Evaluación de Rendimiento del Entrenamiento Distribuido de Redes Neuronales Profundas en Plataformas Heterogéneas

Sergio Moreno-Álvarez[1] , Mercedes E. Paoletti[2] , Juan M. Haut[2] , Juan-Antonio Rico-Gallego[1] , Javier Plaza[2] y Juan-Carlos Díaz-Martín[2]

Resumen— ***Asynchronous stochastic gradient descent* es una técnica de optimización comúnmente utilizada en el entrenamiento distribuido de redes neuronales profundas. En distribuciones basadas en particionamiento de datos, se entrena una réplica del modelo en cada unidad de procesamiento de la plataforma, utilizando conjuntos de muestras denominados *mini-batches*. Este es un proceso iterativo en el que al final de cada *mini-batch*, las réplicas combinan los gradientes calculados para actualizar su copia local de los parámetros. Sin embargo, al utilizar asincronismo, las diferencias en el tiempo de entrenamiento por iteración entre réplicas provocan la aparición del *staleness*, esto es, las réplicas progresan a diferente velocidad y en el entrenamiento de cada réplica se utiliza una versión no actualizada de los parámetros. Un alto grado de *staleness* tiene un impacto negativo en la precisión del modelo resultante. Además, las plataformas de computación de alto rendimiento suelen ser heterogéneas, compuestas por CPUs y GPUs de diferentes capacidades, lo que agrava el problema de *staleness*. En este trabajo, se propone aplicar técnicas de equilibrio de carga computacional, bien conocidas en el campo de la Computación de Altas Prestaciones, al entrenamiento distribuido de modelos profundos. A cada réplica se asignará un número de *mini-batches* en proporción a su velocidad relativa. Los resultados experimentales obtenidos en una plataforma heterogénea muestran que, si bien la precisión se mantiene constante, el rendimiento del entrenamiento aumenta considerablemente, o desde otro punto de vista, en el mismo tiempo de entrenamiento, se alcanza una mayor precisión en las estimaciones del modelo. Discutimos las causas de tal incremento en el rendimiento y proponemos los próximos pasos para futuras investigaciones.**

Palabras clave— **Aprendizaje Profundo, Computación de Altas Prestaciones, Entrenamiento Distribuido, Plataformas Heterogéneas, Redes Neuronales**

I. Introducción

El aprendizaje profundo *(Deep Learning)* ha alcanzado niveles de precisión muy altos en áreas como la clasificación de imágenes [1], [2] y el reconocimiento de voz [3], [4]. Estas mejoras son posibles gracias a los avances en las técnicas de entrenamiento, las plataformas HPC y el acceso a grandes conjuntos de datos utilizados para entrenar estos modelos [5].

Los clusters de computación de alto rendimiento permiten acelerar el entrenamiento de estos modelos, que normalmente se basa en técnicas de optimización como Stochastic Gradient Descent (SGD) aplicado a conjuntos de muestras, que se denominan *mini-batches* [6]. Cuando se trata de redes profundas de gran tamaño y grandes conjuntos de muestras, las plataformas paralelas HPC se vuelven indispensables. La paralelización del proceso de entrenamiento y el despliegue en los recursos de la plataforma se realiza mediante dos esquemas principales, conocidos como *model parallelism* (particionamiento del modelo) y *data parallelism* (particionamiento de datos).

El particionamiento del modelo divide el modelo a entrenar, es decir, sus parámetros, entre los recursos computacionales disponibles. Cada proceso, conocido como *worker* o *learner*, entrena una parte del modelo utilizando el mismo *mini-batch* de muestras. Los procesos comunican los resultados intermedios utilizando diferentes estrategias, como por ejemplo, un *pipeline* entre las capas del modelo desplegado en los recursos de la plataforma [7]. Como el entrenamiento es un proceso fundamentalmente secuencial, este tipo de esquemas dificultan el uso eficiente de los recursos computacionales, y su rendimiento estaría limitado por la comunicación entre las diferentes partes del modelo. Este método se utiliza cuando el modelo es lo suficientemente grande como para que no se pueda alojar en memoria de una sola unidad de proceso.

El particionamiento de datos consiste en ejecutar réplicas del modelo completo en cada recurso de computo disponible. Por tanto, cada réplica contiene una copia local de los parámetros a aprender, y se entrena utilizando subconjuntos de datos disjuntos. En cada paso del entrenamiento, los valores resultantes (gradientes) deben comunicarse al resto de las réplicas para combinar los resultados y actualizar los parámetros.

En un entorno distribuido que utiliza particionamiento de datos, el método de optimización denominado *Stochastic Gradient Descent* (SGD) síncrono recorre varias veces el conjunto completo de muestras, en lo que se denominan *épocas*. En cada época, a cada réplica se asigna un subconjunto disjunto de muestras, que a su vez se divide en *mini-batches* para entrenar su propia copia local del modelo. Después de cada *mini-batch*, los gradientes se calculan utilizando una función de pérdida *(loss function)* con respecto a sus valores locales actuales. Finalmente, los procesos se coordinan para combinar sus gradientes locales y actualizar sus parámetros, comenzando

[1]Dpto. de Ingeniería de Sistemas Informáticos y Telemáticos, Universidad de Extremadura, e-mail: smoreno@unex.es y jarico@unex.es.

[2]Dpto. de Tecnología de los Computadores y las Comunicaciones, Universidad of Extremadura, e-mail: mpaoletti@unex.es, juanmariohaut@unex.es, jplaza@unex.es y juancarl@unex.es.

una nueva iteración. La actualización de los valores de los parámetros requiere comunicación entre procesos. Dos métodos comúnmente usados son la utilización de operaciones colectivas de reducción (del tipo `MPI_Allreduce`) [8], o la utilización de un servidor centralizado de parámetros (*parameter server*) [9]. En cualquier caso, la optimización mediante SGD síncrono es determinista con respecto a la actualización de los valores de los parámetros, pues impone puntos de sincronización entre procesos en el momento de combinar los gradientes. Como consecuencia, sin embargo, el tiempo de espera de los procesos en la sincronización tiene un impacto negativo en el rendimiento general.

