Grado en Ciencia e Ingeniería de Datos

Escuela de Ingeniería Informática

Motor de Redes Neuronales

Kimberly Casimiro Torres
Claudia Álvarez González

16 de enero de 2026

Índice general

Índice de figuras

Índice de cuadros

Índice de algoritmos

Resumen

Este proyecto consiste en el desarrollo de un motor de redes neuronales implementado en Python, utilizando exclusivamente la librería NumPy como soporte numérico. Su propósito ha sido construir un sistema capaz de crear, entrenar y evaluar redes neuronales sin depender de frameworks de deep learning, garantizando un control total sobre cada fase del proceso y una comprensión profunda de su funcionamiento interno.

La estructura del proyecto se ha diseñado de forma modular, priorizando claridad, mantenibilidad y extensibilidad. El núcleo del sistema se organiza en módulos para capas y activaciones, funciones de pérdida, optimizadores, utilidades de preprocesamiento y métricas, y un módulo central que coordina la ejecución de la red y el entrenamiento mediante un *Trainer* con soporte para mini-batches, regularización y estrategias de ajuste del learning rate.

El motor ha sido validado experimentalmente en dos escenarios representativos: el dataset Iris, donde se alcanzó una precisión del 100 %, y MNIST, donde se obtuvo una precisión de prueba del 97.7 %. Estos experimentos incluyen el seguimiento de curvas de entrenamiento, matrices de confusión y análisis del comportamiento del proceso de aprendizaje.

Como mejora principal incorporada en la segunda entrega, se amplió la validación automática mediante un conjunto de pruebas unitarias que verifica explícitamente el funcionamiento de los métodos implementados: propagación hacia adelante y hacia atrás en capas y activaciones, cálculo de gradientes en pérdidas, actualización de parámetros en optimizadores (SGD+Momentum, RMSProp y Adam), regularización mediante Dropout, estrategias de aprendizaje Step Decay y Cosine Annealing, ejecución completa del flujo forward/backward en la red secuencial, entrenamiento gestionado por el *Trainer* (incluyendo Early Stopping), y utilidades de particionado y generación de mini-batches. Además, se mantiene la verificación numérica de gradientes mediante GradCheck, garantizando coherencia entre gradientes analíticos y numéricos.

En conjunto, el proyecto proporciona una implementación funcional, verificable y extensible de un motor de redes neuronales desde cero, capaz de reproducir el comportamiento esencial de modelos reales en tareas de clasificación.

1. Introducción

En la actualidad, la inteligencia artificial (IA) se ha convertido en un elemento clave dentro del desarrollo tecnológico y científico, influyendo de forma directa en la toma de decisiones, la automatización de procesos y la generación de conocimiento a partir de grandes volúmenes de datos. Entre sus múltiples ramas, el aprendizaje profundo (Deep Learning) destaca por su capacidad para extraer patrones complejos y representar la información de forma jerárquica, permitiendo resolver tareas de clasificación, predicción o reconocimiento con una precisión sin precedentes. No obstante, el uso extensivo de librerías y frameworks de alto nivel ha provocado que, en muchos casos, el funcionamiento interno de estas técnicas quede oculto para el estudiante o desarrollador, dificultando la comprensión profunda de su lógica matemática y computacional.

En este contexto surge este proyecto, cuyo propósito es implementar desde cero un motor de redes neuronales capaz de reproducir los mecanismos fundamentales del aprendizaje supervisado sin recurrir a librerías externas especializadas. Este enfoque pretende reforzar el entendimiento conceptual de los procesos que intervienen en la propagación hacia adelante, la retropropagación del error y la optimización de los parámetros, mostrando de manera explícita cómo una red neuronal aprende a partir de los datos.

El proyecto aborda la construcción modular de una red neuronal que incluye capas densas, funciones de activación, algoritmos de optimización, y funciones de pérdida tanto para clasificación como para regresión. Además, se integran componentes esenciales como el control del sobreajuste mediante dropout y early stopping, y la verificación numérica de gradientes, garantizando la coherencia entre los cálculos analíticos y numéricos.

Finalmente, la validación del motor se llevó a cabo utilizando dos conjuntos de datos ampliamente conocidos: Iris, que permite comprobar la correcta implementación de los cálculos fundamentales, y MNIST, que representa un escenario más complejo basado en imágenes de dígitos manuscritos. A través de ambos casos se analiza el comportamiento del modelo, su capacidad de generalización y la estabilidad del proceso de aprendizaje, ofreciendo así una visión completa del funcionamiento interno de una red neuronal desarrollada íntegramente desde la base.

2. Conjuntos de Datos

Para la validación del motor de redes neuronales se emplearon dos conjuntos de datos de referencia ampliamente utilizados en el ámbito del aprendizaje automático: **Iris** y **MNIST**. Ambos fueron seleccionados por su naturaleza complementaria, permitiendo evaluar el comportamiento del modelo tanto en un entorno numérico de baja dimensionalidad como en uno visual y de mayor complejidad.

Dataset Iris

El conjunto de datos **Iris**, introducido por Ronald Fisher en 1936, constituye un clásico en la clasificación supervisada. Contiene un total de 150 muestras correspondientes a tres especies de flores: *Iris setosa*, *Iris versicolor* e *Iris virginica*. Cada muestra se describe mediante cuatro características numéricas: longitud y anchura del sépalo, y longitud y anchura del pétalo, todas medidas en centímetros.

Atributo	Tipo	Descripción
Sepal length	Numérico	Longitud del sépalo (cm)
Sepal width	Numérico	Anchura del sépalo (cm)
Petal length	Numérico	Longitud del pétalo (cm)
Petal width	Numérico	Anchura del pétalo (cm)

Cuadro 2.1: Estructura del dataset Iris

En la Tabla 2.1 se resumen las cuatro variables numéricas que describen cada muestra del dataset Iris. Las variables se normalizaron al rango $[0, 1]$ y las etiquetas fueron codificadas en formato *one-hot*. El conjunto se dividió en subconjuntos de entrenamiento, validación y prueba con proporciones de 70 %, 15 % y 15 %, respectivamente.

Con el objetivo de complementar la descripción anterior, la Figura 2.1 presenta imágenes representativas de las tres especies (*Iris setosa*, *Iris versicolor* e *Iris virginica*). En ellas se aprecian diferencias morfológicas, especialmente en el tamaño y la forma de los pétalos, lo que ayuda a contextualizar la separabilidad entre clases en este problema de clasificación.

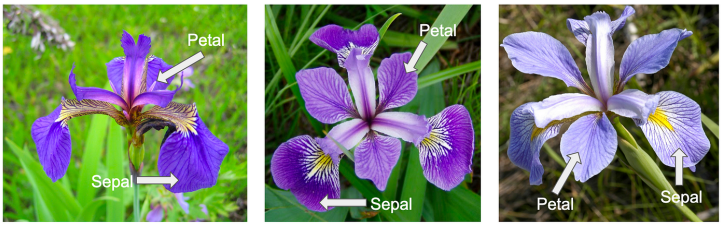

Figura 2.1: Ejemplos visuales de las tres especies del dataset Iris: *setosa*, *versicolor* y *virginica*.

