UNIVERSIDAD EUROPEA DE MADRID

ESCUELA DE ARQUITECTURA, INGENIERÍA Y DISEÑO

MÁSTER UNIVERSITARIO EN

BIG DATA ANALYTICS - MBI

TRABAJO FIN DE MÁSTER

CLASIFICADOR DE MARCAS MEDIANTE ALGORITMOS DE DEEP LEARNING

NOMBRE:

FERNANDO DEL POZO ITUERO

CURSO 2020-2021

Universidad Europea

TÍTULO: CLASIFICADOR DE MARCAS MEDIANTE ALGORITMOS DE DEEP LEARNING

AUTOR: FERNANDO DEL POZO ITUERO

TITULACIÓN: MÁSTER UNIVERSITARIO EN BIG DATA ANALYTICS

DIRECTOR DEL PROYECTO: LUIS FERNÁNDEZ ORTEGA

FECHA: OCTUBRE DE 2021

AGRADECIMIENTOS

A mis profesores y tutores, por enseñarme este maravilloso mundo de los datos en el cual me quiero convertir en un auténtico profesional.

A mis hermanos quiero darles las gracias por sus consejos y por toda su ayuda, siempre han sido todo un referente para mí.

A mi padre y a mi madre, por todo el apoyo que me han dado, por darme más de lo que he necesitado, gracias a vosotros estoy cumpliendo mis metas.

A mi novia, la cual se merecería que escribiera hojas enteras de las razones por las que darle las gracias.

RESUMEN

El presente Trabajo Fin de Máster pretende desarrollar un modelo de aprendizaje profundo (*Deep Learning*) basado en redes convolucionales, cuya finalidad es clasificar imágenes de marcas comerciales conocidas. Para ello, se parte de un número determinado de marcas y conjunto de imágenes previamente clasificadas en función de estas, las cuales se dividirán en dos grupos: uno de entrenamiento y otro de testeo. Finalmente, se pone a prueba el modelo desarrollado con nuevas imágenes, se sacan las principales conclusiones en base a los resultados del modelo realizado y se plantean futuras líneas de investigación del trabajo realizado.

PALABRAS CLAVES

Python

Clasificador

Aprendizaje profundo

Redes convolucionales

Transferencia de aprendizaje

ABSTRACT

This Master's thesis aims to develop a deep learning model based on convolutional networks, whose purpose is to classify images of well-known commercial brands. To do this, we start with a certain number of brands and a set of images previously classified according to these, which will be divided into two groups: one for training and the other for testing. Finally, the model developed is tested with new images, the main conclusions are drawn based on results of the model created and possible future lines of the work carried out are proposed.

KEY WORDS

Python

Classifier

Deep learning

Convolutional networks

Transfer learning

ÍNDICE

ÍNDICE DE FIGURAS:

ÍNDICE DE TABLAS

1. INTRODUCCIÓN

1.1. PLANTEAMIENTO

El presente Trabajo de Fin de Máster (TFM) plantea el desarrollo de un modelo de Machine Learning (Aprendizaje Automático) que realice la clasificación de imágenes en las que aparecen los logotipos de marcas comerciales conocidas.

La idea surge del reto lanzado por el Máster Universitario en Análisis de Grandes Cantidades de Datos de la Universidad Europea (UE) junto con la empresa International Business Machines Corporation (IBM), el cual pretende buscar la mejor solución para este problema.

En el presente TFM se pretende dar una solución a este reto, partiendo de lo más básico (selección del framework, selección del tipo de Machine Learning, etc.) hasta lo más complejo, siendo esto el desarrollo del propio modelo.

1.2. PROBLEMAS

Para el desarrollo de este algoritmo se afrontarán distintos problemas. Para ello, se parte de la elección de la mejor tecnología para hacer frente al reto de un mercado que, observando el landscape de Turck (2020), aporta una gran variedad de posibilidades (véase la figura 1).

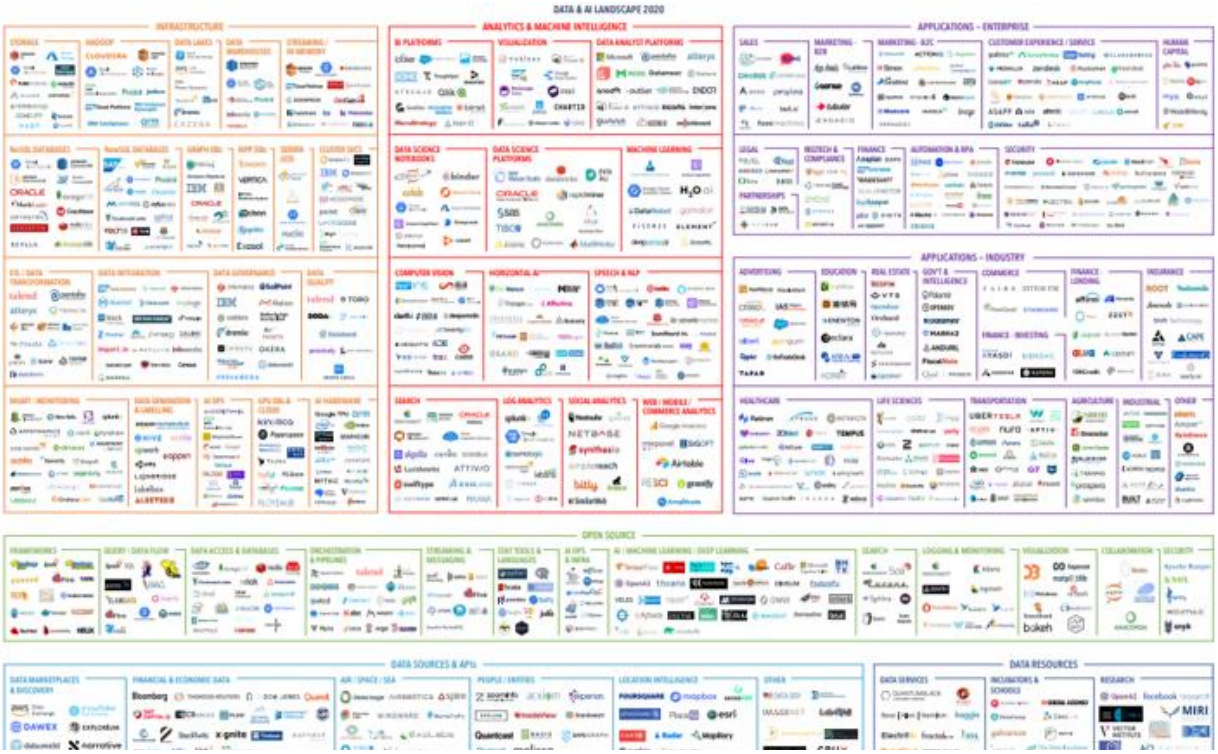


Figura 1: DATA & AI LANDSCAPE. Fuente: Turck, 2020.

Una vez seleccionada la tecnología, habrá que elegir el tipo de solución que se ha de tomar, es decir, especificar qué herramienta de Machine Learning se va a usar (árbol de decisión, redes neuronales, etc.).

Finalmente, habrá que entender los hiperparámetros que utiliza la herramienta escogida, desarrollar el modelo, regularizarlo y ajustarlo de forma que no se realice underfitting ni overfitting.

1.3. OBJETIVOS

Los objetivos del presente TFM son varios, siendo **el principal desarrollar el modelo que realice la clasificación de imágenes** de marcas superando un porcentaje mínimo de precisión.

Concretamente, se pretende llegar a una precisión superior al 85%, aunque esta precisión se podrá mejorar en futuras líneas del trabajo, donde se obtengan nuevos conocimientos/tecnologías que permitan llegar a él.

Como objetivos secundarios, y para contribuir a la consecución del principal, se pueden indicar los siguientes:

- **Obtener conocimiento y visión del Machine Learning,** realizando un estudio e investigación de la taxonomía, entendiendo las distintas clasificaciones y algoritmos centrándose en la posible solución del problema.
- **Elegir el tipo de algoritmo que se considere más apropiado para la solución.** Una vez obtenido el conocimiento, elegir el tipo de algoritmo que se considere más adecuado para desarrollar el modelo.
- **Entender el tipo de algoritmo elegido.** Una vez escogido este, analizar su funcionamiento para entenderlo: qué hiperparámetros usa, cómo afectan, cómo se recomienda su utilización etc.

Finalmente, se pretende extraer conclusiones sobre el cumplimiento de estos objetivos y valorar las posibles futuras líneas de desarrollo de este TFM.

2. MARCO TEÓRICO

2.1. ESTUDIO SOBRE EL MACHINE LEARNING

2.1.1. Taxonomía de algoritmos de Machine Learning

Aunque hay distintas clasificaciones para los algoritmos de Machine Learning, vamos a seguir la propuesta estudiada en el máster (ver figura 2) y que aparece en varias investigaciones, como la de González (2020).

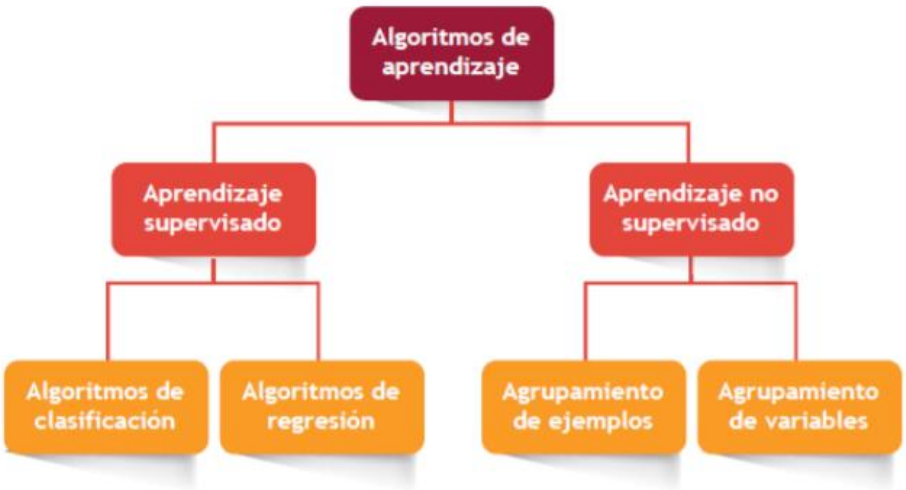


Figura 2: Organización de tipos de algoritmos de aprendizaje. Fuente: UE.

Según González (2020), hay dos grandes grupos de aprendizaje:

- **Aprendizaje supervisado**: es el aprendizaje orientado a predecir el valor de una variable objetivo a partir de las características de los datos.
- **Aprendizaje no supervisado**: el objetivo de este aprendizaje es explorar los datos para encontrar estructuras que permitan organizarlos, como, por ejemplo, agrupar los clientes por compras similares.

Dentro de cada uno de estos grupos hay otros dos subgrupos. Para el caso del aprendizaje supervisado tendremos los subgrupos de:

- **Algoritmos de clasificación:** la finalidad de estos algoritmos es predecir la categoría de la variable objetivo, es decir, a qué grupo pertenecen. Este tipo de algoritmos se aplica sobre variables categóricas. El ejemplo que se suele utilizar para estos algoritmos es el de la clasificación de los correos basura o spam.
- **Algoritmos de regresión:** se trata de algoritmos de predicción numérica que se suelen utilizar predecir el valor de una variable continua. Este tipo de algoritmos también se utilizan como algoritmos de clasificación (en función de si la variable objetivo supera un umbral definido, clasificándose o no en un grupo). Un ejemplo de este tipo de

algoritmos es la predicción de contagiados por el COVID-19 en una comunidad autónoma.

En lo referido al aprendizaje no supervisado, tenemos los siguientes subgrupos:

- **Agrupamiento de ejemplares:** buscan relaciones entre los distintos ejemplares (las filas de una tabla) para poder agruparlos de tal forma que, en un grupo, las características de estos ejemplares sean similares. Como ejemplo se puede poner la clasificación de los clientes de una tienda.
- **Agrupamiento de variables:** realizan lo mismo que el anterior grupo pero con las variables (columnas de una tabla), con el fin de determinar patrones y correlaciones entre las distintas variables. Un problema típico es el análisis de la cesta de la compra para correlacionar las variables (productos) en función de las compras.

2.1.2. Principales algoritmos

Una vez analizados los tipos de aprendizaje automático que hay, se determinan, a continuación, los principales algoritmos (Mahdavinejad et al., 2018) basados en estos tipos que ya se han estudiado (véase la figura 2).

Muchos de estos algoritmos (la mayoría) se han estudiado en el máster de Análisis de Grandes Cantidades de Datos de la UE, por lo que se prestará especial interés en estos algoritmos para implementar la solución del problema, ya que son estos de los que más conocimiento se tiene de ellos.

Machine learning algorithm	Data processing tasks
K-Nearest Neighbors	Classification
Naive Bayes	Classification
Support Vector Machine	Classification
Linear Regression	Regression
Support Vector Regression	Regression
Classification and Regression Trees	Classification/Regression
Random Forests	Classification/Regression
Bagging	Classification/Regression
K-Means	Clustering
Density-Based Spatial Clustering of Applications with Noise	Clustering
Principal Component Analysis	Feature extraction
Canonical Correlation Analysis	Feature extraction
Feed Forward Neural Network	Regression/Classification/Clustering/Feature extraction
One-class Support Vector Machines	Anomaly detection

Tabla 1: algoritmos frecuentemente utilizados en machine learning para data análisis. Fuente: Mahdavinejad et al., 2018.

2.2. ELECCIÓN DEL ALGORITMO

2.2.1. Filtrado de los tipos de algoritmos por objetivos

Por el momento, ya se tienen los algoritmos más frecuentes para la posible solución y se conoce la categoría a la que pertenecen (y, por consiguiente, el objetivo del algoritmo).

A continuación, se analizarán los tipos de algoritmos para determinar cuál se implementará o se tratará de implementar en el modelo. Para ello, se va a realizar un primer filtrado de tipos de algoritmos.

Siguiendo la presentación de Berzal (2015), donde en una de sus diapositivas (ver figura 3) indica las propiedades de los dos grandes grupos, se puede descartar el grupo de aprendizaje no supervisado (clustering), ya que para este problema sí que se cuenta con el grupo definido (que es el conjunto de marcas que se quieren detectar). Dentro de este grupo también se encuentran los clasificados como Feature extraction.

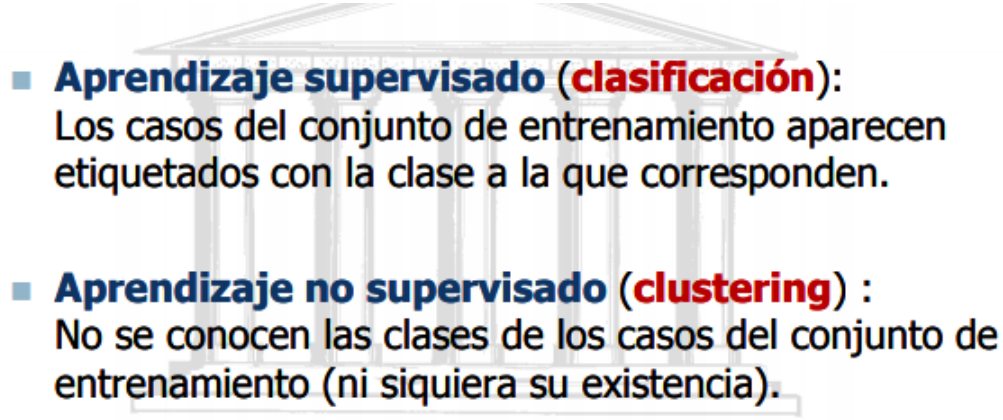


Figura 3: Propiedades del aprendizaje supervisado y no supervisado. Fuente: Berzal, 2015.

El resultado de este primer filtrado deja como posibles opciones las que se muestran en la tabla 2:

Machine learnin algorithm	Data processing tasks
K-Nearest Neighbors	Classification
Naive Bayes	Classification
Support Vector Machine	Classification
Linear Regression	Regression
Support Vector Regression	Regression
Classification and Regression Trees	Classification/Regression
Random Forests	Classification/Regression
Bagging	Classification/Regression
Feed Forward Neural Network	Regression/Classification/Clustering/Feature extraction

Tabla 2: Algoritmos tras el primer filtrado. Fuente: elaboración propia.

Como puede observarse, se ha reducido bastante la lista. Para seguir con la elección, se realizará un segundo filtrado; esta vez se descartarán los algoritmos dedicados a la regresión. Esto es porque son algoritmos que requieren una mayor capacidad de cómputo y una valoración extra para determinar los umbrales en las clasificaciones. Por esta razón, se escogerán algoritmos dedicados a la clasificación y se eliminarán los de regresión (véase tabla 3).

Machine learnin algorithm	Data processing tasks
K-Nearest Neighbors	Classification
Naive Bayes	Classification
Support Vector Machine	Classification
Classification and Regression Trees	Classification/Regression
Random Forests	Classification/Regression
Bagging	Classification/Regression
Feed Forward Neural Network	Regression/Classification/ Clustering/Feature extraction

Tabla 3: Algoritmos tras el segundo filtrado. Fuente: elaboración propia.

Finalmente, como se menciona en el apartado anterior, se mantendrán los algoritmos con los que se han trabajado durante el máster de la UE, descartado aquellos que no se han visto o que solo han sido mencionados (véase la tabla 4).

Machine learnin algorithm	Data processing tasks
Naive Bayes	Classification
Support Vector Machine	Classification
Classification and Regression Trees	Classification/Regression
Feed Forward Neural Network	Regression/Classification/ Clustering/Feature extraction

Tabla 4: Algoritmos tras el último filtrado. Fuente: elaboración propia.

2.2.2. Elección del tipo de algoritmo para la solución

Habiéndose conseguido cuatro algoritmos válidos y competitivos para la resolución del problema, la elección, a continuación, se basa en intentar escoger el mejor para nuestra solución, aunque lo ideal sería probar estos cuatro algoritmos y someterlos a evaluaciones de aprendizaje automático (efectividad, precisión, matriz de confusión, etc.). Debido a su complejidad y al

no plantearse como uno de los objetivos principales de este trabajo, se escogerá por las características de los algoritmos.

En primer lugar, se analiza Bayes ingenuo. Este está basado en el teorema de Bayes que, aunque lanza una hipótesis falsa sobre la independencia de variables, es muy útil debido a su simplicidad.

El inconveniente que presenta es que muchas veces no llega al grado de precisión de algoritmos más complejos, como el de las redes neuronales. Como la precisión es uno de los elementos más valorados en el reto, descartamos Naive Bayes.

Por otro lado, Clasification Trees es un modelo muy aleatorio. De este tipo de algoritmos destaca Random Forest, el cual no se ha trabajado en el máster, por lo que tampoco se escogerá este tipo de algoritmos.

Finalmente, la decisión está entre las redes neuronales y SVM. No hay un criterio que sea eliminatorio para ninguno de los dos y lo ideal sería hacer pruebas. Para el presente TFM, se optará finalmente por la utilización de redes neuronales, ya que estas son muy buenas para la detección de patrones (que es lo que se necesita para la resolución de este problema) y pueden alcanzar grados de precisión muy elevados, siendo el principal inconveniente la capacidad de cómputo que necesitan.

2.3. ESTUDIO SOBRE LAS REDES NEURONALES

2.3.1. Funcionamiento de las redes neuronales

Una vez decido el uso de las redes neuronales para la solución del problema, ahora hay que decidir qué red neuronal se va a desarrollar.

Concretamente, dentro de las redes neuronales están los algoritmos de aprendizaje profundo (del inglés Deep Learning), los cuales podemos considerar la evolución de estas redes.

El Deep Learning permite la ejecución de varias capas de la red neuronal en paralelo, y un tipo de Deep Learning que se verá más adelante tiene la capacidad de reconocer “figuras” dentro de imágenes, siendo esta la solución que se necesita.

Pero ¿cómo es el funcionamiento de una red neuronal? Básicamente estas neuronas imitan las neuronas de nuestro cerebro, pues reciben entradas que transforman en salidas (impulsos eléctricos en el caso de las neuronas cerebrales).

Las redes neuronales reciben valores de entrada (input layer) y dan lugar a salidas (ouput layers), habiendo procesos de por medio (hidden layers). Esto da lugar a una red neuronal (Matich, 2001) (véase la figura 4).

