



Instituto Politécnico Nacional

Escuela Superior de Computo

Ingeniería en sistemas computacionales

Pattern Recognition

**Reconocimiento de objetos con redes
neuronales: Perceptrón Multicapa**



Profesor: Dra. María Elena Cruz Meza.
Alumno: Cruz Villalba Edwin Bernardo.
Fecha de entrega: 31 de mayo del 2025



Índice

Índice.....	2
Introducción.....	3
Objetivo.....	3
Marco teórico.....	4
El perceptrón simple (Una neurona de una sola entrada)	4
Neurona de múltiples entradas.....	6
Arquitecturas de red	7
Una capa de neuronas.....	7
Red Neuronal Multicapa	7
Python	10
Desarrollo	10
Conjunto de datos	10
Arquitectura	12
Entrenamiento.....	13
Resultados	15
Pruebas	15
Interfaz Gráfica	17
Conclusiones	18
Referencias	19

Introducción

Unos de los enigmas que ha inquietado al hombre desde el origen de los tiempos es entender su propia naturaleza, encontrar las características que nos hace humanos, encontrar la diferencia entre el hombre y los animales, y que nos hace únicos. Ese enigma viene asociado con el de la inteligencia dentro de nuestra naturaleza humana esta el ser inteligentes y esta es una característica que nos discrimina absolutamente a nuestra especie.

A medida que la ciencia y la tecnología han ido avanzando el objetivo se ha ido perfilando, uno de esos retos mas importantes es la construcción de sistemas inteligentes, el cual puede ser cualquier dispositivo físico o lógico capaz de realizar una tarea requerida, ese es el objetivo de la disciplina científica conocida como Inteligencia Artificial.

La neurociencia demuestra que el cerebro esta especialmente dotado para el reconocimiento de patrones y básicamente es así como asociamos las imágenes que recibimos visualmente con las personas u objetos que conocemos. En la Inteligencia Artificial se llevan años aplicando con éxito técnicas basadas en el funcionamiento del cerebro, concretamente en el funcionamiento de las neuronas.

Las redes neuronales artificiales no se acercan a la complejidad del cerebro. Sin embargo, existen dos similitudes clave entre las redes neuronales biológicas y artificiales. En primer lugar, los componentes básicos de ambas redes son dispositivos computacionales simples (aunque las neuronas artificiales son mucho más simples que las neuronas biológicas) que están altamente interconectados.

El uso de redes neuronales para clasificación de imágenes es común en aplicaciones de visión por computadora, y este proyecto sirve como un caso práctico para demostrar conceptos clave de aprendizaje profundo, como el preprocesamiento de datos, el diseño de redes neuronales y la evaluación de modelos.

Objetivo

1. Mostrar la habilidad para identificar los parámetros y el manejo e implementación de los algoritmos de reconocimiento de patrones, basados en el enfoque neuronal.
2. Desarrollar un modelo de aprendizaje automático basado en un perceptrón multicapa (MLP) para clasificar imágenes de dos tipos de frutas: fresas y plátanos, siendo abordado como un problema de clasificación binaria, donde el modelo aprende a distinguir entre dos clases a partir de imágenes preprocesadas.
3. Crear un dataset con un conjunto de imágenes ya etiquetadas y preprocesadas para el entrenamiento de la red neuronal.

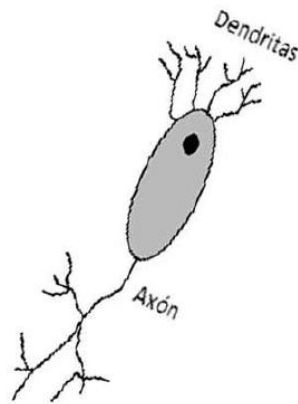
Marco teórico

Las **redes neuronales artificiales** o RNAs son un intento de emular la forma de trabajar del cerebro humano, y aunque estamos lejos de alcanzar su misma capacidad, son un instrumento de gran potencia para gran cantidad de aplicaciones y un campo de investigación muy prometedor.

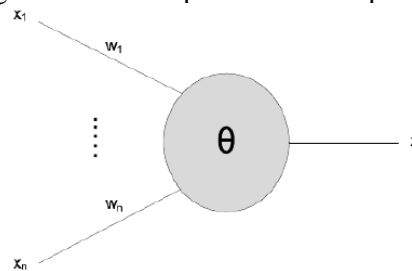
Una **neurona** está compuesta por el núcleo, que no es más que una célula especializada, rodeada millones de conexiones que la unen a otras neuronas. Estas conexiones se denominan sinapsis. Cada neurona está conectada de media a otras mil, aunque pueden ser muchas más. Se estima que un niño tiene alrededor de 1.000. [1]

El perceptrón simple (Una neurona de una sola entrada)

El **perceptrón** es un modelo simple de neurona. Al igual que una neurona real, a nuestro perceptrón llegarán señales de entrada (como si fueran dendritas de una neurona real) y saldrá una salida (simulando un axón). [1]



Las entradas de nuestro perceptrón podrán tomar los valores uno y cero. Además, a cada una de las entradas (x) se le asigna un valor al que llamaremos peso (w). [1]



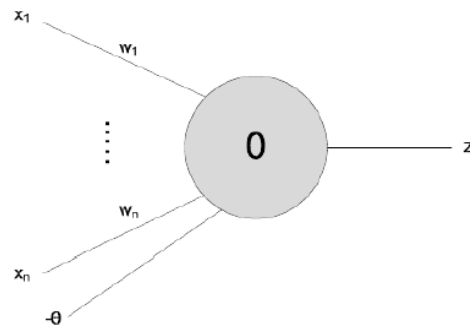
La salida (z) del perceptrón es una función que depende del valor de las entradas, de sus pesos y del umbral de disparo de la neurona (θ) y su valor siempre está entre 0 y 1. [1]

