UNIVERSIDAD
COMPLUTENSE
MADRID

Trabajo Fin de Máster - Curso 2022/2023

Modelado de Redes Neuronales Convolucionales para la clasificación de cuadros según su movimiento artístico

Autor: **Ignacio Correcher Sánchez**

FACULTAD DE
ESTUDIOS ESTADÍSTICOS
UNIVERSIDAD COMPLUTENSE DE MADRID

Máster en Big Data

Índice general

Capítulo 1

Introducción

La clasificación de imágenes y detección de objetos en las mismas, es una de las tareas más importantes en las que se pueden ver realmente las aplicaciones de la inteligencia artificial, estas pueden ir desde un "simple" clasificador de animales según su especie, a la detección prematura de tumores.
El objetivo de este trabajo será desarrollar y encontrar el mejor modelo de Deep Learning (DL) posible para la clasificación de cuadros según el movimiento artístico al que pertenezcan (Renacentismo, Impresionismo, Realismo...), como ya se comentará más en detalle en los próximos capítulos, esta clasificación requiere detectar detalles en algunos casos muy sutiles, es por ello que la profundidad de las capas de la red neuronal y el número de parámetros que se utilicen serán puntos clave a la hora de obtener un buen resultado.
Todo el mundo ha experimentado la sensación de ir a un museo con una audioguía o un experto en Historia del Arte, y queda claro lo mucho que puede llegar a enriquecer la visita. La motivación para este proyecto surge como un primer paso en esta dirección, tener a tu propio experto en Historia del Arte en el bolsillo. Concretamente, la implementación del modelo que se decrete como ganador a gran escala, y con él, desarrollar una aplicación para dispositivos móviles, donde el usuario pueda realizar una fotografía de un cuadro que encuentre en un museo o incluso de un cuadro propio, y que de esta forma la fotografía sea evaluada por el modelo final y aporte información sobre cuál sería el estilo pictórico donde encajaría de una mejor forma la obra.
Se cree que es interesante puesto que, a pesar de que si tratamos con cuadros muy conocidos con una búsqueda en internet podríamos obtener los mismos resultados, cuando ya se están estudiando cuadros menos famosos o incluso cuadros propios, o de estudiantes del Grado de Bellas Artes que estén practicando ciertos estilos, una búsqueda en Google puede no ser de tanta ayuda, como un modelo que haya aprendido cuáles son los rasgos característicos de cada movimiento y pueda así ayudar en la clasificación.
En el futuro, se pretende seguir desarrollando y completando este proyecto, para que el modelo aporte mucha más información como el autor de la obra o el periodo de publicación y, por supuesto, desarrollar dicha aplicación cuando se tenga este modelo mucho más completo, para que todo el mundo pueda descargarla y tenerla en sus dispositivos.

Una obra de arte nunca se termina, sólo se abandona
Leonardo Da Vinci

Capítulo 2

Análisis del dataset

Como se ha comentado previamente, el objetivo de este proyecto es realizar una red neuronal convolucional para la clasificación de cuadros en función de la corriente o estilo pictórico al que pertenecen. Para ello, es muy importante encontrar una base de datos con una gran cantidad de imágenes de cuadros, pero que también tengan información sobre los mismos ya que se necesitan saber los estilos de cada uno de los cuadros para poder crear un buen conjunto de pruebas sobre el que entrenar el modelo.
La base de datos que se ha utilizado es de la **National Gallery of Art** de Washington DC, los motivos por los cuales ha sido la elegida son muy claros:

- Tiene gran cantidad de cuadros y objetos artísticos listados.
- Una parte bastante considerable de estos objetos tienen la información necesaria para crear los conjuntos de entrenamiento necesarios.
- Tanto la base de datos como las imágenes son Open Access, bajo una licencia Creative Commons 0, por lo que se pueden utilizar de la forma que necesiten.
- La base de datos se encuentra alojada en GitHub, por lo que es de fácil acceso.

2.1. Estudio de la BBDD y obtención de los mismos

El primer obstáculo que se ha encontrado a la hora de obtener los datos es que, como se ha comentado, estos se encuentran en una base de datos construida y diseñada por el propio museo, por lo que un primer paso ha sido entender el modelo entidad-relación que se obtiene para identificar cada una de las tablas y seleccionar aquellas que sean relevantes para el proyecto.

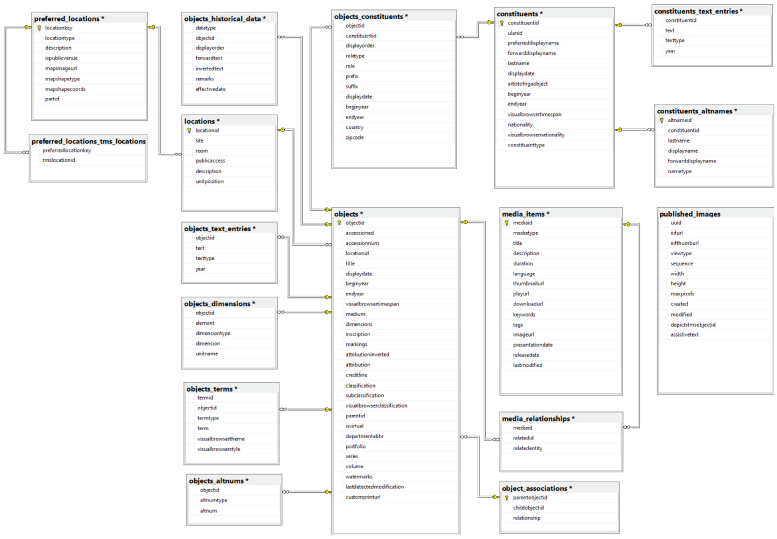

Figura 2.1: Modelo Entidad-Relación de la BBDD de la National Gallery of Art

Leyendo la documentación asociada que podemos encontrar en GitHub, se encuentra una descripción de los datos que contiene cada tabla y cuál puede ser su objetivo. Como se puede comprobar, el modelo parece tener una disposición en estrella, donde se tiene una tabla de hechos central llamada "objects" y luego tablas maestras alrededor que complementan la información de la tabla central. En este caso las tablas que interesan para la obtención de los datos son:

- **Objects**: Es indispensable, puesto que es la tabla central del modelo y contiene todos los objetos listados por el museo.
- **Ids**: Se trata de una tabla ajena al modelo, donde podemos encontrar la relación entre los diferentes IDs utilizados a lo largo de las tablas.
- **Images**: En esta tabla se pueden encontrar todos los objetos que tienen imágenes asociadas, estas imágenes se encuentran en forma de enlace a un servidor externo.
- **Terms**: Esta tabla nos aporta toda la información necesaria para construir el conjunto de entrenamiento, pues en ella se encuentra una clasificación de los objetos por movimiento artístico.

La obtención de los datos de las tablas que se necesitan del modelo se puede encontrar en el anexo (5.1).

2.2. Filtrado y limpieza de los datos

A pesar de haber reducido la totalidad de la base de datos a 4 tablas, todavía se tiene que realizar un proceso de filtrado de los datos, puesto que no todos serán de interés por diferentes motivos, como pueden ser falta de información u objetos que no encajan dentro de las necesidades, ya que únicamente interesan pinturas.

