$, $h^{(l-1)}$, $w^{(l-1)}$ y el tamaño del lote b), las dimensiones de los pesos ($h_W^{(l)}$, $w_W^{(l)}$ y $k^{(l)}$) y los parámetros de aplicación de estos ($s_h^{(l)}$, $s_w^{(l)}$, $p_h^{(l)}$ y $p_w^{(l)}$).

En cuanto a la salida de las redes de tipo MLP, teniendo en cuenta que el objetivo es estimar el coste temporal del entrenamiento, el valor a predecir por

Tabla 4.5: Núcleos asociados al entrenamiento de cada tipo de capa.

Capa	Núcleo

Convolucional	`cudnnConvolutionForward`
	`cublasSgemm`
	`cudnnConvolutionBackwardFilter`
	`cudnnConvolutionBackwardData`
	`cudnnAddTensor` (add biases)
	`cudnnConvolutionBackwardBias`

FC	`cublasSgemm` `cudnnAddTensor` (add biases) `cublasSgemm` (weights+data gradients) `cublasSgemv` (biases gradient)
Batch Normalization	`cudnnBatchNormalizationForwardTraining` `cudnnBatchNormalizationBackward`
Max Pool	`cudnnPoolingForward` `cudnnPoolingBackward`
Average Pool	`cudnnPoolingForward` `cudnnPoolingBackward`
ReLU	`cudnnActivationForward` `cudnnActivationBackward`
Softmax	`cudnnSoftmaxForward` `cudnnSoftmaxBackward`
Optimizador SGD	`pydtnnSgdKernel`
Copias	`cudaMemcpy`

estas redes es el tiempo de ejecución de cada uno de los núcleos; conformándose, por tanto, la salida de cada red en una única neurona asociada a este tiempo.

De este modo, para calcular el coste computacional en este modelo se emplea una red MLP diferente por cada núcleo utilizado durante el entrenamiento. El diseño de cada red se ha determinado como la combinación de cinco bloques de capas FC+ReLU con cincuenta neuronas cada uno y una última capa FC cuya

salida es la neurona correspondiente a la estimación temporal (Ver Tabla 4.6).

Tabla 4.6: Esquema de las redes de tipo MLP utilizadas para la modelización del coste temporal de los núcleos de la Tabla 4.5.

Id.	Tipo de Capa	#Neuronas
0	Entrada	#Entradas
1–10	FC, ReLU $\times 5$	50
11	FC	1

Una vez diseñadas las redes, es necesario recolectar los datos a utilizar para el entrenamiento y validación de estas. Para ello, inicialmente debe generarse un amplio conjunto de datos de entrada posibles, dando diferentes valores a cada parámetro, de modo que se contemplen las cifras más representativas en las redes neuronales para cada uno. En la Tabla 4.7 puede observarse un ejemplo de esta selección de datos.

Tras tener el conjunto de datos de entrada definido, deben obtenerse las etiquetas asociadas a cada combinación de estos valores, es decir, el coste asociado a cada núcleo. Estos valores se obtienen midiendo experimentalmente el tiempo de ejecución en la arquitectura de computador sobre la que se desee que actúe el modelo, teniendo en cuenta para ello que, a fin de evitar el ruido del sistema en estas mediciones, es conveniente calcular este tiempo como la media de múltiples ejecuciones. Una vez se tienen los datos etiquetados, es recomendable considerar la normalización de estos para reducir el rango de los valores de entrenamiento.

El último paso que quedaría para conseguir el modelo de la parte computacional mediante MLP sería, pues, el entrenamiento de cada una de estas redes con los datos generados y el uso de estas para estimar los costes.

4.3. Coste de las Comunicaciones

La estimación del coste de las comunicaciones *AllReduce* implicadas en la actualización de pesos del paralelismo de datos se aproximará mediante un modelo

Tabla 4.7: Conjunto de datos de entrada para las redes MLP de estimación.

Parámetros	Conjunto de valores
$n^{(l-1)}$, $n^{(l)}$	$128, 256, 512, \ldots, 1280 \times 10^3\}$
$k^{(l-1)}$	$\{3, 4, 8, 16, 32, 64, 128, 256, 512\}$
$h^{(l-1)}$, $w^{(l-1)}$	$\{8, 14, 16, 28, 32, 56, 112, 224\}$
$h_W^{(l)}$, $w_W^{(l)}$	$\{1, 3, 5, 7, 9\}$
$s_w^{(l)}$, $s_h^{(l)}$	$\{1, 2, 4\}$
$p_w^{(l)}$, $p_h^{(l)}$	$\{1, 3, 5, 7, 9\}$
b	$\{16, 32, 64, 128, 256\}$

teórico de esta comunicación [20, 55]. Para ello, debe conocerse el algoritmo utilizado durante el entrenamiento y es que esta primitiva en concreto, dispone de diferentes algoritmos de implementación, siendo algunos de los más conocidos *basic linear*, *ring*, *recursive doubling* y *Rabenseifner* [20, 32, 70]; denotados en adelante por las abreviaciones LIN, RNG, RDB y RSA respectivamente.

Para realizar la comunicación colectiva estos algoritmos dividen el trabajo en dos partes: un primer paso de reducción en el que se calcula el resultado deseado y un segundo paso en el que este resultado es difundido a todos los procesos. El modo de llevar a cabo estos dos pasos es lo que distingue a cada algoritmo*:

- LIN: Para obtener la reducción con este algoritmo se utilizan comunicaciones punto a punto, centrando el cálculo en un único proceso, de modo que uno de los procesos participantes recibe del resto sus datos, calcula el resultado de la reducción y lo envía al resto de procesos.

- RNG: En este algoritmo se comunican los procesos mediante intercambios entre pares, utilizando para la elección de los procesos que se intercambian los datos una topología de anillo e intercambiando secciones equivalentes de datos en vez del mensaje completo.

- RDB: Utilizando también intercambios entre pares de procesos, el paso de reducción de este algoritmo se basa en realizar intercambios del mensaje

*Pueden consultarse esquemas gráficos del proceso de cada algoritmo en el Apéndice A

completo entre procesos separados a una distancia de 2^{iter}. Inicialmente se intercambian los datos entre procesos contiguos (2^0: 0 y 1, 2 y 3, etc.), de modo que cada proceso es capaz de obtener la reducción de dos datos. Tras ello, se intercambian la reducción entre los siguientes procesos más cercanos (2^1: 0 y 2, 1 y 3, etc.) y se continúa iterando estos intercambios entre los procesos 2^{iter} hasta alcanzar la reducción total por algún proceso. En este momento, al trabajar con mensajes completos, todos los procesos tienen el resultado de la reducción y no es necesario realizar una difusión puesto que esta se ha integrado durante el proceso de reducción.

- RSA: Este algoritmo tiene como base el mismo procedimiento que el anterior algoritmo, RDB. Sin embargo, en este caso los datos que los procesos se intercambian son fracciones del mensaje completo: en cada iteración tratan con una sección $\frac{1}{2^{iter}}$-ésima del mensaje. Análogamente, el paso de difusión del resultado se realiza en modo inverso con comunicaciones de secciones de este resultado del mismo tamaño.

Así pues, del mismo modo que se aproximó el coste computacional en la Sección 4.2.1 en función de las operaciones realizadas y de sus particularidades, puede estimarse el coste de cada algoritmo teniendo en consideración sus particularidades: las comunicaciones y operaciones de reducción que se producen durante la ejecución y el volumen de datos que involucran.

Para este cómputo se considerará que la red de interconexión entre los nodos se ajusta a una topología en estrella, de modo que, a través de un conmutador central al cual se conecta mediante un enlace cada nodo, se proporciona un ancho de banda completo entre cada par de nodos a la vez. Además, por simplicidad a la hora de contabilizar los pasos de los algoritmos, el número de procesos se asumirá que corresponde a una potencia de 2.

Teniendo en cuenta estas premisas y en función de lo arriba descrito, la Tabla 4.8 detalla para cada uno de los algoritmos las comunicaciones y operaciones que se deben contabilizar para estimar su coste y, conforme a estas, se esquematiza dicho coste en: el coste de las operaciones sujeto al rendimiento γ, y el coste de las comunicaciones dependiente de la latencia (α) y del ancho de banda del enlace (β).

Tabla 4.8: Desglose de costes de la primitiva Allreduce.

	Reducción		Difusión		Costes		
	Nº Com. y Op.	Bytes	Nº Com.	Bytes	($\times\alpha$)	($\times\beta$)	($\times\gamma$)
LIN	$p-1$	n	$p-1$	n	$2(p-1)$	$2n\ (p-1)$	$n\ (p-1)$
RNG	$p(p-1)$	n/p	$p(p-1)$	n/p	$2(p-1)$	$2n\ \frac{p-1}{p}$	$n\frac{p-1}{p}$
RDB	$\log p$	n	-	-	$\log p$	$n\ \log p$	$n\ \log p$
RSA	$\log p$	$n/2^{iter}$	$\log p$	$n/2^{iter}$	$2\log p$	$2n\ \frac{p-1}{p}$	$n\frac{p-1}{p}$

4.4. Modelo Analítico

En esta sección se evalúa el modelo íntegramente teórico diseñado para la estimación del coste del entrenamiento, es decir, el modelo cuyos costes de cómputo y comunicación se aproximan mediante modelizaciones analíticas en base a lo expresado en las Secciones 4.2.1 y 4.3. La definición de este modelo se deriva, por tanto, de la suma de ambos costes, considerando las operaciones y comunicaciones que se realizan en cada una de las capas contempladas –convolucionales, FC y de *pooling*– durante los pasos *forward* y *backward*.

4.4.1. Validación

Para comprobar que dicho modelo resulta una herramienta válida para la predicción de estos costes es necesario validar que las estimaciones que realiza son aproximadas al coste real del entrenamiento. A continuación se muestra este contraste entre los valores esperados y los reales medidos experimentalmente. Para la obtención de estos datos se han medido los tiempos de entrenamiento en PyDTNN de la conocida red neuronal VGG11. Estos experimentos se han realizado sobre una plataforma que consta de 12 nodos, cada uno compuesto por dos procesadores Intel Xeon E5645 Westmere de 6 núcleos; una memoria RAM DDR3 de 48 GiB y un conmutador Mellanox QDR Infiniband para las conexiones entre los nodos. En esta aplicación, el valor de los parámetros que definen el coste del modelo son:

- $L, n^{(l)}, h^{(l)}$, etc: definidos según la red VGG11.
- $\lambda = 8$ bytes/dato: Uso de datos en coma flotante de 64 bits.
- $P = 12$ procesos: Ejecución con un único proceso por nodo.
- $\gamma = 115,2$ GFLOPS: Obtenido como resultado del producto 8 flops/ciclo de rendimiento máximo por nodo × 2.4 GHz de frecuencia × 6 núcleos/nodo.
- $\mu = 32$ Gbytes/seg: Calculado como el producto de la multiplicación 8 Bytes/ciclo de ancho de cada bus × 3 buses × 1.333 GHz de ratio de reloj de memoria.
- $\alpha = 1{,}4$ μseg: medido experimentalmente con el benchmark `ib_send_lat` de *libibverbs v5.5*.
- $\beta = 25{,}8$ Gbps: medido experimentalmente con el benchmark `ib_send_bw` de la misma biblioteca.

Además, en cuanto a las comunicaciones, para llevar a cabo la comunicación colectiva *AllReduce* se ha decidido utilizar el algoritmo de *Rabeinseifer* (RSA); por ello la ecuación utilizada para calcular el coste de las comunicaciones será la asociada a este algoritmo en la Tabla 4.8.

La Figura 4.2 muestra los resultados obtenidos durante los pasos *forward* y backward: para cada capa puede observarse en el eje izquierdo de la gráfica el tiempo estimado por el modelo teórico en comparación con el tiempo real de ejecución. Asimismo, se representa en el eje derecho la diferencia de tiempos calculada tanto relativamente $((T_{aprox} - T_{real})/T_{real})$, como ponderada $((T_{aprox} - T_{real})/T_{medio})$ para ofrecer una perspectiva sobre la importancia del efecto de estos errores sobre el coste total de la red, pues el valor de un error relativo grande asociado a una capa de poco coste no es tan significativo como pueda parecer, como ocurre en este caso, por ejemplo, en el paso *forward* de la capa 7.

Observando estos resultados, se aprecia que el error relativo se posiciona mayoritariamente en torno a un $\pm 20\,\%$ con algunos máximos en los que se sobreestima el coste en hasta un $50\,\%$. Nótese, sin embargo, que cuando se compara este error con el ponderado, los resultados con mayor error relativo se asocian a partes del

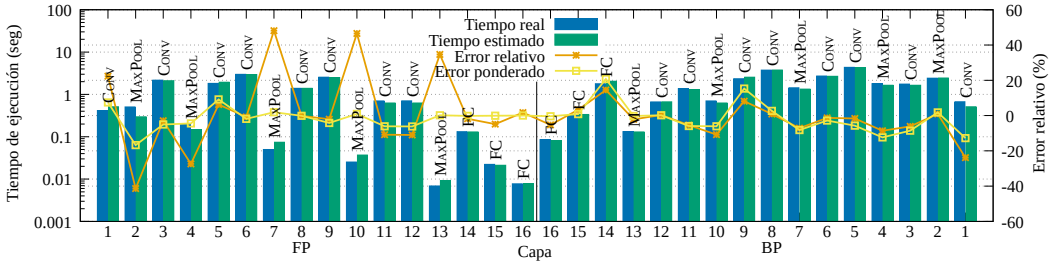

Figura 4.2: Tiempos estimados y reales de un paso FP+BP de la red VGG11.

entrenamiento con poco peso sobre el coste total (*forward* de las capas *pooling*) y, en consecuencia, un bajo error ponderado; mientras que los tiempos más significativos (*forward* - 6 y 9; *backward* - 8 y 5) tienen unas estimaciones muy precisas y mantienen un error, tanto relativo como ponderado, considerablemente bajo. En efecto, si se observa únicamente el error ponderado, este se mantiene en valores pequeños, generalmente entre $\pm 10\,\%$. Particularmente, la media del error ponderado (en valores absolutos) cometido en estas estimaciones es de un $5{,}3\,\%$. Por este motivo, y teniendo en cuenta que las sobreestimaciones de ciertos costes se compensan con las subestimaciones de otros, el tiempo total estimado del paso *forward-backward* de este entrenamiento tiene un error relativo de alrededor de un $1\,\%$ respecto al coste real.

A la vista de los resultados obtenidos, dado el bajo error cometido, puede considerarse este modelo como una herramienta válida para realizar estimaciones del coste temporal del entrenamiento.

4.5. Modelo con redes de tipo MLP

El modelo que se analiza en esta sección combina la modelización analítica para la parte del coste asociada a las comunicaciones, con la modelización basada en redes neuronales de tipo MLP para la predicción del coste vinculado al cómputo.

De este modo, el coste temporal del entrenamiento se predice en este modelo como la suma de la estimación teórica de las comunicaciones necesarias para la actualización de pesos –Sección 4.3– y la predicción obtenida de las redes neuronales de tipo MLP para el coste computacional –Sección 4.2.2– en función

de los núcleos ejecutados. Cabe recordar que, para este modelo, es necesario crear y entrenar previamente las redes de tipo MLP que lo constituyen, así como generar los datos a utilizar en dicho entrenamiento.

Para combinar ambos modelos se ha optado por el desarrollo de un simulador que realice la emulación de las diferentes fases del entrenamiento y, en función de los parámetros de la red neuronal, de entrenamiento (b y λ) y del clúster (P, modelo de GPU, configuración de la red). Así, el simulador se encarga de realizar las invocaciones pertinentes a las redes de tipo MLP y al modelo analítico de la comunicación para calcular los costes parciales que, en conjunto, aproximen el tiempo total del paso *forward-backward* de entrenamiento.

4.5.1. Validación

Al igual que en la sección anterior, en esta sección se desea evaluar la efectividad de este modelo comprobando si la predicción de los costes del entrenamiento que este genera son afines a los tiempos reales de ejecución. Para ello, en este caso se valida inicialmente el uso de cada red de tipo MLP individualmente y, una vez queda comprobado que sus predicciones aproximan correctamente el coste de cada sección de cómputo del entrenamiento, puede validarse si el uso combinado de estas redes predice correctamente el tiempo de cómputo.

Entrenamiento. Antes de explicar las pruebas realizadas y analizar los resultados obtenidos, tal y como se ha mencionado, es necesario crear y entrenar las redes neuronales que permitirán predecir el coste relativo al cómputo y generar el conjunto de datos que permitirá dicho entrenamiento.

En primer lugar, se crean las propias redes neuronales MLP tal y como se describía en la sección correspondiente y se genera el conjunto de datos de entrenamiento. Para la generación de datos se ha considerado como datos de entrada las posibles combinaciones de los conjuntos descritos en la Tabla 4.7. Las etiquetas de estos datos se han obtenido, para cada red de tipo MLP, mediante la medición del tiempo de ejecución del núcleo correspondiente, realizando dicha medición 100 veces y tomando como valor para la etiqueta la media de dichas repeticiones. En este caso, para contar con una mayor cantidad de datos y, por tanto, obtener una mayor fiabilidad de los resultados, estas ejecuciones se han realizado sobre dos

GPU –una NVIDIA Tesla Volta V100 PCIe y una Ampere A100 PCIe– de modo que se cuenta con dos conjuntos diferentes de datos. Una vez generados los datos, se han normalizado utilizando la función log y se han particionado los datos en dos conjuntos: datos de entrenamiento (80 %: 64 % entrenar, 16 % validación) y datos de testeo (20 %).

El entrenamiento de cada red se ha realizado utilizando el entorno Tensorflow (Versión 2.2.0). Los parámetros utilizados para estos entrenamientos han sido: optimizador Adam, función de pérdida MSE, ratio de aprendizaje inicial 10^{-3} con un decrecimiento de $\times 0{,}1$ cada vez que la validación no mejora durante 15 épocas consecutivas y un criterio de detener el entrenamiento si la mejora no se ha alcanzado en 20 épocas.

Validación de las redes MLP. Una vez entrenadas las redes de tipo MLP se han puesto a prueba con las muestras de los dos conjuntos de datos reservadas para ello. La Figura 4.3 muestra dos gráficas con los resultados obtenidos para cada GPU. En cada gráfica se representa un histograma y diagrama de cajas del error relativo cometido en las predicciones de cada red de tipo MLP respecto del tiempo real de ejecución del correspondiente núcleo.

Nótese que ambas gráficas muestran unos resultados bastante similares: en el caso de las redes de tipo MLP de los núcleos asociados a las capas convolucionales y *batch normalization*, en ambos tipos de capas la media del error relativo cometido ronda el 15 %; mientras que para los núcleos asociados al resto de capas –FC, *pooling*, ReLU, *softmax*– así como la copia a memoria y el optimizador SGD, este valor se reduce considerablemente, quedando el error relativo medio alrededor de un 5 %.

En base a estas observaciones, puede concluirse que las estimaciones que realizan las redes de tipo MLP en ambas GPU analizadas son una buena aproximación de los tiempos reales de ejecución de cada núcleo por separado.

El siguiente paso, pues, es evaluar el efecto de utilizar estas redes de tipo MLP en conjunto. Para ello, los experimentos que se han realizado consisten, de nuevo, en ejecutar el entrenamiento de la red neuronal VGG11 en el entorno PyDTNN. En este caso la plataforma utilizada se compone de 8 nodos, cada uno con un procesador Intel Xeon Gold 5120 de 14 núcleos con una frecuencia de 2.20 GHz;

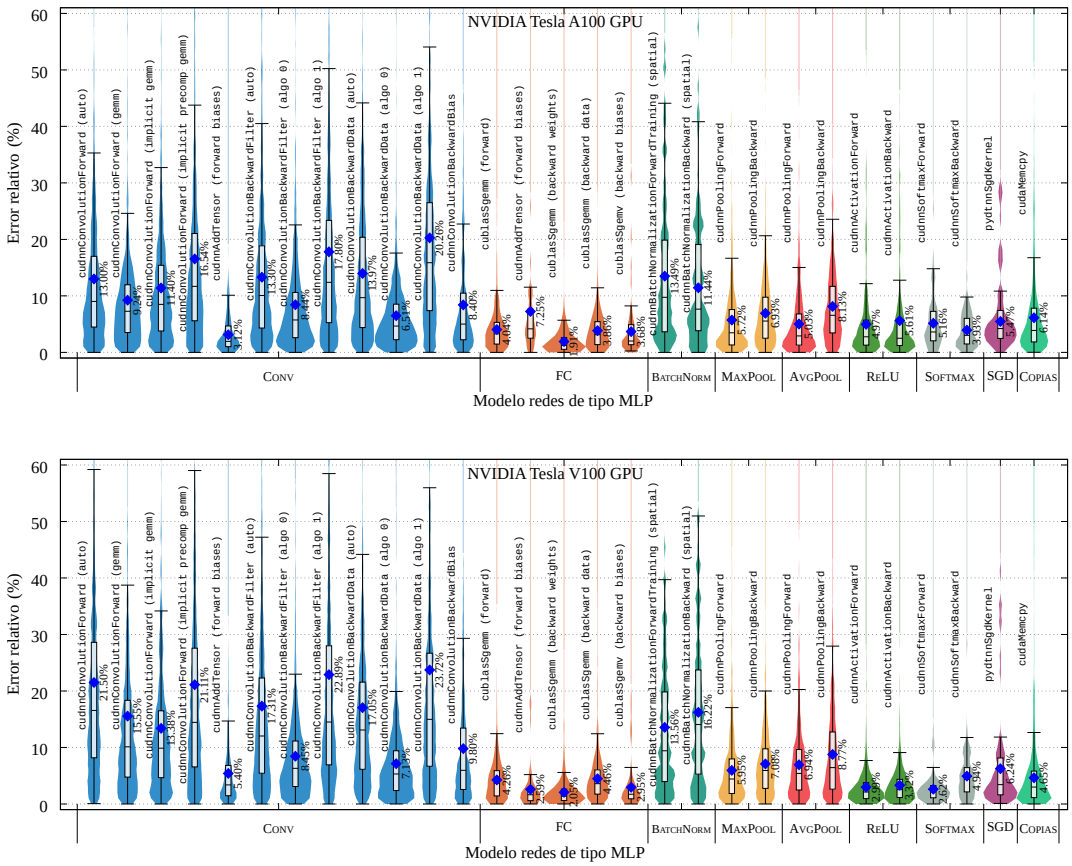

Figura 4.3: Histogramas y gráficos de cajas de los errores relativos cometidos en las estimaciones realizadas para cada núcleo por las redes de tipo MLP, usando las GPU NVIDIA Tesla A100/V100.

una memoria RAM DDR4 de 187 GiB y una NVIDIA Tesla V100 PCIe con 32 GiB de HBM2. La conexión de los nodos se establece mediante una red Infiniband EDR cuyo ancho de banda es 100Gbps. El valor de los parámetros que definen el coste del modelo son, en este caso, los siguientes:

- $L, n^{(l)}, h^{(l)}$, etc: definidos según la red VGG11.
- $b = 128$.
- $\lambda = 4$ bytes/dato: Uso de datos en coma flotante de 32 bits.
- $P = 8$ procesos: Ejecución con un único proceso por nodo.

- $\alpha = 30$ μseg: medido experimentalmente con el benchmark de NCCL.
- $\beta = 12{,}24$ Gbps: medido experimentalmente con el benchmark de NCCL.

Nótese que en este modelo se han omitido los parámetros relativos a la plataforma, γ y μ, pues las propias redes de tipo MLP son las encargadas de aprender y aplicar el efecto de estas propiedades de la plataforma.

