Modelado y Optimización del Trade-off Cómputo–Memoria mediante Estrategias de Rematerialización en Entrenamiento de Redes Neuronales

Andrey Felipe Leguizamo Cespedes

Santiago González Flores

Trabajo de Grado para Optar el Título de Ingeniero de Sistemas

Director

Luis Alejandro Torres Niño

Magíster en Ingeniería de Sistemas e Informática

Universidad Industrial de Santander

Facultad de Ingenierías Físico-Mecánicas

Escuela de Ingeniería de Sistemas e Informática

Bucaramanga

2026

Agradecimientos

La culminación de este trabajo de grado bajo la modalidad investigativa, titulado «Modelado y Optimización del Trade-off Cómputo–Memoria mediante Estrategias de Rematerialización en Entrenamiento de Redes Neuronales», no habría sido posible sin el apoyo de la Universidad Industrial de Santander.

Extendemos nuestro reconocimiento al profesor MSc. Luis Alejandro Torres Niño, por su acompañamiento y orientación académica durante todas las etapas del proyecto.

A la Unidad de Supercómputo y Cómputo Científico (SC3), vinculada a la Vicerrectoría de Investigación y Extensión (VIE) de la Universidad Industrial de Santander, por brindarnos el acceso al supercomputador GUANE, recurso de hardware fundamental para el desarrollo experimental de este trabajo.

Un agradecimiento final a nuestras familias y docentes que hicieron parte de nuestra formación profesional, quienes con sus enseñanzas contribuyeron al fortalecimiento de nuestras competencias académicas y personales.

Tabla de Contenido

Lista de tablas

Lista de figuras

Lista de Apéndices

Los apéndices están adjuntos y puede visualizarlos en la base de datos de la biblioteca UIS.

Resumen

Título: Modelado y Optimización del Trade-off Cómputo–Memoria mediante Estrategias de Rematerialización en Entrenamiento de Redes Neuronales.*

Autores: Andrey Felipe Leguizamo Cespedes; Santiago González Flores.**

Palabras Clave: redes neuronales profundas; consumo de memoria; rematerialización de activaciones; gradient checkpointing; optimización cómputo memoria.

Descripción: Durante el entrenamiento de redes neuronales profundas, la etapa de retropropagación exige retener en memoria las activaciones intermedias generadas en el forward pass, lo que provoca un consumo de memoria que restringe el tamaño de lote viable y, en dispositivos con recursos acotados, puede hacer inviable el entrenamiento de modelos de gran escala. La rematerialización de activaciones es una técnica que mitiga este problema descartando selectivamente dichas activaciones y recomputándolas cuando se necesitan durante la retropropagación, trasladando así la presión del sistema de memoria hacia el presupuesto de cómputo.

Este trabajo desarrolló un estudio experimental y de modelado predictivo de ese compromiso. Se construyó un conjunto de datos de 2.064 configuraciones de entrenamiento, obtenidas mediante perfilado con PyTorch Profiler sobre seis arquitecturas de referencia (MLP-3, MLP-5, MLP-7, LeNet-5, AlexNet y VGG-16) ejecutadas en dos GPU de distinta familia (NVIDIA y AMD), empleando la infraestructura del supercomputador GUANE de la Unidad SC3–VIE de la UIS. Sobre estos datos se entrenaron el SGD Regressor y Random Forest que alcanzaron, en el conjunto de prueba, $R^2 = 0{,}8594$ para el tiempo total y $R^2 = 0{,}8330$ para el consumo de memoria.

La validación mediante las pruebas estadísticas mostró que aplicar rematerialización produce diferencias significativas tanto en memoria como en tiempo frente al entrenamiento sin la técnica, mientras que la elección entre las variantes selectiva y uniforme no introduce diferencias estadísticamente distinguibles del ruido experimental. El análisis desagregado por arquitectura permitió establecer lineamientos concretos: en redes MLP y LeNet-5 la técnica no reporta beneficio; en AlexNet ofrece mejoras moderadas; y en VGG-16 reduce el consumo de memoria, con un sobrecoste computacional acotado. Estos resultados proveen criterios reproducibles para seleccionar y configurar estrategias de rematerialización en entornos académicos e industriales con restricciones de hardware.

*Trabajo de Grado

** Facultad de FisicoMecánicas. Escuela de Sistemas. Director: Luis Alejandro Torres Niño. Magíster en Ingeniería de Sistemas e Informática

Abstract

Title: Modeling and Optimization of the Compute-Memory Trade-off through Rematerialization Strategies in Neural Network Training

Authors: Andrey Felipe Leguizamo Cespedes; Santiago González Flores

Keywords: deep neural networks; memory consumption; activation rematerialization; gradient checkpointing; compute-memory optimization.

Description: During the training of deep neural networks, the backpropagation stage requires storing the intermediate activations generated during the forward pass in memory, which results in memory consumption that limits the viable batch size and, on devices with limited resources, can make training large-scale models unfeasible. Activation rematerialization is a technique that mitigates this problem by selectively discarding these activations and recomputing them when needed during backpropagation, thereby shifting the pressure from the memory system to the computational budget.

This study conducted an experimental and predictive modeling analysis of this trade-off. A dataset of 2,064 training configurations was constructed, obtained through profiling with the PyTorch Profiler on six reference architectures (MLP-3, MLP-5, MLP-7, LeNet-5, AlexNet, and VGG-16) running on two GPUs from different families (NVIDIA and AMD), using the infrastructure of the GUANE supercomputer at the SC3–VIE Unit of the UIS. The SGD Regressor and Random Forest were trained on this data and achieved, on the test set, $R^2 = 0.8594$ for total time and $R^2 = 0.8330$ for memory consumption.

Validation through statistical tests showed that applying rematerialization produces significant differences in both memory and time compared to training without the technique, while the choice between the selective and uniform variants does not introduce differences statistically distinguishable from experimental noise. The analysis broken down by architecture allowed us to establish specific guidelines: in MLP and LeNet-5 networks, the technique offers no benefit; in AlexNet, it offers moderate improvements; and in VGG-16, it reduces memory consumption with a limited computational overhead. These results provide reproducible criteria for selecting and configuring rematerialization strategies in academic and industrial environments with hardware constraints.

---

*Degree Work

** Faculty of Physicomechanical Engineering. School of Systems Engineering and Informatics. Director: Luis Alejandro Torres Niño. Master of Science in Systems Engineering and Computer Science

Introducción

El entrenamiento de redes neuronales profundas ha impulsado avances significativos en áreas como la visión por computador y el procesamiento del lenguaje natural. No obstante, estos modelos requieren grandes volúmenes de recursos computacionales, siendo la memoria uno de los principales cuellos de botella. Durante la propagación hacia adelante (*forward pass*), las activaciones intermedias deben almacenarse para calcular los gradientes en la retropropagación (*backward pass*). Este proceso genera un pico de uso de memoria que, en dispositivos con recursos limitados, restringe el tamaño de lote y reduce la eficiencia global del entrenamiento.

Una estrategia ampliamente utilizada para mitigar esta limitación es la rematerialización de activaciones, también conocida como *gradient checkpointing*. Esta técnica consiste en recomputar ciertos valores intermedios en lugar de almacenarlos, lo que disminuye el consumo máximo de memoria a costa de un mayor número de operaciones de cómputo. En los últimos años han surgido variantes más sofisticadas: algunas orientadas a escalar hacia grafos de gran tamaño, otras enfocadas en arquitecturas heterogéneas, e incluso propuestas dinámicas que ajustan el plan de recomputación durante la ejecución (Bartan et al., 2023; Beaumont, Eyraud-Dubois, Herrmann, & Lima, 2024; Kirisame et al., 2021).

A pesar de estos avances, aún no se dispone de un modelo predictivo validado experimentalmente que permita estimar con precisión el balance entre coste computacional y uso de memoria en distintos escenarios de entrenamiento. Esta ausencia limita la posibilidad de definir configuraciones óptimas que sean reproducibles y defendibles tanto en investigación como en aplicaciones prácticas.

El presente proyecto busca atender esta brecha mediante el modelado y optimización del *trade-off* cómputo–memoria en el entrenamiento de redes neuronales utilizando estrategias de rematerialización. Se plantea la formulación de un modelo matemático predictivo, basado en datos de perfilado y validado experimentalmente, que permita predecir y optimizar configuraciones bajo diferentes restricciones de hardware y topología de red. Con ello, se pretende no solo explicar el fenómeno, sino también ofrecer lineamientos prácticos que faciliten el entrenamiento eficiente en entornos de recursos limitados, con aplicaciones tanto en la academia como en la industria.

1. Planteamiento Y Justificación Del Problema

El entrenamiento de redes neuronales profundas a gran escala está fuertemente condicionado por la disponibilidad de memoria, en particular en entornos híbridos CPU–GPU y en dispositivos con recursos limitados. Durante la propagación hacia adelante (*forward pass*), las activaciones intermedias deben almacenarse para calcular los gradientes en la retropropagación (*backward pass*). Este proceso genera un pico elevado de uso de memoria que restringe el tamaño de lote y, en algunos casos, compromete la viabilidad misma del entrenamiento.

Una estrategia ampliamente utilizada para mitigar esta limitación es la rematerialización de activaciones, también conocida como *gradient checkpointing*. Esta técnica consiste en recomputar de manera selectiva ciertos valores intermedios en lugar de almacenarlos, lo que reduce el consumo máximo de memoria a costa de un incremento controlado en el coste computacional. En los últimos años, la investigación ha propuesto variantes más sofisticadas: algunas orientadas a escalar hacia grafos de gran tamaño, otras enfocadas en arquitecturas heterogéneas, e incluso estrategias dinámicas que ajustan el plan de recomputación durante la ejecución. Entre ellas destacan:

Bartan et al. (2023) presentan *Moccasin*, una formulación basada en *constraint programming* con complejidad lineal en el número de nodos, lo que permite optimizar grafos de gran tamaño de forma práctica.

Beaumont, Eyraud-Dubois, Herrmann, & Lima (2024) exploran la combinación de *checkpointing* óptimo con heterogeneidad de dispositivos, formulando modelos para cadenas de cómputo en arquitecturas mixtas.

Kirisame et al. (2021) proponen una estrategia dinámica de rematerialización capaz de adaptar el plan de recomputación durante la ejecución para equilibrar carga y uso de memoria.

No obstante, a pesar de estos avances, aún no se dispone de un modelo predictivo validado experimentalmente que capture con precisión el compromiso entre cómputo y memoria en distintos escenarios de entrenamiento, considerando arquitecturas, topologías de red y restricciones de hardware. Esta ausencia limita la posibilidad de establecer configuraciones óptimas que sean defendibles, reproducibles y aplicables en la práctica.

En este contexto, surge la siguiente pregunta de investigación:

¿Cómo puede un modelo predictivo del coste computacional, calibrado con métricas de perfilado en GPU, identificar configuraciones óptimas de puntos de rematerialización que optimicen el *trade-off* cómputo–memoria en función de la topología de red, el tamaño de lote y las restricciones de memoria?

El presente trabajo se enmarca en esta brecha. A través de la formulación matemática del problema, la definición de métricas específicas y la validación experimental en arquitecturas MLP y CNN, se busca desarrollar un modelo que no solo explique, sino que también prediga y optimice este compromiso. De esta manera, se pretende ofrecer lineamientos prácticos aplicables tanto en investigación como en entornos productivos, contribuyendo a la formación de ingenieros capaces de diseñar soluciones eficientes en escenarios de recursos limitados.

2. Objetivos

2.1 Objetivo General

Formular y validar un modelo matemático predictivo que caracterice y optimice el *trade-off* entre coste computacional y uso de memoria en el entrenamiento de redes neuronales mediante rematerialización de activaciones, integrando la configuración arquitectónica, parámetros de entrenamiento, ubicación de *checkpoints* y métricas experimentales, con el fin de establecer un marco analítico y experimental reproducible para identificar configuraciones óptimas bajo restricciones de hardware y topología de red.

2.2 Objetivos Específicos

- Definir métricas e indicadores para evaluar el *trade-off* cómputo–memoria en entrenamientos con y sin rematerialización de activaciones, considerando medidas como pico de uso de memoria, tiempo promedio por iteración, *throughput*, factor de recomputación y eficiencia global, obtenidas mediante perfilado en GPU con PyTorch Profiler.
- Diseñar y entrenar un modelo matemático predictivo del coste global de entrenamiento en función de la granularidad y ubicación de puntos de rematerialización, empleando técnicas de *machine learning* como Regresión Lineal Múltiple y *Random Forest*, a partir de datos de perfilado por capa (tiempos y uso de memoria) obtenidos en GPU.
- Calibrar y validar el modelo predictivo mediante un protocolo experimental definido, comparando las configuraciones óptimas sugeridas por el modelo exclusivamente contra dos estrategias de referencia: (i) entrenamiento sin rematerialización y (ii) *uniform*

segment checkpointing (*checkpointing* segmentado uniforme), evaluando su desempeño en entornos GPU con métricas predefinidas y pruebas estadísticas para determinar significancia.

- Analizar e interpretar los resultados para confirmar o refutar la hipótesis de que existen configuraciones óptimas dependientes de la topología de red, el tamaño de lote y las restricciones de memoria, generando lineamientos prácticos de aplicación.

3. Marco De Referencia

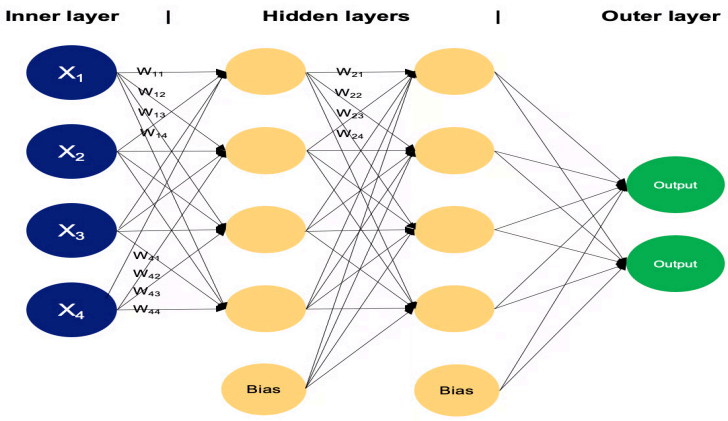
3.1 Marco Conceptual

3.1.1 Perceptrón Multicapa (MLP)

Un perceptrón multicapa (MLP) está compuesto por tres tipos de capas: la capa de entrada, las capas ocultas y la capa de salida. La capa de entrada recibe el conjunto de características del problema, representadas como un vector de valores (x_1, x_2, ..., x□). Cada una de estas variables es transmitida a las neuronas de la siguiente capa (Jaiswal, 2025).

Figura 1

Arquitectura Esquemática de un Perceptrón Multicapa (MLP)



Las capas ocultas constituyen el núcleo del procesamiento. En ellas, cada neurona está completamente conectada con las neuronas de la capa anterior mediante pesos w_i, que determinan la influencia de cada entrada, y posee un sesgo b que ajusta su umbral de activación. La *preactivación* z de una neurona –es decir, la combinación lineal de sus entradas ponderadas– se expresa como:

$$z = \sum_{i=1}^{n} w_i \cdot x_i + b$$

donde n es el número total de conexiones de entrada, w_i es el peso asociado a la i-ésima entrada, x_i es el valor de dicha entrada y b es el término de sesgo. Esta preactivación es transformada mediante una función de activación no lineal σ, produciendo la activación de salida de la neurona:

$$a = \sigma(z)$$

La no linealidad de σ es lo que confiere al MLP la capacidad de representar relaciones complejas entre entradas y salidas, más allá de lo que permitiría una transformación puramente lineal. La capa de salida recibe la información procesada por la última capa oculta y produce la

predicción final, cuyo número de neuronas y función de salida dependen de la naturaleza de la tarea: un valor escalar en regresión o una distribución de probabilidad sobre clases en clasificación (Jaiswal, 2025).

El entrenamiento del MLP se lleva a cabo mediante el algoritmo de retropropagación (*backpropagation*), que calcula los gradientes de la función de pérdida respecto a todos los parámetros de la red y los utiliza para actualizar iterativamente pesos y sesgos mediante un optimizador basado en gradiente, como el descenso de gradiente estocástico (SGD). Este proceso es de importancia central en el presente trabajo, pues el costo de memoria que impone el paso hacia atrás es precisamente el problema que la rematerialización de activaciones busca mitigar. Los detalles matemáticos completos, incluyendo la derivación de gradientes mediante la regla de la cadena y su implicación directa en el consumo de memoria, se desarrollan en la subsección siguiente (Chen et al., 2016; Jaiswal, 2025).

3.1.2 Retropropagación (Backpropagation)

La retropropagación es el algoritmo que permite calcular de forma eficiente los gradientes de la función de pérdida respecto a todos los parámetros de una red neuronal profunda, habilitando así el ajuste iterativo de pesos y sesgos durante el entrenamiento. Su comprensión es fundamental en el contexto de este trabajo porque determina directamente qué información debe conservarse en memoria durante el forward pass y, en consecuencia, qué es lo que la rematerialización de activaciones busca optimizar (Chen et al., 2016).

Función de pérdida y formulación del problema

Sea una red neuronal con *L* capas. La capa *k* aplica una transformación afín seguida de una no linealidad: recibe la activación a_{k-1} de la capa anterior, calcula la preactivación $z = W_k \cdot a_{k-1} + b_k$ y produce la activación $a_k = \sigma(z_k)$, donde W_k y b_k son los pesos y sesgos de la capa *k* y σ es la función de activación. La salida de la red a_L, se evalúa con una función de pérdida *L* que mide la discrepancia entre la predicción y el valor objetivo *y*. Para problemas de regresión es habitual emplear el error cuadrático medio.

$$L = \frac{1}{2}\left\|a_L - y\right\|^2$$

El objetivo del entrenamiento es encontrar los parámetros $\{W_k, b_k\}_{k=1}^{L}$ que minimicen *L* sobre el conjunto de entrenamiento, lo cual requiere calcular el gradiente de *L* respecto a cada parámetro en cada capa.

Propagación hacia atrás del error

Se define el término de error δk como el gradiente de la pérdida respecto a la preactivación de la capa k:

$$\delta_k = \frac{\partial L}{\partial z_k}$$

Para la capa de salida L, aplicando la regla de la cadena directamente sobre la función de pérdida:

$$\delta_L = \frac{\partial L}{\partial a_L} \odot \sigma'(z_L)$$

donde $\odot$ denota el producto elemento a elemento (producto de Hadamard) y σ' es la derivada de la función de activación. Para las capas interiores, el error se propaga hacia atrás mediante la siguiente recursión, que le da nombre al algoritmo:

$$\delta_k = W_{k+1}^T \delta_{k+1} \odot \sigma'(z_k), \qquad k = L\text{-}1,\ldots,1$$

La actualización de parámetros por descenso de gradiente estocástico (SGD) queda entonces:

$$W_k \leftarrow W_k - \alpha \cdot \frac{\partial L}{\partial W_L}, \qquad b_k \leftarrow b_k - \alpha - \frac{\partial L}{\partial b_L}$$

donde $\alpha > 0$ es la tasa de aprendizaje (Chen et al., 2026).

Implicación de memoria y conexión con la rematerialización

La ecuación de retropropagación $\delta_k = W_{k+1}^T \delta_{k+1} \odot \sigma'(z_k)$ revela el requerimiento crítico de memoria del algoritmo: para computar

3.1.3 Red Neuronal Convolucional (CNN)

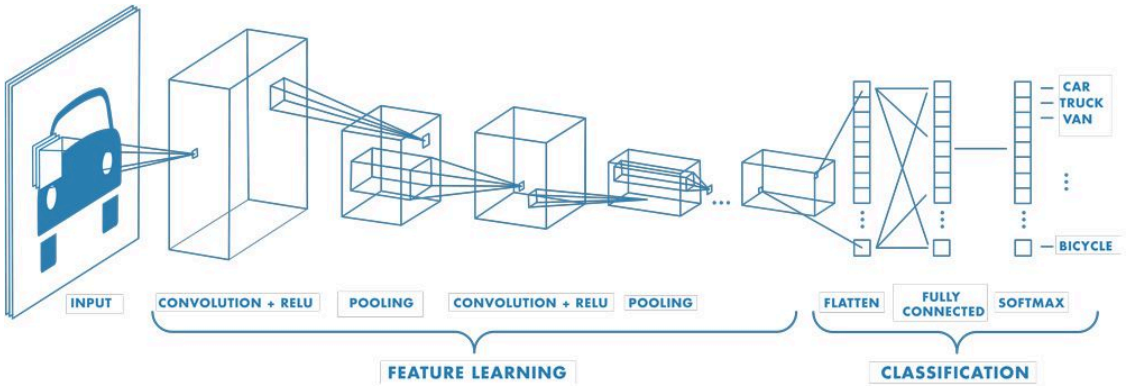
Una red neuronal convolucional (CNN) es un modelo avanzado de red neuronal artificial especialmente diseñado para el procesamiento de datos estructurados en forma de matrices, como imágenes. Técnicamente, una CNN está compuesta por varias capas principales: capas convolucionales, capas de activación (usualmente ReLU) y capas de agrupamiento (*pooling*), que se combinan para extraer y jerarquizar características de complejidad creciente (IBM, s.f.-a).

La capa convolucional es el núcleo del modelo y consiste en un conjunto de filtros o *kernels* que se aplican convolucionando sobre la entrada, usualmente una matriz tridimensional. Cada filtro detecta características locales específicas en la entrada, generando mapas de características (*feature maps*) que representan la presencia y localización de dichas características. El proceso incluye parámetros técnicos como *stride* (el desplazamiento del filtro sobre la entrada) y *padding* (relleno con ceros para controlar la dimensión espacial de la salida). La profundidad de la salida está dada por el número de filtros usados (IBM, s.f.-a).

La activación no lineal, generalmente aplicada mediante la función ReLU, introduce no linealidad al modelo, permitiendo captar patrones complejos que no serían detectables con una combinación lineal de entradas. Posteriormente, la capa de agrupamiento reduce la dimensionalidad espacial de los mapas de características mediante técnicas como *max pooling*, lo que reduce la carga computacional y ayuda a generalizar al eliminar ruido de detalles insignificantes (IBM, s.f.-a).

Figura 2

Esquema Conceptual de una Red Neuronal Convolucional



Nota. Representación del flujo de datos a través de capas convolucionales, de agrupamiento y totalmente conectadas. Adaptado de IBM (s.f.-a).

Una de las formas de entrenar este tipo de redes neuronales es la optimización por gradiente, conocida comúnmente como descenso por gradiente: un algoritmo iterativo ampliamente utilizado para minimizar funciones de costo en aprendizaje automático y redes neuronales. Su propósito es encontrar los valores óptimos de los parámetros del modelo (por ejemplo, los pesos en una red neuronal) que minimizan el error entre la predicción del modelo y los valores reales (IBM, s.f.-b).

Técnicamente, el algoritmo calcula el gradiente (el vector de derivadas parciales) de la función de costo respecto a los parámetros, y ajusta estos parámetros en la dirección opuesta al gradiente, porque esta dirección indica el descenso más rápido de la función. Matemáticamente, el algoritmo actualiza los parámetros **w** según la regla (IBM, s.f.-b):

$$w_{t+1} = w_t - \alpha \cdot \nabla_w J(w_t)$$

Donde:

- w_t son los parámetros en la iteración t.
- α es la tasa de aprendizaje, que controla el tamaño del paso.
- $\nabla_w J(w_t)$ es el gradiente de la función de costo J respecto a w_t.

