# Motor de Redes Neuronales

Luis Perera Pérez
Carlos Francisco Suárez Sauca
Nicolás Trujillo Estévez

Curso 2024/25

Este trabajo presenta el desarrollo de un motor de redes neuronales que implementa diferentes funciones de activación y métodos de optimización. Se utiliza el conjunto de datos Iris para clasificar tres especies de flores basadas en cuatro características. Se exploran funciones de activación como sigmoide, ReLU, tanh, Leaky ReLU, ELU, Swish y GELU, y se implementan métodos de optimización incluyendo Adam, Momentum, Nesterov, Adagrad y RMSProp. Los resultados demuestran la efectividad de cada función de activación y método de optimización en términos de precisión y pérdida durante el entrenamiento.

## 1 Introducción

Las redes neuronales han transformado el campo del aprendizaje automático, permitiendo resolver problemas complejos en diversas áreas, como visión por computadora y procesamiento del lenguaje natural. Este proyecto tiene como objetivo construir un motor de redes neuronales capaz de clasificar datos utilizando múltiples funciones de activación y métodos de optimización. El conjunto de datos Iris se utiliza como caso de estudio, proporcionando una base sólida para evaluar el rendimiento de diferentes configuraciones.

## 2 Métodos y Conjuntos de Datos

### 2.1 Conjunto de Datos

El conjunto de datos Iris es un conjunto de datos ampliamente utilizado en el campo de la clasificación y el aprendizaje automático. Contiene 150 muestras

de flores, cada una con cuatro características que describen sus dimensiones, y una etiqueta que indica la especie a la que pertenece. Estas características son:

- Longitud del sépalo (en cm)
- Ancho del sépalo (en cm)
- Longitud del pétalo (en cm)
- Ancho del pétalo (en cm)

Cada muestra tiene asociada una etiqueta que corresponde a una de las tres especies de flores Iris:

- **Setosa**
- **Versicolor**
- **Virginica**

El conjunto de datos contiene 150 muestras distribuidas en 3 especies, con 50 muestras por especie. La distribución de las especies en el conjunto es la siguiente:

- 50 muestras de Iris Setosa
- 50 muestras de Iris Versicolor
- 50 muestras de Iris Virginica

## 2.2 Preprocesamiento

Se normalizan las características para mejorar la convergencia del modelo. La normalización se realiza restando la media y dividiendo por la desviación estándar. Las etiquetas se convierten en vectores one-hot para ser compatibles con la función de pérdida.

## 2.3 Funciones de Activación

Se implementan las siguientes funciones de activación:

- **Sigmoid**: Utilizada en la capa de salida para problemas de clasificación binaria, aunque también es compatible con otros problemas. La fórmula de la función Sigmoid es:

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

Su derivada es:

$$\sigma'(z) = \sigma(z) \cdot (1 - \sigma(z))$$

- **ReLU (Rectified Linear Unit)**: Común en capas ocultas debido a su eficiencia y su capacidad para mitigar problemas de desvanecimiento del gradiente. La fórmula de la función ReLU es:

$$\text{ReLU}(z) = \max(0, z)$$

Su derivada es:

$$\text{ReLU}'(z) = \begin{cases} 1 & \text{si } z > 0 \\ 0 & \text{si } z \leq 0 \end{cases}$$

- **Tanh**: Con rango de salida entre -1 y 1, útil para centrar los datos y evitar el desplazamiento de los valores de activación. La fórmula de la función Tanh es:

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

Su derivada es:

$$\tanh'(z) = 1 - \tanh^2(z)$$

- **Leaky ReLU, ELU, Swish y GELU**: Variantes que abordan problemas de desvanecimiento del gradiente en redes profundas. A continuación se presentan sus fórmulas y derivadas:

  - **Leaky ReLU**: La fórmula de la función Leaky ReLU es:

$$\text{Leaky ReLU}(z) = \begin{cases} z & \text{si } z > 0 \\ \alpha z & \text{si } z \leq 0 \end{cases}$$

Su derivada es:

$$\text{Leaky ReLU}'(z) = \begin{cases} 1 & \text{si } z > 0 \\ \alpha & \text{si } z \leq 0 \end{cases}$$

