Trabajo Fin de Máster
Máster Universitario en Ingeniería Industrial

Diseño y entrenamiento de una red neuronal para el reconocimiento de imágenes de señales de tráfico

Autor: Álvaro Caballero Novoa
Tutor: Alejandro J. Del Real Torres

Dpto. de Ingeniería de Sistemas y Automática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2023

Trabajo Fin de Máster
Máster Universitario en Ingeniería Industrial

Diseño y entrenamiento de una red neuronal para el reconocimiento de imágenes de señales de tráfico

Autor:
Álvaro Caballero Novoa

Tutor:
Alejandro J. Del Real Torres
Profesor Contratado Doctor

Dpto. de Ingeniería de Sistemas y Automática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2023

Trabajo Fin de Máster: Diseño y entrenamiento de una red neuronal para el reconocimiento de imágenes de señales de tráfico

Autor: Álvaro Caballero Novoa

Tutor: Alejandro J. Del Real Torres

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

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2023

El Secretario del Tribunal

A mis padres y a mi hermana

Agradecimientos

En primer lugar, quiero agradecer a mis padres su apoyo y su ayuda, no solo en este proyecto, sino también en otros muchos aspectos de mi carrera y de mi vida.

Mi agradecimiento también a Alejandro Del Real, tutor de este proyecto, y a Álvaro Cabeza, cotutor de este proyecto, por su dedicación y sus consejos.

Álvaro Caballero Novoa

Sevilla, 2023

Resumen

El reconocimiento de señales de tráfico es una tecnología de seguridad vial basada en visión artificial que está presente en cada vez más vehículos, de manera que sean capaces de identificar estas señales durante la circulación.

Actualmente, existen muchos sistemas comerciales de reconocimiento de señales de tráfico, pero solamente son capaces de reconocer un conjunto limitado de señales. Sin embargo, existen conjuntos de datos con imágenes de muchas más señales de tráfico, por lo que se pueden crear modelos que sean más versátiles y robustos en esta tarea.

En este proyecto, se persigue diseñar y entrenar un modelo basado en redes neuronales profundas que sea capaz de reconocer y, por tanto, de clasificar imágenes de diversas señales de tráfico. Para ello, se programarán y probarán tres arquitecturas de redes neuronales convolucionales, se compararán los resultados obtenidos y se analizarán las métricas extraídas de cada una de ellas para comprobar cuál ofrece mejor rendimiento para esta tarea.

Abstract

Traffic sign recognition is a road safety technology based on computer vision that is present in more and more vehicles, so that they are capable of identifying these signs while driving.

Currently, there are many commercial traffic sign recognition systems, but they are only capable of recognizing a limited set of signs. However, there are datasets with images of many more traffic signs, so models that are more versatile and robust in this task can be created.

This project aims to design and train a model based on deep neural networks that is capable of recognizing and, therefore, classifying images of various traffic signs. To do this, three convolutional neural network architectures will be programmed and tested, the results obtained will be compared and the metrics extracted from each of them will be analyzed to check which one offers the best performance for this task.

Índice

ÍNDICE DE TABLAS

ÍNDICE DE FIGURAS

Notación

a^T	Vector *a* traspuesto
e	Número e
max	Función máximo
ms	Milisegundo
s	Segundo
tanh	Función tangente hiperbólica
$\partial y/\partial x$	Derivada parcial de *y* respecto de *x*
Σ	Sumatorio
σ	Función sigmoide
$\lfloor \ \rfloor$	Función suelo

1 INTRODUCCIÓN

El reconocimiento de señales de tráfico (RST o TSR, por sus siglas en inglés) es una tecnología de seguridad vial presente en cada vez más vehículos que es capaz de reconocer señales de tráfico en tiempo real durante la conducción con el fin de mantener al conductor informado en todo momento y evitar despistes. En síntesis, se trata de un avisador automático de límites de velocidad, símbolos de prohibición o de maniobras involuntarias. De esta forma, el RST opera como un refuerzo de seguridad [1].

El primer sistema de reconocimiento de señales de tráfico se introdujo en el mercado de la mano de BMW en su modelo serie 7 del año 2008. Esta tecnología ha sido desarrollada por empresas como Ayonix y Continental, y en un primer momento solo detectaban límites de velocidad [1].

1.1. Motivación

Actualmente, existen muchos sistemas comerciales de reconocimiento de señales de tráfico que son capaces de reconocer solamente un conjunto limitado de señales. Sin embargo, existen conjuntos de datos con imágenes de muchas más señales de tráfico, por lo que se pueden crear modelos que sean más versátiles y robustos en esta tarea. La motivación principal de este proyecto es desarrollar un modelo inicial basado en redes neuronales profundas sobre el cual se podrían aplicar diferentes mejoras o capas para lograr un sistema de RST más completo.

Por otro lado, la motivación personal de este proyecto es adentrarse en el campo de la inteligencia artificial y, más concretamente, en el campo del aprendizaje profundo y las redes neuronales artificiales. Para ello, previamente a la realización del trabajo en sí, se ha cursado un programa de especialización sobre aprendizaje profundo llamado *Deep Learning Specialization* impartido por Andrew Ng en la plataforma Coursera [2].

Tras la realización de este programa de especialización, ya se dispone de los conocimientos teóricos y prácticos necesarios para el desarrollo de este trabajo, con el que he tenido la oportunidad de haber aprendido sobre un tema que apenas conocía y del que quería saber más, así como poder poner en práctica esos conocimientos adquiridos.

1.2. Objetivo

El objetivo de este proyecto es diseñar y entrenar una red neuronal que sea capaz de reconocer y, por ende, de clasificar diversas imágenes de señales de tráfico, concretamente 43 tipos de señales. Para ello, se programarán y probarán tres arquitecturas de redes neuronales convolucionales, se compararán los resultados obtenidos y se analizarán las métricas extraídas de cada una de ellas para comprobar cuál ofrece un mejor rendimiento para esta tarea.

Para alcanzar dicho objetivo, se seguirán una serie de pasos que serán expuestos detalladamente a lo largo de los siguientes capítulos de esta memoria:

- En primer lugar, se explicarán muchos de los conocimientos adquiridos en el programa de especialización. Se expondrá una breve historia de los hitos más relevantes de la inteligencia artificial, se abordarán los aspectos más importantes de las redes neuronales y se explicará la arquitectura y el funcionamiento de los tres modelos de redes neuronales convolucionales que se probarán. Cada uno de estos temas será expuesto en el capítulo 2.
- Después, se planteará el problema que se busca resolver en este trabajo y se explicarán los datos con los que se entrenará el modelo y las herramientas informáticas que se usarán para ello. Esto será comentado en el capítulo 3.

- Luego, se podrá pasar al diseño y el entrenamiento del modelo, en el que se probarán tres arquitecturas de redes neuronales convolucionales bajo cuatro casos diferentes cada una. Todo ello será detallado en el capítulo 4.
- A continuación, se deberán comparar los resultados obtenidos después del entrenamiento, analizando las métricas de cada una de las pruebas realizadas para verificar el rendimiento de cada modelo. Esto se abordará en el capítulo 5.
- Finalmente, en el capítulo 6, se obtendrán conclusiones sobre el análisis de los resultados junto con posibles modificaciones o ampliaciones futuras.

2 ESTADO DEL ARTE

Hace tiempo que la inteligencia artificial abandonó el espectro de la ciencia ficción y, aunque todavía en una fase muy inicial, está llamada a protagonizar una revolución equiparable a la que generó Internet. Sus aplicaciones en múltiples sectores, como la salud, las finanzas, el transporte o la educación, han provocado que la Unión Europea quiera regular su desarrollo y uso por la Ley de Inteligencia Artificial, la primera ley integral sobre IA del mundo [3], [4].

2.1. Historia de la inteligencia artificial

La idea de "una máquina que piensa" se remonta a la antigua Grecia. Sin embargo, desde la llegada de la informática electrónica, los acontecimientos clave en la evolución de la inteligencia artificial son los siguientes [5]:

- **1936.** Alan Turing, considerado el padre de la ciencia computacional y célebre por haber descifrado el aparato de comunicaciones Enigma de la Alemania nazi durante la Segunda Guerra Mundial, propuso un sistema teórico que aplicaba una serie de reglas a una cinta codificada para ejecutar cualquier algoritmo, lo que equivalía a una máquina de computación general [5], [6].
- **1943.** Walter Pitts y Warren McCulloch publicaron un influyente estudio titulado *A logical calculus of ideas immanent in nervous activity* (*Un cálculo lógico de ideas inmanentes a la actividad nerviosa*), en el que describían la neurona artificial, la primera formulación teórica de lo que después se llamaría red neuronal [6], [7].
- **1950.** Alan Turing publicó su famoso artículo *Computing Machinery and Intelligence* (*Maquinaria computacional e inteligencia*), en el que definía el "juego de imitación" como una prueba para comprobar la capacidad de una máquina de hacer creer a su interlocutor humano que ella también lo es. El llamado test de Turing ha permanecido desde entonces como una medida de la capacidad de "pensar" de una IA [5], [6].
- **1951.** Marvin Minsky y Dean Edmonds, dos alumnos de la Universidad de Harvard, diseñaron la SNARC (Stochastic Neural Analog Reinforcement Calculator o Calculadora de refuerzo analógico neuronal estocástico, en español), la primera máquina basada en una red neuronal [6], [7].

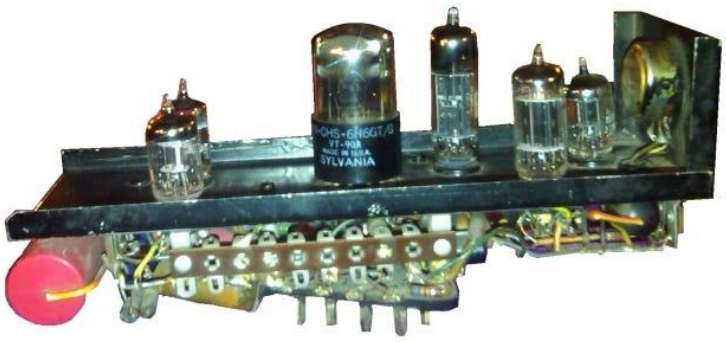

Figura 2-1. SNARC [8]

- **1952.** Arthur Samuel escribió el primer programa de ordenador capaz de aprender. Consistía en un programa que jugaba a las damas y que podía aprender de sus errores partida tras partida [7], [9].
- **1955.** Allen Newell, Herbert A. Simon y Cliff Shaw desarrollaron Logic Theorist (Teórico Lógico), un programa que, con su razonamiento automatizado, fue capaz de resolver 38 de 52 teoremas de la obra

Principia Mathematica[1]. Simon atribuyó a su programa la facultad de dar una mente a una máquina, lo que luego se llamaría IA fuerte o general. Logic Theorist se presentó en 1956 en el simposio fundacional de Dartmouth [6].

- **1956.** John McCarthy acuñó el término *inteligencia artificial* en el Dartmouth Summer Research Project on Artificial Intelligence [10], un simposio organizado por el propio McCarthy, junto con Marvin Minsky, Claude Shannon y Nathan Rochester. En este acto, los investigadores presentaron los objetivos y la visión de la IA, y muchos lo consideran como el verdadero nacimiento de la inteligencia artificial tal y como se conoce hoy en día [5], [6], [7].
- **1957.** Herbert Simon, J. C. Shaw, y Allen Newell crearon el programa General Problem Solver (Solucionador General de Problemas) con el objetivo de construir una máquina capaz de resolver problemas de carácter general [6], [11].
- **1958.** Frank Rosenblatt, inspirado en el trabajo de Walter Pitts y Warren McCulloch, creó el Perceptron Mark 1, el primer ordenador basado en una red neuronal que "aprendía" mediante prueba y error. Justo un año después, Marvin Minsky y Seymour Papert publican un libro titulado *Perceptrons*, que se convirtió tanto en la obra fundamental sobre redes neuronales como, al menos durante un tiempo, en un argumento en contra de futuros proyectos de investigación en redes neuronales [5], [12].
- **1959.** Arthur Samuel acuñó el término *aprendizaje automático* (*machine learning*, en inglés) mientras trabajaba en IBM. Por su parte, John McCarthy y Marvin Minsky fundaron ese mismo año el MIT Artificial Intelligence Project [6], [7].
- **1963.** John McCarthy creó el AI Lab en la Universidad de Stanford [6], [7].
- **1964.** Joseph Weizenbaum creó ELIZA entre 1964 y 1966, una psicoterapeuta automatizada en lenguaje natural, precursora de los chatbots [6], [13].

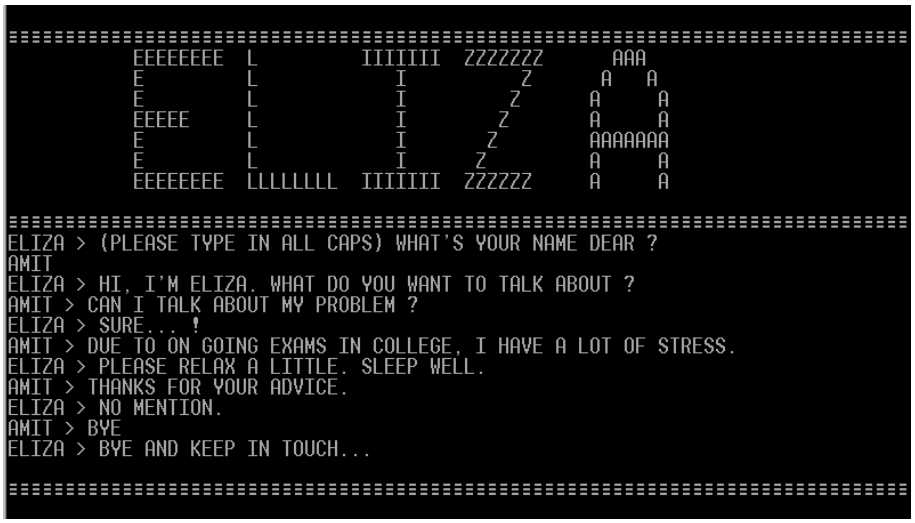

Figura 2-2. ELIZA, el primer chatbot de la historia [14]

- **1966.** Charles Rosen, Nils Nilsson, Peter Hart y otros crearon Shakey entre 1966 y 1972 en Stanford. Fue el primer robot móvil capaz de percibir su entorno, tomar decisiones y comunicarse en lenguaje natural. Sus diseños inspiraron los vehículos autónomos, la robotización industrial o los *rovers* de Marte [6].
- **1970-1980.** Los grandes avances posteriores a Dartmouth sembraron un entusiasmo que llevó a los científicos a predecir la creación de una IA general en unos pocos años. Sin embargo, en la década de 1970, comenzaron a cernerse dudas sobre el campo de la IA [6], [7].

  En 1966, el informe estadounidense ALPAC puso de manifiesto la falta de avances en la investigación de la traducción automática destinada a traducir simultáneamente la lengua rusa en el contexto de la Guerra Fría, por lo que muchos proyectos financiados por el gobierno estadounidense fueron cancelados. Más

[1] *Principia Mathematica* es un conjunto de tres libros con las bases de la matemática escritos por Bertrand Russell y Alfred North Whitehead, y publicados entre 1910 y 1913 [123].

tarde, el gobierno británico publicó su informe Lighthill en 1973, en el que destacaba las decepciones de la investigación en IA. Una vez más, los proyectos de investigación fueron reducidos por los recortes presupuestarios. Este periodo de duda duró hasta 1980, conocido como el *primer invierno de la IA* [7].

- **1970.** Seppo Linnainmaa publicó la primera idea sobre la retropropagación, conocida como *modo inverso de diferenciación automática*, para calcular eficientemente la derivada de una función compuesta diferenciable que se puede representar como un gráfico, aplicando recursivamente la regla de la cadena a los componentes básicos de la función [15]. Sin embargo, no fue hasta 1986, cuando David Rumelhart, Geoffrey Hinton y Ronald Williamsen popularizaron el algoritmo de retropropagación en un artículo [15], [16].
- **Década de 1980.** Japón y Estados Unidos hicieron grandes inversiones en la investigación de la IA. Muchas empresas se gastaron más de mil millones de dólares al año en sistemas expertos y el sector no paraba de crecer.

  Desgraciadamente, el mercado de las máquinas Lisp[2] se desplomó en 1987 al surgir alternativas más baratas. Este fue el *segundo invierno de la IA*. Las empresas perdieron el interés por los sistemas expertos y los gobiernos de Estados Unidos y Japón abandonaron sus proyectos de investigación habiendo gastado miles de millones de dólares para nada [7].
- **1996.** El interés en la IA renació en la década de 1990, sobre todo gracias a un golpe de efecto, cuando la máquina Deep Blue de IBM venció al entonces campeón mundial de ajedrez Garry Kasparov. Aunque al final Kasparov ganó 3 partidas más, derrotando a Deep Blue, fue la primera vez que el hombre fue derrotado por la máquina. Se volvieron a enfrentar en 1997, pero esta vez con una nueva versión de la máquina llamada Deeper Blue. Se jugaron 6 partidas en las que la máquina resultó vencedora de nuevo [5], [6], [7], [9].

Figura 2-3. Garry Kasparov vs. Deep Blue [17]

- **2002.** La primera introducción de la IA en los hogares fue de un modo que pocos esperaban: a través de la limpieza del suelo. La compañía iRobot lanzó en ese año Roomba, el primer robot doméstico autónomo capaz de navegar y tomar decisiones gracias a un conjunto de sensores. En 2010, Roomba ingresó en el Robot Hall of Fame de la Universidad Carnegie Mellon [6].
- **2006.** Geoffrey Hinton presentó el concepto de *aprendizaje profundo* (*deep learning*, en inglés). Con este concepto, se explicaron los nuevos algoritmos que permitían que las computadoras distinguiesen diversos objetos y textos tanto en imágenes como en vídeos [9].
- **2008.** Google logró grandes avances en el reconocimiento de voz y lanzó esa función en sus aplicaciones para teléfonos inteligentes [7].

[2] Una máquina Lisp es una computadora de uso general destinada, gracias a su particular *hardware*, a ejecutar eficientemente programas escritos en Lisp, un lenguaje de programación desarrollado en 1958 por John McCarthy y sus colaboradores en el MIT [124], [125].

- **2011.** Apple presentó Siri, el primer asistente virtual con reconocimiento de voz e interacción con lenguaje natural en un *smartphone*. Google respondería en 2012 con Google Now, Microsoft con Cortana en 2014 y Amazon con Echo/Alexa el mismo año. También en 2011, la IA Watson de IBM ganó a los campeones del concurso televisivo estadounidense *Jeopardy!*, logrando el premio de un millón de dólares [5], [6], [9].
- **2012.** Jeff Dean y Andrew Ng crearon Google Brain, un proyecto cuyo propósito fue crear una red neuronal utilizando toda la capacidad de infraestructura de Google para detectar patrones en vídeos e imágenes. Esta red neuronal fue alimentada con 10 millones de vídeos de YouTube como conjunto de datos de entrenamiento y, gracias al aprendizaje profundo, aprendió a identificar vídeos de gatos sin que se le enseñara lo que es un gato. Este fue el inicio de una nueva era para el aprendizaje profundo [7], [9].
- **2014.** Un programa de ordenador, que obedecía al nombre de Eugene Goostman, superó el test de Turing. Se trataba de un chatbot que fue capaz de convencer al 33% de los jueces que participaron en la prueba de que estaban chateando con un niño ucraniano de 13 años. En ese mismo año, Facebook desarrolló DeepFace, un algoritmo que podía reconocer individuos en fotos al mismo nivel que los humanos [9].
- **2015.** El programa de redes neuronales AlphaGo de la compañía DeepMind de Google venció al campeón Fan Hui en el go (un juego de mesa oriental) por cinco victorias a cero. Al año siguiente, ganaría al también campeón Lee Sedol. AlphaGo aprendía de los juegos humanos, pero la siguiente versión, AlphaGo Zero, fue construida desde cero para aprender solamente jugando contra sí misma. Venció a su antecesora por cien a cero [6].
- **2020.** La segunda versión del programa de aprendizaje automático AlphaFold de DeepMind, basado en la experiencia aprendida por esta compañía con AlphaGo Zero, conseguía resolver la estructura tridimensional de virtualmente cualquier proteína, un problema científico de enorme complejidad que llevaba medio siglo esperando resolución. Este logro ha sido calificado como el avance más importante en la historia de la IA [6].
- **2022.** OpenAI lanzó al público su chatbot ChatGPT, convertido en una estrella mediática y en una fuente de tantas aplicaciones como controversias. ChatGPT y sus sucesores y competidores son actualmente la cara más visible de la IA, junto con las redes generativas antagónicas (RGA o GAN, por sus siglas en inglés), dedicadas sobre todo a la creación artística, pero también a la creación de *deepfakes*. De ChatGPT y sus equivalentes se discute si han superado el test de Turing, algo en lo que no hay acuerdo entre los expertos [6].

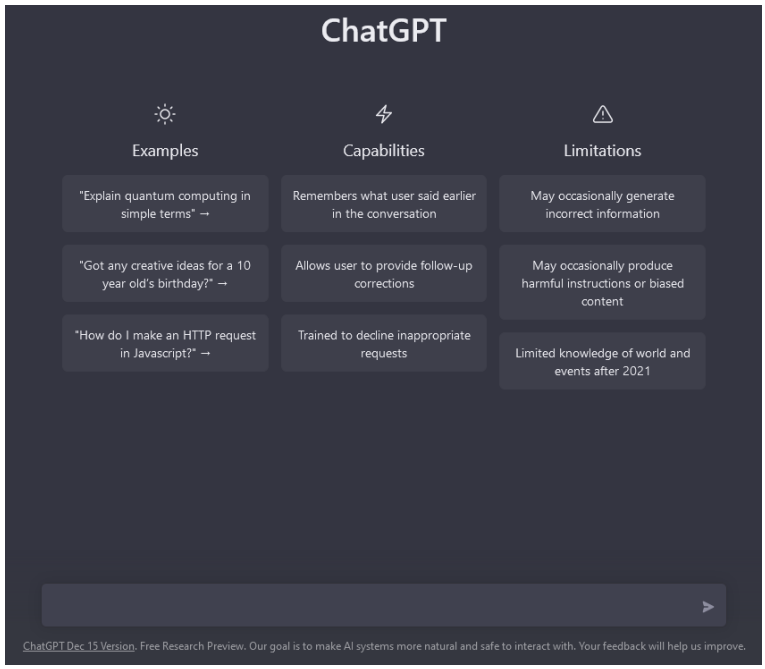

Figura 2-4. Interfaz de usuario de ChatGPT [18]

2.2. Redes neuronales

2.2.1. Relación entre inteligencia artificial, aprendizaje automático, aprendizaje profundo y redes neuronales

Antes de explicar qué es una red neuronal artificial, es importante conocer cuál es la diferencia entre inteligencia artificial, aprendizaje automático, aprendizaje profundo y red neuronal, ya que algunos de estos términos a veces se usan indistintamente a pesar de que expresen conceptos diferentes pero relacionados.

2.2.1.1. Inteligencia artificial

La inteligencia artificial (IA) es un campo de la informática dedicado a la resolución de problemas cognitivos asociados comúnmente a la inteligencia humana, como el aprendizaje, la resolución de problemas y el reconocimiento de patrones [19]. Es un campo amplio que abarca muchas disciplinas diferentes, incluidas la informática, el análisis de datos, la estadística, la ingeniería de *hardware* y *software*, la lingüística, la neurociencia y hasta la filosofía y la psicología [20].

Generalmente, se distinguen dos tipos de IA [5]:

- La IA débil, también llamada inteligencia artificial estrecha (IAE o ANI, por sus siglas en inglés), está entrenada y enfocada en la IA para realizar tareas específicas. La inteligencia artificial débil sustenta la mayor parte de la IA que nos rodea hoy en día. *Estrecha* podría ser un calificativo más preciso para este tipo de IA, ya que es cualquier cosa menos débil, pues permite algunas aplicaciones muy sólidas, como Siri de Apple, Alexa de Amazon, IBM watsonx y los vehículos autónomos.
- La IA fuerte, también conocida como inteligencia artificial general (IAG o AGI, por sus siglas en inglés), es una forma teórica de IA en la que una máquina tendría una inteligencia equiparable a la humana, tendría una conciencia capaz de resolver problemas, aprender y planificar el futuro. Aunque la IAG sigue siendo totalmente teórica y no existen ejemplos prácticos en la actualidad, eso no significa que los investigadores no estén explorando su desarrollo.

2.2.1.2. Aprendizaje automático

El aprendizaje automático es una rama de la inteligencia artificial enfocada en algoritmos que pueden aprender de datos registrados y hacer predicciones a partir de ellos, optimizar una función de utilidad determinada bajo certeza, extraer estructuras ocultas de datos y clasificar los datos en descripciones concisas. El aprendizaje automático se suele implementar cuando la programación explícita resulta demasiado rígida o poco práctica y, a diferencia de un código informático desarrollado para intentar generar una salida específica a partir de una entrada determinada, utiliza los datos para generar un código estadístico llamado modelo, que produce la salida adecuada basada en un patrón reconocido a partir de ejemplos anteriores de entrada-salida [19].

Existen principalmente tres modelos de aprendizaje automático [20]:

- Aprendizaje supervisado. Es un modelo de aprendizaje automático que asigna una entrada específica a un resultado mediante datos de entrenamiento etiquetados, es decir, datos estructurados. Por ejemplo, para entrenar un algoritmo que reconozca imágenes de gatos, se alimenta con imágenes etiquetadas como gatos.
- Aprendizaje no supervisado. Es un modelo de aprendizaje automático que aprende patrones en función de datos no etiquetados, es decir, datos no estructurados. A diferencia del aprendizaje supervisado, el resultado final no se conoce con anticipación. En cambio, el algoritmo aprende de los datos y los clasifica en grupos en función de diversos atributos. Por ejemplo, el aprendizaje no supervisado es bueno para identificar patrones y realizar modelado descriptivo.
- Aprendizaje por refuerzo. Es un modelo de aprendizaje automático que aprende a realizar una tarea definida mediante prueba y error hasta que su rendimiento está dentro de un rango deseado. El algoritmo recibe un refuerzo positivo cuando realiza la tarea correctamente y un refuerzo negativo cuando tiene bajo rendimiento. Un ejemplo de aprendizaje por refuerzo sería enseñar a una mano robótica a coger una pelota.

2.2.1.3. Aprendizaje profundo

El aprendizaje profundo es una rama del aprendizaje automático que conlleva la colocación por capas de los algoritmos con el fin de comprender mejor los datos. Los algoritmos ya no se limitan a crear un conjunto de relaciones explicables del mismo modo que una regresión básica, sino que se usan para crear representaciones distribuidas que interactúan a partir de una serie de factores. Con conjuntos de gran tamaño de datos de entrenamiento, los algoritmos de aprendizaje profundo comienzan a poder identificar las relaciones entre elementos, que pueden tener lugar entre figuras, colores o palabras, por ejemplo [19].

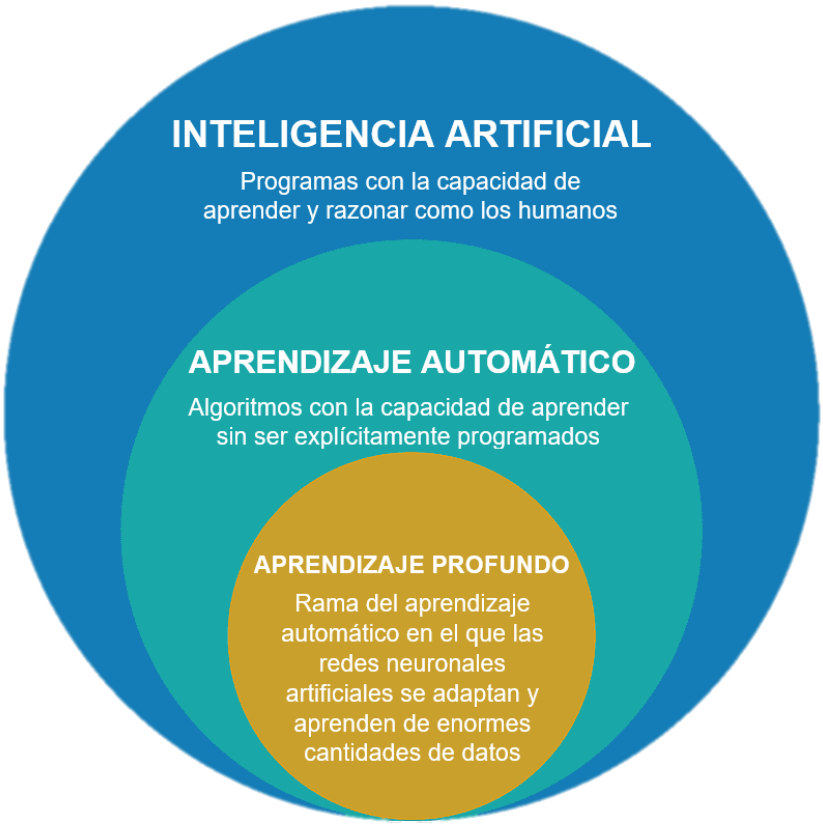

Figura 2-5. Inteligencia artificial, aprendizaje automático y aprendizaje profundo [21]

2.2.1.4. Redes neuronales

Las redes neuronales constituyen la base sobre la que se sustentan los algoritmos de aprendizaje profundo. Una red neuronal es un sistema de neuronas artificiales, que son nodos de procesamiento que se usan para clasificar y analizar datos. Los datos se introducen en la primera capa de una red neuronal, cada neurona toma una decisión y pasa esa información a varios nodos de la siguiente capa, y así sucesivamente. Los modelos de entrenamiento con más de tres capas se denominan redes neuronales profundas. Algunas redes neuronales modernas tienen cientos o miles de capas. La salida de la última capa de neuronas permite realizar la tarea impuesta a la red neuronal [20]. Con esto, se consigue crear un sistema adaptable que las computadoras utilizan para aprender de sus errores y mejorar continuamente, de manera que las redes neuronales puedan resolver problemas complicados, como la realización de resúmenes de documentos o el reconocimiento de rostros, con mayor precisión [22].

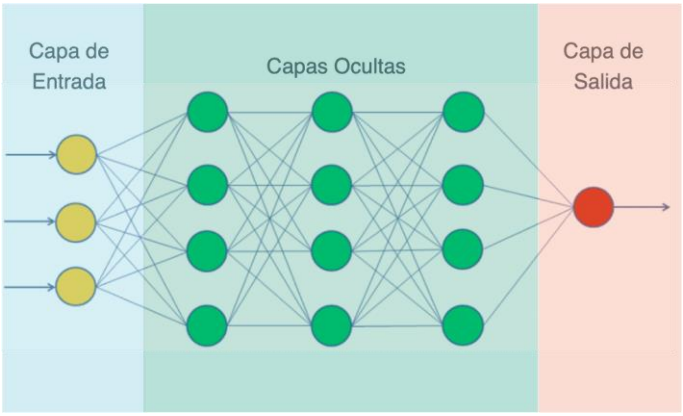

Figura 2-6. Red neuronal artificial [23]

2.2.2. El perceptrón

Como se acaba de comentar, una red neuronal está compuesta de neuronas, que son sus constituyentes elementales. De hecho, la red neuronal más simple que existe es aquella formada por una sola neurona, conocida como perceptrón, creado por Frank Rosenblatt en 1958 [12]. La principal aplicación del perceptrón es separar un conjunto de datos de entrada mediante una clasificación binaria [24].

En la Figura 2-7, se observa el modelo de un perceptrón. Este consta de n entradas, denominadas x_1, x_2, x_3… x_n. Cada una de estas entradas se multiplica por un coeficiente llamado peso, cuya notación es w_1, w_2, w_3… w_n. Al sumatorio del producto de cada entrada multiplicada por su peso, se suma también un coeficiente llamado sesgo, representado por b. De esta manera, si este resultado se denomina z, se obtiene:

$$z = \sum_{i=1}^{n} x_i w_i + b \tag{2–1}$$

Este resultado intermedio z pasa por una función de activación que genera la salida, denominada y, como un valor entre 0 y 1. Como el perceptrón debe clasificar una determinada entrada como perteneciente a una de las dos posibles clases, es decir, se espera que la salida sea un valor discreto (generalmente, 0 o 1), se suele tomar la salida como 0 si el valor generado por la función es menor que 0,5, o como 1 si el valor es igual o mayor que 0,5. De esta forma, la expresión que relaciona las entradas con la salida en el perceptrón es:

$$y = \sigma(z) = \sigma\left(\sum_{i=1}^{n} x_i w_i + b\right) \tag{2–2}$$

En la ecuación 2-2, σ representa la función de activación.

