Implementación de un motor de Red Neuronal

Joel Clemente López, Adonai Ojeda, Daniel Medina

January 7, 2025

Abstract

Este proyecto consiste en la implementación de una red neuronal desde cero en Python, sin utilizar librerías de aprendizaje profundo como TensorFlow o PyTorch. La red neuronal se ha entrenado y evaluado en tres conjuntos de datos clásicos: **Iris**, **Digits** y **Wine**. Este documento explica las clases y funciones diseñadas, los modelos desarrollados, así como el análisis de rendimiento mediante curvas ROC y matrices de confusión. El objetivo es demostrar el proceso de diseño, entrenamiento y evaluación de una red neuronal simple para tareas de clasificación multiclase.

1 Introducción

Las redes neuronales artificiales han emergido como una herramienta fundamental en el área del aprendizaje automático, con aplicaciones que abarcan desde el reconocimiento de patrones hasta la clasificación de datos complejos. Inspiradas en la estructura y funcionamiento del cerebro humano, estas redes permiten resolver problemas que, de otro modo, resultarían inabordables mediante técnicas tradicionales. En este contexto, el presente trabajo se centra en el diseño e implementación desde cero de un motor de redes neuronales en Python, evitando el uso de librerías de aprendizaje profundo como TensorFlow o PyTorch.

El proyecto aborda de manera integral todas las etapas necesarias para construir una red neuronal funcional. Esto incluye la definición de capas, el desarrollo de funciones de activación, la implementación del algoritmo de retropropagación y la optimización de los pesos. Para evaluar la eficacia del modelo desarrollado, se utilizaron tres conjuntos de datos clásicos del aprendizaje automático: Iris, Digits y Wine. Estos conjuntos permiten analizar cómo diferentes arquitecturas de red y funciones de activación impactan en el rendimiento general del modelo.

El documento está estructurado de la siguiente manera:

- En la sección **Métodos y Conjuntos de Datos**, se describen los enfoques metodológicos empleados, junto con una descripción detallada de los datasets utilizados.

- La sección **Detalles de Implementación** presenta una descripción técnica del código desarrollado, incluyendo las clases y funciones clave.
- En **Experimentos y Resultados**, se discuten los resultados obtenidos, acompañados de análisis visuales como curvas ROC y matrices de confusión.
- Finalmente, las secciones de **Conclusiones** y **Trabajo Futuro** resumen los hallazgos más relevantes y proponen líneas de mejora y extensión del trabajo.

2 Métodos y Conjuntos de Datos

2.1 Conjuntos de Datos

En este trabajo se utilizaron tres conjuntos de datos clásicos ampliamente conocidos en el ámbito del aprendizaje automático: Iris, Digits y Wine. Cada uno de ellos aporta características únicas que permiten evaluar el rendimiento de la red neuronal en diferentes contextos.

2.1.1 Iris

El conjunto de datos Iris contiene 150 muestras distribuidas uniformemente entre tres clases de flores: *Iris-setosa*, *Iris-versicolor* e *Iris-virginica*. Cada muestra se describe mediante cuatro atributos numéricos: la longitud y el ancho del sépalo, y la longitud y el ancho del pétalo. [5]

Table 1: Estadísticas descriptivas del conjunto de datos Iris

Atributo	Mínimo	Máximo	Media	Desviación Estándar
Longitud de Sépalo	4.3	7.9	5.84	0.83
Ancho de Sépalo	2.0	4.4	3.05	0.43
Longitud de Pétalo	1.0	6.9	3.76	1.76
Ancho de Pétalo	0.1	2.5	1.20	0.76

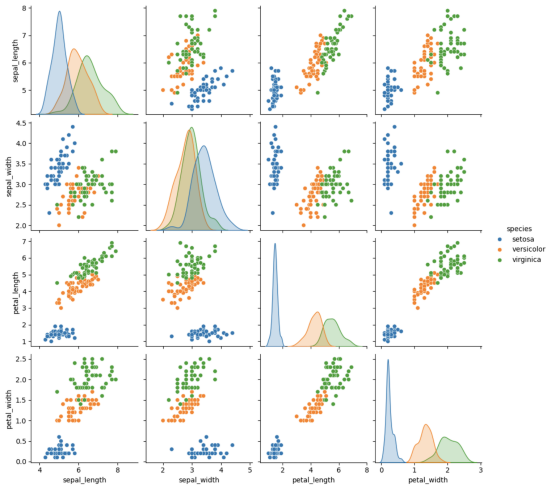


Figure 1: Ejemplo de distribución de las clases en el conjunto de datos Iris.

2.1.2 Digits

El conjunto de datos Digits contiene 1,797 imágenes de dígitos escritos a mano, con 10 clases correspondientes a los dígitos del 0 al 9. Cada imagen tiene una

resolución de 8x8 píxeles, y los valores de los píxeles oscilan entre 0 y 16. [6]

Table 2: Estadísticas descriptivas del conjunto de datos Digits

Métrica	Valor
Número de muestras	1,797
Dimensiones de la imagen	8x8
Clases	10 (0 a 9)

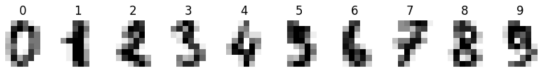


Figure 2: Ejemplo de imágenes del conjunto de datos Digits.

2.1.3 Wine

El conjunto de datos Wine contiene información química de 178 muestras de vino clasificadas en tres tipos diferentes. Cada muestra está caracterizada por 13 atributos continuos, como la acidez, el contenido de alcohol y el nivel de fenoles. [7]

Table 3: Estadísticas descriptivas del conjunto de datos Wine

Atributo	Mínimo	Máximo	Media	Desviación Estándar
Alcohol	11.03	14.83	13.00	0.81
Acidez	1.36	3.23	2.34	0.47
Fenoles totales	0.98	3.88	2.29	0.62
Color	1.28	13.00	5.06	2.31

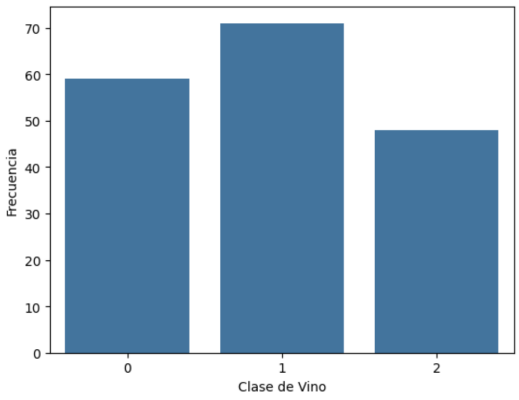


Figure 3: Distribución de clases en el conjunto de datos Wine.

2.2 Métodos

La metodología empleada se centra en la implementación de una red neuronal multicapa con retropropagación, optimización mediante descenso de gradiente y funciones de activación como ReLU, sigmoid, tanh y softmax. A continuación, se detallan las etapas principales:

- Inicialización de los pesos de las capas de forma aleatoria.
- Propagación hacia adelante (feedforward) para calcular la salida de la red.
- Cálculo del error en la capa de salida comparando la salida predicha con las etiquetas reales.
- Retropropagación del error para ajustar los pesos mediante el gradiente descendente.
- Evaluación del rendimiento del modelo usando métricas como la precisión y matrices de confusión.