Dataset MNIST

El conjunto de datos **MNIST** (Modified National Institute of Standards and Technology) es uno de los más conocidos para el reconocimiento de dígitos manuscritos. Contiene un total de **70 000 imágenes** en escala de grises de tamaño 28×28 píxeles, distribuidas en diez clases correspondientes a los dígitos del 0 al 9. De este total, 60 000 imágenes se emplean para entrenamiento y 10 000 para la evaluación final.

Cada imagen está acompañada de su etiqueta correspondiente, codificada en formato *one-hot*. Antes del entrenamiento, los valores de los píxeles se normalizan al rango $[0, 1]$ dividiendo por 255, y cada imagen se reestructura como un vector de 784 componentes.

Atributo	**Tipo**	**Descripción**
Imagen	Matriz numérica 28×28	Píxeles en escala de grises (0–255)
Etiqueta	Categórico	Dígito correspondiente (0–9)

Cuadro 2.2: Estructura del dataset MNIST

En la Tabla 2.2 se resumen los dos elementos fundamentales del conjunto: la imagen (matriz de píxeles) y su etiqueta asociada (clase entre 0 y 9). Este esquema permite justificar el preprocesado aplicado (normalización y vectorización) y la naturaleza multiclase del problema.

Para ilustrar visualmente el contenido del dataset, la Figura 2.2 muestra una muestra aleatoria de dígitos manuscritos. En ella se aprecia la variabilidad en forma y trazo dentro de una misma clase, lo que hace de MNIST un caso adecuado para evaluar la estabilidad del entrenamiento y la capacidad de generalización del modelo.

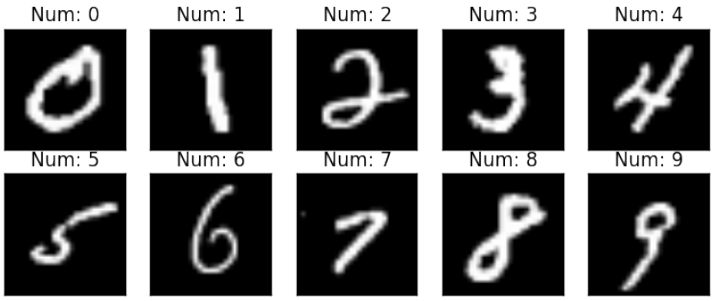

Figura 2.2: Ejemplos de dígitos manuscritos del dataset MNIST.

En conjunto, los datasets Iris y MNIST permiten comprobar la correcta implementación del motor y su desempeño en tareas de distinta naturaleza, ofreciendo una validación completa del proceso de aprendizaje.

3. Métodos

Se describen en detalle los métodos empleados en el desarrollo del motor de redes neuronales. Se incluyen las técnicas de inicialización de pesos, las capas principales, las funciones de activación y pérdida, los algoritmos de optimización y las estrategias de entrenamiento y regularización. Cada apartado presenta las ecuaciones fundamentales, una descripción paso a paso del procedimiento y una explicación conceptual del funcionamiento interno, conectando la teoría con las decisiones de implementación adoptadas en el código del proyecto.

3.1. Inicialización de Pesos

Una correcta inicialización de los pesos es esencial para la estabilidad y la velocidad de convergencia del entrenamiento. Si los pesos iniciales son demasiado pequeños, los gradientes pueden desvanecerse al propagarse hacia atrás; si son demasiado grandes, las activaciones pueden explotar y saturar las funciones no lineales. En este proyecto se implementaron dos estrategias clásicas y contrastadas: Xavier (Glorot) y He. Xavier asume activaciones aproximadamente lineales o simétricas, y distribuye los pesos para mantener la varianza entre capas; He está especialmente diseñada para ReLU y variantes, compensando el truncado de activaciones negativas con una mayor varianza inicial. En la práctica, seleccionar He cuando se usan ReLU y Xavier cuando se usan activaciones saturables reduce ajustes manuales de la tasa de aprendizaje y mejora la estabilidad del descenso del gradiente.

Inicialización Xavier

La inicialización Xavier busca mantener la varianza de las activaciones y gradientes constante a lo largo de capas profundas, asumiendo entradas independientes y una activación aproximadamente lineal alrededor del cero. En el proyecto se emplea la versión uniforme, que distribuye los pesos en un intervalo simétrico dependiente de los tamaños de entrada y salida. Esta elección reduce el riesgo de que las señales se amplifiquen o atenúen de manera sistemática al avanzar en la red, disminuyendo la saturación en funciones como tanh o sigmoid en redes moderadamente profundas.

$$W_{ij} \sim \mathcal{U}\Big(-\sqrt{\tfrac{6}{n_{\text{in}}+n_{\text{out}}}},\ \sqrt{\tfrac{6}{n_{\text{in}}+n_{\text{out}}}}\Big)$$

Inicialización He

La inicialización de He fue propuesta para activaciones ReLU, donde la mitad de las activaciones se clipean a cero y, por tanto, la varianza efectiva cambia frente al caso lineal. Ajustar la desviación estándar a $\sqrt{2/n_{\text{in}}}$ compensa esa pérdida, manteniendo la magnitud de la señal a través de las capas. En nuestro código, esta inicialización se implementa con distribución normal, y resulta especialmente útil para acelerar las primeras épocas del entrenamiento y minimizar estados con neuronas muertas por falta de activación.

$$W_{ij} \sim \mathcal{N}\Big(0,\ \sqrt{\tfrac{2}{n_{\text{in}}}}\Big)$$

3.2. Capa Densa

La capa densa realiza una transformación afín sobre un mini-batch de ejemplos, lo que constituye el bloque básico de redes de perceptrones multicapa. Dada una entrada $\mathbf{X} \in \mathbb{R}^{N\times d}$, unos pesos $\mathbf{W} \in \mathbb{R}^{d\times k}$ y un sesgo $\mathbf{b} \in \mathbb{R}^{1\times k}$, el cálculo hacia adelante produce una salida $\mathbf{Y} \in \mathbb{R}^{N\times k}$ mediante una multiplicación matricial y una traslación. En la implementación, se cachea la entrada para el paso de retropropagación, donde se calculan los gradientes $\mathbf{dW}$, $\mathbf{db}$ y $\mathbf{dX}$ de manera vectorizada. Este diseño evita bucles sobre muestras, aprovecha BLAS subyacente y hace que tanto el entrenamiento como la inferencia sean eficientes. Además, se respeta el broadcasting de $\mathbf{b}$ por filas y se asegura coherencia de formas con `NumPy`.

