Facultad de Ciencias

PARALELIZACIÓN DEL ENTRENAMIENTO DE REDES NEURONALES EN SISTEMAS HETEROGÉNEOS

Parallelization of the training of neural networks in heterogeneous systems

Trabajo de Fin de Grado
para acceder al

GRADO EN INGENIERÍA INFORMÁTIA

Autor: Luis Cruz Varona

Director: Jose Luis Bosque Orero

Julio - 2021

Índice general

Resumen

En los últimos años el uso de las redes neuronales ha incrementado drásticamente, debido a su capacidad de adaptación a infinidad de problemas. Sin embargo, para que una red neuronal funcione correctamente es necesario un proceso de entrenamiento previo, donde ésta aprende a identificar patrones en los datos que recibe. Este proceso es muy largo y computacionalmente intensivo, debido a las operaciones que se realizan y a que es necesaria una gran cantidad de datos de ejemplo. Para mitigar esto, han ido surgiendo a lo largo del tiempo varias soluciones que reducen el tiempo, la complejidad y el consumo energético del proceso de entrenamiento. Una de estas soluciones es el uso de aceleradores dedicados como GPUs, en vez de procesadores convencionales. Esto se debe a su velocidad a la hora de realizar ciertas operaciones y a su excelente eficiencia energética respecto a las CPUs tradicionales. Sin embargo, debido a la creciente complejidad de las redes neuronales y de los problemas a los que éstas se aplican, el uso de un solo acelerador para entrenarlas se ha vuelto insuficiente. Es por esto por lo que es necesario paralelizar el entrenamiento de redes neuronales, distribuyendo la carga de trabajo entre los dispositivos disponibles de manera que se optimice el rendimiento. Existen una gran variedad de técnicas de paralelización de redes neuronales y cada framework de ML proporciona sus propias estrategias de paralelización, con lo que es difícil saber cuál es la más beneficiosa para cada situación y cuáles son sus efectos.

En este proyecto se estudia cuál es exactamente el impacto que tienen estas técnicas de paralelización en el proceso de entrenamiento de una red neuronal. Para ello, se han evaluado varios frameworks de redes neuronales y sus estrategias de paralelización, eligiendo el más adecuado para este proyecto. Además, se ha desarrollado un benchmark en uno de estos frameworks, Pytorch. Este benchmark entrena un modelo ResNet-34 usando un dataset de clasificación de imágenes, grabando varias métricas del proceso, como la duración del mismo y de cada una de sus fases, la evolución de la precisión del modelo a lo largo del tiempo o la precisión final obtenida. Para poder realizar un estudio en profundidad, se han diseñado y realizado experimentos en torno a este benchmark, ejecutándolo en varios ambientes de paralelización: usando solo la CPU, usando una GPU, usando varias GPUs de manera paralela, etc; y guardando sus salidas, además del consumo energético del benchmark. También se propone y estudia un modelo de paralelización híbrido, que explota tanto las GPUs como los CPUs disponibles para entrenar un modelo de red neuronal distribuyendo para ello una copia del modelo a cada dispositivo y parte de los datos de entrenamiento, determinando al final si es viable o no. Los resultados obtenidos de los experimentos han sido positivos, se ha obtenido una escalabilidad casi lineal para el tiempo de la parte paralelizada; además, el consumo energético no se ha visto incrementado significativamente como resultado de la paralelización, obteniendo casi el doble de eficiencia energética respecto del entrenamiento no paralelizado.

Abstract

In recent times, Artificial Neural Networks have seen a significant increase in their use, due to their flexibility and adaptability to a myriad of tasks. However, in order for a neural network to work correctly, a training period is necessary, where the model learns to identify certain patterns in the input data it receives. This process is long and computationally expensive, due to the complexity of its operations and the sheer amount of training data required. To mitigate this, some techniques have appeared over the years that attempt to reduce training time, complexity and energy consumption. One of the most commonly used approaches is the use of dedicated accelerators, such as GPUs, instead of conventional processors. This is because of their higher speed and energy efficiency regarding some tasks relative to general purpose processors. However, due to the rising complexity of neural networks and the problems the attempt to solve, the use of a single accelerator has become insufficient. This has made the use of several accelerators in parallel necessary, distributing the workload equally between them in order to optimize performance. Several parallelization techniques exist nowadays and almost every ML framework implements its own distribution strategies, so knowing which is best for each situation and its effects has become a very difficult task.

In this project, a benchmark is proposed to study the impact of these techniques, recording for that several metrics, such as training time or model precision. For this, an evaluation of several neural network frameworks is conducted, studying their parallelization strategies, picking one of them to use for the rest of the project. From this framework, a benchmark is created that trains a ResNet-34 model on an image classification dataset, measuring certain variables such as end-to-end training time, evolution of the model's precision over time or final model precision. To gain more insight into these metrics, various experiments have been designed and conducted around this benchmark, using each of them in a different execution environment: using only CPU, using 1 GPU, using multiple GPUs in parallel, etc.; documenting not only the output from the benchmark but its energy consumption as well, in order to evaluate energy efficiency. A hybrid parallelization model is also proposed, in which the available GPUs are used in conjunction with the CPU to train the network, giving each of the components a copy of the model and a subset of the data, and evaluating afterwards its effectiveness and viability. The results obtained from these experiments are very positive, the scalability of the distributed model is almost lineal regarding the parallelized part; on top of that, energy consumption has not seen a significant increase as a result of the parallelization, meaning the energy efficiency of this paradigm is almost double the non-distributed training.

Capítulo 1

Introducción

1.1. Redes Neuronales y aceleradores hardware

1.1.1. Redes Neuronales

En los últimos años, la inteligencia artificial ha aumentado en popularidad como tecnología para resolver ciertos problemas. Uno de los modelos de inteligencia artificial cuyo uso ha aumentado más son las redes neuronales. Esto se debe a que son muy flexibles y adaptables a numerosos ambientes, gracias a su capacidad para aprender y extraer patrones del conjunto de datos que se les proporcionan como entrada.

Es por esto que las redes neuronales han sido empleadas resolver una gran cantidad de problemas. Entre sus usos se encuentran aplicaciones para situaciones tan diversas como la predicción del consumo de agua de tomates en un invernadero [12], el modelado de las emisiones de CO_2 de ciertos países [49], la mejora de la claridad de fotografías con poca luz [53] o incluso la generación de imágenes artificiales [30] (como caras de personas que no existen [57]).

Incluso una red simple, con una sola capa de entrada y una de salida, es lo suficientemente potente como para aprender a identificar números manuscritos con una precisión del 75 % [32], lo que demuestra como, aún en su forma más humilde, las redes neuronales son una herramienta poderosa para la resolución de ciertos problemas. Para que una red alcance tal precisión es necesario entrenarla previamente, enseñándole grandes cantidades de datos para que pueda "aprender" a extrapolar dichos partrones de los ejemplos que se le dan, diciéndole en cada caso si ha acertado o no y cuánto se ha "equivocado". Este proceso puede llegar a ser muy largo, dependiendo de muchos factores, entre ellos la arquitectura y tamaño de la red, la complejidad del problema a resolver y el hardware disponible para entrenar al modelo.

A pesar de su explosión en uso en los últimos años, la tecnología detrás de las redes neuronales es relativamente antigua. Los primeros componentes de la red neuronal (como la neurona artificial), fueron propuestos matemáticamente alrededor de los años 50, surgiendo en 1974 la primera red neuronal, el perceptron, capaz de aprender a identificar patrones gracias al algoritmo de propagación hacia atrás [56].

Sin embargo, debido a la gran cantidad de operaciones y datos requeridos para entrenar una red neuronal, su uso fue relativamente bajo, y cuando se usaban, se implementaban redes muy simples y con pocas neuronas, ya que el hardware no era lo suficientemente potente como para ejecutar modelos más complejos. Esto ha cambiado en la última década debido a los avances que se han dado tanto en aceleradores dedicados a tareas de machine learning como en sistemas heterogéneos[42] [41].

1.1.2. Aceleradores hardware y sistemas heterogéneos

Uno de los aceleradores más usados en la actualidad es la GPU. Las GPUs (*Graphics Processing Unit*) son componentes hardware especializados en acelerar las operaciones relacionadas con gráficos por computador. Originalmente surgieron en los años 90 y durante muchos años se utilizaron para acelerar videojuegos y programas destinados a la creación de gráficos 3D (animación, modelado CAD, ...). Sin embargo, debido al gran número de cores que contienen, se han comenzado a usar en los últimos años para otras aplicaciones que requieren un alto grado de paralelismo (como multiplicación de matrices, por ejemplo); llegando a aparecer GPUs sin salida de pantalla, destinadas al uso en servidores [40].

Una GPU puede acelerar considerablemente el proceso de entrenamiento de una red neuronal sencilla; sin embargo, con la creación de modelos más grandes destinados a resolver problemas más complejos, la potencia de una GPU empieza a dejar de ser suficiente. Por esto, surgen los sistemas heterogéneos con varios aceleradores hardware.

Los sistemas heterogéneos son equipos que tienen varios procesadores. Generalmente contienen uno de propósito general (una CPU) y varios de carácter más específico, destinados a acelerar el cómputo de ciertas operaciones o tareas [33] [54] [7]. Estos aceleradores pueden ser aceleradores gráficos (GPUs), altamente reconfigurables (FPGAs) [24] o aceleradores destinados a tareas muy específicas (ASICs), como los TPU (*Tensor Processing Unit*) de google - procesadores especializados en acelerar la ejecución de redes neuronales [4]. Estos sistemas pueden mejorar aún más el rendimiento del proceso de entrenamiento, y su consumo energético, debido a la eficiencia energética de sus aceleradores respecto a los CPUs tradicionales. Gracias a esto, han surgido arquitecturas de redes neuronales más grandes; un ejemplo notable son las DNN (*Deep Neural Networks*), redes neuronales con muchas capas intermedias, usadas hoy en día para la resolución de la mayoría de los problemas orientados a *Deep Learning*.

Sin embargo, para maximizar el uso de estos sistemas heterogéneos con varios aceleradores es necesario paralelizar el proceso de entrenamiento, ejecutándolo en varios dispositivos a la vez. Esto plantea una serie de retos que es necesario evaluar, como:

- **El modelo de paralelización más adecuado**: Existen varios modos de paralelizar el proceso de entrenamiento, como partir la red neuronal en varios trozos, asignando a cada dispositivo una parte de la red; o dar a cada dispositivo una copia del modelo, partiendo entonces los datos de entrada y entregando una fracción a cada dispositivo.
- **El impacto de la paralelización en el resultado del entrenamiento:** Es necesario estudiar si la precisión final de una red neuronal se ve afectada por el proceso de paralelización, y si es así, en qué medida y de qué maneras se puede mitigar esto.
- **La gestión de la comunicación de los dispositivos:** En todo sistema paralelo es necesario gestionar la comunicación entre los diferentes dispositivos. En la paralelización de redes neuronales esto es más importante todavía, ya que la comunicación sirve para mantener el modelo de red neuronal consistente durante todo el proceso de entrenamiento; de no hacerlo, esto podría afectar al resultado final, pudiendo dejar a la red inservible. Sin embargo, la comunicación entre los dispositivos utilizados puede introducir una carga de trabajo significativa, que limite las ganancias que se obtienen de paralelizar el entrenamiento.
- **Cómo afecta el uso de múltiples dispositivos al consumo energético:** Hay que valorar cuál es el impacto que tiene la paralelización en el costo energético del proceso de entrenamiento. Al aumentar el número de dispositivos utilizados, aumenta el uso de potencia del proceso, pero a la vez se reduce la duración de éste, con lo que es necesario estudiar si estas dos métricas se compensan lo suficiente como para mejorar el consumo energético del sistema o su eficiencia.

1.2. Objetivos

El objetivo principal de este proyecto es verificar si la paralelización del proceso de entrenamiento en entornos multi-GPU obtiene ganancias tanto en rendimiento como en consumo energético. Además se debe determinar qué método de paralelización es más adecuado para esta tarea, ya que existen diferentes alternativas (como partir la red neuronal entre los dispositivos, o partir los datos) con diferentes características, impactando determinados aspectos del proceso, como la comunicación entre dispositivos. Por ello, será necesario hacer uso de diferentes ambientes de ejecución:

- Ejecución del problema solo en CPU
- Ejecución en una GPU
- Ejecución en una GPU usando la plataforma de paralelización
- Ejecución en dos GPUs

Para este fin es necesario realizar los siguientes sub-objetivos:

- Realizar un estudio de los diferentes frameworks de redes neuronales que perminten la paralelización del entrenamiento de un modelo en sistemas heterogéneos; seleccionando, al concluir, el que se considere más adecuado para el proyecto. En este estudio se valoran aspectos como la facilidad de uso, la flexibilidad del framework o los dispositivos compatibles con el mismo.
- Desarrollar un programa que realice el entrenamiento de una red neuronal. Dicho programa debe ser representativo de la realidad y, a la vez, debe producir una carga de trabajo lo suficientemente alta en el hardware a emplear. Además, debe terminar en un tiempo razonable, ya que se va a utilizar para medir el rendimiento del proceso, para lo que es necesario hacer varias ejecuciones del programa. Éste debe soportar la paralelización del entrenamiento en varias GPUs, usando para ello las herramientas disponibles en el framework de redes neuronales escogido.
- Crear experimentos que midan las diferentes facetas del entrenamiento de la red, teniendo en cuenta métricas como el tiempo de ejecución, la precisión al final del proceso o la energía consumida por los componentes utilizados. Cada uno de los experimentos debe representar un ambiente de ejecución: desde entrenar usando solo la CPU, hasta entrenar usando varias GPUs de manera paralela.

1.3. Plan de trabajo

Para el desarrollo del proyecto se han planificado varias fases, comenzando por un estudio previo de dos de los diferentes frameworks de redes neuronales (Tensorflow y Pytorch) que existen, evaluando sus características y desarrollando programas de prueba, para poder determinar cuál utilizar. Ambos frameworks proponen diferentes maneras de abordar la paralelización de redes neuronales, ya sea partiendo la red neuronal o los datos entre los dispositivos disponibles. Una vez seleccionado el framework a usar, se procederá a seleccionar un dataset que se ajuste a ciertos requisitos, como que el tamaño de sus muestras sea fijo o que estas muestras puedan usarse en blanco y negro. A continuación se elegirá un modelo de red neuronal que se especialice en tareas de visión[1] y que tenga un tamaño adecuado para el hardware a utilizar, para luego ser implementado en Pytorch.

Tras esto, se implementará un pequeño programa de pruebas que utilice una GPU para entrenar el modelo de red neuronal elegido. Asimismo, se procederá a realizar una investigación sobre la viabilidad de un paradigma de paralelización que haga uso tanto de los dispositivos GPU disponibles como de la CPU para entrenar redes neuronales.

[1] Problemas cuyos datos de entrada son imágenes. Entre estas tareas están la clasificación de muestras o la localización de objetos dentro de una muestra.

Una vez completada la implementación, se procederá a crear un benchmark que evalúe el rendimiento de una red neuronal durante su entrenamiento. Para ello, primero se formalizará la estructura del programa, basándose para ello en el código desarrollado durante la investigación anterior. Este programa entrenará una red neuronal haciendo uso de las dos GPUs disponibles en el equipo en el que se ejecutará, distribuyendo los datos de manera equitativa entre ellas. Además, será necesario la realización de ciertas pruebas de rendimiento (en las que se examinen el tiempo de ejecución del programa, así como los resultados que se obtengan) para determinar los parámetros óptimos a utilizar durante la realización de los experimentos finales.

Finalmente, terminado el desarrollo del programa de pruebas, se diseñarán los experimentos a realizar, implementando para cada uno una script que lance el benchmark las veces necesarias y tome las medidas apropiadas. Cada experimento guardará el rendimiento (tiempo de entrenamiento y precisión) de la red neuronal escogida, así como la potencia utilizada por los dispositivos empleados para el entrenamiento en los diferentes ambientes de paralelización (usando solo la CPU, usando una GPU y usando dos GPUs). Tras ejecutar los experimentos, se guardarán los datos obtenidos en un repositorio y se tratarán, para más tarde extraer conclusiones.