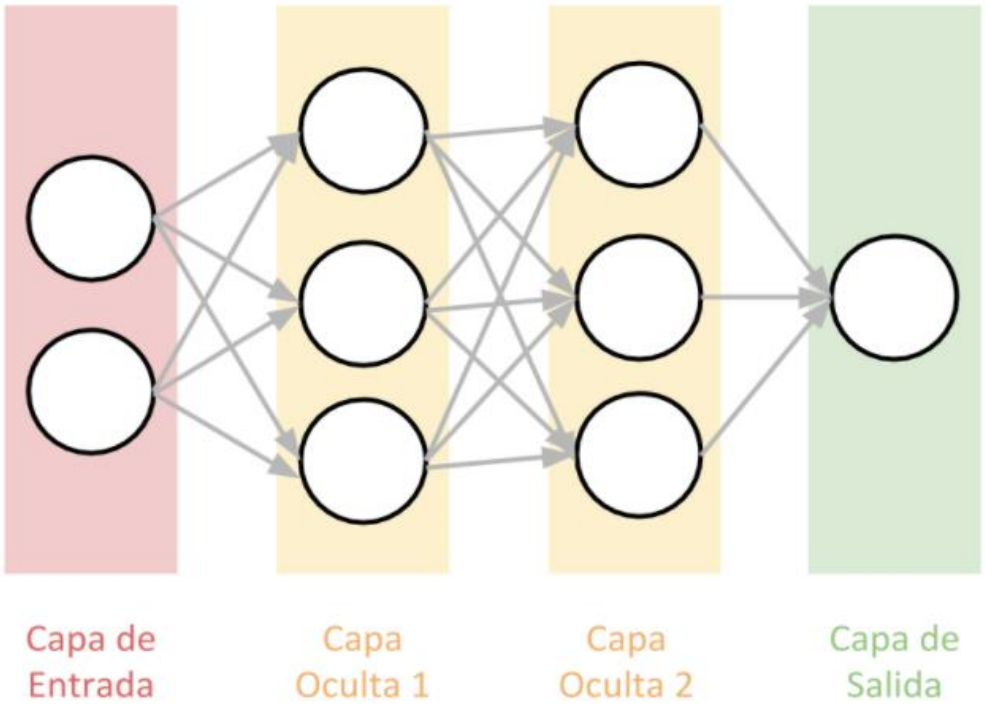


Figura 4: Esquema de capas de una red neuronal. Fuente: AprendeMachineLearning.

La capa de entrada recibe los datos, que los pasa a la primera capa oculta (hay que elegir el número de capas ocultas y la cantidad de neuronas) y esta se la pasará a la siguiente capa, así hasta llegar a la capa de salida.

Cada conexión de neurona representa un peso. Este peso simboliza la importancia de esa conexión en la neurona (véase la figura 5), y para que la neurona se active y mande una señal, es necesario que supere el umbral de la función de la neurona (López y Fernández, 2008) (véase la figura 6).

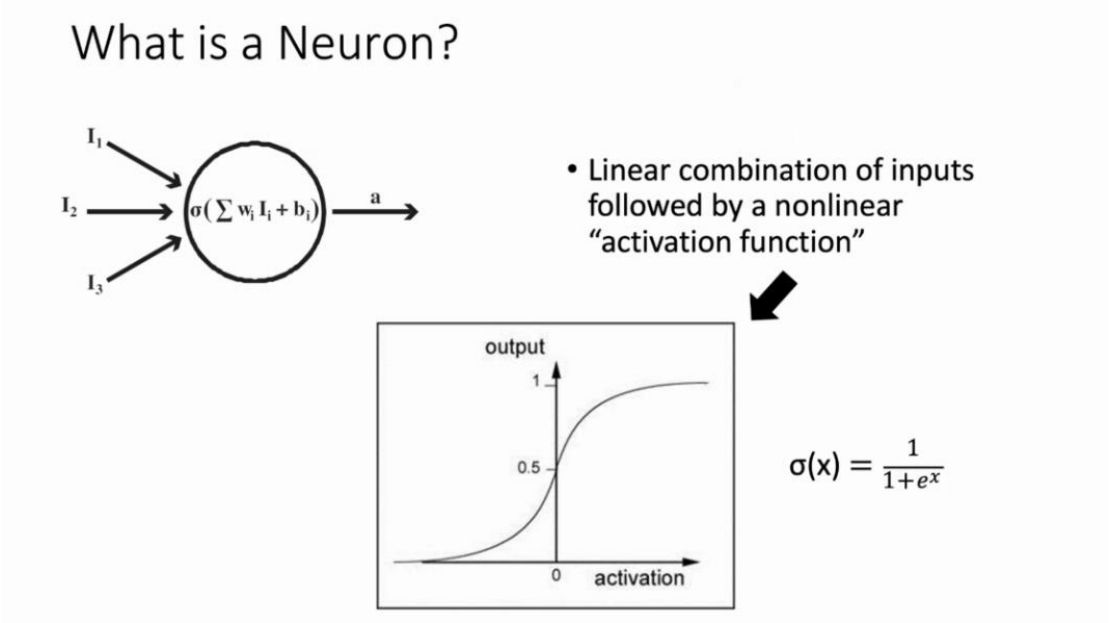


Figura 5: Distribución de pesos de una red neuronal. Fuente: UE.

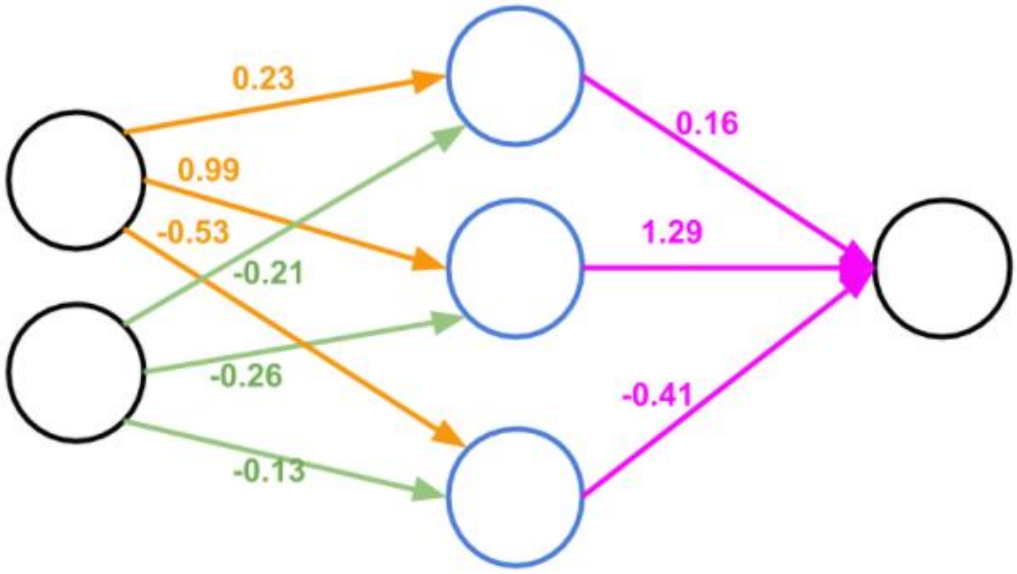


Figura 6: Distribución de pesos de una red neuronal. Fuente: AprendeMachineLearning.

Estos pesos inicialmente serán aleatorios y se irán ajustando durante el proceso de entrenamiento.

¿Cómo se ajustan? Utilizando una función denominada función coste, la cual indica qué tan bueno o malo es la predicción realizada. A mayor coste mayor error y peor predicción, por lo que interesa ajustar las neuronas hasta que el valor sea lo más próximo a 0. A este proceso se le llama Back-propagation.

Esta operación consiste en ir buscando el gradiente descendiente de la función coste (véase la figura 7). Para ello, se van dando valores en pequeños intervalos a los pesos y se obtiene la derivada del intervalo de pesos para ver en qué dirección descender (ver figura 8).

$$J(\mathbf{w}) = \frac{1}{2}\sum_i (\text{target}^{(i)} - \text{output}^{(i)})^2$$

Figura 7: Función coste. Fuente:GitHub

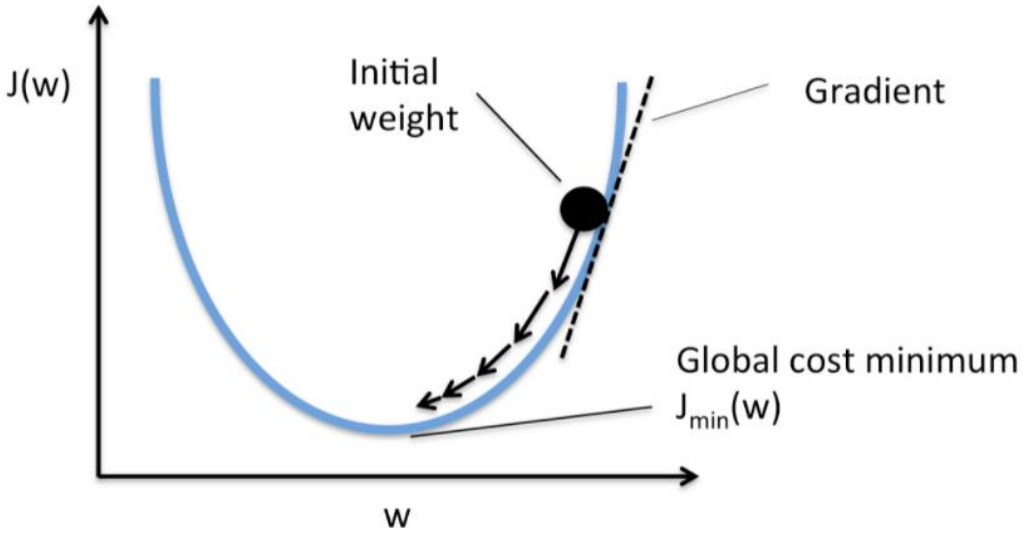


Figura 8: Gradient Descent. Fuente GitHub

Ahora que ya sabemos cómo funciona una red neuronal, se debe especificar qué red neuronal vamos a utilizar, que en este caso será un algoritmo de Convolutional Neural Network (CNN).

2.3.2. Convolutional Neural Networks

Es una red neuronal que procesa capas imitando al ojo humano. Para ello, contiene varias capas ocultas especializadas y jerárquicas que pueden detectar líneas, curvas, y con las suficientes capas detecta formas (LeCun, Bengio, & Hinton, 2015).

En primer lugar, esta red recibe como entrada las imágenes. Concretamente recibe los píxeles de estas imágenes, que en el caso de ser a color recibiría 3 valores por cada píxel (3 canales, red, green y blue). No obstante, una posible opción para el reto es transformar las imágenes a blanco y negro y así solo obtener por cada píxel un valor, pero se reduciría la información.

El siguiente paso es realizar convoluciones sobre la imagen, que consiste en pasar un kernel (una matriz de X*X de tamaño) por toda la imagen realizando el producto escalar por cada una de ellas, generando una nueva matriz de salida. En la figura 9 se ve una operación que está realizando un kernel de 3x3:

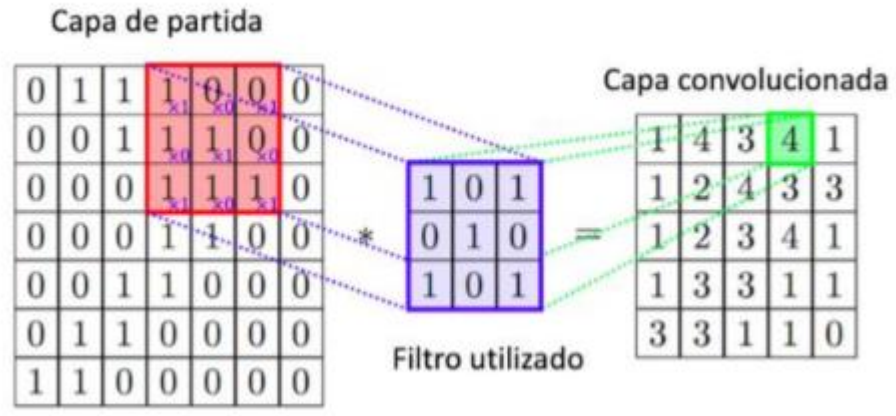


Figura 9:Kernel pasando por la imagen. Fuente: Artola, 2019.

Esta operación, en realidad, no se hace con un único kernel, sino con varios kernel de distintos valores que dan lugar a una gran variedad de neuronas ocultas.

A continuación, se aplica sobre estas matrices obtenidas una función de activación, siendo la más conocida ReLu, la cual también se utilizará como función de activación para la solución de este problema.

Hay que tener en cuenta que ahora hay un gran número de neuronas, dado que se han aplicado varios Kernel. Para simplificar en las próximas operaciones, se realiza un simplificado llamado Subsampling.

Este proceso de Subsampling consiste en reducir las matrices iniciales. Por ejemplo, con Max-Pooling se divide la imagen en matrices más pequeñas y se queda con el mayor valor obtenido (He, Zhang, Ren & Sun, 2015) (véase la figura 10).

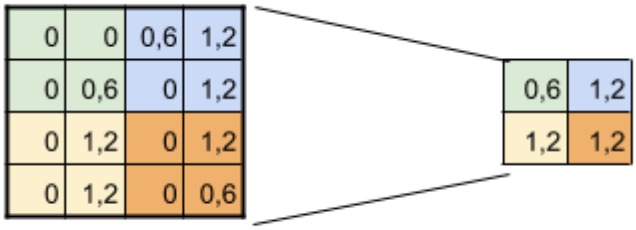


Figura 10: Max-Pooling 2x2 Fuente: AprendeMachineLearning.

Este primer proceso que se ha realizado permite obtener elementos sencillos, como líneas o curvas. El resumen de esta primera convolución se muestra en la figura 11.

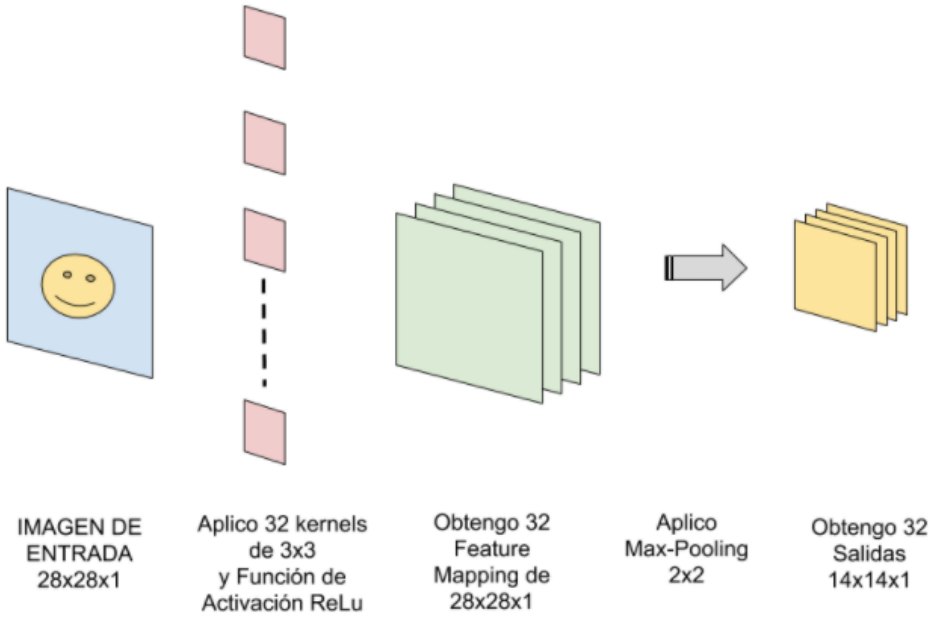


Figura 11: Proceso de convolución. Fuente: AprendeMachineLearning.

El siguiente paso es repetir este proceso, es decir, seguir realizando convoluciones hasta conseguir una imagen muy reducida. A mayor convoluciones mayor profundidad y más información se puede extraer.

Esta última imagen reducida se extrae y se aplana, de forma que se conecta con una red neuronal tradicional (como las que hemos visto en el punto 1 de este apartado) y dará lugar a la salida de la predicción (Calvo, 2017) (véase la figura 12).

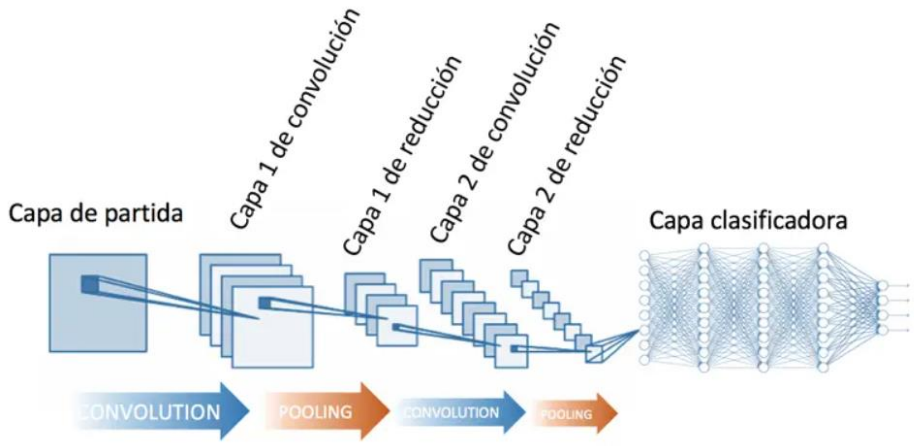


Figura 12: Proceso completo de una CNN. Fuente: Calvo, 2017.

Pero ¿cómo aprende la CNN? Si en las redes neuronales tradicionales se modificaba el peso de las conexiones, aquí se modifica el peso de la matriz kernel, realizando para ello la misma metodología para su corrección (backpropagation).

2.3.3. CNN más populares

2.3.3.1. LetNet

Modelo de red neuronal convolucional desarrollada en el año 1998 por el equipo de Yann LeCun, capaz de detectar caracteres utilizando el backpropagation (LeCun, Bottou, Bengio y Haffner, 1998).

LetNet asienta las bases de las redes convolucionales, siendo una de las primeras redes de este tipo. Está formada por numerosas capas de convolución y pooling, teniendo al final capas de fully connected.

2.3.3.2. AlexNet

AlexNet es una CNN que surge del desafío de reconocimiento visual de 2012 de ImageNet.

Se trata de una red que proporciona mucha profundidad al modelo aumentando el cómputo, pero manejable por la utilización de unidades de procesamiento gráficos (GPU) (véase la figura 13).

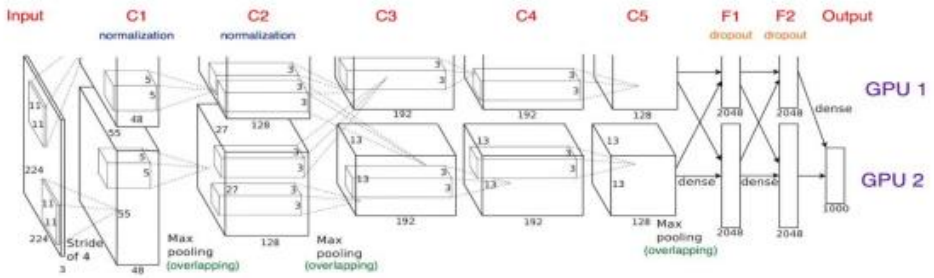


Figura 13: Arquitectura AlexNet. Fuente: Delbracio, Lezama y Carbajal, 2017

Una aportación importante es el uso de la función de activación ReLU, que aporta una mayor no linealidad. También destaca por el uso de dropout en la regularización (Delbracio, Lezama y Carbajal, 2017).

2.3.3.3. VGG16

VGG16 es un modelo de red neuronal convolucional que alcanza una precisión de prueba del 92,5%, entrenada con un conjunto de más de 14 millones de imágenes a clasificar en 1000 clases (Simonyan y Zisserman, 2014).

Es similar a AlexNet, pero siempre utiliza en sus trece capas de convolución un kernel de 3x3. Al final de la red cuenta con un clasificador de tres capas fully connected con función de activación ReLU (véase la figura 14).

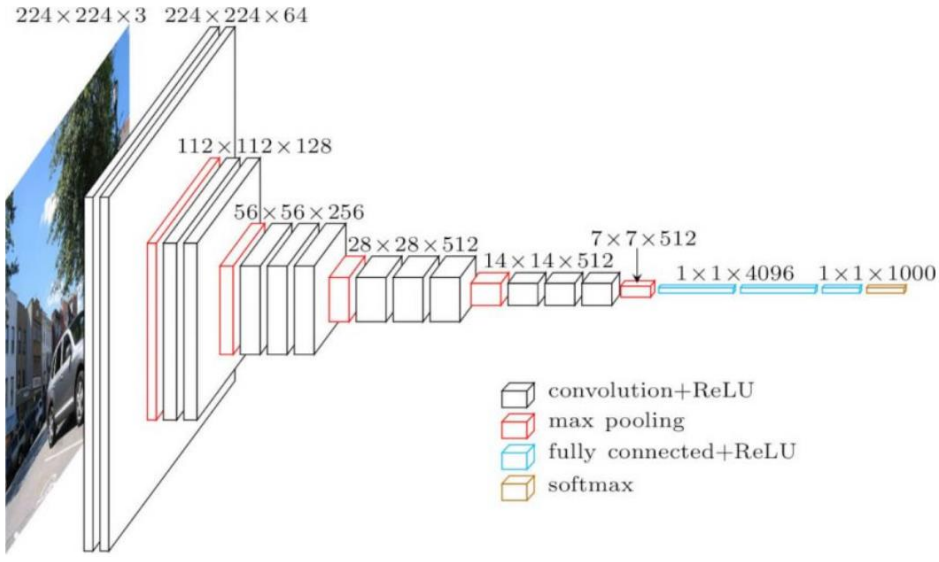


Figura 14: Arquitectura VGG16. Fuente: neurohive.

2.3.3.4. GoogLeNet

Es una red diseñada por Google para ser eficiente tanto en memoria como en cómputo, teniendo hasta 12 veces menos parámetros que AlexNet.

Concretamente consta de 27 capas de las cuales 5 son de agrupación (Alake, 2014).

