# Implementación del algoritmo Backpropagation en Redes Neuronales con Python

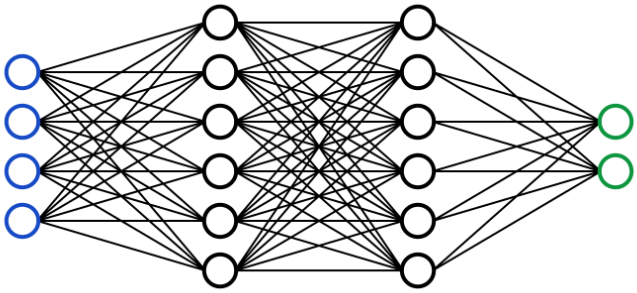
**Daniel Quinteiro Donaghy** daniel.quinteiro101@alu.ulpgc.es
**Carlos Mendoza Eggers** carlos.mendoza103@alu.ulpgc.es

## Resumen

*Este trabajo detalla el desarrollo y la implementación de una red neuronal profunda utilizando el algoritmo Backpropagation en Python, con un enfoque particular en la clasificación y regresión en conjuntos de datos variados. Se ha empleado una arquitectura flexible de red que permite variar el número de capas, sus tamaños y funciones de activación como ReLU, Sigmoid y Softmax. La metodología destaca por su modularidad y organización del código, facilitando la comprensión y futura expansión del proyecto.*

*Los experimentos se centraron en evaluar la eficacia de la red en datasets clásicos como XOR, MNIST, Titanic y en conjuntos de datos de regresión, incluyendo el análisis de la progresión de la diabetes y el rendimiento académico de estudiantes. El enfoque modular permitió ajustar la arquitectura de la red para cada caso específico, y se compararon distintos algoritmos de optimización de descenso de gradiente para mejorar el aprendizaje.*

*Los resultados indican que la red diseñada puede lograr una alta precisión en tareas de clasificación y regresión, aunque con variaciones dependiendo de la complejidad del conjunto de datos. El estudio concluye que la integración de mejoras en la arquitectura de la red y las técnicas de optimización son cruciales para el rendimiento del modelo y establece una base sólida para futuras investigaciones que podrían incluir la exploración de estructuras de red más complejas y técnicas de regularización avanzadas.*



## 1. Introducción

En la entrega anterior de nuestro trabajo, nos enfocamos en la implementación y comprensión del algoritmo de ***Backpropagation*** en el contexto de las redes neuronales, utilizando el lenguaje de programación Python. Este algoritmo, esencial en el aprendizaje automático, permite la actualización efectiva de los pesos y sesgos en una red neuronal, optimizando así su desempeño predictivo.

Nuestro enfoque se centró en una red neuronal simple, diseñada para clasificar entradas binarias que representan letras vocales. Esta red se componía de tres capas: una capa de entrada con 64 neuronas, una capa oculta con 16 neuronas y una capa de salida con 5 neuronas. La arquitectura de la red y el proceso de entrenamiento fueron programados y explicados detalladamente, proporcionando una comprensión profunda del funcionamiento interno de las redes neuronales y del algoritmo de ***Backpropagation***.

Aunque el proyecto proporcionó resultados satisfactorios, su arquitectura e implementación presentaban ciertas limitaciones. El cálculo de los gradientes y el proceso de retropropagación se realizaban de manera casi manual, lo que limitaba mucho la flexibilidad de la arquitectura de la red. Además, la función de activación se limitaba exclusivamente al uso de la sigmoide, y la función de pérdida empleada era únicamente el Error Cuadrático Medio (MSE, por sus siglas en inglés). Otro aspecto a considerar es que no se experimentó con una amplia variedad de ejemplos o con diferentes conjuntos de datos y problemas, lo que podría haber proporcionado una evaluación más robusta de su desempeño.

En esta nueva entrega, nuestro objetivo es expandir y mejorar el trabajo anterior, destacando mejoras y avances en varios aspectos clave de la implementación de redes neuronales. A continuación, se detallan las diferencias notables y las mejoras introducidas respecto al último trabajo:

- **Arquitectura de la red.** Este nuevo trabajo introduce una estructura más modular y flexible, permitiendo la creación de redes con múltiples capas de diferentes tamaños y configuraciones.
- **Retropropagación.** El primer proyecto implementa la retropropagación de manera bastante manual y específica para su arquitectura fija, mientras que en el segundo es mucho más flexible, adaptado para una red con arquitectura variable.
- **Mejoras a nivel de código.** Exhibe una mayor modularidad y organización, facilitando la expansión, el mantenimiento y la comprensión del código
- **Funciones de activación.** Amplía el repertorio de funciones de activación, incluyendo Sigmoid, Softmax y ReLU. Esta diversidad permite adaptar mejor la red a diferentes tipos de problemas y mejorar su capacidad de aprendizaje.
- **Funciones de pérdida.** Añade flexibilidad en la elección de la función de pérdida, ofreciendo tanto MSE como CrossEntropy, lo que es particularmente beneficioso para tareas de clasificación.
- **Problemas y Conjuntos de Datos:** En esta entrega, exploramos diversas arquitecturas aplicadas a una amplia variedad de conjuntos de datos, abordando tanto problemas de Regresión como de Clasificación.
- **Algoritmos de optimización.** Se han incorporado distintos métodos de descenso de gradiente: SGD con Momentum, RMSprop y Adam. Cada uno de estos algoritmos tiene sus propias ventajas y es seleccionado en función de las necesidades específicas del problema y la naturaleza del conjunto de datos.

## 2. Métodos y Conjuntos de Datos

### 2.1 Métodos

Una red neuronal se compone de varias capas de neuronas. Cada neurona en una capa está conectada a todas las neuronas de la capa anterior y posterior (en el caso de las capas densas). Nuestra arquitectura permite la incorporación de múltiples capas densas, así como diferentes funciones de activación (ReLU, Sigmoid, Softmax).

El entrenamiento de una red neuronal sigue un proceso metodológico, que incluye una serie de pasos estructurados, aunque estos pueden variar ligeramente en función de la arquitectura específica de la red y el problema en cuestión. El procedimiento general que sigue una red neuronal para configurar y ajustar sus pesos se caracteriza por dos fases principales: la propagación hacia adelante o feedforward, donde se calculan las salidas de la red, y la retropropagación o backpropagation, donde se ajustan los pesos en función del error calculado.

A continuación, explicaremos más en detalle el flujo que se sigue a la hora de entrenar una red:

#### 2.1.1 Proceso de Entrenamiento de la Red Neuronal

1. **Inicialización de la Red:**
   - Crear la red con las capas especificadas, incluyendo las capas densas y las funciones de activación.
2. **Inicio del Entrenamiento:**
   - Comenzar el proceso de entrenamiento iterativo a través de varias épocas.
3. **Feedforward:**
   - Pasar los datos de entrada a través de cada capa de la red.
   - En cada capa, calcular la salida lineal y aplicar la función de activación.
   - Almacenar las salidas de las activaciones para su uso en la retropropagación.
4. **Cálculo de Pérdida:**
   - Evaluar el rendimiento de la red utilizando una función de pérdida (MSE o CrossEntropy), comparando la salida de la red con los valores objetivos reales.
5. **Backpropagation:**
   - Calcular el gradiente de la función de pérdida.
   - Propagar este gradiente hacia atrás a través de la red, actualizando los pesos y sesgos de cada capa.
6. **Actualización de Pesos y Sesgos:**
   - Utilizar los gradientes calculados en la retropropagación para ajustar los pesos y sesgos utilizando algún algoritmo de descenso por el gradiente.
7. **Evaluación y Early Stopping (opcional):**

- Evaluar el modelo en un conjunto de validación.
- Aplicar early stopping si el rendimiento no mejora durante un número determinado de épocas.

8. **Repetición:**
   - Repetir los pasos 3 a 7 para cada época hasta completar todas las épocas o hasta que se active el early stopping. Esto ayuda a reducir el overfitting y optimizar el coste computacional del entrenamiento.
9. **Fin del Entrenamiento:**
   - Concluir el entrenamiento y guardar los parámetros del modelo entrenado.
10. **Evaluación del Modelo:**
    - Evaluar el rendimiento del modelo con un conjunto de datos de prueba.

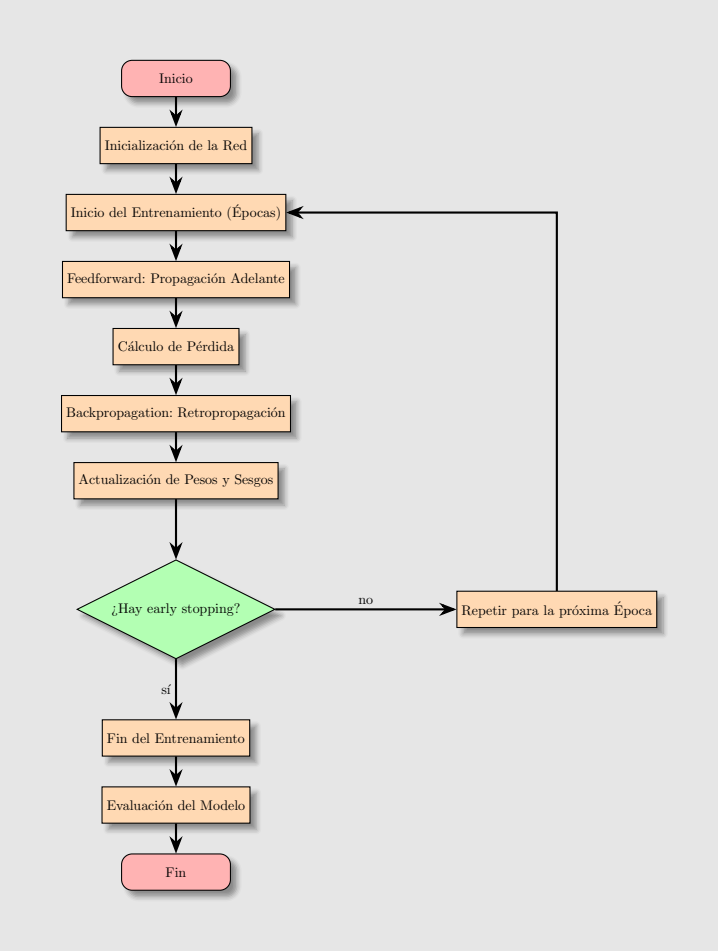



Figura 1: Diagrama de flujo del proceso de entrenamiento.

