<a href="https://estudios.upct.es/grado/5251/inicio" target="_blank">
<img src="https://www.upct.es/contenido/universidad/galeria/identidad-2021/logos/logos-upct/
<a href="https://www.um.es/web/estudios/grados/ciencia-ingenieria-datos/" target="_blank">
<img src="https://www.um.es/documents/1073494/42130150/LogosimboloUMU-positivo.png" alt="UMU

Asignatura: Deep Learning

Titulación: Grado en Ciencia e Ingeniería de Datos

Práctica 1: Introducción al Deep Learning

Sesión 1/3: Implementación de modelos de aprendizaje automático

Autores: Juan Morales Sánchez, Antonio Martínez Sánchez, José Luís Sancho Gómez y Juan Antonio Botía Blaya

---

Objetivos

- Familiarización con TensorFlow y Python para *Deep Learning*
- Manipulación de datos con tensores
- Diseño y configuración de modelos supervisados
- Entrenamiento y evaluación de modelos

Contenidos

Bibliografía

- Deep Learning with Python (segunda edición)
- Dive into Deep Learning

Requisitos

- Numpy (computación numérica)
- Scipy (computación científica)
- Scikit-learn (*Machine Learning*)
- Scikit-image (*Image Processing*)
- Matplotlib y Seaborn (visualización de datos)
- Tensorflow 2.x que incluye a Keras 2.x (*Deep Learning*)

Entorno de trabajo

Se trabajará con notebooks de Jupyter con código Python empleando como intérprete la última versión de Miniconda. A continuación se enumeran los pasos a seguir para configurar un entorno virtual de Python adecuado.

Microconda + Visual Studio Code

Para instalar miniconda: 1. Accede al repositorio de miniconda. 2. Descarga e instala la última version (*latest*) disponible para tu sistema operativo.

Para instalar Visual Studio Code: 1. Descarga e instala VS Code desde su sitio web oficial. 2. Instala las extensiones de Python y Jupyter desde VS Code.

Configuración del entorno virtual

1. Abre un nuevo terminal en tu sistema operativo (Linux, Windows o macOS).
2. OPCIÓN 1: Crea un nuevo entorno virtual e instala paquetes desde repositorio de `conda`:

```
conda create --name dl --channel defaults python=3.9
conda activate dl
conda install numpy scipy scikit-learn scikit-image matplotlib seaborn mrcfile ipykernel
```

3. OPCIÓN 2: Crea un nuevo entorno virtual e instala paquetes desde repositorio de `pip` (preferible si su equipo dispone de una GPU Nvidia):

```
conda create --name dl --channel defaults python=3.9 pip
conda activate dl
pip install numpy scipy scikit-learn scikit-image matplotlib seaborn mrcfile ipykernel t
```

4. Para gestionar el entorno, puedes desactivarlo y opcionalmente eliminarlo:

```
conda deactivate
conda env remove --name dl --all
```

Nota: Tenga en cuenta que TensorFlow 2.10 es la última versión de TensorFlow que admite GPU en Windows nativo. A partir de TensorFlow 2.11 para utilizar GPU en Windows es necesario instalar TensorFlow sobre WSL2.

Marcos para aprendizaje profundo

En la práctica las librerías de código abierto para la construcción y entrenamiento de modelos de aprendizaje profundo resultan ideales para implementar redes neuronales y procesar datos complejos como imágenes y texto.

A continuación se resumen en una tabla comparativa las características de las 3 principales iniciativas o marcos de cómputo para *Deep Learning* que cumplen estos requisitos, y que actualmente concentran más de 90% del desarrollo en este campo: PyTorch, TensorFlow y JAX.

Aspecto	JAX	TensorFlow	PyTorch
Origen	Desarrollado por Google Research (inspirado en Autograd)	Creado y mantenido por Google Brain (incluye componentes como Keras, TFLite, TensorFlow.js, etc.)	Desarrollado por Meta AI (anteriormente Facebook), con apoyo de la comunidad de GitHub
Paradigma	Enfoque funcional: usa transformaciones como `jit`, `grad`, `vmap`, etc., sobre sintaxis parecida a NumPy.	Ejecución imperativa por defecto (Eager Execution) y optimizaciones basadas en grafos internos (p. ej., XLA).	Ejecución imperativa (dinámica), muy similar a la programación en Python nativo.
Curva de aprendizaje	Moderada: sintaxis de NumPy es familiar, pero el paradigma funcional (p. ej., `jit`, `vmap`) puede requerir un cambio de mentalidad.	Bastante accesible gracias a la integración con Keras. En casos avanzados, se puede profundizar en niveles de API más bajos (por ej., `tf.*` a bajo nivel).	Normalmente considerada sencilla, útil para prototipado rápido y depuración, gracias a la ejecución imperativa paso a paso.

Aspecto	JAX	TensorFlow	PyTorch
Rendimiento	Muy alto en GPU y TPU gracias a la integración con XLA.	Excelente rendimiento en GPU y TPU; TensorFlow se integra con XLA y ofrece muchas optimizaciones para despliegue en producción (TensorFlow Serving, etc.).	Muy buen rendimiento en GPU (CUDA). Soporte limitado para TPU (existen iniciativas de la comunidad, pero no un soporte oficial completo).
Ventajas	- Control granular sobre transformaciones y compilación- Sintaxis similar a NumPy- Integración potente en TPU.	- Amplio ecosistema (Keras, TFLite, TensorFlow Serving, TensorFlow.js, etc.)- Respaldo empresarial- Facilidad de despliegue	- Flujo imperativo muy intuitivo- Amplia adopción en investigación (especialmente en visión, NLP, etc.)- Gran cantidad de ejemplos y repositorios de la comunidad
Desventajas	- Ecosistema más pequeño que PyTorch/TensorFlow- El enfoque funcional no es tan común y puede ser menos intuitivo para principiantes	- Puede ser abrumador por la gran cantidad de documentación y APIs- A veces requiere atención a la versión y a la configuración de entornos (GPU/TPU)	- No siempre tan optimizado como TensorFlow en escenarios de producción a gran escala- Soporte oficial para TPU inexistente o muy limitado
Comunidad y recursos	En crecimiento, con proyectos recientes en GitHub y foros especializados, aunque menor que las de TensorFlow/PyTorch.	Muy grande, con abundantes recursos de aprendizaje (foros, cursos, documentación oficial, etc.).	Muy activa y enfocada en investigación; gran cantidad de *repos* y tutoriales en GitHub.

En conclusión, se puede afirmar que PyTorch y TensorFlow son hoy por hoy los marcos más utilizadas para *Deep Learning* por diferentes motivos, mientras que JAX se está abriendo paso especialmente en investigación y proyectos que busquen optimizaciones avanzadas mediante XLA.

En las prácticas de esta asignatura nos utilizaremos TensorFlow, al considerar que el interfaz de integrado de Keras ofrece la alternativa más accesible para un primer contacto con la implmentación de modelos de aprendizaje automático. Y a largo plazo también parece ofrecer una visión más amplia y completa de posibilidades y usos dentro del ámbito académico y profesional. La experiencia adquirida debería permitir una fácil adaptación a otros entornos como PyTorch, e incluso una migración e código bastante directa.

Aprendizaje supervisado

El **aprendizaje supervisado** constituye una categoría del aprendizaje automático en la cual se enseña a un modelo a realizar predicciones o clasificaciones basándose en un conjunto de datos etiquetados. Este tipo de aprendizaje se denomina "supervisado" porque el proceso de entrenamiento está guiado por una supervisión explícita a través de etiquetas que indican las respuestas correctas.

Tipos de Aprendizaje Supervisado

1. **Regresión**:
   - El objetivo es predecir valores continuos.
   - Ejemplo: Predecir el precio de una vivienda en función de sus características.
2. **Clasificación**:
   - El objetivo es asignar categorías a las entradas.
   - Ejemplo: Clasificar correos electrónicos como "spam" o "no spam".

Componentes del aprendizaje supervisado

1. **Datos de entrenamiento**:
   - Conjunto de ejemplos que incluyen:
     - **Entradas** (X): Características o variables independientes.
     - **Etiquetas** (Y): Salidas esperadas o variables dependientes (valores conocidos).
2. **Modelo**:
   - Una función f(X) que el sistema entrena para aproximar la relación entre X y Y.
3. **Coste o pérdida (error)**:
   - Una métrica que evalúa lo bien que las predicciones del modelo coinciden con las etiquetas reales.
4. **Optimización**:
   - Proceso de ajuste de los parámetros del modelo para minimizar la pérdida mediante algoritmos como el descenso estocástico de gradiente.

```
#### Cargando el conjunto de datos MNIST
import numpy as np
from tensorflow.keras.datasets import mnist
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
```

```
2025-01-24 08:51:06.514648: E external/local_xla/xla/stream_executor/cuda/cuda_fft.cc:477] Un
WARNING: All log messages before absl::InitializeLog() is called are written to STDERR
E0000 00:00:1737708666.529277   27217 cuda_dnn.cc:8310] Unable to register cuDNN factory: Att
E0000 00:00:1737708666.533731   27217 cuda_blas.cc:1418] Unable to register cuBLAS factory: A
2025-01-24 08:51:06.550751: I tensorflow/core/platform/cpu_feature_guard.cc:210] This Tensorl
To enable the following instructions: AVX2 FMA, in other operations, rebuild TensorFlow with
```