1.4. Estructura del documento

El resto de este documento sigue la siguiente estructura:

- En el capítulo 2 se explican los conceptos usados durante todo el trabajo, pasando por una breve explicación de las redes neuronales y su proceso de entrenamiento, las herramientas utilizadas para la realización del proyecto, y algunos trabajos relacionados con éste.
- En el capítulo 3 se explica en profundidad el proceso de desarrollo del proyecto de manera cronológica, detallando la investigación previa realizada, el proceso seguido para seleccionar tanto el dataset como el modelo de red neuronal utilizados, el desarrollo del modelo de paralelización híbrido que explota tanto las GPUs disponibles como el CPU y, por último, las pruebas realizadas para determinar algunos de los parámetros del programa.
- En el capítulo 4 se pasa a definir los experimentos realizados, explicando para ello los parámetros escogidos finalmente, el equipo en el que se han ejecutado las pruebas y la metodología utilizada. Tras ello, se pasan a mostrar y explicar los resultados obtenidos de los experimentos, dividiendo dichos resultados en tres categorías: tiempo de ejecución, precisión final y energía consumida.
- Finalmente, en el capítulo 5, se explican las conclusiones extraídas del trabajo realizado, así como los posibles trabajos futuros con los que ampliar este proyecto y profundizar en los resultados obtenidos.
- Tanto el código desarrollado como los resultados obtenidos de los experimentos se encuentran en sendos repositorios públicos de GitHub[2]

[2]Repositorio del código del proyecto: https://github.com/luiscruzv99/TFG-Repo
Repositorio de los resultados: https://github.com/luiscruzv99/TFG-Resultados

Capítulo 2

Background

Antes de comenzar a detallar el desarrollo del proyecto y sus resultados, es importante explicar algunos términos que se utilizarán durante el resto del trabajo, de manera que sea más fácil entenderlo. En concreto, se presentarán algunos conceptos de redes neuronales, además de las herramientas utilizadas y, por último, algunos trabajos previos relacionados con este proyecto.

2.1. Redes neuronales

Una red neuronal es un conjunto de neuronas artificiales distribuidas en capas [51]. Las redes neuronales tienen infinidad de usos, sin embargo, nos centraremos en su capacidad de reconocer patrones y clasificar muestras para estas explicaciones. Estas estructuras suelen tener dos capas de neuronas especiales: la capa de entrada, que admite los datos que se le dan a la red, y la capa de salida, que devuelve la predicción, o inferencia, de la red. El resultado de la capa de salida tiene tantas neuronas como categorías tiene el problema de clasificación. Ésta se suele conectar a una función llamada *SoftMax*, que ajusta los valores de salida de tal manera que su suma sea 1. De esta forma, lo que se obtiene de su salida son una lista de porcentajes, que indican la "confianza" que tiene la red en que la muestra dada pertenezca a cada categoría. Se puede ver en la figura 2.1 un diagrama simplificado de una red neuronal.

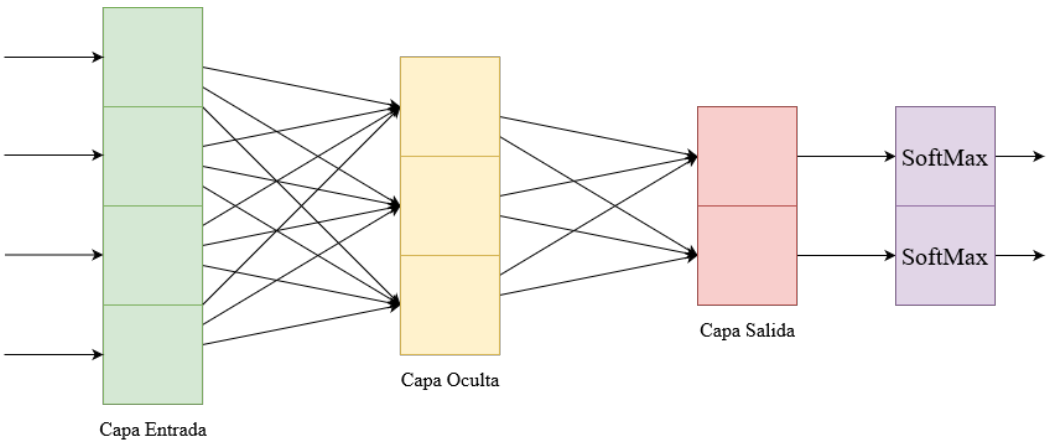

Figura 2.1: Diagrama de una red neuronal simple, que toma un input de tamaño 4, y tiene 2 clases de salida y una capa oculta (intermedia) de 3 neuronas.

Entre las capas de entrada y salida puede haber una o varias capas intermedias (también llamadas capas ocultas). Estas capas definen la estructura interna de la red y, dependiendo de cómo se interconecten éstas o cúantas haya, el modelo resultante será más adecuado para ciertas tareas. Un ejemplo de

esto es *ResNet*, un tipo de red neuronal caracterizado por tener muchas capas ocultas, lo que le hace especialmente bueno en tareas de reconocimiento de imágenes [26].

2.1.1. Neuronas artificiales

Una neurona artificial no es más que una función matemática que toma una lista de entradas (o inputs), le asigna una importancia (o peso) a cada entrada y las opera (la más básica es un sumatorio, aunque hay muchos tipos de neuronas artificiales) [51]. La salida (output) de esta operación se le suele dar a otra función, que la transforma para determinar si se activa o no (de nuevo, hay muchas de estas funciones de activación, pero la más simple determina si la neurona se activa, 1, o no, 0). Un ejemplo de neurona artificial se puede ver en la figura 2.2 y la ecuación que resume cómo opera dicha neurona en la figura 2.3.

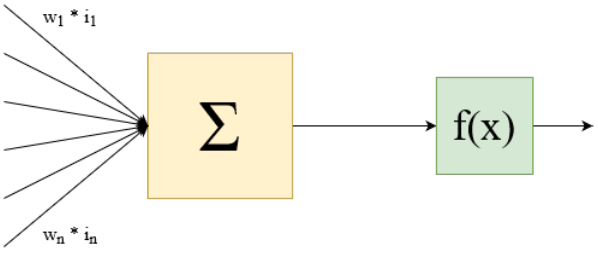

Figura 2.2: Diagrama de una neurona artificial simple, que toma n inputs.

2.1.2. Entrenamiento de una red neuronal

Hasta ahora solo se ha descrito qué es una neurona artificial y una red neuronal, pero no cómo aprende a reconocer patrones y a clasificar correctamente las muestras que se le proporcionan. Para ello, es necesario entender el proceso de entrenamiento de una red neuronal. El entrenamiento se puede descomponer en 2 fases que se intercalan: propagaciones hacia delante, y propagaciones hacia atrás.

• **Propagación hacia delante:** La fase de propagación hacia delante es el proceso que sigue la red neuronal cuando se le da una muestra y, por tanto, se realiza también fuera del entrenamiento. Durante esta fase, cada una de las neuronas de la red calcula su salida, en función de los inputs que recibe y el peso de cada input. La ecuación de la figura 2.3 describe matemáticamente el cálculo que realiza cada neurona de la red. Este proceso se realiza capa a capa (desde la de inputs hasta la de outputs), ya que las salidas de una capa son, generalmente, las entradas de la siguiente capa. [51]

• **Propagación hacia atrás (backpropagation):** La propagación hacia atrás es dónde la red realmente "aprende" de la muestra que acaba de ver; por ello, es más compleja que la propagación hacia delante. El proceso de propagación hacia atrás consiste en calcular cuánto se ha "equivocado" cada neurona de la red, es decir, cuánto tiene que variar sus pesos para que la salida de la red concuerde con lo que se esperaba. Para ello, se utiliza el algoritmo de Descenso Gradiente (*Gradient Descent*) [50]. Para calcular los gradientes, es necesario primero calcular los términos de error de cada neurona, para

$$y = F(\sum_{i=0}^{n} X_i W_i) \tag{2.1}$$

Figura 2.3: Operación interna de una neurona artificial simple, donde X es el vector de entradas, W es el vector de pesos, n es el número de entradas de la neurona y $F(x)$ es la función de activación.

$$\delta_k = (l_k - o_k) * f'(o_k) \tag{2.2}$$

Figura 2.4: Cálculo de los términos de error para la capa de salida (δ_k), donde k es una neurona de la capa de salida, l_k es la salida que se esperaba de dicha neurona, y o_k es la salida actual. $f'(x)$ es la derivada de la función de activación (en este caso SoftMax).

$$\delta_j = f'(o_j) \sum_{k \in sig.capa} (\delta_k W_{kj}) \tag{2.3}$$

Figura 2.5: Cálculo de los términos de error de una capa oculta (δ_j), donde $f'(x)$ es la derivada de la función de activación de la neurona j; o_j es la salida que produjo dicha neurona durante la propagación hacia delante, k es una neurona de la capa siguiente, δ_k es el término de error de dicha neurona y W_{kj} es el peso que la neurona k asigna a la salida de la neurona j.

lo que existen dos fórmulas diferentes, dependiendo de si la neurona pertenece a la capa de salida o no. Para la capa de salida, la ecuación es sencilla, como se puede ver en la ecuación 2.4. [51]

El cálculo de los términos de error en las capas ocultas es un tanto más complejo, ya que cada neurona de la capa tiene tantas entradas como neuronas tiene la capa anterior; además, dichas entradas se utilizarán para calcular los términos de error de la capa anterior, ya que son sus salidas. La ecuación se puede ver en la ecuación 2.5.

Una vez calculados todos estos gradientes, solo es necesario calcular los nuevos pesos de cada neurona. Para ello, se les suman los términos de error calculados anteriormente, multiplicados por el ratio de aprendizaje de la red (*learning rate*), como se puede ver en las ecuaciones 2.6 y 2.7. En resumen, el entrenamiento de una red neuronal sigue el esquema detallado en el algoritmo 1.

$$\Delta W_{ij} = \eta \delta_j x_{ji} \tag{2.4}$$

Figura 2.6: Cálculo de los gradientes de cada peso de la capa actual ΔW (matriz de dimensiones neuronas de la capa x inputs de la capa), donde η es la learning rate (o cuánto aprende la red en cada propagación hacia atrás), δ_j es el término de error de la neurona j, y x_{ji} es el input que recibió la neurona j de la neurona i, de la capa anterior, durante la propagación hacia delante.

$$W'_{ij} = W_{ij} + \Delta W_{ij} \tag{2.5}$$

Figura 2.7: Cálculo de los nuevos pesos de cada neurona de la capa.

Para las explicaciones usadas en esta sección se ha procurado simplificar los conceptos de red neuronal, neurona artificial y sus ecuaciones, ya que el objetivo de esta sección del trabajo no es explicar su funcionamiento interno en detalle, sino dar una idea general de cómo funcionan. Sin embargo, cabe destacar que existen muchos tipos de red neuronal, así como de neuronas artificiales o incluso de algoritmos de Descenso Gradiente (como descenso en batches, descenso estocástico, etc... [50]).

Algoritmo 1: Algoritmo de entrenamiento simplificado de una red neuronal [51].

```
1  Resultado Red neuronal entrenada
2  Funcion Entrenamiento(red: RedNeuronal, ejemplos: Lista(x, y))
3      Para cada muestra: Ejemplo en ejemplos haz
           // Propagación hacia delante de la red
4          inputs = muestra[0];
5          numCapas = red.numCapas - 1;
6          Para x = 0 hasta x = numCapas haz
7              inputs = red.capas[x].propagaDelante(inputs);
8          fin
           // Propagación hacia atrás de la red
9          errores = red.capas[numCapas].calculaErrores(muestra[1]);
10         Para x = numCapas-1 hasta x = 0 haz
11             errores = red.capas[x].calculaErrores(errores);
12         fin
13         red.actualizaPesos();
14     fin
15     devuelve red;
16 fin
```

2.2. Herramientas

Durante el desarrollo de este trabajo, se ha hecho uso de varias herramientas, tanto de manera directa (como los frameworks de redes neuronales utilizados), como de manera indirecta (como las librerías subyacientes que dichos frameworks utilizan). En este apartado se describen algunas de las herramientas utilizadas.

2.2.1. Pytorch

Pytorch [31] es un framework de redes neuronales basado en python y con un backend de c++ creado por Facebook. Permite la creación rápida de modelos de redes neuronales, mediante el uso de modelos secuenciales, así como su entrenamiento y validación. También permite la creación de modelos más complejos mediante la extensión de clases como `nn.module`. Éstos requieren la definición de las capas de la red, sus funciones de activación, así como la definición de su función de propagación hacia adelante (`forward(x)`), en la que se indica cómo están conectadas dichas capas y funciones de activación.

Para alimentar a los modelos de red neuronal, Pytorch cuenta con sus propias estructuras de datos: `Tensors`, `Dataloaders` y `Datasets` [16]. El uso combinado de estas estructuras de datos permite a los programadores poder pasar conjuntos enteros de muestras a la red, pudiendo modificar parámetros como el tamaño de las divisiones dentro de ese conjunto (`batch_size`) o si los elementos contenidos dentro de él se deben reorganizar aleatoriamente (`shuffle`).

Además de esto, Pytorch permite de manera fácil descargar ejecuciones del modelo a aceleradores (como GPUs) mediante la llamada a la función `.to(device)`, nativa a ciertos objetos de Pytorch (como tensores o modelos de red neuronal) [13]. Para que esto funcione bien, sin embargo, es necesario que tanto el modelo, como los datos que se le van a pasar en una propagación hacia adelante, se encuentren en el mismo dispositivo.

Por último, Pytorch permite acelerar aún más la ejecución de las operaciones de una red neuronal mediante la paralelización. En Pytorch se consideran tres tipos de paralelización:

• **Model-Parallel:** Pytorch permite dividir un modelo de red neuronal y enviar cada parte a un dispositivo distinto, de tal manera que cada dispositivo albergue algunas de las capas de dicho modelo. Este tipo de paralelización no requiere llamadas a APIs externas, a excepción de la mencionada anteriormente (`.to(device)`). En el pseudocódigo 2.1 se puede ver un ejemplo de una implementación model-parallel. [35]

```
class RedModelParallel():
  capa1 = nuevaCapa.to(disp0)
  capa2 = nuevaCapa.to(disp1)

  def forward(x):
    salida1 = relu(capa1(x.to(disp1)))
    salida2 = softmax(capa2(salida1.to(disp2))
    return salida2
```

Pseudocódigo 2.1: Modelo que distribuye sus capas entre 2 dispositivos, de acuerdo al paradigma Model-Parallel.

• **Data-Parallel:** Pytorch también permite distribuir entre los dispositivos locales los datos de entrada a la red, en vez del modelo en sí. Para ello es necesario que cada dispositivo albergue una copia completa del modelo. En cada propagación hacia atrás, las gradientes del modelo (cuánto tiene que cambiar cada peso de la red) se sincronizan, para mantener la consistencia entre dispositivos. Este tipo de paralelismo no requiere tanto esfuerzo por parte del programador como el paradigma anterior, ya que Pytorch proporciona un *wrapper* en el que se puede envolver el modelo, transformándolo así en uno paralelo. Un ejemplo de la utilización de dicho *wrapper* se puede ver en el pseudocódigo 2.2. [14]

```
class RedDataParallel():
  capa1 = nuevaCapa
  capa2 = nuevaCapa
  def forward(x):
    ...
def main():
  # Instancia de la red sin paralelizar
  modelo = nueva RedDataParallel()
  # Modelo envuelto en el wrapper de DataParallel de Pytorch
  # No es necesario mandar los datos a los dispositivos, se hace automaticamente
  modeloParalelo = torch.nn.DataParallel(modelo, device_ids=[0,1])
  salida = modeloParalelo(x)
  modeloParalelo.backward()
```

Pseudocódigo 2.2: Transformación de un modelo simple en uno distribuido localmente.

• **Distributed Data-Parallel (DDP):** Este último método de paralelización es similar al anterior, sin embargo, es más flexible y potente que Data-Parallel. DDP permite distribuir los datos entre varios dispositivos, ya se encuentren en un solo nodo de cómputo o distribuidos en una red de equipos. Asimismo, DDP permite mayor control de dicha distribución, permitiendo especificar parámetros como la librería que se utilizará para comunicar los dispositivos, o controlar a qué dispositivo va cada dato, especificándolo manualmente. Para ello, DDP crea un proceso para cada dispositivo que se haya especificado. Sin embargo, esto viene con el costo de mayor esfuerzo por parte del programador, que necesitará añadir más lineas de código para convertir el modelo a DDP. Este protocolo de paralelización es el que recomiendan los desarrolladores de Pytorch [36]. Se puede ver un ejemplo de uso en el pseudocódigo 2.3. [15]

```
def entrena(rango, tamMundo):

  # Funcion que crea un comunicador para los procesos
  inicializa_comunicador()


  modelo = nueva RedDDP()
  # Transformacion del modelo en uno DDP
  modeloParalelo = torch.nn.DDP(modelo)

  # En este caso si es necesario especificar el dispositivo
  # en el que se encuentra nuestra instancia de modelo
  salida = modeloParalelo(x.to(rango))
  modeloParalelo.backward()

  # Funcion que destruye el comunicador de procesos
  # una vez se ha terminado la ejecucion paralela
  finaliza_comunicador()


def main():
  numDisp = 2
  paraleliza_funcion(entrena, args=(numDisp,), numProcesos=numDisp)
```

Pseudocódigo 2.3: Transformación de un modelo simple en uno DDP (asumimos que la clase RedDDP es similar a la red de los pseudocódigos anteriores).