La optimización ASGD relaja la consistencia de los parámetros al permitir que los procesos combinen gradientes de forma asíncrona. Este esquema desacopla el cómputo y la comunicación, lo que beneficia enormemente el rendimiento del proceso de entrenamiento. Sin embargo, también desacopla los valores de los parámetros en las réplicas. Como consecuencia, el entrenamiento de un *mini-batch* en una réplica puede utilizar una versión desactualizada de los parámetros. La diferencia entre versiones de un parámetro local usado para calcular los gradientes en una réplica y su valor real se conoce como *staleness*. El grado de *staleness* de un parámetro se puede cuantificar asignándole una marca de tiempo en cada cálculo del valor del gradiente. Algunos estudios empíricos ([9]) muestran que un grado de *staleness* bajo no penaliza la precisión del modelo. Mientras que otros trabajos ([10], [11]) proponen mecanismos para reducir el impacto del *staleness* en la precisión del entrenamiento de los modelos, por ejemplo, reforzando negativamente la velocidad de aprendizaje con respecto al valor medio del *staleness* de los parámetros en las réplicas.

Los métodos anteriores de reducción del impacto del *staleness* en la precisión de un modelo están estrechamente relacionados con el comportamiento del método SGD. Sin embargo, hay otro factor que influye en el proceso de aprendizaje de un modelo. Las plataformas HPC suelen ser heterogéneas, y las diferencias en las capacidades computaciones de los recursos asignados a las réplicas tienen impacto en el *staleness* de los parámetros.

En este trabajo estudiamos los efectos en la precisión y en el rendimiento del entrenamiento de redes profundas utilizando un esquema distribuido de particionamiento de datos en plataformas HPC heterogéneas. Nuestro objetivo es estudiar el impacto de la heterogeneidad de la plataforma en la precisión del modelo entrenado, al mismo tiempo que se mejora el rendimiento utilizando mecanismos de equilibrado de carga.

Abordamos el problema en dos pasos. Partiendo de una distribución de réplicas en la plataforma HPC heterogénea, primero, equilibramos la carga de trabajo (datos de entrenamiento) entre las réplicas en proporción a sus capacidades computacionales. Después, usamos el método de optimización ASGD y comunicación basada en operaciones colectivas de los gradientes que proporciona el framework *PyTorch* [12]. Como veremos, este enfoque establece como límite de *staleness* la máxima diferencia relativa entre las velocidades de cómputo de las réplicas. Además, el equilibrado de la carga de trabajo entre las réplicas mejora notablemente el rendimiento del entrenamiento con respecto al SGD síncrono. El efecto general que observamos en nuestros experimentos es que la precisión del modelo aumenta en el mismo número de épocas/*batches* con respecto a la precisión del modelo de entrenamiento en una carga de trabajo no equilibrada, es decir, suponiendo una distribución de carga de trabajo homogénea. Por lo tanto, desde el otro punto de vista, el rendimiento aumenta para alcanzar la misma precisión.

Las principales contribuciones de este trabajo son:

- Evaluar la precisión del entrenamiento de modelos distribuidos utilizando particionamiento de datos cuando las réplicas procesan diferente número de muestras.
- Aplicar técnicas de equilibrio de carga de trabajo (estáticas), comunes en HPC, para distribuir el entrenamiento de modelos profundos, con el objetivo de optimizar el uso de recursos y el tiempo de ejecución.
- Establecer un límite superior para el valor del *staleness* cuando las réplicas se ejecutan en una plataforma heterogénea utilizando una optimización SGD asíncrona.
- Evaluar el impacto de la heterogeneidad computacional de la plataforma en el entrenamiento distribuido de una red neuronal profunda.

El resto de este documento se estructura como sigue. En la Sección II se describe nuestra implementación, incluyendo la distribución de los procesos en el sistema y el procedimiento de entrenamiento del modelo. En la Sección III se detalla la evaluación de nuestro sistema y se presentan los resultados. La Sección IV discute el trabajo relacionado, y finalmente, la Sección V presenta nuestras conclusiones y describe el trabajo futuro.

II. Implementación

Esta sección detalla la implementación de nuestro desarrollo para el entrenamiento de un modelo distribuido con particionamiento de datos, que se evaluará en la sección III.

Desarrollamos un esquema de particionamiento de datos para el entrenamiento distribuido de un modelo en una plataforma HPC heterogénea utilizando el *framework* PyTorch. Asumimos una plataforma heterogénea dedicada, compuesta por un conjunto de nodos de cómputo con diferentes capacidades o velocidades. Actualmente, los nodos de cómputo en clusters heterogéneos comunes combinan diferentes CPUs y GPUs, que se comunican mediante redes de alto rendimiento. Un problema clave es determinar tales capacidades computacionales. Utilizamos la herramienta FuPerMod [13] para determinar de forma

empírica las velocidades de los nodos utilizados para entrenar el modelo.

En el proceso de entrenamiento, cada nodo de cómputo ejecutará una réplica del modelo con una copia local de sus parámetros. Para entrenar el modelo se utiliza un algoritmo de optimización SGD asíncrono. Cada iteración del procedimiento ASGD realiza la siguiente secuencia de tareas:

1. Cada réplica obtiene un conjunto de *batches* aleatorio de tamaño $|B|$ muestras del conjunto total de datos N. Los *batches* asignados entre réplicas en cada época son disjuntos.
2. Las réplicas entrenan el modelo utilizando sus *batches* y calculan sus gradientes basándose en la función de pérdida $l(b|w)$, que calcula el error de la muestra b usando los parámetros en la réplica w con respecto al valor real.
3. Después de calcular los gradientes $\nabla l(b|w)$, cada réplica usa un hilo que realiza una operación de comunicación colectiva para actualizar sus parámetros locales con los valores obtenidos de las otras réplicas.
4. Los parámetros actualizados se calculan utilizando los gradientes y una tasa de aprendizaje (denominada *learning rate*), y comienza una nueva iteración del proceso de entrenamiento.

