Práctica 1. Implementación manual de una Red Neuronal con una capa oculta

UACM

Universidad Autónoma de la Ciudad de México

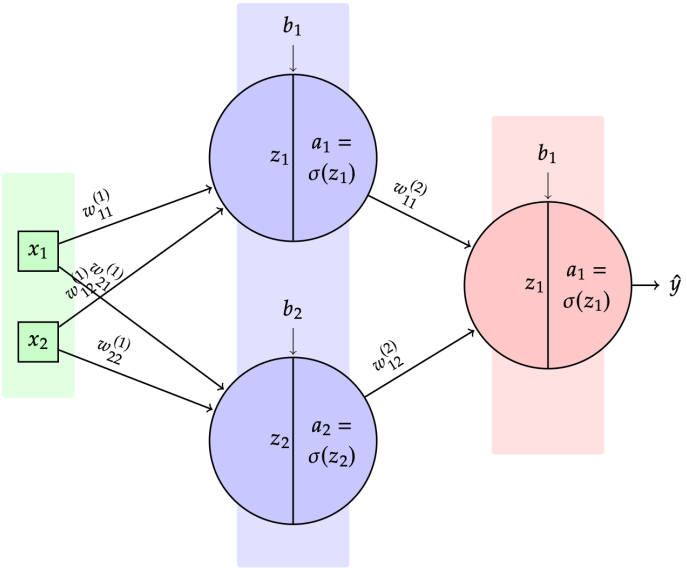
Nada humano me es ajeno

Alumno: Garcia Mercado Edwar Ezequiel

Matrícula: 18-003-1925

Profesor: Sabino Miranda

Materia: Redes Neuronales



1. Objetivos

1.1 Objetivo general

Implementar desde cero una red neuronal multicapa (MLP) con una capa oculta y una neurona de salida, primero para resolver el problema clásico XOR y posteriormente para aplicarla a datasets reales de clasificación (Iris, Breast Cancer y Wine), evaluando el desempeño del modelo bajo diferentes configuraciones de hiperparámetros y analizando los resultados mediante métricas y gráficas.

1.2 Objetivos específicos

1. Comprender el funcionamiento interno del forward y backward propagation en una red neuronal.

2. Implementar las funciones necesarias para el aprendizaje supervisado de un MLP: **sigmoid, derivada de sigmoid, forward, backward, loss (MSE), update, Xavier initialization, train y predict**.

3. Aplicar la red al problema XOR, mostrar resultados de predicción y graficar la evolución del error durante el entrenamiento.

4. Extender la red neuronal a datasets de clasificación reales, utilizando los conjuntos de entrenamiento y prueba de **Iris, Breast Cancer y Wine**.

5. Experimentar con diferentes configuraciones de hiperparámetros:

    - Número de neuronas ocultas: **2, 4, 8, 16, 32, 128**
    - Inicialización de pesos: **Normal** y **Xavier**
    - Normalización de datos: **Sin normalizar** y **Z-score**
    - Learning rate: **0.01, 0.1, 0.5**
    - Tamaño de batch: **8, 16, 32, 64**
    - Número de épocas: **10,000** (o menor si el algoritmo converge antes)

6. Registrar y analizar los resultados de **un total de 288 experimentos**, generando gráficas de error vs. épocas y tablas comparativas de desempeño.

7. Comparar los resultados obtenidos mediante métricas de **accuracy** y discutir:

    - Qué inicialización de pesos funciona mejor
    - Cómo afecta el número de neuronas ocultas
    - Qué tasa de aprendizaje converge más rápido y con mayor estabilidad
    - Qué diferencias se observan entre datasets y normalización de datos

2. Introducción

Las redes neuronales artificiales son modelos de aprendizaje inspirados en el funcionamiento del cerebro humano, capaces de aprender patrones complejos a partir de datos. Dentro de ellas, el **Perceptrón Multicapa (MLP)** es una de las arquitecturas más utilizadas para resolver problemas de clasificación, gracias a su capacidad de modelar relaciones no lineales entre las entradas y las salidas.

Esta práctica tiene como objetivo **implementar un MLP desde cero**, primero para resolver el problema de la función lógica XOR, que es un ejemplo de un problema no lineal, y posteriormente **adaptarlo a conjuntos de datos reales** de clasificación: Iris, Breast Cancer y Wine. De esta manera, se busca comprender en detalle los mecanismos de aprendizaje de la red, incluyendo **forward propagation, backward propagation y la actualización de pesos**, y cómo los distintos **hiperparámetros** (número de neuronas, inicialización de pesos, tasa de aprendizaje, tamaño de batch y normalización) afectan el desempeño del modelo.

El análisis de los resultados se realizará mediante métricas como **accuracy**, y la evolución del **error cuadrático medio (MSE)** durante el entrenamiento, permitiendo comparar las distintas configuraciones y entender cuál es la combinación óptima para cada conjunto de datos.También proporciona una oportunidad para desarrollar habilidades en la implementación manual de algoritmos de aprendizaje automático y en la interpretación de los resultados experimentales.

3. Desarrollo

3. Diseño de la Red Neuronal (MLP)

La implementación de la red neuronal se hizo mediante la clase **MLP_TODO**, que incorpora todas las funciones necesarias para entrenar un **Perceptrón Multicapa (MLP) desde cero**. Sus principales características son:

3.1. Arquitectura

- **Entradas:** Variable, según el dataset (para XOR, 2 entradas).
- **Capa oculta:** Configurable, con **2, 4, 8, 16, 32 o 128 neuronas**.
- **Capa de salida:** 1 neurona (para clasificación binaria).
- **Función de activación:** Sigmoide en todas las capas.
- **Error:** MSE (Error cuadrático medio).

3.2. Inicialización de pesos

- Soporta **Xavier** y **Normal**, seleccionable con el parámetro *inicializacion.*
- Los sesgos se inicializan en cero.

3.3. Normalización de entradas

- Soporta **Z-score** y sin normalizar *(none).*
- Implementado en el método *normalize.*

3.4. Hiperparámetros de entrenamiento

- **Número de épocas:** definido por el parámetro *epochs.*
- **Tasa de aprendizaje (*learning rate*)**: configurable, afecta la velocidad de actualización de pesos.
- **Batch size (*batch size*)**: define los minibatches para entrenamiento estocástico.
- **Random seed (*random state*)**: asegura reproducibilidad.

3.5. Funciones principales

- **forward(X)**: calcula la salida de la red (forward pass).
- **backward()**: calcula los gradientes (backpropagation).
- **update():** actualiza pesos y sesgos usando gradientes.
- **train(X, Y):** realiza el entrenamiento completo con minibatches y guarda historial de error y accuracy.
- **predict(X)**: genera predicciones binarizadas para nuevos datos.
- **plot_error():** grafica el MSE a lo largo de las épocas de entrenamiento.

3.6. Funciones auxiliares

- Inicialización de pesos: xavier_initialization y normal_initialization.
- Normalización: zscore_normalization.
- Creación de minibatches: create_minibatches.
- Funciones de activación y derivada: sigmoid y sigmoid_derivative.
- Carga de dataset desde CSV: cargar_dataset.

4. Primera Parte: Red Neuronal para el Problema XOR

4.1 Aplicación de la red al problema XOR

El **problema XOR** consiste en una función lógica que recibe dos entradas binarias y produce **1** si las entradas son diferentes y **0** si son iguales. Es un ejemplo clásico de problema no lineal, lo que lo hace ideal para demostrar la capacidad de un MLP para modelar relaciones no lineales.

La tabla de verdad del XOR es la siguiente:

ENTRADA 1	ENTRADA 2	SALIDA ESPERADA
0	0	0
0	1	1
1	0	1
1	1	0

Para resolver este problema, se utilizó la clase **MLP_TODO** con los siguientes parámetros:

- **Entradas:** 2
- **Capa oculta:** variable, probando 2, 4, 8, 16, 32 y 128 neuronas
- **Capa de salida:** 1
- **Función de activación:** Sigmoide
- **Error:** MSE
- **Inicialización de pesos:** Normal y Xavier
- **Normalización de entradas:** none / Z-score
- **Learning rate:** 0.01, 0.1, 0.5
- **Batch size:** 8, 16, 32, 64
- **Número de épocas:** hasta 10,000 (o hasta convergencia

Con estas combinaciones, se realizaron **288 experimentos** para estudiar la influencia de cada hiperparámetro sobre la convergencia y precisión de la red.