En la Figura 4.4 pueden observarse los tiempos estimados y de ejecución de cada núcleo llamado durante el entrenamiento de la red –eje izquierdo–. También puede observarse el error relativo cometido, así como el ponderado –eje derecho–. Este último, de nuevo, deja apreciar que los valores más altos del error relativo están asociados a núcleos cuyo peso es menor y que, por tanto, no ejercen un gran efecto sobre el error total cometido. En particular, los puntos con peores estimaciones se asocian a los núcleos de las capas FC y ReLU durante el paso *forward*. El error ponderado, de hecho, toma valores generalmente pequeños: en este experimento su valor absoluto medio es, durante el paso *forward* de 1.6 %, en el cálculo de gradientes del paso *backward* la media es de 3.2 % y para la actualización de pesos, la media del error ponderado absoluto es de 3.1 %. En cuanto al error relativo, se observa que se mantiene en valores máximos de ± 20 %, siendo el error relativo total de ± 5 %, con la compensación de los costes sobre y subestimados.

Se ha realizado también este mismo experimento con un tamaño de lote inferior, $b = 64$, mostrando resultados similares: error relativo en el intervalo $[-20\,\%, +20\,\%]$ y errores ponderados de bajos valores, siendo la media de los errores relativos absolutos de 3.2 % para el paso *forward* y 2.5 % para el cálculo del gradiente durante el paso *backward**.

En general, por tanto, se mantiene un error en la estimación global de los núcleos considerablemente bajo, como ocurría en las estimaciones individuales. De este modo se confirma la validación de las redes de tipo MLP como herramienta válida para la estimación del tiempo de cómputo.

*El cálculo de los errores de la actualización de pesos no se ha vuelto a realizar al ser este cálculo independiente del tamaño del lote.

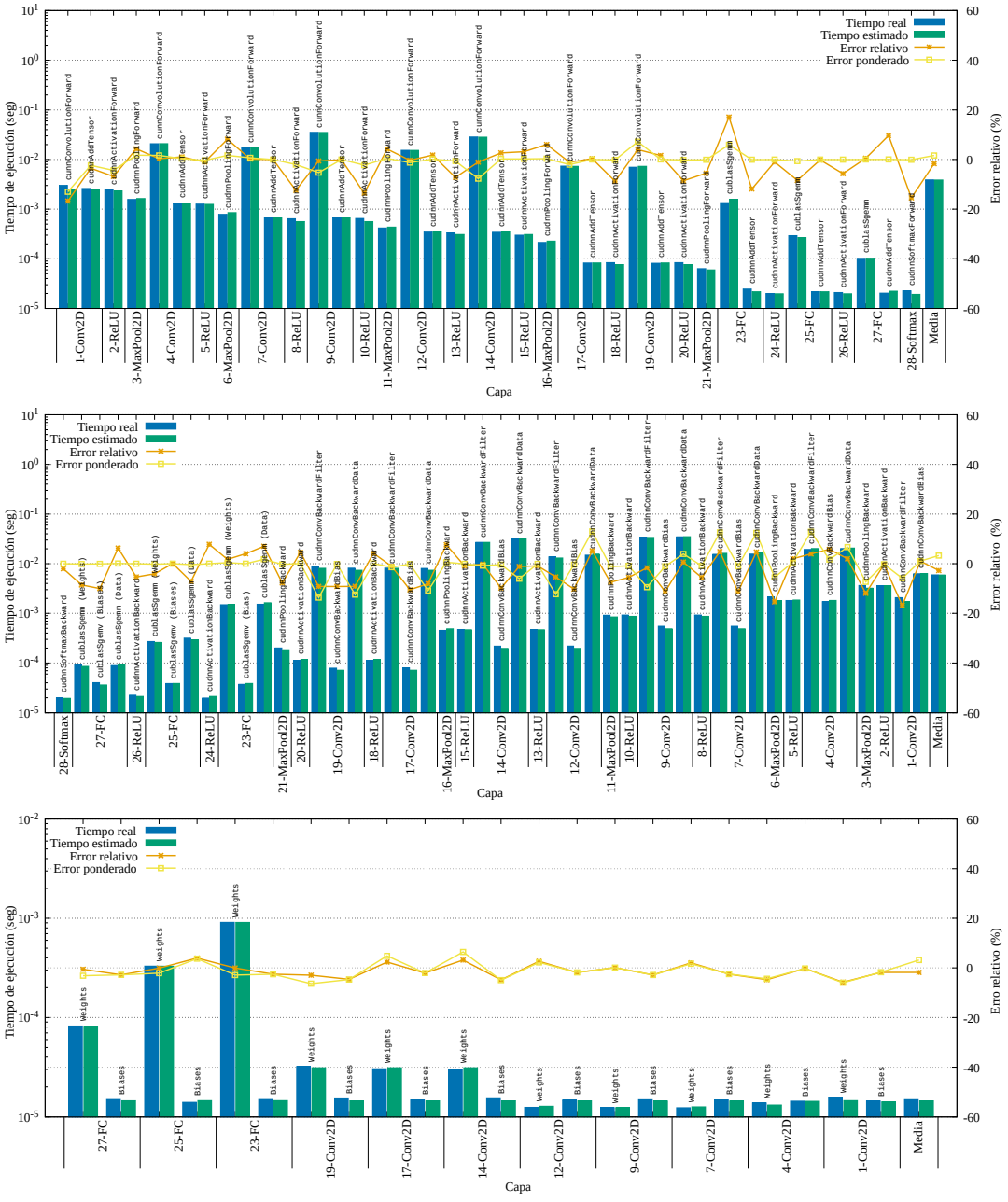

Figura 4.4: Tiempos reales y esperados de cómputo por cada núcleo invocado durante el entrenamiento (paso *forward*, cálculo del gradiente y actualización de pesos) de la red VGG11.

Validación de comunicaciones. Por último, es necesario validar el modelo para el coste de las comunicaciones. En este caso, se ha realizado este análisis por separado para poder observar la validez de estas predicciones sobre diferentes escenarios. En este caso, para la ejecución de la comunicación colectiva *AllReduce* se ha optado por el algoritmo *Ring* (RNG) y, por tanto, se utiliza la ecuación asociada a este algoritmo para la estimación teórica de su coste. Para la obtención de los tiempos reales, se han realizado mediciones individuales de la primitiva NCCL *AllReduce* en la plataforma descrita anteriormente. En estos experimentos se ha analizado el coste de estos intercambios para 6 y 8 procesos (asignando un proceso por GPU), y para tamaños de mensajes entre 500 KiB y 1 GiB. La Figura 4.5 muestra los resultados obtenidos de estas mediciones, junto con la estimación que ofrece el modelo teórico y el error relativo cometido. Obsérvese que el error relativo disminuye a medida que se incrementa el tamaño de los mensajes, tomando este valores por debajo del 10 % cuando se trata de intercambios de al menos 80 MiB. El hecho de que estos errores se incrementan cuando el tamaño del mensaje es pequeño, se vincula con que su coste temporal es muy bajo y una pequeña desviación en su aproximación incrementa notablemente el error relativo. Teniendo esto en cuenta, estos errores relativos, pese a ser notables, no afectan significativamente en la estimación total del coste del entrenamiento.

En consecuencia, el modelo teórico puede considerarse que genera unas aproximaciones válidas para calcular el coste de comunicaciones del entrenamiento. Por tanto, dado que previamente las aproximaciones del cómputo mediante redes de tipo MLP también han sido validadas, es factible aceptar que la combinación de ambos modelos sumando ambas partes de los costes, cómputo y comunicación, ofrecerán una estimación del coste total del entrenamiento bastante aproximada al coste real.

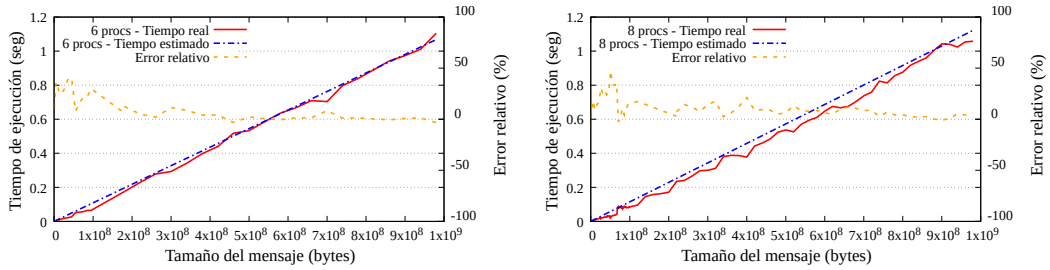

Figura 4.5: Tiempos reales y esperados para la comunicación colectiva *AllReduce* para 6 y 8 procesos.

4.6. Análisis de Rendimiento y Escalabilidad del Paralelismo de Datos

Tras presentar los modelos de coste diseñados y validados, en esta sección se hace uso de estos como herramienta para analizar el rendimiento y la escalabilidad que ofrece el paralelismo de datos que se ha estudiado en capítulos anteriores. A continuación, se muestra cómo estos modelos permiten estudiar teóricamente los rendimientos alcanzables y los diferentes cuellos de botella que pueden aparecer en función de los recursos con los que se cuente.

Para este estudio se realiza un análisis sobre cinco de los parámetros que influyen en el tiempo de ejecución: rendimiento aritmético en coma flotante (γ), ancho de banda de la memoria (μ), ancho de banda del enlace (β), número de nodos (P) y tamaño del lote local (b). El objetivo de este estudio es, partiendo de un clúster base, observar qué efecto tiene variar cada uno de dichos parámetros: cómo varía el rendimiento, qué limitaciones se observan y qué factores determinan dichas limitaciones.

El hipotético clúster base sobre el que se realiza este estudio se ha definido con las siguientes características: $P = 100$ nodos, cada uno equipado con un procesador Intel Xeon 8180M (28 cores con una frecuencia de 2.5 GHz), 256 GiB de memoria RAM y una GPU NVIDIA Tesla A100 con 40 GiB de memoria HBM2. La red de conexión Infiniband HDR con $\beta = 400$ Gbps, $\alpha = 0{,}5\mu$seg y una topología de estrella. Además, en el caso base de entrenamiento se establece el tamaño del batch en $b = 128$.

4.6 Análisis de Rendimiento y Escalabilidad del Paralelismo de Datos

Para tener un visión de la escalabilidad del paralelismo de datos más amplia, se ha realizado este análisis para diversas redes neuronales – VGG16, ResNet-50 y DenseNet-121 –, así como dos conjuntos de datos – CIFAR-10 y ImageNet– de diferentes características. Las Figuras 4.6 y 4.7, muestran los tiempos para cada uno de los conjuntos de datos. En cada figura puede observarse cómo afecta la variación del valor de cada uno de los cinco ejes de estudio (filas) para cada red estudiada (columnas). Cada gráfica representa el tiempo estimado para las comunicaciones (Sección 4.3) y el cómputo, este último tanto con el modelo analítico (Sección 4.2.1), como con el modelo de redes de tipo MLP (Sección 4.2.2); asimismo, en cada gráfica se indica con una línea vertical negra el valor base para dicho parámetro.

A partir de los resultados contenidos en estas figuras puede realizarse las siguientes observaciones:

- Eje γ: En este caso solo se muestra el cómputo estimado mediante el modelo analítico. Esto se debe a que, al estar el modelo basado en redes de tipo MLP ligado a la plataforma sobre la que estas han estado entrenadas, no es posible realizar estimaciones variando este parámetro. Los datos que se extraen de los modelos analíticos muestran que el hecho de aumentar el rendimiento, γ, resulta en una reducción del tiempo; sin embargo, esta reducción se ve limitada llegado cierto punto, de modo que, a partir de este, el hecho de mejorar el rendimiento de los nodos no supone una mejora del tiempo. Esto se debe a que, hasta dicho punto, el coste había estado dominado por la cantidad de *flops* que podían ejecutarse por unidad de tiempo; sin embargo, a partir de este punto, el coste pasa a estar limitado por los accesos a memoria, por lo que contar con un mayor valor de FLOPS no será significativo. En el caso de la red VGG16 con CIFAR-10, sin embargo, cuando se mejora el rendimiento del nodo, el cuello de botella no lo define el acceso a memoria sino las comunicaciones (en $\gamma = 30$).

- Eje μ: Mejorar el ancho de banda a memoria supone, excepto en el caso de la red VGG16 con CIFAR-10, una constante mejora del tiempo de ejecución. Esto demuestra en el resto de casos la alta vinculación con la memoria. Estas gráficas, de nuevo, no muestran la estimación del cómputo mediante

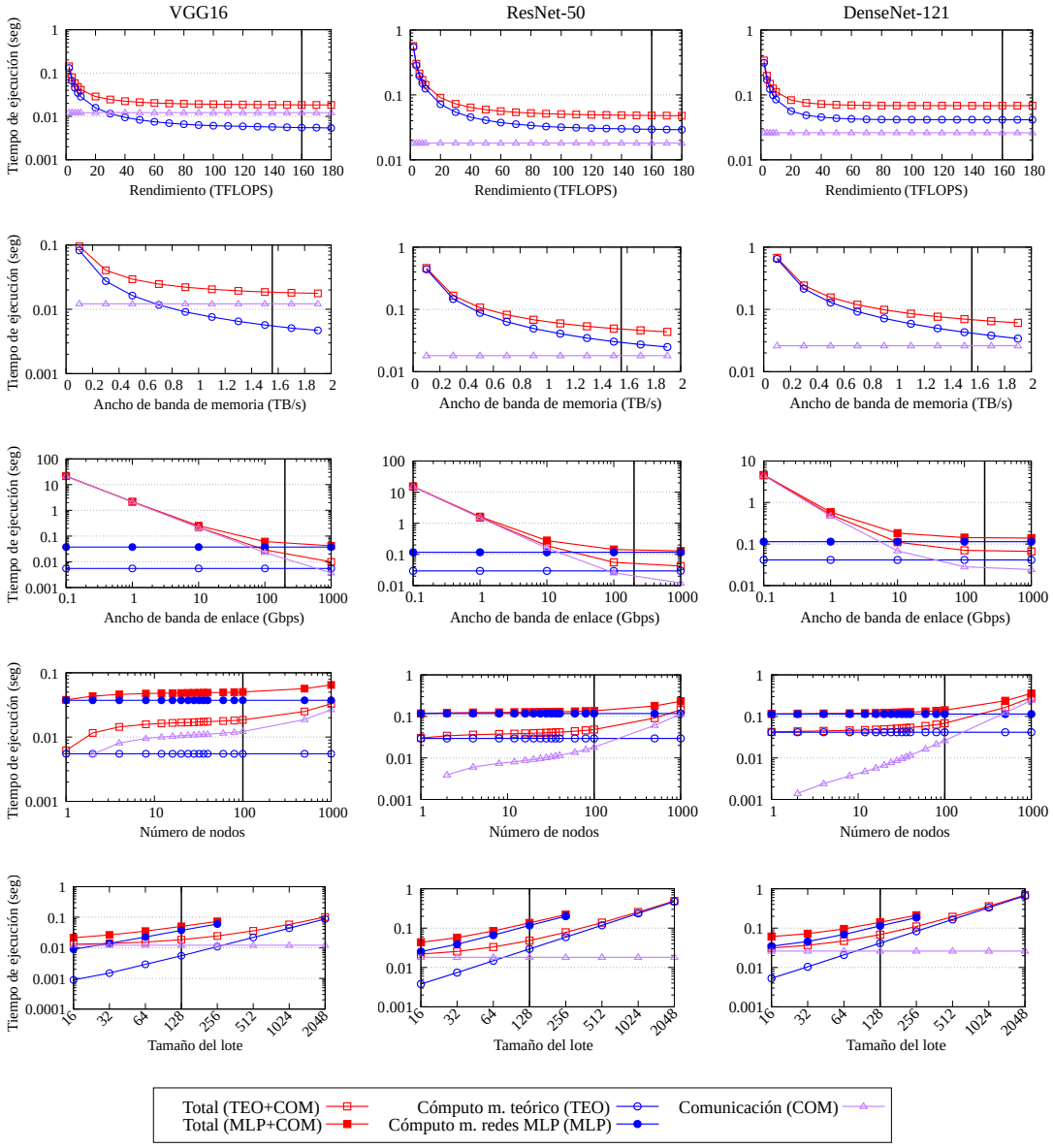

Figura 4.6: Tiempo de ejecución estimado para un paso de entrenamiento *forward-backward* de un lote de la base de datos CIFAR-10, utilizando diferentes redes neuronales (columnas: VGG16, ResNet50 y Densenet-121) y variando los diferentes parámetros que influyen (filas: rendimiento, ancho de banda de memoria, ancho de banda de enlace, número de nodos y tamaño del lote).

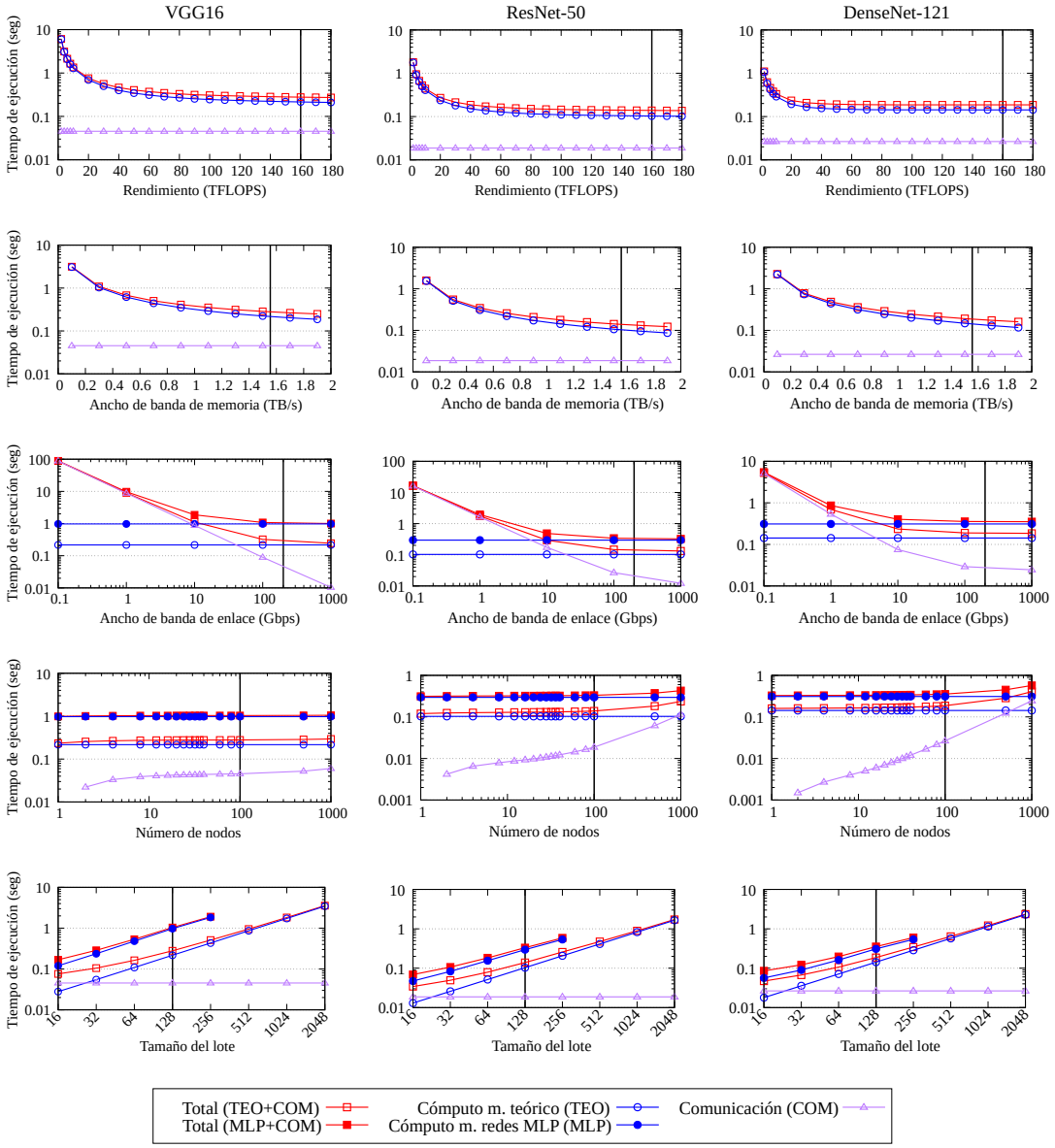

Figura 4.7: Tiempo de ejecución estimado para un paso de entrenamiento *forward-backward* de un lote de la base de datos ImageNet, utilizando diferentes redes neuronales (columnas: VGG16, ResNet50 y Densenet-121) y variando los diferentes parámetros que influyen.

redes de tipo MLP por la dependencia que tiene el aprendizaje de dichas redes con este parámetro de la plataforma.

- Eje β: Estas gráficas revelan el papel determinante de la red de comunicaciones en el tiempo de entrenamiento. Nótese que un ancho de banda de enlace bajo, $\beta = 10$ Gbps, genera un cuello de botella significativo, pasando a suponer las comunicaciones la mayor parte del tiempo de entrenamiento. Por esto, redes con un ancho de banda de enlace considerablemente bajo, como Gigabit Ethernet, no ofrecen buenas opciones para el entrenamiento de redes neuronales.

- Eje P: Al incrementar la dimensión del clúster, se incrementa también el tamaño del lote global ($P \times b$). Este hecho no resulta en un mayor coste de cómputo debido a la paralelización, pero sí que tiene un importante efecto en el coste de comunicaciones. Si se recuerdan las ecuaciones que definen el coste de los algoritmos *AllReduce*, estas tienen una dependencia lineal respecto de P, por lo que aumentar el número de nodos supone un incremento lineal del coste de las comunicaciones, pasando a ser estas, a partir de cierto valor de P, el cuello de botella.

- Eje b: Lo más significativo en estas gráficas es la limitación que se observa, debido a las comunicaciones y accesos de memoria, cuando el tamaño del lote es muy pequeño. En estas gráficas el alcance las predicciones por redes de tipo MLP solo llega hasta lotes de tamaño 256 por la generación de datos que se pudo realizar para el entrenamiento de estas redes (el espacio de trabajo de los núcleos excedería la memoria disponible en las GPU).

- Cómputo analítico vs redes de tipo MLP: El coste computacional estimado por las MLP es superior al coste ofrecido por el modelo analítico. Ambos, sin embargo, muestran tendencias similares.

4.7. Conclusiones

A lo largo de este capítulo se muestran dos modelos para la estimación del coste temporal del entrenamiento distribuido de las RNP mediante el paralelis-

mo de datos. El primero de ellos consistente en una modelización teórica en la que, tanto el coste computacional, como el de comunicación, se obtienen a partir de fórmulas análiticas que aproximan el tiempo de ejecución en base a la contabilización de operaciones y transferencias y a la velocidad a la que estas se ejecutan. El segundo modelo expuesto utiliza, en cambio, una aproximación del cómputo con base experimental, empleando redes del tipo MLP que predicen el coste de cada núcleo ejecutado en el entrenamiento. Se ha validado experimentalmente que ambos modelos ofrecen unas estimaciones del coste real bastante aproximadas. En los experimentos con el modelado teórico, se ha cometido un error relativo ponderado* de $\pm 10\,\%$ por capa, siendo la media de un $5{,}3\,\%$. Al analizar los costes aproximados por las redes MLP del segundo modelo propuesto, el error relativo ponderado decae a una media de un $3{,}1\,\%$. Cabe destacar, sin embargo, que al contrario del modelo analítico, este modelo conlleva un coste inicial adicional, al tener que entrenar las redes MLP y, por tanto, tener también que crear el banco de datos para dicho entrenamiento; quedando así asociado este proceso a la plataforma sobre la que se hayan obtenido dichos datos y, por tanto, dando estimaciones válidas únicamente para dicha plataforma. No obstante, el entrenamiento también permite que este modelo aproxime los costes en base a características y consideraciones inherentes a la plataforma que no serían factibles incluir en un modelo exclusivamente analítico.

