Estrategias para la mejora de gestión de memoria en algoritmos de aprendizaje profundo para el realce de contraste de imágenes radiológicas de alta resolución

Daniel A. Rodríguez López[1 2], Daniel Sanderson[1 2] , Javier García Blas[2 3], Manuel Desco[1 2 4 5] y Mónica Abella[1 2 5]

Resumen— **La radiografía convencional es una valiosa herramienta para el diagnóstico, gracias a su asequibilidad, facilidad de interpretación y manejo. A pesar de estas ventajas, tiene limitaciones debido a la mínima diferencia en la absorción de rayos X entre los tejidos blandos y al contraste reducido causado por la dispersión de los rayos X. La aplicación de técnicas de procesamiento de imágenes puede mejorar la calidad de la radiografía y al mismo tiempo disminuir la exposición del paciente a la radiación ionizante. Para ello, en los últimos años, se ha comenzado a aplicar técnicas de procesamiento de imagen basadas en aprendizaje profundo. El empleo de estas técnicas requiere de una gran cantidad de recursos computacionales, especialmente en términos de memoria en los dispositivos GPU.**

Este trabajo presenta un análisis detallado de distintas técnicas de reducción del consumo de memoria durante las etapas de entrenamiento. Esto permite trabajar con imágenes de gran tamaño, como es el caso de la imagen radiológica, evitando los problemas derivados de trabajar con parches o la pérdida de resolución espacial que supone hacer un submuestreo. Para ello se han estudiado de forma experimental distintas aproximaciones, tanto en ejecuciones en un solo computador como en sistemas distribuidos basados en paralelismo de modelo. La evaluación experimental demuestra que es posible alcanzar hasta una reducción del 20 % para un único nodo y hasta 70 % para el modelo distribuido sin pérdida de precisión.

Palabras clave— **Paralelismo de modelo, aprendizaje distribuido, imagen médica, U-Net, aprendizaje profundo.**

I. Introducción

La radiología es uno de los métodos de imagen más utilizados por su facilidad de uso y bajo coste [1], [2]. En los últimos años, el uso de técnicas de aprendizaje profundo para el procesamiento y análisis de imágenes en el ámbito médico se ha popularizado igualmente [3]. Para poder visualizar las imágenes generadas en los distintos procesos médicos lo más cercano a su valor original es necesario utilizar resoluciones muy altas. Esto puede ser un impedimento para la mayoría de redes de neuronas, ya que por lo general se trabaja con tamaños de imagen pequeños, como es el caso de U-Net[4] con 512x512 píxeles, siendo más común utilizar resoluciones inferiores, tal y como ocurre en redes como ResNet[5] o VGG[6], ambas con tamaño de entrada de 224x224 píxeles. Esta reducción de resolución no suele ser un problema en la mayoría de casos. Sin embargo, en imágenes médicas puede significar la pérdida de los píxeles necesarios para identificar problemas de salud de forma prematura. Debido a esto, a más resolución se pueda trabajar, más fiables podrán ser los resultados obtenidos.

Hoy en día existen distintas aproximaciones para reducir el consumo de memoria en entornos distribuidos. No obstante, la mayoría de estas técnicas están enfocadas a modelos de lenguaje, donde el número de parámetros de la propia red pueden llegar a ocupar decenas de GBs. Ejemplos son GPT-2[7] con 1,5 mil millones de parámetros, o T5[8], llegando hasta 11 mil millones de parámetros. Estas reducciones vienen a través de un uso más eficiente de los parámetros y las comunicaciones [9]. En el ámbito médico suelen solventar estos problemas desarrollando nuevo métodos de entrenamiento [10].

Como se relata en [11], a pesar de ello, esta técnica ofrece poco contraste en tejido blando. Existen algoritmos de procesamiento convencional para realizar un realce de contraste sobre las mismas y solventar parcialmente estos inconvenientes (ver figura 1).

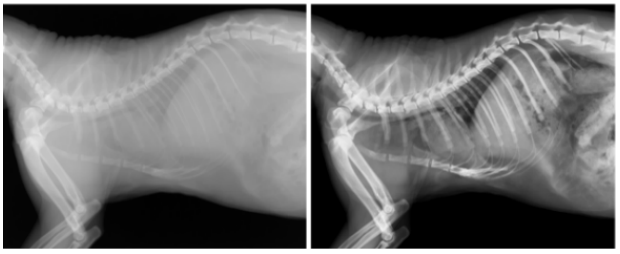


Fig. 1: Radiografía de tórax de gato en datos crudos (izquierda) y procesada mediante algoritmos convencionales (derecha).

Los algoritmos tradicionales suelen depender de un alto número de parámetros y requieren ajustes manuales dependiendo del caso de uso [12]. Por ello, se han empezado a desarrollar alternativas basadas en aprendizaje profundo para eliminar esta selección de parámetros y permitir un procesamiento automático

[1]Departamento de Bioingeniería, Universidad Carlos III de Madrid, Leganés, España, e-mail: mabella@ing.uc3m.es

[2]Laboratorio de Imagen Médica, Instituto de Investigación Sanitaria Gregorio Marañón, Madrid, España

[3]Departamento de Informática, Universidad Carlos III de Madrid, Leganés, España

[4]Centro de Investigación Biomédica en Red de Salud Mental (CIBERSAM), Madrid, España

[5]Centro Nacional de Investigaciones Cardiovasculares Carlos III (CNIC), Madrid, España

[13], [14].