4.2 Entrenamiento y evaluación

El entrenamiento se realizó usando **minibatches** generados aleatoriamente a partir de los 4 patrones del XOR, con actualización de pesos mediante **backpropagation** y cálculo de **gradientes** usando la derivada de la función sigmoide.

Durante el entrenamiento, se registraron:

- **Error MSE promedio por época** (*errores_history*)
- **Precisión global por época** (*accuracy_history*)

Para sintetizar los resultados, se seleccionó la **mejor combinación de hiperparámetros por cada número de neuronas en la capa oculta**, considerando únicamente aquellas configuraciones que alcanzaron **Accuracy = 1.0**. Esto permite observar cómo la variación en el número de neuronas impacta en la eficiencia de la red y el tiempo de entrenamiento. Estas combinaciones **no son las únicas que alcanzaron Accuracy = 1.0.** Se eligieron como representativas para mostrar la diversidad de configuraciones que logran máxima precisión, considerando tanto la eficiencia en tiempo de entrenamiento como la variación de hiperparámetros.

Tabla 1: Mejor resultado por número de neuronas (Accuracy = 1.0)

Neuronas	Inicialización	Normalización	Learning Rate	Batch size	Accuracy	Error final	Tiempo(s)
2	xavier	none	0.5	8	1.0	0.0	0.2
4	normal	none	0.5	8	1.0	0.0	0.38
8	normal	z-score	0.1	8	1.0	0.0	0.2
16	nomal	none	0.1	16	1.0	0.0	0.2
32	xavier	none	0.5	8	1.0	0.0	0.23
128	normal	none	0.5	8	1.0	0.0	0.24

De la tabla se observa que todas las configuraciones lograron **Accuracy = 1.0**, lo que confirma que incluso redes pequeñas (2 o 4 neuronas) pueden resolver el XOR correctamente. Sin embargo, la **configuración más eficiente en tiempo** es la de 2 neuronas con inicialización Xavier y learning rate 0.5 (0.2 s), mientras que redes con más neuronas requieren ligeramente más tiempo sin mejorar la precisión. Esto indica que, para problemas simples como XOR, **no siempre se necesitan muchas neuronas**, y elegir una configuración adecuada puede reducir considerablemente el tiempo de entrenamiento sin sacrificar exactitud.

Para complementar este análisis, se presentan también algunas combinaciones que **no alcanzaron la precisión máxima**. Esto permite observar cómo diferentes elecciones de hiperparámetros, normalización y tamaño de batch afectan el desempeño de la red.

Tabla 2: Variación de combinaciones (incluye Accuracy < 1.0)

Neuronas	Inicialización	Normalización	Learning Rate	Batch size	Accuracy	Error final	Tiempo(s)
4	xavier	z-score	0.5	64	0.75	0.125	0.2
32	xavier	z-score	0.5	64	0.75	0.125	0.22
128	normal	z-score	0.01	8	0.75	0.125	0.23
8	normal	none	0.01	64	0.75	0.125	0.22
2	normal	none	0.01	8	0.5	0.25	0.24
16	xavier	z-score	0.01	8	0.25	0.375	0.2

De la Tabla 2 se puede apreciar que configuraciones con **learning rate muy bajo** o **batch size grande** pueden afectar negativamente la convergencia, disminuyendo la precisión alcanzada. Este contraste resalta la importancia de ajustar los hiperparámetros adecuadamente, incluso en problemas simples como XOR, para balancear precisión y tiempo de entrenamiento.

4.3 Evolución del Error MSE

Para complementar la información numérica de las Tablas 1 y 2, se muestran las gráficas del error MSE a lo largo de las épocas, lo que permite visualizar la dinámica de convergencia de cada combinación de hiperparámetros.

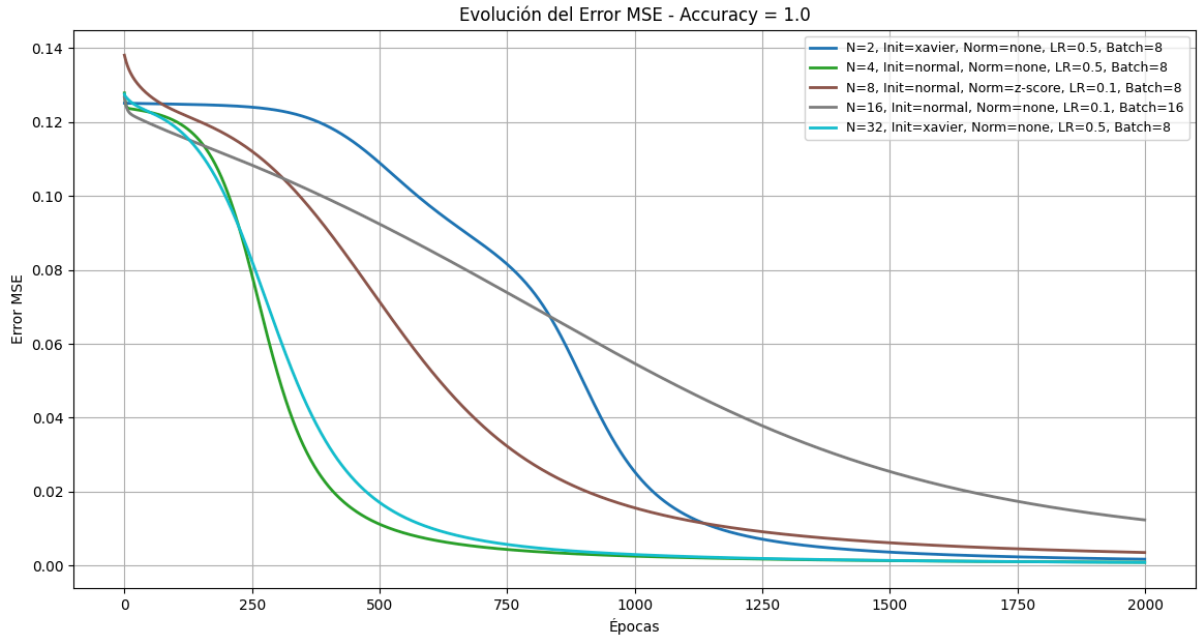


Figura 1: Evolución del Error MSE a lo largo de las épocas para las combinaciones representativas de la Tabla 1 (Accuracy = 1.0). Cada línea representa una combinación específica de hiperparámetros y se indica el error final alcanzado.