2.2.2. Tensorflow

Tensorflow es otro framework de redes neuronales parecido a Pytorch, creado por Google. Ambos están basados en python, utilizando un backend de c++ para ciertas operaciones. Una de las características que distingue a Tensorflow de Pytorch es la posibilidad de crear modelos de forma más fácil, mediante la utilización de la API *Keras*. *Keras*[21] es una API de alto nivel que permite la creación de modelos secuenciales. Además, permite entrenar al modelo sin necesidad de escribir un bucle de entrenamiento manualmente (este no era el caso en Pytorch), aunque también da la opción a escribir bucles de entrenamiento propio. [1]

Por defecto, Tensorflow manda todos los entrenamientos a GPU (si está instalado el soporte para GPU en la máquina) y también permite entrenar modelos en aceleradores TPU (*Tensor Processing Unit*) [22], ASICs de google especialmente diseñados para acelerar tareas de Machine Learning.

Además de todo lo anterior, Tensorflow proporciona varios métodos de paralelización del entrenamiento de redes neuronales, que denomina estrategias [20]:

•**Mirrored Strategy:** Esta estrategia permite distribuir el entrenamiento de una red neuronal entre varias GPUs disponibles en una misma máquina. Para ello, entrega a cada GPU una copia del modelo, y parte de los datos, y sincroniza los modelos durante el entrenamiento.

•**TPU Strategy:** Ya mencionada anteriormente, permite entrenar un modelo en aceleradores TPU disponibles en la nube de google. Sigue el mismo concepto que Mirrored Strategy. Para ello, sin embargo, es necesario que los datos a los que va acceder el modelo estén disponibles en un "cubo" de google cloud (*google cloud storage bucket*).

•**Estrategias distribuidas:** Tensorflow proporciona varias estrategias para distribuir el entrenamiento de un modelo a lo largo de varios dispositivos, o nodos de una red: *Multiworker Mirrored*, *Central Storage* y *Parameter Server*. Estas tres estrategias diferentes describen diferentes topologías. La primera

es análoga al *Mirrored Strategy*, utilizando las GPUs de varios nodos. En la segunda estrategia, *Central Storage*, el modelo se aloja en la CPU de una máquina, pero las operaciones a realizar sobre ese modelo se pasan a las GPUs locales. La tercera estrategia, *Parameter Server*, usa pares de nodos de una red, donde uno almacena el modelo (sus pesos, biases, etc) y el otro realiza las operaciones sobre dicho modelo; además, se puede utilizar uno de los nodos de la red como coordinador de todos los pares.

2.2.3. Librerías de Nvidia

Para poder ejecutar los modelos de red neuronal en GPU (ya sea en Pytorch o Tensorflow) es necesario utilizar *CUDA Toolkit* [18], el SDK de Nvidia que permite escribir aplicaciones que corren en GPU. Además de esto, los frameworks de RN necesitan ciertas librerías de Nvidia adicionales.

• **CuDNN:** Se trata de una librería de Nvidia que proporciona implementaciones de operaciones primitivas de redes neuronales (como operaciones de convolución, funciones de activación...) que corren en GPU. Esta librería se usa en conjunto con otra librería de Nvidia, *CuBLAS*. *CuBLAS* tiene un comportamiento similar a *CuDNN*, ofreciendo implementaciones de operaciones de álgebra lineal que corren en GPU. [8]

• **NCCL:** es una librería que ofrece operaciones para la comunicación entre varias GPUs de Nvidia. Al ser una librería de comunicaciones, implementa las típicas operaciones de comunicación (`all-reduce`, `broadcast`, `gather`, etc...). Además de esto, *NCCL* permite la comunicación entre varias GPUs que se encuentran en diferentes dispositivos de una misma red, ya que soporta varias tecnologías de interconexión, como Infiniband o IP. [43]

2.2.4. Sauna

Sauna es una herramienta de perfilado desarrollada por Pérez et al. [47] [46] que permite tomar medidas de la potencia instantánea de varios componentes, como los cores del CPU, RAM del equipo, o la GPU, durante la ejecución de un proceso. Esta herramienta permite definir parámetros como el fichero de salida, la frecuencia con la que tomar las mediciones de potencia (cada cuántos milisegundos hacer una medición) y la región del programa a perfilar (mediante el uso de `stdout`).

2.3. Trabajos Relacionados

Existen varios trabajos anteriores que se centran en examinar el rendimiento de redes neuronales en diferentes ambientes. Entre ellos se encuentran benchmarks, o pruebas de rendimiento, que examinan cómo rinden diferentes componentes hardware en tareas de machine learning.

DNNMark [11] es un ejemplo de benchmark que se centra en evaluar el rendimiento de las primitivas relacionadas con redes neuronales (las operaciones de bajo nivel que se realizan durante la ejecución de una red neuronal, como puede ser multiplicación de matrices). Otro ejemplo de este tipo de benchmark es DeepBench [3], de Baidu. En él también se hacen pruebas de las diferentes operaciones simples que componen los procesos de *Deep Learning*, ya sean de la fase de entrenamiento o en de la de inferencia[1], en diferentes equipos hardware. Para ello, este tipo de benchmarks hace uso de las librerías de ML (como CuDNN) directamente, sin usar ningún framework.

Por otro lado, existen benchmarks como el que proponen Adolf et al. [2]; una colección de ocho modelos de red neuronal desarrollados en Tensorflow, cuyo objetivo es estudiar los tipos de operaciones que más tiempo consumen a la hora de entrenar y ejecutar una red neuronal, además de estudiar los efectos de la paralelización en la escalabilidad de un modelo. Asimismo, existen trabajos como MLPerf Training [37] y MLPerf Inference [48], que se centran más en estudiar el comportamiento de redes neuronales en

[1] La fase en el ciclo de vida de una red neruonal en el que ya no se realizan propagaciones hacia atrás, pues se considera que la red ya ha sido entrenada correctamente.

su conjunto, usando modelos de referencia en diferentes equipos. El primero de éstos, MLPerf Training [37] se centra en redes neuronales durante su entrenamiento, mientras que el segundo, MLPerf Inference [48] está enfocado a la comparación de modelos neuronales en su entorno de despliegue, es decir, una vez ya han sido entrenados. Estos dos benchmarks son similares al que proponen Coleman et al. [9] en DAWNBench. Este benchmark mide el tiempo que tarda una red neuronal en entrenarse hasta alcanzar una precisión establecida como meta, al igual que MLPerf Training; sin embargo, a pesar de proporcionar implementaciones en Tensorflow y Pytorch, no cuenta con el amplio soporte hardware de éste. Además, DAWNBench solo prueba modelos en tareas de clasificación de imágenes, con lo que la diversidad de modelos soportados es también escasa. Tango [29] es otro benchmark de redes neuronales que destaca por su énfasis en soportar dispositivos no solo compatibles con CUDA, sino también con OpenCL [39], permitiendo así comparar una gama amplia de dispositivos.

También existen trabajos que crean perfiles de diferentes sistemas, con la finalidad de entender mejor cómo interactúan los algoritmos de ML con el hardware subyacente. En [38] Mojumder et al. realizan un perfilado de un sistema multi-gpu con diferentes modelos de red neuronal para comparar dos sistemas de comunicación entre dispositivos gpu, *NCCL*(ya descrito anteriormente) y *Peer2Peer* (P2P). Por otro lado, en [34], Li et al. proponen un nuevo diseño de software de perfilado que permite monitorizar cada capa de abstracción en un sistema de ML (hardware, APIs de bajo nivel, modelo de red neuronal...).

Por último, hay trabajos que centran sus esfuerzos en paralelizar modelos de redes neuronales y examinar la ganancia de rendimiento que se obtiene al utilizar varios dispositivos para entrenar una red neuronal y cómo hacer que ésta sea lo más alta posible. En [36], Li et al. investigan los beneficios de usar el entrenamiento distribuido de Pytorch para entrenar un modelo en varios dispositivos y cómo los overheads de diversas librerías de comunicación pueden afectar a dicho entrenamiento negativamente. Jäger et al. [28], sin embargo, optan por realizar una comparativa entre diferentes frameworks de ML, como Tensorflow, para examinar su rendimiento a la hora de paralelizar modelos.

Capítulo 3

Entrenamiento de RRNN en ambientes distribuidos

Para la elaboración de los experimentos de este proyecto fue necesario realizar ciertas actividades previamente. En este capítulo se describe la investigación previa al comienzo del desarrollo del proyecto, tras lo que se pasa a explicar la elección del dataset y modelo de red neuronal escogidos, así como la investigación realizada sobre un posible modelo de paralelización de red neuronal en Pytorch, usando tanto las GPUs disponibles como el CPU. Por último se detallan las pruebas realizadas a la hora de determinar los valores de ciertos parámetros del benchmark.

3.1. Selección del Framework de Desarrollo

Previo al desarrollo del código del benchmark, se realizaron unas investigaciones, con los siguientes objetivos: familiarizarse con las tecnologías a utilizar durante el proyecto y estudiar el grado de soporte a entrenamientos distribuidos que proporciona cada uno de los frameworks explicados (en la sección 2.2), para poder así determinar cuál utilizar en el programa de benchmark.

Se comenzó haciendo un estudio de Tensorflow [1] y sus opciones de paralelización (ya explicadas en 2.2.2) [20]. Una vez hecho esto, se siguió el mismo procedimiento con Pytorch [31], estudiando esta vez no sólo las estrategias de paralelización disponibles en dicho framework, sino también cómo ejecutar los modelos creados en GPU [13], ya que (al contrario que Tensorflow) Pytorch no utiliza por defecto la GPU.

Al mismo tiempo, se decidió investigar si, de forma oficial, los frameworks seleccionados daban soporte a GPUs que no utilizaran la tecnología CUDA. Sin embargo, sólo se encontraron artículos que proponían extensiones a estas plataformas por parte de terceros, como la propuesta de Goli et al. [19] en la que describen una adaptación de Tensorflow, diseñada para utilizar `SYCL`, una API de programación de dispositivos OpenCL a alto nivel [23]. Es por este motivo por el que, durante el desarrollo del proyecto, solo se han utilizado GPUs de Nvidia.

Framework	Precisión final	Tiempo total
Tensorflow	95 %	9.4”
Tensorflow GPU	96 %	6.13”
Pytorch	90 %	14”
Pytorch GPU	90 %	19.5”

Tabla 3.1: Algunos de los resultados obtenidos durante el estudio de los diferentes frameworks. Como se puede ver, Tensorflow es más rápido que Pytorch. Además, el rendimiento de Pytorch utilizando GPU es menor que utilizando la CPU, esto se debe a que la red era demasiado simple, y el tamaño de los datos también, con lo que el framework no era capaz de aprovechar la GPU al máximo

Una vez hecho esto, se crearon programas de pruebas en ambos frameworks, para familiarizarse con ellos, y con el lenguaje de programación Python (sobre el que estaban construidos), además de servir para comparar ambas implementaciones y escoger la plataforma en la que basar el resto del desarrollo. Los programas de prueba creados eran implementaciones de una red neuronal simple que aprendía a reconocer dígitos manuscritos usando el dataset de MNIST[32]. La estructura de la red era la siguiente: 784 neuronas de entrada (inputs de 28x28), 192 neuronas en la única capa oculta de la red y 10 neuronas representando las clases del problema (de 0 a 9). La versión final de estos programas de prueba hacía uso de la GPU, para lo que fue necesario esfuerzo adicional en la implementación de Pytorch. En los pseudocódigos 3.1 y 3.2 se pueden ver las implementaciones de las pruebas, mientras que en la tabla 3.1 se pueden ver los resultados que se obtuvieron.

```
# Implementacion de la red
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.l1 = nn.Linear(784, 192)  # CAPA DE ENTRADA
        self.out = nn.Linear(192, 10)  # CAPA DE SALIDA

    def forward(self, x):
        ...
#Definicion del entrenamiento
def train():
    ...

#Definicion de la validacion
def validate():
    ...

#Definicion del test
def test():
    ...

#Definicion del modelo y de su Gradient Descent
net = Net()
loss_fn = nn.MSELoss(reduction='mean')
optimizer = optim.SGD(net.parameters(), lr=0.5)

#Entrenamiento de la red
for i in range(8):
    print("\n===============")
    print("=NUEVO EPOCH " + str(i) + "=")
    print("===============")
    train()
    validate()

#Test de la red
test()
```

Pseudocódigo 3.1: Extracto de código de la implementación de Pytorch

A pesar de que Tensorflow dio mejores resultados en los programas de pruebas (como se puede ver en la tabla 3.1), se decidió usar finalmente Pytorch. Esto se debe a dos factores. El primero de ellos fue la incompatibilidad de Tensorflow con el programa de perfilado que se planeaba usar inicialmente para los experimentos (`nvprof`). Éste era incapaz de detectar cuándo Tensorflow hacía uso de la GPU, y por lo tanto no reportaba sus métricas. Además de esto, se consideró que el framework era de muy alto nivel, dejando pocas posibilidades al desarrollador a la hora de modificar los bucles de entrenamiento, validación y test. Como se puede ver en el pseudocódigo 3.2, para entrenar se realiza una invocación a la función `.fit()`, no pudiendo definir cuándo realizar la fase de validación[1], por ejemplo; al contrario que la implementación de Pytorch (pseudocódigo 3.1), en la que es necesario definir estos bucles.

[1]Se realiza una vez por época, tras iterar sobre el conjunto de entrenamiento.

```
#Definicion del modelo
model = tf.keras.models.Sequential([
    tf.keras.layers.Dense(192, activation='sigmoid'),
    tf.keras.layers.Dense(10, activation='softmax')
])

#Definicion del algoritmo de Gradient Descent del modelo
model.compile(optimizer=...,
              loss=...,
              metrics=['accuracy'])

#Entrenamiento del modelo
model.fit(..., batch_size=50, epochs=8,...)

#Test del modelo
model.evaluate(test_examples, test_labels)
```

Pseudocódigo 3.2: Extracto de código de la implementación de Tensorflow

3.2. Elección del dataset y modelo de red neuronal

Una vez concluida la fase de estudio de la plataforma escogida, se pasó a iniciar el desarrollo del proyecto. Primero se estudiaron varios benchmarks de redes neuronales, para estudiar los aspectos a tener en cuenta a la hora de desarrollar el nuestro, como DNNMark [11], MLPerf Inference [48] o MLPerf Training [37], siendo éste último el más usado como referencia. Una vez hecho esto, se comenzó el desarrollo, empezando por seleccionar el dataset y el modelo de red neuronal a utilizar.

Para la selección del dataset se tuvo en cuenta que el proyecto no se centra en un estudio de redes neuronales, sino en la evaluación de su rendimiento frente a varias estrategias de paralelización. Debido a esto, sería necesario realizar muchas pruebas, con lo que éstas debían tener una duración razonable. Por ello, el dataset escogido inicialmente fue "Concrete Crack Images For Classification" [45], que contiene 40,000 muestras de hormigón distribuidas a lo largo de dos clases, hormigón con grietas y hormigón sin grietas. Este dataset fue escogido en vez de los típicos para tareas de visión (como MNIST [32] e Imagenet [10]) por varias razones: se trataba de un dataset mucho más liviano que Imagenet (1,331,167 muestras en total) y MNIST (70,000 muestras en total), por lo que la carga que generaría en el hardware a utilizar en las pruebas se consideró adecuado; además de esto, las muestras tenían un tamaño fijo (reduciendo la complejidad a la hora de tratarlas) de 227x227 píxeles, al contrario que Imagenet. Dicho tamaño era además mayor que el de las muestras de MNIST (28x28 píxeles), lo que daba una mayor carga al hardware. Por último, las muestras estaban distribuidas uniformemente, al contrario que Imagenet, en el que cada clase tiene un tamaño distinto, haciendo que el entrenamiento de la red fuera más efectivo y consistente. Sin embargo, debido al número reducido de clases del dataset, la red conseguía la precisión máxima de 95 % tras ver 3,000 de las 40,000 muestras, con lo que se decidió usar este dataset en el proyecto hasta encontrar uno con características similares pero que tuviese más clases. El dataset fue reemplazado más adelante por "A Large Scale Fish Dataset" [55], pero los motivos de su elección serán discutidos más adelante.

Como modelo de red neuronal, nos inspiramos en el benchmark MLPerf Training [37], debido a que es un benchmark muy usado en la actualidad y que ha sido creado en colaboración con muchas organizaciones dedicadas a la investigación de *Machine Learning*, entre ellas la que desarrolla y mantiene Pytorch. En él, los autores usan para las tareas de visión un modelo de red neuronal llamado ResNet [26]. ResNet es un modelo de red neuronal profunda (DNN[2]) que incorpora "atajos" entre determinadas capas, para propagar la entrada de una capa a la siguiente sin procesarla. Esta peculiaridad le ayuda a resolver un problema nativo a las DNNs, el problema de los *vanishing gradients* en el que los gradientes

[2]Tipo de red neuronal con muchas capas ocultas.

de las capas cercanas a la de entrada acaban siendo tan pequeños que el entrenamiento de dichas capas se vuelve casi imposible [26]. Esta propiedad de ResNet le ha ayudado a ganar muchas competiciones de redes neuronales y a convertirse en un referente en modelos orientados a tareas de visión (como clasificación de imágenes) [26].