Finalmente, en este capítulo se han utilizado los modelos expuestos para analizar el rendimiento del entrenamiento distribuido mediante paralelismo de datos. De este análisis se distingue la proporcionalidad entre el tamaño de los lotes y el tiempo de ejecución de los pasos *forward+backward* del entrenamiento, manteniendo el coste global de este; y, aunque el incremento de este valor beneficia la intensidad aritmética por nodo, debe tenerse en consideración que sobredimensionar b puede suponer la pérdida de convergencia del entrenamiento. Así mismo, destacan las comunicaciones como el factor que predominantemente limita el rendimiento del entrenamiento. El coste de las primitivas *AllReduce* se convierte en cuello de botella, tanto en el escenario en que se incrementa la dimensión del

*Al ponderar un peso, se le aplica un factor en función del coste que dicha parte del entrenamiento aporta al coste total.

clúster (en los datos extraídos, para $P > 100$), así como cuando el ancho de banda de enlace toma valores inferiores a un umbral mínimo (particularmente, en los casos observados este valor se define en los 5–40 Gbps) y, del mismo modo, cuando los lotes no alcanzan un tamaño mínimo.

Capítulo 5

Análisis de la Primitiva *AllReduce*

En este capítulo se analiza el rendimiento de la primitiva de comunicación colectiva *AllReduce*, clave para el entrenamiento distribuido en paralelo de las RNP [15, 17, 18]. A lo largo del capítulo se estudiará el coste temporal de esta comunicación, analizando la diferencia entre el coste teórico y el real, se mostrarán nuevos modelos creados para reducir esta diferencia y se estudiará la relevancia de la elección del algoritmo de implementación de esta primitiva.

5.1. Introducción

Anteriormente se ha mostrado que la primitiva *AllReduce* aparece durante el entrenamiento distribuido de las RNP cuando se explota el paralelismo de datos. En el campo de las RNP, es frecuente que los entornos de entrenamiento empleen el esquema de paralelización de datos para aprovechar los recursos de los que se dispone, utilizando para la capa de comunicaciones la interfaz de programación de aplicaciones MPI. En particular, en este tipo de paralelismo se emplea la primitiva *AllReduce* que realiza, como se ha visto anteriormente en la Sección 2.5.1, la reducción y distribución de los cálculos de gradientes locales generados durante el entrenamiento. Este es el caso de entornos de entrenamiento populares, como TensorFlow y PyTorch, pero también, particularmente, del entorno desarrollado PyDTNN [1, 42].

En el ámbito de la computación de altas prestaciones, esta y otras comunicaciones colectivas son empleadas frecuentemente durante la ejecución distribuida de numerosas aplicaciones. En este sentido, el incremento de recursos hardware y la consecuente distribución de tareas implica, en gran parte de los casos, que durante la ejecución de estas aplicaciones paralelas, los procesos cooperantes requieran compartir información entre sí. De este modo, mientras que el incremento de núcleos y/o nodos supone generalmente una mejora del rendimiento global del sistema, los movimientos de datos necesarios para la correcta ejecución de la aplicación implican, en muchas ocasiones, que la red de interconexión se convierta en un cuello de botella que limite el rendimiento global alcanzable [20, 24]. En particular, el entrenamiento de datos resulta uno de estos casos, tal y como muestran los resultados de la Sección 4.6, en los que las comunicaciones se convierten en un factor decisivo para determinar el rendimiento máximo alcanzable [13, 38].

Si se observa la evolución de las interconexiones y se compara con el desarrollo producido en la potencia de cálculo, tanto en GPU como en CPU, puede apreciarse que el progreso alcanzado en este último caso ha aumentado a una velocidad significativamente mayor. A muestra de ejemplo, en 2020 la lista Top500 contenía 25 sistemas conectados a través de Infiniband HDR, que alcanza velocidades de enlace por canal de 50 Gbps; y 61 sistemas con Infiniband EDR, que únicamente alcanza 25 Gbps [71]. Mientras que entre estos dos resultados transcurren cuatro años de diferencia, el rendimiento que se ha mejorado de un caso a otro es, únicamente, del doble. Por tanto, no solo debe tenerse en cuenta el hecho de que actualmente las comunicaciones son un cuello de botella, sino que, atendiendo a los datos actuales, si la tendencia continúa, las interconexiones serán, cada vez más, el componente que limite el rendimiento. Así pues, la optimización de las interconexiones y la reducción del coste de las comunicaciones puede ser una parte clave en el futuro de las aplicaciones distribuidas, más aún si se tiene en consideración que las comunicaciones también son una parte significativa del consumo de energía.

De entre las múltiples primitivas implementadas en el estándar MPI, las comunicaciones colectivas de reducción son unas de las más utilizadas; es por ello que el estudio y análisis de la comunicación *AllReduce* realizado en este capítulo resulta de interés más allá del campo de las redes neuronales.

5.2. Análisis del Coste de la Primitiva *AllReduce*

La primitiva de comunicación colectiva *AllReduce*, como se ha mencionado previamente en el Capítulo 4, puede implementarse utilizando diferentes algoritmos, siendo algunos de los más populares los algoritmos explicados en la Sección 4.3. Cada uno de estos algoritmos sigue un esquema diferente para completar la comunicación y, por tanto, el coste variará en función del algoritmo que la implemente y de las características de los factores involucrados: el mensaje y la red de comunicación del clúster. Como punto inicial de este análisis, se ha estudiado cómo estas características influyen en el coste de la comunicación y, en consecuencia, en su rendimiento medido en MiB (tamaño del mensaje) por unidad de tiempo (segundos). Para ello se ha estimado el tiempo que costaría realizar dicha comunicación en un cierto clúster teniendo en cuenta los modelos detallados en la Tabla 4.8 para cada algoritmo. En la Figura 5.1 se representan cuatro gráficas

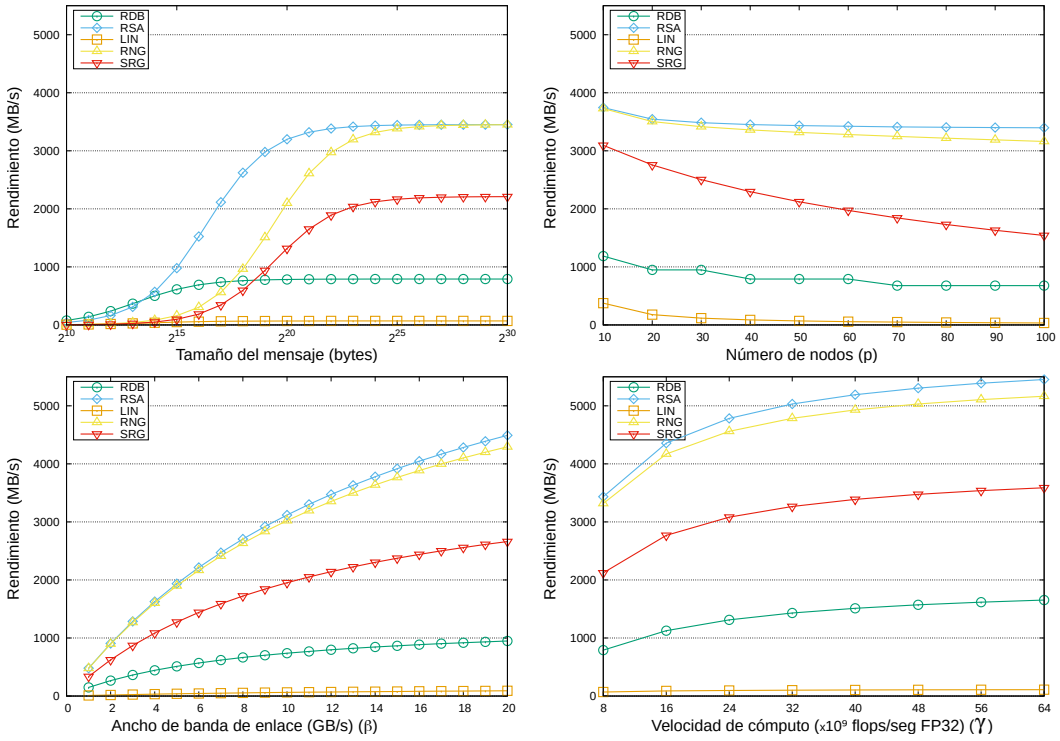

Figura 5.1: Rendimientos teóricos de la primitiva *AllReduce*.

que muestran cómo varía el rendimiento de la comunicación en función de cuatro parámetros: tamaño del mensaje en bytes, número de nodos que participan, ancho de banda de enlace del clúster y velocidad de cómputo. Estos valores se han obtenido tomando como parámetros base para el modelo: $\alpha = 2\mu s$, $\beta = 11,770$ MiB/s, $p = 50$ nodos, $n = 16 \cdot 2^{20}$ bytes y $\gamma = 8 \cdot 10^9$ flops/s FP32. Además, en el caso del algoritmo SRG, se ha tomado como valor de segmentación $s = 64$.

Observando los resultados, destacan los algoritmos RNG y RSA como los que ofrecen mejores resultados al estudiar, tanto la cantidad de nodos, como el ancho de banda de enlace y la velocidad de cómputo. En las gráficas, sin embargo, el factor que más influye en el rendimiento de la primitiva *AllReduce* a la hora de escoger qué algoritmo utilizar, es el tamaño del mensaje a transmitir, en función del cual, puede ser más conveniente escoger un algoritmo u otro. En base a estos resultados, se estudia a continuación, con mayor profundidad, el coste de cada algoritmo en función del tamaño del mensaje. Para ello, se ha analizado el coste real de esta comunicación para cada algoritmo midiendo experimentalmente el tiempo de ejecución al utilizar diferentes tamaños de mensajes.

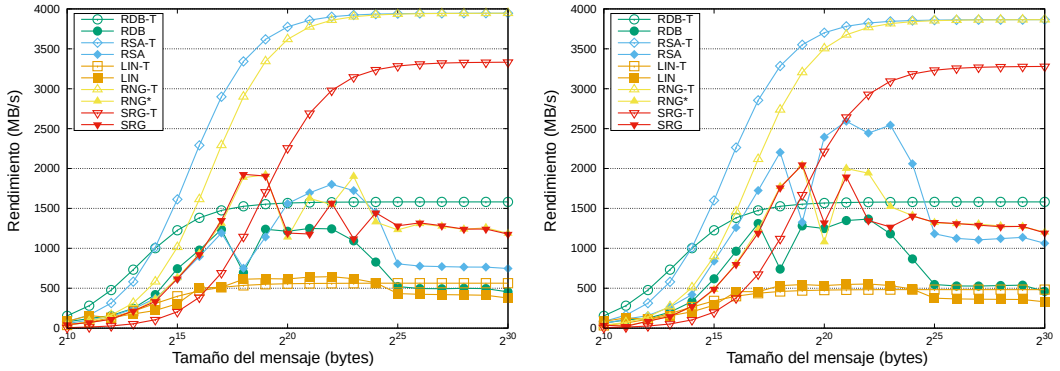

Figura 5.2: Rendimiento real y teórico (-T) de la primitiva *AllReduce* en Open-MPI utilizando 7 y 8 nodos (izquierda y derecha respectivamente).

La Figura 5.2 representa, junto a los rendimientos teóricos vistos, los rendimientos reales alcanzados al utilizar la biblioteca Open-MPI. Se evidencia en dichas gráficas que hay una desviación significativa entre el coste efectivo de la comunicación y el estimado por los modelos teóricos, por lo que la decisión del algoritmo óptimo para cada caso no debería centrarse únicamente en la estimación

teórica, sino que requiere de un estudio en mayor profundidad. A partir de los resultados experimentales, se puede observar que el algoritmo RSA, tan prometedor en base a los modelos teóricos y que es elegido como opción de referencia en gran parte de las bibliotecas MPI, resulta no ser la opción óptima al considerar las mediciones reales en gran parte de los casos analizados. Esto ocurre especialmente en el caso de usar un número de procesos que no es potencia de 2 y en la transmisión de mensajes de más de 2^{25} bytes. Destaca, sin embargo, junto al algoritmo RNG, el uso del algoritmo SRG que en un principio no se mostraba tan favorable en el análisis de los modelos teóricos. Finalmente, se observa que en el estudio experimental resulta un factor significativo para la elección del algoritmo, además del tamaño de mensajes, el número de procesos que intervienen.

5.2.1. Estudio de las posibles causas

Si bien los modelos teóricos ofrecen una predicción aproximada del rendimiento de los esquemas de comunicación colectiva, la diferencia que existe entre esta estimación y su coste real sigue siendo considerable, tal y como se ha mostrado previamente. Este error se debe a la diferencia entre aquello que el computador debería hacer y aquello que realmente hace cuando realiza una comunicación. Para justificar esta diferencia, se han considerado en este trabajo las siguientes razones como posibles causas asociadas al modelo teórico:

- Asumir, para la estimación del tiempo, que el ancho de banda de enlace y de memoria alcanzan continuamente el rendimiento máximo con independencia de otras variables, como por ejemplo el número de bytes transmitidos.

- Considerar que las llamadas a comunicaciones dentro de un proceso, tal como la combinación de `iRecv` seguido de una llamada `Send`, consiguen en la práctica una superposición perfecta; es decir, como si en el ejemplo mencionado el coste de ambas llamadas se redujera al coste la primitiva `Send`, asumiendo una ejecución completamente simultánea.

- No considerar las posibles implicaciones de las características del equipo, tanto del software, como del hardware; como puede ser el caso, por ejemplo,

de los búferes de las implementaciones subyacentes: qué cantidad hay y qué tamaño tienen.

De estas posibles causas, se muestra a continuación un mayor análisis de las dos primeras.

Ancho de banda de enlace y de memoria. Dos de los valores clave para estimar el coste de los movimientos de datos son el ancho de banda de enlace y el ancho de banda de memoria, determinantes de la velocidad y, en consecuencia, del tiempo de ejecución de las comunicaciones. El valor de ambos parámetros que se especifica en la información del hardware no ofrece una velocidad práctica de estas variables, sino su máximo alcanzable. Por tanto, atender a estos valores para aproximar los costes de realizar ciertos movimientos de datos, conllevaría una infravaloración de estos. Para evitar la sobreestimación de dichas velocidades, un modo de obtener valores más aproximados consiste en realizar una batería de experimentos que nos muestre el tiempo real y nos permita estimar en base a estos la velocidad efectiva que se puede alcanzar en la práctica. Así, por ejemplo, para aproximar el ancho de banda de enlace, un conjunto de pruebas de tipo ping-pong nos pueden ofrecer dichos tiempos y, por ende, las velocidades. La Figura 5.3

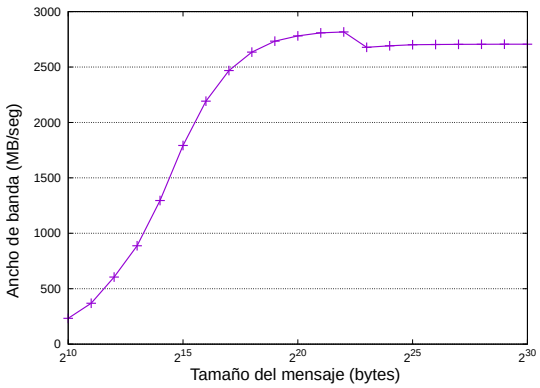

Figura 5.3: Rendimiento punto a punto de una red medido mediante una prueba *ping-pong*.

representa una muestra de estas estimaciones. En dicha gráfica puede apreciarse que la velocidad del ancho de banda de enlace no es siempre constante, sino que

varía en función de la cantidad de bytes que se transmiten y que, por este motivo, para estimar este parámetro con un valor más preciso debería contemplarse el tamaño del mensaje que se transmite. Como se mostrará más adelante, en el caso del ancho de banda de memoria ocurre algo similar, de modo que su valor también difiere en función de la cantidad de bytes con los que se está operando. En consecuencia, dado que el valor escogido para estos parámetros repercute significativamente en la estimación del coste final de la comunicación, deberá tenerse en cuenta la cantidad de bytes con los que se está trabajando, teniendo en cuenta para ello el proceso que sigue el algoritmo mediante el que se está llevando a cabo la comunicación. Esta consideración será más evidente en algunos casos en los que los intercambios de datos siempre son del mismo tamaño, como es el caso del algoritmo LIN, que hace siempre transmisiones del mensaje completo. Sin embargo, en otros algoritmos, como el caso de RSA, se transmiten únicamente fracciones del mensaje, siendo estas fracciones en el caso del RSA de diferente tamaño a lo largo de la comunicación, por lo que la comunicación podría no realizarse a una velocidad constante.

Asincronía. Otro factor que se asocia como causa de la brecha entre el coste teórico y el real de la primitiva *AllReduce* es la sincronía alcanzable en las comunicaciones. Si se analizan las funciones internas de las bibliotecas que realizan la comunicación colectiva, puede observarse que en todos los algoritmos se emplean rutinas MPI no bloqueantes, como `iRecv`, con la finalidad de solapar envíos y recepciones. Este tipo de rutinas se utilizan, tanto en intercambios entre pares (p_0 envía a p_1 a la vez que este último le envía al primero) como en intercambios en cadena (p_1 recibe un mensaje de p_0 a la vez que envía un mensaje a p_2). Esto ocurre, por ejemplo, en el caso de las implementaciones en la biblioteca Open-MPI de los algoritmos RDB, RSA y RNG respectivamente. Al comparar estas funciones con los modelos teóricos expuestos en la Tabla 4.8, se observa que estos asumen que las rutinas MPI no bloqueantes utilizadas durante la ejecución de la comunicación *AllReduce* son capaces de realizar un solapamiento completo. De ese modo, se asume que, tanto en los intercambios entre pares, como en los intercambios en cadena, el envío y la recepción de datos se realizan simultáneamente. No obstante, en la práctica, no siempre es posible alcanzar dicha sincronía. Para

exponer la inexactitud de esta hipótesis, en la Figura 5.4 se muestra el tiempo de ejecución estimado de los algoritmos mencionados previamente, RDB, RSA y RNG, si las comunicaciones se contabilizarán como totalmente asíncronas y si lo fueran como síncronas, junto con el coste real de ejecutar estos algoritmos. Puede apreciarse en estos experimentos que el coste real no alcanza el supuesto asíncrono, quedando más cerca del tiempo estimado mediante el modelo síncrono en los tres casos estudiados. Por tanto, pese a estar implementados los algoritmos con comunicaciones no bloqueantes, puede intuirse de estos resultados que se alcanza una sincronía parcial que permite ejecutar a la vez ciertas comunicaciones y cálculos, pero sin llegar a lograr el solapamiento completo deseado.

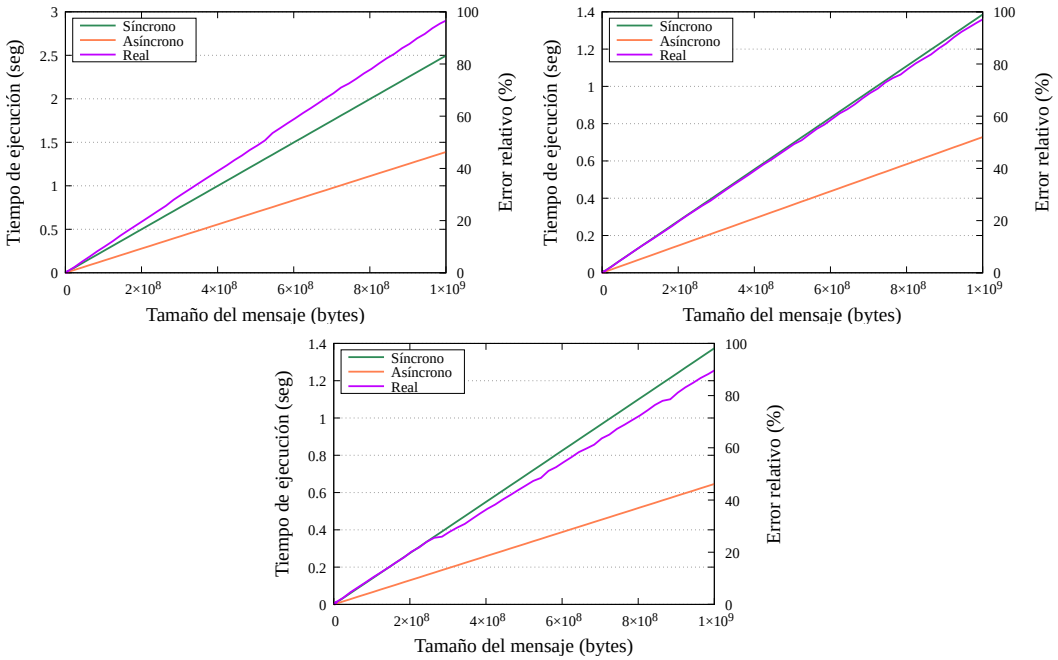

Figura 5.4: Tiempos de ejecución y estimaciones suponiendo comunicaciones síncronas y asíncronas para los algoritmos RDB, RSA y RNG de la primitiva *All-Reduce*.

5.3. Mejora de las Estimaciones

El análisis realizado en la sección anterior pone de manifiesto la brecha existente entre el coste real de la comunicación *AllReduce* y el coste que los modelos teóricos estiman (ver Tabla 4.8). Además expone algunas de las posibles causas por las que estos costes difieren. Tomando como punto de partida dicho análisis, en esta sección se muestra el proceso mediante el que se han abordado las causas mencionadas, obteniendo nuevos métodos de estimación que permitan aproximar con mayor precisión el coste de la primitiva *AllReduce*. En este estudio nos hemos centrado en la biblioteca Open-MPI y en los algoritmos LIN, RDB, RNG y RSA que esta implementa. El tiempo de ejecución de la comunicación depende de múltiples factores y, en consecuencia, de la aproximación que se haga de este. Cuantas más variables se tengan en cuenta, más precisión se obtendrá en la estimación, pero más complejo será el modelo y, por ende, su uso. Por tanto, debe valorarse el equilibrio que desea alcanzarse entre estos dos términos. El modelo por el que se ha optado en este caso contempla las dos causas mencionadas en la sección previa, es decir, la asincronía de las comunicaciones y la velocidad de los parámetros del ancho de banda de enlace y de memoria. No obstante, omite o generaliza otras características, pues se aspira a diseñar un método de aproximación de los costes que reduzca el error de estimación, manteniendo un método de estimación sencillo. Así, por ejemplo, no se valorarán los casos en los que el mensaje sea de un tamaño tal que la duración total de la comunicación no suponga un coste significativo en el cómputo global.

Las siguientes secciones abordan cada uno de los factores en los que se desea reducir el error de aproximación. Una vez analizados todos ellos, se evalúan los resultados alcanzados. Todos los datos experimentales se han obtenido utilizando un clúster de 16 nodos, cada uno de ellos con dos procesadores Intel Xeon E5645 hexa-core (2.40 GHz), conectados a través de una red Infiniband QDR (Mellanox MTS3600). Los experimentos se han realizado asignando un único proceso MPI por nodo.

5.3.1. Ancho de banda de enlace

Para obtener la velocidad que alcanza el ancho de banda de enlace, debe diferenciarse en la velocidad de envío de los mensajes la parte correspondiente a la latencia, eliminando de las mediciones temporales la fracción de tiempo asociada a esta. La obtención del tiempo de latencia se ha aproximado mediante mediciones de comunicaciones con mensajes de tamaños insignificantes. Así pues, se han restado de los tiempos de comunicación medidos la unidad de tiempo aproximada para la latencia y, a partir de los tiempos restantes, se han calculado las velocidades representadas en la gráfica de la Figura 5.5. De esta figura se observa que el ancho de banda de enlace, para este caso, empieza a estabilizar su valor para mensajes de tamaño superior a 2^{18} bytes. Como se ha mencionado previamente, en nuestro caso nos centraremos en minimizar los errores de mensajes cuyo coste sea significativo, por lo que en adelante no se tendrá en cuenta la variación del ancho de banda de enlace que afecta a los mensajes de tamaños inferiores a 2^{18} y se tomará la velocidad alcanzada al estabilizarse (v_{Est}) como el valor constante de β, independientemente de los bytes transmitidos. Si se trabajara potencialmente con mensajes de estos tamaños, sería relevante aproximar la curva inicial que se observa en la gráfica. En este caso, una opción sencilla consiste en utilizar una función de velocidad

$$\beta(n) = v_{Est}\,\frac{n+c}{n},$$

adaptando la constante c según las circunstancias. De este modo, se aproxima la velocidad de transmisión en función de la cantidad de bytes (n) que se transmiten.