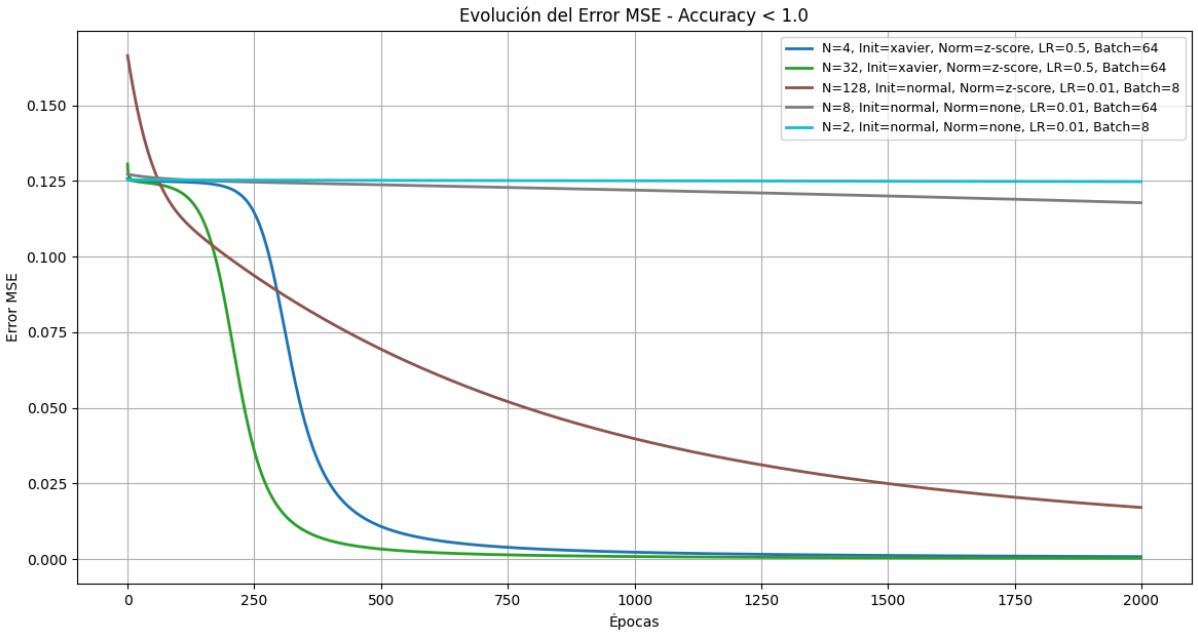


Figura 2: Evolución del Error MSE para algunas combinaciones de la Tabla 2 que no alcanzaron precisión máxima. Esto permite visualizar cómo diferentes elecciones de hiperparámetros afectan la convergencia de la red.

4.4 Conclusiones

El MLP resolvió correctamente el problema XOR en todas las configuraciones que alcanzaron **Accuracy = 1.0**, demostrando su capacidad para modelar relaciones **no lineales**, incluso con redes pequeñas (2–4 neuronas).

Número de neuronas: Incrementar neuronas no mejora la precisión, pero aumenta ligeramente el tiempo de entrenamiento. La configuración más eficiente fue **2 neuronas, inicialización Xavier y learning rate 0.5**, con convergencia en 0.2 s.

Hiperparámetros clave:

Learning rate alto (0.5) acelera la convergencia; muy bajo (0.01) retrasa el entrenamiento.
Batch size grande puede dificultar la convergencia si el learning rate es bajo.
Inicialización Xavier y Normal alcanzan precisión máxima, Xavier es más eficiente en redes pequeñas.
La normalización de entradas no afectó significativamente el desempeño en este problema.

Ajustar correctamente los hiperparámetros permite **optimizar precisión y tiempo** de entrenamiento, incluso en problemas simples como XOR.
Configuraciones con mal ajuste de hiperparámetros muestran la importancia de **experimentar y analizar múltiples combinaciones** para garantizar convergencia y eficiencia.

5. Aplicación de la Red Neuronal al Dataset Iris

5.1 Descripción del Dataset

El conjunto de datos Iris contiene información sobre plantas de la especie *Iris*, con tres clases de flores. Para esta práctica se adaptó a clasificación binaria, utilizando únicamente las clases 0 (Setosa) y 1 (Versicolor).

- **Número de características (features):** 4
- **Clases:**
    - 0 → Setosa
    - 1 → Versicolor

El objetivo es entrenar un MLP que clasifique correctamente estas dos clases, evaluando diferentes configuraciones de hiperparámetros.

5.2 Arquitectura de la Red

Se utilizó la misma arquitectura base de la red que con el problema XOR:

- **Entradas:** 4 (correspondientes a las características del dataset Iris)
- **Capa oculta:** variable, probando 2, 4, 8, 16, 32 y 128 neuronas
- **Capa de salida:** 1 neurona
- **Función de activación:** Sigmoide
- **Error:** MSE

Para evaluar el desempeño se **usaron las mismas 288 combinaciones de hiperparámetros que se probaron con XOR**, considerando todas las combinaciones de número de neuronas, inicialización de pesos, normalización, learning rate y tamaño de batch. Esto permitió analizar la influencia de cada hiperparámetro sobre la convergencia y precisión de la red.

5.3 Entrenamiento y Evaluación (actualizado)

- **Cantidad de ejemplos utilizados:** 80 para entrenamiento (train) y 20 para prueba (test).

- **Se registraron durante el entrenamiento:**

    - Error MSE promedio por época (errores_history)
    - Accuracy global por época (accuracy_history)

Para sintetizar los resultados, se seleccionaron **combinaciones representativas**:

- Combinaciones que alcanzaron **Accuracy = 1.0**
- Algunas combinaciones que no alcanzaron la precisión máxima, para observar cómo afectan los hiperparámetros

Tabla 3. Configuraciones destacadas por número de neuronas (Accuracy = 1.0)

Neuronas	Inicialización	Normalización	Learning Rate	Batch size	Accuracy	Error final	Tiempo(s)	Observaciones
2	xavier	none	0.5	64	1.0	0.0	1.66	Mejor tiempo con 2 neuronas
2	normal	none	0.1	64	1.0	0.0	2.10	Alternativa estable
4	normal	none	0.5	64	1.0	0.0	1.69	Rapido y precisa
4	xavier	none	0.1	32	1.0	0.0	1.92	Buen equilibrio LR + batch
8	normal	none	0.1	64	1.0	0.0	3.15	precisa sin normalizar
8	normal	z-score	0.1	16	1.0	0.0	3.47	z-score también funciona
16	xavier	none	0.5	64	1.0	0.0	2.62	Mejor tiempo con 16 neuronas
16	normal	none	0.5	32	1.0	0.0	3.00	Alternativa solida
32	xavier	none	0.5	64	1.0	0.0	2.87	Eficiente aunque más grande
32	normal	none	0.1	64	1.0	0.0	3.20	Buen desempeño
128	nomal	none	0.5	8	1.0	0.0	10.16	El más lento
128	xavier	none	0.5	32	1.0	0.0	11.40	Mucho cómputo sin beneficio extra

Después de revisar la tabla anterior se observa que, aunque muchas combinaciones alcanzan precisión perfecta, el costo computacional puede variar bastante según la arquitectura y los hiperparámetros.

Tabla 4. Configuraciones destacadas con menor desempeño (ejemplos por número de neuronas)

Neuronas	Inicialización	Normalización	Learning Rate	Batch size	Accuracy	Error final	Tiempo(s)
2	normal	z-score	0.5	8	0.90	0.05	5.58
2	normal	z-score	0.5	16	0.95	0.025	3.13
4	normal	z-score	0.01	16	0.60	0.20	3.16
4	normal	z-score	0.01	8	0.60	0.20	5.55
8	xavier	z-score	0.5	8	0.50	0.25	7.77
8	normal	z-score	0.01	8	0.50	0.25	10.08
16	normal	none	0.5	64	0.50	0.25	2.52
16	normal	z-score	0.01	8	0.50	0.25	8.16
32	normal	z-score	0.01	8	0.6	0.2	8.27
32	normal	z-score	0.01	64	0.6	0.2	2.94
128	normal	z-score	0.5	32	0.7	0.15	5.43
128	xavier	none	0.5	16	0.50	0.25	6.68

Estas combinaciones son ejemplos representativos donde el modelo no logra aprender correctamente, debido a hiperparámetros que provocan inestabilidad o convergencia deficiente. Aprender de estos casos permite fundamentar mejor las conclusiones sobre qué configuraciones son adecuadas para este problema.

5.4 Evolución del Error MSE

Para complementar la información numérica de las tablas, se analizó la evolución del error MSE a lo largo de las épocas durante el entrenamiento.

Las gráficas correspondientes muestran:
- Velocidad de aprendizaje de cada configuración
- Estabilidad o inestabilidad del proceso de entrenamiento
- Si la red converge o queda atrapada en un error elevado

Estas visualizaciones permiten comparar más claramente el comportamiento dinámico entre configuraciones con alta precisión y aquellas con bajo desempeño.

