Informe del Proyecto: Motor de Redes Neuronales

Optimización y Heurística

Alberto Rivero Monzón, Amai Suárez Navarro, José Mataix Pérez

7 de noviembre de 2025

Repositorio de GitHub: `https://github.com/albertuti1910/nn-engine-project`

Índice

Resumen

Este trabajo presenta la implementación desde cero de un motor de redes neuronales en Python, utilizando exclusivamente la librería NumPy para las operaciones numéricas. El objetivo principal es profundizar en los mecanismos fundamentales del aprendizaje profundo, derivando e implementando los algoritmos de forward pass, backpropagation y diversos métodos de optimización. Se detallan la arquitectura modular del motor, las funciones de activación y pérdida, y las técnicas de regularización avanzadas como Dropout y L2 (Weight Decay).

El motor se valida a través de una serie de experimentos rigurosos en los datasets clásicos IRIS y MNIST. Los resultados demuestran el correcto funcionamiento del sistema, con una convergencia estable y un rendimiento notable. En el dataset IRIS se supera el objetivo inicial, alcanzando una precisión del 95.65 %. En el dataset MNIST, más complejo, se logra una precisión máxima del 98.20 %, validando la eficacia de las técnicas de regularización implementadas. El informe concluye con un análisis comparativo de los diferentes hiperparámetros estudiados y una discusión sobre los hallazgos obtenidos.

1. Introducción

El campo del aprendizaje profundo (deep learning) ha revolucionado la inteligencia artificial, ofreciendo soluciones potentes a problemas complejos en áreas como la visión por computador, el procesamiento del lenguaje natural y el análisis de datos. Sin embargo, el uso de frameworks de alto nivel como TensorFlow o PyTorch a menudo abstrae los detalles fundamentales que gobiernan el aprendizaje de estos modelos.

El objetivo de este proyecto es construir un motor de redes neuronales funcional desde cero, utilizando únicamente Python y la librería NumPy. Este enfoque nos obliga a comprender y derivar las ecuaciones matemáticas que sustentan el proceso de aprendizaje, desde la propagación hacia adelante (forward pass) hasta el algoritmo de retropropagación del error (backpropagation) y los métodos de optimización basados en gradiente.

A lo largo de este informe, se detallará la arquitectura del motor, las decisiones de diseño y la implementación de sus componentes clave. Posteriormente, se presentarán los resultados obtenidos al validar el motor en dos conocidos problemas de clasificación: el dataset IRIS, como prueba de concepto, y el dataset MNIST de dígitos escritos a mano, para evaluar su rendimiento en un escenario más complejo y realizar un estudio de hiperparámetros.

2. Métodos Teóricos

El núcleo de este proyecto reside en la implementación desde cero de los algoritmas fundamentales que permiten el aprendizaje en una red neuronal. A diferencia de utilizar frameworks de alto nivel, este enfoque requiere una comprensión detallada de las ecuaciones matemáticas que gobiernan el proceso. A continuación, se detallan estos métodos y cómo se relacionan con la estructura del código.

2.1. Deducción de Ecuaciones: Backpropagation

El objetivo del entrenamiento es minimizar una función de pérdida (Coste) L ajustando los parámetros de la red (pesos W y sesgos b). El algoritmo de retropropagación del error, o *backpropagation*, es el método que nos permite calcular de manera eficiente el gradiente de la función de pérdida con respecto a cada uno de estos parámetros ($\frac{\partial L}{\partial W}$ y $\frac{\partial L}{\partial b}$). Este cálculo se basa fundamentalmente en la regla de la cadena del cálculo diferencial.

Consideremos una red neuronal con L capas. La propagación hacia adelante (forward pass) para una capa l se define como:

$$Z^{[l]} = W^{[l]} A^{[l-1]} + b^{[l]} \tag{1}$$

$$A^{[l]} = g^{[l]}(Z^{[l]}) \tag{2}$$

donde $A^{[l-1]}$ es la activación de la capa anterior, $Z^{[l]}$ es la entrada lineal de la capa actual, y $A^{[l]}$ es la salida activada por la función $g^{[l]}$.

