PROYECTO DE PROGRAMACIÓN.

Presentación del dataset a analizar.

El dataset que vamos a analizar en el desarrollo de la presente práctica (***the-nature-conservancy-fisheries-monitoring***) está formado por un conjunto imágenes que donde aparece una especie de pez en cada una de ellas.El objetivo de este análisis es el de predecir que tipo de pez es el que hay en cada imagen.

En la siguiente ilustración se presentan algunas de las especies objeto de este estudio:

Para llevar a cabo esta tarea se hará uso de diferentes redes neuronales, evaluando el rendimiento de cada una de ellas para conseguir así el mejor modelo de clasificación posible y aplicando distintas técnicas vistas a lo largo de la asignatura para conseguir el mejor rendimiento y la mayor eficiencia de las citadas redes neuronales.

Se utilizará las librerias de Keras y TensorFlow, así como otras librerias más comunes como Numpy, Matplotlib o Seaborn para facilitar la visualización gráfica de los resultados o para trabajar de forma más ágil el cálculo numérico.

Desarrollo del análisis.

0. Carga de librerías y ajustes iniciales

In [ ]:

```
# CARGA DE LIBRERIAS

from google.colab import drive, files
from numpy.random import seed
import numpy as np
import cv2
from os import listdir
import random
import matplotlib.pyplot as plt
import seaborn as sns
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.utils import to_categorical
```

```
from sklearn.model_selection import train_test_split
from tensorflow.keras.preprocessing import image_dataset_from_directory
from sklearn.utils import class_weight
from tensorflow.keras import backend as K
from tensorflow.keras.layers import Input, Conv2D, Activation, Flatten, Dense, Dropout, BatchNormali
from tensorflow.keras.models import Model, Sequential
from tensorflow.keras.models import Sequential
from tensorflow.keras.optimizers import SGD, Adam
from tensorflow.keras.callbacks import EarlyStopping
from sklearn.metrics import classification_report
from tensorflow.keras.applications import imagenet_utils
from sklearn.preprocessing import LabelBinarizer
from tensorflow.keras import layers
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from google.colab import files
# Ploteado del esquema gráfico del modelo
from keras.utils.vis_utils import plot_model
```

In [ ]:
```
# CONFIGURACIÓN Y CARGA DE SEEDS

# semilla para numpy
seed(7)
# semilla para Tensorflow
tf.random.set_seed(8)
```

CONEXIÓN DIRECTORIO DE TRABAJO EN GOOGLE DRIVE

In [ ]:
```
# Conectamos con Google Drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

In [ ]:
```
# Definimos ruta absoluta a directorio de trabajo en Google Drive
BASE_FOLDER = '/content/drive/MyDrive/07_MIAR_RNDL_CONV2/ProyectoProgramacion/'
```

0.1. Definición de funciones auxiliares y utilidades

Función para ploteado de curva de aprendizaje

In [ ]:
```
def plot_learning_curves(H,epochs):
  plt.style.use("ggplot")
  plt.figure()
  plt.plot(np.arange(0, epochs), H.history["loss"], label="train_loss")
  plt.plot(np.arange(0, epochs), H.history["val_loss"], label="val_loss")
  plt.plot(np.arange(0, epochs), H.history["accuracy"], label="train_acc")
  plt.plot(np.arange(0, epochs), H.history["val_accuracy"], label="val_acc")
  plt.title("Entrenamiento: Loss and Accuracy")
  plt.xlabel("Epoch #")
  plt.ylabel("Loss/Accuracy")
  plt.legend()
  plt.show()
```

Función diseñada para extraer imágenes y sus correspondientes etiquetas

Se utiliza el método as_numpy_iterator con el objetivo de crear un nuevo array que contenga la citada información.

In [ ]:
```
# Función de conversión a array NumPy
def toNumPyArray_x_y(image_ds):
  x = []
  y = []
  #for image, label in tfds.as_numpy(image_ds):
  #for image, label in image_ds.as_numpy_iterator():
  for image, label in image_ds:
    x.append(image.numpy())
    y.append(label.numpy())
  x = np.array(x)
  y = np.array(y)
  return x, y
```

Función para ploteado de diagrama de arquitectura

Utilizado para representar de forma gráfica un esquema de la arquitectura del modelo de red.

In [ ]:
```
# Función para mostrar gráfico de arquitectura de red
def plot_schema(model_name, file_name='model_plot.png', shapes=True, layer_names=True):


 return plot_model(model=model_name,
             to_file=file_name,
             show_shapes=shapes,
             show_layer_names=layer_names)
```

1. Carga del conjunto de datos (desde Kaggle)

In [ ]:
```
# Verificación de la versión de la API de Kaggle en Colab
!pip install --upgrade --force-reinstall --no-deps kaggle
```

```
Collecting kaggle
  Downloading kaggle-1.5.12.tar.gz (58 kB)
     |████████████████████████████████| 58 kB 6.7 MB/s  eta 0:00:01
Building wheels for collected packages: kaggle
  Building wheel for kaggle (setup.py) ... done
  Created wheel for kaggle: filename=kaggle-1.5.12-py3-none-any.whl size=73051 sha256=cb182fd00ca9807
3d00ef2982fb7ee4ca9b1d794c0bf82068321058fa23e65a0
  Stored in directory: /root/.cache/pip/wheels/62/d6/58/5853130f941e75b2177d281eb7e44b4a98ed46dd155f5
56dc5
Successfully built kaggle
Installing collected packages: kaggle
  Attempting uninstall: kaggle
    Found existing installation: kaggle 1.5.12
    Uninstalling kaggle-1.5.12:
      Successfully uninstalled kaggle-1.5.12
Successfully installed kaggle-1.5.12
```

A EJECUTAR SOLO LA PRIMERA VEZ

In [ ]:
```
# Seleccionamos y cargamos el API de Kaggle descargado (kaggle.json)

files.upload()
```

In [ ]:
```
# Creamos directorio donde guardar el fichero kaggle.json
!mkdir ~/.kaggle
!cp kaggle.json ~/.kaggle/
!chmod 600 ~/.kaggle/kaggle.json
```

In [ ]:
```
# Descargamos dataset de Kaggle
!kaggle competitions download -c the-nature-conservancy-fisheries-monitoring
```

```
Downloading the-nature-conservancy-fisheries-monitoring.zip to /content
100% 2.10G/2.11G [00:57<00:00, 39.2MB/s]
100% 2.11G/2.11G [00:57<00:00, 39.6MB/s]
```

In [ ]:
```
# Creamos directorio para descomprimir los datos
!mkdir my_dataset
```

In [ ]:
```
# Descomprimimos datos
!unzip the-nature-conservancy-fisheries-monitoring.zip -d my_dataset
```

```
Archive:  the-nature-conservancy-fisheries-monitoring.zip
  inflating: my_dataset/sample_submission_stg1.csv.zip
  inflating: my_dataset/sample_submission_stg2.csv.zip
  inflating: my_dataset/test_stg1.zip
  inflating: my_dataset/test_stg2.7z
  inflating: my_dataset/train.zip
```

El conjunto ***train*** proporcionado por defecto desde kaggle es el dataset etiquetado. Se usará dicho conjunto para realizar el entrenamiento completo. Cuando se realice el entrenamiento de la red se procederá a dividirlo en los conjuntos de ***train***, ***validation*** y ***test***

In [ ]:
```
# descomprimo en mi directorio google drive
%%capture
!unzip my_dataset/train.zip -d my_dataset
```

El dataset ***test*** de Kaggle es en realidad el set de *datos no etiquetados* que originalmente se utilizó durante la

competición. No va a ser utilizado en este proyecto, simplemente se adjunta con el objeto de utilizarlo en posibles futuros experimentos.

In [ ]:
```
#!unzip my_dataset/test_stg1.zip
```

ACLARACIÓN: Hemos cargado el siguiente dataset del Nature Conservacy project:

- ***train***: es en realidad el conjunto de imágenes ya etiquetadas, y el que usaremos para todo el entrenamiento (por lo que será dividido en train, validation y test)

Con respecto al test ***test_stg1***, se reitera que es un conjunto de imáges no etiquetadas que se proporcionó como parte de la competición Kaggle y en principio no se usará en este trabajo.

2. Inspección del conjunto datos

2.1 Tratamiento previo de las imágenes del dataset

Se revisa el formato de la información que nos proporciona inicialmente el dataset.

Disponemos de una serie de imágenes previamente etiquetadas y clasificadas dentro de carpetas nombradas precisamente con dicha etiqueta.

Empezamos realizando una selección aleatoria desde una carpeta con el objeto de comprobar la resolución de las imágenes proporcionadas.

In [ ]:
```
path= BASE_FOLDER + 'my_dataset/train/ALB/'
print(path)
files=listdir(path)
sample_files = [path + file for file in random.sample(files, 20)]
images_shapes = [cv2.imread(file).shape for file in sample_files]
images_shapes
```

```
/content/drive/MyDrive/07_MIAR_RNDL_CONV2/ProyectoProgramacion/my_dataset/train/ALB/
```

Out[ ]:
```
[(720, 1280, 3),
 (720, 1280, 3),
 (720, 1280, 3),
 (718, 1276, 3),
 (720, 1280, 3),
 (720, 1280, 3),
 (720, 1280, 3),
 (670, 1192, 3),
 (720, 1280, 3),
 (720, 1280, 3),
 (670, 1192, 3),
 (670, 1192, 3),
 (720, 1280, 3),
 (720, 1280, 3),
 (720, 1280, 3),
 (750, 1280, 3),
 (750, 1280, 3),
 (720, 1280, 3),
 (720, 1280, 3),
 (720, 1280, 3)]
```

- Se observa que el tamaño que aparece con mayor frecuencia es el de 720x1280 y el resto está en un tamaño similar.
- Se normaliza la información para agilizar y minimizar el trabajo computacional del proceso de training, para lo cual realizamos un reajuste del tamaño de las imágenes a un formato de 224x224. De esta forma, también se puede volver a utilizar estos datos en el transfer-learning mediante el uso del método ***image_dataset_from_directory***.

2.2 Carga desde disco (nuestra unidad de Google Drive definida como "BASE_FOLDER")

Se va a utilizar el método comentado anteriormente, **image_dataset_from_directory**. Este método nos proporciona una serie de utilidades que va a facilitar, entre otros, la detección automática de las etiquetas en base al nombre de la carpeta contenedora, con lo que podemos usar la codificación categórica proporcionada por defecto.

Los datos quedan almacenados dentro de un objeto del tipo BatchDataset, con un tamaño de batch=32, de forma automática.

Como el número de imágenes a cargar y procesar no es extremadamente elevado, se cargan desde el directorio. (En caso de haber tenido un número muy alto de imágenes y para evitar problemas de memoria, habríamos usado un data generator usando el método flow_from_directory de ImageDataGenerator).

Las imágenes pasarán directamente a cargarse sobre los dos datasets **train** y **test**. (La función image_dataset_from_directory se usa para separar *train* y *validation*. En esta ocasión, se va a separar inicialmente **train** y **test**, porque no se proporcionaba la separación en el conjunto de imágenes original. Por tanto, siguiendo la pauta más habitual, se usará un 80% de todo el dataset para ***train*** y dejaremos el 20% para ***test***. Más tarde se procederá a realizar una separación de un 10% del conjunto ***train*** para ser utilizado como ***validation***.

Por último se pretende ejecutar un **unbatch**, para evitar que quede fijado el tamaño del batch en el proceso de entrenamiento.

In [ ]:
```
image_size = (224, 224)
path= BASE_FOLDER + "my_dataset/train/"
# uso del método image_dataset_from_directory (cargar la información en batches de tamaño 32)
train_batch_ds = image_dataset_from_directory(
    directory = path,
    validation_split=0.2,
    subset="training",
    labels='inferred',
    label_mode='int',
    seed=1337,
    image_size=image_size
)
test_batch_ds = image_dataset_from_directory(
    directory = path,
    validation_split=0.2,
    subset="validation",
    labels='inferred',
    label_mode='int',
    seed=1337,
    image_size=image_size
)

# Proceso de UNBATCH de ambos conjuntos TRAIN y TEST
train_ds = train_batch_ds.unbatch()
test_ds = test_batch_ds.unbatch()

# Extracción y muestra por pantalla de las etiquetas para usar como los nombres de las clases
class_names = train_batch_ds.class_names.copy()
print()
print ("Las etiquetas de las clases del dataset son: ")
print (class_names)
```

```
Found 3777 files belonging to 8 classes.
Using 3022 files for training.
Found 3777 files belonging to 8 classes.
Using 755 files for validation.

Las etiquetas de las clases del dataset son:
['ALB', 'BET', 'DOL', 'LAG', 'NoF', 'OTHER', 'SHARK', 'YFT']
```

2.3 Conversión datos de TF a NumPy

Para facilitar el tratamiento del dataset, convertimos el mismo a un objeto NumPy utilizando la función creada al uso en el apartado ***0.1 Definición de funciones auxiliares y utilidades***.

In [ ]:
```
x_train, y_train = toNumPyArray_x_y(train_ds)
x_test, y_test = toNumPyArray_x_y(test_ds)
```

Mostramos como nos quedan ahora nuestros datos:

In [ ]:
```
print(x_train.shape)
print(y_train.shape)
print(x_test.shape)
print(y_test.shape)
```

```
(3022, 224, 224, 3)
(3022,)
(755, 224, 224, 3)
(755,)
```

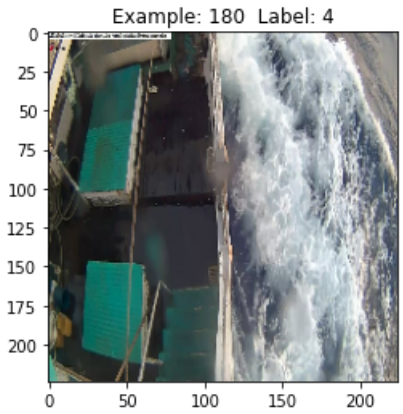
2.4 Inspección de las imágenes reprocesadas

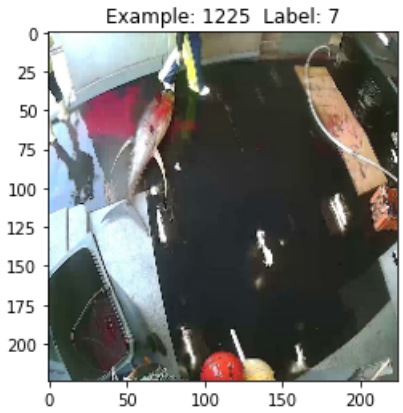
Se revisa en este punto la tipología de las imágenes a tratar, centrados en el tamaño de las mismas para asumir el tipo de resolución más adecuado para cargar los distintos lotes (***batch***).

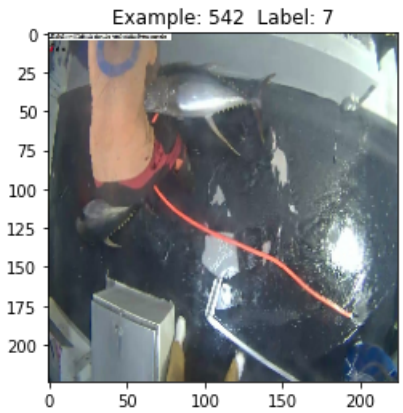
Tomaremos un conjunto pequeño de imágenes que forma parte del conjunto de ***train*** para agilizar este cálculo:

In [ ]:

```
# Inspección imágenes aleatoria
def display_image(num):
  # Seleccionar la imagen num del x_train y normalizar
  image = x_train[num,:,:]/255.0
  # Seleccionar el target num de y_train
  label = y_train[num]
  # Mostrar
  plt.title('Example: %d  Label: %d' % (num, label))
  plt.imshow(image)
  plt.show()

display_image(np.random.randint(5, x_train.shape[0]))
display_image(np.random.randint(5, x_train.shape[0]))
display_image(np.random.randint(5, x_train.shape[0]))
```







En las imágenes mostradas se puede comprobar que los peces forman parte de la escena, compartiendo la misma con pescadores, enseres, cubierta,...

Podría ser necesario un tratamiento previo de las imágenes para realizar 'recortes' (usando la detección de objetos mediante 'bounding boxes' con SSD, por ejemplo, para obtener una clasificación binaria) que permitan aislar los 'objetos

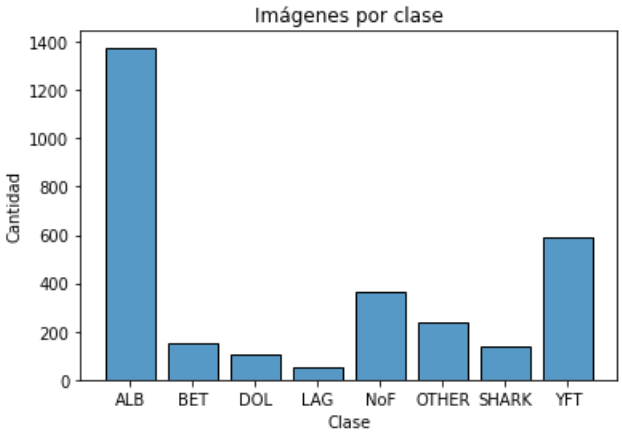
de interés' (los peces, en este caso). A partir de ese punto, se utilizarían las imágenes resultantes como fuente para trabajar únicamente con los peces, prescidiendo del resto de elementos que pueden aportar algún tipo de 'interferencia' en un tratamiento menos granular.

2.5 Clasificación de la información

En este apartado se procede a realizar un rápido análisis sobre la clasificación y distribución de las imágenes procesadas versus las etiquetas que tienen asignadas, a través de un histograma donde reflejar las etiquetas utilizadas en el conjunto de entrenamiento (y_train), a la vez que obtenemos la cantidad de imágenes existentes en cada una de las clases.

In [ ]:
```
# Clasificación y conteo
unique, counts = np.unique(y_train, return_counts=True)
print(np.asarray((unique,counts)).T)