El esquema de un perceptrón con sus entradas x_1, \dots, x_n , los respectivos pesos de cada entrada w_1, \dots, w_n , y el umbral de disparo a partir del cual la neurona se activa. La salida z se calcula según la siguiente función: [1]

$$z = \sum_i x_i \times w_i$$

La suma del producto de cada entrada por su correspondiente peso. Si el

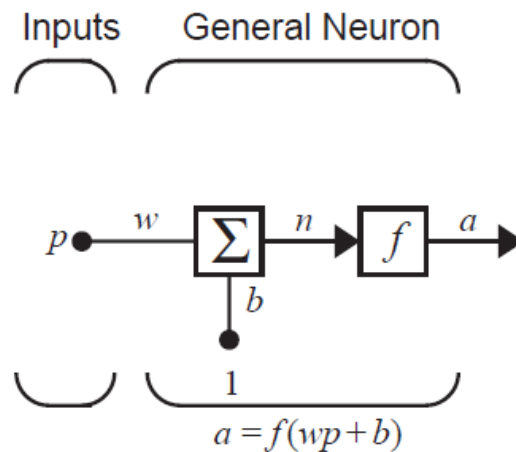
valor de z es mayor que θ , entonces el perceptrón se dispara. [1]



Nuestra tarea será ir ajustando los valores de los pesos w_1, \dots, w_n y de θ a través de un proceso llamado **entrenamiento**. En este tipo de redes neuronales las llamamos de **aprendizaje supervisado**, ya que durante el entrenamiento vamos a ir proveyendo de ejemplos a la red y según la respuesta de la red comparada con la respuesta esperada, vamos a ir ajustando los valores correspondientes. [1]

La entrada escalar p se multiplica por el peso escalar w para formar, uno de los términos que se envía al sumador. La otra entrada, 1, se multiplica por un sesgo b y luego se pasa al sumador. La salida del sumador, a menudo denominada entrada neta, se introduce en una función de transferencia, que produce la salida de la neurona escalar a . (Algunos autores utilizan el término "función de activación" en lugar de función de transferencia y "desplazamiento" en lugar de sesgo). [2]

Si relacionamos este modelo simple con la neurona biológica, el peso corresponde a la fuerza de una sinapsis, el cuerpo celular está representado por la suma y la función de transferencia, y la salida de la neurona representa la señal en el axón. [2]



La salida de la neurona se calcula como: $a = f(wp + b)$

Si $aw = 3$, $p = 2$ y $b = -1.5$ entonces: $a = f(3(2) - 1.5) = f(4.5)$

La salida real depende de la función de transferencia específica elegida. [2]

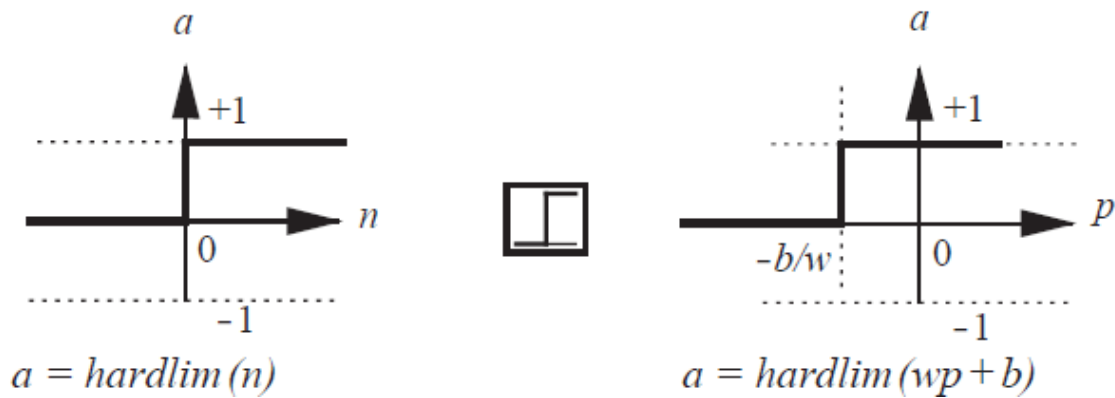
El sesgo es muy similar a un peso, excepto que tiene una entrada constante de 1.

Sin embargo, si no se desea tener un sesgo en una neurona en particular, se puede omitir.

Tenga en cuenta que w y b son parámetros escalares ajustables de la neurona. Normalmente, el diseñador elige la función de transferencia y luego los parámetros se ajustan mediante una regla de

aprendizaje para que la relación entrada/salida de la neurona cumpla un objetivo específico, existen diferentes funciones de transferencia para distintos propósitos. [2]

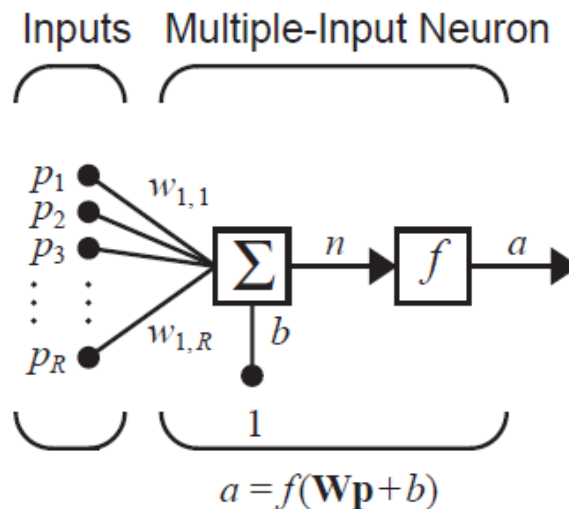
La función de transferencia puede ser lineal o no lineal de n . Se elige una función de transferencia específica para satisfacer alguna especificación del problema que la neurona intenta resolver.



