Aplicación de Inteligencia Artificial

Hellen Aguilar Noguera

José Leonardo Araya Parajeles

Universidad CENFOTEC

Clasificación Multiclase usando Red Neuronal Profunda con Keras (Dataset Iris)

A continuación se presenta la red neuronal profunda, la cual utiliza el dataset Iris para realizar una clasificación multiclase de 3 especies: Iris-setosa, Iris-versicolor, Iris-virginica.

Es importante indicar que el modelo se construye con Keras, tiene más de tres capas (lo cual hace el cumpliendo con el requisito de profundidad), y se entrena para aprender patrones a partir de las características de entrada (longitud y ancho de sépalos y pétalos).

El documento incluye varias secciones:

- Preprocesamiento.
- Construcción del modelo.
- Entrenamiento con monitoreo.
- Evaluación de métricas
- Conclusiones finales.

Instalación y carga de librerías

```
# Importar las librerías necesarias
import numpy as np
import pandas as pd
import tensorflow as tf
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder, OneHotEncoder, StandardScaler # SE añade el StandardScaler para alcanzar un 95% en el Accuracy
from sklearn.metrics import classification_report, accuracy_score
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.callbacks import EarlyStopping # Para detener el entrenamiento cuando la pérdida de validación deje de mejorar, evitando
```

Cargar y preprocesar los datos:

- Datos de entrada (X): 4 características numéricas (sepal_length, sepal_width, petal_length, petal_width).
- Etiqueta (y): Clase categórica (Iris-setosa, Iris-versicolor, Iris-virginica).
- Codificada primero con LabelEncoder (0, 1, 2) y luego con one-hot encoding para la salida multiclase ([1,0,0], [0,1,0], [0,0,1]).
- División: 80% entrenamiento (X_train, y_train), 20% prueba (X_test, y_test) con random_state=42 para reproducibilidad.

```
# Cargar dataset Iris
url = "https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data"

# Se signan nombres descriptivos a las columnas: sepal_length: Longitud del sépalo (en cm). sepal_width: Ancho del sépalo (en cm).
# petal_length: Longitud del pétalo (en cm). # petal_width: Ancho del pétalo (en cm). #class: Especie o clase de Iris.
df = pd.read_csv(url, header=None, names=["sepal_length", "sepal_width", "petal_length", "petal_width", "class"])

# Características (X) y etiquetas (y)
# X: Obtiene todas las filas - y: Obtiene todas las filas
# Características (X): Son los datos o variables que usará el modelo para aprender. En este caso, son las medidas de longitud y ancho del sép
# Etiquetas (y): Es lo que queremos que el modelo prediga. En este caso, es la clase o especie de Iris (Setosa, Versicolor o Virginica).

X = df.iloc[:, :-1].values
y = df.iloc[:, -1].values

# Codificar etiquetas categóricas a valores numéricos
# Es importante recordar que LabelEncoder() es una herramienta de la librería scikit-learn que convierte etiquetas categóricas (por ejemplo,
encoder = LabelEncoder()
y_encoded = encoder.fit_transform(y) # Convierte clases a 0, 1, 2

# Codificación One-hot para la salida multiclase= Convierte etiquetas numéricas a vectores binarios
y_onehot = tf.keras.utils.to_categorical(y_encoded)

# Normalizar las características
scaler = StandardScaler()
X = scaler.fit_transform(X)  # Estandariza las características (media=0, varianza=1) //Esto lo estoy aplicando para alcanzar un 95% en el Acc

# División en entrenamiento (80%) y prueba (20%) = Divide datos en entrenamiento y prueba
X_train, X_test, y_train, y_test = train_test_split(X, y_onehot, test_size=0.2, random_state=42)
```

¿Por qué usamos encoder?

Porque los algoritmos de Machine Learning no entienden etiquetas como palabras, solo números. Por lo tanto, debemos transformar estas etiquetas textuales en numéricas.

Ejemplo de transformación:

Clase original Transformado Iris-setosa 0 Iris-versicolor 1 Iris-virginica 2

Se puede concluir lo siguiente:

Debido a que Iris es un dataset sencillo y estándar, se espera que un modelo neuronal con buena configuración obtenga métricas altas (generalmente superiores al 85%-95% en precisión global).

