TRABAJO FIN DE GRADO

Doble Grado Ingeniería Informática y Matemáticas

Análisis del algoritmo de gradiente descendente y estudio empírico comparativo con técnicas metaheurísticas

Autor
Eduardo Morales Muñoz

Directores
Pablo Mesejo Santiago
Javier Meri de la Maza

Facultad de Ciencias
Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

—

Granada, 15 de junio de 2025

Declaración de originalidad

Yo, **Eduardo Morales Muñoz**, con DNI 77200029T, declaro explícitamente que el presente trabajo, titulado ***Análisis del algoritmo de gradiente descendente y estudio empírico comparativo con técnicas metaheurísticas***, y presentado como Trabajo de Fin de Grado (TFG), correspondiente al curso académico 2024-2025, es original, entendida esta, en el sentido de que no han sido utilizadas para la elaboración del trabajo fuentes sin citarlas debidamente.

Fdo: Eduardo Morales Muñoz

Granada a 15 de junio de 2025.

D. **Pablo Mesejo Santiago**, Profesor del Departamento de Ciencias de la Computación e Inteligencia Artificial de la Universidad de Granada.

D. **Francisco Javier Meri de la Maza**, Profesor del Departamento de Análisis Matemático de la Universidad de Granada.

Informan:

Que el presente trabajo, titulado ***Análisis del algoritmo de gradiente descendente y estudio empírico comparativo con técnicas metaheurísticas***, ha sido realizado bajo su supervisión por **Eduardo Morales Muñoz**, y autorizamos la defensa de dicho trabajo ante el tribunal que corresponda.

Y para que conste, expiden y firman el presente informe en Granada a 15 de junio de 2025.

Los directores:

Pablo Mesejo Santiago **Francisco Javier Meri de la Maza**

Yo, **Eduardo Morales Muñoz**, alumno de la titulación Doble Grado Ingeniería Informática y Matemáticas de la **Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación de la Universidad de Granada**, con DNI 77200029T, autorizo la ubicación de la siguiente copia de mi Trabajo Fin de Grado en la biblioteca del centro para que pueda ser consultada por las personas que lo deseen.

Fdo: Eduardo Morales Muñoz

Granada a 15 de junio de 2025.

Agradecimientos

Gracias a mi abuelo, por hacerme ser quien soy ahora. Gracias a mis amigos y mi familia por acompañarme en este largo proceso y ofrecerme el apoyo, espacio y entendimiento. Gracias a mis tutores Javier y Pablo.

Índice

Índice de Figuras

Índice de Tablas

Resumen

En el presente Trabajo de Fin de Grado se realiza un estudio integral del algoritmo de gradiente descendente (GD) y su comparación con técnicas metaheurísticas (MH) en el entrenamiento de redes neuronales profundas. La investigación se divide en dos partes: una **teórica-matemática**, centrada en el **análisis del GD y su convergencia**, y otra **empírica-informática**, enfocada en la **evaluación comparativa del rendimiento de GD y MH** en diferentes tareas de aprendizaje profundo.

En la parte matemática, se aborda el algoritmo de GD desde una perspectiva teórica, analizando su funcionamiento, variantes, y problemas asociados a la convergencia. Se exploran las diferentes versiones del algoritmo, así como los elementos clave de su funcionamiento, prestando especial atención a su implementación en redes neuronales mediante el proceso de ***backpropagation***. Se estudia también el **concepto de subgradiente y su aplicación en funciones no diferenciables**.

El análisis de la convergencia se desarrolla utilizando herramientas avanzadas de teoría de probabilidad, como el **teorema de Robbins-Siegmund, el concepto de martingalas** y otros derivados del mismo como las casi supermartingalas. Se demuestra que, **bajo condiciones específicas de convexidad y suavidad, el algoritmo de GD converge al mínimo global en su versión por *batches*** (*Batch Gradient Descent*), mientras que **la versión estocástica** (*Stochastic Gradient Descent*) **requiere un tratamiento probabilístico detallado para caracterizar las condiciones bajo las cuales la convergencia es posible**. Este análisis teórico proporciona una comprensión más profunda de los aspectos matemáticos subyacentes al comportamiento del GD y establece las condiciones necesarias para garantizar una convergencia estable en entornos de alta dimensionalidad y con ruido estocástico.

En la parte informática, se presenta un estudio empírico comparativo entre el algoritmo de GD y técnicas MH para el entrenamiento de redes neuronales profundas. Utilizando **Perceptrones Multicapa** (MLP) para conjuntos de datos tabulares y **Redes Convolucionales** (ConvNets) para análisis de imágenes, se comparan optimizadores basados en GD, como RMSProp, NAG, Adam y AdamW, con algoritmos MH como SHADE y SHADE-ILS, que actualmente representan métodos de referencia en optimización continua [Mar+21]. **Como contribución original, se introducen dos enfoques híbridos meméticos novedosos: SHADE-GD y SHADE-ILS-GD, que combinan técnicas MH con GD**.

La evaluación experimental se estructura en torno a cuatro ejes principales:

- **Impacto de la tarea (clasificación vs. regresión) en el rendimiento de los MLP entrenados con MH**: Entrenamos varios

MLPs usando GD y MH para tareas de clasificación y regresión. Observamos que el rendimiento de los modelos entrenados con GD no varía según el tipo de tarea, medimos el rendimiento relativo de los modelos entrenados con MH respecto a los entrenados con GD. Tras comprobar la no normalidad de los datos, aplicamos el test de los rangos con signo de Wilcoxon y obtenemos un p-valor de 0.023, concluyendo que **existe una diferencia de rendimiento significativa** entre ambos tipos de tareas para los MLPs entrenados con MH, aunque la detección parece bastante sensible a los optimizadores incluidos, por lo que deben tomarse los resultados con cautela.

- **Estudio de factores que afectan el rendimiento en tareas de clasificación, tanto en MLPs como en ConvNets**: Realizamos un análisis de dependencias parciales, y utilizamos como medida de rendimiento la ganancia de *accuracy* con respecto al clasificador aleatorio. De esta manera se determinó que **el tamaño del conjunto de datos es el factor más influyente en el empeoramiento del rendimiento de los modelos entrenados con MH**, mientras que en el caso del entrenamiento **con optimizadores basados en GD el factor más influyente resulta la complejidad del conjunto de datos**, entendiendo por ella factores como el número de clases, el desbalanceo de las mismas, o la cantidad y complejidad de información estructural y semántica dentro de una imagen (para los conjuntos de imágenes).

- **Tiempo de ejecución de las MH**: Analizamos los tiempos de ejecución obtenidos en la experimentación. Comprobamos si sus valores corresponden con los esperados y desglosamos el tiempo del algoritmo. Obtenemos la conclusión de que **las MH requieren de menor tiempo de ejecución por época, pero la necesidad de realizar muchas más épocas (del orden de 100 o 200 veces) para obtener resultados que se puedan acercar a las técnicas clásicas, las convierte en una opción más lenta**. Tanto en GD como en MH, el número de instancias afecta en un factor de entre 7 y 10 veces mayor que el número de parámetros del modelo.

- **Rendimiento general de las técnicas propuestas**: SHADE-GD demostró un mejor rendimiento que SHADE en 17 de las 25 tareas ejecutadas, mientras que SHADE solo destacó en 4 de ellas. Esto hace de **SHADE-GD una versión más consistente y con mejores resultados que el original**. Por otro lado, **SHADE-ILS-GD no presenta una mejora consistente con respecto a SHADE-ILS**. Aunque mejora el rendimiento hasta en un 14 % de *accuracy* en algunas tareas sencillas concretas, es una opción menos estable en redes profundas. Entre las dos propuestas existe un rendimiento similar, aunque

consideramos mejor a SHADE-GD ya que tiene mayor capacidad de generalización, y obtiene mejor resultado en la métrica en 11 tareas, por las 10 en las que lo hace SHADE-ILS-GD.

Este estudio nos ha permitido realizar una aportación que, hasta donde sabemos, es nueva en la literatura referente a optimizadores MH:

- **El rendimiento de los modelos entrenados con MH se ve más afectado por el tamaño del conjunto de datos que por la complejidad del mismo o por el número de parámetros del modelo**. En nuestros experimentos, las MH tienden a obtener peores resultados en conjuntos de datos más grandes, posiblemente debido a su limitada capacidad para extraer patrones complejos cuando el número de iteraciones se mantiene constante.

En conclusión, nuestros resultados respaldan la preferencia por los optimizadores basados en GD frente a las MH en el entrenamiento de modelos profundos, debido a su mayor eficiencia y robustez. Sin embargo, este trabajo aporta una visión más precisa sobre las limitaciones y el potencial de las MH, ofreciendo un marco analítico útil para futuras investigaciones que busquen mejorar su rendimiento o explorar aplicaciones específicas donde puedan ser competitivas.

Parte I
Parte matemática: análisis del gradiente descendente, su convergencia y *backpropagation*

1. Introducción

El **aprendizaje automático** es una rama de la inteligencia artifical en la que **los sistemas son capaces de adquirir conocimiento a partir de datos** [GBC16]. Se dice que un programa aprende de la experiencia E respecto de alguna tarea T y una medición de rendimiento P si su rendimiento en T, medido por P, mejora con la experiencia E [Mit97]. Nos referimos a este programa como modelo. Existen muchos tipos o subramas de aprendizaje automático dependiendo de la naturaleza de esta tarea T y de su medidor de rendimiento P.

El **entrenamiento de un modelo es el proceso de optimizar sus parámetros** (equivalentemente pesos), es decir, su representación interna; para **minimizar una función de coste** (equivalentemente función de error o de pérdida) C que mide el error en el rendimiento. El dominio de dicha función es el espacio de valores que pueden tomar los pesos, normalmente representado de forma tensorial; y su imagen es comúnmente un real no negativo. El objetivo principal del entrenamiento es que el modelo sea capaz de aprender los patrones en un conjunto de datos para luego poder generalizarlos en otros que no ha visto previamente. Diremos que existe un sobreajuste cuando se aprenden los patrones específicos de los datos pero luego no se generaliza bien. La estrategia que usamos para optimizar los pesos es llamada algoritmo de aprendizaje.

El **aprendizaje profundo** es un paradigma del aprendizaje automático en el que los modelos tienen varios niveles de representación obtenidos a través de la composición de módulos sencillos pero comúnmente no lineales, que transforman la representación de los datos sin procesar hacia un nivel de abstracción mayor [LBH15]. Esta rama comenzó a ganar peso en la década de los 2000 y un punto de inflexión fue el resultado de la competición de ImageNet[1] en 2012 [KSH17]. Actualmente este enfoque es el que mejores resultados consigue, siendo una parte fundamental en la investigación y estructura de las grandes compañías tecnológicas y pudiendo ofrecer aplicaciones comerciales a nivel usuario [Sej18; BLH21].

La mayoría de los modelos en aprendizaje automático se entrenan usando técnicas basadas en el algoritmo de aprendizaje de gradiente descendente (equivalentemente descenso del gradiente), ya que es la estrategia que mejores resultados ofrece actualmente en cuanto a capacidad de generalización del modelo y rendimiento computacional [GBC16; Cau09]. Ésta se basa en la idea de que puedo moverme hacia puntos de menor valor en la función de error del modelo realizando pequeños movimientos en sentido contrario a su gradiente como se esquematiza en la Figura 1, con el objetivo de minimizar el valor de salida. Al tratarse de un algoritmo iterativo, **es fundamental estudiar su convergencia**, que depende de varios

[1] http://www.image-net.org/challenges/LSVRC/

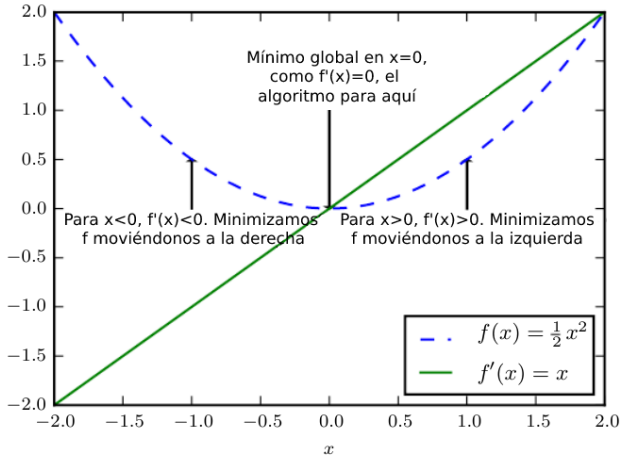

Figura 1: Ejemplo del proceso de optimización mediante el algoritmo de gradiente descendente aplicado a la función $f(x) = \frac{1}{2}x^2$ (curva azul discontinua). La derivada $f'(x) = x$ (línea verde sólida) indica la pendiente de la función en cada punto. Cuando $x < 0$, entonces $f'(x) < 0$ y el algoritmo se mueve hacia la derecha para minimizar la función. Cuando $x > 0$, entonces $f'(x) > 0$ y el algoritmo se mueve hacia la izquierda. El proceso se detiene en $x = 0$, que es el mínimo global, ya que en este punto la derivada es cero. Este ejemplo ilustra el principio básico del gradiente descendente: ajustar los parámetros en la dirección opuesta al gradiente para minimizar una función de pérdida. Imagen obtenida y traducida de [GBC16]

factores y se enfrenta a diversas dificultades, como veremos en secciones posteriores.

El algoritmo de ***backpropagation*** **(BP)** permite transmitir la información desde la salida de la función de coste hacia atrás en un modelo con varios módulos de abstracción para así poder **computar el gradiente de una manera sencilla y eficiente** [RHW86]. Aunque existen otras posibilidades a la hora de realizar éste cómputo, BP es la más usada y extendida gracias a propiedades como su flexibilidad, eficiencia y escalabilidad, que lo hacen destacar por encima de otras opciones [GBC16].

Dependiendo de la familia de modelos que usemos podremos utilizar una estrategia de aprendizaje distinta, como el caso del *Perceptron* y su *Perceptron Learning Algorithm* [Bis06]. En otros casos como la regresión lineal se usa la estrategia de descenso de gradiente pero el gradiente no tiene por qué calcularse a través de BP. Esto se debe a que en este caso se puede obtener eficientemente a través de librerías matemáticas como *Numpy*[2] en

[2] https://numpy.org/

el caso del lenguaje *Python*[3], ya que esta familia de modelos conlleva menos costo computacional en sus cálculos principalmente debido al escaso número de parámetros en comparación con los de aprendizaje profundo. Para éste sí que es necesario el uso de BP en el caso de que elijamos entrenar mediante gradiente descendente, ya que aunque existen otras alternativas como los métodos numéricos o algunas aproximaciones recientes , no consiguen igualar su rendimiento [LeC+12; GBC16; Nov17; Nøk16].

Otra de las características de este algoritmo para el cálculo del gradiente es que los conceptos en los que se basa son simples: optimización, diferenciación, derivadas parciales y regla de la cadena. Lo cual lo convierte a priori en objeto de estudio accesible. En la práctica, los cálculos que se realizan en esta estrategia se implementan a través de la **diferenciación automática**, que es una técnica más general que extiende a BP y se usa para el cómputo de derivadas de funciones numéricas de una manera eficiente y precisa [Bay+17].

1.1. Motivación

Tenemos pues que el aprendizaje profundo es el paradigma del aprendizaje automático que mejores resultados obtiene actualmente y más desarrollo e investigación está concentrando, y que basa el entrenamiento (una de las partes fundamentales que determinan el rendimiento del modelo, además de su arquitectura) de los modelos casi por completo en el algoritmo de descenso de gradiente, ya que es el que mejores resultados de generalización y rendimiento ofrece. Éste a su vez depende casi enteramente del algoritmo de BP para calcular el gradiente, ya que aunque existen otras alternativas no son realmente viables. Tanto es así que es muy común la confusión entre este algoritmo y el de gradiente descentente, que se suelen tomar por la misma cosa. Queda así clara la importancia que tiene BP en el campo del aprendizaje profundo y por extensión también al aprendizaje automático. También conviene destacar la cantidad de veces que se utiliza ésta técnica durante el entrenamiento de un modelo. Cada vez que se actualizan los pesos debemos calcular el gradiente, y teniendo en cuenta la duración de los entrenamientos de los modelos más grandes (con mayor número de parámetros), **el algoritmo de BP puede ser usado miles de veces durante un entrenamiento**.

La eficiencia, escalabilidad y flexibilidad de BP lo han convertido en la opción por defecto para el entrenamiento basado en gradiente descendente para modelos de aprendizaje profundo, sin embargo no hay que olvidar que no se trata de una tarea sencilla: **la obtención de un mínimo global y la verificación, dado un punto, de que es un mínimo global, se trata de un problema NP-Difícil** generalmente [MK87], por lo que se buscan

[3] https://es.python.org/

estrategias aproximadas capaces de obtener buenas soluciones en tiempos razonables. Uno de los problemas abiertos en el aprendizaje profundo y en el que influye directamente BP es la reducción computacional del entrenamiento: si se ajustan los pesos en un modelo con un número muy alto de parámetros y usando un conjunto de entrenamiento muy grande (que es una tendencia reciente en aprendizaje profundo), los recursos computacionales pueden resultar insuficientes incluso para las grandes compañías, pudiendo requerir de meses para el entrenamiento. Por lo que se necesitan algoritmos más escalables y eficientes para afrontarlo [Dea+12].

Por ello resulta esencial, mientras no existan alternativas viables, poder ofrecer modificaciones a ambos algoritmos para mejorar sus cualidades. **Atendiendo a la cantidad de uso del algoritmo de BP y su extensión en el campo, una pequeña mejora tendría un alcance enorme**. Sin embargo esta línea de investigación no es muy extensa ya que principalmente se buscan alternativas en lugar de mejoras, pudiendo deberse principalmente a que a priori puede parecer una técnica muy enrevesada y compleja. Veremos en el desarrollo de esta parte que esto no es cierto, y que **los principios en los que se basa son muy simples**. Es clave comprender su base teórica, funcionamiento e implementación práctica para poder proponer mejoras.

En este contexto, el estudio de la convergencia del algoritmo de gradiente descendente adquiere una relevancia fundamental. Comprender las condiciones bajo las cuales converge (y en qué medida lo hace hacia soluciones óptimas o subóptimas) permite no solo anticipar su comportamiento en distintos escenarios, sino también diseñar estrategias para mejorar su desempeño. Analizar la convergencia no solo tiene implicaciones teóricas, sino también consecuencias prácticas directas en la eficiencia del entrenamiento, la estabilidad del proceso de optimización y la calidad final del modelo.

1.2. Objetivos

El objetivo principal de esta parte matemática es realizar una investigación sobre los algoritmos de descenso de gradiente y BP, proporcionando una visión detallada acerca de los mismos y su implementación. Para ello se divide este objetivo en dos:

1. Estudiar de manera teórica el gradiente descendente, haciendo énfasis en su convergencia.

2. Explorar el uso de BP para el cálculo del gradiente, analizando su implementación a través de la diferenciación automática.

2. Fundamentos previos

A continuación se definirán los conceptos básicos necesarios con los que se trabajará durante el desarrollo de esta parte. Se tratarán los elementos que se usan en el algoritmo de gradiente descendente y BP. Se presenta únicamente el material estrictamente necesario para comprender el trabajo.

2.1. Cálculo diferencial

Se ha usado para la elaboración de esta sección los apuntes en línea del profesor de la UGR Rafael Payá Albert en su curso de Análisis Matemático I[4]. Salvo otras especificaciones, el material de consulta para el desarrollo de esta parte matemática ha sido el curso en línea de Ciencias de Computación de la universidad Bristish Columbia[5] y los libros Probabilistic Machine Learning [Mur22] y Deep Learning [GBC16].

Los algoritmos de gradiente descendente y BP se basan principalmente en el cálculo diferencial, y el hecho de que no usen herramientas matemáticas demasiado complejas resulta precisamente una de sus virtudes, ya que gracias a la abstracción y a un diseño ingenioso consiguen obtener grandes resultados a partir de operaciones relativamente sencillas. Empezamos con los conceptos más elementales que subyacen durante todo el trabajo.

En lo que sigue se fijan los abiertos $X \subseteq \mathbb{R}^n$, $Y \subseteq \mathbb{R}^m$ y las funciones $f : X \to Y$ y $g : X \to \mathbb{R}$. Salvo que se indique lo contrario utilizaremos la norma euclídea en $\mathbb{R}^n$.

Definición 2.1 (Función diferenciable) *f es diferenciable en el punto $a \in X$ si existe una aplicación lineal y continua $Df(a) \in L(\mathbb{R}^n, \mathbb{R}^m)$, que verifica:*

$$\lim_{x \to a} \frac{\|f(x) - f(a) - Df(a)(x-a)\|}{\|x - a\|} = 0.$$

Decimos que f es diferenciable si es diferenciable en todo punto de su dominio.

Definición 2.2 (Derivada parcial de un campo escalar) *La derivada parcial de g con respecto a la k-ésima variable x_k en el punto $a = (a_1, \ldots, a_n) \in X$ se define como*

$$\frac{\partial g}{\partial x_k}(a) = \lim_{h \to 0} \frac{g(a_1, \ldots, a_{k-1}, a_k + h, a_{k+1}, \ldots, a_n) - g(a_1, \ldots, a_n)}{h}$$

si existe el límite.

[4] https://www.ugr.es/~rpaya/docencia.htm#Analisis
[5] https://www.cs.ubc.ca/~schmidtm/Courses/5XX-S22/

Notamos por $f = (f_1, f_2, \ldots, f_m)$ indicando las m componentes de f que es un campo vectorial definido en X, siendo $f_j = \pi_j \circ f$.

Definición 2.3 (Derivada parcial de un campo vectorial) *f es parcialmente derivable con respecto a la k-ésima variable x_k en $a = (a_1, \ldots, a_n) \in X$ si, y sólo si, lo es $f_j \quad \forall j \in I_m$, en tal caso,*

$$\frac{\partial f}{\partial x_k}(a) = \left(\frac{\partial f_1}{\partial x_k}(a), \ldots, \frac{\partial f_m}{\partial x_k}(a)\right) \in \mathbb{R}^m.$$

f es parcialmente derivable en a si, y sólo si, lo es respecto de todas sus variables.

Definimos ahora los elementos clave del algoritmo de entrenamiento: el vector gradiente y la matriz jacobiana. En el algoritmo de descenso de gradiente, el objetivo principal es calcular el vector gradiente, ya que la función de error de los modelos siempre nos devuelve un escalar, es decir que la dimensión de la imagen es 1. Sin embargo las matrices jacobianas también juegan un papel fundamental ya que para calcular ese vector gradiente, el algoritmo de BP necesita realizar cálculos intermedios, que son las matrices jacobianas asociadas a la salida de las capas ocultas (que tienen mayor dimensionalidad) bien con respecto a los parámetros de la capa o bien con respecto al error de la predicción del modelo. En lo que sigue fijamos $x \in X$.

Definición 2.4 (Vector gradiente) *Cuando h es parcialmente derivable en x, el gradiente de h en x es el vector $\nabla h(x) \in X$ dado por*

$$\nabla h(x) = \left(\frac{\partial h}{\partial x_1}(x), \frac{\partial h}{\partial x_2}(x), \ldots, \frac{\partial h}{\partial x_n}(x)\right).$$

Definición 2.5 (Matriz jacobiana) *Si f es diferenciable en x, la matriz jacobiana es la matriz de la aplicación lineal $Df(x) \in L(\mathbb{R}^n, \mathbb{R}^m)$ y se escribe como J_f. Viene dada por:*

$$J_f(x) = \begin{pmatrix} \frac{\partial f_1}{\partial x_1}(x) & \frac{\partial f_1}{\partial x_2}(x) & \cdots & \frac{\partial f_1}{\partial x_n}(x) \\ \frac{\partial f_2}{\partial x_1}(x) & \frac{\partial f_2}{\partial x_2}(x) & \cdots & \frac{\partial f_2}{\partial x_n}(x) \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1}(x) & \frac{\partial f_m}{\partial x_2}(x) & \cdots & \frac{\partial f_m}{\partial x_n}(x) \end{pmatrix}$$
$$= \begin{pmatrix} \nabla f_1(x) \\ \vdots \\ \nabla f_m(x) \end{pmatrix} = \left(\frac{\partial f}{\partial x_1}(x)^T, \ldots, \frac{\partial f}{\partial x_n}(x)^T\right).$$

Si f es de clase C^2 (derivable dos veces con sus derivadas continuas) y derivamos el gradiente obtenemos una matriz cuadrada simétrica con derivadas parciales de segundo orden, a la que llamamos matriz Hessiana.

Definición 2.6 (Matriz Hessiana) *Definimos la matriz Hessiana de h en x como*

$$\nabla^2 h(x) = \begin{pmatrix} \frac{\partial^2 h}{\partial x_1^2}(x) & \frac{\partial^2 h}{\partial x_1 \partial x_2}(x) & \cdots & \frac{\partial^2 h}{\partial x_1 \partial x_n}(x) \\ \frac{\partial^2 h}{\partial x_2 \partial x_1}(x) & \frac{\partial^2 h}{\partial x_2^2}(x) & \cdots & \frac{\partial^2 h}{\partial x_2 \partial x_n}(x) \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 h}{\partial x_n \partial x_1}(x) & \frac{\partial^2 h}{\partial x_n \partial x_2}(x) & \cdots & \frac{\partial^2 h}{\partial x_n^2}(x) \end{pmatrix}$$

Este es un concepto muy importante del cálculo multivariable y la optimización. Cuando hablemos del gradiente descendente, especialmente de la convergencia, usaremos algunas de sus propiedades, como por ejemplo la aproximación cuadrática para desplazamientos pequeños: $h(x + \Delta x) \approx h(x) + \nabla h(x)^T \Delta x + \frac{1}{2} \Delta x^T \nabla^2 h(x) \Delta x$.

Se presenta a continuación una de las reglas más útiles para el cálculo de diferenciales, que afirma que la composición de aplicaciones preserva la diferenciabilidad. Será parte clave en el desarrollo próximo ya que a los modelos de aprendizaje automático basados en capas podemos describirlos como una función que se descompone en una función por cada capa, por tanto será una herramienta que usaremos continuamente para calcular estas matrices jacobianas y gradientes. Se define $D(X, Y)$ como el espacio de las funciones entre X e Y que son diferenciables.

Teorema 2.7 (Regla de la cadena) *Sea $Z \subseteq \mathbb{R}^p$ un abierto y $g : Y \to Z$. Entonces si f es diferenciable en x y g es diferenciable en $y = f(x)$ se tiene que $g \circ f$ es diferenciable en x con*

$$D(g \circ f)(x) = Dg(y) \circ Df(x) = Dg(f(x)) \circ Df(x).$$

Si $f \in D(X, Y)$ y $g \in D(Y, Z)$, entonces $g \circ f \in D(X, Z)$.

En ocasiones en algunos modelos tenemos que lidiar con funciones que no son diferenciables en un punto, y para poder manejarlas extenderemos el concepto de diferenciabilidad a lo que llamaremos subdiferenciabilidad. Esto se expondrá más adelante ya que son conceptos que no se han explorado a lo largo del grado de matemáticas.

El último concepto, que también resulta de gran importancia en los resultados teóricos sobre la convergencia del gradiente descendente, es el de la condición de Lipschitz, en concreto aplicada al gradiente. No forma parte del cálculo diferencial ya que la condición de Lipschitz no requiere diferenciabilidad, pero lo usaremos en éste ámbito ya que la condición que nos interesa usar está aplicada al gradiente.

Definición 2.8 (Función Lipschitziana) *El campo vectorial f es lipschitziano si existe una constante $M \in \mathbb{R}_0^+$ que verifica:*

$$\|f(x) - f(y)\| \leq M\|x - y\| \qquad \forall x, y \in X. \tag{1}$$

La definición nos dice de manera intuitiva que el gradiente de la función no puede cambiar a una velocidad arbitraria. Decimos que la función f tiene gradiente lipschitziano si la condición anterior se aplica a su gradiente, es decir:

$$\|\nabla f(x) - \nabla f(y)\| \leq M\|x - y\| \qquad \forall x, y \in X. \tag{2}$$

La mínima constante $M_0 = L$ que verifica la desigualdad (1) es denominada la constante de Lipschitz de f y viene definida por

$$L = sup\left\{\frac{\|f(x) - f(y)\|}{\|x - y\|} : x, y \in X, x \neq y\right\}.$$

2.2. Algunos conceptos sobre álgebra lineal

Para demostrar algunos de los resultados fundamentales del TFG, necesitamos algunos conceptos básicos del Álgebra Lineal. Utilizamos como referencia [Axl24]. En lo que sigue, $A \in \mathbb{R}^{n\times n}$ es una matriz cuadrada real.

Definición 2.9 (Autovalores y autovectores) *$\lambda \in \mathbb{C}$ es un autovalor de A si existe un vector no nulo $v \in \mathbb{R}^n$, denominado autovector, que verifica:*

$$Av = \lambda v.$$

De manera equivalente, λ satisface la ecuación característica

$$det(A - \lambda I_n) = 0.$$

Definición 2.10 (Radio espectral) *El radio espectral de la matriz A, $\rho(A)$, se define como el máximo valor absoluto de sus autovalores, es decir,*

$$\rho(A) = máx\{|\lambda| : \lambda \text{ es un valor propio de } A\}.$$

Para matrices simétricas ($A = A^T$), definimos lo siguiente

Definición 2.11 (Matriz semidefinida) *La matriz A es semidefinida positiva, y denotamos por $A \succeq 0$ si se cumple alguna de estas propiedades equivalentes:*

- *Todos sus autovalores son no negativos.*
- $\forall x \in \mathbb{R}^n \quad x^T Ax \geq 0.$

De modo análogo decimos que la matriz A es semidefinida negativa, y denotamos por $A \preceq 0$ si se cumple alguna de estas propiedades equivalentes:

- *Todos sus autovalores cumplen $\lambda_i \leq 0$.*

- $\forall x \in \mathbb{R}^n \quad x^T A x \leq 0$.

Cuando la matriz A es simétrica, se cumplen dos propiedades relevantes:

1. Todos los autovalores y autovectores de A son reales.
2. $\|A\| = \rho(A)$.

2.3. Algunos conceptos sobre probabilidad

Ahora vamos a introducir los conceptos necesarios para el desarrollo teórico relacionado con la versión estocástica del algoritmo de gradiente descendente. Desarrollamos las herramientas necesarias para llegar a las definiciones de esperanza condicionada y convergencia casi segura, necesarias para entender las martingalas. Para esta parte usaremos los apuntes de la asignatura de Probabilidad, los apuntes de la profesora de la UGR Patricia Román Román[6] y el curso de Probabilidad del grupo CDPYE[7]. Empezamos con recordando el concepto de espacio medible, para construir sobre él un espacio de probabilidad:

Definición 2.12 (Espacio medible) *Un espacio medible es un par* $(\Omega, \mathcal{A})$*, donde* Ω *es un conjunto arbitrario y* $\mathcal{A} \subseteq \mathcal{P}(\Omega)$ *tiene estructura de* σ*-álgebra, es decir, verifica que*

- $\mathcal{A} \neq \emptyset \quad y \quad \Omega \in \mathcal{A}$.
- $A \in \mathcal{A} \Rightarrow A^c = \Omega \backslash A \in \mathcal{A}$ *(cerrada bajo complementarios).*
- $A_n \in \mathcal{A} \quad \forall n \in \mathbb{N} \Rightarrow \underset{n\in\mathbb{N}}{\cup} A_n \in \mathcal{A}$ *(cerrada bajo uniones numerables).*

Definición 2.13 (Espacio de probabilidad) *Un espacio de probabilidad es una terna* $(\Omega, \mathcal{A}, P)$ *donde:*

1. Ω *es un conjunto arbitrario.*
2. $\mathcal{A} \subseteq \mathcal{P}(\Omega)$ *tiene estructura de* σ*-álgebra.*
3. $P : \mathcal{A} \to [0, 1]$ *es una función de probabilidad, es decir,*
   - $P(A) \geq 0, \quad \forall A \in \mathcal{A}$.
   - $P(\Omega) = 1$.
   - *Para cualquier sucesión* $\{A_n\}_{n\in\mathbb{N}} \subseteq \mathcal{A}$ *de sucesos disjuntos se tiene* $P\left(\underset{n\in\mathbb{N}}{\cup} A_n\right) = \sum_{n\in\mathbb{N}} P(A_n)$.

[6] https://www.ugr.es/~proman/PyE/Condicionadas.pdf
[7] https://www.ugr.es/~cdpye/CursoProbabilidad/

Decimos que dos sucesos son disjuntos si $A \cap B = 0$. Vemos ahora las funciones medibles y seguidamente las variables aleatorias.

Definición 2.14 (Función medible) *Una función medible (unidimensional) sobre un espacio medible $(\Omega, \mathcal{A})$ es una función $f : \Omega \to \mathbb{R}$ que verifica*

$$f^{-1}(B) \in \mathcal{A}, \quad \forall B \in \mathcal{B}$$

o, equivalentemente, que $f^{-1}(\mathcal{B}) \subseteq \mathcal{A}$, siendo $\mathcal{B}$ la σ-álgebra de Borel en $\mathbb{R}$, es decir, la mínima σ-álgebra que contiene a todos los intervalos.

Definición 2.15 (Variable aleatoria) *Una variable aleatoria sobre un espacio de probabilidad $(\Omega, \mathcal{A}, \mathcal{P})$ es una función medible $X : \Omega \to \mathbb{R}$.*

Definición 2.16 (Función de distribución de una variable aleatoria) *Una función $F_X : \mathbb{R} \to [0, 1]$, definida por*

$$F_X(x) = P(X \leq x) = P_X((-\infty, x]) = P(X \in (-\infty, x]), \quad \forall x \in \mathbb{R},$$

se dice que es función de distribución de la variable aleatoria X.

Una variable aleatoria puede ser discreta o continua. Decimos que es discreta si existe un conjunto numerable $E_X \subseteq \mathbb{R}$, tal que $P_X(X \in E_X) = 1$. Para el objetivo que nos interesa sólo vamos a necesitar del tipo discreto, por lo que nos ceñiremos a ellas y, en caso de no especificar, nos estaremos refiriendo a una variable aleatoria discreta. Pasamos a presentar el último concepto intermedio antes de llegar a la esperanza matemática.

Definición 2.17 (Función masa de probabilidad) *Una variable aleatoria discreta $X : (\Omega, \mathcal{A}, P) \to (\mathbb{R}, \mathcal{B}, P_X)$ tiene como función masa de probabilidad*

$$p_X : E_X \to [0, 1], \quad p_X(x) = P(X = x), \quad \forall x \in E_X.$$

Tenemos entonces que $\sum_{x \in E_X} p_X(x) = 1$.

Definimos ahora la esperanza matemática, que además de servirnos para la posterior definición de martingala, la usaremos en la Sección 4.4.2 para intentar controlar la explosión y desvanecimiento del gradiente en BP a través de la inicialización de los pesos.

Definición 2.18 (Esperanza matemática) *Si para la variable aleatoria X existe $\sum_{x \in E_X} |x| p_X(x) < \infty$, entonces se define la esperanza matemática de X como:*

$$E[X] = \sum_{x \in E_X} x p_X(x) = \sum_{x \in E_X} x P(X = x).$$

Dadas dos variables aleatorias X e Y, vamos a ver las dos propiedades más importantes de la esperanza matemática:

- Linealidad: $E[aX + bY] = aE[X] + bE[Y], \quad \forall a, b \in \mathbb{R}$.
- Conservación del orden: $X \leq Y \Rightarrow E[X] \leq E[Y]$.

Ahora vamos a presentar las distribuciones condicionadas para, seguidamente, definir finalmente la esperanza condicionada.

Definición 2.19 (Distribución condicionada) *Sea $X = (X_1, \ldots, X_n)$ un vector de variables aleatorias discretas sobre el mismo espacio de probabilidad. Sea X_i una componente arbitraria y $x_i \in \mathbb{R}$ tal que $P(X_i = x_i) > 0$. Se define la distribución condicionada de $(X_1, \ldots, X_{i-1}, X_{i+1}, \ldots, X_n)$ a $X_i = x_i$ como la determinada por la función masa de probabilidad*

$$P(X_1 = x_1, \ldots, X_{i-1} = x_{i-1}, X_{i+1} = x_{i+1}, \ldots, X_n = x_n | X_i = x_i) = \frac{P(X_1 = x_1, \ldots, X_{i-1} = x_{i-1}, X_i = x_i, X_{i+1} = x_{i+1}, \ldots, X_n = x_n)}{P(X_i = x_i)}$$
$$\forall (x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n) / (x_1, \ldots, x_{i-1}, x_i, x_{i+1}, \ldots, x_n) \in E.$$

Vamos a definir la esperanza condicionada de una variable a otra (caso bidimensional), pero para más variables sería un razonamiento análogo.

Definición 2.20 (Esperanza condicionada) *Sean X e Y variables aleatorias discretas sobre el mismo espacio de probabilidad. Se define la esperanza matemática condicionada de X a Y como la variable aleatoria, que se denota $E[X|Y]$, que toma el valor $E[X|Y = y]$ cuando $Y = y$, siendo*

$$E[X|Y = y] = \sum_{x \in E_X} xP(X = x|Y = y) \quad \textit{si } P(Y = y) > 0. \tag{3}$$

Ahora definimos el concepto de casi seguridad, necesario también para el mismo desarrollo teórico. Se dice que un suceso es casi seguro cuando la probabilidad de que ocurra es uno. Dada una variable aleatoria X, y una sucesión de variables aleatorias $\{X_n\}_{n \in \mathbb{N}}$ definidas sobre un mismo espacio de probabilidad, se dice que dicha sucesión converge casi seguramente a X si

$$P\left(\lim_{n \to +\infty} X_n = X \right) = 1.$$

Ahora vamos a definir un último concepto, que nos servirá para tratar con las versiones estocásticas del gradiente descendente. Es un proceso aleatorio que evoluciona con el tiempo. Más concretamente, un proceso estocástico

es una colección de variables aleatorias X_t indexadas por el tiempo. Si el tiempo es un subconjunto de los enteros no negativos $\{0, 1, 2, \ldots\}$ entonces llamaremos al proceso discreto, mientras que si es un subconjunto de $[0, \infty)$ entonces trataremos con un proceso estocástico continuo. Las variables aleatorias X_t toman valores en un conjunto que llamamos espacio de estados. Este espacio de estados puede ser a su vez discreto, si es un conjunto finito o infinito numerable; o continuo, por ejemplo el conjunto de los números reales $\mathbb{R}$ o un espacio d-dimensional $\mathbb{R}^d$.

3. Gradiente Descendente

Se trata de un algoritmo de aprendizaje iterativo clásico, basado en el método de optimización para funciones lineales de Cauchy. Haskell Curry lo estudió por primera vez para optimización no lineal en 1944 [Cur44], siendo ampliamente usado a partir de las décadas de 1950-1960. Actualmente se trata de **la estrategia de entrenamiento de modelos más ampliamente usada, especialmente en los modelos de aprendizaje profundo, siendo la estrategia que mejores resultados consigue en cuanto a capacidad de generalización de los modelos y eficiencia computacional gracias a su aplicación a través del algoritmo de BP**. Como muestra de su gran adopción, en la Figura 2 vemos la cantidad de artículos indexados al año en la base de datos Scopus[8], con la siguiente búsqueda realizada a 23 de marzo de 2025:

```
( optimization OR learning ) AND ( ( gradient AND descent )
OR ( steepest AND descent ) )
```

Sin embargo a nivel práctico no se usa en su versión original, sino que a lo largo del tiempo han ido surgiendo numerosas modificaciones con el objetivo de mejorar el algoritmo en diversos ámbitos: aumento de la estabilidad y la velocidad de convergencia, reducción computacional del entrenamiento, capacidad de evitar mínimos locales, etc. La literatura en este sentido es extensa, es claro que el gradiente descendente sigue siendo la mejor estrategia de optimización de parámetros de un modelo de forma general [Mar+21], aunque la elección del algoritmo de optimización concreto y de su ajuste depende del problema particular que estemos tratando y generalmente se realiza de manera experimental.

Debemos ver que **el entrenamiento de los modelos, está intrínsecamente ligado a la optimización, en concreto a la minimización de la función de coste** C. Este no es un problema sencillo, y como se ha mencionado antes **se trata de un problema NP-Difícil**, por tanto de existir algoritmos exactos éstos requieren demasiado coste computacional como para utilizarlos en la práctica, por lo que se buscan estrategias aproximadas como el descenso de gradiente para obtener buenas soluciones en un tiempo asequible.

Otro factor a tener en cuenta es la generalización: no es importante únicamente obtener un error bajo en el entrenamiento sino que se mantenga cuando usamos datos de entrada nuevos, ya que nuestro objetivo es ser capaces de encontrar patrones que podamos aplicar en situaciones nuevas y no ajustar el modelo a unos datos dados.

[8] https://www.scopus.com/

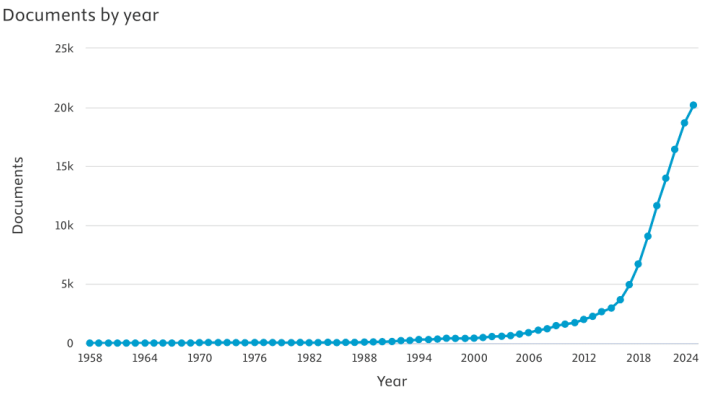

Figura 2: Evolución del número de artículos indexados anualmente en la base de datos Scopus relacionados con optimización y aprendizaje mediante métodos de descenso de gradiente. La consulta se realizó el 23 de marzo de 2025. Desde el primer registro en 1958, el total de documentos acumulados asciende a 135,000. Se observa un crecimiento exponencial en la cantidad de publicaciones a partir de aproximadamente 2010, alcanzando más de 20,000 documentos en 2024.

3.1. Gradiente descendente de Cauchy

Procedemos a describir el método original de descenso de gradiente, propuesto en 1847 por Augustin-Louis Cauchy [Cau09]. Es una versión más primitiva y limitada que sus desarrollos posteriores pero que nos permite obtener de forma más sencilla una visión de su funcionamiento.

Fijamos $f : \mathbb{R}^n \to \mathbb{R}_0^+$ una función continua que no toma valores negativos. Sea $x = (x_1, \ldots, x_n) \in \mathbb{R}^n$. Si queremos encontrar los valores de $x_1, \ldots, x_n$ que verifican $f(x) = 0$, que suponemos que existen, bastará con hacer decrecer indefinidamente los valores de la función f hasta que sean muy cercanos a 0.

Fijamos ahora unos valores concretos $x_0 \in \mathbb{R}^n$, $u = f(x_0)$, $Du = (D_{x_1}u, D_{x_2}u, \ldots, D_{x_n}u)$. Si tomamos $x_0' = x_0 + \epsilon$ tendremos:

$$f(x_0') = f(x_0 + \epsilon) = u + \epsilon Du$$

Sea ahora $\eta > 0$, tomando $\epsilon = -\eta Du$ con la fórmula anterior tenemos:

$$f(x_0') = f(x_0 + \epsilon) = u - \eta \sum_{i=1}^{n} (D_{x_i}u)^2$$

Por tanto hemos obtenido un decremento en el valor de la función f modificando los valores de sus variables en sentido contrario al gradiente, para η suficientemente pequeño. El objetivo de la estrategia es repetir esta operación hasta que se desvanezca el valor de la función f.

3.2. Gradiente descendente en el entrenamiento de modelos

En el caso del entrenamiento de modelos la función que debemos minimizar es la función de coste C, que efectivamente es continua por ser composición de funciones continuas, como se verá más adelante. Como no podemos realizar un cálculo continuo para comprobar con qué valores de η la función decrece, lo hacemos de manera iterativa, y a este η lo llamamos ratio de aprendizaje o más comúnmente ***learning rate***.

Si $C(W)$ es la función de coste del modelo y W representa los parámetros del modelo, entonces **la regla iterativa de actualización de los pesos en la estrategia del descenso del gradiente es la siguiente**:

$$W_{t+1} = W_t - \eta \nabla C(W) \tag{4}$$

En su descripción original, el gradiente se calcula usando todos los datos de entrenamiento, pero en versiones posteriores se propone dividir el conjunto de entrenamiento en varios subconjuntos disjuntos, denominados ***batches***. Cada vez que se calcula el gradiente se actualizan los pesos del modelo, y denominamos a esto una **iteración**. Cada vez que se usan todos los datos de entrenamiento para calcular el gradiente, ya sea tras una sola iteración usando todo el conjunto de entrenamiento o varias si dividimos en *batches*, lo denominamos **época**.

3.2.1. Estrategias de gradiente descendente

En base a los *batches* en que dividamos el conjunto de entrenamiento tenemos varios tipos de gradiente descendente [GBC16].

- ***Batch Gradient Descent*** (BGD): tenemos un único *batch*, cada iteración se corresponde con una época. Calculamos el gradiente usando todo el conjunto de entrenamiento. Esto ofrece un comportamiento mejor estudiado a nivel teórico, con más resultados demostrados; pero aumenta mucho el coste computacional del entrenamiento hasta el punto de que lo vuelve demasiado lento para ser usado en la práctica.
- ***Stochastic Gradient Descent*** (SGD): Actualiza los pesos calculando el gradiente con sólo un elemento del conjunto de entrenamiento. Cada época tiene tantas iteraciones como número de elementos haya en el conjunto de entrenamiento. Esta estrategia introduce ruido en el entrenamiento ya que el gradiente se calcula de una manera aproximada, aunque esto tiene un efecto positivo ya que al provocar más irregularidad en la trayectoria de convergencia es más probable poder escapar de mínimos locales. Además es más eficiente computacionalmente que el anterior y converge más rápido en la práctica.

- ***Mini-Batch Gradient Descent*** (MBGD): Se divide el conjunto de entrenamiento en M *batches* disjuntos de tamaño fijo, y se calcula el gradiente con cada *batch*, por lo que habrá M iteraciones en cada época. Se consigue una aproximación del gradiente con menos error al usar más datos para su cálculo y además se siguen manteniendo las propiedades que veíamos en la anterior estrategia. Es más eficiente que la anterior al conllevar menos actualizaciones de pesos. Es prácticamente la única estrategia utilizada en la realidad ya que ofrece la mayor eficiencia computacional, estabilidad y rapidez en la convergencia.

En estos dos últimos casos, **el conjunto de entrenamiento no permanece fijo**, por lo que la regla de actualización de los pesos que hemos visto en (4) quedaría de la siguiente manera:

$$W_{t+1} = W_t - \eta \nabla C(X_{t+1}, W_t) \tag{5}$$

Aunque la política para computar el gradiente sea distinta en estos 3 tipos, los englobaremos dentro de lo que denominaremos el algoritmo de gradiente descendente original, ya que existen varias modificaciones del algoritmo que aportan mejoras a través de modificar la regla de actualización de los pesos y no solo la cantidad de datos con la que se aproxima el gradiente.

3.2.2. *Learning rate*

El parámetro η que observamos en la ecuación (4) del gradiente descendente se denomina ***learning rate*** y lo usamos para controlar la convergencia reduciendo el efecto de la magnitud del gradiente en la actualización de los parámetros. Este valor es positivo y situado en la práctica alrededor de 0.01 y 0.001 usualmente, aunque para su elección conviene realizar un análisis teórico previo o realizar pruebas prácticas (mucho más común) para elegir un valor adecuado. Este tipo de parámetros, que no son parte del modelo sino del algoritmo de aprendizaje, se denominan **hiperparámetros**. Dependiendo del tipo de algoritmo o modificación del mismo que usemos habrá diferentes hiperparámetros, siendo el *learning rate* el más importante de manera general, ya que **de su valor dependerá en gran parte la convergencia del algoritmo**, pudiendo hacer que converja demasiado lento o que directamente diverja, como podemos observar en la Figura 3 o en resultados sobre la convergencia en la Sección 3.4.

