UNIVERSIDAD EUROPEA DE MADRID

ESCUELA DE ARQUITECTURA, INGENIERÍA Y DISEÑO

MÁSTER UNIVERSITARIO EN

BIG DATA ANALYTICS - MBI

TRABAJO FIN DE MÁSTER

Reconocimiento de logotipos de marcas mediante Redes Neuronales de Convolución (CNN)

NOMBRE:

RAÚL CASTILLA BRAVO

CURSO 2020-2021

TÍTULO: RECONOCIMIENTO DE LOGOTIPOS DE MARCAS MEDIANTE REDES NEURONALES DE CONVOLUCIÓN (CNN)

AUTOR: RAÚL CASTILLA BRAVO

TITULACIÓN: MÁSTER UNIVERSITARIO EN BIG DATA ANALYTICS

DIRECTORES DEL PROYECTO: LUIS FERNÁNDEZ ORTEGA

FECHA: OCTUBRE de 2021

RESUMEN

El objetivo del trabajo es desarrollar un modelo predictivo basado en Redes Neuronales de Convolución (CNN) para reconocer logos comerciales de 23 marcas distintas. Las imágenes de los logos de entrenamiento están recortadas, así que, se trata de un problema de reconocimiento, no de detección. Durante el proyecto, se muestran arquitecturas básicas de CNNs, modelos de CNNs ensamblados y modelos de *Transfer Learning* como ResNet y GoogLeNet. Además, se incluye un ejemplo de despliegue del modelo final.

ABSTRACT

This work aims to develop a predictive model based on Convolutional Neuronal Networks (CNN) to classify commercial logos from 23 different brands. The logo images are crop so, it's a recognition problem, not a detection one. During the project, basic CNNs, assembly CNNs models, and Transfer Learning models such as ResNet and GoogLeNet are shown. In addition, an example of the final model deployment is included.

Key words: *Logo Recognition, Convolutional Neuronal Network, CNN, Transfer Learning.*

Índice

Índice de Figuras

Índice de Tablas

Capítulo 1. INTRODUCCIÓN

1.1 Necesidad del proyecto

La presencia digital se ha convertido en un requisito indispensable para las empresas modernas. La explotación de los datos generados a través de las redes es fundamental para la toma de decisiones de negocio. Sin embargo, gran parte de la información se presenta de manera desestructurada en forma de imágenes y vídeos.

Por ejemplo, en los últimos años, las redes sociales se han convertido en un medio muy potente para que las empresas puedan obtener *feedback* de sus clientes. A través de las imágenes que comparten los usuarios, se puede saber cuál es el perfil de consumo y sus opiniones. Es lo que se conoce como *User Generated Content* (UGC).

Algunas de estas imágenes contienen **logos comerciales** que pueden ser analizados para extraer conocimiento de valor en futuras estrategias de marketing. De hecho, en ocasiones, este contenido multimedia en sí mismo, se puede utilizar en los medios de promoción de la empresa para reflejar de forma más real la experiencia de consumo de los productos (Orti et al., 2019).

Por otro lado, la identidad digital de las empresas se encuentra constantemente amenazada debido al crecimiento de los ataques de *phishing* (APWG, 2021). En estos ataques, los ciberdelincuentes imitan la apariencia de las páginas webs de empresas reales para engañar a sus clientes y conseguir datos privados como nombres de usuarios, contraseñas o datos bancarios. Esto provoca daños empresariales que van desde pérdidas financieras hasta un deterioro de la confianza.

No obstante, según Geng, Lee (Geng et al., 2014), el 86,2% de las páginas webs de *phishing* contenían el **logo de la marca comercial** a la que estaban suplantando y según Chiew, Leng (Chiew et al., 2015) para ganarse la confianza de los clientes en las páginas de *phishing*, es fundamental incluir el **logo de la marca comercial**.

Por lo tanto, se pueden analizar las páginas webs en forma de imágenes, localizar los **logos comerciales** y detectar intentos de *phishing*, así como averiguar qué empresas están siendo suplantadas.

Al igual que ocurre en los ámbitos que se han detallado, existen muchos otros donde la detección y reconocimiento de **logos comerciales** aporta gran valor, como la detección de logos de vehículos para control de tráfico y los sistemas de alerta de violación de *copyright*.

1.2 Estado del arte

El campo de la detección y reconocimiento de logos se ha visto muy influenciado por los algoritmos de detección de objetos y los modelos de Redes Neuronales de Convolución (CNN) propios del *Deep Learning*.

Dentro de esta área se distinguen dos vertientes: *Logo Detection* (LD), que trata de detectar la posición del logo dentro de una imagen, y *Logo Recognition* (LR) que, una vez detectado el logo, trata de identificar a qué empresa pertenece.

En la actualidad, existen algoritmos que detectan y reconocen los objetos de una imagen, por ejemplo, R-CNN, Fast R-CNN, Faster R-CNN y YOLO (*You Only Look Once*). Exceptuando YOLO, los modelos que se han mencionado son versiones mejoradas de una misma técnica que intentan acelerar lo máximo posible la ejecución del algoritmo, mientras que YOLO lleva a cabo una aproximación distinta para resolver el problema.

Por un lado, los tres primeros modelos se basan en la idea de generar regiones de interés donde puede haber un objeto y, por cada región, aplican un reconocimiento de imágenes. Por otro lado, YOLO divide la imagen original en un *grid* y clasifica cada celda. De esta forma, grupos de celdas colindantes que pertenezcan a la misma clase, pertenecen al mismo objeto.

Gracias a estos algoritmos, muchos investigadores han podido crear modelos de aprendizaje automático capaces de identificar con gran precisión la posición de los logos y la empresa a la que pertenecen.

Para que sirva de referencia, a continuación, se muestran los resultados de algunas investigaciones relacionadas con la detección y reconocimiento de logos. Todas ellas se han basado en el dataset *Flick-Logos32* que está formado por fotos de 32 logos comerciales.

Autor	Año	Precisión	Cobertura	F1
Ahmet et al.	2020	0.935	0.779	0.850
Bianco et al.	2017	0.989	0.906	0.946
Oliveira et al.	2016	0.955	0.908	0.931
Indola et al.	2015	0.896	-	-

Tabla 1. Eficacia de los modelos predictivos antiphishing de los últimos años

Del mismo modo, dentro del ámbito del *User Generated Content*, han surgido muchas propuestas comerciales que aplican estos modelos para analizar imágenes, extraer información de los logos comerciales y ayudar a las empresas con sus estrategias de marketing. Algunas de estas plataformas son: Adsmurai's Visual Commerce Platform [10], Curalate [2] y Olapic [7].

1.3 Inteligencia Artificial

Los algoritmos utilizados para la detección y reconocimiento de logos están recogidos dentro del área de la Inteligencia Artificial. La Inteligencia Artificial es un área de conocimiento que asocia las matemáticas y la informática para conseguir que las máquinas sean capaces de resolver problemas igual o mejor que los humanos. Para conseguir esto, es necesario que las máquinas sean capaces de pensar, ver, oír y actuar como las personas.

El pensamiento humano se puede simular utilizando modelos de *Machine Learning*, la capacidad de ver y oír puede conseguirse con técnicas de procesamiento de imágenes y audio, y la habilidad para actuar se trata llevar a la realidad a través de la robótica.

En un trabajo como este, donde se tiene que reconocer una imagen y tomar una decisión de clasificación, se estarían poniendo a prueba dos de las capacidades de la Inteligencia Artificial: ver y razonar.

Por un lado, el sistema debe ser capaz de recibir una imagen de un logo y reconocer los elementos que hay en él, por ejemplo, formas, bordes, letras y colores, y ser capaz de extraer la información necesaria para entender lo que está viendo. Por otro lado, utilizando la información que ha captado de la imagen, debe ser capaz de decidir cuál es la marca comercial a la que pertenece.

Además, el reto de esta Inteligencia Artificial radica en el hecho de que cada imagen que recibe puede tener el logo comercial dispuesto de formas distintas. Por ejemplo, una misma empresa puede haber tenido varias versiones de su logo corporativo y el sistema debe seguir siendo capaz de entenderlo y clasificarlo.

De igual modo, un mismo diseño de logo puede aparecer en distintos escenarios. Por ejemplo, puede ser tan sencillo como una imagen plana sobre un fondo blanco o puede aparecer impreso en la superficie de un producto. En cualquiera de estas situaciones, un humano sería capaz de identificar a qué empresa pertenece así que, la Inteligencia Artificial también debe ser capaz de hacerlo.

Asimismo, si un logo tiene suciedad, partes ocultas o partes que no se pueden reconocer por la calidad de la imagen, si un humano puede deducir la empresa viendo solo un fragmento, la Inteligencia Artificial también debería poder.

Como se puede observar, se trata de un reto muy complejo debido a la gran variedad de circunstancias que pueden darse y, como consecuencia, este tipo de problemas no se aborda con una solución puramente algorítmica. Si se intentara crear un algoritmo que reconociera los logos, probablemente requeriría mucho tiempo, habría casuísticas que no se contemplarían y se cometerían muchos errores.

Es en este punto donde el Machine Learning aparece para dar una solución. En lugar de indicar en detalle qué es lo que debe hacer la máquina para reconocer los logos, la máquina puede aprenderlo por sí misma.

1.4 Machine Learning

Tal y como indicó Tom Mitchell, se dice que un ordenador aprende de una experiencia E en relación con una tarea T y una medida de efectividad P, si su eficacia en T, medido por P, mejora con la experiencia E. En otras palabras, un ordenador aprende si en base a una experiencia puede aprender a resolver una tarea.

En el caso del reconocimiento de logos, la experiencia son las imágenes disponibles de los logos comerciales, la tarea es clasificarlos por empresas y la medida de efectividad será el error cometido en la clasificación.

Dentro del Machine Learning existen muchas técnicas, pero se puede clasificar en tres grandes grupos: aprendizaje supervisado, no supervisado y por refuerzo.

Se dice que el aprendizaje es supervisado cuando se dispone de un conjunto de datos y se sabe a priori a qué clase pertenecen, es decir, se tiene constancia de la respuesta que debe dar el modelo. Por el contrario, en el aprendizaje no supervisado, no se conoce cuál es la respuesta correcta, sino que hay que buscar patrones que ayuden a entender los datos.

Por otra parte, el aprendizaje por refuerzo hace un planteamiento distinto. En lugar de disponer de un conjunto de datos, el modelo captura experiencias durante el aprendizaje y mejora mediante prueba y error. Por ejemplo, si se quiere aprender a tirar a canasta, el modelo empezará probando diferentes formas de lanzar la pelota e irá modificándolas para acercarse al aro y llegar a encestar.

Para el problema de clasificación de logos, se dispone de imágenes y se sabe a qué empresa pertenecen. Por lo tanto, se trata de un problema de aprendizaje supervisado.

Dentro del Machine Learning existen otras jerarquías que permiten clasificar las técnicas. Una de ellas tiene en cuenta lo que se conoce como interpretabilidad. Se dice que un modelo es interpretable si un humano puede entender fácilmente el porqué de las decisiones que toma, y aquí se distinguen dos corrientes: la simbólica y la subsimbólica.

Por un lado, la corriente simbólica pretende simular la forma de razonar que tienen las personas y son fácilmente interpretables, por ejemplo, un sistema de reglas. Por otro lado, la corriente subsimbólica pretende simular la estructura del cerebro humano a nivel de conexiones y neuronas. En este caso, los modelos que se generan son muy difíciles de interpretar y acaban tratándose como cajas negras.

Los mayores representantes de la corriente subsimbólica son las redes neuronales, unos modelos que, a pesar de la falta de interpretabilidad, han demostrado ser capaces de resolver una gran variedad de problemas de suma complejidad. Por ello, los modelos basados en redes neuronales han ido creciendo hasta dar lugar a una nueva área de conocimiento dentro del Machine Learning llamada Deep Learning.

1.5 Deep Learning

El Deep Learning lleva a las Redes Neuronales a su máximo potencial con modelos capaces de procesar imágenes, texto, audio y prácticamente cualquier tipo de dato. Son capaces de resolver tareas de extremada complejidad con una eficacia y eficiencia muchas veces superior a la de los humanos.

Dentro de esta rama es donde se sitúan las Redes Neuronales de Convolución (CNN), la técnica de Deep Learning que se utilizará a lo largo del trabajo para resolver el problema de clasificación de logos.

Otras arquitecturas de redes neuronales se pueden observar en el cuadro resumen de la *figura 2* creado por Fjord van Veen del instituto de Asimov [12]. En él, se recogen las arquitecturas básicas como el perceptrón y la red *Feed Forward,* que fueron el inicio de las redes neuronales, y arquitecturas más complejas como las Redes Neuronales Recurrentes y las Redes Neuronales de Convolución.

Asimismo, a modo de resumen, se muestra en la *figura 1* un diagrama con las áreas de conocimiento que se han mencionado hasta este momento representando la pertenencia de cada una de ellas.

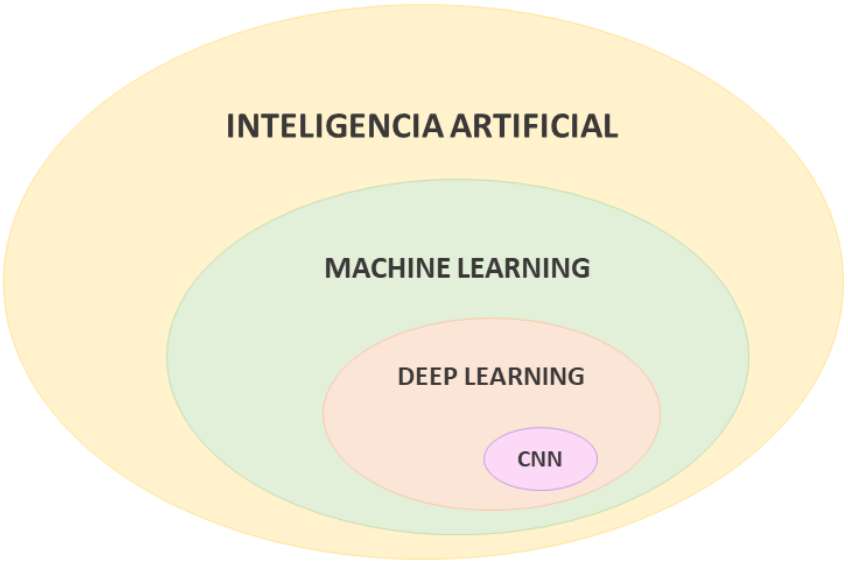

Figura 1 Áreas de conocimiento en las que se encapsula el Deep Learning

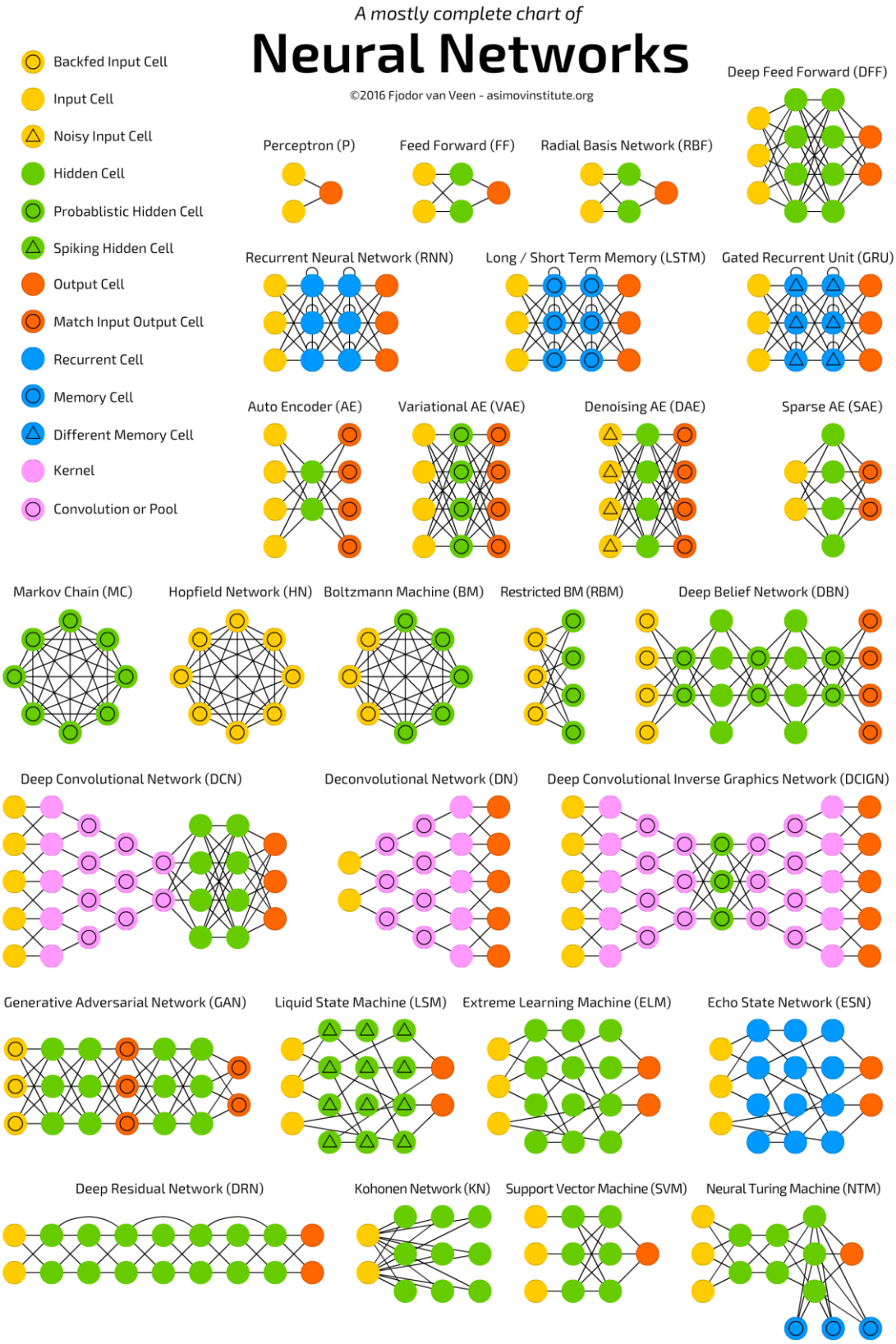

Figura 2. Recopilación de arquitecturas de Redes Neuronales. Tomada de van Veen et al., 2017

1.6 El perceptrón

De todas las arquitecturas de red que se han mostrado en la *figura 2*, el perceptrón es la más sencilla de todas. De hecho, no conforma una red en sí misma porque es una única neurona que no se conecta con otras, pero solo con esa neurona ya es posible modelar y resolver determinados problemas.

Los elementos que intervienen en un perceptrón son: los datos de entrada, los pesos y la función de activación.

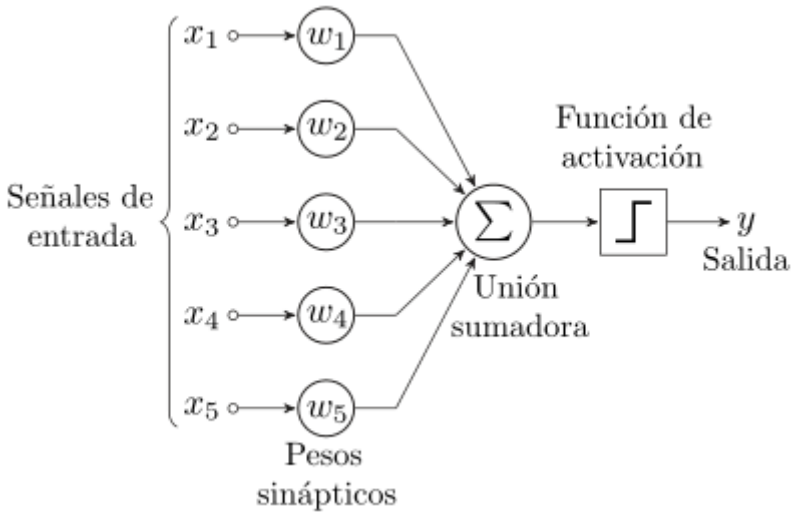

Figura 3. Diagrama de un perceptrón. Tomada de Wikipedia

Matemáticamente, un perceptrón es una función f que recibe n variables, las pondera según unos pesos w y devuelve una salida.

$$f(X) = w_1x_1 + w_2x_2 + \cdots + w_nx_n + b$$

Representado de esta forma, un perceptrón define la ecuación de una recta n-dimensional, es decir, se puede utilizar para resolver problemas de regresión donde los datos presentan una dispersión lineal o para problemas de clasificación donde los conjuntos son linealmente separables.

Dado que los datos de entrada son los datos del problema y no se pueden modificar, el perceptrón adapta los pesos de la red para que la recta que representa se ajuste a la dispersión de los datos (regresión) o consiga separarlos (clasificación).

Inicialmente, el perceptrón tiene unos valores de peso aleatorios y, para calcular la modificación que hay que hacer a los pesos, se debe definir una función de error, que será la que se trate de minimizar.

Por ejemplo, en un perceptrón con dos entradas, la función que representa sería:

$$f(X) = w_1 x_1 + w_2 x_2 + b$$

Y los valores que toma el **error** en función de las variables w_1 y w_2 se podría representar en un espacio tridimensional como se muestra en la *figura 4*. El objetivo será aplicar pequeñas modificaciones a los valores w_1 y w_2 en la dirección de mayor decrecimiento del error hasta llegar a un valor mínimo.

La dirección de mayor crecimiento/decrecimiento es lo que se conoce como gradiente y el factor que pondera la variación de los pesos es el factor de aprendizaje (*learning rate*). Cuanto mayor sea el factor de aprendizaje, mayor será la variación que sufren los pesos en la dirección del gradiente.

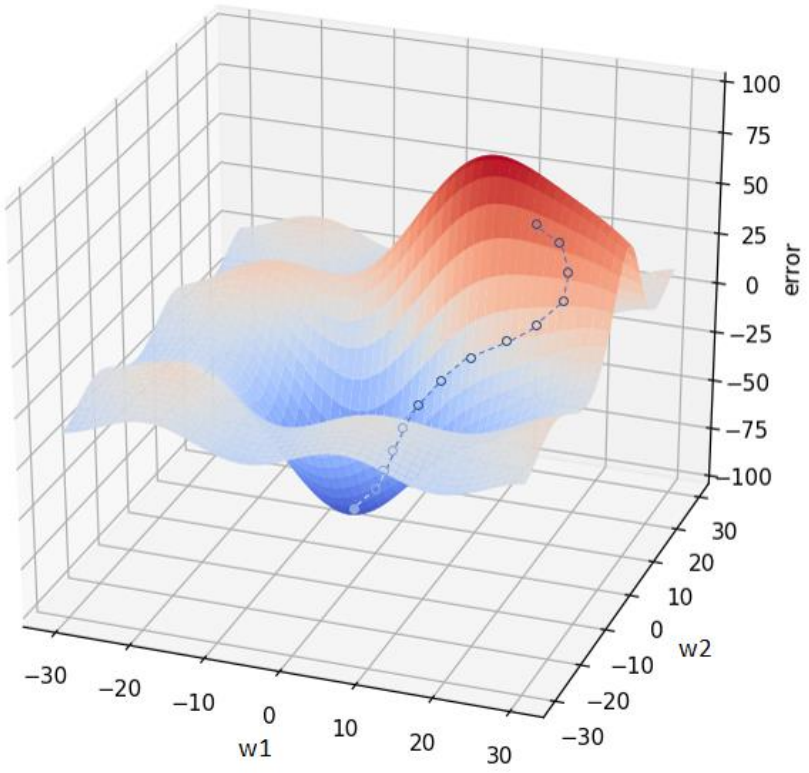

Figura 4. Representación de los valores de una función de error. Tomada de InteractiveChaos

Hasta este punto, la definición de perceptrón no es más que la de una regresión lineal y solo puede resolver problemas a los que se les pueda hacer una aproximación lineal. Sin embargo, el potencial del perceptrón radica en la **no linealidad**, lo cual le permite resolver problemas más variados.

El carácter no lineal de los perceptrones está en la función de activación, que recibe el resultado de ponderar los valores de entrada por los pesos de la red y los introduce en una función con un comportamiento no lineal.

$$y = g(f(x))$$

Siendo y la salida de la red, $g(x)$ la función de activación y $f(x)$ la unión sumadora de los datos de entrada x por los pesos w.

A modo de ejemplo, en la *figura 5* se muestran las funciones de activación más comunes. ReLu devuelve 0 si la entrada es menor a 0, Softmax aplica una función exponencial, Sigmoid devuelve valores entre 0 y 1, y Tanh devuelve valores entre -1 y 1.

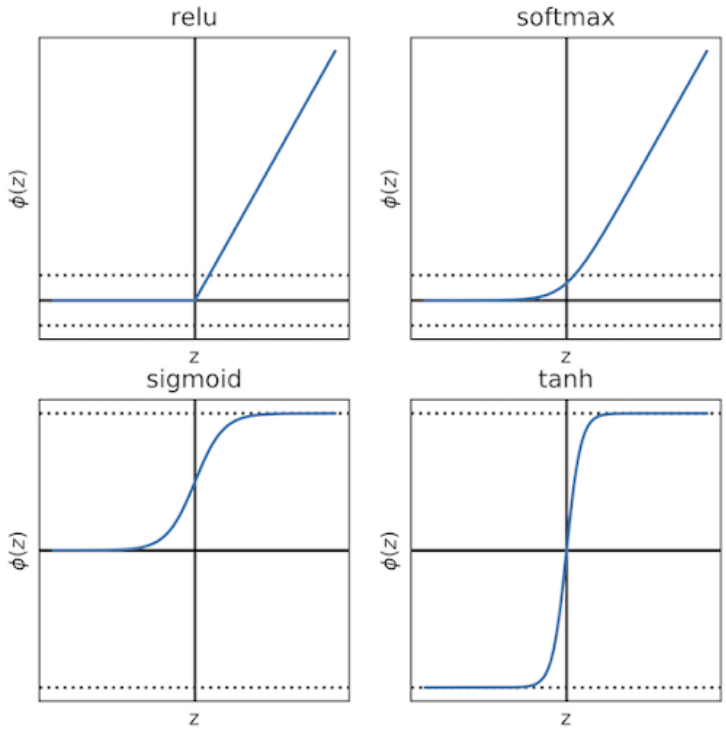

Figura 5. Funciones de activación no lineales. Tomada de documentación de Keras

1.7 Redes Neuronales

A pesar de utilizar una función de activación no lineal, el perceptrón por sí solo tiene una capacidad de expresividad limitada. Esto quiere que, si el problema es muy complejo y se requieren modelar muchos aspectos, es posible que el perceptrón no sea capaz de aproximar una solución óptima.

En su lugar, se utilizan varios perceptrones que se combinan entre ellos conectando las salidas de unos con las entradas de otros para crear una red neuronal. De esta forma, la expresividad y la potencia del conjunto permite modelar comportamientos más complejos.

En su forma más genérica, las redes neuronales tienen las neuronas dispuestas en capas que conectan la salida de cada una de ellas con la entrada de cada neurona de la capa siguiente. La capa que recibe los datos del problema es la capa de entrada, la que devuelve el resultado es la capa de salida y todas las intermedias son las capas ocultas.

Cada neurona de la red se comporta como un perceptrón que tiene unos pesos y una función de activación no lineal. Aunque la arquitectura de la red sea muy compleja, si no se usaran funciones de activación no lineales, la red solo podría modelar una recta.

Por otra parte, al igual que ocurre con los perceptrones, la red empieza con valores de peso aleatorios y es necesario modificarlos para ajustarse a los datos del problema. El algoritmo que se utiliza para ello es más complejo que el del perceptrón, pero se sigue apoyando en la idea del gradiente y el factor de aprendizaje para modificar los pesos.

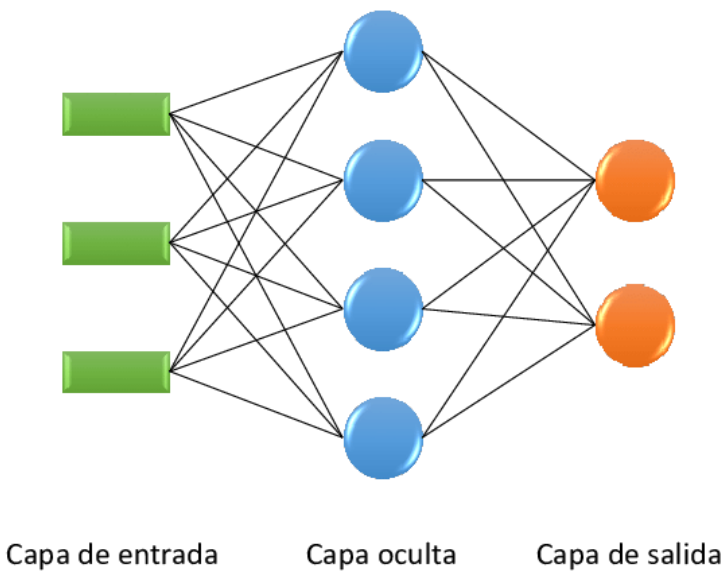

Figura 6. Estructura de una red neuronal. Tomada de Relaciones Neuronales Para Determinar la Atenuación del Valor de la Aceleración Máxima en Superficie de Sitios en Roca Para Zonas de Subducción, por Lino Manjarrez, 2014

1.8 Redes Neuronales *Feed Forward* para procesamiento de imágenes

Las redes neuronales de tipo *Feed Forward* como la que se muestra en la *figura 6*, son de gran utilidad para problemas complejos y son capaces de trabajar con muchas variables. Sin embargo, para problemas de clasificación de imágenes, las aproximaciones que consiguen estas redes no son muy buenas y la razón se explica a continuación.

Una imagen está formada por una o varias matrices donde cada valor recoge la luminosidad y color de cada píxel. Para poder introducir una imagen dentro de una red neuronal *Feed Forward*, es necesario que los valores estén expresados como un único vector. Esto implica que, previamente, es necesario aplanar la matriz a una única dimensión. Por ejemplo, una matriz de 32x32 daría lugar a un vector de 256 valores.

Esto puede funcionar cuando las dimensiones de la imagen son reducidas. Sin embargo, en la realidad no suele ocurrir así. Generalmente, las imágenes tienen un tamaño muy superior y cada vez tienen más resolución. Por ejemplo, una imagen *Full HD* de 1920x1080 daría lugar a un vector de 2 millones de valores y la red tendría, como mínimo, 2 millones de valores de peso a optimizar, lo cual supone un gran problema de cómputo.

Además, convertir las matrices a vectores provoca la pérdida de una característica fundamental que tienen las imágenes, la localidad. A diferencia de otros tipos de datos, la localización espacial de los valores en la matriz aporta información, porque los píxeles que están cercanos unos a otros pueden formar parte de un mismo elemento de la imagen.

Por lo tanto, es necesario hacer una aproximación al problema utilizando una red neuronal capaz de reducir las dimensiones de la imagen para trabajar con menos parámetros y que pueda extraer información del valor del píxel, de su posición en la imagen y de los píxeles vecinos a él. Es aquí donde entran las Redes Neuronales de Convolución (CNN).

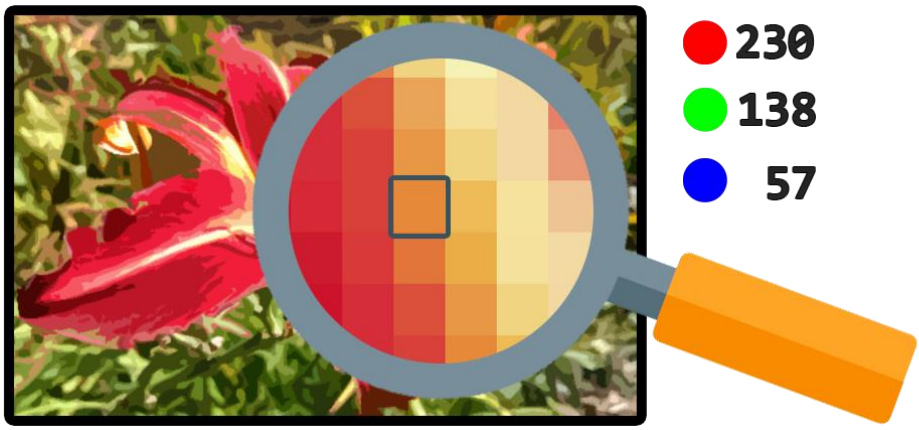

Figura 7. Representación de imágenes como matrices de valores. Tomada de pngfind

1.9 Redes Neuronales de Convolución

Las CNN son capaces de abordar los problemas de localidad y dimensionalidad de los píxeles a través de dos operaciones llamadas convolución y *pooling*. Estas operaciones se aplican previamente a utilizar una red neuronal *Feed Forward* para que no trabaje directamente sobre la imagen original, sino sobre sus características principales.

Teniendo esto en cuenta, una CNN se puede dividir en dos partes: la parte convolucional y la red neuronal. En la *figura 8* se puede apreciar un diseño básico de una CNN que tiene en su parte convolucional por 2 pares de capas de convolución y de *pooling,* y en la parte *Feed Forward* tiene 2 capas ocultas y 1 de salida.

Además, se puede observar cómo evoluciona la dimensionalidad de la imagen a medida que avanza por las capas de la red. Si bien, esto se entenderá mejor a continuación cuando se detalle el funcionamiento de las operaciones de convolución y *pooling*.

Por otra parte, también se detalla el uso de una capa *softmax* que tiene su explicación en la función de error que se utiliza para optimizar los pesos de la red. Este aspecto también se explicará más adelante.

A continuación, se explicarán cada una de las capas de la CNN que, en su versión más básica, cuenta con los siguientes tipos de capas:

- Convolucionales y *pooling*
- *FeedForward* o *Linear*

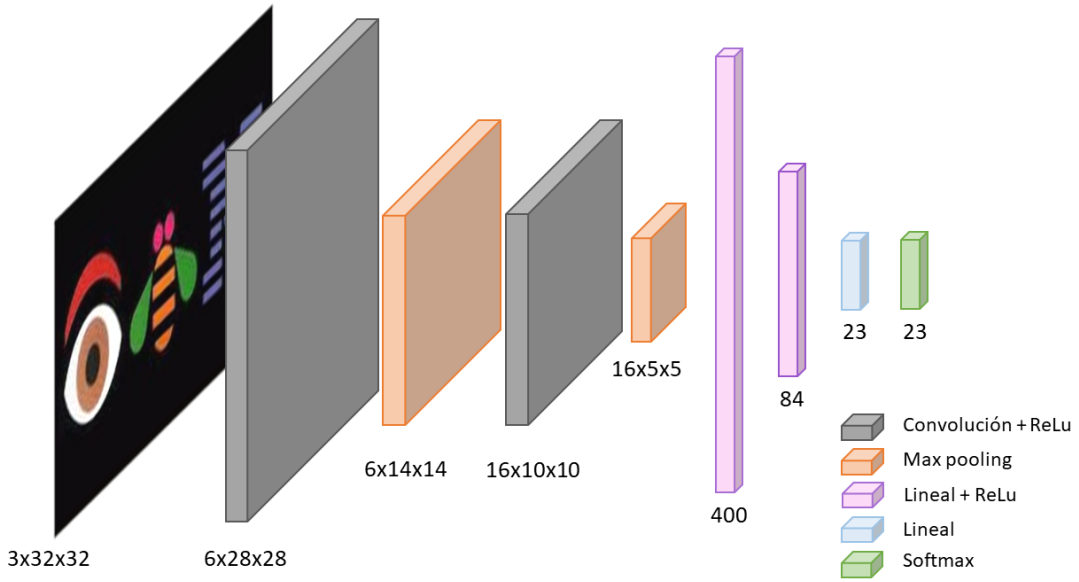

Figura 8. Arquitectura básica de una Red Neuronal Convolucional (CNN)

1.9.1 Convolución

La convolución se utiliza para extraer información de localidad. Para ello, usa una matriz llamada *kernel* que recorre cada píxel de la imagen original y genera un único valor, utilizando tanto el valor del píxel en sí, como el valor de los píxeles vecinos. De esta forma, se genera una matriz nueva llamada matriz convolucionada, que contiene la información del píxel y sus vecinos.

A modo de ejemplo, en la *figura 9* se puede apreciar cómo se aplica la operación de convolución sobre una imagen. En este caso, cada píxel que se encuentre en una de las diagonales del *kernel* suma su valor al resultado final. Para el píxel situado en la esquina superior izquierda, de los 5 píxeles que hay en las diagonales, 4 de ellos tienen el valor 1, así que, el resultado final es 4.

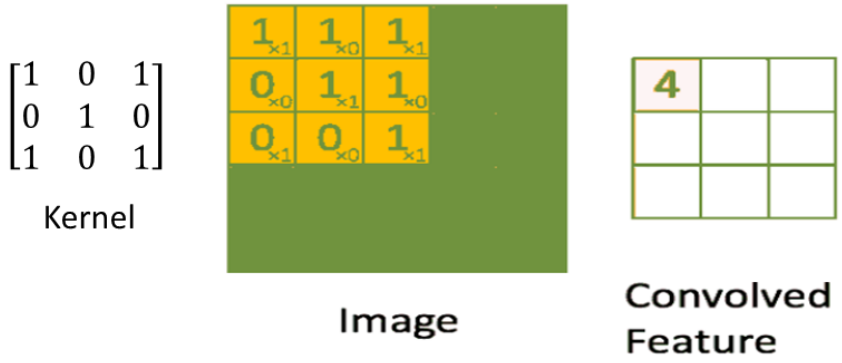

Figura 9. Inicio de operación de convolución. Tomada de GitHub (https://github.com/vdumoulin/conv_arithmetic)

Figura 10. Progreso de operación de convolución. Tomada de GitHub (https://github.com/vdumoulin/conv_arithmetic)

Respecto a las operaciones de convolución, cabe destacar que, como ocurre en la *figura 10*, la matriz convolucionada tiene menor dimensión que la original. Aunque esto puede ser útil, hay situaciones donde se prefiere mantener el mismo tamaño.

Para regular este comportamiento, se utiliza una técnica llamada *padding*, que añade píxeles a los bordes de la imagen original. De esta forma, después de aplicar la convolución, el resultado mantiene las mismas filas y columnas.

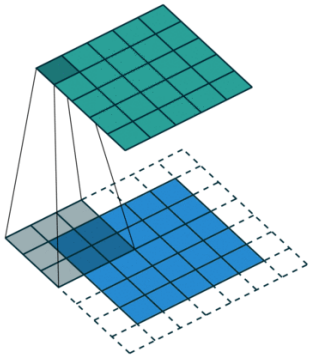

Figura 11. Convolución aplicando padding. Tomada de GitHub (https://github.com/vdumoulin/conv_arithmetic)

Por otra parte, cuando se aplica una convolución sobre una imagen que está formada por varias matrices, el *kernel* debe recoger información de todas ellas y resumirlas en una única matriz. Esto implica que el *kernel* tendrá tantas matrices como la imagen original y el resultado de la convolución será la suma del resultado de cada matriz del *kernel*, como muestra la *figura 12*.

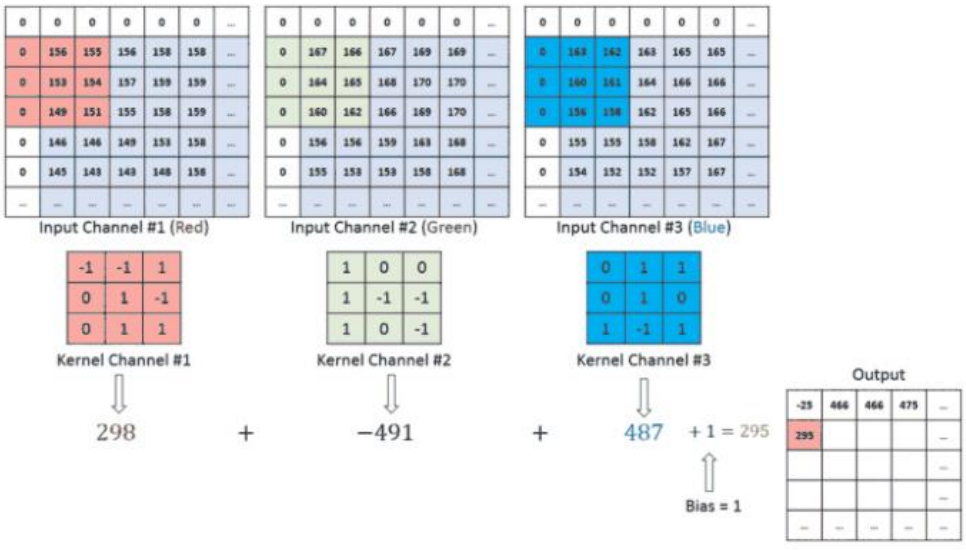

Figura 12. Convolución sobre una imagen con tres canales de color. Tomada de GitHub (https://github.com/vdumoulin/conv_arithmetic)

1.9.2 *Pooling*

Por otra parte, la operación de *pooling* permite reducir la dimensionalidad y extraer las características más relevantes. Además, se reducen los valores anómalos porque acaban combinándose con el resto de los píxeles de su alrededor.

Para aplicar *pooling*, se divide la imagen en un *grid* y en cada celda se aplica una operación que devuelva un único valor. Las operaciones de *pooling* más habituales consisten en extraer el valor máximo (*max pooling*) o el promedio (*average pooling*).

Esto implica que, si se divide la imagen en un *grid* formado por celdas de 2x2 y de cada celda solo se extrae un valor, la dimensión de la matriz de salida se divide a la mitad en filas y columnas. Por ejemplo, en la *figura 13* se puede apreciar esta reducción de dimensiones.

Generalmente, la operación de *pooling* que más se utiliza es el *max pooling* porque actúa además como un supresor de ruido eliminando las activaciones residuales. Esto significa que, si una neurona ha tenido una activación muy baja, quiere decir que la información que aporta es poco relevante así que, el *max pooling* ignora esa información y permite a la red trabajar únicamente con las características más relevantes.

Por el contrario, si se utilizara el *average pooling*, el ruido podría acabar mezclándose con los datos importantes y la precisión de la red sería menor.

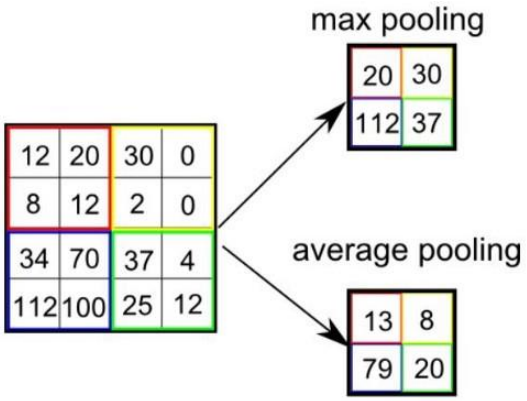

Figura 13. Operaciones de pooling. Tomada de A Comprehensive Guide to Convolutional Neural Networks — the ELI5 way, S. Saha, 2017, Towards Data Science

1.9.3 Optimización de la parte convolucional de la CNN

Una vez entendido que la convolución permite extraer la información de localidad y que el *pooling* ayuda a reducir la dimensionalidad para trabajar solo con las características más importantes, llega el momento de aclarar qué es lo que se optimiza en una red CNN.

Las operaciones de convolución son anteriores a la llegada de las arquitecturas de redes convolucionales y se empezaron a usar dentro del campo de la visión por computador. El objetivo de la convolución es extraer determinadas características de la imagen y, para ello, los ingenieros y matemáticos diseñaban *kernels* concretos pensados para tareas específicas.

Por ejemplo, aplicando los *kernels* k_x y k_y que se indican a continuación, se pueden detectar los bordes horizontales y verticales de las imágenes.

$$k_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \qquad k_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

Estos *kernels* se basan en la misma idea del gradiente. Si se convolucionan, se generan valores altos cuando hay una variación alta en píxeles que son vecinos y esto ocurre en una imagen cuando hay un borde. A modo de ejemplo, en la *figura 14* se puede observar lo que ocurre al convolucionar los *kernels* detectores de bordes sobre una imagen de una persona.

De esta forma, a través de una convolución se pueden extraer características concretas de la imagen. Sin embargo, es necesario que una persona diseñe los *kernels* adecuados, lo cual no siempre es fácil y más aún si se trabaja con *kernels* de varias dimensiones.

Figura 14. Ejemplo de convolución de kernels detectores de bordes. Tomada de Canny Edge Detection Step by Step in Python — Computer Vision, S. Sahir, 2019, Towards Data Science

En lugar de un diseño manual de los *kernels*, las Redes Neuronales de Convolución tratan de aprender cuáles son los *kernels* que permiten extraer el mayor conocimiento posible para después enviar esas características a la red neuronal *Feed Forward*. Así, la red *Feed Forward* no trabaja sobre la imagen original sino sobre sus características.

Por lo tanto, si se divide la CNN en parte convolucional y parte *Feed Forward,* lo que se optimiza en la parte convolucional son los *kernels* y lo que se optimiza en la red *Feed Forward* son los pesos de las neuronas.

1.9.4 Flujo de operaciones de una CNN

Una vez visto el detalle de todas las operaciones que se aplican en una CNN, se puede hacer un recorrido completo de los datos desde que entran hasta que se obtiene una salida. Para ello, se utilizará como referencia la *figura 15* donde se detallan las capas, los tamaños de los *kernels* y las dimensiones de las matrices en cada instante.

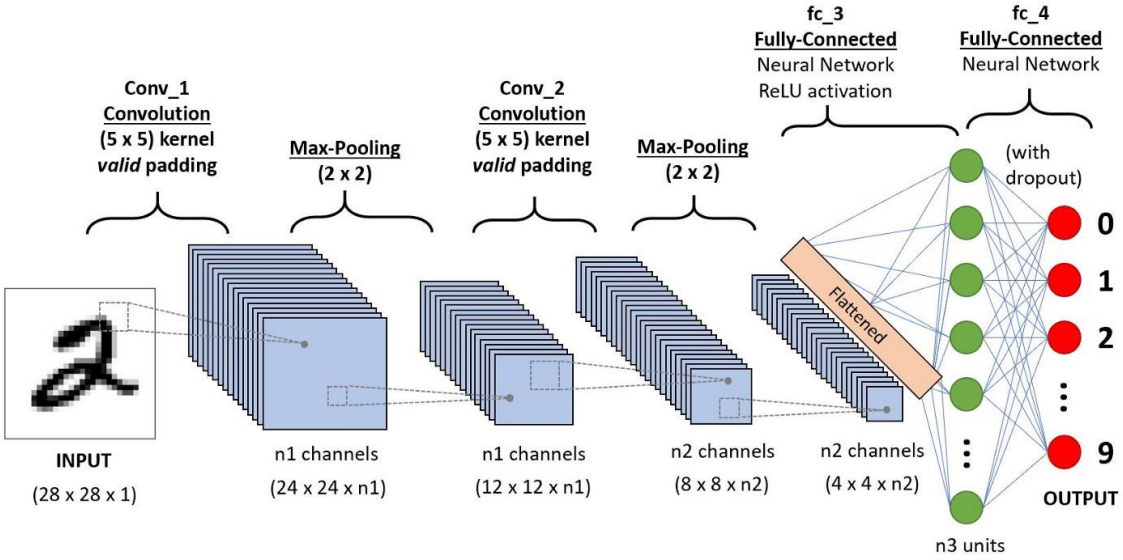

Figura 15. Diagrama de operaciones de una CNN. Tomada de A Comprehensive Guide to Convolutional Neural Networks — the ELI5 way, S. Saha, 2017, Towards Data Science

El proceso comienza con la imagen de entrada que tiene un solo canal de color y sus dimensiones son 28 x 28 x 1. A continuación, entra en la primera capa convolucional (*Conv_1*) que aplica un kernel de dimensiones 5 x 5 sin utilizar *padding,* así que, se reducen las dimensiones originales y el resultado son matrices de 24 x 24.

En las capas convolucionales se pueden aplicar tantos *kernels* como se desee y cada uno de ellos extraerá características distintas de la imagen. Si se define n_1 como el número de *kernels* que se aplican en la capa *Conv_1*, el resultado que se obtiene después de procesar la imagen es de dimensiones 24 x 24 x n_1. Además, como la imagen de entrada tiene un solo canal, los *kernels* de la capa *Conv_1* son de 5 x 5 x 1.

Seguidamente, se aplica una capa de *max pooling* de 2 x 2 que reduce el número de filas y columnas a la mitad, dando lugar a un resultado de dimensiones 12 x 12 x n_1.

Después, los datos entran en una segunda capa de convolución (*Conv_2*). En esta capa, los *kernels* vuelven a tener 5 filas y 5 columnas, pero como los datos de entrada tienen n_1 canales, los kernels también tienen n_1 canales, es decir, son *kernels* de 5 x 5 x n_1.

Además, como se ha mencionado antes, en cada capa de convolución se pueden aplicar tantos *kernels* como se desee. Así que, si se define n_2 como el número de *kernels* de la capa *Conv_2*, como tampoco se usa *padding,* el resultado de *Conv_2* tiene dimensiones 8 x 8 x n_2.

Finalmente, entra en una capa de *max pooling* de 2 x 2 y el resultado adquiere la dimensión 4 x 4 x n_2. Ahora, este resultado se aplana en un único vector que alimenta a la red *Feed Forward* para llevar a cabo la tarea de clasificación.

1.9.5 Introducción al *Transfer Learning*

Atendiendo a la forma operar que tiene la CNN, se puede observar que, realmente, la tarea de clasificación está en la parte *Feed Forward* y que la parte convolucional actúa como un extractor de características. Este es el principio sobre el cual se fundamenta el *Transfer Learning*.

El *Transfer Learning* consiste en apoyarse en redes CNN pre entrenadas con millones de imágenes para que actúen como extractores de características, ya que sus *kernels* son válidos para extraer información de cualquier imagen. Es como utilizar los *kernels* detectores de bordes para distintos problemas.

Lo único que se debe hacer para que la CNN se adapte a un problema concreto es entrenar la parte *Feed Forward*, lo cual es mucho menos costoso a nivel computacional que entrenar una red CNN completa con millones de imágenes.

Para este paso existen dos opciones. Por un lado, se pueden "congelar" las capas convolucionales para que no se modifiquen en absoluto y, por otro lado, se pueden optimizar levemente junto con la red *Feed Forward* para que se adapten mejor al problema. A esto último también se lo conoce como *Finetunning*.

Como se puede intuir, la ventaja de congelar las capas es que el tiempo de entrenamiento es mucho menor, pero el *Finetunning* suele tener mejores resultados.

1.10 Funciones de pérdida

El entrenamiento de un modelo de aprendizaje como las CNN se consigue mediante ajustes en los pesos de la red durante el entrenamiento. Para ello, es necesario definir una función que permita calcular el error que se comete entre el resultado obtenido y el esperado, y sobre esa función se calculará el gradiente que permitirá ajustar los pesos.

A las funciones de error también se las conoce como funciones de pérdida y varían según el tipo de problema que se esté resolviendo. Por ejemplo, supóngase que el problema de clasificación de logos comerciales se reduce a distinguir únicamente entre dos marcas, es decir, que fuera un problema de clasificación binaria.

En este caso, se podría definir una CNN que tuviera en la capa de salida de la red *Feed Forward* una única neurona cuyo resultado estuviera normalizado entre 0 y 1. Si el resultado es menor a 0.5 se clasificaría en la primera clase y, si es mayor o igual, en la segunda.

Para entrenar esta red, se podría utilizar una función de pérdida conocida como *Binary Crossentropy* que se define a continuación:

$$Loss = -(y \cdot \log \hat{y} + (1 - y) \cdot \log(1 - \hat{y}))$$

Donde y es el valor esperado e $\hat{y}$ es el valor que obtiene la red. Para probar esta función de pérdida, se pueden definir 4 casos extremos:

- $y = 0, \quad \hat{y} = 0 \rightarrow Predicción\ perfecta \rightarrow Loss = 0 \cdot \log 0 + \log 1 = 0$
- $y = 1, \quad \hat{y} = 1 \rightarrow Predicción\ perfecta \rightarrow Loss = \log 1 + 0 \cdot \log 0 = 0$
- $y = 1, \quad \hat{y} = 0 \rightarrow Máximo\ error \quad \rightarrow Loss = -(\log 0 + 0 \cdot \log 1) = \infty$
- $y = 0, \quad \hat{y} = 1 \rightarrow Máximo\ error \quad \rightarrow Loss = -(0 \cdot \log 1 + \log 0) = \infty$

Cuando la coincidencia es perfecta, uno de los logaritmos acaba siendo $\log 1$ que es 0, y el otro acaba con $\log 0$ que es $-\infty$, pero queda multiplicado por 0 y se convierte en 0.

Cuando se produce el máximo error, uno de los logaritmos vuelve a ser $\log 1$, y el otro $\log 0$, pero en este caso, el $\log 0$ acaba multiplicado por 1, así que el resultado es $-\infty$, que se convierte a ∞ por el $-$ que engloba a la expresión. De esta forma, el error que devuelve la *Binary Crossentropy* está comprendido entre $[0, \infty)$.

Para el caso de problemas multiclase, si la red tiene una única neurona de salida, puede llegar a ser difícil distinguir las clases del problema. Por ejemplo, si el problema tiene 10 clases, la salida podría estar comprendida entre 0 y 10, y se podría aplicar la siguiente regla:

- Si el resultado es menor que 1 → Clase 1
- Si el resultado es mayor que 1 y menor que 2 → Clase 2
- …

Sin embargo, esto supone que hay un orden implícito en las clases cuando, en realidad, puede no ser así. En su lugar, la salida de la red se plantea de forma distinta, habrá una neurona de salida por cada clase del problema y cada valor estará normalizado entre 0 y 1. Por ejemplo, en un problema de 10 clases, la red devolverá un vector de 10 posiciones.

Para continuar con este planteamiento, es necesario que los valores esperados también estén expresados como un vector. Para ello, las clases se presentan en formato *One-Hot*, por ejemplo, la clase 1 será un vector de 10 posiciones que tendrá todos los valores a 0 y un 1 en la primera posición.

- Clase 1 → $[1, 0, 0, 0 \dots 0]$
- Clase 2 → $[0, 1, 0, 0 \dots 0]$
- Clase 3 → $[0, 0, 1, 0 \dots 0]$

De esta forma, para medir el error, se puede utilizar una expresión similar a la *Binary Crossentropy* llamada *Categorical Crossentropy* que se define a continuación:

$$Loss = -\sum_{i=1}^{size} y_i \cdot \log \hat{y}_i$$

Siendo y_i el valor esperado en la posición i e $\hat{y}_i$ el valor predicho por la neurona i. Por tanto, esta función de pérdida calcula el error de cada posición del vector y así, no hay un orden implícito en las clases del problema.

Al igual que con la *Binary Crossentropy*, la *Categorical Crossentropy* tiene 4 casos extremos:

- $y = 0, \quad \hat{y} = 0 \rightarrow Predicción\ perfecta \rightarrow Loss = 0 \cdot \log 0 = 0$
- $y = 1, \quad \hat{y} = 1 \rightarrow Predicción\ perfecta \rightarrow Loss = 1 \cdot \log 1 = 0$
- $y = 1, \quad \hat{y} = 0 \rightarrow Máximo\ error \quad \rightarrow Loss = -1 \cdot \log 0 = \infty$
- $y = 0, \quad \hat{y} = 1 \rightarrow Sin\ error \quad \rightarrow Loss = 0 \cdot \log 1 = 0$

Como se puede observar, el error solo se contabiliza en la posición del vector donde se espera que haya un 1 y el caso $y = 0, \ \hat{y} = 1$ no suma.

Después de haber definido la *Categorical Crossentropy* como función de pérdida, es importante asegurar que la salida del modelo está normalizada entre 0 y 1 para que se pueda usar correctamente.

Para ello, se utiliza una operación llamada *Softmax* que se define a continuación:

$$\sigma(x_j) = \frac{e^{x_j}}{\sum_k^K e^{x_k}}$$

Esta operación asegura que los valores están normalizados entre 0 y 1 y que la suma de todos ellos es 1 para que se pueda interpretar como un vector de probabilidades. De esta forma, para determinar a qué clase pertenece una muestra de entrada, basta con seleccionar la más probable según el vector que se obtiene del *Softmax*.

Ahora bien, , estas mismas propiedades se pueden obtener con una operación más sencilla como la que se muestra a continuación:

$$f(x_j) = \frac{x_j}{\sum_k^K x_k}$$

Básicamente es la misma idea, pero sin utilizar exponenciales. A priori, la segunda opción es más sencilla de computar y tiene el mismo efecto. Sin embargo, el *Softmax* tiene otras propiedades que lo hacen más interesante.

Supóngase que se está resolviendo un problema de 4 clases, la salida esperada es [0, 1, 0, 0] y la salida obtenida es [2, 4, 2, 1]. Para normalizar el resultado, hay tres opciones:

- Extraer el argumento máximo $\rightarrow [0, 1, 0, 0]$
- Normalizar con $f(x) \rightarrow [0.2222, 0.4444, 0.2222, 0.1111]$
- Normalizar con $\sigma(x) \rightarrow [0.1025, 0.7573, 0.1025, 0.0377]$

Lo ideal sería extraer el argumento máximo porque daría error 0 cuando la predicción es perfecta y devolvería error máximo cuando la predicción falla. Sin embargo, no se puede hacer así porque la red vería que ha cometido un error enorme siempre que se equivoque, a pesar de que, a lo mejor, la diferencia que ha provocado el error es mínima.

Por tanto, hay que elegir una alternativa similar al argumento máximo, pero que permita diferencia los valores que se obtienen en otras posiciones. Si se observa el resultado entre $f(x)$ y $\sigma(x)$, el *Softmax* es la que más se aproxima al argumento máximo. Además, hay otra propiedad que hace más atractivo aún al *Softmax* y es que, para el cálculo del gradiente, es necesario hacer cálculos de derivadas y la derivada de e^x no requiere operaciones porque es e^x.

1.11 Optimizadores

1.11.1 Descenso por Gradiente vs Descenso por Gradiente Estocástico

Una vez entendido el diseño de red y de haber definido la función de pérdida, lo último que falta para poder entrenar un modelo es un optimizador. El optimizador será el encargado de calcular cuál es la variación que se debe aplicar a los pesos de la red para que se minimice el error.

Para ello, la opción más sencilla es el Descenso por Gradiente que consiste en calcular el gradiente de todos los pesos de la red. Esto permite reducir el error en la dirección de máximo decrecimiento. Sin embargo, tiene un alto coste computacional porque en cada iteración del entrenamiento hay que calcular todos los gradientes.

Como alternativa, existe la posibilidad de calcular solo algunos de los gradientes en cada iteración. Esto provoca que no se reduzca el error en la dirección de máximo decrecimiento, pero se sigue reduciendo y en algún momento llegará al valor mínimo. Esto es lo que se conoce como Descenso por Gradiente Estocástico (*Stocastic Gradient Descent, SGD)*.

La comparación entre ambas técnicas se puede ver en la *figura 16*. Lo que ocurre es que, en el Descenso por Gradiente, la trayectoria que describe el error es directa hacia el mínimo, mientras que, en el SGD, la trayectoria es más errática, pero ambos acaban en el mismo mínimo.

Desde el punto de vista práctico, el SGD necesita más iteraciones de entrenamiento, pero como el tiempo de cómputo de cada iteración es menor, acaba siendo más rápido que el Descenso por Gradiente.

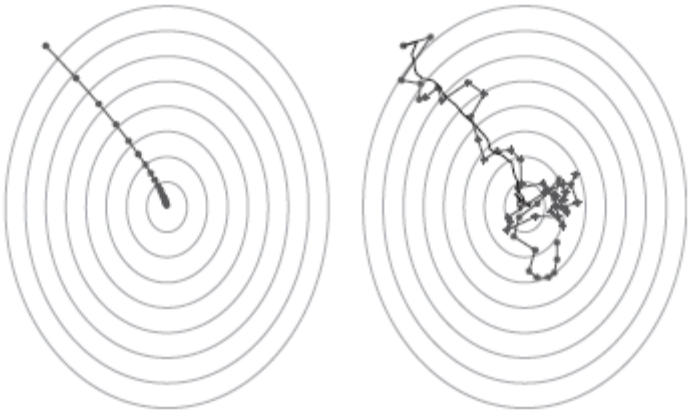

Figura 16. Descenso por Gradiente (izquierda) vs Descenso por Gradiente Estocástico (derecha). Tomada de Optimizers for machine Learning, R. Mohammad, Medium

1.11.2 Optimizador Adam

El optimizador SGD se utiliza en todo tipo de problemas de aprendizaje automático. No obstante, con la llegada del *Deep Learning*, los modelos pasaron a tener muchísimos parámetros y aunque el SGD solo calcula el gradiente de un subconjunto en cada iteración, los entrenamientos se prolongaban durante mucho tiempo y era necesario encontrar técnicas más eficientes.

Por esta razón, para problemas de *Deep Learning*, hay optimizadores específicos que siguen apoyándose en la idea del gradiente, pero con algunas modificaciones. Uno de los más conocidos y utilizados para el entrenamiento de redes CNN es el optimizador Adam.

Adam es un método de aprendizaje adaptativo que ajusta un *learning rate* para cada parámetro. Tanto en el Descenso por Gradiente como en el SGD, cada iteración del entrenamiento modifica los pesos apoyándose en el *learning rate* para cuantificar la magnitud de la variación y todos los parámetros tienen el mismo *learning rate*.

Adam, sin embargo, ajusta el *learning rate* de cada parámetro y así consigue que cada iteración reduzca el error más de lo que lo haría SGD. De esta forma, se necesitan menos iteraciones de entrenamiento.

1.11.3 Optimizador AdaBound

Durante muchos años, Adam ha predominado como uno de los mejores optimizadores y se ha utilizado en gran cantidad de *papers* y artículos. No obstante, los investigadores siempre buscan la forma de mejorar aún más estas técnicas y en 2019 Liangchen et. al. [6] publicaron un artículo en el ICLR que definía un nuevo optimizador llamado AdaBound que prometía ser mejor que Adam.

AdaBound es una variante de Adam que utiliza *learning rates* dinámicos para conseguir una transición gradual y moderada al SGD. Inicialmente, AdaBound comienza siendo un optimizador Adam y poco a poco se convierte en un SGD.

Según los autores del artículo, AdaBound es muy robusto a cambios en los hiperparámetros y obtiene mejores resultados que Adam en menos iteraciones. Este optimizador es tan novedoso que no aparece en los *frameworks* de *Deep Learning* populares como *PyTorch* y *Tensorflow*.

Actualmente, solo existe una implementación de AdaBound para *PyTorch* publicada por los autores del artículo en *GitHub* [4] y para utilizarlo es necesario importar el código fuente como una librería externa.

En este trabajo, se compararán los tres optimizadores para probar si AdaBound acaba siendo mejor que Adam y el SGD.

1.12 Herramientas y tecnologías

El área del *Deep Learning* ha tenido un gran desarrollo tanto a nivel teórico como práctico. En la actualidad existen *frameworks* que implementan todo lo necesario para entrenar y desplegar modelos de *Deep Learning* sin necesidad de llevar a cabo una programación intensiva. Algunos de estos *frameworks* son *Tensorflow, Pytorch* y *Keras*.

Tensorflow está desarrollado por Google y cuenta con lo necesario para trabajar con Redes Neuronales de todo tipo y viene acompañado de uno de los complementos que mayor interés ha generado en la comunidad en los últimos años: *Tensorboard*.

Tensorboard es una herramienta que se combina con *Tensorflow* para estudiar y comparar diseños de modelos, métricas de entrenamiento y cualquier otro parámetro que se quiera monitorizar. Se trata de un cuadro de mando que puede personalizarse con pocas líneas de código.

PyTorch es el *framework* de *Deep Learning* desarrollado por Microsoft y, al igual que *Tensorflow*, cuenta con herramientas para trabajar con todo tipo de redes. En comparación con *Tensorflow*, *PyTorch* tiene a sus espaldas menos años de desarrollo, pero se ha convertido en una alternativa que cada vez está tomando más protagonismo.

PyTorch se apoya en la programación orientada a objetos para diseñar los modelos de *Deep Learning* y también permite el uso de *Tensorboard*. Comparándolo con *Tensorflow*, se puede decir que *PyTorch* trabaja a más bajo nivel porque, desde que *Tensorflow* adquirió *Keras* dentro de su librería, muchas de las tareas han quedado abstraídas en funciones de más alto nivel.

Para este proyecto, el *framework* que se ha seleccionado es *PyTorch* porque ya se tenían conocimientos previos de *Tensorflow* y este trabajo era una oportunidad para aprender una herramienta nueva.

Respecto al lenguaje de programación, se ha utilizado *Python* porque es el lenguaje más recomendado por los desarrolladores. Además, *Python* cuenta con una amplia variedad de librerías para análisis de datos y *Machine Learning* que se complementan a la perfección con los *frameworks* de *Deep Learning*.

El código se ha desarrollado sobre *Jupyter Notebooks* utilizando el editor *Visual Studio Code* que, además, permite administrar el control de versiones con *Git* y gestionar los paquetes de *Python* con *Anaconda* de forma sencilla en un mismo entorno.

1.13 Planteamiento del problema

El problema por resolver consiste en clasificar imágenes de logos comerciales de 23 marcas distintas utilizando un modelo predictivo basado en Redes Neuronales de Convolución (CNN). Para ello, se dispone de un *dataset* de imágenes con los logos comerciales recortados. No será necesario localizarlos en la imagen, solo clasificarlos.

1.14 Metodología

La metodología que se ha utilizado en el trabajo ha estado basada en *springs*. Los objetivos se han planteado para obtener resultados lo antes posible y empezar a iterar sobre diferentes modelos para mejorarlos. Inicialmente se definió un modelo muy sencillo para tomarlo como punto de partida.

Una vez se obtuvieron los primeros resultados, se trató de mejorar el diseño base aplicando técnicas que dificultaban el entrenamiento y trataban de conseguir que la red fuera más precisa. Llegado un determinado momento, el diseño no podía mejorarse mucho más y se pasaba a una arquitectura diferente. En total, se han probado 5 diseños y se han tratado de optimizar para comprobar cuál era la precisión máxima que podían obtener.

Para medir la eficacia de los modelos, se ha utilizado el *macro Fscore*, que combina la precisión y la cobertura en una única métrica que está normalizada entre 0 y 1.

Para entrenar y evaluar los modelos, se dividieron los datos de entrada en tres subconjuntos: entrenamiento, validación y test. La división de conjuntos se hizo implementando una función que aseguraba un reparto uniforme de las clases, de manera que todas tuvieran la misma cantidad de imágenes en cada conjunto.

Por cada modelo, se estudió con *Tensorboard* la curva que describía el *Fscore* durante el entrenamiento en el conjunto de datos entrenamiento y validación. Al final del entrenamiento, se analizaba el *Fscore* que se obtenía en el conjunto de test tanto a nivel global como a nivel de clase y así, se podía averiguar cuáles eran las clases más difíciles de predecir.

Además, el entrenamiento se programó para que el modelo resultante fuera aquel que hubiera obtenido el mayor *Fscore* en los datos de validación y, en caso de igualdad, el que hubiera obtenido mayor *Fscore* en los datos de entrenamiento. Para ello, después de cada época de entrenamiento, se calculaba el *Fscore* sobre el conjunto de entrenamiento y de validación y se comparaba con el mejor resultado obtenido hasta ese momento. Si el modelo actual era mejor, se exportaba a un fichero, y al terminar todas las épocas se cargaba el mejor de todos.

Todos los modelos se han exportado a ficheros con extensión .pth utilizando el método basado en el diccionario de parámetros. Este método consiste en guardar los parámetros entrenados sin almacenar la arquitectura del modelo. Para cargarlo es necesario disponer del código que contiene la definición del diseño. Primero se debe crear un modelo con parámetros aleatorios y luego sustituir esos parámetros por los que están guardados en el fichero *.pth*.

1.15 Estructura del proyecto

El proyecto comienza con un análisis exploratorio de los datos donde se detallan las dimensiones de las imágenes, la distribución de datos y los rangos de valores, entre otras cosas (Ap. 2.1). A continuación, se explica el método utilizado para la carga de datos en forma de *batches,* junto con todas las clases de *PyTorch* necesarias para ello (Ap. 2.2).

Seguidamente, se introduce el diseño de una CNN básica y se analizan las métricas de efectividad. Después, se detallan las técnicas que se pueden utilizar para ayudar al modelo a generalizar más y que obtenga mejores resultados (Ap. 2.3).

La CNN básica tiene un límite de eficacia y se opta por una arquitectura de red más avanzada para intentar extraer más características de las imágenes y así poder conseguir una mayor precisión (Ap. 2.4).

La CNN avanzada no consigue mejorar lo suficiente respecto a la línea base y se replantea el problema dividiendo las clases en grupos según su dificultad. Esto da lugar a un modelo compuesto por 4 CNNs que se ensambla en una red *Feed Forward* (Ap. 2.5).

El modelo ensamblado tampoco consigue grandes avances y al final se opta por el *Transfer Learning* usando dos arquitecturas de red: *ResNet* y *GoogLeNet*. Ambas consiguen superar la línea base y se intenta mejorar ResNet aumentando la profundidad del modelo (Ap. 2.6). Finalmente, se despliega la CNN con una aplicación de Streamlit (Ap. 2.8)

1.16 Resultados

Inicialmente se fijó una línea base con una CNN sencilla que se llamó CNN básica y que tenía un *FScore* de 0.501 sobre los datos de test. Se probó a aplicar técnicas de regularización como *Dropout* y regularizadores L2 y técnicas de *Data Augmentation*, pero ninguno consiguió mejorar el resultado base.

Se probó con una arquitectura más compleja llamada CNN avanzada y se intentó dividir las clases en niveles de dificultad. Por cada nivel se entrenó un modelo y los modelos de cada nivel se ensamblaron, pero tampoco tuvo éxito.

Los resultados consiguieron mejorarse utilizando técnicas de *Transfer Learning*. En concreto, se usaron dos arquitecturas: ResNet18 y GoogLeNet y ambas superaron los 0.7 de *FScore*. Se intentó mejorar el modelo ResNet utilizando arquitecturas más profundas como ResNet34 y ResNet50, pero los resultados fueron idénticos.

También se intentó utilizar *Data Augmentation*, pero el resultado acabó siendo menos equilibrado, así que, se tomó como definitivo el ResNet18 sin *Data Augmentation.* Finalmente, se desplegó el modelo utilizando una aplicación desarrollada con Streamlit.

Capítulo 2. MEMORIA TÉCNICA

2.1 Análisis exploratorio de los datos de partida

Los datos de partida para crear el modelo son imágenes recortadas de logos comerciales de 23 marcas distintas. Para cada marca hay 70 imágenes disponibles, todas son a color (tienen los tres canales RGB), sus dimensiones son de 256 x 256 y los valores que toman los píxeles al cargarlos en el entorno de *PyTorch* están normalizados entre 0 y 1.

Las imágenes se encuentran organizadas en carpetas siguiendo una estructura como la que se muestra en la *figura 17*. Las imágenes ya están recortadas, así que, no es necesario localizar los logos, solo hay que clasificarlos en la clase correcta. Se trata de un problema de *Logo Recognition* (LR), no de *Logo Detection* (LD).

Figura 17. Organización de datos de partida

Asimismo, los logos se pueden presentar de distintas formas. En su versión más sencilla, aparecen en el centro de la imagen, sobre un fondo blanco y sin ningún tipo de ruido como se muestra en la *figura 18*.

Figura 18. Ejemplos de logos centrados sobre fondo blanco y sin ruido

También es posible encontrar el logo estampado en la superficie de un producto. En este caso, puede haber imágenes donde el producto se vea claramente sobre un fondo blanco como se ve en la *figura 19* o puede ser una foto tomada por una cámara como en la *figura 20*.

Figura 19. Ejemplos de logos en productos sobre fondo blanco

Figura 20. Ejemplos de logos en productos fotografiados por una cámara

Además, hay ocasiones donde una misma empresa ha tenido diferentes versiones de su logo a lo largo de los años y todas esas variantes también se tienen en cuenta. Véase cómo ha cambiado el logo de Aquarius a lo largo de los años como se muestra en la *figura 21*.

Figura 21. Versiones por las que ha pasado el logo de Aquarius

Por otra parte, existe la posibilidad de que haya imágenes con mucho ruido y que sean prácticamente irreconocibles. Por ejemplo, en la *figura 22* y *23* se muestran algunos logos recogidos en la carpeta de Milka y AMD que están incompletos, difuminados o distorsionados.

Figura 22. Logos de Milka incompletos, difuminados o distorsionados

Figura 23. Logos de AMD incompletos, difuminados o distorsionados

Por último, algunas marcas presentan imágenes con logos de otras empresas que también se están intentando reconocer por el clasificador. Por ejemplo, en la *figura 23* se muestran imágenes de la carpeta de LG que tienen el logo de Pac-Man, Sony y Samsumg, y todas ellas son marcas que el clasificador puede reconocer porque forman parte del conjunto de datos.

Figura 24. Imágenes de LG con logos de otras marcas

Con todas estas casuísticas en cuenta, es posible que el clasificador tenga ciertas dificultades para conseguir un *FScore* alto en determinadas marcas. Igualmente, todo lo que se muestra en el trabajo son resultados obtenidos de considerar todas las imágenes que aparecen en el conjunto de datos de partida.

2.2 Carga de datos

Una vez concluido el análisis exploratorio, llega el momento de definir cómo va a ser la entrada de datos del modelo. Para ello, es detallarán las clases de *PyTorch* que se han utilizado y cuáles son las dependencias que existen entre cada una de ellas.

2.2.1 Clase *ImageFolder*

PyTorch utiliza el estilo de programación orientado a objetos para implementar los modelos, así que, es frecuente que se tengan que implementar clases que hereden de otras ya predefinidas para hacer tareas como cargar datos y diseñar la arquitectura de la red.

Si bien, no siempre es necesario definir la entrada de datos y el diseño del modelo a nivel de clase, porque el entorno ya cuenta con muchas implementaciones que cubren las necesidades más comunes de los problemas de *Deep Learning*. Por ejemplo, para el reconocimiento de imágenes se utiliza la clase *ImageFolder* contenida dentro del paquete *torchvision.datasets*.

Usando *ImageFolder*, basta con indicar la ruta donde se encuentran los datos para que se puedan cargar en memoria, saber cuántas clases hay, y qué etiqueta de clase le corresponde a cada imagen. No obstante, para poder utilizar *ImageFolder*, es necesario que los datos de entrada estén dispuestos como se ha mostrado en la *figura 17*, utilizando una subcarpeta para cada clase.

2.2.2 Aplicando *Transforms*

Cuando se define una entrada de datos en *PyTorch*, es posible aplicar transformaciones (*Transforms*) sobre ellos durante el proceso de carga. Por ejemplo, en el caso de las imágenes, se pueden hacer operaciones de escalado, zoom, rotación, traslación y modificaciones de color que se pueden concatenar y combinar.

Es importante destacar que, aunque no se quiera hacer ninguna modificación sobre los datos de entrada, existe una transformación que es indispensable para poder cargar los datos y es la transformación *ToTensor*, que convierte las imágenes de entrada a tensores de *PyTorch*.

La razón de esta operación es que, por defecto, las imágenes se cargan como objetos *PIL* (*Python Imaging Library*) que pertenecen a la librería *Pillow* de *Python*. Sin embargo, para que puedan ser usados por *PyTorch*, deben utilizar los tipos de datos de *PyTorch*, en concreto, los tensores.

2.2.3 Procesamiento *batch* con *DataLoader*

La clase *ImageFolder* permite cargar imágenes una por una de forma lineal siguiendo el orden alfabético de las clases. Esto quiere decir que, si la primera clase de todas (por orden alfabético) es AMD, *ImageFolder* cargará las 70 imágenes de AMD, luego las 70 imágenes de la siguiente clase y así hasta pasar por todas.

Sin embargo, para entrenar un modelo de aprendizaje, esta forma de introducir los datos no es apropiada porque si se cargan todas las imágenes de una sola clase de forma consecutiva, el modelo se ajustará mucho a ella para reducir el error y luego, cuando se pase a la siguiente clase, el modelo se ajustará mucho a la segunda y así hasta pasar por todas las clases. El resultado al final del entrenamiento será que no llegará a generalizar bien cuando reciba nuevos datos.

Además, desde el punto de vista de la eficiencia, cargar las imágenes una por una puede alargar el tiempo de entrenamiento al tener que estar iniciando constantemente operaciones con la unidad de almacenamiento.

Por ello, la alternativa que se usa para cargar los datos es el procesamiento *batch*, que consiste en cargar bloques de imágenes cada vez que se hace una operación de entrada/salida.

Dentro de *PyTorch*, existe una clase llamada *DataLoader* que, utilizando un *Dataset* definido en *PyTorch* como *ImageFolder*, ofrece la funcionalidad de procesamiento *batch* sin necesidad de implementarlo manualmente. Además, cuenta con un parámetro opcional para hacer *shuffle* y que los datos se carguen de manera aleatoria. Así, se evitan los problemas de cargar las imágenes una por una de manera lineal.

2.2.4 *Pipeline* básico de entrada de datos

Por lo general, una vez se define la entrada de datos, lo único que habrá que hacer para entrenar el modelo será llamar a la instancia de *DataLoader*. De esta forma, se obtendrán conjuntos de imágenes de todas las clases a las que se les habrán aplicado las transformaciones oportunas. El resumen del *pipeline* de entrada se puede ver en la *figura 25*.

A nivel de implementación, desde el punto de vista de la jerarquía de clases, *ImageFolder* hereda de la clase *Dataset* y sobrescribe algunos de sus métodos para crear datasets de imágenes. Por otra parte, la clase *DataLoader* se apoya en objetos *Dataset* para ofrecer la funcionalidad del procesamiento *batch*.

Con esta estructura, cualquier conjunto de datos que se cargue en memoria usando una clase que herede de *Dataset* puede conseguir la funcionalidad de procesamiento *batch* solo con utilizar después la clase *DataLoader*.

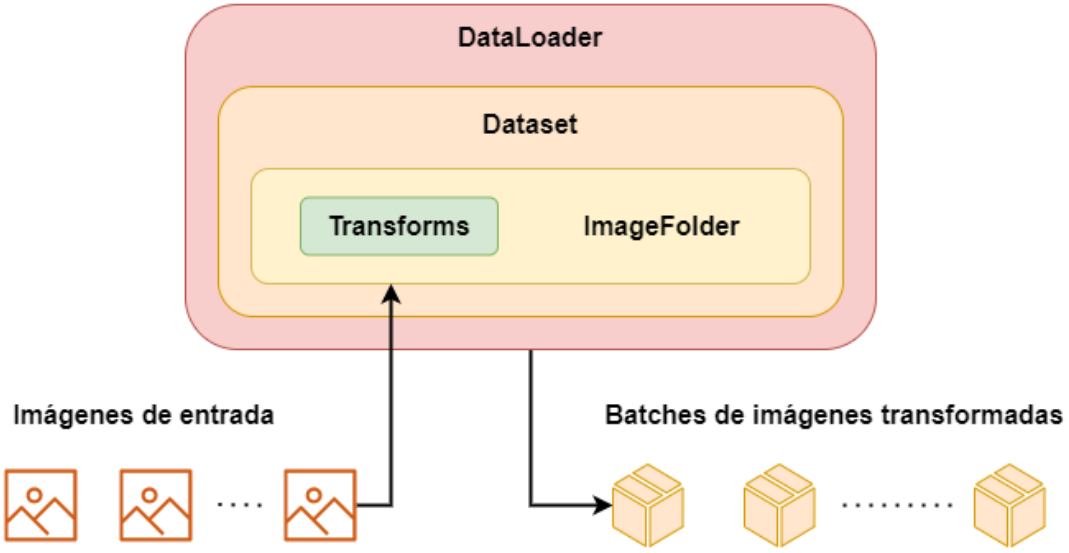

Figura 25. Diagrama de clases involucradas en el pipeline de entrada de datos

2.2.5 Train – Test Split

Para entrenar un modelo de aprendizaje automático y evaluar su rendimiento, es necesario dividir los datos disponibles en dos subconjuntos como mínimo: uno de entrenamiento y otro de test. *PyTorch* contempla la posibilidad de dividir un *dataset* y ofrece funciones como *random_split*.

Sin embargo, esta función no asegura que la división de las clases sea uniforme. Por ejemplo, si se quiere hacer una división 80 % - 20 % para entrenamiento y test, si cada clase tiene 70 imágenes, una división uniforme sería tener 56 imágenes de entrenamiento y 14 de test, pero *random_split* no lo hace así.

En su lugar, *random_split* utiliza el conjunto de imágenes completo para hacer la división y asegura que, uniendo todas las imágenes de cada conjunto, se respeten los porcentajes. No obstante, puede haber clases que tengan muchas imágenes de test y pocas de entrenamiento, y viceversa.

Para conseguir una división uniforme, se ha implementado una función que hace la división del *dataset* a nivel de sistema operativo. Esta función crea dos árboles de directorios iguales a los de la *figura 17* y copia las imágenes en las subcarpetas correspondientes respetando los porcentajes de división.

Además, se asegura de que las clases estén balanceadas para que todas tengan las mismas imágenes de entrenamiento y test, y que no haya solapamiento, es decir, una imagen de entrenamiento no estará en el conjunto de test, y viceversa.

Al hacer la división a nivel de sistema operativo, para cargar los datos es necesario definir dos *pipelines* de entrada como el que se muestra en la *figura 25*, uno para entrenamiento y otro para test. La ventaja de esto es que se pueden utilizar *Transforms* distintos.

Por un lado, se pueden aplicar transformaciones a los datos de entrenamiento para hacer *Data Augmentation* como rotaciones, zoom, traslaciones y cambios de color y, por otro, dejar los datos de evaluación intactos aplicando las transformaciones mínimas necesarias para que puedan usarse en el modelo.

Para este trabajo, se ha dividido el *dataset* en 70% entrenamiento, 10% validación y 20% test.

2.2.6 Estructura de un *batch*

Una vez definidos los *DataLoaders*, basta con iterar sobre ellos para extraer tanto las imágenes como las clases a las que pertenecen. Los *batches* se estructuran en forma de tupla donde el primer elemento es el *batch* y el segundo es un vector de enteros con el índice de la clase a la que pertenece cada imagen.

$$next(iter(dataloader) = (batch, labels)$$

Los *batches* a su vez se estructuran como una cuádrupla donde se recoge: la imagen, el canal de color, las filas y las columnas.

$$batch = (imagen, canal, alto, ancho)$$

Para ejemplificar cómo es un *batch* de imágenes, en la *figura 26* se muestra un *batch* con 10 logos. El título de la imagen muestra la clase a la que pertenece cada logo, pero es importante destacar que el vector *labels* es un vector de enteros que recoge los índices de los nombres de las clases y lo que se muestra en el título es la clase que le corresponde a cada índice.

Figura 26. Ejemplo de un batch de imágenes

2.3 Diseño básico de una CNN

2.3.1 Arquitectura

La arquitectura con la que se va a establecer la línea base del proyecto se muestra en la *figura 27*, donde se detallan las capas que intervienen con su dimensión de salida. En este diseño, la parte convolucional cuenta con 2 pares de capas de convolución y *pooling*, y la parte *Feed Forward* tiene 2 capas ocultas.

Todas las funciones de activación son ReLu siguiendo las recomendaciones de los artículos científicos que se han consultado. Las capas de convolución usan *kernels* de dimensiones 5 x 5 y aplican *padding* para evitar que la dimensionalidad se reduzca. Asimismo, las capas de *pooling* aplican *kernels* de 2 x 2 y por ello, después de cada pooling, las dimensiones se reducen a la mitad.

La función de pérdida que se usa en todo el trabajo es la *Categorical Crossentropy*, así que, en todos los diseños siempre habrá una capa de *Softmax* al final.

Las imágenes de entrada se han redimensionado para pasar de 256 x 256 a 64 x 64. Esto permite a la red trabajar con menos parámetros y el tiempo de entrenamiento es más corto, pero se pierden los detalles pequeños. Esta consecuencia se va a asumir en la línea base y en futuras iteraciones se aumentará la resolución.

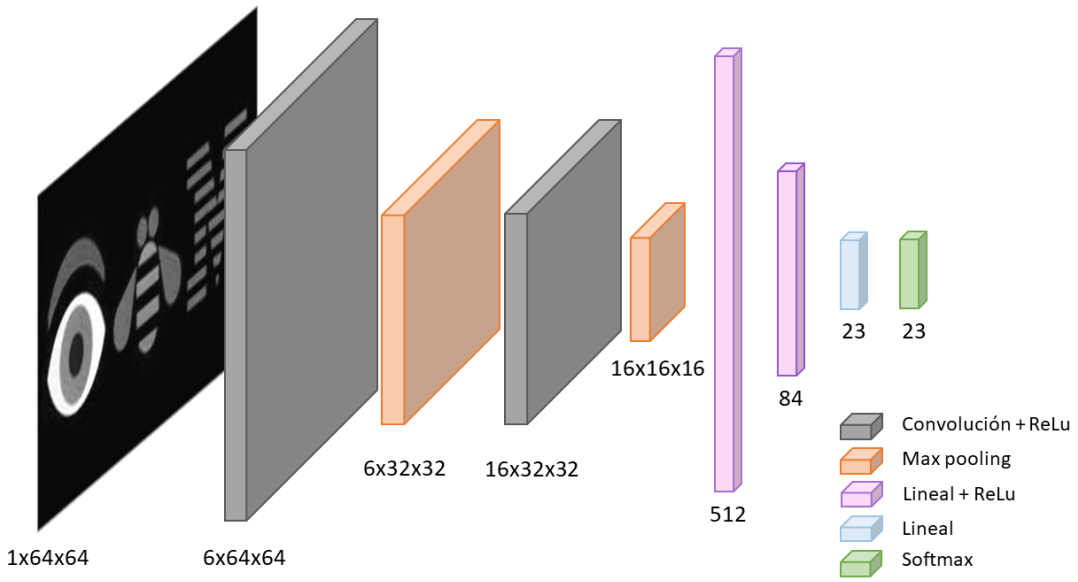

Figura 27. Arquitectura de CNN básica

Además, las imágenes estarán expresadas en escala de grises. Esto se debe a que hay algunas marcas que tienen colores muy característicos y la CNN podría apoyarse únicamente en este aspecto para hacer la clasificación.

Sin embargo, el color no debería ser un factor determinante para clasificar una marca. Si a una persona se le muestra una imagen en escala de grises de un logo comercial, es capaz de reconocerlo así que, la red debe extraer la información suficiente como para poder identificarlo también.

A modo de ejemplo, en la *figura 28* se muestran algunas marcas con colores muy destacados como KitKat (rojo), Lipton (amarillo) y Milka (morado), y debajo de ellos, su equivalente en escala de grises.

Además, la transformación a blanco y negro funciona como una técnica de regularización que ayuda a reducir el sobreajuste de la red y, en menor medida, reduce el número de parámetros a optimizar, con lo cual se acelera el entrenamiento. De ahora en adelante, en todas las arquitecturas de red que se van a mostrar, las imágenes siempre pasan por la transformación en escala de grises.

Asimismo, es importante tener en cuenta que, si la CNN está diseñada para recibir imágenes con un solo canal de color, cuando se vaya a evaluar el modelo, las imágenes también tienen que pasar por esta transformación. Por lo tanto, los logos de evaluación y de test también estarán en blanco y negro.

Figura 28. Comparación de logos a color y en escala de grises

2.3.2 Comparación de optimizadores

Para la optimización de los parámetros de la CNN básica se van a estudiar tres optimizadores: Descenso por Gradiente Estocástico (SGD), Adam y AdaBound. El más popular de todos ellos es Adam, un optimizador pensado para entrenar modelos de *Deep Learning* que se ha utilizado en muchos artículos científicos y que es ampliamente recomendado para entrenar Redes Neuronales de Convolución.

Si bien, AdaBound es un optimizador muy reciente que se publicó en un artículo en 2019 bajo la premisa de que era capaz de conseguir mejores resultados que Adam y el SGD. En este trabajo, se pretende poner a prueba esa afirmación comparando los tres bajo el mismo conjunto de datos y utilizando la misma arquitectura de red.

Por defecto, AdaBound no está incluido dentro de los *frameworks* de *Deep Learning* como *Tensorflow o PyTorch*. Solo existe una implementación publicada por los autores del artículo [6] en *GitHub* [4] que es válida para *PyTorch* y que se debe importar como una librería externa.

La configuración que se va a utilizar para el entrenamiento se muestra en la *tabla 2*. Para todos los casos se utiliza un 70% de las imágenes para entrenamiento, 10% para validación y 20% para test. Todos los entrenamientos se hacen sobre el mismo conjunto de datos y todos entrenan durante 100 épocas.

La única diferencia está en el *Learning Rate* debido a que Adam y AdaBound necesitan comenzar con un valor más bajo que el SGD porque sino no consiguen aprender. Se planteó reducir el *Learning Rate* del SGD para que fuera igual al de Adam y AdaBound, pero era demasiado bajo y después de las 100 épocas la mejora era mínima.

Nótese que para estos entrenamientos no se está aplicando ningún tipo de regularización, por lo que es muy fácil que estos modelos acaben sobre ajustando. Sin embargo, el objetivo del experimento es comprobar cuál de los optimizadores converge más rápido y cuál acaba teniendo mejores resultados sobre el conjunto de test.

Optimizador	SGD	Adam	AdaBound
Imágenes de entrenamiento por clase	49	49	49
Imágenes de validación por clase	7	7	7
Imágenes de test por clase	14	14	14
Épocas	100	100	100
Tamaño del *batch*	10	10	10
Learning Rate	0.001	0.0001	0.0001

Tabla 2. Configuración de entrenamiento de la CNN básica

Después de cada época de entrenamiento se evaluó el *Fscore* que obtenían los modelos en el conjunto de *train* y de validación. Los resultados sobre el conjunto de *train* se muestran en la *figura 29*.

Observando estos resultados, se aprecia que todos alcanzan el valor máximo de FScore. Sin embargo, el SGD es el que tarda más tiempo en hacerlo y en segundo lugar queda Adam. AdaBound es el que consigue converger más rápido y en solo 10 épocas casi alcanza el valor máximo de *Fscore*.

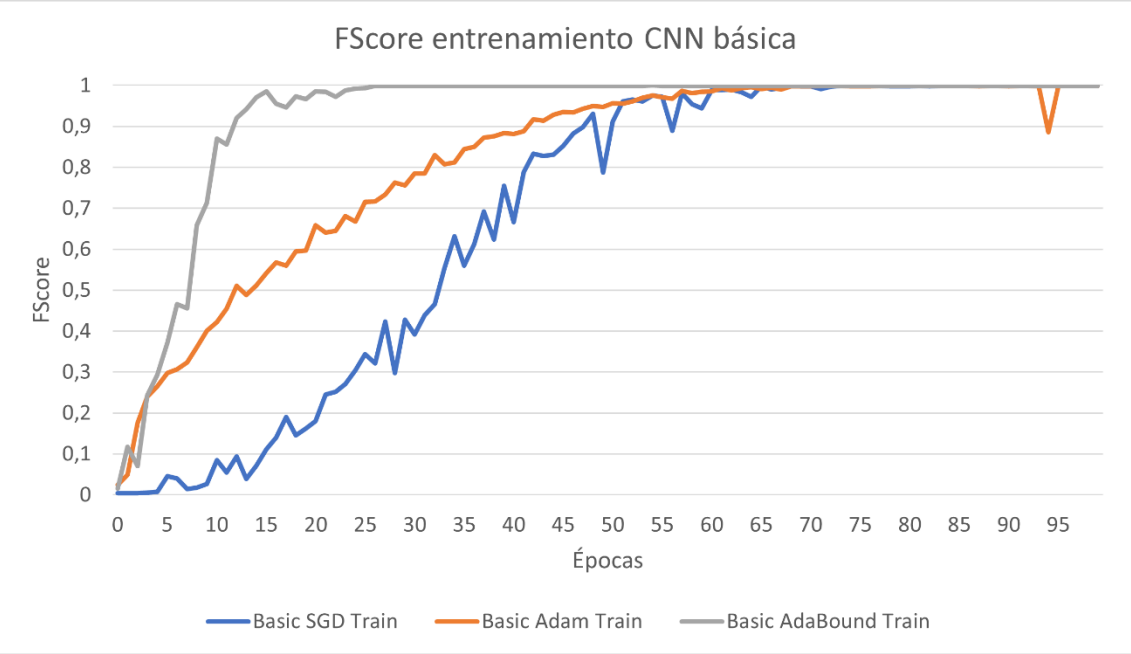

Figura 29. Curva de FScore de la CNN básica sobre el conjunto de entrenamiento para comparar optimizadores

Los resultados sobre el conjunto de validación están recogidos en la *figura 30* y, en todos los casos, los modelos no consiguen superar el 0.5 en el *FScore*. Esto evidencia que se está produciendo un sobreajuste considerable, pero este aspecto se tratará de resolver en experimentos posteriores.

Sobre el conjunto de validación, se repiten los mismos resultados que sobre el conjunto de entrenamiento. El SGD es el que tarda más en converger y tiene la curva más irregular, en segundo lugar, queda Adam y AdaBound se posiciona como el optimizador más rápido hasta el momento.

Además del *FScore*, durante el entrenamiento también se ha medido la evolución del **error** para los tres optimizadores y los resultados se muestran en la *figura 31*. Como podía intuirse por los valores previos, el SGD es el que necesita más épocas para reducir el error y es el que tiene la curva más irregular, en segundo lugar, queda Adam y el más rápido es AdaBound.

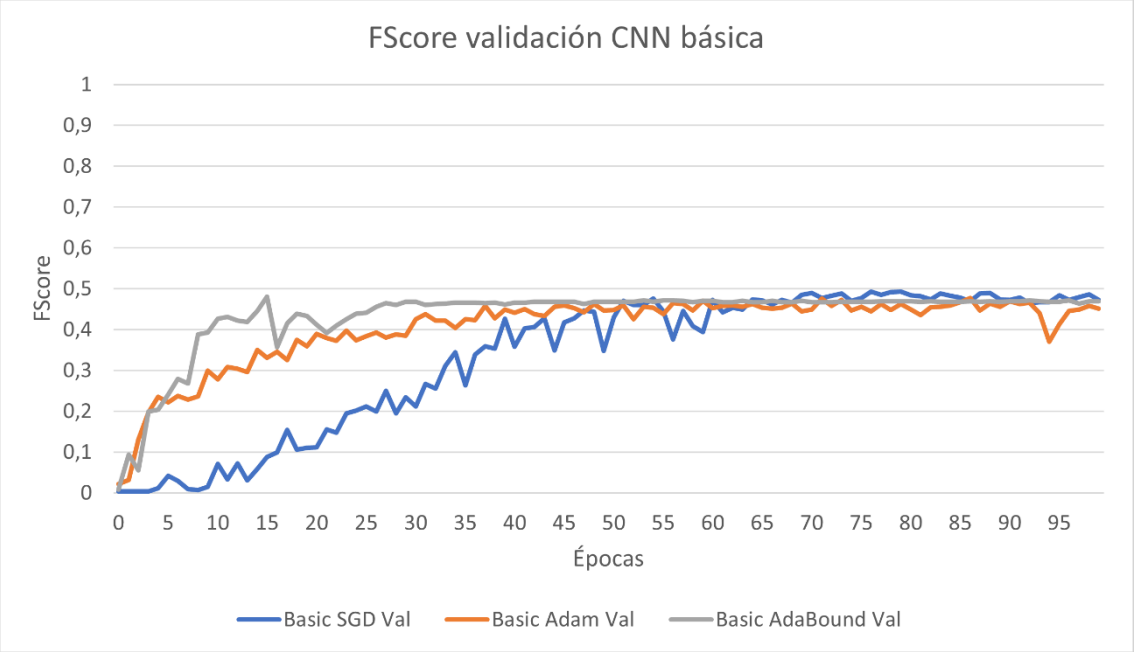

Figura 30. Curva de FScore de la CNN básica sobre el conjunto de validación para comparar optimizadores

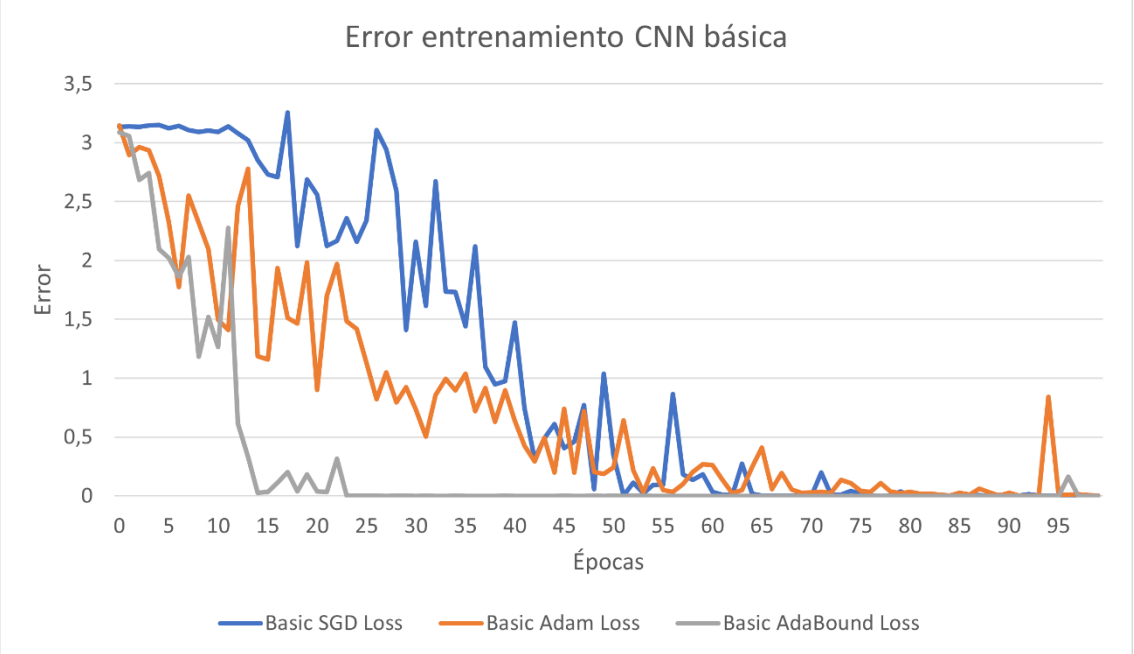

Figura 31. Curva de error de la CNN básica sobre el conjunto de entrenamiento para comparar optimizadores

Por último, para concluir con el experimento, se evaluó el modelo entrenado con cada optimizador sobre los datos de test. Los resultados se muestran en la *tabla 3* y por lo que se observa, se mantiene el mismo ranking. El SGD es el que más dificultad ha tenido para generalizar sobre datos que no ha visto en el entrenamiento, seguidamente queda Adam y el mejor es AdaBound.

Optimizador	SGD	Adam	AdaBound
FScore en conjunto de test	0.415	0.447	0.501

Tabla 3. FScore de la CNN básica con cada optimizador sobre los datos de test

Teniendo en cuenta los resultados, parece que AdaBound es el mejor optimizador, al menos para la arquitectura de CNN básica con la que se está trabajando. Este mismo experimento se repetirá para un diseño de red más complejo para confirmar si AdaBound es realmente mejor que Adam y el SGD.

Por lo que respecta a la CNN básica, para todos los experimentos que se harán sobre este diseño se utilizará AdaBound con un *learning rate* de 0.001.

2.3.3 Entrenamiento de la CNN básica sin regularizadores

Tras concluir que AdaBound era el mejor optimizador para la arquitectura de CNN básica, se analizó cuál era la eficacia del modelo en cada clase para averiguar cuáles eran más difíciles de predecir. Los resultados se muestran en la *figura 32* y se puede observar que hay una gran dispersión en los valores.

Por un lado, hay clases que el modelo predice con gran facilidad como D-Link, Domino's Pizza y Tic Tac, pero hay otras donde tiene serias dificultades como Pac-Man, Aquarius y Aquafina.

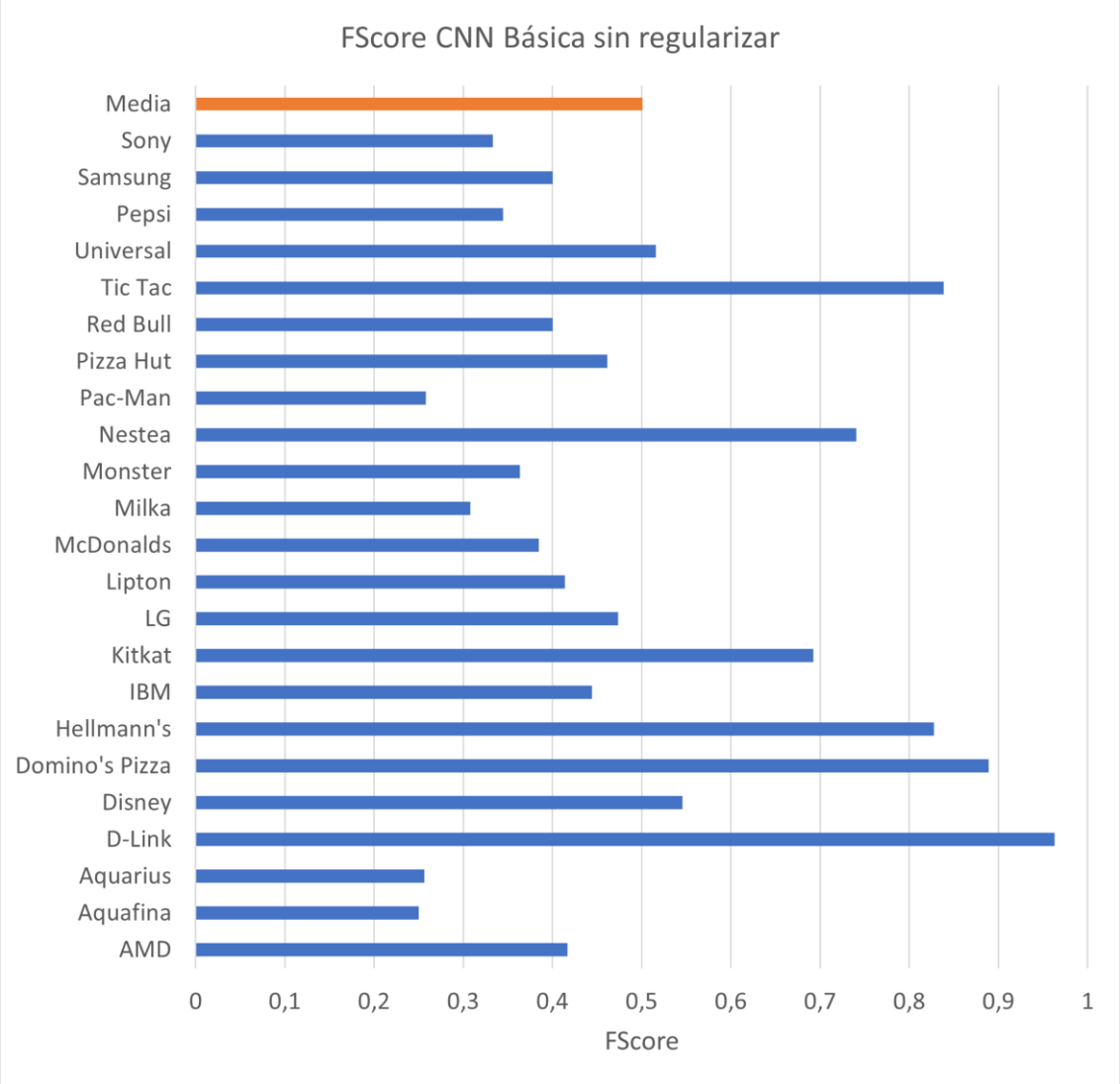

Figura 32. FScore de la CNN básica sin regularizar por cada clase en los datos de test

Para entender por qué se estaban produciendo estas diferencias tan notables, se estudiaron los logos que había en las marcas que mejor y peor predecía el modelo. Lo que se observó fue que todas las imágenes de D-Link, Domino's Pizza y Tic Tac son muy similares entre sí mientras que las de Pac-Man, Aquarius y Aquafina eran muy dispares y algunas tenían mucho ruido.

A modo de ejemplo, en la *figura 33* se muestran algunas imágenes de Pac-Man que, aunque un humano sabe identificar que son de Pac-Man, la CNN tiene muchas dificultades porque hay pocos patrones comunes entre ellas.

Por lo tanto, se puede decir que, hasta este momento, el modelo es capaz de predecir casos relativamente sencillos, pero sufre mucho cuando empiezan a aparecer distorsiones en la imagen o los logos de una misma marca no muestran patrones comunes.

Figura 33. Imágenes de Pac-Man poco relacionadas

2.3.4 Entrenamiento de la CNN básica con regularizadores

Para intentar que el modelo sea capaz de generalizar más y pueda hacer mejores predicciones, se pueden utilizar una serie de técnicas que dificultan el entrenamiento para evitar el sobreajuste. En concreto, se han utilizado capas de *Dropout* y regularizadores L2.

2.3.4.1 Capas de Dropout

Las capas de *Dropout* se basan en la idea de que una red neuronal debe seguir funcionando correctamente, aunque un porcentaje de los nodos no estén operativos. El objetivo de esta capa es "apagar" algunas neuronas de la red durante el entrenamiento para que este principio se cumpla.

Su funcionamiento es bastante sencillo: dado un conjunto de valores, la capa de *Dropout* los convierte a 0 con una probabilidad *p*. Estas capas se sitúan en la red *Feed Forward* y así obliga a la red a trabajar con menos información. En el modelo básico las capas de *Dropout* se añadieron antes de cada capa *Linear*, dando lugar a un total de 3 capas de *Dropout*.

2.3.4.2 Regularización L1 y L2

Los regularizadores L1 y L2 se basan en la idea de que todas las neuronas deben tener un peso similar en la red y que no debe haber neuronas con más protagonismo que otras. Cuando una neurona tiene mucho peso, quiere decir que se está dando mucha importancia a determinadas características y que se están ignorando otras que podrían ser importantes.

Los regularizadores L1 y L2 penalizan los valores de peso elevados con un pequeño porcentaje sobre la función de pérdida. De esta forma, se evita que los parámetros del modelo acaben tomando valores muy elevados.

La función de coste utilizando regularizadores L1 y L2 queda de la siguiente forma:

$$Coste\ con\ L1 = Función\ de\ pérdida + \lambda * |w|$$

$$Coste\ con\ L2 = Función\ de\ pérdida + \lambda * w^2$$

Siendo el parámetro λ un factor que regula la importancia de la penalización. A este parámetro también se lo conoce como *weight decay*.

2.3.4.3 Análisis del entrenamiento con regularizadores

Tanto las capas de *Dropout* como los regularizadores L2 dependen de un factor que regule el impacto de la regularización. Para determinar cuáles eran las mejores combinaciones de parámetros, se entrenaron 7 modelos variando la probabilidad de *Dropout* y el *weight decay*. Todos se entrenaron 50 épocas y se evaluaron sobre el conjunto de test.

La metodología que se ha seguido es modificar primero uno de los parámetros y dejar el otro sin efecto y después combinarlos los dos. Los resultados de *FScore* se muestran en la *tabla 4* comparados con los que se obtuvieron en el entrenamiento sin aplicar ningún tipo de regularización.

Aparentemente, parece que la regularización ha tenido un efecto negativo en los resultados porque ninguno mejora al caso sin regularizar. Observando el *Dropout*, el valor que mejor ha funcionado es el de 0.4, y en el caso del *weight decay*, el mejor ha sido el 0.01.

Lo ideal podría ser combinar estos dos valores para entrenar un modelo. Sin embargo, se observó que la regularización era demasiado fuerte y el modelo no conseguía aprender. De hecho, el modelo era más robusto soportando un aumento en la probabilidad de *Dropout*, pero era más sensible aumentando el *weight decay*.

Por lo tanto, se decidió utilizar un *dropout* de 0.6 y un *weight decay* de 0.001. Los resultados de este modelo se pueden ver en la última fila de la tabla y, a pesar de la regularización, no obtiene mejores resultados en el conjunto de test que la versión sin regularizar.

Dropout	*Weight Decay*	FScore en datos de test
0	0	0.501
0.2	0	0.462
0.4	0	0.470
0.6	0	0.450
0	0.0001	0.454
0	0.001	0.459
0	0.01	0.474
0.6	0.001	0.451

Tabla 4. Entrenamiento de CNN básica con diferentes parámetros para los regularizadores

Para visualizar mejor el impacto que habían tenido los regularizadores, se comparó la evolución del *Fscore* en el modelo sin regularizar y regularizado, considerando la versión con 0.6 en *Dropout* y 0.001 en *weight decay*.

Los resultados se muestran en la *figura 34* y lo que se puede observar es que la versión con regularizadores no llega a converger en el punto máximo, sino que se mantiene oscilando hasta concluir las 100 épocas.

Esto demuestra que los regularizadores están actuando para dificultar el entrenamiento y, por ello, las curvas son más irregulares. Sin embargo, en el conjunto de validación no se consigue superar el 0.5 de *FScore* y, además, el sobreajuste es demasiado grande en cualquiera de los casos. Por lo tanto, la regularización no es suficiente y es necesario aplicar otras técnicas.

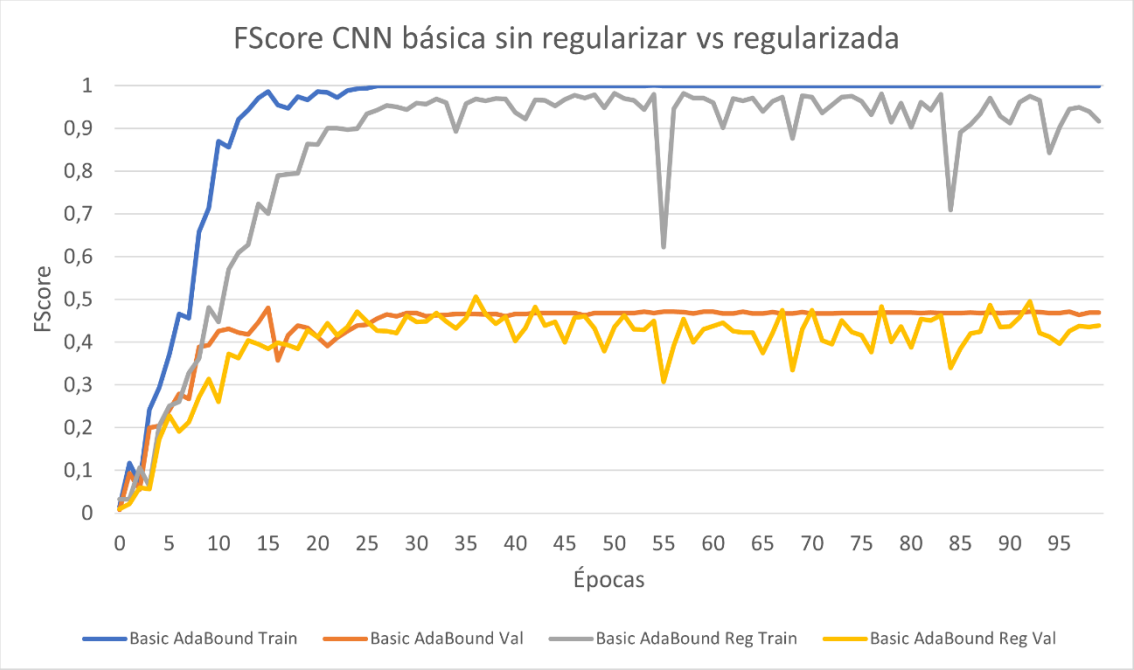

Figura 34. Evolución FScore en CNN básica sin regularizar y regularizada

2.3.5 Entrenamiento de la CNN básica con *Data Augmentation*

El *Data Augmentation* consiste en generar datos artificiales a partir de transformar los datos de entrenamiento. Para el caso de problemas de clasificación de imágenes, se pueden aplicar transformaciones como rotación, traslación, zoom y cambios de color, entre otras.

Estas transformaciones se pueden aplicar de manera aleatoria con una probabilidad *p* para que el modelo reciba tanto los datos originales sin transformar como datos artificiales generados con *Data Augmentation*.

Para este experimento, se consideraron dos transformaciones sencillas basadas en aplicar un efecto espejo horizontal, un espejo vertical o ambas al mismo tiempo. A modo de ejemplo, se puede observar el efecto de estas transformaciones en la *figura 35*. La primera imagen es la original, la segunda tiene un espejo horizontal, la tercera uno vertical y la cuarta tiene ambas.

Figura 35. Efecto de aplicar las transformaciones de Data Augmentation

Dado que las transformaciones se aplican según una probabilidad *p*, se estudió el efecto del *Data Augmentation* con tres probabilidades distintas. Los resultados se muestran en la *tabla 5* comparados con la ejecución sin aplicar *Data Augmentation*.

Lo que se observa es que el *Data Augmentation* no ha conseguido superar al modelo base y que el aumento en la probabilidad de transformación ha tenido un efecto mínimo. Por lo tanto, tampoco es una solución al problema del sobreajuste y es necesario probar otra alternativa.

Probabilidad de transformación	FScore en datos de test
0	0.501
0.1	0.477
0.2	0.467
0.3	0.483

Tabla 5. Resultados de aplicar Data Augmentation en la CNN básica

2.3.6 Conclusiones del diseño de CNN básica

La comparación de optimizadores SGD, Adam y AdaBound han posicionado a AdaBound como el mejor optimizador y es el que se ha utilizado para todos los experimentos que se han hecho con el diseño de CNN básica.

La CNN básica consigue tener un *FScore* muy alto en los datos de entrenamiento. Sin embargo, no consigue generalizar en el conjunto de validación y de test, y el *FScore* queda por debajo del 0.5. Esto quiere decir que se está produciendo un sobreajuste considerable.

Probando con técnicas de regularización como *Dropout* y regularizadores L2, no se ha conseguido mejorar los resultados. De hecho, no se consigue superar al modelo sin regularizar. Lo que sí es cierto, es que la regularización afecta a la curva de *FScore* durante el entrenamiento evitando que converja al valor máximo.

Asimismo, aplicando *Data Augmentation*, tampoco se ha conseguido superar al modelo base y las variaciones en la probabilidad de transformación de las imágenes han tenido un efecto mínimo.

En definitiva, es necesario probar otra vía para intentar que el modelo obtenga más información de las imágenes, pueda generalizar mejor y así, se evite el sobreajuste.

2.4 Modelo avanzado

Después de probar que la CNN básica no mejora el *FScore* aplicando diferentes técnicas, se planteó la siguiente hipótesis: aumentar la complejidad del modelo en la parte convolucional para que se extraigan más características de las imágenes y así se mejore el *FScore.*

Para probar si esta hipótesis era correcta, se repitieron los mismos experimentos que se hicieron con la CNN básica y se compararon.

2.4.1 Arquitectura

La nueva arquitectura de red, que a partir de ahora se nombrará CNN avanzada, se muestra en la *figura 36* detallando las dimensiones de salida de cada capa. En su parte convolucional, cuenta con 4 pares de capas de convolución y *pooling,* y en su parte *Feed Forward, tiene* 2 capas ocultas. Se sigue utilizando ReLu como función de activación y la *Categorical Crossentropy* como función de pérdida, por ello se usa la capa de *Softmax* al final.

Se ha duplicado la resolución de las imágenes de entrada para poder capturar mejor los detalles pequeños y que sea posible aplicar más etapas de convolución y *pooling*. Al igual que en la CNN básica, los *kernels* que se utilizan en las capas de convolución son de 5 x 5 y se ha utilizado *padding* para evitar que se reduzca la dimensión. Asimismo, las capas de *pooling* usan *kernels* de 2 x 2, así que, las dimensiones de salida se dividen a la mitad.

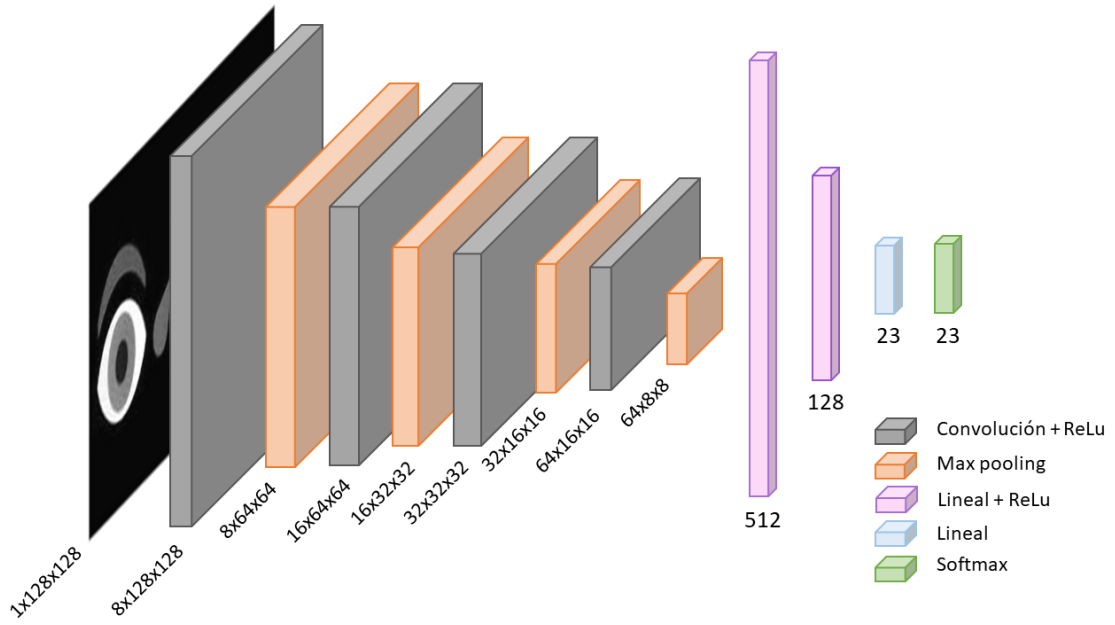

Figura 36. Arquitectura de CNN avanzada

2.4.2 Comparación de optimizadores

Al igual que se hizo con la CNN básica, para la CNN avanzada se compararon los optimizadores SGD, Adam y AdaBound para terminar de concluir si, verdaderamente, AdaBound obtenía mejores resultados que Adam y el SGD.

Probablemente sean necesarias muchas más pruebas para dar por concluyente esta afirmación, pero estos experimentos permiten fijar el optimizador que se usará en los próximos diseños de CNNs de este trabajo.

Los datos utilizados y la configuración del entrenamiento son los mismos que los que se usaron para la CNN básica. Como recordatorio, se vuelven a detallar los parámetros de configuración en la *tabla 6*. Asimismo, tampoco se ha aplicado ningún tipo de regularización a los modelos.

Optimizador	SGD	Adam	AdaBound
Imágenes de entrenamiento por clase	49	49	49
Imágenes de validación por clase	7	7	7
Imágenes de test por clase	14	14	14
Épocas	100	100	100
Tamaño del *batch*	10	10	10
Learning Rate	0.01	0.001	0.001

Tabla 6. Configuración de entrenamiento de la CNN avanzada

Después de cada época de entrenamiento se evaluó el *Fscore* que obtenían los modelos en el conjunto de *train* y de validación. Los resultados sobre el conjunto de *train* se muestran en la *figura 29*.

Lo que se puede observar es que el SGD es el que necesita más épocas para alcanzar el valor máximo de *FScore*, seguidamente queda Adam y el más rápido es AdaBound. Con esta arquitectura, el SGD no ha tenido una curva tan irregular como le ocurrió con la CNN básica y, aunque AdaBound consigue converger más rápido que Adam, la diferencia entre ambos es menor a la que había con la CNN básica.

Los resultados de medir el *FScore* sobre los datos de validación se muestran en la *figura 38* y se vuelve a observar la misma barrera de los 0.5 en el *FScore*. Además, en este caso, el resultado de Adam y AdaBound es prácticamente idéntico, así que, hay un empate en esta comparación y el que queda claramente por debajo es el SGD.

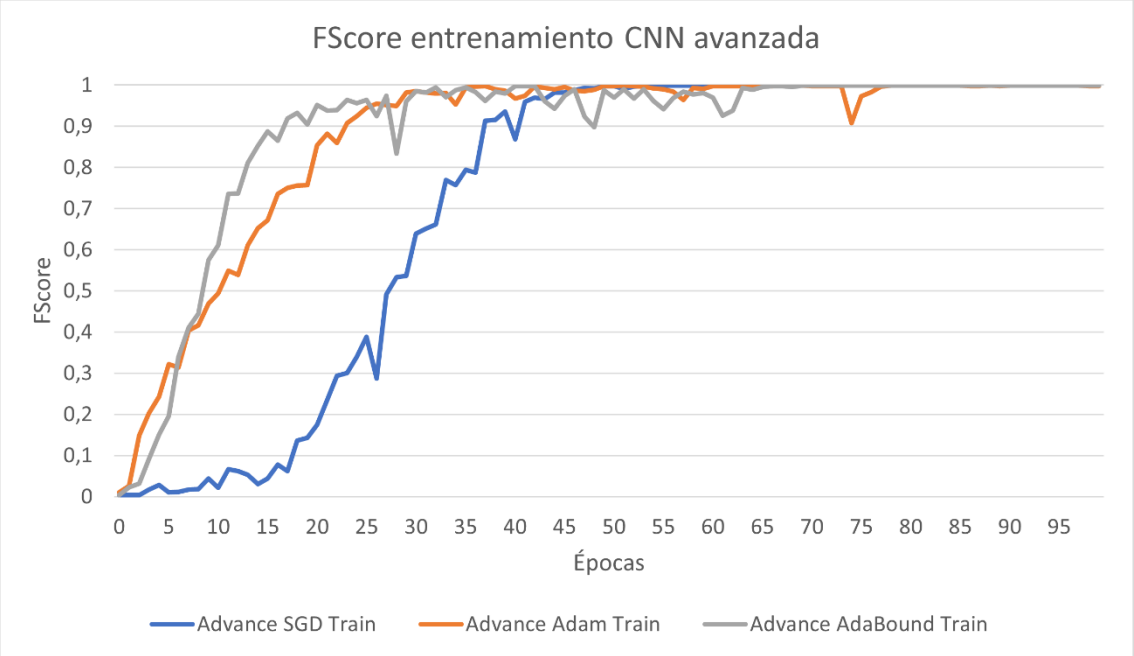

Figura 37. Curva de FScore de la CNN avanzada sobre el conjunto de entrenamiento para comparar optimizadores

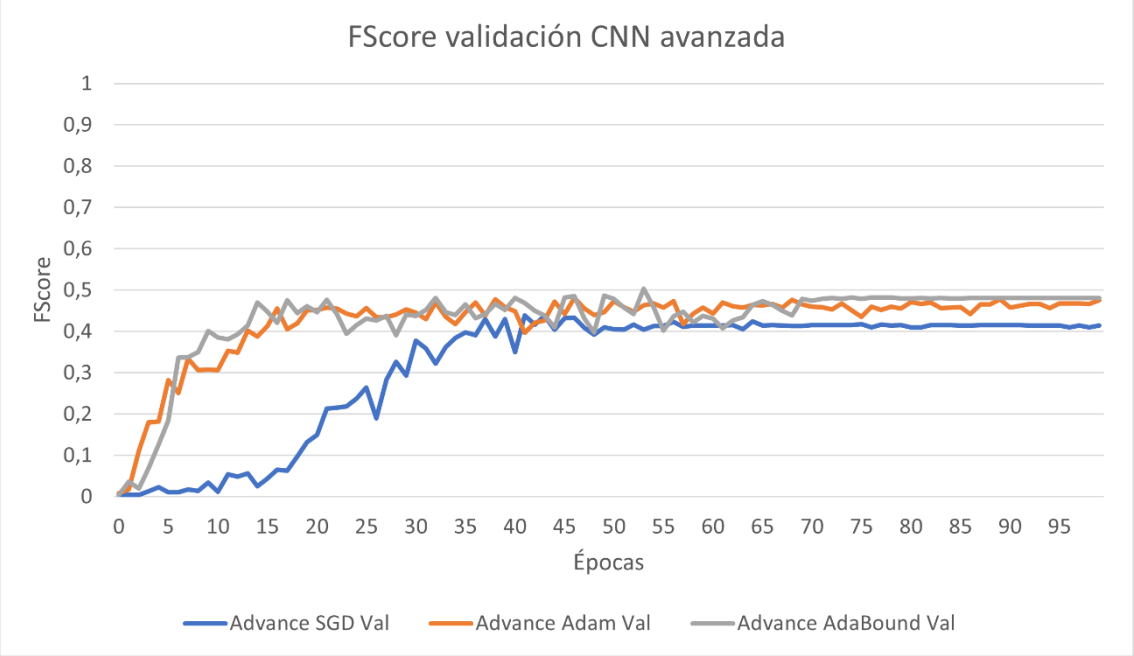

Figura 38. Curva de FScore de la CNN avanzada sobre el conjunto de validación para comparar optimizadores

Por otra parte, la evolución del error en el entrenamiento con cada optimizador se muestra en la *figura 39*. Los resultados evidencian que tanto Adam como AdaBound reducen el error a un ritmo similar. De hecho, AdaBound parece ser más irregular y presenta picos muy notables. Asimismo, una vez más, el SGD es el que más tarda en reducir el error y queda como el optimizador más lento.

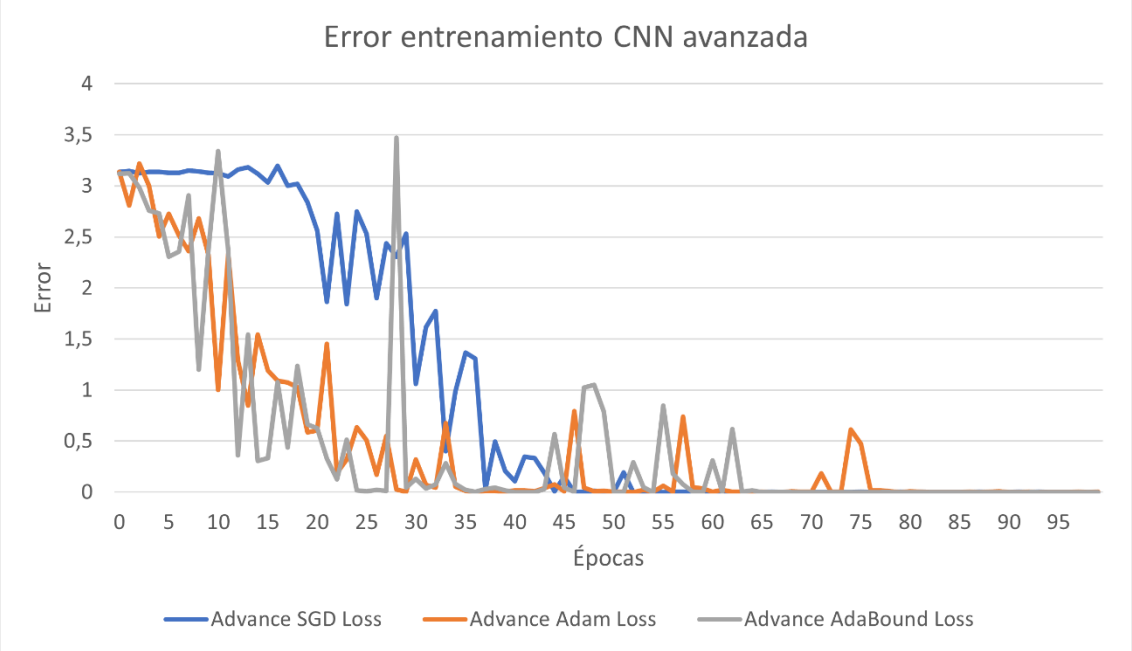

Figura 39. Curva de error de la CNN básica sobre el conjunto de entrenamiento para comparar optimizadores

Por último, se evaluó el *FScore* de los modelos sobre el conjunto de test y los resultados se muestran en la *tabla 7* junto con los que se obtuvieron con la CNN básica. Lo que se observa en todos los casos es que AdaBound es el que mejor generaliza sobre datos desconocidos y, como se ha comprobado, es el que más rápido aprende. Por lo tanto, para las siguientes arquitecturas de CNNs se usará como optimizador AdaBound.

Optimizador	SGD	Adam	AdaBound
FScore CNN básica en conjunto test	0.415	0.447	0.501
FScore CNN avanzada en conjunto test	0.419	0.434	0.477

Tabla 7. FScore de la CNN avanzada con cada optimizador sobre los datos de test

2.4.3 Entrenamiento de la CNN avanzada sin regularizadores

Para estudiar cómo actuaba la CNN avanzada en cada una de las clases, se desglosó el *FScore* y se comparó con el desglose que se hizo con la CNN básica. Los resultados se muestran en la *figura 40* y lo que se observa es que las fortalezas y debilidades de ambas no son las mismas. Por un lado, la CNN básica es mejor que la CNN avanzada prediciendo Samsung, Pizza Hut, Nestea, Monster, McDonalds, Lipton y KitKat, y, por otro lado, la CNN avanzada es mejor en Sony, Universal, LG, Hellmann's, Domino's Pizza, Aquarius y AMD.

Observando esta situación, cabe pensar que, si ambos modelos se combinaran, se podrían compensar las debilidades de uno con las fortalezas del otro y viceversa. Esta idea se desarrollará más adelante en otro apartado. Por lo general, el *FScore* medio de ambos es similar.

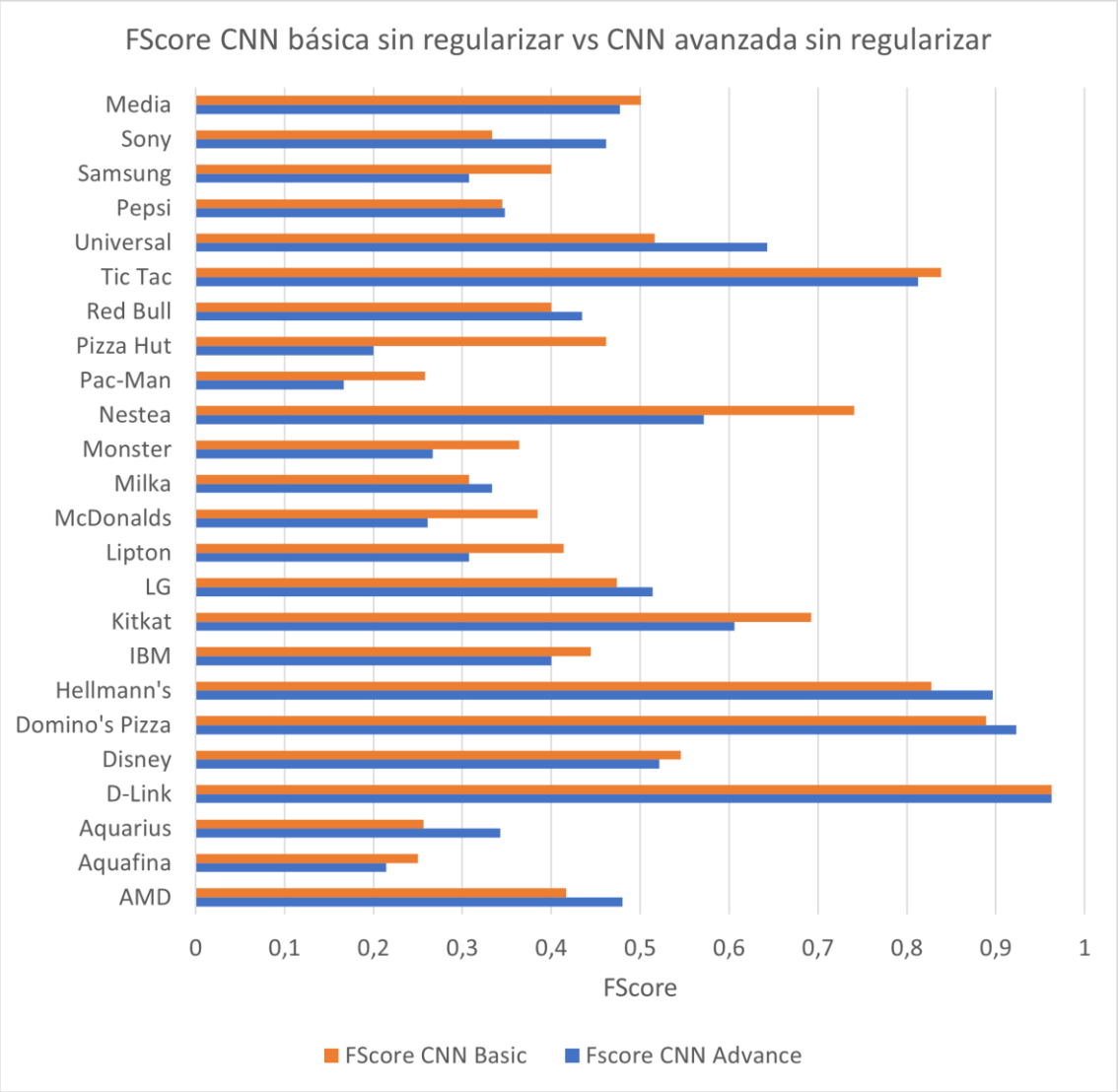

Figura 40. FScore de la CNN avanzada sin regularizar por cada clase en los datos de test

2.4.4 Entrenamiento de la CNN avanzada con regularizadores

Para regularizar la CNN avanzada se retomaron las técnicas de *Dropout* y la regularización L2. Se entrenaron 7 modelos con diferentes parámetros de regularización y se midió el *FScore* que obtenían en los datos de test. Los resultados se muestran en la *tabla 8* comparados con la versión sin regularizar de la CNN básica y la CNN avanzada.

Lo que se puede observar es que la CNN básica sin regularizar está por encima de la gran mayoría de las versiones que aplican regularización, exceptuando una de ellas que aplica *Dropout* de 0.6 y *weight decay* de 0.

Además, hay un caso en el que el modelo no ha conseguido aprender, concretamente en *Dropout* igual a 0 y *weight decay* igual a 0.01. Como se adelantó con la CNN básica, parece que el modelo es más sensible a aumentar el *weight decay* que a aumentar la probabilidad de *Dropout*. Por esta razón, la combinación de las dos regularizaciones se ha hecho tomando un *Dropout* alto (0.6) y un *weight decay* más moderado (0.001).

En líneas generales, la regularización tampoco parece ayudar al modelo avanzado a mejorar el *FScore* y será necesario aplicar otra técnica.

Modelo	Dropout	*Weight Decay*	FScore en datos de test
CNN básica	0	0	0.501
CNN avanzada	0	0	0.477
CNN avanzada	0.2	0	0.470
CNN avanzada	0.4	0	0.470
CNN avanzada	0.6	0	0.506
CNN avanzada	0	0.0001	0.421
CNN avanzada	0	0.001	0.499
CNN avanzada	0	0.01	0.004*
CNN avanzada	0.6	0.001	0.467

Tabla 8. Entrenamiento de CNN avanzada con diferentes parámetros para los regularizadores

Para visualizar el impacto que han tenido los regularizadores en el modelo avanzado, se comparó la curva de *FScore* sin regularizar con la regularizada. El resultado se muestra en la *figura 41* y lo que se comprueba es que la regularización hace que la curva tenga más dificultades para converger y sea más irregular. De hecho, al terminar las 100 épocas, no parece haber llegado a la convergencia. Respecto al resultado en el conjunto de validación, sigue sin superarse los 0.5 en el *FScore* y el sobreajuste es demasiado alto.

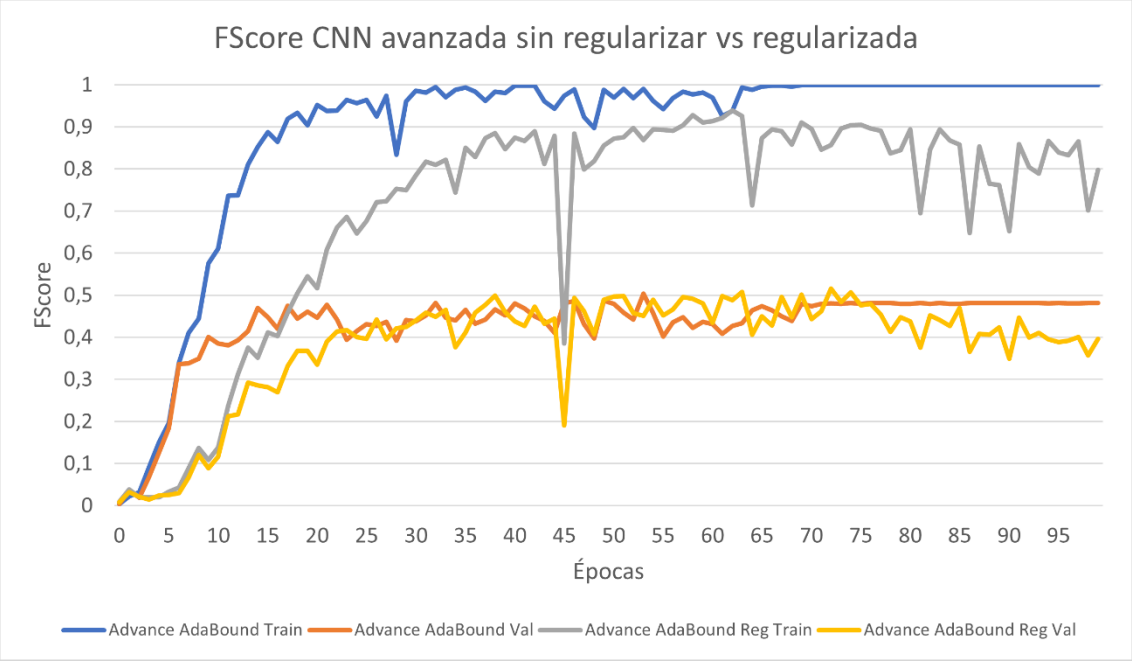

Figura 41. Evolución FScore en CNN básica sin regularizar y regularizada

2.4.5 Entrenamiento de la CNN avanzada con *Data Augmentation*

El último experimento que se aplicó sobre la CNN avanzada fue utilizar las mismas técnicas de *Data Augmentation* que se usaron con la CNN básica. Las transformaciones posibles eran un espejo horizontal, uno vertical o ambos al mismo tiempo.

Se entrenaron 3 modelos utilizando distintas probabilidades de transformación y los resultados se muestran en la *tabla 9,* comparados con las versiones sin regularizar de la CNN básica y avanzada.

Observando el *FScore* que se obtuvo en los datos de test, parece que la versión de CNN avanzada con una pequeña probabilidad de transformación ha obtenido el mejor resultado hasta el momento, superando al de la CNN básica.

Sin embargo, aumentando levemente la probabilidad de transformación, el *FScore* se reduce considerablemente. Así que, no parece ser un modelo muy robusto a cambios en los hiperparámetros y es probable que el resultado haya superado a la línea base por suerte.

Esto quiere decir que el *Data Augmentation*, tampoco consigue que la CNN avanzada generalice y mejore el *FScore* de la línea base. Por lo tanto, será necesario utilizar otra aproximación al problema.

Modelo	Probabilidad de transformación	FScore en datos de test
CNN básica	0	0.501
CNN avanzada	0	0.477
CNN avanzada	0.1	0.527
CNN avanzada	0.2	0.475
CNN avanzada	0.3	0.443

Tabla 9. Resultados de aplicar Data Augmentation en la CNN avanzada

2.4.6 Conclusiones del diseño de CNN avanzada

Después de repetir todos los experimentos que se hicieron sobre la CNN básica en una nueva arquitectura con más capas convolucionales, se pueden extraer las siguientes conclusiones.

Por un lado, la comparación de optimizadores ha permitido comprobar que, al menos en los modelos de este trabajo, AdaBound es el optimizador más rápido y el que consigue generalizar mejor. Así que, será el que se use en los próximos diseños de CNN.

Por otro lado, la hipótesis de aumentar las capas convolucionales para extraer más características y mejorar el *FScore*, ha resultado ser incorrecta. Todos los experimentos han quedado al mismo nivel o por debajo de la CNN básica, tanto en los datos de validación como en los de test.

No obstante, comparando cuáles son las fortalezas de la CNN básica y la CNN avanzada, parece que cada arquitectura tiene ventajas donde la otra tiene debilidades y viceversa. Por lo tanto, es posible que, si se combinan, se compensen las fortalezas de uno con las debilidades del otro. Esta será la hipótesis que se pondrá a prueba en el siguiente diseño de CNN.

2.5 Modelo ensamblado

Aumentar la complejidad de un único modelo ha demostrado no ser suficiente para conseguir buenos resultados sobre el conjunto de test. Sin embargo, las dos arquitecturas que se han probado parecen tener fortalezas que se pueden complementar. Por ello, la hipótesis que se plantea en este diseño es combinar ambos modelos en un modelo ensamblado.

Además, después de analizar los resultados de todos los experimentos que se han hecho hasta el momento, se ha observado que hay clases que son muy difíciles de predecir en comparación con otras.

Por esta razón, se ha decidido tomar un enfoque *Divide y Vencerás* para abordar el problema. Las clases se han agrupado en 4 niveles de dificultad de acuerdo con los criterios de la *tabla 10*, siendo el nivel 0 el más sencillo y el nivel 3 el más complejo.

Nivel 0	La gran mayoría de las imágenes presenta un logo similar
	La gran mayoría de las imágenes presenta el logo sin ruido, en el centro, sobre un fondo blanco.
Nivel 1	La mayoría de las imágenes presenta un logo similar
	Algunas imágenes están rotadas o no están en el centro.
	Hay imágenes que tienen un ruido leve.
Nivel 2	Hay varias versiones de diseño para el logo de una misma empresa.
	Hay imágenes que tienen el logo muy pequeño, no está centrado o está rotado.
	Hay imágenes que no presentan el logo en sí mismo, sino una referencia al producto de la empresa.
Nivel 3	Hay muchas versiones de diseño para el logo de una misma empresa.
	Muchas de las imágenes son fotos tomadas con una cámara y el logo no se ve bien.
	Hay imágenes con mucho ruido.

Tabla 10. Criterios para la división de clases en niveles de dificultad

Aplicando estos criterios, las 23 clases del problema quedaron divididas en los grupos que se muestran en la *tabla 11*. En los niveles de dificultad 0 y 1 quedan 7 clases, en el nivel 2 hay 6 y en el nivel de mayor dificultad hay 3.

Para ejemplificar de forma gráfica cómo actúan estos criterios, en la *figura 42* se recogen logos de cada uno de los niveles de dificultad.

Nivel 0	D-Link	Nestea	Hellmann's	Tic Tac
	Domino's Pizza	Pepsi	KitKat	
Nivel 1	IBM	Lipton	Pizza Hut	Universal
	LG	Milka	Samsumg	
Nivel 2	Disney	Pac-Man	Monster	
	McDonalds	Red Bull	Sony	
Nivel 3	AMD	Aquafina	Aquarius	

Tabla 11. Detalle de las clases divididas por niveles de dificultad

Figura 42. Ejemplos de logos en cada uno de los niveles de dificultad

Una vez hecha la división de clases, para cada nivel se entrenó un modelo utilizando la arquitectura de CNN básica y de CNN avanzada para comprobar cuál de las dos funcionaba mejor y combinar las fortalezas de ambas en el modelo final.

La configuración que se siguió para estos entrenamientos se muestra en la *tabla 12*. Todos los modelos se entrenaron 50 épocas con AdaBound y se aplicaron regularizadores para evitar el sobreajuste.

Épocas	50
Tamaño del *batch*	10
Optimizador	AdaBound
Learning Rate	0.0001
Dropout	0.6
Weight decay	0.001

Tabla 12. Configuración de entrenamiento de los módulos del modelo ensamblado

Los resultados del entrenamiento se pueden ver en la *tabla 13* y lo que se puede destacar es que la división por clases ha resultado ser un éxito para algunos de los niveles. En el nivel 0, tanto la CNN básica como la avanzada han tenido un *FScore* muy superior al de la línea base, llegando a superar el 0.8 para el caso del modelo básico.

En el nivel 1, la CNN avanzada ha sido la que mejor resultado ha obtenido, quedando levemente por encima del 0.5 de *FScore*. El nivel 2 es el peor, pues ninguno de los dos modelos ha conseguido superar el 0.5 de *FScore*. Por último, el nivel 3 ha llegado al 0.66 de *FScore* con el modelo avanzado. No obstante, este nivel tiene más facilidad porque solo tiene que distinguir entre 3 clases.

Nivel	Modelo	FScore en datos de test	Seleccionado
0	CNN básica	**0.834**	Seleccionado
0	CNN avanzada	0.794	-
1	CNN básica	0.478	-
1	CNN avanzada	**0.521**	Seleccionado
2	CNN básica	**0.473**	Seleccionado
2	CNN avanzada	0.411	-
3	CNN básica	0.644	-
3	CNN avanzada	**0.664**	Seleccionado

Tabla 13. Resultados del entrenamiento de los módulos del modelo ensamblado

Para analizar estos resultados, es importante tener en cuenta que, al dividir un problema de 23 clases a problemas más pequeños, los resultados de *FScore* no se pueden comparar directamente con los de la línea base porque la dificultad de la tarea es distinta. Solo se podrá hacer una comparación justa cuando se ensamble el modelo y se evalúe.

Atendiendo a los resultados sobre los datos de test, el modelo ensamblado estará formado por 2 CNNs básicas (nivel 0 y 2) y 2 CNNs avanzadas (nivel 1 y 3). Para ensamblarlos y hacerlos trabajar como un único modelo, se va a utilizar una red *Feed Forward* sencilla, creando así una arquitectura como la que se muestra en la *figura 43*.

El flujo de cálculo consiste en introducir la imagen de entrada en cada uno de los submodelos y aquel que haya sido entrenado para identificar la marca que se representa en la imagen, debería obtener la salida más alta.

Sin embargo, es posible que el rango de valores de salida sea distinto entre dos modelos. Por ejemplo, que un modelo devuelva una salida de 1,5 puede suponer una salida muy alta para uno, pero puede ser muy baja para otro. Por tanto, la red *Feed Forward* que va a ensamblar todos los modelos, debe aprender a ponderar las salidas de cada uno de ellos y darle la importancia que mejor se adapte a la salida esperada.

La red *Feed Forward* tendrá una capa oculta de 23 neuronas cuya función activación será ReLu. Durante el entrenamiento de esta red, los submodelos permanecerán congelados, es decir, no se optimizarán ninguno de sus pesos.

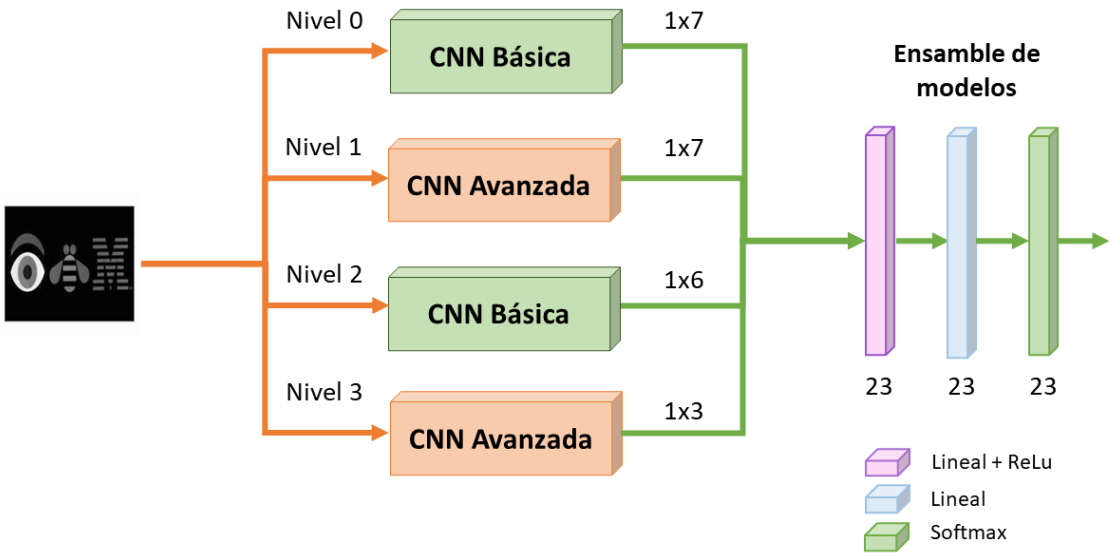

Figura 43. Arquitectura de modelo ensamblado

La configuración que se siguió para el entrenamiento del modelo ensamblado se muestra en la *tabla 14*. En la parte *Feed Forward* no se van a considerar capas de *Dropout* porque podría eliminar la predicción correcta y esto no tendría mucho sentido desde el punto de vista de diseño. Lo que sí se ha utilizado son regularizadores L2.

Épocas	100
Tamaño del *batch*	10
Optimizador	AdaBound
Learning Rate	0.0001
Weight decay	0.001

Tabla 14. Configuración entrenamiento modelo ensamblado

La curva de *Fscore* que describe el modelo ensamblado durante su entrenamiento se puede apreciar en la *figura 44* comparado con el de la CNN básica sin regularizar. Lo que se observa es que el modelo ensamblado no alcanza el máximo valor de *FScore* en los datos de entrenamiento y en los datos de validación no se consigue superar el 0.5. De hecho, ambas curvas quedan por debajo de las de la CNN básica en todo momento.

Por lo tanto, parece que el modelo ensamblado no consigue superar la línea base y el sobreajuste tampoco se ha conseguido solucionar.

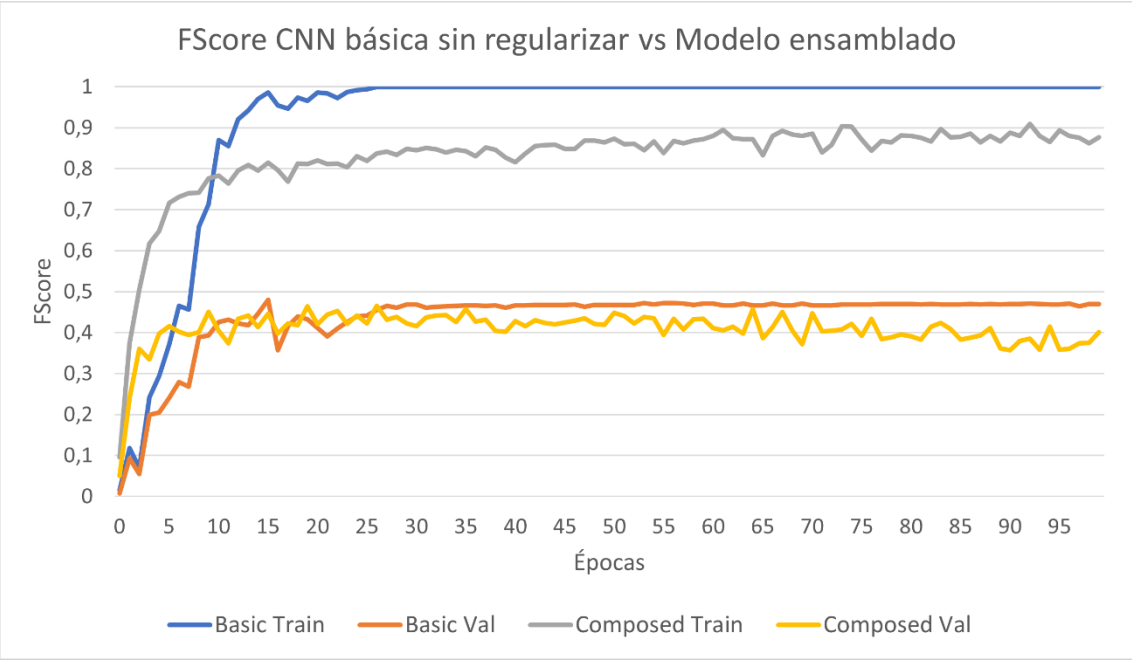

Figura 44. FScore de entrenamiento y validación del modelo ensamblado y de la CNN básica

En lo que respecta a la evolución del error, en la *figura 45* se puede observar cómo ha variado en el entrenamiento del modelo ensamblado comparado con el de la CNN básica sin regularizar. Lo que se observa es que el modelo ensamblado no llega a converger y es bastante irregular, mientras que el de la CNN básica alcanza rápidamente el valor 0.

La razón de esto puede deberse al del regularizador L2. Recuérdese que el *weight decay* añade un factor a la función de error que está ponderado por los pesos de la red y, como la mayoría de los pesos de la red será mayor que 0, este factor siempre estará influyendo en la función de error.

Por lo tanto, es posible que el error real haya convergido y que lo que se está mostrando en la gráfica sea la contribución del *weight decay*.

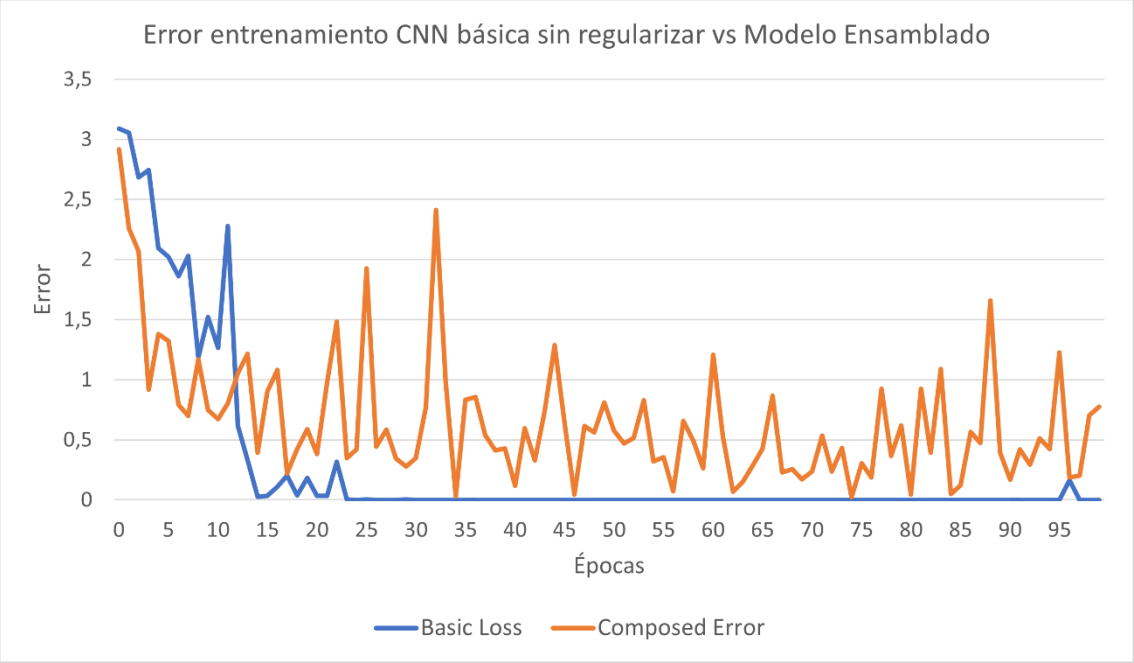

Figura 45. Evolución del error del modelo ensamblado y de la CNN básica sin regularizar

Los resultados de evaluar el modelo ensamblado sobre los datos de test se muestran en la *tabla 15* y, como se podía intuir por los resultados previos, no consigue superar la línea base.

Modelo	FScore en datos de test
CNN básica	0.501
Modelo ensamblado	0.441

Tabla 15. Resultados de evaluar el modelo ensamblado sobre los datos de test

Asimismo, el resultado de evaluación sobre los datos de test se desglosó a nivel de clase y aparece representado en la *figura 46* comparado con los de la CNN básica sin regularizar. Lo que se puede observar es que el modelo ensamblado ha mejorado en marcas como Samsung, Universal, Tic Tac, Pac Man y Monster, pero es notablemente peor en otras como Sony, Pizza Hut, Lipton, McDonalds y Disney.

Por lo tanto, parece que la combinación de modelos no ha sido buena y los resultados que se obtienen son peores que los de la línea base.

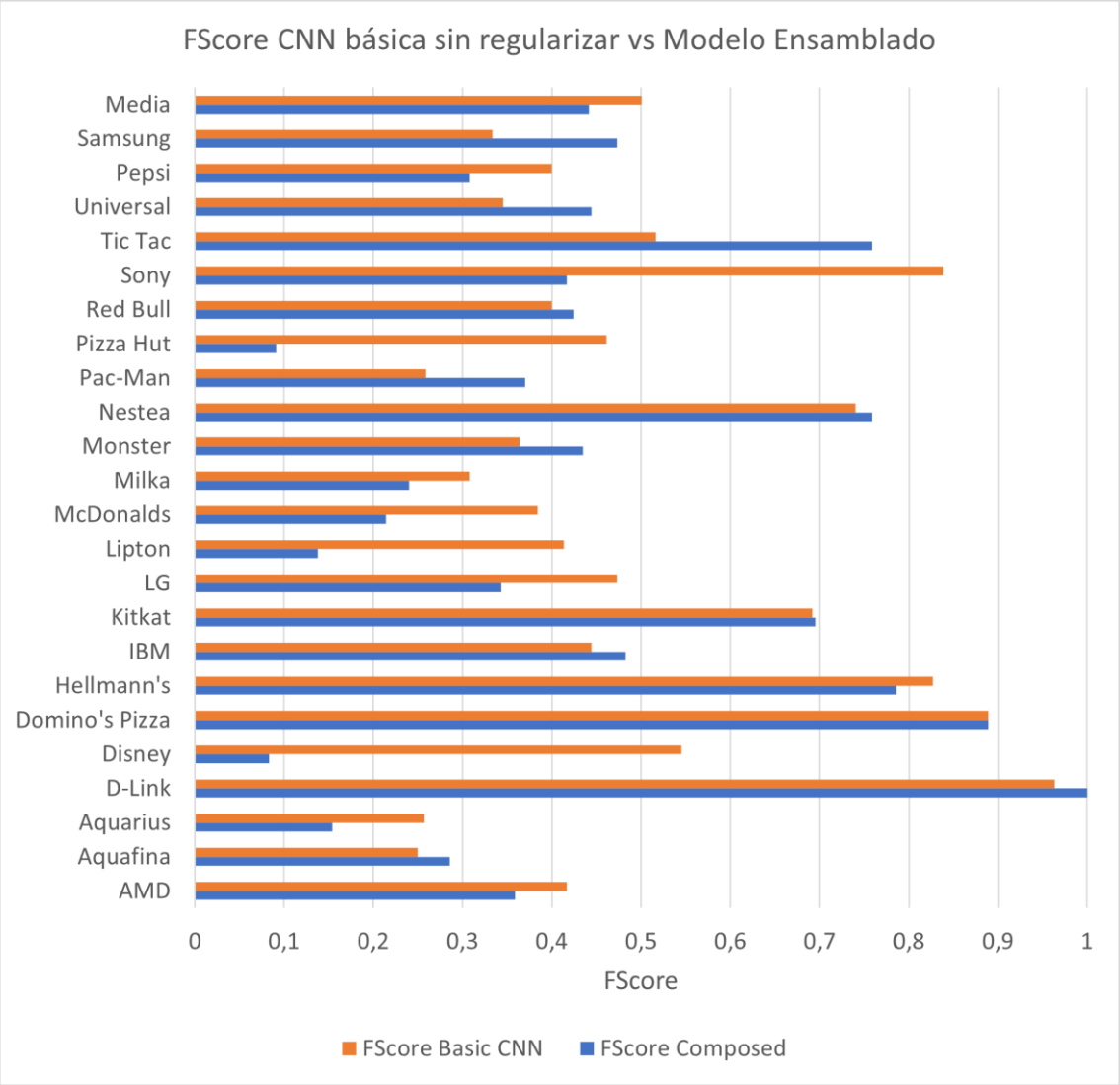

Figura 46. FScore del modelo ensamblado sobre los datos de test a nivel de clase

2.5.1 Conclusiones del diseño del modelo ensamblado

La división de las clases en niveles de dificultad demostró ser una buena alternativa cuando se entrenaron los modelos para cada nivel. Sin embargo, el ensamble no ha obtenido los resultados esperados.

La hipótesis de combinar las fortalezas de la CNN básica con las de la CNN avanzada para suplir las debilidades de uno y de otro ha resultado ser incorrecta. No se ha conseguido mejorar los resultados de la línea base en ningún aspecto. El modelo ensamblado sigue sobre ajustando y es peor que la CNN básica cuando se evalúa sobre los datos de test.

No obstante, obviando los resultados de eficacia, desde el punto de vista técnico ha sido muy enriquecedor entender cómo se pueden entrenar submodelos y combinarlos para trabajar en conjunto. Esto ha permitido entender muy bien cómo funciona el *framework* de *PyTorch* y sirve como experiencia para futuros problemas.

De cara a conseguir una mayor eficacia, parece que no se está consiguiendo avanzar con los modelos diseñados desde cero. Por lo tanto, la vía que se va a plantear es utilizar modelos pre entrenados para adaptarlos al problema de clasificación de logos.

La hipótesis es que los modelos entrenados desde cero, como los que se han hecho hasta ahora, no consiguen extraer las características necesarias porque hay muy pocas imágenes en el conjunto de datos y los *kernels* que se generan en las capas convolucionales no son lo suficientemente buenos.

Sin embargo, es posible que una red pre entrenada con millones de imágenes tenga *kernels* mucho mejores que puedan extraer características más relevantes y así, se puedan conseguir mejores resultados. Por ello, el siguiente paso será utilizar técnicas de *Transfer Learning.*

2.6 Transfer Learning

El *Transfer Learning* en Redes Neuronales de Convolución se apoya en la idea de reutilizar los *kernels* de las capas convolucionales de otras CNNs. Generalmente, las CNNs que se usan para hacer *Transfer Learning* han sido preentrenadas con millones de ejemplos y son capaces de extraer gran cantidad de información de las imágenes.

Llegados a este punto del proyecto, los modelos de *Transfer Learning* suponen una alternativa muy interesante para tratar de superar el resultado que se obtiene con la CNN básica porque, hasta el momento, no se han conseguido grandes avances con los modelos entrenados desde cero y no se dispone de millones de imágenes para entrenar.

Para probar el *Transfer Learning*, se van a utilizar dos arquitecturas de red distintas: una diseñada por Microsoft llamada ResNet y otra diseñada por Google llamada GoogLeNet. Para adaptar estas arquitecturas al problema de reconocimiento de logos, lo único que hay que hacer es adaptar la última capa de la parte *Feed Forward* para que tenga tantas neuronas como clases haya que reconocer.

Una vez hecho esto, hay dos opciones para entrenar los modelos. Por un lado, se pueden congelar todas las capas excepto la de salida para que solo sea necesario optimizar esta última. Cuando se hace de esta forma, se dice que se está utilizando la CNN pre entrenada como un extractor de características. Por otro lado, existe la posibilidad de entrenar todas las capas de la red para conseguir una mayor eficacia, pero esto supone un coste computacional mucho más elevado.

En este trabajo, los modelos se entrenan utilizando la CPU del equipo porque no se dispone del *hardware* necesario para hacer los entrenamientos sobre una tarjeta gráfica. Por lo tanto, la opción que se va a seguir será la de congelar todas las capas y entrenar solo la última.

2.6.1 ResNet

La arquitectura ResNet surge para resolver un problema muy conocido llamado *vanishing gradient*. Se trata de un fenómeno que se produce cuando las arquitecturas de red tienen muchas capas y que provoca que haya parámetros que no se optimicen. Como consecuencia, las arquitecturas se encuentran con una limitación en cuanto a la profundidad del modelo y no pueden escalar a problemas más complejos.

La razón del *vanishing gradient* se debe a que el algoritmo que se usa para actualizar los pesos atribuye una variación alta a los pesos de las últimas capas y conforme retrocede hacia las primeras capas, la variación es menor. Si hay mucha profundidad, la variación se reduce tanto que se convierte a 0 y no se actualizan los pesos.

Para resolver este problema, la arquitectura ResNet utiliza un *bypass* cada cierto número de capas convolucionales. Esto significa que, a la salida de las capas convolucionales, se les añade el valor de entrada como se muestra en la *figura 47*. Esto permite que "refrescar" la información y se evita el *vanishing gradient*.

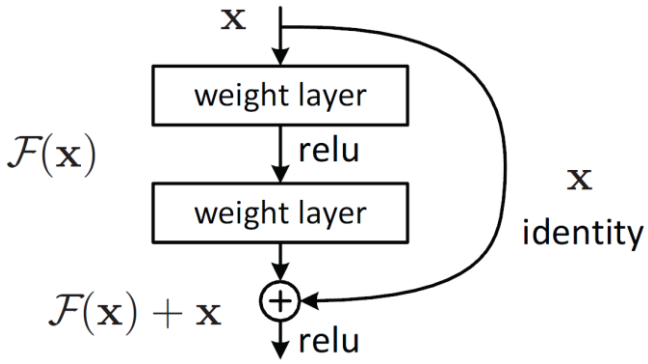

Figura 47. Ejemplo de bypass en la red ResNet. Tomada de Deep Learning: GoogLeNet Explained, R. Alake, 2020, Towards Data Science

Utilizando esta técnica, la arquitectura ResNet puede aumentar la profundidad de sus modelos sin temor a que no se actualicen los pesos. Por esta razón, existen varios diseños de ResNet con diferentes profundidades.

A mayor profundidad, más potente será la red y más parámetros tendrá. Generalmente, a cada diseño de ResNet se le asocia un número y cuanto mayor sea, más profunda será la arquitectura. Para entender la magnitud de estos modelos, en la *tabla 16* se recoge el número de parámetros para diferentes versiones de ResNet.

Modelo	Número de parámetros (M = Millones)
ResNet 18	11.174 M
ResNet 34	21.282 M
ResNet 50	23.521 M
ResNet 101	42.513 M
ResNet 152	58.157 M

Tabla 16. Número de parámetros de los modelos ResNet

2.6.2 GoogLeNet

GoogLeNet es una CNN con 22 capas de profundidad (27 incluyendo las capas de *pooling*) que ha sido diseñada por Google. Se trata de una arquitectura de 6.797 millones de parámetros que se ha utilizado en numerosas áreas del *Deep Learning* tales como: clasificación de imágenes, detección de objetos y reconocimiento de objetos, entre otras.

Dentro de su arquitectura, 9 de las capas de convolución son módulos *Inception*. Estos módulos tienen como objetivo reducir el coste computacional del entrenamiento aplicando convoluciones y *pooling* en paralelo y concatenando los resultados para que el modelo crezca en anchura en vez de en profundidad.

Un ejemplo sencillo de un módulo *Inception* se muestra en la *figura 48*. En este caso, se aplican 3 convoluciones de tamaño 1x1, 3x3 y 5x5, y *max pooling* de 3x3. Después de aplicar estas operaciones, los resultados se concatenan y se pasan a la siguiente capa.

Dada la profundidad de *GoogLeNet*, esta arquitectura también sufre del problema del *vanishing gradient*. Para solucionarlo, se han diseñado módulos *Inception* más avanzados que pretender reducir este fenómeno lo máximo posible.

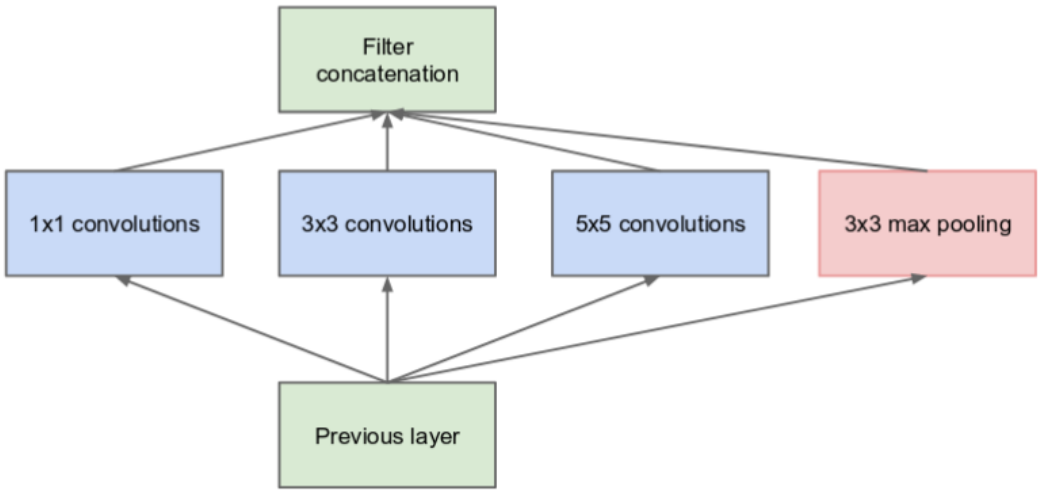

Figura 48. Ejemplo de un módulo Inception sencillo. Tomada de Inception Module, DeepAI

2.6.3 ResNet18 vs GoogLeNet

Para comparar cuál de las dos arquitecturas funcionaba mejor para el problema de reconocimiento de logos, se entrenaron sobre el mismo conjunto de datos y se compararon los *FScore* que obtenían.

La configuración del entrenamiento se muestra en la *tabla 17*. Para optimizar la capa de salida de ambos modelos, se entrenó durante 25 épocas utilizando AdaBound y se aplicaron regularizadores L2.

Épocas	25
Tamaño del *batch*	10
Optimizador	AdaBound
Learning Rate	0.0001
Weight decay	0.001

Tabla 17. Configuración entrenamiento ResNet18 y GoogLeNet

La evolución del *FScore* durante el entrenamiento se muestra en la *figura 49* y lo que se puede observar es que la distancia entre la curva de entrenamiento y validación es, en ambos casos, considerablemente menor a todos los entrenamientos que se han hecho hasta el momento. La diferencia entre ambas curvas es menor a 0.1 por lo que, se puede empezar a intuir que estos modelos están consiguiendo generalizar mucho mejor.

Asimismo, las curvas parecen ser bastante regulares y presentan pocos picos, lo cual da a entender que el entrenamiento es muy estable y robusto.

Comparando ResNet18 con GoogLeNet, los resultados parecen bastante similares, pero ResNet18 queda levemente por encima casi en todo momento. Por lo tanto, aunque hay poca diferencia, ResNet18 está actuando mejor.

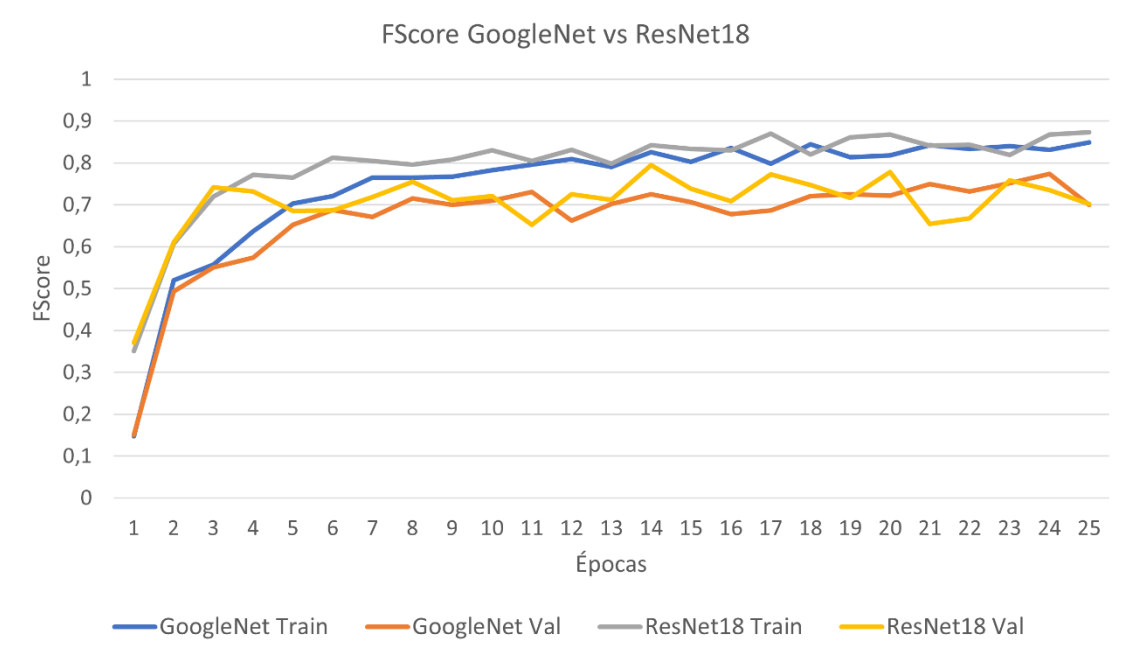

Figura 49. Evolución de FScore de ResNet18 y GoogLeNet

Para terminar de decidir cuál de los dos modelos se usará en las siguientes iteraciones del proyecto, se evaluó el *FScore* sobre los datos de test y los resultados se muestran en la *tabla 18* comparados con los de la CNN básica sin regularizar.

Lo que se observa es que ambos consiguen superar la línea base por más de dos décimas, lo cual demuestra que el *Transfer Learning* ha sido un éxito. Entre ResNet18 y GoogLeNet, el que ha conseguido un resultado mejor es ResNet18, aunque, como ya se podía intuir, la diferencia ha sido mínima.

Para las siguientes iteraciones se considerará ResNet18 no solo porque ha superado levemente a GoogLeNet, sino porque cuenta con versiones más complejas que pueden llegar a conseguir resultados aún mejores.

Modelo	FScore sobre los datos de test
CNN básica	0.501
ResNet 18	0.737
GoogLeNet	0.723

Tabla 18. FScore de ResNet18 y GoogLeNet sobre los datos de test

Para visualizar más claramente la mejora que ha supuesto utilizar *Transfer Learning*, en la *figura 50* se muestra el desglose del *FScore* que obtiene en los datos de test comparados con los de la CNN básica sin regularizar.

Lo que se observa es que ResNet18 ha supuesto una mejora muy considerable en la gran mayoría de las clases, solo en Disney y D-Link la CNN básica es mejor. Algunas clases donde se estaba teniendo gran dificultad, ahora alcanzan o superan el 0.5 de *FScore* y muchas otras, se aproximan al 0.7 y 0.8.

En pocas palabras, ResNet18 ha conseguido superar con creces los resultados de la línea base del proyecto y será el modelo de referencia que se tome a partir de ahora.

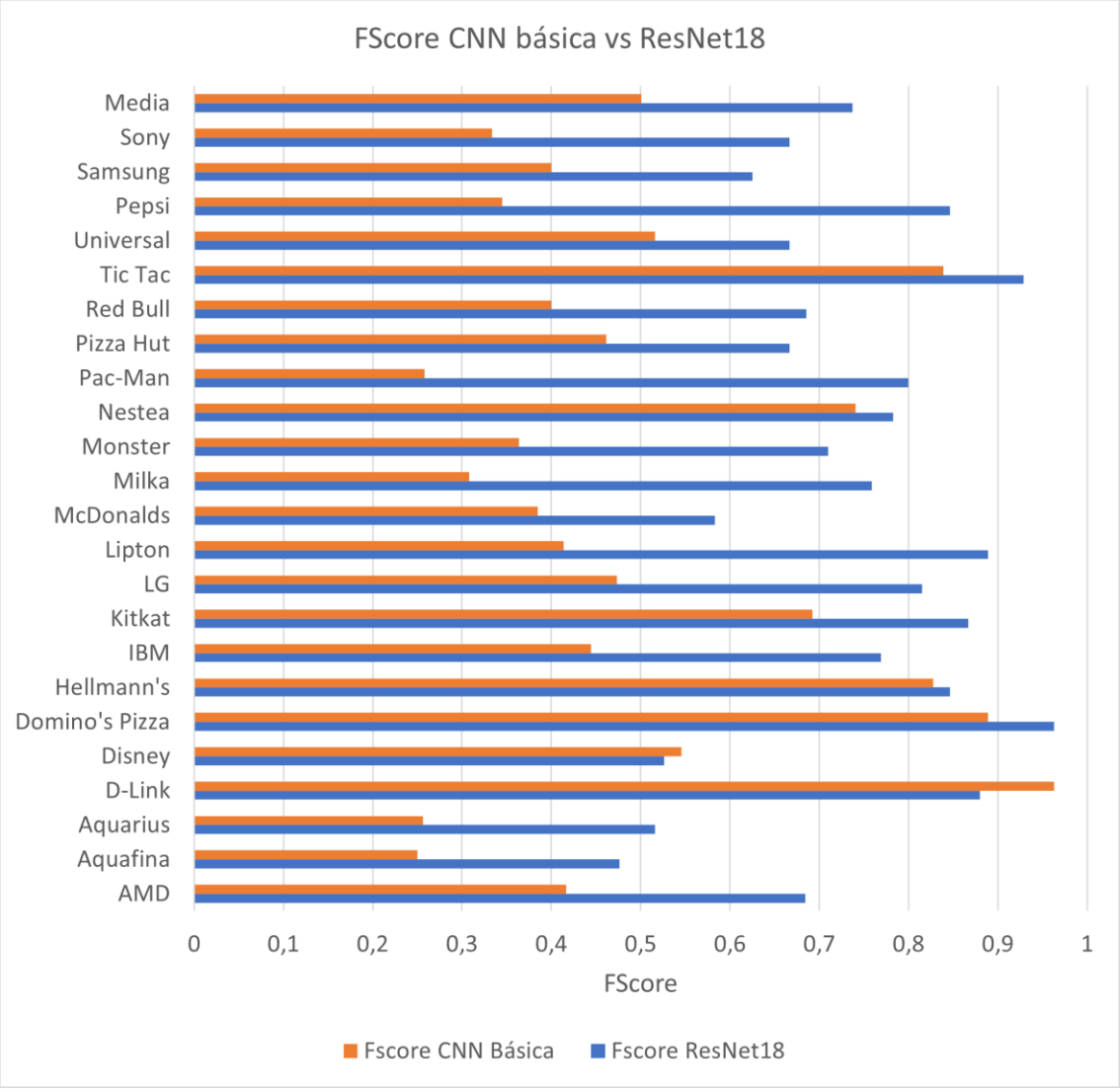

Figura 50. FScore del ResNet18 en los datos de test desglosado por clases

2.6.4 ResNet18 vs ResNet34 vs ResNet50

Después del éxito de ResNet18, cabe preguntarse si aumentando la profundidad del modelo se podrían obtener mejores resultados. ResNet tiene la característica de que puede aumentar su profundidad con facilidad gracias a los *bypass* que utiliza.

Para probar si esto es cierto, se entrenaron 2 modelos de ResNet con más profundidad y se compararon con el ResNet18. Para entrenarlos se usó la misma configuración que tuvo ResNet18 y que se muestra en la *tabla 19*.

Épocas	25
Tamaño del *batch*	10
Optimizador	AdaBound
Learning Rate	0.0001
Weight decay	0.001

Tabla 19. Configuración entrenamiento ResNet34 y ResNet50

Los resultados de FScore se muestran en la *figura 51* comparados con los de ResNet18. Lo que se puede observar es que todas las arquitecturas tienen un resultado muy similar en el conjunto de entrenamiento y de validación.

Esto permite intuir que el aumento de la profundidad del modelo no tiene un efecto positivo. De hecho, trabajar con modelos con más parámetros implica un mayor tiempo de cómputo y mayor riesgo al sobreajuste.

Los resultados de evaluar el *FScore* sobre los datos de test se muestra en la *tabla 20* y lo que se destaca es que todos tienen una eficacia similar. Por lo tanto, atendiendo a la complejidad del modelo, se escogerá el ResNet18 que es el que tiene menor número de parámetros.

Modelo	FScore sobre los datos de test
ResNet 18	0.737
ResNet 34	0.737
ResNet 50	0.716

Tabla 20. FScore de ResNet18, ResNet34 y ResNet50 sobre los datos de test

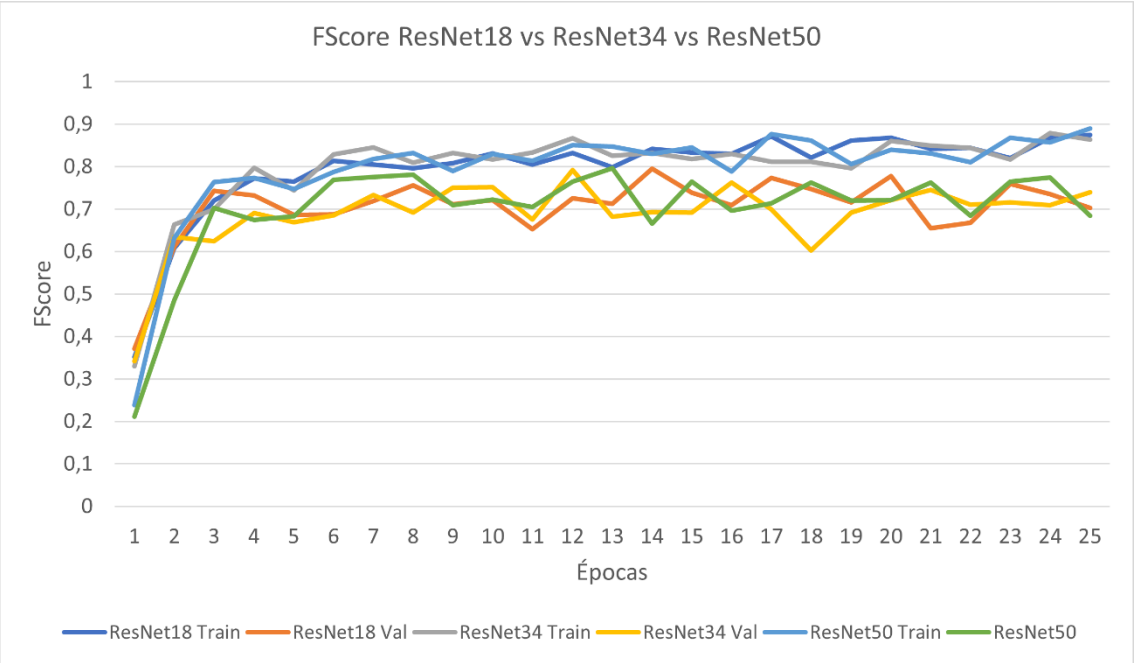

Figura 51. FScore de ResNet18, ResNet34 y ResNet50 sobre los datos de entrenamiento y validación

2.6.5 ResNet18 con *Data Augmentation*

Dado que el aumento de la profundidad del modelo no mejora el *Fscore* que se obtiene sobre los datos de test, se probó una última opción basada de nuevo en técnicas de *Data Augmentation*.

En experimentos anteriores, las transformaciones que se habían utilizado para hacer *Data Augmentation* eran espejos horizontales y verticales. Sin embargo, observando las imágenes del conjunto de datos, no había casos de logos que tuvieran un espejo horizontal o vertical. Por lo tanto, se estaban aplicando transformaciones que incluían casuísticas que en el problema no aparecían.

Para esta ocasión, se diseñó una nueva política de *Data Augmentation* que se adaptara mejor a las situaciones que podían darse en las imágenes. En concreto, se aplicaron rotaciones aleatorias comprendidas entre -45º y 45º y giros aleatorios en perspectiva. Esto permite contemplar las situaciones donde los logos aparecen rotados o sobre la superficie de un producto.

Para visualizar gráficamente cómo son estas transformaciones, en la *figura 52* se puede ver un ejemplo sencillo de rotación y giro.

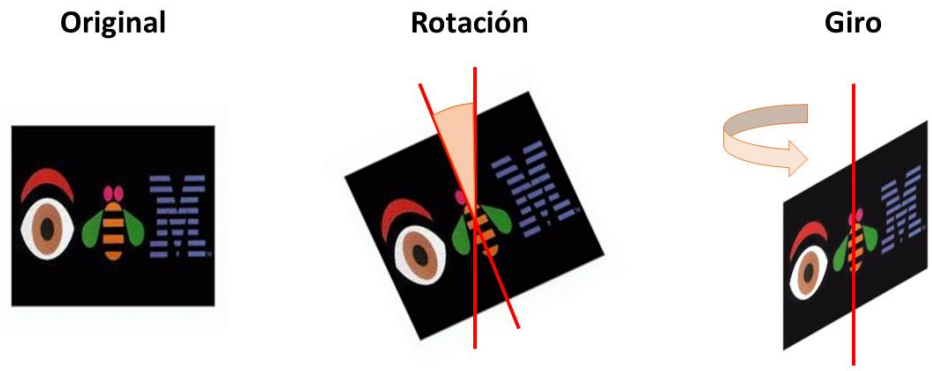

Figura 52. Ejemplos de rotación y giro sobre imágenes

Estas transformaciones se aplican de manera aleatoria y pueden darse de forma simultánea, es decir, una rotación y un giro al mismo tiempo. La probabilidad de transformación se fijó a un 30%, por lo que no todas las imágenes acaban siendo transformadas. Así, la red también entrena con las imágenes sin modificar y el entrenamiento no acaba distorsionado.

A modo de ejemplo, en la *figura 53* se muestra un *batch* de imágenes como las que podría recibir la red durante el entrenamiento. Se puedo observar que hay imágenes sin transformar como la de IBM y Nestea, otras que tienen solo una rotación como Aquarius o solo un giro como KitKat, y otras que tienen las dos transformaciones como PizzaHut y McDonalds.

Nótese que estas imágenes aparecen a color porque ResNet18 está diseñado para recibir imágenes de color.

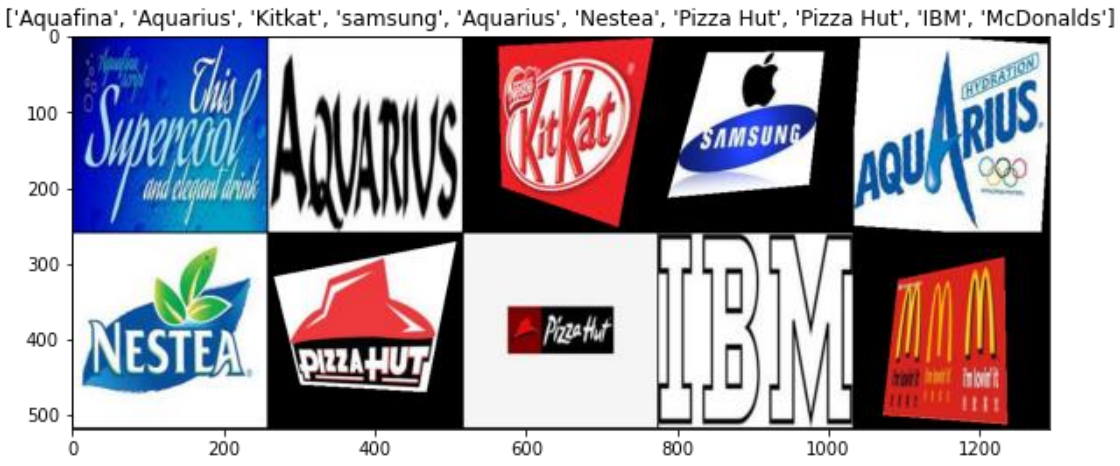

Figura 53. Ejemplo de batch con transformaciones de rotación y giro

Una vez definida la política de *Data Augmentation*, se volvió a entrenar ResNet18 utilizando la configuración de entrenamiento que se muestra en *tabla 21*.

Épocas	25
Tamaño del *batch*	10
Optimizador	AdaBound
Learning Rate	0.0001
Weight decay	0.001

Tabla 21. Configuración de entrenamiento de ResNet18 con Data Augmentation

La comparación entre los *FScores* de entrenamiento de ResNet18 y ResNet18 con *Data Augmentation* se puede ver en la *figura 54*. Los resultados muestran un comportamiento muy similar tanto en el conjunto de entrenamiento como en el conjunto de validación.

Lo más destacable es que la curva de entrenamiento de la versión con *Data Augmentation* es más estable, lo cual puede ser un indicio de mejora. Sin embargo, esto solo se verá en la validación con los datos de test.

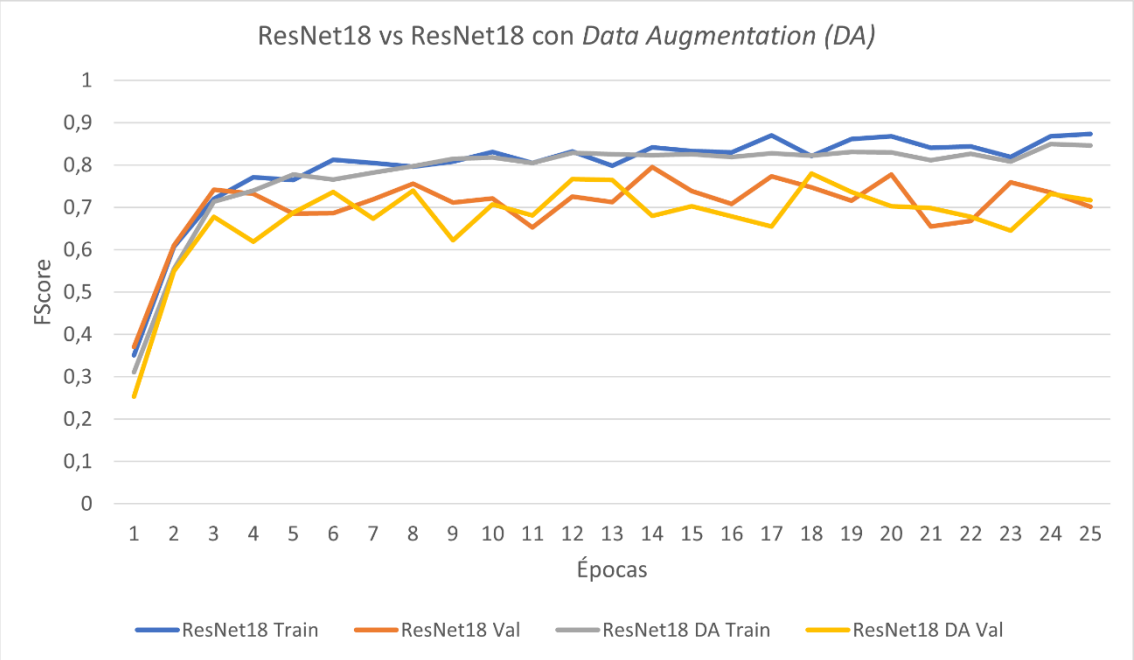

Figura 54. FScore de entrenamiento de ResNet18 vs ResNet18 con Data Augmentation

La comparación de los modelos sobre los datos de test se muestra en la *tabla 22* y el detalle del *FScore* a nivel de clase en la *figura 55.* Lo que se observa es que ResNet18 sin *Data Augmentation* es levemente mejor tanto a nivel global como a nivel de clase. La versión con *Data Augmentation* consigue mejorar en clases donde el modelo ya es bueno y es peor en clases difíciles. En otras palabras, la versión sin *Data Augmentation* es más equilibrada y es buena incluso en clases difíciles. Por ello, se tomará la versión sin *Data Augmentation*.

Modelo	FScore sobre los datos de test
ResNet 18	0.737
ResNet 18 *Data Augmentation*	0.721

Tabla 22. FScore sobre los datos de test para ResNet18 y ResNet18 con Data Augmentation

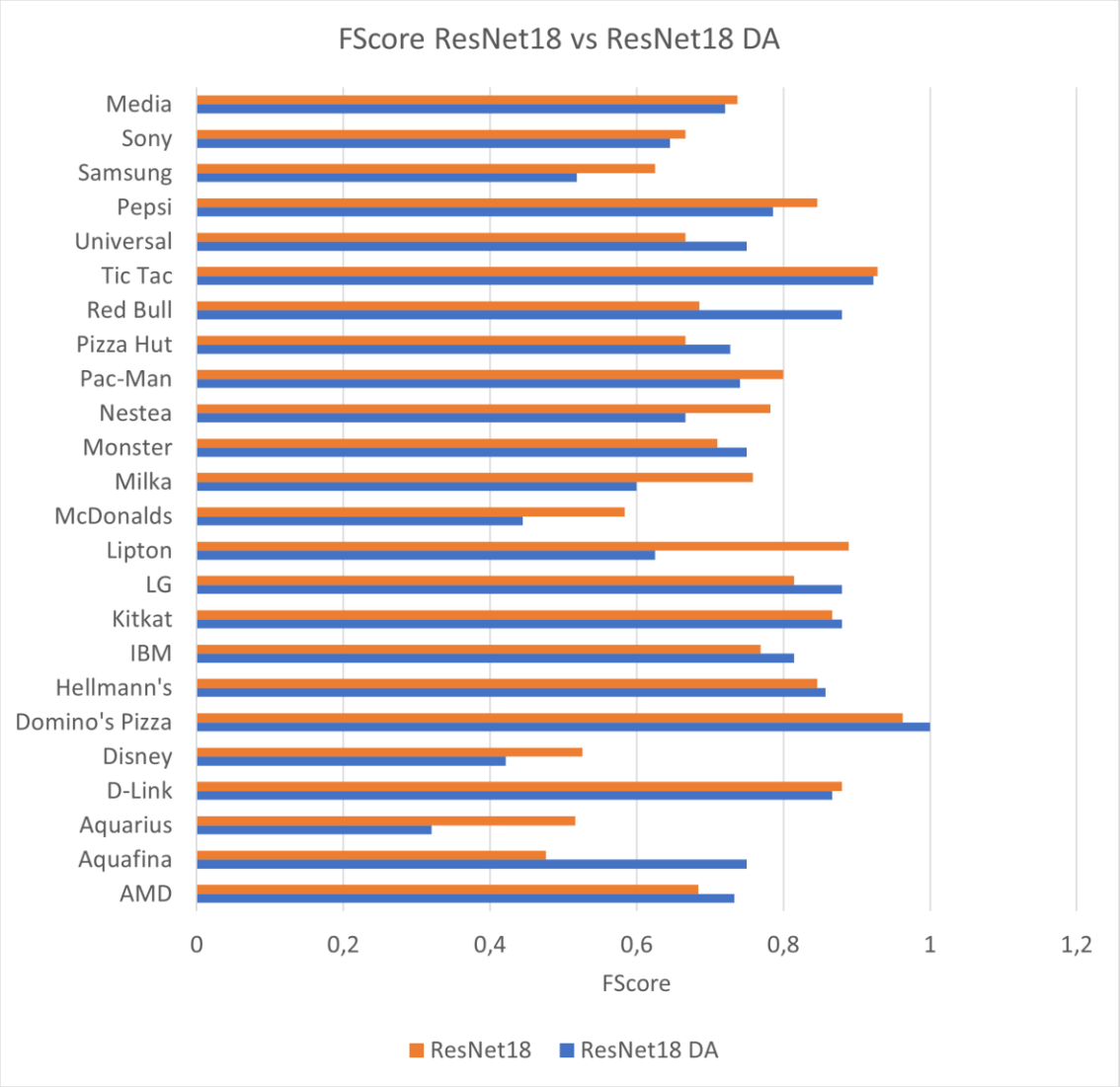

Figura 55. Fscore de ResNet18 y ResNet18 con Data Augmentation a nivel de clase

2.6.6 Conclusiones del *Transfer Learning*

Los modelos que se habían entrenado hasta el momento se habían encontrado con un límite en los 0.5 de *FScore* y no eran capaces de superarlo ni aplicando regularizaciones, ni técnicas de *Data Augmentation* ni dividiendo el problema.

Por esta razón, se optó a probar el *Transfer Learning* y los resultados han sido todo un éxito. Inicialmente se probó con dos arquitecturas, una diseñada por Microsoft llamada ResNet18 y otra de Google llamada GoogLeNet, y ambas superaron los 0.7 de *FScore*.

Se decidió continuar con ResNet18 porque mostró resultados levemente mejores a los de GoogLeNet y, además, gracias a la técnica de *bypass* que utiliza para evitar el *vanishing gradient*, ResNet puede aumentar la profundidad de su arquitectura y crear modelos más complejos.

A continuación, se probó a entrenar dos arquitecturas ResNet más complejas llamadas ResNet34 y ResNet50, pero los resultados fueron similares a la ResNet18. Por lo tanto, se decidió continuar con la ResNet18 porque tenía menos parámetros.

Por último, se intentó mejorar el *FScore* de la ResNet18 utilizando técnicas de *Data Augmentation*. Sin embargo, se observó que el modelo resultante era mejor en clases donde ya era bueno el modelo y peor en clases difíciles, es decir, la versión sin *Data Augmentation* era más equilibrada. Por lo tanto, se determinó continuar con la versión sin *Data Augmentation*.

Llegados a este punto del proyecto, la versión de ResNet18 sin *Data Augmentation* que tiene un *FScore* 0.737 será la que se considere como definitiva.

2.7 Despliegue con Streamlit

El proyecto no podría quedar acabado sin llevar el modelo al último estadio de su desarrollo: el despliegue. En numerosas ocasiones, los proyectos de Machine Learning se quedan en conseguir una buena métrica de precisión y muy pocos son los que tratan de darle un enfoque comercial para llevarlo a un producto que pueda ser consumido por un usuario.

La figura del Data Scientist ya combina áreas muy variadas como son estadística, bases de datos y modelado predictivo, entre otras. Sin embargo, suele carecer del conocimiento necesario para el desarrollo de aplicaciones software capaces de desplegar los modelos predictivos que construyen y, de hecho, no debería ser necesario que tenga esas habilidades.

Para dar solución a este problema surge *Streamlit*, una librería creada para *Python* capaz de construir una aplicación web con poquísimas líneas de código y con una dificultad técnica ínfima. La curva de aprendizaje es muy baja, tiene numerosas opciones de personalización y la estética de las aplicaciones es muy profesional. Además, existe una amplia comunidad de usuarios y desarrolladores trabajando con *Streamlit* y hay numerosos tutoriales para aprender a utilizarlo.

Para este trabajo, se diseñó una aplicación sencilla donde el usuario podía arrastrar una imagen de un logo comercial y la aplicación le indicaría la marca predicha por el modelo, junto con un ranking de todas las demás posibilidades. En la *figura 56* se puede ver cómo es la interfaz de usuario. Basta con arrastrar una imagen al cuadro *Drag and Drop* para que se ejecute el proceso.

Una vez se arrastre la imagen, la aplicación mostrará una previsualización de ella y la predicción del modelo tal y como se muestra en la *figura 57*.

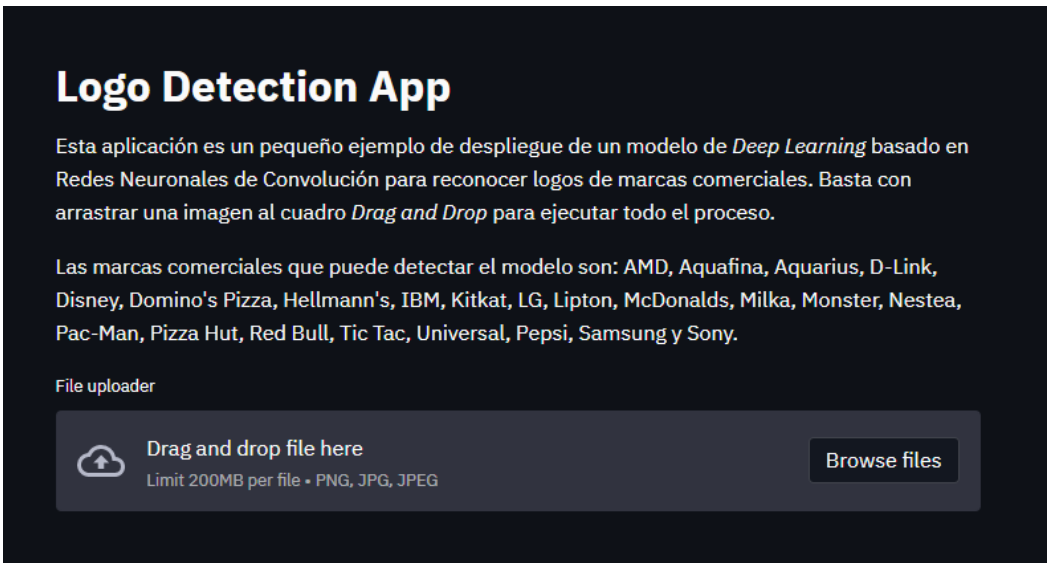

Figura 56. Interfaz de usuario de aplicación en Streamlit

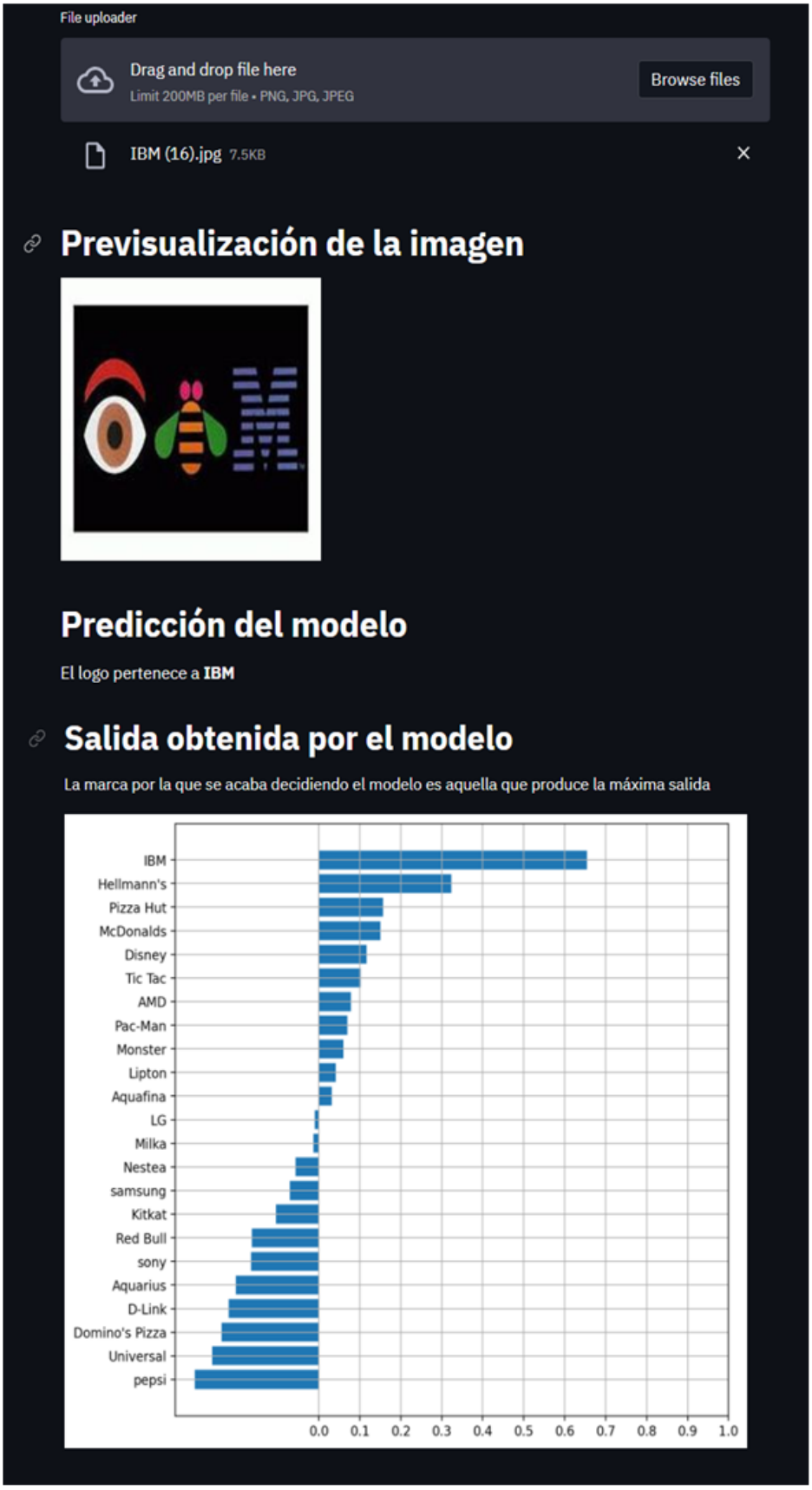

Figura 57. Ejecución de aplicación de Streamlit

2.8 Resumen de experimentos

Para resumir todos los experimentos que se han hecho a lo largo del proyecto, se ha diseñado un diagrama del flujo de trabajo completo que se puede ver en la *figura 58.* El cuadro gris indica el inicio del proyecto, los cuadros naranjas indican experimentos que no superaron la línea base y los cuadros verdes son experimentos exitosos.

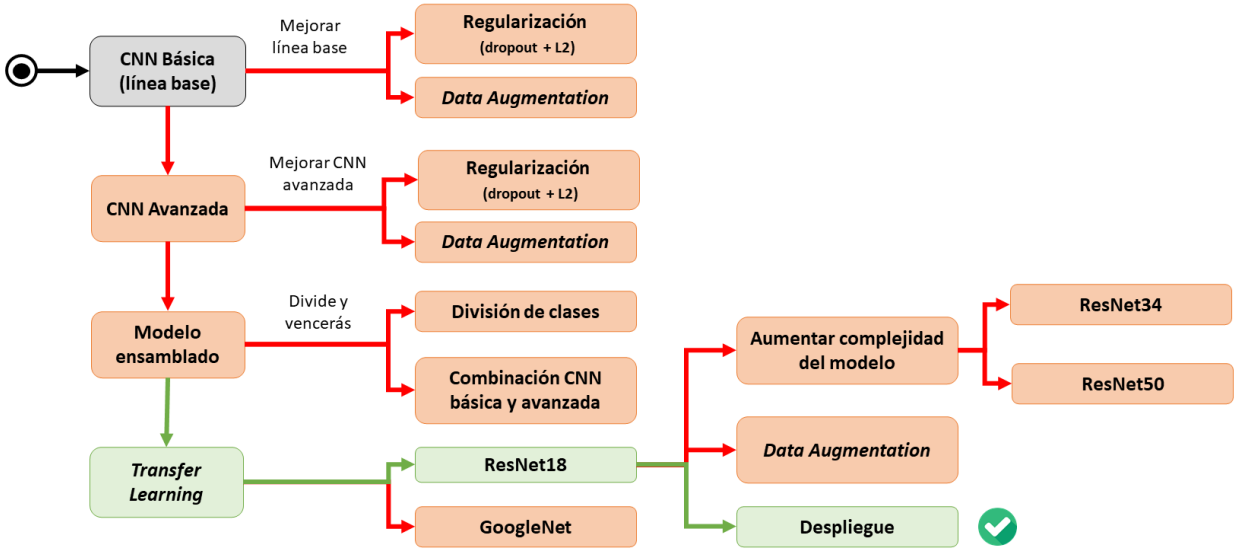

Figura 58. Resumen de experimentos del proyecto

Capítulo 3. CONCLUSIONES Y FUTURAS LÍNEAS DE TRABAJO

El objetivo del proyecto era crear un modelo predictivo basado en Redes Neuronales de Convolución (CNN) capaz de reconocer los logos comerciales de 23 marcas distintas. Después de realizar un análisis exploratorio de los datos, se determinó que las imágenes contenían los logos recortados y que solo sería necesario reconocerlos, no localizarlos dentro de la imagen.

Sin embargo, existían algunos casos donde las imágenes eran de mala calidad, contenían ruido o mezclaban logos de otras marcas que también podían ser reconocidas por el clasificador. Por lo tanto, se intuyó que los datos de entrenamiento iban a inducir cierto error en los resultados. Sin embargo, a pesar de ello, se tomó la decisión de continuar con el conjunto de datos completo.

El primer acercamiento a la resolución del problema se hizo a través de una arquitectura de CNN sencilla que se nombró CNN básica. Este diseño conseguía obtener un *FScore* de 0.501 sobre los datos de test, pero sufría de un sobreajuste muy alto. Este resultado se tomó como línea base para el resto de los modelos que se plantearon a lo largo del proyecto. El objetivo era superar la marca de 0.501 de *FScore* y conseguir que no hubiera sobreajuste.

Para intentar mejorar el modelo básico, se utilizaron técnicas de regularización basadas en capas de *Dropout* y regularizadores L2. Sin embargo, los resultados fueron equivalentes o peores los de la línea base. Asimismo, se intentaron utilizar técnicas de *Data Augmentation* para enriquecer los datos de entrenamiento, pero el resultado tampoco tuvo éxito.

La hipótesis que se planteó fue que el modelo base era demasiado sencillo y no era capaz de extraer la información necesaria para clasificar las imágenes. Por ello, se planteó un segundo modelo que incluía el doble de capas convolucionales y aumentaba la resolución de las imágenes de entrada para poder capturar los detalles pequeños. A este nuevo modelo se lo llamó modelo avanzado.

Sobre el modelo avanzado se aplicaron los mismos experimentos que sobre el modelo base y ninguno de los resultados consiguió superar los 0.501 de *FScore*. Las capas de *Dropout*, los regularizadores L2 y el *Data Augmentation* no estaban dando buenos resultados y el modelo seguía sobre ajustando mucho.

Durante los experimentos, se observó que el modelo básico era bueno prediciendo clases donde el modelo avanzado tenía dificultades y viceversa. Por lo cual, se planteó la hipótesis de que podrían combinarse ambos para unir sus fortalezas y obtener un mejor resultado. Asimismo, en los análisis se detectó que había marcas que siempre se predecían con facilidad y otras que eran mucho más complejas. Por esta razón, se decidió tomar en enfoque *Divide y Vencerás* y se fragmentaron las clases del problema en 4 niveles según la dificultad.

Para cada nivel de dificultad se entrenó un modelo de CNN básica y de CNN avanzada para determinar cuál de las dos arquitecturas funcionaba mejor. Después de verificar el *FScore* en cada nivel, se observó que los resultados estaban siendo superiores, pero era necesario ensamblar los submodelos para poder evaluar el *FScore* de todo el conjunto.

El modelo ensamblado se formó con 2 CNNs básicas y 2 CNNs avanzadas y el ensamble se hizo a través de una red *Feed Forward* sencilla. Desde el punto de vista técnico, este experimento supuso un ejercicio de mucho valor para entender cómo funcionaba el *framework* de *PyTorch*. Sin embargo, los resultados tampoco consiguieron superar a los de la línea base.

Llegados a este punto, se tomó la decisión de dejar de usar modelos diseñados y entrenados desde cero para dar paso al *Transfer Learning*. La ventaja de utilizar *Transfer Learning* es que los modelos han sido pre entrenados con millones de imágenes y son capaces de extraer gran cantidad de información de las imágenes.

Para probar esta técnica, se utilizaron dos arquitecturas, una de ellas desarrollada por Microsoft llamada ResNet18 y otra desarrollada por Google llamada GoogLeNet. Para entrenarlas, se congelaron todas las capas del modelo excepto la última, que fue la que se entrenó para resolver el problema de reconocimiento de logos.

Ambos modelos consiguieron un *FScore* superior al 0.7, por lo cual, el *Transfer Learning* había sido todo un éxito. Para continuar mejorando este resultado, se decidió tomar ResNet18 como nueva línea base y se probó a entrenar dos modelos de tipo ResNet más profundos: ResNet34 y ResNet50.

Sin embargo, el aumento de profundidad no surtió efecto y los resultados que se obtuvieron fueron los mismos que ResNet18, así que, como ResNet18 tenía menos parámetros, se decidió continuar con él.

El último intento para mejorar el modelo fue incluir *Data Augmentation* en el entrenamiento y con ello, se consiguió que la curva de entrenamiento fuera más estable. No obstante, cuando es analizó el *FScore* que obtenía en los datos de test, se descubrió que la versión con *Data Augmentation* había mejorado en clases donde el modelo ya era bueno y había empeorado en clases difíciles de clasificar. Por lo tanto, se descartó la versión con *Data Augmentation* porque era mucho menos equilibrado y se tomó como definitivo el ResNet18 sin *Data Augmentation*.

Finalmente, para concluir con el desarrollo técnico del proyecto, se desplegó el modelo final en una aplicación de Streamlit que permitía introducir una imagen y, automáticamente, se lanzaba el modelo, se predecía y se mostraban los resultados de la predicción.

En definitiva, después de muchos intentos y de plantear el problema de varias formas, se ha conseguido tener un modelo con un *FScore* de 0.737 y se ha desplegado sobre una aplicación que puede ser utilizada por un usuario de forma fácil e intuitiva.

De cara al futuro, este proyecto puede llegar a extenderse y mejorarse por muchas vías. En primer lugar, sería de gran interés diseñar algún método que permita distinguir las imágenes donde el logo sea irreconocible para que no formen parte del entrenamiento y de la evaluación. El modelo podría responder a estas imágenes ruidosas indicando que no tiene información suficiente para reconocer el logo.

Por otra parte, se podría aumentar la cantidad de imágenes disponibles para cada marca. Actualmente solo se dispone de 70 imágenes por clase y algunas de ellas son ruidosas y están distorsionadas. Asimismo, se podría aumentar el número de marcas que reconoce el clasificador.

Por otro lado, también hay ocasiones donde el logo no está centrado en la imagen o es demasiado pequeño. En este caso se podría utilizar algún algoritmo de detección de objetos para localizarlo, recortarlo y clasificarlo. En esta misma línea, existen algunas imágenes que contienen varios logos de distintas marcas, así que, se podría usar el algoritmo de reconocimiento de objetos para detectar cuántos logos hay en la imagen y evaluar el clasificador en cada uno de ellos.

En resumidas cuentas, la detección y reconocimiento de logos comerciales es un área donde aún queda mucho desarrollo y que es de gran utilidad a nivel comercial. Este trabajo sirve como un ejemplo de la metodología y lo modelos que se pueden usar y de cómo se pueden desplegar en una aplicación que pueda consumir un usuario.

REFERENCIAS

1. APWG | Phishing Activity Trends Reports. (s. f.). Anti-Phishing Working Group. Recuperado 23 de agosto de 2021, de https://apwg.org/trendsreports/

2. Bazaarvoice: Meet shoppers in all the moments that matter. (2021, 6 agosto). Bazaarvoice. https://www.bazaarvoice.com/?redirect=curalatecom

3. Geng, G., Lee, X., & Zhang, Y. (2014, agosto). *Combating phishing attacks via brand identity and authorization features*. Security Comm. https://onlinelibrary.wiley.com/doi/epdf/10.1002/sec.1045

4. L. (s. f.). *GitHub - Luolc/AdaBound: An optimizer that trains as fast as Adam and as good as SGD.* GitHub. Recuperado 24 de septiembre de 2021, de https://github.com/Luolc/AdaBound

5. Leng Chiew, K., Hung Chang, E., Nah Sze, S., & King Tiong, W. (2015, octubre). *Utilisation of website logo for phishing detection*. Computers & Security. https://doi.org/10.1016/j.cose.2015.07.006

6. Liangchen, L., Yuanhao, X., Yan, L., & Xu, S. (2019). *Adaptive Gradient Methods With Dynamic Bound Of Learning Rate*. ICLR. https://openreview.net/pdf?id=Bkg3g2R9FX

7. Olapic. (s. f.). *User Generated Content Platform (UGC), Content Marketing | Olapic*. Olapic | User-Generated Content | Influencer Marketing | Short-Form Video. Recuperado 21 de agosto de 2021, de https://www.olapic.com/

8. Oliveira, G., Frazão, X., Pimentel, A., & Ribeiro, B. (2020). *Automatic graphic logo detection via Fast Region-based Convolutional Networks*. International Joint Conference on Neural Networks (IJCNN). https://ieeexplore.ieee.org/abstract/document/7727305

9. Orti, O., Tous, R., Poveda, J., Cruz, L., Wust, O., & Gómez, M. (2019, mayo). *Real-Time Logo Detection in Brand-Related Social Media Images*. https://doi.org/10.1007/978-3-030-20518-8_11

10. Selman Bozkir, A., & Aydos, M. (2020, agosto). *LogoSENSE: A companion HOG based logo detection scheme for phishing web page and E-mail brand recognition, Computers & Security*. https://doi.org/10.1016/j.cose.2020.101855

11. Tous, R., Gómez, M., Poveda, J., Cruz, L., Wüst, O., Makni, M., & Ayguadé, E. (2018, abril). *Automated curation of brand-related social media images with deep learning*. Multimedia Tools Appl. https://doi.org/10.1007/s11042-018-5910-z

12. van Veen, F., van Veen, F., van Veen, F., & van Veen, F. (2017, 31 marzo). *Fjodor van Veen, Author at*. The Asimov Institute. https://www.asimovinstitute.org/author/fjodorvanveen/

BIBLIOGRAFÍA

Alake, R. (2020, 23 diciembre). *Deep Learning: GoogLeNet Explained - Towards Data Science*. Medium. https://towardsdatascience.com/deep-learning-googlenet-explained-de8861c82765

Alsharnouby, M., Furkan, A., & Chiasson, S. (2015, octubre). *Why phishing still works: User strategies for combating phishing attacks*. International Journal of Human-Computer Studies. https://doi.org/10.1016/j.ijhcs.2015.05.005

Bianco, S., Buzzelli, M., Mazzini, D., & Schettini, R. (2017, julio). *Deep learning for logo recognition*. Neurocomputing. https://doi.org/10.1016/j.neucom.2017.03.051

Bushaev, V. (2018, 24 octubre). *Adam — latest trends in deep learning optimization.* Medium. https://towardsdatascience.com/adam-latest-trends-in-deep-learning-optimization-6be9a291375c

DeepAI. (2020, 25 junio). *Inception Module*. https://deepai.org/machine-learning-glossary-and-terms/inception-module

Dhamija, R., Tygar, J. D., & Hearst, M. (2006, abril). *Why phishing works. Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. Association for Computing Machinery. https://doi.org/10.1145/1124772.1124861

Gandhi, R. (2018, 3 diciembre). *R-CNN, Fast R-CNN, Faster R-CNN, YOLO — Object Detection Algorithms*. Medium. https://towardsdatascience.com/r-cnn-fast-r-cnn-faster-r-cnn-yolo-object-detection-algorithms-36d53571365e

Hassan, M. U. (2019, 25 enero). *ResNet (34, 50, 101): Residual CNNs for Image Classification Tasks*. Neurohive. https://neurohive.io/en/popular-networks/resnet/

Hoppe, T. (s. f.). *Streamlit • The fastest way to build and share data apps*. Streamlit. Recuperado 24 de septiembre de 2021, de https://streamlit.io/

Illustration of transforms — Torchvision 0.10.0 documentation. (s. f.). PyTorch Official Documentation. Recuperado 24 de septiembre de 2021, de https://pytorch.org/vision/stable/auto_examples/plot_transforms.html#sphx-glr-auto-examples-plot-transforms-py

Indola, F. N., Shen, A., Gao, P., & Keutzer, K. (2015, octubre). *DeepLogo: hitting Logo Recognition with the Deep Neural Network Hammer*. https://arxiv.org/abs/1510.02131v1

Juschka, A. (2019, 18 agosto). *Classifying Logos in Images with Convolutionary Neural Networks*. Medium. https://medium.com/twodigits/classifying-logos-in-images-with-convolutionary-neural-networks-cnns-in-keras-21f02fcea5c2

Ruiz, P. (2019, 23 abril). *Understanding and visualizing ResNets - Towards Data Science*. Medium. https://towardsdatascience.com/understanding-and-visualizing-resnets-442284831be8

S. (2019, 7 marzo). *ICLR 2019 | 'Fast as Adam & Good as SGD' — New Optimizer Has Both*. Medium. https://medium.com/syncedreview/iclr-2019-fast-as-adam-good-as-sgd-new-optimizer-has-both-78e37e8f9a34

Saha, S. (2021, 17 agosto). *A Comprehensive Guide to Convolutional Neural Networks — the ELI5 way*. Medium. https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53

Sahel, S., Alsahafi, M., Alghamdi, M., & Alsubait, T. (2021, febrero). *Logo Detection Using Deep Learning with Pretrained CNN Models*. https://doi.org/10.48084/etasr.3919

Sahir, S. (2019, 27 enero). *Canny Edge Detection Step by Step in Python — Computer Vision*. Medium. https://towardsdatascience.com/canny-edge-detection-step-by-step-in-python-computer-vision-b49c3a2d8123

Shukla, L. (2020, 26 junio). *Dropout in PyTorch – An Example*. W&B. https://wandb.ai/authors/ayusht/reports/Dropout-in-PyTorch-An-Example--VmlldzoxNTgwOTE

Taunk, D. (2020, 16 abril). *L1 vs L2 Regularization: The intuitive difference - Analytics Vidhya*. Medium. https://medium.com/analytics-vidhya/l1-vs-l2-regularization-which-is-better-d01068e6658c

Tch, A. (2021, 16 febrero). *The mostly complete chart of Neural Networks, explained*. Medium. https://towardsdatascience.com/the-mostly-complete-chart-of-neural-networks-explained-3fb6f2367464

torchvision.datasets — Torchvision 0.10.0 documentation. (s. f.). PyTorch Official Documentation. Recuperado 24 de septiembre de 2021, de https://pytorch.org/vision/stable/datasets.html

Vickery, R. (2020, 27 mayo). *10 Lesser-Known Python Libraries for Machine Learning*. Medium. https://towardsdatascience.com/10-lesser-known-python-libraries-for-machine-learning-fca7ad32e53c