5.3.2. Ancho de banda de memoria

Durante la ejecución de la primitiva *AllReduce*, independientemente del algoritmo escogido, además de realizar intercambios de datos entre los procesos, también se efectúa cómputo. En el caso que atañe al entrenamiento de las redes neuronales, la operación de reducción que se realiza es la suma. Por tanto, durante la ejecución, los procesos realizan sumas con los datos que se están intercambiando y, por las características de esta operación, el tiempo de ejecución de estas operaciones se ve limitado por el coste de acceso a memoria. Por tanto, el coste temporal de la parte de cómputo que interviene en la primitiva *AllReduce*

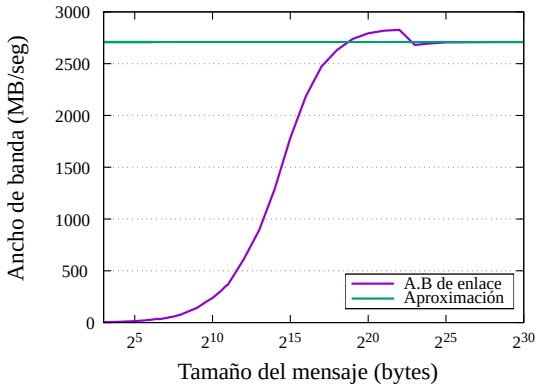

Figura 5.5: Ancho de banda de enlace alcanzado en experimentos *ping-pong*.

vendrá definido por la velocidad de acceso a los datos operados, es decir, por el ancho de banda de memoria γ.

La metodología empleada para estimar la velocidad de acceso a memoria en este estudio es análoga a la utilizada en el caso del ancho de banda de enlace: calcular, para diferentes tamaños de operandos (vectores de diferentes longitudes), el tiempo promedio al realizar múltiples mediciones de sumas. Con el fin de simular con precisión el cálculo que genera una llamada *AllReduce* de Open-MPI y obtener unos tiempos comparables a los reales, las sumas de vectores medidas en las pruebas se han realizado utilizando un único proceso. La Figura 5.6 recoge

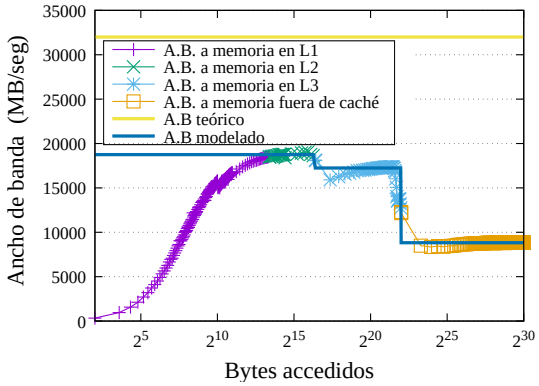

Figura 5.6: Ancho de banda de memoria alcanzado en experimentos de suma de vectores.

los valores del ancho de banda estimados según los resultados obtenidos. Nótese que la velocidad marcada por el ancho de banda de enlace se corresponde con los bytes accedidos y, por tanto, debe tenerse en cuenta que, para cada suma, se accede a 3 elementos en memoria (dos operandos y el resultado). Por tanto, la velocidad que se alcanzará en una operación con operandos de n bytes (γ_n) se obtiene a partir del ancho de banda referente al acceso de $3n$ bytes, en adelante denominado AB_{3n}. Considerando, además, que al acceder al triple de elementos la velocidad se reducirá a un tercio, el valor de γ_n queda como $3\frac{1}{AB_{3n}}$ s/byte computado. Puede advertirse en la gráfica que, para una cantidad pequeña de bytes, la velocidad alcanzada dibuja una curva ascendente, igual que ocurría con el ancho de banda de enlace. Sin embargo, una vez alcanzado el punto de inflexión, en este caso la velocidad no se estabiliza, sino que el ancho de banda de acceso a memoria sigue variando a medida que se incrementa el número de bytes, dibujando dos caídas a partir de ese punto y estabilizándose, finalmente, tras esta última. En la Figura 5.6 se muestra, mediante diferenciación por colores, la relación existente entre estos cuatro niveles, definidos por el crecimiento inicial y los descensos mencionados, con la jerarquía de memoria del clúster. Puede apreciarse de este modo 1) la relación existente entre la caché L1 (barras violetas) y el ancho de banda de memoria creciente para tamaños de mensajes pequeños; 2) la caché L2 (cruces verdes) y el punto de inflexión; la 3) la caché L3 (estrellas azules) y la sección intermedia entre ambos descensos y, por último; y 4) la memoria principal (cuadrados naranja) y el valor de estabilización final. Esta misma relación se ha encontrado también al realizar el estudio sobre el clúster utilizado en la Sección 4.5.1, por lo que se consideró válido aproximar el valor de γ en base a estas relaciones. Mediante una línea azul añil, se ha representado en dicha gráfica la estimación del ancho de banda de memoria. Dado que no se pretende modelar el rendimiento para tamaños reducidos, en estos casos se vuelve a optar por generalizar el ancho de banda al valor más cercano, de modo que la curva correspondiente al nivel de memoria en caché L1 se estima del mismo modo que L2. En caso de que para el estudio las comunicaciones pequeñas fueran relevantes, podría optarse por una estimación por función semejante a la mencionada en la sección anterior. Pese a que esta aproximación no considera el caso de tamaños

pequeños ni los valores intermedios que pueden apreciarse en los límites de los niveles, resulta una estimación que reduce significativamente la diferencia respecto de la velocidad real de acceso a memoria, sin añadir complejidades relevantes en el modelo.

5.3.3. Asincronía

La implementación interna de los diferentes algoritmos *AllReduce* emplean rutinas no bloqueantes cuando se realizan intercambios de datos a fin de reducir el coste de estas transmisiones. Pese a ello, como se ha mostrado previamente (Sección 5.2.1), en la práctica no se logra la sincronía esperada. Dada la dificultad de analizar qué partes de las rutinas adquieren esa superposición y cuáles no, el modo de estudiar este factor se ha basado en generar modelos teóricos que, contemplando las diferentes opciones de sincronía, permitan esclarecer qué fenómeno tiene más probabilidades de ocurrir (solapamiento o no) y en qué grado. Para ello, se toman como versiones asíncronas los modelos presentados en la Tabla 4.8, los cuales presuponen que los intercambios (entre pares y en cadena) se solapan completamente, efectuándose los dos envíos y las recepciones al mismo tiempo. Como versión síncrona, se definen además los nuevos modelos teóricos que asumen el extremo opuesto, es decir, contabilizando los intercambios de datos como dos envíos consecutivos. Estos modelos se establecen de manera análoga a los ya estudiados, con las únicas modificaciones encargadas de contabilizar los intercambios de datos por duplicado. En consecuencia, de los algoritmos con los que se está trabajando –LIN, RDB, RNG y RSA– los tres últimos repetirían la parte del coste relativo a la comunicación doblando, simplemente, el coeficiente de la latencia y el ancho de banda de enlace (α y β). El algoritmo LIN, por contra, no realiza intercambio de datos entre procesos, dado que su procedimiento consiste en el envío desde un único proceso al resto. Por ello, el modelo original no consideraba la asincronía de dichos envíos y, para este caso, el modelo síncrono resulta equivalente al asíncrono.

Por otro lado, atendiendo a las consideraciones que se han propuesto para la estimación del ancho de banda de memoria y de la velocidad de cómputo (diferenciando en función del número de bytes), el algoritmo RSA precisa replantear el

coste de su cómputo. Dado que durante la ejecución de este algoritmo el tamaño del mensaje con el que se trabaja va cambiando, la velocidad a la que se realizan las sumas parciales y, en consecuencia, el valor γ empleado para su estimación también variará ajustándose a la cantidad de bytes que se están operando. De este modo, el coste temporal relativo al cómputo del algoritmo RSA se ajustará como

$$4\log_2(p)\alpha + 4n\frac{p-1}{p}\beta +$$
$$n(\gamma_{FueraCache}\frac{a-1}{a} + \gamma_{L3}(\frac{1}{a}-\frac{1}{b}) + \gamma_{L2}(\frac{1}{b}-\frac{1}{p})),$$

donde $a = 2^{\text{máx}(0,\text{mín}(\lfloor\log_2 \frac{bytes-1}{capacidad_{L3}}\rfloor),\log_2 p)}$ y b es análogo para la capacidad de la caché L2.

Finalmente, nótese que estos modelos teóricos están diseñados para estimar el coste cuando el número de procesos involucrados en la comunicación (p) es potencia de dos. De lo contrario, los algoritmos RSA y RDB, al tener un procedimiento basado en el método divide y vencerás, requerirán una iteración inicial y otra final que reduzca el problema a un número de procesos potencia de dos. Como consecuencia de este trabajo previo, el coste estimado por el modelo deberá incrementarse en

$$2\alpha + 2n\beta + n\gamma,$$

correspondiente a dos envíos y una operación de adición del mensaje inicial (tamaño completo). Además, deberá sustituirse el valor exacto del logaritmo por la parte entera de este, es decir, $\lfloor\log_2(p)\rfloor$.

5.3.4. Resultados

Para poder observar el efecto de aplicar las consideraciones definidas en las secciones anteriores (ver Secciones 5.3.1, 5.3.2 y 5.3.3), se han obtenido los tiempos estimados en función de estas. Por un lado, se han calculado las aproximaciones utilizando la aplicación de γ adaptado (GA) a la jerarquía de memoria sobre el modelado asíncrono (MA), es decir, las ecuaciones definidas en los modelos teóricos de la Tabla 4.8. Por otra parte, se han obtenido las aproximaciones ofrecidas al combinar el uso del γ adaptado (GA) con el modelado síncrono (MS) diseñado,

de manera que pueda apreciarse la aproximación final del coste estimada según las dos consideraciones descritas. Finalmente, para poder comparar con el punto de referencia inicial, se han calculado los tiempos de estimación originales mediante el uso de los MA y sin utilizar la nueva versión del γ, esto es, haciendo uso del valor constante que se presupone habitualmente (Gamma constante o GC).

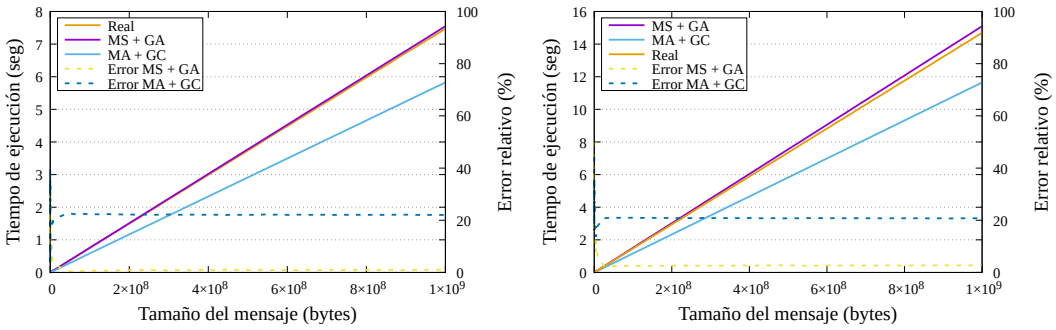

Figura 5.7: Tiempos de ejecución y errores relativos para el algoritmo LIN con 8 y 15 procesos.

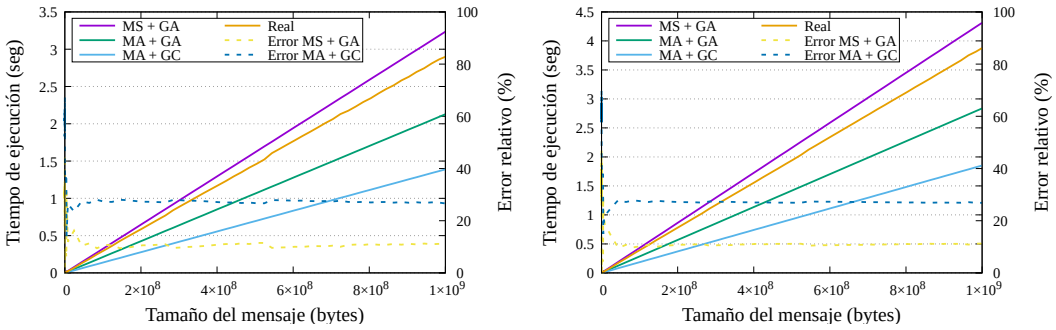

Figura 5.8: Tiempos de ejecución y errores relativos para el algoritmo RDB con 8 y 15 procesos.

Al representar gráficamente estos tiempos estimados, junto con el coste real de la ejecución de la comunicación *AllReduce* según lo representado en las Figuras 5.7, 5.8, 5.9 y 5.10, se observa que:

- En el caso del algoritmo LIN, dado que el modelo original ya contemplaba comunicaciones síncronas, solo se ha aplicado el uso de γ adaptada. El

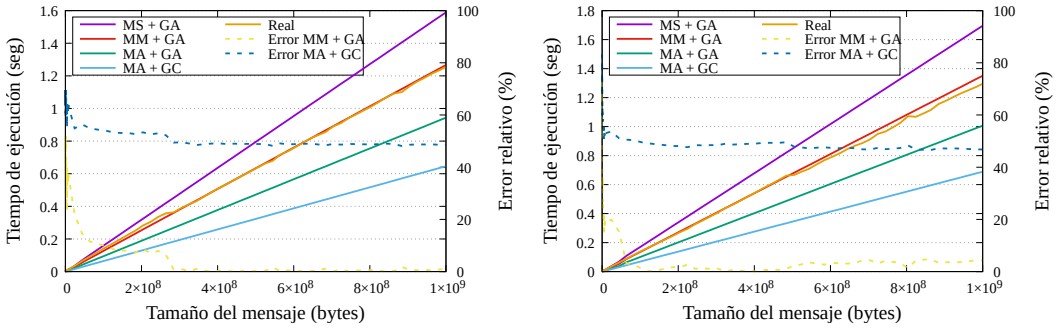

Figura 5.9: Tiempos de ejecución y errores relativos para el algoritmo RNG con 8 y 15 procesos.

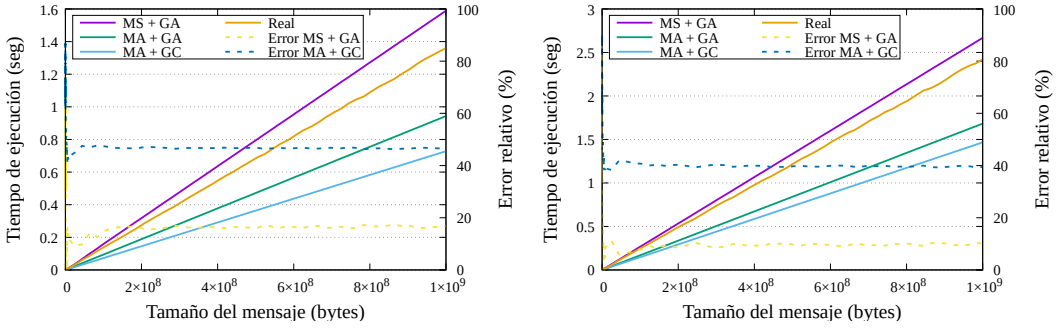

Figura 5.10: Tiempos de ejecución y errores relativos para el algoritmo RSA con 8 y 15 procesos.

tiempo estimado difiere muy poco del tiempo real, por lo que puede valorarse que la adaptación del ancho de banda de memoria y de la velocidad de cómputo ofrece unos buenos resultados.

- Respecto de la asincronía alcanzada por los algoritmos, todas las gráficas muestran que no se consigue la superposición completa de comunicaciones que se presupone en los modelos teóricos originales. Los costes reales medidos no coinciden con el modelo teórico asíncrono, quedando, en general, más cercanos al modelo síncrono que se ha diseñado.

- El algoritmo RNG mantiene su tiempo real equidistante del modelo síncrono y el asíncrono. De los algoritmos analizados, este es el único caso en el que el modelo síncrono no es el más cercano al tiempo real, dado que se alcan-

za dicha posición intermedia. Esta mayor superposición de comunicaciones puede deberse a que, en este algoritmo, los intercambios de datos no se realizan entre dos procesos (el proceso p_i envía a p_j y viceversa), sino que se realizan intercambios en cadena (p_i envía a p_j mientras recibe los datos de p_k). Teniendo en cuenta esta semi-asincronía, el tiempo estimado de ejecución para este algoritmo puede aproximarse mediante una ecuación que combine un modelado mixto (MM) –síncrono y asíncrono– y que se ha definido mediante una media de los coeficientes que tomaban ambos modelos MS y MA:

$$3\log_2(p)\alpha + 3n\log_2(p)\beta + n\log_2(p)\gamma,$$

y que se incluye también en la Figura 5.9.

Estas figuras muestran también los errores relativos que se cometen (representados con líneas discontinuas), tanto con la aproximación original (MA+GC) como con las estimaciones diseñadas (MS/MM+GA). La Tabla 5.1 presenta un valor representativo de estos errores, así como del error que se comete al aplicar únicamente el factor γ variable (MA+GA). Estos datos reflejan el efecto positivo de las modelizaciones diseñadas. El hecho de adaptar la estimación de la velocidad de cómputo en función del tamaño del mensaje consigue reducir el error relativo entre un 10 y un 25 %. La combinación de esta mejora junto con el uso de modelos no asíncronos disminuye significativamente el error. En los casos en los que se ha adoptado un modelo completamente síncrono, se ha reducido el error relativo un 30 %, alcanzando errores de un 10–17 %. Con respecto a los valores del algoritmo RNG, el hecho de considerar un MM resulta en un decremento aún mayor, quedándose un error de un 4 %. Por tanto, utilizar modelos híbridos puede considerarse una buena estrategia para estimar los tiempos de la primitiva *AllReduce*. Finalmente, es importante resaltar que los resultados mostrados en esta sección para 8 y 15 procesos, han sido contrastados también con experimentos análogos en los que participaban 4, 5 y 12 procesos, obteniendo resultados similares.

Tabla 5.1: Errores relativos respecto del coste real cuando la tendencia se estabiliza.

Algoritmo	MA + GC	MA + GA	MS/MM + GA
LIN	21 %	-	2 %
RDB	52 %	27 %	11 %
RNG	47 %	24 %	4 %
RSA ($p = 2^x$)	40 %	30 %	17 %
RSA ($p \neq 2^x$)	46 %	30 %	4 %

5.4. Elección del Algoritmo

A lo largo de esta sección se analiza el efecto de utilizar diferentes algoritmos para la implementación de la primitiva *AllReduce* con el objetivo de enfatizar la relevancia de una correcta selección del algoritmo. Para ello se estudian experimentalmente los algoritmos que ofrecen mejores prestaciones para cada caso y se comparan con la selección que las bibliotecas MPI realizan automáticamente, cuando el usuario no establece el uso de un algoritmo en concreto. En adelante, nos referiremos a esta configuración como AUTO. En las pruebas expuestas en esta sección se evalúan los algoritmos *AllReduce* disponibles en las tres bibliotecas MPI que se estudian en este capítulo, siendo estos algoritmos los siguientes:

- MPICH*: RDB y RSA.
- Open-MPI†: RDB, RSA, LIN, RNG y SRG
- Intel-MPI‡:RDB, RSA, RNG, RB, TA-RB, BGS, TA-BGS, SHR, KNO, TA-SHM-BF, TA-SHM-KNO y TA-SHM-KNA§.

*MPICH 3.3.1, https://www.mpich.org/

†Open MPI 4.1, https://www.open-mpi.org/

‡IntelMPI 2020, https://www.intel.com/content/www/us/en/developer/tools/oneapi/mpi-library.html

§Estos últimos nueve algoritmos, no trabajados anteriormente, corresponden a las siglas inglesas referentes a las siguientes variantes de *AllReduce* respectivamente:Reduce+Bcast, To-

Todos los datos expuestos en esta sección se han obtenido utilizando el clúster descrito en la sección 4.5.1, es decir, un sistema compuesto por 8 nodos equipados con un procesador Intel Xeon Gold 5120 y con una red de interconexión Infiniband EDR. El procedimiento para calcular las velocidades es análogo al de las secciones previas, basándose en mediciones experimentales. En cuanto a las especificaciones con las que se han ejecutado los experimentos, en todos ellos se ha utilizado un único proceso MPI por nodo. En los experimentos relativos al algoritmo SRG, este se aplica del mismo modo que el algoritmo RNG para mensajes cuyo tamaño es inferior a 8 MiB y para tamaños superiores, realiza segmentaciones de 1 MiB. Finalmente, en aquellos experimentos de la biblioteca Intel-MPI en los que se utiliza un parámetro de segmentación, se ha asignado a este el valor 64.

5.4.1. Evaluación de los diferentes algoritmos

Este primer estudio sobre la elección del algoritmo que implementa el *AllReduce* tiene como finalidad comparar experimentalmente, para cada biblioteca, los rendimientos alcanzados por cada algoritmo disponible, así como los alcanzados por la elección seleccionada automáticamente por la biblioteca. La Figura 5.11 muestra los rendimientos obtenidos experimentalmente para cada biblioteca utilizando 7 y 8 procesos. En estos resultados, puede identificarse qué algoritmo selecciona la biblioteca y en qué casos existe una opción cuyo rendimiento es mejor al seleccionado. Algunas observaciones a destacar son:

- La biblioteca MPICH tiende a seleccionar el algoritmo RDB como opción por defecto. No obstante, en ambos casos el algoritmo RSA tiende a ofrecer rendimientos superiores, siendo la diferencia significativa cuando se están comunicando mensajes de tamaño superior a 2^{17} bytes.

- En el caso de Open-MPI, RSA es el algoritmo mayormente seleccionado por la biblioteca. Esta vez, la opción por defecto sí que coincide parcialmente con el algoritmo que mejor rendimiento ofrece, a excepción de algunos casos,

pology Aware Reduce+Bcast, Binomial Gather+Scatter, Shumilin's Ring, Knomial, Topology Aware SHM-Based Flat, Topology Aware SHM-based Knomial y Topology Aware SHM-based Knary.