Este trabajo utiliza el modelo desarrollado previamente en [11], el cual utiliza una red U-Net [4] basada en ResNet-34, usando el Índice de Similitud Estructura Multiescala (MS-SSIM) [15] como función de coste. U-Net es una red que se caracteriza por dividirse en dos partes bien diferenciadas: *Encoder*, que reduce y codifica la imagen, y *Decoder*, que se encarga de recuperar el tamaño original.

La resolución original de las imágenes utilizadas en este proceso pueden llegar hasta los 4188x4188 píxeles. Sin embargo, por las limitaciones de memoria que presentan estos algoritmos, el tamaño máximo con el que se puede entrenar en una GPU de 8GB es 1440x1440 píxeles.

En este trabajo se evalúan y comparan las distintas técnicas disponibles para reducir el consumo de memoria en GPU con el objetivo de poder incrementar el tamaño de las imágenes de entrenamiento sin perjudicar a la calidad de los resultados. La combinación de técnicas permite llegar a reducir hasta en un 20 % el consumo con una única GPU y hasta un 70 % en entornos distribuidos, con resultados comparables a la ejecución original.

El resto del documento se divide en las siguientes secciones. En la sección II se introducen las técnicas evaluadas y aplicables para un entrenamiento en un solo nodo y distribuido del modelo. En la sección III se detalla la metodología empleada para llevar a cabo los experimentos de este trabajo. A continuación, la sección IV presenta nuestra discusión sobre los resultados obtenidos para las distintas técnicas. Finalmente, la sección V resumen las conclusiones obtenidas en base al trabajo realizado y los resultados obtenidos.

II. Técnicas

En este apartado se presentan y describen brevemente las técnicas evaluadas para optimizar el consumo de memoria durante el entrenamiento de una red de neuronas.

A. Técnicas secuenciales

Las técnicas secuenciales buscan reducir el consumo de memoria en una única GPU, ya sea retrasando cálculos, descartando valores intermedios o cambiando la algoritmia de la propia red.

Gradient checkpointing [16], [17]: técnica que reduce el consumo de memoria a cambio de tiempo de cómputo. Para ello, los cálculos intermedios que se guardan durante la etapa de *forward* son descartados y calculados desde 0 durante el *back propagation*.

Automatic Mixed Precision (AMP) [18] se basa en determinar los cálculos durante el entrenamiento que no requieren operar con números en coma flotante de simple precisión. En estos casos la precisión se puede reducir hasta los 16 bits sin pérdidas de calidad. Hay tres posibles aproximaciones:

- *float16*: 1 bit de signo, 10 bits de mantisa y 5 de exponente.
- *bfloat16*: Desarrollado por Google[19]. 1 bit de signo, 7 bits de mantisa y 8 de exponente.
- *TensorFloat32*: Desarrollado por Nvidia[20] y disponible a en sus tarjetas gráficas a partir de la serie 3000. 1 bit de signo, 10 bits de mantisa y 8 de exponente.

Quantization[21], [22] corresponde con un conjunto de técnicas que buscan reducir el consumo de memoria y recursos de la red reduciendo la precisión global (similar a AMP) y combinando operaciones para reducir el consumo de memoria intermedio. Esta reducción de precisión tiene un impacto negativo en la precisión.

ResNet invertible[23], [24], [25] es una técnica específica para redes residuales (ResNet). Se basa en poder reconstruir las activaciones de una capa solamente con información de la capa anterior. De esta manera sólo es necesario almacenar las últimas activaciones.

B. Técnicas distribuidas

Las técnicas distribuidas buscan repartir la carga de trabajo entre distintos nodos de cómputo. En el caso de redes de neuronas, repartir el entrenamiento entre varias GPUs que pueden estar en un mismo nodo o repartidas entre varias máquinas interconectadas.

Paralelismo de datos (PD): técnica clásica de paralelismo donde el modelo es replicado íntegramente en distintas GPUs y se distribuyen los datos de entrenamiento. En este tipo de paralelismo hay que sincronizar las actualizaciones de parámetros de la red para evitar incongruencias. Pensado principalmente para acelerar el entrenamiento de la red, no reducir el consumo de memoria.

Paralelismo de modelo (PM): a diferencia del paralelismo de datos, los datos se introducen a la red secuencialmente, pero el modelo está repartido entre distintas GPUs. Hay que transmitir las salidas de la red entre dispositivos y sincronizar la actualización de parámetros de la red. Esta aproximación, aunque puede reducir considerablemente el uso de memoria en la GPU, no es automática (se están desarrollando aproximaciones automáticas [26], [27]) y suele requerir de modificaciones en la red y cambios en la función de entrenamiento, lo que hace que no sea tan trivial como realizar PD.

Fragmentar parámetros[28], [29]: mejora aplicable tanto en PD y PM, siendo en PD donde ocurre la mayor ganancia. Este algoritmo de optimización busca reducir la duplicidad de parámetros sobre el paralelismo de datos. Para ello, al realizar PD, los parámetros no son duplicados, si no, repartidos entre los distintos dispositivos.