La Figura 1 muestra el proceso de entrenamiento de un modelo con varias réplicas. A continuación, detallamos los pasos anteriores.

A. Distribución de la Carga de Trabajo

Nuestro enfoque se basa en realizar un equilibrado estático de la carga de trabajo entre los nodos de cómputo involucrados en el entrenamiento, por lo tanto, el primer problema que nos encontramos es determinar la velocidad de dichos nodos. FuPerMod es una herramienta comúnmente utilizada en la optimización del rendimiento de sistemas heterogéos HPC. Se encarga de determinar empíricamente las capacidades computacionales de los nodos involucrados. Para ello, ejecuta un *benchmark* proporcionado por el usuario en cada nodo. Este *benchmark* debe ser representativo de los cálculos realizados en el proceso de entrenamiento, con el fin de obtener mediciones significativas. En nuestro caso, como *benchmark*, utilizamos la función bien conocida GEMM, ejecutada en un rango de diferentes tamaños de problema x. Como salida, FuPerMod devuelve las velocidades de los P nodos (réplicas) en la plataforma, es decir, devuelve un conjunto de P funciones $S = \{s_1(x), s_2(x), \ldots, s_P(x)\}$, que varían a lo largo del rango del tamaño del problema x.

Un aspecto importante es que el conjunto de funciones de velocidad S que caracterizan la plataforma es independiente del proceso de entrenamiento, y se determina estáticamente en un paso previo. Después, las funciones de velocidad S junto con el tamaño específico del conjunto de datos de entrenamiento $|N|$ se utilizan como entradas para la utilidad FuPerMod *partitioner*, que calcula la cantidad de muestras que se deben asignar a cada réplica[1]. Como resultado, se asignan μ_i muestras a cada réplica i, con $\sum_{i=1}^{P} \mu_i = |N|$.

Una vez que se determina el particionamiento de la carga de trabajo, la distribución de dicha carga se realiza al comienzo de cada época. El conjunto de datos (*dataset*) N se divide en P subconjuntos disjuntos según el vector $M = \{\mu_1, \mu_2, \ldots, \mu_P\}$, y se asignan a las réplicas correspondientes. Además, la distribución del conjunto de datos entre réplicas se realiza de tal manera que cada réplica entrena su copia del modelo con todo conjunto de datos a lo largo de las épocas.

B. Entrenamiento y Cálculo de Gradientes

Una vez que el conjunto de datos se divide y se distribuye entre las réplicas en función del vector M, cada réplica entrena iterativamente su copia del modelo en *batches* de tamaño $|B|$. El tamaño de cada *batch* es constante $|B|$, mientras que el número de *batches* utilizados en cada iteración de entrenamiento en una réplica, varía en función de su velocidad relativa.

Nuestra implementación modifica ligeramente el cargador de datos de PyTorch de la siguiente manera. Partimos de la salida de FuPerMod *partitioner*, $M = \{\mu_1, \mu_2, \ldots, \mu_P\}$, siendo μ_i el número de muestras asignadas a la réplica i, y construimos un vector $R = \{r_1, r_2, \ldots, r_P\}$, con $r_i = \lfloor \frac{\mu_i}{min_j \mu_j} \rfloor$, es decir, r_i es el número de *batches* de tamaño $|B|$ que cada réplica podrá entrenar en una iteración en relación con la réplica más lenta.

Para equilibrar la carga de trabajo, las réplicas calculan los gradientes en cada iteración de la siguiente manera:

$$g_i = \frac{1}{r_i \cdot |B|} \sum_{b \in B^\star} \nabla l(b|w_i), \qquad (1)$$

siendo $B^\star$ el número de muestras en r_i *batches* de tamaño $|B|$, y $l(b|w_i)$ la función de pérdida de una muestra b calculada usando los valores de parámetros actuales w_i en la réplica i.

Efectivamente, esto es equivalente a tener tamaños de *batch* desiguales de $r_i \cdot |B|$ en cada réplica ([14]), sin embargo, nuestro método de equilibrado de carga usando diferente cantidad de *batches* se puede implementar como un módulo independiente sin necesidad de modificar el cargador de datos *(dataloader)* de PyTorch. Como principal inconveniente, este método establece una granularidad en el equilibrado de carga dependiente de $|B|$, que, sin embargo, en nuestros experimentos tiene una influencia limitada en el rendimiento.

Finalmente, equilibrar la carga de trabajo modificando el número de *batches* que va a procesar cada réplica en una iteración, garantiza que todas las réplicas terminarán una época al mismo tiempo, con un

[1] Mientras que FuPerMod *partitioner* funciona originalmente con tamaños de problema en bytes, convertimos los datos resultantes en números de muestras, considerando muestras del mismo tamaño.