Las implementaciones específicas se describen en detalle en la sección de Detalles de Implementación.

3 Detalles de Implementación

3.1 Funciones de Activación - activations.py

En `activations.py`, se definen funciones de activación que introducen no linealidad en la red neuronal, permitiendo que el modelo aprenda representaciones complejas. Las funciones implementadas son:

- **ReLU** (*Rectified Linear Unit*): La función ReLU devuelve 0 para valores negativos y el valor de entrada para valores positivos. Su derivada es 1 para valores positivos y 0 para valores negativos.

$$f(x) = \max(0, x)$$

- **Softmax**: Normaliza las salidas en probabilidades, útil en la capa de salida para clasificación multiclase. Su derivada calcula el gradiente necesario para la retropropagación.

$$f(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

- **Sigmoid**: La función sigmoide transforma la entrada a un rango entre 0 y 1. Es utilizada en capas ocultas, y su derivada es útil en retropropagación.

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

- **Tanh**: La función tangente hiperbólica mapea las entradas a un rango entre -1 y 1. Suele converger más rápido que la función sigmoide.

$$f(x) = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

3.2 Clase de Capa - layer.py

La clase `Layer` en `layer.py` define una capa individual de la red. Sus principales atributos y métodos son:

- **Atributos**: Dimensiones de la capa, función de activación, y su derivada.
- **Método forward**: Realiza la propagación hacia adelante en la capa, calculando las activaciones de las neuronas mediante la función de activación. El flujo de datos para una capa es:

$$a = f(Wx + b)$$

  donde W es la matriz de pesos, x es el vector de entradas, b es el sesgo, y a es la activación de salida.

3.3 Red Neuronal - neural_network.py

La clase `NeuralNetwork` en `neural_network.py` maneja la estructura completa de la red. Incluye los métodos fundamentales para el aprendizaje supervisado:

- **Método feedforward**: Propaga las entradas a través de todas las capas hasta la capa de salida. En cada capa, se realiza una combinación lineal de las entradas y los pesos, seguida de la aplicación de una función de activación.

- **Método backpropagation**: Calcula los gradientes de la función de pérdida con respecto a los pesos mediante el algoritmo de retropropagación. Este es el proceso esencial para el aprendizaje en redes neuronales, que se describe a continuación. [8]

  El algoritmo de retropropagación es un procedimiento iterativo que ajusta los pesos de la red para minimizar la función de pérdida. Este proceso se lleva a cabo mediante la regla de la cadena y el cálculo de los gradientes. Los pasos son los siguientes:

  1. Propagación hacia adelante: Se calculan las activaciones de todas las capas de la red.

  2. Cálculo del error en la capa de salida: La diferencia entre la salida deseada y y la salida calculada $\hat{y}$ se usa para calcular el error.

  $$\delta^L = \frac{\partial L}{\partial a^L} = a^L - y$$

  Donde L es la capa de salida y y es la etiqueta real.

  3. Retropropagación del error: El error en la capa de salida se propaga hacia las capas anteriores, utilizando la derivada de la función de activación en cada capa.

  $$\delta^l = (\delta^{l+1} W^{l+1}) \odot f'(z^l)$$

  Donde $\odot$ es el producto Hadamard, $f'(z^l)$ es la derivada de la función de activación en la capa l, y W^{l+1} son los pesos de la capa siguiente.

  4. Actualización de los pesos: Los pesos se actualizan utilizando el gradiente calculado:

  $$W^l \leftarrow W^l - \alpha \cdot \nabla W^l$$

  Donde α es la tasa de aprendizaje.

  El algoritmo se repite hasta que se alcanza una convergencia aceptable o se cumplen los criterios de detención.

Algorithm 1 Backpropagation

Require: X: Datos de entrada, y: Etiquetas verdaderas
Ensure: Lista de gradientes para cada capa

Inicializar una lista vacía: *derivates* $\leftarrow []$
Realizar la propagación hacia adelante para calcular las activaciones:
$A \leftarrow \texttt{feedforward}(X)$
$A \leftarrow \texttt{reshape}(A, \texttt{len}(A), 1)$
$y \leftarrow \texttt{reshape}(y, \texttt{len}(y), 1)$
$A_{\text{prev}} \leftarrow \texttt{feedforward}(X, -1)$
$A_{\text{prev}} \leftarrow \texttt{reshape}(A_{\text{prev}}, \texttt{len}(A_{\text{prev}}), 1)$
Calcular el error en la capa de salida:
$\delta z \leftarrow \texttt{activation\_derivate}(\textit{última capa}, A, y)$
$dW \leftarrow \delta z \cdot A_{\text{prev}}^T$
Añadir gradiente a la lista: *derivates.append*$((\delta z, dW))$
for k en $[1, \texttt{número de capas - 1}]$ **do**
    $A_{\text{prev}} \leftarrow \texttt{feedforward}(X, -k-2)$
    $A_{\text{prev}} \leftarrow \texttt{reshape}(A_{\text{prev}}, \texttt{len}(A_{\text{prev}}), 1)$
    Calcular derivada de activación:
    $\texttt{derivate} \leftarrow \texttt{activation\_derivate}(\textit{capa } k, \texttt{compute\_z}(A_{\text{prev}}))$
    $\delta z \leftarrow \texttt{derivate} \cdot (\textit{pesos de la capa siguiente})^T \cdot \delta z$
    $dW \leftarrow \delta z \cdot A_{\text{prev}}^T$
    Añadir gradiente a la lista: *derivates.append*$((\delta z, dW))$
end for
return *derivates*

3.4 Preprocesamiento - preprocessing.py

El archivo `preprocessing.py` incluye la función `preprocess_data`, que realiza dos pasos principales para preparar los datos antes de entrenar la red neuronal:

- **Escalado de las características**: Utiliza `StandardScaler` de *scikit-learn* para normalizar los datos de entrada X. Este proceso transforma las características para que tengan una media de 0 y una desviación estándar de 1, mejorando la convergencia del modelo durante el entrenamiento y reduciendo posibles problemas debido a diferencias de escala entre las características.
- **Codificación de las etiquetas**: Utiliza `OneHotEncoder` de *scikit-learn* para transformar las etiquetas y en un formato de codificación *one-hot*. Esto es especialmente útil para problemas de clasificación multiclase, ya que cada clase se representa como un vector binario.

El flujo de datos de la función `preprocess_data` es el siguiente:

1. Escalar los datos de entrada X con `StandardScaler`, generando un conjunto normalizado X_{scaled}.
2. Codificar las etiquetas y con `OneHotEncoder`, generando un conjunto codificado y_{encoded}.
3. Devolver los datos escalados y las etiquetas codificadas.

3.4.1 Ejemplo de uso

Para ilustrar el impacto del preprocesamiento, consideremos un conjunto de datos con tres características y etiquetas categóricas. Tras aplicar el preprocesamiento:

- **Los datos escalados** presentarán una distribución con media 0 y desviación estándar 1, reduciendo la influencia de magnitudes desiguales entre las características.
- **Las etiquetas codificadas** serán representadas como vectores binarios, facilitando la interpretación de clases en tareas de clasificación multiclase.

