

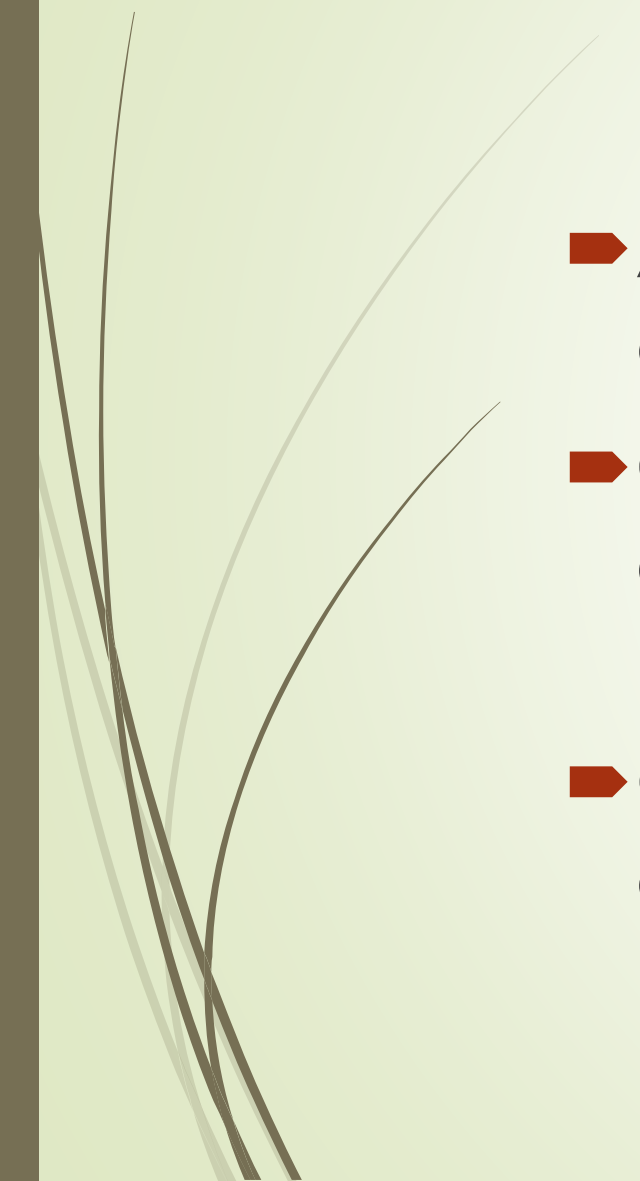


Proyecto 7 Bootcamp Data Science UDD

Sergio Briones Pérez



Objetivos

- Aplicar con éxito todos los conocimientos que has adquirido a lo largo del Bootcamp.
 - Consolidar las técnicas de limpieza, entrenamiento, graficación y ajuste a modelos de Machine Learning.
 - Generar una API que brinde predicciones como resultado a partir de datos enviados.
- 



Proyecto desarrollado

- Imágenes de rayos X de pecho para detectar neumonía:

<https://www.kaggle.com/datasets/paultimothymooney/chest-xray-pneumonia>



EXPLICACIÓN DEL DATA

El archivo Chest X-Ray Images (Pneumonia) es un conjunto de datos de imágenes de radiografías de tórax que se utilizan para entrenar y evaluar modelos de inteligencia artificial (IA) para la detección de neumonía. El conjunto de datos está organizado en tres carpetas:

train: Contiene imágenes de rayos X de tórax de pacientes normales y con neumonía.

test: Contiene imágenes de rayos X de tórax de pacientes normales y con neumonía.

val: Contiene imágenes de rayos X de tórax de pacientes normales y con neumonía.

Cada una de las tres carpetas está subdividida en carpetas NORMAL y PNEUMONIA.



Elección del modelo

Elegí el conjunto de datos **Chest X-Ray Images (Pneumonia)** para entrenar y evaluar mi modelo de IA para la detección de neumonía por las siguientes razones:

- **El conjunto de datos es grande y diverso.** Contiene más de 10.000 imágenes de rayos X de tórax de pacientes con neumonía y sin neumonía. Esto proporciona una gran cantidad de datos para entrenar el modelo y mejorar su precisión.
- **El conjunto de datos está bien etiquetado.** Cada imagen está etiquetada con una clase, ya sea "normal" o "neumonía". Esto permite al modelo aprender a identificar los patrones que distinguen entre las dos clases.
- **El conjunto de datos está bien equilibrado.** El número de imágenes de rayos X de tórax de pacientes con neumonía y sin neumonía es aproximadamente el mismo. Esto ayuda a garantizar que el modelo no se sesgue hacia una de las dos clases.



IMPORTANCIA DEL ANÁLISIS

El análisis de este conjunto de datos es importante por las siguientes razones:

- **Para mejorar la precisión de los modelos de IA para la detección de neumonía.** Los modelos de IA se benefician de un conjunto de datos de entrenamiento grande y representativo. El archivo Chest X-Ray Images (Pneumonia) es un conjunto de datos grande y diverso que incluye imágenes de rayos X de pacientes de diferentes edades, géneros y etnias.
- **Para desarrollar nuevos métodos de detección de neumonía.** Los modelos de IA pueden utilizarse para detectar neumonía en imágenes de rayos X de tórax de forma más precisa y eficiente que los métodos tradicionales. El análisis de este conjunto de datos puede ayudar a los investigadores a desarrollar nuevos métodos de detección de neumonía que sean más precisos y eficaces.



¿QUÉ ES LA NEUMONÍA?

La **neumonía es una infección de los pulmones** que causa inflamación y acumulación de líquido o pus en los sacos de aire (alvéolos). Los alvéolos son las estructuras en los pulmones donde se produce el intercambio de oxígeno y dióxido de carbono. Cuando los alvéolos están inflamados o llenos de líquido o pus, es difícil que el oxígeno llegue a la sangre.

La neumonía **puede ser causada por bacterias, virus u hongos**. Las bacterias son la causa más común de neumonía, seguidas de los virus. Los hongos son una causa menos común, pero pueden ser más graves, especialmente en personas con sistemas inmunitarios debilitados.



¿CUÁLES SON LOS SÍNTOMAS DE LA NEUMONÍA?

Los síntomas de la neumonía incluyen:

- Tos con o sin moco
- Fiebre
- Escalofríos
- Dificultad para respirar
- Dolor en el pecho
- Cansancio

La neumonía puede ser una enfermedad grave, especialmente en niños pequeños, adultos mayores y personas con sistemas inmunitarios debilitados. Si cree que puede tener neumonía, es importante que consulte a un médico lo antes posible.



TRATAMIENTO DE LA NEUMONÍA

El tratamiento de la neumonía depende de la causa. La neumonía bacteriana suele tratarse con antibióticos. La neumonía viral suele curarse por sí sola, pero puede tratarse con medicamentos antivirales para aliviar los síntomas. La neumonía por hongos puede ser más grave y puede requerir medicamentos antifúngicos.

Hay algunas cosas que puede hacer para reducir su riesgo de contraer neumonía, como:

- Vacunarse contra la influenza y la neumonía.
- Lavarse las manos con frecuencia.
- Evitar el contacto cercano con personas que estén enfermas.
- Si fuma, dejar de fumar es una de las cosas más importantes que puede hacer para proteger sus pulmones.



¿CÓMO ES EL PROCESO DE ANÁLISIS DE LAS IMÁGENES?

El proceso de análisis de las imágenes de radiografías de tórax en este conjunto de datos es el siguiente:

- Control de calidad: Todas las radiografías de tórax se examinan inicialmente para eliminar aquellas que son de baja calidad o ilegibles.
- Clasificación por médicos expertos: Dos médicos expertos califican los diagnósticos de las imágenes.
- Revisión por un tercer experto: Un tercer experto revisa el conjunto de evaluación para tener en cuenta posibles errores de calificación.

Este proceso de análisis garantiza que las imágenes de rayos X en el conjunto de datos sean de alta calidad y que los diagnósticos sean precisos.

CARGAR DATOS DEL MODELO

El primer paso es **IMPORTAR LIBRERÍAS**. Las librerías son colecciones de código que se pueden utilizar para realizar tareas específicas. Las siguientes librerías nos permitirán trabajar con el modelo que necesitamos entrenar.

```
import pandas as pd
import numpy as np
from mpl_toolkits.mplot3d import Axes3D
from sklearn.preprocessing import
StandardScaler
import matplotlib.pyplot as plt
import os
import requests
```

```
import seaborn as sns
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.models import
Sequential
from tensorflow.keras.layers import
Conv2D, MaxPooling2D, Dropout, Flatten,
Dense
from tensorflow.keras import optimizers
```



CARGAR DATOS DEL MODELO

RUTA ACCESO CARPETA: a través del siguiente código, asignaremos la ruta al directorio de la variable path. Esto nos permitirá acceder a los archivos de imágenes de rayos X de tórax que se encuentran en un ambiente local.

```
path = 'D:\Data Science\Modulo 7\Proyecto 7\chest_xray'
```



ANÁLISIS EXPLORATORIO

IMPRIMIR CONTENIDO CARPETA: Ahora imprimiremos una lista de los archivos y directorios que se encuentran en el directorio especificado por la variable `path`. La función `os.listdir()` devuelve una lista de los nombres de los archivos y directorios que se encuentran en el directorio especificado por el parámetro `path`.

```
print(os.listdir(path))
```

Muestra lo siguiente:

```
['chest_xray', 'test', 'train', 'val', '__MACOSX']
```



ANÁLISIS EXPLORATORIO

RUTA ACCESO CONJUNTO DE DATOS: Los siguientes códigos no permite asignar la ruta de acceso al directorio de las variables `train_path`, `val_path` y `test_path`. Es importante para el posterior acceso a las carpetas.

```
train_path = 'D:\Data Science\Modulo 7\Proyecto 7\chest_xray/train'  
val_path = 'D:\Data Science\Modulo 7\Proyecto 7\chest_xray/val'  
test_path = 'D:\Data Science\Modulo 7\Proyecto 7\chest_xray/test'
```




