MEMORIA LABORATORIO III

Julián Prieto Velasco, Pedro José Paniagua Falo,
Guillermo Ramírez Cárdenas, Javier Muñoz Rojas

Índice

Índice de Ilustraciones

1 Introducción

La siguiente memoria describe el desarrollo de una práctica de laboratorio en la que se implementa un modelo de DNN para clasificación usando redes de convolución, utilizando Python y las librerías *NumPy*, *pandas*, *scikit-learn, Matplotlib, Keras y TensorFlow*. Consiste en desarrollar de forma autónoma distintas implementaciones de redes neuronales profundas de aprendizaje supervisado que permitan resolver distintos casos de uso.

En la primera práctica, se implementa una CNN con *Keras* para realizar dos clasificaciones diferentes. Los resultados se entregan en el archivo L3P1-Pneumonia.ipynb.

En la segunda práctica, se escoge un *dataset* en KAGGLE acerca de la clasificación de imágenes de zapatillas. El *dataset* completo se encuentra dentro de la carpeta "archive2". Se crea un CNN, probando con diferentes números de neuronas en la capa oculta y varios valores de épocas, usando como optimizador 'SGD', funciones de activación *'ReLU'* y *'Softmax'*, y función de error *'categorical_crossentropy'*.

Una CNN (*Convolutional Neural Network*) es un tipo de red neuronal profunda utilizada principalmente para el procesamiento de imágenes y reconocimiento de patrones. Se compone de capas de convolución, capas de *pooling* y capas completamente conectadas.

En las capas de convolución, se utilizan filtros que se aplican a la imagen de entrada para detectar características relevantes en la imagen, como bordes o texturas. Luego, en las capas de *pooling*, se reduce la dimensión de la imagen para disminuir la cantidad de parámetros y acelerar el entrenamiento. Finalmente, en las capas completamente conectadas, se utiliza la información recopilada por las capas anteriores para clasificar la imagen en las diferentes categorías.

En resumen, las CNN utilizan una arquitectura específica para procesar imágenes y extraer características útiles para la clasificación o el análisis de imágenes. Esta arquitectura es muy efectiva para tareas de visión por computadora, como el reconocimiento de objetos, la detección de rostros, la clasificación de imágenes médicas, entre otras.

2 Práctica 1

2.1 Objetivo

El objetivo de esta práctica es analizar el *dataset* de radiografías torácicas *Chest X-Chest X-Ray Images* (*Pneumonia*). El *dataset* contiene 3 carpetas (*train,test,val*) y por cada carpeta hay dos etiquetas(Pneumonia/Normal).Dentro de la carpeta Pneumonia se distingue entre bacteriana y vírica.

Hay que hacer dos clasificaciones:

1. Pneumonia(ambos tipos como si fuesen uno solo) vs Normal
2. Pneumonia bacteriana vs Pneumonía vírica vs Normal (3 opciones a clasificar)

Se ha buscado en internet en busca de algún ejemplo parecido y se ha encontrado un enlace de la página "Saturdays.ia" [1] donde se explica la misma problemática que en esta práctica y además se han encontrado varios TFG de los cuales se ha obtenido el modelo pre-entrenado. [2] [3] [4]

2.2 implementación

En esta parte de la práctica se van a explicar las decisiones tomadas para conseguir la solución de la práctica.

2.3 Pneumonia vs normal

En este caso se tratan de contrastar aquellas imágenes que pertenezcan a pulmones normales frente a aquellas que pertenecen a ambos tipos de Pneumonia. Para poder hacer esta clasificación, se usa la carpeta *chest_array* dentro de la carpeta archive. Como se observa en la Ilustración.

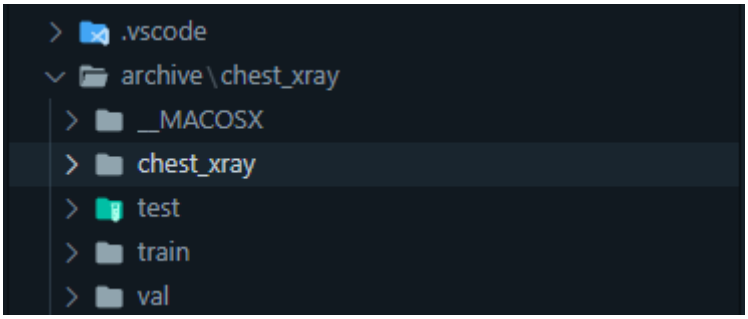


Ilustración 1 Carpetas usadas para el desarrollo de la práctica

En primer lugar, se realiza la importación de librerías necesarias para el correcto funcionamiento de la práctica.

```
from PIL import Image
import os
import shutil
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from keras.models import Sequential
from keras.layers import Conv2D, MaxPooling2D, Flatten, Dense
from keras.preprocessing.image import ImageDataGenerator
from keras.optimizers import Adam
from keras.utils import to_categorical, plot_model
from tensorflow.keras.preprocessing import image
from tensorflow.keras.layers import Dropout, BatchNormalization
from tensorflow.keras.preprocessing.image import ImageDataGenerator,array_to_img, img_to_array, load_img
from tensorflow.keras import optimizers,layers
from sklearn.metrics import confusion_matrix
import tensorflow as tf
from IPython.display import Image
import itertools
import seaborn as sns
from keras.models import Sequential
from keras.layers import Dense, Activation, Dropout, Flatten, Conv2D, MaxPooling2D
from keras.optimizers import SGD
from IPython.display import Image
```

Ilustración 2 Código de las librerías.

Inicialmente se trató de entrenar a la IA con los datos presentes en el *dataset* de *Kaggle*, sin realizar ninguna alteración o crear nuevos datos, no obstante, la IA sufría de *overfitting* en la que no era capaz de generalizar pues no había datos suficientes para lograr un entrenamiento y por lo tanto no logrando el objetivo, el *dataset* original presentaba una cantidad de datos muy pequeña en el archivo de “normal” frente a “Pneumonia”, provocando el *overfitting*, de manera que, antes de poder dividir los datos en conjuntos de entrenamiento, validación y test se decide crear más imágenes de Normal y Pneumonia.

2.3.1 Preprocesamiento de datos

Para mostrar la diferencia de la distribución de los datos entre las clases se ha decidido generar diagramas de barras para asegurar que la hipótesis de no generalización por *overfitting* de la CNN era correcta. El código y la salida son los siguientes:

```
# Ruta de la carpeta con los datos
ruta_datos = "archive/chest_xray/train/"

# Crear una lista con los nombres de todas las subcarpetas en la carpeta con los datos
subcarpetas = [nombre for nombre in os.listdir(ruta_datos) if os.path.isdir(os.path.join(ruta_datos, nombre))]

# Crear un diccionario con la información del número de datos en cada subcarpeta
datos = {}
for subcarpeta in subcarpetas:
    ruta_subcarpeta = os.path.join(ruta_datos, subcarpeta)
    numero_datos = len(os.listdir(ruta_subcarpeta))
    datos[subcarpeta] = numero_datos

# Crear un dataframe con la información del número de datos en cada subcarpeta
df = pd.DataFrame.from_dict(datos, orient='index', columns=['Número de datos'])

# Crear un gráfico de barras para visualizar la diferencia en el número de datos
plt.bar(df.index, df['Número de datos'])
plt.title('Diferencia en el número de datos')
plt.xlabel('Subcarpeta')
plt.ylabel('Número de datos')
plt.show()
```

Ilustración 3 Código de gráfico de barras con el número de imágenes de cada clase

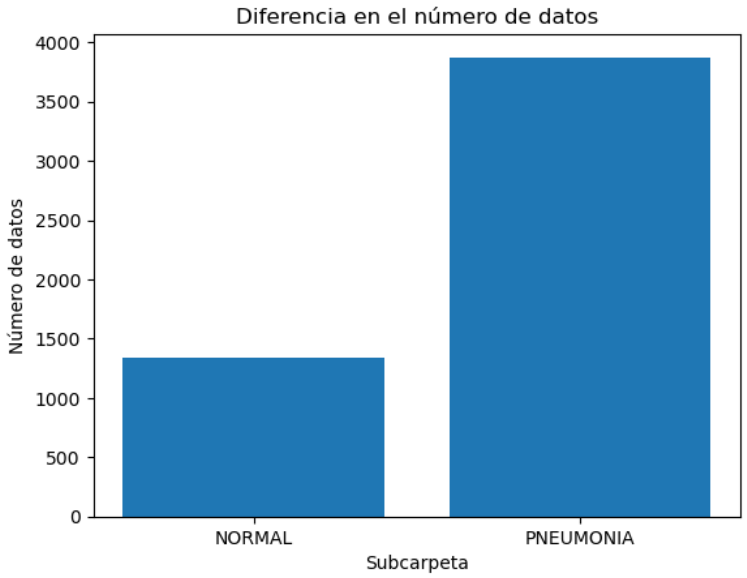


Ilustración 4 Diagrama de barras para las clases Normal y Pneumonia

Como se observa en el diagrama de barras la carpeta de *"PENUMONIA"* tiene muchas más imágenes que la carpeta *"NORMAL"* para solucionar esto se decide crear nuevos datos sintéticos.

Para ello se implementa el siguiente código:

```python
# Ruta de la carpeta con las imágenes originales
ruta_originales = "archive/chest_xray/train/NORMAL/"

# Cargar todas las imágenes en la carpeta
imagenes_originales = []
for filename in os.listdir(ruta_originales):
    imagen = Image.open(os.path.join(ruta_originales, filename))
    imagenes_originales.append(imagen)

# Definir las transformaciones de data augmentation
datagen = ImageDataGenerator(
        rotation_range=20,
        width_shift_range=0.1,
        height_shift_range=0.1,
        shear_range=0.2,
        zoom_range=0.2,
        horizontal_flip=True,
        fill_mode='nearest')
j = 0
# Generar nuevas imágenes utilizando las transformaciones definidas
for imagen in imagenes_originales:
    if j < 1200:
        x = imagen.resize((256, 256))
        x = x.convert('RGB')
        x = np.array(x)
        x = x.reshape((1,) + x.shape)
        i = 0
        for batch in datagen.flow(x, batch_size=1, save_to_dir='archive/chest_xray/train/NORMAL/', save_prefix='aug', save_format='jpeg'):
            i += 1
            if i > 2: # Generar 2 imágenes nuevas por cada imagen original
                break
        j += 1
```

Ilustración 5 Código de generación de imágenes

Como vemos, se generan el doble de imágenes de las que ya existen en la propia carpeta para poder acercar el número de imágenes de "*NORMAL"* a las de "*PNEUMONIA*" a partir de las transformaciones de las originales. En la carpeta de "*test*" realizamos lo mismo para que la IA al entrenar tenga más datos para así asegurar la buena generalización de la IA. Además, se generan más imágenes de normal que de pneumonía debido a que, estas tienen bastante ruido y el modelo necesita más imágenes para poder obtener las características necesarias para diferenciarlas, cosa que no ocurre con las imágenes de pneumonía.

Las imágenes se crean con el prefijo "*aug-" para* poder diferenciarlas de las originales y mantienen el formato de las que ya se tenían en el *dataset*. Las imágenes quedarían repartidas de la siguiente manera:

La carpeta de "*train*" quedarían los datos de la siguiente manera:

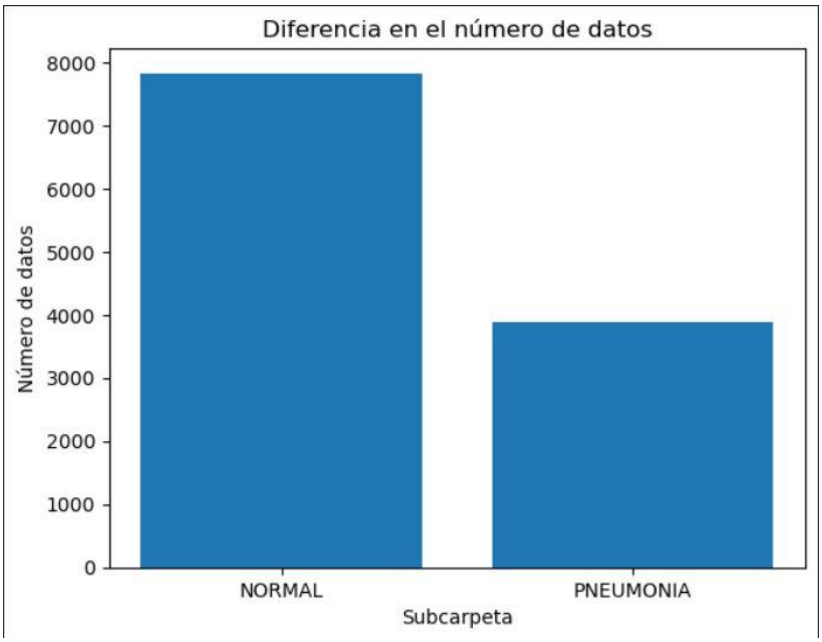


Ilustración 6 Gráfico de barras para las clases normal y penumonia en la carpeta train

La carpeta de "*test*" quedaría de esta manera:

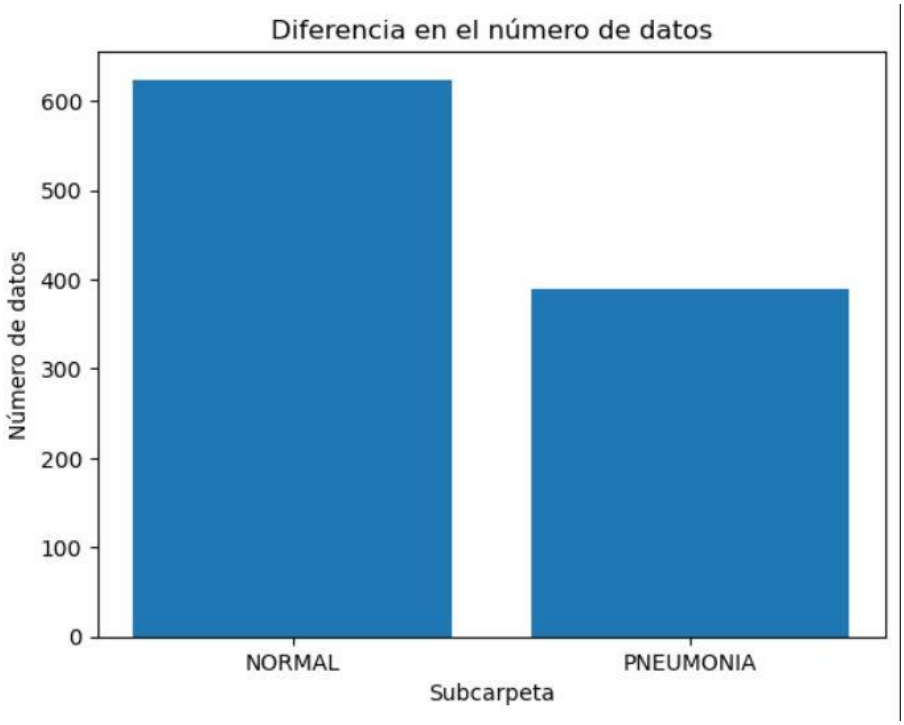


Ilustración 7 Gráfico de barras para las clases normal y penumonia en la carpeta test

Además, según la teoría vista en clase la distribución de las carpetas usadas debe ser alrededor de un 80% de entrenamiento, 5% de prueba y 20% de los datos, aproximadamente, por esto se ha decidido usar la carpeta denominada como *"test"* para hacer la validación durante el entrenamiento pues nos proporciona el porcentaje de imágenes adecuado.

El siguiente paso a realizar, ha sido la creación de generados a partir de los directorios del *dataset*, haciendo uso de la función *image_dataset_from_directory()* [5] de *keras*.

Primero, se definen los directorios donde se encuentran los conjuntos de datos de entrenamiento, validación y prueba. Luego, se especifica el tamaño de la imagen de entrada (256x256 píxeles), el tamaño del lote (*batch size*) y el número de épocas de entrenamiento (1000).

Para preprocesar las imágenes, se utiliza la clase *ImageDataGenerator* de *TensorFlow*, que permite aplicar una serie de transformaciones de forma aleatoria a las imágenes, como rotación, zoom, volteo, entre otras. En este caso, no se aplican transformaciones adicionales.

A continuación, se definen tres generadores de datos utilizando la función *image_dataset_from_directory()* de *TensorFlow*. Esta función permite cargar imágenes de un directorio y generar lotes de imágenes en formato tensor.

El primer generador se utiliza para el conjunto de datos de entrenamiento y se especifica que las imágenes están en escala de grises, que se deben generar lotes de tamaño 64 y que las etiquetas deben ser categóricas (es decir, una matriz con un valor 1 en la columna correspondiente a la clase y 0 en las demás columnas).

El segundo generador se utiliza para el conjunto de datos de validación y se especifica lo mismo que en el caso anterior, pero para el conjunto de datos de validación.

El tercer generador se utiliza para el conjunto de datos de prueba y se utiliza la función *flow_from_directory()* para generar lotes de imágenes. Se especifica que las imágenes están en escala de grises, que se deben generar lotes de tamaño 32 y que las etiquetas deben ser categóricas. Además, se especifica que no se deben barajar las imágenes.

En resumen, este código carga y preprocesa imágenes de radiografías de tórax para su uso en un modelo de clasificación de dos clases: 'Normal' y 'Pneumonia'. Los generadores de datos creados permiten generar lotes de imágenes y etiquetas para el entrenamiento y la evaluación del modelo.

```
# Directorios de entrenamiento, validación y prueba
train_dir = 'archive/chest_xray/train'
val_dir = 'archive/chest_xray/val'
test_dir = 'archive/chest_xray/test'

# Altura y anchura de la imagen
img_width, img_height = 256, 256
# Tamaño de lote
batch_size = 64
# Epocas
epochs = 1000

# Preprocesar los datos de imagen utilizando ImageDataGenerator

test_datagen = ImageDataGenerator()

train_generator_1 = tf.keras.preprocessing.image_dataset_from_directory(train_dir,
                                                    image_size=(img_width, img_height),
                                                    batch_size=batch_size,
                                                    color_mode="grayscale",
                                                    seed=1234,
                                                    label_mode='categorical',)

val_generator_1 = tf.keras.preprocessing.image_dataset_from_directory(val_dir,
                                                  image_size=(img_width, img_height),
                                                  batch_size=batch_size,
                                                  color_mode="grayscale",
                                                  seed=1234,
                                                  label_mode='categorical')

test_generator_1 = test_datagen.flow_from_directory(test_dir,
                                                   color_mode="grayscale",
                                                  seed=1234,
                                                  target_size=(img_height, img_width),
                                                  class_mode="categorical", batch_size=32,
                                                  shuffle=False)
```

Ilustración 8 Código de normalización de las imágenes

Este sería el último paso antes de comenzar a entrenar la CNN.