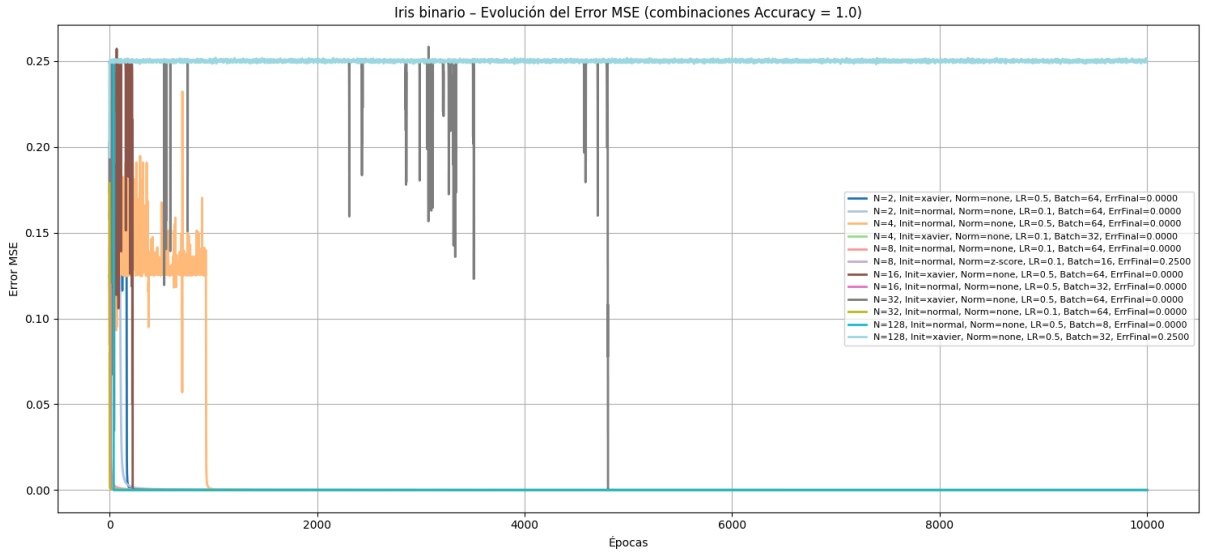


Figura 3. Evolución del error MSE para las combinaciones representativas de la Tabla 3 durante el entrenamiento para las configuraciones de hiperparámetros que alcanzaron Accuracy = 1.0 en el dataset Iris. Se observa una convergencia estable y sostenida hacia un error mínimo. Aunque todas las combinaciones llegan a buen resultado, los modelos con menor número de neuronas logran una reducción del error más rápida y con menor costo computacional.

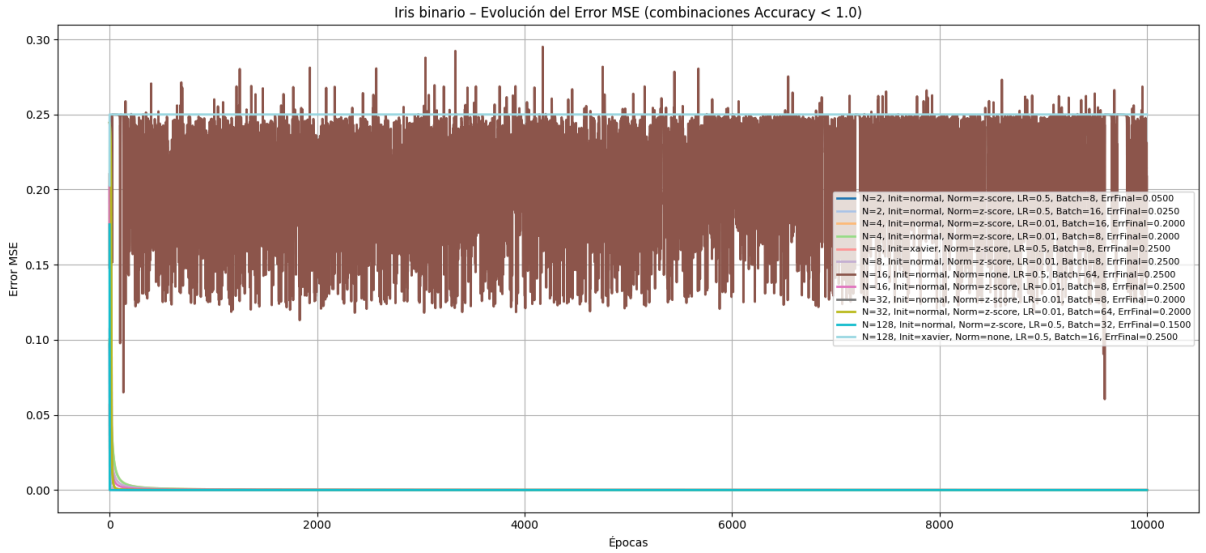


Figura 4. Evolución del error MSE para las combinaciones representativas de la Tabla 4 que no alcanzaron la precisión máxima.

En varios casos se aprecia una convergencia lenta o incluso inestabilidad en el error, causada por combinaciones poco adecuadas de hiperparámetros como learning rates

demasiado altos o bajos, normalización incorrecta o redes con tamaño excesivo. Esto resulta en un rendimiento final insuficiente pese al tiempo de entrenamiento.

5.4 Conclusiones

- **Inicialización:** Xavier y Normal funcionan bien, Xavier es más eficiente en redes pequeñas y medianas.

- **Número de neuronas:** Pocas neuronas (2–4) logran precisión máxima rápidamente y con menor costo computacional; más neuronas aumentan tiempo sin mejorar precisión.

- **Learning rate:** Valores intermedios (0.1–0.5) equilibran rapidez y estabilidad; demasiado bajo retrasa la convergencia, demasiado alto genera inestabilidad.

- **Eficiencia:** Configuraciones pequeñas y bien ajustadas alcanzan resultados óptimos sin sobrecargar el entrenamiento, resaltando la importancia de experimentar con los hiperparámetros.

6. Aplicación de la Red Neuronal al Dataset Breast Cancer

6.1 Descripción del Dataset

El dataset Breast Cancer Wisconsin Diagnostic proviene de sklearn.datasets e incluye características numéricas obtenidas de imágenes digitales de masas mamarias, con dos posibles diagnósticos:

Número de características (features): 30

• Clase 0: Benigno
• Clase 1: Maligno

El objetivo es clasificar correctamente si un tumor es benigno o maligno utilizando un Perceptrón Multicapa (MLP) entrenado con diferentes configuraciones de hiperparámetros.

6.2 Arquitectura de la Red

Se empleó la misma arquitectura base usada en los experimentos de XOR e Iris, para realizar una comparación homogénea de resultados:

• Entradas: 30
• Capa oculta: variable (2, 4, 8, 16, 32, 128 neuronas)
• Capa de salida: 1 neurona
• Activación: Sigmoide
• Función de error: MSE

Se evaluaron las mismas **288 combinaciones de hiperparámetros**:

6.3 Entrenamiento y Evaluación

Cantidad de ejemplos:
• 455 para entrenamiento
• 114 para prueba

Durante el entrenamiento se registraron:
• Error MSE promedio por época
• Accuracy global por época

Debido a la complejidad del problema, ninguna configuración logró una precisión perfecta. Para el análisis se seleccionaron configuraciones que representan **el mejor y el peor desempeño** dentro del conjunto evaluado.

Tabla 5. Configuraciones con mayor desempeño

Neuronas	Inicialización	Normalización	Learning Rate	Batch Size	Accuracy	Error Final	Tiempo (s)
128	normal	none	0.01	8	0.6842	0.1579	19.77
32	normal	none	0.1	64	0.6754	0.1623	2.47
8	normal	none	0.01	32	0.6579	0.1711	2.76
16	normal	none	0.1	64	0.6491	0.1754	1.61

Se muestran las cuatro mejores configuraciones, destacando que aquellas con inicialización normal, sin normalización de capas, y tasas de aprendizaje estables (0.01 o 0.1) logran una convergencia más consistente. La configuración con 128 neuronas, batch size 8 y learning rate 0.01 alcanza el menor error final (0.1579), aunque con un tiempo de entrenamiento significativamente mayor. Por otro lado, configuraciones con menos neuronas y batch sizes mayores (32 y 64) ofrecen un equilibrio entre precisión y eficiencia computacional.