Este preprocesamiento asegura que los datos estén en el formato adecuado para el entrenamiento, mejorando la estabilidad numérica y optimizando el rendimiento del modelo.

3.5 Optimizadores - optimizers.py

El archivo `optimizers.py` incluye la implementación de algoritmos de optimización que ajustan los pesos de la red neuronal durante el entrenamiento. Estos algoritmos son fundamentales para garantizar la convergencia hacia un mínimo local de la función de pérdida y mejorar el rendimiento del modelo.

3.5.1 Descenso de Gradiente

El algoritmo de **Descenso de Gradiente** es uno de los métodos más básicos y ampliamente utilizados en el entrenamiento de redes neuronales. Su funcionamiento se basa en actualizar los pesos del modelo en la dirección opuesta al gradiente de la función de pérdida con respecto a los pesos.

- **Estrategia**: Los pesos W se actualizan según la fórmula:

$$W \leftarrow W - \alpha \cdot \nabla L(W)$$

  donde α es la tasa de aprendizaje y $\nabla L(W)$ es el gradiente de la pérdida con respecto a W.

- **Ventajas**: Es fácil de implementar y eficiente para problemas simples.

- **Limitaciones**: Puede ser lento para funciones de pérdida con múltiples mínimos locales o valles poco pronunciados, y es sensible a la elección de α.

Algorithm 2 Descenso de Gradiente

Require: Modelo: Red neuronal, X: datos de entrada, y: etiquetas verdaderas, α: tasa de aprendizaje, epochs: número de épocas

Dividir los datos en conjuntos de entrenamiento y validación.
for epoch = 1 hasta epochs **do**
  for cada muestra i en X_{train} **do**
    Calcular derivadas mediante retropropagación.
    Actualizar pesos y sesgos del modelo:

$$\text{weights} \leftarrow \text{weights} - \alpha \cdot \text{gradiente de pesos}$$

$$\text{bias} \leftarrow \text{bias} - \alpha \cdot \text{gradiente de sesgo}$$

  end for
  if epoch % 10 == 0 **then**
    Calcular precisión y pérdida en el conjunto de validación.
    Mostrar resultados de la época.
  end if
end for
return Listas de precisión y pérdida.

3.5.2 Momentum

El algoritmo **Momentum** es una extensión del descenso de gradiente que busca acelerar la convergencia acumulando una "inercia" de las actualizaciones pasadas. Esto permite que el optimizador supere gradientes pequeños en valles poco pronunciados y avance más rápido en direcciones importantes.

- **Estrategia**: Momentum introduce una velocidad v que acumula el gradiente con un factor de decaimiento β:

$$v \leftarrow \beta v - \alpha \nabla L(W)$$

$$W \leftarrow W + v$$

donde β controla el grado de inercia, α es la tasa de aprendizaje, y $\nabla L(W)$ es el gradiente de la pérdida.

- **Ventajas**: Convergencia más rápida en problemas con valles alargados.
- **Limitaciones**: Puede oscilar si los hiperparámetros no están bien ajustados.

Algorithm 3 Momentum

Require: Modelo: Red neuronal, X: datos de entrada, y: etiquetas verdaderas, α: tasa de aprendizaje, β: factor de inercia, epochs: número de épocas

Dividir los datos en conjuntos de entrenamiento y validación.
Inicializar las velocidades v_{weights} y v_{bias} en ceros para cada capa.
for epoch = 1 hasta epochs **do**
  for cada muestra i en X_{train} **do**
    Calcular derivadas mediante retropropagación.
    Actualizar velocidades:

$$v_{\text{weights}} \leftarrow \beta v_{\text{weights}} + \alpha \cdot \text{gradiente de pesos}$$

$$v_{\text{bias}} \leftarrow \beta v_{\text{bias}} + \alpha \cdot \text{gradiente de sesgo}$$

    Actualizar pesos y sesgos del modelo:

$$\text{weights} \leftarrow \text{weights} - v_{\text{weights}}$$

$$\text{bias} \leftarrow \text{bias} - v_{\text{bias}}$$

  end for
  if epoch % 10 == 0 **then**
    Calcular precisión y pérdida en el conjunto de validación.
    Mostrar resultados de la época.
  end if
end for
return Listas de precisión y pérdida.

3.5.3 Comparación entre Descenso de Gradiente y Momentum

Table 4: Comparación entre Descenso de Gradiente y Momentum

Característica	**Descenso de Gradiente**	**Momentum**
Ajuste de tasa de aprendizaje	Fija	Fija
Convergencia	Lenta	Rápida
Sensibilidad a hiperparámetros	Alta	Moderada
Complejidad computacional	Baja	Moderada
Uso en la práctica	Problemas simples	Problemas con topologías complejas

3.5.4 Impacto de los Optimizadores en el Modelo

La elección del optimizador tiene un impacto directo en:

- **Velocidad de convergencia**: Momentum suele ser más rápido y estable en problemas complejos.
- **Rendimiento final**: Puede superar al descenso de gradiente en escenarios donde la topología de la pérdida sea complicada.
- **Estabilidad numérica**: Momentum maneja mejor gradientes pequeños o ralos, mientras que el descenso de gradiente puede tener dificultades.

En resumen, `optimizers.py` proporciona herramientas esenciales para ajustar los pesos del modelo de manera eficiente, facilitando el aprendizaje en redes neuronales tanto simples como complejas.

3.6 Visualización de Resultados - visualizations.py

El archivo `visualizations.py` incluye funciones esenciales para evaluar visualmente el rendimiento de la red neuronal. Estas herramientas proporcionan una comprensión intuitiva de cómo el modelo clasifica los datos y cómo se comporta en tareas de clasificación multiclase.

3.6.1 Matriz de Confusión

La función correspondiente genera una matriz de confusión que compara las etiquetas verdaderas con las etiquetas predichas por el modelo. Esta matriz permite identificar clases que son fáciles o difíciles de distinguir, destacando patrones en las clasificaciones incorrectas y correctas.

- Las filas de la matriz representan las clases verdaderas, mientras que las columnas representan las clases predichas.
- Una diagonal dominante indica que la mayoría de las predicciones son correctas.
- Los valores fuera de la diagonal destacan los errores de clasificación.

3.6.2 Curva ROC

La curva ROC (Receiver Operating Characteristic) es una herramienta clave para evaluar el rendimiento del modelo en clasificación multiclase. En este archivo, se genera una curva ROC para cada clase, proporcionando una representación visual de la tasa de verdaderos positivos frente a la tasa de falsos positivos.

- Para cada clase, se considera una configuración de clasificación binaria en la que esa clase se compara contra todas las demás.
- La curva se construye trazando el par (FPR, TPR), donde:
  - **FPR (False Positive Rate)**: Proporción de negativos incorrectamente clasificados como positivos.
  - **TPR (True Positive Rate)**: Proporción de positivos correctamente clasificados.
- Una curva más cercana al punto (0, 1) indica un mejor rendimiento.

3.6.3 Beneficios de la Visualización

Las herramientas implementadas en `visualizations.py` permiten:

- Identificar rápidamente las debilidades del modelo en clases específicas.
- Evaluar el balance entre la sensibilidad (TPR) y la especificidad (1-FPR).
- Proporcionar un enfoque visual para comunicar los resultados del modelo a audiencias técnicas y no técnicas.