– **ELU (Exponential Linear Unit)**: La fórmula de la función ELU es:

$$\text{ELU}(z) = \begin{cases} z & \text{si } z > 0 \\ \alpha(\exp(z) - 1) & \text{si } z \leq 0 \end{cases}$$

Su derivada es:

$$\text{ELU}'(z) = \begin{cases} 1 & \text{si } z > 0 \\ \alpha \exp(z) & \text{si } z \leq 0 \end{cases}$$

– **Swish**: La fórmula de la función Swish es:

$$\text{Swish}(z) = z \cdot \sigma(z)$$

Su derivada es:

$$\text{Swish}'(z) = \sigma(z) + z \cdot \sigma(z) \cdot (1 - \sigma(z))$$

– **GELU (Gaussian Error Linear Unit)**: La fórmula de la función GELU es:

$$\text{GELU}(z) = 0.5 \cdot z \cdot \left(1 + \text{erf}\left(\frac{z}{\sqrt{2}}\right)\right)$$

Su derivada es:

$$\text{GELU}'(z) = 0.5 \left(1 + \text{erf}\left(\frac{z}{\sqrt{2}}\right)\right) + \frac{z}{\sqrt{2\pi}} \exp\left(-0.5z^2\right)$$

## 2.4 Métodos de Optimización

Se implementan varios métodos de optimización para actualizar los pesos y sesgos de la red, mejorando la convergencia:

- **Adam**: Este algoritmo combina las ventajas de Momentum y RMSProp ajustando la tasa de aprendizaje de manera adaptativa. Su fórmula de actualización es:

$$v_t = \beta_1 v_{t-1} + (1 - \beta_1) \nabla L(w)$$

$$s_t = \beta_2 s_{t-1} + (1 - \beta_2)(\nabla L(w))^2$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_1^t}, \quad \hat{s}_t = \frac{s_t}{1 - \beta_2^t}$$

$$w = w - \eta \frac{\hat{v}_t}{\sqrt{\hat{s}_t} + \epsilon}$$

donde $v_t$ y $s_t$ son las estimaciones del primer y segundo momento de los gradientes, $\beta_1$ y $\beta_2$ son los coeficientes de decaimiento, $\eta$ es la tasa de aprendizaje, y $\epsilon$ es un término pequeño para evitar divisiones por cero.

- **RMSProp**: Este algoritmo es similar a Adagrad pero con una media exponencialmente ponderada de los gradientes en lugar de la acumulación acumulativa. La fórmula para la actualización en RMSProp es:

$$cache_t = \rho cache_{t-1} + (1 - \rho)(\nabla L(w))^2$$

$$w = w - \eta \frac{\nabla L(w)}{\sqrt{cache_t} + \epsilon}$$

donde $\rho$ es el factor de decaimiento del cache de gradientes, $cache_t$ es la acumulación exponencialmente ponderada, $\eta$ es la tasa de aprendizaje y $\epsilon$ evita divisiones por cero.

- **Gradient Descent**: El descenso de gradiente es el método básico para optimizar una función, ajustando los pesos en la dirección opuesta del gradiente de la función de pérdida. Su fórmula general es:

$$w = w - \eta \nabla L(w)$$

donde $w$ son los pesos actuales, $\eta$ es la tasa de aprendizaje y $\nabla L(w)$ representa el gradiente de la función de pérdida con respecto a los pesos.

# 3 Detalles de Implementación

## 3.1 Propagación hacia Adelante

La propagación hacia adelante es el proceso de calcular las salidas de la red neuronal a partir de una entrada dada. Para un modelo con una capa oculta y una capa de salida:

1. **Cálculo en la capa oculta:**

$$Z_1 = XW_1 + b_1 \tag{1}$$
$$A_1 = f(Z_1) \tag{2}$$

donde $X$ es la entrada, $W_1$ y $b_1$ son los pesos y sesgos de la capa oculta, y $f$ es la función de activación (por ejemplo, ReLU o sigmoide).

2. **Cálculo en la capa de salida:**

$$Z_2 = A_1 W_2 + b_2 \tag{3}$$
$$A_2 = \text{softmax}(Z_2) \tag{4}$$

La función *softmax* convierte los valores $Z_2$ en probabilidades:

$$\text{softmax}(z_i) = \frac{\exp(z_i)}{\sum_j \exp(z_j)} \tag{5}$$

El resultado $A_2$ es la salida predicha para la entrada $X$.

