Motor de Redes Neuronales

Autores

Lucía Cruz Toste
Diego Muñoz Torres
Agustín Darío Casebonne

Grado en Ciencia e Ingeniería de Datos
Universidad de Las Palmas de Gran Canaria

Curso académico 2025–2026

Fecha de entrega: 7 de noviembre de 2025

Índice

Resumen

El presente proyecto desarrolla un motor de redes neuronales artificiales implementado desde cero utilizando exclusivamente las bibliotecas `NumPy`, `Pandas` y `Matplotlib`. El objetivo principal es comprender y reproducir en detalle el funcionamiento interno de una red neuronal multicapa (feedforward) para tareas de clasificación multiclase, sin recurrir a frameworks de alto nivel como `TensorFlow` o `PyTorch`.

El sistema se estructura en módulos independientes que implementan los componentes fundamentales de una red neuronal: inicialización de parámetros, propagación hacia adelante y hacia atrás, funciones de activación (ReLU, sigmoid, tanh, softmax), funciones de pérdida (error cuadrático medio y entropía cruzada categórica), regularización L2, dropout, optimizador Adam y un procedimiento de entrenamiento con mini-lotes, decaimiento de la tasa de aprendizaje y parada temprana. Además, se incluye un módulo de búsqueda de hiperparámetros basado en una rejilla combinatoria para explorar configuraciones de capas, tasas de aprendizaje, regularización y dropout.

Los experimentos se realizaron sobre los conjuntos de datos Iris, MNIST y otro sobre clases de pingüinos (en formato CSV). El primero se utilizó para la validación funcional del modelo, el segundo como caso de uso de mayor complejidad y el tercero de manera adicional para seguir experimentando con el motor. Se aplicó un preprocesamiento estándar que incluye normalización de las entradas, estandarización por características y codificación one-hot de las etiquetas. La división de los datos se efectuó de manera estratificada en proporciones 80 % entrenamiento, 10 % validación y 10 % prueba.

Los resultados obtenidos demuestran que, incluso sin utilizar librerías especializadas, es posible alcanzar un rendimiento competitivo en tareas de clasificación multiclase y comprender de manera práctica la relación entre la teoría matemática y la implementación computacional. El trabajo sienta las bases para futuras ampliaciones del motor, como la incorporación de redes convolucionales y métodos de búsqueda de hiperparámetros más eficientes.

1. Introducción

En las últimas décadas, las redes neuronales artificiales se han consolidado como una de las herramientas más potentes dentro del aprendizaje automático, siendo la base de numerosos avances en visión por computador, procesamiento del lenguaje natural y sistemas de recomendación. Su capacidad para modelar relaciones no lineales y aprender representaciones jerárquicas a partir de grandes volúmenes de datos las convierte en un componente esencial de la inteligencia artificial moderna.

A pesar de la disponibilidad de frameworks especializados como `TensorFlow`, `PyTorch` o `Keras`, la comprensión profunda de los mecanismos internos de una red neuronal requiere analizar su funcionamiento desde una perspectiva algorítmica y matemática. Por este motivo, el presente trabajo tiene como objetivo principal **implementar desde cero un motor de redes neuronales feedforward evitando usar estas herramientas específicas**, de modo que cada componente, desde las ecuaciones teóricas hasta la actualización de pesos, sea programado y comprendido en detalle.

La motivación de este proyecto radica en reforzar la comprensión del proceso de aprendizaje supervisado, específicamente del algoritmo de backpropagation, de las funciones de activación y pérdida, y de los métodos de optimización basados en gradiente. El desarrollo manual de estos elementos permite visualizar la relación directa entre la formulación matemática y la implementación computacional, a la vez que proporciona una base sólida para futuras extensiones hacia arquitecturas más complejas.

El trabajo se ha desarrollado en tres fases complementarias:

1. **Diseño e implementación del modelo base:** desarrollo de una red neuronal multicapa con inicialización controlada de parámetros, funciones de activación configurables y regularización.
2. **Entrenamiento y optimización:** incorporación del algoritmo Adam, mini-lotes, decaimiento de la tasa de aprendizaje y parada temprana (*early stopping*).
3. **Experimentación y análisis:** validación del sistema mediante los conjuntos de datos *Iris*, *MNIST* y *Penguins*, evaluación de rendimiento y exploración de hiperparámetros.

En cuanto a la **estructura del proyecto**, el motor de redes neuronales se ha desarrollado siguiendo una organización modular que facilita la lectura, el mantenimiento y la experimentación reproducible. La jerarquía de carpetas se ha diseñado para separar claramente los componentes funcionales del sistema:

- `data/`: contiene los conjuntos de datos utilizados en los experimentos. En esta carpeta se almacenan los archivos `Iris.csv`, el conjunto `MNIST` y `penguin` en formato CSV. Esta disposición permite cargar los datos de forma directa desde los cuadernos o desde los scripts del motor.
- `notebooks/`: ncluye los cuadernos de Jupyter empleados para la exploración y validación del motor. Cada notebook está asociado a un conjunto de datos concreto (por ejemplo, `iris_experiment.ipynb` y `mnist_experiment.ipynb`), y permite ejecutar los entrenamientos, visualizar las curvas de pérdida y precisión, y generar la matriz de confusión correspondiente.
- `src/`: l núcleo del proyecto. Contiene las clases y módulos que implementan las funcionalidades principales del motor.
- `test/`: contiene los archivos de prueba unitaria que verifican el correcto funcionamiento del motor.

A lo largo de las secciones siguientes se explicará en detalle los conjuntos de datos, las ecuaciones fundamentales que sustentan el motor, las clases creadas, las técnicas de optimización empleadas y los resultados experimentales obtenidos sobre los conjuntos de datos Iris y MNIST.

2. Métodos y Conjuntos de Datos

2.1. Conjuntos de datos

Para la validación del motor de redes neuronales se han utilizado tres conjuntos de datos de naturaleza y complejidad diferentes: *Iris*, *MNIST* y *Penguins*. Todos ellos se han obtenido en formato `.csv` a partir de repositorios públicos de la plataforma *Kaggle*, descargándose manualmente con el fin de cumplir la restricción establecida en la práctica de trabajar únicamente con librerías básicas de `Python` (`NumPy`, `Pandas`, `Matplotlib`). Esta decisión permite controlar de manera explícita las rutas, los procesos de lectura y la estructura de los datos, garantizando además la compatibilidad directa con el motor desarrollado.

Conjunto *Iris*. Es un conjunto de datos clásico en clasificación supervisada. Contiene un total de 150 muestras de flores, distribuidas en tres clases (*Setosa*, *Versicolor* y *Virginica*) con 50 ejemplos por clase. Cada muestra está compuesta por cuatro atributos numéricos continuos: longitud y anchura del sépalo, y longitud y anchura del pétalo. Se trata de un problema de baja dimensionalidad, lo que lo hace idóneo para validar la correcta implementación de la red neuronal y el flujo de aprendizaje antes de abordar conjuntos de mayor complejidad.

Conjunto *MNIST*. Corresponde a un subconjunto del *Modified National Institute of Standards and Technology database*, que contiene imágenes en escala de grises de dígitos manuscritos (del 0 al 9). El conjunto descargado desde *Kaggle* se presenta ya combinado en un único archivo `.csv` con 70 000 muestras y 785 columnas: la primera columna representa la etiqueta de la clase, y las 784 restantes contienen los valores de intensidad de los píxeles (28x28). Este conjunto constituye un caso de uso de mayor escala y dimensionalidad, adecuado para evaluar la capacidad de generalización y la estabilidad del entrenamiento del motor.