Este proceso se repite iterativamente hasta que la función de costo alcanza un mínimo (local o global), o hasta que se cumple un criterio de parada.

3.1.3.1 Convolución. La convolución en arquitecturas de Redes Neuronales Convolucionales (CNN) es una operación matemática avanzada y el elemento fundamental para que estas redes aprendan de manera eficiente representaciones jerárquicas espaciales a partir de datos estructurados en cuadrículas, como imágenes o señales en 2D y 3D (Szegedy et al., 2015).

Esta operación consiste en deslizar sistemáticamente un filtro o núcleo (una pequeña matriz de parámetros entrenables) sobre las dimensiones espaciales de la entrada (ancho y alto), para realizar productos escalares locales entre los valores de la entrada y los pesos del filtro. El resultado es un mapa de activación o mapa de características, donde cada valor refleja la presencia e intensidad de una característica concreta detectada por el filtro en una región específica de la entrada (Szegedy et al., 2015).

Formalmente, se expresa la entrada a la capa convolucional por I (que puede ser una imagen o un mapa de características previo) y el filtro o *kernel* por K de tamaño $F \times F$ (Szegedy et al., 2015). La convolución 2D estándar en la que el núcleo se rota 180° está dada por:

$$S(i, j) = (I * K)(i, j) = \Sigma\square_{=0}\text{^}(F\text{-}1)\ \Sigma\square_{=0}\text{^}(F\text{-}1)\ I(i+m, j+n) \cdot K(F-m, F-n)$$

No obstante, en CNN se usa la operación equivalente, conocida como correlación cruzada, donde el núcleo no se rota:

$$S(i, j) = \Sigma_{u=0}\text{^}(F\text{-}1)\ \Sigma_{v=0}\text{^}(F\text{-}1)\ K(u, v) \cdot I(i+u, j+v)$$

Donde:

- $I(i+u, j+v)$ representa el valor en la posición $(i+u, j+v)$ de la entrada.

- *K(u, v)* es el valor del filtro en la posición *(u, v)*.
- *S(i, j)* es el resultado de la convolución en la posición *(i, j)* para el mapa de activación.

El mapa resultante *S* conserva la estructura espacial de la entrada, resaltando localmente patrones detectados por el filtro. Para controlar el tamaño del mapa de salida se introducen otros parámetros como el *stride* s (paso con el que se mueve el filtro) y el *padding* p (relleno de ceros alrededor de la entrada), ajustando el cálculo del tamaño de salida según:

$$W\_out = (W - F + 2p) / s + 1$$

Donde *W* es el ancho o alto de la entrada (suponiendo cuadrada la entrada).

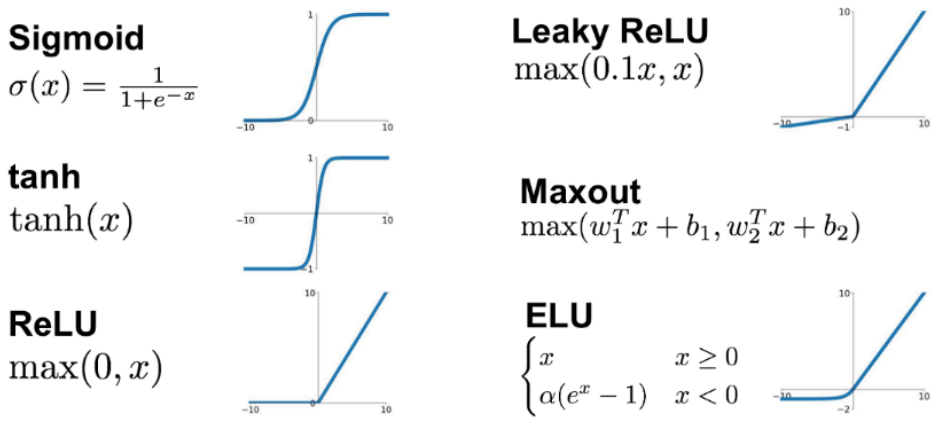
Las CNN modernas usan múltiples filtros en cada capa convolucional, donde cada filtro aprende a detectar diferentes características, generando múltiples mapas de activación, que luego son apilados como un tensor para capas posteriores. Esto permite modelar representaciones complejas y jerárquicas, desde detección de bordes simples en las primeras capas, hasta formas y patrones de alto nivel en capas profundas (Szegedy et al., 2015).

Finalmente, la convolución aprovecha la compartición de pesos y la vecindad local para reducir drásticamente la cantidad de parámetros y operaciones comparado con redes totalmente conectadas, haciendo el aprendizaje eficiente y efectivo en datos espaciales y temporales (Szegedy et al., 2015).

3.1.3.2 Funciones de Activación. Las funciones de activación en inteligencia artificial son funciones matemáticas aplicadas a la salida de una neurona artificial que introducen no linealidad en el modelo. Esto permite que las redes neuronales aprendan y representen relaciones complejas y no lineales entre los datos de entrada y salida, lo cual es esencial para tareas de clasificación, regresión y reconocimiento de patrones. Entre las funciones de activación más comunes están la sigmoide, que limita la salida entre 0 y 1; la tangente hiperbólica (tanh), que produce una salida entre −1 y 1; y la ReLU (unidad lineal rectificada), que es actualmente la más usada porque es computacionalmente eficiente y evita problemas de saturación y gradientes muy pequeños que dificultan el entrenamiento. En general, las funciones de activación determinan si una neurona se activa o no, lo que define el aprendizaje y la capacidad predictiva de la red neuronal (Navarro, 2025).

Figura 3

Ejemplo de Funciones de Activación en Deep Learning



Nota. Gráficas comparativas de las funciones sigmoide, tanh y ReLU. Adaptado de Navarro (2025).

3.1.4 Métricas de Eficiencia y Rendimiento

Las métricas de eficiencia y rendimiento en inteligencia artificial evalúan qué tan bien un sistema utiliza recursos computacionales y cómo responde en términos de tiempo y capacidad operativa, aspectos críticos para aplicaciones en tiempo real y a gran escala.

3.1.4.1 Memoria Promedio. Mide el consumo promedio de memoria a lo largo de todas las iteraciones o pasos de entrenamiento, considerando tanto activaciones almacenadas como recomputadas.

$$M\_avg = (1/T)\ \Sigma\square_{=1}\hat{}T\ M\square$$

Donde:

- T: número total de pasos de entrenamiento (o iteraciones).
- $M\square$: memoria consumida en el momento t.

Un *M_avg* menor indica que la estrategia de rematerialización está logrando reducir la presión global sobre la memoria.

3.1.4.2 Ahorro Porcentual. Indica el porcentaje de memoria ahorrada gracias a la rematerialización, en comparación con el entrenamiento estándar sin aplicar esta técnica.

$$A\% = [(M\_base - M\_estrategia) / M\_base] \times 100$$

Donde:

- *M_base*: memoria máxima o promedio consumida sin rematerialización.
- *M_estrategia*: memoria máxima o promedio consumida con rematerialización.

Un valor alto de *A%* significa que la estrategia es efectiva en liberar memoria.

3.1.4.3 Tiempo de Resolución. Mide el tiempo requerido para completar una operación forward–backward o un epoch bajo una estrategia de rematerialización. Esta métrica captura el costo computacional adicional que introduce el volver a calcular activaciones.

$$T\_res = T\_base + T\_extra$$

Donde:

- *T_base*: tiempo sin aplicar rematerialización.
- *T_extra*: tiempo adicional generado por las recomputaciones.

Si *T_res* crece demasiado, la ganancia en memoria puede no justificar la pérdida en tiempo.

3.1.4.4 Factor de Recomputación. Es la proporción entre la cantidad total de operaciones realizadas con rematerialización frente a las operaciones en el escenario base. Representa el sobrecosto computacional introducido.

$$F\_rec = OPS\_estrategia / OPS\_base$$

Donde:

- *OPS_estrategia*: número de operaciones (flops) realizadas bajo rematerialización.
- *OPS_base*: número de operaciones realizadas en el entrenamiento normal.
- $F\_rec = 1$: no hay recomputación.
- $F\_rec > 1$: existe sobrecosto en cómputo.

Mientras más grande sea *F_rec*, mayor es el impacto en el tiempo de ejecución.

3.1.5 PyTorch Profiler

Es una herramienta de *profiling* y monitoreo de experimentos integrada en la biblioteca de aprendizaje profundo PyTorch que permite analizar y visualizar el rendimiento de un modelo

durante su ejecución, uso de GPU y memoria, a nivel de operaciones y de capas. Su utilidad facilita identificar cuellos de botella y cuantificar métricas como el tiempo promedio por iteración, *throughput* y consumo máximo de memoria, aspectos fundamentales en la evaluación del *trade-off* cómputo–memoria (Smith, 2021).

3.1.6 Modelo Predictivo: Regresión Lineal Múltiple

La regresión lineal múltiple es una técnica estadística utilizada para analizar y explicar por qué ocurren ciertos fenómenos, permitiendo identificar las variables independientes que influyen sobre una variable dependiente. A diferencia de la regresión lineal simple, que se basa en una única variable explicativa, esta técnica considera varias variables independientes y evalúa simultáneamente su influencia conjunta en la variable dependiente, que debe ser numérica o una variable ordinal con más de cinco categorías. Este método permite comparar modelos explicativos, así como predecir valores de la variable dependiente en función de ciertas características observadas en las variables independientes (Networkianos, s.f.).

Fórmula general:

$$Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \ldots + \beta_n X_n + \varepsilon$$

Donde:

- Y: variable dependiente o respuesta. Es el valor que se quiere explicar o predecir.
- β_0: intercepto (constante). Representa el valor esperado de Y cuando todas las variables independientes son iguales a 0.
- $\beta_1, \beta_2, \ldots, \beta_n$: coeficientes de regresión. Indican la magnitud y dirección del efecto que tiene cada variable independiente sobre Y, manteniendo constantes las demás.

- $X_1, X_2, ..., X_n$: variables independientes o explicativas, que representan los factores que influyen en la variable dependiente.
- ε: término de error, que recoge la variabilidad no explicada por el modelo y factores no considerados.

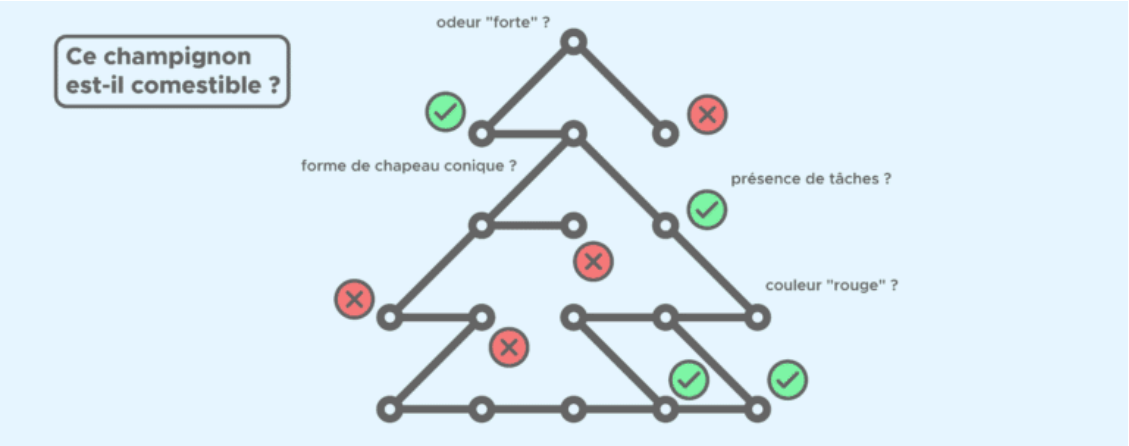
3.1.7 Modelo Predictivo: Random Forest

Es un algoritmo de aprendizaje automático (*machine learning*). Este método combina los resultados de múltiples árboles de decisión para mejorar la precisión y la robustez de las predicciones, tanto en tareas de clasificación como de regresión. El algoritmo funciona generando una gran cantidad de árboles de decisión, cada uno entrenado con diferentes subconjuntos aleatorios de los datos y de las variables predictoras. Este proceso implica que cada árbol es un modelo ligeramente diferente, lo que aumenta la diversidad y reduce la correlación entre ellos. Para realizar una predicción, *Random Forest* toma un voto mayoritario (en clasificación) o calcula un promedio (en regresión) de las salidas de todos los árboles, generando así un resultado más estable y fiable (Ras, 2023).

En la construcción de cada árbol, en cada nodo se elige la pregunta o la variable (*feature*) que maximiza la «ganancia de información» para dividir los datos. La importancia de las variables se puede medir evaluando cuánto mejora la pureza de las particiones al usar una variable específica. Este proceso ayuda a identificar qué variables o características tienen un mayor impacto en la predicción final del modelo (Ras, 2023).

Figura 4

Esquema de un Random Forest



Nota. Diagrama conceptual del funcionamiento de un Random Forest con múltiples árboles de decisión que emiten una predicción agregada. Adaptado de Ras (2023).

El *bagging* es una técnica de ensamblado basada en *bootstrapping* que usa *random forest*, cuyo objetivo es reducir la varianza de modelos de alto sobreajuste, como los árboles de decisión. Consiste en generar B subconjuntos de entrenamiento a partir de muestreo con reemplazo de los datos originales, entrenar un modelo en cada subconjunto y luego agregar sus predicciones (media en regresión, moda en clasificación) (Rodrigo, s.f.).

Matemáticamente, dada una muestra $\{Z_1, Z_2, ..., Z\square\}$ de varianza σ^2, la varianza de la media se reduce como:

$$Var(\bar{Z}) = \sigma^2 / n$$

Lo que explica por qué el promedio de múltiples modelos es más estable que un único estimador (Rodrigo, s.f.).

El procedimiento de *bagging* se resume en:

- Generar B *pseudo-training sets* mediante *bootstrapping* a partir de la muestra de entrenamiento original (Rodrigo, s.f.).

- Entrenar un árbol con cada una de las *B* muestras del paso anterior. Cada árbol se crea sin apenas restricciones y no se somete a *pruning*, por lo que tiene varianza alta pero poco sesgo. En la mayoría de los casos, la única regla de parada es el número mínimo de observaciones que deben tener los nodos terminales. El valor óptimo de este hiperparámetro puede obtenerse comparando el *out of bag error* o por validación cruzada (Rodrigo, s.f.).
- Para cada nueva observación, obtener la predicción de cada uno de los *B* árboles. El valor final de la predicción se obtiene como la media de las *B* predicciones en el caso de variables cuantitativas y como la clase predicha más frecuente (moda) para variables cualitativas (Rodrigo, s.f.).

3.1.8 Trade-off Cómputo–Memoria

El *trade-off* cómputo–memoria se refiere a la relación de balance entre dos recursos esenciales en el diseño y la ejecución de algoritmos, particularmente en redes neuronales y aprendizaje automático. Es decir, si se busca reducir el uso de memoria, puede ser necesario incrementar la cantidad de cálculos (recomputación), y viceversa. Esto significa que optimizar uno de estos recursos puede tener un coste en el otro, y en muchas circunstancias, es necesario decidir cuál opción resulta más eficiente dependiendo del contexto y las limitaciones del hardware (Minakova & Stefanov, 2022).

En literatura de redes neuronales profundas y técnicas como rematerialización de activaciones, el costo puede expresarse así:

$$C\_total = \alpha \cdot M + \beta \cdot F$$

Donde:

- **M**: memoria utilizada (número de parámetros, activaciones almacenadas, etc.).
- **F**: cantidad de operaciones de cómputo (número de recomputaciones).
- **α, β**: coeficientes de ponderación que dependen del hardware (tiempo de acceso a memoria vs. costo de cómputo).

3.1.9 Rematerialización de Activaciones (Gradient Checkpointing)

Los modelos grandes son costosos, tanto en sus dimensiones estáticas como dinámicas. Son difíciles de integrar en la GPU desde el principio y difíciles de entrenar una vez instalados en el dispositivo, ya que fuerzan un tamaño de lote demasiado pequeño para converger. El *Gradient Checkpointing* funciona omitiendo algunos valores de activación del grafo computacional. Esto reduce la memoria utilizada por el grafo computacional, lo que reduce la presión de memoria en general (y permite lotes más grandes en el proceso) (Chen et al., 2016).

El *forward pass* es el proceso de cálculo donde los datos de entrada fluyen a través de la red neuronal hacia la salida, generando una predicción (Chen et al., 2016).

Para una red con secuencia de funciones $f_1, f_2, ..., f\_L$:

$$a_0 = x; \quad a_1 = f_1(a_0); \quad a_2 = f_2(a_1); \quad ...; \quad a\_L = f\_L(a\_(L-1))$$

Donde:

- **x**: la entrada al modelo (vector de características).
- $\mathbf{a}_0$: se define como la entrada.
- **f_k**: la función de la capa *k*. Incluye la operación lineal (multiplicación por los pesos + *bias*) y la no linealidad (ReLU, sigmoide, etc.).
- **a_k**: la activación producida por la capa *k*. Es el resultado de aplicar *f_k* a la activación anterior.

- **a_L**: la salida final del modelo (antes de calcular la pérdida).

El *backward pass* es el proceso de calcular gradientes mediante la regla de la cadena, propagando errores desde la salida hacia la entrada para actualizar los parámetros. Los gradientes se calculan mediante la regla de la cadena (Chen et al., 2016):

$$\partial L/\partial a\_k = (\partial L/\partial a\_(k+1)) \cdot (\partial a\_(k+1)/\partial a\_k)$$

Donde:

- **L**: la función de pérdida (*loss function*), mide qué tan mal predice la red respecto a la salida deseada.
- **a_k**: la activación en la capa *k*, la salida que produce la capa *k*.
- **∂L/∂a_k**: el gradiente de la pérdida respecto a la activación *a_k*. Indica cuánto cambia el error si se cambia un poco *a_k*.
- **∂L/∂a_(k+1)**: el gradiente de la pérdida respecto a la activación de la siguiente capa.
- **∂a_(k+1)/∂a_k**: la derivada de la activación de la capa *k+1* respecto a la de la capa *k*; incluye pesos y función de activación.

Durante el *forward pass*:

- Seleccionar puntos de *checkpoint* cada *n* capas.
- Guardar solo las activaciones en esos puntos.
- Las activaciones intermedias no se guardan.

Durante el *backward pass* cuando se necesitan activaciones entre *checkpoints*:

- Re-ejecutar el *forward* desde el último *checkpoint*.
- Calcular las activaciones necesarias.
- Proceder con el *backward*.

Forward: $\mathbf{A_1 \rightarrow A_2 \rightarrow}$ **[checkpoint]** $\mathbf{\rightarrow A_3 \rightarrow A_4 \rightarrow}$ **[checkpoint]** $\mathbf{\rightarrow A_5}$

Backward: $\mathbf{A_1 \leftarrow A_2 \leftarrow [recompute] \leftarrow A_3 \leftarrow A_4 \leftarrow [recompute] \leftarrow A_5}$

3.1.10 Análisis de Varianza (ANOVA)

La prueba ANOVA unidireccional se utiliza cuando hay una variable independiente con dos o más grupos. El objetivo es determinar si existe una diferencia significativa entre las medias de los distintos grupos (Pedregosa et al., 2011).

Matemáticamente, el ANOVA descompone la variabilidad total de los datos en dos componentes:

- Variabilidad dentro del grupo: variabilidad causada por las diferencias dentro de los grupos individuales, que refleja fluctuaciones aleatorias.
- Variabilidad entre grupos: variabilidad causada por las diferencias entre las medias de los distintos grupos (Pedregosa et al., 2011).

$$F = MS\_between / MS\_within = [SS\_between/(k-1)] / [SS\_within/(N-k)]$$

Donde:

- $SS\_between = \Sigma_{i=1}^{k} n_i(\bar{x}_i - \bar{x})^2$ mide la variabilidad entre grupos.
- $SS\_within = \Sigma_{i=1}^{k} \Sigma_{□=1}^{(n_i)} (x_{i□} - \bar{x}_i)^2$ mide la variabilidad dentro de los grupos.
- k: número de grupos.
- N: número total de observaciones.

La prueba produce un estadístico F, que muestra la relación entre la variabilidad entre grupos y la variabilidad dentro de los grupos. Si el estadístico F es suficientemente grande, indica que al menos una de las medias de grupo es significativamente diferente de las demás.

Hipótesis nula H_0:

$$H_0: \mu_1 = \mu_2 = ... = \mu_{□}$$

Significa que todas las medias poblacionales de los grupos comparados son iguales. En otras palabras: no importa cuántos grupos (k) se tengan, bajo esta hipótesis se asume que no hay diferencias reales entre ellos.

Hipótesis alternativa H_1:

$$H_1: \exists\ i, j\ tal\ que\ \mu_i \neq \mu\square$$

Aquí se plantea que al menos dos de las medias poblacionales son diferentes. No necesariamente todas deben diferir, basta con que una sola media se separe de otra para rechazar la hipótesis nula.

3.1.11 Prueba Estadística: Tukey HSD (Honestly Significant Difference)

La prueba de Tukey HSD (Diferencia Honestamente Significativa) es un método estadístico avanzado utilizado para realizar comparaciones múltiples entre medias cuando un análisis de varianza (ANOVA) ha rechazado la hipótesis nula global de igualdad de medias. Su finalidad principal es determinar qué pares específicos de medias presentan diferencias estadísticamente significativas, mientras se controla rigurosamente el error tipo I derivado de la realización simultánea de múltiples pruebas. Para ello, la prueba se basa en la distribución del rango estudentizado, que evalúa la variabilidad máxima esperada entre las medias bajo la hipótesis nula (Tukey, 1949).

La fórmula central de la prueba para diseños balanceados, esto es, con igual número de observaciones en cada grupo, define la Diferencia Honestamente Significativa HSD y se escribe como (Tukey, 1949):

$$HSD = q\_(\alpha; k, df\_error) \times \sqrt{(MSE / n)}$$

En esta fórmula $q\_(\alpha; k, df\_error)$ representa el valor crítico de la distribución de rango estudentizado para un nivel de significancia α, k grupos comparados, y *df_error* grados de libertad del error del ANOVA. El término *MSE*, o *Mean Square Error*, es el estimador insesgado de varianza residual derivado del análisis de varianza, mientras que n es el tamaño muestral uniforme por grupo. Para contrastar si dos medias $\bar{y}_i$ y $\bar{y}$□ difieren significativamente, se calcula la diferencia absoluta entre ellas y se compara con HSD. Si esta diferencia supera el umbral, se concluye que las medias son significativamente distintas bajo el nivel de confianza seleccionado (Tukey, 1949).

En escenarios con tamaños muestrales desiguales, el procedimiento ajusta el denominador empleando la media armónica n_h entre las muestras implicadas según:

$$n\_h = 2 / (1/n_i + 1/n□)$$

El cálculo de HSD en este caso se adapta a:

$$HSD = q\_(\alpha; k, df\_error) \times \sqrt{(MSE / n\_h)}$$

Este ajuste asegura la precisión en el umbral dado que la variabilidad depende inversamente del tamaño efectivo de la muestra para cada comparación. A través de este análisis cuidadoso, la prueba de Tukey HSD proporciona un equilibrio entre la protección ante falsos positivos y la sensibilidad para detectar diferencias reales (Tukey, 1949).