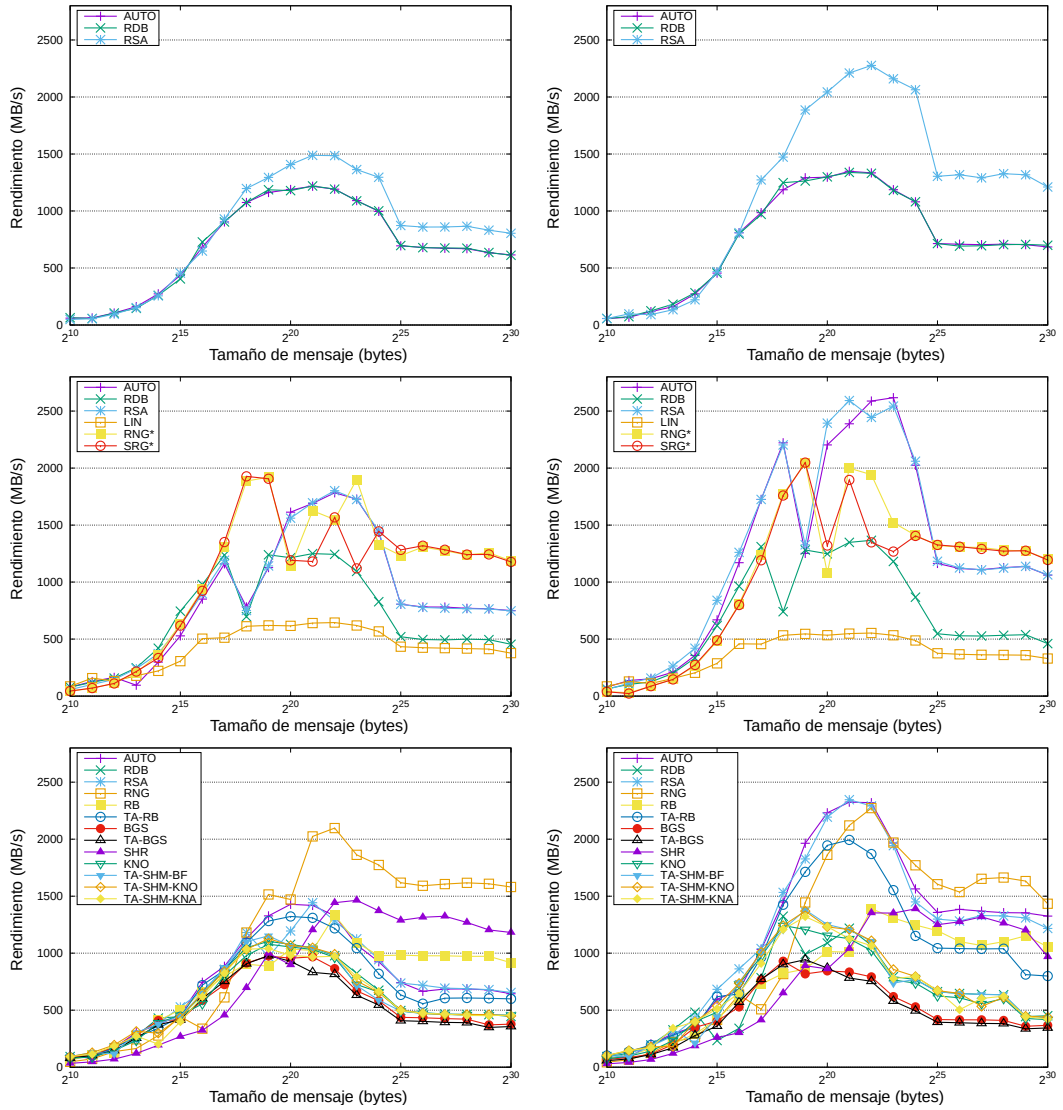

Figura 5.11: Rendimientos de los algoritmos *AllReduce* disponibles en MPICH (arriba), Open-MPI (centro) e Intel-MPI (abajo), utilizando 7 y 8 procesos (izquierda y derecha, respectivamente).

$2^{[17,19]}$ para 7 procesos y $2^{[25,]}$ para 8, en los que algoritmos de anillo, RNG y SRG, tienen rendimientos superiores.

- En Intel-MPI, el algoritmo RSA vuelve a ser uno de los más seleccionados. En este caso los resultados difieren en función de si el número de procesos involucrados es potencia de dos o no. Cuando el número de procesos es

8, el algoritmo por defecto seleccionado por la biblioteca ofrece un buen rendimiento en parte de los casos. Sin embargo, no se realiza una buena selección para los experimentos en los que el número de procesos es 7. En ambos casos, el algoritmo RNG ofrece mejores resultados que el algoritmo seleccionado para tamaños medianos y grandes de mensaje: en el caso de 8 procesos, puede verse este efecto en los tamaños superiores a 2^{24}, mientras que en el caso de no utilizar un número de procesos potencia de dos, se observa a partir de tamaños superiores a 2^{18}.

Nótese que en la Sección 5.3.3, el análisis relativo a la asincronía desmiente el supuesto solapamiento completo que asumen los modelos teóricos y muestra que el algoritmo basado en anillo es capaz de alcanzar una mayor asincronía en las comunicaciones por el tipo de procedimiento que sigue. Así pues, la infravaloración de los algoritmos RNG y SRG que se aprecia en el análisis de estas gráficas, se corresponde con la sobreestimación de la capacidad de asincronía que se le asignan a otros algoritmos, como RSA, que no consigue alcanzarse en la práctica.

5.4.2. Evaluación de las diferentes bibliotecas

Una vez analizados los resultados para cada biblioteca por separado, resulta interesante relacionar los rendimientos ofrecidos por estas entre sí. La Figura 5.12

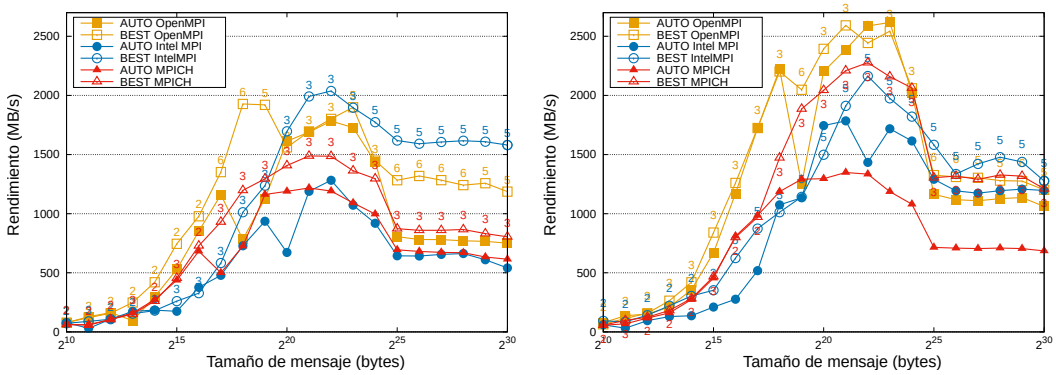

Figura 5.12: Rendimiento de los algoritmos que mejor rendimiento ofrecen para cada caso (BEST) y del escogido por defecto (AUTO) por la biblioteca, utilizando Open-MPI, MPICH e Intel-MPI y comparando el uso de 7 y 8 procesos (izquierda y derecha respectivamente).

permite comparar los valores que cada biblioteca obtiene, tanto al realizar la elección automática del algoritmo, como al utilizar en cada caso el algoritmo que mejor rendimiento ofrece (etiquetado en este caso como BEST). También se ha indicado mediante números el tipo de algoritmo que se corresponde con el algoritmo que mejor rendimiento ofrece, correspondiéndose como: 2 - RDB, 3 - RSA, 4 - LIN, 5 - RNG, 6 - SRG. De los resultados que se observan en las gráficas, destacan las siguientes reflexiones:

- Para tamaños pequeños ($[1, 2^{16}]$) la biblioteca que mejor selecciona la opción por defecto entre los posibles algoritmos implementados es MPICH.

- En los casos en los que interviene un número de procesos que no es potencia de dos, ninguna biblioteca realiza una buena selección del algoritmo. Existe una gran diferencia entre la mejor opción y la opción elegida por la biblioteca Intel-MPI para gran parte de los tamaños ($2^{[18,)}$). Este intervalo también es crítico para MPICH, aunque en bastante menor medida. En el caso de Open-MPI, existe un pequeño espacio en los tamaños intermedios ($2^{[20,24]}$) en los que AUTO resulta una buena opción, pero para tamaños mayores y menores de mensaje esta opción obtiene un rendimiento muy por debajo del mejor algoritmo disponible.

- Para los casos donde el número de procesos participantes sí es potencia de dos, los resultados de Open-MPI e Intel-MPI mejoran, reduciendo significativamente la distancia entre la opción AUTO y la BEST. Sin embargo, en la biblioteca MPICH no ocurre lo mismo. Mientras que para tamaños pequeños la opción por defecto sigue siendo una buena decisión, en el resto de casos se incrementa drásticamente la brecha entre AUTO y BEST, al seleccionar algoritmos que ofrecen un rendimiento mucho menor al que se podría alcanzar si se realizara una buena selección.

- La biblioteca MPICH es la que menor rendimiento ofrece en estos resultados, aunque una mejor selección del algoritmo por defecto superaría para un gran intervalo de tamaños las prestaciones obtenidas por la biblioteca Intel-MPI para los casos en los que participan un número de procesos potencia de dos.

- Las bibliotecas Open-MPI e Intel-MPI tienen un gran margen de mejora en el rendimiento alcanzable cuando interviene un número de procesos que no es potencia de dos. En particular, para mensajes de gran tamaño Intel-MPI es capaz de obtener rendimientos significativamente superiores al resto de bibliotecas analizadas, siendo ahora, la opción por defecto que menos rendimiento alcanza.

5.4.3. Impacto sobre el entrenamiento de RNP

En esta sección se realiza un último análisis sobre los algoritmos de comunicación para observar cómo afecta la elección de estos en el campo de estudio de las redes neuronales. Los experimentos de este análisis se han realizado utilizando como entorno de entrenamiento TensorFlow v2.3.0, ejecutado sobre Horovod v0.19.

Vista la relevancia de seleccionar un algoritmo u otro en el rendimiento de la primitiva *AllReduce*, a continuación se muestra el efecto que dicha elección puede suponer en el rendimiento del entrenamiento de las redes neuronales. Para valorar este impacto, se analizan cuatro escenarios que contemplan modelos y bases de datos conocidas, y cuyas combinaciones abarcan características diversas: AlexNet+ImageNet (limitado por comunicaciones), VGG11+ImageNet (mayor intensidad de cómputo), ResNet50+ImageNet y ResNet110+Cifar10 (Redes neuronales con un número de parámetros elevado con bases de datos diferentes: una de gran tamaño y otra de tamaño relativamente mucho menor). Para empezar, dada la dependencia observada entre el tamaño del mensaje y la relevancia del algoritmo utilizado, se realiza un estudio inicial consistente en analizar el tamaño de las comunicaciones necesarias durante el entrenamiento de estos escenarios. Como se observa en la Figura 5.13, los mensajes a comunicar abarcan tamaños desde 1 KiB hasta 1 GiB, por lo que el rango de bytes transmitidos se mantienen dentro de los márgenes estudiados en las secciones anteriores. Así pues, utilizando los resultados previos, se estudia a continuación cómo afecta la selección del algoritmo en el rendimiento de la red, midiendo el rendimiento global como la cantidad de imágenes procesadas por unidad de tiempo durante el entrenamiento de estas redes neuronales.

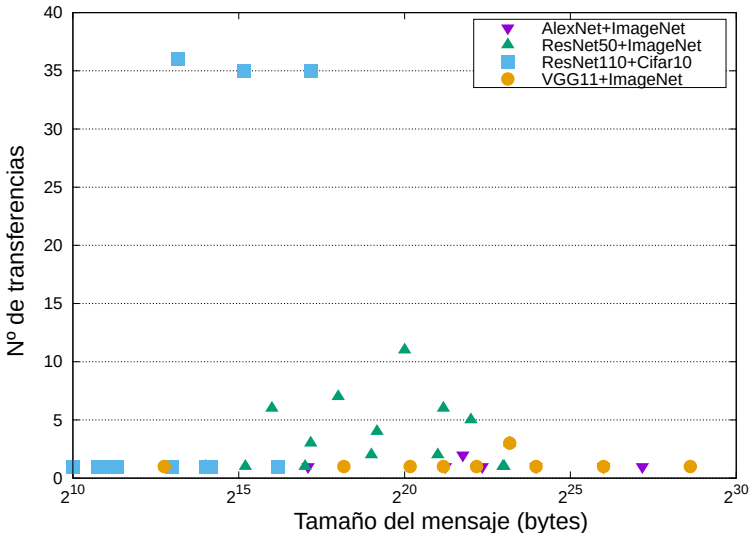

Figura 5.13: Número de mensajes y tamaño de estos en el entrenamiento de los escenarios planteados.

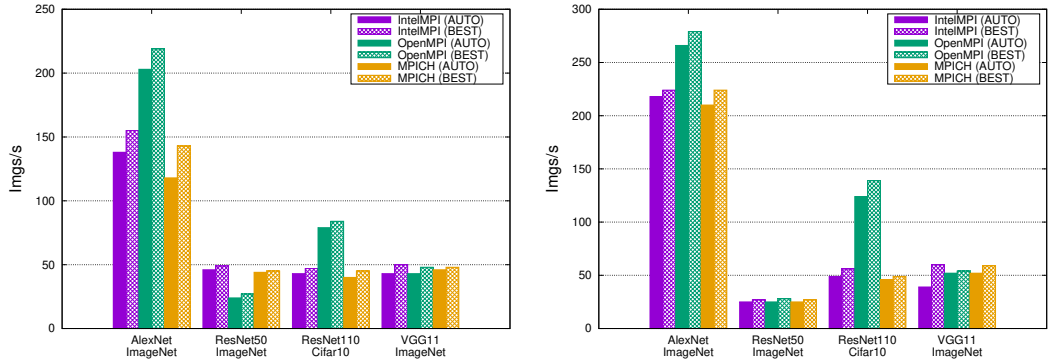

Figura 5.14: Rendimiento en imágenes por segundo del entrenamiento utilizando 8 procesos y tamaño de lote $b = 16$ (izquierda) y $b = 32$ (derecha).

En la Figura 5.14 se representan los datos recogidos en los escenarios planteados –redes y bases de datos– utilizando para cada biblioteca estudiada –Intel-MPI, Open-MPI y MPICH– los algoritmos escogidos por defecto por las propias bibliotecas y los algoritmos que previamente se ha observado que ofrecen mejor rendimiento para la comunicación. En una primera vista general puede apreciarse con estos primeros datos cómo varía el rendimiento del entrenamiento: cada escenario analizado ofrece unos valores muy distintos, la elección del algoritmo

puede tener un efecto significativo en el rendimiento alcanzable y la decisión sobre qué biblioteca ofrece las mejores prestaciones no resulta tan sencilla, ya que el rendimiento que ofrecen depende de múltiples factores. Además de variar en función del algoritmo seleccionado, del número de procesos involucrados, del modelo de red que se está entrenando y de la base de datos utilizada, también influye el tamaño del lote escogido para su entrenamiento. A modo de ejemplo, tal como se observa para la configuración ResNet50+ImageNet, el mejor rendimiento lo ofrece Intel-MPI para un tamaño de lote pequeño ($b = 16$), mientras que al incrementar el tamaño del lote ($b = 32$), los rendimientos de todas las bibliotecas decrecen, concluyendo en unos valores similares entre los cuales resultan ligeramente superiores los resultados de Open-MPI.

Para finalizar, se ha realizado un análisis más profundo de los rendimientos que ofrece cada biblioteca. Para ello, se han realizado experimentos utilizando ocho nodos y variando los diferentes parámetros que se ha observado que influyen en el rendimiento de los algoritmos. El objetivo de estos experimentos es poder estudiar la mejoría que se puede alcanzar en los diferentes escenarios planteados dentro de una misma biblioteca y comparando las diferentes bibliotecas entre sí. Para ello, se ha calculado la aceleración que se logra con respecto al algoritmo seleccionado por defecto por cada biblioteca y con respecto al algoritmo elegido por la biblioteca Intel-MPI. Dichas normalizaciones permiten comparar los resultados, tanto dentro de cada biblioteca, como respecto a las demás. Los resultados se han obtenido para el algoritmo por defecto y para el mejor algoritmo. Además, se ha ampliado el número de tamaños de lotes analizados a los siguientes: $b = 16$, 32, 64, de modo que se puedan observar mejor las tendencias de los comportamientos al incrementar este parámetro.

La Figura 5.15 muestra los resultados obtenidos y, en su análisis, se pueden realizar las siguientes observaciones:

- Al comparar las aceleraciones que se consiguen dentro de una misma biblioteca (columna izquierda), se observa en cada una de ellas que una buena selección del algoritmo, en vez de la opción por defecto, puede suponer una aceleración del rendimiento del proceso de entrenamiento por encima del 10 % en la biblioteca Open-MPI y del 20 % en el caso de las bibliotecas

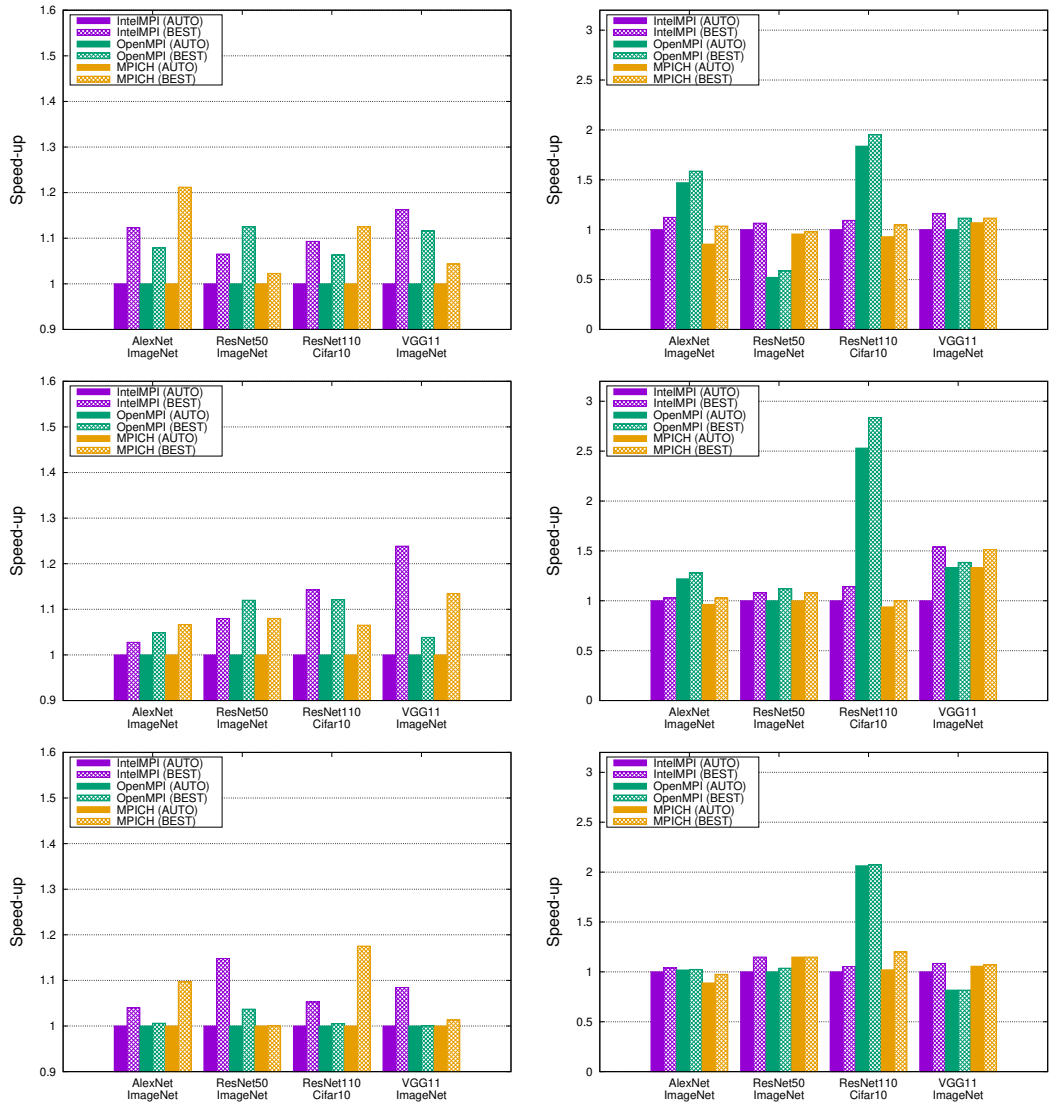

Figura 5.15: Aceleración (Speed-up) del rendimiento en TF+Horovod con un único proceso MPI por nodo. Utilizando $b = 16$ (arriba), 32 (centro), y 64 (abajo). Los resultados se han normalizado respecto al algoritmo AUTO de cada biblioteca (izquierda) y respecto del algoritmo AUTO escogido por Intel-MPI (derecha).

MPICH e Intel-MPI. Así pues, estos resultados muestran que la biblioteca Open-MPI tiende a realizar una mejor selección automática del algoritmo de *AllReduce*.

- El impacto de la elección del algoritmo en el rendimiento del entrenamiento

es menor a medida que el tamaño de lote se incrementa. Esto se debe a que la parte computacional del entrenamiento aumenta cuando lo hace el tamaño de lote y, en consecuencia, el coste relativo que representan las comunicaciones mengua.

- Al comparar las aceleraciones entre las diferentes bibliotecas (columna derecha) se aprecia que Open-MPI destaca como la biblioteca que ofrece mejores resultados, sobre todo en cuanto al escenario ResNet110+Cifar10, tanto con la opción escogida por defecto, como con el mejor algoritmo. Puede relacionarse esta superioridad con que en el rango de $2^{(17,23)} = 2^{(13,17)} \times 2^{(4,5,6)}$ (tamaño del mensaje × tamaño del lote), donde más concentración de transferencias tiene, la biblioteca Open-MPI es la que ofrece mejores rendimientos (véase las Figuras 5.13 y 5.12, derecha).

- La biblioteca Intel-MPI obtiene una mayor aceleración que Open-MPI en escenarios puntuales. En concreto, en el caso de ResNet50+Imagenet cuando se emplea un tamaño de lote $b = 16$ y VGG11+ImageNet para $b = 32$ en ambos casos (mejor algoritmo –BEST– y opción por defecto –AUTO–). En el caso del mejor algoritmo, también resulta la mejor opción en AlexNet+ImageNet y ResNet50+Imagenet con $b = 64$ y VGG11+ImageNet en los tres tamaños de lotes analizados. Nótese que VGG11+Imagenet es la red que trabaja con mayores tamaños de mensajes (ver Figura 5.13) y que en el análisis de rendimientos, para transferencias de tamaños grandes la biblioteca Intel-MPI es la que ofrece mayor rendimiento (Figura 5.12, derecha).

5.5. Conclusiones

En este capítulo se ha realizado un análisis de la primitiva *AllReduce* que profundiza en el coste temporal de ejecución de los diferentes algoritmos que implementan esta comunicación desde tres de las bibliotecas más populares: MPICH, Intel-MPI y Open-MPI.

El primer resultado que se extrae de este estudio es la distancia existente entre el coste experimental de los algoritmos y el que estiman los modelos teóricos.

De entre las diferentes variables que causan esta brecha, se ha indagado en los parámetros que influyen en dichos modelos, en particular, en β y γ, cuyo valor en la estimación debe considerar la limitación práctica y la variabilidad en función del tamaño de los mensajes. Así mismo, otra causa analizada ha sido el comportamiento de los envíos de las comunicaciones internas: partiendo de las implementaciones de los algoritmos, se ha observado que pese a buscar una asincronía en los intercambios de datos, experimentalmente no sucede el solapamiento completo que los modelos teóricos presuponen. Los resultados obtenidos muestran que dicho solapamiento es menos probable cuando los algoritmos implementan intercambios asíncronos entre pares (p_i envía y recibe datos simultáneamente con p_j), que en el caso de intercambios en cadena (envío y recepción simultánea con diferentes procesos), pero sin llegar a alcanzar en ninguno de los casos una asincronía total. Así, los algoritmos cuya estructura se basa en un anillo, tienden a conseguir mayor solapamiento de comunicaciones. Además se ha observado que la adaptación de los modelos teóricos a estas consideraciones consigue reducir drásticamente el error de aproximación respecto al coste real de los algoritmos.

Como continuación de este resultado se ha analizado la elección por defecto que escogen las bibliotecas para la ejecución de esta primitiva. El rendimiento que ofrecen los diferentes algoritmos varía en función de la cantidad de bytes con la que se está trabajando y el número de nodos que están participando. En las tres bibliotecas estudiadas se han observado importantes márgenes de mejora en la selección del algoritmo por defecto, dado que en múltiples casos –tanto en pruebas de diferentes tamaños de mensaje, como con diferente cantidad de nodos– la opción automática no se corresponde con el algoritmo que ofrece mejor rendimiento. En este sentido, la biblioteca MPICH tiende a seleccionar el algoritmo RDB, mientras que es el RSA el que mejor rendimiento ofrece. En el caso de las bibliotecas Intel-MPI y Open-MPI, la elección automática, aunque varía en función del tamaño, se decanta predominantemente por el algoritmo RSA, siendo en múltiples ocasiones los algoritmos basados en anillos la selección óptima.