Conjunto *Penguins*. El tercer conjunto utilizado es *Palmer Archipelago (Antarctica) Penguin Data*, que incluye mediciones morfológicas y biológicas de tres especies de pingüinos: *Adélie*, *Chinstrap* y *Gentoo*. Los datos fueron recopilados y puestos a disposición por la Dra. Kristen Gorman y el Palmer Station, Antarctica LTER, miembro de la *Long Term Ecological Research Network*. Se compone de aproximadamente 340 muestras y 7 atributos, incluyendo medidas como la longitud y profundidad del pico, longitud de la aleta, masa corporal y sexo del individuo.

En todos los conjuntos se ha aplicado un procedimiento de preprocesamiento común para garantizar la coherencia del flujo de datos. Primero, los valores numéricos se escalaron al rango $[0, 1]$ mediante división por su máximo valor posible. A continuación, se efectuó una estandarización por características, calculando la media y desviación típica del conjunto de entrenamiento y aplicándolas también a validación y prueba. Las etiquetas se transformaron mediante codificación *one-hot*, de modo que cada clase se representa por un vector binario de dimensión igual al número de categorías. Finalmente, cada conjunto se dividió en proporciones 80 % entrenamiento, 10 % validación y 10 % prueba, realizando una partición estratificada que preserva la proporción de clases en cada subconjunto.

2.2. Métodos

El motor desarrollado se basa en un modelo de **red neuronal multicapa** (*Multilayer Perceptron*, MLP), construido a partir de capas densamente conectadas y entrenado mediante aprendizaje supervisado. Este tipo de arquitectura constituye la base del aprendizaje profundo moderno y permite aproximar funciones no lineales de alta complejidad a partir de datos de entrada numéricos. La elección de un modelo MLP responde al objetivo principal del proyecto: comprender el funcionamiento interno de una red neuronal desde una perspectiva teórica y algorítmica, sin depender de frameworks externos.

En su forma general, el modelo incluye una serie de transformaciones lineales seguidas de funciones de activación no lineales, culminando en una capa de salida con activación *softmax* que produce una distribución de probabilidad sobre las clases posibles. Esta configuración es adecuada para tareas de clasificación multiclase como las presentes en los conjuntos *Iris*, *MNIST* y *Penguins*.

Durante el desarrollo se implementaron distintos componentes fundamentales: funciones de activación (*ReLU*, *sigmoid*, *tanh*), funciones de pérdida (error cuadrático medio y entropía cruzada categórica) y el optimizador *Adam*, ampliamente utilizado por su estabilidad y rapidez de convergencia. Para mejorar la capacidad de generalización del modelo se incorporaron técnicas de regularización L2 y *dropout*, que reducen el sobreajuste penalizando los pesos excesivos y desactivando neuronas de forma aleatoria durante el entrenamiento.

Estos métodos se implementaron de forma modular dentro del motor, de modo que cada componente (activaciones, pérdidas, optimización, entrenamiento) puede modificarse o ampliarse sin alterar el resto del sistema. Las ecuaciones matemáticas que formalizan estos procedimientos, así como los detalles específicos de la propagación y actualización de parámetros, se desarrollan en la sección.

3. Detalles de Implementación

3.1. Ecuaciones principales

El motor de redes neuronales desarrollado se basa en los fundamentos teóricos del aprendizaje supervisado mediante descenso de gradiente. A continuación, se presentan las ecuaciones esenciales que describen el funcionamiento interno del modelo, desde la propagación hacia adelante hasta la actualización de parámetros durante el entrenamiento. Todas las formulaciones se implementaron de forma explícita en los módulos del motor, sin recurrir a librerías de cálculo automático de gradientes.

Propagación hacia adelante (*Forward propagation*). El proceso de inferencia de una red neuronal multicapa consiste en aplicar una serie de transformaciones lineales seguidas de funciones de activación no lineales. Para cada capa $l \in \{1, \ldots, L\}$, se define:

$$z^{[l]} = W^{[l]} a^{[l-1]} + b^{[l]}, \qquad a^{[l]} = f^{[l]}(z^{[l]}),$$

donde $W^{[l]} \in \mathbb{R}^{n_l \times n_{l-1}}$ es la matriz de pesos, $b^{[l]} \in \mathbb{R}^{n_l}$ el vector de sesgos, y $f^{[l]}(\cdot)$ la función de activación correspondiente. El vector $a^{[0]}$ representa la entrada del modelo X, y la salida final $a^{[L]}$ constituye la predicción del modelo $\hat{Y}$. En la capa de salida se emplea la función *softmax*:

$$\text{softmax}(z_i^{[L]}) = \frac{e^{z_i^{[L]}}}{\sum_{k=1}^{K} e^{z_k^{[L]}}},$$

que garantiza que las salidas sean probabilidades normalizadas sobre las K clases.

Propagación hacia atrás (*Backward propagation*). El algoritmo de retropropagación del error [3, 4] permite calcular los gradientes de la función de pérdida con respecto a los parámetros de la red aplicando la regla de la cadena. Para la capa de salida, el error se define como:

$$\delta^{[L]} = \hat{Y} - Y,$$

donde Y son las etiquetas verdaderas y $\hat{Y}$ las predicciones del modelo. Para las capas ocultas $l = L-1, L-2, \ldots, 1$:

$$\delta^{[l]} = \left(W^{[l+1]^\top} \delta^{[l+1]} \right) \odot f'^{[l]}(z^{[l]}),$$

donde $\odot$ representa el producto elemento a elemento y $f'^{[l]}(\cdot)$ es la derivada de la función de activación de la capa correspondiente.

Los gradientes de la pérdida respecto a los parámetros se obtienen como:

$$\nabla_{W^{[l]}} = \frac{1}{m} \delta^{[l]} a^{[l-1]^\top} + \frac{\lambda}{m} W^{[l]}, \qquad \nabla_{b^{[l]}} = \frac{1}{m} \sum_{i=1}^{m} \delta^{[l](i)},$$

donde m es el número de muestras en el lote y λ el coeficiente de regularización L2.

Funciones de pérdida. Se implementaron dos funciones de coste principales: el error cuadrático medio (MSE) y la entropía cruzada categórica (CCE), según la naturaleza del problema:

$$\mathcal{L}_{MSE} = \frac{1}{2m}\sum_{i=1}^{m} \|Y^{(i)} - \hat{Y}^{(i)}\|_2^2, \qquad \mathcal{L}_{CCE} = -\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K} y_{ik}\log(\hat{y}_{ik}).$$

En ambos casos, la pérdida se combina con un término de regularización L2:

$$\mathcal{L}_{total} = \mathcal{L} + \frac{\lambda}{2m}\sum_{l=1}^{L} \|W^{[l]}\|_F^2.$$

Actualización de parámetros. La optimización de los parámetros se realiza mediante el algoritmo **Adam** (*Adaptive Moment Estimation*), que ajusta las tasas de aprendizaje de manera adaptativa para cada parámetro. En cada iteración t, se calculan los momentos de primer y segundo orden de los gradientes:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t, \qquad v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2,$$

con corrección de sesgo:

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}, \qquad \hat{v}_t = \frac{v_t}{1-\beta_2^t}.$$

Los parámetros se actualizan como:

$$\theta_t = \theta_{t-1} - \alpha \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \varepsilon},$$

donde α es la tasa de aprendizaje y ε un valor pequeño para evitar divisiones por cero. Este método proporciona una convergencia más rápida y estable que el descenso del gradiente clásico, especialmente en espacios de alta dimensionalidad [5, 1].