La función de activación *hardlim*, que se muestra a la izquierda de la Figura anterior, establece la salida de la neurona en 0 si el argumento de la función es menor que 0, o en 1 si su argumento es mayor o igual que 0. [2] Este tipo de funciones de activación son las mas usadas para clasificación binaria.

Neurona de múltiples entradas

Normalmente, una neurona tiene más de una entrada. Cada entrada individual p_1, p_2, \dots, p_R se pondera mediante los elementos w_1, w_2, \dots, w_R correspondientes de la matriz de ponderaciones \mathbf{W} .



La neurona tiene un sesgo b ., que se suma con las entradas ponderadas para formar la entrada neta n : $n = w_{1,1}p_1 + w_{1,2}p_2 + \dots + w_{1,R}p_R + b$. [2]

Esta expresión puede escribirse en forma matricial $n = \mathbf{W}\mathbf{p} + b$ donde la matriz \mathbf{W} para el caso de una sola neurona tiene solo una fila. Ahora, la salida de la neurona puede escribirse como [2]:

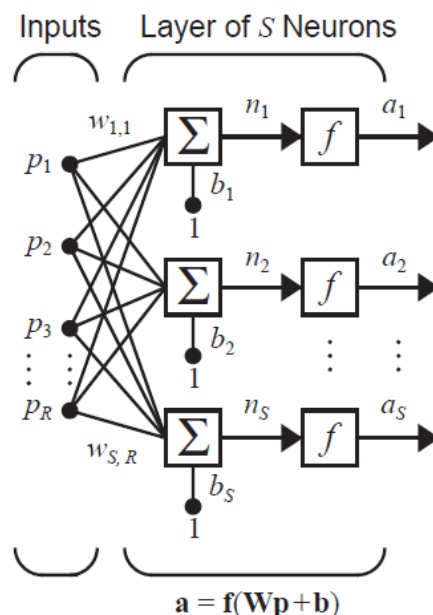
$$a = f(\mathbf{W}\mathbf{p} + b)$$

Arquitecturas de red

Comúnmente, una sola neurona, incluso con muchas entradas, puede no ser suficiente. Podríamos necesitar cinco o diez, operando en paralelo, en lo que llamaremos una "capa".

Una capa de neuronas

En la siguiente imagen se muestra una red de neuronas S de una sola capa. Observe que cada entrada R está conectada a cada neurona y que la matriz de ponderaciones ahora tiene S filas.

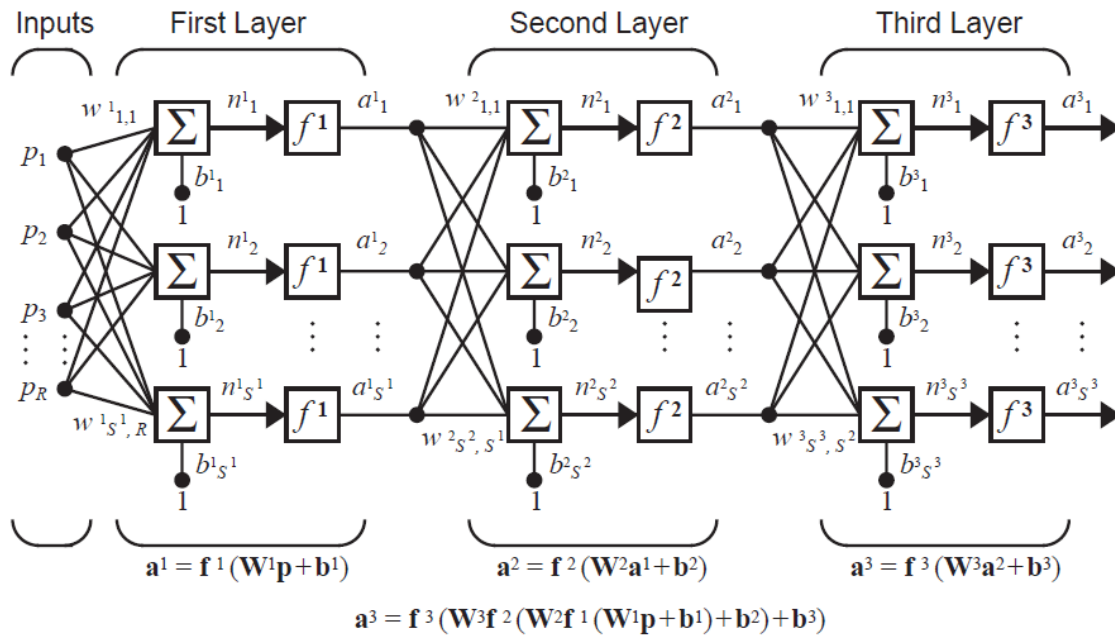


La capa incluye la matriz de pesos, los sumadores, el vector de sesgo \mathbf{b} , las casillas de función de transferencia y el vector de salida \mathbf{a} . Cada elemento del vector de entrada \mathbf{p} está conectado a cada neurona a través de la matriz de pesos \mathbf{W} . Cada neurona tiene un sesgo b , un sumador Σ , una función de transferencia f y una salida a_i . En conjunto, las salidas forman el vector de salida \mathbf{a} . [2]

Es común que el número de entradas de una capa sea diferente del número de neuronas. Podría preguntarse si todas las neuronas de una capa deben tener la misma función de transferencia. La respuesta es no; se puede definir una sola capa (compuesta) de neuronas con diferentes funciones de transferencia combinando dos de las redes mostradas arriba en paralelo. Ambas redes tendrían las mismas entradas y cada una generaría algunas de las salidas. [2]