Lo que hace GoogLeNet es reducir la imagen de entrada, pero reteniendo la información mas importante. GoogLeNet usa sus propias redes de inception (véase la figura 15)

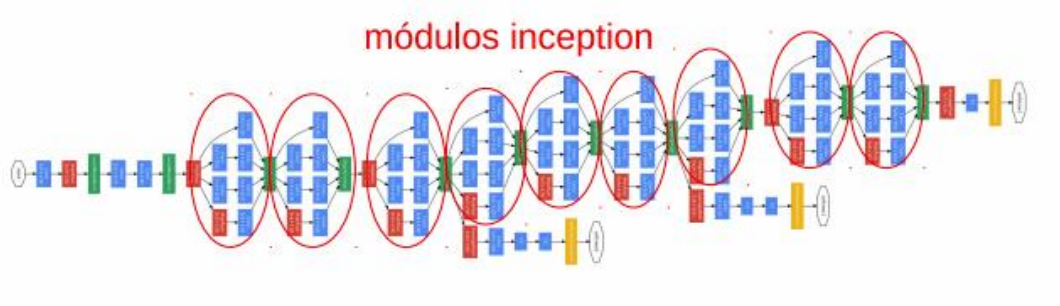


Figura 15: Arquitectura GoogLeNet. Fuente: Delbracio, Lezama y Carbajal, 201.

GoogLeNet también destaca por incorporar clasificadores auxiliares que solo se activan durante el entrenamiento.

2.3.3.5. ResNet

Estas redes proponen insertar bloques residuales en los que las capas intermedias de un bloque aprendan una función residual respecto al input del bloque. Se trata de una mejora en la que se aprende cómo ajustar el mapa de características con la finalidad obtener una mejor calidad. Si no se necesita este "refinamiento", los pesos de esta función residual se irán ajustando a cero, de forma que se comporten como una función de identidad (es decir, que no influyen).

De esta forma, las redes residuales (residual networks) aprenden funciones complejas y obtienen un mejor rendimiento dando solución al problema de degradación (cuando una red neuronal profunda converge a un error mayor que una red neuronal menos profunda).

3. ENTORNO DE TRABAJO

En primer lugar, se utilizará Anaconda, que como se indica en su web, es un entorno de trabajo creado por data scientists para data scientists y proporciona muchas facilidades para el desarrollo de modelos de machine learning.

Es un software de interés para esta solución por la facilidad de crear ambientes de trabajo, viene con Python 3 y también trae incorporado Jupyter Notebook.

2.1. PYTHON

Creado por Guido Van Rossum en 1991, Python es actualmente uno de los lenguajes de programación más populares del mundo y cada vez más (Challenger-Pérez, Díaz-Ricardo y Becerra-García, 2014). Es una opción muy recomendada para el desarrollo del software libre y sobre todo para Deep Learning gracias a los numerosos paquetes con los que cuenta para esta finalidad, como ThensorFlow y Keras.

Algunas ideas de su cultura según Challenger-Pérez, Díaz-Ricardo y Becerra-García (2014) son:

- Hermoso es mejor que feo.
- Simple es mejor que complejo.
- El código legible cuenta.
- Plano es mejor que anidado.

2.2. KERAS

Keras es una API de redes neuronales de código abierto escrita en Python que es capaz de ejecutarse sobre TensorFlow (como es el caso de este TFM), Theano, etc.

Como dice su lema, está diseñado para los seres humanos, no para la máquina, ya que se trata de una API sencilla que permite experimentar de forma rápida, pasando de la idea a la ejecución en el menor tiempo posible,

Keras admite tanto redes convolucionales como recurrentes, así como la combinación de estas, y funciona tanto en CPU´s como en GPU´s (Ketkar, 2017).

2.3. TENSORFLOW

Se trata de una biblioteca desarrollada por Google de código abierto destinada al desarrollo del machine learning. Se utilizará en este proyecto para hacer tanto llamadas a la API de Keras, como alguna de sus funciones (Shukla yFricklas, 2018).

Una herramienta de gran utilidad de TensorFlow son sus tensores (tensors). Un tensor es un conjunto de valores primitivos organizados por un array donde el rango del tensor es el número de dimensiones (Artola, 2019).

4. DESARROLLO DEL MODELO

4.1. PREPARACIÓN DEL SET DE IMÁGENES

En primer lugar, se ha hecho la división del pool de imágenes proporcionado por IBM para este trabajo. Algo bastante estandarizado es dividir el conjunto en un 80% para entrenamiento y un 20% para test. En este caso, utilizaremos un 80% para entrenamiento, un 15% para test y un 5% para realizar pruebas.

El set tiene que quedar divido en tres carpetas para cada uno de los grupos, y en cada grupo, las imágenes deben estar clasificadas por cada una de las posibles marcas o clases:

- AMD
- Aquafina
- Disney
- D-Link
- Domino's Pizza
- Hellman's
- IBM
- Kitkat
- LG
- Lipton
- McDonald's
- Milka
- Monster
- Nestea
- Pac-man
- Pepsi
- Pizza Hut
- Red Bull
- Samsung
- Sony
- Tic Tac
- Universal

Tras realizar esta división, aproximadamente, tenemos para cada marca 56 imágenes de entrenamiento, 11 de validación y 3 para pruebas.

4.2. PRIMERA VERSIÓN DEL MODELO

En esta primera versión se realiza la primera toma de contacto con la librería de TensorFlow, solucionando los problemas más básicos de nuestro entorno de trabajo.

La idea inicial de esta primera versión consistía en realizar el algoritmo completo "manualmente" del código. Para ello, se pretendía crear la capa de entrada, seguida de una capa de convolución con un tamaño de kernel 3x3, seguida de capa de pooling, concretamente de una capa de MaxPooling con una matriz de 2x2 y repitiendo estas dos últimas capas dos veces más, conectando la última capa a la de salida, la cual tiene las 22 clases posibles en esta solución (véase la figura 16). Adicionalmente, se añadió antes de la capa de salida una capa Dense con 256 neuronas.

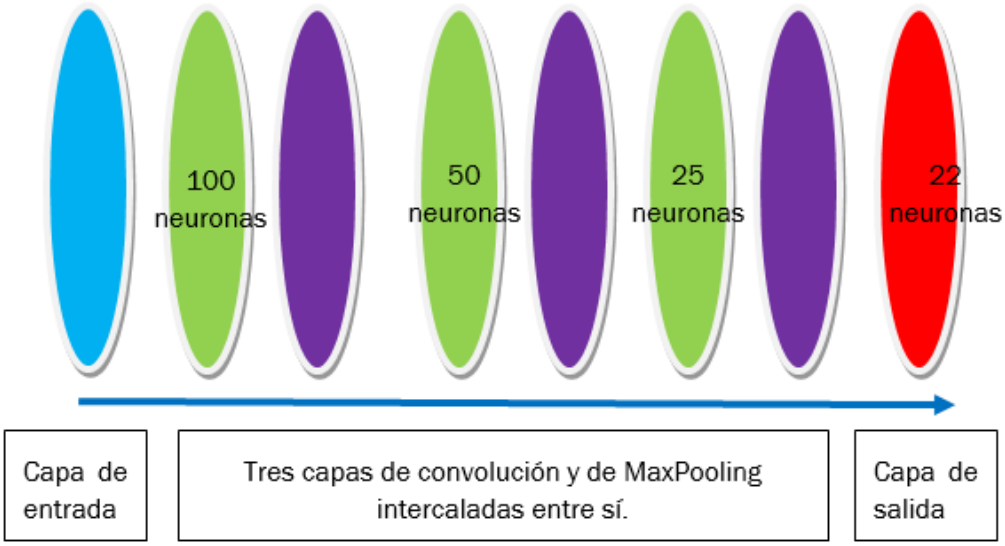


Figura 16: Primera solución propuesta. Fuente: elaboración propia.

Se han utilizado guías y trabajos ya realizados como referencia para elaborar esta primera red convolucional (Goldsborough, 2016; Zeng, Gong, & Zhang, 2019 y Ertam & Aydın, 2017).

Como ya se ha mencionado, el código se realizará en el entorno de trabajo de Jupyter Notebook con TensorFlow y se realizarán llamadas al API de Keras.

Lo primero a desarrollar en el código es importar las librerías (véase la figura 17).

```
In [1]: # Importamos todas las librerarías que vamos a necesitar para ejecutar nuestro modelo

In [2]: import sys
        import os
        import tensorflow as tf
        from tensorflow.python.keras.preprocessing.image import ImageDataGenerator
        from tensorflow.python.keras import optimizers
        from tensorflow.python.keras.models import Sequential
        from tensorflow.python.keras.layers import Dropout, Flatten, Dense, Activation
        from tensorflow.python.keras.layers import Convolution2D, MaxPooling2D
        from tensorflow.python.keras import backend as K
```

Figura 17: Librerías del primer modelo. Fuente: elaboración propia.

Cada una se importa por las siguientes razones:

- Sys y os: para poder moverse dentro de las carpetas de nuestro sistema operativo.
- Tensorflow: librería de tensorflow. Herramienta que se utiliza para el Deep learning de este trabajo.
- ImageDataGenerator: permite realizar tratamiento de imágenes. Se utilizará para preprocesar las imágenes.
- Optimizers: conjunto de optimizadores con los cuales vamos a entrenar nuestro algoritmo.
- Sequential: librería que permite realizar redes neuronales secuenciales (las capas siguen un orden).
- Dropout: capa que “mata neuronas”.
- Flatten: capa utilizada para aplanar un tensor.
- Dense: capa fully connected.
- Convolution2D: librería para importar las capas de convolución.
- MaxPooling2D: librería para importar las capas de pooling.
- Backend: se utilizará para poder “matar” los procesos de keras que estén ejecución.

Tras la ejecución de la celda saltó el error que se muestra en la figura 18.

```
In [2]: import sys
        import os
        import tensorflow as tf
        from tensorflow.python.keras.preprocessing.image import ImageDataGenerator
        from tensorflow.python.keras import optimizers
        from tensorflow.python.keras.models import Sequential
        from tensorflow.python.keras.layers import Dropout, Flatten, Dense, Activation
        from tensorflow.python.keras.layers import Convolution2D, MaxPooling2D
        from tensorflow.python.keras import backend as K

        ---------------------------------------------------------------------------
        ModuleNotFoundError                       Traceback (most recent call last)
        <ipython-input-2-6a0aa866a5d1> in <module>
              1 import sys
              2 import os
        ----> 3 import tensorflow as tf
              4 from tensorflow.python.keras.preprocessing.image import ImageDataGenerator
              5 from tensorflow.python.keras import optimizers

        ModuleNotFoundError: No module named 'tensorflow'
```

Figura 18: Error por falta de instalación de tensorflow. Fuente: elaboración propia.

De este error se aprende que es necesario crear un ambiente con las herramientas necesarias instaladas. Concretamente, se está utilizando Anaconda, por lo que se crea un nuevo ambiente llamado tf en el que se instalan todos los módulos necesarios para la ejecución de este trabajo. Estos módulos se instalan a medida que saltan errores indicando la ausencia de estos (véanse las figuras 19, 20 y 21).

```
(base) C:\Users\ferna>conda create -n tf tensorflow
WARNING: A conda environment already exists at 'D:\Users\ferna\anaconda3\envs\tf'
Remove existing environment (y/[n])? y

Collecting package metadata (current_repodata.json): done
Solving environment: failed with repodata from current_repodata.json, will retry with next repodata source.
Collecting package metadata (repodata.json): done
Solving environment: done
```

Figura 19: Instalación de tensorflow y creación de ambiente tf. Fuente: elaboración propia.

```
(tf) C:\Users\ferna>pip install pillow
Collecting pillow
  Downloading Pillow-8.3.2-cp39-cp39-win_amd64.whl (3.2 MB)
     |████████████████████████████████| 3.2 MB 2.2 MB/s
Installing collected packages: pillow
Successfully installed pillow-8.3.2
```

Figura 20: Instalación pipllow en el ambiente tf. Fuente: elaboración propia.

```
(tf) C:\Users\ferna>
(tf) C:\Users\ferna>pip install keras
Collecting keras
  Downloading keras-2.6.0-py2.py3-none-any.whl (1.3 MB)
     |████████████████████████████████| 1.3 MB 3.3 MB/s
Installing collected packages: keras
ERROR: pip's dependency resolver does not currently take into account all the packages that are installed. This behaviour is the source of the following dependency conflicts.
tensorflow 2.6.0 requires clang~=5.0, which is not installed.
tensorflow 2.6.0 requires flatbuffers~=1.12, but you have flatbuffers 20210226132247 which is incompatible.
tensorflow 2.6.0 requires grpcio<2.0,>=1.37.0, but you have grpcio 1.36.1 which is incompatible.
Successfully installed keras-2.6.0
```

Figura 21: instalación de keras en el ambiente tf. Fuente: elaboración propia.

Una vez realizadas estas instalaciones, activamos tf y abrimos y ejecutamos el código desde este ambiente, solucionando así este problema (véase la figura 22).

```
(base) C:\Users\ferna>conda activate tf

(tf) C:\Users\ferna>Jupyter notebook
[I 13:16:09.306 NotebookApp] The port 8888 is already in use, trying another port.
[I 13:16:09.335 NotebookApp] Serving notebooks from local directory: C:\Users\ferna
[I 13:16:09.335 NotebookApp] Jupyter Notebook 6.4.3 is running at:
[I 13:16:09.335 NotebookApp] http://localhost:8889/?token=793d022fcc8b7a84779955da5cfde3c1634ff721ad645f25
[I 13:16:09.336 NotebookApp]  or http://127.0.0.1:8889/?token=793d022fcc8b7a84779955da5cfde3c1634ff721ad645f25
[I 13:16:09.336 NotebookApp] Use Control-C to stop this server and shut down all kernels (twice to skip confirmation).
[C 13:16:09.396 NotebookApp]

    To access the notebook, open this file in a browser:
        file:///C:/Users/ferna/AppData/Roaming/jupyter/runtime/nbserver-9532-open.html
    Or copy and paste one of these URLs:
        http://localhost:8889/?token=793d022fcc8b7a84779955da5cfde3c1634ff721ad645f25
     or http://127.0.0.1:8889/?token=793d022fcc8b7a84779955da5cfde3c1634ff721ad645f25
D:\Users\ferna\anaconda3\envs\tf\lib\json\encoder.py:257: UserWarning: date_default is deprecated since jupyter_client 7
.0.0. Use jupyter_client.jsonutil.json_default.
  return _iterencode(o, 0)
```

Figura 22: activación del ambiente. Fuente: elaboración propia.

A continuación, limpiamos sesión y se declaran las variables con la ruta de entrenamiento y validación (véase la figura 23).

```
In [3]: #Se limpia sesión

In [4]: K.clear_session()

In [5]: #Se especifica la ruta del set de entrenamiento y validación

In [6]: datos_entrenamiento='./data/entrenamiento'
        datos_validacion='./data/validacion'
```

Figura 23: Limpieza de sesión y declaración de rutas del set de entrenamiento y validación. Fuente: elaboración propia.

El siguiente es paso es definir tanto los parámetros como los hiperparámetros que va a utilizar el modelo (véase la figura 24).

```
In [7]: #Se especifican los parametros, tanto de las imagenes que vamos a utilizar como de la red neuronal

In [8]: epocas=20
        ancho,alto=256,256
        batch_size=28
        filtrosConvolucion1=100
        filtrosConvolucion2=50
        filtrosConvolucion3=25
        tamano_kernel=(3,3)
        tamano_MaxPool=(2,2)
        clases=22
```

Figura 24: Parámetros del primer modelo de aprendizaje. Fuente: elaboración propia.

Las épocas son los llamados Epochs, que consiste en el número de pasadas que realiza el modelo por todo el set de imágenes. Ancho y alto corresponden a las dimensiones de las imágenes y el batch_size se basa en la división del set de imágenes en grupos más pequeños, con la finalidad de optimizar en memoria; por cada vez que se recorre un batch se actualizan los parámetros de las neuronas (pesos y bias). Los filtros de convolución es el número de neuronas por capa; el tamano_kernel es la dimensión de la matriz del kernel de convolución, y el tamano_MaxPool la dimensión del pooling. Finalmente, las clases es el número de marcas que se tienen.

Seguidamente, se realiza el tratamiento de imágenes, en el cual se normaliza el valor de los píxeles (un píxel toma un valor entre 0 y 255 en función de su intensidad; para normalizarlo lo dividimos por 255). Indicamos los parámetros anteriormente mencionados (ancho, alto, batch_size) e indicamos la clasificación es de tipo categórico (no se quiere un valor concreto, sino determinar a qué grupo pertenece) (véase la figura 25).

```
In [10]: #Normalizamos los valores de los pixeles que van de 0 a 255
         entrenamiento_datagen= ImageDataGenerator(
         rescale=1./255
         )

In [12]: entrenamiento_generator=entrenamiento_datagen.flow_from_directory(
             datos_entrenamiento,
             target_size=(alto,ancho),
             batch_size=batch_size,
             class_mode='categorical'
         )

         Found 1234 images belonging to 22 classes.

In [13]: validacion_generator=entrenamiento_datagen.flow_from_directory(
             datos_validacion,
             target_size=(alto,ancho),
             batch_size=batch_size,
             class_mode='categorical'
         )

         Found 241 images belonging to 22 classes.
```

Figura 25: Tratamiento de imágenes primer modelo. Fuente elaboración propia.

En el siguiente paso se diseña ya la red neuronal, indicando que es una CNN de tipo secuencial se van creando y añadiendo una a una las capas que se han mencionado al principio del apartado (3 convolucionales y 3 de pooling).

En cada una de las capas de convolución también se especifica el relleno padding, que en este caso siempre mantenemos las dimensiones, y la función de activación será la más común: ReLU.

Sin entrar mucho en detalle, la función de activación ReLU devuelve cero si el valor de entrada es negativo, y devuelve el propio valor de entrada si este es positivo (Gómez, González y Harewood, 2019) (véase la figura 26).

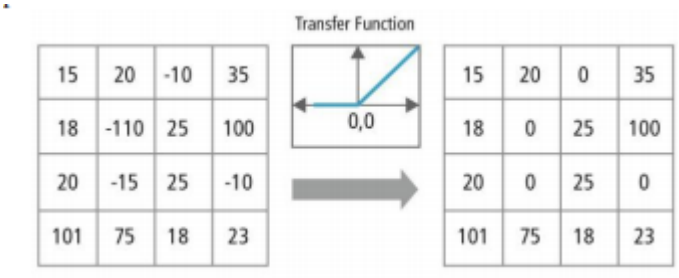


Figura 26: Función de activación ReLU. Fuente: Gómez, González y Harewood (2019).

Las últimas capas se forman de la siguiente manera: en primer lugar, utilizamos una capa de flatten para que nuestra imagen, que es profunda, "aplanarla" en una dimensión. Se añade otra capa con 256 neuronas con función de activación ReLU que esté fully connected (todas las neuronas conectadas a todas las neuronas) con la capa anterior. A esta última capa le añadimos un Dropout para "matar" aleatoriamente neuronas al azar y así

evitar que determinadas neuronas sobreaprendan. Finalmente, añadimos una última capa fully connected con la anterior y con el mismo número de neuronas que clases con activación solftmax, es decir, solo se activará la neurona (clase) más probable.

```
In [14]: # se diseña la red neuronal

In [15]: cnn=Sequential()
         cnn.add(Convolution2D(filtrosConvolucion1, tamano_kernel, padding='same',input_shape=(ancho,alto,3),activation='relu'))
         cnn.add(MaxPooling2D(pool_size=tamano_MaxPool))

         cnn.add(Convolution2D(filtrosConvolucion2,tamano_kernel,padding='same'))
         cnn.add(MaxPooling2D(pool_size=tamano_MaxPool))

         cnn.add(Convolution2D(filtrosConvolucion3,tamano_kernel,padding='same'))
         cnn.add(MaxPooling2D(pool_size=tamano_MaxPool))

In [16]: cnn.add(Flatten())
         cnn.add(Dense(256, activation='relu'))
         cnn.add(Dropout(0.1))
         cnn.add(Dense(clases, activation='softmax'))

In [17]:
         cnn.compile(loss='categorical_crossentropy',
                     metrics=['accuracy','Recall'])
```

Figura 27: Diseño del primer modelo. Fuente: elaboración propia.

Al final del desarrollo del modelo, se indica con compile los parámetros para optimizar (donde indicamos las métricas de Recall y accuracy) y finalmente se entrena la red neuronal. Una vez finalice el entrenamiento, se guarda tanto el modelo como los pesos de este (véase la figura 28).

```
In [*]: cnn.fit_generator(
            entrenamiento_generator,
            epochs=epocas,
            validation_data=validacion_generator,
            )

D:\Users\ferna\anaconda3\envs\tf\lib\site-packages\tensorflow\python\keras\engine\training.py:1969: UserWarning: `Model.fit_gen
erator` is deprecated and will be removed in a future version. Please use `Model.fit`, which supports generators.
  warnings.warn('`Model.fit_generator` is deprecated and '

Epoch 1/20
21/45 [=============>.............] - ETA: 1:29 - loss: 11.4875 - accuracy: 0.0872 - recall: 0.0427

In [46]: target_dir = './modelo/'
         if not os.path.exists(target_dir):
           os.mkdir(target_dir)
         cnn.save('./modelo/modelo.h5')
```

Figura 28: Entrenamiento del primer modelo. Fuente: elaboración propia.