Atributos clave de un conjunto de datos: rango, dimensiones y tipo

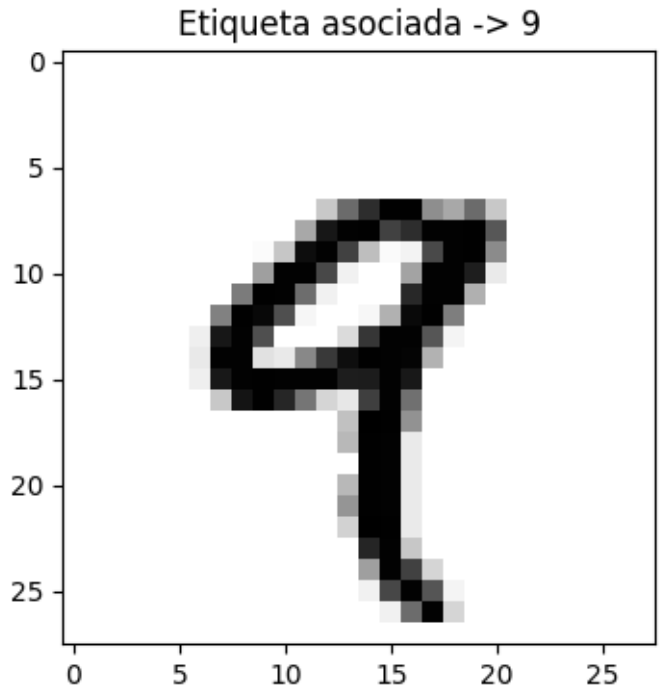
```
print("Rango del tensor del conjunto de entrenamiento:\n", train_images.ndim)
print("Dimensiones del conjunto de entrenamiento [samples x height x width]:\n", train_images
print("Número de etiquetas del conjunto de entrenamiento:\n", len(train_labels))
print("Tipo del conjunto de entrenamiento:\n", type(train_images), train_images.dtype)
print("Tipo de las etiquetas de entrenamiento:\n", type(train_labels), train_labels.dtype)
print("Rango dinánico de valores del conjunto de entrenamiento:\n", [np.min(train_images), np
print("Valores de las etiquetas de entrenamiento:\n", np.unique(train_labels))

print("Rango del tensor del conjunto de test:\n", test_images.ndim)
print("Dimensiones del conjunto de test [samples x height x width]:\n", test_images.shape)
print("Número de etiquetas del conjunto de test:\n", len(test_labels))
print("Tipo del conjunto de test:\n", type(test_images), test_images.dtype)
print("Tipo de las etiquetas de test:\n", type(test_labels), test_labels.dtype)
print("Rango dinánico de valores del conjunto de entrenamiento:\n", [np.min(test_images), np
print("Valores de las etiquetas de entrenamiento:\n", np.unique(test_labels))
```

```
Rango del tensor del conjunto de entrenamiento:
 3
Dimensiones del conjunto de entrenamiento [samples x height x width]:
 (60000, 28, 28)
Número de etiquetas del conjunto de entrenamiento:
 60000
Tipo del conjunto de entrenamiento:
 <class 'numpy.ndarray'> uint8
Tipo de las etiquetas de entrenamiento:
 <class 'numpy.ndarray'> uint8
Rango dinánico de valores del conjunto de entrenamiento:
 [np.uint8(0), np.uint8(255)]
Valores de las etiquetas de entrenamiento:
 [0 1 2 3 4 5 6 7 8 9]
Rango del tensor del conjunto de test:
```

```
 3
Dimensiones del conjunto de test [samples x height x width]:
 (10000, 28, 28)
Número de etiquetas del conjunto de test:
 10000
Tipo del conjunto de test:
 <class 'numpy.ndarray'> uint8
Tipo de las etiquetas de test:
 <class 'numpy.ndarray'> uint8
Rango dinánico de valores del conjunto de entrenamiento:
 [np.uint8(0), np.uint8(255)]
Valores de las etiquetas de entrenamiento:
 [0 1 2 3 4 5 6 7 8 9]
```

Visualización de datos y etiquetas

```
#### Visualizando el quinta muestra del conjunto de imágenes y su etiqueta asociada
import matplotlib.pyplot as plt
digit = train_images[4]
plt.figure(figsize=(5, 4))
plt.imshow(digit, cmap=plt.cm.binary)
plt.title(f"Etiqueta asociada -> {train_labels[4]}")
plt.show()
```



Acondicionaniento de datos

El **acondicionamiento de datos** es una etapa fundamental en el flujo de trabajo del aprendizaje automático y redes neuronales en general. Consiste en preparar los datos para que sean compatibles con los modelos y para mejorar el rendimiento y estabilidad del entrenamiento. Esta preparación incluye varias operaciones que ajustan los datos a los formatos, rangos o estructuras requeridas. Entre las operaciones típicas que podemos englobar dentro de esta categoría se encuentran habitualmente:

1. **Normalización de datos**:
    - Escalar los valores de las características a un rango común, como [0, 1] ó [-1, 1].
    - Ejemplo: Dividir los valores de píxel de imágenes (en el rango 0-255) por 255 para normalizarlos entre [0, 1].
    - Beneficio:
        - Mejora la estabilidad del modelo.
        - Acelera la convergencia durante el entrenamiento.
2. **Aplanamiento de dimensiones**:
    - Transformar datos multidimensionales en vectores unidimensionales.
    - Ejemplo: Imágenes 2D (28x28 píxeles) a un vector de 784 elementos para alimentarlas en capas densas.

- Beneficio:
  - Permite la entrada de datos a modelos que requieren estructuras lineales.

3. **Conversión de tipo de datos**:
   - Cambiar el tipo de los datos o las etiquetas, como de enteros (`int`, `uint`) a coma flotante (`float`).
   - Ejemplo: Convertir etiquetas categóricas a `float32` para cálculos numéricos.
   - Beneficio:
     - Asegura compatibilidad con el diseño del modelo.

```
train_images = train_images.reshape((60000, 28 * 28))  # Aplanamiento
train_images = train_images.astype("float32") / 255  # Normalización + tipo flotante
test_images = test_images.reshape((10000, 28 * 28))  # Aplanamiento
test_images = test_images.astype("float32") / 255  # Normalización + tipo flotante

print("Rango del tensor del conjunto de entrenamiento:\n", train_images.ndim)
print("Dimensiones del conjunto de entrenamiento [samples x height x width]:\n", train_images
print("Número de etiquetas del conjunto de entrenamiento:\n", len(train_labels))
print("Tipo del conjunto de entrenamiento:\n", type(train_images), train_images.dtype)
print("Tipo de las etiquetas de entrenamiento:\n", type(train_labels), train_labels.dtype)
print("Rango dinánico de valores del conjunto de entrenamiento:\n", [np.min(train_images), np
print("Valores de las etiquetas de entrenamiento:\n", np.unique(train_labels))

print("Rango del tensor del conjunto de test:\n", test_images.ndim)
print("Dimensiones del conjunto de test [samples x height x width]:\n", test_images.shape)
print("Número de etiquetas del conjunto de test:\n", len(test_labels))
print("Tipo del conjunto de test:\n", type(test_images), test_images.dtype)
print("Tipo de las etiquetas de test:\n", type(test_labels), test_labels.dtype)
print("Rango dinánico de valores del conjunto de entrenamiento:\n", [np.min(test_images), np
print("Valores de las etiquetas de entrenamiento:\n", np.unique(test_labels))
```

```
Rango del tensor del conjunto de entrenamiento:
 2
Dimensiones del conjunto de entrenamiento [samples x height x width]:
 (60000, 784)
Número de etiquetas del conjunto de entrenamiento:
 60000
Tipo del conjunto de entrenamiento:
 <class 'numpy.ndarray'> float32
Tipo de las etiquetas de entrenamiento:
 <class 'numpy.ndarray'> uint8
Rango dinánico de valores del conjunto de entrenamiento:
```

```
 [np.float32(0.0), np.float32(1.0)]
Valores de las etiquetas de entrenamiento:
 [0 1 2 3 4 5 6 7 8 9]
Rango del tensor del conjunto de test:
 2
Dimensiones del conjunto de test [samples x height x width]:
 (10000, 784)
Número de etiquetas del conjunto de test:
 10000
Tipo del conjunto de test:
 <class 'numpy.ndarray'> float32
Tipo de las etiquetas de test:
 <class 'numpy.ndarray'> uint8
Rango dinánico de valores del conjunto de entrenamiento:
 [np.float32(0.0), np.float32(1.0)]
Valores de las etiquetas de entrenamiento:
 [0 1 2 3 4 5 6 7 8 9]
```

Noción de lote de datos

Recordemos en primer lugar que una **época** es un ciclo completo en el que el modelo procesa todo el conjunto de datos de entrenamiento una vez. Durante una época, el modelo utiliza todos los ejemplos del conjunto de datos para calcular los gradientes y ajustar los pesos. Por otra parte, y a lo largo de cada época, un **lote de datos** (*batch*) es un subconjunto del conjunto de datos de entrenamiento que el modelo procesa en cada iteración del entrenamiento.