En primer lugar se va a filtrar toda la información de la tabla de objetos que no interese:

```
df_objects = df_objects.loc[df_objects['classification'].isin(['Print',
↪  'Paintings']), ['objectid','wikidataid', 'title', 'displaydate', 'medium',
↪  'classification']]
```

Lo que se realiza aquí es filtrar por la columna *classification* para solo retener los objetos que tengan el valor "Print" o "Paintings", puesto que el resto no interesa, con lo que se pasa de un total de 140240 objetos que habían en el dataset original a 70040 objetos. Además, solamente se tomarán las columnas que se indican puesto que el resto no son nada relevantes para el proyecto.

```
df_style = df_terms.loc[df_terms['termtype'] == 'Style', ['objectid', 'term']]
top10styles = df_style['term'].value_counts()[:10].index.tolist()
df_style = df_style.loc[df_terms['term'].isin(top10styles)]
df_style['term'].value_counts()
```

En este trozo de código, se filtra solamente la clasificación por *Style* de la tabla terms, puesto que hay diferentes terminologías asociadas a los objetos por las que se pueden clasificar. Acto seguido, se toma únicamente el top10 de estilos, porque pasado este percentil de la muestra, la mayoría de estilos tienen 1 cuadro asociado, por lo que no son relevantes y además crearán problemas en el proceso de entrenamiento de la red neuronal. Finalmente la distribución de estilos que queda es la siguiente:

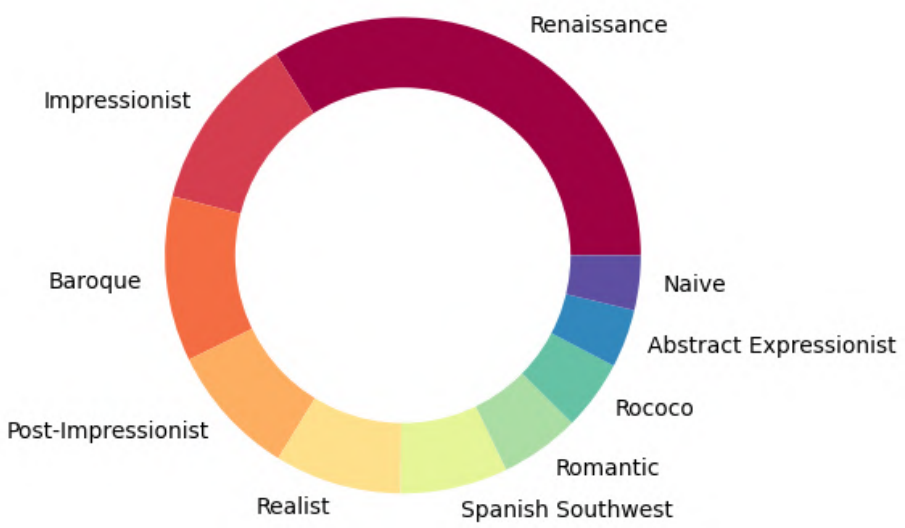

Figura 2.2: Styles distribution

Con esta distribución del top10 se obtienen finalmente un total de 8543 cuadros clasificados, ahora el objetivo es construir un DataFrame con pandas donde se tengan todos los IDs de los objetos, con su estilo asociado y el enlace a la imagen del propio objeto. Para ello se van a realizar varios joins entre las tablas que tenemos, mediante el campo *objectid* que hace de nexo de unión entre las mismas:

```
df_final = pd.merge(df_style, df_images, on = 'objectid', how = 'inner')
df_final = pd.merge(df_final, df_objects, on = 'objectid', how = 'inner')

df_final = df_final[['objectid', 'term', 'iiifthumburl']]
df_final.rename(columns = {'term':'style', 'iiifthumburl': 'img'}, inplace = True)
```

Con esto se obtiene un DataFrame final de 7977 cuadros con los filtros que se han comentado, es decir, solamente se incluyen los cuadros que tienen una clasificación y esta misma se encuentra en el top 10 de estilos más representados. Una muestra del Dataframe conseguido es la siguiente:

	objectid	style	img
0	115	Romantic	https://api.nga.gov/iiif/644bd0f2-3a24-4971-a2...
1	116	Romantic	https://api.nga.gov/iiif/d90b641f-b6fe-4fc4-a5...
2	895	Romantic	https://api.nga.gov/iiif/b5fa4110-a05c-40cc-b7...
3	996	Romantic	https://api.nga.gov/iiif/0a340e99-bbba-4c0f-94...
4	998	Romantic	https://api.nga.gov/iiif/09d2c030-bcf6-46b7-90...
...	...	...	...
7972	133026	Renaissance	https://api.nga.gov/iiif/0c737b99-86f5-4ebf-ab...
7973	133029	Renaissance	https://api.nga.gov/iiif/2b75437f-d4ff-4d36-a5...
7974	133635	Renaissance	https://api.nga.gov/iiif/c19b6e7d-2ccd-4226-82...
7975	133729	Renaissance	https://api.nga.gov/iiif/ce6c6c47-dde5-4251-9e...
7976	139319	Renaissance	https://api.nga.gov/iiif/5fbc07c2-830b-46ad-b3...

Figura 2.3: Muestra del DF final

2.3. Obtención de las imágenes

Una vez tenemos el DataFrame con las objetos, su clasificación y los enlaces a las imágenes puede parecer que ya se tiene todo lo necesario, pero queda un paso muy importante. Para entrenar el modelo se necesitan tener un total de 10 carpetas, una por cada estilo que se quiera clasificar, y dentro de cada una de ellas las imágenes que pertenezcan a dicha clase, también se añadirá como nombre de la imagen el objectID del objeto para poder tener una mejor trazabilidad de los datos y poder realizar validaciones en el futuro.
Cuando se trata con cantidades de datos tan grandes, nunca es planteable tener que tratar con ellos de forma manual, es decir, no se puede acceder a las casi 8000 imágenes una a una e ir descargándolas y guardándolas en las carpetas correspondientes, para ello se ha realizado un script en Python que realiza esta tarea. Aunque probablemente dicho script sea mejorable, ya que presenta un tiempo de ejecución muy alto, cumple su función principal que es ahorrar mucho tiempo.

Se quiere destacar que el script completo se encuentra en el anexo (5.2), ya que aquí se comentará por encima el funcionamiento del mismo.

1. En primer lugar, se define una carpeta donde se crearán todas las subcarpetas con cada una de las clases del modelo.

2. En segundo lugar, se recorre todo el dataframe final y para cada iteración se guardan los 3 valores que se tienen para cada elemento (id del objeto, enlace a la imagen y nombre del estilo). Luego, se crea una carpeta con el nombre del estilo en cuestión, en caso de no existir ya (esto ocurre cuando es la primera vez que se descarga una imagen de cada estilo).

3. Finalmente, se descarga la imagen accediendo al enlace, se le adjudica el nombre de archivo correspondiente al objectID y se guarda en la carpeta pertinente.

2.4. Validación final de los datos

En este punto, se tienen ya las casi 8000 imágenes descargadas y clasificadas en las diferentes carpetas. Antes de empezar a entrenar el modelo, es muy importante asegurarse de que los datos se encuentran bien etiquetados y no se tienen fallos, ya que un error en este aspecto podría perjudicar todo el desarrollo del modelo.
Por lo que, a pesar de haber tenido un buen seguimiento de los datos desde su origen, se decidió contactar con una experta en Historia del Arte, la licenciada Belén Blanco, para realizar validaciones sobre el dataset final. Para ello, se tomaron varias muestras aleatorias de cada una de las distintas clases obtenidas, y se validó tanto con la clasificación del museo, como con la opinión de la experta que dicha clasificación no tuviera errores.

2.5. Preparación de los datos

Una vez se tienen los datos validados, se deben formar los conjuntos de entrenamiento y de validación, para ello, como se puede ver en el anexo (5.3), se utilizará la función `train_test_split` de *sklearn*. Además de formar estos conjuntos de entrenamiento, también fijamos un primer batch_size que dictaminará la cantidad de imágenes que se utilizarán en cada iteración de entrenamiento, a menor batch_size más iteraciones de entrenamiento se tendrán que hacer y por tardo el proceso de entrenamiento será más largo, pero generalmente también más exitoso. Finalmente, se establecen unos arrays de imágenes y etiquetas que se utilizarán en unos scripts de validación que se verán más adelante.