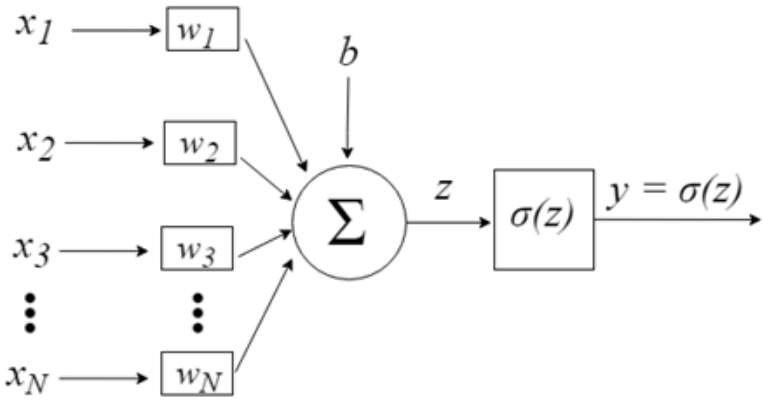

Figura 2-7. Diagrama del perceptrón [25]

2.2.3. Funciones de activación

Una función de activación es aquella que transmite la información generada por la combinación lineal de los pesos y las entradas, es decir, son la manera de transmitir la información por las conexiones de salida. La información puede ser transmitida sin modificaciones o simplemente no ser transmitida. Como el objetivo es que la red neuronal sea capaz de resolver problemas cada vez más complejos, las funciones de activación generalmente harán que los modelos sean no lineales. Entre las funciones de activación más usadas, se encuentran [26]:

- Función identidad. Esta función lineal devuelve una salida igual a la entrada, por lo que la señal pasa sin cambios [27]. Se usa principalmente si se requiere una regresión lineal a la salida [28], [29].

$$f(x) = x \tag{2–3}$$

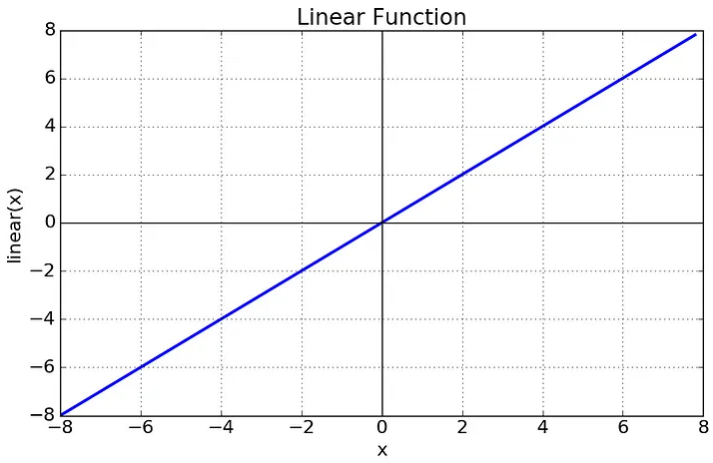

Figura 2-8. Función identidad [28]

- Función escalón. Esta función devuelve 0 si la entrada es menor que un cierto umbral, y 1 cuando la entrada es mayor o igual que dicho umbral (este umbral suele ser 0). Principalmente se usa cuando se quiere clasificar o cuando se tienen salidas categóricas [28].

$$f(x) = \begin{cases} 0 \; si \; x < 0 \\ 1 \; si \; x \geq 0 \end{cases} \tag{2–4}$$

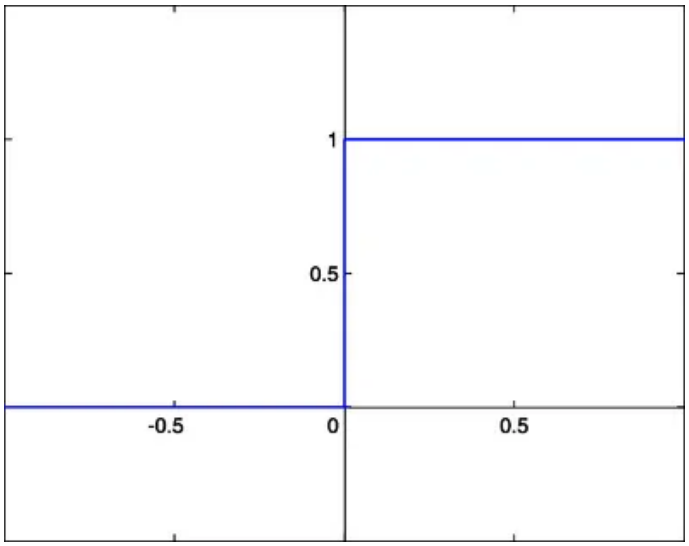

Figura 2-9. Función escalón [28]

- Función sigmoide o logística. Esta función tiene un rango de valores de salida entre 0 y 1, por lo que la salida es interpretada como una probabilidad. La salida de la función será 0 si se evalúa con valores de entrada muy negativos, será 0,5 si se evalúa en 0 y será 1 para valores de entrada muy positivos. Esta función se usa en la última capa de la red neuronal para clasificar datos en dos categorías [28], [29].

$$f(x) = \frac{1}{1 - e^{-x}} \tag{2–5}$$

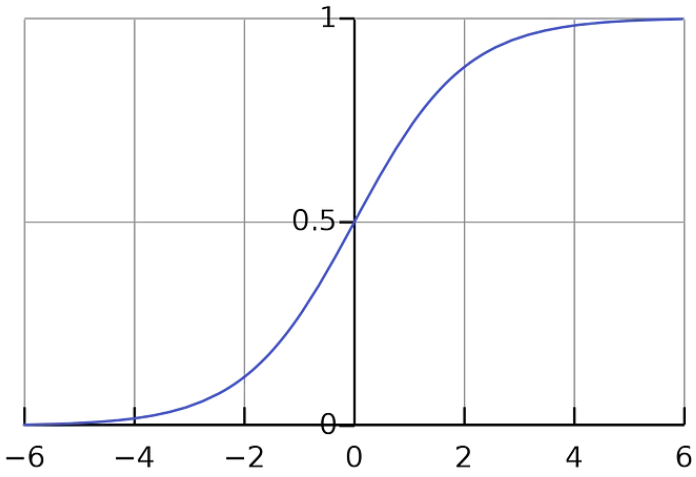

Figura 2-10. Función sigmoide o logística [28]

- Función tangente hiperbólica. Esta función de activación tiene un rango de valores de salida entre -1 y 1, y es un escalamiento de la función logística [28].

$$f(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \tag{2–6}$$

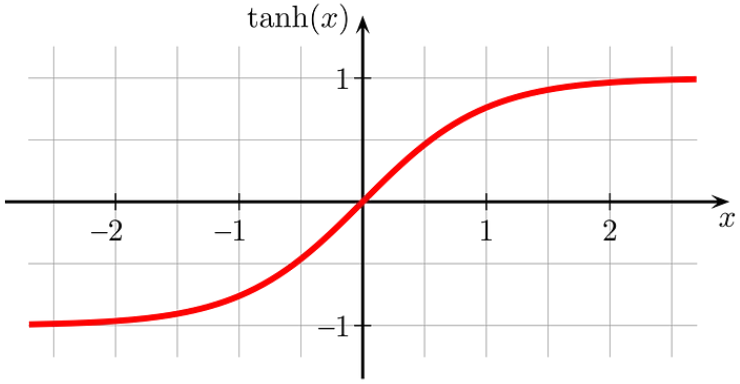

Figura 2-11. Función tangente hiperbólica [28]

- Función rectificadora o ReLU. Es la más utilizada debido a que permite un aprendizaje muy rápido en las redes neuronales. Esta función devuelve 0 para valores de entrada negativos o nulos, pero devuelve el valor de entrada si este es positivo. Además, el gradiente de esta función es 0 en el segundo cuadrante y 1 en el primer cuadrante [28].

$$f(x) = max(0, x) \tag{2–7}$$

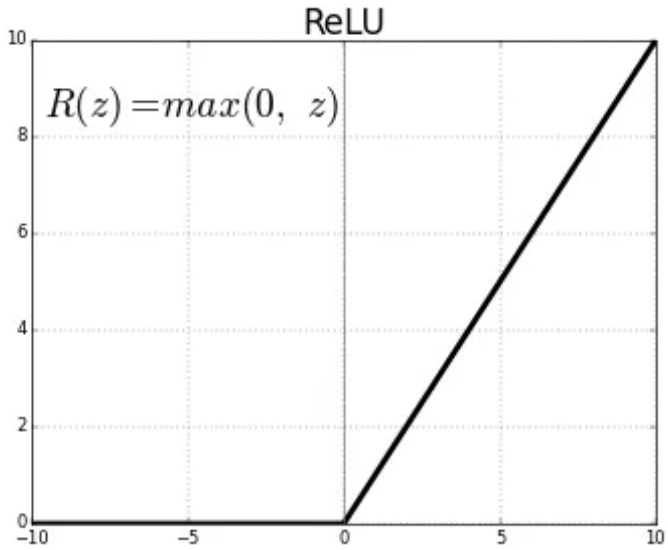

Figura 2-12. Función rectificadora o ReLU [28]

- Función SoftMax. Esta función de activación devuelve la distribución de probabilidad de cada una de las clases soportadas en el modelo, es decir, calcula la distribución de probabilidades del evento sobre *n* eventos diferentes. En términos generales, esta función calcula las probabilidades de cada clase sobre todas las clases posibles, de manera que su rango de salida estará comprendido entre 0 y 1 y la suma de todas las probabilidades será igual a 1. De esta forma, la clase objetivo será aquella con la probabilidad más alta [27].

$$f(x_i) = \frac{e^{x_i}}{\sum_{j=0}^{k} e^{x_j}} \quad j = 0, 1, 2, \ldots, k \quad \text{(k es el número de clases)} \tag{2–8}$$

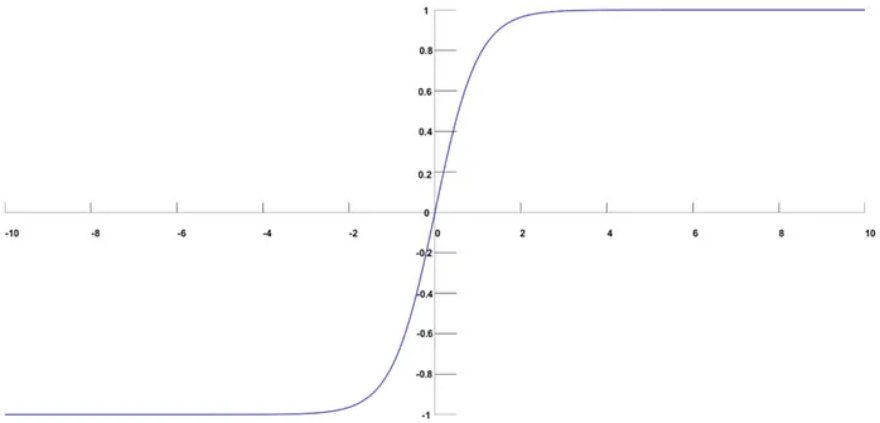

Figura 2-13. Función SoftMax [28]

2.2.4. Estructura de una red neuronal

Si se unen varias neuronas, se obtendrá una red neuronal más compleja donde cada neurona tendrá el mismo comportamiento que un perceptrón. En una red neuronal simple, las neuronas se agrupan formando capas dispuestas una tras otra, de modo que las neuronas de una capa estén conectadas con las de la capa anterior.

Existen generalmente tres tipos de capas en una red neuronal [22]:

- Capa de entrada. La información entra en la red neuronal artificial desde la capa de entrada. Los nodos de entrada procesan los datos, los analizan o los clasifican, y los pasan a la siguiente capa.
- Capa oculta. Las capas ocultas toman su entrada de la capa de entrada o de otras capas ocultas anteriores. Las redes neuronales pueden tener una gran cantidad de capas ocultas. Cada capa oculta analiza la salida

de la capa anterior, la procesa aún más y la pasa a la siguiente capa. Se denominan capas ocultas porque contienen unidades no observables [24].

- Capa de salida. La capa de salida proporciona el resultado final de todo el procesamiento de datos que realiza la red neuronal. Esta capa puede tener uno o varios nodos. Por ejemplo, en un problema de clasificación binaria, la capa de salida tendrá un nodo que dará como resultado 0 o 1. Sin embargo, en un problema de clasificación multiclase, la capa de salida puede estar formada por más de un nodo.

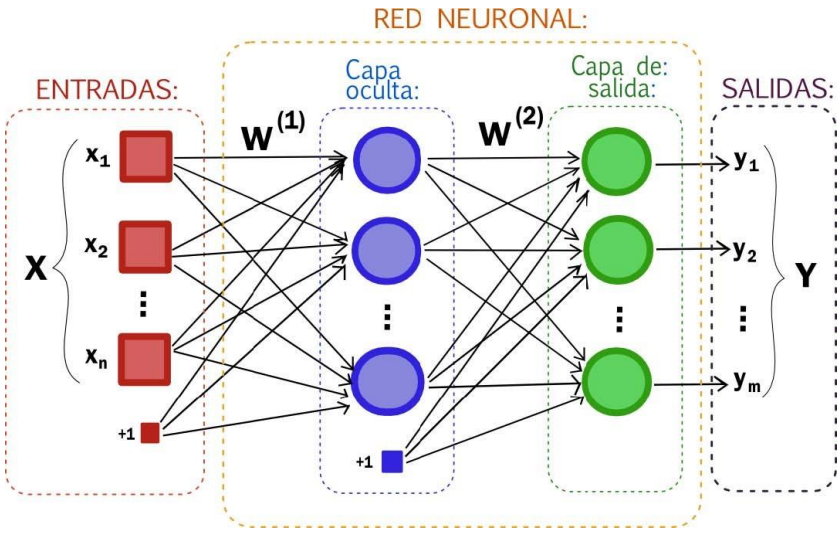

Figura 2-14. Estructura de una red neuronal [30]

2.2.5. Tipos de redes neuronales

Hay una gran variedad de redes neuronales que, en función de su tipología, tienen un funcionamiento distinto y son más útiles para la resolución de algunas tareas que de otras.

Aunque hay varias maneras de clasificar las redes neuronales, las tres más comunes son [31]:

- Según el número de capas. En esta clasificación, se pueden dividir las redes neuronales en dos grupos:
  - Redes neuronales monocapa. Son las más sencillas que existen. En ellas, las neuronas de la capa de entrada se conectan a las neuronas de la capa de salida. La capa de entrada también podría considerarse una capa pero, al no hacer cálculos, no se tiene en cuenta a la hora de clasificarse.
  - Redes neuronales multicapa. Entre las conexiones de entrada y de salida, existen diversas capas ocultas. Estas capas de neuronas pueden conectarse entre ellas o no.

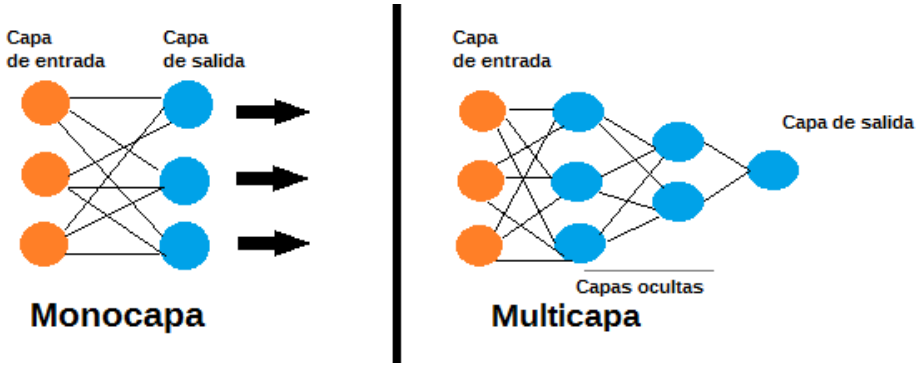

Figura 2-15. Redes neuronales monocapa y multicapa [31]

- Según el tipo de conexiones. En esta clasificación, existen dos tipos de conexiones:
  - Redes neuronales no recurrentes. En este tipo de redes neuronales, la información circula en un solo sentido. No existe ninguna realimentación y también carecen de memoria.
  - Redes neuronales recurrentes. Las neuronas tienen la posibilidad de realizar conexiones de realimentación, ya sea entre neuronas de una misma capa o de diferentes capas. Esto permite que

las redes neuronales recurrentes tengan memoria y suelan ser más potentes que las redes neuronales no recurrentes. Se usan principalmente para procesamiento de lenguaje natural o reconocimiento de voz [32].

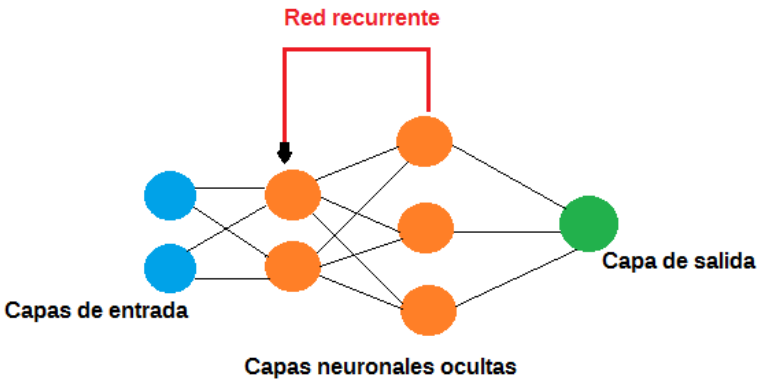

Figura 2-16. Red neuronal recurrente [31]

- Según el grado de conexiones. Pueden dividirse en dos grupos:
  - Redes neuronales completamente (o densamente) conectadas. En este tipo de redes neuronales, las neuronas de cada capa están conectadas a todas las de la capa anterior.
  - Redes parcialmente conectadas. En ellas, no todas las neuronas están conectadas entre sí para cada par de capas adyacentes.

Además, a la hora de conectar redes neuronales, se pueden realizar estructuras realmente complejas, en las que diferentes redes tienen determinadas jerarquías y trabajan en procesos diferentes. Algunos ejemplos son las redes neuronales convolucionales, las redes neuronales de base radial, las redes neuronales transformadoras, las redes neuronales siamesas, las redes generativas antagónicas y las redes neuronales líquidas [31].

- Una red neuronal convolucional es un tipo de red neuronal que se utiliza principalmente para analizar imágenes, clasificar elementos visuales y realizar diversas tareas de visión artificial [33]. La mayor ventaja de estas redes respecto a otros tipos es que diferentes partes de la red neuronal se pueden entrenar para tareas diversas, con lo que se consigue aumentar la velocidad de entrenamiento e identificar patrones de una forma más avanzada. Se hablará sobre este tipo de redes con mayor detalle más adelante, ya que es el que se ha usado para este proyecto [31].
- Las redes neuronales de base radial son aquellas en las que la salida es calculada en función de la distancia a un punto denominado centro, de manera que la salida de la red es una combinación lineal de funciones de base radial y parámetros neuronales. Sus aplicaciones son diversas, como la aproximación de funciones, la predicción de series de tiempo y la clasificación, pero también se pueden aplicar para otros casos de uso [31], [34].
- Las redes neuronales transformadoras se caracterizan por ser capaces de aprender contexto y, por lo tanto, significado mediante el seguimiento de relaciones en datos secuenciales como, por ejemplo, las palabras de una oración. Aplican un conjunto en evolución de técnicas matemáticas, conocido como autoatención, para detectar formas sutiles en que los elementos de datos en una serie dependen entre sí [35]. Son especialmente útiles para tareas de procesamiento de lenguaje natural, ya que entienden mejor el contexto que otros tipos de redes [31].
- Las redes neuronales siamesas son aquellas que contienen dos o más subredes que tienen la misma configuración y los mismos parámetros. Se utilizan principalmente para encontrar similitudes entre las entradas comparando sus vectores de características[3] [36]. Algunas aplicaciones de estas redes son el reconocimiento facial o la verificación de la autenticidad de un documento [31].

[3] Un vector de características es una representación numérica de un conjunto de características de un objeto o evento. En el aprendizaje automático, los vectores de características se utilizan para representar muestras de datos en un formato que los algoritmos puedan procesar fácilmente [126].

- Una red generativa antagónica es una arquitectura de aprendizaje profundo que entrena dos redes neuronales de modo que compitan entre sí para generar nuevos datos más auténticos a partir de un conjunto de datos de entrenamiento determinado. Este tipo de red se denomina *antagónica* porque entrena dos redes diferentes y las enfrenta entre sí. Una red genera nuevos datos al tomar una muestra de datos de entrada y modificarla en la medida de lo posible, mientras que la otra red intenta predecir si la salida de datos generada pertenece al conjunto de datos original. En otras palabras, la red de predicción determina si los datos generados son reales o falsos. El sistema genera versiones nuevas y mejoradas de valores de datos falsos hasta que la red de predicción ya no puede distinguir el falso del real. Un ejemplo de aplicación de este tipo de redes es la generación de imágenes [37].

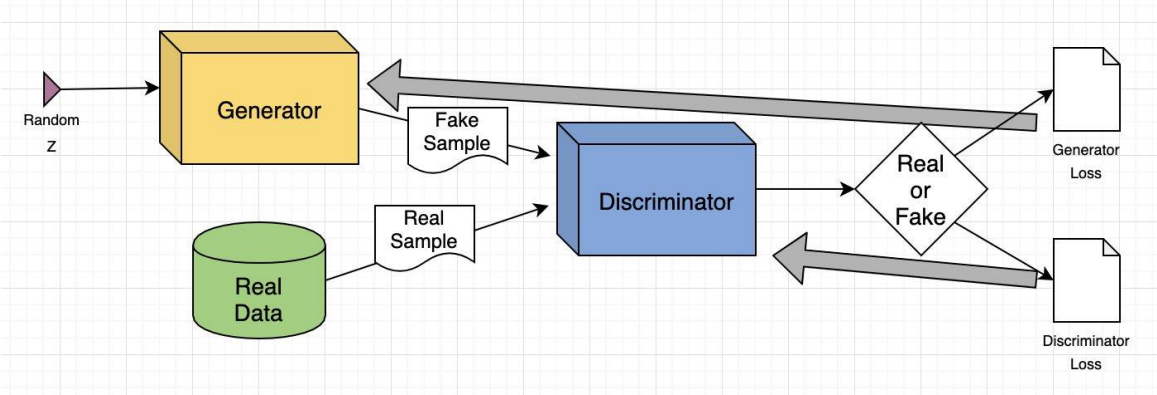

Figura 2-17. Red generativa antagónica [37]

- Una red neuronal líquida es un tipo de red neuronal capaz de aprender en base a sus experiencias, configurando sus parámetros en tiempo real. Entre sus ventajas destacan su capacidad para procesar datos de forma tanto temporal como espacial y el hecho de que se pueda emplear con menor potencia informática [31].

2.2.6. Aplicaciones

Las redes neuronales están presentes en varios casos de uso en muchos sectores, como los siguientes [22]:

- Diagnóstico médico mediante la clasificación de imágenes médicas.
- *Marketing* orientado mediante el filtrado de redes sociales y el análisis de datos de comportamiento.
- Predicciones financieras mediante el procesamiento de datos históricos de instrumentos financieros.
- Previsión de la carga eléctrica y la demanda de energía.
- Proceso y control de calidad.
- Identificación de compuestos químicos.

A continuación, se muestran cuatro de las aplicaciones más importantes de las redes neuronales.

2.2.6.1. Visión artificial

La visión artificial es la capacidad que tienen las computadoras para extraer información y conocimientos de imágenes y vídeos. Con las redes neuronales, las computadoras pueden distinguir y reconocer imágenes de forma similar a los humanos. La visión artificial tiene varias aplicaciones, como las siguientes [22]:

- Reconocimiento visual en los vehículos autónomos para que puedan reconocer las señales de tráfico y a otros usuarios de la carretera. Esta primera aplicación es la que se desarrollará en este trabajo.
- Moderación de contenido para eliminar de forma automática los contenidos inseguros o inapropiados de los archivos de imágenes y vídeos.
- Reconocimiento facial para identificar rostros y reconocer atributos como ojos abiertos, gafas y vello facial.

- Etiquetado de imágenes para identificar logotipos de marcas, ropa, equipos de seguridad y otros detalles de la imagen.

Sobre esta aplicación de las redes neuronales, se hablará más detalladamente en la subsección 2.5.4.

2.2.6.2. Reconocimiento de voz

Las redes neuronales pueden analizar el habla humana a pesar de los diferentes patrones de habla, el tono, el idioma y el acento. Los asistentes virtuales y el *software* de transcripción automática utilizan el reconocimiento de voz para realizar tareas como las siguientes [22]:

- Asistir a los agentes de los centros de llamadas y clasificar las llamadas automáticamente.
- Convertir las conversaciones clínicas en documentación en tiempo real.
- Subtitular con precisión vídeos y grabaciones de reuniones para aumentar el alcance del contenido.

2.2.6.3. Procesamiento de lenguaje natural

El procesamiento de lenguaje natural (PLN) es la capacidad de procesar texto natural creado por humanos. Las redes neuronales ayudan a las computadoras a obtener información y significado a partir de los datos y los documentos de texto. El PLN está presente en varios casos de uso, entre los que se incluyen los siguientes [22]:

- Chatbots y agentes virtuales automatizados.
- Organización y clasificación automáticas de datos escritos.
- Análisis de inteligencia empresarial de documentos con formato largo, como correos electrónicos y formularios.
- Indexación de frases clave que indican sentimientos, como los comentarios positivos y negativos en las redes sociales.
- Resumen de documentos y producción de artículos para un tema determinado.

2.2.6.4. Motores de recomendaciones

Las redes neuronales pueden hacer un seguimiento de la actividad de un usuario para elaborar recomendaciones personalizadas. También pueden analizar todo el comportamiento de los usuarios y descubrir productos o servicios nuevos que interesen a un usuario específico [22].

2.3. Entrenamiento de una red neuronal

2.3.1. Algoritmo de retropropagación

Como se ha explicado, una neurona puede ser considerada como una función matemática que recibe unos valores de entrada y genera un valor de salida. Además, los valores de entrada introducidos en la red se multiplican por unos parámetros llamados pesos. Estos parámetros determinan la influencia que tiene un valor de entrada en la función, es decir, tienen una influencia directa en el error que pueda cometer la red neuronal en sus predicciones, como se va a explicar más adelante. Por ello, para cuantificar el error entre la predicción y el valor real con el fin de optimizar los parámetros, se usa una función denominada función de coste [38].

Recuperando la ecuación del perceptrón, la salida de una neurona en función de sus entradas podía ser representada por la siguiente expresión:

$$\hat{y} = \sigma(z) = \sigma(w^T x + b) \tag{2–9}$$

En esta ocasión, el valor de salida será representado por $\hat{y}$ en vez de por y, a diferencia de la ecuación 2-2, de manera que el primero expresará el valor de salida predicho por el modelo y el segundo denotará el valor de

salida real. Por tanto, lo que se persigue es que el valor de $\hat{y}$ sea lo más parecido posible al valor de y. Además, x y w aparecen en la ecuación 2-9 representados como vectores cuya dimensión es igual al número de entradas. Así, si la neurona tiene n entradas, estos vectores serán:

$$x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \qquad w = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{bmatrix} \tag{2–10}$$

Dado que se quiere conocer si el modelo es capaz de predecir bien el valor de salida de las entradas, se usará una función denominada función de pérdida. En aprendizaje automático, se denomina *pérdida* el error cometido por una red neuronal en sus predicciones [39]. Por tanto, esta función de pérdida sirve para medir cuán bien predice el modelo, es decir, sirve para cuantificar el error de $\hat{y}$ respecto de y.

Sin embargo, la función de pérdida solamente mide el error para un dato o un ejemplo dado. Para saber el error cometido por la red neuronal en todos los datos introducidos como entrada, se emplea la función de coste. Así, para un conjunto de m datos, la función de coste J se expresa de la siguiente forma (L representa la función de pérdida y el superíndice (i) representa el i-ésimo dato o ejemplo):

$$J = \frac{1}{m}\sum_{i=1}^{m} L\left(\hat{y}^{(i)}, y^{(i)}\right) \tag{2–11}$$

Existen varias funciones de coste para medir el error de predicción de un modelo, entre las que destacan las siguientes [39]:

Tabla 2-1. Funciones de coste

Error absoluto medio (EAM)	Se calcula tomando la media de las diferencias absolutas entre los valores predichos y los valores reales.	$EAM = \frac{1}{m}\sum_{i=1}^{m}\left\|\hat{y}^{(i)} - y^{(i)}\right\|$
Error cuadrático medio (ECM)	Es muy similar al EAM excepto que eleva al cuadrado los errores individuales en lugar de calcular sus valores absolutos.	$ECM = \frac{1}{m}\sum_{i=1}^{m}\left(\hat{y}^{(i)} - y^{(i)}\right)^2$
Raíz del error cuadrático medio (RECM)	Se calcula simplemente tomando la raíz cuadrada del ECM.	$RECM = \sqrt{\frac{1}{m}\sum_{i=1}^{m}(\hat{y}^{(i)} - y^{(i)})^2}$
Entropía cruzada binaria (ECB)	Se usa para calcular la pérdida de una red neuronal que realiza una clasificación binaria, es decir, que predice una de dos clases posibles.	$ECB = -\frac{1}{m}\sum_{i=1}^{m}[y^{(i)}\log\hat{y}^{(i)} + \left(1 - y^{(i)}\right)\log\left(1 - \hat{y}^{(i)}\right)]$
Entropía cruzada categórica (ECC)	Se usa para tareas de clasificación multiclase. Se diferencia de la ECB en que $\hat{y}$ e y son valores de salida codificados en caliente[4].	$ECC = -\frac{1}{m}\sum_{i=1}^{m}\sum_{j=1}^{k} y_j^{(i)}\log\hat{y}_j^{(i)}$ (k representa el número de clases)

[4] La codificación en caliente (*one-hot encoding*, en inglés) sirve para crear variables ficticias que son variables duplicadas que representan un nivel de una variable categórica. La presencia de un nivel se representa con un 1 y la ausencia, con un 0 [127]. De este modo, en una clasificación multiclase de k clases diferentes, una etiqueta de valor numérico n pasaría a ser codificada en caliente como un vector de k componentes donde el n-ésimo componente es 1 y el resto de componentes es 0.

Entropía cruzada categórica dispersa (ECCD)	Se usa cuando hay un gran número de clases (por ejemplo, 1 000). Funciona con datos codificados con etiquetas (es decir, con números enteros) en lugar de datos codificados en caliente, lo que hace que el cálculo sea muy rápido cuando se trabaja con una gran cantidad de clases.	$ECCD = -\frac{1}{m}\sum_{i=1}^{m}\sum_{j=1}^{k} y_j^{(i)} \log \hat{y}_j^{(i)}$ (k representa el número de clases)

El objetivo será encontrar el valor mínimo de la función de coste para así minimizar el error cometido por la red neuronal en sus predicciones. Mediante el método del descenso de gradiente, se puede tratar de minimizar dicha función de coste [38].

El descenso de gradiente es un algoritmo de optimización iterativo que permite encontrar mínimos locales en una función diferenciable. La idea es tomar pasos de manera repetida en dirección opuesta al gradiente, ya que el gradiente apunta a los puntos de la función que tienen mayor crecimiento [40], [41]. No obstante, hay que tener en cuenta el tamaño de los pasos que se van a dar durante el algoritmo de descenso de gradiente. El tamaño de estos pasos se controla mediante la tasa de aprendizaje, que es un hiperparámetro[5] que afecta la velocidad a la que el algoritmo alcanza el mínimo de la función y, por tanto, los valores óptimos de los pesos [42].