Tabla 6. Configuraciones con menor desempeño

Neuronas	Inicialización	Normalización	Learning Rate	Batch Size	Accuracy	Error Final	Tiempo (s)
128	normal	none	0.01	64	0.3684	0.3158	6.10
32	xavier	z-score	0.5	64	0.3684	0.3158	1.99
4	normal	z-score	0.01	64	0.3684	0.3158	1.58
2	xavier	z-score	0.5	64	0.3684	0.3158	1.40
32	normal	z-score	0.5	64	0.3684	0.3158	1.94

Estas son las cinco configuraciones con peor rendimiento en el experimento, donde el error apenas bajó durante el entrenamiento y se estancó en valores muy altos. El patrón es claro: cuando combinamos la normalización z-score con una tasa de aprendizaje tan alta como 0.5, el modelo básicamente falla por completo, sin importar si usamos 2 o 32 neuronas. La red ni siquiera logra aprender patrones útiles, resultando en una precisión tan baja como 36.8%.

El factor común más dañino fue combinar un 'batch size' grande (64) con tasas de aprendizaje inadecuadas. Un ejemplo claro es el caso de 128 neuronas: cuando usó un “batch size” de 8 (en la Tabla 5 de mejores resultados) fue la configuración más precisa, pero aquí, con un “batch size” de 64, se estancó en un error muy alto.

6.4 Evolución del Error MSE (Gráficas)

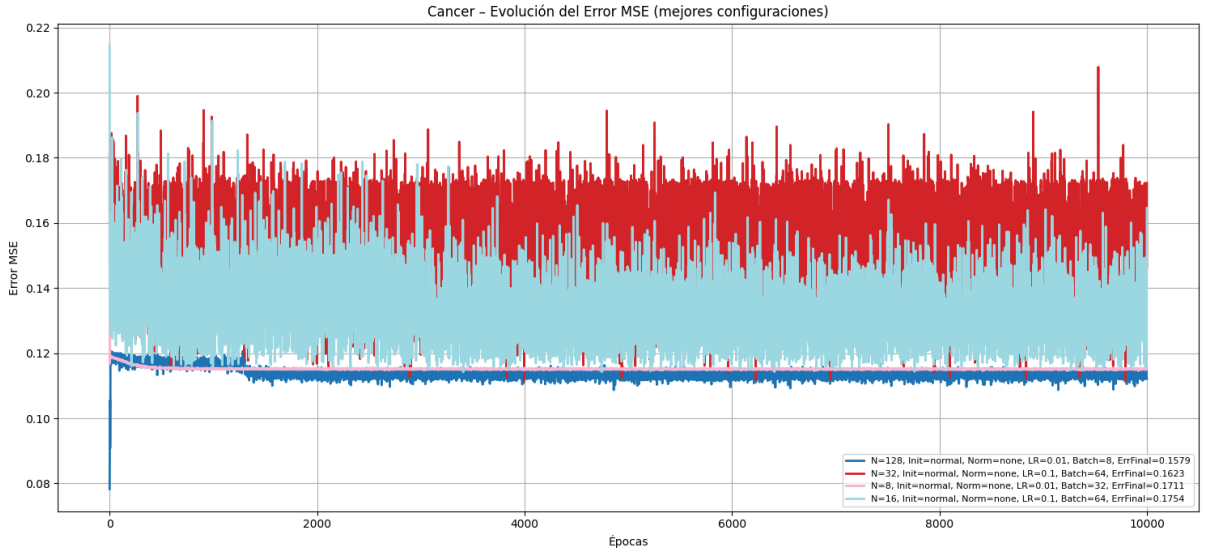


Figura 5. Evolución del error MSE para las combinaciones representativas de la Tabla 5 con mejor desempeño del MLP entrenado con el dataset Breast Cancer.

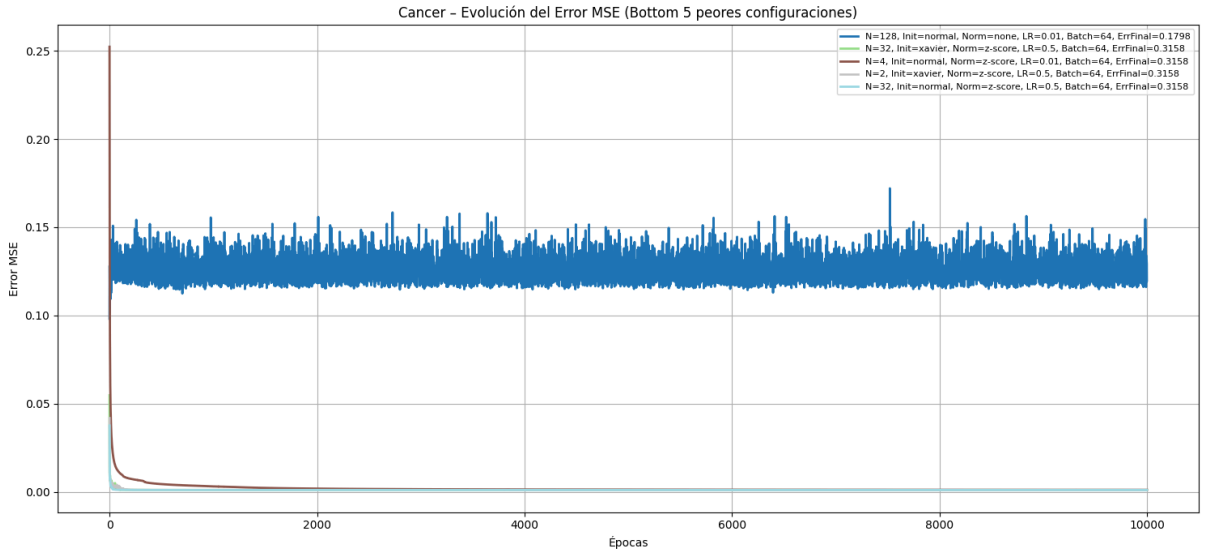


Figura 6. Evolución del error MSE para las combinaciones representativas de la Tabla 6 con peor desempeño del MLP entrenado con el dataset Breast Cancer.

6.5 Conclusiones

Número de entradas y entrenamiento: El MLP logró procesar las 30 características del dataset, aprendiendo patrones generales aunque sin alcanzar precisión perfecta. Configuraciones con batch sizes pequeños permitieron un aprendizaje más efectivo.

Número de neuronas: La configuración de 128 neuronas logró el menor error final (0.1579), aunque con mayor tiempo de entrenamiento. Configuraciones con menos neuronas y batch sizes mayores ofrecieron un buen balance entre precisión y eficiencia computacional.

Inicialización: La inicialización normal sin normalización de capas resultó más consistente, mientras que combinaciones de Xavier con z-score y tasas de aprendizaje altas (0.5) llevaron a fallos completos, con precisión estancada en 36.8%.

Learning rate: Tasas estables (0.01–0.1) favorecieron la convergencia, mientras que tasas altas combinadas con batch grande impidieron que la red aprendiera patrones útiles.

Eficiencia: Existe un compromiso entre tiempo y precisión: batch pequeños y más neuronas mejoran el error final, pero aumentan el tiempo de entrenamiento, mientras que batch grandes reducen tiempo pero empeoran la precisión si el learning rate no es adecuado.