Pipelines[30]: mejora aplicable al PM. Dado que la ingesta de datos es secuencial, la utilización de las GPUs decae en comparación con PD o ningún tipo de paralelismo. *Pipelines* buscan incrementar la tasa de utilización introduciendo los siguientes datos tan pronto como sea posible. Aumenta la ocupación de la GPU y, por tanto, reduce el tiempo de entrena-

miento.

III. Metodología

Para el entrenamiento de la red se han utilizado los datos procedentes de [11], 24 imágenes de gato, 34 imágenes de perro y 2 de tortuga, todas ellas adquiridas en el Hospital Clínico Veterinario de la Universidad Complutense de Madrid. La resolución de estas radiografías va desde 1288x950 hasta 4188x4188 píxeles. Las imágenes han sido normalizadas en un rango de intensidades entre 0 y 1. El conjunto de entrenamiento consta de 48 imágenes elegidas aleatoriamente y 12 imágenes para la validación, las cuales son 5 imágenes de perro, 6 de gato y 1 de tortuga. No se ha utilizado un conjunto de test. Las imágenes de referencia se han obtenido mediante algoritmos convencionales y validadas por un radiólogo experto.

Para la generación de los resultados se ha utilizado Python 3.10, junto con la librería Pytorch[31] versión 2.2. Para la implementación de la ResNet invertible se ha utilizado la librería MomentumNet[32]. Los resultados en un único nodo se han obtenido con una GPU Nvidia RTX 2060 con 8GB de RAM y driver 545.23.08. Para las pruebas distribuidas se han utilizado tres nodos, cada uno con una única GPU Nvidia 3070 con 8GB de RAM, driver 535.161.07 e interconectados mediante Ethernet de 10 Gbits/s.

Los consumos de memoria han sido realizados mediante el paquete *pynvml*, que ofrece acceso a *NVIDIA Management Library*, API en C de Nvidia para conocer el estado de la GPU. En ambos escenarios se han medido los mismos consumos: las fases de *forward* y *backward* durante el entrenamiento de la red de neuronas. Para ello se han utilizado tres tamaños de imágenes: 512x512, 1024x1024 (que ya eran entrenables en la versión original) y 2048x2048 píxeles, que sería el siguiente tamaño ideal de imagen con el que entrenar el modelo y actualmente no es posible. Sobre cada tamaño de imagen se han aplicado distintas combinaciones de las técnicas anteriormente presentadas. Para el cálculo de funciones de pérdida se ha entrenado la red durante 50 épocas.

En la ejecución para un único nodo se han evaluado todas las técnicas indicadas en la subsección II-A. En la experimentación distribuida solamente se ha evaluado paralelismo de modelo junto con las mejoras para un único nodo, ya que los nodos utilizados solo disponían de una única GPU. Por lo tanto, no era posible combinar paralelismo de datos y de modelo. Para lograr el paralelismo de modelo, se han utilizados 3 nodos: 1 maestro y 2 *workers*. Dado que la U-Net consta principalmente de dos partes: *Encoder* y *Decoder*, cada una ha sido enviada a una GPU distinta. La división del modelo, sincronización de parámetros y nodos ha sido realizada íntegramente con Pytorch. Dado que cada GPU va a procesar etapa de *forward* y *backward*, los consumos de memoria indicados son la media.

IV. Resultados experimentales

En este apartado se muestran y discuten los resultados obtenidos aplicando las distintas técnicas en los dos escenarios planteados: Nodo único y múltiples nodos.

A. Nodo único

Los resultados se pueden visualizar para tamaño de imagen 512x512 píxeles y 1024x1024 píxeles en la figura 2. No hay resultados para 2048x2048 píxeles ya que solamente los métodos *AMP* y *Checkpointing* eran capaces de ejecutar una iteración de *forward*, pero ninguna de *backward*.

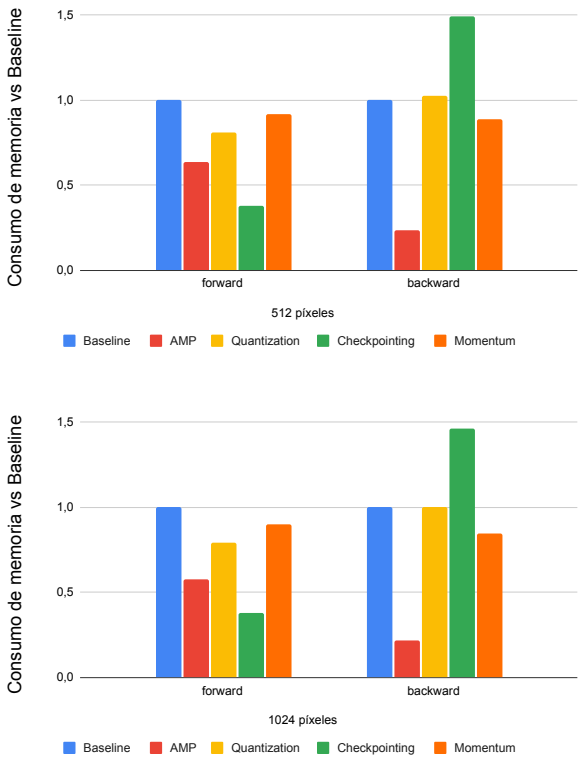


Fig. 2: Relación del consumo de memoria analizado para las distintas técnicas ejecutadas en un único nodo e imagen de 512 y 1024 píxeles. Menos es mejor.