### 2.1.2 **FeedForward**

El feedforward es el proceso de pasar una entrada a través de las capas de la red para obtener una salida. En cada capa, la entrada se transforma usando los pesos y sesgos de la capa, seguido de una función de activación. La salida de una capa se convierte en la entrada de la siguiente, hasta alcanzar la última capa.

Las funciones de activación son cruciales en las redes neuronales, especialmente cuando se acoplan con capas densas, ya que introducen no linealidades esenciales para que la red aprenda patrones complejos.

#### Capa Densa

Dada una capa densa con entradas $\mathbf{x}$, pesos $\mathbf{W}$, y sesgos $\mathbf{b}$, la salida $\mathbf{y}$ se calcula de la siguiente manera:

$$\mathbf{y} = \mathbf{x}\mathbf{W} + \mathbf{b} \tag{1}$$

Donde:

- $\mathbf{x}$ es el vector de entrada o la matriz de entrada en caso de múltiples entradas.
- $\mathbf{W}$ es la matriz de pesos de la capa.
- $\mathbf{b}$ es el vector de sesgos.
- $\mathbf{y}$ es la salida calculada de la capa.

#### Ecuaciones de Métodos Forward para Funciones de Activación

**ReLU (Rectified Linear Unit)** La función de activación ReLU, aplicada a la salida $y$ de una capa densa, se define como:

$$\text{ReLU}(y) = \text{máx}(0, y) \tag{2}$$

Donde $y$ es la salida de la capa densa.

La ReLU se ha convertido en la función de activación por defecto para muchas redes neuronales, especialmente en las capas ocultas, porque ayuda a resolver el problema del gradiente que desvanece que es común con las activaciones sigmoidales en redes profundas. Al ser una función de partes, permite que el gradiente pase sin cambios cuando el valor de entrada es positivo, lo que facilita el aprendizaje rápido y efectivo. Además, su simplicidad computacional es una ventaja significativa en términos de eficiencia.

**Sigmoide** La función de activación Sigmoide, aplicada a la salida $y$ de una capa densa, se define como:

$$\sigma(y) = \frac{1}{1 + e^{-y}} \tag{3}$$

Donde $y$ es la salida de la capa densa y $e$ es la base del logaritmo natural.

La función Sigmoide es históricamente importante ya que modela la activación como una función que varía suavemente entre 0 y 1, lo que la hace una buena aproximación inicial para imitar el disparo de las neuronas en el cerebro. Se utiliza comúnmente en la capa de salida de un clasificador binario, donde necesitamos una probabilidad como salida. Sin embargo, no es adecuada para capas ocultas en redes profundas debido al problema del gradiente que desvanece, ya que los gradientes pueden ser extremadamente pequeños y pueden detener el aprendizaje a medida que avanza la retropropagación.

**Softmax** La función de activación Softmax, aplicada a un vector $y \in \mathbb{R}^n$ que es la salida de una capa densa, se define como:

$$\text{Softmax}(y)_i = \frac{e^{y_i}}{\sum_{j=1}^{n} e^{y_j}} \tag{4}$$

Donde $y_i$ es la i-ésima entrada del vector de salida $y$.

Softmax es una extensión de la función Sigmoide que se utiliza en la capa de salida de las redes neuronales para clasificación multiclase. Lo que la hace única es que puede manejar múltiples clases y garantiza que la suma de las valores predichos de todas las clases sea 1.

### 2.1.3 Backpropagation

La retropropagación es un método para actualizar los pesos y sesgos de la red. Se basa en el cálculo del gradiente de la función de pérdida con respecto a cada peso y sesgo, utilizando la regla de la cadena para propagar este gradiente de vuelta a través de la red.

Dependiendo de la función de pérdida utilizada (MSE para regresión, CrossEntropy para clasificación), el gradiente se calcula con respecto a la salida de la red. Este gradiente es luego propagado hacia atrás.

En cada capa, el gradiente se utiliza para actualizar los pesos y sesgos, generalmente con un factor conocido como tasa de aprendizaje. Esto asegura que la red aprenda de los errores cometidos en las predicciones anteriores.

Vamos a desglosar el proceso de retropropagación en las capas DenseLayer y las funciones de activación, explicando su orden.

#### Funciones de Pérdida

Primero, se calcula la salida de la red usando la fase de feedforward. Una vez se ha calculado la salida de la red, se evalua con alguna de estas funciones de pérdida.

#### Mean Squared Error (MSE):

$$\text{MSE} = \frac{1}{N} \sum (y_{\text{pred}} - y_{\text{real}})^2 \tag{5}$$

**Cross-Entropy (Binary Classification with Sigmoid):**

$$\text{CrossEntropy} = -\frac{1}{N}\sum \left(y_{\text{real}} \log(y_{\text{pred}}) + (1 - y_{\text{real}}) \log(1 - y_{\text{pred}})\right) \tag{6}$$

**Cross-Entropy (Multiclass Classification with Softmax):**

$$\text{CrossEntropy} = -\sum y_{\text{real}} \log(y_{\text{pred}}) \tag{7}$$

A continuación, se calcula el gradiente de la función de pérdida con respecto a la salida de la última capa. Este gradiente es crucial durante la retropropagación, ya que indica cómo los pesos de la red deben ajustarse para minimizar el error en las predicciones.

**Mean Squared Error (MSE)**

La derivada del Mean Squared Error con respecto a la salida predicha es:

$$\frac{\partial \text{MSE}}{\partial y_{\text{pred}}} = 2 \times \frac{y_{\text{pred}} - y_{\text{real}}}{N} \tag{8}$$

**Cross-Entropy (Clasificación para problemas biclásicos)**

La derivada de la Cross-Entropy para clasificación binaria es:

$$\frac{\partial \text{CrossEntropy}}{\partial y_{\text{pred}}} = -\left(\frac{y_{\text{real}}}{y_{\text{pred}}} - \frac{1 - y_{\text{real}}}{1 - y_{\text{pred}}}\right) \tag{9}$$

**Cross-Entropy (Clasificación para problemas multiclase)**

La derivada de la Cross-Entropy para clasificación multiclase es:

$$\frac{\partial \text{CrossEntropy}}{\partial y_{\text{pred}}} = y_{\text{pred}} - y_{\text{real}} \tag{10}$$

Este gradiente también se propaga hacia atrás a través de las funciones de activación.

**Funciones de Activación**

**ReLU**

$$\text{ReLU}'(y) = \begin{cases} 0 & \text{if } y \leq 0 \\ 1 & \text{if } y > 0 \end{cases} \tag{11}$$

**Sigmoid**

$$\sigma'(y) = \sigma(y) \cdot (1 - \sigma(y)) \tag{12}$$

### Softmax

La softmax aquí no tiene ecuación de retropropagación. Su combinación con la entropía cruzada como función de pérdida simplifica los cálculos del gradiente durante la retropropagación, lo que facilita la actualización de los pesos en el entrenamiento. Posteriormente en la Implementación, se dará una explicación más elaborada.

### Capa Densa

Finalmente, se calculan los gradientes con respecto a los pesos y el sesgo en las capas densas y se actualizan según el algoritmo de optimización elegido.

En esta capa tenemos la opción de escoger entre diversos optimizadores como Adam, RMSprop, o SGD con momentum para actualizar los pesos y sesgos.

### Gradiente de los Pesos

El gradiente de los pesos $\mathbf{W}$ en una capa densa se calcula como el producto de la transpuesta de la entrada $\mathbf{X}^\top$ y el gradiente del error que se ha propagado hasta la salida de la capa actual $\frac{\partial L}{\partial \mathbf{o}}$:

$$\frac{\partial L}{\partial \mathbf{W}} = \mathbf{X}^\top \frac{\partial L}{\partial \mathbf{o}} \tag{13}$$

### Gradiente de los Sesgos

De manera similar, el gradiente del sesgo $\mathbf{b}$ se obtiene sumando el gradiente del error que se ha propagado hasta la salida de la capa actual $\frac{\partial L}{\partial \mathbf{o}}$, sobre todos los ejemplos del batch:

$$\frac{\partial L}{\partial \mathbf{b}} = \sum \frac{\partial L}{\partial \mathbf{o}} \tag{14}$$

Donde:

- $\mathbf{X}$ es el vector de entrada o la matriz de entrada en caso de múltiples entradas.
- $\mathbf{W}$ es la matriz de pesos de la capa.
- $\mathbf{b}$ es el vector de sesgos.
- $\mathbf{y}$ es la salida calculada de la capa.
- $\mathbf{L}$ es la función de pérdida.