Este primer modelo arrojaba valores en las métricas de entrenamiento muy buenas (superior al 95%), pero en las métricas de validación no llegaba al 50% (véase la figura 29), es decir, estamos en un caso de sobreajuste.

```
al_accuracy: 0.4315 - val_recall: 0.4315
Epoch 19/20
45/45 [==============================] - 151s 3s/step - loss: 0.1517 - accuracy: 0.9733 - recall: 0.9733 - val_loss: 6.9818 - v
al_accuracy: 0.4647 - val_recall: 0.4647
Epoch 20/20
45/45 [==============================] - 152s 3s/step - loss: 0.1109 - accuracy: 0.9806 - recall: 0.9806 - val_loss: 6.2250 - v
al_accuracy: 0.4896 - val_recall: 0.4855

<tensorflow.python.keras.callbacks.History at 0x27d0bb8a910>
```

Figura 29: Valores de las métricas en el entrenamiento del primer modelo. Fuente: elaboración propia.

Se pone en práctica este primer modelo desarrollando un pequeño archivo de Python que carga el modelo y sus pesos, poniéndolo en práctica con imágenes que antes no había visto el modelo.

El resultado de esta prueba es que el modelo fallaba la mayor parte de las veces, acertando solo aquellas imágenes en las que la marca se ve claramente.

Tras este primer intento, se modifican los hiperparámetros aleatoriamente buscando la solución óptima, pero esta metodología no ha sido muy acertada y el mejor accuracy que se ha obtenido es del 49%, es decir, acierta la mitad de las imágenes.

La conclusión del desarrollo de este primer modelo es que se necesita más información de lo que está pasando en él (añadir gráficas que permitan detectar problemas como overfitting, etc.), así como dar mayor complejidad a las imágenes, reducir la complejidad del modelo o probar otras soluciones.

4.3. SEGUNDA VERSIÓN DEL MODELO

De la primera solución que se ha desarrollado se determina la lección más importante: saber qué está pasando en nuestro modelo. Para ello, lo primero es identificar y entender las dos métricas básicas que arroja el entrenamiento: accuracy y loss.

Accuracy es el valor de la precisión en los datos de entrenamiento, es decir, cuántos datos ha acertado de todas las predicciones. Val_accuracy, por el contrario, mide la precisión en los datos de validación, por lo que es la mejor métrica de precisión del modelo.

Loss es el valor de la función de coste en los datos de entrenamiento. Mide la distancia entre el valor predicho y el real. Podría decirse que es cuánto le está costando aprender. Por el contrario, Val_loss mide lo mismo, pero en los datos de validación.

Una vez teniendo claro estas variables, es necesario entender dos de los errores más comunes cuando se realiza el desarrollo de un modelo: overfitting y underfiting (Jabbar & Khan, 2015).

El overfitting se produce cuando el modelo desarrollado aprende tanto en la fase de entrenamiento que aprende hasta del ruido, por lo que cualquier imagen que diferencie un poco de las entrenadas no la reconocerá y dará un valor más aleatorio.

Una forma de identificar el overfitting es enfrentar las gráficas de las variables Val_accuracy y accuracy. Si el accuracy del entrenamiento es mucho mayor que el de validación, estamos en un caso de overfitting.

El **underfiting** es justo lo contrario. El modelo no es capaz de aprender lo suficiente durante el entrenamiento y durante la fase de validación inclusive se pueden tener mejores resultados.

Un modelo óptimo es aquel que se encuentra en el punto medio del undefiting y el overfitting. Una forma más de identificar el overfitting y underfiting es graficar el error, buscando siempre el mínimo (véase la figura 30).

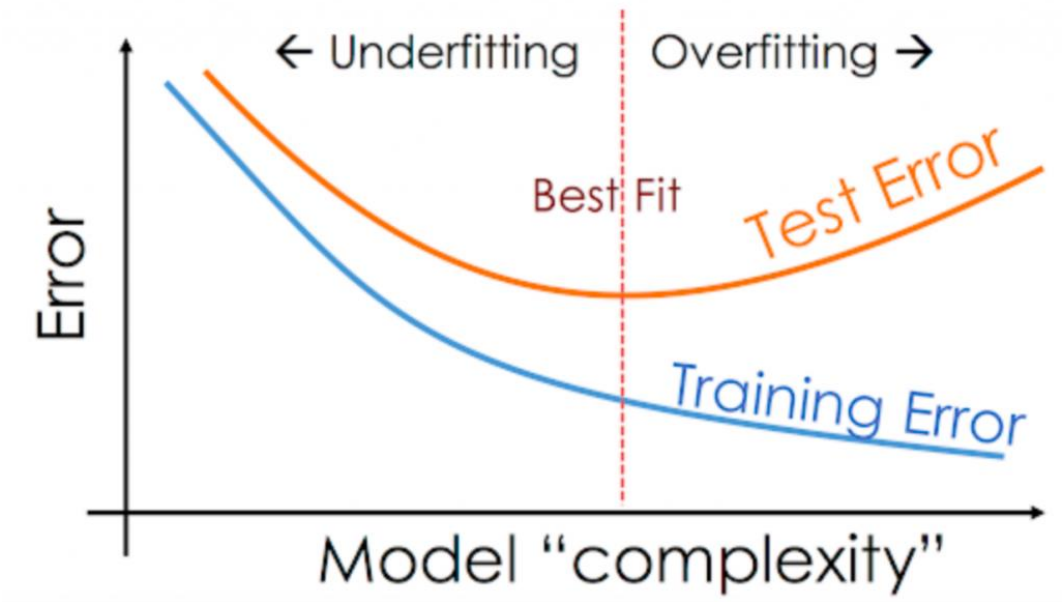


Figura 30: Underfiting y Overfitting en el error frente a la complejidad. Fuente: vitalflux.

Una vez que se tienen estos conceptos claros, se procede a la modificación del primer modelo. En primer lugar, se implementa la solución propuesta por Gómez, González y Harewood (2019) para resolver el famoso problema de clasificar perros y gatos (véase la figura 31).

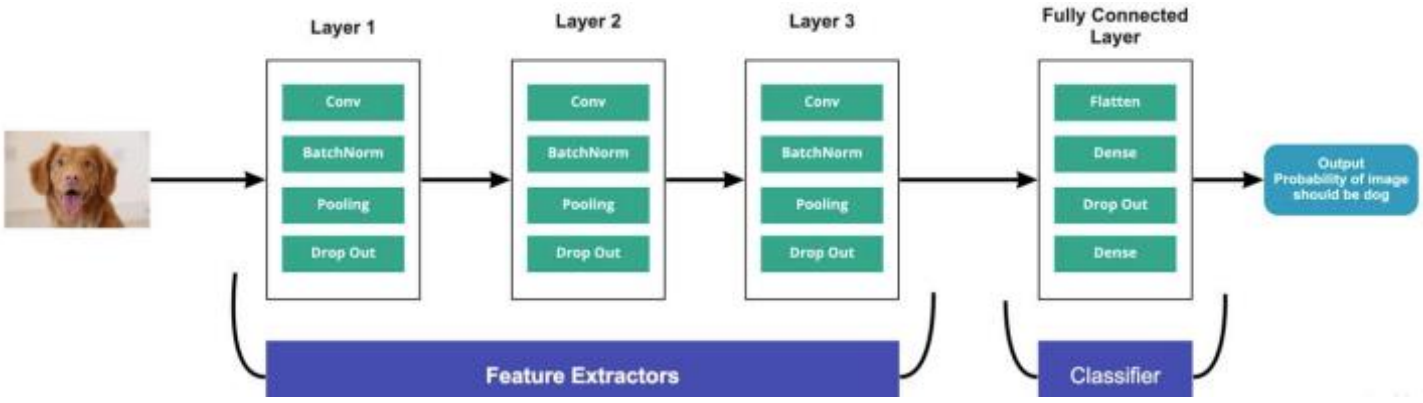


Figura 31:Modelo base de la segunda versión. Fuente: Gómez, González y Harewood (2019).

En este modelo se aplica un nuevo tipo de capa: BatchNormalization. Esta se utiliza para normalizar el valor de los resultados tras la convolución, es decir, que estos estén comprendidos entre 0 y 1.

En el nuevo modelo además se especifica el optimizador, que para esta solución se utilizará Adam, ya que es el más común para soluciones de clasificaciones categóricas.

Tras la ejecución por primera vez de este modelo, y sin llegar a su finalización, se observa que no consigue aprender (véase la figura 32) dado que sus valores de accuracy permanecen constantes.

```
Epoch 11/100
50/50 [==============================] - 19s 387ms/step - loss: 3.0742 - accuracy: 0.0648 - recall: 0.0113 - val_loss: 11.6783
- val_accuracy: 0.0664 - val_recall: 0.0581
Epoch 12/100
50/50 [==============================] - 20s 400ms/step - loss: 3.0388 - accuracy: 0.0689 - recall: 0.0186 - val_loss: 22.9942
- val_accuracy: 0.0622 - val_recall: 0.0622
Epoch 13/100
50/50 [==============================] - 20s 389ms/step - loss: 3.0858 - accuracy: 0.0527 - recall: 0.0073 - val_loss: 14.0328
- val_accuracy: 0.0498 - val_recall: 0.0456
Epoch 14/100
50/50 [==============================] - 20s 397ms/step - loss: 3.0974 - accuracy: 0.0454 - recall: 0.0024 - val_loss: 23.9482
- val_accuracy: 0.0705 - val_recall: 0.0498
Epoch 15/100
40/50 [========================>.....] - ETA: 3s - loss: 3.0961 - accuracy: 0.0450 - recall: 0.0060
```

Figura 32: Primera ejecución del segundo modelo. Fuente: elaboración propia.

Esto quiere decir que el modelo o no es lo suficiente complejo para la solución o se le ponen muchos “inconvenientes”. Tras realizar varias pruebas, se observa que el modelo aprende mejor si se eliminan las capas de BatchNormalization.

Además, con la finalidad de facilitar al modelo el aprendizaje, se eliminan también las capas de Dropout. También se diseña la primera gráfica para interpretar el modelo (véase la figura 33), en la cual se observa como ahora hay un caso de overfitting (en entrenamiento se llega casi al 100% pero en validación solo se llega al 50%).

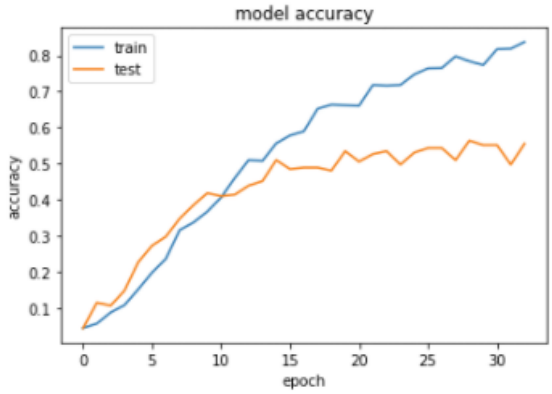


Figura 33: Accuracy del segundo modelo sin Dropout. Fuente: elaboración propia.

Con la finalidad de mejorar este resultado, se hicieron varias pruebas, añadiendo más capas convolucionales, eliminándolas, modificando la cantidad de neuronas del clasificador, modificando el número de filtros de las

capas convolucionales etc. Con todo ello, se obtuvo como mejor resultado una precisión del 56% (véase la figura 34).

```
In [*]: history=cnn.fit(
            entrenamiento_generator,
            epochs=epocas,
            validation_data=validacion_generator,
            )

Epoch 1/35
50/50 [==============================] - 20s 390ms/step - loss: 5.5300 - accuracy: 0.1434 - recall: 0.0194 - val_loss: 2.6796 -
val_accuracy: 0.2697 - val_recall: 0.0332
Epoch 2/35
50/50 [==============================] - 19s 387ms/step - loss: 2.1925 - accuracy: 0.3663 - recall: 0.1524 - val_loss: 2.1305 -
val_accuracy: 0.4523 - val_recall: 0.1618
Epoch 3/35
50/50 [==============================] - 19s 387ms/step - loss: 1.3273 - accuracy: 0.6305 - recall: 0.4206 - val_loss: 1.8301 -
val_accuracy: 0.4730 - val_recall: 0.2822
Epoch 4/35
50/50 [==============================] - 20s 393ms/step - loss: 0.7887 - accuracy: 0.7763 - recall: 0.6151 - val_loss: 1.7727 -
val_accuracy: 0.5270 - val_recall: 0.3402
Epoch 5/35
50/50 [==============================] - 21s 420ms/step - loss: 0.4174 - accuracy: 0.8784 - recall: 0.7893 - val_loss: 1.7466 -
val_accuracy: 0.5602 - val_recall: 0.3734
```

Figura 34: Mejor resultado obtenido modificando el modelo 2. Fuente: elaboración propia.

Para intentar superar este porcentaje, se han buscado soluciones adicionales que permitan mejorar el modelo sin sobreajustarlo, dando con la herramienta de Data Augmentation.

Data Augmentation consiste en modificar y aumentar el número de imágenes de las que se disponen en el set inicial de imágenes, algo de gran utilidad en este caso ya que el set está limitado.

Para ello, se ha realizado un generador de imágenes aleatorio (el cual genera 1 imagen por cada 10 imágenes del set) además de modificar las existentes, con la finalidad de aumentar el set de imágenes y dificultar el aprendizaje de las existentes. Esto favorece a la reducción del overfitting.

Concretamente, las imágenes pueden generarse o verse modificadas aleatoriamente de la siguiente forma:

- Rotación entre más/menos 20 grados.
- Zoom entre el 20%.
- Desplazamientos verticales y horizontales.
- Invertir la imagen horizontalmente.

Además, con el fin de mejorar la visualización se ha modificado la generación de las gráficas, de forma que se pueda ver con mejor detalle lo que está sucediendo.

Utilizando el Data Augmentation en el último modelo realizado (véase la figura 35), en el cual había overfitting, se observa como con las nuevas imágenes y el ruido añadido se ha conseguido pasar del overfitting al undefiting (véase la figura 36).

```
In [72]: cnn.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 125, 125, 24)      672
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 62, 62, 24)        0
_________________________________________________________________
dropout (Dropout)            (None, 62, 62, 24)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 62, 62, 40)        8680
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 31, 31, 40)        0
_________________________________________________________________
flatten (Flatten)            (None, 38440)             0
_________________________________________________________________
dense (Dense)                (None, 256)               9840896
_________________________________________________________________
dropout_1 (Dropout)          (None, 256)               0
_________________________________________________________________
dense_1 (Dense)              (None, 80)                20560
_________________________________________________________________
dropout_2 (Dropout)          (None, 80)                0
_________________________________________________________________
dense_2 (Dense)              (None, 22)                1782
=================================================================
Total params: 9,872,590
Trainable params: 9,872,590
Non-trainable params: 0
```

Figura 35: Modelo de prueba. Fuente: elaboración propia.

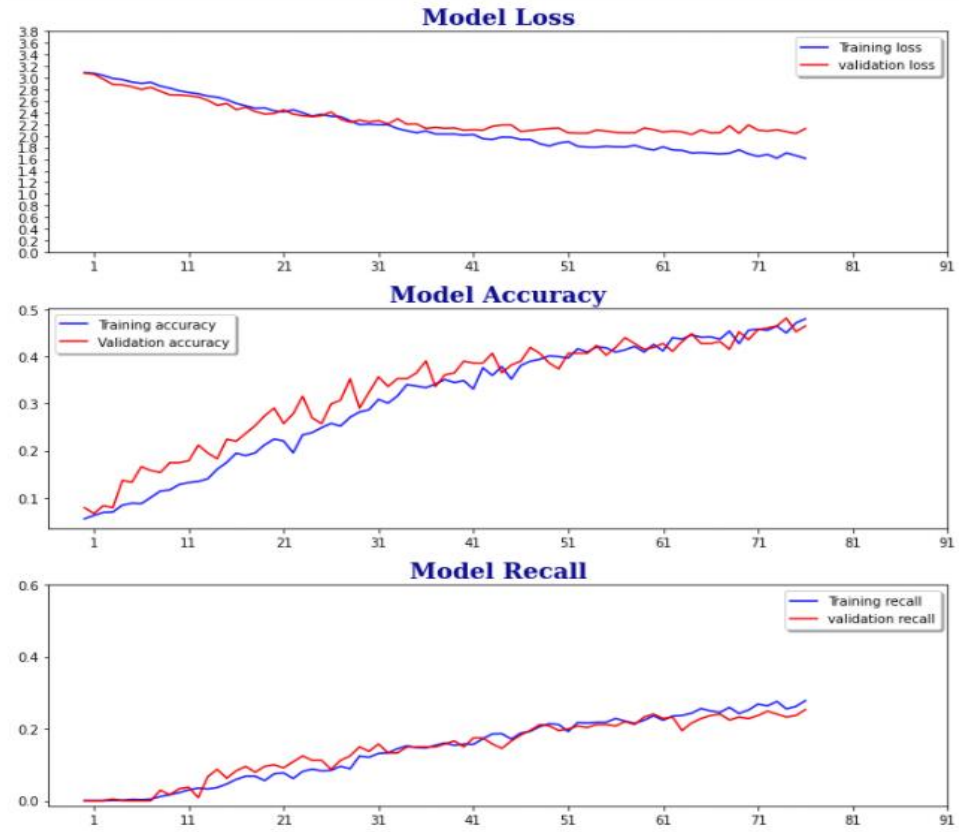


Figura 36: Primeras métricas con Data Augmentation. Fuente: elaboración propia.

Tras varias pruebas, se vuelve al modelo original desarrollado por Gómez, González y Harewood (2019), eliminando las capas de Dropout (salvo la de clasificador) y BatchNormalization. Ajustando los parámetros y realizando varias pruebas de entrenamiento, el mejor modelo obtenido (véase la figura 37) alcanza casi el 70% de precisión en los datos de validación.