Como se puede apreciar, todas las técnicas permiten reducir la memoria en la etapa de *forward*, destacando *AMP*, ya que los propios datos ocupan 16 bits en vez de 32 en memoria, y *Checkpointing* ya que todos los cálculos que deberían guardarse son descartados para ser posteriormente recalculados. En la fase de *backward*, AMP reduce considerablemente el uso de memoria hasta casi el 20 % de la memoria original. Por contra, dado que *Checkpointing* debe recalcular todos los valores descartados previamente, su consumo de memoria se multiplica haciendo que el consumo total sea igual o superior al consumo original. *Quantization* y *Momentum* mantienen unos valores similares en la fase de *backward*, llegando a reducir la memoria entorno a un 15 % en el caso de *Momentum*.

Otro parámetro a considerar, dado que se está modificando cómo funciona el modelo y los valores que se guardan en el mismo, son los valores de la función

de pérdida en entrenamiento y validación para cada una de las técnicas.

Tabla I: Valores de pérdida para 512x512 píxeles en cada una de las distintas técnicas. Se resaltan los mejores resultados.

	Train Loss	Validation Loss
Original	4,0504	6,0083
AMP	4,8612	6,2515
Quantization	4,7272	**5,9758**
Checkpointing (Chkp)	**4,0482**	6,0592
Momentum (Mom)	4,1456	5,9941

Como se puede apreciar en la tabla I, los peores resultados son arrojados por *AMP*. Esto se debe a la reducción de precisión de los tensores, ya que como se comentó al comienzo de este trabajo, a menor resolución, peores resultados. Por otro lado, el resto de técnicas mantienen unos resultados cercanos, o mejores, a los valores originales.

Dado que no son técnicas excluyentes entre si, se ha evaluado distintas combinaciones con el objetivo de estudiar el impacto sobre ellas al combinarlas. Los resultados se pueden visualizar en la tabla II. La mejor combinación se encuentra entre *Checkpointing* y *Momentum*, con un consumo de memoria bastante menor que individualmente, como se puede apreciar en la tabla III, aunque no consigue ser un consumo de memoria menor que usando *AMP*. Esto se debe a que es posible mantener la ganancia de *Checkpointing* en la fase de *forward* sin una penalización tan grande en la fase de *backward* gracias a la *ResNet* invertible introducida con *MomentumNet*. Esta combinación permite alcanzar reducciones de hasta el 20 %. En cuanto al tiempo medio de ejecución para cada época, se mantiene estable sin incrementar ni disminuir considerablemente para cada una de las técnicas.

Tabla II: Valores de pérdida para 512x512 píxeles combinando distintas técnicas. Se resaltan los mejores resultados.

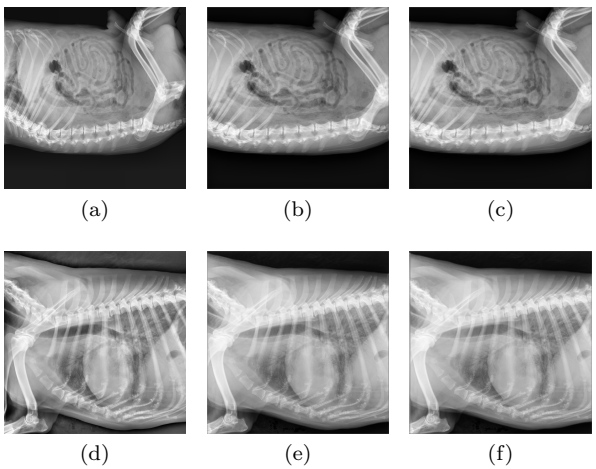
	Train Loss	Validation Loss
Original	**4,0504**	**6,0083**
AMP+Chkp	4,8612	6,2761
AMP+Quant	-	-
Chkp+Mom	4,1456	6,0115
AMP+Chkp+Mom	15,1742	17,2537

Tabla III: Consumo de memoria en MB para 512x512 píxeles y tiempo medio para la ejecución de una época.

Fase	Original	AMP	Chkp + Mom
Forward	757,76	481,28	337,92
Backward	440,32	102,40	563,20
Total	1.198,08	583,68	901,12
Tiempo (s)	35,63	34,02	37,52

Esta reducción de memoria no es tan grande como con *AMP*, que llega hasta un 50 %. Sin embargo, si nuestro caso de uso no permite reducir la precisión, la combinación de estas técnicas es una buena alternativa. La combinación de *Quantization* o *AMP* con otras técnicas aumenta la pérdida final debido a los cambios de precisión. Por otro lado, también aumenta el consumo de memoria en comparación con sus versiones individuales.

Finalmente, al estar tratando con imágenes, es necesario realizar una inspección visual sobre las mismas. Para ello se han escogido la imagen con mejor y peor precisión obtenidas del modelo original. Estos resultados se pueden observar en las figuras 3. Se puede apreciar que los resultados son prácticamente idénticos, sin ningún cambio destacable.



(a) (b) (c)

(d) (e) (f)

Fig. 3: Comparativa entre la salida con mejor tasa de acierto (a-c) y la peor tasa de acierto (d-f) en el conjunto de validación para la ejecución en un único nodo. A la izquierda (a,d), la imagen objetivo. En el centro (b,e) la imagen generada por el código original. A la derecha (c,f), la imagen generada usando *Checkpointing* y *Momentum*.