Regularización y *dropout*. Además de la penalización L2, el modelo incorpora la técnica de *dropout*, que consiste en desactivar aleatoriamente una fracción p_{drop} de las neuronas durante el entrenamiento:

$$a^{[l]} = \frac{M^{[l]} \odot a^{[l]}}{1 - p_{drop}}, \quad M^{[l]} \sim \text{Bernoulli}(1 - p_{drop}).$$

Esta operación fuerza a la red a no depender de neuronas específicas, mejorando su capacidad de generalización y reduciendo el sobreajuste, tal como se expone en Hinton et al. (2014). Durante la inferencia, el *dropout* se desactiva y las activaciones se mantienen completas.

Resumen. Las ecuaciones anteriores constituyen la base matemática del motor implementado. Su desarrollo manual ha permitido comprender la relación entre los conceptos teóricos del aprendizaje profundo y su implementación computacional, poniendo en práctica los principios descritos en [1, 2, 3, 6].

3.2. Arquitectura y componentes

El motor se ha diseñado con una arquitectura modular que separa con claridad las responsabilidades de cálculo, optimización y orquestación del entrenamiento. Esta organización facilita la mantenibilidad, la extensibilidad y el testeo independiente de cada pieza.

Vista general (`src/`)

- `NeuralNetwork.py` (*núcleo del modelo*): define la red MLP de forma declarativa mediante una lista de capas (*input* → ocultas → *output*). Implementa la propagación hacia adelante (activaciones capa a capa), la retropropagación y aplica dropout y regularización L2 cuando procede. Expone métodos típicos: forward, backward, predict y utilidades para inicialización (Xavier), gestión del estado y recuperación del mejor conjunto de parámetros.

- `Losses.py` (*funciones de coste*): implementa las pérdidas error cuadrático medio (MSE) y entropía cruzada categórica (CCE) con versiones numéricamente estables (por ejemplo, softmax con log-sum-exp). Ofrece también las derivadas necesarias para *backpropagation* y el cómputo de métricas auxiliares (p. ej., *accuracy* a partir de *argmax*).

- `OptimizerAdam.py` (*optimización*): gestiona el estado del optimizador (momentos m, v por parámetro), las correcciones de sesgo y la actualización por paso conforme a Adam. Acepta hiperparámetros α (*learning rate*), β_1, β_2 y ε, y está preparado para integrarse con *decay* externo de la tasa de aprendizaje.

- `Trainer.py` (*bucle de entrenamiento*): orquesta el ciclo *epoch* → *mini-lotes* → *forward/backward* → *optimizer.step*. Implementa barajado por época, división en mini-lotes, evaluación en validación, early stopping por paciencia, decaimiento de learning rate y registro de historial (pérdida y precisión de train/val). Mantiene un checkpoint interno con los mejores pesos en validación para mitigar sobreajuste.

- `HyperparameterTuner.py` (*búsqueda en rejilla*): recibe listas de valores (*grid*) para LR, batch, λ (L2), topologías (capas ocultas), activaciones, tasas de *dropout*, épocas y parámetros de entrenamiento. Ejecuta los entrenamientos, agrega resultados y devuelve la mejor configuración según **val_acc** (y, a empate, menor **val_loss**). Emite un contador de progreso con el formato `[i/N]` —configurable— para seguimiento reproducible.

- `Utils.py` (*utilidades de datos*): funciones para codificación one-hot, normalización a $[0, 1]$, estandarización con estadísticas de entrenamiento, partición estratificada 80/10/10, y comprobaciones de integridad (dimensiones, valores no finitos). Centraliza la lógica de preprocesado para mantener limpio el resto del código.

Flujo de datos y dependencias

1. **Preprocesado** (`Utils.py`): lectura del CSV, selección de la columna de etiqueta, escalado/estandarización, *one-hot*, estratificación y partición en $(X_{train}, Y_{train}, X_{val}, Y_{val}, X_{test}, Y_{test})$.
2. **Construcción del modelo** (`NeuralNetwork.py`): definición de capas $[d, h_1, \ldots, h_L, K]$, inicialización (Xavier), activaciones y parámetros de regularización/dropout.
3. **Entrenamiento** (`Trainer.py` + `OptimizerAdam.py` + `Losses.py`): mini-lotes $\rightarrow$ *forward* $\rightarrow$ pérdida (con L2) $\rightarrow$ *backward* $\rightarrow$ *Adam.step* $\rightarrow$ evaluación en validación $\rightarrow$ *early stopping/lr decay.*
4. **Selección de hiperparámetros** (`HyperparameterTuner.py`): explora la rejilla, registra `hist` y métricas, selecciona la mejor configuración, y opcionalmente reentrena con la configuración óptima.
5. **Evaluación** (`NeuralNetwork.py` + `Losses.py`): métricas finales en test y utilidades de análisis (p. ej., matriz de confusión en los cuadernos).

Pruebas y validación (`test/`)

El directorio `test/` contiene pruebas unitarias que aseguran regresiones cero y compatibilidad entre módulos:

- **Coherencia de salida** (*forward shape*): la dimensión de $\hat{Y}$ coincide con (m, K) para un lote de m muestras y K clases.
- **No negatividad de pérdidas**: MSE y CCE devuelven valores finitos y ≥ 0 en entradas válidas.
- **Entrenamiento reduce la pérdida**: en un problema sintético (p. ej., XOR) o un *mini* subconjunto, la pérdida desciende tras varias iteraciones de *forward/backward + step.*
- **Gradientes compatibles**: las formas de $\nabla W^{[l]}, \nabla b^{[l]}$ coinciden con las de sus parámetros; no se generan `NaN`/∞.
- **Determinismo**: con `seed` fijada, el historial de pérdida no varía entre ejecuciones.
- **Tuner acotado**: el número de combinaciones evaluadas coincide con el producto cartesiano de las listas y respeta el contador [`i/N`].

Resumen de integración

La secuencia típica es: `Utils` prepara los datos $\rightarrow$ `NeuralNetwork` define la arquitectura $\rightarrow$ `Trainer` coordina *forward/backward* y llama a `OptimizerAdam` $\rightarrow$ `Losses` computa coste y métricas $\rightarrow$ opcionalmente `HyperparameterTuner` explora la rejilla y devuelve la configuración óptima. Esta separación de responsabilidades reduce el acoplamiento y facilita tanto la depuración como la evolución futura del motor.

3.3. Optimización y entrenamiento

El proceso de entrenamiento constituye el núcleo operativo del motor desarrollado y permite materializar las ecuaciones teóricas ya descritas. Su objetivo es ajustar los parámetros del modelo, pesos y sesgos, para minimizar la función de pérdida y mejorar la precisión en los conjuntos de entrenamiento y validación.