Para nuestra implementación del modelo ResNet, se escogió la versión de 34 capas, un modelo un tanto más pequeño que el que usan Mattson et al. en MLPerf Training [37]. Esto se debe a que el modelo ResNet-34 es más liviano que otros de los modelos ResNet que proponen los creadores, como los de 50, 100 o incluso más de 1000 capas[26]. Además, se consideró que la versión de 34 capas era más que suficiente dada la complejidad del dataset escogido y la capacidad del hardware disponible para el proyecto.

Una vez elegidos tanto el dataset como el modelo, se pasó a la implementación de una script que preprocesaba los datos del dataset, para reducir el tiempo de lectura de los datos durante la ejecución del benchmark. Esta script admite como parámetros los nombres de los directorios donde se almacenan las muestras de cada clase (cada directorio representa una clase) y trasforma las imágenes contenidas en ellos en matrices bidimensionales de FP16, para lo que se coge solo el canal de brillo de las imágenes; en otras palabras, se convierten a fotos en blanco y negro. La decisión de convertir las imágenes en monocromas, vino del contenido de los datasets elegidos; en ambos casos las imágenes eran reconocibles en blanco y negro. En cuanto a la decisión de usar números de punto flotante de 16 bits, se debe a que las imágenes en blanco y negro venían codificadas con valores de entre 0 y 255 para cada píxel [3] y se consideró que la precisión adicional que dan los números de más bits era innecesaria. Además de esto, el uso de precisiones reducidas es una técnica muy utilizada en el entrenamiento de redes neuronales, ya que permite optimizar el rendimiento al poder hacerse varias operaciones por ciclo de reloj. Asimismo, cada vez más hardware cuenta con módulos especializados en operaciones con precisión reducida, como los *Tensor cores* de las últimas generaciones de GPUs de Nvidia [25].

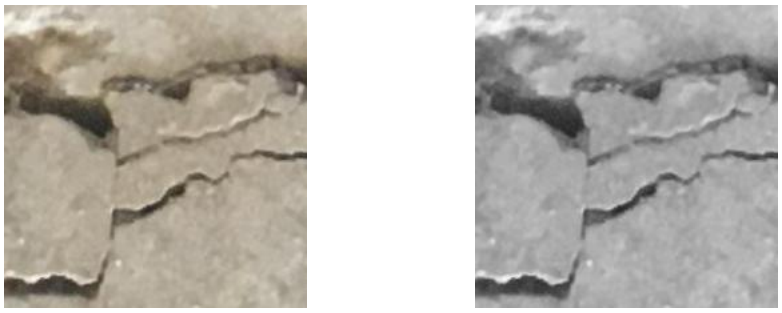

Figura 3.1: Imagenes del dataset inicial [45]. Izquierda: Imagen del dataset sin preprocesar, con los tres canales RGB. Derecha: Imagen del dataset procesada y convertida a blanco y negro.

Tras convertir las imágenes en matrices, se distribuyen en una lista de manera aleatoria, anotando en otra lista la clase a la que pertenecen; esto se hace para asegurar que, a la hora de particionar el dataset, no pueda quedar una clase sin representación en cualquiera de los tres conjuntos (entrenamiento, validación y evaluación o test). Por último, se escribe la lista de matrices en dos archivos binarios: (`data.pkl`) y (`labels.pkl`), donde el primero contiene las muestras codificadas y el segundo las etiquetas de cada muestra.

Para la implementación del modelo escogido usando Pytorch, se siguió la propuesta de Zuppichini [58], en la que describe cómo implementar los componentes que conforman cualquier modelo ResNet y cómo combinarlos para crear un modelo ResNet del tamaño deseado. Sobre esta implementación de

[3]Dichos valores deben ser normalizados (el rango de valores debía estar entre 0 y 1) para que la red neuronal los pueda procesar correctamente.

referencia se realizaron cambios menores, eliminando componentes no necesarios para implementar el modelo de 34 capas.

```
def entrena(red, muestras, optimizador, fn_perdida, dispositivo):

    for ejemplo, etiqueta in muestras:
        salida = red.forward(ejemplo.to(dispositivo))
        perdida = fn_perdida(salida, etiqueta.to(dispositivo))
        perdida.backward()
        optimizador.step()

def valida_o_test(red, muestras, perdida_fn, dispositivo):

    # Esto impide que la red actualice sus pesos durante la
    # validacion
    red.eval()

    precision = 0
    perdida_med = 0

    for ejemplo, etiqueta in muestras:
        salida = red.forward(ejemplo.to(dispositivo))
        perdida = fn_perdida(salida, etiqueta.to(dispositivo))

        if salida == etiqueta:
            precision++

        perdida_med += perdida

    precision = precision/muestras.size()
    perdida_med = perdida_med/muestras.size()

    return [precision, perdida_med]
```

Pseudocódigo 3.3: Bucles de entrenamiento y validación

Cuando se hubo terminado de implementar el modelo de ResNet, se crearon bucles de entrenamiento y validación/test siguiendo el ejemplo del pseudocódigo 3.3. Para comprobar que todo funcionara bien, se creó un pequeño programa que entrenaba una instancia del modelo en GPU usando los datos preprocesados del dataset. Este programa de prueba supuso el primer prototipo del benchmark, usando tan solo una de las dos GPUs para entrenar la red.

3.3. Desarrollo de un modelo de paralelización híbrido

Con el modelo de red neuronal implementado y el dataset elegido y procesado, se propuso realizar una investigación sobre la viabilidad de un modelo de paralelización híbrido basado en el paradigma *data-parallel* de pytorch. Este modo de paralelizar utilizaría tanto la CPU como las GPUs disponibles para entrenar la red neuronal, entregando a cada dispositivo una copia del modelo y los datos de entrada. La primera implementación de este concepto usaba la API de DDP de Pytorch [15] con una pequeña modificación; a la lista de dispositivos se le añadía el CPU, para que el benchmark también le mandara una instancia del modelo y de los datos. Sin embargo, el rendimiento que se obtuvo de esta primera versión no fue el esperado, tardando mucho más en ejecutarse que la implementación que usaba una sola GPU. Esto se debe a que la distribución de trabajo para los dispositivos no era la adecuada, haciendo que la CPU retrasara la ejecución del resto de componentes.

Se sospechaba que el culpable de este déficit de rendimiento era el tamaño de los datos mandados al CPU (era igual que los datos mandados a una GPU). Para solucionar esto se modificó el programa, incorporando un equilibrado de carga en función del dispositivo. Desafortunadamente, esta solución

inicial no funcionó, haciendo que el programa entero se colgase, esperando a que el CPU procesara datos que no se le iban a mandar.

Para solucionar el problema anterior se intentó cambiar la librería que se usaba para crear los comunicadores de procesos por MPI, sin éxito alguno. Tras revisar la documentación de DDP de Pytorch [15], se encontró una llamada a la API que permitía el uso de tamaños disjuntos de entrenamiento en diferentes dispositivos (`.join()`). Se modificó el programa para que usara esta llamada, pero se observó que el comportamiento no era el esperado, quedándose la GPU con muy poca carga de trabajo, esperando a que el CPU terminara su ejecución para comenzar la suya (esencialmente haciendo el entrenamiento secuencial).

Se consiguieron solucionar estos problemas de la paralelización híbrida tras descubrir que era posible usar tamaños disjuntos de entrenamiento en cada dispositivo sin la llamada a `.join()` si se ajustaba el tamaño de batch de cada dispositivo adecuadamente, haciendo que el bucle de entrenamiento de cada dispositivo tuviera el mismo número de iteraciones. Debido a esto, se introdujo un "factor de ajuste" en el algoritmo de equilibrado de carga, que reducía proporcionalmente el tamaño de los datos y de los batches mandados al CPU respecto de los mandados a una GPU. El esquema del equilibrado de carga utilizado se puede ver en el psedocódigo 3.4.

```
def particiona_datos(datos, tamanhos, dispositivo, factor_ajuste, batch_size):
    if dispositivo == 'cpu':
        datos_entren = datos[:(tamanhos[0]/factor_ajuste)]
        batch_size = batch_size/factor_ajuste
    else:
        datos_entren = datos[:tamanhos[0]]
    ...
```

Pseudocódigo 3.4: Esquema del equilibrado de carga utilizado.

Conseguido esto, se pasaron a hacer pruebas para determinar el mejor factor de ajuste, de tal manera que el tiempo de entrenamiento fuera inferior al de una GPU sola. Para ello, se calculó primero el speedup de pasar de la ejecución en CPU a la ejecución en GPU, siguiendo la ecuación 3.2, este sería el factor de ajuste que se creyó óptimo. Sin embargo, por muy pequeño que fuera el tamaño de los datos enviados al CPU, el tiempo de entrenamiento siempre era superior al de una GPU. Creemos que esto puede ser debido a que se produce una sobrecarga en el CPU, ya que no solo tiene que entrenar su instancia del modelo, sino también gestionar las comunicaciones con las GPUs, lo que provoca un aumento significativo en el tiempo de entrenamiento .

$$F_{cpu} = \frac{T_{cpu}}{T_{gpu}} \tag{3.1}$$

Figura 3.2: Ecuación utilizada para el cálculo del factor de ajuste inicial, donde F_{cpu} es el factor de ajuste, T_{cpu} es el tiempo de entrenamiento en CPU y T_{gpu} es el tiempo de entrenamiento en GPU.

Al final, se desestimó el modelo de paralelización híbrido en favor de un entrenamiento paralelo en varias GPUs. En la figura 3.3 se ilustra la diferencia de rendimiento entre el modelo que usa una GPU y el modelo que usa 2 GPUs y el CPU para entrenar una instancia de ResNet-34 (las especificaciones concretas del hardaware utilizado se encuentran en la sección 4.2.1.

Como se puede observar en la figura 3.3, en el modelo híbrido no solo el tiempo de entrenamiento es muy alto, sino que además la precisión de la red a lo largo del entrenamiento es más baja e inestable que la del modelo entrenado en una sola GPU. Esto puede deberse al ajuste de los tamaños de batch para los diferentes dispositivos; los resultados de la CPU acaban teniendo más peso (al utilizar batches más pequeños) y "confundiendo" a la red.

No todo fue en vano, durante el desarrollo de este programa se realizaron optimizaciones que resultaron en un uso significativamente menor de la VRAM de las GPUs. Para ello, se cambió la manera en la que se enviaban los datos a las GPUs: al principio esto se hacía a la vez que se creaban las estructuras de datos, es decir, al principio del programa. Sin embargo, se observó que al hacerlo así, pytorch reservaba una parte de la VRAM para la estructura de datos, y luego expandía esa región cuando le llegaba un batch al dispositivo, limitando mucho el tamaño máximo de éste. Por lo tanto, si se pasaban los datos al dispositivo solo durante cada iteración del entrenamiento, se podían mandar batches con más del doble de muestras, a costa de una pequeña penalización de tiempo. Además de esto, el programa sentó las bases de los benchmark y se utilizaría como plantilla para desarrollar el resto del código más adelante en el proyecto.

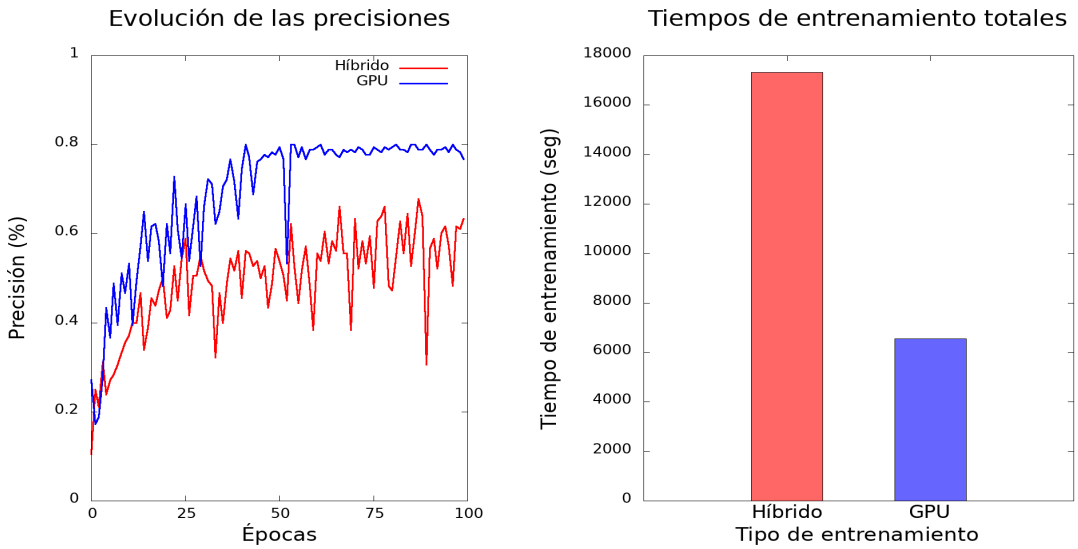

Figura 3.3: Comparativa del rendimiento del entrenamiento usando paralelización híbrida (CPU-GPU) vs entrenamiento en una sola GPU. El factor de ajuste empleado fue 12 (se corresponde con el speedup de GPU vs CPU, como se verá en el capítulo 4), con lo que el CPU vio 12 veces menos muestras que las GPUs. Izquierda: comparativa de las precisiones a lo largo de las épocas. Derecha: comparativa de los tiempos de entrenamiento totales para 100 épocas.

3.4. Experimentos con hiperparámetros

Una vez concluido el desarrollo del modelo híbrido, se pasó a buscar un dataset para usar de manera definitiva. Entre los requisitos que debía tener este dataset estaban: que tuviera un número de muestras suficiente, que pudiera ser utilizable en blanco y negro, y que su número de clases fuera elevado, aportando así más complejidad al entrenamiento que el dataset inicial. Con estos requisitos en mente, se eligió el dataset "Large-Scale Dataset for Fish Segmentation and Classification" [55]. Este dataset contiene imágenes y máscaras[4] de pescados y mariscos encontrados en una pescadería, distribuidos a lo largo de nueve clases.

Como se quería entrenar el modelo de ResNet con imágenes en blanco y negro, se decidió usar tan solo las máscaras del dataset. El número de estas máscaras era de 9,000 (1,000 por clase) y, aunque era significativamente inferior al del dataset inicial (40,000 muestras), se consideró que había suficientes como para garantizar un entrenamiento correcto usando múltiples épocas. Otra de las consideraciones que hubo que tomar con este nuevo dataset fue el tamaño de dichas imágenes(590x445 píxeles), ya que era mayor al del dataset anterior. Se decidió redimensionar las imágenes para hacerlas más pequeñas y fáciles de procesar para la red neuronal. Esto no tendría impacto en la calidad de las muestras, ya que al ser máscaras éstas podían ser encogidas bastante antes de perder detalles importantes que afectaran a la precisión de la red.

Figura 3.4: Imagenes del dataset utilizado finalmente [55]. Izquierda: Imagen sin preprocesar, con la resolución original y en color. Derecha: Imagen procesada, redimensionada a 197x148 píxeles y convertida a blanco y negro.

También se escribió la versión definitiva del programa de benchmark, en el que se reutilizó el mecanismo de equilibrado de carga del programa híbrido para que se repartiese a todas las GPUs disponibles en el equipo la carga de manera equitativa, independientemente de su cantidad. Se modificó la invocación al bucle de entrenamiento para que soportara múltiples épocas[5], tomando al final de cada una una medición de la precisión de la red usando el conjunto de validación. De esta manera, se podía evaluar *a posteriori* la evolución de la red a lo largo del tiempo. El pseudocódigo 3.5 ilustra un esquema de la implementación final del programa de benchmark.

El último paso en el desarrollo del benchmark fue determinar todos los hiperparámetros[6] a utilizar en las pruebas; estos son: el tamaño de los conjuntos de entrenamiento, validación y evaluación, el tamaño de los batches del conjunto de entrenamiento y el *learning rate* de la red. Para determinar estos hiperparámetros se recurrió al método de prueba y error, ya que éstos dependen de muchos factores, como el dataset, el modelo de red neuronal y el hardware utilizados, y por lo tanto no existe un método matemático capaz de calcularlos [51]. Cabe destacar que no se experimentó con todos los hiperparámetros, algunos se dejaron pre-establecidos. Un ejemplo es la *learning rate* a utilizar (0.05), que se trata del punto medio entre las dos *learning rates* que utilizan Mattson et al. [37] en sus benchmarks de visión (utilizan una lr de 0.1 para el comienzo del entrenamiento y otra de 0.01 hacia el final).