B. Múltiples Nodos

En la figura 4 se pueden apreciar las relaciones del consumo de memoria en el entorno distribuido con distintas técnicas comparado con la ejecución en un único nodo (*baseline*).

Paralelizar el modelo en dos GPUs ha reducido el consumo total de memoria hasta un 39 % y 49 % para imágenes de 512 y 1024 píxeles, respectivamente. Si aplicamos sobre esta implementación distribuida las técnicas vistas para la ejecución en un único nodo, podemos observar que los resultados siguen una tendencia similar. *Checkpointing* reduce considerablemente la memoria en la etapa de *forward*, pero incrementa en *backward*, *Momentum* reduce principalmente en *backward* y *AMP* reduce considerablemente la memoria en ambas etapas.

Si combinamos técnicas de la misma forma que en la ejecución para un nodo único, se puede llegar a alcanzar una reducción media del consumo de memoria por GPU hasta del 70 %. Esto permite entrenar el modelo con imágenes de 2048x2048 píxeles, algo que antes no era posible.

En la figura 5 se pueden ver la reducción del consumo respecto a la ejecución del modelo distribuido usando AMP. Como se puede apreciar, el consumo de memoria aumenta, considerablemente según la técnica, durante la etapa de *backwards* para todas las técnicas. Solamente la combinación de *AMP*

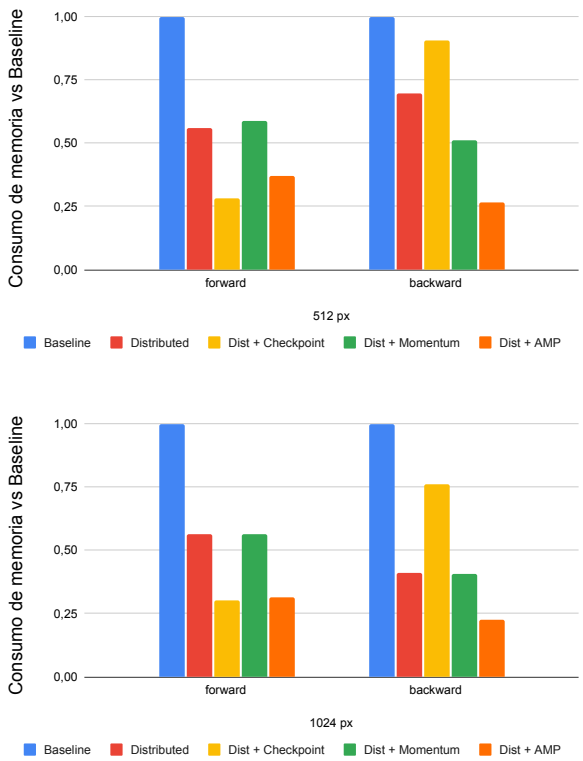


Fig. 4: Relación del consumo de memoria analizado para las distintas técnicas en ejecución distribuida e imagen de 512 y 1024 píxeles. Menos es mejor.

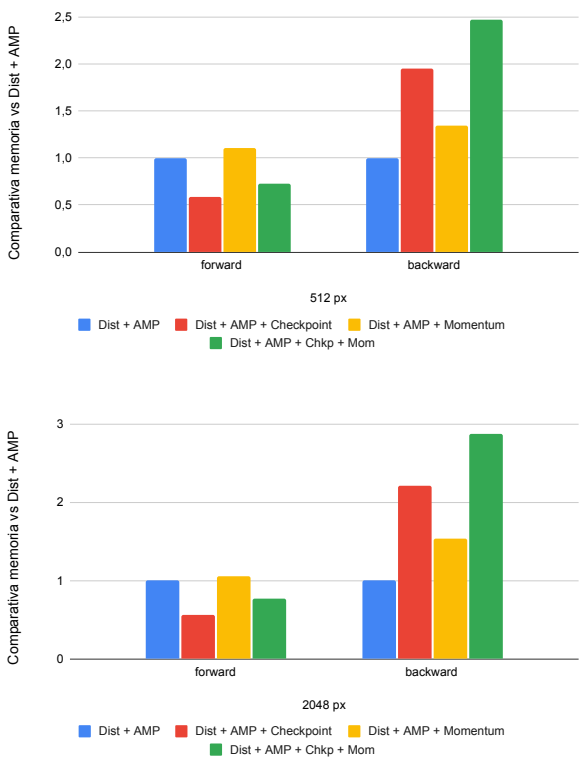


Fig. 5: Relación del consumo de memoria comparando la ejecución distribuida usando AMP y distintas combinaciones posibles en ejecución distribuida e imagen de 512 y 1024 píxeles. Menos es mejor.

y *Checkpointing* ofrece un consumo medio de memoria igual a la ejecución distribuida con *AMP* (ver tabla V). Esto se debe a que el consumo durante la etapa de *forward* decrece lo mismo que aumenta el de *backward*. Esta configuración es interesante para modelos cuyo consumo de memoria en *forward* sea considerablemente superior a *backward*, de forma que se equilibre el consumo.

De forma similar al nodo único, las modificaciones en la red se comprueban verificando que la función de pérdida devuelve valores similares a la red original. Como se puede apreciar en la tabla IV.