$$\mathbf{y} = \mathbf{XW} + \mathbf{b}$$

$$\mathbf{dW} = \mathbf{X}^\top \mathbf{dY}, \qquad \mathbf{db} = \sum_{i=1}^{N} \mathbf{dY}_i, \qquad \mathbf{dX} = \mathbf{dYW}^\top$$

Algoritmo 1 Algoritmo de propagación en capa densa (Forward y Backward)

1: Dado $\mathbf{X}$, calcular $\mathbf{Y} \leftarrow \mathbf{XW}$ y, si hay sesgo, $\mathbf{Y} \leftarrow \mathbf{Y} + \mathbf{b}$.
2: Guardar $\mathbf{X}$ para el proceso de retropropagación.
3: Recibir $\mathbf{dY}$ y computar $\mathbf{dW} = \mathbf{X}^\top \mathbf{dY}$.
4: Si hay sesgo, computar $\mathbf{db} = \text{sum\_rows}(\mathbf{dY})$.
5: Devolver $\mathbf{dX} = \mathbf{dYW}^\top$ al nivel anterior.

3.3. Funciones de Activación

Las funciones de activación introducen no linealidad, lo que permite a la red aproximar funciones complejas y separar clases no linealmente separables. En este proyecto se emplean Sigmoid, Tanh y ReLU para capas ocultas, así como Softmax en la salida para problemas de clasificación multiclase. Cada función tiene propiedades numéricas distintas: sigmoid y tanh pueden saturarse para valores grandes en magnitud, afectando a los gradientes, mientras que ReLU evita la saturación positiva y acelera el aprendizaje, a costa de poder "apagar" neuronas si sus entradas son negativas de forma persistente. Softmax convierte logits en probabilidades normalizadas, y su jacobiano se usa en retropropagación para propagar correctamente derivadas entre clases.

Sigmoid

La función sigmoide mapea números reales al intervalo $[0, 1]$, lo que facilita interpretaciones probabilísticas por neurona. Sin embargo, si x es muy positivo o muy negativo, $\sigma(x)$ se satura cerca de 1 o 0 y $\sigma'(x)$ se hace pequeña, ralentizando el aprendizaje. En capas intermedias profundas puede provocar desvanecimiento del gradiente, aunque sigue siendo útil en salidas binarias o como componente didáctico para validar derivadas y retropropagación.

$$f(x) = \frac{1}{1 + e^{-x}}, \qquad f'(x) = f(x)(1 - f(x))$$

Tanh

La tangente hiperbólica centra las activaciones alrededor de cero, lo que en general favorece una dinámica de aprendizaje mejor que la sigmoide al reducir desplazamientos constantes. Aun así, también puede saturar en ± 1, de forma que sus derivadas decrecen para $|x|$ grandes. Es una alternativa razonable a sigmoid cuando se desea simetría y cierta robustez frente a sesgos en la distribución de activaciones.

$$f(x) = \tanh(x), \qquad f'(x) = 1 - f(x)^2$$

ReLU

La unidad lineal rectificada deja pasar valores positivos y anula negativos. Esta pieza sencilla evita saturación en el semieje positivo y, con inicialización de He, mantiene la varianza de las activaciones. Su derivada es 1 para $x > 0$ y 0 en otro caso. En la práctica, acelera el entrenamiento y es el estándar de facto, aunque puede generar neuronas

"muertas" si los pesos llevan a entradas negativas persistentes.

$$f(x) = \text{máx}(0, x), \qquad f'(x) = \begin{cases} 1 & \text{si } x > 0, \\ 0 & \text{en otro caso} \end{cases}$$

Softmax

Softmax transforma logits en una distribución de probabilidad sobre C clases, garantizando positividad y suma uno. En la implementación se resta el máximo por fila para estabilizar la exponenciación. En el backward, se usa el jacobiano para propagar gradientes entre componentes, aunque cuando se combina con entropía cruzada, la derivada se simplifica notablemente.

$$f_i(\mathbf{x}) = \frac{e^{x_i}}{\sum_j e^{x_j}}, \qquad \frac{\partial f_i}{\partial x_j} = f_i(\delta_{ij} - f_j)$$

3.4. Regularización: Dropout

El *dropout* combate el sobreajuste anulando aleatoriamente activaciones durante el entrenamiento. Para una probabilidad p, se genera una máscara $\mathbf{m} \sim \text{Bernoulli}(1-p)$ del tamaño de la activación y se aplica un escalado por $\frac{1}{1-p}$ para conservar el valor esperado. De este modo, cada *forward* "muestrea" una subred distinta, y el modelo aprende representaciones más robustas. En inferencia no se aplican máscaras y se usa directamente el valor sin apagado. La implementación en el proyecto sigue este esquema con máscaras `float32` y el mismo escalado en el backward para mantener coherencia de magnitudes.

$$\tilde{h}_i = \begin{cases} \dfrac{h_i}{1-p}, & \text{con probabilidad } (1-p), \\ 0, & \text{con probabilidad } p. \end{cases}$$

Algoritmo 2 Algoritmo de regularización Dropout (entrenamiento y evaluación)

1: **if** modo entrenamiento **then**
2: Generar $\mathbf{mask} = (\text{rand} \geq p)$ y calcular $\mathbf{Y} = \mathbf{X} \odot \mathbf{mask}/(1-p)$.
3: **else**
4: Devolver $\mathbf{Y} = \mathbf{X}$.
5: **end if**
6: En la fase de retropropagación, multiplicar el gradiente por $\mathbf{mask}/(1-p)$.

3.5. Funciones de Pérdida

Las funciones de pérdida cuantifican la discrepancia entre predicción y objetivo, guiando el ajuste de los parámetros. En el proyecto se incluyen MSE para tareas afines a regresión o salidas continuas, y entropía cruzada para clasificación multiclase con logits y softmax estable. En ambos casos se normaliza por el tamaño del lote, y en MSE también por el número de canales en su implementación, lo que equilibra las escalas de gradientes frente a distintos tamaños de salida. Además, en entropía cruzada se emplea *log-sum-exp* para prevenir desbordamientos numéricos, restando el máximo por fila antes de calcular probabilidades y logaritmos.