Red Neuronal Multicapa

Consideremos ahora una red con varias capas. Cada capa tiene su propia matriz de pesos \mathbf{W} , su propio vector de sesgo \mathbf{b} , un vector de entrada neta \mathbf{n} y un vector de salida \mathbf{a} . Necesitamos introducir notación adicional para distinguir entre estas capas. Usaremos superíndices para identificarlas. Específicamente, añadimos el número de la capa como superíndice a los nombres de cada una de estas variables. Por lo tanto, la matriz de pesos de la primera capa se escribe como \mathbf{W}^1 , y la matriz de pesos de la segunda capa se escribe como \mathbf{W}^2 . Esta notación se utiliza en la red de tres capas que se muestra en la siguiente imagen [2]:












Como se muestra, hay entradas: neuronas en la primera capa, neuronas en la segunda capa, etc. Como se indicó, las diferentes capas pueden tener diferentes números de neuronas. Las salidas de las capas uno y dos son las entradas de las capas dos y tres. [2]

Las redes multicapa son más potentes que las redes de una sola capa. Por ejemplo, una red de dos capas con una primera capa sigmoidea y una segunda capa lineal puede entrenarse para aproximarse a la mayoría de las funciones con una precisión arbitraria. Las redes de una sola capa no pueden hacerlo [2].

En primer lugar, recuerde que el número de entradas y salidas de la red se definen mediante especificaciones externas del problema. Por lo tanto, si se utilizan cuatro variables externas como entradas, la red tiene cuatro entradas. De igual manera, si la red tiene siete salidas, la capa de salida debe tener siete neuronas [2].

Finalmente, las características deseadas de la señal de salida también ayudan a seleccionar la función de transferencia para la capa de salida. Si una salida debe ser 0 o -1, se debe utilizar una función de transferencia simétrica de límite rígido. Por lo tanto, la arquitectura de una red de una sola capa está casi completamente determinada por las especificaciones del problema, incluyendo el número específico de entradas y salidas, y la característica particular de la señal de salida [2].

Estas son algunas de las funciones de activación más conocidas:

Name	Input/Output Relation	Icon
Hard Limit	$a = 0 \quad n < 0$ $a = 1 \quad n \geq 0$	
Symmetrical Hard Limit	$a = -1 \quad n < 0$ $a = +1 \quad n \geq 0$	
Linear	$a = n$	
Saturating Linear	$a = 0 \quad n < 0$ $a = n \quad 0 \leq n \leq 1$ $a = 1 \quad n > 1$	
Symmetric Saturating Linear	$a = -1 \quad n < -1$ $a = n \quad -1 \leq n \leq 1$ $a = 1 \quad n > 1$	
Log-Sigmoid	$a = \frac{1}{1 + e^{-n}}$	
Hyperbolic Tangent Sigmoid	$a = \frac{e^n - e^{-n}}{e^n + e^{-n}}$	
Positive Linear	$a = 0 \quad n < 0$ $a = n \quad 0 \leq n$	
Competitive	$a = 1 \quad \text{neuron with max } n$ $a = 0 \quad \text{all other neurons}$	

Python

Cualquier lenguaje puede utilizarse en Inteligencia Artificial, sin embargo, algunos se adaptan mejor que otros, sobre todo aquellos que son capaces de manejar complejas estructuras de datos. También nos ofrece la flexibilidad necesaria y a la vez es un lenguaje con una curva de aprendizaje muy rápida. Frente a otros lenguajes, como Java o C, Python tiene la ventaja de ser un lenguaje interpretado y no necesita una compilación previa, lo que nos facilita el ciclo de programación y pruebas [2].

Python es ampliamente reconocido como el lenguaje de programación líder en el campo de la Inteligencia Artificial (IA) debido a las siguientes razones clave:

- **Amplia disponibilidad de librerías especializadas:** Python cuenta con un ecosistema robusto de librerías enfocadas en IA y aprendizaje automático, como TensorFlow, Keras, PyTorch, Scikit-learn y OpenCV, que facilitan el desarrollo de modelos complejos con pocas líneas de código.
- **Simplicidad y legibilidad:** Su sintaxis clara y concisa permite a los desarrolladores enfocarse en la lógica de los algoritmos en lugar de lidiar con complejidades del lenguaje, lo cual acelera el desarrollo y reduce errores.
- **Gran comunidad y soporte activo:** La comunidad global de desarrolladores en Python es muy activa, lo que garantiza documentación extensa, resolución rápida de problemas y contribuciones continuas que enriquecen las herramientas disponibles.
- **Compatibilidad multiplataforma y flexibilidad:** Python se puede ejecutar en diversos sistemas operativos y se integra fácilmente con otros lenguajes y tecnologías, permitiendo la creación de soluciones escalables e interoperables.
- **Uso en la industria y la investigación:** Las principales empresas tecnológicas y centros de investigación utilizan Python como estándar en proyectos de IA, lo que refuerza su posicionamiento como la mejor opción para desarrollos en este campo.

Desarrollo

El presente proyecto desarrolla un modelo de aprendizaje automático basado en un **perceptrón multicapa** (MLP) para clasificar imágenes de dos tipos de frutas: **fresas** y **plátanos**. Este tipo de tarea es un problema de clasificación binaria, donde el modelo aprende a distinguir entre dos clases a partir de imágenes preprocesadas.

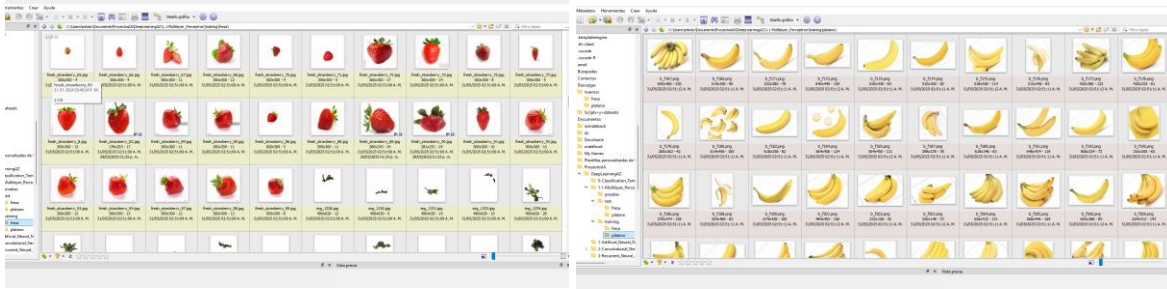
Conjunto de datos

El conjunto de datos consiste en imágenes de fresas y plátanos organizadas en dos directorios: **training** y **test**.