En cuanto a la selección de los hiperparámetros, no se enfoca como un problema donde se busque el óptimo de estos valores ya que la mayoría no son tan decisivos en la convergencia como el *learning rate*, y se ofrecen valores teóricos en sus artículos de presentación que funcionan bien en casos generales. Si bien la convergencia es sensible a los valores iniciales de estos hiperparámetros que se tratan de optimizar a nivel experimental a través del

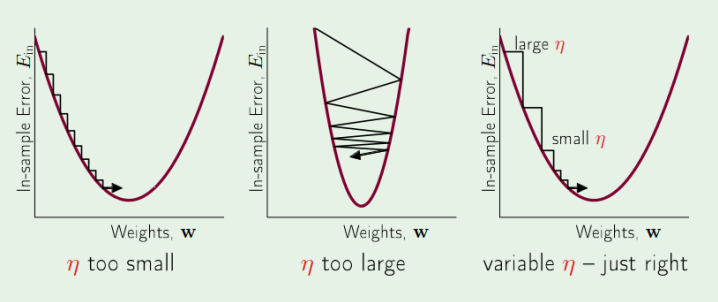

Figura 3: Efecto del tamaño del *learning rate* (η) en el comportamiento del algoritmo de gradiente descendente. La figura ilustra tres escenarios distintos según el valor del *learning rate* (η). *Izquierda*: Un valor de η demasiado pequeño provoca una convergencia extremadamente lenta, con pasos muy cortos que tardan en alcanzar el mínimo del error de entrenamiento E_{in}. *Centro*: Un η demasiado grande genera oscilaciones e incluso puede impedir la convergencia, ya que los pasos del algoritmo sobrepasan el mínimo óptimo. *Derecha*: Un *learning rate* variable, que comienza con un valor grande y se reduce progresivamente, permite una convergencia eficiente y estable, combinando rapidez en la fase inicial y precisión cerca del óptimo. Imagen obtenida de [AML12]

ensayo y error, aunque no se realiza una búsqueda exhaustiva, invirtiéndose muchos más recursos computacionales en el entrenamiento.

Una táctica habitual es usar una política de *learning rate* que decrezca conforme avanza el entrenamiento, de manera que el algoritmo avance con pasos más grandes cuando aún está lejos de un óptimo, con un objetivo explorador, y con pasos más pequeños cuando se va acercando, con un objetivo explotador, procurando una convergencia más estable. [GBC16]. Otro enfoque común es tener un vector de *learning rate* en lugar de un solo escalar, teniendo un valor para cada peso del modelo.

3.3. Subgradientes

Con el objetivo central de calcular el gradiente es lógico pensar que necesitamos ciertas condiciones de diferenciabilidad, aunque sean mínimas, para poder calcular el gradiente que necesitamos. Vamos a presentar el concepto de **subgradiente**, que proporciona una definición más flexible que la del gradiente y en la que nos podemos apoyar para el algoritmo de gradiente descendente.

Podemos pensar en un modelo como una composición de la suma y producto de operaciones lineales con operaciones no lineales (funciones de activación), y componiendo ésta con la función de coste del modelo obtendríamos la función $f : X \times \Omega \times Y \to \mathbb{R}^+$, que recibe los pesos del modelo, los datos de

entrada y sus etiquetas correctas para proporcionar el error del modelo. Esta es la función que necesitaríamos que fuera diferenciable. Las operaciones lineales preservan la diferenciabilidad, y la composición de funciones diferenciables es diferenciable, por lo que si la función de pérdida y las funciones de activación son diferenciables, no tendremos ningún problema a la hora de calcular el gradiente.

Las funciones de coste son diferenciables de manera general, y la más común para problemas de clasificación es la entropía cruzada (o *Cross Entropy*, CE, en inglés), mientras que para regresión son comunes el error cuadrático medio (ECM) y el error absoluto medio (EAM).

- **ECM**: $\frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$
- **EAM**: $\frac{1}{N}\sum_{i=1}^{N}|y_i - \hat{y}_i|$
- **CE**: $-\sum_c \hat{y}_{i,c} log(\frac{e^{y_{i,c}}}{\sum_{c'=1}^{C} e^{y_{i,c'}}})$

En regresión $\hat{y}_i$ es el valor real e y_i es el predicho por el modelo para el dato i que será un real en ambos casos. En clasificación $\hat{y}_{i,c}$ es la etiqueta real del dato i para la clase c, que valdrá 1 en caso de que el dato pertenezca a la clase c y 0 en caso contrario, e $y_{i,c} \in [0, 1]$ representa la probabilidad predicha por el modelo de que el dato i pertenezca a la clase c. Finalmente N es el número de datos y C el número de clases.

Hasta el año 2010, las funciones de activación más comunes para las capas ocultas eran la función sigmoide y la tangente hiperbólica. Estas funciones son diferenciables por lo que su uso no suponía ningún problema en la aplicación del descenso de gradiente [LeC+12]. Sobre ese año se empezó a popularizar la función de activación ReLU (*Rectified Linear Unit*), gracias a su simplicidad, reducción de coste computacional y su aparición en modelos ganadores de competiciones de ImageNet como AlexNet en 2012. Desde entonces esta función, junto a algunas de sus variantes que aparecen en la Figura 4 son ampliamente usadas y con buenos resultados. Sin embargo salta a la vista que esta función no es diferenciable.

Vamos a presentar entonces el concepto de subgradiente junto con algunas de sus propiedades, obtenidas de [Bub15], para ver que será una extensión del gradiente (para funciones convexas) que nos permitirá usar el método de gradiente descendente con funciones que no sean diferenciables en algunos puntos pero que sí sean subdiferenciables.

Definición 3.1 (Subgradiente) *Sea $A \subset \mathbb{R}^n$ y $f : A \to \mathbb{R}$, $g \in \mathbb{R}^n$ es un subgradiente de f en $a \in A$ si existe un entorno de a U_a tal que $\forall y \in U_a$ se tiene:*

$$f(a) - f(y) \leq g^T(a - y)$$

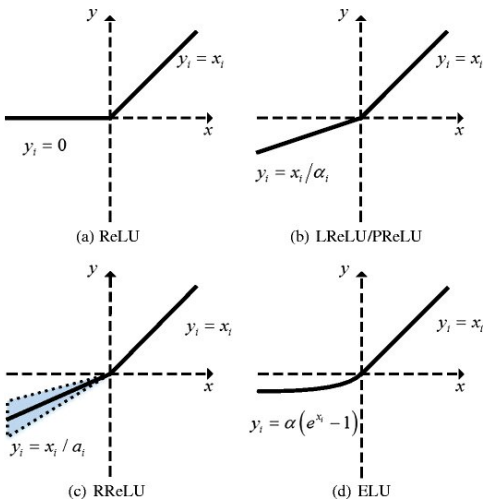

Figura 4: Funciones de activación basadas en ReLU y su impacto en la optimización mediante gradiente descendente. La figura muestra cuatro variantes de la función de activación ReLU ($max(0, x)$), utilizadas en redes neuronales profundas para introducir no linealidad y mejorar la capacidad de representación. (a) ReLU: Mantiene valores positivos sin cambios y asigna cero a valores negativos, lo que puede causar la "neurona muerta"[9] debido a gradientes nulos para $x < 0$. (b) *Leaky ReLU/Parametric ReLU* (LReLU/-PReLU): Introduce una pequeña pendiente controlada por un parámetro α_i en la región negativa para mitigar el problema de neuronas muertas. c) *Randomized ReLU* (RReLU): Usa un coeficiente aleatorio α_i en la parte negativa durante el entrenamiento, introduciendo regularización y variabilidad en la optimización. (d) *Exponential Linear Unit* (ELU): Utiliza una transformación exponencial para valores negativos, asegurando continuidad y derivabilidad en $x = 0$, lo que acelera el entrenamiento y mejora la estabilidad en la actualización de pesos. Imagen obtenida de [Zha+18]

El conjunto de los subgradientes de f en a se denota por $\partial f(a)$. Si existe el subgradiente de f en a, decimos que f es subdiferenciable en a.

Necesitamos también un comportamiento similar al de las funciones diferenciables, en particular necesitamos que las funciones subdiferenciables se preserven a través de las operaciones de suma, multiplicación por escalares y composición.

1. **Multiplicación escalar no negativa**: $\partial(cf) = c \cdot \partial f, c \geq 0$

   Por definición g es un subgradiente de f en x_0 si:

$$f(x) \geq f(x_0) + g^T(x - x_0), \quad \forall x \in U_{x_0}.$$

Multiplicando la desigualdad por $c \geq 0$:

$$cf(x) \geq cf(x_0) + cg^T(x - x_0), \quad \forall x \in U_{x_0}.$$

Por tanto cg es un subgradiente de $h(x) = cf(x)$ en x_0.

2. **Suma**: $\partial(f_1 + f_2)(x) = \partial f_1(x) + \partial f_2(x)$

   Sea g_1 un subgradiente de f_1 y g_2 un subgradiente de f_2, considerando el punto $x_0 \in dom(f_1) \cap dom(f_2)$, por definición tenemos:

$$f_1(x) \geq f_1(x_0) + g_1^T(x - x_0), \quad \forall x \in U_{x_0},$$

$$f_2(x) \geq g_2(x_0) + g_2^T(x - x_0), \quad \forall x \in U_{x_0}.$$

   Sumamos las dos desigualdades para obtener que $g_1 + g_2$ es un subgradiente de $(f_1 + f_2)(x_0)$:

$$f_1(x) + f_2(x) \geq f_1(x_0) + f_2(x_0) + (g_1 + g_2)^T (x - x_0).$$

3. **Composición afín**: Si $h(x) = f(Mx+b) \Rightarrow \partial h(x) = M^T \partial f(Mx+b)$, donde $M \in \mathbb{R}^{m \times n}, b \in \mathbb{R}^m$.

   Tenemos que g es un subgradiente de f en $y_0 = Mx_0 + b$:

$$f(y) \geq f(y_0) + g^T(y - y_0), \quad \forall y \in U_{y_0}.$$

   Tomamos $y = Mx + b$ y, sustituyendo:

$$f(Mx+b) \geq f(Mx_0+b)+g^T(Mx+b-(Mx_0+b)) \quad \forall x : Mx+b \in U_{y_0},$$

$$h(x) = f(Mx+b) \geq h(x_0)+g^T M(x-x_0) \quad for all x \text{tal que} Mx+b \in U_{y_0}.$$

   Por tanto $M^T g$ es un subgradiente de $h(x) = f(Mx + b)$ en x_0.

Tenemos que comprobar que **el subgradiente extiende al gradiente para funciones convexas y diferenciables**, es decir, que cuando existe gradiente entonces existe un único subgradiente y coincide con él. Además hay funciones que no son diferenciables pero sí subdiferenciables. Esto último se hace evidente con el Ejemplo 3.6 de la función ReLU. Vamos a demostrar por tanto que si $f : X \subseteq \mathbb{R}^n \to \mathbb{R}^m$ es diferenciable en el punto $x \in X$ entonces $\partial f(x) = \{\nabla f(x)\}$. Primero recordamos algunas nociones de convexidad:

Definición 3.2 (Conjunto convexo) *Un subconjunto $E \subseteq \mathbb{R}^n$ es convexo cuando, para cualesquiera dos puntos de E, el segmento que los une está contenido en E:*

$$x, y \in E \Rightarrow \{(1-t)x + ty : t \in [0,1]\} \subset E.$$

Definición 3.3 (Función convexa) *Sea $E \subset \mathbb{R}^n$ un conjunto convexo no vacío y sea $f : E \to \mathbb{R}$, f es una función convexa en E si, y solo si:*

$$f((1-t)y + tx) \leq (1-t)f(y) + tf(x), \quad \forall t \in [0,1], \forall x, y \in E.$$

Cuando f es convexa y diferenciable en un entorno U_x de x el gradiente satiface

$$f(y) = f(x) + \nabla f(x)^T (y - x) + o(\|y - x\|) \quad \forall y \in U_x.$$

Por tanto tenemos

$$f(y) \geq f(x) + \nabla f(x)^T (y - x) \quad \forall y \in U_x.$$

Es decir que el gradiente de f en x es también un subgradiente de f en x. Tenemos que $\nabla f(x) \in \partial f(x)$, nos queda demostrar que es único. Para ello vamos a suponer que existe otro subgradiente de f en x, $g \in \partial f(x)$. Sea $u = x + tw$, definimos la función

$$\phi(t) = f(u) - f(x) - g^T(u - x) = f(x + tw) - f(x) - g^T(tw) \geq 0,$$

donde se ha usado que $g \in \partial f(x)$ para ver que es no negativa. Derivamos la función para obtener $\phi'(t) = \nabla f(x)^T w - g^T w$. Vemos que para $t = 0$ se tiene que $\phi(0) = 0$, con lo que hay un mínimo en ese punto. Tenemos por tanto que $\phi'(0) = 0$ o equivalentemente $\nabla f(x)^T w = g^T w$. Como w es arbitrario, concluimos que $g = \nabla f(x)$. Como el gradiente es un subgradiente, y todo subgradiente coincide con él, se tiene que es el único subgradiente de f en x, $\partial f(x) = \{\nabla f(x)\}$.

Ahora vamos a ver cómo se relacionan los subgradientes y las funciones convexas que no necesariamente sean diferenciables.

Proposición 3.4 (Existencia de subgradientes) *Sea $E \subset \mathbb{R}^n$ un conjunto convexo y $f : E \to \mathbb{R}$. Si $\forall x \in E, \partial f(x) \neq \emptyset$ entonces f es una función convexa. Recíprocamente, si f es convexa entonces se tiene que $\forall x \in int(E)$ $\partial f(x) \neq \emptyset$.*

Esta proposición nos asegura que **las funciones convexas siempre tienen subgradiente en su interior**, en particular las funciones de activación convexas como es el caso de la función ReLU. Para demostrarla, primero vamos a necesitar de un teorema en el ámbito de la convexidad:

Teorema 3.5 (Teorema del Hiperplano de apoyo) *Sea $E \subset \mathbb{R}^n$ un conjunto convexo y $x_0 \in Fr(E)$ un puntro de la frontera de E. Entonces, $\exists w \in \mathbb{R}^n, w \neq 0$ tal que*

$$\forall x \in E, \quad w^T x \geq w^T x_0.$$

Demostración de la Proposición 3.4.

Para la primera implicación, queremos probar que si para todo punto $x \in E$ existe al menos un subgradiente de $f(x)$ entonces se verifica

$$f((1-t)x + ty) \leq (1-x)f(x) + tf(y) \quad \forall x, y \in E, t \in [0,1],$$

es decir, que f es convexa. Tomamos $g \in \partial(z)$, $z \in E$ ya que $\forall z \in E, \partial f(z) \neq \emptyset$, y tenemos por la definición de subgradiente:

$$\begin{aligned} f(z) - f(x) &\leq g^T(z-x), \\ f(z) - f(y) &\leq g^T(z-y), \end{aligned} \tag{6}$$

para $x, y \in E$. Tomamos $z = (1-t)x + ty$ y sustituimos:

$$\begin{aligned} f((1-t)x + ty) - f(x) &\leq g^T(((1-t)x + ty) - x), \\ f((1-t)x + ty) + g^T(x - ((1-t)x - ty)) &\leq f(x), \\ f((1-t)x + ty) + g^T(t(x-y)) &\leq f(x), \\ f((1-t)x + ty) + tg^T(x-y) &\leq f(x). \end{aligned} \tag{7}$$

Desarrollando en (6) de manera análoga obtenemos

$$f((1-t)x + ty) + (1-t)g^T(y-x)) \leq f(y). \tag{8}$$

Ahora multiplicamos la desigualdad (7) por $(1-t)$ y la (8) por t, y de su suma obtenemos:

$$\begin{aligned} (1-t)f(x) + tf(y) &\geq (1-t)f((1-t)x + ty) + t(1-t)g^T(x-y) \\ &\quad + tf((1-t)x + ty) + t(1-t)g^T(y-x) \\ &= f((1-t)x + ty) + t(1-t)g^T(x-y) + t(1-t)g^T(y-x) \\ &= f((1-t)x + ty) \end{aligned}$$

donde se ha usado que $g^T(x-y)+g^T(y-x)=0$. Entonces tenemos que $(1-t)f(x)+tf(y) \geq f((1-t)x+ty)$, $\forall x, y \in E$, $t \in [0,1]$. Por tanto f es convexa, como queríamos probar.

Ahora vamos a probar que f tiene algún subgradiente en $int(E)$ si es convexa. Definimos el epigrafo de una función f como

$$epi(f) = \{(x,t) \in E \times \mathbb{R} : t \geq f(x)\}.$$

Es obvio que f es convexa si y sólo si su epigrafo es un conjunto convexo. Vamos a aprovechar esta propiedad y vamos a construir un subgradiente usando un hiperplano de apoyo al epigrafo de la función. Sea $x \in E$, claramente $(x, f(x)) \in Fr(epi(f))$, y $epi(f)$ es un conjunto convexo por ser f convexa. Entonces usando el Teorema del Hiperplano de Apoyo, existe $(a,b) \in \mathbb{R}^n \times \mathbb{R}$ tal que

$$a^T x + bf(x) \geq a^T y + bt, \quad \forall (y,t) \in epi(f). \tag{9}$$

Reordenando tenemos

$$b(f(x)-t) \geq a^T y - a^T x.$$

Como $t \in [f(x), +\infty[$, para que se mantenga la igualdad incluso cuando $t \to \infty$, debe ocurrir que $b \leq 0$. Ahora vamos a asumir que $x \in int(E)$. Entonces tomamos $\epsilon > 0$, verificando que $y = x + \epsilon a \in E$, lo que implica que $b \neq 0$, ya que si $b = 0$ entonces necesariamente $a = 0$. Reescribiendo (9) con $t = f(y)$ obtenemos

$$f(x) - f(y) \leq \frac{1}{|b|} a^T (x-y).$$

Por tanto $\frac{a}{|b|} \in \partial f(x)$, lo que demuestra la otra parte de la implicación.

□

Tenemos entonces que el subgradiente es una extensión del gradiente para funciones convexas en aquellos puntos que no son diferenciables. Por ello podríamos decir que existe el método de descenso de subgradiente, que permite usar funciones que no son diferenciables en todos los puntos, y que se usa de manera implícita en el momento en el que en un modelo se usan funciones de la familia ReLU. Conviene destacar esta diferencia para no perder la rigurosidad, aunque solo sea una formalidad, ya que realmente no se hacen diferencias entre uno y otro método, así que nos seguiremos refiriendo al método de descenso de gradiente aunque estemos trabajando con subgradientes. **En la práctica simplemente se elige un valor predeterminado para la derivada en el punto que estas funciones no son diferenciables**.

Ejemplo 3.6 (Subgradiente de la función ReLU) *La función ReLU es continua en todo el dominio y diferenciable en* $]-\infty, 0[\cup]0, \infty[$. *Su subgradiente es el siguiente:*

$$\nabla ReLU(x) = \begin{cases} 1 & si \quad x \in]0, \infty[\\ c \in [0,1] & si \quad x = 0 \\ 0 & si \quad x \in]-\infty, 0[\end{cases}$$

En [Ber+21] se analiza la **elección del valor que toma el subgradiente** en el punto $x = 0$ y se analiza su influencia en el entrenamiento de modelos. Se discuten varios valores y se observa que **el 0 es el que mejor resultados ofrece** en cuanto al rendimiento de los modelos entrenados.

3.4. Convergencia

La convergencia es un aspecto fundamental en el algoritmo de gradiente descendente. Dado que se trata de un método de optimización iterativo, su objetivo es aproximarse al mínimo global de la función de coste a lo largo de sucesivas iteraciones o, en su defecto, alcanzar un mínimo local que proporcione una solución subóptima. El algoritmo actualiza los parámetros en dirección opuesta al gradiente, por lo que, en caso de converger, lo hará hacia puntos donde el gradiente se aproxima a cero. Dos factores clave en el análisis de su convergencia son la variante del algoritmo utilizada (estocástica o determinista) y la convexidad de la función de coste.

Los estudios teóricos sobre la convergencia del algoritmo de descenso del gradiente son numerosos y diversos [RSS12; DRO15; Mer+20; FGJ20; Tib13]. Sin embargo, su aplicabilidad práctica es limitada, ya que suelen basarse en hipótesis demasiado estrictas sobre la función de coste, las cuales rara vez se cumplen en problemas reales o resultan demasiado costosas de verificar. A pesar de estas limitaciones, dichos desarrollos contribuyen al avance hacia una teoría más aplicable en la práctica, lo que resalta la importancia de comprender sus fundamentos y proponer mejoras. Los principales desafíos para establecer un marco teórico con aplicaciones prácticas son:

- No existen resultados generales que nos permitan conocer el comportamiento de la convergencia del algoritmo en el problema que estemos tratando con un coste computacional asequible. Los resultados son muy específicos y dependen de la función de coste, el valor de los hiperparámetros y la versión del algoritmo de gradiente descendente que estemos utilizando.

- El estudio teórico de la función de coste es muy complejo y requiere muchos recursos computacionales. Por lo tanto, la tendencia a nivel experimental es invertir esos recursos en el entrenamiento, ya que ofrece mejores resultados en relación coste/beneficio de manera general que

el estudio teórico de los elementos del algoritmo. Además, es un procedimiento genérico aplicable en cualquier problema, por lo que resulta más sencillo.

3.4.1. Convergencia para *Batch Gradient Descent*

La convergencia de la versión BGD es menos compleja que en las versiones por *batches*, ya que se emplea la totalidad del conjunto de entrenamiento para calcular el gradiente en cada iteración. A diferencia de las versiones SGD y MBGD, aquí no introducimos ruido estocástico por la aproximación de éste cálculo, por lo que al no haber variabilidad en la estimación del gradiente no se requiere de herramientas de análisis probabilístico. Como resultado, **el estudio de la convergencia de BGD puede abordarse principalmente desde un enfoque determinista, simplificando el tratamiento teórico con respecto a sus versiones estocásticas**.

Analizaremos el caso en que la función de coste sea convexa. En esta situación, probablemente sólo existirá un punto crítico[10] y será un mínimo global, por lo que no es necesario preocuparse por la posibilidad de que el algoritmo se estanque en un mínimo local, ya que, si converge, alcanzará la solución óptima. Además, **el análisis de la convergencia en este contexto es considerablemente más accesible, lo que ha permitido el desarrollo de resultados teóricos más sólidos y abundantes en comparación con el caso no convexo**. No obstante, en la práctica, la función de coste rara vez es convexa [Li+18a], y verificar su convexidad es un problema NP-Difícil [Ahm+11]. Por esta razón, generalmente no se realiza un análisis teórico previo sobre la función ni sobre la convergencia antes de entrenar el modelo.

Antes de enunciar el Teorema 3.9, presentaremos un resultado fundamental para su demostración. Demostraremos esta proposición, ya que, si bien es sencilla, no es trivial.

Proposición 3.7 *Sea $f : \mathbb{R}^n \to \mathbb{R}$ de clase C^2 y convexa, con gradiente globalmente Lipschitz continuo con constante L. Entonces la matriz Hessiana de f está mayorada por LI en el sentido semidefinido positivo, es decir, se tiene que para todo $v \in \mathbb{R}^n$*

$$v^T \nabla^2 f(x) v \leq L v^T v = L\|v\|^2.$$

Equivalentemente, se tiene que todos los autovalores de $\nabla^2 f(x)$ están mayorados por L. Usaremos la notación $\nabla^2 f(x) \preceq LI$.

[10] Si existiera una región donde la función fuera constante, cada punto de la región sería un punto crítico pero esto sería extraordinariamente extraño.

Demostración.

Por definición, la continuidad Lipschitz global del gradiente implica que para cualquier $x, y \in \mathbb{R}^n$,

$$\|\nabla f(y) - \nabla f(x)\| \leq L\|y - x\|.$$

Consideramos un vector unitario $v \in \mathbb{R}^n$ y un real $t > 0$. Para $y = x + tv$, la condición de Lipschitz nos da:

$$\|\nabla f(x + tv) - \nabla f(x)\| \leq L\|tv\| = Lt.$$

Dividiendo en ambos lados por t y tomando el límite cuando $t \to 0$ obtenemos que

$$\|\nabla^2 f(x)v\| \leq L. \tag{10}$$

Por otro lado, como la matriz Hessiana está bien definida por ser f de clase C^2, entonces es simétrica. Ello implica que todos sus autovalores λ_i son reales, y que su norma es igual al radio espectral, es decir,

$$\|\nabla^2 f(x)\| = \rho(\nabla^2 f(x)) = \max\{|\lambda_i|\}.$$

Utilizando que f es convexa, sabemos que su matriz Hessiana es semidefinida positiva, lo que implica que todos sus autovalores son no negativos, y por tanto de la ecuación anterior obtenemos que

$$\|\nabla^2 f(x)\| = \rho(\nabla^2 f(x)) = \max\{\lambda_i\}.$$

Como $\|\nabla^2 f(x)v\| \leq L$ para cualquier vector unitario v, utilizando la igualdad anterior y la desigualdad (10) se tiene que

$$\max\{\lambda_i\} = \|\nabla^2 f(x)\| = \sup_{\|v\|=1} \|\nabla^2 f(x)v\| \leq L.$$

Por tanto concluimos que todo autovalor de $\nabla^2 f(x)$ está mayorado por L, y entonces $\nabla^2 f(x) \preceq LI$.

□

Observación 3.8 *Es importante la globalidad de la continuidad Lipschitz del gradiente. Podemos pensar en la función $f(x) = x^4$, que es de clase C^2, convexa y su gradiente es localmente Lipschitz continuo en cualquier intervalo $[-M, M]$, con constante de Lipschitz $12M^2$. Sin embargo $\nabla f(x)$ no es globalmente Lipschitz continuo, ya que $f''(x) = 12x^2$ no está acotada en $\mathbb{R}$. Por tanto no se verifica la desigualdad de la proposición anterior.*

Ahora sí ya estamos en condiciones de enunciar y demostrar el siguiente teorema clave:

Teorema 3.9 (Convergencia para BGD) *Suponemos la función* $f : \mathbb{R}^n \to \mathbb{R}$ *de clase* C^2 *y convexa, con gradiente globalmente Lipschitz continuo con constante* $L > 0$, $\|\nabla f(x) - \nabla f(y)\|_2 \leq L\|x - y\|_2 \quad \forall x, y \in \mathbb{R}^n$. *Si ejecutamos el algoritmo de gradiente descendente* k *iteraciones con un* $\eta < 1/L$ *constante, el error disminuirá tras cada iteración, llegando a una solución* $x^{(k)}$ *que satisface la siguiente desigualdad:*

$$f(x^{(k)}) - f(x^*) \leq \frac{\|x^{(0)} - x^*\|_2^2}{2\eta k}$$

donde x^* *es el mínimo global de la función de error. Es decir,* $x^{(k)}$ *representa los pesos del modelo en la iteración* k *y* x^* *es el conjunto de parámetros que minimiza la función de error.*

Demostración.

Suponemos que el conjunto de datos con el que entrenamos es constante, por lo tanto el error del modelo, $f(x)$, sólo dependerá de los parámetros x. Partimos de una función $f : \mathbb{R}^n \to \mathbb{R}$ que cumple las hipótesis del teorema.

Al ser $f \in C^2$, usamos la expansión de Taylor de segundo orden alrededor de x:

$$f(y) = f(x) + \nabla f(x)^T(y - x) + \frac{1}{2}(y - x)^T\nabla^2 f(x)(y - x)$$

Como f cumple las condiciones de la Proposición 3.7 entonces se tiene

$$(y - x)^T\nabla^2 f(x)(y - x) \leq Lv^T v = L\|y - x\|^2.$$

Y por tanto:

$$f(y) \leq f(x) + \nabla f(x)^T(y - x) + \frac{1}{2}L\|y - x\|_2^2.$$

Consideramos ahora y como la actualización de los pesos del gradiente descendente, $y = x - \eta\nabla f(x) = x^+$.

$$\begin{aligned}
f(x^+) &\leq f(x) + \nabla f(x)^T(x^+ - x) + \frac{1}{2}L\|x^+ - x\|_2^2 \\
&= f(x) + \nabla f(x)^T(x - \eta\nabla f(x) - x) + \frac{1}{2}L\|x - \eta\nabla f(x) - x\|_2^2 \\
&= f(x) - \eta\nabla f(x)^T\nabla f(x) + \frac{1}{2}L\|\eta\nabla f(x)\|_2^2 \\
&= f(x) - \eta\|\nabla f(x)\|_2^2 + \frac{1}{2}L\eta^2\|\nabla f(x)\|_2^2 \\
&= f(x) - (1 - \frac{1}{2}L\eta)\eta\|\nabla f(x)\|_2^2.
\end{aligned}$$

Usamos $\eta \leq \frac{1}{L}$ para ver que $-(1-\frac{1}{2}L\eta) = \frac{1}{2}L\eta - 1 \leq \frac{1}{2}L(\frac{1}{L}) - 1 = -\frac{1}{2}$, y sustituyendo esta expresión en la desigualdad anterior obtenemos

$$f(x^+) \leq f(x) - \frac{1}{2}\eta\|\nabla f(x)\|_2^2. \tag{11}$$

Esta última desigualdad se traduce en que tras cada iteración del algoritmo del descenso de gradiente el valor del error del modelo es estrictamente decreciente, ya que el valor de $\frac{1}{2}\eta\|\nabla f(x)\|_2^2$ siempre es mayor que 0 a no ser que $\nabla f(x) = 0$, en cuyo caso habremos encontrado el óptimo.

Ahora vamos a acotar el valor del error en la siguiente iteración, $f(x^+)$, en términos del valor óptimo de error $f(x^*)$. Como f es una función convexa se tiene

$$f(x) \leq f(x^*) + \nabla f(x)^T(x - x^*).$$

Sustituyendo en (11) obtenemos

$$\begin{aligned}
f(x^+) &\leq f(x^*) + \nabla f(x)^T(x - x^*) - \frac{\eta}{2}\|\nabla f(x)\|_2^2 \\
f(x^+) - f(x^*) &\leq \frac{1}{2\eta}\left(2\eta\nabla f(x)^T(x - x^*) - \eta^2\|\nabla f(x)\|_2^2\right) \\
f(x^+) - f(x^*) &\leq \frac{1}{2\eta}\left(2\eta\nabla f(x)^T(x - x^*) - \eta^2\|\nabla f(x)\|_2^2 - \|x - x^*\|_2^2 + \|x - x^*\|_2^2\right).
\end{aligned}$$

Como $2\eta\nabla f(x)^T(x-x^*)-\eta^2\|\nabla f(x)\|_2^2-\|x-x^*\|_2^2 = \|x-\eta\nabla f(x)-x^*\|_2^2$, se tiene que

$$f(x^+) - f(x^*) \leq \frac{1}{2\eta}\left(\|x - x^*\|_2^2 - \|x - \eta\nabla f(x) - x^*\|_2^2\right).$$

Usamos ahora la definición de x^+ en esta última desigualdad

$$f(x^+) - f(x^*) \leq \frac{1}{2\eta}\left(\|x - x^*\|_2^2 - \|x^+ - x^*\|_2^2\right).$$

Hacemos la sumatoria sobre las k primeras iteraciones y tenemos

$$\begin{aligned}
\sum_{i=1}^{k}\left(f(x^{(i)}) - f(x^*)\right) &\leq \sum_{i=1}^{k}\frac{1}{2\eta}\left(\|x^{(i-1)} - x^*\|_2^2 - \|x^{(i)} - x^*\|_2^2\right) \\
&= \frac{1}{2\eta}\left(\|x^{(0)} - x^*\|_2^2 - \|x^{(k)} - x^*\|_2^2\right) \\
&\leq \frac{1}{2\eta}\left(\|x^{(0)} - x^*\|_2^2\right).
\end{aligned}$$

El sumatorio de la derecha ha desaparecido ya que es una suma telescópica. Usando que f decrece con cada iteración, e introduciendo la anterior desigualdad, finalmente llegamos a donde queríamos:

$$f(x^{(k)}) - f(x^*) \leq \frac{1}{k}\sum_{i=1}^{k}\Big(f(x^{(i)}) - f(x^*)\Big) \leq \frac{\|x^{(0)} - x^*\|_2^2}{2\eta k}.$$

□

Este teorema garantiza que, **bajo las condiciones establecidas, el algoritmo BGD converge con una tasa de convergencia de** $O(1/k)$. Se trata de un resultado teórico sólido; sin embargo, su aplicabilidad en la práctica es limitada en la mayoría de los casos. En primer lugar, el cálculo exacto de la constante de Lipschitz L es computacionalmente costoso, por lo que el valor de η suele determinarse mediante aproximaciones experimentales.

3.4.2. Convergencia para versiones estocásticas

Podemos obtener un resultado mucho más práctico para versiones estocásticas del algoritmo (recordamos que en la práctica MBGD es la única versión utilizada). Dicho resultado será un poco más débil que el anterior, ya que solo aseguraremos converger a un mínimo local, pero podremos relajar ciertas condiciones de diferenciabilidad y suavidad. A partir de ahora usaremos SGD para referirnos de manera general a las versiones estocásticas del algoritmo de gradiente descendente, tanto SGD como MBGD. Usando la teoría de algoritmos aproximados estocásticos, con el teorema de Robbins-Siegmund tenemos que bajo las siguientes condiciones, cuando la función es convexa se tiene la convergencia casi segura al mínimo global y cuando no lo es hay convergencia casi segura a un mínimo local. Esto nos da un criterio sencillo con el que aseguramos la convergencia, y que no depende de parámetros como la constante de Lipschitz que son complejos de computar.

Usamos como referencia el libro [Law06]. En primer lugar, vamos a introducir los conceptos de martingala, supermartingala y casi supermartingala, que son tipos de procesos estocásticos. Luego enunciaremos un teorema, el de Robbins-Siegmund, que proporciona un fuerte resultado de convergencia para los procesos casi supermartingalas. Demostrando que el algoritmo de SGD es un proceso de este tipo precisamente, enunciaremos el teorema de convergencia para estos algoritmos, demostrándolo en gran parte gracias al teorema de Robbins-Siegmund.

Vamos a hacer la notación un poco más compacta. Si $X_1, X_2, \ldots$ es una sucesión de variables aleatorias usaremos $\mathcal{F}_n$ para denotar "la información contenida en $X_1, \ldots, X_n$". Usamos $E[Y|\mathcal{F}_n]$ en lugar de $E[Y|X_1, \ldots, X_n]$.

Una **martingala** es un modelo de juego justo, que en términos de procesos estocásticos se refiere a **una situación donde el valor esperado de ganancias de un jugador, dada toda la información pasada, permanece inalterado a lo largo del tiempo**. Si X_n representa la riqueza de un jugador en el momento n, el juego se considera justo si $E[X_{n+1}|X_1, X_2, \ldots, X_{n-1}, X_n] = X_n \quad \forall n$, lo que significa que la riqueza futura esperada es igual a la riqueza presente, dado todo el historial previo. Esto implica que no hay una estrategia sistemática para ganar o perder dinero con el tiempo basándose solo en los resultados pasados.

Denotamos por $\{\mathcal{F}_n\}$ una sucesión creciente de información, es decir, para cada n tenemos una sucesión de variables aleatorias $\mathcal{A}_n$ tal que $\mathcal{A}_m \subseteq \mathcal{A}_n$ si $m < n$. La información que tenemos en el momento n es el valor de todas las variables en $\mathcal{A}_n$. La suposición $\mathcal{A}_m \subseteq \mathcal{A}_n$ implica que no perdemos información. Decimos que una variable aleatoria X es $\mathcal{F}_n$-medible si podemos determinar el valor de X en caso de conocer el valor de todas las variables aleatorias en $\mathcal{A}_n$. A menudo está sucesión creciente de información $\mathcal{F}_n$ se denomina filtración.

Decimos que una sucesión de variables aleatorias $M_0, M_1, M_2, \ldots$ con $E[|M_i|] < \infty$ es una martingala con respecto a $\{\mathcal{F}\}_n$ si cada M_n es medible con respecto a $\mathcal{F}_n$ y para cada $m < n$,

$$E[M_n|\mathcal{F}_m] = M_n, \tag{12}$$

o equivalentemente,

$$E[M_n - M_m|\mathcal{F}_m] = 0. \tag{13}$$

La condición $E[|M_i|] < \infty$ es necesaria para garantizar que las esperanzas condicionadas están bien definidas. Si $\mathcal{F}_n$ es la información en variables aleatorias $X_1, \ldots, X_n$ entonces también diremos que $M_0, M_1, \ldots$ es una martingala con respecto a $X_0, X_1, \ldots$. A veces diremos que $M_0, M_1, \ldots$ es una martingala sin hacer referencia a la filtración $\mathcal{F}_n$. En ese caso significará que la sucesión M_n es una martingala con respecto a sí misma.

Un proceso M_n con $E[|M_n|] < \infty$ es una supermartingala con respecto a $\{\mathcal{F}_n\}$ si para cada $m < n$ se tiene $E[M_n|\mathcal{F}_m] \leq M_m$. En otras palabras, una supermartingala es un juego injusto. Si un proceso estocástico no verifica la desigualdad anterior, pero verifica que

$$E[M_n|\mathcal{F}_m] \leq (1 + \beta_m)M_m + \xi_m - \zeta_m$$

para $m < n$ y $\beta_n, \xi_n, \zeta_n \geq 0$ siendo $\mathcal{F}_n$-medibles, **decimos entonces que M_n es una casi supermartingala**, y la denominaremos no negativa si $M_n > 0 \quad \forall n$. El Teorema de Robbins-Siegmund es un resultado clave para la demostración de la convergencia de las versiones estocásticas del gradiente descendente que enunciaremos a continuación. Primero vamos a enunciar el

Teorema de Doob sobre la convergencia de supermartingalas, herramienta clave en la teoría de martingalas.

Teorema 3.10 (Teorema de Convergencia de Supermartingalas) *Si V_n es una supermartingala y está acotada inferiormente casi seguramente, entonces V_n converge casi seguro a una variable aleatoria integrable V_∞.*

Ahora sí, enunciamos y demostramos el teorema de Robbins-Siegmund, que es un **poderoso resultado de convergencia para procesos estocásticos no negativos que son casi supermartingalas**.

Teorema 3.11 (Teorema de Robbins-Siegmund) *Suponemos que V_n es una casi supermartingala no negativa. Si*

$$\sum_{n=1}^{\infty} \beta_n < \infty \quad y \quad \sum_{n=1}^{\infty} \xi_n < \infty \quad \textit{casi seguro,}$$

entonces existe una variable aleatoria no negativa V_∞ que verifica

$$\lim_{n\to\infty} V_n = V_\infty \quad y \quad \sum_{n=1}^{\infty} \zeta < \infty \quad \textit{casi seguro.}$$

Demostración.

Dada V_n una casi supermartingala no negativa, vamos a construir una supermartingala auxiliar a partir de ella y aplicarle el Teorema de Convergencia de Supermartingalas para ver que converge, y posteriormente demostramos que la convergencia de la casi supermartingala está relacionada con la convergencia de nuestra supermartingala auxiliar. Por definición V_n es un proceso estocástico no negativo que verifica la desigualdad

$$E[V_n|\mathcal{F}_m] \leq (1+\beta_m)V_m + \xi_m - \zeta_m \tag{14}$$

para $m < n$ y $\beta_n, \xi_n, \zeta_n \geq 0$ siendo $\mathcal{F}_n$-medibles. A partir de V_n definimos un proceso auxiliar supermartingala W_n, como

$$W_n = V_n + \sum_{k=1}^{n-1} \zeta_k - \sum_{k=1}^{n-1} (\beta_k + \xi_k).$$

Ahora vamos a verificar que W_n es una supermartingala calculando su esperanza condicionada:

$$E[W_{n+1} \mid \mathcal{F}_n] = E\left[V_{n+1} + \sum_{k=1}^{n} \zeta_k - \sum_{k=1}^{n}(\beta_k + \xi_k) \mid \mathcal{F}_n\right]$$
$$= E[V_{n+1} \mid \mathcal{F}_n] + \sum_{k=1}^{n} \zeta_k - \sum_{k=1}^{n}(\beta_k + \xi_k).$$

Como V_n es por hipótesis una casi supermartingala, usamos la desigualdad (14) en la ecuación anterior, y simplificando obtenemos:

$$E[W_{n+1} \mid \mathcal{F}_n] \leq V_n + \beta_n - \zeta_n + \xi_n + \sum_{k=1}^{n} \zeta_k - \sum_{k=1}^{n}(\beta_k + \xi_k)$$
$$= V_n + \sum_{k=1}^{n-1} \zeta_k - \sum_{k=1}^{n-1}(\beta_k + \xi_k) = W_n.$$

Y por tanto W_n es una supermartingala. Ahora vamos a comprobar que está acotada inferiormente para poder aplicar el Teorema 3.10. Como por definición $V_n \geq 0$ y $\sum_{k=1}^{n-1} \zeta_k \geq 0$, tenemos que $W_n \geq -\sum_{k=1}^{n-1}(\beta_k + \xi_k)$. Por la hipótesis del Teorema $\sum_{k=1}^{\infty}(\beta_k + \xi_k) < \infty$ casi seguro, por lo que la cota inferior $-\sum_{k=1}^{n-1}(\beta_k + \xi_k)$ es finita casi seguro. Podemos ahora aplicar el Teorema de Convergencia de Supermartingalas para afirmar que W_n converge casi seguro a W_∞.

Como W_n converge casi seguro y $\sum_{k=1}^{n-1}(\beta_k + \xi_k) < \infty$ casi seguro, de la definición de W_n tenemos que

$$V_n + \sum_{k=1}^{n-1} \zeta_k = W_n + \sum_{k=1}^{n-1}(\beta_k + \xi_k)$$

y que los dos términos del lado derecho convergen casi seguro. Eso significa que $V_n + \sum_{k=1}^{n-1} \zeta_k$ también converge casi seguro a un límite L. Como $\sum_{k=1}^{n-1} \zeta_k$ es una serie no decreciente y acotada casi seguro por convergencia, entonces debe converger a $\sum_{k=1}^{\infty} \zeta_k$. Por tanto V_n converge casi seguro a

$$V_\infty = L - \sum_{k=1}^{\infty} \zeta_k.$$

Como $V_n \geq 0, V_\infty \geq 0$, y $L = V_\infty + \sum_{k=1}^{\infty} \zeta_k$, se sigue que $\sum_{k=1}^{\infty} \zeta_k < \infty$ casi seguro. Concluimos por tanto, que bajo las hipótesis del teorema, existe $V_{\infty \geq 0}$ tal que

$$\lim_{n\to\infty} V_n = V_\infty \quad \text{y} \quad \sum_{n=1}^{\infty} \zeta_n < \infty \quad \text{casi seguro.}$$

□

Ahora **vamos a comprobar que el algoritmo SGD es una casi supermartingala**. Vamos a definir las funciones fuertemente convexas, porque las necesitaremos para este desarrollo.

Definición 3.12 (Función fuertemente convexa) *Sea $E \subset \mathbb{R}^n$ un conjunto convexo no vacío y sea $f : E \to \mathbb{R}$, f es una función fuertemente convexa en E si es estrictamente convexa*[11] *y además se verifica para algún $m \geq 0$:*

$$(\nabla f(x) - \nabla f(y))^T (x - y) \geq m\|x - y\|^2 \quad \forall x, y \in E.$$

Ahora nos fijamos en la minimización, para un conjunto abierto de parámetros W, de la función objetivo

$$H(W) = E[C(X, W)]$$

para una función de pérdida C. Asumimos que $\nabla H(W) = E[\nabla C(X, W)]$ y que

$$G(W) := E[\|\nabla C(X, W)\|^2] \leq A + B\|W\|^2.$$

Asumimos también que $\nabla H(W^*) = 0$ y que

$$(W - W^*)^T \nabla H(W) \geq c\|W - W^*\|^2,$$

lo que implica que W^* es un minimizador único de H, es decir, que es un mínimo global y es único. Esta última condición se mantiene siempre que H sea fuertemente convexa, pero solo necesitamos que se mantenga en W^*.

Añadiendo el *learning rate* como una sucesión en lugar de una constante y expresando las sucesiones como en el Teorema 3.11 para facilitar la comprensión, la regla de actualización de los pesos que vimos en (5) quedaría como

$$W_n = W_{n-1} - \eta_{n-1} \nabla C(X_n, W_{n-1})$$

para una sucesión $X_1, X_2, \ldots$ de variables aleatorias independientes e idénticamente distribuidas. Asumimos que el *learning rate* η_{n-1} puede depender de $X_1, \ldots, X_{n-1}$ y $W_0, \ldots, W_{n-1}$. De esto obtenemos que

$$\begin{aligned} V_n &= \|W_n - W^*\|^2 \\ &= \|W_{n-1} - \eta_{n-1} \nabla C(X_n, W_{n-1}) - W^*\|^2 \\ &= V_{n-1} + \eta_{n-1}^2 \|\nabla C(X_n, W_{n-1})\|^2 - 2\eta_{n-1}(W_{n-1} - W^*)^T \nabla C(X_n, W_{n-1}). \end{aligned}$$

[11] La convexidad estricta es igual que la convexidad normal, pero la desigualdad de su definición es una desigualdad estricta.

Tomando la esperanza condicionada resulta

$$\begin{aligned} E[V_n|\mathcal{F}_{n-1}] &= V_{n-1} + \eta_{n-1}^2 E[\|\nabla C(X_n, W_{n-1})\|^2|\mathcal{F}_{n-1}] \\ &\quad - 2\eta_{n-1}(W_{n-1} - W^*)^T E[\nabla C(X_n, W_{n-1})|\mathcal{F}_{n-1}] \\ &= V_{n-1} + \eta_{n-1}^2 G(W_{n-1}) - 2\eta_{n-1}(W_{n-1} - W^*)^T \nabla H(W_{n-1}) \\ &\leq V_{n-1} + \eta_{n-1}^2 (A + B\|W_{n-1}\|^2) - 2c\eta_{n-1}\|W_{n-1} - W^*\|^2. \end{aligned} \tag{15}$$

Ahora observamos que

$$\begin{aligned} \|W_{n-1}\|^2 &= \|W_{n-1} - W^* + W^*\|^2 \\ &\leq (\|W_{n-1} - W^*\| + \|W^*\|)^2 \\ &\leq 2\|W_{n-1} - W^*\|^2 + 2\|W^*\|^2 \\ &= 2V_{n-1} + 2\|W^*\|^2, \end{aligned}$$

e introduciendo esta desigualdad en (15) obtenemos

$$\begin{aligned} E[V_n|\mathcal{F}_{n-1}] &\leq V_{n-1} + \eta_{n-1}^2 (A + 2BV_{n-1} + 2B\|W^*\|^2) - 2c\eta_{n-1}V_{n-1} \\ &= (1 + 2B\eta_{n-1}^2)V_{n-1} + \eta_{n-1}^2 (A + 2B\|W^*\|^2) - 2c\eta_{n-1}V_{n-1}. \end{aligned}$$

Esto demuestra que V_n es una casi supermartingala con $\beta_n = 2B\eta_n^2$, $\xi_n = \eta_n^2(A + 2B\|W^*\|^2)$ y $\zeta_n = 2c\eta_n V_n$.

Estamos ya en condiciones de enunciar y demostrar el teorema relativo a la convergencia del algoritmo SGD.

Teorema 3.13 (Convergencia de algoritmos SGD) *Con las suposiciones realizadas anteriormente sobre la función de coste C, el proceso V_n converge casi seguro a un límite V_∞ si*

$$\sum_{n=1}^{\infty} \eta_n^2 < \infty \quad \textit{casi seguro.} \tag{16}$$

Si también

$$\sum_{n=1}^{\infty} \eta_n = \infty \quad \textit{casi seguro} \tag{17}$$

entonces el límite es $V_\infty = 0$ y se tiene

$$\lim_{n\to\infty} W_n = W^* \quad \textit{casi seguro.}$$

Demostración.