3.1.12 Prueba Estadística: Kruskal–Wallis

La prueba estadística de Kruskal–Wallis es un método no paramétrico utilizado para contrastar si existen diferencias significativas entre tres o más grupos independientes cuando no se cumplen las condiciones necesarias para aplicar un ANOVA tradicional, principalmente la suposición de normalidad en la distribución de los datos o la homogeneidad de varianzas. Esta

prueba constituye una generalización de la prueba de Mann–Whitney–Wilcoxon para comparar más de dos grupos, evaluando si las medianas poblacionales difieren significativamente (Kruskal & Wallis, 1952).

El fundamento conceptual de la prueba es transformar los datos originales en sus rangos dentro del conjunto combinado de todas las muestras, ordenando todas las observaciones desde la menor hasta la mayor y asignándoles números enteros consecutivos. En caso de empates, se asigna a esas observaciones el rango promedio correspondiente. La hipótesis nula H_0 establece que todas las muestras provienen de poblaciones con la misma distribución (específicamente, con medianas iguales), mientras que la alternativa H_1 postula que al menos una muestra difiere en su mediana.

Sea k el número de grupos o muestras independientes, con tamaños individuales n_i para $i = 1, ..., k$, y $N = \Sigma_{i=1}^{k} n_i$ el total de observaciones. Se define $R_{i\square}$ como el rango asignado a la j-ésima observación del i-ésimo grupo tras ordenar conjuntamente todos los datos (Kruskal & Wallis, 1952).

Luego, para cada grupo se calcula la suma de rangos:

$$R_i = \Sigma_{\square=1}^{(n_i)} R_{i\square}$$

El estadístico de prueba de Kruskal–Wallis H se expresa como:

$$H = [12 / (N(N+1))] \cdot \Sigma_{i=1}^{k} (R_i^2 / n_i) - 3(N+1)$$

Este estadístico mide la dispersión de las sumas de rangos ajustadas por los tamaños de muestra individuales en relación con la distribución uniforme esperada bajo H_0. Si las medianas fueran iguales, las sumas de rangos deberían ser similares, resultando en un H bajo. Valores altos de H sugieren diferencias significativas entre grupos. Para corregir el efecto de observaciones empatadas, si existen, se debe ajustar H mediante un factor de corrección $C = 1 - \Sigma(t^3 - t) / (N^3$

$- N)$, donde la suma es sobre los grupos de datos empatados con tamaño t (Kruskal & Wallis, 1952). Entonces:

$$H' = H / C$$

En presencia de muestras grandes, bajo la hipótesis nula, la distribución de H' se aproxima a una distribución χ^2 con $k - 1$ grados de libertad, lo que permite obtener el valor p asociado (Kruskal & Wallis, 1952).

La prueba es especialmente adecuada para datos ordinales o con desviaciones notables a la normalidad, ofreciendo robustez frente a heterogeneidad de varianzas. Es también frecuentemente utilizada en situaciones donde el tamaño muestral es pequeño y no se desea asumir distribuciones paramétricas estrictas (Kruskal & Wallis, 1952).

3.1.13 Prueba de Levene

La prueba de Levene es un método estadístico que se utiliza para comprobar si k muestras tienen varianzas iguales. La igualdad de varianzas entre las muestras se denomina homogeneidad de varianza. Algunas pruebas estadísticas, como el análisis de varianza, asumen que las varianzas son iguales entre grupos o muestras. La prueba de Levene puede utilizarse para verificar esta suposición (Levene, 1960).

Hipótesis de la prueba:

Hipótesis nula (H_0):

$$H_0: \sigma_1^2 = \sigma_2^2 = \ldots = \sigma\square^2$$

Todas las varianzas poblacionales son iguales.

Hipótesis alternativa (H_1):

$$H_1: \exists\ i, j \text{ tal que } \sigma_i^2 \neq \sigma\square^2$$

Al menos una varianza difiere de otra.

En lugar de comparar varianzas directamente, la prueba de Levene transforma cada observación en la desviación absoluta respecto a un centro del grupo (media o mediana) (Levene, 1960):

$$Z_{i\square} = |Y_{i\square} - \bar{Y}_i|$$

Donde:

- $Y_{i\square}$: observación en el grupo i.
- $\bar{Y}_i$: medida de tendencia central del grupo i.
- $Z_{i\square}$: desviación absoluta de cada dato respecto al centro del grupo.

El estadístico de prueba se define como:

$$W = [(N-k)/(k-1)] \times [\Sigma_{i=1}^{k} n_i(\bar{Z}_i - \bar{Z}..)^2] / [\Sigma_{i=1}^{k} \Sigma_{\square=1}^{(n_i)} (Z_{i\square} - \bar{Z}_i.)^2]$$

Donde:

- $N = \Sigma_{i=1}^{k} n_i$: número total de observaciones en todos los grupos.
- k: número de grupos.
- n_i: tamaño de la muestra en el grupo i.
- $Z_{i\square}$: desviación absoluta de la observación j en el grupo i.
- $\bar{Z}_i. = (1/n_i) \Sigma_{\square=1}^{(n_i)} Z_{i\square}$: media de las desviaciones absolutas en el grupo i.
- $\bar{Z}.. = (1/N) \Sigma_{i=1}^{k} \Sigma_{\square=1}^{(n_i)} Z_{i\square}$: media global de todas las desviaciones absolutas.

El estadístico W sigue una distribución F con $(k - 1, N - k)$ grados de libertad. La regla de decisión es:

- Si $p > \alpha$ no se rechaza H_0: se asume homogeneidad de varianzas.
- Si $p \leq \alpha$ se rechaza H_0: existe evidencia de heterocedasticidad.

3.2 Estado del Arte

La instrucción de Modelos de Lenguaje Grandes (LLM) y otras redes neuronales profundas (DNN) se topa con un obstáculo importante: la capacidad de almacenamiento en las memorias GPU (Chen et al., 2016; Kirisame et al., 2021). El principal problema radica en la necesidad de almacenar las activaciones intermedias del *forward pass* para calcular los gradientes en el *backward pass* (Chen et al., 2016; Kirisame et al., 2021). En este contexto, la rematerialización de activaciones (también conocida como *gradient checkpointing* o *recomputation*) ha emergido como una técnica clave, que intercambia un costo computacional adicional por una reducción significativa en el consumo de memoria (Kirisame et al., 2021; Zhang et al., 2023).

La idea de descartar resultados intermedios para luego recalcularlos tiene raíces en la literatura de Diferenciación Automática (Chen et al., 2016; Beaumont, Eyraud-Dubois, & Shilova, 2021), pero su aplicación sistemática al entrenamiento de DNN fue popularizada por Chen et al. (2016). Este trabajo demostró que era posible entrenar una red de n capas con un costo de memoria sublineal, $O(\sqrt{n})$, a cambio de un pase adicional hacia adelante por mini-lote. Aunque se trataba de una heurística voraz sin garantía de optimalidad, sentó las bases para romper la barrera de la memoria lineal (Chen et al., 2016).

Posteriormente, la comunidad buscó soluciones óptimas al problema de la rematerialización. *Checkmate* formalizó el problema como un MILP, capaz de encontrar el plan de rematerialización que minimizaba el costo computacional para un presupuesto de memoria dado (Kirisame et al., 2021; Zhang et al., 2023). Sin embargo, su elevado costo de resolución lo hacía impracticable para redes de gran escala (Kirisame et al., 2021; Zhang et al., 2023). A partir de esta limitación, surgieron enfoques más eficientes orientados a reducir la complejidad del

problema: *Moccasin* aplicó programación con restricciones para acelerar la resolución en grafos grandes (Bartan et al., 2023); *XEngine* reformuló el problema como un MIQP en entornos CPU–GPU, logrando cronogramas hasta un 22,5 % más rápidos que *Checkmate* (Schuler, Membarth, & Slusallek, 2022); y *Rockmate* adoptó un enfoque híbrido, combinando *Checkmate* a nivel de bloque con heurísticas rápidas a nivel de secuencia, alcanzando reducciones de memoria de 2–5× y sobrecarga de cómputo de sólo 10–20 % (Zhao et al., 2023). En conjunto, estas propuestas marcaron una transición desde métodos óptimos pero costosos hacia soluciones híbridas más escalables y aplicables en redes de gran tamaño.

En paralelo, surgieron enfoques de planificación dinámica. DTR (*Dynamic Tensor Rematerialization*) propuso un algoritmo en línea que no requiere conocer el grafo completo de antemano. Teóricamente, puede entrenar una red de N capas con un presupuesto de memoria $\Omega(\sqrt{N})$ y $O(N)$ operaciones tensoriales (Kirisame et al., 2021). Su ventaja es manejar grafos dinámicos, aunque su adopción en LLMs ha sido limitada por la sobrecarga de monitoreo en tiempo real (Kirisame et al., 2021).

La investigación también ha abordado problemas más complejos. *Coop* señaló que la fragmentación de memoria era ignorada en trabajos previos, proponiendo una cooptimización de asignación de tensores y rematerialización mediante un algoritmo de ventana deslizante, logrando ahorros de memoria de hasta 2× (Zhang et al., 2023). Por su parte, en 2021 se exploraron la sinergia entre rematerialización y *offloading* hacia CPU, desarrollando un algoritmo de programación dinámica (*pofo*) que redujo el uso de memoria entre 4–6× con sobrecarga inferior al 20 % (Beaumont, Eyraud-Dubois, & Shilova, 2021).

En síntesis, el campo ha alcanzado una notable madurez. Se dispone de formulaciones matemáticas robustas (*Checkmate*, *pofo*) que permiten soluciones óptimas en grafos estáticos

(Zhang et al., 2023; Beaumont, Eyraud-Dubois, & Shilova, 2021), así como heurísticas prácticas (*Rockmate*, *selective checkpointing*) que ofrecen un buen compromiso entre rendimiento y velocidad (Zhao et al., 2023; Kirisame et al., 2021). Además, se han comenzado a abordar problemas de segundo orden como la fragmentación de memoria (*Coop*) y la interacción con técnicas complementarias como el *offloading* (Zhang et al., 2023; Beaumont, Eyraud-Dubois, & Shilova, 2021).

No obstante, persisten limitaciones importantes. En primer lugar, la escalabilidad: los solucionadores óptimos basados en MILP (Zhang et al., 2023), MIQP (Schuler, Membarth, & Slusallek, 2022) o programación dinámica (Kirisame et al., 2021; Beaumont, Eyraud-Dubois, & Shilova, 2021) son demasiado costosos para LLMs de gran escala (Bartan et al., 2023; Beaumont, Eyraud-Dubois, & Shilova, 2021). En segundo lugar, existe una ausencia de modelos predictivos ligeros (Duan et al., 2024), lo que genera una brecha entre solucionadores exactos (precisos pero lentos) y heurísticas simples (rápidas pero subóptimas). En tercer lugar, la falta de protocolos estandarizados de evaluación dificulta la comparación entre propuestas, como indica Duan et al. (2024).

El presente trabajo se posiciona para atender estos vacíos. En primer lugar, propone el desarrollo de un modelo matemático predictivo que relacione la granularidad y ubicación de los puntos de rematerialización con el consumo de memoria y el coste computacional, ofreciendo una herramienta ligera de análisis y planificación (Bartan et al., 2023; Beaumont, Eyraud-Dubois, Herrmann, & Lima, 2024; Chen et al., 2016; Zhao et al., 2023; Schuler, Membarth, & Slusallek, 2022). En segundo lugar, plantea un protocolo experimental reproducible para calibrar y validar el modelo, contribuyendo a la estandarización de métricas, una necesidad destacada en la literatura (Duan et al., 2024). Finalmente, se realizará una

comparación controlada frente a dos estrategias de referencia: entrenamiento sin rematerialización y *Uniform Segment Checkpointing*, lo que permitirá cuantificar de manera clara el beneficio del modelo propuesto.

En contraste con la polarización actual entre soluciones óptimas pero ineficientes y heurísticas rápidas pero limitadas, este proyecto busca ocupar el espacio intermedio mediante un modelo predictivo que combina rigor en la validación con rapidez en la toma de decisiones. De esta manera, no solo se optimiza la aplicación práctica de la rematerialización, sino que también se enriquece el marco teórico, aportando un enfoque equilibrado frente a los extremos que predominan en la literatura.

4. Metodología

La metodología empleada desarrolló el experimento en dos tipo diferentes de GPU disponibles en el supercomputador GUANE, dándonos métricas para entrenar el modelo matemático y validando estadísticamente para analizar el trade-off cómputo–memoria en el entrenamiento de redes neuronales con rematerialización de activaciones. El proceso se desarrolló en cinco fases consecutivas que abarcan desde la definición de métricas hasta el análisis de resultados.

4.1 Definición de Métricas e Indicadores

El primer componente metodológico consistió en establecer un conjunto de métricas que permitieran caracterizar de manera precisa el *trade-off* cómputo–memoria en el entrenamiento de redes neuronales. Para ello, se realizó una revisión bibliográfica que identifica los indicadores más relevantes en la literatura reciente. Se contemplaron métricas de memoria como el pico de

uso, la memoria promedio y el ahorro porcentual de desempeño, tiempo por iteración, *throughput* y tiempo de resolución, así como de costo adicional factor de recomputación. Estas métricas se obtuvieron mediante el uso de PyTorch Profiler (Smith, 2021), mientras que su análisis y visualización se apoyó en matplotlib y seaborn, lo que permitió disponer de un marco de evaluación consistente y reproducible.

4.2 Diseño Experimental y Construcción del Modelo Predictivo

El diseño experimental constituye el núcleo metodológico del proyecto, ya que de él se derivan los datos necesarios para entrenar y validar el modelo predictivo. La estrategia combina la selección de arquitecturas representativas, la definición de configuraciones experimentales, la recolección sistemática de métricas y la construcción de un dataset estructurado que sirvió como insumo para los algoritmos de aprendizaje automático.

4.2.1 Selección de Arquitecturas de Referencia

Se trabajó con seis modelos ampliamente reconocidos en la literatura y con implementaciones disponibles en PyTorch: tres MLP (MLP-3, MLP-5 y MLP-7) y tres CNN (LeNet-5, AlexNet y VGG-16). Estas arquitecturas fueron elegidas porque representan distintos niveles de complejidad: desde redes pequeñas y poco profundas, hasta modelos medianos y grandes que demandan un uso intensivo de memoria. Esta diversidad permitió evaluar el impacto de la rematerialización en escenarios contrastantes y garantizar que el modelo predictivo capture patrones generales y no casos aislados.

4.2.2 Definición de Configuraciones Experimentales

Cada arquitectura se entrenó bajo un conjunto de variaciones controladas en sus parámetros de configuración. Se consideraron las siguientes dimensiones:

- Número de capas y neuronas/filtros: variando la profundidad y el ancho de la red para modificar la complejidad computacional.
- Funciones de activación: ReLU, Leaky ReLU, Tanh y Sigmoide, con el fin de observar cómo la no linealidad influye en el consumo de memoria y el tiempo de cómputo.
- Tamaño de lote: 16, 32, 64 y 128, lo que permitió analizar el efecto directo del *batch size* sobre el pico de memoria y el *throughput*.
- Granularidad y ubicación de *checkpoints*: desde configuraciones uniformes hasta segmentaciones más finas, con el objetivo de capturar el efecto de la rematerialización en distintos niveles de recomputación.

4.2.3 Recolección de Métricas y Construcción del Dataset

Cada configuración experimental se ejecutó en GPU, registrando métricas de tiempo (latencia por iteración, throughput), memoria (pico de uso, promedio, ahorro porcentual) y costo adicional (factor de recomputación). Estas mediciones se obtuvieron con PyTorch Profiler. Los resultados se almacenaron en un dataset estructurado en formato tabular, donde cada fila representa una configuración experimental y cada columna corresponde a una variable independiente asociada a la arquitectura del modelo y a la configuración de entrenamiento, o una variable dependiente correspondiente a métricas de tiempo y memoria.

Los experimentos se ejecutaron en los servidores de supercomputación del SC3, utilizando nodos equipados con GPU de NVIDIA GeForce Titan X y AMD MI210. Sobre cada una de estas tarjetas gráficas se entrenaron todos los modelos que fueron considerados en el diseño experimental, extrayendo en cada caso los resultados del perfilado mediante PyTorch

Profiler. La ejecución se organizó mediante una serie de jobs lanzados secuencialmente, lo que permitió cubrir de forma sistemática todas las configuraciones definidas y obtener así el dataset completo requerido para el entrenamiento del modelo predictivo. El dataset resultante, junto con los scripts de ejecución y los registros de perfilado, se encuentra disponible en el repositorio de GitHub del proyecto como evidencia del proceso experimental realizado.

4.2.4 Construcción de los Modelos Predictivos y su Entrenamiento

Con el fin de mantener la claridad y concisión del documento, el código fuente y los detalles de implementación del modelo predictivo no se presentan de manera íntegra en esta sección. En su lugar, se ha dispuesto un repositorio en GitHub que contiene el notebook completo con la construcción, entrenamiento y validación de los modelos propuestos.

Durante la etapa de experimentación y ejecución de los modelos, se observó que la regresión lineal múltiple no lograba capturar adecuadamente el comportamiento de los datos. Esto se evidenció en su bajo desempeño, ya que asume relaciones estrictamente lineales y es sensible a la multicolinealidad presente en el conjunto de variables, afectando la estabilidad y precisión de las predicciones. Ante esta situación, se optó por utilizar el SGD Regressor, que mantiene un enfoque lineal pero incorpora regularización (Elastic Net) y un entrenamiento más eficiente mediante descenso de gradiente estocástico, permitiendo obtener un modelo más robusto y con mejor capacidad de generalización.

El SGD Regressor se implementa dentro de un pipeline que primero aplica la estandarización de los datos y luego utiliza MultiOutputRegressor para predecir simultáneamente

memoria y tiempo. Este modelo se entrena actualizando los pesos de forma iterativa con subconjuntos de datos, utilizando una función de pérdida cuadrática y regularización Elastic Net, lo que ayuda a controlar el sobreajuste. Además, incorpora early_stopping y una tasa de aprendizaje adaptativa para optimizar la convergencia durante el entrenamiento.

Por otro lado, el Random Forest Regressor se construye como un ensamble de múltiples árboles de decisión, donde cada uno se entrena con subconjuntos aleatorios de datos y variables. Este enfoque reduce la varianza y mejora la generalización, permitiendo capturar relaciones no lineales e interacciones complejas entre las variables. A diferencia del SGD, no requiere normalización de datos, y su configuración ajusta el número de árboles, la profundidad máxima y el tamaño mínimo de las divisiones, buscando un equilibrio entre precisión y control del sobreajuste mediante validación.

Sobre el dataset obtenido se entrenaron los dos modelos supervisados:

- SGD Regressor, utilizada como línea base por su interpretabilidad y capacidad de mostrar relaciones directas entre variables.
- *Random Forest*, seleccionada por su robustez frente a ruido experimental y su capacidad de capturar interacciones no lineales entre parámetros.

El preprocesamiento incluyó la normalización de variables, la detección y eliminación de valores atípicos y la división del dataset en subconjuntos de entrenamiento (70 %), validación (15 %) y prueba (15 %). La implementación se realizó en *scikit-learn-intelex* (Pedregosa et al., 2011), lo que permitió aprovechar optimizaciones específicas para CPU Intel en la fase de entrenamiento.

4.2.5 Objetivo de la Fase de Modelado

El resultado esperado fue un modelo predictivo capaz de estimar, a partir de la configuración de la red (arquitectura, tamaño de lote, ubicación de checkpoints), el consumo de memoria y el coste computacional asociado. De esta manera, se buscó disponer de una herramienta ligera que permitiera anticipar el impacto de la rematerialización sin necesidad de ejecutar exhaustivos experimentos de perfilado en cada escenario.

4.3 Calibración y validación del modelo

La validación del modelo predictivo se llevará a cabo mediante métricas de error como MAE, RMSE y R^2, que permitirán cuantificar la precisión de las predicciones. Una vez calibrado, los modelos se utilizarán para sugerir configuraciones óptimas de rematerialización (topología, granularidad, ubicación de checkpoints y tamaño de lote). Estas configuraciones se contrastaron con dos estrategias de referencia: el entrenamiento sin rematerialización y el Uniform Segment Checkpointing.

Los experimentos se ejecutarán en el supercomputador GUANE (UIS), con un máximo de diez epochs por configuración, lo que permitirá capturar patrones estables sin incurrir en costos excesivos. Para garantizar la validez estadística de los resultados, se aplicará un protocolo de verificación que incluye la prueba de Levene para homogeneidad de varianzas, análisis de varianza (ANOVA) con un nivel de confianza del 95% y, en caso de significancia, pruebas post-hoc Tukey HSD. Cuando los supuestos de normalidad o homogeneidad no se cumplan, se recurrirá a la prueba no paramétrica de Kruskal–Wallis, que permite comparar medianas sin asumir distribuciones normales.

5. Resultados

5.1 Descripción del Dataset Experimental obtenido y Preprocesamiento

El dataset experimental se obtuvo mediante perfilado en GPU con PyTorch Profiler en el supercomputador GUANE de la Unidad SC3–VIE de la Universidad Industrial de Santander. En esta fase experimental se emplearon dos arquitecturas de unidades de procesamiento gráfico con características marcadamente distintas, lo que permitió analizar el comportamiento del trade-off cómputo–memoria en escenarios contrastantes.

La primera es la AMD Instinct MI210, un acelerador orientado a entornos de alto rendimiento (HPC) y cargas de inteligencia artificial. Posee arquitectura CDNA2, diseñada para el cómputo científico y el entrenamiento de grandes modelos. Cuenta con 64 GB de memoria HBM2e, 6.656 núcleos de procesamiento y una frecuencia boost de hasta 1.700 MHz.

La segunda GPU es la NVIDIA GeForce GTX TITAN X, basada en la arquitectura Maxwell. Esta tarjeta incorpora 3.072 CUDA Cores, ampliamente utilizados en frameworks de aprendizaje profundo como PyTorch. Dispone de 12 GB de memoria GDDR5, suficiente para cargas de entrenamiento medianas, aunque con menor capacidad y ancho de banda que la MI210; su frecuencia boost alcanza aproximadamente 1.706 MHz.

Sobre estas dos tarjetas gráficas se ejecutaron 2.064 configuraciones de entrenamiento distribuidas entre las seis arquitecturas de referencia definidas en la metodología (MLP-3, MLP-5, MLP-7, LeNet-5, AlexNet y VGG-16). Cada configuración combina una arquitectura, una función de activación (ReLU, Leaky ReLU, Tanh o Sigmoid), un tamaño de lote (16, 32, 64 o 128) y una estrategia de rematerialización concreta, acompañada de las métricas de memoria,

tiempo y exactitud resultantes. De este total, 160 configuraciones correspondieron a baselines de uniform segment checkpointing (uniform_seg2 y uniform_seg3) que se reservaron para la fase de evaluación de estrategias, mientras que las 1.904 restantes —que combinan únicamente las estrategias sin_rematerialización y checkpoint_selectivo— se emplearon para entrenar y validar los modelos predictivos. Tras el preprocesamiento se obtuvieron 73 variables independientes y dos variables objetivo (memory_avg_MB y total_time_ms), conformando la matriz de entrada de los modelos.