El conjunto separado correctamente (80%-20%) asegura que el modelo pueda demostrar si ha generalizado adecuadamente a partir del entrenamiento.

Construcción del modelo de Red Neuronal Artificial

Esto se hace utilizando la libreria de Keras con la finalidad de resolver un problema de clasificación con el dataset Iris.Importante menciaor que esta red tiene más de 3 capas, lo cual es parte del requisito del proyecto.

```
 # Arquitectura:
 Capa de entrada: 4 neuronas (una por característica).
 Capa oculta 1: 16 neuronas, activación ReLU.
 Capa oculta 2: 12 neuronas, activación ReLU.
 Capa oculta 3: 8 neuronas, activación ReLU.
 Capa de salida: 3 neuronas, activación Softmax (una por clase).
```

```
# Crear el modelo neuronal secuencial
model = Sequential()

# Añadir capa de entrada explícitamente (forma recomendada)
model.add(tf.keras.Input(shape=(4,), name='Entrada'))

# Primera capa oculta
model.add(Dense(16, activation='relu', name='Oculta_1'))

# Segunda capa oculta
model.add(Dense(12, activation='relu', name='Oculta_2'))

# Tercera capa oculta
model.add(Dense(8, activation='relu', name='Oculta_3'))

# Capa de salida
model.add(Dense(3, activation='softmax', name='Salida'))

# Compilar modelo = Función de pérdida categorical_crossentropy (apropiada para clasificación multiclase), optimizador adam, métrica accuracy
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])

# Generar el resumen del modelo
model.summary()

# Crear un DataFrame con detalles explícitos del modelo
detalles_modelo = []

# Asegurarse que el modelo esté construido antes
model.build(input_shape=(None, 4))

for capa in model.layers:
    detalles_modelo.append({
        'Capa': capa.name,
        'Tipo': capa.__class__.__name__,
        'Cantidad_neuronas': capa.output.shape[-1],
        'Función_activación': capa.activation.__name__ if hasattr(capa, 'activation') else 'N/A',
        'Forma_salida': capa.output.shape,
        'Parámetros': capa.count_params()
    })

# Crear DataFrame para visualizar mejor los detalles
df_modelo = pd.DataFrame(detalles_modelo)

# Mostrar DataFrame
print("\nArquitectura completa del Modelo:")
print(df_modelo)
```

Model: "sequential"

Layer (type)	Output Shape	Param #
Oculta_1 (Dense)	(None, 16)	80
Oculta_2 (Dense)	(None, 12)	204
Oculta_3 (Dense)	(None, 8)	104
Salida (Dense)	(None, 3)	27

Total params: 415 (1.62 KB)
Trainable params: 415 (1.62 KB)
Non-trainable params: 0 (0.00 B)

```
Arquitectura completa del Modelo:
      Capa   Tipo  Cantidad_neuronas Función_activación Forma_salida  \
0  Oculta_1  Dense                16               relu   (None, 16)
1  Oculta_2  Dense                12               relu   (None, 12)
2  Oculta_3  Dense                 8               relu    (None, 8)
3    Salida  Dense                 3            softmax    (None, 3)

   Parámetros
0          80
1         204
2         104
3          27
```

Explicación del cuadro:

- Capa = Nombre asignado a la capa neuronal
- Tipo = Tipo de capa neuronal (Dense = totalmente conectada)
- Cantidad_neuronas = Número exacto de neuronas en esa capa específica
- Función_activación = Función matemática usada para activar neuronas (relu, softmax.)
- Forma_salida = Forma dimensional de salida generada por cada capa
- Parámetros = Número total de parámetros entrenables por capa

Dense (Fully Connected):

- Cada neurona está conectada con todas las neuronas de la capa anterior, capturando relaciones complejas.

ReLU (Rectified Linear Unit):

- Es una función de activación no lineal que permite aprender relaciones complejas en los datos, evitando ciertos problemas comunes como el desvanecimiento del gradiente.

Softmax:

- La última capa utiliza Softmax para convertir salidas numéricas en probabilidades, ideales para clasificar múltiples categorías (en este caso, las 3 especies de Iris).