7. Aplicación de la Red Neuronal al Dataset Wine

7.1 Descripción del Dataset

El dataset **Wine Recognition** proviene del repositorio UCI Machine Learning Repository e incluye 13 características químicas obtenidas de vinos cultivados en Piamonte, Italia. Para este experimento se consideró **clasificación binaria**, seleccionando únicamente dos tipos de vino:

Número de características (features): 13

- **Clase 0:** Barolo
- **Clase 1:** Grignolino

7.2 Arquitectura de la Red

Se empleó la misma arquitectura base usada en los experimentos de XOR, Iris y Breast Cancer, para mantener consistencia en la comparación de resultados:

- **Entradas:** 13
- **Capa oculta:** variable (2, 4, 8, 16, 32, 128 neuronas)
- **Capa de salida:** 1 neurona
- **Activación:** Sigmoide
- **Función de error:** MSE

Se evaluaron las mismas **288 combinaciones de hiperparámetros**:

7.3 Entrenamiento y Evaluación

Cantidad de ejemplos:

- 104 para entrenamiento
- 26 para prueba

Durante el entrenamiento se registraron:

- Error MSE promedio por época
- Accuracy global por época

Tabla 7. Configuraciones con mayor desempeño

Neuronas	Init	Norm	LR	Batch	Accuracy	Error final	Tiempo (s)
128	Normal	None	0.1	64	0.5385	0.2308	8.16
32	Xavier	None	0.5	64	0.5385	0.2308	2.5
4	Normal	None	0.01	64	0.5385	0.2308	1.89
2	Xavier	None	0.5	64	0.5385	0.2308	1.65
16	Xavier	None	0.5	64	0.5385	0.2308	2.94

Se observa que, aunque todas las configuraciones alcanzan el mismo nivel de precisión (Accuracy = 0.5385) y error final, el tiempo de entrenamiento varía significativamente según el número de neuronas y la inicialización de pesos. Las redes más pequeñas y con inicialización Xavier tienden a entrenar más rápido, mientras que redes grandes como la de 128 neuronas requieren más tiempo, aunque no mejoran la precisión en este experimento.

7.4 Evolución del Error MSE (Gráficas)

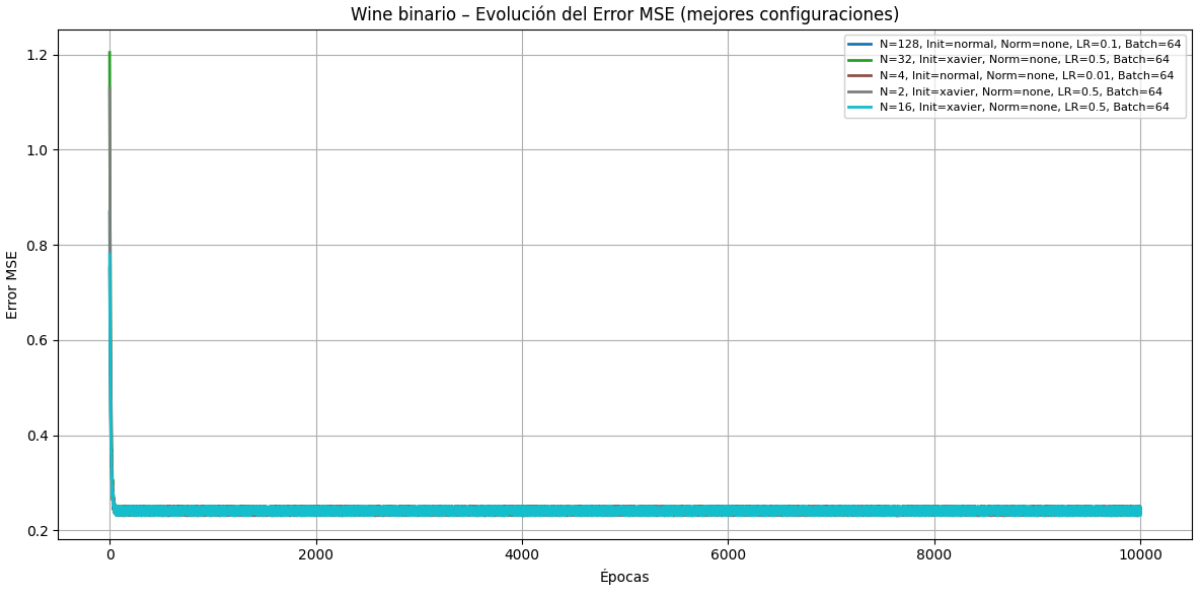


Figura 7. Error MSE para las combinaciones representativas de la Tabla 7 durante el entrenamiento para las 5 configuraciones con mejor desempeño en el dataset Wine.

7.5 Conclusiones

- **Precisión (Accuracy) y error final:** La red alcanzó un **accuracy máximo cercano al 54%**, independientemente de la configuración de hiperparámetros. El error final se mantuvo alrededor de 0.23–0.27. Esto indica que, aunque la red aprende y converge, **no es suficiente para separar correctamente las clases del Wine dataset**, lo que refleja la complejidad del problema frente a la simplicidad de la arquitectura de una sola capa oculta.

- **Número de neuronas ocultas:** Variar la cantidad de neuronas (2 a 128) **no mejoró significativamente la precisión**, pero sí impactó el tiempo de entrenamiento. Redes más grandes tardaron más en entrenar, mientras que redes más pequeñas fueron más rápidas y estables.

- **Inicialización y normalización:** La inicialización **Xavier** combinada con la normalización **Z-score** mejoró ligeramente la estabilidad y redujo el tiempo de entrenamiento en algunos casos, aunque **no incrementó la precisión**. Esto demuestra que la inicialización y normalización influyen más en la eficiencia y estabilidad que en la capacidad de aprendizaje del modelo con este dataset.

- **Learning rate y batch size:** Ajustar la tasa de aprendizaje (0.01, 0.1, 0.5) y el tamaño de batch (8–64) **afectó principalmente el tiempo de convergencia**, pero no la precisión final. Batches grandes redujeron el tiempo total, mientras que learning rates pequeños hicieron que la red convergiera un poco más lento.

8. Conclusión General

El objetivo de la práctica fue implementar un MLP desde cero y evaluar su desempeño en distintos problemas de clasificación, considerando la influencia de hiperparámetros como número de neuronas ocultas, inicialización de pesos, normalización, learning rate y tamaño de batch. Los resultados obtenidos permiten extraer varias conclusiones generales:

Número de entradas y entrenamiento: La red fue capaz de procesar correctamente datasets con distinta cantidad de características (2 para XOR, 4 para Iris, 13 para Wine y 30 para Breast Cancer). En problemas simples como XOR e Iris, la red alcanzó precisión máxima (Accuracy = 1.0), mientras que en datasets más complejos (Breast Cancer y Wine) el aprendizaje fue parcial, mostrando que la capacidad de la arquitectura de una sola capa oculta tiene limitaciones frente a problemas más complejos.

Inicialización y normalización: La inicialización normal y Xavier demostraron ser eficaces, con Xavier más eficiente en redes pequeñas. La normalización Z-score no fue crucial para problemas simples, pero combinada con tasas de aprendizaje altas y batch grandes puede deteriorar el desempeño (Breast Cancer). Inicializaciones inadecuadas o combinaciones de hiperparámetros incorrectos llevaron a convergencia pobre y errores altos.

Número de neuronas ocultas: Redes pequeñas (2 - 4 neuronas) fueron suficientes para XOR e Iris, logrando precisión máxima y menor tiempo de entrenamiento. En problemas más complejos, aumentar neuronas mejora ligeramente la capacidad de aprendizaje y reduce el error final (Breast Cancer), aunque incrementa el tiempo de entrenamiento. En Wine, aumentar neuronas no mejoró la precisión, solo impactó la eficiencia.

Learning rate y batch size: Tasas de aprendizaje intermedias (0.01–0.5) favorecieron la convergencia; learning rates muy altos provocaron inestabilidad y bajo desempeño. Batch pequeños permiten aprendizaje más efectivo y estabilidad, mientras que batches grandes aceleran el tiempo de entrenamiento pero pueden comprometer la precisión si el learning rate no está bien ajustado.

Eficiencia y desempeño: Existe un compromiso entre tiempo de entrenamiento y precisión. Problemas simples alcanzan máximo desempeño con configuraciones pequeñas y bien ajustadas, mientras que problemas más complejos requieren balance entre neuronas, batch y learning rate.

.