2.3.2 Arquitectura

En este apartado se van a definir todos los pasos llevados a cabo para definir la arquitectura utilizada para poder llegar a la solución.

Primero se define una función para detener el entrenamiento cuando se consigue un *loss* menor que 0.2. como se muestra en la siguiente ilustración.

```
class StopTrainingCallback(tf.keras.callbacks.Callback):
    def on_epoch_end(self, epoch, logs={}):
        if logs.get('loss') < 0.2:
            print('\nSe ha alcanzado el valor de pérdida deseado.')
            self.model.stop_training = True
```

Ilustración 9 Código para detener el entrenamiento cuando el loss es menor que 0,2

Se define la arquitectura de entrenamiento:

```
model = Sequential()
normalization_layer = layers.experimental.preprocessing.Rescaling(1./255, input_shape=(img_height, img_width, 1))
model.add(normalization_layer)
# capa de convolución 1
model.add(Conv2D(filters=96, kernel_size=(11,11), strides=(4,4), padding='valid', input_shape=(200,200,1)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(3,3), strides=(2,2), padding='valid'))

# capa de convolución 2
model.add(Conv2D(filters=256, kernel_size=(5,5), strides=(1,1), padding='same'))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(3,3), strides=(2,2), padding='valid'))

# capa de convolución 3
model.add(Conv2D(filters=384, kernel_size=(3,3), strides=(1,1), padding='same'))
model.add(Activation('relu'))

# capa de convolución 4
model.add(Conv2D(filters=384, kernel_size=(3,3), strides=(1,1), padding='same'))
model.add(Activation('relu'))

# capa de convolución 5
model.add(Conv2D(filters=256, kernel_size=(3,3), strides=(1,1), padding='same'))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(3,3), strides=(2,2), padding='valid'))

# capa completamente conectada 1
model.add(Flatten())
model.add(Dense(4096, input_shape=(200*200,)))
model.add(Activation('relu'))
model.add(Dropout(0.4))

# capa completamente conectada 2
model.add(Dense(4096))
model.add(Activation('relu'))
model.add(Dropout(0.4))

# capa de salida
model.add(Dense(2))
model.add(Activation('softmax'))

# compilar el modelo
model.compile(loss='categorical_crossentropy', optimizer=SGD(learning_rate = 0.01), metrics=['accuracy'])

stop_training_callback = StopTrainingCallback()
history = model.fit(
        train_generator_1,
        epochs = epochs,
        validation_data = test_generator_1,
        callbacks = [stop_training_callback],
)
```

Ilustración 10 Código de la arquitectura de la CNN para clasificar Pneumonia y normal

Se comienza declarando el modelo y realizando una capa de normalización donde se ajustan las imágenes para que todas cumplan con el tamaño establecido de 256 x 256 y que se realice en un único canal pues es una imagen en escala de grises, esto se realiza pues se necesita que todas las imágenes se encuentren en el mismo tamaño y dimensión de manera que cuando vayan a ser procesadas por la red no haya inconsistencias o se provoquen errores en el entrenamiento.

Esta es la arquitectura está inspirada en la red *AlexNet* [6], pues esta tiene una funcionalidad parecida y es similar, esta sirve para tener una base sobre la que trabajar.

- Normalización: La capa de normalización es una capa de preprocesamiento que escala los valores de píxeles de entrada entre 0 y 1. En este caso, se usa la capa *Rescaling* del módulo experimental de preprocesamiento de *Keras*, que normaliza la entrada dividiendo cada valor de píxel por 255.
- Capas de convolución: El modelo utiliza cinco capas de convolución con diferentes filtros de tamaño para extraer características de la imagen de entrada. Cada capa de convolución aplica un conjunto de filtros convolucionales a la entrada, creando un mapa de características. Los filtros convolucionales son matrices que se deslizan por la imagen de entrada y multiplican los valores de píxeles por los pesos correspondientes. La capa de convolución 1 tiene 96 filtros de tamaño 11x11, la capa de convolución 2 tiene 256 filtros de tamaño 5x5, la capa de convolución 3 tiene 384 filtros de tamaño 3x3, la capa de convolución 4 tiene 384 filtros de tamaño 3x3 y la capa de convolución 5 tiene 256 filtros de tamaño 3x3.
- Función de activación *ReLU*: Después de cada capa de convolución, se aplica una función de activación *Rectified Linear Unit* (*ReLU*) para introducir no linealidad en el modelo. La función *ReLU* establece a cero cualquier valor negativo y deja pasar cualquier valor positivo.
- Capas de Max Pooling: Después de cada capa de convolución, se aplica una capa de *Max Pooling* para reducir el tamaño del mapa de características y hacer que el modelo sea más eficiente en términos de memoria y cálculo. La capa de *Max Pooling* toma la salida de la capa de convolución y divide la imagen en regiones no superpuestas y aplica una operación de máximo para obtener el valor máximo de cada región.
- Capas completamente conectadas: Después de las capas de convolución y de *Max Pooling*, se utilizan dos capas completamente conectadas para clasificar las características extraídas en categorías. La primera capa completamente conectada tiene 4096 neuronas y la segunda capa completamente conectada también tiene 4096 neuronas.
- Capa de salida: La capa de salida es la última capa del modelo y tiene dos neuronas, una para cada posible clase de salida. En este caso, la función de activación *Softmax* se utiliza para generar la distribución de probabilidad de la salida.
- Compilación: Después de definir el modelo, se compila utilizando el optimizador SGD con una tasa de aprendizaje de 0.01 y la función de pérdida categórica_crossentropy. La métrica de evaluación es la precisión (*accuracy*) del modelo.

Se hace uso de SGD pues es un optimizador que para el entrenamiento de redes neuronales más simple que Adam y tiene menos cantidad de hiperparámetros. Es más eficaz en problemas con conjuntos de datos pequeños y con una tasa de aprendizaje adecuada puede converger a una solución cercana al óptimo global.

- Tasa de aprendizaje: Tras la realización de numerosas pruebas se comprueba que el *lr* optimo es 0.01, pues de otra manera la red no es capaz de aprender y si es más bajo sufre de generalización, además de tener un coste computación temporal muy alto.

Dibujo de la arquitectura de la red:

```
from IPython.display import Image
# Guarda la imagen del modelo
plot_model(model, to_file = 'modeloCalificación1.png', show_shapes=True)

# Evaluar el modelo con los datos de prueba
test_loss, test_acc = model.evaluate(val_generator_1)
print('Test accuracy:', test_acc)
print('Test loss:', test_loss)

# Muestra la imagen del modelo
Image(filename = 'modeloCalificación1.png')
```

Ilustración 11 Código para calcular el test loss, test accuracy y mostrar la arquitectura de la red

En este código se usa el modelo entrenado para crear una imagen de la red CNN a partir de este.

Se realiza una evaluación del modelo entrenado con el *dataset* de validación, obteniendo de esta tanto la precisión (*accuracy*) como el error (*loss*) de este. Finalmente se muestra la imagen de la red.

Gráfica de variación del error según las épocas y matriz de confusión:

Para hacer esta gráfica se han utilizado como ejes, el error y las épocas. Primero se obtienen los valores de error y precisión del entrenamiento y la validación. Se crean dos gráficas una con la variación del error y otra con la variación de la precisión.

```
# Obtener los valores de loss y accuracy del entrenamiento y validación
train_loss = history.history['loss']
train_acc = history.history['accuracy']
val_loss = history.history['val_loss']
val_acc = history.history['val_accuracy']

# Crear una figura con dos subplots para mostrar loss y accuracy
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 5))

# Gráfico de loss
ax1.plot(train_loss, label='train_loss')
ax1.plot(val_loss, label='val_loss')
ax1.set_title('Training and Validation Loss')
ax1.set_xlabel('Epoch')
ax1.set_ylabel('Loss')
ax1.legend()

# Gráfico de accuracy
ax2.plot(train_acc, label = 'train_acc')
ax2.plot(val_acc, label = 'val_acc')
ax2.set_title('Training and Validation Accuracy')
ax2.set_xlabel('Epoch')
ax2.set_ylabel('Accuracy')
ax2.legend()

plt.show()

# Obtener predicciones para el conjunto de prueba
y_pred = model.predict(test_generator_1)
# Convertir las probabilidades en etiquetas de clase
y_pred = np.argmax(y_pred, axis=1)
# Obtener las etiquetas verdaderas del conjunto de prueba
y_true = test_generator_1.classes
```

Ilustración 12 Código para mostrar gráficas de la variación de los patrones de entrenamiento y la matriz de confusión

Una vez entrenada la red se muestran las gráficas del *loss* y el *accuracy* por época. Las gráficas que se obtienen tras el entrenamiento son las siguientes:

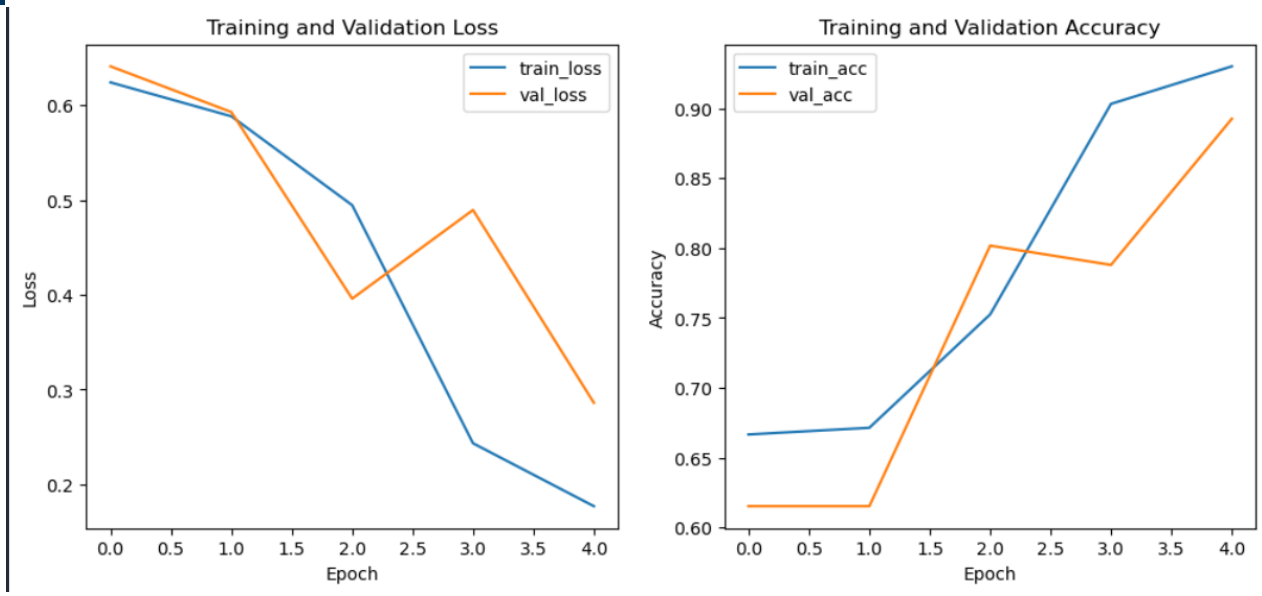


Ilustración 13 Variación del train loss, train accuracy, validation loss y train loss

Una matriz de confusión es una tabla que se utiliza para describir el rendimiento de un modelo de clasificación en un conjunto de datos de prueba, donde las etiquetas verdaderas son conocidas. Tiene filas y columnas que representan las etiquetas de clase verdaderas y las etiquetas de clase predichas por el modelo, respectivamente.

Cada celda de la matriz muestra el recuento de instancias en el conjunto de prueba que pertenecen a una combinación específica de etiqueta verdadera y etiqueta predicha. Proporciona una visión detallada de cómo un modelo de clasificación está realizando predicciones para cada clase en particular.

- Se utiliza el método "*predict*()" del modelo para hacer predicciones en el conjunto de prueba (test_generator_1). Esto devuelve un array de probabilidades para cada clase en lugar de etiquetas de clase directamente.
- Se utiliza la función "*np.argmax*()" para obtener las etiquetas de clase predichas a partir de las probabilidades obtenidas en el paso anterior. Esta devuelve el índice del valor máximo en un array, que en este caso corresponde a la etiqueta de clase con la mayor probabilidad de predicción.
- Se obtienen las etiquetas verdaderas del conjunto de prueba, que son las etiquetas reales de las imágenes del conjunto de prueba.
- Se crea una matriz de confusión (una tabla que muestra la cantidad de predicciones correctas e incorrectas para cada clase), para mostrarla utilizamos la librería "*seaborn*" para visualizar la matriz de confusión en forma de un mapa de calor, utilizando la función "*heatmap*()".

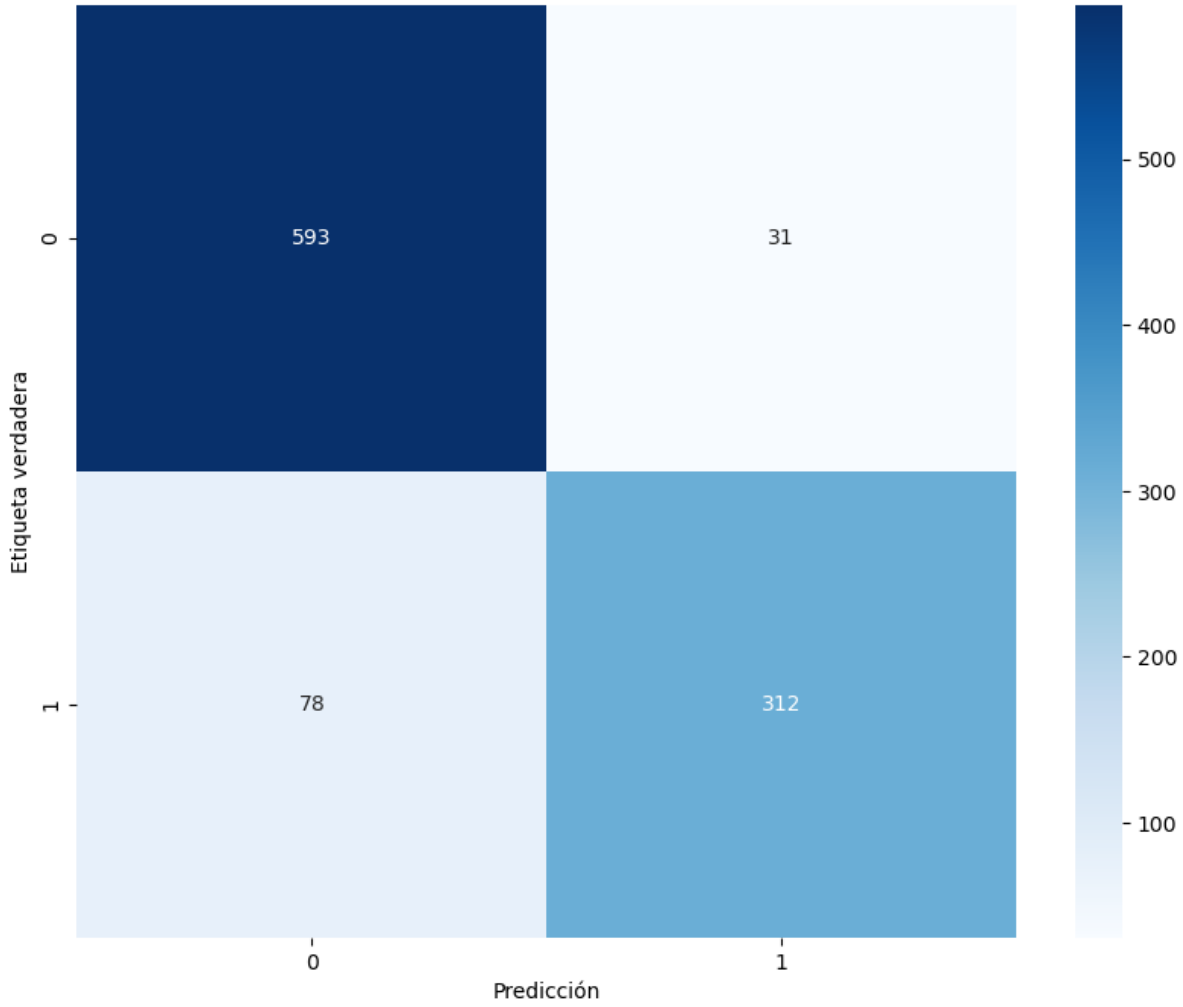


Ilustración 14 Matriz de confusión entre neumonia y normal

Las etiquetas '0' son NORMAL y las '1' son NEUMONÍA, por lo tanto:

- Hay 593 casos que son verdaderos positivos (NORMAL-NORMAL)
- Hay 31 casos que son verdaderos negativos (NORMAL-NEUMONÍA)
- Hay 312 casos que son falsos positivos (NEUMONÍA-NEUMONÍA)
- Hay 78 casos que son falsos negativos (NEUMONÍA-NORMAL)

Evaluación y predicción del modelo:

```
current_dir = os.getcwd()
i = 0
probs = []
plt.figure(figsize=(25, 8))
plt.subplots_adjust(bottom=0, left=.01, right=.99, top=.90, hspace=.35)
for file in os.listdir( current_dir + '/archive/chest_xray/val/NORMAL/'):
    plt.subplot(2, 7, i + 1)
    i = i+1
    img = load_img(current_dir + '/archive/chest_xray/val/NORMAL/' + file, target_size = (img_width, img_height), grayscale=True)
    x = img_to_array(img)
    img_batch = np.expand_dims(x, axis=0)
    prediction = model.predict(img_batch)
    plt.title(prediction)
    plt.title( 'Probabilidad Normal:' + str(prediction[0][0]) + '\nProbabilidad Pneumonia: ' + str(prediction[0][1]))
    plt.imshow(img, cmap='gray')
```

Ilustración 15 Código para mostrar la predicción de cada imagen de validación

- Se obtiene el directorio actual y se almacena en la variable "*current_dir*".
- Se utiliza un bucle para iterar a través de los archivos en el directorio de imágenes normales en el conjunto de validación.
- Dentro del bucle, se crea un *subplot* en la posición i + 1 en una cuadrícula de 2 filas y 7 columnas.
- Se carga la imagen utilizando la ruta la cual se construye concatenando el directorio actual, el directorio de imágenes normales en el conjunto de validación y el nombre del archivo del bucle.
- Se convierte la imagen en un array y se expande la dimensión para que tenga un tamaño de *batch* de imágenes.
- Se realiza la predicción del modelo utilizado, pasando el *batch* de imágenes como entrada y la almacenamos en la variable "*prediction***"**.
- Se establece el título del *subplot* con la probabilidad de predicción para la clase normal y la probabilidad de predicción para la clase de neumonía, obtenidas de la variable "*prediction*".
- Al final, se muestra la figura con todas las imágenes y sus predicciones.