Error Cuadrático Medio (MSE)

Mide la distancia euclídea al cuadrado entre predicción y objetivo, adecuada cuando se desea penalizar simétricamente errores positivos y negativos. Su derivada es lineal en la diferencia, lo que facilita análisis y depuración, y resulta útil como punto de partida para validar el motor de backpropagation, aunque en clasificación su superficie puede ser menos informativa que la de la entropía cruzada.

$$L = \frac{1}{2N} \sum_{i=1}^{N} \|\hat{\mathbf{y}}_i - \mathbf{y}_i\|^2, \qquad \frac{\partial L}{\partial \hat{\mathbf{y}}} = \frac{\hat{\mathbf{y}} - \mathbf{y}}{NC}.$$

Entropía Cruzada Multiclase

La entropía cruzada mide la desemejanza entre la distribución verdadera (one-hot) y la estimada por softmax. Es la elección estándar en clasificación multiclase, ya que su gradiente respecto a los logits se simplifica a la diferencia entre probabilidades predichas y etiquetas. En la implementación se vectoriza el cálculo para mini-batches y se asegura estabilidad numérica restando el máximo y usando $\log \sum \exp$.

$$L = -\frac{1}{N} \sum_{i=1}^{N} \sum_{c=1}^{C} y_{ic} \log \hat{y}_{ic}, \qquad \frac{\partial L}{\partial \mathbf{z}} = \frac{\hat{\mathbf{p}} - \mathbf{y}}{N}, \;\; \hat{\mathbf{p}} = \mathrm{softmax}(\mathbf{z}).$$

Algoritmo 3 Algoritmo de cálculo de la Entropía Cruzada (forward y backward)

1: Centrar los logits por fila: $\mathbf{z} \leftarrow \mathbf{z} - \max(\mathbf{z})$.
2: Calcular las probabilidades logarítmicas: $\log \hat{\mathbf{p}} = \mathbf{z} - \log \sum_j e^{z_j}$.
3: Calcular la pérdida media: $L = -\frac{1}{N} \sum y \log \hat{p}$.
4: En la retropropagación, devolver el gradiente: $(\hat{\mathbf{p}} - \mathbf{y})/N$.

3.6. Optimizadores

Los optimizadores actualizan los parámetros usando gradientes y, en su caso, memorias de momento o acumuladores. En el proyecto se implementan SGD con momento, RMSProp y Adam, cada uno con decaimiento L2 opcional aplicado únicamente a los pesos (no a sesgos). Esta elección reduce el sobreajuste sin desplazar sistemáticamente las salidas por el sesgo. Asimismo, todas las actualizaciones están vectorizadas por parámetro y compatibles con `NumPy`.

Descenso de Gradiente Estocástico (SGD)

SGD actualiza parámetros con el gradiente de un mini-batch, lo que introduce ruido beneficioso que puede ayudar a escapar de mínimos locales o mesetas. El momento acumula una media exponencial de gradientes pasados para suavizar oscilaciones y acelerar en direcciones coherentes; el L2 añade una fuerza de atracción al origen sobre los pesos, penalizando magnitudes grandes y mejorando la generalización. En nuestro código, el estado de momento se mantiene por clave de parámetro y se respeta el patrón de sólo regularizar las matrices de pesos.

$$\mathbf{g}_t \leftarrow \nabla_{\mathbf{w}} L_t + \lambda\, \mathbf{w}_t \;\; (\text{si } \mathbf{w} \text{ es peso}), \qquad \mathbf{v}_t = \mu \mathbf{v}_{t-1} + \mathbf{g}_t, \qquad \mathbf{w}_{t+1} = \mathbf{w}_t - \eta\, \mathbf{v}_t.$$

Algoritmo 4 Algoritmo de optimización SGD con momento y decaimiento de peso

for cada parámetro w **do**
    Calcular el gradiente $g \leftarrow \text{grad}(w)$.
    if w es un peso **then**
        Aplicar decaimiento L2: $g \leftarrow g + \lambda w$.
    end if
    Actualizar la velocidad: $v \leftarrow \mu v + g$.
    Actualizar el parámetro: $w \leftarrow w - \eta v$.
end for

RMSProp

RMSProp adapta la escala del paso por componente en función de un promedio móvil de los gradientes al cuadrado. De este modo, coordenadas con gradientes sistemáticamente grandes reciben pasos más pequeños, y viceversa, lo que ayuda en superficies mal condicionadas. En la implementación, cada parámetro mantiene su acumulador de segundo orden y se suma L2 a los pesos antes de actualizar. RMSProp suele requerir menos ajuste fino de η que SGD puro y es estable en una amplia gama de problemas.

$$\mathbf{s}_t = \beta \mathbf{s}_{t-1} + (1-\beta)\,\mathbf{g}_t^2, \qquad \mathbf{w}_{t+1} = \mathbf{w}_t - \eta\,\frac{\mathbf{g}_t}{\sqrt{\mathbf{s}_t} + \epsilon}.$$

Algoritmo 5 Algoritmo de optimización RMSProp

for cada parámetro w **do**
    Calcular el gradiente $g \leftarrow \text{grad}(w)$.
    if w es un peso **then**
        Aplicar decaimiento L2: $g \leftarrow g + \lambda w$.
    end if
    Actualizar el promedio móvil de los gradientes al cuadrado: $s \leftarrow \beta s + (1-\beta)g^2$.
    Actualizar el parámetro: $w \leftarrow w - \eta\, g/(\sqrt{s} + \epsilon)$.
end for

Adam

Adam combina momento de primer orden y acumulación de segundo orden con correcciones de sesgo temporal, ofreciendo pasos adaptativos y direccionales. En fases iniciales, las correcciones por $1-\beta_1^t$ y $1-\beta_2^t$ son críticas para no infraestimar las magnitudes. En el proyecto, se implementa la fórmula estándar, con estados $\mathbf{m}$ y $\mathbf{v}$ por parámetro y L2 opcional en los pesos. Suele converger rápido y de forma estable con hiperparámetros por defecto ($\beta_1 = 0{,}9, \beta_2 = 0{,}999, \epsilon = 10^{-8}$).

$$\mathbf{m}_t = \beta_1 \mathbf{m}_{t-1} + (1-\beta_1)\mathbf{g}_t, \quad \mathbf{v}_t = \beta_2 \mathbf{v}_{t-1} + (1-\beta_2)\mathbf{g}_t^2,$$
$$\hat{\mathbf{m}}_t = \frac{\mathbf{m}_t}{1-\beta_1^t}, \qquad \hat{\mathbf{v}}_t = \frac{\mathbf{v}_t}{1-\beta_2^t}, \qquad \mathbf{w}_{t+1} = \mathbf{w}_t - \eta\,\frac{\hat{\mathbf{m}}_t}{\sqrt{\hat{\mathbf{v}}_t} + \epsilon}.$$

Algoritmo 6 Algoritmo de optimización Adam con corrección de sesgo y regularización L2 opcional

Inicializar $m \leftarrow 0$, $v \leftarrow 0$, $t \leftarrow 0$.
for cada iteración **do**
    $t \leftarrow t + 1$.
    for cada parámetro w **do**
        Calcular gradiente $g \leftarrow \text{grad}(w)$.
        if w es un peso **then**
            Aplicar regularización L2: $g \leftarrow g + \lambda w$.
        end if
        Actualizar promedios móviles: $m \leftarrow \beta_1 m + (1-\beta_1)g$, $v \leftarrow \beta_2 v + (1-\beta_2)g^2$.
        Corregir sesgos: $\hat{m} \leftarrow m/(1-\beta_1^t)$, $\hat{v} \leftarrow v/(1-\beta_2^t)$.
        Actualizar parámetro: $w \leftarrow w - \eta\,\hat{m}/(\sqrt{\hat{v}} + \epsilon)$.
    end for
end for

3.7. Estrategias de Entrenamiento

Además de la arquitectura y el optimizador, la rutina de entrenamiento influye de forma decisiva en la calidad y estabilidad del aprendizaje. El uso de mini-batches permite aprovechar la vectorización y reduce la varianza de los gradientes frente al caso puramente estocástico. El *early stopping* previene el sobreajuste interrumpiendo el proceso cuando la pérdida de validación deja de mejorar durante un número de épocas; en nuestro código, además, se restauran los mejores parámetros observados. Por último, los programadores de tasa de aprendizaje (*schedulers*) ajustan η a lo largo de las épocas para explorar mejor al inicio y refinar al final, reduciendo saltos bruscos y ayudando a salir de mesetas.