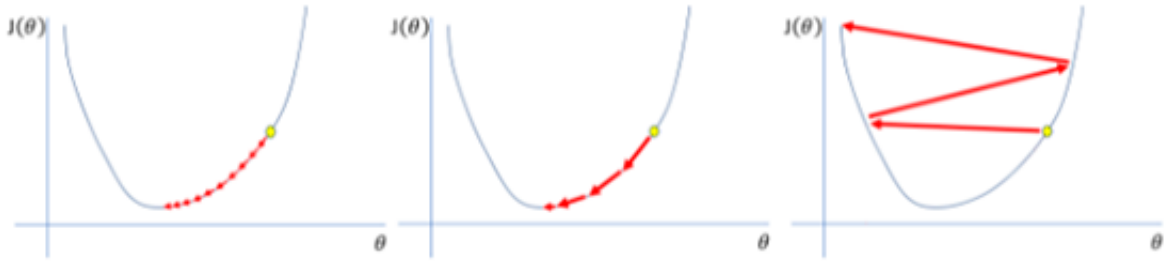

Figura 2-18. Representación del concepto de tasa de aprendizaje [43]

Como puede observarse en la Figura 2-18:

- Una tasa de aprendizaje demasiado pequeña, como la de la imagen de la izquierda, hace que el algoritmo converja en el mínimo, pero requiere un gran número de pasos. En la práctica, esto se traduce en un alto coste computacional, ya que habrá que realizar una gran cantidad de derivadas. Además, una tasa de aprendizaje pequeña puede hacer converger a mínimos locales en lugar de al mínimo absoluto.
- Una tasa de aprendizaje demasiado grande, como la de la imagen de la derecha, puede hacer que el algoritmo diverja del mínimo. Los pasos son tan largos que se hace muy difícil introducirse en la zona de mínimo coste [38].

Llegado a este punto, es necesario que la red neuronal sea capaz de encontrar por sí misma los valores óptimos de los parámetros, en vez de tener que reajustarlos manualmente. Para ello, las redes neuronales emplean un algoritmo denominado algoritmo de retropropagación.

El algoritmo de retropropagación fue presentado por primera vez en la década de 1960 y, varios años más tarde (1986), popularizado por David E. Rumelhart, Geoffrey E. Hinton y Ronald J. Williams en un artículo titulado *Learning representations by back-propagating errors* [16], [44]. Es un proceso involucrado en el entrenamiento de una red neuronal, e implica tomar la tasa de error de una propagación hacia adelante y alimentar esta pérdida hacia atrás a través de las capas de la red neuronal para ajustar sus parámetros [45].

Las redes neuronales prealimentadas procesan los datos en una dirección, desde la capa de entrada hasta la capa de salida. La retropropagación es un algoritmo que sirve para ajustar los parámetros de las conexiones

[5] Los parámetros del modelo son las variables que la técnica de aprendizaje automático seleccionada para el entrenamiento usa para ajustarse a los datos (por ejemplo, los pesos y los sesgos de las neuronas de la red), mientras que los hiperparámetros del modelo son las variables que rigen el proceso de entrenamiento en sí (por ejemplo, la tasa de aprendizaje, el número de capas de la red o el número de neuronas por capa) [128].

entre neuronas con el fin de reducir el valor de la función de coste, es decir, de minimizar la diferencia entre la salida predicha y la salida real y, de este modo, mejorar sus predicciones a lo largo del tiempo [22], [46].

Volviendo al ejemplo del perceptrón, su función de coste puede representarse mediante la función de entropía cruzada binaria, ya que la clasificación que realiza el perceptrón es binaria. De esta forma, su función de coste se puede expresar del siguiente modo, sustituyendo de ahora en adelante $\hat{y}$ por a:

$$J = -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}\log a^{(i)} + \left(1-y^{(i)}\right)\log\left(1-a^{(i)}\right)\right] \tag{2–12}$$

$$a^{(i)} = \hat{y}^{(i)} = \sigma\left(z^{(i)}\right) = \sigma\left(w^T x^{(i)} + b\right) \tag{2–13}$$

Puesto que el objetivo es minimizar la función de coste para así minimizar el error de $\hat{y}$ respecto de y, se usará el descenso de gradiente, como ya se ha explicado. De esta forma, se podrán actualizar los valores de los parámetros del perceptrón[6]. Las expresiones para hacer este cálculo son las siguientes (α representa la tasa de aprendizaje y el subíndice j representa la j-ésima entrada de la neurona):

$$w_j \coloneqq w_j - \alpha\frac{\partial J}{\partial w_j} \qquad\qquad b \coloneqq b - \alpha\frac{\partial J}{\partial b} \tag{2–14}$$

Aunque las expresiones sean sencillas, no es fácil calcular directamente las derivadas que aparecen en ellas porque la función de coste J depende de la función de pérdida L, que es función de $\hat{y}$ y esta, a su vez, es función de z, que es función de w y b. Por tanto, será necesario aplicar la regla de la cadena para calcular estas derivadas:

$$\frac{\partial L}{\partial w_j} = \frac{\partial L}{\partial a^{(i)}}\cdot\frac{\partial a^{(i)}}{\partial z^{(i)}}\cdot\frac{\partial z^{(i)}}{\partial w_j} \qquad\qquad \frac{\partial L}{\partial b} = \frac{\partial L}{\partial a^{(i)}}\cdot\frac{\partial a^{(i)}}{\partial z^{(i)}}\cdot\frac{\partial z^{(i)}}{\partial b} \tag{2–15}$$

La expresión de cada una de las derivadas es la siguiente:

$$\frac{\partial L}{\partial a^{(i)}} = -\frac{y^{(i)}}{a^{(i)}} + \frac{1-y^{(i)}}{1-a^{(i)}} \qquad \frac{\partial a^{(i)}}{\partial z^{(i)}} = a^{(i)}\left(1-a^{(i)}\right) \qquad \frac{\partial z^{(i)}}{\partial w_j} = x_j^{(i)} \qquad \frac{\partial z^{(i)}}{\partial b} = 1 \tag{2–16}$$

De esta forma, aplicando la regla de la cadena, se obtiene:

$$\frac{\partial L}{\partial w_j} = \left(-\frac{y^{(i)}}{a^{(i)}} + \frac{1-y^{(i)}}{1-a^{(i)}}\right)a^{(i)}\left(1-a^{(i)}\right)x_j^{(i)} = x_j^{(i)}\left(a^{(i)}-y^{(i)}\right) \tag{2–17}$$

$$\frac{\partial L}{\partial b} = \left(-\frac{y^{(i)}}{a^{(i)}} + \frac{1-y^{(i)}}{1-a^{(i)}}\right)a^{(i)}\left(1-a^{(i)}\right) = a^{(i)}-y^{(i)} \tag{2–18}$$

$$\frac{\partial J}{\partial w_j} = -\frac{1}{m}\sum_{i=1}^{m}\frac{\partial L\left(a^{(i)},y^{(i)}\right)}{\partial w_j} = -\frac{1}{m}\sum_{i=1}^{m}x_j^{(i)}\left(a^{(i)}-y^{(i)}\right) \tag{2–19}$$

$$\frac{\partial J}{\partial b} = -\frac{1}{m}\sum_{i=1}^{m}\frac{\partial L\left(a^{(i)},y^{(i)}\right)}{\partial b} = -\frac{1}{m}\sum_{i=1}^{m}\left(a^{(i)}-y^{(i)}\right) \tag{2–20}$$

[6] En el proceso de entrenamiento de una red neuronal, pueden escogerse unos valores iniciales aleatorios para los parámetros, pero existen métodos que ayudan a evitar problemas relacionados con gradientes que explotan o desaparecen, y a reducir el tiempo de entrenamiento, como la inicialización de He o la inicialización de Xavier [129].

Con estos cálculos, ya se pueden actualizar los valores de los parámetros mediante la ecuación 2-14 y las expresiones quedarían del siguiente modo:

$$w_j \coloneqq w_j + \frac{\alpha}{m}\sum_{i=1}^{m} x_j^{(i)}\left(a^{(i)} - y^{(i)}\right) \qquad b \coloneqq b + \frac{\alpha}{m}\sum_{i=1}^{m}\left(a^{(i)} - y^{(i)}\right) \tag{2–21}$$

Las ecuaciones que se han expuesto hasta ahora describen la propagación hacia adelante y hacia atrás[7] de una sola neurona (un perceptrón). No obstante, estas fórmulas son extrapolables a todas las neuronas que conforman una red neuronal para calcular la función de coste de dicha red y aplicar el algoritmo de retropropagación.

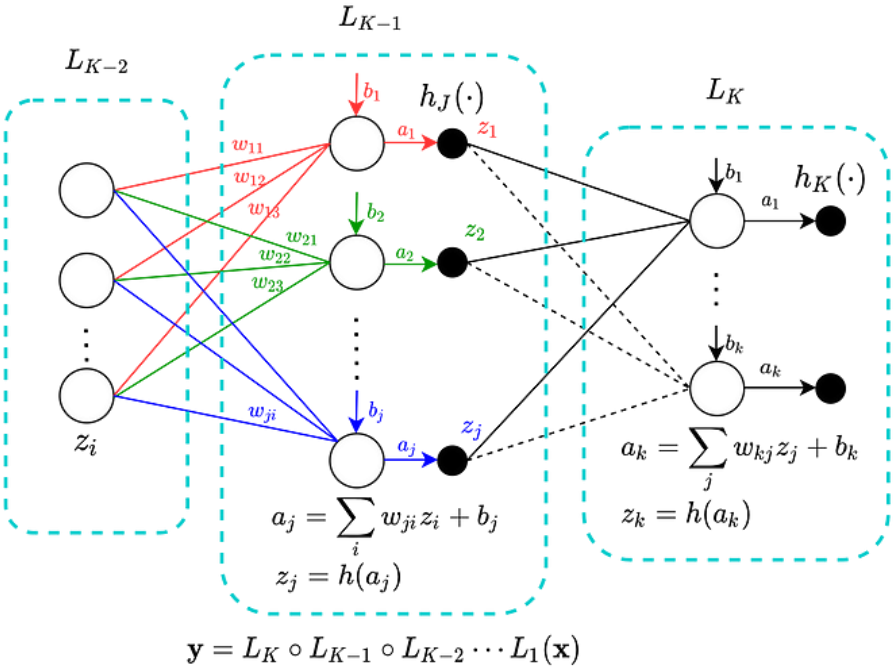

Figura 2-19. Propagación hacia adelante en una red neuronal [47]

Las redes neuronales artificiales aprenden de forma continua mediante el uso de bucles de retroalimentación correctivos para mejorar su análisis predictivo. Los datos fluyen desde la capa de entrada hasta la capa de salida a través de muchas rutas diferentes en la red neuronal, pero solamente una ruta es la correcta: la que asigna el nodo de entrada al nodo de salida correcto. Para encontrar esta ruta, la red neuronal utiliza un bucle de retroalimentación que funciona de la siguiente manera [22]:

1. Cada nodo intenta adivinar el siguiente nodo de la ruta.
2. Se comprueba si la suposición es correcta. Los nodos asignan valores de peso más altos a las rutas que conducen a más suposiciones correctas y valores de peso más bajos a las rutas de los nodos que conducen a suposiciones incorrectas.
3. Para el siguiente punto de datos, los nodos realizan una predicción nueva con las trayectorias de mayor peso y luego repiten el paso 1.

[7] En aprendizaje profundo, se denomina propagación hacia adelante el proceso de introducir la entrada en la red neuronal para generar la salida, mientras que la propagación hacia atrás (o retropropagación) es el proceso en el que se calcula el error entre la salida predicha por la red neuronal y la salida real, y se ajustan los pesos y sesgos de cada neurona para reducir dicho error [130].

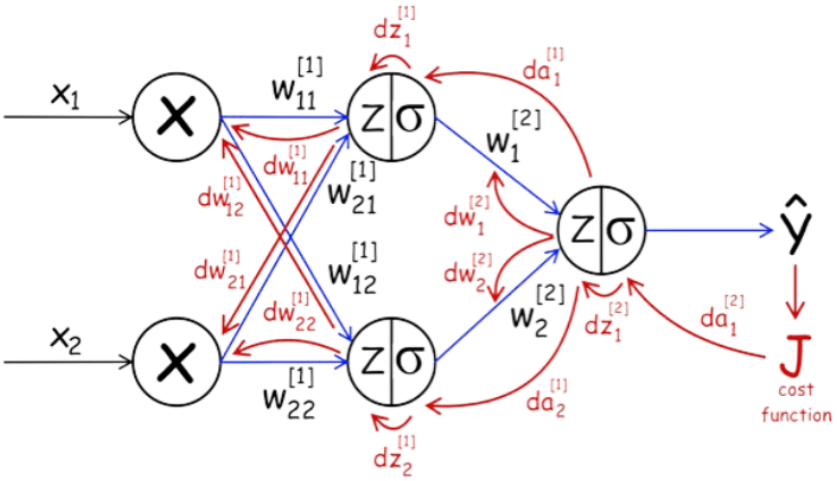

Figura 2-20. Retropropagación [48]

El motivo de este procedimiento radica en que un parámetro de entrada de alguna de las neuronas de una capa puede afectar la salida de alguna de las neuronas de la siguiente capa y, en consecuencia, el resultado final. Por consiguiente, el error de una capa depende directamente del error de las capas anteriores [49]. Para cada capa, se calcula la contribución relativa de cada neurona a la función de coste y se utilizan estas contribuciones para ajustar los parámetros de las conexiones entre neuronas [46]. Con esto, el algoritmo de retropropagación permite que el modelo pueda cambiar los parámetros cada vez que la salida predicha no sea la esperada y, de este modo, cuando un parámetro cambie, el error también cambia hasta que la red neuronal encuentra la salida deseada [50].

La retropropagación es un algoritmo muy popular porque es conceptualmente simple y computacionalmente eficiente [51]. Entre sus principales ventajas, cabe destacar [45]:

- Es sencillo de programar, ya que no hay otros parámetros además de las entradas.
- No es necesario aprender las características de una función, lo que acelera el proceso.
- El modelo es flexible por su simplicidad y aplicable a muchos escenarios.

Sin embargo, también presenta algunas limitaciones como las siguientes [45]:

- Los datos de entrenamiento pueden afectar el rendimiento del modelo, por lo que los datos de alta calidad son esenciales.
- Los datos ruidosos también pueden afectar la retropropagación, contaminando los resultados.
- Puede llevar algún tiempo entrenar modelos mediante retropropagación y ponerlos a punto.
- La retropropagación requiere un enfoque basado en matrices, lo que puede generar otros problemas.

2.3.2. Conjuntos de datos

Para entrenar una red neuronal, hacen falta datos. A partir de ellos, la red neuronal intenta encontrar o inferir patrones que le permitan predecir el resultado para un nuevo dato. Pero, para poder evaluar si un modelo funciona, será necesario probarlo con un conjunto de datos diferente. Por ello, en todo proceso de aprendizaje automático, los datos de trabajo se suelen dividir en tres partes [52]:

- Conjunto de datos de entrenamiento. Son los datos que se utilizan durante el proceso de aprendizaje para entrenar el modelo. La calidad del modelo va a ser directamente proporcional a la calidad de estos datos [52], [53].
- Conjunto de datos de validación. Son los datos usados para proporcionar una evaluación del ajuste del modelo a los datos de entrenamiento para ajustar los hiperparámetros del modelo o también, en caso de haber entrenado más de un modelo, para comparar métricas entre esos modelos [53].
- Conjunto de datos de prueba. Son los datos que se usan para proporcionar una evaluación del ajuste final del modelo a los datos de entrenamiento [53]. El conjunto de datos de prueba debe ser independiente del

conjunto de datos de entrenamiento, pero debe seguir la misma distribución de probabilidad que el conjunto de datos de validación [54].

En cuanto a la división del total de los datos disponibles en los conjuntos de entrenamiento, validación y prueba, esta depende principalmente de dos factores: la cantidad de datos de la que se dispone y el modelo que se va a entrenar [53].

- En cuanto a la cantidad de datos, si no es demasiado grande, es decir, unas 100, 1 000 o incluso 10 000 muestras de datos, la división de estos suele ser en torno a 60% para el conjunto de entrenamiento, en torno a 20% para el de validación y en torno a 20% para el de prueba. Sin embrago, para cantidades de datos mucho más grandes, es decir, cerca de 1 000 000 de muestras o más, la división de los datos suele ser de un 98% para el conjunto de entrenamiento, cerca de un 1% para el conjunto de validación y en torno a un 1% para el conjunto de prueba [55].
- Con respecto a las características del modelo, hay algunos que necesitan una cantidad sustancial de datos para ser entrenados, por lo que en este caso es aconsejable que el porcentaje del conjunto de datos de entrenamiento sea mayor. Los modelos con muy pocos hiperparámetros suelen ser fáciles de validar y ajustar, por lo que probablemente no necesiten un conjunto de validación grande. Por el contrario, si el modelo tiene muchos hiperparámetros, será aconsejable tener también un gran conjunto de validación. Además, si se tiene un modelo sin hiperparámetros o que no se pueden ajustar fácilmente, probablemente no sea necesario un conjunto de validación [53].

2.3.3. Sobreajuste y subajuste

Cuando se trabaja con un conjunto de datos para resolver un problema, se tiende a encontrar la precisión implementando el modelo con el conjunto de entrenamiento, después ajustando los hiperparámetros del modelo con el conjunto de validación y, finalmente, comprobando el desempeño final del modelo con el conjunto de prueba [56].

En caso de que la exactitud sea satisfactoria, se suele aumentar la exactitud de la predicción con el conjunto de datos, bien aumentando o disminuyendo la selección de las características, bien modificando las condiciones del modelo. Sin embrago, el modelo a veces puede dar resultados pobres haciendo esto. Este bajo rendimiento puede ser debido a que el modelo sea demasiado simple para describir el objetivo, efecto conocido como subajuste, o puede ser debido, por el contrario, a que el modelo sea demasiado complejo para expresar el objetivo, efecto conocido como sobreajuste [56].

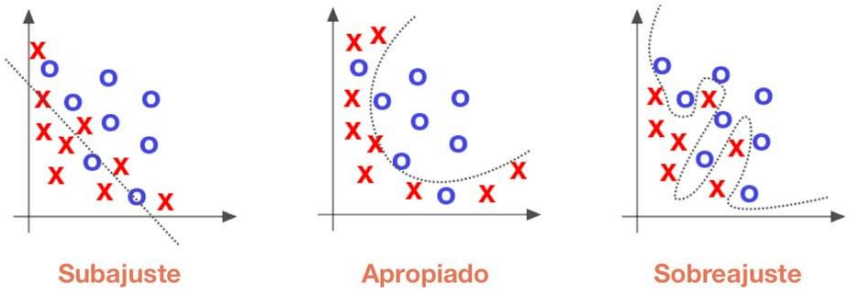

Figura 2-21. Representación de los conceptos de subajuste y sobreajuste [56]

En el lado izquierdo de la Figura 2-21, se observa que la línea predicha por el modelo no cubre todos los puntos que se muestran en el gráfico, lo que significa que el modelo tiende a causar un ajuste insuficiente de los datos. En este caso, se dice que el modelo tiene un alto sesgo [56].

En el lado derecho, se muestra que la línea predicha por el modelo cubre todos los puntos del gráfico. Se podría pensar que es un buen modelo ya que la línea cubre todos los puntos, pero en realidad no es cierto, porque también cubre todos los puntos que son ruido y valores atípicos. En este caso, se dice que el modelo tiene una alta varianza [56].

Por último, si se observa el gráfico situado en medio, se aprecia una línea bien predicha por el modelo, pues cubre la mayoría de los puntos en el gráfico y también mantiene el equilibrio entre el sesgo y la varianza [56].

Se describe a continuación en qué consisten el subajuste y el sobreajuste, y sus principales causas [56]:

- El subajuste se produce cuando un modelo no es capaz de ajustarse suficientemente bien a los datos de entrenamiento para así poder generalizar con nuevos datos, síntoma de que el modelo es muy simple. Suele suceder cuando se tienen menos datos para construir un modelo preciso y también cuando se intenta construir un modelo lineal con datos no lineales. La falta de adaptación se puede evitar utilizando más datos y también reduciendo las características mediante selección de características[8].
- El sobreajuste se produce cuando un modelo se ajusta "demasiado bien" a los datos de entrenamiento. Esto ocurre cuando un modelo aprende el detalle, incluido el ruido en los datos de entrenamiento, lo que tiene un impacto negativo en el rendimiento del modelo en datos nuevos. Esto significa que el ruido o las fluctuaciones aleatorias en los datos de entrenamiento son recogidos y aprendidos por el modelo. El problema es que estos conceptos no se aplican a los datos nuevos y afectan negativamente la capacidad de los modelos para generalizar. De entre las posibles soluciones que se pueden implementar para contrarrestar este efecto, la mejor técnica probablemente es la regularización del modelo.

2.3.4. Regularización

Una de las maneras con las que se puede contrarrestar el efecto del sobreajuste en un modelo es mediante técnicas de regularización. La regularización consiste en penalizar de alguna forma las predicciones de un modelo durante su entrenamiento con el objetivo de reducir su error de generalización, de manera que sepa generalizar ante otros conjuntos de datos. La cantidad de regularización afectará el rendimiento de validación del modelo, de tal manera que muy poca regularización no resolverá el problema de sobreajuste, y demasiada regularización hará que el modelo sea mucho menos efectivo, llegando a generar subajuste [57], [58].

Se explican a continuación algunas de las técnicas de regularización más comunes:

Tabla 2-2. Técnicas de regularización

Regularización L2	Esta técnica trata de reducir el valor de los parámetros para que sean pequeños. Introduce un término adicional de penalización en la función de coste *J*, añadiendo la suma de los cuadrados de los parámetros. Sin embargo, este nuevo término puede ser alto; tanto que la red minimizaría *J* haciendo los parámetros muy cercanos a 0, lo que no sería conveniente. Es por ello que ese sumando se multiplica por una constante pequeña, cuyo valor se escoge de forma arbitraria [59]. La función de coste *J* quedaría de este modo (λ es el parámetro de regularización y n es el número de entradas de la neurona correspondiente): $$J = \frac{1}{m}\sum_{i=1}^{m} L\left(\hat{y}^{(i)}, y^{(i)}\right) + \frac{\lambda}{2m}\sum_{j=1}^{n} w_j^2 = \frac{1}{m}\left[\sum_{i=1}^{m} L\left(\hat{y}^{(i)}, y^{(i)}\right) + \frac{\lambda}{2}\sum_{j=1}^{n} w_j^2\right]$$
Regularización L1	Esta técnica es muy parecida a la anterior, con la diferencia de que los parámetros en el término de penalización no se elevan al cuadrado, sino que se usa su valor absoluto. Esta variante empuja el valor de los parámetros hacia valores menores, haciendo incluso que la influencia de algunas variables de entrada sea nula en la salida de la red, lo que supone una selección de variables automática [59]. De este modo, la función de coste *J* quedaría de la siguiente forma: $$J = \frac{1}{m}\sum_{i=1}^{m} L\left(\hat{y}^{(i)}, y^{(i)}\right) + \frac{\lambda}{m}\sum_{j=1}^{n} \left|w_j\right| = \frac{1}{m}\left[\sum_{i=1}^{m} L\left(\hat{y}^{(i)}, y^{(i)}\right) + \lambda\sum_{j=1}^{n} \left|w_j\right|\right]$$

[8] La selección de características es el proceso de seleccionar las características más importantes o relevantes de un conjunto de datos con el principal objetivo de mejorar el rendimiento de predicción de un modelo [56].

Regularización por abandono (*dropout regularization*, en inglés)	Esta técnica implica desactivar una cierta cantidad de neuronas al azar durante el entrenamiento, lo que evita que la red aprenda una ruta singular de entrada a salida. Del mismo modo, debido a la gran parametrización de las redes neuronales, es posible que la red neuronal memorice la entrada. Sin embargo, con el abandono, esto es mucho más difícil, ya que la entrada se coloca en una red diferente cada vez, por lo que el abandono entrena efectivamente un número infinito de redes que son diferentes cada vez [58].
Aumento de datos	Esta técnica consiste en aplicar diversas transformaciones sobre las entradas originales, obteniendo muestras ligeramente diferentes pero iguales en esencia, lo que permite a la red desenvolverse mejor en la fase de inferencia. Es muy usada en el campo de la visión artificial, ya que esta técnica es especialmente útil cuando las entradas son imágenes. Algunos ejemplos de transformaciones sobre imágenes son: voltear horizontal o verticalmente, rotar, recortar, aplicar deformaciones de perspectiva, ajustar brillo o contraste, introducir ruido, etc. [59].
Parada temprana	Esta técnica consiste básicamente en detener el entrenamiento del modelo cuando se observa que la función de pérdida de validación comienza a aumentar, de manera que el modelo quede ajustado con los mejores parámetros hasta ese momento. Esto evita que los pesos crezcan demasiado y comiencen a empeorar el rendimiento de la validación en algún momento [58], [59].

2.3.5. Normalización por lotes

La normalización por lotes es un método algorítmico que hace que el entrenamiento de una red neuronal sea más rápido y más estable a través de la normalización de los vectores de activación de las capas ocultas, haciendo que, para cada minilote[9], su media sea cercana a 0 y su varianza, cercana a 1. Esta normalización suele ser aplicada justo antes de la función de activación [60].

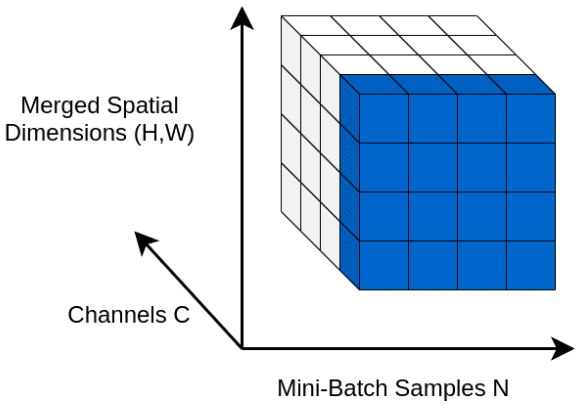

Figura 2-22. Normalización por lotes [61]

Aunque la normalización por lotes sea más una técnica de ayuda al entrenamiento que una estrategia de regularización, tiene cierto efecto regularizador. Esto último se logra realmente aplicando algo adicional conocido como momento. La idea de este momento es que, cuando introduzcamos un nuevo minilote de entrada, no se usen una media y una varianza muy distintas de las de la iteración anterior, para lo que se tendrá en cuenta el histórico, y se elegirá una constante que pondere la importancia de los valores del minilote actual frente a los valores del anterior. Gracias a esto, se puede conseguir reducir el sobreajuste [59].

[9] Un minilote en un conjunto de muestras procesadas en paralelo. En la práctica, se suelen usar minilotes formados por 32 o 64 muestras. Si el conjunto de datos es muy grande, se puede incluso usar minilotes de 128 muestras [60].

2.3.6. Algoritmos de optimización

Dado que el entrenamiento de una red neuronal puede ser un proceso muy lento, existen formas de acelerarlo, conocidas como algoritmos de optimización. Algunos de los algoritmos de optimización más comunes son:

Tabla 2-3. Algoritmos de optimización

Descenso de gradiente estocástico	Este algoritmo simplifica el cálculo del descenso de gradiente solo una muestra antes de obtener las derivadas parciales. Esto supone que las derivadas parciales obtenidas no van a ser óptimas, es decir, no van a apuntar en la dirección de mayor descenso de la función de coste, pero este problema se compensa con el mayor número de modificaciones a los parámetros que se aplican, lo que lleva a tiempos de entrenamiento mucho menores [62].
Descenso de gradiente por minilotes	Este algoritmo calcula el gradiente no a partir de todo el conjunto de entrenamiento ni a partir de una única muestra, sino a partir de un minilote. Normalmente, este algoritmo tarda menos en alcanzar el mínimo que el descenso de gradiente estocástico, pero corre el riesgo de caer en un mínimo local y no ser capaz de salir de él [63].
Descenso de gradiente con momento	Este algoritmo, también conocido simplemente como momento, es una variación del algoritmo de descenso de gradiente estocástico que utiliza una técnica de suavizado para evitar oscilaciones excesivas en la dirección del gradiente y aumentar la velocidad de convergencia. En lugar de actualizar los parámetros utilizando solo la información del gradiente actual, se usa un promedio ponderado del gradiente actual y de los gradientes anteriores para calcular la dirección de actualización, es decir, el gradiente que considerar para la actualización de los parámetros [64].
AdaGrad: Adaptive Gradient Algorithm (algoritmo de gradiente adaptativo)	Este algoritmo es una modificación del descenso de gradiente estocástico en la que se utilizan diferentes tasas de aprendizaje para las variables teniendo en cuenta el gradiente acumulado en cada una de ellas, de forma que a aquellas que acumulan un gradiente mayor se les aplica una tasa de aprendizaje menor y viceversa. No obstante, un posible problema es que en ocasiones puede ocurrir que la tasa de aprendizaje para una variable decrezca demasiado deprisa debido a la acumulación de valores altos del gradiente al comienzo del entrenamiento, lo que puede llevar a que el modelo no sea capaz de aproximarse al mínimo en dicha dimensión [65].
RMSProp: Root Mean Square Propagation (propagación del valor cuadrático medio)	Este algoritmo es una variación de AdaGrad en la que, en lugar de mantener un acumulado de los gradientes, se utiliza el concepto de "ventana" para considerar solo los gradientes más recientes, aplicándoles una media exponencial ponderada para suavizar los cambios aplicados a los parámetros. Este enfoque puede ayudar a evitar que las tasas de aprendizaje se vuelvan excesivamente pequeñas [66].
Adam: Adaptive Moment Estimation (estimación adaptativa de momento)	Este algoritmo combina las ventajas de los algoritmos RMSProp y momento para mejorar el proceso de aprendizaje de un modelo. Al igual que el algoritmo de momento, Adam utiliza una estimación del momento y de la magnitud de los gradientes anteriores para actualizar los parámetros del modelo en cada iteración. Sin embargo, en lugar de utilizar una tasa de aprendizaje constante para todos los parámetros, Adam adapta la tasa de aprendizaje de cada parámetro individualmente en función de su estimación del momento y de la magnitud del gradiente. Esto permite que el modelo se ajuste de manera más eficiente y efectiva a los datos de entrenamiento, lo que puede llevar a una mayor exactitud de la predicción en comparación con otros métodos de optimización [67].

2.3.7. Aprendizaje por transferencia

El aprendizaje por transferencia, o transferencia de aprendizaje, es una metodología en la que un modelo que ha sido entrenado para una tarea se utiliza como punto de partida para el entrenamiento de un modelo que realiza otra tarea similar. Suele ser mucho más fácil y rápido actualizar y volver a entrenar una red con transferencia de aprendizaje que entrenarla desde cero. Esta metodología se utiliza frecuentemente en aplicaciones de detección de objetos, reconocimiento de imágenes y reconocimiento de voz, entre otras [68].

La transferencia de aprendizaje es una técnica de uso habitual, ya que [68]:

- Permite entrenar modelos con menos datos etiquetados reutilizando modelos de uso habitual que ya han sido entrenados con conjuntos de datos de gran tamaño.
- Ayuda a reducir el tiempo de entrenamiento y los recursos informáticos. Los pesos no se aprenden desde cero, dado que el modelo previamente entrenado ya ha aprendido los pesos a partir de los aprendizajes anteriores.

Por ejemplo, en visión artificial, las redes neuronales se utilizan para resolver tareas relacionadas con imágenes, ya que pueden funcionar bien identificando características complejas de una imagen. Las capas densas contienen la lógica para detectar la imagen; de este modo, ajustar las capas superiores no afectará la lógica base. El reconocimiento de imágenes, la detección de objetos o la eliminación de ruido en imágenes son áreas de aplicación típicas del aprendizaje por transferencia porque todas estas tareas relacionadas con imágenes requieren la detección de características generales en ellas, como bordes o formas [69].