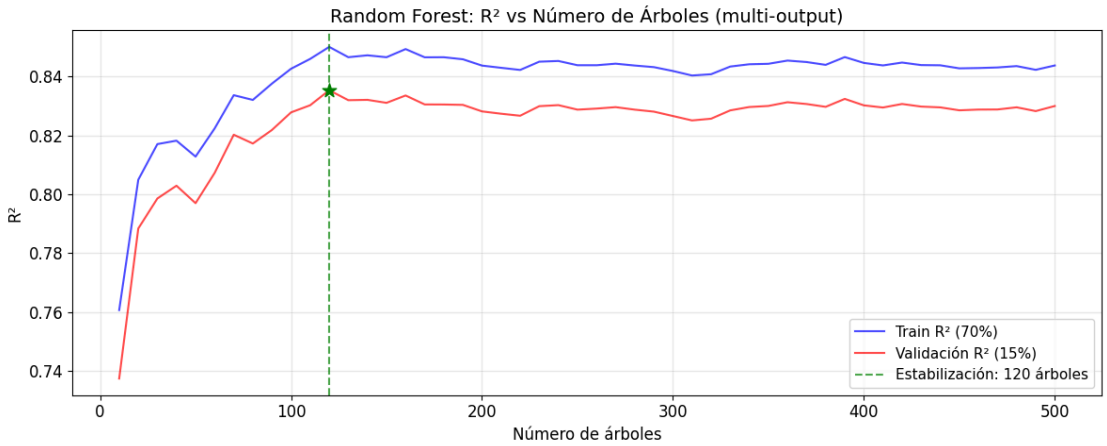
Dada la presencia de valores extremos asociados a configuraciones reales de modelos grandes (VGG-16 con batch = 128), se optó por no eliminar outliers: los valores extremos no son errores de medición, sino el segmento del espacio experimental donde la rematerialización resulta más pertinente. El subconjunto de entrenamiento (1.904 filas) se particionó aleatoriamente con semilla fija (random_state = 42) en tres bloques: entrenamiento (70 %, n = 1.332), validación (15 %, n = 286) y prueba (15 %, n = 286). La partición se realizó con shuffle = True para evitar que el orden original del dataset (ordenado por modelo) introdujera sesgo en la evaluación.

5.2 Convergencia de los Modelos Predictivos

Las curvas de convergencia confirman que ambos modelos alcanzan un régimen estable dentro del rango explorado. El Random Forest (Figura 5) muestra un crecimiento monótono del coeficiente de determinación en validación hasta estabilizarse alrededor de 120 árboles, con una brecha train–validación acotada que indica control adecuado del sobreajuste ($R^2 \approx 0{,}84$ en train y $R^2 \approx 0{,}83$ en validación). Este comportamiento es coherente con la teoría del bagging, según la cual el promedio de estimadores independientes reduce la varianza sin afectar significativamente el sesgo.

Figura 5

Curva de Convergencia del Random Forest: R² vs. Número de Árboles

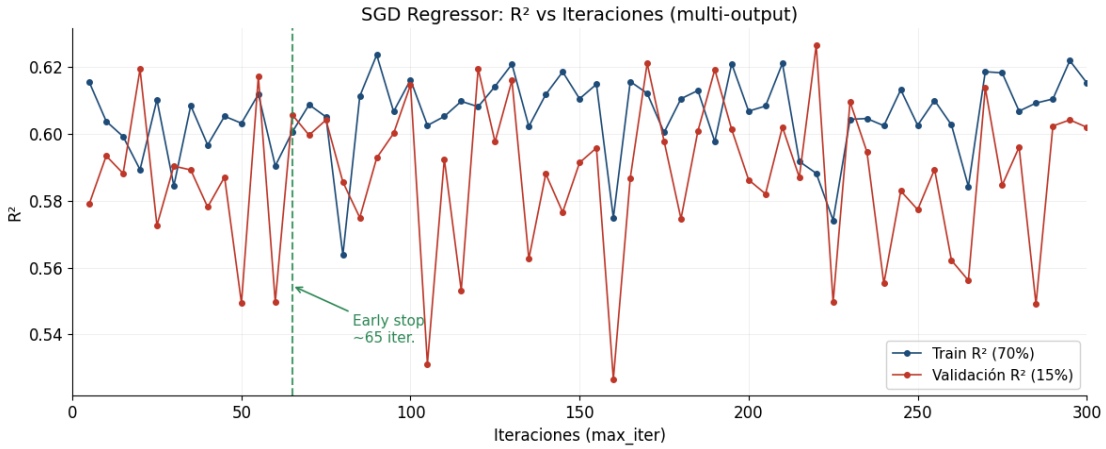


Nota. Las curvas muestran R² del Random Forest evaluado sobre el conjunto de entrenamiento (azul) y el conjunto de validación (rojo) al variar el número de árboles. La línea punteada verde marca la estabilización en 120 árboles. Elaboración propia a partir de las ejecuciones del notebook.

Por su parte, el SGD Regressor (Figura 6) presenta oscilaciones notables en sus curvas de entrenamiento y validación, comportamiento característico del carácter estocástico del algoritmo, donde cada iteración introduce ruido al actualizar los pesos con subconjuntos aleatorios de datos. Ambas curvas se mantienen en un rango similar (≈ 0,55–0,62), lo que sugiere ausencia de sobreajuste significativo, ya que el modelo no llega a memorizar el conjunto de entrenamiento. Asimismo, su desempeño medido mediante R² resulta inferior al de Random Forest, lo cual evidencia que la relación entre las variables de configuración y las métricas del trade-off no es estrictamente lineal, sino que involucra interacciones complejas que un modelo lineal no logra capturar. El criterio de detención temprana se activa alrededor de la iteración 65, indicando que el modelo alcanza rápidamente su capacidad de aprendizaje y que iteraciones adicionales no generan mejoras sustanciales.

Figura 6

Curva de Convergencia del SGD Regressor: R² vs. Número de Iteraciones



Nota. R² del SGD Regressor con early stopping. La línea punteada verde indica el corte temprano cercano a la iteración 65. Elaboración propia a partir de las ejecuciones del notebook.

5.3 Calibración del Modelo Predictivo

Los modelos calibrados se evaluaron sobre el conjunto de prueba (n = 286) con las métricas MAE, RMSE y R² en la escala original (MB y ms). La Tabla 1 resume el desempeño obtenido para las dos variables objetivo, centrando la comparación en el R². Adicionalmente, el R² global multi-output sobre el conjunto de prueba fue de 0,6800 para el SGD Regressor y de 0,8462 para el Random Forest, mientras que el R² medio en validación cruzada de 5 pliegues sobre el conjunto de entrenamiento fue de 0,5970 ± 0,0257 para el SGD y de 0,8168 para el Random Forest tras el ajuste de hiperparámetros.

Tabla 1

Desempeño de los Modelos en el Conjunto de Prueba (n = 286)

Modelo	Variable objetivo	R²	MAE	RMSE
SGD Regressor	memory_avg_MB (MB)	0,7974	335,15	421,14
Random Forest	memory_avg_MB (MB)	0,8330	239,94	382,40
SGD Regressor	total_time_ms (ms)	0,5626	7.769,08	12.525,96
Random Forest	total_time_ms (ms)	0,8594	3.029,95	7.100,74

Nota. R^2 es adimensional; MAE y RMSE se expresan en la unidad nativa de cada variable. Los valores sobresalientes del R^2 indican el mejor modelo para cada objetivo. Elaboración propia a partir de las ejecuciones del notebook.

Con base en las características teóricas del SGD Regressor, este modelo representa relaciones esencialmente lineales, por lo que su capacidad para capturar interacciones complejas entre variables resulta limitada. En el contexto del trade-off cómputo–memoria, la rematerialización de activaciones introduce comportamientos no lineales derivados de la interacción entre arquitectura de red, tamaño de lote, profundidad y ubicación de checkpoints.

Para la variable memory_avg_MB, el Random Forest supera al SGD tanto en R^2 (0,8330 frente a 0,7974) como en MAE (239,94 MB frente a 335,15 MB) y en RMSE (382,40 MB frente a 421,14 MB). Esto indica que, si bien la memoria mantiene una componente parcialmente lineal con respecto a las variables de entrada, el Random Forest aprovecha mejor las interacciones entre arquitectura, batch y capas marcadas como checkpoint para reducir tanto el error promedio como los errores extremos.

En cuanto a la variable total_time_ms, la diferencia es aún más marcada: el Random Forest alcanza $R^2 = 0{,}8594$ frente a $R^2 = 0{,}5626$ del SGD, y reduce el MAE de 7.769,08 ms a 3.029,95 ms y el RMSE de 12.525,96 ms a 7.100,74 ms. Este comportamiento confirma que el tiempo total de entrenamiento depende de relaciones altamente no lineales e interacciones complejas entre los parámetros experimentales tales como la sobrecarga por recomputación, la topología de la red y la utilización del hardware, aspectos que son modelados con mayor eficacia por métodos basados en árboles de decisión.

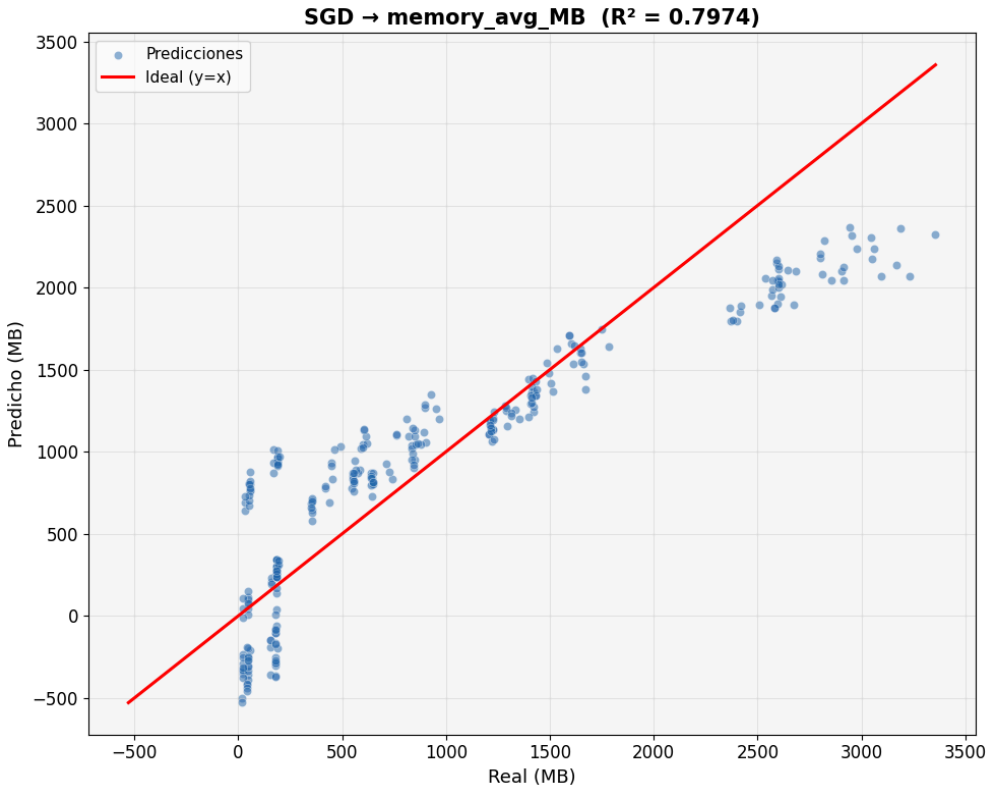
En términos generales, el Random Forest presenta el mejor desempeño global tanto en la estimación de memoria como, especialmente, en la predicción del costo computacional. Por ello,

se considera el modelo más adecuado para representar y optimizar el trade-off cómputo–memoria planteado en esta investigación.

Las Figuras 7 a 10 ilustran la calidad de las predicciones mediante diagramas de calibración predicho-vs-real. Cada punto corresponde a una configuración del conjunto de prueba y la línea roja representa el ideal y = x: a mayor proximidad de la nube respecto a esa línea, mejor la calibración.

Figura 7

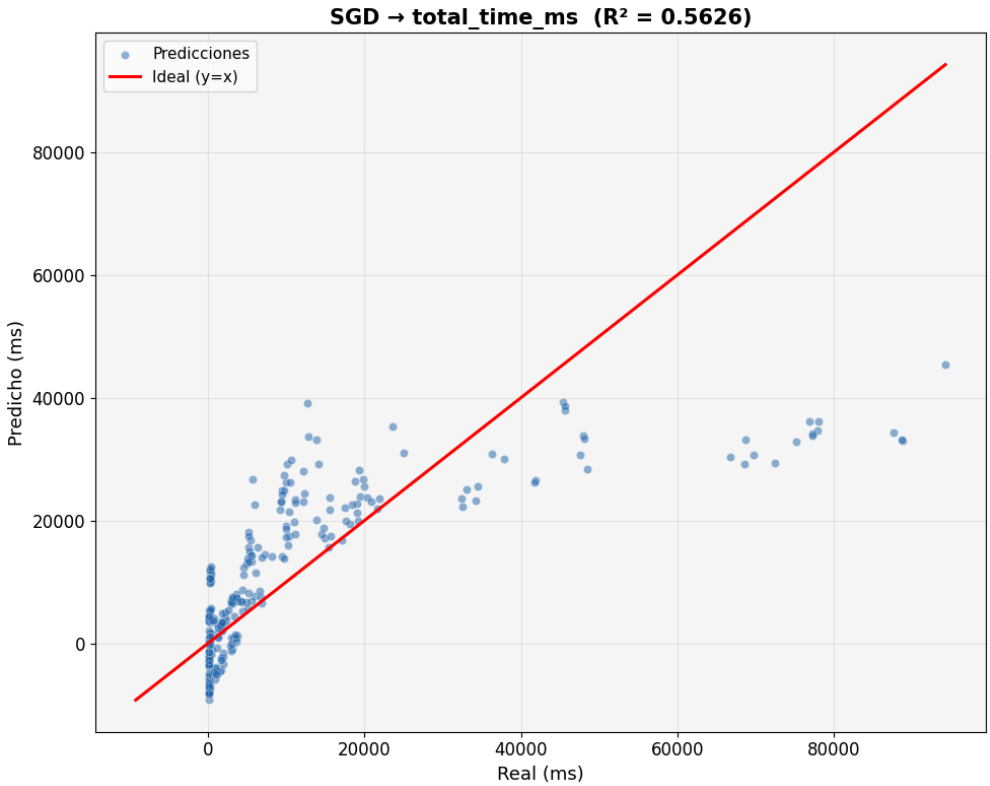
Calibración (Valor Predicho vs. Valor Real) del SGD Regressor para memoria promedio



Nota. Diagrama predicho vs. real del SGD Regressor sobre el conjunto de prueba para la variable memory_avg_MB (R^2 = 0,7974). Elaboración propia a partir de las ejecuciones del notebook.

Figura 8

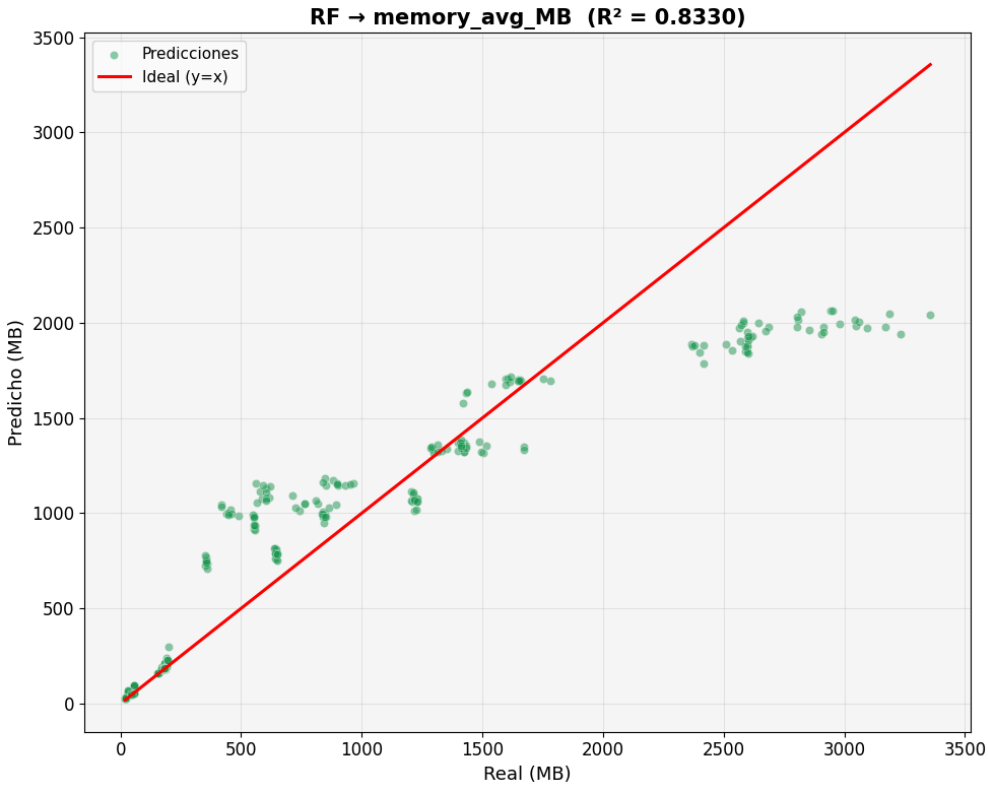
Calibración (Valor Predicho vs. Valor Real) del SGD Regressor para tiempo total de entrenamiento



Nota. Diagrama predicho vs. real del SGD Regressor sobre el conjunto de prueba para la variable total_time_ms ($R^2 = 0{,}5626$). Elaboración propia a partir de las ejecuciones del notebook.

Figura 9

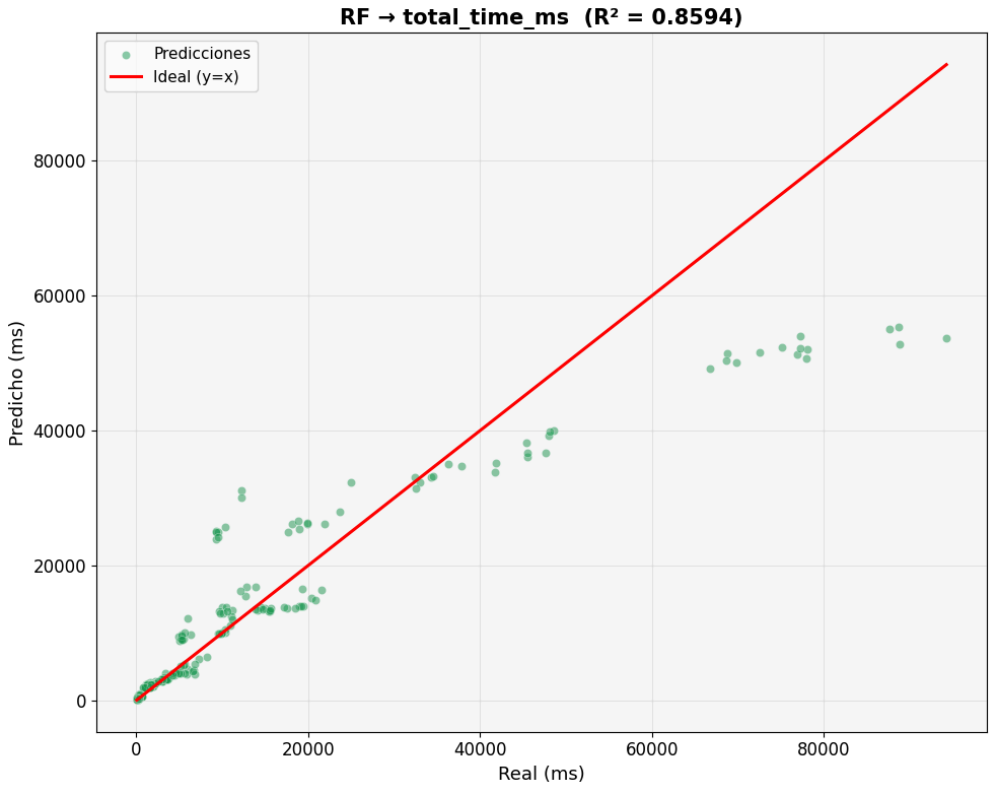
Calibración (Valor Predicho vs. Valor Real) del Random Forest para memoria promedio



Nota. Diagrama predicho vs. real del Random Forest sobre el conjunto de prueba para la variable memory_avg_MB ($R^2 = 0{,}8330$). Elaboración propia a partir de las ejecuciones del notebook.

Figura 10

Calibración (Valor Predicho vs. Valor Real) del Random Forest para tiempo total de entrenamiento



Nota. Diagrama predicho vs. real del Random Forest sobre el conjunto de prueba para la variable total_time_ms ($R^2 = 0{,}8594$). Elaboración propia a partir de las ejecuciones del notebook.

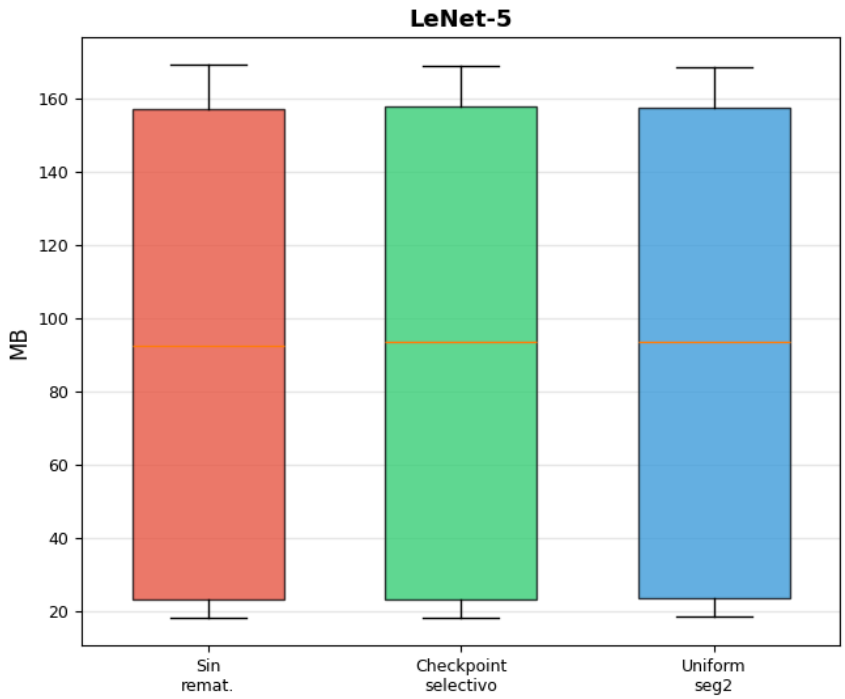
El Random Forest concentra los puntos sobre la diagonal en ambas métricas, con menor dispersión en los rangos altos de memoria y tiempo donde precisamente la rematerialización resulta más relevante. El SGD tiende a comprimir los extremos, subestimando los valores altos de memoria y tiempo, lo cual es consistente con su menor R^2 en la variable total_time_ms.

5.4 Comparación entre Estrategias de Rematerialización

Las configuraciones CNN se clasificaron en cuatro estrategias: sin rematerialización, checkpoint selectivo, uniform segment con 2 segmentos (uniform_seg2) y uniform segment con 3 segmentos (uniform_seg3). LeNet-5 no fue evaluado bajo uniform_seg3 dada su escasa profundidad, por lo que en ese modelo se reportan únicamente tres estrategias. Las Figuras 11 a 16 presentan la distribución de ambas variables objetivo por modelo y por estrategia.