# Histograma
label_names = class_names.copy()
label_names.insert(0,'') # visualizamos nombre de clase
fig = sns.histplot(y_train, discrete=True, shrink=.8)
fig.set_xticklabels(label_names)
plt.xlabel('Clase')
plt.ylabel('Cantidad')
plt.title('Imágenes por clase')

plt.show()
```

```
[[   0 1377]
 [   1  151]
 [   2  102]
 [   3   55]
 [   4  367]
 [   5  240]
 [   6  140]
 [   7  590]]
```



El dataset presenta una distribución muy desigual. Se puede observar como la clase 0-ALB (Albacore Tuna) es la que presenta mayor número de muestras. Este hecho puede repercutir seriamente al aprendizaje de la red neuronal a diseñar.

Frente a esta situación debe plantearse la ejecución de acciones sobre este dataset que puedan paliar este hecho. Entre las opciones que se pueden estudiar tenemos la aplicación de *data augmentation* o el uso de técnicas tambien de *class weight*, con el objeto de 'compensar' la abundancia de la clase mayoritaria. ***Se analizará este hecho más adelante.***

3. Acondicionamiento del conjunto de datos

3.1 Preprocesado

1. Se prepara la información para reducir los valores a un rango entre 0 y 1.
2. Ejecución de un 'split' sobre el conjunto de entrenamiento para generar los conjuntos de entrenamiento y validación.

Se usará la codificación 'original' de las clases (se prepara también un futuro uso de ONE-HOT-ENCODING')

In [ ]:
```
# Cálcular número de clases
number_classes = len(np.unique(y_train))
```

```
# Conversión a rango 0-1 (reducción coste de cómputo)
x_train, x_te = x_train/255.0, x_test/255.0

# ONE-HOT ENCODING
#y_train = to_categorical(y_train, num_classes = number_classes))
#y_yest = to_categorical(y_test, num_classes = number_classes))

# Split 80/20 para train y validation
x_tr, x_val, y_tr, y_val = train_test_split(x_train, y_train, test_size=0.2, random_state=42)
```

3.2 Almacenamiento de train y validation en el disco (opcional)

In [ ]:
```
#Creando directorio para guardar los arrays de imágenes
!mkdir numpy_sets
```

In [ ]:
```
# Se almacena una copia de los datasets en formato numpy en drive. De esta forma se pueden usar sin
path=BASE_FOLDER + "numpy_sets/"
np.save(path+"x_train.npy", x_train)
np.save(path+"y_train.npy", y_train)

np.save(path+"x_tr.npy", x_tr)
np.save(path+"x_val.npy", x_val)
np.save(path+"x_te.npy", x_te)

np.save(path+"y_tr.npy", y_tr)
np.save(path+"y_val.npy", y_val)
np.save(path+"y_test.npy", y_test)
```

4. Desarrollo de la arquitectura de la red neuronal y entrenamiento de la solución

Como se ha comprobado en puntos anteriores, la información del dataset que vamos a utilizar está muy desbalanceada. Se pudo observar que hay una clase (*la clase 0 con la etiqueta ALB*) que contiene un número mucho más elevado de muestras que las clases restantes.

A pesar de ello, iniciaremos el experimento usando los datos originales "*as is*" para, en el siguiente paso, utilizar la técnica de ***data augmentation*** con la que se intentará paliar esta descompensación.

4.1 Primer modelo: CNN '*from scratch*' sobre dataset original

4.1.1 Construcción del modelo y posterior entrenamiento

Ya se ha comentado anteriormente las características y peculiaridades de el dataset a analizar dado el tipo y origen de las imágenes que lo componen a lo que, a mayores, debemos sumarle el citado desbalanceo del reparto de las muestras sobre las clases que componen el dataset. Para este primer modelado se pretende hacer uso del método class_weight en el entrenamiento (fase de fit)

Proponemos un primer modelo convolucional formado por 3 bloques simples que estará definido con la siguiente arquitectura:

- Una única capa convolucional de 32, 64 y 256 filtros progresivamente
- Una capa de BatchNormalization intercalada entre la capa convolucional y la capa de pooling (también a la salida de la capa 'fully dense'). La intención es por un lado normalizar las salidas de las capas convolucionales a fin de optimizar el modelo y reducir el 'overfitting'. Lo que consigue es que se estandarizan las salidas de las activaciones de la capa de convolución (la media y varianza) reduciendo el 'desplazamiento' interno de dichos valores (osea de la distribución), de tal manera que ayuda a estabilizar el aprendizaje.
- Una capa de Pooling a la salida de la cada capa convolucional.
- La capa de activación ReLu. Cabe considerar la posible incorporación de esta capa antes de la de pooling. Aunque MaxPool(ReLu) es equivalente a ReLu(MaxPool), se considera que, puesta después se proporciona cierta mejora de carácter computacional puesto que, en este paso después de aplicar la capa de MaxPool, ya se ha conseguido reducir el número de elementos a gestionar.

Incorporamos también, con el objeto de incrementar el rendimiento de esta primera arquitectura, una capa de **Dropout** a la salida de la capa densa del 'top model', justo en la que se dispone del mayor número de neuronas. Usando el dropout se consigue desactivar aleatoriamente un tanto por ciento concreto de neuronas para esquivar un posible 'overfitting'. Se usará, inicialmente un valor bajo en Dropout (0.2) colocado a la salida de las capas pooling, y un valor alto en Dropout sobre la capa 'fully dense', donde, como se ha mencionado, existe un mayor número de parámetros.

In [ ]:
```
# Mostramos tamaño de las imágenes del conjunto train
ds_shape = x_tr[0].shape
print (ds_shape)

model_1 = Sequential()

#BASE MODEL
# Primer set de capas
model_1.add(Conv2D(32, (3,3), padding="same", input_shape=(ds_shape)))
model_1.add(BatchNormalization())
model_1.add(MaxPooling2D())
model_1.add(Activation("relu"))
model_1.add(Dropout(0.2))

# Segundo set de capas
model_1.add(Conv2D(64, (3,3), padding="same"))
model_1.add(BatchNormalization())
model_1.add(MaxPooling2D())
model_1.add(Activation("relu"))
model_1.add(Dropout(0.2))

# Tercer set de capas
model_1.add(Conv2D(256, (3,3), padding="same"))
model_1.add(BatchNormalization())
model_1.add(MaxPooling2D())
model_1.add(Activation("relu"))
model_1.add(Dropout(0.2))

#TOP MODEL
model_1.add(Flatten())
model_1.add(Dense(units = 512))
model_1.add(BatchNormalization())
model_1.add(Activation("relu"))
model_1.add(Dropout(0.5))
# Clasificador softmax
model_1.add(Dense(units = number_classes, activation = 'softmax'))
```

```
(224, 224, 3)
```

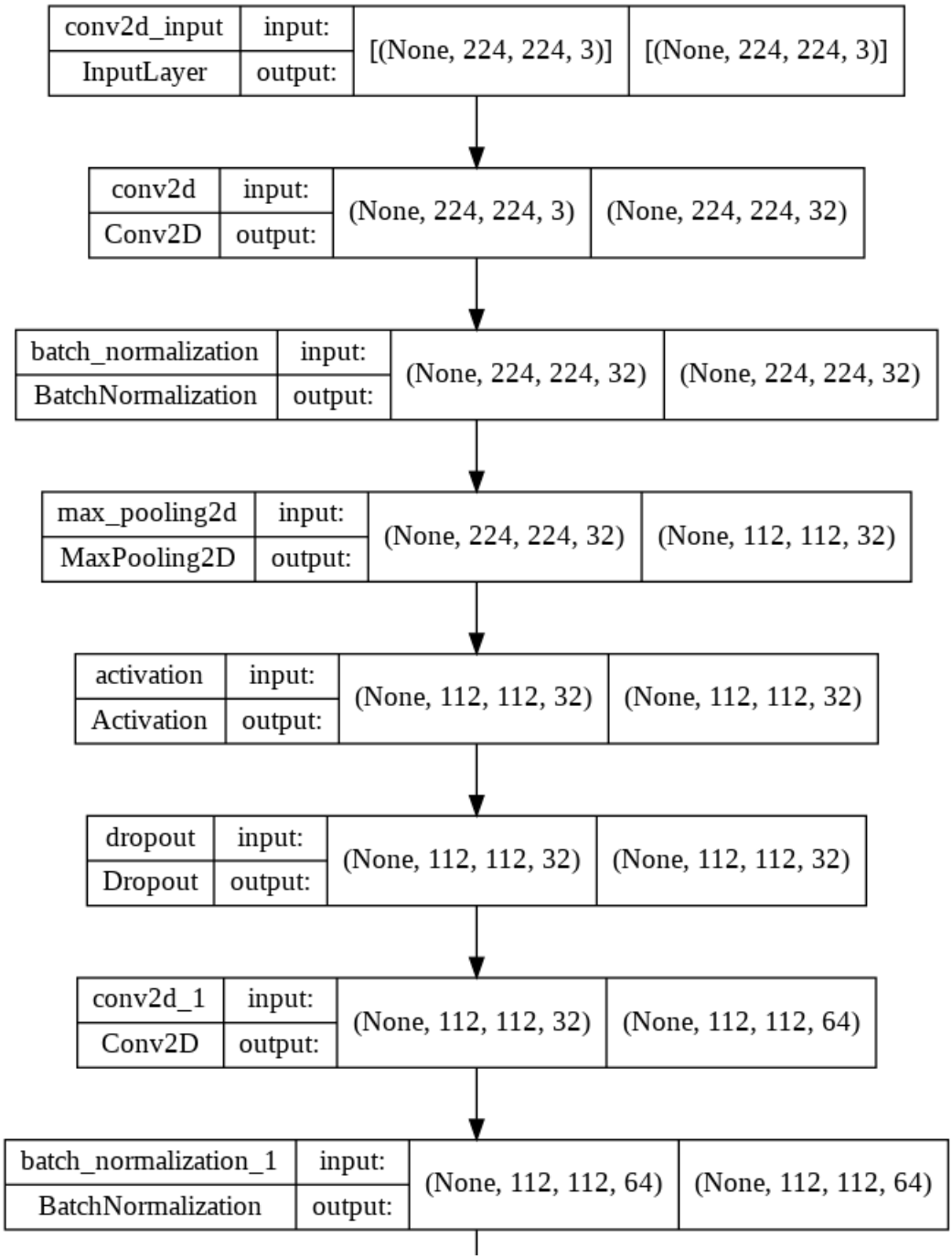
In [ ]:
```
model_1.summary() # display de la arquitectura
```

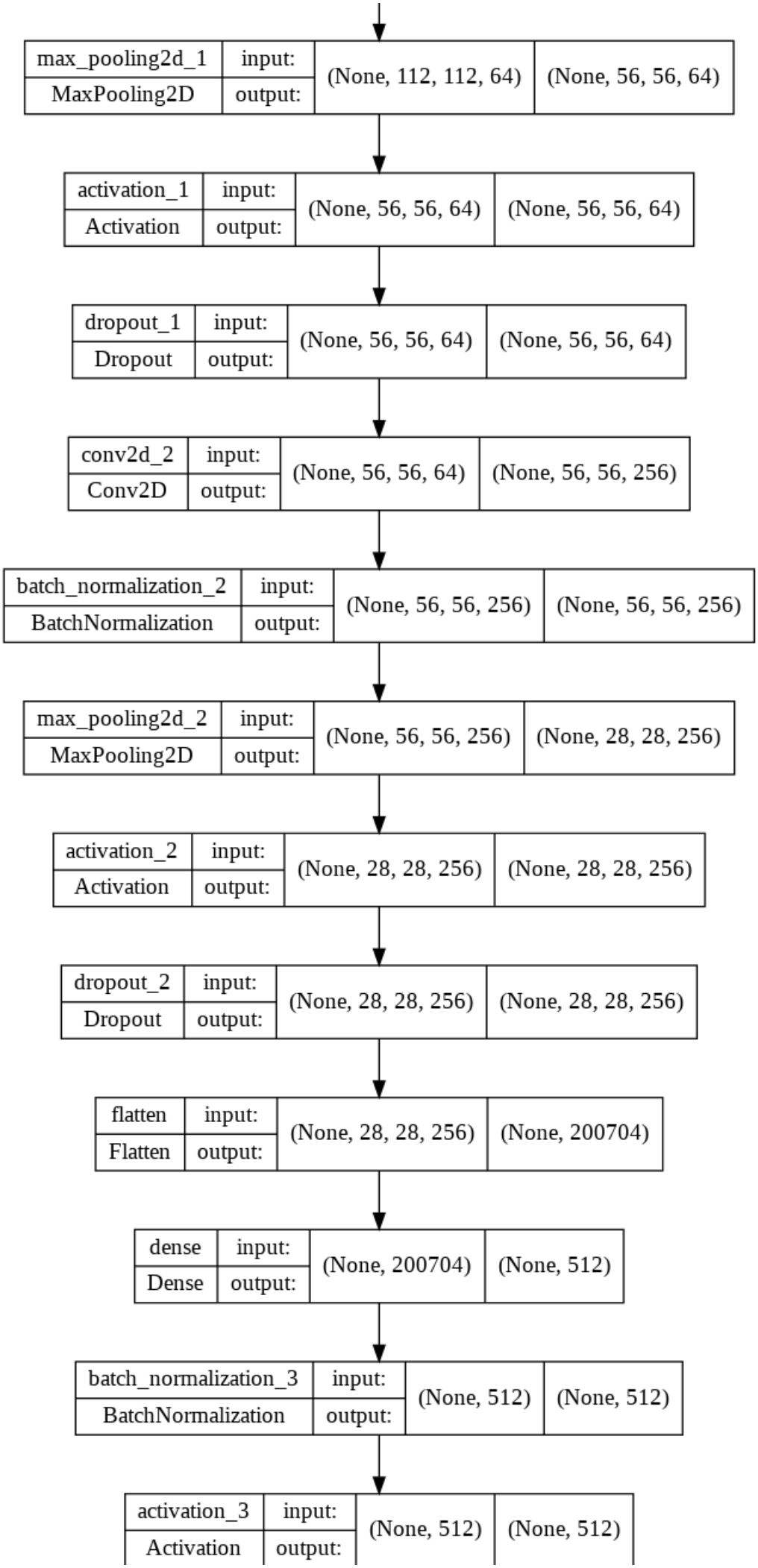
```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 224, 224, 32)      896

 batch_normalization (BatchN  (None, 224, 224, 32)     128
 ormalization)

 max_pooling2d (MaxPooling2D  (None, 112, 112, 32)     0
 )

 activation (Activation)     (None, 112, 112, 32)      0

 dropout (Dropout)           (None, 112, 112, 32)      0

 conv2d_1 (Conv2D)           (None, 112, 112, 64)      18496

 batch_normalization_1 (Batc  (None, 112, 112, 64)     256
 hNormalization)

 max_pooling2d_1 (MaxPooling  (None, 56, 56, 64)       0
 2D)

 activation_1 (Activation)   (None, 56, 56, 64)        0

 dropout_1 (Dropout)         (None, 56, 56, 64)        0

 conv2d_2 (Conv2D)           (None, 56, 56, 256)       147712

 batch_normalization_2 (Batc  (None, 56, 56, 256)      1024
 hNormalization)
```

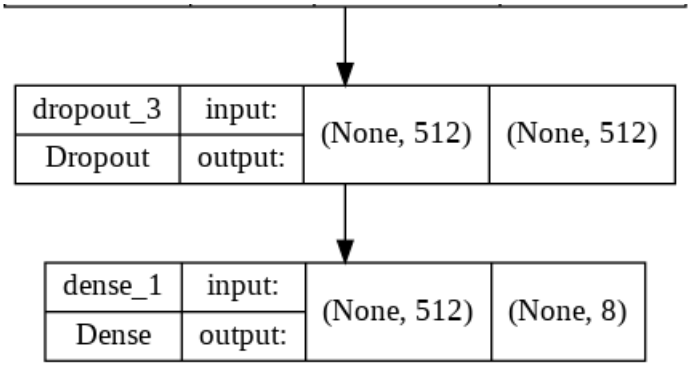
```
max_pooling2d_2 (MaxPooling  (None, 28, 28, 256)      0
2D)

activation_2 (Activation)   (None, 28, 28, 256)       0

dropout_2 (Dropout)         (None, 28, 28, 256)       0

flatten (Flatten)           (None, 200704)            0

dense (Dense)               (None, 512)               102760960

batch_normalization_3 (Batc  (None, 512)              2048
hNormalization)

activation_3 (Activation)   (None, 512)               0

dropout_3 (Dropout)         (None, 512)               0

dense_1 (Dense)             (None, 8)                 4104

=================================================================
Total params: 102,935,624
Trainable params: 102,933,896
Non-trainable params: 1,728
_________________________________________________________________
```

In [ ]:
```
plot_schema(model_1, 'model1_plot.png')
```

Out[ ]:




max_pooling2d_1 | MaxPooling2D | input: | output: | (None, 112, 112, 64) | (None, 56, 56, 64)
activation_1 | Activation | input: | output: | (None, 56, 56, 64) | (None, 56, 56, 64)
dropout_1 | Dropout | input: | output: | (None, 56, 56, 64) | (None, 56, 56, 64)
conv2d_2 | Conv2D | input: | output: | (None, 56, 56, 64) | (None, 56, 56, 256)
batch_normalization_2 | BatchNormalization | input: | output: | (None, 56, 56, 256) | (None, 56, 56, 256)
max_pooling2d_2 | MaxPooling2D | input: | output: | (None, 56, 56, 256) | (None, 28, 28, 256)
activation_2 | Activation | input: | output: | (None, 28, 28, 256) | (None, 28, 28, 256)
dropout_2 | Dropout | input: | output: | (None, 28, 28, 256) | (None, 28, 28, 256)
flatten | Flatten | input: | output: | (None, 28, 28, 256) | (None, 200704)
dense | Dense | input: | output: | (None, 200704) | (None, 512)
batch_normalization_3 | BatchNormalization | input: | output: | (None, 512) | (None, 512)
activation_3 | Activation | input: | output: | (None, 512) | (None, 512)



In [ ]:

```
# Compilación del modelo
print("[INFO]: Compilando el modelo...")
model_1.compile(loss='sparse_categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
```

```
[INFO]: Compilando el modelo...
```

In [ ]:

```
# Entrenamiento de la red
print("[INFO]: Entrenando la red...")
callback = EarlyStopping(monitor='val_loss', patience=20) # Callback EarlyStopping
n_epochs = 35
H = model_1.fit(x_tr, y_tr, epochs=n_epochs, callbacks=[callback], batch_size=64, verbose=1, validati
```

```
[INFO]: Entrenando la red...
Epoch 1/35
38/38 [==============================] - 9s 203ms/step - loss: 1.3601 - accuracy: 0.5755 - val_loss:
1.5927 - val_accuracy: 0.4347
Epoch 2/35
38/38 [==============================] - 7s 194ms/step - loss: 0.6095 - accuracy: 0.8002 - val_loss:
1.6780 - val_accuracy: 0.4347
Epoch 3/35
38/38 [==============================] - 7s 197ms/step - loss: 0.3109 - accuracy: 0.9024 - val_loss:
1.9247 - val_accuracy: 0.4562
Epoch 4/35
38/38 [==============================] - 8s 199ms/step - loss: 0.2041 - accuracy: 0.9392 - val_loss:
2.3213 - val_accuracy: 0.4909
Epoch 5/35
38/38 [==============================] - 8s 198ms/step - loss: 0.1168 - accuracy: 0.9690 - val_loss:
1.9655 - val_accuracy: 0.5190
Epoch 6/35
38/38 [==============================] - 7s 197ms/step - loss: 0.0706 - accuracy: 0.9810 - val_loss:
2.0780 - val_accuracy: 0.4463
Epoch 7/35
38/38 [==============================] - 7s 195ms/step - loss: 0.0498 - accuracy: 0.9909 - val_loss:
1.6588 - val_accuracy: 0.5190
Epoch 8/35
38/38 [==============================] - 7s 194ms/step - loss: 0.0409 - accuracy: 0.9909 - val_loss:
1.6658 - val_accuracy: 0.6050
Epoch 9/35
38/38 [==============================] - 7s 193ms/step - loss: 0.0368 - accuracy: 0.9917 - val_loss:
1.3354 - val_accuracy: 0.6281
Epoch 10/35
38/38 [==============================] - 7s 193ms/step - loss: 0.0331 - accuracy: 0.9930 - val_loss:
1.0363 - val_accuracy: 0.7372
Epoch 11/35
38/38 [==============================] - 7s 193ms/step - loss: 0.0353 - accuracy: 0.9934 - val_loss:
0.7908 - val_accuracy: 0.7603
Epoch 12/35
38/38 [==============================] - 7s 194ms/step - loss: 0.0157 - accuracy: 0.9967 - val_loss:
0.5667 - val_accuracy: 0.8562
Epoch 13/35
38/38 [==============================] - 7s 194ms/step - loss: 0.0170 - accuracy: 0.9975 - val_loss:
0.6631 - val_accuracy: 0.7934
Epoch 14/35
38/38 [==============================] - 7s 195ms/step - loss: 0.0156 - accuracy: 0.9963 - val_loss:
0.4207 - val_accuracy: 0.8975
Epoch 15/35
38/38 [==============================] - 7s 195ms/step - loss: 0.0146 - accuracy: 0.9946 - val_loss:
0.3559 - val_accuracy: 0.9124
Epoch 16/35
38/38 [==============================] - 7s 196ms/step - loss: 0.0126 - accuracy: 0.9983 - val_loss:
0.3204 - val_accuracy: 0.9190
Epoch 17/35
38/38 [==============================] - 7s 196ms/step - loss: 0.0103 - accuracy: 0.9979 - val_loss:
0.2422 - val_accuracy: 0.9488
Epoch 18/35
38/38 [==============================] - 7s 194ms/step - loss: 0.0121 - accuracy: 0.9975 - val_loss:
0.2484 - val_accuracy: 0.9521
Epoch 19/35
```

```
38/38 [==============================] - 7s 195ms/step - loss: 0.0120 - accuracy: 0.9967 - val_loss:
0.2091 - val_accuracy: 0.9570
Epoch 20/35
38/38 [==============================] - 7s 194ms/step - loss: 0.0085 - accuracy: 0.9988 - val_loss:
0.2155 - val_accuracy: 0.9554
Epoch 21/35
38/38 [==============================] - 7s 194ms/step - loss: 0.0089 - accuracy: 0.9983 - val_loss:
0.2597 - val_accuracy: 0.9488
Epoch 22/35
38/38 [==============================] - 7s 194ms/step - loss: 0.0066 - accuracy: 0.9983 - val_loss:
0.2156 - val_accuracy: 0.9587
Epoch 23/35
38/38 [==============================] - 7s 195ms/step - loss: 0.0098 - accuracy: 0.9975 - val_loss:
0.2276 - val_accuracy: 0.9537
Epoch 24/35
38/38 [==============================] - 7s 194ms/step - loss: 0.0060 - accuracy: 0.9988 - val_loss:
0.2324 - val_accuracy: 0.9554
Epoch 25/35
38/38 [==============================] - 7s 195ms/step - loss: 0.0125 - accuracy: 0.9975 - val_loss:
0.2548 - val_accuracy: 0.9521
Epoch 26/35
38/38 [==============================] - 7s 195ms/step - loss: 0.0057 - accuracy: 0.9992 - val_loss:
0.2376 - val_accuracy: 0.9537
Epoch 27/35
38/38 [==============================] - 7s 195ms/step - loss: 0.0068 - accuracy: 0.9988 - val_loss:
0.2414 - val_accuracy: 0.9504
Epoch 28/35
38/38 [==============================] - 7s 196ms/step - loss: 0.0039 - accuracy: 0.9988 - val_loss:
0.2511 - val_accuracy: 0.9488
Epoch 29/35
38/38 [==============================] - 7s 195ms/step - loss: 0.0072 - accuracy: 0.9988 - val_loss:
0.2319 - val_accuracy: 0.9537
Epoch 30/35
38/38 [==============================] - 7s 195ms/step - loss: 0.0070 - accuracy: 0.9983 - val_loss:
0.2522 - val_accuracy: 0.9504
Epoch 31/35
38/38 [==============================] - 7s 195ms/step - loss: 0.0103 - accuracy: 0.9971 - val_loss:
0.2477 - val_accuracy: 0.9438
Epoch 32/35
38/38 [==============================] - 7s 195ms/step - loss: 0.0066 - accuracy: 0.9996 - val_loss:
0.4092 - val_accuracy: 0.9190
Epoch 33/35
38/38 [==============================] - 7s 194ms/step - loss: 0.0063 - accuracy: 0.9979 - val_loss:
0.2170 - val_accuracy: 0.9521
Epoch 34/35
38/38 [==============================] - 7s 196ms/step - loss: 0.0060 - accuracy: 0.9992 - val_loss:
0.2471 - val_accuracy: 0.9455
Epoch 35/35
38/38 [==============================] - 7s 194ms/step - loss: 0.0101 - accuracy: 0.9983 - val_loss:
0.2115 - val_accuracy: 0.9537
```

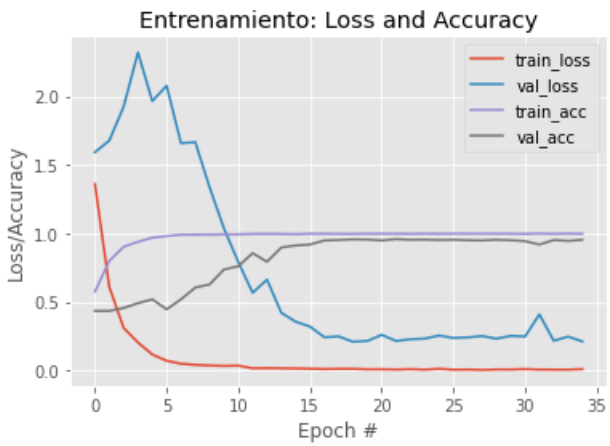
In [ ]:

```
# Almacenamos el modelo empleando la función model.save de Keras
model_1.save('deepCNN_model_1_class_weight.h5')
```

4.1.2 Ploteado de gráficas

In [ ]:

```
plot_learning_curves(H,epochs=n_epochs)
```



4.1.3 Análisis del modelo

In [ ]:

```
# Análisis del modelo
print("[INFO]: Evaluando el modelo...")
```

```
# Predicción (empleamos el mismo valor de batch_size que en training)
predictions = model_1.predict(x_te, batch_size=64) #(X)
# Informe para test
print(classification_report(y_test, predictions.argmax(axis=1), target_names=class_names)) #(X)
```

```
[INFO]: Evaluando el modelo...
              precision    recall  f1-score   support

         ALB       0.97      0.98      0.97       342
         BET       0.96      0.90      0.93        49
         DOL       0.94      1.00      0.97        15
         LAG       1.00      0.92      0.96        12
         NoF       0.92      0.93      0.92        98
       OTHER       0.95      0.95      0.95        59
       SHARK       1.00      1.00      1.00        36
         YFT       0.98      0.97      0.97       144

    accuracy                           0.96       755
   macro avg       0.96      0.95      0.96       755
weighted avg       0.96      0.96      0.96       755
```

4.2 Segundo modelo: Corrigiendo el desbalanceado de las clases originales: aplicando técnicas de '**CLASS-WEIGHTING**' y '**DATA AUGMENTATION**'

Redistribución de muestras: aplicando **CLASS-WEIGHT**

Como ya se ha comentado en puntos anteriores, una de las clases del dataset analizado contiene un número mucho más elevado de muestras que cualquiera de las restantes que componen dicho dataset.

Para empezar, nos encontramos con la siguiente situación de la información original:

In [ ]:
```
print ('Distribución por clases')
print (np.asarray((unique, counts)).T)
```

```
Distribución por clases
[[   0 1377]
 [   1  151]
 [   2  102]
 [   3   55]
 [   4  367]
 [   5  240]
 [   6  140]
 [   7  590]]
```

El siguiente paso es calcular los pesos que debemos asignar a las distintas clases para intentar 'equilibrar' la información. Para ello usaremos la función `compute_class_weight` que nos proporciona la librería `scikit-learn` .

In [ ]:
```
# usamos aquí la distribución de etiquetas previamente calculada
class_weights = class_weight.compute_class_weight(class_weight='balanced', classes=np.unique(y_tr), y
# creamos un diccionario con los pesos asignados a cada clase
tr_class_weights = dict(enumerate(class_weights))
tr_class_weights
```

Out[ ]:
```
{0: 0.27120736086175945,
 1: 2.650219298245614,
 2: 3.7299382716049383,
 3: 6.713888888888889,
 4: 1.0206925675675675,
 5: 1.6784722222222221,
 6: 2.6736725663716814,
 7: 0.6373945147679325}
```

In [ ]:
```
number_classes = len(np.unique(y_tr))
```

Redistribución de muestras: preparación de los ***ImageDataGenerators*** para aplicar técnicas de **DATA AUGMENTATION**

Se construye un ImageDataGenerator para realizar el proceso de de data augmentation sobre el conjunto de entrenamiento.

Para el conjunto de validación se utilizará un 'data generator' por defecto.

In [ ]:
```
train_gen = ImageDataGenerator(
```

```
    rotation_range=15, # grados de rotacion aleatoria
    width_shift_range=0.2, # fraccion del total (1) para mover la imagen
    height_shift_range=0.2, # fraccion del total (1) para mover la imagen
    horizontal_flip=True, # giro horizontal de imagenes (sobre eje vertical)
    # shear_range=0, # deslizamiento
    zoom_range=0.2, # rango de zoom
    # fill_mode='nearest', # como rellenar posibles nuevos pixeles
    # channel_shift_range=0.2 # cambios aleatorios en los canales de la imagen
)

val_gen = ImageDataGenerator() # no se aplica data-augmentation sobre el conjunto de validaciónn
```

In [ ]:

```
# para resetear el modelo (en caso de hacer varias pruebas)
tf.keras.backend.clear_session()
```

Aplicación de las técnicas de rebalanceado de muestras

In [ ]:

```
# MODELADO
# dimensiones de las imágenes de train
ds_shape = x_tr[0].shape

model_2 = Sequential()

#BASE MODEL
# Primer set de capas
model_2.add(Conv2D(32, (3,3), padding="same", input_shape=(ds_shape)))
model_2.add(BatchNormalization())
model_2.add(MaxPooling2D())
model_2.add(Activation("relu"))
model_2.add(Dropout(0.2))

# Segundo set de capas
model_2.add(Conv2D(64, (3,3), padding="same"))
model_2.add(BatchNormalization())
model_2.add(MaxPooling2D())
model_2.add(Activation("relu"))
model_2.add(Dropout(0.2))

# Tercer set de capas
model_2.add(Conv2D(256, (3,3), padding="same"))
model_2.add(BatchNormalization())
model_2.add(MaxPooling2D())
model_2.add(Activation("relu"))
model_2.add(Dropout(0.2))