La primera parte se sigue directamente del teorema de Robbins-Siegmund. Para la segunda, asumimos $V_\infty > 0$ en un conjunto de probabilidad positiva y procedemos por contradicción. Hay entonces una variable aleatoria N tal que en ese conjunto $V_n \geq \frac{V_\infty}{2}$ para $n \geq N$, y

$$\sum_{n=1}^{\infty} \zeta_n = 2c \sum_{n=1}^{\infty} \eta_n V_n \geq c V_\infty \sum_{n=N}^{\infty} \eta_n = \infty$$

con probabilidad positiva. Esto contradice el teorema de Robbins-Siegmund. Concluimos que $V_\infty = 0$ casi seguro, entonces $V_n = \|W_n - W^*\|^2 \to 0$ casi seguro, o

$$W_n \to W^*$$

casi seguro cuando $n \to \infty$.

□

Aunque la suposición de que H es globalmente fuertemente convexa es demasiado fuerte, ya que hace que el teorema no sea útil en la práctica, **podemos esperar que el algoritmo tenga un comportamiento similar en el entorno de un minimizador local W^* si H es fuertemente convexa en ese entorno.**

La conclusión más importante que obtenemos de este resultado es que **el *learning rate* debe tender a cero para asegurarnos la convergencia teórica del algoritmo**. En el Teorema 3.13 la condición (16) nos dice cómo de rápido debe converger, mientras que la condición (17) nos dice que no debe converger demasiado rápido.

Ejemplo 3.14 *Tomamos como valores del learning rate la sucesión $\eta_n = e^{-n}$. Entonces tenemos que el proceso V_n, es decir el algoritmo SGD, converge a V_∞ ya que se cumple la primera condición del teorema. Sin embargo la segunda condición no se cumple, lo que quiere decir que no sabemos si $V_\infty = 0$, por tanto no nos aseguramos converger a un minimizador.*

3.4.3. Problemas en la convergencia

En el Teorema 3.9 tenemos asegurada la convergencia a un mínimo global aunque con unos requisitos que no se suelen encontrar en la práctica. En el Teorema 3.13 nos garantizamos el mismo resultado para versiones estocásticas, con condiciones más estrictas, pero con la intuición de que podemos conseguir convergencia a un minimizador local si las hipótesis planteadas se verifican a nivel local. Encontramos aquí el mayor problema para la convergencia del algoritmo de gradiente descendente: la convergencia prematura a puntos con gradiente muy cercano a cero que no son soluciones subóptimas.

Cuando el algoritmo se aproxima a un punto crítico, la magnitud del gradiente se aproxima a cero, y teniendo en cuenta la regla de actualización de los pesos, $W_{t+1} = W_t - \eta \nabla C(W)$, tenemos por tanto que $W_{t+1} - W_t \approx 0$. Es decir que las modificaciones de los pesos con las actualizaciones serán prácticamente nulas, haciendo que el algoritmo se pare o que progrese de manera muy lenta cerca de estos puntos, lo que en un primer momento podría aparentar una falsa convergencia en regiones planas.

Los puntos críticos más comunes son los puntos de silla, que definimos como un punto x_s que verifica que $\nabla f(x_s) = 0$ pero x_s no es ni un mínimo local ni un máximo local. En x_s la matriz Hessiana de f tiene valores propios tanto positivos como negativos, lo que indica que la función f se curva hacia abajo en unas direcciones y hacia arriba en otras en el punto x_s.

En espacios de alta dimensionalidad, que son comunes en las redes neuronales, la probabilidad de encontrar puntos de silla es mucho mayor que la de encontrar máximos y minímos locales. Para una función $f : \mathbb{R}^n \to \mathbb{R}$, el número de puntos de silla normalmente crece exponencialmente con respecto a la dimensión n. Esto se debe a que la probabilidad de encontrar valores propios de ambos signos en la matriz Hessiana aumenta con la dimensionalidad del espacio de parámetros [Dau+14].

La manera de solventar estos problemas es utilizar modificaciones en el algoritmo de gradiente descendente que proporcionan mejores propiedades a su convergencia, ya que las estrategias estocásticas ofrecen una pequeña pero insuficiente solución a este problema. Al calcular el gradiente mediante una aproximación con un subconjunto de los datos, se introduce un ruido ϵ en su cálculo con lo que $W_{t+1} - W_t \approx \epsilon$, que puede servir para conseguir escapar de ese punto de silla. Dichas modificaciones se denominan optimizadores y a diferencia de las versiones vistas en la Sección 3.2.1, que variaban solo en la cantidad de datos usados para calcular el gradiente, estos optimizadores cambian la regla de actualización de los pesos añadiendo nuevos cálculos, hiperparámetros y estrategias para conseguir que el algoritmo mejore en estabilidad, robustez y velocidad de convergencia.

Existen otros problemas como la explosión o el desvanecimiento del gradiente, pero están ligados a BP como herramienta para calcularlo, por lo que se abordarán en la sección siguiente junto a la inicialización de pesos del modelo, que es la manera principal de superar estos problemas.

4. *Backpropagation*

Ya conocemos el algoritmo de aprendizaje del gradiente descendente y, en esta sección, abordaremos el algoritmo BP, que, como hemos mencionado, es el método más utilizado para calcular el gradiente durante el entrenamiento de un modelo. Cuando se desean obtener las predicciones de una red neuronal para un conjunto de datos de entrada, la información fluye desde la capa de entrada x, atravesando las capas ocultas hasta generar una salida o, que si es evaluada con la función de coste C produce un escalar E que representa el error del modelo. Este proceso, refiriéndonos a cómo se transmite la información entre capas, se conoce como **propagación hacia delante** (*forward propagation*).

Para calcular el gradiente de la función de coste con respecto a los pesos del modelo, es necesario que la información fluya en sentido inverso, es decir, que el error E se propague desde la salida, pasando por las capas ocultas, hasta la capa de entrada x. Este proceso se conoce como **propagación hacia atrás** (*backpropagation*). **El algoritmo de BP toma su nombre de aquí** ya que durante su aplicación necesitamos que la información se propage hacia atrás. Si bien, **no se trata del mismo concepto**, ya que podemos propagar la información hacia atrás sin necesidad de calcular el gradiente, en cuyo caso no estaríamos aplicando BP.

4.1. Diferenciación automática

El algoritmo de BP se implementa en la práctica a través de la diferenciación automática [Bay+17], que es un algoritmo más general para calcular derivadas y que engloba a BP. Se fundamenta en descomponer las funciones en una secuencia de operaciones fundamentales para calcular sus derivadas a través de la regla de la cadena, haciendo este cómputo muy eficiente. Por ello se distingue de la diferenciación simbólica, que manipula las expresiones matemáticas para encontrar derivadas, y de la diferenciación numérica, que calcula las derivadas a través de aproximaciones con diferencias finitas. Esta es la implementación que se usa en las librerías de aprendizaje automático más usadas, como TensorFlow 2[12] y PyTorch[13].

En la diferenciación automática existen dos estrategias para calcular un vector gradiente o una matriz jacobiana: diferenciación hacia delante y diferenciación hacia atrás. Su distinción reside principalmente en si realizamos multiplicaciones de un vector por un jacobiano (hacia atrás) o de un jacobiano por un vector (hacia delante). La elección dependerá de las dimensiones de la matriz jacobiana que queramos calcular, en otras palabras, debemos comparar la dimensión de la entrada y de la salida del modelo. Si la dimensión de entrada es mayor que la de salida, el cálculo de la matriz

[12] https://www.tensorflow.org/
[13] https://pytorch.org/

jacobiana requiere menos operaciones cuando se emplea la diferenciación hacia atrás; por el contrario, si la dimensión de salida es mayor que la de entrada, resulta más eficiente utilizar la diferenciación hacia adelante..

Debido a la estructura general de una red neuronal donde la dimensión de la entrada es mucho mayor que la de la salida, resulta más eficiente calcular el gradiente con la diferenciación hacia atrás, y esto es lo que entendemos como el algoritmo de BP: la información se propaga hacia atrás en el modelo mientras que se usa la diferenciación hacia atrás con el objetivo de calcular el gradiente del error del modelo con respecto a sus pesos. Si utilizáramos la diferenciación hacia delante, aunque estuviéramos propagando la información hacia atrás no estaríamos usando el algoritmo de BP, y además el cálculo resultaría mucho más ineficiente. Se suele afirmar que el algoritmo de BP es una aplicación concreta de la diferenciación automática hecha a medida para el entrenamiento de redes neuronales.

Vamos a explorar el algoritmo de BP de manera progresiva: en primer lugar veremos la diferenciación hacia delante y hacia atrás, viendo por qué es más eficiente usar la segunda y acotando este algoritmo general para llegar al algoritmo de BP, para lo que veremos cómo calcular la matriz jacobiana de la salida de un perceptrón multicapa (MLP, por sus siglas en inglés) respecto a la entrada, en una situación que no será de entrenamiento, ya que no habrá parámetros entrenables, pero nos servirá para ilustrar el funcionamiento de BP. Luego veremos cómo obtenemos el gradiente del error con respecto a los pesos de cada capa usando el algoritmo de BP en un MLP con parámetros entrenables, concretando con ejemplos para las capas más comúnmente utilizadas. Finalmente vamos a generalizar este concepto hacia modelos más abstractos usando grafos dirigidos acíclicos.

Los MLP son un tipo de red neuronal, divididos en capas compuestas de nodos llamados neuronas, donde cada nodo de cada capa está conectado con todos los nodos de la capa siguiente. Son usados principalmente con conjuntos de entrenamiento tabulares, es decir aquellos que sus datos tienen formato de tabla. Usaremos esta arquitectura para realizar el desarrollo teórico ya que es más sencilla conceptualmente y facilita la notación.

4.2. Diferenciación hacia delante vs hacia atrás

Definimos nuestro modelo como $f : \mathbb{R}^n \to \mathbb{R}^m$, $o = f(x)$ con $x \in \mathbb{R}^n$ y $o \in \mathbb{R}^m$. Asumimos que f es una composición de funciones:

$$f = f_k \circ f_{k-1} \circ \cdots \circ f_2 \circ f_1.$$

Donde $k-1$ es el número de capas ocultas del MLP y $k+1$ el total de capas. Cada función f_i representa el cálculo que se realiza en la capa i-ésima. Se tiene para $i \in \{1, \ldots, k\}$

$$f_i : \mathbb{R}^{m_i} \to \mathbb{R}^{m_{i+1}}$$

$$f_i(x_i) = x_{i+1}$$

Donde la entrada del modelo viene representada en la primera capa, $x = x_1$. Además se tiene que $m_1 = n, m_{k+1} = m, x_{k+1} = o$. Para obtener la predicción del modelo, $o = f(x) = f_k(x_k)$, necesitamos calcular el resultado de todas las capas intermedias $x_{i+1} = f_i(x_i)$.

Podemos ver que la matriz jacobiana de la salida con respecto a la entrada $J_f(x) \in \mathbb{R}^{m\times n}$ puede ser calculada usando la regla de la cadena . Esto nos va a servir para ilustrar las diferencias entre la diferenciación hacia atrás y hacia delante

$$J_f(x) = J_{f_k}(x_k)J_{f_{k-1}}(x_{k-1})\cdots J_{f_2}(x_2)J_{f_1}(x_1).$$

Se discute ahora cómo calcular el jacobiano $J_f(x)$ de manera eficiente. Recordamos[14] que

$$\begin{aligned} J_f(x_1) &= \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{pmatrix} \\ &= \begin{pmatrix} \nabla f_1(x)^T \\ \vdots \\ \nabla f_m(x)^T \end{pmatrix} = \begin{pmatrix} \frac{\partial f}{\partial x_1}, \ldots, \frac{\partial f}{\partial x_n} \end{pmatrix} \in \mathbb{R}^{m\times n}. \end{aligned}$$

Donde $\nabla f_i(x)^T \in \mathbb{R}^{1\times n}$ es la fila i-ésima y $\frac{\partial f}{\partial x_j} \in \mathbb{R}^m$ es la columna j-ésima de la matriz jacobiana , para $i = 1, \ldots, m$ y $j = 1, \ldots, n$.

Podemos extraer la fila i-ésima del jacobiano usando un producto vector-jacobiano (PVJ) de la forma $e_i^T J_f(x)$, donde $e_i \in \mathbb{R}^m$ es el vector de la base canónica. De manera análoga se puede extraer la columna j-ésima de $J_f(x)$ usando un producto jacobiano-vector (PJV) de la forma $J_f(x)e_j$, donde $e_j \in \mathbb{R}^n$. Se tiene entonces que el cálculo de la matriz jacobiana $J_f(x)$ equivale a n PJV o m PVJ.

Para construir el jacobiano a partir de operaciones PJV o PVJ, podemos suponer que el cálculo del gradiente de $f_i(x)$ tiene el mismo coste computacional que el cálculo de la derivada parcial de f con respecto de alguna de las variables x_j. Por tanto la forma de cálculo más eficiente de la matriz jacobiana depende de qué valor es mayor: si n o m.

Si $n \leq m$ será más eficiente construir el jacobiano $J_f(x)$ usando PJV de derecha a izquierda.

$$J_f(x)v = J_{f_k}(x_k)\cdots J_{f_2}(x_2)J_{f_1}(x_1)v.$$

[14] Aquí interpretamos los vectores gradiente y derivada parcial como vectores columna

Donde $J_{f_k}(x_k)$ tiene tamaño $m \times m_{k-1}$, $J_{f_i}(x_i)$ tiene tamaño $m_i \times m_{i-1}$ para $i \in 2, \ldots, k-1$ y $J_{f_1}(x_1)$ tiene tamaño $m_1 \times n$ mientras que el vector columna v será $n \times 1$. Esta multiplicación se puede calcular usando el agoritmo de diferenciación hacia delante definido en el Algoritmo 1.

Algorithm 1 Diferenciación hacia delante

$x_1 := x$
for $j \in \{1, \ldots, n\}$ **do**
  $v_j := e_j \in \mathbb{R}^n$
end for
for $i \in \{1, \ldots, k\}$ **do**
  $x_{i+1} := f_i(x_i)$
  for $j \in \{1, \ldots, n\}$ **do**
    $v_j := J_{f_i}(x_i)v_j$
  end for
end for
  return $o = x_{k+1}, (v_1, v_2, \ldots, v_n)$

Donde los elementos $v_j, j \in \{1, \ldots, n\}$ de la matriz fila v se corresponden con las derivadas parciales de la función del MLP respecto a la entrada de la capa j, es decir la columna j-ésima de la matriz jacobiana, $v_j = \frac{\partial f}{\partial x_j}$.

Si $n \geq m$ es más eficiente calcular $J_f(x)$ para cada fila $i = 1, \ldots, m$ usando PVJ de izquierda a derecha. La multiplicación izquierda con un vector fila u^T es

$$u^T J_f(x) = u^T J_{f_k}(x_k) \cdots J_{f_2}(x_2) J_{f_1}(x_1).$$

Donde u^T tiene tamaño $1 \times m$, $J_{f_k}(x_k)$ tiene tamaño $m \times m_{k-1}$, $J_{f_i}(x_i)$ tiene tamaño $m_i \times m_{i-1}$ para $i \in 2, \ldots, k-1$ y $J_{f_1}(x_1)$ tiene tamaño $m_1 \times n$. Esto puede calcularse usando la diferenciación hacia atrás (ver Algoritmo 2).

Donde los elementos u_i^T se corresponden con el gradiente de la función f_i, $u_i^T = \nabla f_i(x)^T$, es decir la fila i-ésima de la matriz jacobiana. Asumimos que $m = 1$ ya que la salida de la función de error del modelo es un escalar, y que $n = m_i$ $i \in \{2, \ldots, k-1\}$; entonces el coste de computar el jacobiano usando la diferenciación hacia atrás es $O(n^2)$. Este algoritmo aplicado al cálculo del gradiente del error del modelo con respecto a los pesos, propagando la información hacia atrás y con el objetivo de usar gradiente descendente, es lo que conocemos como BP. Podemos ver una representación gráfica de ambos modos de diferenciación en la Figura 5.

Con la notación que estamos empleando, cuando $m = 1$ el gradiente $\nabla f(x)$ tiene la misma dimensión que x. Por tanto es un vector columna mientras que $J_f(x)$ es un vector fila, por lo que técnicamente se tiene que $\nabla f(x) = J_f(x)^T$. Es de vital importancia aclarar esto ya que es el caso en

Algorithm 2 Diferenciación en modo reverso

$x_1 := x$
for $k \in \{1, \ldots, K\}$ **do**
  $x_{k+1} = f_k(x_k)$
end for
for $i \in \{1, \ldots, m\}$ **do**
  $u_i := e_i \in \mathbb{R}^m$
end for
for $k \in \{K, \ldots, 1\}$ **do**
  for $i \in \{1, \ldots, m\}$ **do**
    $u_i^T := u_i^T J_{f_k}(x_k)$
  end for
end for
  return $o = x_{k+1}, \left(u_1^T, u_2^T, \ldots, u_m^T\right)$

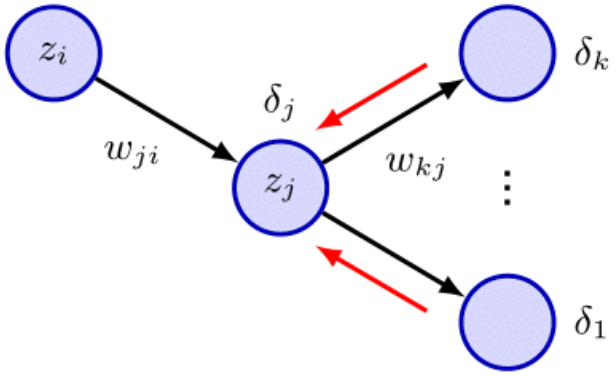

Figura 5: Cálculo del error δ_j para la unidad oculta j mediante la propagación hacia atrás del error (δ). En la imagen, el nodo z_j representa una neurona en una capa oculta de la red neuronal. Esta neurona recibe una entrada desde otra neurona z_i con un peso sináptico asociado w_{ji} (indicado por la flecha negra). A su vez, la neurona z_j está conectada a varias neuronas z_k en la siguiente capa, con pesos w_{kj}. Durante la propagación hacia adelante (flechas negras), la información fluye desde las entradas hacia las salidas de la red. Durante la propagación hacia atrás del error (flechas rojas), los valores de error δ_k de las neuronas de la capa siguiente se propagan hacia atrás para calcular δ_j, lo que permite ajustar los pesos en función del gradiente del error. Imagen obtenida de [BB23].

el que nos situamos cuando usamos BP. La dimensión de salida siempre es uno, ya que calculamos la matriz jacobiana de la función de error del modelo con respecto a los pesos, con lo que será un vector gradiente de dimensión igual a la dimensión de los pesos del modelo. La predicción del modelo puede tener dimensión 1 en tareas de regresión, o una dimensión mayor para tareas de clasificación, aunque de manera general no suele ser mayor de 100. La función de error del modelo siempre tendrá como imagen un valor real.

Hemos demostrado que, para el cálculo de una matriz jacobiana, la diferenciación hacia adelante es más eficiente cuando la dimensión de la entrada es menor que la de la salida, mientras que la diferenciación hacia atrás resulta más adecuada cuando la dimensión de la salida es menor que la de la entrada. Dado que en las redes neuronales y los problemas en los que se aplican la dimensión de salida es, por lo general, considerablemente menor que la de entrada, la diferenciación hacia atrás es la opción más eficiente en estos casos.

4.3. *Backpropagation* en perceptrones multicapa

En la sección anterior, analizamos un modelo que no tenía parámetros entrenables. Ahora, utilizaremos un modelo que sí los tiene y **exploraremos cómo calcular el gradiente de la función de coste con respecto a esos parámetros entrenables**. Los parámetros son valores reales y tienen la forma $W = W_1 \times W_2 \times \cdots \times W_k \subset \Omega$, con $W_i \in \mathbb{R}^{n_i \times n_{i+1}}$ donde n_i es el número de neuronas de la i-ésima capa. El modelo que tendríamos añadiendo los pesos es $f : \mathbb{R}^n \times \Omega \to \mathbb{R}^m$, $o = f(x, W)$ con $x \in \mathbb{R}^n$ y $o \in \mathbb{R}^m$. Donde las funciones de cada capa son de la forma $f_i(x_i, W_i) = \sigma_i(W_i x_i) = x_{i+1}$ donde σ_i es una función de activación generalmente no lineal. Dependiendo del tipo de problema, la función f_k puede ser distinta: en un problema de regresión usamos la identidad, en clasificación usamos la función *Softmax* que convierte el vector de la predicción del modelo en uno cuyos elementos suman 1 y donde el elemento de la posición i-ésima representa la probabilidad de que la entrada pertenezca a la clase i.

Ahora vamos a considerar la función de coste del modelo como una capa más de las funciones de las capas ocultas que nos permiten obtener la predicción. Siguiendo con la notación anterior, incluyendo la función de error $\mathcal{L} : \mathbb{R}^n \times \Omega \times \mathbb{R}^m \to \mathbb{R}$, $E = \mathcal{L}((x, W, y) = C(f(x, W), y)$, con $y \in \mathbb{R}^m$ siendo la etiqueta correcta para la entrada x y $C : \mathbb{R}^m \times \mathbb{R}^m \to \mathbb{R}$ la función de coste del modelo. Con esto tenemos que $\mathcal{L} = C \circ f$.

Ejemplo 4.1 *Suponemos un MLP con dos capas ocultas, la salida escalar (problema de regresión) y una función de pérdida* $C(f(x, W), y) = \frac{1}{2}\|f(x, W) - y\|^2$. *Entonces tenemos* $\mathcal{L} : \mathbb{R}^n \to \mathbb{R}$ *y cada capa tiene ecuación*

$$f_1(x_1, W_1) = \sigma_1(W_1 x_1) = x_2$$
$$f_2(x_2, W_2) = W_2 x_2 = x_3 = f(x, W) = o$$
$$C(f(x, W), y) = \frac{1}{2}\|x_3 - y\|^2 = E$$

El objetivo será calcular el gradiente del error con respecto a los parámetros $\frac{\partial E}{\partial W}$ para poder utilizarlo el entrenamiento a través del gradiente descendente. Buscamos obtener un vector gradiente de la misma dimensión que W, pero el cálculo no es directo, calcularemos progresivamente el gradiente de la función de coste con respecto a los pesos de cada capa, desde la capa final hasta la inicial por lo que buscamos calcular $\frac{\partial E}{\partial W_i}, \forall i = 1, \ldots, k$. Para la última capa $\frac{\partial E}{\partial W_k}$ el cálculo es inmediato, mientras que para el resto podemos usar la regla de la cadena para obtener que

$$\frac{\partial \mathcal{L}}{\partial W_i} = \frac{\partial \mathcal{L}}{\partial x_k}\frac{\partial x_k}{\partial x_{k-1}}\frac{\partial x_{k-1}}{\partial x_{k-2}} \cdots \frac{\partial x_{i+1}}{\partial W_i}$$

Cada $\frac{\partial \mathcal{L}}{\partial W_i} = \left(\nabla_{W_i} \mathcal{L}^T\right)$ es un vector gradiente con el mismo número de elementos que W_i, es decir, es una submatriz de la matriz jacobiana que contiene los gradientes asociados a los pesos de la capa W_i. Estos se calculan propagando hacia atrás la información en el modelo y usando la estrategia de diferenciación hacia atrás a través de PVJ, es decir, el algoritmo de BP que podemos ver en el pseudocódigo del Algoritmo 3.

Algorithm 3 BP para MLP con k capas

```
//Propagación hacia delante
x_1 := x
for l ∈ {1, ..., L} do
    x_{l+1} = f_k(x_l, W_l)
end for
//Propagación hacia atrás
u_{L+1} = 1
for l ∈ {L, ..., 1} do
    g_l := u_{l+1}^T ∂f_l(x_l, W_l)/∂W_l
    u_l^T := u_{l+1}^T ∂f_l(x_l, W_l)/∂x_l
end for
    return {∇_{W_l}; l = 1, ..., L}
```

Podemos ver una esquematización del proceso en la Figura 6. Para tener una idea más profunda y completa acerca del algoritmo de BP, vamos a ver cómo calcular el PVJ para las capas más comunes en los modelos, para lo que analizaremos sus matrices jacobianas respecto de la entrada de la capa.

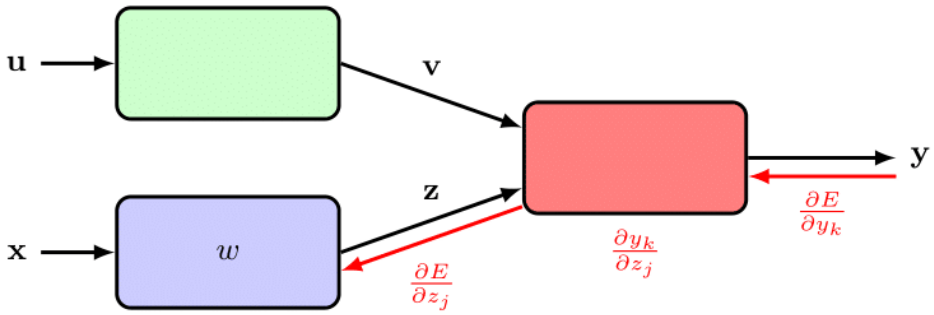

Figura 6: Proceso de *backpropagation*, donde una arquitectura modular de aprendizaje profundo en la que la matriz jacobiana se emplea para propagar hacia atrás las señales de error desde las salidas hasta los módulos anteriores en el sistema. En la imagen, las entradas **u** y **x** se procesan en módulos distintos. El módulo superior recibe **u** y produce una salida intermedia **v**, mientras que el módulo inferior recibe **x**, con un conjunto de pesos representados por w, y genera una salida intermedia **z**. Ambos valores, **v** y **z**, sirven como entrada para un tercer módulo (en rojo), que produce la salida final **y**. Durante la propagación hacia atrás del error (indicada por las flechas rojas), la derivada del error con respecto a la salida, $\frac{\partial E}{\partial y_k}$, se propaga hacia atrás a través de la derivada $\frac{\partial y_k}{\partial z_j}$ hasta obtener $\frac{\partial E}{\partial z_j}$. Posteriormente, este error se retropropaga aún más hasta el módulo anterior, permitiendo la actualización de los pesos w. Imagen obtenida de [BB23].

4.3.1. Capa no-lineal

Consideramos primero una capa que aplica una función no lineal, normalmente el caso de las funciones de activación. $z = \sigma(x)$, con $z^{(i)} = \sigma(x^{(i)})$. El elemento en la posición (i, j) del jacobiano es dado por:

$$\frac{\partial z^{(i)}}{\partial x^{(j)}} = \begin{cases} \sigma'(x^{(i)}) & \text{si } i = j \\ 0 & \text{en otro caso.} \end{cases}$$

Donde $\sigma'(a) = \frac{d\sigma}{da}(a)$. En otras palabras, el jacobiano con respecto de la entrada es

$$J = \frac{\partial \sigma}{\partial x} = diag(\sigma'(x)).$$

Si tomamos como ejemplo la función $ReLU$, para un vector arbitratio u, podemos calcular su PVJ $u^T J$ a través de la multiplicación de elementos de la diagonal de J con el vector u.

$$\sigma(a) = ReLU(a) = max(a, 0),$$

$$\sigma'(a) = \begin{cases} 0 & \text{si } a < 0 \\ 1 & \text{si } a > 0. \end{cases}$$

Como hemos visto en la Sección 3.3 la función $ReLU$ no es diferenciable en el punto 0, pero sí que admite subderivada en todo su dominio, y en el punto $a = 0$ es cualquier valor entre $[0, 1]$, y usualmente en la práctica se toma el valor 0. Por tanto

$$ReLU'(a) = \begin{cases} 0 & \text{si } a \leq 0 \\ 1 & \text{si } a > 0. \end{cases}$$

$$J = \frac{\partial \sigma}{\partial x} = diag(ReLU'(x)).$$

4.3.2. Capa de entropía cruzada

Consideramos ahora la capa cuya función es la función de coste, en concreto con una medición del error usando CE donde tenemos C clases, que toma las predicciones x y las etiquetas y como entrada y devuelve un escalar. Recordamos que en esta capa la matriz jacobiana es un vector fila, identificado con el gradiente, ya que la salida es un escalar.

$$\begin{aligned} z = f(x) &= CE(x, y) \\ &= -\sum_c y_c log(softmax(x)_c) = -\sum_c y_c log(p_c) \end{aligned}$$

donde $p_c = softmax(x)_c = \frac{e^{x_c}}{\sum_{c'=1}^{C} e^{x_{c'}}}$ son las probabilidades de las clases predichas, e y es la etiqueta correcta con codificación *one-hot encoded vector*, es decir un vector de C elementos que representa la clase real a la que pertenece; si ese elemento pertenece a la clase k, todos las posiciones del vector y serán 0 a excepción de la posición k, que será 1. El jacobiano con respecto a la entrada es

$$J = \frac{\partial z}{\partial x} = (p - y)^T \in \mathbb{R}^{1 \times C}.$$

Vamos a asumir que la clase objetivo es la etiqueta c:

$$z = f(x) = -log(p_c) = -log\left(\frac{e^{x_c}}{\sum_j e^{x_j}}\right) = log\left(\sum_j e^{x_j}\right) - x_c.$$

Entonces

$$\frac{\partial z}{\partial x_i} = \frac{\partial}{\partial x_i} log \sum_j e^{x_j} - \frac{\partial}{\partial x_i} x_c = \frac{e^{x_i}}{\sum_j e^{x_j}} - \frac{\partial}{\partial x_i} x_c = p_i - \mathbb{I}(i = c).$$

4.3.3. Capa lineal

Consideramos por último una capa lineal $z = f(x, W) = Wx$, donde $W \in \mathbb{R}^{m\times n}$, con $x \in \mathbb{R}^n$ y $z \in \mathbb{R}^m$ son respectivamente la entrada y la salida de esa capa.

Conviene aclarar, para evitar confusiones, que en la descripción previa hemos considerado las capas ocultas como una combinación de las operaciones lineales que aquí se describen con las funciones de activación, aquí sin embargo las analizamos por separado con el objetivo de una descripción más sencilla y un análisis más individualizado. Esta agrupación es una abstracción y por tanto no varía en cuanto a resultados.

Como z es lineal, el jacobiano de la función de la capa con respecto al vector de entrada de esa capa coincide con su matriz de coeficientes, $\frac{\partial z}{\partial x} = W$

El PVJ entre $u^T \in \mathbb{R}^{1\times m}$ y $J \in \mathbb{R}^{m\times n}$ es

$$u^T \frac{\partial z}{\partial x} = u^T W \in \mathbb{R}^{1\times n}.$$

Ahora consideramos el jacobiano con respecto a la matriz de los pesos, $J = \frac{\partial z}{\partial W}$. Esto se puede representar como una matriz de tamaño $m \times (m \times n)$, que resulta compleja de manejar. Por tanto en lugar de eso veremos de manera individual cómo calcular el gradiente con respecto a un único peso W_{ij}. Esto es más sencillo de calcular ya que $\frac{\partial z}{\partial W_{ij}}$ es un vector. Para su cómputo nos fijamos en que

$$z_l = \sum_{t=1}^{n} W_{lt} x_t, \qquad \text{y}$$

$$\frac{\partial z_l}{\partial W_{ij}} = \sum_{t=1}^{n} x_t \frac{\partial}{\partial W_{ij}} W_{lt} = \sum_{t=1}^{n} x_t \mathbb{I}(i = l \text{ y } j = t).$$

Por tanto

$$\frac{\partial z}{\partial W_{ij}} = (0, \ldots, 0, x_j, 0, \ldots, 0)^T$$

Donde el elemento no nulo ocupa la posición i-ésima. El PVJ entre $u^T \in \mathbb{R}^{1\times m}$ y $\frac{\partial z}{\partial W} \in \mathbb{R}^{m\times(m\times n)}$ se puede representar como una matriz de tamaño $1 \times (m \times n)$. Vemos que

$$u^T \frac{\partial z}{\partial W_{ij}} = \sum_{l=1}^{m} u_l \frac{\partial z_l}{\partial W_{ij}} = u_i x_j.$$

Con lo cual

$$u^T \frac{\partial z}{\partial W} = u x^T \in \mathbb{R}^{m\times n}.$$

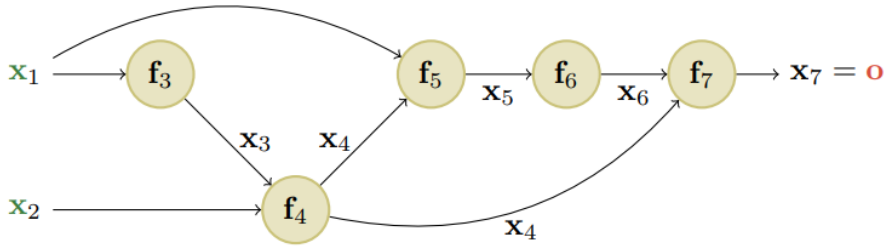

Figura 7: Representación de una red computacional como un grafo dirigido acíclico para diferenciación automática y BP. La figura muestra un grafo dirigido acíclico donde los nodos representan operaciones matemáticas f_i y las aristas indican el flujo de información entre variables intermedias x_i. Las entradas x_1 y x_2 alimentan las funciones f_3 y f_4, generando las variables x_3 y x_4, que a su vez son utilizadas en pasos posteriores. El proceso continúa hasta obtener la salida final x_7. Esta estructura permite calcular eficientemente derivadas mediante diferenciación automática en modo de diferenciación hacia atrás, utilizada en el algoritmo de BP para optimizar redes neuronales profundas. Imagen obtenida de [Mur22].

4.3.4. Grafos computacionales

Los MLP son un tipo de Redes Neuronales Profundas donde cada capa se conecta directamente con la siguiente formando una estructura de cadena. Sin embargo, **las Redes Neuronales Profundas más recientes combinan componentes diferenciables de forma mucho más compleja, creando un grafo computacional de forma similar a como en la programación se combinan funciones simples para hacer otras más complejas**. La restricción es que el grafo resultante debe de ser un grafo acíclico dirigido, donde cada nodo es una función subdiferenciable. Un grafo acíclico dirigido es un tipo de grafo donde cada arista que une dos nodos tiene un sentido específico desde un nodo al otro (dirigido) y en el que no se forman ciclos, es decir, partiendo de un nodo dado no existe una secuencia de aristas dirigidas por la que se pueda volver a él.

Vamos a ver un ejemplo usando la función $f(x_1, x_2) = x_2 e^{x_1} \sqrt{x_1 + x_2 e^{x_1}}$, cuyo grafo se puede ver representado en la Figura 7.

Las funciones intermedias que vemos en el grafo son:

$$
\begin{aligned}
x_3 &= f_3(x_1) = e^{x_1} \\
x_4 &= f_4(x_2, x_3) = x_2 x_3 \\
x_5 &= f_5(x_1, x_4) = x_1 + x_4 \\
x_6 &= f_6(x_5) = \sqrt{x_5} \\
x_7 &= f_7(x_4, x_6) = x_4 x_6.
\end{aligned}
$$

Ahora no tenemos una estructura de cadena y en algunos casos necesitaremos sumar los gradientes a través de diferentes caminos, como es el caso del nodo x_4 que influye en x_5 y x_7. Para asegurar un funcionamiento correcto basta con nombrar los nodos en orden topológico (los padres antes que los hijos) y luego hacer la computación en orden topológico inverso. En general usamos

$$
\frac{\partial o}{\partial x_j} = \sum_{k \in Hijos(j)} \frac{\partial o}{\partial x_k} \frac{\partial x_k}{\partial x_j}.
$$

En nuestro ejemplo para el nodo x_4:

$$
\frac{\partial o}{\partial x_4} = \frac{\partial o}{\partial x_5} \frac{\partial x_5}{\partial x_4} + \frac{\partial o}{\partial x_7} \frac{\partial x_7}{\partial x_4}.
$$

En la práctica el grafo computacional se puede calcular previamente, usando una librería que nos permita definir un grafo estático. Alternativamente podemos calcular el grafo en tiempo real, siguiendo la ejecución de la función en un elemento de entrada. Esta segunda opción hace más fácil trabajar con grafos dinámicos cuya forma puede cambiar dependiendo de los valores calculados por la función. Por ejemplo, Tensorflow 1 usaba los grafos estáticos mientras que su versión más reciente TensorFlow 2 y PyTorch usan los grafos en tiempo real.

4.4. Problemas con el cálculo del gradiente

4.4.1. Desvanecimiento y explosión del gradiente

Siguiendo el hilo de la Sección 3.4, vamos a ver dos problemas que surgen a la hora de entrenar modelos usando gradiente descendente y que pueden impedir la convergencia, pero con la diferencia de que estas limitaciones en la convergencia están ligadas únicamente al algoritmo de BP, es decir, a cómo se calcula el gradiente y no a cómo se usa en la búsqueda de soluciones.

Cuando entrenamos modelos muy profundos (con muchas capas ocultas), los gradientes tienen tendencia bien a volverse muy pequeños (desvanecimiento del gradiente) o bien a volverse muy grandes (explosión del gradiente) ya que la señal de error es pasada a través de una serie de capas

que o lo amplifican o lo mitigan. Esto provoca que o bien se deje de actualizar el peso del cual se desvanece su gradiente o que el gradiente diverja en el otro caso [Hoc+01]. Para ver el problema con detalle, consideramos el gradiente de la función de pérdida con respecto a un nodo en la capa l:

$$\frac{\partial \mathcal{L}}{\partial z_l} = \frac{\partial \mathcal{L}}{\partial z_{l+1}} \frac{\partial z_{l+1}}{\partial z_l} = g_{l+1} J_l$$

donde $J_l = \frac{\partial z_{l+1}}{\partial z_l}$ es la matriz jacobiana, y $g_{l+1} = \frac{\partial \mathcal{L}}{\partial z_{l+1}}$ es el gradiente de la siguiente capa. Si J_l es constante entre capas, es claro que la contribución del gradiente de la capa final g_L a la capa l será $g_L J^{L-l}$. Entonces el comportamiento del sistema dependerá de los valores propios de J.

J es una matriz de valores reales pero generalmente no es simétrica, por lo que sus autovalores y autovectores pueden ser complejos, representando un comportamiento oscilatorio. Sea λ el radio espectral de J, que es el máximo del valor absoluto de sus autovalores. **Si es mayor que 1, el gradiente puede explotar; y si es menor que 1 su gradiente se puede desvanecer.**

El problema de la explosión del gradiente se puede resolver de manera rápida y cómoda a través de acotar el gradiente con su magnitud y una constante $c \in \mathbb{R}^+$ en caso de que se vuelva muy grande

$$g' = min(1, \frac{c}{\|g\|} g)$$

De esta manera la norma de g' nunca puede ser mayor que c, pero el vector apunta siempre en la misma dirección que el gradiente.

También existen otras soluciones que además son aplicables al problema del desvanecimiento de gradiente, que no se soluciona de manera tan sencilla:

- Adaptar las funciones de activación para prevenir que el gradiente se vuelva muy grande o muy pequeño.
- Modificar la arquitectura del modelo para estandarizar las funciones de activación en cada capa, para que la distribución de las activaciones sobre el conjunto de datos permanezca constante durante el entrenamiento.
- Elegir cuidadosamente los valores iniciales de los pesos del modelo.

En la siguiente sección veremos detenidamente el último punto, ya que es la práctica más estandarizada. Existen además familias concretas de modelo que mitigan específicamente estos efectos con su arquitectura como son las ResNets, de las cuales hablaremos en la parte Informática de la memoria.

4.4.2. Inicialización de los pesos

Cómo inicializamos los pesos es una decisión importante a la hora de determinar cómo converge (y si lo hace) un modelo. La convergencia o no, su velocidad y la solución a la que se converge en el entrenamiento de un modelo mediante el algoritmo de gradiente descendente es muy sensible al punto inicial desde el que comenzamos la búsqueda de una solución. Hay que remarcar que esto sucede cuando la función de error no es convexa, pero como es el caso mayoritario, lo asumimos de manera general.

Basándonos en [GB10], donde se observa que inicializar parámetros de una distribución normal con varianza fija puede resultar en el problema de la explosión de gradiente, vamos a ver por qué ocurre esto y, a través de añadir restricciones para evitarlo vamos a llegar a las heurísticas de inicialización de pesos más comunes.

Consideramos el pase hacia delante en una neurona lineal sin capa de activación dada por $o_i = \sum_{j=1}^{n_{in}} w_{ij}x_j$. Suponemos $w_{ij} \sim \mathcal{N}(0, \sigma^2)$, con $\mathbb{E}[x_j] = 0$ y $\mathbb{V}[x_j] = \gamma^2$, donde asumimos x_j independiente de w_{ij}, y n_{in} es el número de conexiones de entrada que recibe la neurona. La media y la varianza de la salida vienen dadas por

$$\mathbb{E}[o_i] = \sum_{j=1}^{n_{in}} \mathbb{E}[w_{ij}x_j] = \sum_{j=1}^{n_{in}} \mathbb{E}[w_{ij}]\mathbb{E}[x_j] = 0$$

$$\mathbb{V}[o_i] = \mathbb{E}[o_i^2] - (\mathbb{E}[o_i])^2 = \sum_{j=1}^{n_{in}} \mathbb{E}[w_{ij}^2 x_j^2] - 0 = \sum_{j=1}^{n_{in}} \mathbb{E}[w_{ij}^2]\mathbb{E}[x_j^2] = n_{in}\sigma^2\gamma^2.$$

Para evitar que la varianza diverja, necesitamos que $n_{in}\sigma^2$ se mantenga constante. Si consideramos el pase hacia atrás y realizamos un razonamiento análogo vemos que la varianza del gradiente puede explotar a menos que $n_{out}\sigma^2$ sea constante, donde n_{out} son las conexiones de salida de la neurona. Para cumplir con esos dos requisitos, imponemos $\frac{1}{2}(n_{in} + n_{out})\sigma^2 = 1$, o equivalentemente

$$\sigma^2 = \frac{2}{n_{in} + n_{out}}.$$

Esta se conoce como inicialización de Xavier o inicialización de Glorot [GB10]. Si usamos $\sigma^2 = \frac{1}{n_{in}}$ tenemos un caso especial conocida como la inicialización de LeCun, propuesta por Yann LeCun en 1990. Es equivalente a la inicialización de Glorot cuando $n_{in} = n_{out}$. Si usamos $\sigma^2 = \frac{2}{n_{in}}$, tenemos la llamada inicialización de He, propuesta por Kaiming He en [He+15].

Cabe resaltar que no ha sido necesario usar una distribución Gaussiana. De hecho, las derivaciones de arriba funcionan en términos de la media y la varianza, y no hemos hecho suposiciones sobre si era Gaussiana.

Aunque hemos supuesto que se trataba de una neurona lineal sin función de activación que añada una componente no lineal, se conoce de manera empírica que estas técnicas son extensibles a unidades no lineales. La inicialización que elijamos dependerá mayoritariamente de la función de activación que usemos. Se conoce que para funciones de activación *ReLU* funciona mejor la inicialización de He, para las funciones *SELU* se recomienda la inicialización de LeCun y para las funciones lineales, logística, tangente hiperbólica y *Softmax* se recomienda el uso de la inicialización de Glorot [Mur22].

5. Conclusiones y trabajos futuros

Al comienzo de esta parte matemática, se establecieron dos objetivos principales:

- Analizar la convergencia del gradiente descendente.
- Explorar el uso de BP para este algoritmo de aprendizaje.

En primer lugar hemos introducido los conceptos y términos necesarios con los que íbamos a trabajar, y presentamos la idea original de Cauchy de la que surge el algoritmo de aprendizaje que conocemos hoy día. Luego, diferenciamos tres versiones distintas del algoritmo según la cantidad de datos que se usen para calcular el gradiente y vimos los componentes básicos de este método. También introdujimos el concepto de subgradiente, que nos permite analizar de manera rigurosa el algoritmo de gradiente descendente sin necesidad de que todas las funciones que intervienen en el modelo sean diferenciables. **Hemos visto el enunciado y demostración de un teorema que nos asegura la convergencia al mínimo global del algoritmo en su versión BGD**, aunque con condiciones muy estrictas como que la función de coste sea convexa o de clase C^2.

Presentamos el concepto de martingala, un tipo de proceso estocástico con el que podemos modelar las versiones estocásticas del algoritmo de gradiente descendente. A través del **teorema de Siegmund-Robbins**, que proporciona convergencia para las casi supermartingalas, **conseguimos demostrar un teorema que nos asegura la convergencia de SGD y MBSGD hacia un minimizador global con probabilidad 1**, consiguiendo un teorema mucho más práctico que el anterior. Aunque es necesario para ello la hipótesis, demasiado estricta, de que la función sea globalmente fuertemente convexa, **se espera que haya convergencia hacia un minimizador local en caso de que la función de coste sea fuertemente convexa a nivel local**.

Para analizar y comprender el algoritmo de BP, hemos visto qué es la diferenciación automática y qué estrategia utiliza para realizar cálculos de manera eficiente. Usando como ejemplo un MLP, pudimos ver cómo funciona esta técnica y distinguimos entre diferenciación hacia delante y hacia detrás. Una vez presentados estos conceptos, **exploramos propiamente el algoritmo de BP, en qué se basa y por qué resulta eficiente dicho algoritmo**. Concluimos finalmente poniendo varios ejemplos prácticos de cálculos realizados con esta técnica y resaltando problemas que trae consigo.

Esto ha permitido conocer los procesos estocásticos de tipo martingala, que no solo son una herramienta poderosa para el análisis del gradiente descendente, sino que resultan una estrategia ampliamente usada para derivar

resultados sobre convergencia o probar límites probabilísticos. De esta manera se buscan modelar ciertos procesos con estos objetos para aprovechar sus cualidades y propiedades.

Los conceptos de subgradiente y subdiferenciabilidad, aunque menos ampliamente usados que el anterior, constituyen un recurso indispensable en el campo del aprendizaje profundo para el análisis de técnicas relacionadas con el gradiente descendente a nivel teórico. Gracias a este trabajo he adquirido una visión más teórica y formal del aprendizaje profundo y de cómo se lleva a cabo su entrenamiento. **En un campo donde en muchas ocasiones prima el ensayo y error, adquirir una perspectiva teórica formal resulta indispensable a la hora de investigar y poder ofrecer mejoras**.

Como conclusión, creo que este trabajo me ha servido para poner en común el aprendizaje de ambos grados y gracias a él he adquirido además destreza y soltura para buscar, consultar y leer publicaciones matemáticas en este ámbito, lo cual es una habilidad muy importante.

Después de comentar los resultados obtenidos, en el futuro podrían resultar interesantes los siguientes trabajos:

1. Para el Teorema 3.13, sobre la convergencia de SGD, demostrar de manera rigurosa las suposiciones que se plantean ante la relajación de las condiciones de convexidad: cuando la función de coste no es globalmente fuertemente convexa, sino que es fuertemente convexa a nivel local, entonces el algoritmo converge a un minimizador local en lugar de a uno global.

2. Los procesos estocásticos de tipo martingala han demostrado ser herramientas poderosas para el análisis teórico de la convergencia de SGD. Se podrían reformular optimizadores modernos basados en GD, como Adam o RMSProp, en términos de procesos de martingalas para analizar de manera particular y más rigurosa sus propiedades de convergencia y estabilidad.

Parte II

Parte informática: Estudio empírico comparativo entre gradiente descendiente y metaheurísticas para el entrenamiento de redes neuronales profundas

6. Introducción

Las **redes neuronales** son modelos computacionales inspirados originalmente en los mecanismos de aprendizaje y procesamiento de información del cerebro humano [MP43]. Estos modelos, organizados en capas, están formados por un conjunto de unidades de procesamiento simples, comúnmente denominadas neuronas artificiales, que están altamente interconectadas y, mediante un proceso de aprendizaje, son capaces de resolver tareas específicas [Bis06; Mur22; AML12]. Las redes neuronales constituyen uno de los pilares fundamentales del éxito actual del aprendizaje profundo[15] y, por extensión, es una de las bases del desarrollo de la inteligencia artificial (IA) [BB23; Pri23; Rus16].