Por tanto, en lugar de procesar todos los datos de entrenamiento de una vez en cada época, se dividen en pequeños grupos (lotes), y el modelo ajusta sus parámetros utilizando únicamente los datos del lote actual (*Mini-Batch Training*). Algunas de las ventajas del uso de lotes de datos son:

1. **Eficiencia computacional**:
    - Procesar datos en lotes aprovecha la computación paralela de GPUs/TPUs.
2. **Mejor estimación del gradiente**:
    - Mini-batches ofrecen un compromiso entre ruido y precisión en el cálculo del gradiente.
3. **Eficiencia del uso de memoria**:
    - Los lotes permiten entrenar con conjuntos de datos grandes sin necesidad de cargar todo el conjunto en memoria.

```
n = 3 # lote número 3
batch_size = 128 # tamaño del lote
batch = train_images[batch_size * n:batch_size * (n + 1)]
label_batch = train_labels[batch_size * n:batch_size * (n + 1)]
print("Dimensiones del lote de datos [samples x length]:\n", batch.shape)
print("Dimensiones del lote de etiquetas [samples]:\n", len(label_batch))
```

```
Dimensiones del lote de datos [samples x length]:
 (128, 784)
Dimensiones del lote de etiquetas [samples]:
 128
```

La arquitectura de red

La **arquitectura de red** refleja la estructura y diseño de una red neuronal, incluyendo las capas, cómo están conectadas, los tipos de operaciones que realizan y los hiperparámetros asociados. Es una especificación que define cómo se organiza la red para procesar datos y alcanzar predicciones.

La arquitectura influye directamente en la capacidad del modelo para aprender patrones de los datos. Elegir la arquitectura correcta es el primer requisito para un buen rendimiento del modelo y su capacidad de generalización. Algunos componentes claves de la arquitectura de red son:

1. **Capas**:
    - Agrupación de operaciones de un determinado tipo, que transforman los datos desde la entrada hasta la salida.
    - Ejemplo:
        - Capas densas (*fully connected*).
        - Capas convolucionales (*convolutional layers*).
        - Capas recurrentes (*recurrent layers*).
2. **Neuronas por capa**:
    - El número de neuronas determina la capacidad de cada capa para reconocer patrones. El número de capas y la cantidad de neuronas de cada capa determina el número total de parámetros o pesos entrenables del modelo.
3. **Funciones de activación**:
    - Introducen no linealidades para que la red pueda aprender relaciones complejas.
    - Ejemplo: `ReLU`, `sigmoid`, `tanh`.

4. **Conexiones entre capas**:
    - Puede fluir solo hacía delante (*feedforward*) o incluir retroalimentación (como en redes recurrentes).
5. **Hiperparámetros**:
    - Aspectos configurables de la red, como el número de capas, tamaño del lote, tasa de aprendizaje, etc.

Existen varias formas de definir la arquitectura de un modelo de aprendizaje automático en TensorFlow:

```
# Definición secuencial (opción 1)
from tensorflow import keras
from tensorflow.keras import layers, Sequential, Model
model = Sequential([
    layers.Dense(512, activation="relu", input_shape=(28 * 28, )), # capa oculta
    layers.Dense(10, activation="softmax") # capa de salida
])
model.summary()

# Definición secuencial (opción 2)
model = Sequential()
model.add(layers.Dense(512, activation="relu", input_shape=(28 * 28, ))) # capa oculta
model.add(layers.Dense(10, activation="softmax")) # capa de salida
model.summary()

# Definición funcional
input_layer = layers.Input(shape=(28 * 28, ))  # Capa de entrada
hidden_layer = layers.Dense(512, activation="relu")(input_layer)  # Capa oculta
output_layer = layers.Dense(10, activation="softmax")(hidden_layer)  # Capa de salida
model = Model(inputs=input_layer, outputs=output_layer)
model.summary()
```

```
/usr/local/python/3.12.1/lib/python3.12/site-packages/keras/src/layers/core/dense.py:87: User
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
2025-01-24 08:51:09.636792: E external/local_xla/xla/stream_executor/cuda/cuda_driver.cc:152]
```

```
Model: "sequential"
```

```
 Layer (type)                    Output Shape                  Param #
```

```
 dense (Dense)                    (None, 512)                   401,920

 dense_1 (Dense)                  (None, 10)                      5,130

 Total params: 407,050 (1.55 MB)

 Trainable params: 407,050 (1.55 MB)

 Non-trainable params: 0 (0.00 B)
```

```
Model: "sequential_1"
```

```
 Layer (type)                     Output Shape                  Param #

 dense_2 (Dense)                  (None, 512)                   401,920

 dense_3 (Dense)                  (None, 10)                      5,130

 Total params: 407,050 (1.55 MB)

 Trainable params: 407,050 (1.55 MB)

 Non-trainable params: 0 (0.00 B)
```

```
Model: "functional_3"
```

```
 Layer (type)                     Output Shape                  Param #

 input_layer_2 (InputLayer)       (None, 784)                         0

 dense_4 (Dense)                  (None, 512)                   401,920

 dense_5 (Dense)                  (None, 10)                      5,130
```

```
 Total params: 407,050 (1.55 MB)

 Trainable params: 407,050 (1.55 MB)

 Non-trainable params: 0 (0.00 B)
```

Observa que con una definición `Sequential` del modelo la `InputLayer` no aparece en el resumen, incluso si se incluye explícitamente. En `Functional`, siempre aparece porque la entrada es explícita como parte del diseño del modelo. Esto sucede porque en el flujo típico de `Sequential` la capa de entrada se infiere automáticamente a partir de los datos proporcionados en la primera capa, y no se considera como una capa separada en el grafo de computación, si no como parte del flujo de datos implícito, lo cual es más simple y resulta óptimo para arquitecturas convencionales, en particular aquellas que tienen una sola entrada.

Configurar el modelo para el aprendizaje

La configuración y compilación del modelo conlleva definir las siguientes componentes claves:

- **Función de coste o pérdida**:
  - Establece una métrica para valorar lo bien/mal que está prediciendo el modelo.
  - Ejemplo: `mean squared error` para regresión, `categorical crossentropy` para clasificación.
- **Optimizador**:
  - Algoritmo que ajusta los parámetros del modelo para minimizar la pérdida.
  - Ejemplo: `Adam`, `SGD`.
- **Métricas adicionales**:
  - Evalúan las prestaciones del modelo durante el entrenamiento y la validación.
  - Ejemplo: precisión (`accuracy`), error absoluto medio (`MAE`).

```python
model.compile(
    optimizer='rmsprop',                        # Optimizador
    loss='sparse_categorical_crossentropy',     # Función de pérdida
    metrics=['accuracy']                        # Métrica
)
```

Entrenamiento del modelo

El método `fit` de TensorFlow es una función fundamental en la API de Keras que entrena un modelo de aprendizaje profundo utilizando un conjunto de datos. Este método ejecuta el proceso de entrenamiento, integrando todas las fases necesarias: cálculo de pérdidas, optimización, y evaluación de métricas.

Al llamar al método `fit`, TensorFlow sigue estos pasos:

1. **Preparación del entrenamiento**:
   - Verifica que el modelo esté correctamente compilado con:
     - Una función de pérdida (`loss`).
     - Un optimizador (`optimizer`).
     - Opcionalmente, métricas (`metrics`) para evaluar las prestaciones.
2. **Procesamiento de los datos**:
   - Divide los datos en lotes según el tamaño definido en `batch_size`.
   - Si se emplea validación, separa una parte de los datos para evaluación.
3. **Bucle de entrenamiento**: Para cada época:
   - **Paso hacia adelante (*forward pass*)**:
     - Calcula las predicciones del modelo para un lote de datos.
   - **Cálculo de la pérdida**:
     - Compara las predicciones con las etiquetas reales usando la función de pérdida.
   - **Paso hacia atrás (*backward pass*)**:
     - Calcula los gradientes de los parámetros del modelo con respecto a la pérdida.
   - **Actualización de los pesos**:
     - Ajusta los pesos del modelo utilizando el optimizador configurado.
4. **Validación (opcional)**:
   - Al final de cada época, evalúa el modelo en el conjunto de validación.
5. **Almacenamiento de resultados**:
   - Guarda métricas como la pérdida y otras definidas en `metrics` para cada época.

El método `fit` devuelve un histórico (objeto `history`) que contiene un registro de las prestaciones del modelo durante el entrenamiento, mediante un diccionario con las métricas por época (`loss`, `mae`, `val_loss`, etc.). Además `fit` es compatible con `callbacks` para personalizar y extender el entrenamiento.

```python
history = model.fit(
    train_images, train_labels, # datos y etiquetas de entrenamiento
    epochs=5,                # Número de épocas
    batch_size=64            # Tamaño del lote
)
```

```
Epoch 1/5

2025-01-24 08:51:09.806334: W external/local_xla/xla/tsl/framework/cpu_allocator_impl.cc:83]

938/938             3s 2ms/step - accuracy: 0.8916 - loss: 0.3733
Epoch 2/5
938/938             2s 2ms/step - accuracy: 0.9696 - loss: 0.1004
Epoch 3/5
938/938             2s 2ms/step - accuracy: 0.9820 - loss: 0.0608
Epoch 4/5
938/938             2s 2ms/step - accuracy: 0.9872 - loss: 0.0421
Epoch 5/5
938/938             3s 2ms/step - accuracy: 0.9894 - loss: 0.0331
```

Inferencia o predicciones del modelo

El método `predict` de TensorFlow en la API de Keras se utiliza para realizar **inferencia** con un modelo ya entrenado. Es decir, genera predicciones a partir de nuevos datos de entrada, aplicando únicamente la fase de **propagación hacia adelante** (*forward pass*), sin actualizar los pesos del modelo, es decir realiza los siguientes pasos:

1. **Propagación hacia adelante**:
    - Toma los datos de entrada y los pasa a través de las capas del modelo.
    - Realiza todas las operaciones definidas en el modelo, como multiplicaciones de tensores, funciones de activación y normalización.
2. **Cálculo de las salidas**:
    - Produce una salida en función de las configuraciones del modelo:
        - Para problemas de regresión, devuelve valores continuos: el valor estimado por el modelo basado en los datos de entrada.
        - Para problemas de clasificación, devuelve probabilidades de cada clase (si la última capa es una función `softmax`). Normalmente se elige la clase con mayor probabilidad para obtener las clases predichas (`np.argmax(y_pred, axis=-1)`).