Entrenamiento con Mini-Batches

En cada época se baraja el conjunto de entrenamiento y se itera por lotes de tamaño fijo. Para cada mini-batch, se calcula el *forward* para obtener logits o predicciones, se computa la pérdida, se obtiene el gradiente de la pérdida respecto a la salida, se realiza la retropropagación capa a capa y finalmente el optimizador actualiza los parámetros. Este proceso reduce el coste de memoria frente al *full batch*, mejora la estabilidad frente al caso online y explota eficientemente las operaciones matriciales.

Algoritmo 7 Algoritmo de entrenamiento con mini-batches

for cada época $= 1, \ldots, E$ **do**
  for cada lote $(\mathbf{X}_b, \mathbf{y}_b)$ **do**
    Calcular la salida directa: $\mathbf{z}_b \leftarrow \text{forward}(\mathbf{X}_b)$.
    Evaluar la función de pérdida: $L_b \leftarrow \text{loss}(\mathbf{z}_b, \mathbf{y}_b)$.
    Calcular el gradiente de la pérdida: $\mathbf{g}_b \leftarrow \text{loss\_backward}(\mathbf{z}_b, \mathbf{y}_b)$.
    Propagar el gradiente hacia atrás: backward_network($\mathbf{g}_b$).
    Actualizar los parámetros del modelo: optimizer.step().
  end for
end for

Early Stopping

El *early stopping* monitoriza la pérdida de validación y detiene el entrenamiento si no hay mejora significativa durante un número de épocas definido por *patience*. En nuestro `Trainer`, cuando la validación mejora se guardan copias profundas de los parámetros; cuando se agota la paciencia, se restauran esos "mejores pesos" y se concluye. Este enfoque es sencillo, robusto y suele dar mejores resultados que entrenar un número fijo de épocas sin control del ajuste excesivo.

Algoritmo 8 Algoritmo de Early Stopping

Inicializar $best\_val \leftarrow +\infty$, $patience\_left \leftarrow P$.
for cada época **do**
    Entrenar una época y evaluar la pérdida de validación val_loss.
    if val_loss mejora respecto a $best\_val$ con margen δ **then**
        Actualizar $best\_val \leftarrow val\_loss$.
        Guardar los parámetros actuales y reiniciar la paciencia: $patience\_left \leftarrow P$.
    else
        Reducir paciencia: $patience\_left \leftarrow patience\_left - 1$.
        if $patience\_left = 0$ **then**
            Restaurar los mejores parámetros y finalizar el entrenamiento.
        end if
    end if
end for

Programadores de Tasa de Aprendizaje (Schedulers)

Los *schedulers* modifican η a lo largo del entrenamiento. El *step decay* reduce la tasa de aprendizaje cada cierto número de épocas, generando escalones que facilitan escapar de mesetas y consolidar avances; su simplicidad lo hace predecible y fácil de combinar con *early stopping*. El *cosine annealing* sigue una trayectoria suave desde η_0 hasta casi cero, favoreciendo exploración al principio y refinamiento al final sin cambios abruptos, lo que suele traducirse en curvas de pérdida más estables.

$$\eta_t = \eta_0 \cdot \mathrm{drop}^{\lfloor t/\mathrm{every} \rfloor} \qquad \text{y} \qquad \eta_t = \frac{\eta_0}{2}\left(1 + \cos\Big(\frac{\pi t}{T_{\max}}\Big)\right).$$

Algoritmo 9 Algoritmo de programación de la tasa de aprendizaje (Step Decay y Cosine Annealing)

for cada época t **do**
    if programador = Step Decay **then**
        Actualizar la tasa de aprendizaje: $\eta \leftarrow \eta_0 \cdot \mathrm{drop}^{\lfloor t/\mathrm{every} \rfloor}$.
    else
        Actualizar la tasa de aprendizaje con Cosine Annealing: $\eta \leftarrow \frac{\eta_0}{2}\big(1 + \cos(\pi t/T_{\mathrm{máx}})\big)$.
    end if
    Asignar η al optimizador y entrenar la época correspondiente.
end for

4. Detalles de Implementación

La implementación del sistema se basa en una arquitectura modular que prioriza la claridad del código, la escalabilidad y la separación de responsabilidades entre componentes. Cada módulo cumple una función dentro del proceso de construcción, entrenamiento y evaluación de la red neuronal, permitiendo extender o modificar partes del motor sin afectar su funcionamiento general. Esta estructura facilita la depuración, el mantenimiento y la experimentación con diferentes configuraciones o algoritmos.

En primer lugar, la carpeta `src` constituye el núcleo del motor y contiene las clases y funciones que implementan los fundamentos de la red neuronal. Incluye `layers.py` (capas densas y activaciones con *forward/backward*), `losses.py` (funciones de pérdida como *MSE* y *Cross Entropy*), `optimizers.py` (métodos de optimización), `network.py` (control del flujo de entrenamiento y retropropagación) y `utils.py` (utilidades de inicialización, métricas y manipulación de datos).

El directorio `notebooks` contiene los cuadernos de experimentación empleados para validar el funcionamiento del motor mediante los conjuntos de datos Iris y MNIST. En ellos se documenta el proceso de entrenamiento, la evolución de las métricas y las representaciones visuales que ayudan a interpretar el comportamiento del modelo.

En la carpeta `results` se almacenan las salidas gráficas generadas a partir de los experimentos, incluyendo las curvas de pérdida y precisión, las matrices de confusión y otros gráficos que permiten analizar visualmente la calidad del aprendizaje y la generalización de la red.

La carpeta `data` organiza los conjuntos de datos utilizados en los experimentos. Contiene tanto el archivo `iris.csv`, empleado para la validación básica del motor, como los ficheros comprimidos del conjunto MNIST, necesarios para la clasificación de imágenes de dígitos manuscritos.