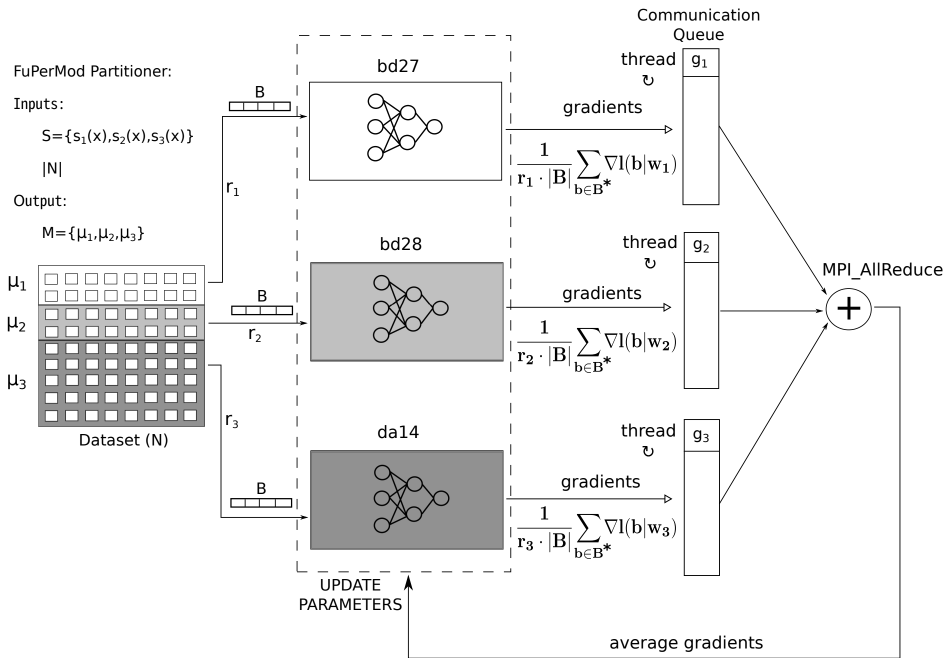


Fig. 1. Estructura de la implementación de un modelo de aprendizaje distribuido utilizando mecanismos de equilibrado de carga y el framework PyTorch. La figura representa una época en el proceso de entrenamiento para $P = 3$ réplicas, con un número desigual de muestras asignadas ($\mu_i, i = 1, \ldots, P$). En cada iteración del entrenamiento, cada réplica utiliza diferente número de *batches* del mismo tamaño ($|B|$) para entrenar su copia del modelo. Para combinar los gradientes resultantes, las réplicas lanzan hilos asíncronos encargados de la comunicación, utilizando la operación colectiva *MPI_Allreduce*.

error que viene determinado por (1) el error en la medición de la velocidad de los nodos con FuPerMod, y (2) la granularidad del equilibrado dependiente de $|B|$, ambos generalmente despreciables.

C. Combinación de Gradientes y Actualización de Parámetros

Existen dos métodos comunes de combinación de gradientes para actualizar la copia de los parámetros del modelo en todas las réplicas. El primero utiliza un servidor de parámetros *(parameter server)* que contiene la copia global actualizada de los parámetros y se encarga de actualizarlos en función de los gradientes recibidos de cada réplica. El servidor de parámetros permite la recepción asíncrona de gradientes. El principal inconveniente de este enfoque es su naturaleza centralizada, que se mitiga manteniendo varios procesos servidores [9]. El segundo enfoque se implementa en el marco PyTorch y consiste en actualizar los parámetros mediante el uso de comunicación colectiva MPI [15]. Cuando una réplica finaliza el cálculo de los gradientes, invoca una operación colectiva de reducción (`MPI _Allreduce`) para combinar los vectores de gradientes de cada réplica y actualizar su copia local de los parámetros. Dicha operación colectiva se encuentra definida en el estándar MPI y es bloqueante, por lo tanto, debido a la necesidad de sincronización tiene un impacto significativo en el rendimiento. La solución asíncrona implementada en PyTorch es utilizar un hilo independiente que se bloquea en la operación colectiva. Antes de la siguiente iteración del entrenamiento, una réplica comprueba si la operación colectiva ha terminado. En caso positivo, actualiza los parámetros locales, y en otro caso realiza una nueva iteración de entrenamiento utilizando la copia actual (por tanto desactualizada) de los parámetros.

En nuestra implementación, una réplica comunica sus gradientes g_i después de procesar $r_i \cdot |B|$ número de muestras, es decir, lanza una operación colectiva después de entrenar r_i *batches*. Este método limita el valor de *staleness*, que dependerá de la heterogeneidad de la plataforma, y tendrá un límite superior de $max_i r_i$. Como conclusión, en el proceso de optimización SGD asíncrono utilizando equilibrado de carga en función de las capacidades computacionales de cada réplica, se reducen los tiempos de espera en la comunicación, mientras se limita el *staleness*, lo que resulta en una mejora general del rendimiento.

III. Experimentación y Evaluación

Esta sección evalúa la implementación propuesta en una pequeña plataforma de prueba. Aunque pequeña, la plataforma sirve como prueba de concepto para obtener resultados iniciales del comportamiento de dicha implementación. Primero, introducimos los elementos de *hardware* y *software* utilizados, y posteriormente discutimos los resultados del entrenamiento distribuido con respecto a la precisión del modelo y el rendimiento del proceso de aprendizaje.

A. Plataforma Experimental

La plataforma heterogénea está compuesta por $P = 3$ nodos multinúcleo conectados por una red Infiniband QDR (4x), en el cluster *Fermi* del centro de cómputo CETA-Ciemat. En este trabajo inicial, elegimos nodos de cómputo tipo CPU con diferente número de núcleos y, por lo tanto, con un rendimiento relativo diferente como se muestra en la Tabla I. Se ejecuta una réplica del modelo en cada nodo de

TABLA I
Plataforma heterogénea compuesta de tres nodos (réplicas) con diferentes capacidades computacionales, junto con la salida del número de muestras obtenidas con FuPerMod para un problema total de $|N| = 60.000$, y las velocidades relativas de los nodos de cómputo para el tamaño del conjunto de datos.

Nombre del nodo	MPI Rank	Número de Cores	Tipo de Recurso	Número de Muestras (μ)	Velocidad Relativa
bd27	0	12	CPUs	14.028	1.01
bd28	1	12	CPUs	13.889	1.00
da14	2	24	CPUs	32.083	2.31

cómputo. La columna *número de muestras* contiene el resultado devuelto por FuPerMod *partitioner* para un conjunto de datos de tamaño $|N| = 60.000$ y un vector S que se calcula previamente y que contiene la funciones que determinan la velocidad de las réplicas ejecutando un *benchmark* GEMM para diferentes tamaños de datos.

Los tipos de CPU en los nodos *bd27* y *bd28* son procesadores Intel Westmere de 12 núcleos a 2.53 GHz con 24 GB de RAM, mientras que el nodo nodo *da14* es de tipo Intel Haswell con 24 núcleos a 2.50 GHz y 64 GB de RAM. Los nodos *bd* muestran una ligera diferencia en su velocidad. Esto se puede deber a dos motivos, errores de medición de al determinar la velocidad $s_i(x)$ (que consideramos despreciables) y pequeñas diferencias en el rendimiento del *hardware* y *software* que se ejecuta en los nodos.