Por último, se ha estudiado la influencia que esta elección tiene en el campo de las RNP. Debido a la limitación que suponen las comunicaciones en el rendimiento

del entrenamiento distribuido, reducir su coste puede tener un efecto positivo significativo en los tiempos globales. Al analizar diferentes escenarios, los resultados han reflejado que una elección cuidadosa del algoritmo de ejecución efectúa una mejora respecto de la selección automática de hasta un 20 % en el rendimiento del entrenamiento. Si se presta atención, además de al algoritmo, a una correcta elección de la biblioteca empleada, dicha mejora puede resultar significativamente importante, observándose en los resultados obtenidos rendimientos hasta un 280 % superiores.

Capítulo 6

Conclusiones

Este último capítulo está dedicado a recoger las aportaciones más relevantes del trabajo realizado, así como las conclusiones obtenidas de los estudios desarrollados y las publicaciones que se han derivado de esta tesis. Para finalizar el capítulo se apuntan algunas líneas abiertas de investigación a la conclusión de esta tesis.

6.1. Conclusiones y Principales Contribuciones

Dentro de la línea del objetivo general –implementación y validación experimental de soluciones software paralelas para el aprendizaje automático mediante RNP– se plantearon al inicio de este trabajo tres objetivos específicos relacionados con desarrollar en un entorno de simulación para RNP, estudiar los costes temporales del proceso de entrenamiento y analizar la escalabilidad de dicho proceso. En este sentido, pueden considerarse alcanzados dichos objetivos como consecuencia de las contribuciones que se han desarrollado a lo largo de la tesis y que se resumen a continuación.

Respecto al primer objetivo, relativo a la creación de un entorno para estudiar las RNP, se ha desarrollado la herramienta PyDTNN que permite ejecutar los procesos de inferencia y entrenamiento en redes neuronales de tipo perceptrón multicapa, así como en las redes convolucionales. El diseño de este entorno se fundamenta en la accesibilidad y simplicidad de cara al usuario. Resulta, por

consiguiente, un entorno práctico, a la vez que sencillo de modificar y personalizar, permitiendo así implementar múltiples estrategias y técnicas para mejorar el rendimiento del entrenamiento. Algunas de estas estrategias, como es el caso del paralelismo de datos, han sido integradas en este entorno, logrando como resultado rendimientos y precisiones comparables con los ofrecidos por el conocido entorno TensorFlow.

En cuanto al segundo objetivo propuesto, el estudio del coste temporal que conlleva el entrenamiento y la inferencia de las redes neuronales, ha sido abordado desde diversas perspectivas. Por un lado, se han investigado los métodos que se exponen a continuación para la estimación de dichos costes y, por otro, se ha realizado un estudio en profundidad sobre la estimación del coste de las comunicaciones utilizadas en el entrenamiento distribuido mediante el uso de paralelismo de datos. Las conclusiones relacionadas con este último estudio se comentarán más adelante junto con el resto de resultados relativos a la primitiva *AllReduce*.

En cuanto a los métodos de estimación, se han diseñado y validado dos modelos, uno íntegramente teórico y otro con una base experimental. El primero de ellos se fundamenta en una estimación analítica, tanto del cómputo, como de las comunicaciones, que contabiliza las operaciones a realizar y calcula los tiempos parciales de ejecución en base a la velocidad aproximada a la que estas se realizan. El segundo modelo mantiene una estimación teórica del coste de las comunicaciones, pero en la parte computacional basa el coste aritmético y de transferencias de cada núcleo en la estimación ofrecida por redes de tipo MLP entrenadas con datos experimentales de la misma plataforma. Esta versión del modelado permite tener en cuenta en las estimaciones consideraciones inherentes a la plataforma que no pueden incluirse en un modelo teórico. Por ello, aunque ambos modelos realizan una buena estimación del coste del entrenamiento, este último ofrece aproximaciones más precisas. No obstante, debe tenerse en cuenta el coste superior de este modelo derivado, del aprendizaje de dichas redes MLP, incluyendo la creación de la base de datos, de las redes neuronales y el posterior entrenamiento.

Respecto al último objetivo establecido, una vez validados los modelos previos y mediante el uso de estos, se afrontó el análisis de escalabilidad del entrena-

miento distribuido basado en paralelismo de datos. De este análisis destaca, por un lado, la proporcionalidad existente entre el tiempo de ejecución de un paso de entrenamiento (FP+BP) y el número de datos que componen cada lote. Así, aumentar el tamaño del lote (b) mejora la intensidad aritmética por nodo, aunque debe tenerse presente la posible pérdida de convergencia del entrenamiento en caso que se incremente demasiado el valor de dicha variable. Por otro lado, resultan relevantes las comunicaciones como potencial cuello de botella para el rendimiento del entrenamiento. Debe estudiarse si estas se convierten en un factor limitante, tanto al incrementar el número de nodos participantes, como al utilizar plataformas cuyo ancho de banda de enlace no supere un umbral mínimo.

Por último, en base a las limitaciones encontradas en las comunicaciones, se realizó un estudio más intenso de la primitiva *AllReduce* necesaria en el paralelismo de datos. Este análisis contribuye también a los objetivos planteados en cuanto que se investigó cómo reducir la diferencia entre los costes temporales reales de dicha comunicación y las estimaciones teóricas utilizadas en los modelos. En base a los experimentos realizados, resaltan como posibles causas de dicha brecha las falsas expectativas respecto del valor alcanzable por ciertos parámetros (ancho de banda de memoria y de enlace), así como de la capacidad de la red para ejecutar comunicaciones asíncronas que obtengan un solapamiento completo de las rutinas no bloqueantes. En el caso del ancho de banda de memoria, se observó la necesidad de adoptar un valor diferente en función del tamaño de los mensajes transmitidos. En cuanto a la asincronía, los algoritmos que implementan la comunicación cuyo procedimiento se basa en estructuras de tipo anillo alcanzan experimentalmente mayor solapamiento que aquellos que realizan intercambios entre pares. Las bibliotecas tienden a elegir estos segundos más que los primeros como algoritmos por defecto, lo cual hace que habitualmente no se escoja automáticamente el algoritmo que realmente ofrece mejor rendimiento. En base a dichos resultados se han propuesto nuevos modelos que mejoran la estimación del coste temporal.

Por otro lado, se ha analizado el algoritmo que ofrece mejores resultados, y por tanto debería elegirse como opción por defecto, en diferentes bibliotecas (MPICH, Intel-MPI y Open-MPI) para un amplio rango de tamaño de mensajes.

En todas las bibliotecas se ha observado margen de mejora, al tender estas a seleccionar algoritmos con intercambios entre pares (como RDB y RSA) en vez de algoritmos con intercambios en cadena, tipo anillo (como RNG o SRG). El efecto que supone una buena elección del algoritmo afecta también al rendimiento del entrenamiento de las RNP. Los resultados obtenidos al analizar diferentes redes y conjuntos de datos muestran que, dada una biblioteca, una buena elección del algoritmo incrementa en hasta un 20 % el rendimiento, mientras que una buena elección, tanto de biblioteca como de algoritmo, ofrece rendimientos hasta un 280 % superiores. Esta mejora, por tanto, contribuye también a los objetivos planteados al reducir la limitación en el rendimiento que estas comunicaciones pueden suponer para la escalabilidad del entrenamiento.

6.2. Publicaciones

Los resultados de la investigación realizada en esta tesis han sido publicados tanto en congresos, nacionales e internacionales, como en artículos de revistas indexadas. En las siguientes secciones se detallan las contribuciones científicas derivadas de la investigación realizada, indicando los datos de cada una de ellas.

6.2.1. Artículos en revista

Using Machine Learning to Model the Training Scalability of Convolutional Neural Networks on Clusters of GPUs [7] Computing Springer - Science Edition - Computer Science Theory and Methods. 2021 (ISI 2,220). DOI 10.1007/s00607-021-00997-9. Resumen: In this work, we build a general piecewise model to analyze data-parallel (DP) training costs of convolutional neural networks (CNNs) on clusters of GPUs. This general model is based on i) multilayer perceptrons (MLPs) in charge of modeling the NVIDIA cuDNN/cuBLAS library kernels involved in the training of some of the state-of-the-art CNNs; and ii) an analytical model in charge of modeling the NVIDIA NCCL Allreduce collective primitive using the Ring algorithm. The CNN training scalability study performed using this model in combination with the Roofline technique on varying batch sizes, node (floating-point) arithmetic performance, node memory

bandwidth, network link bandwidth, and cluster dimension unveil some crucial bottlenecks at both GPU and cluster level. To provide evidence of this analysis, we validate the accuracy of the proposed model against a Python library for distributed deep learning training.

PyDTNN: A User-Friendly and Extensible Framework for Distributed Deep Learning [5] Journal of Supercomputing - Computer Science Theory and Methods. 2021 (JCR 2,469 - Q2). DOI 10.1007/s11227-021-03673-z. Resumen: We introduce a framework for training deep neural networks on clusters of computers with the following appealing properties: (1) It is developed in Python, exposing an amiable interface that provides an accessible entry point for the newcomer; (2) it is extensible, offering a customizable tool for the more advanced user in deep learning; (3) it covers the main functionality appearing in convolutional neural networks; and (4) it delivers reasonable inter-node parallel performance exploiting data parallelism by leveraging MPI via MPI4Py for communication and NumPy for the efficient execution of (multithreaded) numerical kernels.

Analyzing the impact of the MPI AllReduce in distributed training of Convolutional Neural Networks [18] Computing - Science Edition - Computer Science Theory and Methods. 2022 (ISI 2,220). DOI 10.1007/s00607-021-01029-2. Resumen: For many distributed applications, data communication poses an important bottleneck from the points of view of performance and energy consumption. As more cores are integrated per node, in general the global performance of the system increases yet eventually becomes limited by the interconnection network. This is the case for distributed data-parallel training of convolutional neural networks (CNNs), which usually proceeds on a cluster with a small to moderate number of nodes. In this paper, we analyze the performance of the Allreduce collective communication primitive, a key to the efficient data-parallel distributed training of CNNs. Our study targets the distinct realizations of this primitive in three high performance instances of Message Passing Interface (MPI), namely MPICH, OpenMPI, and IntelMPI, and employs a cluster equipped with state-of-the-art processor and network technologies. In addition, we apply the insights gained from the experimental analysis to the optimization

of the TensorFlow framework when running on top of Horovod. Our study reveals that a careful selection of the most convenient MPI library and Allreduce (ARD) realization accelerates the training throughput by a factor of 1.2× compared with the default algorithm in the same MPI library, and up to 2.8× when comparing distinct MPI libraries in a number of relevant combinations of CNN model+dataset.

6.2.2. Artículos en congresos

PyDTNN: An Extensible and User-Friendly Python Library for Distributed Deep Learning [14] 20th International Conference Computational and Mathematical Methods in Science and Engineering. 31/07/2020 - Cádiz (España) - Universidad de Cádiz. Resumen: Artificial intelligence in general, and machine learning via deep neural networks (DNNs) in particular, are experiencing an explosive growth due to the combined effect of new algorithmic techniques, vast amounts of computer power, and the explosion in the amount of training data [3, 1]. This scenario has pushed the industry to design customized architectures and components –e.g., NVIDIA's tensor cores, Google's tensor processing units (TPUs), etc.– as well as frameworks for DL such as Google's TensorFlow, Facebook's PyTorch and Caffe, and Keras, among others. DL frameworks usually expose a high-level application programming interface (API), in many cases accessible from a user-friendly programming language, such as Python, for ease the interaction with DNN models and datasets. From the perspective of high performance computing (HPC), most of these frameworks can exploit the compute power of a conventional multicore architecture; some of them can off-load the most compute-intensive calculations to some sort of accelerator (e.g., a graphics processing unit, or GPU); and a few only can be integrated into some other software, such as Horovod [2], to perform distributed training, e.g., on a cluster of computers. While the existence of these frameworks has doubtless contributed to the adoption of DL technologies, we also find that the level of internal complexity of these packages turns their modification (customization) into a fairly difficult task. A particular problem that we are concerned with is the high-level

implementation of distributed training for DNNs, which adds an extra layer of intricacy to these frameworks.

Performance Modeling for Distributed Training of Convolutional Neural Networks [16] 29th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP 2021). 11/03/2021 - Valladolid (España) - Universidad de Valladolid. Resumen: We perform a theoretical analysis comparing the scalability of data versus model parallelism, applied to the distributed training of deep convolutional neural networks (CNNs), along five axes: batch size, node (floating-point) arithmetic performance, node memory bandwidth, network link bandwidth, and cluster dimension. Our study relies on analytical performance models that can be configured to reproduce the components and organization of the CNN model as well as the hardware configuration of the target distributed platform. In addition, we provide evidence of the accuracy of the analytical models by performing a validation against a Python library for distributed deep learning training.

Evaluation of MPI Allreduce for Distributed Training of Convolutional Neural Network [15] 29th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP 2021). 11/03/2021 - Valladolid (España) - Universidad de Valladolid. Resumen: Training deep neural networks is a costly procedure, often performed via sophisticated deep learning frameworks on clusters of computers. As faster processor technologies are integrated into these cluster facilities (e.g., NVIDIA's graphics accelerators or Google's tensor processing units), the communication component of the training process rapidly becomes a performance bottleneck. In this paper, we offer a complete analysis of the key collective communication primitive for the distributed data-parallel training of convolutional network networks (CNNs) focused on three relevant instances of the Message Passing Interface (MPI): MPICH, OpenMPI, and IntelMPI. In addition, our experimental evaluation is extended to expose the practical impact of this collective primitive when the training is performed using TensorFlow+Horovod on a 16-node cluster. Finally, the theoretical analysis is further

refined to a number of accelerated cluster configurations that are emulated by adjusting the communication-arithmetic ratio of the training process.

A Flexible Research-Oriented Framework for Distributed Training of Deep Neural Networks [4] The 22nd IEEE International Workshop on Parallel and Distributed Scientific and Engineering Computing en IPDPS 2021. 21/05/2021 - Oregon (USA) - St. Francis Xavier University. Resumen: We present PyDTNN, a framework for training deep neural networks (DNNs) on clusters of computers that has been designed as a research-oriented tool with a low learning curve. Our parallel training framework offers a set of functionalities that cover several must-have features for advanced deep learning (DL) software: 1) it is developed in Python in order to expose an accessible entry point for the newcomer; 2) it is extensible, allowing users to prototype new research ideas without requiring them to deal with complex software-stacks; and 3) it delivers high parallel performance, exploiting MPI via mpi4py/NCCL for communication; and NumPy, cuDNN, and cuBLAS for computation. This paper provides practical evidence that PyDTNN attains similar accuracy and parallel performance to those exhibited by Google's TensorFlow (TF), though we recognize that PyDTNN cannot compete with a production-level framework such as TF or PyTorch in terms of maturity and functionality. Instead, PyDTNN is designed as an accessible and customizable tool for prototyping ideas related to distributed training of DNN models on clusters.

PyDTNN: Entorno para Entrenamiento e Inferencia con Redes Neuronales Profundas [6] Jornadas SARTECO 2021. 23/09/2021 - Málaga (España) - Universidad de Málaga. Resumen: En este trabajo se presenta PyDTNN, un entorno para el entrenamiento de redes neuronales profundas (DNNs) sobre clústeres de computadores. PyDTNN se ha diseñado para ser un entorno orientado a la investigación con una curva de aprendizaje baja, capaz de ejecutarse en paralelo y que cubriera los aspectos imprescindibles en un software avanzado de aprendizaje profundo (DL). Sus principales características son: i) está desarrollado en Python, lo que permite que los nuevos usuarios puedan comenzar a utilizarlo fácilmente; ii) es flexible, facilitando el prototipado de nuevas ideas

sin tener que enfrentarse a código excesivamente complejo; iii) ofrece un alto rendimiento paralelo, utilizando mpi4py/NCCL para la comunicación; y NumPy, cuDNN y cuBLAS para los cálculos. Aunque PyDTNN no puede competir a nivel de producción con entornos como TensorFlow o PyTorch en términos de madurez y funcionalidad, consideramos que PyDTNN es un entorno mucho más accesible y adaptable para el desarrollo y evaluación de prototipos relacionados con el entrenamiento e inferencia distribuida de modelos de DNNs en clústeres. No obstante lo anterior, en este artículo se muestra que PyDTNN alcanza una precisión y un rendimiento paralelo similar a los obtenidos por TensorFlow de Google con Horovod en entrenamiento y como los obtenidos por ArmNN y TFLite en inferencia.

Mejora de los Modelos Predictivos para la Comunicación Colectiva MPI AllReduce [17] Jornadas SARTECO 2021. 23/09/2021 - Málaga (España) - Universidad de Málaga. Resumen: Las comunicaciones colectivas son una parte clave en las aplicaciones distribuidas. Sin embargo, al incrementar el número de nodos que intervienen en la aplicación paralela, las comunicaciones a menudo se vuelven más costosas y se convierten en un cuello de botella. Por ello, modelar el coste temporal de ejecución de estas comunicaciones es una importante tarea, a la que a menudo se recurre para analizar el coste y las limitaciones de una cierta aplicación. En este artículo nos centramos en el estudio y modelización de la comunicación colectiva Allreduce para la operación suma. Al analizar los modelos predictivos actuales para los distintos algoritmos de ejecución de esta primitiva, los resultados experimentales muestran una diferencia considerable entre el coste real y el estimado por el modelo. En este estudio, se analizan las causas de esta diferencia y, en base a estas, se diseñan unos nuevos modelos que permiten obtener aproximaciones más precisas al coste real de la comunicación Allreduce para los distintos algoritmos estudiados.

6.3. Líneas de Investigación Abiertas

Las líneas en las que se puede continuar la investigación realizada se centran en el estudio de técnicas algorítmicas para la aceleración de capas de conexión

completa y convolucionales de las RNP. En este sentido, existen diferentes tipos de estrategias que permiten dicha aceleración. Algunas de las técnicas más conocidas son las técnicas de Winograd y Strassen, las cuales obtienen dicha aceleración mediante la reducción del número de operaciones [2, 53]. Sin embargo, para la continuación de este estudio se ha planteado abordar las estrategias que logran la aceleración mediante la compresión de las matrices de pesos, reduciendo así el tamaño de las RNP. La técnica en la que se fundamentan dichas compresiones consiste en la factorización de rango incompleto, concretamente en el uso de la descomposición en valores singulares o SVD por sus siglas en inglés, consistente en la factorización de la matriz de pesos W en tres nuevas matrices,

$$W = U\Sigma V,$$

donde la matriz Σ resulta una matriz diagonal cuyos elementos coinciden con los valores singulares en orden descendiente. Esta casuística permite eliminar las filas y columnas correspondientes a los valores singulares más pequeños, controlando el efecto sobre la matriz resultante del producto $W*$.

La compresión alcanzada vendrá definida, por tanto, por la cantidad de valores singulares eliminados, siendo mayor cuantos más elementos se supriman en la descomposición. No obstante, debe tenerse en cuenta que la eliminación de valores singulares tiene un efecto directo en la precisión de la matriz reconstruida $W*$ y, por tanto, en la precisión que la red ofrece cuando se utiliza como pesos las matrices reconstruidas. El efecto de estas técnicas, en consecuencia, deberá valorarse en un equilibrio entre compresión y precisión de la red una vez comprimida y, por tanto, su valor dependerá del margen de error que puede tolerarse en los resultados de la red.

A continuación se muestran las diferentes alternativas que se han contemplado para comprimir las redes neuronales mediante esta factorización.

Particionado de matriz La primera técnica planteada para combinar con la factorización SVD y aumentar así la comprensión de la red, consiste en particionar en bloques las matrices de pesos, de modo que se aplique la factorización por cada bloque resultante [47, 50]. En este sentido se ha estudiado el impacto que se alcanza en función de las dimensiones de los bloques en los que se realiza el

particionado de la matriz, comprobando el efecto de utilizar bloques cuadrados de tamaños fijos, bloques de filas (es decir, bloques cuyo número de columnas coincide con el número de columnas de la matriz de pesos), bloques de columnas (coincidiendo el número de filas del bloque con el de la matriz) y bloques por posición en la capa del filtro. Esta técnica se ha implementado sobre PyDTNN y los resultados iniciales muestran que la selección de bloques de filas ofrece habitualmente mejores resultados. Asimismo, aunque no generan a modo general tan buenos resultados, sí que existen ciertos tamaños de bloques cuadrados que ofrecen una alta compresión con poca pérdida de precisión.

Reorganización de matriz Otra técnica que puede combinarse junto con el particionado de matriz, consiste en aplicar previamente una transformación que intercambie tanto filas como columnas. La finalidad de esta reorganización es que, al aplicar el particionado sobre la matriz resultante, los bloques tengan una mayor dependencia lineal y puedan, así, conseguir una mayor capacidad de compresión. Un ejemplo sencillo puede observarse en la Figura 6.1.

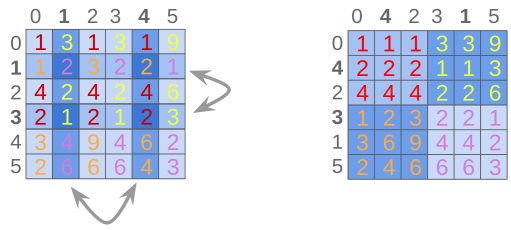

Figura 6.1: Ejemplo de reorganización de una matriz (izq.) para tener bloques con mayor dependencia lineal (derecha).

Se ha realizado un primer estudio investigando las propiedades de diversas técnicas y métricas algebraicas para hallar las particiones óptimas. Sin embargo, la dependencia entre la matriz completa y las técnicas contempladas ha dificultado encontrar un método que, sin ser extremadamente costoso temporalmente, ofrezca una reorganización que maximice la dependencia lineal dentro de cada futuro bloque particionado de la matriz, es decir, dentro de cada bloque resultante al particionar la matriz reorganizada. En esta línea, como posible alternativa también se propone diseñar una heurística que proporcione una reorganización

de las matrices que, pese a no ser la organización óptima, incremente significativamente las dependencias. En este sentido, se han revisado artículos relacionados con técnicas similares de compresión de matrices, como [51].

Separación de capas Esta última estrategia que se propone supone una gran diferencia respecto de las anteriores. Mientras que las técnicas previas se aplicaban únicamente a la matriz de pesos de cada capa de manera individual, esta técnica se aplica sobre el modelo en sí, dado que consiste en separar cada una de las capas que utilizan pesos (capas de conexión completa y convolucionales), en dos capas cuyos pesos se deriven de la factorización SVD [52]. Esta técnica, por tanto, modifica el modelo lo cual conlleva ciertos inconvenientes, dado que ya no se va a trabajar con la misma red, pero abre la opción a tolerar una mayor pérdida de precisión a costa de un reentrenamiento posterior de la red. Se ha iniciado la integración de esta estrategia en PyDTNN, teniendo ya los primeros resultados para la separación de capas FC.

Combinación de técnicas Para finalizar, notar que las estrategias mencionadas pueden combinarse entre sí, así como utilizarlas junto a otras conocidas técnicas de compresión como, por ejemplo, la cuantización [39]. De este modo, se podría mejorar la capacidad de compresión de los pesos. La alternativa prevista como futuro trabajo consistiría en realizar una primera separación de capas con reentrenamiento. Una vez obtenido el nuevo modelo con los pesos reentrenados, utilizar la técnica de reorganización de filas y columnas para incrementar la dependencia lineal de los bloques para, finalmente, realizar el particionado de la matriz reorganizada. También se han realizado pruebas combinando algunas de estas estrategias con la cuantización, las cuales han mostrado resultados positivos, por lo que se espera que el uso simultáneo de dichas técnicas en un futuro ofrezca compresiones significativas.

Chapter 6

Conclusions

This last chapter is dedicated to the main contributions of the work carried out, as well as the conclusions drawn from the studies and publications derived from this thesis. The chapter concludes by pointing out some open research to be carried out before concluding this thesis.