ANÁLISIS EXPLORATORIO

RUTA ACCESO CONJUNTO DE DATOS: Los siguientes códigos no permite asignar la ruta de acceso al directorio de las variables `train_path`, `val_path` y `test_path`. Es importante para el posterior acceso a las carpetas.

```
train_path = 'D:\Data Science\Modulo 7\Proyecto 7\chest_xray/train'  
val_path = 'D:\Data Science\Modulo 7\Proyecto 7\chest_xray/val'  
test_path = 'D:\Data Science\Modulo 7\Proyecto 7\chest_xray/test'
```

ANÁLISIS EXPLORATORIO

CONTAR NÚMERO DE IMÁGENES: Estos resultados se pueden utilizar para evaluar la distribución de las clases en los conjuntos de datos. Posteriormente, se hace un print para conocer el número de imágenes en train normal, train neumonía, val normal, val neumonía, test normal y test neumonía.

```
# Contamos el número de imágenes de cada clase en cada conjunto de datos
num_train_normal = len(os.listdir(os.path.join(train_path, 'NORMAL')))
num_train_pneumonia = len(os.listdir(os.path.join(train_path,
'PNEUMONIA')))
num_val_normal = len(os.listdir(os.path.join(val_path, 'NORMAL')))
num_val_pneumonia = len(os.listdir(os.path.join(val_path, 'PNEUMONIA')))
num_test_normal = len(os.listdir(os.path.join(test_path, 'NORMAL')))
num_test_pneumonia = len(os.listdir(os.path.join(test_path, 'PNEUMONIA')))
```

ANÁLISIS EXPLORATORIO

CONTAR NÚMERO DE IMÁGENES:

```
print('Número de imágenes train normal:', num_train_normal)
print('Número de imágenes train pneumonia:', num_train_pneumonia)
print('Número de imágenes val normal:', num_val_normal)
print('Número de imágenes val pneumonia:', num_val_pneumonia)
print('Número de imágenes test_normal:', num_test_normal)
print('Número de imágenes test pneumonia:', num_test_pneumonia)
```

```
Número de imágenes train normal: 1341
Número de imágenes train pneumonia: 3875
Número de imágenes val normal: 8 Número de imágenes val pneumonia: 8
Número de imágenes test_normal: 234
Número de imágenes test pneumonia: 390
```

ANÁLISIS EXPLORATORIO

VISUALIZAR EL NOMBRE DE LAS CARPETAS: Al ocupar `os.listdir` nos devuelve una lista de los nombres de los archivos y directorios que se encuentran en el directorio especificado por el parámetro `path`. La función `print` imprime el contenido de la lista en la consola.

```
print(os.listdir('D:\Data Science\Modulo 7\Proyecto 7\chest_xray'))
```

```
['chest_xray', 'test', 'train', 'val', '__MACOSX']
```

```
print(os.listdir('D:\Data Science\Modulo 7\Proyecto 7\chest_xray/train'))
```

```
['NORMAL', 'PNEUMONIA']
```

Con cada carpeta haremos lo mismo.



Visualizar Imágenes

VISUALIZAR LEER IMÁGENES: para leer una imagen de cada conjunto de datos para fines de visualización o preprocesamiento. El código se compone de lo siguiente:

`plt.imread(path)`: Esta función lee una imagen a partir de una ruta de archivo y la devuelve como un array de NumPy.

`os.path.join(path1, path2, ...)`: Esta función une varias rutas de archivo en una sola ruta.

ANÁLISIS EXPLORATORIO

Leemos una imagen de cada conjunto de datos

```
img_train_normal = plt.imread(os.path.join(train_path, 'NORMAL', 'IM-0249-0001.jpeg'))
img_train_pneumonia = plt.imread(os.path.join(train_path, 'PNEUMONIA', 'person101_virus_187.jpeg'))
img_val_normal = plt.imread(os.path.join(val_path, 'NORMAL', 'NORMAL2-IM-1436-0001.jpeg'))
img_val_pneumonia = plt.imread(os.path.join(val_path, 'PNEUMONIA', 'person1954_bacteria_4886.jpeg'))
img_test_normal = plt.imread(os.path.join(test_path, 'NORMAL', 'IM-0027-0001.jpeg'))
img_test_pneumonia = plt.imread(os.path.join(test_path, 'PNEUMONIA', 'person113_bacteria_541.jpeg'))
```

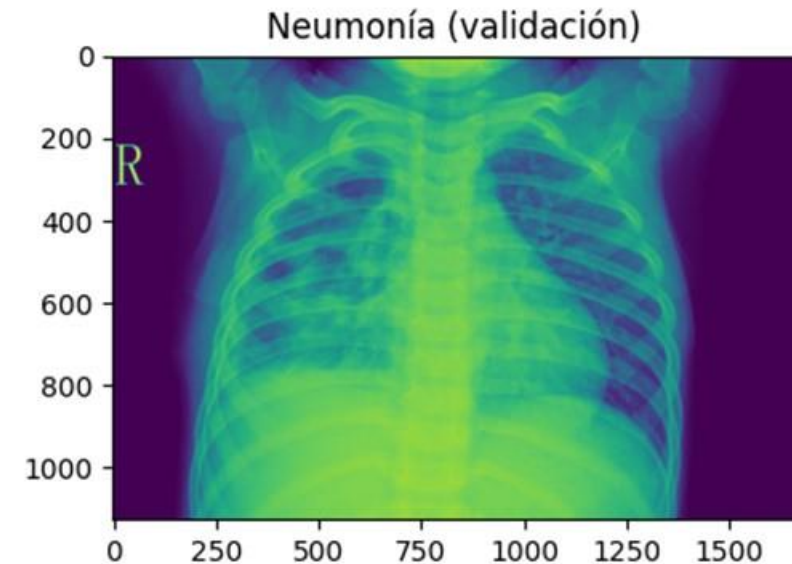
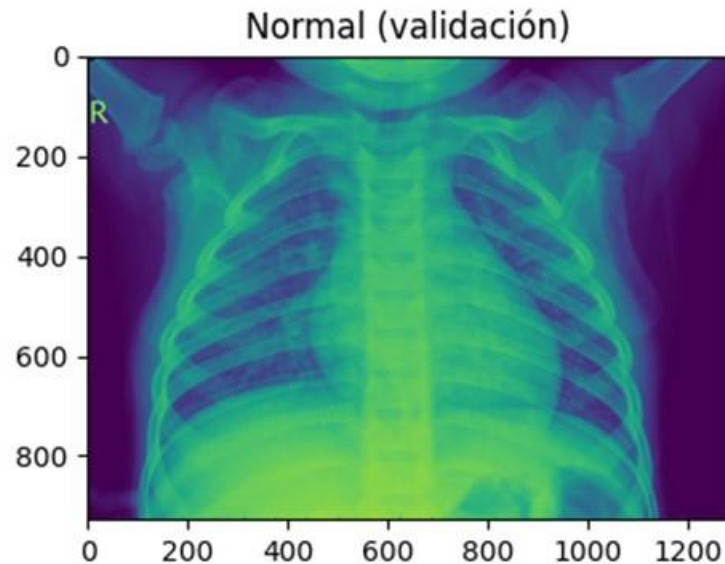
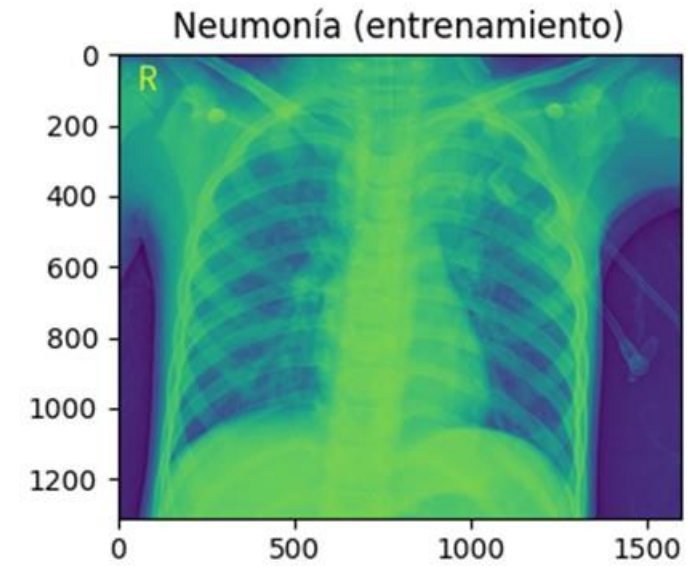
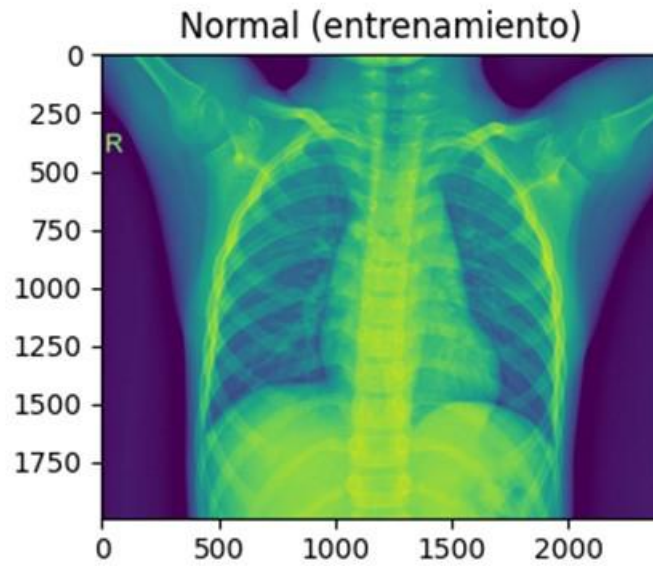

ANÁLISIS EXPLORATORIO

Mostramos las imágenes

```
plt.figure(figsize=(12, 6))
plt.subplot(2, 2, 1)
plt.imshow(img_train_normal)
plt.title('Normal (entrenamiento)')
plt.subplot(2, 2, 2)
plt.imshow(img_train_pneumonia)
plt.title('Neumonía (entrenamiento)')
plt.subplot(2, 2, 3)
plt.imshow(img_val_normal)
plt.title('Normal (validación)')
plt.subplot(2, 2, 4)
plt.imshow(img_val_pneumonia)
plt.title('Neumonía (validación)')
plt.tight_layout()
plt.show()
```

ANÁLISIS EXPLORATORIO

Resultado:



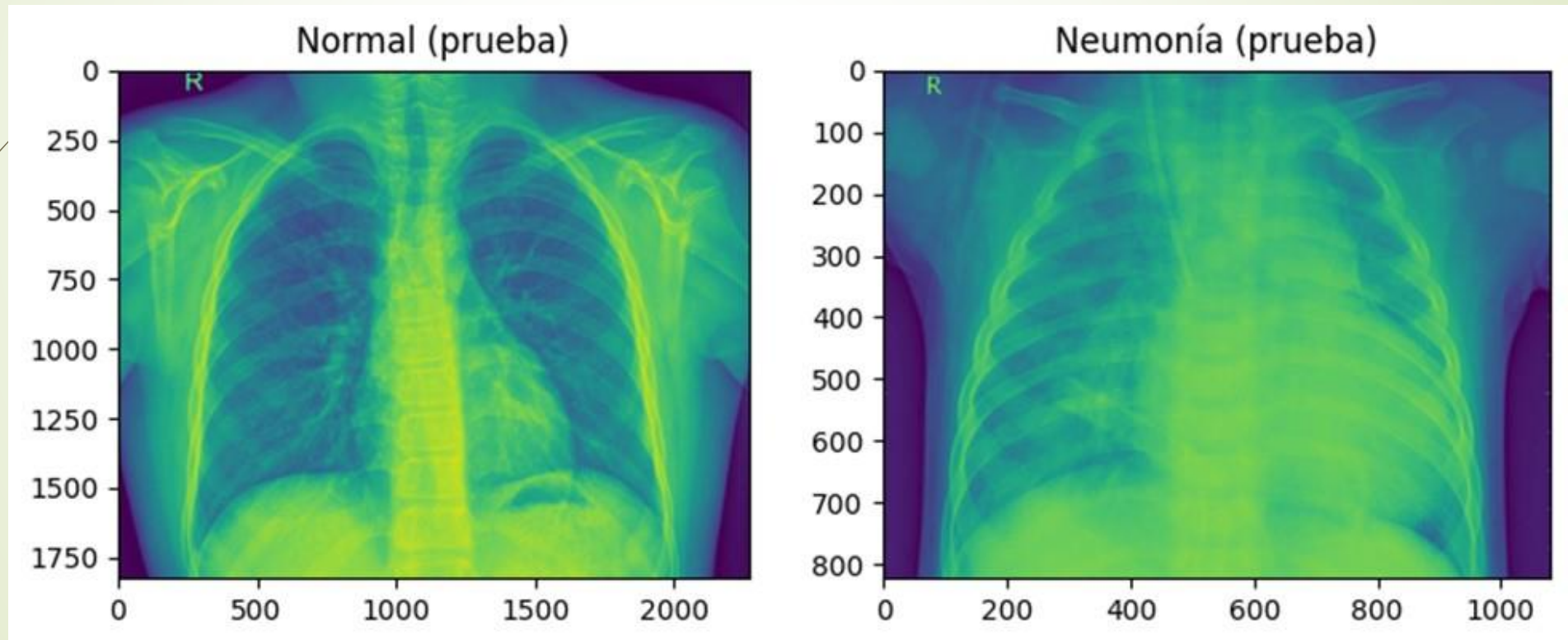
ANÁLISIS EXPLORATORIO

Mostrar las imágenes de prueba

```
plt.figure(figsize=(12, 6))  
plt.subplot(2, 3, 1)  
plt.imshow(img_test_normal)  
plt.title('Normal (prueba)')  
plt.subplot(2, 3, 2)  
plt.imshow(img_test_pneumonia)  
plt.title('Neumonía (prueba)')  
plt.tight_layout()  
plt.show()
```

ANÁLISIS EXPLORATORIO

Resultado:



ANÁLISIS EXPLORATORIO

CONTAR NÚMERO DE IMÁGENES: Estos resultados se pueden utilizar para evaluar la distribución de las clases en los conjuntos de datos. Posteriormente, se hace un print para conocer el número de imágenes en train normal, train neumonía, val normal, val neumonía, test normal y test neumonía.

```
# Contamos el número de imágenes de cada clase en cada conjunto de datos
num_train_normal = len(os.listdir(os.path.join(train_path, 'NORMAL')))
num_train_pneumonia = len(os.listdir(os.path.join(train_path,
'PNEUMONIA')))
num_val_normal = len(os.listdir(os.path.join(val_path, 'NORMAL')))
num_val_pneumonia = len(os.listdir(os.path.join(val_path, 'PNEUMONIA')))
num_test_normal = len(os.listdir(os.path.join(test_path, 'NORMAL')))
num_test_pneumonia = len(os.listdir(os.path.join(test_path, 'PNEUMONIA')))
```



CREACIÓN DEL MODELO

Se crearon tres modelos, voy a explicar en general los pasos, ya que cada modelo se entrenó con Sequential de Keras, pero se cambiaron los parámetros en cada modelo.

Lo primero es asignar las rutas a los conjuntos de datos de entrenamiento, validación y prueba a las variables `train_path`, `val_path` y `test_path`, respectivamente.

Es decir, la variable `train_path` contiene la ruta al directorio de imágenes de rayos X de tórax de entrenamiento. La variable `val_path` contiene la ruta al directorio de imágenes de rayos X de tórax de validación. La variable `test_path` contiene la ruta al directorio de imágenes de rayos X de tórax de prueba.



CREACIÓN DEL MODELO

Códigos: Esto permite al modelo acceder a los datos de entrenamiento, validación y prueba. Sin estas rutas, el modelo no podría leer las imágenes y realizar las tareas de aprendizaje automático.

```
# Obtenemos la ruta de acceso a los conjuntos de datos
train_path = 'D:\Data Science\Modulo 7\Proyecto
7\chest_xray/train'
val_path = 'D:\Data Science\Modulo 7\Proyecto
7\chest_xray/val'
test_path = 'D:\Data Science\Modulo 7\Proyecto
7\chest_xray/test'
```



CREACIÓN DEL MODELO

Lo primero que hacemos es **importar la clase ImageDataGenerator de TensorFlow Keras**. Esta clase permite generar lotes de imágenes de manera eficiente a partir de directorios con subcarpetas para cada clase.

Segundo, **definimos los parámetros de procesamiento**. En este caso, `batch_size`, que establece el número de imágenes que se procesarán juntas en cada lote. En este caso, se procesarán 32 imágenes por lote y `rescale`, que normaliza la intensidad de píxel de las imágenes dividiendo cada valor por 255.



CREACIÓN DEL MODELO

Por último, configuración de generadores con `flow_from_directory`:

flow_from_directory: Este método del `ImageDataGenerator` se encarga de leer las imágenes de los directorios especificados, aplicar el preprocesamiento y generar lotes de imágenes.

train_path, val_path, test_path: Especifican la ruta a los directorios de entrenamiento, validación y prueba, respectivamente.

target_size: Define el tamaño de redimensionamiento de las imágenes a (150, 150) píxeles.

class_mode: Indica cómo se codifican las etiquetas de las clases. En este caso, se asume que hay dos clases y se asignan como 0 y 1. Si hubiera más clases, se usaría "categorical".



CREACIÓN DEL MODELO

¿Cuáles son los beneficios de usar generadores?

Eficiencia de memoria: Los generadores cargan las imágenes en lotes, evitando cargar todo el conjunto de datos a la vez, lo que ahorra memoria.

Aumenta la precisión: La mezcla de imágenes durante el entrenamiento (a través de generadores) ayuda a prevenir el sobreajuste y mejora la precisión del modelo.

Facilidad de uso: ImageDataGenerator permite la configuración sencilla de preprocesamiento y generación de lotes.

CREACIÓN DEL MODELO

```
from tensorflow.keras.preprocessing.image import ImageDataGenerator

# Definir generadores de datos para entrenamiento, validación y prueba
batch_size = 32
train_datagen = ImageDataGenerator(rescale=1./255)
val_datagen = ImageDataGenerator(rescale=1./255)
test_datagen = ImageDataGenerator(rescale=1./255)
train_generator = train_datagen.flow_from_directory(train_path,
target_size=(150, 150), batch_size=batch_size, class_mode='binary' # 0
'categorical' si hay más de dos clases)
val_generator = val_datagen.flow_from_directory(val_path,
target_size=(150, 150), batch_size=batch_size, class_mode='binary')

test_generator = test_datagen.flow_from_directory(test_path,
target_size=(150, 150), batch_size=batch_size, class_mode='binary')
```




ENTRENAR LOS DATOS

El entrenamiento lo entrenaremos en TensorFlow Keras. Lo primero es realizar las importaciones:

`from tensorflow.keras.models import Sequential`: Importa la clase Sequential para construir el modelo como una secuencia de capas.

`from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense`: Importa las capas que usarás en el modelo.

`from tensorflow.keras.optimizers import Adam`: Importa el optimizador Adam para entrenar el modelo. 0 y 1).

ENTRENAR LOS DATOS

Luego definimos el Modelo: `model2 = Sequential`: Crea una instancia del modelo `model2` usando la clase `Sequential` y le pasa una lista de capas en el orden en que se deben apilar.

Explicación de las capas:**

`Conv2D(32, (3, 3), activation='relu', input_shape=(150, 150, 3))`. Esta capa convolucional tiene 64 filtros de tamaño 3x3 y aplica la función de activación `relu`. La entrada de la red tiene forma (150, 150, 3), que indica el ancho, alto y número de canales de la imagen (RGB).

`MaxPooling2D((2, 2))`: Aplica pooling máximo con un kernel de 2x2, reduciendo la resolución de la imagen a la mitad en cada dimensión.

`Conv2D(64, (3, 3), activation='relu')`. similar a la primera capa convolucional, pero con 128 filtros.



ENTRENAR LOS DATOS

`MaxPooling2D((2, 2))`: similar a la capa anterior.

`Flatten()`: Aplana la salida de la capa convolucional a un vector de 1 dimensión.

`Dense(64, activation='relu')`: Capa densa con 128 neuronas y activación relu.

`Dense(1, activation='sigmoid')`: Capa de salida con 1 neurona y activación sigmoid. La salida de esta neurona será la predicción del modelo (entre 0 y 1).