3. **Agrupación en lotes (opcional)**:
    - Divide los datos en lotes si el conjunto de datos es grande, para optimizar el uso de memoria.

```
test_digits = test_images[0:20]
predictions = model.predict(test_digits) # probabilidades de clase (one hot encoding)
predictions = predictions.argmax(axis=-1) # clase de mayor probabilidad

for i, pred in enumerate(predictions):
    print(f"Etiqueta: {test_labels[i]}, Predicción: {pred}")
```

```
1/1             0s 40ms/step
Etiqueta: 7, Predicción: 7
Etiqueta: 2, Predicción: 2
Etiqueta: 1, Predicción: 1
Etiqueta: 0, Predicción: 0
Etiqueta: 4, Predicción: 4
Etiqueta: 1, Predicción: 1
Etiqueta: 4, Predicción: 4
Etiqueta: 9, Predicción: 9
Etiqueta: 5, Predicción: 5
Etiqueta: 9, Predicción: 9
Etiqueta: 0, Predicción: 0
Etiqueta: 6, Predicción: 6
Etiqueta: 9, Predicción: 9
Etiqueta: 0, Predicción: 0
Etiqueta: 1, Predicción: 1
Etiqueta: 5, Predicción: 5
Etiqueta: 9, Predicción: 9
Etiqueta: 7, Predicción: 7
Etiqueta: 3, Predicción: 3
Etiqueta: 4, Predicción: 4
```

Evaluación del modelo

La función `evaluate` de TensorFlow en la API de Keras evalúa el modelo entrenado en un conjunto de datos específico, calculando la pérdida y las métricas definidas durante la compilación del modelo. Es una forma de medir el rendimiento del modelo, ya sea en datos de validación o de test, sin modificar los parámetros del modelo. Conlleva la siguiente secuencia:

1. **Propagación hacia adelante (*forward pass*)**:

- Pasa los datos de entrada a través del modelo para generar predicciones.

2. **Cálculo de la pérdida**:
   - Compara las predicciones con las etiquetas reales utilizando la función de pérdida definida.
3. **Cálculo de métricas**:
   - Evalúa las métricas configuradas al compilar el modelo, como precisión, MAE, etc.
4. **Agrupación en lotes (opcional)**:
   - Si el conjunto de datos es grande, divide los datos en lotes para optimizar el uso de memoria.
5. **Promedio de resultados**:
   - Calcula la pérdida y las métricas promedio en todos los lotes.

```
test_loss, test_acc = model.evaluate(test_images, test_labels)
print(f"test_acc: {test_acc:.4f}")
```

```
313/313             0s 1ms/step - accuracy: 0.9741 - loss: 0.0910
test_acc: 0.9784
```

Ejercicios

E1: Crear un modelo de clasificación binaria para predecir si la suma de dos números es mayor a 1 usando un conjunto de datos sintético.

- Se generan 1000 muestras con dos características cada una (`x_data`).
- La etiqueta (`y_data`) es `1` si la suma de las dos características es mayor que 1, y `0` en caso contrario.
  - Ejemplo:
    - Entrada: `[0.6, 0.7]` → Suma: `1.3` → Etiqueta: `1`.
    - Entrada: `[0.3, 0.2]` → Suma: `0.5` → Etiqueta: `0`.
- Se divide el conjunto en entrenamiento (80%) y prueba (20%).
- Construir una red neuronal con dos capas:
  - Capa oculta: 64 neuronas y función de activación `relu`.
  - Capa de salida: 1 neurona con activación `sigmoid` para obtener una probabilidad entre 0 y 1 (para clasificación binaria).

- Optimizador: `adam`, que ajusta los pesos durante el entrenamiento.
- Pérdida: `binary_crossentropy`, que es adecuada para problemas de clasificación binaria.
- Métrica: `accuracy`, que mide el porcentaje de predicciones correctas.
- Se entrena en el conjunto de entrenamiento durante 30 épocas con un tamaño de lote de 32.
- Utiliza el 20% de los datos de entrenamiento como conjunto de validación para monitorear el rendimiento.
- El modelo se evalúa en el conjunto de prueba, calculando la pérdida y la precisión.
- Se realizan predicciones en nuevas muestras (`x_sample`).
- La salida del modelo son probabilidades (valores entre 0 y 1), y se convierten en clases (`0` o `1`) comparando con un umbral de 0.5.
  - Ejemplo:
    - Entrada: `[0.6, 0.6]` → Probabilidad: `0.95` → Clase: `1`.

E2: Cambiar el tamaño de la red (número de neuronas o capas) y analizar el impacto.

E3: Probar diferentes funciones de activación y analizar el impacto.

E4: Crear un modelo de regresión para predecir la suma de dos números usando un conjunto de datos sintético..

- Se generan números aleatorios para las entradas `x_1` y `x_2`, separándolo un 30% de ellos para test.
- Las etiquetas `y` son la suma de `x_1 + x_2`.
- Una red neuronal simple con:
  - Una capa oculta de 64 neuronas con activación ReLU.
  - Una capa de salida con 1 neurona (regresión).
- Optimización con Adam.
- Pérdida: `mse` (error cuadrático medio), adecuada para problemas de regresión.
- Métrica: `mae` (error absoluto medio), que mide la desviación promedio.
- Se entrena el modelo durante 20 épocas con un tamaño de lote de 32.
- El 20% de los datos se reserva para validación en el entrenamiento.
- El modelo se evalúa en el conjunto de test no visto durante el entrenamiento.

- Se realiza inferencia con nuevas muestras para verificar el rendimiento del modelo, por ejemplo para una entrada como `[0.3, 0.7]`, la salida debería estar cerca de `1.0`.

E5: Cambiar el tamaño del conjunto de datos y analizar el impacto.