El algoritmo de backpropagation comienza en la capa de salida y propaga el error hacia atrás:

1. **Capa de Salida (L):** Primero, se calcula el gradiente de la pérdida con respecto a la entrada lineal de la última capa, $Z^{[L]}$. Esto depende de la función de pérdida y la activación de salida. Por ejemplo, para Cross-Entropy con Softmax, este gradiente es simplemente $\frac{\partial L}{\partial Z^{[L]}} = A^{[L]} - Y$, donde Y son las etiquetas verdaderas.

2. **Gradientes de la Capa L:** Con $\frac{\partial L}{\partial Z^{[L]}}$, podemos calcular los gradientes para los parámetros de esta capa:

$$\frac{\partial L}{\partial W^{[L]}} = \frac{1}{m}\frac{\partial L}{\partial Z^{[L]}}(A^{[L-1]})^T \tag{3}$$

$$\frac{\partial L}{\partial b^{[L]}} = \frac{1}{m}\sum_{i=1}^{m}\frac{\partial L}{\partial Z_i^{[L]}} \tag{4}$$

3. **Propagación a la Capa L-1:** Para continuar hacia atrás, necesitamos el gradiente de la pérdida con respecto a la activación de la capa anterior, $A^{[L-1]}$:

$$\frac{\partial L}{\partial A^{[L-1]}} = (W^{[L]})^T\frac{\partial L}{\partial Z^{[L]}} \tag{5}$$

4. **Capa Oculta (l):** Este proceso se repite para cada capa l de forma recursiva. El gradiente de la pérdida respecto a $Z^{[l]}$ se calcula como:

$$\frac{\partial L}{\partial Z^{[l]}} = \frac{\partial L}{\partial A^{[l]}} \odot g'^{[l]}(Z^{[l]}) \tag{6}$$

donde $\odot$ es la multiplicación elemento a elemento y $g'^{[l]}$ es la derivada de la función de activación. Los gradientes $\frac{\partial L}{\partial W^{[l]}}$ y $\frac{\partial L}{\partial b^{[l]}}$ se calculan de forma análoga a la capa L.

Esta derivación de ecuaciones es la base teórica que permite generalizar el algoritmo. En nuestra implementación, cada clase de capa (Layer) tiene un método backward() que toma $\frac{\partial L}{\partial A^{[l]}}$ como entrada y calcula $\frac{\partial L}{\partial W^{[l]}}$, $\frac{\partial L}{\partial b^{[l]}}$ y $\frac{\partial L}{\partial A^{[l-1]}}$, relacionando directamente las ecuaciones con el código modular. El pseudocódigo genérico de este proceso se encuentra en el Apéndice.

2.2. Métodos de Optimización de la Red

Una vez calculados los gradientes, el optimizador es el encargado de actualizar los parámetros W y b para minimizar la función de pérdida. Se implementaron los siguientes métodos:

- **Descenso de Gradiente Estocástico con Momentum (SGD+Momentum):** En lugar de usar únicamente el gradiente actual, este método introduce un término de "velocidad"(v) que acumula un promedio ponderado de los gradientes pasados. Esto ayuda a suavizar las actualizaciones y a acelerar la convergencia, especialmente en superficies de pérdida con valles estrechos. La actualización es:

$$v = \beta v + (1-\beta)\frac{\partial L}{\partial W} \quad ; \quad W = W - \eta v \tag{7}$$

- **RMSprop (Root Mean Square Propagation):** Este método mantiene un promedio móvil de los cuadrados de los gradientes para cada parámetro. Adapta la tasa de aprendizaje por parámetro, dividiéndola por la raíz cuadrada de este promedio. Esto permite que los parámetros con gradientes pequeños reciban actualizaciones más grandes y viceversa.