6.1. Conclusions and Main Contributions

Within the general objective –to implement and experimentally validate parallel software solutions for machine learning using RNPs– three specific objectives were set at the beginning of this work, related to the development of a simulation environment for RNPs, the study of the time cost of the training process and the analysis of the scalability of this process. In this sense, these objectives can be considered to have been achieved as a result of the contributions developed throughout the thesis, which are summarized below.

For the first objective, which concerns the creation of an environment for studying PNNs, the PyDTNN tool has been developed. This tool allows inference and training processes to be carried out on multilayer perceptron neural networks, as well as on convolutional networks. The design of this environment is based on accessibility and simplicity for the user. It is therefore a practical environment. At the same time, it is easy to modify and adapt, allowing the implementation of various strategies and techniques to improve training performance. Some of these

strategies, such as data parallelism, have been integrated into this environment, resulting in performance and accuracy comparable to the well-known TensorFlow environment.

With regard to the second proposed objective, the study of the time costs involved in the training and inference of neural networks, several variants have been developed. On the one hand, the methods described below for estimating these costs have been studied and, as will be shown below, an in-depth study has been carried out on the estimation of the communication costs used in distributed training through the use of data parallelism.

In terms of estimation methods, two models have been designed and validated, one entirely theoretical and the other with an experimental basis. The first is based on an analytical estimation of both computation and communication, counting the operations to be performed and calculating the partial execution times based on the approximate speed at which they are performed. The second model maintains a theoretical estimate of the communication costs, but in the computation part it bases the arithmetic and transmission costs of each core on the estimate provided by MLP-type networks trained on experimental data from the same platform. This version of modeling allows the estimates to take into account considerations inherent to the platform that cannot be included in a theoretical model. Therefore, although both models provide a good estimate of the training cost, the latter provides more accurate approximations. However, the higher cost of this model due to the learning of these MLPs, including the creation of the database, the neural networks and the subsequent training, must be taken into account.

Regarding the last objective, the scalability analysis of distributed training based on data parallelism was carried out after validating and using the previous models. This analysis highlights, on the one hand, the existing proportionality between the execution time of a training step (FP+BP) and the number of data that make up each batch. Thus, increasing the batch size (b) improves the computational intensity per node. However, the possible loss of training convergence should be taken into account if the value of this variable is increased too much.

On the other hand, communication is relevant as a potential bottleneck for training performance. It should be investigated whether communication becomes a limiting factor, both by increasing the number of participating nodes and by using platforms whose link bandwidth does not exceed a minimum threshold.

Finally, a more in-depth study of the necessary primitives in data parallelism was carried out, based on the limitations encountered in communication. This analysis also contributes to the objectives set, by investigating how to reduce the difference between the real time cost of such communication and the theoretical estimates used in the models. Based on the experiments performed, false expectations regarding the achievable value of certain parameters (memory and link bandwidth), as well as the network's ability to perform asynchronous communication with full non-blocking routine overlap, stand out as possible causes of this gap. In the case of memory bandwidth, the need to adopt a different value depending on the size of the messages transmitted was noted. In terms of asynchrony, algorithms implementing communication based on ring structures experimentally achieve greater overlap than those implementing peer-to-peer exchanges. Libraries tend to select the latter rather than the former as default algorithms, which means that the algorithm that actually offers better performance is usually not automatically selected. Based on these results, new models have been proposed to improve the estimation of time costs.

On the other hand, different libraries (MPICH, Intel-MPI and Open-MPI) have been analyzed for a wide range of message sizes to determine which algorithm gives the best results and should therefore be chosen as the default option. In all libraries, room for improvement was observed, as the libraries tend to select algorithms with pairwise exchanges (such as RDB and RSA) instead of algorithms with chain-like, ring-like exchanges (such as RNG or SRG). The effect of a good choice of algorithm also affects the training performance of the RNPs. Results obtained by analyzing different networks and datasets show that, given a library, a good choice of algorithm improves performance by up to 20%, while a good choice of both library and algorithm improves performance by up to 280%. This improvement therefore also contributes to the stated goals by reducing the per-

formance limitation that these communications can impose on the scalability of training.

6.2. Publications

The results of the research carried out in this thesis have been published both in national and international conferences and in articles in indexed journals. The following sections describe in detail the scientific contributions derived from the research carried out, giving the details of each one of them.

6.2.1. Journal articles

Using Machine Learning to Model the Training Scalability of Convolutional Neural Networks on Clusters of GPUs [7] Computing Springer - Science Edition - Computer Science Theory and Methods. 2021 (ISI 2,220). DOI 10.1007/s00607-021-00997-9. Abstract: In this work, we build a general piecewise model to analyze data-parallel (DP) training costs of convolutional neural networks (CNNs) on clusters of GPUs. This general model is based on i) multilayer perceptrons (MLPs) in charge of modeling the NVIDIA cuDNN/cuBLAS library kernels involved in the training of some of the state-of-the-art CNNs; and ii) an analytical model in charge of modeling the NVIDIA NCCL Allreduce collective primitive using the Ring algorithm. The CNN training scalability study performed using this model in combination with the Roofline technique on varying batch sizes, node (floating-point) arithmetic performance, node memory bandwidth, network link bandwidth, and cluster dimension unveil some crucial bottlenecks at both GPU and cluster level. To provide evidence of this analysis, we validate the accuracy of the proposed model against a Python library for distributed deep learning training.

PyDTNN: A User-Friendly and Extensible Framework for Distributed Deep Learning [5] Journal of Supercomputing - Computer Science Theory and Methods. 2021 (JCR 2,469 - Q2). DOI 10.1007/s11227-021-03673-z. Abstract: We introduce a framework for training deep neural networks on clusters of

computers with the following appealing properties: (1) It is developed in Python, exposing an amiable interface that provides an accessible entry point for the newcomer; (2) it is extensible, offering a customizable tool for the more advanced user in deep learning; (3) it covers the main functionality appearing in convolutional neural networks; and (4) it delivers reasonable inter-node parallel performance exploiting data parallelism by leveraging MPI via MPI4Py for communication and NumPy for the efficient execution of (multithreaded) numerical kernels.

Analyzing the impact of the MPI AllReduce in distributed training of Convolutional Neural Networks [18] Computing - Science Edition - Computer Science Theory and Methods. 2022 (ISI 2,220). DOI 10.1007/s00607-021-01029-2. Abstract: For many distributed applications, data communication poses an important bottleneck from the points of view of performance and energy consumption. As more cores are integrated per node, in general the global performance of the system increases yet eventually becomes limited by the interconnection network. This is the case for distributed data-parallel training of convolutional neural networks (CNNs), which usually proceeds on a cluster with a small to moderate number of nodes. In this paper, we analyze the performance of the Allreduce collective communication primitive, a key to the efficient data-parallel distributed training of CNNs. Our study targets the distinct realizations of this primitive in three high performance instances of Message Passing Interface (MPI), namely MPICH, OpenMPI, and IntelMPI, and employs a cluster equipped with state-of-the-art processor and network technologies. In addition, we apply the insights gained from the experimental analysis to the optimization of the TensorFlow framework when running on top of Horovod. Our study reveals that a careful selection of the most convenient MPI library and Allreduce (ARD) realization accelerates the training throughput by a factor of 1.2× compared with the default algorithm in the same MPI library, and up to 2.8× when comparing distinct MPI libraries in a number of relevant combinations of CNN model+dataset.

6.2.2. Artículos en congresos

PyDTNN: An Extensible and User-Friendly Python Library for Distributed Deep Learning [14] 20th International Conference Computational and Mathematical Methods in Science and Engineering. 31/07/2020 - Cádiz (España) - Universidad de Cádiz. Abstract: Artificial intelligence in general, and machine learning via deep neural networks (DNNs) in particular, are experiencing an explosive growth due to the combined effect of new algorithmic techniques, vast amounts of computer power, and the explosion in the amount of training data [3, 1]. This scenario has pushed the industry to design customized architectures and components –e.g., NVIDIA's tensor cores, Google's tensor processing units (TPUs), etc.– as well as frameworks for DL such as Google's TensorFlow, Facebook's PyTorch and Caffe, and Keras, among others. DL frameworks usually expose a high-level application programming interface (API), in many cases accessible from a user-friendly programming language, such as Python, for ease the interaction with DNN models and datasets. From the perspective of high performance computing (HPC), most of these frameworks can exploit the compute power of a conventional multicore architecture; some of them can off-load the most compute-intensive calculations to some sort of accelerator (e.g., a graphics processing unit, or GPU); and a few only can be integrated into some other software, such as Horovod [2], to perform distributed training, e.g., on a cluster of computers. While the existence of these frameworks has doubtless contributed to the adoption of DL technologies, we also find that the level of internal complexity of these packages turns their modification (customization) into a fairly difficult task. A particular problem that we are concerned with is the high-level implementation of distributed training for DNNs, which adds an extra layer of intricacy to these frameworks.

Performance Modeling for Distributed Training of Convolutional Neural Networks [16] 29th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP 2021). 11/03/2021 - Valladolid (España) - Universidad de Valladolid. Abstract: We perform a theoretical analysis comparing the scalability of data versus model parallelism, applied to the

distributed training of deep convolutional neural networks (CNNs), along five axes: batch size, node (floating-point) arithmetic performance, node memory bandwidth, network link bandwidth, and cluster dimension. Our study relies on analytical performance models that can be configured to reproduce the components and organization of the CNN model as well as the hardware configuration of the target distributed platform. In addition, we provide evidence of the accuracy of the analytical models by performing a validation against a Python library for distributed deep learning training.

Evaluation of MPI Allreduce for Distributed Training of Convolutional Neural Network [15] 29th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP 2021). 11/03/2021 - Valladolid (España) - Universidad de Valladolid. Abstract: Training deep neural networks is a costly procedure, often performed via sophisticated deep learning frameworks on clusters of computers. As faster processor technologies are integrated into these cluster facilities (e.g., NVIDIA's graphics accelerators or Google's tensor processing units), the communication component of the training process rapidly becomes a performance bottleneck. In this paper, we offer a complete analysis of the key collective communication primitive for the distributed data-parallel training of convolutional network networks (CNNs) focused on three relevant instances of the Message Passing Interface (MPI): MPICH, OpenMPI, and IntelMPI. In addition, our experimental evaluation is extended to expose the practical impact of this collective primitive when the training is performed using TensorFlow+Horovod on a 16-node cluster. Finally, the theoretical analysis is further refined to a number of accelerated cluster configurations that are emulated by adjusting the communication-arithmetic ratio of the training process.

A Flexible Research-Oriented Framework for Distributed Training of Deep Neural Networks [4] The 22nd IEEE International Workshop on Parallel and Distributed Scientific and Engineering Computing en IPDPS 2021. 21/05/2021 - Oregon (USA) - St. Francis Xavier University. Abstract: We present PyDTNN, a framework for training deep neural networks (DNNs) on clusters of computers that has been designed as a research-oriented tool with a

low learning curve. Our parallel training framework offers a set of functionalities that cover several must-have features for advanced deep learning (DL) software: 1) it is developed in Python in order to expose an accessible entry point for the newcomer; 2) it is extensible, allowing users to prototype new research ideas without requiring them to deal with complex software-stacks; and 3) it delivers high parallel performance, exploiting MPI via mpi4py/NCCL for communication; and NumPy, cuDNN, and cuBLAS for computation. This paper provides practical evidence that PyDTNN attains similar accuracy and parallel performance to those exhibited by Google's TensorFlow (TF), though we recognize that PyDTNN cannot compete with a production-level framework such as TF or PyTorch in terms of maturity and functionality. Instead, PyDTNN is designed as an accessible and customizable tool for prototyping ideas related to distributed training of DNN models on clusters.

PyDTNN: Entorno para Entrenamiento e Inferencia con Redes Neuronales Profundas [6] Jornadas SARTECO 2021. 23/09/2021 - Málaga (España) - Universidad de Málaga. Abstract: En este trabajo se presenta PyDTNN, un entorno para el entrenamiento de redes neuronales profundas (DNNs) sobre clústeres de computadores. PyDTNN se ha diseñado para ser un entorno orientado a la investigación con una curva de aprendizaje baja, capaz de ejecutarse en paralelo y que cubriera los aspectos imprescindibles en un software avanzado de aprendizaje profundo (DL). Sus principales características son: i) está desarrollado en Python, lo que permite que los nuevos usuarios puedan comenzar a utilizarlo fácilmente; ii) es flexible, facilitando el prototipado de nuevas ideas sin tener que enfrentarse a código excesivamente complejo; iii) ofrece un alto rendimiento paralelo, utilizando mpi4py/NCCL para la comunicación; y NumPy, cuDNN y cuBLAS para los cálculos. Aunque PyDTNN no puede competir a nivel de producción con entornos como TensorFlow o PyTorch en términos de madurez y funcionalidad, consideramos que PyDTNN es un entorno mucho más accesible y adaptable para el desarrollo y evaluación de prototipos relacionados con el entrenamiento e inferencia distribuida de modelos de DNNs en clústeres. No obstante lo anterior, en este artículo se muestra que PyDTNN alcanza una precisión y un rendimiento paralelo similar a los obtenidos por TensorFlow de Google

con Horovod en entrenamiento y como los obtenidos por ArmNN y TFLite en inferencia.

Mejora de los Modelos Predictivos para la Comunicación Colectiva MPI AllReduce [17] Jornadas SARTECO 2021. 23/09/2021 - Málaga (España) - Universidad de Málaga. Abstract: Las comunicaciones colectivas son una parte clave en las aplicaciones distribuidas. Sin embargo, al incrementar el número de nodos que intervienen en la aplicación paralela, las comunicaciones a menudo se vuelven más costosas y se convierten en un cuello de botella. Por ello, modelar el coste temporal de ejecución de estas comunicaciones es una importante tarea, a la que a menudo se recurre para analizar el coste y las limitaciones de una cierta aplicación. En este artículo nos centramos en el estudio y modelización de la comunicación colectiva Allreduce para la operación suma. Al analizar los modelos predictivos actuales para los distintos algoritmos de ejecución de esta primitiva, los resultados experimentales muestran una diferencia considerable entre el coste real y el estimado por el modelo. En este estudio, se analizan las causas de esta diferencia y, en base a estas, se diseñan unos nuevos modelos que permiten obtener aproximaciones más precisas al coste real de la comunicación Allreduce para los distintos algoritmos estudiados.

6.3. Open Lines of Research

The lines in which the research carried out can be continued focus on the study of algorithmic techniques for the acceleration of full and convolutional connection layers of RNPs. In this sense, there are different types of strategies that allow such acceleration. Some of the best known techniques are the Winograd and Strassen techniques, which achieve such acceleration by reducing the number of operations [2, 53]. However, for the continuation of this study, it has been proposed to look at strategies that achieve acceleration by compressing the weight matrices, thus reducing the size of the RNPs. The technique on which these compressions are based consists of incomplete rank factorization, specifically the use of singular value decomposition or SVD, which consists of factorizing the

weight matrix W into three new matrices,

$$W = U\Sigma V,$$

where the matrix *Sigma* is a diagonal matrix whose elements coincide with the singular values in descending order. This casuistry allows us to eliminate the rows and columns corresponding to the smallest singular values, thus controlling the effect of the product $W*$ on the resulting matrix.

The compression achieved is therefore defined by the number of singular values eliminated. The more elements that are eliminated in the decomposition, the greater the compression. However, it should be noted that the elimination of singular values has a direct effect on the accuracy of the reconstructed $W*$ matrix, and therefore on the accuracy that the network provides when the reconstructed matrices are used as weights. The effect of these techniques must therefore be assessed in a trade-off between compression and the accuracy of the network once compressed. The value of these techniques will therefore depend on the margin of error that can be tolerated in the network results.

The different alternatives that have been considered for compressing neural networks using this factorization are shown below.

Matrix partitioning The first technique proposed to be combined with the SVD factorization, and thus to improve the understanding of the network, consists in partitioning the weight matrices into blocks, so that the factorization is applied to each resulting block [47, 50]. In this sense, we have studied the effect obtained depending on the dimensions of the blocks into which the matrix is partitioned, checking the effect of using square blocks of fixed size, blocks of rows (i.e. blocks whose number of columns coincides with the number of columns of the weight matrix), blocks of columns (where the number of rows of the block coincides with that of the matrix) and blocks per position in the filter layer. This technique has been implemented in PyDTNN and initial results show that selecting blocks of rows usually gives better results. Also, although they do not generally give such good results, there are certain square block sizes that give high compression with little loss of accuracy.

Matrix reordering Another technique that can be combined with matrix partitioning is to first apply a transformation that swaps both rows and columns. The purpose of this rearrangement is that when the partitioning is applied to the resulting matrix, the blocks will have a greater linear dependency and can therefore achieve a higher compression capacity. A simple example is shown in Figure 6.1. In a first study, the properties of different algebraic techniques and

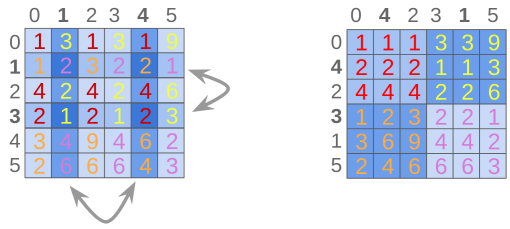

Figure 6.1: Example of reorganizing a matrix (left) to have blocks with greater linear dependence (right).

metrics were investigated to find the optimal partitions. However, the dependence between the complete matrix and the techniques considered has made it difficult to find a method that, without being extremely time-consuming, offers a reorganization that maximizes the linear dependence within each future partitioned block of the matrix, i.e. within each block resulting from the partitioning of the reorganized matrix. In this line, as a possible alternative, it is also proposed to design a heuristic that provides a reorganization of the matrices that, although not the optimal one, significantly increases the dependencies. In this sense, articles related to similar matrix compression techniques have been reviewed, such as [51].

Layer Separation This last proposed strategy is a major departure from the previous ones. While the previous techniques were applied only to the weight matrix of each layer individually, this technique is applied to the model itself, since it consists of separating each of the layers that use weights (full connection and convolutional layers) into two layers whose weights are derived from the SVD factorization [52]. This technique therefore modifies the model, which has certain drawbacks since we are no longer working with the same network, but it opens

the possibility of tolerating a greater loss of accuracy at the cost of a subsequent retraining of the network. The integration of this strategy in PyDTNN has been started, and first results have already been obtained for the separation of FC layers.

Combination of Techniques Finally, note that the above strategies can be combined with each other, as well as with other known compression techniques such as quantization [39]. In this way, the compressibility of the weights could be improved. The alternative for future work would be to perform a first layer separation with retraining. Once the new model is obtained with the retrained weights, the technique of row and column reorganization is used to increase the linear dependence of the blocks and finally partition the reorganized matrix. Tests have also been carried out combining some of these strategies with quantization, with positive results, so it is expected that the simultaneous use of these techniques will provide significant compression in the future.

Apéndice A

Algoritmos del *AllReduce*

A continuación se muestran gráficamente unos esquemas que pretenden mostrar el proceso interno a la comunicación AllReduce para los algoritmos LIN, RNG, RSA, RDB.

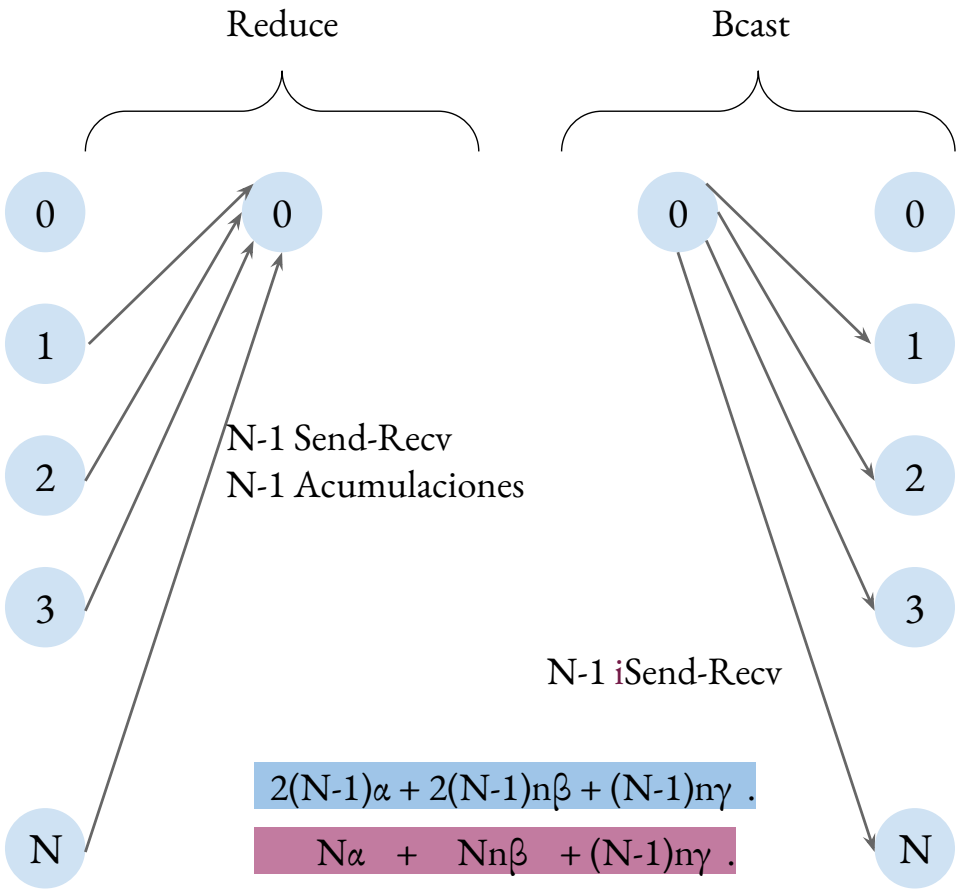

Figura A.1: Esquema del algoritmo *basic linear* de la primitiva *AllReduce*.

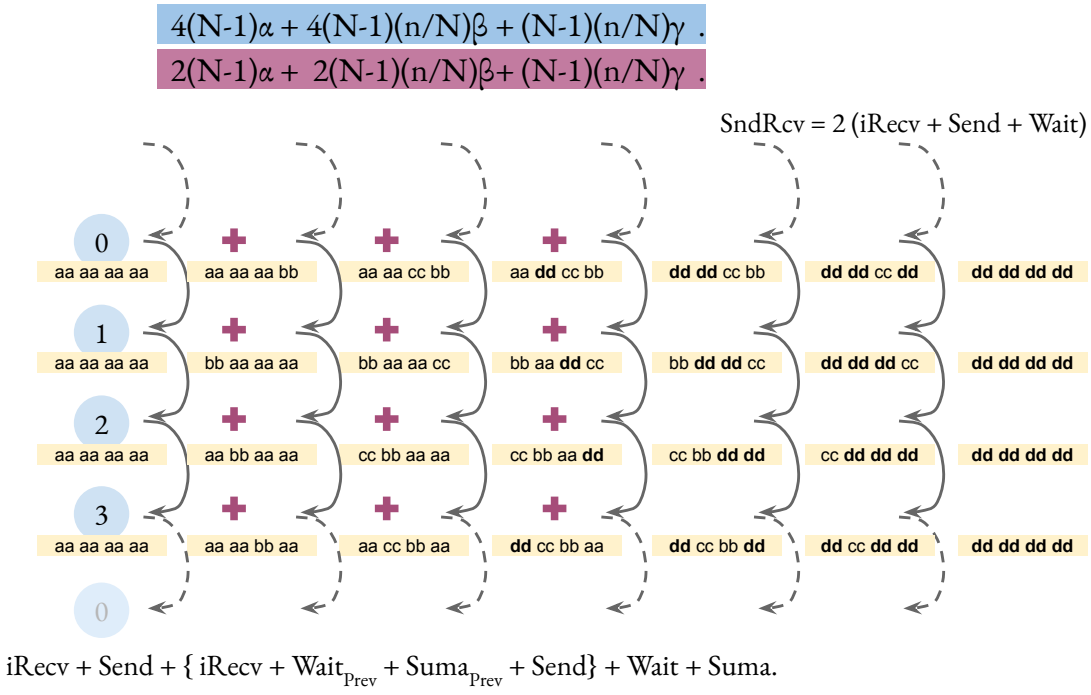

Figura A.2: Esquema del algoritmo *ring* de la primitiva *AllReduce*.