El modelo mostrado es una Red Neuronal Secuencial (Sequential) que está estructurada de la siguiente forma:

```
# Arquitectura del Modelo:
Capa Oculta 1 ("Oculta_1"):

Tiene 16 neuronas con activación ReLU.

Salida dimensional: (None, 16).

Total de 80 parámetros entrenables.

Capa Oculta 2 ("Oculta_2"):

Tiene 12 neuronas con activación ReLU.

Salida dimensional: (None, 12).

Total de 204 parámetros entrenables.

Capa Oculta 3 ("Oculta_3"):

Tiene 8 neuronas con activación ReLU.

Salida dimensional: (None, 8).

Total de 104 parámetros entrenables.

Capa de Salida ("Salida"):

Tiene 3 neuronas (una por cada clase Iris).

Usa activación Softmax (ideal para clasificaciones múltiples).

Salida dimensional: (None, 3).

Total de 27 parámetros entrenables.
Total de parámetros entrenables: 415.
```

Conclusión:

Los 415 parámetros entrenables es el total de conexiones (pesos y sesgos) que la red aprenderá durante el entrenamiento.

Cuantos más parámetros, más capacidad tiene la red para aprender patrones complejos, aunque es importante resaltar que en este caso (dataset Iris) es una cantidad equilibrada que evitará un exceso de complejidad innecesaria.

1) Entenamiento y Monitoreo de cómo el modelo (el aprendizaje) aprende a lo largo del tiempo y detecta los problemas tempranos (sobreajuste).

Entrena el modelo por 100 épocas con lotes de 5 muestras.

Se utiliza un 10% del conjunto de entrenamiento como datos de validación interna.

Con verbose=1 se muestra el progreso del entrenamiento, la pérdida y precisión después de cada época.

Parámetros adicionales :

validation_split=0.1: Reserva automáticamente un 10% del conjunto de entrenamiento para monitorear internamente cómo mejora el modelo en datos no vistos durante el entrenamiento.

verbose=1: Muestra progreso detallado (útil para observar directamente cómo evoluciona el entrenamiento).

```
# Entrenar modelo (ajusta epochs según tu preferencia)
historial = model.fit(X_train, y_train, epochs=100, batch_size=5, verbose=1, validation_split=0.1)
```

```
Epoch 1/100
22/22 ━━━━━━━━━━━━━━━━━━━━ 2s 16ms/step - accuracy: 0.2230 - loss: 1.0081 - val_accuracy: 0.4167 - val_loss: 1.0412
Epoch 2/100
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.7207 - loss: 0.9555 - val_accuracy: 0.5833 - val_loss: 1.0144
```

```
Epoch 3/100
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.6803 - loss: 0.9307 - val_accuracy: 0.5833 - val_loss: 0.9802
Epoch 4/100
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.6769 - loss: 0.8776 - val_accuracy: 0.5833 - val_loss: 0.9311
Epoch 5/100
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.6854 - loss: 0.7907 - val_accuracy: 0.6667 - val_loss: 0.8758
Epoch 6/100
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.6530 - loss: 0.7478 - val_accuracy: 0.6667 - val_loss: 0.8119
Epoch 7/100
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - accuracy: 0.7853 - loss: 0.5956 - val_accuracy: 0.9167 - val_loss: 0.7554
Epoch 8/100
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - accuracy: 0.8077 - loss: 0.5506 - val_accuracy: 0.9167 - val_loss: 0.6980
Epoch 9/100
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - accuracy: 0.7694 - loss: 0.5642 - val_accuracy: 0.9167 - val_loss: 0.6422
Epoch 10/100
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 11ms/step - accuracy: 0.8555 - loss: 0.4610 - val_accuracy: 0.8333 - val_loss: 0.5950
Epoch 11/100
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - accuracy: 0.8866 - loss: 0.4153 - val_accuracy: 0.8333 - val_loss: 0.5600
Epoch 12/100
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - accuracy: 0.9363 - loss: 0.4052 - val_accuracy: 0.8333 - val_loss: 0.5198
Epoch 13/100
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - accuracy: 0.8963 - loss: 0.3987 - val_accuracy: 0.8333 - val_loss: 0.4914
Epoch 14/100
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - accuracy: 0.9252 - loss: 0.3379 - val_accuracy: 0.9167 - val_loss: 0.4645
Epoch 15/100
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.9547 - loss: 0.3278 - val_accuracy: 0.9167 - val_loss: 0.4402
Epoch 16/100
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9838 - loss: 0.2355 - val_accuracy: 0.9167 - val_loss: 0.4089
Epoch 17/100
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9736 - loss: 0.2149 - val_accuracy: 0.9167 - val_loss: 0.3993
Epoch 18/100
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.9448 - loss: 0.2316 - val_accuracy: 0.9167 - val_loss: 0.3727
Epoch 19/100
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9606 - loss: 0.1987 - val_accuracy: 0.9167 - val_loss: 0.3667
Epoch 20/100
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9722 - loss: 0.1579 - val_accuracy: 0.9167 - val_loss: 0.3604
Epoch 21/100
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9558 - loss: 0.1527 - val_accuracy: 0.9167 - val_loss: 0.3544
Epoch 22/100
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9837 - loss: 0.1179 - val_accuracy: 0.9167 - val_loss: 0.3568
Epoch 23/100
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9559 - loss: 0.1567 - val_accuracy: 0.9167 - val_loss: 0.3583
Epoch 24/100
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9488 - loss: 0.1265 - val_accuracy: 0.9167 - val_loss: 0.3497
Epoch 25/100
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9517 - loss: 0.1436 - val_accuracy: 0.9167 - val_loss: 0.3573
Epoch 26/100
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9841 - loss: 0.1004 - val_accuracy: 0.9167 - val_loss: 0.3643
Epoch 27/100
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9903 - loss: 0.1008 - val_accuracy: 0.9167 - val_loss: 0.3678
Epoch 28/100
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.9557 - loss: 0.1159 - val_accuracy: 0.9167 - val_loss: 0.3720
Epoch 29/100
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.9787 - loss: 0.0903 - val_accuracy: 0.9167 - val_loss: 0.3726
```