- **Adam (Adaptive Moment Estimation):** Es el método implementado como requisito obligatorio y a menudo el más efectivo en la práctica. Combina las ideas de Momentum (primer momento del gradiente) y RMSprop (segundo momento del gradiente). Mantiene dos promedios móviles, m y v, para cada parámetro y los utiliza para realizar una actualización adaptativa y con corrección de sesgo.

Estos algoritmos se implementaron como clases intercambiables (SGD, Adam, RMSprop), cada una con un método update(). Esta abstracción permite que el Trainer pueda usar cualquier optimizador sin cambiar su lógica interna, demostrando un diseño flexible y preparado para futuras extensiones.

3. Implementación

El motor se ha diseñado siguiendo un enfoque modular y orientado a objetos para facilitar su extensión y reutilización, cumpliendo con los requisitos del proyecto.

3.1. Estructura del Motor

El núcleo del proyecto es una librería (src/) que contiene clases para los componentes fundamentales de una red neuronal:

- **NeuralNetwork:** Orquesta las capas y gestiona los procesos de forward pass y predicción. Acepta arquitecturas configurables con un número variable de capas y neuronas.
- **Capas (Layers):** Se implementó una capa densa (Dense) que contiene los pesos (W) y sesgos (b), y la lógica para su inicialización y para la propagación hacia adelante y hacia atrás.
- **Funciones de Activación:** Se implementaron como clases separadas para facilitar su intercambio, incluyendo **Sigmoid**, **ReLU**, **Softmax** y **Tanh**.
- **Funciones de Pérdida:** Se incluyeron **Categorical Cross-Entropy** para problemas de clasificación multiclase y **Mean Squared Error (MSE)** para regresión.
- **Optimizadores (Optimizers):** Siguiendo los requisitos, se implementó una clase base y se desarrollaron los optimizadores **Adam** (obligatorio), **SGD con Momentum** y **RMSprop**.
- **Trainer:** Una clase que gestiona todo el bucle de entrenamiento, incluyendo la gestión de mini-batches, el cálculo de la pérdida, la actualización de parámetros mediante el optimizador seleccionado y la evaluación de métricas.

3.2. Características Implementadas

Además de la estructura base, se implementaron las siguientes funcionalidades requeridas y opcionales:

- **Forward Pass y Backpropagation:** Se implementó el algoritmo completo, incluyendo el cálculo correcto de los gradientes para pesos y sesgos.

- **Entrenamiento por Mini-batches:** El Trainer permite un tamaño de batch ajustable y maneja correctamente los casos en que el dataset no es divisible exactamente.
- **Inicialización de Pesos:** Se incluyeron los métodos **Xavier/Glorot** y **He**, adecuados para diferentes funciones de activación.
- **Técnicas de Regularización:** Para mejorar la generalización y combatir el sobreajuste, se implementaron **Dropout** y **L2 Weight Decay**.
- **Planificadores de Tasa de Aprendizaje (Schedulers):** Se añadieron varias estrategias para ajustar la tasa de aprendizaje durante el entrenamiento, como **Step Decay**, **Exponential Decay** y **Cosine Annealing**.
- **Mecanismo de Early Stopping:** Para detener el entrenamiento cuando el rendimiento en el conjunto de validación deja de mejorar.

4. Experimentos y Resultados

La validación del motor se realizó de manera progresiva. Primero, se utilizó el dataset IRIS como una prueba de concepto para verificar el funcionamiento fundamental del sistema. Posteriormente, se empleó el dataset MNIST para llevar a cabo una evaluación exhaustiva y un estudio detallado de las funcionalidades avanzadas implementadas.

4.1. Demostración con Dataset IRIS: Prueba de Concepto

El propósito principal de usar el dataset IRIS fue realizar una validación rápida y sencilla de los componentes básicos del motor. Dado su tamaño reducido y baja complejidad, es ideal para verificar que los algoritmos de *forward pass* y *backpropagation* funcionan correctamente.