Figura 11

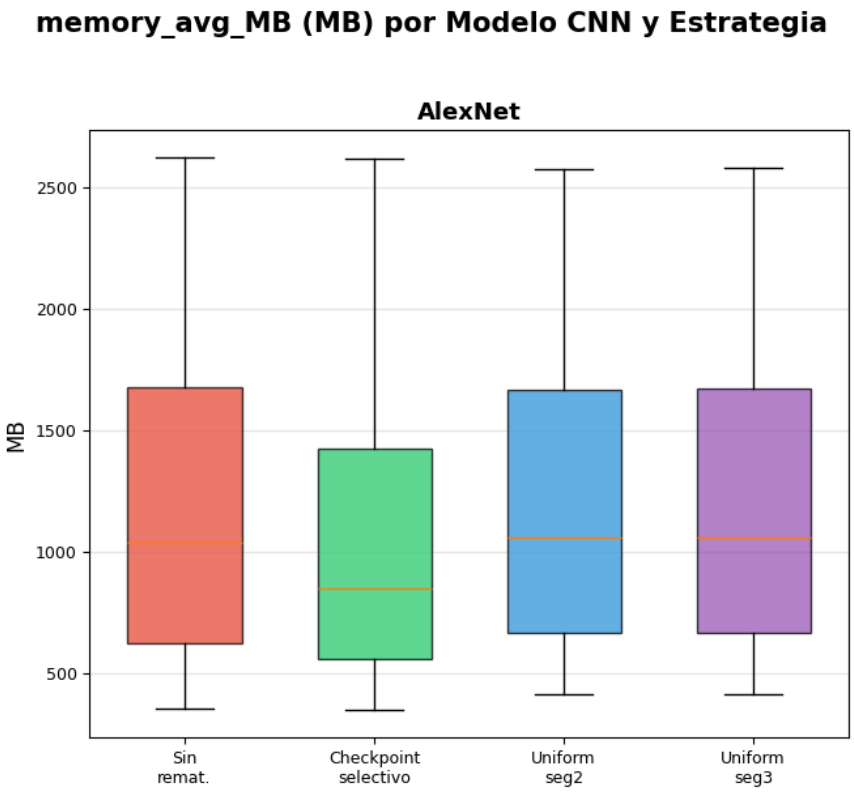
Comparación de estrategias de checkpointing en el modelo LeNet-5: consumo de memoria



Nota. Distribución de memory_avg_MB para LeNet-5 según estrategia. Elaboración propia a partir de las ejecuciones del notebook.

Figura 12

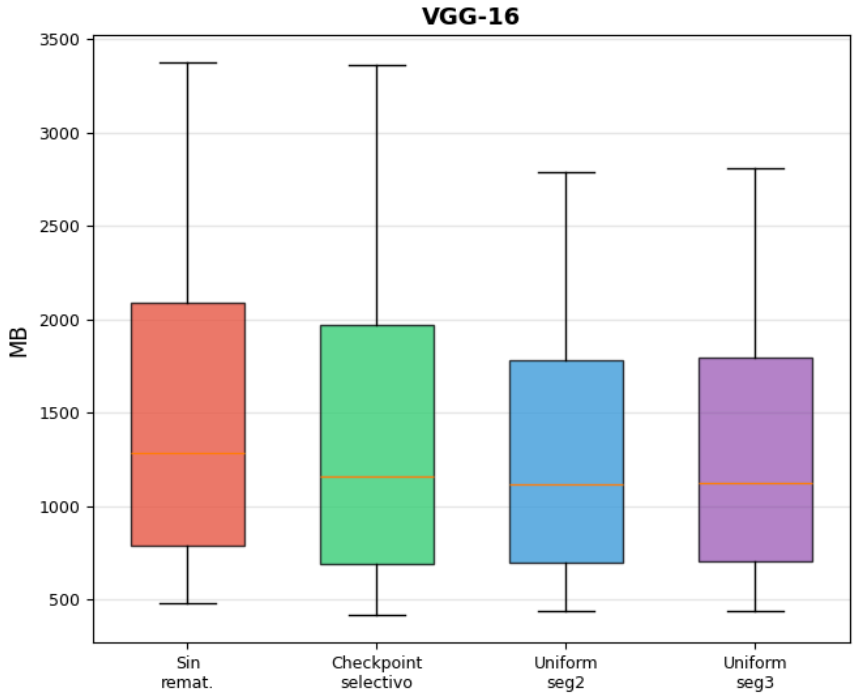
Comparación de estrategias de checkpointing en el modelo AlexNet: consumo de memoria



Nota. Distribución de memory_avg_MB para AlexNet según estrategia. Elaboración propia a partir de las ejecuciones del notebook.

Figura 13

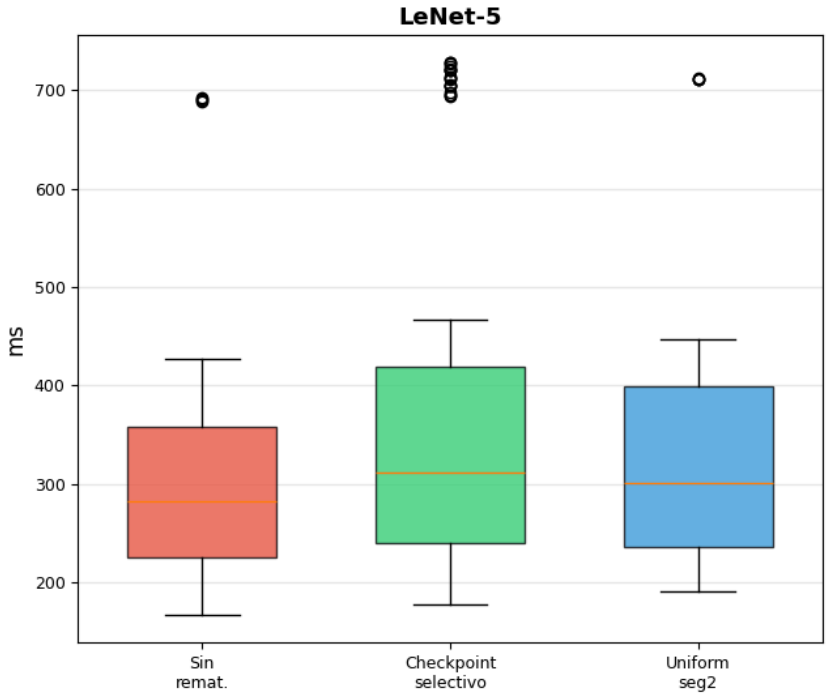
Comparación de estrategias de checkpointing en el modelo VGG-16: consumo de memoria



Nota. Distribución de memory_avg_MB para VGG-16 según estrategia. Elaboración propia a partir de las ejecuciones del notebook.

Figura 14

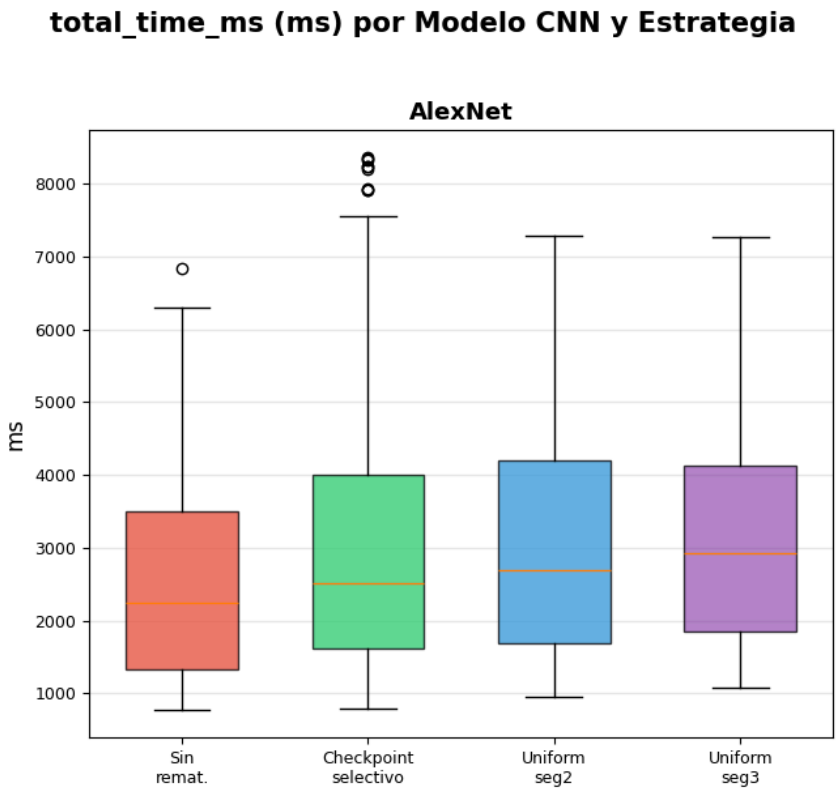
Comparación de estrategias de checkpointing en el modelo LeNet-5: tiempo de resolución



Nota. Distribución de total_time_ms para LeNet-5 según estrategia. Elaboración propia a partir de las ejecuciones del notebook.

Figura 15

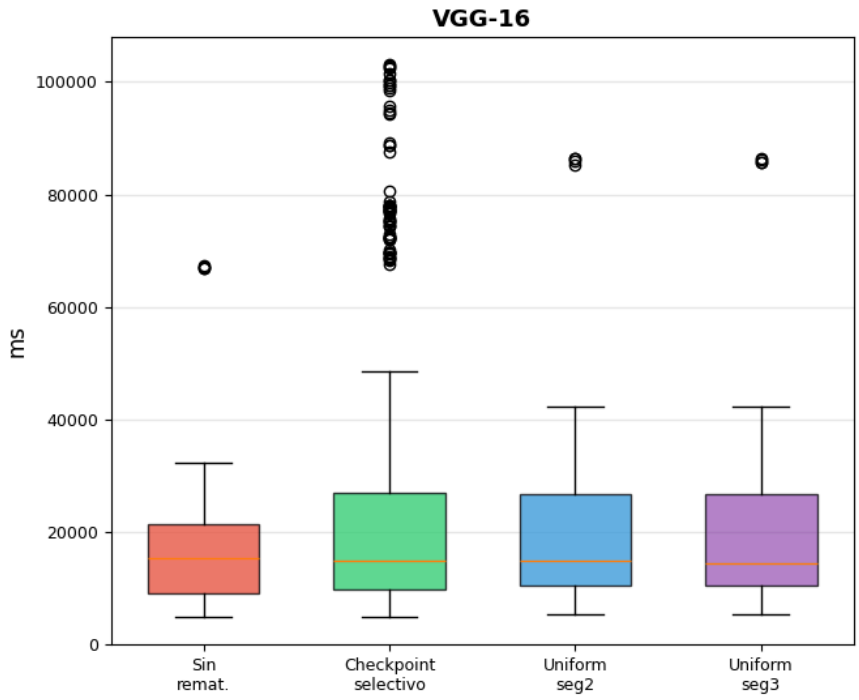
Comparación de estrategias de checkpointing en el modelo AlexNet: tiempo de resolución



Nota. Distribución de total_time_ms para AlexNet según estrategia. Elaboración propia a partir de las ejecuciones del notebook.

Figura 16

Comparación de estrategias de checkpointing en el modelo VGG-16: tiempo de resolución



Nota. Distribución de total_time_ms para VGG-16 según estrategia. Elaboración propia a partir de las ejecuciones del notebook.

Los boxplots permiten visualizar la distribución de las variables objetivo (memory_avg_MB y total_time_ms) para cada estrategia: sin rematerialización (rojo), checkpoint selectivo (verde), uniform_seg2 (azul) y uniform_seg3 (morado), en tres paneles correspondientes a los modelos LeNet-5, AlexNet y VGG-16. La representación por modelo muestra un patrón escalonado según la profundidad de la red.

En LeNet-5 las tres cajas son prácticamente idénticas en memoria, coherente con el reducido presupuesto de activaciones sobre el que puede actuar un modelo tan pequeño; en tiempo de resolución, las estrategias de rematerialización son ligeramente más altas, siendo uniform_seg2 la más lenta.

En AlexNet comienza a observarse una separación más evidente: la caja de checkpoint selectivo desplaza la mediana de memoria por debajo del baseline (sin rematerialización), mientras que las variantes uniform se superponen casi por completo con el baseline en términos de memoria. En tiempo, todas las variantes de rematerialización son más lentas que el baseline, con uniform_seg3 mostrando la mayor mediana.

En VGG-16 las variantes uniform_seg2 y uniform_seg3 consumen claramente menos memoria que el checkpoint selectivo y que el baseline sin rematerialización, evidenciando el mayor ahorro relativo entre todas las arquitecturas. En cuanto al tiempo, las variantes con rematerialización son más lentas que el baseline, con uniform_seg3 a la cabeza, como cabe esperar por su mayor factor de recomputación.

El checkpoint selectivo solo aplica rematerialización en capas específicas, por lo que reduce memoria de forma moderada y añade menor costo en tiempo. En cambio, uniform_seg2 y uniform_seg3 dividen toda la red en segmentos regulares y guardan menos activaciones en conjunto, logrando mayor ahorro de memoria, pero con más recomputación y mayor tiempo de ejecución.

5.5 Análisis de los Resultados de las Pruebas Estadísticas

Para evaluar si las diferencias observadas entre estrategias de rematerialización son estadísticamente significativas, se aplicó un protocolo de tres pasos: Levene → ANOVA o Kruskal–Wallis → Tukey HSD, con un nivel de significancia $\alpha = 0{,}05$. El análisis se restringe a los modelos CNN (LeNet-5, AlexNet y VGG-16), donde las estrategias de rematerialización están bien definidas y la comparación tiene sentido físico; en los MLP el uniform segment específico por arquitectura no aplica y el ahorro de memoria es despreciable por construcción.

A diferencia de un análisis sobre valores absolutos donde la heterogeneidad de los modelos domina la varianza, el protocolo se ejecutó sobre métricas relativas pareadas por escenario: el ahorro porcentual de memoria respecto a la baseline y el overhead porcentual de tiempo respecto a la baseline, sobre n = 95 escenarios CNN emparejados por estrategia. Esto controla el efecto de tamaño del modelo y aísla la contribución de la estrategia. Las tablas siguientes presentan, primero, las estadísticas descriptivas por estrategia y el resultado del test de Levene; luego, el ANOVA o Kruskal–Wallis con la validación post-hoc mediante Tukey HSD; y, finalmente, un resumen consolidado.

Tabla 2

Estadísticas descriptivas por estrategia y prueba de Levene (homogeneidad de varianzas) sobre el subconjunto CNN (métricas pareadas, n = 95).

Variable	Estrategia	n	Media	Desv. estándar	Mediana	Levene (W)	Levene (p)	Varianzas homogéneas
memory_avg_MB	sin_rematerialización	95	+0,000	0,000	+0,000	80,2396	$2{,}60 \times 10^{-28}$	No
memory_avg_MB	checkpoint_selectivo	95	−0,034	1,865	+0,126	80,2396	$2{,}60 \times 10^{-28}$	No
memory_avg_MB	uniform_segment	95	+2,309	7,662	+0,202	80,2396	$2{,}60 \times 10^{-28}$	No
total_time_ms	sin_rematerialización	95	+0,000	0,000	+0,000	80,0469	$2{,}93 \times 10^{-28}$	No
total_time_ms	checkpoint_selectivo	95	+2,226	9,022	+3,226	80,0469	$2{,}93 \times 10^{-28}$	No
total_time_ms	uniform_segment	95	+17,376	14,069	+13,018	80,0469	$2{,}93 \times 10^{-28}$	No

Nota. Métricas relativas pareadas por escenario CNN: ahorro de memoria como (baseline − estrategia) / baseline × 100 y overhead de tiempo como (estrategia − baseline) / baseline × 100. Elaboración propia a partir de las ejecuciones del notebook.

La prueba de Levene rechaza la homogeneidad de varianzas tanto para el ahorro de memoria como para el overhead de tiempo ($p < 0{,}001$ en ambos casos). Esto invalida el supuesto requerido por el ANOVA paramétrico, por lo que se utiliza la prueba no paramétrica de

Kruskal–Wallis para comparar las tres estrategias en cada variable y, posteriormente, Tukey HSD como prueba post-hoc por pares.

Tabla 3

Kruskal–Wallis y post-hoc Tukey HSD por pares de estrategias sobre el subconjunto CNN (α = 0,05).

Variable	Prueba	Estadístico	p-valor	Significativo	Par comparado (Tukey)	Diferencia de medias	IC 95 % inferior	IC 95 % superior	Rechaza H_0
memory_avg_MB	Kruskal–Wallis	H = 49,1043	$2{,}17 \times 10^{-11}$	Sí	checkpoint_selectivo vs sin_rematerialización	+0,0342	−1,5305	+1,5989	No
memory_avg_MB	Kruskal–Wallis	H = 49,1043	$2{,}17 \times 10^{-11}$	Sí	uniform_segment vs checkpoint_selectivo	+2,3429	+0,7782	+3,9076	Sí
memory_avg_MB	Kruskal–Wallis	H = 49,1043	$2{,}17 \times 10^{-11}$	Sí	uniform_segment vs sin_rematerialización	+2,3087	+0,7440	+3,8734	Sí
total_time_ms	Kruskal–Wallis	H = 156,2221	$1{,}19 \times 10^{-34}$	Sí	checkpoint_selectivo vs sin_rematerialización	−2,2262	−5,5425	+1,0901	No
total_time_ms	Kruskal–Wallis	H = 156,2221	$1{,}19 \times 10^{-34}$	Sí	uniform_segment vs checkpoint_selectivo	+15,1495	+11,8332	+18,4658	Sí
total_time_ms	Kruskal–Wallis	H = 156,2221	$1{,}19 \times 10^{-34}$	Sí	uniform_segment vs sin_rematerialización	+17,3758	+14,0595	+20,6921	Sí

Nota. Estadístico H de Kruskal–Wallis y comparaciones por pares con corrección de Tukey HSD. IC = intervalo de confianza al 95 %. Elaboración propia a partir de las ejecuciones del notebook.

Los resultados indican que la elección de estrategia tiene un efecto significativo tanto sobre el ahorro de memoria ($H = 49{,}10$; $p \approx 2 \times 10^{-11}$) como sobre el overhead de tiempo ($H = 156{,}22$; $p \approx 1 \times 10^{-34}$). Las comparaciones post-hoc revelan dos hallazgos clave. En memoria, uniform_segment se separa de forma significativa tanto del baseline sin rematerialización (+2,31 % de ahorro adicional) como del checkpoint selectivo (+2,34 % de ahorro adicional), mientras que checkpoint_selectivo no se distingue estadísticamente del baseline en el conjunto agregado. En tiempo, uniform_segment introduce un overhead promedio adicional altamente significativo respecto a las otras dos estrategias (+15,1 % frente al selectivo y +17,4 % frente al baseline), mientras que checkpoint_selectivo no se distingue estadísticamente del baseline en términos de tiempo. Este patrón es consistente con la naturaleza de las dos técnicas: uniform_segment

maximiza el ahorro de memoria a costa de un overhead temporal claro, mientras que checkpoint_selectivo es prácticamente neutro en tiempo y obtiene un ahorro de memoria selectivo y configuración-dependiente.

Tabla 4

Resumen consolidado de las pruebas estadísticas aplicadas a cada variable objetivo sobre el subconjunto CNN.

Variable	Levene (p)	Varianzas homogéneas	Prueba utilizada	Estadístico	p-valor	Significativo
memory_avg_MB	$2{,}60 \times 10^{-28}$	No	Kruskal–Wallis	H = 49,1043	$2{,}17 \times 10^{-11}$	Sí
total_time_ms	$2{,}93 \times 10^{-28}$	No	Kruskal–Wallis	H = 156,2221	$1{,}19 \times 10^{-34}$	Sí

Nota. Resumen del protocolo Levene → Kruskal–Wallis → Tukey HSD. Elaboración propia a partir de las ejecuciones del notebook.

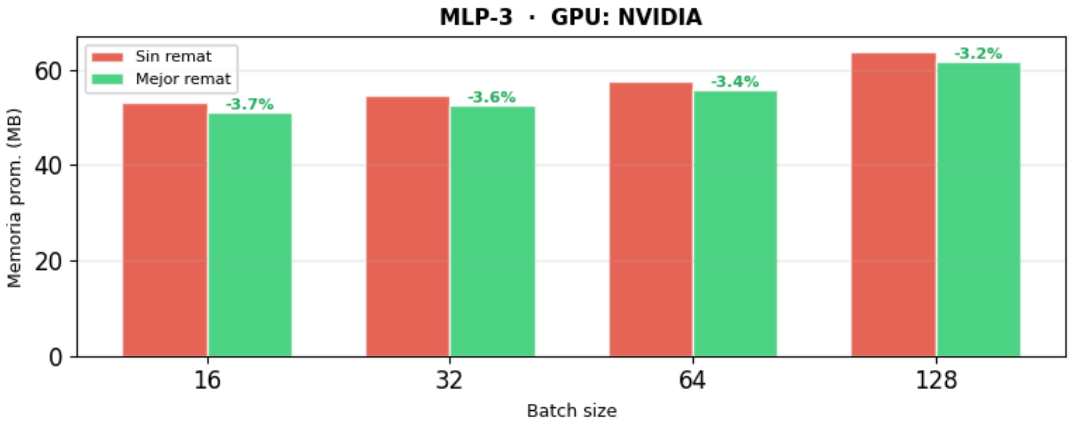
En síntesis, la estrategia de rematerialización afecta de forma significativa tanto al consumo relativo de memoria como al tiempo relativo de entrenamiento. La rematerialización es eficaz para liberar memoria particularmente en la variante uniform_segment y, al mismo tiempo, introduce un costo en tiempo cuya magnitud depende fuertemente de la variante elegida: el checkpoint selectivo es prácticamente neutro frente al baseline, mientras que uniform_segment paga un overhead claro a cambio de su mayor ahorro de memoria.

5.6 Impacto del Tamaño de Lote sobre la Memoria

Las Figuras 17 a 28 comparan, para cada combinación de (modelo, GPU, batch), el consumo medio de memoria sin rematerialización (barra roja) frente al mejor consumo alcanzado con rematerialización (barra verde, agregando las cuatro variantes evaluadas). El rotulado superior indica el ahorro porcentual logrado en cada combinación.

Figura 17

Memoria promedio del modelo MLP-3 en GPU NVIDIA: sin rematerialización vs. mejor rematerialización por tamaño de batch.

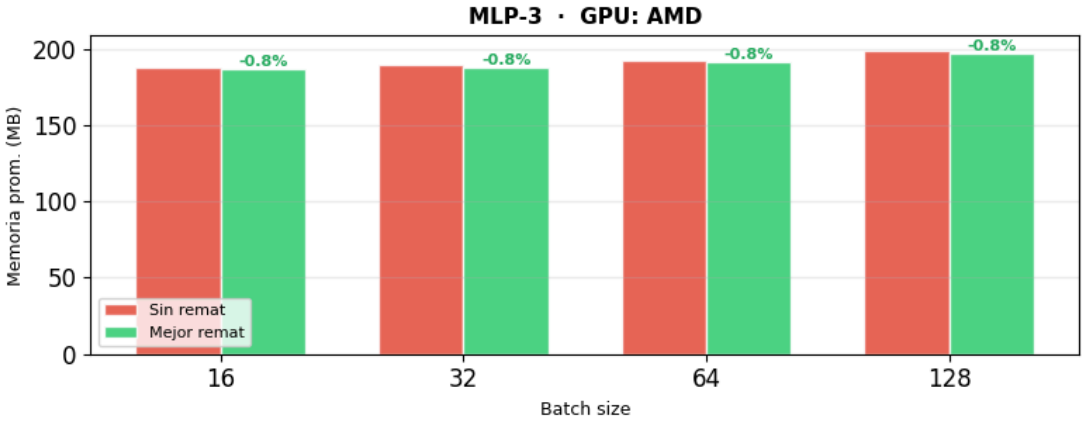


Nota. Comparación de memory_avg_MB con y sin rematerialización (mejor variante entre checkpoint_selectivo, uniform_seg2 y uniform_seg3). El porcentaje indica el ahorro relativo frente al baseline. Elaboración propia a partir de las ejecuciones del notebook.