En trabajos futuros, planeamos utilizar GPUs, y también una combinación de ambos tipos de procesadores, para explotar la capacidad de cómputo total de la plataforma en el entrenamiento.

B. Descripción del Conjunto de Datos

Utilizamos MNIST [16] como el conjunto de datos para probar nuestra implementación. Este conjunto de datos está compuesto por imágenes en escala de grises que representan dígitos de 0 a 9 escritos a mano. El tamaño del conjunto de entrenamiento es de 60.000 muestras, e incluye un conjunto de verificación de 10.000 muestras, utilizado para calcular la precisión del modelo. Los dígitos se han normalizado y centrado en una imagen de tamaño fijo de $28 \times 28 \times 1$, en la que cada pixel se representa mediante un float de 32 bits. En la Figura 2 se pueden observar algunos ejemplos de las imágenes que componen el conjunto de datos.

C. Resultados

Las réplicas ejecutan el proceso de entrenamiento en su propia copia local del modelo. En particular, el modelo es una red neuronal convolucional (CNN) [17] compuesta de dos partes principales: un extractor de características y un clasificador. Los detalles de la red se muestran en la Tabla II.

Con respecto a la primera parte, se compone de dos etapas de capas convolucionales y de agrupación *(pooling)*, mientras que la segunda parte se compone

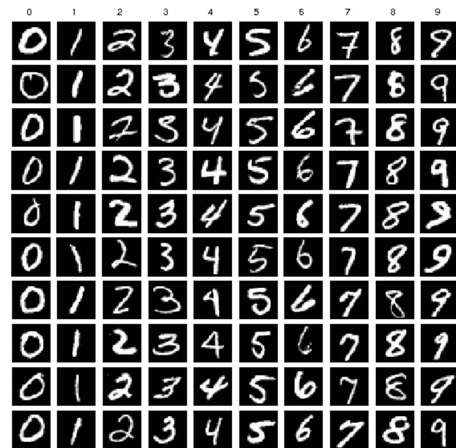


Fig. 2. Muestra de algunas imágenes de digitos escritos a mano en el conjunto MNIST.

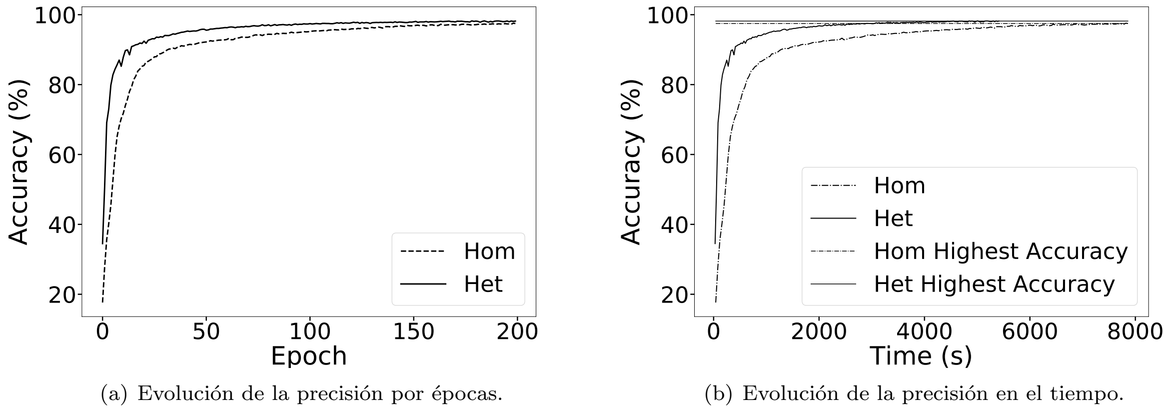
TABLA II
Capas de la red neuronal convolucional para la clasificación de imagenes en el conjunto de datos MNIST

ID Capa	*Kernel*/Neuronas	Función Act.	Pooling
Conv1	$20 \times 5 \times 5 \times 1$	ReLU	2×2
Conv2	$50 \times 5 \times 5 \times 20$	ReLU	2×2
FC1	500	ReLU	-
FC2	10	Softmax	-

de dos capas completamente conectadas (FC). Las capas convolucionales bidimensionales emplean como función de activación la Unidad Lineal Rectificada (ReLU, Rectified Linear Unit). Los mapas de características obtenidos en la parte convolucional se transforman en una representación vectorial para alimentar las capas de clasificación.

Definimos la *precisión* del modelo como porcentaje de acierto en la clasificación de las muestras, teniendo en cuenta que la distribución de las clases (digitos) en el conjunto de muestras no contiene sesgo. Por otro lado, medimos el *rendimiento* como el tiempo empleado en el proceso de entrenamiento. Los resultados globales se obtienen tomando la máxima precisión de los modelos y el tiempo máximo de entrenamiento de las réplicas. Comparamos la precisión y el rendimiento del proceso de entrenamiento en la plataforma heterogénea descrita en la sección III-A siguiendo las distribuciones de equilibrado de carga de trabajo tanto homogéneas (todas las réplicas reciben la misma carga) como heterogéneas (cada réplica recibe un número de muestras proporcional a sus capacidades).

La Figura 3 muestra la comparación en la precisión del modelo considerando una distribución homogénea de muestras (*Hom*) entre réplicas, y con equilibrado de carga de acuerdo con sus velocidades (*Het*). El tamaño del *batch* B/P se estable arbitrariamente con $B = 6400$. Hemos evaluado otros tamaños obteniendo resultados muy similares. En la distribución homogénea *Hom*, se usa un solo *batch* por iteración para entrenar el modelo ($r_i = 1, \forall_i$), para más tarde lanzar un hilo para combinar gradientes y ac-



(a) Evolución de la precisión por épocas.

(b) Evolución de la precisión en el tiempo.

Fig. 3. Resultados de precisión obtenidos para el proceso de entrenamiento. *Hom* representa los resultados en la precisión del modelo con un número de *batches* homogéneo. Por el contrario, *Het* representa los resultados en la precisión cuando las réplicas en cada iteración utilizan un número de *batches* proporcional a sus velocidades.