Tabla IV: Valores de pérdida para 512x512 píxeles combinando distintas técnicas en entorno distribuido. Se destacan los mejores resultados.

	Train Loss	Validation Loss
Original	**4,0504**	6,0083
Dist	4,5295	5,9778
Dist + AMP	4,6431	**5,7533**
Dist + AMP + Chkp	4,6431	**5,7533**
Dist + AMP + Mom	4,5370	5,7758
Dist + AMP + Chkp +Mom	4,5370	5,7758

Tabla V: Consumo de memoria en MB para 512x512 píxeles en entorno distribuido y tiempo medio para la ejecución de una época.

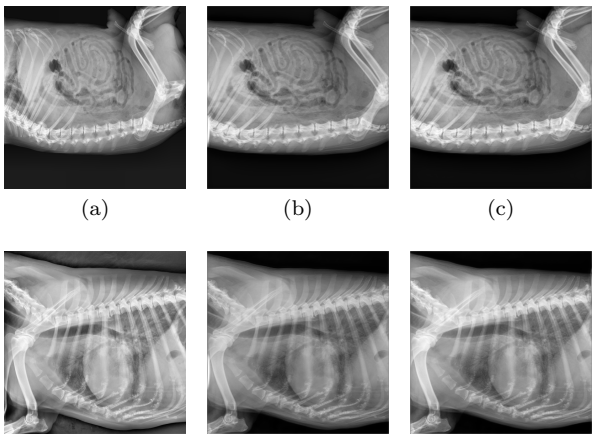
Fase	Dist	Dist + AMP	Dist + AMP + Chkp
Forward	424,96	281,60	163,84
Backward	307,20	117,76	230,40
Total	732,16	399,36	394,24
Tiempo (s)	53,29	40,69	57,26

La pérdida obtenida en la implementación con múltiples nodos es distinta del nodo único debido a pequeños cambios que se han tenido que realizar para poder hacer paralelismo de modelo. Concretamente la ausencia de un planificador para la tasa de aprendizaje, fijado durante todo el entrenamiento.

A diferencia de la versión con un solo nodo (ver tabla II), en esta ocasión incluir *AMP* no empeora los resultados. Esto se debe a que en los nodos distribuidos se utilizan tarjetas de la serie 3000 de Nvidia, las cuales incluyen *TensorCores*, que permiten aumentar ligeramente la precisión con respecto a *float16* cuando se realiza *AMP*. Para el caso de uso en el que nos encontramos, esta pequeña diferencia es muy importante, ya que puede significar la diferencia entre tejido blando o hueso. En este caso, el tiempo de ejecución medio para las épocas aumenta entre un 15 % y un 35 %, dependiendo de la configuración. Esto se debe al *overhead* introducido al tener que comunicar y sincronizar los nodos durante el entrenamiento.

Finalmente, al igual que en la ejecución de un único nodo, hay que realizar una inspección visual sobre los resultados. Para ello se han escogido la imagen con mejor y peor precisión obtenidas de la versión distribuida, que coinciden con el mejor y peor resultado del modelo original. Estos resultados se pueden observar en la figura 6. Al igual que se comentó en la sección anterior, los resultados son prácticamente idénticos, sin ningún cambio aparente. Destacar que en la figura 6, al usar AMP se consigue que los

huesos tengan una intensidad similar a la salida objetivo, aunque el tejido blando se visualiza de forma similar a la ejecución distribuida.

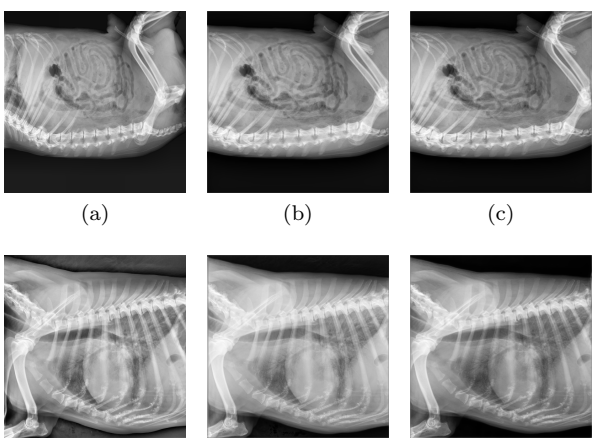


(a) (b) (c)

(d) (e) (f)

Fig. 6: Comparativa entre la salida con mejor tasa de acierto (a-c) y la peor tasa de acierto (d-f) en el conjunto de validación para la ejecución distribuida. A la izquierda (a,d), la imagen objetivo. En el centro (b,e) la imagen generada por el código distribuido. A la derecha (c,f), la imagen generada por el código distribuido usando AMP.

En la figura 7 se comparan los resultados entre la ejecución de un único nodo y el entrenamiento distribuido. En ella se pueden apreciar algunas diferencias para los dos resultados evaluados. En ambos casos, se puede apreciar en la zona de las vértebras que AMP define mejor los huesos. Por contra, en el peor resultado, la salida distribuida, independientemente de AMP como se puede apreciar en la figura 6, no llega a resaltar tanto el tejido blando. Esto muy probablemente se debe a las diferencias en el proceso de entrenamiento debido a la incompatibilidad con el mecanismo de sincronización actual de Pytorch para paralelismo de modelo en múltiples nodos.