En la GPU NVIDIA, el modelo MLP-3 consume entre 53 y 64 MB de memoria promedio, y la rematerialización logra ahorros pequeños pero consistentes entre −3,2 % y −3,7 % en los cuatro tamaños de batch. El ahorro más alto se observa con batch = 16 (−3,7 %) y disminuye levemente al crecer el batch. Como MLP-3 es una red muy simple con sólo tres capas totalmente conectadas, el presupuesto absoluto de activaciones que puede liberarse es reducido y los ahorros se mantienen en el orden de 2 MB en valor absoluto.

Figura 18

Memoria promedio del modelo MLP-3 en GPU AMD: sin rematerialización vs. mejor rematerialización por tamaño de batch.

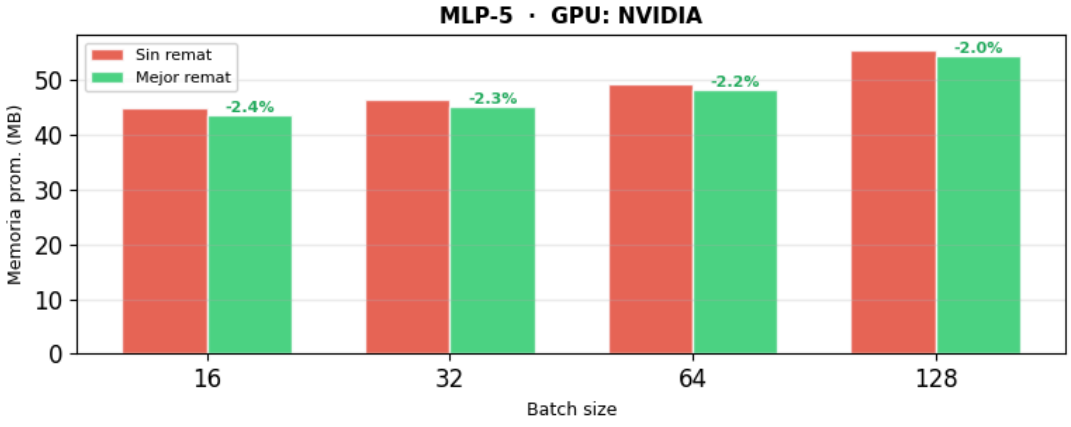


Nota. Comparación de memory_avg_MB con y sin rematerialización (mejor variante entre checkpoint_selectivo, uniform_seg2 y uniform_seg3). El porcentaje indica el ahorro relativo frente al baseline. Elaboración propia a partir de las ejecuciones del notebook.

En la GPU AMD el mismo modelo MLP-3 consume alrededor de 187--200 MB, un valor mucho más alto que en NVIDIA porque la MI210 reserva memoria base diferente por arquitectura. El ahorro porcentual se mantiene prácticamente fijo en --0,8\,\% para todos los tamaños de batch, lo que indica que la rematerialización casi no logra liberar memoria en esta combinación: la MI210 cuenta con 12 GB de VRAM y no existe presión de memoria que un modelo tan pequeño pueda aliviar.

Figura 19

Memoria promedio del modelo MLP-5 en GPU NVIDIA: sin rematerialización vs. mejor rematerialización por tamaño de batch.

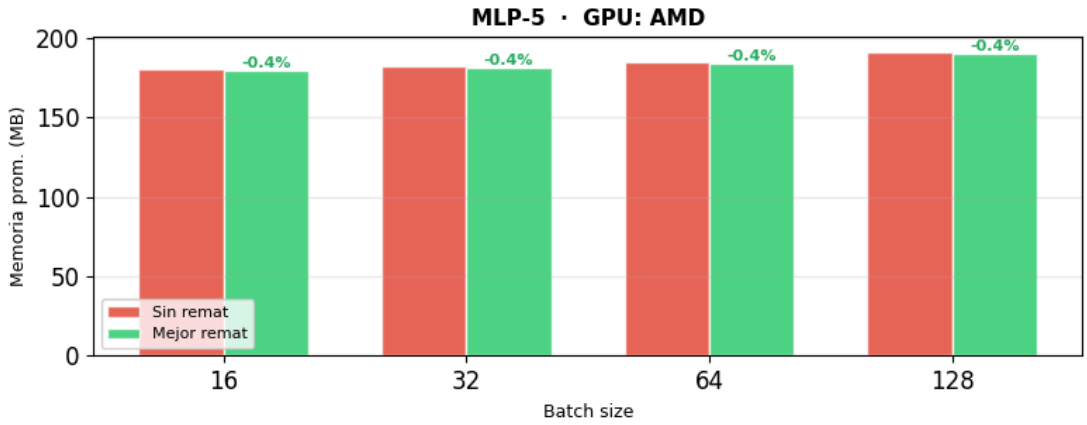


Nota. Comparación de memory_avg_MB con y sin rematerialización (mejor variante entre checkpoint_selectivo, uniform_seg2 y uniform_seg3). El porcentaje indica el ahorro relativo frente al baseline. Elaboración propia a partir de las ejecuciones del notebook.

MLP-5 sobre GPU NVIDIA consume entre 44 y 55 MB de memoria promedio. Los ahorros con rematerialización son moderados y estables, entre −2,0 % y −2,4 %, con el mejor caso en batch = 16 (−2,4 %). Al agregar dos capas ocultas adicionales el consumo aumenta ligeramente, pero el ahorro relativo cae porque los tensores intermedios siguen siendo pequeños frente al costo fijo que introduce el mecanismo de checkpoint.

Figura 20

Memoria promedio del modelo MLP-5 en GPU AMD: sin rematerialización vs. mejor rematerialización por tamaño de batch.



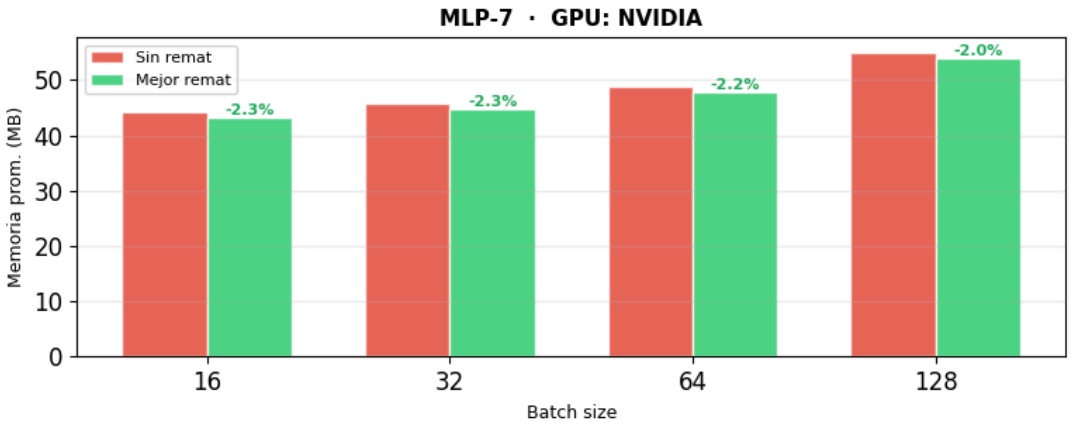
Nota. Comparación de memory_avg_MB con y sin rematerialización (mejor variante entre checkpoint_selectivo, uniform_seg2 y uniform_seg3). El porcentaje indica el ahorro relativo frente al baseline. Elaboración propia a partir de las ejecuciones del notebook.

En GPU AMD el modelo MLP-5 consume entre 180 y 191 MB, con un ahorro prácticamente constante de −0,4 % en todos los tamaños de batch. Nuevamente la GPU AMD domina el consumo base con overhead propio del driver y la rematerialización no aporta beneficio medible. El comportamiento es consistente con la hipótesis de que en modelos

pequeños sobre hardware con VRAM abundante el asignador no está bajo presión y, por tanto, no responde a la liberación agresiva de tensores.

Figura 21

Memoria promedio del modelo MLP-7 en GPU NVIDIA: sin rematerialización vs. mejor rematerialización por tamaño de batch.

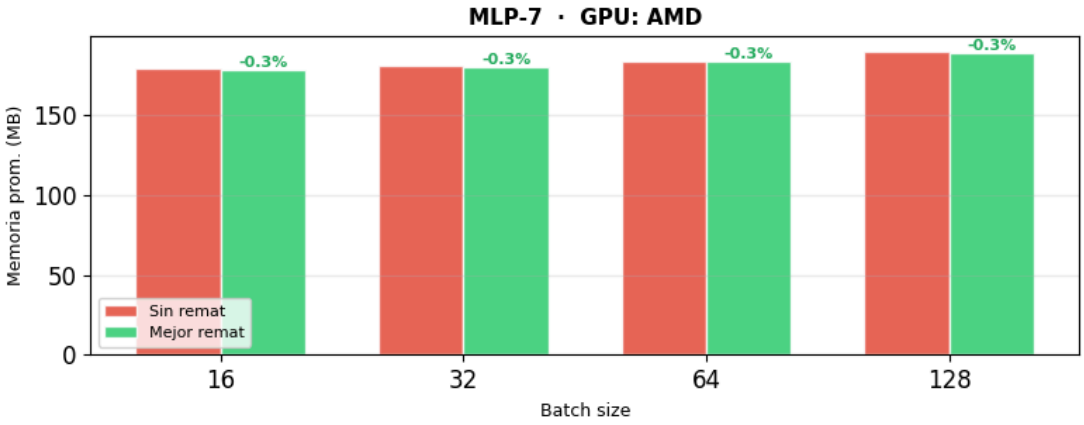


Nota. Comparación de memory_avg_MB con y sin rematerialización (mejor variante entre checkpoint_selectivo, uniform_seg2 y uniform_seg3). El porcentaje indica el ahorro relativo frente al baseline. Elaboración propia a partir de las ejecuciones del notebook.

MLP-7 sobre GPU NVIDIA consume entre 44 y 55 MB, con ahorros de rematerialización entre −2,0 % y −2,3 %. El comportamiento es prácticamente idéntico al de MLP-5, lo que confirma que agregar más capas ocultas en un MLP no aumenta significativamente el beneficio de la rematerialización: cada capa adicional añade activaciones pequeñas (vectores de pocos cientos de neuronas), muy lejos del volumen de activaciones que manejan las CNN profundas.

Figura 22

Memoria promedio del modelo MLP-7 en GPU AMD: sin rematerialización vs. mejor rematerialización por tamaño de batch.

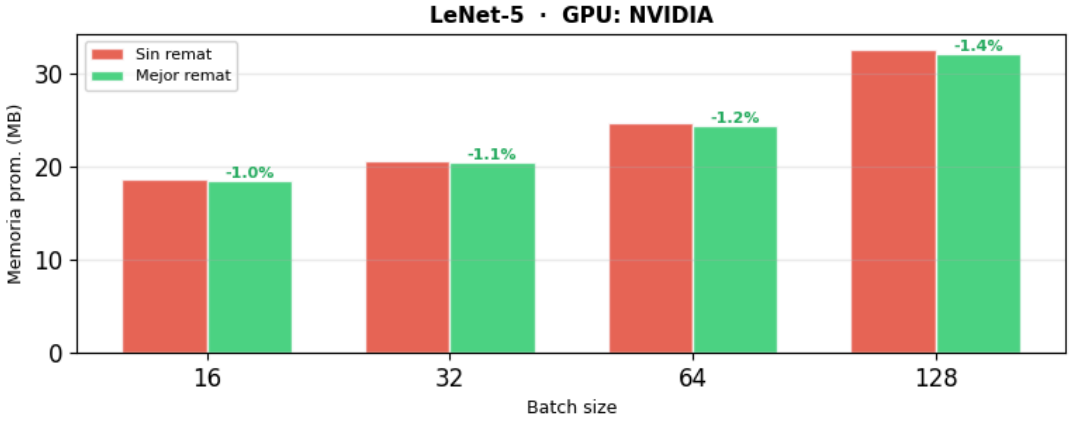


Nota. Comparación de memory_avg_MB con y sin rematerialización (mejor variante entre checkpoint_selectivo, uniform_seg2 y uniform_seg3). El porcentaje indica el ahorro relativo frente al baseline. Elaboración propia a partir de las ejecuciones del notebook.

En GPU AMD el modelo MLP-7 consume entre 180 y 191 MB y el ahorro de la rematerialización es de apenas −0,3 % en todos los tamaños de batch, el más bajo de toda la serie MLP. La tendencia es clara: sobre la MI210 los tres modelos MLP (3, 5 y 7 capas) producen ahorros residuales. Esto refuerza la conclusión de que, para arquitecturas totalmente conectadas y superficiales en volumen de activaciones, la rematerialización no es una estrategia rentable en GPU AMD.

Figura 23

Memoria promedio del modelo LeNet-5 en GPU NVIDIA: sin rematerialización vs. mejor rematerialización por tamaño de batch.

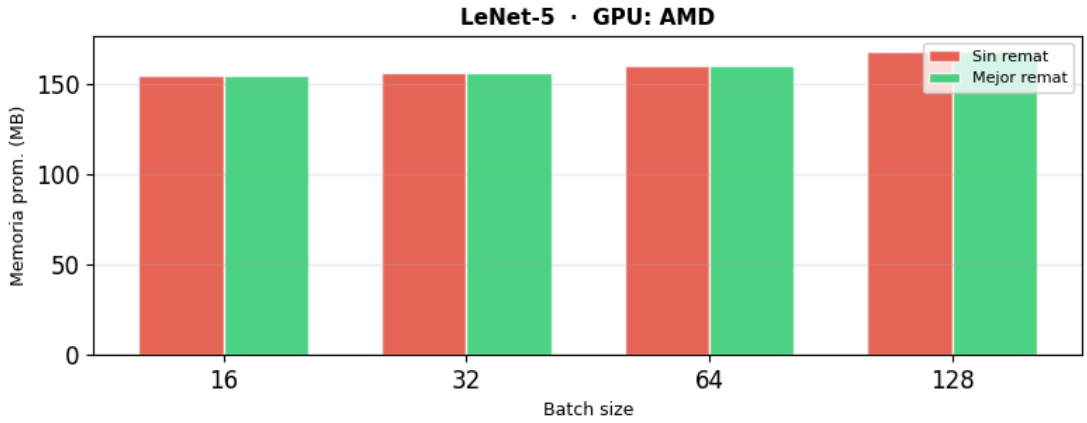


Nota. Comparación de memory_avg_MB con y sin rematerialización (mejor variante entre checkpoint_selectivo, uniform_seg2 y uniform_seg3). El porcentaje indica el ahorro relativo frente al baseline. Elaboración propia a partir de las ejecuciones del notebook.

LeNet-5 sobre GPU NVIDIA consume entre 18 y 33 MB. Aquí se observa un patrón creciente del ahorro: desde −1,0 % con batch = 16 hasta −1,4 % con batch = 128, lo que muestra que el beneficio de la rematerialización aumenta conforme crece el volumen de datos procesado por lote. Aun así, el ahorro máximo es marginal porque LeNet-5 es una CNN muy liviana: sus dos capas convolucionales tienen pocos filtros y los mapas de activación son de dimensión modesta.

Figura 24

Memoria promedio del modelo LeNet-5 en GPU AMD: sin rematerialización vs. mejor rematerialización por tamaño de batch.



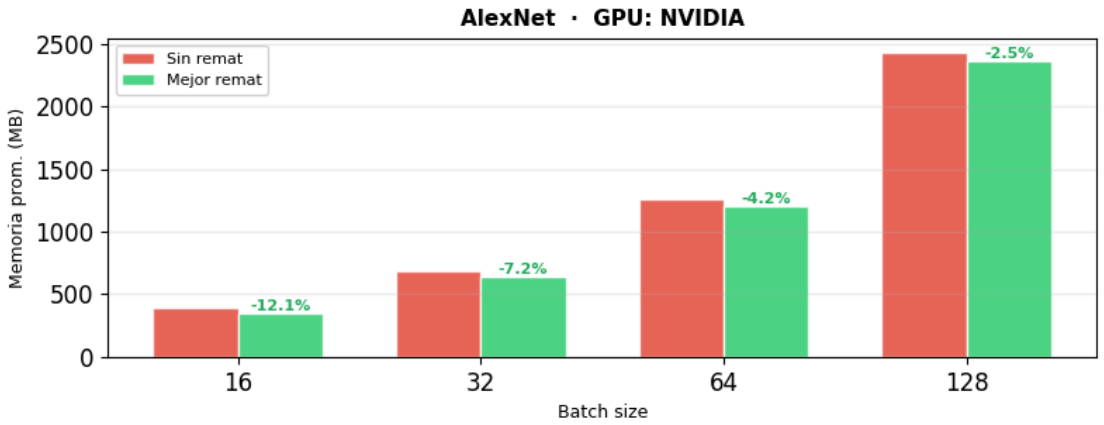
Nota. Comparación de memory_avg_MB con y sin rematerialización (mejor variante entre checkpoint_selectivo, uniform_seg2 y uniform_seg3). El porcentaje indica el ahorro relativo frente al baseline. Elaboración propia a partir de las ejecuciones del notebook.

En LeNet-5 sobre GPU AMD las barras roja y verde se superponen casi por completo y el ahorro porcentual es tan pequeño que no aparece etiquetado sobre las barras. El consumo se mantiene alrededor de 154--169 MB en los cuatro tamaños de batch y no se aprecia liberación de memoria. La combinación de LeNet-5 (modelo muy liviano) con la MI210 (64 GB de VRAM,

limitada a 12 GB) significa que no hay presión de memoria que la rematerialización pueda aliviar, y el overhead fijo del mecanismo compensa cualquier ahorro potencial.

Figura 25

Memoria promedio del modelo AlexNet en GPU NVIDIA: sin rematerialización vs. mejor rematerialización por tamaño de batch.

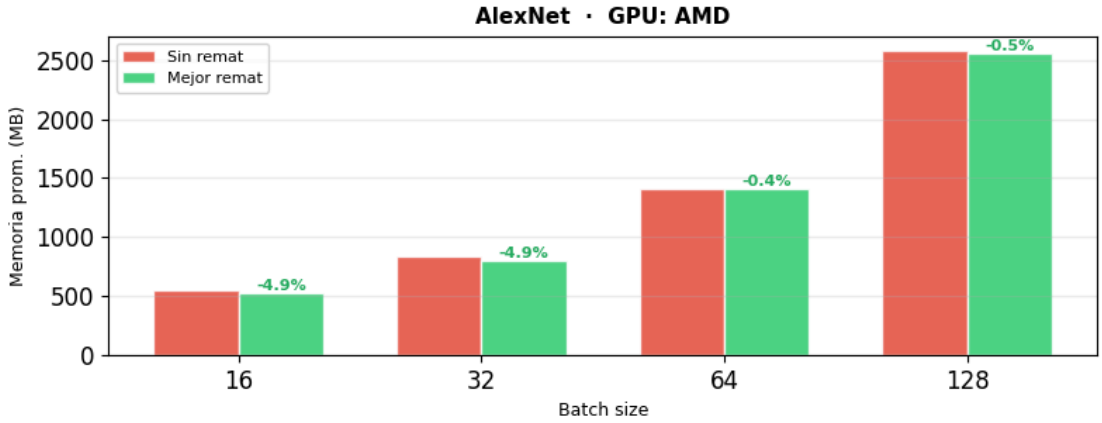


Nota. Comparación de memory_avg_MB con y sin rematerialización (mejor variante entre checkpoint_selectivo, uniform_seg2 y uniform_seg3). El porcentaje indica el ahorro relativo frente al baseline. Elaboración propia a partir de las ejecuciones del notebook.

AlexNet sobre GPU NVIDIA muestra un cambio de escala evidente: el consumo pasa de alrededor de 360 MB con batch = 16 hasta 2.419 MB con batch = 128, evidenciando cómo el tamaño del batch multiplica el requerimiento de memoria en CNN profundas. Los ahorros oscilan entre −2,5 % y −12,1 %, con el máximo en batch = 16 (−12,1 %) y disminuyendo a medida que crece el batch. En términos absolutos, el ahorro alcanza decenas de MB y confirma que la rematerialización empieza a tener un impacto tangible cuando el volumen de activaciones crece, aunque la mejor variante deja de ser la más agresiva cuando el modelo ya consume gran parte de la VRAM.

Figura 26

Memoria promedio del modelo AlexNet en GPU AMD: sin rematerialización vs. mejor rematerialización por tamaño de batch.

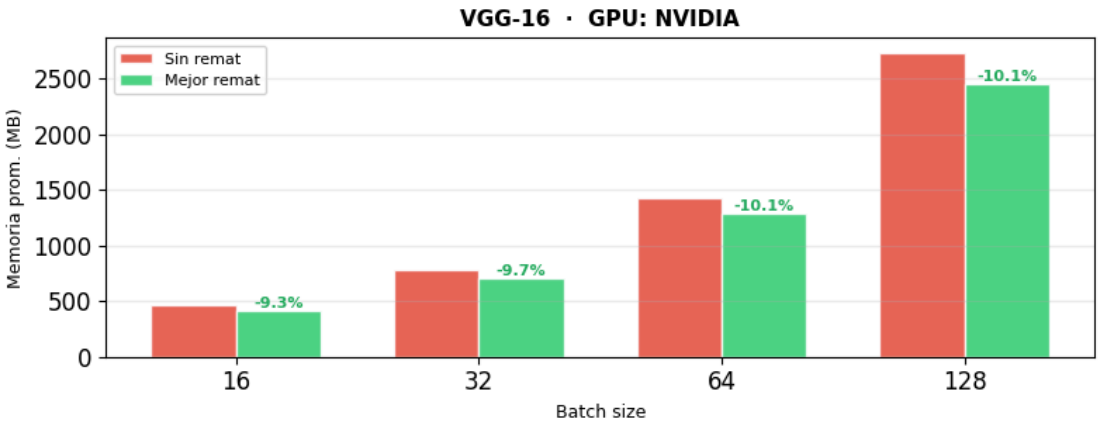


Nota. Comparación de memory_avg_MB con y sin rematerialización (mejor variante entre checkpoint_selectivo, uniform_seg2 y uniform_seg3). El porcentaje indica el ahorro relativo frente al baseline. Elaboración propia a partir de las ejecuciones del notebook.

AlexNet en GPU AMD sigue el mismo patrón de crecimiento con el batch (~496–2.554 MB), y los ahorros se sitúan entre −0,4 % y −4,9 %, con los ahorros más altos en los batches pequeños (−4,9 % en batch = 16 y batch = 32). A diferencia de los MLP, aquí sí se observa ahorro visible en la MI210 cuando el modelo es lo suficientemente grande, lo que confirma que el factor determinante no es la GPU sino la arquitectura: las CNN profundas generan suficientes activaciones intermedias para que la rematerialización aporte beneficio incluso en hardware con VRAM abundante.

Figura 27

Memoria promedio del modelo VGG-16 en GPU NVIDIA: sin rematerialización vs. mejor rematerialización por tamaño de batch.