Por último, el directorio `tests` integra los cuadernos y scripts destinados a la validación técnica del motor. En esta entrega, se ha ampliado la verificación automática mediante el archivo `unit_tests.py`, que ejecuta un conjunto completo de pruebas unitarias para comprobar el funcionamiento de los métodos principales del sistema. En concreto, se valida el comportamiento de `layers.py` (capa densa y activaciones), `losses.py` (funciones de pérdida), `optimizers.py` (métodos de optimización), `network.py` (red secuencial y entrenamiento con `Trainer`), así como las utilidades de `utils.py` (codificación *one-hot*, creación de mini-lotes y particionado *train/val/test*). Adicionalmente, el cuaderno `test_gradcheck.ipynb` mantiene la verificación numérica de gradientes mediante diferencias finitas, contrastando los gradientes analíticos del motor con aproximaciones numéricas.

5. Experimentos y Resultados

Para evaluar el rendimiento del modelo desarrollado, se realizaron una serie de experimentos enfocados en analizar su comportamiento en términos de pérdida, precisión, capacidad de generalización y errores de clasificación. Los resultados obtenidos reflejan la evolución del proceso a lo largo de las fases de entrenamiento, validación y prueba, permitiendo observar la estabilidad del modelo y su capacidad para ajustarse a los datos sin incurrir en sobreajuste. Además, se incluyen representaciones gráficas y ejemplos visuales que ilustran el desempeño final alcanzado bajo las configuraciones más óptimas.

5.1. Experimento con el conjunto de datos Iris

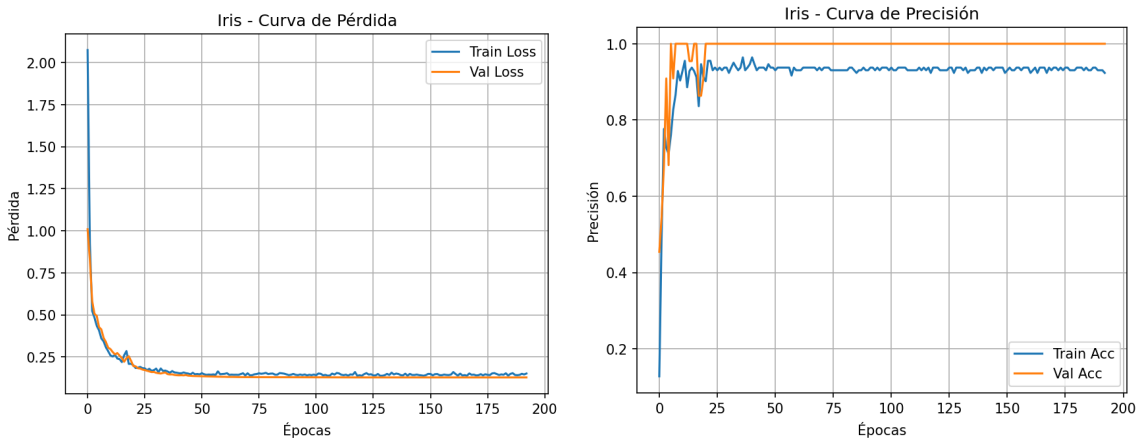

Figura 5.1: Curvas de pérdida y precisión durante el entrenamiento para el dataset Iris.

Como se observa en la Figura 5.1, la curva de pérdida evidencia un descenso pronunciado en las primeras fases del entrenamiento, con valores iniciales superiores a 2.0 en el conjunto de entrenamiento y cercanos a 1.0 en validación. Ambas curvas disminuyen rápidamente hasta situarse por debajo de 0.20 durante las primeras veinte épocas. A partir de ese punto, la pérdida se estabiliza en un rango aproximado de 0.13 a 0.15, manteniendo una correspondencia muy estrecha entre entrenamiento y validación. Esta coherencia refleja un ajuste adecuado del modelo, sin señales de sobreajuste ni inestabilidades numéricas.

En cuanto a la curva de precisión, el modelo muestra un proceso de aprendizaje rápido y estable. La precisión de entrenamiento parte de valores cercanos al 12 %, pero experimenta un incremento abrupto durante las primeras épocas, superando el 90 % alrededor de la iteración seis. Tras ello, se estabiliza en un intervalo aproximado de 0.93–0.94. En el conjunto de validación se observan fluctuaciones iniciales durante las primeras épocas, pero la precisión alcanza el 100 % alrededor de la época diez y se mantiene constante hasta

el final del entrenamiento. Este comportamiento indica que el modelo logra identificar correctamente las fronteras de decisión en un problema de baja complejidad y con un conjunto de datos reducido, mostrando una excelente capacidad de generalización.

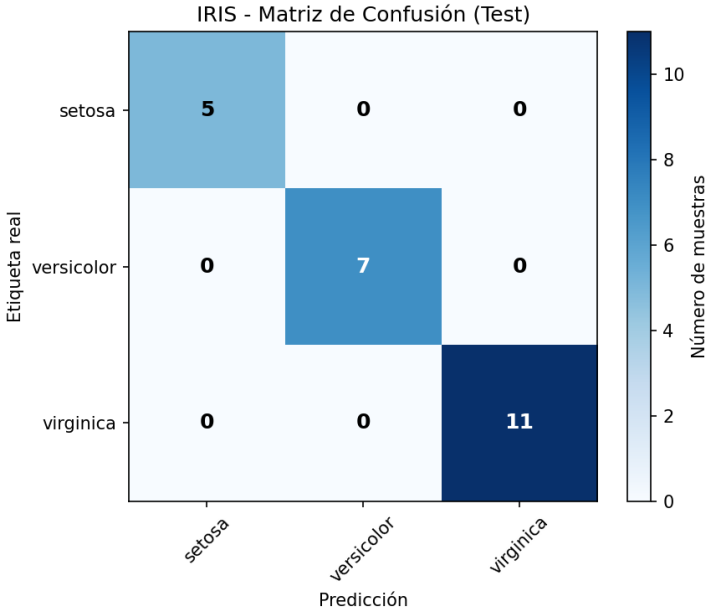

Figura 5.2: Matriz de confusión en el conjunto de prueba para el dataset Iris.

En la Figura 5.2 se presenta la matriz de confusión obtenida en el conjunto de prueba para el dataset Iris. La matriz muestra un comportamiento excepcional del modelo, con un 100 % de aciertos en las tres clases del conjunto de prueba. Todas las muestras de setosa, versicolor y virginica fueron clasificadas correctamente, registrando 5, 7 y 11 aciertos respectivamente. Este patrón refleja una separación clara entre las clases y pone de manifiesto que las fronteras aprendidas por la red neuronal son coherentes con la estructura estadística del dataset.