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 125, 125, 24)      672
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 62, 62, 24)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 62, 62, 60)        13020
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 31, 31, 60)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 31, 31, 32)        17312
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 15, 15, 32)        0
_________________________________________________________________
flatten (Flatten)            (None, 7200)              0
_________________________________________________________________
dense (Dense)                (None, 64)                460864
_________________________________________________________________
dropout (Dropout)            (None, 64)                0
_________________________________________________________________
dense_1 (Dense)              (None, 22)                1430
=================================================================
Total params: 493,298
Trainable params: 493,298
Non-trainable params: 0
_________________________________________________________________
```

Figura 37: Mejor versión del modelo 2. Fuente: elaboración propia.

El resultado obtenido es el alcance de un punto muerto que se puede mejorar mucho más siguiendo esta vía. La razón es que el modelo se encuentra en la parte de underfiting (véase la figura 38), es decir, no es capaz de aprender lo suficiente, por lo que para mejorarlo hay que aumentar la complejidad del modelo, lo que a su vez aumenta el número de parámetros. Esto da lugar a que el modelo tenga que reajustar un mayor número de parámetros durante el entrenamiento, pero el data set es limitado y no se tienen las suficientes imágenes para que el modelo aprenda.

Es por esta razón por la que se da por finalizada esta posible solución y se busca una tercera, con la finalidad de mejorar el porcentaje de accuracy y, por consiguiente, mejorar las predicciones del modelo.

Figura 38: Métricas de la última versión del modelo 2. Fuente: elaboración propia.

4.4. VERSIÓN FINAL DEL MODELO

Como se ha visto en el apartado anterior, se llega a una situación en la que entrenar el modelo desde cero se convierte en una solución poco óptima cuando el set de imágenes es limitado; es por ello por lo que se ha buscado una tercera y última solución.

Esta, la solución final, se realizará a través de transfer leraning. El transfer learning consiste en utilizar el conocimiento almacenado en redes que han sido ya entrenadas con millones de imágenes y tienen la capacidad de identificar contornos, formas etc.

Aunque la mayoría de los algoritmos estén diseñados para una tarea específica, cada vez hay más interés en el desarrollo de estos algoritmos genéricos para la comunidad (Torrey y Shavlik 2010).

Hay muchas redes que están entrenadas e incluso están en la librería de keras; algunas de ellas se han visto en el apartado de CNN más populares. Para la solución del problema de este TFM se ha elegido VGG16, ya que no es una red muy compleja y no requiere de excesiva capacidad de cómputo.

Lo primero que se ha realizado es la adaptación de los hiperparámetros para poder adaptar nuestras imágenes al input de VGG16. Para ello se ha cambiado el tamaño al que se transforman las imágenes a 224x224 con tres canales, cumpliendo así el input de VGG16 (véase la tabla 5).

Capa	Tamaño
INPUT	224x224x3
CONV3-64	224x224x64
CONV3-64	224x224x64
POOL2	112x112x64
CONV3-128	112x112x128
CONV3-128	112x112x128
POOL2	56x56x128
CONV3-256	56x56x256
CONV3-256	56x56x256
CONV3-256	56x56x256
POOL2	28x28x256
CONV3-512	28x28x512
CONV3-512	28x28x512
CONV3-512	28x28x512
POOL2	14x14x512
CONV3-512	14x14x512
CONV3-512	14x14x512
CONV3-512	14x14x512
POOL2	7x7x512
FC	1x1x4096
FC	1x1x4096
FC	1x1x1000

Tabla 5: Capas y características VGG16. Fuente: Delbracio, Lezama y Carbajal, 2017.

A continuación, se carga el modelo. Como se ha mencionado, lo que se quiere es obtener un modelo ya entrenado que identifique formas, por lo que también es necesario cargar los pesos del modelo que ha sido entrenado.

También se ha modificado la salida, ya que VGG16 clasifica entre 1000 clases, pero esta solución solo clasifica entre las 22 marcas posibles. Por ello, la primera solución que se ha aportado ha sido eliminar la última capa de VGG16 y sustituirla por una con 22 salidas con activación solftmax.

El siguiente paso ha sido indicar el número de capas que se congelan en el modelo. En un primer intento se ha congelado solo la última capa, que es la de clasificación.

Cuando se hace referencia a congelar quiere decir que las capas congeladas no se entrenan y mantienen sus pesos, siendo el resto de las capas las que cambian sus parámetros para realizar la clasificación.

Además, se ha añadido más regularización, concretamente se ha añadido paciencia y checkpoint.

Al implementar la paciencia se monitoriza una o varias variables; si estas al cabo de x interaciones especificadas no mejoran, el modelo se para antes de finalizar todos los Epochs.

El checkpoint se utiliza para guardar el mejor modelo o sus pesos. Para ello, se le ha indicado monitorizar una variable (en este caso el accuracy) para que cuando alcance o supere su valor máximo guarde el modelo.

Layer (type)	Output Shape	Param #
input_11 (InputLayer)	[(None, 224, 224, 3)]	0
block1_conv1 (Conv2D)	(None, 224, 224, 64)	1792
block1_conv2 (Conv2D)	(None, 224, 224, 64)	36928
block1_pool (MaxPooling2D)	(None, 112, 112, 64)	0
block2_conv1 (Conv2D)	(None, 112, 112, 128)	73856
block2_conv2 (Conv2D)	(None, 112, 112, 128)	147584
block2_pool (MaxPooling2D)	(None, 56, 56, 128)	0
block3_conv1 (Conv2D)	(None, 56, 56, 256)	295168
block3_conv2 (Conv2D)	(None, 56, 56, 256)	590080
block3_conv3 (Conv2D)	(None, 56, 56, 256)	590080
block3_pool (MaxPooling2D)	(None, 28, 28, 256)	0
block4_conv1 (Conv2D)	(None, 28, 28, 512)	1180160
block4_conv2 (Conv2D)	(None, 28, 28, 512)	2359808
block4_conv3 (Conv2D)	(None, 28, 28, 512)	2359808
block4_pool (MaxPooling2D)	(None, 14, 14, 512)	0
block5_conv1 (Conv2D)	(None, 14, 14, 512)	2359808
block5_conv2 (Conv2D)	(None, 14, 14, 512)	2359808
block5_conv3 (Conv2D)	(None, 14, 14, 512)	2359808
block5_pool (MaxPooling2D)	(None, 7, 7, 512)	0
flatten (Flatten)	(None, 25088)	0
fc1 (Dense)	(None, 4096)	102764544
fc2 (Dense)	(None, 4096)	16781312
output (Dense)	(None, 22)	90134

Total params: 134,350,678
Trainable params: 90,134
Non-trainable params: 134,260,544

Figura 39: Arquitectura del primer modelo 3. Fuente: elaboración propia.

En este primer entrenamiento, sin optimizar los hiperparámetros y sin descongelar más capas alcanza un accuracy de hasta el 65% (véase la figura 40), que es casi la mejor versión del segundo modelo que se había realizado.

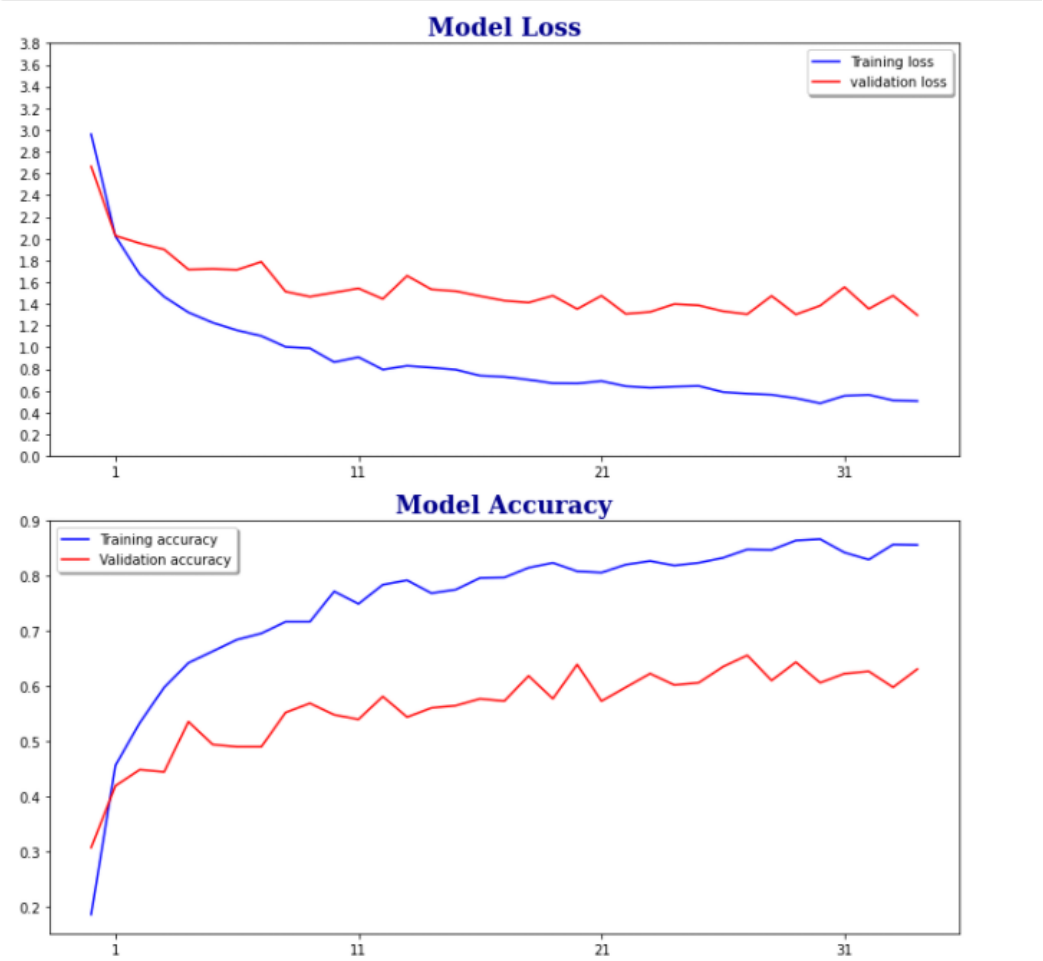


Figura 40: Métricas primer entrenamiento modelo 3. Fuente: elaboración propia.

Como se observa analizando la gráfica, el modelo empieza a hacer overfitting y no se consiguen unos mejores resultados. Es por ello por lo que se realizan más modificaciones en el modelo.

En primer lugar, se carga de nuevo el modelo VGG16, pero esta vez no se cargan las capas del clasificador, sino que se crean unas capas nuevas. Concretamente, se creará un GlobalAveragePooling, seguido de una fully connected que irá conectada finalmente a la capa de salida (véase la figura 41).

```
Model: "model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_3 (InputLayer)         [(None, 224, 224, 3)]     0
_________________________________________________________________
block1_conv1 (Conv2D)        (None, 224, 224, 64)      1792
_________________________________________________________________
block1_conv2 (Conv2D)        (None, 224, 224, 64)      36928
_________________________________________________________________
block1_pool (MaxPooling2D)   (None, 112, 112, 64)      0
_________________________________________________________________
block2_conv1 (Conv2D)        (None, 112, 112, 128)     73856
_________________________________________________________________
block2_conv2 (Conv2D)        (None, 112, 112, 128)     147584
_________________________________________________________________
block2_pool (MaxPooling2D)   (None, 56, 56, 128)       0
_________________________________________________________________
block3_conv1 (Conv2D)        (None, 56, 56, 256)       295168
_________________________________________________________________
block3_conv2 (Conv2D)        (None, 56, 56, 256)       590080
_________________________________________________________________
block3_conv3 (Conv2D)        (None, 56, 56, 256)       590080
_________________________________________________________________
block3_pool (MaxPooling2D)   (None, 28, 28, 256)       0
_________________________________________________________________
block4_conv1 (Conv2D)        (None, 28, 28, 512)       1180160
_________________________________________________________________
block4_conv2 (Conv2D)        (None, 28, 28, 512)       2359808
_________________________________________________________________
block4_conv3 (Conv2D)        (None, 28, 28, 512)       2359808
_________________________________________________________________
block4_pool (MaxPooling2D)   (None, 14, 14, 512)       0
_________________________________________________________________
block5_conv1 (Conv2D)        (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_conv2 (Conv2D)        (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_conv3 (Conv2D)        (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_pool (MaxPooling2D)   (None, 7, 7, 512)         0
_________________________________________________________________
marcas_pooling (GlobalAverag (None, 512)               0
_________________________________________________________________
marcasFull (Dense)           (None, 1024)              525312
_________________________________________________________________
output (Dense)               (None, 22)                22550
=================================================================
Total params: 15,262,550
Trainable params: 15,262,550
Non-trainable params: 0
```

Figura 41: Arquitectura de la segunda versión del modelo 3. Fuente: elaboración propia.

El entrenamiento de este modelo sigue mostrando un problema de overfitting, pero por primera vez se supera el 70% de precisión llegando a picos con un 75% de accuracy (véase la figura 42).

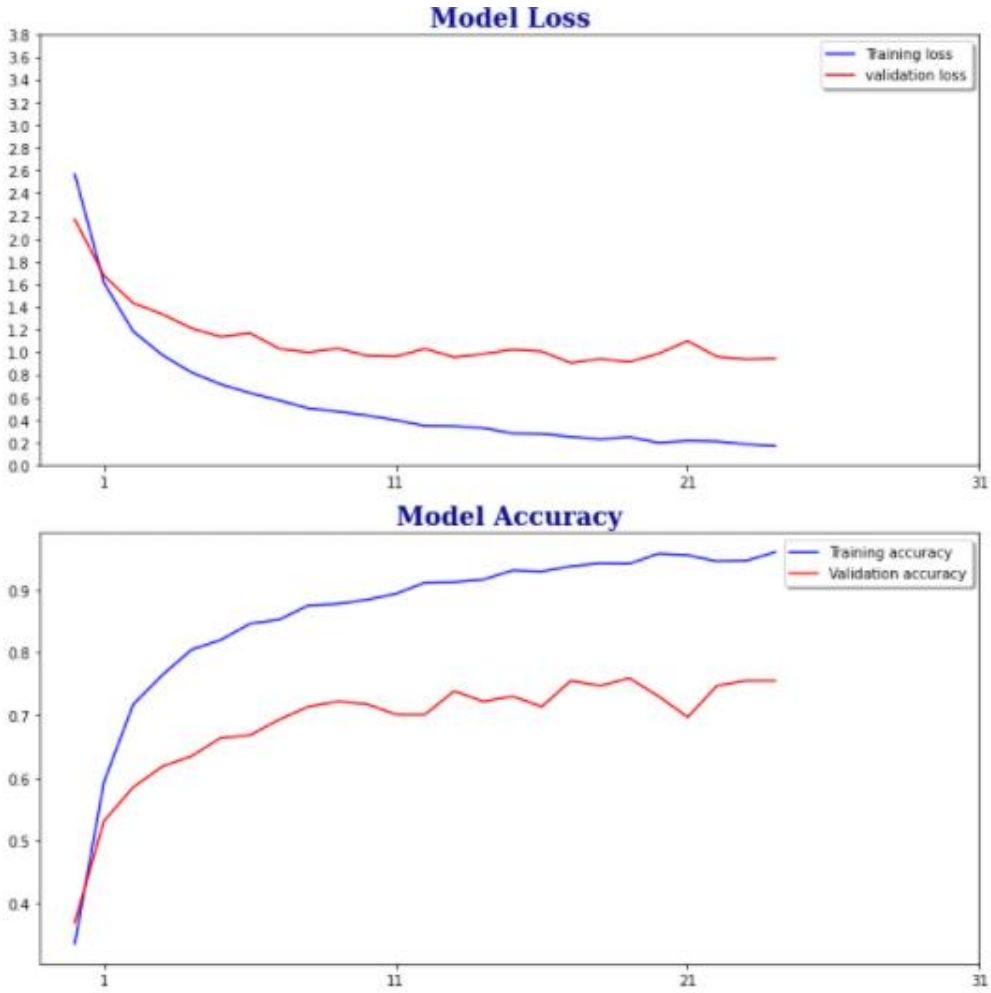


Figura 42: Métricas primer entrenamiento modelo 3. Fuente: elaboración propia.

Con este porcentaje de accuracy en la validación se considera que es un buen modelo para hacer predicciones. Para comprobarlo, se desarrollan las métricas y herramientas que nos indiquen la fiabilidad del modelo. Estas métricas son probadas con el grupo de imágenes reservado, que no ha visto aún el propio modelo.

Para entender estas medidas primero hay que entender la matriz de confusión, la cual simplemente mide los aciertos tanto si ha acertado una clase como si ha descartado las que no son, como los fallos (véase la figura 43).

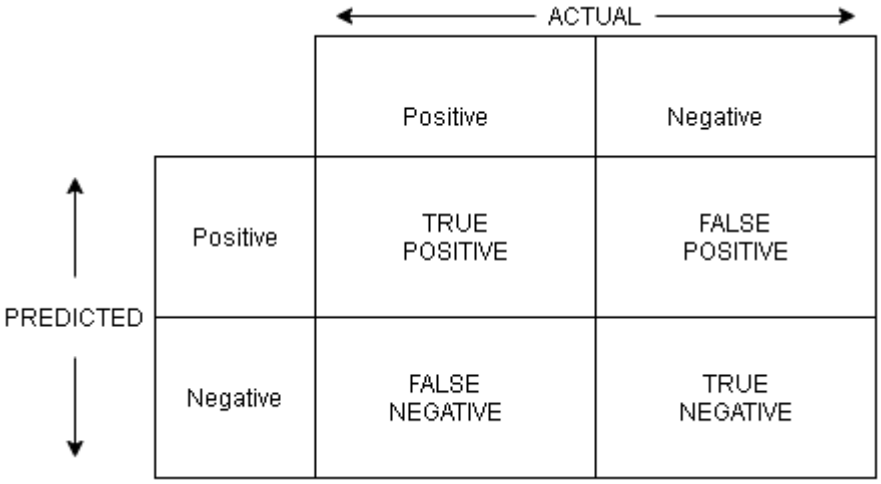


Figura 43: Matriz de confusión. Fuente: Towards Data Science.

La precisión mide la calidad del modelo, e indica el número de predicciones positivas correctas sobre todas las predicciones positivas (véase la figura 44).

$$Precision_C = \frac{True\ PositiveC}{True\ PositiveC + False\ Positive_C}$$

Figura 44: Ecuación de la precisión. Fuente: UE.

Por otro lado, se tiene el Recall que mide los casos positivos sobre todos los que realmente sí eran positivos (véase la figura 45).

$$Recall_C = \frac{True\ PositiveC}{True\ PositiveC + False\ Negative_C}$$

Figura 45: Ecuación del Recall: Fuente: UE.

El F1 Score es un balance entre las dos métricas anteriores que favorece siempre a la métrica con peor valor (véase la figura 46).

$$Macro\ FScore = 2 \times \frac{Precision_W \times Recall_W}{Precision_W + Recall_W}$$

Figura 46: Ecuación del FScore: Fuente: Fuente: UE.

Una vez se tienen claros estos conceptos, se desarrolla un algoritmo que utilice el modelo ya entrenado para realizar predicciones sobre el set de prueba.

Habiéndose hecho las predicciones de este set, se realiza el mapeo de la matriz de confusión (véase la figura 47).

```
In [17]: matc=confusion_matrix(y_real, y_pred)

         plot_confusion_matrix(conf_mat=matc, figsize=(9,9), class_names = names, show_normed=False)
         plt.tight_layout()
```

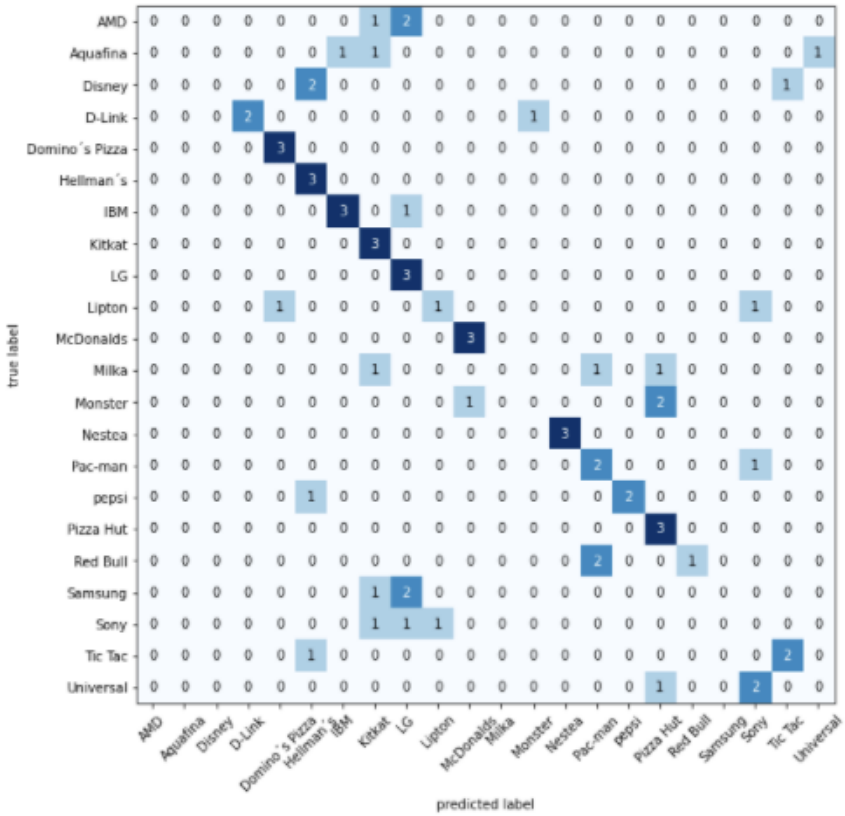


Figura 47: Matriz de confusión de la segunda versión del modelo 3. Fuente: elaboración propia.

En la matriz de confusión interesa que la diagonal principal haya el mayor número posible, ya que es cuando coinciden la predicción con la realidad. En este caso, se tienen bastantes datos centrados, pero también bastantes dispersos.

Para ver con mayor claridad el resultado, se calculan los estadísticos mencionados anteriormente (véase la figura 48).

	precision	recall	f1-score	support
AMD	0.0000	0.0000	0.0000	3
Aquafina	0.0000	0.0000	0.0000	3
Disney	0.0000	0.0000	0.0000	3
D-Link	1.0000	0.6667	0.8000	3
Domino´s Pizza	0.7500	1.0000	0.8571	3
Hellman´s	0.4286	1.0000	0.6000	3
IBM	0.7500	0.7500	0.7500	4
Kitkat	0.3750	1.0000	0.5455	3
LG	0.3333	1.0000	0.5000	3
Lipton	0.5000	0.3333	0.4000	3
McDonalds	0.7500	1.0000	0.8571	3
Milka	0.0000	0.0000	0.0000	3
Monster	0.0000	0.0000	0.0000	3
Nestea	1.0000	1.0000	1.0000	3
Pac-man	0.4000	0.6667	0.5000	3
pepsi	1.0000	0.6667	0.8000	3
Pizza Hut	0.4286	1.0000	0.6000	3
Red Bull	1.0000	0.3333	0.5000	3
Samsung	0.0000	0.0000	0.0000	3
Sony	0.0000	0.0000	0.0000	3
Tic Tac	0.6667	0.6667	0.6667	3
Universal	0.0000	0.0000	0.0000	3
accuracy			0.5075	67
macro avg	0.4265	0.5038	0.4262	67
weighted avg	0.4313	0.5075	0.4310	67

Figura 48: Estadísticos de la segunda versión del modelo 3. Fuente: elaboración propia.

El resultado, como se puede observar, no es del todo satisfactorio, ya que donde teníamos un accuracy del 75% en validación, en las pruebas llega al 50%, siendo el F1 más bajo aún.

Es por esto por lo que se sigue intentando perfeccionar el modelo, llegando finalmente a la versión final de este.

En primer lugar, se elimina la penúltima capa fully connected. Esto se realiza para hacer el modelo más simple y reducir su complejidad, con el objetivo de evitar el overfitting.

Finalmente, se pone en práctica la última técnica aprendida: fine tuning. Esta consiste en descongelar algunas de las últimas capas de la red que está entrenada para adaptarlas a nuestro modelo.