- **Configuración:** Se empleó una red neuronal densa simple (arquitectura 4-16-8-3), activaciones ReLU y Softmax, y el optimizador **Adam**. Para evitar un sobreentrenamiento innecesario en un dataset tan pequeño, se utilizó un mecanismo de **early stopping**.
- **Resultado:** La red convergió de manera estable, alcanzando una **precisión final del 95.65 %** en el conjunto de test, superando con creces el objetivo recomendado del 90 %. Las curvas de aprendizaje (Figura 1) mostraron una clara disminución de la pérdida y un aumento consistente de la precisión, validando el correcto funcionamiento del algoritmo.
- **Análisis de Optimizadores:** Adicionalmente, se realizó una comparación preliminar de los optimizadores **Adam**, **SGD con Momentum** y **RMSprop**, demostrando que todos estaban correctamente implementados y eran funcionales.

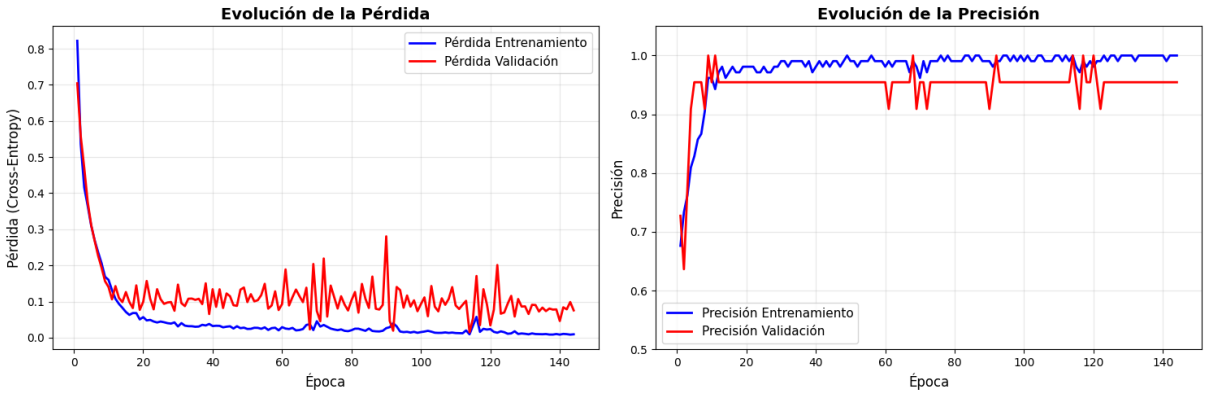

Figura 1: *Curvas de aprendizaje para el dataset IRIS.*

4.2. Análisis Exhaustivo con Dataset MNIST

El dataset MNIST, con su mayor complejidad y tamaño, fue el escenario principal para evaluar el rendimiento del motor y demostrar la eficacia de las funcionalidades avanzadas implementadas.

4.2.1. Línea Base y Superación del Objetivo Mínimo

Con una arquitectura base (784-128-64-10) y el optimizador Adam, se obtuvo una precisión en test del **97.36 %**. Este resultado inicial ya superó ampliamente el valor orientativo del 80 % solicitado en los requisitos, estableciendo una sólida línea base.

4.2.2. Análisis Comparativo de Hiperparámetros y Técnicas

Se diseñó una serie de experimentos controlados para evaluar el impacto de las diferentes funcionalidades opcionales implementadas:

- **Optimización Adicional:** Se comparó el rendimiento de **Adam**, **SGD con Momentum** y **RMSprop**. Como se observa en la Figura 2, todos mostraron una convergencia estable, alcanzando precisiones muy altas (>97 %), con RMSprop obteniendo un ligero máximo de 97.77 % en este experimento.
- **Regularización:** Se estudió el efecto de **Dropout** y **Weight Decay (L2)**. El uso de Dropout demostró ser la estrategia más efectiva. La Figura 3 muestra cómo reduce significativamente el sobreajuste (la diferencia entre las curvas de entrenamiento y validación). Esta mejora en la generalización llevó a obtener la **mejor precisión global del proyecto: 98.20 %**.