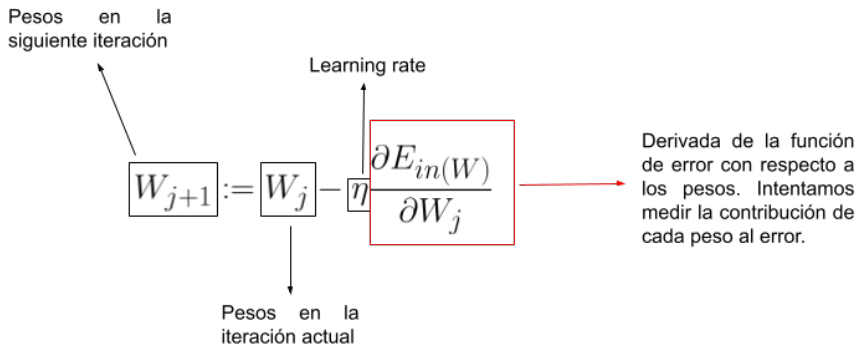

Figura 8: En la figura observamos la fórmula de actualización de los pesos de un modelo a través del algoritmo de gradiente descendente. Los elementos se indican en un recuadro señalado con su correspondiente explicación. El elemento señalado en rojo es el correspondiente al algoritmo de *backpropagation*, es decir, usamos este algoritmo para computar esta derivada parcial. Con esta figura se pretende resaltar la diferencia entre el algoritmo de gradiente descendente, utilizado para la optimización de los parámetros en el entrenamiento de modelos, y el algoritmo de *backpropagation*, cuyo fin es el cálculo eficiente del gradiente de una función.

El **entrenamiento** de una red neuronal es el proceso por el que optimizamos sus parámetros internos para minimizar una **función de pérdida** que cuantifica el error entre las predicciones realizadas por la red y las etiquetas reales del conjunto de datos. Este proceso es conceptualizado como un **problema de optimización** [LBH15], en el que utilizando un **algoritmo de aprendizaje** actualizamos iterativamente los parámetros de la red en dirección al mínimo de la función de pérdida. El entrenamiento no

[15] El aprendizaje profundo permite que estos modelos computacionales adquieran representaciones jerárquicas con múltiples niveles de abstracción a partir de los datos de entrada [LBH15; Sch15].

sólo busca aprender los patrones en los datos para minimizar el error, sino también garantizar la **capacidad de generalización** del modelo a nuevos datos [BB23; GBC16].

El algoritmo de aprendizaje más habitual para redes neuronales es el **gradiente descendente** [BV04; Cur44] (GD), que calcula iterativamente el gradiente de la función de pérdida con respecto a los parámetros de la red, y los actualiza en dirección opuesta al gradiente para disminuir el error. Existen diferentes versiones del algoritmo según el tamaño del subconjunto de datos usado para calcular el gradiente en cada iteración. Para este cálculo se utiliza el algoritmo de ***backpropagation*** [RHW86; LeC+12], que implica la propagación del error hacia atrás a través de las capas de la red y que implica la aplicación de la regla de la cadena.

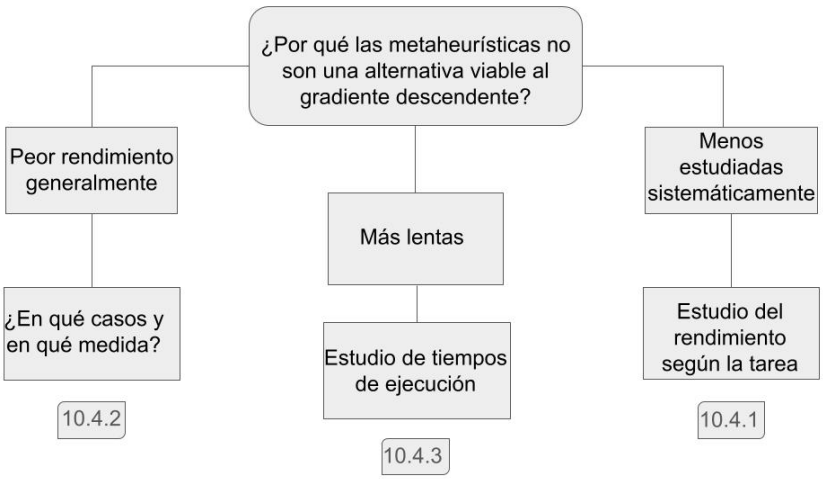

Figura 9: Esquema general de este TFG con los objetivos parciales a abordar en el mismo. Se parte de la premisa que se plantea en la parte superior a modo de pregunta: "¿Por qué las metaheurísticas no son una alternativa viable al gradiente descendente [en el momento actual]?". A continuación, planteamos tres posibles respuestas, que son campos de investigación abiertos, y se observan en los tres recuadros centrales. Finalmente, establecemos los enfoques/experimentos desde los que intentaremos abordar la investigación. En la parte baja se indica la sección del TFG en la que se aborda cada punto. No se muestra el objetivo parcial correspondiente a la propuesta algorítmica propia, cuya idea se sintetiza en la Tabla 1, se explica en la Sección 9 y se evalúa en la Sección 10.4.4.

A pesar de su eficacia y popularidad, el GD presenta varias limitaciones. Una de las principales es su susceptibilidad a **quedar atrapado en mínimos locales** o puntos de silla [Dau+14], especialmente en problema de optimización no convexos, lo que impide alcanzar la solución óptima. Otra es su gran **sensibilidad**, tanto a los propios hiperparámetros del algoritmo

como a los valores iniciales de los parámetros de la red [GB10], que si no se eligen de manera adecuada pueden afectar negativamente al rendimiento e incluso impedir la convergencia. Existen modificaciones del algoritmo que intentan paliar estos problemas al incorporar factores adicionales en la actualización de los pesos.

Por otro lado, las **metaheurísticas** (MH) [GP10] son técnicas de optimización basadas normalmente en componentes bio-inspirados, que son flexibles y adaptables a gran variedad de problemas. Ofrecen una solución aproximada en un tiempo razonable en muchos problemas cuya solución óptima es computacionalmente inalcanzable, como en problemas NP-Difícil. Son técnicas iterativas que no ofrecen una garantía teórica de hallar una buena solución [Tal09], pero a través de restricciones en el algoritmo se espera que lo hagan. Estas estrategias **no necesitan el uso de gradiente** por lo que exploran el espacio de soluciones de manera más global y no tienen algunas de las restricciones anteriormente mencionadas.

En el presente TFG exploraremos las limitaciones de las técnicas MH como alternativa al GD, realizando un estudio comparativo entre ambos enfoques como se observa de manera esquemática en la Figura 9. Tomaremos como referencia el artículo [Mar+21], donde se analizan las razones por las que las MH aún no igualan el rendimiento del GD, y plantearemos pruebas experimentales diseñadas para identificar las principales limitaciones de estas técnicas. A su vez, estas pruebas permitirán evaluar las virtudes y defectos de las estrategias MH, con el objetivo de ofrecer propuestas para futuras mejoras en su aplicación. Finalmente, **propondremos dos técnicas nuevas** a través de combinar MH con el descenso del gradiente, cuyas cualidades se resumen en la Tabla 1.

Aspecto	Técnicas Clásicas	Metaheurísticas	Propuestas Propias
Fundamento teórico	Basadas en el gradiente.	Se basan en heurísticas de búsqueda (aleatorias, evolutivas, inspiradas en la naturaleza, etc.).	Combinar el uso del gradiente con mecanismos evolutivos.
Ventajas	- Alta eficiencia cuando se dispone de funciones diferenciables. - Buena capacidad de explotación en zonas prometedoras. - Mayor conocimiento teórico: comportamiento más predecible, criterios de convergencia más claros y más estudiados.	- Permiten explorar grandes espacios de búsqueda sin requerir información del gradiente. - Flexibles ante problemas complejos o con múltiples óptimos locales.	- Toman la fuerza de explotación del gradiente y la capacidad exploradora de las metaheurísticas. - Flexibles y adaptables.
Limitaciones	- Pueden estancarse en óptimos locales si la función es compleja. - Requieren derivadas (no siempre disponibles o fiables).	- Suelen requerir más evaluaciones de la función objetivo. - No aprovechan información de derivadas incluso si está disponible, lo que puede resultar menos eficiente.	Aumenta la complejidad de implementación.
Capacidad	Principalmente explotadora: refinan soluciones en regiones vecinas.	Principalmente exploradora: buscan soluciones en todo el espacio, reduciendo la probabilidad de caer en óptimos locales.	Combina explotación (guiada por el gradiente) con exploración (a través de SHADE, basado en *Differential Evolution*).
Algoritmos	Adam [KB14], NAG [Nes83], RMSProp [Hin12], AdamW [LH19]	SHADE [TF13], SHADE-ILS[16] [MLH18]	SHADE-GD, SHADE-ILS-GD.

Tabla 1: Tabla comparativa entre las técnicas clásicas basadas en gradiente, las metaheurísticas y las dos propuestas algorítmicas propias (SHADE-GD y SHADE-ILS-GD) para el entrenamiento de modelos. Las propuestas híbridas buscan combinar la eficiencia y capacidad explotadora de los optimizadores basados en gradiente con la capacidad exploradora de las metaheurísticas. Este enfoque permite superar algunas limitaciones del gradiente, como el estancamiento en óptimos locales. El análisis detallado de estas propuestas y los resultados empíricos pueden encontrarse en la Sección 9 y la Sección 10.4.4, respectivamente.

[16] Dentro de los algoritmos MH, SHADE-ILS pertenece al grupo de algoritmos meméticos, que sí incorporan búsqueda local, y en particular este algoritmo utiliza información de gradiente.

6.1. Motivación

El ajuste de parámetros es crucial en el aprendizaje profundo ya que determina directamente el rendimiento y la funcionalidad de nuestro modelo. Éste depende de una optimización que permita minimizar en el menor tiempo posible la función de pérdida y capturar los patrones subyacentes a los datos de entrada. Una correcta optimización de los parámetros también mejora la eficiencia computacional e incrementa la fiabilidad de las aplicaciones de aprendizaje profundo.

La optimización de los parámetros de las redes neuronales profundas es inherentemente un problema desafiante. Las superficies de pérdida de estos modelos, sobre todo a medida que aumentan en profundidad (número de capas) y complejidad (número de parámetros), son altamente no lineales y están plagadas de mínimos locales, regiones planas y puntos de silla [Dau+14]. Además, la alta dimensionalidad del espacio de parámetros, que a menudo llega a millones o miles de millones de dimensiones, incrementa esta dificultad, ya que resulta computacionalmente prohibitivo explorar con técnicas exhaustivas el espacio de búsqueda.

Mientras que las técnicas de optimización basadas en gradientes son eficientes y escalan bien, depender de información local (los gradientes) las hacen vulnerables a ciertas limitaciones como quedar atrapadas en mínimos locales, especialmente en tareas complejas. Por otro lado, las MH son un enfoque completamente distinto, siendo estrategias de búsqueda global que no requieren de información del gradiente, haciéndolas menos sensibles a mínimos locales y potencialmente más robustas navegando superficies de pérdida complejas.

Aun así estas técnicas son estudiadas de manera menos sistemática que los métodos basados en gradiente, y aunque se conoce que aún no son capaces de igualar el rendimiento de las técnicas clásicas [Mar+21], son realmente prometedoras y un campo muy activo de investigación, en especial su escalabilidad, coste computacional, y su habilidad de generalización en redes de gran escala [Mar+21]. Además, las diferencias en diseño, requerimientos computacionales, dependencia de hiperparámetros y criterios de comparación o evaluación, hacen difícil comparar objetivamente y de manera no sesgada las causas y contextos específicos de la diferencia de rendimiento entre el GD y las técnicas MH.

Este TFG intentará abordar algunas de estas temáticas abiertas, a través de comparar empíricamente el descenso de gradiente y las técnicas MH en el entrenamiento de redes neuronales profundas. Al hacerlo, se pretende arrojar luz sobre el balance de estos métodos, identificando escenarios donde las MH podrían ofrecer un mejor rendimiento, y proporcionando una comprensión más profunda de las estrategias de optimización en aprendizaje profundo.

6.2. Objetivos

El objetivo principal de este TFG es evaluar y analizar la eficacia de las técnicas MH para el entrenamiento de redes neuronales profundas. Lo haremos comparándolas con el algoritmo de GD, para tener una mejor comprensión de cuáles son las diferencias principales en el rendimiento de estas dos estrategias y cuáles son las causas de las mismas. De esta manera, se podrían ofrecer modificaciones que mejoren estas técnicas tan prometedoras. Para ello vamos a dividir este objetivo en varios secundarios:

1. Estudio de la literatura existente, tanto para optimizadores basados en GD clásicos como para el uso de MH en el entrenamiento de modelos de aprendizaje automático.

2. Análisis de los factores que provocan que el rendimiento de las técnicas MH empeore conforme aumenta la complejidad de la tarea, el número de parámetros del modelo y el tamaño del conjunto de datos.

3. Evaluar si existe una diferencia de rendimiento significativa en los modelos entrenados con técnicas MH en función del tipo de tarea a resolver.

4. Medición y análisis de los tiempos de ejecución de las MH, comparando y entendiendo las diferencias con el GD.

5. Propuesta original de dos algoritmos de optimización que hibriden las técnicas MH con el GD.

6.3. Planificación

La planificación del TFG ha sido pensada basándonos en los conocimientos previos del grado y en el libro [Som22]. Usamos un modelo de cascada retroalimentada, aunque se han realizado varios cambios en la planificación a lo largo del desarrollo. La naturaleza de este proyecto es práctica en el marco de la investigación, por lo que está centrado en la realización y análisis de los experimentos propuestos. Además requiere de una investigación y aprendizaje previos del tema a abordar, seguida de una etapa de diseño de los propios experimentos y el código relacionado.

Una vez implementados y ejecutados los experimentos, analizamos sus resultados y, tras realizar las pertinentes revisiones, podemos decidir modificar ciertos aspectos debido a que se hubieran detectado fallos o identificado potenciales mejoras. Podemos ver una representación gráfica del modelo seguido en la Figura 10.

Las etapas en las que se dividirá el desarrollo son las siguientes:

1. Investigación: en la primera etapa se buscará bibliografía para profundizar en los conocimientos sobre el entrenamiento de modelos de

aprendizaje profundo con GD, se analizará literatura reciente para ver las posibles alternativas y se indagará sobre el estado del arte del entrenamiento de modelos para observar posibles vías de investigación abiertas.

2. Diseño: en base a la información anterior, se concretará el objetivo principal del TFG y en base a él se diseñarán uno o varios experimentos que permitan cumplirlo. En base a esos experimentos, se comenzará a diseñar el código a alto nivel.

3. Implementación: se implementa el código necesario diseñado en la etapa anterior.

4. Ejecución: en esta etapa se ejecutan los experimentos.

5. Análisis: una vez obtenidos los resultados, analizamos los datos para ver si confirmamos las hipótesis de la experimentación, las refutamos, o no obtenemos conclusiones. Durante esta etapa pueden llevarse a cabo, por ejemplo, test estadísticos para confirmar ciertas hipótesis.

6. Revisión: Una vez analizados los datos, vemos la experimentación en su conjunto, analizamos posibles fallos metódicos y en caso de ser necesario modificamos cosas puntuales de las etapas de diseño, implementación y ejecución.

El proyecto estaba pensado originalmente para ser empezado en enero y entregarlo en la convocatoria de julio, siguiendo con la planificación que se muestra en la Figura 11a. Debido a la carga de trabajo del curso y a que finalmente se comenzó más tarde su trabajo se decidió modificar la propuesta inicial para adaptarla a la realidad, resultando en la planificación que se observa en la Figura 11b. En dicha modificación se le asigna más tiempo a la implementación ya que durante los meses de verano disminuyó la cantidad de horas semanal invertidas. A partir de marzo se ha realizado alguna modificación puntual pero no se ha establecido una cantidad de trabajo semanal, por lo que no se computa.

Suponiendo que el sueldo medio de un investigador en España es de unos 30€/h[17], y con una media de 15 horas semanales para el periodo entre febrero y mayo, 5 entre junio y agosto y 30 entre septiembre y marzo (sin contar con diciembre y enero), el coste del personal del proyecto ha resultado ser de 23,400€. Si atendemos a la planificación original, donde se tenía pensado dedicar 20 horas semanales hasta mayo y 15 en junio y julio, el coste del personal hubiera sido de 15,600€, con lo que tendríamos un sobrecoste de 7,800€.

Para la ejecución de los experimentos se requiere una plataforma que proporcione acceso a hardware en la nube bajo demanda. En este caso se

[17] https://www.whitecarrot.io/salary-progression/ai-researcher

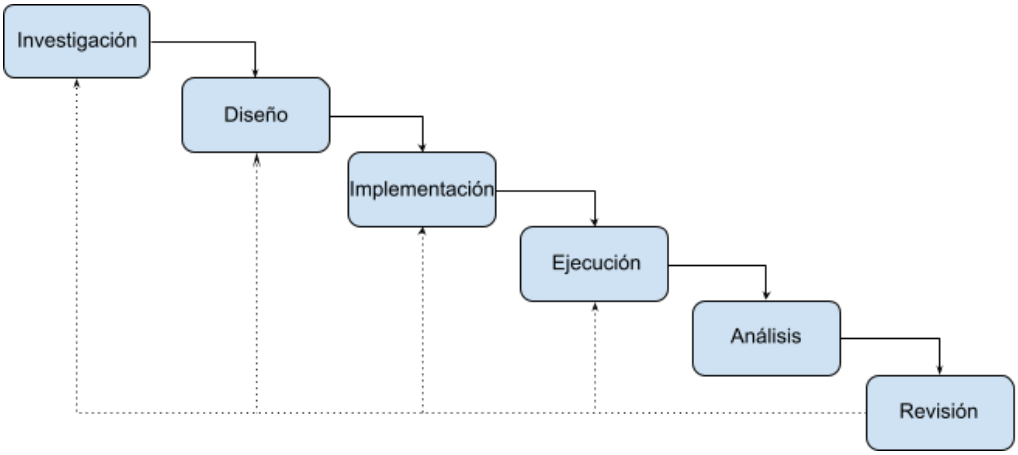

Figura 10: Modelo en cascada retroalimentada utilizado en la planificación del proyecto. El proceso inicia con una fase de investigación previa, seguida por el diseño del código y de los experimentos. Luego, se implementa el código necesario para ejecutar los experimentos y, posteriormente, se analizan los resultados obtenidos. En la etapa final de revisión, es posible retroceder a fases anteriores (investigación, diseño, implementación y ejecución) para realizar ajustes y mejoras según sea necesario..

ha seleccionado Paperspace[18] plataforma utilizada en este TFG, aunque en una versión inferior. La elección de una versión superior responde a la necesidad de mayor versatilidad y al supuesto de contar con un presupuesto más amplio. Se ha optado por un plan de suscripción de 34 dólares mensuales (aproximadamente 31.50€). Dado el cronograma del proyecto y considerando que las etapas de investigación y diseño no requieren el uso de la plataforma se ha estimado un periodo de suscripción de 8 meses con un coste total de 252€. En comparación con la planificación inicial que contemplaba únicamente 3 meses de suscripción (94.50€), esto representa un incremento de 157.50€. En consecuencia el presupuesto total estimado para el proyecto asciende a 23,652€.

[18] https://www.paperspace.com/

Figura 11: Diagramas de Gantt de la planificación inicial (arriba) y final (abajo) del proyecto. El primero muestra una planificación inicial más compacta, mientras que el segundo refleja la planificación final ajustada según el progreso y la complejidad de las tareas. Las etapas incluyen: investigación previa, diseño, implementación, ejecución, análisis de resultados y revisión. Los cambios entre ambas versiones evidencian una mayor desagregación de las fases de implementación, ejecución y análisis, lo que permitió una gestión más detallada del tiempo y los recursos. A partir de marzo no se computa ya que, aunque se ha realizado alguna modificación puntual, no se ha establecido una cantidad de trabajo semanal.

7. Fundamentos teóricos

En esta sección se detallan los conceptos, familias de modelos y algoritmos necesarios para la elaboración del trabajo posterior. Se usarán los conocimientos aprendidos en las asignaturas de Aprendizaje Automático, Visión por Computador y Metaheurísticas además de las referencias indicadas en cada caso.

En la Sección 7.4 aparecen los optimizadores basados en GD que se utilizarán en la experimentación: Adam, AdamW, NAG y RMSProp, mientras que en la Sección 7.5 se hace lo propio con los algoritmos MH: SHADE y SHADE-ILS. Para finalizar, en la Sección 7.6.2 se describen los tests estadísticos que más tarde se emplearán en la experimentación para un análisis riguroso de los rendimientos de los modelos.

7.1. Redes neuronales y aprendizaje profundo

7.1.1. Red neuronal

Una red neuronal es un modelo computacional inspirado en la manera en la que las neuronas se conectan en el cerebro humano procesando la información. Consiste en capas interconectadas con nodos llamados neuronas, donde cada conexión tiene un peso asociado. Cada neurona normalmente aplica una función no linear, llamada función de activación, a la suma ponderada de las entradas de la capa anterior, permitiendo al modelo aprender relaciones complejas [Fei24]. Este tipo de redes se denominan totalmente conectadas. Sus componentes básicos son:

- Capa de entrada: recibe la información.
- Capas ocultas: son las capas intermedias, que realizan los cálculos.
- Capa de salida: produce la salida del modelo.

El ejemplo más sencillo es el Perceptrón [Ros58], una red neuronal de una sola capa oculta y una sola neurona como podemos ver en la Figura 12. Es un clasificador lineal, es decir, sólo puede resolver tareas cuyos datos sean linealmente separables. En su versión original su función de activación f es la función signo. Para problemas más complejos que no sean lineales necesitamos usar redes neuronales con varias capas ocultas.

7.1.2. Aprendizaje profundo y redes neuronales profundas

Las redes neuronales profundas tienen varias capas ocultas, generalmente más de dos. La profundidad de una red viene determinada por el número de capas ocultas que tiene, y ésta permite aprender representaciones jerárquicas de los datos, lo que habilita a las redes neuronales profundas a capturar

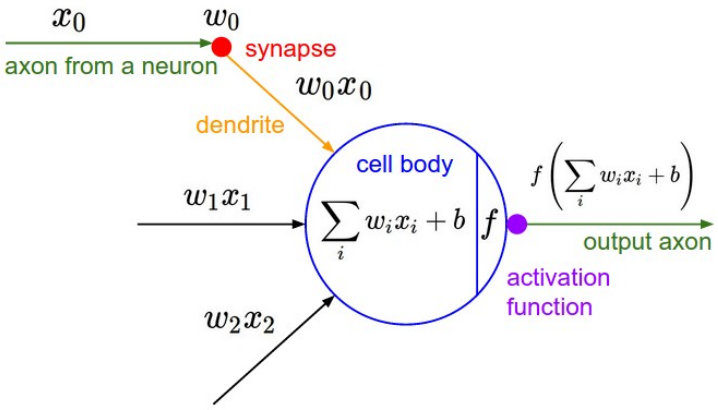

Figura 12: Representación esquemática de una neurona artificial inspirada en el modelo biológico. La imagen ilustra la analogía entre una neurona biológica y una neurona artificial en una red neuronal. Las entradas x_0, x_1, x_2 representan señales provenientes de otras neuronas, con pesos sinápticos w_0, w_1, w_2 que modulan su importancia. En la biología, estas señales viajan a través del axón de una neurona y se transmiten mediante sinapsis (en rojo) a la dendrita (en naranja) de otra neurona. En la neurona artificial, las entradas ponderadas se combinan el cuerpo celular (en azul) mediante una suma ponderada $\sum_i w_i x_i + b$, donde b es un sesgo. Esta combinación es procesada por una función de activación f (en morado), cuya salida viaja a través del axón de salida (en verde), propagando la información a la siguiente capa de la red. Obtenida de [Fei24].

patrones más complejos en los datos en comparación con redes con menos profundidad [Fei24].

El aprendizaje profundo es una rama del aprendizaje automático que se centra en las redes neuronales profundas. Generalmente se usan modelos muy profundos con un alto número de parámetros y grandes cantidades de datos para resolver tareas complejas. Esto supone que se requiere de mucho poder computacional y de algoritmos avanzados para optimizar sus parámetros. Este tipo de modelos obtiene un gran rendimiento en tareas como el procesamiento del lenguaje natural, reconocimiento de voz o visión por computador. Los perceptrones multicapa son el ejemplo más clásico.

7.1.3. Perceptrones Multicapa

La familia de modelos que usaremos en la experimentación para las tareas de regresión será la de los Perceptrones Multicapa o *Multilayer Perceptrons* (MLP), que son una versión más compleja del Perceptrón, que cuenta con varias capas ocultas y varias neuronas en cada una como el ejemplo de la Figura 13. Son capaces de procesar datos no linealmente separables ya

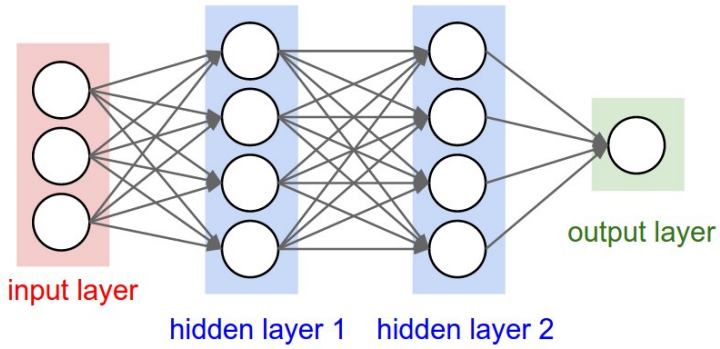

Figura 13: Estructura de una red neuronal profunda con dos capas ocultas. La imagen representa una red neuronal artificial multicapa compuesta por una capa de entrada (*input layer*, en rojo), dos capas ocultas (*hidden layers*, en azul) y una capa de salida (*output layer*, en verde). Cada neurona en una capa está completamente conectada con las neuronas de la siguiente capa mediante pesos sinápticos, representados por líneas. La capa de entrada recibe los datos del problema, las capas ocultas permiten la extracción de características y la modelización de relaciones complejas, y la capa de salida genera la predicción final. Esta arquitectura es típica en aprendizaje profundo y se entrena mediante algoritmos de optimización como el descenso de gradiente. Obtenida de [Fei24].

que pueden aprender información más compleja. Sus capas son totalmente conectadas y la información fluye sólo hacia delante a la hora de hacer una predicción con el modelo [GBC16].

En su definición original, usan la función signo como función de activación en todas las neuronas y sólo se usan para tareas de clasificación. Sin embargo actualmente son sinónimo de redes profundas totalmente conectadas, siendo usadas con cualquier tipo de función de activación y para tareas de clasificación o regresión.

7.2. Redes convolucionales

Las ConvNets o redes convolucionales son una familia de modelos de aprendizaje profundo usadas en la visión por computador. Obtienen un rendimiento al nivel del estado del arte en tareas como el reconocimiento de imágenes o la detección de objetos. Se caracterizan por tener una o varias capas (al menos una) basadas en convoluciones para luego tener una o varias capas totalmente conectadas. Las primeras sirven como extractores de características que capturan propiedades espaciales de las imágenes, mientras que las segundas sirven para clasificación [GBC16].

7.2.1. Operación de convolución

La convolución es una operación matemática que expresa la relación entre la entrada, la salida y la respuesta del sistema a los impulsos. En el contexto del procesamiento de señales, la convolución combina dos señales para producir una tercera [GBC16]. Se define matemáticamente para funciones continuas como

$$(f * g)(t) = \int_{-\infty}^{\infty} f(x)g(t-x)dx.$$

Para funciones discretas se define como

$$(f * g)(t) = \sum_{x=-\infty}^{\infty} f(x)g(t-x).$$

Nos referimos a f como la entrada y a g como el núcleo o filtro. En el aprendizaje automático la entrada suele ser un tensor de datos y el filtro un tensor de parámetros que adaptamos con el algoritmo de aprendizaje. Ambos son de dimensión finita y asumimos que su valor es 0 en todos los puntos donde no almacenamos su valor. Por tanto en la práctica podemos implementar la sumatoria infinita como una suma finita de los elementos de un vector. Si usamos una imagen bidimensional I como entrada, seguramente usaremos un filtro bidimensional K:

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n).$$

Las convoluciones cumplen las siguientes propiedades:

- **Conmutatividad**: $f * g = g * f$.
- **Asociatividad**: $f * (g * h) = (f * g) * h$.
- **Distributividad**: $f * (g + h) = (f * g) + (f * h)$.

La propiedad conmutativa es útil a nivel matemático pero no es demasiado práctica en la implementación de una red neuronal. Por ello muchas librerías de aprendizaje automático optan por implementar la función llamada relación cruzada en lugar de la convolución, volteando el núcleo como podemos ver en la Figura 14.

$$C(i,j) = (I \cdot K)(i,j) = \sum_m \sum_n I(i+m, j+n)K(m,n).$$

Al igual que hacen estas librerías, llamamos a estas dos operaciones indistintamente convolución, ya que en el contexto del aprendizaje de modelos no habrá diferencia, porque el algoritmo de aprendizaje obtendrá los mismos valores para el núcleo y sólo variará su posición. Podemos considerar

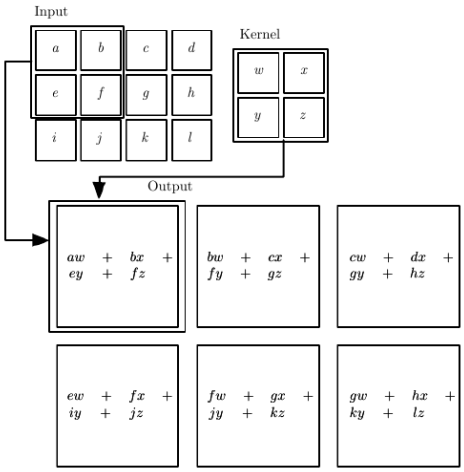

Figura 14: Operación de correlación cruzada en una red neuronal convolucional. La imagen ilustra el proceso de correlación cruzada (*cross-correlation*), una operación fundamental en redes neuronales convolucionales (ConvNets), y que es equivalente a la operación de convolución pero sin voltear el filtro. Se muestra una matriz de entrada (*Input*) de 3×4 con elementos etiquetados (a a l) y un kernel o filtro (*Kernel*) de 2×2 con pesos (w, x, y, z). La operación consiste en deslizar el kernel sobre la matriz de entrada, calculando productos ponderados entre los valores superpuestos y sumando los resultados para formar la matriz de salida (*Output*). Cada celda en la matriz de salida representa la suma de productos en una región local de la entrada. Este proceso permite la extracción de características espaciales en imágenes, siendo una base fundamental en la detección de patrones en ConvNets. Obtenida de [GBC16].

la convolución como una multiplicación matricial donde la matriz tiene restricciones en muchas posiciones las cuales deben tener el mismo valor.

7.2.2. Capa Convolucional

Las capas convolucionales son las más importantes en la arquitectura de una ConvNet. Las claves de su gran rendimiento en el campo de la visión por computador son: la conectividad local, la disposición espacial y compartir parámetros [GBC16].

La conectividad local hace referencia a las conexiones de las neuronas. En entradas de alta dimensionalidad como imágenes no es práctico conectar una neurona con todas las neuronas del volumen anterior, por tanto cada neurona se conecta solo a una región local del volumen de entrada. Esto viene determinado por un hiperparámetro llamado campo receptivo, que es el tamaño del filtro que aplicamos. Mientras que las conexiones son locales

en el espacio 2D (ancho y altura), siempre abarcan toda la profundidad del volumen de entrada.

Con la disposición espacial nos referimos al tamaño del volumen de salida y cómo están organizadas estas neuronas. Hay tres hiperparámetros con los que controlamos esto:

1. Profundidad del volumen de salida: corresponde a la cantidad de filtros que queremos usar.
2. *Stride*: Indica el número de píxeles (hablando en términos de imágenes) que usamos para desplazar el filtro al realizar la convolución.
3. *Padding*: A veces, para mantener la dimensión de la salida es conveniente rellenar el borde de la entrada con ceros.

Las dimensiones del volumen de salida podemos calcularlas como una función dependiente del tamaño del volumen de entrada W, el tamaño del filtro F, el *stride* S y el *padding* P que queramos aplicar. La fórmula es la siguiente:

$$\frac{W - F - 2P}{S + 1}. \tag{18}$$

Esta nos dará las dimensiones en ancho y altura del volumen de salida como podemos ver en la Figura 15, y su profundidad vendrá totalmente determinada por el número de filtros que queramos usar.

Compartir parámetros en una ConvNet nos permite reducir el número de éstos, reduciendo el coste del entrenamiento. Se basa en la suposición de que si una característica es útil en una posición espacial (x, y) tambien lo será en otra cercana (x', y'). En un volumen $W \times H \times D$, en lugar de que cada neurona tenga su conjuntos de pesos, tenemos D conjuntos de pesos, reduciendo drásticamente su número.

7.2.3. Capa *Pooling*

Su función es reducir progresivamente el tamaño de la representación para reducir el número de parámetros y la carga computacional en la red, además de controlar el sobreajuste. Opera independientemente a lo largo de la profundidad del volumen, usando la operación máximo. La opción más común es usar filtros 2×2 con un *stride* $S = 2$ como se observa en la Figura 16. La dimensión de la profundidad permanece intacta. Existen otros tipos de *pooling*, por ejemplo realizar la media entre los elementos, pero esta opción fue dejando paso a la de seleccionar el máximo ya que obtiene mejores resultados en la práctica [Fei24].

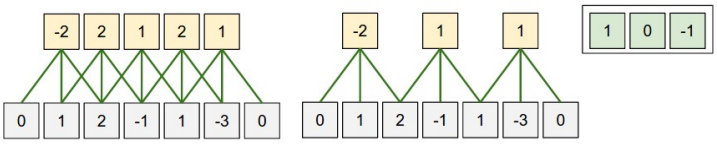

Figura 15: La imagen ilustra el proceso de convolución en una red neuronal convolucional. Los rectángulos grises representan las entradas, mientras que los vectores amarillos son las salidas resultantes de convolucionar las entradas con el filtro (ubicado en la esquina superior derecha). Los tamaños de las salidas pueden calcularse según la fórmula 18. El filtro de tamaño 1×3 tiene valores [1, 0, -1]. El concepto de *padding*, visible como bordes adicionales alrededor de las entradas, se utiliza para controlar el tamaño de las salidas. En ninguno de los dos ejemplos se usa ($P = 1$). El *stride*, o desplazamiento del filtro, determina cómo el filtro se mueve a través de las entradas, afectando el solapamiento y el tamaño resultante de las salidas. En el ejemplo de la izquierda se usa un *stride* de $S = 1$, mientras que en el de la derecha se usa $S = 2$. Estos conceptos son esenciales en el aprendizaje profundo, ya que permiten ajustar la resolución y características extraídas por las ConvNets. Obtenida de [Fei24].

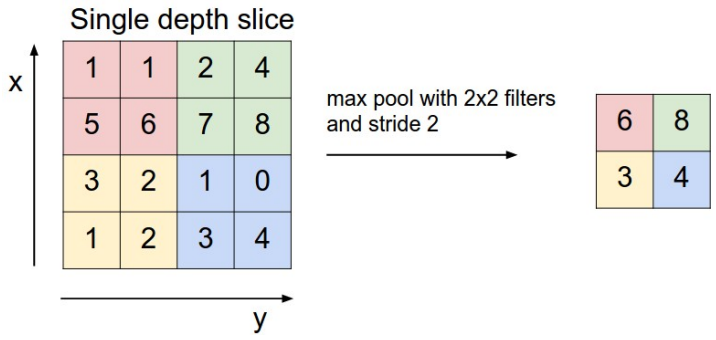

Figura 16: La imagen muestra un ejemplo de una operación *max pooling* en una ConvNet. A la izquierda, se presenta una matriz de 4x4 , con valores que van del 0 al 8. La operación de max pooling se realiza con filtros de 2×2 y un *stride* de 2, lo que significa que el filtro se mueve dos posiciones a la vez tanto en la dirección x como en la dirección y. El resultado de esta operación se muestra a la derecha como una matriz de 2×2, donde cada valor es el máximo de los valores en la correspondiente submatriz de 2×2 de la original. Los valores resultantes son 6, 8, 3 y 4. Esta operación es relevante porque reduce la dimensionalidad de la entrada, manteniendo las características más importantes, lo que ayuda a disminuir el costo computacional y a evitar el sobreajuste en modelos de aprendizaje profundo. Obtenida de [Fei24].

7.2.4. Capa *Batch Normalization*

Batch Normalization (BatchNorm) [IS15] es un método de reparametrización adaptativa motivado por la dificultad de entrenar modelos muy profundos. En una capa de BatchNorm se estandariza la entrada a través de escalarla y trasladarla, lo que ayuda a estabilizar y acelerar el entrenamiento [Fei24]. Para cada *mini-batch*, se realiza el siguiente proceso:

1. Se calcula la media $\nu_B = \frac{1}{m}\sum_{i=1}^{m} x_i$ y la varianza $\sigma_B^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \nu_B)^2$.
2. Se normaliza la entrada: $\hat{x}_i = \frac{x_i - \nu_B}{\sqrt{\sigma_B^2 + \epsilon}}$, donde ϵ es una constante pequeña para evitar la división por 0.
3. Se escala y se traslada la entrada normalizada: $y_i = \gamma\hat{x}_i + \beta$, donde γ y β son parámetros aprendibles por el modelo.

Normalizando la entrada conseguimos hacer el proceso de entrenamiento más estable e intentar evitar el problema de la explosión o desvanecimiento del gradiente. También proporciona flexibilidad y mejora el rendimiento al reescalar y trasladar la entrada, y que esto dependa de parámetros aprendibles.

7.2.5. Capa totalmente conectada

Al final de las redes convolucionales lo más común es encontrarnos una o varias capas totalmente conectadas o *fully connected* (FC), es decir, que cada neurona está conectada a todas las neuronas de la capa anterior, de la misma manera que ocurre en un MLP. Esta parte de la red permite clasificar las características extraídas por las capas convolucionales [Fei24].

7.3. *Residual Networks*

Las *Residual Networks* (ResNets) [He+16] son una familia de modelos dentro de las ConvNets que también usaremos en la experimentación, en versiones de 15 y 57 capas. Su característica principal es que usan bloques residuales, que agrupan varias capas en los cuales se suma la identidad (la entrada al bloque) a la salida del bloque. Que las redes neuronales profundas aprendan esta función identidad previene el problema de la degradación, es decir, que el rendimiento de la red decaiga a medida que aumenta el número de capas. Esto puede surgir por varias causas como la inicialización de los pesos, la función de activación o el desvanecimiento/explosión del gradiente [Zha+23].

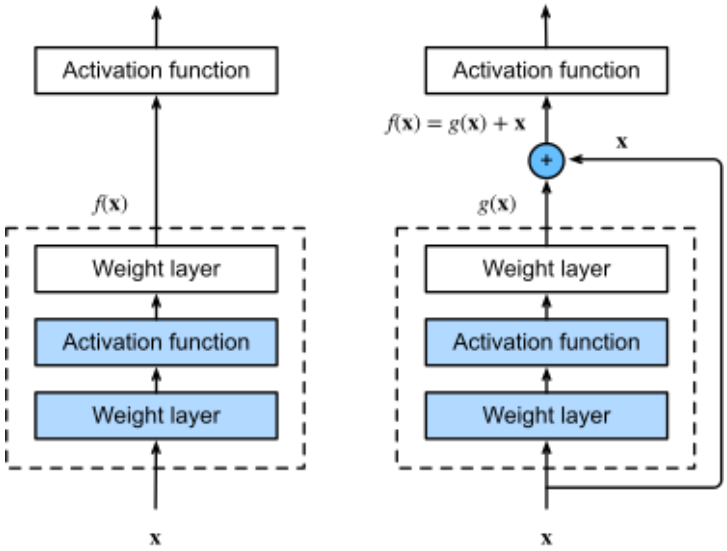

Figura 17: La imagen muestra una comparación entre una red neuronal tradicional y una red residual. En la red tradicional (izquierda), la entrada x pasa a través de una capa de pesos y una función de activación para producir la salida $H(x)$. En la red residual (derecha), la entrada x se suma a la salida de una capa de pesos y una función de activación, formando $H(x)+x$. Esta estructura de suma de atajos permite que la red residual aprenda funciones de identidad más fácilmente, lo que facilita el entrenamiento de redes profundas al mitigar el problema del desvanecimiento del gradiente. Obtenida de [Zha+23].

7.3.1. Bloques residuales

La función identidad se aprende a través de las conexiones residuales, que conectan el inicio y el final de los bloques residuales pasando la identidad. Estas conexiones además permiten aliviar el problema del desvanecimiento de gradiente. Vamos a centrarnos en una red neuronal de manera local, como se muestra en la Figura 17.

Si la función identidad $f(x) = x$ es el mapeo subyacente deseado, la función residual equivale a $g(x) = 0$ y, por tanto, es más fácil de aprender: sólo tenemos que llevar a cero los pesos y sesgos de la última capa de pesos dentro de la línea de puntos. Con los bloques residuales, las entradas pueden propagarse más rápidamente a través de las conexiones residuales entre capas.

Para ello necesitamos que la entrada y la salida del bloque tengan el mismo tamaño. Si reducimos la dimensionalidad de la entrada o aumentamos el número de filtros entonces deberemos modificar la entrada a través de convoluciones 1×1 para que tenga el mismo tamaño que la salida, como se

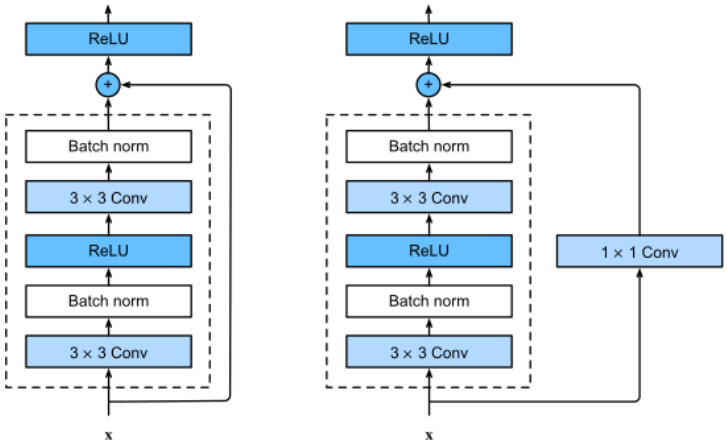

Figura 18: La imagen muestra dos bloques residuales utilizados en ResNets. A la izquierda, se presenta un bloque residual básico que consiste en dos capas convolucionales de 3×3 con una función de activación ReLU entre ellas, y una conexión de atajo que suma la entrada original al resultado de las convoluciones. A la derecha, se muestra un bloque residual con una convolución 1×1 adicional, que se utiliza para ajustar las dimensiones de la entrada antes de la suma. Este bloque también incluye dos capas convolucionales de 3×3 con funciones de activación ReLU entre ellas. La convolución 1x1 es crucial para permitir que la suma de la conexión de atajo sea posible cuando las dimensiones de la entrada y la salida no coinciden, mejorando así la capacidad de la red para aprender características complejas sin aumentar significativamente el costo computacional. Obtenida de [Zha+23].

muestra en la Figura 18.

7.3.2. Convoluciones 1x1

Los bloques residuales nos permiten aumentar la profundidad de la red evitando ciertos problemas asociados, pero al añadir más capas estamos aumentando considerablemente el número de parámetros. Las convoluciones con tamaño de filtro 1×1 son una herramienta poderosa para reducir el número de parámetros manteniendo la expresividad de la red. Fueron presentadas en [LCY14]. Este tipo de convoluciones se realizan antes de realizar la convolución requerida, de manera que la dividamos en dos, una con tamaño de filtro 1 y la otra con el tamaño original.

Ejemplo 7.1 *Para ver la diferencia en el número de parámetros al usar esta herramienta, calcularemos los parámetros necesarios para realizar una convolución en el caso de tener* $C = 256$ *canales de entrada,* $O = 512$ *canales de salida y tamaño del filtro* $F = 3$.

Si no usamos convoluciones 1×1*, tendríamos* $P = F \times F \times C \times O + O =$ 1.180.160 *parámetros.*

Usándolas debemos elegir un tamaño de filtro intermedio, por ejemplo $C' = 64$*. Aplicamos primero la convolución* 1×1*:* $P'_1 = 1 \times 1 \times C \times C' + C' =$ 16.448 *parámetros. A continuación realizamos la convolución con el tamaño de filtro original:* $P'_2 : F \times F \times C' \times O + O = 295.424$*. En total, sumando las dos capas tendríamos* 311.872 *parámetros, unas cuatro veces menos que en el caso anterior.*

7.4. Optimizadores basados en gradiente descendente

Con el objetivo de intentar abordar los principales problemas del algoritmo de aprendizaje del GD se han propuesto en la literatura diversas variantes, modificando la regla de actualización de los pesos. Existen optimizadores de primer y segundo orden, en función de si hacen uso sólo de la información del gradiente o también de la matriz Hessiana, respectivamente. Vamos a ver en esta sección únicamente los de primer orden, y veremos un método de segundo orden en la sección siguiente como parte de una MH memética. Se dan tres enfoques en este ámbito: el uso de momento, tasas de aprendizaje adaptativas y la combinación de los dos anteriores. De cada uno hacemos hincapié en el que vamos a usar en el presente TFG, en orden respectivo: NAG, RMSProp, Adam y AdamW.

7.4.1. NAG

El algoritmo del GD es problemático en regiones de la función de error donde una dimensión tiene mucha más pendiente que otra, que son comunes alrededor de óptimos locales. En estos escenarios el algoritmo oscila y realiza poco progreso real. El momento [Qia99] es un método que acelera al algoritmo en la dirección relevante y compensa las oscilaciones, como podemos ver en la Figura 19. Esto se realiza añadiendo una fracción γ del vector gradiente de la última iteración al vector gradiente actual [Zha+23; GBC16].

$$\begin{aligned} v_t &= \gamma v_{t-1} + \eta \nabla C(W_t) \\ W_{t+1} &= W_t - v_t. \end{aligned}$$

El valor de γ se sitúa normalmente alrededor de 0.9. El término del momento se incrementa en las dimensiones en las que el gradiente apunta en la misma dirección y se reduce en las que el gradiente cambia de dirección, consiguiendo una convergencia más rápida y estable. Dotamos al algoritmo de cierta inercia para reducir la brusquedad en los cambios de dirección.

El optimizador *Nesterov Accelerated Gradient* (NAG) [Nes83] modifica esta idea de manera que podamos "predecir" a dónde nos lleva esa inercia. A

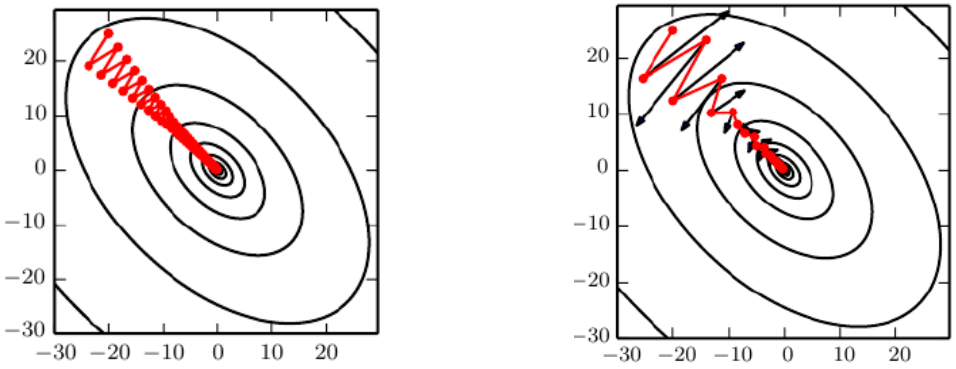

Figura 19: Comparación entre el algoritmo de GD estocástico original (izquierda) y usando el momento (derecha). Vemos que en la figura de la derecha se diferencia la dirección del gradiente (flechas negras) y el camino seguido por el algoritmo en rojo. Se observa que usando el momento se producen oscilaciones al principio que se van reduciendo a lo largo de la ejecución del algoritmo. Usando el momento se necesitan menos iteraciones para converger. Imágenes obtenidas de [GBC16].

la hora de calcular el gradiente de la función de coste, no lo hacemos respecto a los parámetros, sino respecto a una aproximación de los parámetros tras la iteración actual, de manera que podamos saber de forma aproximada dónde nos encontraremos después de actualizar los pesos. Se puede interpretar como una corrección del método de momento original. El valor del momento se sitúa también alrededor de 0.9.

$$\begin{aligned} v_t &= \gamma v_{t-1} + \eta \nabla C(W - v_t) \\ W_{t+1} &= W_t - v_t. \end{aligned}$$

Mientras que usando el optimizador momento original primero calculamos el gradiente y luego realizamos un salto grande en la dirección del gradiente acumulado, NAG primero realiza un salto grande en la dirección del gradiente acumulado, mide el gradiente y despúes realiza una corrección. Esta comparación se ilustra en la Figura 20. Esta estrategia previene al algoritmo de avanzar demasiado rápido.