```
In [19]: custom_vgg_model.summary()
Model: "model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 224, 224, 3)]     0
_________________________________________________________________
block1_conv1 (Conv2D)        (None, 224, 224, 64)      1792
_________________________________________________________________
block1_conv2 (Conv2D)        (None, 224, 224, 64)      36928
_________________________________________________________________
block1_pool (MaxPooling2D)   (None, 112, 112, 64)      0
_________________________________________________________________
block2_conv1 (Conv2D)        (None, 112, 112, 128)     73856
_________________________________________________________________
block2_conv2 (Conv2D)        (None, 112, 112, 128)     147584
_________________________________________________________________
block2_pool (MaxPooling2D)   (None, 56, 56, 128)       0
_________________________________________________________________
block3_conv1 (Conv2D)        (None, 56, 56, 256)       295168
_________________________________________________________________
block3_conv2 (Conv2D)        (None, 56, 56, 256)       590080
_________________________________________________________________
block3_conv3 (Conv2D)        (None, 56, 56, 256)       590080
_________________________________________________________________
block3_pool (MaxPooling2D)   (None, 28, 28, 256)       0
_________________________________________________________________
block4_conv1 (Conv2D)        (None, 28, 28, 512)       1180160
_________________________________________________________________
block4_conv2 (Conv2D)        (None, 28, 28, 512)       2359808
_________________________________________________________________
block4_conv3 (Conv2D)        (None, 28, 28, 512)       2359808
_________________________________________________________________
block4_pool (MaxPooling2D)   (None, 14, 14, 512)       0
_________________________________________________________________
block5_conv1 (Conv2D)        (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_conv2 (Conv2D)        (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_conv3 (Conv2D)        (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_pool (MaxPooling2D)   (None, 7, 7, 512)         0
_________________________________________________________________
marcas_pooling (GlobalAverag (None, 512)               0
_________________________________________________________________
output (Dense)               (None, 22)                11286
=================================================================
Total params: 14,725,974
Trainable params: 4,730,902
Non-trainable params: 9,995,072
_________________________________________________________________
```

Figura 49: Arquitectura modelo final. Fuente: elaboración propia.

El resultado en la fase de validación es bastante bueno, alcanzado su máximo en 85% (gran mejora con respecto la mejor versión del primer modelo que solo alcanzaba el 48%).

El siguiente paso es validar estos resultados con las imágenes de pruebas. Se realizan las predicciones y, a continuación, realizamos la matriz de confusión (véase la figura 50).

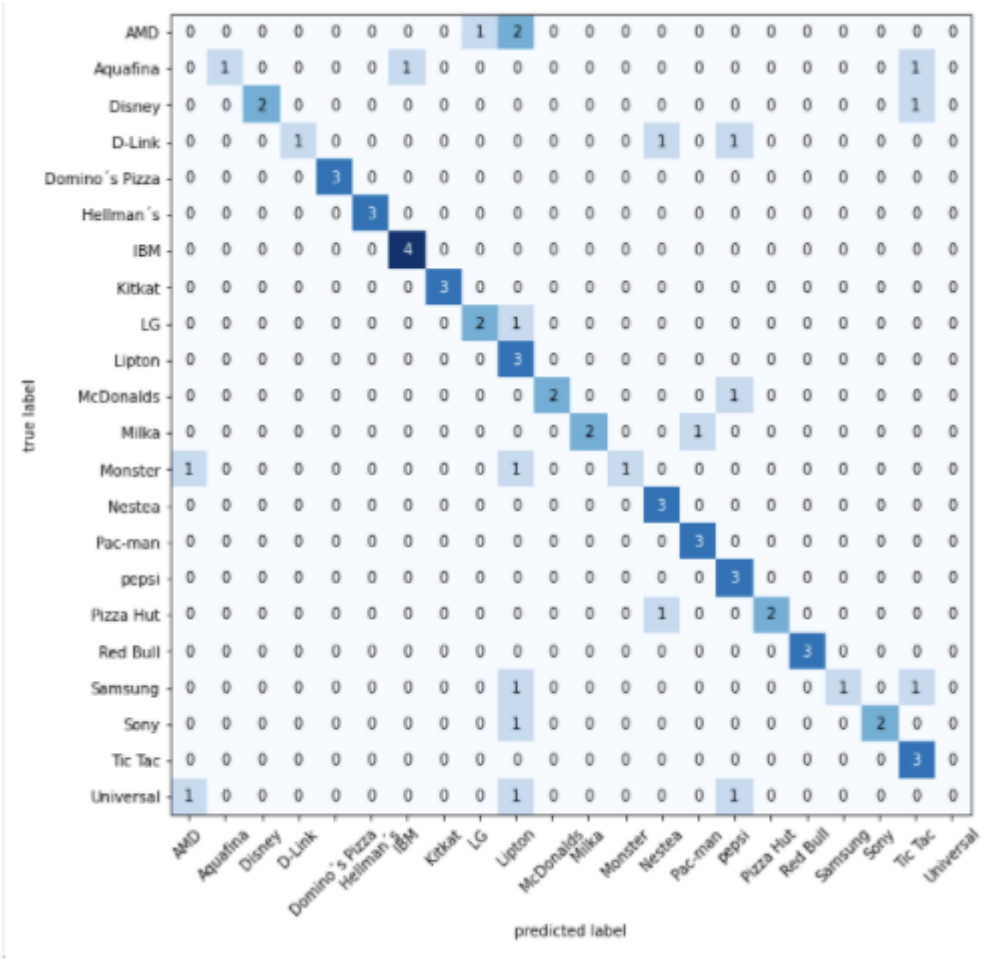


Figura 50: Matriz de confusión modelo final. Fuente: elaboración propia.

Como se observa, la matriz consigue muchos más valores que en la diagonal principal de la versión anterior. Se procede a obtener las métricas de las pruebas (véase la figura 51).

El resultado empeora hasta bajar al 70% de accuracy, pero se consigue un modelo con una precisión bastante buena, sobre todo comparada con los valores obtenidos en las primeras versiones.

	precision	recall	f1-score	support
AMD	0.0000	0.0000	0.0000	3
Aquafina	1.0000	0.3333	0.5000	3
Disney	1.0000	0.6667	0.8000	3
D-Link	1.0000	0.3333	0.5000	3
Domino´s Pizza	1.0000	1.0000	1.0000	3
Hellman´s	1.0000	1.0000	1.0000	3
IBM	0.8000	1.0000	0.8889	4
Kitkat	1.0000	1.0000	1.0000	3
LG	0.6667	0.6667	0.6667	3
Lipton	0.3000	1.0000	0.4615	3
McDonalds	1.0000	0.6667	0.8000	3
Milka	1.0000	0.6667	0.8000	3
Monster	1.0000	0.3333	0.5000	3
Nestea	0.6000	1.0000	0.7500	3
Pac-man	0.7500	1.0000	0.8571	3
pepsi	0.5000	1.0000	0.6667	3
Pizza Hut	1.0000	0.6667	0.8000	3
Red Bull	1.0000	1.0000	1.0000	3
Samsung	1.0000	0.3333	0.5000	3
Sony	1.0000	0.6667	0.8000	3
Tic Tac	0.5000	1.0000	0.6667	3
Universal	0.0000	0.0000	0.0000	3
accuracy			0.7015	67
macro avg	0.7780	0.6970	0.6799	67
weighted avg	0.7784	0.7015	0.6830	67

Figura 51: estadísticos modelo final. Fuente: elaboración propia.

5. CONCLUSIONES

De los resultados obtenidos en las pruebas y desarrollo del presente TFM, y atendiendo los objetivos presentados en este, se pueden extraer las siguientes conclusiones:

Se ha desarrollado un modelo de Deep Learning capaz de predecir qué marca aparece en una determinada imagen. Concretamente, se ha alcanzado el objetivo de que el accuracy sea igual o superior al 70%.

Se ha realizado un estudio para obtener una visión global del Machine Learning, analizando la taxonomía de este y obteniendo una visión global de esta tecnología.

Se han seleccionado las CNN como el tipo de Machine Learning empleado para esta solución y se ha estudiado sobre ellas, entendiendo su funcionamiento, sus hiperparámetros, así como soluciones específicas como la regularización o el fin tuning.

Se puede concluir entonces que en el presente TFM se han cumplido todos los objetivos marcados inicialmente y que se ha obtenido un aprendizaje y conocimiento muy amplio sobre el Deep Learning, concretamente, de las redes neuronales convolucionales.

6. LÍNEAS FUTURAS DE TRABAJO

En este apartado se van a analizar las posibles líneas de futuro que tiene el presente TFM:

- **Perfeccionar el modelo.** El modelo ha llegado a una precisión del 70%, por lo que sería interesante seguir trabajando en el modelo y aumentar esta precisión.
- Con una mayor cantidad de imágenes, implementar **object detection.** Sería interesante este algoritmo para determinar, por ejemplo, qué marcas aparecen en una foto.
- Realizar una comparación con otros modelos semejantes para detectar errores o posibles mejoras.
- Utilizar esta red neuronal desarrollada manteniendo el valor de los pesos para detectar marcas que no ha visto anteriormente. Concretamente, se puede estudiar como reacciona con nuevas imágenes e ir descongelando capas, analizando si es un buen modelo para detectar los logotipos de las marcas independientemente si las ha visto anteriormente o no.

7. REFERENCIAS BIBLIOGRÁFICAS

Alake, R., 2014. Aprendizaje profundo: explicación de GoogLeNet. ICHI.

Artola, Á. (2019). Clasificación de imágenes usando redes neuronales convolucionales en Python. (Trabajo Fin de Grado Inédito). Universidad de Sevilla, Sevilla.

Berzal, F. (2015). Clasificación y predicción. *Obtenido de DECSAI*.

Calvo, D. (2017). Red Neuronal Convolucional CNN. *Diego Calvo*.

Challenger-Pérez, I., Díaz-Ricardo, Y., y Becerra-García, R. A. (2014). El lenguaje de programación Python. *Ciencias Holguín*, *20*(2), 1-13.

Delbracio, M., Lezama, J. y Carbajal, G. (2017). Aprendizaje profundo para visión artificial. *Instituto de Ingeniería Eléctrica Facultad de Ingeniería Universidad de la República.*

Ertam, F., & Aydın, G. (2017). *Data classification with deep learning using Tensorflow*. International conference on computer science and engineering (UBMK) IEEE.

Goldsborough, P. (2016). A tour of tensorflow. *arXiv.*

Gómez, C., González, J. y Harewood, K. (2019). Red Neuronal Convolucional: Modelo Clasificador de Perros y Gatos. Universidad Latina de Panamá, Panamá.

González, A. (2020). Conceptos Básicos de Machine Learning. *Cleverdata*. https://cleverdata.io/conceptos-basicos-machine-learning/

He, K., Zhang, X., Ren, S., & Sun, J. (2015). Spatial pyramid pooling in deep convolutional networks for visual recognition. *IEEE transactions on pattern analysis and machine intelligence*, *37*(9), 1904-1916.

Jabbar, H., & Khan, R. Z. (2015). Methods to avoid over-fitting and under-fitting in supervised machine learning (comparative study). *Computer Science, Communication and Instrumentation Devices*, 163-172.

Ketkar, N. (2017). Introduction to keras. En F. Chollet. *Deep learning with Python* (pp. 97-111). Manning Publications

LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. *Nature*, *521*(7553), 436-444.

LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, *86*(11), 2278 – 2324.

López, R. F., y Fernández, J. M. F. (2008). *Las redes neuronales artificiales*. Netbiblo.

Mahdavinejad, M. S., Rezvan, M., Barekatain, M., Adibi, P., Barnaghi, P., & Sheth, A. P. (2018). Machine learning for Internet of Things data analysis: A survey. *Digital Communications and Networks*, *4*(3), 161-175.

Matich, D. J. (2001). Redes Neuronales: Conceptos básicos y aplicaciones. *Universidad Tecnológica Nacional, México*.

Moreno, A., Armengol, E., Béjar, J., Belanche, L., Cortés, U., Gavaldà, R., Gimeno J.M., López, B., Martín, M. y Sánchez, M. (1994). *Aprendizaje automático*. Edicions UPC

Ren, S., He, K., Girshick, R., Sun, J., Donahue, J., Darrell, T., Malik, J., Liu, W., Anguelov, D., Erhan, D., Szegedy, C., Reed, S., Impiombato, D., Giarrusso, S., Mineo, T., Catalano, O., Gargano, C., La Rosa, G.,... Anguelov, D. (2015). Rich feature hierarchies for accurate object detection and semantic segmentation. *Proc. IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit., 749*, 1-15.

Shukla, N., & Fricklas, K. (2018). *Machine learning with TensorFlow*. Manning Publications.

Simonyan, K., & Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. *arXiv.*

Targ, S., Almeida, D., & Lyman, K. (2016). Resnet in resnet: Generalizing residual architectures. *arXiv.*

Turck, M. (30 de septiembre de 2020). Resilience and Vibrancy: The 2020 Data & AI Landscape. *FirstMark.* https://mattturck.com/data2020/

Zeng, Z., Gong, Q., & Zhang, J. (2019). *CNN model design of gesture recognition based on tensorflow framework.* En IEEE 3rd Information Technology, Networking, Electronic and Automation Control Conference (ITNEC).

ANEXOS

ANEXO I: TIEMPO DE DESARROLLO

En este anexo se muestra el tiempo empleado para el desarrollo del presente TFM.

Como se ve en la tabla del anexo, en las primeras semanas se dedica poco tiempo, debido a que aún se tenían trabajos y clases del máster. La semana de los exámenes no se dedica tiempo al TFM y las primeras de agosto tampoco se dedica tiempo y se emplean como vacaciones.

En agosto y principios de septiembre, se dedica dos horas diarias entre semana y entre 8 y 10 horas por fin de semana.

Finalmente, las últimas semanas de septiembre se consigue dedicar más tiempo para realizar las correcciones del tutor y el desarrollo final de la solución (véase tabla del anexo).

Semana	Horas empleadas	Comentarios
Julio 1 - 4	4	Investigación y lectura de bibliografía
Julio 5 - 11	4	Investigación y lectura de bibliografía
Julio 12 - 18	4	Investigación y lectura de bibliografía
Julio 19 - 25	0	Exámenes
Julio 26 - agosto 1	0	Vacaciones
Agosto 2 - 8	0	Vacaciones
Agosto 9 - 15	10	Investigación sobre redes convolucionales
Agosto 16 - 22	18	Aprender conceptos básicos de Python
Agosto 23 - 29	18	Primeros programas de Python
Agosto 30 - septiembre 5	18	Primeras redes neuronales
Septiembre 6 - 12	18	Separación del set de imágenes y primera solución
Septiembre 13 - 19	30	Segunda solución
Septiembre 20 - 26	30	Solución final
Total	154	

TABLA ANEXO I: Tiempos de ejecución

ANEXO II: CÓDIGO

TFM_MODELO_V1

In [ ]:
```
# Importamos todas las librerarías que vamos a necesitar para ejecutar nuestro modelo
```

In [1]:
```
import sys
import os
import tensorflow as tf
from tensorflow.python.keras.preprocessing.image import ImageDataGenerator
from tensorflow.python.keras import optimizers
from tensorflow.python.keras.models import Sequential
from tensorflow.python.keras.layers import Dropout, Flatten, Dense, Activation
from tensorflow.python.keras.layers import Convolution2D, MaxPooling2D
from tensorflow.python.keras.layers import BatchNormalization
from tensorflow.python.keras import backend as K
import matplotlib.pyplot as plt
```

In [2]:
```
#Cerramos todos los procesos abiertos por Keras
```

In [3]:
```
K.clear_session()
```

In [4]:
```
#Rutas del set de imágenes
```

In [5]:
```
datos_entrenamiento='./data/entrenamiento'
datos_validacion='./data/validacion'
```

In [6]:
```
#Vamos a especificar los parametros, tanto de las imagenes que vamos a utilizar como de la
```

In [7]:
```
epocas=35
ancho,alto=125,125
batch_size=25
pasos_validacion=200
filtrosConvolucion1=96/2
filtrosConvolucion2=48
filtrosConvolucion3=32
tamano_kernel=(3,3)
tamano_MaxPool=(2,2)
clases=22
lr=0.0005
```

In [8]:
```
#Primero se preparan las imagenes
```

In [9]:
```
#Normalizamos los valores de los pixeles que van de 0 a 255
entrenamiento_datagen= ImageDataGenerator(
    rescale=1. / 255,
)
```

In [10]:
```
validacion_datagen=ImageDataGenerator(
    rescale=1. / 255)
```

In [11]:
```
entrenamiento_generator=entrenamiento_datagen.flow_from_directory(
    datos_entrenamiento,
    target_size=(alto,ancho),
    batch_size=batch_size,
    class_mode='categorical'
)
```

```
Found 1234 images belonging to 22 classes.
```

In [12]:
```
validacion_generator=validacion_datagen.flow_from_directory(
    datos_validacion,
    target_size=(alto,ancho),
    batch_size=batch_size,
    class_mode='categorical'
)
```

```
Found 241 images belonging to 22 classes.
```

In [13]:
```
def plot_model_history(model_history):
    fig, axs = plt.subplots(1,2,figsize=(15,5))
    #Summarize history for accuracy
    axs[0].plot(range(1,len(model_history.history['acc'])+1),model_history.history['acc'])
    axs[0].plot(range(1,len(model_history.history['val_acc'])+1),model_history.history['va
    axs[0].set_title('Model Accuracy')
    axs[0].set_ylabel('Accuracy')
    axs[0].set_xlabel('Epoch')
    #axs[0].set_xticks(np.arange(1,len(model_history.history['acc'])+1),len(model_his
    axs[0].legend(['train', 'val'], loc='best')
    #summarize history for loss
    axs[1].plot(range(1,len(model_history.history['loss'])+1),model_history.history['loss'
    axs[1].plot(range(1,len(model_history.history['val_loss'])+1),model_history.history['v
    axs[1].set_title('Model Loss')
    axs[1].set_ylabel('Loss')
    axs[1].set_xlabel('Epoch')
    #axs[1].set_xticks(np.arange(1,len(model_history.history['loss'])+1),len(model_hi
    axs[1].legend(['train', 'val'], loc='best')
    plt.show()
```

In [14]:
```
cnn=Sequential()

cnn.add(Convolution2D(filtrosConvolucion1, tamano_kernel, padding='same',input_shape=(anch
#cnn.add(BatchNormalization())
cnn.add(MaxPooling2D(pool_size=tamano_MaxPool))
cnn.add(Dropout(0.5))


#cnn.add(Convolution2D(filtrosConvolucion2,tamano_kernel,padding='same'))
#cnn.add(BatchNormalization())
#cnn.add(MaxPooling2D(pool_size=tamano_MaxPool))
#cnn.add(Dropout(0.5))

#cnn.add(Convolution2D(filtrosConvolucion3,tamano_kernel,padding='same'))
#cnn.add(BatchNormalization())
#cnn.add(MaxPooling2D(pool_size=tamano_MaxPool))
#cnn.add(Dropout(0.4))
```

In [15]:
```
cnn.add(Flatten())
cnn.add(Dense(256, activation='relu'))
cnn.add(Dropout(0.5))
cnn.add(Dense(clases, activation='softmax'))
```

In [16]:
```
cnn.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 125, 125, 48)      1344
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 62, 62, 48)        0
_________________________________________________________________
dropout (Dropout)            (None, 62, 62, 48)        0
_________________________________________________________________
flatten (Flatten)            (None, 184512)            0
_________________________________________________________________
dense (Dense)                (None, 256)               47235328
_________________________________________________________________
dropout_1 (Dropout)          (None, 256)               0
_________________________________________________________________
dense_1 (Dense)              (None, 22)                5654
=================================================================
Total params: 47,242,326
Trainable params: 47,242,326
Non-trainable params: 0
_________________________________________________________________
```

In [17]:
```
optimizer1=tf.keras.optimizers.Adam(learning_rate=lr)
```

In [18]:
```
cnn.compile(loss='categorical_crossentropy',
            optimizer='adam',
             metrics=['accuracy','Recall'])
```

In [19]:
```
history=cnn.fit(
    entrenamiento_generator,
    epochs=epocas,
    validation_data=validacion_generator,
    )
```