2.3.8. Aprendizaje multitarea

El aprendizaje multitarea es una técnica de aprendizaje automático en el que se resuelven varias tareas de aprendizaje a la vez, al mismo tiempo que se aprovechan los puntos en común y las diferencias entre dichas tareas [70].

Generalmente, el aprendizaje multitarea es útil cuando las tareas están relacionadas y cuando los datos son limitados, ya que permite que el modelo aproveche la información compartida entre tareas para mejorar el rendimiento de la generalización. Existen dos métodos de aprendizaje multitarea [71]:

- Compartición dura de parámetros. Se utilizan capas ocultas comunes para todas las tareas, pero se mantienen intactas las capas de salida del modelo específicas para cada tarea. Esta técnica es muy útil, ya que al aprender una representación para varias tareas mediante capas ocultas comunes, se reduce el riesgo de sobreajuste.
- Compartición blanda de parámetros. Cada modelo tiene sus propios conjuntos de pesos y sesgos, y la distancia entre estos parámetros en diferentes modelos se regulariza para que los parámetros se vuelvan similares y puedan representar todas las tareas.

2.3.9. Aprendizaje de extremo a extremo

El aprendizaje de extremo a extremo es una técnica de aprendizaje automático en la que se entrena una única red neuronal para tareas complejas utilizando directamente como entrada los datos sin procesar y sin ninguna extracción manual de características [72].

Debido a la creación de conjuntos de datos a gran escala, el aprendizaje profundo de extremo a extremo ha revolucionado una variedad de dominios como el reconocimiento de voz, la traducción automática, el reconocimiento facial, etc. [72]

En el aprendizaje automático tradicional, el entrenamiento de un modelo consta de al menos dos etapas [72]:

1. El objetivo de la primera etapa es generar características discriminatorias a partir de los datos de entrada sin procesar. Esto se hace utilizando conocimientos específicos para identificar las características más relevantes para cada tarea.
2. La siguiente etapa toma las características extraídas y genera las predicciones utilizando algún algoritmo tradicional de aprendizaje automático.

A pesar de su éxito, el procedimiento anterior lleva mucho tiempo y requiere conocimientos específicos en función de la tarea [72].

El aumento en el tamaño de los conjuntos de datos disponibles en los últimos años ha permitido el surgimiento del aprendizaje profundo de extremo a extremo, que tiene como objetivo aprender el mapeo de entrada-salida directamente de los datos sin extraer características manualmente. De este modo, se introducen los datos de entrada en una red neuronal generalmente grande y la red procesa estos datos y extrae automáticamente las características relevantes que luego se utilizan para generar predicciones [72].

Todo el procedimiento se realiza sin necesidad de ingeniería manual, lo que reduce la cantidad de tiempo y esfuerzo necesarios. El éxito de esta técnica radica en la enorme cantidad de datos usados durante el entrenamiento y la capacidad de las redes neuronales para aprender características de alto nivel simplemente entrenándolas con una gran cantidad de datos [72].

2.4. Rendimiento de una red neuronal

2.4.1. Matriz de confusión

Una vez que el modelo haya sido entrenado y probado, es interesante conocer qué tal funciona, es decir, cuán bueno es haciendo predicciones. En caso de haber entrenado y probado más de un modelo, también será interesante saber qué modelo hace mejores predicciones. Para ello, se puede usar una herramienta conocida como matriz de confusión.

La matriz de confusión permite visualizar en una tabla el desempeño de un modelo en base a sus predicciones. Es usada principalmente en modelos de clasificación [73].

Tabla 2-4. Matriz de confusión

	Valores predichos		
Valores reales		Positivos	Negativos
	Positivos	Verdaderos positivos	Falsos negativos
	Negativos	Falsos positivos	Verdaderos negativos

En la Tabla 2-4, se pueden observar cuatro casos posibles [74], [75]:

- Verdadero positivo (VP). En este caso, una entrada pertenece a una determinada clase y el modelo ha predicho que pertenece a esa misma clase.
- Falso negativo (FN). Ocurre cuando una entrada pertenece a una cierta clase, pero el modelo ha predicho que no pertenece a esa clase.
- Falso positivo (FP). Sucede cuando una entrada no pertenece a una determinada clase, pero el modelo ha predicho que sí pertenece.
- Verdadero negativo (VN). En este último caso, una entrada no pertenece a una cierta clase y el modelo ha predicho que efectivamente no pertenece a esa clase.

2.4.2. Métricas de rendimiento

A partir de los resultados obtenidos a través de la matriz de confusión, se pueden definir unas métricas de rendimiento del modelo. Las más usadas son [75]:

- Exactitud. Representa el porcentaje de predicciones correctas frente al número total de predicciones. Por tanto, es el cociente entre los casos bien clasificados por el modelo y la suma de todos los casos.

$$Exactitud = \frac{VP + VN}{VP + VN + FN + FP} \quad (2–22)$$

Sin embargo, cuando un conjunto de datos es poco equilibrado, la exactitud no es una métrica útil. Por ejemplo, si se intenta predecir una enfermedad rara y el algoritmo clasifica a todos los individuos como sanos, podría ser muy exacto (incluso un 99%), pero totalmente inútil. Por ello, se suele recurrir en estos casos a otras métricas como la exhaustividad, que representa la habilidad del modelo de detectar los casos relevantes.

- Precisión. Se refiere a cuán cerca está el resultado de una predicción del valor verdadero. Por tanto, es el cociente entre los casos positivos bien clasificados por el modelo y el total de predicciones positivas.

$$Precisión = \frac{VP}{VP + FP} \quad (2–23)$$

- Exhaustividad. También conocida como sensibilidad o recuperación [74], [75], representa la tasa de verdaderos positivos. Es la proporción entre los casos positivos bien clasificados por el modelo respecto al total de positivos. Representa, pues, la habilidad del modelo de detectar los casos relevantes.

$$Exhaustividad = \frac{VP}{VP + FN} \quad (2–24)$$

- Especificidad. Representa la tasa de verdaderos negativos. Es la proporción entre los casos negativos bien clasificados por el modelo respecto al total de negativos.

$$Especificidad = \frac{VN}{VN + FP} \quad (2–25)$$

Cabe mencionar una métrica más avanzada del rendimiento de un modelo denominada métrica F1. Esta métrica combina las métricas de precisión y exhaustividad, de manera que el resultado sea una media armónica de ambos valores [76].

$$F1 = \frac{2 \cdot precisión \cdot exhaustividad}{precisión + exhaustividad} \quad (2–26)$$

La conveniencia de usar una métrica u otra como medida de un modelo dependerá de cada caso en particular y, en concreto, del coste asociado a cada error de clasificación del algoritmo [75].

2.5. Redes neuronales convolucionales

Una red neuronal convolucional (RNC) es un tipo de modelo de aprendizaje profundo para procesar datos que tiene un patrón de cuadrícula, como las imágenes, que está inspirado en la organización de la corteza visual animal y está diseñada para aprender de manera automática y adaptativa jerarquías espaciales de características, de patrones más simples a patrones más complejos [77].

Las redes neuronales convolucionales reciben este nombre por una de las capas que las componen: la capa convolucional. Una capa convolucional está compuesta de una pila de operaciones matemáticas, como la convolución, un tipo especializado de operación lineal[10]. En las imágenes digitales, los valores de los píxeles se almacenan en una cuadrícula bidimensional, es decir, una matriz numérica, y a la que se aplica una pequeña cuadrícula de parámetros llamada filtro, un extractor de características optimizable, en cada posición de la imagen, lo que hace que las RNC sean muy eficientes para el procesamiento de imágenes, ya que una función

[10] En matemáticas, una convolución es un operador matemático que transforma dos funciones f y g en una tercera función que representa la magnitud en la que se superponen f y una versión trasladada e invertida de g [131].

puede aparecer en cualquier parte de la imagen. Conforme una capa alimenta con su salida la siguiente capa, las características extraídas pueden volverse más complejas de manera jerárquica y progresiva [77].

2.5.1. Paso y relleno

El paso y el relleno son dos conceptos básicos propios de las redes neuronales convolucionales que es necesario conocer.

El paso denota el número de píxeles que se desplaza el filtro con respecto a la matriz numérica en la operación de convolución. Tras una operación de convolución, la salida suele ser de menor tamaño que la entrada. Para mantener las dimensiones de salida como las de entrada, se puede utilizar una técnica conocida como relleno. El relleno consiste en añadir píxeles en los bordes de una imagen de manera simétrica, lo que permite un mejor análisis de la imagen y ayuda el filtro a mejorar su rendimiento. Además, el relleno puede ser muy útil para la detección de bordes en una imagen [78], [79].

Existen algunos tipos de relleno como relleno válido, relleno igual, relleno causal, relleno constante, reflexión y replicación. De estos, los más comunes son el relleno válido y el relleno igual. El primero consiste en no añadir relleno y el segundo implica añadir relleno de tal manera que la salida tenga el mismo tamaño que la entrada [79].

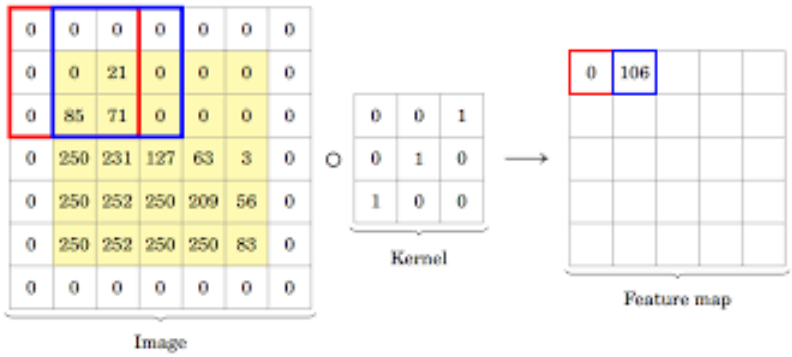

Figura 2-23. Convolución de una matriz con relleno [80]

La expresión que relaciona las dimensiones de la matriz de salida con las de la matriz de entrada en una convolución es la siguiente (*f* es el tamaño del filtro, *p* es el relleno y *s* es el paso):

$$n_{salida} = \left\lfloor \frac{n_{entrada} + 2p - f}{s} + 1 \right\rfloor \qquad (2\text{–}27)$$

2.5.2. Tipos de capas en una RNC

Las redes neuronales convolucionales se distinguen de otras redes neuronales por su alto rendimiento con entradas de imagen, voz o señales de audio. Se componen de tres tipos principales de capas: capa convolucional, capa de agrupación y capa completamente conectada [32].

La capa convolucional es la primera capa de una red convolucional. Si bien las capas convolucionales pueden ir seguidas de otras capas convolucionales o de capas de agrupación, la capa final es la capa completamente conectada. Con cada capa, la RNC aumenta en complejidad, identificando partes cada vez más grandes de la imagen. Las primeras capas se centran en características simples, como colores y bordes. Conforme los datos de la imagen avanzan a través de las capas, la RNC comienza a reconocer elementos o formas más grandes y complejos hasta que finalmente identifica el objeto esperado [32].

2.5.2.1. Capa convolucional

La capa convolucional es el bloque de creación principal de una RNC y es donde se realizan la mayoría de los cálculos. Requiere algunos componentes, como datos de entrada, un filtro y un mapa de características. Si la entrada es una imagen en color, estará compuesta por una matriz de píxeles en 3D, lo que significa que la entrada tendrá tres dimensiones: altura, anchura y profundidad. Esta última corresponde a la composición RGB (o RVA, en español) en una imagen. También hay un detector de características, conocido como filtro,

que se mueve por los campos receptivos de la imagen para comprobar si la característica está presente. Este proceso se denomina convolución [32].

El filtro es una matriz bidimensional de pesos que representa una parte de la imagen. Aunque su tamaño puede variar, el filtro suele ser una matriz de 3×3; esto también determina el tamaño del campo receptivo. A continuación, el filtro se aplica a un área de la imagen y se calcula un producto escalar entre los píxeles de entrada y el filtro, que se introduce en una matriz de salida. Después, el filtro se desplaza y repite el proceso hasta que el filtro haya recorrido toda la imagen. El resultado final de la serie de productos escalares de la entrada y el filtro se conoce como mapa de características[11], mapa de activación o característica convolucionada. Finalmente, tras cada operación de convolución, la RNC aplica una transformación de unidad lineal rectificada (ReLU) al mapa de características, introduciendo la no linealidad en el modelo [32].

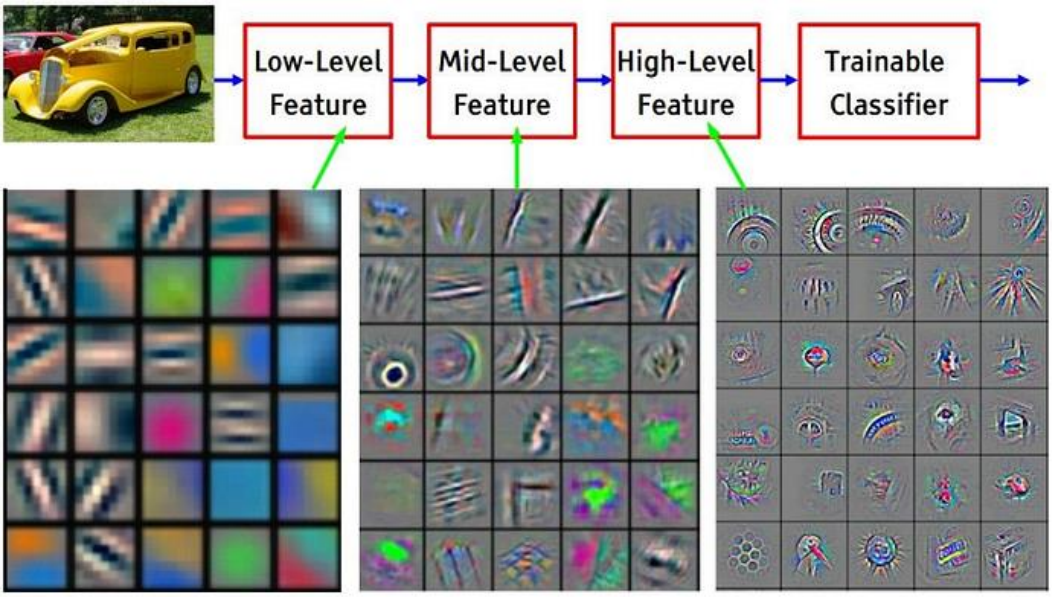

Figura 2-24. Características de distintos niveles [81]

Como se ha mencionado anteriormente, la capa de convolución inicial puede ir seguida de otra capa de convolución. Cuando esto sucede, la estructura de la RNC puede volverse jerárquica, ya que las capas siguientes pueden ver los píxeles en los campos receptivos de las capas anteriores. Por ejemplo, si se quiere determinar si una imagen contiene una bicicleta, se puede considerar la bicicleta como una suma de varias partes, es decir, compuesta de un cuadro, un manillar, ruedas, pedales, etc. Cada parte de la bicicleta forma un patrón de nivel inferior en la red neuronal, y la combinación de todas las partes representa un patrón de nivel superior, lo que crea una jerarquía de características dentro de la RNC [32].

2.5.2.2. Capa de agrupación

La agrupación de capas, también conocida como submuestreo, permite reducir la dimensión mediante la reducción del número de parámetros de entrada. De manera similar a la capa convolucional, la operación de agrupación barre toda la entrada con un filtro, pero la diferencia radica en que este filtro no tiene ningún peso. En su lugar, el filtro aplica una función de agrupación a los valores dentro del campo receptivo y llena así la matriz de salida. Hay dos tipos principales de agrupación [32]:

- Agrupación máxima. A medida que el filtro recorre la entrada, selecciona el píxel con el valor más alto para enviarlo a la matriz de salida. Este enfoque suele utilizarse más que la agrupación media.
- Agrupación media. Conforme el filtro avanza por la entrada, calcula el valor promedio dentro del campo receptivo para enviarlo a la matriz de salida.

[11] En las RNC, un mapa de características es la salida de una capa convolucional que representa características específicas en la imagen de entrada o en el mapa de características de la capa anterior. Cada mapa de características corresponde a un filtro específico y representa la respuesta de ese filtro a la imagen de entrada. Cada elemento en el mapa de características representa la activación de una neurona específica en la red y su valor representa el grado en que la característica correspondiente está presente en la imagen de entrada [132].

Aunque se pierde mucha información en la capa de agrupación, ayuda a reducir la complejidad, mejora la eficiencia y limita el riesgo de sobreajuste [32].

Figura 2-25. Agrupación máxima (izquierda) y agrupación media (derecha) [82]

2.5.2.3. Capa completamente conectada

El nombre de la capa completamente conectada describe con precisión la capa en sí. Como se ha mencionado anteriormente en el apartado 2.5.2.1, los valores de píxel de la imagen de entrada no están conectados directamente con la capa de salida en las capas parcialmente conectadas. Sin embargo, en la capa completamente conectada, cada nodo de la capa de salida sí está conectado directamente a un nodo de la capa anterior, como se explicó en la subsección 2.2.5 [32].

Esta capa realiza la tarea de clasificación basándose en las características extraídas de las capas anteriores y sus diferentes filtros. Las capas convolucionales y de agrupación suelen utilizar funciones de activación ReLU, mientras que las capas completamente conectadas generalmente usan una función de activación SoftMax para clasificar adecuadamente las entradas y generar una distribución de probabilidades entre 0 y 1 para cada una de las clases del modelo [32].

2.5.3. El neocognitrón

Kunikiho Fukushima fue el primero que introdujo, en un trabajo suyo de 1980, el concepto de neuronas convolucionales en el ámbito del aprendizaje profundo, apoyándose en los trabajos que los biólogos David Hubel y Torsten Wiesel habían realizado en la década de los años 60 sobre las células de la corteza visual. Hubel y Wiesel postularon la existencia de dos tipos de neuronas, a las que llamaron células simples y células complejas. Las células simples cubrían exclusivamente una porción limitada de la corteza visual y se encargaban de detectar, a través de las pautas de excitación e inhibición de sus dendritas, la existencia de patrones geométricos como líneas, aristas o esquinas. Las células complejas recibían sus aferencias de las células simples y se excitaban cada vez que un patrón (por ejemplo, una línea horizontal) era detectado en cualquier región del campo visual [82].

Fukushima concibió un tipo de perceptrón multicapa al que denominó neocognitrón, en el que se alternaban capas de células S (hoy en día, denominadas capas de convolución) y de células C (actualmente, capas de agrupación). Las primeras realizaban labores análogas a las de las células simples mientras que las segundas ejercían un papel similar al de las células complejas [82].

Cada capa convolucional se subdividía en planos (en la actualidad, mapas de características), y las células de cada uno de los planos (todas ellas con los mismos pesos) se encargaban de reconocer una determinada pauta visual, tal y como se muestra en la Figura 2-26 [82]:

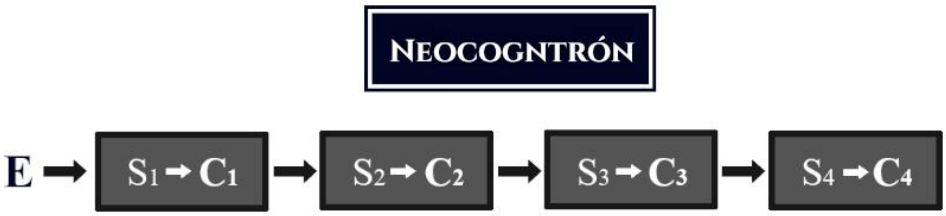

Figura 2-26. Diagrama del neocognitrón [82]

En la Figura 2-26, conforme se avanza hacia la derecha, las diferentes capas se encargan de reconocer patrones de cada vez mayor complejidad geométrica. Las capas de células complejas existentes entre las células simples realizan una suerte de pixelado de la salida de la capa anterior, reduciendo así su resolución [82].

El neocognitrón fue capaz de reconocer eficientemente dígitos con una resolución de 19×19 píxeles. Sin embargo, los pesos de sus capas de convolución tenían que ser determinados a mano, y Kunikiho Fukushima no llegó a diseñar ningún algoritmo de aprendizaje para optimizar automáticamente los pesos de la red [82].

2.5.4. Aplicaciones

Una de las principales aplicaciones de las RNC es la visión artificial. La visión artificial, o visión por computadora, es la capacidad que tienen las computadoras para extraer información significativa de imágenes, vídeos y otras entradas visuales, y así poder tomar una decisión o realizar alguna acción en base a esa información. Con las redes neuronales, las computadoras pueden distinguir y reconocer imágenes de forma similar a los humanos [22], [83].

La visión artificial necesita muchos datos. Ejecuta análisis de datos una y otra vez hasta que percibe diferencias y finalmente reconoce imágenes. En este caso, para entrenar un ordenador para que reconozca señales de tráfico, es necesario incorporarle una gran cantidad de imágenes de señales de tráfico para que aprenda las diferencias y pueda reconocerlas [83].

Una de las técnicas más comunes de visión artificial es el reconocimiento de objetos, que sirve para identificar objetos en imágenes o vídeos y constituye una salida clave de los algoritmos de aprendizaje profundo. Cuando las personas miramos una fotografía o vemos un vídeo, detectamos rápidamente personas, objetos, lugares y detalles visuales. El objetivo es enseñar a un ordenador a hacer lo que resulta natural para los humanos: adquirir cierto nivel de comprensión del contenido de una imagen [84].

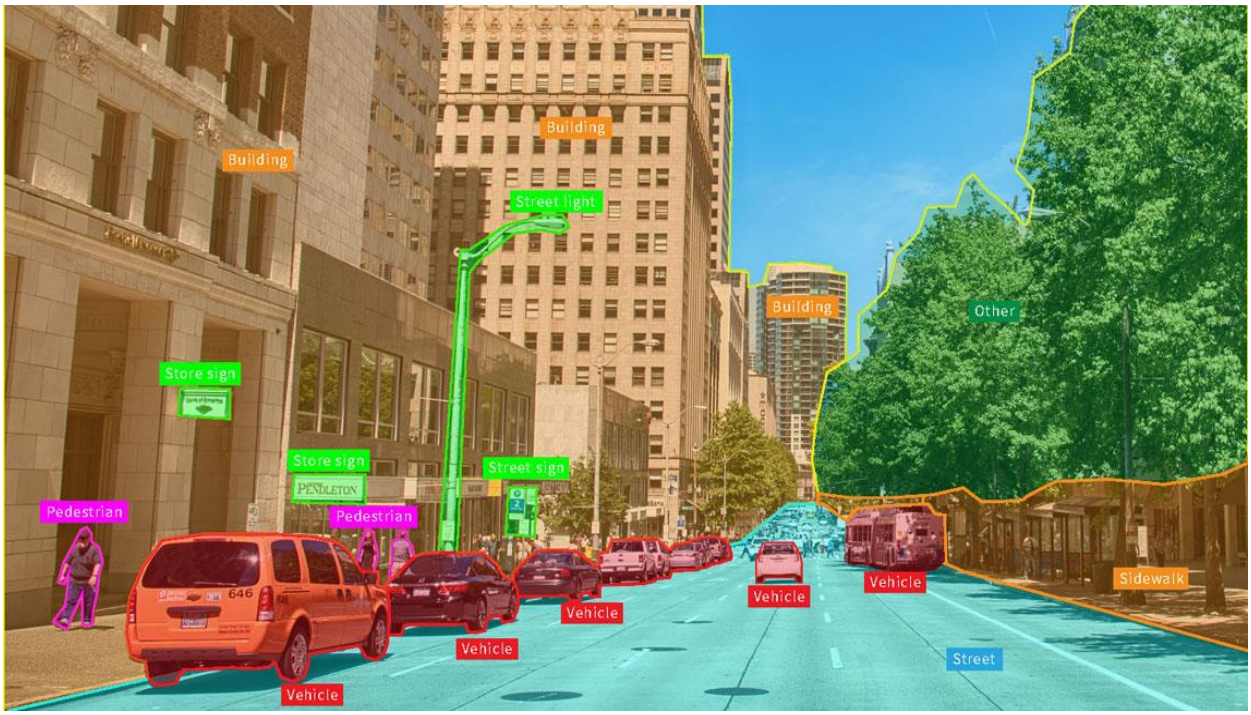

Figura 2-27. Reconocimiento de objetos mediante segmentación semántica[12] [85]

Es importante distinguir la diferencia entre la detección de objetos y el reconocimiento de objetos, ya que son técnicas similares para identificar objetos, pero varían en cuanto a su ejecución. En el aprendizaje profundo, el reconocimiento de objetos es el proceso de identificar objetos presentes en imágenes, mientras que la detección de objetos es un proceso que no solo identifica el objeto, sino que también lo localiza en una imagen.

[12] La segmentación semántica es un algoritmo de aprendizaje profundo que asocia una etiqueta o categoría a cada píxel presente en una imagen. Se utiliza para reconocer un conjunto de píxeles que conforman distintas categorías y se emplea en numerosas aplicaciones, como la conducción autónoma, la generación de imágenes médicas y la inspección industrial [133].

Esto permite identificar y localizar varios objetos en la misma imagen [84].

2.5.5. Arquitecturas de redes convolucionales

Una de las primeras redes neuronales convolucionales fue LeNet-5, presentada en 1998 por Yann LeCun, Léon Bottou, Yoshua Bengio y Patrick Haffner [86]. Consiste en un perceptrón multicapa capaz de reconocer dígitos manuscritos y que podía aprender a través del mecanismo de retropropagación. Todavía hoy es considerado como el modelo estándar más simple de una RNC y es utilizado para introducir este modelo a los estudiantes de aprendizaje profundo [82].

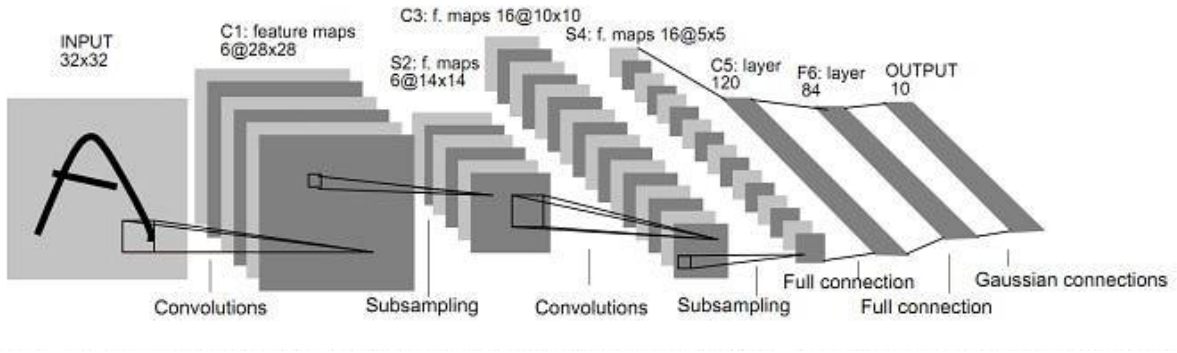

Figura 2-28. Arquitectura de LeNet-5 [86]

Desde entonces, han surgido diferentes arquitecturas de RNC, entre las que se incluyen tres que se han usado en este trabajo para comparar cuál ofrece mejores resultados para el reconocimiento de señales de tráfico: ResNet50, Inception v3 y VGG16.

2.5.5.1. ResNet50

Las redes neuronales residuales (*ResNets*, en inglés) son un tipo de arquitectura de red neuronal convolucional profunda que fue introducida por primera vez en diciembre de 2015 por Kaiming He, Xiangyu Zhang, Shaoqing Ren y Jian Sun de Microsoft Research en un artículo titulado *Deep Residual Learning for Image Recognition* [87], [88].

La necesidad de un modelo como el de red residual surgió debido a una serie de dificultades que se presentaban en las redes neuronales cada vez más profundas, principalmente las tres siguientes [89]:

- Dificultad para entrenar redes neuronales profundas. Conforme el número de capas en un modelo aumenta, el número de parámetros en el modelo aumenta exponencialmente. Un aumento en el número de capas en aras de la experimentación conduce a un aumento igual en la complejidad para entrenar el modelo. Entonces, el entrenamiento requiere mayor poder computacional y memoria [89].
- La paradoja de agregar más capas. A menudo, se considera que una red neuronal es un aproximador de funciones. Tiene la capacidad de modelar funciones dadas la entrada, el objetivo y una comparación entre la salida de la función y el objetivo. Agregar múltiples capas a una red neuronal hace que sea capaz de modelar funciones más complejas, pero los resultados publicados en el artículo indicaron que una red neuronal simple de 18 capas funciona considerablemente mejor que una red neuronal simple de 34 capas, mientras que para una red residual sucedía justo lo contrario, como se puede ver en la Figura 2-29 [89].
- Gradientes que desaparecen o explotan. Este es uno de los problemas más comunes que afectan el entrenamiento de las redes neuronales más profundas.

  Durante la retropropagación, conforme se avanza desde las capas profundas hacia las superficiales, la regla de la cadena de diferenciación hace que los gradientes se multipliquen. A menudo, estos gradientes son muy pequeños, del orden de 10^{-5} o menos. Conforme estos números tan pequeños se van multiplicando entre sí, se vuelven infinitamente más pequeños, haciendo que los cambios sean casi insignificantes en los pesos [89].

  En el otro extremo, hay casos en los que el gradiente alcanza órdenes de hasta 10^4 o más. A medida que estos gradientes tan grandes se multiplican entre sí, los valores tienden a moverse hacia el infinito.

Permitir que un rango tan grande de valores esté en el dominio numérico de los pesos hace que sea difícil lograr la convergencia [89].

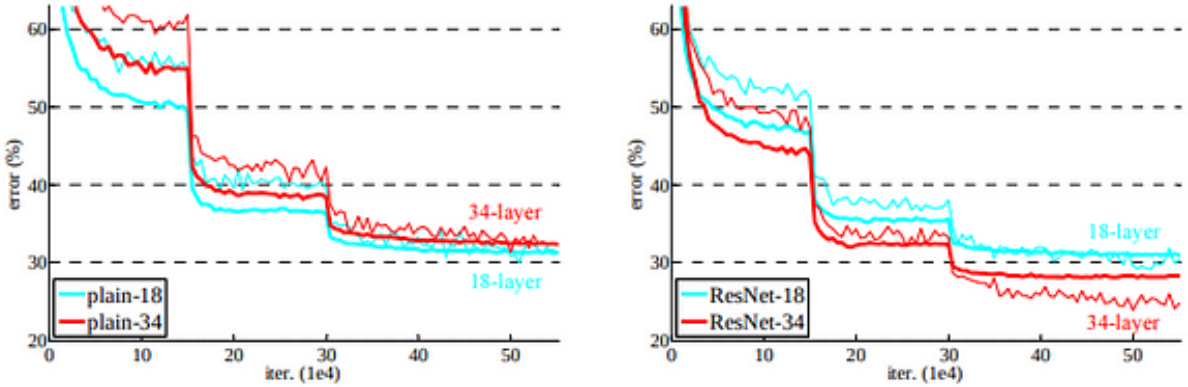

Figura 2-29. Error de entrenamiento entre redes simples (izquierda) y redes residuales (derecha) [89]

Una red residual, debido a su arquitectura, evita que estos problemas ocurran. Concretamente, las conexiones de salto son la característica clave para que estos problemas no sucedan [90].

Las conexiones de salto, o conexiones residuales, funcionan de dos maneras [91]:

- Por un lado, alivian el problema del gradiente que desaparece o que explota al configurar un atajo alternativo para que el gradiente pase.
- Por otro lado, permiten que el modelo aprenda una función de identidad. Esto garantiza que las capas superiores del modelo no funcionen peor que las capas inferiores.

En resumen, los bloques residuales facilitan considerablemente que las capas aprendan funciones de identidad. Como resultado, mejora la eficiencia de las redes neuronales profundas con más capas y minimiza el porcentaje de errores. En otras palabras, las conexiones de salto agregan las salidas de las capas anteriores a las salidas de las capas apiladas, lo que permite entrenar redes mucho más profundas [91].