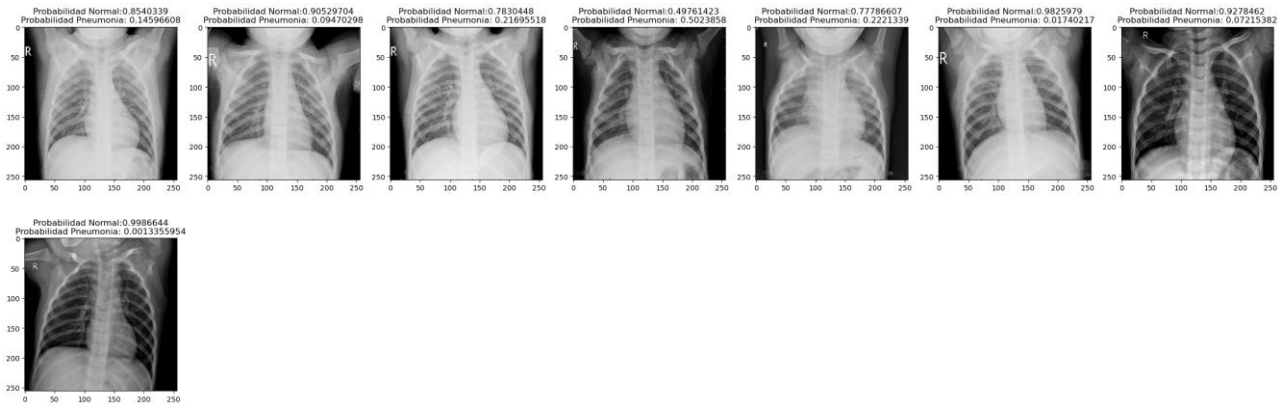


Ilustración 16 Predicción de cada imagen para la clase de normal

Se realiza el mismo proceso para las imágenes neumonía en el conjunto de validación:

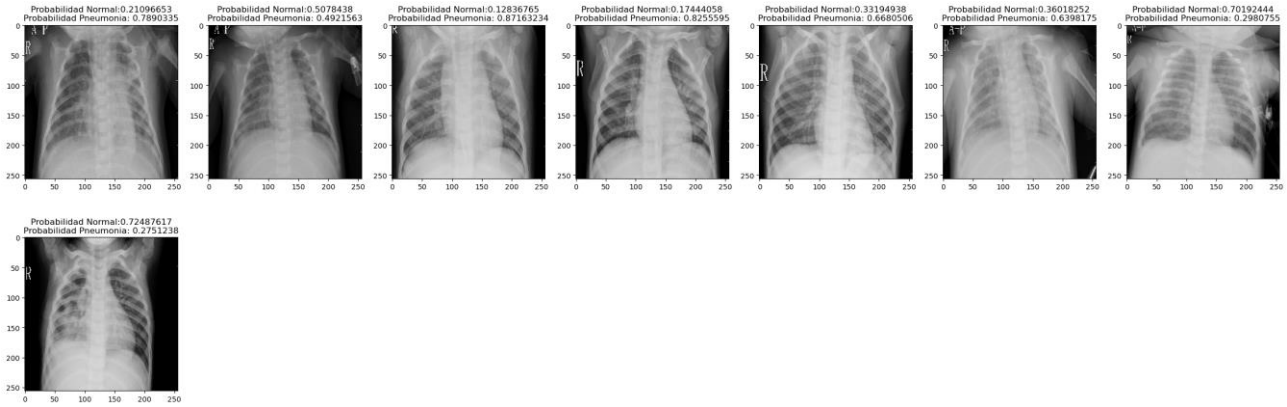


Ilustración 17 Predicción de cada imagen para la clase pneumonia

2.4 Pneumonia Bacteriana vs Pneumonia Virica vs normal

En este caso se trata de contrastar aquellas imágenes que pertenezcan a pulmones con una pulmonía vírica frente a aquellas que pertenecen a pulmones con una pulmonía bacteriana frente a aquellos que no poseen neumonía.

2.4.1 Preprocesamiento de datos

En esta segunda parte se realizan pasos similares a los realizados en la primera, pero realizando diferentes cambios.

Para esta parte se debe comenzar por subdividir en carpetas aquellas imágenes que pertenecen a neumonía bacteriana y vírica de cada uno de los directorios de *train*, *test* y *val* , para ello se elabora un código que cree los directorios y mueva las imágenes, para poder identificar que imagen pertenece a que clase, se usa el identificador que se encuentra en su nombre, aquellas que pertenecen a neumonía bacteriana contienen la palabra bacteria y aquellas que pertenecen a neumonía vírica contienen la palabra virus de manera que quedaría de la siguiente manera:

```python
# Especificar la ruta principal donde se crearán los directorios
ruta_principal = 'archive/chest_xray/chest_xray/train/'

# Crear el directorio "bacteria" en la ruta principal si no existe
ruta_bacteria = os.path.join(ruta_principal, 'PNEUMONIA_Bacteriana')
if not os.path.exists(ruta_bacteria):
    os.makedirs(ruta_bacteria)

# Crear el directorio "virus" en la ruta principal si no existe
ruta_virus = os.path.join(ruta_principal, 'PNEUMONIA_Virica')
if not os.path.exists(ruta_virus):
    os.makedirs(ruta_virus)

# Definir el directorio de la carpeta PNEUMONIA
pneumonia_dir = 'archive/chest_xray/chest_xray/train/PNEUMONIA/'

# Definir el directorio de la carpeta para las imágenes de bacteria
bacteria_dir = 'archive/chest_xray/chest_xray/train/PNEUMONIA_Bacteriana'

# Definir el directorio de la carpeta para las imágenes de virus
virus_dir = 'archive/chest_xray/chest_xray/train/PNEUMONIA_Virica'

# Iterar a través de los archivos en la carpeta PNEUMONIA
if os.path.exists('archive/chest_xray/chest_xray/train/PNEUMONIA'):
    shutil.rmtree('archive/chest_xray/chest_xray/train/PNEUMONIA')
    for filename in os.listdir(pneumonia_dir):
        # Si el nombre de archivo contiene la palabra "bacteria", copiar a la carpeta de bacteria
        if 'bacteria' in filename:
            shutil.copy(pneumonia_dir + filename, bacteria_dir)
        # Si el nombre de archivo contiene la palabra "virus", copiar a la carpeta de virus
        elif 'virus' in filename:
            shutil.copy(pneumonia_dir + filename, virus_dir)
    shutil.rmtree('archive/chest_xray/chest_xray/train/PNEUMONIA')
```

Ilustración 19 Creación de nuevas carpetas para las clases pneumonia vírica y pneumonia bacteriana

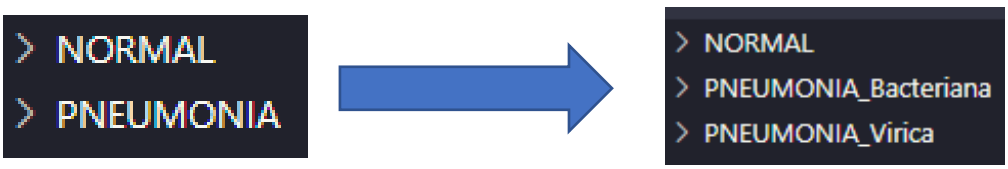


Ilustración 18 Transformación de las carpetas

Una vez realizado con cada una de las carpetas, se vuelve a tener el mismo problema de falta de datos para entrenar correctamente a la IA de manera que volvemos a tener que generar datos sintéticos de la misma manera que realizamos en la primera parte. Como resultado se obtiene la siguiente gráfica con los datos.

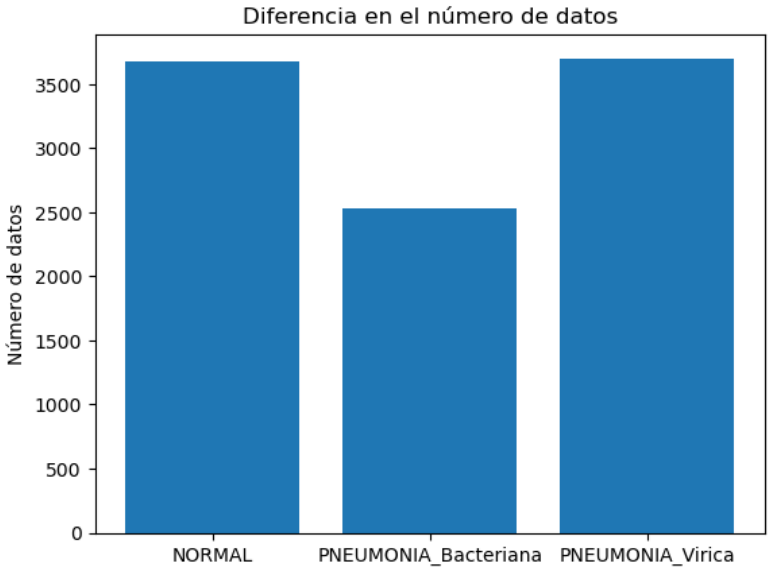


Ilustración 20 Gráfico de barras de la distribución de datos de las clases normal, pneumonia bacteriana y pneumonia virica

2.4.2 Arquitectura

La primera capa es una capa de normalización que normaliza los valores de píxeles de la imagen a un rango de 0 a 1.

Luego sigue una capa convolucional con 96 filtros de tamaño 11x11 y un paso (*stride*) de 4 píxeles. La capa de activación es *ReLU* como se indica la practica Luego, una capa de *pooling* que reduce la dimensión espacial de la salida.

Las siguientes tres capas son similares en estructura: una capa convolucional seguida de una capa de activación *ReLU* y una capa de *pooling MaxPooling*. La última capa de convolución tiene 256 filtros de tamaño 3x3. Después de la capa de *pooling* de la quinta capa convolucional, se agrega una capa completamente conectada de 4096 neuronas, seguida de una capa de activación *ReLU* y una capa de *dropout*. La capa de *dropout* apaga aleatoriamente algunas neuronas durante el entrenamiento, lo que ayuda a prevenir el sobreajuste en el modelo. Luego se agrega otra capa completamente conectada idéntica a la anterior, seguida de una capa de activación *ReLU* y una capa de *dropout*.

La última capa es la capa de salida que tiene tres neuronas, debió a la existencia de 3 clases en las que clasificar y utiliza la activación *softmax*, que normaliza los valores de salida a un rango de 0 a 1 y los interpreta como probabilidades.

Se usa una función de pérdida de entropía cruzada categórica (*categorical cross-entropy loss*) y un optimizador de descenso de gradiente estocástico (*Stochastic Gradient Descent o SGD*) con una tasa de aprendizaje (*learning rate*) de 0.0001.

```
model = Sequential()
normalization_layer = layers.experimental.preprocessing.Rescaling(1./255, input_shape=(img_height, img_width, 1))
model.add(normalization_layer)
# capa de convolución 1
model.add(Conv2D(filters=96, kernel_size=(11,11), strides=(4,4), padding='valid', input_shape=(200,200,1)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(3,3), strides=(2,2), padding='valid'))

# capa de convolución 2
model.add(Conv2D(filters=256, kernel_size=(5,5), strides=(1,1), padding='same'))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(3,3), strides=(2,2), padding='valid'))

# capa de convolución 3
model.add(Conv2D(filters=384, kernel_size=(3,3), strides=(1,1), padding='same'))
model.add(Activation('relu'))

# capa de convolución 4
model.add(Conv2D(filters=384, kernel_size=(3,3), strides=(1,1), padding='same'))
model.add(Activation('relu'))

# capa de convolución 5
model.add(Conv2D(filters=256, kernel_size=(3,3), strides=(1,1), padding='same'))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(3,3), strides=(2,2), padding='valid'))

# capa completamente conectada 1
model.add(Flatten())
model.add(Dense(4096, input_shape=(200*200,)))
model.add(Activation('relu'))
model.add(Dropout(0.4))

# capa completamente conectada 2
model.add(Dense(4096))
model.add(Activation('relu'))
model.add(Dropout(0.4))

# capa de salida
model.add(Dense(3))
model.add(Activation('softmax'))

# compilar el modelo
model.compile(loss='categorical_crossentropy', optimizer=SGD(learning_rate = 0.0001), metrics=['accuracy'])

stop_training_callback = StopTrainingCallback()
history = model.fit(
        train_generator_1,
        epochs = epochs,
        validation_data = test_generator_1,
        callbacks = [stop_training_callback],
)
```

Ilustración 21 Arquitectura para la red de la clasificación de neumonía bacteriana, vírica y normal.

Una vez entrenada la red mostramos las gráficas del *loss* y el *accuracy* por época. Las gráficas que obtenemos tras el entrenamiento son las siguientes:

Como se observa, se comienza con valores de pérdida altos y posteriormente va reduciéndose conforme las iteraciones avanzan, del mismo modo ocurre con la validación y el entrenamiento pero en sentido contrario, es decir, comienza con valores muy bajos y posteriormente aumenta, esto significa que la red aprende correctamente pues progresivamente clasifica de mejor manera.

Al terminar el entrenamiento al igual que la anterior parte, se dibuja la matriz de confusión, en este caso con las 3 clases existentes, donde se puede observar la clasificación de los resultados.

Tal como se ilustra en la matriz de confusión la amplia mayoría de casos se encuentran clasificados correctamente, es decir, la salida obtenida por la red es idéntica a la salida esperada por los datos.

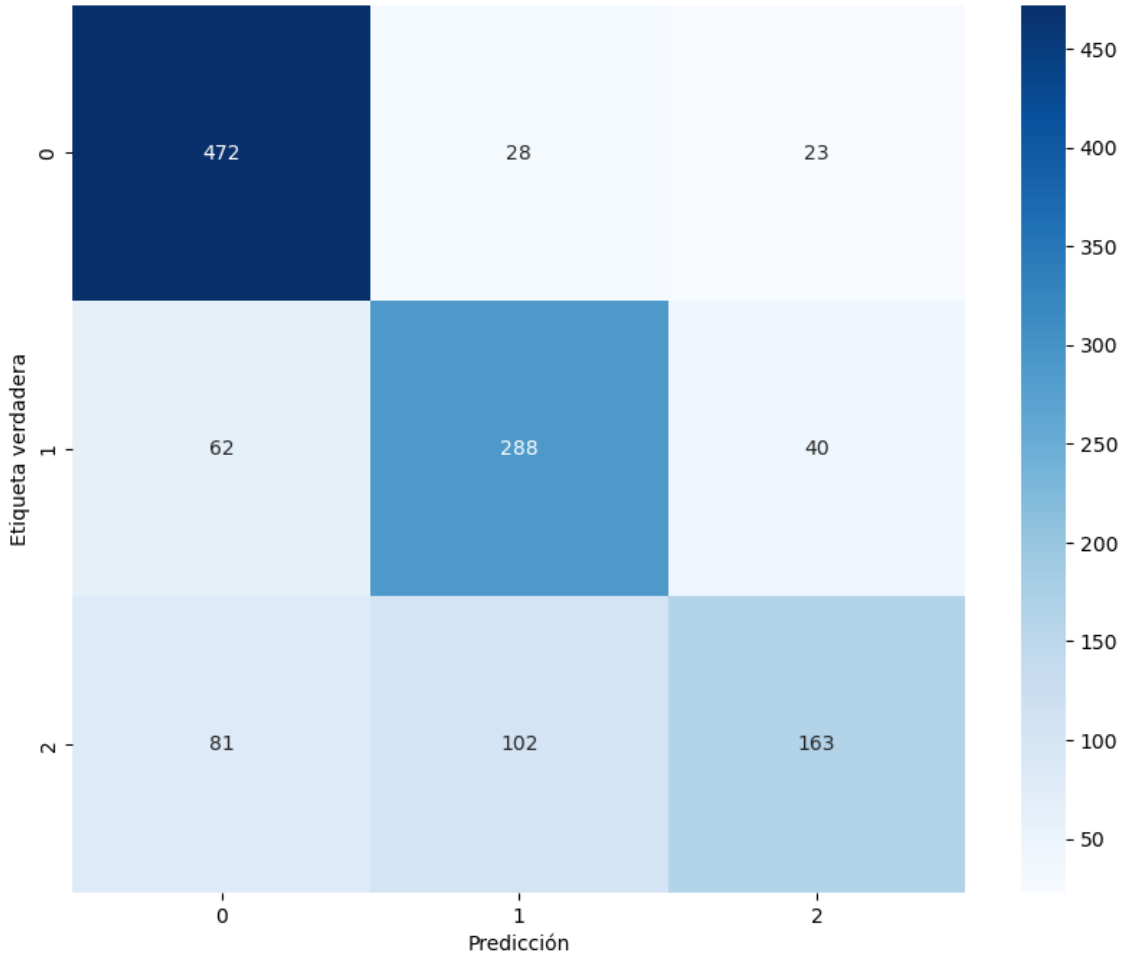


Ilustración 22 Matriz de confusión de normal, pneumonia vírica, pneumonía bacteriana

Las etiquetas '0' son NORMAL, las '1' son NEUMONÍA BACTERIANA y '2' son NEUMONÍA VIRICA, por lo tanto:

- Hay 472 casos que son verdaderos positivos (NORMAL-NORMAL)
- Hay 28 casos que son verdaderos negativos (NORMAL-NEUMONÍA BACTERIANA)
- Hay 23 casos que son falsos positivos (NORMAL-NEUMONÍA VIRICA)
- Hay 62 casos que son falsos negativos (NEUMONÍA BACTERIANA - NORMAL)
- Hay 288 casos que son verdaderos positivos (NEUMONÍA BACTERIANA - NEUMONÍA BACTERIANA)
- Hay 40 casos que son falsos positivos (NEUMONÍA BACTERIANA - NEUMONÍA VIRICA)
- Hay 81 casos que son falsos negativos (NEUMONÍA VIRICA - NORMAL)
- Hay 102 casos que son falsos positivos (NEUMONÍA VIRICA - NEUMONÍA BACTERIANA)

- Hay 163 casos que son verdaderos positivos (NEUMONÍA VIRICA - NEUMONÍA VIRICA)

Evaluación y predicción del modelo:

```
current_dir = os.getcwd()
i = 0
probs = []
plt.figure(figsize=(25, 8))
plt.subplots_adjust(bottom=0, left=.01, right=.99, top=.90, hspace=.35)
for file in os.listdir( current_dir + '/archive/chest_xray/chest_xray/val/NORMAL/'):
    plt.subplot(2, 7, i + 1)
    i = i+1
    img = load_img(current_dir + '/archive/chest_xray/chest_xray/val/NORMAL/' + file, target_size = (img_width, img_height), grayscale=True)
    x = img_to_array(img)
    img_batch = np.expand_dims(x, axis=0)
    prediction = model.predict(img_batch)
    plt.title(prediction)
    plt.title( 'Probabilidad Normal:' + str(prediction[0][0]) + '\nProbabilidad Pneumonia Bacteriana: ' + str(prediction[0][1]) + '\nProbabilidad Pneumonia Vírica: ' + str(prediction[0][2]))
    plt.imshow(img, cmap='gray')
```

Ilustración 23 Código de la evaluación de normal, pneumonía víricia y pneumonía bacteriana

Una vez entrenada la red se procede a clasificar las distintas imágenes presentes en validación.