ENTRENAR LOS DATOS

Códigos:

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense
from tensorflow.keras.optimizers import Adam

model1 = Sequential([
    Conv2D(32, (3, 3), activation='relu', input_shape=(150, 150, 3)),
    MaxPooling2D((2, 2)),
    Conv2D(64, (3, 3), activation='relu'),
    MaxPooling2D((2, 2)),
    Flatten(),
    Dense(64, activation='relu'),
    Dense(1, activation='sigmoid')
])

model1.compile(optimizer='adam',
               loss='binary_crossentropy',
               metrics=['accuracy'])
```



ENTRENAMIENTO

Definimos el número de épocas (epochs), en este caso son 10.

Método `fit()` del modelo, que entrena el modelo utilizando los generadores de entrenamiento y validación.

train_generator: El generador de entrenamiento que se utilizará para entrenar el modelo.

steps_per_epoch: El número de pasos por época. En este caso, cada época consistirá en recorrer todo el conjunto de datos de entrenamiento una vez.

epochs: El número de épocas.

validation_data: El generador de validación que se utilizará para evaluar el modelo durante el entrenamiento.

validation_steps: El número de pasos por época de validación. En este caso, cada época de validación consistirá en recorrer todo el conjunto de datos de validación una vez.



ENTRENAMIENTO

Código:

```
# Entrenamiento del modelo
epochs = 10

history1 = model1.fit(
    train_generator,
    steps_per_epoch=len(train_generator),
    epochs=epochs,
    validation_data=val_generator,
    validation_steps=len(val_generator)
)
```


ENTRENAMIENTO

Salida:

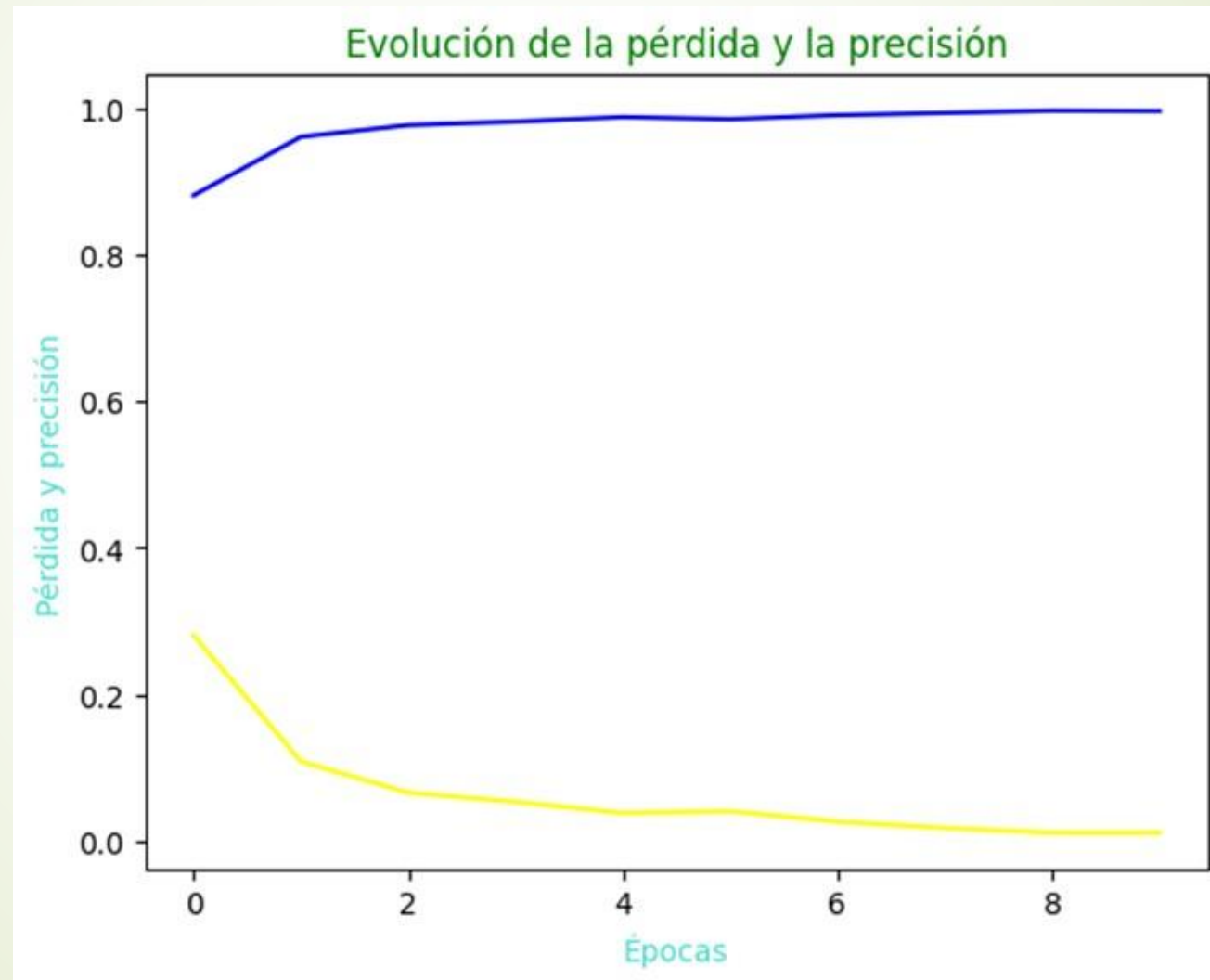
```
Epoch 1/10 163/163 [=====] - 50s 298ms/step - loss: 0.2805 -  
accuracy: 0.8806 - val_loss: 0.3229 - val_accuracy: 0.8750  
Epoch 2/10 163/163 [=====] - 50s 305ms/step - loss: 0.1084 -  
accuracy: 0.9603 - val_loss: 0.1784 - val_accuracy: 0.9375  
Epoch 3/10 163/163 [=====] - 48s 294ms/step - loss: 0.0661 -  
accuracy: 0.9762 - val_loss: 0.1167 - val_accuracy: 0.9375  
Epoch 4/10 163/163 [=====] - 48s 295ms/step - loss: 0.0535 -  
accuracy: 0.9810 - val_loss: 0.0617 - val_accuracy: 1.0000  
Epoch 5/10 163/163 [=====] - 50s 305ms/step - loss: 0.0385 -  
accuracy: 0.9872 - val_loss: 0.0695 - val_accuracy: 0.9375  
Epoch 6/10 163/163 [=====] - 48s 294ms/step - loss: 0.0405 -  
accuracy: 0.9843 - val_loss: 0.5403 - val_accuracy: 0.8125  
Epoch 7/10 163/163 [=====] - 48s 293ms/step - loss: 0.0267 -  
accuracy: 0.9898 - val_loss: 0.6191 - val_accuracy: 0.8125  
Epoch 8/10 163/163 [=====] - 48s 296ms/step - loss: 0.0178 -  
accuracy: 0.9929 - val_loss: 0.0854 - val_accuracy: 0.9375  
Epoch 9/10 163/163 [=====] - 48s 295ms/step - loss: 0.0117 -  
accuracy: 0.9960 - val_loss: 0.2399 - val_accuracy: 0.9375  
Epoch 10/10 163/163 [=====] - 49s 299ms/step - loss: 0.0117 -  
accuracy: 0.9954 - val_loss: 0.4188 - val_accuracy: 0.9375
```

GRÁFICAS DEL MODELO

El siguiente código sirve para graficar la evolución de la pérdida y la precisión del modelo de clasificación de rayos X de tórax durante el entrenamiento. La pérdida es una medida de la distancia entre las predicciones del modelo y las etiquetas reales. La precisión es una medida de la proporción de predicciones correctas realizadas por el modelo.

```
# Graficar la evolución de la pérdida y la precisión
plt.plot(history1.history["loss"], color='yellow')
plt.plot(history1.history["accuracy"], color='blue')
plt.title("Evolución de la pérdida y la precisión", color='green')
plt.xlabel("Épocas", color='turquoise')
plt.ylabel("Pérdida y precisión", color='turquoise')
plt.show()
```

GRÁFICAS DEL MODELO

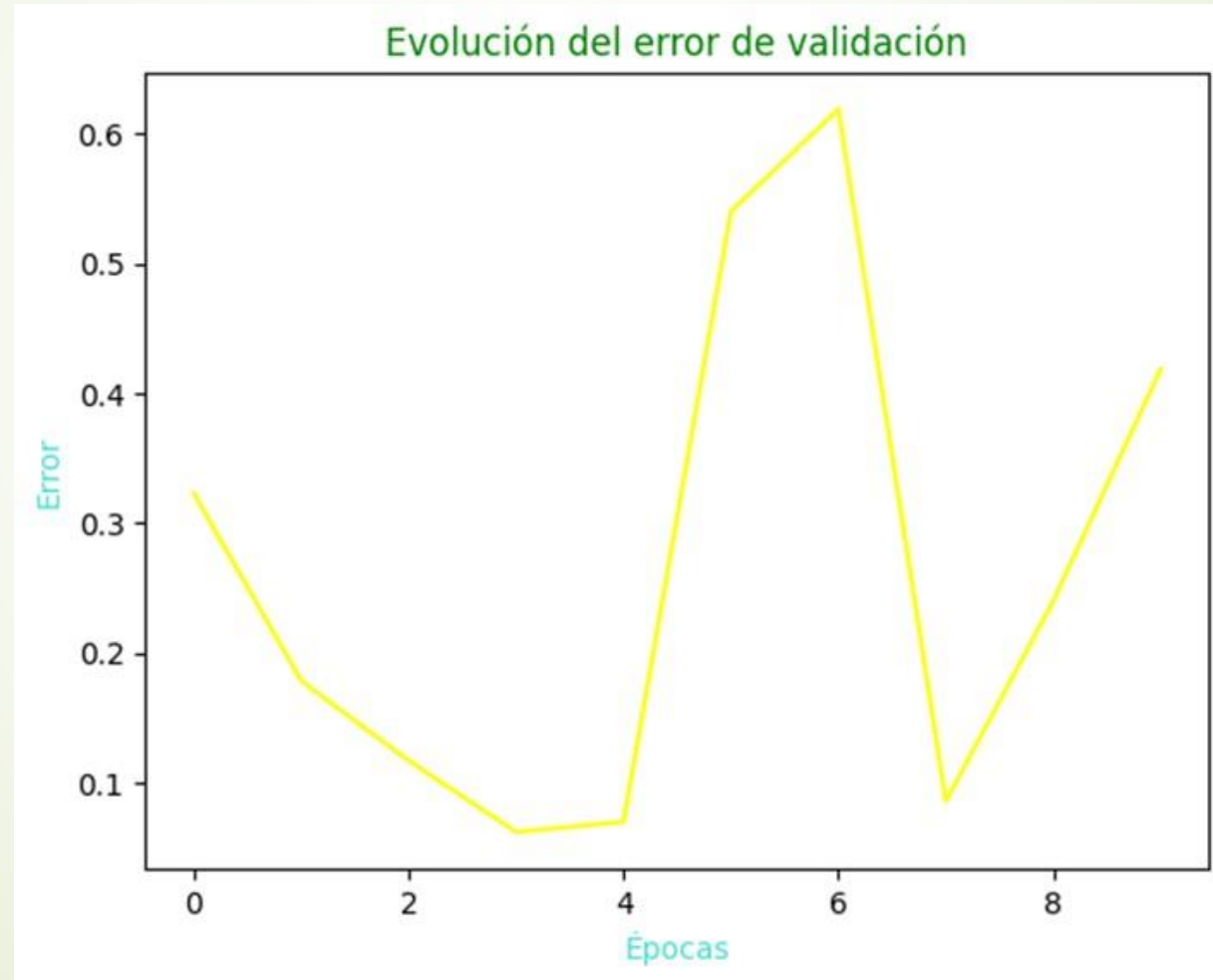


GRÁFICAS DEL MODELO

Códigos gráfico 2: graficaremos la evolución del error de validación del modelo de clasificación de rayos X de tórax durante el entrenamiento. Es otra forma de graficar y observar como se comportan los datos.