(a) (b) (c)

(d) (e) (f)

Fig. 7: Comparativa entre el mejor resultado (a-c) y el peor resultado (d-f) de la ejecución en un único nodo usando *Checkpointing* y *Momentum* y la ejecución distribuida usando AMP. A la izquierda (a,d), la imagen objetivo. En el centro (b,e) la imagen generada en un único nodo usando *Checkpointing* y *Momentum*. A la derecha (c,f), la imagen generada por el código distribuido.

V. Conclusiones

En este trabajo se ha demostrado la utilidad de las técnicas actuales para la reducción de memoria, tanto en entornos con una única GPU, como distribuidos. Combinando distintas técnicas se puede llegar a reducir el consumo de memoria entre un 20 % en entornos con un único nodo, y 70 % en entornos distribuidos sin pérdidas de precisión significativas. No por ello las técnicas discutidas son siempre compatibles con cualquier caso de uso o red de neuronas, donde reducir la precisión de las operaciones empeoró considerablemente los resultados en algunos casos.

Adicionalmente, hay que tener en cuenta las capacidades de la librería que se esté utilizando, ya que no todas incluyen implementaciones para las técnicas discutidas. Por ejemplo, Pytorch no incluye herramientas para construir fácilmente una red invertible y por tanto se utilizó la librería MomentumNet para crear una ResNet invertible, y en esos casos recaería en el lado del investigador realizar la implementación pertinente para esa técnica y asegurarse de la compatibilidad con el resto de técnicas. Debido a esto, actualmente en Pytorch utilizar paralelismo de modelo mediante nodos interconectados (una GPU por nodo) limita las técnicas que se pueden utilizar. Por ejemplo, no es posible utilizar planificadores para la tasa de aprendizaje, fragmentar parámetros ni utilizar *pipelines*, ya que no son formatos compatibles con el sistema de comunicación y sincronización implementado por Pytorch para este paralelismo de modelo. Aun así, los resultados son prometedores e incluso con estas limitaciones los resultados finales son muy parecidos a la ejecución original, probando ser una alternativa viable para reducir la memoria.

Como trabajo futuro, estamos barajando el uso de soluciones elásticas que permitan adaptar las cargas de entrenamiento a los recursos disponibles. Esto permitiría un mejor balanceo de carga sobre los recursos existentes entre CPU y GPU. Adicionalmente, también estamos trabajando en abordar técnicas de reducción de consumo de memoria bajo sistemas de paralelización datos. Finalmente, también se está trabajando en la implementación de enjambres de redes de neuronas que permitan mejorar la precisión del modelo. Este trabajo también requerirá de un mayor grado de paralelismo, por lo que será necesario abordar técnicas de mejora del rendimiento.

Agradecimientos

Este trabajo ha sido apoyado por el Ministerio de Ciencia e Innovación, Agencia Estatal de Investigación: PDC2021-121656-I00 (MULTIRAD), financiado por MCIN/AEI/10.13039/501100011033 y por la Unión Europea 'NextGenerationEU'/PRTR; PID2019-110369RB-100/AEI/10.13039/501100011033 (RADHOR). También ha sido financiada por el Instituto de Salud Carlos III a través de los proyectos PMPTA22/00121 y PMPTA22/00118, cofinanciados por la UE 'NextGenerationEU'/PRTR. Y también ha sido financiada por la Comunidad de

Madrid, S2017/BMD-3867 RENIM-CM, cofinanciado por el Fondo Estructural y de Inversión Europeo. El CNIC cuenta con el apoyo del Instituto de Salud Carlos III, el Ministerio de Ciencia e Innovación y la Fundación Pro CNIC. Finalmente este trabajo ha sido financiado parcialmente por la Agencia Española de investigación a través del proyecto "New scalable I/O techniques for hybrid HPC and data-intensive workloads (SCIOT)" con referencia PID2022-138050NB-I00.

Referencias

[1] Mythreyi Bhargavan and Jonathan H. Sunshine, "Utilization of radiology services in the united states: Levels and trends in modalities, regions, and populations," *Radiology*, vol. 234, no. 3, pp. 824–832, 2005, PMID: 15681686.

[2] Rebecca Smith-Bindman, Diana Miglioretti, and Eric Larson, "Rising use of diagnostic medical imaging in a large integrated health system," *Health affairs (Project Hope)*, vol. 27, pp. 1491–502, 11 2008.

[3] Andrés Anaya-Isaza, Leonel Mera-Jiménez, and Martha Zequera-Diaz, "An overview of deep learning in medical imaging," *Informatics in Medicine Unlocked*, vol. 26, pp. 100723, 2021.

[4] Olaf Ronneberger, Philipp Fischer, and Thomas Brox, "U-net: Convolutional networks for biomedical image segmentation," 2015.

[5] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun, "Deep residual learning for image recognition," 2015.

[6] Karen Simonyan and Andrew Zisserman, "Very deep convolutional networks for large-scale image recognition," 2015.

[7] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever, "Language models are unsupervised multitask learners," 2019.

[8] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," *Journal of Machine Learning Research*, vol. 21, no. 140, pp. 1–67, 2020.

[9] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro, "Megatron-lm: Training multi-billion parameter language models using model parallelism," 2020.