7.4.2. RMSProp

RMSProp (Root Mean Square Propagation) es un optimizador de primer orden que introduce tasas de aprendizaje adaptativas. Presentado por Geoff

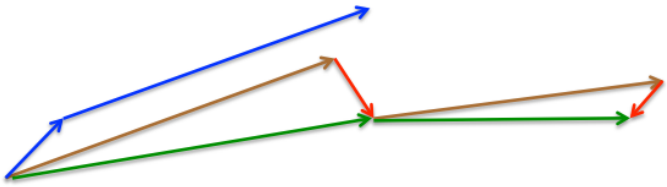

Figura 20: Comparación entre los métodos del momento original (vectores azules) y el momento de Nesterov, que muestra el paso en dos iteraciones del algoritmo de GD para mostrar las diferencias en su cálculo. En este último, primero realizamos un salto grande en la dirección del gradiente acumulado (vector marrrón) para luego medir el gradiente de la posición al acabar el salto y realizar una corrección (vector rojo). La flecha verde indica la posición final corregida donde acaba una iteración del método NAG. Obtenida de [Hin12].

Hinton en sus materiales docentes [Hin12], la fórmula de actualización de los pesos es la siguiente:

$$E[g]_t = 0.9E[g]_{t-1} + 0.1\nabla C(W_t)^2$$
$$W_{t+1} = W - t - \frac{\eta}{\sqrt{E[g_t^2]_t + \epsilon}}\nabla C(W_t)$$

Donde $E[g^2]_t$ es la media móvil en la iteración t, que depende solamente de la iteración anterior y del gradiente actual. Las operaciones anteriores se realizan elemento a elemento, es decir, cada peso recibe una actualización con un factor personalizado. Actualizando los pesos de esta forma, cuando el gradiente es grande en una dirección entonces el factor de ese peso será pequeño, evitando oscilaciones; mientras que si en otra dirección el gradiente es relativamente pequeño entonces el valor será grande, acelerando el proceso de entrenamiento.

Existen otros optimizadores que usan tasas de aprendizaje variables como Adagrad [DHS11], sus diferencias residen en la ventana de iteraciones y el cálculo del factor de multiplicación del peso. En este optimizador sólo se tienen en cuenta la iteración pasada y la actual, mientras que el uso de una media exponencial decreciente permite que las tasas de aprendizaje no se vuelvan demasiado pequeñas [Zha+23; GBC16].

7.4.3. Adam

Adaptative Moment Estimation (Adam) [KB14] es otro método que calcula tasas de aprendizaje adaptativas para cada parámetro. Además mantiene una media exponencial decreciente de gradientes de iteraciones anteriores m_t similar al momento.

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1)\nabla C(W)$$
$$v_t = \beta_2 v_{t-1} + (1 - \beta_2)\nabla C(W)^2.$$

m_t y v_t son estimaciones del momento de primer orden (media) y de segundo orden (varianza no centrada) de los gradientes, respectivamente. Son inicializados como vectores de 0, por lo que sus autores encontraron que tenían un sesgo al 0, especialmente durante las primeras iteraciones y cuando β_1 y β_2 son próximos a 1. Por tanto se calculan nuevas variables corrigiendo el sesgo:

$$\hat{m}_t = \frac{m_t}{a - \beta_1}$$
$$\hat{v}_t = \frac{v_t}{1 - \beta_2}.$$

Ahora se usan para ajustar los parámetros como hemos visto en el optimizador anterior:

$$W_{t+1} = W_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon}\hat{m}_t$$

Los autores proponen valores por defecto de 0.9 para β_1, 0.999 para β_2 y 10^{-8} para ϵ [Zha+23; GBC16].

7.4.4. AdamW

AdamW (Adam con Weight decay) [LH19] es una modificación de Adam que arregla un error de implementación, desacoplando la regularización de la actualización del gradiente. En lugar de incorporar el decaimiento de pesos en el cálculo de gradiente, como en Adam original, AdamW directamente modifica los parámetros tras el paso de actualización del gradiente añadiendo una pequeña fracción del término de decaimiento de pesos, lo que conlleva una regularización más efectiva.

$$W_t = W_{t-1} - \eta \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} - \lambda \cdot W_{t-1}$$

7.4.5. L-BFGS-B

En el presente TFG utilizamos L-BFGS-B no como optimizador a utilizar en la experimentación, sino como componente del algoritmo SHADE-ILS. Aun así, este es en sí mismo un optimizador basado en gradiente, por lo cual aparece en esta sección.

El método L-BFGS-B (*Limited-memory Broyden-Fletcher-Goldfarb-Shanno with Box constraints*) [Byr+95] es un algoritmo Quasi-Newton, es

decir, un algoritmo de optimización iterativo. Los métodos de Newton usan la matriz Hessiana de la función a optimizar para usar más información del problema y ofrecer una convergencia más rápida y estable. Para problemas complejos de dimensionalidad elevada, calcular la Hessiana en cada paso es una tarea computacionalmente inabarcable, y los métodos de Quasi-Newton implementan una aproximación de la Hessiana para rebajar esta carga computacional. Estos métodos se diferencian entre ellos principalmente en la forma de aproximar la matriz Hessiana [NW06].

Uno de los métodos Quasi-Newton más populares es BFGS [ES96], que usa una aproximación de la Hessiana de forma que mantiene su propiedad de definida positiva, lo que asegura una convergencia estable. Sin embargo, aunque se reduce el coste computacional, para almacenar la matriz Hessiana se requiere demasiada memoria. El método L-BFGS [LN89], en lugar de guardar una matriz con $n \times n$ aproximaciones, guarda únicamente un vector de tamaño n que guarda todas las aproximaciones de manera implícita. Esta variante está diseñada para problemas de alta dimensionalidad, y produce resultados similares a su versión sin la memoria limitada [Fei24].

La última variante L-BFGS-B, que usamos en el presente TFG en el algoritmo SHADE-ILS, es una modificación que maneja restricciones en los valores de las variables, lo que la hace incorporar información sobre el dominio. Al usar información del gradiente y de la Hessiana, es un optimizador de segundo orden, aunque es mucho menos popular que los optimizadores de primer orden. Aunque mejora la rapidez y estabilidad de la convergencia al usar más información del problema y proporciona mejores soluciones, los problemas de aprendizaje automático a día de hoy han adquirido una dimensionalidad demasiado alta para que este tipo de métodos resulten computacionalmente asequibles, y se prefiere usar los de primer orden. Aún así vemos que se implementa dentro del algoritmo de SHADE-ILS con resultados muy satisfactorios, no usándose como método de optimización principal sino de manera complementaria al algoritmo SHADE [NW06].

7.5. Metaheurísticas

En su definición original las MH son métodos que combinan técnicas de mejora local con estrategias de alto nivel para crear un proceso capaz de escapar óptimos locales y realizar una búsqueda robusta del espacio de soluciones. Aunque no hay garantía teórica de que puedan encontrar la solución óptima, su rendimiento es muy superior en algunos casos al de algoritmos exactos que requieren demasiado tiempo para completar su ejecución, especialmente en problemas complejos del mundo real. En problemas NP-Difícil por ejemplo se prioriza el uso de MH que dan una solución cercana a la óptima en un tiempo mucho menor que algoritmos exactos [GP10].

Podemos clasificar a las MH en dos grandes grupos en función de cómo se realiza la búsqueda por el espacio de soluciones: basadas en trayectorias

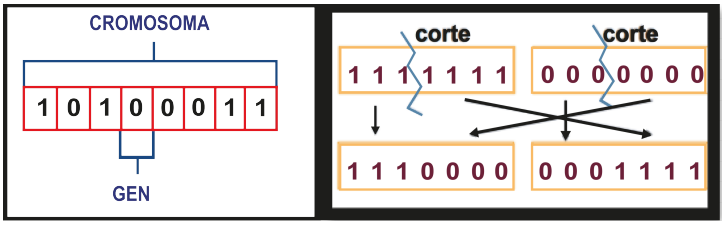

Figura 21: La imagen ilustra la representación y el proceso de cruce de cromosomas en algoritmos genéticos. A la izquierda, se muestra un cromosoma compuesto por una cadena de bits (1 y 0) en la parte superior destacando un gen específico dentro del cromosoma. A la derecha, se presenta el proceso de cruce (corte) entre dos cromosomas parentales a través del operador de cruce en un punto. Los cromosomas parentales, representados por las cadenas '1111111' y '0000000', se cortan en un punto específico, intercambiando segmentos para formar dos nuevos cromosomas hijos: '1110000' y '0001111'. Este proceso es fundamental en los algoritmos genéticos, ya que permite la combinación y variación de información genética para la optimización y solución de problemas complejos.

y basadas en poblaciones. En las primeras el proceso de búsqueda se caracteriza por realizar una trayectoria en el espacio de búsqueda, que puede ser visto como la evolución en tiempo discreto de un sistema dinámico. En las MH basadas en poblaciones, en cada iteración hay un conjunto de soluciones que interactúan entre sí. Nos centraremos en este último tipo ya que es el que vamos a usar [GP10].

7.5.1. Metaheurísticas basadas en poblaciones

Son técnicas de optimización probabilística que con frecuencia mejoran a otros métodos clásicos, e intentan imitar el mecanismo de evolución de la naturaleza a través de similitudes con la genética, como se ilustra en la Figura 21. Tiene un conjunto de soluciones denominado población, donde cada solución se llama individuo, y son generados de forma aleatoria. En cada iteración o generación, estos individuos se recombinan entre sí para intentar obtener mejores soluciones cuyo rendimiento es medido con una función objetivo. Las etapas de cada generación se pueden ver esquemáticamente en la figura 22, y son:

- Selección: se elige una parte de la población actual, normalmente con criterios elitistas (se elige a los mejores) aunque introduciendo cierta aleatoriedad. Si el número de individuos elegidos es igual al tamaño de la población, hablamos de un modelo generacional, mientras que si es menor hablamos de un modelo estacionario.

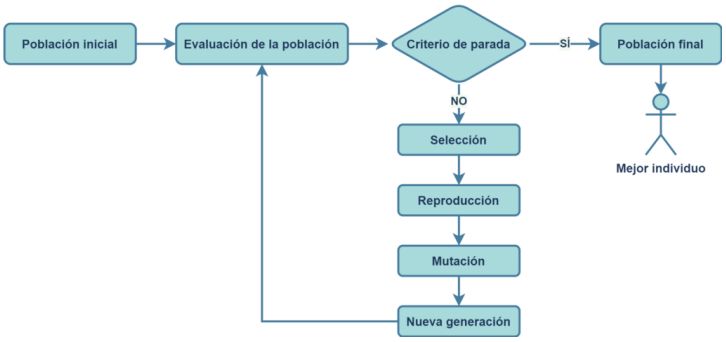

Figura 22: La imagen presenta un diagrama de flujo de un algoritmo genético, que incluye los siguientes pasos: iniciar con una "Población inicial", donde se crea una población de posibles soluciones, que se somete a una "Evaluación de la población", en la que cada individuo se puntúa según una función objetivo. Si se cumple el "Criterio de parada", se obtiene la "Población final" y se identifica el "Mejor individuo". Si el criterio no se cumple, se procede a la "Selección" de individuos, donde se seleccionan los individuos que se reproducirán a través de un proceso predefinido que asegure diviersidad en la población y buenas solucions, seguida de los procesos de "Reproducción" y "Mutación", en los que se cruzan los individuos seleccionados y luego se les aplica una pequeña modificación, respectivamente; para generar una "Nueva generación". Este ciclo se repite hasta que se cumple el criterio de parada, optimizando las soluciones tras cada generación. Obtenida de https://blogs.imf-formacion.com/blog/tecnologia/

- Cruce: Los individuos seleccionados se agrupan por parejas y se combinan a través del operador de cruce. Las soluciones resultantes se denominan hijos. Operadores comunes son el cruce en un punto, el cruce en dos puntos y el cruce uniforme.
- Mutación: A los hijos se les aplican cambios aleatorios en sus valores para mantener cierta diversidad genética en la población.
- Reemplazo: Se reemplaza la población actual con la nueva generación. Podemos reemplazarla entera o aplicar criterios elitistas, como reemplazar sólo con los mejores o reemplazar sólo si la nueva generación es mejor que la anterior.
- Terminación: se comprueba si se cumple la condición de parada. Criterios comunes son un número máximo de iteraciones o la convergencia de la población (falta de mejoras entre generaciones).

Los criterios elitistas hacen que la convergencia sea más rápida, pero podemos caer en una convergencia prematura por la falta de diversidad que

conllevan estos criterios, de manera que nuestro algoritmo pare antes de encontrar una solución lo suficientemente buena.

7.5.2. *Differential Evolution*

Los algoritmos de DE [SP97] son modelos basados en poblaciones que son particularmente efectivos para problemas de optimización continuos. Enfatizan la mutación y la realizan antes de aplicar el operador de cruce. Usan los parámetros factor de mutación F y probabilidad de cruce C_r [PSL05]. Las etapas que varían, descritas en el orden que se realizan en cada generación, son las siguientes:

- Operador de mutación: Para cada solución x_i de la población, se genera un vector mutante v_i a partir de la siguiente expresión:

$$v_i = x_{r1} + F \cdot (x_{r2} - x_{r3}).$$

  Donde x_{r1}, x_{r2} y x_{r3} son individuos seleccionados aleatoriamente con las restricciones de que $x_i \neq x_{rj}$ y $x_{rj} \neq x_{rj'}$ con $j, j' \in \{1, 2, 3\}$. F se suele situar en la práctica ente 0 y 2.

- Cruce: se combinan el vector solución de partida x_i y el vector mutante v_i para generar el vector de prueba u_i. Se usa el cruce binomial:

$$u_{ij} = \begin{cases} v_{ij} & \text{si } \text{rand}_j(0,1) \leq C_r \\ x_{ij} & \text{en otro caso} \end{cases}$$

  donde $\text{rand}_j(0, 1)$ es generado aleatoriamente con una distribución uniforme entre 0 y 1 para cada componente j.

- Selección: se compara el valor de la función objetivo de los vectores iniciales con el de los vectores de prueba correspondientes, y el que tenga mayor valor pasa a la generación siguiente.

En el pseudocódigo del Algoritmo 4 podemos apreciar el cambio de orden en las etapas de cada generación con respecto al esquema general de los algoritmos basados en poblaciones que veíamos en la Figura 22.

7.5.3. SHADE

SHADE (*Success-History based Adaptative Differential Evolution*) [TF13] es una variante avanzada del algoritmo original de DE. Consigue mejorar éste a través de guardar información histórica sobre configuraciones de los parámetros de factor de mutación (F) y el ratio de cruce (CR) que han tenido buenos resultados para poder ajustar de manera adaptativa

Algorithm 4 Esquema general de DE

$t := 0$
Inicializar Pob_t
Evaluar $x \quad \forall x \in \text{Pob}_t$
while No se cumpla condición de parada **do**
  $t := t + 1$
  Mutar Pob_t para obtener Pob'
  Recombinar Pob' y Pob para obtener Pob"
  Evaluar Pob"
  Reemplazar Pob_t a partir de Pob" y Pob_{t-1}
end while
  return $x_i \in Pob_t : f(x_i) \leq f(x_j)$

estos parámetros y guiar el proceso de evolución, su pseudocódigo puede observarse en el Algoritmo 5.

Los mecanismos de cruce y de selección son los mismos que en DE, variando principalmente el mecanismo de mutación. Para generar el vector mutante v_i a partir de la solución x_i, SHADE usa la siguiente estrategia[19]:

$$v_i = x_{r1} + F \cdot (x_p - x_i) + F \cdot (x_{r1} - x_{r2}).$$

Donde x_p es un individuo seleccionado aleatoriamente de entre los p mejores de la población y que es distinto a x_i. También se verifica que $x_i \neq x_{rj}$ y $x_{rj} \neq x_{rj'}$ con $j, j' \in \{1, 2\}$. En el algoritmo de SHADE se usa $p = 1$, es decir, se elige al mejor individuo de la población.

El algoritmo inicia los parámetros de factor de mutación y ratio de cruce al valor 0.5, y los va adaptando según se va ejecutando. Para ello mantiene un archivo de memoria, que se actualiza al final de cada generación y en el que se guardan parejas de los valores de los dos parámetros que han dado lugar a mejores soluciones. Al actualizar el archivo se usa la media de Lehmer[20] de manera que se le da más peso a las parejas de parámetros que mejor rendimiento obtienen. Al comienzo de cada generación el algoritmo obtiene valores de F y C_r para cada individuo basándose en el archivo de memoria e introduciendo pequeñas modificaciones.

SHADE está basado en DE, una familia de algoritmos de optimización que ofrece muy buenos resultados con parámetros continuos. En este ámbito, SHADE obtiene resultados del estado del arte en espacios de búsqueda de media-alta dimensionalidad, aunque no tan alta como el caso del entrenamiento de modelos de aprendizaje profundo [TF13]. En tareas donde la dimensionalidad es muy eleveda como la optimización de parámetros de

[19] Esta es la estrategia original, existen más modificaciones, aunque basadas en esta propuesta

[20] $Lehmer(X) = \frac{\sum_{x \in X} x^2}{\sum_{x \in X} x}$

Algorithm 5 Algoritmo SHADE

$t := 0$
Inicializar Pob_t
Inicializar A (archivo externo)
Inicializar M (memoria de parámetros)
Evaluar $x \quad \forall x \in \text{Pob}_t$
while evals $<$ total_evals **do**
  $t := t + 1$
  Seleccionar p soluciones para la mutación
  Mutar Pob_{t-1} para obtener Pob'
  Recombinar Pob' y Pob_{t-1} para obtener Pob"
  Evaluar Pob"
  Actualizar A y M a partir de Pob" y Pob_{t-1}
  Obtener Pob_t a partir de Pob" y Pob_{t-1}
end while
  return $x_i \in \text{Pob}_t : f(x_i) \leq f(x_j) \quad \forall j$

grandes modelos, su versión memética SHADE-ILS, que veremos a continuación, consigue muy buenos resultados.

7.5.4. Algoritmos meméticos

Los algoritmos meméticos son técnicas de optimización MH basadas en el interacción entre componentes de búsqueda globales y locales, y tienen la explotación de conocimiento específico del problema como uno de sus principios. De manera general se componen principalmente de un algoritmo basado en poblaciones al cual se le ha integrado un componente de búsqueda local [GP10].

Su principal diferencia con los algoritmos evolutivos tradicionales es que usan de manera concienzuda todo conocimiento disponible acerca del problema. Esto no es algo opcional sino que es una característica fundamental de los algoritmos meméticos. Al igual que los algoritmos genéticos se inspiran en los genes y la evolución, estas estrategias se inspiran en el concepto de "meme", análogo al de gen pero en el contexto de la evolución cultural. Normalmente se llama "hibridar" a incorporar información del problema a un algoritmo de búsqueda ya existente y que no usaba esta información.

Esta característica de incorporar información del problema está respaldada por fuertes resultados teóricos. En el teorema *No Free Lunch* [WM97] se establece que un algoritmo de búsqueda tiene un rendimiento acorde con la cantidad y calidad de información del problema que usa. Más precisamente, el teorema establece que el rendimiento de cualquier algoritmo de búsqueda es indistinguible de media de cualquier otro cuando consideramos el conjunto de todos los problemas.

7.5.5. SHADE-ILS

SHADE-ILS [MLH18] es un algoritmo memético para problemas de optimización continua a gran escala. Combina la exploración del algoritmo basado en poblaciones SHADE, usado en cada generación para evolucionar a la población de soluciones, con la explotación de una búsqueda local que se aplica a la mejor solución que se tenga en esa generación.

En la parte de búsqueda local, en el algoritmo original existe un mecanismo de elección para usar entre varias búsquedas locales, una de ellas L-BFGS-B. En el presente TFG se ha decidido usar sólo esta última, por facilidad de implementación y porque usa más información específica del problema. Por tanto no se detallará este mecanismo de elección entre búsquedas.

Las características fundamentales de esta técnica y que la diferencia con respecto a otros algoritmos meméticos son la elección de los algoritmos empleados (tanto el de búsqueda local como el basado en poblaciones) y su mecanismo de reinicio. Éste se activa cuando a lo largo de tres generaciones el rendimiento de la mejor solución no supera en más de un 5 % al de la anterior. En dicho caso, se elige una solución aleatoria de la población y se le aplica una pequeña perturbación usando una distribución normal y el resto de la población se vuelve a generar aleatoriamente. Cuando ocurre esto los parámetros adaptativos son reiniciados a los valores por defecto.

Cabe destacar que esto se realiza ya que SHADE-ILS mantiene los parámetros adaptativos del algoritmo SHADE entre generaciones. Esto tiene mucho sentido ya que al finalizar una ejecución de dicho algoritmo, sólo aplicamos búsqueda local a una solución, con lo que la gran mayoría de la población queda intacta y por tanto podemos reutilizar estos parámetros que se han ido adaptando a ella.

SHADE-ILS mantiene un variable para guardar la mejor solución hasta ahora y otra para guardar la mejor solución desde el último reinicio, devolviendo la primera cuando finaliza el algoritmo. En la versión utilizada se ha añadido además un array para guardar el histórico de las mejores soluciones junto con su fitness correspondiente, de manera que podamos analizar y representar las mejoras que realiza el algoritmo.

Vemos el pseudocódigo de la implementación realizada en el Algoritmo 6 y aclaramos algunas cosas que pueden no haber quedado del todo claras en favor de la claridad del pseudocódigo. Cuando generamos la población inicial, seleccionamos la peor y la mejor solución y la combinamos haciendo una media de sus elementos. A esa solución se le aplica la búsqueda local y se incluye en la población reemplazando a la peor solución. Se guardan los valores de mejora de las últimas 3 generaciones y en caso de que todas estén por debajo del 5 % se activa el mecanismo de reinicio.

En el artículo su publicación [MLH18] se atienden tres cuestiones, todas a través de técnicas MH: diseño de la arquitectura, optimización de hiperparámetros y entrenamiento de los parámetros de un modelo. Nos centrare-

Algorithm 6 Algoritmo SHADE-ILS

$t := 0$
Inicializar Pob_t
$x_0 :=$ (maximo+minimo)/2
$x_{\text{best}} :=$ L-BFGS-B(x_0)
$x_{\text{global}} := x_{\text{best}}$
while evals < total_evals **do**
  $t := t + 1$
  $f_{\text{prev}} := f(x_{\text{best}})$
  $x_{\text{best}}, \text{Pob}_t :=$ SHADE(Pob_{t-1})
  $x_{\text{best}} :=$ L-BFGS-B(x_{best})
  $\Delta f := \frac{f_{\text{prev}} - f(x_{\text{best}})}{f_{\text{prev}}} \times 100$
  if $f(x_{\text{best}}) < f(x_{\text{global}})$ **then**
    $x_{\text{global}} := x_{\text{best}}$
  end if
  if $\Delta f < 5$ **then**
    $g := g + 1$
  else
    $g := 0$
  end if
  if $g \geq 3$ **then**
    Reiniciar población conservando x_{best} con perturbación normal
    $g := 0$
  end if
end while
  return x_{global}

mos en la última. Se utilizan seis conjuntos de datos distintos con diferente complejidad, y en base a ésta, se elige una arquitectura de modelo concreta dentro de la familia de las ConvNets, de manera que tenga buen rendimiento en su entrenamiento a través de GD. Se utiliza el optimizador Adam. En el entrenamiento con SHADE-ILS se utilizan diferentes estrategias que hacen uso de la estructura por capas de los modelos de aprendizaje profundo, realizando el entrenamiento en los pesos de diversas capas, según la estrategia, mientras se mantienen congelados los demás. También se realiza el entrenamiento de todo el modelo a la vez.

Los resultados de la experimentación son claros: solo en una de las seis tareas el modelo entrenado con SHADE-ILS minimiza más la función de pérdida que el modelo entrenado con GD. Además, en todos los casos, el error de test es mayor. Cabe mencionar que la generalización en los modelos es bastante buena, manteniéndose estos errores en valores cercanos a los que se obtiene en el entrenamiento, y aumentando el error en proporciones

similares a lo que lo hace el modelo entrenado con Adam.

7.6. Tests estadísticos

Un test estadístico es una herramienta utilizada para evaluar hipótesis sobre una población basada en una muestra de datos. Formalmente, un test estadístico se define como un procedimiento que, dado un conjunto de datos observados, proporciona una regla de decisión para aceptar o rechazar una hipótesis nula H_0, en función de la evidencia contenida en los datos. La hipótesis alternativa H_1 representa la afirmación contraria a H_0.

El p-valor representa la probabilidad de obtener un resultado tan extremo como el observado, bajo la suposición de que la hipótesis nula es verdadera. En otras palabras, mide la evidencia en contra de H_0: valores pequeños de p indican que la muestra es poco probable, lo que respalda la hipótesis alternativa H_1. Generalmente, se establece un umbral de significancia α, comúnmente 0.05 o 0.01, y para p-valores por debajo de ese umbral se rechaza la hipótesis nula.

Los tests estadísticos pueden clasificarse en diferentes categorías según su propósito, como tests paramétricos y no paramétricos, tests de comparación de medias, tests de independencia y tests de ajuste a una distribución específica. Los tests paramétricos asumen que los datos provienen de una distribución específica con ciertos parámetros desconocidos, los cuales se estiman a partir de la muestra. Estos tests suelen ser más potentes cuando se cumplen sus supuestos, como la normalidad de los datos.

7.6.1. Test de Shapiro-Wilk

Sea $X = \{x_1, x_2, , \ldots, x_n\}$ una muestra de tamaño n proveniente de una distribución desconocida. El test paramétrico de Shapiro-Wilk [SW65] evalúa la hipótesis nula H_0 de que la muestra sigue una distribución normal:

$$H_0 : X \sim \mathcal{N}(\mu, \sigma^2), \qquad H_1 : X \nsim \mathcal{N}(\mu, \sigma^2).$$

El estadístico W se define como $W = \frac{\left(\sum_n^{i=1} a_i x_i\right)^2}{\sum_n^{i=1}(x_i - \bar{x})^2}$ donde:

- x_i son los valores de la muestra ordenados de manera creciente.
- $\bar{x}$ es la media muestral.
- a_i son coeficientes óptimos predefinidos que dependen de la media, varianza y covarianza de los orden estadísticos de una muestra de tamaño proveniente de una distribución normal .

7.6.2. Test de los rangos con signo de Wilcoxon

El test de los rangos con signo de Wilcoxon [Wil45] es una prueba estadística no paramétrica que permite evaluar la hipótesis nula de que no existe diferencia significativa entre dos muestras emparejadas. Este test se emplea en contextos en los que no se puede asumir la normalidad de las distribuciones de las muestras y constituye una alternativa al test t de muestras relacionadas. Su utilidad se extiende a la comparación de condiciones de medición previas y posteriores o al análisis de diferencias entre dos muestras dependientes en cualquier tipo de experimento.

Supongamos que tenemos un conjunto de pares de observaciones (x_i, y_i), $i = 1, 2, \ldots, n$, donde x_i y y_i corresponden a los valores observados en las dos condiciones (por ejemplo, mediciones antes y después de un tratamiento). El objetivo del test es verificar si la mediana de las diferencias $\Delta_i = x_i - y_i$ es igual a cero, es decir, si no existe una diferencia sistemática entre las dos condiciones.

8. Estado del arte

En esta sección, el objetivo es analizar la literatura reciente y los artículos publicados sobre el entrenamiento de modelos de aprendizaje profundo, tanto a través de técnicas basadas en GD como MH. Para un mejor contexto, realizaremos una búsqueda en la base de datos de referencias bibliográficas y citas Scopus[21], con el fin de conocer el estado actual de la literatura.

Vamos a realizar una búsqueda en el ámbito del entrenamiento de modelos de aprendizaje profundo, diferenciando entre técnicas clásicas y técnicas MH. Para ello, utilizaremos términos definitorios y los nombres de las técnicas empleadas en este TFG, realizando las siguientes búsquedas con fecha del 11 de noviembre de 2024:

```
TITLE-ABS-KEY ( ( deep AND learning ) AND training AND
( metaheuristic OR metaheuristics OR shade OR shade-ils
OR ( differential AND evolution ) OR memetic OR
genetic ) ) AND ( LIMIT-TO ( SUBJAREA , "COMP" ) )
```

```
TITLE-ABS-KEY ( ( deep AND learning ) AND training AND
( gradient OR adam OR optimizer OR rmsprop OR nag ) )
AND ( LIMIT-TO ( SUBJAREA , "COMP" ) )
```

Obtenemos una cantidad de 6,753 artículos en lo referente al entrenamiento de modelos de aprendizaje profundo con técnicas basadas en GD y 997 para técnicas MH. **Vemos que la diferencia entre ambas cantidades es considerable, siendo la primera 7 veces mayor que la segunda**. Destacamos sin embargo que la tendencia en ambos casos es muy similar, como se observa en la Figura 23, aumentando prácticamente en la misma proporción desde el año 2012.

8.1. Gradiente descendente y optimizadores

El GD es el algoritmo de aprendizaje utilizado por defecto en prácticamente todas las tareas de aprendizaje profundo, gracias a su eficiencia y buenos resultados. Los problemas que surgen en su convergencia se intentan evitar en la práctica mediante el desarrollo de modificaciones a su algoritmo, denominadas optimizadores. La literatura en este ámbito es extensa, con una gran variedad de ellos disponibles. A continuación, realizaremos una distinción clara entre optimizadores de primer y de segundo orden.

Aunque los optimizadores de segundo orden presentan mejores propiedades teóricas y ofrecen una convergencia más rápida y estable al utilizar más

[21] https://www.scopus.com/

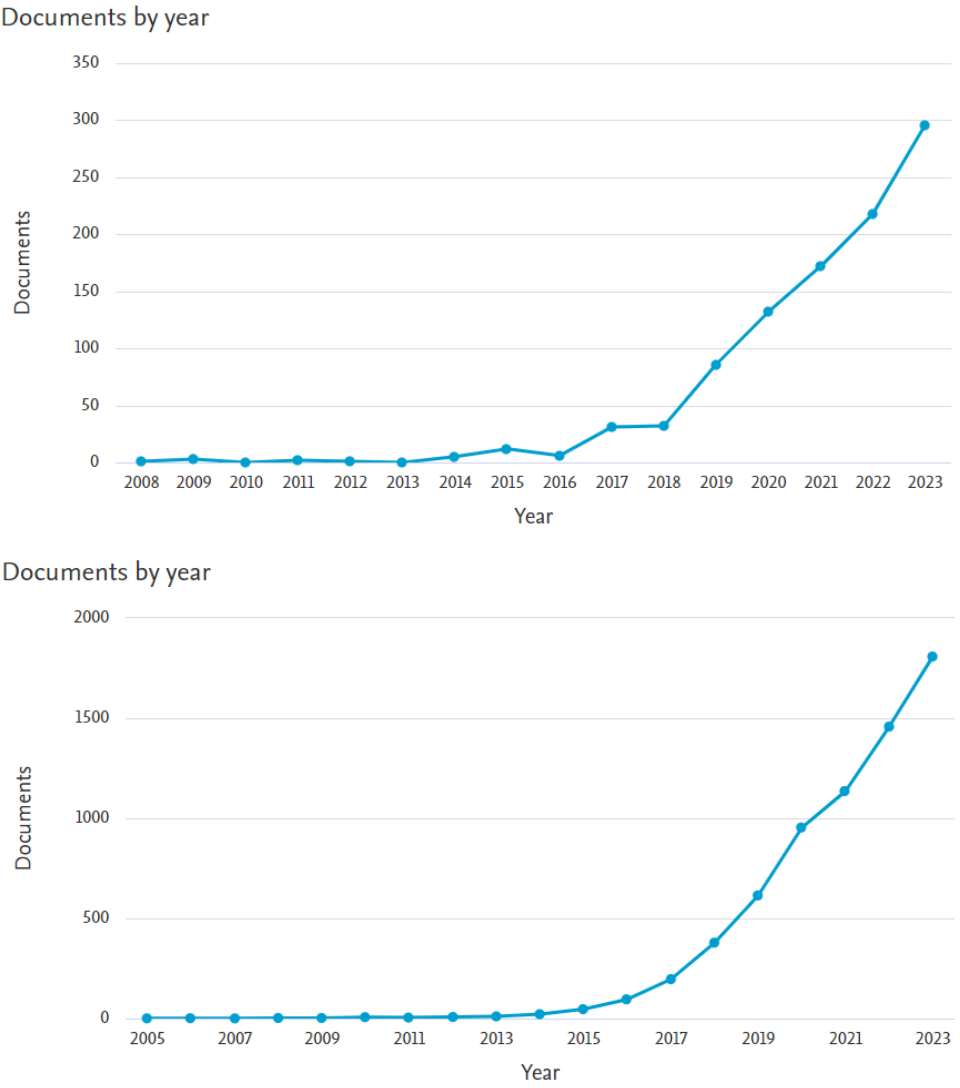

Figura 23: Número de documentos indexados anualmente en la base de datos Scopus relacionados con el entrenamiento de modelos de aprendizaje profundo mediante diferentes enfoques. La primera gráfica (arriba) muestra la cantidad de artículos que exploran el uso de MH para la optimización del entrenamiento de redes neuronales profundas, obteniendo un total de 997 publicaciones. La segunda gráfica (abajo) presenta el volumen de publicaciones centradas en el entrenamiento basado en GD, obteniendo un total de 6,753, lo que resulta en una diferencia de 7 veces mayor. Se observa que, si bien ambos enfoques han ganado interés en los últimos años, el uso de GD domina ampliamente la producción científica, con un crecimiento exponencial en el número de publicaciones desde 2017. La búsqueda se realiza a fecha del 11 de noviembre de 2024.

información del problema, el cálculo o la aproximación de la matriz Hessiana incrementa significativamente el poder computacional necesario para su uso, lo que ralentiza el entrenamiento. Además, hay un problema de memoria: para una red neuronal con 1 millón de parámetros, se requeriría almacenar una matriz de tamaño $1,000,000 \times 1,000,000$, que ocuparía aproximadamente 3,725 GB de memoria RAM. Esto resulta inviable, especialmente considerando que en el top-10 de modelos de clasificación de la competición *ImageNet*, ningún modelo tiene menos de mil millones de parámetros.

Incluso si eliminamos estos inconvenientes de memoria con métodos como L-BFGS (ver Sección 7.4.5), enfrentamos un problema significativo: estos métodos requieren el cálculo del gradiente sobre todos los datos de entrenamiento. Conjuntos como *ImageNet*, que contienen más de un millón de ejemplos, hacen que esto sea computacionalmente inviable. Conseguir que este tipo de algoritmos como L-BFGS funcionen con *batches* es más complejo que en MBGD y de hecho es un área abierta de investigación.

En la práctica no es común ver el algoritmo L-BFGS u otros optimizadores de segundo orden aplicados a modelos de aprendizaje profundo a gran escala. En su lugar se utilizan variantes de MBGD basadas en el uso de momento y en tasas de aprendizaje adaptativas, ya que son mucho más simples y más escalables. Existen varias opciones bastante asentadas, que forman parte de las librerías de aprendizaje automático más usadas como PyTorch y TensorFlow, entre las cuales destacan Adam, NAG, RMSProp, AdaGrad o SGD con momento. Vamos a analizar la comparativa [Kyr20] entre los optimizadores basados en GD con rendimiento del estado del arte para obtener una visión general.

En ella se diferencia entre los algoritmos que tienen tasa de aprendizaje adaptativa (Adam, AMSGrad, AdamW, QHAdam, Demon Adam, YellowFin) y los que no (SGDM, AggMo, QHM, Demon Momentum). Una consideración muy importante que se realiza en dicha comparativa, y que es bien sabida en el campo del aprendizaje automático, es que el rendimiento de una técnica de entrenamiento está muy ligado al dominio específico del problema. Puede ocurrir que un método que no sea de los mejores en términos generales sí lo sea en un problema específico. Pasamos ahora a describir rápidamente las técnicas más interesantes.

YellowFin [ZM19] es un optimizador con tasa de aprendizaje y momento adaptativos, de manera que mantiene dichos hiperparámetros en un intervalo donde el ratio de convergencia es una constante igual a la raíz del momento. AdamW es una extensión de Adam en la que se utiliza penalización en los pesos del modelo de manera que exista un sesgo hacia valores más pequeños de los mismos durante el entrenamiento, ya que normalmente se asocian valores grandes en los parámetros con el sobreajuste. Aunque Adam ya incorpora este mecanismo, AdamW realiza una pequeña modificación a través de desacoplar esta penalización de la actualización del gradiente, resultando en un impacto notable.

QHM es una extensión del método de momento clásico que introduce un término cuasi-hiperbólico. Esto permite una mezcla controlada entre el momento y el algoritmo de GD original, proporcionando una mayor flexibilidad y mejorando la estabilidad del entrenamiento. QHAdam combina las ventajas del optimizador Adam con las del GD con momento cuasi-hiperbólico. Introduce factores de amortiguación para controlar la contribución de las medias móviles de primer y segundo orden, ofreciendo un equilibrio entre estabilidad y rapidez en la convergencia.

Demon Adam es una variante de Adam que ajusta dinámicamente el momento durante el entrenamiento. Utiliza una estrategia de decaimiento del momento para mejorar la adaptación a diferentes fases del entrenamiento, permitiendo una mejor convergencia y evitando caer en mínimos locales. Similar a Demon Adam, Demon Momentum aplica la técnica de decaimiento del momento, pero se usa con optimizadores basados solo en el momento clásico, no en Adam. Mejora la capacidad de adaptación del optimizador durante el entrenamiento al ajustar el momento de manera dinámica. AggMo combina múltiples trayectorias de momento con diferentes factores de decaimiento. Esto ayuda a mejorar la exploración del espacio de parámetros y a mitigar la dependencia de los hiperparámetros del momento, proporcionando una convergencia más robusta y rápida.

Como conclusión, y atendiendo siempre al dominio específico del problema, se establece que YellowFin es la mejor opción en caso de no disponer de recursos para ajustar los hiperparámetros, ya que adapta el momento y la tasa de aprendizaje a lo largo del entrenamiento. Si se dispone de recursos, pero no demasiados, lo mejor son algoritmos de tasa de aprendizaje adaptativa de manera que sólo se tenga que ajustar el valor del momento; en concreto destacan AdamW, QHAdam y Demon Adam. En cambio si se quiere obtener el mejor rendimiento a toda costa, invirtiendo muchos recursos en el ajuste de parámetros, usar MBGD con momento es la mejor opción, aunque sea un método más clásico.

8.2. Metaheurísticas en el entrenamiento de modelos

Aún con el uso de optimizadores, hay ciertos inconvenientes en el entrenamiento que son insalvables, como los que están provocados por los cálculos del gradiente con el algoritmo de *backpropagation*. Las técnicas MH son una gran alternativa, ya que sus operadores de búsqueda no dependen de *backpropagation*, evitando así sus problemas.

Uno de los enfoques de aplicación de estas técnicas es la combinación con las técnicas clásicas, utilizando diferentes aproximaciones. Por ejemplo en [Ban19] se usa el algoritmo *Artificial Bee Colony* [Kar05] sobre un conjunto de soluciones aleatorias para generar una población inicial de conjuntos de parámetros de un modelo que se entrenan con GD. En [PKN18] se combina un algoritmo genético con el GD, de manera que las nuevas soluciones

son generadas con el operador de búsqueda del primero pero son evaluadas tras realizar varias épocas con el segundo. De manera similar en [Kha+17] se usa la técnica MH *Particle Swarm Optimization* [BM17] para entrenar los parámetros de la última capa de una ConvNet, mientras que el resto se entrenan a través del algoritmo de GD. La comparación arroja que la hibridación de ambas técnicas alcanza mejores resultados en términos de rapidez de convergencia y de *accuracy*. Prácticamente la totalidad de la literatura referente a esta estrategia está centrada en ConvNets.

Otro enfoque es entrenar el modelo usando exclusivamente algoritmos MH. En este ámbito destacan los estudios [RFA15] y [Ayu+16], en los que se proponen dos técnicas basadas en *Simulated Annealing* [KGV83] para entrenar los parámetros de una ConvNet, consiguiendo mejor rendimiento y mayor velocidad de convergencia que en el mismo modelo entrenado mediante el algoritmo de GD. Al igual que ocurre con el enfoque anterior, la gran mayoría de estos estudios están centrados en ConvNets y *Recurrent Neural Networks*.

Algo importante a destacar en la literatura de entrenamiento de modelos con técnicas MH es la falta de un marco común en los estudios, lo que impide realizar comparaciones objetivas entre ellos. Esta cuestión, comentada con más detalle en la Sección 6.1, evidencia la necesidad de más experimentos bajo condiciones similares para poder sacar conclusiones objetivas entre ellos.

El rendimiento de estas técnicas todavía no es comparable al de las técnicas clásicas. Si bien es cierto que para tareas sencillas y modelos con pocos parámetros pueden mejorar al GD en la minimización de la función de pérdida, generalmente en términos de generalización su rendimiento es inferior. Además, es importante considerar la complejidad computacional: para alcanzar un rendimiento similar al del GD, estas técnicas requieren mucho más tiempo y recursos computacionales, por lo que no resultan una alternativa viable para este tipo de tareas.

8.3. Neuroevolución

La neuroevolución es un enfoque que combina algoritmos evolutivos con redes neuronales, con el objetivo de optimizar sus pesos, arquitecturas o reglas de aprendizaje [Xin99]. A diferencia de los métodos de optimización basados en gradiente, la neuroevolución utiliza principios inspirados en la evolución biológica para desarrollar y entrenar redes neuronales, ofreciendo alternativas robustas especialmente en escenarios donde el cálculo de gradientes resulta complejo o inviable.

El paradigma de la neuroevolución se basa en la aplicación de operadores evolutivos (selección, cruce y mutación) para desarrollar redes neuronales, sin depender de información de gradiente. [WSB90] establecieron algunas de

las bases iniciales al demostrar cómo los algoritmos genéticos podían optimizar tanto las conexiones como la conectividad en redes neuronales. Este enfoque resulta particularmente valioso en problemas con espacios de búsqueda no diferenciables, funciones objetivo ruidosas o con múltiples óptimos locales [Sta+19]. La neuroevolución puede aplicarse a diferentes aspectos de las redes neuronales:

- Optimización de pesos: evolucionamos los parámetros del modelo.
- Evolución de arquitecturas: donde la topología de la red (número de capas, neuronas y conexiones) es determinada evolutivamente.
- Evolución de reglas de aprendizaje: donde los propios algoritmos de aprendizaje son evolucionados.

El algoritmo *NeuroEvolution of Augmenting Topologies* (NEAT), propuesto por [SM02], representa uno de los avances más significativos en neuroevolución. NEAT permite la evolución simultánea de la topología y los pesos de las redes neuronales, comenzando con estructuras mínimas que gradualmente se vuelven más complejas. Este enfoque ha demostrado capacidad para generar soluciones más compactas y eficientes que otros métodos neuroevolutivos convencionales. Sus características distintivas incluyen:

1. Operadores de mutación que pueden añadir nodos y conexiones.
2. Un sistema de marcado histórico que resuelve el problema de cruzamiento entre redes con diferentes topologías.
3. Especiación para asegurar innovaciones estructurales.

Con el auge del aprendizaje profundo, la investigación ha evolucionado hacia la aplicación de técnicas evolutivas en redes neuronales de múltiples capas. [DG14] presentaron un enfoque para evolucionar redes neuronales profundas mediante algoritmos genéticos, demostrando su viabilidad en problemas de clasificación. [Suc+17] ampliaron este campo al demostrar que los algoritmos genéticos constituyen una alternativa competitiva frente a métodos como el aprendizaje por refuerzo profundo. Su investigación reveló que la simple búsqueda de mutaciones en el espacio de pesos de redes neuronales puede superar a algoritmos complejos basados en gradiente en ciertas tareas de refuerzo.

La neuroevolución ha encontrado aplicaciones en diversas áreas:

1. **Aprendizaje por refuerzo**: Donde la ausencia de gradientes claros dificulta los métodos tradicionales. [Mii+19] demostraron la eficacia de la neuroevolución en entornos de aprendizaje por refuerzo complejos.

2. **Redes neuronales de impulsos (*Spiking Neural Networks*)**: [Pav+05] aplicaron algoritmos evolutivos para entrenar este tipo de redes, donde los métodos basados en gradiente resultan particularmente difíciles de implementar debido a la naturaleza discontinua de los impulsos neuronales.

3. **Problemas con funciones objetivo no diferenciables**: La neuroevolución permite optimizar redes neuronales respecto a criterios que no proporcionan gradientes, como métricas discretas o comportamientos emergentes complejos.

En artículos recientes, [Sta+19] sugieren que el futuro de la neuroevolución reside en su capacidad para descubrir arquitecturas novedosas y adaptativas, complementando más que compitiendo con los métodos basados en gradiente. Esta visión híbrida podría conducir a sistemas que combinen la eficiencia local del GD con la exploración global de los métodos evolutivos. En [SB08] se demuestra cómo los algoritmos de evolución diferencial pueden complementar los métodos tradicionales de entrenamiento, ofreciendo mejoras significativas en términos de tiempo de convergencia y calidad de las soluciones. La revisión comprehensiva de [KM23] sobre algoritmos MH para el entrenamiento de redes neuronales confirma esta tendencia hacia enfoques híbridos, donde la neuroevolución juega un papel fundamental en la optimización global del proceso de aprendizaje, mientras que los métodos basados en gradiente se encargan del refinamiento local.

8.4. Aprendizaje Automático Automatizado

El Aprendizaje Automático Automatizado (AutoML) emerge como una respuesta a la creciente complejidad y especialización requerida en el desarrollo de modelos de aprendizaje automático. Esta disciplina busca automatizar el proceso de selección, configuración y optimización de modelos, reduciendo la necesidad de intervención humana especializada y democratizando el acceso a técnicas avanzadas de aprendizaje automático [HKV19].

Esta automatización se fundamenta en métodos de búsqueda y optimización que evalúan iterativamente diferentes configuraciones, buscando maximizar métricas de rendimiento definidas [HKV19]. La relación entre AutoML y las MH es estrecha, ya que muchas técnicas de AutoML utilizan algoritmos evolutivos, búsqueda bayesiana u otros métodos MH como motores de optimización [KM23].

La optimización bayesiana constituye uno de los enfoques más utilizados en AutoML, modelando la función objetivo (el rendimiento del modelo) como un proceso gaussiano y utilizando técnicas de adquisición para seleccionar puntos de evaluación prometedores. Este enfoque permite balancear eficientemente la exploración y explotación en el espacio de hiperparámetros [SBF16].

Los algoritmos evolutivos ofrecen una alternativa robusta para la optimización en AutoML. En [AM06] documentan cómo diversos procedimientos MH, incluyendo algoritmos genéticos y evolución diferencial, pueden aplicarse efectivamente al entrenamiento y configuración de redes neuronales. Estos métodos resultan particularmente valiosos para espacios de búsqueda complejos y no diferenciables.

[Mii+19] proponen la evolución de redes neuronales profundas completas, incluyendo tanto arquitecturas como hiperparámetros, demostrando que los métodos evolutivos pueden generar soluciones competitivas para problemas complejos de aprendizaje automático.

Los algoritmos de *multi-armed bandit* y sus variantes permiten la asignación dinámica de recursos computacionales a configuraciones prometedoras, descartando eficientemente opciones subóptimas. Estos métodos han sido implementados en sistemas como *Hyperband* [Li+18b], que combina la búsqueda aleatoria con una estrategia de asignación de recursos basada en *bandits*.