Estos gráficos son esenciales para ajustar hiperparámetros, mejorar arquitecturas y evaluar el impacto del preprocesamiento en el rendimiento del modelo.

3.7 Carga de Datos - load_data.py

El archivo `load_data.py` es responsable de la carga y preparación inicial de los conjuntos de datos utilizados en este trabajo: Iris, Digits y Wine. Este archivo proporciona funciones que permiten obtener de manera eficiente las matrices de características y las etiquetas asociadas, asegurando su compatibilidad con los pasos posteriores de preprocesamiento y entrenamiento.

3.7.1 Estructura de las Funciones de Carga

Cada función de carga realiza los siguientes pasos clave:

1. **Obtención de los datos**: Utiliza *scikit-learn* para acceder a los conjuntos de datos estandarizados.

2. **Separación de características y etiquetas**: Extrae la matriz de características X y el vector de etiquetas y de los datos.

3. **Conversión a formatos compatibles**: Asegura que los datos se devuelvan en formatos que puedan ser directamente utilizados en las etapas posteriores, como arrays de `NumPy`.

3.7.2 Impacto en el Flujo de Trabajo

El archivo `load_data.py` desempeña un papel crucial en el flujo de trabajo del modelo. Sus principales contribuciones incluyen:

- Garantizar la consistencia en la estructura de los datos proporcionados a las etapas de preprocesamiento y entrenamiento.
- Reducir la complejidad del código principal, al centralizar la carga de datos en un módulo dedicado.
- Facilitar la extensión del modelo a nuevos conjuntos de datos, al permitir la adición de funciones de carga específicas sin alterar el núcleo del código.

3.7.3 Ejemplo de Datos Cargados

Para el conjunto de datos Iris, por ejemplo, la matriz de características X contendrá atributos como la longitud y el ancho del sépalo y del pétalo, mientras que el vector de etiquetas y indicará la especie correspondiente de cada muestra (*setosa*, *versicolor* o *virginica*).

- **Características (X)**:

$$\begin{bmatrix} 5.1 & 3.5 & 1.4 & 0.2 \\ 4.9 & 3.0 & 1.4 & 0.2 \\ \vdots & \vdots & \vdots & \vdots \end{bmatrix}$$

- **Etiquetas (y)**:

$$\begin{bmatrix} 0 \\ 0 \\ \vdots \end{bmatrix}$$

En conjunto, `load_data.py` asegura una preparación adecuada de los datos para todas las etapas posteriores del modelo, desde el preprocesamiento hasta la evaluación del rendimiento.

3.8 Notebook de Pruebas - main.ipynb

El archivo `main.ipynb` es un notebook interactivo utilizado para experimentar y evaluar diferentes configuraciones de la red neuronal implementada. Este notebook centraliza las tareas de entrenamiento, prueba y visualización de resultados, permitiendo una exploración flexible del modelo.

3.8.1 Estructura del Notebook

El notebook está organizado en varias celdas que corresponden a las etapas principales del flujo de trabajo:

1. **Carga de bibliotecas y configuraciones**: Importa las librerías necesarias, como *NumPy*, *Matplotlib*, y los módulos del proyecto, y configura los hiperparámetros iniciales (como la tasa de aprendizaje, el número de épocas, etc.).

2. **Carga y preprocesamiento de datos**: Utiliza las funciones de `load_data.py` y `preprocessing.py` para cargar y preparar los conjuntos de datos.

3. **Definición de la arquitectura**: Permite definir y ajustar las capas, funciones de activación y optimizadores del modelo.

4. **Entrenamiento del modelo**: Ejecuta el algoritmo de entrenamiento utilizando las funciones implementadas en `neural_network.py`. Se registra la evolución de las métricas de rendimiento, como la precisión y la pérdida, a lo largo de las épocas.

5. **Evaluación del modelo**: Calcula métricas como la matriz de confusión y las curvas ROC, utilizando las herramientas de `visualizations.py`.

6. **Visualización de resultados**: Genera gráficos que permiten analizar el comportamiento del modelo, incluyendo curvas de entrenamiento (pérdida y precisión) y distribuciones de predicciones.

3.8.2 Beneficios del Notebook

El uso de `main.ipynb` ofrece varias ventajas para el desarrollo y análisis del modelo:

- Facilita la experimentación interactiva al permitir ajustes rápidos en la configuración del modelo y en los hiperparámetros.
- Proporciona un entorno visual para observar el impacto de los cambios en tiempo real.
- Centraliza todas las etapas del flujo de trabajo en un único archivo, mejorando la organización del proyecto.

3.8.3 Casos de Uso

En `main.ipynb`, se pueden realizar las siguientes actividades específicas:

- Comparar arquitecturas de red variando el número de capas y neuronas.
- Evaluar el impacto de diferentes funciones de activación y optimizadores.
- Analizar cómo el preprocesamiento de datos afecta al rendimiento del modelo.

- Identificar problemas como sobreajuste u optimización deficiente mediante la observación de curvas de entrenamiento.

3.8.4 Resultados Generados

El notebook produce una variedad de resultados que permiten evaluar el rendimiento del modelo y su capacidad de generalización:

- **Gráficos de Precisión y Pérdida**: Muestran la evolución de estas métricas a lo largo de las épocas, facilitando la detección de problemas como el sobreajuste.
- **Matrices de Confusión**: Resumen las predicciones correctas e incorrectas para cada clase.
- **Curvas ROC**: Permiten analizar el equilibrio entre sensibilidad y especificidad para cada clase.
- **Resúmenes de Hiperparámetros**: Documentan las configuraciones que producen los mejores resultados, ayudando a optimizar futuras ejecuciones.

En resumen, `main.ipynb` actúa como un entorno versátil para desarrollar, probar y evaluar la red neuronal, proporcionando tanto resultados cuantitativos como visuales que mejoran la comprensión del modelo y su comportamiento.

4 Experimentos y Resultados

Los resultados obtenidos muestran la precisión antes y después del entrenamiento, con una mejora significativa en cada modelo. Además, las matrices de confusión y las curvas ROC permiten analizar el rendimiento de los modelos en cada clase, resaltando fortalezas y áreas de mejora.

4.1 Test del Modelo Iris

En este análisis, se han implementado tres modelos de red neuronal con diferentes arquitecturas y configuraciones de capas para evaluar su capacidad de clasificación. El primer modelo, diseñado con una arquitectura simple,. El segundo modelo, que incrementa la complejidad al introducir múltiples capas ocultas. Finalmente, el tercer modelo, que continúa añadiendo capas, sugiriendo que la sobrecomplicación de la red puede llevar a un ligero descenso en la capacidad de generalización.