Normal:

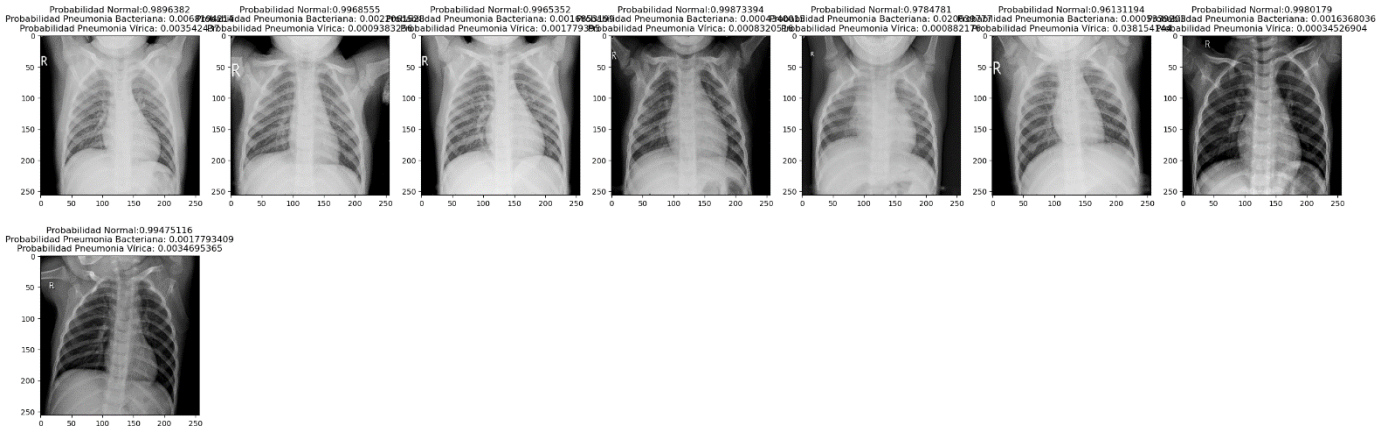


Ilustración 24 Predicción de normal para la parte 2

Neumonía Bacteriana:

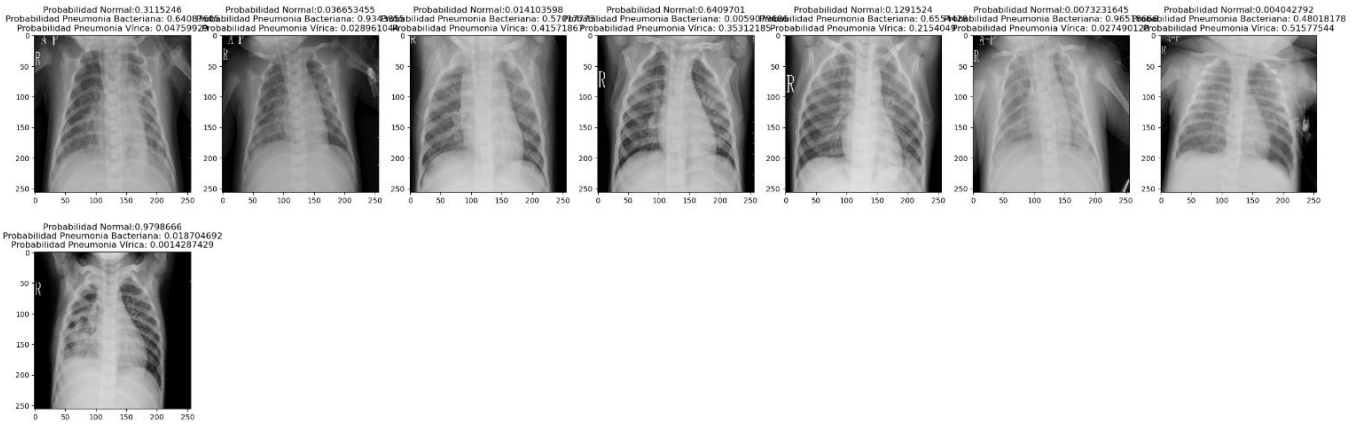


Ilustración 25 Predicción de neumonía bacteriana

Neumonía Vírica:

Es importante destacar que se han generado imágenes de validación en este apartado pues no existían, estas imágenes generadas provienen de una modificación de aquellas usadas para el entrenamiento de manera que resultan nuevas para la red.

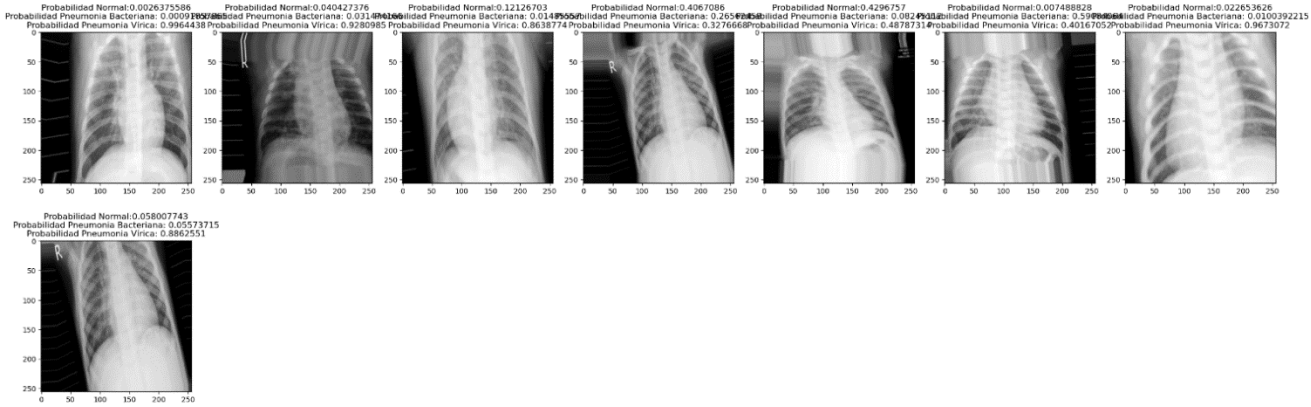


Ilustración 26 Predicción de neumonía Vírica

2.5 Cuestiones:

En este apartado se resuelven las diferentes cuestiones planteadas en el enunciado del laboratorio.

2.5.1 Cuestión 1:

2.5.1.1 Practica 1, parte 1:

"Explica las funciones y el algoritmo para convertir las imágenes a matrices. ¿Qué tamaños tienen las imágenes? ¿Has normalizado? ¿Cómo y por qué?"

El *dataset* de imágenes está dividido en tres grupos (entrenamiento, validación y test) y los tres son convertidos en matrices mediante las funciones de *Keras* para poder utilizarlos.

Las imágenes que se han utilizado tienen un tamaño especificado mediante las variables '*img_width* e *img_height*'. Se han normalizado utilizando una capa de normalización en la que se realiza un rescaldado de cada píxel para que tenga un valor entre 0 y 1, además de hacer que todas las imágenes se encuentren en el tamaño indicado, es decir, 256 x 256.

Se normalizan las imágenes en esa capa para facilitar el entrenamiento del modelo y mejorar su capacidad de generalización. También evita problemas de saturación en las funciones de

activación.

```
normalization_layer = layers.experimental.preprocessing.Rescaling(1./255, input_shape=(img_height, img_width, 1))
```

Ilustración 27 Código de normalización de los datos

Estas funciones tienen los siguientes parámetros:

- '*image_dataset_from_directory*':
  - Directorio que contiene las imágenes
  - Tamaño al que se van a redimensionar las imágenes
  - *batch_size*: Es el número de imágenes que se utilizan en cada iteración del entrenamiento del modelo. En este caso, se utilizan lotes de 64 imágenes durante el entrenamiento y validación, y lotes de 32 imágenes durante la prueba.
  - El modo de color en el que cargan las imágenes. En este caso se cargan en escala de grises.
  - *seed*: Es una semilla utilizada para la aleatorización en la generación de datos.
  - *label_mode*: Es el modo en que se generan las etiquetas para las clases de las imágenes.
- '*flow_from_directory*':
  - Directorio que contiene las imágenes
  - Tamaño al que se van a redimensionar las imágenes
  - *batch_size*: Es el número de imágenes que se utilizan en cada iteración del entrenamiento del modelo. En este caso, se utilizan lotes de 64 imágenes durante el entrenamiento y validación, y lotes de 32 imágenes durante la prueba.
  - El modo de color en el que cargan las imágenes. En este caso se cargan en escala de grises.
  - *seed*: Es una semilla utilizada para la aleatorización en la generación de datos.
  - *label_mode*: Es el modo en que se generan las etiquetas para las clases de las imágenes.

- *shuffle*: Es un parámetro utilizado para indicar si se deben mezclar los datos en el conjunto de prueba. En este caso, se establece en False para mantener el orden de las imágenes en el conjunto de prueba.
-

2.5.1.2 Practica 1, parte 2:

Se sigue el mismo procedimiento que en la primera parte, es decir, se añade una primera capa de normalización con el mismo fin que en la primera parte.

2.5.2 Cuestión 2:

"Entrena una red para cada clasificación usando como criterio de parada un loss ≤ 0,2. Recoge en la memoria los valores de loss y accuracy tanto del conjunto de entrenamiento como del de validación de los distintos experimentos que hayas llevado a cabo indicando la arquitectura de cada una de las redes empleadas.

Dibuja la arquitectura de red que mejor clasifica y pinta, para esa red, la gráfica de variación del error (loss) en función de las epochs, para los conjuntos de entrenamiento y validación".

2.5.2.1 Practica 1, parte 1:

A lo largo del desarrollo de la práctica se han utilizado múltiples arquitecturas con diferentes parámetros, se han probado múltiples variaciones, pero únicamente se mostrarán las más significativas.

Es necesario aclarar que debido al coste computacional en tiempo y recursos que se necesita para entrenar las distintas arquitecturas se han llegado a los mejores resultados posibles con los recursos disponibles, si se tuviese un mejor hardware se podrían lograr resultados más efectivos.

Como se ha enseñado previamente, para lograr detener el entrenamiento cuando se llega al *loss* de 0,2 se introduce una función de parada:

```
class StopTrainingCallback(tf.keras.callbacks.Callback):
    def on_epoch_end(self, epoch, logs={}):
        if logs.get('loss') < 0.2:
            print('\nSe ha alcanzado el valor de pérdida deseado.')
            self.model.stop_training = True
```

Ilustración 28 Función para detener el entrenamiento cuando el loss baja de 0,2

Una de las primeras arquitecturas que se probó contaba con 3 capas de convolución en la que se realizaba un *MaxPooling* por cada capa convolucional y posteriormente se añadía capas densas con diversos filtros, donde no se normalizaban las imágenes, se trataban como imágenes en RGB, tenían un tamaño pequeño de 150 x 150 y se usaba *Adam* como optimizador.

Tras trabajar con este modelo y realizar variaciones nos percatamos de varios fallos:

- Las imágenes necesitaban ser normalizadas para un entrenamiento eficaz
- Las imágenes eran tratadas como RGB y debía ser tratadas como escala de grises
- Se necesitaba más capas con más filtros para obtener más características de cada imagen

- Adam no era un optimizador tan efectivo como SGD

```
model = Sequential()

# Capas convolucionales
model.add(Conv2D(32, kernel_size=(3, 3), activation='relu', input_shape=(img_width, img_height, 3)))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Conv2D(64, kernel_size=(3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Conv2D(128, kernel_size=(3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

# Capas densas
model.add(Flatten())
model.add(Dense(256, activation='relu'))
model.add(Dropout(0.3))
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.3))
model.add(Dense(2, activation='softmax'))


model.compile(loss='categorical_crossentropy', optimizer=Adam(learning_rate=0.0001), metrics=['accuracy'])

stop_training_callback = StopTrainingCallback()

history = model.fit(train_generator_1,
                    steps_per_epoch=batch_size,
                    epochs= epochs,
                    validation_data=val_generator_1,
                    validation_steps=batch_size,
                    callbacks=[stop_training_callback])
```

Ilustración 30 Arquitectura antigua para la clasificación de Pneumonia bacteriana y normal

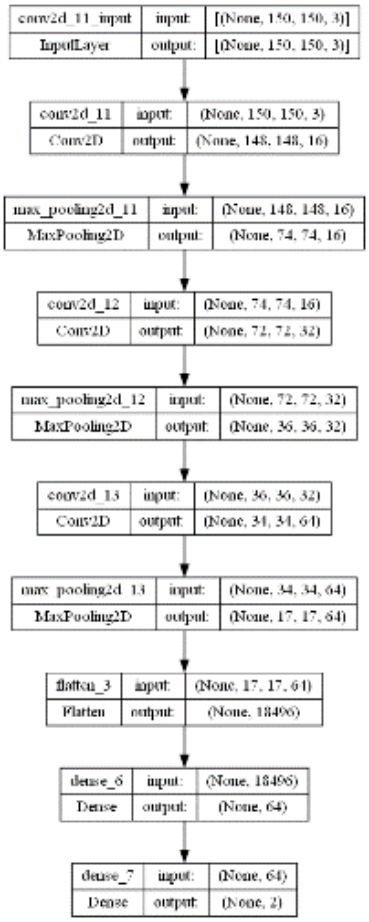


Ilustración 29 Arquitectura antigua para clasificar normal y neumonia

```
Epoch 35/200
16/16 [==============================] - 6s 392ms/step - loss: 0.2553 - accuracy: 0.8945 - val_loss: 0.3677 - val_accuracy: 0.8477
Epoch 36/200
16/16 [==============================] - 6s 386ms/step - loss: 0.3416 - accuracy: 0.8516 - val_loss: 0.3218 - val_accuracy: 0.8789
Epoch 37/200
16/16 [==============================] - 6s 393ms/step - loss: 0.2970 - accuracy: 0.8750 - val_loss: 0.3246 - val_accuracy: 0.8750
Epoch 38/200
16/16 [==============================] - 6s 404ms/step - loss: 0.2472 - accuracy: 0.8828 - val_loss: 0.2946 - val_accuracy: 0.8750
Epoch 39/200
16/16 [==============================] - 6s 397ms/step - loss: 0.2357 - accuracy: 0.8789 - val_loss: 0.3627 - val_accuracy: 0.8477
Epoch 40/200
16/16 [==============================] - 6s 387ms/step - loss: 0.2358 - accuracy: 0.8984 - val_loss: 0.2634 - val_accuracy: 0.8828
Epoch 41/200
16/16 [==============================] - 6s 383ms/step - loss: 0.2534 - accuracy: 0.9102 - val_loss: 0.3365 - val_accuracy: 0.8359
Epoch 42/200
16/16 [==============================] - 7s 418ms/step - loss: 0.2584 - accuracy: 0.8867 - val_loss: 0.4029 - val_accuracy: 0.8438
Epoch 43/200
16/16 [==============================] - ETA: 0s - loss: 0.1908 - accuracy: 0.9180
Se ha alcanzado el valor de pérdida deseado.
16/16 [==============================] - 6s 387ms/step - loss: 0.1908 - accuracy: 0.9180 - val_loss: 0.3887 - val_accuracy: 0.8594
```

Ilustración 31 Valores de accuracy y loss de la antigua arquitectura para clasificar normal y neumonia

Tras esto se probaron con más arquitecturas, más complejas:

Aquí se tienen el *loss* y *accuracy* de los conjuntos de entrenamiento y validación para la mejor red creada, que cuenta con las siguientes características:

batch_size = 64

epochs = 1000

Optimizador = *SGD*

Tasa de aprendizaje = 0.01

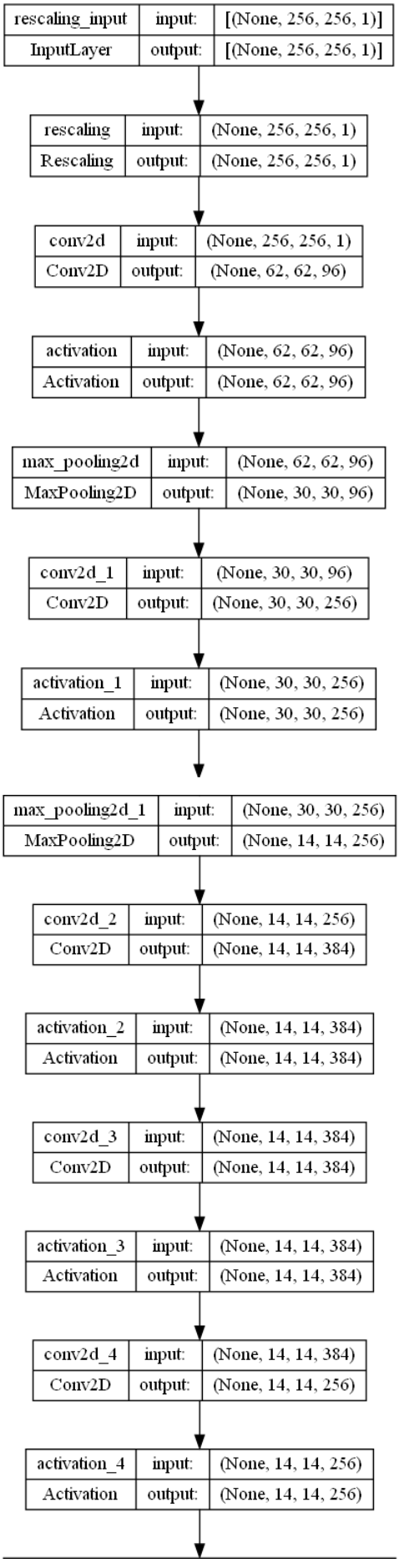
Tamaño de la imagen = 256 x 256

Se introducen un total de 1000 épocas en la red como un número fijo, pues el entrenamiento se detendrá una vez llegue al valor de pérdida deseado es decir 0.2 de manera que así se garantiza que se llegue a ese valor.

```
Epoch 1/1000
183/183 [==============================] - 661s 4s/step - loss: 0.6241 - accuracy: 0.6667 - val_loss: 0.6411 - val_accuracy: 0.6154
Epoch 2/1000
183/183 [==============================] - 641s 3s/step - loss: 0.5884 - accuracy: 0.6715 - val_loss: 0.5929 - val_accuracy: 0.6154
Epoch 3/1000
183/183 [==============================] - 599s 3s/step - loss: 0.4946 - accuracy: 0.7526 - val_loss: 0.3960 - val_accuracy: 0.8018
Epoch 4/1000
183/183 [==============================] - 559s 3s/step - loss: 0.2434 - accuracy: 0.9032 - val_loss: 0.4895 - val_accuracy: 0.7880
Epoch 5/1000
183/183 [==============================] - ETA: 0s - loss: 0.1770 - accuracy: 0.9300
Se ha alcanzado el valor de pérdida deseado.
183/183 [==============================] - 576s 3s/step - loss: 0.1770 - accuracy: 0.9300 - val_loss: 0.2861 - val_accuracy: 0.8925
```

Ilustración 32 Resultados de la red para tamaño de batch 64, épocas 1000, optimizador SGD, tasa de aprendizaje 0,01 y tamaño de imagen 256 x 256

Ilustración 33 Gráficas del Loss y Accuracy para Validation y Train


rescaling_input | input: | [(None, 256, 256, 1)]
InputLayer | output: | [(None, 256, 256, 1)]
rescaling | input: | (None, 256, 256, 1)
Rescaling | output: | (None, 256, 256, 1)
conv2d | input: | (None, 256, 256, 1)
Conv2D | output: | (None, 62, 62, 96)
activation | input: | (None, 62, 62, 96)
Activation | output: | (None, 62, 62, 96)
max_pooling2d | input: | (None, 62, 62, 96)
MaxPooling2D | output: | (None, 30, 30, 96)
conv2d_1 | input: | (None, 30, 30, 96)
Conv2D | output: | (None, 30, 30, 256)
activation_1 | input: | (None, 30, 30, 256)
Activation | output: | (None, 30, 30, 256)
max_pooling2d_1 | input: | (None, 30, 30, 256)
MaxPooling2D | output: | (None, 14, 14, 256)
conv2d_2 | input: | (None, 14, 14, 256)
Conv2D | output: | (None, 14, 14, 384)
activation_2 | input: | (None, 14, 14, 384)
Activation | output: | (None, 14, 14, 384)
conv2d_3 | input: | (None, 14, 14, 384)
Conv2D | output: | (None, 14, 14, 384)
activation_3 | input: | (None, 14, 14, 384)
Activation | output: | (None, 14, 14, 384)
conv2d_4 | input: | (None, 14, 14, 384)
Conv2D | output: | (None, 14, 14, 256)
activation_4 | input: | (None, 14, 14, 256)
Activation | output: | (None, 14, 14, 256)

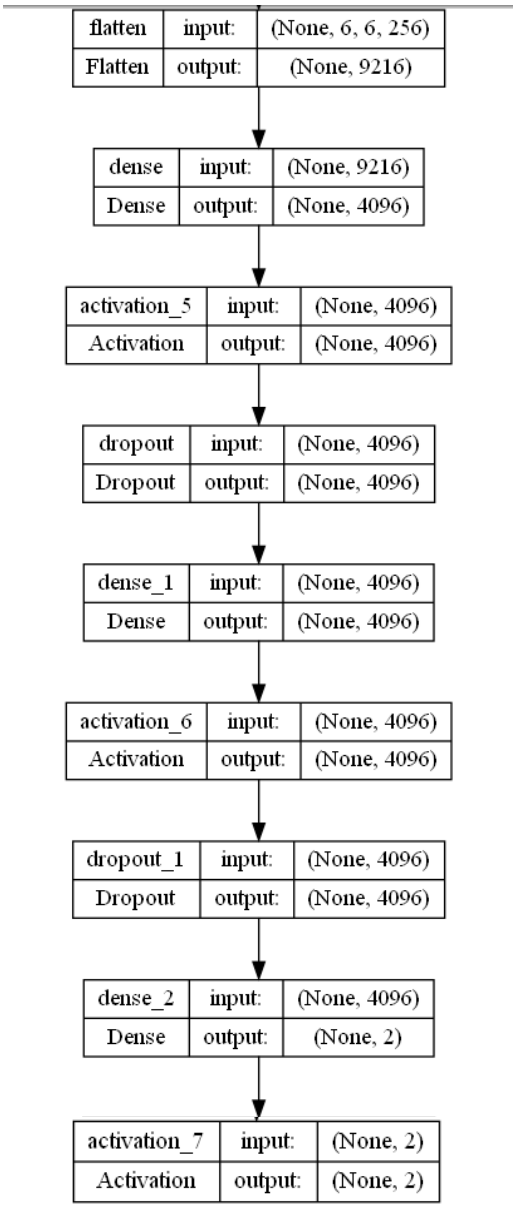


Ilustración 34 Arquitectura final para la clasificación de normal y neumonía

Se han realizado bastantes experimentos con la arquitectura final debido a su potencial, los elementos modificados eran número de imágenes, tamaño de estas, el tamaño del lote (*batch size*) y diferentes coeficientes de aprendizajes. Lo primero que se observó fue el no poder modificar el coeficiente de aprendizaje de 0.01 ya que con un mínimo cambio sufría muy rápido de *underfitting* o *overfitting.* El tamaño del lote también se mantuvo en 64 para no pasarse de tiempo por interacción. Si era menor, o mayor entrenaba demasiado rápido y no le daba tiempo a generalizar al modelo. El siguiente ejemplo fue uno de los más significativos, ya que se observa que las imágenes de neumonía el modelo las clasificaba bien, pero, al igualar el número de imágenes en ambas clases, generando más tampoco clasificaba bien del todo. En ese momento se descubrió que había mucho ruido en esas imágenes y por eso en el modelo definitivo recibe el doble de imágenes de la clase normal que de neumonía.

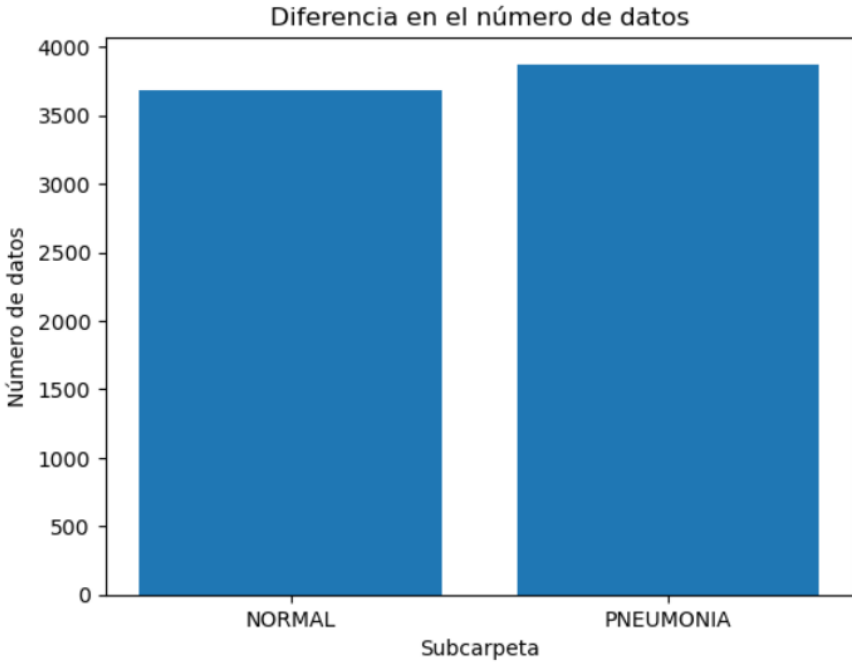


Ilustración 35 Diagrama de barras experimento

```
Epoch 1/1000
119/119 [==============================] - 419s 4s/step - loss: 0.6896 - accuracy: 0.5976 - val_loss: 0.6869 - val_accuracy: 0.5012
Epoch 2/1000
119/119 [==============================] - 423s 4s/step - loss: 0.6791 - accuracy: 0.6910 - val_loss: 0.6782 - val_accuracy: 0.4842
Epoch 3/1000
119/119 [==============================] - 438s 4s/step - loss: 0.6515 - accuracy: 0.6709 - val_loss: 0.6222 - val_accuracy: 0.7027
Epoch 4/1000
119/119 [==============================] - 358s 3s/step - loss: 0.5869 - accuracy: 0.6985 - val_loss: 0.4451 - val_accuracy: 0.8471
Epoch 5/1000
119/119 [==============================] - 354s 3s/step - loss: 0.3941 - accuracy: 0.8337 - val_loss: 0.9359 - val_accuracy: 0.5680
Epoch 6/1000
119/119 [==============================] - 354s 3s/step - loss: 0.2488 - accuracy: 0.9002 - val_loss: 0.2952 - val_accuracy: 0.8859
Epoch 7/1000
119/119 [==============================] - ETA: 0s - loss: 0.1960 - accuracy: 0.9195
Se ha alcanzado el valor de pérdida deseado.
119/119 [==============================] - 355s 3s/step - loss: 0.1960 - accuracy: 0.9195 - val_loss: 0.5451 - val_accuracy: 0.8010
```

Ilustración 36 Valores de entrenamiento experimento

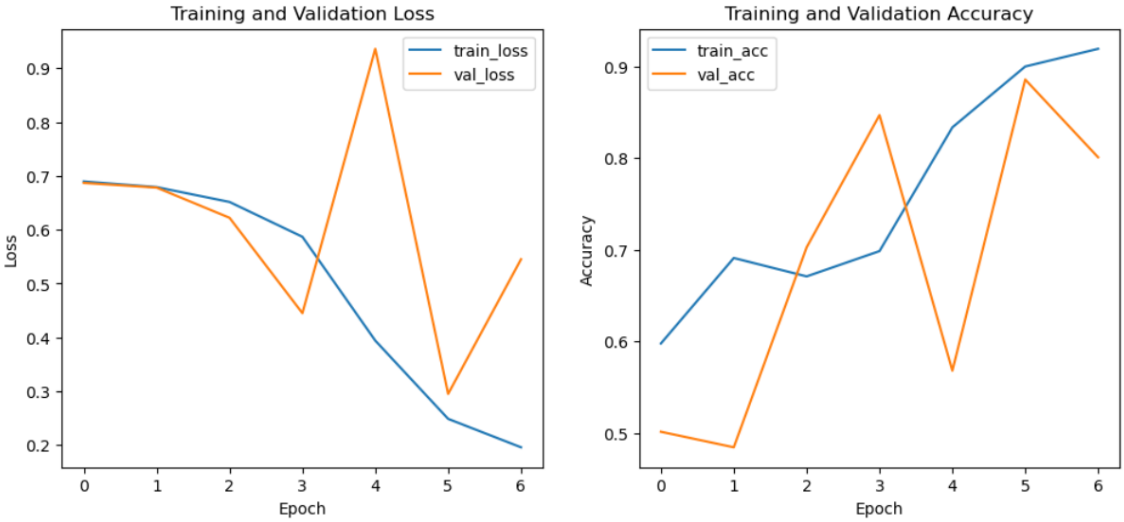


Ilustración 37 Gráficas de entrenamiento experimento 1

```
1/1 [==============================] - 0s 241ms/step - loss: 0.7847 - accuracy: 0.6250
Test accuracy: 0.625
Test loss: 0.7847040891647339
```

Ilustración 38 Datos de precisión en la prueba experimento 1

Se puede observar un entrenamiento correcto, pero con una validación con muchos picos además de poca generalización en la prueba al tener mucho error y fallar mucho sobre todo en las imágenes de normal como podemos observar a continuación.

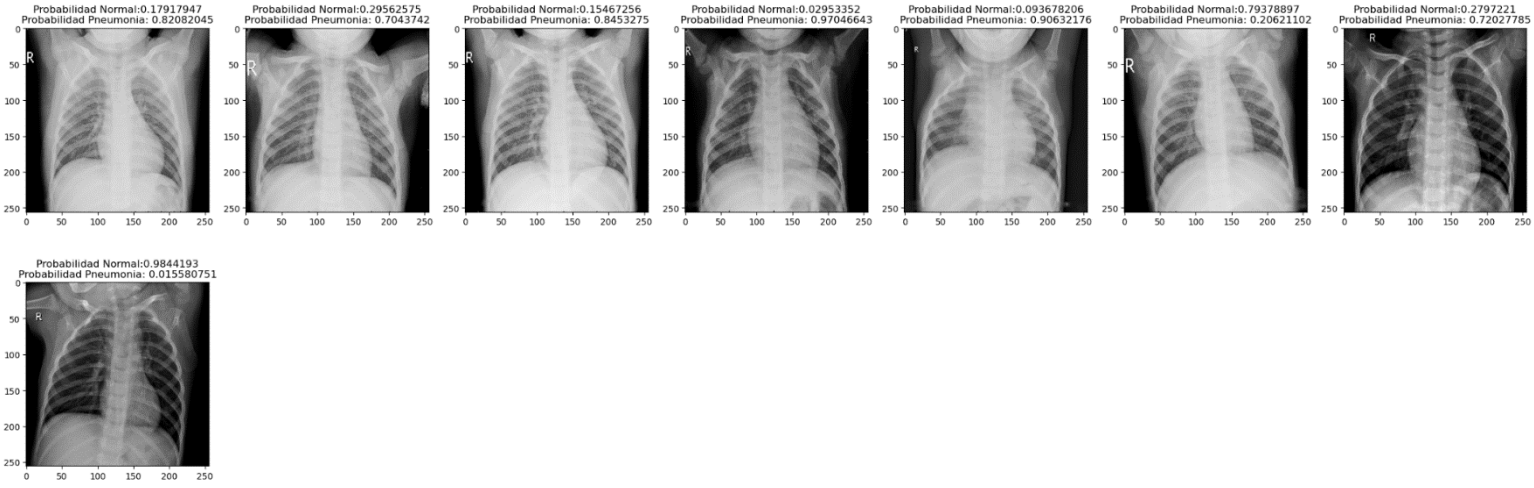


Ilustración 39 Resultados de predicción normal experimento 1

2.5.2.2 Practica 1, parte 2:

En esta parte de la práctica se hace uso de la misma arquitectura que en la parte anterior, debido a que es una arquitectura funcional a partir de la que ajustamos los hiperparámetros para adaptarla a los nuevos datos.

```
model = Sequential()

# Capa de entrada y normalización
normalization_layer = layers.experimental.preprocessing.Rescaling(1./255, input_shape=(img_height, img_width, 1))
model.add(normalization_layer)

# Capa de convolución 1
model.add(Conv2D(filters=16, kernel_size=(3,3), padding='same'))
model.add(BatchNormalization())
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2,2)))

model.add(Conv2D(filters=16, kernel_size=(3,3), padding='same'))
model.add(BatchNormalization())
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2,2)))

# Capa de convolución 2
model.add(Conv2D(filters=32, kernel_size=(3,3), padding='same'))
model.add(BatchNormalization())
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2,2)))

model.add(Conv2D(filters=32, kernel_size=(3,3), padding='same'))
model.add(BatchNormalization())
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2,2)))

# Capa de convolución 3
model.add(Conv2D(filters=64, kernel_size=(3,3), padding='same'))
model.add(BatchNormalization())
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2,2)))

# Capa completamente conectada 1
model.add(Flatten())
model.add(Dense(64))
model.add(BatchNormalization())
model.add(Activation('relu'))
model.add(Dropout(0.5))

# Capa de salida
model.add(Dense(3))
model.add(Activation('softmax'))

# Compilar el modelo
model.compile(loss='categorical_crossentropy', optimizer=Adam(learning_rate=0.0001), metrics=['accuracy'])
```

Ilustración 40:Arquitectura antigua de la Practica 1, parte 2

En primera instancia tratamos de reducir en número de filtros para procesar las imágenes de manera más rápida además de trabajar con las imágenes en escala de grises y usar Adam como optimizador, resultando en la siguiente arquitectura:

No obstante, este enfoque no era el correcto pues no lograba que la red aprendiese, debido a que:

- Debíamos tratar las imágenes como RGB
- Debíamos añadir más filtros para obtener más características de las imágenes
- Adam no era el mejor optimizador