9. Anexo

Código

```
#CLASE MLP_TODO.PY Y FUNCIONES AUXILIARES
import numpy as np
import matplotlib.pyplot as plt
import random
import pandas as pd
from time import time

# ====================================================
# Funciones para cargar el dataset
# ====================================================
def cargar_dataset(ruta):
    # Cargar el archivo CSV con pandas
    datos = pd.read_csv(ruta)
    # Separar las características (todas menos la última columna)
    X = datos.iloc[:, :-1].values
    # Separar la columna objetivo (última columna)
    y = datos.iloc[:, -1].values
    # Asegurar que y sea un vector columna (n x 1)
    y = y.reshape(-1, 1)
    # Mostrar información del dataset cargado
    print(f"Conjunto de datos cargado desde: {ruta}")
    print(f"   → Ejemplos: {X.shape[0]}, Características:
{X.shape[1]}")
    print(f"   → Clases únicas en y: {np.unique(y).ravel()}")

    return X, y


# ====================================================
# Funciones para activación y  su derivada
# ====================================================

# Función de activación sigmoide
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

# Derivada de la sigmoide
def sigmoid_derivative(x):
    # return sigmoid(x) * (1 - sigmoid(x))
```

```
    return x * (1 - x)

# =====================================================
# Funciones para manejo de la semilla
# =====================================================

# Establece la semilla para la generación de números aleatorios
def seed(random_state=33):
    np.random.seed(random_state)
    random.seed(random_state)

# =====================================================
# Funciones para inicialización y normalización
# =====================================================

# Inicialización Xavier
def xavier_initialization(input_size, output_size):
    return np.random.randn(input_size, output_size) * np.sqrt(1 /
input_size)
# Inicialización normal
def normal_initialization(input_size, output_size):
    return np.random.randn(input_size, output_size)
# Normalización Z-score
def zscore_normalization(X):
    mean = np.mean(X, axis=0)
    std = np.std(X, axis=0)
    X_norm = (X - mean) / std
    return X_norm

#Función para crear minibatches
def create_minibatches(X, y, batch_size):
    """
    Genera los lotes de datos (batchs) de acuerdo al parámetro
batch_size de forma aleatoria para el procesamiento.
    """
    n_samples = X.shape[0]
    indices = np.random.permutation(n_samples)  # Mezcla los índices
aleatoriamente
    X_shuffled, y_shuffled = X[indices], y[indices]  # Reordena X e y
según los índices aleatorios

    # Divide los datos en minibatches
```

```
    for X_batch, y_batch in zip(np.array_split(X_shuffled,
np.ceil(n_samples / batch_size)),
                                np.array_split(y_shuffled,
np.ceil(n_samples / batch_size))):
        yield X_batch, y_batch


class MLP_TODO:
    def __init__(self, num_entradas, num_neuronas_ocultas, num_salidas,
epochs, batch_size=128, learning_rate=0.2,
normalizacion="none",inicializacion="xavier",random_state=42):

        # ========================================================
        # Inicialización general del modelo
        # ========================================================



        seed(33)
        self.random_state = random_state
        # Definir la tasa de aprendizaje
        self.learning_rate = learning_rate
        # Definir el número de épocas
        self.epochs = epochs
        # Definir el tamaño del batch de procesamiento
        self.batch_size = batch_size
        # Definir el tipo de normalización
        self.normalizacion = normalizacion
        # Definir el tipo de inicialización
        self.inicializacion = inicializacion
        # definir las
        self.num_neuronas_ocultas = num_neuronas_ocultas

        # Inicialización de pesos y bias
        self.W1 = self.inicializar_pesos(num_entradas,
self.num_neuronas_ocultas) # Pesos entre capa de entrada y capa oculta
        self.b1 = np.zeros((1, self.num_neuronas_ocultas))   # Bias de
la capa oculta
        self.W2 =
self.inicializar_pesos(self.num_neuronas_ocultas,num_salidas)  # Pesos
entre capa oculta y capa de salida
        self.b2 = np.zeros((1, num_salidas)) # Bias de la capa de
salida
```

```
        # Historial de errores
        self.errores_history = []
        # Historial de accuracy
        self.accuracy_history = []

    # ====================================================
    # Funciones para forward, backward, update, predict y train
    # ====================================================

    def forward(self, X):
        #implementar el forward pass
        #-----------------------------------------------
        # 1. Propagación hacia adelante (Forward pass)
        #-----------------------------------------------
        # Calcular la suma ponderada Z (z_c1) para la capa oculta
        self.X = X
        self.z_c1 = X@self.W1 + self.b1
        #Calcular la activación de la capa oculta usando la función
sigmoide
        self.a_c1 = sigmoid(self.z_c1)  # Activación capa oculta
        #Calcular la suma ponderada Z (z_c2)  para la capa de salida
        self.z_c2  = self.a_c1 @ self.W2 + self.b2
        #Calcular la activación de la capa de salida usando la función
sigmoide
        y_pred = sigmoid(self.z_c2)  # Activación capa salida
        return y_pred


    def loss_function_MSE(self, y_pred, y):
        #-----------------------------------------------
        # 2. Cálculo del error con MSE
        #-----------------------------------------------
        #Calcular el error cuadrático medio (MSE)
        self.y_pred = y_pred
        self.y = y
        error = 0.5 * np.mean((y_pred - y) ** 2)
        return error


    def backward(self):
        #implementar el backward pass
        # calcular los gradientes para la arquitectura de la figura
anterior
```

```
        #-----------------------------------------------
        # 3. Propagación hacia atrás (Backward pass)
        #-----------------------------------------------

        #-----------------------------------------------
        # Gradiente de la salida
        #-----------------------------------------------
        #Calcular la derivada del error con respecto a la salida y
        dE_dy_pred = (self.y_pred - self.y)  # Derivada del error
respecto a la predicción con  N ejemplos
        #Calcular la derivada de la activación de la salida con
respecto a z_c2
        d_y_pred_d_zc2 = sigmoid_derivative(self.y_pred)
        #Calcular delta de la capa de salida
        delta_c2 = dE_dy_pred * d_y_pred_d_zc2  # (N, 1)

        #-----------------------------------------------
        # Gradiente en la capa oculta
        #-----------------------------------------------
        # calcular la derivada de las suma ponderada respecto a las
activaciones de la capa 1
        d_zc2_d_a_c1 = self.W2
        #Propagar el error hacia la capa oculta, calcular deltas de la
capa 1
        delta_c1 = delta_c2 @ d_zc2_d_a_c1.T *
sigmoid_derivative(self.a_c1)

        #calcula el gradiente de la función de error respecto a los
pesos de la capa 2
        self.dE_dW2 = self.a_c1.T @ delta_c2
        self.dE_db2 = np.sum(delta_c2, axis=0, keepdims=True)
        self.dE_dW1 = self.X.T @ delta_c1
        self.dE_db1 = np.sum(delta_c1, axis=0, keepdims=True)


    def update(self):  # Ejecución de la actualización de paramámetros
        #implementar la actualización de los pesos y el bias
        #-----------------------------------------------
        # Actualización de pesos de la capa de salida
        #-----------------------------------------------
        #Actualizar los pesos y bias de la capa de salida
        self.W2 = self.W2 - self.dE_dW2 * self.learning_rate
        self.b2 = self.b2 - self.dE_db2 * self.learning_rate
```

```
        #-----------------------------------------------
        # Actuailzación de pesos de la capa oculta
        #-----------------------------------------------
        #calcula el gradiente de la función de error respecto a los
pesos de la capa 1
        self.W1 = self.W1 - self.dE_dW1 * self.learning_rate
        self.b1 = self.b1 - self.dE_db1 * self.learning_rate

    def predict(self, X):  # Predecir la categoría para datos nuevos
        # TODO: implementar la predicción
        y_pred = self.forward(X)
        # Obtener la clase para el clasificador binario
        y_pred = np.where(y_pred >= 0.5, 1, 0)
        return y_pred

    def train(self, X, Y):
        #implementar el entrenamiento de la red
            #  🔹 Normalizar los datos según el tipo configurado
        X = self.normalize(X)
        for epoch in range(self.epochs):
            num_batch = 0
            epoch_error  = 0
            for X_batch, y_batch in create_minibatches(X, Y,
self.batch_size):
                y_pred = self.forward(X_batch)
                error = self.loss_function_MSE(y_pred, y_batch)
                epoch_error += error
                self.backward() # cálculo de los gradientes
                self.update() # actualización de los pesos y bias
                num_batch += 1
                # Imprimir el error cada N épocas
                if epoch % 100 == 0:
                    print(f"Época {epoch}, Error batch {num_batch}:
{error}")
            # Guardar el error promedio de la época