Aunque AutoML busca reducir la necesidad de experiencia humana, la integración efectiva de conocimiento experto en el proceso automatizado representa un área de investigación prometedora. Los enfoques que permiten a los expertos guiar el proceso de búsqueda o incorporar restricciones específicas del dominio podrían mejorar significativamente la efectividad de los sistemas AutoML [HKV19]. Las direcciones futuras apuntan hacia sistemas más integrados que combinen diferentes enfoques de optimización, aprovechando tanto métodos basados en gradiente como MH evolutivas, y que puedan adaptarse dinámicamente a los requisitos específicos de cada problema [EMH19].

8.5. Búsqueda de Arquitectura Neuronal

Búsqueda de Arquitectura Neuronal (NAS, por sus siglas en inglés, *Neural Architecture Seach*) representa un subcampo especializado dentro del AutoML, centrado específicamente en la automatización del diseño de arquitecturas de redes neuronales. Esta disciplina busca sustituir el proceso manual de diseño arquitectónico por metodologías algorítmicas que puedan descubrir estructuras óptimas o casi óptimas para problemas específicos [EMH19]. El diseño manual de arquitecturas neuronales ha sido tradicionalmente un proceso que requiere considerable experiencia y conocimiento especializado, además de seguir un ciclo iterativo de prueba y error. Esta complejidad ha motivado el desarrollo de métodos automáticos que puedan explorar sistemáticamente el espacio de arquitecturas posibles [Bel+17].

Los métodos de NAS pueden clasificarse en tres grupos: basados en aprendizaje por refuerzo, evolutivos y basados en gradiente. Los primeros formulan el diseño arquitectónico como un problema de decisión secuencial, donde un agente aprende a seleccionar componentes arquitectónicos que ma-

ximizan métricas de rendimiento. El trabajo pionero de [Bel+17] demostró la viabilidad de este enfoque, utilizando redes neuronales recurrentes como controladores para generar descripciones de arquitecturas.

Los algoritmos evolutivos constituyen una alternativa robusta para NAS, al modelar las arquitecturas como individuos dentro de una población que evoluciona mediante operadores genéticos. Un ejemplo destacado de este enfoque es el trabajo de Xie et al., quienes propusieron un método basado en algoritmos genéticos para evolucionar arquitecturas de redes convolucionales, logrando resultados competitivos en tareas de clasificación de imágenes [XY17]. Posteriormente, Lu et al. ampliaron este enfoque con NSGA-Net, un algoritmo genético multiobjetivo diseñado para optimizar de manera simultánea la precisión y la complejidad computacional de las arquitecturas [Lu+19]. Este trabajo evidencia cómo los métodos evolutivos pueden abordar de manera eficaz la naturaleza multiobjetivo inherente al diseño de arquitecturas neuronales. Por otro lado, Stanley et al. destacan las ventajas de la neuroevolución en el contexto de NAS, subrayando su capacidad para explorar de forma eficiente espacios de búsqueda complejos y no diferenciables [Sta+19]. En esta línea, el algoritmo NEAT, desarrollado por Stanley y Miikkulainen, ofrece un marco natural para la evolución de topologías neuronales, lo que lo convierte en una estrategia adecuada para la búsqueda automatizada de arquitecturas [SM02].

A diferencia de los enfoques anteriores, los métodos basados en gradiente relajan el problema discreto de selección arquitectónica a un problema continuo, permitiendo la aplicación de técnicas de optimización basadas en gradiente. DARTS (*Differentiable Architecture Search*) ejemplifica este enfoque, utilizando relajaciones *softmax* para hacer diferenciable el proceso de selección de operaciones [LSY18].

La principal limitación de NAS es su coste computacional, lo que ha motivado diversas estrategias para mejorar su eficiencia, como compartir pesos entre las distintas arquitecturas candidatas [Pha+18] en ENAS (*Efficient Neural Architecture Search*), usar tareas simplificadas o predictores de rendimiento para ahorrar coste computacional [Bak+18], o descomposición jerárquica del problema en otros más pequeños (entrenar partes de la arquitectura por separado y luego combinarlas) [Mii+19; LSY18].

Los estudios comparativos entre arquitecturas generadas automáticamente y diseñadas manualmente han arrojado resultados significativos:

1. **Rendimiento**: En diversas tareas, las arquitecturas generadas mediante NAS han igualado o superado a las diseñadas manualmente. Por ejemplo, NASNet [Zop+18] superó el rendimiento de ResNet e Inception en tareas de clasificación de imágenes.

2. **Eficiencia computacional**: Mediante optimización multiobjetivo, como en NSGA-Net [Lu+19], es posible generar arquitecturas que

balancean precisión y eficiencia computacional, superando a modelos manuales en ambas dimensiones.

3. **Transferibilidad**: Las arquitecturas descubiertas en tareas específicas han demostrado buena capacidad de transferencia a otras tareas relacionadas, sugiriendo que NAS puede identificar estructuras con propiedades generalizables [Lu+19].

4. **Novedad estructural**: Los métodos de NAS han descubierto estructuras novedosas que difieren significativamente de los diseños convencionales, como conexiones densas no intuitivas o patrones de conexión irregulares [Rea+19].

La adaptación de NAS a dominios específicos como redes neuronales recurrentes, redes generativas adversarias o redes de grafos representa un área de investigación activa [Gao+20]. La integración de NAS con otros componentes del ciclo de aprendizaje automático, como la selección de características o la optimización de hiperparámetros, apunta hacia sistemas más comprensivos que puedan optimizar *pipelines* completos de aprendizaje automático [KM23]. La evolución de NAS refleja una tendencia más amplia hacia la automatización del diseño de sistemas de aprendizaje automático, donde los métodos evolutivos y otras MH juegan un papel fundamental al proporcionar marcos robustos para la exploración de espacios de diseño complejos y no diferenciables.

9. Métodos propuestos

Con el objetivo de explorar nuevas estrategias para el entrenamiento de modelos de aprendizaje profundo, **proponemos dos algoritmos meméticos originales** que, hasta donde sabemos, no han sido previamente estudiados en la literatura. Estos algoritmos combinan un optimizador basado en GD como búsqueda local y una técnica MH utilizada en la literatura para optimización de valores continuos como búsqueda global.

Los algoritmos meméticos han sido seleccionados debido a su **capacidad para aprovechar las ventajas de la búsqueda global (clave para evitar mínimos locales) y de la búsqueda local**, que incorpora conocimiento específico del problema y permite la convergencia hacia puntos críticos de la función de pérdida [MLH18]. Estas características proporcionan una base teórica sólida para obtener un buen rendimiento en el entrenamiento de modelos.

Las técnicas propuestas, **SHADE-GD y SHADE-ILS-GD**, se diferencian en la estrategia de búsqueda global empleada: la primera utiliza SHADE como técnica MH, mientras que la segunda emplea SHADE-ILS. Aunque SHADE-ILS ya es un algoritmo memético, pues incorpora una componente de búsqueda local, en nuestra propuesta añadimos una búsqueda local adicional mediante un optimizador basado en GD. Esto se justifica por el conocimiento específico que estos optimizadores tienen sobre el problema, lo que podría mejorar el rendimiento general del algoritmo. Además, esta combinación no ha sido ampliamente explorada en la literatura, y el diseño de algoritmos meméticos ofrece múltiples grados de libertad, lo que puede llevar a resultados interesantes.

La elección de SHADE y SHADE-ILS se basa en que **ambos pertenecen a la familia de algoritmos DE, la cual ha demostrado ser altamente efectiva en la optimización de parámetros continuos**, como es el caso del entrenamiento de modelos de aprendizaje profundo [TF13]. SHADE es un algoritmo bien establecido con un rendimiento sólido en diversos problemas, mientras que SHADE-ILS ha sido específicamente utilizado en la optimización de modelos de aprendizaje profundo, obteniendo buenos resultados en comparación con otras técnicas MH, aunque inferiores a los optimizadores basados en GD [MLH18]. En consecuencia, consideramos que estos algoritmos tienen un alto potencial de mejora mediante su hibridación con técnicas de GD.

Para la búsqueda local basada en GD, seleccionamos el optimizador en función de su rendimiento. Dado que el entrenamiento con optimizadores basados en GD es considerablemente más rápido que con MH, realizamos pruebas con NAG, RMSProp, Adam y AdamW para identificar el más adecuado en cada caso. Este proceso es computacionalmente eficiente y permite adaptar el algoritmo a cada tarea específica. En la mayoría de los experimentos, RMSProp y AdamW han demostrado ser las opciones más competitivas.

Por último, incorporamos el mismo mecanismo de reinicio de población de SHADE-ILS. **Al final de cada generación, evaluamos la mejora porcentual de la mejor solución respecto a la generación anterior. Si esta mejora es inferior al 5 % durante tres generaciones consecutivas, reiniciamos la población**. En cada reinicio, conservamos la mejor solución, generamos una nueva población y reintroducimos la mejor solución con una pequeña perturbación proveniente de una distribución normal. Mantenemos los valores originales de estos hiperparámetros de SHADE-ILS, aunque realizamos pruebas experimentales con valores cercanos, en las que observamos que sin este mecanismo la población tiende a converger prematuramente y deja de mejorar. Pasamos ahora a describir de forma más pausada el funcionamiento particular de cada propuesta. En la Figura 24 se puede observar el diagrama de flujo que representa el funcionamiento de ambos algoritmos.

9.1. SHADE-GD

La propuesta SHADE-GD (Algoritmo 7) es una variante del algoritmo SHADE en la que se incorpora un mecanismo de optimización basado en GD para mejorar el refinamiento de las soluciones encontradas. Esta hibridación permite aprovechar la capacidad exploratoria de SHADE y la eficiencia de GD en la explotación local.

1. Inicialización

El algoritmo comienza en $t = 0$ inicializando una población aleatoria de soluciones Pob_t, un archivo externo A que almacena soluciones descartadas para mejorar la diversidad, y una memoria M utilizada para adaptar los parámetros de mutación y recombinación. Posteriormente, cada solución de la población es evaluada.

Además, se definen dos contadores:

- g: Número de generaciones transcurridas.
- m: Número de generaciones consecutivas sin mejora significativa.

2. Bucle principal de optimización

El proceso de optimización se ejecuta mientras el número total de evaluaciones t no haya alcanzado el límite total_evals.

Cada generación en SHADE-GD consiste en un bloque de Gen_{SHADE} iteraciones SHADE (equivalente a una época de entrenamiento), asegurando que la evolución de la población tenga suficiente estabilidad antes de aplicar la optimización local.

Para cada iteración dentro de la generación:

1. Se actualiza el contador de evaluaciones t y la iteración interna i.

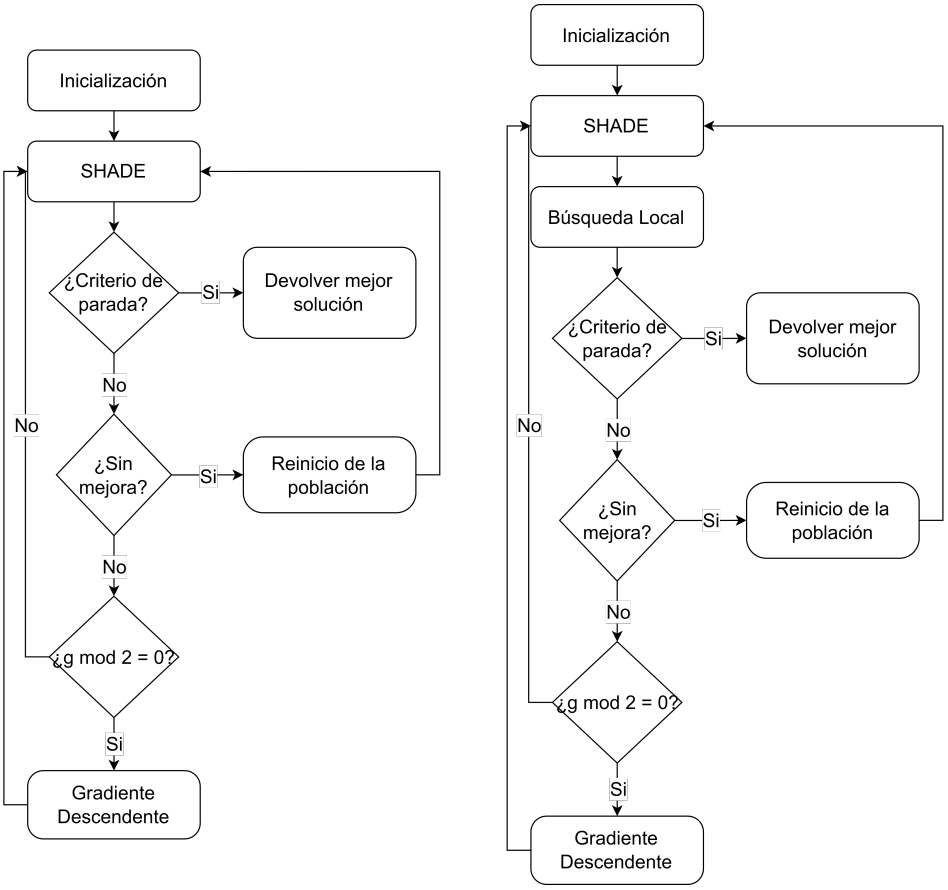

Figura 24: La figura muestra la estructura de SHADE-GD (izquierda) y SHADE-ILS-GD (derecha), destacando sus diferencias clave. Ambos algoritmos comienzan con una fase de inicialización en la que se genera una población de soluciones y aplican iterativamente el algoritmo SHADE para actualizar la población mediante operadores evolutivos. SHADE-ILS-GD introduce una etapa adicional de búsqueda local después de cada iteración de SHADE, utilizando L-BFGS-B para refinar la mejor solución encontrada. Luego, ambos algoritmos evalúan un criterio de parada, devolviendo la mejor solución si se cumple. En caso contrario, verifican si ha habido generaciones consecutivas sin mejora significativa y, de ser así, reinician la población conservando la mejor solución con una pequeña perturbación para evitar el estancamiento en óptimos locales. Además, cada dos generaciones, ambos algoritmos aplican descenso de gradiente a un individuo seleccionado aleatoriamente para mejorar la precisión sin comprometer la exploración global. SHADE-ILS-GD se diferencia de SHADE-GD por la inclusión del refinamiento mediante L-BFGS-B en cada iteración y por su estrategia de gestión de la mejor solución global a lo largo del proceso de optimización, lo que mejora la convergencia y reduce el riesgo de estancamiento.

Algorithm 7 Algoritmo SHADE-GD

$t := 0$
$g := 0$
Inicializar Pob_t
Inicializar A
Inicializar M
Evaluar $x \quad \forall x \in \text{Pob}_t$
$m := 0$
while $t <$ total_evals **do**
  $i := 0$
  $g := g + 1$
  while $i <$Gen$_{SHADE}$ **do**
    $t := t + 1$
    $i := i + 1$
    $f_{\text{prev}} := \min f(x) \quad \forall x \in \text{Pob}_{t-1}$
    Seleccionar p soluciones para la mutación
    Mutar Pob_{t-1} para obtener Pob'
    Recombinar Pob' y Pob_{t-1} para obtener Pob"
    Evaluar Pob"
    Actualizar A y M a partir de Pob" y Pob_{t-1}
    Obtener Pob_t a partir de Pob" y Pob_{t-1}
  end while
  if $g\text{mod}2 = 0$ **then**
    Seleccionar aleatoriamente $x \in \text{Pob}_t$
    $x := \text{GD}(x)$
  end if
  $f_{\text{new}} := \min f(x) \quad \forall x \in \text{Pob}_t$
  if $\frac{f_{\text{prev}} - f_{\text{new}}}{f_{\text{prev}}} < 0.05$ **then**
    $m := m + 1$
  else
    $m := 0$
  end if
  if $m \geq 3$ **then**
    Reiniciar población conservando la mejor solución con perturbación
    $m := 0$
  end if
end while
  return $x_i \in \text{Pob}_t : f(x_i) \leq f(x_j) \quad \forall j$

2. Se almacena el mejor valor de la función objetivo en la población anterior, f_{prev}.

3. Se seleccionan p soluciones para la mutación.

4. Se genera una población intermedia Pob' mediante mutación y una nueva población Pob" mediante recombinación.

5. Se evalúa Pob" y se actualizan tanto el archivo externo A como la memoria de parámetros M.

6. Se obtiene la nueva población Pob_t seleccionando los mejores individuos entre Pob_{t-1} y Pob".

3. Aplicación periódica de GD

Cada dos generaciones ($g \mod 2 = 0$), se selecciona aleatoriamente un individuo de la población y se somete a una optimización local mediante GD. Esta estrategia se justifica experimentalmente, ya que aplicar GD en cada generación conducía a un comportamiento similar al de un optimizador basado exclusivamente en GD, perdiendo la exploración propia de los algoritmos evolutivos.

Asimismo, la selección aleatoria del individuo evita la concentración de la búsqueda local sobre un único punto, lo que podría inducir sobreajuste y reducir la diversidad de la población.

4. Criterio de reinicio de población

Al finalizar cada generación, se evalúa la mejora relativa de la mejor solución respecto a la generación anterior:

$$\Delta f = \frac{f_{\text{prev}} - f_{\text{new}}}{f_{\text{prev}}} \times 100$$

Si la mejora Δf es inferior al 5 % durante tres generaciones consecutivas, se activa un mecanismo de reinicio en el que:

- Se reconstruye la población aleatoriamente.
- Se conserva la mejor solución encontrada hasta el momento.
- Se aplica una pequeña perturbación aleatoria a la mejor solución para evitar estancamientos en óptimos locales.

Las pruebas experimentales han demostrado que, sin este mecanismo, la población tiende a converger prematuramente y la mejora del algoritmo se detiene.

5. Finalización
El proceso continúa hasta que se alcanzan las total_evals evaluaciones. Finalmente, se devuelve la mejor solución encontrada en la población final:

$$x^* = \arg \min_{x_i \in \text{Pob}_t} f(x_i)$$

Esta solución corresponde al individuo con el menor valor de la función objetivo en la población final.

Conclusión
SHADE-GD mantiene la estructura de SHADE, pero incorpora un refinamiento periódico mediante GD, mejorando la eficiencia de la búsqueda local sin comprometer la capacidad de exploración. Además, el mecanismo de reinicio evita la convergencia prematura y promueve una exploración más efectiva del espacio de soluciones.

9.2. SHADE-ILS-GD

El algoritmo SHADE-ILS-GD (Algoritmo 8) es una extensión híbrida del algoritmo SHADE-ILS que combina tres enfoques de optimización: la exploración global de SHADE, el refinamiento local mediante L-BFGS-B y la optimización periódica con GD. Esta combinación tiene como objetivo mejorar la precisión de las soluciones y evitar la convergencia prematura.

1. Inicialización
El algoritmo comienza en $t = 0$, e inicia los siguientes elementos:

- Pob_t: Población inicial de soluciones generadas aleatoriamente.
- x_0: Punto medio del dominio de búsqueda, calculado como (máximo + mínimo)/2.
- x_{best}: Mejor solución optimizada localmente mediante el algoritmo L-BFGS-B aplicado a x_0. Es la mejor solución encontrada entre dos reinicios de la población.
- x_{global}: Mejor solución global encontrada durante la ejecución del algoritmo.
- g: Contador de generaciones consecutivas sin mejora significativa.

2. Bucle principal de optimización El algoritmo ejecuta iteraciones hasta que se alcanza el número máximo de evaluaciones total_evals. Cada iteración realiza las siguientes operaciones:

1. Se actualiza el contador de evaluaciones t.

Algorithm 8 Algoritmo SHADE-ILS-GD

$t := 0$
Inicializar Pob_t
$x_0 := (\max_{Pob_t} + \min_{Pob_t})/2$
$x_{\text{best}} :=$ L-BFGS-B(x_0)
$x_{\text{global}} := x_{\text{best}}$
$g := 0$
while t < total_evals **do**
  $t := t + 1$
  $f_{\text{prev}} := f(x_{\text{best}})$
  $x_{\text{best}}, \text{Pob}_t :=$ SHADE(Pob_{t-1})
  $x_{\text{best}} :=$ L-BFGS-B(x_{best})
  if $t$mod2 $= 0$ **then**
    Seleccionar aleatoriamente $x \in \text{Pob}_t$
    $x :=$ GD(x)
  end if
  $\Delta f := \frac{f_{\text{prev}} - f(x_{\text{best}})}{f_{\text{prev}}} \times 100$
  if $f(x_{\text{best}}) < f(x_{\text{global}})$ **then**
    $x_{\text{global}} := x_{\text{best}}$
  end if
  if $\Delta f < 5$ **then**
    $g := g + 1$
  else
    $g := 0$
  end if
  if $g \geq 3$ **then**
    Reiniciar población conservando la mejor solución con perturbación
    $g := 0$
  end if
end while
  return x_{global}

2. Se almacena la mejor solución anterior, $f_{\text{prev}} = f(x_{\text{best}})$.

3. Se ejecuta una generación completa del algoritmo SHADE sobre la población, obteniendo una nueva población Pob_t y la mejor solución x_{best}.

4. Se aplica el algoritmo L-BFGS-B a x_{best} para un refinamiento local un número de Evals_{LS} iteraciones.

3. Aplicación periódica de GD

Cada dos iteraciones ($t \mod 2 = 0$), se selecciona aleatoriamente un individuo de la población Pob_t y se optimiza localmente utilizando GD. Esta estrategia permite realizar un refinamiento adicional en puntos no explorados por L-BFGS-B y con un método distinto dentro de la misma estrategia, manteniendo la diversidad de soluciones.

4. Actualización de la mejor solución global

En cada iteración, si la mejor solución actual x_{best} mejora la mejor solución global x_{global}, se actualiza $x_{\text{global}} = x_{\text{best}}$.

5. Criterio de reinicio de población

Se calcula la mejora relativa de la función objetivo como:

$$\Delta f = \frac{f_{\text{prev}} - f(x_{\text{best}})}{f_{\text{prev}}} \times 100$$

Si la mejora relativa es inferior al 5 % durante tres generaciones consecutivas, se activa un mecanismo de reinicio que:

- Reconstruye la población de manera aleatoria.
- Conserva la mejor solución global x_{global}.
- Aplica una pequeña perturbación a x_{global} para escapar de óptimos locales.

Este mecanismo previene la convergencia prematura y fomenta una exploración más amplia del espacio de búsqueda.

6. Finalización

El algoritmo concluye cuando se alcanza el límite de evaluaciones total_evals, devolviendo la mejor solución global encontrada x_{global}:

$$x^* = x_{\text{global}}$$

Conclusión

SHADE-ILS-GD combina la exploración global de SHADE con dos técnicas de optimización local: L-BFGS-B para un refinamiento intensivo y GD para mejorar la precisión periódicamente. El mecanismo de reinicio asegura que el algoritmo mantenga la diversidad de soluciones y evita el estancamiento en óptimos locales, lo que lo hace particularmente efectivo para problemas complejos y de alta dimensionalidad.

10. Experimentación y resultados

En la tabla 2 se muestra un esquema de los optimizadores, modelos y conjuntos de datos utilizados en cada experimento.

Experimentos	Optimizadores	Conjuntos de datos	Modelos
1. Diferencia del rendimiento según la tarea	SHADE, SHADE-ILS, AdamW, RMSProp	BCW, BHP, WQR, WQC	MLP
2. Impacto de factores en el rendimiento	SHADE, SHADE-ILS, AdamW, RMSProp	BCW, WQC, MNIST, F-MNIST, CIFAR-10G	MLP, LeNet5, ResNet15, ResNet57
3. Análisis de tiempos de ejecución	SHADE-ILS, SHADE, AdamW	BCW, BHP, WQR, WQC, MNIST, F-MNIST, CIFAR-10G	MLP, LeNet5, ResNet15, ResNet57
4. Evaluación de propuestas propias	SHADE, SHADE-ILS, SHADE-GD, SHADE-ILS-GD	BCW, BHP, WQR, WQC, MNIST, F-MNIST, CIFAR-10G	MLP, LeNet5, ResNet15, ResNet57

Tabla 2: Descripción de los experimentos realizados, optimizadores evaluados, conjuntos de datos utilizados y modelos empleados en el estudio. El Experimento 1 compara el rendimiento los optimizadores MH SHADE y SHADE-ILS en relación al de los basados en gradiente AdamW y RMSProp, comprobando si existen diferencias en el rendimiento relativo según el tipo de tarea. En el Experimento 2, se analiza el impacto de diversos factores en el rendimiento de los algoritmos, empleando AdamW y RMSProp como representantes de GD y SHADE y SHADE-ILS como MH. El Experimento 3 evalúa los tiempos de ejecución de los optimizadores en distintos conjuntos de datos y modelos. SHADE se analiza de manera implícita, ya que constituye el núcleo del algoritmo SHADE-ILS. Finalmente, en el Experimento 4, se comparan las propuestas SHADE-GD y SHADE-ILS-GD con sus versiones originales, para posteriormente compararlas entre ellas. Se elige los optimizadores basados en GD AdamW y RMSProp ya que son los que mejores resultados obtienen en la experimentación de manera genreal. Adicionalmente, se emplean estos dos junto con los optimizadores Adam y NAG en pruebas preliminares para determinar cuál de ellos ofrece el mejor desempeño en cada tarea. Este análisis permite integrar el optimizador GD más adecuado dentro de SHADE-GD y SHADE-ILS-GD, maximizando así su eficiencia en cada contexto.

10.1. Entorno de ejecución y detalles de implementación

En esta sección se detallará el entorno de pruebas junto con las justificaciones de las elecciones realizadas a lo largo de la experimentación. Para el desarrollo del código se usa el lenguaje Python principalmente con las librerías PyTorch, FastAI, Numpy, SKlearn y Pandas; implementado y eje-

cutado en la plataforma Paperspace, que proporciona un IDE y un entorno de ejecución online similar a Google Colab, pero en el que podemos elegir manualmente el hardware sobre el que ejecutamos el código, de manera que la comparación de tiempos y recursos entre las distintas técnicas sea objetiva.

Las librerías utilizadas son: FastAI 2.7.17, NBdev 2.3.31, Ucimlrepo 0.0.7, Torchvision 0.16.1+cu121, Matplotlib 3.7.3, Scikit-learn 1.3.0, Scipy 1.11.2, Torch 2.1.1+cu121, Numpy 1.26.3, Pandas 2.2.0 y Pyade 1.0. El hardware usado es Nvidia Quadro P5000, proporcionado por la plataforma. El código puede encontrarse en: https://github.com/eedduu/TFG.

Dada la cantidad de modelos distintos que vamos a entrenar, se ha decidido dividir el código en un archivo por tarea, teniendo cada conjunto de datos su propio archivo `conjunto_de_datos.ipynb`. Se ha elegido este formato de archivo en lugar de `conjunto_de_datos.py` de manera que se puedan comprobar las salidas del proyecto fácilmente.

Se ha creado un módulo de python llamado `utilsTFG.py` que contiene funciones comunes al código, como métricas de error propias, herramientas para el preprocesado de datos, los modelos ConvNets, funciones para graficar resultados o los algoritmos MH. También hay un archivo `comparative.ipynb` donde se realizan comparativas a posteriori de los resultados, como por ejemplo graficar relaciones entre los rendimientos de algunas técnicas o llevar a cabo test estadísticos.

En todas las funciones y librerías usadas en las que intervienen generación de números aleatorios se fija el valor de su semilla a 42. Esto se hace al iniciarse el proyecto, de manera que afecte a la separación de los datos en entrenamiento, de test y a la generación de parámetros iniciales para los modelos que vamos a entrenar con GD. Luego se vuelve a fijar la semilla para todas las librerías que corresponda para generar la población que usaremos con las técnicas MH y se fija también de nuevo antes de iniciar cada entrenamiento, de manera que se puedan repetir los experimentos por separado.

Esto último también se realiza ya que la experimentación ha debido realizarse en ejecuciones separadas, y fijando la semilla de nuevo obtenemos el mismo estado para los generadores de números aleatorios, con lo que no tenemos problema al dividir las ejecuciones. Se han guardado además, a través de la librería `pickle`, tanto las poblaciones iniciales como los modelos y tiempos obtenidos para cada tarea, de manera que estos son comprobables.

10.1.1. Gradiente descendente

El criterio para elegir los optimizadores basados en GD ha sido el siguiente: en primer lugar decidimos incorporar tres técnicas basadas en GD para tener diversidad en los resultados, como se expone arriba, y en concreto ese es el número de estrategias distintas en los que se dividen a grandes

rasgos los optimizadores de primer orden (ver Sección 7.4). Para cada enfoque distinto, seleccionamos el optimizador en función del número de citas de su publicación, su uso en la literatura y su estandarización en librerías de aprendizaje automático. Por tanto hemos elegido **NAG, RMSProp, Adam y AdamW.**

Entrenamos usando la política de un ciclo de Leslie para alcanzar una convergencia más rápida. Para elegir el valor máximo de la tasa de aprendizaje usamos la función `lr_find()` de FastAI, opción usada ampliamente en la literatura. Hay que destacar que tres de los cuatro optimizadores que usamos tienen tasas de aprendizaje adaptativas, por lo que la elección de la tasa de aprendizaje es notablemente menos influyente en ellos.

Usamos 20 épocas para el entrenamiento de los modelos con estos optimizadores, valor obtenido del artículo comentado y comprobado experimentalmente que permite la convergencia en todos los entrenamientos. Durante el entrenamiento, guardamos los parámetros del mejor modelo en términos de error de validación, que será el que usemos para calcular el error de generalización y realizar las comparativas.

10.1.2. Metaheurísticas

Las principales librerías de aprendizaje automático no incluyen herramientas para entrenar a través de técnicas metaheurísticas ni para manejar los modelos usando estas técnicas, por lo que debemos realizar ciertas implementaciones que nos permitan integrarlas.

El algoritmo de SHADE se implementa a través de pyade[22], una librería de Python que nos permite usar varios algoritmos basados en DE controlando sus parámetros. Se le han realizado modificaciones para mantener los parámetros adaptativos del algoritmo SHADE entre ejecuciones distintas y adaptar las estructuras de datos a las del resto del código.

Dicho algoritmo optimiza un vector de valores flotantes, por lo que debemos crear las funciones necesarias para obtener los parámetros de un modelo en forma de array y luego volverlos a cargar en el modelo respetando la estructura por capas del mismo. Además se han creado funciones de coste que funcionan igual que las implementadas por FastAI, ya que están basadas en ellas, pero que manejan la estructura de datos que tenemos que usar con estos algoritmos. Dichas funciones permiten evaluar sobre el conjunto de entrenamiento, validación o test según corresponda.

A partir del algoritmo SHADE se implementan manualmente el resto. Para la búsqueda local L-BFGS que se usa en SHADE-ILS y SHADE-ILS-GD usamos la función que ofrece la librería Scipy. Como función de error en dicha búsqueda realizamos una modificación de la función de error antes mencionada para que devuelva además el gradiente, necesario para dicho

[22] https://github.com/xKuZz/pyade/tree/master

algoritmo. Para los algoritmos meméticos, a la hora de realizar el entrenamiento a través de GD, simplemente usamos las funciones mencionadas anteriormente para cargar los pesos en un objeto `learner` de FastAI y entrenar una época, para después devolver los pesos actualizados a la estructura de datos necesaria.

Cuando entrenamos los modelos usando técnicas MH tenemos que asignar muchos más recursos al entrenamiento si queremos alcanzar unos resultados parecidos. Una época ocurre cada vez que evaluamos el conjunto de entrenamiento entero con la función de pérdida. Utilizando 20 épocas para el entrenamiento de nuestros modelos no ocurren apenas mejoras, obteniendo un resultado equiparable al que conseguimos con las inicializaciones aleatorias de pesos, ya que en el algoritmo SHADE en cada generación tenemos que evaluar el modelo sobre el conjunto de entrenamiento un total de N_{pob} veces. Con los valores usados el algoritmo solo ejecutaría dos generaciones, una cifra insignificante en este aspecto.

Para que el entrenamiento pueda ser comparable y se siga un criterio claro a la hora de asignar recursos, vamos a redefinir el concepto de época para el entrenamiento con estos algoritmos. Siguiendo el criterio establecido en [Mar+21] y usando como referencia el algoritmo SHADE-ILS, vamos a establecer que una época realiza un total de 210 evaluaciones sobre el conjunto de entrenamiento. Esto surge de utilizar 200 evaluaciones para el algoritmo SHADE y 10 para el algoritmo de búsqueda local. Así, al entrenar 20 épocas, tendríamos un total de 4200 evaluaciones sobre el conjunto de entrenamiento, mientras que los optimizadores realizarían 20. En el caso de ejecutar SHADE, otorgamos esas 10 evaluaciones pertenecientes a la búsqueda local también al algoritmo poblacional.

En los algoritmos meméticos, la ejecución del GD se realiza cada dos épocas, es decir cada 420 evaluaciones. Por tanto aunque contamos las evaluaciones realizadas por el optimizador que corresponda, a efectos prácticos no restarían ejecuciones ni a la búsqueda local ni al algoritmo generacional ya que la comprobación de que se ha superado el número máximo de evaluaciones se realiza siempre al final de la generación.

Durante el entrenamiento con las técnicas MH se evalúan los modelos únicamente sobre el conjunto de entrenamiento, guardando un array con el mejor modelo hasta esa generación y su correspondiente error. Luego, se evalúa cada modelo del array sobre el conjunto de validación y se selecciona el que menor error tenga sobre él de cara a calcular el error de generalización y comparar con el resto de técnicas. De esta manera establecemos un criterio similar al que usamos para seleccionar el mejor modelo entrenado con un optimizador basado en GD.

En las técnicas meméticas, a la hora de entrenar una época con GD no usamos la política de ciclos de Leslie. En primer lugar porque no está diseñada para entrenamientos tan cortos y no sería efectiva. En segundo lugar el uso de tasas de aprendizaje que aumenten hasta valores altos tiene

un objetivo exploratorio del paisaje de la función de coste, elemento que ya tenemos gracias a la parte basada en poblaciones del algoritmo memético, por lo que nos interesa centrarnos más en la explotación de una buena región de dicho paisaje.

10.1.3. Entrenamiento

Usamos los cuatro optimizadores de primer orden basados en GD mencionados anteriormente con un doble objetivo. En primer lugar así tenemos resultados más diversos con los que comparar las técnicas MH, pudiendo observar si estos algoritmos mejoran a alguno o ninguno de los optimizadores propuestos. Como los cuatro optimizadores son ampliamente usados en la literatura, creemos que esta información es pertinente. Por otro lado, sabemos que el optimizador que mejores resultados consiga depende ampliamente de la tarea y del modelo, por tanto, al ejecutar las técnicas hibridadas con el GD podemos asignar a cada modelo el optimizador que mejor rendimiento haya ofrecido en la tarea.

En las estrategias MH elegimos usar SHADE y SHADE-ILS. El primero es uno de los algoritmos sin búsqueda local que mejores resultados ofrece en optimización de problemas continuos, mientras que el segundo es un referente en la optimización de problemas continuos a gran escala y especialmente en el entrenamiento de modelos. Además proponemos dos técnicas nuevas: SHADE-GD y SHADE-ILS-GD, que resultan de la hibridación de las anteriores con el GD.

Debido a la sensibilidad del entrenamiento a los parámetros iniciales, se han usado los mismos pesos iniciales para los entrenamientos con distintos optimizadores de un mismo modelo. De manera similar, se usa la misma población inicial de soluciones para un mismo modelo en cada entrenamiento con técnicas MH. Se ha usado la inicialización de pesos Glorot, al igual que en el paper de referencia. Como diferencia respecto a dicha publicación, no usamos validación cruzada por los excesivos recursos computacionales que supondría, con lo que dividimos los datos en entrenamiento-validación-test tal como se indica en la Sección 10.2.

Para las tareas de regresión se ha usado el error cuadrático medio como función de coste. Es ampliamente usada en la literatura y aunque es sensible a los valores extremos, como tenemos preprocesamiento de datos vemos reducido el efecto. Se ha usado también la métrica R^2 para medir la explicación de la varianza con respecto a la media como predicción, para tener un criterio objetivo de comparación ya que en las dos tareas de regresión la escala del objetivo es distinta. Para las tareas de clasificación se ha usado la entropía cruzada como función de error y *accuracy* como métrica, opciones ampliamente usadas en la literatura. En los conjuntos de datos tabulares se ha usado *Balanced Accuracy* en lugar de *accuracy*, ya que las clases no están balanceadas, y se conoce que en estos casos la segunda no es una métrica

		Parámetro	**Valor**
GD	**ADAM y ADAMW**	β_1	0.9
		β_2	0.999
	RMSPROP	α	0.99
	NAG	mom	0.9
MH		N_{pob}	10
		Max_evals	4200
		Gen_{SHADE}	20
		$Evals_{LS}$	10
		Reinicio	3
		% mejora	5

Tabla 3: Hiperparámetros utilizados en el entrenamiento según el optimizador. Hay dos parámetros comunes a todos los algoritmos: 20 épocas de ejecución e inicialización de pesos Glorot, aunque hay que aclarar que el concepto de época no tiene el mismo significado en el caso de entrenar con técnicas MH: en dicho caso una época corresponde a unas 210 evaluaciones del conjunto de entrenamiento, es decir 4200 evaluaciones en total, representadas en el hiperparámetro Max_evals. En las MH, Gen_{SHADE} y $Evals_{LS}$ corresponden al número de iteraciones por generación del algoritmo SHADE y de la búsqueda local (en caso de SHADE-ILS y SHADE-ILS-GD), respectivamente. Por ejemplo, SHADE-ILS realiza 20 épocas de 210 evaluaciones del conjunto de entrenamiento cada una, 200 correspondientes al algoritmo de SHADE y 10 al algoritmo de búsqueda local L-BFGS. Cada generación en SHADE realiza 10 evaluaciones del conjunto de entrenamiento. Los hiperparámetros usados en los optimizadores basados en GD son los estándar asociados a dichos algoritmos.

representativa, de hecho se hace especial mención a esto en [Mar+21]. En los conjuntos de datos de imágenes las clases están perfectamente balanceadas por lo que se usa *accuracy*, aunque en este caso su versión balanceada coincidiría con la normal.

Para la elección de hiperparámetros usamos los elegidos en [Mar+21], en los casos en que no podamos basarnos en el paper, usaremos los valores por defecto de PyTorch y los propuestos en los papers originales, en ese orden. Estos valores predeterminados de PyTorch, aunque no conseguirán el mejor rendimiento posible, están optimizados para funcionar bien en una variedad muy amplia de situaciones. Además el propósito es una comparación objetiva entre las técnicas de entrenamiento, no obtener el máximo rendimiento de cada una de ellas. Debido al gran número de modelos que entrenamos, no podríamos invertir el tiempo necesario para ajustar correctamente todos los hiperparámetros, en especial los referentes a los algoritmos MH. Podemos ver los valores usados en la Tabla 3.

10.1.4. Métricas de evaluación utilizadas

Para evaluar el desempeño de los modelos de aprendizaje profundo entrenados mediante los algoritmos propuestos, se emplean métricas específicas en función del tipo de tarea de aprendizaje (clasificación o regresión) y de las características de los conjuntos de datos (balance de clases).

Regresión. Para los conjuntos de datos asociados a tareas de regresión (BHP, WQR), se utiliza como métrica el coeficiente de determinación R^2, el cual cuantifica la proporción de la varianza de la variable dependiente que es explicada por el modelo. Esta métrica toma valores en el intervalo $(-\infty, 1]$, donde un valor de 1 indica un ajuste perfecto, y valores cercanos o menores a 0 indican que el modelo no explica mejor que una predicción constante (como la media de los datos). Su definición formal es la siguiente:

$$R^2 = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2}$$

donde y_i representa los valores reales, $\hat{y}_i$ las predicciones del modelo, $\bar{y}$ la media de los valores reales, y n el número total de muestras.

Clasificación con clases balanceadas. En tareas de clasificación donde las clases están balanceadas (MNIST, FMNIST, CIFAR-10G), se emplea como métrica la *accuracy* (precisión global), que representa la proporción de predicciones correctas sobre el total de muestras. Esta métrica se define como:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

donde TP, TN, FP y FN corresponden, respectivamente, a los verdaderos positivos, verdaderos negativos, falsos positivos y falsos negativos.

Clasificación con clases desbalanceadas. En los casos en que las clases presentan un desbalance significativo (BCW, WQC), la métrica de *accuracy* puede inducir a interpretaciones erróneas sobre el rendimiento del modelo, ya que una alta precisión puede alcanzarse simplemente favoreciendo la clase mayoritaria. Por ello, en estos casos se emplea la métrica de *Balanced Accuracy*, que consiste en el promedio del *recall* (sensibilidad) obtenido en cada clase. Para un problema de clasificación con C clases, su definición es:

$$\text{Balanced Accuracy} = \frac{1}{C}\sum_{i=1}^{C}\frac{TP_i}{TP_i + FN_i}$$

donde TP_i y FN_i corresponden a los verdaderos positivos y falsos negativos para la clase i. Esta métrica permite evaluar el rendimiento de forma

Conjunto de datos	BCW	BHP	WQ
Tarea	C	R	C y R
Nº instancias	569	506	4898
Nº características	30	13	11
Objetivo	Diagnosis	MEDV	Quality
Ratio balance	1.68	-	80.5
Faltan valores	No	Si	No
Complejidad	Media-baja	Media-baja	Media-alta

Tabla 4: Características de los conjuntos de datos tabulares utilizados en la experimentación. Se incluyen tres conjuntos: Breast Cancer Wisconsin (BCW), Boston Housing Price (BHP) y Wine Quality (WQ). La tarea asociada a cada conjunto puede ser de clasificación (C), regresión (R) o ambas. Se presentan el número de instancias y características, la variable objetivo (diagnóstico para BCW, precio medio de vivienda (MEDV) para BHP y calidad del vino para WQ), el ratio de balance (calculado como la razón entre la clase mayoritaria y la minoritaria en tareas de clasificación), la presencia de valores faltantes y una estimación de la complejidad del conjunto de datos.

equitativa entre clases, penalizando los modelos que ignoran clases minoritarias.

10.2. Conjuntos de datos

10.2.1. Tabulares

En la Tabla 4 podemos ver un resumen de estos conjuntos de datos. Vamos a describirlos un poco más en profundidad:

- BCW [23]: las características se calculan a partir de una imagen digitalizada de un aspirado con aguja fina de una masa mamaria. Describen las características de los núcleos celulares presentes en la imagen. Se extraen diez características como el radio, perímetro, área, etc. y de cada una de ellas se calcula la media, la desviación estándar y la peor, resultando en las 30 características finales. El objetivo a clasificar es binario, el diagnóstico puede resultar beningno o maligno.

- BHP[24]: se obtiene a partir de datos sobre el mercado de la vivienda en Boston. Sus características describen varios factores como los impuestos sobre cada vivienda o la tasa de criminalidad en el barrio. El

[23] https://www.kaggle.com/conjuntosdedatos/uciml/breast-cancer-wisconsin-data

[24] https://www.kaggle.com/conjuntosdedatos/altavish/boston-housing-conjuntodedatos

objetivo a predecir es MEDV (*Median Value*), es decir el valor mediano de las casas habitadas en escala de mil dólares.

- WQ [25]: describe varias características del vino en base a tests fisicoquímicos como la densidad, el pH o los sulfatos que contiene. Debemos predecir la calidad (1-10) mediante clasificación o regresión. La mayor complejidad reside en el poco balance entre las clases a predecir.

Para estos conjuntos de datos se ha realizado un preprocesado de los datos básico y con decisiones comunes basadas en la literatura. Se han eliminado las variables que tienen menos de un 5 o 10 % (dependiendo de la cantidad de variables del conjunto de datos) de correlación con la variable objetivo. Con las parejas de variables que tienen más de un 90 % de correlación entre sí se elimina una de las dos. Se han eliminado outliers con el método *zscore* (BHP, BCW) y el rengo intercuartílico (WQ) y se han escalado los datos de entrada a través de la normalización. El tamaño del *batch* se ha elegido mediante pruebas experimentales entre los valores 32, 64 y 128. Se divide el conjunto de datos en entrenamiento-validación-test, con un porcentaje 70-10-20.

10.2.2. Imágenes

Usamos como conjuntos de datos MNIST[26], F-MNIST[27] y CIFAR-10G[28]. Se reducen a 10 mil imágenes para el conjunto de entrenamiento, del cual se toman 3 mil para validación; y 5 mil para test. Nos aseguramos de que las clases sigan perfectamente balanceadas después de la reducción. Se usan las imágenes con una resolución de 32x32 y un solo canal de escala de grises, adaptando las imágenes a estas dimensiones cuando sea necesario. No se realiza preprocesamiento de datos ya que se entiende que la propia red a través de las convoluciones realiza las transformaciones necesarias. Estas elecciones se realizan, al igual que la elección del tamaño del *batch*, para establecer un marco común con el artículo de referencia. La decisión de tomar la partición de validación del conjunto de entrenamiento corresponde principalmente a reducir el tamaño del mismo debido a las limitaciones de memoria del hardware y a la necesidad de tener un conjunto de validación debido a que sólo realizamos una ejecución del entrenamiento.

Vamos a conocer un poco más estos conjuntos de datos. MNIST contiene imágenes de resolución 28×28 de dígitos manuscritos (0-9) como se muestra en la Figura 25a. Es un *benchmark* estandarizado para algoritmos de clasificación de imágenes sencillos.

[25] https://www.kaggle.com/conjuntosdedatos/yasserh/wine-quality-conjuntodedatos

[26] https://yann.lecun.com/exdb/mnist/

[27] https://www.kaggle.com/conjuntodedatoss/zalando-research/fashionmnist

[28] https://www.kaggle.com/c/cifar-10/

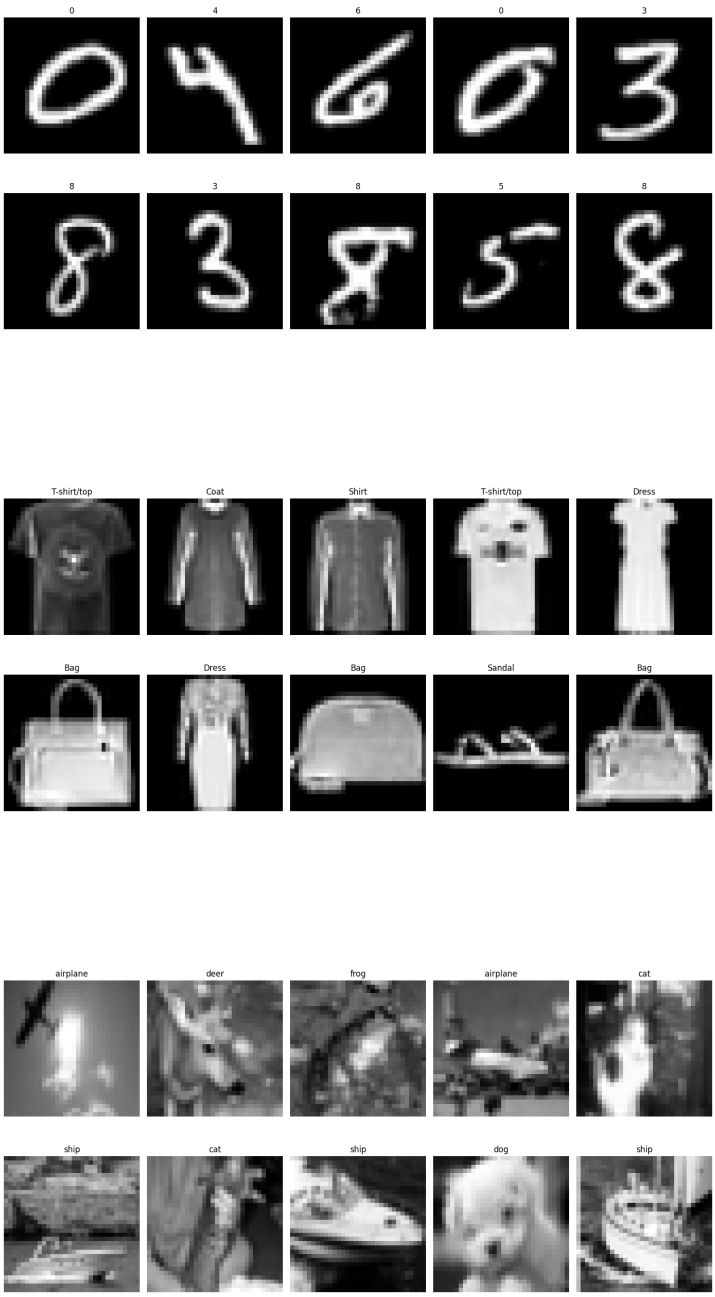

Figura 25: Ejemplos de imágenes de los conjuntos de datos utilizados en la experimentación para clasificación de imágenes. La primera fila (arriba) muestra muestras del conjunto MNIST, compuesto por dígitos escritos a mano en escala de grises (28×28 píxeles). La segunda fila (en medio) corresponde a Fashion-MNIST (F-MNIST), un conjunto de imágenes en escala de grises (28×28 píxeles) que representan distintos tipos de prendas de vestir y accesorios. La tercera fila (abajo) presenta ejemplos de CIFAR-10G, una versión escalada y en escala de grises (28×28 píxeles) del conjunto CIFAR-10G, que contiene imágenes de objetos como vehículos y animales. Sobre cada imagen se muestra su correspondiente etiqueta.