Realizando estos cambios se obtuvo la siguiente red:

epochs = 1000

Optimizador = *SGD*

Tasa de aprendizaje = 0.0001

Tamaño de la imagen = 256 x 256

```
model = Sequential()
normalization_layer = layers.experimental.preprocessing.Rescaling(1./255, input_shape=(img_height, img_width, 1))
model.add(normalization_layer)
# capa de convolución 1
model.add(Conv2D(filters=96, kernel_size=(11,11), strides=(4,4), padding='valid', input_shape=(200,200,1)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(3,3), strides=(2,2), padding='valid'))

# capa de convolución 2
model.add(Conv2D(filters=256, kernel_size=(5,5), strides=(1,1), padding='same'))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(3,3), strides=(2,2), padding='valid'))

# capa de convolución 3
model.add(Conv2D(filters=384, kernel_size=(3,3), strides=(1,1), padding='same'))
model.add(Activation('relu'))

# capa de convolución 4
model.add(Conv2D(filters=384, kernel_size=(3,3), strides=(1,1), padding='same'))
model.add(Activation('relu'))

# capa de convolución 5
model.add(Conv2D(filters=256, kernel_size=(3,3), strides=(1,1), padding='same'))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(3,3), strides=(2,2), padding='valid'))

# capa completamente conectada 1
model.add(Flatten())
model.add(Dense(4096, input_shape=(200*200,)))
model.add(Activation('relu'))
model.add(Dropout(0.4))

# capa completamente conectada 2
model.add(Dense(4096))
model.add(Activation('relu'))
model.add(Dropout(0.4))

# capa de salida
model.add(Dense(3))
model.add(Activation('softmax'))

# compilar el modelo
model.compile(loss='categorical_crossentropy', optimizer=SGD(learning_rate = 0.0001), metrics=['accuracy'])

stop_training_callback = StopTrainingCallback()
history = model.fit(
        train_generator_1,
        epochs = epochs,
        validation_data = test_generator_1,
        callbacks = [stop_training_callback],
)
```

Ilustración 42 Código de implementación de la arquitectura de clasificación de las clases normal, pneumonía bacteriana y pneumonia vírica

```
Epoch 15/1000
182/182 [==============================] - 313s 2s/step - loss: 0.2464 - accuracy: 0.9049 - val_loss: 0.6922 - val_accuracy: 0.7625
Epoch 16/1000
182/182 [==============================] - 312s 2s/step - loss: 0.2281 - accuracy: 0.9111 - val_loss: 0.7760 - val_accuracy: 0.7315
Epoch 17/1000
182/182 [==============================] - 327s 2s/step - loss: 0.2158 - accuracy: 0.9204 - val_loss: 0.7509 - val_accuracy: 0.7577
Epoch 18/1000
182/182 [==============================] - 386s 2s/step - loss: 0.2014 - accuracy: 0.9266 - val_loss: 0.7593 - val_accuracy: 0.7403
Epoch 19/1000
182/182 [==============================] - ETA: 0s - loss: 0.1923 - accuracy: 0.9261
Se ha alcanzado el valor de pérdida deseado.
182/182 [==============================] - 363s 2s/step - loss: 0.1923 - accuracy: 0.9261 - val_loss: 0.8217 - val_accuracy: 0.7331
```

Ilustración 41 Resultados de la arquitectura de clasificación de las clases normal, neumonía bacteriana y neumonía vírica

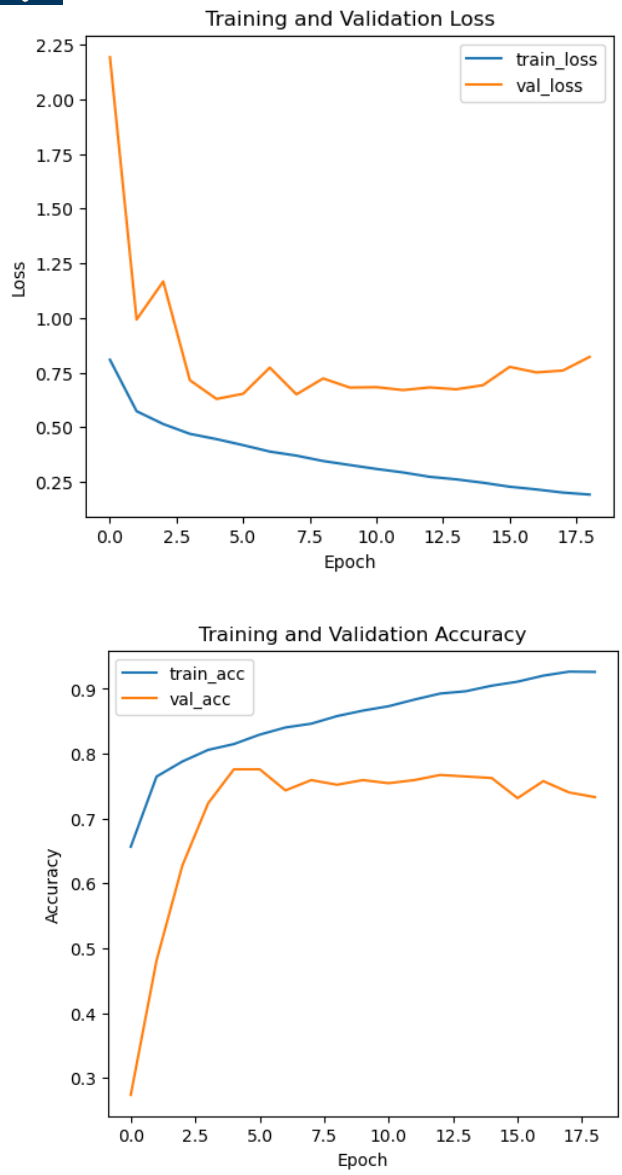
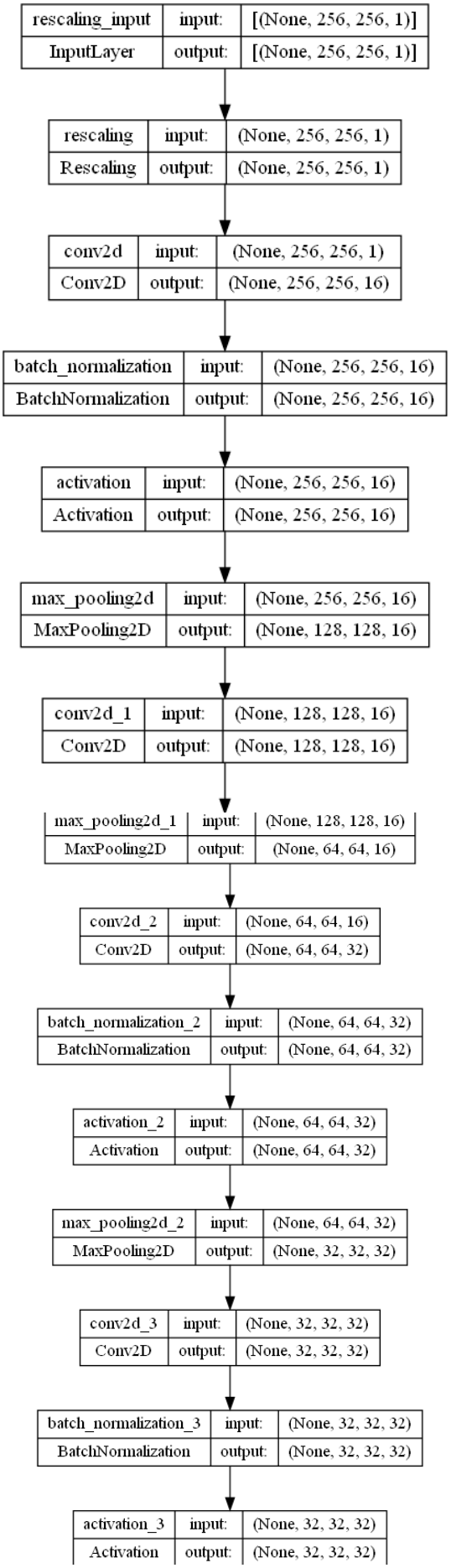


Ilustración 43 Gráficas del Loss y Accuracy para Validation y Train



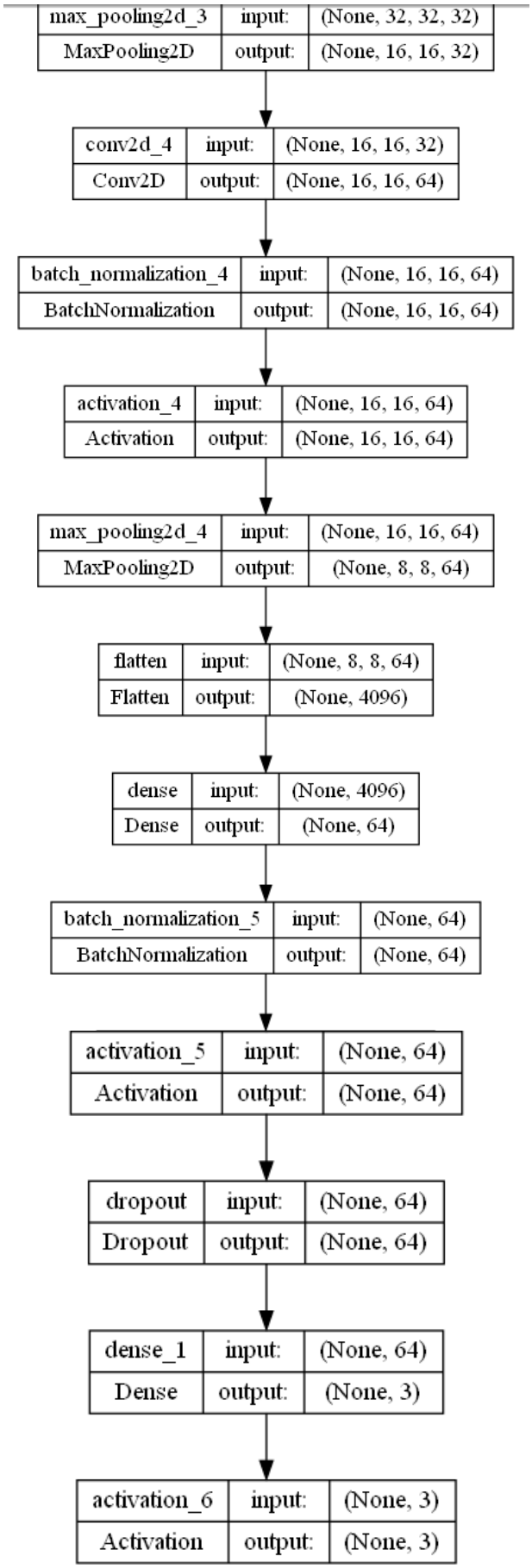


Ilustración 44 Arquitectura de clasificación de las clases normal, pneumonía bacteriana y pneumonia vírica

2.5.3 Cuestión 3:

"¿Qué es más fácil: distinguir entre dos tipos de neumonía o distinguir si el paciente está sano o no? ¿Por qué?"

Es más fácil distinguir entre saber si un paciente está sano o no, a distinguir entre los distintos tipos de neumonía.

Esto es debido a que cuando se distingue entre 2 tipos de neumonía se ha de establecer una red neuronal más compleja que extraiga más características y que cuente con más capas y más filtros por capa, todo ello con el objetivo de poder diferenciar entre 2 tipos de neumonía que son similares entre sí, esto aumenta el coste de computación y el tiempo necesario para entrenarla además de contar con un preprocesamiento más extenso y complejo.

Mientras que al tener que distinguir entre si un paciente está sano o no únicamente se tiene que diferenciar entre 2 clases con características completamente diferentes y se pueden agrupar las 2 clases de neumonía (vírica y bacteriana) en 1, haciendo más sencillo el preprocesamiento de datos y la red creada para poder clasificar.

2.5.4 Cuestión 4:

"¿En caso de no tener suficientes imágenes, que técnicas usarías? Explícalas con detalle "

Si no se tienen suficientes imágenes para entrenar un modelo CNN, se pueden utilizar diversas técnicas para aumentar el tamaño del conjunto de datos de entrenamiento. Algunas de estas técnicas son las siguientes:

1. **Aumento de datos:** el aumento de datos es una técnica que consiste en aplicar transformaciones aleatorias a las imágenes existentes, como rotaciones, recortes, volteos y ajustes de brillo. Esto aumenta la cantidad de datos de entrenamiento disponibles para el modelo y ayuda a evitar el sobreajuste (*overfitting*) del modelo a los datos existentes.
2. **Transferencia de aprendizaje:** la transferencia de aprendizaje es una técnica en la que se utiliza un modelo pre-entrenado en un conjunto de datos grande y luego se ajusta (*fine-tuning*) para una tarea específica con un conjunto de datos más pequeño. Esta técnica permite aprovechar el conocimiento aprendido por el modelo pre-entrenado en la tarea específica.
3. **Generación de imágenes sintéticas:** la generación de imágenes sintéticas es una técnica en la que se generan imágenes sintéticas a partir de datos existentes. Por ejemplo, se pueden utilizar técnicas de síntesis de texturas para crear nuevas imágenes a partir de las imágenes existentes. Estas nuevas imágenes se pueden utilizar para aumentar el conjunto de datos de entrenamiento.
4. **Transferencia de estilo:** la transferencia de estilo es una técnica en la que se aplica el estilo de una imagen a otra imagen. Esta técnica se puede utilizar para generar nuevas imágenes a partir de las imágenes existentes. Por ejemplo, se puede transferir el estilo de una imagen de una flor a una imagen de una persona para crear una imagen de una persona con un fondo de flores. Estas nuevas imágenes se pueden utilizar para aumentar el conjunto de datos de entrenamiento.

Es importante tener en cuenta que estas técnicas no siempre son efectivas y que pueden tener un impacto en la precisión del modelo. Por lo tanto, es importante evaluar cuidadosamente la precisión del modelo y ajustar las técnicas utilizadas en consecuencia.

2.5.5 Cuestión 5:

"*¿Se te ocurre alguna manera de modificar las imágenes para que el modelo mejore (en velocidad de entrenamiento o precisión)?* "

Hay varias técnicas que se pueden utilizar para mejorar el desempeño de un modelo de redes neuronales convolucionales (CNN) en el procesamiento de imágenes. Algunas de estas técnicas son las siguientes:

1. **Normalización:** normalizar las imágenes puede ayudar a mejorar el rendimiento de un modelo CNN. La normalización puede involucrar escalar los píxeles de la imagen a un rango de 0 a 1, o normalizar las imágenes utilizando otras técnicas.
2. **Aumento de datos:** aumentar el conjunto de datos mediante la aplicación de transformaciones aleatorias a las imágenes existentes, como rotaciones, recortes, volteos y ajustes de brillo, puede mejorar el rendimiento del modelo CNN. Esto aumenta la cantidad de datos de entrenamiento disponibles para el modelo, lo que puede ayudarlo a generalizar mejor a nuevas imágenes.
3. **Redimensionamiento:** cambiar el tamaño de las imágenes a un tamaño estándar puede mejorar el rendimiento del modelo CNN. Esto puede involucrar el agrandamiento o reducción de las imágenes para que todas las imágenes tengan el mismo tamaño.
4. **Filtrado de ruido:** eliminar el ruido de las imágenes, como manchas o granos, puede mejorar el rendimiento del modelo CNN. Esto puede involucrar la aplicación de filtros de suavizado, como filtro gaussianos o medianos, a las imágenes antes de que se utilicen para entrenar el modelo.
5. **Preprocesamiento específico de la tarea:** para tareas específicas, como la detección de bordes, se pueden aplicar técnicas específicas de preprocesamiento para mejorar el rendimiento del modelo CNN. Por ejemplo, se pueden aplicar filtros de detección de bordes a las imágenes antes de utilizarlas para entrenar el modelo.

En general, es importante experimentar con varias técnicas de preprocesamiento de imágenes para determinar qué técnicas funcionan mejor para una tarea específica y conjunto de datos.

Para mejorar la velocidad de la red podemos optar por estas opciones:

1. **Reducción del tamaño de la red:** disminuir el número de capas y/o neuronas en cada capa puede reducir el número de operaciones necesarias para procesar una imagen y, por lo tanto, mejorar la velocidad de la red. También se puede utilizar una arquitectura de red más eficiente, como *MobileNet* o *SqueezeNet*.
2. **Optimización de código:** optimizar el código para la ejecución en una GPU o TPU, como utilizar funciones vectorizadas, reducir el uso de memoria, etc., puede mejorar la velocidad de procesamiento de las imágenes.
3. **Uso de datos de menor resolución:** utilizar imágenes de menor resolución puede disminuir el número de píxeles a procesar y mejorar la velocidad de la red. Sin embargo, esto puede tener un impacto en la precisión del modelo.
4. **Uso de técnicas de paralelización:** utilizar técnicas de paralelización, como la división de lotes (*batching*) o la distribución de la carga de trabajo en múltiples GPUs o TPUs, puede mejorar la velocidad de procesamiento.
5. **Cuantización de modelos:** la cuantización de modelos es una técnica que reduce el número de bits necesarios para representar los pesos y activaciones de una red. Esto puede mejorar la velocidad de procesamiento y reducir el uso de memoria.

Es importante tener en cuenta que mejorar la velocidad de un modelo CNN puede tener un costo en términos de precisión. Por lo tanto, es necesario encontrar un equilibrio entre velocidad y precisión para la tarea específica que se está abordando.

3 Practica 2:

3.1 Cuestión 1:

"*Motivación del problema seleccionado y justificación de la solución que se quiere obtener*".

La clasificación manual de grandes volúmenes de calzado deportivo puede ser una tarea tediosa y propensa a errores, especialmente en entornos con una gran cantidad de productos de diferentes marcas. Además, en un entorno empresarial, el tiempo y los recursos pueden ser limitados, lo que dificulta aún más la tarea de clasificar los productos de manera precisa y eficiente.

En este contexto, la utilización de técnicas de aprendizaje profundo, como las redes neuronales convolucionales (CNN), pueden ser una solución efectiva para automatizar la tarea de clasificación de los zapatos deportivos por marca. Las CNN son capaces de extraer características importantes de las imágenes y utilizarlas para clasificarlas en categorías predefinidas.

Para aplicar una CNN a este problema en particular, se necesita un conjunto de datos etiquetados con información sobre la marca de cada par de zapatos. Una vez que se dispone de un conjunto de datos adecuado, se puede entrenar la CNN para que reconozca automáticamente la marca de los zapatos en las imágenes que se tomen con la cámara.