Capítulo 3

Redes Neuronales Convolucionales

Las Redes Neuronales Convolucionales (CNN) nacieron del estudio del funcionamiento de la corteza visual del cerebro y se utilizan hoy en día para muchas tareas complejas, como la clasificación de imágenes, reconocimiento de voz o conducción autónoma.
Diversos estudios realizados por David H. Hubel y Torsten N. Wiesel entre 1958 y 1968 como [1], [2] y [3], por los que recibieron el premio Nobel de Medicina en 1981, demostraron que muchas neuronas de la corteza visual solamente reaccionan a ciertos estímulos del campo receptivo, es decir, toda la información del campo visual no pasa por todas las neuronas, sino que ocurre de forma localizada, más tarde toda esta información se junta y se forma un mosaico que resultará en la imagen final que estamos viendo.
Estos estudios inspiraron en 1980 el artículo por Fukushima sobre el neocognitrón [4], que fue el primer paso hacia las CNN y posteriormente su asentamiento con el modelo LeNet-5 en el artículo de *LeCun et al* [5] en 1998.
Una vez comentado el origen de las CNN, se considera importante entender cómo funcionan las capas convolucionales del modelo. Estas capas, son realmente parecidas a las capas densas de una red neuronal estándar, pero como diferencia destacable tienen que las neuronas de la primera capa no reciben la información de todos los píxeles de la imagen, sino solo los píxeles que están en sus "campos receptivos", y de la misma forma funcionan el resto de capas convolucionales, cada neurona no recibe información de todas las de la capa anterior sino de una selección. Esta estructura, que se entiende mejor con la ayuda de la imagen 3.1, permite centrarnos en detalles que pueden marcar la diferencia a la hora de realizar tareas complejas, pero sin perder el contexto general de la imagen de entrada.

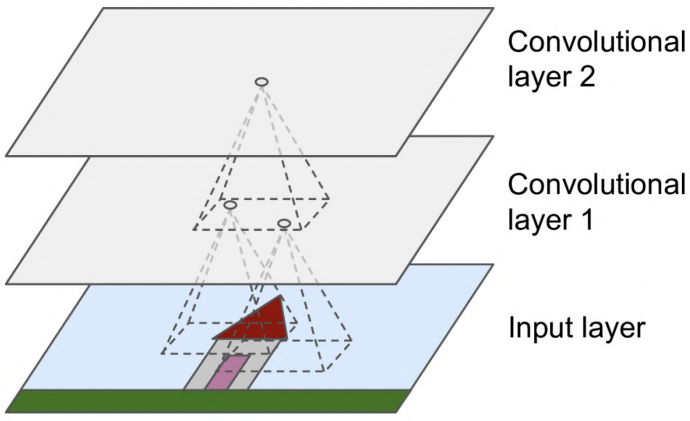

Figura 3.1: Estructura de capas convolucionales

3.1. Modelos CNN

Se van a presentar varios modelos, de menor a mayor complejidad y utilizando diferentes técnicas, aprendidas tanto en la asignatura de Deep Learning del Máster como en el libro [6] y en diversos artículos que serán citados en su debido momento.

3.1.1. Modelo 1: Modelo estándar sin regularización

El código del modelo completo se puede ver en el anexo (5.4), se trata de un modelo estándar sin aplicar ninguna técnica de regularización, para tener claro cuál es el "punto de partida", en lo que a *accuracy* se refiere.

En primer lugar se realiza un *rescaling* de las imágenes para estandarizar sus valores al intervalo $[1, 255]$. En cuanto a capas convolucionales se tienen 2, una con 16 filtros y otra con 32, ambas usando la función de activación *Relu*.

El cambio en una capa convolucional de un campo receptivo al siguiente se mide con el parámetro *strides*, en este caso por ejemplo, como el kernel de los filtros es de 3x3 y el paso es de 1, quiere decir que cada uno de los filtros de 3x3 "píxeles" se reducirá a 1 píxel en la siguiente capa.

Otro parámetro a tener en cuenta es el *padding*, este parámetro se encarga de modificar o no el tamaño de las imágenes, por la forma de actuación de las CNN, si no se interviene cada vez el tamaño de las imágenes se iría reduciendo, usando el parámetro padding como el valor "same"lo que ocurre es que se sustituyen los píxeles que se perderían con 0, para que el tamaño no se vaya reduciendo en cada capa.

Entre las capas convolucionales, se han introducido las capas de *pooling*, cuyo objetivo es reducir la imagen, para disminuir la carga computacional. Se pueden diferenciar 2 grandes técnicas, *Average Pooling* y *Max Pooling*, las diferencias entre ambas técnicas es la forma de manejar los píxeles, en ambas se elige un tamaño de píxeles y se usa la función de agregación indicada, por ejemplo con un paso de 1 y *average pooling* el píxel resultante será la media de los pixeles indicados en el tamaño, y en el *max pooling* será el valor del pixel máximo. Se quiere destacar, que en la literatura se ha demostrado que el *Max Pooling* ofrece mejores resultados en tareas como el reconocimiento de objetos mientras que el *average pooling* muestra mejores resultados en problemas que requieren de una mayor atención al detalle, como la segmentación de imágenes.

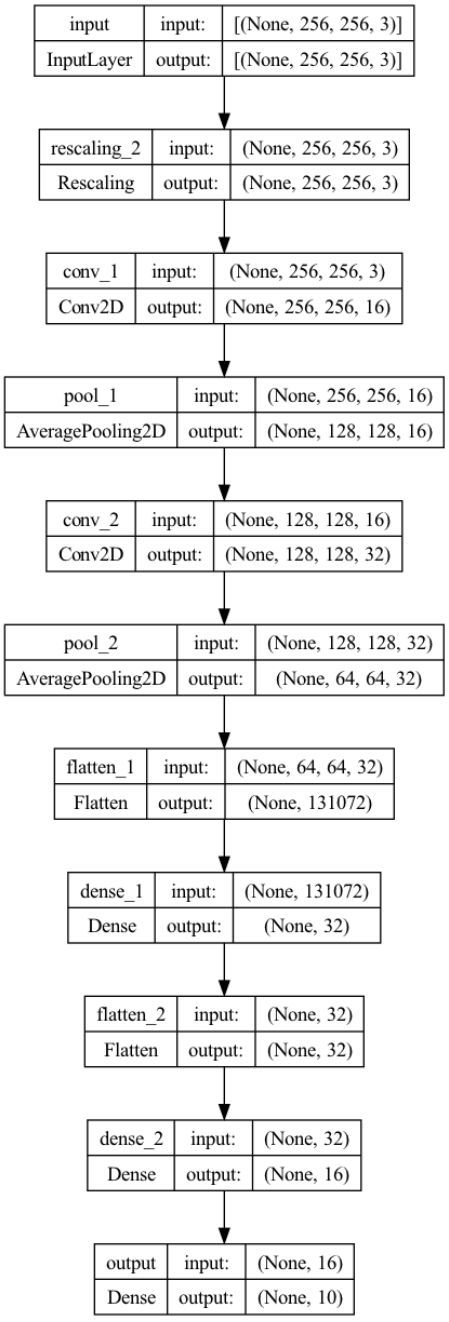

Figura 3.2: Estructura del Modelo 1

Después de las capas convolucionales se encuentran las capas densas, también con funciones de activación Relu y finalmente una capa *output* donde la función que se utiliza es una *softmax* con 10 clases, ya que la clasificación es no binaria, con 10 estilos diferentes.

Finalmente, para entrenar el modelo se han fijado 8 *epochs* y como función de pérdida se ha utilizado la mejor en este tipo de clasificaciones, conocida como *Categorical Cross Entropy*.
Con todos estos parámetros que hemos indicado, se obtiene un porcentaje de acierto en los cuadros del conjunto de validación alrededor del 50 %, como se puede ver en (**??**), lo cual no es demasiado alto pero teniendo en cuenta de que se trata de una clasificación de 10 clases distintas todo lo que sea por encima del 10 % será mejor que la aleatoriedad.

```
results = model_1.evaluate(val_ds, verbose=0)
print('Test Loss: ' + str(round(results[0],3))+ '\nTest Accuracy: ' +
↪  str(round(results[1]*100,3))+'%')

Test Loss: 1.433
Test Accuracy: 49.311%
```

Figura 3.3: Resultados del Modelo 1

En cuanto al Test Loss que se obtiene, si se hiciera una clasificación de manera aleatoria, por la forma de la función de pérdida que se usa (Categorical Cross Entropy), se puede calcular el test loss de la siguiente forma:

$$\text{Loss} = -\sum_{i=1}^{C} y_i \log(p_i)$$

Donde C es el número de clases, en este caso 10, y_i es 1 si la clase es verdadera y 0 en otro caso, y p_i es la probabilidad predicha para esa clase, de forma aleatoria es 0,10, por lo que haciendo la suma se tiene que de forma aleatoria se obtendría como Test Loss $-log(0{,}10) = 2{,}302$. Así, el valor que se ha obtenido con nuestro modelo es inferior a ese, por lo que parece que vamos por el buen camino.
Como último apunte para este modelo, se han preparado unos gráficos para observar el resultado de la clasificación y validar las predicciones. El código para la obtención de estas validaciones se puede encontrar en el anexo (5.5).