F-MNIST por su parte tiene imágenes en escala de grises de 10 tipos de ropa distintos (por ejemplo pantalones, camisetas, zapatos) como vemos en la Figura 25b, con la misma resolución que MNIST pero una complejidad media-baja. Esta diferencia se debe principalmente a la mayor variabilidad en los objetos de ropa y sus características. Las imágenes son más complejas y tienen patrones más intrincados que los dígitos.

Por último CIFAR-10G contiene imágenes en resolución $32 \times 32 \times 3$, aunque las convertimos a un solo canal en escala de grises como podemos ver en la Figura 25c, por lo que nos referimos a este conjunto de datos como CIFAR-10G. Incluye 10 clases de objetos como aviones, coches, pájaros, gatos, etc. La complejidad es alta debido a la naturaleza de las imágenes, que contienen una amplia variedad de objetos con diferentes formas, texturas y fondos. Las imágenes son relativamente pequeñas en resolución para la cantidad de información contenida en ellas, haciendo más difídil distinguir pequeños detalles necesarios para una correcta clasificación. La variabilidad en los datos requiere de técnicas más avanzadas para clasificación, haciéndolo un *benchmark* estandarizado para evaluar modelos de aprendizaje profundo. Reducir las imágenes a un solo canal ayuda a reducir la cantidad de parámetros del modelo y el tiempo de entrenamiento, pero para saber si afecta a la complejidad de la tarea habría que realizar un análisis específico, ya que aunque perdemos información sobre los datos no está claro que sea información relevante. Para ello, por ejemplo, los colores deberían ser consistentes dentro de una misma clase, y diferentes a los colores de las demás.

10.3. Modelos

Usaremos dos familias de modelos: MLP y ConvNets. Con los primeros usaremos conjuntos de datos tabulares para clasificación y regresión, y con los segundos conjuntos de datos de imágenes para la tarea de clasificación. La implementación de los MLP se ha realizado a través de la librería FastAI por simplicidad ya que ofrece lo necesario para usarlos directamente. La implementación de las ConvNets se ha realizado desde cero, observando la topología de LeNet5 y las ResNets en sus papers originales, ya que en ellas sí que se han introducido ciertos cambios que se comentan más adelante. Todos los modelos han sido entrenados desde cero.

Usaremos 4 modelos de tipo MLP, con 1,2,5 y 11 capas ocultas cada uno. El número de neuronas por capa es una potencia de 2 y con estructura piramidal incremental, es decir primero aumentando el número de neuronas por capa y luego disminuyéndolo. Estas son elecciones comunes en la literatura ya que facilitan las operaciones por su estructura (la primera) y el tratamiento de los datos (la segunda).

Antes de cada capa linal hay una capa BatchNorm1D, ya que es la implementación por defecto de FastAI y mejora el rendimiento en el entrenamiento. Los parámetros asociados a este tipo de capa van incluidos en el

Capas ocultas	Neuronas por capa	Parámetros
1	64	2238
2	64, 64	6462
5	64, 128, 256, 128, 64	85k
11	32, 64, 128, 256, 512, 1024, 512, 256, 128, 64 y 32	1.4M

Tabla 5: Características de los modelos MLP utilizados en la experimentación. Se detalla el número de capas ocultas, la cantidad de neuronas en cada capa oculta (en orden) y el número total de parámetros del modelo, incluyendo aquellos asociados a capas BatchNorm. Los modelos varían desde arquitecturas simples con una sola capa oculta de 64 neuronas y 2,238 parámetros, hasta arquitecturas profundas con 11 capas ocultas y 1.4 millones de parámetros, permitiendo evaluar el impacto de la profundidad en el desempeño de distintas estrategias de entrenamiento.

cómputo de la Tabla 5. En caso de que la tarea sea clasificación, la librería PyTorch incluye automáticamente una capa de *SoftMax* al final del modelo.

Para los modelos basados en convoluciones usamos LeNet5 y dos ResNets, con 15 y 57 capas. En el primero sustituimos las funciones de activación por ReLU, ya que en la literatura posterior a la presentación del modelo se han demostrado superiores a las sigmoides y la tangente hiperbólica. También se han sustituido las capas de *AveragePool* por *MaxPool* y añadido capas de BatchNorm por los mismos motivos. En la Tabla 6 se muestra la topología de este modelo, obviando las capas de *Flatten* y de *SoftMax*. Tiene un total de 62 mil parámetros.

Se han diseñado dos modelos de ResNet, uno con 15 capas y otro con 57. Los bloques convolucionales agrupan 3 capas de convolución con sus respectivas capas BatchNorm, y se usan convoluciones 1x1 para hacer cuello de botella, reduciendo así el número de parámetros sin perder expresividad de la red. Se sigue el diseño usual de esta familia de modelos, por ejemplo agrupando más bloques convolucionales en mitad de la red, con una convolución previa a los bloques convolucionales y usando solo una capa lineal. El modelo ResNet57 que se implementa tiene un total de 1.3M de parámetros, mientras que ResNet15 tiene 500 mil. Sus topologías pueden observarse en la Tabla 7 y la Tabla 8 respectivamente.

10.4. Experimentos

En esta sección se presentan cuatro experimentos diseñados para evaluar sistemáticamente el rendimiento, eficiencia y aplicabilidad de las MH en comparación con los métodos de GD para el entrenamiento de redes neuronales profundas. Los experimentos

Capa	**Dimensión**	**Kernel**	**Canales**
Convolución	28x28	5x5	6
BatchNorm2D	28x28	-	-
ReLU	28x28	-	-
Max Pool	14x14	2x2, stride 2	-
Convolución	10x10	5x5	16
BatchNorm2D	10x10	-	-
ReLU	10x10	-	-
Max Pooling	5x5	2x2	-
Lineal	120	-	-
BatchNorm1D	120	-	-
ReLU	120	-	-
Lineal	84	-	-
BatchNorm1D	84	-	-
ReLU	84	-	-
Lineal	num_classes	-	-

Tabla 6: Arquitectura del modelo LeNet5 utilizado en la experimentación para clasificación de imágenes de 28×28 píxeles con un solo canal de entrada. Se detallan las capas que componen la red, incluyendo convoluciones, capas BatchNorm, activaciones ReLU, *Max Pooling* y capas totalmente conectadas (Lineal). La columna "Dimensión" indica el tamaño espacial de la salida de cada capa, mientras que la columna "Canales" representa el número de mapas de características generados. En las capas lineales, la dimensión corresponde al número de neuronas.

abordan aspectos complementarios que, en conjunto, proporcionan una visión integral del comportamiento de ambos enfoques en diversos escenarios.

El primer experimento analiza si existen diferencias estadísticamente significativas en el rendimiento de redes MLP entrenadas con MH según el tipo de tarea (clasificación vs. regresión). Utilizando conjuntos de datos de complejidad equiparable y aplicando rigurosos tests estadísticos, se evalúa el rendimiento relativo de SHADE-ILS frente a AdamW en ambos tipos de tareas.

El segundo experimento profundiza en los factores que más influyen en la pérdida de rendimiento en tareas de clasificación, tanto en MLPs como en ConvNets. Mediante un análisis de dependencias parciales, se examina cómo la cantidad de ejemplos, la complejidad del conjunto de datos y el número de parámetros del modelo afectan al rendimiento predictivo de los modelos entrenados con ambos tipos de optimizadores (MH y GD).

El tercer experimento se centra en analizar los tiempos de eje-

Capa	**Dimensión**	**Kernel/Stride**	**Canales**
Convolución	26x26	7x7	64
BatchNorm2d	26x26	-	-
ReLU	26x26	-	-
MaxPool2d	13x13	2x2, stride 2, padding 1	-
BottleneckBlock x3	13x13	1x1, 3x3, 1x1	64
BottleneckBlock x4	7x7	1x1, 3x3, 1x1, stride 2	128
BottleneckBlock x4	4x4	1x1, 3x3, 1x1, stride 2	256
BottleneckBlock x3	2x2	1x1, 3x3, 1x1, stride 2	512
AdaptiveAvgPool2d	512	-	-
BatchNorm1d	512	-	-
Dropout	512	-	-
Lineal	num_classes	-	-

Tabla 7: Arquitectura del modelo ResNet57 utilizado en la experimentación para clasificación de imágenes de 28×28 píxeles con un solo canal de entrada. La tabla describe la secuencia de capas de la red, incluyendo convoluciones iniciales, capas BatchNorm, activaciones ReLU, *MaxPool*, bloques residuales tipo Bottleneck, *AdaptiveAvgPool*, regularización mediante *Dropout* y una capa lineal final. La columna "Dimensión" indica el tamaño espacial de la salida de cada capa, mientras que la columna "Canales" representa el número de mapas de características generados. En las capas lineales, la dimensión corresponde al número de neuronas.

cución durante el entrenamiento con MH en comparación con el GD. Este análisis desglosado permite comprender no solo las diferencias absolutas en tiempo, sino también los factores que más impactan en la eficiencia computacional de cada enfoque.

Finalmente, el cuarto experimento evalúa el rendimiento de propuestas híbridas propias, específicamente SHADE-GD y SHADE-ILS-GD, que combinan elementos de MH y GD. Este análisis comparativo determina si estos enfoques híbridos logran capitalizar las fortalezas de ambas técnicas y superar sus limitaciones individuales.

10.4.1. Experimento 1: Análisis de diferencias en el rendimiento de MLPs entrenados con MH según el tipo de tarea

En este experimento, planteamos la hipótesis de que **los modelos MLP entrenados con MH presentan diferencias significativas en su rendimiento según el tipo de tarea** (clasificación o regresión). En particular, investigamos si estas diferencias se deben a la naturaleza de la tarea, más que a factores como la complejidad del problema o la arquitectura de la red.

Capa	**Dimensión**	**Kernel/Stride**	**Canales**
Convolución	26x26	7x7	64
BatchNorm2d	26x26	-	-
ReLU	26x26	-	-
MaxPool2d	13x13	2x2, stride 2, padding 1	-
BottleneckBlock x1	13x13	1x1, 3x3, 1x1	64
BottleneckBlock x1	7x7	1x1, 3x3, 1x1, stride 2	128
BottleneckBlock x1	4x4	1x1, 3x3, 1x1, stride 2	256
BottleneckBlock x1	2x2	1x1, 3x3, 1x1, stride 2	512
AdaptiveAvgPool2d	512	-	-
BatchNorm1d	512	-	-
Dropout	512	-	-
Lineal	num_classes	-	-

Tabla 8: Arquitectura del modelo ResNet15 utilizado en la experimentación para clasificación de imágenes de 28x28 píxeles con un solo canal de entrada. Se detallan las capas de la red, incluyendo convoluciones, capas BatchNorm, activaciones ReLU, agrupamiento máximo (MaxPool), bloques tipo Bottleneck, agrupamiento adaptativo promedio (AdaptiveAvgPool), regularización mediante Dropout y una capa lineal final. La columna "Dimensiónïndica el tamaño espacial de la salida de cada capa, mientras que la columna Çanalesepresenta el número de mapas de características generados. En las capas lineales, la dimensión corresponde al número de neuronas.

Esta hipótesis **se fundamenta en evidencias experimentales**, como los resultados presentados en la Tabla 9, que sugieren un comportamiento diferenciado en el rendimiento de SHADE-ILS según el tipo de tarea, a diferencia de lo que ocurre con AdamW.

En los conjuntos de datos de clasificación (BCW y WQC), se observa que AdamW tiende a obtener un mejor desempeño en términos de métrica *Balanced Accuracy* en comparación con SHADE-ILS a medida que aumenta la profundidad de la red. Por ejemplo, en el conjunto WQC con 11 capas, la diferencia relativa (Dif_{rel}) es de -0.725, lo que indica una ventaja significativa a favor de AdamW. Por otro lado, en los conjuntos de regresión (BHP y WQR), el comportamiento es más errático y con diferencias relativas mucho más extremas. Destaca especialmente el caso de WQR con 11 capas, donde la Dif_{rel} alcanza valores negativos elevados (-597967.932), lo que sugiere que SHADE-ILS tiene dificultades para optimizar redes profundas en tareas de regresión.

Estos resultados nos llevan a plantear la hipótesis de que el rendimiento de los modelos entrenados con MH podría estar influenciado de manera significativa por el tipo de tarea. Mientras que para tareas de clasificación

Conjunto	Capas	ADAMW			SHADE-ILS			**Dif**$_{rel}$
		Train	Test	Metric	Train	Test	Metric	
BCW	1	0.190	0.108	0.960	0.073	0.142	0.970	**0.010**
	2	0.140	0.041	0.980	0.063	0.350	0.932	**-0.049**
	5	0.115	0.026	1.000	0.055	0.475	0.890	**-0.110**
	11	0.112	0.094	0.971	0.098	0.491	0.903	**-0.07**
WQC	1	0.970	2.172	0.436	1.098	1.235	0.201	**-0.539**
	2	0.873	3.177	0.500	1.078	1.340	0.167	**-0.666**
	5	0.770	3.833	0.606	0.963	1.405	0.167	**-0.724**
	11	0.940	3.873	0.607	1.137	1.509	0.167	**-0.725**
WQR	1	0.475	0.972	-0.012	0.611	0.970	-1.988	**-164.667**
	2	0.430	1.126	0.072	0.551	0.826	-2.491	**-35.597**
	5	0.390	1.180	0.236	0.589	0.724	-983.152	**-4166.898**
	11	0.377	2.162	0.133	0.601	0.734	-79529.602	**-597967.932**
BHP	1	84.429	6.332	0.790	10.048	7.549	0.576	**-0.271**
	2	55.283	4.577	0.832	9.560	4.648	0.746	**-0.103**
	5	94.640	3.511	0.847	13.594	14.555	-0.187	**-1.221**
	11	54.540	3.268	0.853	43.183	26.794	-97.404	**-115.190**

Tabla 9: Resultados del entrenamiento y evaluación de los modelos asociados al primer experimento. A partir de estos, intentamos averiguar, a través del test de los rangos con signo de Wilcoxon, si existe una diferencia estadísticamente significativa en el rendimiento de modelos entrenados con MH según el tipo de tarea. Los conjuntos de datos usados son BCW y WQC para clasificación y BHP y WQR para regresión. Estas tareas tienen una complejidad similar dos a dos (BCW-BHP y WQC-WQR). Los MLP, de entre 1 y 11 capas ocultas, son entrenados usando SHADE-ILS como técnica MH y ADAMW como optimizador basado en GD. Como métricas, se usa ***Balanced Accuracy*** en clasificación y $\boldsymbol{R}^2$ en regresión. La comparación, realizada de manera relativa observando el rendimiento de los modelos entrenados con MH en relación con su análogo entrenado con GD, no arroja resultados concluyentes.

SHADE-ILS muestra una degradación de rendimiento más gradual, en tareas de regresión se observan desviaciones mucho más marcadas, especialmente con arquitecturas más complejas.

Para realizar esta comparación, entrenaremos en cuatro conjuntos de datos tabulares: dos de clasificación (BCW y WQC) y dos de regresión (BHP y WQR), agrupados por complejidad (BCW-BHP y WQC-WQR), a través de los optimizadores MH SHADE y SHADE-ILS, y los basados en GD AdamW y RMSProp, ya que son los que arrojan mejores resultados. Utilizamos arquitecturas de MLP con 1, 2, 5 y 11 capas ocultas.

Como las métricas usadas para clasificación y regresión son distintas y no son comparables directamente, debemos establecer la comparación en términos relativos. Observamos que los modelos entrenados con GD no muestran diferencias en el rendimiento dependiendo de la tarea, y analizaremos la diferencia relativa de rendimiento entre MH y GD para cada tipo de tarea,

Test	Estadístico	P-valor
Shapiro-Wilk	0.635	**0.003**
Wilcoxon rangos con signo	6.000	**0.023**

Tabla 10: Resultado de los tests estadísticos llevados a cabo durante el primer experimento. Para medir objetivamente que no haya diferencia de rendimiento según el tipo de tarea para modelos de la familia MLP entrenados con MH, hemos medido el rendimiento relativo en comparación a un modelo análogo entrenado con GD. Aplicamos el test Shapiro-Wilk a la muestra resultante de las medidas relativas[29], con la hipótesis nula siendo que la muestra sigue una distribución normal. Al comprobar que no lo hace, y como se trata de muestras dependientes, aplicamos el test de Wilcoxon de los rangos con signos, con la hipótesis nula siendo que no hay diferencia de rendimiento entre clasificación y regresión en modelos MLP entrenados con MH. Obtenemos un p-valor que no nos da una evidencia suficiente para rechazar la hipótesis nula.

realizando el promedio entre los dos optimizadores utilizados de cada tipo (SHADE y SHADE-ILS; AdamW y RMSProp). Si las diferencias relativas son estadísticamente significativas, es decir, si la diferencia de rendimiento entre MH y GD para cada tipo de tarea es distinta, entonces concluiremos que los modelos entrenados con MH muestran diferente rendimiento dependiendo del tipo de tarea. Definimos la fórmula

$$Dif_{rel} = \frac{Metric_{MH} - Metric_{GD}}{|Metric_{GD}|}.$$

A partir de esta fórmula, obtenemos dos muestras: las diferencias relativas para clasificación y las diferencias relativas para regresión. El objetivo es determinar si existe una diferencia estadísticamente significativa entre ambas. Como son muestras dependientes, deberemos utilizar bien el t-test para muestras pareadas o bien el test de Wilcoxon de los rangos con signos. La elección de uno u otro dependerá de si las muestras siguen una distribución normal o no, utilizando t-test en el caso afirmativo y Wilcoxon en el contrario.

En primer lugar, verificamos la normalidad de las muestras mediante la prueba de Shapiro-Wilk, donde la hipótesis nula establece que las diferencias relativas siguen una distribución normal. Obtenemos un p-valor de 0.003, por debajo de los niveles de significacnia habituales (0.05, 0.01) lo que implica el rechazo de la hipótesis nula y confirma que las muestras no siguen una distribución normal.

En consecuencia, aplicamos la prueba no paramétrica de Wilcoxon. La hipótesis nula en este caso establece que no existen diferencias en el rendimiento relativo al GD entre las tareas de clasificación y regresión. **Obtene-**

mos un p-valor de 0.023, que es inferior al nivel de significancia habitual de 0.05, lo que nos permite rechazar la hipótesis nula. Esto sugiere que existen diferencias significativas en el rendimiento relativo al GD entre ambas tareas. Sin embargo, estos resultados parecen bastante sensibles a los optimizadores incluidos en el experimento, por lo que se deben tomar las conclusiones con cautela.

10.4.2. Experimento 2: Evaluación de los factores que más afectan a la pérdida de rendimiento en tareas de clasificación, tanto en MLPs como en ConvNets

La hipótesis que planteamos en este segundo experimento es que **la pérdida de rendimiento en modelos de clasificación, ya sean entrenados con MH o con GD, está fuertemente influenciada por la cantidad de ejemplos en el conjunto de datos, la complejidad del conjunto de datos y el número de parámetros del modelo**. Específicamente, esperamos que un menor número de ejemplos y un aumento en la complejidad del conjunto de datos o en el número de parámetros del modelo conduzcan a un deterioro significativo en el rendimiento predictivo.

Hemos seleccionado tres factores, comunes a cualquier tarea, que pueden influir notablemente: cantidad de ejemplos del conjunto de datos, complejidad del conjunto de datos y número de parámetros del modelo. Para esta tarea vamos a analizar los factores que acabamos de proponer a través de un análisis de dependencias parciales, con el fin de aislar la contribución de cada factor al rendimiento del modelo.

El análisis de dependencias parciales es una técnica utilizada en aprendizaje automático para interpretar modelos predictivos. Permite visualizar la relación entre una o más características de entrada y la salida del modelo, manteniendo fijas otras variables para aislar su efecto. Nosotros no lo usaremos sobre el modelo en sí, sino que tomaremos como características los factores que queremos evaluar, como desarrollaremos más adelante, y al aplicarlo obtendremos información sobre la importancia de cada factor en la salida del modelo. Si la gráfica es creciente indica un impacto positivo; si es decreciente el impacto es negativo, y si es no lineal indica relaciones complejas, como que esa variable dependa de otras. Cuanto mayor sea la pendiente de la gráfica, mayor es la relación de la variable con la salida.

La valoración de la métrica que usemos para medir el rendimiento será dependiente de la tarea. No es lo mismo obtener un 50 % de *accuracy* en CIFAR-10G, que al haber 10 clases no tiene por que ser un mal rendimiento, que obtener ese dato en BCW donde solo hay dos clases a clasificar, lo que indicaría un rendimiento parejo al de un clasificador aleatorio. Por tanto, vamos a quitar el sesgo del número de clases, y **como métrica de rendimiento vamos a usar el *accuracy* obtenido en relación al del clasificador aleatorio para esa tarea. Denominaremos a dicha métrica**

RA, con el valor de 0 simbolizando un rendimiento igual al del clasificador aleatorio. Un valor negativo indica que el rendimiento empeora con respecto al clasificador aleatorio, y un valor positivo indica que mejora. Un valor de uno equivaldría a un *accuracy* de 1. En WQC consideraremos solo 6 clases a predecir ya que es el número de clases que contienen ejemplos dentro del conjunto de datos.

$$RA = \frac{balanced\_accuracy - accuracy_{clasificador\_aleatorio}}{accuracy_{clasificador\_aleatorio}}.$$

Compararemos el rendimiento usando AdamW y RMSProp como optimizadores basados en GD, ya que son los que mejor resultado consiguen en las tareas correspondientes, con SHADE y SHADE-ILS como técnicas MH. Para obtener diferentes valores de los tres factores mencionados, usaremos 5 conjuntos de datos: BCW, WQC, MNIST, F-MNIST y CIFAR-10G. Entrenaremos los modelos LeNet5, ResNet15 y ResNet57 en tareas con imágenes y MLP capas 1,2,5,11 para tareas tabulares.

Para un correcto análisis vamos a asignar valores numéricos que representen a los factores que hemos mencionado. Englobamos un conjunto de datos, según el número de ejemplos, en tres valores: pocos (1), intermedio (2) y muchos (3). A MNIST, F-MNIST y CIFAR-10G les asignamos un 3 (15 mil ejemplos), a WQC un 2 (5 mil ejemplos) y a BCW un 1 (unos 500 ejemplos).

Realizamos lo mismo con el número de parámetros del modelo, agrupando de la siguiente forma con valores entre 1 y 4:

1. MLP1 (2238) y MLP2 (6462).
2. LeNet5 (60 mil) y MLP5 (85 mil).
3. ResNet15 (500 mil).
4. ResNet57 (1.3 millones) y MLP11 (1.4 millones).

Por último lo hacemos con la complejidad del conjunto de datos, agrupando en fácil (1), media (2) y alta (3). Entendemos la complejidad del conjunto de datos, y no de la tarea, como las características inherentes a los datos que afectan a la dificultad de modelar patrones, generalizar y realizar predicciones precisas. Vamos a justificar las elecciones en base a cada conjunto de datos, en orden creciente:

- BCW: le asignamos una complejidad fácil (1). Tiene solo 500 ejemplos y 30 características, las cuales ya están preprocesadas y son fáciles de interpretar (radio, textura, perímetro). La tarea de clasificación es binaria, haciéndola más sencilla que para varias clases. Además no hay ruido ni falta de datos significativos.

- MNIST[30]: complejidad fácil (1). Las imágenes son sencillas (28×28) con poca variabilidad. Aunque haya 10 clases, estas tienen características claramente distintas y bien separadas, facilitando la clasificación. Además las clases están perfectamente balanceadas. Es un *benchmark* estandarizado, sobre el que se obtienen resultados muy buenos con modelos muy sencillos.

- WQ: complejidad media (2). Cuenta con 5 mil ejemplos con 11 características, las cuales son variables continuas que no tienen una separación clara. Esta tarea tiene naturaleza de regresión, introduciendo variaciones sutiles en la predicción del objetivo. Está muy desbalanceado, habiendo clases donde no hay siquiera representación de los ejemplos, y otras donde solo hay unos pocos, haciendo difícil la generalización de los modelos.

- F-MNIST: complejidad media (2). Tiene una estructura similar a MNIST (tamaño 28×28, 10 clases) pero sus clases son visualmente más complejas. Tiene mucha más variabilidad dentro de cada clase como también similaridad entre clases, requiriendo refinar más el entrenamiento para este conjunto de datos.

- CIFAR-10G: complejidad alta (3). Aunque tenga la misma dimensionalidad que en los dos casos anteriores, sus clases son muy diversas entre sí con una variabilidad intra clase significativa. Son imágenes del mundo real, por lo que tienen fondo y aumenta la variabilidad. Además existe ruido y similitudes entre clases.

Podemos ver de forma esquemática en la Tabla 13 los valores de los factores asociados a los conjuntos de datos, mientras que en las tablas Tabla 11 y Tabla 12 podemos ver las representaciones finales que usaremos para realizar el análisis de dependencias.

[30] Recordamos que todos los conjuntos de datos de imágenes han sido reducidos a 15 mil ejemplos.

Conjunto	Modelo	Complejidad	Tamaño	Parámetros	SHADE	SHADE-ILS	RA
BCW	MLP1	1	1	1	0.931	0.970	**0.779**
BCW	MLP2	1	1	1	0.725	0.932	**0.779**
BCW	MLP5	1	1	2	0.825	0.890	**0.715**
BCW	MLP11	1	1	4	0.500	0.903	**0.403**
WQC	MLP1	2	2	1	0.295	0.201	**0.063**
WQC	MLP2	2	2	1	0.214	0.167	**0.063**
WQC	MLP5	2	2	2	0.042	0.167	**-0.076**
WQC	MLP11	2	2	4	0.000	0.167	**-0.100**
MNIST	LeNet5	1	2	2	0.171	0.066	**0.021**
MNIST	ResNet15	1	2	3	0.100	0.115	**0.008**
MNIST	ResNet57	1	2	4	0.082	0.100	**-0.010**
FMNIST	LeNet5	2	3	2	0.180	0.366	**0.192**
FMNIST	ResNet15	2	3	3	0.104	0.100	**0.002**
FMNIST	ResNet57	2	3	4	0.100	0.100	**0.000**
CIFAR	LeNet5	3	3	2	0.102	0.114	**0.009**
CIFAR	ResNet15	3	3	3	0.111	0.103	**0.007**
CIFAR	ResNet57	3	3	4	0.099	0.100	**-0.001**

Tabla 11: Resultados del entrenamiento para SHADE y SHADE-ILS en tareas de clasificación, considerando la complejidad del conjunto de datos, su tamaño, y el número de parámetros del modelo. Se usa la métrica *Balanced Accuracy*, mientras que la columna 'RA' representa la métrica de rendimiento promedio relativa al clasificador aleatorio. Los conjuntos de datos incluyen BCW, WQC, MNIST, F-MNIST y CIFAR-10-G, con valores numéricos asignados para caracterizar su complejidad (1: fácil, 2: media, 3: alta), tamaño (1: pocos ejemplos, 2: intermedio, 3: muchos) y cantidad de parámetros del modelo (1 a 4, donde 1 representa modelos más simples y 4 modelos más complejos), con el fin de realizar a posteriori un análisis de dependencias parciales. Un valor de RA negativo indica un rendimiento inferior al del clasificador aleatorio, mientras que valores positivos indican una mejora respecto a este.

Conjunto de datos	Complejidad	Cantidad de datos
BCW	1	1
MNIST	1	3
WQ	2	2
F-MNIST	2	3
CIFAR-10-G	3	3

Tabla 13: Valores atribuidos a dos de los factores a analizar en este experimento (complejidad del conjunto y cantidad de datos, falta el tamaño del modelo). El objetivo es obtener una representación numérica de los modelos cuantificando las intensidades de los factores para medir su influencia en el rendimiento a traves de un análisis de dependencias parciales. El valor de la complejidad se ha asignado atendiendo a los patrones y características intrínsecas a los datos. Para evaluar la cantidad nos fijamos en el número de ejemplos de dicho conjunto de datos. Las representaciones numéricas toman valores enteros entre 1 y 3, incluidos, siendo un 1 un valor bajo (poca cantidad de datos, baja complejidad) y un 3 alto.

Conjunto de datos	Modelo	Complejidad	Tamaño	Parámetros	ADAMW	RMSPROP	RA
BCW	MLP1	1	1	1	0.960	0.971	**0.950**
BCW	MLP2	1	1	1	0.980	0.990	**0.950**
BCW	MLP5	1	1	2	1.000	1.000	**1.000**
BCW	MLP11	1	1	4	0.971	0.981	**0.952**
WQC	MLP1	2	2	1	0.436	0.493	**0.347**
WQC	MLP2	2	2	1	0.500	0.394	**0.347**
WQC	MLP5	2	2	2	0.606	0.476	**0.449**
WQC	MLP11	2	2	4	0.607	0.451	**0.435**
MNIST	LeNet5	1	2	2	0.977	0.981	**0.977**
MNIST	ResNet15	1	2	3	0.962	0.967	**0.961**
MNIST	ResNet57	1	2	4	0.901	0.894	**0.886**
FMNIST	LeNet5	2	3	2	0.854	0.849	**0.835**
FMNIST	ResNet15	2	3	3	0.843	0.841	**0.824**
FMNIST	ResNet57	2	3	4	0.758	0.803	**0.756**
CIFAR	LeNet5	3	3	2	0.399	0.360	**0.311**
CIFAR	ResNet15	3	3	3	0.375	0.382	**0.309**
CIFAR	ResNet57	3	3	4	0.353	0.283	**0.242**

Tabla 12: Resultados del entrenamiento para AdamW y RMSProp en tareas de clasificación, considerando la complejidad del conjunto de datos, su tamaño, y el número de parámetros del modelo. Se usa la métrica *Balanced Accuracy*, mientras que la columna 'RA' representa la métrica de rendimiento promedio relativa al clasificador aleatorio. Los conjuntos de datos incluyen BCW, WQC, MNIST, F-MNIST y CIFAR-10G, con valores numéricos asignados para caracterizar su complejidad (1: fácil, 2: media, 3: alta), tamaño (1: pocos ejemplos, 2: intermedio, 3: muchos) y cantidad de parámetros del modelo (1 a 4, donde 1 representa modelos más simples y 4 modelos más complejos), con el fin de realizar a posteriori un análisis de dependencias parciales. Un valor de RA negativo indica un rendimiento inferior al del clasificador aleatorio, mientras que valores positivos indican una mejora respecto a este.

Llevamos a cabo el análisis de dependencias parciales usando un modelo *Random Forest* y con las herramientas proporcionadas por la líbreria `sklearn`, en concreto `ensemble` e `inspection`, que hace de éste es un proceso sencillo. Vemos los resultados obtenidos en la Figura 26 para proceder a analizarlos.

En el caso de las MH, vemos que **las gráficas correspondientes al tamaño del modelo y la complejidad del conjunto de datos son prácticamente planas**, por lo que concluimos que **estos dos factores no afectan al rendimiento de los modelos entrenados con técnicas MH**. En contraposición, y contrariamente a la hipótesis planteada, la pendiente correspondiente a **la cantidad de ejemplos** en el conjunto de datos es muy acusada y negativa, lo que quiere decir que **este factor influye de manera inversamente proporcional al rendimiento de los modelos. Es decir, cuantos más datos de entrenamiento, peor rinde el modelo.**

Con respecto al GD, tenemos que **el número de ejemplos y el tamaño del modelo afectan en mucha menor medida que la complejidad de la tarea**. Aunque estos efectos son suaves, podemos obser-

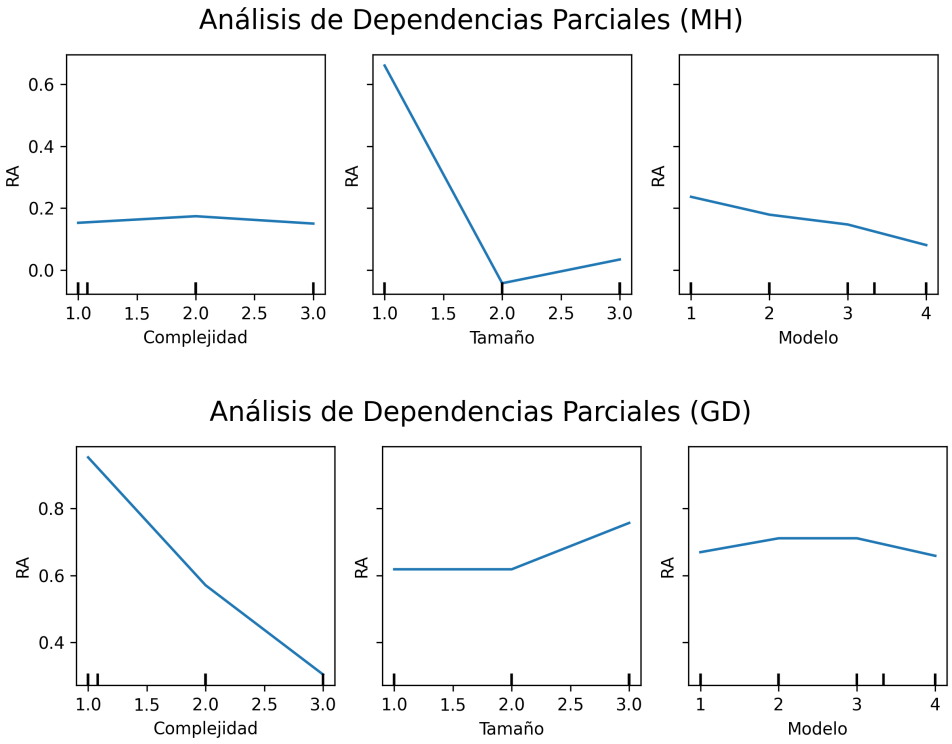

Figura 26: Análisis de dependencias parciales del rendimiento relativo al clasificador aleatorio (RA) en función de la complejidad del conjunto de datos, el tamaño del conjunto de datos y el número de parámetros del modelo, para modelos entrenados con AdamW y RMSProp (GD); y SHADE y SHADE-ILS (MH). La primera fila de gráficos muestra los resultados obtenidos con MH, mientras que la segunda corresponde a GD. En cada conjunto de gráficos, el primer panel representa la relación entre la complejidad de la tarea y el RA, el segundo panel muestra el efecto del tamaño del conjunto de datos sobre el RA, y el tercero analiza la influencia del número de parámetros del modelo en el RA. Una tendencia decreciente indica un impacto negativo en el rendimiento, mientras que una tendencia creciente sugiere una mejora en el desempeño del modelo. Se observa que GD es más sensible a la complejidad de la tarea, mientras que MH muestra una caída abrupta en rendimiento con conjuntos de datos más grandes.

var ciertas tendencias que corroboran lo que conocemos a nivel teórico y verifica la hipótesis planteada. Generalmente, cuantos más ejemplos de entrenamiento tengamos mejor rendimiento tendremos, ya que al tener una muestra mayor los modelos pueden aprender más patrones y más complejos. Por otro lado, sabemos que aumentar el número de parámetros en un modelo no siempre equivale a un mejor rendimiento, y si lo aumentamos sin un criterio concreto podemos caer en un sobreajuste que empeore el rendimiento. Estas ideas pueden observarse de manera leve.

Con la complejidad de la tarea sí que se observa una caída brusca de rendimiento conforme ésta crece. De nuevo corroboramos la literatura y, además, podemos cuantificar y comparar. **En resumen, para optimizadores basados en GD el incremento de la complejidad de la tarea afecta muy negativamente al rendimiento del modelo**, mientras que el número de parámetros y el número de ejemplos afectan de manera leve, con unas interacciones que deben ser analizadas más minuciosamente. **En el caso de las MH, es el incremento del número de ejemplos el que contribuye a empeorar rápidamente el rendimiento de los modelos**, mientras que la complejidad del conjunto de datos y el número de parámetros del modelo no afectan prácticamente.

10.4.3. Experimento 3: Análisis de los tiempos de ejecución en el entrenamiento con MH, tanto en MLPs como en ConvNets

La hipótesis que planteamos ahora es que **el tiempo de ejecución de los algoritmos de entrenamiento varía significativamente entre las MH y el GD, siendo las primeras generalmente más costosas computacionalmente**. Específicamente, esperamos que SHADE-ILS tenga un mayor tiempo de ejecución en comparación con AdamW, debido a la naturaleza exploratoria de las metaheurísticas en el espacio de soluciones. Esperamos también que esta diferencia de tiempo tenga como base una ejecución menos optimizada, traducida en mayor tiempo de ejecución por época.

Con la intención de comprender mejor los tiempos de ejecución de ambas estrategias, vamos a comparar los tiempos de los algoritmos SHADE-ILS y AdamW, ya que son dos métodos utilizados en los dos experimentos anteriores y por tanto con los que venimos tratando. Como SHADE forma parte de la ejecución del algoritmo SHADE-ILS, analizaremos también el tiempo del primero a partir del segundo. Las tareas que hemos medido, aprovechando experimentos anteriores, junto con el tiempo de ejecución (en segundos), el número de instancias del conjunto y el número de parámetros del modelo, se pueden observar en la Tabla 14. En base a estas dos variables mostramos gráficamente el tiempo consumido en la Figura 28.

Observando la Figura 27b, vemos que la diferencia de tiempos entre

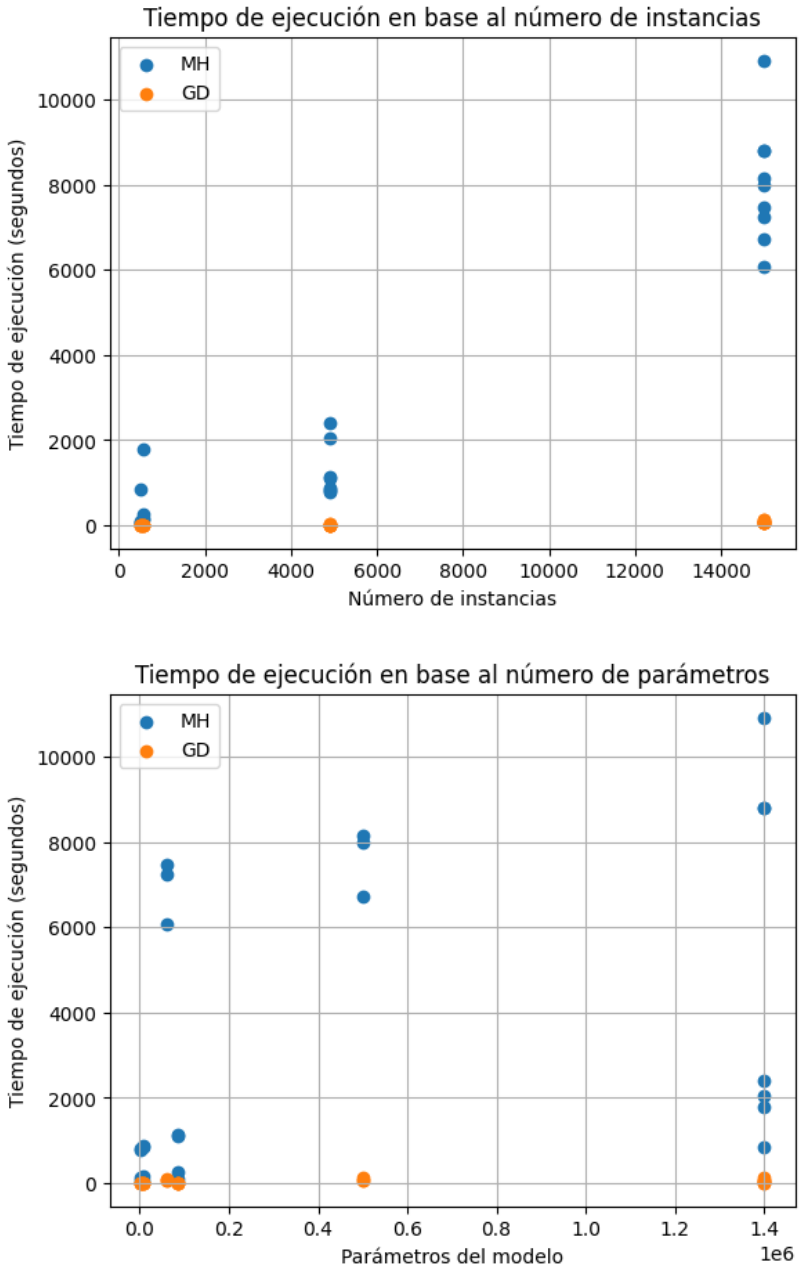

Figura 27: Tiempo de ejecución (en segundos) de SHADE-ILS (MH) y AdamW (GD) en el entrenamiento en función del número de instancias del conjunto de datos (arriba) y del número de parámetros del modelo (abajo). Se observa que SHADE-ILS (azul) presenta tiempos de ejecución considerablemente mayores en comparación con AdamW (naranja), especialmente a medida que aumenta el número de instancias y la cantidad de parámetros del modelo. En particular, el tiempo de ejecución de SHADE-ILS crece de manera pronunciada con ambos factores, especialmente el número de instancias del conjunto de datos.

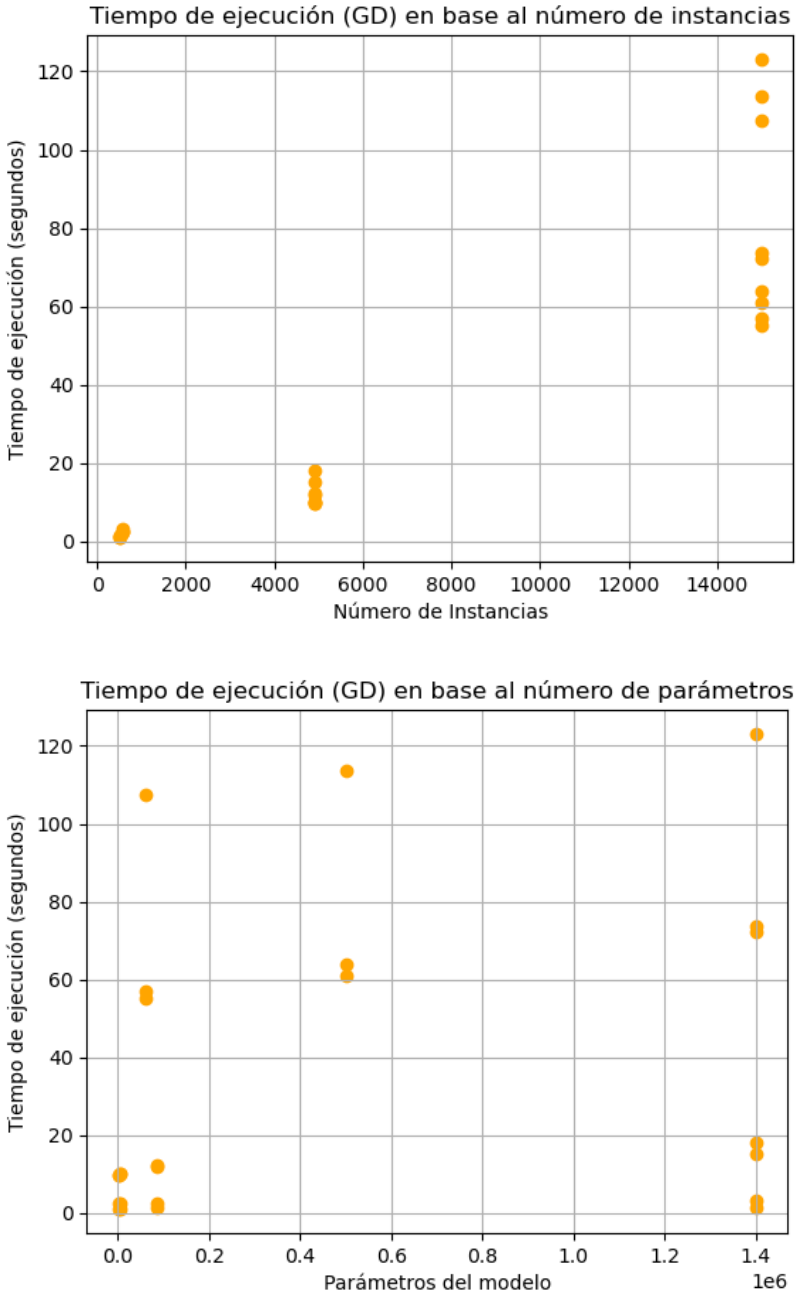

Figura 28: Tiempo de ejecución (en segundos) de AdamW (GD) en el entrenamiento en función del número de instancias del conjunto de datos (arriba) y del número de parámetros del modelo (abajo). Se observa que el número de instancias tiene un efecto mucho mayor que el número de parámetros del modelo.

Conjunto	Instancias	Modelo	Tamaño Modelo	SHADE-ILS	AdamW
BCW	569	MLP1	2238	**136.436**	**2.442**
BCW	569	MLP2	6462	**147.795**	**2.448**
BCW	569	MLP5	85000	**251.935**	**2.611**
BCW	569	MLP11	1400000	**1791.067**	**3.103**
BHP	506	MLP1	2238	**56.217**	**1.041**
BHP	506	MLP2	6462	**61.645**	**1.005**
BHP	506	MLP5	85000	**110.071**	**1.197**
BHP	506	MLP11	1400000	**833.789**	**1.385**
WQC	4898	MLP1	2238	**800.998**	**9.691**
WQC	4898	MLP2	6462	**867.628**	**10.065**
WQC	4898	MLP5	85000	**1111.359**	**12.283**
WQC	4898	MLP11	1400000	**2395.200**	**17.922**
WQR	4898	MLP1	2238	**785.936**	**9.677**
WQR	4898	MLP2	6462	**856.806**	**10.111**
WQR	4898	MLP5	85000	**1123.316**	**11.907**
WQR	4898	MLP11	1400000	**2054.672**	**15.203**
MNIST	15000	LeNet5	62000	**7254.432**	**54.999**
MNIST	15000	ResNet15	500000	**7991.074**	**61.035**
MNIST	15000	ResNet57	1400000	**8785.166**	**72.317**
FMNIST	15000	LeNet5	62000	**6068.308**	**56.922**
FMNIST	15000	ResNet15	500000	**6735.368**	**63.989**
FMNIST	15000	ResNet57	1400000	**10914.907**	**73.486**
CIFAR	15000	LeNet5	62000	**7453.761**	**107.512**
CIFAR	15000	ResNet15	500000	**8164.679**	**113.781**
CIFAR	15000	ResNet57	1400000	**8812.613**	**123.113**

Tabla 14: Tabla comparativa que muestra los tiempos de ejecución (en segundos) de los algoritmos SHADE-ILS y AdamW. Se muestra el conjunto de datos con el número de ejemplos correspondiente, además del modelo que se entrena con el número de parámetros que tiene. Se observa un incremento significativo en los tiempos de ejecución de SHADE-ILS con respecto a AdamW conforme aumenta la complejidad y el tamaño de los modelos y/o de los conjuntos de datos.

algoritmos aumenta en menor medida que en la Figura 27a. Además, en la Figura 27b existe una amplitud muy grande entre la ejecución más rápida y más lenta del algoritmo SHADE-ILS, variabilidad que viene dada por las ejecuciones con distinto número de instancias, mientras que en la Figura 27a los tiempos de las distintas ejecuciones son más compactos. Por tanto podemos afirmar que el número de instancias del conjunto de datos influye más en el tiempo de ejecución de las MH que el número de parámetros del modelo

Vemos clara la lentitud de la opción MH en comparación con la estrategia de GD, vamos pues a analizar de dónde surge esa diferencia y si es esperable a nivel teórico ya que asignamos más ejecuciones a los algoritmos MH. Elegiremos como referencia el caso de FMNIST con el modelo ResNet57. Mientras

que SHADE-ILS tarda 10914.907 segundos, AdamW consume 73.486, es decir 148 veces más. Tenemos que tener en cuenta que asignamos más recursos computacionales a la ejecución de SHADE-ILS, pero la diferencia no se corresponde con la teoría. AdamW realiza 20 evaluaciones del conjunto de entrenamiento, mientras que SHADE-ILS hace 4200, 210 veces más. Por lo tanto mejoramos el tiempo que se supone que tendría a nivel teórico.