## 3.2 Retropropagación

La retropropagación es el proceso de calcular los gradientes necesarios para actualizar los pesos mediante el algoritmo del gradiente descendente. Los pasos son:

1. **Error en la capa de salida:**

$$dZ_2 = A_2 - y \tag{6}$$

donde $y$ son las etiquetas reales.

2. **Gradientes de la capa de salida:**

$$dW_2 = \frac{1}{m} A_1^\top dZ_2 \tag{7}$$
$$db_2 = \frac{1}{m} \sum dZ_2 \tag{8}$$

donde $m$ es el número de ejemplos en el lote.

3. **Error propagado hacia la capa oculta:**

$$dA_1 = dZ_2 W_2^\top \tag{9}$$
$$dZ_1 = dA_1 \cdot f'(Z_1) \tag{10}$$

donde $f'(Z_1)$ es la derivada de la función de activación.

4. **Gradientes de la capa oculta:**

$$dW_1 = \frac{1}{m} X^\top dZ_1 \tag{11}$$
$$db_1 = \frac{1}{m} \sum dZ_1 \tag{12}$$

## 3.3 Entrenamiento del Modelo

El modelo se entrena en múltiples épocas siguiendo estos pasos:

1. **Inicialización:** Se convierten las etiquetas a formato *one-hot* y se inicializan los pesos aleatoriamente.
2. **Propagación hacia adelante:** Se calcula la pérdida mediante la función de entropía cruzada:

$$L = -\frac{1}{m} \sum (y \log(A_2) + (1 - y) \log(1 - A_2)) \tag{13}$$

3. **Retropropagación:** Se calculan los gradientes y se actualizan los pesos utilizando un optimizador, como:
   - Gradiente Descendente (GD)
   - Adam
   - RMSProp
4. **Evaluación:** Se calcula la precisión y la pérdida en el conjunto de validación.
5. **Parada temprana:** Si la pérdida de validación no mejora en un número de épocas consecutivas, el entrenamiento se detiene.

## 3.4 Evaluación del Modelo

Una vez entrenado, el modelo se evalúa en el conjunto de prueba (*test set*) mediante los siguientes pasos:

1. **Predicción:** Se calcula la salida $A_2$ y se selecciona la clase con la mayor probabilidad:

$$\text{Predicción} = \arg\max(A_2) \tag{14}$$

2. **Precisión:** Se calcula como:

$$\text{Precisión} = \frac{\text{número de predicciones correctas}}{\text{número total de ejemplos}} \tag{15}$$

# 4 Experimentos y Resultados

## 4.1 Entrenamiento

Se entrenaron redes neuronales con diferentes configuraciones de funciones de activación y métodos de optimización. A continuación, se presentan los resultados de precisión obtenidos en el conjunto de prueba.

## 4.2 Resultados

Se entrenaron redes neuronales utilizando distintas configuraciones de funciones de activación y métodos de optimización. Como ejemplo, se incluyen las gráficas correspondientes a aquellas configuraciones de métodos de optimización, función de activación, tamaño de capa oculta y learning rate que han obtenido mejores resultado. Sin embargo, en el código están disponibles todas las gráficas para todas las combinaciones utilizadas.

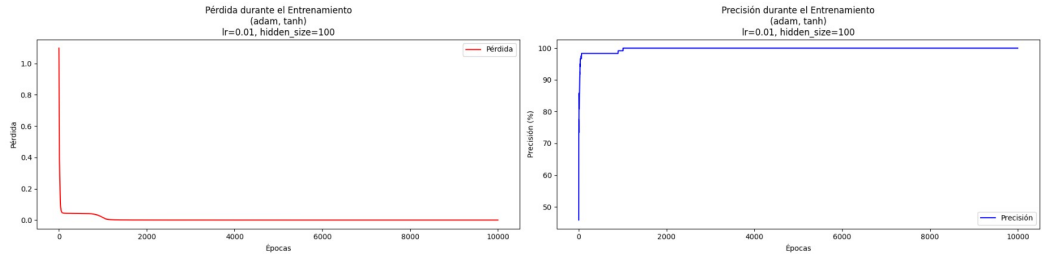


Figure 1: Gráfica de precisión y pérdida durante el entrenamiento con Adam y tanh como función de activación. lr=0.01, size=100