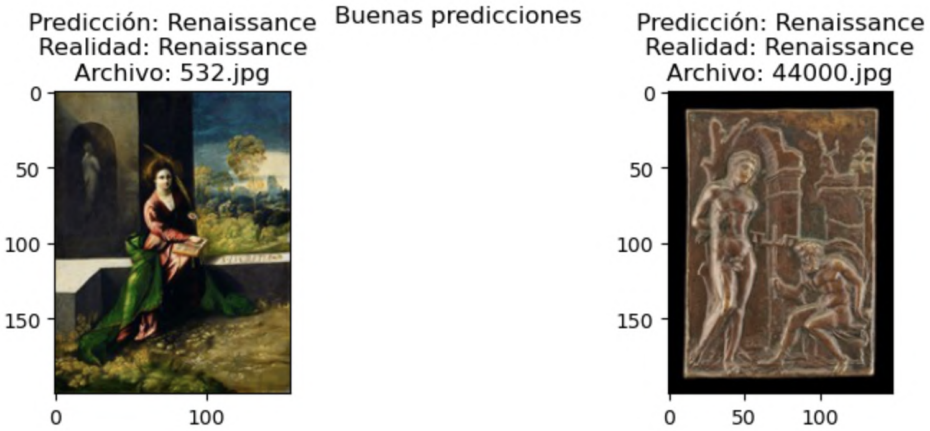

Figura 3.4: Buenas predicciones del Modelo 1

En esta imagen se observan 2 cuadros, concretamente los asociados a los objectID 532 y 44000 que han obtenido una buena clasificación, puesto que ambos pertenecían al Renacimiento y así han sido clasificados.

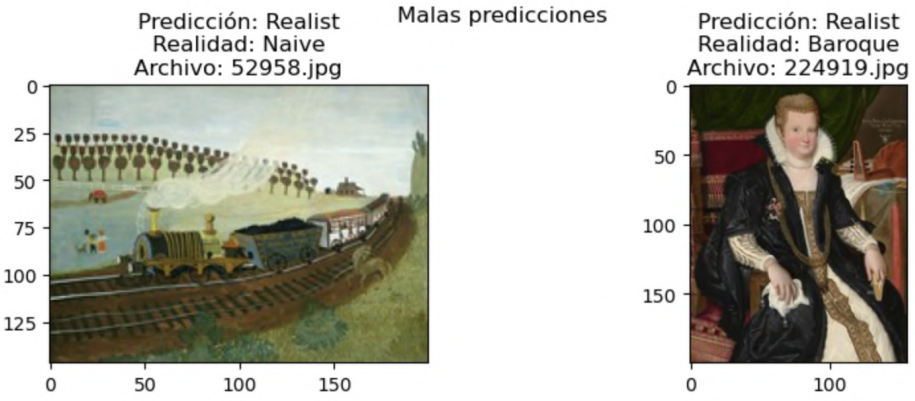

Figura 3.5: Malas predicciones del Modelo 1

En esta segunda imagen, se pueden observar los cuadros que hacen referencia a los objectID 52958 y 224919, donde la predicción no ha sido acertada, puesto que el modelo los ha clasificado como realistas pero pertenecen al Naive y al Barroco.

3.1.2. Modelo 2: Con regularización y más filtros

En este segundo modelo se van a utilizar diferentes técnicas de regularización, cuyo objetivo es reducir el sobreajuste u overfitting, es decir, que el modelo se ajuste tanto a los datos de entrenamiento que realmente no esté captando los atributos generales que se quieren, y por consecuencia la precisión en con los datos de entrenamiento siga aumentando mientras que cada vez se va reduciendo más con los datos de validación.
Para ello se utilizan 2 técnicas:

- **Dropouts**: Cuando se encuentra una capa de este tipi, un porcentaje de los datos que entran en ella, que viene marcado por el parámetro de entrada, son ignorados, reduciendo así las posibilidades de sobreajuste.
- **Early Stopping**: Es una técnica más avanzada, a diferencia del *dropout*, no es una capa del modelo como tal, sino un "parámetro" que añadimos en su entrenamiento. En este parámetro, indicamos que métrica se quiere monitorizar y cuál es la paciencia que se le atribuye. De esta forma, monitorizando la métrica *validation loss*, cuando se detecte que no ha mejorado en un número determinado de iteraciones (dado por la paciencia), parará totalmente el entrenamiento, para que no siga sobreajustándose el modelo.

En este punto, se quiere destacar que con la realización del *callback* de *early stopping*, se tuvo la idea de tomar este parámetro de paciencia como una función de las épocas en lugar de un valor fijo, ya que realmente no es lo mismo entrenar con 10 épocas y que cuando no haya mejorado la métrica en 5 ya se pare el entrenamiento, que entrenar 100 épocas y parar también cuando no se mejore en 5. Este pensamiento llevó a realizar una mayor búsqueda por internet , y contrastando diferentes puntos de vista, se llegó a que el valor que se iba a tomar de paciencia era el entero más próximo a un 10 % del número de épocas que se usen en el entrenamiento.

Al utilizar la regularización con *dropouts*, y teóricamente reducir la información que pasa por cada capa, el modelo se entrenará de una forma más rápida, por lo que se ha añadido también una capa convolucional más y se han usado más filtros y con kernels más grandes, en comparación al modelo previo.

Una idea que se ha obtenido de [6], es aprovechar que con las capas de *pooling*, las cuales se encuentran situadas entre las convolucionales, se reducen los píxeles de entrada y de esta manera poder doblar el número de filtros de una capa a la siguiente. En este caso se parte de 64 filtros en la primera capa, se dobla a 128 en la segunda y se termina con 256 filtros. Esta idea de aprovechar las técnicas de regularización para poder ser más codiciosos en la densidad de nuestras capas es un concepto que se utiliza muchísimo en modelos más avanzados.
Con todos estos cambios en el modelo, se ha vuelto a realizar un entrenamiento con un máximo de 50 *epochs*, a pesar de que con el *callback* de *early stopping* han sido un total de 22 *epochs*, y los resultados han sido los siguientes:

```
results = model_2.evaluate(val_ds, verbose=0)
print('Test Loss: ' + str(round(results[0],3))+ '\nTest Accuracy: ' +
↪  str(round(results[1]*100,3))+'%')

Test Loss: 1.368
Test Accuracy: 52.569%
```

Como se puede observar, se han mejorado en ambas métricas, el *test loss* se ha reducido y el test *accuracy* ha incrementado. Estas mejoras nos indican que los cambios que se han hecho van en el buen camino, por lo que se mantendrán en las siguientes iteraciones del modelo.

3.1.3. Modelo 3: Data Augmentation

A pesar de que el modelo 2 ha presentado mejoras sobre el anterior, parece que se ha llegado a un punto donde las mejoras sobre la complejidad del modelo no aportan tanto valor al resultado final. Uno de los motivos por los cuales el modelo no parece mejorar es que la muestra de datos con la que se realiza el entrenamiento no sea lo suficientemente grande, para ello en este nuevo modelo se va a optar por ampliarla de forma artificial, usando una técnica que se conoce como *Data Augmentation*.