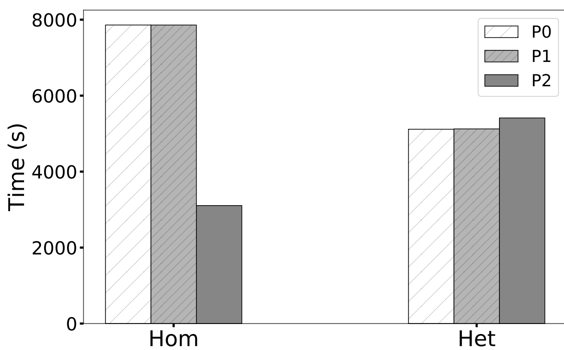


Fig. 4. Tiempo obtenido en 200 épocas de entrenamiento del modelo en las réplicas para ambos métodos de particionamiento de carga: homogéneo y heterogéneo.

tualizar parámetros. En la distribución heterogénea *Het*, el número de *batches* utilizados para entrenar cada réplica cambia, y es proporcional a las velocidades relativas obtenidas utilizando la herramienta *FuPerMod*. Los valores resultantes por iteración son $R = \{1, 1, 2\}$, derivados de la Tabla I. La Figura 3(a) muestra los diferentes valores de precisión para *Hom* y *Het* a lo largo de 200 épocas. La precisión de la distribución homogénea alcanza 96.49 %, mientras que la heterogénea alcanza 98.17 %. Aunque ambos valores de precisión tienden a un mismo valor máximo de aproximadamente 98.65 % en un número mayor de épocas, la gráfica muestra cómo el esquema homogéneo converge en mayor número de épocas. Esto se debe a que el grado de *staleness* causado por las réplicas más lentas repercute negativamente en la precisión, debido a que provocan inestabilidad en la convergencia, como se describe en [18]. La Figura 3(b) muestra cómo la distribución heterogénea *Het* converge más rápido a su valor de máxima precisión en las 200 épocas.

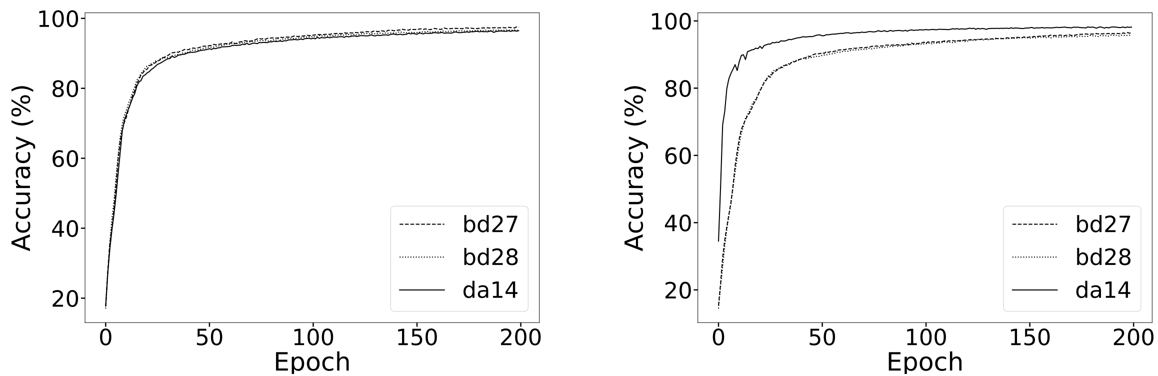
La Figura 4 muestra los detalles de rendimiento de las réplicas después de 200 épocas. En la distribución homogénea nos encontramos con una gran diferencia de tiempo en función de la velocidad de los nodos. Este escenario mejora utilizando un equilibrado de la carga de trabajo en la distribución heterogénea *Het*. Existe una ligera diferencia en los tiempos de *Het* entre nodos, causado por el tamaño de $|B|$, que establece la granularidad en la distribución (ver sección II-B).

La Figura 5 muestra la precisión a lo largo de las épocas para las réplicas involucradas en el entrenamiento de las distribuciones *Hom* y *Het*. Todas las réplicas en la distribución homogénea tienen una precisión similar a lo largo de las épocas, como se muestra en la Figura 5(a), debido a que procesan el mismo tamaño de *batch* con diferentes rendimientos. Sin embargo, en la Figura 5(b) se muestra cómo en una distribución heterogénea, el nodo más rápido alcanza una mayor precisión, porque entrena su modelo con un mayor número de muestras en cada iteración. Por otro lado, las réplicas combinan gradientes con mayor frecuencia y, por lo tanto, tienen un grado de *staleness* más bajo en los parámetros, lo que mejora la precisión y el rendimiento general.

IV. Trabajo Relacionado

El crecimiento en el tamaño de los conjuntos de datos y la cantidad de parámetros en modelos profundos han impulsado el uso de las plataformas HPC para acelerar el entrenamiento. El trabajo [19] proporciona un excelente estudio de las técnicas distribuidas que se utilizan actualmente para paralelizar y distribuir el entrenamiento.

Los principales esquemas de paralelismo, tanto de datos como de modelos, se analizan en el trabajo [20], que propone un entrenamiento distribuido de redes convolucionales utilizando paralelismo de datos en capas convolucionales y paralelismo de modelos en capas completamente conectadas, además de diferentes métodos de sincronización para la actualización de parámetros entre trabajadores. En [9] se propone un algoritmo SGD asíncrono *Downpour* implementado en el *framework DistBelief*. Este algoritmo permite el entrenamiento paralelo de modelos de datos de gran escala utilizando un servidor de parámetros centralizado y *workers* asíncronos. Los autores deter-



(a) Evolución de la precisión en la distribución homogénea. (b) Evolución de la precisión en la distribución heterogénea.

Fig. 5. Resultados obtenidos en términos de precisión de cada uno de las réplicas para las distribuciones homogénea y heterogénea.

minaron que un nivel de tolerancia de *staleness* no afecta significativamente a la precisión del modelo.