La solución propuesta consiste en implementar la CNN en una línea de producción automatizada, donde una cámara integrada captura imágenes de cada par de zapatos que se envían a la línea de producción. La CNN analiza las imágenes en tiempo real y envía una señal al brazo robótico indicando en qué palé debe colocar el par de zapatos correspondiente.

La automatización de la clasificación de los zapatos por marca con una CNN tiene varias ventajas, como la reducción del tiempo de procesamiento y la eliminación de errores humanos. Además, la solución puede ser escalable y adaptable a diferentes entornos y requisitos empresariales, lo que puede resultar en una mayor eficiencia y rentabilidad a largo plazo.

En resumen, la utilización de una CNN para la clasificación automática de zapatos deportivos por marca en un entorno de producción puede ser una solución eficaz y eficiente para mejorar la eficiencia y precisión de la clasificación de productos, lo que puede conducir a una mayor rentabilidad y satisfacción del cliente.

3.2 Cuestión 2:

3.2.1 Introducción:

El *dataset* proporciona una carpeta archive2 que aloja dos carpetas de imágenes, una con nombre *train* y otra denominada *test*. Estas, dentro tienen otras con imágenes de zapatillas de Nike, Adidas y Converse, cada una en su propia carpeta. El fin es entrenar una CNN para poder clasificar correctamente cada una de las zapatillas.

3.2.2 Preprocesamiento:

El preprocesamiento de datos es un paso importante en el aprendizaje automático, y puede ayudar a mejorar la precisión del modelo y reducir el tiempo de entrenamiento.

En el código se comienza definiendo los directorios de entrenamiento y prueba para los datos de imagen. Se utiliza el directorio de entrenamiento para crear conjuntos de datos de entrenamiento y validación. Los datos de prueba se almacenan en otro directorio diferente.

En la siguiente parte del código se establece el tamaño de la imagen que se utilizará durante el entrenamiento y la validación. En este caso, se utiliza un tamaño de 256x256 píxeles para la altura y la anchura de la imagen. También se establece el tamaño del lote a 64 y el número de épocas a 1000.

A continuación, se define un objeto de generador de datos de imagen de prueba utilizando la clase *ImageDataGenerator* de *Keras*. Esto se utiliza para preprocesar los datos de imagen de prueba antes de que se utilicen para evaluar el modelo.

Luego se definen tres generadores de conjuntos de datos de imagen diferentes utilizando la función *tf.keras.preprocessing.image_dataset_from_directory*() de *Keras*. La primera y la segunda generan conjuntos de datos de entrenamiento y validación respectivamente a partir del directorio de entrenamiento. Estos conjuntos de datos se dividen utilizando el argumento validation_split para separar una fracción del conjunto de entrenamiento para su uso como conjunto de validación. Además, se utiliza el argumento *label_mode* para especificar que la clasificación es categórica.

El tercer generador de datos utiliza la función *flow_from_directory*() de *Keras* para generar un conjunto de datos de prueba a partir del directorio de prueba. Este conjunto de datos se utiliza para evaluar el modelo después del entrenamiento.

En resumen, el código define tres generadores de conjuntos de datos de imagen diferentes utilizando la clase *ImageDataGenerator* de *Keras* y la función *tf.keras.preprocessing.image_dataset_from_directory*(). El primer y el segundo generador de conjuntos de datos se utilizan para entrenar y validar el modelo, mientras que el tercer generador se utiliza para evaluar el modelo después del entrenamiento.

```
# Directorios de entrenamiento, validación y prueba
train_dir = 'archive2/train'
test_dir = 'archive2/test'

# Altura y anchura de la imagen
img_width, img_height = 256, 256
# Tamaño de lote
batch_size = 64
# Epocas
epochs = 1000

# Preprocesar los datos de imagen utilizando ImageDataGenerator

test_datagen = ImageDataGenerator()

train_generator_1 = tf.keras.preprocessing.image_dataset_from_directory(train_dir,
                                                image_size=(img_width, img_height),
                                                batch_size=batch_size,
                                                validation_split=0.2,
                                                  subset="training",
                                                seed=1234,
                                                label_mode='categorical',)

val_generator_1 = tf.keras.preprocessing.image_dataset_from_directory(train_dir,
                                               image_size=(img_width, img_height),
                                               batch_size=batch_size,
                                               validation_split=0.2,
                                               subset="validation",
                                               seed=1234,
                                               label_mode='categorical')

test_generator_1 = test_datagen.flow_from_directory(test_dir,
                                               seed=1234,
                                               target_size=(img_height, img_width),
                                               class_mode="categorical", batch_size=32,
                                               shuffle=False)
```

Ilustración 45 Código de creación de los generadores y las clases

En los primeros experimentos que se realizaron el modelo no terminaba de generalizar correctamente la red, por lo que se decide crear nuevas imágenes tal y como hicimos en la primera parte de la práctica. De esta manera la red tendría más datos y no se sobreentrenaría.

Una vez se comprueba que no había sobreentreno en la red se manifiesta que las zapatillas que peor clasificaba el modelo, eran las Nike y las Adidas. Esto se debe a que son las que más varían entre unas y otras. Para solucionar este problema se decidió generar más zapatillas Nike que de las demás para que aprendiese mejor la red.

El número de imágenes quedaría de la siguiente manera:

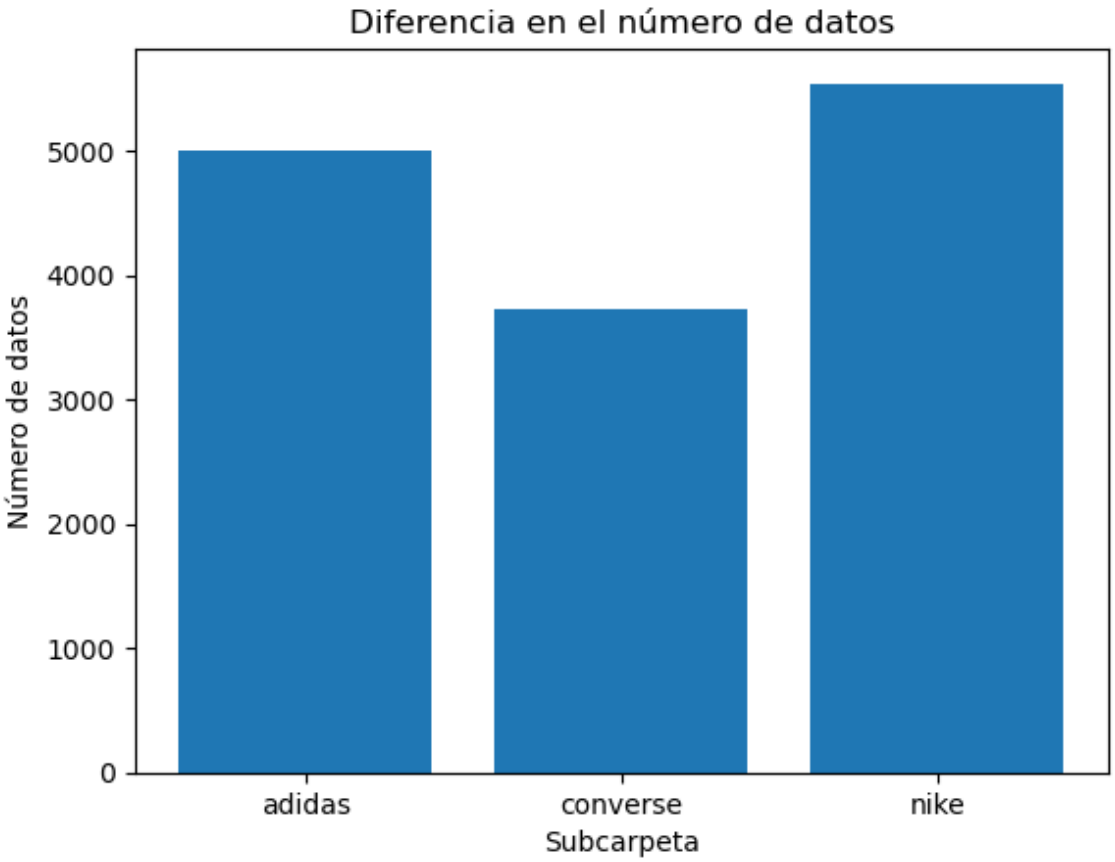


Ilustración 46 Gráfica de barras con la distribución de cada clase

3.2.3 Arquitectura de red y entrenamiento:

Para el entrenamiento de la red se ha decidido usar una CNN con una arquitectura basada en la red convolucional de *AlexNet.*

AlexNet es una arquitectura de red neuronal convolucional (CNN) profunda desarrollada por Alex Krizhevsky, Ilya Sutskever y Geoffrey Hinton en 2012. Fue la primera red neuronal profunda que ganó el concurso *ImageNet Large Scale Visual Recognition Challenge* (ILSVRC) en 2012, con un error de clasificación del 16,4%.

La arquitectura AlexNet tiene las siguientes características principales:

Capas convolucionales: AlexNet tiene cinco capas convolucionales, seguidas de capas de agrupamiento (*pooling*). Las capas convolucionales tienen diferentes tamaños de filtro y profundidad, que van desde 96 filtros de tamaño 11x11 en la primera capa hasta 256 filtros de tamaño 3x3 en la quinta capa. Las capas de agrupamiento reducen la dimensión espacial de las características.

Capas completamente conectadas: Después de las capas convolucionales, AlexNet tiene tres capas completamente conectadas con 4096 unidades cada una. Estas capas están destinadas a aprender representaciones no lineales de alto nivel de las características extraídas por las capas convolucionales.

Funciones de activación: AlexNet utiliza la función de activación *ReLU* (*rectified linear unit*) en todas las capas, excepto en la última capa de salida que utiliza la función de activación *softmax*.

Normalización de lotes: AlexNet utiliza la normalización de lotes (*batch normalization*) en todas las capas convolucionales y completamente conectadas para acelerar el entrenamiento y mejorar la generalización.

Regularización: AlexNet utiliza la regularización de *dropout* en las capas completamente conectadas para evitar el sobreajuste.

En general, la arquitectura de AlexNet es muy profunda para su tiempo, y utiliza una combinación de técnicas que ayudan a mejorar la eficacia de la red y la capacidad de generalización. Además, la arquitectura de AlexNet se convirtió en una referencia para la construcción de redes neuronales convolucionales profundas en la investigación y la industria.

Para desarrollarlo implementamos el siguiente código:

```
model = Sequential()
normalization_layer = layers.experimental.preprocessing.Rescaling(1./255, input_shape=(img_height, img_width, 3))
model.add(normalization_layer)
# capa de convolución 1
model.add(Conv2D(filters=96, kernel_size=(11,11), strides=(4,4), padding='valid', input_shape=(img_height,img_width,3)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(3,3), strides=(2,2), padding='valid'))

# capa de convolución 2
model.add(Conv2D(filters=256, kernel_size=(5,5), strides=(1,1), padding='same'))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(3,3), strides=(2,2), padding='valid'))

# capa de convolución 3
model.add(Conv2D(filters=384, kernel_size=(3,3), strides=(1,1), padding='same'))
model.add(Activation('relu'))

# capa de convolución 4
model.add(Conv2D(filters=384, kernel_size=(3,3), strides=(1,1), padding='same'))
model.add(Activation('relu'))

# capa de convolución 5
model.add(Conv2D(filters=256, kernel_size=(3,3), strides=(1,1), padding='same'))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(3,3), strides=(2,2), padding='valid'))

# capa completamente conectada 1
model.add(Flatten())
model.add(Dense(4096, input_shape=(200*200,)))
model.add(Activation('relu'))
model.add(Dropout(0.4))

# capa completamente conectada 2
model.add(Dense(4096))
model.add(Activation('relu'))
model.add(Dropout(0.4))

# capa de salida
model.add(Dense(3))
model.add(Activation('softmax'))

# compilar el modelo
model.compile(loss='categorical_crossentropy', optimizer=SGD(learning_rate = 0.01), metrics=['accuracy'])

stop_training_callback = StopTrainingCallback()
history = model.fit(
        train_generator_1,
        epochs = epochs,
        validation_data = val_generator_1,
        callbacks = [stop_training_callback],
)
```

Ilustración 47 Código de la arquitectura para el modelo de la práctica 2

Este código define una arquitectura de red neuronal convolucional basada en el modelo AlexNet, y luego la entrena con un conjunto de datos de entrenamiento y un conjunto de datos de validación.

La arquitectura de la red comienza con una capa de normalización que escala los valores de píxeles de las imágenes de entrada para que estén en el rango de [0,1]. Luego se agregan cinco capas convolucionales, cada una seguida de una función de activación *ReLU* y una capa de *pooling* máximo. La primera capa convolucional tiene 96 filtros con un tamaño de *kernel* de 11x11 y un *stride* de 4x4. La segunda capa convolucional tiene 256 filtros con un tamaño de *kernel* de 5x5 y un *stride* de 1x1. La tercera, cuarta y quinta capa convolucional tienen 384, 384 y 256 filtros, respectivamente, con un tamaño de *kernel* de 3x3 y un *stride* de 1x1. Después de

las capas convolucionales, hay dos capas completamente conectadas con 4096 neuronas cada una y una función de activación *ReLU*, seguidas de una capa de salida con tres neuronas y una función de activación *softmax*.

El modelo se compila utilizando la función de pérdida de entropía cruzada categórica y el optimizador SGD con una tasa de aprendizaje de 0.01. Luego se entrena el modelo en los datos de entrenamiento y validación utilizando el método *fit*() de *Keras*. Se define una instancia de la clase *StopTrainingCallback* para detener el entrenamiento si se alcanza una pérdida de entrenamiento menor a 0.2. La historia del entrenamiento se almacena en la variable *'history'*.

Se ha implementado una función para detener el entrenamiento cuando el *loss* de la red llega a ser menor que 0,2.

```
class StopTrainingCallback(tf.keras.callbacks.Callback):
    def on_epoch_end(self, epoch, logs={}):
        if logs.get('loss') < 0.2:
            print('\nSe ha alcanzado el valor de pérdida deseado.')
            self.model.stop_training = True
```

Ilustración 48 Código para detener el entrenamiento si el loss es menor que 0,2

Esta es una función personalizada de retroalimentación (*callback*) en *Keras* que se utiliza durante el entrenamiento del modelo para detener el entrenamiento temprano si se alcanza un cierto nivel de pérdida deseado.

La función *StopTrainingCallback* es una subclase de la clase *Callback* de *Keras*, que proporciona una serie de funciones de retroalimentación (*callbacks*) que se pueden llamar en diferentes etapas durante el entrenamiento del modelo.

La función *on_epoch_end* es una de estas funciones de retroalimentación que se llama al final de cada época durante el entrenamiento. Esta función acepta dos argumentos: *epoch* y *logs. epoch* es el número actual de épocas durante el entrenamiento y logs es un diccionario que contiene diferentes métricas y valores de pérdida del modelo en la época actual.

En esta función personalizada, se establece una condición para detener el entrenamiento si el valor de pérdida (*logs.get('loss'*)) es menor que 0.2. Si se cumple esta condición, se imprime un mensaje que indica que se ha alcanzado el valor de pérdida deseado y se establece la variable *self.model.stop_training* en True para detener el entrenamiento.

En resumen, la función *StopTrainingCallback* es una función personalizada de retroalimentación que se utiliza para detener el entrenamiento temprano si se alcanza un nivel de pérdida deseado. Se basa en la función *on_epoch_end* de la clase *Callback* de *Keras* para detener el entrenamiento si se cumple una condición específica.

Se ha hecho uso de la librería *Image* de *tensorflow* para mostrar la arquitectura de la red, además se ha calculado el *test_loss y accuracy y el train_loss y accuracy*. Los resultados son estos:

```
179/179 [==============================] - 55s 306ms/step - loss: 0.0908 - accuracy: 0.9710
Train accuracy: 0.9710233807563782
Train loss: 0.09077208489179611
4/4 [==============================] - 1s 139ms/step - loss: 0.8429 - accuracy: 0.7456
Test accuracy: 0.7456140518188477
Test loss: 0.8428589105606079
```

Ilustración 49 Resultados de la validación y el loss en el entrenamiento y el test

La arquitectura de la red quedaría de la siguiente manera:

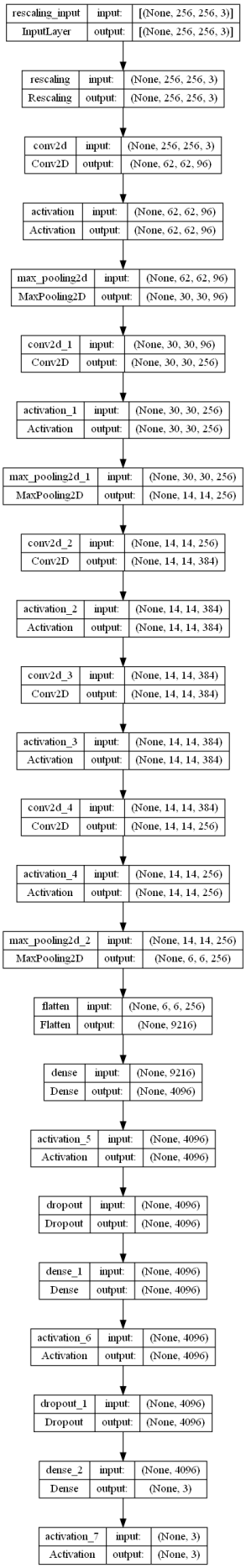


Ilustración 50 Arquitectura de la red de la práctica 2

Además se han generado gráficas para poder mostrar como varía el *train accuracy y loss y el validation accuracy y validation loss* durante las épocas del entrenamiento.

```
# Obtener los valores de loss y accuracy del entrenamiento y validación
train_loss = history.history['loss']
train_acc = history.history['accuracy']
val_loss = history.history['val_loss']
val_acc = history.history['val_accuracy']

# Crear una figura con dos subplots para mostrar loss y accuracy
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 5))

# Gráfico de loss
ax1.plot(train_loss, label='train_loss')
ax1.plot(val_loss, label='val_loss')
ax1.set_title('Training and Validation Loss')
ax1.set_xlabel('Epoch')
ax1.set_ylabel('Loss')
ax1.legend()

# Gráfico de accuracy
ax2.plot(train_acc, label = 'train_acc')
ax2.plot(val_acc, label = 'val_acc')
ax2.set_title('Training and Validation Accuracy')
ax2.set_xlabel('Epoch')
ax2.set_ylabel('Accuracy')
ax2.legend()

plt.show()

# Obtener predicciones para el conjunto de prueba
y_pred = model.predict(test_generator_1)
# Convertir las probabilidades en etiquetas de clase
y_pred = np.argmax(y_pred, axis=1)
# Obtener las etiquetas verdaderas del conjunto de prueba
y_true = test_generator_1.classes

# Crear matriz de confusión
confusion_mtx = confusion_matrix(y_true, y_pred)

# Mostrar la matriz de confusión
plt.figure(figsize=(10,8))
sns.heatmap(confusion_mtx, annot=True, fmt='d', cmap='Blues')
plt.xlabel('Predicción')
plt.ylabel('Etiqueta verdadera')
plt.show()
```