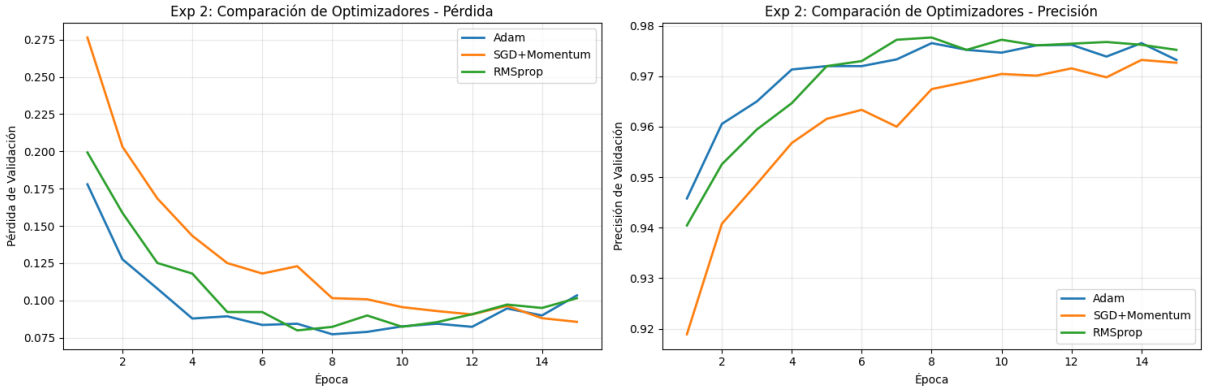

Figura 2: *Comparación de la pérdida de validación para diferentes optimizadores en MNIST.*

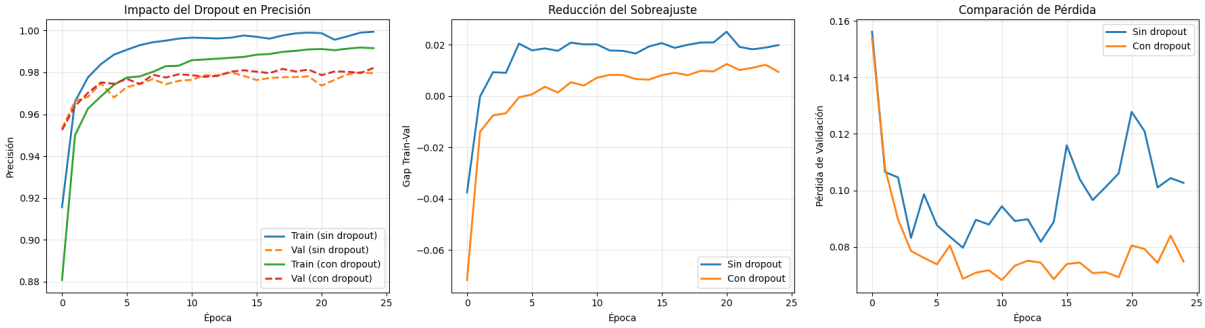

Figura 3: *Impacto de Dropout en la reducción del sobreajuste (gap Train-Val).*

4.3. Resumen Comparativo de MNIST

La Tabla 1 y la Figura 4 resumen los resultados de precisión de todos los experimentos realizados en el dataset MNIST, ordenados de mayor a menor rendimiento.

Cuadro 1: Tabla comparativa de resultados en MNIST.