La ausencia total de errores de clasificación indica que el modelo ha capturado adecuadamente las relaciones entre las características del conjunto y que la arquitectura empleada es más que suficiente para resolver este problema multiclase de baja complejidad. Asimismo, la correspondencia entre estos resultados y las curvas de pérdida y precisión confirma que el entrenamiento ha sido estable, sin señales de sobreajuste ni divergencia, reforzando la solidez del motor en entornos controlados y de pequeña escala.

5.2. Experimento con el conjunto de datos MNIST

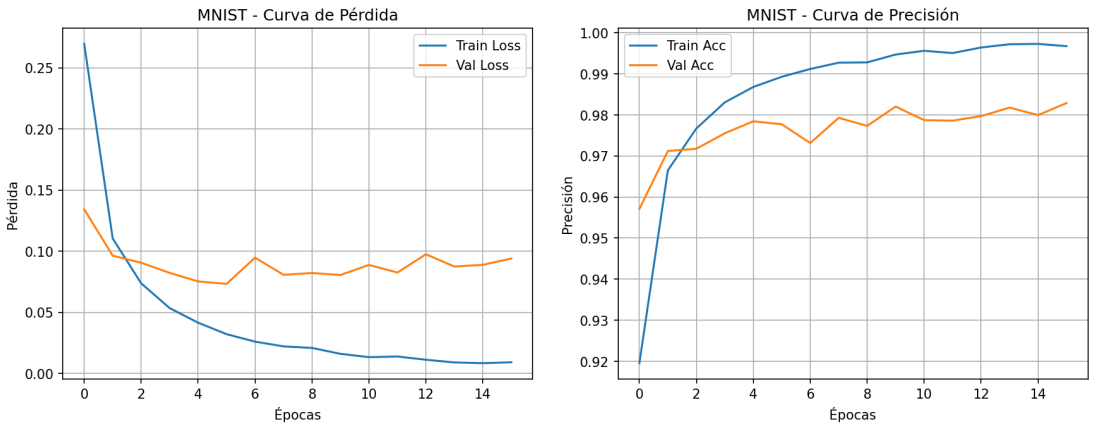

Figura 5.3: Curvas de pérdida y precisión durante el entrenamiento para el dataset MNIST.

Como se observa en la Figura 5.3, la evolución de la pérdida y la precisión del modelo durante el entrenamiento sobre MNIST evidencia un proceso de optimización altamente estable. En las primeras épocas, la pérdida de entrenamiento experimenta un descenso pronunciado, reduciéndose desde valores cercanos a 0.27 hasta situarse en torno a 0.05 antes de la quinta época. A partir de ese punto, la disminución continúa de manera más gradual hasta estabilizarse alrededor de 0.01, lo que indica que el modelo converge de forma eficiente hacia un mínimo adecuado. La pérdida de validación presenta un comportamiento coherente, iniciándose alrededor de 0.13 y manteniéndose de manera estable entre 0.07 y 0.10 durante el resto del entrenamiento. La ausencia de divergencias entre ambas curvas confirma que el modelo generaliza correctamente sin incurrir en sobreajuste.

La precisión de entrenamiento asciende rápidamente desde un valor inicial cercano al 92 % hasta superar el 98 % en las primeras cuatro épocas, alcanzando finalmente valores próximos al 99.7 %. Por su parte, la precisión de validación muestra un rendimiento elevado y estable desde las primeras iteraciones, comenzando en aproximadamente un 95.7 % y situándose de manera sostenida entre el 97.3 % y el 98.3 % a lo largo del entrenamiento. Las ligeras oscilaciones observadas en la curva de validación son propias del conjunto de datos y no indican inestabilidad del proceso de aprendizaje. La proximidad entre ambas curvas confirma la robustez del modelo y su capacidad para aprender patrones complejos sin pérdida de generalización.

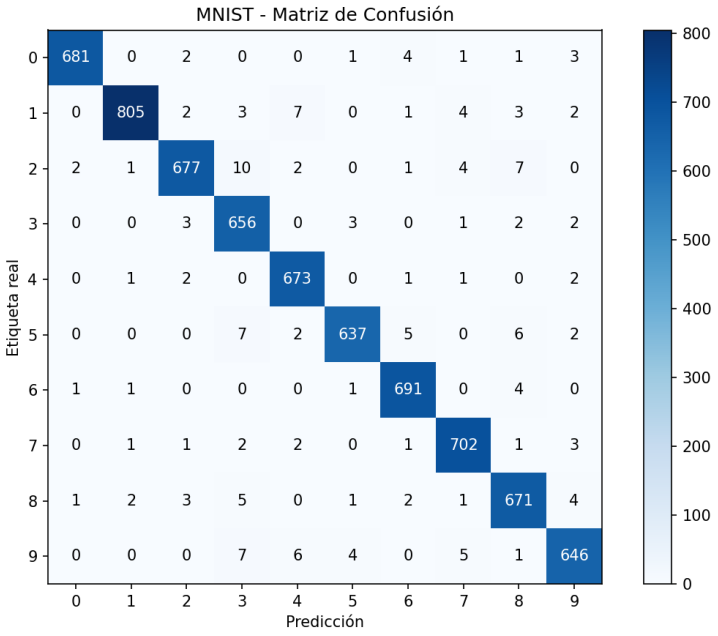

Figura 5.4: Matriz de confusión en el conjunto de prueba para el dataset MNIST.

En la Figura 5.4 se observa la matriz de confusión correspondiente al conjunto de prueba del dataset MNIST. La matriz muestra un rendimiento altamente satisfactorio en la tarea de clasificación de dígitos manuscritos. La diagonal principal concentra la gran mayoría de las predicciones, evidenciando una identificación correcta y consistente de todas las clases. En particular, los dígitos 0, 1, 4, 6, 7 y 8 registran un número de aciertos especialmente elevado, con niveles de error prácticamente despreciables, lo que indica que sus patrones visuales son capturados con gran precisión por la red.

Las confusiones observadas se concentran en dígitos que presentan similitudes estructurales o trazos parcialmente superpuestos. Destacan, por ejemplo, los casos del dígito 3 confundido ocasionalmente con el 5, o del 5 clasificado en ciertas ocasiones como 3 o 8. Del mismo modo, el dígito 9 muestra algunas confusiones con 4 y 7, patrones coherentes con la morfología variable que muestran estas clases dentro del conjunto MNIST. No obstante, estas desviaciones son cuantitativamente reducidas y no afectan de forma significativa al comportamiento global del modelo.

5.2.1. Predicciones con el conjunto de datos MNIST

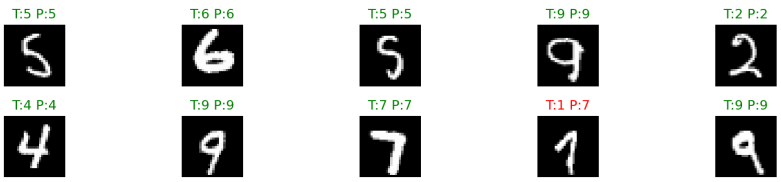

Figura 5.5: Ejemplos de predicciones realizadas por el modelo sobre imágenes del dataset MNIST.