Ilustración 51 Código para dibujar las gráficas de los resultados

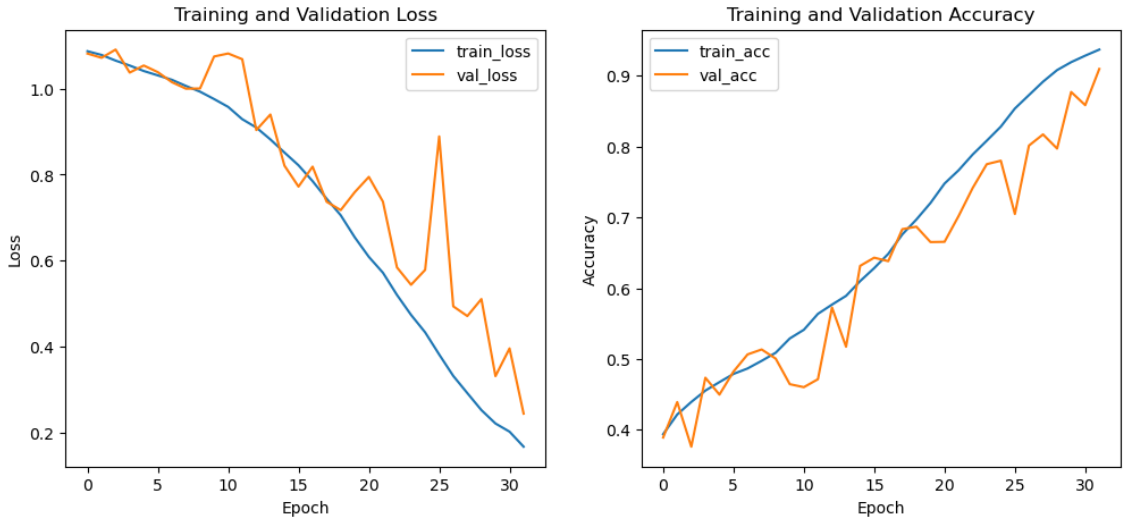


Ilustración 52 Gráfica de variación del accuracy y el loss para el train y el validation

Estos resultados se refieren al rendimiento del modelo después de ser entrenado y evaluado con los datos de entrenamiento y prueba proporcionados.

La primera línea indica que, durante el entrenamiento, el modelo alcanzó una precisión de entrenamiento del 97.10% y una pérdida de entrenamiento de 0.0908. La precisión de entrenamiento indica la proporción de imágenes de entrenamiento que fueron clasificadas correctamente por el modelo, mientras que la pérdida de entrenamiento representa el error del modelo durante el entrenamiento.

La segunda línea indica que, durante la evaluación del modelo en el conjunto de datos de prueba, la precisión del modelo fue del 74.56% y la pérdida fue de 0.8429. Esto significa que el modelo clasificó correctamente el 74.56% de las imágenes del conjunto de datos de prueba.

En general, un modelo con una precisión alta y una pérdida baja tanto en los datos de entrenamiento como en los datos de prueba indica que el modelo es capaz de aprender patrones importantes en los datos de entrenamiento y generalizar bien a nuevos datos de prueba. Sin embargo, un modelo con una precisión alta en los datos de entrenamiento, pero baja en los datos de prueba puede estar sobre ajustando (*overfitting*) a los datos de entrenamiento y no generalizar bien a nuevos datos.

Además, se ha analizado la matriz de confusión de la predicción del modelo.

La matriz de confusión representa el desempeño de un modelo de clasificación. Es una tabla que muestra el número de predicciones correctas e incorrectas realizadas por el modelo en cada clase.

En este caso, la matriz de confusión es de tamaño 3x3, lo que indica que hay 3 clases de productos que se están clasificando: Adidas, Converse y Nike. Los valores en la diagonal principal representan el número de predicciones correctas para cada clase, mientras que los valores fuera de la diagonal principal representan las predicciones incorrectas.

En particular, la matriz de confusión es la siguiente:

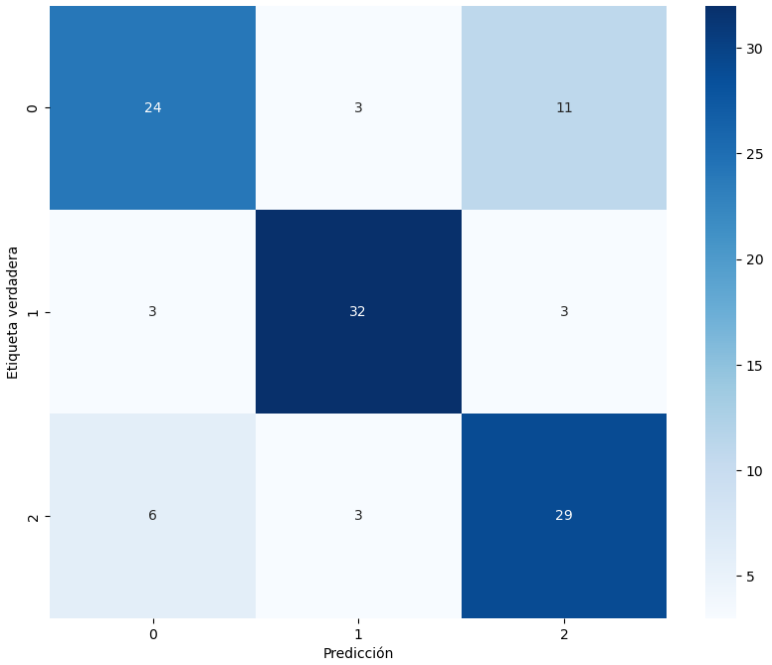


Ilustración 53 Matriz de confusión tras el entrenamiento de la práctica 2

Las etiquetas '0' son ADIDAS, las '1' son NIKE y '2' son CONVERSE, por lo tanto:

- Hay 24 casos que son verdaderos positivos (ADIDAS - ADIDAS)
- Hay 3 casos que son verdaderos negativos (ADIDAS - NIKE)
- Hay 11 casos que son falsos positivos (ADIDAS - CONVERSE)
- Hay 3 casos que son falsos negativos (NIKE - ADIDAS)
- Hay 32 casos que son verdaderos positivos (NIKE - NIKE)
- Hay 3 casos que son verdaderos negativos (NIKE - CONVERSE)
- Hay 6 casos que son falsos negativos (CONVERSE - ADIDAS)
- Hay 3 casos que son verdaderos negativos (CONVERSE - NIKE)
- Hay 29 casos que son verdaderos positivos (CONVERSE - CONVERSE)

.

En general, la matriz de confusión proporciona una idea de cómo se está desempeñando el modelo en términos de precisión y errores de clasificación. Además, se puede utilizar para calcular diversas métricas de evaluación del modelo, como la precisión, la sensibilidad y la especificidad.

3.2.4 Clasificación/predicción:

Por último, se ha realizado una clasificación y predicción del modelo con los datos de la carpeta test. Los resultados son los siguientes:

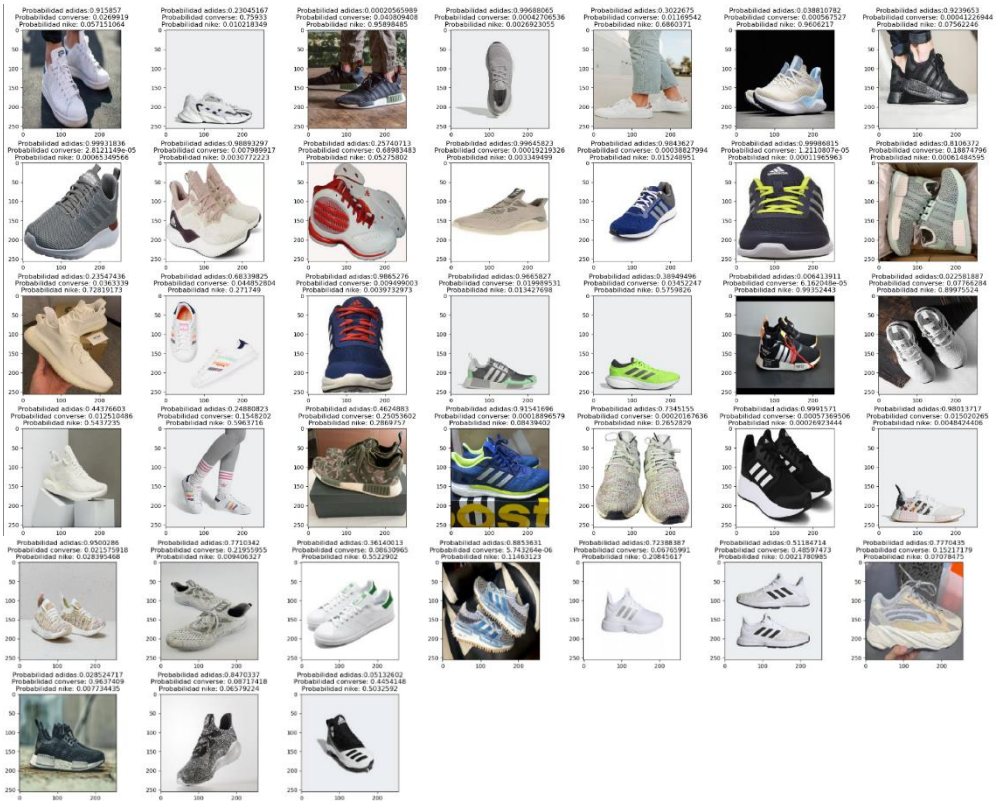


Ilustración 54 resultados de la predicción y evaluación para la clase Adidas

Como se puede observar, en la gran mayoría de las imágenes la red acierta, pero en las más complicadas se confunde, probablemente con más datos podría conseguirse que mejorase completamente la predicción y no fallase.

Ilustración 55 resultados de la predicción y evaluación para la clase Converse

Como se puede observar, este tipo de zapatillas al ser la gran mayoría muy parecidas, son clasificadas sin apenas fallos. Por esta razón, como ya se ha comentado, es la clase que menos datos tiene.

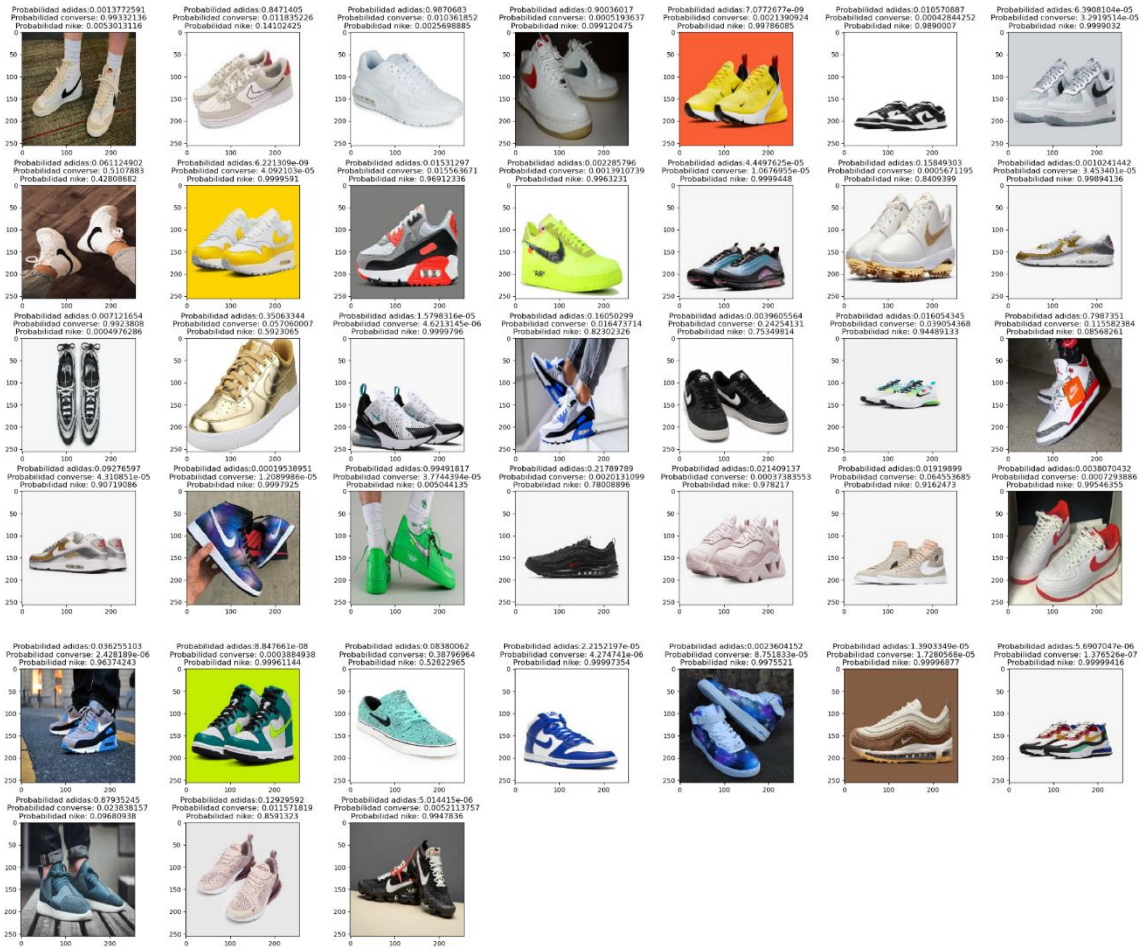


Ilustración 56 resultados de la predicción y evaluación para la clase Nike

Por último, la clasificación de Nike es la que más falla, pues son muy distintas unas con otras, a lo largo del estudio se ha ido mejorando la predicción y la clasificación de estas, siendo esta última la óptima.

4 Conclusiones

En primer lugar, se puede afirmar que el entrenamiento de una CNN depende en gran medida de la cantidad y calidad de los datos suministrados. En este estudio, se ha demostrado que a medida que se dispone de más datos para entrenar la red, ésta tiende a generalizar mejor, lo que significa que es capaz de reconocer patrones en imágenes nuevas y no vistas previamente.

En particular, en la primera parte del estudio, se ha realizado una clasificación de neumonías en imágenes de radiografías. Aunque los investigadores no son expertos en el campo médico, la CNN ha generado un buen clasificador para esta tarea. Esto sugiere que la CNN es capaz de aprender y reconocer patrones en imágenes médicas a pesar de que el equipo investigador no tenga una formación especializada en este campo. Es decir, la CNN es capaz de captar patrones relevantes para la tarea de clasificación, lo que es una ventaja importante en el diagnóstico médico.

En la segunda parte del estudio, se ha llevado a cabo un buen análisis de los datos y se ha obtenido una buena precisión en la tarea de clasificación. Esto indica que el equipo investigador ha logrado identificar patrones relevantes en las imágenes, lo que ha permitido una buena clasificación. Es importante destacar que el análisis de los datos y la selección de los mejores atributos son clave para obtener buenos resultados en una tarea de clasificación.

En resumen, se puede concluir que la cantidad y calidad de los datos son fundamentales para el entrenamiento de una CNN. Además, la capacidad de la CNN para identificar patrones relevantes en las imágenes, incluso en áreas donde los investigadores no son especialistas, la convierte en una herramienta útil en tareas de clasificación y diagnóstico médico. Finalmente, la correcta selección de atributos y el análisis detallado de los datos son clave para obtener buenos resultados en una tarea de clasificación.

5 Bibliografía

[1] Saturdays.ia, «Saturdays.ia,» [En línea]. Available: https://saturdays.ai/2021/11/11/neumonia/.

[2] A. B. Sanchiz, *TFG - Diseño de un modelo Deep Learning de clasificación de imágenes para la detección de la neumonía.*

[3] V. L. G. Francisco, *TFG - Algoritmo de diagnóstico preliminar de neumonía a partir de imágenes radiográficas del tórax.*

[4] R. S. Salas, *TFM - Aprendizaje profundo y neumonía: modelo de clasificación de imágenes de rayos-X para una detección más rápida.*

[5] TensorFlow, «TensorFlow.org,» [En línea]. Available: https://www.tensorflow.org/api_docs/python/tf/keras/utils/image_dataset_from_directory.

[6] R. R. Abril, «AlexNet y clasificación de imágenes,» LMO, [En línea]. Available: https://lamaquinaoraculo.com/computacion/alexnet/. [Último acceso: 13 04 2023].

6 Anexo I

INSTALACIÓN DE LIBRRÍAS PARA LA BUENA EJECUCIÓN DE LA PRÁCTICA

Primero, asegúrate de tener Python 3.x instalado en tu sistema.

Abre una terminal o línea de comandos en tu sistema operativo.

Para instalar las librerías necesarias, ejecuta el siguiente comando:

- pip install Pillow keras numpy pandas matplotlib tensorflow scikit-learn seaborn

Este comando instalará todas las librerías necesarias para ejecutar el código proporcionado.

Además, para poder generar las arquitecturas deberás descargar la librería graphviz de esta url:

https://graphviz.gitlab.io/download/ y ejecutar -> pip install graphviz.

También puedes ejecutar los archivos librerias.bat o librerias.sh según tu sistema operativo.

Una vez que se hayan instalado todas las librerías, puedes ejecutar el script Python.

Si todo funciona correctamente, el script Python debería ejecutarse sin problemas y producir los resultados esperados.

Si tienes algún problema para instalar o utilizar alguna de las librerías, consulta la documentación oficial o la comunidad de desarrolladores correspondiente para obtener ayuda.