UNIVERSIDAD EUROPEA DE MADRID

ESCUELA DE ARQUITECTURA, INGENIERÍA Y DISEÑO

MÁSTER UNIVERSITARIO EN

BIG DATA ANALYTICS - MBI

TRABAJO FIN DE MÁSTER

Clasificación de logos de marcas mediante Deep Learning

ADRIÁN SANTOS MENA

CURSO 2020-2021

Universidad Europea

TÍTULO: Clasificación de logos de marcas mediante Deep Learning

AUTOR: ADRIÁN SANTOS MENA

TITULACIÓN: BIG DATA ANALYTICS - MBI

DIRECTOR DEL PROYECTO: Luis Fernández Ortega

FECHA: Octubre de 2021

RESUMEN

Para el proyecto que se va a realizar, se ha escogido el título **"Clasificador de logos"** debido a que es un título que describe a la perfección y en pocas palabras la finalidad de dicho proyecto: **crear un modelo de clasificación que consiga distinguir y clasificar imágenes con logos de unas marcas previamente seleccionadas.**

La elección del tema, además de haber sido una propuesta realizada por la empresa IBM, tiene también que ver con el auge que está teniendo actualmente la inteligencia artificial.

Hace unos años nadie se pensaría que la inteligencia artificial estaría tan presente en nuestras vidas y, para muchas de las empresas actuales, es un medio de mejora tanto económica como organizacional muy importante. Esto se debe a que gracias a la cantidad de datos existentes y la capacidad de creación de modelos de aprendizaje que hay actualmente, se pueden obtener patrones e información nueva que puede ayudar a la empresa a mejorar en cualquier sentido.

También podemos encontrar la inteligencia artificial en nuestra vida cotidiana, ya que sin ir más lejos, asistentes virtuales como Siri o Google Assistant hacen uso de ella, además de muchas otras aplicaciones que usamos diariamente y no nos damos cuenta.

Es por ello que se ha elegido este tema como Trabajo de Fin de Máster, ya que además de la importancia que la inteligencia artificial va a ir cogiendo a lo largo de los años, siempre está bien adentrarse y obtener conocimiento sobre algo que va a repercutir tanto en el futuro.

Para la realización del proyecto se va a empezar desde 0, es decir, como si nunca hubiésemos oído hablar de lo que es el Deep Learning.

Primero se va a indagar sobre cómo funciona el paradigma del Deep Learning comenzando por un poco de historia sobre esta técnica y como se ha ido desarrollando a lo largo de los años desde su descubrimiento y primera implementación.

Posteriormente se explicará qué es y cómo funciona la técnica del Deep Learning, describiendo al detalle el funcionamiento de una red neuronal y de las neuronas que la conforman.

También se describirán cada uno de los tipos de redes neuronales y neuronas que existen, adentrándonos en una explicación un poco más técnica pero con intención de que sea sencilla y fácil de entender y así tener las bases para ir entendiendo paso a paso la realización del proyecto.

Y, finalmente, además del presupuesto que se necesitaría en caso de ser un proyecto financiado y cuál ha sido la cronología de desarrollo mediante un diagrama Gantt, se explicará cómo se han desarrollado los modelos de clasificación, analizando los cambios que se han ido implementando, qué tipo de pruebas se han realizado, librerías con su respectiva explicación y análisis del modelo finalmente escogido.

ABSTRACT

For the project to be carried out, the title "Logo classifier" has been chosen because it is a title that perfectly fits with the purpose of the project: to create a classification model that manages to distinguish and classify images with logos of previously selected brands.

The choice of the topic, in addition to being a proposal made by IBM, is also related to the current boom in artificial intelligence.

Years ago nobody would have thought that artificial intelligence would be so present in our lives and, for many of today's companies, can be so influential in economic and organizational improvement. That's the reason why the amount of existing data and the capacity to create learning models help to find new patterns and information that can help the company to improve in any way.

We can also find artificial intelligence in our daily lives. For example we use virtual assistants such as Siri or Google Assistant, in addition to many other applications that we use daily and do not realize it.

That is why this topic has been chosen as a Master's Thesis, because in addition to the importance that artificial intelligence will be taking over the years, it is always good to go deep and gain knowledge about something that will impact so much in the future.

First we will investigate how the Deep Learning paradigm works, starting with the history of this technique and how it has developed over the years since its discovery and first implementation.

Afterwards, we will explain what Deep Learning is and how it works, describing how a neural network and its neurons work.

We will also describe each of the types of neural networks and neurons that exist, going into a slightly more technical explanation but with the intention of making it simple and easy to understand and thus have the basis to understand step by step the realization of the project.

And finally, in addition to the budget that would be needed in case of being a funded project and what has been the chronology of development through a Gantt chart, it will be explained how the classification models have been developed, analyzing the changes that have been implemented, what kind of tests have been performed, libraries with their respective explanation and analysis of the model finally chosen.

TABLA RESUMEN

	DATOS
Nombre y apellidos:	Adrián Santos Mena
Título del proyecto:	Clasificación de logos de marcas mediante Deep Learning
Directores del proyecto:	Luis Fernández Ortega
El proyecto se ha realizado en colaboración de una empresa o a petición de un amepresa	SI El proyecto se ha desarrollado a petición de la empresa tecnológica IBM.
El proyecto ha implementado un producto	SI
El proyecto ha consistido en el desarrollo de una investigación o innovación	NO
Objetivo general del proyecto	Crear un modelo de Deep Learning que sea capaz de asociar automáticamente los logos que se le pase a su respectiva marca o empresa

Índice

Índice de Figuras

Universidad Europea

Índice de códigos

Índice de Tablas

Capítulo 1. RESUMEN DEL PROYECTO

La finalidad de este apartado es poner en contexto al lector, explicando de una manera mas minimizada cual es el planteamiento del problema, los objetivos finales del proyecto y el resultado final obtenido.

1.1 Contexto y justificación

El proyecto se enfoca con la intención de resolver un problema que se nos ha planteado por parte de IBM, empresa tecnológica que se define como "*una empresa que combina su tecnología y conocimientos para intentar dar solución a diversos problemas que afectan a la sociedad*",

IBM es una de las únicas empresas de tecnología que tiene historia continua desde el siglo XIX, y es una empresa que trabaja para crear y comercializar tanto con hardware como con software, ofreciendo todo tipo de servicios (SaaS, PaaS, IaaS, consultoría, …).

1.2 Planteamiento del problema

El problema que nos plantea IBM reside en la cantidad de imágenes con logos de marcas que tiene la empresa sin ordenar ni catalogar. Para ello se requiere de algún tipo de desarrollo con inteligencia artificial con el que se puedan clasificar dichas imágenes de una manera rápida y automática.

1.3 Objetivos del proyecto

El objetivo principal del proyecto es desarrollar un modelo de clasificación de imágenes que se base en Deep Learning y mediante el cual se pueda ordenar y catalogar en marcas un dataset desordenado de imágenes dependiendo del logo que contenga cada una estas imágenes.

1.4 Resultados obtenidos

El resultado de la solución es un **modelo de clasificación de imágenes** basado en Deep Learning que, tal y como dice su nombre, clasifique automáticamente los logos del dataset en sus correspondientes marcas.

1.5 Estructura de la memoria

Mediante este apartado se pretende explicar la estructura de la memoria y qué se explica en cada uno de los apartados que contiene.

1.5.1 Resumen del proyecto

En este apartado se pretender dar al lector una idea general de lo que va a tratar el proyecto y como se pretende llevar a cabo.

1.5.2 Antecedentes y estado del arte

Este apartado se va a dividir en tres subapartados para el mejor entendimiento del problema al que se le pretende dar una solución mediante el desarrollo de este proyecto.

Primero, mediante el estado del arte se pretende buscar bibliografía que coincida con la idea de proyecto y explicar cual es el estado hasta la actualidad de las soluciones que se le dan a los problemas parecidos al planteado en este proyecto.

En segundo lugar, mediante el contexto, tal y como describe el título, pretende poner en contexto al lector, es decir, a quién le sucede el problema, en qué sector de la tecnología se aplica y una pequeña explicación de dicho sector.

Y, finalmente, mediante el planteamiento del problema, se explica el problema al cual se le va a dar una solución mediante el desarrollo del proyecto.

1.5.3 Objetivos del proyecto

En este apartado se especifican el objetivo principal del proyecto y los objetivos más específicos a los cuales se pretenden llegar, además de los beneficios que tendrá la realización del proyecto.

1.5.4 Desarrollo del proyecto

Este apartado es el más extenso de todos y se va a dividir en varios apartados:

- **Planificación del proyecto**. En este apartado se da una visión de cual ha sido la planificación de todo el proyecto durante los 3 meses de toma de requisitos y desarrollo.
- **Historia de la inteligencia artificial y el Machine Learning**. En este apartado se explica por encima la historia de la inteligencia artificial y el Machine Learning hasta la actualidad, destacando únicamente los puntos más importantes de la historia para no sobrecargarlo.
- **Explicación detallada del Deep Learning**. Mediante este apartado se pretende documentar al lector sobre el funcionamiento de las técnicas de inteligencia artificial que se van a usar para dar una solución al problema planteado.
- **Descripción de la solución**. En este apartado se van a describir cada uno de los modelos que se han desarrollado durante el proyecto. La explicación se realizará paso a paso para que se entienda todo a la perfección.
  Al final del punto se explicará detalladamente cuál de los modelos ha sido el elegido y los motivos de su elección.
- **Metodología usada**. En este apartado se pretende explicar cuál ha sido la metodología usada para el desarrollo del proyecto.

1.5.5 Conclusiones

En este apartado se muestran las conclusiones del trabajo realizado, basándose en las ideas iniciales y las implementadas. También se describen los conocimientos adquiridos durante el desarrollo del proyecto.

1.5.6 Futuras líneas de trabajo

En este apartado se toman en cuenta desarrollos que han quedado pendientes de hacer o ideas nuevas que se tienen pensadas para implementar en un futuro.

1.5.7 Bibliografía

Este apartado está dedicado a la lista de las referencias usadas en todo el documento.

Capítulo 2. ANTECENDENTES / ESTADO DEL ARTE

2.1 Estado del arte

Como primer paso, se ha realizado una búsqueda sobre trabajos existentes sobre el mismo problema planteado en este proyecto y que pueden servir como referentes para su desarrollo.

1. ***Inception V3*** (Advanced Guide to Inception v3 on Cloud TPU , s.f.).
   Inception V3 es un modelo de reconocimiento de imágenes desarrollado bajo el dataset de ImageNet, obteniendo más del 78 % de exactitud en él.
   El proyecto está desarrollado por Google y es el fruto de años de investigación con el fin de reformular la arquitectura de la primera versión del modelo.

2. ***Hybrid intelligent techniques for MRI brain images classification*** (El-Sayed A. El-Dahshan).
   Consiste en un modelo de clasificación de imágenes de resonancias magnéticas con un 97% de éxito usando la técnica de *back-propagation*, y un 98% usando el *k-nearest neighbor* (k-NN).

3. ***Classification of remotely sensed images*** (Ian L. Thomas, 2008)***.***
   Consiste en un proyecto de clasificación de imágenes para introducir, a los usuarios interesados en las imágenes de teledetección, a nuevos métodos de interpretación y nuevos sistemas de análisis.
   La idea general del proyecto es mostrar la integración de estas nuevas técnicas a las ya existentes.

4. ***Classification of Fingerprint Images*** (Lin Hong)
   Consiste en un proyecto de clasificación de imágenes cuyo fin es crear un modelo con el que se puedan detectar los distintos tipos de huellas dactilares existentes (arch, tented arch, left loop, …).

5. ***Automatic skin cancer images classification*** (Elgamal, Automatic Skin Cancer Images Classification, 2013)
   Consistge en un proyecto de clasificación de imágenes cuyo fin es detectar la presencia de cancer en la piel mediante imágenes.

Como se puede observar, existen montones de proyectos que aplican técnicas de Deep Learning para la solución de problemas mediante imágenes.

Muchos de ellos tienen varios años, por lo que en este proyecto se procurarán actualizar las técnicas de aprendizaje automático utilizadas.

2.2 Contexto y justificación

IBM es una empresa que se dedica a la **fabricación** y **comercialización** de **hardware** y **software**, además de ofrecer **soluciones cloud** y **servicio de consultoría** en un amplio conjunto de areas relacionadas con la informática.

La empresa se fundó en 1911, por lo que es una de las empresas mas longevas en el sector de la tecnología y una de las más grandes a nivel mundial, posicionándose en el puesto 53 de la lista Forbes que recoge las compañías mas grandes de la actualidad.

Además, dentro de los campos en los que trabaja, la **inteligencia artificial** es uno en los que más énfasis le pone debido a la cantidad de problemas que se les plantea y la cantidad de soluciones posibles que se les puede dar, además del auge que tiene actualmente.

Uno de esos problemas a los que IBM se enfrenta, y por el cual se ha empezado a implementar más y más la inteligencia artificial, es la cantidad de datos existentes y la cantidad de información que se puede sacar de ellos. Estos datos pueden ser documentos de texto, vídeos, archivos de audio o imágenes, los cuales son procesados mediante la técnica **Big Data**, que es la encargada de procesar cantidades enormes de datos en poco tiempo.

El término Big Data viene de años atrás, para ser mas exactos de finales de los años 90, en los que se produjo el *boom* tecnológico, haciendo aparecer grandes avances en internet, telefonía móvil y redes de conexión.

Aunque se aplica desde hace poco mas de dos décadas, el proceso al que se refiere el término de Big Data viene de mucho más atrás, cuando no era un proceso tan automatizado, pero para las necesidades que había y la cantidad de datos existentes se podía considerar como tal.

Como ya se ha dicho, la técnica de **Big Data** consiste en **la extracción, análisis y transformación de grandes cantidades de datos** para su posterior almacenamiento y procesamiento, y se considera la predecesora de la inteligencia artificial, ya que es necesaria su implementación para conseguir unos buenos datos (datos bien procesados) y así poder crear un modelo de inteligencia artificial fiable y preciso.

2.3 Planteamiento del problema

La inteligencia artificial se puede aplicar de distintas maneras dependiendo de las necesidades de cada situación, es decir, no se va a aplicar la misma inteligencia artificial cuando la intención es replicar la actuación de un pianista, que cuando se pretende contestar a una pregunta realizada por un usuario en una web.

En muchas ocasiones se puede dar el caso en el que la solución que se le da a un problema planteado puede ser muy lenta y costosa debido a la repetición en bucle del proceso, y es por eso por lo que surgen problemas como la clasificación manual de imágenes, donde la clasificación individual es sencilla pero el proceso en sí es largo y tedioso.

Si dicho proceso se mira con una perspectiva más empresarial, dentro de una empresa en la que se tiene la necesidad de clasificar miles y miles de imágenes, el simple hecho de contratar a

personas para la realización de la tarea conllevaría unos gastos elevados e innecesarios para la empresa.

Con este ejemplo se explica perfectamente el problema planteado por IBM para el desarrollo de este proyecto, en el que se da a entender que mediante el desarrollo de un programa o modelo de inteligencia artificial, se podría monitorizar el proceso de clasificación para conseguir ahorrar tanto tiempo como dinero, dependiendo de las necesidades que se tengan en el momento.

En el caso de este proyecto, se nos va a proporcionar un dataset con imágenes de logos de marcas con el fin de desarrollar un modelo capaz de clasificar futuras imágenes dependiendo del logo que contengan, y de esta manera conseguir catalogar un dataset completo de imágenes desordenadas.

Capítulo 3. OBJETIVOS DEL PROYECTO

3.1 Objetivos principales

El objetivo general del proyecto es desarrollar un modelo de clasificación de imágenes mediante el uso de la técnica del Deep Learning, que permita ordenar y catalogar un *dataset* o conjunto de imágenes desordenadas.

3.2 Objetivos específicos

A continuación se van a listar unos objetivos más específicos en los que se va a basar el desarrollo del proyecto:

- Estudiar y explicar la historia del Deep Learning.
- Implementar técnicas de Deep Learning para el desarrollo del modelo de IA.
- Indagar y elegir el mejor tipo de red neuronal existente para el modelo.
- Validación y mejora del modelo mediante técnicas de IA.
- Documentación y realización de pruebas con el modelo desarrollado.

3.3 Beneficios del proyecto

Gracias al desarrollo de este proyecto, el beneficiario principal será la empresa solicitante, que en este caso es IBM.

Aplicando el modelo desarrollado al dataset de imágenes sin clasificar conseguirán un **ahorro considerable de tiempo y dinero** gracias a que se podrá realizar de una manera totalmente auotmática y no será necesario el contrato de personal capacitado para realizar dicha tarea.

Además, otra de las ventajas consiguientes a la anterior mencionada es que, gracias a las nuevas imágenes clasificadas, se podrá **mejorar el modelo** para conseguir una mayor precisión y efectividad con futuras imágenes sin clasificar.

Personalmente hablando, los beneficios que se han conseguido tras el desarrollo de este proyecto han sido una **buen primer contacto** con el mundo de la inteligencia artificial y el **seguimiento de una buen línea de aprendizaje**, consiguiendo de esta manera unas buenas bases para la resolución de cualquier otro problema relacionado con el campo de la inteligencia artificial en el que se ha trabajado.

Capítulo 4. DESARROLLO DEL PROYECTO

El código y los archivos del proyecto se pueden encontrar en el siguiente repositorio:

https://github.com/adrisantos11/TFM_BigData

4.1 Planificación del proyecto

En este apartado se va a explicar la planificación que se tuvo del proyecto, teniendo en cuenta todas las fases mínimas a llevar a cabo y sabiendo la envergadura y duración del proyecto, que como se va a poder ver en el diagrama Gantt, no excede los 2 meses de duración.

Nombre actividad	Inicio	Final	Total días	16-ago	23-ago	06-sep	13-sep	20-sep	27-sep	04-oct	11-oct	17-oct
Estudio del problema	16/08/2021	18/08/2021	2									
Planteamiento de la solución	18/08/2021	22/08/2021	4									
Estudio del campo de la IA aplicado	16/08/2021	19/09/2021	34									
Desarrollo de la solución	23/08/2021	26/09/2021	34									
Pruebas	13/09/2021	10/10/2021	27									
Desarrollo de la memoria	16/08/2021	10/10/2021	55									
Entrenga final	--	17/10/2021	1									

Tiempo medio trabajado por día	1,5
Días totales de proyecto	157
Horas totales invertidas en el proyecto	235,5

Las actividades mencionadas dentro del diagrama son:

- **Estudio del problema**. Fase en la que se analiza el problema que se nos ha dado y cuales son los requisitos que se tienen que cumplir.
- **Planteamiento de la solución**. Una vez analizado el problema planteado, se piensa una solución a dicho problema, analizando todas las opciones posibles y escogiendo entre el listado de opciones la más válida.
- **Estudio del campo de la IA aplicado**. Al ser nuevo en el mundo de la inteligencia artificial, son necesarias mínimo 5 semanas para poner al día el conocimiento sobre el campo de la inteligencia artificial necesario para el desarrollo del proyecto.
- **Desarrollo de la solución**. Fase de desarrollo en la que se implementan en la solución los conocimientos que se van adquiriendo.
- **Pruebas**. Fase de pruebas del modelo. Tras tener finalizado el primer modelo, se prueban y desarrollan nuevos modelos en base a las métricas obtenidas, mejorando hiperparámetros, variables y datos del modelo antiguo.
- **Desarrollo de la memoria de trabajo**. Fase que avarca casi el 100% del tiempo del proyecto, ya que ha medida que se investigan y adquieren nuevos conocimientos, se va rellenando la memoria.
- **Entrega final**. Fecha final de entrega de documentación y solución.

4.2 Historia de la inteligencia artificial y el Machine Learning

La inteligencia artificial es un campo de la tecnología que ha ido cogiendo fuerza en la última década y que se puede definir como el **ámbito de la tecnología cuyo fin es crear programas y modelos capaces de replicar y automatizar las acciones humanas**.

Es un campo que puede aplicarse en muchos ámbitos distintos como por ejemplo el empresarial, el hogar, el educativo, etc., dando lugar a soluciones completamente diferentes.

También es importante recalcar el manejo de la ética, ya que es un aspecto de la inteligencia artificial que da mucho de qué hablar, debido a que, como ya se ha comentado, el fin de la inteligencia artificial es suplantar mediante máquinas inteligentes funciones que son realizadas por el ser humano.

Para saber cuando se comenzó a utilizar el término de inteligencia artificial, nos tenemos que remontar los años 50, más exactamente al 1956 en Dortmouth College (Estados Unidos), donde dicho término se definió como la "*ciencia de hacer máquinas inteligentes*".

Pero si nos vamos unos años más atrás, en 1950, Alan Turin propuso un test para saber si realmente las máquinas podían pensar cumpliéndose si al interactuar un humano con una máquina el humano es incapaz de reconocer si está interactuando con otro ser humano o con una máquina.

Avanzando dentro de la historia de la inteligencia artificial, se pueden encontrar varias fechas significativas [https://www.crehana.com/es/blog/desarrollo-web/historia-de-la-inteligencia-artificial]:

- **1936**. Introducción del concepto del **algoritmo**, por Alan Turin.
- **1941**. Se introducen, mediante el relato "Círculo vicioso" de Isaac Asimv, las 3 leyes de la robótica: ningún robot puede hacer daño a un ser humano, un robot siempre debe cumplir las órdenes de otro ser humano y un robot tiene que proteger su propia existencia a no ser que dicha regla entre en conflicto con las dos primeras mencionadas.
- **1950**. Turin propone el método conocido como **test de Turin**.
- **1952**. Arthur Samuel desarrolla el primer algoritmo capaz de aprender. Consistió en un algoritmo capaz de jugar a las damas, mejorando tras cada partida su manera de jugar.
- **1956**. Se define formalmente el término de inteligencia artificial, por John McCarthy, Marvin Minsky y Claude Shannon, como "*la ciencia e ingenia de hacer máquinas inteligentes, especialmente programas de cálculo inteligente.*".
- **1966**. Se crea Eliza, el primer chatbot del mundo.
- **1993**. Creación del primero robot con visión artificial.
- **1997**. La máquina Deep Blue de IBM gana por primera vez al campeón mundial de ajedrez.
- **2011**. Lanzamiento de los asistentes virtuales Siri, Cortana y Google Assistant.
- **2012**. Google lanza un sistema de identificación facial en videos de Youtube.

Todas estas fechas han sido significativas hasta el punto en el que se encuentra la inteligencia artificial actualmente, ya que ha ido avanzando hasta poder influir y ser muy importante en actividades fuera del ámbito académico.

La inteligencia puede ser aplicada de muchas maneras:

- Generación de lenguaje natural.
- Reconocimiento de voz.
- Agentes virtuales.
- Plataformas Machine Learning.
- Big Data.
- Biométricas.
- Defensa cibernética.
- …

Entre estas aplicaciones en este proyecto nos vamos a centrar en dos de ellas que son las de **plataformas Machine Learning** y el **Big Data**, aunque nos vayamos a centrar mucho más en el Machine Learning que en el Big Data.

El **Machine Learning** es una rama de la Inteligencia artificial que permite que las máquinas aprendan por si solas sin necesidad de que ese aprendizaje sea programado, es decir, que realicen **aprendizaje automático**.

A lo largo de la historia, desde la "creación" de la inteligencia artificial, el Machine Learning a sufrido dos grandes decandencias. La primera se produjo tras el desarrollo del algoritmo de "Nearest Neighbor" (primer algoritmo de reconocimiento de patrones), durante la década de los años 70, debido a la gran inversión producida en este campo y los pocos avances que se produjeron.

Posteriormente, en los años 80 surgieron los primeros sistemas expertos, lo que volvió a generar un gran interés por el Machine Learning, pero volvió a producirse una gran decadencia hasta principios del siglo XXI, cuando apareció el paradigma de ***Map-reduce***, desarrollado por Google.

Mas adelante, en 2006, Google crea la primera **base de datos de Big Data NoSQL**, y el año se culmina con la creación de la primera plataforma ***Big Data Open*** Source llamada ***Hadoop***.

Y aquí es donde entra la relación entre el Machine Learning y el Big Data, ya que gracias a la cantidad de datos existentes, es decir, la existencia de Big Data y aplicandole el procesamiento mediante Hadoop, permitió al Machine Learning desarrollarse de una manera increíblemente rápida.

Toda su evolución llega hasta la actualidad, donde se encuentra en su **tercera gran explosión**, aplicándose en muchísimos más ámbitos como marketing, negocio, …

4.3 Explicación detallada del Deep Learning

En esta sección se va a realizar un pequeño estudio sobre el funcionamiento del Deep Lerning y los componentes que lo conforman, además de la conexión entre ellos y los tipos de operaciones que se pueden aplicar mediante su uso.

4.3.1 Redes neuronales: ¿cómo funcionan?

Una red neuronal es un modelo que intenta **simular el funcionamiento del cerebro humano** mediante neuronas artificiales programadas que se conectan y transmiten información entre ellas.

La idea de las redes neuronales es conseguir que **aprendan por sí solas** y de esta forma conseguir realizar tareas que solo pueden ser realizadas por los seres humanos y no pueden ser reproducidas mediante la programación convencional.

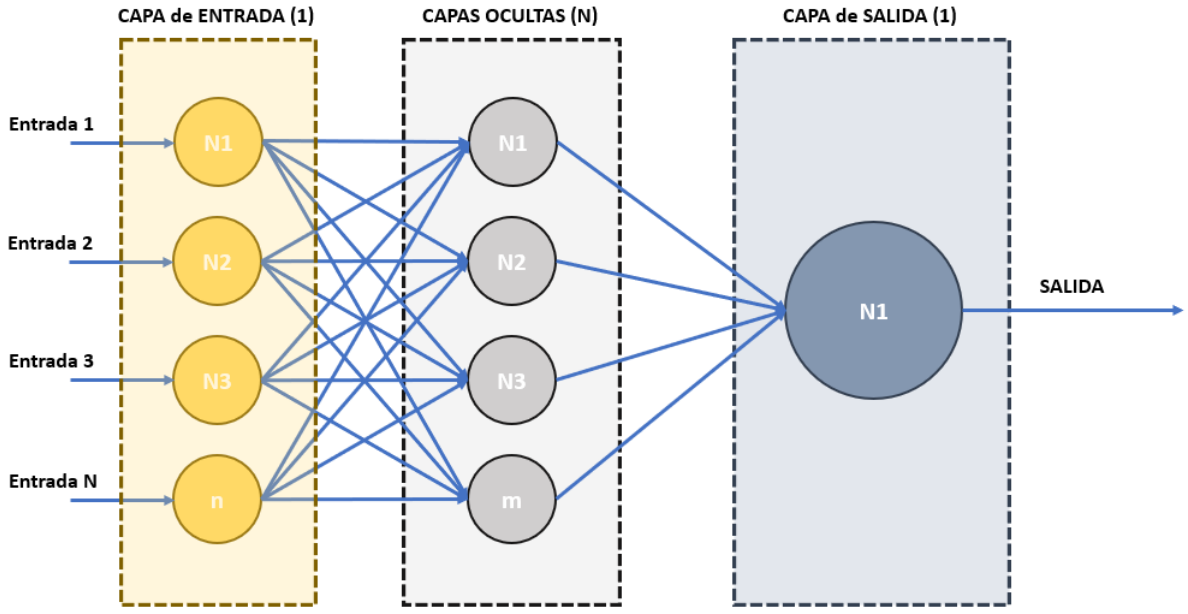


Figura 1. Estructura de una red neuronal

Como se muestra en la imagen, la red neuronal se divide por capas, pudiendo encontrar 3 diferentes:

- La **capa de entrada**, en la que se encuentran las primeras neuronas encargadas de recibir los datos objeto de estudio.
- Las **capas ocultas**, que están constituidas por más grupos de neuronas que realizan sus respectivas operaciones sobre los datos recibidos de la anterior capa.
- La **capa de salida**, que es la encargada de realizar la última operación que equivale a la predicción de la red neuronal completa.

Como ya se ha dicho, las redes neuronales se basan en la estructura del cerebro humano, es decir, en cómo se conectan y funcionan las neuronas entre sí. Pero ¿qué es una neurona y cómo funciona? Se explica detalladamente en el siguiente apartado.

4.3.2 La unidad básica de la red neuronal: la neurona

Una neurona es la **unidad básica de procesamiento** que se encuentra en una red neuronal, y su funcionamiento generalmente es como el representado en la *Figura 2. Funcionamiento de una neurona*.

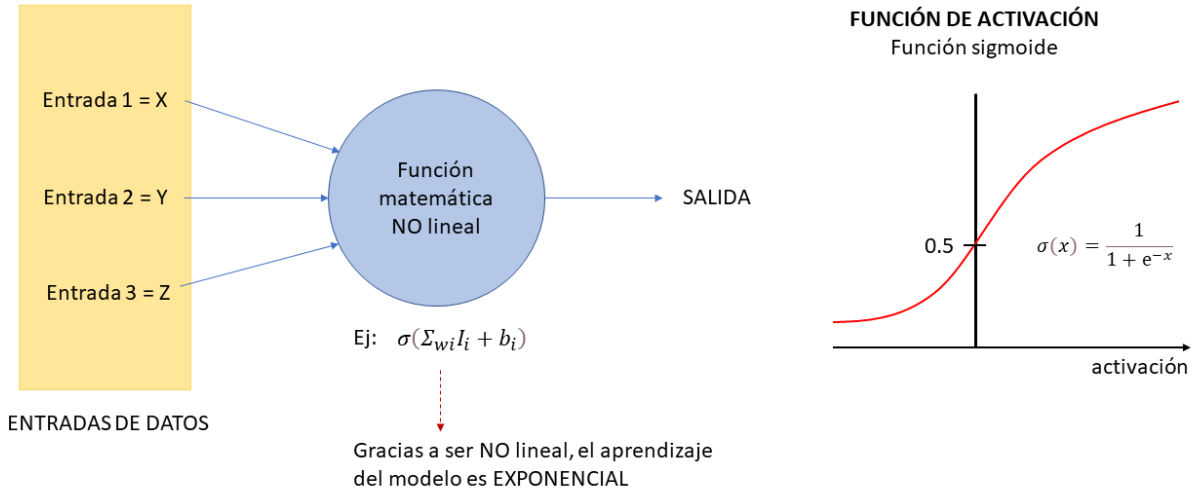


Figura 2. Funcionamiento de una neurona

Según la *Figura 2. Funcionamiento de una neurona*, se describe el proceso completo desde que la neurona recibe los datos de entrada hasta que los procesa y los manda por la salida. De este modo se pueden detectar 3 pasos:

PASO 1. Obtención de datos de entrada.

La neurona recibe una serie de estímulos externos, o en este caso, **valores de entrada** los cuales va a utilizar para realizar una suma ponderada de todos ellos. Esta suma ponderada se puede interpretar a su vez como una función, la cual se utilizará para obtener la salida de la neurona.

PASO 2. Aplicación de una ecuación no lineal.

Cada uno de los valores de entrada tiene un **peso** o ***weigth (w)***, que describe la **importancia** o **intensidad** con la que afectará cada uno de los datos dentro de la función de la neurona y los cuales se suelen introducir a través de hiperparámetro en el modelo que se esté usando.

Para que se entienda mejor, en la imagen tenemos 3 valores de entrada: X, Y y Z, que, multiplicándolos mediante sus respectivos pesos y realizando una suma ponderada, daría lugar a la siguiente función:

$$Y = w_1 X + w_2 Y + w_3 Z$$

Analizándola se puede decir que la función es parecida a una **regresión lineal múltiple** ($y = w_0 + w_1 x_1 + w_2 x_2$), por lo que se llega a la conclusión de que una neurona hace internamente un **modelo de regresión lineal**.

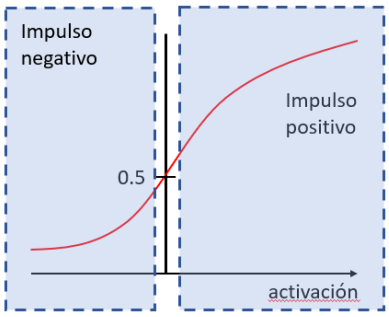


Figura 3. Ejemplo de función de activación de una neurona

A la función neuronal falta añadirle el valor correspondiente al w_0 de la regresión lineal, que es el término que permite mover verticalmente la recta. Este término dentro de la función de la neurona corresponde a la *b*, también denominado **sesgo de la neurona** (*bias)*, que es el dato que controla la predisposición de una neurona de devolver un 0 o un 1.

La función final que queda es la siguiente:

$$y = \sum_{n=0} w_n + b$$

Este parámetro, al igual que los pesos de los datos de entrada, se establece dentro del modelo mediante un hiperparámetro.

La función que va a utilizar la neurona es muy importante que sea **no lineal**, ya que sabiendo que va a estar dentro de una red neuronal, si aplicamos una función de regresión lineal durante toda la red, el resultado obtenido por la última neurona de la red será exactamente el mismo al resultado obtenido por la primera.

Esto se puede explicar matemáticamente hablando, ya que el hecho de sumar muchas operaciones de regresión lineal (líneas rectas) equivale a realizar una única operación, dando lugar a una **línea recta**.

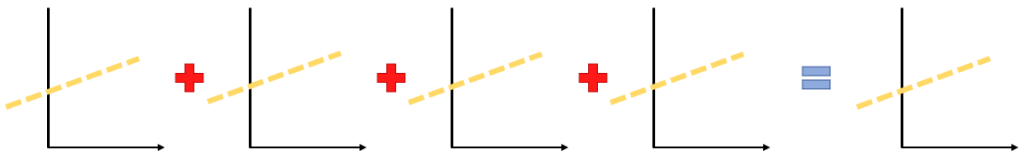


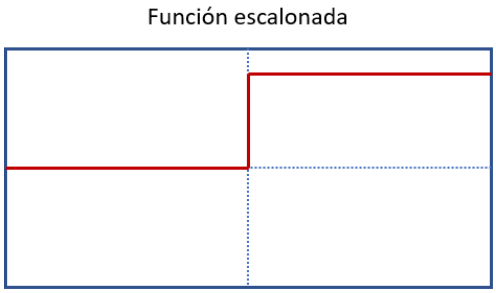
Figura 4. Resultado de red neuronal sin función de activación

Para que el resultado final no sea el mismo que el obtenido por la primera neurona, es necesario que las líneas producidas por las funciones de cada una de las neuronas sufran alguna **deformidad** o **manipulación** con la finalidad de conseguir una función no lineal, y de esta forma obtener un resultado final completamente diferente al obtenido por las primeras neuronas.

Para conseguir esta "manipulación", se aplican las **funciones de activación**, cuya función principal es distorsionar el valor de salida de la función lineal de la neurona.

La manera de distorsionar la recta depende del tipo de función de activación que se use. Existen varias funciones de activación:

- **Función de activación escalonada.**
  Esta función trabaja con un umbral, y dependiendo del umbral y el valor del dato obtenido por la función lineal, se le aplica 0 o 1.

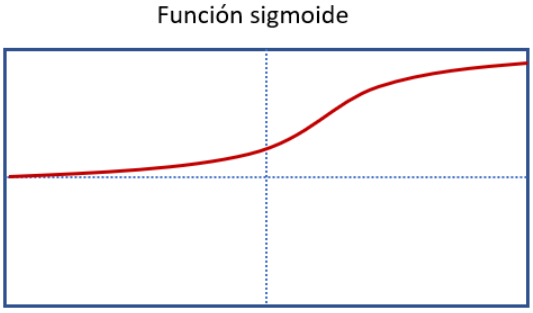


$$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$$

Figura 5. Función de activación escalonada

Se llamada escalonada porque el cambio de valor es **instantáneo** y **no gradual**.

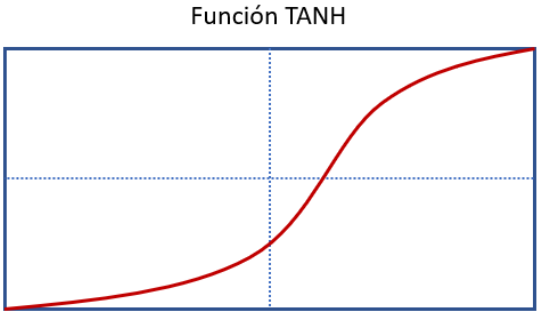
- **Función de activación sigmoide.**
  Es una función que además de permitir aplicar la deformación buscada, es una función que se basa en la **probabilidad**, ya que sus datos se encuentran entre 0 y 1.



$$f(x) = \sigma(x) = \frac{1}{1 + e^{-x}}$$

Figura 6. Función de activación sigmoide

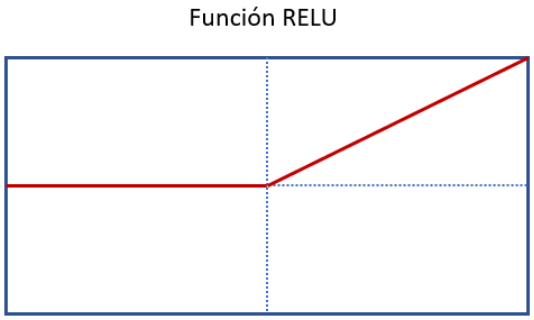
- **Función de activación tangente-hiperbólica (TANH).**
  Es igual que la sigmoide pero los datos que abarca se encuentran entre -1 y 1.



$$f(x) = \tanh(x) = \frac{(e^{x} - e^{-x})}{(e^{x} + e^{-x})}$$

Figura 7. Función de activación TANH

- **Función de activación rectificada-lineal (RELU).**
  Es una función que toma como valor 0 cuando el dato de entrada es negativo y como una función lineal si es positivo.



$$f(x) = \begin{cases} 0 & \text{for x} < 0 \\ x & \text{for x} \geq 0 \end{cases}$$

Figura 8. Función de activación RELU

Aplicando la deformación producida por cualquiera de las funciones de activación mencionadas se consigue solucionar el problema de **no poder encadenar varias neuronas sin conseguir resultados diferentes**.

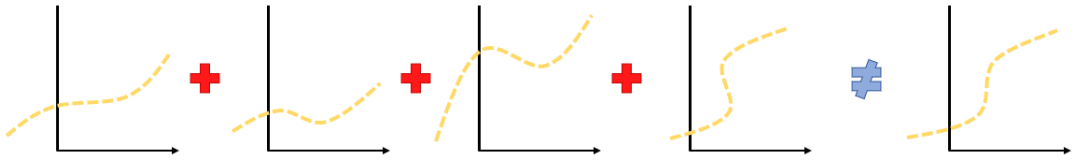


Figura 9. Resultado de red neuronal con función de activación

La función resultante a todo este proceso se forma aplicando la función de activación al resultado de la regresión lineal de la neurona, es decir, se le aplicala función de activación al sumatorio de los pesos de las conexiones con las neuronas de la anterior capa más el sesgo:

$$F = \sigma(y) = \sigma\left(\sum_{n=0} w_n + b\right)$$

PASO 3. Salida que produce la neurona.

El resultado que devuelve la neurona puede ser un **impulso negativo o positivo**, dependiendo en qué parte de la gráfica se encuentre.

4.3.3 Aprendizaje automático dentro de una red neuronal

A continuación se va a detallar cómo una red neuronal aprende y actualiza el peso de sus conexiones para mejorar automáticamente la respuesta que produce.

Para ello es necesario saber que son las técnicas del **descenso del gradiente**, la cual es la más utilizada actualmente para el aprendizaje automático, la **función de coste**, que indica cómo se está comportando la red neuronal en cada ciclo de aprendizaje, y finalmente el **algoritmo de**

back-propagation, que es el algoritmo que se encarga de la actualización de los pesos dentro de la red mediante el cálculo de las derivadas parciales de la función de coste, pesos y sesgos.

4.3.3.1 Descenso del gradiente

El **descenso del gradiente** es en un algoritmo que se usa dentro del Machine Learning para **entrenar las redes neuronales**. Permite a la red neuronal acercarse más y más al resultado esperado por salida, disminuyendo la desviación a medida que se va aplicando recursivamente.

Existen 3 técnicas diferentes del descenso del gradiente (Durán, 2019):

- **Descenso del gradiente en lotes.**
  En esta opción se introducen directamente **todos los datos disponibles** de una vez (en una sola iteración).
  Aplicando esta técnica de descenso del gradiente nos encontraremos problemas como el estancamiento, falta de memoria de procesamiento, no es eficaz cuando los datos que se usan son muy similares entre ellos, etc.

- **Descenso del gradiente estocástico (aleatorio).**
  A diferencia del anterior, en este caso se utiliza **una muestra única aleatoria para cada iteración**. Esto permitirá introducir una aleatoriedad más robusta al modelo, pero al aplicarse con una única muestra, el proceso requerirá de un mayor nº de iteraciones para llegar al resultado esperado.

- **Descenso del gradiente en mini-lotes.**
  Es una mezlca de las dos técnicas anteriores, ya que en vez de introducir directamente todos los datos disponibles o aplicar una única muestra por cada iteración, lo que se realizan son **grupos de N muestras para introducir en cada iteración**.
  De esta forma se consigue las ventajas de ambas técnicas, se mantiene la aleatoriedad robusta pero añadiendo un entrenamiento más rápido debido al pararelismo de operaciones.
  La idea final de esta técnica es elegir el valor de N que nos aporte mas balance entre la aleatoriedad y la rapidez de entrenamiento.

Analizando todas las técnicas disponibles, la más recomendable de usar es sin duda la tercera opción mencionada, **el descenso del gradiente por mini-lotes** debido a ese paralelismo de operaciones que junta la rapidez y la aleatoriedad robusta.

De esta forma, habiendo elegido ya la técnica, las iteraciones que va a llevar a cabo el algoritmo de descenso del gradiente son las siguientes (Durán, 2019):

1. En primer lugar se introduce un lote de **N muestras aleatorias y etiquetadas** (es decir, sabiendo la clase a la que pertenecen) extraídas del conjunto de datos de entrenamiento.

2. Tras todas las operaciones realizadas por las capas de la neurona se obtiene el resultado de las predicciones de salida. A este paso se le denomina ***forward propagation***, debido a que el proceso de entrenamiento se está realizando desde la primera capa a la última.

3. Se realiza una **evaluación de la función de coste o pérdida** para el lote de muestras que se ha cogido.
   Esta función se ha elegido previamente en el proceso de análisis ya que dependiendo del problema al que nos enfrentemos se elige una u otra.
   La idea de la repetición del algoritmo es minimizar al máximo la función de coste.
   Las funciones de coste más comunes son las siguientes:

Tipo del problema	Tipo de salida	Función de activación utilizada	Función de coste o pérdida
Regresión lineal	Numérico	Linal	Mean Squared Error (MSE)
Clasificación	Salida binaria	Sigmoide	Binary Cross Entropy
Clasificación	Label única, clases múltiples	Softmax	Cross Entropy
Clasificación	Label múltiple, clases múltiples	Sigmoid	Binary Cross Entropy

4. Obtenemos el **gradiente** calculando la **derivada multivariable** de la función de coste respecto al resto de los parámetros de la red neuronal.

   Respresentada gráficamente corresponde a la pendiente de la tangente de la función en el punto en el que nos encontramos dentro de la gráfica, y matemáticamente se corresponde a un vector que nos índica la dirección y el sentido hacia dónde la función aumenta su valor más rápidamente, esto quiere decir que si queremos ir disminuyendo el valor, habría que calcular la pendiente contraria a dicho vector.

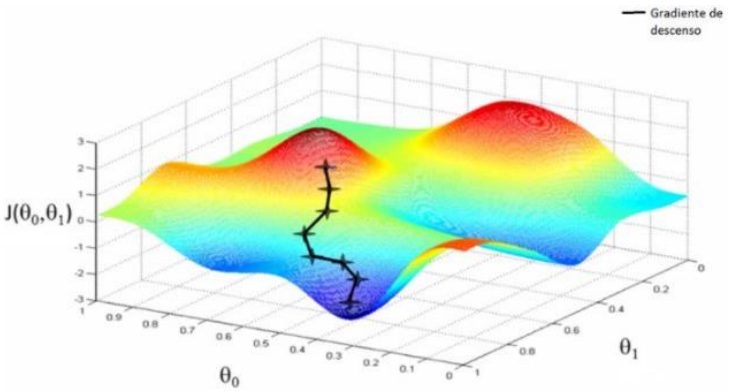


Figura 10. Visualización del funcionamiento descenso del gradiente

   Debido a la dificultad de operación de las derivadas y la cantidad de parámetros y dimensiones, se aplica el algoritmo de **back-propagation** para llegar a la solución de la derivada multivariable. Este algoritmo se explicará más adelante en el punto 4.3.3.3, pero para adelantar, consiste en un algoritmo que coge como entrada la función final de coste de

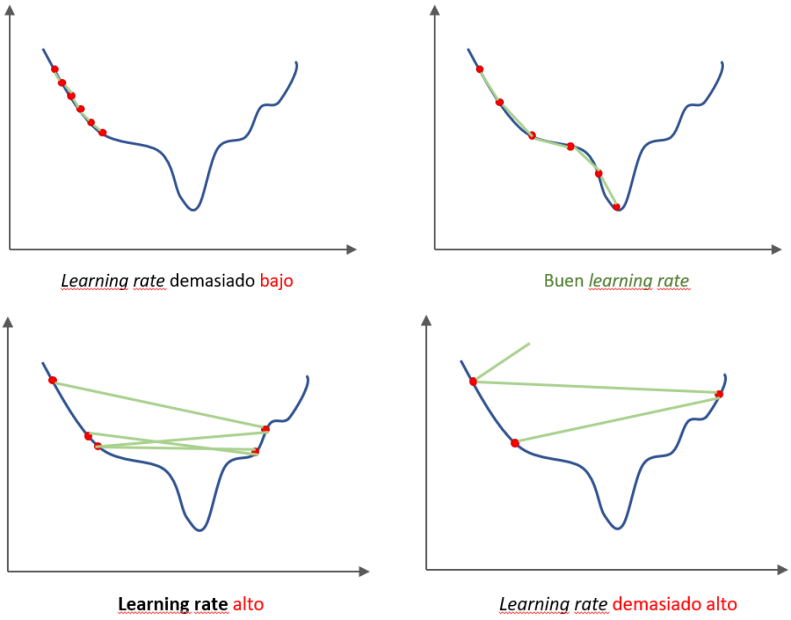
la red neuronal y va aplicando derivadas parciales según va retrocediendo capas dentro de la red, visualizando la influencia de cada neurona dentro del valor de coste obtenido y modificando los pesos respectivamente.

5. Finalmente, obtenido el vector del gradiente, **actualizamos** el valor del gradiente actual **restándole** el obtenido y **multiplicándolo** por un **tasa de aprendizaje** que permite ajustar la magnitud de los pesos. El gradiente se resta debido a que queremos ir disminuyendo la función de coste y acercarla a un valor lo más próximo a 0.

Este proceso se repite una y otra vez hasta que los resultados de la función de coste empiecen a mantenerse o a empeorar.

Es muy importante elegir correctamente la **tasa de aprendizaje** para evitar alcanzar mínimos locales (no global) o puntos de silla (mínimo local de una zona pero máximo de otra). Para ello se pueden tener en cuenta las siguientes premisas (Hammel, 2019):

- **El valor de la tasa de aprendizaje**. Es muy importante que el valor de la tasa de aprendizaje no sea ni muy alto, lo que provoca oscilaciones, ni muy bajo, ya que provoca lentitud y más iteraciones.



- El **tipo de valor**, es decir, si es **fijo** o **variable** en el tiempo. Un ejemplo de una tasa de aprendizaje variable se produce cuando se pretende disminuir la tasa de aprendizaje a medida que se va acercando al mínimo de la función de coste, para que poco a poco sea más precisa la estimación.
- **Variabilidad de la tasa de aprendizaje**, es decir, que en distintas capas de la red neuronal la tasa de aprendizaje sea completamente diferente. Esto se puede dar cuando tenemos un modelo ya pre-entrenado al que le queremos añadir más capas. La tasa de aprendizaje será menor en la parte del modelo ya entrenada, y será mayor en las nuevas capas que le añadamos.

4.3.3.2 Función de coste o función de pérdida

Tal y como se ha descrito, la función de coste es la función encargada de **obtener el error entre el valor estimado y el valor real** con el fin de obtener el valor de error y de esta manera optimizar la red neuronal.

Para la función de coste se pueden usar lo que se denominan **estimadores**, con el fin de conseguir estimar el error obtenido en ese ciclo de aprendizaje.

Entre los estimadores podemos encontrar:

- **Error cuadrático medio – ECM**
  Consiste en el sumatorio del cuadrado de la diferencia entre el valor real a devolver y el valor devuelto por el modelo.

$$ECM = \frac{1}{M}\sum_{i=1}^{M}(v_i - p_i)^2$$

  Donde **M** es el número de muestras totales, **v** el valor real y **p** el valor predicho por el modelo.

- **Raiz cuadrada media – RMSE**
  Consiste en la raíz cuadrada de la media de la diferencia entre el valor correcto y el valor obtenido.
  Es útil para **eliminar valores extremos** dentro del conjunto de datos y se utiliza mucho para **optimizar regresiones**.

$$RMSE = \sqrt{\sum_{i=1}^{n}\frac{(\hat{y}_i - y_i)^2}{n}}$$

- **Error absoluto medio – MAE**
  Tal y como dice su nombre, es una medida de precisión para obtener el **valor medio absoluto del sumatorio de los errores**.

$$MAE = \sum_{i=1}^{n}\frac{|\hat{y}_i - y_i|}{n}$$

- **Entropía cruzada categórica – Categorical Cross-Entrophy**
  Es una medida de precisión para problemas en los que se usan **variables categóricas**.

$$\mathcal{L}(\theta) = -\frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{m} y_{ij}\log(p_{ij})$$

- **Entropía cruzada binaria – Binary Cross-Entrophy**
  Es una medida de precisión destinada a problemas que hacen uso de **variables binarias**.

$$\mathcal{L}(\theta) = -\frac{1}{n}\sum_{i=1}^{n} y_i \log(p_i + (1-y)\log(1-pi))$$

4.3.3.3 Algoritmo de back-propagation

El algoritmo de ***back-propagation*** es un algoritmo que recibe de entrada la función de coste y cuyo fin es recoger la influencia de las neuronas sobre el resultado de esa función de coste y así modificar los pesos de las conexiones para minimizar todo lo posible el error.

La aparición de el algoritmo de *back-propagation* se debe a que, tras el surgimiento de las redes neuronales con miles de neuronas y miles de conexiones, era imposible realizar la reestimación de los pesos y sesgos de cada una de ellas mediante la técnica de **perturbación aleatoria**, es decir, mediante "fuerza bruta". Esto se debe a que es necesario realizar una derivada parcial respecto a cada uno de los pesos de las conexiones entre las neuronas de la red, y hacerlo manualmente y a la fuerza bruta sería poco fiable y computacionalmente muy caro.

Lo primero y más importante de todo para poder entender a la perfección el algoritmo de back-propagation es familiarizarse con una serie de notaciones para la expresión de los pesos, la función de coste, …, es decir, la notación usada para representar todos los datos que se van a usar durante el proceso de "propagación hacia atrás" (Blanco, 2021).

Para la representación de los pesos de las conexiones entre neuronas utilizaremos lo siguiente:

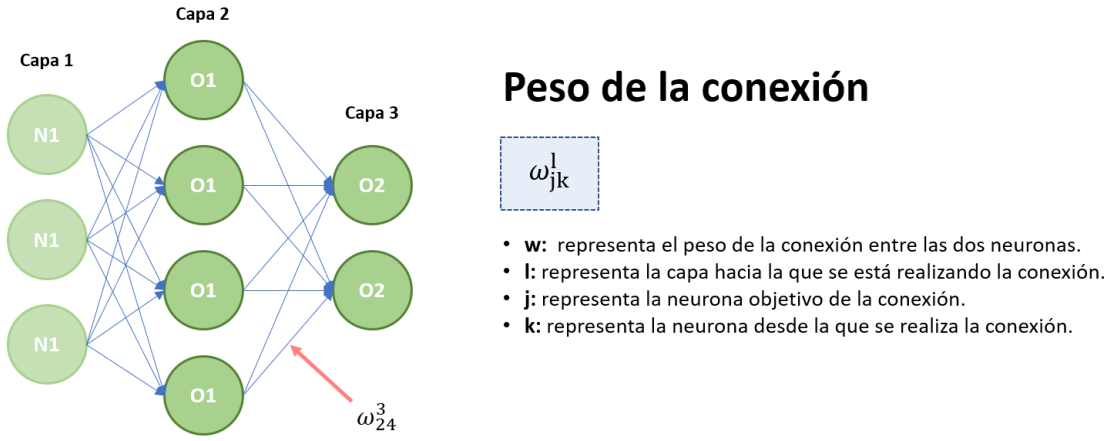
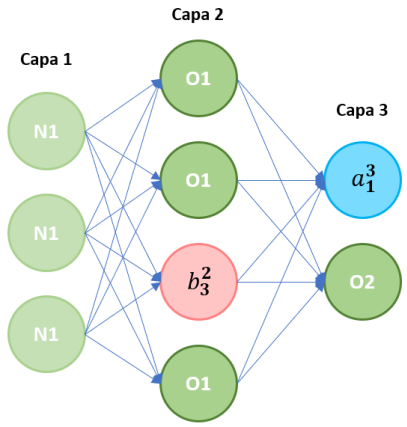


Figura 11. Anotación peso de conexión entre neuronas

Por lo que se puede decir que w_{jk}^{l} es el peso desde la K neurona en la capa L-1 hasta la J neurona de la capa L.



Sesgo de la neurona

b_j^l

- **l:** representa la capa.
- **j:** representa la posición de la neurona en la capa.

Función de activación

a_j^l

- **l:** representa la capa.
- **j:** representa la posición de la neurona en la capa.

Figura 12. Anotación para sesgos y función de activación de una neurona

Sabiendo esto, podríamos decir que el comportamiento de la segunda neurona de la tercera capa viene dado por el peso de los enlaces con respecto a las neuronas anteriores w_{21}^3, w_{22}^3, w_{23}^3 y w_{24}^3, por el bias de la neurona b_2^3 y la función de activación que aplica a_2^3.

Por lo que, ¿cómo se obtendrían las funciones parciales de la función de coste respecto a ambos parámetros (peso y bias)?

Para ello se realizan las siguientes operaciones, $\frac{\partial C_i}{\partial w}$ y $\frac{\partial C_i}{\partial b}$, donde en el **denominador** se encuentra la **función de coste** y en el **numerador** los parámetros **peso** y **bias**.

Ahora supongamos que tenemos la función de error cuadrático medio como función de coste, la cual ya se ha explicado en el punto 4.3.3.2 del documento:

$$ECM = \frac{1}{M}\sum_{i=1}^{M}(v_i - p_i)^2$$

En esta función sabemos que M es el número de muestras con las que está trabajando el modelo, v_i es el valor que se espera que devuelva y p_i es el valor que devuelve.

Ahora que sabemos qué función de coste estamos usando, se pueden realizar una serie de operaciones hasta entender el funcionamiento del algoritmo *back-propagation* (Chaos, Backpropagation | Interactive Chaos, s.f.):

1. Tal y como se ha explicado en el punto "*4.3.2 La unidad básica de la red nuronal: la neurona*", se sabe cómo y por qué se le debe aplicar una función de activación al funcionamiento de una neurona y cual sería el resultado de salida de dicha neurona, quedando de la siguiente manera:

$$p_i = \sigma(y) = \sigma\left(\sum_{n=0} w_n + b\right)$$

coincidiendo con el p_i de la función de coste.

2. Se **simplifica la función de coste sustituyendo los valores**, quedando como la función de coste final de una única neurona:

$$ECM = C_i = \left(v_i - \sigma \left(\sum_{n=0} w_n + b \right) \right) = v_i - \sigma(y)$$

3. Y aquí es donde entra el cálculo infinitesimal, que dice que la derivada de la función de coste con respecto a los pesos, $\frac{\partial C_i}{\partial w}$, coincide con el **producto** de las siguientes derivadas:
   - Derivada de la función de coste respecto de la función de activación, $\frac{\partial C_i}{\partial \sigma}$.
   - La derivada de la función de activación respecto a y, $\frac{\partial \sigma}{\partial y}$.
   - La derivada de la función y respecto a los pesos, $\frac{\partial y}{\partial w}$.

Quedando así de la siguiente forma:

$$\frac{\partial C_i}{\partial w} = \frac{\partial C_i}{\partial \sigma} * \frac{\partial \sigma}{\partial y} * \frac{\partial y}{\partial w}$$

4. Para finalizar, simplemente habría que **calcular las derivadas parciales** de la anterior operación y retroceder a la capa anterior para volver a repetir de proceso, utilizando el resultado de las derivadas parciales realizadas.

Después de haber explicado el proceso, se puede decir que el algoritmo de back-propagation es el que **permite calcular el gradiente en cualquier punto de la función de coste** y **minimizarlo** al máximo posible.

Ejemplo visual

Tenemos una neurona *fully-connected* cuyas posibles salidas son gato, perro, ratón y gallo.

El primer paso que realiza el algoritmo de *back-propagation* es "volver hacia atrás" a las neuoras de la capa anterior con el resultado de la función de coste memorizada. En este caso vamos a coger el ejemplo del gato.

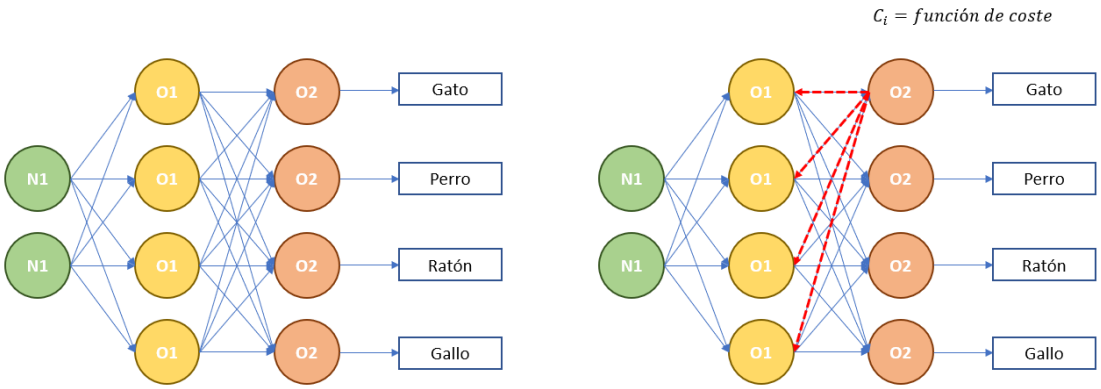


Figura 13. Proceso back-propagation 1

Y junto a la función de coste se obtiene la responsabilidad de cada neurona sobre el resultado final obtenido, es decir, lo que ha influido cada neurona en el error del modelo.

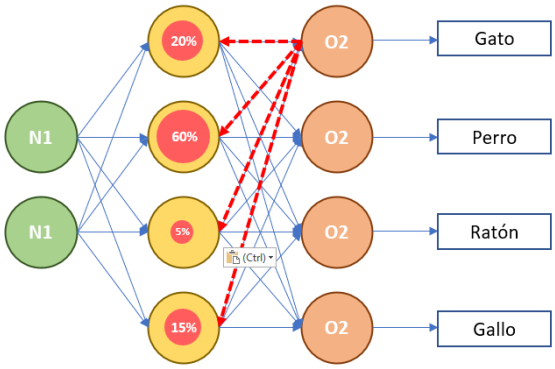


Figura 14. Proceso back-propagation 2

Obtenida la influencia de cada neurona en el error lo que se hará es, mediante ese nuevo dato, calcular los nuevos pesos de las conexiones a esa neurona, y al ser un proceso iterativo y recurrente, se puede tomar la capa siguiente como la última capa de la red.

4.3.4 Tipos de optimizadores

Los optimizadores son los encargados de actualizar los parámetros de los pesos de las conexiones entre las neuronas para disminuir el valor de la función de coste.

Entre los optimizadores más usados podemos encontrar (Calvo, 2018):

- **Descenso estocástico del gradiente – SGD (Stochastic Gradiente Descent).**
- **RMSProp.**
- **Adam.**
- **Algoritmo de gradiente adaptativo (Adagrad).**
- **Adadelta.**
- **Impulso.**
- **Gradiente acelerado de Nesterov (NAG).**
- **Estimación de momento adaptativo acelerada por Nadam-Nesterov.**

4.3.5 Tipos de redes neuronales

Actualmente se pueden aplicar diferentes redes neuronales dependiendo del tipo de problema que se pretende solucionar, y para elegir la correcta es necesario saber muy bien cómo se quiere plantear la solución y qué se espera.

Para ello, es necesario saber cuales son los tipos de redes existentes y qúe maneras de clasificarlas hay.

Las clasificaciones existentes para las redes neuronales son clasificación por número de capas, clasificación por el tipo de conexiones y clasificación por el grado de conexiones (Sossa, 2021).

- **Clasificación por número de capas.**

Dentro de esta clasificación se pueden encontrar dos grupos diferentes de redes neuronales:

  - **Redes neuronales de monocapa.** Son redes neuronales que contienen una única capa en la que se realizan las operaciones y se obtiene la salida. Hay que tener en cuenta que la capa de entrada puede considerarse otra capa pero en ella no se realiza ningún tipo de operación.
  - **Redes neuronales multicapa.** Entre la capa de entrada y de salida existen más capas en las que se producen operaciones. A estas capas se les denomina **capas ocultas**.

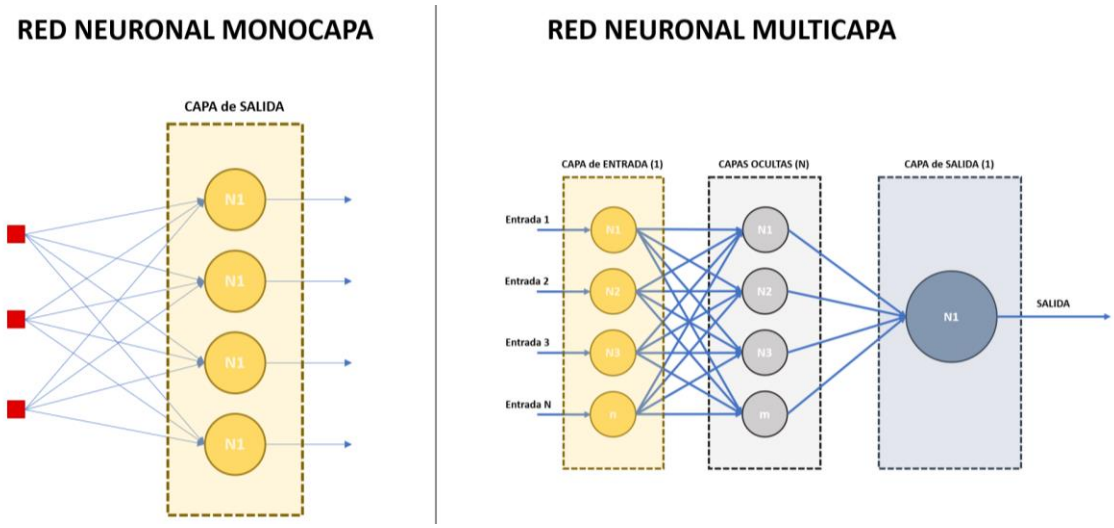


Figura 15. Redes monocapa y multicapa

- **Clasificación por el tipo de conexiones.**
  Este tipo de clasificación también nos puede ayudar a decidir que red neuronal elegir dependiendo del tipo de solución que se quiera obtener. Dentro de esta clasificación encontramos:
  - **Redes neuronales no recurrentes**. Se trata de redes neuronales que no tienen ningún tipo de retroalimentación y carecen de memoria, es decir, no pueden guardar ningún dato para una segunda iteración, lo que explica la primera carencia de retroalimentación.

- **Redes neuronales recurrentes**. A diferencia de las no recurrentes, si que permiten la realimentación entre neuronas, sean de la misma capa o de diferentes. La realimentación se produce debido a la memoria que contienen las neuronas.
  Son neuronas generalmente mas potentes y rápidas que las no recurrentes.

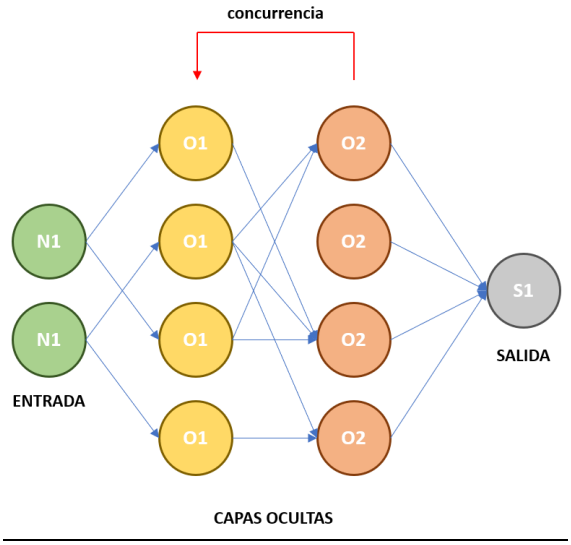


Figura 16. Red neuronal recurrente

- **Clasificación por el grado de conexiones.**
  Las redes neuronales también se pueden clasificar dependiendo de la cantidad de conexiones producidas entre sus neuronas:
  - **Redes neuronales completamente conectadas (*fully-connected)*.** En ellas, todas las neuronas de todas las capas están conectadas entre sí al 100%, es decir, todas las neuronas de una capa están conectadas con todas las neuronas de la capa anterior (en el caso de haber capa anterior) y de la siguiente.
  - **Redes neuronales parcialmente conectadas (*partially-connected*).** En ellas no existe un 100% de conexiones entre las neuronas de distintas capas.

  Conectar determinadas neuronas de una capa con determinadas neuronas de la siguiente hace que se creen **jerarquías** que trabajan en procesos diferentes.

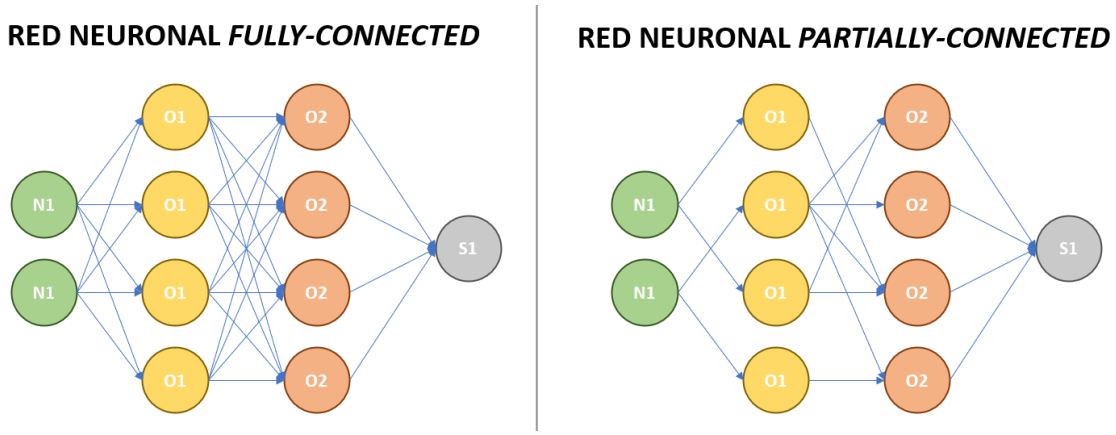


Figura 17. Redes neuronales fully-connected y partially-connected

También existen otros tipos de redes neuronales, que se pueden encontrar dentro de las clasificaciones anteriormente mencionadas y son los que se detallan a continuación:

- **Perceptrón multicapa.**
  También denominada *Multilayer Perceptron (MLP)*, se considera la arquitectura más simple de una red neuronal y es la que se muestra mediante todas las Figuras representativas del documento.
  En esta red tenemos una **capa de entrada** que no aplica ningún tipo de función a los datos, simplemente reciben los valores de entrada, **una o varias capas ocultas** cuyas neuronas están conectadas al 100% con las capas anterior y posterior, y una **capa de salida** (Chaos, El Perceptrón Multicapa, s.f.).

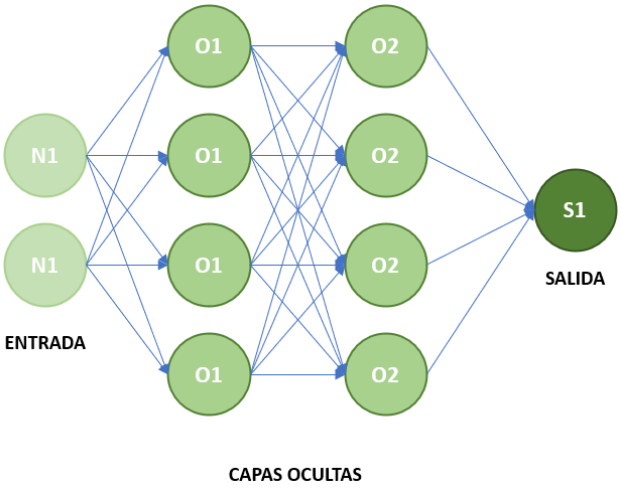


Figura 18. Perceptrón multicapa

- **Redes neuronales convolucionales.**
  Son un tipo de red neuronal que intenta simular el funcionamiento de las neuronas de la **corteza visual**. Son parecidas al perceptrón multicapa pero se diferencian en que su aplicación se basa en matrices bidimensionales, por lo que su utilización suele darse en tareas de visión artificial como la clasificación de imágenes.

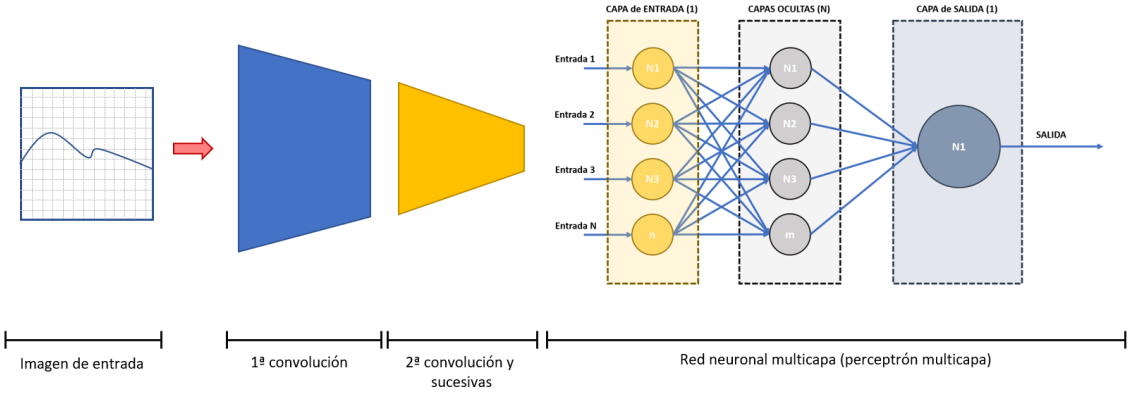


Ilustración 1. Proceso completo de convolución

- **Redes neuronales recurrentes.**
  Son un tipo de redes neuronales que se utilizan mayoritariamente para analizar datos de series temporales, considerando la variable del tiempo.
  Son redes neuronales en las que las neuronas tienen la capacidad de devolverse su propia respuesta, es decir, contienen una nueva conexión para retroalimentarse con su salida.
  Un ejemplo de una neurona de este tipo de red neuronal sería el expuesto en la Figura 19. Neurona recurrente.

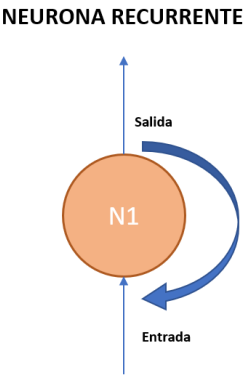


Figura 19. Neurona recurrente

Se considera una especie de recursividad, ya que en el instante *t*, la neurona recibe el valor de entrada normal y a su vez recibe también el valor de salida generado en el instante *t-1*.

Por lo que, mediante una línea temporal, el funcionamiento de esta neurona durante el tiempo es el siguiente:

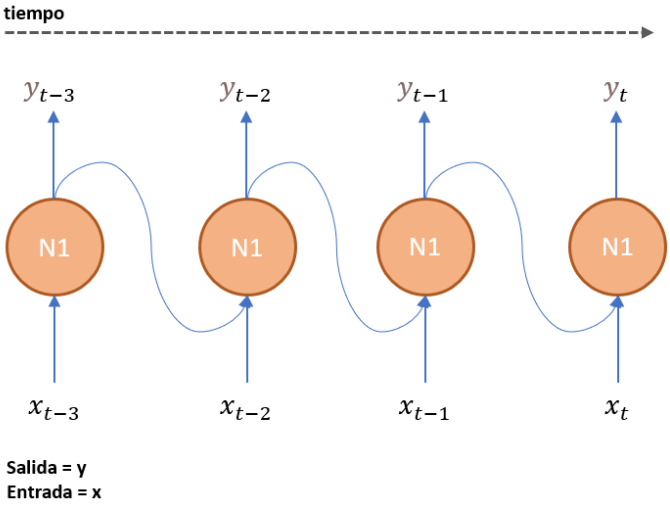


Figura 20. Funcionamiento de una neurona concurrente con el paso del tiempo

Es decir, en cada unidad de tiempo, cada neurona de una red neuronal recurrente puede recibir hasta dos parámetros de entrada: la salida de la neuora de la capa anterior y su misma salida de la anterior iteración (Torres, 2021).

- **Redes neuronales de base radial.**
  Son redes neuronales cuyas funciones de activación dependen de la distancia a un punto denominado **Centro**.

En nuestro caso, como el proyecto consiste en la creación de un modelo de clasificación de imágenes, se va a hacer uso de redes **neuronales convolucionales**, ya que son las mejores para crear soluciones en el tratamiento de imágenes.

4.3.6 Redes neuronales convolucionales

Para entender las redes neuronales convolucionales es necesario entender nuestro sistema visual, es decir, como percibimos y analizamos los estímulos externos a través de la vista.

Por ejemplo, si ponemos de fondo una imagen de un gato, nada mas visualizarla ya vamos a saber lo que es debido a un previo escaneo que hemos realizado. Con este escaneo hemos detectado ciertos elementos que, gracias al aprendizaje que hemos tenido durante toda la vida y la cantidad de animales que hemos visto, nos han hecho llegar a la conclusión de que es un gato en vez de un perro o un caballo.

Estos elementos que se han mencionado son las orejas puntiagudas, los bigotes, los ojos, …, que ha su vez hemos podido detectar gracias a otros elementos mas pequeños que los conforman, como por ejemplo en el caso del ojo: la forma de éste, el iris, la pupila, etc, y que, nuevamente, hemos podido diferenciar gracias a la capacidad de detectar patrones, colores y formas diferentes.

La idea de este ejemplo es entender el funcionamiento del córtex visual, el cual, primero identifica los elementos mas básicos (patrones, formas, colores, ...), a los que irá uniendo en las

siguientes capas para formar nuevos elementos más completos y que aportarán más información.

Una vez se sabe el funcionamiento general del córtex visual de un ser humano, entender cómo se describe el funcionamiento una **red neuronal convolucional** va a ser mucho más sencillo: es una red que puede recibir por entrada una imagen y, mediante mapas de características, pesos y sesgos, obtener objetos/aspectos diferentes de la imagen y diferenciarlos entre ellos.

Son muy diferentes a otros algoritmos de clasificación, ya que éstos necesitan ser diseñados a mano y las redes convolucionales no.

Las redes convolucionales, al igual que el resto de las redes neuronales, están compuestas por una red de neuronas que realizarán el proceso de análisis, aunque de lo que se diferencian es de la existencia de una serie de capas anteriores a la red de neuronas que se encargarán de procesar la imagen para que pueda ser utilizada correctamente por las neuronas de la red.

Estas capas mencionadas se llaman las **capas de convolución** y **pooling**.

4.3.6.1 ¿Qué es una capa de convolución?

Una **capa de convolución** es aquella capa que contiene una matriz de convolución que se encarga de aplicar la convolución a la imagen que se está procesando.

El **proceso de convolución** consiste en el tratamiento de una matriz por otra, que en nuestro caso va a ser el tratamiento de una imagen de entrada por la matriz de convolución denominada ***kernel***.

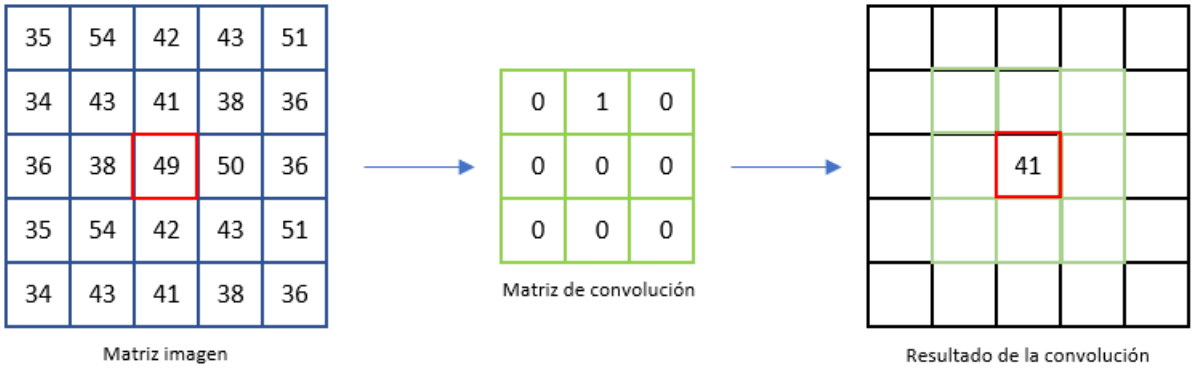
El proceso de convolución consiste en lo siguiente (N., 2020):

1. Primero se realiza un **desglose de la imagen pixel por pixel**, utilizando los valores de cada uno de los canales RGB, obteniendo tres matrices asociadas a cada uno de los canales de color de la imagen.

2. Se **crea una matriz de convolución o *kernel***, cogiendo valores aleatorios en el primer procesamiento, que será la encargada de aplicar la convolución a cada uno de los pixeles de la imagen.
   El *kernel* se irá ajustando mediante la técnica de ***back-propagation*** a medida que se va avanzando por las diferentes capas de convolución que conforman la red.

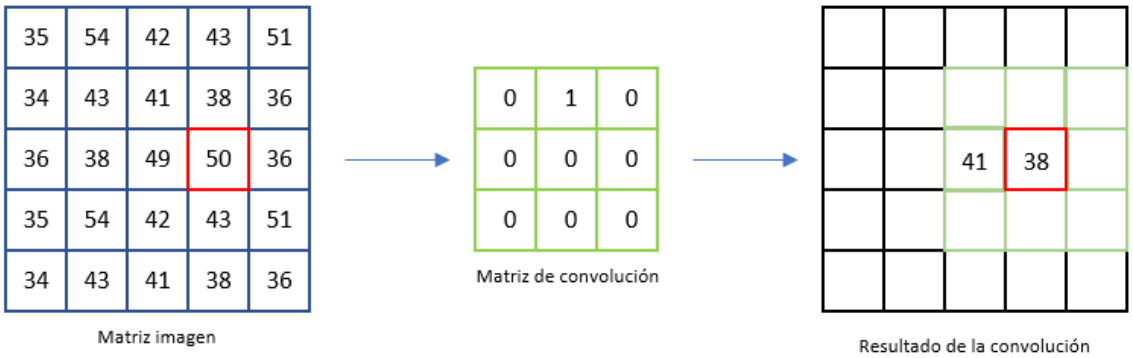
   Esto se realiza de la siguiente forma:
   Si tenemos un *kernel* de 3x3, se coge el pixel sobre el que se quiere realizar la convolución como referencia central del kernel y los vecinos a él, coincidiendo con el resto de los valores que componen la matriz kernel. Una vez asociados los valores, se realiza un sumatorio del resultado del producto del valor del pixel de la imagen por el que le corresponda del *kernel*.

3. El *kernel* recorrerá uno a uno los pixeles de la imagen, **generando una nueva imagen** con los resultados obtenidos tras la aplicación de la convolución.



Operación => (43*0)+(41*1)+(38*0)+(38*0)+(49*0)+(50*0)+(54*0)+(42*0)+(43*0)

Figura 21. Primer paso del proceso de convolución



Operación => (41*0)+(38*1)+(36*0)+(49*0)+(50*0)+(36*0)+(42*0)+(43*0)+(51*0)

Figura 22. Segundo paso del proceso de convolución

El ejemplo anterior es una **muestra gráfica** de lo que se produce en una operación de convolución, dando de esta forma un resultado como el que se muestra en la *Figura 23. Resultado convolución.*

--	--	--	--	--
--	54	42	43	--
--	43	41	38	--
--	38	49	50	--
--	--	--	--	--

Figura 23. Resultado convolución

En él se pueden visualizar los resultados de las operaciones de convolución en cada uno de los pixeles de la imagen, dando como resultado una **nueva imagen con valores RGB** distintos a la inicial.

Este tipo de operación es la que se usa para aplicar filtros a una imagen, dando lugar a muchos de los que conocemos como: el desenfocar, el negativo, el blanco y negro, el relieve entre otros.

Una de las pegas que nos encontramos en el resultado es que en el borde exterior no tenemos resultados específicos debido a la falta de datos. Para solucionar esto lo que se hace es añadir un ***padding*** o **relleno** a la imagen para que el resultado de la convolución tenga el mismo tamaño que la matriz de entrada.

Cabe señalar tambien que, dentro de una red neuronal, aunque los valores de RGB de una imagen se sitúen entre 0 y 255, se normaliza para que se comprendan entre los valores 0 y 1 y así la red neuronal los pueda interpretar de una forma más sencilla.

También, debido a la existencia de un triple canal de color, no se crea una única matriz de convolución sino que se crean 3 *kernels* de 3x3 que se aplican independientemente a cada canal de color y, posteriormente, se realiza una suma de los resultados obtenidos dando lugar a una única salida como si solo hubiese operado un *kernel*.

4.3.6.2 ¿Qué son las capas de pooling/subsampling?

Las **capas de *pooling o subsampling*** son las capas que permiten disminuir el tamaño de la matriz de la imagen para agilizar el proceso de aprendizaje.

En la capa de pooling, al igual que en la de convolución, existe una matriz del tamaño que se le asigne que irá aplicando la técnica de pooling sobre toda la matriz de la imagen.

De esta manera lo que se consigue a parte de reducir el tamaño de la imagen, es obtener los píxeles con mayor influencia dentro de la imágenes para conseguir las características más destacadas.

Para entenderlo mejor, a continuación se muestra el proceso de manera gráfica en la *Figura 24. Ejemplo subsampling.*

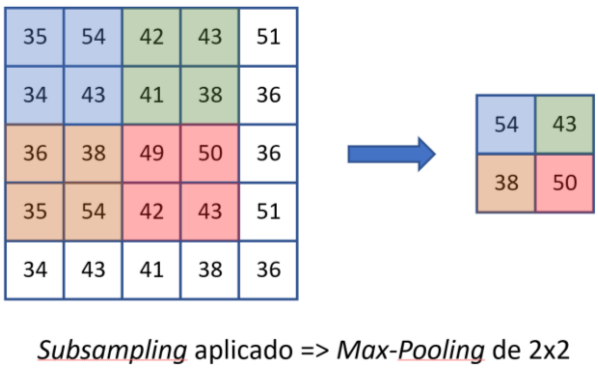


Figura 24. Ejemplo subsampling

Dentro de las capas de pooling, podemos encontrar varios tipos (Brownlee, 2019):

- **Average Pooling Layers.**
  Son un tipo de capas de pooling que calculan la media de los valores que el filtro de pooling recoge, es decir, cogiendo como ejemplo la *Figura 24. Ejemplo subsampling* y aplicando una capa *average pooling* de 2x2, las salidas serían las siguientes:
  - **Azul**: avg(35, 54, 34, 43) = **41,5**
  - **Verde**: avg(42, 43, 41, 38) = **41**
  - **Naranja**: avg(36, 38, 35, 54) = **40,75**
  - **Rojo**: avg(49. 50, 42, 43) = **46**

- **Max Pooling Layers.**
  Son un tipo de capas de pooling que extraen el valor más alto delos que el filtro de pooling a recogido. Un ejemplo visual es el representado en la *Figura 24. Ejemplo subsampling.*

- **Global Pooling Layers.**
  Son un tipo de capas de pooling que reducen el mapa de características a un único valor. De esta manera, el tamaño del filtro de pooling sería del mismo tamaño que el de la imagen que se está procesando.

4.3.7 Transfer Learning

El ***"Transfer learning"*** o **transferencia de aprendizaje** es un concepto muy interesante dentro del mundo del Deep Learning, ya que permite conseguir resultados realmente sorprendentes en problemas con pocos datos de entrenamiento y validación.

El *transfer learning* consiste en la reutilización de un modelo preentrenado con miles de imágenes para resolver un problema de clasificación o detección de imágenes en el que se tienen pocos datos de entrenamiento para conseguir una red neuronal creada desde 0 con buenos resultados.

Para ello, librerías como la por ejemplo la usada en este proyecto, **Keras**, dan acceso a bastantes modelos preentrenados para este tipo de problemas.

Por ejemplo, en Keras podemos encontrar modelos preentrenados como:

- Xception.
- VGG16.
- ResNet.
- InceptionV3.
- Mobile Net.
- …

Pero, ¿cómo realmente funciona la transferencia de aprendizaje y cómo se consigue aplicar a un nuevo modelo sin entrenar?

Al aplicar un modelo ya preentrenado al nuevo que se pretende desarrollar, tenemos que tener en cuenta que el primer modelo ya tiene sus capas entrenadas, esto quiere decir que, al ser una red neuronal convolucional, la estructura y el comportamiento de cada una de las capas será similar a cualquier otra red neuronal que desarrollemos.

Al igual que en el resto de redes neuronales convolucionales, las primeras capas de la red neuronal preentrenada distinguirán **patrones y formas mucho más simples**, a diferencia de las capas finales, que contienen mapas de características mucho más **específicos** para el problema al que daba solución dicho modelo.

Es por ello que a la hora de implementar una nueva red sobre la ya preentrenada, el primer paso es **eliminar completamente la red fully-connected** de la red preentrenada para **meter una nueva** y que aprenda en base a las nuevas imágenes de entrenamiento. También es necesario congelar todas las capas de la red preentrenada para que no vuelvan a entrenar. Esto se hace para que se mantengan todos los mapas de características con los que viene por defecto ya que son muy genéricos y útiles para cualquier proceso nuevo de clasificación de imágenes.

Una vez realizada la primera fase del entrenamiento, se vuelve a entrenar pero descongelando un conjunto específico de las capas de convolución finales del modelo preentrenado. Esto va a permitir **extraer nuevas características** bastante específicas de las imágenes de entrenamiento, consiguiendo que el mapa generado por la capa *Flatten* que recibe la nueva red fully-connected sea más específico la clasificación de las nuevas clases utilizadas.

Para verlo de una manera más visual, se adjunta a continuación la *Figura 25. Proceso transfer-learning*, en la que se describe de manera simplificada y sencilla los procesos explicados previamente.

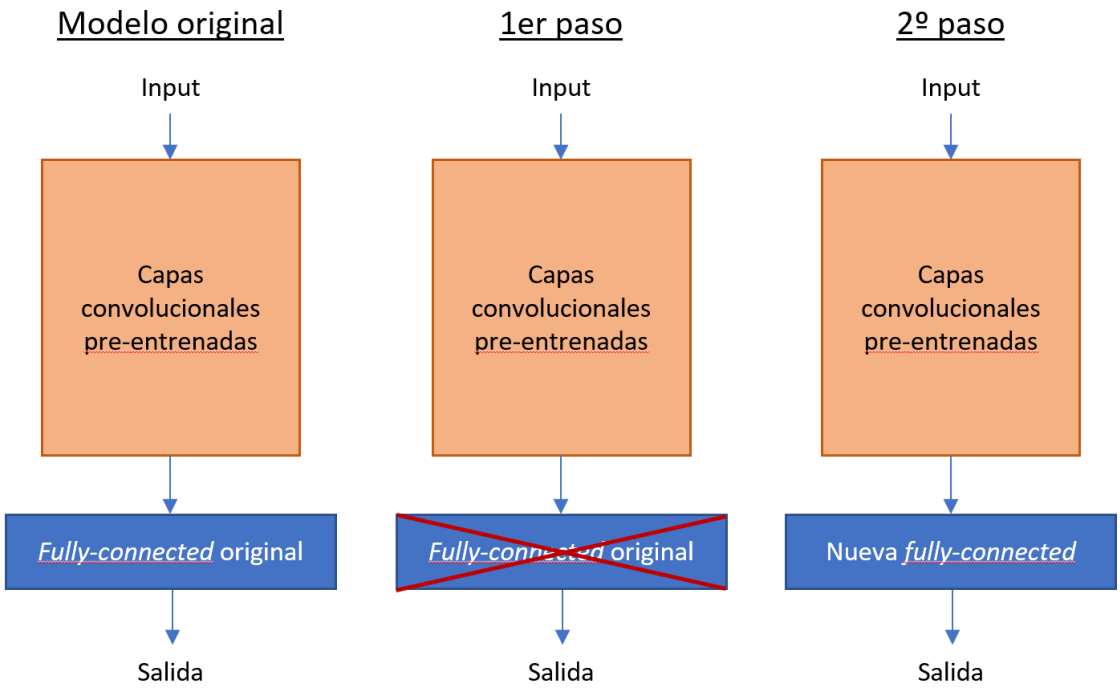


Figura 25. Proceso transfer-learning

4.3.8 Métricas para la medición de resultados del Deep Learning

Las métricas más usadas para problemas de clasificación son las siguientes:

- **Tabla o matriz de confusión.** Se usa para clacular el grado de acierto de un modelo de clasificación.

Medida	Clasificado como K	Clasificado como no K
En K	TP = True Positive	FN = False Negative
No es K	FP = False Positive	TN = True Negative

TP = corresponde a los objetos que se han clasificado correctamente de la clase K.
FN = Corresponde a los objetos que se han clasificado de otra clase diferente a K pero realmente son de la clase K.
FP = Corresponde a los objetos que se han clasificado como de la clase K, pero realmente no lo son.
TN = Corresponde a los objetos que se han clasificado de una clase diferente a K y realmente son de una clase distinta.

- **Exactitud (*accuracy*).** Corresponde al número de aciertos, es decir, a la suma de TP y TN entre el número total de ejemplos.

$$\boldsymbol{Acc} = \frac{TP + TN}{TP + FP + TN + FN}$$

- **Error**. Corresponde al número total de fallos, es decir, a la suma de FP y FN.

$$\boldsymbol{Err} = \frac{FP + FN}{TP + FP + TN + FN}$$

- **Cobertura (*recall*).** Corresponde al número de ejemplos de la clase clasificados correctamente sobre el número total de la clase.

$$\boldsymbol{Re} = \frac{TP}{TP + TN}$$

- **Precisión *(precision)*.** Corresponde al número de ejemplos de la clase clasificados correctamente sobre el número total de ejemplos clasificados de esa clase.

$$\boldsymbol{Pr} = \frac{TP}{TP + FP}$$

- **Medida F *(F-score)*.** Corresponde a la media armónica de la precisión y la cobertura.

$$\boldsymbol{F} = \frac{2 * Re * Pr}{Re + Pr}$$

4.3.9 Overfitting

El overfitting se considera un estado del entrenamiento de un modelo. Se produce cuando el **modelo se empieza a sobreajustar**, es decir, cuando se ajusta demasiado a los ejemplos que recibe de entrada. Esto significa que si recibe como entrada una imagen demasiado generalizada, será incampaz de clasificarla correctamente, ya que los mapas de características mediante los que ha aprendido son demasiado específicos a las imágenes con las que ha entrenado.

Para entenderlo mejor, supongamos que un estudiante tiene que realizar un examen y únicamente se aprende al pie de la letra los resultados de las preguntas de ese examen específico, ¿qué pasaría si le cambian todas las preguntas del examen, aunque pertenezcan al mismo temario que las preguntas del examen anterior?

Pues que se puede decir que el estudiante se encuentra en un estado de ***overfitting***, sabiéndose a la perfección las respuestas del primer examen, pero al cambiárselas no tendría ni idea de resolverlas.

Pues esto pasa cuando se **sobreentrena una red neuronal** y se aprende demasiado bien los mapas de características de un conjunto de imágenes en concreto, produciéndose el *overfitting* y consiguiendo que no sea capaz de clasificar correctamente una nueva imagen con un mapa de características más generalizado de la misma clase.

4.3.9.1 ¿Cómo se identifica el overfitting?

El primer paso para identificar el *overfitting* es dividir el conjunto de entrenamiento en dos subconjuntos:

- **Conjunto de entrenamiento.** Conjunto que se usará para entrenar el modelo. Generalmente corresponde al 80% del conjunto original.
- **Conjunto de validación.** Conjunto para validar el comportamiento del modelo frente a nuevos datos de clasificación. Corresponde al resto del conjunto original, en este caso al 20 % de los datos.

Es aquí cuando empezamos a entrenar el modelo y comenzamos a visualizar los datos en cada una de las épocas de entrenamiento.

Generalmente el error en la fase de entrenamiento suele ser menor al error de validación, ya que se está aprendiendo directamente sobre las imágenes de entrenamiento, pero ambas gráficas tienen que ir disminuyendo.

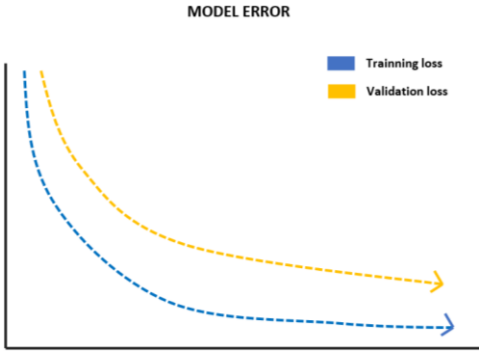


Figura 26. Gráfica entrenamiento y validación

Pero se puede dar el caso en el que el error de entrenamiento siga descendiendo (siempre va a ser así) y que el error de validación comienze a ascender. En este caso es cuando se comienza a entrar en fase de ***overfitting***, ya que el modelo está dejando de generalizar y comienza a centrarse en las características específicas de los ejemplos con los que se está entrenando.

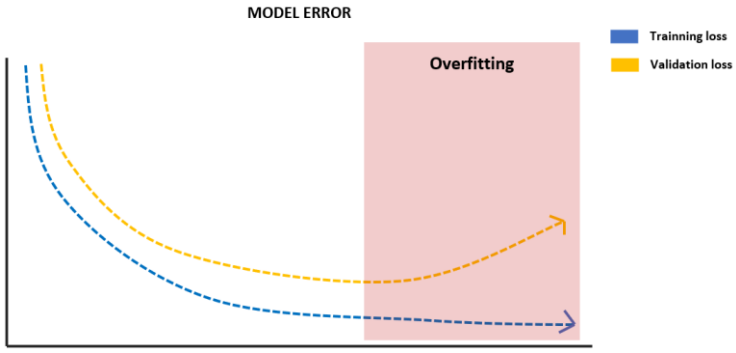


Figura 27. Gráfica entrenamiento y validación con overfitting

4.3.9.2 ¿Cómo se soluciona el overfitting?

Para evitar el overfitting se pueden aplicar varias soluciones (M., 2021):

- **Aumentar el número de imágenes de entrenamiento.** Esto ayudará al modelo a no perder esa generalización de características, evitando centrarse en características demasiado específicas de algunos de los ejemplos.
- **Detención anticipada.** Durante el entrenamiento del modelo, si se muestran las métricas del entrenamiento según se va produciendo, permite pararlo en el momento en el que se detecte el sobreajuste.
- **Parada anticipada.** Consiste en introducir una función al modelo que controle el cambio de una métrica en concreto. Esto va a permitir que si dicha métrica sobrepasa un umbral establecido, la función detenga el entrenamiento.
- **Validación cruzada.** Consiste en la división del conjunto de entrenamiento en K grupos y se realizan varios entrenamientos juntando todos los grupos creados menos uno, que será el usado para la validación.

4.4 Descripción de la solución

Como ya se ha descrito en el Capítulo 3, el objetivo principal del proyecto es **desarrollar un modelo de clasificación** para diferenciar una serie de imágenes dependiendo la marca a la que representa el logo que aparece en ellas.

Es por ello por lo que se ha usado la biblioteca de **Keras**, implementada en TensorFlow, para el uso e implementación de **redes neuronales convolucionales**.

Los apartados dentro de este punto serán:

1. Explicación paso a paso del **desarrollo de cada uno de los modelos** que se han creado.
2. Explicación del fichero de **predicción** mediante el cual se obtienen los resultados de cada uno de los modelos. El fichero de predicción es común a todos los modelos, es decir, se ha desarrollado y adaptado para que se pueda predecir un conjunto de imágenes sin catalgorar únicamente cambiando el nombre del modelo que se quiera usar.

4.4.1 Tratamiento de datos

El fichero de tratamiento de datos se ha creado para que, a partir de la carpeta que se nos ofrece con todas las clases e imágenes para usar, se creen las carpetas con las subcarpetas e imágenes correspondientes para cada uno de los procesos de entrenamiento, validación y clasificación.

Se va a crear **una carpeta por cada proceso,** y en el caso de los procesos de entrenamiento y validación, adicionalmente se crearán las carpetas correspondientes a cada una de las clases para poder meter en cada una de ellas el porcentaje de imágenes correspondiente a cada proceso.

TRATAMIENTO DE IMÁGENES. PASO 1: Imports y declaración de variables

En este apartado se añaden todas las librerías necesrias para la realización del tratamiento de imágenes y las variables globales que se van a usar.

- **pathDatasetComun**. Ruta donde se encuentra el dataset facilitado por IBM.
- **pathClasificacion**. Ruta en la que se guardan las imágenes para la clasificación.
- **pathEntrenamiento**. Ruta en la que se guardan las carpetas con las imágenes de cada clase para el entrenamiento.
- **pathPreview**. Ruta en la que se guardan las copias generadas por el *ImageDataGenerator* del entrenamiento.
- **pathValidacion**. Ruta en la que se guardan las carpetas con las imágenes de cada clase para la clasificación.
- **trainPercentage**. Porcentaje correspondiente a las imágenes de entrenamiento.

```
import os, shutil
from pathlib import Path
import random
from shutil import rmtree
pathDatasetComun = './Dataset común'
pathClasificacion = './splitted_data/Clasificación'
pathEntrenamiento = './splitted_data/Entrenamiento'
pathPreview = './splitted_data/preview'
pathValidacion = './splitted_data/Validacion'
trainPercentage = 0.88
```

Código 1. Tratamiento imágenes: import y declaración de variables

TRATAMIENTO DE IMÁGENES. PASO 2: Funciones

En este apartado se van a crear todas las funciones necesarias para tener un código más estructurado y organizado.

- ***createFolder(path).***
  Mendiante esta función se pretende crear todas las carpetas necesarias para formar la estructura de carpetas deseada.
  Recibe como parámetro *path*, que es la ruta y el nombre de la carpeta que se pretende crear.

```
def createFolder(path):
    try:
        if os.path.isdir(path) == False:
            os.mkdir(path)
    except OSError:
        print (f"La creación de la carpeta {path} ha fallado.")
```

Código 2. Tratamiento imágenes: createFolder()

- ***getFoldersName(path).***
  Función con la que se obtiene el nombre de todas las carpetas de un directorio, consiguiendo de esta forma el nombre de las clases con las que va a trabajar el modelo.
  Recibe como parámetro el **directorio** del que se obtienen los nombres de las carpetas.
  Devuelve una **lista con los nombres obtenidos**.

```
def getFoldersName(path):
    return [name for name in os.listdir(f"{path}")]
```

Código 3. Tratamiento imágenes: getFoldersName()

- ***getImagesPath(path)***
  Función que se usa para obtener el listado de las rutas de las imágenes almacenadas en un directorio específico.
  Cabe señalar que unicamente devuelve la ruta de los ficheros con extensión ***.jpg***.

```
def getImagesPath(path):
    return sorted(list(Path(f"{path}").glob('*.jpg')), key=lambda k:
```

Código 4. Tratamiento imágenes: getImagesPath()

TRATAMIENTO DE IMÁGENES. PASO 2: Proceso final

En la parte final del script, primero se comprueba que la carpeta ***'splitted_data'*** no existe, y en el caso de existir se borra para crear una nueva con nuevas imágenes randomizadas. Luego se crean las carpetas correspondientes a los procesos de entrenamiento, validación y clasificación.

Se van a usar las mismas imágenes para validación como para entrenamiento. De esta forma, el modelo a la hora de clasificar las interpretará como imágenes nuevas.

Se crea una lista con los directorios en los que las imágenes van a estar separadas por clases y se crea una lista con los nombres de las clases existentes, mediante la función *getFoldersName()*.

```
if os.path.isdir('./splitted_data'):
    rmtree('./splitted_data')

createFolder('./splitted_data')
createFolder(pathClasificacion)
createFolder(pathEntrenamiento)
createFolder(pathPreview)
createFolder(pathValidacion)

folderPathsList = [pathEntrenamiento, pathValidacion]
classNamesList = getFoldersName(pathDatasetComun)
```

Código 5. Tratamiento imágenes: proceso final parte 1

Finalmente recorremos el listado de clases existentes:

- En el caso de las carpetas de **Validación** y **Entrenamiento**, se crean las carpetas asociadas a las clases y se copia dentro de ellas las imágenes correspondientes al porcentaje indicado en la variable ***trainPercentage*** (Ej: Train --> 0.88, Val --> 0.12).

- En cambio, en el caso de la carpeta de **Clasificación**, simplemente se van introduciendo las imágenes de validación para, posteriormente, poder probar manualmente el modelo.

```
for className in classNamesList:
    imagesListToCopy = getImagesPath(f"{pathDatasetComun}/{className}")
    trainImagesLen = int(len(imagesListToCopy) * trainPercentage) + 1
    train_data = imagesListToCopy[:trainImagesLen]
    test_data = imagesListToCopy[trainImagesLen:]

    for trainImg in train_data:
        destinyFolderTrain = f"{pathEntrenamiento}/{className}"
        createFolder(destinyFolderTrain)
        shutil.copy(trainImg, destinyFolderTrain)

    for valImg in test_data:
        destinyFolderVal = f"{pathValidacion}/{className}"
        createFolder(destinyFolderVal)

        shutil.copy(valImg, destinyFolderVal)
        shutil.copy(valImg, pathClasificacion)
```

Código 6. Tratamiento imágenes: proceso final parte 2

4.4.2 Modelo 1: Modelo desarrollado desde 0

El primer modelo que se ha desarrollado no toma como referencia niguno otro y se ha creado desde 0, es decir, implementando una a una las capas que han ido haciendo falta y probándolo a medida que se iba modificando.

El modelo consta de la siguiente arquitectura final:

Layer (type)	Output Shape	N params
Conv2D	(100, 100, 32)	896
MaxPooling2D	(50, 50, 32)	0
Conv2D	(50, 50, 64)	8256
MaxPooling2D	(50, 50, 32)	0
Conv2D	(25, 25, 128)	73856
MaxPooling2D	(50, 50, 32)	0
Flatten	8192	0
Dense	1024	8389632
Dropout	1024	0
Dense 1	23	23575

Contando con un total de **8.496.215 parámetros entrenables**.

Los pasos que se han seguido para su desarrollo han sido los siguientes:

MODELO 1. PASO 1: Declaración de variables e hiperparámetros

El primer paso a realizar es, mediante la opción **backend** de Keras, limpiar y purgar todas las posibles sesiones pasadas para empezar el proceso desde una sesión limpia y recien iniciada.

```
K.clear_session()
```

Código 7. Modelo 1. Clear session()

Creamos varias variables con los parámetros que vamos a necesitar para la creación del modelo de la red neuronal convolucional.

Los parámetros que se han necesitado son los siguientes:

- **ALTURA y LONGITUD**. Tamaño al que vamos a procesar las imágenes de entrenamiento.
- **BATCH_SIZE**. Es el nº de imágenes que le vamos a mandar a la computador para procesar en cada uno de los pasos. Cuanto más grande se la variable, mayor será el tamaño del batch y por lo tanto, mayor número de imágenes simultáneas se podrán procesar. El tamaño elegido es totalmente aleatorio y modificado mediante la realización de pruebas.
- **N_FILTERS_X**. Son la cantidad de filtros que se van a utilizar en cada una de las convoluciones, es decir, tras aplicar la primera capa de convolución nuestra imagen va a tener una profundidad de *N_FILTERS_1* y al aplicar la segunda pasará a tener una profundidad de *64*. A este conjunto de filtros se le conoce como "*feature mapping*".
  El número de filtros se ha querido que sea exponencial para conseguir que la profundidad del mapa de características fuera aumentando exponencialmente a medida que se avanza en las capas de convolución.
- **ACTIVATION_X.** Es la función de activación utilizada en cada una de las capas de convolución. Se ha elegido la función ReLU porque es el que mejores resultados da.
- **KERNEL_SIZE_X**. Es el tamaño del filtro que se va a usar en cada convolución (ALTURA=3, ancho=3) y (ALTURA=2, ancho=2). Generalmente se suelen usar filtros de dimensiones 2x2 o 3x3, por lo que es lo que se ha elegido, aparte de que son los que mejores resultados dan.
- **POOL_SIZE_X**. Tamaño del filtro de maxPooling.
- **Clases**. Son los diferentes tipos de datos que van a existir en nuestro dataset.
- **lr. Learning rate**. Es el ratio de aprendizaje, que cambios va a hacer nuestra red neuronal para acercarse a una solución óptima.

```
ALTURA, LONGITUD = 100, 100
BATCH_SIZE = 4
N_FILTERS_1 = 32
N_FILTERS_2 = 64
N_FILTERS_3 = 128
ACTIVATION_1 = 'relu' # Función de activación ReLU (Rectifier Linear Unit)
ACTIVATION_2 = 'relu' # Función de activación ReLU (Rectifier Linear Unit)
ACTIVATION_3 = 'relu' # Función de activación ReLU (Rectifier Linear Unit)
KERNEL_SIZE_1 = (3,3) # ALTURA=3, anchura=3
KERNEL_SIZE_2 = (2,2) # ALTURA=2, anchura=2
KERNEL_SIZE_3 = (3,3) # ALTURA=3, anchura=3
POOL_SIZE_1 = (2,2) # ALTURA=2, anchura=2
POOL_SIZE_2 = (2,2) # ALTURA=2, anchura=2
POOL_SIZE_3 = (3,3) # ALTURA=3, anchura=3
clases = 23
lr = 0.0005
```

Código 8. Modelo 1: declaración de variables

MODELO 1. PASO 2: Creación del modelo

El segundo paso es **crear la red neuronal** compuesta por las **capas de convolución** y la **red *fully-connected*** con sus hiperparámetros correspondientes que procesarán las imágenes para su futuro aprendizaje.

```
cnn = Sequential()
```

Código 9. Modelo 1: tipo de red neuronal

PRIMERA CAPA

Capa de convolución

1. Decimos que la capa va a ser una capa **CONVOLUCIONAL**, se usará la capa *Convolution2D* que nos ofrece Keras.
2. ***N_FILTERS_1***: el nº de filtros que va a tener la capa.
3. ***KERNEL_SIZE_1***: el tamaño del kernel.
4. ***padding***: Se utiliza para que se cree una imagen del mismo tamaño que la de origen.
5. ***input_shape***: establecemos el tamaño de las imágenes de la primera capa y el nº de canales de color, que generalmente van a ser 3 (RGB).
6. ***ACTIVATION_1***: es la función de activación que se va a usar. En la mayoria de los clasificadores de imágenes se hace uso de *ReLU*.

EXPLICACIÓN:

Esta va a ser la primera capa de convolución, a la que se le van a aplicar N filtros (*N_FILTERS_1*) con un tamaño de *KERNEL_SIZE_1*.

Sabiendo esto y el tamaño de las imágenes que se van a procesar (*ALTURA*, *LONGITUD*), que en este caso son 100 y 100, podemos obtener el nº de neuronas de salida que daría lugar el primer proceso de convolución:

```
100x100x1 * 32 = 320.000 neuronas (altura = 100, ancho = 100,
profundidad = 1)
```

320.000 neuronas necesarias para la primera capa de convolución.

```
cnn.add(Convolution2D(filters=N_FILTERS_1, kernel_size=KERNEL_SIZE_1,
padding='same', activation=ACTIVATION_1))
```

Código 10. Modelo 1: primera capa de convolución

Capa de *subsampling*

Debido a esta cantidad tan grande de neuronas para procesar los datos generados por la capa de convolución, se hace necesaria una capa intermedia de ***subsampling***, es decir, una capa que se encargue de minimizar el tamaño de las imágenes filtradas pero quedándose con las características más importantes detectadas por el anterior filtro.

El más usado actualmente es el filtro de ***Max-Pooling*** (*MaxPooling2D*).

El tamaño de pool es de (2, 2) por lo que quedarían unas imágenes de tamaño **50 x 50** y una profundidad de **32**.

```
Tamaño pool = 2x2
Salida convolución = 50x50x32
```

```
cnn.add(MaxPooling2D(pool_size=POOL_SIZE_1))
```

Código 11. Modelo 1: primera capa de subsampling

SEGUNDA CAPA

Capa de convolución

Aplicamos nuevamente una capa de convolución de KERNEL_SIZE_2 y con N filtros (N_FILTERS_2).

```
50x50x32 * 64 = 160.000 neuronas
Feature mapping = 50x50x64
```

160.000 neuronas necesarias para la segunda capa de convolución.

```
cnn.add(Convolution2D(filters=N_FILTERS_2, kernel_size=KERNEL_SIZE_2,
padding='same', activation=ACTIVATION_2))
```

Código 12. Modelo 1: segunda capa de convolución

Capa de *subsampling*

Creamos una nueva capa de subsampling para minimizar el tamaño de las imágenes a procesar:

```
Tamaño pool = 2x2
Salida convolución = 25x25x64
```

```
cnn.add(MaxPooling2D(pool_size=POOL_SIZE_2))
```

Código 13. Modelo 1: segunda capa de subsampling

TERCERA CAPA

Capa de convolución

Aplicamos nuevamente una capa de convolución de KERNEL_SIZE_3 y con N filtros (N_FILTERS_3).

```
25x25x64 * 128 = 80.000 neuronas
Feature mapping = 25x25x128
```

80.000 neuronas necesarias para la tercera capa de convolución.

```
cnn.add(Convolution2D(filters=N_FILTERS_3,kernel_size=KERNEL_SIZE_3
, padding='same', activation=ACTIVATION_3))
```

Código 14. Modelo 1: tercera capa de convolución

Capa de *subsampling*

Creamos una nueva capa de subsampling para minimizar el tamaño de las imágenes a procesar:

```
Tamaño pool = 3x3
Salida convolución = 5x5x128
```

```
cnn.add(MaxPooling2D(pool_size=POOL_SIZE_3))
```

Código 15. Modelo 1: tercera capa de subsampling

CAPAS FLATTEN Y DENSE

A continuación, el resultado de la última capa de *MaxPooling* se convierte en un array plano para poder pasarselo a la red *fully-connected* y de esta forma conseguir que cada neurona se encargue de una parte de la imagen procesada. Esta imagen va a ser muy pequeña pero a la vez muy profunda, y con la función ***Flatten()*** lo que se pretende es obtener la imagen plana, o como se ha dicho antes en forma de array.

```
cnn.add(Flatten())
```

Código 16. Modelo 1: capa Flatten()

Después de aplanar la información, la enviamos a una capa de X neuronas para que empiecen a entrenar con los datos de entrada. Cabe señalar que la **función de activación** que aplican las neuronas puede establecerse como hiperparámetro dentro de la clase *Dense(),* por lo que en este caso se ha elegido la función de activación ***ReLU***.

La capa *Dense()* que vamos a crear va a tener los siguientes hiperparámetros:

- ***Activation***: corresponde a la función de activación que queremos que usen las neuronas de esa capa. En este caso la elegida es ReLU.
- ***Units***: indica el nº de neuronas que formarán la capa.
- Kernel_regularizer.
- Bias_regularizer.
- _regularizer.

```
cnn.add(Dense(activation='relu', units=1024,
kernel_regularizer=regularizers.l1_l2(l1=1e-5, l2=1e-4),
bias_regularizer=regularizers.l2(1e-4),
_regularizer=regularizers.l2(1e-5) ))
```

Código 17. Modelo 1: primera capa red fully-connected

Durante el entrenamiento que van a llevar a cabo las neuronas de la red, mediante la clase ***Dropout()*** le vamos a apagar el 50% de las neuronas en cada paso.

Esto se realiza para evitar el **sobreajuste**, ya que si las neuronas están activadas, la red neuronal puede que aprenda un único camino para clasificar una clase y mediante esta técnica las neuronas conseguirán diferentes caminos para clasificar.

```
cnn.add(Dropout(0.5))
```

Código 18. Modelo 1: capa Dropout()

Finalmente se añade una nueva capa ***Dense()*** pero con la función de activación '***softmax***', la cual ayudará a identificar a qué clase pertenece la imagen con la que se ha estado entrenando previamente.

Para ello es necesario indicarle el **nº de clases** con las que está trabajando el modelo, que en nuestro caso son 23, incluyendo todas las marcas con las que está realizando el entrenamiento.

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

Código 19. Modelo 1: segunda capa red fully-connected

MODELO 1. PASO 3: Optimización y compilación del modelo

Durante el entrenamiento, la red neuronal va a usar el algoritmo de ***CrossEntropy*** para ver que tan bien o que tan mal va.

El algoritmo de optimización va a ser ***Adam***, con un learning rate de **0.0005**, y la métrica mediante la cual se irá optimizando el modelo será ***accuracy***.

```
optimizer = adam_v2.Adam(learning_rate=0.0005)
```

Código 20. Modelo 1: optimizador de la red

Mediante la función ***compile()*** lo que se consigue es configurar el modelo para su posterior entrenamiento, y de la cual vamos a usar los siguiente parámetros:

- ***loss:*** es la función de pérdida mediante la cual se va a basar el algoritmo de back-propagation para cambiar los pesos de las conexiones entre las neuronas y así disminuir todo los posible el error final.
- ***optimizer***: es el optimizador que se va a usar a la hora de realizar el entrenamiento. Keras nos ofrece una gran cantidad de optimizadores entre los que podemos encontrar FTRL, SGD (Gradient Descent), …, y el que se ha elegido entre todos ellos es **Adam**.
- ***metrics***: es la lista de métricas que van a ser evaluadas por el modelo durante las fases de entrenamiento y test.

```
cnn.compile(loss='categorical_crossentropy',
optimizer=optimizer, metrics=['accuracy'])
```

Código 21. Modelo 1: compilación

MODELO 1. PASO 4: Preparación del dataset de imágenes

El siguiente paso a llevar a cabo es la **carga** y **procesamiento** de las imágenes que se van a usar tanto para el entrenamiento como para la validación del modelo.

Es necesaria la creación de ambos datasets (entramiento y validación) para poder detectar problemas como el *overfitting* y aplicar mejoras al modelo.

Establecemos las variables mediante los directorios donde se encuentran las imágenes de entrenamiento y de validación y creamos dos instancias de ***ImageDataGenerator*** para generar nuevas imágenes distorsionadas con las que el modelo pueda entrenar.

Cabe enfatizar que al conjunto de imágenes para validación no es necesario aplicarle ningún tipo de deformación ya que queremos que el modelo se evalue con imágenes reales.

```
data_entrenamiento='./2_Datasets/splitted_data/Entrenamiento'
data_validacion='./2_Datasets/splitted_data/Validacion'
```

Código 22. Modelo 1: directorios datasets de entrenamiento y validación

Para ello, la clase ofrece una serie de parámetros para indicar cómo puede deformar esas imágenes de entrada:

- ***rotation_range***: el rango de la rotación aleatoria aplicada a las imágenes para generar nuevos ejemplares.
- ***rescale***: rescala las imagenes a la cifra proporcionada, es decir, como el canal RGB es de 0 a 255, se realiza una conversión para que dicho valor se encuentre entre 0 y 1, siendo valor/255.
- ***shear_range***: inclina las imágenes para que el modelo sepa que las imagenes pueden estar giradas.
- ***zoom_range***: al igual que la enterior, le hace zoom al 30% de las imágenes para que el modelo aprenda.
- ***horizontal_flip***: voltea las imágenes.
- width_shift_range.
- height_shift_range.

```
entrenamiento_datagen = ImageDataGenerator(rotation_range=15,
rescale=1./255, shear_range=0.1, zoom_range=0.2,  horizontal_flip=True,
width_shift_range=0.1, height_shift_range=0.1)
validation_datagen = ImageDataGenerator(rescale=1./255)
```

Código 23. Modelo 1: Data Augmentation mediante ImageDataGenerator

Para un ejemplo más visual de las imágenes que genera el *ImageDataGenerator*, se ha creado un pequeño trozo de código para guardar el primer batch de imágenes que crea con la primera imagen de una de las marcas, en este caso AMD.

1. El primer paso es **cargar la imagen** de la ruta en una variable mediante la función ***load_img()*** que nos ofrece Keras, la cual convierte la imagen en el formato PIL (Python Imagen Library) con el fin de que el *interpreter* de Python pueda procesarla correctamente.

```
img = load_img('./2_Datasets/splitted_data/Entrenamiento/AMD/AMD (1).jpg')
```

Código 24. Modelo 1: ejemplo de data aumentation

2. Luego se pasa la imagen a ***Numpy array*** con un tamaño de 3 dimensiones usando ***img_to_array()***, al cual hay que añadirle posteriormente otra dimensión más mediante ***reshape()*** para poder guardarla como imagen, quedando un array con dimensiones *(1, 3, 150, 150)*.

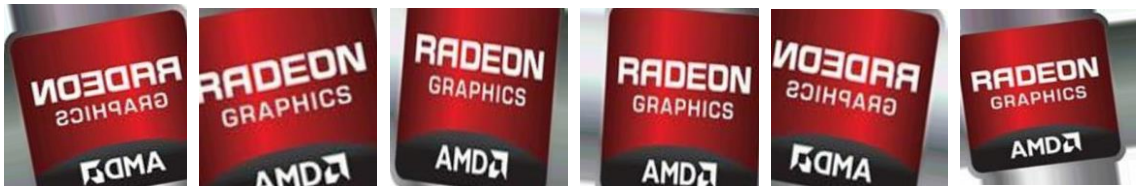
```
x = img_to_array(img)
x = x.reshape((1,) + x.shape)
```

3. Por último se genera un *loop* mediante un *for* para generar tantas imágenes modificadas como queramos mediante la función ***flow()*** del *ImageDataGenerator*, que ofrece el

parámetro ***save_to_dir()*** para poder guardar la nueva imagen generada en un directorio específico.

```
i = 0
for batch in entrenamiento_datagen.flow(x, batch_size=1,
save_to_dir='./2_Datasets/splitted_data/preview',
save_prefix='amd', save_format='jpg'):
    i += 1
    if i > 20:
        break
```

A continuación se muestran varios ejemplos de las nuevas imágenes generadas:



Una vez indicados los parámetros para generar las nuevas imágenes, se le indica al generador la ruta de las imágenes. La forma de carga depende de cómo se tengan las imágenes almacenadas, ya que se pueden cargar desde un ***DataFrame***, desde un **directorio de carpetas** o desde un ***array Numpy***.

En nuestro caso se va a cargar desde un directorio de carpetas, donde tenemos organizado qué imágenes se van a usar para el **entrenamiento** y cuáles para la **validación**.

La carga se realizará mediante la función de ***flow_from_directory***, la cual necesita:

1. Un parámetro de entrada para indicar la **ruta del directorio de carpetas** donde se encuentran las imágenes de entrenamiento y validación.
2. Las carpetas de **entrenamiento** y **validación** tienen que tener las clases a predecir bien estructuradas en otro directorio de carpetas, correspondiendo el nombre de cada carpeta a la clase de las imágenes que contiene, quedando un árbol de carpetas parecido al siguiente:

```
|-- Entrenamiento
|       |-- Clase 1
|               |-- ImgClase1_1.jpg
|               |-- ImgClase1_2.jpg
|               |-- ImgClase1_3.jpg
|               |-- ...
|       |-- Clase 2
|       |-- Clase 3
|       |-- ...
|-- Validación
|       |-- Clase 1
```

```
|               |-- ImgClase1_1.jpg
|               |-- ImgClase1_2.jpg
|               |-- ImgClase1_3.jpg
|               |-- ...
|       |-- Clase 2
|       |-- Clase 3
|       |-- ...
```

```
imagen_entrenamiento =
entrenamiento_datagen.flow_from_directory(data_entrenamiento,
target_size=(ALTURA, LONGITUD), batch_size=BATCH_SIZE,
class_mode='categorical')
imagen_validacion = validation_datagen.flow_from_directory(data_validacion,
target_size=(ALTURA, LONGITUD), batch_size=BATCH_SIZE,
class_mode='categorical')
```

Código 25. Modelo 1: preparación de los datos de entrenamiento/validación

MODELO 1. PASO 5. Entrenamiento del modelo

Lo primero que se ha creado han sido los *callbacks* de Keras. Estos *callbacks* son funciones que ofrece Keras para mejorar tanto el análisis como el rendimiento del entrenamiento, y que se pueden aplicar al modelo de una manera muy sencilla.

Los callbacks que se han elegido por su utilidad han sido:

- ***ReduceLROnPlateau()***. Este *callback* se encarga de disminuir el *learning rate* en el caso en el que la métrica indicada haya dejado de mejorar durante X épocas (Team, s.f.).

```
model_reducelr_callback = ReduceLROnPlateau(monitor='val_loss',
factor=0.2, patience=4, min_lr=0.0005)
```

Código 26. Modelo 1: callback ReduceLROnPlateau()

- ***ModelCheckpoint().*** Callback que se encarga de guardar los pesos en cualquier momento del entrenamiento según los parámentros que se le indique (Team, s.f.).

```
checkpoint_filepath = '/'
model_checkpoint_callback =
ModelCheckpoint(filepath=checkpoint_filepath, save_weights_only=True,
monitor='val_accuracy', mode='max', save_best_only=True)
```

Código 27. Modelo 1: callback ModelCheckpoint()

Y antes de comenzar con el entrenamiento hay que establecer el número de épocas que va a tener y el nº de pasos por época tanto en la fase de entrenamiento como en la de validación.

- **Épocas**. Nº de veces que el modelo va a iterar sobre las imágenes de entrenamiento.

- **Steps_per_epoch**. Son las imágenes generadas por el *ImageDataGenerator()* en cada época, sabiendo que cada época tiene un tamaño específico de batch, por lo que se calculan de la siguiente forma:

```
nº de imagenes de entrenamiento / tamaño de batch
```

- **Validation_steps.** Al igual que en el entrenamiento, a la hora de validarlo también se necesita establecer el nº de paso por época. En este caso se obtiene de la siguiente forma:

```
nº de imagenes de validación / tamaño de batch
```

```
epocas = 5
steps_per_epoch_train = int(imagen_entrenamiento.n // BATCH_SIZE)
steps_per_epoch_val = int(imagen_validacion.n // BATCH_SIZE)
```

Código 28. Modelo 1: declaración de épocas y pasos por época en el entrenamiento

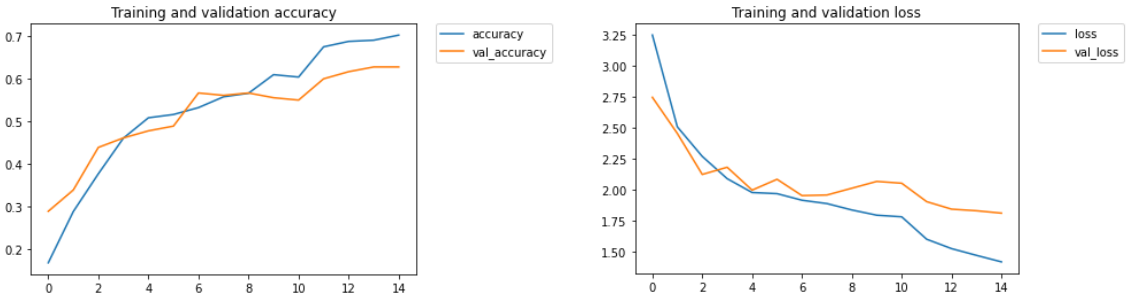
Y finalmente entrenamos el modelo mediante la función *fit()* **guardándolo en una variable** para su futuro **testeo**.

```
modelo = cnn.fit(imagen_entrenamiento,
steps_per_epoch=steps_per_epoch_train, epochs=epocas,
validation_data=imagen_validacion, validation_steps=steps_per_epoch_val,
callbacks=keras_callbacks)
```

Código 29. Modelo 1: entrenamiento y guardado del modelo

MODELO 1. PASO 6: Análisis de resultados

En este paso se pretende analizar los datos obtenidos del entrenamiento del modelo, visualizando las métricas guardadas en gráficas y de esta manera conseguir detectar anomalías en el entrenamiento de una manera más sencilla y rápida.



4.4.3 Modelo 2: Modelos desarrollados mediante *Transfer Learning*

En este apartado se van a explicar paso a paso los modelos desarrollados bajo la técnica del *transfer learning*. Como los modelos han sido desarrollados siguiendo el proceso explicado en el apartado de Transfer Learning, se va a explicar el que mejores resultados ha dado.

Los modelos que nos ofrece Keras y los cuales hemos probado para el desarrollo de clasificador son los siguientes:

- **ResNet101_v2.**
- **ResNet50_v2.**
- **ResNet152_v2.**
- **VGG16.**
- **InceptionV3.**
- **Xception.**

MODELO 2. PASO 1: Imports y declaración de variables

IMPORTS

En este primer apartado, la idea es importar todas las librerías necesarias para realizar el proceso de transferencia de aprendizaje y declarar todas las variables que sean necesarias para el modelo que se pretende entrenar.

Las librerías que se van a usar son las siguientes:

- **matplotlib**: Es una librería desarrollada en Python para representar gráficas.
- **keras**: Librería contenedora de todas las capas y algoritmos necesarios para la creación de la nueva red neuronal.
- **pathlib**: Librería que permite la creción de rutas a directorios.
- **os**: Librería interna del sistema operativo.

```
import matplotlib.pyplot as plt
from pathlib import Path
from keras.applications.resnet_v2 import ResNet101V2
from keras.layers import GlobalAveragePooling2D, Dense, Dropout
from keras.models import Model
from keras.callbacks import ModelCheckpoint, EarlyStopping, ReduceLROnPlateau
from keras.preprocessing import image
from tensorflow.keras.optimizers import SGD, RMSprop
```

Código 30. Modelo 2: imports

VARIABLES

- **FREEZE_IMG_SIZE**: Tamaño de la imagen en el primer freeze de la red neuronal.
- **FIRST_UNFREEZE_IMG_SIZE**: Tamaño de la imagen tras el primer descongelamiento. La idea es que el tamaño de las imagenes del primer conjunto sea mas pequeño para que en la primera fase de descongelamiento se procesen el conjunto de imágenes como uno totalmente diferente al primero.
- **NUM_CHANNELS**: Total de canales de color de las imágenes. Generalmente es 3, haciendo referencia a los canales RGB.

- **IMG_SHAPE**: Tamaño de la imagen final, cogiendo como referencia la variable *FIRST_UNFREEZE_IMG_SIZE*.
- **TOP_EPOCHS_1**. Nº de épocas en la primera fase de entrenamiento.
- **TOP_EPOCHS_2**. Nº de épocas en la segunda fase de entrenamiento.
- **BATCH_SIZE**. Tamaño del batch. Dependiendo del tamaño del batch, se procesarán instantáneamente X imágenes.
- **DATA_PATH**. Directorio donde se encuentran las carpetas de ENTRENAMIENTO y VALIDACIÓN.

```
FREEZE_IMG_SIZE = 75
FIRST_UNFREEZE_IMG_SIZE = FREEZE_IMG_SIZE * 3
NUM_CHANNELS = 3
IMG_SHAPE = (FIRST_UNFREEZE_IMG_SIZE, FIRST_UNFREEZE_IMG_SIZE, NUM_CHANNELS)
TOP_EPOCHS_1 = 10
TOP_EPOCHS_2 = 12
BATCH_SIZE = 4
DATA_PATH = Path('./splitted_data')
```

Código 31. Modelo 2: definición de variables

MODELO 2. PASO 2: Creación de la red neuronal mediante *Transfer Learning*

En este modelo se va a hacer uso del modelo preentrenado de **ResNet101V2** que nos ofrece Keras para poder aplicar *transfer Learning* y de esta manera crear un nuevo modelo adaptado a las necesidades del problema planteado.

Para ello se le va a pasar por hiperparámetros:

- ***input_shape***. Tamaño de las imagenes que tendrán en la futura clasificación, una vez el modelo entrenado.
- ***include_top***. Mediante este parámetro se omite la red neuronal *fully-connected* que viene como predeterminada en la red *ResNet101V2* para poder meter la nueva que se creeará a mano.
- ***weights***. Para incluir en la red preentrenada los pesos generados mediante el dataset de *imagenet* o unos precargados, indicando su ruta en el directorio.

Con la función *summary()* del modelo podemos ver la estructura actual del modelo y así ver las capas existentes y el nº de parámetros entrenables.

```
resNet101_classifier= ResNet101V2(input_shape=IMG_SHAPE, include_top=False,
                                  weights='imagenet')
resNet101_classifier.summary()
```

Código 32. Modelo 2: especificación del modelo preentrenado

Después se realiza una extracción del mapa de características que devuelve la última capa de la red preentrenada para añadírsela a la nueva red neuronal fully-connected que se va a crear.

```
incv3_output = resNet101_classifier.output
print(incv3_output.shape)
```

Código 33. Modelo 2: obtención del shape de salida del modelo preentrenado

Existen dos opciones después de obtener el output de la última capa de convolución:

1. Crear una ***red fully-connected*** para procesar el mapa de características (*output*) proporcionado por la última capa de convolución.
2. Añadir una capa de ***global average pooling***, que extraiga un mapa de características por cada una de las clases existentes y se lo pase directamente a una capa de neuronas con la función de activación *softmax* para que devuelva directamente el resultado del entrenamiento.

En nuestro caso la hemos hecho de las dos formas.

De la **primera forma**:

1. Se crea una capa de **GlobalAveragePooling2D**.
2. Se conecta la salida de la anterior capa a una capa *Dense()* de 1024 neuronas *ReLU*.
3. Finalmente, se conecta la anterior capa a una capa *softmax* de 23 neuronas (nº de clases existentes) con un dropout de 0.5 para evitar *overfitting* en el modelo.

```
avg_pool = GlobalAveragePooling2D(name='logos_pooling')(incv3_output)
avg_pool.get_shape()
fully_connected = Dense(1024, activation='relu',
name='logos_fullyconnected')(avg_pool)
dropout = Dropout(0.5)(fully_connected)
predictions = Dense(23, activation='softmax', name='logos_softmax')(dropout)
```

Código 34. Modelo 2: primera forma de creación de la red fully-connected

En la **segunda forma** se omitirían los pasos de crear la red fully-connected y se crearía directamente la capa softmax con las 23 neuronas.

Esto se debe a que directamente, en el primer paso cuando se indica el modelo preentrenado, se ha indicado que se use como *pooling* una capa ***global average pooling***.

```
predictions = Dense(23, activation='softmax',
name='logos_softmax')(incv3_output)
```

Tran finalizar con la construcción de la nueva red, se genera el modelo y se imprime por pantalla su nueva estructura.

```
model = Model(inputs= resNet101_classifier.input, outputs=predictions)
model.summary()
```

Código 35. Modelo 2: segunda forma de creación de la red fully-connected

MODELO 2. PASO 3: Primera fase de entrenamiento

Dentro de la primera fase del entrenamiento, el primer paso a realizar es **congelar todas las capas convolucionales del modelo preentrenado** para mantener los pesos fijos en el primer entrenamiento de la red.

```
for layer in resNet101_classifier.layers:
    layer.trainable = False
model.summary()
```

Código 36. Modelo 2: congelado de las capas del modelo preentrenado

El segundo paso de la fase es **compilar el modelo**.

Se compila el modelo con:

- La función de ***categorical_crossentropy*** como función de error.
- ***Rms*** como optimizador para el modelo.
- Se guarda la métrica de **exactitud** para mostrarla en la fase de entrenamiento junto al error. La exactitud se explica detalladamente en el apartado "*Métricas para la medición de resultados del Deep Learning*".

```
model.compile(loss='categorical_crossentropy',
optimizer=RMSprop(learning_rate=0.001), metrics=['accuracy'])
```

Código 37. Modelo 2: primera compliación del modelo

Y el último paso de la primera fase de entrenamiento es la preparación de los conjuntos de imágenes de **entrenamiento** y **validación**, que es exactamente lo mismo que lo explicado en los **PASOS 4** y **5** del **MODELO 1,** y el **entrenamiento** del modelo**.**

MODELO 2. PASO 4: Segunda fase de entrenamiento

El proceso es el mismo que en el paso anterior, solamente se realizan una serie de cambios para que el proceso de entrenamiento esté mas enfocado a las nuevas imágenes de entrenamiento:

- Se **descongela** un grupo específico de las capas de convolución finales del modelo preentrenado. Esto va a permitir **extraer nuevas características** de las imágenes de entrenamiento, consiguiendo que los mapas de características devueltos por la capa de ***Global Average Pooling*** a la capa de ***softmax*** sean más específicos para las imágenes actuales que las de la primera fase de entrenamiento.

```
freeze_until_layer = 153
for layer in model.layers[:freeze_until_layer]: layer.trainable = False
for layer in model.layers[freeze_until_layer:]: layer.trainable = True
```

Código 38. Modelo 2: descongelado de X capas del modelo preentrenado

- Se **cambia el tamaño de las imágenes procesadas** con respecto al tamaño de las imágenes de procesamiento de la primera fase. Esto se hace para que el modelo las interprete como imágenes completamente diferentes y el tamaño de los datasets de entrenamiento y validación aumenten.

- Se **añaden una serie de *callbacks*** a la hora de comenzar el entrenamiento para tener una mayor control sobre los pesos y el learning rate con el que el modelo entrena.
  Los *callbacks* usados de los que nos ofrece Keras son:
  1. ***EarlyStopping***: es un callback que para la ejecución del entrenamiento cuando una métrica a dejado de mejorar (Team, s.f.).
     a. *monitor*: selecciona la métrica en la que se va a basar para parar el entrenamiento.
     b. *patience*: nº de épocas sin mejorar la métrica seleccionada.

  2. ***ModelCheckPoint***: Callback que se encarga de guardar los pesos en cualquier momento del entrenamiento según los parámentros que se le indique (Team, s.f.).
     a. *filepath*: ruta de guardado de los pesos.
     b. *monitor*: es el valor en el que se basa para guardar los pesos.
     c. *save_best_only*: Si el valor es mejor que los anteriores pesos guardados, lo guarda reemplazando el anterior.
     d. *mode*.

     Este callback se ha utilizado para guardar el modelo en fases donde aún no se haya entrado en *overfitting* (en el caso en el que se haya entrado).

  3. ***ReduceLROnPlateau***: es un *callback* que ofrece Keras para ir disminuyendo el *learning rate* a medida que la métrica indicada deja de mejorar o se estanca (Team, s.f.).
     a. *monitor*: es el valor en el que se basa el callback para reducir el learning rate.
     b. *factor*: el factor por el que se multiplica el learning rate cada vez que se aplica la reducción.
     c. *patience*: el número de épocas que espera el callback para reducir el learning rate.
     d. *min_lr*: cantidad mínima a la que se puede reducir el learning rate.

```
cb_early_stopper = EarlyStopping(monitor = 'val_loss', patience = 3)
cb_checkpointer = ModelCheckpoint(filepath =
'./weights/ResNet101V2_1_weights.hdf5', monitor = 'val_loss', save_best_only
= True, mode = 'auto')
model_reducelr_callback = ReduceLROnPlateau(monitor='val_loss', factor=0.1,
patience=2, min_lr=0.0001)
```

Código 39. Modelo 2: callbacks utilizados

Y, finalmente, **guardamos** el modelo para su futuro análisis, aplicándole las nuevas imágenes de entrenamiento generadas.

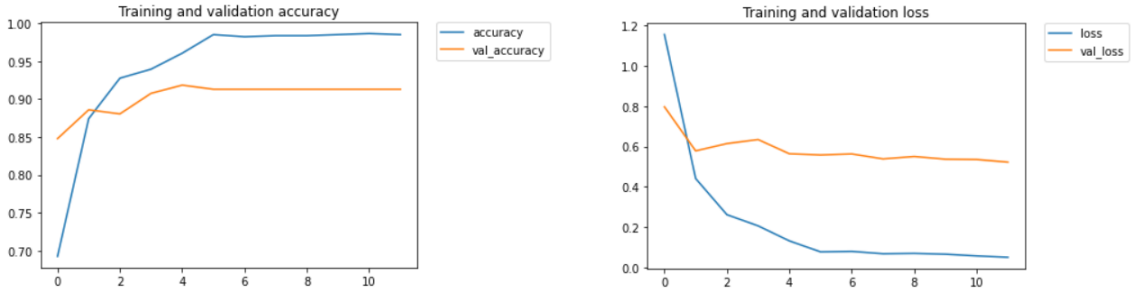
```
history = model.fit(imagenes_entrenamiento_2,
steps_per_epoch=imagenes_entrenamiento_2.n // BATCH_SIZE,
epochs=TOP_EPOCHS_2, validation_data=imagenes_validacion_2,
validation_steps=imagenes_validacion_2.n // BATCH_SIZE,
callbacks=[cb_early_stopper, cb_checkpointer, model_reducelr_callback])

model.save('./models/ResNet101V2_1_model.hdf5')
```

Código 40. Modelo 2: entrenamiento y guardado del modelo

MODELO 2. PASO 5: Análisis de resultados

En este paso se pretende analizar los datos obtenidos del entrenamiento del modelo, visualizando las métricas guardadas en gráficas y de esta manera conseguir detectar anomalías en el entrenamiento de una manera más sencilla y rápida.

Un ejemplo de resultado de salida es el siguiente:



4.4.4 Clasificador

Para poder realizar una clasificación de imágenes desordenadas, se ha creado un fichero aparte mediante el cual se podrán probar todos los modelos desarrollados.

El archivo consta de varios apartados.

1. **Imports de librerías.**
   Apartado en el que se importan todas las funcionalidades necesarias para aplicar la clasificación de las imágenes.

2. **Lista de objetos.**
   Para poder probar y guardar todos los ficheros necesarios dependiendo del modelo aplicado, se ha creado un **array de objetos** en el que se guardan todos los datos necesarios para que la clasificación se realice correctamente, dependiendo del modelo que se quiera aplicar.
   El objeto almacenado dentro del array tiene la siguiente estructura:
   - **modelName**: Nombre de modelo que se está utilizando.

- **preprocessInput**: Process input necesario para la la aplicación del modelo. Este dato es necesario ya que cada uno de los modelos creados a partir del transfer learning tiene un processInput diferente.
- **modelPath**: Path en el que se encuentra el archivo del modelo.
- **weightsPath**: Path en el que se encuentran los pesos obtenidos tras el entrenamiento del modelo.
- **reportPath**: Path en el que se generará un archivo excel con todas las métricas obtenidas de la clasificación.
- **confusionMatrixPath**: Path en el que se guardará la matriz de confusión obtenida trás la clasificación.
- **imgSize**: Tamaño de la imagen de entrada.
- **Colors**. Array en el que se almacenan los colores de las gráficas de reporte de *accuracy*, *recall* y *f1-score*. Esto se hace para que los colores de cada uno de los modelos sean distintos al del resto de los modelos.

A continuación se muestra un ejemplo de los objetos creados:

```
{"modelName": "ResNet101V2_v1", "preprocessInput": preprocessInputResNet,
    "modelPath": "./models/ResNet101V2_1_model.hdf5", "weightsPath":
        "./weights/ResNet101V2_1_weights.hdf5", "reportPath":
      "./reports/ResNet101V2_report.xlsx", "confusionMatrixPath":
"./confusion_matrix_images/ResNet101V2_classifier_output.png", "imgSize":
            225, "colors": ['#1B4F72','#2E86C1','#5DADE2']}
```

Código 41. Clasificador: objeto de características de los modelos

Luego se guarda el modelo deseado en la variable **MODEL_SELECTED**, accediendo al objeto del array mediente su índice.
Finalmente, se crea un array con todas las **clases** posibles a clasificar por los modelos llamada **LABELS_OBJ**.

3. **Funciones**.
   - *predictor(img, model).*
     Función que recibe por parámetro la **imagen a clasificar** y el **modelo** mediante el cual se realizará dicha clasificación.
     Devuelve el array de probabilidades obtenidas tras la clasificación.

```
def predictor(img, model):
    x = image.img_to_array(img)
    x = np.expand_dims(x, axis=0)
    x = MODEL_SELECTED['preprocessInput'](x)
    probabilities = model.predict(x)
    return probabilities
```

Código 42. Clasificador: Función predictor()

- *getImgsForPrediction(path)*
  Función que obtiene las rutas de las imágenes a clasificar.
  Recibe por parámetro la **ruta** donde se encuentran las imágenes a clasificar.
  Devuelve una **lista con las rutas** de cada una de las imágenes.

```
def getImgsForPrediction(path):
    return list(Path(f"{path}").glob('*.jpg'))
```

Código 43. Clasificador: Función getImgsForPrediction()

- *getImgClassByName(imgName)*
  Función que se encarga de obtener la clase de la imagen únicamente con el nombre de la imagen.
  Recibe por parámetro la **ruta de la imagen**.
  Devuelve el **nombre de la clase** a la que pertenece la imagen.
  Para que devuelva correctamente la clase de la imagen, la ruta de la imagen tiene que tener la siguiente estructura:

  './Clasificacion/Clase_Imagen (x).jpg'

  Y en el nombre de la imagen debe de ir la clase a la que pertenece.
  De esta manera se obtendrá únicamente la parte de **Clase_Imagen**.

```
def getImgClassByName(imgName):
    return imgName.split('\\')[2].split(' (')[0]
```

Código 44. Clasificador: Función getImgClassByName()

- *createReportBarGraph(metric, df, modelSelected).*
  Función que se encarga de crear y guardar una gráfica con la métrica seleccionada del resultado de la clasificación.
  Recibe por parámetro la métrica correspondiente, el DataFrame con los datos y el modelo seleccionado.

```
metrics = {'precision': 0, 'recall': 1, 'f1-score': 2}
def createReportBarGraph(metric, df, df_labels, modelSelected):
    df_labels = list(df.index.values)
    fig, ax = plt.subplots(figsize=(120, 30))
    axes = plt.gca()
    axes.set_ylim([0, 1])
    plt.bar(df_labels, df[metric].tolist(),
        color=modelSelected['colors'][metrics[metric]])
plt.savefig(f"./reports/{metric}_img/{modelSelected['modelName']}_{metric}.png",
                                format='png')
```

Código 45. Clasificador: Función createReportBarGraph()

4. **Clasificación.**

En este apartado se va a producir la clasificación de cada una de las imágenes, imprimiendo por pantalla la clase real y la predicha en cada uno de los casos.

Imagen real: X -> Predicción: Y

Para ello se **recorre el array de rutas** obtenido mediante la función *getImgsForPrediction()*, y se accede una a una. El proceso por cada imagen es el siguiente:

a. Se **carga la imagen** en una variable mediante la función ***load_img*** de image, a la que se le pasa por parámetro la **ruta de la imagen** y el **tamaño** al que se le quiere transformar (que en nuestro caso es el atributo 'imgSize' del modelo seleccionado del objeto), y devuelve una **imagen PIL** para que sea interpretable por el interpreter de Python.
b. Se le pasa la imagen PIL y el modelo a la función *predictor()* creada anteriormente, la cual devolverá las **probabilidades predichas** mediante el modelo.
c. Se coge el nombre de la imagen a apredecir mediante la función *getImgClassByName()*.
d. Se almacenan las etiquetas en los arrays correspondientes:
    - **y_true**: lista para las etiquetas reales de las imágenes.
    - **y_pred_proba**: lista para la probabilidad máxima predicha de la imagen.
    - **y_pred_label**: lista para almacenar la etiqueta predicha.

```
cnn = load_model(MODEL_SELECTED['modelPath'])
cnn.load_weights(MODEL_SELECTED['weightsPath'])
y_true = []
y_pred_proba = []
y_pred_label = []
test_set = getImgsForPrediction(PREDICT_FOLDER)
for img in test_set:
    imag_to_pred = image.load_img(img, target_size=(MODEL_SELECTED['imgSize'],
    MODEL_SELECTED['imgSize']))
    prob = predictor(imag_to_pred, cnn)[0]
    img_full_name = str(img)
    img_name = getImgClassByName(img_full_name)
    y_true.append(img_name)
    y_pred_proba.append(prob[prob.argmax()])
    y_pred_label.append(LABELS_OBJ[prob.argmax()])
    print(f"Imagen real: {img_name} -> Predicción: {LABELS_OBJ[prob.argmax()]}")
```

Código 46. Clasificador: proceso de clasificación

5. **Análisis de resultados.**

Finalmente, se realiza un **análisis de los resultados** obtenidos en la clasificación mediante la librería de Sklearn, que ofrece funciones muy útiles para ello.

La primera usada es ***classification_report***, la cual recibe por parámetro las **labels reales**, las **predichas** y el **array de posibles clases**. Esta devuelve una tabla con la **precisión**, el **recall** y el **f1-score** de la clasificación.

Con esa tabla se crea un **dataframe** para poder **exportarlo** y guardar los resultados en un archivo externo Excel.

La segunda función usada es ***confusion_matrix*** que, al igual que la anterior, utiliza el array ***y_true***, ***y_pred_label*** y ***LABELS_OBJ***. Devuelve a su vez la **matriz de confusión** que, mediante la función ***ConfusionMatrixDisplay***, se puede desplegar en un gráfico de la librería *matplotlib*.

```
report_to_excel = classification_report(y_true, y_pred_label, labels=LABELS_OBJ,
zero_division=0, output_dict=True)
df = pd.DataFrame(report_to_excel).transpose()
df.to_excel(MODEL_SELECTED['reportPath'])
print(df)
cm = confusion_matrix(y_true=y_true, y_pred=y_pred_label, labels=LABELS_OBJ)
display = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=LABELS_OBJ)
fig, ax = plt.subplots(figsize=(30,30))
display.plot(ax=ax)
plt.savefig(MODEL_SELECTED['confusionMatrixPath'], format='png')
plt.show()
```

Código 47. Clasificador: análisis y extracción de resultados

4.5 Herramientas y recursos utilizados

En este apartado se va a detallar que tipo de herramientas se han utilizado para el desarrollo del proyecto y cual ha sido la finalidad de cada una de ellas. También se van a especificar los recursos de los que se han hecho uso para conseguir llevar a cabo el proyecto.

Se van a dividir en dos grupos ya que las herramientas han aportado las vías necesarias para conseguir desarrollar el proyecto y los recursos han sido los elementos necesarios para que dicho desarrollo se realizara correctamente.

4.5.1 Herramientas

4.5.1.1 Visual Studio Code

Visual Studio Code es una **herramienta de escritorio** muy liviana para el **desarrollo de código** y que consta de muchas **funcionalidades** prácticas de las que el usuario puede hacer uso para amenizar su escritura de código y desarrollo de aplicaciones.

La edición de código no está limitada en esta herramienta, ya que existen lo que se llaman **extensiones** que permiten el uso de cualquier lenguaje disponible (Java, Python, C, C++, HTML, CSS, PHP, Ruby, PowerShell, JSON, …).

Está disponible para **todos los sistemas operativos** (herramienta multiplataforma), y consta de *intellisense,* lo que ayuda a predecir lo que el usuario pretende escribir a medida que programa.

Dentro de este proyecto ha sido esencial, ya que gracias a las extensiones que aporta para crear ***Jupyter Notebooks*** dentro del propio programa se han conseguido los límites de rendimiento que se necesitaban para las pruebas de los modelos que se han ido desarrollando.

Además se ha utilizado también como línea de comandos, ya que el propio Visual Studio Code viene con el terminal que el usuario quiera instalado, y de esta manera poder instalar dentro del ***interpreter*** de Python las dependencias necesarias para trabajar.

4.5.1.2 TensorFlow

TensorFlow es una **plataforma abierta al público** destinada para el **desarrollo del aprendizaje automático**. Ofrece a los usuarioas un ecosistema integral con una gran cantidad de herramientas, bibliotecas y recursos desarrollados por la comunidad con el fin de que desarrolladores e investigadores puedan usarlas de una manera gratuita para sus proyectos de

innovaciones, y de esta manera aumentar el número de desarrollos en aplicación con tecnonolgía de AA.

Entre las herramientas, bibliotecas y recursos que ofrece podemos encontrar:

- Redes neuronales entrenadas.
- Sistemas de recomendación.
- Redes generativas adversarias.
- …

Por último, enfatizar que es una plataforma usada por muchas de las grandes empresas del sector como AirBNB, Googel, Intel, Twitter, Coca-Cola, entre muchos otros .

4.5.1.3 Keras

Keras es una biblioteca destinada al desarrollo de redes neuronales, desarrollada bajo en lenguaje de programación Python e implementada en plataformas como TensorFlow o Theano.

Es una biblioteca muy intuitiva para la iniciación dentro del Deep Learning ya que está diseñado para ir construyendo una red neuronal por bloques, incluyendo todo tipo de redes neuronales como las convolucionales, que se han usado en este proyecto u otras como las redes neuronales recurrentes.

Es de código abierto, por lo que cualquier usuario con un ordenador en mano podrá hacer uso de está biblioteca para comenzar a desarrollar su primera red neuronal.

Finalmente, enfatizar que no funciona como un framework independiente, sino que funciona como una API conectada a los Frameworks de aprendizaje automático de plataformas como TensorFlow o Theano, ya anteriormente mencionados.

4.5.1.4 Google Collab

Google Colab, también conocido como Google Colaboratory) es una herramienta que ha desarrollado Google para poder crear proyectos de programación en el lenguaje de Python dentro de un Jupyter Notebook.

Es una herramienta super sencilla de utilizar ya que simplemente teniendo una cuenta de Google e iniciando sesión, ya tienes acceso a una PaaS para comenzar a desarrollar en Pyhton.

Una de las ventajas que nos ha aportado a este proyecto ha sido su facilidad de uso, sobretodo para el comienzo de desarrollo ya que viene con TensorFlow preinstalado, por lo que no ha sido necesaria su descarga e instalación en la máquina.

De todas formas también soporta lenguaje de consola, por lo que en el caso de querer instalar alguna dependencia más para usarla en el proyecto, simplemente con poner una exclamación antes de la línea de comandos, el propio Google Colab te lo detecta como tal y lo ejecuta sin problemas.

Finalmente, otra de las grandes ventajas es que está asociado al Google Drive, por lo que se pueden almacenar tantos proyectos de Jupyter Notebook en el Drive como espacio se tenga, además de poder acceder a través del Google Collab a archivos almacenados en el Drive otorgándole una serie de permisos.

4.5.1.5 Pandas

Pandas es una librería especializada para el **manejo** y **análisis** de estructuras. Es una librería que implementa una cantidad de opciones para el manejo, almacenamiento y análisis de datos como la implementación de nuevas **estructuras de datos** basadas en los arrays de la librería de Numpy, métodos de **lectura** y **escritura** en ficheros Excel, CSV y SQL, **acceso** a datos mediante **índices** de columnas y filas, entre muchas otras características.

En este proyecto se ha usado para extraer las métricas de los modelos desarrollados en un Excel. De esta manera se han podido adaptar de la mejor forma posible para una mejor comprensión de los datos.

4.5.1.6 MatplotLib

Matplotlib es una librería desarrollada en Python para la **visualización de gráficas de datos**.

Ofrece una API muy extensa de gráficas, pudiendo usar gráficas de barras, histogramas, de líneas, de áreas, entre muchas otras.

También viene con muchas otras funciones para adaptar las gráficas a las necesidades que tenga el usuario, es decir, opciones como meter varias gráficas en el mismo frame, incluir una gráfica aparte con sus ejes propios en el mismo frame, añadir leyenda, añadir título, y muchas opciones más.

En este proyecto se ha utilizado para mostrar los valores de las métricas en cada una de las épocas de entrenamiento y de esta manera poder analizar de una manera más visual como ha ido el proceso de entrenamiento y detectar anomalías o comportamientos extraños.

También se ha utilizado para guardar en una imagen PNG la matriz de confusión del resultado de predicción de cada uno de los modelos.

4.5.1.7 Sklearn

ScikitLearn es una librería desarrollada bajo el lenguaje de programación Python que se usa principalmente para dar solución **problemas relacionados con el data science**, es decir, problemas de análisis de datos y modelado estadístico.

La librería trae muchos **algoritmos** de clasificación, regresión, clustering y reducción de dimensionalidad.

Además tiene **compatibilidad** con otras librerías que se usan en el mismo ámbito como NumPy, SciPy y Matplotlib, las cuales se han usado también en este proyecto.

Dentro de este proyecto, la librería de ScikitLearn se ha utilizado para la extracción de datos en la fase de predicción. Además, aparte de las métricas obtenidas de los modelos, tambíen se ha utilizado para la visualización de la matriz de confusión de cada uno de ellos y de esta forma ver de una manera más visual el resultado de la predicción.

4.5.2 Recursos

4.5.2.1 Dataset de imágenes

El dataset que se nos ha proporcionado por parte de IBM consta de un dataset común de 1540 imágenes repartidas en **22 clases** (70 imágenes por clase) más una clase adicional independiente con 70 imágenes para cada alumno, completando así el dataset con **1610 imágenes** correspondientes a las **23 clases** existentes.

Estas clases corresponden a 23 marcas conocidas y las imágenes que asociadas a cada una de estas marcas son sus correspondientes logos de visualizados de distintas maneras.

La lista de marcas comunes que nos encontramos es: **AMD**, **Aquafina**, **D-link**, **Disney**, **Domino's Pizza**, **Hellmann's**, **IBM**, **KitKat**, **LG**, **Lipton**, **McDonalds**, **Milka**, **Monster**, **Nestea**, **Pac-Man**, **Pepsi**, **Pizza Hut**, **Red Bull**, **Samsung**, **Sony**, **Tic Tac** y **Universal**.

A esta lista de marcas hay que añadirle la individual, que en mi caso ha sido **Nintendo**.

4.5.2.2 Componentes físicos

- Ordenador de sobremesa:
  - Procesador Intel® Core™ i7-9700K CPU 3.60GHz
  - Memoria RAM de 32 GB
  - Sistema Operativo Windows 10 Pro de 64 bits.
  - Tarjeta Gráfica Nvidia GTX 2080 Ti XTREME 11GB GDDR6
- Dos monitores:
  - DELL S2715H 27" (1920x1080).
  - DELL P2720DC 27" (2560x1440).

4.6 Presupuesto

Mediante este apartado se pretende estimar de la manera más acertada cual sería el presupuesto necesario para el desarrollo del proyecto en el caso en el que se necesitara financiación.

Para ello se va a hacer uso de los siguientes datos:

- El **tiempo** utilizado para el desarrollo del proyecto.
- El **material usado** y los **recursos consumidos** para ello.

En el primero caso, en referencia al **tiempo utilizado**, se va a coger de referencia el diagrama Gantt mostrado en el apartado *Planificación del proyecto*, en el que se muestra un pequeño cronograma junto a las actividades realizadas durante el desarrollo del proyecto.

El propio diagrama viene con una tabla con el total de horas trabajadas:

Tiempo medio trabajado por día	1,5
Días totales de proyecto	157
Horas totales invertidas en el proyecto	235,5

Como se puede observar, el total de horas trabajadas para el proyecto han sido de **235,5 horas**.

Suponiendo que el trabajador es un ingeniero infomático con poca experiencia, especializado en el campo de la inteligencia artificial y al cual se le paga **8 €/hora**, el total del gasto simplemente para el salario del trabajador será de **1884 €** totales.

Respecto a los materiales usados, descritos en el apartado *Componentes físicos*, se realizará un conteo del precio gastado en cada uno de ellos.

- Ordenador de sobremesa:
  - Procesador Intel® Core™ i7-9700K CPU 3.60GHz: **369 €**
  - Memoria RAM de 32 GB: **194,99 €**
  - Sistema Operativo Windows 10 Pro de 64 bits: **10 €**
  - Tarjeta Gráfica Nvidia GTX 2080 Ti XTREME 11GB GDDR6: **1.169 €**
  - Extras (fuente alimentación, carcasa, placa base, …): **655 €**
- Dos monitores:
  - DELL P2720DC 27” (2560x1440): **524 €**
  - DELL S2721QS 27” 4K: **369,99 €**

Además habría que añadir el extra de gastos del Wi-Fi y la luz, que redondeando durante los dos meses han sido:

- Wi-Fi: **30 €**
- Luz: **20 €**

Por lo que el total necesario para cubrir todos los gastos mínimos del proyecto son **3.342 €** aproximadamente.

4.7 Estructura final del proyecto

En este apartado se va a describir al detalle la estructura del proyecto, incluyendo todas las carpetas que hay en él y que tipo de archivos se almacenan en cada una de ellas.

La estructura final del proyecto es la siguiente:

```
|-- confusion_matrix_images. Carpeta con las matrices de confusión obtenidas
|      |                     en cada una de las clasificaciones realizadas con
|      |                     los modelos.
|      |-- classifier_from_scratch_output.png
|      |-- …
|
|-- models. Carpeta con los modelos entrenados.
|      |-- classifier_from_scratch_model.hdf5
|      |-- …
|
|-- reports. Carpeta con las métricas obtenidas de la clasificación de cada
|      |      uno de los modelos utilizados.
|      |-- f1-score_img. Carpeta con las gráficas respecto al f1-score obtenido
|      |      |          en la clasificación de cada uno de los modelos.
|      |      |-- Classifier_from_scratch_f1-score.png
|      |      |-- …
|      |-- precision_img. Carpeta con las gráficas respecto a la exactitud
|      |      |           obtenida en la clasificación de cada uno de los
|      |      |           modelos.
|      |      |-- Classifier_from_scratch_precision.png
|      |      |-- …
|      |-- recall_img. Carpeta con las gráficas respecto al recall obtenido en
|      |               la clasificación de cada uno de los modelos.
|             |-- Classifier_from_scratch_recall.png
|             |-- …
|
|-- splitted_data. Carpeta con las imágenes de entrenamiento, validación y
|      |           clasificación.
|      |-- Entrenamiento. Carpeta con las imágenes usadas para la fase de
|      |                  entrenamiento.
|      |-- preview. Carpeta con ejemplos de imágenes generadas por el
|      |            ImageDataGenerator.
|      |-- Validacion. Carpeta con las imágenes para la fase de validación.
|      |-- Clasificación: Carpeta con las imágenes usadas para la
|                         clasificación.
|
|-- weigths. Carpeta con los pesos obtenidos en los entrenamientos de cada uno
|      |     de los modelos.
|      |-- classifier_from_scratch_weights.hdf5
|      |-- …
|
|-- [0] Prediction.ipynb
|-- [1] Classifier_from_scratch.ipynb
|-- [2.1] ResNet101V2_classifier.ipynb
|-- [2.2] ResNet101V2_classifier.ipynb
|-- ….
```

4.8 Resultados del proyecto

En este apartado se van a analizar cada uno de los **resultados obtenidos** tanto en el **entrenamiento** como en la **clasificación** de los modelos creados a lo largo del desarrollo del proyecto y, posteriormente, se realizará una pequeña comparación entre las métricas obtenidas en la clasificación con cada uno de ellos.

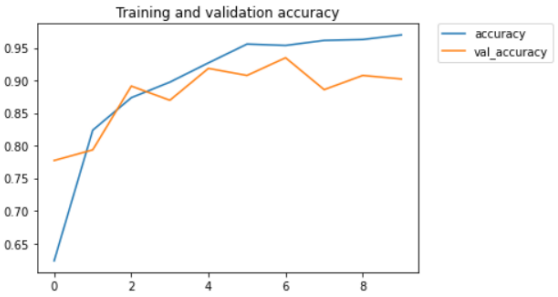
4.8.1 Métricas de entrenamiento

En cada fase de entrenamiento de los modelos se han obtenido dos **gráficos lineares** para mostrar los valores de **exactitud** y **error** en cada una de las épocas de las fases de entrenamiento y validación para comparar ambas fases.

Estas gráficas sobre todo se han usado para identificar *overfitting* en el entrenamiento del modelo y de esta manera poder evitarlo cambiando parámetros como el ***learning rate***, modificando el nº de épocas de entrenamiento o aplicando cualquiera de las técnicas explicadas en el punto *"4.3.9.2. ¿Cómo se soluciona el overfitting?"*.

Únicamente se va a describir los resultados del entrenamiento del modelo con los mejores resultados: **ResNet101v2**.

Las imágefnes referentes a los entrenamientos del resto de los modelos se adjuntarán en el apartado *"Capítulo 8. ANEXOS"*.



En este caso se puede observar como en la gráfica de error, los errores siguen una buena dinámica hasta la época 6, descenciendo ambos regularmente.

En cambio, en la época 7 se produce un aumento bastante significativo en el error de validación, lo que da suponer que a partir de esa época se va a producir sobreajuste.

Para evitarlo, se ha hecho uso de varios callbacks que ofrece Keras:

- ***EarlyStopping***. Callback que para la ejecución del entrenamiento cuando una métrica a dejado de mejorar. En este caso se ha usado como métrica el val_loss.
- ***ModelCheckPoint***. Callback que se encarga de guardar los pesos en cualquier momento del entrenamiento según los parámentros que se le indique. En este caso, se da el caso que el error de validación más bajo se produce en la época 6, por lo que los pesos almacenados para la clasificación son los generados en esa época.

4.8.2 Resultados de la clasificación

Para ello se ha hecho uso de las imágenes separadas para la validación, aplicandolas la clasificación de cada uno de los modelos para obtener las métricas como si fuera un caso de uso real haciendo uso del fichero ***Prediction.ipynb***, explicado en el apartado "*4.4.4. Clasificador"* del documento.

La idea es primero mostrar los resultados de los modelos con los que peores métricas se han obtenido para, finalmente, mostrar el modelo con los mejores resultados, el cual va a ser el que se va a entregar a IBM como modelo final.

1. **Classifier_from_scratch.**

Tabla 1. Reporte Classifier_from_scratch

Clase	precision	recall	f1-score	support
AMD	0	0	0	8
Aquafina	0,25	0,125	0,166667	8
D-link	0,7	0,875	0,777778	8
Disney	0,142857	0,125	0,133333	8
Domino_s Pizza	0,777778	0,875	0,823529	8
Hellmann_s	0,583333	0,875	0,7	8
IBM	0,25	0,625	0,357143	8
Kitkat	1	0,375	0,545455	8
LG	0,5	0,125	0,2	8
Lipton	0,625	0,625	0,625	8
McDonalds	0,230769	0,75	0,352941	8
Milka	1	0,5	0,666667	8
Monster	0,375	0,375	0,375	8
Nestea	0,538462	0,875	0,666667	8
Nintendo	0,1875	0,375	0,25	8
PacMan	0,8	0,5	0,615385	8
Pizza Hut	1	0,625	0,769231	8
Red Bull	0,5	0,875	0,636364	8
Tic Tac	0,6	0,375	0,461538	8
Universal	0	0	0	8
Pepsi	0,833333	0,625	0,714286	8
samsung	0,666667	0,25	0,363636	8
sony	0,333333	0,125	0,181818	8
accuracy	0,472826	0,472826	0,472826	0,472826
macro avg	0,517132	0,472826	0,45141	184
weighted avg	0,517132	0,472826	0,45141	184

2. *Transfer learning*: **Inception_v3.**

Tabla 2. Reporte Inception_v3

Clase	precision	recall	f1-score	support
AMD	0,777778	0,875	0,823529	8
Aquafina	0,7	0,875	0,777778	8
D-link	1	1	1	8
Disney	0,583333	0,875	0,7	8
Domino_s Pizza	1	1	1	8
Hellmann_s	1	0,875	0,933333	8
IBM	1	0,875	0,933333	8
Kitkat	1	0,875	0,933333	8
LG	0,75	0,75	0,75	8
Lipton	0,545455	0,75	0,631579	8
McDonalds	0,5	0,5	0,5	8
Milka	1	0,625	0,769231	8
Monster	1	0,875	0,933333	8
Nestea	0,875	0,875	0,875	8
Nintendo	0,8	1	0,888889	8
PacMan	0,75	0,75	0,75	8
Pizza Hut	0,875	0,875	0,875	8
Red Bull	0,857143	0,75	0,8	8
Tic Tac	0,833333	0,625	0,714286	8
Universal	0,75	0,75	0,75	8
Pepsi	0,75	0,75	0,75	8
samsung	1	0,875	0,933333	8
sony	0,714286	0,625	0,666667	8
accuracy	0,809783	0,809783	0,809783	0,809783
macro avg	0,828753	0,809783	0,812549	184
weighted avg	0,828753	0,809783	0,812549	184

3. *Transfer learning*: **NasNetLarge.**

Tabla 3. Reporte NasNetLarge

Clase	precision	recall	f1-score	support
AMD	0,7	0,875	0,777778	8
Aquafina	0,875	0,875	0,875	8
D-link	1	1	1	8
Disney	0,625	0,625	0,625	8
Domino_s Pizza	1	1	1	8
Hellmann_s	1	1	1	8
IBM	1	0,75	0,857143	8
Kitkat	0,875	0,875	0,875	8
LG	0,857143	0,75	0,8	8

Lipton	0,875	0,875	0,875	8
McDonalds	1	0,5	0,666667	8
Milka	0,625	0,625	0,625	8
Monster	0,777778	0,875	0,823529	8
Nestea	1	0,875	0,933333	8
Nintendo	0,875	0,875	0,875	8
PacMan	0,777778	0,875	0,823529	8
Pizza Hut	0,8	1	0,888889	8
Red Bull	0,777778	0,875	0,823529	8
Tic Tac	0,8	1	0,888889	8
Universal	0,8	1	0,888889	8
Pepsi	1	0,375	0,545455	8
samsung	1	0,875	0,933333	8
sony	0,636364	0,875	0,736842	8
accuracy	0,836957	0,836957	0,836957	0,836957
macro avg	0,855515	0,836957	0,832079	184
weighted avg	0,855515	0,836957	0,832079	184

4. *Transfer learning*: **ResNet50.**

Tabla 4. Reporte ResNet50

Clase	precision	recall	f1-score	support
AMD	0,636364	0,875	0,736842	8
Aquafina	0,636364	0,875	0,736842	8
D-link	1	1	1	8
Disney	0,75	0,75	0,75	8
Domino_s Pizza	1	0,875	0,933333	8
Hellmann_s	1	1	1	8
IBM	1	0,75	0,857143	8
Kitkat	1	1	1	8
LG	0,875	0,875	0,875	8
Lipton	0,857143	0,75	0,8	8
McDonalds	0,857143	0,75	0,8	8
Milka	1	0,75	0,857143	8
Monster	1	1	1	8
Nestea	1	1	1	8
Nintendo	0,875	0,875	0,875	8
PacMan	1	0,75	0,857143	8
Pizza Hut	0,888889	1	0,941176	8
Red Bull	0,857143	0,75	0,8	8
Tic Tac	1	1	1	8
Universal	0,875	0,875	0,875	8
Pepsi	0,714286	0,625	0,666667	8
samsung	0,777778	0,875	0,823529	8

sony	0,636364	0,875	0,736842	8
accuracy	0,86413	0,86413	0,86413	0,86413
macro avg	0,879847	0,86413	0,866159	184
weighted avg	0,879847	0,86413	0,866159	184

5. *Transfer learning*: **ResNet101.**

Tabla 5. Reporte ResNet101

Clase	precision	recall	f1-score	support
AMD	1	0,875	0,933333	8
Aquafina	1	0,875	0,933333	8
D-link	1	1	1	8
Disney	0,875	0,875	0,875	8
Domino's Pizza	1	1	1	8
Hellmann's	1	1	1	8
IBM	0,666667	1	0,8	8
Kitkat	1	1	1	8
LG	0,857143	0,75	0,8	8
Lipton	1	1	1	8
McDonalds	0,888889	1	0,941176	8
Milka	1	1	1	8
Monster	1	1	1	8
Nestea	1	1	1	8
Nintendo	1	1	1	8
PacMan	1	0,75	0,857143	8
Pizza Hut	1	0,875	0,933333	8
Red Bull	1	0,875	0,933333	8
Tic Tac	1	1	1	8
Universal	1	0,625	0,769231	8
Pepsi	1	1	1	8
samsung	0,875	0,875	0,875	8
sony	0,615385	1	0,761905	8
accuracy	0,929348	0,929348	0,929348	0,929348
macro avg	0,946873	0,929348	0,930991	184
weighted avg	0,946873	0,929348	0,930991	184

A continuación se realizará una **comparación entre las métricas obtenidas** con cada uno de los modelos, definiendo la métrica y finalmente indicando el modelo con el mejor valor.

El orden de las clases dentro de las gráficas es igual al orden mostrado en las anteriores tablas:

```
['AMD', 'Aquafina', 'D-link', 'Disney', 'Domino's Pizza','Hellmann's', 'IBM', 'Kitkat',
'LG', 'Lipton', 'McDonalds', 'Milka', 'Monster', 'Nestea', 'Nintendo', 'PacMan', 'Pizza
        Hut', 'Red Bull', 'Tic Tac', 'Universal', 'Pepsi', 'samsung', 'sony']
```

COMPARACIÓN de la *"precision"* y "*accuracy*" entre modelos

En esta comparación se muestran las gráficas de la **precisión** y la **exactitud** de los modelos para clasificar las clases correctamente.

Para entender mejor el concepto **precisión**, tal y como se ha descrito en el punto "*Métricas para la medición de resultados del Deep Learning"*, corresponde al **número de ejemplos de la clase clasificados correctamente** sobre el número total de ejemplos clasificados de esa clase, y el concepto de **exactitud** corresponde al **número de aciertos**, es decir, a la suma de TP y TN entre el número total de ejemplos.

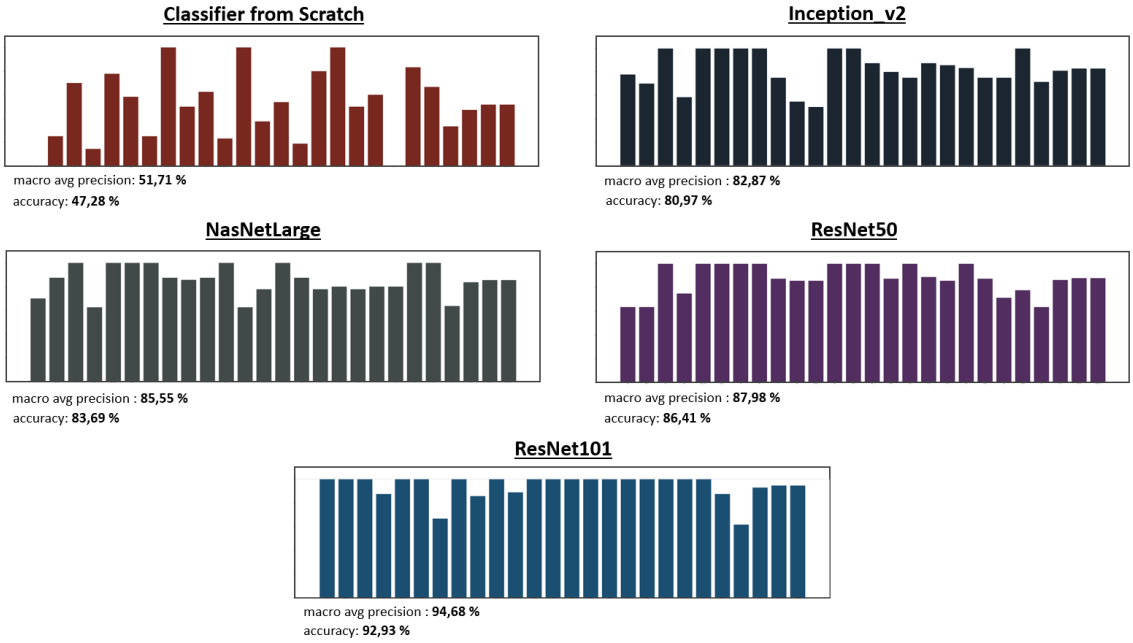


Figura 28. Gráficas "precision" & "accuracy"

Como se puede observar, de esta manera se puede detectar perfectamente cual es el modelo que mejor exactitud tiene de los 5, siendo la red neuronal creada a partir de la **ResNet101**, con una exactitud del **93,68 %**.

COMPARACIÓN de la *"recall"* entre modelos

En esta comparación se muestran las gráficas creadas mediante el **recall** de las clasificaciones por clase, añadiéndole el **recall medio global** de cada uno de los modelos utilizados.

Para hacer memoria, en el punto "*4.3.8. Métricas para la medición de resultados del Deep Learning"* se define como el **número de ejemplos de la clase clasificados correctamente** sobre el número total de la clase.

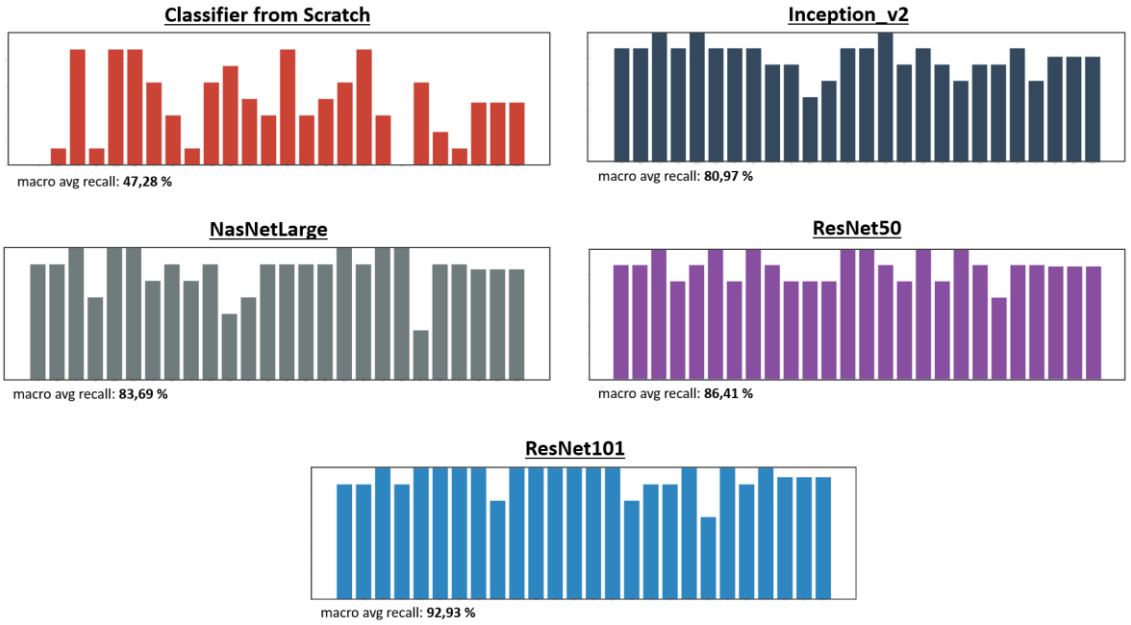


Figura 29. Gráficas "recall"

Viendo los resultados obtenidos en las gráficas, al igual que en la precisión, el modelo con mayor **recall** es el modelo desarrollado bajo el **ResNet101**.

COMPARACIÓN de la *"f1-score"* entre modelos

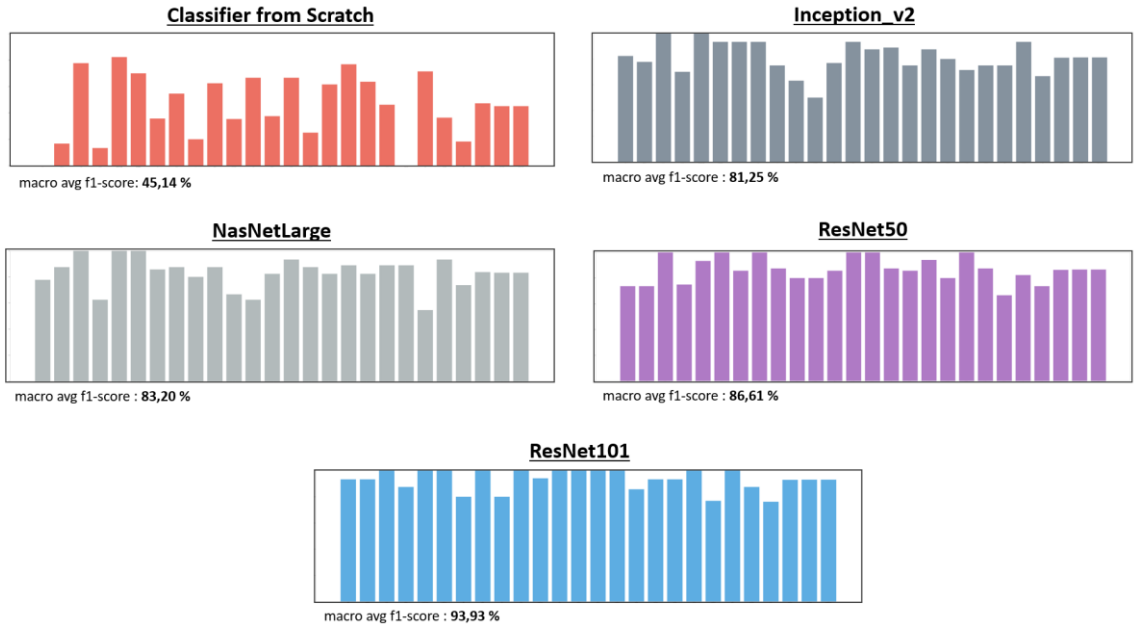


Figura 30. Gráficas "f1-score"

Al igual que en las dos anteriores métricas, el mejor resultado corresponde al modelo desarrollado bajo el modelo preentrenado de **ResNet101**, siendo un **f1-score** de **92,54 %**.

Para finalizar, el modelo elegido para entrega a IBM ha sido sin duda alguna el desarrollado bajo el modelo preentrenado **ResNet101**, consiguiendo unas métricas excepcionales con respecto al

resto de los modelos creados. La mátriz de confusión obtenida en las pruebas de predicción del modelo ResNet101 se muestra en la *Figura 31. Matriz confusión ResNet101*.

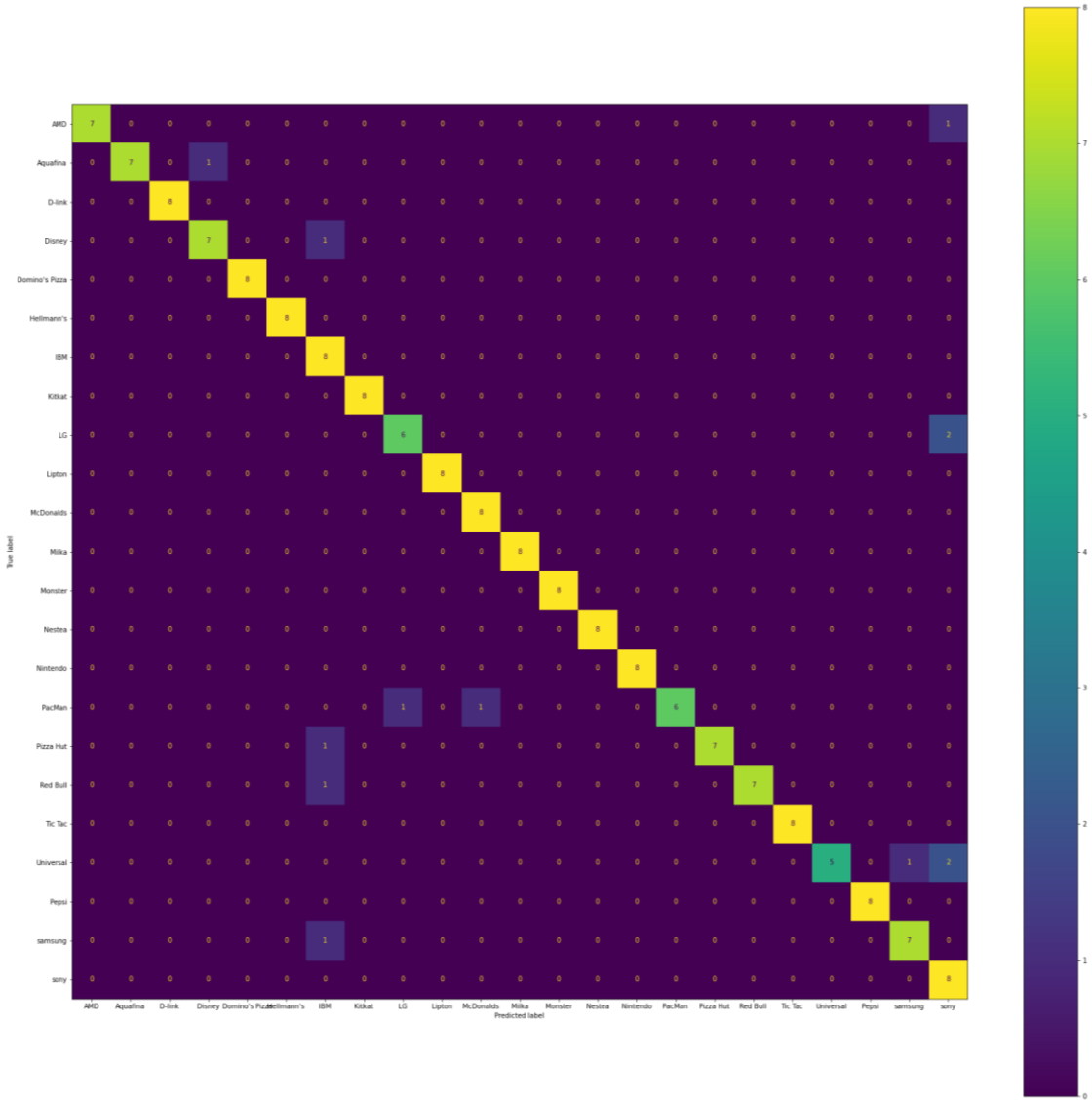


Figura 31. Matriz confusión ResNet101

Capítulo 5. CONCLUSIONES

5.1 Conclusiones del trabajo

Tras la realización del proyecto, una de las grandes conclusiones del trabajo es que la complicación de crear una red neuronal no se encuentra a la hora de construirla, ya que la propia librería que se usa te da todas las herramientas necesarias para ello, sino que lo complicado es **darle solución a problemas** como el *overfitting* y *underfitting*, **ajustar correctamente los hiperparámetros** de la red neuronal y realizar un correcto **tratamiento de los datos**.

Además, se ha querido explicar el Deep Learning desde 0 para poder poner en contexto al lector y de esta manera se pueda entender al 100% el desarrollo del proyecto, también explicado paso a paso para poder seguirlo sin problemas.

También se ha hecho uso de muchas **gráficas** e **imágenes explicativas**.

Las **gráficas** han ayudado en gran parte a **detectar anomalías** dentro de la fase de entrenamiento y validación de los modelos, además de **identificar** más rápidamente cual de los modelos entrenados ha sido el que **mejores resultados** ha obtenido en la fase de clasificación.

Las **imágenes explicativas** dispersadas por todo el documento han ayudado, en gran parte, a **de una manera más clara** los conceptos necesarios para el entendimiento del proyecto y cada una de las partes de la fase de desarrollo.

Finalmente, es un trabajo que ha llevado su tiempo finalizar, ya que, al ser tantos modelos que probar, el **proceso de testeo** de cada uno de ellos ha dependido en gran parte de los recursos computacionales utilizados, ya que ha sido necesario entrenarlos varias veces para visualizar las diferencias entre las métricas de los entrenamientos. Para ello es imprescindible tener una buena computadora o alquilar una gran cantidad recursos computacionales en la nube para minimizar lo máximo posible el tiempo de espera de entrenamiento y validación de los modelos.

5.2 Conclusiones personales

Personalmente, la realización de este proyecto me ha introducido al mundo de la inteligencia artificial, ya que nunca antes lo había estudiado.

La parte más complicada fue la introducción, ya que, al no haber tocado ni leído nunca sobre la inteligencia artificial, no tenía ningún tipo idea de como podía ser implementada en problemas reales.

Según avanzaba con el desarrollo del proyecto, el entendimiento sobre el tema y la rapidez de desarrollo fueron aumentando exponencialmente, consiguiendo finalmente un proyecto que, además de ofrecer un modelo con una exactitud, recall y f1-score bastante buenos, aporta al usuario todo tipo de métricas y ficheros para el análisis de los modelos implementados, dando de esta manera pruebas de por qué la elección de un modelo y no de otro.

Para finalizar, estoy bastante satisfecho con el trabajo realizado, y encantado de haberme introducido en un campo que da tantas posibilidades como lo es el campo de la inteligencia artificial.

Capítulo 6. FUTURAS LINEAS DE TRABAJO

Una vez desarrollado el proyecto, surgen una serie de **posibles futuras líneas de trabajo** para mejorar tanto el proyecto en sí como nuevos futuros proyectos.

Entre estas líneas de trabajo encontramos:

- **Aumento del número de clases a clasificar.**
  Esta línea de trabajo es bastante simple, ya que únicamente habría que obtener el mismo número de imágenes por cada clase a mayores y reentrenar el modelo.

- **Testeo del modelo para su posible mejora.**
  Cuantas más pruebas y mas casos de entrenamiento se hagan, mas se ajustarán los hiperparámetros de los modelos desarrollados, consiguiendo mejores resultados.

- **Aumento del tamaño del dataset para la mejora del modelo**
  Cuanto mayor sea el dataset de imágenes, más ejemplos tendremos y por lo tanto más y mejor podrán entrenar los modelos creados.
  Esta línea tambien es relativamente fácil ya que, al ser un proyecto destinado para una empresa tan grande como lo es IBM, no creemos que haya ningún problema con el obtención de imágenes nuevas para el aumento del tamaño del dataset, tanto en cantidad de imágenes como en el número de clases.

- **Creación de un modelo desde 0 más eficaz.**
  Si se produce el aumento de imágenes del dataset, se puede dar el caso en el que salgan mejores resultados con un modelo específico para clasificar las clases obtenidas, en vez de usar *transfer learning*, ya que éste se suele usar generalmente cuando el tamaño del dataset es demasiado pequeño para entrenar una red neuronal desde 0.

- **Otros casos de uso con clases distintas.**
  Se puede dar el caso de que se necesite un modelo para otras clases totalmente distintas. La idea es tomar como base el modelo creado para este proyecto para ir adpatando el nuevo modelo a las nuevas clases a clasificar.

Capítulo 7. BIBLIOGRAFÍA

Advanced Guide to Inception v3 on Cloud TPU . (s.f.). Obtenido de Google Cloud: https://cloud.google.com/tpu/docs/inception-v3-advanced

Blanco, E. (25 de Junio de 2021). *¿Cómo funciona el algoritmo Backpropagation en una Red Neuronal? - Think Big Empresas*. Obtenido de https://empresas.blogthinkbig.com/como-funciona-el-algoritmo-backpropagation-en-una-red-neuronal/

Brownlee, J. (5 de Julio de 2019). *A Gentle Introduction to Pooling Layers for Convolutional Neural Networks*. Obtenido de https://machinelearningmastery.com/pooling-layers-for-convolutional-neural-networks/

Calvo, D. (10 de Diciembre de 2018). *Función de coste – Redes neuronales*. Obtenido de https://www.diegocalvo.es/funcion-de-coste-redes-neuronales/

Chaos, I. (s.f.). *Backpropagation | Interactive Chaos*. Obtenido de https://interactivechaos.com/es/manual/tutorial-de-machine-learning/backpropagation

Chaos, I. (s.f.). *El Perceptrón Multicapa*. Obtenido de https://interactivechaos.com/es/manual/tutorial-de-deep-learning/el-perceptron-multicapa

CSV, D. (12 de Noviembre de 2020). Obtenido de https://www.youtube.com/watch?v=V8j1oENVz00&list=PL-Ogd76BhmcBaUXZGPJkmQpLgrBgGZ7v0

Durán, J. (4 de Septiembre de 2019). *MetaDatos*. Obtenido de Todo lo que Necesitas Saber sobre el Descenso del Gradiente Aplicado a Redes Neuronales: https://medium.com/metadatos/todo-lo-que-necesitas-saber-sobre-el-descenso-del-gradiente-aplicado-a-redes-neuronales-19bdbb706a78

Elgamal, M. (3 de Noviembre de 2013). *Automatic Skin Cancer Images Classification.* Obtenido de http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.299.3590&rep=rep1&type=pdf

Elgamal, M. (s.f.). *AUTOMATIC SKIN CANCER IMAGES.* Obtenido de http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.299.3590&rep=rep1&type=pdf

El-Sayed A. El-Dahshan, T. M.-B. (s.f.). *Hybrid intelligent techniques for MRI brain images classification.* Obtenido de https://www.sciencedirect.com/science/article/abs/pii/S1051200409001377

Hammel, B. (23 de Marzo de 2019). *B. D. Hammel*. Obtenido de What learning rate should I use?: http://www.bdhammel.com/learning-rates/

Ian L. Thomas, V. M. (12 de Septiembre de 2008). *Classification of remotely sensed images.* Obtenido de tandfonline.com/doi/abs/10.1080/01060487093541l3?journalCode=tgei20

Lin Hong, A. J. (s.f.). *Classification of Fingerprint Images.* Obtenido de https://www.researchgate.net/profile/Lin-Hong-12/publication/2929466_Classification_of_Fingerprint_Images/links/54733a5a0cf216f8cfaeb8b7/Classification-of-Fingerprint-Images.pdf

M. (13 de Octubre de 2021). *Overfitting. Qué es, causas, consecuencias y cómo solucionarlo.* Obtenido de https://protecciondatos-lopd.com/empresas/overfitting/

N. (25 de Junio de 2020). *Convolutional Neural Networks: La Teoría explicada en Español.* Obtenido de Aprende Machine Learning: https://www.aprendemachinelearning.com/como-funcionan-las-convolutional-neural-networks-vision-por-ordenador/

Sossa. (23 de Mayo de 2021). *Tipos de redes neuronales (Clasificación)*. Obtenido de https://inteligencia-artificial.dev/tipos-redes-neuronales/

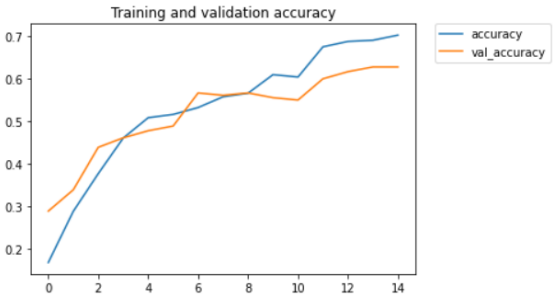
Team, K. (s.f.). *Keras documentation: Callbacks API*. Obtenido de https://keras.io/api/callbacks/

Torres, J. (22 de Marzo de 2021). *Redes Neuronales Recurrentes*. Obtenido de https://torres.ai/redes-neuronales-recurrentes/

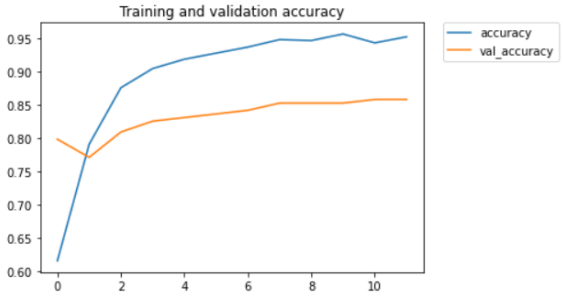
Capítulo 8. ANEXOS

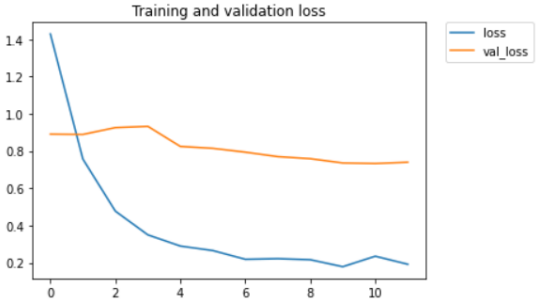
8.1 Resultados entramiento

8.1.1 Classifier_from_scratch

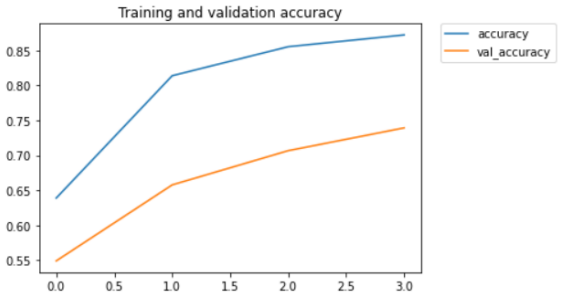


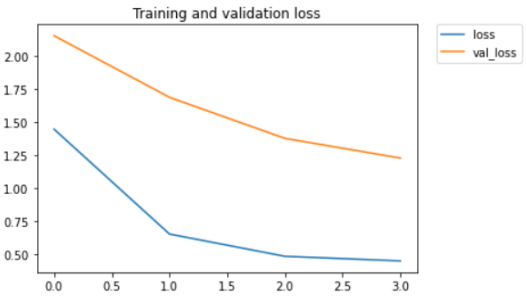
8.1.2 Transfer learning: Inception_v3



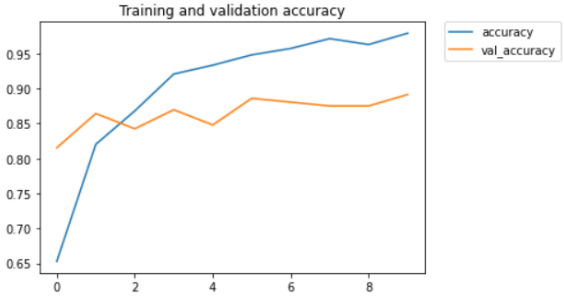


8.1.3 Transfer learning: NasNetLarge



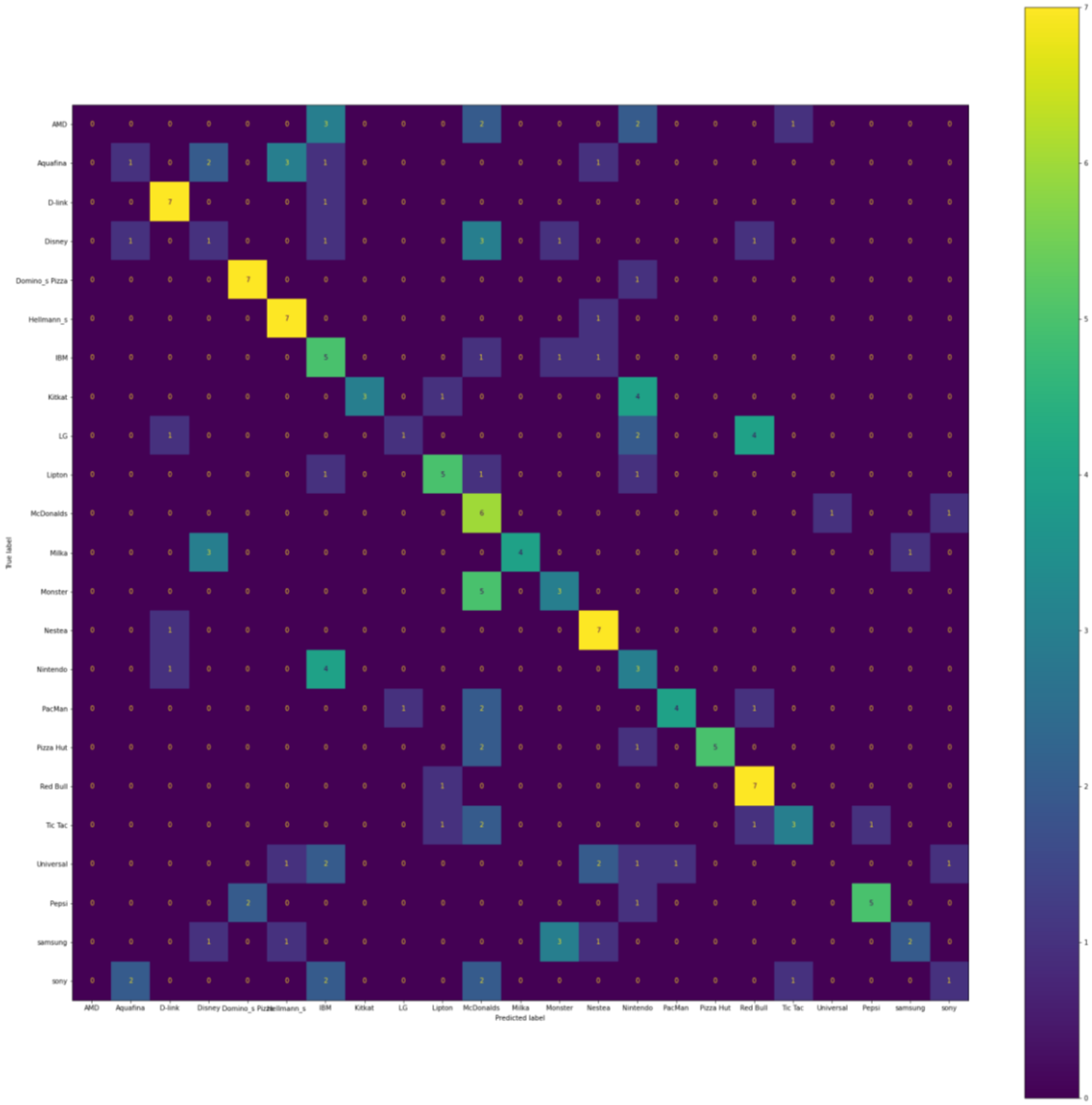


8.1.4 Transfer learning: ResNet50V2

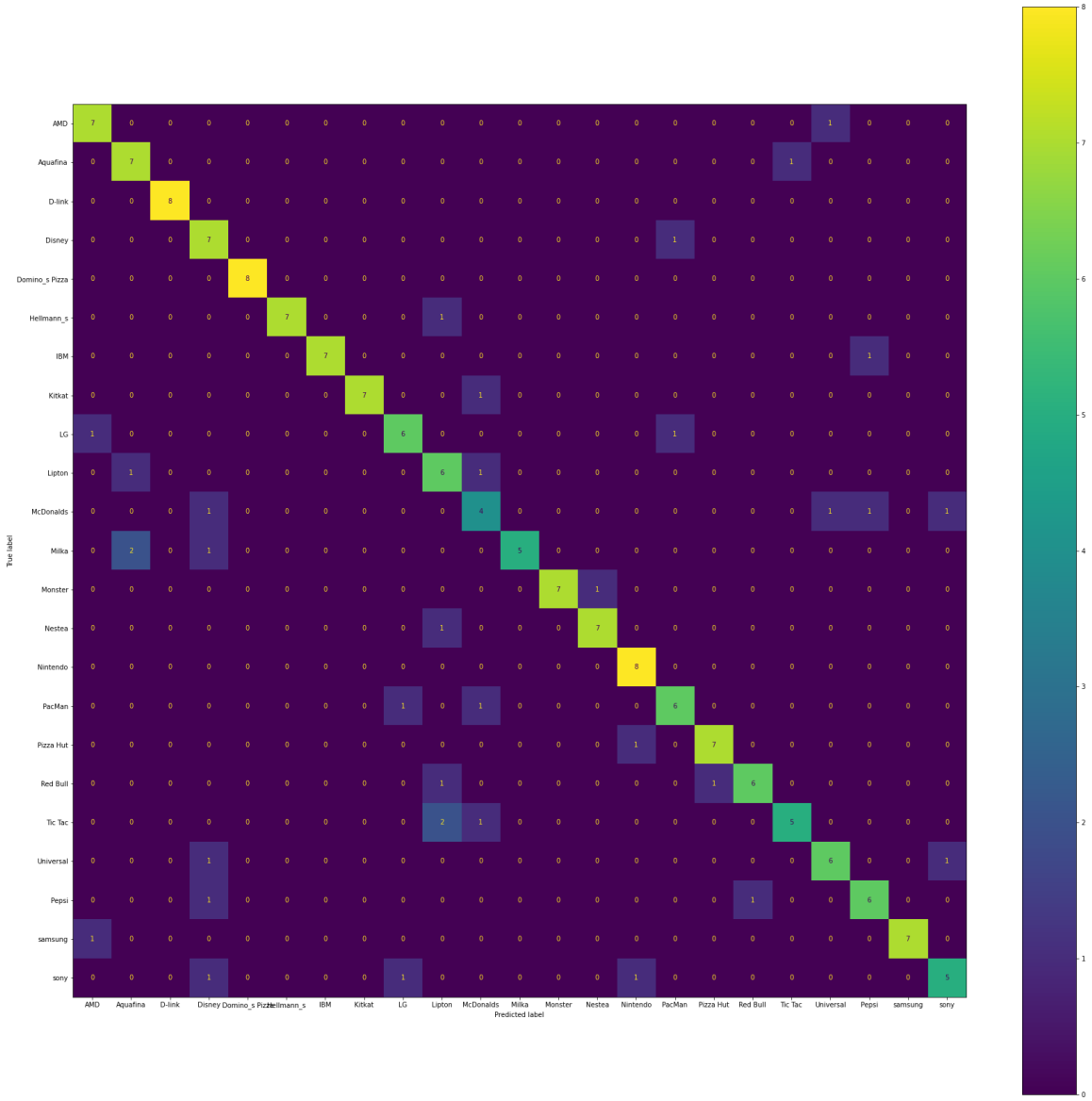


8.2 Matrices de confusión

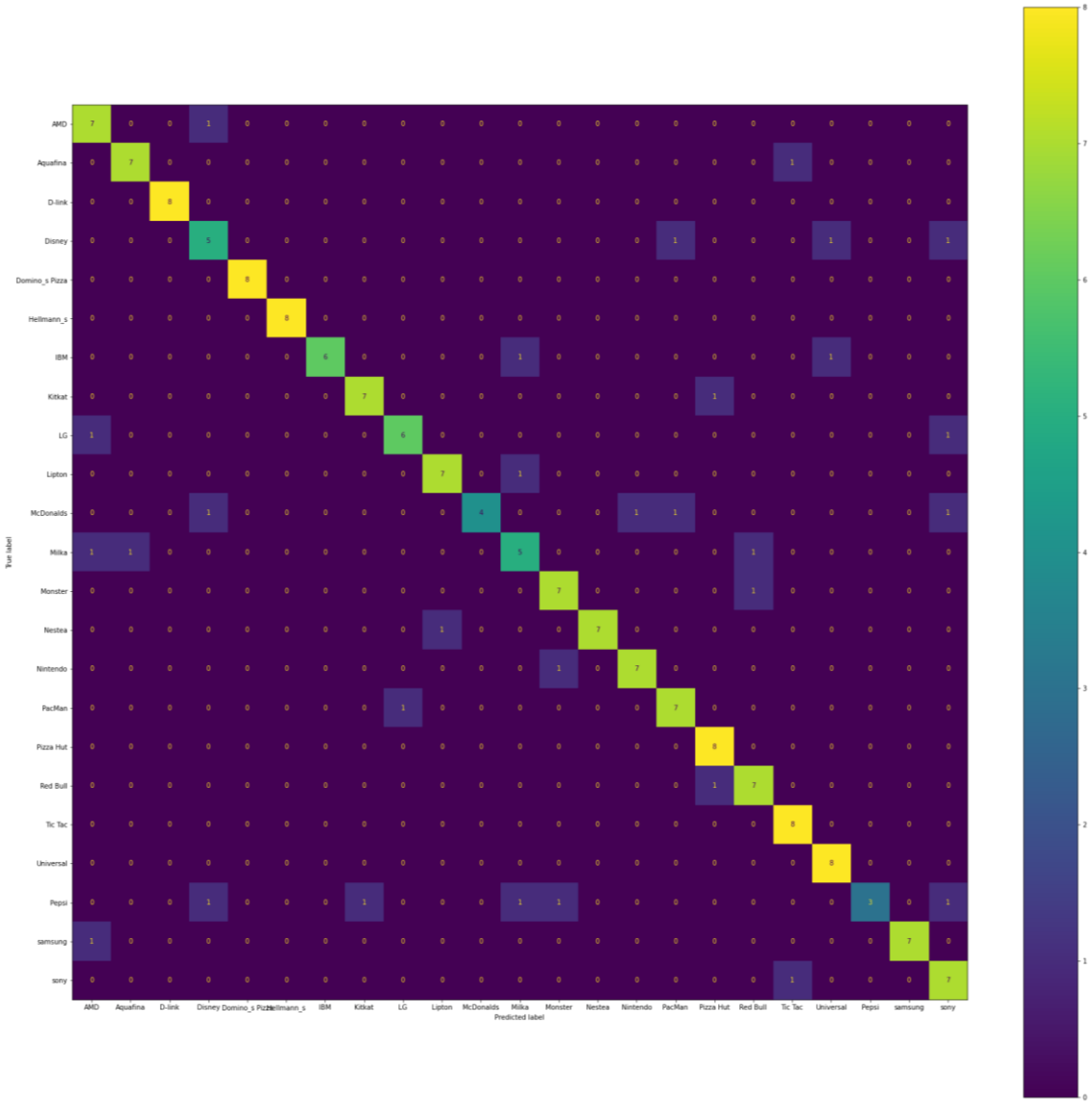
8.2.1 Classifier_from_scratch



8.2.2 Transfer learning: Inception_v3



8.2.3 Transfer learning: NasNetLarge



8.2.4 Transfer learning: ResNet50V2