### Fórmulas de actualización de pesos

A continuación se presentan las fórmulas utilizadas para la actualización de pesos en una capa densa utilizando diferentes algoritmos de optimización:

- **SGD** (Stochastic Gradient Descent): Es el algoritmo de optimización más simple y uno de los más utilizados. Actualiza los pesos en la dirección opuesta al gradiente de

la función de pérdida. Es efectivo pero puede ser lento y menos estable.

$$\mathbf{W} := \mathbf{W} - \eta \cdot \frac{\partial L}{\partial \mathbf{W}} \tag{15}$$

$$\mathbf{b} := \mathbf{b} - \eta \cdot \frac{\partial L}{\partial \mathbf{b}} \tag{16}$$

- **SGD con Momentum**: Este método ayuda a acelerar el SGD en la dirección correcta y amortigua las oscilaciones. Utiliza una fracción del gradiente de la actualización anterior para influir en la actualización actual, lo que permite superar mínimos locales y puntos de silla.

$$\mathbf{v} := \mu \cdot \mathbf{v} - \eta \cdot \frac{\partial L}{\partial \mathbf{W}} \tag{17}$$

$$\mathbf{W} := \mathbf{W} + \mathbf{v} \tag{18}$$

- **RMSprop**: RMSprop modifica la tasa de aprendizaje para cada parámetro, dividiéndola por un promedio móvil del cuadrado de los gradientes, lo que permite una convergencia más rápida y eficiente en entrenamientos con gradientes muy variables.

$$\mathbf{s} := \rho \cdot \mathbf{s} + (1 - \rho) \cdot \left( \frac{\partial L}{\partial \mathbf{W}} \right)^2 \tag{19}$$

$$\mathbf{W} := \mathbf{W} - \frac{\eta}{\sqrt{\mathbf{s} + \epsilon}} \cdot \frac{\partial L}{\partial \mathbf{W}} \tag{20}$$

- **Adam** (Adaptive Moment Estimation): Adam combina las ventajas de dos extensiones de SGD, el Momentum y el escalado adaptativo de la tasa de aprendizaje por RMSprop. Mantiene estimaciones del primer y segundo momento de los gradientes para ajustar la tasa de aprendizaje de cada parámetro. Esta combinación lo convierte en uno de los optimizadores más efectivos para problemas de aprendizaje profundo.

$$\mathbf{m} := \beta_1 \cdot \mathbf{m} + (1 - \beta_1) \cdot \frac{\partial L}{\partial \mathbf{W}} \tag{21}$$

$$\mathbf{v} := \beta_2 \cdot \mathbf{v} + (1 - \beta_2) \cdot \left( \frac{\partial L}{\partial \mathbf{W}} \right)^2 \tag{22}$$

$$\hat{\mathbf{m}} := \frac{\mathbf{m}}{1 - \beta_1^t} \tag{23}$$

$$\hat{\mathbf{v}} := \frac{\mathbf{v}}{1 - \beta_2^t} \tag{24}$$

$$\mathbf{W} := \mathbf{W} - \frac{\eta}{\sqrt{\hat{\mathbf{v}} + \epsilon}} \cdot \hat{\mathbf{m}} \tag{25}$$

### 2.2 Conjuntos de Datos

Para realizar las pruebas de nuestra implementación, decidimos escoger 3 datasets relacionados con problemas de Clasificación y 2 datasets que tuviesen información enfocada a resolver problemas de Regresión.

- ♢ **Datasets empleados en Clasificación.**
  - ♠ **XOR.** Es un ejemplo clásico en el estudio de Redes Neuronales. Este dataset se utiliza para ilustrar el problema XOR (exclusive OR), que es un problema de clasificación binaria.
    - **Estructura de Datos:**
      - **Entradas (X):** Consiste en cuatro pares de valores binarios $[0, 0]$, $[0, 1]$, $[1, 0]$, y $[1, 1]$. Cada par representa una combinación única de dos características binarias.
      - **Salidas (y):** El conjunto de salidas es [0], [1], [1], [0], respectivamente, para cada par de entrada. El valor de salida es 1 si solo uno de los valores de entrada es 1 (comportamiento XOR), y 0 en otros casos.
    - **Objetivo del Dataset:** El objetivo es clasificar correctamente cada par de entrada en una de dos categorías: 0 o 1, basándose en la operación lógica XOR.
    - **Ejemplos de Samples:**
      - Entrada: $[0, 0]$, Salida esperada: 0
      - Entrada: $[0, 1]$, Salida esperada: 1
      - Entrada: $[1, 0]$, Salida esperada: 1
      - Entrada: $[1, 1]$, Salida esperada: 0
    - **Análisis del Dataset**
      - **Número y Variedad de los Datos:** El dataset es simple y pequeño, con solo 4 puntos de datos. Representa un desafío interesante para ciertos tipos de modelos de aprendizaje automático, ya que requiere una separación no lineal del espacio.
      - **Clases o Resultados Esperados:** Hay dos clases en este conjunto de datos: 0 y 1. La distribución de clases es uniforme.
  - ♠ **MNIST.(2)** Es un conjunto de datos fundamental en el campo del aprendizaje automático para la clasificación de imágenes. Consiste en imágenes en escala de grises de dígitos escritos a mano, cada una de 28x28 píxeles.
    - **Estructura de Datos:**
      - **Entradas (X):** Cada fila representa una imagen de un dígito, desglosada en 784 valores de píxeles (28x28), donde cada valor de píxel es un entero entre 0 (blanco) y 255 (negro).
      - **Salidas (y):** La primera columna de cada fila indica la etiqueta de clase del dígito, que varía de 0 a 9.
    - **Objetivo del Dataset:** El objetivo es clasificar cada imagen en la categoría numérica correspondiente que representa el dígito.

- **Ejemplos de Samples:**

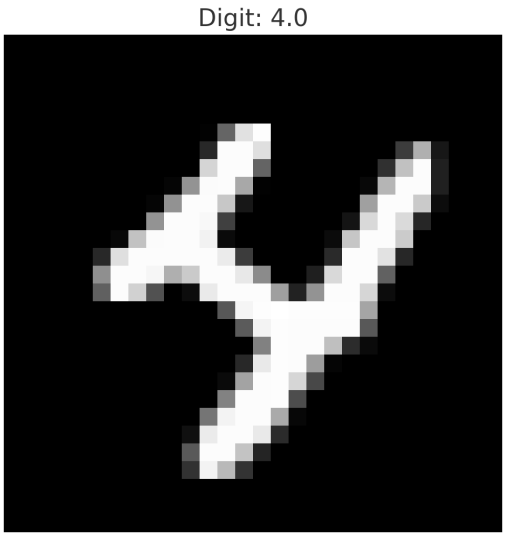



Figura 2: Ejemplo de 4 en el MNIST.

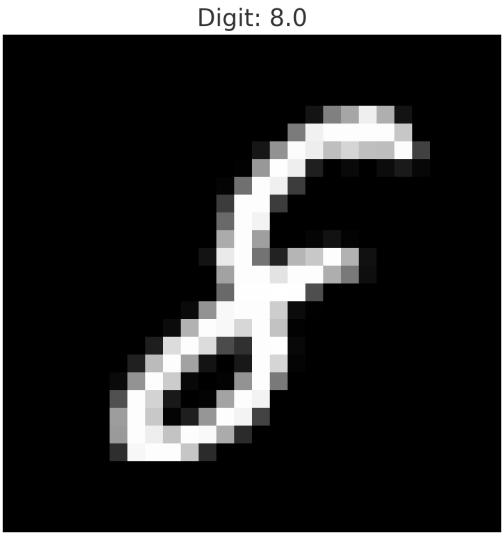



Figura 3: Ejemplo de 8 en el MNIST.

Si cogemos cualquier vector de entrada de 784 valores, y los mostramos como una matriz de 28x28, el resultado es un dígito como los que podemos apreciar en las figuras **2 y 3.**

- **Análisis del Dataset:**
  - **Número y Variedad de los Datos:** El dataset MNIST es extenso, proporcionando un gran número de ejemplos para cada una de las 10 categorías de dígitos (0-9), lo que lo hace ideal para entrenar modelos robustos de clasificación de imágenes.
  - **Clases o Resultados Esperados:** Hay 10 clases en este conjunto de datos, correspondientes a los 10 dígitos. Cada clase tiene una distribución relativamente equilibrada de ejemplos.

♠ **Titanic.(3)** Este dataset proviene de una competición en Kaggle y es ampliamente utilizado para problemas de clasificación en el aprendizaje automático. Contiene datos de los pasajeros del Titanic y se utiliza para predecir la supervivencia de los pasajeros durante su trágico hundimiento.

- **Estructura de Datos:**
  - **Entradas(X)** Incluye diversas características como 'PassengerId', 'Pclass' (clase del pasajero), 'Name', 'Sex', 'Age', 'SibSp' (número de hermanos/esposas a bordo), 'Parch' (número de padres/hijos a bordo), 'Ticket', 'Fare' (tarifa), 'Cabin' y 'Embarked' (puerto de embarque).
  - **Salidas(y)** La columna 'Survived' es la variable objetivo, que indica si el pasajero sobrevivió (1) o no (0).
- **Objetivo del Dataset:** El objetivo principal es predecir la supervivencia de los pasajeros basándose en sus características.