La idea detrás del *Data Augmentation* es que, a un humano no le importa, por ejemplo, ver la imagen de un perro girada, con más zoom o menos zoom o desplazada, ya que va a poder reconocer de la misma forma al animal. De la misma forma queremos que ocurra con nuestro modelo de clasificación, vamos a aplicar al *dataset* de imágenes ciertas transformaciones de este tipo y, de esta forma, se aumentará artificialmente la cantidad de imágenes de las que disponemos para entrenar, a la vez que se reducirá también el sobreajuste.
El código completo para la creación de este nuevo modelo se puede encontrar en (5.7), ya que aquí se va a comentar únicamente la nueva mejora que se ha introducido, las transformaciones de Data Augmentation.

```
data_augmentation= keras.Sequential(

    [
        layers.experimental.preprocessing.RandomFlip(),
        layers.experimental.preprocessing.RandomRotation(0.25),
        layers.experimental.preprocessing.RandomZoom(0.25)
    ], name= 'data_aug'

)
```

Aquí se puede observar el detalle de la nueva capa, que hemos llamado `data_aug`. En ella, las imágenes que entren como input recibirán un *random flip* en alguno de los ejes, una rotación aleatoria de 0.25 rad, y un zoom del 25 %.

Con estas mejoras, usando el nuevo modelo con Data Augmentation, los resultados que se han obtenido son los siguientes:

```
results = model_3.evaluate(val_ds, verbose=0)
print('Test Loss: ' + str(round(results[0],3))+ '\nTest Accuracy: ' +
↪  str(round(results[1]*100,3))+'%')

Test Loss: 1.278
Test Accuracy: 53.624%
```

3.1.4. Modelo 4: *Inception*

Existen diferentes competiciones de modelos de predicción, gracias a esto y a las diferentes publicaciones se va formando una historia, donde unos modelos mejoran a otros y los suceden en el trono. En 2014, Christian Szegedy *et al.* desarrollaron la arquitectura *GoogLeNet* [7], que mejoraba en muchísimo a los modelos anteriores, usando además menos parámetros. Esta mejora se debió al uso de unas subredes llamadas *inception*.
En este nuevo modelo, se ha usado esta idea de utilizar subredes con la arquitectura de las inception y se han adaptado a los datos que se están manejando. Se va a estudiar cómo funcionan estas subredes *inception*, ya que el resto del modelo, que se puede consultar en (5.8) es como los que se han presentado anteriormente pero con más capas y más filtros.

```
def inception(x,
              filters_1x1,
              filters_3x3_reduce,
              filters_3x3,
              filters_5x5_reduce,
              filters_5x5,
              filters_pool):
  path1 = layers.Conv2D(filters_1x1, (1, 1), padding='same',activation='relu')(x)

  path2 = layers.Conv2D(filters_3x3_reduce, (1, 1), padding='same',
  ↪  activation='relu')(x)
  path2 = layers.Conv2D(filters_3x3, (3, 3), padding='same',
  ↪  activation='relu')(path2)

  path3 = layers.Conv2D(filters_5x5_reduce, (1, 1), padding='same',
  ↪  activation='relu')(x)
  path3 = layers.Conv2D(filters_5x5, (5, 5), padding='same',
  ↪  activation='relu')(path3)

  path4 = layers.MaxPool2D((3, 3), strides=(1, 1), padding='same')(x)
  path4 = layers.Conv2D(filters_pool, (1, 1), padding='same',
  ↪  activation='relu')(path4)

  return tf.concat([path1, path2, path3, path4], axis=3)
```

La arquitectura inception está formada por 4 caminos en paralelo, donde se encuentran filtros de 1x1, 3x3, 5x5 y capa de *pooling*. En las capas de 3x3 y 5x5, de una forma muy inteligente, se encuentran primero un conjunto de capas con el sufijo "reduce", formada por filtros de 1x1, cuyo objetivo es reducir la dimensionalidad de los datos.
Los inputs de la función *inception* serán el número de filtros de cada tamaño que se necesiten, y el output es un tensor con las capas.

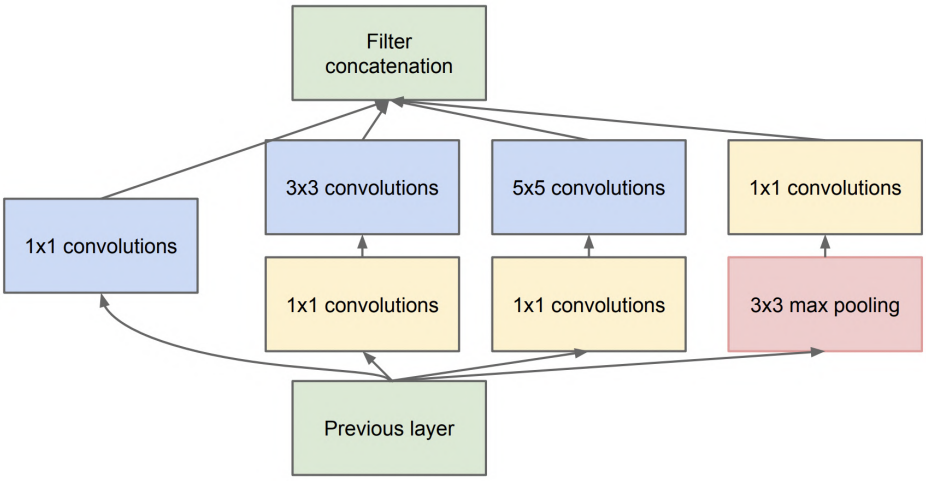

Figura 3.6: Esquema de la arquitectura Inception. Fuente: [7]

Como se puede observar en el siguiente output, los resultados de este modelo son muy superiores a los anteriores.

```
results = model_4_inception.evaluate(val_ds, verbose=0)
print('Test Loss: ' + str(round(results[0],3))+ '\nTest Accuracy: ' +
↪  str(round(results[1]*100,3))+'%')

Test Loss: 1.196
Test Accuracy: 58.466%
```

En parte, se está utilizando un modelo mucho más denso y profundo, pero no hay que quitar importancia a la idea de la arquitectura *inception*, la cual ha servido para mejorar al anterior modelo en un 5 %.

3.1.5. Modelo 5: EfficientNet

En este punto, se tiene un porcentaje de acierto bastante considerable para tratarse de 10 clases distintas. Pero, a pesar de ello, se va a realizar un modelo final utilizando una de las tecnologías por las cuales las redes convolucionales son tan interesantes y existe tanto empeño en ir mejorándolas, estamos hablando del *Transfer Learning*.

El *transfer learning* sirve para poder utilizar modelos ya preentrenados, los cuales tienen muchos parámetros y que corresponden a redes neuronales muy densas y profundas. Estos modelos referentes se desarrollan por grandes empresas o instituciones y se suelen compartir de manera libre mediante artículos académicos. En este caso, se ha escogido un modelo de la familia EfficientNet [8], concretamente el B0 que es el modelo más básico de todos.