#TOP MODEL
model_2.add(Flatten())
model_2.add(Dense(units = 512))
model_2.add(BatchNormalization())
model_2.add(Activation("relu"))
model_2.add(Dropout(0.5))
# Clasificador softmax
model_2.add(Dense(units = number_classes, activation = 'softmax'))
```

In [ ]:

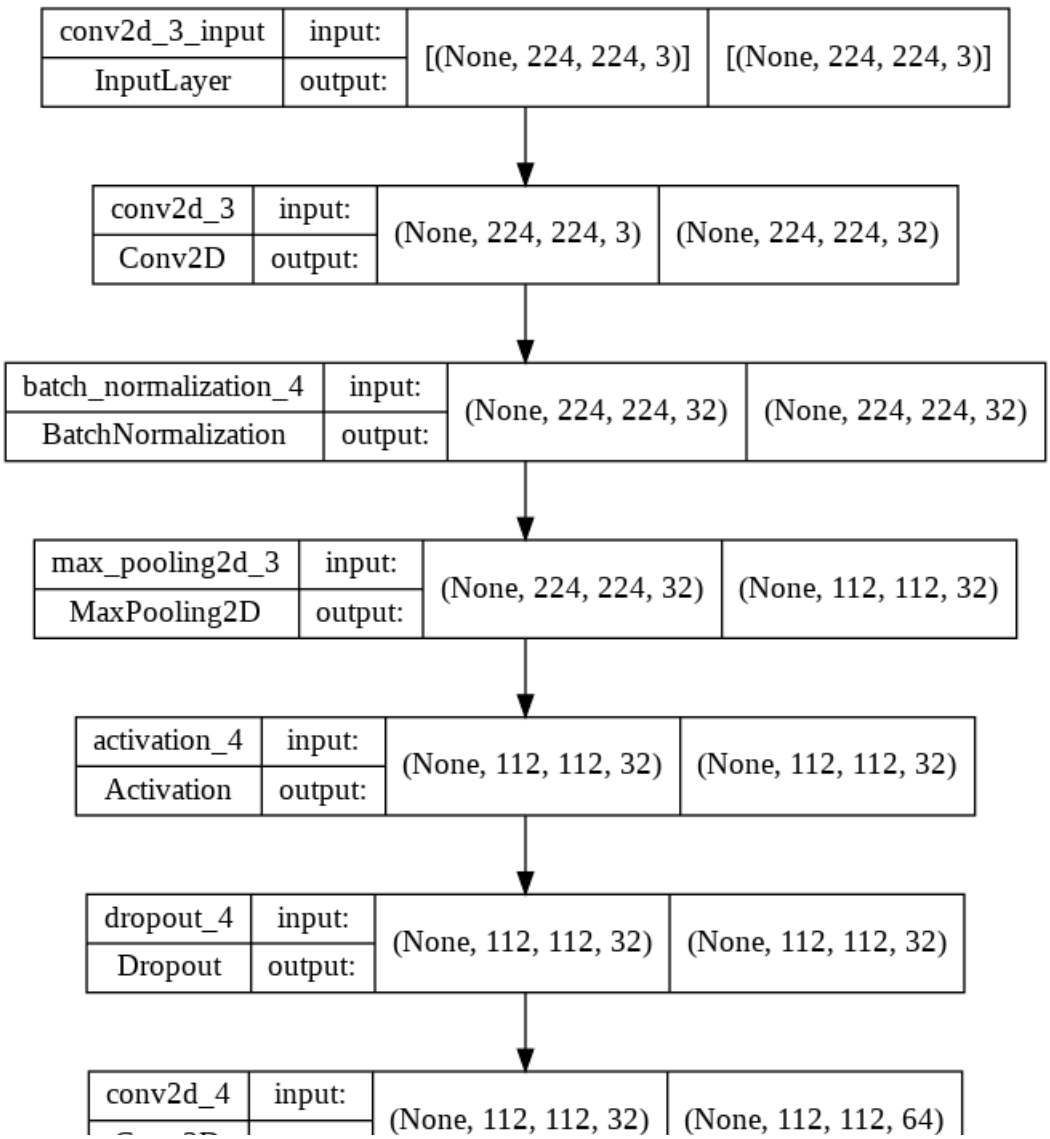
```
model_2.summary()
```

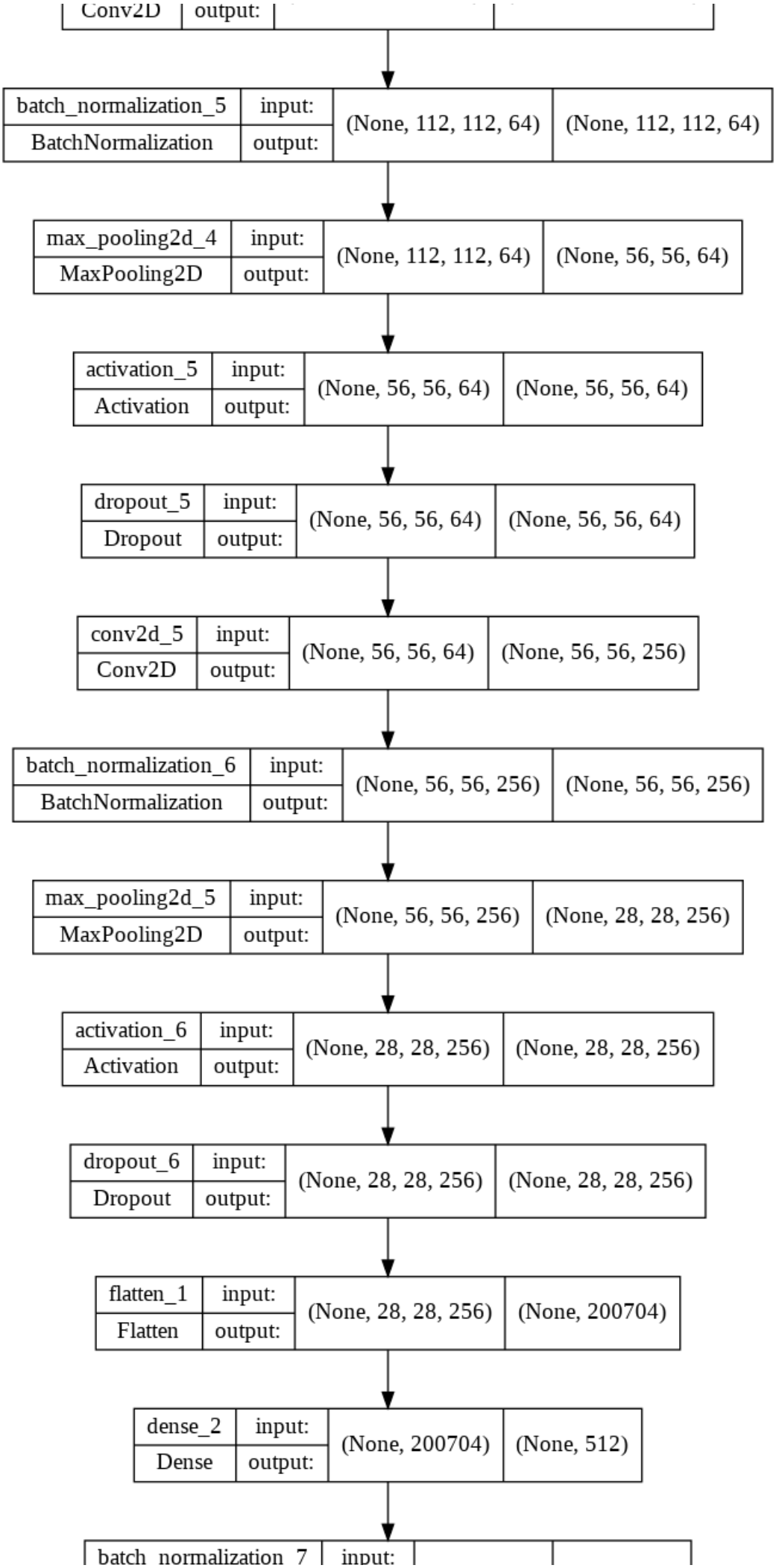
```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                 Output Shape              Param #
=================================================================
 conv2d_3 (Conv2D)            (None, 224, 224, 32)      896

 batch_normalization_4 (Batc  (None, 224, 224, 32)     128
 hNormalization)

 max_pooling2d_3 (MaxPooling  (None, 112, 112, 32)     0
 2D)

 activation_4 (Activation)    (None, 112, 112, 32)      0

 dropout_4 (Dropout)          (None, 112, 112, 32)      0

 conv2d_4 (Conv2D)            (None, 112, 112, 64)      18496

 batch_normalization_5 (Batc  (None, 112, 112, 64)     256
 hNormalization)

 max_pooling2d_4 (MaxPooling  (None, 56, 56, 64)       0
```

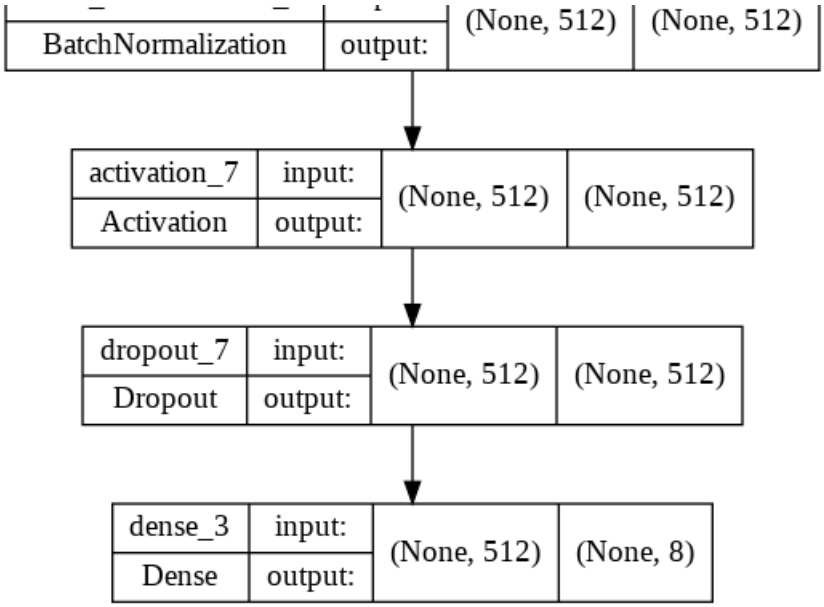
```
2D)

activation_5 (Activation)   (None, 56, 56, 64)        0

dropout_5 (Dropout)         (None, 56, 56, 64)        0

conv2d_5 (Conv2D)           (None, 56, 56, 256)       147712

batch_normalization_6 (Batc  (None, 56, 56, 256)      1024
hNormalization)

max_pooling2d_5 (MaxPooling  (None, 28, 28, 256)      0
2D)

activation_6 (Activation)   (None, 28, 28, 256)       0

dropout_6 (Dropout)         (None, 28, 28, 256)       0

flatten_1 (Flatten)         (None, 200704)            0

dense_2 (Dense)             (None, 512)               102760960

batch_normalization_7 (Batc  (None, 512)              2048
hNormalization)

activation_7 (Activation)   (None, 512)               0

dropout_7 (Dropout)         (None, 512)               0

dense_3 (Dense)             (None, 8)                 4104

=================================================================
Total params: 102,935,624
Trainable params: 102,933,896
Non-trainable params: 1,728
_________________________________________________________________
```

In [ ]:
```
plot_schema(model_2,'model2_plot.png')
```

Out[ ]:




Conv2D output:
batch_normalization_5 BatchNormalization input: output: (None, 112, 112, 64) (None, 112, 112, 64)
max_pooling2d_4 MaxPooling2D input: output: (None, 112, 112, 64) (None, 56, 56, 64)
activation_5 Activation input: output: (None, 56, 56, 64) (None, 56, 56, 64)
dropout_5 Dropout input: output: (None, 56, 56, 64) (None, 56, 56, 64)
conv2d_5 Conv2D input: output: (None, 56, 56, 64) (None, 56, 56, 256)
batch_normalization_6 BatchNormalization input: output: (None, 56, 56, 256) (None, 56, 56, 256)
max_pooling2d_5 MaxPooling2D input: output: (None, 56, 56, 256) (None, 28, 28, 256)
activation_6 Activation input: output: (None, 28, 28, 256) (None, 28, 28, 256)
dropout_6 Dropout input: output: (None, 28, 28, 256) (None, 28, 28, 256)
flatten_1 Flatten input: output: (None, 28, 28, 256) (None, 200704)
dense_2 Dense input: output: (None, 200704) (None, 512)
batch_normalization_7 input:



In [ ]:

```
# Compilar el modelo
print("[INFO]: Compilando el modelo...")
model_2.compile(loss='sparse_categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
```

```
[INFO]: Compilando el modelo...
```

In [ ]:

```
# Entrenamiento de la red ... PARAMETRIZACION
n_epochs = 35
n_batches = 64
```

In [ ]:

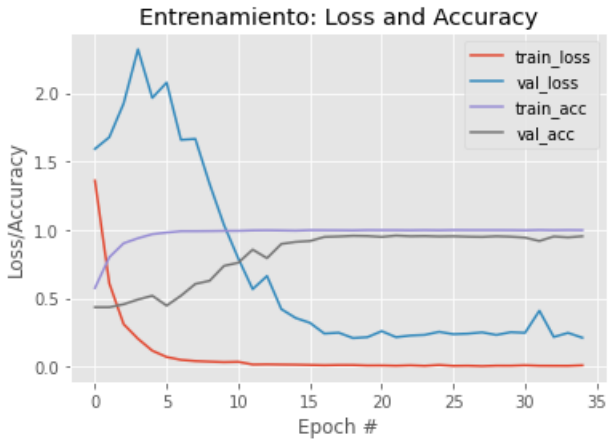
```
# Entrenamiento de la red... EJECUCIÓN
print("[INFO]: Entrenando la red...")
H2 = model_2.fit(train_gen.flow(x_tr, y_tr, batch_size = n_batches),
                 validation_data = val_gen.flow(x_val, y_val, batch_size=n_batches),
                 steps_per_epoch = x_tr.shape[0] / n_batches,
                 epochs = n_epochs,
                 class_weight = tr_class_weights,
                 verbose = 1)
```

```
[INFO]: Entrenando la red...
Epoch 1/35
37/37 [==============================] - 24s 605ms/step - loss: 2.2088 - accuracy: 0.2019 - val_loss:
2.3801 - val_accuracy: 0.0446
Epoch 2/35
37/37 [==============================] - 23s 603ms/step - loss: 1.9083 - accuracy: 0.2218 - val_loss:
2.3214 - val_accuracy: 0.0248
Epoch 3/35
37/37 [==============================] - 23s 605ms/step - loss: 1.7003 - accuracy: 0.2826 - val_loss:
2.2054 - val_accuracy: 0.0248
Epoch 4/35
37/37 [==============================] - 23s 601ms/step - loss: 1.5907 - accuracy: 0.3012 - val_loss:
2.4179 - val_accuracy: 0.0298
Epoch 5/35
37/37 [==============================] - 23s 606ms/step - loss: 1.5131 - accuracy: 0.3384 - val_loss:
2.2207 - val_accuracy: 0.0860
Epoch 6/35
37/37 [==============================] - 23s 608ms/step - loss: 1.4422 - accuracy: 0.3537 - val_loss:
2.1532 - val_accuracy: 0.1273
Epoch 7/35
37/37 [==============================] - 23s 609ms/step - loss: 1.3590 - accuracy: 0.3719 - val_loss:
2.2578 - val_accuracy: 0.1355
Epoch 8/35
37/37 [==============================] - 23s 601ms/step - loss: 1.3085 - accuracy: 0.3769 - val_loss:
2.1442 - val_accuracy: 0.1934
Epoch 9/35
37/37 [==============================] - 23s 609ms/step - loss: 1.2589 - accuracy: 0.4133 - val_loss:
2.2104 - val_accuracy: 0.2661
Epoch 10/35
37/37 [==============================] - 23s 605ms/step - loss: 1.1995 - accuracy: 0.4241 - val_loss:
2.4358 - val_accuracy: 0.1967
Epoch 11/35
37/37 [==============================] - 23s 604ms/step - loss: 1.1255 - accuracy: 0.4365 - val_loss:
1.9190 - val_accuracy: 0.2959
Epoch 12/35
37/37 [==============================] - 23s 606ms/step - loss: 1.1154 - accuracy: 0.4423 - val_loss:
2.1840 - val_accuracy: 0.2562
```