- **Modelo 1**:

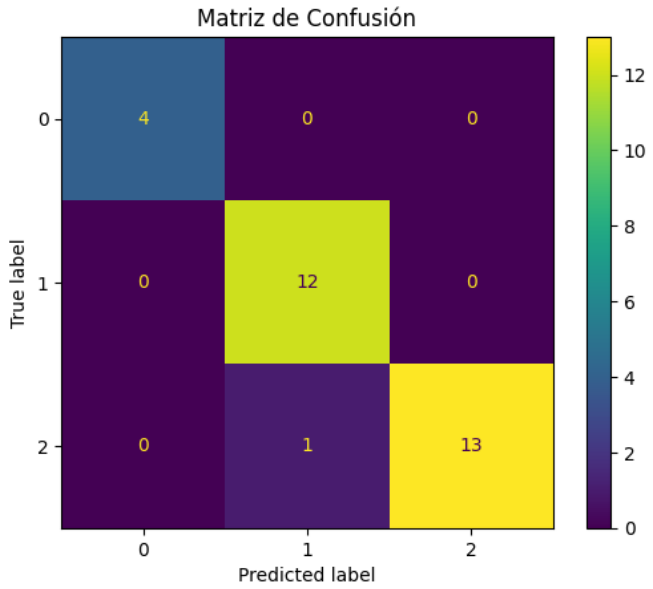


Figure 4: Model 1

- **Modelo 2**:

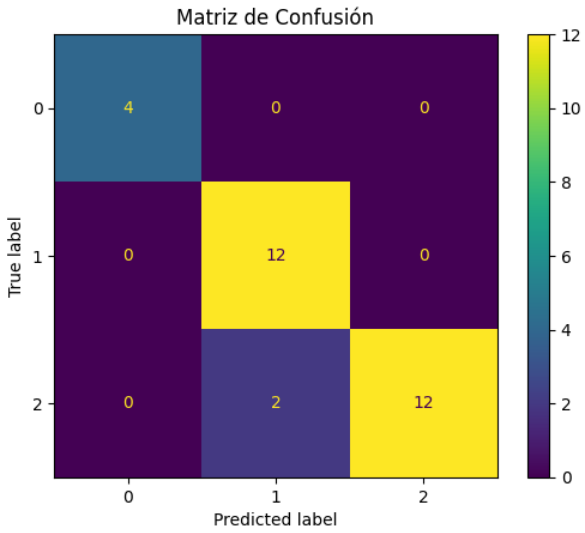


Figure 5: Model 2

- **Modelo 3**:

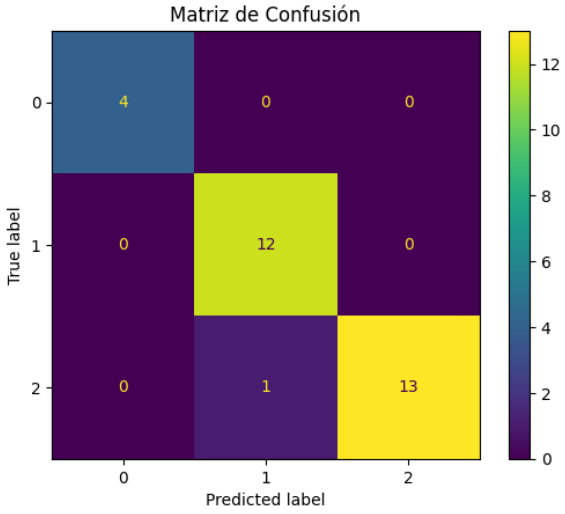


Figure 6: Model 3

- **Modelo 4**:

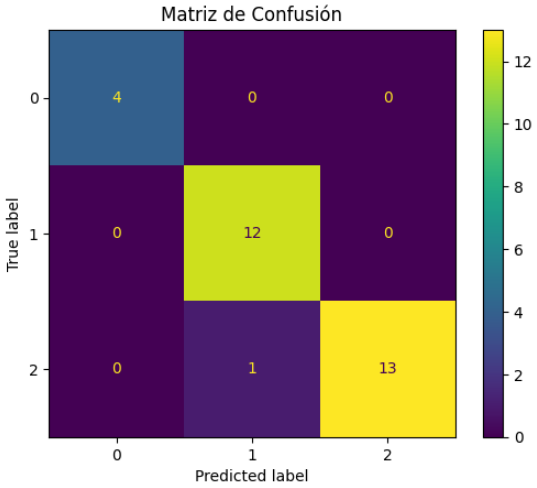


Figure 7: Model 3

Los resultados del entrenamiento de los modelos en el conjunto de datos Iris reflejan claramente la relación entre la complejidad del modelo y su rendimiento en términos de precisión.

El primer modelo, con una arquitectura sencilla compuesta por una única capa oculta de 5 neuronas y una función de activación ReLU, alcanzó una precisión del 97%. Este resultado demuestra que incluso configuraciones simples pueden capturar de manera efectiva las relaciones subyacentes en conjuntos de datos bien estructurados como Iris, evitando riesgos significativos de sobreajuste.

Por otro lado, el segundo modelo introdujo una mayor complejidad al incluir múltiples capas ocultas con tamaños de 20, 10 y 6 neuronas, todas utilizando la función de activación ReLU. A pesar de esta sofisticación adicional, la precisión alcanzada fue de un 93%, ligeramente inferior a la del modelo inicial. Este descenso sugiere que el aumento en el número de parámetros puede provocar un sobreajuste en conjuntos de datos más pequeños, limitando la capacidad de generalización del modelo.

El tercer modelo, diseñado con una arquitectura intermedia, combinó funciones de activación ReLU y Tanh en tres capas ocultas de 10, 8 y 5 neuronas respectivamente. Sorprendentemente, este modelo logró igualar la precisión del primer modelo con un 97%. Esto resalta que la complejidad adicional no siempre se traduce en un mejor rendimiento y que, en este caso, una combinación adecuada de funciones de activación y tamaños de capas fue suficiente para capturar eficazmente las relaciones en los datos.

Finalmente, el cuarto modelo incorporó un diseño más complejo, utilizando funciones de activación Sigmoide en cuatro capas ocultas de 20, 10 y 6 neuronas,

junto con el método de optimización de descenso de gradiente con momentum. Este modelo también alcanzó una precisión del 97%, demostrando que, si bien las arquitecturas más avanzadas pueden igualar el desempeño de modelos más simples, su estabilidad y capacidad de aprendizaje pueden depender en gran medida de la técnica de optimización utilizada.

En general, estos resultados subrayan la importancia de equilibrar la complejidad del modelo con la capacidad de generalización. Mientras que las arquitecturas simples, como la del primer modelo, son altamente efectivas para conjuntos de datos estructurados y compactos como Iris, las configuraciones más complejas no necesariamente garantizan mejores resultados, aunque pueden beneficiarse de optimizaciones avanzadas como el momentum. Este análisis destaca la necesidad de evaluar cuidadosamente las características del problema y del conjunto de datos antes de diseñar la arquitectura de la red neuronal, buscando siempre maximizar la precisión sin comprometer la generalización.