```
plt.plot(history1.history["val_loss"], color='yellow')
plt.title("Evolución del error de validación", color='green')
plt.xlabel("Épocas", color='turquoise')
plt.ylabel("Error", color='turquoise')
plt.show()
```

GRÁFICAS DEL MODELO

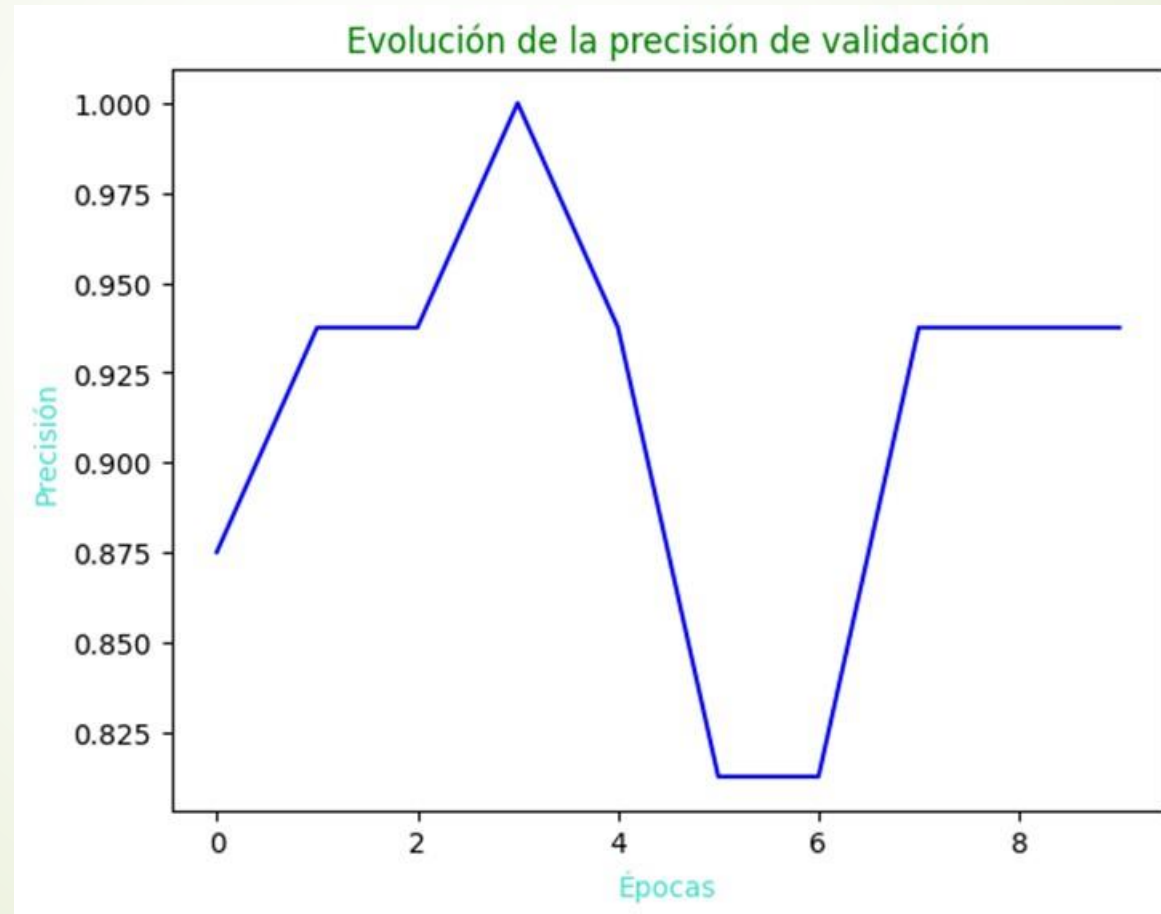


GRÁFICAS DEL MODELO

Códigos gráfico 3: graficaremos la evolución de la precisión de validación del modelo de clasificación de rayos X de tórax durante el entrenamiento.

```
plt.plot(history1.history["val_accuracy"], color='blue')
plt.title("Evolución de la precisión de validación",
color='green')
plt.xlabel("Épocas", color='turquoise')
plt.ylabel("Precisión", color='turquoise')
plt.show()
```

GRÁFICAS DEL MODELO





PREDICCIÓN DEL MODELO

El **código num_predictions = 10** establece el número de predicciones que se realizarán en el conjunto de datos de prueba. En este caso, se realizarán 10 predicciones.

El **código test_images, test_labels = next(test_generator)** obtiene 10 imágenes y sus etiquetas correspondientes del conjunto de datos de prueba.

Por último, el **código predictions = model.predict(test_images)** realiza las predicciones del modelo en las imágenes de prueba.

```
# Predicciones con los datos de prueba
num_predictions = 10
test_images, test_labels = next(test_generator)
```




PREDICCIÓN DEL MODELO

Predicciones:

```
predictions = model1.predict(test_images)
```

Salida:

```
1/1 [=====] - 0s 82ms/step
```





MÉTRICAS DEL MODELO

ACCURACY: `accuracy_score(y_true, binary_predictions)` calcula la precisión del modelo. La función `accuracy_score()` toma como entrada dos vectores, uno con las etiquetas reales y otro con las predicciones del modelo. La función devuelve la proporción de predicciones correctas realizadas por el modelo.

Código:

```
accuracy = accuracy_score(y_true, binary_predictions)
print("Accuracy:", accuracy)
```

Resultado:

Accuracy: 0.75



MÉTRICAS DEL MODELO

PRECISIÓN: `precision_score(y_true, binary_predictions)` calcula la precisión del modelo. La función devuelve la proporción de predicciones correctas realizadas por el modelo, donde solo se consideran las predicciones positivas.

Código:

```
precision = precision_score(y_true, binary_predictions)
print("Precisión:", precision)
```

Resultado:

Precisión: 0.7142857142857143



MÉTRICAS DEL MODELO

RECALL: `recall_score(y_true, binary_predictions)` calcula el recall del modelo. La función devuelve la proporción de etiquetas positivas reales que fueron predichas correctamente por el modelo.

Código:

```
recall = recall_score(y_true, binary_predictions)
print("Recall:", recall)
```

Resultado:

Recall: 1.0



MÉTRICAS DEL MODELO

RECALL: `recall_score(y_true, binary_predictions)` calcula el recall del modelo. La función devuelve la proporción de etiquetas positivas reales que fueron predichas correctamente por el modelo.

Código:

```
recall = recall_score(y_true, binary_predictions)
print("Recall:", recall)
```

Resultado:

Recall: 1.0



MÉTRICAS DEL MODELO

F1-SCORE: `f1_score(y_true, binary_predictions)` calcula el F1-score del modelo. La función devuelve una medida del equilibrio entre la precisión y el recall del modelo.

Código:

```
f1 = f1_score(y_true, binary_predictions)
print("F1 Score:", f1)
```

Resultado:

F1 Score: 0.7906976744186047



MATRIZ DE CONFUSIÓN

MATRIZ DE CONFUSIÓN: `confusion_matrix(y_true, binary_predictions)` crea una matriz de confusión para un modelo de clasificación binaria. La matriz de confusión es una tabla que resume el rendimiento del modelo en términos de las predicciones correctas e incorrectas que realiza.

La matriz de confusión tiene cuatro celdas, una para cada combinación de predicción y etiqueta real. Las celdas de la esquina superior izquierda y la esquina inferior derecha representan las predicciones correctas, mientras que las celdas de la esquina superior derecha y la esquina inferior izquierda representan las predicciones incorrectas.



MATRIZ DE CONFUSIÓN

MATRIZ DE CONFUSIÓN:

Código:

```
conf_matrix = confusion_matrix(y_true,  
binary_predictions)  
print("Confusion Matrix:\n", conf_matrix)
```

Resultado:

Confusion Matrix:

```
[[ 6  9]  
 [ 0 17]]
```

MATRIZ DE CONFUSIÓN

GRÁFICO DE MATRIZ DE CONFUSIÓN: El código usa la función `sns.heatmap()` para crear el mapa de calor. En este caso, el código usa las siguientes opciones:

`annot=True` indica que se deben mostrar los valores de la matriz de confusión en el mapa de calor.

`fmt='d'` indica que los valores se deben mostrar como números enteros.

`cmap='Blues'` indica que se debe usar la paleta de colores "Blues".

`xticklabels=labels` indica que las etiquetas de los ejes x deben ser las etiquetas definidas anteriormente.

`yticklabels=labels` indica que las etiquetas de los ejes y deben ser las etiquetas definidas anteriormente.

Finalmente, el código usa la función `plt.show()` para mostrar el mapa de calor.

MATRIZ DE CONFUSIÓN

GRÁFICO DE MATRIZ DE CONFUSIÓN:

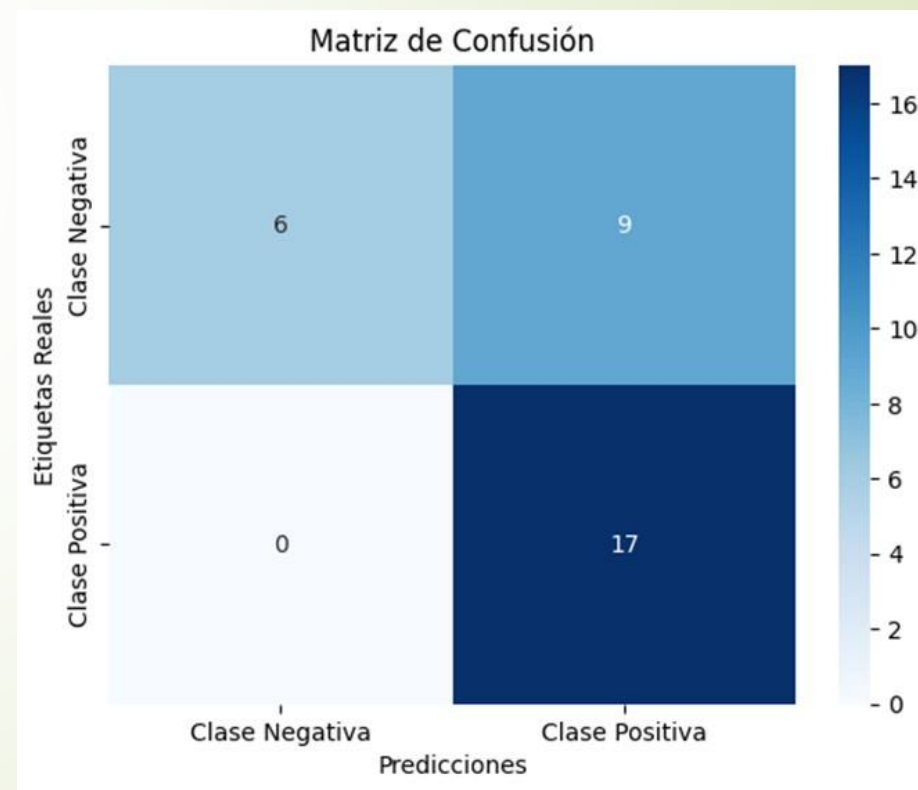
```
import seaborn as sns
import matplotlib.pyplot as plt

# Definir etiquetas para los ejes
labels = ['Clase Negativa', 'Clase Positiva']

# Crear la matriz de confusión
conf_matrix = confusion_matrix(y_true,
                                binary_predictions)

# Crear el mapa de calor
sns.heatmap(conf_matrix, annot=True, fmt='d',
             cmap='Blues', xticklabels=labels, yticklabels=labels)

# Configuraciones adicionales para mejorar la
# visualización
plt.title('Matriz de Confusión')
plt.xlabel('Predicciones')
plt.ylabel('Etiquetas Reales')
plt.show()
```



VISUALIZACIÓN DE IMÁGENES

Con el siguiente código mostrará una figura con 2 filas y 5 columnas de sub-plots, cada uno mostrando una imagen junto con la predicción correspondiente del modelo. Esto permite visualizar rápidamente un conjunto de imágenes con sus predicciones asociadas para evaluar el rendimiento del modelo.

```
plt.figure(figsize=(12, 6))
for i in range(num_predictions):
    plt.subplot(2, 5, i+1)
    plt.imshow(test_images[i])
    plt.title(f"Prediction:
{round(predictions[i][0])}")
    plt.axis('off')
```

VISUALIZACIÓN DE IMÁGENES

Prediction: 1



Prediction: 0



Prediction: 0



Prediction: 1



Prediction: 1



Prediction: 1



Prediction: 1



Prediction: 1



Prediction: 1



Prediction: 1





GUARDAR MODELOS

El código siguiente llama al método `save_weights()` del modelo, que guarda los pesos del modelo en un archivo. En este caso, el archivo se guardará como `model3_weights.h5`. Los modelos se guardan para el posterior proceso de ensamble.

```
model.save_weights('model_weights.h5')
```


CARGAR LOS TRES MODELOS

El siguiente código nos permite para cargar los tres modelos de redes neuronales convolucionales entrenados previamente: model1, model2, y model3.

1. Importaciones: `from tensorflow.keras.models import Sequential, load_model:` Importa las funciones Sequential para crear modelos CNNs y load_model para cargar modelos previamente guardados.

2. Definición de los modelos: Se definen tres modelos CNNs model1, model2, y model3 cada uno con una arquitectura específica (secuencias de capas convolucionales, pooling, capas densas, etc.).

3. Carga de pesos: `model1.load_weights('model1_weights.h5')`: Carga los pesos previamente entrenados del modelo model1 desde el archivo model1_weights.h5. `model2.load_weights('model2_weights.h5')`: similarmente, carga los pesos del modelo model2 desde su propio archivo de pesos. `model3.load_weights('model3_weights.h5')`: similarmente, carga los pesos del modelo model3 desde su propio archivo de pesos.

CARGAR LOS TRES MODELOS

```
model1 = Sequential([
    Conv2D(32, (3, 3), activation='relu', input_shape=(150, 150, 3)),
    MaxPooling2D((2, 2)),
    Conv2D(64, (3, 3), activation='relu'),
    MaxPooling2D((2, 2)),
    Flatten(),
    Dense(64, activation='relu'),
    Dense(1, activation='sigmoid')])
model1.load_weights('model1_weights.h5')
model2 = Sequential([
    Conv2D(64, (3, 3), activation='relu', input_shape=(150, 150, 3)),
    MaxPooling2D((2, 2)),
    Conv2D(128, (3, 3), activation='relu'),
    MaxPooling2D((2, 2)),
    Flatten(),
    Dense(128, activation='relu'),
    Dense(1, activation='sigmoid')])
model2.load_weights('model2_weights.h5')
model3 = Sequential([
    Conv2D(128, (3, 3), activation='relu', input_shape=(150, 150, 3)),
    MaxPooling2D((2, 2)),
    Conv2D(128, (3, 3), activation='relu'),
    MaxPooling2D((2, 2)),
    Conv2D(128, (3, 3), activation='relu'),
    MaxPooling2D((2, 2)),
    Flatten(),
    Dense(128, activation='relu'),
    Dense(1, activation='sigmoid')])
model3.load_weights('model3_weights.h5')
```



PROCESO DE HIPERPARÁMETROS

El **proceso de hiperparámetros** nos sirve para mejorar el rendimiento de los modelos de aprendizaje automático. El **hiperparámetro** consiste en ajustar los parámetros de los modelos para que estos funcionen mejor en un conjunto de datos específico.

DEFINIR EL DATA GENERATORS: Definimos dos generadores de datos para entrenar y validar el modelo de aprendizaje profundo basado en imágenes. Vamos a desglosarlo paso a paso:

- 1. Definir el tamaño del batch:** `batch_size = 32` define el tamaño del batch, que es la cantidad de imágenes que se procesarán y alimentarán a la red neuronal en cada paso de entrenamiento. En este caso, el modelo entrenará con grupos de 32 imágenes a la vez.
- 2. Crear ImageDataGenerator:** `train_datagen` y `val_datagen` crean dos instancias de `ImageDataGenerator`. Esta clase sirve para preprocesar y aumentar el dataset de imágenes.
- 3. Crear data generators:** `train_generator` y `val_generator` crean dos data generators usando el método `flow_from_directory`. Este método genera grupos de imágenes (batches) a partir de directorios que contienen carpetas para cada clase.



PROCESO DE HIPERPARÁMETROS

```
# Definir el data generators
batch_size = 32

train_datagen = ImageDataGenerator(rescale=1./255)
val_datagen = ImageDataGenerator(rescale=1./255)

train_generator = train_datagen.flow_from_directory(
    train_path,
    target_size=(150, 150),
    batch_size=batch_size,
    class_mode='binary')

val_generator = val_datagen.flow_from_directory(
    val_path,
    target_size=(150, 150),
    batch_size=batch_size,
    class_mode='binary')
```



PROCESO DE HIPERPARÁMETROS

DEFINIR FUNCIÓN DE CREACIÓN DEL MODELO:

- 1. Definición de la función:** `create_model(hp)` define la función que toma un hiperparámetro (`hp`) como entrada.
- 2. Construcción del modelo:** `model = Sequential([...])` inicia un modelo secuencial que agrega capas una tras otra.
- 3. Compilación del modelo:** `model.compile(...)`: configura el modelo para el entrenamiento.
- 4. Retorno del modelo:** `return model`: La función devuelve el modelo compilado para ser utilizado en el proceso de entrenamiento.



PROCESO DE HIPERPARÁMETROS