Por el contrario, los trabajos [10], [11] proponen un mecanismo para reducir el *staleness* modificando la tasa de aprendizaje utilizando los valores actuales del *staleness* promedio de los gradientes. Proporcionan una discusión sobre la interacción de los hiperparámetros de entrenamiento y las opciones de distribución, utilizando el *framework Rudra*.

El problema del *staleness* también se aborda en [21] con un enfoque diferente. El trabajo propone un entrenamiento basado en un particionamiento de datos entre p réplicas de respaldo, además de las P réplicas principales, utilizando optimización SGD síncrona. Para actualizar los parámetros del modelo, considera las P réplicas más rápidas en el cálculo de los gradientes y descarta el resto. Este enfoque reduce el *staleness* y los tiempos de espera por las réplicas más lentas, sin embargo, el consumo de recursos es mayor.

En cuanto a plataformas heterogéneas, el trabajo [18] estudia la degradación del rendimiento de la optimización SGD en sistemas heterogéneos con respecto a los esquemas de entrenamiento distribuidos homogéneos. Se centra en los sistemas *Stale-Synchronous Parallel*, en los que el protocolo de actualización de parámetros y el servidor de parámetros limitan el grado de *staleness* del sistema. Los autores proponen aplicar tasas de aprendizaje tanto constantes como dinámicas para reducir la inestabilidad en la convergencia del modelo, causada por las réplicas atrasadas, mejorando la precisión y el rendimiento.

Con respecto a la modificación del tamaño del *batch*, *AdaBatch* [22] adapta el tamaño del *batch* a lo largo del proceso de entrenamiento junto con la tasa de aprendizaje, de modo que su relación se mantenga constante, mejorando el rendimiento para *batches* grandes y la precisión para los *batches* pequeños. Sin embargo, este enfoque se aplica en plataformas homogéneas, y no tiene como objetivo contrarrestar la heterogeneidad de la plataforma.

El trabajo [14] propone adaptar los tamaños de los *batches* a las velocidades en cada réplica en una plataforma heterogénea para minimizar los tiempos de espera. A diferencia de nuestra propuesta, este trabajo utiliza un esquema de paralelización *Bulk Synchronous Parallel* en el entrenamiento, esto es, optimización SGD síncrona. La fuente de heterogeneidad (simulada en sus experimentos) proviene del uso no dedicado de los recursos en plataformas *cloud*. El uso del método de optimización SGD síncrono evita el *staleness*. La medición de la velocidad de las réplicas, necesaria para calcular el tamaño de sus *batches* asignados, se logra utilizando una *Recurrent Neural Network* por trabajador, entrenada con valores del uso de memoria y CPU en cada iteración. Vale la pena señalar que nuestro trabajo utiliza SGD asíncrono y explora el impacto del *staleness* en la precisión y el rendimiento en clusters heterogéneos, sin embargo, el trabajo [14] tiene ideas adicionales que planeamos incluir en trabajos futuros, como agregación ponderada de los gradientes en función del tamaño del *batch* para evitar sesgos por muestra en las réplicas, y la evaluación de la implementación en un mayor número de recursos computacionales.

V. Conclusiones y Trabajo Futuro

Este trabajo realiza un estudio preliminar en el contexto del entrenamiento distribuido de redes profundas en plataformas heterogéneas. Proponemos un particionamiento de datos estático, previo al proceso de entrenamiento que ejecutan las réplicas en los recursos de cómputo de la plataforma. Este particionamiento de datos utiliza herramientas tradicionalmente usadas en optimización de aplicaciones HPC como FuPerMod, y asigna a cada réplica un número de muestras proporcional a su velocidad, que se determina con anterioridad. Utilizado junto con un algoritmo de optimización Stochastic Gradient Descent asíncrono, el equilibrado de carga establece un límite superior para el grado de *staleness* entre réplicas, a la vez que reduce los tiempos de espera en los puntos de sincronización y comunicación al final de cada iteración en el entrenamiento. El *staleness* determina el grado en el que una réplica mantiene

parámetros obsoletos o no actualizados para entrenar su copia del modelo. Otros trabajos anteriores analizan cómo el *staleness* afecta negativamente a la precisión del modelo resultante. En nuestra propuesta, dicho grado de *staleness* tiene como límite la máxima diferencia en la velocidad relativa de los recursos heterogéneos asignados a cada réplica, y por tanto, depende de la plataforma.

La implementación de la carga y distribución de datos se realiza a través del *framework* PyTorch, y aprovecha sus capacidades internas basadas en comunicación colectiva MPI para la comunicación asíncrona de los gradientes con el fin de actualizar los parámetros.

Los resultados experimentales, en una plataforma dedicada HPC heterogénea, muestran una mejora en el tiempo de entrenamiento del equilibrado de carga con respecto al método homogéneo, en el que a cada réplica se le asigna un número de muestras uniforme. Desde otro punto de vista, los mecanismos de equilibrado de carga heterogéneos logran una mayor precisión en el mismo tiempo que el mecanismo homogéneo.

Nuestro trabajo futuro se centra en la evaluación de la escalabilidad de la implementación propuesta en dos vertientes. La primera relacionado con el *hardware* y la utilización de un mayor número de nodos heterogéneos de cómputo, incluido el uso de GPUs. La segunda vertiente se relaciona con el *software* y la utilización de un conjunto de datos mayor y más complejo, y una red neuronal convolucional más profunda.