[4] Imágen en blanco y negro que resalta la silueta del objeto que contiene.
[5] Iteración completa sobre el conjunto de muestras que conforman el set de entrenamiento y el set de validación.
[6] El conjunto de parámetros de alto nivel que describen al proceso de entrenamiento del modelo

```
def entrena(datos, rango, tamMundo):

    # Conversion de los datos en tensores de Pytorch y particionado
    # en los diferentes conjuntos
    entren, val, test = a_conjuntos(datos)

    #Creacion del modelo de red paralelo
    red = nueva ResNet34().to(rango)
    red_paralela = torch.nn.DDP(red)

    precisiones = [] # Evolucion de la red
    ts_entren = []
    ts_val = []

    # Entrenamiento y validacion de la red durante 100 epocas
    # Las llamadas a entrena_red y valida_red usan el rango para
    # determinar a que dispositivo mandar los datos
    for i in range(100):
        t = time.time()
        entrena_red(red, entren, rango)
        ts_entren.append(time.time()-t)

        t=time.time()
        precisiones.append(valida_red(red, entren, rango))
        ts_val.append(time.time()-t)

    t = time.time()
    precision_fnal = valida_red(red, test, rango)
    t_test = time.time()-t

    return [precision_fnal, precisiones, ts_entren, ts_val]

def main():

    # Deteccion de todas las GPU disponibles
    numDisp = len(pytorch.get_cuda_devices())

    # Lectura de los datos desde sus archivos
    datos = lee_archivo()

    # Entrenamiento de la red usando los datos cargados en los
    # dispositivos detectados
    resultados = paraleliza_funcion(entrena, args=(datos,
                                numDisp, ), numProcesos=numDisp)

    # Guardado de los resultados en un archivo para posterior
    # inspeccion y procesado
    with open('Resultados.csv', 'w') as archivo:
        archivo.write_csv(resultados)

main()
```

Pseudocódigo 3.5: Esquema de la implementación final del benchmark

Primero se decidió redimensionar las imágenes del dataset, para ello se experimentó con 2 resoluciones: 295x222 (aproximadamente 1/4 de la resolución original) y 148x112 (aproximadamente 1/16). Para cada una de las 2 resoluciones se probaron varios tamaños de batch, empezando por el usado durante el desarrollo como *placeholder*, 72. Para la resolución más grande se observó que se llenaba la VRAM para un tamaño de 128, con lo que no se pudo pasar de ahí. Por otro lado, la resolución más pequeña permitió usar tamaños de batch de hasta 256 muestras. Para cada resolución y tamaño de batch se midieron sus tiempos generales, así como los de entrenamiento y validación de cada epoch y la evolución de sus precisiones, para poder así determinar la combinación con mejores resultados. Se pueden ver en las figuras 3.5 y 3.6 los resultados.

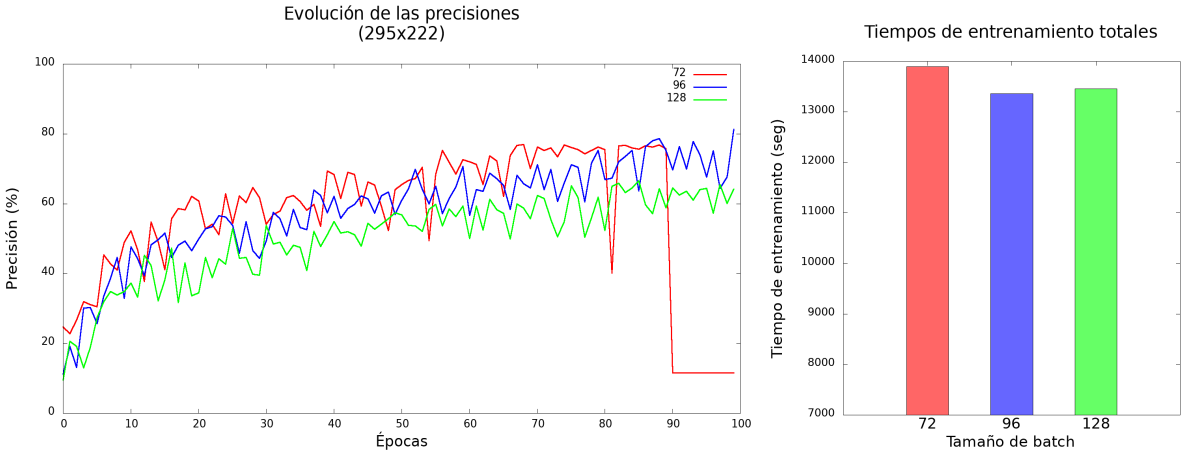

Figura 3.5: Comparativa de precisiones y tiempos de entrenamiento para muestras de 295x222 píxeles utilizando diferentes tamaños de batch. Izquierda: Evolución de las precisiones a lo largo de los epoch para tamaños de batch de 72, 96 y 128 muestras. Derecha: Tiempos de entrenamiento totales para tamaños de batch de 72, 96 y 128 muestras.

Como se puede observar, las pruebas para la resolución 295x222 tardaron más tiempo que las de 148x112 y sus precisiones fueron más inestables. Este comportamiento se debe a una learning rate muy agresiva para esta resolución, que impide a la red converger hacia un resultado deseable, llegando a producirse un colapso de la precisión para el tamaño de batch de 72 muestras.

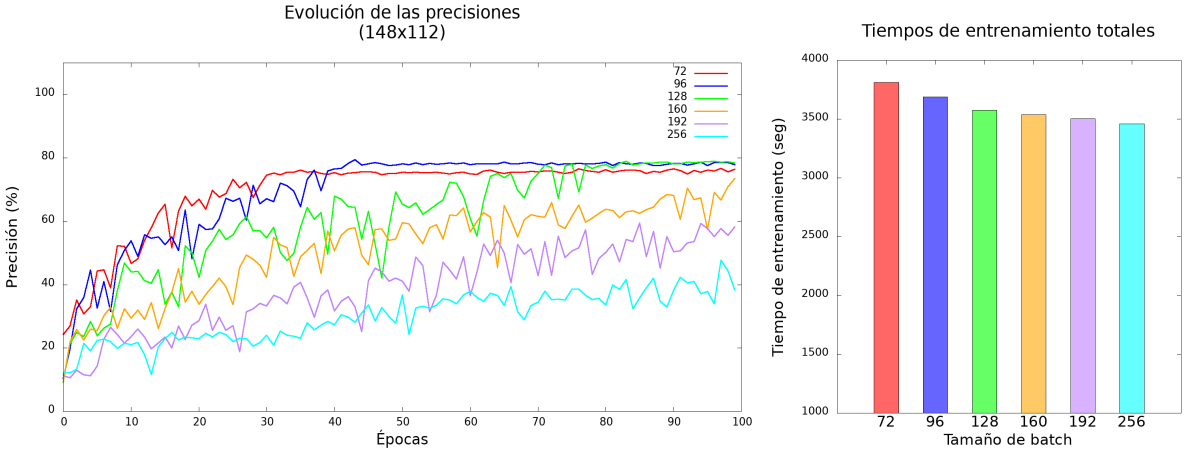

Figura 3.6: Comparativa de precisiones y tiempos de entrenamiento para muestras de 148x112 píxeles utilizando diferentes tamaños de batch. Izquierda: Evolución de las precisiones a lo largo de los epoch para tamaños de batch de 72, 96, 128, 160, 192 y 256 muestras. Derecha: Tiempos de entrenamiento totales para tamaños de batch de 72, 96, 128, 160, 192 y 256 muestras.

Por otro lado, la resolución 148x112, tiene unos tiempos razonables (aproximadamente 1 hora para 100 épocas), si bien se puede ver como, para tamaños de batch más grandes, la red encuentra más dificultades para generalizar y obtiene peores precisiones, no superando el 60 % para tamaños superiores a 128. Asimismo, también se puede observar que la reducción de tiempo que se obtiene de utilizar batches más grandes es lineal, pero minúscula, siendo alrededor de un 3 %.

Se decidió finalmente usar un tamaño de batch de 128, ya que, aunque tamaños menores producen una convergencia más rápido, era el que más memoria de los dispositivos GPU llenaba. Esta decisión se debe a que se estaba tratando de conseguir un benchmark realista pero que supusiera una carga de trabajo considerable para todos los componentes de la GPU.

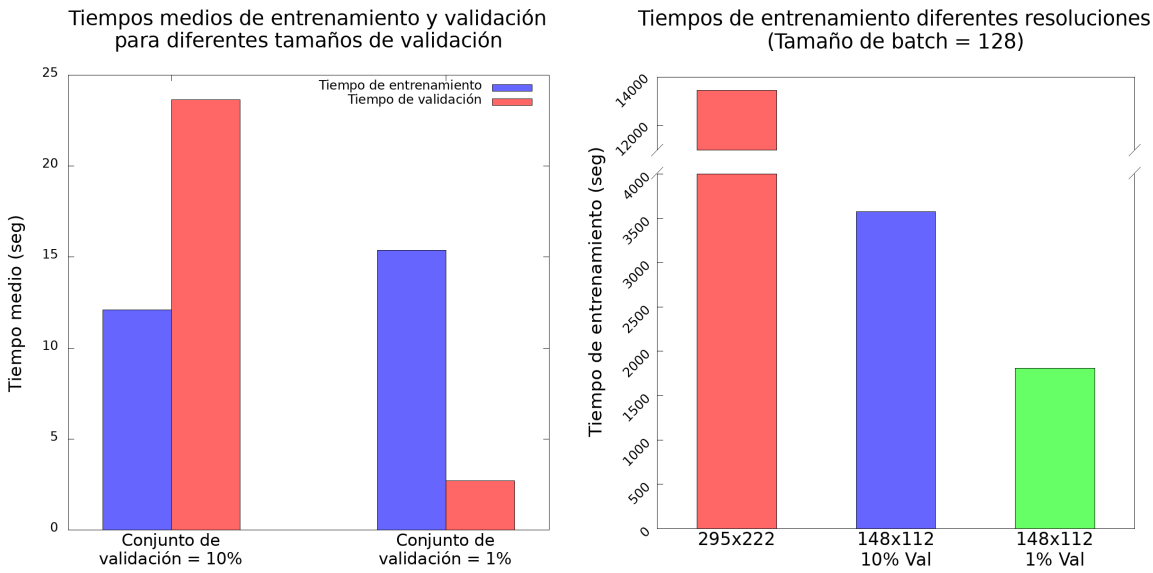

Figura 3.7: Comparativa de los tiempos de entrenamiento y validación para diferentes tamaños del conjunto de validación. Izquierda: Comparativa de los tiempos medios de entrenamiento y validación para muestras de 148x112 píxeles en batches de 128 para los diferentes tamaños del conjunto de validación. Derecha: Comparativa de los tiempos totales de entrenamiento para muestras de 148x112 y 295x222 píxeles, en batches de 128, para los diferentes tamaños del conjunto de entrenamiento.

Se observó durante estas pruebas que, debido al particionado escogido para los conjuntos (70 % para entrenamiento, 10 % para validación y 20 % para test), el tiempo de validación de cada época era superior al tiempo de entrenamiento. Como las validaciones en cada época solo se estaban usando para poder visualizar la evolución de la red a lo largo del entrenamiento, se decidió reducir considerablemente el tamaño de este conjunto, hasta que solo fuera de un 1 %. Como se puede observar en la figura 3.7 esto aportó una reducción de tiempo considerable.

Por último, se decidió usar una resolución de 197x148 (aproximadamente 1/9 de la resolución original), para dar más carga a la GPU y poder llenar más la memoria. Se eligió esta resolución final por ser un punto medio entre las 2 probadas anteriormente, aportando así más carga a las GPUs. Como consecuencia de esto, la precisión final no se vio afectada, y se volvieron a los tiempos de entrenamiento previos al ajuste del conjunto de validación, que se habían considerado como ideales para la prueba, 100 épocas en una hora.

Capítulo 4

Evaluación

Terminado el desarrollo del benchmark y escogidos los parámetros a utilizar, se pasó a definir los experimentos a realizar y a recoger, y analizar, los resultados obtenidos de ellos. En este capítulo se describen todos los parámetros del benchmark elegidos finalmente, así como los experimentos que se han llevado a cabo y los datos extraídos de éstos.

4.1. Parámetros Definitivos

Parámetro del programa	Valor
Tamaño del dataset	9000 muestras
Número de clases del dataset	9 clases
Tamaño de las muestras del dataset	197x148 px (ByN)
Tamaño del conjunto de entrenamiento	7650 muestras (85 %)
Tamaño del conjunto de validación	180 muestras (2 %)
Tamaño del conjunto de test	1170 muestras (13 %)
Tamaño del batch	128 muestras
Learning rate de la red	0.05
Algoritmo de gradient descent	Stochastic Gradient Descent
Número épocas del entrenamiento	100 épocas
Librería de comunicacion de procesos	NCCL

Tabla 4.1: Resumen de los parámetros utilizados finalmente para los experimentos

Las variables escogidas finalmente para el benchmark son las descritas a continuación. Se ha utilizado un tamaño de batch de 128, debido a las razones expuestas en la sección 3.4. Además de esto, se decidió usar los siguientes tamaños para los diferentes conjuntos: 85 % del dataset para entrenamiento, 2 % para validación y el 13 % restante para test; esto se debe a que se consideró que un 1 % (o 90 muestras) eran muy poco para el conjunto de validación, a pesar de que sólo se fuera a usar de manera orientativa. La *learning rate* del modelo ResNet utilizado fue de 0.05, como ya se expuso en la sección 3.4.

En cuanto a las muestras del dataset, se decidió finalmente redimensionarlas a 197x148 píxeles, por ser un buen punto medio entre las dos resoluciones probadas durante los experimentos de hiperparámetros. También se decidió convertirlas a blanco y negro, debido a que eran máscaras monocromas, a pesar de estar codificadas en *RGB*. Finalmente, se transformaron en matrices de números de punto flotante de 16 bits normalizados entre 0 y 1.

Uno de los parámetros más importantes a la hora de entrenar una red neuronal es el tipo de algoritmo de *Gradient Descent* usado. En este caso se ha utilizado el *Stochastic Gradient Descent*, que actualiza los pesos de la red tomando solo uno de los ejemplos de un batch para hacer propagación hacia atrás.

La razón para usar este tipo de *Gradient Descent* es que ayuda a la red a no quedarse atrapada en un mínimo local (una solución no tan óptima como podría ser) a costa de ser más inestable que los otros tipos de *Gradient Descent* [50].

Debido al número de muestras del dataset utilizado, se decidió que, para que la red alcanzase su máxima precisión, serían necesarias varias épocas (o iteraciones sobre los conjuntos de entrenamiento y de validación). Como número de épocas se decidió utilizar 100, dado que era el mismo número de épocas utilizado por Mattson et al. [37] en sus benchmarks de visión; además de esto, se observó que, para los hiperparámetros escogidos, la red alcanzaba su máxima precisión del 80 % unas épocas antes (para la mayoría de los experimentos realizados), cesando su mejora pasados los 100; esto se explicará más adelante, en la sección 4.3.2.

Por último, cabe destacar que se hizo uso de la librería de comunicaciones de procesos *NCCL* [43] para los benchmarks que usaban la API DDP de pytorch, dado que se observó que su uso de memoria VRAM y de CPU eran menores que los de la librería usada por defecto, *Gloo* [17]. Asimismo, Li et al muestran que la latencia de comunicaciones entre dispositivos en *NCCL* es menor que en *Gloo* [36].

4.2. Metodología

A continuación se detalla el equipo usado para los experimentos realizados, así como la metodología que siguieron dichos experimentos y las métricas que se han estudiado en ellos.

4.2.1. Equipo utilizado

Para el desarrollo de este proyecto se ha hecho uso del servidor del Grupo de Arquitectura y Tecnología de Computadores (ATC) de la Universidad de Cantabria, llamado *Batel*. Este servidor usa la distribución `CentOS 7` con la versión del kernel de Linux 3.10. Asimismo, cuenta con 2 CPUs y 2 GPUs, además de 16 GB de RAM.

Los CPUs son 2 Intel Xeon E5-2620 con 6 cores y 12 threads cada uno, funcionando a 2.0GHz (con una frecuencia máxima de 2.5GHz), y 15MB de caché [27]. Por otro lado, las GPUs son dos Nvidia Tesla K20m (GK110), con 2496 CUDA cores corriendo a una velocidad de 705MHz, y 5GB de VRAM. A pesar de que las GPUs pueden alcanzar una velocidad máxima de 750MHz, se fijó su velocidad a 705MHz, ya que es la frecuencia que recomienda Nvidia por ser un buen punto medio entre rendimiento y eficiencia energética. [44]

Cabe destacar que la arquitectura de estas GPUs (Compute Capability 3.5) no está soportada en Pytorch por defecto, con lo que fue necesario recompilar su código fuente en el servidor, activando las *flags* de compilación apropiadas.

4.2.2. Metodología de los experimentos

Para la realización del proyecto se diseñaron varios experimentos, con el objetivo de medir varios aspectos (como el tiempo de ejecución o el consumo energético). Se decidió examinar el rendimiento del benchmark diseñado en varias situaciones:

- Ejecutándolo solo en el CPU
- Ejecutándolo en una GPU, con y sin el *wrapper* de distribución
- Ejecutándolo en las dos GPUs del servidor a la vez

Para examinar el rendimiento del benchmark, se siguió la metodología propuesta a continuación. En total se realizaron 13 invocaciones del benchmark por cada experimento, 3 de esas ejecuciones fueron para perfilar el consumo energético de los componentes del servidor y las otras 10 sirvieron para medir los diferentes tiempos y precisiones de la red neuronal, dejando entre cada ejecución un tiempo de "enfriamiento" de 1 minuto, para que se descargasen los drivers de las GPUs. Se realizaron invocaciones diferentes del benchmark para perfilar la energía y los tiempos para así evitar que estos últimos pudieran verse afectados debido al *overhead* introducido por la herramienta de perfilado.