E6: Aumentar la complejidad del problema (por ejemplo, usando ruido en las etiquetas).

```
# E1
import numpy as np
from sklearn.model_selection import train_test_split
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.optimizers import Adam

# Generar datos sintéticos
np.random.seed(42)
x_data = np.random.rand(1000, 2)
y_data = (np.sum(x_data, axis=1) > 1).astype(int)

# Dividir el conjunto de datos en entrenamiento (80%) y prueba (20%)
x_train, x_test, y_train, y_test = train_test_split(x_data, y_data, test_size=0.2, random_sta

# Construir el modelo
model = Sequential([
    Dense(64, activation="relu", input_shape=(2,)),
    Dense(1, activation="sigmoid")
])

# Compilar el modelo
model.compile(optimizer=Adam(), loss="binary_crossentropy", metrics=["accuracy"])

# Entrenar el modelo
history = model.fit(x_train, y_train, epochs=30, batch_size=32, validation_split=0.2)

loss, accuracy = model.evaluate(x_test, y_test)
print(f"Pérdida en el conjunto de prueba: {loss}")
print(f"Precisión en el conjunto de prueba: {accuracy}")

# Nuevas pruebas para predicción
x_sample = np.array([[0.6, 0.6], [0.3, 0.2]])

# Realizar predicciones
probabilidades = model.predict(x_sample)
```

```python
clases = (probabilidades > 0.5).astype(int)

# Mostrar resultados
for i, sample in enumerate(x_sample):
    print(f"Entrada: {sample} → Probabilidad: {probabilidades[i][0]:.2f} → Clase: {clases[i]
```

```
Epoch 1/30
20/20             1s 8ms/step - accuracy: 0.4875 - loss: 0.6709 - val_accuracy: 0.6062 - val_
Epoch 2/30
20/20             0s 3ms/step - accuracy: 0.5317 - loss: 0.6523 - val_accuracy: 0.6250 - val_
Epoch 3/30
20/20             0s 3ms/step - accuracy: 0.5715 - loss: 0.6332 - val_accuracy: 0.6500 - val_
Epoch 4/30
20/20             0s 3ms/step - accuracy: 0.6269 - loss: 0.6157 - val_accuracy: 0.6750 - val_
Epoch 5/30
20/20             0s 3ms/step - accuracy: 0.6725 - loss: 0.5931 - val_accuracy: 0.7125 - val_
Epoch 6/30
20/20             0s 3ms/step - accuracy: 0.6966 - loss: 0.5801 - val_accuracy: 0.7563 - val_
Epoch 7/30
20/20             0s 3ms/step - accuracy: 0.7579 - loss: 0.5566 - val_accuracy: 0.7812 - val_
Epoch 8/30
20/20             0s 4ms/step - accuracy: 0.7854 - loss: 0.5246 - val_accuracy: 0.8500 - val_
Epoch 9/30
20/20             0s 3ms/step - accuracy: 0.8163 - loss: 0.4924 - val_accuracy: 0.8813 - val_
Epoch 10/30
20/20             0s 3ms/step - accuracy: 0.8512 - loss: 0.4766 - val_accuracy: 0.9062 - val_
Epoch 11/30
20/20             0s 3ms/step - accuracy: 0.8708 - loss: 0.4569 - val_accuracy: 0.9250 - val_
Epoch 12/30
20/20             0s 3ms/step - accuracy: 0.8715 - loss: 0.4310 - val_accuracy: 0.9375 - val_
Epoch 13/30
20/20             0s 3ms/step - accuracy: 0.9098 - loss: 0.4153 - val_accuracy: 0.9563 - val_
Epoch 14/30
20/20             0s 3ms/step - accuracy: 0.9219 - loss: 0.3809 - val_accuracy: 0.9563 - val_
Epoch 15/30
20/20             0s 3ms/step - accuracy: 0.9304 - loss: 0.3622 - val_accuracy: 0.9688 - val_
Epoch 16/30
20/20             0s 3ms/step - accuracy: 0.9279 - loss: 0.3489 - val_accuracy: 0.9688 - val_
Epoch 17/30
20/20             0s 3ms/step - accuracy: 0.9436 - loss: 0.3280 - val_accuracy: 0.9750 - val_
Epoch 18/30
20/20             0s 3ms/step - accuracy: 0.9382 - loss: 0.3109 - val_accuracy: 0.9750 - val_
```

```
Epoch 19/30
20/20             0s 3ms/step - accuracy: 0.9573 - loss: 0.2995 - val_accuracy: 0.9812 - val_
Epoch 20/30
20/20             0s 3ms/step - accuracy: 0.9635 - loss: 0.2795 - val_accuracy: 0.9875 - val_
Epoch 21/30
20/20             0s 3ms/step - accuracy: 0.9494 - loss: 0.2806 - val_accuracy: 0.9875 - val_
Epoch 22/30
20/20             0s 3ms/step - accuracy: 0.9578 - loss: 0.2657 - val_accuracy: 0.9875 - val_
Epoch 23/30
20/20             0s 3ms/step - accuracy: 0.9692 - loss: 0.2501 - val_accuracy: 0.9875 - val_
Epoch 24/30
20/20             0s 3ms/step - accuracy: 0.9721 - loss: 0.2308 - val_accuracy: 0.9875 - val_
Epoch 25/30
20/20             0s 3ms/step - accuracy: 0.9791 - loss: 0.2235 - val_accuracy: 0.9875 - val_
Epoch 26/30
20/20             0s 3ms/step - accuracy: 0.9829 - loss: 0.2168 - val_accuracy: 0.9875 - val_
Epoch 27/30
20/20             0s 3ms/step - accuracy: 0.9737 - loss: 0.2061 - val_accuracy: 0.9875 - val_
Epoch 28/30
20/20             0s 3ms/step - accuracy: 0.9739 - loss: 0.2113 - val_accuracy: 0.9875 - val_
Epoch 29/30
20/20             0s 3ms/step - accuracy: 0.9785 - loss: 0.1895 - val_accuracy: 0.9875 - val_
Epoch 30/30
20/20             0s 3ms/step - accuracy: 0.9830 - loss: 0.1990 - val_accuracy: 0.9875 - val_
7/7             0s 3ms/step - accuracy: 0.9987 - loss: 0.1817
Pérdida en el conjunto de prueba: 0.17997205257415771
Precisión en el conjunto de prueba: 0.9950000047683716
1/1             0s 38ms/step
Entrada: [0.6 0.6] → Probabilidad: 0.83 → Clase: 1
Entrada: [0.3 0.2] → Probabilidad: 0.03 → Clase: 0
```

```python
# E2
# Construir el modelo con una capa oculta de 32 neuronas
model_32 = Sequential([
    Dense(32, activation="relu", input_shape=(2,)),
    Dense(1, activation="sigmoid")
])

# Compilar el modelo
model_32.compile(optimizer=Adam(), loss="binary_crossentropy", metrics=["accuracy"])

# Entrenar el modelo
```

```python
history_32 = model_32.fit(x_train, y_train, epochs=30, batch_size=32, validation_split=0.2)

# Evaluar el modelo

loss_32, accuracy_32 = model_32.evaluate(x_test, y_test)
loss, accuracy = model.evaluate(x_test, y_test)
print(f"Pérdida en el conjunto de prueba (32 neuronas): {loss_32}")
print(f"Precisión en el conjunto de prueba (32 neuronas): {accuracy_32}")
```

```
Epoch 1/30
20/20             1s 8ms/step - accuracy: 0.4980 - loss: 0.6648 - val_accuracy: 0.6000 - val_
Epoch 2/30
20/20             0s 3ms/step - accuracy: 0.5564 - loss: 0.6475 - val_accuracy: 0.6125 - val_
Epoch 3/30
20/20             0s 3ms/step - accuracy: 0.5759 - loss: 0.6392 - val_accuracy: 0.6375 - val_
Epoch 4/30
20/20             0s 3ms/step - accuracy: 0.6072 - loss: 0.6286 - val_accuracy: 0.6812 - val_
Epoch 5/30
20/20             0s 3ms/step - accuracy: 0.6537 - loss: 0.6112 - val_accuracy: 0.7063 - val_
Epoch 6/30
20/20             0s 3ms/step - accuracy: 0.6785 - loss: 0.6080 - val_accuracy: 0.7250 - val_
Epoch 7/30
20/20             0s 3ms/step - accuracy: 0.7104 - loss: 0.5870 - val_accuracy: 0.7500 - val_
Epoch 8/30
20/20             0s 3ms/step - accuracy: 0.7511 - loss: 0.5629 - val_accuracy: 0.7875 - val_
Epoch 9/30
20/20             0s 3ms/step - accuracy: 0.7631 - loss: 0.5643 - val_accuracy: 0.8062 - val_
Epoch 10/30
20/20             0s 3ms/step - accuracy: 0.7880 - loss: 0.5500 - val_accuracy: 0.8188 - val_
Epoch 11/30
20/20             0s 3ms/step - accuracy: 0.8412 - loss: 0.5234 - val_accuracy: 0.8438 - val_
Epoch 12/30
20/20             0s 3ms/step - accuracy: 0.8567 - loss: 0.5039 - val_accuracy: 0.8813 - val_
Epoch 13/30
20/20             0s 3ms/step - accuracy: 0.8463 - loss: 0.4945 - val_accuracy: 0.9062 - val_
Epoch 14/30
20/20             0s 3ms/step - accuracy: 0.8831 - loss: 0.4722 - val_accuracy: 0.9125 - val_
Epoch 15/30
20/20             0s 3ms/step - accuracy: 0.8858 - loss: 0.4568 - val_accuracy: 0.9250 - val_
Epoch 16/30
20/20             0s 3ms/step - accuracy: 0.9058 - loss: 0.4420 - val_accuracy: 0.9187 - val_
Epoch 17/30
```

```
20/20            0s 3ms/step - accuracy: 0.9064 - loss: 0.4163 - val_accuracy: 0.9187 - val_
Epoch 18/30
20/20            0s 3ms/step - accuracy: 0.9263 - loss: 0.4032 - val_accuracy: 0.9438 - val_
Epoch 19/30
20/20            0s 3ms/step - accuracy: 0.9220 - loss: 0.3972 - val_accuracy: 0.9563 - val_
Epoch 20/30