La Figura 5.5 muestra ejemplos de predicciones realizadas por el modelo sobre imágenes del conjunto MNIST. Las predicciones muestran, la capacidad del modelo en identificar correctamente la gran mayoría de las imágenes, incluso cuando los dígitos presentan variaciones en trazo, grosor o curvatura. Los aciertos observados en clases como 5, 6, 7, 9 y 4 reflejan una representación interna del modelo suficientemente robusta para reconocer patrones manuscritos que pueden diferir notablemente entre muestras.

El conjunto también incluye un caso de predicción errónea una imagen etiquetada como 1 clasificada como 7 que coincide con los errores puntuales identificados en la matriz de confusión. Este tipo de confusión es coherente con la similitud morfológica que ciertos usuarios presentan al escribir ambos dígitos, especialmente cuando el trazo del 1 se inclina o adquiere un pequeño serif superior.

5.3. Comparativa general de resultados

Dataset	**Optimizador**	**Épocas**	**Train Acc**	**Test Acc**
Iris	Adam	193	93.33 %	100.00 %
MNIST	Adam	16	99.46 %	97.70 %

Cuadro 5.1: Resumen comparativo de resultados para Iris y MNIST.

En la Tabla 5.1 se resumen los resultados obtenidos en los experimentos realizados con los conjuntos de datos Iris y MNIST. En ambos casos, el motor de redes neuronales implementado desde cero mostró un comportamiento estable y una convergencia adecuada durante el proceso de entrenamiento.

En el conjunto Iris, caracterizado por un tamaño reducido y clases bien diferenciadas, el modelo alcanzó una precisión perfecta del 100 % en el conjunto de prueba tras 193 épocas. Este resultado indica que la arquitectura diseñada basada en tres capas densas con activación ReLU fue suficiente para aprender las relaciones entre las características del conjunto sin presentar signos de sobreajuste.

Por su parte, el conjunto MNIST, considerablemente más complejo debido a su naturaleza visual y dimensionalidad, obtuvo una precisión de entrenamiento del 99.46 % y una precisión de prueba del 97.70 %. Estos valores confirman que la red implementada generaliza correctamente, manteniendo un equilibrio adecuado entre capacidad de aprendizaje y regularización, favorecido por el uso de dropout y la estabilidad del optimizador Adam.

En términos generales, los resultados evidencian la versatilidad del motor neuronal desarrollado, capaz de adaptarse a datasets de distinta complejidad.

6. Conclusiones

En conclusión, el desarrollo de este motor de redes neuronales ha permitido comprender con profundidad los mecanismos internos que hacen posible el aprendizaje automático. Más allá de los resultados obtenidos en los experimentos, el verdadero valor del proyecto radica en haber construido una herramienta completamente funcional partiendo de los principios fundamentales, lo que ofrece una perspectiva clara sobre cómo interactúan entre sí las capas, las funciones de activación, la propagación del error y los métodos de optimización.

El proyecto demuestra que es posible implementar un sistema de aprendizaje robusto sin recurrir a frameworks especializados, siempre que se estructure el código de forma modular y se mantenga un control preciso sobre cada operación matemática. Este enfoque ha permitido detectar y resolver de manera directa problemas habituales en el entrenamiento, reforzando la comprensión de conceptos como la estabilidad numérica, la inicialización de parámetros o la importancia de la regularización.

Asimismo, la validación del motor en dos escenarios tan distintos como Iris y MNIST confirma que la arquitectura diseñada es flexible y capaz de adaptarse a retos de diferente complejidad. Esto evidencia que el motor no solo cumple su propósito académico, sino que puede servir como base para futuras extensiones y experimentación en entornos de aprendizaje más avanzados.

En definitiva, el proyecto constituye una aportación sólida tanto desde el punto de vista técnico como formativo. Ha permitido profundizar en la lógica interna del aprendizaje profundo, desarrollar competencias en diseño e implementación de software científico y establecer un punto de partida para abordar modelos más complejos con un entendimiento claro de sus fundamentos.

7. Trabajo Futuro

Como línea de trabajo futuro, se propone evaluar el motor en conjuntos de datos de complejidad significativamente superior a los utilizados en este proyecto, especialmente aquellos que presenten un mayor número de clases, una mayor variabilidad entre muestras o dimensiones más elevadas. Este tipo de escenarios permitiría analizar la capacidad del sistema para generalizar cuando el problema exige representaciones más ricas y modelos de mayor profundidad.

Otra posible mejora consiste en ampliar la arquitectura actual mediante el incremento de capas o neuronas, con el objetivo de estudiar cómo se comporta el motor cuando se incrementa su capacidad representativa. Esto permitiría explorar el equilibrio entre expresividad del modelo, tiempo de entrenamiento y tendencia al sobreajuste, proporcionando una visión más completa de las limitaciones y posibilidades del sistema.

También se plantea incorporar técnicas avanzadas que ayuden a reducir la confusión entre clases similares y a mejorar la robustez del aprendizaje. Entre ellas se incluyen ajustes más finos de los hiperparámetros, nuevos métodos de regularización y la aplicación de estrategias de data augmentation, que permiten aumentar artificialmente la diversidad del conjunto de entrenamiento sin necesidad de disponer de más datos reales.

Asimismo, sería conveniente integrar esquemas de validación más exhaustivos como la validación cruzadacon el fin de obtener estimaciones más estables del rendimiento del modelo y comparar de forma rigurosa distintas configuraciones. Este enfoque proporcionaría una base experimental más sólida para seleccionar parámetros y evaluar la consistencia del motor.

Por último, aunque el sistema ya incorpora varios algoritmos de optimización, queda abierta la posibilidad de realizar un análisis comparativo más profundo entre ellos. Estudiar de forma sistemática el comportamiento de Adam, SGD, RMSProp y AdaGrad en distintos contextos permitiría identificar sus ventajas y limitaciones, así como determinar qué configuraciones resultan más eficaces en problemas de mayor complejidad.

Bibliografía

[1] Goodfellow, I., Bengio, Y. y Courville, A. *Deep Learning*. MIT Press, Vol. 1, 2016.

[2] Bishop, C. M. *Pattern Recognition and Machine Learning*. Springer, Vol. 4, 2006.

[3] Haykin, S. *Neural Networks and Learning Machines*. Pearson Education, 3ª edición, 2009.

[4] LeCun, Y., Cortes, C. y Burges, C. J. C. *The MNIST Database of Handwritten Digits*. New York University, Vol. 1, 1998.

[5] Dua, D. y Graff, C. *Iris Dataset (CSV Format)*. UCI Machine Learning Repository, University of California, Irvine, Vol. 1, 2019.

Repositorio del proyecto

El código fuente completo desarrollado para este trabajo está disponible en el repositorio de GitHub.