Para el perfilado de energía se consideró usar inicialmente la herramienta de Nvidia `nvprof`; sin embargo, tras encontrar fallos en los reportes de consumo energético de ésta (el consumo medio era a veces inferior al consumo mínimo reportado), se decidió usar `sauna` [46] [47], una herramienta de perfilado desarrollada por el Grupo ATC de la UC, que reporta el consumo de las CPUs, la RAM y las GPUs. La frecuencia de muestreo utilizada con `sauna` fue la que viene establecida por defecto, 500ms.

El esquema de los experimentos se puede ver en el pseudocódigo 4.1. Una de las pruebas no siguió este esquema; la ejecución de 13 invocaciones del benchmark se consideró excesiva para la prueba del CPU, ya que ésta solo se utilizaría como una línea base sobre la que comparar los resultados del resto. Para este experimento, por lo tanto, se hicieron tan solo 2 ejecuciones del benchmark, una para perfilar el consumo energético, y otra para obtener los tiempos de ejecución.

```
#!/bin/bash
sauna -oenergia1.dat -t benchmark.py e
sleep 1m

for i in  `seq 0 4`; do
    benchmark.py $i
    sleep 1m
done

sauna -oenergia2.dat -t benchmark.py e
sleep 1m


for i in `seq 5 9`; do
    benchmark.py $i
    sleep 1m
done

sauna -oenergia3.dat -t benchmark.py e
rm e
```

Pseudocódigo 4.1: Esquema de un experimento del proyecto. Las ejecuciones del benchmark que miden energia y tiempos se intercalan. Esto se hace para minimizar las desviaciones de los resultados.

Debido a los resultados iniciales de precisiones, se decidió además realizar otras 2 pruebas auxiliares, con motivo de complementarlos y poder confirmar ciertas hipótesis que surgieron. La primera prueba auxiliar fue la ejecución del benchmark 5 veces utilizando las 2 GPUs, pero con un tamaño de batch de 64 (las razones se discutirán más adelante, en la sección 4.3.2). La segunda prueba fue otra ejecución del benchmark de CPU, para poder determinar si la inestabilidad inicial que se observaba era resultado de una sola ejecución, o si por el contrario era un comportamiento innato del programa en CPU.

4.2.3. Medidas tomadas

Los datos que se decidió extraer de los experimentos realizados fueron:

- **La potencia instantánea** (en intervalos de 500ms) de los dispositivos, tanto de las dos CPUs como de su RAM, así como de las dos GPUs. La potencia se utilizó para calcular el consumo energético total y la eficiencia energética del sistema.

- **Los tiempos** de entrenamiento y de test totales de la red neuronal, así como los tiempos de entrenamiento y validación de cada epoch. Se decidió no medir el tiempo de ejecución total, ya que en él no solo se comprenderían los tiempos de entrenamiento y test, sino también el tiempo empleado en el cargado de los datos e inicialización del modelo, métricas que se decidió no tomar, siguiendo las reglas definidas en MLPerf Training[1] [37].

Tras obtener los tiempos de ejecución de cada una de las fases del benchmark (entrenamiento, validación y test), se procesaron, hallando las medias de las 10 invocaciones para cada uno de los experimentos. Además, se calcularon los speedups respecto de las pruebas de CPU y de una GPU sola, así como la eficiencia del sistema paralelo, usando para ello su speedup y número de dispositivos. Para el cálculo de estas métricas se utilizaron las fórmulas detalladas en las figuras 4.1 y 4.2.

$$S_{vs\ cpu} = \frac{T_{cpu}}{T_{disp}} \tag{4.1}$$

$$S_{vs\ gpu} = \frac{T_{gpu}}{T_{disp}} \tag{4.2}$$

Figura 4.1: Ecuaciones usadas para el cálculo de los speedups para los diferentes benchmarks respecto de los tiempos de ejecución totales de la CPU y de una GPU.

$$E_x = \frac{S_x}{x} \tag{4.3}$$

Figura 4.2: Ecuación del cálculo de la eficiencia de un sistema paralelo, donde E es la eficiencia, x es el número de dispositivos del sistema y S_x es el speedup de los x dispositivos respecto a 1.

- **La precisión** de cada época y la obtenida del conjunto de test tras finalizar el entrenamiento, así como los parámetros utilizados en el experimento para usar más tarde como referencia a la hora de clasificar los resultados.

4.3. Ejecución de las pruebas y resultados

Para facilitar la explicación de los resultados, se ha dividido esta sección en 3 apartados, cada una correspondiéndose con una de las métricas tomadas (tiempos, precisiones y consumo energético).

4.3.1. Tiempos

La figura 4.3 muestra los tiempos de cada fase del benchmark para cada prueba. Como se puede observar, el experimento que usaba las 2 GPUs fue el más rápido; además, la prueba que utilizaba el modelo distribuido con una sola GPU fue ligeramente más lenta que la de la GPU sin el modelo distribuido. También se puede ver que las fases de validación y test tienen un tiempo constante a lo largo de los experimentos; esto se debe a que dichas fases no pueden ser paralelizadas, ya que, de partir

[1] Estas reglas permiten un periodo de gracia de hasta 20 minutos para cargado e inicialización de estructuras de datos. En el caso de nuestro benchmark, estas operaciones duran aproximadamente 3 minutos.

los datos de dichas fases entre los dispositivos disponibles, cada dispositivo podría obtener una precisión diferente, lo que haría difícil determinar el verdadero rendimiento de la red neuronal.

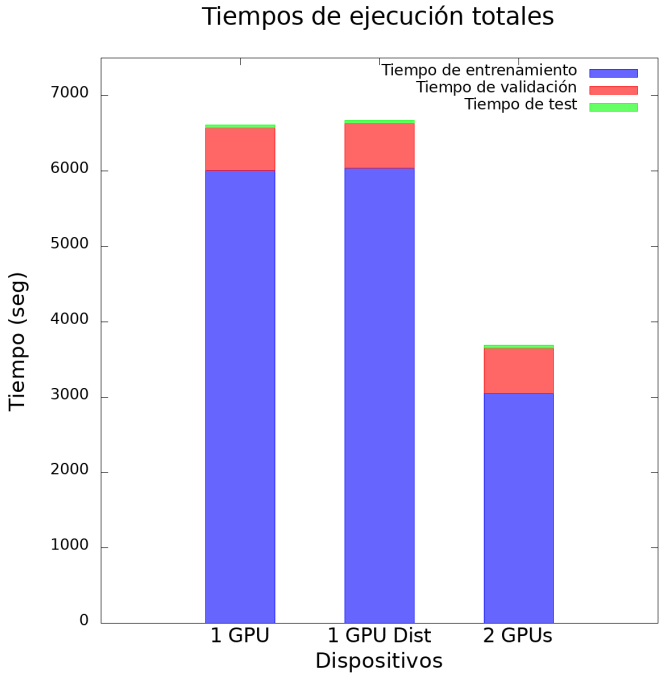

Figura 4.3: Tiempos de ejecución por cada fase de los experimentos. Tiempos del experimento de CPU omitidos por ser demasiado grandes (12 veces mayores, aproximadamente) y dificultar la lectura de las otras métricas.

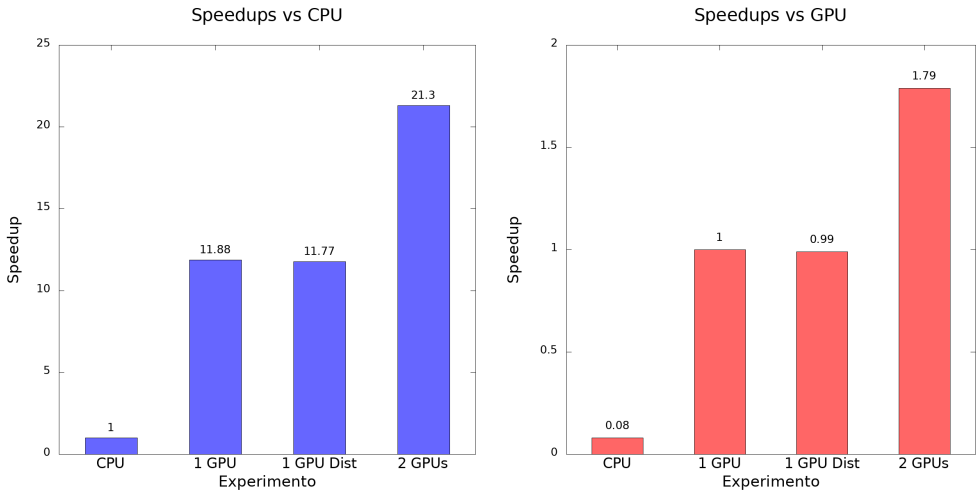

Figura 4.4: Speedups de los diferentes experimentos. Izquierda: speedups de las pruebas respecto de la CPU. Derecha: speedups de las pruebas respecto de una GPU sin usar el modelo distribuido.

Por otro lado, en la figura 4.4, donde se muestran los speedups de cada experimento, se ve que el paso de ejecutar el modelo en CPU a GPU proporciona una aceleración sustancial (del orden de 11.8 veces), mientras que el paso de la ejecución en una GPU a 2 GPUs es algo inferior a 2, quedándose en 1.79x. Esto se debe a la duración fija de las fases de validación y test (ya mencionado anteriormente). Si solo se tiene en cuenta la fase de entrenamiento[2], el speedup sube hasta el 1.97, no llegando a 2 debido a

[2]No se calcularon los speedups de solo esta fase, ya que se consideró que el benchmark debía representar el conjunto de todas las actividades que componen el entrenamiento de una red neuronal.

overheads de comunicación entre las dos gráficas. También se puede ver en esta figura como el speedup de una GPU usando el modelo distribuido es inferior a 1 respecto de la GPU sola. Esto, de nuevo, se debe a los *overheads* de comunicación que introduce la API DDP de Pytorch.

Estos resultados pueden considerarse muy positivos, ya que en el caso del benchmark paralelizado se obtiene una eficiencia, sobre el benchmark que utiliza 1 gpu, del 89.5 %. Si sólo se tiene en cuenta para los cálculos la parte paralelizada realmente (el bucle de entrenamiento) se obtiene una eficiencia muy cercana al 100 %, quedándose en 0.985, con lo que el sistema se está explotando prácticamente al máximo.

4.3.2. Precisiones

Pasemos ahora a realizar un análisis de las precisiones. Como ya se ha mencionado en apartados anteriores, se eligió iterar sobre el conjunto de entrenamiento 100 veces, validando la precisión de la red tras cada entrenamiento. De esta manera se pudo observar la evolución de la red a lo largo de la fase de entrenamiento. En la figura 4.5 se muestra la evolución de la red a lo largo de los epoch para cada experimento. Como se puede ver, la precisión del modelo entrenado en CPU es más inestable al principio, estabilizándose alrededor del epoch 50 y alcanzando una precisión del 80 %[3]. Los experimentos de GPU muestran en ambos casos más estabilidad en sus precisiones durante todo el entrenamiento, alcanzando una precisión máxima del 80 % alrededor del epoch 60.

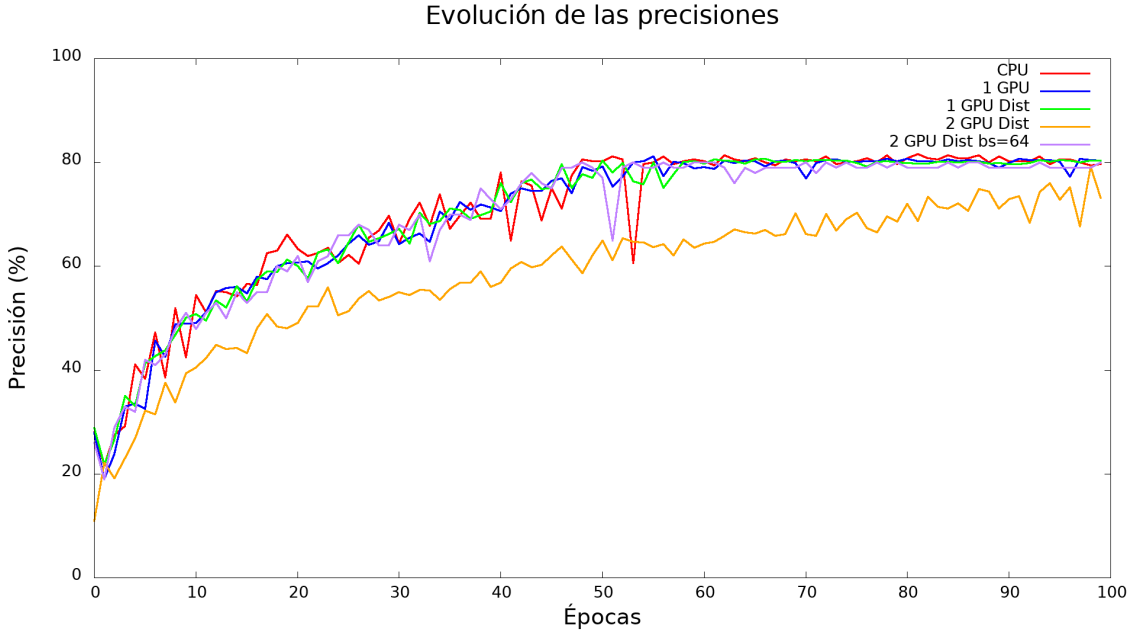

Figura 4.5: Evolución de las precisiones a lo largo de las épocas para cada experimento. Debido a los resultados obtenidos inicialmente, se decidió hacer una prueba extra del benchmark que usa las dos GPUs, usando un batch size de 64 muestras; en este experimento extraordinario solo se hicieron 5 invocaciones al benchmark, para medir precisiones.

Por otro lado, el experimento que utilizaba ambas GPUs muestra un crecimiento de la evolución más lento que el resto, mostrando incluso una bajada justo antes de la última iteración. Además, la red paralelizada no alcanza en los epoch especificados la misma precisión que las redes sin paralelizar, quedándose cerca, con un 74 %.

[3]Para las precisiones de CPU se realizaron dos invocaciones del benchmark.

Esta menor precisión del modelo paralelo (usando el mismo tamaño de batch que el resto de modelos) es resultado directo de la paralelización. Al recibir cada GPU la mitad de las muestras conservando el mismo tamaño de batch, la red se comporta como si estuviera siendo entrenada con batches de 256 muestras. Esto se debe a que se hacen la mitad de propagaciones hacia atrás y, aunque durante dichas propagaciones los modelos de ambas GPUs se sincronizan actualizando sus pesos en la misma cantidad, esto no es equivalente al número de propagaciones hacia atrás de los modelos no paralelizados. Una posible solución a esto es ajustar debidamente el tamaño de batch, para compensar el menor número de datos que ve cada modelo. Por ello se realizó un pequeño experimento de 5 invocaciones, usando el modelo paralelo con un tamaño de batch de 64 muestras. Como se puede observar, esta red sí se comporta igual que sus análogos no paralelizados, obteniendo la misma precisión final. Sin embargo, es necesario hacer un estudio más en profundidad de los resultados, para ver cuál es el impacto que el tamaño de batch tiene en el tiempo de entrenamiento y, por tanto, en el speedup respecto a una GPU.

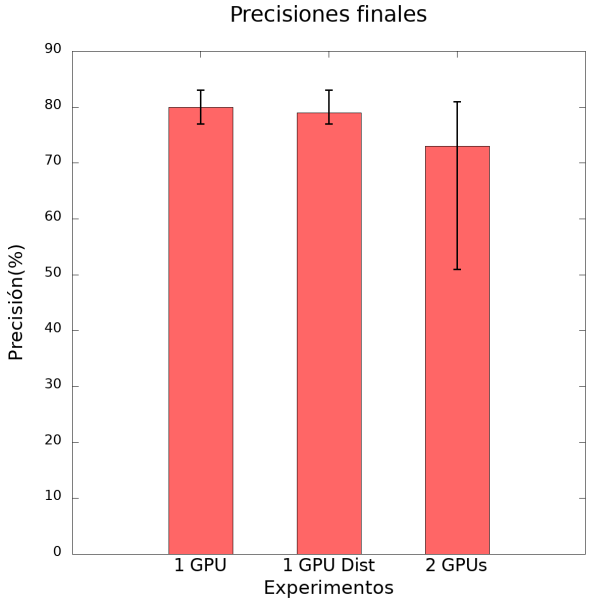

Figura 4.6: Precisiones medias finales de cada experimento. También se muestran tanto la precisión mínima obtenida como la máxima de cada experimento. Los resultados de la prueba en CPU no se muestran debido a la falta de datos para poder calcular medias, mínimas y máximas.