- **Ejemplos de Samples:**
  - Un pasajero de 22 años en tercera clase, hombre, sin parientes a bordo, con una tarifa de 7.25, no sobrevivió.
  - Una pasajera de 38 años en primera clase, mujer, con un cónyuge a bordo y una tarifa de 71.28, sobrevivió.
- **Análisis del Dataset:**
  - **Número y Variedad de los Datos:** El dataset contiene una mezcla de características numéricas y categóricas, con un total de 891 registros en el conjunto de entrenamiento.
  - **Clases o Resultados Esperados:** La variable objetivo 'Survived' es binaria. La proporción de supervivientes frente a no supervivientes ofrece una perspectiva interesante para el análisis y la modelización.

♢ **Datasets empleados en Regresión.**

♣ **scikit-learn Diabetes.(4)** Este conjunto de datos es utilizado para problemas de regresión y consiste en diez variables basales de 442 pacientes con diabetes, así como una medida de la progresión de la enfermedad un año después del inicio del estudio.

- **Estructura de Datos:**
  - **Entradas (X):** Las variables basales incluyen edad, sexo, índice de masa corporal, presión arterial promedio y seis mediciones de suero sanguíneo.
  - **Salida (y):** La medida de la progresión de la enfermedad, que se ha cuantificado utilizando una variable continua.
- **Objetivo del Dataset:** El objetivo es predecir la progresión de la enfermedad un año después de la línea de base en función de las variables basales.
- **Ejemplos de Samples:**
  - Un paciente podría tener un perfil de entrada con valores normalizados para BMI, presión arterial, y niveles de suero, y una progresión de enfermedad de 151.
  - Otro paciente podría presentar diferentes valores basales y una progresión de enfermedad de 75.
- **Análisis del Dataset:**
  - **Número y Variedad de los Datos:** El conjunto de datos ofrece una muestra de tamaño moderado con una buena cantidad de variables independientes para el análisis de regresión.
  - **Resultado Esperado:** Se espera obtener un modelo que pueda predecir la variable de progresión de la enfermedad.

♣ **Student Performance.(5)** Este conjunto de datos se utiliza para problemas de regresión y está diseñado para examinar los factores que influyen en el rendimiento académico de los estudiantes.

- **Estructura de Datos:**
  - **Entradas (Predictores):** Incluye horas estudiadas, puntuaciones previas, participación en actividades extracurriculares (Sí/No), horas de

sueño y número de cuestionarios de muestra practicados por el estudiante.

- ◇ **Salida (Variable Objetivo):** El índice de rendimiento representa el rendimiento académico del estudiante, con valores que van de 10 a 100, donde valores más altos indican un mejor rendimiento.

- ○ **Objetivo del Dataset:** Explorar el impacto de las horas de estudio, las puntuaciones previas, las actividades extracurriculares, las horas de sueño y la práctica de cuestionarios de muestra en el rendimiento de los estudiantes.

- ○ **Ejemplos de Samples:**

  - ◇ Un estudiante que ha estudiado 7 horas, con una puntuación previa de 99, participa en actividades extracurriculares, duerme 9 horas y ha practicado 1 cuestionario de muestra tiene un índice de rendimiento de 91.

  - ◇ Un estudiante que ha estudiado 4 horas, con una puntuación previa de 82, no participa en actividades extracurriculares, duerme 4 horas y ha practicado 2 cuestionarios de muestra tiene un índice de rendimiento de 65.

- ○ **Análisis del Dataset:**

  - ◇ **Número y Variedad de los Datos:** El conjunto de datos consta de 10,000 registros de estudiantes, cada uno con información sobre diversos predictores y un índice de rendimiento, lo que permite realizar análisis detallados sobre los factores que contribuyen al rendimiento académico.

  - ◇ **Resultado Esperado:** Se espera obtener un modelo que pueda predecir con precisión el índice de rendimiento de los estudiantes utilizando las variables proporcionadas.

## 3. Detalles de Implementación

El proyecto está enteramente programado dentro de un módulo llamado neural_network, en el que podemos encontrar las diversas clases que componen este pequeño framework.

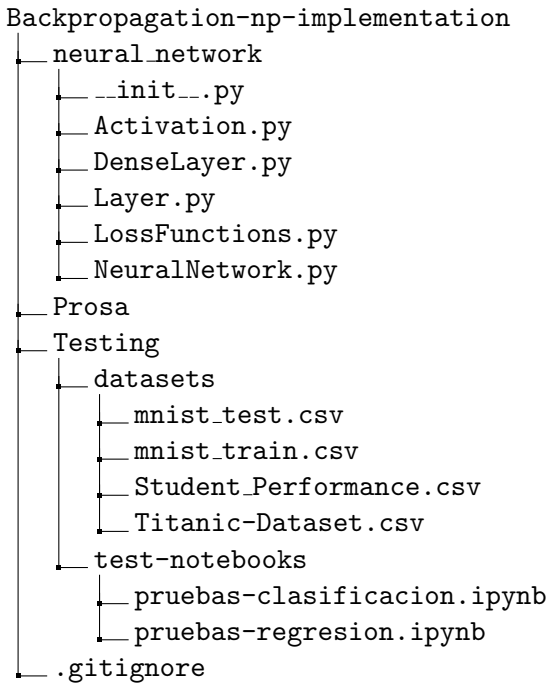



Figura 4: Estructura del proyecto.

En la Figura 4 se muestra la estructura del proyecto. En este apartado pasaremos a hablar sobre todo del módulo neural_network, nombrado anteriormente. Los detalles sobre los archivos situados dentro del directorio Testing se darán a conocer más a delante en el apartado de Experimentos y Pruebas.

- **NeuralNetwork.py.** En este archivo está definida la clase **NeuralNetwork**, pilar fundamental de nuestro trabajo y la cual define una red neuronal. Esta se compone de una lista de capas, donde cada capa puede ser una capa densa o una capa de activación. Veamos más a fondo los detalles de su implementación:

  - **Inicialización (__init__).**La función de inicialización define la estructura de la red neuronal. Toma un número variable de capas (*layers) y las almacena en una lista (self.layers). Si la última capa es una capa de activación (Sigmoid o Softmax), se almacena en self.last_activation_layer para su uso en el cálculo de la función de pérdida.
  - **Predicción (predict).** La función predict toma un conjunto de datos de entrada y realiza una propagación hacia adelante a través de todas las capas de la red, utilizando la función forward de cada capa. El resultado final es la predicción de la red para la entrada dada.

- **Entrenamiento (train).**La función train se encarga de entrenar la red utilizando un conjunto de datos de entrada X, las etiquetas correspondientes y, un número de épocas, una tasa de aprendizaje, y una función de pérdida especificada ('mse' o 'crossentropy'). Además, incluye la posibilidad de detener el entrenamiento antes de tiempo si la pérdida en un conjunto de validación no mejora durante un número dado de épocas (early stopping con patience). Se hace una descripción exhaustiva de este procedimiento en la **sección 2.1.1.**
- **Cálculo de la Pérdida (compute_loss y loss_derivative)**. Estas funciones calculan el valor de la pérdida y su derivada, respectivamente, utilizando la función de pérdida especificada. Esto es esencial para la retropropagación, donde se ajustan los pesos de la red en función del error calculado.
- **Actualización de la Tasa de Aprendizaje (__set_learning_rate).** Esta función privada se encarga de poner el learning rate especificado en el entrenamiento en todas las capas del modelo.
- **Early Stopping (__early_stopping).** Este método, también privado, implementa la lógica para detener el entrenamiento si la pérdida de validación no mejora durante un número especificado de épocas.

- **Layer.py.** En este archivo se define la clase abstracta **Layer**, que sirve como base para las diferentes capas que se pueden construir en nuestro modelo de red neuronal. A continuación, se describen las funciones que deben ser sobreescritas por las subclases:

  - **Propagación hacia adelante (forward).** Cualquier capa que herede de **Layer** debe implementar su propia versión de la función `forward`. Esta función debe tomar la entrada y calcular la salida de la capa para la propagación hacia adelante a través de la red.
  - **Propagación hacia atrás (backward).** De forma similar, las subclases deben proporcionar una implementación para la función `backward`, que se utilizará durante el proceso de retropropagación. La función debe calcular los gradientes de la capa, basándose en el gradiente del error recibido desde la capa siguiente.

  Al declarar estas funciones como métodos abstractos, la clase **Layer** establece un contrato que las subclases deben seguir, asegurando que cada capa definida en el marco de trabajo tenga una interfaz coherente y predecible.

- **DenseLayer.py.** Este archivo contiene la clase **DenseLayer**, que hereda de la clase base **Layer**. La clase **DenseLayer** representa una capa densa o completamente conectada de una red neuronal y es responsable de la mayor parte del cálculo durante la propagación hacia adelante y hacia atrás. Veamos con más detalle su implementación:

  - **Inicialización (__init__).** La función de inicialización establece los pesos y sesgos iniciales, y selecciona el optimizador que se usará para el entrenamiento. La inicialización de los pesos sigue la recomendación de He initialization. Esta inicialización es una estrategia de inicialización de pesos que toma en cuenta el tamaño de la capa anterior para mantener la varianza de las activaciones a través de las capas.