20/20            0s 3ms/step - accuracy: 0.9255 - loss: 0.3729 - val_accuracy: 0.9500 - val_
Epoch 21/30
20/20            0s 3ms/step - accuracy: 0.9339 - loss: 0.3591 - val_accuracy: 0.9563 - val_
Epoch 22/30
20/20            0s 3ms/step - accuracy: 0.9467 - loss: 0.3562 - val_accuracy: 0.9563 - val_
Epoch 23/30
20/20            0s 3ms/step - accuracy: 0.9376 - loss: 0.3329 - val_accuracy: 0.9688 - val_
Epoch 24/30
20/20            0s 3ms/step - accuracy: 0.9625 - loss: 0.3217 - val_accuracy: 0.9688 - val_
Epoch 25/30
20/20            0s 3ms/step - accuracy: 0.9438 - loss: 0.3130 - val_accuracy: 0.9688 - val_
Epoch 26/30
20/20            0s 3ms/step - accuracy: 0.9622 - loss: 0.2931 - val_accuracy: 0.9688 - val_
Epoch 27/30
20/20            0s 3ms/step - accuracy: 0.9602 - loss: 0.2930 - val_accuracy: 0.9812 - val_
Epoch 28/30
20/20            0s 3ms/step - accuracy: 0.9714 - loss: 0.2693 - val_accuracy: 0.9750 - val_
Epoch 29/30
20/20            0s 3ms/step - accuracy: 0.9569 - loss: 0.2767 - val_accuracy: 0.9750 - val_
Epoch 30/30
20/20            0s 4ms/step - accuracy: 0.9840 - loss: 0.2482 - val_accuracy: 0.9875 - val_
7/7            0s 3ms/step - accuracy: 0.9761 - loss: 0.2498
7/7            0s 3ms/step - accuracy: 0.9987 - loss: 0.1817
Pérdida en el conjunto de prueba (32 neuronas): 0.24888759851455688
Precisión en el conjunto de prueba (32 neuronas): 0.9800000190734863
```

```python
# E3
# Construir el modelo con función de activación tanh
model_tanh = Sequential([
    Dense(64, activation='tanh', input_shape=(2,)),
    Dense(1, activation='sigmoid')
])

# Compilar el modelo
model_tanh.compile(optimizer=Adam(), loss='binary_crossentropy', metrics=['accuracy'])
```

```python
# Entrenar el modelo
history_tanh = model_tanh.fit(x_train, y_train, epochs=30, batch_size=32, validation_split=0

# Evaluar el modelo
loss_tanh, accuracy_tanh = model_tanh.evaluate(x_test, y_test)
print(f'Pérdida en el conjunto de prueba (tanh): {loss_tanh}')
print(f'Precisión en el conjunto de prueba (tanh): {accuracy_tanh}')

# Construir el modelo con función de activación sigmoid
model_sigmoid = Sequential([
    Dense(64, activation='sigmoid', input_shape=(2,)),
    Dense(1, activation='sigmoid')
])

# Compilar el modelo
model_sigmoid.compile(optimizer=Adam(), loss='binary_crossentropy', metrics=['accuracy'])

# Entrenar el modelo
history_sigmoid = model_sigmoid.fit(x_train, y_train, epochs=30, batch_size=32, validation_sp

# Evaluar el modelo
loss_sigmoid, accuracy_sigmoid = model_sigmoid.evaluate(x_test, y_test)
print(f'Pérdida en el conjunto de prueba (sigmoid): {loss_sigmoid}')
print(f'Precisión en el conjunto de prueba (sigmoid): {accuracy_sigmoid}')
```

```
Epoch 1/30
20/20            1s 8ms/step - accuracy: 0.5034 - loss: 0.6679 - val_accuracy: 0.6375 - val_
Epoch 2/30
20/20            0s 3ms/step - accuracy: 0.6160 - loss: 0.6253 - val_accuracy: 0.6562 - val_
Epoch 3/30
20/20            0s 3ms/step - accuracy: 0.6278 - loss: 0.6040 - val_accuracy: 0.7063 - val_
Epoch 4/30
20/20            0s 3ms/step - accuracy: 0.7454 - loss: 0.5627 - val_accuracy: 0.7688 - val_
Epoch 5/30
20/20            0s 3ms/step - accuracy: 0.7714 - loss: 0.5468 - val_accuracy: 0.8438 - val_
Epoch 6/30
20/20            0s 3ms/step - accuracy: 0.8094 - loss: 0.5192 - val_accuracy: 0.8562 - val_
Epoch 7/30
20/20            0s 3ms/step - accuracy: 0.8301 - loss: 0.4924 - val_accuracy: 0.8813 - val_
Epoch 8/30
20/20            0s 3ms/step - accuracy: 0.8862 - loss: 0.4591 - val_accuracy: 0.9125 - val_
Epoch 9/30
```

```
20/20           0s 3ms/step - accuracy: 0.9065 - loss: 0.4199 - val_accuracy: 0.9438 - val_
Epoch 10/30
20/20           0s 3ms/step - accuracy: 0.9249 - loss: 0.4019 - val_accuracy: 0.9563 - val_
Epoch 11/30
20/20           0s 3ms/step - accuracy: 0.9281 - loss: 0.3842 - val_accuracy: 0.9563 - val_
Epoch 12/30
20/20           0s 4ms/step - accuracy: 0.9351 - loss: 0.3544 - val_accuracy: 0.9688 - val_
Epoch 13/30
20/20           0s 3ms/step - accuracy: 0.9549 - loss: 0.3400 - val_accuracy: 0.9750 - val_
Epoch 14/30
20/20           0s 3ms/step - accuracy: 0.9432 - loss: 0.3130 - val_accuracy: 0.9750 - val_
Epoch 15/30
20/20           0s 3ms/step - accuracy: 0.9709 - loss: 0.2838 - val_accuracy: 0.9812 - val_
Epoch 16/30
20/20           0s 3ms/step - accuracy: 0.9627 - loss: 0.2873 - val_accuracy: 0.9875 - val_
Epoch 17/30
20/20           0s 3ms/step - accuracy: 0.9704 - loss: 0.2599 - val_accuracy: 0.9812 - val_
Epoch 18/30
20/20           0s 3ms/step - accuracy: 0.9599 - loss: 0.2510 - val_accuracy: 0.9875 - val_
Epoch 19/30
20/20           0s 3ms/step - accuracy: 0.9697 - loss: 0.2492 - val_accuracy: 0.9875 - val_
Epoch 20/30
20/20           0s 3ms/step - accuracy: 0.9671 - loss: 0.2299 - val_accuracy: 0.9875 - val_
Epoch 21/30
20/20           0s 3ms/step - accuracy: 0.9740 - loss: 0.2117 - val_accuracy: 0.9875 - val_
Epoch 22/30
20/20           0s 3ms/step - accuracy: 0.9818 - loss: 0.2086 - val_accuracy: 0.9875 - val_
Epoch 23/30
20/20           0s 3ms/step - accuracy: 0.9856 - loss: 0.2123 - val_accuracy: 0.9875 - val_
Epoch 24/30
20/20           0s 3ms/step - accuracy: 0.9747 - loss: 0.1977 - val_accuracy: 0.9812 - val_
Epoch 25/30
20/20           0s 3ms/step - accuracy: 0.9689 - loss: 0.2031 - val_accuracy: 0.9875 - val_
Epoch 26/30
20/20           0s 3ms/step - accuracy: 0.9836 - loss: 0.1939 - val_accuracy: 0.9875 - val_
Epoch 27/30
20/20           0s 3ms/step - accuracy: 0.9852 - loss: 0.1898 - val_accuracy: 0.9875 - val_
Epoch 28/30
20/20           0s 3ms/step - accuracy: 0.9830 - loss: 0.1860 - val_accuracy: 0.9875 - val_
Epoch 29/30
20/20           0s 3ms/step - accuracy: 0.9900 - loss: 0.1772 - val_accuracy: 0.9875 - val_
Epoch 30/30
20/20           0s 3ms/step - accuracy: 0.9812 - loss: 0.1575 - val_accuracy: 0.9812 - val_
```

```
7/7             0s 3ms/step - accuracy: 0.9761 - loss: 0.1593
Pérdida en el conjunto de prueba (tanh): 0.15928113460540771
Precisión en el conjunto de prueba (tanh): 0.9800000190734863
Epoch 1/30
20/20             1s 8ms/step - accuracy: 0.5242 - loss: 0.6853 - val_accuracy: 0.8750 - val_
Epoch 2/30
20/20             0s 3ms/step - accuracy: 0.9120 - loss: 0.6742 - val_accuracy: 0.9750 - val_
Epoch 3/30
20/20             0s 3ms/step - accuracy: 0.8295 - loss: 0.6653 - val_accuracy: 0.9625 - val_
Epoch 4/30
20/20             0s 3ms/step - accuracy: 0.9249 - loss: 0.6573 - val_accuracy: 0.9625 - val_
Epoch 5/30
20/20             0s 3ms/step - accuracy: 0.9201 - loss: 0.6513 - val_accuracy: 0.9000 - val_
Epoch 6/30
20/20             0s 3ms/step - accuracy: 0.8347 - loss: 0.6384 - val_accuracy: 0.8813 - val_
Epoch 7/30
20/20             0s 3ms/step - accuracy: 0.9241 - loss: 0.6333 - val_accuracy: 1.0000 - val_
Epoch 8/30
20/20             0s 3ms/step - accuracy: 0.9335 - loss: 0.6226 - val_accuracy: 0.9500 - val_
Epoch 9/30
20/20             0s 3ms/step - accuracy: 0.9228 - loss: 0.6173 - val_accuracy: 0.9438 - val_
Epoch 10/30
20/20             0s 3ms/step - accuracy: 0.9232 - loss: 0.6055 - val_accuracy: 0.9750 - val_
Epoch 11/30
20/20             0s 3ms/step - accuracy: 0.9656 - loss: 0.5945 - val_accuracy: 0.9625 - val_
Epoch 12/30
20/20             0s 3ms/step - accuracy: 0.9653 - loss: 0.5905 - val_accuracy: 0.9875 - val_
Epoch 13/30
20/20             0s 3ms/step - accuracy: 0.9864 - loss: 0.5814 - val_accuracy: 0.9438 - val_
Epoch 14/30
20/20             0s 3ms/step - accuracy: 0.9368 - loss: 0.5760 - val_accuracy: 0.9750 - val_
Epoch 15/30
20/20             0s 4ms/step - accuracy: 0.9374 - loss: 0.5607 - val_accuracy: 0.9625 - val_
Epoch 16/30