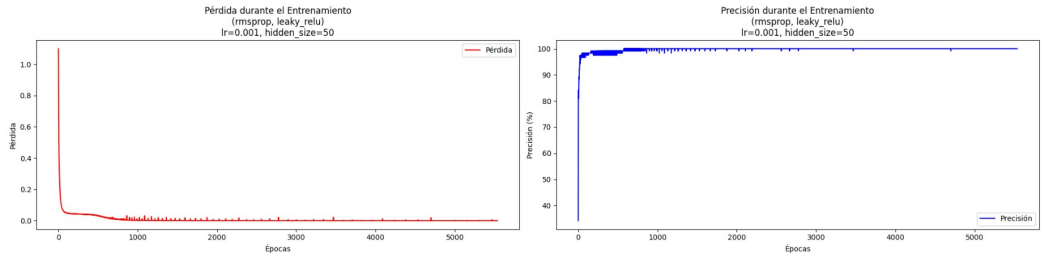


Figure 2: Gráfica de precisión y pérdida durante el entrenamiento con RMSprop y leaky relu como función de activación. lr=0.001, size=50

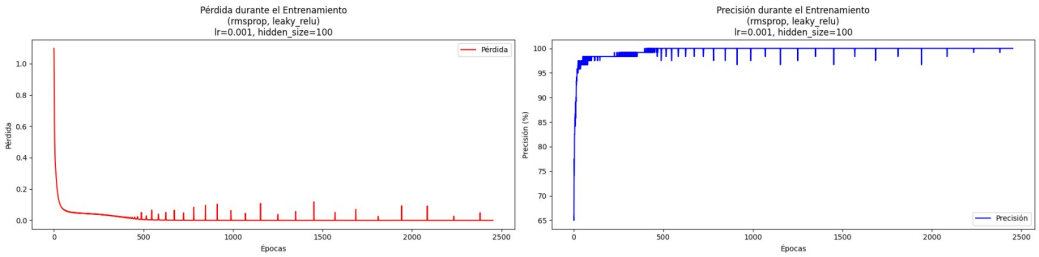


Figure 3: Gráfica de precisión y pérdida durante el entrenamiento con RMSprop y leaky relu como función de activación. lr=0.01, size=100

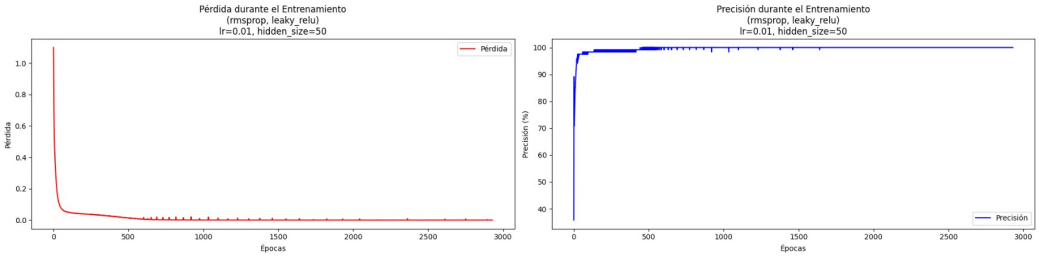


Figure 4: Gráfica de precisión y pérdida durante el entrenamiento con RMSprop y leaky relu como función de activación. lr=0.01, size=50

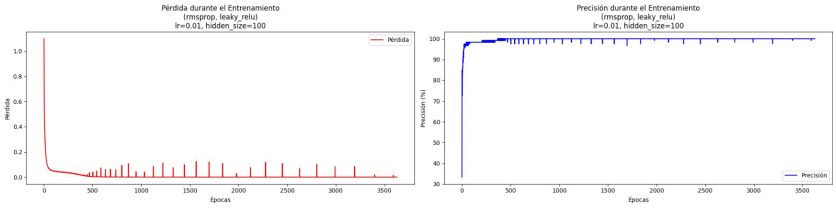


Figure 5: Gráfica de precisión y pérdida durante el entrenamiento con RMSprop y leaky relu como función de activación. lr=0.01, size=100

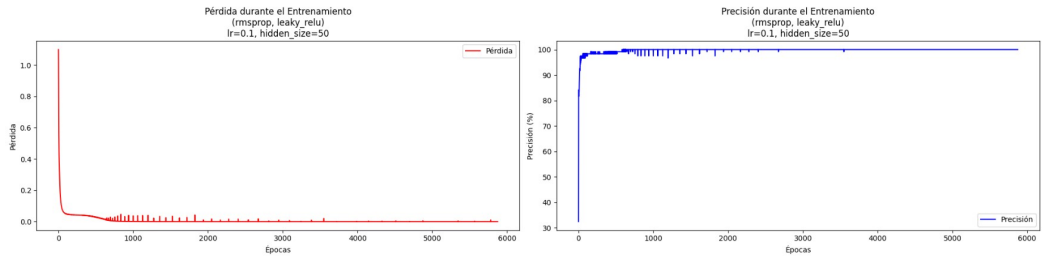


Figure 6: Gráfica de precisión y pérdida durante el entrenamiento con RMSprop y leaky relu como función de activación. lr=0.1, size=50

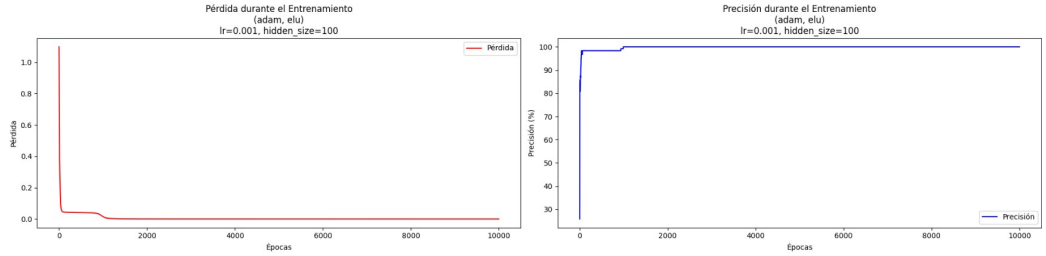


Figure 7: Gráfica de precisión y pérdida durante el entrenamiento con Adam y elu como función de activación. lr=0.01, size=100

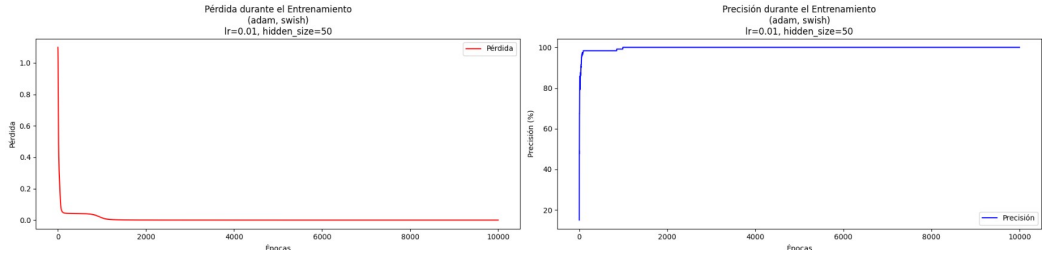


Figure 8: Gráfica de precisión y pérdida durante el entrenamiento con Adam y Swish como función de activación. lr=0.01, size=50

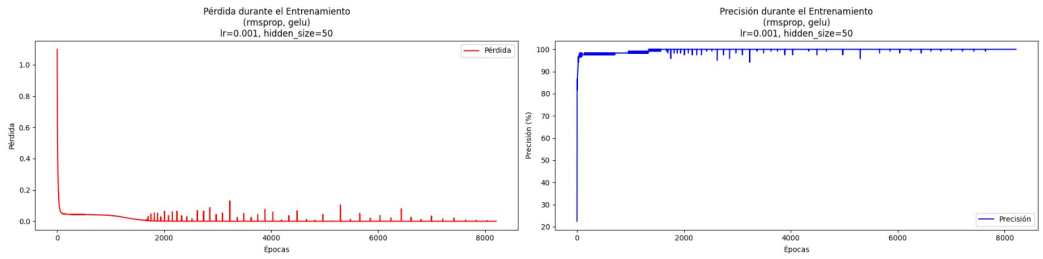


Figure 9: Gráfica de precisión y pérdida durante el entrenamiento con RMSprop y gelu como función de activación. lr=0.001, size=50

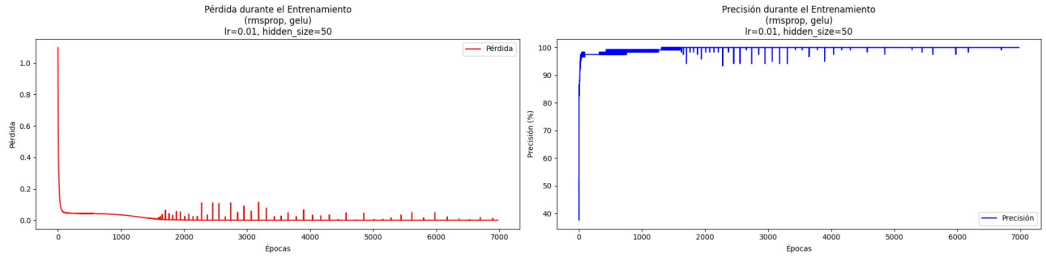


Figure 10: Gráfica de precisión y pérdida durante el entrenamiento con RMSprop y gelu como función de activación. lr=0.01, size=50

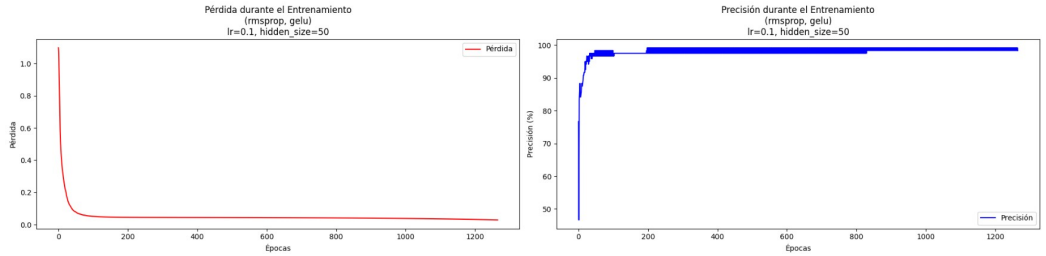


Figure 11: Gráfica de precisión y pérdida durante el entrenamiento con RMSprop y gelu como función de activación. lr=0.1, size=50

# 5 Conclusiones

Entre las combinaciones de parámetros utilizadas, los resultados más destacados, con una pérdida de validación excepcionalmente baja, se obtuvieron

principalmente con configuraciones que utilizaron los optimizadores RMSprop y Adam. Esto sugiere que ambos optimizadores, que comparten en parte fundamentos similares, lograron adaptarse eficazmente a las características de los datos y las redes entrenadas. En cuanto a las funciones de activación, aquellas que introducen no linealidades avanzadas, como GELU y Leaky ReLU, se destacaron. Estas funciones, combinadas con RMSprop, permitieron que las redes capturaran patrones complejos y entrenaran de manera eficiente. Por último, las tasas de aprendizaje más bajas y moderadas (como 0.001 y 0.01), contribuyeron al éxito de estas configuraciones. Este balance en los parámetros posibilitó un entrenamiento preciso, maximizando el rendimiento y minimizando la pérdida y el estancamiento.

# 6 Trabajo Futuro

Para futuras investigaciones, se sugiere investigar arquitecturas de redes neuronales avanzadas, como redes neuronales convolucionales (CNN) o recurrentes (RNN), que puedan abordar problemas en diferentes dominios, como procesamiento de imágenes y texto. También se podría explorar el uso de técnicas de regularización para evitar el sobreajuste en modelos más complejos y mejorar la generalización del modelo en datos desconocidos.

# 7 Repositorio de GitHub

Todo el código fuente, los conjuntos de datos utilizados y la documentación detallada de este proyecto están disponibles en el repositorio de GitHub. Este repositorio proporciona acceso completo a los archivos necesarios para replicar el experimento, junto con instrucciones para su uso.

`https://github.com/LuisPereraPerez/OH_preoyecto`

En el repositorio se incluyen:

- **Código fuente**: Implementación del motor de redes neuronales con diferentes funciones de activación y métodos de optimización.
- **Dataset**: El conjunto de datos Iris utilizado para la clasificación de especies de flores.
- **Documentación**: Archivos y notas explicativas adicionales, que proporcionan una guía para el uso del código y los experimentos realizados.