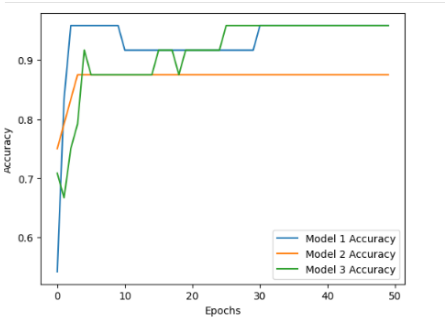


Figure 8: Accuracy 1,2,3

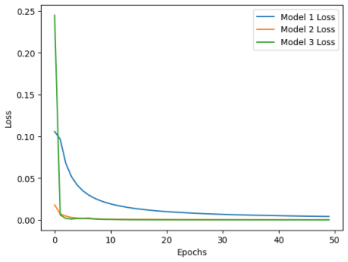


Figure 9: Loss 1,2,3

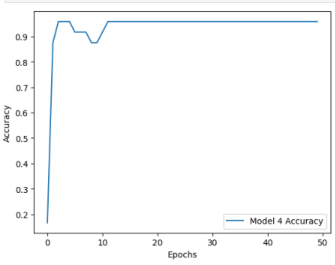


Figure 10: Accuracy 4

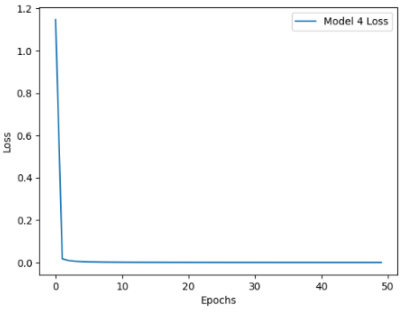


Figure 11: Loss 4

4.2 Test del Modelo Digits

En este análisis, se desarrollan y comparan tres modelos de red neuronal con diferentes arquitecturas para optimizar la precisión en la clasificación. El primer modelo, diseñado para ser compacto, logra una precisión aceptable, pero su simplicidad limita su capacidad para capturar la complejidad de los datos. El segundo modelo, con una estructura más profunda y múltiples capas ocultas, muestra una mejora significativa en la precisión, sugiriendo que ha aprendido patrones más complejos. Sin embargo, un tercer modelo con mayor complejidad no logra una mejora en la precisión esperada, lo que plantea la necesidad de un equilibrio entre la complejidad de la arquitectura y la capacidad de generalización.

Modelo 1:

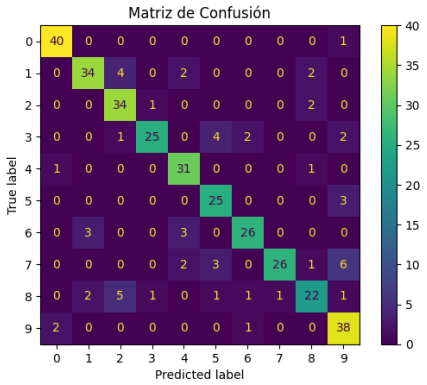


Figure 12: Confusion Matrix Model 1

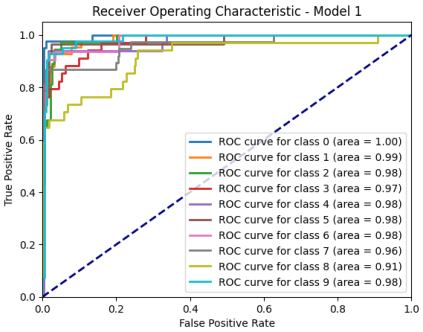


Figure 13: ROC Curve Model 1

Modelo 2:

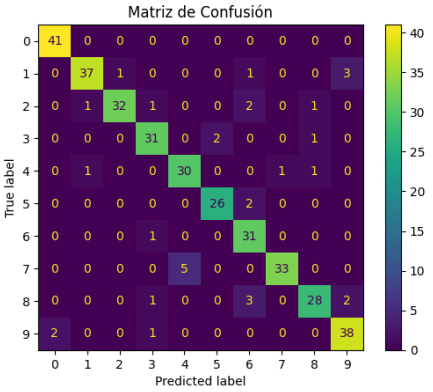


Figure 14: Confusion Matrix Model 2

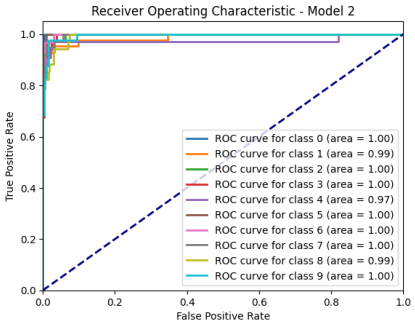


Figure 15: ROC Curve Model 2

Modelo 3:

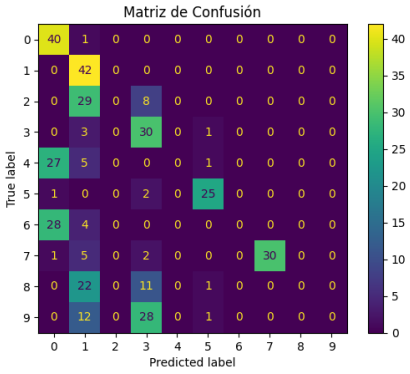


Figure 16: Confusion Matrix Model 3

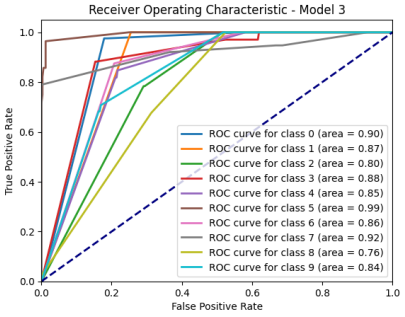


Figure 17: ROC Curve Model 3

Modelo 4:

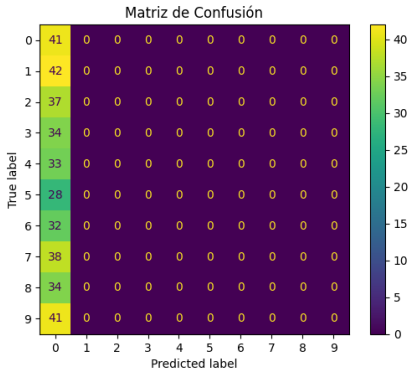


Figure 18: Confusion Matrix Model 4

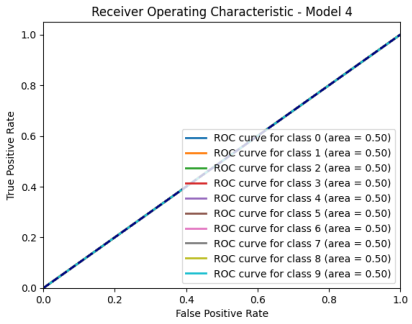


Figure 19: ROC Curve Model 4

Los resultados del entrenamiento utilizando el conjunto de datos DIGITS ilustran cómo diferentes configuraciones arquitectónicas de redes neuronales afectan el rendimiento en tareas de clasificación. A continuación, se analizan los resultados de los cuatro modelos evaluados, al igual que se comentarán las gráficas.

El primer modelo fue diseñado con una arquitectura compacta, compuesta por una capa oculta con 5 neuronas que utiliza la función de activación Tanh, y una capa de salida con 10 neuronas utilizando Softmax. Este modelo logró una precisión inicial del 6.1%, que aumentó significativamente hasta un 83.6% después del entrenamiento.

La matriz de confusión y la curva ROC muestran que, aunque este modelo alcanza una precisión decente, presenta limitaciones al clasificar dígitos visualmente similares. La simplicidad de la arquitectura resulta en un aprendizaje eficaz para patrones generales, pero insuficiente para capturar características más específicas y complejas.