La inicialización de He se define con la siguiente fórmula:

$$W \sim \mathcal{N}\left(0, \sqrt{\frac{2}{n_{\text{in}}}}\right)$$

donde $W$ denota los pesos de la red neuronal, $\mathcal{N}(0, \sigma)$ es una distribución normal con media cero y desviación estándar $\sigma$, y $n_{\text{in}}$ representa el número de unidades de entrada en la capa anterior.

Por último, se configuran los parámetros para optimizadores como Adam, SGD con momentum y RMSProp.

- **Propagación hacia adelante (forward).** Esta función calcula la salida de la capa densa realizando un producto punto entre las entradas y los pesos, y luego sumando los sesgos.Corresponde con la ecuación [**1**].
- **Propagación hacia atrás (backward).** Esta función se encarga de calcular los gradientes de los pesos [**13**] y sesgos [**14**] basándose en el gradiente de la salida. Además, aplica el paso de optimización correspondiente para actualizar los pesos y sesgos de la capa.

  Dependiendo del optimizador seleccionado, se aplican diferentes estrategias para la actualización de los pesos y sesgos, incluyendo Adam, SGD con momentum y RMSProp. Cada uno de estos métodos utiliza diferentes enfoques para ajustar los pesos en función del gradiente de la función de pérdida. La implementación de estos métodos se corresponde con las ecuaciones en la **sección 2.1.3.**
- **Ajuste de la tasa de aprendizaje (learning_rate).** Se proporciona un getter y un setter para la tasa de aprendizaje, utilizado generalmente al principio del entrenamiento para que todas las capas tengan el mismo learning rate definido.

- **LossFunctions.py.** Este módulo contiene las definiciones de las clases **MSE** y **CrossEntropy**, que implementan las funciones de pérdida utilizadas para evaluar el rendimiento de nuestra red neuronal. Estas funciones de pérdida son críticas durante el proceso de entrenamiento, ya que proporcionan una medida de qué tan bien la red está realizando predicciones en comparación con los valores reales. Se exponen sus fórmulas en la **sección 2.1.3**. Aún así, pasaremos a detallar más a fondo estas, junto con su implementación:

  - **Mean Squared Error (MSE).**
    - **Cálculo de la pérdida (compute_loss).** Esta función calcula la pérdida cuadrática media (MSE) entre las predicciones y los valores reales.[**5**]
    - **Derivada de la pérdida (loss_derivative).** Proporciona el gradiente de la función de pérdida MSE con respecto a las predicciones.[**8**]
  - **Cross Entropy.**
    - **Inicialización (__init__).** El constructor toma una capa de activación, que se utiliza para ajustar el cálculo de la pérdida basado en si la capa es una Sigmoid o una Softmax, asegurando la compatibilidad con diferentes tipos de problemas de clasificación.

- **Cálculo de la pérdida (compute_loss).** Calcula la pérdida de entropía cruzada, una función de pérdida común para problemas de clasificación. La implementación incluye la estabilización numérica mediante el recorte de las predicciones para evitar el logaritmo de 0.**[6],[7].**
- **Derivada de la pérdida (loss_derivative).** Devuelve el gradiente de la función de pérdida de entropía cruzada.**[9],[10].**

- **Activation.py.** Este archivo define las clases para las funciones de activación **ReLU**, **Sigmoid** y **Softmax**, que son bloques de construcción esenciales para introducir no linealidades en nuestra red neuronal. Cada una de estas funciones de activación juega un papel crucial en la red, como se discute en las **sección 2.1.2**. Las clases son las siguientes:

  - **ReLU.**
    - **Propagación hacia adelante (forward).** Implementa la función de activación ReLU, que retorna cero para todos los valores negativos de entrada y el mismo valor para los valores positivos[**2**].
    - **Propagación hacia atrás (backward).** Calcula el gradiente de la función de activación ReLU, que es 0 para entradas negativas y 1 para entradas no negativas, como parte de la retropropagación[**11**].
  - **Sigmoid.**
    - **Propagación hacia adelante (forward).** Realiza la función de activación Sigmoid, que convierte las entradas en un rango entre 0 y 1, adecuado para la clasificación binaria[**3**].
    - **Propagación hacia atrás (backward).** Calcula el gradiente de la función de activación Sigmoid[**2.1.3**].
  - **Softmax.**
    - **Propagación hacia adelante (forward).** La función de activación Softmax convierte las entradas en un conjunto de valores, donde la suma de todas esos valores es 1. Es ideal para la clasificación multiclase [**4**].
    - **Propagación hacia atrás (backward).** Para la función Softmax, en combinación con la función de pérdida de entropía cruzada, el gradiente es simplificado durante la retropropagación. En el proceso de backpropagation, cuando se retrocede a través de la capa Softmax que se utiliza junto con la pérdida de entropía cruzada, la derivada de la pérdida respecto a los logits (las predicciones de la red antes de que se mapeen en la función softmax) es simplemente la diferencia entre los valores predichos y las etiquetas reales[**10**]. Este gradiente ya incorpora la contribución tanto de la función de activación Softmax como de la pérdida de entropía cruzada.

# 4. Experimentos y Pruebas

## 4.1 Datasets Clasificación

### 4.1.1 Dataset XOR

Cuadro 1: Reporte de Clasificación

| | **precision** | **recall** | **f1-score** | **support** |
|---|---|---|---|---|
| 0 | 1.00 | 1.00 | 1.00 | 2 |
| 1 | 1.00 | 1.00 | 1.00 | 2 |
| **accuracy** | | | 1.00 | 4 |
| **macro avg** | 1.00 | 1.00 | 1.00 | 4 |
| **weighted avg** | 1.00 | 1.00 | 1.00 | 4 |

Los resultados obtenidos para el problema de clasificación XOR sugieren un éxito total en el aprendizaje de la función XOR por parte de la red neuronal implementada. La red neuronal, compuesta por dos capas densas y una función de activación ReLU seguida de una Sigmoid en la capa de salida, ha logrado una precisión, un recall y un puntaje F1 de 1.00 en ambas clases. Esto indica que el modelo ha podido clasificar perfectamente los ejemplos del conjunto de datos XOR.

La arquitectura de la red, aunque simple, ha demostrado ser suficientemente potente para capturar la complejidad no lineal del problema XOR. El uso de una capa oculta con la función de activación ReLU es clave aquí, ya que permite a la red crear las fronteras de decisión no lineales necesarias para resolver XOR. Además, el entrenamiento durante 100 épocas con una tasa de aprendizaje de 0.1 parece haber sido adecuado para converger a una solución óptima.

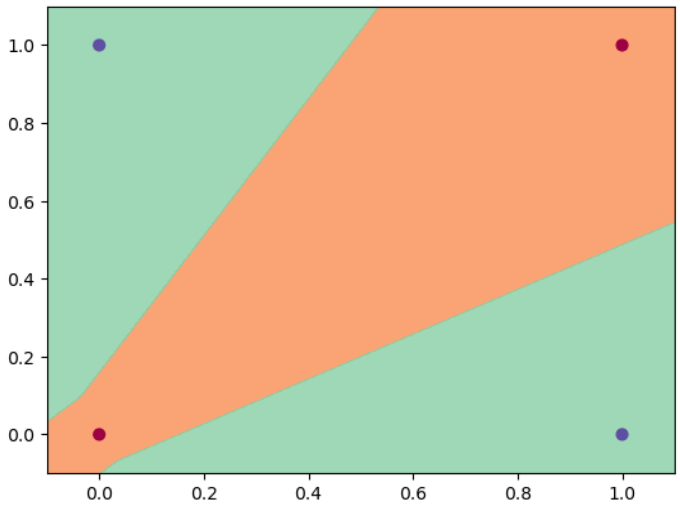



Figura 5: Superficie de decisión XOR.

La visualización de la superficie de decisión en la figura 5 proporciona una representación gráfica de cómo la red neuronal ha aprendido a separar las diferentes clases. Idealmente, para el problema XOR, esperaríamos ver una superficie de decisión que divida el espacio de entrada en cuatro regiones que corresponden a las salidas 0 y 1 de la función XOR. Los puntos (0,0) y (1,1) deberían clasificarse como una clase, y los puntos (0,1) y (1,0) como otra

#### 4.1.2 **Conjunto MNIST**

Cuadro 2: Reporte de Clasificación

| | **precision** | **recall** | **f1-score** | **support** |
|---|---|---|---|---|
| 0 | 0.92 | 0.96 | 0.94 | 980 |
| 1 | 0.95 | 0.97 | 0.96 | 1135 |
| 2 | 0.84 | 0.88 | 0.86 | 1032 |
| 3 | 0.89 | 0.80 | 0.84 | 1010 |
| 4 | 0.86 | 0.88 | 0.87 | 982 |
| 5 | 0.77 | 0.79 | 0.78 | 892 |
| 6 | 0.91 | 0.88 | 0.89 | 958 |
| 7 | 0.88 | 0.88 | 0.88 | 1028 |
| 8 | 0.85 | 0.80 | 0.83 | 974 |
| 9 | 0.82 | 0.84 | 0.83 | 1009 |
| **accuracy** | | | 0.87 | 10000 |
| **macro avg** | 0.87 | 0.87 | 0.87 | 10000 |
| **weighted avg** | 0.87 | 0.87 | 0.87 | 10000 |

En la tarea de clasificación del conjunto de datos MNIST, la red neuronal construida demuestra una competencia notable en el reconocimiento de dígitos escritos a mano. Utilizando una configuración de red con capas densas y funciones de activación Sigmoid, seguidas de una capa Softmax, la red ha logrado una precisión global del 87 %. Esta precisión, junto con los puntajes de recall y f1-score presentados, indica que el modelo es efectivo en diferenciar entre las diversas clases de dígitos.