20/20             0s 3ms/step - accuracy: 0.9685 - loss: 0.5580 - val_accuracy: 0.9875 - val_
Epoch 17/30
20/20             0s 3ms/step - accuracy: 0.9779 - loss: 0.5454 - val_accuracy: 0.9625 - val_
Epoch 18/30
20/20             0s 3ms/step - accuracy: 0.9381 - loss: 0.5337 - val_accuracy: 0.9750 - val_
Epoch 19/30
20/20             0s 3ms/step - accuracy: 0.9675 - loss: 0.5261 - val_accuracy: 0.9812 - val_
Epoch 20/30
20/20             0s 3ms/step - accuracy: 0.9546 - loss: 0.5281 - val_accuracy: 0.9937 - val_
```

```
Epoch 21/30
20/20             0s 3ms/step - accuracy: 0.9657 - loss: 0.5146 - val_accuracy: 0.9750 - val_
Epoch 22/30
20/20             0s 3ms/step - accuracy: 0.9459 - loss: 0.5110 - val_accuracy: 0.9750 - val_
Epoch 23/30
20/20             0s 3ms/step - accuracy: 0.9703 - loss: 0.4945 - val_accuracy: 0.9875 - val_
Epoch 24/30
20/20             0s 3ms/step - accuracy: 0.9483 - loss: 0.4804 - val_accuracy: 0.9750 - val_
Epoch 25/30
20/20             0s 3ms/step - accuracy: 0.9557 - loss: 0.4785 - val_accuracy: 0.9937 - val_
Epoch 26/30
20/20             0s 3ms/step - accuracy: 0.9879 - loss: 0.4678 - val_accuracy: 0.9875 - val_
Epoch 27/30
20/20             0s 3ms/step - accuracy: 0.9587 - loss: 0.4592 - val_accuracy: 0.9750 - val_
Epoch 28/30
20/20             0s 3ms/step - accuracy: 0.9552 - loss: 0.4435 - val_accuracy: 0.9812 - val_
Epoch 29/30
20/20             0s 3ms/step - accuracy: 0.9603 - loss: 0.4471 - val_accuracy: 0.9750 - val_
Epoch 30/30
20/20             0s 3ms/step - accuracy: 0.9805 - loss: 0.4267 - val_accuracy: 0.9875 - val_
7/7             0s 3ms/step - accuracy: 0.9761 - loss: 0.4204
Pérdida en el conjunto de prueba (sigmoid): 0.41852736473083496
Precisión en el conjunto de prueba (sigmoid): 0.9800000190734863
```

```python
# E4
# Generar datos sintéticos
np.random.seed(42)
x_1 = np.random.rand(1000)
x_2 = np.random.rand(1000)
x_data = np.column_stack((x_1, x_2))
y_data = x_1 + x_2

# Dividir el conjunto de datos en entrenamiento (70%) y prueba (30%)
x_train, x_test, y_train, y_test = train_test_split(x_data, y_data, test_size=0.3, random_sta

# Construir el modelo
model = Sequential([
    Dense(64, activation='relu', input_shape=(2,)),
    Dense(1)
])

# Compilar el modelo
```

```
model.compile(optimizer=Adam(), loss='mse', metrics=['mae'])

# Entrenar el modelo
history = model.fit(x_train, y_train, epochs=20, batch_size=32, validation_split=0.2)

# Evaluar el modelo
loss, mae = model.evaluate(x_test, y_test)
print(f"Pérdida en el conjunto de prueba: {loss}")
print(f"Error absoluto medio en el conjunto de prueba: {mae}")

# Nuevas muestras para predicción
x_sample = np.array([[0.3, 0.7], [0.5, 0.5]])

# Realizar predicciones
predicciones = model.predict(x_sample)

# Mostrar resultados
for i, sample in enumerate(x_sample):
    print(f"Entrada: {sample} → Probabilidad: {predicciones[i][0]:.2f} → Valor esperado: {np
```

```
Epoch 1/20
18/18             1s 9ms/step - loss: 1.0729 - mae: 0.9482 - val_loss: 0.6195 - val_mae: 0.69
Epoch 2/20
18/18             0s 3ms/step - loss: 0.5565 - mae: 0.6592 - val_loss: 0.2535 - val_mae: 0.42
Epoch 3/20
18/18             0s 3ms/step - loss: 0.2099 - mae: 0.3826 - val_loss: 0.0788 - val_mae: 0.22
Epoch 4/20
18/18             0s 3ms/step - loss: 0.0671 - mae: 0.2155 - val_loss: 0.0341 - val_mae: 0.15
Epoch 5/20
18/18             0s 3ms/step - loss: 0.0309 - mae: 0.1481 - val_loss: 0.0310 - val_mae: 0.15
Epoch 6/20
18/18             0s 3ms/step - loss: 0.0286 - mae: 0.1416 - val_loss: 0.0279 - val_mae: 0.14
Epoch 7/20
18/18             0s 3ms/step - loss: 0.0245 - mae: 0.1292 - val_loss: 0.0248 - val_mae: 0.13
Epoch 8/20
18/18             0s 3ms/step - loss: 0.0231 - mae: 0.1267 - val_loss: 0.0223 - val_mae: 0.12
Epoch 9/20
18/18             0s 3ms/step - loss: 0.0200 - mae: 0.1165 - val_loss: 0.0199 - val_mae: 0.11
Epoch 10/20
18/18             0s 4ms/step - loss: 0.0168 - mae: 0.1078 - val_loss: 0.0177 - val_mae: 0.11
Epoch 11/20
18/18             0s 3ms/step - loss: 0.0162 - mae: 0.1045 - val_loss: 0.0157 - val_mae: 0.10
```

```
Epoch 12/20
18/18             0s 3ms/step - loss: 0.0133 - mae: 0.0958 - val_loss: 0.0138 - val_mae: 0.09
Epoch 13/20
18/18             0s 3ms/step - loss: 0.0121 - mae: 0.0888 - val_loss: 0.0120 - val_mae: 0.09
Epoch 14/20
18/18             0s 3ms/step - loss: 0.0106 - mae: 0.0841 - val_loss: 0.0105 - val_mae: 0.08
Epoch 15/20
18/18             0s 3ms/step - loss: 0.0092 - mae: 0.0783 - val_loss: 0.0089 - val_mae: 0.07
Epoch 16/20
18/18             0s 3ms/step - loss: 0.0076 - mae: 0.0726 - val_loss: 0.0075 - val_mae: 0.07
Epoch 17/20
18/18             0s 3ms/step - loss: 0.0064 - mae: 0.0660 - val_loss: 0.0063 - val_mae: 0.06
Epoch 18/20
18/18             0s 3ms/step - loss: 0.0056 - mae: 0.0624 - val_loss: 0.0051 - val_mae: 0.05
Epoch 19/20
18/18             0s 3ms/step - loss: 0.0043 - mae: 0.0539 - val_loss: 0.0041 - val_mae: 0.05
Epoch 20/20
18/18             0s 3ms/step - loss: 0.0034 - mae: 0.0483 - val_loss: 0.0033 - val_mae: 0.04
10/10             0s 2ms/step - loss: 0.0030 - mae: 0.0451
Pérdida en el conjunto de prueba: 0.003026240970939398
Error absoluto medio en el conjunto de prueba: 0.04461654648184776
1/1             0s 36ms/step
Entrada: [0.3 0.7] → Probabilidad: 1.01 → Valor esperado: 1.00
Entrada: [0.5 0.5] → Probabilidad: 1.04 → Valor esperado: 1.00
```

```python
# E5
# Generar datos sintéticos más pequeños
np.random.seed(42)
x_1_small = np.random.rand(500)
x_2_small = np.random.rand(500)
x_data_small = np.column_stack((x_1_small, x_2_small))
y_data_small = x_1_small + x_2_small

# Dividir el conjunto de datos en entrenamiento (70%) y prueba (30%)
x_train_small, x_test_small, y_train_small, y_test_small = train_test_split(x_data_small, y_

# Construir y entrenar el modelo
model_small = Sequential([
    Dense(64, activation='relu', input_shape=(2,)),
    Dense(1)
])
model_small.compile(optimizer=Adam(), loss='mse', metrics=['mae'])
```

```python
history_small = model_small.fit(x_train_small, y_train_small, epochs=20, batch_size=32, valic

# Evaluar el modelo
loss_small, mae_small = model_small.evaluate(x_test_small, y_test_small)
print(f"Pérdida en el conjunto de prueba (500 muestras): {loss_small}")
print(f"Error absoluto medio en el conjunto de prueba (500 muestras): {mae_small}")
```

```
Epoch 1/20
9/9             1s 18ms/step - loss: 1.3704 - mae: 1.0604 - val_loss: 1.0665 - val_mae: 0.965
Epoch 2/20
9/9             0s 6ms/step - loss: 1.1502 - mae: 0.9712 - val_loss: 0.8386 - val_mae: 0.8478
Epoch 3/20
9/9             0s 6ms/step - loss: 0.8366 - mae: 0.8068 - val_loss: 0.6364 - val_mae: 0.7278
Epoch 4/20
9/9             0s 7ms/step - loss: 0.6299 - mae: 0.6830 - val_loss: 0.4564 - val_mae: 0.6054
Epoch 5/20
9/9             0s 6ms/step - loss: 0.4977 - mae: 0.5994 - val_loss: 0.3019 - val_mae: 0.4776
Epoch 6/20
9/9             0s 7ms/step - loss: 0.3005 - mae: 0.4586 - val_loss: 0.1845 - val_mae: 0.3594
Epoch 7/20
9/9             0s 6ms/step - loss: 0.1790 - mae: 0.3409 - val_loss: 0.1053 - val_mae: 0.2704
Epoch 8/20
9/9             0s 6ms/step - loss: 0.1104 - mae: 0.2721 - val_loss: 0.0615 - val_mae: 0.2117
Epoch 9/20
9/9             0s 6ms/step - loss: 0.0764 - mae: 0.2344 - val_loss: 0.0438 - val_mae: 0.1786
Epoch 10/20
9/9             0s 8ms/step - loss: 0.0612 - mae: 0.2139 - val_loss: 0.0392 - val_mae: 0.1656
Epoch 11/20
9/9             0s 6ms/step - loss: 0.0530 - mae: 0.1934 - val_loss: 0.0377 - val_mae: 0.1599
Epoch 12/20
9/9             0s 6ms/step - loss: 0.0550 - mae: 0.1997 - val_loss: 0.0354 - val_mae: 0.1546
Epoch 13/20