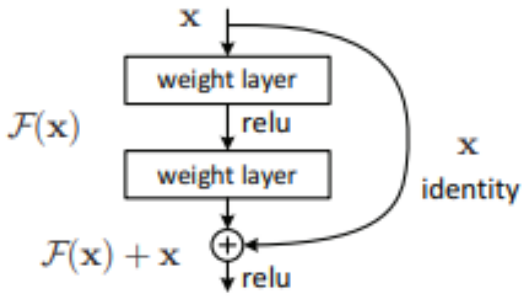

Figura 2-30. Bloque residual [89]

Una de las arquitecturas de red residual más conocidas es ResNet50, que consta de 50 capas y contiene 16 bloques residuales. Cada bloque residual consta de varias capas convolucionales con conexiones residuales. La arquitectura también incluye capas de agrupación, capas completamente conectadas y una capa de salida SoftMax para clasificación [87].

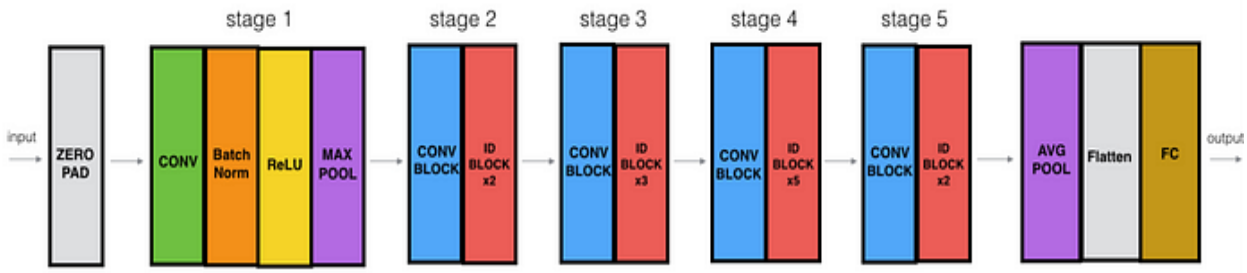

Figura 2-31. Arquitectura de ResNet50 [92]

Como se puede apreciar en la Figura 2-31, la arquitectura de ResNet50 se divide en cuatro partes principales: las capas convolucionales, el bloque identidad, el bloque convolucional y las capas completamente conectadas. Las capas convolucionales son responsables de extraer características de la imagen de entrada, mientras que el bloque identidad y el bloque convolucional son responsables de procesar y transformar estas características. Finalmente, las capas completamente conectadas se utilizan para realizar la clasificación final [90].

Las capas convolucionales en ResNet50 constan de varias capas convolucionales seguidas de una normalización por lotes y una activación ReLU. Estas capas se encargan de extraer características de la imagen de entrada, como bordes, texturas y formas. A las capas convolucionales les siguen capas de agrupación máxima, que reducen las dimensiones espaciales de los mapas de características y, al mismo tiempo, preservan las características más importantes [90].

El bloque identidad y el bloque convolucional son los componentes clave de ResNet50. El bloque identidad pasa la entrada a través de una serie de capas convolucionales y vuelve a agregar la entrada a la salida. Esto permite que la red neuronal aprenda funciones residuales que asignan la entrada a la salida deseada. El bloque convolucional es similar al bloque identidad, pero incorpora una capa convolucional de 1×1 que se utiliza para reducir la cantidad de filtros antes de la capa convolucional de 3×3 [90].

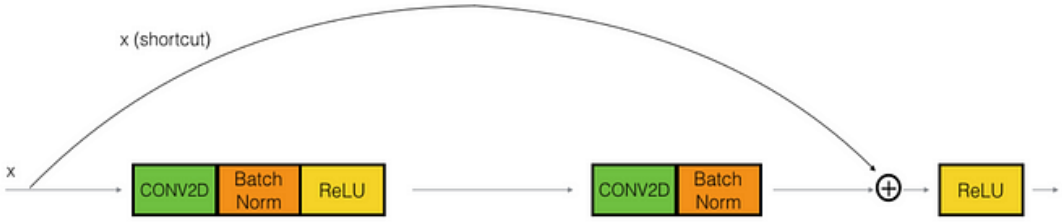

Figura 2-32. Bloque identidad [92]

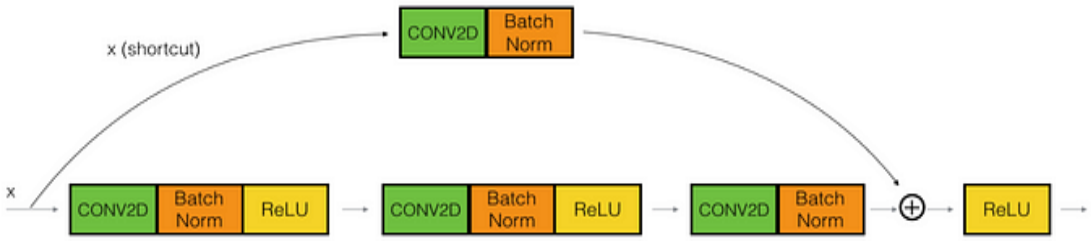

Figura 2-33. Bloque convolucional [92]

La parte final de ResNet50 son las capas completamente conectadas, encargadas de realizar la clasificación final. La salida de la capa final completamente conectada se introduce en una función de activación SoftMax para producir la distribución de probabilidades de cada clase [90].

2.5.5.2. Inception v3

Inception v3 es un modelo de reconocimiento de imágenes que representa la culminación de muchas ideas que desarrollaron varios investigadores durante años. Se basa en el documento original *Rethinking the Inception Architecture for Computer Vision* de Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens y Zbigniew Wojna, publicado en 2015 [93], [94].

El modelo está formado por bloques de compilación simétricos y asimétricos, que incluyen convoluciones, agrupación media, agrupación máxima, concatenaciones, regularización por abandono y capas completamente conectadas. La normalización por lotes se usa ampliamente en todo el modelo y se aplica a las entradas de activación. La pérdida se calcula con una activación SoftMax [94].

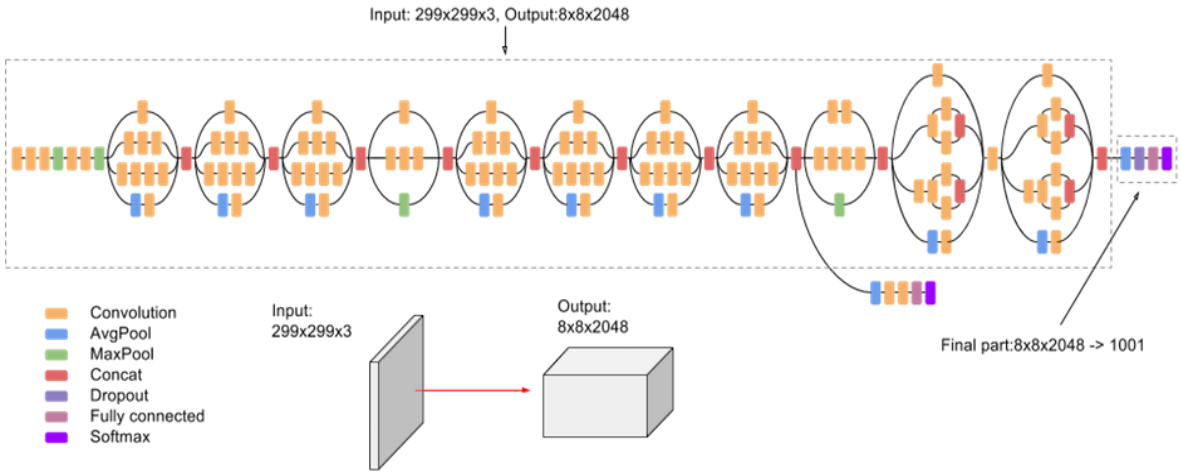

Figura 2-34. Arquitectura de Inception v3 [94]

2.5.5.3. VGG16

VGG16 es un tipo de red neuronal convolucional considerada uno de los mejores modelos de visión artificial. Karen Simonyan y Andrew Zisserman propusieron este modelo y lo publicaron en un artículo titulado *Very Deep Convolutional Networks for Large-Scale Image Recognition* en 2014 [95], [96].

Este modelo se diferenciaba de los modelos anteriores en varios aspectos:

- En primer lugar, usaba un pequeño campo receptivo de 3×3 con un paso de 1 píxel. Los filtros de 3×3 se combinan para proporcionar la función de un campo receptivo más grande. El beneficio de utilizar múltiples capas más pequeñas en lugar de una sola capa grande es que hay más capas de activación no lineales que acompañan las capas convolucionales, lo que mejora las funciones de decisión y permite que la red converja rápidamente [97].
- En segundo lugar, VGG utiliza un filtro convolucional más pequeño, lo que reduce la tendencia de la red al sobreajuste durante el entrenamiento. Un filtro de 3×3 es el tamaño óptimo porque un tamaño menor no puede capturar información de izquierda a derecha y de arriba a abajo. Por tanto, VGG es el modelo más pequeño posible para comprender las características espaciales de una imagen [97].

VGG16, como su nombre indica, es una red neuronal profunda de 16 capas. Por lo tanto, es una red relativamente extensa con un total de 138 millones de parámetros; es enorme incluso para los estándares actuales. Sin embargo, la simplicidad de su arquitectura es su principal atractivo. Consta de pequeños filtros de convolución y tiene 3 capas completamente conectadas y 13 capas convolucionales [97].

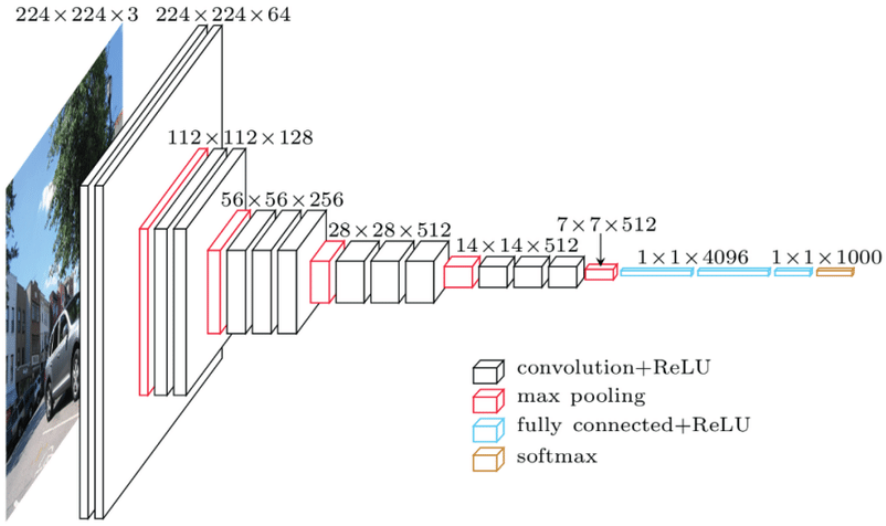

Figura 2-35. Arquitectura de VGG16 [97]

3 PROBLEMA QUE RESOLVER

El problema que se pretende resolver en este trabajo es el reconocimiento de señales de tráfico. Para ello, se probarán las tres arquitecturas de redes neuronales convolucionales explicadas en la subsección 2.5.5, que serán entrenadas con un conjunto de datos de imágenes de 43 tipos de señales de tráfico distintas, y se analizarán y compararán los resultados obtenidos para determinar cuál de las tres arquitecturas es mejor para esta tarea.

3.1. Reconocimiento de señales de tráfico

El reconocimiento de señales de tráfico (RST) es un problema desafiante de gran relevancia industrial. Se trata de un problema de clasificación múltiple con frecuencias de clase desequilibradas. Las señales de tráfico pueden ofrecer una amplia gama de variaciones entre clases en términos de color, forma y presencia de pictogramas o texto. Sin embargo, existen subconjuntos de clases (por ejemplo, señales de límite de velocidad) que son muy similares entre sí. Además, se tiene que hacer frente a grandes variaciones en la apariencia visual debido a cambios de iluminación, oclusiones parciales, rotaciones, condiciones climáticas, etc. [98].

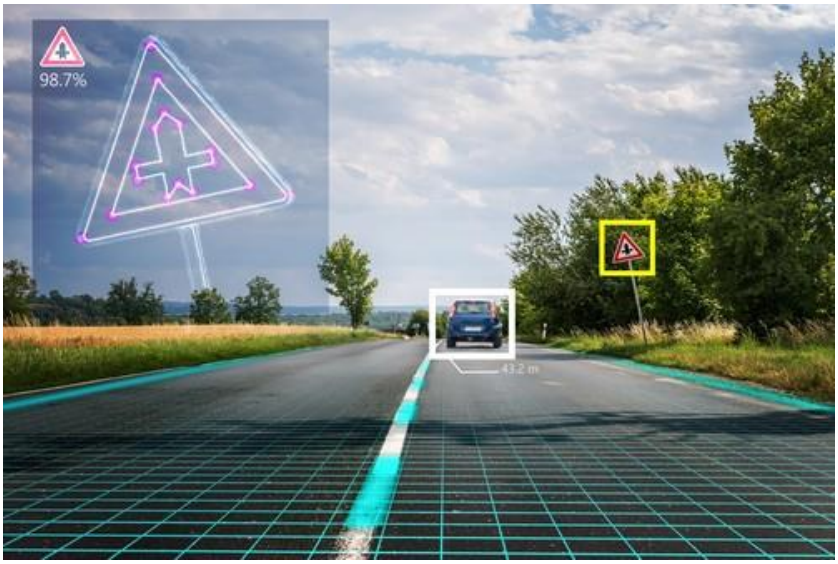

Figura 3-1. Reconocimiento de señales de tráfico [99]

Los seres humanos son capaces de reconocer la gran variedad de señales de tráfico existentes con una precisión cercana al 100%. Esto no solo se aplica a la conducción en el mundo real, que proporciona el contexto y múltiples vistas de una única señal de tráfico, sino también al reconocimiento a partir de imágenes individuales [98].

La tecnología de RST ha empezado a aparecer en los coches más nuevos del mercado y es un refuerzo en la seguridad vial del conductor que aprovecha al máximo posible el potencial de la IA [100]. Sin embargo, aunque ya haya sistemas comerciales en el mercado y se hayan publicado varios estudios sobre este tema, a día de hoy la mayoría de vehículos que cuentan con sistemas de RST solo son capaces de reconocer señales de límite de velocidad, de prohibición de adelantamiento y de fin de prohibiciones [98], [100].

Un coche que tiene un sistema de RST es capaz de detectar algunas indicaciones que se sitúen frente al coche en la carretera. Esta información se traspasa inmediatamente al piloto del conductor, en tiempo real, aumentando así las probabilidades de ver la señal y evitar más accidentes [100].

El sistema de RST funciona mediante una cámara frontal, que detecta la información que transmitir. Esta señal puede verse después en el piloto o en la proyección del parabrisas, dependiendo del modelo de coche. De hecho, hay modelos de coches que van un poco más allá y ayudan en la conducción de forma similar al control de crucero y su limitador de velocidad [100].

Según esto, hay sistemas de RST activos y pasivos [100]:

- En un sistema de RST pasivo, solo se informa al conductor de la existencia de la limitación y se le avisa cuando está infringiéndola para que tome medidas. Este aviso puede incluir sonidos además de indicaciones visuales.
- Un sistema de RST activo, en el caso de que el coche esté equipado con esta capacidad, intervendrá para limitar la velocidad del coche ante señales de límite de velocidad. De esta forma, al conductor le será casi imposible rebasar límites establecidos por la ley de seguridad vial y tendrá que obedecer todas las señales.

El primer sistema de reconocimiento de señales se introdujo en el mercado de la mano de BMW en su modelo serie 7 del año 2008. Esta tecnología ha sido desarrollada por empresas como Ayonix y Continental y, en un primer momento, solo detectaban límites de velocidad. No obstante, con el paso del tiempo y el avance tecnológico, algunos modelos han incluido sensores de lluvia para indicar la velocidad permitida en estas condiciones; otros son capaces de leer indicaciones restrictivas de cualquier índole; y otros, de emitir avisos activos indicando al conductor que debe reducir la velocidad [1].

Sin embargo, la visión artificial generalmente no es tan fiable como la visión humana cuando el contexto y la complejidad del entorno son importantes. Por ejemplo, un sistema de RST podría interpretar erróneamente las pequeñas señales de restricción de velocidad en la parte trasera de los camiones, que indican su velocidad máxima, como el límite de velocidad actual en la carretera [101]. En un artículo publicado en 2021 por Darko Babić, Dario Babić, Mario Fiolić y Željko Šarić, titulado *Analysis of Market-Ready Traffic Sign Recognition Systems in Cars: A Test Field Study*, se intentó determinar si existían diferencias entre la detección y la precisión de la legibilidad de los sistemas de RST que ya existían en el mercado y en qué medida, así como de qué manera los diferentes niveles de cambios gráficos en las señales afectaban su precisión. Para ello, se colocaron señales de límite de velocidad y de prohibición de adelantamiento en un campo de pruebas, y 17 vehículos de 14 marcas diferentes fueron sometidos a pruebas. En general, los resultados mostraron que la detección y legibilidad de señales mediante sistemas de RST diferían entre marcas de automóviles y que incluso pequeños cambios en el diseño de las señales podían afectar drásticamente la exactitud de estos sistemas. Incluso en un ambiente controlado donde no se había alterado ninguna señal, hubo un margen del 5% de señales mal reconocidas [102].

3.2. Datos

Los datos que se van a utilizar provienen de la página web INI Benchmark Website del Institut für Neuroinformatik de la Universidad Ruhr de Bochum (Alemania). Actualmente, hay dos conjuntos de datos disponibles, el *German Traffic Sign Recognition Benchmark* (GTSRB) y el *German Traffic Sign Detection Benchmark* (GTSDB), ambos gratuitos [103]. De estos dos conjuntos de datos, el primero es el que se usará para este trabajo.

Este conjunto de datos está formado por 43 clases de señales de tráfico con 51 839 imágenes en total, dividido en dos archivos comprimidos, uno con datos de entrenamiento (39 209 imágenes) y otro con datos de prueba (12 630 imágenes). Esta división es adecuada, ya que se extraerá el 50% de datos del conjunto de prueba para crear el conjunto de validación, lo que da como resultado un porcentaje de casi el 76% de datos de entrenamiento, un 12% de datos de validación y un 12% de datos de prueba. Cada imagen contiene una sola señal de tráfico, con un borde del 10% alrededor de esta (al menos 5 píxeles) para permitir aproximaciones basadas en bordes. Todas están almacenadas en formato PPM y sus tamaños varían entre 15×15 y 250×250 píxeles, sin ser necesariamente cuadradas [104].

Las anotaciones con la información de cada imagen se proporcionan en archivos CSV, en campos separados por punto y coma. Las anotaciones contienen información sobre el nombre del archivo al que pertenece cada imagen, sus dimensiones, su clase e incluso las coordenadas x e y de los vértices superior izquierdo e inferior derecho que conforman el cuadro delimitador de la señal de tráfico en la imagen [104].

Además, en la carpeta de datos de entrenamiento, hay un código de ejemplo de Python que proporciona una función para iterar sobre el conjunto de datos de entrenamiento para leer las imágenes y la identificación de clase correspondiente [104].

Figura 3-2. Las 43 clases de señales de tráfico presentes en el conjunto de datos [105]

3.3. Herramientas

3.1.1 Python

Python es un lenguaje de programación ampliamente utilizado en las aplicaciones web, el desarrollo de *software*, la ciencia de datos y el aprendizaje automático. Los desarrolladores utilizan Python porque es eficiente y fácil de aprender, además de que se puede ejecutar en muchas plataformas diferentes. El *software* Python se puede descargar gratis, se integra bien en todos los tipos de sistemas y aumenta la velocidad del desarrollo [106].

Figura 3-3. Imagotipo de Python [107]

Estas son las principales características del lenguaje de programación Python [106]:

- Un lenguaje interpretado. Python es un lenguaje interpretado, lo que significa que ejecuta directamente el código línea por línea. Si existen errores en el código del programa, su ejecución se detiene. Así, los programadores pueden encontrar errores en el código rápidamente.
- Un lenguaje fácil de utilizar. Python utiliza palabras similares a las del inglés. A diferencia de otros lenguajes de programación, Python no utiliza llaves, sino sangría.
- Un lenguaje tipeado dinámicamente. Los programadores no tienen que declarar tipos de variables cuando escriben código porque Python los determina en el tiempo de ejecución. Debido a esto, es posible escribir programas de Python con mayor rapidez.
- Un lenguaje de alto nivel. Python es más cercano a un idioma que otros lenguajes de programación. Por lo tanto, los programadores no deben preocuparse sobre sus funcionalidades subyacentes, como la

arquitectura y la administración de la memoria.

- Un lenguaje orientado a objetos. Python considera todo como un objeto, pero también admite otros tipos de programación, como la programación estructurada y la funcional.

Además, Python presenta muchas ventajas, entre las que se incluyen las siguientes [106]:

- Los desarrolladores pueden leer y comprender fácilmente los programas de Python debido a su sintaxis básica similar a la del inglés.
- Python permite que los desarrolladores sean más productivos, ya que pueden escribir un programa con menos líneas de código en comparación con muchos otros lenguajes.
- Python cuenta con una gran biblioteca estándar que contiene códigos reutilizables para casi cualquier tarea. De esta manera, los desarrolladores no tienen que escribir el código desde cero.
- Los desarrolladores pueden utilizar Python fácilmente con otros lenguajes de programación conocidos, como Java, C y C++.
- La comunidad activa de Python incluye millones de desarrolladores alrededor del mundo que prestan su apoyo. Si se presenta un problema, se puede obtener soporte rápido de la comunidad.
- Hay muchos recursos útiles disponibles en Internet para aprender Python. Por ejemplo, se pueden encontrar fácilmente vídeos, tutoriales, documentación y guías para desarrolladores.
- Python se puede trasladar a través de diferentes sistemas operativos de ordenador, como Windows, macOS, Linux y Unix.

3.1.2 TensorFlow y Keras

Para el desarrollo de la inteligencia artificial y el aprendizaje automático, TensorFlow juega un papel muy importante. Se trata de un marco de código abierto para aprendizaje automático que fue desarrollado por Google para satisfacer sus necesidades a partir de redes neuronales artificiales. TensorFlow permite construir y entrenar redes neuronales y, además, es multiplataforma [108].

Figura 3-4. Imagotipo de TensorFlow [109]

En Tensorflow, el flujo de trabajo consiste en recolectar información, crear o escoger un determinado modelo que se acople a los intereses del procesamiento de los macrodatos, ajustar la información almacenada al modelo y que realice la primera predicción, evaluar la exactitud del modelo, continuar experimentando para mejorarlo y, finalmente, guardarlo para seguir nutriendo su efectividad [110].

Algunas de sus características más importantes son [110]:

- TensorFlow utiliza tensores para realizar las operaciones.
- En TensorFlow, primero se definen las operaciones que realizar (se construye el grafo) y, después, se ejecutan (se ejecuta el grafo).
- TensorFlow permite ejecutar el código implementado paralelamente o en una o varias GPU, a elección del usuario.

Entre las ventajas de TensorFlow, cabe destacar las dos siguientes [110]:

- Flexibilidad y escalabilidad. Gracias a la estructura de TensorFlow, es posible implementar cálculos en plataformas como CPU, GPU e incluso TPU, entre otros. Estas plataformas aumentan los campos de acción del sistema TensorFlow. Por ejemplo, con una GPU se pueden crear imágenes que derivan a sistemas de visualización y utilizarlo para el desarrollo de aplicaciones móviles. De igual forma, con una CPU se puede decodificar una amplia cantidad de operaciones para aprender TensorFlow.
- Popularidad. TensorFlow es una plataforma compatible con Python. Este factor es de gran relevancia, ya que Python es un lenguaje de programación muy popular por su sencillez y flexibilidad. Además, ambos sistemas pertenecen a Google Brain, lo que permite adaptabilidad en los diseños.

  Esta dupla de TensorFlow ofrece la legibilidad de código desde el lenguaje de programación y una inferencia propicia de redes neuronales profundas. Por otra parte, debido a la recurrencia de Python y su compatibilidad, hay una gran demanda en el campo laboral.

Keras es la API de alto nivel de la plataforma TensorFlow. Proporciona una interfaz accesible y altamente productiva para resolver problemas de aprendizaje automático, con un enfoque en el aprendizaje profundo moderno. Keras cubre cada paso del flujo de trabajo del aprendizaje automático, desde el procesamiento de datos hasta el ajuste de hiperparámetros y la implementación. Fue desarrollado con un enfoque en permitir una experimentación rápida [111].

Figura 3-5. Imagotipo de Keras [112]

Keras está diseñado para reducir la carga cognitiva, logrando los siguientes objetivos [111]:

- Ofrecer interfaces simples y consistentes.
- Minimizar el número de acciones necesarias para casos de uso comunes.
- Proporcionar mensajes de error claros y procesables.
- Seguir el principio de divulgación progresiva de la complejidad: es fácil comenzar y se pueden completar flujos de trabajo avanzados aprendiendo sobre la marcha.
- Ayudar a escribir código conciso y legible.

Las estructuras de datos centrales de Keras son capas y modelos. Una capa es una simple transformación de entrada/salida y un modelo es un grafo acíclico dirigido de capas [111].

3.1.3 Jupyter Notebook

Jupyter Notebook es una aplicación web interactiva de código abierto usada, generalmente, en la ciencia de datos y el aprendizaje automático. A través de ella, se pueden crear y compartir diferentes documentos que contengan código en vivo, ecuaciones, visualizaciones y texto narrativo [113].

Figura 3-6. Isologo de Jupyter [114]

No obstante, aunque los principales campos de utilización de esta herramienta son los comentados, se puede utilizar para cualquier función que requiera documentación y programación. Esto se debe a que también permite incluir texto, vídeo, audio e imágenes utilizando diferentes lenguajes de programación [113].

Jupyter Notebook se organiza en cuadernos y estos, en celdas que contienen código en diferentes lenguajes de programación, tales como R, Python o Julia, entre otros. Estos cuadernos, en su totalidad, son una herramienta de gran valor de cara al análisis de datos, la exploración de datos y la comunicación científica [113].

3.1.4 Google Colab

Google Colab es una herramienta que ha sido desarrollada por Google para otorgar acceso gratuito a GPU y TPU a cualquier persona en cualquier lugar. Se trata de una herramienta diseñada específicamente para el desarrollo de aplicaciones de IA y análisis de datos. Se puede considerar una versión avanzada de Jupyter Notebook [115].

Figura 3-7. Isotipo de Google Colab [116]

Así, Google Colab va un paso más allá y se trata, en esencia, de un IDE que proporciona muchas características interesantes. Algunas de las más importantes son [115]:

- Ejecutar comandos de terminal desde el portátil.
- Importar conjuntos de datos de fuentes externas como Kaggle.
- Guardar los cuadernos editados con Google Colab en Google Drive y poder importarlos desde ahí.
- Servicio gratuito en la nube para el uso de GPU y TPU.
- Integración con PyTorch, TensorFlow y OpenCV para el desarrollo de análisis de datos e IA.
- Importación o publicación directa en o desde GitHub.

4 DESARROLLO

En este capítulo, se van a detallar los pasos realizados para el desarrollo de la parte de programación del trabajo hasta la fase de entrenamiento, ya que la parte de análisis y comparación de resultados de cada una de las pruebas será tratada en el siguiente capítulo.

Aunque la programación inicialmente se realizó en Jupyter Notebook usando Anaconda, se prefirió finalmente entrenar la red neuronal en Google Colab, dado que llevaba muchísimo menos tiempo al tener acceso gratuito (aunque limitado en tiempo) a una GPU.

Se han llevado a cabo pruebas con las tres arquitecturas mencionadas en la subsección 2.5.5, variando en cada una de ellas dos aspectos: el número de ejemplos (muestras) por minilote (64 o 128) y añadiendo o no una capa de regularización por abandono.

4.1. Importar bibliotecas

En primer lugar, se importan todas las bibliotecas necesarias para el desarrollo del proyecto. Se usarán las bibliotecas *matplotlib* (para el tratamiento de las imágenes), *csv* (para el tratamiento de los archivos CSV con las anotaciones sobre cada imagen), *tensorflow*, *numpy*, *random*, *pandas* (para el tratamiento de los resultados y su posterior representación en gráficas), *sklearn* (para poder crear la matriz de confusión) y *seaborn* (para poder visualizar la matriz de confusión).

```
import matplotlib.pyplot as plt
import csv
import tensorflow as tf
import tensorflow.keras.layers as tfl
import tensorflow.keras.metrics as tfm
import numpy as np
import random
import pandas as pd
from tensorflow.keras.initializers import random_uniform, glorot_uniform, constant, identity
from sklearn.metrics import confusion_matrix
import seaborn as sns
```

Figura 4-1. Importación de bibliotecas

4.2. Preprocesar datos

Una vez importadas las bibliotecas, se pasa a cargar los datos.