Configuración	**Precisión en Test (%)**
Exp 3 - Con Dropout	98.20
Exp 3 - Sin Dropout	97.93
Exp 5 - Very Deep (5 capas)	97.88
Exp 5 - Deep (4 capas)	97.87
Exp 5 - Medium (3 capas)	97.85
Exp 5 - Shallow (2 capas)	97.81
Exp 2 - RMSprop	97.77
Exp 2 - Adam	97.69
Exp 2 - SGD+Momentum	97.47
Exp 1 - Baseline	97.36

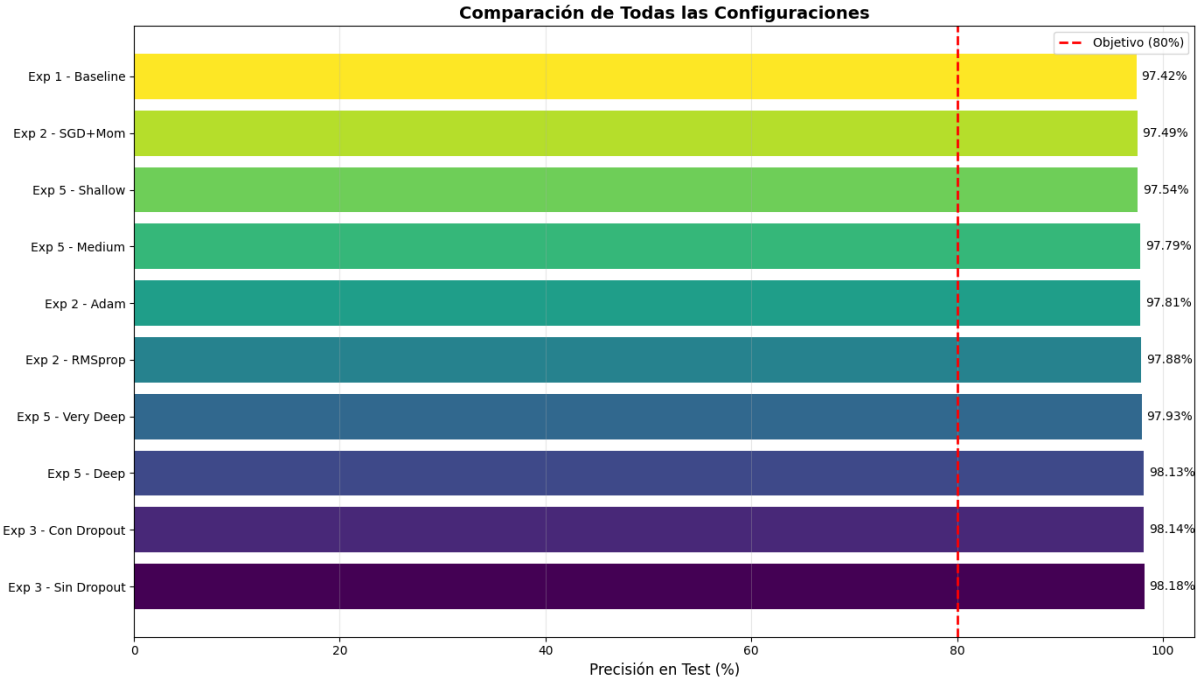

Figura 4: *Gráfico de barras comparando la precisión de todos los experimentos en MNIST.*

4.4. Validación Cruzada del Mejor Modelo (k=5)

Para obtener una estimación más robusta y fiable del rendimiento del modelo, y para asegurar que los buenos resultados no dependen de una única división aleatoria de los datos, se aplicó la técnica de validación cruzada (*k-fold cross-validation*) con **k=5 pliegues**.

Se utilizó la mejor configuración de hiperparámetros encontrada en los experimentos anteriores (la arquitectura profunda con regularización **Dropout**). Para agilizar el proceso de validación, el experimento se ejecutó sobre un subconjunto de 15,000 muestras del dataset de entrenamiento y durante 5 épocas por cada pliegue.