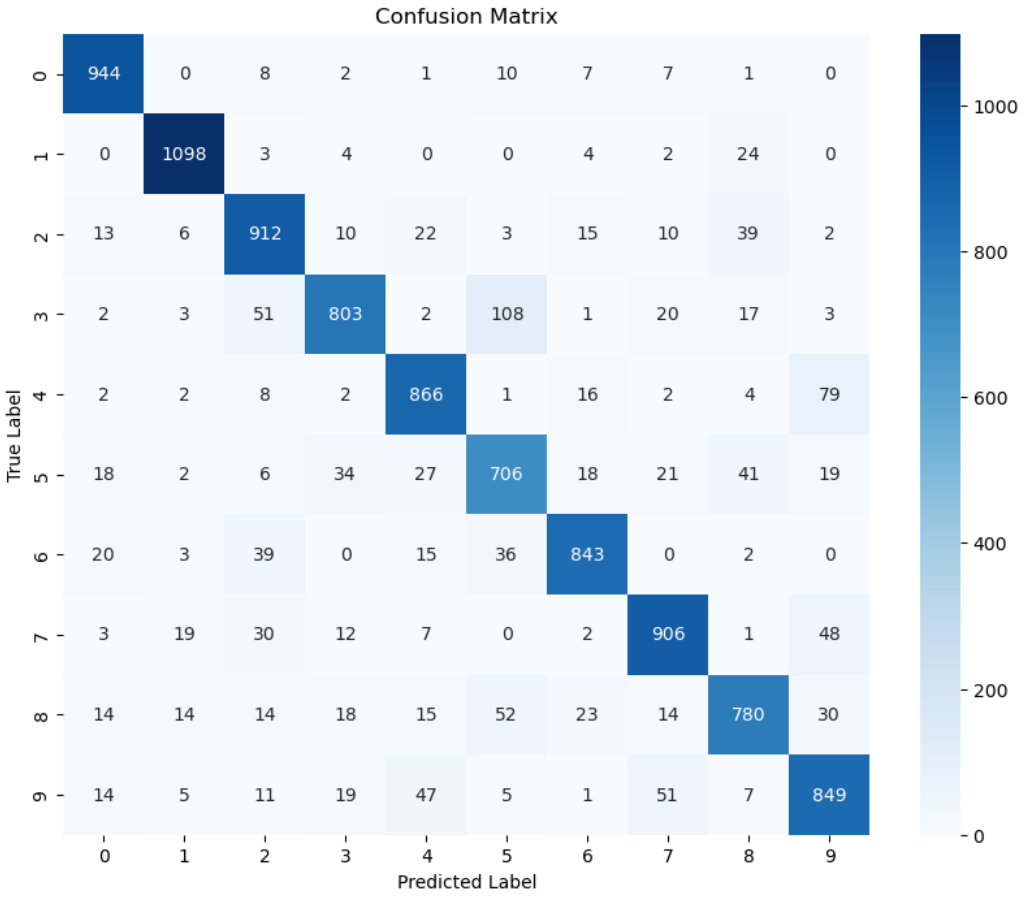



Figura 6: Matriz de confusión.

La matriz de confusión que vemos en la figura 7 subraya la habilidad del modelo para clasificar con precisión la mayoría de los dígitos, aunque se observan algunas dificultades en ciertas categorías. Por ejemplo, dígitos con formas similares, como los 3 y los 5, o los 4 y los 9, presentan mayores tasas de confusión, lo cual es un reto común en la clasificación visual debido a las variaciones inherentes en la escritura manual.

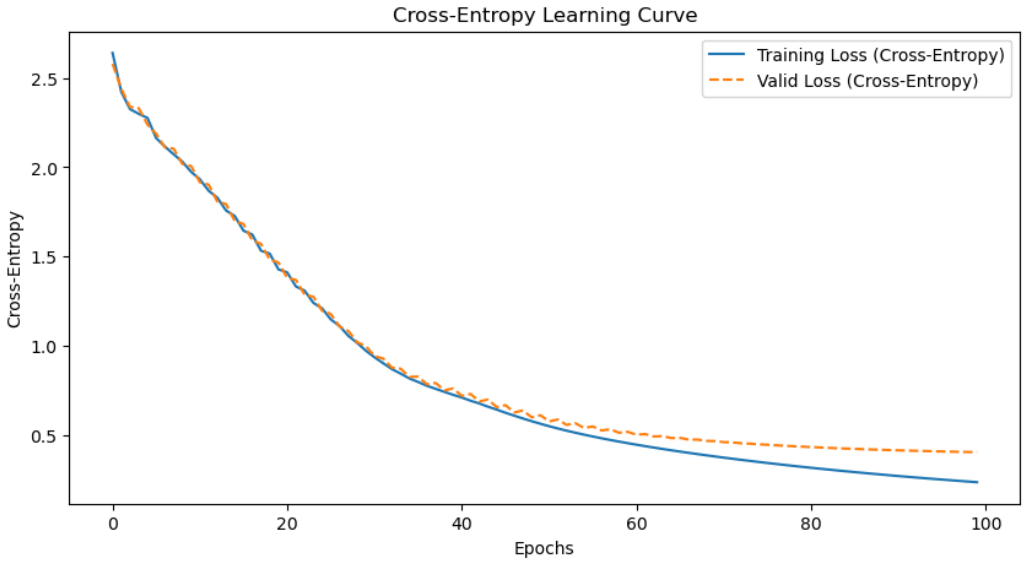



Figura 7: Curvas de aprendizaje conjunto MNIST.

Las curvas de aprendizaje para la pérdida de entropía cruzada en la figura en la figura 7 muestran una disminución constante tanto en el conjunto de entrenamiento como en el de validación, lo que implica una convergencia estable del modelo sin evidencia de sobreajuste. La tendencia de la curva de validación sigue de cerca a la de entrenamiento, lo que indica que el modelo generaliza bien a nuevos datos.

Este análisis destaca la capacidad del modelo para equilibrar la precisión de clasificación y la generalización, dos aspectos cruciales en el aprendizaje profundo. Sin embargo, existe espacio para optimizar aún más el rendimiento mediante la exploración de técnicas avanzadas, como el ajuste de hiperparámetros, la expansión de la arquitectura de la red o la implementación de técnicas de regularización más sofisticadas. El análisis integral de los resultados muestra un modelo robusto que sirve como un punto de partida sólido para futuras mejoras y experimentación.

### 4.1.3 **Dataset Titanic**

Cuadro 3: Reporte de Clasificación

| | **precision** | **recall** | **f1-score** | **support** |
|---|---|---|---|---|
| 0 | 0.81 | 0.87 | 0.84 | 105 |
| 1 | 0.79 | 0.72 | 0.75 | 74 |
| **accuracy** | | | 0.80 | 179 |
| **macro avg** | 0.80 | 0.79 | 0.80 | 179 |
| **weighted avg** | 0.80 | 0.80 | 0.80 | 179 |

La tarea de clasificación utilizando el conjunto de datos del Titanic ha arrojado resultados prometedores, evidenciados por una precisión general del 80 %. La red neuronal, compuesta por capas densas y activación ReLU, seguida por una capa de salida con activación Sigmoid, ha demostrado una capacidad adecuada para predecir la supervivencia de los pasajeros a bordo del Titanic.

Los resultados del informe de clasificación revelan una precisión y recall equilibrados para las clases de supervivencia y no supervivencia, con una ligera ventaja en la identificación de los no supervivientes. Este rendimiento es notable considerando la naturaleza desbalanceada y las complejidades inherentes al conjunto de datos, que incluyen la presencia de datos faltantes y la necesidad de preprocesamiento para manejar variables categóricas y numéricas.

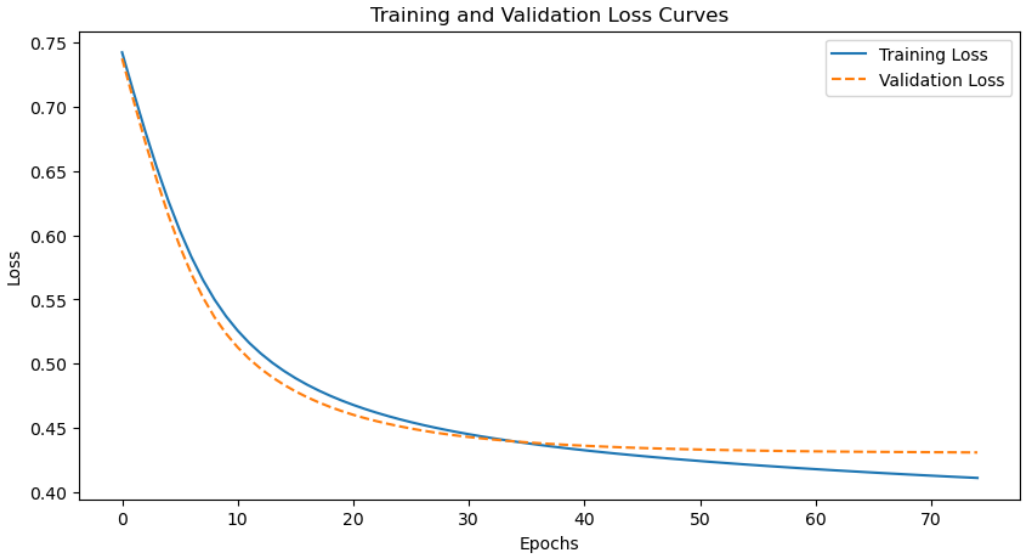



Figura 8: Curvas de aprendizaje del dataset Titanic.