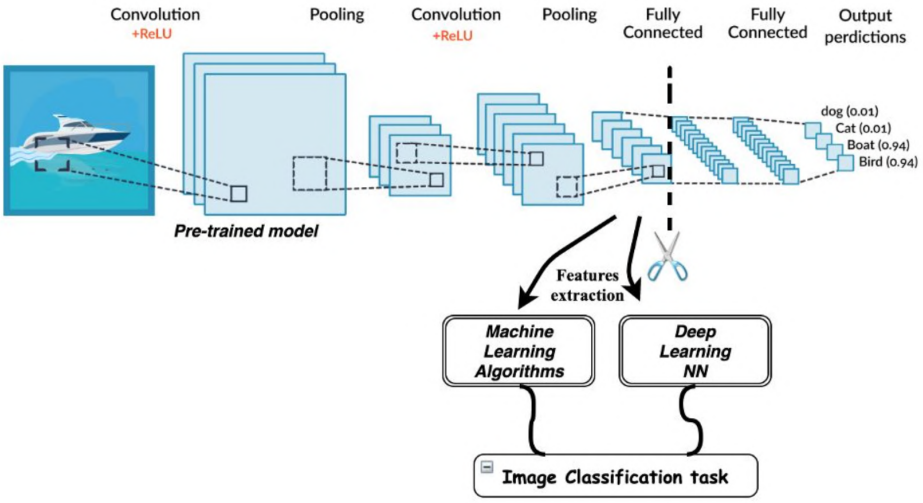

Figura 3.7: Transfer Learning

La idea del transfer learning es utilizar un modelo preentrenado con un dataset distinto, pero este modelo tiene muchisimos parámetros y el dataset con el que se ha entrenado suele ser muy grande y de muy buena calidad. Cuando ya se tiene este modelo preentrenado, se cogen las capas convolucionales del mismo y se exportan al nuevo modelo que se quiera construir. Ahora se personaliza el modelo, añadiendo a continuación de las capas que se han importado un conjunto de capas nuevas que se adapten al dataset en cuestión.
Es importante que se congele el modelo preentrenado que se ha importado, si no se tienen los recursos o el tiempo necesario para reentrenarlo. De esta forma, en el proceso de entrenamiento solo se irán ajustado las capas extra que se hayan añadido, puesto que los pesos de las neuronas del modelo preentrenado estarán congeladas, y por tanto el entrenamiento será mucho más rápido.

```
import efficientnet.keras as efn
B0efn = efn.EfficientNetB0(input_shape = (img_size, img_size, 3), include_top =
↪  False, weights = 'imagenet')

for layer in B0efn.layers:
    layer.trainable = False
```

En este *snippet* de código se puede ver como se importa el modelo B0 de la familia EfficientNet y como se realiza un bucle `for` para congelar todas las capas del modelo que se ha llamado `B0efn`.

Ahora que ya hemos fijado las capas preentrenadas, se añaden las capas restantes que harán que el modelo importado se adapte al problema que se está intentando resolver.

```
inputs = tf.keras.Input(shape=(img_size,img_size,3), name='input')
data_aug= data_augmentation(inputs)
reescaling = layers.Rescaling(1. / 255)(data_aug)
x = B0efn.output
x = layers.Flatten()(x)
x = layers.Dense(1024, activation="relu")(x)
x = layers.Dropout(0.5)(x)

outputs = layers.Dense(num_classes, activation='softmax', name='output')(x)
model_5_EFN = keras.Model(inputs=B0efn.input, outputs=outputs, name='Model_5_EFN')
```

Finalmente, se puede comprobar que los resultados usando un modelo preentrenado mejoran en gran medida a los que se habían obtenido antes, esto es porque realmente se está usando un modelo muchísimo más profundo que los que se habían desarrollado en secciones previas.

```
results = model_5_EFN.evaluate(val_ds, verbose=0)
print('Test Loss: ' + str(round(results[0],3))+ '\nTest Accuracy: ' +
↪  str(round(results[1]*100,3))+'%')

Test Loss: 1.081
Test Accuracy: 66.228%
```

Capítulo 4

Conclusiones

Se han desarrollado 5 modelos distintos, cada uno de mayor complejidad y con técnicas más avanzadas que el anterior. Por esto, y como es evidente, el mejor modelo de todos lo que se han obtenido es el Modelo 5, usando la técnica de *Transfer Learning* con la rama de modelos *EfficientNet*. Es un modelo que usa muchísimos más parámetros que los modelos anteriores, muchas más capas y más filtros en las partes convolucionales, lo que aporta una ventaja muy clara sobre el resto de modelos. Esto es, porque como se ha visto en su propia sección, el modelo preentrenado que se importa tiene las capas congeladas, por lo tanto no necesitan reentrenarse y se puede manejar un modelo tan denso y profundo como es el B0 de *EfficientNet*.
Está claro que, si se tuvieran los recursos y el tiempo necesarios para procesar modelos tan densos como el Modelo 5, otros modelos como el que utiliza la tecnología *Inception* de *GoogleNet*, tendrían muchos mejores resultados, aunque en el artículo de lanzamiento del modelo *EfficientNet* [8] se hace una comparación con *Inception v4* y aún así resulta ganador, también hay que tener en cuenta que modelos más recientes tienen ventaja y suelen obtener mejores resultados que los del pasado.
Finalmente, se quiere destacar, que a pesar de que el modelo ganador (Modelo 5) haya obtenido una precisión del 66 % siendo superior al resto por un buen margen, y que un modelo aleatorio tendría una precisión del 10 % pues se está trabajando con 10 clases distintas, no son los resultados que se busca obtener. El motivo más claro para no ver esta mejora tan grande en los resultados de los modelos es, seguramente, tener un banco de datos tan reducido, pues para entrenar modelos de Deep Learning, con un aprendizaje no supervisado, tan densos como los que se han visto se necesitan muchísimos más datos de los se han podido obtener.
Es por esto, que con los mismos desarrollos o muy parecidos que se han llevado a cabo, si se consigue ampliar esta base de datos con la que se han entrenado los modelos, podrían mejorar mucho los resultados.

Bibliografía

[1] David H. Hubel (1958). *Single unit activity in striate cortex of unrestrained cats.*

[2] David H. Hubel & Torsten N. Wiesel (1959). *Receptive fields of single neurones in the cat's striate cortex.*

[3] David H. Hubel & Torsten N. Wiesel (1968). *Receptive fields and functional architecture of monkey striate cortex.*

[4] Kunihiko Fukushima (1980). *Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position.*

[5] Y. Lecun, L. Bottou, Y. Bengio & P. Haffner (1998). *Gradient-based learning applied to document recognition.*

[6] Aurélien Géron (2019). *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow, 2nd Edition.*

[7] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke & Andrew Rabinovich (2014). *Going Deeper with Convolutions.*

[8] Mingxing Tan & Quoc V. Le (2020). *EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks.*

Capítulo 5

Anexos

5.1. Obtención de los datos

```python
import pandas as pd
import numpy as np

df_objects = pd.read_csv(
    'https://raw.githubusercontent.com/NationalGalleryOfArt/opendata/main/data/
    objects.csv',
    sep = ','
)

df_ids = pd.read_csv(
    'https://raw.githubusercontent.com/NationalGalleryOfArt/opendata/main/data/
    alternative_identifiers.csv',
    sep = ','
)
df_ids = df_ids.loc[df_ids['idschemelabel'] == 'Wikidata ID']
df_ids.rename(columns = {'identifier':'wikidataid'}, inplace = True)


df_images = pd.read_csv(
    'https://raw.githubusercontent.com/NationalGalleryOfArt/opendata/main/data/
    published_images.csv',
    sep = ','
)
df_images.rename(columns = {'depictstmsobjectid':'objectid'}, inplace = True)


df_terms = pd.read_csv(
    'https://raw.githubusercontent.com/NationalGalleryOfArt/opendata/main/data/
    objects_terms.csv',
    sep = ','
)
```

5.2. Descarga de las imágenes

```python
import os
import requests

# Directorio para almacenar todas las carpetas e imágenes
carpeta_base_local = "/Users/ignacio/Documents/TFM/Paintings"

# Recorremos las filas del DataFrame
for index, row in df_final.iterrows():
    id = row["objectid"]
    enlace_imagen = row["img"]
    estilo = row["style"]

    # Creamos la carpeta específica para el estilo si no existe
    carpeta_destino = os.path.join(carpeta_base_local, estilo)
    os.makedirs(carpeta_destino, exist_ok=True)

    # Descargamos la imagen desde el enlace
    response = requests.get(enlace_imagen, stream=True)

    if response.status_code == 200:
        # Extraemos el nombre del archivo
        nombre_archivo = f"{id}.jpg"
        ruta_archivo_local = os.path.join(carpeta_destino, nombre_archivo)

        # Guardamos la imagen en el directorio
        with open(ruta_archivo_local, 'wb') as file:
            for chunk in response.iter_content(chunk_size=1024):
                file.write(chunk)

        print(f"Imagen {nombre_archivo} descargada y guardada en
        ↪ {ruta_archivo_local}")
    else:
        print(f"Error al descargar la imagen desde {enlace_imagen}")
```