Resultados Obtenidos

Los resultados de precisión en el conjunto de validación para cada uno de los 5 pliegues fueron los siguientes:

- **Pliegue 1:** 95.43 %
- **Pliegue 2:** 95.30 %
- **Pliegue 3:** 94.70 %
- **Pliegue 4:** 95.80 %
- **Pliegue 5:** 95.50 %

Al promediar estos resultados, se obtuvieron las siguientes métricas finales:

- **Precisión Media: 95.35 %**
- **Desviación Estándar: 0.36 %**

Análisis de los Resultados

La precisión media de **95.35 %** confirma que el modelo tiene un rendimiento muy alto y consistente. Aunque esta cifra es ligeramente inferior a la precisión máxima obtenida en la división única de entrenamiento/test, este valor es una estimación **más realista y fiable** del comportamiento del modelo ante datos completamente nuevos.

La desviación estándar de tan solo **0.36 %** es un resultado excelente. Un valor tan bajo indica que el rendimiento del modelo es **muy estable** y no varía significativamente al cambiar los datos de entrenamiento y validación. Esto demuestra que el modelo generaliza bien y no es sensible a la aleatoriedad de la partición de los datos, lo que aumenta la confianza en su capacidad predictiva.

5. Conclusiones

Este proyecto ha culminado con la creación exitosa de un motor de redes neuronales desde sus fundamentos, utilizando únicamente NumPy. A continuación se resumen los aspectos más relevantes del trabajo realizado.

5.1. Aspectos Positivos y Logros Principales

- **Profundidad de Comprensión:** La principal fortaleza del proyecto es la comprensión profunda que se ha adquirido sobre los mecanismos internos del aprendizaje automático. La derivación manual de las ecuaciones de backpropagation y la implementación de los optimizadores han desmitificado la çaja negra"de los frameworks modernos.

- **Rendimiento Sólido y Validación Empírica:** El motor no es solo un ejercicio teórico; ha demostrado ser empíricamente robusto. Los resultados, con precisiones del 95.65 % en IRIS y hasta 98.20 % en MNIST, validan que la implementación de los algoritmos es correcta y efectiva, superando los objetivos mínimos establecidos.

- **Diseño Modular y Extensible:** La arquitectura orientada a objetos ha resultado ser una decisión acertada. Permite intercambiar componentes con facilidad y sienta una base sólida para futuras expansiones, como la adición de nuevos tipos de capas.

5.2. Dificultades Encontradas y Lecciones Aprendidas

- **Depuración de Gradientes:** La principal dificultad fue asegurar la correcta implementación del algoritmo de backpropagation. Pequeños errores en las derivadas o en la aplicación de la regla de la cadena resultaban en un modelo que no aprendía. Esto subraya la importancia de técnicas de verificación de gradientes (como la comprobación numérica) en desarrollos de este tipo.

- **Gestión de Dimensiones (Matrix Shapes):** Mantener la consistencia en las dimensiones de las matrices de pesos, sesgos y activaciones a lo largo de las capas, especialmente durante la retropropagación (que involucra transpuestas), fue un desafío técnico constante que requirió una atención meticulosa.

6. Trabajo Futuro

El diseño modular del motor actual abre la puerta a numerosas extensiones que permitirían abordar problemas más complejos y explorar arquitecturas de vanguardia.

- **Implementación de Batch Normalization:**
  - **Porqué:** Esta técnica normaliza las salidas de las capas intermedias, lo que estabiliza y acelera significativamente el entrenamiento, especialmente en redes profundas. Reduce la dependencia de una inicialización de pesos cuidadosa y permite el uso de tasas de aprendizaje más altas.
- **Añadir Capas Convolucionales (CNNs):**
  - **Porqué:** Las redes neuronales convolucionales son el estándar para tareas de visión por computador. Su capacidad para preservar la estructura espacial de los datos (imágenes) y aprender jerarquías de características (de bordes a objetos complejos) las hace mucho más eficientes y potentes que las redes densas para este tipo de problemas. Sería el siguiente paso natural para probar el motor en datasets como CIFAR-10.
- **Añadir Capas Recurrentes (RNNs):**
  - **Porqué:** Para abordar datos secuenciales, como series temporales o texto, se necesitan arquitecturas con "memoria". Las RNNs, y sus variantes más avanzadas como LSTMs o GRUs, mantienen un estado interno que les permite procesar secuencias de longitud variable y capturar dependencias temporales.