```
Epoch 13/35
37/37 [==============================] - 23s 609ms/step - loss: 1.0787 - accuracy: 0.4708 - val_loss:
1.6832 - val_accuracy: 0.3835
Epoch 14/35
37/37 [==============================] - 23s 600ms/step - loss: 1.0603 - accuracy: 0.4824 - val_loss:
1.9204 - val_accuracy: 0.3207
Epoch 15/35
37/37 [==============================] - 23s 608ms/step - loss: 1.0655 - accuracy: 0.4729 - val_loss:
1.6323 - val_accuracy: 0.3818
Epoch 16/35
37/37 [==============================] - 23s 607ms/step - loss: 0.9876 - accuracy: 0.5039 - val_loss:
1.2985 - val_accuracy: 0.5107
Epoch 17/35
37/37 [==============================] - 23s 607ms/step - loss: 0.9748 - accuracy: 0.5023 - val_loss:
1.2213 - val_accuracy: 0.5471
Epoch 18/35
37/37 [==============================] - 23s 604ms/step - loss: 0.9460 - accuracy: 0.5101 - val_loss:
1.3240 - val_accuracy: 0.5074
Epoch 19/35
37/37 [==============================] - 23s 604ms/step - loss: 0.9318 - accuracy: 0.5139 - val_loss:
1.5720 - val_accuracy: 0.4413
Epoch 20/35
37/37 [==============================] - 23s 610ms/step - loss: 0.8838 - accuracy: 0.5267 - val_loss:
1.1180 - val_accuracy: 0.5983
Epoch 21/35
37/37 [==============================] - 23s 608ms/step - loss: 0.8821 - accuracy: 0.5478 - val_loss:
1.0631 - val_accuracy: 0.5917
Epoch 22/35
37/37 [==============================] - 23s 608ms/step - loss: 0.8079 - accuracy: 0.5304 - val_loss:
1.0510 - val_accuracy: 0.6066
Epoch 23/35
37/37 [==============================] - 24s 620ms/step - loss: 0.8466 - accuracy: 0.5532 - val_loss:
1.9016 - val_accuracy: 0.3653
Epoch 24/35
37/37 [==============================] - 23s 615ms/step - loss: 0.8535 - accuracy: 0.5465 - val_loss:
0.9934 - val_accuracy: 0.6149
Epoch 25/35
37/37 [==============================] - 23s 603ms/step - loss: 0.8134 - accuracy: 0.5792 - val_loss:
1.0356 - val_accuracy: 0.6248
Epoch 26/35
37/37 [==============================] - 23s 604ms/step - loss: 0.7868 - accuracy: 0.5635 - val_loss:
1.2385 - val_accuracy: 0.5421
Epoch 27/35
37/37 [==============================] - 23s 603ms/step - loss: 0.7767 - accuracy: 0.5780 - val_loss:
0.9877 - val_accuracy: 0.6529
Epoch 28/35
37/37 [==============================] - 23s 600ms/step - loss: 0.7432 - accuracy: 0.5854 - val_loss:
1.0029 - val_accuracy: 0.6165
Epoch 29/35
37/37 [==============================] - 23s 605ms/step - loss: 0.7813 - accuracy: 0.5689 - val_loss:
1.2099 - val_accuracy: 0.6364
Epoch 30/35
37/37 [==============================] - 23s 606ms/step - loss: 0.7464 - accuracy: 0.5854 - val_loss:
1.5903 - val_accuracy: 0.4149
Epoch 31/35
37/37 [==============================] - 23s 598ms/step - loss: 0.7285 - accuracy: 0.5921 - val_loss:
0.8379 - val_accuracy: 0.6959
Epoch 32/35
37/37 [==============================] - 23s 605ms/step - loss: 0.7119 - accuracy: 0.6136 - val_loss:
0.8629 - val_accuracy: 0.6893
Epoch 33/35
37/37 [==============================] - 23s 605ms/step - loss: 0.7432 - accuracy: 0.6177 - val_loss:
1.0700 - val_accuracy: 0.5901
Epoch 34/35
37/37 [==============================] - 23s 605ms/step - loss: 0.7293 - accuracy: 0.5883 - val_loss:
1.3638 - val_accuracy: 0.5802
Epoch 35/35
37/37 [==============================] - 23s 606ms/step - loss: 0.6404 - accuracy: 0.6471 - val_loss:
0.8853 - val_accuracy: 0.7041
```

In [ ]:

```
plot_learning_curves(H,epochs=n_epochs)
```



In [ ]:
```
# Evaluación del modelo
print("[INFO]: Evaluando el modelo...")
# Predicción sobre el conjunto test original
# predictions = model_2.predict(x_te, batch_size=64) #(X)
predictions = model_2.predict(x_te, batch_size=32) #(X)
print(classification_report(y_test, predictions.argmax(axis=1), target_names=class_names)) #(X)
```

```
[INFO]: Evaluando el modelo...
              precision    recall  f1-score   support

         ALB       0.85      0.68      0.76       342
         BET       0.61      0.80      0.69        49
         DOL       0.43      1.00      0.60        15
         LAG       0.48      0.92      0.63        12
         NoF       0.75      0.79      0.77        98
       OTHER       0.64      0.63      0.63        59
       SHARK       0.38      1.00      0.55        36
         YFT       0.71      0.49      0.58       144

    accuracy                           0.69       755
   macro avg       0.60      0.79      0.65       755
weighted avg       0.74      0.69      0.70       755
```

COMENTARIO

Tras aplicar las técnicas para ajustar el balance de muestras da la impresión de que la convergencia empeora sobre lo analizado inicialmente sobre los datos originales 'en bruto'. Esto puede ser ocasionado por un aprendizaje del modelo sobre las clases mayoritarias (en concreto sobre la clase 0-ALB), dando la sensación, posiblemente errónea, de un ajuste en las predicciones casi perfecto.

Por último, mencionar el hecho que supone el uso del Data Augmentation como técnica para aumentar la variabilidad de las muestras y, gracias a ello, reconducir los resultados hacia unos datos de predicción más 'creíbles'/'realistas'.

---

5. Probando otras topologías de red.

In [ ]:
```
# para resetear el modelo (en caso de hacer varias pruebas)
tf.keras.backend.clear_session()
```

In [ ]:
```
# Eliminando el tercer set de capas
ds_shape = x_tr[0].shape
print (ds_shape)

model_2b = Sequential()

#BASE MODEL
# Primer set de capas
model_2b.add(Conv2D(32, (3,3), padding="same", input_shape=(ds_shape)))
model_2b.add(BatchNormalization())
model_2b.add(MaxPooling2D())
model_2b.add(Activation("relu"))
model_2b.add(Dropout(0.2))

# Segundo set de capas
```

```
model_2b.add(Conv2D(64, (3,3), padding="same"))
model_2b.add(BatchNormalization())
model_2b.add(MaxPooling2D())
model_2b.add(Activation("relu"))
model_2b.add(Dropout(0.2))


#TOP MODEL
model_2b.add(Flatten())
model_2b.add(Dense(units = 512))
model_2b.add(BatchNormalization())
model_2b.add(Activation("relu"))
model_2b.add(Dropout(0.3))
# Clasificador softmax
model_2b.add(Dense(units = number_classes, activation = 'softmax'))
```

```
(224, 224, 3)
```

In [ ]:

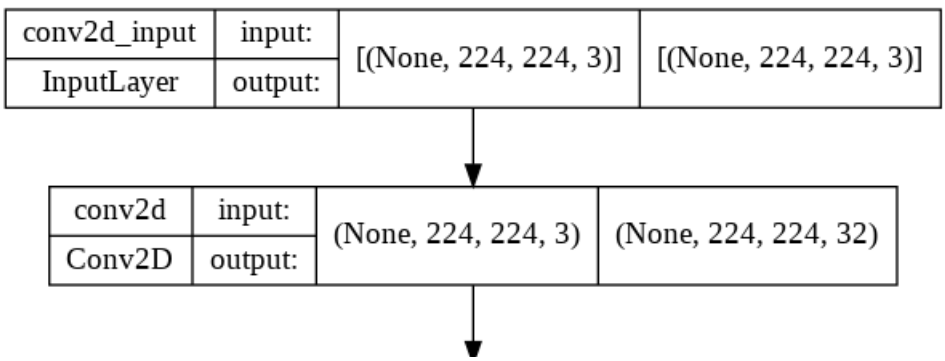
```
model_2b.summary()
```

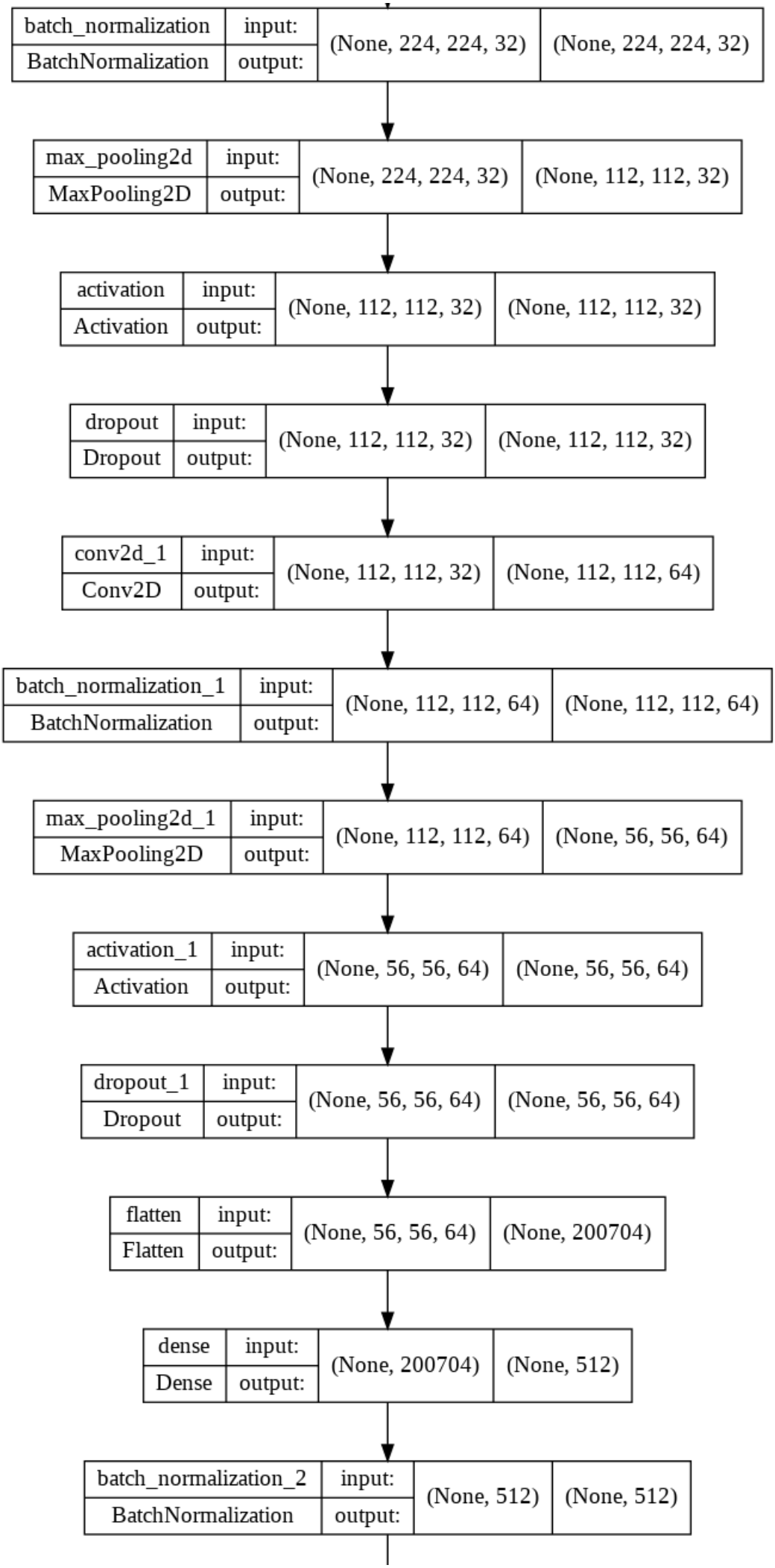
```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 224, 224, 32)      896

 batch_normalization (BatchN  (None, 224, 224, 32)     128
 ormalization)

 max_pooling2d (MaxPooling2D  (None, 112, 112, 32)     0
 )

 activation (Activation)     (None, 112, 112, 32)      0

 dropout (Dropout)           (None, 112, 112, 32)      0

 conv2d_1 (Conv2D)           (None, 112, 112, 64)      18496

 batch_normalization_1 (Batc  (None, 112, 112, 64)     256
 hNormalization)

 max_pooling2d_1 (MaxPooling  (None, 56, 56, 64)       0
 2D)

 activation_1 (Activation)   (None, 56, 56, 64)        0

 dropout_1 (Dropout)         (None, 56, 56, 64)        0

 flatten (Flatten)           (None, 200704)            0

 dense (Dense)               (None, 512)               102760960

 batch_normalization_2 (Batc  (None, 512)              2048
 hNormalization)

 activation_2 (Activation)   (None, 512)               0

 dropout_2 (Dropout)         (None, 512)               0

 dense_1 (Dense)             (None, 8)                 4104

=================================================================
Total params: 102,786,888
Trainable params: 102,785,672
Non-trainable params: 1,216
_________________________________________________________________
```

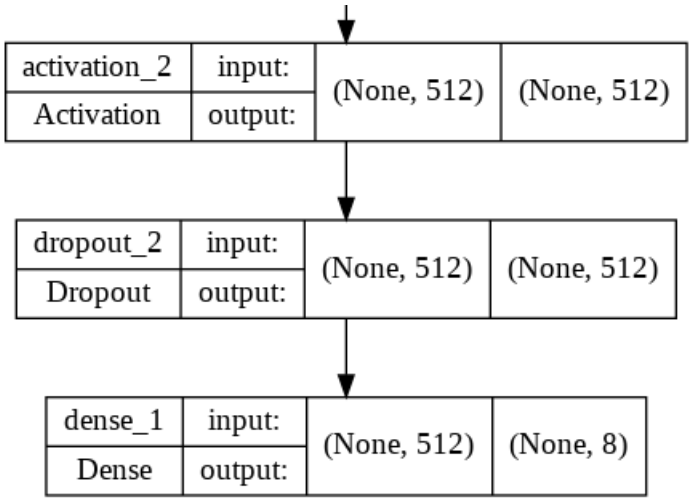
In [ ]:

```
plot_schema(model_2b, 'model_2b_plot.png')
```

Out[ ]:




batch_normalization
BatchNormalization
input:
output:
(None, 224, 224, 32)
(None, 224, 224, 32)
max_pooling2d
MaxPooling2D
input:
output:
(None, 224, 224, 32)
(None, 112, 112, 32)
activation
Activation
input:
output:
(None, 112, 112, 32)
(None, 112, 112, 32)
dropout
Dropout
input:
output:
(None, 112, 112, 32)
(None, 112, 112, 32)
conv2d_1
Conv2D
input:
output:
(None, 112, 112, 32)
(None, 112, 112, 64)
batch_normalization_1
BatchNormalization
input:
output:
(None, 112, 112, 64)
(None, 112, 112, 64)
max_pooling2d_1
MaxPooling2D
input:
output:
(None, 112, 112, 64)
(None, 56, 56, 64)
activation_1
Activation
input:
output:
(None, 56, 56, 64)
(None, 56, 56, 64)
dropout_1
Dropout
input:
output:
(None, 56, 56, 64)
(None, 56, 56, 64)
flatten
Flatten
input:
output:
(None, 56, 56, 64)
(None, 200704)
dense
Dense
input:
output:
(None, 200704)
(None, 512)
batch_normalization_2
BatchNormalization
input:
output:
(None, 512)
(None, 512)



In [ ]:

```
# Compilación del modelo
print("[INFO]: Compilando el modelo...")
model_2b.compile(loss='sparse_categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
```

```
[INFO]: Compilando el modelo...
```

In [ ]:

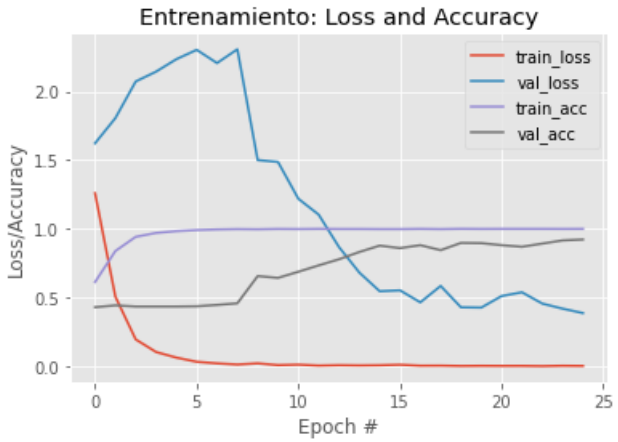
```
# Entrenamiento de la red
print("[INFO]: Entrenando la red...")
n_epochs = 25
H = model_2b.fit(x_tr, y_tr, epochs=n_epochs, batch_size=64, verbose=1, validation_data=(x_val, y_va
```

```
[INFO]: Entrenando la red...
Epoch 1/25
38/38 [==============================] - 7s 157ms/step - loss: 1.2588 - accuracy: 0.6127 - val_loss:
1.6226 - val_accuracy: 0.4298
Epoch 2/25
38/38 [==============================] - 6s 145ms/step - loss: 0.5066 - accuracy: 0.8386 - val_loss:
1.8058 - val_accuracy: 0.4430
Epoch 3/25
38/38 [==============================] - 6s 147ms/step - loss: 0.1952 - accuracy: 0.9417 - val_loss:
2.0722 - val_accuracy: 0.4347
Epoch 4/25
38/38 [==============================] - 6s 147ms/step - loss: 0.1036 - accuracy: 0.9702 - val_loss:
2.1437 - val_accuracy: 0.4347
Epoch 5/25
38/38 [==============================] - 6s 148ms/step - loss: 0.0628 - accuracy: 0.9830 - val_loss:
2.2344 - val_accuracy: 0.4347
Epoch 6/25
38/38 [==============================] - 6s 148ms/step - loss: 0.0325 - accuracy: 0.9909 - val_loss:
2.3025 - val_accuracy: 0.4364
Epoch 7/25
38/38 [==============================] - 6s 147ms/step - loss: 0.0208 - accuracy: 0.9954 - val_loss:
2.2061 - val_accuracy: 0.4463
Epoch 8/25
38/38 [==============================] - 6s 147ms/step - loss: 0.0127 - accuracy: 0.9979 - val_loss:
2.3065 - val_accuracy: 0.4579
Epoch 9/25
38/38 [==============================] - 6s 145ms/step - loss: 0.0210 - accuracy: 0.9967 - val_loss:
1.5001 - val_accuracy: 0.6562
Epoch 10/25
38/38 [==============================] - 6s 146ms/step - loss: 0.0083 - accuracy: 0.9992 - val_loss:
1.4865 - val_accuracy: 0.6430
Epoch 11/25
38/38 [==============================] - 6s 145ms/step - loss: 0.0117 - accuracy: 0.9983 - val_loss:
1.2188 - val_accuracy: 0.6876
Epoch 12/25
38/38 [==============================] - 6s 146ms/step - loss: 0.0049 - accuracy: 0.9996 - val_loss:
1.1032 - val_accuracy: 0.7339
Epoch 13/25
38/38 [==============================] - 6s 145ms/step - loss: 0.0077 - accuracy: 0.9988 - val_loss:
0.8671 - val_accuracy: 0.7785
Epoch 14/25
38/38 [==============================] - 6s 145ms/step - loss: 0.0061 - accuracy: 0.9988 - val_loss:
0.6818 - val_accuracy: 0.8314
Epoch 15/25
38/38 [==============================] - 5s 145ms/step - loss: 0.0073 - accuracy: 0.9979 - val_loss:
0.5460 - val_accuracy: 0.8777
Epoch 16/25
38/38 [==============================] - 6s 146ms/step - loss: 0.0110 - accuracy: 0.9979 - val_loss:
0.5524 - val_accuracy: 0.8595
Epoch 17/25
38/38 [==============================] - 5s 145ms/step - loss: 0.0041 - accuracy: 0.9996 - val_loss:
```