Conclusiones

Aprendizaje:

La pérdida disminuye consistentemente (0.9776 → 0.0697 en entrenamiento, 0.8992 → 0.0174 en validación), indicando que el modelo aprende patrones efectivamente. La precisión mejora de ~68% a ~96% en entrenamiento y de ~66% a 100% en validación, mostrando un aprendizaje sólido.

Sobreajuste:

No hay evidencia clara de sobreajuste, ya que la pérdida de validación y la precisión de validación mejoran junto con las métricas de entrenamiento. Esto sugiere buena generalización.

Épocas:

100 épocas podrían ser excesivas, ya que el modelo estabiliza su rendimiento mucho antes (alrededor de la época 30-40, donde val_accuracy alcanza 1.0).

Validación:

El 10% de validación (12 muestras) es pequeño, pero suficiente para monitorear tendencias en un dataset como Iris.

Observamos que en tiempo real (por cada época) cómo mejora el modelo.

Puedes detectar claramente sobreajuste (si ocurre):

Si la pérdida de entrenamiento baja, pero la de validación no mejora o aumenta, el modelo está memorizando en lugar de generalizar.

Si ambas pérdidas (validación y entrenamiento) disminuyen de forma similar, indica que el modelo generaliza bien.

Finalmente se puede indicar que l modelo aprende rápidamente (en menos de 100 épocas), con una precisión alta tanto en entrenamiento como validación. No se observa sobreajuste significativo. Por lo tanto, el modelo generaliza bien con este dataset

2) Entrenamiento del modelo, pero aquí especificamente evaluamos la precisión y las métricas finales.

Se entrena el modelo por 100 épocas con lotes de 5 muestras por iteración.

Además, se valúa el modelo directamente sobre el conjunto de prueba (X_test, y_test) y muestra la precisión global (Accuracy).

Genera predicciones del conjunto de prueba.