Referencias

[1] Javier Sánchez Pérez (2025). *Apuntes de la asignatura.*

[2] Mykel J. Kochenderfer, Tim A. Wheeler. (2019). *Algorithms for Optimization.* MIT Press

[3] Jorge Nocedal Stephen J. Wright. (2006). *Numerical Optimization.* En *Mathematics Subject Classification (2000): 90B30, 90C11, 90-01, 90-02*

[4] Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep Learning.* MIT Press.

[5] Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, 323(6088), 533-536.

[6] Kingma, D. P., & Ba, J. (2015). Adam: A method for stochastic optimization. En *International Conference on Learning Representations (ICLR).*

[7] Hinton, G. (2012). Neural Networks for Machine Learning, Lecture 6a. Coursera, University of Toronto.

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

[9] Glorot, X., & Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. En *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics (AISTATS).*

[10] He, K., Zhang, X., Ren, S., & Sun, J. (2015). Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. En *Proceedings of the IEEE International Conference on Computer Vision (ICCV).*

[11] Nair, V., & Hinton, G. E. (2010). Rectified Linear Units Improve Restricted Boltzmann Machines. En *Proceedings of the 27th International Conference on Machine Learning (ICML).*

A. Pseudocódigo de Algoritmos Genéricos

A continuación, se presentan los algoritmos genéricos que describen la implementación del motor, sin hacer referencia a código fuente específico.

A.1. Algoritmo 1: Forward Pass

```
FUNCIÓN forward_pass(red, X):
    cache = {}
    A = X  // La activación inicial son los datos de entrada

    PARA cada capa 'l' en red.capas:
        A_anterior = A
        Z, A = capa[l].forward(A_anterior)

        // Guardar valores intermedios para backpropagation
        cache['A' + str(l-1)] = A_anterior
        cache['Z' + str(l)] = Z

    DEVOLVER A_final, cache
```

A.2. Algoritmo 2: Backpropagation

```
FUNCIÓN backward_pass(red, A_final, Y, cache):
    gradientes = {}
    m = Y.shape[1] // Número de ejemplos

    // 1. Iniciar con el gradiente de la capa de salida
    dZ_L = A_final - Y

    // 2. Calcular gradientes para la capa de salida L
    A_anterior = cache['A' + str(L-1)]
    gradientes['dW' + str(L)] = (1/m) * dZ_L @ A_anterior.T
    gradientes['db' + str(L)] = (1/m) * np.sum(dZ_L, axis=1)
    dA_anterior = red.capas[L-1].W.T @ dZ_L

    // 3. Iterar hacia atrás por las capas ocultas
    PARA l en REVERSA(1 hasta L-1):
        dZ = dA_anterior * derivada_activacion(cache['Z' + str(l)])
        A_anterior = cache['A' + str(l-1)]

        gradientes['dW' + str(l)] = (1/m) * dZ @ A_anterior.T
        gradientes['db' + str(l)] = (1/m) * np.sum(dZ, axis=1)
        dA_anterior = red.capas[l-1].W.T @ dZ

    DEVOLVER gradientes
```

A.3. Algoritmo 3: Bucle de Entrenamiento

```
FUNCIÓN train(red, X_train, Y_train, epocas, batch_size, optimizador):

    PARA cada epoca en 1 hasta epocas:
        PARA cada mini_batch (X_b, Y_b) en datos_entrenamiento:

            // 1. Propagación hacia adelante
            A_final, cache = forward_pass(red, X_b)

            // 2. Cálculo de la pérdida (opcional dentro del bucle)
            coste = calcular_perdida(A_final, Y_b)

            // 3. Retropropagación
            gradientes = backward_pass(red, A_final, Y_b, cache)

            // 4. Actualización de parámetros
            red.parametros = optimizador.update(red.parametros, gradientes)

        // Evaluar en conjunto de validación al final de la época
        ...
```