Observando las figuras 28a y28b vemos que el número de ejemplos del conjunto de entrenamiento también tiene mayor peso en el tiempo de ejecución de AdamW, al igual que ocurría en el caso del algoritmo MH. El rango en el que se mueve el número de ejemplos (15 mil) es mucho menor del que se mueve el número de parámetros (1 millón aproximadamente), por lo que **este tiene un efecto proporcionalmente mucho mayor, del orden de 7 a 10 veces más.**

Vamos a desglosar los tiempos del algoritmo SHADE-ILS para comprenderlo mejor. La función de error propia que hemos implementado para las MH tarda 2.107 segundos en cada evaluación del conjunto de entrenamiento. Como SHADE-ILS ejecuta alrededor de 4200 evaluaciones de las cuales 4000 son realizadas con esa función, en total tenemos que consume 8428 segundos. Las otras 200 evaluaciones se llevan a cabo por el algoritmo de búsqueda local, que consume 4.158 segundos por eje por ejecución, lo que se traduce en un total de 831.600 segundos. Como a nivel abstracto ejecutamos 20 épocas (cada época son 200 evaluaciones de SHADE y 10 de búsqueda local), guardamos un histórico con 20 modelos a los que calculamos el error de validación y *accuracy*, en lo que consumimos 66.400 segundos, que hacen un total de 9326.000 segundos.

Esa diferencia de tiempo con los 10914.907 segundos totales son principalmente a las funciones de mutación y de cruce. Vamos a suponer que consumen el mismo tiempo y vamos a despreciar otras operaciones secundarias. Como SHADE ejecuta 4000 evaluaciones y tenemos 10 individuos, habrá 400 generaciones, lo que supone 8000 operaciones entre cruces y mutaciones. La diferencia entre el tiempo de la Tabla 14 y el que hemos calculado en el párrafo anterior es de 1588.907 segundos, que si los atribuimos a las operaciones de cruce y mutación tenemos que consumen 0.199 segundos por operación.

Concluimos por tanto, que **en términos de tiempo de ejecución por época las MH no son más lentas que el GD, de hecho lo mejoran. Sin embargo necesitan de muchas más ejecuciones para conseguir resultados que puedan asemejarse, en el mejor de los casos, a los obtenidos por optimizadores basados en GD**. Tenemos además un desglose de los tiempos que consumen las funciones que integran el algoritmo SHADE-ILS. La búsqueda local es la que más tiempo consume por ejecución, seguida de la operación de evaluar el conjunto de validación y obtener el valor de *accuracy*, aunque precisamente estas dos últimas operaciones son las que menos se ejecutan.

Como conclusión final del experimento, obtenemos dos afirmaciones:

1. El número de instancias del conjunto de datos con el que entrenamos influye más en el tiempo de entrenamiento que el número de parámetros del modelo, del orden de 7 a 10 veces más. Esto ocurre tanto para MH como para GD

2. En términos de tiempo de ejecución por época, las técnicas MH requieren menos tiempo de ejecución que los optimizadores basados en GD, resultado contrario a lo planteado en la hipótesis. Sin embargo, el hecho de que se necesiten muchas más épocas (del orden de 100 o 200 veces más) en las primeras para obtener resultados que se acerquen a las segundas, hace que el tiempo total de ejecución sea mucho mayor en las MH.

10.4.4. Experimento 4: Análisis comparativo de las propuestas propias

SHADE y SHADE-GD

Comentamos los resultados obtenidos del entrenamiento y que se muestran en la Tabla 15. SHADE-GD supera a SHADE en la mayoría de los escenarios, logrando menores errores en entrenamiento y prueba. La incorporación del descenso de gradiente mejora la optimización de los pesos, reduciendo el sobreajuste y favoreciendo la generalización. En contraste, SHADE muestra inconsistencias a medida que aumenta la complejidad del modelo, con errores significativamente mayores en entornos de alta dimensionalidad.

SHADE sufre mayor deterioro en la generalización, especialmente en modelos profundos y conjuntos de datos grandes como MNIST y CIFAR-10G, donde sus métricas de desempeño son muy bajas. SHADE-GD, en cambio, reduce la pérdida en entrenamiento sin aumentar drásticamente la pérdida en prueba, evitando estancamientos en soluciones subóptimas.

SHADE muestra un deterioro progresivo en arquitecturas con mayor número de parámetros, con un rendimiento deficiente en modelos MLP con 5 y 11 capas ocultas, así como en redes convolucionales complejas como ResNet15 y ResNet57. SHADE-GD mantiene un desempeño más estable y presenta mejoras en redes convolucionales, aunque persisten dificultades en modelos profundos.

En clasificación, SHADE-GD obtiene mejores métricas, especialmente en modelos simples como LeNet5, aunque sigue sin igualar métodos basados en gradientes en modelos más complejos. En regresión, SHADE muestra valores extremadamente negativos en R^2, mientras que SHADE-GD logra métricas más aceptables, destacando su ventaja en la minimización del error.

Conjunto	**Modelo**	**SHADE**			**SHADE-GD**		
		Train	Test	Métrica	Train	Test	Métrica
BCW	1	0.112	0.187	**0.931**	**0.108**	**0.181**	0.915
	2	0.215	0.493	0.725	**0.118**	**0.222**	**0.953**
	5	0.636	0.646	0.825	**0.106**	**0.452**	**0.879**
	11	0.701	0.693	0.500	**0.279**	**0.660**	**0.600**
WQC	1	2.100	2.475	**0.295**	**1.082**	**1.304**	0.171
	2	2.668	2.315	**0.214**	**1.025**	**1.486**	0.192
	5	2.303	2.304	0.042	**0.953**	**1.495**	**0.203**
	11	2.662	2.302	0.000	**1.104**	**3.982**	**0.211**
BHP	1	430.906	452.366	-1.922×10^4	**13.977**	**33.099**	**0.650**
	2	430.970	428.226	-9.607×10^2	**384.431**	**128.808**	**-4.610**
	5	429.719	435.000	-7.318×10^3	**396.344**	**324.056**	**-0.932**
	11	469.788	454.786	-5.381×10^6	**463.336**	**453.891**	**-1.294×10^5**
WQR	1	34.645	34.336	-4.150×10^2	**0.622**	**0.712**	**-2.624**
	2	35.425	30.956	-2.020×10^3	**0.562**	**1.548**	**-0.442**
	5	34.767	32.230	-2.135×10^5	**0.568**	**0.934**	**-3.447**
	11	**35.867**	32.176	-1.951×10^6	35.937	**32.039**	**-2.312×10^5**
MNIST	LeNet5	2.424	2.277	0.171	**0.110**	**1.930**	**0.529**
	ResNet15	2.811	2.302	0.100	**0.408**	2.301	0.100
	ResNet57	3.639	2.302	0.082	**2.730**	2.302	**0.120**
F-MNIST	LeNet5	1.498	2.291	0.180	**0.705**	**2.140**	**0.360**
	ResNet15	**2.668**	2.303	0.104	3.547	2.302	0.100
	ResNet57	2.525	2.303	0.100	**1.389**	2.303	0.100
CIFAR-10G	LeNet5	2.548	2.300	0.102	**1.483**	**2.275**	**0.142**
	ResNet15	**2.536**	2.303	**0.111**	2.798	2.303	0.096
	ResNet57	2.723	2.303	0.099	**2.701**	2.303	0.093

Tabla 15: Comparación del desempeño de SHADE y SHADE-GD en el entrenamiento de modelos de aprendizaje profundo sobre conjuntos de datos tabulares e imágenes. Se presentan los errores de entrenamiento (*Train*) y evaluación (*Test*), junto con la métrica de desempeño utilizada en cada caso (***Balanced Accuracy*** para BCW y WQC, $\boldsymbol{R}^2$ para BHP y WQR, y ***accuracy*** para MNIST, Fashion-MNIST y CIFAR-10 en escala de grises). Para datos tabulares, se emplean modelos MLP con distintas profundidades, mientras que para imágenes se evalúan modelos de la familia ConvNet (LeNet5, ResNet15 y ResNet57) (columna Modelo). Los valores resaltados indican el mejor desempeño en cada caso. En las filas sin valores resaltados, ambos métodos presentan un rendimiento equivalente. Consideramos que el modelo con mejor rendimiento es el que tiene mejor valor en su métrica. Se observa que SHADE-GD generalmente supera a SHADE, especialmente en modelos más profundos y en conjuntos de datos tabulares, donde SHADE tiende a deteriorarse.

SHADE-GD sigue sin ser competitivo en redes profundas aplicadas a conjuntos de datos complejos como CIFAR-10G, posiblemente debido a una exploración inicial ineficiente en SHADE. Además, SHADE tiende a estancarse en soluciones subóptimas cuando la dimensionalidad del espacio de búsqueda es alta, resultando en métricas de evaluación deficientes.

Como conclusión, SHADE-GD supera a SHADE en la mayoría de los casos, mostrando menor error y mejor desempeño, aunque aún enfrenta limitaciones en redes profundas y conjuntos de datos grandes. Se requieren estrategias adicionales de optimización para mejorar su eficacia en escenarios más complejos.

SHADE-ILS y SHADE-ILS-GD

Comentamos los resultados obtenidos del entrenamiento y que se muestran en la Tabla 16. El rendimiento de SHADE-ILS y SHADE-ILS-GD varía según la complejidad del problema y la arquitectura de la red utilizada. Aunque SHADE-ILS-GD puede reducir el error en algunos casos, no siempre mejora la capacidad de generalización del modelo. En redes profundas y conjuntos de datos de mayortamaño, **la combinación con GD no siempre aporta ventajas significativas sobre SHADE-ILS**.

SHADE-ILS muestra mayor estabilidad en modelos profundos, donde la introducción de GD en SHADE-ILS-GD a menudo induce fluctuaciones en la convergencia. En particular, en la tarea de clasificación sobre un conjunto de datos con alta dimensionalidad, SHADE-ILS-GD presentó variaciones bruscas en el valor de la métrica entre iteraciones, mientras que SHADE-ILS mantuvo una convergencia más uniforme. En tareas de regresión tabular, SHADE-ILS-GD mostró mejoras esporádicas en el coeficiente R^2, pero también una mayor propensión al sobreajuste cuando se aumentó la cantidad de capas en la red.

En problemas de regresión, SHADE-ILS-GD mejora la generalización en redes poco profundas, como en el caso de un modelo de tres capas donde redujo el error de evaluación en un 7 % (Tabla 16). Sin embargo, **en redes más complejas, esta ventaja desaparece**, e incluso el modelo entrenado con SHADE-ILS mantiene un desempeño más estable en el conjunto de entrenamiento. En clasificación, SHADE-ILS-GD no muestra mejoras consistentes y, en algunos casos, como en MNIST, obtuvo un valor de *accuracy* 3 % inferior a la de SHADE-ILS. En modelos convolucionales aplicados a conjuntos de datos de imágenes, SHADE-ILS-GD rara vez aporta mejoras y, en ocasiones, introduce sobreajuste, como se evidenció en una prueba donde el error de entrenamiento disminuyó sin que la métrica en el conjunto de prueba mejorara.

Como conclusión, SHADE-ILS es una opción más eficiente y

Conjunto	**Modelo**	**SHADE-ILS**			**SHADE-ILS-GD**		
		Train	Test	Métrica	Train	Test	Métrica
BCW	1	0.073	**0.142**	**0.970**	**0.062**	0.159	0.941
	2	0.063	0.350	0.932	0.063	0.350	0.932
	5	0.055	**0.475**	**0.891**	**0.031**	0.616	0.600
	11	**0.099**	**0.491**	**0.903**	0.112	0.704	0.500
WQC	1	1.098	1.235	0.201	**1.068**	**1.224**	**0.226**
	2	1.078	1.340	0.167	**1.065**	**1.400**	**0.306**
	5	0.963	1.405	0.167	**0.951**	**1.328**	0.166
	11	1.138	1.509	0.167	**1.099**	**1.496**	0.167
BHP	1	10.048	7.549	0.576	**8.928**	**3.895**	**0.849**
	2	9.561	4.649	0.747	9.561	4.649	0.747
	5	13.594	14.556	-0.187	**11.689**	**14.368**	**-0.078**
	11	43.184	26.794	-97.405	43.184	26.794	-97.405
WQR	1	0.611	0.970	-1.987	**0.599**	**0.912**	**-1.898**
	2	0.579	0.826	-2.491	**0.521**	**0.620**	**-4.105**
	5	0.589	**0.724**	-9.832×10^2	**0.579**	0.747	$\mathbf{-2.347 \times 10^2}$
	11	0.601	**0.734**	-7.953×10^4	0.601	0.774	-7.953×10^4
MNIST	LeNet5	2.530	2.300	0.066	2.530	2.300	0.066
	ResNet15	0.001	2.294	**0.115**	0.001	2.292	0.100
	ResNet57	2.308	2.303	0.100	**0.001**	2.303	0.100
F-MNIST	LeNet5	0.403	1.807	0.366	**0.327**	**1.646**	**0.462**
	ResNet15	3.547	2.302	**0.100**	**0.349**	2.302	0.063
	ResNet57	2.673	2.303	0.100	**1.208**	2.303	0.100
CIFAR-10G	LeNet5	2.568	2.301	0.114	2.568	2.301	0.114
	ResNet15	**1.697**	2.303	**0.103**	3.504	2.303	0.093
	ResNet57	2.579	2.303	**0.100**	**2.415**	2.303	0.096

Tabla 16: Comparación del desempeño de SHADE-ILS y SHADE-ILS-GD en el entrenamiento de modelos de aprendizaje profundo sobre conjuntos de datos tabulares e imágenes. Se presentan los errores de entrenamiento (*Train*) y evaluación (*Test*), junto con la métrica de desempeño utilizada en cada caso (*Balanced Accuracy* para BCW y WQC, R^2 para BHP y WQR, y *accuracy* para MNIST, Fashion-MNIST y CIFAR-10 en escala de grises). Para datos tabulares, se emplean modelos MLP con distintas profundidades, mientras que para imágenes se evalúan modelos de la familia ConvNet (LeNet5, ResNet15 y ResNet57) (columna Modelo). Los valores resaltados indican el mejor desempeño en cada caso. En las filas sin valores resaltados, ambos métodos presentan un rendimiento equivalente. Consideramos que el modelo con mejor rendimiento es el que tiene mejor valor en su métrica. No hay una diferencia clara entre ambos optimizadores, aunque se observa que SHADE-ILS en su versión original consigue más robustez, siendo más estable en la mayoría de tareas.

estable en redes profundas, mientras que SHADE-ILS-GD puede aportar mejoras en regresión con arquitecturas simples. Sin embargo, el refinamiento con GD no garantiza una mejor generalización y puede inducir sobreajuste o aumentar significativamente el tiempo de cómputo sin beneficios claros. En datos tabulares, SHADE-ILS-GD presenta mejoras ocasionales, pero en imágenes y redes profundas, SHADE-ILS sigue siendo la alternativa más confiable. Estos resultados sugieren que la incorporación de GD en SHADE-ILS debe evaluarse caso por caso, ya que sus beneficios no son universales y dependen del tipo de tarea y arquitectura utilizada.

SHADE-GD y SHADE-ILS-GD

En términos generales, SHADE-ILS-GD muestra un mejor rendimiento en los conjuntos de datos más complejos o con mayor variabilidad. En particular, destaca en el conjunto BHP, donde logra mejoras sustanciales en las métricas de evaluación. Por ejemplo, en el modelo MLP1, el error en el conjunto de prueba se reduce drásticamente de 33.099 con SHADE-GD a 3.895 con SHADE-ILS-GD, lo que se traduce en un incremento significativo en la métrica de evaluación (de 0.650 a 0.849). Este patrón se repite en otros modelos de BHP, donde SHADE-GD presenta errores elevados mientras que SHADE-ILS-GD los mantiene controlados, incluso revirtiendo resultados negativos.

En los conjuntos de datos de clasificación como BCW y WQC, el comportamiento de ambos algoritmos es más equilibrado, con ventajas alternadas en función del modelo específico. Por ejemplo, en BCW, SHADE-ILS-GD logra mejores resultados en los primeros modelos (e.g., modelo 1 con una métrica de 0.941 frente a 0.915), pero en modelos más complejos como MLP5 y el MLP11, su rendimiento disminuye notablemente en comparación con SHADE-GD. Por otro lado, en WQC, SHADE-ILS-GD tiene una ligera ventaja en la mayoría de los casos, con métricas más altas y errores de prueba más bajos en tres de los cuatro modelos.

En los conjuntos de imágenes como MNIST, F-MNIST y CIFAR-10G, SHADE-GD tiende a ofrecer un rendimiento más consistente en los modelos más simples, especialmente con arquitecturas como LeNet5. Por ejemplo, en MNIST con LeNet5, SHADE-GD alcanza una mejor métrica (0.529 frente a 0.066 de SHADE-ILS-GD) con menores errores en el conjunto de prueba. Sin embargo, en arquitecturas más profundas como ResNet57, ambos algoritmos muestran dificultades para lograr un desempeño óptimo, con métricas cercanas a 0.100 en múltiples instancias.

Un patrón recurrente es que SHADE-ILS-GD presenta un mejor ajuste en el conjunto de entrenamiento, lo que indica una mayor capacidad de minimizar el error durante la optimización. No obstante, **esta**

Conjunto	**Modelo**	**SHADE-GD**			**SHADE-ILS-GD**		
		Train	Test	Métrica	Train	Test	Métrica
BCW	1	0.108	0.181	0.915	**0.062**	**0.159**	**0.941**
	2	0.118	**0.222**	**0.953**	**0.063**	0.350	0.932
	5	0.106	**0.452**	**0.879**	**0.031**	0.616	0.600
	11	0.279	**0.660**	**0.600**	**0.112**	0.704	0.500
WQC	1	1.082	1.304	0.171	**1.068**	**1.224**	**0.226**
	2	**1.025**	1.486	0.192	1.065	**1.400**	**0.306**
	5	0.953	1.495	**0.203**	0.951	**1.328**	0.166
	11	1.104	3.982	**0.211**	**1.099**	**1.496**	0.167
BHP	1	13.977	33.099	0.650	**8.928**	**3.895**	**0.849**
	2	384.431	128.808	-4.610	**9.561**	**4.649**	**0.747**
	5	396.344	324.056	-0.932	**11.689**	**14.368**	**-0.078**
	11	463.336	453.891	-1.294×10^5	**43.184**	**26.794**	**-97.405**
WQR	1	0.622	**0.712**	-2.624	**0.599**	0.912	**-1.898**
	2	0.562	1.548	**-0.442**	**0.521**	**0.620**	-4.105
	5	**0.568**	0.934	**-3.447**	0.579	**0.747**	-2.347×10^2
	11	35.937	32.039	-2.312×10^5	**0.601**	**0.774**	$\mathbf{-7.953 \times 10^4}$
MNIST	LeNet5	**0.110**	**1.930**	**0.529**	2.530	2.300	0.066
	ResNet15	0.408	**2.301**	0.100	**0.001**	2.292	0.100
	ResNet57	2.730	2.302	**0.120**	**0.001**	2.303	0.100
F-MNIST	LeNet5	0.705	2.140	0.360	**0.327**	**1.646**	**0.462**
	ResNet15	3.547	2.302	**0.100**	**0.349**	2.302	0.063
	ResNet57	1.389	2.303	0.100	**1.208**	2.303	0.100
CIFAR-10G	LeNet5	**1.483**	**2.275**	**0.142**	2.568	2.301	0.114
	ResNet15	**2.798**	2.303	0.096	3.504	2.303	0.093
	ResNet57	2.701	2.303	0.093	**2.415**	2.303	0.096

Tabla 17: Comparación del desempeño de SHADE-ILS y SHADE-ILS-GD en el entrenamiento de modelos de aprendizaje profundo sobre conjuntos de datos tabulares e imágenes. Se presentan los errores de entrenamiento (*Train*) y evaluación (*Test*), junto con la métrica de desempeño utilizada en cada caso (*Balanced Accuracy* para BCW y WQC, R^2 para BHP y WQR, y *accuracy* para MNIST, Fashion-MNIST y CIFAR-10 en escala de grises). Para datos tabulares, se emplean modelos MLP con distintas profundidades, mientras que para imágenes se evalúan modelos de la familia ConvNet (LeNet5, ResNet15 y ResNet57) (columna Modelo). Los valores resaltados indican el mejor desempeño en cada caso. En las filas sin valores resaltados, ambos métodos presentan un rendimiento equivalente. Consideramos que el modelo con mejor rendimiento es el que tiene mejor valor en su métrica. Aunque no hay una diferencia clara entre los dos optimizadores, existe un patrón reconocible, en el que se evidencia que SHADE-GD tiende a generalizar mejor mientras que SHADE-ILS-GD minimiza más la función de pérdida.

ventaja no siempre se traduce en un mejor rendimiento en el conjunto de prueba, sugiriendo una posible tendencia al sobreajuste en ciertos modelos y conjuntos de datos. Por el contrario, **SHADE-GD ofrece una generalización más estable en tareas de clasificación simples**, aunque con dificultades evidentes en casos más complejos como BHP y ciertos modelos en WQR.

En resumen, SHADE-ILS-GD destaca por su capacidad de optimización en problemas complejos, particularmente en tareas de regresión con alto error, mientras que SHADE-GD ofrece un rendimiento más equilibrado y generalizable en tareas de clasificación. La elección entre ambos algoritmos depende en gran medida de la naturaleza del problema: SHADE-ILS-GD es más eficiente para reducir errores en entrenamientos difíciles, mientras que SHADE-GD es más fiable para obtener resultados consistentes en problemas de menor complejidad. En definitiva, **es un resultado parejo, aunque consideramos ganador a SHADE-GD** ya que generaliza mejor y consigue mejor métrica en 11 tareas frente a las 10 en las que lo hace SHADE-ILS-GD.

11. Conclusiones y trabajos futuros

En la parte informática de este TFG hemos investigado, desde una perspectiva eminentemente empírica, el uso de técnicas MH como alternativa al GD para el entrenamiento de modelos de aprendizaje profundo. Resulta crucial comprender cómo los enfoques clásicos se comparan con técnicas como las MH, que pueden ofrecer ventajas en términos de exploración del espacio de soluciones. Este estudio es relevante porque permite no solo evaluar el rendimiento de estos métodos en diferentes contextos, sino también aportar una visión sobre cómo pueden integrarse de manera más eficiente en la práctica de la inteligencia artificial. El código asociado al proyecto se puede encontrar en https://github.com/eedduu/TFG.

Al principio nos fijamos como objetivo principal "evaluar y analizar la eficacia de las técnicas MH para el entrenamiento de redes neuronales profundas en comparación con el algoritmo de GD, para tener una mejor comprensión de cuáles son las diferencias principales en el rendimiento de estas dos estrategias y cuáles son las causas de las mismas". Creemos que este objetivo se ha cumplido de manera satisfactoria, basándonos en el desglose de los objetivos parciales:

- En relación con el estudio de la literatura existente, se ha constatado el dominio absoluto de los métodos basados en GD en el ámbito del aprendizaje profundo, tanto en términos de rendimiento como de adopción. El análisis bibliométrico refleja esta hegemonía, con una diferencia de proporción de 7 a 1 en publicaciones frente a las MH. Sin embargo, también se ha evidenciado un creciente interés por las MH y enfoques híbridos, especialmente en áreas como la neuroevolución, AutoML y NAS, donde las limitaciones del GD abren la puerta a técnicas alternativas más flexibles. Estas estrategias, aunque aún menos eficientes en términos computacionales y de generalización, muestran potencial en tareas específicas y complejas. Esta revisión ha permitido contextualizar los objetivos experimentales y ha confirmado que la hibridación entre GD y MH no solo es prometedora, sino que se alinea con las tendencias más recientes en la literatura científica.

- Nos planteamos si el tipo de tarea (clasificación o regresión) afecta al rendimiento de las MH. Para ello utilizamos modelos de la familia MLP entrenados con MH, donde medimos su rendimiento de manera relativa en comparación al mismo modelo entrenado con un optimizador basado en GD. Al aplicar el test de los rangos con signo de Wilcoxon obtuvimos un p-valor de 0.023, confirmando que **existe una diferencia estadística significativa entre el rendimiento, relativo al GD, de las tareas de regresión y de clasificación.** Aun así, esta detección parece bastante sensible a los optimizadores incluidos, por lo que este resultado debe tomarse con cautela.

- Sabemos que cuando aumenta la dificultad de la tarea el rendimiento de los modelos decae, especialmente en los entrenados con MH. Averiguamos qué factores concretos influían más en ésto, seleccionando tres: la complejidad del conjunto de datos, el número de instancias y el número de parámetros del modelo. **A través de un análisis de dependencias parciales concluimos que el aumento del número de ejemplos en una tarea es la variable que más hace decrecer el rendimiento de los modelos entrenados con MH, mientras que en el caso de los optimizadores basados en GD la complejidad del conjunto de datos es el factor más influyente**. Se han encontrado fuertes evidencias que respaldan tanto las observaciones empíricas como los conocimientos teóricos.

- Con el objetivo de abordar la diferencia de tiempos en la ejecución de las dos estrategias, realizamos un análisis pormenorizado de los tiempos de ejecución de los algoritmos MH. Aunque necesitan de muchas más épocas de ejecución para alcanzar rendimientos aceptables, haciendo que se alargue más el entrenamiento, **comprobamos que el tiempo de ejecución por época es menor en el caso de las estrategias MH**. Además con este estudio hemos confirmado que **el tamaño del conjunto de datos afecta más al tiempo de ejecución que el número de parámetros del modelo a entrenar**, tanto en MH como en GD. Para nuestros experimentos, esta diferencia se daba en una proporción de entre 7 y 10 veces más.

- **Hemos propuesto dos algoritmos originales, SHADE-GD y SHADE-ILS-GD**, y los hemos comparado a sus respectivas versiones originales. **De un total de 25 tareas afrontadas, SHADE-GD ha destacado en 17 de ellas, mientras que SHADE solo mejora el rendimiento a nuestra propuesta en 4 de ellas**. Por otro lado, **aunque SHADE-ILS-GD mejora el rendimiento respecto a su versión original en algunas tareas concretas de baja dificultad, llegando a obtener un 14 % más de *accuracy* en algún caso, no se posiciona como una mejora consistente**. En modelos más profundos es menos estable y obtiene peores resultados. No obstante, cabe destacar que una de nuestras propuestas mejora a la versión original. **Cuando hemos comparado las dos propuestas entre sí, hemos obtenido un resultado parejo, en el que no hay un ganador claro, aunque de manera general se observa que SHADE-GD generaliza mejor**.

Ha sido necesario asentar y poner en práctica los conocimientos adquiridos a lo largo de todo el grado, como la capacidad de resolver problemas, así como el análisis y planificación de los mismos. Muchos de los contenidos de las asignaturas de Aprendizaje Automático, Visión por Computador y

Metaheurísticas han tenido que ser revisados minuciosamente. Al comienzo del proyecto, ha sido necesario entender de manera profunda cómo funciona el proceso de entrenamiento del algoritmo de aprendizaje del GD y los optimizadores basados en éste. También se ha requerido una investigación para analizar cómo se relacionan las técnicas MH con problemas de optimización continua a gran escala, como es el entrenamiento de modelos de aprendizaje profundo. Además, fue necesario investigar sobre la literatura reciente para tomar decisiones en cuanto a la experimentación y el entorno de trabajo.

Destacamos que, a parte de corroborar de manera empírica algunos aspectos ya conocidos en la literatura en lo relativo a la superioridad de los optimizadores basados en GD sobre las MH, hemos podido obtener una conclusión que, hasta donde sabemos, es nueva en la literatura:

- **El rendimiento de los modelos entrenados con MH se ve más afectado por el tamaño del conjunto de datos que por la complejidad del mismo o por el número de parámetros del modelo**. En nuestros experimentos, las MH tienden a obtener peores resultados en conjuntos de datos más grandes, posiblemente debido a su limitada capacidad para extraer patrones complejos cuando el número de iteraciones se mantiene constante.

En resumen, los hallazgos obtenidos confirman una inclinación a favor de los optimizadores basados en GD frente a las MH para el entrenamiento de redes profundas, destacando su superior eficiencia y solidez. No obstante, este estudio proporciona una comprensión más detallada de las debilidades y posibilidades de las MH, y propone un marco analítico valioso para investigaciones futuras orientadas a optimizar su desempeño o identificar contextos específicos donde puedan resultar competitivas.

Después de analizar los resultados de este TFG, exponemos algunos posibles trabajos futuros:

1. Un posible trabajo futuro es explorar estrategias híbridas entre MH y GD que consideren la estructura jerárquica de las redes neuronales profundas. Una idea interesante sería aplicar diferentes técnicas de optimización según el nivel de abstracción de las capas, por ejemplo, evaluando si las MH pueden ser más efectivas en las capas profundas, donde la optimización es más compleja, mientras que el GD podría aprovecharse en capas superficiales para ajustes más finos. Esto permitiría investigar si una asignación diferenciada de estrategias contribuye a mejorar la eficiencia y la convergencia del entrenamiento.

2. Otro enfoque a explorar es el uso de MH no solo para entrenar redes neuronales, sino para generar automáticamente su arquitectura. Esto incluiría definir aspectos como el número de capas, la cantidad de neuronas por capa y otros hiperparámetros estructurales mediante un

proceso de optimización. Comparar este enfoque con técnicas existentes, como NAS, permitiría evaluar hasta qué punto las MH pueden ser una alternativa viable y qué ventajas o desafíos presentan en términos de exploración del espacio de arquitecturas y costos computacionales.

Bibliografía

[Ahm+11] Amir Ali Ahmadi et al. "NP-hardness of deciding convexity of quartic polynomials and related problems". *Mathematical Programming* 137.1–2 (2011), págs. 453-476.

[AM06] Enrique Alba y Rafael Martí. *Metaheuristic procedures for training neural networks*. 1.ª ed. Springer Science, 2006.

[AML12] Yaser S. Abu-Mostafa, Malik Magdon-Ismail y Hsuan-Tien Lin. *Learning From Data*. AMLBook, 2012.

[Axl24] Sheldon Axler. *Linear algebra done right*. 4.ª ed. Springer Nature, 2024.

[Ayu+16] Vina Ayumi et al. "Optimization of Convolutional Neural Network using Microcanonical Annealing Algorithm". *Proceedings of the International Conference on Advanced Computer Science and Information Systems (ICACSIS)*. 2016, págs. 95-100.

[Bak+18] Bowen Baker et al. "Accelerating neural architecture search using performance prediction". *International Conference on Learning Representations (ICLR)*. 2018.

[Ban19] Anan Banharnsakun. "Towards improving the convolutional neural networks for deep learning using the distributed artificial bee colony method". *International Journal of Machine Learning and Cybernetics* 10.6 (2019), págs. 1301-1311.

[Bay+17] Atilim Gunes Baydin et al. "Automatic differentiation in machine learning: a survey". *Journal of Machine Learning Research* 18.1 (2017), págs. 5595-5637.

[BB23] Christopher Michael Bishop y Hugh Bishop. *Deep Learning - Foundations and Concepts*. 1.ª ed. Springer, 2023.

[Bel+17] Irwan Bello et al. "Neural optimizer search with reinforcement learning". *International Conference on Machine Learning (ICML)*. 2017, págs. 459-468.

[Ber+21] David Bertoin et al. "Numerical influence of ReLU'(0) on backpropagation". *Neural Information Processing Systems (NeurIPS)*. 2021, págs. 468-479.

[Bis06] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. 1.ª ed. Springer, 2006.

[BLH21] Yoshua Bengio, Yann LeCun y Geoffrey E. Hinton. "Deep learning for AI". *Association for Computing Machinery (ACM)* 64.7 (2021), págs. 58-65.

[BM17] Mohammad reza Bonyadi y Zbigniew Michalewicz. "Particle Swarm Optimization for Single Objective Continuous Space Problems: A Review". *Evolutionary Computation* 25.1 (2017), págs. 1-54.

[Bub15] Sébastien Bubeck. "Convex Optimization: Algorithms and Complexity". *Foundations and Trends in Machine Learning* 8.3-4 (2015), págs. 231-357.

[BV04] Stephen Boyd y Lieven Vandenberghe. *Convex Optimization*. Cambridge University Press, 2004.

[Byr+95] Richard H. Byrd et al. "A Limited Memory Algorithm for Bound Constrained Optimization". *Society for Industrial and Applied Mathematics (SIAM) Journal on Scientific Computing* 16.5 (1995), págs. 1190-1208.

[Cau09] Augustin-Louis Cauchy. "Analyse mathématique – Méthodc générale pour la résolution des systèmes d'équations simultanées". *Oeuvres complètes: Series 1*. 1.ª ed. Cambridge University Press, 2009, págs. 57-61.

[Cur44] Haskell B. Curry. "The method of steepest descent for non-linear minimization problems". *Quarterly of Applied Mathematics* 2.3 (1944), págs. 258-261.

[Dau+14] Yann N Dauphin et al. "Identifying and attacking the saddle point problem in high-dimensional non-convex optimization". *Neural Information Processing Systems (NeurIPS)*. 2014, págs. 2933-2941.

[Dea+12] Jeffrey Dean et al. "Large scale distributed deep networks". *Neural Information Processing Systems (NeurIPS)*. 2012, págs. 1223-1231.

[DG14] Omid E David e Iddo Greental. "Genetic algorithms for evolving deep neural networks". *Genetic and Evolutionary Computation Conference (GECCO)*. 2014, págs. 1451-1452.

[DHS11] John Duchi, Elad Hazan y Yoram Singer. "Adaptive subgradient methods for online learning and stochastic optimization". *Journal of Machine Learning Research* 12.7 (2011), págs. 2121-2159.

[DRO15] Christopher De Sa, Christopher Re y Kunle Olukotun. "Global convergence of stochastic gradient descent for some non-convex matrix problems". *International Conference on Machine Learning (ICML)*. 2015, págs. 2332-2341.

[EMH19] Thomas Elsken, Jan Hendrik Metzen y Frank Hutter. "Neural architecture search: A survey". *Journal of Machine Learning Research* 20.55 (2019), págs. 1-21.

[ES96] John E. y Robert B. Schnabel. *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*. 1.ª ed. Society for Industrial y Applied Mathematics on Scientific Computing, 1996.

[Fei24] Li Fei-Fei. *CS231n: Deep Learning for Computer Vision*. 2024.

[FGJ20] Benjamin Fehrman, Benjamin Gess y Arnulf Jentzen. "Convergence rates for the stochastic gradient descent method for non-convex objective functions". *Journal of Machine Learning Research* 21.136 (2020), págs. 1-48.

[Gao+20] Yang Gao et al. "Graph neural architecture search". *International Joint Conference on Artificial Intelligence (IJCAI)*. 2020, págs. 1403-1409.

[GB10] Xavier Glorot y Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks". *International Conference on Artificial Intelligence and Statistics (AISTATS)*. 2010, págs. 249-256.

[GBC16] Ian J. Goodfellow, Yoshua Bengio y Aaron Courville. *Deep Learning*. 2.ª ed. MIT Press, 2016.

[GP10] Michel Gendreau y Jean-Yves Potvin. *Handbook of metaheuristics*. 2.ª ed. Springer, 2010.

[He+15] Kaiming He et al. "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification". *International Conference on Computer Vision (ICCV)*. 2015, págs. 1026-1034.

[He+16] Kaiming He et al. "Deep residual learning for image recognition". *International Conference on Computer Vision (ICCV)*. 2016, págs. 770-778.

[Hin12] Geoffrey Hinton. *Lecture 6e rmsprop: Divide the gradient by a running average of its recent magnitude*. https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf. Neural Networks for Machine Learning, Coursera. 2012.

[HKV19] Frank Hutter, Lars Kotthoff y Joaquin Vanschoren. *Automated machine learning: methods, systems, challenges*. 1.ª ed. Springer Nature, 2019.

[Hoc+01] S. Hochreiter et al. "Gradient flow in recurrent nets: the difficulty of learning long-term dependencies". *A Field Guide to Dynamical Recurrent Neural Networks*. Wiley-IEEE Press, 2001, págs. 237-243.

[IS15] Sergey Ioffe y Christian Szegedy. "Batch normalization: accelerating deep network training by reducing internal covariate shift". *International Conference on Machine Learning (ICML)*. 2015, págs. 448-456.

[Kar05] Dervis Karaboga. *An idea based on honey bee swarm for numerical optimization*. Inf. téc. Erciyes university, engineering faculty, computer engineering department, 2005, págs. 1-10.

[KB14] Diederik P. Kingma y Jimmy Ba. "Adam: A Method for Stochastic Optimization". 2014.

[KGV83] Scott Kirkpatrick, C. Gelatt y M. Vecchi. "Optimization by Simulated Annealing". *Science* 220.4598 (1983), págs. 671-680.

[Kha+17] Mujahid Khalifa et al. "Particle swarm optimization for deep learning of convolution neural network". *Sudan Conference on Computer Science and Information Technology (SCCSIT)*. 2017, págs. 1-5.

[KM23] Mehrdad Kaveh y Mohammad Saadi Mesgari. "Application of meta-heuristic algorithms for training neural networks and deep learning architectures: A comprehensive review". *Neural Processing Letters* 55.4 (2023), págs. 4519-4622.

[KSH17] Alex Krizhevsky, Ilya Sutskever y Geoffrey E Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". *Neural Information Processing Systems (NeurIPS)*. 2017, págs. 84-90.

[Kyr20] Anastasios Kyrillidis. *Algorithms, Optimization, and Learning*. https://akyrillidis.github.io/2020/03/05/algo.html. 2020.

[Law06] Gregory F. Lawler. *Introduction to Stochastic Processes*. 2nd. Chapman Hall/CRC, 2006.

[LBH15] Yann LeCun, Yoshua Bengio y Geoffrey Hinton. "Deep learning". *Nature* 521.7553 (2015), pág. 436.

[LCY14] Min Lin, Qiang Chen y Shuicheng Yan. "Network In Network". 2014.

[LeC+12] Yann LeCun et al. "Efficient BackProp". *Neural Networks: Tricks of the Trade*. 2.ª ed. Springer, 2012, págs. 9-48.

[LH19] Ilya Loshchilov y Frank Hutter. "Decoupled Weight Decay Regularization". *International Conference on Learning Representations (ICLR)*. 2019.

[Li+18a] Hao Li et al. "Visualizing the loss landscape of neural nets". *Neural Information Processing Systems (NeurIPS)*. 2018, págs. 6391-6401.

[Li+18b] Lisha Li et al. "Hyperband: A novel bandit-based approach to hyperparameter optimization". *Journal of Machine Learning Research* 18.185 (2018), págs. 1-52.

[LN89] Dong C. Liu y Jorge Nocedal. "On the limited memory BFGS method for large scale optimization". *Mathematical Programming* 45.1 (1989), págs. 503-528.

[LSY18] Hanxiao Liu, Karen Simonyan y Yiming Yang. "Darts: Differentiable architecture search". *arXiv* (2018).

[Lu+19] Zhichao Lu et al. "NSGA-Net: neural architecture search using multi-objective genetic algorithm". *Genetic and Evolutionary Computation Conference (GECCO)*. 2019, págs. 419-427.

[Mar+21] Aritz D. Martinez et al. "Lights and Shadows in Evolutionary Deep Learning: Taxonomy, Critical Methodological Analysis, Cases of Study, Learned Lessons, Recommendations and Challenges". *Information Fusion* 67 (2021), págs. 161-194.

[Mer+20] Panayotis Mertikopoulos et al. "On the almost sure convergence of stochastic gradient descent in non-convex problems". *Neural Information Processing Systems (NeurIPS)*. 2020, págs. 1117-1128.

[Mii+19] Risto Miikkulainen et al. "Evolving deep neural networks". *Artificial intelligence in the age of neural networks and brain computing*. 1.ª ed. Elsevier, 2019, págs. 269-287.

[Mit97] Tom M. Mitchell. *Machine learning*. 1.ª ed. McGraw-hill New York, 1997.

[MK87] Katta G. Murty y Santosh N. Kabadi. "Some NP-complete problems in quadratic and nonlinear programming". *Mathematical Programming* 39.2 (1987), págs. 117-129.

[MLH18] Daniel Molina, Alberto LaTorre y Francisco Herrera. "SHADE with Iterative Local Search for Large-Scale Global Optimization". *Congress on Evolutionary Computation (CEC)*. 2018, págs. 1-8.

[MP43] W. S McCulloch y W. Pitts. "A logical calculus of the ideas immanent in nervous activity". *The Bulletin of Mathematical Biophysics* 5 (1943), págs. 115-133.

[Mur22] Kevin P. Murphy. *Probabilistic Machine Learning: An introduction*. 1.ª ed. MIT Press, 2022.

[Nes83] Yurii Nesterov. "A method for solving the convex programming problem with convergence rate $O(1/k^2)$". *Proceedings of the USSR Academy of Sciences* 269.3 (1983), págs. 543-547.

[Nøk16] Arild Nøkland. "Direct Feedback Alignment Provides Learning in Deep Neural Networks". *Neural Information Processing Systems (NeurIPS)*. 2016, págs. 1045-1053.

[Nov17] K. Novak. *Numerical Methods for Scientific Computing*. 1.ª ed. Lulu.com, 2017.

[NW06] Jorge Nocedal y Stephen J. Wright. *Numerical Optimization*. 2.ª ed. Springer, 2006.

[Pav+05] NG Pavlidis et al. "Spiking neural network training using evolutionary algorithms". *International Joint Conference on Neural Networks (IJCNN)*. 2005, págs. 2190-2194.

[Pha+18] Hieu Pham et al. "Efficient neural architecture search via parameters sharing". *International Conference on Machine Learning (ICML)*. 2018, págs. 4095-4104.

[PKN18] Krzysztof Pawełczyk, Michal Kawulok y Jakub Nalepa. "Genetically trained deep neural networks". *Genetic and Evolutionary Computation Conference (GECCO)*. 2018, págs. 63-64.

[Pri23] Simon J.D. Prince. *Understanding Deep Learning*. 1.ª ed. MIT Press, 2023.

[PSL05] K.V. Price, R.N. Storn y J.A. Lampinen. *Differential Evolution: A Practical Approach to Global Optimization*. 2.ª ed. Springer, 2005.

[Qia99] Ning Qian. "On the momentum term in gradient descent learning algorithms". *Neural Networks* 12.1 (1999), págs. 145-151.

[Rea+19] Esteban Real et al. "Regularized evolution for image classifier architecture search". *Proceedings of the aaai conference on artificial intelligence*. 2019, págs. 4780-4789.

[RFA15] L.M. Rere, Mohamad Ivan Fanany y Aniati Arymurthy. "Simulated Annealing Algorithm for Deep Learning". *Procedia Computer Science* 72 (2015), págs. 137-144.

[RHW86] David E Rumelhart, Geoffrey E Hinton y Ronald J Williams. "Learning representations by back-propagating errors". *Nature* 323.6088 (1986), págs. 533-536.

[Ros58] F. Rosenblatt. "The perceptron: A probabilistic model for information storage and organization in the brain". *Psychological Review* 65.6 (1958), págs. 386-408.

[RSS12] Alexander Rakhlin, Ohad Shamir y Karthik Sridharan. "Making gradient descent optimal for strongly convex stochastic optimization". *International Conference on Machine Learning (ICML)*. 2012, pág. 15711578.

[Rus16] P. Russell S. J. Norvig. *Artificial intelligence: a modern approach*. Pearson, 2016.

[SB08] Adam Slowik y Michal Bialko. "Training of artificial neural networks using differential evolution algorithm". *Conference on Human System Interaction (HSI)*. 2008, págs. 60-65.

[SBF16] Bobak Shahriari, Alexandre Bouchard-Cote y Nando Freitas. "Unbounded Bayesian Optimization via Regularization". *International Conference on Artificial Intelligence and Statistics (AISTATS)*. 2016, págs. 1168-1176.

[Sch15] Jürgen Schmidhuber. "Deep learning in neural networks: An overview". *Neural Networks* 61 (2015), págs. 85-117.

[Sej18] Terrence J. Sejnowski. *The Deep Learning Revolution*. 1.ª ed. MIT Press, 2018.

[SM02] Kenneth O. Stanley y Risto Miikkulainen. "Evolving Neural Networks Through Augmenting Topologies". *Evolutionary Computation* 10.2 (2002), págs. 99-127.

[Som22] Ian Sommerville. *Software Engineering*. 10.ª ed. Pearson, 2022.

[SP97] Rainer Storn y Kenneth V. Price. "Differential Evolution - A Simple and Efficient Heuristic for global Optimization over Continuous Spaces". *Journal of Global Optimization* 11.4 (1997), págs. 341-359.

[Sta+19] Kenneth O Stanley et al. "Designing neural networks through neuroevolution". *Nature Machine Intelligence* 1.1 (2019), págs. 24-35.

[Suc+17] Felipe Petroski Such et al. "Deep neuroevolution: Genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning". *arXiv* (2017).

[SW65] S. S. Shapiro y M. B. Wilk. "An analysis of variance test for normality (complete samples)". *Biometrika* 52.3-4 (1965), págs. 591-611.

[Tal09] El-Ghazali Talbi. *Metaheuristics: From Design to Implementation*. 2009.

[TF13] Ryoji Tanabe y Alex Fukunaga. "Success-history based parameter adaptation for Differential Evolution". *Congress on Evolutionary Computation (CEC)*. 2013, págs. 71-78.

[Tib13] Ryan Tibshirani. *Convex Optimization: Lecture 6 - Gradient Descent*. Lecture notes, Carnegie Mellon University. 2013.

[Wil45] Frank. Wilcoxon. "Individual Comparisons by Ranking Methods". *Biometrics* 1 (1945), págs. 196-202.

[WM97] D.H. Wolpert y W.G. Macready. "No free lunch theorems for optimization". *Congress on Evolutionary Computation (CEC)*. 1997, págs. 67-82.

[WSB90] D Whitley, T Starkweather y C Bogart. "Genetic algorithms and neural networks: optimizing connections and connectivity". *Parallel Computing* 14.3 (1990), págs. 347-361.

[Xin99] Yao Xin. "Evolving artificial neural networks". *Proceedings of the IEEE* 87.9 (1999), págs. 1423-1447.

[XY17] Lingxi Xie y Alan Yuille. "Genetic CNN". *International Conference on Computer Vision (ICCV)*. 2017, págs. 1379-1388.

[Zha+18] Huizhen Zhao et al. "A novel softplus linear unit for deep convolutional neural networks". *Applied Intelligence* 48.7 (2018), págs. 1707-1720.

[Zha+23] Aston Zhang et al. *Dive into Deep Learning*. https://d2l.ai/. 2023.

[ZM19] Jian Zhang y Ioannis Mitliagkas. "YellowFin and the Art of Momentum Tuning". *Proceedings of Machine Learning and Systems (MLSys)*. 2019, págs. 289-308.

[Zop+18] Barret Zoph et al. "Learning transferable architectures for scalable image recognition". *Conference on Computer Vision and Pattern Recognition (CVPR)*. 2018, págs. 8697-8710.