Primero, se crean unas funciones que permitan leer las imágenes de las señales de tráfico a partir del directorio donde se encuentran. La función para leer los datos de entrenamiento está tomada de un archivo que contenía el código en Python para realizar dicha lectura y que fue descargado de la misma página web que los conjuntos de datos, mientras que la función para leer los datos de prueba está basada en la anterior:

```python
def leerSenalesEntren(ruta):
    imagenes = []
    etiquetas = []

    for c in range(0,43):
        prefijo = ruta + '/' + format(c, '05d') + '/'
        gtFile = open(prefijo + 'GT-'+ format(c, '05d') + '.csv')
        gtReader = csv.reader(gtFile, delimiter=';')
        next(gtReader)
        for fila in gtReader:
            imagenes.append(plt.imread(prefijo + fila[0]))
            etiquetas.append(np.uint8(fila[7]))
        gtFile.close()
    return imagenes, etiquetas
```

```python
def leerSenalesPrueba(ruta):
    imagenes = []
    etiquetas = []

    prefijo = ruta + '/'
    gtFile = open(prefijo + 'GT-final_test.csv')
    gtReader = csv.reader(gtFile, delimiter=';')
    next(gtReader)
    for fila in gtReader:
        imagenes.append(plt.imread(prefijo + fila[0]))
        etiquetas.append(np.uint8(fila[7]))
    gtFile.close()
    return imagenes, etiquetas
```

Figura 4-2. Funciones para leer los datos de entrenamiento y de prueba

También se asigna el nombre a cada señal de tráfico, ya que si no se identificarían mediante un número. La manera de hacerlo es creando una lista con los nombres de cada señal de tráfico, donde cada nombre ocupa la posición cuyo número es igual al identificador de clase de cada señal.

```python
nombresClases = ['Velocidad máxima (20 km/h)', 'Velocidad máxima (30 km/h)', 'Velocidad máxima (50 km/h)',
                 'Velocidad máxima (60 km/h)', 'Velocidad máxima (70 km/h)', 'Velocidad máxima (80 km/h)',
                 'Fin de la limitación de velocidad (80 km/h)', 'Velocidad máxima (100 km/h)', 'Velocidad máxima (120 km/h)',
                 'Adelantamiento prohibido', 'Adelantamiento prohibido para camiones', 'Intersección con prioridad',
                 'Calzada con prioridad', 'Ceda el paso', 'Detención obligatoria', 'Circulación prohibida',
                 'Entrada prohibida a vehículos destinados al transporte de mercancías', 'Entrada prohibida', 'Otros peligros',
                 'Curva peligrosa hacia la izquierda', 'Curva peligrosa hacia la derecha',
                 'Curvas peligrosas hacia la izquierda', 'Perfil irregular', 'Pavimento deslizante',
                 'Estrechamiento de calzada por la derecha', 'Obras', 'Semáforos', 'Peatones', 'Niños', 'Ciclistas',
                 'Pavimento deslizante por hielo o nieve', 'Paso de animales en libertad', 'Fin de prohibiciones',
                 'Sentido obligatorio (giro a la derecha)', 'Sentido obligatorio (giro a la izquierda)',
                 'Sentido obligatorio (hacia adelante)',
                 'Únicas direcciones y sentidos permitidos (hacia adelante y hacia la derecha)',
                 'Únicas direcciones y sentidos permitidos (hacia adelante y hacia la izquierda)',
                 'Paso obligatorio (hacia la derecha)', 'Paso obligatorio (hacia la izquierda)',
                 'Intersección de sentido giratorio obligatorio', 'Fin de la prohibición de adelantamiento',
                 'Fin de la prohibición de adelantamiento para camiones']
```

Figura 4-3. Lista de nombres de las señales de tráfico

A continuación, se pasa a cargar los datos de entrenamiento y prueba usando las funciones explicadas anteriormente:

```
imagenesEntren, etiquetasEntren = leerSenalesEntren('GTSRB_Final_Training_Images/GTSRB/Final_Training/Images')
n_entren = len(imagenesEntren)
print(n_entren)
```

```
39209
```

```
imagenesPrueba, etiquetasPrueba = leerSenalesPrueba('GTSRB_Final_Test_Images/GTSRB/Final_Test/Images')
n_prueba = len(imagenesPrueba)
print(n_prueba)
```

```
12630
```

Figura 4-4. Carga de los datos de entrenamiento y de prueba

Una vez cargadas las imágenes de los datos de entrenamiento y de prueba, se visualiza una muestra aleatoria de cada uno de los dos, para comprobar que los datos han sido cargados correctamente y que la asignación de nombre es correcta. Se puede apreciar que ambas imágenes tienen un alto nivel de pixelado debido a sus pequeñas dimensiones.

```
i = random.randint(0, n_entren)
plt.imshow(imagenesEntren[i])
plt.title(nombresClases[etiquetasEntren[i]])
print(imagenesEntren[i].shape)
```

```
(47, 46, 3)
```

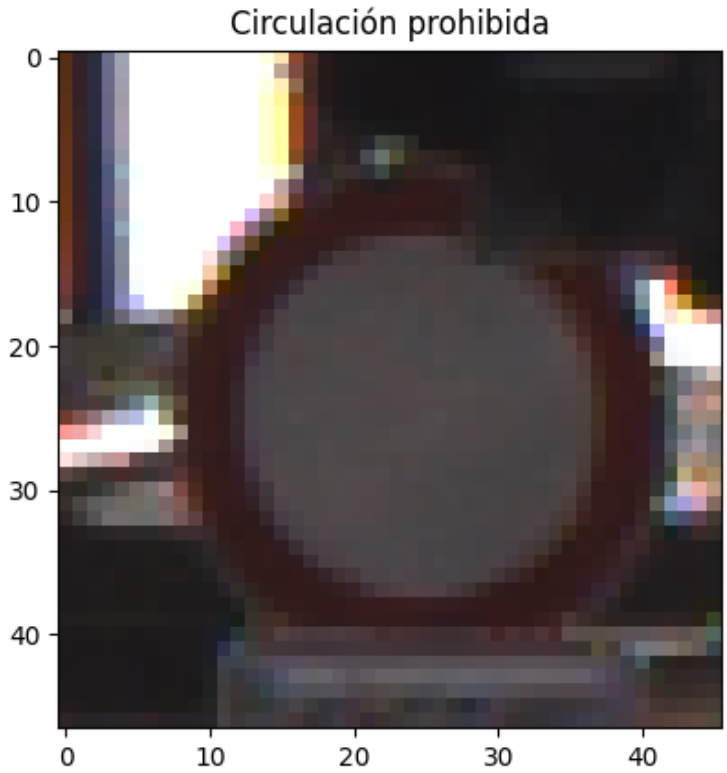

Figura 4-5. Imagen aleatoria del conjunto de datos de entrenamiento

```
i = random.randint(0, n_prueba)
plt.imshow(imagenesPrueba[i])
plt.title(nombresClases[etiquetasPrueba[i]])
print(imagenesPrueba[i].shape)
```

```
(49, 50, 3)
```

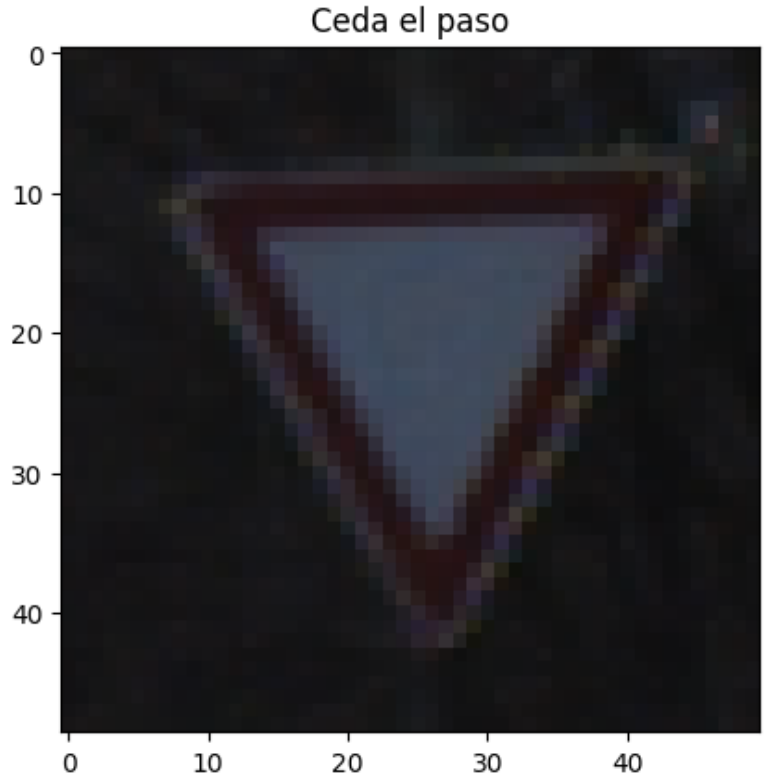

Figura 4-6. Imagen aleatoria del conjunto de datos de prueba

Tras esto, se barajan los datos de entrenamiento porque en el archivo se presentan ordenados en carpetas numeradas en función de la clase a la que pertenecen.

```
orden = list(range(n_entren))
random.shuffle(orden)
```

```
imagenesEntrenb = []
etiquetasEntrenb = []
for k in range(n_entren):
    imagenesEntrenb.append(imagenesEntren[orden[k]])
    etiquetasEntrenb.append(etiquetasEntren[orden[k]])
```

Figura 4-7. Barajadura de los datos de entrenamiento

Después, se redimensionan los datos, ya que es necesario que todas las imágenes tengan las mismas dimensiones para que no haya errores en la ejecución del entrenamiento. Como las imágenes varían entre 15×15 y 250×250, se ha optado por redimensionarlas a 64×64. El principal inconveniente es que aquellas que aumenten su tamaño se verán con un mayor nivel de pixelado, lo que puede suponer una dificultad adicional para el modelo a la hora de reconocer las señales de tráfico.

```
imagenesEntren1 = []
altura = 64
anchura = altura
for k in range(n_entren):
    imagenesEntren1.append(tfl.Resizing(altura, anchura)(imagenesEntrenb[k]))
```

```
imagenesPrueba1 = []
for k in range(n_prueba):
    imagenesPrueba1.append(tfl.Resizing(altura, anchura)(imagenesPrueba[k]))
```

Figura 4-8. Redimensionamiento de las imágenes de los datos

También se ha decidido dividir el conjunto de pruebas en conjunto de validación y conjunto de pruebas en una relación de 50% cada uno, para mantener de este modo una relación porcentual constante entre los diferentes conjuntos de datos.

```
imagenesValid = []
etiquetasValid = []
imagenesPrueba2 = []
etiquetasPrueba2 = []

for i in range(n_prueba):
    if i >= int(0.5*n_prueba):
        imagenesPrueba2.append(imagenesPrueba1[i])
        etiquetasPrueba2.append(etiquetasPrueba[i])
    else:
        imagenesValid.append(imagenesPrueba1[i])
        etiquetasValid.append(etiquetasPrueba[i])

n_valid = len(imagenesValid)
n_prueba = len(imagenesPrueba2)
print(n_entren, n_valid, n_prueba)
```

```
39209 6315 6315
```

Figura 4-9. División del conjunto de datos de prueba

A continuación, para comprobar que todo es correcto, se visualizan 9 imágenes aleatorias de cada uno de los tres conjuntos de datos:

```
rango_entren = np.random.randint(0, n_entren, size = 9)
plt.figure(figsize = (14, 14))
for i in range(9):
        ax = plt.subplot(3, 3, i + 1)
        plt.imshow(imagenesEntren1[rango_entren[i]])
        plt.title(nombresClases[etiquetasEntrenb[rango_entren[i]]], fontsize = 9)
        plt.axis('off')
```

Figura 4-10. Visualización de nueve imágenes aleatorias del conjunto de entrenamiento

Figura 4-11. Imágenes aleatorias del conjunto de entrenamiento

```
rango_valid = np.random.randint(0, n_valid, size = 9)
plt.figure(figsize = (14, 14))
for i in range(9):
        ax = plt.subplot(3, 3, i + 1)
        plt.imshow(imagenesValid[rango_valid[i]])
        plt.title(nombresClases[etiquetasValid[rango_valid[i]]], fontsize = 9)
        plt.axis('off')
```

Figura 4-12. Visualización de nueve imágenes aleatorias del conjunto de validación

Figura 4-13. Imágenes aleatorias del conjunto de validación

```
rango_prueba = np.random.randint(0, n_prueba, size = 9)
plt.figure(figsize = (14, 14))
for i in range(9):
        ax = plt.subplot(3, 3, i + 1)
        plt.imshow(imagenesPrueba2[rango_prueba[i]])
        plt.title(nombresClases[etiquetasPrueba2[rango_prueba[i]]], fontsize = 9)
        plt.axis('off')
```

Figura 4-14. Visualización de nueve imágenes aleatorias del conjunto de prueba

Figura 4-15. Imágenes aleatorias del conjunto de prueba

Luego, se convierten las etiquetas de cada uno de los datos mediante codificación en caliente, ya que hasta ahora las etiquetas han sido representadas por un número entero. De este modo, se puede asegurar que todas las categorías tendrán igual valor y los errores de una clasificación errónea dentro de una categoría que no corresponda tendrán el mismo valor de pérdida [117].

```
clases = 43
etiquetasEntrenOH = tf.one_hot(etiquetasEntrenb, clases)
etiquetasValidOH = tf.one_hot(etiquetasValid, clases)
etiquetasPruebaOH = tf.one_hot(etiquetasPrueba2, clases)
```

Figura 4-16. Codificación en caliente de las etiquetas de los datos

Por último, se crean los conjuntos de datos por minilotes. Aunque en la imagen aparezcan los conjuntos de datos por minilotes de 64 ejemplos (muestras), también se han probado para 128. Esto se ha realizado para evaluar si una estrategia menos demandante de recursos de cálculo (es decir, menos lotes) afecta los resultados y, por consiguiente, el rendimiento del modelo.

```
datosEntren = tf.data.Dataset.from_tensor_slices((imagenesEntren1, etiquetasEntrenOH)).batch(64)
datosValid = tf.data.Dataset.from_tensor_slices((imagenesValid, etiquetasValidOH)).batch(64)
datosPrueba = tf.data.Dataset.from_tensor_slices((imagenesPrueba2, etiquetasPruebaOH)).batch(64)
```

Figura 4-17. Creación de los conjuntos de datos por minilotes

4.3. Crear el modelo

Una vez que se haya completado la fase de preprocesamiento de datos para el entrenamiento, se pasa a crear el modelo. Como ya se ha comentado, se probarán tres arquitecturas de red: ResNet50, Inception v3 y VGG16. Además, en cada una de ellas, se harán pruebas con y sin regularización por abandono. El valor de regularización por abandono que se probará es de 0,2, lo que significa que el 20% de las neuronas de cada capa oculta se omite en cada paso durante el entrenamiento [118].

El código de la arquitectura de ResNet50 se ha tomado de [119], el de la arquitectura de Inception v3, de [120], y el de la arquitectura de VGG16, de [96].

A continuación, se muestra el código de la arquitectura de ResNet50 sin añadirle regularización. Los códigos de las otras dos arquitecturas se muestran en el anexo A.

```
def identity_block(X, f, filters, training=True, initializer=random_uniform):

    F1, F2, F3 = filters

    X_shortcut = X

    X = tfl.Conv2D(filters = F1, kernel_size = 1, strides = (1,1), padding = 'valid', kernel_initializer = initializer(seed=0))(X)
    X = tfl.BatchNormalization(axis = 3)(X, training = training) # Default axis
    X = tfl.Activation('relu')(X)

    X = tfl.Conv2D(filters = F2, kernel_size = f, strides = (1,1), padding = 'same', kernel_initializer = initializer(seed=0))(X)
    X = tfl.BatchNormalization(axis = 3)(X, training = training)
    X = tfl.Activation('relu')(X)

    X = tfl.Conv2D(filters = F3, kernel_size = 1, strides = (1,1), padding = 'valid', kernel_initializer = initializer(seed=0))(X)
    X = tfl.BatchNormalization(axis = 3)(X, training = training)

    X = tfl.Add()([X, X_shortcut])
    X = tfl.Activation('relu')(X)

    return X
```

Figura 4-18. Bloque identidad de ResNet50

```
def convolutional_block(X, f, filters, s = 2, training=True, initializer=glorot_uniform):

    F1, F2, F3 = filters

    X_shortcut = X

    X = tfl.Conv2D(filters = F1, kernel_size = 1, strides = (s, s), padding='valid', kernel_initializer = initializer(seed=0))(X)
    X = tfl.BatchNormalization(axis = 3)(X, training=training)
    X = tfl.Activation('relu')(X)

    X = tfl.Conv2D(filters = F2, kernel_size = f, strides = (1, 1), padding='same', kernel_initializer = initializer(seed=0))(X)
    X = tfl.BatchNormalization(axis = 3)(X, training=training)
    X = tfl.Activation('relu')(X)

    X = tfl.Conv2D(filters = F3, kernel_size = 1, strides = (1, 1), padding='valid', kernel_initializer = initializer(seed=0))(X)
    X = tfl.BatchNormalization(axis = 3)(X, training=training)

    X_shortcut = tfl.Conv2D(filters = F3, kernel_size = 1, strides = (s, s), padding='valid', kernel_initializer = initializer(seed=0))(X_shortcut)
    X_shortcut = tfl.BatchNormalization(axis = 3)(X_shortcut, training=training)

    X = tfl.Add()([X, X_shortcut])
    X = tfl.Activation('relu')(X)

    return X
```

Figura 4-19. Bloque convolucional de ResNet50

```
def ResNet50(input_shape, classes):

    X_input = tfl.Input(input_shape)

    X = tfl.ZeroPadding2D((3, 3))(X_input)

    X = tfl.Conv2D(64, (7, 7), strides = (2, 2), kernel_initializer = glorot_uniform(seed=0))(X)
    X = tfl.BatchNormalization(axis = 3)(X)
    X = tfl.Activation('relu')(X)
    X = tfl.MaxPooling2D((3, 3), strides=(2, 2))(X)

    X = convolutional_block(X, f = 3, filters = [64, 64, 256], s = 1)
    X = identity_block(X, 3, [64, 64, 256])
    X = identity_block(X, 3, [64, 64, 256])

    X = convolutional_block(X, f = 3, filters = [128, 128, 512], s = 2)

    X = identity_block(X, 3, [128, 128, 512])
    X = identity_block(X, 3, [128, 128, 512])
    X = identity_block(X, 3, [128, 128, 512])

    X = convolutional_block(X, f = 3, filters = [256, 256, 1024], s = 2)

    X = identity_block(X, 3, [256, 256, 1024])
    X = identity_block(X, 3, [256, 256, 1024])
    X = identity_block(X, 3, [256, 256, 1024])
    X = identity_block(X, 3, [256, 256, 1024])
    X = identity_block(X, 3, [256, 256, 1024])

    X = convolutional_block(X, f = 3, filters = [512, 512, 2048], s = 2)

    X = identity_block(X, 3, [512, 512, 2048])
    X = identity_block(X, 3, [512, 512, 2048])

    X = tfl.AveragePooling2D((2, 2))(X)
    X = tfl.Dropout(0.1)(X)

    X = tfl.Flatten()(X)
    X = tfl.Dense(classes, activation='softmax', kernel_initializer = glorot_uniform(seed=0))(X)

    model = tf.keras.Model(inputs = X_input, outputs = X)

    return model
```

Figura 4-20. Arquitectura de ResNet50 sin regularización

Por último, se configura y compila el modelo para su entrenamiento. Se usará la siguiente configuración:

- El algoritmo de optimización usado será Adam porque, de todos los que se han expuesto en la Tabla 2-3 de la subsección 2.3.6, es el más eficiente y robusto.
- La función de pérdida empleada es la entropía cruzada categórica, ya que nuestro caso es de reconocimiento de imágenes mediante clasificación multiclase y las etiquetas están codificadas en caliente, de acuerdo a lo expuesto en la Tabla 2-1 de la subsección 2.3.1.
- La métrica escogida es la exactitud porque se busca comprobar cuán bien es capaz el modelo de reconocer las diferentes clases.

```
model = ResNet50(input_shape = (altura, anchura, 3), classes = clases)
print(model.summary())
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
```

Figura 4-21. Compilación del modelo

Debido a la extensión del resultado generado por el fragmento de código anterior, su salida se muestra en el anexo B para no entorpecer la lectura.

4.4. Entrenar el modelo

El entrenamiento de un modelo puede llevar tiempo, así que se recurrirá a las devoluciones de llamada con las que TensorFlow cuenta para detener el entrenamiento cuando se cumplan ciertas condiciones.

Por un lado, se programará una parada temprana, de manera que el entrenamiento termine si la exactitud en la validación no mejora al cabo de 5 épocas[13] consecutivas. Por otro lado, se programará un punto de control del modelo para que, con cada época, vaya almacenando los valores de los pesos obtenidos en la misma si son mejores que los obtenidos en la época anterior, comparándolos en base a una métrica indicada (en este caso, en base a la exactitud en la validación).

El modelo será entrenado durante 10 épocas.

```
RutaPDC = '/content/checkpoint.weights.h5'
ParadaTemprana = tf.keras.callbacks.EarlyStopping(monitor = 'val_accuracy', patience = 5, restore_best_weights = False)
PuntoDeControl = tf.keras.callbacks.ModelCheckpoint(filepath = RutaPDC, save_weights_only = True, monitor = 'val_accuracy', mode = 'max',
```

```
history = model.fit(datosEntren, epochs = 10, validation_data = datosValid, callbacks = [ParadaTemprana, PuntoDeControl])
```

Figura 4-22. Entrenamiento del modelo

```
Epoch 1/10
613/613 [==============================] - ETA: 0s - loss: 1.4759 - accuracy: 0.6506
Epoch 1: val_accuracy improved from -inf to 0.86999, saving model to /content/checkpoint.weights.h5
613/613 [==============================] - 99s 90ms/step - loss: 1.4759 - accuracy: 0.6506 - val_loss: 0.4526 - val_accuracy: 0.8700
Epoch 2/10
613/613 [==============================] - ETA: 0s - loss: 0.1432 - accuracy: 0.9568
Epoch 2: val_accuracy improved from 0.86999 to 0.92035, saving model to /content/checkpoint.weights.h5
613/613 [==============================] - 57s 94ms/step - loss: 0.1432 - accuracy: 0.9568 - val_loss: 0.3242 - val_accuracy: 0.9203
Epoch 3/10
613/613 [==============================] - ETA: 0s - loss: 0.0661 - accuracy: 0.9805
Epoch 3: val_accuracy improved from 0.92035 to 0.92494, saving model to /content/checkpoint.weights.h5
613/613 [==============================] - 62s 101ms/step - loss: 0.0661 - accuracy: 0.9805 - val_loss: 0.2864 - val_accuracy: 0.9249
Epoch 4/10
613/613 [==============================] - ETA: 0s - loss: 0.0729 - accuracy: 0.9789
Epoch 4: val_accuracy did not improve from 0.92494
613/613 [==============================] - 51s 84ms/step - loss: 0.0729 - accuracy: 0.9789 - val_loss: 0.3956 - val_accuracy: 0.8949
Epoch 5/10
613/613 [==============================] - ETA: 0s - loss: 0.1820 - accuracy: 0.9553
Epoch 5: val_accuracy improved from 0.92494 to 0.94220, saving model to /content/checkpoint.weights.h5
613/613 [==============================] - 57s 93ms/step - loss: 0.1820 - accuracy: 0.9553 - val_loss: 0.2152 - val_accuracy: 0.9422
Epoch 6/10
613/613 [==============================] - ETA: 0s - loss: 0.0262 - accuracy: 0.9927
Epoch 6: val_accuracy improved from 0.94220 to 0.94584, saving model to /content/checkpoint.weights.h5
613/613 [==============================] - 57s 93ms/step - loss: 0.0262 - accuracy: 0.9927 - val_loss: 0.2113 - val_accuracy: 0.9458
Epoch 7/10
613/613 [==============================] - ETA: 0s - loss: 0.0226 - accuracy: 0.9936
Epoch 7: val_accuracy improved from 0.94584 to 0.95218, saving model to /content/checkpoint.weights.h5
613/613 [==============================] - 62s 101ms/step - loss: 0.0226 - accuracy: 0.9936 - val_loss: 0.2033 - val_accuracy: 0.9522
Epoch 8/10
613/613 [==============================] - ETA: 0s - loss: 0.0199 - accuracy: 0.9940
Epoch 8: val_accuracy did not improve from 0.95218
613/613 [==============================] - 51s 84ms/step - loss: 0.0199 - accuracy: 0.9940 - val_loss: 0.2251 - val_accuracy: 0.9508
Epoch 9/10
613/613 [==============================] - ETA: 0s - loss: 0.0233 - accuracy: 0.9937
Epoch 9: val_accuracy did not improve from 0.95218
613/613 [==============================] - 51s 83ms/step - loss: 0.0233 - accuracy: 0.9937 - val_loss: 0.2123 - val_accuracy: 0.9504
Epoch 10/10
613/613 [==============================] - ETA: 0s - loss: 0.0302 - accuracy: 0.9907
Epoch 10: val_accuracy did not improve from 0.95218
613/613 [==============================] - 51s 84ms/step - loss: 0.0302 - accuracy: 0.9907 - val_loss: 0.2648 - val_accuracy: 0.9416
```

Figura 4-23. Resultado del entrenamiento

Después del entrenamiento, se carga el modelo con los mejores pesos según el criterio que se ha explicado antes. En la Figura 4-23, se puede observar que los mejores parámetros se han conseguido en la séptima época.

[13] Una época se define como el número total de iteraciones de todos los datos de entrenamiento en un ciclo para entrenar el modelo, es decir, cuando todos los datos de entrenamiento se utilizan a la vez. También se puede definir una época como la cantidad de pasadas (propagaciones) que realiza un conjunto de datos de entrenamiento en un algoritmo. Se cuenta una pasada cuando el conjunto de datos ha realizado la propagación tanto hacia adelante como hacia atrás [134].

```
model.load_weights(RutaPDC)
```

```
perdValid, exactValid = model.evaluate(datosValid)
print("Pérdida en validación: {}, Exactitud en validación: {}".format(perdValid, exactValid))
```

```
99/99 [==============================] - 3s 26ms/step - loss: 0.2033 - accuracy: 0.9522
Pérdida en validación: 0.20325465500354767, Exactitud en validación: 0.9521773457527161
```

Figura 4-24. Carga del modelo con los mejores pesos

5 PRUEBAS Y RESULTADOS

En este capítulo, se van a analizar y comparar los resultados de cada una de las pruebas que se han realizado para las tres arquitecturas de red neuronal, en función de si al modelo se le ha añadido o no una capa de regularización por abandono, y en función del número de ejemplos que conformaban los minilotes de los conjuntos de datos (64 o 128).

En cada una de las secciones, se mostrarán las figuras solamente para una de las pruebas realizadas.

5.1. Analizar las métricas del modelo

La forma más rápida de analizar la evolución de la función de pérdida y de las métricas del modelo (en este caso, la exactitud) tanto en el entrenamiento como en la validación es mediante la representación gráfica de los resultados.

```
df_loss_acc = pd.DataFrame(history.history)
df_loss= df_loss_acc[['loss','val_loss']]
df_loss.rename(columns={'loss':'Entrenamiento','val_loss':'Validación'},inplace=True)
df_acc= df_loss_acc[['accuracy','val_accuracy']]
df_acc.rename(columns={'accuracy':'Entrenamiento','val_accuracy':'Validación'},inplace=True)
df_loss.plot(title='Pérdida del modelo',figsize=(12,8)).set(xlabel='Época',ylabel='Pérdida')
df_acc.plot(title='Exactitud del modelo',figsize=(12,8)).set(xlabel='Época',ylabel='Exactitud')
```

Figura 5-1. Visualización de la pérdida y de la exactitud del modelo

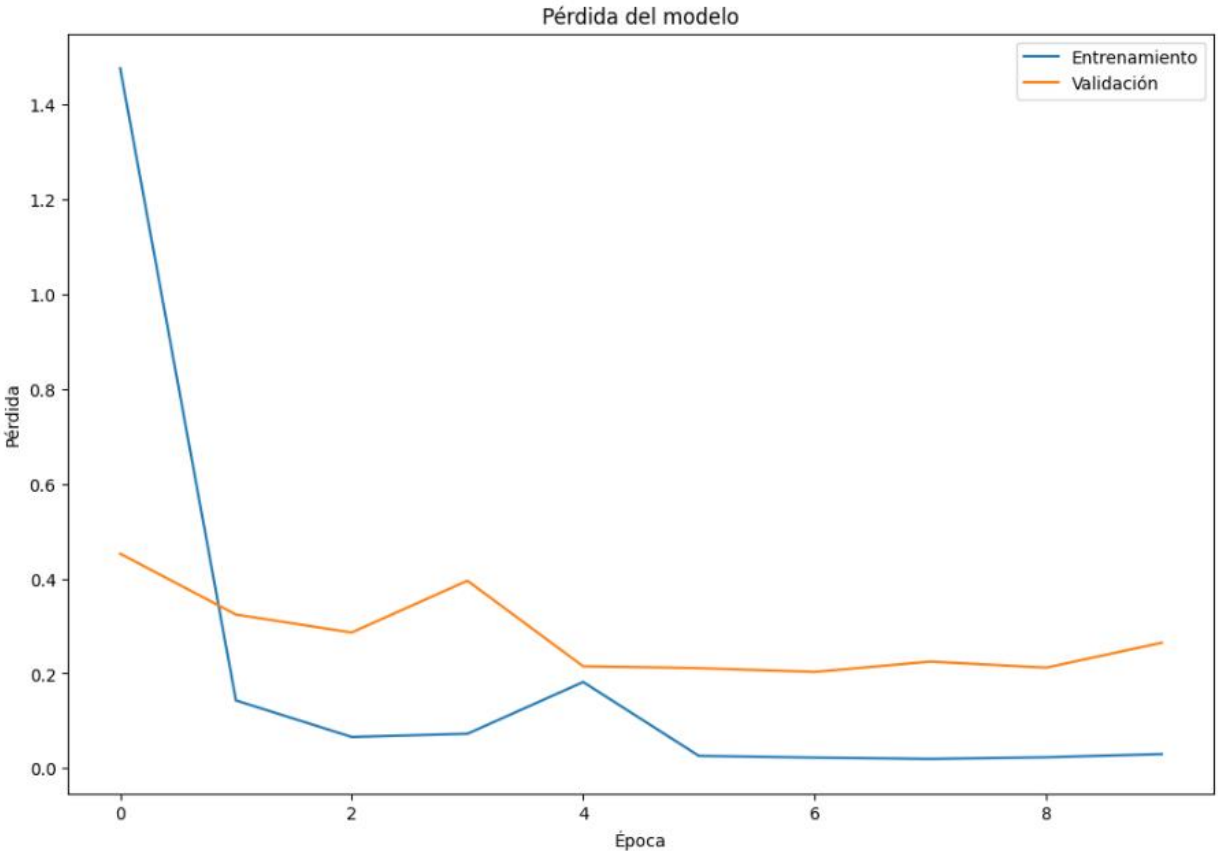

Figura 5-2. Pérdida del modelo

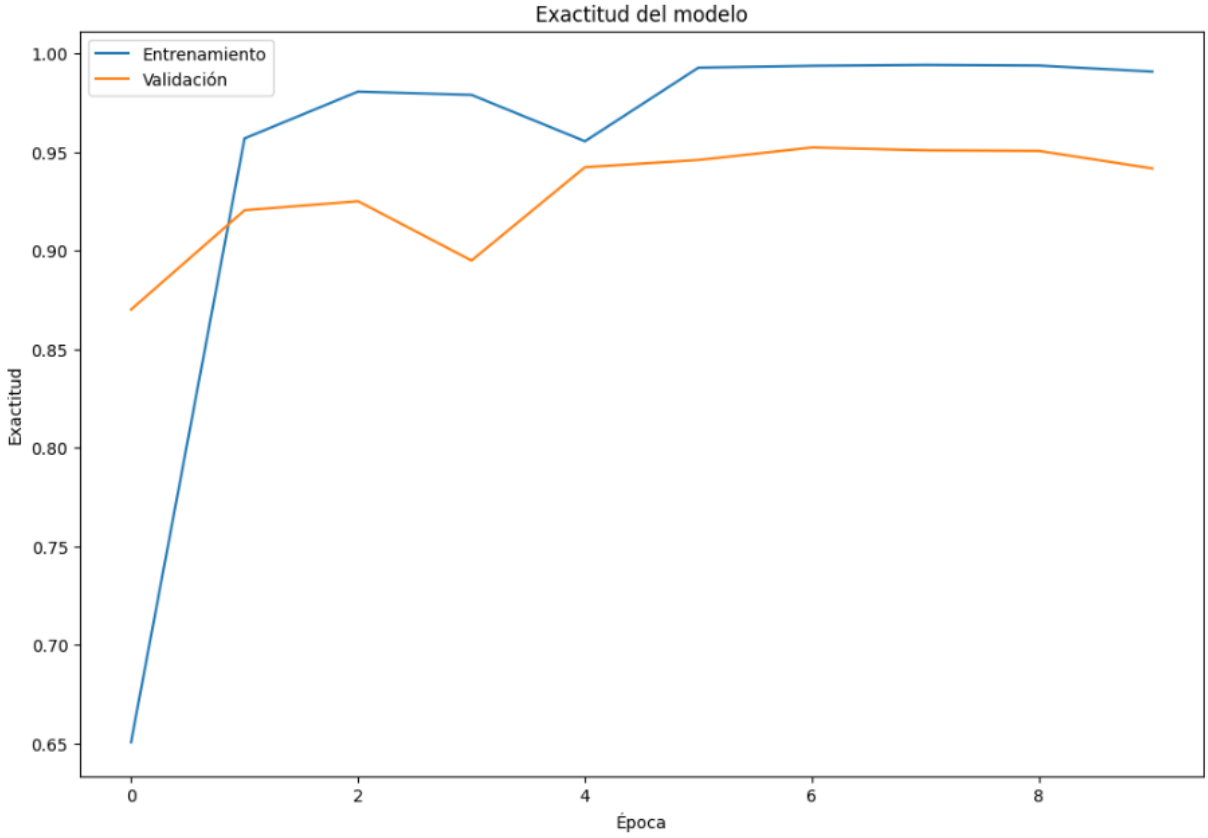

Figura 5-3. Exactitud del modelo

Se puede observar que el modelo presenta un sesgo muy bajo, ya que el error de entrenamiento es muy pequeño; así como una varianza baja, puesto que la diferencia entre los errores de entrenamiento y validación es pequeña.