Agradecimientos

Este trabajo ha sido apoyado conjuntamente por los siguientes proyectos e instituciones:

- Ministerio de Educación (Resolución de 26 de diciembre de 2014 y 19 de noviembre de 2015 de la Secretaría de Estado de Educación, Formación Profesional y Universidades, mediante la cual se solicita información para la capacitación de profesores universitarios, de los subprogramas de Capacitación y Movilidad incluidos en el Programa Estatal para las Promociones de Talento y su Empleabilidad, en el marco del Plan Estatal de Investigación e Innovación Científica y Técnica de 2013-2016.
- Por el Fondo Europeo de Desarrollo Regional 'Una manera de hacer Europa'(FEDER) y el gobierno local de Extremadura (Ref. IB16118).
- Por la Administración Local de Extremadura (Ref. 297/2014, ayuda para llevar a cabo actividades de investigación y desarrollo tecnológico y para el desarrollo, la difusión y la transferencia de conocimientos para los grupos de investigación de Extremadura, Ref. GR15005).
- Por el Proyecto MINECO TIN2015-63646-C5-5-R a su vez con el Fondo Europeo de Desarrollo Regional 'Una manera de hacer Europa' (FEDER).
- Por las instalaciones informáticas del Centro de Investigación de Tecnologías Avanzadas de Extremadura (CETA-CIEMAT), financiado por el Fondo Europeo de Desarrollo Regional (FEDER).

Referencias

[1] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds., pp. 1097–1105. Curran Associates, Inc., 2012.

[2] Gao Huang, Zhuang Liu, and Kilian Q. Weinberger, "Densely connected convolutional networks," *CoRR*, vol. abs/1608.06993, 2016.

[3] Dong Yu and Li Deng, *Automatic Speech Recognition: A Deep Learning Approach*, Signals and Communication Technology. Springer, London, 2015.

[4] Dario Amodei, Sundaram Ananthanarayanan, Rishita Anubhai, and more., "Deep speech 2: End-to-end speech recognition in english and mandarin," in *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*. 2016, ICML'16, pp. 173–182, JMLR.org.

[5] Geoffrey Fox, Judy Qiu, Shantenu Jha, Saliya Ekanayake, and Supun Kamburugamuve, "Big data, simulations and hpc convergence," in *Big Data Benchmarking*, Tilmann Rabl, Raghunath Nambiar, Chaitanya Baru, Milind Bhandarkar, Meikel Poess, and Saumyadipta Pyne, Eds., Cham, 2016, pp. 3–17, Springer International Publishing.

[6] Quoc V. Le, Jiquan Ngiam, Adam Coates, Abhik Lahiri, Bobby Prochnow, and Andrew Y. Ng, "On optimization methods for deep learning," in *Proceedings of the 28th International Conference on International Conference on Machine Learning*, USA, 2011, ICML'11, pp. 265–272, Omnipress.

[7] Yanping Huang, Yonglong Cheng, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, and Zhongqian Chen, "Gpipe: Efficient training of giant neural networks using pipeline parallelism," *CoRR*, vol. abs/1811.06965, 2018.

[8] Alexander Sergeev and Mike Del Balso, "Horovod: fast and easy distributed deep learning in tensorflow," *CoRR*, vol. abs/1802.05799, 2018.

[9] Jeffrey Dean, Greg S. Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, Marc'Aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, and Andrew Y. Ng, "Large scale distributed deep networks," in *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, USA, 2012, NIPS'12, pp. 1223–1231, Curran Associates Inc.

[10] Suyog Gupta, Wei Zhang, and Fei Wang, "Model accuracy and runtime tradeoff in distributed deep learning: A systematic study," in *Proceedings of the 26th International Joint Conference on Artificial Intelligence*. 2017, IJCAI'17, pp. 4854–4858, AAAI Press.

[11] Wei Zhang, Suyog Gupta, Xiangru Lian, and Ji Liu, "Staleness-aware async-sgd for distributed deep learning," in *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*. 2016, IJCAI'16, pp. 2350–2356, AAAI Press.

[12] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer, "Automatic differentiation in pytorch," in *NIPS-W*, 2017.

[13] David Clarke, Ziming Zhong, Vladimir Rychkov, and Alexey Lastovetsky, "Fupermod: A framework for optimal data partitioning for parallel scientific applications on dedicated heterogeneous hpc platforms," in *Parallel Computing Technologies*, Victor Malyshkin, Ed., Berlin, Heidelberg, 2013, pp. 182–196, Springer Berlin Heidelberg.

[14] Chen Chen, Qizhen Weng, Wei Wang, Baochun Li, and Bo Li, "Fast distributed deep learning via worker-adaptive batch sizing," in *Proceedings of the ACM Symposium on Cloud Computing*, New York, NY, USA, 2018, SoCC '18, pp. 521–521, ACM.

[15] Richard L. Graham, Timothy S. Woodall, and Jeffrey M. Squyres, "Open mpi: A flexible high performance mpi," in *Parallel Processing and Applied Mathematics*, Roman Wyrzykowski, Jack Dongarra, Norbert Meyer, and Jerzy

Waśniewski, Eds., Berlin, Heidelberg, 2006, pp. 228–239, Springer Berlin Heidelberg.

[16] Yann LeCun, Corinna Cortes, and Christopher JC Burges, “The mnist database of handwritten digits, 1998,” *URL http://yann. lecun. com/exdb/mnist*, vol. 10, pp. 34, 1998.

[17] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton, “Imagenet classification with deep convolutional neural networks,” in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[18] Jiawei Jiang, Bin Cui, Ce Zhang, and Lele Yu, “Heterogeneity-aware distributed parameter servers,” in *Proceedings of the 2017 ACM International Conference on Management of Data*, New York, NY, USA, 2017, SIGMOD ’17, pp. 463–478, ACM.

[19] Tal Ben-Nun and Torsten Hoefler, “Demystifying parallel and distributed deep learning: An in-depth concurrency analysis,” *CoRR*, vol. abs/1802.09941, 2018.

[20] Alex Krizhevsky, “One weird trick for parallelizing convolutional neural networks,” *CoRR*, vol. abs/1404.5997, 2014.

[21] Jianmin Chen, Rajat Monga, Samy Bengio, and Rafal Jozefowicz, “Revisiting distributed synchronous sgd,” in *International Conference on Learning Representations Workshop Track*, 2016.

[22] Aditya Devarakonda, Maxim Naumov, and Michael Garland, “Adabatch: Adaptive batch sizes for training deep neural networks,” *CoRR*, vol. abs/1712.02029, 2017.