```
def create_model(hp):  
    model = Sequential([  
        Conv2D(hp.Int('conv1_units', min_value=32,  
max_value=128, step=32), (3, 3), activation='relu',  
input_shape=(150, 150, 3)),  
        MaxPooling2D((2, 2)),  
        Flatten(),  
        Dense(1, activation='sigmoid')  
    ])  
  
    model.compile(optimizer=Adam(learning_rate=hp.Choice('learn  
ing_rate', values=[1e-2, 1e-3, 1e-4])),  
                  loss='binary_crossentropy',  
                  metrics=['accuracy'])  
  
    return model
```




PROCESO DE HIPERPARÁMETROS

CONFIGURAR EL SINTONIZADOR

Este código configura un sintonizador de hiperparámetros llamado tuner utilizando la librería KerasTuner. El objetivo es buscar los mejores valores para los hiperparámetros del modelo y mejorar su rendimiento en la validación.

1. Importación: Asumimos que ya importaste KerasTuner y otras librerías necesarias.

2. Clase RandomSearch: `tuner = RandomSearch(...)`: Instancia la clase RandomSearch del módulo KerasTuner.



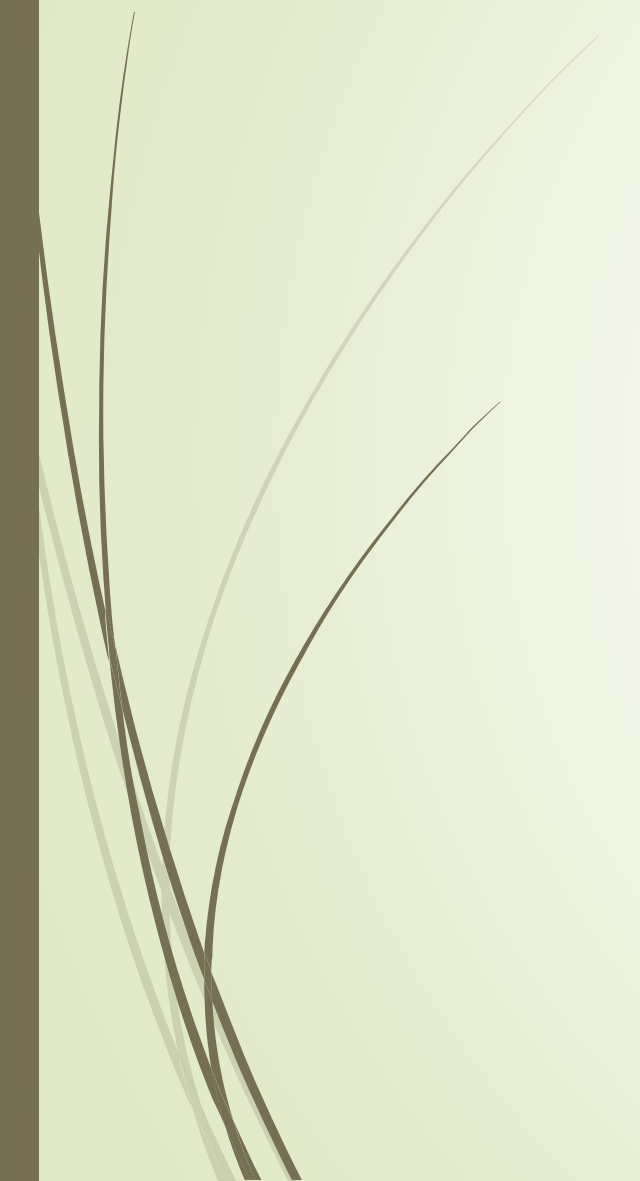
PROCESO DE HIPERPARÁMETROS

3. Argumentos de RandomSearch: `create_model`: Es la función que define tu modelo (la que explicaste anteriormente). KerasTuner la utilizará para generar diferentes versiones del modelo con distintas configuraciones de hiperparámetros. `objective='val_accuracy'`: Especifica el objetivo de la búsqueda. En este caso, quieres maximizar la precisión en el conjunto de validación (`val_accuracy`). `max_trials=5`: Limita el número máximo de intentos a 5. `directory='my_dir'`: Define el directorio donde se guardarán los resultados de la búsqueda de hiperparámetros. `project_name='helloworld'`: Establece un nombre identificativo para tu proyecto.

4. Funcionamiento: `tuner.search` busca iterativamente los mejores valores para los hiperparámetros definidos en la función `create_model`. Evalúa cada variante del modelo en el conjunto de validación y guarda las configuraciones con mejor precisión.



PROCESO DE HIPERPARÁMETROS



```
#Configurar el sintonizador  
tuner = RandomSearch(  
    create_model,  
    objective='val_accuracy',  
    max_trials=5,  
    directory='my_dir',  
    project_name='helloworld')
```



PROCESO DE HIPERPARÁMETROS

BUSCAR HIPERPARÁMETROS

Este código inicia la búsqueda de hiperparámetros utilizando el sintonizador tuner que se configuró anteriormente.

```
# Buscar hiperparámetros
tuner.search(train_generator,
             epochs=10,
             validation_data=val_generator,
             steps_per_epoch=len(train_generator),
             validation_steps=len(val_generator))
```

PROCESO DE HIPERPARÁMETROS

Explicación del código:

1. Argumentos de search(): train_generator: Es el generador de datos para el entrenamiento definido previamente con flow_from_directory. Proporciona imágenes y etiquetas para entrenar el modelo con diferentes configuraciones de hiperparámetros. epochs=10: Establece el número de épocas (pasadas sobre el conjunto de entrenamiento) para cada intento de búsqueda. En este caso, cada modelo generado se entrena durante 10 épocas. validation_data=val_generator: Especifica el generador de datos para la validación. Contiene imágenes y etiquetas del conjunto de validación para evaluar el rendimiento de cada modelo con los diferentes valores de hiperparámetros. steps_per_epoch=len(train_generator): Define el número de pasos por época en el entrenamiento. Indica cuántas veces el generador recorrerá el conjunto de entrenamiento completo en cada época. validation_steps=len(val_generator): Define el número de pasos para la validación. Especifica cuántas veces el generador recorrerá el conjunto de validación para la evaluación de cada modelo.



PROCESO DE HIPERPARÁMETROS

Explicación del código:

2. Funcionamiento: Con estos argumentos, el sintonizador generará diferentes versiones del modelo utilizando la función `create_model` y probando distintas configuraciones de los hiperparámetros definidos en esa función (por ejemplo, la cantidad de filtros en la convolución o la tasa de aprendizaje). Para cada modelo generado, lo entrenará durante 10 épocas con el `train_generator` y evaluará su rendimiento en el `validation_data`. Después de cada intento, el sintonizador registrará la precisión en la validación y la configuración de hiperparámetros que la generó. Al completar todos los intentos (los 5 especificados en la configuración), el sintonizador seleccionará el modelo con la mejor precisión en la validación y te dará acceso a él.



PROCESO DE HIPERPARÁMETROS

OBTENER EL MEJOR HIPERPARÁMETRO:

Este código recupera los mejores hiperparámetros encontrados por el sintonizador tuner.

1. `tuner.oracle.get_best_trials(num_trials=1)[0]`: `tuner.oracle`: Accede al oráculo interno del sintonizador, que almacena los resultados de todas las pruebas de hiperparámetros. `.get_best_trials(num_trials=1)`: Obtiene los mejores intentos (trials) de búsqueda. Con `num_trials=1`, solo recuperas el intento con la mejor precisión. `[0]`: Extrae el primer elemento de la lista de intentos (que será el único en este caso).



PROCESO DE HIPERPARÁMETROS

- 2. `.hyperparameters`:** Accede al diccionario de hiperparámetros utilizados en ese intento.
- 3. `best_hps`:** Asigna el diccionario de hiperparámetros del mejor intento a la variable `best_hps`.

Código:

```
best_hps = tuner.oracle.get_best_trials(num_trials=1)[0].hyperparameters
```



CONSTRUIR EL MODELO

1. **tuner.hypermodel:** Accede al hipermodelo utilizado por el sintonizador. Este hipermodelo es una función que genera modelos con diferentes configuraciones de hiperparámetros.
2. **.build(best_hps):** Utiliza los hiperparámetros best_hps para construir un nuevo modelo.
3. **models:** Asigna el nuevo modelo a la variable models.

Código:

```
models = tuner.hypermodel.build(best_hps)
```

COMPILAR EL MODELO

PREDECIR: Ahora para realizar predicciones utilizando los modelos individuales en el clasificador de ensamble ocuparemos la siguiente función. La función recibe dos argumentos:**

models: Una lista de los modelos individuales que forman parte del ensamble.

X: Los datos de entrada para realizar las predicciones.

La función funciona de la siguiente manera:

- Para cada modelo en la lista, se realiza una predicción para los datos de entrada.
- Las predicciones de todos los modelos se suman.
- La suma se divide por el número de modelos en la lista.

Código:

```
def ensemble_predict(models, X):  
    predictions = [model.predict(X) for model in models]  
    return sum(predictions) / len(models)
```

COMPILAR EL MODELO

1. Definir el modelo:

`model = Sequential([...])`: Se crea un modelo secuencial, que es un tipo de modelo CNN que consta de una secuencia de capas.**

`Conv2D(32, (3, 3), activation='relu', input_shape=(150, 150, 3))`: Se agrega una capa convolucional de 32 filtros de tamaño 3x3 con activación ReLU. La forma de entrada para la capa es (150, 150, 3), que corresponde a imágenes de 150x150 píxeles con tres canales de color.

`MaxPooling2D((2, 2))`: Se agrega una capa de pooling máximo de tamaño 2x2.

`Dense(64, activation='relu')`: Se agrega una capa densa de 64 neuronas con activación ReLU.

`Dense(1, activation='sigmoid')`: Se agrega una capa densa de una neurona con activación sigmoide. Esta capa se utiliza para realizar la clasificación binaria.



COMPILAR EL MODELO

2. Compilación del modelo:

`model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])`: Se compila el modelo. Los argumentos de la función `compile()` son los siguientes:

`optimizer`: El optimizador utilizado para entrenar el modelo.

`loss`: La función de pérdida utilizada para medir el error del modelo.

`metrics`: Las métricas utilizadas para evaluar el rendimiento del modelo.

COMPILAR EL MODELO

```
models = Sequential([
    Conv2D(32, (3, 3), activation='relu', input_shape=(150, 150, 3)),
    MaxPooling2D((2, 2)),
    Conv2D(64, (3, 3), activation='relu'),
    MaxPooling2D((2, 2)),
    Flatten(),
    Dense(64, activation='relu'),
    Dense(1, activation='sigmoid') # Capa de salida para
clasificación binaria
])

model.compile(optimizer='adam',
              loss='binary_crossentropy',
              metrics=['accuracy'])
```



COMPILAR EL MODELO

GENERADOR DE IMÁGENES:

Definición de las variables: test_data_dir: El directorio que contiene las imágenes de prueba. batch_size: El tamaño de los lotes a utilizar para el entrenamiento.

Creación del generador de imágenes: test_datagen: Se crea un objeto de la clase ImageDataGenerator con el parámetro rescale=1./255 para normalizar las imágenes. test_generator: Se crea un generador de imágenes utilizando el objeto test_datagen. El generador toma como argumentos los siguientes parámetros: test_data_dir: El directorio que contiene las imágenes de prueba. target_size=(150, 150): El tamaño de las imágenes de entrada. batch_size=batch_size: El tamaño de los lotes a utilizar. class_mode='binary': El modo de clasificación de las imágenes. shuffle=False: Evita que las imágenes se mezclen al generar los lotes. Esto es importante para que las predicciones coincidan con las etiquetas reales.

El generador de imágenes devuelve una tupla con los siguientes elementos: x: Las imágenes del lote. y: Las etiquetas del lote.

COMPILAR EL MODELO

Código GENERADOR DE IMÁGENES:

```
# Directorio que contiene las imágenes de prueba
test_data_dir = 'D:\Data Science\Modulo 7\Proyecto
7\chest_xray/test'

# Crear un generador de imágenes para el conjunto de prueba
test_datagen = ImageDataGenerator(rescale=1./255)
test_generator = test_datagen.flow_from_directory(
    test_data_dir,
    target_size=(150, 150),
    batch_size=batch_size,
    class_mode='binary',
    shuffle=False # importante para que las predicciones coincidan
con las etiquetas reales
)
```



COMPILAR EL MODELO

IMÁGENES Y ETIQUETAS DE PRUEBA: El siguiente código se utiliza para obtener las imágenes y las etiquetas del conjunto de prueba de un modelo de clasificación de imágenes.

Código:

```
X_test, test_images = [], []
```



COMPILAR EL MODELO

HACER PREDICCIONES: El siguiente código se utiliza para realizar predicciones utilizando los tres modelos CNN preentrenados (model, model2, y model3). El código consta de los siguientes pasos:

- 1. Llamada a la función predict():** Para cada modelo, se llama a la función predict(). La función predict() devuelve una matriz de predicciones para el conjunto de prueba.
- 2. Almacenamiento de las predicciones:** Las predicciones de cada modelo se almacenan en una variable separada.
- 3. Resultado:** Las variables predictions, predictions2, y predictions3 contienen las predicciones de los tres modelos.

COMPILAR EL MODELO

HACER PREDICCIONES:

```
# Hacer predicciones usando los modelos individuales
predictions = model1.predict(test_generator)
predictions2 = model2.predict(test_generator)
predictions3 = model3.predict(test_generator)
```

```
20/20 [=====] - 6s 258ms/step 20/20
[=====] - 5s 229ms/step 20/20
[=====] - 7s 315ms/step
```

COMPILAR EL MODELO

COMBINAR PREDICCIONES: El siguiente código se utiliza para combinar las predicciones de los tres modelos CNN preentrenados (predictions, predictions2, y predictions3) utilizando la estrategia de votación promedio. El código consta de los siguientes pasos:**

1. Suma de las predicciones: Se suman las predicciones de los tres modelos.

2. División por 3: La suma se divide por 3 para obtener una predicción final para cada imagen.

3. Resultado: La variable ensemble_predictions contiene las predicciones combinadas.

Código:

```
ensemble_predictions = (predictions + predictions2 + predictions3) / 3
```

COMPILAR EL MODELO

OBTENER ETIQUETAS REALES DEL GENERADOR DE PRUEBA: El siguiente código se utiliza para obtener las etiquetas reales del conjunto de prueba de un modelo de clasificación de imágenes. El código consta de los siguientes pasos:

- 1. Obtención de las etiquetas:** La función `classes()` del generador de imágenes `test_generator` devuelve una lista con las etiquetas reales del conjunto de prueba.
- 2. Almacenamiento de las etiquetas:** La variable `y_true` se utiliza para almacenar las etiquetas reales.
- 3. Resultado:** La variable `y_true` contiene las etiquetas reales del conjunto de prueba.

Código:

```
y_true = test_generator.classes
```

COMPILAR EL MODELO

CONVERTIR LAS PREDICCIONES: El siguiente código se utiliza para convertir las predicciones de un modelo de clasificación binaria a clases binarias (0 o 1). El código consta de los siguientes pasos:

- 1. Comparación con 0.5:** Se compara cada predicción con el valor 0.5.
- 2. Conversión a True o False:** Los valores de la predicción que son mayores o iguales a 0.5 se convierten a True. Los valores de la predicción que son menores a 0.5 se convierten a False.
- 3. Conversión a enteros:** Los valores de la predicción se convierten a enteros.
- 4. Resultado:** La variable `ensemble_classes` contiene las clases binarias de las predicciones.

Código:

```
ensemble_classes = (ensemble_predictions > 0.5).astype(int)
```



COMPILAR EL MODELO

CALCULO DE MÉTRICAS:

Los siguientes códigos nos permitena calcular las métricas de rendimiento. El código consta de los siguientes pasos:

- 1. Importación de las funciones:** Se importan las funciones `accuracy_score()`, `precision_score()`, `recall_score()` y `f1_score()` de la biblioteca Scikit-Learn.
- 2. Cálculo de las métricas:** Se llaman a las funciones `accuracy_score()`, `precision_score()`, `recall_score()` y `f1_score()` para calcular las métricas de rendimiento.
- 3. Almacenamiento de las métricas:** Las métricas se almacenan en variables separadas.
- 4. Resultado:** Las variables `accuracy`, `precision`, `recall` y `f1` contienen las métricas de rendimiento del modelo.

COMPILAR EL MODELO

CALCULO DE MÉTRICAS:

Código:

```
accuracy = accuracy_score(y_true, ensemble_classes)
precision = precision_score(y_true, ensemble_classes)
recall = recall_score(y_true, ensemble_classes)
f1 = f1_score(y_true, ensemble_classes)
print("Accuracy del modelo ensamblado:", accuracy)
print("Precision del modelo ensamblado:", precision)
print("Recall del modelo ensamblado:", recall)
print("F1 Score del modelo ensamblado:", f1)
```

```
Accuracy del modelo ensamblado: 0.719551282051282
Precision del modelo ensamblado: 0.6916221033868093
Recall del modelo ensamblado: 0.9948717948717949
F1 Score del modelo ensamblado: 0.8159831756046267
```

COMPILAR EL MODELO

MATRIZ DE CONFUSIÓN:

Calcularemos y graficaremos la matriz de confusión del modelo ensamblado. La matriz de confusión es una tabla que muestra la distribución de las predicciones del modelo en comparación con las etiquetas reales.

Código:

```
conf_matrix = confusion_matrix(y_true, ensemble_classes)
print("Matriz de Confusión del modelo ensamblado:")
print(conf_matrix)
```

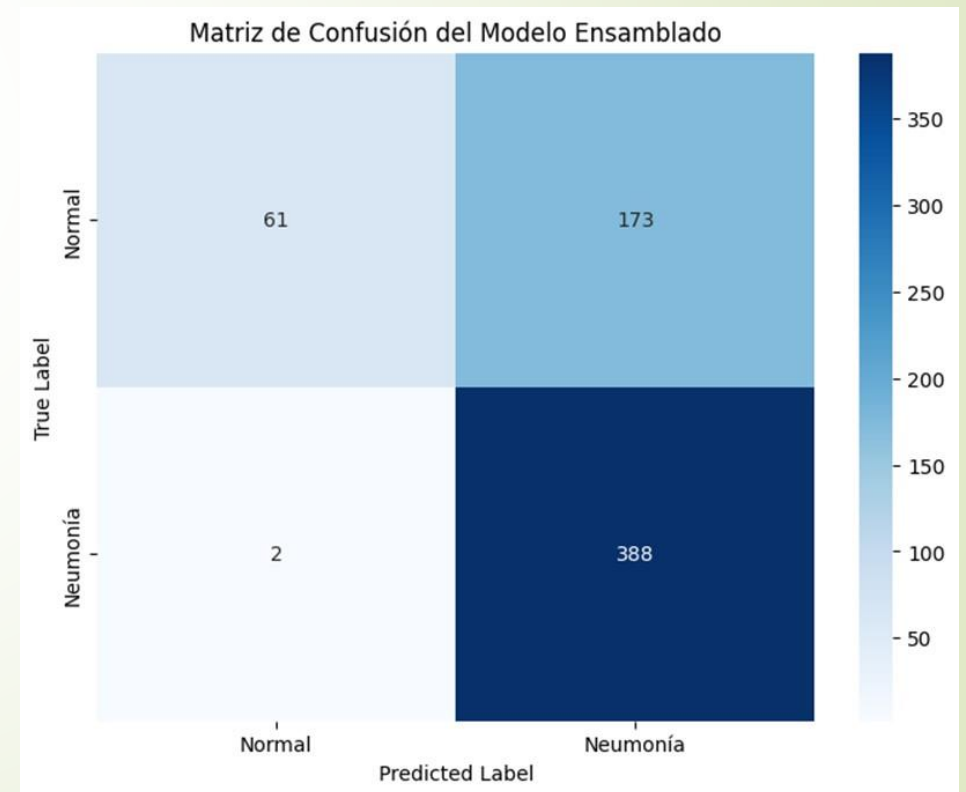
Resultado:

```
Matriz de Confusión del modelo ensamblado:
[[ 61 173]
 [ 2 388]]
```

COMPILAR EL MODELO

GRÁFICO MATRIZ DE CONFUSIÓN:

```
plt.figure(figsize=(8, 6))
sns.heatmap(conf_matrix, annot=True,
            fmt='d', cmap='Blues',
            xticklabels=['Normal', 'Neumonía'],
            yticklabels=['Normal', 'Neumonía'])
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.title('Matriz de Confusión del Modelo Ensamblado')
plt.show()
```





COMPILAR EL MODELO

GUARDAR EL MODELO:

Por último, debemos guardar el modelo para crear posteriormente la API.

Código:

```
models.save('modelo_ensamblado.h5')
```



CONCLUSIÓN

En general, el rendimiento del modelo ensamblado es bueno. El modelo es capaz de clasificar correctamente la mayoría de las imágenes, tanto positivas como negativas.


Sin embargo, hay algunas áreas en las que el modelo podría mejorar. Para mejorar el rendimiento del modelo, se podrían realizar los siguientes cambios:

Se podría entrenar el modelo ensamblado con un conjunto de datos más grande. Esto ayudaría al modelo a aprender a clasificar imágenes más precisa y consistentemente.

Se podría ajustar la configuración del modelo ensamblado. Por ejemplo, se podría cambiar el umbral de clasificación para mejorar la precisión o la sensibilidad.



NOTAS



Las conclusiones e interpretaciones de imágenes, gráficas, métricas de rendimiento, etc. se encuentran en el modelo que se subió a github.