El segundo modelo amplió la complejidad arquitectónica, incluyendo tres capas ocultas de 32, 16 y 8 neuronas con activaciones Sigmoide. Después de un entrenamiento exhaustivo, este modelo alcanzó una precisión de 90.8%, superando notablemente al modelo anterior.

La matriz de confusión muestra una reducción en los errores de clasificación, particularmente para dígitos con formas similares. Además, la curva ROC evidencia una mejora en las áreas bajo las curvas para cada clase, lo que indica un mejor rendimiento al distinguir entre clases. Esta mejora se atribuye a la capacidad de las capas adicionales para aprender patrones más complejos y diferenciadores.

El tercer modelo utilizó una combinación de funciones de activación (ReLU y Tanh) en una arquitectura de cuatro capas ocultas con tamaños decrecientes: 64, 32, 16 y 10 neuronas. Sin embargo, este diseño no logró los resultados

esperados, alcanzando solo un 46.3% de precisión después del entrenamiento.

El bajo desempeño de este modelo se debe, probablemente, a problemas de sobreajuste y a la combinación no óptima de funciones de activación. Aunque las capas adicionales y la variabilidad en las funciones de activación buscaban capturar mejor las características del conjunto de datos, estas no se tradujeron en un aprendizaje efectivo. La matriz de confusión y la curva ROC reflejan un desempeño significativamente inferior, con mayores errores y áreas bajo las curvas más pequeñas.

El cuarto modelo implementó una arquitectura similar al Modelo 3, pero con un optimizador basado en descenso de gradiente con momentum. A pesar de las expectativas, este modelo obtuvo la peor precisión entre todos, alcanzando solo un 11.3%.

Este resultado destaca las limitaciones de aplicar un optimizador más avanzado sin considerar adecuadamente la compatibilidad con la arquitectura y las características del conjunto de datos. La matriz de confusión y la curva ROC confirman que el modelo no fue capaz de distinguir correctamente las clases, lo que sugiere un mal ajuste en los parámetros durante el entrenamiento.

En general, los resultados sugieren que una arquitectura moderadamente compleja y bien diseñada, como la del Modelo 2, es ideal para tareas de clasificación de dígitos en conjuntos de datos como DIGITS. Sin embargo, mejorar el rendimiento requiere un ajuste fino de las funciones de activación, los optimizadores y las configuraciones arquitectónicas.

4.3 Test del Modelo Wine

En este estudio, se han implementado dos modelos de red neuronal para evaluar su capacidad de clasificación. El primer modelo logró una precisión sólida, indicando una adecuada arquitectura y elección de funciones de activación. A partir de estos resultados, se diseñó un segundo modelo más complejo con el objetivo de mejorar el rendimiento.

Modelo 1:

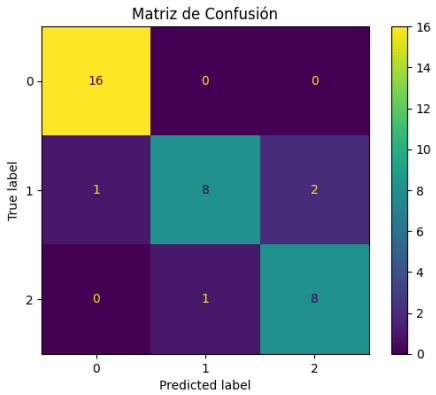


Figure 20: Confusion Matrix Model 1

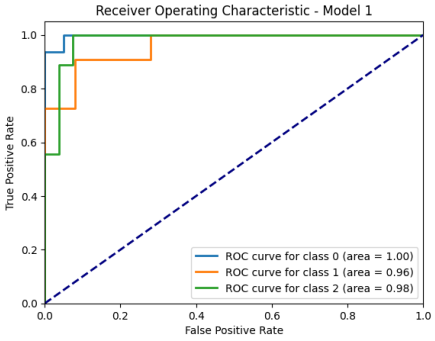


Figure 21: ROC Curve Model 1

Modelo 2:

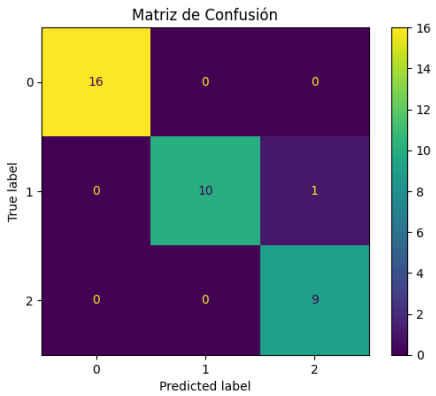


Figure 22: Confusion Matrix Model 2

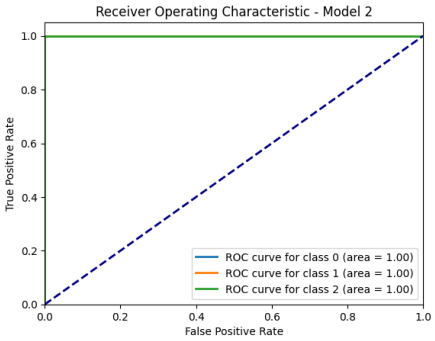


Figure 23: ROC Curve Model 2

Los resultados obtenidos de los modelos de red neuronal aplicados al conjunto de datos Wine destacan la efectividad de las configuraciones elegidas. El

modelo 1 alcanzó una precisión sólida en la clasificación, lo que sugiere que su arquitectura y funciones de activación fueron adecuadas para el problema. Sin embargo, el modelo 2, con su arquitectura más compleja de cinco capas y el uso de la función de activación sigmoid, logró una impresionante precisión del 100 por ciento. Este resultado indica que el modelo pudo aprender eficazmente las complejas relaciones entre las características químicas de los vinos y sus clases correspondientes.

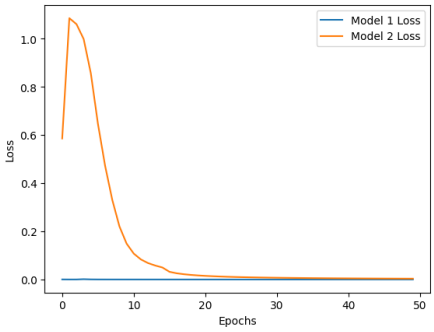


Figure 24: Loss

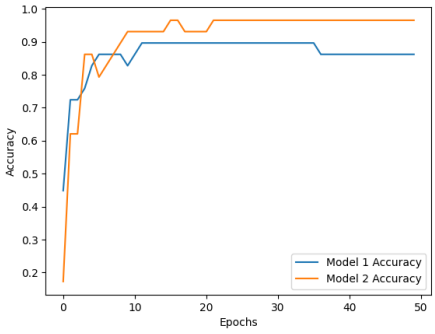
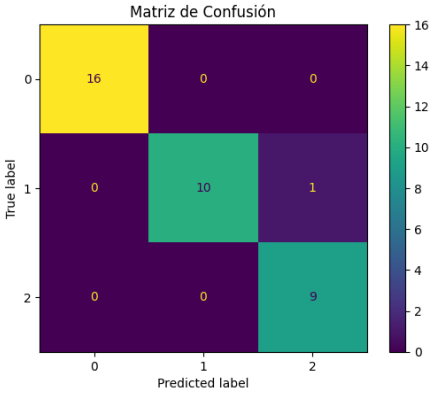


Figure 25: Accuracy

Ahora pasaremos al tercer modelo de red neuronal, que presenta una arquitectura más compleja en comparación con el modelo anterior. Esta red consta de cinco capas, en las que todas las capas ocultas emplean la función de activación tanh para capturar relaciones no lineales en los datos. Por su parte, la capa de salida utiliza la función softmax, lo que permite generar probabilidades para la clasificación. Una característica distintiva de este modelo es la incorporación del optimizador de descenso por gradiente con momentum, lo que contribuye a mejorar significativamente el rendimiento obtenido.