5.3. Training y Validation set

```python
import pandas as pd
import os
import numpy as np
import random
import tensorflow as tf
import matplotlib.pyplot as plt
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from sklearn.model_selection import train_test_split
from tensorflow import keras
from tensorflow.keras import layers
from tensorflow.keras.utils import plot_model

img_size = 256
batch_size = 32

data_dir = '/Users/ignacio/Documents/TFM/Paintings'

datagen = ImageDataGenerator(
    rescale=1.0/255
)

all_images = []
all_labels = []
for class_name in sorted(os.listdir(data_dir)):
    class_path = os.path.join(data_dir, class_name)
    if os.path.isdir(class_path):
        class_images = [os.path.join(class_name, image) for image in
        ↪  os.listdir(class_path)]
        all_images.extend(class_images)
        all_labels.extend([class_name] * len(class_images))

# Train/Test
train_images, val_images, train_labels, val_labels = train_test_split(
    all_images, all_labels, test_size=0.2, stratify=all_labels, random_state=42
)

train_ds = datagen.flow_from_dataframe(
    pd.DataFrame({"image": train_images, "class": train_labels}),
    directory=data_dir,
    x_col="image",
    y_col="class",
    target_size=(img_size, img_size),
    batch_size=batch_size,
    class_mode="categorical",
    shuffle=False
)

val_ds = datagen.flow_from_dataframe(
    pd.DataFrame({"image": val_images, "class": val_labels}),
    directory=data_dir,
    x_col="image",
    y_col="class",
    target_size=(img_size, img_size),
    batch_size=batch_size,
    class_mode="categorical",
    shuffle=False
)
```

5.4. Modelo 1

```
num_classes = len(np.unique(all_labels))

inputs = tf.keras.Input(shape=(img_size,img_size,3), name='input')
reescaling = layers.Rescaling(1. / 255)(inputs)

# Capas Convolucionales

conv_1 = layers.Conv2D(filters = 16, kernel_size = 3, padding='same',
↪  activation='relu', strides = 1, name='conv_1')(reescaling)
pool_1 = layers.AveragePooling2D(pool_size=(2, 2), name='pool_1')(conv_1)

conv_2 = layers.Conv2D(filters = 32, kernel_size = 3, padding='same',
↪  activation='relu', strides = 1, name='conv_2')(pool_1)
pool_2 = layers.AveragePooling2D(pool_size=(2, 2), name='pool_2')(conv_2)

#Capas Densas

flat_1 = layers.Flatten(name='flatten_1')(pool_2)
dense_1 = layers.Dense(32, activation='relu', name='dense_1')(flat_1)

flat_2 = layers.Flatten(name='flatten_2')(dense_1)
dense_2 = layers.Dense(16, activation='relu', name='dense_2')(flat_2)

outputs = layers.Dense(num_classes, activation='softmax', name='output')(dense_2)

model_1 = keras.Model(inputs=inputs, outputs=outputs, name='Model_1')

epochs = 8

model_1.compile(
    optimizer='adam',
    loss=tf.keras.losses.CategoricalCrossentropy(),
    metrics=['accuracy']
)

history = model_1.fit(
    train_ds,
    validation_data = val_ds,
    epochs=epochs
)
```

5.5. Script Validaciones

```python
# Predicciones y labels
pred_probs_1 = model_1.predict(val_ds)
pred_labels_1 = np.argmax(pred_probs_1, axis=1)

# Indices de buenas y malas predicciones
good_preds_1 = np.where(pred_labels_1 == val_ds.labels)[0]
bad_preds_1 = np.where(pred_labels_1 != val_ds.labels)[0]

# Elegir 2 índices de cada tipo de preddición aleatoriamente
good_pred_locs_1 = np.random.choice(good_preds_1, 2, replace=False)
bad_pred_locs_1 = np.random.choice(bad_preds_1, 2, replace=False)

# Diccionario para mapear indices con labels
class_map = val_ds.class_indices
labels_dict = {v: k for k, v in class_map.items()}

plt.figure(figsize=(10, 6))
for i, idx in enumerate(good_pred_locs_1):
    plt.subplot(2, 2, i + 1)
    plt.imshow(plt.imread(val_ds.filepaths[idx]))
    plt.title(f'Predicción: {labels_dict[pred_labels_1[idx]]}\nRealidad:
    ↪ {labels_dict[val_ds.labels[idx]]}\nArchivo:
    ↪ {os.path.basename(val_ds.filepaths[idx])}')
plt.suptitle('Buenas predicciones\n\n')
plt.show()

plt.figure(figsize=(10, 6))
for i, idx in enumerate(bad_pred_locs_1):
    plt.subplot(2, 2, i + 1)
    plt.imshow(plt.imread(val_ds.filepaths[idx]))
    plt.title(f'Predicción: {labels_dict[pred_labels_1[idx]]}\nRealidad:
    ↪ {labels_dict[val_ds.labels[idx]]}\nArchivo:
    ↪ {os.path.basename(val_ds.filepaths[idx])}')
plt.suptitle('Malas predicciones\n\n')
plt.show()
```

5.6. Modelo 2

```
inputs = tf.keras.Input(shape=(img_size,img_size,3), name='input')
reescaling = layers.Rescaling(1. / 255)(inputs)

# Capas Convolucionales
conv_1 = layers.Conv2D(filters = 64, kernel_size = 11, padding='same',
↪  activation='relu', strides = 2, name='conv_1')(reescaling)
pool_1 = layers.AveragePooling2D(pool_size=(3, 3), name='pool_1')(conv_1)

conv_2 = layers.Conv2D(filters = 128, kernel_size = 5, padding='same',
↪  activation='relu', strides = 1, name='conv_2')(pool_1)
pool_2 = layers.AveragePooling2D(pool_size=(2, 2), name='pool_2')(conv_2)

conv_3 = layers.Conv2D(filters = 256, kernel_size = 5, padding='same',
↪  activation='relu', strides = 1, name='conv_3')(pool_2)
pool_3 = layers.AveragePooling2D(pool_size=(2, 2), name='pool_3')(conv_3)

# Capas Densas

flat_1 = layers.Flatten(name='flatten_1')(pool_3)
dense_1 = layers.Dense(64, activation='relu', name='dense_1')(flat_1)
drop_1 = layers.Dropout(0.25, name = 'drop_1')(dense_1)

flat_2 = layers.Flatten(name='flatten_2')(drop_1)
dense_2 = layers.Dense(32, activation='relu', name='dense_2')(flat_2)
drop_2 = layers.Dropout(0.25, name = 'drop_2')(dense_2)

outputs = layers.Dense(num_classes, activation='softmax', name='output')(drop_2)

model_2 = keras.Model(inputs=inputs, outputs=outputs, name='Model_2')

epochs = 50

es_callback = keras.callbacks.EarlyStopping(
    monitor='val_loss',
    patience=int(epochs/10),
    verbose=1)

model_2.compile(
    optimizer='adam',
    loss=tf.keras.losses.CategoricalCrossentropy(),
    metrics=['accuracy']
)

history = model_2.fit(
    train_ds,
    validation_data = val_ds,
    epochs=epochs,
    callbacks=[es_callback]
)