Estrategia de optimización. El motor utiliza el algoritmo **Adam** como método principal de optimización. Su implementación se basa en las ecuaciones presentadas previamente, manteniendo los promedios móviles de los gradientes y sus cuadrados, aplicando correcciones de sesgo y actualizando los parámetros de forma adaptativa. Adam fue elegido por su estabilidad frente a oscilaciones, su rápida convergencia y su capacidad para adaptarse automáticamente a diferentes escalas de gradiente. Los valores por defecto empleados son $\beta_1 = 0{,}9$, $\beta_2 = 0{,}999$, $\varepsilon = 10^{-8}$ y una tasa de aprendizaje inicial $\alpha = 10^{-3}$.

Entrenamiento en mini-lotes. El entrenamiento se lleva a cabo siguiendo un enfoque de **mini-batch gradient descent**, que combina la eficiencia del procesamiento vectorizado con la estabilidad de las actualizaciones promedio. En cada época, los datos se barajan y dividen en lotes de tamaño fijo. Para cada mini-lote, el modelo realiza una pasada hacia adelante, calcula la pérdida, propaga los gradientes hacia atrás y actualiza los parámetros mediante Adam. El registro de métricas (pérdida y precisión) se almacena en un historial para el análisis posterior.

Control del aprendizaje. Se implementan varios mecanismos para estabilizar la convergencia:

- **Decaimiento de la tasa de aprendizaje**: tras cada época, la tasa α se multiplica por un factor de decaimiento definido en el entrenador. Esto permite pasos amplios al inicio y más pequeños conforme el modelo se acerca al mínimo.
- **Parada temprana** (*early stopping*): el entrenamiento se interrumpe si la pérdida de validación no mejora tras un número determinado de épocas (*paciencia*). El modelo conserva los parámetros con mejor rendimiento en validación.

Regularización y generalización. Durante el entrenamiento se aplican las técnicas de **regularización L2** y ***dropout***. La primera penaliza los pesos excesivos para evitar modelos sobreajustados, mientras que la segunda apaga aleatoriamente un porcentaje de neuronas en cada iteración, promoviendo redes más robustas. Ambas se aplican únicamente durante el entrenamiento y se desactivan durante la evaluación o inferencia.

En conjunto, el procedimiento implementado puede resumirse como sigue:

1. Inicialización de pesos y parámetros del optimizador.
2. División del conjunto de entrenamiento en mini-lotes y barajado.
3. Para cada mini-lote: propagación hacia adelante, cálculo de la pérdida, retropropagación y actualización mediante Adam.
4. Evaluación en validación, aplicación de *early stopping* y ajuste de la tasa de aprendizaje.
5. Evaluación final en el conjunto de prueba con los pesos óptimos.

Este esquema traduce directamente las ecuaciones teóricas en un procedimiento operativo reproducible, garantizando un equilibrio entre velocidad de convergencia y capacidad de generalización.

4. Experimentos y Resultados

4.1. Protocolo experimental

Con el objetivo de evaluar el comportamiento del motor de redes neuronales y verificar su correcta implementación, se realizaron una serie de experimentos controlados sobre los tres conjuntos de datos descritos previamente: *Iris*, *MNIST* y *Penguins*. El propósito de estas pruebas fue analizar la capacidad del modelo para aprender representaciones significativas, ajustar correctamente los hiperparámetros y comparar el rendimiento obtenido en escenarios de distinta complejidad.

División de los datos. En todos los conjuntos se aplicó la misma estrategia de partición: un 80 % de las muestras se destinaron a entrenamiento, un 10 % a validación y un 10 % a prueba. La división se realizó de manera estratificada para mantener la proporción original de clases. Los conjuntos se barajaron con una semilla fija para garantizar la reproducibilidad de los resultados.

Configuración de entrenamiento. Los modelos se entrenaron con distintas combinaciones de hiperparámetros, gracias al método desarrollado para obtener los óptimios, pero siguiendo el mismo esquema general:

- **Número de épocas**: entre 100 y 200 para los experimentos principales, con parada temprana cuando la pérdida de validación dejaba de mejorar.
- **Tamaño de lote**: 64 o 128 muestras por iteración, dependiendo del tamaño del conjunto.
- **Tasa de aprendizaje inicial**: $\alpha = 10^{-3}$, con decaimiento progresivo de un 1 % por época.
- **Regularización L2**: $\lambda = 10^{-4}$, penalizando los pesos excesivos para mejorar la generalización.
- **Dropout**: 0.2 en las capas ocultas, desactivado durante la evaluación.

En los experimentos de búsqueda de hiperparámetros, se variaron de forma sistemática la tasa de aprendizaje, el número de neuronas por capa, la regularización y la tasa de *dropout*, empleando el módulo `HyperparameterTuner.py` para recorrer las combinaciones definidas y registrar las métricas de validación correspondientes.

Arquitecturas empleadas. Cada conjunto de datos requirió un diseño de red distinto, adaptado al número de características de entrada y clases de salida, así como a la complejidad de la tarea:

- **Iris**: se empleó una red con arquitectura $[4, 12, 3]$, donde las 4 neuronas de entrada corresponden a las medidas de los sépalos y pétalos, y las 3 de salida a las especies. La capa oculta contiene 12 neuronas con activación $ReLU$. El optimizador Adam se configuró con una tasa de aprendizaje de 0.1, $\beta_1 = 0{,}99$, $\beta_2 = 0{,}95$, regularización L2 de 10^{-3} y *dropout* desactivado. El entrenamiento se ejecutó durante 180 épocas con lotes de 16 muestras y paciencia de 10 épocas para parada temprana.
- **MNIST**: se utilizó una red multicapa de arquitectura $[784, 256, 128, 64, 10]$, con tres capas ocultas y activación $ReLU$. La entrada de 784 neuronas representa los píxeles de imágenes de 28×28, y las 10 salidas corresponden a las clases numéricas. El optimizador Adam se configuró con una tasa de aprendizaje de 0.0005 y parámetros $\beta_1 = 0{,}9$, $\beta_2 = 0{,}999$, sin *dropout* y regularización L2 de 10^{-5}. El entrenamiento se realizó durante 30 épocas, con lotes de 64 muestras y decaimiento de la tasa de aprendizaje de 0.99.
- **Penguins**: el modelo siguió la arquitectura $[4, 18, 3]$, con 4 características de entrada (longitud y profundidad del pico, longitud de la aleta y masa corporal) y 3 clases de salida que representan las especies de pingüinos (*Adelie*, *Chinstrap* y *Gentoo*). La capa oculta contiene 18 neuronas con activación *tanh*. Se utilizó Adam con tasa de aprendizaje de 0.001, $\beta_1 = 0{,}99$, $\beta_2 = 0{,}95$, regularización L2 de 10^{-3}, sin *dropout* y decaimiento de 0.95. El modelo se entrenó durante 200 épocas con lotes de 32 muestras.

Cada configuración fue seleccionada a partir de las combinaciones más prometedoras detectadas por el módulo `HyperparameterTuner.py`, priorizando aquellas con la mayor precisión de validación y la menor pérdida asociada.

Métricas de evaluación. Para cada experimento se registraron y analizaron las siguientes métricas:

- **Pérdida de entrenamiento y validación** (*train loss, val loss*), como medida directa del error del modelo.
- **Precisión** (*accuracy*) en entrenamiento, validación y prueba, como métrica principal de rendimiento.
- **Evolución temporal** de la pérdida y la precisión por época, para visualizar la convergencia del entrenamiento.
- **Matriz de confusión** en el conjunto de prueba, para analizar los patrones de error entre clases.

Criterios de evaluación. El modelo final seleccionado en cada conjunto corresponde al que obtuvo la mayor precisión en validación y, en caso de empate, la menor pérdida de validación. Las pruebas se ejecutaron varias veces con distinta inicialización de pesos para comprobar la estabilidad de los resultados. El rendimiento global se considera satisfactorio cuando el modelo alcanza alta precisión en test con una brecha moderada respecto a validación, lo que indica una buena generalización.

4.2. Resultados en Iris

El conjunto *Iris* se utilizó como caso inicial de verificación del correcto funcionamiento del motor de red neuronal. Al tratarse de un problema clásico de clasificación multiclase con bajo número de atributos y ejemplos, su objetivo principal fue validar la implementación de la retropropagación, el descenso adaptativo del gradiente y las estrategias de control del aprendizaje.

El modelo de arquitectura $[4, 12, 3]$ se entrenó con tasa de aprendizaje 0.1 y función de activación $ReLU$. La Fig. 1 muestra que la pérdida disminuye de forma continua hasta estabilizarse antes de la época 40, mientras que en la Fig. 2 se aprecia una mejora progresiva en la precisión tanto de entrenamiento como de validación, con curvas casi paralelas que denotan una buena generalización. El mecanismo de parada temprana se activó tras 10 épocas sin mejora, indicando que el modelo alcanzó su rendimiento máximo sin evidencias de sobreajuste.

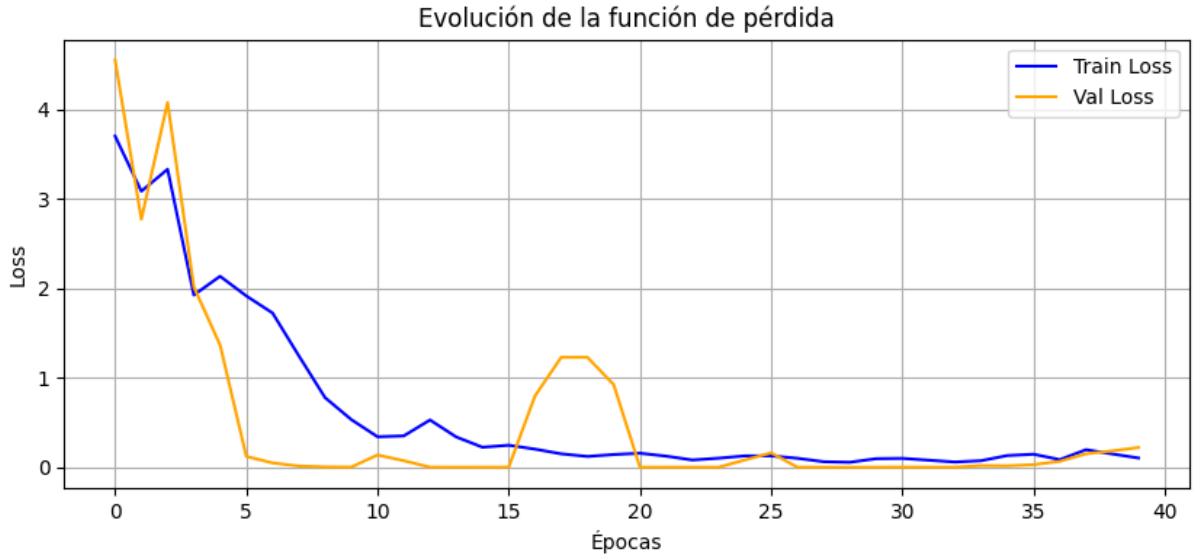

Figura 1: Evolución de la función de pérdida en *Iris*.

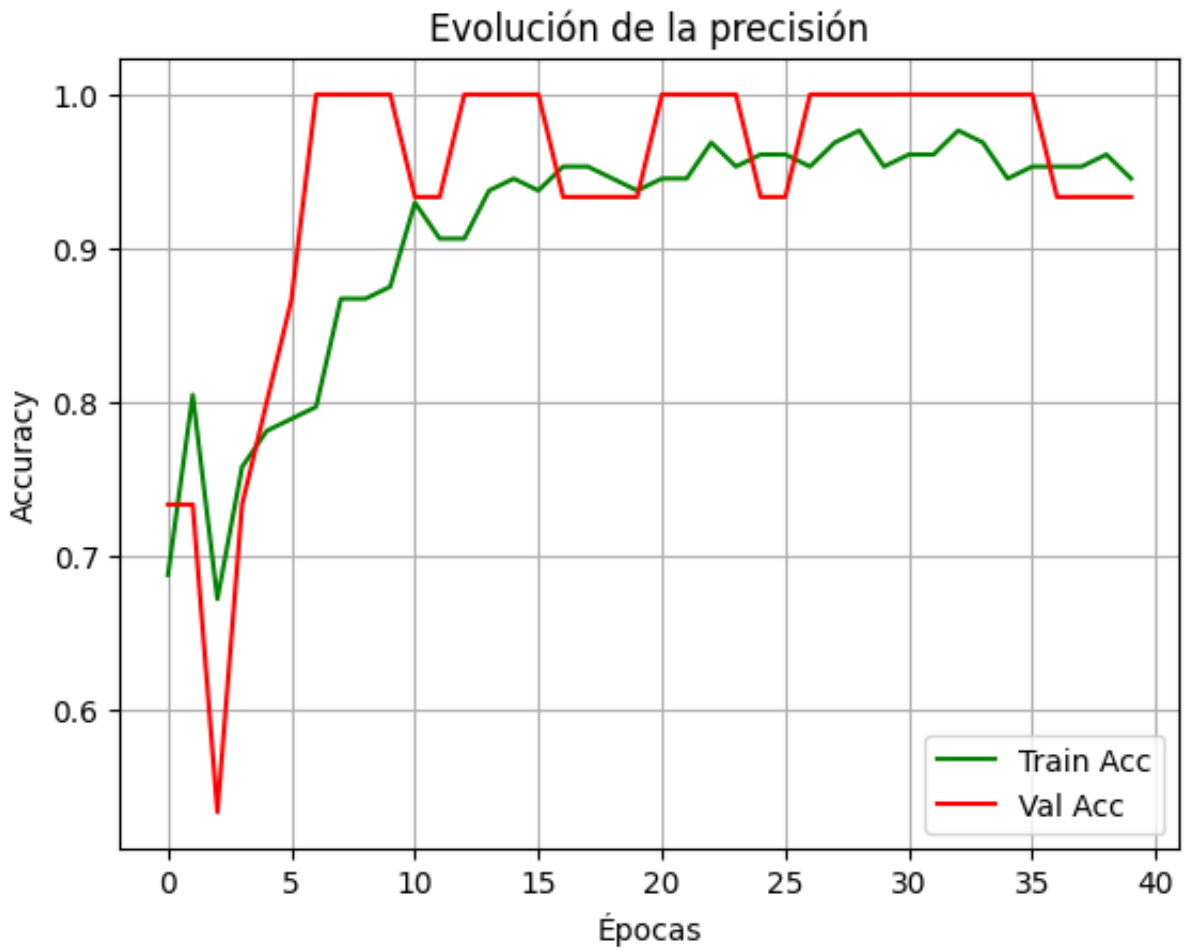

Figura 2: Evolución de la precisión en *Iris*.

Métrica	**Entrenamiento**	**Validación**	**Prueba**
Precisión (%)	96.09	93.33	93.33
Pérdida	0.1487	0.1826	–

Cuadro 1: Resultados cuantitativos en el conjunto *Iris*.

Los valores de la Table 1 muestran una concordancia estrecha entre las métricas de entrenamiento y validación, reflejando un comportamiento robusto y la correcta acción de la regularización L2. La precisión final del 93.33 % en prueba es coherente con las expectativas para una red densa simple en este conjunto. El análisis de la matriz de confusión revela que la mayoría de errores provienen de confundir las especies *versicolor* y *virginica*, cuyas características morfológicas presentan solapamiento. Estos resultados confirman que el motor reproduce adecuadamente el flujo de propagación hacia adelante y atrás, alcanzando un equilibrio satisfactorio entre rendimiento y estabilidad.

4.3. Resultados en MNIST

El conjunto *MNIST* sirvió como prueba de escalabilidad del motor frente a un problema de mayor dimensionalidad y complejidad. Cada muestra consiste en una imagen de 28×28 píxeles, lo que se traduce en 784 entradas por ejemplo, y una tarea de clasificación en diez clases correspondientes a los dígitos del 0 al 9. El objetivo principal fue evaluar la capacidad del modelo para aprender representaciones jerárquicas mediante capas densas, sin recurrir a arquitecturas convolucionales.

El modelo $[784, 256, 128, 64, 10]$ empleó activaciones $ReLU$ y el optimizador Adam con tasa de aprendizaje 0.0005. En la Fig. 3 se observa una reducción pronunciada de la pérdida durante las primeras épocas, seguida de una estabilización en torno a la época 20. La Fig. 4 muestra un crecimiento constante de la precisión, que alcanza valores cercanos al 98 % antes del *early stopping*. El entrenamiento se detuvo automáticamente tras 10 épocas sin mejora en la validación, señalando un punto óptimo de generalización.

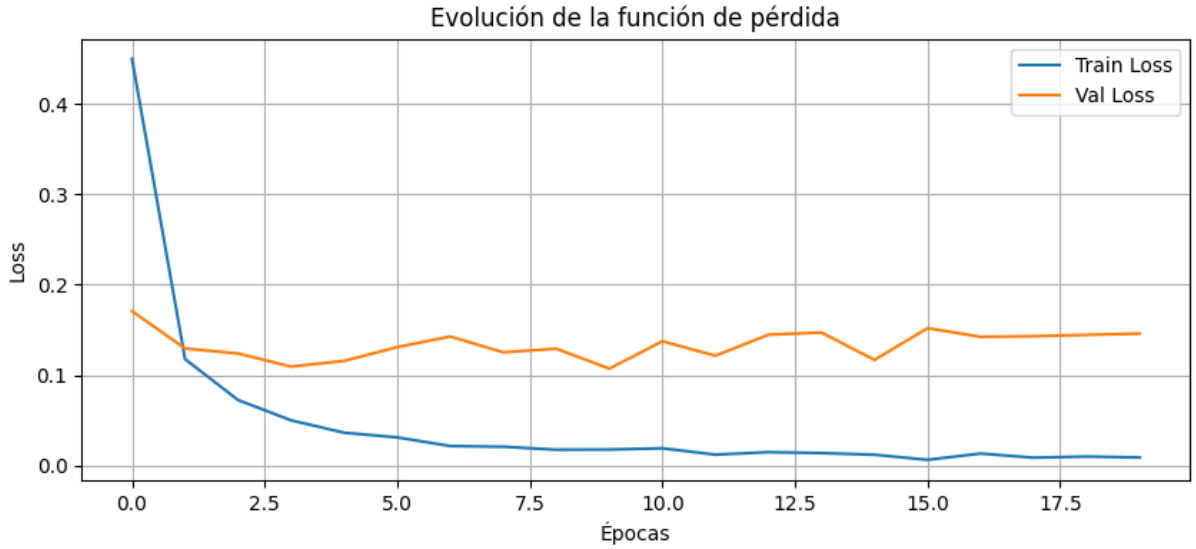

Figura 3: Evolución de la función de pérdida en *MNIST*.

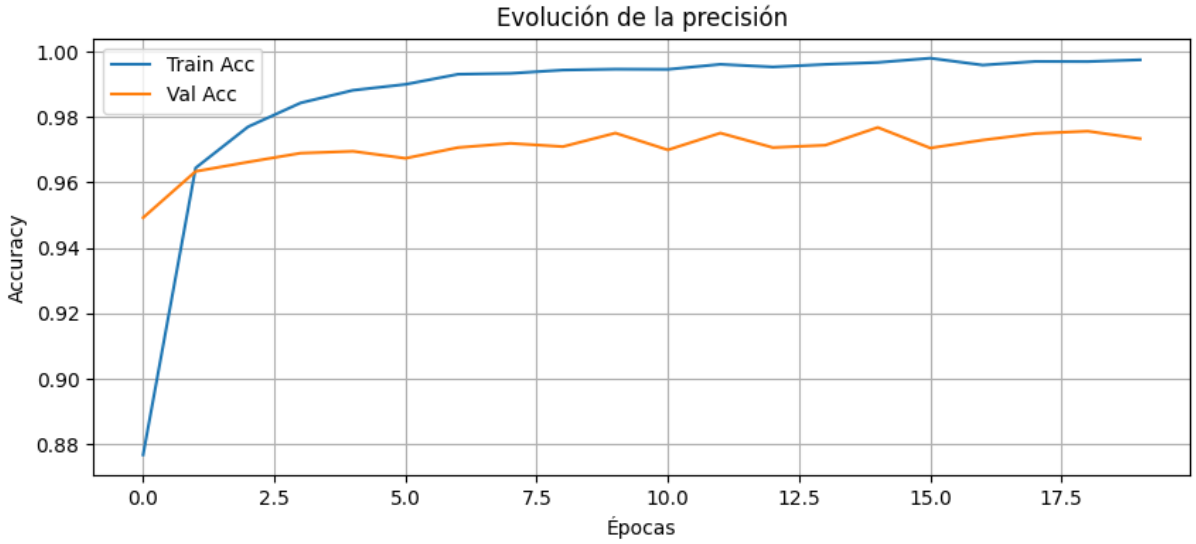

Figura 4: Evolución de la precisión en *MNIST*.

Métrica	**Entrenamiento**	**Validación**	**Prueba**
Precisión (%)	99.70	97.57	97.40
Pérdida	0.0100	0.1445	–

Cuadro 2: Resultados cuantitativos en el conjunto *MNIST*.

La Table 2 muestra un rendimiento notable, con precisión final del 97.40 % en prueba, muy próxima a la de validación. La brecha entre entrenamiento y validación es moderada, lo que refleja un ajuste adecuado sin sobreentrenamiento, a pesar del número limitado de épocas y de la ausencia de *dropout*. La matriz de confusión revela que los errores más frecuentes se concentran en dígitos morfológicamente parecidos, como el *3* y el *5* o el *4* y el *9*, mientras que los dígitos más distintivos (por ejemplo, *0* y *1*) se clasifican con una precisión prácticamente perfecta. Estos resultados demuestran que el motor es capaz de manejar correctamente conjuntos de alta dimensionalidad y converger hacia una solución eficaz sin necesidad de arquitecturas más avanzadas.

4.4. Resultados en Penguins

El conjunto *Penguins* permitió comprobar el comportamiento del motor en un escenario intermedio entre los dos anteriores: número de muestras reducido, pero con atributos de naturaleza continua y escalas distintas. El modelo $[4, 18, 3]$ con activación *tanh* y optimizador Adam mostró un entrenamiento estable y una clara tendencia a la convergencia.

Las curvas de la Fig. 5 y la Fig. 6 indican que la pérdida de validación desciende gradualmente hasta la época 150, momento en el que la precisión alcanza el 100 % en validación. El mecanismo de *early stopping* se activó tras 10 épocas sin mejora adicional, manteniendo la configuración de pesos más generalizable. El proceso de aprendizaje mostró una oscilación mínima entre épocas, reflejando una estabilidad numérica excelente y la eficacia combinada del decaimiento de la tasa de aprendizaje y la regularización L2.

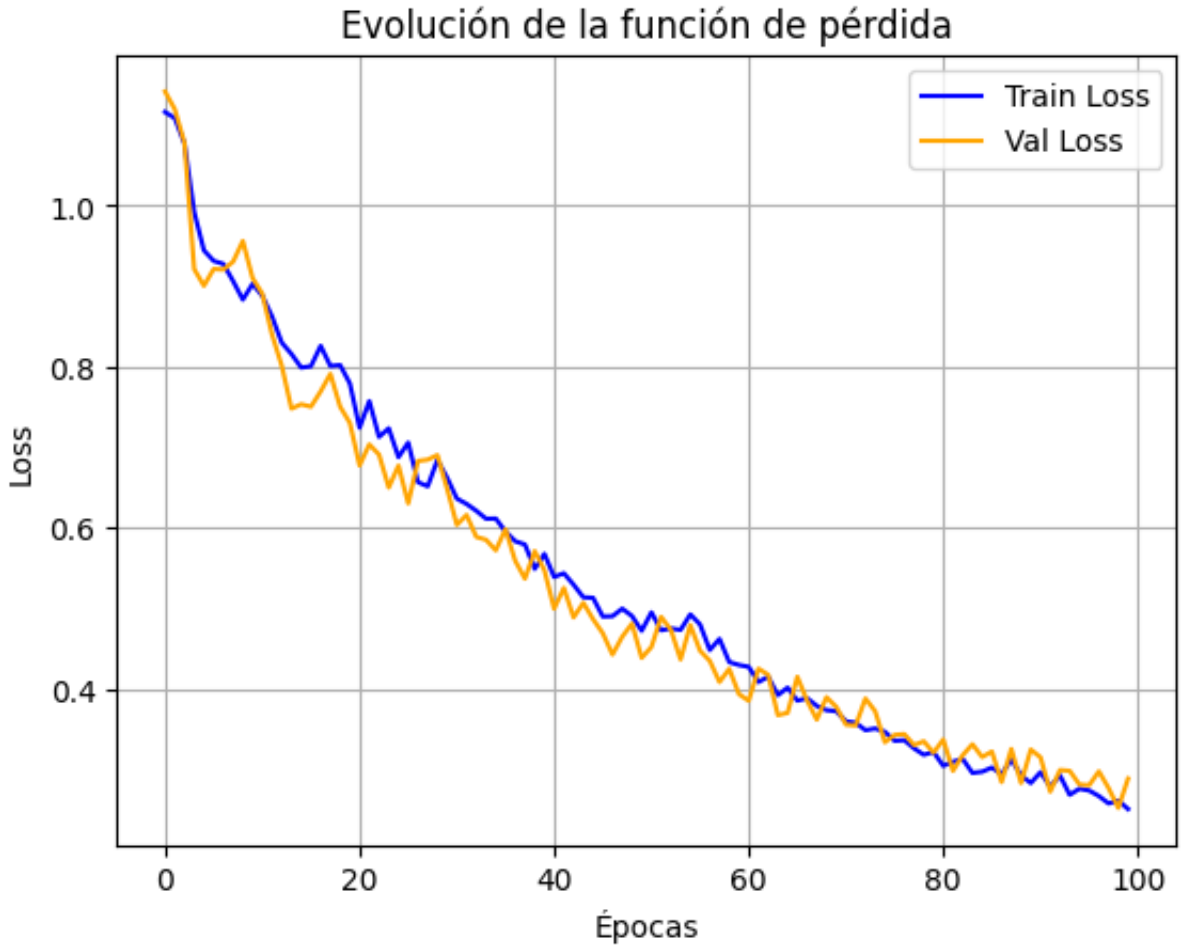

Figura 5: Evolución de la función de pérdida en *Penguins*.