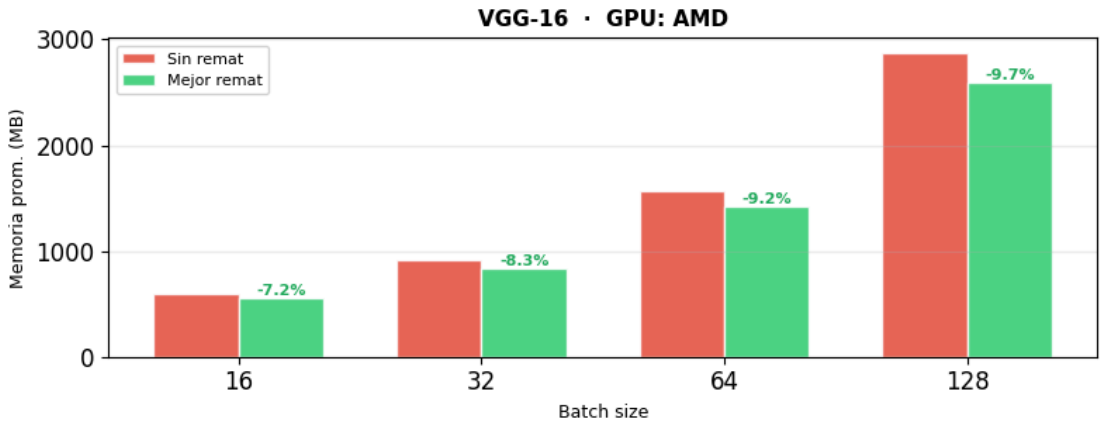


Nota. Comparación de memory_avg_MB con y sin rematerialización (mejor variante entre checkpoint_selectivo, uniform_seg2 y uniform_seg3). El porcentaje indica el ahorro relativo frente al baseline. Elaboración propia a partir de las ejecuciones del notebook.

VGG-16 sobre GPU NVIDIA es el caso donde la rematerialización brilla. El consumo escala desde alrededor de 450 MB con batch = 16 hasta cerca de 2.700 MB con batch = 128, y los ahorros alcanzan valores destacados entre −9,3 % y −10,1 %, creciendo de forma monótona con el tamaño de batch. En términos absolutos, el ahorro pasa de alrededor de 40 MB con batch = 16 a unos 270 MB con batch = 128. Esta figura evidencia que en arquitecturas profundas con muchas capas convolucionales (como VGG-16 con 13 capas convolucionales + 3 fully-connected) la rematerialización es claramente rentable y se convierte en una técnica de primera línea cuando la memoria GPU es limitada.

Figura 28

Memoria promedio del modelo VGG-16 en GPU AMD: sin rematerialización vs. mejor rematerialización por tamaño de batch.



Nota. Comparación de memory_avg_MB con y sin rematerialización (mejor variante entre checkpoint_selectivo, uniform_seg2 y uniform_seg3). El porcentaje indica el ahorro relativo frente al baseline. Elaboración propia a partir de las ejecuciones del notebook.

VGG-16 en GPU AMD confirma los resultados anteriores con ahorros entre −7,2 % y −9,7 %, creciendo monótonamente con el batch size. El consumo sin rematerialización pasa de cerca de 600 MB con batch = 16 hasta unos 2.850 MB con batch = 128 (promedio sobre las cuatro activaciones), y el ahorro absoluto pasa de unos 40 MB en batch = 16 a unos 270 MB en batch = 128. En términos prácticos, el beneficio relativo de la técnica crece junto con la presión de memoria, lo que la convierte en la estrategia recomendada para entrenamiento de CNN profundas en configuraciones exigentes.

5.7 Mejor combinación para máximo ahorro de memoria (por modelo)

Pregunta: para cada modelo, GPU y tamaño de batch, ¿qué configuración de checkpoint logra el mayor ahorro de memoria?

Para responder esta pregunta se identifica la configuración de checkpoint que maximiza el ahorro de memoria comparando directamente los valores reales del dataset en el escenario baseline misma arquitectura, misma GPU, mismo batch y misma función de activación, con los valores reales de la configuración óptima con rematerialización. Adicionalmente, se incluye la

predicción del Random Forest sobre dicha configuración óptima como mecanismo de verificación, lo que permite contrastar la coherencia entre los resultados observados y los estimados por el modelo.

Tabla 5

Máximo ahorro de memoria por GPU y tamaño de batch en MLP-3 (checkpoint óptimo: fc2, activación: leaky_relu).

GPU	Batch	Mem. base (MB)	Tiempo base (ms)	Mem. remat. (MB)	Tiempo remat. (ms)	Δ Mem. (%)	Δ Tiempo (%)	RF mem. (MB)	RF tiempo (ms)	Error RF mem. (%)	Error RF tiempo (%)
NVIDIA	16	53,14	141,04	51,13	183,28	−3,8	+29,9	57,20	222,04	11,9	21,2
NVIDIA	32	54,69	146,85	52,66	189,52	−3,7	+29,1	61,80	224,74	17,4	18,6
NVIDIA	64	57,80	191,47	55,74	254,30	−3,6	+32,8	68,47	312,84	22,8	23,0
NVIDIA	128	64,01	251,62	61,88	338,60	−3,3	+34,6	145,11	959,74	134,5	183,4
AMD	16	188,89	55,50	186,88	66,78	−1,1	+20,3	192,30	128,49	2,9	92,4
AMD	32	190,44	52,19	188,41	62,91	−1,1	+20,5	196,05	136,46	4,1	116,9
AMD	64	193,55	57,04	191,49	67,47	−1,1	+18,3	215,02	273,35	12,3	305,1
AMD	128	199,76	59,72	197,63	71,38	−1,1	+19,5	296,59	455,61	50,1	538,3

Nota. Δ Mem. y Δ Tiempo son cambios porcentuales del escenario remat. respecto al baseline. RF mem. y RF tiempo son las predicciones del Random Forest sobre la misma configuración. Elaboración propia a partir de las ejecuciones del notebook.

En MLP-3 la rematerialización apenas consigue liberar memoria: el ahorro absoluto se mide en pocos MB y, en la GPU AMD, que dispone de mucha memoria libre, el efecto es casi invisible. A cambio, el tiempo de entrenamiento sube de forma apreciable. La predicción del Random Forest se desvía mucho en términos porcentuales especialmente para batch = 128, donde el predictor sobreestima de forma notable la memoria y el tiempo, pero esto no implica que el modelo esté fallando como clasificador de decisiones: en esta arquitectura los valores reales son tan pequeños que cualquier diferencia absoluta se amplifica al expresarse como porcentaje. La lectura práctica es directa: en esta arquitectura no compensa aplicar rematerialización y el predictor debe usarse como guía de decisión (aplicar o no) más que como estimador exacto.

Tabla 6

Máximo ahorro de memoria por GPU y tamaño de batch en MLP-5 (checkpoint óptimo: fc2_fc4, activación: leaky_relu).

GPU	Batch	Mem. base (MB)	Tiempo base (ms)	Mem. remat. (MB)	Tiempo remat. (ms)	Δ Mem. (%)	Δ Tiempo (%)	RF mem. (MB)	RF tiempo (ms)	Error RF mem. (%)	Error RF tiempo (%)
NVIDIA	16	44,92	137,77	43,79	168,28	−2,5	+22,1	53,55	209,43	22,3	24,5
NVIDIA	32	46,47	145,53	45,33	176,53	−2,5	+21,3	54,17	204,12	19,5	15,6
NVIDIA	64	49,58	174,68	48,41	212,80	−2,4	+21,8	66,85	305,25	38,1	43,4
NVIDIA	128	55,78	238,69	54,57	300,49	−2,2	+25,9	96,46	815,54	76,8	171,4
AMD	16	180,67	63,76	179,54	73,86	−0,6	+15,8	188,54	130,83	5,0	77,1
AMD	32	182,22	63,03	181,08	72,90	−0,6	+15,7	190,54	133,07	5,2	82,5
AMD	64	185,33	67,39	184,16	77,53	−0,6	+15,0	212,95	225,31	15,6	190,6
AMD	128	191,53	73,91	190,32	85,44	−0,6	+15,6	230,41	288,54	21,1	237,7

Nota. Δ Mem. y Δ Tiempo son cambios porcentuales del escenario remat. respecto al baseline. Elaboración propia a partir de las ejecuciones del notebook.

En MLP-5 el panorama es prácticamente el mismo que en MLP-3, pero aún más débil: aunque se aplique la mejor combinación de checkpoint, el ahorro de memoria es mínimo y el aumento de tiempo persiste. La diferencia visible es que el Random Forest empieza a acertar mejor en la GPU AMD, donde los valores absolutos de memoria son más parecidos a los que más abundan en el dataset de entrenamiento. Aun así, el balance sigue siendo desfavorable: no es rentable rematerializar en esta red y el predictor funciona mejor como orientador de decisión que como estimador fino.

Tabla 7

Máximo ahorro de memoria por GPU y tamaño de batch en MLP-7 (checkpoint óptimo: fc2_fc4_fc6, activación: leaky_relu).

GPU	Batch	Mem. base (MB)	Tiempo base (ms)	Mem. remat. (MB)	Tiempo remat. (ms)	Δ Mem. (%)	Δ Tiempo (%)	RF mem. (MB)	RF tiempo (ms)	Error RF mem. (%)	Error RF tiempo (%)
NVIDIA	16	44,39	152,51	43,31	184,57	−2,4	+21,0	47,64	200,03	10,0	8,4
NVIDIA	32	45,94	162,69	44,85	198,00	−2,4	+21,7	47,46	200,59	5,8	1,3
NVIDIA	64	49,05	193,11	47,93	234,45	−2,3	+21,4	48,80	219,19	1,8	6,5
NVIDIA	128	55,25	253,14	54,10	318,15	−2,1	+25,7	52,77	258,98	2,5	18,6
AMD	16	180,14	81,09	179,06	93,46	−0,6	+15,3	180,81	93,84	1,0	0,4

GPU	Batch	Mem. base (MB)	Tiempo base (ms)	Mem. remat. (MB)	Tiempo remat. (ms)	Δ Mem. (%)	Δ Tiempo (%)	RF mem. (MB)	RF tiempo (ms)	Error RF mem. (%)	Error RF tiempo (%)
AMD	32	181,69	78,73	180,60	90,51	−0,6	+15,0	182,00	96,10	0,8	6,2
AMD	64	184,80	84,28	183,68	97,16	−0,6	+15,3	186,65	105,82	1,6	8,9
AMD	128	191,00	90,52	189,85	104,82	−0,6	+15,8	193,88	119,13	2,1	13,6

Nota. Δ Mem. y Δ Tiempo son cambios porcentuales del escenario remat. respecto al baseline. Elaboración propia a partir de las ejecuciones del notebook.

En MLP-7 el ahorro de memoria sigue siendo pequeño y el costo en tiempo sigue presente, de manera que la recomendación operativa no cambia: no compensa aplicar rematerialización. Lo interesante aparece en el comportamiento del Random Forest: al tratarse de una red más profunda, el predictor logra acertar mucho mejor en ambas GPUs, con errores bastante más bajos que en MLP-3 y MLP-5 (errores en memoria del orden de 1–10 %). En otras palabras, a medida que la red gana capas el predictor empieza a estabilizarse y a reconocer mejor los patrones, aunque la utilidad práctica de la técnica en esta arquitectura siga siendo limitada.

Tabla 8

Máximo ahorro de memoria por GPU y tamaño de batch en LeNet-5 (checkpoint óptimo: conv1_conv2_fc, activación: leaky_relu).

GPU	Batch	Mem. base (MB)	Tiempo base (ms)	Mem. remat. (MB)	Tiempo remat. (ms)	Δ Mem. (%)	Δ Tiempo (%)	RF mem. (MB)	RF tiempo (ms)	Error RF mem. (%)	Error RF tiempo (%)
NVIDIA	16	18,55	179,87	18,43	213,57	−0,6	+18,7	27,94	268,30	51,6	25,6
NVIDIA	32	20,60	239,76	20,38	275,71	−1,1	+15,0	30,67	278,17	50,5	0,9
NVIDIA	64	24,70	299,59	24,29	357,93	−1,7	+19,5	35,49	377,86	46,1	5,6
NVIDIA	128	32,99	345,69	32,09	418,03	−2,7	+20,9	72,18	998,57	124,9	138,9
AMD	16	154,61	172,60	154,49	209,91	−0,1	+21,6	158,18	342,44	2,4	63,1
AMD	32	156,66	260,18	156,44	312,72	−0,1	+20,2	158,94	350,67	1,6	12,1
AMD	64	160,76	398,26	160,34	460,42	−0,3	+15,6	158,19	423,19	1,3	8,1
AMD	128	168,97	687,96	168,15	724,64	−0,5	+5,3	172,13	567,67	2,4	21,7

Nota. Δ Mem. y Δ Tiempo son cambios porcentuales del escenario remat. respecto al baseline. Elaboración propia a partir de las ejecuciones del notebook.

En LeNet-5 las dos GPUs reaccionan de forma distinta. En la GPU NVIDIA el ahorro de memoria crece a medida que se usan batches más grandes, porque allí hay más activaciones

intermedias por recortar; pero el tamaño absoluto sigue siendo pequeño (menos de 1 MB en valor absoluto). En la GPU AMD la memoria disponible es tan amplia que el asignador nunca se ve obligado a liberar tensores, y por eso el ahorro es prácticamente nulo (entre −0,1 % y −0,5 %). También ocurre algo importante con el Random Forest: en la GPU AMD logra predicciones de memoria muy ajustadas (errores entre 1,3 % y 2,4 %), lo que marca el punto de inflexión en el que el predictor deja de fallar por la escala pequeña de los valores y empieza a funcionar como estimador útil.

Tabla 9

Máximo ahorro de memoria por GPU y tamaño de batch en AlexNet (checkpoint óptimo: full, activación: leaky_relu).

GPU	Batch	Mem. base (MB)	Tiempo base (ms)	Mem. remat. (MB)	Tiempo remat. (ms)	Δ Mem. (%)	Δ Tiempo (%)	RF mem. (MB)	RF tiempo (ms)	Error RF mem. (%)	Error RF tiempo (%)
NVIDIA	16	360,39	1.076,16	349,41	1.809,38	−3,0	+68,1	825,56	3.308,65	136,3	82,9
NVIDIA	32	651,74	1.620,51	635,62	2.594,47	−2,5	+60,1	862,12	3.381,05	35,6	30,3
NVIDIA	64	1.235,64	2.806,96	1.206,56	4.513,44	−2,4	+60,8	1.125,37	4.170,06	6,7	7,6
NVIDIA	128	2.418,54	5.207,88	2.364,53	8.334,96	−2,2	+60,0	1.876,37	6.945,13	20,6	16,7
AMD	16	558,87	768,79	522,12	1.199,20	−6,6	+56,0	982,85	2.277,00	88,2	89,9
AMD	32	850,83	1.580,89	795,16	2.026,22	−6,5	+28,2	1.024,57	2.388,96	28,9	17,9
AMD	64	1.434,87	2.919,12	1.406,53	3.797,13	−2,0	+30,1	1.389,30	3.351,53	1,2	11,7
AMD	128	2.618,27	5.398,96	2.564,72	5.799,31	−2,0	+7,4	1.970,29	4.690,66	23,2	19,1

Nota. Δ Mem. y Δ Tiempo son cambios porcentuales del escenario remat. respecto al baseline. Elaboración propia a partir de las ejecuciones del notebook.

En AlexNet la rematerialización empieza a mostrar un perfil interesante. En la GPU AMD el ahorro real supera el −6 % en los batches pequeños (−6,6 % y −6,5 % con batch = 16 y 32) y disminuye a −2,0 % para batches ≥ 64; en términos absolutos, se liberan decenas de MB en todos los casos. En NVIDIA el ahorro porcentual es más estable pero menor (≈ −2 a −3 %) y el coste en tiempo crece de forma evidente (overhead de hasta +68 %), reflejo de la presión real de memoria que existe en una GPU de 12 GB. Lo más relevante es el desempeño del Random Forest: a partir de batch = 64 el predictor deja de equivocarse de forma exagerada y empieza a

acertar con bastante precisión tanto en memoria como en tiempo (errores en memoria entre 1,2 % y 6,7 % y errores en tiempo entre 7,6 % y 19,1 %). AlexNet opera en un rango de valores bien representado en el dataset de entrenamiento y es la primera arquitectura donde el Random Forest pasa de clasificador general a estimador confiable.

Tabla 10

Máximo ahorro de memoria por GPU y tamaño de batch en VGG-16 (checkpoint óptimo: full, activación: leaky_relu).

GPU	Batch	Mem. base (MB)	Tiempo base (ms)	Mem. remat. (MB)	Tiempo remat. (ms)	Δ Mem. (%)	Δ Tiempo (%)	RF mem. (MB)	RF tiempo (ms)	Error RF mem. (%)	Error RF tiempo (%)
NVIDIA	16	522,28	7.881,56	416,64	12.213,30	−20,2	+55,0	1.033,99	30.099,92	148,2	146,5
NVIDIA	32	906,28	17.026,48	704,52	25.170,24	−22,3	+47,8	1.062,72	31.023,38	50,8	23,3
NVIDIA	64	1.678,77	32.144,87	1.282,93	48.387,40	−23,6	+50,5	1.332,92	38.821,16	3,9	19,8
NVIDIA	128	3.233,53	66.787,07	2.454,02	102.374,68	−24,1	+53,3	1.830,18	54.065,68	25,4	47,2
AMD	16	657,47	4.887,05	551,78	6.075,62	−16,1	+24,3	1.108,11	11.254,17	100,8	85,2
AMD	32	1.043,11	9.428,59	840,94	11.133,12	−19,4	+18,1	1.162,19	12.010,60	38,2	7,9
AMD	64	1.815,39	15.071,58	1.418,82	21.806,38	−21,8	+44,7	1.584,14	16.108,92	11,7	26,1
AMD	128	3.373,01	10.852,25	2.591,05	12.168,50	−23,2	+12,1	1.875,61	16.230,60	27,6	33,4

Nota. Δ Mem. y Δ Tiempo son cambios porcentuales del escenario remat. respecto al baseline. Elaboración propia a partir de las ejecuciones del notebook.

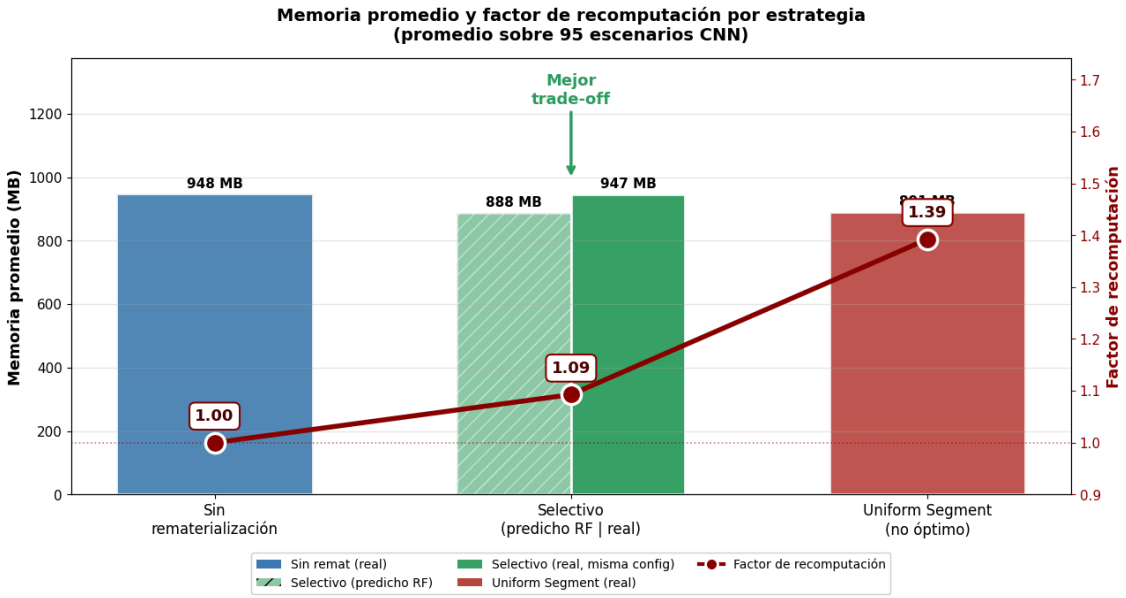
VGG-16 es la única arquitectura donde la rematerialización tiene un impacto realmente significativo: el ahorro de memoria alcanza valores entre −16 % y −24 % y crece con el tamaño de batch, lo que confirma que este tipo de técnica resulta verdaderamente útil cuando la red es grande y se trabaja cerca del límite de memoria. En términos absolutos, el ahorro supera los 780 MB en VGG-16 con batch = 128. A cambio, el tiempo de entrenamiento aumenta de forma evidente. Además, el Random Forest alcanza aquí su mejor desempeño del estudio, con errores en memoria de tan sólo 3,9 % (NVIDIA, batch = 64) y 11,7 % (AMD, batch = 64) y errores en tiempo del orden del 8 % al 26 % para los batches intermedios. La conclusión es clara: VGG-16 es el escenario donde la rematerialización resulta plenamente rentable y donde el modelo predictivo puede utilizarse con confianza como herramienta de decisión operativa.

5.8 Síntesis Comparativa del Trade-off entre Estrategias

Las secciones anteriores describieron el comportamiento de la rematerialización arquitectura por arquitectura. Esta subsección sintetiza esos hallazgos en una vista agregada sobre los 95 escenarios CNN del dataset, contrastando de forma directa las tres estrategias, sin rematerialización, checkpoint selectivo y uniform segment, en términos conjuntos de consumo de memoria, factor de recomputación y tiempo de ejecución. El objetivo es ofrecer una lectura global del compromiso antes de derivar los lineamientos prácticos.

Figura 29

Memoria promedio y factor de recomputación por estrategia de rematerialización sobre el conjunto de escenarios CNN



Nota. Promedios calculados sobre los 95 escenarios CNN (arquitectura × GPU × batch × activación). Las barras indican la memoria promedio en MB —para el selectivo se muestran la predicción del Random Forest y el valor real de la configuración recomendada y la línea roja representa el factor de recomputación asociado a cada estrategia. Elaboración propia a partir de las ejecuciones del notebook.

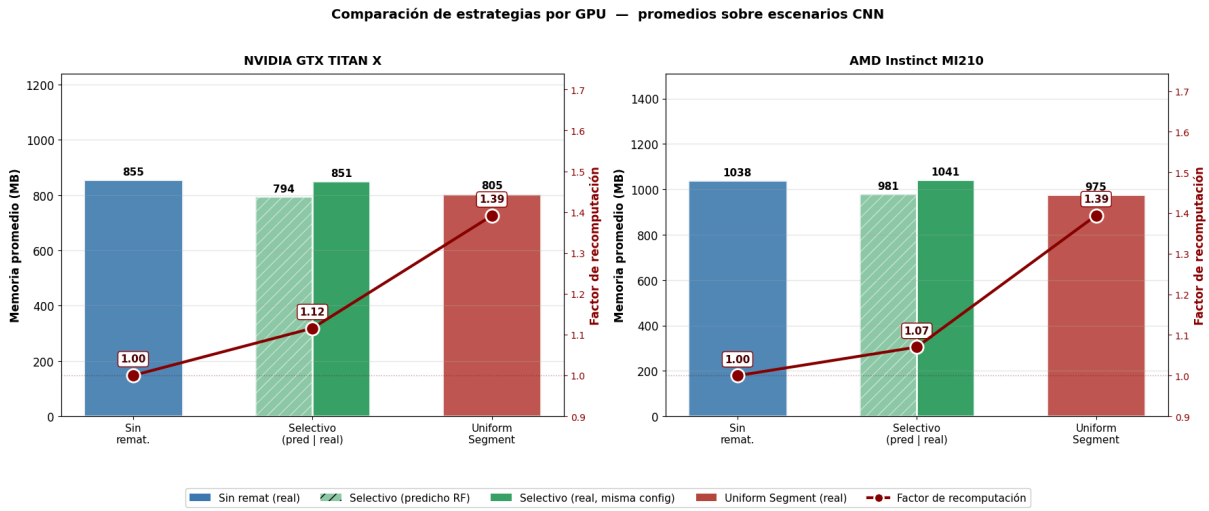
La gráfica resumen condensa el comportamiento promedio del trade-off. En memoria, las tres estrategias producen valores muy cercanos a nivel agregado: el baseline sin

rematerialización promedia alrededor de 948 MB, el checkpoint selectivo recomendado por el Random Forest cerca de 947 MB en su valor real (888 MB en la predicción del modelo) y uniform segment unos 891 MB. La diferencia absoluta es modesta porque el promedio mezcla escenarios de escalas muy distintas LeNet-5 con decenas de MB, AlexNet con cientos y VGG-16 con más de mil, de modo que el ahorro relevante de VGG-16 se diluye al agregarse con arquitecturas donde la técnica casi no actúa.