```

5.7. Modelo 3

```
data_augmentation= keras.Sequential(
    [
        layers.experimental.preprocessing.RandomFlip(),
        layers.experimental.preprocessing.RandomRotation(0.25),
        layers.experimental.preprocessing.RandomZoom(0.25)
    ], name= 'data_aug'
)

inputs = tf.keras.Input(shape=(img_size,img_size,3), name='input')
data_aug= data_augmentation(inputs)
reescaling = layers.Rescaling(1. / 255)(data_aug)

# Capas Convolucionales

conv_1 = layers.Conv2D(filters = 64, kernel_size = 11, padding='same',
↪  activation='relu', strides = 2, name='conv_1')(reescaling)
pool_1 = layers.AveragePooling2D(pool_size=(3, 3), name='pool_1')(conv_1)

conv_2 = layers.Conv2D(filters = 128, kernel_size = 5, padding='same',
↪  activation='relu', strides = 1, name='conv_2')(pool_1)
pool_2 = layers.AveragePooling2D(pool_size=(2, 2), name='pool_2')(conv_2)

conv_3 = layers.Conv2D(filters = 256, kernel_size = 5, padding='same',
↪  activation='relu', strides = 1, name='conv_3')(pool_2)
pool_3 = layers.AveragePooling2D(pool_size=(2, 2), name='pool_3')(conv_3)

#Capas Densas

flat_1 = layers.Flatten(name='flatten_1')(pool_3)
dense_1 = layers.Dense(64, activation='relu', name='dense_1')(flat_1)
drop_1 = layers.Dropout(0.25, name = 'drop_1')(dense_1)

flat_2 = layers.Flatten(name='flatten_2')(drop_1)
dense_2 = layers.Dense(32, activation='relu', name='dense_2')(flat_2)
drop_2 = layers.Dropout(0.25, name = 'drop_2')(dense_2)

outputs = layers.Dense(num_classes, activation='softmax', name='output')(drop_2)

model_3 = keras.Model(inputs=inputs, outputs=outputs, name='Model_3')

epochs = 50

es_callback = keras.callbacks.EarlyStopping(
    monitor='val_loss',
    patience=int(epochs/10),
    verbose=1)
model_3.compile(
    optimizer='adam',
    loss=tf.keras.losses.CategoricalCrossentropy(),
    metrics=['accuracy']
)
history = model_3.fit(
    train_ds,
    validation_data = val_ds,
    epochs=epochs,
    callbacks=[es_callback]
)
```

5.8. Modelo 4

```
def inception(x,
              filters_1x1,
              filters_3x3_reduce,
              filters_3x3,
              filters_5x5_reduce,
              filters_5x5,
              filters_pool):
  path1 = layers.Conv2D(filters_1x1, (1, 1), padding='same',activation='relu')(x)

  path2 = layers.Conv2D(filters_3x3_reduce, (1, 1), padding='same',
  ↪  activation='relu')(x)
  path2 = layers.Conv2D(filters_3x3, (3, 3), padding='same',
  ↪  activation='relu')(path2)

  path3 = layers.Conv2D(filters_5x5_reduce, (1, 1), padding='same',
  ↪  activation='relu')(x)
  path3 = layers.Conv2D(filters_5x5, (5, 5), padding='same',
  ↪  activation='relu')(path3)

  path4 = layers.MaxPool2D((3, 3), strides=(1, 1), padding='same')(x)
  path4 = layers.Conv2D(filters_pool, (1, 1), padding='same',
  ↪  activation='relu')(path4)

  return tf.concat([path1, path2, path3, path4], axis=3)

data_augmentation= keras.Sequential(

    [
        layers.experimental.preprocessing.RandomFlip(),
        layers.experimental.preprocessing.RandomRotation(0.25),
        layers.experimental.preprocessing.RandomZoom(0.25)
    ], name= 'data_aug'

)

inputs = tf.keras.Input(shape=(img_size,img_size,3), name='input')
data_aug= data_augmentation(inputs)
reescaling = layers.Rescaling(1. / 255)(data_aug)

#Capas Convolucionales
x = layers.Conv2D(128, 7, strides=2, padding='same', activation='relu')(data_aug)
x = layers.AveragePooling2D(3, strides=2)(x)
x = layers.Conv2D(32, 1, strides=1, padding='same', activation='relu')(x)
x = layers.AveragePooling2D(3, strides=2)(x)
x = layers.Conv2D(32, 3, strides=1, padding='same', activation='relu')(x)
x = layers.AveragePooling2D(3, strides=2)(x)
x = inception(x, filters_1x1=32, filters_3x3_reduce=96, filters_3x3=128,
↪  filters_5x5_reduce=16, filters_5x5=32, filters_pool=32)
x = inception(x, filters_1x1=64, filters_3x3_reduce=128, filters_3x3=192,
↪  filters_5x5_reduce=32, filters_5x5=96, filters_pool=64)
x = inception(x, filters_1x1=128, filters_3x3_reduce=96, filters_3x3=208,
↪  filters_5x5_reduce=16, filters_5x5=48, filters_pool=64)
x = layers.AveragePooling2D(3, strides=2)(x)
x = inception(x, filters_1x1=32, filters_3x3_reduce=128, filters_3x3=256,
↪  filters_5x5_reduce=32, filters_5x5=64, filters_pool=64)
x = inception(x, filters_1x1=32, filters_3x3_reduce=128, filters_3x3=256,
↪  filters_5x5_reduce=32, filters_5x5=64, filters_pool=64)
```

```
x = inception(x, filters_1x1=32, filters_3x3_reduce=128, filters_3x3=256, filters_5x5_reduce=32, filters_5x5=64, filters_pool=64)
x = layers.AveragePooling2D(3, strides=2)(x)
x = inception(x, filters_1x1=128, filters_3x3_reduce=128, filters_3x3=256, filters_5x5_reduce=32, filters_5x5=128, filters_pool=128)
x = inception(x, filters_1x1=128, filters_3x3_reduce=128, filters_3x3=256, filters_5x5_reduce=32, filters_5x5=128, filters_pool=128)
x = inception(x, filters_1x1=128, filters_3x3_reduce=128, filters_3x3=256, filters_5x5_reduce=32, filters_5x5=128, filters_pool=128)
x = layers.GlobalAveragePooling2D()(x)
x = layers.Dropout(0.4)(x)

#Capas densas
x = layers.Flatten(name='flatten_1')(x)
x = layers.Dense(64, activation='relu', name='dense_1')(x)
x = layers.Dropout(0.25, name = 'drop_1')(x)

x = layers.Flatten(name='flatten_2')(x)
x = layers.Dense(32, activation='relu', name='dense_2')(x)
x = layers.Dropout(0.25, name = 'drop_2')(x)

outputs = layers.Dense(num_classes, activation='softmax', name='output')(x)

model_4_inception = keras.Model(inputs=inputs, outputs=outputs, name='Model_4_inception')


epochs = 200

es_callback = keras.callbacks.EarlyStopping(
    monitor='val_loss',
    patience=int(epochs/10),
    verbose=1)
model_4_inception.compile(
    optimizer='adam',
    loss=tf.keras.losses.CategoricalCrossentropy(),
    metrics=['accuracy']
)

history = model_4_inception.fit(
    train_ds,
    validation_data = val_ds,
    epochs=epochs,
    callbacks = [es_callback]
)
```

5.9. Modelo 5

```
import efficientnet.keras as efn

data_augmentation= keras.Sequential(

    [
        layers.experimental.preprocessing.RandomFlip(),
        layers.experimental.preprocessing.RandomRotation(0.25),
        layers.experimental.preprocessing.RandomZoom(0.25)
    ], name= 'data_aug'

)

B0efn = efn.EfficientNetB0(input_shape = (img_size, img_size, 3), include_top =
↪  False, weights = 'imagenet')

for layer in B0efn.layers:
    layer.trainable = False

inputs = tf.keras.Input(shape=(img_size,img_size,3), name='input')
data_aug= data_augmentation(inputs)
reescaling = layers.Rescaling(1. / 255)(data_aug)
x = B0efn.output
x = layers.Flatten()(x)
x = layers.Dense(1024, activation="relu")(x)
x = layers.Dropout(0.5)(x)

outputs = layers.Dense(num_classes, activation='softmax', name='output')(x)
model_5_EFN = keras.Model(inputs=B0efn.input, outputs=outputs, name='Model_5_EFN')

epochs = 200

es_callback = keras.callbacks.EarlyStopping(
    monitor='val_loss',
    patience=int(epochs/10),
    verbose=1)
model_5_EFN.compile(
    optimizer='adam',
    loss=tf.keras.losses.CategoricalCrossentropy(),
    metrics=['accuracy']
)

history = model_5_EFN.fit(
    train_ds,
    validation_data = val_ds,
    epochs=epochs,
    callbacks = [es_callback]
)
```