Las curvas de pérdida de entrenamiento y validación de la figura 8 muestran una disminución constante y convergen a un punto que indica que el modelo está aprendiendo de manera efectiva sin sobreajustarse a los datos de entrenamiento.

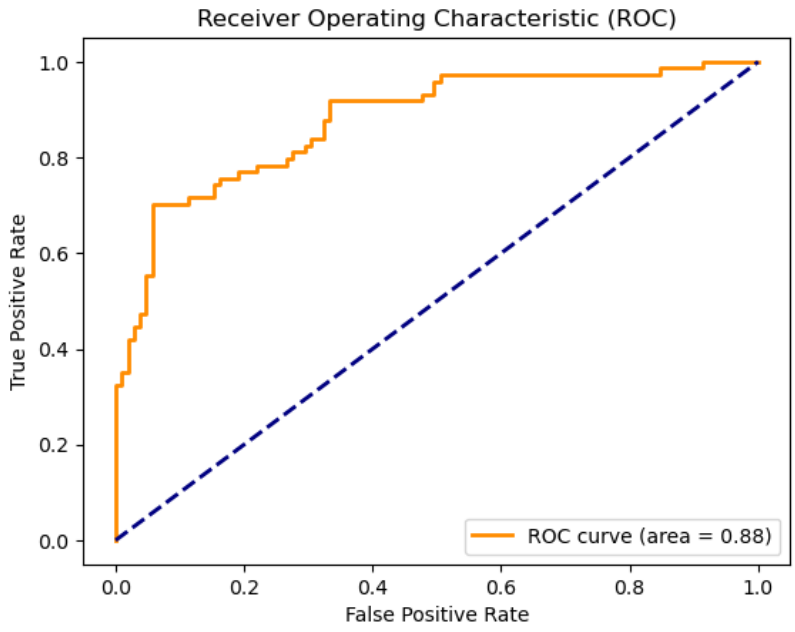



Figura 9: Curvas ROC dataset Titanic.

La curva ROC y el área bajo la curva (AUC) de 0.87 que podemos ver en la figura 9 brindan evidencia adicional de la capacidad del modelo para discriminar entre las clases positiva y negativa.

Estos resultados subrayan el éxito del modelo en capturar la relación entre las características de los pasajeros y sus probabilidades de supervivencia. No obstante, se podrían realizar ajustes y mejoras en la arquitectura de la red, el preprocesamiento de los datos y la ingeniería de características para mejorar aún más la precisión y el recall, especialmente en la clase de supervivientes donde el modelo mostró un rendimiento ligeramente inferior.

## 4.2 Datasets Regresión

### 4.2.1 Dataset Diabetes

Cuadro 4: Resultados de Entrenamiento en la Época 70

| | **Train** | **Validation** |
|---|---|---|
| **Loss (MSE)** | 2816.625 | 2911.411 |
| **R2 Score** | 0.5439 | 0.4585 |

**Nota:** Early stopping activado, MSE no mejora desde la época 60.

El análisis de regresión sobre el conjunto de datos de diabetes ha producido resultados mixtos. La red neuronal, diseñada con dos capas densas y activación ReLU, seguida por una salida lineal, alcanzó un coeficiente de determinación R2 de aproximadamente 0.46 en el

conjunto de prueba. Este valor de R2, que mide la proporción de la varianza de la variable dependiente que es predecible a partir de las variables independientes, indica una capacidad moderada de la red para modelar la relación entre las características de los pacientes y la progresión de la diabetes.

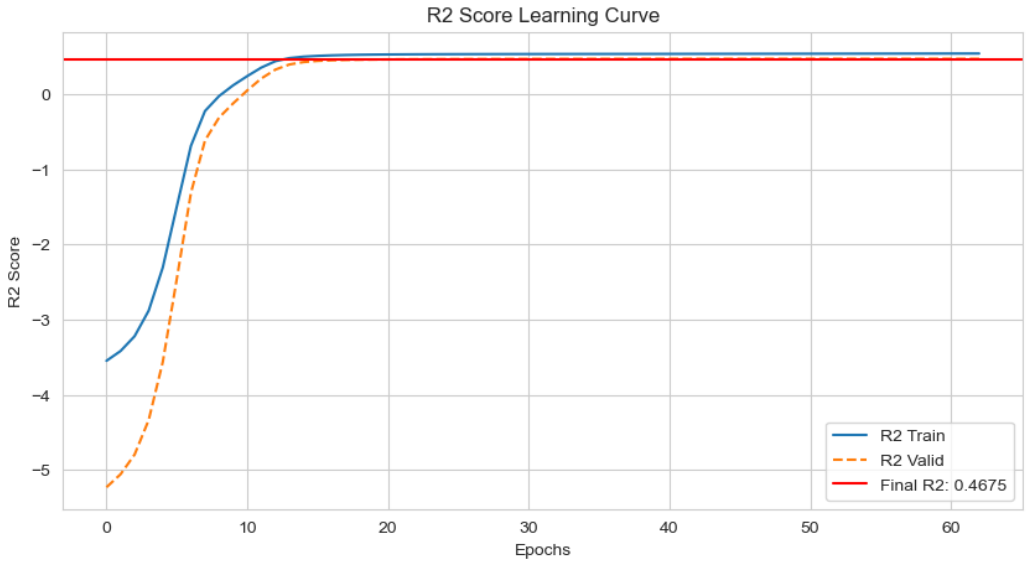



Figura 10: Curvas de aprendizaje del dataset Titanic.

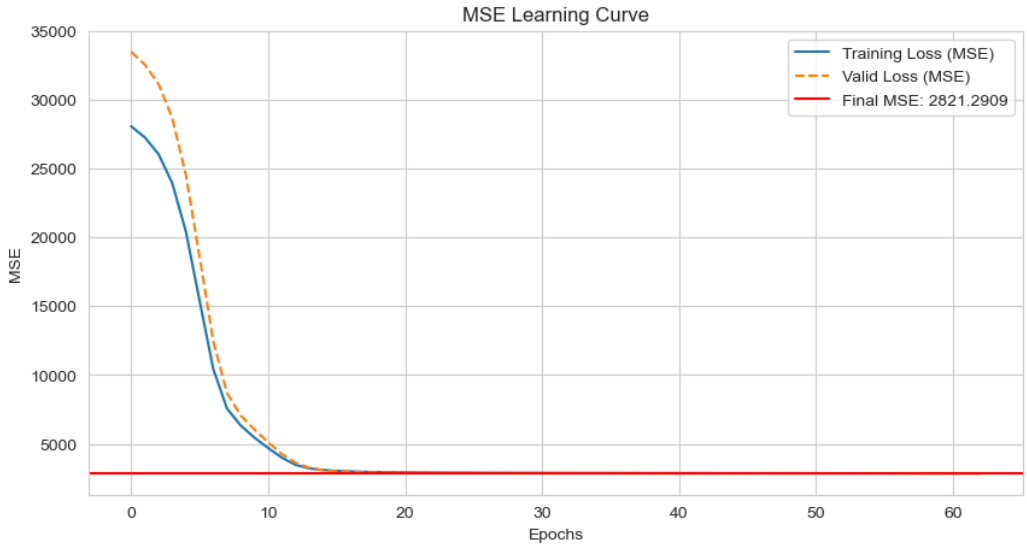



Figura 11: Curvas de aprendizaje del dataset Diabetes.

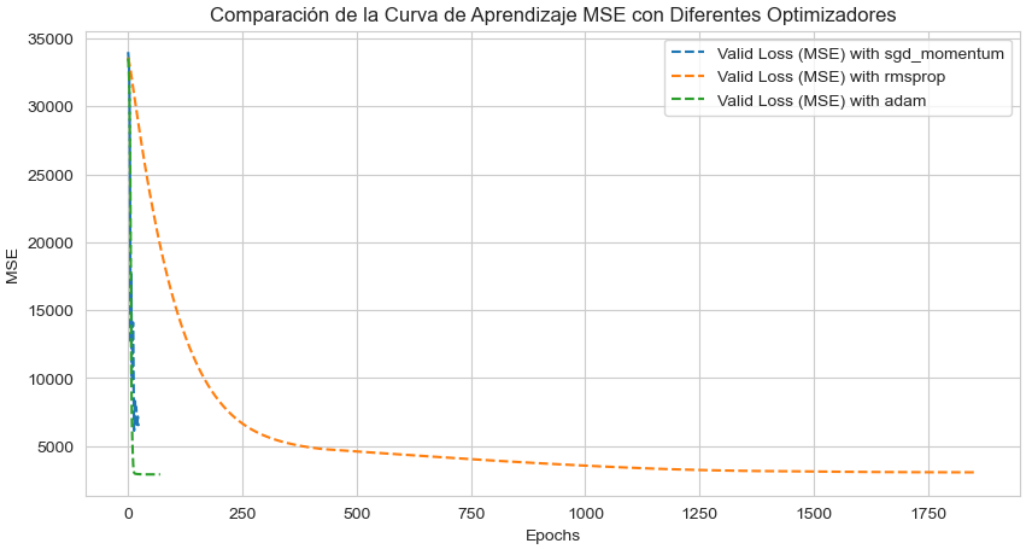



Figura 12: Comparación de optimizadores en el dataset Diabetes.