```
0.4648 - val_accuracy: 0.8810
Epoch 18/25
38/38 [==============================] - 6s 146ms/step - loss: 0.0045 - accuracy: 0.9979 - val_loss:
0.5844 - val_accuracy: 0.8446
Epoch 19/25
38/38 [==============================] - 6s 147ms/step - loss: 0.0020 - accuracy: 1.0000 - val_loss:
0.4288 - val_accuracy: 0.8975
Epoch 20/25
38/38 [==============================] - 6s 146ms/step - loss: 0.0031 - accuracy: 0.9992 - val_loss:
0.4268 - val_accuracy: 0.8959
Epoch 21/25
38/38 [==============================] - 6s 146ms/step - loss: 0.0024 - accuracy: 0.9996 - val_loss:
0.5110 - val_accuracy: 0.8810
Epoch 22/25
38/38 [==============================] - 6s 146ms/step - loss: 0.0026 - accuracy: 0.9996 - val_loss:
0.5382 - val_accuracy: 0.8694
Epoch 23/25
38/38 [==============================] - 6s 145ms/step - loss: 9.2965e-04 - accuracy: 0.9996 - val_lo
ss: 0.4559 - val_accuracy: 0.8926
Epoch 24/25
38/38 [==============================] - 6s 146ms/step - loss: 0.0035 - accuracy: 0.9992 - val_loss:
0.4190 - val_accuracy: 0.9157
Epoch 25/25
38/38 [==============================] - 5s 145ms/step - loss: 0.0022 - accuracy: 0.9996 - val_loss:
0.3869 - val_accuracy: 0.9223
```

In [ ]:

```
plot_learning_curves(H,epochs=n_epochs)
```



In [ ]:

```
# Análisis del modelo
print("[INFO]: Evaluando el modelo...")
# Predicción (empleamos el mismo valor de batch_size que en training)
predictions = model_2b.predict(x_te, batch_size=64) #(X)
# Informe para test
print(classification_report(y_test, predictions.argmax(axis=1), target_names=class_names)) #(X)
```

```
[INFO]: Evaluando el modelo...
              precision    recall  f1-score   support

         ALB       0.92      0.96      0.94       342
         BET       1.00      0.67      0.80        49
         DOL       1.00      0.60      0.75        15
         LAG       1.00      0.92      0.96        12
         NoF       0.98      0.83      0.90        98
       OTHER       0.98      0.95      0.97        59
       SHARK       1.00      0.97      0.99        36
         YFT       0.84      0.99      0.91       144

    accuracy                           0.92       755
   macro avg       0.97      0.86      0.90       755
weighted avg       0.93      0.92      0.92       755
```

In [ ]:

```
# Probando filtro no cuadrado de 5x3
ds_shape = x_tr[0].shape
print (ds_shape)

model_3 = Sequential()

#BASE MODEL
# Primer set de capas
model_3.add(Conv2D(64, (3,3), padding="same", input_shape=(ds_shape)))
model_3.add(BatchNormalization())
model_3.add(MaxPooling2D())
```

```
model_3.add(Activation("relu"))
model_3.add(Dropout(0.2))

# Segundo set de capas
model_3.add(Conv2D(64, (5,3), padding="same"))
model_3.add(BatchNormalization())
model_3.add(MaxPooling2D())
model_3.add(Activation("relu"))
model_3.add(Dropout(0.2))


#TOP MODEL
model_3.add(Flatten())
model_3.add(Dense(units = 512))
model_3.add(BatchNormalization())
model_3.add(Activation("relu"))
model_3.add(Dropout(0.5))
# Clasificador softmax
model_3.add(Dense(units = number_classes, activation = 'softmax'))
```

```
(224, 224, 3)
```

In [ ]:

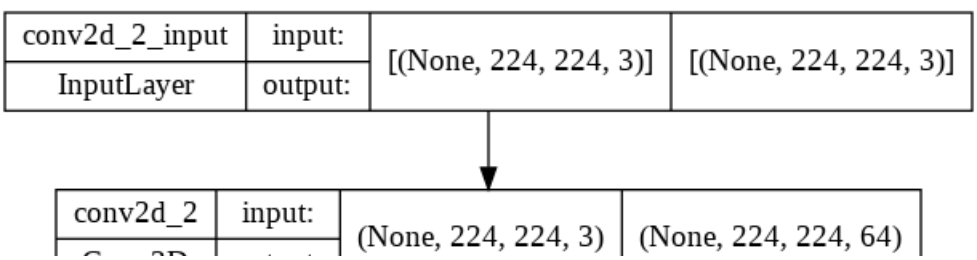
```
model_3.summary()
```

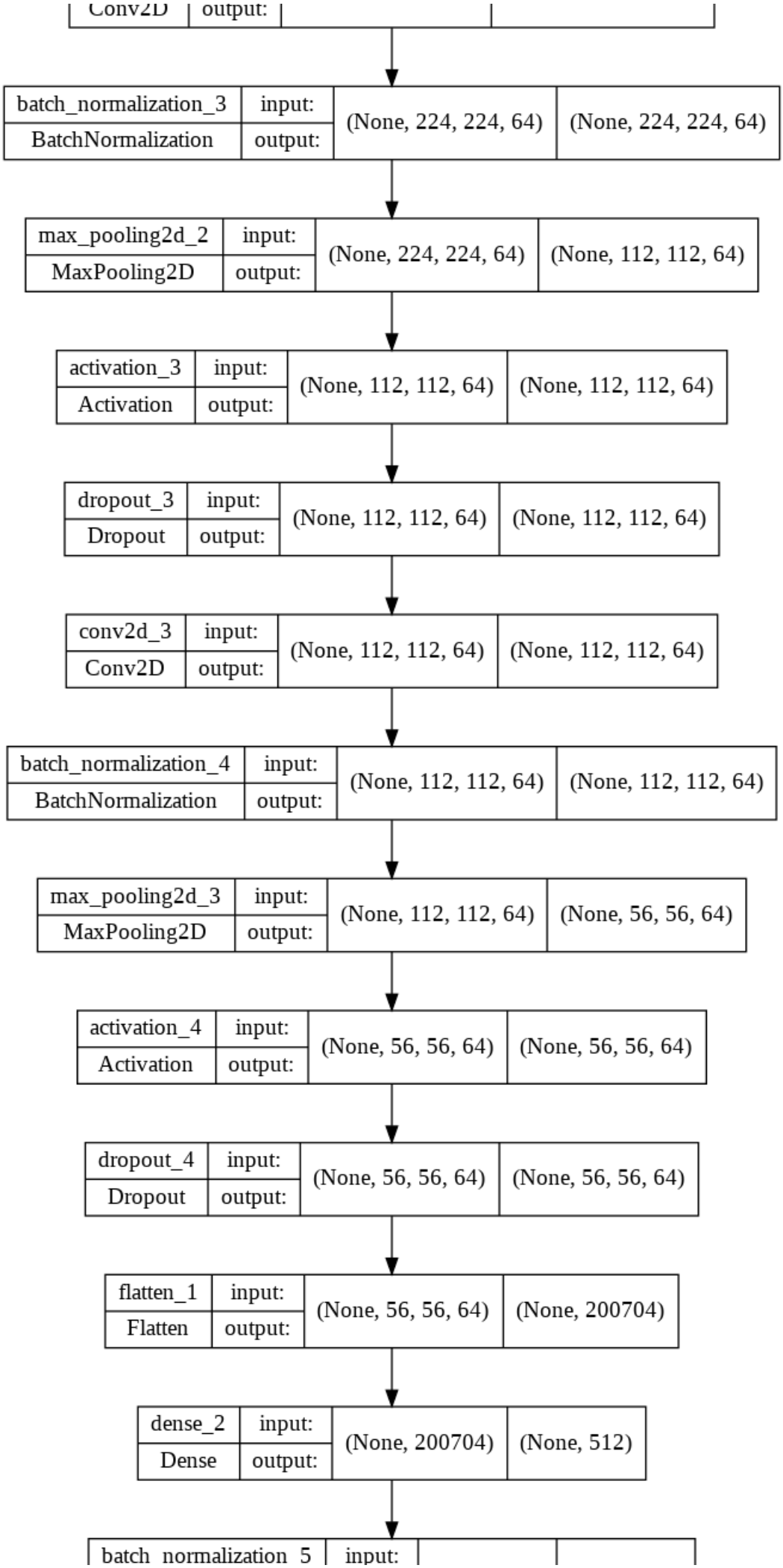
```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                 Output Shape              Param #
=================================================================
 conv2d_2 (Conv2D)            (None, 224, 224, 64)      1792

 batch_normalization_3 (Batc  (None, 224, 224, 64)     256
 hNormalization)

 max_pooling2d_2 (MaxPooling  (None, 112, 112, 64)     0
 2D)

 activation_3 (Activation)    (None, 112, 112, 64)      0

 dropout_3 (Dropout)          (None, 112, 112, 64)      0

 conv2d_3 (Conv2D)            (None, 112, 112, 64)      61504

 batch_normalization_4 (Batc  (None, 112, 112, 64)     256
 hNormalization)

 max_pooling2d_3 (MaxPooling  (None, 56, 56, 64)       0
 2D)

 activation_4 (Activation)    (None, 56, 56, 64)        0

 dropout_4 (Dropout)          (None, 56, 56, 64)        0

 flatten_1 (Flatten)          (None, 200704)            0

 dense_2 (Dense)              (None, 512)               102760960

 batch_normalization_5 (Batc  (None, 512)              2048
 hNormalization)

 activation_5 (Activation)    (None, 512)               0

 dropout_5 (Dropout)          (None, 512)               0

 dense_3 (Dense)              (None, 8)                 4104

=================================================================
Total params: 102,830,920
Trainable params: 102,829,640
Non-trainable params: 1,280
_________________________________________________________________
```

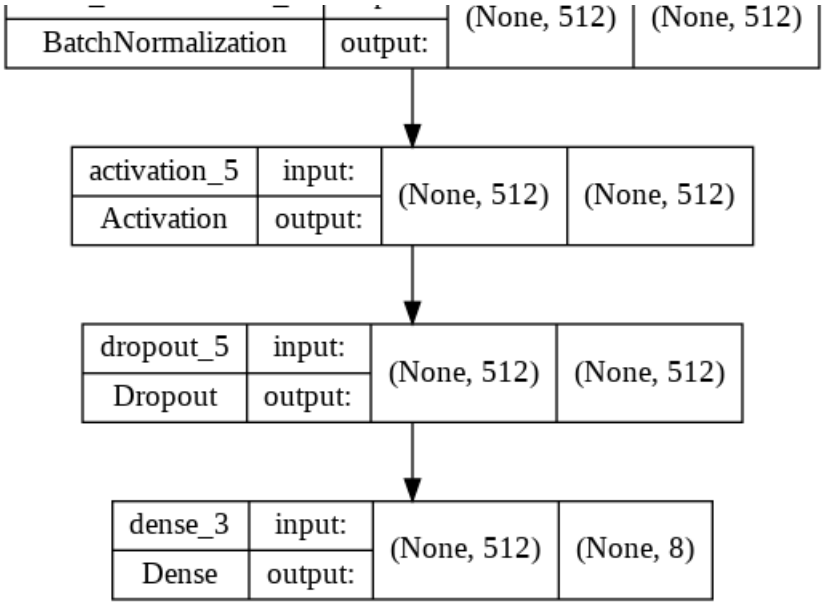
In [ ]:

```
plot_schema(model_3,'model3_plot.png')
```

Out[ ]:




Conv2D
output:
batch_normalization_3
BatchNormalization
input:
output:
(None, 224, 224, 64)
(None, 224, 224, 64)
max_pooling2d_2
MaxPooling2D
input:
output:
(None, 224, 224, 64)
(None, 112, 112, 64)
activation_3
Activation
input:
output:
(None, 112, 112, 64)
(None, 112, 112, 64)
dropout_3
Dropout
input:
output:
(None, 112, 112, 64)
(None, 112, 112, 64)
conv2d_3
Conv2D
input:
output:
(None, 112, 112, 64)
(None, 112, 112, 64)
batch_normalization_4
BatchNormalization
input:
output:
(None, 112, 112, 64)
(None, 112, 112, 64)
max_pooling2d_3
MaxPooling2D
input:
output:
(None, 112, 112, 64)
(None, 56, 56, 64)
activation_4
Activation
input:
output:
(None, 56, 56, 64)
(None, 56, 56, 64)
dropout_4
Dropout
input:
output:
(None, 56, 56, 64)
(None, 56, 56, 64)
flatten_1
Flatten
input:
output:
(None, 56, 56, 64)
(None, 200704)
dense_2
Dense
input:
output:
(None, 200704)
(None, 512)
batch_normalization_5
input:



In [ ]:

```
# Compilación del modelo
print("[INFO]: Compilando el modelo...")
model_3.compile(loss='sparse_categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
```

```
[INFO]: Compilando el modelo...
```

In [ ]:

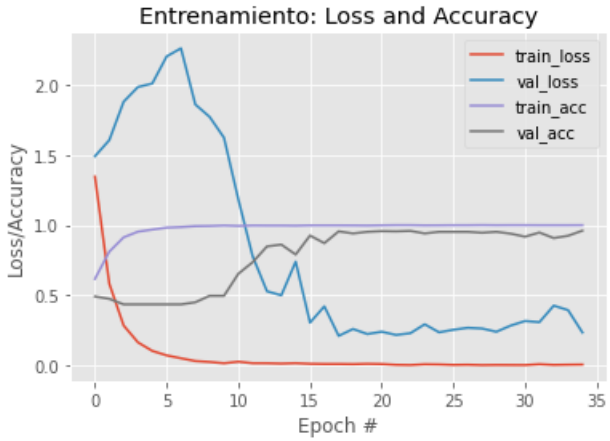
```
# Entrenamiento de la red
print("[INFO]: Entrenando la red...")
n_epochs = 35
H = model_3.fit(x_tr, y_tr, epochs=n_epochs, batch_size=64, verbose=1, validation_data=(x_val, y_val
```

```
[INFO]: Entrenando la red...
Epoch 1/35
38/38 [==============================] - 6s 130ms/step - loss: 1.3454 - accuracy: 0.6169 - val_loss:
1.4910 - val_accuracy: 0.4909
Epoch 2/35
38/38 [==============================] - 5s 121ms/step - loss: 0.5825 - accuracy: 0.8105 - val_loss:
1.6058 - val_accuracy: 0.4744
Epoch 3/35
38/38 [==============================] - 5s 122ms/step - loss: 0.2841 - accuracy: 0.9127 - val_loss:
1.8796 - val_accuracy: 0.4347
Epoch 4/35
38/38 [==============================] - 5s 121ms/step - loss: 0.1638 - accuracy: 0.9532 - val_loss:
1.9846 - val_accuracy: 0.4347
Epoch 5/35
38/38 [==============================] - 5s 121ms/step - loss: 0.1028 - accuracy: 0.9681 - val_loss:
2.0098 - val_accuracy: 0.4347
Epoch 6/35
38/38 [==============================] - 5s 121ms/step - loss: 0.0710 - accuracy: 0.9818 - val_loss:
2.2027 - val_accuracy: 0.4347
Epoch 7/35
38/38 [==============================] - 5s 122ms/step - loss: 0.0505 - accuracy: 0.9859 - val_loss:
2.2608 - val_accuracy: 0.4347
Epoch 8/35
38/38 [==============================] - 5s 121ms/step - loss: 0.0307 - accuracy: 0.9926 - val_loss:
1.8603 - val_accuracy: 0.4496
Epoch 9/35
38/38 [==============================] - 5s 121ms/step - loss: 0.0242 - accuracy: 0.9938 - val_loss:
1.7716 - val_accuracy: 0.4959
Epoch 10/35
38/38 [==============================] - 5s 122ms/step - loss: 0.0150 - accuracy: 0.9971 - val_loss:
1.6231 - val_accuracy: 0.4959
Epoch 11/35
38/38 [==============================] - 5s 121ms/step - loss: 0.0261 - accuracy: 0.9942 - val_loss:
1.1844 - val_accuracy: 0.6529
Epoch 12/35
38/38 [==============================] - 5s 122ms/step - loss: 0.0153 - accuracy: 0.9971 - val_loss:
0.7775 - val_accuracy: 0.7355
Epoch 13/35
38/38 [==============================] - 5s 121ms/step - loss: 0.0151 - accuracy: 0.9967 - val_loss:
0.5279 - val_accuracy: 0.8479
Epoch 14/35
38/38 [==============================] - 5s 120ms/step - loss: 0.0127 - accuracy: 0.9967 - val_loss:
0.4996 - val_accuracy: 0.8612
Epoch 15/35
38/38 [==============================] - 5s 120ms/step - loss: 0.0154 - accuracy: 0.9959 - val_loss:
0.7391 - val_accuracy: 0.7901
Epoch 16/35
```

```
38/38 [==============================] - 5s 120ms/step - loss: 0.0111 - accuracy: 0.9979 - val_loss:
0.3046 - val_accuracy: 0.9256
Epoch 17/35
38/38 [==============================] - 5s 121ms/step - loss: 0.0095 - accuracy: 0.9979 - val_loss:
0.4206 - val_accuracy: 0.8711
Epoch 18/35
38/38 [==============================] - 5s 121ms/step - loss: 0.0096 - accuracy: 0.9979 - val_loss:
0.2101 - val_accuracy: 0.9554
Epoch 19/35
38/38 [==============================] - 5s 121ms/step - loss: 0.0085 - accuracy: 0.9975 - val_loss:
0.2592 - val_accuracy: 0.9405
Epoch 20/35
38/38 [==============================] - 5s 121ms/step - loss: 0.0109 - accuracy: 0.9967 - val_loss:
0.2232 - val_accuracy: 0.9521
Epoch 21/35
38/38 [==============================] - 5s 122ms/step - loss: 0.0092 - accuracy: 0.9983 - val_loss:
0.2398 - val_accuracy: 0.9570
Epoch 22/35
38/38 [==============================] - 5s 121ms/step - loss: 0.0037 - accuracy: 0.9996 - val_loss:
0.2169 - val_accuracy: 0.9554
Epoch 23/35
38/38 [==============================] - 5s 120ms/step - loss: 0.0021 - accuracy: 0.9996 - val_loss:
0.2291 - val_accuracy: 0.9587
Epoch 24/35
38/38 [==============================] - 5s 121ms/step - loss: 0.0076 - accuracy: 0.9979 - val_loss:
0.2929 - val_accuracy: 0.9405
Epoch 25/35
38/38 [==============================] - 5s 120ms/step - loss: 0.0064 - accuracy: 0.9983 - val_loss:
0.2356 - val_accuracy: 0.9521
Epoch 26/35
38/38 [==============================] - 5s 120ms/step - loss: 0.0031 - accuracy: 0.9992 - val_loss:
0.2537 - val_accuracy: 0.9521
Epoch 27/35
38/38 [==============================] - 5s 122ms/step - loss: 0.0043 - accuracy: 0.9992 - val_loss:
0.2678 - val_accuracy: 0.9521
Epoch 28/35
38/38 [==============================] - 5s 121ms/step - loss: 0.0016 - accuracy: 1.0000 - val_loss:
0.2626 - val_accuracy: 0.9455
Epoch 29/35
38/38 [==============================] - 5s 122ms/step - loss: 0.0027 - accuracy: 0.9992 - val_loss:
0.2395 - val_accuracy: 0.9521
Epoch 30/35
38/38 [==============================] - 5s 121ms/step - loss: 0.0023 - accuracy: 0.9996 - val_loss:
0.2836 - val_accuracy: 0.9388
Epoch 31/35
38/38 [==============================] - 5s 120ms/step - loss: 0.0020 - accuracy: 0.9996 - val_loss:
0.3162 - val_accuracy: 0.9174
Epoch 32/35
38/38 [==============================] - 5s 120ms/step - loss: 0.0081 - accuracy: 0.9992 - val_loss:
0.3078 - val_accuracy: 0.9471
Epoch 33/35
38/38 [==============================] - 5s 121ms/step - loss: 0.0032 - accuracy: 0.9992 - val_loss:
0.4257 - val_accuracy: 0.9074
Epoch 34/35
38/38 [==============================] - 5s 121ms/step - loss: 0.0048 - accuracy: 0.9992 - val_loss:
0.3938 - val_accuracy: 0.9240
Epoch 35/35
38/38 [==============================] - 5s 121ms/step - loss: 0.0059 - accuracy: 0.9996 - val_loss:
0.2342 - val_accuracy: 0.9603
```

In [ ]:

```
plot_learning_curves(H,epochs=n_epochs)
```



In [ ]:

```
# Análisis del modelo
print("[INFO]: Evaluando el modelo...")
# Predicción (empleamos el mismo valor de batch_size que en training)
```

```
predictions = model_3.predict(x_te, batch_size=64) #(X)
# Informe para test
print(classification_report(y_test, predictions.argmax(axis=1), target_names=class_names)) #(X)
```

```
[INFO]: Evaluando el modelo...
              precision    recall  f1-score   support

         ALB       0.96      0.98      0.97       342
         BET       1.00      0.88      0.93        49
         DOL       1.00      1.00      1.00        15
         LAG       0.92      0.92      0.92        12
         NoF       0.94      0.96      0.95        98
       OTHER       1.00      0.95      0.97        59
       SHARK       1.00      1.00      1.00        36
         YFT       0.95      0.97      0.96       144

    accuracy                           0.96       755
   macro avg       0.97      0.96      0.96       755
weighted avg       0.96      0.96      0.96       755
```

5.1 Transfer Learning. Usando redes pre-entrenadas.

A continuación se ha decidido aplicar Transfer Learning para intentar conseguir mejores resultados que los obtenidos con las redes 'from scratch'.

Keras incluye varias redes neuronales preentrenadas como por ejemplo:

- VGG16
- ResNet 50
- MobileNetV2
- Xception
- Rasnet
- ....

En este trabajo se realizará transfer learning mediante el uso de VGG16, ResNet 50 V2 y MobileNetV2
https://keras.io/applications/

VGG16

Como primera red se usará la VGG16 con los pesos de Imagenet

In [ ]:

```
from tensorflow.keras.applications import VGG16

VGG16_base_model = VGG16(weights='imagenet',
                 include_top=False, # No incluimos el top model
                 input_shape=(224,224,3))

#VGG16_base_model.summary()
```

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/vgg16/vgg16_weight
s_tf_dim_ordering_tf_kernels_notop.h5
58892288/58889256 [==============================] - 1s 0us/step
58900480/58889256 [==============================] - 1s 0us/step
```

Ahora que tenemos la red VGG16 sin el top model, procedemos a conectar el base model con una serie de capas densas que harán las veces de top model.

In [ ]:

```
# Creando top model y conectándolo a la red VGG16

VGG16_base_model.trainable = False # Congelamos Base Model -> TRANSFER LEARNING
my_VGG16 = Sequential()
my_VGG16.add(VGG16_base_model) #Creamos nuestro Top Model

# Top Model
my_VGG16.add(Flatten())
my_VGG16.add(Dense(units = 32))
my_VGG16.add(BatchNormalization())
my_VGG16.add(Activation("relu"))
my_VGG16.add(Dropout(0.3))

# Clasificador softmax
my_VGG16.add(Dense(units = number_classes, activation = 'softmax'))

my_VGG16.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 vgg16 (Functional)          (None, 7, 7, 512)         14714688

 flatten_1 (Flatten)         (None, 25088)             0

 dense_2 (Dense)             (None, 32)                802848

 batch_normalization_3 (Batc  (None, 32)               128
 hNormalization)

 activation_3 (Activation)   (None, 32)                0

 dropout_3 (Dropout)         (None, 32)                0

 dense_3 (Dense)             (None, 8)                 264

=================================================================
Total params: 15,517,928
Trainable params: 803,176
Non-trainable params: 14,714,752
_________________________________________________________________
```

Antes de realizar el entrenamiento es necesario aplicarle a las imágenes el mismo preprocesamiento que le aplicaron los creadores de la red VGG16

In [ ]:

```
from tensorflow.keras.applications import vgg16

x_tr_vgg16 = vgg16.preprocess_input(x_tr*255.0)
x_val_vgg16 = vgg16.preprocess_input(x_val*255.0)
x_te_vgg16 = vgg16.preprocess_input(x_te*255.0)
```

In [ ]:

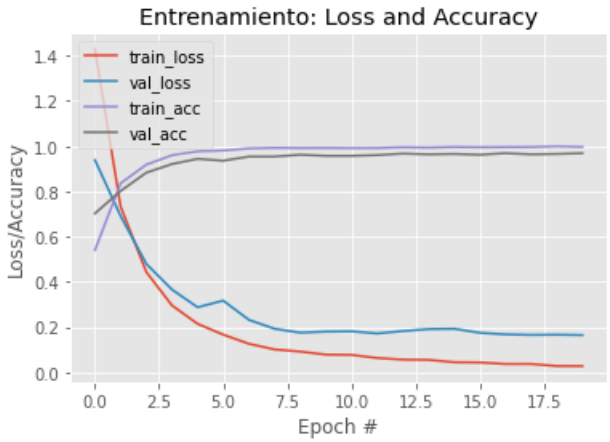
```
# Compilar el modelo
print("[INFO]: Compilando el modelo...")
my_VGG16.compile(loss='sparse_categorical_crossentropy', optimizer='adam', metrics=['accuracy'])

# Entrenamiento de la red
print("[INFO]: Entrenando la red...")
H = my_VGG16.fit(x_tr_vgg16, y_tr, batch_size=64, epochs=20, validation_data=(x_val_vgg16, y_val))

# Almacenamos el modelo empleando la función mdoel.save de Keras
my_VGG16.save("my_VGG16.h5")
```

```
[INFO]: Compilando el modelo...
[INFO]: Entrenando la red...
Epoch 1/20
38/38 [==============================] - 26s 555ms/step - loss: 1.4282 - accuracy: 0.5403 - val_loss:
0.9382 - val_accuracy: 0.7025
Epoch 2/20
38/38 [==============================] - 15s 389ms/step - loss: 0.7328 - accuracy: 0.8357 - val_loss:
0.6912 - val_accuracy: 0.8017
Epoch 3/20
38/38 [==============================] - 15s 401ms/step - loss: 0.4435 - accuracy: 0.9181 - val_loss:
0.4795 - val_accuracy: 0.8826
Epoch 4/20
38/38 [==============================] - 15s 400ms/step - loss: 0.2959 - accuracy: 0.9599 - val_loss:
0.3660 - val_accuracy: 0.9207
Epoch 5/20
38/38 [==============================] - 15s 395ms/step - loss: 0.2143 - accuracy: 0.9768 - val_loss:
0.2875 - val_accuracy: 0.9438
Epoch 6/20
38/38 [==============================] - 15s 391ms/step - loss: 0.1664 - accuracy: 0.9806 - val_loss:
0.3169 - val_accuracy: 0.9355
Epoch 7/20
38/38 [==============================] - 15s 394ms/step - loss: 0.1260 - accuracy: 0.9897 - val_loss:
0.2319 - val_accuracy: 0.9537
Epoch 8/20
38/38 [==============================] - 15s 397ms/step - loss: 0.1010 - accuracy: 0.9921 - val_loss:
0.1924 - val_accuracy: 0.9537
Epoch 9/20
38/38 [==============================] - 15s 397ms/step - loss: 0.0916 - accuracy: 0.9913 - val_loss:
0.1751 - val_accuracy: 0.9620
Epoch 10/20
38/38 [==============================] - 15s 397ms/step - loss: 0.0781 - accuracy: 0.9917 - val_loss:
0.1799 - val_accuracy: 0.9570
Epoch 11/20
38/38 [==============================] - 15s 395ms/step - loss: 0.0769 - accuracy: 0.9909 - val_loss:
0.1811 - val_accuracy: 0.9570
Epoch 12/20
38/38 [==============================] - 15s 394ms/step - loss: 0.0631 - accuracy: 0.9913 - val_loss:
0.1717 - val_accuracy: 0.9603
```

```
Epoch 13/20
38/38 [==============================] - 15s 394ms/step - loss: 0.0560 - accuracy: 0.9950 - val_loss:
0.1822 - val_accuracy: 0.9669
Epoch 14/20
38/38 [==============================] - 15s 394ms/step - loss: 0.0549 - accuracy: 0.9934 - val_loss:
0.1907 - val_accuracy: 0.9636
Epoch 15/20
38/38 [==============================] - 15s 395ms/step - loss: 0.0443 - accuracy: 0.9963 - val_loss:
0.1922 - val_accuracy: 0.9653
Epoch 16/20
38/38 [==============================] - 15s 395ms/step - loss: 0.0434 - accuracy: 0.9950 - val_loss:
0.1746 - val_accuracy: 0.9620
Epoch 17/20
38/38 [==============================] - 15s 395ms/step - loss: 0.0367 - accuracy: 0.9959 - val_loss:
0.1676 - val_accuracy: 0.9686
Epoch 18/20
38/38 [==============================] - 15s 396ms/step - loss: 0.0367 - accuracy: 0.9963 - val_loss:
0.1652 - val_accuracy: 0.9636
Epoch 19/20
38/38 [==============================] - 15s 396ms/step - loss: 0.0276 - accuracy: 0.9992 - val_loss:
0.1661 - val_accuracy: 0.9653
Epoch 20/20
38/38 [==============================] - 15s 396ms/step - loss: 0.0272 - accuracy: 0.9967 - val_loss:
0.1643 - val_accuracy: 0.9686
```

In [ ]:

```
plot_learning_curves(H,epochs=20)
```



In [ ]:

```
# Análisis del modelo
print("[INFO]: Evaluando el modelo...")
# Predicción (empleamos el mismo valor de batch_size que en training)
predictions = my_VGG16.predict(x_te_vgg16, batch_size=64)
# Informe para test
print(classification_report(y_test, predictions.argmax(axis=1), target_names=class_names))
```

```
[INFO]: Evaluando el modelo...
              precision    recall  f1-score   support

         ALB       0.98      0.97      0.98       342
         BET       0.98      0.96      0.97        49
         DOL       0.94      1.00      0.97        15
         LAG       1.00      0.92      0.96        12
         NoF       0.95      0.94      0.94        98
       OTHER       0.97      0.95      0.96        59
       SHARK       1.00      1.00      1.00        36
         YFT       0.96      0.99      0.97       144

    accuracy                           0.97       755
   macro avg       0.97      0.97      0.97       755
weighted avg       0.97      0.97      0.97       755
```

Hasta el momento esta red es con la que hemos conseguido los mejores resultados

ResNet 50 V2

Ahora vamos a emplear otra red preentrenada, en este caso ResNet 50 versión 2 y compararemos los resultados con la anterior. También se usarán los pesos de imagenet

In [ ]:

```
from tensorflow.keras.applications import ResNet50V2

ResNet50V2_base_model = ResNet50V2(weights='imagenet',
                 include_top=False, # No incluimos el top model
```

```
                  input_shape=(224,224,3))

#ResNet50V2_base_model.summary()
```

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/resnet/resnet50v2_
weights_tf_dim_ordering_tf_kernels_notop.h5
94674944/94668760 [==============================] - 3s 0us/step
94683136/94668760 [==============================] - 3s 0us/step
```

In [ ]:
```
# Creando Top Model y conectándolo a la red ResNet50V2

ResNet50V2_base_model.trainable = False # Congelando base model
my_ResNet50V2 = Sequential()
my_ResNet50V2.add(ResNet50V2_base_model) #Añadimos el base model congelado

# Top Model
my_ResNet50V2.add(Flatten())
my_ResNet50V2.add(Dense(units = 32))
my_ResNet50V2.add(BatchNormalization())
my_ResNet50V2.add(Activation("relu"))
my_ResNet50V2.add(Dropout(0.3))

# Clasificador softmax
my_ResNet50V2.add(Dense(units = number_classes, activation = 'softmax'))

my_ResNet50V2.summary()
```

```
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 resnet50v2 (Functional)     (None, 7, 7, 2048)        23564800

 flatten_2 (Flatten)         (None, 100352)            0

 dense_4 (Dense)             (None, 32)                3211296

 batch_normalization_4 (Batc  (None, 32)               128
 hNormalization)

 activation_4 (Activation)   (None, 32)                0

 dropout_4 (Dropout)         (None, 32)                0

 dense_5 (Dense)             (None, 8)                 264

=================================================================
Total params: 26,776,488
Trainable params: 3,211,624
Non-trainable params: 23,564,864
_________________________________________________________________
```

Antes de realizar el entrenamiento es necesario aplicarle a las imágenes el mismo preprocesamiento que le aplicaron los creadores de la red resnet

In [ ]:
```
from tensorflow.keras.applications import resnet_v2

x_tr_resnet = resnet_v2.preprocess_input(x_tr*255.0)
x_val_resnet = resnet_v2.preprocess_input(x_val*255.0)
x_te_resnet = resnet_v2.preprocess_input(x_te*255.0)
```

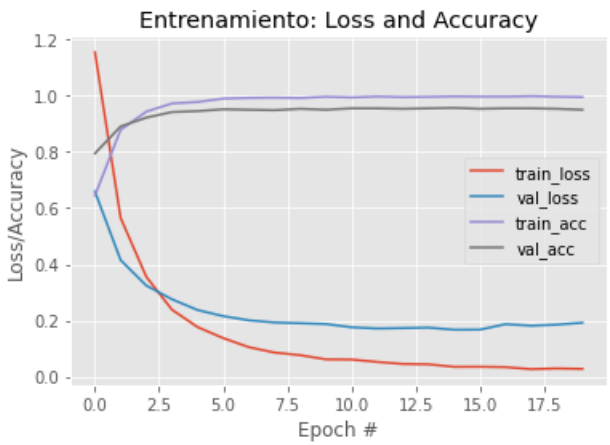
In [ ]:
```
# Compilar el modelo
print("[INFO]: Compilando el modelo...")
my_ResNet50V2.compile(loss='sparse_categorical_crossentropy', optimizer='adam', metrics=['accuracy']

# Entrenamiento de la red
print("[INFO]: Entrenando la red...")
H = my_ResNet50V2.fit(x_tr_resnet, y_tr, batch_size=64, epochs=20, validation_data=(x_val_resnet, y_

# Almacenamos el modelo empleando la función mdoel.save de Keras
my_ResNet50V2.save("my_ResNet50V2.h5") #(X)
```

```
[INFO]: Compilando el modelo...
[INFO]: Entrenando la red...
Epoch 1/20
38/38 [==============================] - 15s 290ms/step - loss: 1.1550 - accuracy: 0.6450 - val_loss:
0.6588 - val_accuracy: 0.7950
Epoch 2/20
38/38 [==============================] - 8s 208ms/step - loss: 0.5655 - accuracy: 0.8788 - val_loss:
0.4151 - val_accuracy: 0.8909
```

```
Epoch 3/20
38/38 [==============================] - 8s 207ms/step - loss: 0.3558 - accuracy: 0.9437 - val_loss:
0.3243 - val_accuracy: 0.9223
Epoch 4/20
38/38 [==============================] - 8s 209ms/step - loss: 0.2385 - accuracy: 0.9727 - val_loss:
0.2756 - val_accuracy: 0.9421
Epoch 5/20
38/38 [==============================] - 8s 212ms/step - loss: 0.1765 - accuracy: 0.9781 - val_loss:
0.2372 - val_accuracy: 0.9455
Epoch 6/20
38/38 [==============================] - 8s 214ms/step - loss: 0.1371 - accuracy: 0.9901 - val_loss:
0.2154 - val_accuracy: 0.9521
Epoch 7/20
38/38 [==============================] - 8s 213ms/step - loss: 0.1049 - accuracy: 0.9926 - val_loss:
0.2006 - val_accuracy: 0.9504
Epoch 8/20
38/38 [==============================] - 8s 211ms/step - loss: 0.0857 - accuracy: 0.9934 - val_loss:
0.1928 - val_accuracy: 0.9488
Epoch 9/20
38/38 [==============================] - 8s 210ms/step - loss: 0.0764 - accuracy: 0.9921 - val_loss:
0.1903 - val_accuracy: 0.9537
Epoch 10/20
38/38 [==============================] - 8s 209ms/step - loss: 0.0616 - accuracy: 0.9967 - val_loss:
0.1873 - val_accuracy: 0.9504
Epoch 11/20
38/38 [==============================] - 8s 208ms/step - loss: 0.0609 - accuracy: 0.9942 - val_loss:
0.1758 - val_accuracy: 0.9554
Epoch 12/20
38/38 [==============================] - 8s 209ms/step - loss: 0.0523 - accuracy: 0.9975 - val_loss:
0.1714 - val_accuracy: 0.9554
Epoch 13/20
38/38 [==============================] - 8s 209ms/step - loss: 0.0452 - accuracy: 0.9954 - val_loss:
0.1729 - val_accuracy: 0.9537
Epoch 14/20
38/38 [==============================] - 8s 210ms/step - loss: 0.0439 - accuracy: 0.9963 - val_loss:
0.1745 - val_accuracy: 0.9554
Epoch 15/20
38/38 [==============================] - 8s 211ms/step - loss: 0.0349 - accuracy: 0.9979 - val_loss:
0.1670 - val_accuracy: 0.9570
Epoch 16/20
38/38 [==============================] - 8s 211ms/step - loss: 0.0352 - accuracy: 0.9971 - val_loss:
0.1673 - val_accuracy: 0.9537
Epoch 17/20
38/38 [==============================] - 8s 211ms/step - loss: 0.0338 - accuracy: 0.9971 - val_loss:
0.1868 - val_accuracy: 0.9554
Epoch 18/20
38/38 [==============================] - 8s 210ms/step - loss: 0.0267 - accuracy: 0.9988 - val_loss:
0.1815 - val_accuracy: 0.9554
Epoch 19/20
38/38 [==============================] - 8s 210ms/step - loss: 0.0291 - accuracy: 0.9967 - val_loss:
0.1850 - val_accuracy: 0.9537
Epoch 20/20
38/38 [==============================] - 8s 209ms/step - loss: 0.0275 - accuracy: 0.9954 - val_loss:
0.1919 - val_accuracy: 0.9504
```

In [ ]:

```
plot_learning_curves(H,epochs=20)
```



In [ ]:

```
# Análisis del modelo
print("[INFO]: Evaluando el modelo...")
# Predicción (empleamos el mismo valor de batch_size que en training)
predictions = my_ResNet50V2.predict(x_te_resnet, batch_size=64)
# Informe para test
print(classification_report(y_test, predictions.argmax(axis=1), target_names=class_names))
```

```
[INFO]: Evaluando el modelo...
              precision    recall  f1-score   support

         ALB       0.95      0.97      0.96       342
         BET       0.98      0.94      0.96        49
         DOL       0.94      1.00      0.97        15
         LAG       1.00      0.92      0.96        12
         NoF       0.91      0.86      0.88        98
       OTHER       0.98      0.95      0.97        59
       SHARK       1.00      1.00      1.00        36
         YFT       0.95      0.97      0.96       144

    accuracy                           0.95       755
   macro avg       0.96      0.95      0.96       755
weighted avg       0.95      0.95      0.95       755
```

MobileNetV2

La última red preentrenada que se usará es MobileNet, empleando también los pesos de imagenet:

In [ ]:

```
from tensorflow.keras.applications import MobileNetV2

MobileNetV2_base_model = MobileNetV2(weights='imagenet',
                 include_top=False, # No incluimos el top model
                 input_shape=(224,224,3))

#MobileNetV2_base_model.summary()
```

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/mobilenet_v2/mobil
enet_v2_weights_tf_dim_ordering_tf_kernels_1.0_224_no_top.h5
9412608/9406464 [==============================] - 0s 0us/step
9420800/9406464 [==============================] - 0s 0us/step
```

In [ ]:

```
# Creando top model y conectándolo a la red MobileNetV2

MobileNetV2_base_model.trainable = False # Congelando base model
my_MobileNetV2 = Sequential()
my_MobileNetV2.add(MobileNetV2_base_model) #Añadimos el base model congelado

# Top Model
my_MobileNetV2.add(Flatten())
my_MobileNetV2.add(Dense(units = 32))
my_MobileNetV2.add(BatchNormalization())
my_MobileNetV2.add(Activation("relu"))
my_MobileNetV2.add(Dropout(0.3))

# Clasificador softmax
my_MobileNetV2.add(Dense(units = number_classes, activation = 'softmax'))

my_MobileNetV2.summary()
```

```
Model: "sequential_3"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 mobilenetv2_1.00_224 (Funct  (None, 7, 7, 1280)       2257984
 ional)

 flatten_3 (Flatten)         (None, 62720)             0

 dense_6 (Dense)             (None, 32)                2007072

 batch_normalization_5 (Batc  (None, 32)               128
 hNormalization)

 activation_5 (Activation)   (None, 32)                0

 dropout_5 (Dropout)         (None, 32)                0

 dense_7 (Dense)             (None, 8)                 264

=================================================================
Total params: 4,265,448
Trainable params: 2,007,400
Non-trainable params: 2,258,048
_________________________________________________________________
```

Antes de realizar el entrenamiento es necesario aplicarle a las imágenes el mismo preprocesamiento que le aplicaron los creadores de la red mobilenet

In [ ]:

```
from tensorflow.keras.applications import mobilenet_v2

x_tr_mobilenet = mobilenet_v2.preprocess_input(x_tr*255.0)
x_val_mobilenet = mobilenet_v2.preprocess_input(x_val*255.0)
x_te_mobilenet = mobilenet_v2.preprocess_input(x_te*255.0)
```

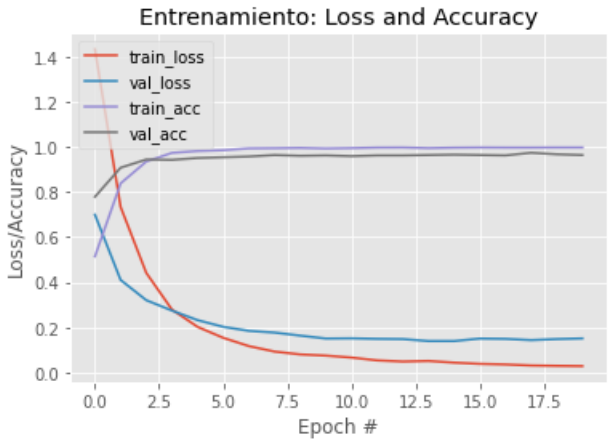
In [ ]:

```
# Compilar el modelo
print("[INFO]: Compilando el modelo...")
my_MobileNetV2.compile(loss='sparse_categorical_crossentropy', optimizer='adam', metrics=['accuracy'

# Entrenamiento de la red
print("[INFO]: Entrenando la red...")
H = my_MobileNetV2.fit(x_tr_mobilenet, y_tr, batch_size=64, epochs=20, validation_data=(x_val_mobile

# Almacenamos el modelo empleando la función mdoel.save de Keras
my_MobileNetV2.save("my_MobileNetV2.h5")
```

```
[INFO]: Compilando el modelo...
[INFO]: Entrenando la red...
Epoch 1/20
38/38 [==============================] - 8s 146ms/step - loss: 1.4323 - accuracy: 0.5143 - val_loss:
0.6989 - val_accuracy: 0.7785
Epoch 2/20
38/38 [==============================] - 4s 95ms/step - loss: 0.7336 - accuracy: 0.8378 - val_loss:
0.4099 - val_accuracy: 0.9074
Epoch 3/20
38/38 [==============================] - 4s 95ms/step - loss: 0.4413 - accuracy: 0.9346 - val_loss:
0.3201 - val_accuracy: 0.9438
Epoch 4/20
38/38 [==============================] - 4s 95ms/step - loss: 0.2787 - accuracy: 0.9731 - val_loss:
0.2742 - val_accuracy: 0.9421
Epoch 5/20
38/38 [==============================] - 4s 95ms/step - loss: 0.2018 - accuracy: 0.9814 - val_loss:
0.2317 - val_accuracy: 0.9504
Epoch 6/20
38/38 [==============================] - 4s 96ms/step - loss: 0.1534 - accuracy: 0.9851 - val_loss:
0.2021 - val_accuracy: 0.9537
Epoch 7/20
38/38 [==============================] - 4s 96ms/step - loss: 0.1166 - accuracy: 0.9930 - val_loss:
0.1842 - val_accuracy: 0.9570
Epoch 8/20
38/38 [==============================] - 4s 95ms/step - loss: 0.0924 - accuracy: 0.9938 - val_loss:
0.1766 - val_accuracy: 0.9636
Epoch 9/20
38/38 [==============================] - 4s 95ms/step - loss: 0.0798 - accuracy: 0.9950 - val_loss:
0.1631 - val_accuracy: 0.9603
Epoch 10/20
38/38 [==============================] - 4s 96ms/step - loss: 0.0745 - accuracy: 0.9926 - val_loss:
0.1502 - val_accuracy: 0.9620
Epoch 11/20
38/38 [==============================] - 4s 95ms/step - loss: 0.0659 - accuracy: 0.9942 - val_loss:
0.1511 - val_accuracy: 0.9587
Epoch 12/20
38/38 [==============================] - 4s 95ms/step - loss: 0.0533 - accuracy: 0.9967 - val_loss:
0.1488 - val_accuracy: 0.9620
Epoch 13/20
38/38 [==============================] - 4s 94ms/step - loss: 0.0483 - accuracy: 0.9971 - val_loss:
0.1478 - val_accuracy: 0.9620
Epoch 14/20
38/38 [==============================] - 4s 94ms/step - loss: 0.0508 - accuracy: 0.9942 - val_loss:
0.1391 - val_accuracy: 0.9636
Epoch 15/20
38/38 [==============================] - 4s 95ms/step - loss: 0.0433 - accuracy: 0.9963 - val_loss:
0.1390 - val_accuracy: 0.9653
Epoch 16/20
38/38 [==============================] - 4s 95ms/step - loss: 0.0380 - accuracy: 0.9971 - val_loss:
0.1501 - val_accuracy: 0.9636
Epoch 17/20
38/38 [==============================] - 4s 94ms/step - loss: 0.0353 - accuracy: 0.9967 - val_loss:
0.1488 - val_accuracy: 0.9620
Epoch 18/20
38/38 [==============================] - 4s 94ms/step - loss: 0.0307 - accuracy: 0.9967 - val_loss:
0.1432 - val_accuracy: 0.9736
Epoch 19/20
38/38 [==============================] - 4s 94ms/step - loss: 0.0292 - accuracy: 0.9971 - val_loss:
0.1478 - val_accuracy: 0.9669
Epoch 20/20
38/38 [==============================] - 4s 94ms/step - loss: 0.0279 - accuracy: 0.9971 - val_loss:
0.1508 - val_accuracy: 0.9636
```

In [ ]:

```
plot_learning_curves(H,epochs=20)
```



In [ ]:

```
# Análisis del modelo
print("[INFO]: Evaluando el modelo...")
# Predicción (empleamos el mismo valor de batch_size que en training)
predictions = my_MobileNetV2.predict(x_te_mobilenet, batch_size=64)
# Informe para test
print(classification_report(y_test, predictions.argmax(axis=1), target_names=class_names))
```

```
[INFO]: Evaluando el modelo...
              precision    recall  f1-score   support

         ALB       0.95      0.98      0.96       342
         BET       0.98      0.96      0.97        49
         DOL       0.88      1.00      0.94        15
         LAG       1.00      0.92      0.96        12
         NoF       0.95      0.81      0.87        98
       OTHER       0.97      0.95      0.96        59
       SHARK       1.00      1.00      1.00        36
         YFT       0.97      0.99      0.98       144

    accuracy                           0.96       755
   macro avg       0.96      0.95      0.95       755
weighted avg       0.96      0.96      0.96       755
```

---

6. Conclusiones

Como se ha ido reflejando a lo largo de los distintos experimentos realizados, el dataset sobre el que trabajamos en este proyecto es rápidamente 'ajustable'.

Es decir, dada la existencia de una clase que dispone de un número de muestras considerablemente mayor que el disponible en el resto de las clases, el análisis de la información por las distintas arquitecturas de red utilizadas, tanto las creadas 'from scratch' como las pre-entrenadas, consiguen adquirir un alto nivel de eficiencia y rendimiento.

Mencionar, como hecho destacable que, si cabe, al utilizar las redes pre-entrenadas el ajuste se produce de forma mucho más rápida, casi inmediata, lo que deja entrever el alto nivel de complejidad y eficiencia de las mismas.

Por último, comentar que al aplicar las técnicas de **DATA AUGMENTATION** y de **CLASS-WEIGHTING**, se consigue levemente obtener unos resultados más cercanos a lo que podría suponer el análisis de un conjunto de datos con una distribución más balanceada, llevandonos a unos niveles de pérdidas y de precisión algo más cercanos a la realidad.