```
Epoch 1/35
50/50 [==============================] - 20s 390ms/step - loss: 5.5300 - accuracy: 0.1434
- recall: 0.0194 - val_loss: 2.6796 - val_accuracy: 0.2697 - val_recall: 0.0332
Epoch 2/35
50/50 [==============================] - 19s 387ms/step - loss: 2.1925 - accuracy: 0.3663
- recall: 0.1524 - val_loss: 2.1305 - val_accuracy: 0.4523 - val_recall: 0.1618
Epoch 3/35
50/50 [==============================] - 19s 387ms/step - loss: 1.3273 - accuracy: 0.6305
- recall: 0.4206 - val_loss: 1.8301 - val_accuracy: 0.4730 - val_recall: 0.2822
Epoch 4/35
50/50 [==============================] - 20s 393ms/step - loss: 0.7887 - accuracy: 0.7763
- recall: 0.6151 - val_loss: 1.7727 - val_accuracy: 0.5270 - val_recall: 0.3402
Epoch 5/35
50/50 [==============================] - 21s 420ms/step - loss: 0.4174 - accuracy: 0.8784
- recall: 0.7893 - val_loss: 1.7466 - val_accuracy: 0.5602 - val_recall: 0.3734
Epoch 6/35
50/50 [==============================] - 21s 409ms/step - loss: 0.2948 - accuracy: 0.9214
- recall: 0.8817 - val_loss: 1.9523 - val_accuracy: 0.5062 - val_recall: 0.3942
Epoch 7/35
50/50 [==============================] - 21s 424ms/step - loss: 0.2345 - accuracy: 0.9368
- recall: 0.8995 - val_loss: 1.9359 - val_accuracy: 0.5187 - val_recall: 0.3859
Epoch 8/35
50/50 [==============================] - 20s 396ms/step - loss: 0.1486 - accuracy: 0.9603
```

- recall: 0.9352 - val_loss: 2.0625 - val_accuracy: 0.5436 - val_recall: 0.4564
Epoch 9/35
50/50 [==============================] - 19s 387ms/step - loss: 0.1218 - accuracy: 0.9676
- recall: 0.9571 - val_loss: 2.1187 - val_accuracy: 0.5270 - val_recall: 0.4357
Epoch 10/35
50/50 [==============================] - 20s 391ms/step - loss: 0.1163 - accuracy: 0.9733
- recall: 0.9587 - val_loss: 2.0277 - val_accuracy: 0.5021 - val_recall: 0.4232
Epoch 11/35
50/50 [==============================] - 20s 393ms/step - loss: 0.0956 - accuracy: 0.9773
- recall: 0.9643 - val_loss: 2.1102 - val_accuracy: 0.5062 - val_recall: 0.4315
Epoch 12/35
50/50 [==============================] - 19s 389ms/step - loss: 0.0805 - accuracy: 0.9797
- recall: 0.9733 - val_loss: 2.1682 - val_accuracy: 0.5145 - val_recall: 0.4564
Epoch 13/35
50/50 [==============================] - 21s 414ms/step - loss: 0.0669 - accuracy: 0.9806
- recall: 0.9757 - val_loss: 2.0558 - val_accuracy: 0.5353 - val_recall: 0.4689
Epoch 14/35
50/50 [==============================] - 23s 452ms/step - loss: 0.0539 - accuracy: 0.9862
- recall: 0.9822 - val_loss: 2.1412 - val_accuracy: 0.5394 - val_recall: 0.4689
Epoch 15/35
50/50 [==============================] - 23s 453ms/step - loss: 0.0351 - accuracy: 0.9919
- recall: 0.9887 - val_loss: 2.1447 - val_accuracy: 0.5270 - val_recall: 0.4523
Epoch 16/35
50/50 [==============================] - 22s 435ms/step - loss: 0.0430 - accuracy: 0.9911
- recall: 0.9878 - val_loss: 2.3289 - val_accuracy: 0.5270 - val_recall: 0.4855
Epoch 17/35
50/50 [==============================] - 22s 440ms/step - loss: 0.0488 - accuracy: 0.9895
- recall: 0.9862 - val_loss: 2.1749 - val_accuracy: 0.5104 - val_recall: 0.4606
Epoch 18/35
50/50 [==============================] - 22s 444ms/step - loss: 0.0682 - accuracy: 0.9797
- recall: 0.9765 - val_loss: 2.3227 - val_accuracy: 0.4938 - val_recall: 0.4357
Epoch 19/35
50/50 [==============================] - 22s 440ms/step - loss: 0.0696 - accuracy: 0.9830
- recall: 0.9773 - val_loss: 2.3492 - val_accuracy: 0.5228 - val_recall: 0.4564
Epoch 20/35
50/50 [==============================] - 21s 426ms/step - loss: 0.0370 - accuracy: 0.9919
- recall: 0.9919 - val_loss: 2.4826 - val_accuracy: 0.5145 - val_recall: 0.4564
Epoch 21/35
50/50 [==============================] - 22s 439ms/step - loss: 0.0351 - accuracy: 0.9887
- recall: 0.9870 - val_loss: 2.4130 - val_accuracy: 0.5187 - val_recall: 0.4481
Epoch 22/35
50/50 [==============================] - 22s 444ms/step - loss: 0.0297 - accuracy: 0.9951
- recall: 0.9927 - val_loss: 2.6839 - val_accuracy: 0.4647 - val_recall: 0.4523
Epoch 23/35
50/50 [==============================] - 22s 444ms/step - loss: 0.0392 - accuracy: 0.9878
- recall: 0.9854 - val_loss: 2.4582 - val_accuracy: 0.5145 - val_recall: 0.4689
Epoch 24/35
50/50 [==============================] - 22s 448ms/step - loss: 0.0339 - accuracy: 0.9943
- recall: 0.9919 - val_loss: 2.8442 - val_accuracy: 0.4813 - val_recall: 0.4315
Epoch 25/35
50/50 [==============================] - 23s 450ms/step - loss: 0.0613 - accuracy: 0.9870
- recall: 0.9838 - val_loss: 2.4081 - val_accuracy: 0.4896 - val_recall: 0.4357
Epoch 26/35
50/50 [==============================] - 21s 427ms/step - loss: 0.0238 - accuracy: 0.9951
- recall: 0.9903 - val_loss: 2.5230 - val_accuracy: 0.5145 - val_recall: 0.4855
Epoch 27/35
50/50 [==============================] - 20s 409ms/step - loss: 0.0352 - accuracy: 0.9919
- recall: 0.9911 - val_loss: 2.5288 - val_accuracy: 0.5145 - val_recall: 0.4689
Epoch 28/35
50/50 [==============================] - 21s 411ms/step - loss: 0.0389 - accuracy: 0.9887
- recall: 0.9862 - val_loss: 2.6503 - val_accuracy: 0.5187 - val_recall: 0.4689
Epoch 29/35
50/50 [==============================] - 20s 409ms/step - loss: 0.0246 - accuracy: 0.9927
- recall: 0.9911 - val_loss: 2.7798 - val_accuracy: 0.5519 - val_recall: 0.4938
Epoch 30/35
50/50 [==============================] - 21s 409ms/step - loss: 0.0316 - accuracy: 0.9903

```
- recall: 0.9887 - val_loss: 2.6262 - val_accuracy: 0.5187 - val_recall: 0.4647
Epoch 31/35
50/50 [==============================] - 20s 411ms/step - loss: 0.0234 - accuracy: 0.9951
- recall: 0.9943 - val_loss: 2.8073 - val_accuracy: 0.5187 - val_recall: 0.4606
Epoch 32/35
50/50 [==============================] - 20s 409ms/step - loss: 0.0258 - accuracy: 0.9903
- recall: 0.9878 - val_loss: 2.7668 - val_accuracy: 0.5353 - val_recall: 0.5062
Epoch 33/35
50/50 [==============================] - 20s 407ms/step - loss: 0.0391 - accuracy: 0.9903
- recall: 0.9895 - val_loss: 2.5861 - val_accuracy: 0.5353 - val_recall: 0.4689
Epoch 34/35
50/50 [==============================] - 20s 406ms/step - loss: 0.0397 - accuracy: 0.9887
- recall: 0.9862 - val_loss: 2.6934 - val_accuracy: 0.4855 - val_recall: 0.4647
Epoch 35/35
50/50 [==============================] - 20s 408ms/step - loss: 0.0533 - accuracy: 0.9878
- recall: 0.9846 - val_loss: 2.3733 - val_accuracy: 0.5104 - val_recall: 0.4440
```

In [ ]:

```
plot_model_history (history)
```

In [ ]:

```
target_dir = './modelo/'
if not os.path.exists(target_dir):
  os.mkdir(target_dir)
cnn.save('./modelo/modelo.h5')
cnn.save_weights('./modelo/pesos.h5')
```

In [ ]:

In [ ]:

TFM_MODELO.V2

In [1]:
```
# Importamos todas las librerarías que vamos a necesitar para ejecutar nuestro modelo
```

In [2]:
```
import sys
import os
import tensorflow as tf
import numpy as np
from tensorflow.python.keras.preprocessing.image import ImageDataGenerator
from tensorflow.python.keras import optimizers
from tensorflow.python.keras.models import Sequential
from tensorflow.python.keras.layers import Dropout, Flatten, Dense, Activation
from tensorflow.python.keras.layers import Convolution2D, MaxPooling2D
from tensorflow.python.keras.layers import BatchNormalization
from tensorflow.python.keras import backend as K
import matplotlib.pyplot as plt
import pathlib
from keras.callbacks import EarlyStopping
```

In [3]:
```
#Cerramos todos los procesos abiertos por Keras
```

In [4]:
```
K.clear_session()
```

In [5]:
```
#Rutas del set de imágenes
```

In [6]:
```
datos_entrenamiento='./data/entrenamiento'
datos_validacion='./data/validacion'
```

In [7]:
```
#Vamos a especificar los parametros, tanto de las imagenes que vamos
#a utilizar como de la red neuronal
```

In [8]:
```
epocas=100
ancho,alto=125,125
batch_size=25
pasos_validacion=200
filtrosConvolucion1=96/4
filtrosConvolucion2=240/4
filtrosConvolucion3=32
tamano_kernel=(3,3)
tamano_MaxPool=(2,2)
clases=22
lr=0.0005
Paciencia=12
```

In [9]:
```
#Primero se preparan las imagenes
```

In [10]:
```
#Normalizamos los valores de los pixeles que van de 0 a 255
entrenamiento_datagen= ImageDataGenerator(
    rescale=1. / 255,
    #rotation_range=20,
```

```
    #zoom_range=0.2,
    #width_shift_range=0.1,
    #height_shift_range=0.1,
    #horizontal_flip=True,
    #vertical_flip=False
)
```

In [11]:
```
validacion_datagen=ImageDataGenerator(
    rescale=1. / 255)
```

In [12]:
```
entrenamiento_generator=entrenamiento_datagen.flow_from_directory(
    datos_entrenamiento,
    target_size=(alto,ancho),
    batch_size=batch_size,
    class_mode='categorical'
)
```

```
Found 1234 images belonging to 22 classes.
```

In [13]:
```
validacion_generator=validacion_datagen.flow_from_directory(
    datos_validacion,
    target_size=(alto,ancho),
    batch_size=batch_size,
    class_mode='categorical'
)
```

```
Found 241 images belonging to 22 classes.
```

In [ ]:
```
# Definimos la paciencia
```

In [14]:
```
es = EarlyStopping(monitor='val_loss', mode='min', verbose=1, patience=Paciencia)
```

In [15]:
```
cnn=Sequential()

cnn.add(Convolution2D(filtrosConvolucion1, tamano_kernel, padding='same',
                      input_shape=(ancho,alto,3),activation='relu'))
#cnn.add(BatchNormalization())
cnn.add(MaxPooling2D(pool_size=tamano_MaxPool))
#cnn.add(Dropout(0.5))


cnn.add(Convolution2D(filtrosConvolucion2,tamano_kernel,padding='same',
                      activation='relu'))
#cnn.add(BatchNormalization())
cnn.add(MaxPooling2D(pool_size=tamano_MaxPool))
#cnn.add(Dropout(0.5))

cnn.add(Convolution2D(filtrosConvolucion3,tamano_kernel,padding='same',
                      activation='relu'))
#cnn.add(BatchNormalization())
cnn.add(MaxPooling2D(pool_size=tamano_MaxPool))
#cnn.add(Dropout(0.4))
```

In [16]:
```
cnn.add(Flatten())
cnn.add(Dense(256/4, activation='relu'))
cnn.add(Dropout(0.5))
#cnn.add(Dense(256/8, activation='relu'))
```

```
#cnn.add(Dropout(0.5))
cnn.add(Dense(clases, activation='softmax'))
```

In [17]:
```
cnn.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 125, 125, 24)      672
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 62, 62, 24)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 62, 62, 60)        13020
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 31, 31, 60)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 31, 31, 32)        17312
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 15, 15, 32)        0
_________________________________________________________________
flatten (Flatten)            (None, 7200)              0
_________________________________________________________________
dense (Dense)                (None, 64)                460864
_________________________________________________________________
dropout (Dropout)            (None, 64)                0
_________________________________________________________________
dense_1 (Dense)              (None, 22)                1430
=================================================================
Total params: 493,298
Trainable params: 493,298
Non-trainable params: 0
_________________________________________________________________
```

In [18]:
```
optimizer1=tf.keras.optimizers.Adam(learning_rate=lr)
```

In [19]:
```
cnn.compile(loss='categorical_crossentropy',
            optimizer='adam',
             metrics=['accuracy','Recall'])
```

In [20]:
```
history=cnn.fit(
    entrenamiento_generator,
    epochs=epocas,
    validation_data=validacion_generator,
    callbacks=[es]
    )
```

```
Epoch 1/100
50/50 [==============================] - 12s 228ms/step - loss: 3.0409 - accuracy: 0.0818
- recall: 0.0000e+00 - val_loss: 2.8857 - val_accuracy: 0.1162 - val_recall: 0.0249
Epoch 2/100
50/50 [==============================] - 12s 230ms/step - loss: 2.8508 - accuracy: 0.1272
- recall: 0.0251 - val_loss: 2.7247 - val_accuracy: 0.2116 - val_recall: 0.0415
Epoch 3/100
50/50 [==============================] - 11s 229ms/step - loss: 2.6022 - accuracy: 0.1994
- recall: 0.0413 - val_loss: 2.5107 - val_accuracy: 0.3029 - val_recall: 0.0373
Epoch 4/100
50/50 [==============================] - 12s 238ms/step - loss: 2.3221 - accuracy: 0.2877
- recall: 0.0948 - val_loss: 2.2835 - val_accuracy: 0.3568 - val_recall: 0.1037
Epoch 5/100
50/50 [==============================] - 12s 240ms/step - loss: 2.0836 - accuracy: 0.3549
```

```
- recall: 0.1467 - val_loss: 2.1471 - val_accuracy: 0.3983 - val_recall: 0.1743
Epoch 6/100
50/50 [==============================] - 12s 236ms/step - loss: 1.8465 - accuracy: 0.4384
- recall: 0.2123 - val_loss: 2.1271 - val_accuracy: 0.4191 - val_recall: 0.2075
Epoch 7/100
50/50 [==============================] - 12s 233ms/step - loss: 1.6627 - accuracy: 0.5016
- recall: 0.2958 - val_loss: 2.0910 - val_accuracy: 0.4523 - val_recall: 0.2365
Epoch 8/100
50/50 [==============================] - 12s 239ms/step - loss: 1.4918 - accuracy: 0.5324
- recall: 0.3436 - val_loss: 1.9589 - val_accuracy: 0.5021 - val_recall: 0.2988
Epoch 9/100
50/50 [==============================] - 12s 236ms/step - loss: 1.3073 - accuracy: 0.5948
- recall: 0.4149 - val_loss: 1.9355 - val_accuracy: 0.5021 - val_recall: 0.3195
Epoch 10/100
50/50 [==============================] - 11s 225ms/step - loss: 1.1941 - accuracy: 0.6110
- recall: 0.4643 - val_loss: 2.0422 - val_accuracy: 0.4938 - val_recall: 0.3361
Epoch 11/100
50/50 [==============================] - 11s 224ms/step - loss: 1.0789 - accuracy: 0.6588
- recall: 0.5154 - val_loss: 2.0042 - val_accuracy: 0.5021 - val_recall: 0.3734
Epoch 12/100
50/50 [==============================] - 11s 229ms/step - loss: 1.0010 - accuracy: 0.6759
- recall: 0.5567 - val_loss: 2.0006 - val_accuracy: 0.5228 - val_recall: 0.3817
Epoch 13/100
50/50 [==============================] - 12s 231ms/step - loss: 0.9068 - accuracy: 0.6985
- recall: 0.5851 - val_loss: 2.1568 - val_accuracy: 0.5353 - val_recall: 0.4191
Epoch 14/100
50/50 [==============================] - 11s 226ms/step - loss: 0.8536 - accuracy: 0.7245
- recall: 0.6094 - val_loss: 2.0795 - val_accuracy: 0.5394 - val_recall: 0.4108
Epoch 15/100
50/50 [==============================] - 12s 235ms/step - loss: 0.7604 - accuracy: 0.7480
- recall: 0.6491 - val_loss: 2.1427 - val_accuracy: 0.5436 - val_recall: 0.4398
Epoch 16/100
50/50 [==============================] - 11s 226ms/step - loss: 0.6146 - accuracy: 0.7836
- recall: 0.7172 - val_loss: 2.2562 - val_accuracy: 0.5228 - val_recall: 0.4232
Epoch 17/100
50/50 [==============================] - 11s 228ms/step - loss: 0.6199 - accuracy: 0.7893
- recall: 0.7123 - val_loss: 2.4163 - val_accuracy: 0.5519 - val_recall: 0.4606
Epoch 18/100
50/50 [==============================] - 12s 240ms/step - loss: 0.5754 - accuracy: 0.7998
- recall: 0.7310 - val_loss: 2.3343 - val_accuracy: 0.5477 - val_recall: 0.4813
Epoch 19/100
50/50 [==============================] - 12s 236ms/step - loss: 0.5565 - accuracy: 0.8120
- recall: 0.7447 - val_loss: 2.3522 - val_accuracy: 0.5519 - val_recall: 0.5021
Epoch 20/100
50/50 [==============================] - 11s 229ms/step - loss: 0.4960 - accuracy: 0.8185
- recall: 0.7682 - val_loss: 2.3516 - val_accuracy: 0.5394 - val_recall: 0.4855
Epoch 21/100
50/50 [==============================] - 11s 223ms/step - loss: 0.4807 - accuracy: 0.8387
- recall: 0.7804 - val_loss: 2.4640 - val_accuracy: 0.5394 - val_recall: 0.4813
Epoch 00021: early stopping
```

In [ ]:

In [21]:

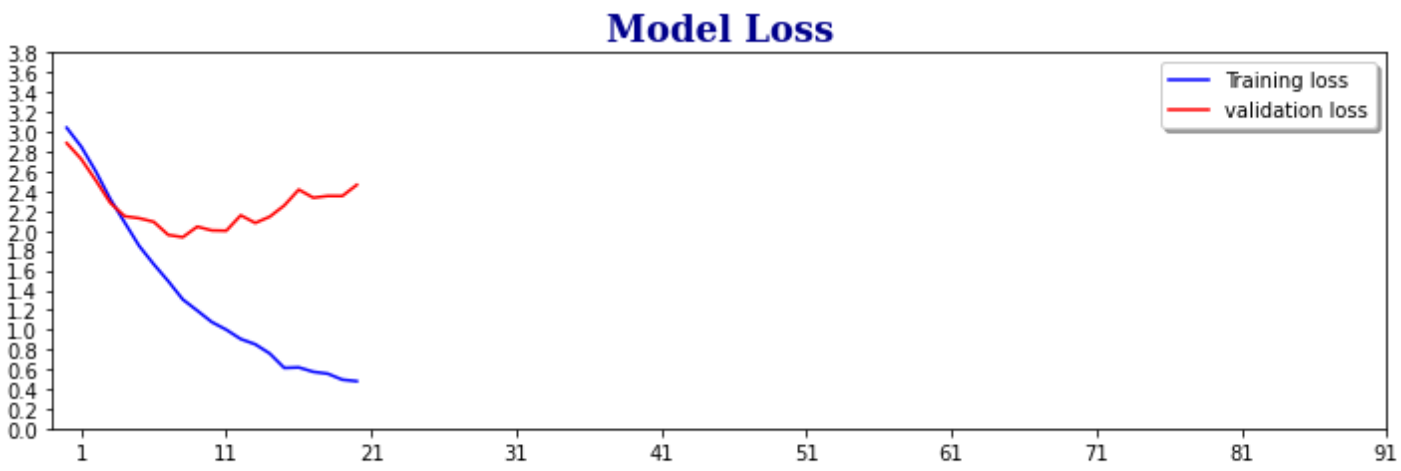
```
target_dir = './modelo/'
if not os.path.exists(target_dir):
  os.mkdir(target_dir)
cnn.save('./modelo/modelo2.h5')
cnn.save_weights('./modelo/pesos2.h5')
```

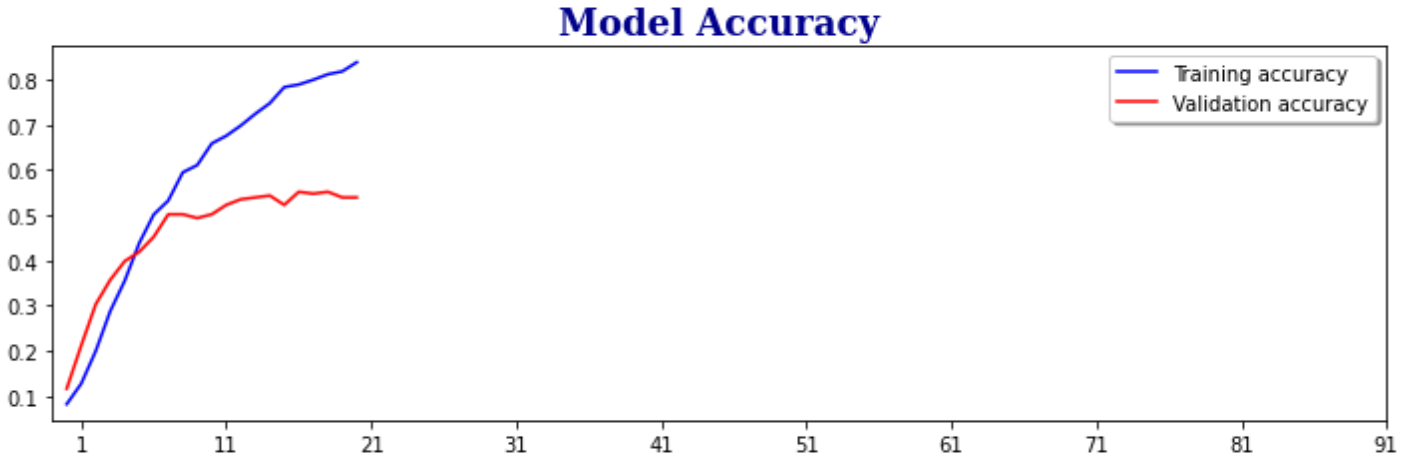
In [22]:

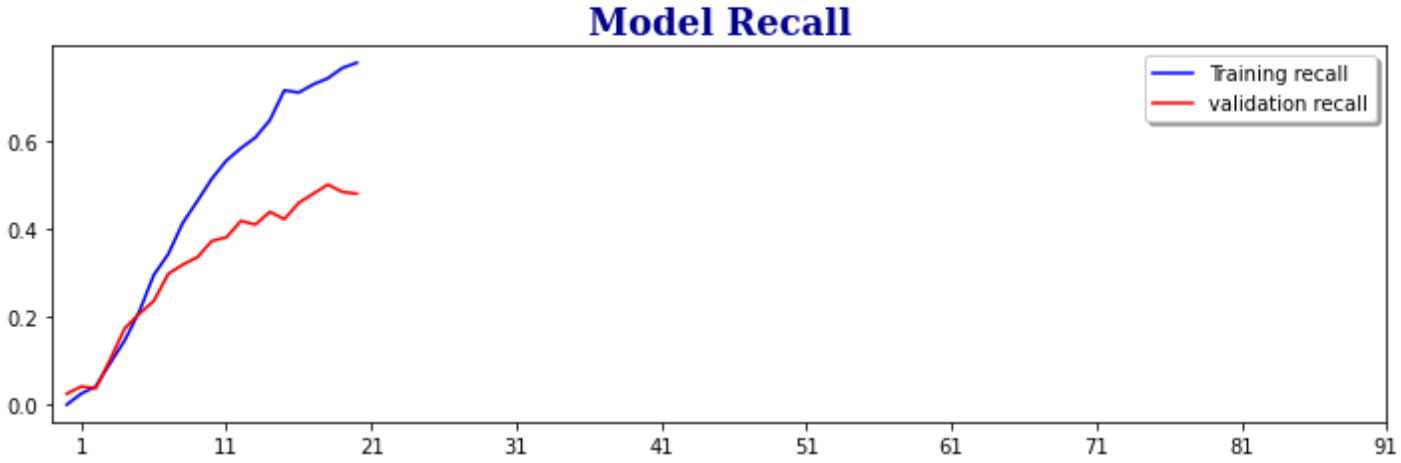
```
fig, (ax1, ax2, ax3) = plt.subplots(3, 1, figsize=(10,
10))
ax1.plot(history.history['loss'], color='b',
```