A continuación, se muestra una tabla con las exactitudes de entrenamiento y validación y del tiempo de ejecución del entrenamiento para cada una de las 3 arquitecturas en función de la regularización y el número de ejemplos por minilote:

Tabla 5-1. Comparación de las exactitudes de entrenamiento y validación y del tiempo de ejecución

Nº de ejemplos por minilote	Regularización por abandono	Inception v3	ResNet50	VGG16
64	No	99,00% / 95,58% 670s 993ms	99,36% / 95,22% 605s 252ms	98,02% / 92,70% 609s 895ms
	Sí	98,34% / 96,82% 752s 372ms	99,21% / 96,06% 611s 968ms	98,24% / 94,62% 632s 409ms
128	No	98,96% / 95,36% 521s 47ms	99,12% / 95,68% 519s 569ms	97,88% / 92,40% 534s 699ms
	Sí	97,57% / 95,91% 537s 547ms	99,50% / 94,06% 548s 679ms	98,38% / 95,33% 525s 694ms

En la Tabla 5-1, aparecen la exactitud de entrenamiento y la exactitud de validación para cada prueba, expuestas en cada celda a la izquierda y a la derecha de la barra, respectivamente. Se pueden extraer las siguientes conclusiones a partir de los resultados:

- Tomando como referencia la exactitud de validación, es mejor usar 64 ejemplos por minilote que 128, aunque haya poca diferencia entre un caso y otro. También se aprecia que el tiempo de entrenamiento es algo menor cuando se usan más ejemplos por minilote.
- Tomando como referencia la exactitud de validación, la regularización mejora un poco el rendimiento del modelo (para la arquitectura VGG16, la mejora es más notable, y para ResNet50, empeora un poco si se usan 128 ejemplos por minilote). La regularización por abandono que se ha usado tiene una tasa de 0,2.
- En cuanto a los tiempos de ejecución, se observa que ResNet50 es la arquitectura que menos tiempo requiere para entrenarse de las tres (aunque las diferencias con las otras dos es de tan solo unos cuantos segundos).
- En general, la arquitectura Inception v3 parece dar mejores resultados de exactitud si se tiene en cuenta la exactitud de validación, mientras que la arquitectura ResNet50 ofrece mejores resultados para la exactitud de entrenamiento. La arquitectura VGG16 produce unos resultados un poco peores que las otras dos. De los 4 casos posibles para cada arquitectura, el mejor para cada una ha sido, en base a la exactitud de validación:
    - Inception v3: 64 ejemplos por minilote y con regularización por abandono.
    - ResNet50: 64 ejemplos por minilote y con regularización por abandono.
    - VGG16: 128 ejemplos por minilote y con regularización por abandono.

Por último, cabe decir que si las imágenes de los datos tuvieran una mejor resolución, los errores de entrenamiento y validación probablemente serían menores.

5.2. Evaluar el modelo y predecir con él

Ahora que el modelo ya está entrenado, se puede comprobar mejor su rendimiento con el conjunto de datos de prueba. Para ello, se evaluará su función de pérdida y su exactitud de prueba:

```
perdPrueba, exactPrueba = model.evaluate(datosPrueba)
print("Pérdida en prueba: {}, Exactitud en prueba: {}".format(perdPrueba, exactPrueba))
```

```
99/99 [==============================] - 3s 26ms/step - loss: 0.1941 - accuracy: 0.9511
Pérdida en prueba: 0.19414524734020233, Exactitud en prueba: 0.9510688781738281
```

Figura 5-4. Evaluación del modelo

A continuación, se muestra una tabla con la exactitud de prueba para cada una de las 3 arquitecturas en función de la regularización y el número de ejemplos por minilote:

Tabla 5-2. Comparación de las exactitudes de prueba

Nº de ejemplos por minilote	Regularización por abandono	Inception v3	ResNet50	VGG16
64	No	95,47%	95,11%	92,13%
	Sí	96,86%	95,96%	94,77%
128	No	95,82%	95,68%	92,34%
	Sí	95,77%	93,84%	95,55%

Puede apreciarse en la Tabla 5-2 que los casos en los que se han conseguido los mejores resultados de exactitud de prueba coinciden con aquellos de la Tabla 5-1 en los que se han conseguido los mejores resultados de exactitud de validación para cada una de las arquitecturas.

Para poder extraer más información sobre la bondad del modelo, se usarán las imágenes del conjunto de datos de prueba para predecir sus clases y, de esta manera, representar una matriz de confusión del modelo.

Para ello, se separará previamente cada dato en su imagen y su etiqueta correspondiente. Después, se predecirá las etiquetas de las imágenes de prueba usando el modelo y se codificará la etiqueta predicha para cada imagen mediante un número entero (las etiquetas estaban codificadas en caliente para el modelo):

```
imagenesEntren = datosEntren.map(lambda x, y: x)
etiquetasEntren = datosEntren.map(lambda x, y: y)

imagenesValid = datosValid.map(lambda x, y: x)
etiquetasValid = datosValid.map(lambda x, y: y)

imagenesPrueba = datosPrueba.map(lambda x, y: x)
etiquetasPrueba = datosPrueba.map(lambda x, y: y)
```

```
predPrueba = model.predict(imagenesPrueba)
predClasesPrueba = np.argmax(predPrueba, axis = 1)
print(predPrueba)
print(predClasesPrueba)
```

```
99/99 [==============================] - 4s 23ms/step
[[1.0479246e-11 1.0642404e-07 1.6568112e-09 ... 3.9620032e-09
  2.8603836e-12 8.6328683e-10]
 [8.4674561e-08 9.9999988e-01 5.2647825e-10 ... 4.9530950e-17
  1.0232223e-15 3.0391324e-15]
 [3.7408434e-08 3.4036653e-09 3.7933455e-11 ... 1.0030619e-12
  9.0951430e-12 7.3126909e-14]
 ...
 [9.7952613e-07 4.7323625e-05 6.3894413e-07 ... 1.0004372e-05
  6.3484251e-05 1.1839574e-05]
 [2.3534280e-06 3.6866631e-04 9.6384592e-06 ... 1.9135881e-04
  3.6387272e-07 1.3848256e-07]
 [7.7034185e-11 9.5914898e-10 4.3315722e-06 ... 7.3223483e-10
  1.8476589e-09 1.8893088e-08]]
[ 7  1 17 ... 32  7 10]
```

```
etiquetasPrueba3 = np.array(etiquetasPrueba2)
print(etiquetasPrueba3)
```

```
[ 7  1 17 ...  6  7 10]
```

Figura 5-5. Predicción de etiquetas usando el modelo

5.3. Matriz de confusión

Con las etiquetas reales de los datos de prueba y las etiquetas predichas por el modelo para esos mismos datos, se puede representar la matriz de confusión:

```
confusion_mtx = confusion_matrix(etiquetasPrueba3, predClasesPrueba)

# Graficar
fig, ax = plt.subplots(figsize = (15, 10))
ax = sns.heatmap(confusion_mtx, annot = True, fmt = 'd', ax = ax, cmap = 'Blues')
ax.set_xlabel('Etiqueta predicha')
ax.set_ylabel('Etiqueta real')
ax.set_title('Matriz de confusión');
```

Figura 5-6. Visualización de la matriz de confusión

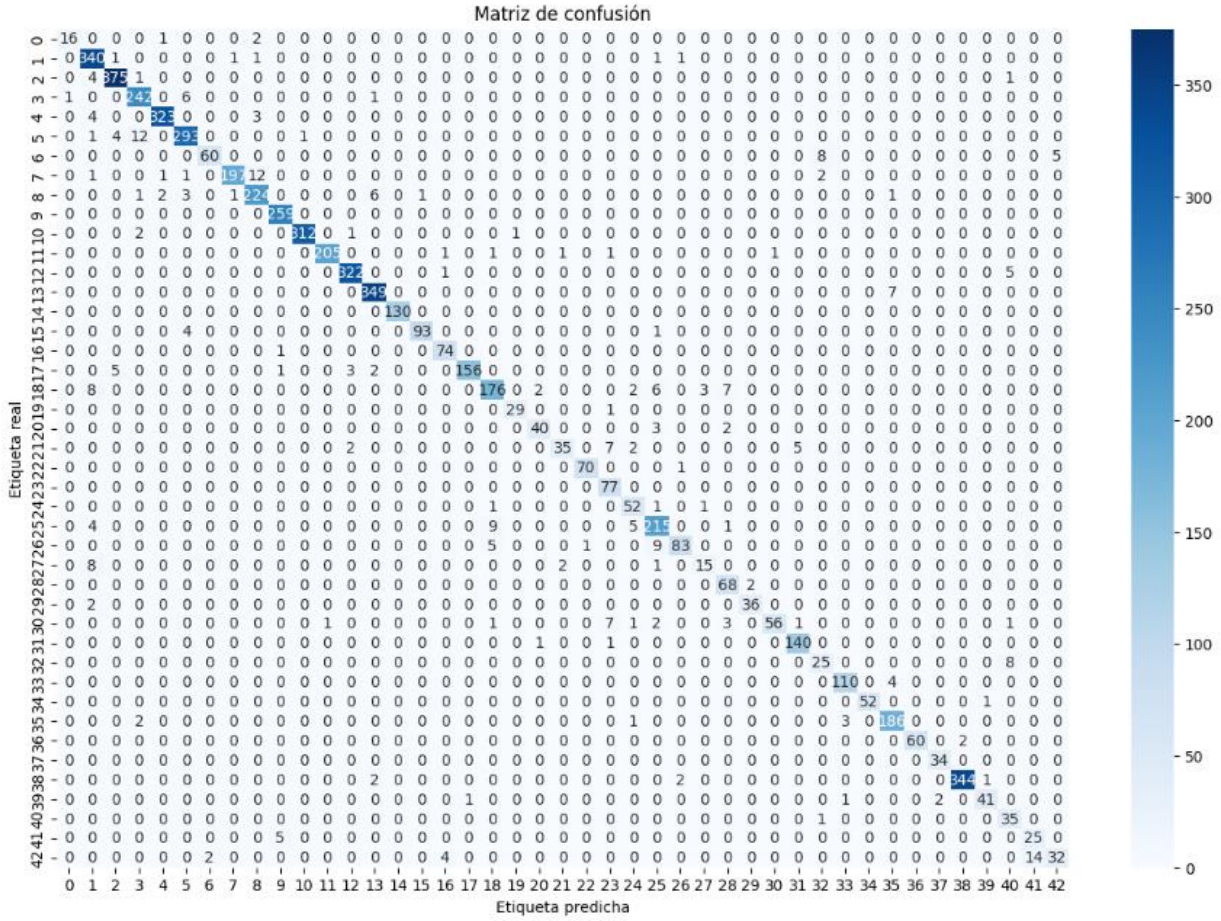

Figura 5-7. Matriz de confusión

A partir de una matriz de confusión, se pueden saber los VP, los VN, los FP y los FN para cada clase. Si la matriz de confusión fuese de la siguiente forma:

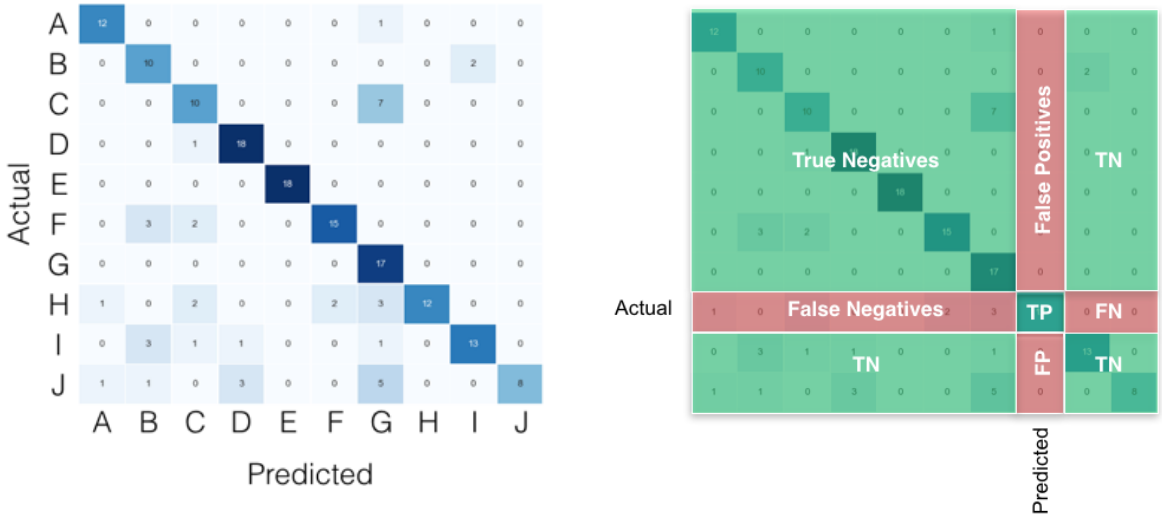

Figura 5-8. Ejemplo de una matriz de confusión (izquierda) y su interpretación (derecha) [121]

Por ejemplo, si se toma la clase 2, que tiene 375 VP (en la Figura 5-7, se aprecia que es la clase que más VP tiene), se pueden contar 6 FN y 10 FP. Como hay 6315 datos de prueba, se puede calcular que, para la clase 2, hay 5924 VN. Con estos datos, se podrían calcular la exactitud, la precisión, la exhaustividad, la especificidad y la métrica F1 para la clase 2:

$$Exactitud = \frac{VP + VN}{VP + VN + FN + FP} = \frac{375 + 5924}{6315} = 0{,}9975 \quad (5\text{–}1)$$

$$Precisión = \frac{VP}{VP + FP} = \frac{375}{375 + 10} = 0{,}9740 \quad (5\text{–}2)$$

$$Exhaustividad = \frac{VP}{VP + FN} = \frac{375}{375 + 6} = 0{,}9843 \quad (5\text{–}3)$$

$$Especificidad = \frac{VN}{VN + FP} = \frac{5924}{5924 + 10} = 0{,}9983 \quad (5\text{–}4)$$

$$F1 = \frac{2 \cdot precisión \cdot exhaustividad}{precisión + exhaustividad} = \frac{2 \cdot 0{,}9740 \cdot 0{,}9843}{0{,}9740 + 0{,}9843} = 0{,}9791 \quad (5\text{–}5)$$

En el anexo C, hay una tabla con todos estos cálculos para todas las clases de la Figura 5-7.

5.4. Comprobar algunos errores de predicción

Para finalizar, se van a comprobar algunos errores en la predicción del modelo. Se mostrarán 9 imágenes aleatorias del conjunto de pruebas cuyas etiquetas han sido predichas erróneamente por el modelo.

Para ello, se hará lo siguiente:

- Primero, se creará una lista llamada *errores* que será el resultado de restar la lista de etiquetas predichas y la de etiquetas reales. De esta manera, una etiqueta bien predicha por el modelo tendrá un error igual a 0 porque su valor será igual al de su etiqueta real correspondiente, mientras que una etiqueta mal predicha tendrá un valor distinto de 0 al tener un valor diferente de su etiqueta real correspondiente. Además, en la expresión se añadirá que se distinga entre errores distintos de cero y errores que no lo son, de modo que se asignará un valor lógico verdadero a aquellos valores distintos de 0, y un valor lógico falso a aquellos valores iguales a 0.
- A continuación, se creará una lista vacía llamada *idxError* que contendrá los índices de las etiquetas cuyo componente en la lista *errores* sea verdadero, es decir, contendrá solamente los índices de las etiquetas mal predichas por el modelo. Para ello, se recorrerá la lista *errores* con un bucle para almacenar aquellos componentes con un valor lógico verdadero. Se observa que la lista resultante *idxError* está compuesta por 309 índices, que supone el 4,89% de los 6315 componentes del conjunto de datos de prueba (cabe recordar que la exactitud de prueba del modelo era del 95,11%, como se mostraba en la Figura 5-4).
- Después, se crearán dos listas vacías: una llamada *etiquetasPruebaError*, que contendrá las etiquetas correspondientes a los índices de la lista *idxError*, e *imagenesPruebaError*, que contendrá las imágenes correspondientes a los índices de la lista *idxError*, es decir, contendrán aquellas etiquetas e imágenes del conjunto de prueba, respectivamente, que han sido mal predichas por el modelo.
- Finalmente, se mostrarán 9 imágenes aleatorias del 4,89% de etiquetas de prueba mal predichas por el modelo para comprobar en qué ha fallado.

```
errores = (predClasesPrueba - etiquetasPrueba2 != 0)
```

```
idxError = []
for idx, item in enumerate(errores):
    if item == True:
        idxError.append(idx)

print(len(idxError))
```

```
309
```

```
predClasesPruebaError = predClasesPrueba[idxError]
predPruebaError = predPrueba[idxError]
etiquetasPruebaError = etiquetasPrueba3[idxError]

imagenesPruebaError = []
for i in np.array(idxError):
    imagenesPruebaError.append(imagenesPrueba2[i])

print(len(etiquetasPruebaError), len(imagenesPruebaError))
```

```
309 309
```

```
rango_errores = np.random.randint(0, len(idxError), size = 9)
plt.figure(figsize = (14, 14))
for i in range(9):
        ax = plt.subplot(3, 3, i + 1)
        plt.imshow(imagenesPruebaError[rango_errores[i]])
        plt.title("Predicho: {}\nReal: {}".format(nombresClases[predClasesPruebaError[rango_errores[i]]],
        plt.axis('off')
```

Figura 5-9. Visualizar ejemplos de etiquetas mal predichas por el modelo

Figura 5-10. Imágenes aleatorias con etiquetas mal predichas por el modelo

Se puede observar que en 7 de estos 9 casos en los que el modelo ha predicho mal la etiqueta, las señales de tráfico correspondientes a la etiqueta predicha y a la etiqueta real tienen cierto parecido.

Por ejemplo, en la imagen superior izquierda, la señal es de intersección con prioridad y el modelo ha predicho que es de pavimento deslizante por hielo o nieve. Estas dos señales tienen en común su forma (ambas son triangulares) y sus colores (las dos son blancas con un borde rojo). El modelo realmente ha fallado en identificar el símbolo de la señal, que aparece un poco borroso y mal iluminado.

Otro ejemplo es la imagen inferior central, en la que la etiqueta real de la señal es sentido obligatorio (hacia adelante) y el modelo ha predicho que es una señal de sentido obligatorio (giro a la derecha). En este caso, las dos señales son también parecidas: ambas son circulares y de color azul con una flecha blanca. Aquí el modelo también ha fallado en identificar la flecha que aparece en la señal. Puede que la pegatina blanca que aparece en la parte inferior izquierda de la señal haya podido influir.

Por último, en la imagen central, el modelo ha predicho que es una señal de estrechamiento de calzada por la derecha y realmente es una señal de obras. Una vez más, ambas señales coinciden en su forma triangular y en su color blanco con borde rojo, por lo que el modelo ha vuelto a fallar en el símbolo de la señal que, como se puede observar en la imagen, está parcialmente oculto tras las ramas de un árbol.

6 CONCLUSIONES Y TRABAJOS FUTUROS

En vista del desarrollo del trabajo, probando con tres arquitecturas de red distintas, así como de los resultados obtenidos en dichas pruebas, se ha logrado diseñar y entrenar una red neuronal para el reconocimiento de imágenes de señales de tráfico con unos resultados de exactitud muy buenos.

Además, a diferencia de muchos sistemas comerciales de RST que solo reconocen un conjunto limitado de clases de señales de tráfico, principalmente de límite de velocidad, de prohibición de adelantamiento y de fin de prohibición, este modelo es capaz de reconocer otras clases, como algunas señales de advertencia de peligro, de prohibición de entrada y de obligación.

6.1. Trabajos futuros

Para concluir, cabría destacar una serie de mejoras y ampliaciones de este trabajo que se podrían desarrollar en futuros proyectos, como las siguientes:

- El principal motivo por el que se ha escogido este conjunto de datos para el entrenamiento y las pruebas de la red neuronal fue porque era una de las más numerosas y gratuita. Sin embargo, se ha podido comprobar cómo algunas imágenes, tras el preprocesamiento, quedaban pixeladas por el aumento de sus dimensiones, hasta el punto de que quizás podrían afectar el rendimiento de la red. Esto es debido a las dimensiones de las imágenes, que oscilaban entre 15×15 y 250×250 píxeles. Como las imágenes que se han usado para los datos se habían extraído de imágenes tomadas con una cámara desde la parte frontal de un vehículo, las más pequeñas correspondían a las de señales que estaban más lejos cuando se tomó la fotografía. Una mejora podría ser hacer pruebas con un conjunto de datos con imágenes que tengan más resolución.
- Este trabajo ha logrado reconocer señales de tráfico, pero podría complementarse con la detección de dichas señales mediante algoritmos de cuadros delimitadores, como YOLO, o mediante redes que implementen segmentación semántica, como U-Net [122]. Esto podría hacerse en dos pasos (primero, detectar la señal de tráfico y, después, reconocer su clase) o en un solo paso (detectar la clase de señal de tráfico).

REFERENCIAS

[1] «El sistema de reconocimiento de señales de tráfico,» Carglass, 4 marzo 2020. [En línea]. Available: https://www.carglass.es/blog/conduce-seguro/sistema-de-reconocimiento-de-senales-de-trafico/. [Último acceso: 25 noviembre 2023].

[2] «Deep Learning Specialization [5 courses] (DeepLearning.AI),» Coursera, [En línea]. Available: https://www.coursera.org/specializations/deep-learning. [Último acceso: 25 noviembre 2023].

[3] «¿Qué es la Inteligencia Artificial?,» Iberdrola, [En línea]. Available: https://www.iberdrola.com/innovacion/que-es-inteligencia-artificial. [Último acceso: 1 diciembre 2023].

[4] «Ley de IA de la UE: primera normativa sobre inteligencia artificial,» Parlamento Europeo, 12 junio 2023. [En línea]. Available: https://www.europarl.europa.eu/news/es/headlines/society/20230601STO93804/ley-de-ia-de-la-ue-primera-normativa-sobre-inteligencia-artificial. [Último acceso: 1 diciembre 2023].

[5] «¿Qué es la Inteligencia Artificial (IA)?,» IBM, [En línea]. Available: https://www.ibm.com/es-es/topics/artificial-intelligence. [Último acceso: 29 noviembre 2023].

[6] J. Yanes, «La historia de la Inteligencia Artificial,» BBVA Openmind, 14 marzo 2023. [En línea]. Available: https://www.bbvaopenmind.com/tecnologia/inteligencia-artificial/historia-de-la-inteligencia-artificial/. [Último acceso: 29 noviembre 2023].

[7] «Inteligencia artificial : definición, historia, usos, peligros,» DataScientest, 10 agosto 2022. [En línea]. Available: https://datascientest.com/es/inteligencia-artificial-definicion. [Último acceso: 29 noviembre 2023].

[8] Z. Parvez, «The Pioneers of AI: Marvin Minsky and the SNARC,» Medium, 25 enero 2023. [En línea]. Available: https://zahid-parvez.medium.com/history-of-ai-the-first-neural-network-computer-marvin-minsky-231c8bd58409. [Último acceso: 1 diciembre 2023].

[9] L. Gonzalez, «Historia de Machine Learning,» Aprende IA, 22 marzo 2018. [En línea]. Available: https://aprendeia.com/historia-de-machine-learning/. [Último acceso: 29 noviembre 2023].

[10] «Artificial Intelligence (AI) Coined at Dartmouth,» Dartmouth, [En línea]. Available: https://home.dartmouth.edu/about/artificial-intelligence-ai-coined-dartmouth. [Último acceso: 19 diciembre 2023].

[11] «General Problem Solver,» Wikipedia, la enciclopedia libre, [En línea]. Available: https://es.wikipedia.org/wiki/General_Problem_Solver. [Último acceso: 30 noviembre 2023].

[12] Na8, «Breve Historia de las Redes Neuronales Artificiales,» Aprende Machine Learning, 12 septiembre 2018. [En línea]. Available: https://www.aprendemachinelearning.com/breve-historia-de-las-redes-neuronales-artificiales/. [Último acceso: 16 noviembre 2023].

[13] «ELIZA,» Wikipedia, la enciclopedia libre, [En línea]. Available: https://es.wikipedia.org/wiki/ELIZA. [Último acceso: 1 diciembre 2023].

[14] F. C. Akyon, «Paper Review 1: ELIZA — A Computer Program for the Study of Natural Language Communication Between Man and Machine,» Medium, 11 octubre 2018. [En línea]. Available: https://medium.com/nlp-chatbot-survey/computational-lingustics-754c16fc7355. [Último acceso: 1 diciembre 2023].

[15] D. Dharanikota, «History of Neural Networks:,» LinkedIn, 23 julio 2017. [En línea]. Available: https://www.linkedin.com/pulse/history-neural-networks-datta-dharanikota. [Último acceso: 30 noviembre 2023].

[16] S. Kostadinov, «Understanding Backpropagation Algorithm,» Towards Data Science, 8 agosto 2019. [En línea]. Available: https://towardsdatascience.com/understanding-backpropagation-algorithm-7bb3aa2f95fd. [Último acceso: 17 noviembre 2023].

[17] «1997: "Deep Blue" vs. Garry Kasparov,» Wisconsin State Journal, 9 mayo 2017. [En línea]. Available: https://madison.com/1997-deep-blue-vs-garry-kasparov/image_eb21b580-34ef-11e7-9661-8fffe60bef5d.html. [Último acceso: 1 diciembre 2023].

[18] T. Susnjak, «ChatGPT: The End of Online Exam Integrity?,» ResearchGate, 20 diciembre 2022. [En línea]. Available: https://www.researchgate.net/figure/The-publicly-accessible-online-interface-to-ChatGPT-shows-the-text-input-prompt-that-the_fig1_366423865. [Último acceso: 1 diciembre 2023].

[19] «¿Qué es la inteligencia artificial (IA)?,» Amazon Web Services, [En línea]. Available: https://aws.amazon.com/es/what-is/artificial-intelligence/. [Último acceso: 18 julio 2023].

[20] «¿Qué es la inteligencia artificial o IA?,» Google Cloud, [En línea]. Available: https://cloud.google.com/learn/what-is-artificial-intelligence?hl=es-419. [Último acceso: 18 julio 2023].

[21] «Introducción al aprendizaje profundo,» AI Planet, [En línea]. Available: https://aiplanet.com/learn/getting-started-with-deep-learning-es/fundamentos-de-deep-learning-y-redes-neuronales/1771/introduccion-al-aprendizaje-profundo. [Último acceso: 18 julio 2023].

[22] «¿Qué es una red neuronal?,» Amazon Web Services, [En línea]. Available: https://aws.amazon.com/es/what-is/neural-network/. [Último acceso: 18 julio 2023].

[23] L. González, «¿Qué son las Redes Neuronales Artificiales?,» Aprende IA, 2 noviembre 2021. [En línea]. Available: https://aprendeia.com/que-son-las-redes-neuronales-artificiales/. [Último acceso: 18 julio 2023].

[24] «Perceptrón: ¿qué es y para qué sirve?,» DataScientest, 7 marzo 2022. [En línea]. Available: https://datascientest.com/es/perceptron-que-es-y-para-que-sirve. [Último acceso: 18 julio 2023].

[25] J. Pereira, «Perceptrón Multicapa y Algoritmo de Retropropagación,» MQL5, 3 marzo 2021. [En línea]. Available: https://www.mql5.com/es/articles/8908. [Último acceso: 18 julio 2023].

[26] «¿Qué es la Función de Activación?,» Telefónica Tech AI of Things, [En línea]. Available: https://aiofthings.telefonicatech.com/recursos/datapedia/funcion-activacion. [Último acceso: 19 julio 2023].

[27] F. Ramírez, «Las matemáticas del Machine Learning: Funciones de activación,» Telefónica Tech, 4 junio 2020. [En línea]. Available: https://telefonicatech.com/blog/las-matematicas-del-machine-learning-funciones-de-activacion. [Último acceso: 20 julio 2023].

[28] Bootcamp AI, «Redes neuronales. Programa de Visión Artificial/Computacional — Sesión2,» Medium, 14 noviembre 2019. [En línea]. Available: https://bootcampai.medium.com/redes-neuronales-13349dd1a5bb. [Último acceso: 20 julio 2023].

[29] F. Alonso, «Redes Neuronales y Deep Learning. Capítulo 2: La neurona,» Future Space, [En línea]. Available: https://www.futurespace.es/redes-neuronales-y-deep-learning-capitulo-2-la-neurona/. [Último acceso: 20 julio 2023].

[30] F. Rodríguez, «7 Consejos para Trabajar con Redes Neuronales en Aprendizaje Automático,» LinkedIn, 2 mayo 2023. [En línea]. Available: https://es.linkedin.com/pulse/7-consejos-para-trabajar-con-redes-neuronales-en-rodríguez-mgs. [Último acceso: 19 julio 2023].

[31] J. Finance, «Tipos de redes neuronales (Clasificación),» Inteligencia-Artificial.dev, 6 enero 2021. [En línea]. Available: https://inteligencia-artificial.dev/tipos-redes-neuronales/. [Último acceso: 19 agosto 2023].

[32] «¿Qué son las redes neuronales convolucionales?,» IBM, [En línea]. Available: https://www.ibm.com/es-es/topics/convolutional-neural-networks. [Último acceso: 21 agosto 2023].

[33] A. Pathak, «Redes neuronales convolucionales (CNN): introducción,» Geekflare, 30 agosto 2022. [En línea]. Available: https://geekflare.com/es/convolutional-neural-networks/. [Último acceso: 19 agosto 2023].

[34] «RNA de base radial,» Wikipedia, la enciclopedia libre, [En línea]. Available: https://es.wikipedia.org/wiki/RNA_de_base_radial. [Último acceso: 19 agosto 2023].

[35] R. Merritt, «¿Qué Es un Modelo Transformer?,» Nvidia, 19 abril 2022. [En línea]. Available: https://la.blogs.nvidia.com/2022/04/19/que-es-un-modelo-transformer/. [Último acceso: 14 diciembre 2023].

[36] R. Nag, «A Comprehensive Guide to Siamese Neural Networks,» Medium, 19 noviembre 2022. [En línea]. Available: https://medium.com/@rinkinag24/a-comprehensive-guide-to-siamese-neural-networks-3358658c0513. [Último acceso: 14 diciembre 2023].

[37] «¿Qué es una GAN? - Explicación sobre las redes generativas antagónicas,» Amazon Web Services, [En línea]. Available: https://aws.amazon.com/es/what-is/gan/. [Último acceso: 14 diciembre 2023].

[38] F. Alonso, «Redes Neuronales y Deep Learning | Descenso por gradiente,» Future Space, [En línea]. Available: https://www.futurespace.es/redes-neuronales-y-deep-learning-descenso-por-gradiente/. [Último acceso: 21 julio 2023].

[39] T. Quddoos, «Neural Network Basics: Loss and Cost Functions,» Medium, 6 noviembre 2021. [En línea]. Available: https://medium.com/artificialis/neural-network-basics-loss-and-cost-functions-9d089e9de5f8. [Último acceso: 18 diciembre 2023].

[40] «Descenso del gradiente,» Wikipedia, la enciclopedia libre, [En línea]. Available: https://es.wikipedia.org/wiki/Descenso_del_gradiente. [Último acceso: 21 julio 2023].

[41] «Gradiente,» Wikipedia, la enciclopedia libre, [En línea]. Available: https://es.wikipedia.org/wiki/Gradiente. [Último acceso: 21 julio 2023].

[42] «Parámetros de entrenamiento - Amazon Machine Learning,» Amazon Web Services, [En línea]. Available: https://docs.aws.amazon.com/es_es/machine-learning/latest/dg/training-parameters1.html. [Último acceso: 22 julio 2023].

[43] J. Jordan, «Setting the learning rate of your neural network.,» Jeremy Jordan, 1 marzo 2018. [En línea]. Available: https://www.jeremyjordan.me/nn-learning-rate/. [Último acceso: 22 julio 2023].

[44] D. E. Rumelhart, G. E. Hinton y R. J. Williams, «Learning representations by back-propagating errors,» 9 octubre 1986. [En línea]. Available: https://www.cs.toronto.edu/~hinton/absps/naturebp.pdf. [Último acceso: 17 noviembre 2023].

[45] A. Al-Masri, «How Does Backpropagation in a Neural Network Work?,» Built In, 7 agosto 2023. [En línea]. Available: https://builtin.com/machine-learning/backpropagation-neural-network. [Último acceso: 17 noviembre 2023].

[46] «Qué es Retropropagación Concepto y definición. Glosario,» Gamco, [En línea]. Available: https://gamco.es/glosario/retropropagacion/. [Último acceso: 22 julio 2023].

[47] H. Haule, «Understanding Error Backpropagation,» Towards Data Science, 23 octubre 2020. [En línea]. Available: https://towardsdatascience.com/error-backpropagation-5394d33ff49b. [Último acceso: 12 diciembre 2023].