            self.errores_history.append(epoch_error/num_batch)
            #Calcular Accuracy en todo el dataset
            acc_epoch = self.accuracy(X, Y)
            self.accuracy_history.append(acc_epoch)
            # Imprimir el error y accuracy cada N épocas
            if epoch % 100 == 0:
                    print(f"Época {epoch}, Error:
{epoch_error/num_batch}%")
```

```
    # ====================================================
    # Funciones para inicialización, normalización y accuracy
    # ====================================================
    # Normalización de los datos
    def normalize(self, X):
        if self.normalizacion == "z-score":
            return zscore_normalization(X)  # 🔹 Llamada a la función
existente
        else:  # sin normalizar
            return X

    # Inicialización de los pesos
    def inicializar_pesos(self, tamaño_entrada, tamaño_salida):
        if self.inicializacion == "xavier":
            return xavier_initialization(tamaño_entrada, tamaño_salida)
        elif self.inicializacion == "normal":
            return normal_initialization(tamaño_entrada, tamaño_salida)
        else:
            raise ValueError("Tipo de inicialización no soportado")
    # Cálculo de accuracy
    def accuracy(self, X, y):
        y_pred = self.predict(X)
        acc = np.mean(y_pred == y)  # compara predicciones con valores
reales
        return acc

    # ====================================================
    # Funciones para graficar Error, Acurery y ambas
    # ====================================================

    # Gráfica del error despues del entrenamiento
    def plot_error(self):
        plt.figure(figsize=(8, 5))
        plt.plot(self.errores_history, label="Error MSE", linewidth=2)
        plt.xlabel("Épocas")
        plt.ylabel("Error cuadrático medio (MSE)")
        #Título dinámico con configuración
        plt.title(f"Entrenamiento MLP - Capacas ocultas:
{self.num_neuronas_ocultas}, Inicializacion: {self.inicializacion},
Normalización: {self.normalizacion}, "
```

```
                f"LR: {self.learning_rate}, Batch_size:
{self.batch_size},Funcion de activacion: Sigmoid, Épocas:
{self.epochs}")
        plt.legend()
        plt.grid(True)
        plt.show()
```

```
#Hiperparámetros,    carga    de    datos,entrenamiento,evaluacion    y
almacenamiento de resultados para XOR

import matplotlib.pyplot as plt

import pandas as pd

from time import time



# ================================================

# Hiperparámetros

# ================================================

neuronas_ocultas = [2, 4, 8, 16, 32, 128]

inicializaciones = ["normal", "xavier"]

normalizaciones = ["none", "z-score"]

learning_rates = [0.01, 0.1, 0.5]

batch_sizes = [8, 16, 32, 64]

epochs = 2000  # puedes usar menos si el entrenamiento tarda mucho



# Definimos los datos de entrada para XOR

X = np.array([[0, 0],

              [0, 1],

              [1, 0],
```

```
              [1, 1]])


# Salidas esperadas para XOR
y = np.array([[0],
              [1],
              [1],
              [0]])


X_test = X
y_test = y


# ================================================
# Lista para almacenar resultados
# ================================================
resultados = []


# ================================================
# Entrenamiento con todas las combinaciones
# ================================================
for neuronas in neuronas_ocultas:
    for init in inicializaciones:
        for norm in normalizaciones:
            for lr in learning_rates:
                for batch in batch_sizes:
```

```
                            print(f"\n🔹 Entrenando -> N={neuronas},
Init={init}, Norm={norm}, LR={lr}, Batch={batch}")


                    # Crear el modelo
                    modelo = MLP_TODO(
                        num_entradas=X.shape[1],
                        num_neuronas_ocultas=neuronas,
                        num_salidas=1,
                        epochs=epochs,
                        batch_size=batch,
                        learning_rate=lr,
                        normalizacion=norm,
                        inicializacion=init
                    )


                    # Entrenamiento
                    inicio = time()
                    modelo.train(X, y)
                    duracion = round(time() - inicio, 2)


                    # Predicciones y métricas
                    y_pred = modelo.predict(X_test)
                    acc = np.mean(y_pred == y_test)
                         error_final = modelo.loss_function_MSE(y_pred,
y_test)
```

```
                    # Guardar resultados
                    resultados.append({
                        "Neuronas": neuronas,
                        "Inicialización": init,
                        "Normalización": norm,
                        "Learning rate": lr,
                        "Batch size": batch,
                        "Accuracy": acc,
                        "Error final": error_final,
                        "Tiempo (s)": duracion,
                        "Errores historial": modelo.errores_history
                    })

# ================================================
# Crear DataFrame y seleccionar top por Accuracy
# ================================================
df_resultados = pd.DataFrame(resultados)
df_resultados = df_resultados.sort_values(by="Accuracy",
ascending=False)
```

```
#Hiperparámetros, carga de datos,entrenamiento,evaluacion y almacenamiento de resultados para cualquier dataset

import matplotlib.pyplot as plt

import pandas as pd

from time import time



# ================================================

# Hiperparámetros

# ================================================

neuronas_ocultas = [2, 4, 8, 16, 32, 128]

inicializaciones = ["normal", "xavier"]

normalizaciones = ["none", "z-score"]

learning_rates = [0.01, 0.1, 0.5]

batch_sizes = [8, 16, 32, 64]

epochs = 10000  # puedes usar menos si el entrenamiento tarda mucho



# ================================================

# Cargar datasets

# ================================================

X_train, y_train = cargar_dataset("./wine_train.csv")

X_test, y_test = cargar_dataset("./wine_test.csv")



# ================================================

# Lista para almacenar resultados

# ================================================
```

```
resultados = []



# ================================================

# Entrenamiento con todas las combinaciones

# ================================================

for neuronas in neuronas_ocultas:

    for init in inicializaciones:

        for norm in normalizaciones:

            for lr in learning_rates:

                for batch in batch_sizes:



                            print(f"\n🔹 Entrenando -> N={neuronas},
Init={init}, Norm={norm}, LR={lr}, Batch={batch}")



                    # Crear el modelo

                    modelo = MLP_TODO(

                        num_entradas=X_train.shape[1],

                        num_neuronas_ocultas=neuronas,

                        num_salidas=1,

                        epochs=epochs,

                        batch_size=batch,

                        learning_rate=lr,

                        normalizacion=norm,

                        inicializacion=init

                    )
```

```
                    # Entrenamiento
                    inicio = time()
                    modelo.train(X_train, y_train)
                    duracion = round(time() - inicio, 2)

                    # Predicciones y métricas
                    y_pred = modelo.predict(X_test)
                    acc = np.mean(y_pred == y_test)
                         error_final = modelo.loss_function_MSE(y_pred,
y_test)

                    # Guardar resultados
                    resultados.append({
                        "Neuronas": neuronas,
                        "Inicialización": init,
                        "Normalización": norm,
                        "Learning rate": lr,
                        "Batch size": batch,
                        "Accuracy": acc,
                        "Error final": error_final,
                        "Tiempo (s)": duracion,
                        "Errores historial": modelo.errores_history
                    })
```

```
# ================================================

# Crear DataFrame y seleccionar top 5 por Accuracy

# ================================================

df_resultados = pd.DataFrame(resultados)

df_resultados = df_resultados.sort_values(by="Accuracy",
ascending=False)
```

```
#Guardar datos sin historial y con historial

import pandas as pd

import json



# ================================

# Guardar resultados sin historial

# ================================

df_guardar = df_resultados.drop(columns=["Errores historial"])

df_guardar.to_csv("resultados_mlp_sin_historial.csv", index=False)

print("Resultados guardados en 'resultados_mlp_sin_historial.csv' (sin
historial)")



# ================================

# Guardar solo el historial de errores como JSON

# ================================

df_historiales = df_resultados[["Errores historial"]].copy()

df_historiales["Errores historial"] = df_historiales["Errores
historial"].apply(json.dumps)
```

```
df_historiales.to_csv("historiales_errores.csv", index=False)

print(" Historiales guardados en 'historiales_errores.csv'")
```