Modelo 3:



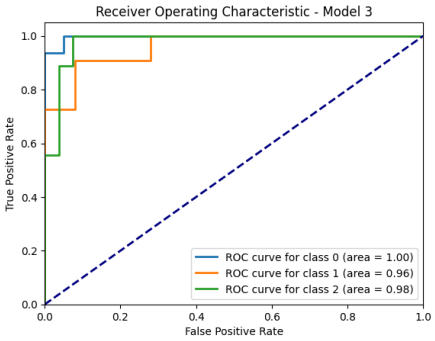


Figure 26: Confusion Matrix Model 1

Figure 27: ROC Curve Model 3

El modelo 3 demuestra un desempeño sólido, con la mayoría de las predicciones correctas reflejadas en los valores altos de la diagonal de la matriz.

La clase 0 y la clase 2 se clasificaron perfectamente, sin errores. La clase 1 mostró un único error, donde un dato fue clasificado incorrectamente como clase 2. En general, el modelo mantiene un buen equilibrio entre las clases, indicando que los datos están adecuadamente representados.

La curva ROC del modelo muestra un excelente rendimiento, con un área bajo la curva (AUC) de 1.00 para la clase 0, lo que indica una clasificación perfecta. Las clases 1 y 2 también alcanzan valores de AUC altos, de 0.96 y 0.98 respectivamente, lo que refleja una alta capacidad de discriminación del modelo. Las curvas cercanas al punto superior izquierdo (0,1) confirman que el modelo mantiene un buen equilibrio entre la tasa de verdaderos positivos y la tasa de falsos positivos.

5 Conclusiones

Este estudio se centró en la implementación y evaluación de redes neuronales para la clasificación de tres conjuntos de datos: Iris, Digits y Wine. A lo largo del trabajo, se desarrollaron diversos modelos con distintas arquitecturas y funciones de activación, analizando su rendimiento en términos de precisión y capacidad de generalización.

Los resultados revelaron que un equilibrio adecuado entre la complejidad del modelo y su rendimiento es esencial. En el caso del conjunto de datos Iris, modelos más simples lograron altas precisiones, destacando que una arquitectura menos compleja puede ser efectiva en conjuntos de datos bien estructurados. Sin embargo, la introducción de más capas y parámetros en los modelos más complejos no siempre se tradujo en mejoras en la precisión, lo que sugiere un riesgo de sobreajuste.

En el análisis del conjunto de datos Digits, se observó que la complejidad de la arquitectura puede permitir que los modelos aprendan patrones más sofisticados. No obstante, también se evidenció que una mayor complejidad no garantiza una mejora en la precisión, lo que pone de relieve la importancia de la selección y ajuste de hiperparámetros adecuados.

Finalmente, en el conjunto de datos Wine, los modelos demostraron una notable capacidad de clasificación, y el modelo más complejo alcanzó un rendimiento sobresaliente. Esto enfatiza el papel crucial de la arquitectura y las funciones de activación en el aprendizaje efectivo de relaciones complejas entre las características de los datos.

En conjunto, estos hallazgos subrayan la necesidad de una estrategia reflexiva en el diseño de modelos de redes neuronales, donde la arquitectura debe ser elegida en función de las características específicas del conjunto de datos y del problema a resolver. Este enfoque permite optimizar la precisión y la capacidad de generalización de los modelos, contribuyendo así a un avance significativo en el campo del aprendizaje automático.

6 Trabajo Futuro

En este proyecto, hemos desarrollado un motor de redes neuronales modular que permite la implementación de arquitecturas personalizadas y una experimentación flexible. Sin embargo, existen varias áreas en las que el trabajo puede ampliarse y mejorarse en futuros desarrollos:

- **Optimización de Desempeño:** Si bien el modelo es funcional, el rendimiento puede optimizarse aún más mediante el uso de técnicas avanzadas de optimización y paralelización, como el uso de procesamiento en GPU.
- **Incorporación de Técnicas de Regularización:** Añadir regularización, como la normalización batch, el dropout o la penalización de pesos, podría mejorar la capacidad general de generalización del modelo y reducir el sobreajuste en conjuntos de datos complejos.
- **Soporte para Redes Neuronales Profundas:** Actualmente, el motor está diseñado para redes de tamaño moderado. En el futuro, se podría adaptar el diseño para soportar arquitecturas de redes neuronales profundas, como redes convolucionales (CNN) y redes recurrentes (RNN), que requieren estructuras y operaciones específicas.
- **Evaluación y Benchmarking Ampliado:** Se podrían realizar evaluaciones de rendimiento en un conjunto más amplio de conjuntos de datos y problemas de diferentes dominios, como clasificación de imágenes, procesamiento de texto y series temporales. Esto proporcionaría una visión más completa de las capacidades y limitaciones del motor.
- **Interfaz de Usuario y Visualización:** Implementar una interfaz gráfica o un tablero de control podría facilitar la visualización de métricas de rendimiento y el seguimiento del progreso durante el entrenamiento, permitiendo a los usuarios ajustar parámetros en tiempo real.
- **Compatibilidad con Otros Lenguajes de Programación:** Una extensión interesante sería hacer que el motor sea compatible con otros lenguajes de programación mediante la creación de API o interfaces que permitan la integración con aplicaciones escritas en Python, Java, o C++.

En resumen, estos elementos representan posibles vías de desarrollo para mejorar y expandir el alcance del motor de redes neuronales, haciendo que sea más robusto y accesible para usuarios en una variedad de aplicaciones.

7 Bibliografía

References

[1] GitHub Repository, *ModularNNEngine*, https://github.com/adoojeda/ModularNNEngine.

[2] OpenAI, *OpenAI: AI Research and Deployment*, https://www.openai.com.

[3] Google DeepMind, *Google: Gemini AI Model*, https://deepmind.google/technologies/gemini/.

[4] Stack Overflow, https://stackoverflow.com.

[5] SciTools, *Iris Dataset*, https://scitools-iris.readthedocs.io/en/stable/.

[6] Scikit-learn, *Digits Dataset*, https://scikit-learn.org/stable/modules/generated/sklearn.datasets.load_digits.html.

[7] Scikit-learn, *Wine Dataset*, https://scikit-learn.org/1.5/modules/generated/sklearn.datasets.load_wine.html.

[8] Backpropagation, *Retropropagación en redes neuronales*, https://www.geeksforgeeks.org/backpropagation-in-neural-network/.