[10] Le Hou, Youlong Cheng, Noam Shazeer, Niki Parmar, Yeqing Li, Panagiotis Korfiatis, Travis M. Drucker, Daniel J. Blezek, and Xiaodan Song, "High resolution medical image analysis with spatial partitioning," 2019.

[11] Sanderson D, Desco M, Del Cerro C, F, García-Real M, I, Ruíz-Fernández M, J, Garcia-Blas J, and Abella M, *Corrección automática del contraste en imagen radiográfica mediante aprendizaje profundo*, pp. 10–13, CASEIB, November 2022.

[12] Haris Ačkar, Ali Almisreb, and Mohd.A. Saleh, "A review on image enhancement techniques," *Southeast Europe Journal of Soft Computing*, vol. 8, 04 2019.

[13] Yuewen Sun, Litao Li, Peng Cong, Zhentao Wang, and Xiaojing Guo, "Enhancement of digital radiography image quality using a convolutional neural network," *Journal of X-Ray Science and Technology*, vol. 25, pp. 1–12, 10 2017.

[14] Houda Krouma, Youcef Ferdi, and Abdelmalik Taleb-Ahmedx, "Neural adaptive fractional order differential based algorithm for medical image enhancement," in *2018 International Conference on Signal, Image, Vision and their Applications (SIVA)*, 2018, pp. 1–6.

[15] Zhou Wang, Eero Simoncelli, and Alan Bovik, "Multiscale structural similarity for image quality assessment," *Proceedings of the IEEE Asilomar Conference Signals, Systems and Computers*, 02 2004.

[16] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin, "Training deep nets with sublinear memory cost," 2016.

[17] Audrūnas Gruslys, Remi Munos, Ivo Danihelka, Marc Lanctot, and Alex Graves, "Memory-efficient backpropagation through time," 2016.

[18] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu, "Mixed precision training," 2018.

[19] "BFloat16: The secret to high performance on Cloud TPUs — Google Cloud Blog — cloud.google.com," `https://cloud.google.com/blog/products/ai-machine-learning/bfloat16-the-secret-to-high-performance-on-cloud-tpus`, [Accessed 04-03-2024].

[20] Paresh Kharya, "NVIDIA Blogs: TensorFloat-32 Accelerates AI Training HPC upto 20x — blogs.nvidia.com," `https://blogs.nvidia.com/blog/tensorfloat-32-precision-format/`, [Accessed 04-03-2024].

[21] Yunchao Gong, Liu Liu, Ming Yang, and Lubomir Bourdev, "Compressing deep convolutional networks using vector quantization," 2014.

[22] Song Han, Huizi Mao, and William J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," 2016.

[23] Bo Chang, Lili Meng, Eldad Haber, Lars Ruthotto, David Begert, and Elliot Holtham, "Reversible architectures for arbitrarily deep residual neural networks," 2017.

[24] Jörn-Henrik Jacobsen, Arnold Smeulders, and Edouard Oyallon, "i-revnet: Deep invertible networks," 2018.

[25] Christian Etmann, Rihuan Ke, and Carola-Bibiane Schönlieb, "iunets: Fully invertible u-nets with learnable up- and downsampling," 2020.

[26] Sanket Tavarageri, Srinivas Sridharan, and Bharat Kaul, "Automatic model parallelism for deep neural networks with compiler and hardware support," 2019.

[27] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P. Xing, Joseph E. Gonzalez, and Ion Stoica, "Alpa: Automating inter- and intra-operator parallelism for distributed deep learning," 2022.

[28] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He, "Zero: Memory optimizations toward training trillion parameter models," 2020.

[29] Yuanzhong Xu, HyoukJoong Lee, Dehao Chen, Hongjun Choi, Blake Hechtman, and Shibo Wang, "Automatic cross-replica sharding of weight update in data-parallel training," 2020.

[30] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Mia Xu Chen, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen, "Gpipe: Efficient training of giant neural networks using pipeline parallelism," 2019.

[31] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala, "PyTorch: An Imperative Style, High-Performance Deep Learning Library," in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché Buc, E. Fox, and R. Garnett, Eds. 2019, pp. 8024–8035, Curran Associates, Inc.

[32] Michael E. Sander, Pierre Ablin, Mathieu Blondel, and Gabriel Peyré, "Momentum residual neural networks," 2021.

[33] Yuanzhong Xu, HyoukJoong Lee, Dehao Chen, Blake Hechtman, Yanping Huang, Rahul Joshi, Maxim Krikun, Dmitry Lepikhin, Andy Ly, Marcello Maggioni, Ruoming Pang, Noam Shazeer, Shibo Wang, Tao Wang, Yonghui Wu, and Zhifeng Chen, "Gspmd: General and scalable parallelization for ml computation graphs," 2021.

[34] Jinhui Yuan, Xinqi Li, Cheng Cheng, Juncheng Liu, Ran Guo, Shenghang Cai, Chi Yao, Fei Yang, Xiaodong Yi, Chuan Wu, Haoran Zhang, and Jie Zhao, "Oneflow: Redesign the distributed deep learning framework from scratch," 2022.

[35] Kai Lv, Yuqing Yang, Tengxiao Liu, Qinghui Gao, Qipeng Guo, and Xipeng Qiu, "Full parameter fine-tuning for large language models with limited resources," 2023.