Además de todo esto, y como se puede ver en la figura 4.6, el modelo entrenado en ambas GPUs muestra más variabilidad en su precisión final. Esto se debe, de nuevo, a la necesidad de ajustar el tamaño de batch en función del número de dispositivos. Por otro lado, podemos ver que, aunque sus tiempos diferían ligeramente, los dos experimentos que hacían uso de una sola GPU obtuvieron idénticos resultados en términos de precisión, lo que era de esperar. Por último, cabe destacar que no se han incluido los datos del experimento que solo utilizaba CPU para la figura 4.6 debido a que solo se tenían datos de dos ejecuciones, por el diseño de los experimentos (como se detallan en la sección 4.2), con lo que no se tenían métricas suficientes para mostrar la varianza de las precisiones finales para esa prueba.

4.3.3. Consumo energético

Para los experimentos se midió, a intervalos regulares, la potencia de cada uno de los dispositivos del servidor. Las figuras 4.7(Izqda) y 4.8 muestran la evolución de esas potencias a lo largo de los experimentos de 1 GPU (usando o no el *wrapper* de distribución) y de 2 GPUs; además, muestran una región de la gráfica ampliada. Como se puede observar, en el experimento de 1 GPU los CPUs se usan de manera alternante, esto se debe a que se turnan manejando las comunicaciones con la GPU. Los experimentos que usan el *wrapper* de distribución no muestran este comportamiento, recibiendo cada

GPU un procesador asignado. En los 3 experimentos se distingue un patrón en la potencia de las GPUs (esto se puede ver claramente en la sección ampliada) en el que ésta oscila entre 150 y 120 W durante unos 50 segundos, bajando después a 95 W (durante 5 segundos). Esto se corresponde con cada una de las épocas del benchamrk, donde la parte de entrenamiento requiere más potencia al utilizar muestras en batches y tener que hacer propagaciones hacia atrás.

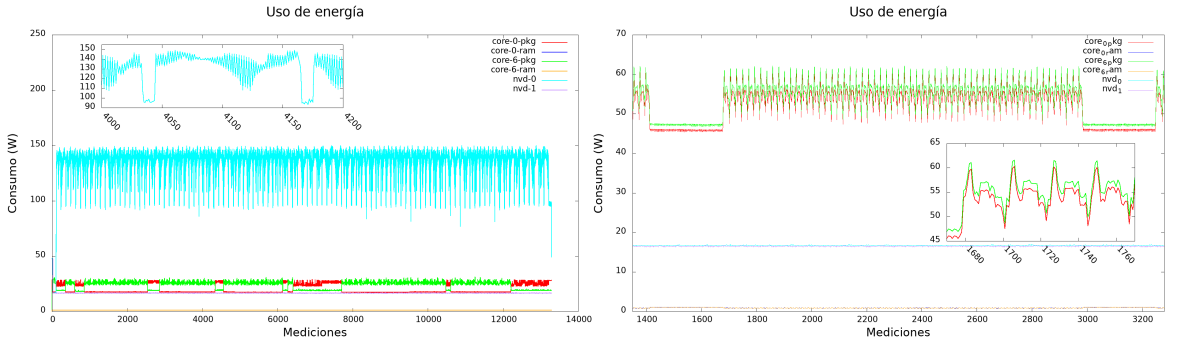

Figura 4.7: Potencias instantáneas de los diferentes dispositivos a a lo largo del tiempo (intervalo de muestreo = 500ms). Izquierda: Experimento con una sola GPU. Derecha: Región de interés del experimento de CPU.

Un comportamiento parecido se puede apreciar en la figura 4.7(Drcha), en la que se muestra una sección del experimento de CPU y una región ampliada de dicha sección. El uso de potencia de los CPUs también sigue el patrón de alto consumo durante el entrenamiento y menor consumo durante la validación (esta vez alternando entre los 50/60 W y los 45W). Además, se distingue otro uso de energía cíclico dentro del entrenamiento; éste se corresponde con cada uno de los batches del conjunto de entrenamiento, donde el pico de potencia se debe a la propagación hacia delante de las 128 muestras, la llanura del medio a la propagación hacia atrás de solo una de las muestras (siguiendo el algoritmo de SGD [50]) y el valle con la descarga de las estructuras de datos, en preparación para la siguiente iteración.

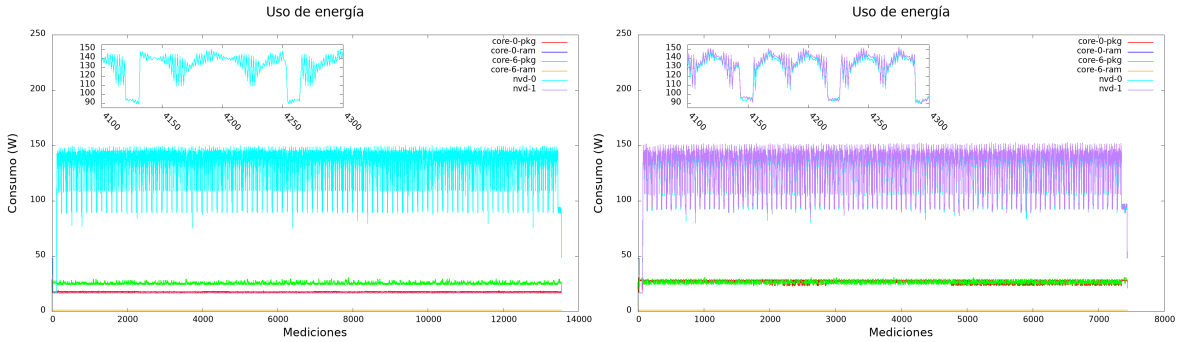

Figura 4.8: Potencias instantáneas de los diferentes dispositivos a lo largo del tiempo (intervalo de muestreo = 500ms). Izquierda: Experimento con una sola GPU usando el modelo distribuido. Derecha: Experimento con 2 GPUs

Cabe destacar además que, como se puede ver en las figuras 4.7 y 4.8(izqda), el uso de los dispositivos en *idle*[4] es alrededor de 16 W, tanto para las GPUs como para los CPUs.

Tras obtener las potencias de los experimentos, se pasó a calcular el consumo energético de los dispositivos usados en cada una de las pruebas. La energía se podría considerar como el área bajo la curva

[4]Los dispositivos que no están siendo usados

de potencia instantánea a lo largo del tiempo, como se ilustra en la ecuación 4.9. Sin embargo, dado que se tenían datos discretos de la potencia instantánea, y que estos venían de muestreos en intervalos regulares, se optó por usar la ecuación 4.10, donde se suman los valores de todas las lecturas realizadas y se multiplican por el tiempo entre cada lectura (en este caso 500ms).

$$e = \int_0^t P\,dt \tag{4.4}$$

Figura 4.9: Cálculo de la energía en base a la potencia y el tiempo, donde e es la energía, t es el tiempo total, y P es la potencia en función del tiempo.

$$e = t_m \cdot \sum_{i=0}^{n} P[i] \tag{4.5}$$

Figura 4.10: Cálculo de la energía en base a mediciones de potencia discretas en intervalos de tiempo regulares, donde e es la energía, t_m es el intervalo de tiempo entre muestreos (en segundos), n es el total de mediciones de potencia realizadas y $P[x]$ es la lista que contiene todas las mediciones de potencia.

Asimismo, se hicieron también cálculos de eficiencia energética, para lo que se utilizó el *Energy Delay Product (EDP)* normalizado respecto del experimento de 1 sola GPU [6] [5]. La ecuación 4.11 es la utilizada para los cálculos de EDP[5]. Para los resultados expuestos en las figuras a continuación, solo se hicieron los cálculos para los componentes utilizados en cada experimento, ignorando los que estuvieron en *idle* (como las GPUs durante el experimento de cpu).

$$EDP = t \cdot \sum e_d \tag{4.6}$$

Figura 4.11: Fórmula para el cálculo del EDP, donde e_d es la energía consumida por el dispositivo d y t es el tiempo empleado en consumir dicha energía.

En la figura 4.12, que muestra el consumo energético medio de cada experimento y su EDP total, se puede ver que el consumo energético del experimento de CPU fue el más alto, siendo casi una orden de magnitud más grande que el resto de las pruebas (con un consumo de 8.5 millones de Julios). También podemos ver que el consumo de los dispositivos activos en las otras tres pruebas es más o menos igual. Sin embargo, si miramos el EDP de cada experimento podemos ver que el que usa las 2 GPUs tiene un EDP de 0.56 veces el de una GPU, siendo más eficiente energéticamente, ya que consume la misma energía en algo más de la mitad del tiempo. También podemos ver que el EDP del CPU es 86 veces superior al de 1 GPU, con lo que es muy ineficiente. En cuanto al uso del modelo distribuido para una sola GPU, este prueba ser un poco menos eficiente que el que no hace uso de la distribución, teniendo un EDP de 1.02x.

Como ya se ha mencionado antes, estos cálculos se hicieron sin considerar los dispositivos no utilizados durante cada experimento; sin embargo, de considerarse dichos consumos, la prueba que usa sólo CPU sería la más afectada, subiendo su EDP hasta 104 veces y su consumo pasando de los 8.5MJ a 11MJ. Esto se debe a que, a pesar de no estar siendo utilizados, los componentes electrónicos utilizan cierta energía para poder ser activados cuando sea necesario. Es por esta misma razón por la que el experimento que usa 2 GPUs sería aún más eficiente, siendo su EDP 0.51x el de 1 sola GPU, ya que es la única prueba en la que todos los componentes del servidor estaban siendo utilizados a la vez.

[5]Para la normalización se dividieron los valores obtenidos de cada uno de los experimentos por el de 1 GPU.

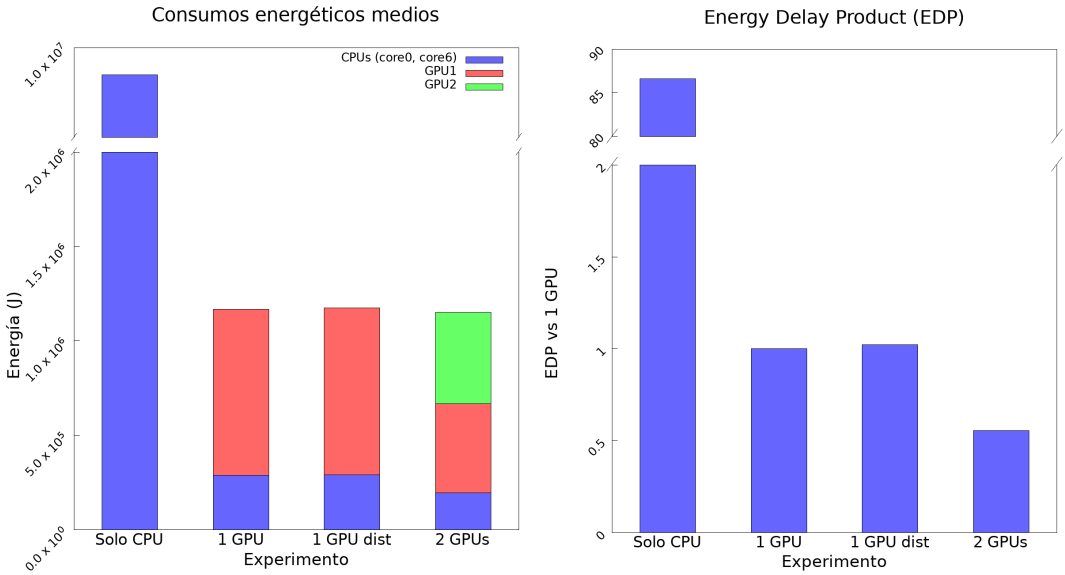

Figura 4.12: Consumo medio y eficiencia energética de los diferentes experimentos. Izquierda: Consumo energético medio de las pruebas en función de los dispositivos utilizados. Derecha: Energy Delay Product (EDP) de los diferentes experimentos, normalizado contra el experimento de 1 GPU.

Experimento	**Consumo (J)**	**Consumo con idle (J)**	**EDP**	**EDP con idle**
Solo CPU	8.48E+06	1.11E+07	86.65	103.79
1 GPU	1.17E+06	1.28E+06	1.00	1.00
1 GPU Distribuido	1.18E+06	1.29E+06	1.02	1.02
2 GPUs	1.15E+06	1.15E+06	0.56	0.51

Tabla 4.2: Tabla de los consumos y EDPs de los diferentes experimentos, con y sin tener en cuenta los dispositivos no utilizados.

En resumen, la paralelización de una red neuronal es capaz de acelerar el proceso de entrenamiento hasta casi dos veces, brindando además una mayor eficiencia energética, gracias a una mejor utilización de los dispositivos disponibles en el equipo. Sin embargo, para que esta paralelización pueda dar los mismos resultados que la red sin paralelizar, es necesario tener en cuenta ciertos factores, como la distribución de los datos, que se ven influenciados por el número de dispositivos a usar. Por ejemplo, es necesario ajustar el tamaño de los batches en función del número de dispositivos disponibles, ya que la red se comportará como si estuviera recibiendo batches de $(tam.\ batch) \cdot (num.\ dispositivos)$.

Capítulo 5

Conclusiones y trabajos futuros

5.1. Conclusiones

Como se puede ver en los resultados obtenidos, se han alcanzado satisfactoriamente todos los objetivos definidos al principio del proyecto. Por un lado, el benchmark implementado es capaz de entrenar una red de manera eficaz, alcanzando el 80 % de precisión final en la mayoría de entornos de ejecución. Además, para el modelo paralelo, el benchmark explota todos los dispositivos al máximo, ya que ambas GPUs exhiben el mismo patrón de uso y su consumo energético es el mismo. Por otro lado, la elección del modelo de red neuronal y del dataset fueron acertados, proporcionando un entrenamiento con número de épocas y de una duración aceptable; gracias a esto y a los parámetros escogidos para el benchmark, éste consigue ser representativo de la realidad.

Asimismo, gracias a los experimentos diseñados y a las métricas recogidas, queda patente que la paralelización del proceso de entrenamiento obtiene ganancias en rendimiento. El modelo paralelo escala de manera casi lineal, con un speedup de 1.97x en la parte paralelizada, respecto al modelo que sólo explota una GPU. Cabe destacar además que, si bien el consumo energético es muy similar ente el modelo multi-GPU y el mono-GPU, la eficiencia del primero es casi el doble de la del segundo, con un EDP de 0.51 respecto a este último, gracias a su mejor utilización de los dispositivos disponibles, así como de su menor tiempo de entrenamiento.

Gracias a estos resultados, se han obtenido las siguientes conclusiones sobre el entrenamiento de redes neuronales en sistemas heterogéneos:

La paralelización del entrenamiento de redes neuronales es una técnica esencial para poder acelerar estos procesos, que generalmente son muy largos y suponen más carga de trabajo que el uso de la red en sí. Sin embargo, esta paralelización no escala linealmente, debido a *overheads* de comunicación y a la existencia de actividades no paralelizables dentro del entrenamiento. Por lo tanto, es importante tener en cuenta varios aspectos si se quiere maximizar el rendimiento y los resultados obtenidos del entrenamiento.

En primer lugar hay que adaptar la distribución del trabajo en función del número de dispositivos disponibles, para garantizar unos resultados óptimos del proceso. Dentro de esta adaptación se encuentran actividades tan cruciales como determinar cuántos datos ha de recibir cada dispositivo y cómo. Como se ha discutido en la sección 4.3, puede ser necesario el ajuste de los tamaños de batch en función del número de dispositivos a utilizar, reduciéndolo proporcionalmente; sin embargo, esto podría llevar a mayor tiempo de entrenamiento, aunque, como se discutió en la sección 3.4, el impacto que tiene el tamaño de los batches en este tiempo de entrenamiento es menor que el que tienen otros factores, como el tamaño de los conjuntos o el tamaño de las muestras.

Otra de las medidas a tomar para poder minimizar el tiempo de entrenamiento en ambientes distribuidos es la maximización del tamaño del conjunto de entrenamiento, para así poder reducir el tiempo de las actividades no paralelizables. En el caso del benchmark propuesto en este proyecto, se podría reducir de nuevo el tamaño del conjunto de validación a 1 % o incluso eliminarlo (sacrificando entonces la visualización de la precisión a lo largo de los epoch), ya que este conjunto se usa de manera orientativa y no sirve otro propósito más que el de poder monitorizar el avance de la red. Para entrenamientos de modelos que van a ser desplegados y usados en el mundo real, sin embargo, no es posible eliminar este conjunto ya que es necesario para detectar cuándo se produce *overfitting*[1] y así parar el entrenamiento o intentar mitigarlo de alguna manera [52]. Otra posible solución para reducir el tiempo de las validaciones, que (como se puede ver en la figura 4.3) es el conjunto no paralelizable que más tiempo contribuye a la duración del benchmark, sería reducir la frecuencia de éstas. En vez de realizar una validación cada época, se podría realizar cada 2 o 5, a cambio de aumentar el riesgo de no detectar *overfitting* temprano. Todas estas opciones deben ser valoradas cuando se diseña y construye un modelo de red neuronal y su programa de entrenamiento.