Las curvas de aprendizaje que vemos en las figuras 10 y 11 muestran una disminución sostenida en la pérdida de entrenamiento y validación, con una convergencia que sugiere un ajuste adecuado sin sobreajuste significativo. Sin embargo, el valor final de MSE es relativamente alto, lo que sugiere que el modelo todavía puede mejorarse. La comparación entre diferentes optimizadores que vemos en la figura 12 revela que el optimizador Adam proporciona una reducción de error más rápida y efectiva en comparación con SGD con momentum y RMSprop, lo que se refleja en las curvas de aprendizaje de la pérdida de validación.

Para mejorar el modelo, podríamos experimentar con una arquitectura de red más compleja, explorar técnicas de regularización para reducir la varianza del modelo, y considerar el ajuste fino de hiperparámetros como la tasa de aprendizaje y el número de neuronas en cada capa.

### 4.2.2 **Dataset Student Performance**

Cuadro 5: Resultados de Entrenamiento en la Época 94

| | **Train** | **Validation** |
|---|---|---|
| **Loss (MSE)** | 4.4233 | 4.3194 |
| **R2 Score** | 0.9880 | 0.9883 |

**Nota:** Early stopping activado, MSE no mejora desde la época 84.

La red neuronal aplicada a la predicción del desempeño de estudiantes ha demostrado ser extraordinariamente efectiva, alcanzando un coeficiente R2 próximo a 0.99 en el conjunto de prueba. Este resultado indica que la red es capaz de explicar casi toda la variabilidad del índice de rendimiento de los estudiantes, sugiriendo una correlación muy fuerte entre las

predicciones del modelo y los valores reales. La pérdida de error cuadrático medio (MSE) del modelo es sorprendentemente baja, con un valor final de aproximadamente 4.10, lo que refleja la alta precisión de las predicciones del modelo en términos de la magnitud de los errores.

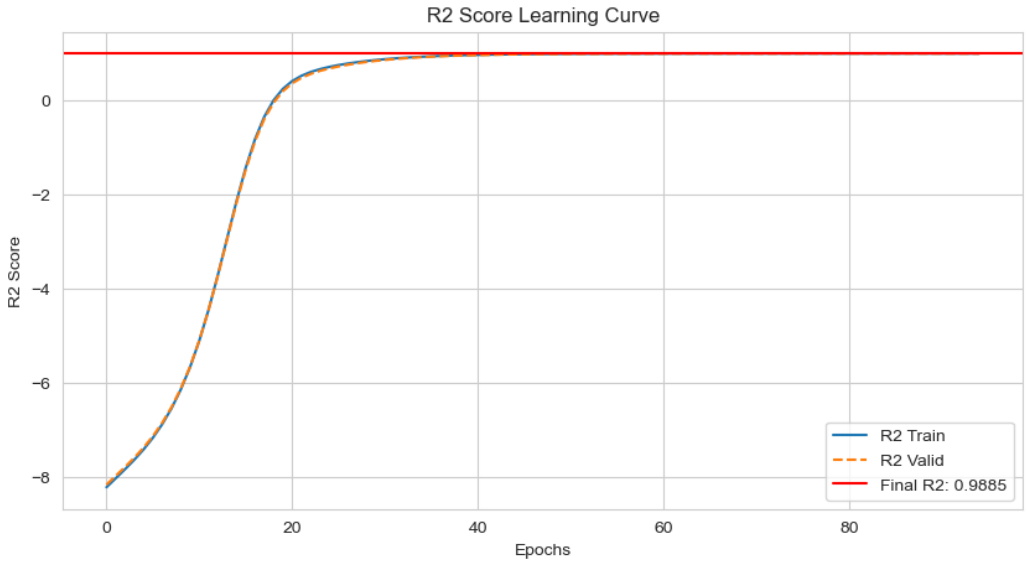



Figura 13: R2 dataset Student Performance.

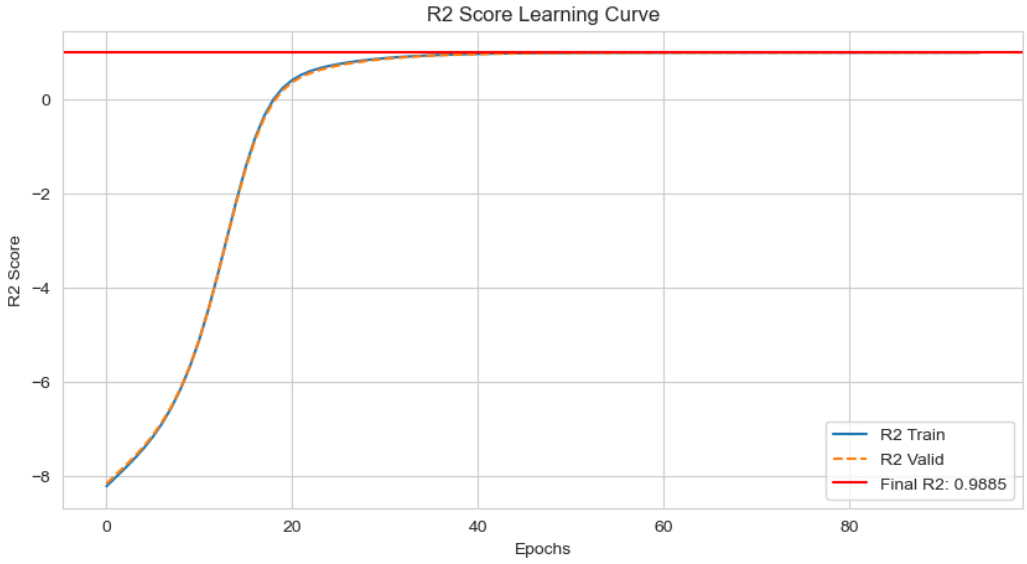



Figura 14: Loss en el dataset Student Performance.

Las curvas de aprendizaje que vemos en las figuras 13 y 14 muestran una rápida convergencia de la pérdida tanto en el conjunto de entrenamiento como en el de validación, lo

cual es indicativo de un buen ajuste del modelo sin señales de sobreajuste ni de falta de ajuste. La arquitectura de la red, que consta de una capa de entrada escalada y normalizada seguida de capas densas y activación ReLU, parece ser adecuada para capturar las complejas relaciones no lineales entre las horas de estudio, las puntuaciones previas, las horas de sueño, la práctica con exámenes de muestra y las actividades extracurriculares con respecto al desempeño académico.

## 5. Conclusiones y Trabajo Futuro

Este trabajo ha presentado una implementación de backpropagation en redes neuronales profundas con Python. Los experimentos realizados sobre conjuntos de datos de clasificación y regresión han demostrado la eficacia de nuestra red neuronal. A pesar de los logros observados en problemas de clasificación, como la precisión perfecta en el conjunto de datos XOR y un desempeño robusto en MNIST y Titanic, el rendimiento en tareas de regresión, como en el conjunto de datos de diabetes, ha sido modesto, con un R2 de aproximadamente 0.45. Este resultado indica que, mientras que la red tiene una capacidad considerable para abordar problemas de clasificación, aún existen oportunidades de mejora en tareas de regresión.

En el caso del rendimiento estudiantil, la red ha mostrado una capacidad notable para predecir resultados, evidenciado por un R2 cercano a 1. Estos resultados validan la arquitectura propuesta y las mejoras implementadas, como la inclusión de diversas funciones de activación y algoritmos de optimización.

Mirando hacia el futuro, hay varias áreas para continuar desarrollando este trabajo. Se planea explorar arquitecturas de red más avanzadas, como las redes neuronales convolucionales para el procesamiento de imágenes y las redes recurrentes para secuencias de datos. La implementación de bloques residuales y técnicas como dropout son también áreas prometedoras para mejorar la generalización y reducir el sobreajuste.

Además, se investigará la integración de mecanismos de atención y transformadores, que han demostrado ser muy efectivos en una amplia gama de tareas de aprendizaje profundo. La experimentación con diferentes formas de regularización y optimización de hiperparámetros también sería clave para mejorar la precisión y la capacidad predictiva de los modelos en problemas de regresión.

Por último, este marco se podría probar en problemas del mundo real para evaluar su aplicabilidad en sistemas y aplicaciones existentes, lo que podría proporcionar una contribución significativa al campo del aprendizaje automático y abrir nuevas vías para la investigación y el desarrollo tecnológico.

## 6. Bibliografía


[1] Daniel Q.D , Carlos M.E . (2023). *GitHub. Backpropagation-np-implementation.* `https://github.com/danqdon/Backpropagation-np-implementation`

[2] Dariel Dato-on. *Kaggle. MNIST in CSV.* `https://www.kaggle.com/datasets/oddrationale/mnist-in-csv`

[3] Kaggle. *Kaggle. Titanic: Machine Learning from Disaster*.https://www.kaggle.com/competitions/titanic/overview

[4] Scikit-learn developers. *Diabetes dataset*.https://scikit-learn.org/stable/datasets/toy_dataset.html#diabetes-dataset

[5] Nikhil Narayan. *Kaggle. Student Performance (Multiple Linear Regression)*.https://www.kaggle.com/datasets/nikhil7280/student-performance-multiple-linear-regression