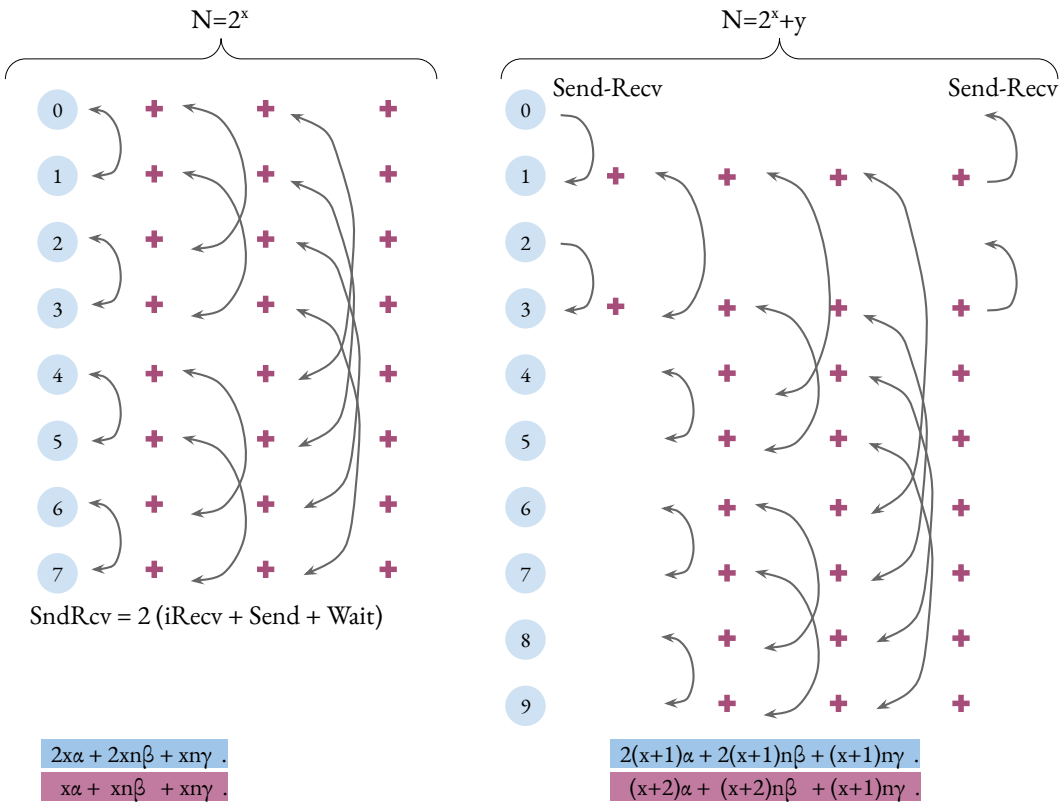

Figura A.3: Esquema del algoritmo *Rabenseifner* de la primitiva *AllReduce*.

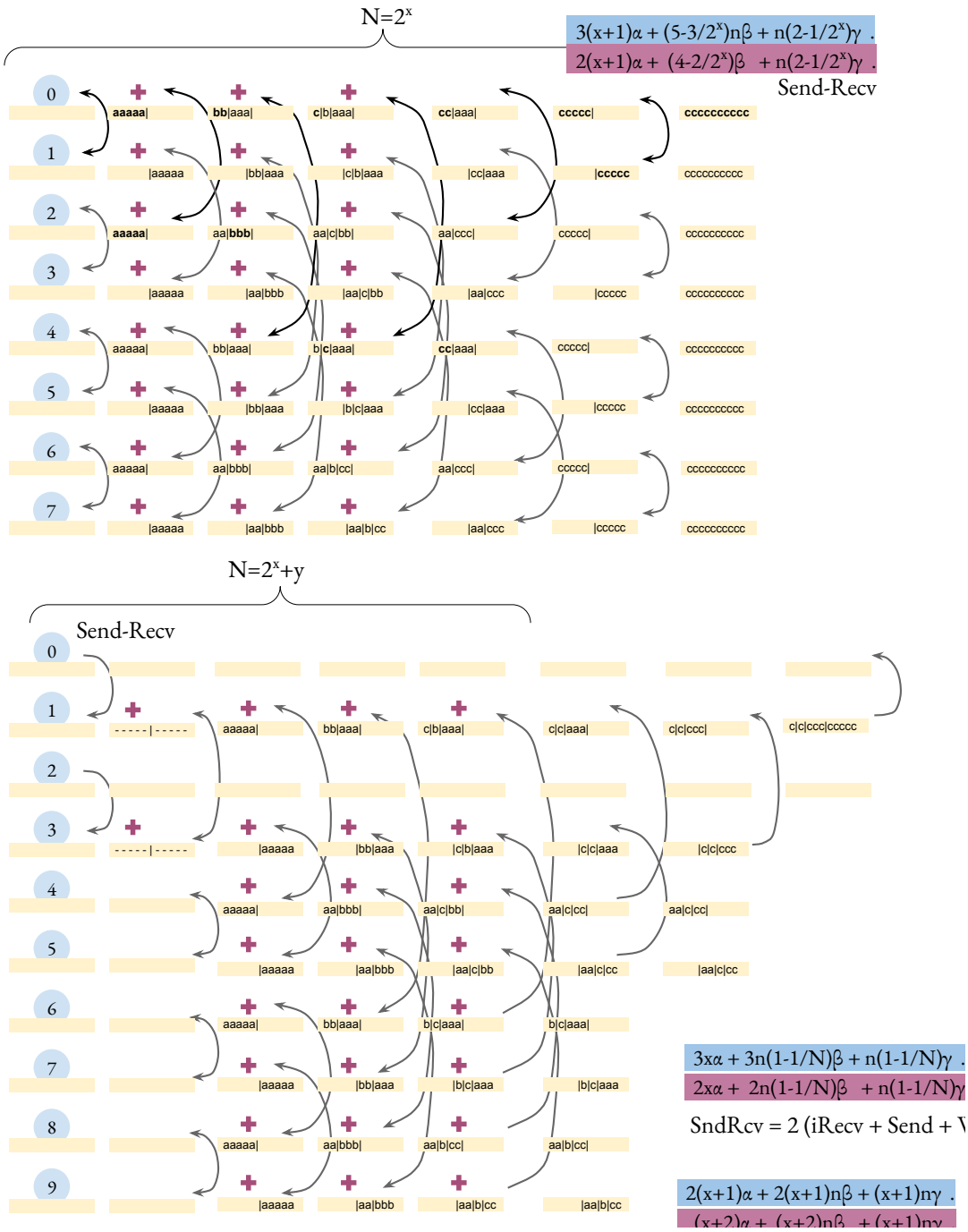

Figura A.4: Esquema del algoritmo *recursive doubling* de la primitiva *AllReduce*.

Bibliografía

[1] Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G.S., Davis, A., Dean, J., Devin, M. *et al.* (2016). Tensorflow: A system for large-scale machine learning. *arXiv preprint arXiv:1605.08695*.

[2] Ali, M., Yin, B., Kunar, A., Sheikh, A.M. & Bilal, H. (2020). Reduction of multiplications in convolutional neural networks. In *2020 39th Chinese control conference (CCC)*, 7406–7411, IEEE.

[3] Ananthaswamy, A. (2019). Faced with a data deluge, astronomers turn to automation. [Online; accessed 26-April-2023].

[4] Barrachina, S., Castelló, A., Catalán, M., Dolz, M.F. & Mestre, J.I. (2021). A flexible research-oriented framework for distributed training of deep neural networks. In *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 730–739, IEEE.

[5] Barrachina, S., Castelló, A., Catalán, M., Dolz, M.F. & Mestre, J.I. (2021). Pydtnn: a user-friendly and extensible framework for distributed deep learning. *The Journal of Supercomputing*, **77**, 9971–9987.

[6] Barrachina, S., Castelló, A., Catalán, M., Dolz, M.F., Mestre, J.I., Ramírez, C. & Rodríguez, D. (2021). Pydtnn: Entorno para entrenamiento e inferencia con redes neuronales profundas. In *Jornadas SARTECO 2021*.

[7] Barrachina, S., Castelló, A., Catalán, M., Dolz, M.F. & Mestre, J.I. (2023). Using machine learning to model the training scalability of convolutional neural networks on clusters of gpus. *Computing*, **105**, 915–934.

[8] Behnel, S., Bradshaw, R., Citro, C., Dalcín, L., Seljebotn, D.S. & Smith, K. (2011). Cython: The best of both worlds. *Computing in Science & Engineering*, **13**, 31–39.

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

[10] Berry, M., Potok, T.E., Balaprakash, P., Hoffmann, H., Vatsavai, R. *et al.* (2015). Machine learning and understanding for intelligent extreme scale scientific computing and discovery. doe workshop report, january 7–9, 2015, rockville, md. Tech. rep., USDOE Office of Science (SC), Washington, DC (United States). Advanced ....

[11] Bruce, P. (2019). A deep dive into deep learning. [Online; accessed 26-April-2023].

[12] Castelló, A., Dolz, M.F., Quintana-Ortí, E.S. & Duato, J. (2019). *Analysis of Model Parallelism for Distributed Neural Networks*. EuroMPI '19, Association for Computing Machinery, New York, NY, USA.

[13] Castelló, A., Dolz, M.F., Quintana-Ortí, E.S. & Duato, J. (2019). Theoretical scalability analysis of distributed deep convolutional neural networks. In *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, 534–541, IEEE.

[14] Castelló, A., Catalán, M., Dolz, M.F. & Quintana-Ortí, E.S. (2020). Pydtnn: An extensible and user-friendly python library for distributed deep learnings. In *20th International Conference Computational and Mathematical Methods in Science and Engineering*.

[15] Castelló, A., Catalán, M., Dolz, M.F., Mestre, J.I., Quintana-Ortí, E.S. & Duato, J. (2021). Evaluation of mpi allreduce for distributed training of convolutional neural network. In *2021 29th Euromicro international conference on parallel, distributed and network-based processing (PDP)*, IEEE.

[16] Castelló, A., Catalán, M., Dolz, M.F., Mestre, J.I., Quintana-Ortí, E.S. & Duato, J. (2021). Performance modeling for distributed training of convolutional neural networks. In *2021 29th Euromicro international conference on parallel, distributed and network-based processing (PDP)*, 99–108, IEEE.

[17] Castelló, A., Catalán, M., Dolz, M.F., Quintana-Ortí, E.S. & Duato, J. (2021). Mejora de los modelos predictivos para la comunicación colectiva mpi allreduce. In *Jornadas SARTECO 2021*.

[18] Castelló, A., Catalán, M., Dolz, M.F., Quintana-Ortí, E.S. & Duato, J. (2023). Analyzing the impact of the mpi allreduce in distributed training of convolutional neural networks. *Computing*, **105**, 1101–1119.

[19] Castelvecchi, D. *et al.* (2015). Artificial intelligence called in to tackle lhc data deluge. *Nat.*, **528**, 18–19.

[20] Chan, E., Heimlich, M., Purkayastha, A. & van de Geijn, R. (2007). Collective communication: Theory, practice, and experience: Research articles. *Concurr. Comput. : Pract. Exper.*, **19**, 1749–1783.

[21] Chellapilla, K., Puri, S. & Simard, P. (2006). High performance convolutional neural networks for document processing. In *International Workshop on Frontiers in Handwriting Recognition*, available as INRIA report inria-00112631 from `https://hal.inria.fr/inria-001126`.

[22] Chollet, F. (Accedido en 2023). Keras: Theano-based deep learning library. *GitHub repository*.

[23] Corporation, I. (2017). Tame the data deluge. https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/analytics-tame-the-data-deluge-whitepaper-fv2.pdf.

[24] Demmel, J. (2012). Communication avoiding algorithms. In *Proceedings of the 2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, SCC '12, 1942–2000, IEEE Computer Society, USA.

[25] Deng, J., Dong, W., Socher, R., Li, L.J., Li, K. & Fei-Fei, L. (2015). Imagenet large scale visual recognition challenge. *International Journal of Computer Vision*, **115**, 211–252.

[26] Dongarra, J., Huss-Lederman, S., Otto, S., Snir, M. & Walker, D. (1996). Mpi: The complete reference.

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

[28] editorial team, I. (2017). The exponential growth of data. https://insidebigdata.com/2017/02/16/the-exponential-growth-of-data/.

[29] Gabriel, E., Fagg, G.E., Bosilca, G., Angskun, T., Dongarra, J.J., Squyres, J.M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A. *et al.* (2008). Open mpi: A high-performance message passing interface. *Concurrency and Computation: Practice and Experience*, **20**, 203–229.

[30] Gholami, A., Azad, A., Jin, P., Keutzer, K. & Buluc, A. (2018). Integrated model, batch, and domain parallelism in training neural networks. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, SPAA '18, 77–86, Association for Computing Machinery, New York, NY, USA.

[31] Harlap, A., Narayanan, D., Phanishayee, A., Seshadri, V., Devanur, N., Ganger, G. & Gibbons, P. (2018). Pipedream: Fast and efficient pipeline parallel dnn training. *arXiv preprint arXiv:1806.03377*.

[32] Hasanov, K. & Lastovetsky, A. (2017). Hierarchical redesign of classic MPI reduction algorithms. *J. Supercomputing*, **73**, 713–725.

[33] He, K., Zhang, X., Ren, S. & Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 770–778.

[34] Hennessy, J.L. & Patterson, D.A. (2019). A new golden age for computer architecture. *Commun. ACM*, **62**, 48–60.

[35] Hey, A.J., Tansley, S., Tolle, K.M. *et al.* (2009). *The fourth Paradigm: data-intensive scientific discovery*, vol. 1. Microsoft research Redmond, WA.

[36] Huang, G., Liu, Z., van der Maaten, L. & Weinberger, K.Q. (2017). Densely connected convolutional networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 4700–4708.

[37] Inc., G. (Accedido en 2023). Tensorflow benchmarks.

[38] Ivanov, A., Dryden, N., Ben-Nun, T., Li, S. & Hoefler, T. (2020). Data movement is all you need: A case study on optimizing transformers.

[39] Jacob, B., Kligys, S., Chen, B., Zhu, M., Tang, M., Howard, A. & Adam, H. (2018). Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2704–2713.

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

[41] Karymov, S. & Addair, T. (2017). Horovod: Distributed deep learning in tensorflow. `https://github.com/horovod/horovod`.

[42] Ketkar, N. (2017). Introduction to pytorch. In *Deep learning with python*, 195–208, Springer.

[43] Krizhevsky, A. & Hinton, G. (2009). Learning multiple layers of features from tiny images. *Technical report, University of Toronto*.

[44] Krizhevsky, A., Sutskever, I. & Hinton, G.E. (2017). Imagenet classification with deep convolutional neural networks. *Communications of the ACM*, **60**, 84–90.

[45] Krizhevsky, A., Sutskever, I. & Hinton, G.E. (2017). Imagenet classification with deep convolutional neural networks. *Communications of the ACM*, **60**, 84–90.

[46] Kurth, T., Zhang, J., Satish, N., Racah, E., Mitliagkas, I., Patwary, M.M.A., Malas, T., Sundaram, N., Bhimji, W., Smorkalov, M. *et al.* (2017). Deep learning at 15pf: supervised and semi-supervised classification for scientific data. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 1–11.

[47] Lavin, A. & Gray, S. (2016). Fast algorithms for convolutional neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 4013–4021.

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

[49] LeCun, Y., Bengio, Y. & Hinton, G. (2015). Deep learning. *Nature*, **521**, 436–444.

[50] Lee, D., Kwon, S.J., Kim, B. & Wei, G.Y. (2019). Learning low-rank approximation for cnns. *arXiv preprint arXiv:1905.10145*.

[51] Levitt, J. & Martinsson, P.G. (2022). Randomized compression of rank-structured matrices accelerated with graph coloring. *arXiv preprint arXiv:2205.03406*.

[52] Maji, P. & Mullins, R. (2018). On the reduction of computational complexity of deep convolutional neural networks. *Entropy*, **20**, 305.

[53] Maji, P., Mundy, A., Dasika, G., Beu, J., Mattina, M. & Mullins, R. (2019). Efficient winograd or cook-toom convolution kernel implementation on widely used mobile cpus. In *2019 2nd Workshop on Energy Efficient Machine Learning and Cognitive Computing for Embedded Applications (EMC2)*, 1–5, IEEE.

[54] Najafabadi, M.M., Villanustre, F., Khoshgoftaar, T.M., Seliya, N., Wald, R. & Muharemagic, E. (2015). Deep learning applications and challenges in big data analytics. *Journal of big data*, **2**, 1–21.

[55] Nuriyev, E. & Lastovetsky, A. (2021). A new model-based approach to performance comparison of mpi collective algorithms. In V. Malyshkin, ed., *Parallel Computing Technologies*, 11–25, Springer International Publishing, Cham.

[56] NVIDIA (Accedido en 2023). The NVIDIA Collective Communication Library (NCCL). `https://developer.nvidia.com/nccl`.

[57] NVIDIA Corporation (2017). Nvidia tesla v100 gpu architecture. `https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf.`, accessed: 2023-04-30.

[58] OpenMP Architecture Review Board (2023). Openmp application program interface, version 5.0. `https://www.openmp.org/specifications/`.

[59] Parashar, A., Raina, P., Shao, Y.S., Chen, Y.H., Ying, V.A., Mukkara, A., Venkatesan, R., Khailany, B., Keckler, S.W. & Emer, J. (2019). Timeloop: A systematic approach to dnn accelerator evaluation. In *2019 IEEE international symposium on performance analysis of systems and software (ISPASS)*, 304–315, IEEE.

[60] Park, J., Naumov, M., Basu, P., Deng, S., Kalaiah, A., Khudia, D., Law, J., Malani, P., Malevich, A., Nadathur, S. *et al.* (2018). Deep learning inference in facebook data centers: Characterization, performance optimizations and hardware implications. *arXiv preprint arXiv:1811.09886*.

[61] Pouyanfar, S., Sadiq, S., Yan, Y., Tian, H., Tao, Y., Reyes, M.P., Shyu, M.L., Chen, S.C. & Iyengar, S.S. (2018). A survey on deep learning: Algorithms, techniques, and applications. *ACM Comput. Surv.*, **51**, 92:1–92:36.

[62] Reuther, A., Michaleas, P., Jones, M., Gadepally, V., Samsi, S. & Kepner, J. (2022). AI and ML accelerator survey and trends. In *2022 IEEE High Performance Extreme Computing Conference*, IEEE.

[63] Schmitt, C., Cox, S., Fecho, K., Idaszak, R., Lander, H., Rajasekar, A. & Thakur, S. (2015). Scientific discovery in the era of big data: More than the scientific method. *RENCI White Paper*, **3**.

[64] Sergeev, A. & Del Balso, M. (2018). Horovod: fast and easy distributed deep learning in tensorflow. *arXiv preprint arXiv:1802.05799*.

[65] Shazeer, N., Cheng, Y., Parmar, N., Tran, D., Vaswani, A., Koanantakool, P., Hawkins, P., Lee, H., Hong, M., Young, C. *et al.* (2018). Mesh-tensorflow: Deep learning for supercomputers. *Advances in neural information processing systems*, **31**.

[66] Simonyan, K. & Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*.

[67] Stevens, R., Taylor, V., Nichols, J., Maccabe, A.B., Yelick, K. & Brown, D. (2020). AI for Science: Report on the Department of Energy (DOE) Town Halls on Artificial Intelligence (AI) for Science. Tech. rep., U.S. Department of Energy Office of Scientific and Technical Information.

[68] Sze, V., Chen, Y.H., Yang, T.J. & Emer, J.S. (2017). Efficient processing of deep neural networks: A tutorial and survey. *Proceedings of the IEEE*, **105**, 2295–2329.

[69] Sze, V., Chen, Y.H., Yang, T.J. & Emer, J.S. (2017). Efficient processing of deep neural networks: A tutorial and survey. *Proceedings of the IEEE*, **105**, 2295–2329.

[70] Thakur, R., Rabenseifner, R. & Gropp, W. (2005). Optimization of collective communication operations in MPICH. *The International Journal of High Performance Computing Applications*, **19**, 49–66.

[71] TOP500 Organization (Accedido en 2023). Top500 supercomputer sites. `https://www.top500.org/`.

[72] Vasudevan, A., Anderson, A. & Gregg, D. (2017). Parallel multi channel convolution using general matrix multiplication. In *2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 19–24.

[73] Williams, S., Waterman, A. & Patterson, D. (2009). Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, **52**, 65–76.

[74] Wu, C.J., Brooks, D., Chen, K., Chen, D., Choudhury, S., Dukhan, M., Hazelwood, K., Isaac, E., Jia, Y., Jia, B. *et al.* (2019). Machine learning at facebook: Understanding inference at the edge. In *2019 IEEE international symposium on high performance computer architecture (HPCA)*, 331–344, IEEE.

[75] Yamazaki, M., Kasagi, A., Tabuchi, A., Honda, T., Miwa, M., Fukumoto, N., Tabaru, T., Ike, A. & Nakashima, K. (2019). Yet another accelerated sgd: Resnet-50 training on imagenet in 74.7 seconds. *arXiv preprint arXiv:1903.12650*.

[76] You, Y., Gitman, I. & Ginsburg, B. (2017). Scaling SGD batch size to 32k for ImageNet training. ArXiv:1708.03888.

[77] You, Y., Demmel, J., Keutzer, K., Hsieh, C.J., Ying, C. & Hseu, J. (2018). Large-batch training for LSTM and beyond. Tech. Rep. UCB/EECS-2018-138, Electrical Engineering and Computer Sciences, University of California at Berkeley.

[78] Zhang, J. & Zong, C. (2015). Deep neural networks in machine translation: An overview. *IEEE Intelligent Systems*, **30**, 16–25.

Glosario

AP Actualización de Pesos.

BLAS del inglés *Basic Linear Algebra Subprograms*.

BLIS del inglés *BLAS-like Library Instantiation Software*.

BP del inglés *Backward Pass*.

CG Cálculo de Gradientes.

CIFAR-10 del inglés *Canadian Institute for Advanced Research, 10 classes*.

CNN del inglés *Convolutional Neural Network*.

CPU del inglés *Central Processing Unit*.

DL del inglés *Deep Learning*.

FC del inglés *Fully Connected*.

flop del inglés *floating operation*.

FLOPS del inglés *Floating Operations Per Second*.

FP del inglés *Forward Pass*.

GA Gamma Adaptado.

GC Gamma Continuo.

GEMM del inglés *General Matrix Multiply*.

GPU del inglés *Graphics Processing Unit*.

HPC del inglés *High Performance Computing*.

HWD Horovod.

IA Inteligencia Artificial.

Intel-MPI del inglés *Intel Message Passing Interface*.

LIN del inglés *Linear*.

LR del inglés *Learning Rate*.

MA Modelo Asíncrono: contemplando que las comunicaciones se solapan entre sí.

memop del inglés *memory operation*.

ML del inglés *Machine Learning*.

MLP del inglés *Multi Layer Perceptron*.

MM Modelo Mixto: contemplando que parte de las comunicaciones se solapan entre sí.

MNIST del inglés *Modified National Institute of Standards and Technology*.

MPI del inglés *Message Passing Interface*.

MS Modelo Síncrono: contemplando que las comunicaciones no se solapan enter sí.

Open-MPI del inglés *Open Message Passing Interface*.

RDB del inglés *Recursive Doubling*.

ResNet del inglés *Residual Network*.

RNG del inglés *Ring*.

RNP Redes Neuronales Profundas.

RSA Rabenseifner.

SGD del inglés *Stochastic Gradient Descent*.

SRG del inglés *Segmented Ring*.

TF TensorFlow.

TPU del inglés *Tensor Processing Unit*.

VGG del inglés *Visual Geometry Group*.