9/9             0s 6ms/step - loss: 0.0491 - mae: 0.1921 - val_loss: 0.0328 - val_mae: 0.1494
Epoch 14/20
9/9             0s 7ms/step - loss: 0.0466 - mae: 0.1841 - val_loss: 0.0305 - val_mae: 0.1443
Epoch 15/20
9/9             0s 6ms/step - loss: 0.0418 - mae: 0.1754 - val_loss: 0.0285 - val_mae: 0.1393
Epoch 16/20
9/9             0s 6ms/step - loss: 0.0397 - mae: 0.1699 - val_loss: 0.0267 - val_mae: 0.1344
Epoch 17/20
9/9             0s 7ms/step - loss: 0.0393 - mae: 0.1680 - val_loss: 0.0250 - val_mae: 0.1295
Epoch 18/20
```

```
9/9             0s 6ms/step - loss: 0.0363 - mae: 0.1619 - val_loss: 0.0235 - val_mae: 0.1247
Epoch 19/20
9/9             0s 7ms/step - loss: 0.0324 - mae: 0.1501 - val_loss: 0.0219 - val_mae: 0.1201
Epoch 20/20
9/9             0s 6ms/step - loss: 0.0293 - mae: 0.1442 - val_loss: 0.0205 - val_mae: 0.1156
5/5             0s 4ms/step - loss: 0.0292 - mae: 0.1400
Pérdida en el conjunto de prueba (500 muestras): 0.029062986373901367
Error absoluto medio en el conjunto de prueba (500 muestras): 0.13966381549835205
```

```python
# Generar datos sintéticos más grandes
np.random.seed(42)
x_1_large = np.random.rand(2000)
x_2_large = np.random.rand(2000)
x_data_large = np.column_stack((x_1_large, x_2_large))
y_data_large = x_1_large + x_2_large

# Dividir el conjunto de datos en entrenamiento (70%) y prueba (30%)
x_train_large, x_test_large, y_train_large, y_test_large = train_test_split(x_data_large, y_

# Construir y entrenar el modelo
model_large = Sequential([
    Dense(64, activation='relu', input_shape=(2,)),
    Dense(1)
])
model_large.compile(optimizer=Adam(), loss='mse', metrics=['mae'])
history_large = model_large.fit(x_train_large, y_train_large, epochs=20, batch_size=32, valid

# Evaluar el modelo
loss_large, mae_large = model_large.evaluate(x_test_large, y_test_large)
print(f'Pérdida en el conjunto de prueba (2000 muestras): {loss_large}')
print(f'Error absoluto medio en el conjunto de prueba (2000 muestras): {mae_large}')
```

```
Epoch 1/20
35/35             1s 5ms/step - loss: 0.7558 - mae: 0.7818 - val_loss: 0.2036 - val_mae: 0.38
Epoch 2/20
35/35             0s 2ms/step - loss: 0.1206 - mae: 0.2819 - val_loss: 0.0234 - val_mae: 0.12
Epoch 3/20
35/35             0s 2ms/step - loss: 0.0248 - mae: 0.1309 - val_loss: 0.0190 - val_mae: 0.11
Epoch 4/20
35/35             0s 2ms/step - loss: 0.0197 - mae: 0.1174 - val_loss: 0.0152 - val_mae: 0.10
Epoch 5/20
35/35             0s 2ms/step - loss: 0.0153 - mae: 0.1020 - val_loss: 0.0122 - val_mae: 0.09
```

```
Epoch 6/20
35/35             0s 2ms/step - loss: 0.0125 - mae: 0.0938 - val_loss: 0.0095 - val_mae: 0.08
Epoch 7/20
35/35             0s 2ms/step - loss: 0.0093 - mae: 0.0800 - val_loss: 0.0073 - val_mae: 0.07
Epoch 8/20
35/35             0s 2ms/step - loss: 0.0073 - mae: 0.0716 - val_loss: 0.0054 - val_mae: 0.06
Epoch 9/20
35/35             0s 2ms/step - loss: 0.0055 - mae: 0.0620 - val_loss: 0.0038 - val_mae: 0.05
Epoch 10/20
35/35             0s 3ms/step - loss: 0.0038 - mae: 0.0516 - val_loss: 0.0024 - val_mae: 0.04
Epoch 11/20
35/35             0s 2ms/step - loss: 0.0023 - mae: 0.0407 - val_loss: 0.0013 - val_mae: 0.02
Epoch 12/20
35/35             0s 2ms/step - loss: 0.0011 - mae: 0.0279 - val_loss: 4.5074e-04 - val_mae:
Epoch 13/20
35/35             0s 2ms/step - loss: 3.6084e-04 - mae: 0.0162 - val_loss: 1.2989e-04 - val_m
Epoch 14/20
35/35             0s 2ms/step - loss: 9.3975e-05 - mae: 0.0077 - val_loss: 5.6501e-05 - val_m
Epoch 15/20
35/35             0s 2ms/step - loss: 4.8877e-05 - mae: 0.0050 - val_loss: 4.0846e-05 - val_m
Epoch 16/20
35/35             0s 2ms/step - loss: 3.3123e-05 - mae: 0.0041 - val_loss: 3.6224e-05 - val_m
Epoch 17/20
35/35             0s 2ms/step - loss: 3.0958e-05 - mae: 0.0039 - val_loss: 3.3604e-05 - val_m
Epoch 18/20
35/35             0s 2ms/step - loss: 2.4671e-05 - mae: 0.0035 - val_loss: 3.1301e-05 - val_m
Epoch 19/20
35/35             0s 2ms/step - loss: 2.5439e-05 - mae: 0.0033 - val_loss: 3.0227e-05 - val_m
Epoch 20/20
35/35             0s 2ms/step - loss: 2.0535e-05 - mae: 0.0029 - val_loss: 2.8711e-05 - val_m
19/19             0s 1ms/step - loss: 3.3619e-05 - mae: 0.0037
Pérdida en el conjunto de prueba (2000 muestras): 3.290706445113756e-05
Error absoluto medio en el conjunto de prueba (2000 muestras): 0.0036516357213258743
```

```python
# E6
# Generar datos sintéticos con ruido
np.random.seed(42)
x_1_noise = np.random.rand(1000)
x_2_noise = np.random.rand(1000)
x_data_noise = np.column_stack((x_1_noise, x_2_noise))
y_data_noise = x_1_noise + x_2_noise + np.random.normal(0, 0.1, 1000)  # Añadir ruido gaussia
```

```python
# Dividir el conjunto de datos en entrenamiento (70%) y prueba (30%)
x_train_noise, x_test_noise, y_train_noise, y_test_noise = train_test_split(x_data_noise, y_

# Construir y entrenar el modelo
model_noise = Sequential([
    Dense(64, activation='relu', input_shape=(2,)),
    Dense(1)
])
model_noise.compile(optimizer=Adam(), loss='mse', metrics=['mae'])
history_noise = model_noise.fit(x_train_noise, y_train_noise, epochs=20, batch_size=32, valid

# Evaluar el modelo
loss_noise, mae_noise = model_noise.evaluate(x_test_noise, y_test_noise)
print(f'Pérdida en el conjunto de prueba (con ruido): {loss_noise}')
print(f'Error absoluto medio en el conjunto de prueba (con ruido): {mae_noise}')
```

```
Epoch 1/20
18/18             1s 9ms/step - loss: 1.0683 - mae: 0.9382 - val_loss: 0.6357 - val_mae: 0.70
Epoch 2/20
18/18             0s 3ms/step - loss: 0.5603 - mae: 0.6669 - val_loss: 0.2875 - val_mae: 0.45
Epoch 3/20
18/18             0s 3ms/step - loss: 0.2222 - mae: 0.4025 - val_loss: 0.1043 - val_mae: 0.26
Epoch 4/20
18/18             0s 3ms/step - loss: 0.0875 - mae: 0.2400 - val_loss: 0.0457 - val_mae: 0.17
Epoch 5/20
18/18             0s 3ms/step - loss: 0.0391 - mae: 0.1598 - val_loss: 0.0386 - val_mae: 0.15
Epoch 6/20
18/18             0s 3ms/step - loss: 0.0343 - mae: 0.1500 - val_loss: 0.0370 - val_mae: 0.15
Epoch 7/20
18/18             0s 3ms/step - loss: 0.0356 - mae: 0.1521 - val_loss: 0.0349 - val_mae: 0.15
Epoch 8/20
18/18             0s 4ms/step - loss: 0.0328 - mae: 0.1442 - val_loss: 0.0329 - val_mae: 0.14
Epoch 9/20
18/18             0s 3ms/step - loss: 0.0298 - mae: 0.1395 - val_loss: 0.0310 - val_mae: 0.14
Epoch 10/20
18/18             0s 3ms/step - loss: 0.0270 - mae: 0.1319 - val_loss: 0.0294 - val_mae: 0.13
Epoch 11/20
18/18             0s 3ms/step - loss: 0.0266 - mae: 0.1312 - val_loss: 0.0276 - val_mae: 0.13
Epoch 12/20
18/18             0s 6ms/step - loss: 0.0247 - mae: 0.1262 - val_loss: 0.0259 - val_mae: 0.12
Epoch 13/20
18/18             0s 3ms/step - loss: 0.0255 - mae: 0.1302 - val_loss: 0.0243 - val_mae: 0.12
```

```
Epoch 14/20
18/18            0s 3ms/step - loss: 0.0227 - mae: 0.1230 - val_loss: 0.0229 - val_mae: 0.12
Epoch 15/20
18/18            0s 4ms/step - loss: 0.0209 - mae: 0.1176 - val_loss: 0.0215 - val_mae: 0.11
Epoch 16/20
18/18            0s 3ms/step - loss: 0.0207 - mae: 0.1135 - val_loss: 0.0202 - val_mae: 0.11
Epoch 17/20
18/18            0s 3ms/step - loss: 0.0172 - mae: 0.1053 - val_loss: 0.0189 - val_mae: 0.11
Epoch 18/20
18/18            0s 3ms/step - loss: 0.0197 - mae: 0.1116 - val_loss: 0.0178 - val_mae: 0.10
Epoch 19/20
18/18            0s 3ms/step - loss: 0.0173 - mae: 0.1063 - val_loss: 0.0167 - val_mae: 0.10
Epoch 20/20
18/18            0s 4ms/step - loss: 0.0174 - mae: 0.1048 - val_loss: 0.0158 - val_mae: 0.10
10/10            0s 3ms/step - loss: 0.0141 - mae: 0.0946
Pérdida en el conjunto de prueba (con ruido): 0.01435171626508236
Error absoluto medio en el conjunto de prueba (con ruido): 0.09611277282238007
```