```
label="Training loss")
ax1.plot(history.history['val_loss'], color='r',
label="validation loss")
ax1.legend(loc='best',shadow=True)
ax1.set_xticks(np.arange(1, epocas, 10))
ax1.set_yticks(np.arange(0, 4, 0.2))
ax1.set_title("Model Loss",
          fontdict={'family': 'serif',
                    'color' : 'darkblue',
                    'weight': 'bold',
                    'size': 18})
ax2.plot(history.history['accuracy'], color='b',
label="Training accuracy")
ax2.plot(history.history['val_accuracy'],
color='r',label="Validation accuracy")
ax2.set_xticks(np.arange(1, epocas, 10))
ax2.legend(loc='best', shadow=True)
ax2.set_title("Model Accuracy",
          fontdict={'family': 'serif',
                    'color' : 'darkblue',
                    'weight': 'bold',
                    'size': 18})
ax3.plot(history.history['recall'], color='b',
label="Training recall")
ax3.plot(history.history['val_recall'], color='r',
label="validation recall")
ax3.legend(loc='best',shadow=True)
ax3.set_xticks(np.arange(1, epocas, 10))
ax3.set_yticks(np.arange(0, 0.8, 0.2))
ax3.set_title("Model Recall",
          fontdict={'family': 'serif',
                    'color' : 'darkblue',
                    'weight': 'bold',
                    'size': 18})


plt.tight_layout()
plt.show()
```


Model Loss
Training loss
validation loss
3.8
3.6
3.4
3.2
3.0
2.8
2.6
2.4
2.2
2.0
1.8
1.6
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
1
11
21
31
41
51
61
71
81
91


Model Accuracy
Training accuracy
Validation accuracy
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
1
11
21
31
41
51
61
71
81
91


Model Recall
Training recall
validation recall
0.6
0.4
0.2
0.0
1
11
21
31
41
51
61
71
81
91

In [ ]:

In [ ]:

In [ ]:

TFM_MODELO_FINAL

In [1]:
```
# Importamos todas las librerarías que vamos a necesitar para ejecutar nuestro modelo
```

In [1]:
```
import sys
import os
import tensorflow as tf
from keras.models import Sequential, Model
from keras.layers import Conv2D, MaxPool2D, Dense, Flatten, Dropout, BatchNormalization, I
from keras.callbacks import TensorBoard, ModelCheckpoint
from keras.utils import np_utils
import os
import numpy as np
from keras.preprocessing import image
from keras.applications.imagenet_utils import preprocess_input, decode_predictions
from keras.applications.vgg16 import VGG16
from keras.preprocessing.image import ImageDataGenerator
from sklearn.utils import shuffle
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
from tensorflow.python.keras import optimizers
from tensorflow.python.keras import backend as K
import matplotlib.pyplot as plt
from keras.applications.vgg16 import VGG16
from keras.callbacks import EarlyStopping
from tensorflow.keras.utils import get_source_inputs
from keras.layers import GlobalAveragePooling2D
```

In [2]:
```
#Cerramos todos los procesos abiertos por Keras
```

In [3]:
```
K.clear_session()
```

In [4]:
```
#Rutas del set de imágenes
```

In [5]:
```
datos_entrenamiento='./data/entrenamiento'
datos_validacion='./data/validacion'
```

In [6]:
```
#Vamos a especificar los parametros, tanto de las imagenes que vamos a utilizar como de la
```

In [7]:
```
epocas=35
ancho,alto=224,224
batch_size=25
tamano_kernel=(3,3)
tamano_MaxPool=(2,2)
clases=22
paciencia=7
IMG_SHAPE=(ancho,alto,3)
tensor=Input(shape=(IMG_SHAPE))
```

In [8]:
```
#Primero se preparan las imagenes
```

In [9]:
```
#Normalizamos los valores de los pixeles que van de 0 a 255
entrenamiento_datagen= ImageDataGenerator(
    rescale=1. / 255,
    rotation_range=20,
    zoom_range=0.2,
    width_shift_range=0.1,
    height_shift_range=0.1,
    horizontal_flip=True,
    vertical_flip=False,
)
```

In [10]:
```
validacion_datagen=ImageDataGenerator(
    rescale=1. / 255)
```

In [11]:
```
entrenamiento_generator=entrenamiento_datagen.flow_from_directory(
    datos_entrenamiento,
    target_size=(alto,ancho),
    batch_size=batch_size,
    class_mode='categorical',
)
```

```
Found 1234 images belonging to 22 classes.
```

In [12]:
```
validacion_generator=validacion_datagen.flow_from_directory(
    datos_validacion,
    target_size=(alto,ancho),
    batch_size=batch_size,
    class_mode='categorical',
)
```

```
Found 241 images belonging to 22 classes.
```

In [13]:
```
#Especificamos el earlyStopping, es decir, el número de veces para parar el entrenamiento
```

In [14]:
```
es = EarlyStopping(monitor='val_loss', mode='min', verbose=1, patience=paciencia)
checkpoint_filepath = './checkpoint'
model_checkpoint_callback = tf.keras.callbacks.ModelCheckpoint(
    filepath=checkpoint_filepath,
    save_weights_only=False,
    monitor='val_accuracy',
    mode='max',
    save_best_only=True)
```

In [15]:
```
image_input = Input(shape=(ancho, alto, 3))

model = VGG16(input_tensor=tensor, include_top=False,weights='imagenet')
model.summary()
```

```
Model: "vgg16"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 224, 224, 3)]     0
_________________________________________________________________
block1_conv1 (Conv2D)        (None, 224, 224, 64)      1792
_________________________________________________________________
block1_conv2 (Conv2D)        (None, 224, 224, 64)      36928
_________________________________________________________________
```

```
block1_pool (MaxPooling2D)   (None, 112, 112, 64)      0
_________________________________________________________________
block2_conv1 (Conv2D)        (None, 112, 112, 128)     73856
_________________________________________________________________
block2_conv2 (Conv2D)        (None, 112, 112, 128)     147584
_________________________________________________________________
block2_pool (MaxPooling2D)   (None, 56, 56, 128)       0
_________________________________________________________________
block3_conv1 (Conv2D)        (None, 56, 56, 256)       295168
_________________________________________________________________
block3_conv2 (Conv2D)        (None, 56, 56, 256)       590080
_________________________________________________________________
block3_conv3 (Conv2D)        (None, 56, 56, 256)       590080
_________________________________________________________________
block3_pool (MaxPooling2D)   (None, 28, 28, 256)       0
_________________________________________________________________
block4_conv1 (Conv2D)        (None, 28, 28, 512)       1180160
_________________________________________________________________
block4_conv2 (Conv2D)        (None, 28, 28, 512)       2359808
_________________________________________________________________
block4_conv3 (Conv2D)        (None, 28, 28, 512)       2359808
_________________________________________________________________
block4_pool (MaxPooling2D)   (None, 14, 14, 512)       0
_________________________________________________________________
block5_conv1 (Conv2D)        (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_conv2 (Conv2D)        (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_conv3 (Conv2D)        (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_pool (MaxPooling2D)   (None, 7, 7, 512)         0
=================================================================
Total params: 14,714,688
Trainable params: 14,714,688
Non-trainable params: 0
_________________________________________________________________
```

In [20]:

```
#last_layer = model.get_layer('fc2').output
model_output=model.output
avg_pool = GlobalAveragePooling2D(name='marcas_pooling')(model_output)
avg_pool.get_shape()
#fully_coneected=Dense(564, activation='relu', name= 'marcasFull')()
out = Dense(clases, activation='softmax', name='output')(avg_pool)
inputs = get_source_inputs(tensor)

custom_vgg_model = Model(inputs=inputs, outputs=out)
```

In [21]:

```
custom_vgg_model.summary()
```

```
Model: "model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 224, 224, 3)]     0
_________________________________________________________________
block1_conv1 (Conv2D)        (None, 224, 224, 64)      1792
_________________________________________________________________
block1_conv2 (Conv2D)        (None, 224, 224, 64)      36928
_________________________________________________________________
block1_pool (MaxPooling2D)   (None, 112, 112, 64)      0
_________________________________________________________________
block2_conv1 (Conv2D)        (None, 112, 112, 128)     73856
_________________________________________________________________
```

```
block2_conv2 (Conv2D)          (None, 112, 112, 128)     147584
_________________________________________________________________
block2_pool (MaxPooling2D)     (None, 56, 56, 128)       0
_________________________________________________________________
block3_conv1 (Conv2D)          (None, 56, 56, 256)       295168
_________________________________________________________________
block3_conv2 (Conv2D)          (None, 56, 56, 256)       590080
_________________________________________________________________
block3_conv3 (Conv2D)          (None, 56, 56, 256)       590080
_________________________________________________________________
block3_pool (MaxPooling2D)     (None, 28, 28, 256)       0
_________________________________________________________________
block4_conv1 (Conv2D)          (None, 28, 28, 512)       1180160
_________________________________________________________________
block4_conv2 (Conv2D)          (None, 28, 28, 512)       2359808
_________________________________________________________________
block4_conv3 (Conv2D)          (None, 28, 28, 512)       2359808
_________________________________________________________________
block4_pool (MaxPooling2D)     (None, 14, 14, 512)       0
_________________________________________________________________
block5_conv1 (Conv2D)          (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_conv2 (Conv2D)          (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_conv3 (Conv2D)          (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_pool (MaxPooling2D)     (None, 7, 7, 512)         0
_________________________________________________________________
marcas_pooling (GlobalAverag (None, 512)               0
_________________________________________________________________
output (Dense)                 (None, 22)                11286
=================================================================
Total params: 14,725,974
Trainable params: 14,725,974
Non-trainable params: 0
_________________________________________________________________
```

In [22]:

```
for layer in custom_vgg_model.layers[:-5]:
    layer.trainable = False
```

In [23]:

```
custom_vgg_model.summary()
```

```
Model: "model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 224, 224, 3)]     0
_________________________________________________________________
block1_conv1 (Conv2D)        (None, 224, 224, 64)      1792
_________________________________________________________________
block1_conv2 (Conv2D)        (None, 224, 224, 64)      36928
_________________________________________________________________
block1_pool (MaxPooling2D)   (None, 112, 112, 64)      0
_________________________________________________________________
block2_conv1 (Conv2D)        (None, 112, 112, 128)     73856
_________________________________________________________________
block2_conv2 (Conv2D)        (None, 112, 112, 128)     147584
_________________________________________________________________
block2_pool (MaxPooling2D)   (None, 56, 56, 128)       0
_________________________________________________________________
block3_conv1 (Conv2D)        (None, 56, 56, 256)       295168
_________________________________________________________________
block3_conv2 (Conv2D)        (None, 56, 56, 256)       590080
```

```
block3_conv3 (Conv2D)        (None, 56, 56, 256)       590080
_________________________________________________________________
block3_pool (MaxPooling2D)   (None, 28, 28, 256)       0
_________________________________________________________________
block4_conv1 (Conv2D)        (None, 28, 28, 512)       1180160
_________________________________________________________________
block4_conv2 (Conv2D)        (None, 28, 28, 512)       2359808
_________________________________________________________________
block4_conv3 (Conv2D)        (None, 28, 28, 512)       2359808
_________________________________________________________________
block4_pool (MaxPooling2D)   (None, 14, 14, 512)       0
_________________________________________________________________
block5_conv1 (Conv2D)        (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_conv2 (Conv2D)        (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_conv3 (Conv2D)        (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_pool (MaxPooling2D)   (None, 7, 7, 512)         0
_________________________________________________________________
marcas_pooling (GlobalAverag (None, 512)               0
_________________________________________________________________
output (Dense)               (None, 22)                11286
=================================================================
Total params: 14,725,974
Trainable params: 4,730,902
Non-trainable params: 9,995,072
_________________________________________________________________
```

In [24]:
```
custom_vgg_model.compile(loss='categorical_crossentropy',optimizer='adam',metrics=['accura
```

In [25]:
```
history=custom_vgg_model.fit(
    entrenamiento_generator,
    epochs=epocas,
    validation_data=validacion_generator,
    callbacks=[es]
)
```

```
Epoch 1/35
50/50 [==============================] - 216s 4s/step - loss: 2.2077 - accuracy: 0.3736 -
val_loss: 1.7419 - val_accuracy: 0.5104
Epoch 2/35
50/50 [==============================] - 217s 4s/step - loss: 1.0243 - accuracy: 0.7075 -
val_loss: 1.3456 - val_accuracy: 0.6556
Epoch 3/35
50/50 [==============================] - 213s 4s/step - loss: 0.5569 - accuracy: 0.8444 -
val_loss: 1.0700 - val_accuracy: 0.7178
Epoch 4/35
50/50 [==============================] - 218s 4s/step - loss: 0.4124 - accuracy: 0.8825 -
val_loss: 1.0368 - val_accuracy: 0.7635
Epoch 5/35
50/50 [==============================] - 216s 4s/step - loss: 0.3152 - accuracy: 0.9060 -
val_loss: 0.9561 - val_accuracy: 0.7718
Epoch 6/35
50/50 [==============================] - 214s 4s/step - loss: 0.2821 - accuracy: 0.9263 -
val_loss: 1.0261 - val_accuracy: 0.7635
Epoch 7/35
50/50 [==============================] - 213s 4s/step - loss: 0.1685 - accuracy: 0.9481 -
val_loss: 0.9758 - val_accuracy: 0.8216
Epoch 8/35
50/50 [==============================] - 217s 4s/step - loss: 0.1777 - accuracy: 0.9530 -
val_loss: 1.0492 - val_accuracy: 0.8008
```

```
Epoch 9/35
50/50 [==============================] - 216s 4s/step - loss: 0.1271 - accuracy: 0.9611 -
val_loss: 1.0618 - val_accuracy: 0.8465
Epoch 10/35
50/50 [==============================] - 218s 4s/step - loss: 0.0978 - accuracy: 0.9684 -
val_loss: 1.1587 - val_accuracy: 0.8050
Epoch 11/35
50/50 [==============================] - 226s 5s/step - loss: 0.0748 - accuracy: 0.9806 -
val_loss: 1.0887 - val_accuracy: 0.8340
Epoch 12/35
50/50 [==============================] - 215s 4s/step - loss: 0.0696 - accuracy: 0.9814 -
val_loss: 1.1885 - val_accuracy: 0.8008
Epoch 00012: early stopping
```

In [ ]:

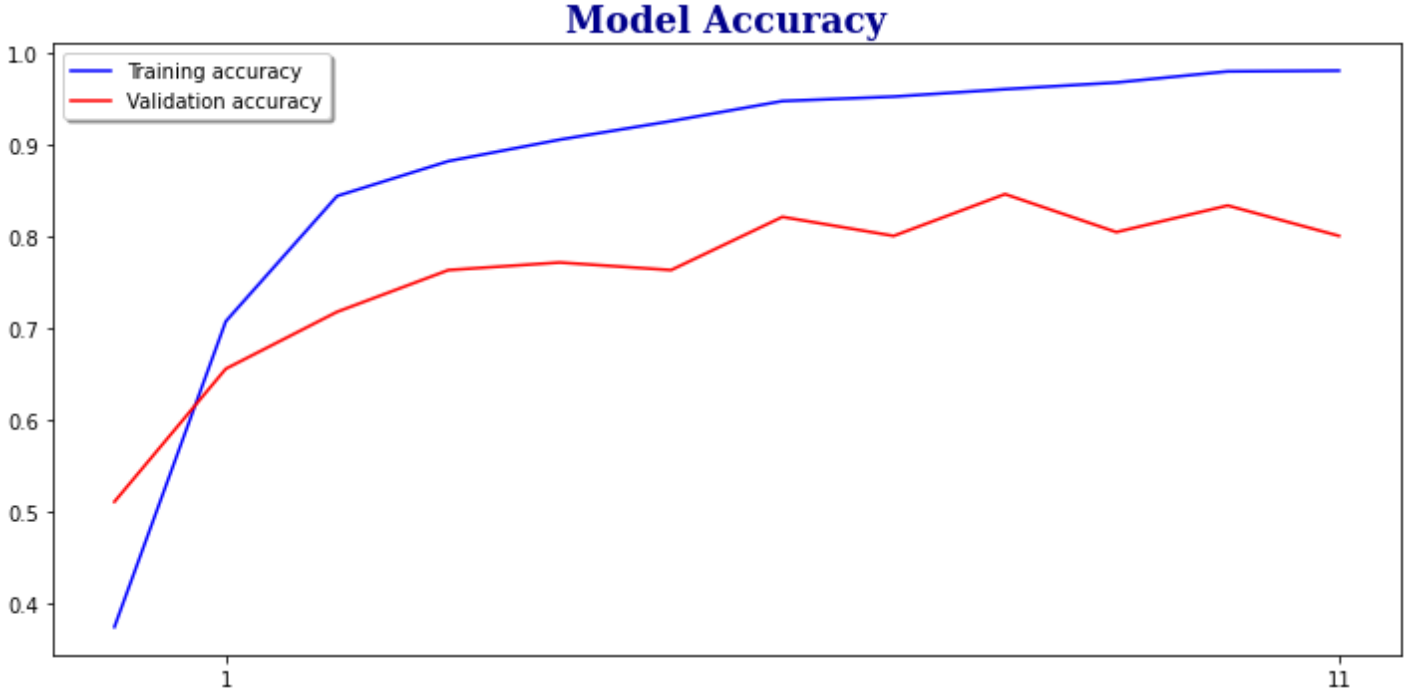
```
target_dir = './modelo/'
if not os.path.exists(target_dir):
  os.mkdir(target_dir)
custom_vgg_model.save('./modelo/modeloTransfer1.h5')
custom_vgg_model.save_weights('./modelo/pesosTransfer1.h5')
```

In [27]:

```
fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(10,
10))
ax1.plot(history.history['loss'], color='b',
label="Training loss")
ax1.plot(history.history['val_loss'], color='r',
label="validation loss")
ax1.legend(loc='best',shadow=True)
ax1.set_xticks(np.arange(1, 12, 10))
ax1.set_yticks(np.arange(0, 4, 0.2))
ax1.set_title("Model Loss",
          fontdict={'family': 'serif',
                    'color' : 'darkblue',
                    'weight': 'bold',
                    'size': 18})
ax2.plot(history.history['accuracy'], color='b',
label="Training accuracy")
ax2.plot(history.history['val_accuracy'],
color='r',label="Validation accuracy")
ax2.set_xticks(np.arange(1, 12, 10))
ax2.legend(loc='best', shadow=True)
ax2.set_title("Model Accuracy",
          fontdict={'family': 'serif',
                    'color' : 'darkblue',
                    'weight': 'bold',
                    'size': 18})
#ax3.plot(history.history['recall'], color='b',
#label="Training recall")
#ax3.plot(history.history['val_recall'], color='r',
#label="validation recall")
#ax3.legend(loc='best',shadow=True)
#ax3.set_xticks(np.arange(1, epocas, 10))
#ax3.set_yticks(np.arange(0, 0.8, 0.2))
#ax3.set_title("Model Recall",
#          fontdict={'family': 'serif',
#                    'color' : 'darkblue',
#                    'weight': 'bold',
#                    'size': 18})


plt.tight_layout()
plt.show()
```


Model Loss
Training loss
validation loss
3.8
3.6
3.4
3.2
3.0
2.8
2.6
2.4
2.2
2.0
1.8
1.6
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
1
11


Model Accuracy
Training accuracy
Validation accuracy
1.0
0.9
0.8
0.7
0.6
0.5
0.4
1
11

In [ ]:

In [ ]:

In [ ]:

In [ ]: