Trabajo Fin de Grado
Grado en Ingeniería de las Tecnologías de Telecomunicación

Reconocimiento de Imágenes de Lugares Emblemáticos Utilizando Aprendizaje Máquina

Autor: Javier Serrano Jodral

Tutor: Francisco José Simois Tirado

Dpto. Teoría de la Señal y Comunicaciones
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2022

Trabajo Fin de Grado
Grado en Ingeniería de las Tecnologías de Telecomunicación

Reconocimiento de Imágenes de Lugares Emblemáticos Utilizando Aprendizaje Máquina

Autor:
Javier Serrano Jodral

Tutor:
Francisco José Simois Tirado
Profesor Contratado Doctor

Dpto. de Teoría de la Señal y Comunicaciones
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2022

Trabajo Fin de Grado: Reconocimiento de Imágenes de Lugares Emblemáticos Utilizando Aprendizaje Máquina

Autor: Javier Serrano Jodral

Tutor: Francisco José Simois Tirado

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

El Secretario del Tribunal

Sevilla, 2022

A mi familia

A mis amigos

A mis maestros

Resumen

En este proyecto, en primer lugar se hace un recorrido teórico por la mayoría de las cuestiones de interés de la Inteligencia Artificial, un campo que hoy en día está en boca de todos, repasando todas sus subramas. Se hace énfasis en el Aprendizaje Automático, del que se analizan todas las fases de su proceso operacional, y en la Visión por Computador.

Posteriormente, se desarrollan varios modelos para una aplicación real derivada de la teoría: el reconocimiento de imágenes de lugares emblemáticos a partir de técnicas de la disciplina del Aprendizaje Automático. Concretamente, la implementación se basa en Redes Neuronales Convolucionales construidas con lenguaje de programación Python y las bibliotecas Tensorflow y Keras, entre otras. Se explica cada línea de los bloques de código que constituyen a estos modelos para después comparar sus resultados en distintas ejecuciones, con el objetivo de buscar el óptimo.

Abstract

In this project, a theoretical tour is made in the first place through most points of interest of Artificial Intelligence, a field that nowadays is part of everyday talk, reviewing all its subbranches. There is an emphasis in Machine Learning, whose operational process phases are analyzed, and in Computer Vision.

Subsequently, several models are developed for an actual application derived from theory: image landmark recognition by means of Machine Learning discipline. Specifically, the implementation is based in Convolutional Neural Networks built with the programming language Python and the libraries Tensorflow and Keras, among others. Each one of the lines of the code blocks that make up the models are explained and then the results of different executions are compared, in order to find the optimal one.

Índice

ÍNDICE DE TABLAS

ÍNDICE DE FIGURAS

ÍNDICE DE CÓDIGOS

Abreviaturas

IA	Inteligencia Artificial
ANI	Inteligencia Artificial Reducida
AGI	Inteligencia Artificial General
ASI	Superinteligencia Artificial
NLP	Procesamiento del Lenguaje Natural
ML	Aprendizaje Automático
SQL	Lenguaje de Consulta Estructurado
LDA	Análisis Discriminante Lineal
RGB	Rojo, Verde y Azul
RMS	Media Cuadrática
NBC	Clasificador Bayesiano Ingenuo
SVM	Máquina de Vectores de Soporte
DAGSVM	Grafo Acíclico Dirigido Máquina de Vectores de Soporte
TP	Verdadero Positivo
FP	Falso Positivo
TN	Verdadero Negativo
FN	Falso Negativo
ROC	Característica de Funcionamiento del Receptor
AUC	Área Bajo la Curva ROC
RNA	Red Neuronal Artificial
GD	Descenso del Gradiente
SGD	Descenso del Gradiente Estocástico
MLP	Perceptrón Multicapa
RNP	Red Neuronal Profunda
CNN	Red Neuronal Convolucional
VA	Visión Artificial
RAM	Memoria de Acceso Aleatorio
CPU	Unidad Central de Procesamiento
GPU	Unidad de Procesamiento Gráfico
TPU	Unidad de Procesamiento Tensorial

Símbolos

x	Escalar
$\boldsymbol{x}$	Vector
$\boldsymbol{x_i}$	Elemento *i* del vector
$\boldsymbol{x_{:j}}$	Vector columna *j* a partir de un conjunto de vectores fila
D	Set de datos
$\mathcal{X}$	Espacio
$(\cdot,\cdot,\cdot)$	Vector fila
$(\cdot;\cdot;\cdot)$	Vector columna
$\{\cdots\}$	Set
$[\cdot,\cdot]$	Intervalo, rango
ϵ	Perteneciente a
$\forall$	Para todo
$\min(\cdot)$	Valor mínimo de
$\max(\cdot)$	Valor máximo de
$\lvert\cdot\rvert$	Valor absoluto
$\log(\cdot)$	Logaritmo en base 10
$P(\cdot)$	Probabilidad de
$P(A \mid B)$	Probabilidad de A habiendo ocurrido B
$P(A, B)$	Probabilidad de A y B simultáneamente
$\hat{y}$	Valor estimado, predicho
$\frac{\partial \cdot}{\partial \cdot}$	Derivada parcial

1 INTRODUCCIÓN

1.1 Motivación del Proyecto

En la actualidad, pocos son los sectores que no recurren a alguna aplicación del campo de la Inteligencia Artificial. Dados los grandes avances tecnológicos y las masivas cantidades de datos que circulan por la red, su incorporación a empresas de diversa índole resulta extremadamente asequible en comparación con los positivos beneficios que conlleva. Los procesos se automatizan a la vez que se aumenta la productividad, sin suponer una inversión excesiva. Prácticamente son todo ventajas y beneficios. Es por ello que cada vez son más y más los lugares hasta los que estas aplicaciones llegan, desembocando en un uso cotidiano y normalizado en la sociedad.

La Inteligencia Artificial es protagonista de nuestro día a día. Desde los smartphones, asistentes de voz o textos predictivos hasta vehículos automatizados y mapas en tiempo real. A medida que más utilidades surgen, mayor reconocimiento por parte de la población adquieren, implicando nuevas inversiones en el campo y un mayor progreso, de nuevo generando interés... Sin embargo, una ínfima parte de sus usuarios conoce su funcionamiento real, el trabajo que tienen detrás o sus fundamentos.

Para arrojar luz en el tema y mostrar una aplicación de primera mano, se ha decidido elaborar este trabajo.

1.2 Objetivos del Proyecto

Con el presente trabajo se pretende adquirir conocimientos básicos del campo del Aprendizaje Automático para posteriormente volcarlos en resolver un problema real, la clasificación de lugares emblemáticos, por medio de lenguaje de programación Python.

1.3 Organización de la Memoria

Tras esta introducción, la memoria se divide en dos grandes partes, antes de llegar a una sección final en la que se redactan las conclusiones. La primera está dedicada a la explicación de los conceptos teóricos que se creen esenciales para la comprensión y resolución del problema. La segunda es la propia resolución del problema en forma de código empleando como guía dichos conceptos teóricos, es decir, la puesta en práctica de los conocimientos adquiridos.

Para la parte teórica, se pretende seguir una estructura que comience desde una perspectiva global del tema, una vista de conjunto, y finalizar con una explicación con cierto nivel de detalle de algunos de sus términos. Por ello se parte de la **Inteligencia Artificial**, de la que se van mostrando todas sus posibles clasificaciones, categorías, subramas... para ayudar a dar contexto y a situarnos dentro del bloque científico tecnológico. Se estudian con mayor profundidad dos de sus subramas: el **Aprendizaje Automático** y la **Visión Artificial**, que podría decirse que son las protagonistas del trabajo, pues el problema a tratar se resuelve gracias a sus técnicas y cooperación. Y para concluir esta parte, se comentan los fundamentos de las **Redes Neuronales Artificiales**, un conjunto de técnicas pertenecientes al Aprendizaje Máquina y con las que se ha optado para resolver el problema.

Para la parte práctica, se exponen las **Soluciones Propuestas al Problema** planteado en un inicio. Se describen todos los pasos para su creación y se adjuntan las líneas de código correspondientes, para más tarde comparar los resultados obtenidos y reflexionar acerca de las posibles formas de mejorarlos.

2 INTELIGENCIA ARTIFICIAL

En este capítulo se pretende dar una visión global de la Inteligencia Artificial, un conveniente punto de partida pues es la disciplina de la que surgen los métodos que se emplearán en la resolución del problema de este proyecto. Se comenzará dando su definición y un poco de trasfondo histórico, para después establecer y explicar los detalles más importantes de las clasificaciones en las que deriva: según el nivel de evolución y según la subrama.

2.1 Introducción a la Inteligencia Artificial

La Inteligencia Artificial (con siglas IA) es aquella disciplina académica, perteneciente a las Ciencias de la Computación, que pretende emular algunos procesos de la percepción sensorial humana, como la visión o la audición, junto a sus correspondientes métodos de reconocimiento de patrones, en sistemas artificiales [1].

Es conveniente aclarar que, aunque mediante Inteligencia Artificial se puedan simular correctamente aspectos de la mente, como nuestra capacidad de cálculo matemático o de reconocimiento de elementos en imágenes, el intelecto humano y un ordenador no son iguales ni funcionan de idéntica forma. Los procesos computacionales internos de un programa son diferentes a los procesos que ocurren en un cerebro humano durante la ejecución de la misma actividad, pese a que externamente parezca que hagan lo mismo. Por ejemplo, al jugar al ajedrez, el ordenador calcula miles de jugadas por segundo y puede predecir todos los movimientos del jugador gracias a un archivo con millones de partidas que puede consultar, algo que escapa a las capacidades humanas [2].

De naturaleza multidisciplinaria, la IA consta de varias áreas de especialización teórica. Ha necesitado de la contribución de herramientas de ramas científicas como la matemática, la estadística, la informática, el tratamiento de señales, el control automático, la robótica o la neurociencia, además de haberse visto beneficiada por investigaciones relativas a los campos de la psicología, la sociología o la filosofía [1]. Su aplicación más común a día de hoy es el tratamiento y análisis de datos, ya sean números, letras, imágenes, vídeos, sonidos, etc.

El origen del término IA se remonta a 1956, cuando el informático y matemático John McCarthy lo acuñó por primera vez durante la Conferencia de Dartmouth. Desde entonces, la Inteligencia Artificial ha vivido en exponencial desarrollo, implementándose en cada vez más actividades de diversos sectores, como la demostración de teoremas, el aprendizaje y juego de juegos de mesa, la traducción de textos, el estudio de estructuras biológicas, la optimización de estrategias de mercado... hasta consolidarse como una de las ciencias más importantes del presente e incluso del futuro, con innumerables aplicaciones y con cada vez mayor influencia en nuestras vidas, desde el punto de vista económico, social y cultural [3].

2.2 Niveles de la IA

La Inteligencia Artificial tiene tres etapas o evoluciones determinadas por el grado de inteligencia del sistema, es decir, el nivel de similitud que alcanza la automatización con determinados procesos cognitivos humanos:

2.2.1 Inteligencia Artificial Reducida o Estrecha

También conocida como IA débil (*weak AI*), *Artificial Narrow Intelligence* o ANI. Es el nivel de la IA al que pertenecen aquellos sistemas cuyas únicas funciones realizables están predefinidas de antemano. Tan solo pueden hacer aquello para lo que han sido diseñados y tomar decisiones basadas en sus datos de entrenamiento. No tienen consciencia ni habilidad para pensar o razonar.

Todos los sistemas de Inteligencia Artificial actuales se encuentran en este nivel. Pueden tener un mejor rendimiento que los humanos en tareas particulares, como identificación de enfermedades o reconocimiento de sonidos, pero al retirarlos de sus 'zonas de confort', su funcionalidad se reduce drásticamente o incluso se anula si no han sido programadas para estas nuevas actividades. A esto hay que sumarle la posibilidad de que, de entre los miles o millones de datos con los que se ha entrenado la máquina, algunos sean erróneos o estén condicionados por determinados intereses (subjetividad, *biased data*), lo que se traduce en un mayor porcentaje de error en su funcionamiento, distorsionando los resultados. La máquina internalizará como verdadero algo que realmente es falso, siendo incapaz de justificar por sí misma el error [4]. Por lo tanto, no puede reproducir el comportamiento humano, solo simularlo en determinadas actividades.

Posiblemente los ejemplos más claros de IA débil que utilizamos en nuestro día a día serían los asistentes personales, como Siri, Alexa o Cortana, los sistemas de reconocimiento facial o imágenes usados por Facebook y Google, los vehículos autónomos y semiautónomos, como los coches Tesla, o los traductores web, como Google Translate. Pero realmente la IA débil lleva años presente en muchos más ámbitos aparte de la ingeniería, la informática, la robótica o las matemáticas, que son normalmente los primeros con los que se le relaciona. Está muy extendida en sectores como la medicina (ayuda a dar diagnósticos, análisis de imágenes médicas...), la biología (análisis de estructuras biológicas y genética), la economía (minería de datos, análisis financiero o de riesgo...) e incluso la física (demostración automática de teoremas) [5].

2.2.2 Inteligencia Artificial General

Llamada IA fuerte (*strong AI*), *Artificial General Intelligence* o AGI. Involucra a aquellos sistemas con la misma capacidad de pensar y tomar decisiones que un ser humano, pero a una velocidad superior y con una mayor base de conocimientos dada su naturaleza computacional.

Actualmente, no existe ningún sistema perteneciente a esta categoría. Los expertos en el campo muestran distintas opiniones sobre la factibilidad de su materialización, pues creen que las máquinas requerirían algunas cualidades propias de los humanos, como la consciencia (facultad para mirar dentro de uno mismo y percatarse de la propia existencia), conciencia (percepción del bien o del mal) o emociones (comprenderlas y tener propias), cuyo funcionamiento ni siquiera se comprende a la perfección desde el punto de vista biológico.

Su programación parece improbable con los conocimientos actuales, pero dada la gran velocidad con la que se progresa en el área, sobre todo en la última década, teniendo en cuenta de ley de Moore y su exponencialidad actual (aproximadamente, cada 2 años se duplica la cantidad de transistores de un microprocesador), no sería descabellado pensar que su fabricación sea inminente. A este instante también se le denomina singularidad tecnológica, definida como el hipotético momento futuro en el que la inteligencia computacional se iguala o supera a la humana, causando un enorme impacto en una sociedad desprevenida ante tal vertiginoso y crucial cambio.

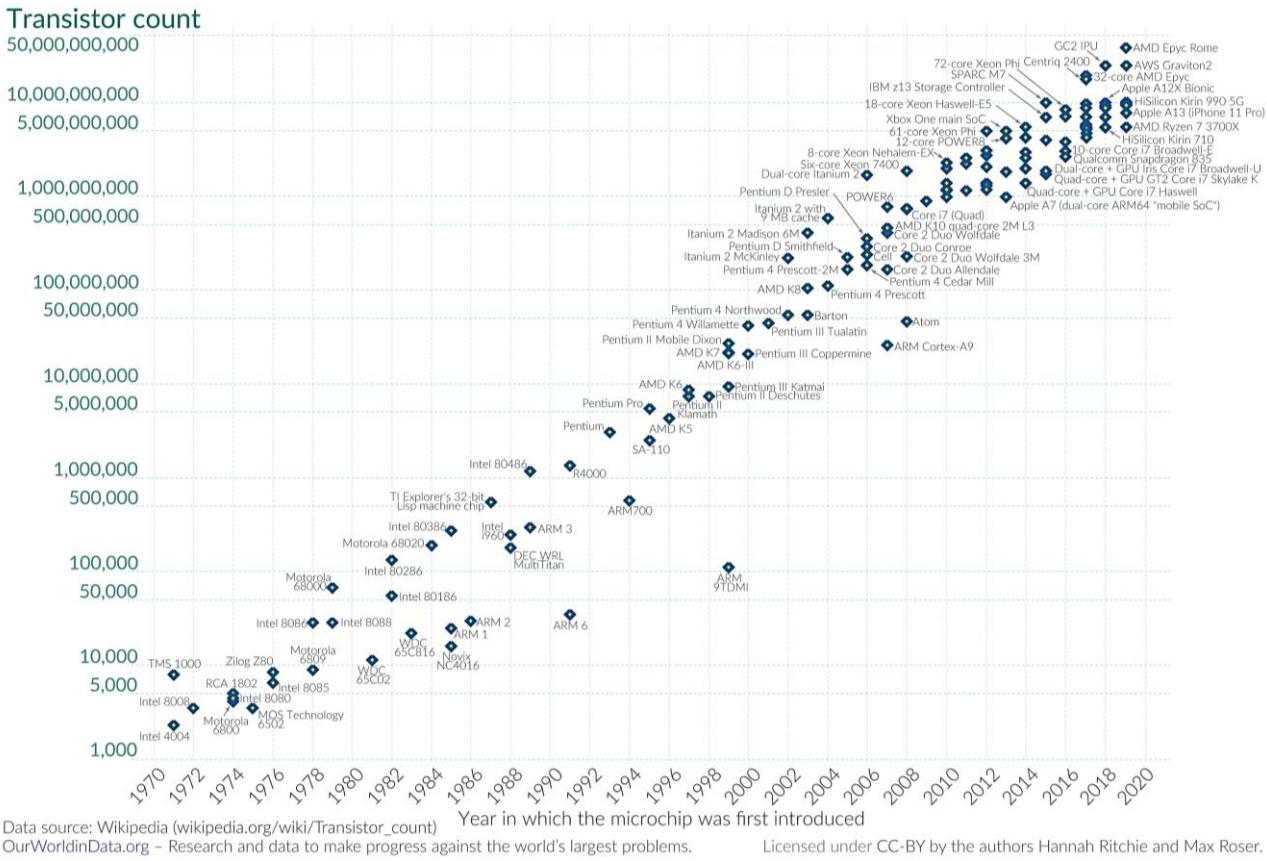


Figura 2-1. Ley de Moore: aumento al doble de transistores en microprocesador cada 2 años [6]

En 2012, se realizó una encuesta a 550 investigadores expertos de Inteligencia Artificial, preguntándoles una estimación del año en el que pensaban que ya estaría creada la primera AGI. De todos los encuestados, solo 37 (un 6.7%) opinaban que nunca se alcanzaría dicha máquina [7]. El resultado de los otros 463 que sí creían se muestra en la siguiente imagen:

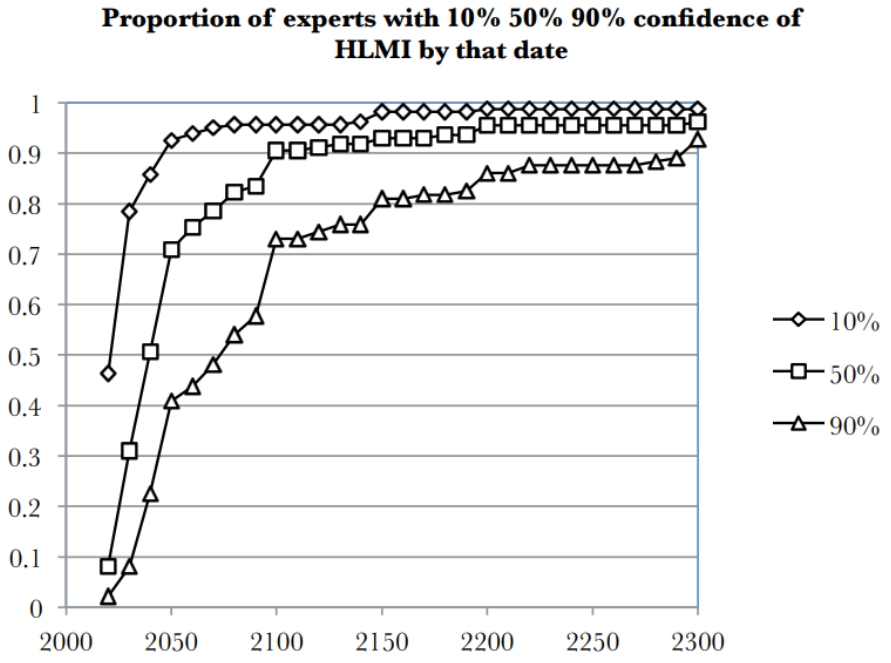


Figura 2-2. Resultado de encuesta para estimación del año de creación de la primera AGI (2012) [7]

Se representan 3 gráficas según la seguridad o confianza que tenían en su predicción: poca (10%), bastante (50%) y mucha (90%). En el eje x se indica los años, y en el eje y se señala el porcentaje de participantes que creen que la singularidad ya habrá ocurrido sobre dicho año. Las respuestas están bastante distribuidas. Se observa que para antes del año 2100 la gran mayoría de los participantes predicen la existencia de una Inteligencia Artificial General.

Más tarde, en 2019, se preguntó a un grupo de 32 investigadores doctores del campo de la Inteligencia Artificial, de nuevo, sobre la fecha estimada en la que se produciría la singularidad tecnológica [8]. Los resultados se recogieron en la siguiente gráfica:

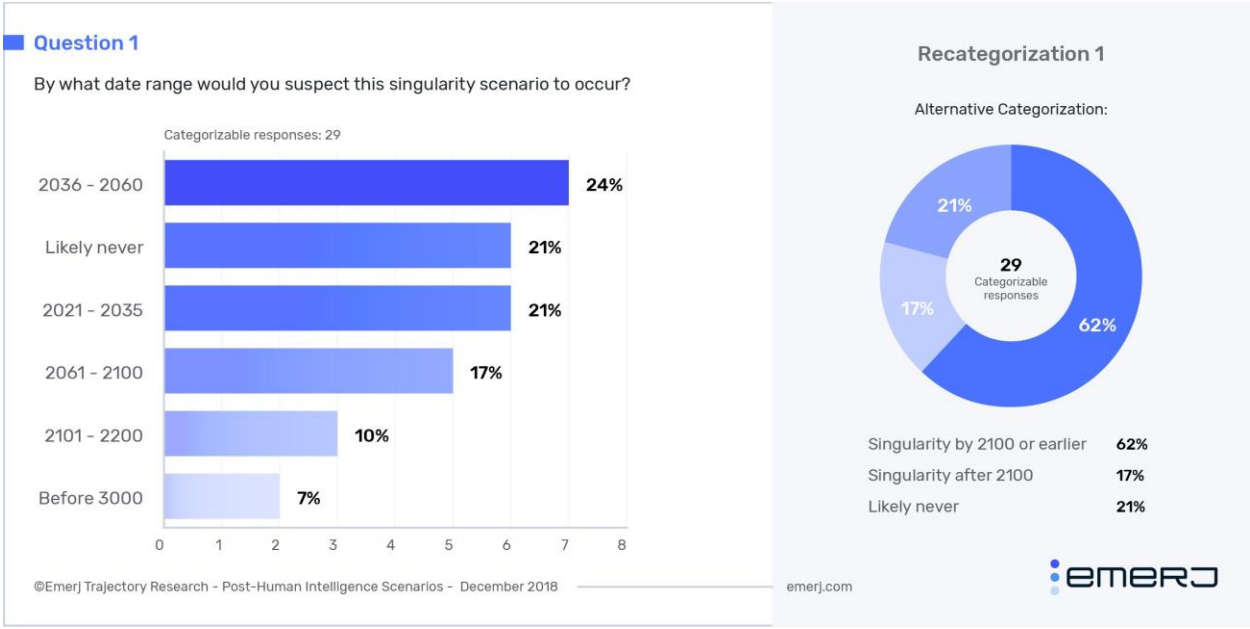


Figura 2-3. Resultado de la encuesta sobre la fecha de la singularidad tecnológica (2019) [8]

La mayoría de los encuestados, un 79%, cree que este evento tendrá lugar tarde o temprano, y su grueso, un 62%, opina que ocurrirá antes del año 2100, igual resultado que en la encuesta de 2012 [7]. Además, el intervalo de tiempo con mayor cantidad de votos es del año 2036 al 2060, con un 24%, seguido de cerca por el rango 2021-2035, con 21%. Estos resultados demuestran la incertidumbre y la disparidad de opiniones que hay en la comunidad científica acerca de no solo una fecha estimada, sino de la certeza de que realmente llegue este momento.

Por otro lado, la ocurrencia de la singularidad tecnológica genera grandes expectativas, pero también inquietudes, desatando preguntas filosóficas, dilemas morales y éticos y disputas en la comunidad científica. Algunos piensan que la creación de AGI podría dar lugar a un desarrollo extremadamente positivo de la sociedad, siendo un momento tan crítico en la historia de la humanidad que provocaría incluso un cambio de era, abandonando la actual Edad Contemporánea. No obstante, otros temen la posibilidad de que esta nueva tecnología se convierta en objeto de estudio en el sector armamentístico y sea protagonista de futuras guerras, como ha venido aconteciendo a lo largo de nuestra historia, o que se construya una IA fuerte cuyos valores e intereses sean contrarios a los de la humanidad, suponiendo una amenaza para la especie [9].

2.2.3 Superinteligencia Artificial

Conocida como *Artificial Super Intelligence* o ASI. Concepto hipotético, basado en conjeturas de expertos, cuya materialización es aún más lejana que la de la IA fuerte. Se refiere a aquellos sistemas de Inteligencia Artificial que superarían a los humanos en cualquier ámbito, incluso en aquellas actividades que implican habilidades humanas que nos hacen únicos, como la empatía, la creatividad, la capacidad de juicio, la planificación, las capacidades físicas e incluso la capacidad de adaptación [10].

Si una máquina evolucionara a AGI, sería capaz de perfeccionarse a sí misma de forma exponencial sin intervención humana, llegando al nivel ASI. Este proceso, de AGI a ASI, se demoraría muchísimo menos que el paso en el que nos encontramos actualmente, de ANI a AGI, pues un ordenador con la capacidad de pensar y razonar humana, a la que se agregarían las características correspondientes propias de su origen sintético (mayor velocidad, infatigabilidad, elevada cantidad de datos almacenados...), podría aprender de la experiencia o por prueba y error a un ritmo frenético y a pasos agigantados, llegando a mejorarse a sí mismo o incluso a diseñar otras máquinas superiores a las de los humanos, ocasionando en un bucle recursivo, perpetuo, de automejora. A esto se le conoce como explosión de inteligencia (*intelligence explosion*) [11], término acuñado por el matemático I. J. Good allá en 1965.

2.3 Ramas de la IA

Partiendo de que el objetivo de la Inteligencia Artificial es, como se ha mencionado previamente, simular ciertos procesos cognitivos humanos, o sea, las operaciones mentales que realiza el cerebro para captar, procesar y almacenar información del entorno, es lógico suponer que cada una de sus ramas está basada en uno o varios pasos de cualquiera de estos procesos humanos. A continuación, se describen brevemente algunas de las ramas más conocidas y utilizadas de la IA [12] :

- **Robótica**

  Es el ya popular estudio y construcción de máquinas, o en líneas generales, ordenadores con capacidad de movimiento, capaces de desempeñar tareas que podrían ser realizadas por el ser humano, pero con aún mayor fuerza o precisión. Ahí es donde entra la Inteligencia Artificial, dotándolos de cierta inteligencia, contribuyendo a crear robots mucho más hábiles, útiles y resueltos.

- **Sistemas Expertos**

  Consiste en la creación de sistemas informáticos que intentan emular el razonamiento o la habilidad de tomar de decisiones propios de un ser humano experto en su área de conocimiento, con el objetivo de resolver problemas complejos dentro de su dominio. Suelen estar compuestos por tres partes bien definidas:

  - Una interfaz de usuario, es decir, el medio por el que los usuarios realizan las preguntas al sistema y por donde posteriormente reciben las respuestas.
  - Una base de conocimientos, que contiene una secuencia de reglas con estructura *if-then* e información de un dominio particular aportada por un experto real.
  - Un sistema de inferencia, que tras recibir entradas desde la interfaz de usuario, consulta la base de conocimientos y extrae las reglas y los datos necesarios, que son manipulados por un algoritmo para finalmente razonar una respuesta que se devuelve al usuario.

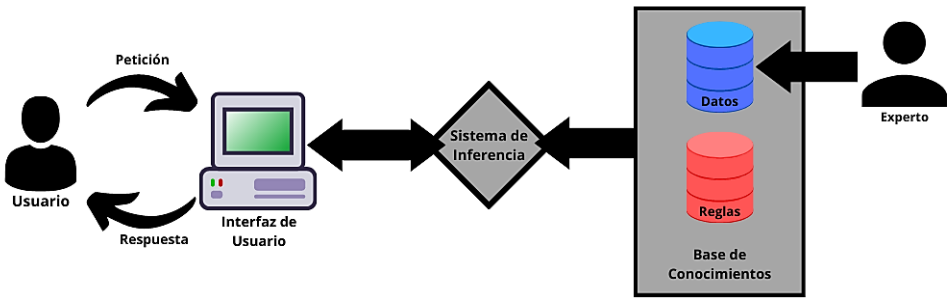


Figura 2-4. Esquema típico de un sistema experto

- **Lógica Difusa (*Fuzzy Logic*)**

  Es la lógica que puede tratar expresiones que no son ni completamente ciertas ni totalmente falsas. Fue presentada en 1964 por el matemático Lofti Zadeh para que las máquinas pudieran decidir, al igual que los humanos, a partir de las llamadas variables lingüísticas de nuestro lenguaje natural que utilizamos

para cuantificar algo, pero que generan imprecisión e incertidumbre (por ejemplo: poco, mucho, casi, apenas, muy, bastante...).

Está basada en la teoría clásica de conjuntos. Si en la lógica binaria se emplean únicamente los dígitos 1 para verdadero o perteneciente al conjunto y 0 para falso o no perteneciente al conjunto, la lógica difusa utiliza también los valores intermedios, de forma que un valor que yace en el intervalo [0, 1] es clasificado en función de su proximidad con los extremos y mediante una función de pertenencia (función de probabilidad a trozos). Es similar a una representación probabilística, con cierta probabilidad de pertenecer a un conjunto (llamado conjunto borroso) o a otro.

Su aplicación más habitual es la de ampliar la funcionalidad de los sistemas expertos, aunque también se usa para crear controladores difusos, como sería el controlador borroso de un termostato, que en lugar de cambiar la temperatura del recinto al detectar solo temperaturas calurosas o heladas, puede modificarla al detectar, por ejemplo, "un poco de frío", "bastante calor", "apenas frío", etc. [13]

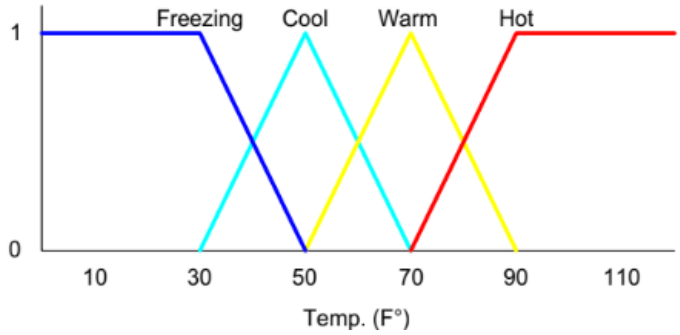


Figura 2-5. Ejemplo de función de pertenencia de un termostato con controlador borroso [14]

- **Procesamiento del Lenguaje Natural, NLP (*Natural Language Processing, NLP*)**

  Es la rama que estudia la comunicación o interacción con las máquinas empleando el lenguaje natural, es decir, las lenguas de todo el mundo. Busca optimizar esta relación en el sentido de la eficacia computacional. El proceso de hacer que una máquina entienda, interprete y procese lo que una persona le diga o le escriba se denomina Procesamiento del Lenguaje Natural.

  El primer paso es la conversión de las entradas a texto. En el caso de que sea audio será necesario un algoritmo para su transcripción. Después, el sistema NLP separa el texto en componentes, averigua el significado de cada una gracias a su base de datos, entiende el contexto de la conversación a partir de unas técnicas de aprendizaje previamente implementadas y, por último, determina qué comando debe ejecutar.

Además de las ya vistas, deben considerarse otras ramas de la IA que, debido a su gran importancia en el actual proyecto, se explicarán con mayor detalle en unos capítulos dedicados. Estas son:

- **Aprendizaje Automático (*Machine Learning*).**
- **Visión por Computador (*Computer Vision*)**

3 APRENDIZAJE AUTOMÁTICO

El propósito de este capítulo es abordar teóricamente todo el proceso de creación de un modelo basado en Aprendizaje Automático, desde la obtención de los datos hasta la puesta a punto, explicando cada uno de los términos que pueden ayudar a hacer frente al problema de clasificación de lugares emblemáticos.

3.1 Introducción al Aprendizaje Automático

Los problemas que son complejos para la mayoría de los humanos, normalmente aquellos que se rigen por reglas muy estrictas, en un entorno delimitado y perfectamente especificado, suelen ser tareas en las que los ordenadores rinden magníficamente, siendo capaces de predecir situaciones futuras a partir del conocimiento de la situación actual y las reglas que modelan el escenario [15], todo introducido por un programador previamente.

Sin embargo, algunas acciones de nuestro día a día, como reconocer un objeto, un sonido o un ser vivo, que podrían ser holgadamente realizadas por incluso niños de preescolar, pueden resultar realmente complejas de implementar y resolver para un ordenador, dado el gran nivel de abstracción necesario. Sería preciso programar de forma explícita cada uno de los parámetros correspondientes, lo que ya conllevaría muchísimas horas de trabajo, que tendrían que repetirse para cada uno de los escenarios posibles, elevando el tiempo requerido a unas cifras disparatadas... Es por ello que surgió el Aprendizaje Automático, a modo de automatizar el proceso y aumentar la eficiencia considerablemente.

El Aprendizaje Automático, también conocido como Aprendizaje Máquina, del inglés *Machine Learning* (*ML*), es la rama de la IA y de la ciencia de datos que estudia la inducción o transferencia de conocimientos a computadoras mediante técnicas informáticas basadas en la capacidad humana de **aprender con la experiencia**, logrando automatizar actividades y mejorar el desempeño en determinadas tareas.

En las máquinas, esta experiencia y proceso de aprendizaje se manifiesta en forma de datos y algoritmos computacionales, respectivamente. Por lo general, al insertar estos datos en el algoritmo tiene lugar un proceso de reconocimiento de patrones que permite al modelo extraer conclusiones o hacer **predicciones** en nuevas observaciones por sí solo [16], con unos resultados que serán mejores o peores según factores como las propiedades de los datos insertados, los datos sobre los que se pretende inferir, la concordancia del modelo con la aplicación, etc.

Fases del proceso de ML

3.2 Análisis del problema

Antes de empezar a pensar una solución para el problema, hay que comprenderlo en su totalidad y estudiar las condiciones que lo rodean. En el área del Aprendizaje Máquina suelen plantearse tres preguntas previas, que sirven como guía al usuario, ayudándolo a entender los elementos del problema e incluso a razonar si un modelo de ML es el apropiado o si existe otra técnica alternativa más simple para alcanzar una respuesta [17].

3.2.1 ¿Cuál es el problema?

Existen varios métodos para lograr una superior asimilación del problema.

- Se recomienda parafrasear el enunciado, es decir, explicarlo con palabras propias para mejorar la comprensión de la información que contiene el texto. Para ello se emplean palabras menos técnicas, más sencillas, informales, que facilitan la memorización de lo leído.
- Otra técnica común es la definición formal de tres términos clave:
  - Tarea: el problema del que se quiere hallar la solución.
  - Experiencia: los datos a insertar en el algoritmo computacional.
  - Rendimiento: la medida a emplear para evaluar el funcionamiento de la solución.
- Anotar en una lista algunas suposiciones, fruto de la posible ambigüedad en la redacción del enunciado, ayudará a tomar decisiones en el momento de elegir un camino para hallar la solución, pudiendo variar a nuestro antojo el alcance o complejidad del problema, al mismo tiempo que aumenta el control que se tiene de sobre este.
- Basarse en otros problemas similares ya resueltos también es de gran utilidad y permite estar preparado ante posibles contratiempos o aclarar confusiones.

3.2.2 ¿Por qué quiere resolverse el problema?

La motivación y los beneficios que impulsen al usuario a buscar una respuesta son un factor determinante. No es lo mismo resolver el problema a modo de ejercicio práctico para adquirir nuevas habilidades, utilizar métodos poco conocidos por uno mismo, que quizás no sean los más adecuados, para ganar destreza, experimentar con los datos por curiosidad... que necesitar hallar una solución con el objetivo de mantener un puesto de trabajo y beneficiarse económicamente. La situación en la que se desarrolle la solución (pretensión, alcance, objetivo, material y tiempo disponible, conocimientos del usuario...) influirá, como es lógico, en los pasos del proceso.

3.2.3 ¿Cómo se resolvería el problema?

Una vez comprendido el enunciado y el por qué se pretende buscar una solución, algunos especialistas en el campo aconsejan en primer lugar planificar y preparar a mano cada uno de los pasos del proceso, logrando un resultado básico inicial, una especie de prototipo, al que posteriormente se le aplican una serie de pulidos, refinamientos y optimizaciones logrando el resultado esperado, o al menos uno satisfactorio.

3.3 Obtención de datos

El Aprendizaje Máquina no existiría de no ser por los datos. Son necesarios para que el modelo aprenda, evolucione, se mejore a sí mismo. La cantidad, en su justa medida, y la calidad, la mejor posible, de la información que se utilice en el proceso es crucial, pues afectará directamente a la efectividad del modelo.

3.3.1 Tipos de organización de datos

El término *Big Data* sirve para describir a los datos que circulan por la red de difícil o imposible almacenamiento/procesamiento con equipos tradicionales, por lo que requieren de sistemas específicos. Estos datos quedan definidos por tres magnitudes fundamentales, conocidas como las tres V: volumen, velocidad y variedad. Volumen porque crecen exponencialmente con el paso del tiempo, tanto que los expertos conjeturan que para el año 2025 su volumen se habrá elevado hasta los 163 zettabytes (10^{21} = mil trillones). Velocidad por

la rapidez con la que son creados, accedidos, almacenados y procesados, a veces en tiempo real. Y variedad porque al provenir de distintos orígenes o fuentes, existen distintos tipos [18], que son:

- **Datos estructurados (*Structured Data*)**

  Son aquellos altamente organizados y formateados de tal manera que cada uno de sus componentes se localizan e indexan con facilidad.

  Se almacenan en archivos o bases de datos relacionales conocidas como ***datasets***. Una base de datos relacional es aquella cuyos elementos están relacionados entre sí, y en la que los datos se representan de forma intuitiva y directa en tablas, administradas por el denominado lenguaje de consulta estructurado (SQL). Normalmente, las tablas son archivos de extensión .xls o .csv.

  En estas tablas, cada fila es un registro con un identificador único llamado clave que representa a una **muestra** o instancia de un tema particular. Cada columna contiene el llamado atributo, variable o **característica (*feature*)** que expresa una propiedad para cada muestra al adoptar distintos valores o categorías [19]. Los atributos pueden ser valores numéricos enteros o decimales, categorías, fechas o incluso matrices que conforman una imagen o un vídeo, entre otros.

  Las columnas que se usen como entrada al modelo de ML se llaman *input variables,* y la columna que contiene la característica a predecir se denomina *output variable* [20]. Los atributos crean un espacio, el **espacio de características**, cuya dimensión es igual al número de *input variables*, y en el que es posible representar cada muestra a partir de sus atributos, como un punto dado por un vector de posición, conocido como **vector de características** (*feature vector*) [16].

  Matemáticamente [16], se tienen las siguientes expresiones con la terminología que se usará durante el resto del trabajo:

$$D = \{\boldsymbol{x}_1, \boldsymbol{x}_2, \dots, \boldsymbol{x}_m\} \tag{3.1}$$

$$\boldsymbol{x}_i = (x_{i1}, x_{i2}, \dots, x_{id}) \in \mathcal{X} \tag{3.2}$$

  Siendo:

  - D el *dataset* que contiene *m* muestras (filas).
  - $\boldsymbol{x}_i$ la muestra i del *dataset*, es decir, un vector de características de *d* componentes en el espacio generado, $\mathcal{X}$.
  - *d* el número de características de las muestras, y por tanto, la dimensión del espacio de características generado, $\mathcal{X}$.
  - x_{ij} la característica j de la muestra i.

Title	Director	Stars	IMDb-Rating	Category	Duration	Censor-bo...	ReleaseYear
The Shawshank Redemption	FrankDarabont	TimRobbins, MorganFreeman, BobGunton, WilliamSadler	9.3	Drama	142min	A	1994
The Godfather	FrancisFordCoppola	MarlonBrando, AlPacino, JamesCaan, DianeKeaton	9.2	Crime,Drama	175min	A	1972
The Dark Knight	ChristopherNolan	ChristianBale, HeathLedger, AaronEckhart, MichaelCaine	9	Action,Crime,Drama	152min	UA	2008
The Godfather Part II	FrancisFordCoppola	AlPacino, RobertDeNiro, RobertDuvall, DianeKeaton	9	Crime,Drama	202min	A	1974
Schindlers List	StevenSpielberg	LiamNeeson, RalphFiennes, BenKingsley, CarolineGoodall	9	Biography,Drama,History	195min	A	1993
12 Angry Men	SidneyLumet	HenryFonda, LeeJ.Cobb, MartinBalsam, JohnFiedler	9	Crime,Drama	96min	U	1957
The Lord of the Rings: The Return of the King	PeterJackson	ElijahWood, ViggoMortensen, IanMcKellen, OrlandoBloom	9	Action,Adventure,Drama	201min	U	2003

Figura 3-1. Ejemplo de tabla relacional: Mejores películas según IMDb [21]

- **Datos no estructurados (*Unstructured Data*)**

  Son datos binarios sin estructura explícita o comprensible, por lo que no tienen valor hasta que se identifican y almacenan organizadamente. Por lo general, se conservan en aplicaciones, bases de datos NoSQL o lagos de datos (repositorios con datos sin procesar).

  Alrededor del 80% de la información total almacenada mundial pertenece a esta categoría. Para extraer información útil de estos grandes volúmenes de datos dispersos existen complejos procesos de minado de datos (*Data Mining*), que emplean a su vez métodos de IA, ML, estadística y sistemas de bases de datos.

  Cuando después se ordenan en forma tabular, como los datos estructurados, los elementos que conforman su contenido pueden ser buscados y categorizados para obtener información, y podrán ser utilizados en modelos de Aprendizaje Automático.

  Algunos ejemplos de datos no estructurados son los archivos PDF, los datos de ubicación o los datos de redes sociales. Tienen en común el hecho de ser almacenados y administrados sin que el sistema entienda el contenido.

  Existen empresas que se dedican a extraer los conocimientos valiosos de los datos no estructurados como nombres de usuarios, sus hábitos, fechas de actividad, gustos, comportamientos, etc., con el objetivo de ganar una gran ventaja competitiva respecto a las demás organizaciones, pues consiguen una visión más completa y profunda del mercado y una ayuda extra para la toma de decisiones [22]. La gran trascendencia que esto tiene en el sector empresarial está causando una aceleración en el desarrollo de las tecnologías de análisis de *Big Data*, gracias a una mayor inversión de esfuerzos y capital.

- **Datos semiestructurados (*Semi-structured Data*)**

  Son una forma de datos estructurados pero sin la estructura típica de estos. En lugar de recurrir a modelos relacionales, contienen unos marcadores que permiten separar semánticamente los elementos y forzar una especie de jerarquía de datos, facilitando el análisis y formando la estructura conocida como autodescriptiva [23]. Los ejemplos más comunes son los códigos HTML, XML y JSON.

3.3.2 Fuentes de datos

El set de datos (*dataset,* datos estructurados) del que se vaya a hacer uso será un factor crítico para la precisión del sistema. Su gran importancia queda reflejada en la popular expresión "*Garbage in, garbage out*", ampliamente usada en informática, refiriéndose a que la calidad de la salida de un modelo viene dada directamente por la calidad de la entrada. Excepto en proyectos formales de alcance mediano o elevado, el *dataset* rara vez es elaborado por uno mismo para un modelo de Aprendizaje Automático específico, dada la imposibilidad de disponer de todo el material, conocimientos teóricos o instrumental particular requerido en determinadas aplicaciones.

Por ejemplo, en el caso de que se pretenda construir un modelo para la detección de enfermedades de la retina será necesario disponer de un oftalmoscopio, de una cantidad elevada de pacientes enfermos y sanos a los que examinar y de los conocimientos mínimos para distinguirlos.

Por esta razón, suelen consultarse fuentes de datos (*data sources*), disponibles en la web, que proporcionan prácticamente cualquier tipo de *dataset* ya contrastado, de confianza, con imágenes, texto, tablas... obtenido de manera individual o resultado de una colaboración entre usuarios cualificados en el tema. Algunas de las páginas web más conocidas y seguras son *Dataset Search by Google*, *Visual Data Discovery*, *OpenML*, *UCI: Machine Learning Repository* y *Kaggle* [24].

3.4 Preparación de datos

Los modelos predictivos del Aprendizaje Automático siempre requerirán de un ajuste de los datos, ya sean *datasets* obtenidos en fuentes de confianza o datos recién recopilados por uno mismo (conocidos como *raw data*, que son datos sin procesar). El tipo de preparación de la información dependerá de su naturaleza (como los tipos de variables) así como de los algoritmos computacionales usados en el modelo (que pueden esperar un formato determinado de los datos de entrada).

En ciertas ocasiones, el ajuste de los datos puede realizarse automáticamente dentro del algoritmo de ML. No obstante, lo práctica más común es hacer las modificaciones manualmente antes de continuar con las demás fases. Esta acción se conoce como preparación o preprocesamiento de datos (*data pre-processing*) [20].

3.4.1 Clasificación y análisis de variables

Las variables en ML son las columnas del *dataset* y pertenecen a diferentes clases que, al mismo tiempo, se dividen en otras subclases. El modus operandi del programador para llegar a la solución propuesta variará en función de los tipos de datos que se manejen, y requerirán de la máxima comprensión posible para manipularlos adecuadamente. Típicamente, se sigue la siguiente clasificación [25] [26] [27]:

3.4.1.1 Variables Numéricas

También llamadas cuantitativas. Son aquellas cuyos valores son números, a los que es posible extraerles información valiosa al llevar a cabo operaciones con ellos, clave en el análisis estadístico. Básicamente, se emplean para contar o medir. Se pueden diferenciar en:

- **Discretas (Enteras).** Habitualmente sin parte decimal. Solo permiten tomar ciertos valores dentro de una escala, que puede ser infinita. Ejemplos típicos suelen ser el número de miembros de una familia, la cantidad de canales de televisión, el número de aulas en una escuela...
- **Continuas (Decimales).** Con parte decimal. Mide cualquier valor teóricamente infinito, o sea, cualquiera de los infinitos valores entre dos números enteros consecutivos dentro del intervalo seleccionado. Es el caso de medidas como el volumen de agua en una botella, el tiempo que tarda un archivo en descargarse, la temperatura de la nevera, el peso de un vehículo... Pueden ser categorizadas a su vez en dos tipos que comparten características de análisis:

- **De intervalo (*Interval*).** Son aquellas variables en las que existe un orden entre sus valores y el valor cero no implica ausencia del atributo. Además, las diferencias entre dos valores de la misma escala indican diferencias iguales también en el atributo en cuestión. Sin embargo, la razón (geométrica) entre dos números de la escala no tiene el mismo significado que la relación entre dichos números desde el punto de vista del atributo. Para comprender mejor este tipo de variable se suele utilizar un ejemplo: la temperatura en grados Celsius (ºC). La diferencia de temperatura de una sala con 20ºC y otra con 30ºC es la misma que si estuvieran a 30ºC y 40ºC, respectivamente. Pero una sala a 30ºC no está el doble de caliente que otra a 15ºC, ya que se utiliza una escala relativa en lugar de la absoluta [28]. Es decir, 0ºC no implican falta de temperatura. Lo correcto sería contar desde el hipotético cero absoluto, situado en -273.15ºC (que sí indica ausencia de temperatura), correspondiente a 0ºK (grados Kelvin). Por lo tanto, 15ºC equivalen a 278.15ºK, mientras que 30ºC a 293.15ºK, que como se indicó, no es el doble de caliente sino tan solo un 5.4% más caliente.
- **De razón (*Ratio*).** Mismas propiedades que las variables de intervalo excepto que el cero sí indica la ausencia de atributo, por lo que la razón geométrica entre dos números de la escala, a diferencia de la subclase anterior, sí tiene el mismo significado que esta relación desde la perspectiva del atributo. El caso más claro es el de la temperatura en grados Kelvin, porque 0ºK implica que no hay temperatura (cero absoluto), como se mencionó previamente. Otros ejemplos son las medidas como el peso, altura, distancia, velocidad...

3.4.1.2 Variables Categóricas

También conocidas como cualitativas. Son aquellas cuyos valores son nombres formados por letras o palabras, e incluso números. Su utilidad puede resumirse en agrupar o separar muestras en función de coincidencia o diferencia de las características, respectivamente. Suelen analizarse mediante moda, mediana o histogramas. Con ciertas técnicas, explicadas más adelante, pueden ser convertidas a valores numéricos, pero de los que no es posible extraer información útil a partir de operaciones aritméticas (no tienen cualidades cuantitativas). A menudo, se subdividen en 3 tipos:

- **Nominales.** Se corresponde con aquellas variables que tienen dos o más categorías (valores) posibles, mutuamente excluyentes, y sin ningún orden intrínseco entre ellas. Por ejemplo, los campos de un formulario que deben ser rellenados con datos personales: país, provincia, nombre, apellidos...
- **Dicotómicas (*Dichotomous*).** A veces llamadas binarias, y en algunos casos se definen como una subclase nominal. Son las variables con únicamente dos categorías posibles. Por ejemplo, al pedir la opinión acerca de una decisión tomada por una entidad las respuestas posibles podrían ser ‘De acuerdo’ o ‘En desacuerdo’. Dentro de esta clase se encuentran las variables **Booleanas**, aquellas cuyos valores solo pueden ser ‘Verdadero’ o ‘Falso’.
- **Ordinales.** Son las variables que tienen dos o más categorías en las que es posible interpretar un orden o clasificación. Para cada muestra, la variable adopta un valor escogido de una escala en la que cada categoría guarda una relación posicional con la siguiente. Por ejemplo, el grado de satisfacción de un cliente en una encuesta puede tomar las categorías ‘Muy insatisfecho’, ‘Insatisfecho’, ‘Neutral’, ‘Satisfecho’ o ‘Muy satisfecho’, siendo evidente la presencia de cierto orden: desde el más negativo hasta el más positivo.

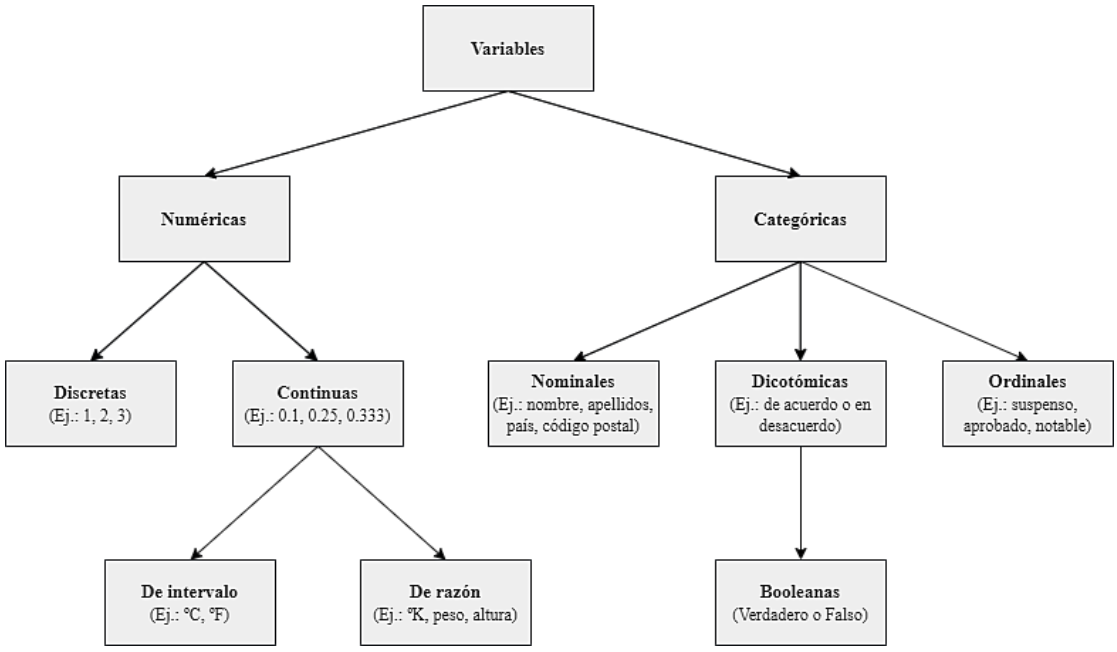


Figura 3-2. Clasificación de variables

3.4.2 Limpieza de los datos

Consiste en arreglar los errores del *dataset*, plasmados en valores incorrectos en filas o columnas. Están normalmente provocados por fallos en la escritura, incongruencias, valores duplicados, valores faltantes, etc. La comprensión del tema del problema y los conocimientos de programación adecuados serán suficientes para identificar estos errores. La solución generalizada es recurrir a la eliminación de la fila o columna correspondiente, si el problema se repite con asiduidad, o a la corrección de los errores manualmente, si el problema es aislado.

En ciertos casos particulares, se opta por una solución menos extrema, si bien más compleja, para no desechar tanta información. En lugar de eliminar una fila o columna completa por la ausencia de algunos de sus datos, se insertan valores que tengan sentido dentro del contexto de la aplicación. Por ejemplo, en un *dataset* sobre encuestas realizadas a usuarios en Internet puede haber muestras (filas del *dataset*) en las que se han rellenado todos los campos (equivalentes a características, columnas) excepto el de la edad. Esta ausencia podría suplirse sustituyendo ese hueco en blanco por el valor medio de la edad de los encuestados que sí han especificado este campo, prácticamente sin perjudicar el resultado de la predicción. Otra solución, mucho más sofisticada y acertada, sería crear un nuevo modelo predictivo basado en Aprendizaje Máquina para predecir los valores faltantes [29].

3.4.3 Transformación de los datos

3.4.3.1 Conversión de variables

En ocasiones es conveniente realizar codificaciones en los tipos de algunas variables del *dataset*, ya que los algoritmos de Aprendizaje Automático pueden rendir mejor o pueden requerir obligatoriamente unos tipos de variables concretos para funcionar.

Algunos algoritmos manejan los datos categóricos son solvencia, pero la realidad es que la mayoría de los modelos esperan variables de entrada numéricas para lograr unas predicciones con resultados competentes. El principal ejemplo son las Redes Neuronales Artificiales.

Para llevar a cabo dichas conversiones, generalmente se recurre a funciones predefinidas propias del lenguaje de programación utilizado (Python, R, JavaScript...), localizables en librerías especializadas en ML, fácilmente descargables e importables. Los tipos de conversiones se resumen a continuación [25].

3.4.3.1.1 Discretización

Para codificar una variable numérica continua como una categórica o numérica discreta.

La discretización es un término utilizado en muchos otros ámbitos, como en el paso de señales analógicas a digitales en Procesado de Señales, para convertir datos continuos en su equivalentes discretos. Existen numerosas técnicas, pero la gran parte se inspiran en el agrupamiento (*binning*).

El funcionamiento de *data binning* se basa en dividir la escala de los datos originales en intervalos (*bins*). Cada grupo de datos originales que se sitúe en un mismo rango o *bin* es sustituido por un único valor representativo de ese intervalo, que puede ser letras o palabras elegidas por el usuario o un número discreto, normalmente el valor central del intervalo [30]. Existen numerosas formas de aplicación diferenciadas por la anchura de los intervalos usados.

Usar este método durante el preprocesamiento de datos ofrece una serie de ventajas. Las más destacables son que disminuye el uso de memoria, pues los datos se almacenan con menor precisión, y además ahorra tiempo al evitar posibles operaciones computacionales con variables más grandes. Asimismo, puede acercar, en el espacio de características, a las muestras que caen en el mismo intervalo en la dimensión de la variable a discretizar (toman exactamente el mismo valor, el representativo, en esa dimensión), facilitando las predicciones al concentrar aún más a las muestras similares.

Por otro lado, usar esta conversión conlleva un inevitable problema: se pierde información decimal y, como consecuencia, precisión en la predicción. Puede implicar errores fatales, dependiendo del nivel de fidelidad a la realidad requerido por el problema y por su contexto de aplicación.

3.4.3.1.2 Transformación Ordinal (*Label Encoding*)

Para codificar una variable categórica como una numérica discreta.

Es un método con unos fundamentos muy sencillos: consiste en convertir cada categoría de una variable en un número entero diferente. El único inconveniente es la posible confusión o malinterpretación que puede causar en el algoritmo. Este, al leer una secuencia de números, puede detectar algún tipo de jerarquía u orden que en realidad no existe, desembocando en un resultado menos correcto. Por otro lado, en ciertas aplicaciones este efecto tal vez se pueda aprovechar y tenga sentido [31].

Un ejemplo de lo susodicho se representa a continuación (Tabla 3-1). En la tabla izquierda se muestra una Transformación Ordinal que podría inducir a errar al algoritmo, introduciendo un orden fútil, una jerarquía que no busca, e inevitable con *Label Encoding*. En la tabla derecha, por otro lado, los números elegidos interesan en ese mismo orden, pues denotan una jerarquía que coincide con la de la escala de la variable categórica.

INGENIERÍAS (Var. Categórica)	**INGENIERÍAS (Var. Num. Discreta)**
Ing. de la Energía	0
Ing. Civil	1
Ing. Aeroespacial	2
Ing. Química	3
Ing. de las Tecnologías de Telecomunicación	4

CALIFICACIÓN (Var. Categórica)	**CALIFICACIÓN (Var. Num. Discreta)**
Suspenso	0
Aprobado	1
Notable	2
Sobresaliente	3
Matrícula de Honor	4

Tabla 3-1. Ejemplos de *Label Encoding*

3.4.3.1.3 Transformación One-Hot (*One-Hot Encoding*)

Para codificar una variable categórica como una binaria.

Soluciona el problema de *Label Encoding*, por ende, no genera malentendidos por la aparición de un orden ficticio. Su implementación no es intrincada: simplemente, cada categoría de una variable se convierte en una nueva columna y se le asigna 1 (verdadero) o 0 (falso) cuando corresponda [31]. La variable remanente se califica como variable *dummy*. El único aspecto negativo es la aparición de más columnas, que pueden suponer un aumento de complejidad y de memoria considerable.

En la Tabla 3-2 se muestra el mismo ejemplo anterior pero transformado mediante *One-Hot Encoding*. Esta vez se no se aprecia ningún orden en las categorías de la variable que conduzca a un error. A cambio, el número de columnas final se ha elevado en gran medida: se ha multiplicado por el número de categorías, aumentando hasta cinco veces el resultado del caso de *Label Encoding*.

	Variable Binaria o *Dummy*				
INGENIERÍAS **(Var. Categórica)**	**INGENIERÍAS** (Ing. de la Energía)	**INGENIERÍAS** (Ing. Civil)	**INGENIERÍAS** (Ing. Aeroespacial)	**INGENIERÍAS** (Ing. Química)	**INGENIERÍAS** (Ing. de las Tecnologías de Telecomunicación)
Ing. de la Energía	1	0	0	0	0
Ing. Civil	0	1	0	0	0
Ing. Aeroespacial	0	0	1	0	0
Ing. Química	0	0	0	1	0
Ing. de las Tecnologías de Telecomunicación	0	0	0	0	1

Tabla 3-2. Ejemplo de *One-Hot Encoding*

3.4.3.2 Escalado de los datos

El escalado de datos (*Data Scaling*) consiste en trasladar todos los datos de una misma variable al mismo rango numérico para que tengan una escala similar, con el objetivo de aumentar el rendimiento, la velocidad y la estabilidad del modelo, o simplemente para ayudar a mejorar la comprensión, interpretación, de los datos. Una mala elección de método o praxis puede arruinar la información de la que se dispone, distorsionando los datos, y con ello, el resultado final del aprendizaje. Entre sus diversas técnicas, destacan [32] [33]:

- **Escalado Min-Max (*Min-Max Scaler*).** Los datos se trasladan de su rango natural al rango [0, 1], acercando los puntos relativos a las muestras en el espacio de características, algo ventajoso. Esta modificación tiene como ventaja que, en este intervalo, los sistemas informáticos tienen una mayor precisión para manejar números continuos (reales) con notación de coma flotante. Se debe aplicar la operación (3.3) al valor de cada muestra que se corresponda a una determinada característica, siguiendo la terminología del apartado 3.3.1:

$$x_{ij}{}' = \frac{(x_{ij} - \min(\boldsymbol{x}_{:,j}))}{(\max(\boldsymbol{x}_{:,j}) - \min(\boldsymbol{x}_{:,j}))}, \qquad \forall\, i \in [1, m] \tag{3.3}$$

  Siendo:

  - $x_{ij}{}'$ el nuevo valor normalizado de la muestra *i* para la característica *j*.
  - $\boldsymbol{x}_{:,j}$ el vector cuyas componentes son todos los *m* valores del *dataset* original correspondientes a la característica *j*.

Suele emplearse cuando los datos se distribuyen de manera uniforme, sin muestras aisladas, y se conocen los límites superiores e inferiores de la distribución, es decir, los valores máximo y mínimo necesarios para la transformación.

- **Escalado con normas L1 y L2.** Consiste en dividir los valores del vector de características de una variable por su norma. Habitualmente se emplean la norma L1 o suma de los valores absolutos de los elementos del vector, cuyo uso se muestra en (3.4), y la norma L2 o raíz cuadrada de la suma de los elementos del vector al cuadrado, en (3.5).

$$x_{ij}{}' = \frac{x_{ij}}{\sum_{i=1}^{m} |x_{ij}|}, \qquad \forall\, i \in [1, m] \tag{3.4}$$

$$x_{ij}{}' = \frac{x_{ij}}{\sqrt{\sum_{i=1}^{m} x_{ij}{}^2}}, \qquad \forall\, i \in [1, m] \tag{3.5}$$

3.4.3.3 Normalización de los datos

La normalización de datos (*Data Normalization*) se encarga de modificar la distribución estadística de los datos. Por lo general, erróneamente suele utilizarse este término para referirse también al escalado de datos, y viceversa. A pesar de que tienen el mismo objetivo (transformar los valores numéricos para lograr otros datos con propiedades de utilidad), sus procedimientos y resultados son totalmente diferentes [34]. Esta técnica suele ser aplicada antes o después de usar una de las técnicas de escalado previamente mencionadas, por lo que puede considerarse complementaria. Destacan algunos métodos [32] [33]:

- **Estandarización (*Standard Scaler*).** También llamado puntaje Z (*Z-score*). Su finalidad es conseguir una nueva distribución de los valores de la variable que tenga media nula y varianza y desviación típica igual a 1. Es realmente útil para el análisis de datos pues facilita las comparaciones de estos al estar bajo el mismo estándar. Para ello se centra (restando la media) y se reduce (dividiendo por la desviación típica) la variable original:

$$x_{ij}{}' = \frac{(x_{ij} - \mu_{:,j})}{\sigma_{:,j}}, \qquad \forall\, i \in [1, m] \tag{3.6}$$

  Siendo:

  - $\mu_{:,j}$ la media de la columna de la característica *j*.
  - $\sigma_{:,j}$ la desviación estándar de la columna de la característica *j*.

- **Normalización Logarítmica (*Log Normalization*).** Se utiliza cuando solo unos pocos valores tienen un elevado número de puntos, repeticiones, en el espacio de características (o sea, muchas muestras poseen ese mismo valor para la misma variable) mientras que la gran mayoría de los valores apenas se repiten, generando una distribución desproporcionada de datos conocida como ley potencial (*power law).* Al aplicar la transformación, dicha distribución puede cambiar a una Gaussiana, facilitando ocasiones la comprensión o el análisis de los datos.

$$x_{ij}{}' = \log(x_{ij}), \qquad \forall\, i \in [1, m] \tag{3.7}$$

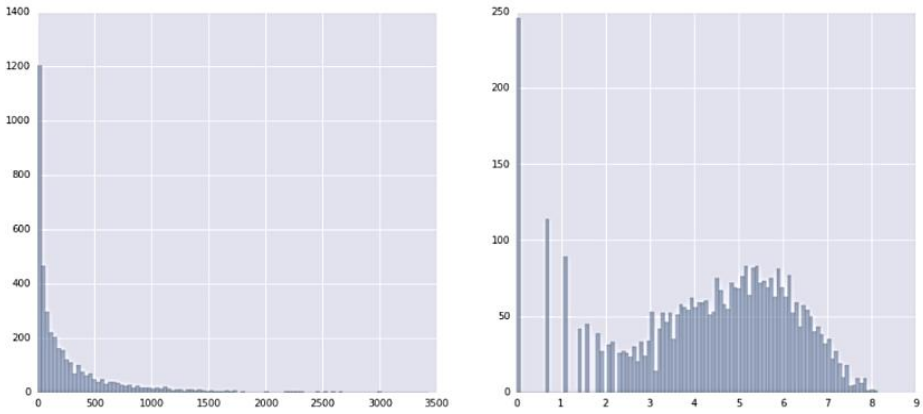


Figura 3-3. Ejemplo de *Log Normalization*: antes y después [33]

- **Recorte (*Clipping*).** Consiste en establecer un máximo y mínimo para el *dataset* que acoja a la gran mayoría de valores cercanos entre sí, y aquellos valores que estén fuera de este rango, que normalmente son los que más alejados están de la media (llamados *outliers*), se reasignan a estos mismos límites.

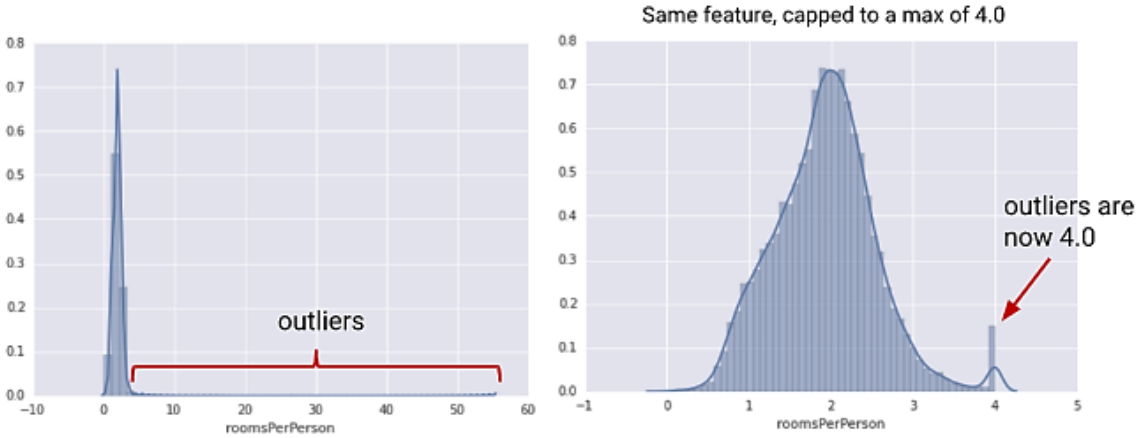


Figura 3-4. Ejemplo de *Clipping*: antes y después [33]

3.4.4 Reducción de dimensionalidad

En muchos casos es recomendable realizar un análisis de los datos para determinar qué variables usar durante el aprendizaje como *input variables*. Esto es uno de los objetos de estudio de la **Ingeniería de Características** (*Feature Engineering*). Esta tiene como tarea principal la transformación de *raw data* a datos procesados (*processed data*), por lo que entre sus subprocesos se encuentran el hallar, calcular, nuevas variables y, además, el seleccionar eficientemente el subconjunto de características que mejor se adapte al problema, eliminando del aprendizaje las variables irrelevantes o redundantes que despistan al programa y reducen su desempeño.

La Ingeniería de Características ocasiona una reducción de la dimensionalidad del espacio de características $\mathcal{X}$, evitando así la llamada "maldición de la dimensionalidad" (*curse of dimensionality*) que, en el caso del Aprendizaje Máquina, supone la necesidad de disponer de una exorbitante cantidad de muestras para asegurar ciertas combinaciones de valores de los atributos, aumentando el tiempo de procesado computacional y empeorando en muchos casos el rendimiento de la predicción [35].

Dicho de otra manera, al aumentar la dimensionalidad (número de características usadas), el espacio generado $\mathcal{X}$ aumenta considerablemente su volumen, elevando el número de dimensiones, haciendo que los puntos correspondientes a las muestras, al representarlos, estén más dispersos, dificultando cualquier método de aprendizaje. Entonces, para mantener el porcentaje de precisión del modelo, habría que aumentar exponencialmente el número de muestras para tener más puntos por región, con la esperanza de revertir la dispersión generada y hacer predicciones más acertadas. Este infructuoso efecto dificulta enormemente el trabajo a realizar por el usuario, por lo que es conveniente evitarlo.

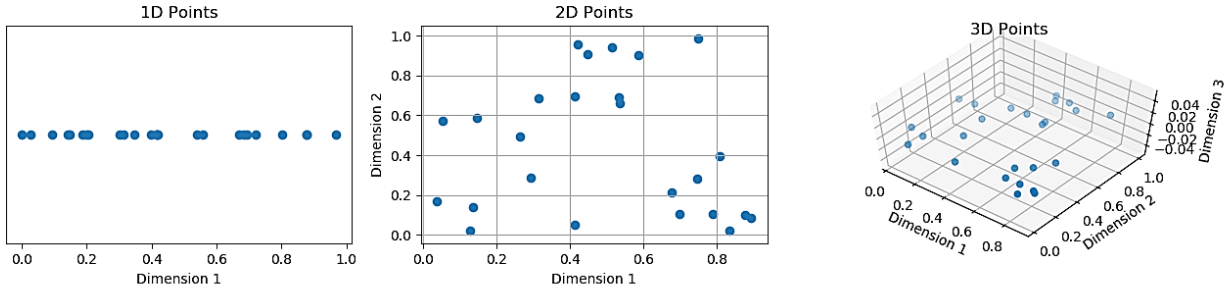


Figura 3-5. Dispersión como efecto negativo de la "maldición de la dimensionalidad" [36]

Según unas reglas generales obtenidas de forma empírica, para evitar este problema se debería contar con al menos 5 muestras por cada dimensión del espacio de características [37]. Según otras fuentes, se sugiere únicamente que el número de características empleadas sea menor que el número de muestras. En la mayoría de los casos prácticos actuales, el número de muestras utilizadas es al menos 10 veces mayor que el número de atributos [38].

El término "maldición de la dimensionalidad" y "fenómeno de Hughes" se utilizan indistintamente en el campo del ML dada su estrecha relación. Este fenómeno indica que al aumentar el número de *input variable*s de las muestras hasta cierto número, mejora el porcentaje de predicción del modelo, pero al sobrepasar este valor óptimo, su rendimiento va deteriorándose, al igual que ocurre con la "maldición de la dimensionalidad", por usar demasiadas características.

En la siguiente figura se muestra un ejemplo orientativo que refleja este fenómeno. En el eje de ordenadas se muestra el porcentaje medio de acierto, la precisión del modelo. En el eje de abscisas, la cantidad de características empleadas por muestra, 'n'. Varias gráficas son representadas, cada una con un número de muestras m distintas. Se observa una pendiente positiva (mejora de precisión) para cada gráfica según 'm', y el valor límite 'n' de cada una a partir del cual la pendiente decae, excepto en el caso hipotético de usar infinitas muestras.

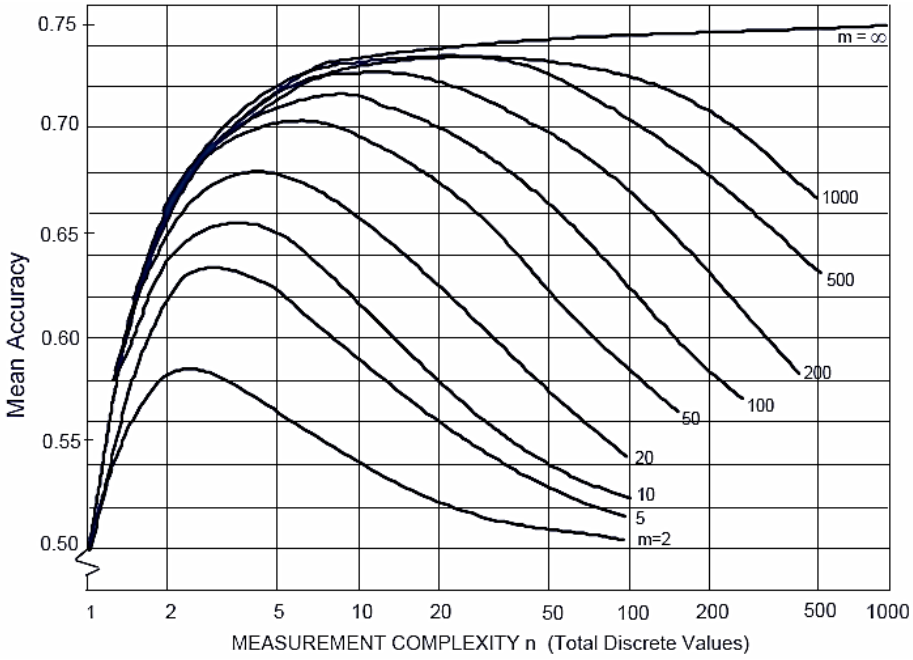


Figura 3-6. Ejemplo del "fenómeno de Hughes" [39]

Dentro de la Ingeniería de Características coexisten dos conjuntos de técnicas con exactamente el mismo propósito: la reducción de la dimensionalidad, que disminuye la complejidad del modelo, y la reducción de

probabilidad de aparición de sobreajuste (*overfitting*). Básicamente, el *overfitting* se produce por forzar al modelo a aprender demasiados datos o durante demasiado tiempo, como se verá en 3.7.1. Estas técnicas, que fueron referidas al inicio de este apartado, suelen confundirse la una con la otra, pero aunque tengan la misma finalidad, no son mutuamente excluyentes (de hecho, la costumbre es utilizar ambas):

3.4.4.1 Selección de Características (*Feature Selection*)

Busca el subconjunto de características, de entre todas disponibles, que produce un mejor resultado, dejando fuera del aprendizaje aquellas que solo despistan o ralentizan al modelo.

Existen dos clases de técnicas principales para la selección de características [40] [41]:

- **Métodos Envolventes** (*Wrapper Methods*). Se ejecutan todos los pasos del proceso (creación del modelo, entrenamiento y evaluación) varias veces, cada vez con un subconjunto de características de los datos de entrada distinto. Al finalizar, se comparan los rendimientos de todas las ejecuciones y se elige aquel subconjunto de atributos de la ejecución que mejor resultado dio. Estos métodos, basados en la validación cruzada, dan siempre el mejor resultado posible, pero su mayor inconveniente es la necesidad de entrenar el modelo para cada subconjunto, lo que conlleva un gran coste computacional y lentitud, además de su dependencia con el tipo de algoritmo empleado.
- **Métodos de Filtro** (*Filter Methods*). Se realizan operaciones estadísticas, similares a correlaciones, entre todas las características de los datos de entrada y la salida esperada del modelo. Tras esto, cada característica de entrada tendrá asignada una puntuación o coeficiente que facilitará la elección del subconjunto óptimo. Las mencionadas operaciones dependerán del tipo de variable (numérica o categórica) de cada característica de entrada y de la salida, y se muestran en la Tabla 3-3. La principal ventaja de este método radica en la gran velocidad con la que se ejecuta en comparación con los Métodos Envolventes, junto a su independencia con el tipo de algoritmo del modelo. Sin embargo, los Métodos de Filtro pueden dar lugar a una multicolinealidad, es decir, algunas de las características con mejor puntuación (más propensas a ser elegidas para el subconjunto) pueden estar muy correlacionadas entre sí, lo que empeoraría la interpretabilidad del modelo y aumentaría innecesariamente la cantidad de atributos utilizados, que como se verá a continuación, resulta perjudicial. Es clave tratar este problema antes de continuar con los siguientes pasos del proceso.

Característica \ Salida	Numérica	Categórica
Numérica	Correlación de Pearson	Análisis Discriminante Lineal (LDA)
Categórica	Análisis de Varianza (ANOVA)	Chi Cuadrado

Tabla 3-3. Coeficientes comúnmente usados en Métodos de Filtro

3.4.4.2 Extracción de Características (*Feature Extraction*)

Crea un *subset* más pequeño de nuevas variables no redundantes calculadas a partir del material disponible originalmente, reduciendo o comprimiendo los datos pero manteniendo la información relevante. Implica la transformación del espacio de características.

A continuación, se nombran algunos ejemplos habituales de extracción de características, cuyo uso depende de la temática de los datos [42]:

- **Variables de color:**
  - Momentos: media, desviación típica, asimetría...
  - Histograma.
  - RGB medio (*Average RGB*).

- **Variables de textura:**
    - Matriz de coocurrencia de nivel de gris (GLCM): entropía, contraste, correlación, energía, homogeneidad...
    - Tamura: aspereza, contraste, direccionalidad, rugosidad, linealidad (*Line-Likeness*), homogeneidad...
- **Variables estadísticas:** contraste, entropía, media cuadrática (RMS), curtosis, correlación, varianza, momentos centrales de orden 5 y 6, suavizado, media, desviación típica...
- **Variables geométricas:** área, pendiente, perímetro, centroide, convexidad, solidez...

3.5 Construcción del modelo

Es esencial decidirse por un modelo de ML que se adecúe en la medida de lo posible al tipo de problema que se pretende resolver y a la naturaleza del *dataset* que se maneja. En esta sección se darán las nociones básicas para comprender todas las técnicas de Aprendizaje Automático actuales, y se hará hincapié en las técnicas de clasificación, pues son el principal interés de este trabajo.

3.5.1 Métodos de Aprendizaje

Un modelo de ML puede aprender de cuatro maneras distintas, resumidas en los siguientes subapartados [43] [44] [45]. Cabe destacar antes que, independientemente del método de aprendizaje escogido, lo que se busca es conseguir modelos que hagan sus predicciones correctamente en nuevas muestras, y no solo en las usadas durante el aprendizaje. A esta habilidad se le llama **generalización** y está ligada al rendimiento del modelo.

3.5.1.1 Aprendizaje supervisado

Es el aprendizaje en el que los algoritmos aprenden de datos etiquetados. Un *dataset* está etiquetado cuando incorpora, además de las columnas de variables utilizadas en el aprendizaje (*input data*), otra columna que representa a la variable objetivo, la salida de cada muestra, conocida como **etiqueta (*label*, *output variable)***. Esta etiqueta es la que se pretende predecir en las nuevas muestras (una vez terminado el aprendizaje), mediante el reconocimiento de patrones en los nuevos *input data* a partir del histórico de *input data* de otras instancias con sus respectivas etiquetas. Si, en el mismo modelo, una muestra tiene una categoría distinta a la de otra muestra para la misma etiqueta, se dice que pertenecen a diferentes **clases**.

Cuando a cada muestra es necesario asignarle, predecirle, más de una etiqueta, el problema se denomina **multi-etiqueta**. Es decir, existen más de una columna de *output variable*.

Dependiendo del tipo de resultados que devuelven, se diferencian dos modelos:

3.5.1.1.1 Modelos de clasificación

Los resultados de la clasificación son etiquetas discretas. Es decir, cada etiqueta toma valores pertenecientes a un conjunto finito, limitado, de categorías posibles. Se hace una distinción según el número de categorías por etiqueta:

- **Problema Binario.** Cuando la etiqueta tiene únicamente dos categorías posibles. Por ejemplo: enfermo o no enfermo, tumor benigno o maligno, susceptible a beca o no...
- **Problema Multiclase.** Cuando la etiqueta puede tomar tres o más categorías. Por ejemplo: clasificación de imágenes de hojas, clasificación de imágenes de monumentos, clasificación del género musical de una canción...

Hoy en día, una gran cantidad de algoritmos de ML proporcionan los resultados de la clasificación como clases con probabilidades asignadas, y prácticamente siempre se asigna como etiqueta final de la muestra aquella con mayor probabilidad. También es costumbre fijar un umbral de probabilidad a la hora de asignar una clase a una muestra. Si no se supera, significa que no hay certeza sobre el acierto de la clasificación, y por lo tanto no se asigna ninguna categoría. Esta forma de proceder es muy habitual en aquellos sectores en los que el coste de una equivocación es muy alto o fatal [46].

Los algoritmos más usados para la clasificación se pueden dividir en no neuronales y neuronales. Se verán con mayor detalle en el apartado 3.5.2 y en la sección 4, respectivamente.

3.5.1.1.2 Modelos de regresión

Asigna a las variables objetivo valores continuos, reales, o lo que es lo mismo, valores numéricos dentro de un conjunto de infinito de valores.

Se utiliza normalmente para predecir el precio de un producto o una propiedad, o cualquier otro valor numérico como estatura, peso...

Las técnicas de ML para problemas de regresión más conocidas son la regresión lineal y no lineal, SVM, los árboles de decisión (*decision trees*), los bosques aleatorios (*random forest*) y las redes neuronales artificiales. Se observa que gran parte de los algoritmos usados en problemas de regresión también son utilizados en problemas de clasificación.

3.5.1.2 Aprendizaje no supervisado

A diferencia del aprendizaje supervisado, los datos utilizados para el aprendizaje no supervisado no incorporan su salida esperada correspondiente, no están etiquetados. Simplemente se dejan trabajar por su cuenta para intentar descubrir propiedades de interés de los datos o para organizarlos.

3.5.1.2.1 Modelos de agrupamiento

Del inglés *Clustering.* Se utilizan para encontrar agrupaciones (*clusters*) de muestras en base a la similitud de sus características, buscando que la homogeneidad dentro de cada grupo sea elevada y que la heterogeneidad sea la mayor posible entre distintos grupos. Hay dos formas de *Clustering* según cómo se organizan los grupos:

- **Jerárquico.** Los *clusters* se organizan jerárquicamente, quedando representados en un diagrama de datos en forma de árbol llamado dendrograma que se va formando con cada iteración a base de fusiones o divisiones. Cada muestra puede pertenecer a varios *clusters,* contenidos uno dentro de otro. A menor altura del nodo común más cercano, mayor será la similitud entre dos muestras. Su gran ventaja es que permite obtener distintas agrupaciones según el nivel en el que se observe el dendrograma (Figura 3-7). Existen diversas técnicas, pero todas basadas en uno de los dos caminos para llegar a la solución [47]:
    - **Aglomerativo.** También llamado ascendente, porque el procedimiento comienza con tantos *clusters* como muestras haya y, a medida que se va iterando, estos *clusters* iniciales se van combinando de par en par hasta que finalmente quede un único clúster y se haya completado el dendrograma. En resumen, se crea el dendrograma desde abajo.
    - **Disociativo.** También llamado descendente, ya que el procedimiento comienza con un único clúster que engloba todos los datos y en cada iteración del proceso se va dividiendo por la mitad, hasta que finalmente, se completa el dendrograma. Es decir, el dendrograma se crea desde arriba.

  El modelo se basará en métricas de similitud para elegir cómo se fusiona y cómo se divide. Las más empleadas para datos numéricos son las distancias euclidiana, de Manhattan, de Mahalanobis, la similitud del coseno, etc. También es posible emplear datos categóricos, utilizando en este caso métricas como las distancias de Hamming, de Levenshtein, etc. [48].

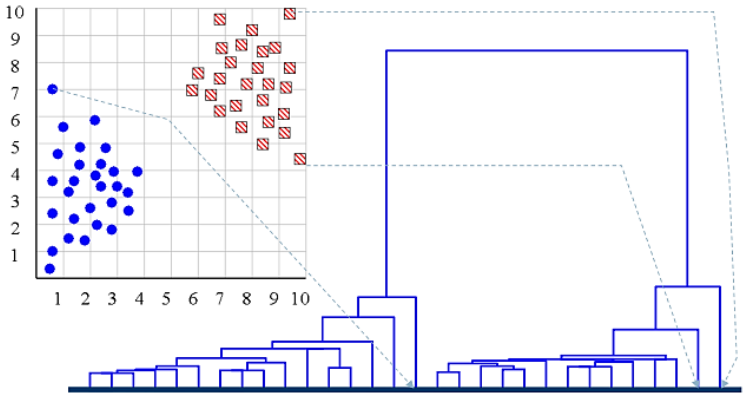


Figura 3-7. Ejemplo de dendrograma [47]

- **Particional (no jerárquico).** Agrupa individuos en *k clusters* excluyentes y sin jerarquía, siendo *k* un parámetro configurado por el usuario. Se elige una agrupación inicial de instancias y se van intercambiando sus miembros hasta conseguir un "mejor" *cluster*. Cada uno de los métodos basados en esta clase se diferencian por la manera en la que se elige el punto de partida, el cómo se intercambian las muestras y el cómo se evalúa un *cluster.* Ejemplos claros son el método de Forgy, *k-means*, *fuzzy k-mean*s, etc. [49].

3.5.1.2.2 Modelos de asociación

Usados frecuentemente para extraer información del *dataset* a partir del modelado y reconocimiento de su estructura o distribución interna, logrando identificar relaciones entre los datos que puede ayudar a la toma de decisiones. El principal uso es predecir la tendencia de una entidad a comprar algo basándose en su historial.

3.5.1.3 Aprendizaje semisupervisado

Se sitúa entre el aprendizaje supervisado y el no supervisado. Se utiliza en aquellas situaciones en las que se posee un *dataset* etiquetado y otro sin etiquetar. También se emplea cuando se dispone de un *dataset* no etiquetado inicialmente al que primero habría que etiquetar manualmente algunas de sus instancias.

Todas las instancias etiquetadas que se posean se introducen posteriormente en un algoritmo de aprendizaje supervisado para conseguir un modelo capaz de predecir. Con este se predicen las etiquetas del resto de muestras no etiquetadas, logrando así un *dataset* completamente etiquetado. Con este nuevo *dataset* se crea otro nuevo modelo supervisado y se obtiene el resultado final al problema.

3.5.1.4 Aprendizaje reforzado

Es aquel aprendizaje en el que se mejora el resultado de un modelo por medio de un proceso retroalimentación: se le recompensan las respuestas/decisiones acertadas pero se le penalizan las incorrectas. De esta forma, el algoritmo modifica su estrategia, forma de proceder, con el objetivo de conseguir más recompensas. Se emplea en modelos cuya funcionalidad óptima se espera para largo plazo. En la actualidad suele emplearse en juegos de mesa, videojuegos, robótica o sistemas de control. El modelo recibe las reglas que debe respetar y aprende a actuar por sí mismo. En un principio cometerá fallos, pero a base de prueba y error conseguirá comportarse adecuadamente.

3.5.2 Técnicas de clasificación no neuronales

En este apartado se describen los conceptos teóricos esenciales de algunos de los modelos de clasificación más empleados dentro del ML en la actualidad [50], exceptuando a las redes neuronales, a las que se les concede un capítulo propio por su trascendencia en este trabajo (Capítulo 4). La explicación se orientará hacia los problemas de clasificación con una sola etiqueta. Cabe recalcar que los métodos que se exponen a continuación se pueden utilizar para problemas tanto de clasificación como de regresión, según el tipo de variable de salida.

Dependiendo del problema o ámbito de aplicación, cada técnica tendrá sus ventajas y sus inconvenientes, y si el alcance del proyecto es de alto nivel, lo ideal sería que para decantarse finalmente por una de ellas se hayan probado individualmente y comparado los resultados.

Sea cual sea el método de clasificación, la meta siempre será predecir la clase de las muestras sin etiquetar a partir de sus vectores de características, $\boldsymbol{x}$, y un historial de ejemplos (muestras ya etiquetadas).

3.5.2.1 Árbol de decisión

Del inglés *Decision Tree*. Esta sencilla técnica está basada en las estructuras de los árboles reales, de ahí su nombre. Las hojas son las terminaciones del árbol y simbolizan los distintos valores posibles de una etiqueta, diferenciando las clases. Las ramas contienen operaciones lógicas, condiciones, para conducir a una muestra sin etiquetar a su hoja correspondiente en función de sus características. Estas operaciones lógicas son establecidas a partir del *dataset* etiquetado. Se considera un algoritmo voraz, adjetivo referido a aquellos algoritmos que se basan en buscar óptimos locales con la esperanza de acabar llegando a una solución general del problema. Su principal ventaja es su magnífico desempeño con cualquier tipo de datos [51].

Cuando se utilizan varios árboles de decisión en una clasificación, el clasificador se conoce como Bosque Aleatorio (*Random Forest*). El resultado final se obtiene tras una votación o promedio entre todos los árboles de decisión que lo componen. Es muy popular en el Aprendizaje Automático dada su simplicidad, gran rendimiento y eficiencia, aunque es tan poco interpretable que suele considerarse una caja negra [52].

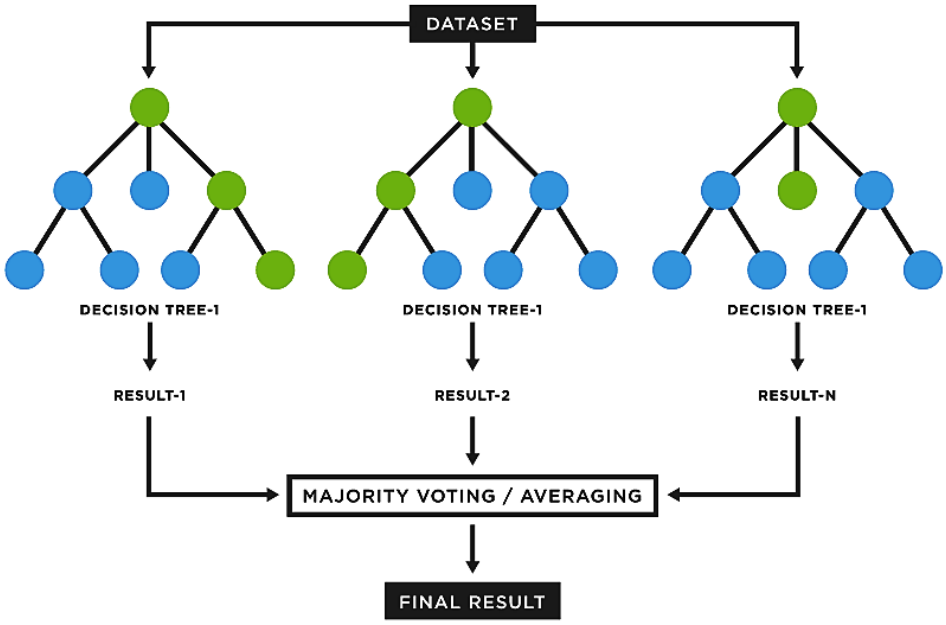


Figura 3-8. Ejemplo de *Random Forest* formado por tres *Decision Trees* [52]

3.5.2.2 Clasificador bayesiano ingenuo

Del inglés *Naïve Bayes Classifier* (NBC). Es un clasificador probabilístico que, a pesar de su sencillez, es de los más útiles y eficientes en la práctica. Se fundamenta en el Teorema de Bayes (3.8) y en la suposición de que las características de una clase son independientes (3.9), aunque realmente sí que sean interdependientes en el

ámbito de aplicación. Esta suposición, que le otorga dicho apodo, se llama independencia condicional de clase, y reduce la computación necesaria. El objetivo es encontrar la etiqueta con la mayor probabilidad de ser correcta dado un vector de características y usando la hipótesis, es decir, hallar la y_i que maximice (3.10) [53].

$$\mathrm{P}(y_i|\boldsymbol{x}) = \mathrm{P}(y_i|x_1, x_2, \dots, x_d) = \frac{\mathrm{P}(x_1, x_2, \dots, x_d|y_i)\ \mathrm{P}(y_i)}{\mathrm{P}(x_1, x_2, \dots, x_d)} \tag{3.8}$$

$$\begin{gathered}\mathrm{P}(x_1, x_2, \dots, x_d) = \mathrm{P}(x_1)\ \mathrm{P}(x_2) \dots \mathrm{P}(x_d) \\ \mathrm{P}(x_1, x_2, \dots, x_d|y_i) = \mathrm{P}(x_1|y_i)\ \mathrm{P}(x_2|y_i) \dots \mathrm{P}(x_d|y_i)\end{gathered} \tag{3.9}$$

$$\mathrm{P}(y_i|\boldsymbol{x}) = \mathrm{P}(y_i|x_1, x_2, \dots, x_d) = \frac{\mathrm{P}(x_1|y_i)\ \mathrm{P}(x_2|y_i) \dots \mathrm{P}(x_d|y_i)\ \mathrm{P}(y_i)}{\mathrm{P}(x_1)\ \mathrm{P}(x_2) \dots \mathrm{P}(x_d)} \tag{3.10}$$

Donde:

- $\boldsymbol{x}$ es el vector de características de una muestra a etiquetar, formado por *d* características.
- $\mathrm{P}(y_i)$ es la probabilidad a priori de la etiqueta *i.*
- $\mathrm{P}(x_1), \mathrm{P}(x_2), \dots, \mathrm{P}(x_d)$ son las probabilidades a priori de las características de una muestra.
- $\mathrm{P}(y_i|\boldsymbol{x})$ es la probabilidad a posteriori de que la etiqueta sea y_i si el vector de características es $\boldsymbol{x}$.
- $\mathrm{P}(x_1|y_i), \mathrm{P}(x_2|y_i), \dots, \mathrm{P}(x_d|y_i)$ son las probabilidades condicionales de que se tengan los valores $x_1, x_2, \dots, x_d$ respectivamente en el vector de características cuando la etiqueta es y_i.

Para calcular las probabilidades a priori $\mathrm{P}(y_i), \mathrm{P}(x_1), \mathrm{P}(x_2), \dots, \mathrm{P}(x_d)$ solo hay que contar el número de ocurrencias en las muestras etiquetadas y dividir entre el número total de muestras etiquetadas.

Para calcular las probabilidades condicionales $\mathrm{P}(x_1|y_i), \mathrm{P}(x_2|y_i), \dots, \mathrm{P}(x_d|y_i)$ únicamente hay que contar las entradas en las que coinciden ambos valores y dividir por el número total de entradas con etiqueta y_i.

Para cada muestra a etiquetar basta con calcular el numerador de (3.10), sin necesidad de calcular el denominador pues será el mismo para todas los valores de y_i. Finalmente, elegir aquella y_i que mayor probabilidad provoque.

3.5.2.3 Máquinas de vectores de soporte

Del inglés *Support Vector Machine* (SVM). Es un clasificador lineal. Para su funcionamiento, primero se representan las muestras en el espacio de características. Los puntos pertenecientes a la misma clase estarán cercanos entre sí. Cada muestra se puede expresar también como vector de características. Después, esta técnica calcula el hiperplano (subespacio plano con una dimensión menos que el espacio en el que se encuentra) que separa de forma óptima la nubes de puntos de dos clases distintas. Por óptima se refiere a que se busca el hiperplano con la distancia mínima entre las muestras y él lo más grande posible, o sea, el hiperplano con mayor margen. Por esta razón también se le conoce como clasificador de margen máximo. Para estos cálculos se requieren métodos de optimización. A los hiperplanos que se trazan paralelamente al hiperplano de separación por estos puntos más alejados de cada clase se denominan vectores de soporte. Así, las etiquetas se asignarán en base a donde se coloquen los nuevos puntos en el espacio de características: a un lado o al otro del hiperplano de separación.

Se suelen emplear cuando las separaciones entre clases son lineales. En caso contrario, se recurre al "truco de kernel", que consiste en añadir una nueva dimensión al espacio de características mediante unas transformaciones para separar clases fácilmente con límites de separación lineal que, al proyectarlos en el espacio original, se convierten en límites de separación no lineal.

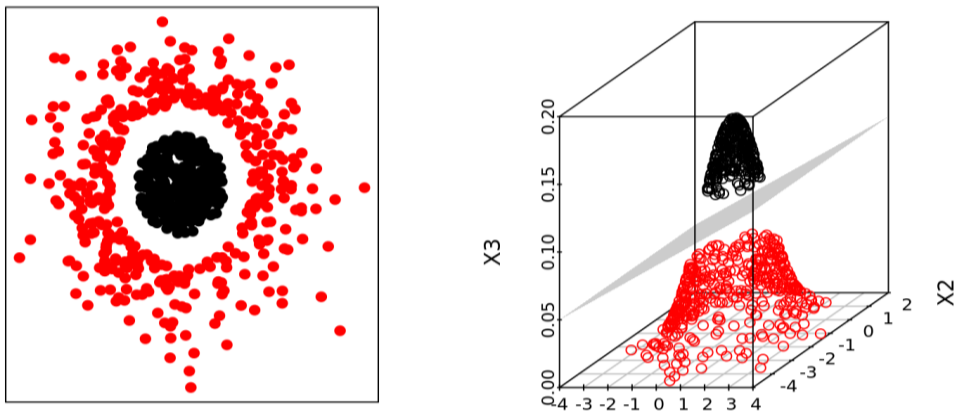


Figura 3-9. El "truco de kernel" para SVM [54]

En el caso de una clasificación con más de dos clases, la potencia computacional necesaria se eleva considerablemente. Hay tres técnicas para asignar etiquetas: *one-versus-one* (se realiza SVM entre todos los posibles pares de clases y se asigna la que más veces haya salido como resultado, similar a una votación, muy ineficiente), *one-versus-all* (se calcula el hiperplano entre una clase y todas las demás clases juntas, para todas las clases existentes y finalmente se asigna aquella clase que mejor puntuación dio) y DAGSVM (mejora de *one-versus-one* en términos de eficiencia, en el que al comparar dos clases se elimina la perdedora para el resto del algoritmo, reduciendo tiempo de ejecución, similar a un torneo con eliminación directa) [54].

3.6 Entrenamiento del modelo

Conocido como *model fitting*. Es la fase en la que el aprendizaje de los datos de entrada tiene lugar. Todos los modelos, sea cual sea el tipo de aprendizaje o la técnica en la que se basan, deben pasar por esta etapa. El entrenamiento consiste en optimizar o ajustar en sucesivas iteraciones los **parámetros** que controlan el comportamiento del modelo y que se calculan a partir de datos de entrada y de sus etiquetas, si se poseen. Estos parámetros se usarán para calibrar el modelo para la fase de predicción de nuevos datos. Por esta razón, un gran ajuste de los parámetros es necesario para desarrollar un modelo predictivo competente.

Estos parámetros son distintos para cada técnica. Por ejemplo, en el caso de un clasificador bayesiano ingenuo los parámetros se corresponderían con las probabilidades a priori de las clases y las probabilidades condicionales de las características con cada clase (3.5.2.2). En el caso de un SVM, los parámetros serían aquellos que definen el hiperplano de separación. Para árboles de decisión, las reglas de decisión serían los parámetros de entrenamiento. Y para algunos modelos basados en redes neuronales, los parámetros ajustados son los pesos de las conexiones, como se verá en el capítulo dedicado a ellas.

3.6.1 Funciones de pérdida

Los algoritmos de aprendizaje supervisado realizan el ajuste de los parámetros al introducirse datos de los que se predicen sus etiquetas y se comparan con sus etiquetas verdaderas. Es decir, reciben realimentación gracias a una referencia de salida. A partir de unas operaciones sobre las diferencias entre las etiquetas predichas y las reales, puede ir ajustando los parámetros iterativamente. Por otro lado, los algoritmos de aprendizaje no supervisado no tienen ninguna referencia de salida, su única forma de comprobar que están realizando el ajuste adecuadamente es introduciendo los datos no etiquetados del *dataset* y maximizando correlaciones entre los que pertenezcan a la misma agrupación.

Para ajustar los valores de los parámetros de entrenamiento, cada modelo utiliza una de las llamadas funciones de pérdida o coste (*loss functions*). El objetivo es **minimizar** su valor, pues representan la diferencia entre valores

reales y predichos (*bias*), y se pretende lograr la mayor similitud entre ellos. A continuación, se muestran las funciones de pérdida más populares en problemas de aprendizaje supervisado, donde:

- n es el número de muestras totales con las que se está entrenando.
- y_i es la etiqueta real de la muestra i con la que se está entrenando.
- $\hat{y}_i$ es la etiqueta predicha para la muestra i con la que se está entrenando.

Binary Cross-Entropy Loss (Negative Log Likelihood)	$BCEL = -(y_i \log(\hat{y}_i) + (1 - \hat{y}_i) \log(1 - \hat{y}_i))$	(3.11)
(Multi-class) Categorical Cross-Entropy Loss *(Codificación One-Hot, 3.4.3.1.3)*	$CCEL = -\sum_{i=1}^{n} y_i \log(\hat{y}_i)$	(3.12)

Tabla 3-4. Funciones de pérdida para modelos de clasificación

Mean Squared Error **(MSE)**	$MSE = \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{n}$	(3.13)
Mean Absolute Error **(MAE)**	$MAE = \frac{\sum_{i=1}^{n}\lvert y_i - \hat{y}_i\rvert^2}{n}$	(3.14)
Mean Bias Error **(MBE)**	$MBE = \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)}{n}$	(3.15)

Tabla 3-5. Funciones de pérdida para modelos de regresión

3.7 Validación del modelo

Una vez finalizado el entrenamiento, la validación del modelo (***model validation***) permite confirmar que el modelo funciona, que cumple su propósito, que tiene una buena capacidad de generalización sobre nuevos datos. De esta forma se solucionan los posibles problemas antes de que el modelo definitivo se someta a un testeo final (***model testing***) simulando su aplicación real o se implemente en su ámbito correspondiente donde pueda causar desastres, errores irreversibles...

3.7.1 Métricas

Es necesario el uso de métricas para evaluar el rendimiento de un modelo y poder compararlo con otros. Las métricas son un resumen del desempeño del modelo tras el aprendizaje, y solo sirven para ser valoradas por el usuario. No influyen directamente en el proceso de optimización. En base a los resultados observados en las métricas, se modifica el modelo en pos de un resultado más favorable. Es posible que los valores de las funciones de pérdida también sean de interés, por lo que eventualmente pueden llegar a actuar como métricas.

Al finalizar el entrenamiento, suele representarse en una gráfica la evolución de las métricas, comparando las calculadas tanto durante el entrenamiento como en validación, facilitando la interpretación del modelo por parte del usuario. También es común representar y comparar la evolución de la función de pérdida durante el entrenamiento y durante la validación.

Se habla de la **capacidad de generalización** de un modelo para referirse básicamente a su habilidad para hacer predicciones sobre datos desconocidos, después de haber extraído conocimientos de los datos de entrenamiento. Deben conocerse dos términos: ***bias*** (diferencia entre valores reales y predichos en entrenamiento) y ***variance*** (diferencia entre tasa de error de entrenamiento -*bias*- y tasa de error de validación). Gracias a las gráficas de las

métricas (y funciones de pérdida) pueden observarse una de las siguientes tres situaciones ligadas a la generalización de un modelo [55]:

- ***Underfitting* (subajuste).** Significa que el modelo no aprende bien los datos de entrenamiento, resultando en una mala generalización sobre nuevos datos. Ocurre cuando se tienen pocos datos de entrenamiento (o tienen ruido), cuando el proceso de entrenamiento es demasiado breve, y cuando el modelo es demasiado simple para el problema. Provoca alto *bias* y baja *variance*. Por esta razón, el modelo hará gran cantidad de predicciones incorrectas en validación. Las formas de solucionarlo son: aumentar los datos (buscar nuevas fuentes, *Data Augmentation*...), reducir las dimensiones del espacio de características disminuyendo *features* del *input data*, preparar mejor los datos, aumentar la complejidad del modelo, entrenar durante más tiempo, etc.
- ***Overfitting* (sobreajuste).** El modelo aprende demasiado los datos de entrenamiento, los "memoriza", incluyendo muestras no representativas, ruido, defectos y otros errores. Su capacidad de generalización sobre datos nunca vistos será muy mala. Solo actuará bien sobre datos idénticos a los del entrenamiento. Es decir, hay poco *bias* y alta *variance*. Las razones son el pequeño tamaño del *dataset* de entrenamiento (se aprende una y otra vez) o el desmedido tiempo de entrenamiento, además de una complejidad del modelo demasiado elevada para el problema. Las soluciones son aumentar los datos de entrenamiento o disminuir su duración, simplificar el modelo y recurrir a técnicas como *Early Stopping* (para de entrenar cuando no se detecta mejoría en métricas o funciones de pérdida), *dropout* en redes neuronales, etc.
- **Ajuste óptimo.** Está en un punto medio entre subajuste y sobreajuste. Se entrena el modelo hasta el momento en el que el error deje de disminuir y comience a aumentar. La generalización del modelo será buena. Se tiene bajo *bias* y baja *variance*.

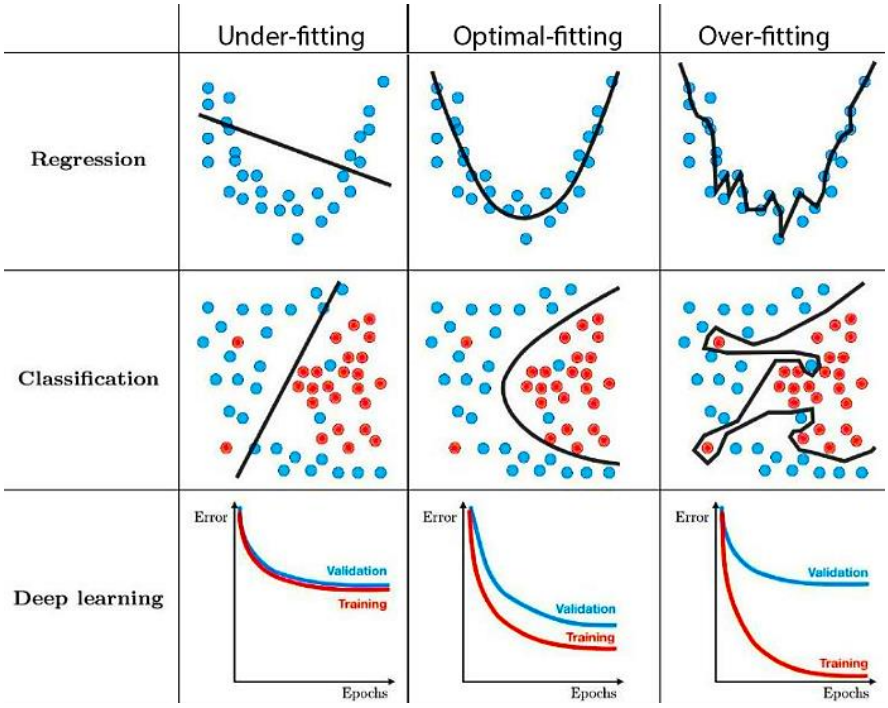


Figura 3-10. Ejemplos de *Underfitting*, *Overfitting* y Ajuste Óptimo [56]

Existe una gran variedad de métricas para la validación. Por simplicidad, se explicarán para el caso de una clasificación binaria, donde se emplean los términos [57]:

- **Verdaderos Positivos** (*True Positives, TP*): muestras con etiqueta real 1 y etiqueta predicha 1.
- **Falsos Positivos** (*False Positives, FP*): muestras con etiqueta real 0 y etiqueta predicha 1.
- **Verdaderos Negativos** (*True Negatives, TN*): muestras con etiqueta real 0 y etiqueta predicha 0.
- **Falsos Negativos** (*False Negatives, FN*): muestras con etiqueta real 1 y etiqueta predicha 0.

3.7.1.1 Matriz de confusión

Se representan los términos anteriores en forma matricial. Cada fila representa a una clase verdadera, y cada columna a una clase predicha. En cada casilla se coloca el recuento de muestras correspondientes. Se busca que la mayor cantidad de muestras se sitúen en la diagonal descendente pues implica acierto.

		PREDICHAS	
	Clases	Positiva (1)	Negativa (0)
REALES	Positiva (1)	TP	FN
	Negativa (0)	FP	TN

Tabla 3-6. Matriz de confusión en problema de clasificación binaria

Para problemas de clasificación multiclase, los valores TP, FP, TN y FN se calculan para cada clase según se indica en la Figura 3-11, simplemente contando las repeticiones. Para el cálculo de las siguientes métricas, se aplican primero las fórmulas para cada clase individualmente, obteniendo la métrica de dicha clase. Después se halla la métrica del modelo completo mediante un promediado, dentro del que existen varias alternativas: *macro* (mismo peso para todas las clases), *micro* (clases con más apariciones contribuyen más, tienen más peso), etc.

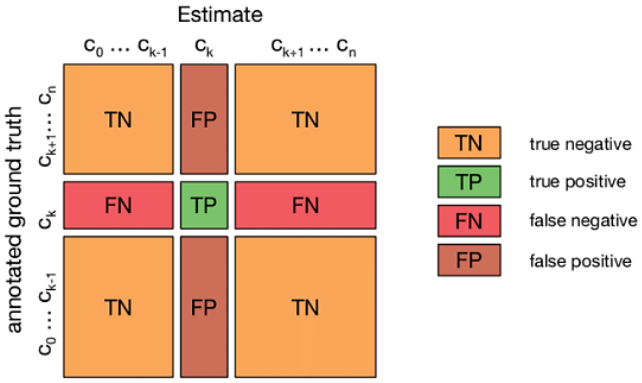


Figura 3-11. Recuento de TN, TP, FN, FP en problemas de clasificación multiclase [58]

3.7.1.2 Accuracy

Es la métrica más usada. No es un buen valor para evaluar el rendimiento en caso de demasiada diferencia entre el número de muestras de cada clase (*Imbalanced Data*).

$$Accuracy = \frac{TP + TN}{TP + FP + TN + FN} \tag{3.16}$$

3.7.1.3 Precision

La precisión indica el porcentaje de positivos predichos ($TP + FP$) que realmente son positivos (TP). Sirve para medir cuánta veracidad tiene el modelo cuando predice como positivo.

$$Precision = \frac{TP}{TP + FP} \tag{3.17}$$

3.7.1.4 Recall

O sensibilidad. Es el porcentaje de positivos reales ($TP + FN$) que fueron predichos como positivos (TP). Se utiliza en ciertas aplicaciones de sectores como el de la medicina, donde los FN penalizan aún más que los FP, pues normalmente es preferible diagnosticar a un paciente como FP que como FN en una enfermedad, es decir, se pretende reducir los FN, aumentando el valor de *Recall*.

$$Recall = \frac{TP}{TP + FN} \tag{3.18}$$

3.7.1.5 Specificity

Es el porcentaje de negativos reales ($TN + FP$) que fueron predichos como negativos (TN). Es un cálculo similar al de *Recall* pero esta vez con valores negativos. De nuevo, en el ámbito médico, *Recall* tendría más importancia que *Specificity* por la misma lógica que antes.

$$Specificity = \frac{TN}{TN + FP} \tag{3.19}$$

3.7.1.6 F1 score

Es la media armónica de *Precision* y *Recall*. Si una de estas métricas disminuye, también lo hace *F1 score*, lo cual es perjudicial. En casos como el comentado médico, no resulta conveniente que la *Precision* tenga la misma importancia que *Recall*, por causas que se vienen comentando. Por ello, se recurre al uso de pesos.

$$F1\ score = \frac{2}{\frac{1}{Precision} + \frac{1}{Recall}} = \frac{2 \cdot Precision \cdot\ Recall}{Precision + Recall} \tag{3.20}$$

3.7.1.7 ROC

Del inglés *Receiver Operating Characteristic*. Consiste en representar $Recall$ frente a $(1 - Specificity)$. Estos términos adoptan los nombres de *True Positive Rate* (TPR) y *False Positive Rate* (FPR), respectivamente. Cuando TPR aumenta, lo hace también FPR. El área bajo la curva ROC es conocida como AUC (*Area Under the ROC Curve*), y a mayor valor, mejor modelo se tiene. Esta métrica se suele emplear cuando las clases tienen igual importancia, como por ejemplo en clasificación de monumentos o animales.

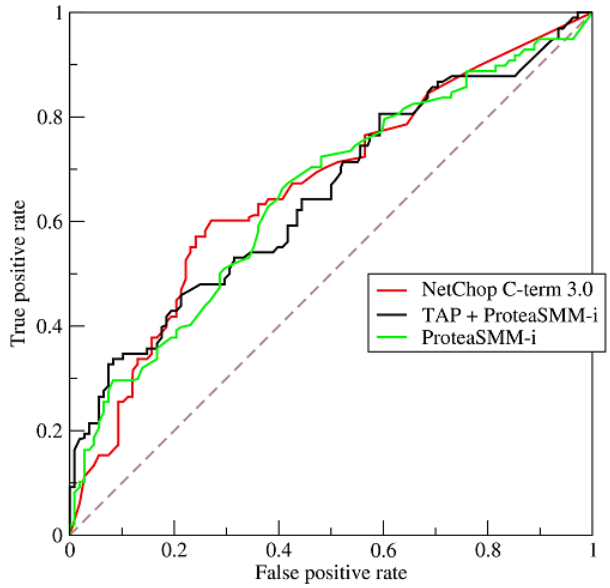


Figura 3-12. Ejemplo de ROC empleando 3 modelos distintos [59]

3.7.2 Optimización de hiperparámetros (*hyperparameter tuning*)

A diferencia de los parámetros de entrenamiento, existe otro concepto de suma importancia, los denominados **hiperparámetros**, que son unos parámetros predefinidos y controlados por el usuario antes de comenzar el entrenamiento y que no varían mientras el modelo entrena. Estos hiperparámetros se asignan al crear el modelo y se modifican tras el análisis de las métricas resultantes de la validación. Determinan el comportamiento de los parámetros y de la red. Unos hiperparámetros comunes serían la profundidad de un árbol de decisión o el número de capas de una red neuronal.

Se intenta buscar la combinación óptima de hiperparámetros que maximice la capacidad predictiva del modelo. A este proceso se le llama optimización de hiperparámetros (*hyperparameter tuning*). Para ello, los hiperparámetros se modifican en función de los resultados de la validación del modelo, y después se vuelve a entrenar el modelo con estos nuevos hiperparámetros, así sucesivamente hasta encontrar una combinación que dé un resultado que cumpla con las expectativas.

También existen dos técnicas para la búsqueda de hiperparámetros de forma automática [60]:

- Búsqueda exhaustiva (*Grid Search*). Hace un entrenamiento completo con todas las combinaciones posibles de hiperparámetros de entre los valores que se le proporcionan. Selecciona aquella combinación que mejor resultado da. Esto supone mucha carga computacional y tiempo, pero encontrará la mejor combinación posible.
- Búsqueda aleatoria (*Random Search*). Prueba combinaciones aleatorias de entre todos los valores que se le proporcionan. Es posible que no se encuentre la óptima, pero es muy probable que se halle una combinación pseudo-óptima, ahorrando mucho tiempo en comparación con la otra técnica.

3.7.3 División de datos

Un paso relevante previo a comenzar el entrenamiento, sobre todo cuando la cantidad de datos es escasa, es el conocido como ***data splitting*** (o *train/test split*). Es la técnica más básica y fácil de implementar, y consiste en dividir el *dataset* original en tres sets, cada uno para distintas partes del proceso, obteniendo así:

- **Set de entrenamiento.** La porción usada para entrenar el modelo. El modelo aprende de estos datos y ajustará los parámetros de entrenamiento en base a ellos.
- **Set de validación.** La porción usada para validar el modelo. En función del resultado de la validación, se modifican los hiperparámetros del modelo.
- **Set de testeo.** La porción usada para simular el comportamiento del modelo en condiciones de trabajo habituales, una vez acabada la validación.

Los porcentajes de división dependen de varios factores, como la cantidad de muestras del *dataset*, si ya se proporciona aparte un set para testeo, si se quiere solucionar el *overfitting*, etc. Generalmente, se dedica un 70% para entrenamiento, un 20% para validación y un 10% para testeo en pequeños *datasets*. Siempre que sea posible, debe maximizarse la cantidad de datos de entrenamiento [61]. El uso de *data splitting* tiene un aspecto negativo: al dividir el *dataset* en tres subsets, necesariamente se reducen las muestras con las que se entrena, y los resultados dependerán de unos sets con datos escogidos aleatoriamente.

Para solucionar este problema, existen otras técnicas creadas a partir de *data splitting* pero con ligeras variaciones, como *Bootstrapping*, *Random Subsampling*, *Leave-One-Out Cross-Validation*... Entre todas ellas destaca *k-fold Cross-Validation*, conocida como validación cruzada de k-subsets. La validación cruzada de k-subsets consiste en dividir el *dataset* resultante de extraer los datos de testeo en *k* subsets (*folds*) y entrenar con *k*-1 de esos subsets, dejando el subset restante para la validación. Este proceso se repite *k* veces, usando cada vez para la validación un subset distinto. La medida final del rendimiento en validación del modelo será el promedio de las medidas de validación de cada iteración [62].

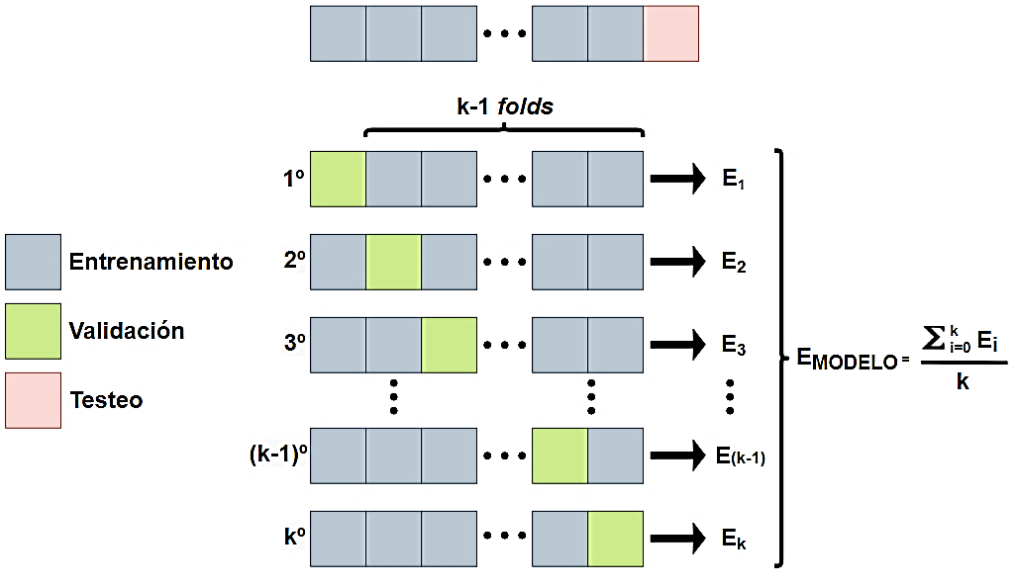


Figura 3-13. Diagrama *k-fold Cross-Validation*

4 REDES NEURONALES ARTIFICIALES

En el capítulo anterior se comentaron algunas de las técnicas de clasificación más conocidas: árboles de decisión, modelos probabilísticos como *Naïve Bayes* o modelos lineales como SVM. Todas ellas tienen el mismo objetivo, que puede sintetizarse en asignar etiquetas a muestras que no tienen gracias a la información proporcionada por otras muestras ya etiquetadas. Para llevar a cabo esta tarea, necesitan disponer de dicha información, es decir, de las *features* o características de las muestras, tanto para entrenar como para validar y predecir, y que además tengan cierto nivel de calidad y concordancia con el ámbito de aplicación. Esto es algo que en la práctica no siempre sucede. Muchas veces los datos no incorporan sus características, por lo cual habría que seleccionarlas y extraerlas manualmente, lo que requiere un gran dominio en Ingeniería de Características (3.4.4.1 y 3.4.4.2). Otras veces, hay datos con propiedades que dificultan su ajuste y empeoran enormemente el rendimiento del modelo, como imprecisiones en las medidas, subjetividad (*biased data*), ruido o falta de integridad... Ante estas adversidades, hay una técnica de clasificación que destaca positivamente: las redes neuronales artificiales. A lo largo de este capítulo se verá su funcionamiento básico enfocado a problemas de clasificación.

4.1 Introducción a las Redes Neuronales

Mejor conocidas como Redes Neuronales Artificiales (RNA) en el ámbito de la IA, son el conjunto de técnicas de Aprendizaje Automático inspiradas en los procesos biológicos del cerebro humano para procesar información. Suelen ser utilizadas para problemas de aprendizaje supervisado, aunque también es factible su uso en modelos no supervisados.

Según Kohonen, uno de los investigadores más reconocidos en el tema, las RNA son redes enormes de elementos simples paralelamente interconectados y con organización jerárquica, con el objetivo de interactuar con los objetos del mundo real de igual forma que nuestro sistema nervioso [63].

Gracias a este gran tamaño y paralelismo, poseen una serie de ventajas como una gran capacidad de generalización y tolerancia a fallos (en su estructura o en datos) e incluso son óptimas para las operaciones de reconocimiento de patrones en tiempo real, actualizando todos los parámetros simultáneamente durante el entrenamiento. Sin embargo, tienen un importante inconveniente: a mayor tamaño de red, mayor cantidad de operaciones realizan para llegar a una conclusión. Es decir, la cantidad de operaciones que tienen lugar en redes grandes es tan elevada que es imposible saber con exactitud lo que sucede en el interior. Conocemos lo que se introduce y lo que sale del proceso, pero solo podemos suponer las actividades ocurriendo en los pasos intermedios, generando inquietud en el sector, pues obliga a confiar ciegamente en estos modelos de ML. A este problema se le denomina caja negra, y de manera similar, se da en nuestro cerebro [64].

En el modelo biológico de las redes neuronales, las neuronas son las células fundamentales del sistema nervioso. Hay millones de ellas organizadas en capas. Al ser excitadas envían unos neurotransmisores, producto de reacciones químicas, a las demás neuronas conectadas a ella, cambiando así sus potenciales eléctricos. Cuando en una de estas otras neuronas el potencial supera un umbral, se activan, es decir, se excitan. El ciclo de excitación continúa hasta que o no se supera el umbral o se llega a la neurona final. De esta forma se consigue conducir impulsos nerviosos hasta el destino a base de comunicaciones entre neuronas.

Análogamente, en las RNA los elementos simples reciben el nombre de **neuronas artificiales**. Son la unidad básica de computación en la red y se agrupan en **capas**. Cada neurona mantiene interconexiones sinápticas con otras neuronas artificiales de capas previas y posteriores. Generalmente, reciben varias señales de entrada de las neuronas anteriores en la red. A cada entrada se le asocia un **peso** (*weight*). En la mayoría de los casos, en cada neurona, la suma de todas sus entradas multiplicadas por sus respectivos pesos se hace pasar por una **función de activación**, que dará la salida de dicha neurona artificial hacia las neuronas de la siguiente capa.

4.1.1 Conceptos básicos de RNA

Se resumen los términos esenciales del ámbito de las Redes Neuronales Artificiales [65] [66]:

- **Neurona artificial.** Conocida como nodo. Por lo general, es un elemento que recibe señales de otras neuronas de capas previas, las computa junto a unos pesos según una regla de propagación (normalmente combinación lineal) y hace pasar el resultado por una función de activación, determinando así su propio estado de excitación. Este estado interno de la neurona, fruto de todas estas operaciones, se envía por la salida a las neuronas de la capa posterior.
- **Capa.** Es el conjunto de neuronas artificiales formado por aquellas neuronas cuyas entradas llegan desde la capa anterior y cuyas salidas se envían a la capa posterior.
- **Peso.** Representa el grado de conexión, la fuerza de unión, entre dos neuronas en diferentes capas. Es un valor numérico que pondera las señales que se envían por una conexión. Puede ser un valor positivo (excitación), negativo (inhibición) o cero (no hay conexión). Los pesos son parámetros de entrenamiento que modifica automáticamente el modelo durante la fase de aprendizaje.
- **Regla de propagación.** Permite hallar, a partir de las entradas a una neurona y los pesos de las conexiones correspondientes, un valor conocido como potencial, que más tarde se procesa con la función de activación. Existen varias reglas, pero la más simple y empleada es la **combinación lineal** de todos los pesos y entradas, junto al *bias*.
- **Función de activación.** O función de transferencia. Se utiliza para obtener la salida de la neurona artificial, que es a su vez su estado de activación, a partir del potencial calculado. Su verdadero propósito es mapear los datos a un rango conveniente (eje y) y añadir no linealidad.
- ***Bias*.** Al potencial se le suma el término de sesgo, *bias*, antes de pasarlo por la función de activación. Pertenece a una neurona previa siempre activa con salida de valor 1 y conexión con peso ajustado automáticamente durante el entrenamiento, lo que permite desplazar la función de activación entera para lograr valores antes inalcanzables, mejorando las predicciones. Solo un nodo *bias* por capa.

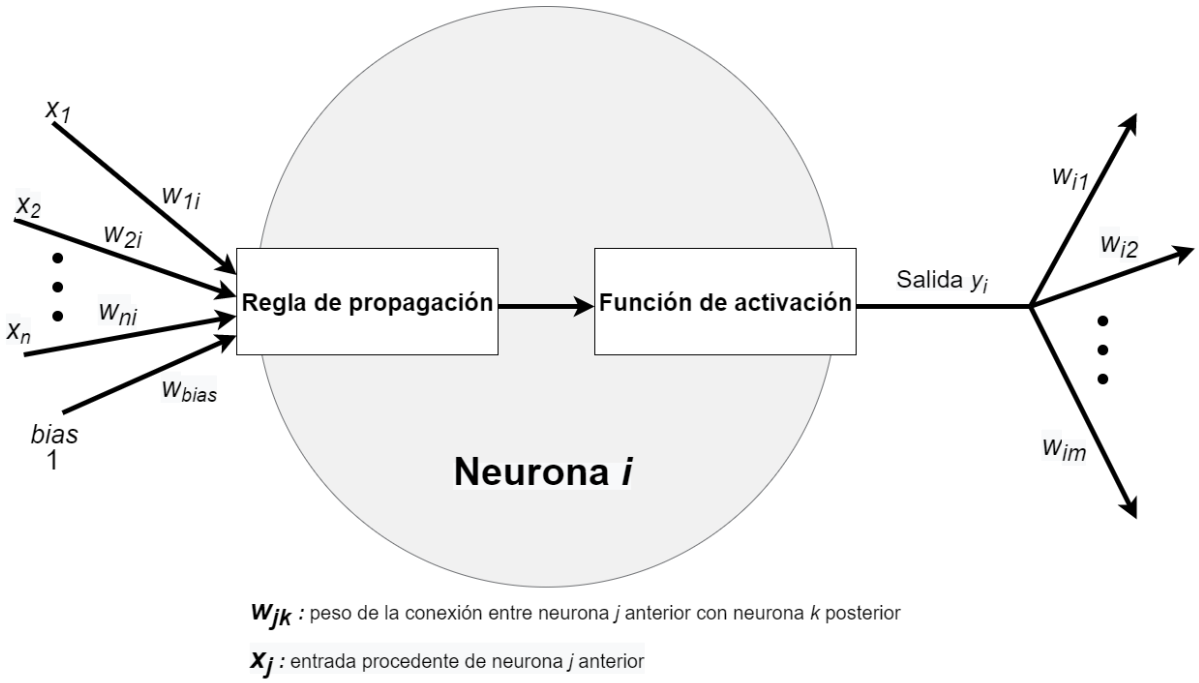


Figura 4-1. Modelo general de una neurona artificial

4.1.2 Estructura de RNA

Todas las Redes Neuronales Artificiales se pueden dividir en tres partes:

- ***Input layer* (capa de entrada).** Es la capa que recibe los datos y después los envía a la primera capa oculta. No realiza ningún cálculo (salvo en ocasiones algún tipo de preprocesado de datos). En realidad, es una manera de definir el tipo, forma y cantidad de los datos que acepta la RNA. Normalmente, no suele tenerse en cuenta para la enumeración de capas totales de una red.
- ***Hidden layers* (capas ocultas).** Una RNA está conformada por una o más capas de neuronas artificiales, dependiendo de la complejidad de los patrones que deben ser aprendidos por el modelo. Se encargan de procesar los datos de entrada, identificando patrones para su posterior clasificación.
- ***Output layer* (capa de salida).** Está situada al final de la red neuronal. Con su salida se determina la etiqueta predicha. En problemas binarios hay una sola neurona en esta capa. En problemas multiclase con una etiqueta y multiclase multi-etiqueta hay un nodo para cada clase.

4.2 Funciones de activación

Como ya se ha mencionado, las funciones de activación son aquellas que se aplican a la combinación lineal de entradas y pesos para determinar el estado de activación de una neurona. Pueden ser binarias o continuas según el valor que envían a la siguiente capa. En el caso de las binarias convierten la combinación lineal a únicamente dos valores que suelen ser {0, 1} o {-1, 1}. En cambio, las funciones de activación continuas mapean los valores a un número cualquiera dentro de un rango predefinido, usualmente [-1, 1]. Dependiendo de su linealidad se observan dos tipos de funciones de activación:

4.2.1 Función lineal

También llamada función identidad. La salida es igual que la entrada, por lo que al usarse como función de activación, el modelo mantendrá la linealidad. En clasificación, solo se usa en problemas con muestras separables linealmente mediante hiperplano.

4.2.2 Funciones no lineales

Son aquellas funciones que añaden no linealidad a la red neuronal, permitiéndole resolver problemas no lineales y por lo tanto mejorando su capacidad de generalización. Su derivada es de gran importancia para el cálculo del algoritmo de Descenso del gradiente en caso de RNA multicapa. Hay una gran variedad de funciones, algunas de las cuales se representan a continuación:

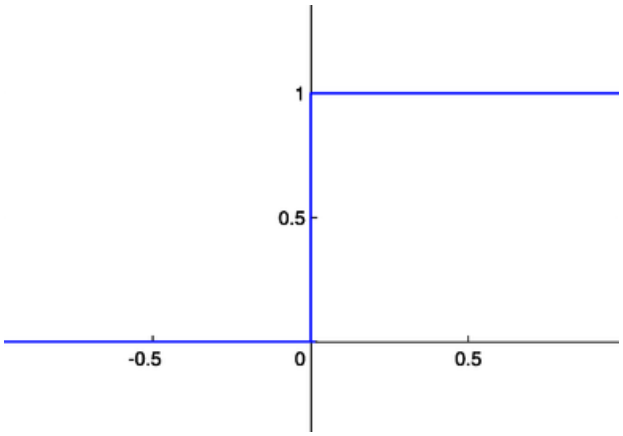


Figura 4-2. Escalón de Heaviside o umbral [67]

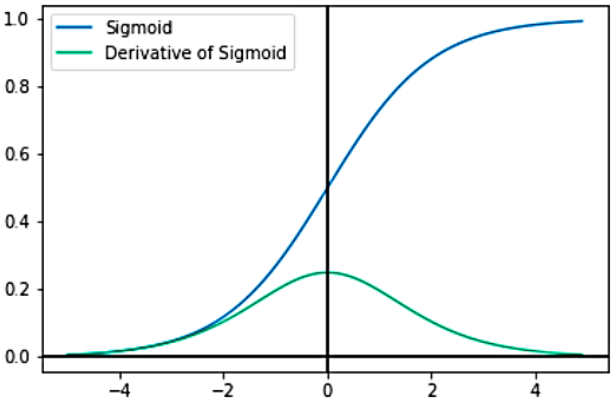


Figura 4-3. Sigmoide y Softmax, y su derivada [68]

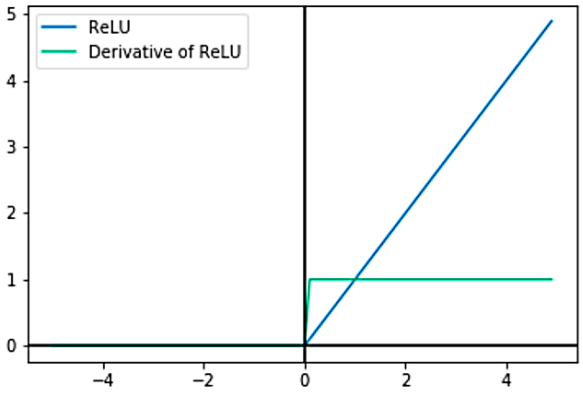


Figura 4-4. ReLU y su derivada [68]

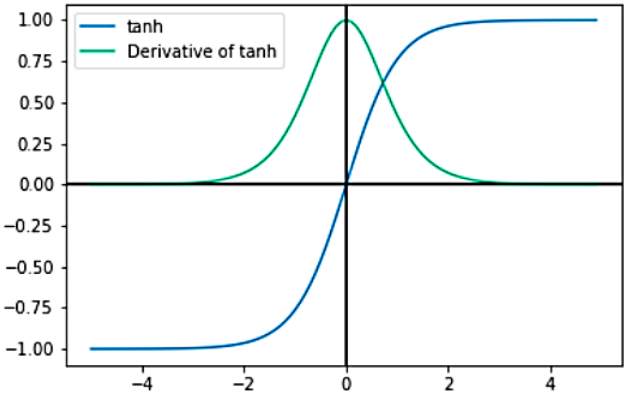


Figura 4-5. Tangente hiperbólica y su derivada [68]

4.2.3 Usos de funciones de activación para clasificación

Si el problema es una clasificación binaria (dos clases mutuamente excluyentes), *output layer* tendrá una única neurona y empleará la función de activación sigmoidal. Si es una RNA monocapa, se puede usar el umbral.

Si el problema es una clasificación multiclase con una etiqueta (más de dos clases mutuamente excluyentes), *output layer* tendrá una neurona por cada clase y usará la función de activación *softmax*. Esta garantiza que la suma de las probabilidades de todas las clases, de todas las neuronas de la *output layer*, sea uno. Se escoge como etiqueta predicha aquella correspondiente con la neurona de mayor valor de salida.

Si el problema es una clasificación multiclase y multi-etiqueta (más de dos clases no excluyentes), *output layer* tendrá una neurona por cada clase y utilizará la función sigmoidal, pues no hay necesidad de que todas las probabilidades sumen uno: las probabilidades de cada clase son independientes entre sí. Esto se da por ejemplo en la detección de elementos en una misma imagen, siendo cada elemento una clase [69]. Se escogen como etiquetas de salida las de aquellas correspondientes a las neuronas con salida mayor a 0.5 (criterio habitual).

4.3 Descenso del gradiente

El descenso del gradiente (*Gradient Descent*, GD) es el proceso iterativo más extendido para optimizar varios algoritmos de Aprendizaje Automático, en particular las Redes Neuronales Artificiales multicapa y profundas. Con optimizar, como se apuntó en el apartado 3.6.1, se hace referencia a minimizar la función de coste elegida, mejorando el rendimiento del modelo al reducir su error.

En RNA, el descenso del gradiente minimiza el error a partir del ajuste de todos los parámetros de la red: los pesos y sesgos. La función de coste estará en función de estos parámetros. Al comenzar el entrenamiento de un modelo, los pesos y sesgos de las conexiones de la red, salvo indicación expresa del usuario, son inicializados con unos valores aleatorios. Esto se traduce en un valor inicial aleatorio de la función de coste, es decir, un punto de la representación multidimensional función de coste-parámetros. Como el objetivo es minimizar el error y no es posible conocer a priori todos los valores de la función de pérdida frente a los parámetros, se opta por calcular la pendiente (derivada de la función de coste) en la posición actual. Al tratarse de una función multidimensional (habitualmente), se calculan las derivadas parciales para cada uno de los pesos, w, y sesgos, b, de la red, consiguiendo la pendiente en el eje de cada parámetro. Si todas estas derivadas parciales se agrupan en un vector, se obtiene el gradiente de la función, $\boldsymbol{\nabla f}$.

$$\boldsymbol{\nabla f} = \left[\frac{\partial coste}{\partial w_1}, \frac{\partial coste}{\partial w_2}, \dots, \frac{\partial coste}{\partial w_N}, \frac{\partial coste}{\partial b_1}, \frac{\partial coste}{\partial b_2}, \dots, \frac{\partial coste}{\partial b_L} \right] \tag{4.1}$$

Cuando una derivada parcial toma un valor grande, urge una modificación del parámetro en cuestión para reducir el error. En cambio, si toma un valor pequeño, significa que el valor actual no perjudica al desempeño de la RNA.

Para calcular cada una de estas derivadas parciales de toda la red, el algoritmo de descenso del gradiente recurre a otro algoritmo conocido como ***backpropagation***, **propagación hacia atrás de errores**. Como su nombre indica, se comienza calculando las derivadas parciales en la *output layer* de la RNA para después ir progresando hacia atrás, hasta la primera *hidden layer*, reutilizando las derivadas parciales ya calculadas y las salidas de las neuronas presinápticas. Este funcionamiento se basa en composición de funciones y regla de la cadena. En la última capa, el coste es función de la salida, la salida es función de la combinación lineal y la combinación lineal es función de los pesos y sesgo con la capa anterior. Puede resumirse en tan solo en cuatro ecuaciones [70]:

$$\delta^L = \frac{\partial coste}{\partial a^L} \cdot \frac{\partial a^L}{\partial z^L} \tag{4.2}$$

$$\delta^{l-1} = \frac{\partial z^l}{\partial a^{l-1}} \cdot \delta^l \cdot \frac{\partial a^{l-1}}{\partial z^{l-1}} \tag{4.3}$$

$$\frac{\partial coste}{\partial w^{l-1}} = \delta^{l-1} \cdot a^{l-2} \tag{4.4}$$

$$\frac{\partial coste}{\partial b^{l-1}} = \delta^{l-1} \tag{4.5}$$

Donde:

- Superíndice *L* representa a la capa de salida de la RNA.
- Superíndice *l* representa a la capa actual, *l-1* a la situada presinápticamente y *l-2* a la anterior a esta.
- a es la salida de la neurona.
- z es la suma ponderada de pesos y sesgos.
- δ es el error de la neurona.

La ecuación (4.2) es el error imputado a la última capa, directamente calculado con la función de coste, y la ecuación (4.3) es el error de las demás capas de la red. Nótese que las funciones de activación que se empleen en las neuronas deben tener derivada distinta de cero o infinito, pues en caso contrario su derivada no permitiría el progreso con este algoritmo de *backpropagation* en las ecuaciones para el cálculo de los errores imputados, (4.2) y (4.3), y por lo tanto no funcionaría el descenso del gradiente, no se ajustarían los pesos ni sesgos, y el modelo tendría una mala capacidad predictiva, en general. Es por esto que no se emplea como función de activación no lineal el escalón de Heaviside en redes neuronales multicapa.

Volviendo a la definición de descenso del gradiente, el vector $\boldsymbol{\nabla f}$ tiene la dirección en la que la función de coste aumenta su valor, esto es, apunta hacia una combinación de parámetros que provocan mayor error. Es justo lo opuesto a nuestros intereses, por lo que lo lógico es utilizar la dirección opuesta: el **negativo del gradiente**. De esta forma, se halla una combinación de pesos y sesgos que dan un menor valor de función de coste que el inicial. Al repetir este proceso sucesivas veces se puede alcanzar una zona con gradiente prácticamente cero, en la que iterar de nuevo no afecta significativamente al resultado. Este nuevo punto, definido por los parámetros, suele tratarse de un mínimo local de la función de coste, aunque puede llegar a ser global, dependiendo del azar en la inicialización (o quizás del conocimiento de un punto inicial más oportuno) y del tipo de función de coste: convexa (un único mínimo) o no convexa (varios mínimos).

También se define otra variable, ***learning rate*** o tasa de aprendizaje, α. Su valor afecta a la velocidad o al tamaño de los pasos con la que se avanza en la dirección del negativo del gradiente. Es preciso hallar un valor óptimo, pues una tasa demasiado baja provoca un excesivo tiempo de aprendizaje, demasiadas iteraciones, pudiendo finalizar en un valor diferente al mínimo local. En cambio, un valor demasiado alto, dificulta considerablemente la convergencia en el mínimo, pudiendo entrar en un bucle infinito [71].

$$w_{i,j+1} = w_{i,j} - \alpha \, \boldsymbol{\nabla f}_j(i) \quad , \quad \forall \, i \in [1, n] \tag{4.6}$$

En la ecuación anterior se muestra como para cada peso *i* se actualiza su valor de la iteración *j* a uno nuevo en la iteración *j+1* gracias a restarle un *learning rate* predefinido y multiplicado por el gradiente calculado en la iteración *j* y evaluado en la posición del peso *i*.

Existen tres variaciones del algoritmo del descenso del gradiente, según la cantidad de muestras de entrenamiento con las que se computa cada iteración [72] [73]:

4.3.1 Batch Gradient Descent

La actualización de los pesos tiene lugar tras entrenar con todos los datos del *dataset* de entrenamiento.

En terminología habitual de RNA, a un período de entrenamiento con todos los datos del *dataset* de entrenamiento se le denomina **época (*epoch*)**. Si se utiliza *Batch* GD como optimizador, al finalizar cada época se actualizarán los pesos, siguiendo las ecuaciones (4.1) y (4.6).

Puede requerir muchísima potencia computacional, sobre todo con *datasets* masivos, pero llega al mínimo en pocas iteraciones y es muy robusto.

4.3.2 Mini-batch Gradient Descent

Es uno de los optimizadores más empleados en la actualidad. La actualización de los pesos se produce tras entrenar con *n* datos indicados por el usuario y extraídos del *dataset* de entrenamiento. Este valor de *n* recibe el nombre de ***batch size*** **(tamaño de lote)**, un término de gran importancia en las redes neuronales. Según la aplicación, *batch size* es un valor comprendido entre 30 y 500 muestras. Este algoritmo es, computacionalmente hablando, más eficiente que *Batch Gradient Descent*.

Además, al período de entrenamiento con *batch size* datos se le denomina ***step*** **(paso)**. De esta forma, si se emplea *Mini-batch* GD como optimizador, tras cada *step* se calcula el gradiente de la función de coste correspondiente a *batch size* muestras y se actualizan los pesos.

La relación entre *epochs* y *steps* se muestra a continuación. El número de épocas es elegido por el usuario.

$$steps\ per\ epoch = \frac{nº\ datos\ entrenamiento}{batch\ size} \tag{4.7}$$

4.3.3 Stochastic Gradient Descent

Con siglas SGD. Consiste en actualizar el valor de los pesos tras entrenar con una única muestra del *dataset* de entrenamiento. Es decir, es una particularización de *Mini-batch Gradient Descent* con un *batch size* de 1. Varios autores suelen emplear los términos SGD y *Mini-batch Gradient Descent* indistintamente.

Requiere una aleatorización del *dataset* de entrenamiento para modificar el orden en el que se actualizan los pesos. Como la actualización de los pesos se hace con cada muestra, el valor de dichos pesos y de la función de coste variará mucho de una iteración a otra del algoritmo. Esta fluctuación permite hallar mejores mínimos locales pero dificulta la convergencia en el mínimo exacto, pues el camino que sigue es muy ruidoso, necesitando un mayor número de iteraciones. A pesar de esto, es el algoritmo de GD que requiere menor potencia computacional y es el más eficiente.

Aparte de estas tres variaciones directas del algoritmo del descenso del gradiente, existen otro conjunto de algoritmos bastante más complejos, que además implementan otros métodos que solucionan problemas de convergencia y fluctuaciones comunes de GD, como *momentum*, *Nesterov accelerated gradient* o momentos. Algunos de estos algoritmos son Adam, RMSprop, Adadelta, AMSGrad, Nadam, etc. [72]

4.4 Redes Neuronales Unidireccionales

Las topologías existentes de RNA pueden clasificarse en función de si las conexiones sinápticas entre nodos forman o no un bucle. Estas estructuras se le conocen como redes neuronales recurrentes y redes neuronales unidireccionales, respectivamente. A continuación, se explican con mayor detalle las redes unidireccionales.

Del inglés *feed-forward neural networks*, las redes neuronales unidireccionales o alimentadas hacia delante son aquellas RNA en las que las conexiones entre nodos no forman un ciclo o bucle. La información se mueve solo en una dirección: desde *input layer* hasta *output layer*. A su vez, se dividen en tres subclases según el número de capas ocultas que conforman la red:

4.4.1 Redes neuronales monocapa. Perceptrón monocapa

El perceptrón de Rosenblatt, también conocido como *Single-Layer Perceptron*, es considerado la red neuronal unidireccional más simple posible. Es un tipo de red neuronal de una sola capa (red monocapa), siendo esta la *output layer* (*input layer* no se enumera). En ocasiones, las RNA monocapa y los perceptrones monocapa son referidos indistintamente aunque realmente son términos diferentes. En un perceptrón las entradas, pesos y el término *bias* son combinados linealmente para luego compararse siempre con un umbral. Si esta combinación lineal es mayor que 1, la salida del nodo es 1, o sea, está excitado. En caso contrario, es 0, no excitado. Es un clasificador lineal. Con *m* nodos de salida un perceptrón monocapa puede clasificar muestras en *m+1* clases.

En otras palabras, un perceptrón monocapa puede considerarse como una RNA monocapa que utiliza como función de activación la función escalón de Heaviside y por lo tanto tiene salidas binarias. La principal limitación de las redes monocapa es que solo pueden discriminar entre conjuntos linealmente separables mediante hiperplanos en el espacio de características, sea cual sea la función de activación elegida.

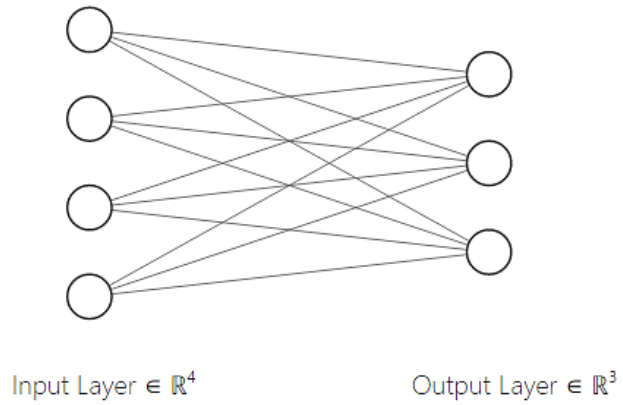


Figura 4-6. Ejemplo de red neuronal monocapa

4.4.2 Redes neuronales multicapa. Perceptrón multicapa

Conocido como *Multi-Layer Perceptron* (MLP), es el término usado ambiguamente en la literatura para referirse a cualquier RNA multicapa unidireccional (con cualquier función de activación). Está formado por una *input layer*, una o varias *hidden layers* y una *output layer*. Puede confundirse con una red multicapa de perceptrones (escalón de Heaviside), que no es factible pues dicha función de activación no funcionaría con *backpropagation,* que necesita un gradiente no nulo ni infinito para el cálculo del descenso del gradiente.

Si se entiende MLP como cualquier RNA unidireccional multicapa, implica que puede emplearse cualquiera de las funciones de activación en las neuronas, tanto lineales como no lineales. Si se utilizan no lineales, el modelo será capaz de clasificar conjuntos no linealmente separables, aumentando su no linealidad a medida que se elevan los nodos o capas con funciones de activación no lineales.

En cambio, si absolutamente todas las funciones de activación son lineales, no se tendría dicha capacidad discriminatoria no lineal. Todas las capas lineales actuarían podrían simplificarse como una única capa de salida lineal. No lineal implica que no puede ser obtenido a partir de una combinación lineal de las entradas, por lo tanto, no importa cuántas capas lineales se utilicen, el resultado nunca será no lineal [74]. Si se quiere resolver un problema lineal, se debe recurrir a un clasificador lineal como *Single-Layer Perceptron*, que no usa *backpropagation* y además requiere menos parámetros (más eficiente).

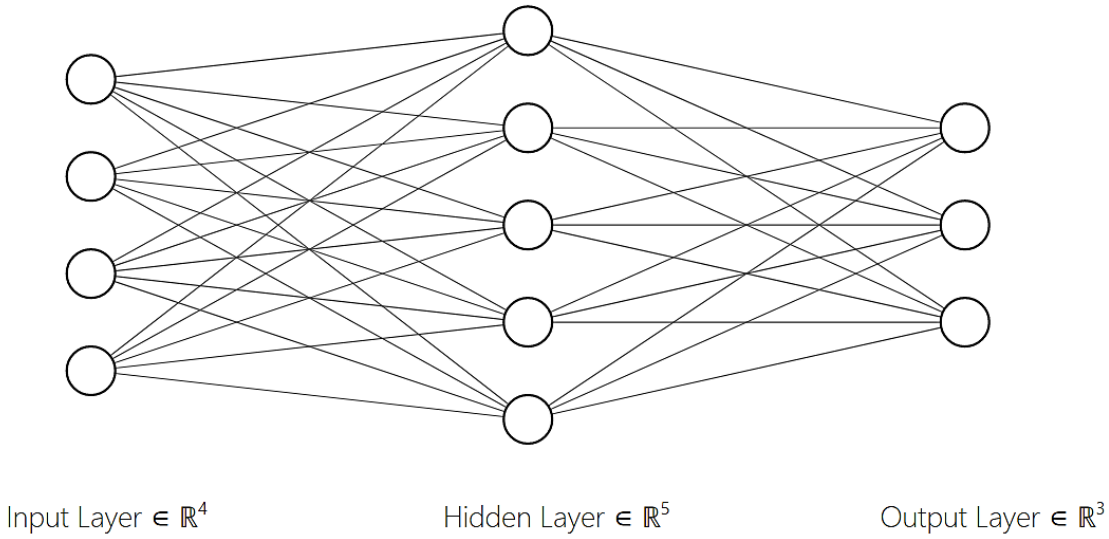


Figura 4-7. Ejemplo de red neuronal multicapa

4.4.3 Red neuronal profunda

Cuando se emplean un número considerable de capas ocultas en una red neuronal multicapa y se manejan una enorme cantidad de datos y relaciones no lineales complejas, se dice que el modelo es una Red Neuronal Profunda (RNP), y se le asocia el término Aprendizaje Profundo o ***Deep Learning***.

Este umbral que separa una red neuronal multicapa superficial (*shallow*) de una profunda (*deep*) no ha sido acordado universalmente por los especialistas en el campo, aunque la mayoría considera que con solo dos capas ocultas ya se trataría de una RNP. Otras fuentes señalan que con tres capas, y además se originan otros términos como "muy profunda" (más de dieciséis capas ocultas) o "extremadamente profunda" (de cincuenta a más de mil capas) [75].

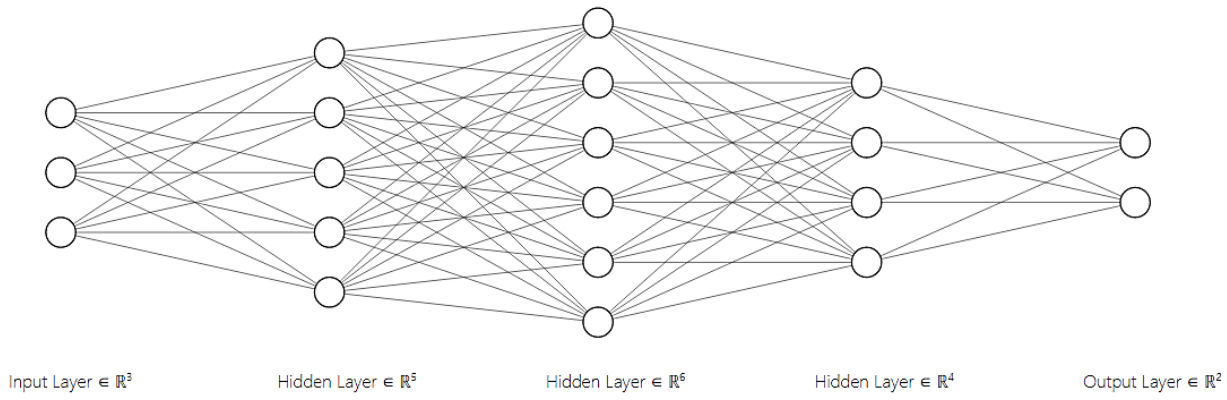


Figura 4-8. Ejemplo de red neuronal profunda

4.5 Redes Neuronales Convolucionales

Una Red Neuronal Convolucional, o *Convolutional Neural Network* (CNN, ConvNet), es un tipo de RNA unidireccional profunda destinada principalmente al campo del reconocimiento de patrones en texto, **imágenes** o vídeos para su clasificación, objetos de estudio de la Visión Artificial. También es posible su uso en audio y en reconocimiento de voz. La aplicación de CNN es recomendada solo en aquellas situaciones donde son clave las propiedades derivadas de la estructura espacial de los datos. Los datos deben redimensionarse adecuadamente a un array sobre el que se opera con convoluciones. Su principal fortaleza es la selección y extracción automática de características, siendo estas invariantes a la posición en la imagen.

Su estructura está inspirada en la organización de la corteza visual humana. Esta dispone de dos tipos de neuronas: simples y compuestas. Ambas se activan cuando aparece un estímulo visual en su campo de visión (capo receptivo). Pero cada una lo hace ante diferentes subconjuntos de estímulos, característica que se conoce como sintonía neural. Las capas de neuronas simples tienen una sintonía simple, por lo que se activan cuando ven líneas orientadas de cierta manera en su pequeño campo receptivo. En cambio, las capas de neuronas complejas agrupan a varias simples, por lo tanto, se activan con cualquier línea en un campo receptivo mayor. El cerebro humano al poseer gran cantidad de ambas capas, las va alternando, consiguiendo imágenes más completas e identificando patrones más complejos como ojos, boca, nariz... hasta llegar a un nivel de abstracción mayor, como una cara [76]. De este modo, las CNN proceden de una forma similar a las RNA multicapa, pero añadiendo además mecanismos basados en el funcionamiento biológico de la corteza visual.

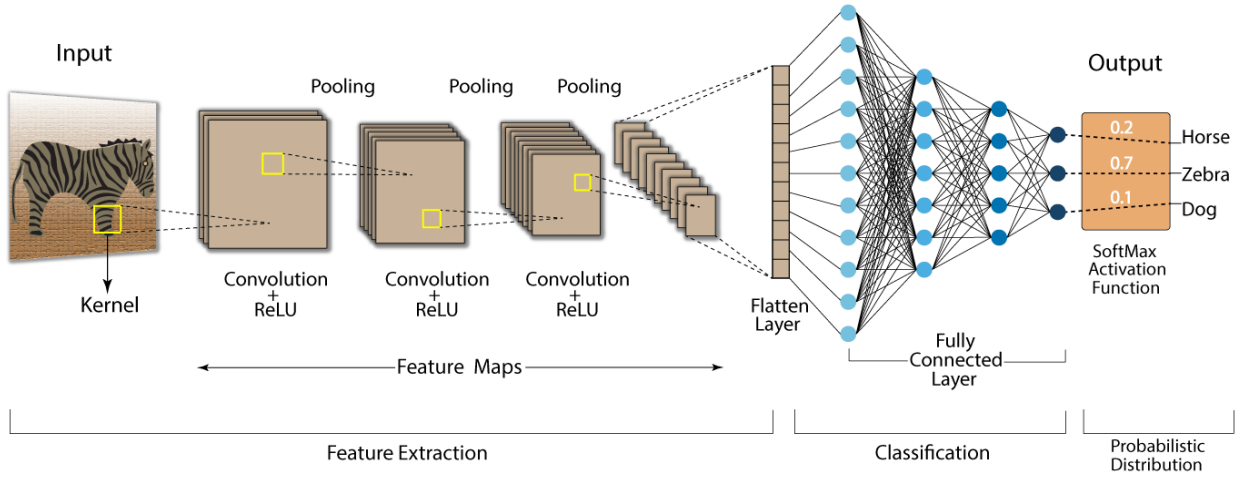


Figura 4-9. Ejemplo de una CNN [77]

En primer lugar, se hacen pasar imágenes a la CNN por su capa de entrada. Esta capa debe tener un número de neuronas igual al número de píxeles de la imagen. Cuando las imágenes digitales están en escala de grises, es decir, en blanco, negro o cualquier gris, son matrices bidimensionales, por lo que su tamaño en píxeles se hallaría con tan solo multiplicar su anchura por su altura. En cambio, si las imágenes están a color significa que son combinación de tres planos de color (RGB: rojo, verde y azul), o sea, matrices tridimensionales, por lo que su tamaño en píxeles sería tres veces mayor que en el caso anterior. Normalmente, los píxeles de cada plano de las imágenes toman valores desde 0 (negro) hasta 255 (blanco). Para reducir la memoria ocupada y optimizar el proceso se suelen convertir al rango [0, 1] mediante un escalado Min-Max, como se vio en 3.4.3.2.

El siguiente paso es extraerles a dichas imágenes las características que permitan su clasificación. En CNN, al conjunto de capas que se encargan de esta tarea se le conoce como ***backbone*** y consta de dos tipos de capas: las capas convolucionales y las capas de agrupación.

Las capas convolucionales, como su nombre indica, realizan convoluciones sobre imágenes. De esta manera, a partir de una imagen se obtienen varias imágenes con diferentes propiedades provechosas para el análisis de la original, como pueden ser sus bordes, texturas, patrones, etc. Al utilizar varias capas convolucionales consecutivas, el número de imágenes que se manejan aumenta considerablemente a la par que se mejora la información del sistema sobre la imagen que inicialmente se insertó por la capa de entrada. Esto recuerda al mencionado funcionamiento de la corteza visual humana: las primeras capas convolucionales aprenden características simples, de bajo nivel, como líneas, y a medida que se va avanzando por las capas ocultas, la

sofisticación de las características va en aumento, hasta que las últimas capas convolucionales ya son capaces de aprender objetos enteros.

Para llevar a cabo las convoluciones, cada capa emplea un conjunto de núcleos o ***kernels*** que reciben el nombre de filtros. Cada *kernel* es una matriz más pequeña que la imagen de entrada, normalmente de tamaño 3x3, 5x5 o 7x7, que se va aplicando sobre los píxeles de la imagen hasta recorrerla en su totalidad, para no perder información espacial relativa al posicionamiento de los píxeles. Al resultado final tras aplicar un *kernel* se le llama **mapa de características**. Finalmente, el mapa de características se hace pasar por una función de activación de las mencionadas en MLP, aunque normalmente se elige la rectificadora ReLU.

La cantidad de píxeles totales entre todos los mapas de características salientes de una capa convolucional será masiva, lo que se traduce en muchas neuronas. Si se volvieran a pasar todos estos píxeles por la siguiente capa convolucional directamente, y después por otra, y otra... repitiendo este proceso varias veces, la cantidad de neuronas requeridas sería una cifra disparatada, y reduciría la eficiencia del modelo. Debido a esto, se opta por colocar algunas capas de agrupación detrás de cada capa o conjuntos de capas de convolución.

Las capas de agrupación se encargan de reducir el tamaño de las imágenes procedentes de las convoluciones con los filtros, manteniendo las características relevantes detectadas. Hay varias operaciones posibles para llevar a cabo este submuestreo, pero normalmente se emplea el llamado ***max-pooling***. Básicamente, esta técnica consiste en elegir un tamaño, por ejemplo 2x2, que simboliza cuánto se reducirán las imágenes tras el submuestreo, en este caso, las dimensiones a la mitad y la cantidad de píxeles a una cuarta parte. Se recorre cada imagen de izquierda a derecha y de arriba a abajo, y se agrupan píxeles según el tamaño de *max-pooling* elegido, en el ejemplo serían bloques 2x2. De cada bloque se selecciona el valor más alto, y se recoloca en una nueva matriz en la posición correspondiente [78]. Así se reduce el número de parámetros, la dimensionalidad, el tiempo necesario para procesar... y evita *overfitting*. Lógicamente, hay un límite de número de capas de agrupación que se pueden aplicar para no perder toda la información previamente obtenida por reducir demasiado la cantidad de píxeles.

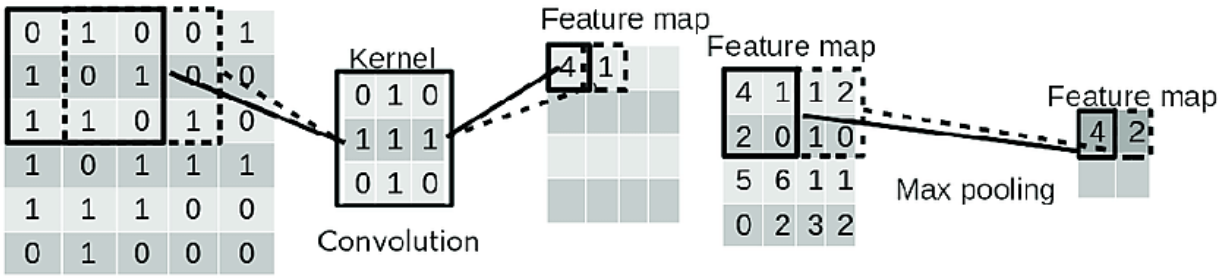


Figura 4-10. Ejemplo de convolución y *max-pooling* [79]

Se puede apreciar en la Figura 4-9 que a medida que se avanza por el *backbone*, la cantidad de imágenes va aumentando considerablemente (debido a las capas convolucionales) pero estas cada vez tienen menos resolución, menos tamaño (por las capas de agrupación).

Una vez se han recorrido todas las capas ocultas pertenecientes al *backbone*, los datos resultantes deben aplanarse (conocido como *flattening*), es decir, convertirse en un vector de características 1-D. La razón de esta transformación se debe a que el siguiente conjunto de capas de la CNN, que recibe el nombre de ***head***, forman un clasificador y, como se ha visto anteriormente, el vector de características es necesario para la representación de la muestra (imagen en este caso) en el espacio de características. El clasificador es igual a una RNA clásica, es decir, capas con neuronas totalmente conectadas a las capas que las rodean y que desembocan en una capa de salida, normalmente con función de activación *softmax*, que devuelve la etiqueta asignada.

Al igual que en MLP, mediante el algoritmo del descenso del gradiente y *backpropagation* es posible actualizar eficientemente los valores de los parámetros durante el entrenamiento de la red para disminuir el valor de la función de coste. En las CNN estos parámetros se corresponden con los valores de los píxeles de cada *kernel* (*backbone*) y con los pesos de las conexiones en la RNA multicapa (*head*), además de los sesgos. De esta manera, la red extrae automáticamente las características más beneficiosas, las que disminuyen el error, para una mejor clasificación, evitando un preprocesado manual mayor de las imágenes de muestra.

5 VISIÓN ARTIFICIAL

En este capítulo se describen brevemente las etapas de la Visión Artificial y su relación con el ML, pues el problema a resolver deriva directamente de estos dos campos.

5.1 Introducción a la Visión Artificial

Debido a la sequía de ideas que hubo en las áreas de Inteligencia Artificial y Aprendizaje Automático desde mediados de los ochenta hasta mitad de los noventa, años en los que menguó el vertiginoso progreso al que estos campos estaban habituados, los científicos se vieron en la necesidad de centrar esfuerzos en otras direcciones, provocando el gran desarrollo de subramas ya existentes por aquel entonces, como la robótica, el procesamiento del lenguaje natural o la Visión Artificial.

La Visión por Computadora, por su nombre en inglés *Computer Vision*, también conocida como Visión Artificial (VA) o Visión Automática, se define como la rama de la Inteligencia Artificial que permite dotar a una máquina de la capacidad de ver e interpretar una escena para, finalmente, actuar en consecuencia [80]. Es un intento de simular la percepción visual humana mediante modelado matemático, generación de algoritmos y programas, que darán lugar a un ordenador capaz de inferir acerca de imágenes o vídeos por sí solo [81] y que puede transferir esta habilidad a otros sistemas electrónicos para que realicen acciones o hagan recomendaciones útiles en base a la información de entrada [82]. Actualmente, tiene infinidad de aplicaciones y prácticamente se emplea en todos los campos existentes, incluso sin haber liberado aún todo su potencial debido a limitaciones computacionales.

5.2 Etapas de la Visión Artificial

Atendiendo a esta descripción, tradicionalmente se definen 5 fases o etapas dentro del proceso, resumidas a continuación. Estas no siempre se siguen de manera secuencial pues en caso de fallo se recurre a una realimentación [83]:

1. **Adquisición.** Consiste en la captura y digitalización de los escenarios a analizar mediante algún tipo de sensor, normalmente una cámara, o directamente la descarga desde un *data source*. Es un paso crucial, puesto que la cantidad y calidad de información visual que se obtenga será la base para el resto del programa y determinará el alcance y rendimiento del mismo. Aquellas escenas que se quieren clasificar con una misma etiqueta (*label*) conformarán una clase. La adquisición de imágenes trata diversos temas, como la óptica, la teoría del color o la electrónica, y dependiendo del alcance del problema, la luz de la escena, el color, la tecnología del sensor, modelo de cámara e incluso el tipo de digitalización son factores a considerar e intentar optimizar. Acto seguido, el set de imágenes (*dataset*) se envía a un ordenador en el que será procesado.

2. **Procesamiento previo.** Se manipulan las imágenes digitales con el fin de adaptarlas a la aplicación y facilitar las etapas posteriores. Suelen emplearse filtros y transformaciones para eliminar desperfectos (por ejemplo, el ruido producido por los dispositivos electrónicos durante la adquisición) o para mejorar la calidad de la imagen, resaltar algunas zonas o generar efectos, como rotaciones, reescalado, recortes o cambios de tono, saturación o brillo, entre otros [84].

3. **Segmentación.** Se lleva a cabo para dividir cada imagen del *dataset* en zonas de interés con atributos similares, llamadas regiones, consiguiendo así disminuir la cantidad de información del *dataset*, eliminando partes irrelevantes [85], y por lo tanto, reduciendo la carga computacional. Al mismo tiempo, la representación de la imagen tendrá mayor significado y será más fácil de analizar. Normalmente, el resultado será una nueva imagen en la que solo se observan las siluetas o los bordes de los objetos de provecho. Existen numerosas técnicas de segmentación divididas en dos grandes grupos: los orientados a regiones y los orientados a bordes [86]. Algunas de las más conocidas son la separación por histograma (umbralización), el algoritmo de *watershed*, las basadas en contornos obtenidos con gradientes o similares u otros métodos morfológicos que utilizan erosión y acreción... Entre ellas también se encuentra la mencionada técnica de Aprendizaje no supervisado para Modelos de agrupamiento particionales, *k-means clustering*, que aplicada a imágenes, recibe como *input data* los píxeles de una imagen y los agrupa en función de su similitud. Al final, cada uno de los *k clusters* se corresponde con una región.

Figura 5-1. Segmentación de imagen con *k-means clustering*, k=3 [87]

4. **Extracción de características.** Las características (*features*) son las propiedades que describen una escena (o parte de ella) y que ayudan a reconocerla en cualquier otra imagen de la misma, aunque sea tomada desde diferente distancia, ángulo, condiciones de iluminación, oclusión, etc. [88]. Las características más comunes van desde el simple color de una escena, el perímetro de un contorno, el área de una región o su número de huecos hasta las conocidas invariantes de Hu (invariantes a cambios de escala, rotaciones y traslaciones), pasando por toda clase de momentos. En los algoritmos más tradicionales de Visión Artificial, la elección de unas u otras características de una escena es objeto de estudio por parte del usuario y dependerá de la naturaleza de las imágenes y del rendimiento deseado, primando siempre la calidad de las características por encima de la cantidad, puesto que se pretende evitar cargas computacionales desmesuradas (Reducción de dimensionalidad) [89]. En el caso de emplearse algoritmos basados en aprendizaje profundo y redes neuronales, no se requiere ningún análisis manual: el proceso de selección y extracción de características de una imagen se realiza de forma automática, sin asistencia externa, lo que supone una gran ventaja respecto a sus alternativas [90]. Al finalizar esta fase, cada imagen tendrá su propio descriptor.

5. **Reconocimiento (clasificación) e interpretación de información.** En la etapa final, cada imagen será clasificada en una clase gracias a su descriptor. Los vectores de características de las imágenes ya etiquetadas del *dataset* se utilizarán como referencia (ajuste de pesos) para decretar a qué clase pertenecen las demás imágenes sin clasificar. Existen diversos métodos (clasificadores) según los criterios que siguen a la hora de clasificar. Como se vio en 3.5.1.1.1, algunos se basan en el concepto de distancia entre los vectores de características, otros en medidas de probabilidad, en árboles de decisión o los que utilizan redes neuronales. Finalmente, el ordenador será capaz de asignar a cada imagen una etiqueta, con mayor o menor precisión de acierto según las decisiones elegidas durante el proceso.

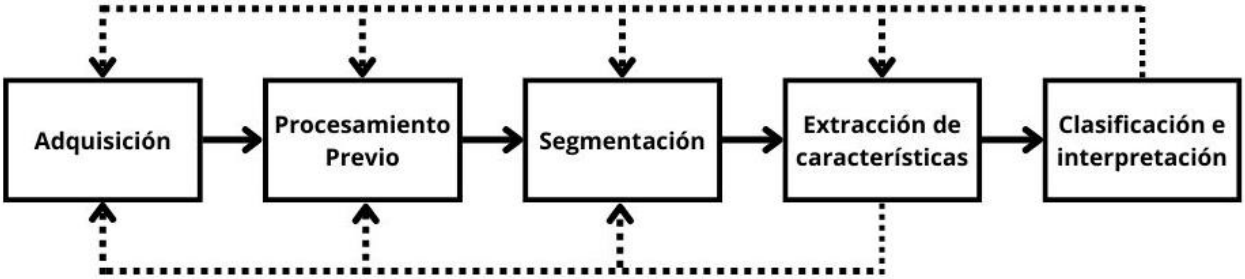


Figura 5-2. Diagrama de bloques de las etapas de un sistema de Visión Artificial y realimentación

Se observa la estrecha relación entre las etapas de la Visión Artificial y las fases del proceso del Aprendizaje Automático. La mayoría de las etapas de VA emplean técnicas propias del ML. Esta similitud se debe a que, a pesar de que sea considerada una rama directa de la IA, la Visión Artificial realmente deriva del ML. No es posible un sistema de VA competente sin técnicas de ML, pero sí una aplicación de ML sin VA (cualquiera que no requiera de información visual).

Las redes neuronales artificiales, en especial las convolucionales, son fundamentales para el desarrollo de la Visión Artificial. Le aportan una versatilidad, robustez y precisión imposible de alcanzar con otras técnicas, permitiendo reconocer texto, elementos complejos con formas variantes, situaciones en diferentes condiciones de iluminación, fondos cambiantes, movimientos faciales o corporales...

6 SOLUCIONES PROPUESTAS AL PROBLEMA

En esta sección se proponen varias soluciones al problema de reconocimiento de lugares emblemáticos. Se pretende elaborar modelos basados en técnicas de ML para la clasificación, en concreto, CNN.

Si bien la práctica más común en el sector es utilizar modelos de RNA pre-entrenados como punto de partida sobre los que se van realizando ajustes menores (*Transfer Learning*), aquí se ha decidido también crear un modelo desde cero, o según la jerga de los especialistas en el tema, ***create a model from scratch***. La razón de esta decisión ha sido principalmente la de poner en práctica la teoría aprendida en las secciones anteriores de este trabajo y conocer de primera mano al menos una pequeña parte de la labor que hay detrás de los modelos más exitosos en la actualidad. Se sabe con anticipación que los resultados no van a estar al nivel de estos modelos, pues se necesitarían muchos más recursos computacionales, tiempo, conocimientos, experiencia y trabajo en equipo para lograr resultados tan competitivos.

Primero, se explicará cada una de las partes del código de la CNN diseñada desde cero. Después, se implementarán algunos modelos con *Transfer Learning*. Finalmente, se compararán los resultados obtenidos.

6.1 Descripción del problema

El problema se ha extraído de la popular página web Kaggle. Es una comunidad en línea orientada a la Ciencia de Datos y Aprendizaje Automático, en la que usuarios pueden participar en competiciones de ML con premios publicadas por empresas, utilizar un banco de trabajo para construir, entrenar y evaluar modelos, compartir y descargar *datasets* y modelos, e incluso hay tutoriales para iniciarse en el vasto mundo de la IA.

Concretamente, el problema es propuesto por Google, de quien Kaggle es subsidiaria desde 2017. El enunciado se corresponde a una competición con el nombre *Google Landmark Recognition 2021* [91], en la que participaron 525 competidores y 383 equipos. El objetivo es crear un modelo capaz de predecir las etiquetas de lugares emblemáticos, ya sean de origen natural o artificial. Se proporcionan dos sets de datos, uno para el entrenamiento y otro para el testeo. Los resultados se evalúan con una métrica indicada, y el tiempo de ejecución del archivo debe ser menor a 12 horas. Se pueden emplear modelos pre-entrenados. El modelo final también debe ser capaz de discernir entre imágenes de lugares emblemáticos e imágenes sin ellos, por lo que se eleva el nivel de dificultad. También se avisa de que habrá un gran número de clases y que el número de muestras por clase puede ser pequeño, algo que resulta bastante desafiante y que complica el diseño del código.

Para la solución propuesta del problema únicamente se utilizará como referencia dicho enunciado, no se seguirá a rajatabla, ni tampoco se pretende competir con los otros usuarios.

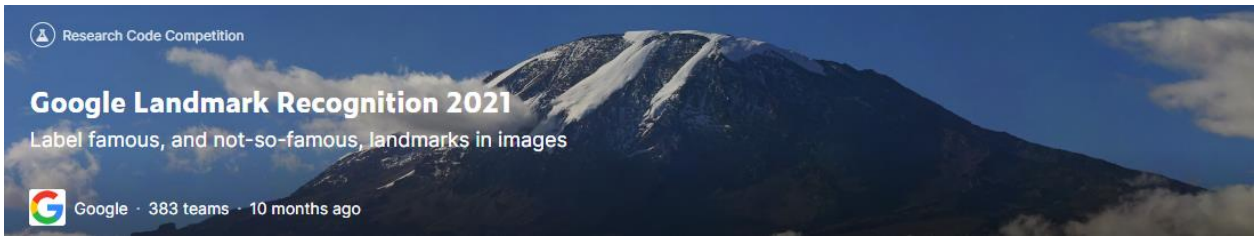


Figura 6-1. Competición *Google Landmark Recognition 2021* [91]

6.2 Entorno de trabajo

Se disponen de varias alternativas de entornos de trabajo en los que desarrollar el código. Las opciones más factibles son realizarlo en un ordenador personal o en versiones gratuitas de entornos de programación en línea como Kaggle o Google Colab. A continuación, se comparan algunas de sus características:

	Portátil MSI GE73VR 7VRE Raider	**Kaggle**	**Google Colab**
RAM	16 GB	13 GB	12 GB
CPU	Intel(R) Core (TM) i7-7700HQ @ 2.80GHz x8	2.30GHz x4	2.30GHz x2
GPU	NVIDIA GeForce GTX 1060	NVIDIA Tesla P100	NVIDIA Tesla K80
Horas máximas por ejecución	Sin límite	12	12
Horas de uso por semana	Sin límite	De 30 a 40	84 (máximo 12 cada 24 horas)

Tabla 6-1. Comparación de posibles entornos de trabajo

Además de estas características, deben tenerse en cuenta otros varios factores. Los entornos en línea pueden ejecutar el código con los recursos del ordenador desde el que se accede, o bien pueden utilizar unos recursos propios reservados al usuario por la plataforma. Aunque estos tengan un límite de uso en una misma sesión y a la semana, pueden ejecutarse aun con el navegador web cerrado, es decir, no necesitan que el ordenador esté encendido durante la ejecución del código, lo que supone una gran ventaja sobre el ordenador personal pues un entrenamiento puede prolongarse hasta varias horas, con el coste y desgaste que eso implica. Además, al usar los entornos en línea no hay que configurar el ordenador ni preocuparse por descargar todas las librerías que se vayan a emplear, basta con importarlas directamente. Asimismo, todas las versiones de un código y sus ejecuciones quedan guardadas en la nube y pueden ser consultadas en cualquier momento, evitando posibles pérdidas.

Ya se han dado suficientes razones para decantarse por los entornos en línea. La cuestión ahora es por cuál de ellos. Hay un detalle crucial para decidirse: el *dataset* que se va a emplear, de tamaño 100 GB, se encuentra originalmente en Kaggle. Si se quisiera enviar a Google Colab sería necesario descargarlo y añadir más líneas de código para incorporarlo. Por esta razón y por la comodidad que supone, se ha decidido utilizar la plataforma Kaggle para el desarrollo del código.

Figura 6-2. Logo de la plataforma Kaggle [91]

Para el entrenamiento se utilizará la GPU de Kaggle en lugar de la CPU. Las CPUs consumen menos energía y funcionan mejor en cálculos secuenciales. En cambio, las GPUs rinden mejor en operaciones que conllevan paralelismo gracias a sus habilidades de procesamiento y gran ancho de banda, pensadas inicialmente para problemas de computación gráfica como mapeo de texturas, iluminación, proyecciones.... Es decir, fueron creadas para el renderizado 3D de escenas, algo que está estrechamente relacionado con el *Deep Learning*, pues ambos se basan en cálculos con matrices y vectores (álgebra lineal) [92]. Por esta razón, la GPU es ideal para el problema a tratar, en el que se manejan imágenes (matrices) y CNN (convoluciones, descenso del gradiente, *backpropagation*...), reduciendo el tiempo de procesado y mejorando la eficiencia del modelo.

6.3 Lenguaje de programación, bibliotecas y módulos

Kaggle ofrece la posibilidad de programar en los dos lenguajes más usados para problemas de esta índole: R y Python. El código se realizará en lenguaje Python dados los conocimientos previos que se poseen, la simpleza y claridad para su lectura y escritura y la gran variedad de bibliotecas que incorpora.

Figura 6-3. Logo del lenguaje de programación Python [93]

Para la correcta implementación del código se importan bibliotecas completas o algunos módulos que son de utilidad mediante las líneas del Código 6-1. A continuación, se resumen sus funcionalidades:

- Biblioteca ***NumPy***. Se utiliza principalmente para el manejo de vectores, matrices y arrays multidimensionales, por lo que es esencial para problemas de álgebra lineal, y además incorpora otras muchas operaciones matemáticas.

- Biblioteca ***Pandas***. Es una herramienta para la adecuación y análisis de *datasets*, que proporciona estructuras y métodos para la lectura, manipulación y escritura de información.

- Módulo ***os***. Se utiliza para leer o escribir archivos y manipular rutas del sistema.

- Biblioteca ***TensorFlow***. Es un software de código abierto basado en Python y desarrollado por Google. Se centra en la creación de modelos de diversas técnicas de ML, y sobresale en Redes Neuronales y *Deep Learning* gracias a otra biblioteca que incorpora, llamada ***Keras***. Puede ser ejecutado en varias CPUs o GPUs (solo NVIDIA). También incluye diversos modelos pre-entrenados para ser importados.

- Biblioteca ***Matplotlib***. Es la herramienta para visualizar y representar gráficamente datos.

- Biblioteca ***cv2***. Contiene funciones para resolver problemas de Visión Artificial.

- Biblioteca ***Scikit-learn***. Es una alternativa a Tensorflow. Ofrece distintos métodos orientados al Aprendizaje Automático en general.

- Módulo ***sys***. Proporciona información adicional sobre parámetros del sistema.

- Módulo ***gc***. Es el recolector de basura de Python, por lo que se encarga de controlar y liberar memoria, aliviar los recursos empleados y optimizar el código.

- Módulo ***time***. Se utiliza para obtener información sobre el tiempo y duraciones de los procesos.

- Biblioteca ***Seaborn***. Contiene métodos para crear gráficas estadísticas más vistosas e informativas. Está basada en la biblioteca Matplotlib.

```
import numpy as np

import pandas as pd

import os

import tensorflow as tf
import tensorflow.keras as keras
from tensorflow.keras import layers
from tensorflow.keras.layers.experimental import preprocessing
from tensorflow.keras.callbacks import EarlyStopping
from tensorflow.keras.callbacks import ReduceLROnPlateau
from tensorflow.keras.callbacks import ModelCheckpoint
from tensorflow.keras import regularizers

import matplotlib.pyplot as plt

import cv2

from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix

import sys

import gc

import time

import seaborn as sns
```

Código 6-1. Importaciones de bibliotecas y módulos

6.4 Análisis del set de datos

El set de datos de entrenamiento proporcionado por los organizadores de la competición es una versión más reducida del set *Google Landmarks Dataset v2* (GLDv2). Este originalmente contenía cinco millones de imágenes etiquetadas, a color y con variadas resoluciones, de lugares emblemáticos tanto construidos por el humano como naturales.

El enunciado dice los siguiente. Las imágenes del *dataset* reducido se ponen a disposición de los participantes en el directorio `train/`. Las etiquetas correspondientes a cada imagen se entregan en el archivo `train.csv`. Por otro lado, hay un *dataset* para el testeo, elaborado específicamente para la competición, que puede contener o no dichos lugares emblemáticos. Este se encuentra en el directorio `test/`. Todas las imágenes tienen un `id` único. Como hay un gran número de imágenes, cada imagen se encuentra dentro de otras tres carpetas, según los primeros tres caracteres del `id` de cada imagen. También se proporciona otro archivo, `sample_submission.csv`, que contiene el `id` de cada imagen de testeo y que habrá que rellenar con la clase predicha y confianza que se tiene en la predicción para cada una. Se indica que hay 1580470 imágenes de entrenamiento divididas entre 203092 clases, y 10345 imágenes de testeo. Toda la estructura de directorios mencionada ocupa 105.52 GB [91].

Los archivos .csv son una forma de datos estructurados (ver 3.3.1), dispuestos de forma tabular. Cada fila representa a una muestra y tiene un identificador, en este caso en la columna `id`. Sin embargo, los archivos .csv de este problema no incorporan características o *features* con los que realizar la clasificación. Inicialmente, el *dataset* de entrenamiento tan solo tiene la columna con las etiquetas que indican el lugar emblemático en cuestión, es decir, la *output variable*. Esta columna recibe el nombre de `landmark_id`, y todas las etiquetas en ella son valores numéricos discretos.

	id	landmark_id
0	17660ef415d37059	1
1	92b6290d571448f6	1
2	cd41bf948edc0340	1
3	fb09f1e98c6d2f70	1
4	25c9dfc7ea69838d	7
...	...	...
1580465	72c3b1c367e3d559	203092
1580466	7a6a2d9ea92684a6	203092
1580467	9401fad4c497e1f9	203092
1580468	aacc960c9a228b5f	203092
1580469	d9e338c530dca106	203092

1580470 rows × 2 columns

Figura 6-4. *Dataset* de entrenamiento inicial

Con el *dataset* tal y como se nos entrega, sería imposible entrenar un modelo, pues la columna a partir de la que se extraería la información de la muestra no define ningún rasgo que discrimine a las clases para una RNA ni hay imágenes que introducir en una CNN.

Es preciso añadir las imágenes de entrenamiento en una columna nueva de su *dataset*, y en la fila correspondiente a su `id` y `landmark_id`, para que al introducirlas en la Red Neuronal Convolucional se extraigan las *features* de cada imagen y después se haga la clasificación en base a estas. Gracias a que el `id` de cada muestra permite encontrar a su imagen correspondiente en la estructura de directorios (según se indicó en el enunciado), basta con crear la siguiente función, que será llamada varias veces a lo largo del código:

```
# Carga y Ajuste del dataset de entrenamiento
def load_traindf():

    traindf = pd.read_csv('../input/landmark-recognition-2021/train.csv')

    #Añadir a train.csv la columna con la dirección de cada imagen para su posterior lectura
    traindf['img_path'] = (traindf['id'].apply(lambda r: os.path.join
                           ('../input/landmark-recognition-2021/train', r[0], r[1], r[2], r + '.jpg')))

    #Conversión de columna landmark_id de int64 a int32 para reducir consumo de memoria
    traindf['landmark_id'] = traindf['landmark_id'].apply(lambda x: np.int32(x))

    return traindf
```

Código 6-2. Función para carga y ajuste del *dataset* de entrenamiento

Con el Código 6-2 se consigue almacenar el archivo .csv de entrenamiento en la variable `traindf`, que es de tipo `Dataframe`, un tipo de tabla con filas y columnas o un array bidimensional de la biblioteca *Pandas*. Después se le añade una nueva columna llamada `img_path`, que se rellena con el camino desde el directorio de trabajo hasta el directorio en el que se encuentra la imagen correspondiente, gracias a los tres primeros valores de su `id`. Posteriormente, la columna con las etiquetas se convierte de `int64` a `int32` para reducir la memoria ocupada durante la ejecución del código, sin perder información. A lo largo del código, se van a observar varias formas de optimización, no por nada, sino porque al manejar grandes cantidades de información, el kernel prestado por Kaggle se colapsa cuando llega al límite de memoria RAM, cerrando la sesión y perdiendo todo el progreso. Al llamar a la función con el Código 6-3 y mostrar el resultado, la salida obtenida es la Figura 6-5:

```
traindf = load_traindf()        #Función previamente definida para cargar el dataset ajustado

landmark_unique = len(traindf['landmark_id'].unique())    #Clases totales del dataset

traindf
```

Código 6-3. Asignación del *dataset* de entrenamiento a una variable como Dataframe

	id	landmark_id	img_path
0	17660ef415d37059	1	../input/landmark-recognition-2021/train/1/7/6...
1	92b6290d571448f6	1	../input/landmark-recognition-2021/train/9/2/b...
2	cd41bf948edc0340	1	../input/landmark-recognition-2021/train/c/d/4...
3	fb09f1e98c6d2f70	1	../input/landmark-recognition-2021/train/f/b/0...
4	25c9dfc7ea69838d	7	../input/landmark-recognition-2021/train/2/5/c...
...	...	...	...
1580465	72c3b1c367e3d559	203092	../input/landmark-recognition-2021/train/7/2/c...
1580466	7a6a2d9ea92684a6	203092	../input/landmark-recognition-2021/train/7/a/6...
1580467	9401fad4c497e1f9	203092	../input/landmark-recognition-2021/train/9/4/0...
1580468	aacc960c9a228b5f	203092	../input/landmark-recognition-2021/train/a/a/c...
1580469	d9e338c530dca106	203092	../input/landmark-recognition-2021/train/d/9/e...

1580470 rows × 3 columns

Figura 6-5. Dataframe con datos de entrenamiento y nueva columna con *paths*

Como se puede observar, se ha decidido no incorporar las imágenes al Dataframe aún y mantener únicamente sus caminos, pues primero se pretenden realizar algunas operaciones previas que no requieren del uso de las fotografías y solo ralentizarían el procesamiento, al tratarse de una gran cantidad de matrices tridimensionales enormes. Es posible visualizar las imágenes a partir de la columna `img_path` añadida. Se muestra el código para su representación (Código 6-4). Se imprime también la **resolución** en píxeles de cada imagen. No hay uniformidad en sus tamaños, lo que supone un problema a resolver porque, para poder ser introducidas por la capa de entrada de la CNN, todas las matrices deben tener las mismas dimensiones.

```
plt.figure(figsize=(25,7))

for i in range(12):
    j = np.random.randint(0, traindf.shape[0])
    img = plt.imread((traindf['img_path'][j]))
    plt.subplot(2 , 6, i+1)
    plt.xticks([])
    plt.yticks([])
    plt.xlabel(str(img.shape[0])+'x'+str(img.shape[1]),
               fontweight = "bold", fontsize=16)      #Resolución
    plt.imshow(img)

plt.show()
```

Código 6-4. Representación de imágenes de entrenamiento sin ajustar

Figura 6-6. Imágenes aleatorias sin ajustar y sus resoluciones

Para ampliar los conocimientos acerca del *dataset* de entrenamiento basta con imprimir por pantalla información obtenida directamente de la estructura del Dataframe y de la *output variable*.

```
print('Datos del dataset de entrenamiento original \n')
print('Número de imágenes en el dataset: ', traindf.shape[0])
print('Número de clases diferentes: ', landmark_unique)
print('Clase más baja: ', min(traindf['landmark_id']))
print('Clase más alta: ', max(traindf['landmark_id']))
print('\nRepeticiones de elementos por clase:')
print(traindf['landmark_id'].value_counts())
```

```
Datos del dataset de entrenamiento original

Número de imágenes en el dataset:  1580470
Número de clases diferentes:  81313
Clase más baja:  1
Clase más alta:  203092

Repeticiones de elementos por clase:
138982    6272
126637    2231
20409     1758
83144     1741
113209    1135
          ...
84677        2
36989        2
133688       2
17316        2
111405       2
Name: landmark_id, Length: 81313, dtype: int64
```

Código 6-5. Información adicional del *dataset* de entrenamiento

Se puede comprobar como el número de imágenes coincide con el indicado en el enunciado. Sin embargo, existen 81313 clases diferentes en el *dataset* distribuidas entre la clase 1 y la clase 203092. De aquí se puede deducir que hay números de clases no usados dentro de este rango. Por otro lado, el número de representaciones por clase toma valores muy dispares: mientras que existen varias clases con únicamente 2 muestras, hay otras con más de mil, incluso una llegando a 6272.

Esto es un problema que debe solucionarse antes del entrenamiento, pues puede generar resultados engañosos, como alta precisión cuando en realidad todas las clases predichas correctamente solo pertenecen a las clases más representadas. A este contratiempo se le conoce como ***Imbalanced Data*** o datos desbalanceados.

Si se representan las ocurrencias de cada clase (`landmark_id`), se puede apreciar este efecto a simple vista. A continuación, se muestra dicha gráfica pero con un tamaño de barra de diez clases por cuestiones de visibilidad.

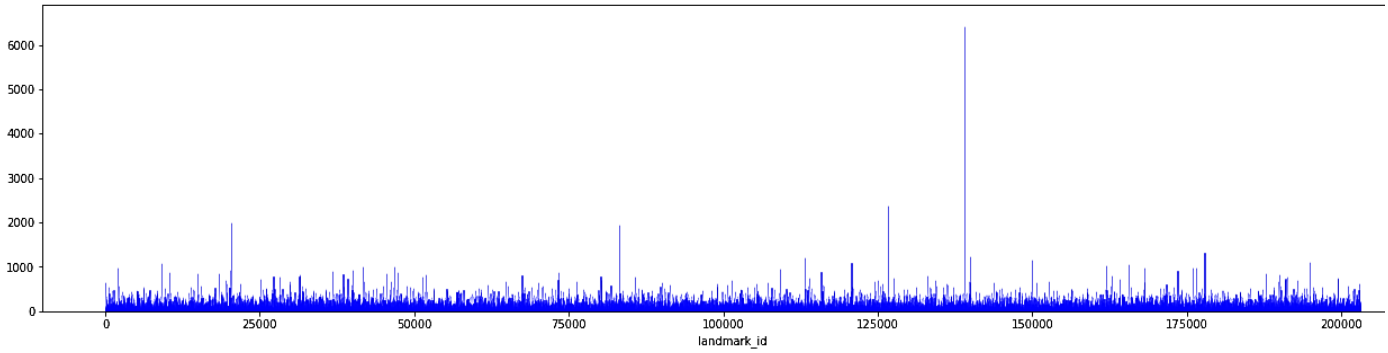


Figura 6-7. Distribución de imágenes cada 10 clases

6.5 Adecuación del set de datos

Una vez se han observado los datos proporcionados, deben ajustarse antes de insertarlos en el entrenamiento. El *dataset* se entrega ya limpiado, es decir, todas las filas y columnas del Dataframe tienen valores correctos asignados, sin duplicados ni huecos vacíos. Esto facilita el preprocesamiento de los datos. Las principales cuestiones surgidas durante el análisis que deben tratarse son: la incorporación de las propias imágenes al Dataframe, igualar las resoluciones y solventar el *Imbalanced Data*.

6.5.1 Obtención de muestra

Debido a la limitación de los recursos disponibles, es necesario reducir el tamaño del *dataset* que se nos entrega, o sea, recortar su tamaño, seleccionar una muestra a partir de la población. Para ello, se asigna a otra variable cierta cantidad de imágenes del set original. Esta cantidad puede venir dada por una variable global indicada por el usuario que simbolice la cifra exacta de imágenes que tendrá la muestra. Una alternativa a esta idea podría ser la creación de dos variables globales, también dadas por el usuario, para establecer una clase mínima y una clase máxima que denoten el rango de `landmark_id` en el que se extraen las imágenes para la muestra. Finalmente, se elige esta última propuesta, garantizando que se seleccionan todas las muestras posibles de las clases indicadas, sin dejar ninguna suelta, ya que sería realmente problemático para clases en las que escasean representantes (por ejemplo, coger una muestra de una clase que tiene dos). De esta forma se rebaja el alcance del problema y se reducen los requisitos computacionales necesarios.

Para materializar esta idea en el código basta con definir las variables y hacer una indexación condicional del Dataframe, como se muestra en Código 6-6.

```
CLASE_MIN = 0
CLASE_MAX = 50

#Obtención de muestra traindf_s
traindf_s = traindf[(CLASE_MIN<traindf['landmark_id']) & (traindf['landmark_id']<=CLASE_MAX)]
```

Código 6-6. Creación de la muestra a partir del *dataset* original

	id	landmark_id	img_path
0	17660ef415d37059	1	../input/landmark-recognition-2021/train/1/7/6...
1	92b6290d571448f6	1	../input/landmark-recognition-2021/train/9/2/b...
2	cd41bf948edc0340	1	../input/landmark-recognition-2021/train/c/d/4...
3	fb09f1e98c6d2f70	1	../input/landmark-recognition-2021/train/f/b/0...
4	25c9dfc7ea69838d	7	../input/landmark-recognition-2021/train/2/5/c...
...	...	...	...
764	7f8b41b6ffa2b57d	50	../input/landmark-recognition-2021/train/7/f/8...
765	905376c48b3627b3	50	../input/landmark-recognition-2021/train/9/0/5...
766	a980c74fe95c383d	50	../input/landmark-recognition-2021/train/a/9/8...
767	dac710c9437ba94d	50	../input/landmark-recognition-2021/train/d/a/c...
768	fc1ab8fad00b6eff	50	../input/landmark-recognition-2021/train/f/c/1...

769 rows × 3 columns

Figura 6-8. Dataframe de la muestra

`CLASE_MIN` y `CLASE_MAX` definen el rango de clases empleadas. Para agilizar la explicación y el proceso, mantendrán los valores mostrados en el Código 6-6, hasta que en el apartado final se prueben distintos valores para evaluar el rendimiento del modelo. De la misma manera se procederá con el resto de las variables.

Como apunte, la diferencia entre estas dos variables no será igual al número de clases disponibles en la muestra, pues como se mencionó antes, hay clases que no se usan. Para ver el número real de clases y la cantidad de datos en la muestra se utiliza el siguiente código. La variable `clases` es un vector con el nombre de las clases utilizadas en la muestra, y será de gran utilidad en las próximas partes del código.

```
N_DATOS = len(traindf_s['landmark_id'])
clases = traindf_s['landmark_id'].unique()
N_CLASES = len(clases)

print('Número de datos de la muestra, N_DATOS: '+str(N_DATOS))
print('\nNúmero de clases de la muestra, N_CLASES: '+str(N_CLASES))
print('\nClase más baja de la muestra: '+str(min(traindf_s['landmark_id'])))
print('\nClase más alta de la muestra: '+str(max(traindf_s['landmark_id'])))
print('\nRepeticiones de elementos por clase en la muestra tras Undersampling:')
print(traindf_s['landmark_id'].value_counts())
```

```
Número de datos de la muestra, N_DATOS: 769

Número de clases de la muestra, N_CLASES: 20

Clase más baja de la muestra: 1

Clase más alta de la muestra: 50

Repeticiones de elementos por clase en la muestra tras Undersampling:
27    504
43     44
37     26
12     25
11     22
32     20
23     18
9      16
41     16
22     14
50     12
48      9
24      9
7       8
30      6
29      5
34      5
1       4
36      3
17      3
Name: landmark_id, dtype: int64
```

Código 6-7. Información adicional sobre la muestra

6.5.2 *Undersampling*

Una de las formas de combatir el *Imbalance Data* es el submuestreo, más conocido como ***Undersampling***. Consiste en ecualizar la distribución de clases de un *dataset* mediante la eliminación de muestras de las clases más frecuentes. Si bien es cierto que con este método se pierde información, no tiene tanta importancia al tratarse de clases con más que suficientes instancias. Además del *Undersampling*, existen otras técnicas para balancear los datos, pero que se han considerado no convenientes para la situación actual. Estos a los que se hacen referencia son la recolección de más datos en **otras fuentes** y el sobremuestreo, u ***Oversampling***. El primero consiste en añadir muestras de otros *datasets*, ampliando así los datos en nuestra mano. Automáticamente se descarta dado su laborioso proceso: buscar una fuente con datos coincidentes, limpiarlos, adecuarlos, incluso etiquetarla manualmente correctamente, mezclarla con el *dataset* actual, etc. La segunda opción, el *Oversampling*, es el procedimiento opuesto al *Undersampling*, es decir, aumenta el número de muestras de las clases menos frecuentes mediante diversos algoritmos, ecualizando la distribución de clases. Puesto que las clases más frecuentes tienen un elevado número de datos, al aumentar las menos representadas hasta tales cifras provocaría una cantidad excesiva de imágenes que ralentizaría el preprocesado y entrenamiento, pudiendo colapsar la memoria. Puesto que ambas técnicas mencionadas aumentan los datos sobre los que se parte, se ha desechado su uso por ahora. Primero se aplicará el *Undersampling*, reduciendo la cantidad máxima de muestras de las clases, y se continuará con el preprocesado de datos. Más adelante sí que se aplicará un *Oversampling* de los datos, elevando las clases menos frecuentes hasta un nivel menor que el establecido por el *Undersampling*.

Para llevar a cabo el submuestreo, se ha diseñado el Código 6-8 que hace uso de otra variable indicada por el usuario, llamada `UMBRAL_UNDERSAMPLING`. Se opera con la muestra extraída previamente. Primero se ordenan aleatoriamente las instancias dentro de cada clase, y después se eliminan muestras de las clases cuya ocurrencia sobrepasa el umbral. Nótese la utilización de la variable `SEED`, la semilla que garantiza la reproducibilidad de procesos de naturaleza aleatoria del modelo en diferentes ejecuciones. Se fija en el inicio del programa a 42 para todo el proceso y su valor se puede modificar en caso de que ese azar fijado no dé buenos resultados.

```
UMBRAL_UNDERSAMPLING = 80

#Undersampling en la muestra
traindf_s = (traindf_s.groupby('landmark_id', group_keys=False).
             apply(lambda x: x.sample(n = min(len(x), UMBRAL_UNDERSAMPLING), random_state= SEED)))
traindf_s.reset_index(inplace=True, drop=True)  #Reiniciar índices. drop=True índices antiguos no sean nueva columna
traindf_s['landmark_id'].value_counts()            #Frecuencias de las clases tras undersampling
```

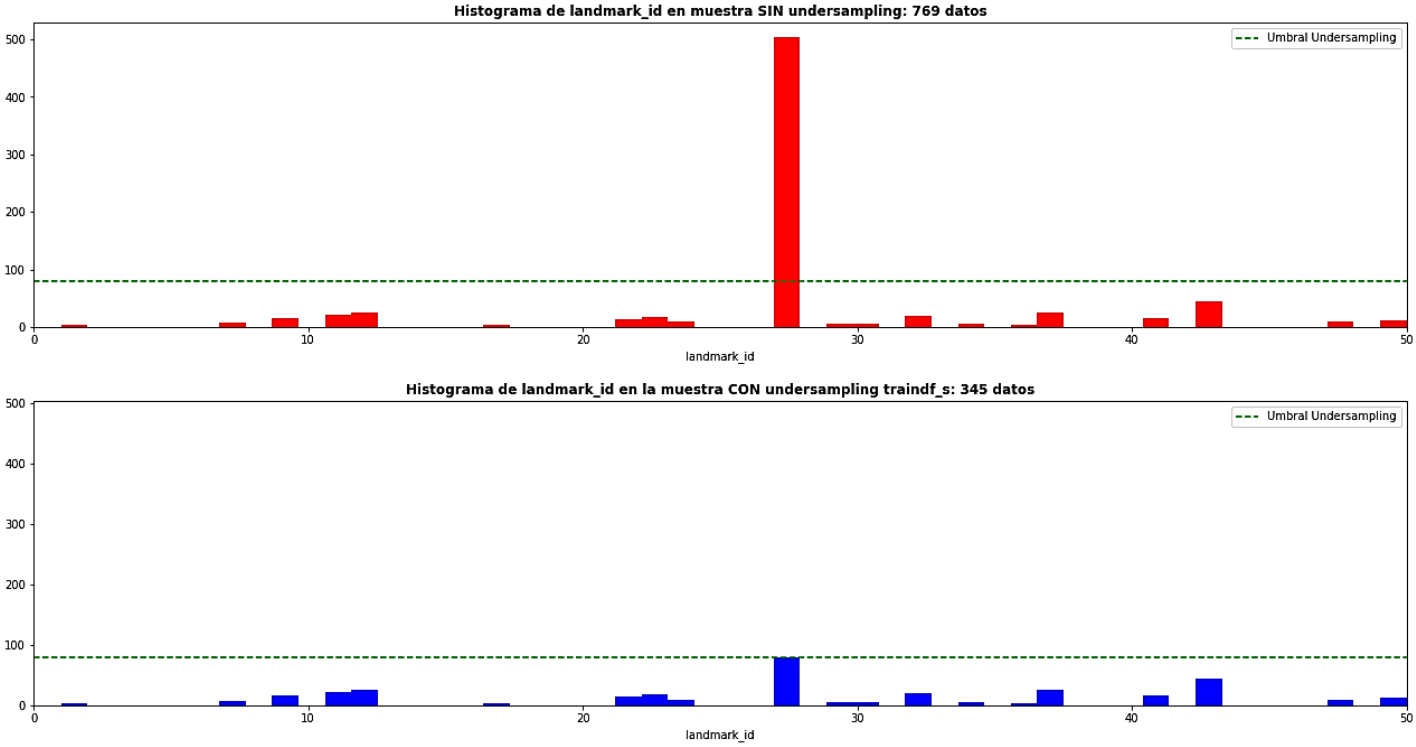
Código 6-8. *Undersampling* en la muestra

A la muestra con *Undersampling* se le aplica después el Código 6-7 previo, obteniendo:

```
Número de datos de la muestra, N_DATOS: 345

Número de clases de la muestra, N_CLASES: 20

Clase más baja de la muestra: 1

Clase más alta de la muestra: 50

Repeticiones de elementos por clase en la muestra tras Undersampling:
27    80
43    44
37    26
12    25
11    22
32    20
23    18
9     16
41    16
22    14
50    12
48     9
24     9
7      8
30     6
29     5
34     5
1      4
36     3
17     3
Name: landmark_id, dtype: int64
```

Figura 6-9. Información adicional sobre muestra tras *Undersampling*

Al comparar la información adicional de la muestra con su información adicional tras el *Undersampling*, se aprecia que la cantidad de datos ha disminuido considerablemente, de 769 a 345, es decir, se han reducido las muestras a menos de la mitad. Si se analizan las clases, se puede ver como únicamente la clase con más representación, la 27, se ha reducido de 504 hasta el valor de `UMBRAL_UNDERSAMPLING` muestras, fijado en 80 inicialmente para no eliminar información en exceso pero tampoco mantener demasiadas imágenes de una única clase. Si se hubiera realizado en su lugar un *Oversampling*, todas las clases de la muestra tendrían 504 instancias, unas cantidad exorbitante de imágenes para tan pocas clases y recursos. A continuación, se representan las ocurrencias de cada clase de la muestra, el antes y el después del *Undersampling*, en forma de histograma, y dibujando el umbral para una mayor interpretabilidad. Se puede observar que se ha conseguido equilibrar en



cierta medida las frecuencias de las clases a costa de disminuir los datos.

Figura 6-10. Comparación de distribución de clases en muestra: antes y después de *Undersampling*

6.5.3 Redimensionamiento de las imágenes y almacenamiento

Las imágenes proporcionadas para la competición tienen diferente resolución, como se vio en la Figura 6-6. Todas son matrices tridimensionales pero con distintas anchuras y alturas. Esto es muy normal en cualquier *dataset* de imágenes para la clasificación. La forma más habitual de actuar ante estos casos consiste en redimensionar todas las imágenes al mismo tamaño. La razón del por qué no se dejan tal y como están inicialmente reside en la estructura de la Red Neuronal Convolucional. La capa de entrada de la CNN debe tener las mismas dimensiones que las imágenes y esta no se puede modificar durante el entrenamiento. La resolución común debe establecerse antes de crear la CNN para poder asignarle el mismo número de neuronas a la capa de entrada. Normalmente se escoge una resolución menor que la de la imagen de menor dimensión en el momento inicial, pues es preferible perder información a inventar nuevos píxeles que puedan dañar la capacidad de generalización del modelo.

En el programa se ha implementado una función (Código 6-9) que, a partir del camino a una imagen en la estructura de directorios, es capaz de leerla y redimensionarla según la variable global `IMG_SIZE`, cuyo valor es indicado por el usuario. Se ha decidido que las imágenes redimensionadas serán cuadradas, o sea, con relación de aspecto 1:1, de tamaño `IMG_SIZE` x `IMG_SIZE`, a pesar de las posibles deformaciones. Lógicamente, a mayor resolución, mayor es el procesamiento que realiza el programa, aumentando el tiempo total de ejecución. Esta variable tomará inicialmente el valor de 128 píxeles.

```python
# Lectura y Redimensionamiento imágenes a partir de su ruta (path)
def img_read_resize(img_path):
    img = plt.imread(img_path)
    img_redim = cv2.resize(img,(IMG_SIZE,IMG_SIZE))
    return img_redim
```

Código 6-9. Función para leer y redimensionar imágenes

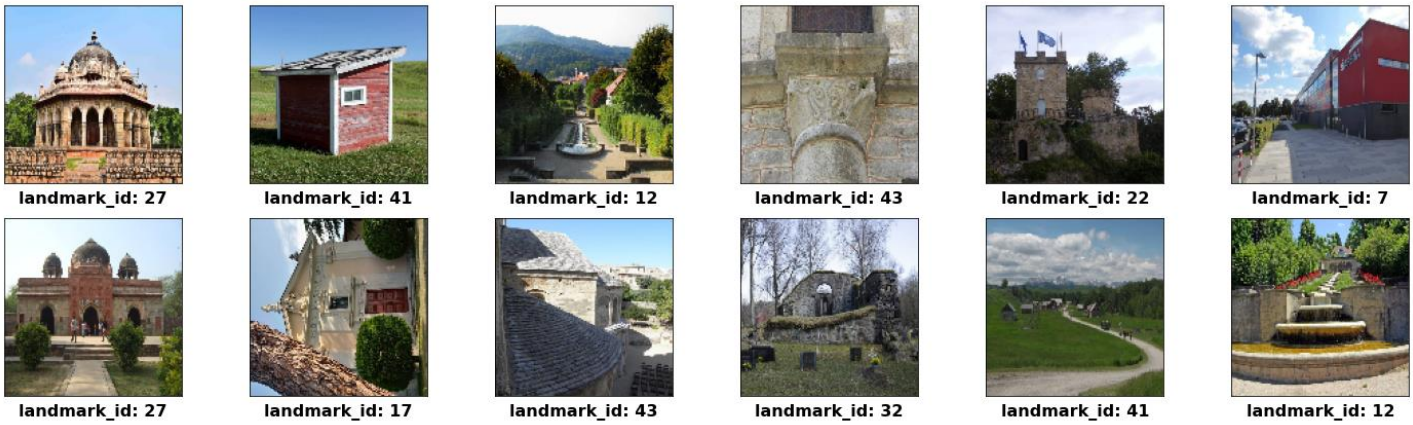


Figura 6-11. Imágenes aleatorias de la muestra redimensionadas a 128x128 píxeles

Normalmente, al crear un modelo de RNA de aprendizaje supervisado, se crean en un principio dos variables: una que contenga los datos para el entrenamiento y otra que contenga las etiquetas verdaderas de cada instancia. En el código programado hasta este momento, se tiene una muestra con *Undersampling* del Dataframe original con una columna con los caminos a cada imagen. Basta con almacenar en una lista, generalmente llamada `X`, todas las imágenes de dicha muestra, las matrices leídas y redimensionadas con la función mostrada (Código 6-9). En otra variable lista, `y`, se almacena la columna `landmark_id` de la muestra, la cual tenía la clase real de cada instancia. Esto se refleja en el siguiente código. Los tipos elegidos para las variables y su contenido se deben principalmente a la compatibilidad entre ellas, necesaria en pasos futuros.

```python
X = []   #Imágenes
y = []   #Clases

for i in range(traindf_s.shape[0]):
    X.append(img_read_resize(traindf_s['img_path'][i]))
    y.append(np.array(traindf_s['landmark_id'][i]))

print('Tipos de las variables: \n')
print('X: ', type(X))
print('Elementos de X: ', type(X[0]))
print('\ny: ', type(y))
print('Elementos de y: ', type(y[0]))
```

```
Tipos de las variables:

X:  <class 'list'>
Elementos de X:  <class 'numpy.ndarray'>

y:  <class 'list'>
Elementos de y:  <class 'numpy.ndarray'>
```

Código 6-10. Asignación de imágenes redimensionadas a X y clases reales en y

6.5.4 Codificación *Label Encoding*

Durante el análisis de los datos se reveló que las clases de las imágenes no son consecutivas, es decir, hay clases que no son asignadas a ninguna muestra pero que siguen contando como etiquetas intrínsecamente. Un claro ejemplo de lo mencionado puede verse en la Figura 6-4, donde se pasa de la etiqueta 1 a la 7 sin usar ninguna de las clases existentes entre ellas. Esto un problema desde el punto de vista de la optimización.

Si las clases se mantuvieran en su forma original, tal y como están ahora mismo, la última capa de la CNN, el final de la RNA completamente conectada, solo funcionaría con un número de neuronas igual al número de clases, que estaría demasiado inflado. El número de neuronas de la capa final, y por lo tanto, el número de pesos totales de la red, se vería aumentado drásticamente en vano, ralentizando el entrenamiento.

En nuestro caso actual, se escogieron para la muestra las instancias con clases en el rango (0, 50], pero solo había entre ellas 20 clases disponibles (Figura 6-9). Esto significa que deberían utilizarse 50 neuronas en la última capa (comprobado empíricamente). Con el planteamiento mencionado, deberían ser únicamente 20 neuronas.

Para conseguir el objetivo, se decide recurrir a la codificación *Label Encoding* (3.4.3.1.2) de las etiquetas. Aunque esta sea usada habitualmente para convertir variables categóricas en variables numéricas discretas, en este código se usará para modificar los valores que pueden tomar las etiquetas usadas, que ya eran enteras. Se pretende redistribuir los valores de las etiquetas al rango [0, nº etiquetas diferentes usadas). Este cambio se hace antes de entrenar al modelo y se revierte al finalizar el entrenamiento y validación, sin que el modelo tenga constancia de lo sucedido. Se mantiene durante el aprendizaje y validación para reducir el número de neuronas inicial (igual a la clase máxima) al número de etiquetas diferentes usadas, un número siempre menor.

Se podría haber empleado *One-Hot Encoding* en lugar de *Label Encoding*, pero como se vio en la parte teórica (3.4.3.1.3), el primero necesitaría añadir más columnas (variables *dummy*) al Dataframe, lo que supone un aumento de la memoria usada que como se dijo en un inicio, escasea.

Para la implementación de esta idea, se emplea la clase `LabelEncoder` y sus métodos, disponibles en la biblioteca *Scikit-learn*. Primero se crea un objeto de dicha clase, `LE`, que será usado también para revertir los cambios tras el entrenamiento y la validación. El método `fit()` sirve para ajustar el objeto para que haga codificaciones en base al argumento dado. Se utiliza el vector `clases` con las etiquetas usadas definido en el Código 6-7. El método `transform()` se usa para codificar con el objeto ya ajustado un array. Para revertir la codificación se usa `fit_transform()` con el mismo objeto sobre un array codificado.

```
LE = LabelEncoder()
LE.fit(clases)
y_LE = LE.transform(y)
```

Código 6-11. *Label Encoding* de las etiquetas de la muestra

```
[ 1  1  1  1  7  7  7  7  7  7  7  7  9  9  9  9  9  9  9  9  9  9  9  9
  9  9  9  9 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11
 11 11 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12
 12 12 12 17 17 17 22 22 22 22 22 22 22 22 22 22 22 22 22 22 23 23 23 23
 23 23 23 23 23 23 23 23 23 23 23 23 24 24 24 24 24 24 24 24 24 24 27 27
 27 27 27 27 27 27 27 27 27 27 27 27 27 27 27 27 27 27 27 27 27 27 27 27
 27 27 27 27 27 27 27 27 27 27 27 27 27 27 27 27 27 27 27 27 27 27 27 27
 27 27 27 27 27 27 27 27 27 27 27 27 27 27 27 27 27 27 27 27 27 27 27 27
 27 27 27 27 27 27 27 29 29 29 29 29 30 30 30 30 30 30 32 32 32 32 32 32
 32 32 32 32 32 32 32 32 32 32 32 32 32 34 34 34 34 34 34 36 36 36 37 37
 37 37 37 37 37 37 37 37 37 37 37 37 37 37 37 37 37 37 37 37 37 37 37 37
 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 43 43 43 43 43 43 43 43
 43 43 43 43 43 43 43 43 43 43 43 43 43 43 43 43 43 43 43 43 43 43 43 43
 43 43 43 43 43 43 43 43 43 43 43 48 48 48 48 48 48 48 48 48 50 50 50 50
 50 50 50 50 50 50 50 50 50]
```

```
[ 0  0  0  0  1  1  1  1  1  1  1  1  2  2  2  2  2  2  2  2  2  2  2  2
  2  2  2  2  3  3  3  3  3  3  3  3  3  3  3  3  3  3  3  3  3  3  3  3
  3  3  4  4  4  4  4  4  4  4  4  4  4  4  4  4  4  4  4  4  4  4  4  4
  4  4  4  5  5  5  6  6  6  6  6  6  6  6  6  6  6  6  6  6  7  7  7  7
  7  7  7  7  7  7  7  7  7  7  8  8  8  8  8  8  8  8  9  9  9  9  9  9
  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9
  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9
  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9
  9  9  9  9  9  9  9 10 10 10 10 10 11 11 11 11 11 11 12 12 12 12 12 12
 12 12 12 12 12 12 12 12 12 12 12 12 13 13 13 13 13 14 14 14 15 15 15 15
 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15
 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 17 17 17 17 17 17 17
 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17
 17 17 17 17 17 17 17 17 17 17 17 17 18 18 18 18 18 18 18 18 19 19 19 19
 19 19 19 19 19 19]
```

Figura 6-12. Comparación: etiquetas en y sin codificar (izquierda) y codificadas (derecha)

6.5.5 Normalización de imágenes

Para mejorar aún más la eficiencia del programa, se aplica un escalado Min-Max a las imágenes almacenadas en la variable `X` para trasladar el valor de los píxeles de cada plano de color del rango [0, 255] al rango [0, 1], siguiendo la fórmula (3.3), es decir, dividiendo entre 255 todos los valores de las matrices tridimensionales. De esta forma, las operaciones que se realicen con ellos no alcanzarán valores tan altos que ralenticen el sistema o que provoquen errores si se utilizaran funciones de activación de las neuronas como sigmoides (siempre saturaría), por ejemplo. Las imágenes seguirán visualizándose de la misma forma, sin percibir ningún cambio.

```
X = [n/255 for n in X]
```

Figura 6-13. Escalado Min-Max de imágenes en X

6.5.6 Separación de los datos

Una práctica muy común en técnicas de ML en aprendizaje supervisado consiste en dividir el *dataset* proporcionado en un inicio en tres partes: set de entrenamiento, set de validación y set de testeo. Cada uno se utiliza en la parte del proceso con el mismo nombre, con cuidado de no utilizar las mismas imágenes en dos procesos distintos, pues daría un resultado poco veraz. El porcentaje de instancias del *dataset* original destinadas a cada subset es elegido por el usuario. No existe un valor universal. Se escoge según la situación en la que se desarrolla el modelo. Como en nuestro caso no abundan muestras de algunas clases, primero se ha decidido destinar un 10% de los datos originales a la validación. Del 90% restante, un 5% se dedica al testeo (4.5% del total) y el resto, que es la gran mayoría, se dedican al entrenamiento (85.5% del total). Existe una función que facilita enormemente esta división, `train_test_split()` de la biblioteca *Scikit-learn*, que utiliza la misma semilla `SEED` y además casi siempre se le activa la opción de barajar los datos para que no se cojan solo muestras de la misma clase al estar consecutivamente situadas. Antes de utilizar esta función dos veces, las variables `X`, `y` e `y_LE` se convierten en arrays de *Numpy* por compatibilidad. Se crearán variables X e y correspondientes al entrenamiento (train), validación (val) y testeo, cada una conteniendo las imágenes y las etiquetas correspondientes, del mismo tipo que `X` e `y`.

```
X = np.array(X)      #Conversión de lista a array necesaria
y_LE = np.array(y_LE)
y = np.array(y)

#Separación de los datos en datos para entrenamiento, datos para validación y datos de testeo

#Validación
X_train, X_val, y_train, y_val = train_test_split(X, y_LE, test_size = 0.10, random_state=SEED, shuffle=True)
        #Mismo tipo que X e y (ndarray formado por ndarrays o int64, respectivamente)

#Testeo
X_train, X_test, y_train, y_test = train_test_split(X_train, y_train, test_size = 0.05, random_state=SEED, shuffle=True)

n_datos_entrenamiento = len(X_train)

print('Imágenes en X: ',len(X), ' y etiquetas en y: ', len(y_LE))
print('Entrenamiento. Imágenes en X_train: ',len(X_train), 'y etiquetas en y_train: ',len(y_train))
print('Validación. Imágenes en X_val: ',len(X_val), 'y etiquetas en y_val: ',len(y_val))
print('Testeo. Imágenes en X_test: ',len(X_test), 'y etiquetas en y_test: ',len(y_test))
```

```
Imágenes en X:  345  y etiquetas en y:  345
Entrenamiento. Imágenes en X_train:  294 y etiquetas en y_train:  294
Validación. Imágenes en X_val:  35 y etiquetas en y_val:  35
Testeo. Imágenes en X_test:  16 y etiquetas en y_test:  16
```

Código 6-12. Separación en datos de entrenamiento, validación y testeo

6.5.7 *Data Augmentation*

Como se viene repitiendo, se disponen de muy pocas muestras de algunas clases. Muchas de ellas tienen en un inicio únicamente tres muestras, y según el azar de la semilla usada, su clase puede dividirse en sets distintos dejando tan solo una o dos muestras de dicha etiqueta para el entrenamiento o incluso ninguna. Esto empeora la capacidad predictiva del modelo, errando gran mayoría o todas las predicciones acerca de dichas clases, provocando un modelo con mala generalización, con un claro *overfitting* por aprender siempre de los mismos datos. Para solucionar este problema, se emplea el aumento de datos, más conocido como ***Data Augmentation***. Es una técnica para crear artificialmente nuevos datos durante el entrenamiento, en nuestro caso nuevas imágenes, a partir de los datos de entrenamiento ya existentes. Para ello se aplican una serie de transformaciones indicadas por el usuario que no deben modificar en exceso los datos para que el sistema extraiga más información de las clases. Las proporciones entre clases siguen manteniéndose tras el aumento de datos. Hay dos vías posibles para implementar el aumento de datos: mediante generador de imágenes o mediante capas de la CNN.

El generador de imágenes, con la clase ImageDataGenerator, emplea un objeto de dicha clase y un iterador que recorre los datos de entrenamiento. Después con otra función se entrena el modelo gracias al iterador que devuelve en cada *step* del entrenamiento un lote de tamaño *batch size* de solo imágenes transformadas a partir del set de entrenamiento inicial.

El aumento de datos mediante capas del modelo tiene una implementación más sencilla. Simplemente se colocan capas o *layers* dentro del modelo de CNN que simbolizan a las transformaciones.

En ambos métodos, el aumento solo se produce en datos de entrenamiento. Sin embargo, hay excepciones: ciertas transformaciones como pueden ser el redimensionamiento o reescalado también se aplican durante la validación y testeo (*inference mode*) a los datos correspondientes. Debe consultarse la documentación de *Keras* para esclarecer las dudas.

En el código que se está desarrollando, tras probarse ambos métodos, se elige definitivamente el basado en *layers*, pues el generador de imágenes no resultó para nada eficiente, aumentando el tiempo de ejecución del programa considerablemente y llenando la memoria. A modo de prueba, se implementa el mismo conjunto de capas para *Data Augmentation* que se usará en el modelo pero fuera de este, con el fin de hacerle pasar imágenes y visualizar los cambios realizados.

```
data_augmentation = tf.keras.Sequential([
    #preprocessing.RandomFlip('horizontal'),
    preprocessing.RandomZoom(height_factor=(-0.2, 0.2)),
    preprocessing.RandomContrast(0.4),
    preprocessing.RandomTranslation(height_factor= (-0.2, 0.2), width_factor=(-0.2, 0.2)),
    preprocessing.RandomRotation(factor= (-0.1, 0.1)),     #Giros entre -10%*2*pi y 10%*2*pi
    #preprocessing.RandomCrop(120, 120, seed=SEED)
])

#Se representa una imagen original de la muestra junto a algunas de sus posibles modificaciones
plt.figure(figsize=(25,7))
j = 1 #np.random.randint(0, N_DATOS)
img_orig = X[j]
plt.subplot(2 , 6, 1)
plt.xticks([])
plt.yticks([])
plt.xlabel('ORIGINAL. landmark_id: '+ str(y[j]), fontweight ="bold", fontsize=12)
plt.imshow(img_orig)

for i in range(11):
    img_mod = data_augmentation(img_orig)
    plt.subplot(2 , 6, i+2)
    plt.xticks([])
    plt.yticks([])
    plt.xlabel('Modificación '+str(i+1), fontweight ="bold", fontsize=12)
    plt.imshow(img_mod)

plt.show()
```

Código 6-13. Aumento de datos fuera del modelo para visualizar cambios

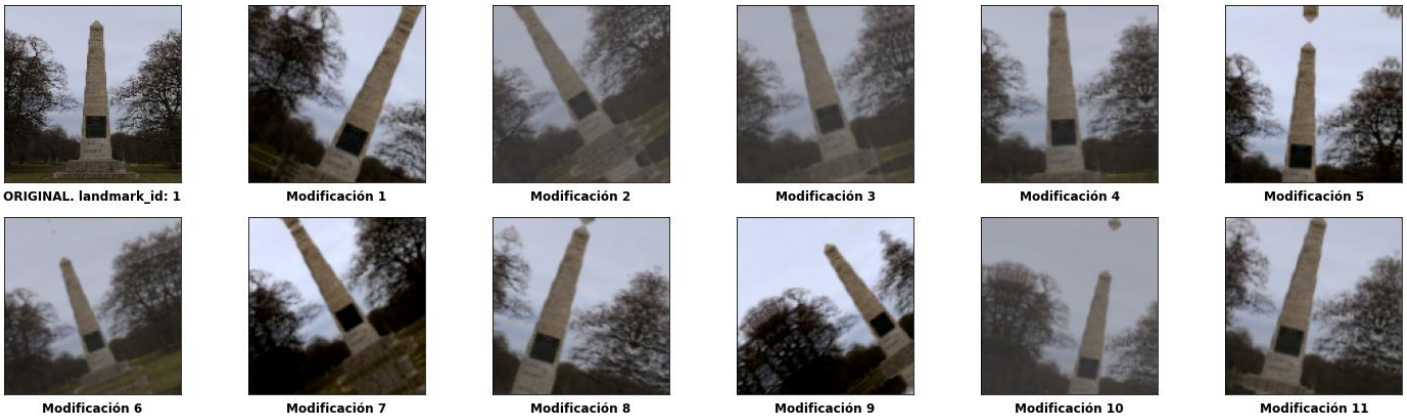


Figura 6-14. Ejemplo de aplicación de *Data Augmentation*

En las primeras líneas del Código 6-13 se define un modelo, exactamente igual al que se usará dentro de la CNN, pero solo con la utilidad de ver las modificaciones en las imágenes originales y ajustar sus capas y parámetros hasta lograr el resultado deseado. Se empleará un zoom aleatorio, una variación aleatoria del contraste, un desplazamiento tanto lateral como vertical y una pequeña rotación. Todas estas transformaciones se limitan para no modificar excesivamente la naturaleza de las imágenes. Se busca evitar la confusión del modelo, que no detecte patrones erróneos debido a variar demasiado la estructura espacial de imágenes de una misma clase. Se ha decidido no usar volteos horizontales de la imagen pues provocan malentendidos, al menos en el caso de lugares emblemáticos, ya que prácticamente ninguno va a ser fotografiado en modo espejo. También se ha desechado la idea de emplear recortes aleatorios, principalmente por el aumento tiempo de procesado que supone y porque no se resulta tan útil como en problemas de detección de elementos.

6.5.8 *Oversampling*

El último paso del preprocesado de los datos previo a la creación del modelo en sí será el *Oversampling*. Como se señaló antes, permite ecualizar la distribución de clases gracias al aumento de muestras de las clases menos frecuentes. En este caso se aplicará únicamente sobre los datos de entrenamiento, `X_train` e `y_train`. Hay dos formas de Oversampling: mediante duplicado o con SMOTE.

El sobremuestreo por duplicado consiste en copiar las imágenes de las clases con menos ocurrencias para aumentar su frecuencia hasta cierto nivel. Esto no añade información nueva, pero junto a *Data Augmentation* da grandes resultados, pues aumenta la cantidad de imágenes transformadas de las clases minoritarias, que sí que dan más información durante el aprendizaje.

Por otro lado, SMOTE, o *Synthetic Minority Oversampling TEchnique*, es una técnica similar al aumento de datos muy empleada en RNA. Crea nuevas muestras solo de las de clases minoritarias mediante un algoritmo de agrupamiento de k vecinos en el espacio de características. Se selecciona un vecino aleatorio de los k existentes en la vecindad de una muestra de la clase poco frecuente. Después se traza una línea entre dicho vecino y la muestra, y aleatoriamente se selecciona un punto de esta línea. Este punto, que tendrá propiedades similares a las de la vecindad, será considerado como una nueva muestra de esa clase minoritaria.

En el código se usará el *Oversampling* por duplicado. La razón de esta decisión es, de nuevo, la eficiencia y la memoria utilizada. SMOTE debe ejecutar un algoritmo completo de agrupamiento, lo que supone una gran carga computacional extra en comparación con el tener solo que duplicar muestras hasta cierto nivel de repeticiones.

Se ha diseñado un bucle que va rellenando los arrays `X_train` e `y_train` con duplicados aleatorios de cada clase minoritaria hasta llegar a un umbral, `UMBRAL_OVERSAMPLING`, determinado por un porcentaje del umbral del *Undersampling* realizado en un inicio, `UMBRAL_UNDERSAMPLING`. En función de la cantidad y resolución de las imágenes que se manejen, este porcentaje será ajustado a un valor mayor o menor, pues el método `np.concatenate()` es bastante lento.

```
start_time = time.time()

UMBRAL_OVERSAMPLING = 0.5 * UMBRAL_UNDERSAMPLING       #PORCENTAJE DEL UMBRAL_UNDERSAMPLING HASTA EL QUE SE AUMENTA
i=0; n=0; k=0

for i in range(N_CLASES):     #Número de clases únicas
    cont=0
    for n in range(len(y_train)):
        if y_train[n] == i:
            cont = cont+1

    if cont < UMBRAL_OVERSAMPLING:
        img_clase_i = X_train[y_train==i]
        for k in range(int(UMBRAL_OVERSAMPLING-cont)):
            X_train = np.concatenate((X_train,np.array([(img_clase_i[np.random.randint(len(img_clase_i))])])),axis = 0)
            y_train = np.append(y_train, i)

print('\nDuración del oversampling: %s segundos' % (time.time() - start_time))
n_datos_train_def = len(X_train)
print('\nDatos de entrenamiento tras oversampling: '+str(n_datos_train_def)+' datos')
print('\nLos datos de entrenamiento han pasado de: '+str(n_datos_entrenamiento)+' a:'+ str(n_datos_train_def))
```

```
Duración del oversampling: 45.73091983795166 segundos

Datos de entrenamiento tras oversampling: 825 datos

Los datos de entrenamiento han pasado de: 294 a:825
```

Código 6-14. Oversampling por duplicado de datos de entrenamiento

Se observa un aumento de imágenes de entrenamiento de 294 a 825, según los valores de las variables utilizadas hasta ahora. Utilizando la GPU de Kaggle, esta operación se ha demorado 45.73 segundos. Si se hubieran empleado más datos en la muestra inicial, este tiempo sería mucho más elevado. En la siguiente figura se representa la distribución de clases del set de entrenamiento después del *Oversampling*. Se han logrado ecualizar las frecuencias en gran medida, aunque podrían haberse ecualizado totalmente, pero por cuestiones de eficiencia no se ha visto necesario. Nótese que el umbral de *Undersampling* (línea roja) no corta a la clase más representada (la 27). Esto se debe a que el *Undersampling* se realizó sobre la muestra completa, y más tarde esta se dividió en tres sets, por lo que la diferencia entre la barra más alta y el umbral se corresponde con las muestras de dicha clase para la validación y el testeo.

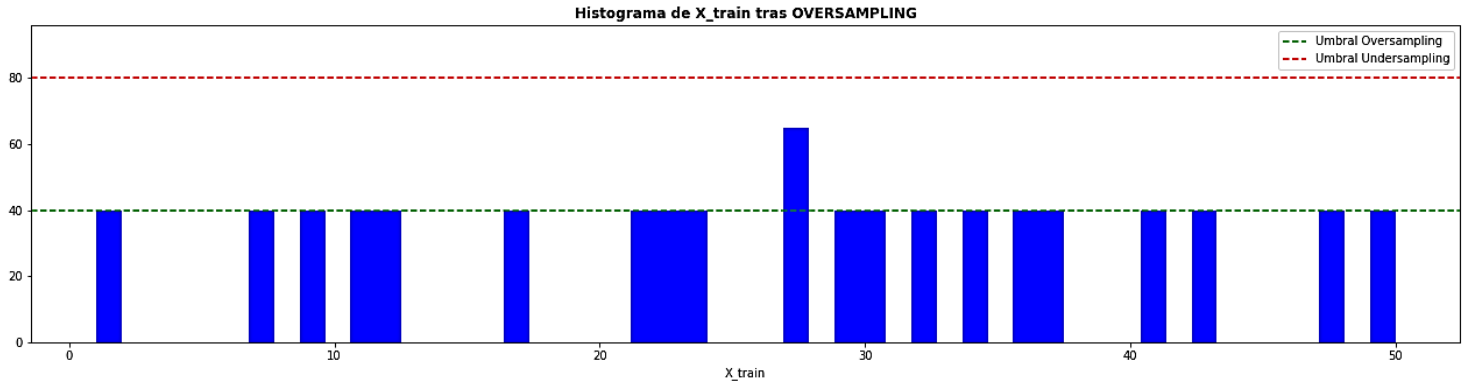


Figura 6-15. Distribución de clases en set de entrenamiento tras *Oversampling*

6.6 Optimización

Debido a la gran carga computacional que supone la ejecución del programa y a la facilidad con la que se colapsa la memoria, se ha visto necesario la incorporación de diversas técnicas de optimización del código, aumentando así la eficiencia de este.

Una de ellas, como se ha podido ver en el apartado anterior, consiste en convertir algunas variables *Numpy* a tipos que no requieran tanto espacio en memoria (***downcasting***), como por ejemplo pasar de `int64` a `int32`. Esta operación debe realizarse con sumo cuidado para no perder información, pues al reducir el tipo también disminuye el rango de números representables por esa variable.

Otra técnica empleada varias veces a lo largo del código es la de **eliminar de la memoria variables y valores** que no se vayan a usar más para no malgastar recursos. Esto se consigue gracias a la actuación conjunta de varios métodos. Primero, para conocer con exactitud el tamaño de una variable, no se va a utilizar el método `getsizeof()` de la biblioteca, pues no es preciso para las estructuras que se manejan. En su lugar se empleará una función que utiliza dicho método y que devuelve el tamaño de la variable en bytes (Código 6-15) [94]. Después, se utiliza la palabra reservada `del` junto a la variable a eliminar. Esto permite borrar la variable al desvincularla del espacio de memoria al que apuntaba. Sin embargo, este espacio de memoria sigue ocupado físicamente, o sea, realmente aún no se ha liberado la memoria del disco. Python dispone de un recolector de basura que automáticamente tras cierto tiempo elimina los valores no referenciados o apuntados de la memoria. Este tiempo a transcurrir es incierto. Para que esta tarea se realice inmediatamente, se invoca al recolector de basura de Python con `gc.collect()`. Uno de los varios usos de la técnica que se ha comentado se tiene en el Código 6-16.

```python
# Obtener la memoria de cualquier objeto en Bytes
def get_size(obj, seen=None):
    """Recursively finds size of objects"""
    size = sys.getsizeof(obj)
    if seen is None:
        seen = set()
    obj_id = id(obj)
    if obj_id in seen:
        return 0
    # Important mark as seen *before* entering recursion to gracefully handle
    # self-referential objects
    seen.add(obj_id)
    if isinstance(obj, dict):
        size += sum([get_size(v, seen) for v in obj.values()])
        size += sum([get_size(k, seen) for k in obj.keys()])
    elif hasattr(obj, '__dict__'):
        size += get_size(obj.__dict__, seen)
    elif hasattr(obj, '__iter__') and not isinstance(obj, (str, bytes, bytearray)):
        size += sum([get_size(i, seen) for i in obj])
    return size
```

Código 6-15. Función para obtener el tamaño exacto de cualquier variable en bytes [94]

```python
#Eliminar de memoria el dataset completo sin imágenes; solo se usará la muestra
print('ELIMINANDO ' + str(round((get_size(traindf)/(1024*1024)),4)) +' MB ocupados por traindf')
del traindf          #Elimina la relación entre la variable y el espacio de memoria al que apunta
gc.collect();        #Elimina de memoria y devuelve los objetos a los que ya no se apunta
```

```
ELIMINANDO 720.4772 MB ocupados por traindf
```

Código 6-16. Técnica para reducir memoria durante la ejecución

6.7 Creación de la CNN

6.7.1 Hiper-parámetros

Antes de idear la estructura de la red, se crean unas variables globales que definen hiperparámetros del modelo, valores que el usuario ajusta tras comprobar los resultados obtenidos en la validación. Estas variables serán usadas dentro de las capas de la CNN, por lo que definen su comportamiento y modelan los resultados. Las variables definidas a lo largo del preprocesado de los datos también serían hiperparámetros del modelo, como el número de clases a utilizar, los umbrales de Undersampling y Oversampling, la resolución de las imágenes...

Se muestra el bloque de código con las declaraciones de algunos de estos hiperparámetros, que serán explicados en mayor detalle cuando corresponda su uso:

```
#Hiperparámetros de la red neuronal convolucional

kernelSize = (3,3)          #Tamaño de la plantilla de convolución
paddingType = 'same'        #Cómo se procede en los bordes de las imágenes ('same' o 'valid')
activationF = 'relu'        #Función de activación
poolSize = (2,2)            #Tamaño de la plantilla de maximum pooling
stridesSize = (2,2)         #Desplazamiento de plantilla durante maximum pooling
dropoutRate = 0.5           #Porcentaje de neuronas que se desactivan con la capa Dropout
batchSize = 128             #Cantidad de datos con los que se entrena en cada época
epochsSize= 1000            #Número de épocas en las que se entrena
lr = 0.007                  #Learning Rate
weightDecay = regularizers.L2(0.01) #Regularización de pesos ---> L1: Suma pesos absolutos.
                            #L2: Suma pesos cuadrados. L1L2: Suma pesos absolutos y cuadrados
```

Código 6-17. Algunos hiperparámetros de la CNN

6.7.2 Capas

Los modelos de CNN creados con la biblioteca *Keras* se reducen a una secuencia de capas predefinidas agregadas por el usuario y a las que les puede modificar sus argumentos. Siempre se comienza creando un objeto de la clase `Sequential`. Dentro de este se irán colocando todas las capas. Debido a que se va a realizar *Data Augmentation*, primero se añaden las capas relativas a dicho proceso, como se mencionaron en el apartado 6.5.7.

```
#Creación del modelo
model = keras.Sequential([

    #DATA AUGMENTATION
    preprocessing.RandomZoom(height_factor=(-0.2, 0.2)),
    preprocessing.RandomContrast(0.4),
    preprocessing.RandomTranslation(height_factor= (-0.2, 0.2), width_factor=(-0.2, 0.2)),
    preprocessing.RandomRotation(factor= (-0.1, 0.1)),     #Giros entre -10%*2*pi y 10%*2*pi
```

Código 6-18. Creación del modelo y *Data Augmentation*

Después se comenzará a construir realmente el modelo CNN, primero el *backbone* para la extracción de las características de las imágenes y a continuación la *head* para la clasificación de los vectores de características obtenidos.

El ***backbone***, como se señaló en el apartado teórico, consta esencialmente de dos tipos de capas: las convolucionales y las de agrupación.

- Capas convolucionales: se corresponden con `layers.Conv2D`. El primer hiperparámetro que reciben es el número de filtros que aplican, es decir, la cantidad de imágenes (mapas de características) que salen de la capa por cada imagen que entra. El tamaño de cada filtro o kernel es el siguiente argumento, normalmente de tamaño 3x3. *Stride* es el desplazamiento del kernel horizontal y vertical en cada paso al recorrer una imagen, por lo que al asignarle el valor 1 se realiza una convolución típica. *Padding*

modifica los valores en los bordes de la matriz y puede tomar dos valores, "valid", que deja la imagen inicial igual y al aplicarle un filtro resulta en una imagen menor dimensión porque la convolución no ha podido hacerse en los bordes, y "same", que permite la convolución en los bordes de la imagen original, rellenando con ceros los valores inexistentes para un correcto cálculo. Al emplear `strides=1` y `padding="same"`, la imagen resultante de aplicar el filtro tendrá exactamente las mismas dimensiones que la original. En cuanto a la función de activación, se utiliza la unidad rectificadora lineal ReLU (Figura 4-4), *f(x)=max(0,x)*, como es típico en la mayoría de RNA multicapa, garantizando discriminación entre clases no linealmente separables. La principal razón de su uso radica en su velocidad de cómputo, manteniendo en las matrices los valores positivos y sustituyendo por cero los negativos. Además, en RNA profundas, el uso de ReLU evita el problema del **desvanecimiento del gradiente** (*vanishing gradient*), que provoca que no se actualicen parámetros durante el entrenamiento al ser demasiado profunda la red, gracias a que su derivada es 0 para valores negativos y 1 para positivos, por lo que la encadenación de productos sucesivos da siempre 0 o 1. La primera capa convolucional recibirá las imágenes de entrada pasadas por *Data Augmentation*, por lo que se le añade el argumento *input shape*, coincidente con las dimensiones de estas porque se redimensionaron en un paso previo.

- Capas de agrupación: se corresponden con `layers.MaxPool2D`, pues se utiliza el agrupamiento *max-pooling* para reducir las dimensiones (altura y anchura) de los mapas de características, cuya cantidad va en aumento a medida que se recorren capas de convolución. Reciben tres hiperparámetros. El primero para indicar el tamaño de los bloques de píxeles de la imagen original en los que se hará la operación de seleccionar el mayor valor. Normalmente se usa un tamaño de 2x2. El siguiente valor hace referencia a *strides*, con el mismo significado que en las capas convolucionales. Y el tercer hiperparámetro es *padding* también con la misma interpretación y valores posibles que en las capas convolucionales. En el código implementado, los hiperparámetros se han ajustado de tal manera que los mapas de características que entran en una capa de *max-pooling* salen de esta con la mitad de los píxeles en altura y anchura.

Aparte de estas dos capas en el *backbone*, se emplea también `layers.BatchNormalization`, cuya función es acelerar la convergencia en el descenso del gradiente y estabilizar el entrenamiento gracias a la normalización de los resultados salientes de las capas de convolución. Esta capa se empleará junto a todas las demás de la CNN.

Se han creado cinco bloques Conv2D-BatchNormalization-MaxPool2D porque tras probar varios entrenamientos con más o con menos capas, el resultado obtenido siempre era peor. Esto puede deberse a que las características extraídas con cinco bloques están en un punto medio entre características de bajo nivel (con cuatro bloques) o demasiado complejas (con seis bloques), es decir, puede haberse conseguido el nivel óptimo de abstracción, al menos para las muestras empleadas y para los hiperparámetros utilizados. Los modelos de RNA son muy relativos, dependen de una gran cantidad de hiperparámetros y de las muestras usadas en el entrenamiento y validación, por lo que es prácticamente imposible encontrar la mejor combinación posible para la situación de desarrollo. La forma de optimizar los hiperparámetros es fijar inicialmente a un valor la gran mayoría de ellos e ir variando unos pocos mientras se hacen varios entrenamientos hasta encontrar un resultado que alcance las expectativas, que genere cierta conformidad.

```
#BASE DEL MODELO ---- Extracción de características
#Bloque convolucional 1
layers.Conv2D(filters=32, kernel_size=kernelSize, strides=1, padding=paddingType, activation=activationF, input_shape=[IMG_SIZE, IMG_SIZE, 3]),
layers.BatchNormalization(),
layers.MaxPool2D(pool_size=poolSize, strides=stridesSize, padding=paddingType),
#Bloque convolucional 2
layers.Conv2D(filters=64, kernel_size=kernelSize, strides=1, padding=paddingType, activation=activationF),
layers.BatchNormalization(),
layers.MaxPool2D(pool_size=poolSize, strides=stridesSize, padding=paddingType),
#Bloque convolucional 3
layers.Conv2D(filters=128, kernel_size=kernelSize, strides=1, padding=paddingType, activation=activationF),
layers.BatchNormalization(),
layers.MaxPool2D(pool_size=poolSize, strides=stridesSize, padding=paddingType),
#Bloque convolucional 4
layers.Conv2D(filters=256, kernel_size=kernelSize, strides=1, padding=paddingType, activation=activationF),
layers.BatchNormalization(),
layers.MaxPool2D(pool_size=poolSize, strides=stridesSize, padding=paddingType),
#Bloque convolucional 5
layers.Conv2D(filters=512, kernel_size=kernelSize, strides=1, padding=paddingType, activation=activationF),
layers.BatchNormalization(),
layers.MaxPool2D(pool_size=poolSize, strides=stridesSize, padding=paddingType),
```

Código 6-19. *Backbone* de la CNN

Una vez se ha recorrido el *backbone* al completo, se tendrá una masiva cantidad de mapas de características pero de tamaño reducido debido a los *max-pooling*. Para que todos estos puedan insertarse en la RNA completamente conectada del final de la CNN (*head*) deben comprimirse, convertirse en un único vector de características. Esta tarea recae en la capa `layers.Flatten`, que equivale a un *reshape* de todos estos mapas de características resultantes del *backbone* sin perder información.

La ***head*** de la CNN será una RNA completamente conectada, siempre formada por capas `layers.Dense`. Puede usarse cualquier número de ellas, pero en el modelo creado se emplearán solo dos porque tras probar sucesivas ejecuciones, es la configuración que mejor resultados da. A la primera capa densa se le pasan tres argumentos: el número de neuronas de la capa, la función de activación de las neuronas de la capa y un *kernel regularizer*. Se han elegido 512 neuronas en la primera capa porque proporcionan un mejor resultado que valores inferiores y superiores. No existe una fórmula universal para elegir el número de neuronas de dicha capa, pero sí que circulan entre los especialistas en RNA varios criterios propios para su elección, aunque no son completamente fiables. La función de activación será igual que la de las capas convolucionales, ReLU. El regularizador de kernel reduce los pesos de la capa (excluyendo el sesgo) mediante un regularizador, en este caso divide cada peso por la suma de los pesos cuadrados, o sea, la norma L2, multiplicada por 0.01 (creación en última línea del Código 6-17). A la segunda capa densa, que es la output layer de la CNN, con el parámetro `units` se le indican el número de salidas del modelo, que siempre coincide con el número de etiquetas posibles de la clasificación. Gracias a que se realizó un *Label Encoding*, el número de neuronas de salida (con todos los parámetros que conlleva) se reduce notablemente. Por otro lado, la función de activación de esta capa no es la misma que la utilizada en las demás capas de la CNN. La última capa en un problema de clasificación multiclase con una única etiqueta por muestra siempre emplea la función de activación ***softmax*** (Figura 4-3), que devuelve en cada neurona la probabilidad de que la imagen pertenezca a la clase que representa dicha neurona. La suma de las probabilidades de todas las neuronas da siempre uno. Típicamente, se escoge como etiqueta predicha de la imagen la de la neurona con mayor probabilidad.

El tener unos pesos grandes es indicador de *overfitting* y de tendencia a aprender el ruido de las muestras. Como esto no interesaba, se recurrió al regularizador de kernel de la primera capa densa, permitiendo ser más agresivos durante el aprendizaje (aumentar el *learning rate*) al disminuir los pesos. La primera causa de *overfitting* es el entrenar demasiado tiempo con pocos datos. Una solución fácil de implementar y computacionalmente eficiente es la popular regularización por ***dropout***. Consiste en desactivar temporalmente (durante un *step*) neuronas aleatorias y sus conexiones pre y postsinápticas correspondientes en una capa durante el entrenamiento. De esta forma se fuerza a otras neuronas más inactivas a aprender. Este método de regularización se usa para cualquier tipo de capa de una RNA, salvo en las capas de salida. Se implementa con `layers.Dropout` y recibe como primer argumento el porcentaje de neuronas que se desactivan cada *step* y como segundo argumento la semilla para garantizar reproducibilidad del proceso aleatorio. La configuración más habitual (y empleada en el código) es la de asignarle el valor 0.5 a la tasa de *dropout* y utilizarla en las capas densas antes de la de salida. Un valor mayor a 0.5 es contraproducente.

```
    #CABEZA DEL MODELO ---- Clasificación
    layers.Flatten(),
    layers.Dense(units = 512, activation=activationF, kernel_regularizer=weightDecay),
    layers.Dropout(rate=dropoutRate,seed=SEED),
    layers.BatchNormalization(),
    #Output Layer
    layers.Dense(units = N_CLASES, activation='softmax'),
])
```

Código 6-20. *Head* de la CNN

6.7.3 Optimizadores

Para ajustar el valor de los parámetros de la CNN, pesos y sesgos, en los filtros de las capas convolucionales y en las conexiones sinápticas de la red completamente conectada, se utilizan los llamados optimizadores. El optimizador que implementa el algoritmo del descenso del gradiente, tal y como se explicó teóricamente en 4.3, es el `SGD`. Aunque con su nombre se pueda referir a *Stochastic Gradient Descent*, realmente puede emplearse para *Batch Gradient Descent* y para *Mini-Batch Gradient Descent*, siendo este último el más común. La

diferencia entre uno y otro radica en la cantidad de muestras usadas para el entrenamiento, *batch size*, en cada *step*, un hiperparámetro definido globalmente y no ligado únicamente a este optimizador. Normalmente el *batch size* empleado es un número potencia de 2.

Además de `SGD`, se han probado otros optimizadores basados también en *gradient descent*, como `RMSprop` y `Adam`, pero todos proporcionando resultados inferiores tras sucesivas pruebas con diferentes hiperparámetros. El optimizador `SGD` es el que mejor resultados produce. Aunque tarde más en converger y sea más complejo de ajustar, la generalización que produce es superior a la de `Adam` y `RMSProp` para este modelo. `SGD` recibe como argumentos el hiperparámetro referente al *learning rate*, el momento, que por defecto es cero, pero que al aumentar su valor se converge más rápido en un mínimo y además amortigua oscilaciones, y por último el momento de Nesterov, que es una modificación del momento para dar más importancia al *learning rate* y gradientes (se mantiene desactivado).

```
#OPTIMIZADORES
opt1 = tf.keras.optimizers.RMSprop(learning_rate=lr, rho=0.9, momentum=0.5, epsilon=1e-07)
opt2 = tf.keras.optimizers.Adam(learning_rate=lr, beta_1=0.9, beta_2=0.999, epsilon=1e-07)
opt3 = tf.keras.optimizers.SGD(learning_rate=lr, momentum=0.9, nesterov=False)
```

Código 6-21. Optimizadores posibles de la CNN

6.7.4 Compilación

Al compilar el modelo se indica qué optimizador, función de coste y métrica se utilizará. El optimizador elegido es el `SGD` actuando como *Mini-Batch Gradient Descent* con tamaño de lote de 128, que produce un mejor resultado que las demás potencias de 2 contiguas, además de un equilibrio entre velocidad de ejecución y potencia computacional requerida. La función de coste es aquella que se pretende minimizar en el cálculo del descenso del gradiente. En problemas de clasificación multiclase es habitual utilizar *Categorical Cross-Entropy Loss* (3.12). Como las variables objetivo del problema, las clases, son numéricas enteras y no están codificadas mediante One-Hot, se utiliza como *loss function* `'sparse_categorical_crossentropy'`. En cuanto a la métrica empleada, se ha decidido utilizar *accuracy*, pues aunque no es un buen indicador para casos en los que existe gran *Imbalanced Data*, los datos que se van a emplear ya han sido ecualizados. *Accuracy* calcula el porcentaje de predicciones correctas del total de predicciones realizadas en una época. De nuevo, como las etiquetas son variables enteras y no tienen *One-Hot Encoding*, se emplea `'sparse_categorical_accuracy'`. Tanto la variación de la función de pérdida como de la métrica empleada serán representadas para el entrenamiento y validación con el fin de comprobar la evolución del comportamiento del modelo y ajustar los hiperparámetros en consecuencia.

```
#Compilación del modelo
model.compile(
    optimizer = opt3,
    loss = 'sparse_categorical_crossentropy',
    metrics = ['sparse_categorical_accuracy']
)
```

Código 6-22. Compilación del modelo

6.7.5 *Callbacks*

Los *callbacks* son objetos creados dentro del modelo que permiten realizar ciertas acciones en mitad del proceso de entrenamiento. En el código se han implementado tres *callbacks* distintos que resultan de utilidad.

El primer callback definido es el de ***Early Stopping***. Como puede intuirse, permite que el modelo termine el entrenamiento antes de completar todas las épocas indicadas en el hiperparámetro. La decisión de finalizar el proceso antes de tiempo depende de si no se produce ninguna mejoría de una variable tras un tiempo de espera. Para ello, al *callback* se le pasan una serie de argumentos. El primero, `monitor`, es la variable a monitorizar, que en este caso es la función de pérdida durante la validación, que como se indicó anteriormente es

`'sparse_categorical_crossentropy'`. El parámetro `mode` sirve para elegir si el detonante del fin del entrenamiento es que la variable monitorizada deje de disminuir o deje de aumentar. Esto dependerá del tipo de variable monitorizada. En este caso al ser la función de coste el *Early Stopping* se activaría tras un período de tiempo sin disminuir, sin mejorar. Al asignarle el modo "auto", el sistema elige automáticamente según el nombre de la variable monitorizada. El siguiente argumento, `min_delta`, es la cantidad de cambio que debe suceder para reiniciar la cuenta atrás del *Early Stopping*. El parámetro `patience` es el que representa las épocas que como máximo pueden transcurrir sin producirse la mejoría antes de finalizar el aprendizaje. Por último, el parámetro `restore_best_weights`, permite que el modelo almacene, al activarse el *Early Stopping*, aquella combinación de pesos que mejor valor de la variable monitorizada dieron. El principal beneficio que se obtiene al usar este *callback* es que se evita el *overfitting* del modelo al cortar el entrenamiento antes de que ocurra.

Otro *callback* empleado es el de ***ReduceLROnPlateau***. Es similar en varios aspectos al *Early Stopping*, pero a diferencia de este no acaba el entrenamiento al no detectar mejoría en una variable, sino que reduce el *learning rate* para intentar converger en el mínimo con mayor facilidad. De nuevo hay que pasarle una variable a monitorizar (otra vez la función de pérdida para la validación) y un tiempo máximo de espera, normalmente bastante menor que el de *Early Stopping*. Otros argumentos adicionales son el factor por el que se multiplica el *learning rate* actual cuando se activa el *callback*, el valor mínimo que este puede alcanzar, el tiempo de espera para volver a monitorizar tras la llamada, o si se quiere mostrar por pantalla cuándo se produce la activación.

El último *callback* es ***ModelCheckpoint***. Este permite guardar el mejor modelo del entrenamiento, es decir, la combinación de pesos que mejor valor de la variable monitorizada provocan. A diferencia de *Early Stopping*, que asignaba al modelo directamente los mejores pesos al activarse su *callback*, *ModelCheckpoint* solo almacena la mejor combinación de pesos en un archivo en el directorio indicado. Cuando detecta una mejor combinación de pesos, sustituye la que tenía guardada por ella. Posteriormente estos pesos pueden cargarse al modelo mediante `model.load_weights()`.

```
#Callback 1
early_stopping = EarlyStopping(
    monitor = "val_loss",
    mode = "auto",
    min_delta = 0.0001,
    patience = 120,
    restore_best_weights = True)

#Callback 2
reduce_lr = ReduceLROnPlateau(
    monitor='val_loss', factor=0.7,
    patience=60, cooldown=1, min_lr=0.0005,
    min_delta=0.001, verbose=1)

#Callback 3
checkpoint = ModelCheckpoint(
    'best-weights.h5', monitor='val_loss',
    save_best_only=True, save_weights_only=True)
```

Figura 6-16. *Callbacks* empleados en la CNN

6.7.6 *Class Weighting*

Aunque se han ecualizado bastante las distribuciones de las clases entre *Undersampling* y *Oversampling*, aún existen diferencias entre las ocurrencias de las clases más frecuentes y las de las menos frecuentes. Para disminuir el efecto que esto puede producir se recurre a *class weighting.* Todas las muestras, por defecto, tienen la misma importancia para el cálculo de la función de pérdida en cada *step*, necesario para el ajuste de pesos con el descenso del gradiente, sin tener en cuenta su número de instancias por clase. Esto provoca que las clases más representadas contribuyan más a ese cálculo, dando lugar a una búsqueda de pesos centrada en reducir el error de solo esas clases, dejando de lado a las minoritarias. Con *class weighting* esto cambia: cada clase tendrá un peso diferente en dicho cálculo, y este será inversamente proporcional a su número de muestras. De esta forma cada clase tendrá exactamente la misma contribución en el cálculo de la función de coste y los pesos se

actualizarán de manera uniforme para todas las clases. Cada fallo en la predicción penalizará igual, sea cual sea la clase.

La biblioteca Scikit-learn incorpora un método para el cálculo de estos pesos de clase, llamado `compute_class_weight()`. Gracias a que se utilizó la codificación *Label Encoding* para cambiar el rango de las etiquetas, es posible utilizar la mencionada función. En caso contrario, en el que las clases están salteadas, provocaría un error que impediría su uso. Para dicha situación, que tuvo lugar durante el desarrollo del código antes de implementar *Label Encoding*, se diseñaron unas líneas de código que cumplen con el mismo cometido (Código 6-24). El diccionario resultante se le pasa al modelo dentro del método `fit()`.

```
# Método funciona correctamente con este set gracias a LABEL ENCODING
from sklearn.utils import class_weight
classWeights = class_weight.compute_class_weight(class_weight ='balanced',classes = np.unique(y_train), y = y_train)
train_classWeights = dict(enumerate(classWeights))
```

Código 6-23. *Class weighting* con método predefinido

```
# Cálculo de pesos de cada clase de los datos para entrenamiento de la muestra
_, freq = np.unique(y_train, return_counts=True)
max_freq = np.max(freq)
print ('Mayor frecuencia: '+str(np.max(freq)))

#Se realiza este procedimiento si no se usa Label Encoding
dic_class_weights ={}
i=0
for i in range(max(y_train)+1):              #Se va a crear un diccionario de max(y_train)+1 elementos
    dic_class_weights[i] = 0                 #Inicialmente todos los pesos (valores del diccionario) a 0
    if i in y_train:                         #Si la clave coincide con el landmark_id (etiqueta) el peso no será 0
        freq = len(y_train[y_train == i])    #Frecuencia de la etiqueta i
        dic_class_weights[i] = max_freq/freq
```

Código 6-24. *Class weighting* con código propio

6.7.7 Entrenamiento

Para entrenar al modelo se emplea la función `fit()`. Necesita que se le pasen como argumentos los datos de entrenamiento (`X_train, y_train`) y validación (`X_val`, `y_val`). Debe recordarse que las clases siguen estando codificadas con *Label Encoding*, por lo que las etiquetas con las que se opera internamente no se llaman igual que las originales. También se le indica el diccionario con los pesos para el *class weighting*. Puede barajar los datos de entrada, por defecto todo el *dataset* una vez antes de comenzar la primera época. Otro argumento de gran importancia es el hiperparámetro *batch size*, que le indica con cuantas muestras se computa el cálculo del gradiente en *backpropagation* para el algoritmo del descenso del gradiente, es decir, cada cuantas muestras se actualizan los pesos. La cantidad de pasos por época viene dada por el tamaño del *set* de entrenamiento dividido por *batch size*, aunque puede ser modificado por el usuario. Las épocas determinan cuántas veces se entrena con el *set* de entrenamiento por completo. Por último se le activan todos los *callbacks* que se pretendan usar, en este caso tres. Se elige imprimir por pantalla para cada época la barra de progreso y el texto con la métrica y función de pérdida para el entrenamiento y validación. El método `fit()` se asigna a una variable para posteriormente poder representar las gráficas informativas del desempeño del modelo.

```
#Entrenamiento del modelo
history = model.fit(
    X_train, y_train,
    validation_data = (X_val, y_val),       #Etiquetas con Label Encoding
    class_weight = train_classWeights,
    shuffle = True,              #Solo afecta a los datos de entrenamiento (1 vez al principio)
    batch_size = batchSize,
    steps_per_epoch = len(X_train)//batchSize,
    epochs = epochsSize,
    callbacks = [early_stopping, reduce_lr, checkpoint],
    verbose=1,     # 0: silencio    1: barra de progreso + texto    2: solo texto
)
```

Código 6-25. Entrenamiento de la CNN

6.8 Validación de la CNN

Todo el apartado anterior, Creación de la CNN, ha sido implementado dentro de una función, `crear_modelo()`. Las variables globales e hiperparámetros fueron definidos con anterioridad. Tan solo queda llamar a la función para comenzar el entrenamiento. La variable a la que se asigna el método `fit()` contiene toda la evolución de la función de coste y la métrica empleada, tanto durante entrenamiento como validación. Se convierte esta variable en Dataframe y se representa. Gracias a estas gráficas se visualiza la capacidad predictiva del modelo, si tiene un subajuste o sobreajuste, si esta ajustado de manera óptima, si es necesario modificar cierto hiperparámetro... De esta forma el usuario puede elegir entre realizar los cambios oportunos y volver a entrenar el modelo o directamente utilizar el modelo para la predicción en los datos de testeo.

Para la representación de estas gráficas se ha diseñado el siguiente código:

```
start_time = time.time()

model, history = crear_modelo()

history_df = pd.DataFrame(history.history)

print('\nbatchSize = '+str(batchSize))

plt.figure(figsize=(15,5))   #Anchura y Altura de las gráficas, respectivamente

plt.subplot(1,2,1)
#history_df.loc[0:, ['loss', 'val_loss']].plot()   #Utilizar dataframe y su método plot dan problemas con plt.subplot
plt.plot(history.history['loss'], label='Entrenamiento')
plt.plot(history.history['val_loss'], label='Validación')
plt.legend()
plt.title('Loss and Validation Loss')
plt.xlabel('batchSize = '+str(batchSize))
print(("Minimum Validation Loss: {:0.4f} in epoch {:0.0f} ").format(history_df['val_loss'].min(), history_df['val_loss'].idxmin()))

plt.subplot(1,2,2)
#history_df.loc[0:, ['accuracy', 'val_accuracy']].plot()
plt.plot(history.history['sparse_categorical_accuracy'], label='Entrenamiento')
plt.plot(history.history['val_sparse_categorical_accuracy'], label='Validación')
plt.legend()
plt.title('Accuracy and Validation Accuracy')
plt.xlabel('batchSize = '+str(batchSize))
print((("Maximum Validation Accuracy: {:0.4f} in epoch {:0.0f} ").format
      (history_df['val_sparse_categorical_accuracy'].max(), history_df['val_sparse_categorical_accuracy'].idxmax()))

print("\nEvaluación del modelo con datos de entrenamiento")
score = model.evaluate(X_train, y_train)
print("Test loss, Test accuracy:", score[0], score[1])

print("\nEvaluación del modelo con datos de validación")
score = model.evaluate(X_val, y_val)
print("Test loss, Test accuracy:", score[0], score[1])

plt.show()    #Se muestran las gráficas

print('\nDuración del entrenamiento: %s minutos' % ((time.time() - start_time)/60))
```

Código 6-26. Mostrar gráficas para la validación del modelo

Con la combinación de hiperparámetros usados desde el inicio de la explicación del código se obtiene:

```
batchSize = 128
Minimum Validation Loss: 0.2537 in epoch 562
Maximum Validation Accuracy: 0.9714 in epoch 405

Evaluación del modelo con datos de entrenamiento
26/26 [==============================] - 0s 12ms/step - loss: 0.0609 - sparse_categorical_accuracy: 0.9976
Test loss, Test accuracy: 0.06089956685900688 0.9975757598876953

Evaluación del modelo con datos de validación
2/2 [==============================] - 0s 53ms/step - loss: 0.2537 - sparse_categorical_accuracy: 0.9429
Test loss, Test accuracy: 0.2537483870983124 0.9428571462631226
```

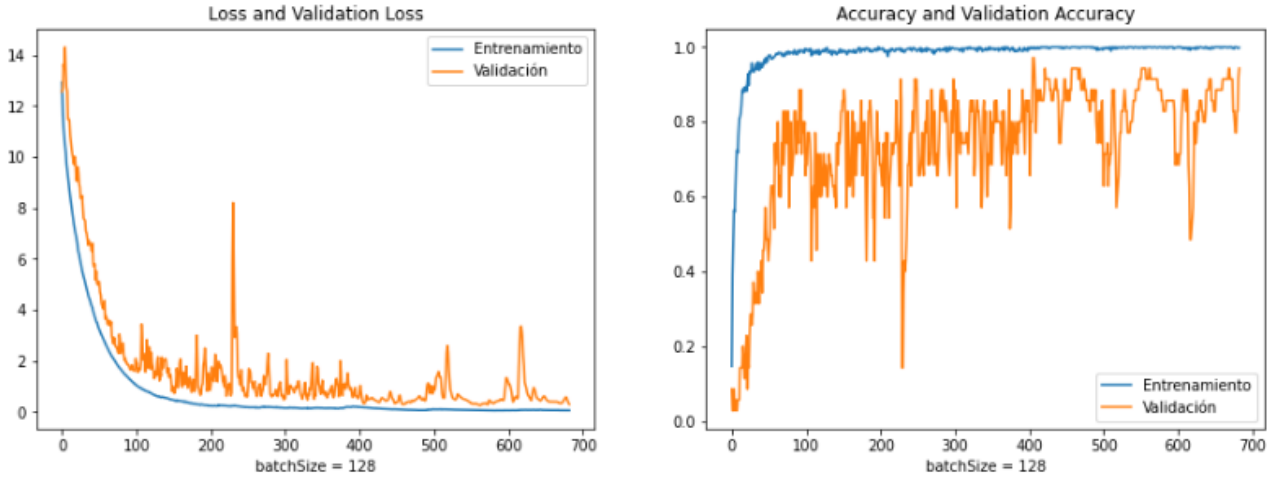


Figura 6-17. Gráficas para hiperparámetros de ejemplo

Para dicho modelo se escogieron las primeras 50 clases del *dataset* proporcionado por *Kaggle*, entre las cuales realmente había 20 clases, redimensionadas más tarde a un tamaño de 128x128x3, habiendo hecho un *Undersampling* con umbral 80 muestras por clase y un *Oversampling* hasta el 50% de dicho umbral, es decir, hasta 40 imágenes por etiqueta. Tras este preprocesado se disponían de 825 imágenes de entrenamiento a partir de las 345 originales. El *learning rate* que se usó fue de 0.01, y se indicó que se entrenara con un tamaño de lote de 128 durante 1000 épocas, aunque el *Early Stopping* saltó en la número 683. Gracias al *Checkpoint*, los pesos que se guardan en el directorio son los que dieron lugar a la menor pérdida durante la validación, indicada encima de las gráficas, en este caso se produjo en la época 562 con un valor de 0.2537.

```
model.load_weights('best-weights.h5')
print('Mejores pesos cargados al modelo')
```

Código 6-27. Cargar mejores pesos al modelo

Estos valores (hiperparámetros) mencionados en el anterior párrafo serán los que se modifiquen para evaluar el rendimiento del modelo, dejando los otros muchos fijados al valor que se han mostrado durante la explicación de esta sección. Esto se hace para disminuir la elevadísima cantidad de combinaciones posibles. Se intentarán ajustar dichos hiperparámetros, buscando la configuración óptima, localmente hablando.

6.9 Predicciones con la CNN

Una vez se tiene un modelo entrenado y se le ha dado luz verde en validación, como es el caso del código desarrollado hasta ahora, se procede con la predicción de las etiquetas de las imágenes del set de testeo.

Primero debe utilizarse el método `predict()` de los objetos de la clase `Sequential` con la que se creó el modelo. Sus argumentos son el set de imágenes de testeo y opcionalmente el si mostrar por pantalla o no información sobre la ejecución del código. Este método devuelve un array en el que cada fila se corresponde con una imagen del set de testeo y cada columna con una clase. Gracias al uso de la función de activación *softmax* en la capa de salida de la CNN, los valores representados en el array son los devueltos por cada neurona final:

la probabilidad de que esa imagen (muestra, fila) corresponda con la etiqueta de la columna (neurona, clase), según el criterio del modelo. La suma de todos los valores de una misma fila da 1.

```
predic = model.predict(X_test, verbose=1)
```

Código 6-28. Predicción de las etiquetas con CNN entrenada

El siguiente paso es crear una nueva variable que almacene las clases predichas para cada muestra. Gracias al uso de *Label Encoding* en las etiquetas durante el procesado de datos, basta con seleccionar el índice de la columna en la que cada fila toma su mayor valor, probabilidad. Estos índices se descodifican con `inverse_transform()` y el objeto `LabelEncoder` y se obtiene la etiqueta correspondiente en la escala real. También se almacenan en otra variable llamada confianza las comentadas mayores probabilidades de cada fila.

```
y_pred = []
confianza = []

for i in range(len(predic)):
    y_pred.append(np.argmax(predic[i]))
    confianza.append(predic[i][y_pred[i]].round(2))

y_pred = LE.inverse_transform(y_pred)
```

Código 6-29. Selección y descodificación de las etiquetas predichas

Después, para mayor interpretación de los resultados, se pueden representar las imágenes a clasificar junto a su clase real (también hay que descodificarlas), su clase predicha y confianza en la predicción para compararlas con las imágenes de la clase predicha visualmente.

```
col = 5  #Número de columnas del subplot (teniendo en cuenta la propia imagen a predecir)
n_pred_rep = 30  #Numero máximo de predicciones a representar

k=0
for k in range(n_pred_rep):
    plt.figure(figsize=(16,7))
    img_pred = X_test[k]
    plt.subplot(1, col, 1)
    plt.xticks([])
    plt.yticks([])
    plt.title('Imagen a predecir nº '+ str(k))
    str1 = 'Clase predicha: ' + str(y_pred[k])
    str2 = 'Confianza: ' + str(confianza[k])
    str3 = 'Clase real: ' + str(y_test[k])
    plt.xlabel(str1 + '\n' + str2 + '\n' + str3, fontsize = 12, weight = 'bold')
    plt.imshow(img_pred)

    i=0
    img_clase_df = traindf[traindf['landmark_id']==y_pred[k]]   #Dataframe con imágenes de igual
                                                                #'landmark_id' que la predicción

    for i in range(len(img_clase_df)):
        if i < (col-1):
            img_clase_path = img_clase_df.iloc[i,2]
            img_clase = img_read_resize(img_clase_path)
            plt.subplot(1, col, i+2)
            plt.xticks([])
            plt.yticks([])
            plt.title('Imagen de la clase '+ str(y_pred[k]))
            plt.xlabel(str())
            plt.imshow(img_clase)
        else:
            break;
```

Código 6-30. Representación de los resultados del testeo

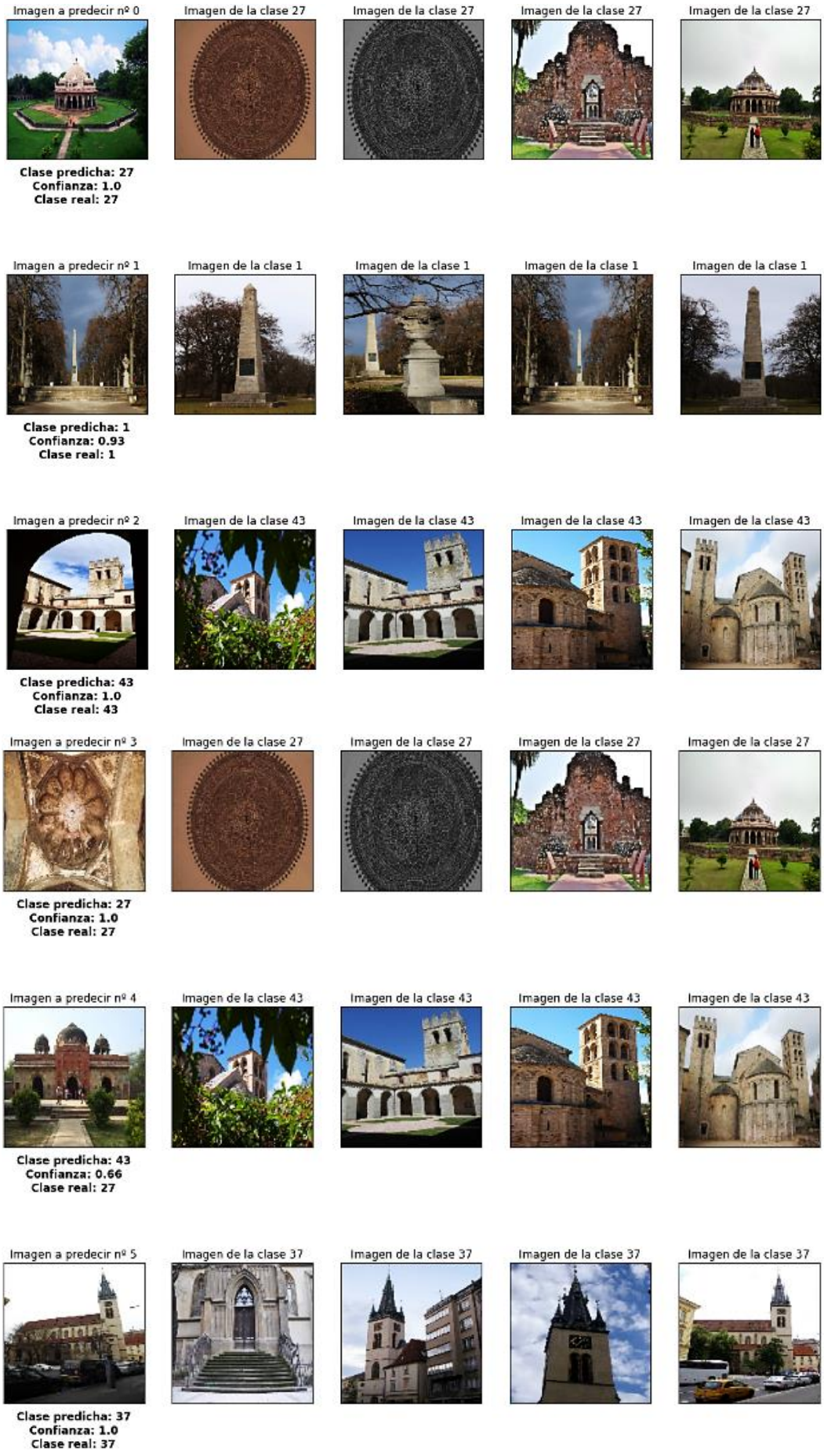


Figura 6-18. Representación visual de la clasificación

En la Figura 6-18 se han representado los resultados obtenidos en 6 muestras. La clasificación es correcta para la gran mayoría de ellas, consiguiendo o rozando una confianza de valor unidad, lo que implica que no hay duda alguna del modelo en dichas predicciones, algo que se podía prever dado el modelo y técnicas empleadas y la poca cantidad de clases en la muestra usada. Sin embargo, se observa un error en la asignación de etiquetas en el penúltimo caso, confundiendo la clase 27 con la 43, visible en una baja confianza en comparación con los demás casos. Este error se debe principalmente al *dataset* empleado. Para una misma etiqueta, por lo general, la cantidad de imágenes es diminuta, y lo que es más, hay muestras nada similares a otras. Las imágenes de una clase no solo varían en perspectiva, ángulo o distancia de la fotografía, sino que se representan distintas caras de un mismo lugar emblemático, tanto exteriores como interiores. Esto implica patrones muy diferentes dentro de una misma etiqueta. Si la muestra a clasificar de cierta clase es una totalmente distinta a las demás imágenes con las que se ha entrenado para dicha clase, los patrones detectados por la CNN pueden ser clasificados como los correspondientes a otra clase diferente. Esto está relacionado con el azar del *data splitting*, pudiendo dividir el *dataset* original de tal forma que las muestras más homogéneas entre sí de un etiqueta pero que más difieren con las demás de dicha clase sean agrupadas todas fuera del set de entrenamiento, causando que sus posteriores predicciones sean posiblemente erróneas.

Otro apoyo visual que resulta de utilidad para evaluar la precisión es la Matriz de confusión, cuya implementación es facilitada por la biblioteca *Seaborn*. Para representarla primero es necesario definir una variable a la que asignarle las clases usadas en el testeo, tanto las predichas como las etiquetas reales, y ordenadas de menor a mayor. Después se utiliza el método `confusion_matrix()` de la biblioteca de métricas de *Scikit-learn,* que devuelve un array cuadrado de dimensión igual a la variable anterior. Este array se convierte en un Dataframe y posteriormente se pasa como argumento a la función de *Seaborn* `heatmap()`. Con esta se puede elegir el estilo de la matriz de confusión: tipo de dígitos, gama de colores, anotaciones... El objetivo de un modelo es tener únicamente la diagonal de dicha matriz poblada de muestras, pues simbolizan aciertos en la clasificación, mientras que el resto de las casillas se desea que valgan cero, o sea, sin ocurrencias, pues son los errores.

```
clases_usadas = np.append(np.unique(y_test), np.unique(y_pred))    #Todas las clases usadas en predicción y testeo (algunas repetidas)
clases_unicas_usadas = np.unique(clases_usadas)                    #Se eliminan las repetidas y se ordenan de menor a mayor

cm = confusion_matrix(y_test, y_pred)  #Número de filas y columnas = número de clases
                                       #distintas usadas tanto en y_test como y_pred (orden menor a mayor)
df_mat_conf = pd.DataFrame(cm, columns=clases_unicas_usadas, index=clases_unicas_usadas)

figure = plt.figure(figsize=(18, 13))
plt.title("MATRIZ DE CONFUSIÓN", fontsize=16, fontweight='bold')

s=sns.heatmap(df_mat_conf, annot=True, cmap='Blues', fmt="d")

plt.tight_layout()
plt.xlabel("Clase predicha", fontsize=16, fontweight='bold')
plt.ylabel("Clase real", fontsize=16, fontweight='bold')
plt.show()
```

Código 6-31. Implementación de la matriz de confusión del set de testeo

La matriz obtenida para el ejemplo que se lleva tratando es la mostrada en Figura 6-19. Como se ha comentado, solo se representan las clases que participan en el testeo, tanto las reales como las predichas. Habrá clases de la muestra que no se empleen en el testeo debido al azar del *data splitting* y los porcentajes elegidos.

Se puede ver que la mayoría de las etiquetas fueron predichas correctamente: de las 16 muestras de testeo (4.5% de las imágenes de la muestra inicial), solo 3 predicciones han sido erróneas, están fuera de la diagonal, lo que se traduce en un 84.21% de precisión. Como observación, la clase de la que se disponían más instancias iniciales distintas, la 27, tiene 5 verdaderos positivos, 1 falso negativo y 1 falso positivo, usando como referencia la Figura 3-11.

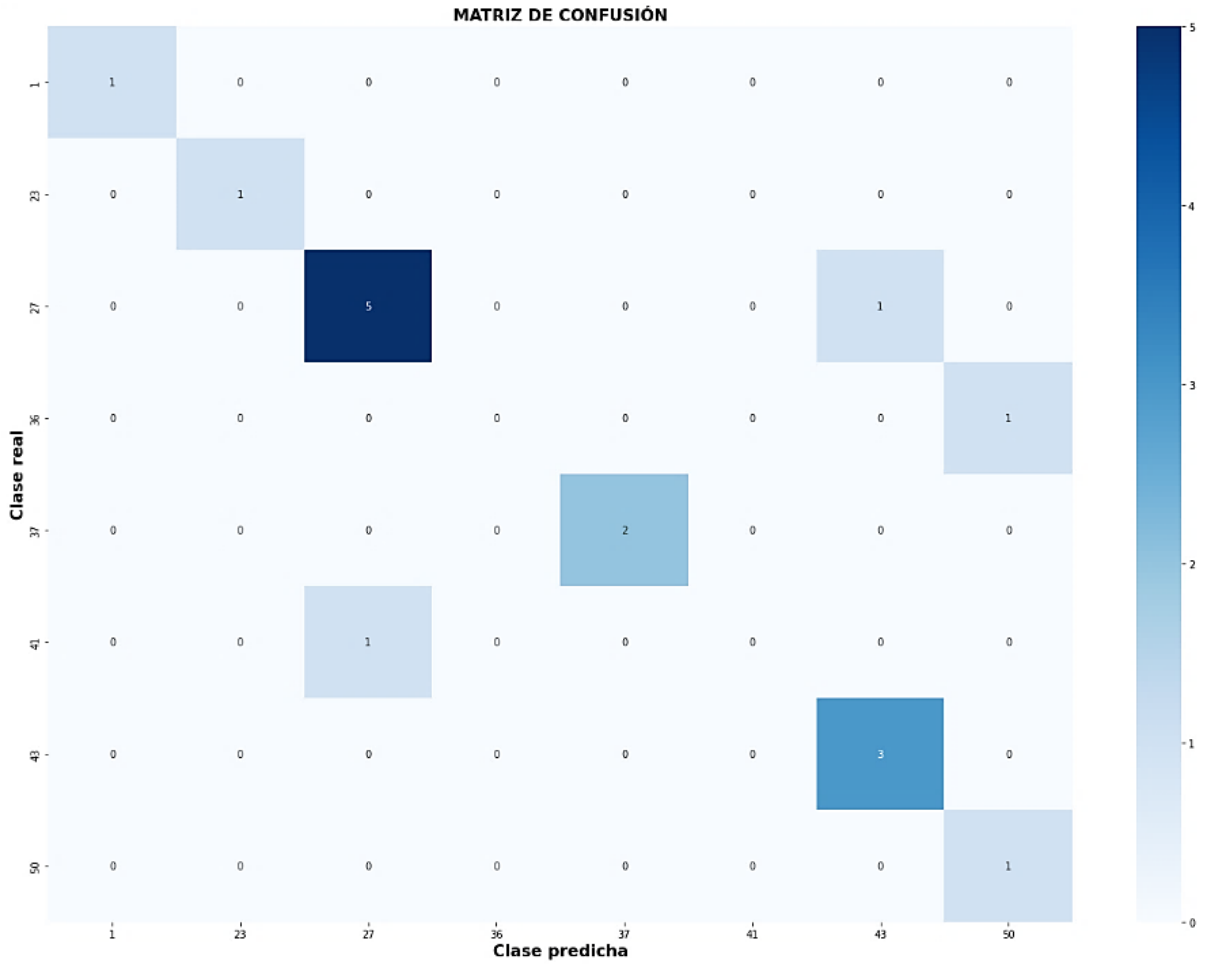


Figura 6-19. Matriz de confusión del modelo elaborado desde cero

6.10 Transfer Learning

La alternativa a la creación de un modelo desde cero y la forma de proceder con más éxito en problemas de *Deep Learning* es el conocido como *Transfer Learning*. Con esta técnica es posible utilizar como punto de partida modelos ya pre-entrenados en grandes cantidades de datos y con una capacidad predictiva excelente, para darles uso en problemas similares, reduciendo así el procesamiento requerido.

La descarga en el código de la estructura de reconocidos modelos de CNN con o sin sus pesos es factible gracias a la biblioteca `keras.applications`. Están disponibles modelos de renombre como Xception, VGG16, ResNet-50, Inception V3, etc. Se pueden consultar en la documentación de *Keras* (Figura 6-20) [95]. El que estos modelos ya estén entrenados supone una gran ventaja: los pesos ya están ajustados a un *dataset*, por lo que el procesado (computación de gradientes) no es necesario si el nuevo *dataset* tiene patrones similares. Si ese *dataset* es parecido o tiene alguna relación con el *dataset* del problema a resolver, las características de las imágenes serán similares, por lo que el *backbone* del modelo debería poseer la misma capacidad extractora de *features* para ambos *datasets*. En este caso, la única modificación a realizar sería cambiar o ajustar la RNA densamente conectada del final de la CNN, pues su función es la de clasificar los vectores de características en clases, que lo más probable es que cambien de un problema a otro. En resumen, se reutilizará el *backbone* del modelo pre-entrenado y se harán cambios solo en su *head*. También es común modificar además algunos pesos de las capas convolucionales con un *learning rate* bajo para lograr una mejor adaptación al nuevo *dataset*.

Model	Size (MB)	Top-1 Accuracy	Top-5 Accuracy	Parameters	Depth	Time (ms) per inference step (CPU)	Time (ms) per inference step (GPU)
Xception	88	79.0%	94.5%	22.9M	81	109.4	8.1
VGG16	528	71.3%	90.1%	138.4M	16	69.5	4.2
VGG19	549	71.3%	90.0%	143.7M	19	84.8	4.4
ResNet50	98	74.9%	92.1%	25.6M	107	58.2	4.6
ResNet50V2	98	76.0%	93.0%	25.6M	103	45.6	4.4
ResNet101	171	76.4%	92.8%	44.7M	209	89.6	5.2
ResNet101V2	171	77.2%	93.8%	44.7M	205	72.7	5.4
ResNet152	232	76.6%	93.1%	60.4M	311	127.4	6.5
ResNet152V2	232	78.0%	94.2%	60.4M	307	107.5	6.6
InceptionV3	92	77.9%	93.7%	23.9M	189	42.2	6.9
InceptionResNetV2	215	80.3%	95.3%	55.9M	449	130.2	10.0

Figura 6-20. Algunos modelos disponibles para *Transfer Learning* [95]

Para adaptar el modelo a nuestro problema, a nuestras etiquetas, se ha decidido diseñar y se entrenar solo las capas encargadas de la clasificación. Para ello, las capas extractoras de características se congelan, se bloquea la modificación de sus pesos para mantenerlos intactos. Durante la fase de aprendizaje, solo la *head* configurará sus pesos permitiendo la clasificación en las etiquetas correspondientes. Todos los modelos de *Keras* han sido entrenados en la enorme base de datos de Google ImageNet, que contiene imágenes de cualquier ámbito. Se espera que las características extraídas por sus *backbone* sean compatibles con las imágenes del *dataset* de lugares emblemáticos proporcionado por *Kaggle*. Si esto no sucediera, habría que entrenar también, parcial o totalmente, el *backbone*, perdiendo así una de las ventajas del uso de *Transfer Learning*, y cuya realización podría ser imposible con los recursos computacionales que se poseen dada la gran cantidad de parámetros de estas CNN. Cada modelo requiere además un preprocesamiento específico de sus datos cuya incorporación al código es facilitada por el método `preprocess_input`, disponible dentro de la biblioteca `keras.applications` del modelo correspondiente.

6.10.1 VGG16 [96]

Como guía para la implementación del *Transfer Learning*, se utilizará en primer lugar el modelo pre-entrenado VGG16. Es un modelo de CNN propuesto por Karen Simonyan y Andrew Zisserman para una competición de reconocimiento de imágenes de la base de datos ImageNet en 2014. El número 16 de su nombre proviene del número de capas convolucionales y densas que emplea. Destacó por el sucesivo uso de filtros lo más pequeños posibles en las capas convolucionales, de dimensión 3x3, con desplazamiento (*stride*) de 1 píxel, porque en ese entonces los modelos más exitosos, AlexNet y ZFNet, tenían un tamaño de 11x11 con *stride* 4 y 7x7 con *stride* 2, respectivamente. Mediante varios de estos *kernels* de tamaño 3x3 se puede replicar el comportamiento de filtros mayores: dos seguidos actúan como uno 5x5, tres seguidos como uno 7x7... La gran ventaja es que aumenta el número de capas sin utilizar más parámetros (pesos y sesgos) por lo que se utilizan más funciones de activación no lineales y con menos parámetros, lo que incrementa la capacidad discriminante del modelo a la vez que aumenta su eficiencia computacional [97]. La estructura VGG16 original tiene como entrada imágenes de resolución 224x224 píxeles. El *backbone* alterna conjuntos de las mencionadas capas convolucionales con una de *max pooling*. Emplea dos capas ocultas densas para la clasificación, de 4096 neuronas cada una, y capa de salida con *softmax* de 1000 neuronas, pues 1000 eran las etiquetas de la competición. Como aspecto negativo, puede resaltarse su lentitud a la hora de aprender por el uso de tantas capas y con pesos tan grandes (no usa capas de *Batch Normalization*), aunque en la actual implementación no afecta pues se congelan los pesos de los filtros y se sustituye su *head*.

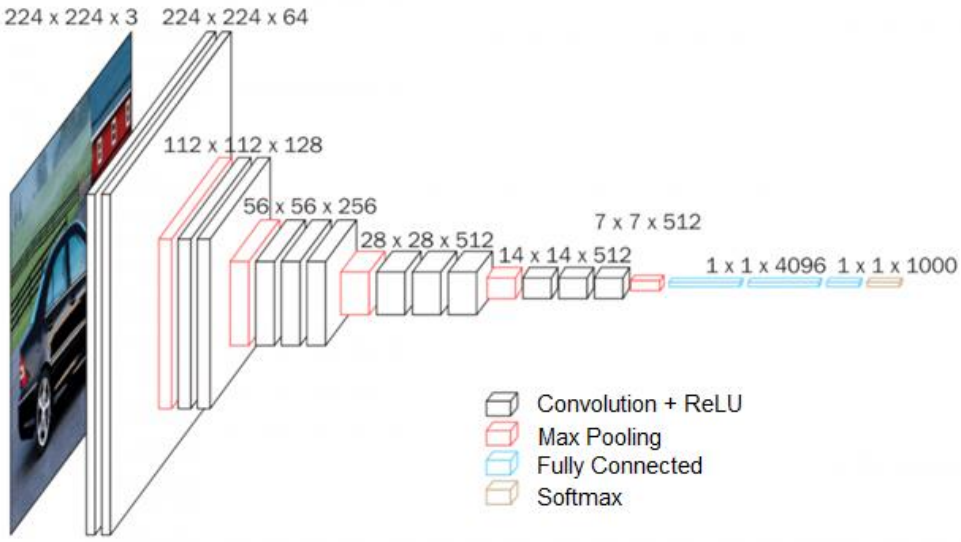


Figura 6-21. Arquitectura VGG16 [97]

En cuanto a los datos, el preprocesamiento necesario consiste en convertir las imágenes de RGB a BGR y restar la media de un plano de color a todos los píxeles de dicho plano, sin escalar.

A continuación, se va a realizar el *Transfer Learning* de VGG16 sobre el código elaborado desde cero, por lo que son necesarios algunos cambios. Primero, aplicar en todos los datos que se vayan a pasar por el modelo (entrenamiento, validación y testeo) el método `preprocess_input` propio de VGG16, eliminando antes cualquier otro procesamiento previo de las imágenes como su escalado Min-Max, que causa resultados erróneos.

```
from keras.applications.vgg16 import preprocess_input

X_train = preprocess_input(X_train)
X_val = preprocess_input(X_val)
X_test_orig = X_test
X_test = preprocess_input(X_test)
```

Código 6-32. Preprocesamiento necesario en VGG16

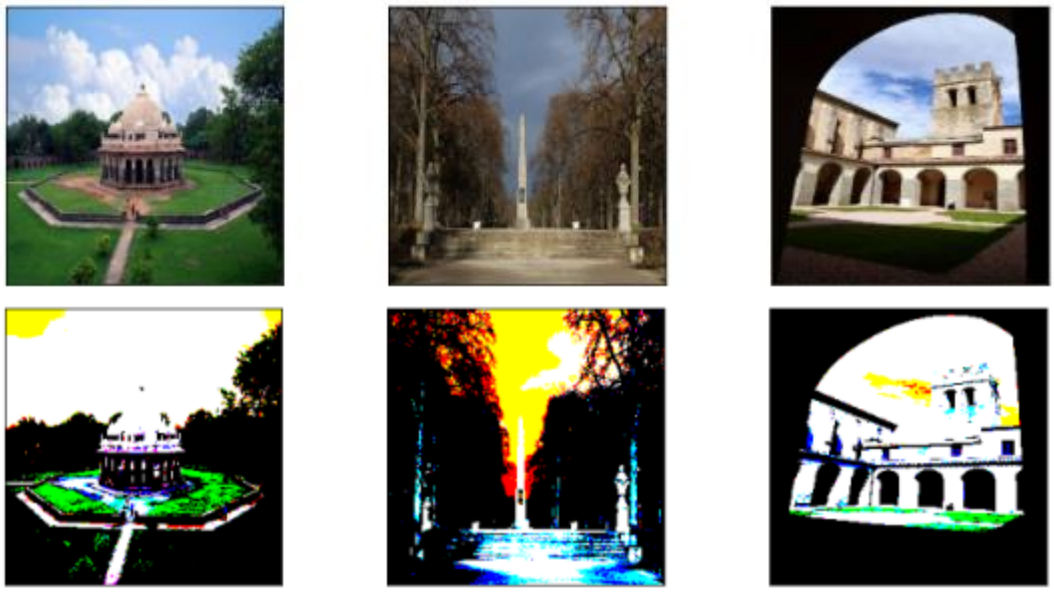


Figura 6-22. Imágenes originales e imágenes preprocesadas para VGG16

Donde antes se creó el modelo desde cero, ahora se llama a la función `VGG16` dentro de `keras.applications`. El argumento `include_top` se establece como `False` para no descargar las últimas 3 capas del modelo, que se quieren eliminar. Se indica que los pesos del *backbone* del modelo sean los obtenidos tras entrenar con el *dataset* ImageNet. La capa de entrada de la CNN tendrá el tamaño de las imágenes del *dataset* con el que se vaya a trabajar, no tienen por qué ser 224x224 como se planeó en un inicio para la estructura VGG16. Después, con el método `trainable` a *False* del objeto `VGG16` se congelan los pesos de las capas convolucionales. De esta forma se descarga el *backbone* de VGG16 en el código. Ahora hay que insertarlo en otro modelo. En este caso se mantienen las capas de *Data Augmentation*, se añaden las capas convolucionales del VGG16 y una nueva RNA para la clasificación, esta vez compuesta por dos capas ocultas densas de 512 neuronas cada una y una capa de salida de neuronas igual al número de etiquetas de la clasificación. Además, se insertan entre ellas capas de *Dropout* y *Batch Normalization* para mejorar la generalización y eficiencia.

```
VGG16 = (keras.applications.VGG16(include_top=False,
            weights='imagenet', input_shape=([IMG_SIZE, IMG_SIZE, 3])))

VGG16.trainable=False

model = keras.Sequential()

model.add(preprocessing.RandomZoom(height_factor=(-0.2, 0.2)))
model.add(preprocessing.RandomContrast(0.4))
model.add(preprocessing.RandomTranslation(height_factor= (-0.2, 0.2), width_factor=(-0.2, 0.2)))
model.add(preprocessing.RandomRotation(factor= (-0.1, 0.1)))

model.add(VGG16)

model.add(layers.Flatten())
model.add(layers.Dense(units = 512, activation=activationF, kernel_regularizer=weightDecay))
model.add(layers.Dropout(rate=dropoutRate,seed=SEED))
model.add(layers.BatchNormalization())
model.add(layers.Dense(units = 512, activation=activationF, kernel_regularizer=weightDecay))
model.add(layers.Dropout(rate=dropoutRate,seed=SEED))
model.add(layers.BatchNormalization())
model.add(layers.Dense(units = N_CLASES, activation='softmax'))
```

Código 6-33. Nuevo modelo basado en Transfer Learning con VGG16

Con solo estos minúsculos cambios en el modelo elaborado desde cero se implementa la técnica de *Transfer Learning*, pudiendo mejorar el rendimiento del modelo y disminuir el tiempo de entrenamiento. Como muestra de ello, se realiza exactamente el mismo entrenamiento con los mismos hiperparámetros que el visto en 6.8 y 6.9. El resultado es bastante superior, de hecho, es perfecto en el set de testeo para ese número de clases.

```
batchSize = 128
Minimum Validation Loss: 0.9598 in epoch 303
Maximum Validation Accuracy: 0.9714 in epoch 99

Evaluación del modelo con datos de entrenamiento
26/26 [==============================] - 7s 159ms/step - loss: 0.8598 - sparse_categorical_accuracy: 1.0000
Test loss, Test accuracy: 0.8597940802574158 1.0

Evaluación del modelo con datos de validación
2/2 [==============================] - 1s 466ms/step - loss: 0.9598 - sparse_categorical_accuracy: 0.9714
Test loss, Test accuracy: 0.9597885012626648 0.9714285731315613
```

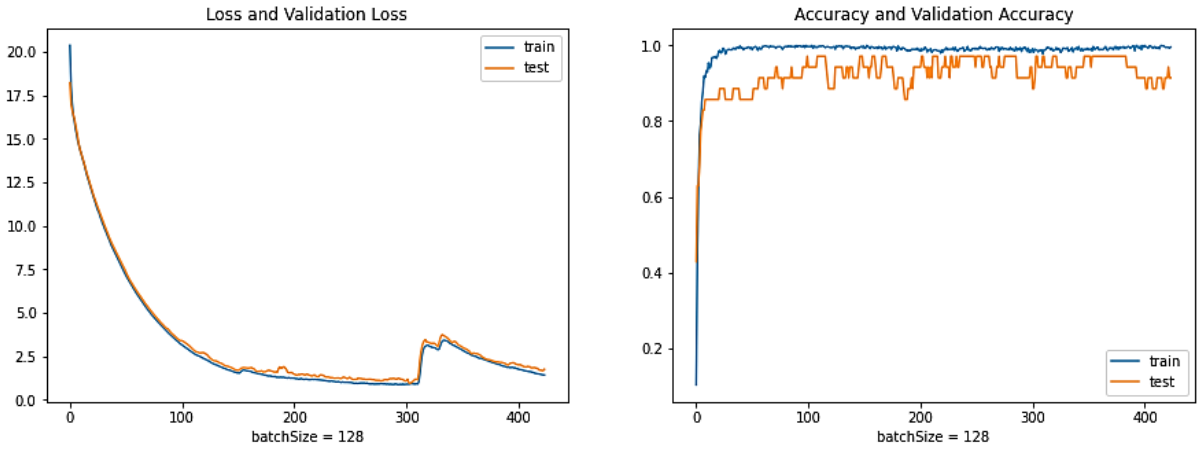


Figura 6-23. Resultado *Transfer Learning* VGG16

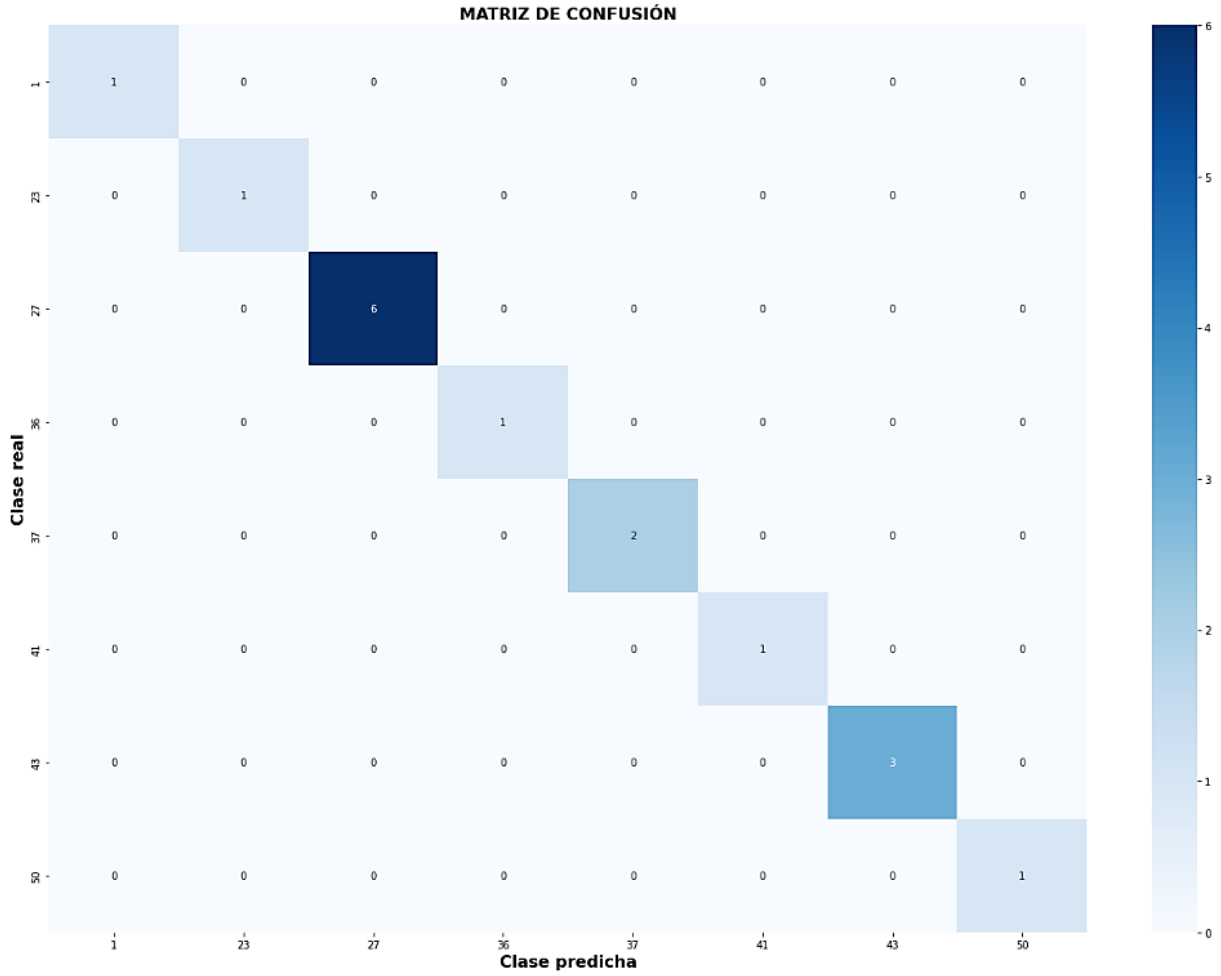


Figura 6-24. Matriz de confusión *Transfer Learning* VGG16

Nótese que la gráfica de la función de coste de entrenamiento y validación tiene un aumento de la pérdida debido quizás a un *learning rate* demasiado alto o a un entrenamiento excesivamente largo. Esto no es problema, pues el *Early Stopping* actúa y almacena los mejores pesos, en este caso justo los previos a dicho aumento de la pérdida, que provocaron 0.9598 de pérdida en validación en la época 303.

6.10.2 ResNet-50 [98]

Los modelos ResNet hacen referencia a *Residual Neural Network* y fueron propuestos por Kaiming He y sus compañeros en 2015, resultando ganador de una competición de ImageNet. Son RNA unidireccionales muy profundas que emplean atajos o puentes, conexiones adicionales de un bloque de capas a otro bloque no inmediatamente posterior. Los bloques de capas reciben el nombre de bloques residuales. Esta estructura fue desarrollada para arreglar un contratiempo en las redes con tantas capas. El problema que reduce es el del desvanecimiento del gradiente, por el que no se actualizan correctamente los pesos de las primeras capas debido a que el valor del gradiente calculado con *backpropagation* mediante regla de la cadena se va haciendo cada vez más pequeño a medida que se avanza hacia atrás, por usar funciones de activación con derivadas muy cercanas a cero para algunos valores y demasiadas capas.

El número de capas, convolucionales más la de salida, de las ResNet varían de 18 a 152, usando bloques residuales de 2 o 3 capas convolucionales. El modelo ResNet-50 emplea en 49 capas convolucionales, siendo la restante la *output layer* para la clasificación. Sus bloques de capas están formados por 3 capas convoluciones. Como problema puede destacarse su largo tiempo de entrenamiento por la gran cantidad de capas, aunque con el actual *Transfer Learning* no afecta.

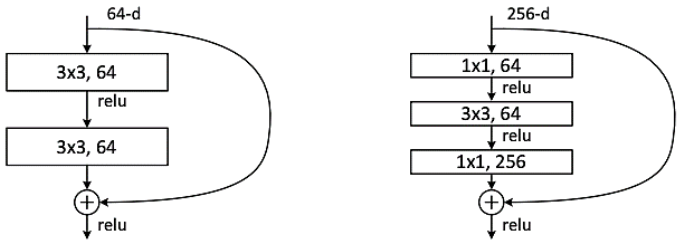


Figura 6-25. Bloques Residuales en ResNet: 2 capas (izquierda) y 3 capas (derecha) [99]

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		[3×3, 64; 3×3, 64] ×2	[3×3, 64; 3×3, 64] ×3	[1×1, 64; 3×3, 64; 1×1, 256] ×3	[1×1, 64; 3×3, 64; 1×1, 256] ×3	[1×1, 64; 3×3, 64; 1×1, 256] ×3
conv3_x	28×28	[3×3, 128; 3×3, 128] ×2	[3×3, 128; 3×3, 128] ×4	[1×1, 128; 3×3, 128; 1×1, 512] ×4	[1×1, 128; 3×3, 128; 1×1, 512] ×4	[1×1, 128; 3×3, 128; 1×1, 512] ×8
conv4_x	14×14	[3×3, 256; 3×3, 256] ×2	[3×3, 256; 3×3, 256] ×6	[1×1, 256; 3×3, 256; 1×1, 1024] ×6	[1×1, 256; 3×3, 256; 1×1, 1024] ×23	[1×1, 256; 3×3, 256; 1×1, 1024] ×36
conv5_x	7×7	[3×3, 512; 3×3, 512] ×2	[3×3, 512; 3×3, 512] ×3	[1×1, 512; 3×3, 512; 1×1, 2048] ×3	[1×1, 512; 3×3, 512; 1×1, 2048] ×3	[1×1, 512; 3×3, 512; 1×1, 2048] ×3
	1×1	average pool, 1000-d fc, softmax				

Tabla 6-2. ResNet existentes según el número de capas convolucionales [99]

Para su implementación tan solo hay que sustituir `vgg16` en el Código 6-32 por `resnet` y `VGG16` en el Código 6-33 por `ResNet50`, para usar la biblioteca correspondiente al modelo de la red residual. Se tienen exactamente los mismos argumentos, por lo que no es necesario ninguna modificación adicional. El preprocesado de imágenes necesario, `keras.applications.resnet`, hace exactamente el mismo que VGG16 (Figura 6-22): de RGB a BGR y centrar en 0 cada plano de color. Se ha decidido mantener la cabeza del modelo anterior: dos capas densas de 512 neuronas, con *Dropout* y *Batch Normalization*, y *output layer* de `N_CLASES` con *softmax*.

```
batchSize = 128
Minimum Validation Loss: 0.4245 in epoch 641
Maximum Validation Accuracy: 1.0000 in epoch 125

Evaluación del modelo con datos de entrenamiento
26/26 [==============================] - 2s 67ms/step - loss: 0.3826 - sparse_categorical_accuracy: 1.0000
Test loss, Test accuracy: 0.3825766146183014 1.0

Evaluación del modelo con datos de validación
2/2 [==============================] - 0s 171ms/step - loss: 0.4245 - sparse_categorical_accuracy: 1.0000
Test loss, Test accuracy: 0.4244561493396759 1.0
```

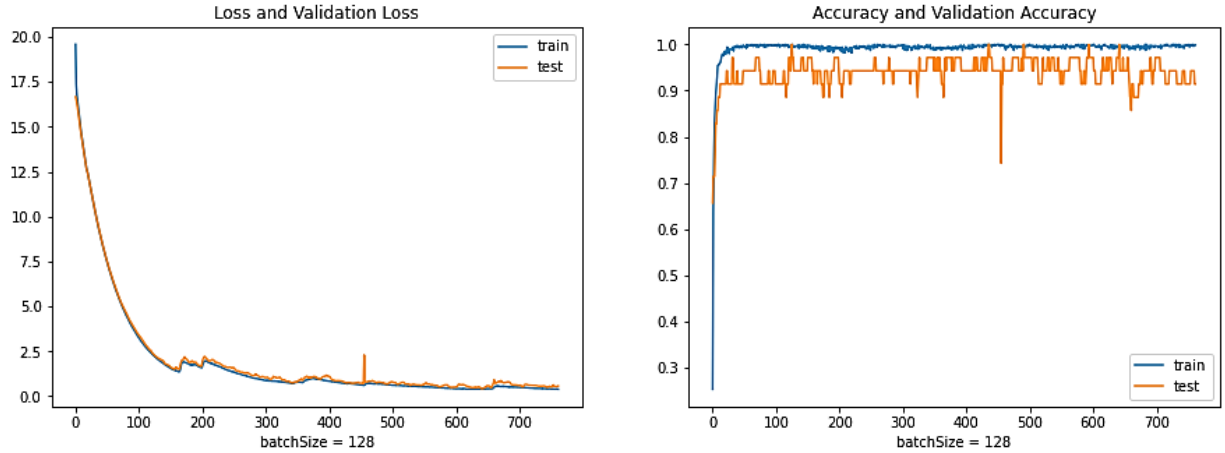


Figura 6-26. Resultado *Transfer Learning* ResNet-50

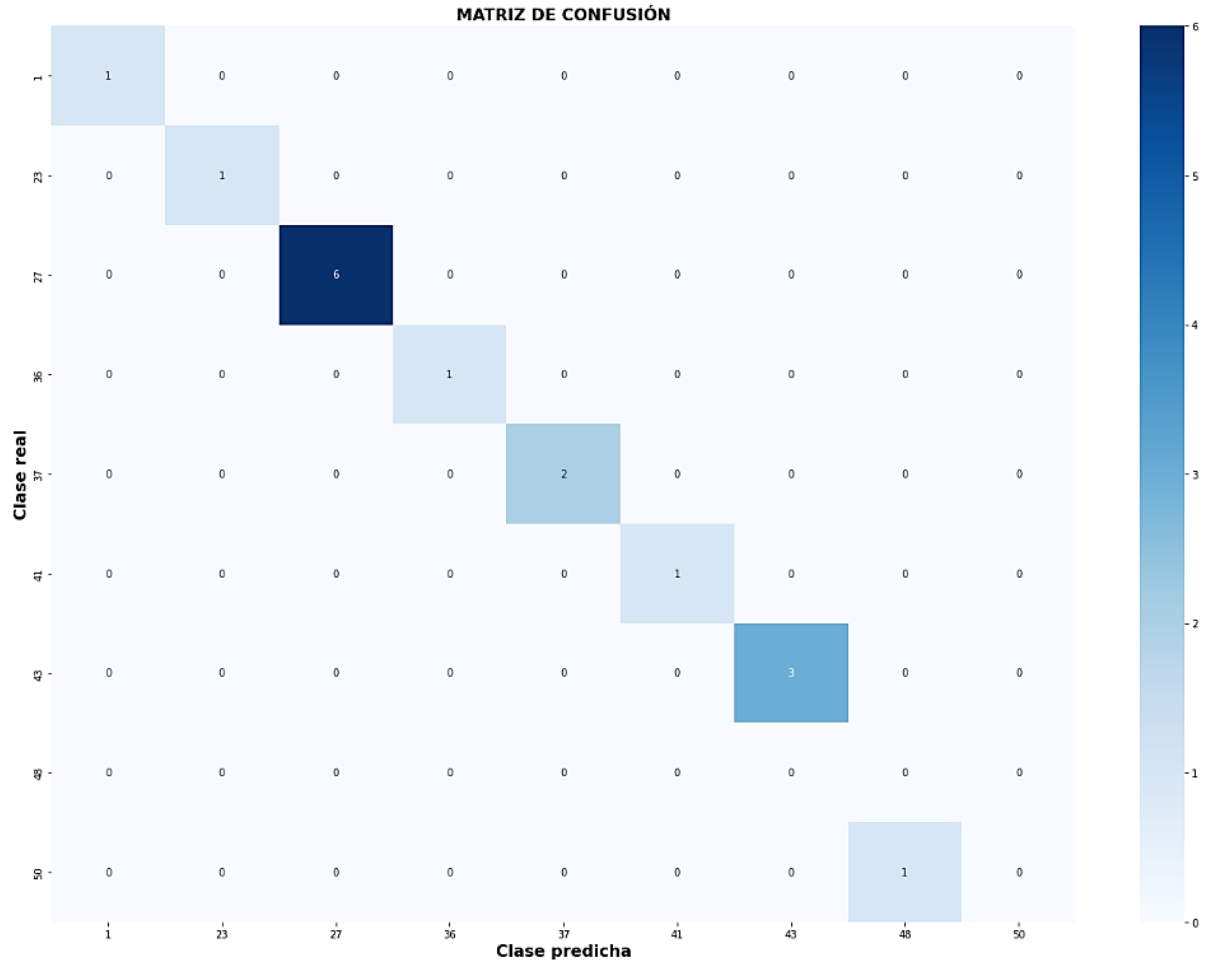


Figura 6-27. Matriz de confusión *Transfer Learning* ResNet-50

Aunque se haya realizado una predicción incorrecta en el *dataset* de testeo, el entrenamiento, si bien ha durado más tiempo que el de VGG16 (por no activarse *Early Stopping* antes), ha dado mejores resultados que dicho modelo en validación, logrando en la época 125 un 100% de precisión en el set de validación y en la época 641 tan solo 0.4245 de pérdida en la validación. Los pesos de este último valor son los que se almacenan tras la activación del *Early Stopping*.

6.10.3 Inception V3 [100]

Los modelos Inception son CNN desarrolladas por Google desde 2014. Al primer modelo, Inception V1, se le presentó como GoogLeNet. La tercera versión de estas RNA es el Inception V3, disponible desde 2015. En comparación con sus predecesores, Inception V3 tiene una mayor eficiencia, requiere menor potencia computacional, tiene más capas, aunque su velocidad de procesamiento sigue siendo la misma, y además, consigue una mayor precisión. Estas ventajas son consecuencia de algunas técnicas de optimización empleadas. La primera de ellas es, como introdujo VGG16, el uso de filtros convolucionales de tamaño 3x3, que como se vio, requieren menos parámetros y aumentan la discriminación entre clases al añadir más no linealidad (más capas, más funciones de activación no lineales). La siguiente implementación es el uso de filtros convolucionales asimétricos, es decir, de tamaño *n*x1, y siguiendo la misma estrategia anterior, sustituir los filtros 3x3 por un filtro 1x3 y otro 3x1. El otro método empleado es el uso de clasificadores auxiliares para reducir el efecto del problema del desvanecimiento del gradiente, utilizados como regularizadores en Inception V3 [101]. En cuanto al preprocesamiento necesario a las imágenes de entrada, se requiere meramente un escalado de los píxeles al rango [-1, 1].

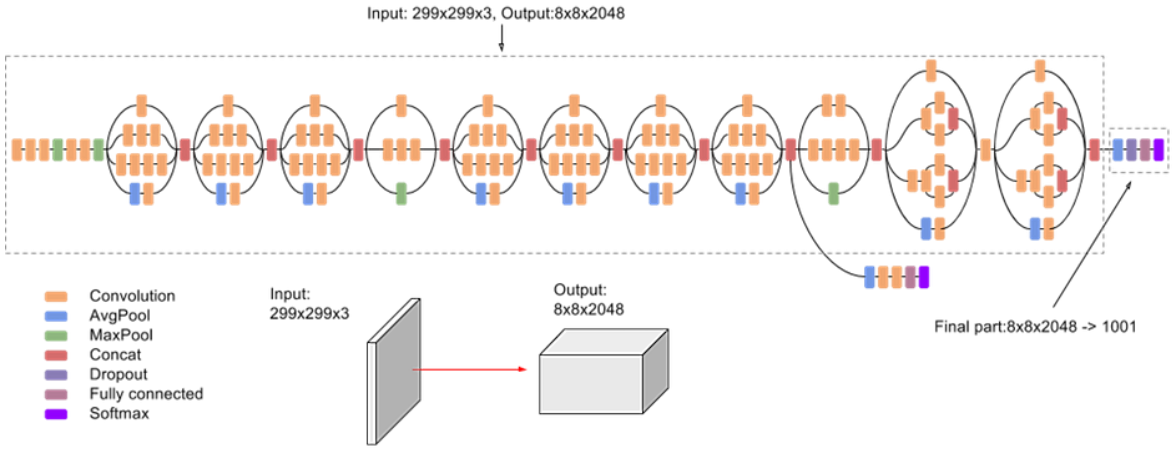


Figura 6-28. Arquitectura Inception V3 [102]

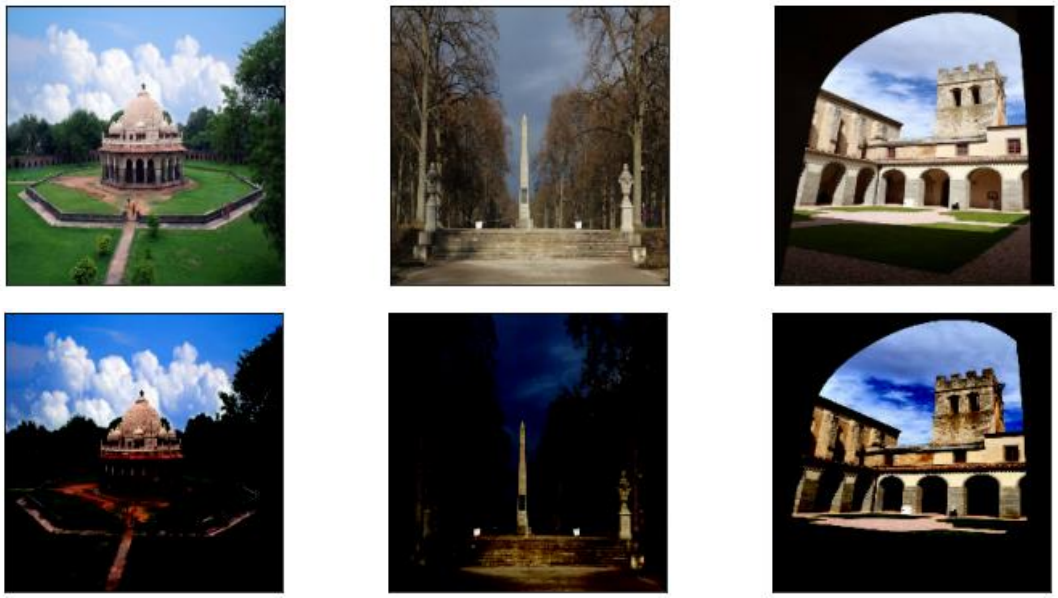


Figura 6-29. Imágenes originales e imágenes preprocesadas para Inception V3

Como ya se ha visto, para su implementación en el código con finalidad de *Transfer Learning* hay que aplicar la función de preprocesado a las imágenes e incorporar su estructura al modelo. Partiendo de los modelos anteriores de VGG16 y ResNet-50, de nuevo basta con cambiar unas pocas líneas, concretamente insertar como biblioteca de preprocesamiento `keras.applications.inception_v3` y el modelo, con exactamente los mismos argumentos ya vistos, desde `keras.applications.InceptionV3`. De nuevo, se añaden dos capas densas de 512 neuronas y *output layer* de `N_CLASES` neuronas con función de activación *softmax*. Se entrena el modelo resultante con los mismos hiperparámetros empleados hasta ahora. Se observa que el tiempo por *step* es aproximadamente 175 ms, bastante menor que el de VGG16 y ResNet-50 que sí que son similares entre ellos, con 250 ms. A pesar de que Inception V3 es una red más profunda, tiene menos parámetros que las otras dos CNN, demostrando la utilidad de las técnicas elegidas para la reducción del número de pesos.

Los resultados de este modelo se muestran en las siguientes figuras. Su rendimiento en el set de testeo es igual que el modelo con ResNet-50, fallando en una sola muestra. Al examinar su pérdida y precisión en validación, se podría decir que su desempeño está en un término medio entre los dos modelos anteriores. En ambas gráficas de validación se presentan dos picos muy pronunciados (con respecto a los valores próximos), debido posiblemente a unos lotes de imágenes más complicados de clasificar con los pesos y sesgos actualizados al acabar de entrenar en dichas épocas. Estos son unos fenómenos puntuales que en este caso no afectan al resultado del modelo, pues se recupera instantáneamente.

```
batchSize = 128
Minimum Validation Loss: 0.8201 in epoch 521
Maximum Validation Accuracy: 0.9714 in epoch 190

Evaluación del modelo con datos de entrenamiento
26/26 [==============================] - 2s 63ms/step - loss: 0.5836 - sparse_categorical_accuracy: 0.9976
Test loss, Test accuracy: 0.5835674405097961 0.9975757598876953

Evaluación del modelo con datos de validación
2/2 [==============================] - 0s 286ms/step - loss: 0.8201 - sparse_categorical_accuracy: 0.9143
Test loss, Test accuracy: 0.8200865387916565 0.9142857193946838
```

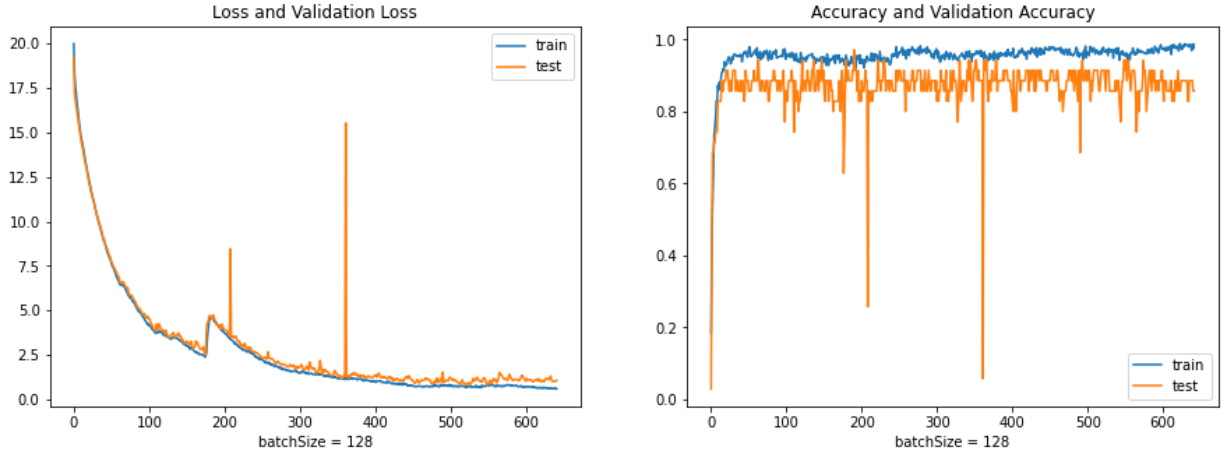


Figura 6-30. Resultado *Transfer Learning* Inception V3

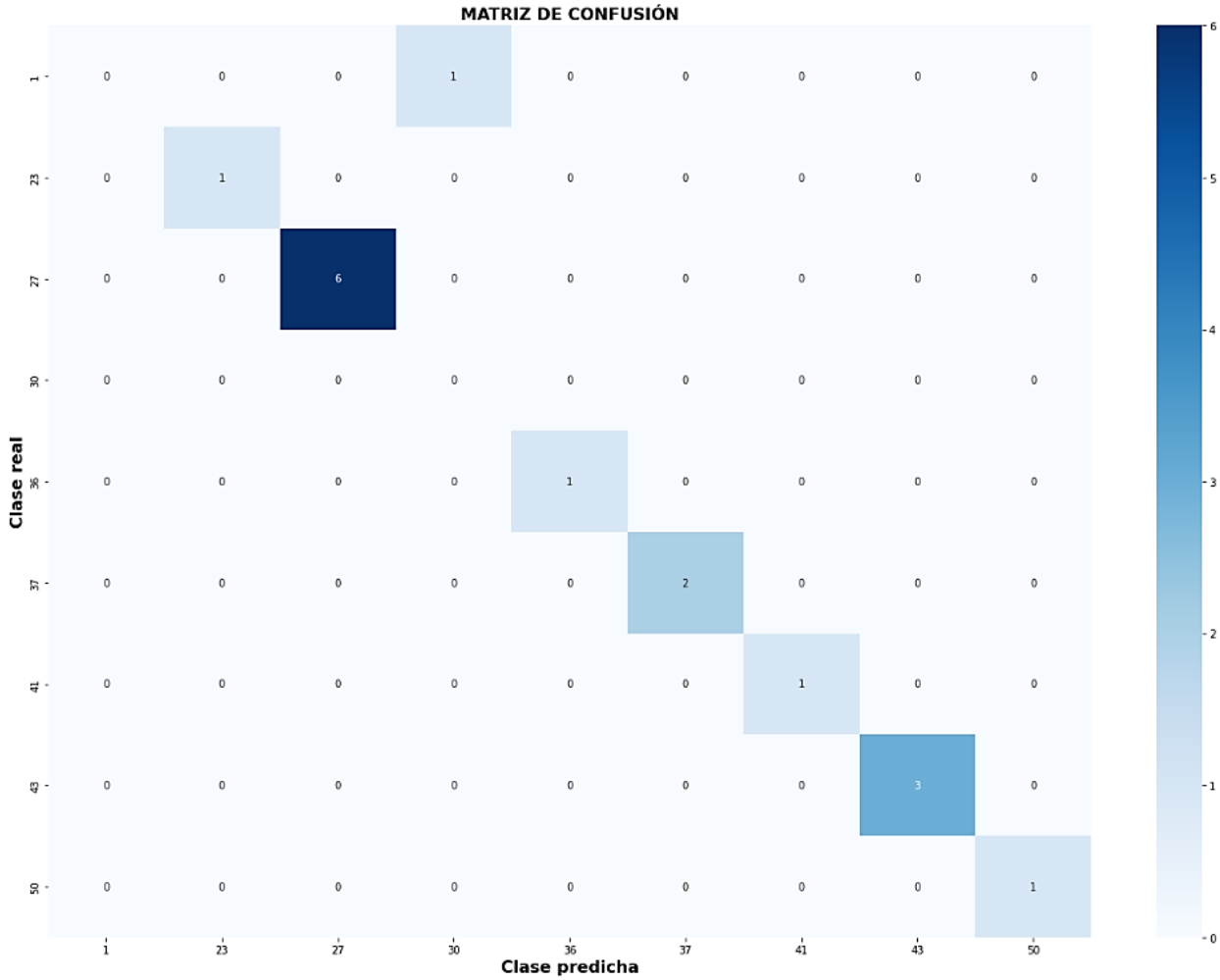


Figura 6-31. Matriz de confusión *Transfer Learning* Inception V3

6.11 Comparación de resultados

En este apartado se van a comparar varias combinaciones de hiperparámetros para la CNN elaborada desde cero y las CNN obtenidas con *Transfer Learning*. Se entrenan con la GPU de Kaggle (Tabla 6-1). Para todas las pruebas se van a mantener constantes los siguientes hiperparámetros, salvo que se indique lo contrario:

- Muestra dividida en: 85.5% entrenamiento, 10% validación, 4.5% testeo.
- Umbral *Undersampling* Muestra: 80 imágenes.
- Función de pérdida: *'sparse_categorical_crossentropy'*.
- Métrica: *'sparse_categorical_accuracy'*.
- Optimizador: SGD con *momentum* 0.9.
- Filtros de capas convolucionales propias: tamaño 3x3, *stride size* 1, *padding 'same'*.
- Capas *Max-Pooling* propias: *pool size* 2x2, *stride size* 2x2, *padding 'same'*.
- Tasa de *dropout*: 0.5
- *Early Stopping*: *patience* 80, mínima mejora 0.0001, monitoriza pérdida de validación.
- *ReduceLROnPlateau*: *patience* 40, factor 0.7, mínimo *learning rate* 0.0005, mínima mejora 0.001, monitoriza pérdida de validación, *cooldown* 1.
- Épocas máximas: 1000

6.11.1 Prueba 1: 20 clases

Se comienza con una prueba sencilla, muy parecida a la vista en los apartados anteriores. Existirán 20 clases diferentes en la muestra inicial, y un número menor de ellas en cada uno de los tres sets resultantes de aplicar data splitting.

Clases (0, 50]. Resolución imágenes: 128x128. *Batch Size*: 128. *Learning Rate*: 0.01.

Umbral *Oversampling* entrenamiento: 40 imágenes (50% de umbral *Undersampling*).

- Cantidad de clases en la muestra: 20.
- Imágenes en la muestra tras *Undersampling*: 345.
- Cantidad de datos usada tras preprocesado: 825 entrenamiento, 35 validación, 16 testeo.

	Min. Val. Loss	***Max. Val. Acc.***	***Test Acc.***	**Épocas**	**Duración**	**ms/*step***
Modelo Propio	0.4219	0.9429	0.75	410	3.42 min	61
VGG16	2.0270	0.9714	0.9375	237	3.50 min	88
ResNet-50	0.8231	0.9714	0.9375	487	8.69 min	95
Inception V3	1.0249	0.9429	0.75	617	6.30 min	75

Tabla 6-3. Prueba 1: comparación de los modelos

Se obtienen buenos resultados en general. Es lo esperado para un caso con pocas etiquetas. Se prevé que la precisión del modelo para el set de testeo mengüe a la par que se aumenta el número de clases de la muestra. Sería interesante ver la relación resolución de las imágenes-capacidad predictiva del modelo, pues mientras menos píxeles se empleen, menos datos se manipulan y más rápido entrena el modelo, pudiendo aumentar la cantidad de etiquetas posibles de la clasificación o el *Oversampling* sin peligro de reinicio del entorno de trabajo.

6.11.2 Prueba 2: reducción de resolución

Se va a comprobar el efecto de disminuir la resolución de las imágenes. Para ello se repite la Prueba 1 pero con un redimensionamiento de las imágenes a 64x64. Según la documentación de Keras, el modelo Inception V3 no acepta entradas menores a 75x75 por lo que no es posible su evaluación en esta prueba.

	Min. Val. Loss	*Max. Val. Acc.*	*Test Acc.*	Épocas	Duración	ms/*step*
Modelo Propio	0.5564	0.8571	0.8125	462	2.42 min	27
VGG16	0.7481	0.9429	0.8125	1000	5.43 min	23
ResNet-50	0.7535	0.9429	0.7500	909	7.50 min	45
Inception V3	-	-	-	-	-	-

Tabla 6-4. Prueba 2: comparación de los modelos

Como cabría esperar, la precisión de los modelos se ve perjudicada ante esta reducción de información excepto en el modelo propio, el cual mejora, un efecto curioso debido posiblemente a la poca cantidad de datos usados, que falsean el resultado, siendo este poco fiable. En cambio, el tiempo de entrenamiento se reduce a la mitad en cada *step*. Se escoge el camino que más interese. Como resolución del problema, se busca un modelo con buena capacidad predictiva pero sin acaparar demasiados recursos computacionales, o sea, un equilibrio.

Debe tenerse en cuenta que el set de validación tiene datos y puede que clases distintos a los que contiene el set de testeo, es por ello que no comparten el mismo resultado. Ambos valores son de interés, pero en este caso el *batch size* con el que se ha evaluado el máximo valor de precisión de validación, 128, valida cada época con todas las muestras de validación, 35, más muestras que las solo 16 del set de testeo. De la forma en la que se han asignado los porcentajes de división de los datos para *data splitting*, el set de validación siempre será más numeroso que el de testeo, con un 10% de las imágenes de la muestra frente a un 4.5%. Por esta razón, la precisión de validación tendrá más importancia al haberse empleado mayor cantidad y posiblemente variedad de imágenes que en el testeo.

6.11.3 Prueba 3: 75 clases

Se aumentan el número de clases con respecto a la Prueba 1 y se disminuye el Oversampling de 40 imágenes a 16 para no consumir tantos recursos computacionales. La resolución vuelve a ser 128x128.

Clases (0, 200]. Resolución imágenes: 128x128. Batch Size: 128. Learning Rate: 0.01.

Umbral *Oversampling* entrenamiento: 16 imágenes (20% de umbral *Undersampling*).

- Cantidad de clases en la muestra: 75.
- Imágenes en la muestra tras *Undersampling*: 1291.
- Cantidad de datos usada tras preprocesado: 1599 entrenamiento, 130 validación, 59 testeo.

	Min. Val. Loss	*Max. Val. Acc.*	*Test Acc.*	Épocas	Duración	ms/*step*
Modelo Propio	1.7816	0.6462	0.5254	383	5.43 min	65
VGG16	3.6526	0.8462	0.7288	691	14.52 min	92
ResNet-50	4.8466	0.8462	0.8644	330	9.55 min	100
Inception V3	5.6166	0.5923	0.3898	242	4.59 min	74

Tabla 6-5. Prueba 3: comparación de los modelos

Destaca positivamente el rendimiento del modelo con *backbone* de ResNet-50, logrando en pocas épocas una combinación de pesos que produce excelentes predicciones en validación y en el set de testeo (Figura 6-32).

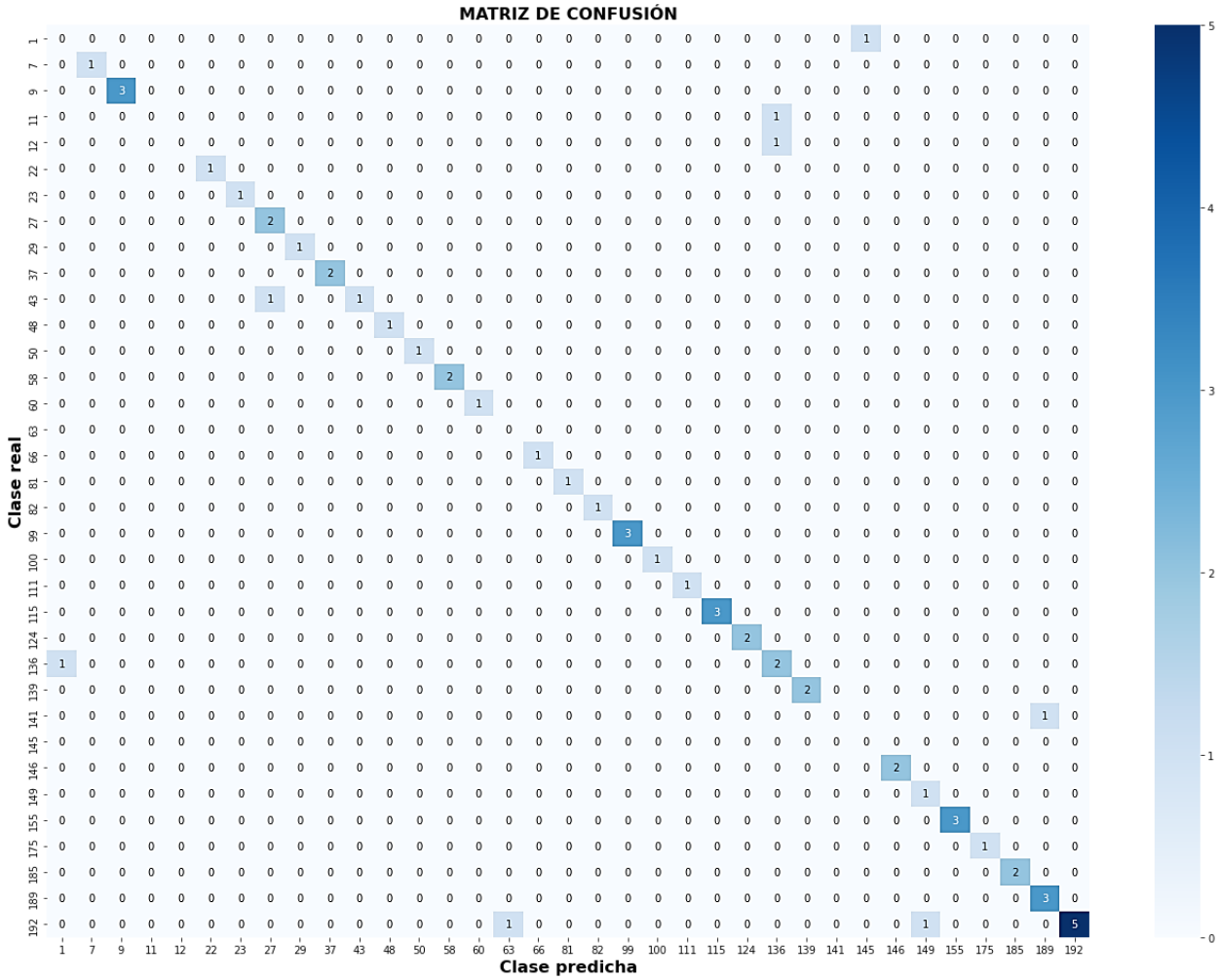


Figura 6-32. Prueba 2: matriz de confusión de ResNet-50 en set de testeo

Para mejorar los resultados de esta prueba, quizás habría que desactivar *Early Stopping* para que los modelos entrenaran durante más tiempo o modificar sus parámetros, como incrementar la paciencia o reducir la mejora mínima esperada. Otra alternativa sería disminuir el *learning rate* para llegar al mínimo local de la función de coste con menos oscilaciones, monotonía constante creciente, aunque se extendería el tiempo de aprendizaje bastante.

6.11.4 Prueba 4: 75 clases y aumento de resolución

En esta prueba se desactivará el *callback* de *Early Stopping* y además se aumentará la resolución de las imágenes de 128x128 a 224x224 píxeles, a sabiendas de que el entrenamiento se extenderá en duración.

Clases (0, 200]. Resolución imágenes: 224x224. Batch Size: 128. Learning Rate: 0.01. No *Early Stopping*.

Umbral *Oversampling* entrenamiento: 16 imágenes (20% de umbral *Undersampling*).

- Cantidad de clases en la muestra: 75.
- Imágenes en la muestra tras *Undersampling*: 1291.
- Cantidad de datos usada tras preprocesado: 1599 entrenamiento, 130 validación, 59 testeo.

	Min. Val. Loss	*Max. Val. Acc.*	*Test Acc.*	Épocas	Duración	ms/*step*
Modelo Propio	1.4656	0.7154	0.6610	1000	37.58 min	182
VGG16	2.2104	0.9000	0.8305	1000	56.65 min	275
ResNet-50	1.2782	0.9154	0.8135	1000	61.61 min	268
Inception V3	1.9961	0.8000	0.7796	1000	41.59 min	190

Tabla 6-6. Prueba 4: comparación de los modelos

Efectivamente, la previsión se ha cumplido. Todos los modelos han mejorado su error mínimo de validación y máxima precisión en la validación. Por lo general, han rendido mejor en la clasificación del set de testeo, en comparación con la Prueba 3.

Los ***trade-off*** son muy comunes a la hora de diseñar Redes Neuronales Artificiales u otras técnicas de clasificación de Aprendizaje Automático. Se mejora el modelo en algún aspecto a costa de empeorarlo en otro. No hay ganancia sin sacrificio. En esta prueba, al eliminar *Early Stopping* se aumenta el tiempo de uso de los recursos, y al elevar la resolución de las imágenes a 224x224, se aumentan los datos con los que trabaja el sistema, incrementando la potencia computacional necesaria. Estas desventajas provocan una ventaja: la precisión del modelo mejora, generalmente.

El optar por aceptar o no el *trade-off* depende del ámbito de aplicación del modelo, del uso que se le pretenda dar. Habrá ocasiones en las que se priorice la eficiencia del entrenamiento del modelo por falta de recursos, y otras en las que lo verdaderamente importante sea la tasa de acierto, la eficacia (como por ejemplo en la detección de tumores malignos en un paciente). También puede llegarse a un punto medio, en el que las características del modelo mantengan cierto equilibrio, justo lo que se está buscando en este trabajo, una tasa de acierto adecuada para una potencia computacional óptima.

Como observación adicional, el modelo propio, por lo general, entrena durante menos tiempo que los obtenidos con *Transfer Learning*, a pesar de que estos solo entrenen sus 3 capas densas finales mientras que el propio entrene todas y cada una de sus capas (5 convolucionales y 2 densas). La causa radica en que para los primeros hay que calcular una miríada de convoluciones adicionales en cada lote consecuencia de su gran cantidad de capas convolucionales y, en mayor medida, en que estos modelos tengan (se les haya añadido) una capa densa final más que entrenar, que eleva el número de parámetros sinápticos exponencialmente, elevando los componentes del vector gradiente a calcular durante *backpropagation*.

6.11.5 Prueba 5: 186 clases y menor *batch size*

Se volverá a activar *Early Stopping* pero alargando el tiempo de espera para su actuación. El número de clases se aumentará para determinar el rendimiento del modelo en una situación más compleja. Como en la prueba anterior el entorno de trabajo de Kaggle ya avisó del uso excesivo de memoria, alertando de un posible reinicio de la ejecución, se ha decidido disminuir la resolución de las imágenes, devolviéndolas a 128x128 a pesar de que los resultados probablemente sean peores. También se reduce el tamaño de lote a 64 para disminuir la carga computacional en el entrenamiento aunque el entrenamiento se prolongue más.

Clases (0, 500]. Resolución imágenes: 128x128. Batch Size: 64. Learning Rate: 0.01. *Early Stopping*: paciencia 150 épocas, mínima mejora 0.0001.

Umbral *Oversampling* entrenamiento: 16 imágenes (20% de umbral *Undersampling*).

- Cantidad de clases en la muestra: 186.
- Imágenes en la muestra tras *Undersampling*: 3218.
- Cantidad de datos usada tras preprocesado: 4040 entrenamiento, 322 validación, 145 testeo.

	Min. Val. Loss	*Max. Val. Acc.*	*Test Acc.*	Épocas	Duración	ms/*step*
Modelo Propio	2.6829	0.6925	0.2344	869	33.48 min	35
VGG16	9.1347	0.6273	0.5793	369	21.59 min	55
ResNet-50	4.9750	0.7298	0.6207	510	33.73 min	60
Inception V3	3.8904	0.3261	0.4000	954	48.65 min	49

Tabla 6-7. Prueba 5: comparación de los modelos

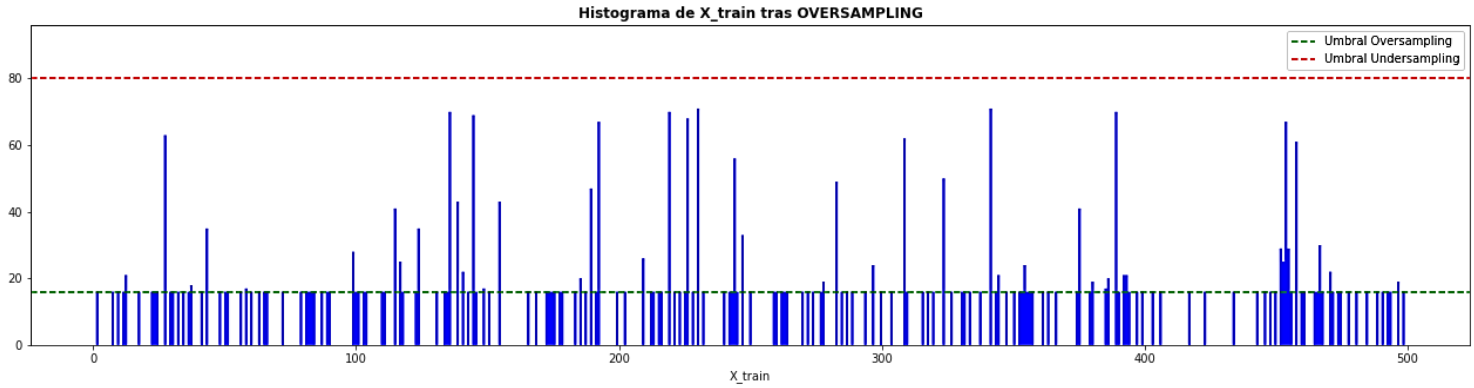


Figura 6-33. Prueba 5: datos de entrenamiento tras preprocesado

El modelo con estructura convolucional Inception V3 da muy malos resultados (Figura 6-34) en precisión en comparación con los otros dos modelos de *Transfer Learning*, a pesar de ser de entrenado durante más tiempo. A continuación, se señalan estos errores y procedimientos viables para su corrección, extrapolables a cualquier otro modelo en las mismas circunstancias:

- Presenta demasiadas fluctuaciones en la gráfica de la métrica y ni su valor durante el entrenamiento ni validación supera el 40%. Todos estos defectos pueden tener su origen en un *learning rate* demasiado alto o un *batch size* demasiado pequeño. Es decir, se sabe que el modelo actualiza el gradiente y los pesos cada *step* por usar *Mini-Batch Gradient Descent* con cierto tamaño de lote. Si la tasa de aprendizaje es demasiado alta, costará más iteraciones converger en el mínimo local al variar demasiado los pesos de un *step* a otro. En cuanto al *batch size*, si es demasiado pequeño, un lote empleado en un *step* puede diferir mucho en dificultad para aprender o clasificar con respecto al siguiente lote. El cambio que arreglaría esto es el incremento del hiperparámetro *batch size*, de forma que el nivel de complejidad se ecualice un poco para todos los lotes.
- Por otro lado se observa un curioso efecto: los valores de pérdida de la validación son inferiores (más favorables) a los de entrenamiento durante todas las épocas. Este efecto puede tener dos orígenes. Los culpables pueden ser los mecanismos de regularización empleados como norma L2 o *dropout*, los cuales solo están habilitados durante el entrenamiento y se apagan para la validación. La otra posible causa puede ser que la pérdida en entrenamiento mostrada en una época es la media de las pérdidas de cada *step* que la componen, siendo por lo general menor en el último *step* de la época que en el primero por la actualización de pesos. Sin embargo, la pérdida de validación únicamente se computa una vez, al final de cada época, momento en el que los pesos ya se han actualizado, por ello es posible que la media de error en el entrenamiento sea superior al error de validación.

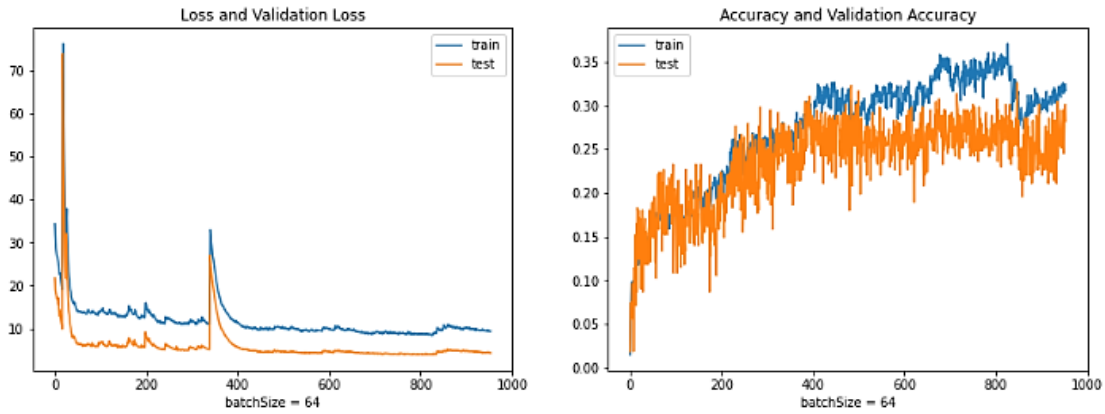


Figura 6-34. Prueba 5: modelo con Inception V3 y *batch size* 64

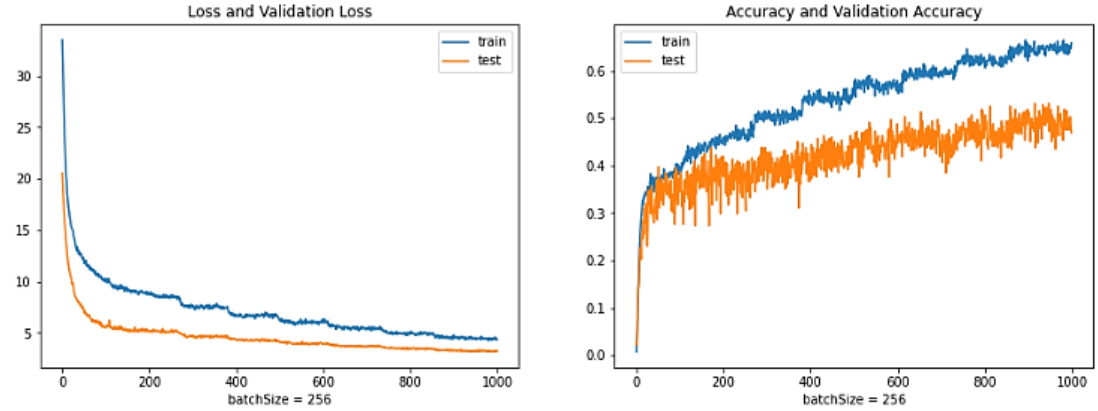


Figura 6-35. Prueba 5: modelo con Inception V3 y *batch size* 256

Se observa una clara mejoría (Figura 6-35) con respecto a la ejecución inicial (Figura 6-34) al usar *batch size* de 256 en vez de 64, en el mismo tiempo de entrenamiento, 1000 épocas, donde no ha saltado *Early Stopping* y se presenta una monotonía creciente, claro indicador de que el modelo puede mejorar si se sigue entrenando durante más tiempo, algo que no se va a realizar por cuestiones de gestión de recursos computacionales.

El código implementado para el *Oversampling* de los datos de entrenamiento no es muy eficiente. En esta prueba, su ejecución ha tardado aproximadamente 5 minutos en aumentar la cantidad de muestras de clases minoritarias del set de entrenamiento hasta 16, resultando en un aumento de 2751 a 4040 imágenes en total. Se analizará el rendimiento del modelo si se desactiva el *Oversampling* en una prueba posterior.

6.11.6 Prueba 6: 186 clases y mayor *batch size*

Se va a realizar una prueba igual que la Prueba 5, pero manteniendo *batch size* a 256 porque se vio que era beneficioso, al menos para el modelo Inception V3.

	Min. Val. Loss	*Max. Val. Acc.*	*Test Acc.*	**Épocas**	**Duración**	**ms/*step***
Modelo Propio	1.7885	0.6553	0.5724	587	19.48 min	123
VGG16	2.1028	0.7826	0.7103	1000	46.48 min	180
ResNet-50	1.7671	0.8168	0.7517	1000	50.23 min	187
Inception V3	3.1434	0.5311	0.5241	1000	34.65 min	130

Tabla 6-8. Prueba 6: comparación de los modelos

Absolutamente todos los modelos mejoran respecto al anterior al disminuir su pérdida de validación y aumentar su acierto en la clasificación del set de testeo. Por lo general, se entrena hasta el límite establecido de 1000 épocas sin activarse *Early Stopping*, lo que se traduce en un monotonía decreciente en la variable monitorizada de pérdida de validación, que normalmente implica crecimiento constante de la precisión en validación. Estos datos son claro indicador de que el modelo puede seguir mejorando con esa misma configuración de hiperparámetros.

El resultado obtenido con ResNet-50 es el mejor obtenido hasta ahora siguiendo la relación eficiencia computacional-precisión respecto al número de clases. Obtiene en tan solo 50 minutos un 75% de acierto en un set de testeo que incorpora 145 imágenes pertenecientes a 42 clases distintas tras haber aprendido patrones de 186 clases con 81.7% de precisión y poca pérdida.

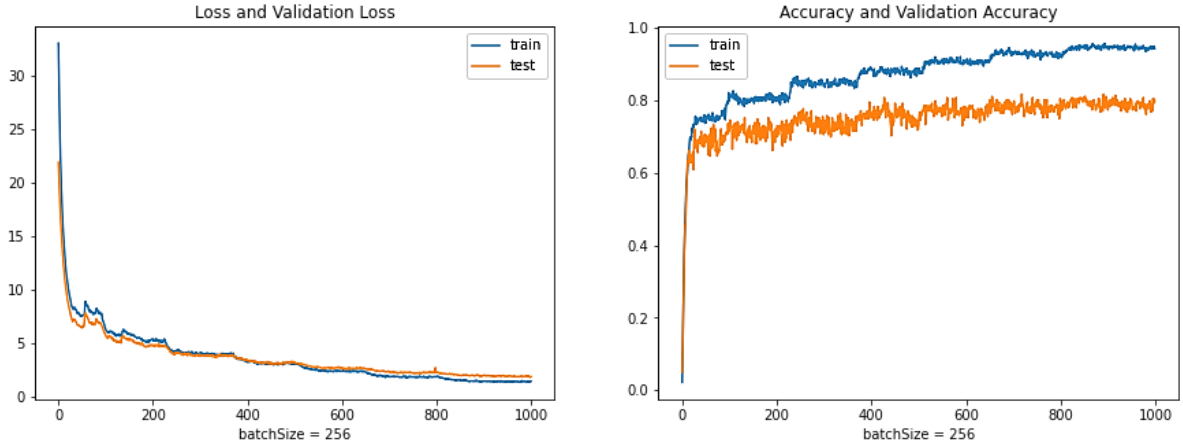


Figura 6-36. Prueba 6: mejor resultado obtenido con ResNet-50 en 1000 épocas

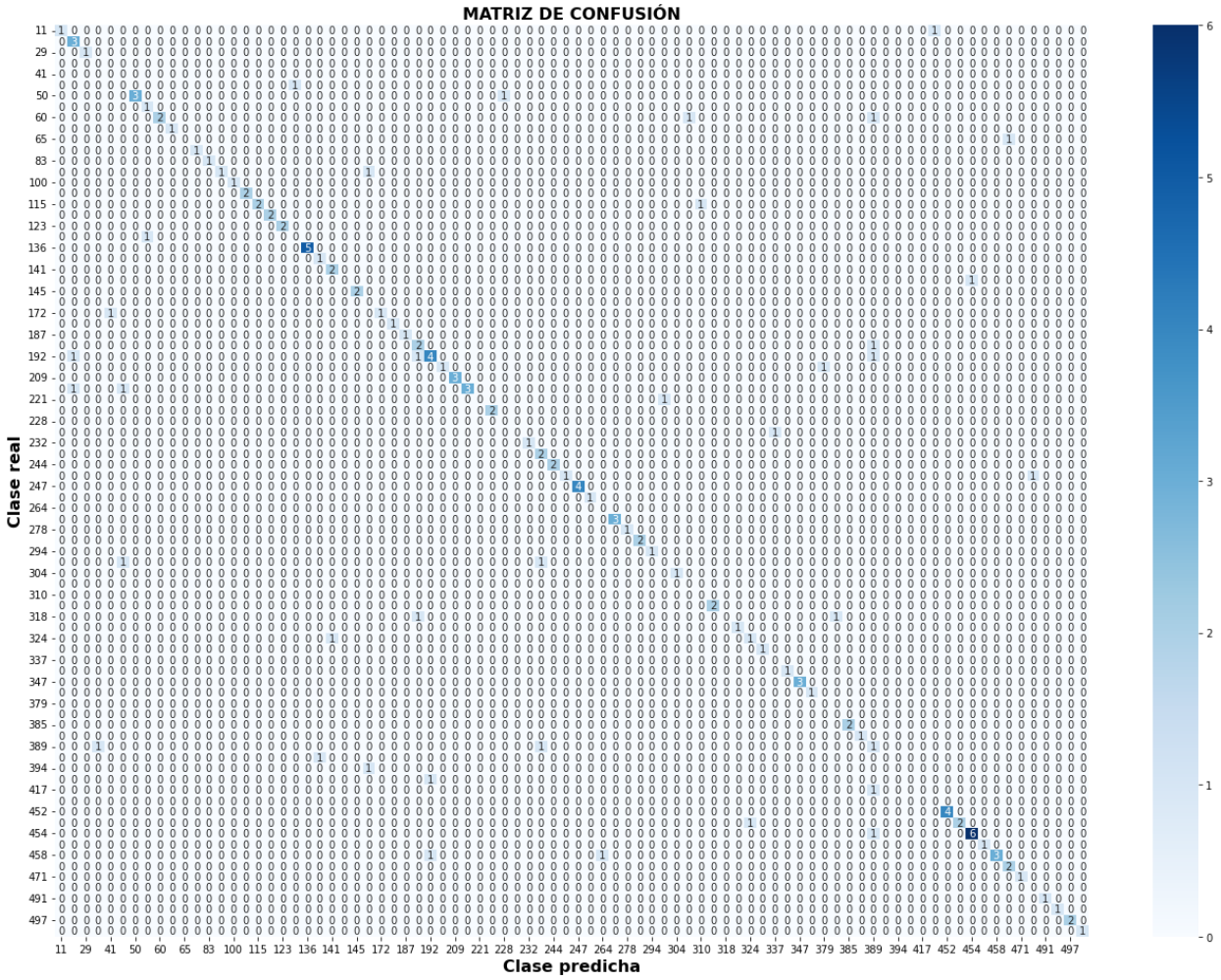


Figura 6-37. Prueba 6: matriz de confusión de ResNet-50 en 1000 épocas en set de testeo

Se va a apostar por el anterior modelo ResNet-50, realizando un nuevo entrenamiento pero con límite 2000 épocas para observar los resultados que puede llegar a conseguir.

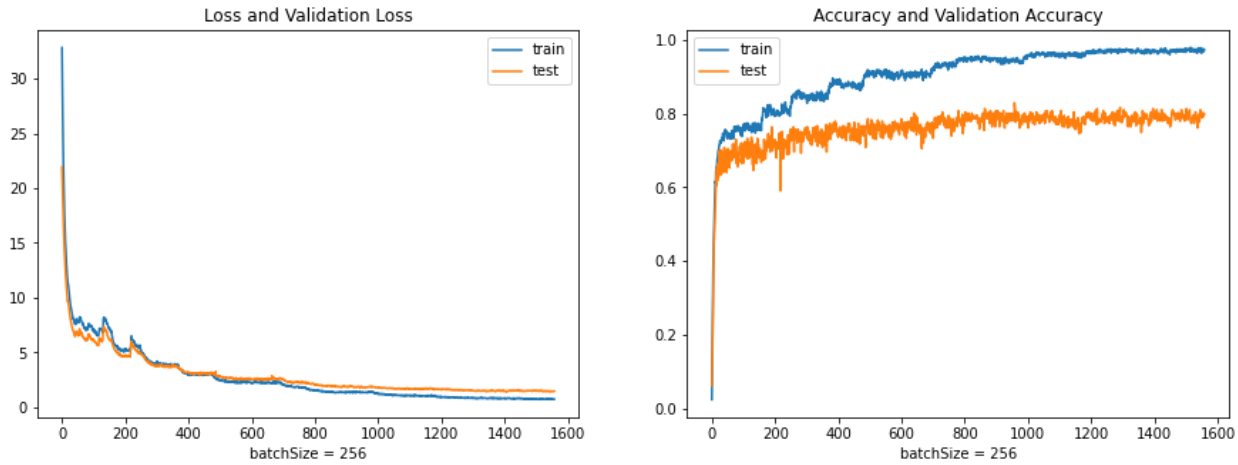


Figura 6-38. Prueba 6: mejor resultado obtenido con ResNet-50 en 2000 épocas

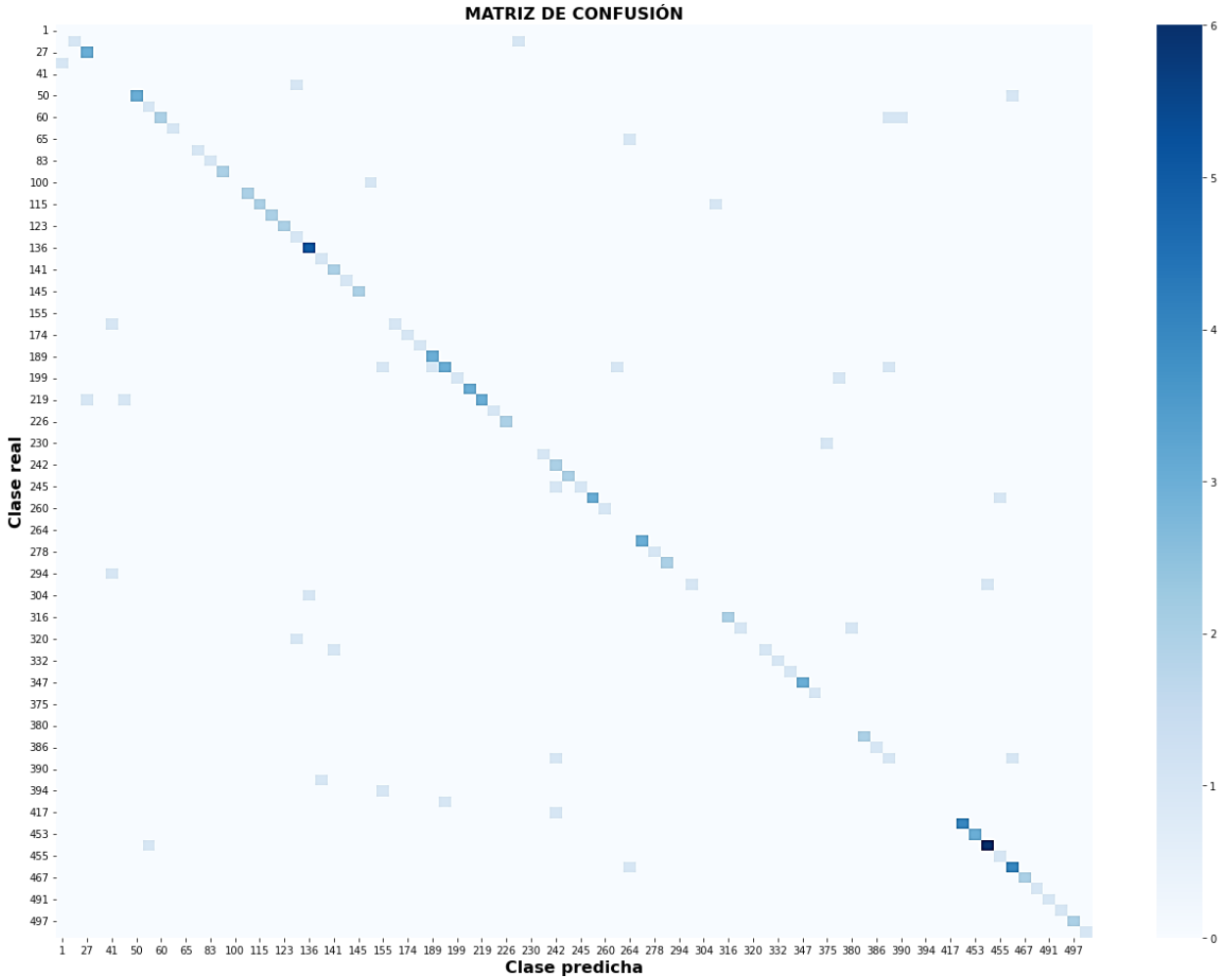


Figura 6-39. Prueba 6: matriz de confusión de ResNet-50 en 2000 épocas en set de testeo

El aprendizaje acaba al activarse *Early Stopping* en la época 1558. Ha mejorado escasamente los resultados previos: error mínimo de validación de 1.4121 y precisión máxima de validación de 82.92% en 77.63 minutos, y 76.55% de tasa de acierto sobre el set de testeo. A costa de aumentar el entrenamiento 27 minutos, el modelo únicamente ha mejorado un 1% en ambas precisiones. Esto no se considera rentable, por lo que se decide que es mejor el primer resultado en esta prueba.

6.11.7 Prueba 7: 186 clases, mayor *batch size* y sin Oversampling

Esta última prueba se realizará con los mismo hiperparámetros que la anterior, pero desactivando el *Oversampling* en el set de entrenamiento para observar su impacto en las predicciones. Gracias a su inhabilitación se ahorran unos 5 minutos de preprocesado de imágenes.

Clases (0, 500]. Resolución imágenes: 128x128. *Batch Size*: 256. Learning Rate: 0.01. *Early Stopping*: paciencia 150 épocas, mínima mejora 0.0001.

Oversampling desactivado.

- Cantidad de clases en la muestra: 186.
- Imágenes en la muestra tras *Undersampling*: 3218.
- Cantidad de datos usada tras preprocesado: 2751 entrenamiento, 322 validación, 145 testeo.

	Min. Val. Loss	***Max. Val. Acc.***	***Test Acc.***	**Épocas**	**Duración**	**ms/*step***
Modelo Propio	1.8469	0.6646	0.6068	1000	23.48 min	133
VGG16	11.0194	0.7236	0.6000	342	12.57 min	190
ResNet-50	6.8068	0.7764	0.7172	532	20.10 min	200
Inception V3	4.4027	0.5031	0.4758	1000	25.02 min	140

Tabla 6-9. Prueba 7: comparación de los modelos

Al desactivar el *Oversampling* de 16 imágenes en las clases menos representadas, el efecto es el esperado. La precisión de los modelos decae, en general, aunque seguramente no tanto como debería gracias al uso de *Class Weighting* (6.7.6) para ayudar a arreglar el *Imbalanced Data*. El *trade-off* en este caso sería escoger entre ejecutar el código en 5 minutos menos o conseguir un 3-4% de precisión adicional. Se opta por el segundo pues 5 minutos no se considera un tiempo descabellado ni un derroche de recursos por ese porcentaje extra.

El modelo propio ha conseguido entrenar 1000 épocas sin activar *Early Stopping,* es por eso que presenta un mejor resultado que en la prueba anterior. Los milisegundos por *step* son aproximadamente los mismos que en la Prueba 6 porque en cada *step* se entrena con *batch size* datos, que son los iguales en ambas pruebas. Sin embargo, el número de *steps* por época sí que es distinto, pues en el caso con *Oversampling* se dan más al aumentar el numerador de la ecuación (4.7).

6.11.8 Prueba 8: ResNet-50 con más clases

Dado que el modelo más prometedor y que mejores resultados ha proporcionado es el que incorpora el *backbone* de ResNet-50, se ha decidido ir más allá con este, probando varias ejecuciones con diferentes etiquetas posibles de la clasificación. La resolución de las imágenes se mantendrá en 128x128 y el *batch size* será 256. El *Oversampling* de las muestras de entrenamiento será de 16 imágenes. *Learning rate* de 0.01 con *ReduceLROnPlateau* y *Early Stopping* con paciencia de 150 épocas y mínima mejora 0.0001.

Al aumentar el número de clases también se elevan las muestras, llegando en el último caso, con 594 clases, al límite de memoria del entorno de trabajo, que son 12724 imágenes de entrenamiento tras *Oversampling* y de 128x128 píxeles. El algoritmo diseñado para *Oversampling*, en este último caso, tarda 54 minutos en duplicar las imágenes hasta 16 en las clases minoritarias.

ResNet-50	*Min. Val. Loss*	*Max. Val. Acc.*	*Test Acc.*	Épocas	Duración	ms/*step*
186 clases	1.7671	0.8168	0.7517	1000	50.23 min	185
268 clases	1.9094	0.7730	0.7286	1471	104.73 min	185
389 clases	2.0655	0.7581	0.7069	1440	144.78 min	185
594 clases	2.8597	0.6456	0.6169	1636	252.00 min	185

Tabla 6-10. Prueba 8: comparación de ejecuciones del modelo ResNet-50

Se observa una clara tendencia: a medida que se aumentan las clases, también lo hace la pérdida en validación y la duración del entrenamiento, pero disminuye la máxima precisión en validación y en el set de testeo.

Para mejorar el resultado se debería adaptar el modelo a cada caso: modificar el número de capas, el *learning rate*, *batch size*, otros hiperparámetros, etc. Debería personalizarse en función del número de clases y no mantener exactamente la misma combinación de hiperparámetros. Sin embargo, esto supone un estudio demasiado laborioso y particularizado, que no se mostrará en esta memoria.

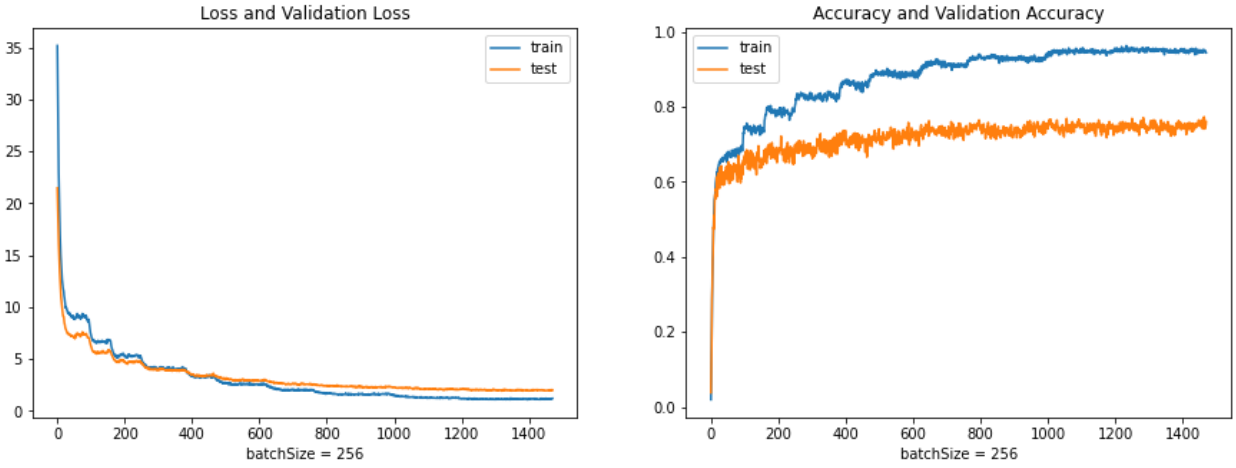


Figura 6-40. Prueba 8: resultado de ResNet-50 con 286 clases

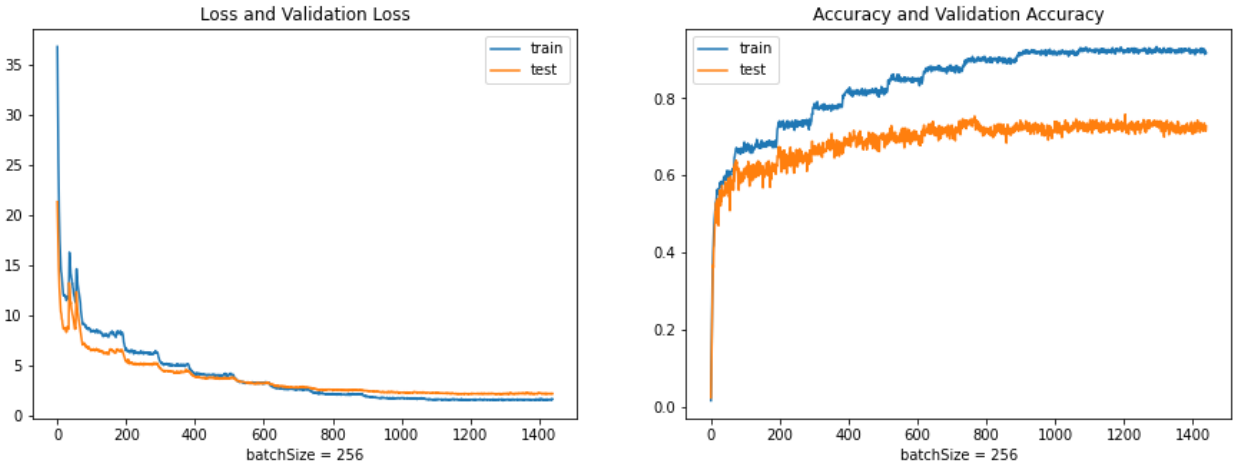


Figura 6-41. Prueba 8: resultado de ResNet-50 con 389 clases

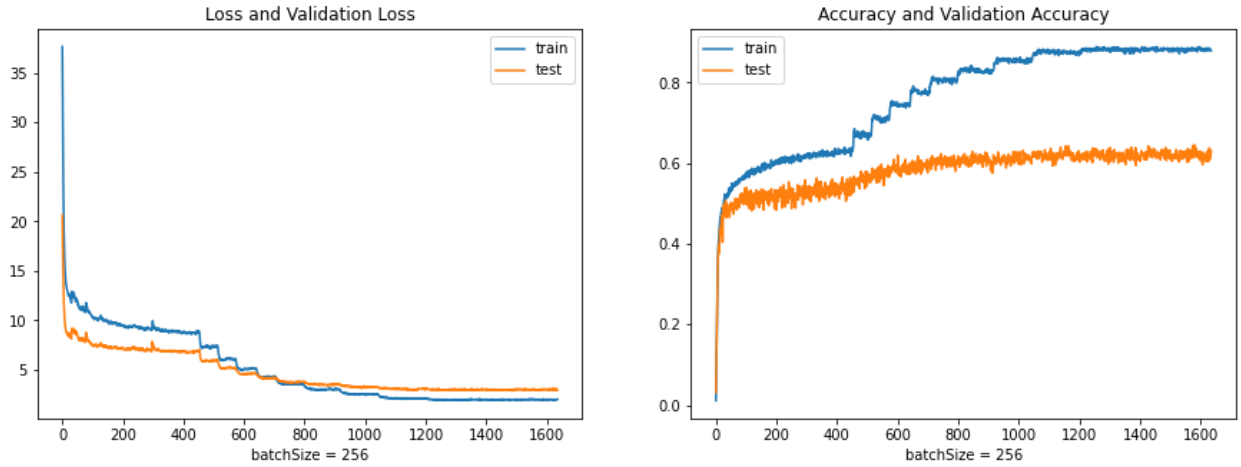


Figura 6-42. Prueba 8: resultado de ResNet-50 con 594 clases

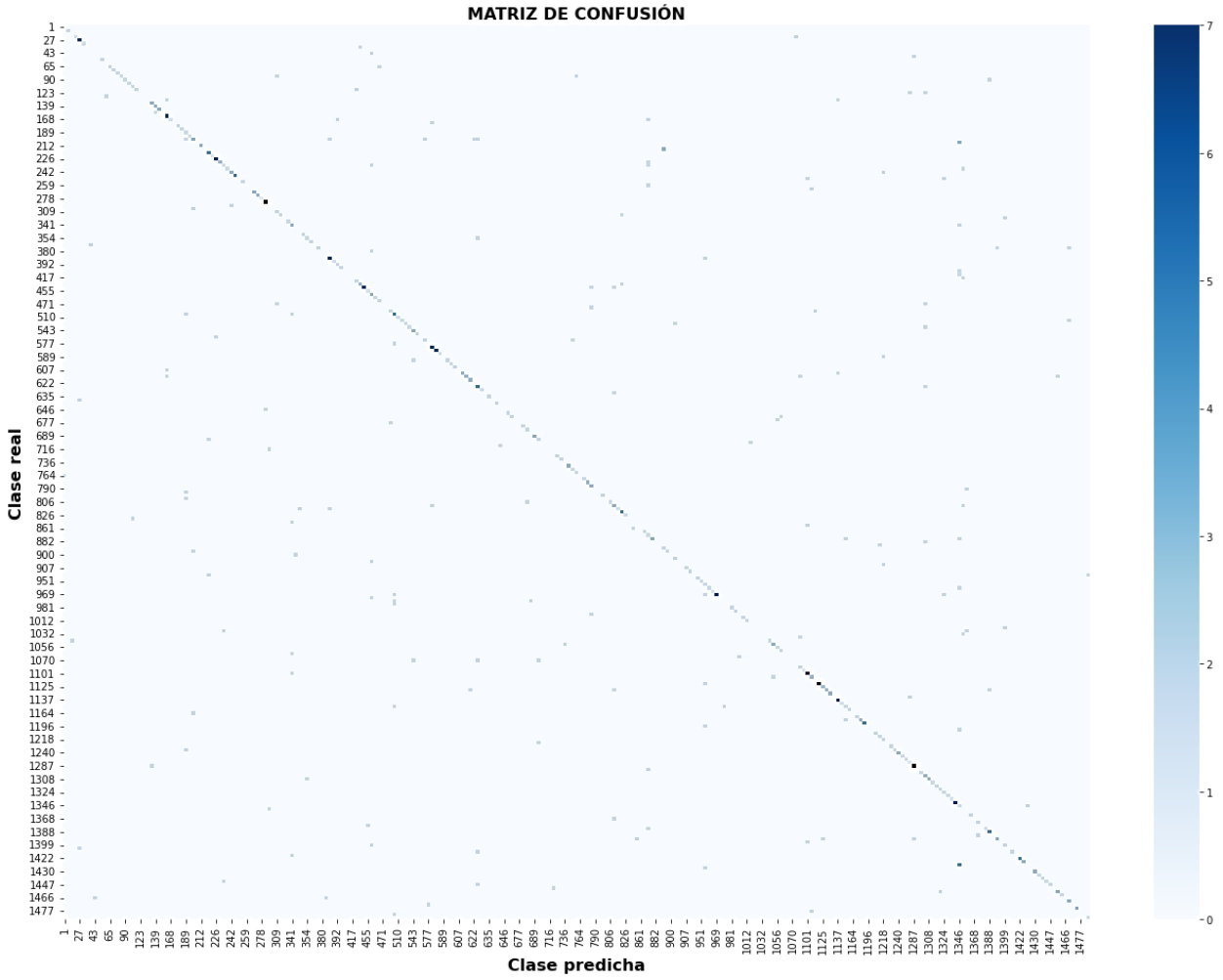


Figura 6-43. Prueba 6: matriz de confusión de ResNet-50 con 594 clases totales

Como experimentación, también se ha realizado una ejecución con 268 clases pero sustituyendo la resolución 128x128 de cada canal de las imágenes por 64x64. Los resultados son bastante lógicos: la precisión máxima en validación disminuye hasta 60% y en testeo se consigue un 55.23% de acierto, tras un entrenamiento de 40 minutos. Si se aumentaran también las clases disponibles, la precisión se desplomaría, bajando del 50%, un valor que no se considera lo suficientemente bueno y que no merece la pena.

De entre todos las estructuras implementadas mediante *Transfer Learning*, se puede concluir con que la referente a **ResNet-50** es la que proporciona unos resultados superiores, superando a sus adversarios en la mayoría de las pruebas realizadas, al menos para las combinaciones de hiperparámetros y situación en la que se ha trabajado.

7 CONCLUSIONES Y LÍNEAS FUTURAS

Tras el diseño, implementación y comparación de varios modelos a lo largo de los últimos apartados, se pueden realizar varias observaciones, sacar conclusiones y proponer algunas mejoras:

- A mayor número de clases, mayor capacidad discriminatoria requiere el modelo. Es decir, se necesitan más funciones de activación no lineales, por lo tanto más capas, para disponer de un mayor nivel de no linealidad. El modelo propio no es tan profundo como los obtenidos con *Transfer Learning*. Posee únicamente 7 capas entre convolucionales y densas. Esta falta de capas provoca una menor precisión, un modelo que generaliza peor al elevar la cantidad de etiquetas posibles, incapaz de establecer una separación con hiperplano adecuada entre las muestras pertenecientes a distintas clases representadas en el espacio de características a partir de sus vectores de características. Para solucionar este contratiempo, se podría optar por aumentar las capas con funciones de activación. Pero esto tiene un precio: el coste computacional necesario para el entrenamiento aumenta. A diferencia del *Transfer Learning* implementado en el que los pesos de las capas convolucionales ya estaban asignados (aunque a otro *dataset*), en el modelo propio sí que es obligatorio entrenar todas sus capas, lo que se traduce en mayor gasto de recursos. Este es un claro *trade-off* típico de las redes neuronales. Hay que preguntarse si ese sacrificio de recursos adicional merece la pena a cambio mejorar los resultados.

- El exceso de capas también es negativo. Al contrario que con el modelo propio, que posee pocas capas, el modelo con *backbone* de Inception V3 es el más profundo implementado, con originalmente 189 capas, entre convolucionales y densas. Su rendimiento al elevar las clases posibles va mermándose, llegando a estar incluso por debajo del rendimiento del modelo propio. Esta degradación de la eficacia del modelo no tiene por qué deberse únicamente al problema del desvanecimiento del gradiente pues precisamente esta red incorpora clasificadores auxiliares para reducir su efecto. Una posible fuente del problema es que el uso de tantas capas extrae características más complejas de lo necesario para clasificar las imágenes del *dataset* de lugares emblemáticos y cuando hay pocos datos de entrenamiento en algunas clases, como en el caso actual, el modelo tiende a un *overfitting*, memorizando dichas instancias de entrenamiento. Otro posible origen del mal rendimiento puede ser que los pesos ajustados con el *dataset* ImageNet, que se han mantenido con *Transfer Learning*, no sean adecuados para predecir el *dataset* utilizado.

- Haciendo referencia a esto último, se puede afirmar que los modelos con *Transfer Learning* no liberan todo su potencial para el *dataset* de lugares emblemáticos de la manera en la que se han implementado, puesto que no se han modificado los pesos de las capas convolucionales. Estos están adaptados al *dataset* ImageNet, pero aun así consiguen grandes resultados para este problema, lo que denota la gran arquitectura que poseen y el estudio que tienen detrás. Si se quisiera mejorar la precisión (muy recomendable para el caso de Inception V3), podría recurrirse al entrenamiento de algunas capas convolucionales de las *backbone* con un *learning rate* pequeño, para no variar los pesos y sesgos en exceso, sobre el *dataset* de lugares emblemáticos, logrando una combinación de parámetros más conveniente, más personalizada al problema.

- Los resultados obtenidos en cada prueba para cada modelo no son completamente reproducibles. Si bien se ha usado una semilla para que la división de la muestra en los tres sets o los pesos afectados por *Dropout* sean iguales de una ejecución a otra, no se ha establecido una inicialización de los pesos, por lo que en cada ejecución todos los pesos de la red del modelo propio y de las últimas capas densas de los modelos con *Transfer Learning* empezarán en un valor distinto. Al empezar en un punto distinto, ni los caminos seguidos hasta llegar a un mínimo local ni el propio mínimo local serán iguales. Además, el set de entrenamiento se baraja antes de entrenar aleatoriamente. Como consecuencia, en cada ejecución puede convergerse en un mínimo local diferente. Como apunte adicional, el inicializar todos

los pesos al mismo valor es contraproducente pues todas las neuronas de una misma capa recibirían la misma combinación lineal de entradas y actuarían exactamente igual en todas las actualizaciones de pesos. Además, si se inicializan en cero, solo se propagarían ceros por la red si se usa ReLU o tangente hiperbólica.

- En situaciones en las que se disponen de tan pocas muestras en algunas clases, es vital la combinación de *Oversampling* en el entrenamiento con un posterior *Data Augmentation*. Estos dos algoritmos aúnan esfuerzos para prevenir un *Overfitting* que tarde o temprano tendría lugar, mejorando la capacidad predictiva del modelo de manera automática. Con *Oversampling* se aumentan las muestras minoritarias totales para el entrenamiento. Con *Data Augmentation*, en cada época los datos son modificados aleatoriamente dentro de un límite preestablecido, añadiendo diversidad a las muestras. El algoritmo para el *Oversampling* diseñado en el código no es muy eficiente, consume demasiado tiempo en tan solo duplicar imágenes. De la misma forma, el aumento de datos en *layers* consume recursos adicionales, aunque muchos menos que su implementación con la clase ImageDataGenerator. Podría barajarse el uso del algoritmo SMOTE en lugar de la tupla *Oversampling* por duplicado y *Data Augmentation*, u optimizar de alguna forma que escapa los actuales conocimientos el código para *Oversampling* (Código 6-14).
- Si se quisiera aumentar el alcance del modelo, esto es, clasificar muchas más imágenes a mayor resolución, se recurrirían a las TPUs (Unidad de Procesamiento Tensorial), proporcionadas por algunas plataformas como Kaggle, que aceleran los cálculos hasta 30 veces más que las GPUs. Como se comentó, las GPUs están pensadas para operaciones en paralelo y rinden bien en álgebra lineal, pero son muy ineficientes para RNA. En cambio, las TPUs han sido diseñadas específicamente para tener un desempeño muy eficiente y trabajar más rápido en RNA, sobre todo con RNA profundas implementadas con la biblioteca *Tensorflow*, pues son desarrolladas por Google.
- Los modelos implementados podrían haber sido usados como clasificador de cualquier otra temática de imágenes diferente a lugares emblemáticos, incluso fuera del ámbito visual, como reconocimiento de sonidos o propiedades atribuidas a ellos. Los únicos cambios necesarios serían utilizar un *dataset* pertinente, ajustar adecuadamente los datos, reajustar algunos hiperparámetros, añadir/quitar capas y entrenar las capas que se crean convenientes. Esto demuestra la potencia, versatilidad y adaptabilidad de las CNN y su infinidad de aplicaciones.
- Si el resultado de un modelo se está evaluando en función de su porcentaje de precisión, debe tenerse en cuenta que no es lo mismo disponer, por ejemplo, de un 50% de precisión para dos clases que para diez. En el primer caso, equivaldría a tirar una moneda, es decir, un proceso estocástico, totalmente aleatorio, por lo que el modelo de CNN conseguido sería inservible. A mayor número de clases, menor es la separación, diferencia, entre los porcentajes de probabilidad dados por la función de activación de la última capa, *softmax*; ergo, mayor es la probabilidad de error, de confundir la clase. Por lo tanto, puede afirmarse que es más meritorio que un modelo obtenga cierto porcentaje de acierto con x clases a que lo obtenga con $y < x$ clases. Siguiendo esta declaración, el último resultado obtenido, en la Prueba 8 (6.11.8), es el mejor en términos de precisión-cantidad de etiquetas, aunque no en eficiencia computacional.

Como evolución del actual proyecto, los conceptos aprendidos y bloques de código implementados podrían llevarse directamente del banco de trabajo a la práctica. Hoy en día, gracias a la globalización tecnológica, muchas herramientas y medios están a nuestra disposición, siendo enormemente facilitada la materialización de ideas, algo que no ocurría en los albores de la Inteligencia Artificial. Esto se ha visto reflejado en el desarrollo de este trabajo, en el que alguien con nula experiencia en el tema del Aprendizaje Automático ha conseguido crear y hacer funcionar un modelo de Redes Neuronales Convolucionales básico potencialmente útil. De igual manera, una aplicación móvil o página web que haga uso de dicho modelo también sería factible sin demasiados conocimientos previos ni recursos. Limitando la utilidad de dicha aplicación a un área geográfica, según el alcance y tiempo que se dedique al proyecto, se elaboraría un *dataset* de imágenes de, por ejemplo, lugares emblemáticos, etiquetados manualmente, que se usaría para entrenar un modelo de CNN como los

implementados previamente. Este modelo con los pesos ya ajustados sería la base de la aplicación, que se optimizaría al máximo para el menor consumo de recursos y a la que se le añadiría una interfaz de usuario intuitiva. El usuario al descargar la aplicación, tan solo tendría que realizar fotografías con la cámara del *smartphone* a zonas de dicha área geográfica, que sirven de entradas a la *input layer* del modelo. Tras una clasificación instantánea de la imagen, se conocería el nombre (la clase) del lugar emblemático casi con certeza, al que se le podría añadir información adicional (fecha de construcción, historia, estilo arquitectónico, curiosidades, etc.) para aumentar la utilidad, enfocando el uso de la aplicación al turismo. Como funcionalidad extra, los usuarios podrían enviar sus propias imágenes etiquetadas de un nuevo lugar emblemático al servidor en el que se alberga el modelo, donde se realizaría cada cierto tiempo un *Transfer Learning*: se congelarían las capas convolucionales mientras que *output layer* modificaría el número de neuronas para adaptarse al nuevo número de clases y se volvería a entrenar únicamente la cabeza del modelo...

Siguiendo esta línea, el provecho que se puede sacar de las tecnologías de CNN, otras RNA y en definitiva, del Aprendizaje Automático, es inabarcable. Ya en la actualidad, la gran mayoría de los sectores las incorporan, forman parte de la rutina habitual aunque desapercibidas, pero a medida que pasa el tiempo, cada vez son más las personas de a pie que van conociéndolas, comprendiendo su utilidad, interesándose por el tema... aumentando así la cantidad de aplicaciones derivadas, contribuyendo aún más a su desarrollo, y generando expectación, en un ciclo sin fin. Se puede afirmar que estas tecnologías son el futuro, y a la vez el presente.

REFERENCIAS

[1] R. Benítez, G. Escudero, S. Kanaan y D. Masip Rodó, «Neuronas y transistores,» de *Inteligencia artificial avanzada*, Barcelona, Editorial UOC, 2013, pp. 11-14.

[2] S. Sánchez-Migallón, «¿Puede la conciencia humana ser calculada y, por tanto, podremos programarla en una máquina?,» Xataka, 9 Febrero 2016. [En línea]. Available: https://www.xataka.com/robotica-e-ia/puede-la-conciencia-humana-ser-calculada-y-por-tanto-podremos-programarla-en-una-maquina.

[3] Á. Díaz, «Premio Princesa de Asturias de Investigación Científica y Técnica para los 'padrinos' de la inteligencia artificial,» *El Mundo,* 15 Junio 2022.

[4] M. Labbe y I. Wigmore, «narrow AI (weak AI),» TechTarget, 2021. [En línea]. Available: https://www.techtarget.com/searchenterpriseai/definition/narrow-AI-weak-AI.

[5] R. Benítez, G. Escudero, S. Kanaan y D. Masip Rodó, «Ámbitos de aplicación de la inteligencia artificial,» de *Inteligencia artificial avanzada*, Barcelona, Editorial UOC, 2013, pp. 18-21.

[6] M. Roser y H. Ritchie, «Moore's Law,» Our World in Data, 1 Noviembre 2020. [En línea]. Available: https://ourworldindata.org/uploads/2020/11/Transistor-Count-over-time.png.

[7] V. C. Müller y N. Bostrom, «Future Progress in Artificial Intelligence: A Survey of Expert Opinion,» Oxford, 2014.

[8] D. Faggella, «When Will We Reach the Singularity? – A Timeline Consensus from AI Researchers,» Emerj, 18 Marzo 2019. [En línea]. Available: https://emerj.com/ai-future-outlook/when-will-we-reach-the-singularity-a-timeline-consensus-from-ai-researchers/.

[9] Open Philanthropy, «Potential Risks from Advanced Artificial Intelligence,» Open Philanthropy, 11 Agosto 2015. [En línea]. Available: https://www.openphilanthropy.org/research/potential-risks-from-advanced-artificial-intelligence/#footnote9_r1nplu1.

[10] Spring Professional, «Habilidades humanas que no pueden suplir las máquinas,» Spring Professional, 21 Septiembre 2020. [En línea]. Available: https://blogcandidatos.springspain.com/transformacion-digital/habilidades-humanas-que-no-pueden-suplir-las-maquinas/.

[11] JuPantaRhei, «ANI, AGI and ASI - what do they mean?,» JuPantaRhei GmbH, [En línea]. Available: https://jupantarhei.com/ani-agi-and-asi-what-do-they-mean/.

[12] P. Mate, «Branches Of Artificial Intelligence,» Medium, 25 Mayo 2020. [En línea]. Available: https://medium.com/myroadtoartificialintelligence/branches-of-artificial-intelligence-812b8e292cdb.

[13] E. Bello, «Lógica Difusa o Fuzzy Logic: Qué es y cómo funciona + Ejemplos,» IEBS Business School, 15 Diciembre 2021. [En línea]. Available: https://www.iebschool.com/blog/fuzzy-logic-que-es-big-

data/.

[14] E. Rifqi Saputra, «Fuzzy Logic: Approximating Imprecise Statement,» Medium, 3 Enero 2020. [En línea]. Available: https://medium.com/it-paragon/fuzzy-logic-approximating-imprecise-statement-54d444237820.

[15] A. Bosch Rué, J. Casas Roma y T. Lozano Bagén, «¿Qué es el deep learning?,» de *Deep Learning: Principios y Fundamentos*, Barcelona, Editorial UOC, 2019, pp. 17-20.

[16] Z.-H. Zhou, «Introduction,» de *Machine Learning*, Singapur, Springer, 2021, pp. 2-24.

[17] J. Brownlee, «How to Define Your Machine Learning Problem,» Machine Learning Mastery, 23 Diciembre 2013. [En línea]. Available: https://machinelearningmastery.com/how-to-define-your-machine-learning-problem/.

[18] UNIR, «Las 3 V del Big Data y el procesamiento de datos,» UNIR - Universidad Internacional de La Rioja, 26 Junio 2020. [En línea]. Available: https://www.unir.net/ingenieria/revista/3-v-big-data/.

[19] Oracle, «¿Qué es una base de datos relacional?,» [En línea]. Available: https://www.oracle.com/es/database/what-is-a-relational-database/.

[20] J. Brownlee, «Why Data Preparation is So Important,» de *Data Preparation for Machine Learning*, 2020, pp. 9-10.

[21] D. Gohil, «IMDb Dataset Top-Rated films 1898-2022,» Kaggle, 2022. [En línea]. Available: https://www.kaggle.com/datasets/digvijaysinhgohil/imdb-dataset-toprated-films-18982022.

[22] Equipo Next, «Datos estructurados y no estructurados: ¿en qué se diferencian?,» Grupo Next, 7 Abril 2021. [En línea]. Available: https://gruponext.es/blog/datos-estructurados-y-no-estructurados/.

[23] Wikipedia, «Semi-structured data,» Wikipedia, 10 Marzo 2022. [En línea]. Available: https://en.wikipedia.org/wiki/Semi-structured_data.

[24] A. Zubchenko, «Data Collection for Machine Learning: The Complete Guide,» Waverley Software Inc, 28 Septiembre 2021. [En línea]. Available: https://waverleysoftware.com/blog/data-collection-for-machine-learning-guide/.

[25] J. Brownlee, «Data Transforms,» de *Data Preparation for Machine Learning*, 2020, pp. 20-22.

[26] The FullStory Education Team, «Categorical vs. quantitative data: The difference plus why they're so valuable,» FullStory, Inc, 12 Octubre 2021. [En línea]. Available: https://www.fullstory.com/blog/categorical-vs-quantitative-data/#:~:text=Categorical%20data%20is%20a%20type,a%20method%20known%20as%20matching.. [Último acceso: 25 Julio 2022].

[27] Laerd Statistics, «Types of Variable,» Lund Research Ltd, 2018. [En línea]. Available: https://statistics.laerd.com/statistical-guides/types-of-variable.php. [Último acceso: 25 Julio 2022].

[28] C. Ochoa Sangrador y M. Molina Arias, «Estadística. Tipos de variables. Escalas de medida,» *Evidencias en Pediatría,* vol. 14, nº 2, 2018.

[29] J. Martínez Heras, «Las 7 Fases del Proceso de Machine Learning,» IArtificial.net, 19 Septiembre 2020.

[En línea]. Available: https://www.iartificial.net/fases-del-proceso-de-machine-learning/.

[30] Wikipedia, «Data binning,» Wikipedia, 15 Mayo 2022. [En línea]. Available: https://en.wikipedia.org/wiki/Data_binning. [Último acceso: 27 Julio 2022].

[31] D. Yadav, «Categorical encoding using Label-Encoding and One-Hot-Encoder,» Towards Data Science, 6 Diciembre 2019. [En línea]. Available: https://towardsdatascience.com/categorical-encoding-using-label-encoding-and-one-hot-encoder-911ef77fb5bd. [Último acceso: 26 Julio 2022].

[32] S. Morante, «Precauciones a la hora de normalizar datos en Data Science,» Telefónica S.A., 1 Noviembre 2018. [En línea]. Available: https://empresas.blogthinkbig.com/precauciones-la-hora-de-normalizar/. [Último acceso: 29 Julio 2022].

[33] H. Singh, «Data Normalization for Numeric features,» Analytics Vidhya, 26 Diciembre 2019. [En línea]. Available: https://medium.com/analytics-vidhya/data-transformation-for-numeric-features-fb16757382c0. [Último acceso: 29 Julio 2022].

[34] A. Cook, «Scaling and Normalization,» Kaggle, [En línea]. Available: https://www.kaggle.com/code/alexisbcook/scaling-and-normalization. [Último acceso: 30 Julio 2022].

[35] I. Guyon y A. Elisseeff, «An Introduction to Variable and Feature Selection,» *Journal of Machine Learning Research,* nº 3, pp. 1157-1182, 3 Marzo 2003.

[36] N. B. Subramanian, «Why High Dimensional Data are a Curse?,» AI Aspirant, 2019. [En línea]. Available: https://aiaspirant.com/curse-of-dimensionality/.

[37] S. Theodoridis y K. Koutroumbas, Pattern Recognition, Burlington, 2008.

[38] Aprende IA, «¿Por qué los modelos de Machine Learning no funcionan?,» Aprende IA, 2022. [En línea]. Available: https://aprendeia.com/que-hacer-para-que-un-modelo-de-machine-learning-funcione/.

[39] H. Bouzgou, «Developement of Multiple Regression Systems for Hyperdimensional Spectral Spaces,» ResearchGate GmbH, Septiembre 2006. [En línea]. Available: https://www.researchgate.net/figure/This-graph-illustrates-the-so-called-Hughes-effect-5-by-means-of-the-expected-mean_fig1_316455274.

[40] J. Brownlee, «What is Dimensionality Reduction,» de *Data Preparation for Machine Learning*, 2020, pp. 338-340.

[41] S. Kaushik, «Introduction to Feature Selection methods with an example (or how to select the right variables?),» Analytics Vidhya, 1 Diciembre 2016. [En línea]. Available: https://www.analyticsvidhya.com/blog/2016/12/introduction-to-feature-selection-methods-with-an-example-or-how-to-select-the-right-variables/. [Último acceso: 24 Julio 2022].

[42] W. K. Mutlag, S. K. Ali, Z. M. Aydam y B. H. Taher, «Feature Extraction Methods: A Review,» *Journal of Physics: Conference Series,* vol. 1591, 2020.

[43] J. A. Sánchez, «¿Cómo aprenden las máquinas? Machine Learning y sus diferentes tipos,» datos.gob.es, 31 Agosto 2020. [En línea]. Available: https://datos.gob.es/es/blog/como-aprenden-las-maquinas-machine-learning-y-sus-diferentes-tipos. [Último acceso: 31 Julio 2022].

[44] P. Recuero de los Santos, «Tipos de aprendizaje en Machine Learning: supervisado y no supervisado,» Telefónica Tech, 2 Diciembre 2021. [En línea]. Available: https://empresas.blogthinkbig.com/que-

algoritmo-elegir-en-ml-aprendizaje/. [Último acceso: 31 Julio 2022].

[45] P. Rudin, «SINGULARITY 2030,» SINGULARITY 2030, 13 Enero 2017. [En línea]. Available: https://singularity2030.ch/thoughts-on-human-learning-vs-machine-learning/. [Último acceso: 1 Agosto 2022].

[46] J. Martínez Heras, «¿Clasificación o Regresión?,» IArtificial.net, 29 Septiembre 2020. [En línea]. Available: https://www.iartificial.net/clasificacion-o-regresion/. [Último acceso: 31 Julio 2022].

[47] F. Berzal, «Clustering jerárquico,» Universidad de Granada, [En línea]. Available: https://elvex.ugr.es/idbis/dm/slides/42%20Clustering%20-%20Hierarchical.pdf. [Último acceso: 1 Agosto 2022].

[48] Wikipedia, «Agrupamiento jerárquico,» Wikipedia, 15 Mayo 2022. [En línea]. Available: https://es.wikipedia.org/wiki/Agrupamiento_jer%C3%A1rquico#cite_note-3. [Último acceso: 1 Agosto 2022].

[49] P. Larrañaga, I. Inza y A. Moujahid, «Tema 14. Clustering,» País Vasco: Universidad del País Vasco.

[50] M. Alloghani, D. Al-Jumeily, J. Mustafina, A. Hussain y A. J. Aljaaf, «A Systematic Review on Supervised and Unsupervised Machine Learning Algorithms for Data Science,» de *Supervised and Unsupervised Learning for Data Science*, Springer, 2020, pp. 9-15.

[51] J. Martínez Heras, «Árboles de Decisión con ejemplos en Python,» IArtificial.net, 19 Septiembre 2020. [En línea]. Available: https://www.iartificial.net/arboles-de-decision-con-ejemplos-en-python/#Arboles_de_Decision_para_Clasificacion. [Último acceso: 4 Agosto 2022].

[52] TIBCO Software Inc., «What is a Random Forest?,» TIBCO Software Inc., 2022. [En línea]. Available: https://www.tibco.com/reference-center/what-is-a-random-forest. [Último acceso: 4 Agosto 2022].

[53] F. Cardellino, «Cómo funcionan los clasificadores Naive Bayes: con ejemplos de código de Python,» freeCodeCamp, 28 Abril 2021. [En línea]. Available: https://www.freecodecamp.org/espanol/news/como-funcionan-los-clasificadores-naive-bayes-con-ejemplos-de-codigo-de-python/. [Último acceso: 4 Agosto 2022].

[54] J. Amat Rodrigo, «Máquinas de Vector Soporte (Support Vector Machines, SVMs),» cienciadedatos.net, Abril 2017. [En línea]. Available: https://www.cienciadedatos.net/documentos/34_maquinas_de_vector_soporte_support_vector_machines. [Último acceso: 4 Agosto 2022].

[55] GeeksforGeeks, «ML | Underfitting and Overfitting,» GeeksforGeeks, 4 Agosto 2022. [En línea]. Available: https://www.geeksforgeeks.org/underfitting-and-overfitting-in-machine-learning/. [Último acceso: 8 Agosto 2022].

[56] M. Singh Minhas, «Techniques for handling underfitting and overfitting in Machine Learning,» Towards Data Science, 6 Junio 2021. [En línea]. Available: https://towardsdatascience.com/techniques-for-handling-underfitting-and-overfitting-in-machine-learning-348daa2380b9. [Último acceso: 8 Agosto 2022].

[57] K. Nighania, «Various ways to evaluate a machine learning model's performance,» Towards Data Science, 30 Diciembre 2018. [En línea]. Available: https://towardsdatascience.com/various-ways-to-evaluate-a-machine-learning-models-performance-230449055f15. [Último acceso: 7 Agosto 2022].

[58] F. Krüger, *Activity, Context, and Plan Recognition with Computational Causal Behaviour Models,* Rostock, 2016.

[59] Raintwoto, «Receiver operating characteristic,» Wikipedia, 24 Junio 2010. [En línea]. Available: https://en.wikipedia.org/wiki/Receiver_operating_characteristic. [Último acceso: 7 Agosto 2022].

[60] The Machine Learners, «Encuentra tu modelo perfecto (Búsqueda de hiperparámetros),» The Machine Learners, 2022. [En línea]. Available: https://www.themachinelearners.com/busqueda-hiperparametros/. [Último acceso: 6 Agosto 2022].

[61] A. S. Gillis, «data splitting,» TechTarget, Abril 2022. [En línea]. Available: https://www.techtarget.com/searchenterpriseai/definition/data-splitting#:~:text=In%20machine%20learning%2C%20data%20splitting,into%20three%20or%20four%20sets.. [Último acceso: 5 Agosto 2022].

[62] scikit-learn, «Cross-validation: evaluating estimator performance,» scikit-learn, [En línea]. Available: https://scikit-learn.org/stable/modules/cross_validation.html. [Último acceso: 6 Agosto 2022].

[63] Z.-H. Zhou, «Neural Networks,» de *Machine Learning*, Singapur, Springer, 2021, pp. 104-123.

[64] I. de la Peña, «El problema de la caja negra: por qué la inteligencia artificial es una amenaza,» *El Confidencial,* 7 Febrero 2017.

[65] J. Areli-Toral Barrera, «Redes Neuronales,» [En línea]. Available: http://www.cucei.udg.mx/sites/default/files/pdf/toral_barrera_jamie_areli.pdf. [Último acceso: 9 Agosto 2022].

[66] M. Gestal Pose, «Introduccion a las Redes de Neuronas Artificiales,» Agosto 2013. [En línea]. Available: https://www.researchgate.net/publication/242099672_Introduccion_a_las_Redes_de_Neuronas_Artificiales. [Último acceso: 9 Agosto 2022].

[67] E. Freire y S. Silva, «Redes neuronales,» Bootcamp AI, 14 Noviembre 2019. [En línea]. Available: https://bootcampai.medium.com/redes-neuronales-13349dd1a5bb. [Último acceso: 11 Agosto 2022].

[68] A. D. Rasamoelina, F. Adjailia y P. Sinčák, *A Review of Activation Function for Artificial Neural Network,* Herl'any, 2020, pp. 281-286.

[69] S. Thye Goh, «Activation function at output layer for multi label classification,» Stack Exchange, 6 Marzo 2019. [En línea]. Available: https://stats.stackexchange.com/questions/395934/activation-function-at-output-layer-for-multi-label-classification. [Último acceso: 9 Agosto 2022].

[70] C. S. Vega, «¿Qué es una Red Neuronal? Parte 3.5 : Las Matemáticas de Backpropagation | DotCSV,» 2018.

[71] C. S. Vega, «¿Qué es el Descenso del Gradiente? Algoritmo de Inteligencia Artificial | DotCSV,» 2018.

[72] R. Sebastian, «An overview of gradient descent optimization algorithms,» 19 Enero 2016. [En línea]. Available: https://ruder.io/optimizing-gradient-descent/index.html. [Último acceso: 10 Agosto 2022].

[73] J. Brownlee, «Gradient Descent For Machine Learning,» Machine Learning Mastery, 23 Marzo 2019. [En línea]. Available: https://machinelearningmastery.com/gradient-descent-for-machine-learning/. [Último acceso: 10 Agosto 2022].

[74] J. Travnik y nbro, «Can a neural network with linear activation functions produce a connection of linear functions?,» Stack Exchange, 21 Mayo 2021. [En línea]. Available: https://ai.stackexchange.com/questions/3753/can-a-neural-network-with-linear-activation-functions-produce-a-connection-of-li/18765. [Último acceso: 11 Agosto 2022].

[75] dontloo, «Minimum number of layers in a deep neural network,» StackExchange, 13 Agosto 2016. [En línea]. Available: https://stats.stackexchange.com/questions/229619/minimum-number-of-layers-in-a-deep-neural-network. [Último acceso: 11 Agosto 2022].

[76] R. Tech, «Redes Neuronales Convolucionales - Clasificación avanzada de imágenes con IA / ML (CNN),» 2021.

[77] S. K. E, «Convolutional Neural Network (CNN),» Developers Breach, [En línea]. Available: https://developersbreach.com/convolution-neural-network-deep-learning/. [Último acceso: 13 Agosto 2022].

[78] Na8, «¿Cómo funcionan las Convolutional Neural Networks? Visión por Ordenador,» Aprende Machine Learning, 29 Noviembre 2018. [En línea]. Available: https://www.aprendemachinelearning.com/como-funcionan-las-convolutional-neural-networks-vision-por-ordenador/. [Último acceso: 13 Agosto 2022].

[79] J. Schäfer, P. Schmitt, M. Hlawitschka y H.-J. Bart, «Measuring Particle Size Distributions in Multiphase Flows Using a Convolutional Neural Network,» *Chemie Ingenieur Technik,* vol. 91, 2019.

[80] I. García y V. Caranqui, «La visión artificial y los campos de aplicación,» *Tierra Infinita,* nº 1, pp. 98-108, 2015.

[81] L. Gamboa Uribe, «Santander Global Tech,» 26 Noviembre 2020. [En línea]. Available: https://santandergto.com/vision-artificial-vs-vision-humana-asi-ven-los-ordenadores/.

[82] IBM, «What is computer vision?,» [En línea]. Available: https://www.ibm.com/topics/computer-vision.

[83] J. F. Vélez Serrano, Visión por computador, Madrid: Dykinson, 2003.

[84] M. F. Cárdenas Vera y O. R. Llerena Pizarro, «Automatización de un sistema de centrado de componentes utilizando visión artificial,» Cuenca, 2012.

[85] D. Iglesias, «Segmentación de Imágenes con Redes Convolucionales,» 27 Mayo 2021. [En línea]. Available: https://www.iartificial.net/segmentacion-imagenes-redes-convolucionales/#%C2%BFQue_es_la_segmentacion_de_imagenes.

[86] N. L. Fernández García, «Tema 1.- Introducción a la Visión Artificial: Segmentación,» [En línea]. Available: http://www.uco.es/users/ma1fegan/2012-2013/vision/Temas/segmentacion.pdf.

[87] GeeksforGeeks, «Image Segmentation using K Means Clustering,» GeeksforGeeks, 21 Julio 2021. [En línea]. Available: https://www.geeksforgeeks.org/image-segmentation-using-k-means-clustering/. [Último acceso: 5 Agosto 2022].

[88] Wikipedia, «Detección de características (visión por computadora),» [En línea]. Available: https://es.wikipedia.org/wiki/Detecci%C3%B3n_de_caracter%C3%ADsticas_(visi%C3%B3n_por_computadora).

[89] Y. Chtioui, D. Bertrand y D. Barba, «Feature selection by a genetic algorithm. Application to seed

discrimination by artificial vision,» *Journal of the Science of Food and Agriculture,* pp. 77-86, 26 Marzo 1999.

[90] ATRIA Innovation, «Guía sobre Visión Artificial para principiantes,» [En línea]. Available: https://www.atriainnovation.com/guia-sobre-vision-artificial-para-principiantes/.

[91] Google, «Google Landmark Recognition 2021,» Kaggle, 10 Agosto 2021. [En línea]. Available: https://www.kaggle.com/c/landmark-recognition-2021.

[92] C. Bupe, «Why do we need GPUs in AI?,» Quora, 2018. [En línea]. Available: https://www.quora.com/Why-do-we-need-GPUs-in-AI. [Último acceso: 15 Agosto 2022].

[93] Python Software Foundation, «The Python Logo,» Python Software Foundation, [En línea]. Available: https://www.python.org/community/logos/. [Último acceso: 15 Agosto 2022].

[94] W. Jarjoui, «Measure the Real Size of Any Python Object,» Shippo, 21 Julio 2016. [En línea]. Available: https://goshippo.com/blog/measure-real-size-any-python-object/. [Último acceso: 17 Agosto 2022].

[95] Keras, «Keras Applications,» Keras, [En línea]. Available: https://keras.io/api/applications/. [Último acceso: 20 Agosto 2022].

[96] K. Simonyan y A. Zisserman, «Very Deep Convolutional Networks for Large-Scale Image Recognition,» Oxford, 2015.

[97] A. Thite, «Introduction to VGG16 | What is VGG16?,» Great Learning, 1 Octubre 2021. [En línea]. Available: https://www.mygreatlearning.com/blog/introduction-to-vgg16/#VGG%20%E2%80%93%20The%20Idea. [Último acceso: 20 Agosto 2022].

[98] K. He, X. Zhang, S. Ren y J. Sun, *Deep Residual Learning for Image Recognition,* 2015.

[99] K. He, X. Zhang, S. Ren y J. Sun, «Deep Residual Learning for Image Recognition,» de *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 770-778.

[100] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens y Z. Wojna, *Rethinking the Inception Architecture for Computer Vision,* 2015.

[101] A. Narein T y B. QoChuk, «Inception V3 Model Architecture,» OpenGenus IQ, [En línea]. Available: https://iq.opengenus.org/inception-v3-model-architecture/. [Último acceso: 21 Agosto 2022].

[102] Google Cloud, «Guía avanzada de Inception v3,» Google, [En línea]. Available: https://cloud.google.com/tpu/docs/inception-v3-advanced?hl=es_419. [Último acceso: 21 Agosto 2022].