Cada imagen fue estandarizado a 128x128 píxeles y 3 canales de color (RGB). Los datos de entrenamiento y validación se dividen en dos carpetas:

- **Entrenamiento:** Contiene imágenes de ambas clases (fresas y plátanos) para entrenar el modelo.
- **Validación:** Conjunto separado para evaluar el desempeño del modelo.





Para mejorar la generalización del modelo, se aplicaron técnicas de aumento de datos (data augmentation) durante el entrenamiento, como rotaciones, cambios de brillo, zoom y traslaciones, descritas en la sección 2.3.

Se utilizaron los generadores de datos de TensorFlow/Keras (ImageDataGenerator) para cargar y preprocesar las imágenes. Las técnicas aplicadas incluyen:

- **Normalización:** Escalado de los valores de píxeles a $[0, 1]$
- **Estandarización por imagen:** Aplicación de `per_image_standardization` para normalizar los valores de píxeles restando la media y dividiendo por la desviación estándar de cada imagen.
- **Aumento de datos:** Para el conjunto de entrenamiento, se aplicaron transformaciones aleatorias como:
 - Rotación hasta 20 grados.
 - Zoom de hasta 30%.
 - Volteo horizontal.
 - Cambios de brillo entre 70% y 130%.
 - Desplazamientos horizontales y verticales de hasta 20%.
 - Cizallamiento de hasta 20%.

Estas técnicas ayudan a prevenir el **sobreaajuste (overfitting)** al aumentar la variabilidad de las imágenes de entrenamiento.

```
# GENERADORES
train_datagen = ImageDataGenerator(
    rescale=1./255,
    rotation_range=20,
    zoom_range=0.3,
    horizontal_flip=True,
    brightness_range=[0.7, 1.3],
    shear_range=0.2,
    width_shift_range=0.2,
    height_shift_range=0.2,
    preprocessing_function=tf.image.per_image_standardization
)

test_datagen = ImageDataGenerator(
    rescale=1./255,
    preprocessing_function=tf.image.per_image_standardization
)

train_generator = train_datagen.flow_from_directory(
    'training',
    target_size=IMG_SIZE,
    batch_size=BATCH_SIZE,
    class_mode='binary',
    shuffle=True
)

validation_generator = test_datagen.flow_from_directory(
    'test',
    target_size=IMG_SIZE,
    batch_size=BATCH_SIZE,
    class_mode='binary',
    shuffle=False
)
```

Arquitectura

- **Capa de aplanamiento (Flatten):** Convierte las imágenes de 128x128x3 (49,152 píxeles) en un vector de entrada unidimensional.
- **Capa densa (512 neuronas):** Activación ReLU con inicialización de pesos he_normal para mejorar la convergencia.
- **Normalización por lotes (BatchNormalization):** Estabiliza y acelera el entrenamiento al normalizar las activaciones.
- **Dropout (20%):** Regularización para reducir el sobreajuste.
- **Capa densa (256 neuronas):** ReLU, he_normal, seguida de BatchNormalization y Dropout (20%).
- **Capa densa (128 neuronas):** ReLU, he_normal, seguida de BatchNormalization y Dropout (10%).
- **Capa de salida (1 neurona):** Activación sigmoide para clasificación binaria (0: plátano, 1: fresa).

```
# PESOS DE CLASE
fresa_count = len(os.listdir('training/fresa'))
platano_count = len(os.listdir('training/platano'))
total = fresa_count + platano_count
class_weights = {
    0: (1 / platano_count) * (total / 2.0),
    1: (1 / fresa_count) * (total / 2.0)
} if fresa_count != platano_count else None

# MODELO MLP
model = Sequential([
    Flatten(input_shape=(IMG_SIZE[0], IMG_SIZE[1], 3)),
    Dense(512, activation='relu', kernel_initializer='he_normal'),
    BatchNormalization(),
    Dropout(0.2),
    Dense(256, activation='relu', kernel_initializer='he_normal'),
    BatchNormalization(),
    Dropout(0.2),
    Dense(128, activation='relu', kernel_initializer='he_normal'),
    BatchNormalization(),
    Dropout(0.1),
    Dense(1, activation='sigmoid')
])

model.compile(
    optimizer=Adam(learning_rate=0.0005),
    loss='binary_crossentropy',
    metrics=['accuracy', tf.keras.metrics.Precision(), |
            tf.keras.metrics.Recall()]
)
```

- **Optimizador:** Adam con una tasa de aprendizaje inicial de 0.0005.
- **Función de pérdida:** binary_crossentropy, adecuada para problemas de clasificación binaria.
- **Métricas:** Precisión (accuracy), precisión (precision) y sensibilidad (recall).
- **Pesos de clase:** Se calcularon pesos para balancear las clases si el número de imágenes de fresas y plátanos no es igual, utilizando la fórmula $(1 / \text{conteo_clase}) * (\text{total} / 2.0)$.

Entrenamiento

El modelo se entrenó durante un máximo de 30 épocas con las siguientes configuraciones:

- **Tamaño del lote (batch size):** 32.
- **Callbacks:**
 - **EarlyStopping:** Detiene el entrenamiento si la pérdida de validación (val_loss) no mejora después de 10 épocas, restaurando los mejores pesos.
 - **ReduceLROnPlateau:** Reduce la tasa de aprendizaje por un factor de 0.2 si la pérdida de validación no mejora tras 5 épocas, con un mínimo de 1e-6.
- **Pesos de clase:** Aplicados para manejar desbalance de clases, si lo hubiera.

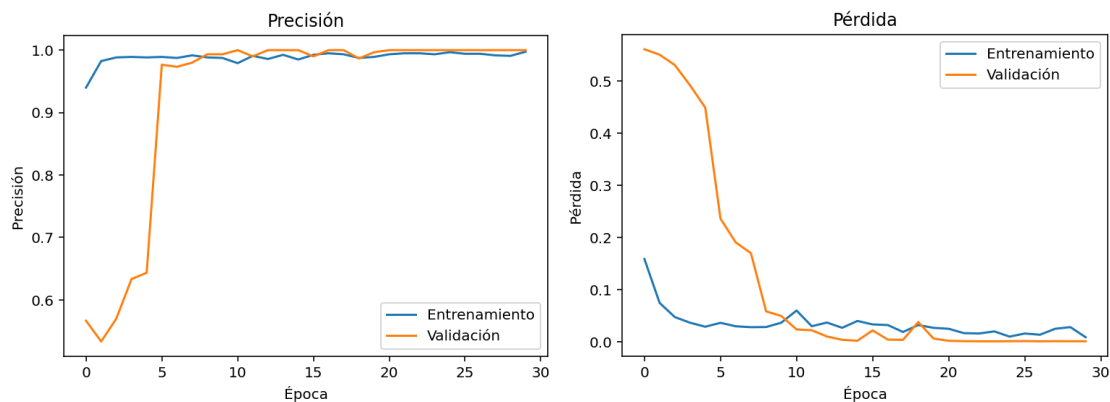
```
# ENTRENAMIENTO
history = model.fit(
    train_generator,
    validation_data=validation_generator,
    epochs=EPOCHS,
    callbacks=[early_stop, reduce_lr],
    class_weight=class_weights
)
```

El modelo fue evaluado utilizando el conjunto de validación, midiendo precisión, pérdida, precisión (precision) y sensibilidad (recall).

```
plt.plot(history.history['accuracy'], label='Entrenamiento')
plt.plot(history.history['val_accuracy'], label='Validación')
plt.title('Precisión')
plt.xlabel('Época')
plt.ylabel('Precisión')
plt.legend()
plt.show()

plt.plot(history.history['loss'], label='Entrenamiento')
plt.plot(history.history['val_loss'], label='Validación')
plt.title('Pérdida')
plt.xlabel('Época')
plt.ylabel('Pérdida')
plt.legend()
plt.show()
```

Los resultados se visualizaron mediante gráficas de precisión y pérdida en función de las épocas.



- **Gráfica de precisión:** La precisión de entrenamiento aumenta rápidamente y luego se estabiliza. La precisión de validación sigue un patrón similar, pero puede ser ligeramente menor, lo que indica una generalización aceptable. Si las curvas divergen significativamente, podría haber sobreajuste.
- **Gráfica de pérdida:** La pérdida de entrenamiento disminuye constantemente, mientras que la pérdida de validación puede mostrar fluctuaciones iniciales debido al aumento de datos. La convergencia de ambas curvas sugiere un modelo bien entrenado.

Resultados

Pruebas

El modelo entrenado, guardado como `fruit_classifier_model_improved.keras`, se carga utilizando la función `load_model` de TensorFlow/Keras. Este modelo es un perceptrón multicapa diseñado para clasificar imágenes de 128x128 píxeles en dos clases: **fresas (1)** y **plátanos (0)**.

Preprocesamiento de Imágenes Externas

Las imágenes externas, almacenadas en el directorio pruebas, se procesan de la siguiente manera:

- **Carga de imágenes:** Se utilizan archivos con extensiones `.png`, `.jpg` o `.jpeg`. La función `image.load_img` de Keras carga cada imagen y la redimensiona a 128x128 píxeles.
- **Conversión a arreglo:** La imagen se convierte en un arreglo numérico (`image.img_to_array`) con dimensiones 128x128x3 (RGB).
- **Estandarización:** Se aplica `tf.image.per_image_standardization` para normalizar los valores de píxeles restando la media y dividiendo por la desviación estándar.
- **Normalización:** Los valores de píxeles se escalan a `[0, 1]` dividiendo por 255.
- **Expansión de dimensiones:** Se agrega una dimensión adicional (`np.expand_dims`) para que el arreglo tenga la forma (1, 128, 128, 3), compatible con el modelo.

```
import os
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.models import load_model
from tensorflow.keras.preprocessing import image
import tensorflow as tf

# CARGAR MODELO
model = load_model("fruit_classifier_model_improved.keras")
IMG_SIZE = (128, 128)

# CLASES
class_labels = ['fresa', 'plátano'] # 0: plátano, 1: fresa

# CARGAR Y MOSTRAR PREDICCIONES
test_dir = 'pruebas'
image_files = [f for f in os.listdir(test_dir) if f.lower().endswith(
    ('.png', '.jpg', '.jpeg'))]

for fname in image_files:
    img_path = os.path.join(test_dir, fname)
    img = image.load_img(img_path, target_size=IMG_SIZE)
    img_array = image.img_to_array(img)
    img_array = tf.image.per_image_standardization(img_array)
    img_array = img_array / 255.0
    img_array = np.expand_dims(img_array, axis=0)

    prediction = model.predict(img_array)[0][0]
    label = class_labels[1] if prediction > 0.5 else class_labels[0]

    # --- MOSTRAR IMAGEN ---
    plt.figure()
    plt.imshow(img)
    plt.axis('off')
    plt.title(f"Predicción: {label} ({prediction:.2f})")
    plt.show()
```


Para cada imagen, se genera una visualización que incluye la imagen original y la etiqueta según el modelo.

Predicción: plátano (1.00)



Predicción: fresa (0.21)



Predicción: fresa (0.00)



Predicción: fresa (0.00)



Predicción: fresa (0.00)



Predicción: fresa (0.01)



Predicción: plátano (1.00)



Predicción: plátano (1.00)



Predicción: fresa (0.00)



Predicción: fresa (0.01)



Predicción: fresa (0.00)



Predicción: plátano (0.99)



Predicción: plátano (1.00)



Predicción: plátano (0.98)



Predicción: plátano (0.68)



Predicción: fresa (0.00)



Predicción: fresa (0.00)



Predicción: plátano (1.00)

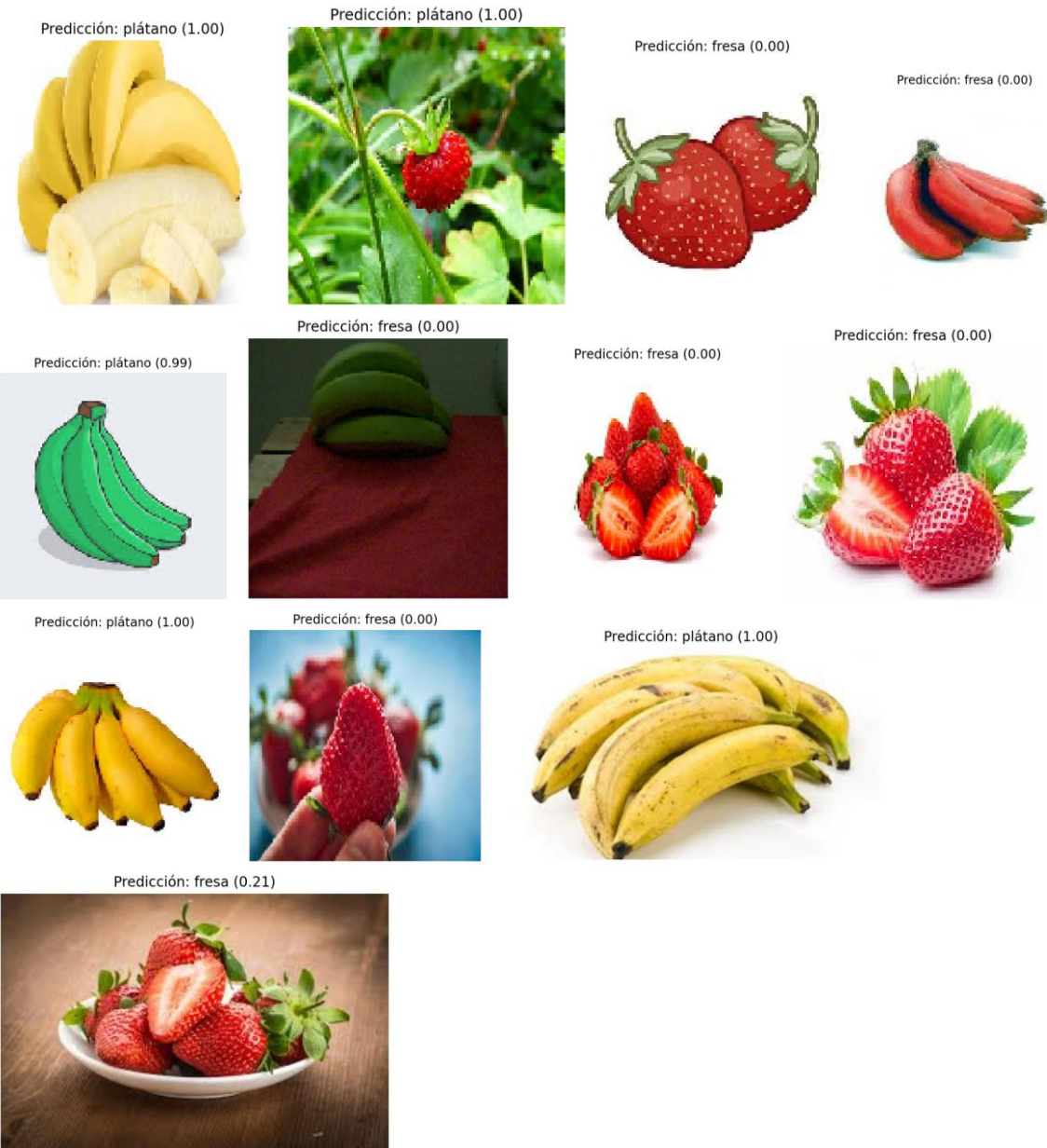


Predicción: plátano (1.00)



Predicción: plátano (1.00)

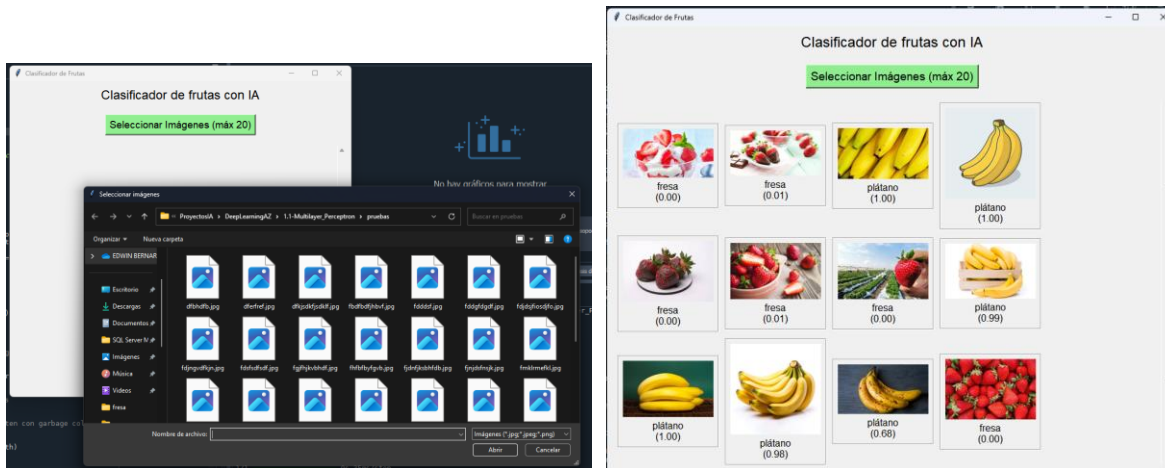




Interfaz Gráfica

La interfaz se implementó utilizando la biblioteca tkinter de Python, que permite crear aplicaciones gráficas de manera sencilla. Las características principales son:

- **Selección de imágenes:** Un botón permite al usuario seleccionar hasta 20 imágenes en formatos .jpg, .jpeg o .png mediante un cuadro de filedialog.
- **Visualización:** Las imágenes se muestran en una cuadrícula de 4 columnas dentro de un marco desplazable, junto con la predicción de la clase y la probabilidad asociada.
- **Preprocesamiento:** Cada imagen se carga, redimensiona a 128x128 píxeles, se estandariza (tf.image.per_image_standardization), se normaliza (división por 255) y se pasa al modelo para predecir la clase.



Conclusiones

La experiencia en la construcción de esta (MLP) para clasificar fresas y plátanos resalta que el conjunto de datos es un factor crítico en el desempeño de cualquier modelo de aprendizaje automático. En este proyecto, se enfrentaron desafíos relacionados con la calidad y cantidad de los datos. Inicialmente, se trabajó con un conjunto de más de 3000 imágenes recopiladas de diversas fuentes. Sin embargo, para mejorar la calidad del dataset, se realizó un proceso de limpieza exhaustivo, seleccionando imágenes con fondo blanco o neutro y eliminando aquellas que no cumplían con criterios claros, como plátanos verdes, plátanos macho, o plátanos en etapas de maduración avanzada. Este proceso redujo el conjunto a aproximadamente 700 imágenes, lo que permitió un aprendizaje más consistente, aunque limitó la diversidad de los datos. Esta experiencia subraya que un dataset bien curado, aunque más pequeño, puede ser más efectivo que uno grande pero ruidoso. La calidad del dataset, incluyendo la representatividad de las clases y la consistencia visual, es fundamental para garantizar la generalización del modelo.

El modelo MLP desarrollado, a pesar de su simplicidad, logró clasificar imágenes con un número razonable de parámetros, gracias a técnicas de aumento de datos, regularización y balanceo de clases mediante pesos. Las transformaciones aplicadas a las imágenes durante el entrenamiento, como rotaciones, cambios de brillo, zoom, volteo horizontal, y estandarización, mejoraron la robustez del modelo frente a variaciones en las imágenes. Sin embargo, el uso de un MLP presenta limitaciones, ya que no aprovecha la estructura espacial de las imágenes, a diferencia de las **redes neuronales convolucionales** (CNN). Esto puede reducir su capacidad para detectar características complejas en tareas de visión por computadora más avanzadas. Además, el tamaño de las imágenes (128x128 píxeles) resultó adecuado para reducir el tiempo de entrenamiento y mantener un desempeño aceptable, aunque podría haber perdido detalles finos presentes en imágenes de mayor resolución.

Un dataset reducido (700 imágenes) puede limitar la capacidad del modelo para generalizar a escenarios del mundo real, especialmente en casos con variaciones no representadas, como frutas con fondos complejos o en diferentes etapas de maduración. Además, el MLP, aunque efectivo para esta tarea binaria, no es óptimo para problemas de visión por computadora más complejos, donde las CNN o modelos preentrenados serían más adecuados.

El desarrollo de esta red neuronal, como una de las primeras experiencias en aprendizaje profundo, destacó varios aspectos clave:

- **Funciones de activación y error:** La elección de la activación sigmoide en la capa de salida y la función de pérdida `binary_crossentropy` fue adecuada para la clasificación

binaria, facilitando la convergencia del modelo.

- **Normalización:** La estandarización por imagen y la normalización a $[0, 1]$ mejoraron la estabilidad del entrenamiento, aunque su impacto debe evaluarse cuidadosamente para evitar pérdida de información.
- **Dataset como prioridad:** Contrario a la creencia inicial de que ajustar la arquitectura (por ejemplo, agregar capas ocultas o cambiar el tamaño de las imágenes) resolvería problemas de desempeño, el dataset resultó ser el factor más crítico. La limpieza y selección cuidadosa de imágenes fueron esenciales para obtener resultados consistentes.

Para futuros desarrollos, se proponen las siguientes mejoras:

- **Adoptar una CNN:** Implementar una red neuronal convolucional para aprovechar la estructura espacial de las imágenes, lo que mejoraría el reconocimiento de patrones visuales y el desempeño general.
- **Ampliar el dataset:** Incluir más imágenes que representen variaciones realistas, como frutas con fondos complejos, diferentes etapas de maduración (por ejemplo, plátanos verdes o maduros), o condiciones de iluminación diversas. Esto aumentaría la robustez del modelo.
- **Ajuste de hiperparámetros:** Experimentar con diferentes tasas de aprendizaje, tamaños de lote, o configuraciones de aumento de datos para optimizar el entrenamiento.

Referencias

- [1] A. G. Serrano, INTELIGENCIA ARTIFICIAL Fundamentos, práctica y aplicaciones, Ciudad de Mexico: Alfaomega Grupo Editor, S.A. de C.V., 2016.
- [2] H. B. D. M. H. B. O. D. J. Martin T. Hagan, Neural Network Design, 2nd Edition, eBook.