Además de esto, para poder minimizar el costo energético de entrenar a la red es necesario emplear todos los dispositivos disponibles en conjunto, ya que solo el consumo en *standby* de ciertos dispositivos puede suponer un gasto innecesario de energía (cuando se compara contra el rendimiento que se obtendría usando todos los dispositivos). Asimismo, es necesario emplear cada componente del equipo para la tarea en la que es más eficiente; por ejemplo, dado que el CPU tiene acceso a la memoria RAM y virtual del equipo (y que estas suelen tener mayor capacidad que la VRAM de aceleradores gráficos) sería mejor utilizarlo para distribuir los datos y recoger los resultados, mientras que las GPUs (que son capaces de realizar operaciones paralelas más eficientemente) computan las soluciones dados los datos que les entrega el CPU. Además, si se tiene un equipo con un solo dispositivo, es mejor diseñar un programa de entrenamiento que no haga uso de APIs de paralelización, ya que (aunque el costo de rendimiento y consumo no es muy elevado) se introduce cierto *overhead*; como se puede ver en la figura 4.8(Drcha) del capítulo 4, donde el benchmark no hizo uso de todos los CPUs disponibles.

5.2. Trabajos Futuros

Tras finalizar este proyecto hay ciertos aspectos en los que se podría indagar y profundizar. Uno de ellos sería la realización de más pruebas de rendimiento, estudiando más en detalle cómo afecta el escalado de una red neuronal a lo largo de varios dispositivos al tamaño de batch necesario para obtener resultados óptimos. Asimismo, se podría ampliar el catálogo de experimentos, ejecutando el benchmark en equipos con mayor número de dispositivos, incluyendo dispositivos heterogéneos (como GPUs con diferentes prestaciones), o con nodos de cómputo distribuidos a lo largo de una red, para así poder evaluar la escalabilidad de una red neuronal en dichos ambientes.

Otra posibilidad sería mejorar el sistema de reparto de carga actual del benchmark distribuido para que ajuste el tamaño de batch proporcionalmente al número de dispositivos, tal y como se hace con el tamaño del conjunto de entrenamiento. Además, se podría implementar una detección del hardware subyaciente, ajustando las proporciones de los conjuntos de acuerdo a las capacidades de cada uno de los dispositivos detectados.

También sería interesante profundizar en el desarrollo de un modelo de paralelización que haga uso del CPU para algo más que gestionar comunicaciones y repartir datos. Aunque durante el proyecto se determinó que no era viable, solo se tuvo en cuenta un framework de redes neuronales (Pytorch) y un paradigma de paralelización (*data-parallel*); sin embargo, existen otros frameworks de redes neuronales (como tensorflow [1]). Además, se podría probar el paradigma de paralelización *model-parallel*, deter-

[1] Fenómeno de *machine learning* en el que una red pasa de generalizar patrones a memorizar los datos del conjunto de entrenamiento, esto ocurre en entrenamientos con muchas épocas.

minando cuáles de las capas del modelo de red neuronal explotarían mejor la CPU. Se podría incluso proponer un paradigma que combinase los mejores aspectos de *model-parallel* y *data-parallel*.

Además de esto, se podría realizar un estudio sobre la eficiencia energética del modelo híbrido desarrollado, para ver cómo se compara contra el benchmark que solo utiliza CPU y el que usa GPU. Asimismo, se podría complementar el estudio de energía realizado, añadiendo los datos de consumo y eficiencia del benchmark que hace uso de 2 GPUs con el tamaño de batch ajustado (de 64 muestras), de esta manera se podrían comparar benchmarks que producen resultados similares. También sería una posibilidad el desarrollo de un perfilado más exhaustivo del uso de las GPUs, estudiando cuánta potencia usa cada componente de las mismas (de la misma manera que `sauna` permite perfilar la potencia tanto de los CPUs como de la RAM asociada a ellos) para nuestro benchmark.

Por último, en este proyecto sólo se han estudiado los efectos de la paralelización de una red neuronal durante su entrenamiento. Sin embargo, podría ser provechoso evaluar los efectos de la paralelización de una red ya desplegada (es decir, que haya sido pre-entrenada), definiendo varios escenarios de llegada de datos, como hacen Reddi et al. [48] en sus benchmarks de MLPerf Inference.

Bibliografía

[1] Martın Abadi y col. «TensorFlow: A System for Large-Scale Machine Learning». En: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. Savannah, GA: USENIX Association, nov. de 2016, págs. 265-283. ISBN: 978-1-931971-33-1. URL: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi.

[2] Robert Adolf y col. «Fathom: Reference workloads for modern deep learning methods». En: *2016 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE. 2016, págs. 1-10.

[3] Baidu. *Baidu-Research, DeepBench*. URL: https://github.com/baidu-research/DeepBench. (accessed: 30-6-2021).

[4] Stephen Cass. «Taking AI to the edge: Google's TPU now comes in a maker-friendly package». En: *IEEE Spectrum* 56.5 (2019), págs. 16-17.

[5] Emilio Castillo y col. «Architectural Support for Task Dependence Management with Flexible Software Scheduling». En: *IEEE International Symposium on High Performance Computer Architecture, HPCA 2018, Vienna, Austria, February 24-28, 2018*. IEEE Computer Society, 2018, págs. 283-295. DOI: 10.1109/HPCA.2018.00033. URL: https://doi.org/10.1109/HPCA.2018.00033.

[6] Emilio Castillo y col. «CATA: Criticality Aware Task Acceleration for Multicore Processors». En: *2016 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2016, Chicago, IL, USA, May 23-27, 2016*. IEEE Computer Society, 2016, págs. 413-422. DOI: 10.1109/IPDPS.2016.49. URL: https://doi.org/10.1109/IPDPS.2016.49.

[7] Emilio Castillo y col. «Financial applications on multi-CPU and multi-GPU architectures». En: *J. Supercomput.* 71.2 (2015), págs. 729-739. DOI: 10.1007/s11227-014-1316-5. URL: https://doi.org/10.1007/s11227-014-1316-5.

[8] Sharan Chetlur y col. «cudnn: Efficient primitives for deep learning». En: *arXiv preprint arXiv:1410.0759* (2014).

[9] Cody A. Coleman y col. «DAWNBench : An End-to-End Deep Learning Benchmark and Competition». En: 2017.

[10] Jia Deng y col. «Imagenet: A large-scale hierarchical image database». En: *2009 IEEE conference on computer vision and pattern recognition*. Ieee. 2009, págs. 248-255.

[11] Shi Dong y David Kaeli. «DNNMark: A Deep Neural Network Benchmark Suite for GPUs». En: *Proceedings of the General Purpose GPUs*. GPGPU-10. Austin, TX, USA: Association for Computing Machinery, 2017, págs. 63-72. ISBN: 9781450349154. DOI: 10.1145/3038228.3038239. URL: https://doi.org/10.1145/3038228.3038239.

[12] David L Ehret y col. «Neural network modeling of greenhouse tomato yield, growth and water use from automated crop monitoring data». En: *Computers and Electronics in Agriculture* 79.1 (2011), págs. 82-89.

[13] Facebook. *Pytorch docs - CUDA semantics*. URL: https://pytorch.org/docs/stable/notes/cuda.html. (accessed: 16-11-2020).

[14] Facebook. *Pytorch docs - Data Parallel*. URL: https://pytorch.org/docs/stable/generated/torch.nn.DataParallel.html. (accessed: 10-11-2020).

[15] Facebook. *Pytorch docs - Distributed Data Parallel*. URL: https://pytorch.org/docs/stable/notes/ddp.html. (accessed: 10-11-2020).

[16] Facebook. *Pytorch docs - torch.utils.data*. URL: https://pytorch.org/docs/stable/data.html. (accessed: 8-12-2020).

[17] FacebookIncubator. *facebookincubator/gloo: Collective communications library with various primitives for multi-machine training*. URL: https://github.com/facebookincubator/gloo. (accessed: 3-5-2021).

[18] Michael Garland. «Parallel computing with CUDA». En: *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*. IEEE Computer Society. 2010, págs. 1-1.

[19] Mehdi Goli, Luke Iwanski y Andrew Richards. «Accelerated Machine Learning Using TensorFlow and SYCL on OpenCL Devices». En: *Proceedings of the 5th International Workshop on OpenCL*. IWOCL 2017. Toronto, Canada: Association for Computing Machinery, 2017. ISBN: 9781450352147. DOI: 10.1145/3078155.3078160. URL: https://doi.org/10.1145/3078155.3078160.

[20] Google. *Tensorflow guide, distributed training strategies*. URL: https://www.tensorflow.org/guide/distributed_training. (accessed: 18-10-2020).

[21] Google. *Tensorflow guide, keras API*. URL: https://www.tensorflow.org/guide/keras. (accessed: 18-10-2020).

[22] Google. *Tensorflow guide, use TPUs*. URL: https://www.tensorflow.org/guide/tpu. (accessed: 18-10-2020).

[23] Khronos Group. *SYCL 2020 Especification*. URL: https://www.khronos.org/registry/SYCL/specs/sycl-2020/pdf/sycl-2020.pdf. (accessed: 1-6-2021).

[24] Maria Angelica Davila Guzman y col. «Cooperative CPU, GPU, and FPGA heterogeneous execution with EngineCL». En: *J. Supercomput.* 75.3 (2019), págs. 1732-1746. DOI: 10.1007/s11227-019-02768-y. URL: https://doi.org/10.1007/s11227-019-02768-y.

[25] Azzam Haidar y col. «Harnessing GPU tensor cores for fast FP16 arithmetic to speed up mixed-precision iterative refinement solvers». En: *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE. 2018, págs. 603-613.

[26] Kaiming He y col. «Deep Residual Learning for Image Recognition». En: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016, págs. 770-778. DOI: 10.1109/CVPR.2016.90.

[27] Intel. *Procesador Intel Xeon E5-2620*. URL: https://ark.intel.com/content/www/es/es/ark/products/64594/intel-xeon-processor-e5-2620-15m-cache-2-00-ghz-7-20-gt-s-intel-qpi.html. (accessed: 28-6-2021).

[28] Sebastian Jäger y col. «Parallelized training of Deep NN: comparison of current concepts and frameworks». En: *Proceedings of the Second Workshop on Distributed Infrastructures for Deep Learning*. 2018, págs. 15-20.

[29] Aajna Karki y col. «Tango: A deep neural network benchmark suite for various accelerators». En: *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE. 2019, págs. 137-138.

[30] Tero Karras, Samuli Laine y Timo Aila. «A Style-Based Generator Architecture for Generative Adversarial Networks». En: *CoRR* abs/1812.04948 (2018). arXiv: 1812.04948. URL: http://arxiv.org/abs/1812.04948.

[31] Nikhil Ketkar. «Introduction to PyTorch». En: *Deep Learning with Python: A Hands-on Introduction*. Berkeley, CA: Apress, 2017, págs. 195-208. ISBN: 978-1-4842-2766-4. DOI: 10.1007/978-1-4842-2766-4_12. URL: https://doi.org/10.1007/978-1-4842-2766-4_12.

[32] Yann LeCun y Corinna Cortes. «MNIST handwritten digit database». En: (2010). URL: http://yann.lecun.com/exdb/mnist/.

[33] J. Lee y col. «Chapter 2 - SnuCL: A unified OpenCL framework for heterogeneous clusters». En: *Advances in GPU Research and Practice*. Ed. por Hamid Sarbazi-Azad. Emerging Trends in Computer Science and Applied Computing. Boston: Morgan Kaufmann, 2017, págs. 23-55. ISBN: 978-0-12-803738-6. DOI: https://doi.org/10.1016/B978-0-12-803738-6.00002-1. URL: https://www.sciencedirect.com/science/article/pii/B9780128037386000021.

[34] Cheng Li y col. «Xsp: Across-stack profiling and analysis of machine learning models on gpus». En: *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE. 2020, págs. 326-327.

[35] Shen Li. *Single-Machine Model Parallel Best Practices*. URL: https://pytorch.org/tutorials/intermediate/model_parallel_tutorial.html. (accessed: 10-11-2020).

[36] Shen Li y col. «PyTorch Distributed: Experiences on Accelerating Data Parallel Training». En: *Proc. VLDB Endow.* 13.12 (ago. de 2020), págs. 3005-3018. ISSN: 2150-8097. DOI: 10.14778/3415478.3415530. URL: https://doi.org/10.14778/3415478.3415530.

[37] Peter Mattson y col. *MLPerf Training Benchmark*. 2020. arXiv: 1910.01500 [cs.LG].

[38] Saiful A Mojumder y col. «Profiling DNN workloads on a volta-based DGX-1 system». En: *2018 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE. 2018, págs. 122-133.

[39] Aaftab Munshi. «The opencl specification». En: *2009 IEEE Hot Chips 21 Symposium (HCS)*. IEEE. 2009, págs. 1-314.

[40] John Nickolls y William J Dally. «The GPU computing era». En: *IEEE micro* 30.2 (2010), págs. 56-69.

[41] Raúl Nozal, José Luis Bosque y Ramón Beivide. «EngineCL: Usability and Performance in Heterogeneous Computing». En: *Future Gener. Comput. Syst.* 107 (2020), págs. 522-537. DOI: 10.1016/j.future.2020.02.016. URL: https://doi.org/10.1016/j.future.2020.02.016.

[42] Raúl Nozal y col. «Load balancing in a heterogeneous world: CPU-Xeon Phi co-execution of data-parallel kernels». En: *J. Supercomput.* 75.3 (2019), págs. 1123-1136. DOI: 10.1007/s11227-018-2318-5. URL: https://doi.org/10.1007/s11227-018-2318-5.

[43] Nvidia. *NCCL Documentation, Overview*. URL: https://docs.nvidia.com/deplearning/nccl/user-guide/docs/overview.html. (accessed: 2-5-2021).

[44] Nvidia. *Nvidia Kepler GK110 architecture whitepaper*. URL: https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/tesla-product-literature/NVIDIA-Kepler-GK110-GK210-Architecture-Whitepaper.pdf. (accessed: 28-6-2021).

[45] Çağlar Fırat Özgenel. *Concrete Crack Images for Classification*. DOI: 10.17632/5y9wdsg2zt.2. URL: https://data.mendeley.com/datasets/5y9wdsg2zt/2.

[46] Borja Pérez y col. «Energy efficiency of load balancing for data-parallel applications in heterogeneous systems». En: *J. Supercomput.* 73.1 (2017), págs. 330-342. DOI: 10.1007/s11227-016-1864-y. URL: https://doi.org/10.1007/s11227-016-1864-y.

[47] Borja Pérez y col. «Sigmoid: an auto-tuned load balancing algorithm for heterogeneous systems». En: *Journal of Parallel and Distributed Computing* (2021). ISSN: 0743-7315. DOI: https://doi.org/10.1016/j.jpdc.2021.06.003. URL: https://www.sciencedirect.com/science/article/pii/S0743731521001325.

[48] Vijay Janapa Reddi y col. *MLPerf Inference Benchmark*. 2020. arXiv: 1911.02549 [cs.LG].

[49] Mohammad Hossein Rezaei y col. «Applying GMDH artificial neural network in modeling CO2 emissions in four nordic countries». En: *International Journal of Low-Carbon Technologies* 13.3 (2018), págs. 266-271.

[50] Sebastian Ruder. «An overview of gradient descent optimization algorithms». En: *arXiv preprint arXiv:1609.04747* (2016).

[51] Stuart Russell y Peter Norvig. *Artificial Intelligence: A Modern Approach*. 3.ª ed. Prentice Hall, 2010.

[52] Warren S Sarle. «Stopped training and other remedies for overfitting». En: *Computing science and statistics* (1996), págs. 352-360.

[53] Li Tao y col. «LLCNN: A convolutional neural network for low-light image enhancement». En: *2017 IEEE Visual Communications and Image Processing (VCIP)*. IEEE. 2017, págs. 1-4.

[54] Pablo Toharia y col. «Shot boundary detection using Zernike moments in multi-GPU multi-CPU architectures». En: *J. Parallel Distributed Comput.* 72.9 (2012), págs. 1127-1133. DOI: 10.1016/j.jpdc.2011.10.011. URL: https://doi.org/10.1016/j.jpdc.2011.10.011.

[55] Oguzhan Ulucan, Diclehan Karakaya y Mehmet Turkan. «A Large-Scale Dataset for Fish Segmentation and Classification». En: *2020 Innovations in Intelligent Systems and Applications Conference (ASYU)*. IEEE. 2020, págs. 1-5.

[56] Haohan Wang y Bhiksha Raj. «On the origin of deep learning». En: *arXiv preprint arXiv:1702.07800* (2017).

[57] Phillip Wang. *This person does not exist*. URL: https://thispersondoesnotexist.com/. (accessed: 28-6-2021).

[58] Francesco Zuppichini. *Residual Networks: Implementing ResNet in Pytorch*. URL: https://bit.ly/2SG8S3m. (accessed: 6-3-2021).