Finalmente, mostramos un reporte detallado de clasificación con métricas individuales para cada clase (Precision, Recall, F1-score).

```
# Entrenar el modelo, recorre todo el dataset de entrenamiento 100 veces.
# batch_size=5 esto hace que se actualizan los pesos tras ver 5 ejemplos cada vez
model.fit(X_train, y_train, epochs=100, batch_size=5)

# Evaluar el modelo
```

```
loss, accuracy = model.evaluate(X_test, y_test, verbose=0)
print(f'Accuracy del modelo: {accuracy*100:.2f}%')

# Obtener reporte detallado
y_pred = model.predict(X_test)
y_pred_classes = np.argmax(y_pred, axis=1)
y_true_classes = np.argmax(y_test, axis=1)

print(classification_report(y_true_classes, y_pred_classes, target_names=encoder.classes_))
```

```
Epoch 1/100
24/24 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.9872 - loss: 0.0564
Epoch 2/100
24/24 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.9953 - loss: 0.0226
Epoch 3/100
24/24 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.9814 - loss: 0.0617
Epoch 4/100
24/24 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.9789 - loss: 0.0973
Epoch 5/100
24/24 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.9926 - loss: 0.0435
Epoch 6/100
24/24 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.9673 - loss: 0.0725
Epoch 7/100
24/24 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9863 - loss: 0.0408
Epoch 8/100
24/24 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.9886 - loss: 0.0260
Epoch 9/100
24/24 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.9955 - loss: 0.0333
Epoch 10/100
24/24 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9955 - loss: 0.0264
Epoch 11/100
24/24 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9695 - loss: 0.0772
Epoch 12/100
24/24 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9891 - loss: 0.0314
Epoch 13/100
24/24 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9877 - loss: 0.0387
Epoch 14/100
24/24 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9891 - loss: 0.0349
Epoch 15/100
24/24 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9986 - loss: 0.0252
Epoch 16/100
24/24 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9929 - loss: 0.0259
Epoch 17/100
24/24 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.9949 - loss: 0.0231
Epoch 18/100
24/24 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.9990 - loss: 0.0246
Epoch 19/100
24/24 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.9914 - loss: 0.0281
Epoch 20/100
24/24 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.9945 - loss: 0.0175
Epoch 21/100
24/24 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.9841 - loss: 0.0285
Epoch 22/100
24/24 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.9815 - loss: 0.0394
Epoch 23/100
24/24 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.9978 - loss: 0.0087
Epoch 24/100
24/24 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.9912 - loss: 0.0228
Epoch 25/100
24/24 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.9841 - loss: 0.0277
Epoch 26/100
24/24 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.9993 - loss: 0.0116
Epoch 27/100
24/24 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.9911 - loss: 0.0234
Epoch 28/100
24/24 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.9990 - loss: 0.0178
Epoch 29/100
24/24 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.9898 - loss: 0.0258
```

Conclusiones:

Accuracy global del modelo (90%): Indicando que el modelo clasifica correctamente 27 de las 30 muestras de prueba, por lo que se permite verificar directamente qué tan bueno es el modelo para clasificar flores Iris desconocidas (datos de prueba).

Reporte de clasificación con métricas como:

Precisión (Precision): Cuántas predicciones positivas son correctas.

Recall (Sensibilidad): Cuántas muestras positivas reales identificó correctamente.

Finalmente se puede decir que el modelo logra una precisión global alta (>85%), lo cual indica que puede clasificar correctamente flores Iris con gran confianza. Las métricas detalladas (precisión, recall, F1-score) demuestran un excelente rendimiento específico para cada clase

Evaluación final del modelo entrenado

```
# Predicciones, se obtiene probabilidades predichas para cada clase del dataset de prueba.
y_pred_prob = model.predict(X_test)
# Se convierte las probabilidades en clases concretas (0, 1 o 2).
y_pred = np.argmax(y_pred_prob, axis=1)
y_test_classes = np.argmax(y_test, axis=1)

# Accuracy, raliza el calculo del porcentaje total de predicciones correctas.
accuracy = accuracy_score(y_test_classes, y_pred)
print(f"\nPrecisión (Accuracy): {accuracy * 100:.2f}%\n")

# Reporte completo de clasificación (precision, recall, f1-score)
print("Reporte completo de clasificación:")
print(classification_report(y_test_classes, y_pred, target_names=encoder.classes_))
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 36ms/step

Precisión (Accuracy): 96.67%

Reporte completo de clasificación:
                precision    recall  f1-score   support
```

```
                 precision    recall  f1-score   support

    Iris-setosa       1.00      1.00      1.00        10
Iris-versicolor       1.00      0.89      0.94         9
 Iris-virginica       0.92      1.00      0.96        11

       accuracy                           0.97        30
      macro avg       0.97      0.96      0.97        30
   weighted avg       0.97      0.97      0.97        30
```

Conclusion final:

(Accuracy del 95 a un 100%)

Clasifica de fcorma correcta Setosa, Versicolor y Virginica, con un rendimiento alto.

El modelo logró una precisión general del 90%.