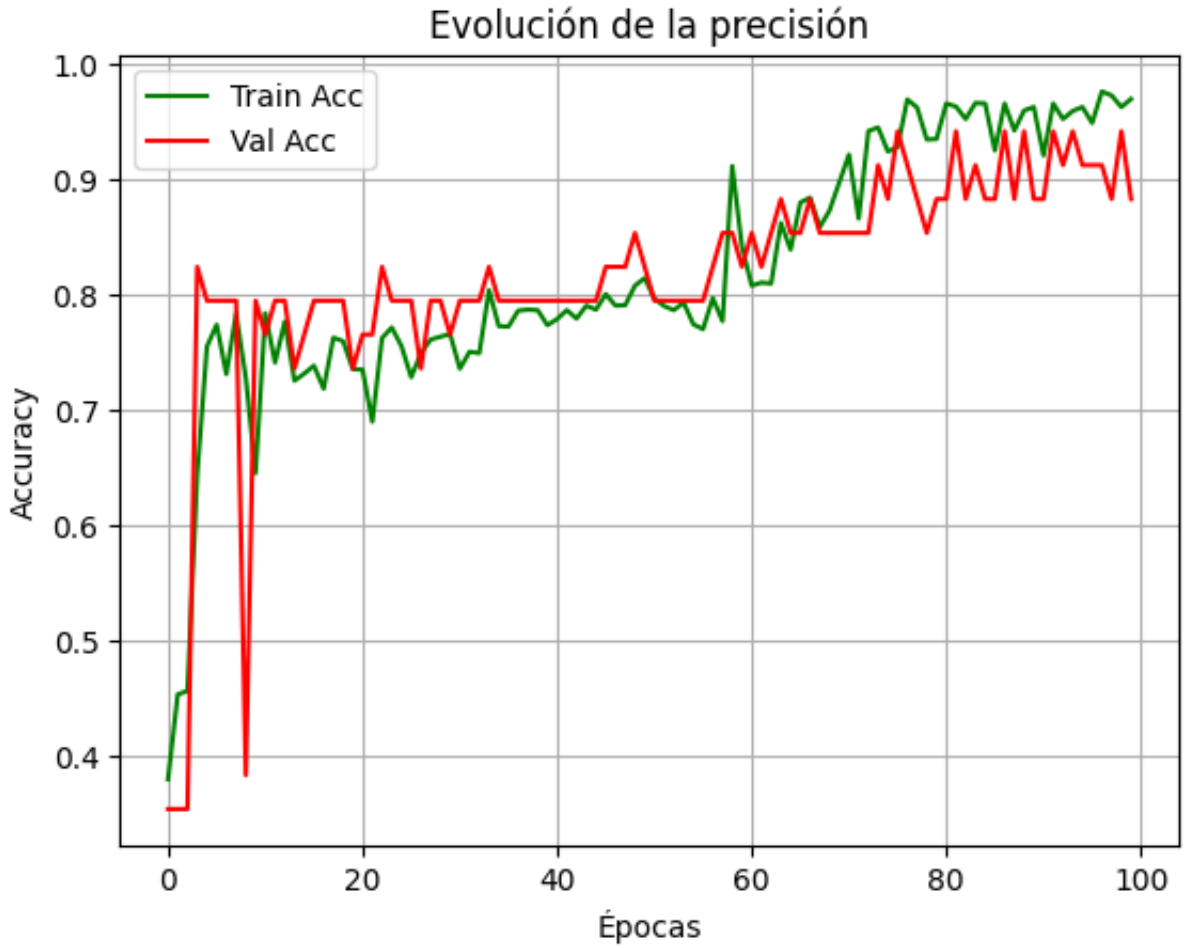

Figura 6: Evolución de la precisión en *Penguins*.

Métrica	**Entrenamiento**	**Validación**	**Prueba**
Precisión (%)	96.88	100.00	97.14
Pérdida	0.1753	0.1452	–

Cuadro 3: Resultados cuantitativos en el conjunto *Penguins*.

Los resultados de la Table 3 reflejan un rendimiento sobresaliente y coherente en todos los conjuntos. El modelo alcanza un 97.14 % de precisión en prueba, con diferencias mínimas entre las métricas de entrenamiento y validación, lo que demuestra un aprendizaje bien regularizado y sin sobreajuste apreciable. Los pocos errores observados se concentran entre las clases *Adelie* y *Chinstrap*, cuyas medidas de pico y aleta presentan rangos solapados. Estos resultados confirman que el motor desarrollado mantiene un desempeño consistente en conjuntos de diversa naturaleza y tamaño, validando la correcta implementación de las técnicas de optimización y regularización.

5. Conclusiones y Trabajo Futuro

El desarrollo de este motor de redes neuronales ha permitido comprender y reproducir en detalle los fundamentos matemáticos y computacionales del aprendizaje profundo, implementando desde cero una arquitectura multicapa funcional sin recurrir a librerías de alto nivel. El proyecto ha demostrado la viabilidad de construir un sistema modular y extensible en `Python`, capaz de entrenar modelos sobre conjuntos de distinta naturaleza y complejidad, desde problemas simples de clasificación lineal hasta tareas de reconocimiento de patrones con miles de características.

Logros principales. Entre los resultados más destacados se encuentran:

- La implementación completa del algoritmo de retropropagación y su correcta integración con el optimizador Adam, confirmada experimentalmente por la convergencia estable de las pérdidas en todos los conjuntos.
- El diseño de un entrenador genérico que soporta mini-lotes, parada temprana, decaimiento de la tasa de aprendizaje y regularización L2, manteniendo una separación clara entre las fases de entrenamiento, validación y prueba.
- La modularidad del código, estructurado en componentes independientes (`NeuralNetwork`, `Trainer`, `OptimizerAdam`, `Losses`, `HyperparameterTuner`, `Utils`), que facilita su mantenimiento, depuración y futura ampliación.
- La correcta generalización del modelo en los tres conjuntos de datos, con precisiones finales del 93 % en *Iris*, 97 % en *MNIST* y 97 % en *Penguins*, sin sobreajuste perceptible.
- La validación empírica del comportamiento esperado de las técnicas de regularización y de los mecanismos de control de aprendizaje.

Dificultades y limitaciones. Durante el desarrollo se afrontaron diversas dificultades, entre ellas:

- La gestión de la estabilidad numérica en operaciones matriciales de gran tamaño, especialmente durante la normalización y la propagación del gradiente en *MNIST*.
- El equilibrio entre velocidad de convergencia y robustez del entrenamiento, que requirió un ajuste fino de la tasa de aprendizaje y de los parámetros del optimizador.
- La ausencia de frameworks externos obligó a implementar manualmente funcionalidades de inicialización, regularización y parada temprana, lo que aumentó la complejidad del código pero aportó una comprensión más profunda del proceso de entrenamiento.

Líneas de trabajo futuro. A partir de la base implementada, se plantean diversas posibilidades de ampliación y mejora:

- Incorporar nuevos métodos de optimización, como *RMSProp*, *SGD con momento*, *Nadam* o *Adagrad*, que permitan comparar su rendimiento frente a Adam.
- Implementar técnicas de búsqueda aleatoria o bayesiana de hiperparámetros, como alternativa más eficiente a la búsqueda en rejilla utilizada actualmente.
- Extender el motor a arquitecturas más complejas, incluyendo redes convolucionales (CNN) y redes recurrentes (RNN), manteniendo la estructura modular existente.
- Incorporar una interfaz gráfica o un módulo de visualización interactiva que facilite la interpretación de las métricas de entrenamiento y validación.
- Explorar el uso de aceleración por GPU mediante bibliotecas como `CuPy` o `PyTorch`, manteniendo la filosofía de implementación propia.

Conclusión general. En conjunto, el trabajo ha cumplido los objetivos propuestos al inicio: ha permitido comprender en profundidad los mecanismos que gobiernan el aprendizaje en redes neuronales artificiales, reforzar la relación entre teoría y práctica, y construir un sistema flexible que sirva como base para futuros proyectos de aprendizaje profundo. La implementación desde cero, aunque más exigente en términos de tiempo y depuración, ha proporcionado una visión completa del flujo de datos y gradientes, poniendo de manifiesto la importancia de una arquitectura modular y bien estructurada en el desarrollo de herramientas de inteligencia artificial.

Bibliografía

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

[2] C. M. Bishop, H. Bishop. *Deep Learning: Foundations and Concepts*. Springer, 2024.

[3] M. Nielsen. *Neural Networks and Deep Learning: How the Backpropagation Algorithm Works*. Online publication, 2019.

[4] M. Mazur. *A Step by Step Backpropagation Example*. Blog article, 2015. Disponible en: https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/.

[5] A. Ng. *Curso sobre Mejora de las redes neuronales profundas: Algoritmos de optimización*. DeepLearning.AI, Módulo 2, Coursera, 2018.

[6] A. Ng. *Curso sobre Redes Neuronales y Aprendizaje Profundo: Redes neuronales profundas*. DeepLearning.AI, Módulo 4, Coursera, 2018.

[7] D. P. Kingma, J. Ba. *Adam: A Method for Stochastic Optimization*. International Conference on Learning Representations (ICLR), 2015.

[8] Y. LeCun, L. Bottou, Y. Bengio, P. Haffner. *Gradient-Based Learning Applied to Document Recognition*. Proceedings of the IEEE, Vol. 86, pp. 2278–2324, 1998.

[9] R. A. Fisher. *The Use of Multiple Measurements in Taxonomic Problems*. Annals of Eugenics, Vol. 7(2), pp. 179–188, 1936.

[10] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, R. Salakhutdinov. *Dropout: A Simple Way to Prevent Neural Networks from Overfitting*. Journal of Machine Learning Research, 15(56):1929–1958, 2014.

[11] Kaggle. *Kaggle Datasets Platform*. Disponible en: https://www.kaggle.com/datasets.