El factor de recomputación es donde aparece la diferencia decisiva. Sin rematerialización vale 1,00 por definición; el checkpoint selectivo se mantiene en 1,09, lo que significa que recomputa muy poco; y uniform segment salta a 1,39, es decir, introduce cerca de un 40 % de cómputo adicional. Aunque uniform segment libere algo más de memoria absoluta, paga un sobrecoste de cómputo casi cuatro veces mayor que el del selectivo. Por ello la anotación “mejor trade-off” se ubica sobre la barra del checkpoint selectivo: esta estrategia captura casi todo el ahorro útil con una fracción del coste, mientras que uniform segment solo resulta preferible cuando la restricción dominante es la memoria absoluta y no la eficiencia global.

Figura 30

Comparación de estrategias de rematerialización por GPU: memoria promedio y factor de recomputación



Nota. Promedios sobre los escenarios CNN separados por GPU: NVIDIA GTX TITAN X (47 escenarios) y AMD Instinct MI210 (48 escenarios). Las barras indican la memoria promedio en MB y la línea roja el factor de recomputación. Elaboración propia a partir de las ejecuciones del notebook.

La comparación entre GPUs revela patrones distintos según el hardware. En la NVIDIA GTX TITAN X, de 12 GB, el checkpoint selectivo recomendado por el Random Forest se sitúa en 851 MB de memoria real frente a 855 MB del baseline, con un factor de recomputación de apenas 1,12; uniform segment ahorra una cantidad similar (805 MB) pero paga un factor de 1,39. En esta GPU, donde la presión de memoria es real en VGG-16 y AlexNet con batches grandes, el selectivo es estrictamente mejor: ofrece un ahorro útil comparable con un coste de cómputo mucho menor.

En la AMD Instinct MI210, con la memoria limitada a 12 GB en este proyecto, la lectura cambia. El checkpoint selectivo queda en 1.041 MB de memoria real, prácticamente igual al baseline de 1.038 MB, mientras que uniform segment desciende a 975 MB. Pese a que en este caso sí existe presión de memoria, las heurísticas del Random Forest siguen recomendando configuraciones que casi no recomputan, por lo que el selectivo se vuelve prácticamente inerte. En esta GPU solo uniform segment produce un ahorro mensurable, de nuevo a costa del mismo

factor de recomputación de 1,39. La concordancia entre la barra predicha por el Random Forest y la real es buena en NVIDIA y algo menor en AMD, lo que confirma que la mayor utilidad práctica del modelo se obtiene bajo condiciones de memoria más restrictivas.

5.9 Precisión del Modelo Predictivo por Escenario

Como cierre del análisis del modelo predictivo, esta subsección evalúa la precisión del Random Forest sobre el conjunto de prueba (15 % del dataset, 286 configuraciones nunca vistas durante el entrenamiento), desagregada por la combinación de modelo, GPU y tamaño de lote. La métrica empleada es la precisión definida como el complemento del error porcentual absoluto medio, es decir, precisión = máx(0; 1 − MAPE), truncada a cero cuando el error relativo medio supera el 100 %. De forma global sobre las 286 filas del test, el Random Forest alcanza una precisión del 71,8 % para la memoria promedio y del 57,6 % para el tiempo total de entrenamiento. La Tabla 11 detalla este desempeño por escenario.

Tabla 11

Precisión del Random Forest en el conjunto de prueba (15 %) desagregada por modelo, GPU y tamaño de lote.

Modelo	GPU	Batch	Precisión memoria	Precisión tiempo	Régimen
MLP-3	AMD	16	99.5%	9.6%	Red superficial
MLP-3	AMD	64	83.8%	0.0%	Red superficial
MLP-3	AMD	128	48.1%	0.0%	Red superficial
MLP-3	NVIDIA	32	89.0%	55.4%	Red superficial
MLP-5	AMD	16	96.3%	36.5%	Red superficial
MLP-5	AMD	32	95.3%	6.4%	Red superficial
MLP-5	AMD	64	86.3%	0.0%	Red superficial
MLP-5	AMD	128	76.8%	0.0%	Red superficial
MLP-5	NVIDIA	16	84.5%	70.6%	Red superficial
MLP-5	NVIDIA	32	91.2%	80.9%	Red superficial
MLP-5	NVIDIA	64	70.2%	50.2%	Red superficial
MLP-5	NVIDIA	128	32.2%	0.0%	Red superficial
MLP-7	AMD	16	98.8%	95.0%	Red superficial
MLP-7	AMD	32	98.9%	88.7%	Red superficial
MLP-7	AMD	64	99.3%	90.8%	Red superficial

Modelo	GPU	Batch	Precisión memoria	Precisión tiempo	Régimen
MLP-7	AMD	128	96.3%	70.6%	Red superficial
MLP-7	NVIDIA	16	90.5%	81.4%	Red superficial
MLP-7	NVIDIA	32	95.1%	94.3%	Red superficial
MLP-7	NVIDIA	64	97.3%	92.5%	Red superficial
MLP-7	NVIDIA	128	96.9%	82.5%	Red superficial
LeNet-5	AMD	16	98.2%	43.6%	CNN ligera
LeNet-5	AMD	32	97.9%	79.8%	CNN ligera
LeNet-5	AMD	64	97.4%	96.0%	CNN ligera
LeNet-5	AMD	128	91.7%	78.6%	CNN ligera
LeNet-5	NVIDIA	16	47.4%	68.5%	CNN ligera
LeNet-5	NVIDIA	32	52.5%	91.6%	CNN ligera
LeNet-5	NVIDIA	64	44.4%	79.9%	CNN ligera
LeNet-5	NVIDIA	128	0.0%	0.0%	CNN ligera
AlexNet	AMD	16	27.9%	1.5%	CNN intermedia
AlexNet	AMD	32	82.9%	81.8%	CNN intermedia
AlexNet	AMD	64	95.5%	93.6%	CNN intermedia
AlexNet	AMD	128	74.0%	76.0%	CNN intermedia
AlexNet	NVIDIA	16	0.0%	22.6%	CNN intermedia
AlexNet	NVIDIA	32	77.8%	72.2%	CNN intermedia
AlexNet	NVIDIA	64	87.7%	92.0%	CNN intermedia
AlexNet	NVIDIA	128	77.8%	88.4%	CNN intermedia
VGG-16	AMD	16	13.5%	22.1%	CNN profunda
VGG-16	AMD	32	70.8%	96.7%	CNN profunda
VGG-16	AMD	64	93.9%	81.7%	CNN profunda
VGG-16	AMD	128	69.4%	74.7%	CNN profunda
VGG-16	NVIDIA	16	0.0%	0.0%	CNN profunda
VGG-16	NVIDIA	32	66.5%	67.7%	CNN profunda
VGG-16	NVIDIA	64	93.2%	88.0%	CNN profunda
VGG-16	NVIDIA	128	69.5%	67.5%	CNN profunda

Nota. Precisión = máx(0; 1 − MAPE) calculada por grupo y variable objetivo sobre el conjunto de prueba. El régimen agrupa las arquitecturas por familia y volumen de activaciones. Elaboración propia a partir de las ejecuciones del notebook.

La tabla muestra que la precisión del Random Forest depende fuertemente del régimen de la arquitectura y de la escala absoluta de las métricas. En memoria, el modelo alcanza valores muy altos en los MLP y en LeNet-5 sobre la GPU AMD con frecuencia por encima del 95 %, porque allí los consumos se concentran en un rango estrecho bien representado en el dataset de entrenamiento. En la GPU NVIDIA, en cambio, los escenarios de modelos pequeños registran caídas marcadas: cuando los valores absolutos son de apenas decenas de MB, una desviación

mínima se amplifica al expresarse como porcentaje, lo que explica las precisiones bajas de LeNet-5 con batch grande.

En tiempo, el patrón es coherente con lo observado en las gráficas de calibración: el modelo es más preciso en las CNN intermedias y profundas con batches medianos —AlexNet y VGG-16 entre batch 32 y 64 superan habitualmente el 80 %— y pierde precisión en los extremos, tanto en los MLP con batches grandes como en algunos escenarios de batch 16. La lectura conjunta de la tabla confirma la conclusión de las secciones anteriores: el Random Forest funciona como estimador confiable en el régimen donde la rematerialización es realmente útil —las CNN profundas cercanas al límite de memoria— y debe interpretarse como guía cualitativa de decisión, más que como estimador exacto, en las arquitecturas pequeñas donde la técnica no aporta beneficio neto.

5.10 Lineamientos Prácticos

La presente síntesis orienta el uso de la rematerialización en los escenarios cubiertos por los modelos definidos desde el inicio, identificando los casos en los que la técnica resulta más eficiente y aquellos en los que es preferible entrenar sin ella. Estos lineamientos constituyen un marco de aplicación práctica que complementa el modelo predictivo y su validación experimental, centrando el análisis en el impacto del tamaño de lote, la topología de red y la GPU utilizada sobre el consumo de memoria y el tiempo de cómputo. La Tabla 12 condensa las recomendaciones operativas derivadas del análisis experimental.

Tabla 12

Lineamientos prácticos para la aplicación de rematerialización de activaciones según topología de red, tamaño de lote y GPU.

Categoría	Criterio	Recomendación
Topología de red	MLP (3, 5 y 7 capas)	No aplicar rematerialización; las activaciones son pequeñas y el ahorro (~ 0,3–3,8 %) es despreciable frente al overhead en tiempo (~ 15–35 %).
Topología de red	LeNet-5	No activar por defecto. Sólo bajo presión real de memoria, y en ese caso uniform_seg2. La técnica está sobredimensionada para esta arquitectura ligera.
Topología de red	AlexNet	Aplicar checkpoint selectivo guiado por el Random Forest a partir de batch ≥ 32 en ambas GPUs. En AMD con batches grandes puede incluso reducirse el tiempo gracias al reuso de cache/kernel en ROCm.
Topología de red	VGG-16	Activar rematerialización siempre desde batch ≥ 32. Usar checkpoint_selectivo guiado por el RF para el mejor balance; usar uniform_seg2 o uniform_seg3 si se busca el máximo ahorro absoluto de memoria.
Tamaño de lote	Batch = 16	Rematerialización útil sólo en VGG-16 cuando la GPU está cercana al límite de memoria.
Tamaño de lote	Batch = 32 a 64	Zona de mayor rendimiento del checkpoint selectivo en AlexNet y VGG-16; los overheads del selectivo se mantienen contenidos (3–7 % en NVIDIA, incluso negativos en AMD).
Tamaño de lote	Batch = 128	En VGG-16 se libera la mayor cantidad absoluta de memoria (más de 780 MB de ahorro real). Usar selectivo guiado por el RF; evitar uniform_seg3 salvo restricción dura de memoria.
GPU	NVIDIA GTX Titan X (12 GB)	Particularmente relevante para VGG-16 y AlexNet con batch ≥ 64, donde existe presión real de memoria. Aplicar selectivo del RF siempre que sea posible; uniform_seg2 como fallback.
GPU	AMD Instinct MI210 (64 GB)	Raramente necesaria en MLP/LeNet-5 (poca o ninguna presión de memoria). En AlexNet y VGG-16 desde batch = 64, sí conviene activarla: libera memoria y, por reuso de cache/kernel en ROCm, ocasionalmente reduce también el tiempo.
Criterio de decisión	Modelo predictivo Random Forest	Activar rematerialización si el Random Forest estima una eficiencia (1 − overhead/ahorro) suficiente o si existe restricción dura de memoria.
Cuándo no aplicar	MLP y LeNet-5 en GPU con VRAM amplia	Descartar la técnica cuando la eficiencia predicha quede por debajo del umbral o cuando no exista presión de memoria.
Cuándo aplicar	VGG-16 y CNN profundas en GPU con VRAM apretada	Recomendada en situaciones de borde de OOM y cuando se prioriza sostener el batch máximo para mejorar la convergencia.

Nota. Síntesis de los criterios de aplicación de la rematerialización derivados del análisis experimental realizado sobre las seis arquitecturas de referencia y las dos GPUs evaluadas. Elaboración propia a partir de las ejecuciones del notebook.

6. Conclusiones

El desarrollo de este trabajo permitió cumplir los cuatro objetivos específicos planteados y, con ellos, el objetivo general de caracterizar y optimizar el trade-off cómputo–memoria mediante rematerialización de activaciones. Se construyó un conjunto de datos experimental a partir del perfilado sistemático de varias arquitecturas de referencia sobre dos GPU de familias y capacidades distintas, registrando para cada configuración el consumo de memoria, el tiempo de entrenamiento y un amplio conjunto de variables descriptoras. Sobre esos datos se entrenaron y compararon dos modelos predictivos de naturaleza diferente, uno lineal y otro basado en árboles de decisión. El modelo de árboles resultó claramente superior, lo que confirma que el coste de entrenamiento bajo rematerialización responde a relaciones no lineales que un modelo lineal no logra representar. Aun así, su precisión fue desigual: muy buena en los escenarios donde la técnica produce efectos apreciables y más limitada en los regímenes de bajo consumo, una consecuencia esperable del desbalance del conjunto de datos.

El análisis estadístico y la interpretación de los resultados confirmaron la hipótesis de partida: sí existen configuraciones óptimas, y estas dependen fuertemente de la topología de la red, del tamaño de lote y de las restricciones de memoria del hardware. La validación mediante pruebas no paramétricas mostró que aplicar rematerialización produce diferencias significativas frente al entrenamiento sin la técnica, mientras que la elección entre sus variantes apenas se distingue del comportamiento esperado por azar. En cuanto a la topología, el beneficio es prácticamente nulo en las redes más simples, moderado en las intermedias y verdaderamente relevante solo en las arquitecturas convolucionales profundas, donde el ahorro de memoria llega a ser sustancial. El hardware también influye: la técnica aporta más cuando la GPU opera cerca de su límite de memoria y pierde utilidad cuando la capacidad disponible es holgada. A partir de

estas observaciones se elaboró una guía práctica que indica cuándo aplicar la rematerialización, cuándo evitarla y qué variante conviene priorizar en cada caso.

En conjunto, el estudio ofrece un marco reproducible para decidir el uso de la rematerialización con fundamento experimental en lugar de hacerlo por defecto. El modelo predictivo obtenido, aunque imperfecto en los rangos de bajo consumo, funciona como una herramienta de decisión confiable precisamente en los escenarios donde la técnica es relevante: el entrenamiento de redes profundas bajo presión de memoria. Sus principales limitaciones, un conjunto de datos desbalanceado y un espacio experimental acotado a un número reducido de arquitecturas, GPU y tamaños de lote, no invalidan los hallazgos, pero sí delimitan su alcance y señalan el camino para trabajos futuros. La contribución central no es un valor numérico concreto, sino la evidencia de que la rematerialización debe aplicarse de forma selectiva y guiada por el contexto, y de que es posible anticipar su efecto con modelos sencillos entrenados sobre datos de perfilado.

6.1 Trabajo Futuro

El presente proyecto sienta una base experimental y metodológica que futuros estudiantes pueden retomar y ampliar en distintas direcciones. Una primera línea natural consiste en extender el dataset incorporando arquitecturas modernas basadas en transformadores (Vision Transformer, BERT) y redes de grafos (GNN), que introducen patrones de activación distintos a los capturados por las CNN clásicas. Ampliar el número de epochs por configuración permitiría además caracterizar el comportamiento de memoria a lo largo del entrenamiento completo, enriqueciendo el modelo predictivo con una dimensión temporal que el presente estudio no aborda.

Una segunda oportunidad de mejora reside en balancear la representación de estrategias en el dataset, igualando el número de configuraciones para checkpoint selectivo, uniform segment y baseline. Este ajuste permitiría entrenar modelos predictivos más precisos en el rango de valores pequeños donde el error relativo actual es más alto. Complementariamente, aplicar transformaciones logarítmicas o entrenar modelos segmentados por familia arquitectónica podría reducir la dispersión de las predicciones en MLP y LeNet-5 sin necesidad de recolectar nuevos datos.

Una tercera dirección de continuación comprende la evaluación sobre hardware más moderno: GPUs con tensor cores (NVIDIA Ampere, Hopper), aceleradores especializados (TPU, IPU) y entornos distribuidos multi-GPU. Dado que la MI210 dispone de 64 GB de VRAM y la Titan X de 12 GB, los resultados evidencian que el impacto de la rematerialización depende fuertemente de la presión de memoria real; explorar hardware intermedio o con distintas jerarquías de caché completaría este mapa de aplicabilidad.

Finalmente, futuros trabajos podrían incorporar técnicas de explicabilidad al modelo predictivo, como SHAP o LIME, para justificar las recomendaciones de forma comprensible ante usuarios finales. También se propone combinar la rematerialización con otras estrategias de optimización de memoria como offloading a CPU, cuantización y mixed precision training, y evaluar su interacción conjunta. La publicación del dataset y los modelos entrenados como benchmark abierto facilitaría que otros equipos de investigación repliquen, comparen y extiendan los resultados aquí obtenidos

Referencias Bibliográficas

Bartan, B., Li, H., Teague, H., Lott, C., & Dilkina, B. (2023). *Moccasin: Efficient tensor rematerialization for neural networks*. arXiv. https://arxiv.org/abs/2304.14463

Beaumont, O., Eyraud-Dubois, L., Herrmann, J., & Lima, A. (2024). Optimal checkpointing for heterogeneous chains: How to train deep neural networks with limited memory. En *Proceedings of the 36th ACM Symposium on Parallelism in Algorithms and Architectures* (pp. 253–263). ACM. https://doi.org/10.1145/3648633

Beaumont, O., Eyraud-Dubois, L., & Shilova, A. (2021). Efficient combination of rematerialization and offloading for training DNNs. En *Advances in Neural Information Processing Systems* (Vol. 34, pp. 23844–23857). Curran Associates.

Breiman, L. (2001). Random forests. *Machine Learning, 45*(1), 5–32. https://doi.org/10.1023/A:1010933404324

Chen, T., Xu, B., Zhang, C., & Guestrin, C. (2016). *Training deep nets with sublinear memory cost*. arXiv. https://arxiv.org/abs/1604.06174

Duan, J., Li, S., Xu, P., Chang, C., He, D., Lin, M., Xu, D., & You, Y. (2024). *Efficient training of large language models on distributed infrastructures: A survey*. arXiv. https://arxiv.org/abs/2407.20018

International Business Machines. (s. f.-a). *Convolutional neural networks*. IBM Think. https://www.ibm.com/es-es/think/topics/convolutional-neural-networks

International Business Machines. (s. f.-b). *¿Qué es el descenso de gradiente?* IBM Think. https://www.ibm.com/es-es/think/topics/gradient-descent

Jaiswal, S. (2025, abril 5). *Multilayer perceptrons in machine learning: A comprehensive guide*. DataCamp. https://www.datacamp.com/es/tutorial/multilayer-perceptrons-in-machine-learning

Kirisame, M., Lyubomirsky, S., Haan, A., Brennan, J., He, M., Roesch, J., Chen, T., & Tatlock, Z. (2021). Dynamic tensor rematerialization. En *Proceedings of the International Conference on Learning Representations (ICLR)*. https://arxiv.org/abs/2006.09616

Kruskal, W. H., & Wallis, W. A. (1952). Use of ranks in one-criterion variance analysis. *Journal of the American Statistical Association, 47*(260), 583–621. https://doi.org/10.1080/01621459.1952.10483441

Levene, H. (1960). Robust tests for equality of variances. En I. Olkin (Ed.), *Contributions to probability and statistics: Essays in honor of Harold Hotelling* (pp. 278–292). Stanford University Press.

Liu, X., Jha, S., & Cheung, A. (2023). *An evaluation of memory optimization methods for training neural networks*. arXiv. https://arxiv.org/abs/2303.14633

Metz, L., Freeman, C. D., Harrison, J., Maheswaranathan, N., & Sohl-Dickstein, J. (2022). *Practical tradeoffs between memory, compute, and performance in learned optimizers*. arXiv. https://arxiv.org/abs/2203.11860

Minakova, S., & Stefanov, T. (2022). Memory-throughput trade-off for CNN-based applications at the edge. *ACM Transactions on Design Automation of Electronic Systems*. https://liacs.leidenuniv.nl/~stefanovtp/pdf/TODAES_22_preprint.pdf

National Institute of Standards and Technology. (s. f.). Kruskal-Wallis test. En *Engineering statistics handbook*. https://www.itl.nist.gov/div898/handbook/eda/section3/eda35a.htm

Navarro, S. (2025, enero 22). *Función de activación en deep learning: Guía 2025*. KeepCoding. https://keepcoding.io/blog/funcion-de-activacion-en-deep-learning/

Networkianos. (s. f.). *Regresión lineal múltiple*. Networkianos. https://networkianos.com/regresion-lineal-multiple/

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., & Duchesnay, É. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research, 12*, 2825–2830.

Ras, A. (2023, octubre 29). *Random forest: Bosque aleatorio. Definición y funcionamiento*. Datascientest. https://datascientest.com/es/random-forest-bosque-aleatorio-definicion-y-funcionamiento

Rodrigo, J. A. (s. f.). *Random forest con Python*. Ciencia de Datos. https://cienciadedatos.net/documentos/py08_random_forest_python.html

Schuler, M., Membarth, R., & Slusallek, P. (2022). *XEngine: Optimal tensor rematerialization for neural networks in heterogeneous environments*. arXiv. https://arxiv.org/abs/2212.09290

Smith, S. (2021, agosto 3). *What's new in PyTorch Profiler 1.9?* PyTorch Blog. https://pytorch.org/blog/pytorch-profiler-1.9-released/

Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., & Rabinovich, A. (2015). Going deeper with convolutions. En *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (pp. 1–9). https://doi.org/10.1109/CVPR.2015.7298594

Tukey, J. W. (1949). Comparing individual means in the analysis of variance. *Biometrics, 5*(2), 99–114. https://doi.org/10.2307/3001913

Zhang, J., Ma, S., Liu, P., & Yuan, J. (2023). *Coop: Memory is not a commodity*. arXiv. https://arxiv.org/abs/2311.00591

Zhao, X., Le Hellard, T., Eyraud-Dubois, L., Gusak, J., & Beaumont, O. (2023). *Rockmate: An efficient, fast, automatic and generic tool for re-materialization in PyTorch*. arXiv. https://arxiv.org/abs/2307.01236