[48] A. Arora, «8 Unique Machine Learning Interview Questions on Backpropagation,» Analytics Arora, 30 septiembre 2021. [En línea]. Available: https://analyticsarora.com/8-unique-machine-learning-interview-questions-on-backpropagation/. [Último acceso: 12 diciembre 2023].

[49] F. Alonso, «Redes Neuronales y Deep Learning | Backpropagation,» Future Space, [En línea]. Available: https://www.futurespace.es/redes-neuronales-y-deep-learning-brackpropagation/. [Último acceso: 22 julio 2023].

[50] Data Science Team, «La simple explicación del concepto de retropropagación,» Data Science, 3 mayo 2020. [En línea]. Available: https://datascience.eu/es/inteligencia-artificial/como-funciona-el-algoritmo-de-retropropagacion/. [Último acceso: 21 julio 2023].

[51] Y. LeCun, L. Bottou, G. B. Orr y K.-R. Müller, «Efficient BackProp,» 18 enero 2008. [En línea]. Available: https://cseweb.ucsd.edu/classes/wi08/cse253/Handouts/lecun-98b.pdf. [Último acceso: 17 noviembre 2023].

[52] P. Recuero, «Datos de entrenamiento vs datos de test,» Telefónica Tech, 24 enero 2022. [En línea]. Available: https://telefonicatech.com/blog/datos-entrenamiento-vs-datos-de-test. [Último acceso: 23 julio 2023].

[53] T. Shah, «About Train, Validation and Test Sets in Machine Learning,» Towards Data Science, 6 diciembre 2017. [En línea]. Available: https://towardsdatascience.com/train-validation-and-test-sets-72cb40cba9e7. [Último acceso: 23 julio 2023].

[54] «Training, validation, and test data sets,» Wikipedia, the free encyclopedia, [En línea]. Available: https://en.wikipedia.org/wiki/Training,_validation,_and_test_data_sets. [Último acceso: 23 julio 2023].

[55] C. Yang, «Deep learning: orthogonalization, evaluation metrics, train/dev/test set,» LinkedIn, 2 marzo 2018. [En línea]. Available: https://www.linkedin.com/pulse/deep-learning-orthogonalization-evaluation-metrics-set-chen-yang. [Último acceso: 23 julio 2023].

[56] L. Gonzalez, «Sobreajuste y Subajuste en Machine Learning,» Aprende IA, 16 noviembre 2018. [En línea]. Available: https://aprendeia.com/sobreajuste-y-subajuste-en-machine-learning/. [Último acceso: 24 julio 2023].

[57] Redacción KeepCoding, «¿Qué es la regularización en red convolucional?,» KeepCoding, 29 agosto 2022. [En línea]. Available: https://keepcoding.io/blog/regularizacion-red-convolucional/. [Último acceso: 25 julio 2023].

[58] K. Otness, X. Zhang, S. Chandrakaladharan y C. Raach, «Deep Learning,» Alfredo Canziani, 15 septiembre 2020. [En línea]. Available: https://atcold.github.io/pytorch-Deep-Learning/es/week14/14-3/. [Último acceso: 25 julio 2023].

[59] J. Durán, «Técnicas de Regularización Básicas para Redes Neuronales,» Medium, 7 octubre 2019. [En línea]. Available: https://medium.com/metadatos/t%C3%A9cnicas-de-regularizaci%C3%B3n-b%C3%A1sicas-para-redes-neuronales-b48f396924d4. [Último acceso: 25 julio 2023].

[60] J. Huber, «Batch normalization in 3 levels of understanding,» Towards Data Science, 6 noviembre 2020. [En línea]. Available: https://towardsdatascience.com/batch-normalization-in-3-levels-of-understanding-14c2da90a338. [Último acceso: 25 julio 2023].

[61] N. Adaloglou, «In-layer normalization techniques for training very deep neural networks,» AI Summer, 15 octubre 2020. [En línea]. Available: https://theaisummer.com/normalization/. [Último acceso: 12 diciembre 2023].

[62] «Stochastic Gradient Descent,» Interactive Chaos, [En línea]. Available: https://interactivechaos.com/es/manual/tutorial-de-machine-learning/stochastic-gradient-descent. [Último acceso: 26 julio 2023].

[63] «Mini-batch Gradient Descent,» Interactive Chaos, [En línea]. Available: https://interactivechaos.com/es/manual/tutorial-de-machine-learning/mini-batch-gradient-descent. [Último acceso: 26 julio 2023].

[64] «Momentum,» Interactive Chaos, [En línea]. Available: https://interactivechaos.com/es/manual/tutorial-de-machine-learning/momentum. [Último acceso: 26 julio 2023].

[65] «AdaGrad,» Interactive Chaos, [En línea]. Available: https://interactivechaos.com/es/manual/tutorial-de-machine-learning/adagrad. [Último acceso: 26 julio 2023].

[66] «RMSProp,» Interactive Chaos, [En línea]. Available: https://interactivechaos.com/es/manual/tutorial-de-machine-learning/rmsprop. [Último acceso: 26 julio 2023].

[67] «Adam,» Interactive Chaos, [En línea]. Available: https://interactivechaos.com/es/manual/tutorial-de-machine-learning/adam. [Último acceso: 26 julio 2023].

[68] «Transfer Learning para Deep Learning,» MathWorks, [En línea]. Available: https://es.mathworks.com/discovery/transfer-learning.html. [Último acceso: 18 agosto 2023].

[69] P. Baheti, «A Newbie-Friendly Guide to Transfer Learning,» V7 Labs, 12 octubre 2021. [En línea].

Available: https://www.v7labs.com/blog/transfer-learning-guide. [Último acceso: 15 diciembre 2023].

[70] «Multi-task learning,» Wikipedia, the free encyclopedia, [En línea]. Available: https://en.wikipedia.org/wiki/Multi-task_learning. [Último acceso: 18 agosto 2023].

[71] srivastava41099, «Introduction to Multi-Task Learning(MTL) for Deep Learning,» GeeksforGeeks, 19 enero 2023. [En línea]. Available: https://www.geeksforgeeks.org/introduction-to-multi-task-learningmtl-for-deep-learning/. [Último acceso: 18 agosto 2023].

[72] P. Antoniadis, «What Is End-to-End Deep Learning?,» Baeldung, 2 mayo 2023. [En línea]. Available: https://www.baeldung.com/cs/end-to-end-deep-learning. [Último acceso: 18 agosto 2023].

[73] «Matriz de confusión,» Wikipedia, la enciclopedia libre, [En línea]. Available: https://es.wikipedia.org/wiki/Matriz_de_confusión. [Último acceso: 28 julio 2023].

[74] G. Gasca, «Precisión y recuperación (Precision and recall),» Medium, 26 enero 2018. [En línea]. Available: https://medium.com/@gogasca_/precisi%C3%B3n-y-recuperaci%C3%B3n-precision-recall-dc3c92178d5b. [Último acceso: 28 julio 2023].

[75] P. Recuero, «Cómo interpretar la matriz de confusión: ejemplo práctico,» Telefónica Tech, 13 diciembre 2021. [En línea]. Available: https://telefonicatech.com/blog/como-interpretar-la-matriz-de-confusion-ejemplo-practico. [Último acceso: 28 julio 2023].

[76] «Resultados de los modelos de aprendizaje automático,» Microsoft Learn, 7 marzo 2023. [En línea]. Available: https://learn.microsoft.com/es-es/dynamics365/finance/finance-insights/confusion-matrix. [Último acceso: 28 julio 2023].

[77] R. Yamashita, M. Nishio, R. Kinh Gian Do y K. Togashi, «Convolutional neural networks: an overview and application in radiology,» SpringerOpen, 22 junio 2018. [En línea]. Available: https://insightsimaging.springeropen.com/articles/10.1007/s13244-018-0639-9. [Último acceso: 21 agosto 2023].

[78] dshahid380, «Convolutional Neural Network,» Towards Data Science, 24 febrero 2019. [En línea]. Available: https://towardsdatascience.com/covolutional-neural-network-cb0883dd6529. [Último acceso: 21 agosto 2023].

[79] M. Kumar, «How Padding helps in CNN ?,» Numpy Ninja, 22 diciembre 2020. [En línea]. Available: https://www.numpyninja.com/post/how-padding-helps-in-cnn. [Último acceso: 21 agosto 2023].

[80] D. Erroz, «Visualizando neuronas en Redes Neuronales Convolucionales,» 10 junio 2019. [En línea]. Available: https://academica-e.unavarra.es/bitstream/handle/2454/33694/memoria_TFG.pdf?sequence=1&isAllowed=y. [Último acceso: 21 agosto 2023].

[81] C. Kevin, «Feature Maps,» Medium, 11 mayo 2018. [En línea]. Available: https://medium.com/@chriskevin_80184/feature-maps-ee8e11a71f9e. [Último acceso: 15 diciembre 2023].

[82] R. Rodríguez, «Redes Convolucionales,» La Máquina Oráculo, 21 marzo 2021. [En línea]. Available: https://lamaquinaoraculo.com/deep-learning/redes-neuronales-convolucionales/. [Último acceso: 16 noviembre 2023].

[83] «¿Qué es la visión artificial?,» IBM, [En línea]. Available: https://www.ibm.com/es-es/topics/computer-vision. [Último acceso: 27 noviembre 2023].

[84] «Reconocimiento de objetos,» MathWorks, [En línea]. Available: https://es.mathworks.com/solutions/image-video-processing/object-recognition.html. [Último acceso: 27 noviembre 2023].

[85] «Reconocimiento de Objetos en Seguridad: Todo lo que Debes Saber,» Algotive, 7 septiembre 2023. [En línea]. Available: https://www.algotive.ai/es-mx/blog/reconocimiento-de-objetos-en-seguridad. [Último acceso: 2 diciembre 2023].

[86] Atulanand, «LeNet-5 Complete Architecture,» Medium, 29 octubre 2022. [En línea]. Available: https://medium.com/codex/lenet-5-complete-architecture-84c6d08215f9. [Último acceso: 16 noviembre 2023].

[87] T. Sharma, «Detailed Explanation of Resnet CNN Model.,» Medium, 6 marzo 2023. [En línea]. Available: https://medium.com/@sharma.tanish096/detailed-explanation-of-residual-network-resnet50-cnn-model-106e0ab9fa9e. [Último acceso: 20 noviembre 2023].

[88] K. He, X. Zhang, S. Ren y J. Sun, «Deep Residual Learning for Image Recognition,» 10 diciembre 2015. [En línea]. Available: https://arxiv.org/pdf/1512.03385.pdf. [Último acceso: 20 noviembre 2023].

[89] S. Mukherjee, «The Annotated ResNet-50,» Towards Data Science, 18 agosto 2022. [En línea]. Available: https://towardsdatascience.com/the-annotated-resnet-50-a6c536034758. [Último acceso: 20 noviembre 2023].

[90] N. Kundu, «Exploring ResNet50: An In-Depth Look at the Model Architecture and Code Implementation,» Medium, 23 enero 2023. [En línea]. Available: https://medium.com/@nitishkundu1993/exploring-resnet50-an-in-depth-look-at-the-model-architecture-and-code-implementation-d8d8fa67e46f. [Último acceso: 23 noviembre 2023].

[91] G. Boesch, «Deep Residual Networks (ResNet, ResNet50) 2024 Guide,» viso.ai, [En línea]. Available: https://viso.ai/deep-learning/resnet-residual-neural-network/. [Último acceso: 23 noviembre 2023].

[92] raghunandepu, «Understanding and implementation of Residual Networks(ResNets),» Medium, 30 octubre 2019. [En línea]. Available: https://medium.com/analytics-vidhya/understanding-and-implementation-of-residual-networks-resnets-b80f9a507b9c. [Último acceso: 22 agosto 2023].

[93] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens y Z. Wojna, «Rethinking the Inception Architecture for Computer Vision,» 11 diciembre 2015. [En línea]. Available: https://arxiv.org/pdf/1512.00567.pdf. [Último acceso: 23 noviembre 2023].

[94] «Guía avanzada de Inception v3,» Google Cloud, [En línea]. Available: https://cloud.google.com/tpu/docs/inception-v3-advanced?hl=es-419. [Último acceso: 23 noviembre 2023].

[95] K. Simonyan y A. Zisserman, «Very Deep Convolutional Networks for Large-Scale Image Recognition,» 10 abril 2015. [En línea]. Available: https://arxiv.org/pdf/1409.1556.pdf. [Último acceso: 24 noviembre 2023].

[96] Great Learning, «Everything you need to know about VGG16,» Medium, 23 septiembre 2021. [En línea]. Available: https://medium.com/@mygreatlearning/everything-you-need-to-know-about-vgg16-

7315defb5918. [Último acceso: 24 noviembre 2023].

[97] «Understanding VGG16: Concepts, Architecture, and Performance,» Datagen, [En línea]. Available: https://datagen.tech/guides/computer-vision/vgg16/. [Último acceso: 24 noviembre 2023].

[98] «About,» Ruhr-Universität Bochum, 16 septiembre 2010. [En línea]. Available: https://benchmark.ini.rub.de/gtsrb_about.html. [Último acceso: 2 diciembre 2023].

[99] O. Zuk, «As Computer Vision Explodes, Data Collection Needs To Change, Too.,» Datagen, 10 septiembre 2020. [En línea]. Available: https://datagen.tech/blog/computer-vision/. [Último acceso: 2 diciembre 2023].

[100] «¿Cómo funciona el reconocimiento de señales de tráfico?,» Swipcar, [En línea]. Available: https://swipcar.com/es/blog/reconocimiento-senales-trafico. [Último acceso: 2 diciembre 2023].

[101] R. Faragher, «Tesla's Full Self Driving Isn't The Only Technology With Speed Sign Detection Problems,» Forbes, 27 febrero 2023. [En línea]. Available: https://www.forbes.com/sites/ramseyfaragher/2023/02/27/teslas-full-self-driving-isnt-the-only-technology-with-speed-sign-detection-problems/. [Último acceso: 2 diciembre 2023].

[102] D. Babić, D. Babić, M. Fiolić y Ž. Šarić, «Analysis of Market-Ready Traffic Sign Recognition Systems in Cars: A Test Field Study,» MDPI, 21 junio 2021. [En línea]. Available: https://www.mdpi.com/1996-1073/14/12/3697. [Último acceso: 2 diciembre 2023].

[103] «German Traffic Sign Benchmarks,» Ruhr-Universität Bochum, 16 septiembre 2010. [En línea]. Available: https://benchmark.ini.rub.de/. [Último acceso: 2 diciembre 2023].

[104] «GTSRB dataset,» Ruhr-Universität Bochum, 16 septiembre 2010. [En línea]. Available: https://benchmark.ini.rub.de/gtsrb_dataset.html. [Último acceso: 2 diciembre 2023].

[105] «GTSRB Dataset,» Papers With Code, [En línea]. Available: https://paperswithcode.com/dataset/gtsrb. [Último acceso: 2 diciembre 2023].

[106] «¿Qué es Python? - Explicación del lenguaje Python,» Amazon Web Services, [En línea]. Available: https://aws.amazon.com/es/what-is/python/. [Último acceso: 3 diciembre 2023].

[107] «The Python Logo,» Python Software Foundation, [En línea]. Available: https://www.python.org/community/logos/. [Último acceso: 3 diciembre 2023].

[108] J. Larkin, «¿Qué es TensorFlow y para qué sirve?,» Incentro, 15 junio 2022. [En línea]. Available: https://www.incentro.com/es-ES/blog/que-es-tensorflow. [Último acceso: 3 diciembre 2023].

[109] «Crea modelos de aprendizaje automático de nivel de producción con TensorFlow,» TensorFlow, [En línea]. Available: https://www.tensorflow.org/?hl=es-419. [Último acceso: 3 diciembre 2023].

[110] Redacción KeepCoding, «¿Para qué sirve TensorFlow?,» KeepCoding, 8 agosto 2022. [En línea]. Available: https://keepcoding.io/blog/para-que-sirve-tensorflow/. [Último acceso: 3 diciembre 2023].

[111] «Keras: la API de alto nivel para TensorFlow,» TensorFlow, 22 enero 2022. [En línea]. Available: https://www.tensorflow.org/guide/keras?hl=es. [Último acceso: 3 diciembre 2023].

[112] «Keras: Deep Learning for humans,» Keras, [En línea]. Available: https://keras.io/. [Último acceso: 3 diciembre 2023].

[113] Tokio School, «Cómo usar Jupyter Notebook,» Tokio School, 13 abril 2023. [En línea]. Available: https://www.tokioschool.com/noticias/como-usar-jupyter-notebook/. [Último acceso: 3 diciembre 2023].

[114] «Project Jupyter | Home,» Jupyter Notebook, [En línea]. Available: https://jupyter.org/. [Último acceso: 3 diciembre 2023].

[115] Redacción Tokio, «Google Colab, una nueva herramienta de Google pensaba para especialistas de IA y Data Analysis,» Tokio School, 26 agosto 2022. [En línea]. Available: https://www.tokioschool.com/noticias/google-colab/. [Último acceso: 3 diciembre 2023].

[116] «Google Colab,» Google, [En línea]. Available: https://colab.research.google.com/?hl=es. [Último acceso: 3 diciembre 2023].

[117] J. M. Álvarez, «Categorías y la codificación One-Hot,» Blog de Jose Mariano Alvarez, 15 marzo 2018. [En línea]. Available: https://blog.josemarianoalvarez.com/2018/03/15/categorias-y-la-codificacion-one-hot/. [Último acceso: 17 diciembre 2023].

[118] «Dropout layer,» Keras, [En línea]. Available: https://keras.io/api/layers/regularization_layers/dropout/. [Último acceso: 19 diciembre 2023].

[119] «Convolutional Neural Networks Course (DeepLearning.AI),» Coursera, [En línea]. Available: https://www.coursera.org/learn/convolutional-neural-networks. [Último acceso: 22 agosto 2023].

[120] A. Brital, «Inception V3 CNN Architecture Explained .,» Medium, 23 octubre 2021. [En línea]. Available: https://medium.com/@AnasBrital98/inception-v3-cnn-architecture-explained-691cfb7bba08. [Último acceso: 24 noviembre 2023].

[121] «Scikit-learn: How to obtain True Positive, True Negative, False Positive and False Negative,» Stack Overflow, 9 julio 2015. [En línea]. Available: https://stackoverflow.com/questions/31324218/scikit-learn-how-to-obtain-true-positive-true-negative-false-positive-and-fal. [Último acceso: 4 diciembre 2023].

[122] «U-NET : todo lo que tienes que saber sobre la red neuronal de Computer Vision,» DataScientest, 26 abril 2022. [En línea]. Available: https://datascientest.com/es/u-net-lo-que-tienes-que-saber. [Último acceso: 17 diciembre 2023].

[123] «Principia Mathematica,» Wikipedia, la enciclopedia libre, [En línea]. Available: https://es.wikipedia.org/wiki/Principia_Mathematica. [Último acceso: 30 noviembre 2023].

[124] «Máquina Lisp,» Wikipedia, la enciclopedia libre, [En línea]. Available: https://es.wikipedia.org/wiki/Máquina_Lisp. [Último acceso: 30 noviembre 2023].

[125] «Lisp,» Wikipedia, la enciclopedia libre, [En línea]. Available: https://es.wikipedia.org/wiki/Lisp. [Último acceso: 30 noviembre 2023].

[126] «Feature Vector Definition,» Encord, [En línea]. Available: https://encord.com/glossary/feature-vector-definition/. [Último acceso: 14 diciembre 2023].

[127] J. G. Gomila, «Segmentación en Python: One hot, estandarización y PCA,» Frogames Formación, 16

septiembre 2021. [En línea]. Available: https://cursos.frogamesformacion.com/pages/blog/segmentacion-en-python-preparacion-de-datos. [Último acceso: 15 diciembre 2023].

[128] «Descripción general del ajuste de hiperparámetros,» Google Cloud, [En línea]. Available: https://cloud.google.com/ai-platform/training/docs/hyperparameter-tuning-overview?hl=es-419. [Último acceso: 23 julio 2023].

[129] S. Yadav, «Weight Initialization Techniques in Neural Networks,» Towards Data Science, 9 noviembre 2018. [En línea]. Available: https://towardsdatascience.com/weight-initialization-techniques-in-neural-networks-26c649eb3b78. [Último acceso: 19 diciembre 2023].

[130] SaffronEdge, «Feedforward vs Backpropagation ANN,» LinkedIn, 24 febrero 2023. [En línea]. Available: https://www.linkedin.com/pulse/feedforward-vs-backpropagation-ann-saffronedge1. [Último acceso: 18 diciembre 2023].

[131] «Convolución,» Wikipedia, la enciclopedia libre, [En línea]. Available: https://es.wikipedia.org/wiki/Convolución. [Último acceso: 21 agosto 2023].

[132] baeldung, «What Is the Purpose of a Feature Map in a Convolutional Neural Network,» Baeldung, 24 mayo 2023. [En línea]. Available: https://www.baeldung.com/cs/cnn-feature-map. [Último acceso: 15 diciembre 2023].

[133] «Segmentación semántica,» MathWorks, [En línea]. Available: https://es.mathworks.com/solutions/image-video-processing/semantic-segmentation.html. [Último acceso: 2 diciembre 2023].

[134] Simplilearn, «What is Epoch in Machine Learning?,» Simplilearn, 7 noviembre 2023. [En línea]. Available: https://www.simplilearn.com/tutorials/machine-learning-tutorial/what-is-epoch-in-machine-learning. [Último acceso: 17 diciembre 2023].

GLOSARIO

ANEXOS

Anexo A. Arquitecturas de ResNet50, Inception v3 y VGG16

En este anexo, se muestran las arquitecturas de ResNet50 [119], Inception v3 [120] y VGG16 [96]. La arquitectura de ResNet50 sin regularización se ha mostrado en la Figura 4-20.

Anexo A.1. Arquitectura de Inception v3 con regularización

```
def conv_with_Batch_Normalisation(prev_layer , nbr_kernels , filter_size , strides = (1,1) , padding = 'same'):

    X = tfl.Conv2D(filters=nbr_kernels, kernel_size = filter_size, strides = strides , padding = padding)(prev_layer)
    X = tfl.BatchNormalization(axis=3)(X)
    X = tfl.Activation('relu')(X)

    return X
```

Figura A-1. Bloque de convolución con normalización por lotes

```
def StemBlock(prev_layer):

    X = conv_with_Batch_Normalisation(prev_layer, nbr_kernels = 32, filter_size = (3,3) , strides = (2,2))
    X = conv_with_Batch_Normalisation(X, nbr_kernels = 32, filter_size = (3,3))
    X = conv_with_Batch_Normalisation(X, nbr_kernels = 64, filter_size = (3,3))
    X = tfl.MaxPool2D(pool_size = (3,3) , strides = (2,2))(X)
    X = conv_with_Batch_Normalisation(X, nbr_kernels = 80, filter_size = (1,1))
    X = conv_with_Batch_Normalisation(X, nbr_kernels = 192, filter_size = (3,3))
    X = tfl.MaxPool2D(pool_size=(3,3) , strides=(2,2))(X)

    return X
```

Figura A-2. Bloque troncal

```
def InceptionBlock_A(prev_layer, nbr_kernels):

    branch1 = conv_with_Batch_Normalisation(prev_layer, nbr_kernels = 64, filter_size = (1,1))
    branch1 = conv_with_Batch_Normalisation(branch1, nbr_kernels = 96, filter_size = (3,3))
    branch1 = conv_with_Batch_Normalisation(branch1, nbr_kernels = 96, filter_size = (3,3))

    branch2 = conv_with_Batch_Normalisation(prev_layer, nbr_kernels = 48, filter_size = (1,1))
    branch2 = conv_with_Batch_Normalisation(branch2, nbr_kernels = 64, filter_size = (3,3)) # may be 3*3

    branch3 = tfl.AveragePooling2D(pool_size = (3,3), strides = (1,1), padding = 'same')(prev_layer)
    branch3 = conv_with_Batch_Normalisation(branch3, nbr_kernels = nbr_kernels, filter_size = (1,1))

    branch4 = conv_with_Batch_Normalisation(prev_layer, nbr_kernels = 64, filter_size = (1,1))

    output = tfl.concatenate([branch1 , branch2 , branch3 , branch4], axis=3)

    return output
```

Figura A-3. Bloque A de incepción

```
def InceptionBlock_B(prev_layer , nbr_kernels):

    branch1 = conv_with_Batch_Normalisation(prev_layer, nbr_kernels = nbr_kernels, filter_size = (1,1))
    branch1 = conv_with_Batch_Normalisation(branch1, nbr_kernels = nbr_kernels, filter_size = (7,1))
    branch1 = conv_with_Batch_Normalisation(branch1, nbr_kernels = nbr_kernels, filter_size = (1,7))
    branch1 = conv_with_Batch_Normalisation(branch1, nbr_kernels = nbr_kernels, filter_size = (7,1))
    branch1 = conv_with_Batch_Normalisation(branch1, nbr_kernels = 192, filter_size = (1,7))

    branch2 = conv_with_Batch_Normalisation(prev_layer, nbr_kernels = nbr_kernels, filter_size = (1,1))
    branch2 = conv_with_Batch_Normalisation(branch2, nbr_kernels = nbr_kernels, filter_size = (1,7))
    branch2 = conv_with_Batch_Normalisation(branch2, nbr_kernels = 192, filter_size = (7,1))

    branch3 = tfl.AveragePooling2D(pool_size = (3,3) , strides = (1,1) , padding = 'same')(prev_layer)
    branch3 = conv_with_Batch_Normalisation(branch3, nbr_kernels = 192, filter_size = (1,1))

    branch4 = conv_with_Batch_Normalisation(prev_layer, nbr_kernels = 192, filter_size = (1,1))

    output = tfl.concatenate([branch1 , branch2 , branch3 , branch4], axis = 3)

    return output
```

Figura A-4. Bloque B de incepción

```
def InceptionBlock_C(prev_layer):

    branch1 = conv_with_Batch_Normalisation(prev_layer, nbr_kernels = 448, filter_size = (1,1))
    branch1 = conv_with_Batch_Normalisation(branch1, nbr_kernels = 384, filter_size = (3,3))
    branch1_1 = conv_with_Batch_Normalisation(branch1, nbr_kernels = 384, filter_size = (1,3))
    branch1_2 = conv_with_Batch_Normalisation(branch1, nbr_kernels = 384, filter_size = (3,1))
    branch1 = tfl.concatenate([branch1_1 , branch1_2], axis = 3)

    branch2 = conv_with_Batch_Normalisation(prev_layer, nbr_kernels = 384, filter_size = (1,1))
    branch2_1 = conv_with_Batch_Normalisation(branch2, nbr_kernels = 384, filter_size = (1,3))
    branch2_2 = conv_with_Batch_Normalisation(branch2, nbr_kernels = 384, filter_size = (3,1))
    branch2 = tfl.concatenate([branch2_1 , branch2_2], axis = 3)

    branch3 = tfl.AveragePooling2D(pool_size = (3,3) , strides = (1,1) , padding = 'same')(prev_layer)
    branch3 = conv_with_Batch_Normalisation(branch3, nbr_kernels = 192, filter_size = (1,1))

    branch4 = conv_with_Batch_Normalisation(prev_layer, nbr_kernels = 320, filter_size = (1,1))

    output = tfl.concatenate([branch1 , branch2 , branch3 , branch4], axis = 3)

    return output
```

Figura A-5. Bloque C de incepción

```
def ReductionBlock_A(prev_layer):

    branch1 = conv_with_Batch_Normalisation(prev_layer, nbr_kernels = 64, filter_size = (1,1))
    branch1 = conv_with_Batch_Normalisation(branch1, nbr_kernels = 96, filter_size = (3,3))
    branch1 = conv_with_Batch_Normalisation(branch1, nbr_kernels = 96, filter_size = (3,3) , strides = (2,2)) #, padding='valid'

    branch2 = conv_with_Batch_Normalisation(prev_layer, nbr_kernels = 384, filter_size=(3,3) , strides = (2,2))

    branch3 = tfl.MaxPool2D(pool_size = (3,3) , strides = (2,2) , padding = 'same')(prev_layer)

    output = tfl.concatenate([branch1 , branch2 , branch3], axis = 3)

    return output
```

Figura A-6. Bloque A de reducción

```
def ReductionBlock_B(prev_layer):

    branch1 = conv_with_Batch_Normalisation(prev_layer, nbr_kernels = 192, filter_size = (1,1))
    branch1 = conv_with_Batch_Normalisation(branch1, nbr_kernels = 192, filter_size = (1,7))
    branch1 = conv_with_Batch_Normalisation(branch1, nbr_kernels = 192, filter_size = (7,1))
    branch1 = conv_with_Batch_Normalisation(branch1, nbr_kernels = 192, filter_size = (3,3), strides = (2,2), padding = 'valid')

    branch2 = conv_with_Batch_Normalisation(prev_layer, nbr_kernels = 192, filter_size = (1,1))
    branch2 = conv_with_Batch_Normalisation(branch2, nbr_kernels = 320, filter_size = (3,3), strides = (2,2), padding = 'valid')

    branch3 = tfl.MaxPool2D(pool_size = (3,3), strides = (2,2))(prev_layer)

    output = tfl.concatenate([branch1 , branch2 , branch3], axis = 3)

    return output
```

Figura A-7. Bloque B de reducción

```
def InceptionV3(input_shape, classes):

    X_input = tfl.Input(input_shape)

    X = StemBlock(X_input)

    X = InceptionBlock_A(prev_layer = X, nbr_kernels = 32)
    X = InceptionBlock_A(prev_layer = X, nbr_kernels = 64)
    X = InceptionBlock_A(prev_layer = X, nbr_kernels = 64)

    X = ReductionBlock_A(prev_layer = X)

    X = InceptionBlock_B(prev_layer = X, nbr_kernels = 128)
    X = InceptionBlock_B(prev_layer = X, nbr_kernels = 160)
    X = InceptionBlock_B(prev_layer = X, nbr_kernels = 160)
    X = InceptionBlock_B(prev_layer = X, nbr_kernels = 192)

    X = ReductionBlock_B(prev_layer = X)

    X = InceptionBlock_C(prev_layer = X)
    X = InceptionBlock_C(prev_layer = X)

    X = tfl.GlobalAveragePooling2D()(X)
    X = tfl.Dense(2048, 'relu')(X)
    X = tfl.Dropout(0.2)(X)
    X = tfl.Dense(classes, 'softmax')(X)

    model = tf.keras.Model(inputs = X_input, outputs = X)

    return model
```

Figura A-8. Arquitectura de Inception v3 con regularización

Anexo A.2. Arquitectura de Inception v3 sin regularización

El código de los bloques de incepción y de reducción es el mismo, lo único que cambia es la eliminación de la capa de regularización por abandono (*Dropout*) en el código de la arquitectura de la red neuronal:

```
def InceptionV3(input_shape, classes):

    X_input = tfl.Input(input_shape)

    X = StemBlock(X_input)

    X = InceptionBlock_A(prev_layer = X, nbr_kernels = 32)
    X = InceptionBlock_A(prev_layer = X, nbr_kernels = 64)
    X = InceptionBlock_A(prev_layer = X, nbr_kernels = 64)

    X = ReductionBlock_A(prev_layer = X)

    X = InceptionBlock_B(prev_layer = X, nbr_kernels = 128)
    X = InceptionBlock_B(prev_layer = X, nbr_kernels = 160)
    X = InceptionBlock_B(prev_layer = X, nbr_kernels = 160)
    X = InceptionBlock_B(prev_layer = X, nbr_kernels = 192)

    X = ReductionBlock_B(prev_layer = X)

    X = InceptionBlock_C(prev_layer = X)
    X = InceptionBlock_C(prev_layer = X)

    X = tfl.GlobalAveragePooling2D()(X)
    X = tfl.Dense(2048, 'relu')(X)
    X = tfl.Dense(classes, 'softmax')(X)

    model = tf.keras.Model(inputs = X_input, outputs = X)

    return model
```

Figura A-9. Arquitectura de Inception v3 sin regularización

Anexo A.3. Arquitectura de ResNet50 con regularización

El código del bloque identidad y del bloque convolucional son iguales. Solo cambia la inclusión de una capa de regularización por abandono (*Dropout*) en el código de la arquitectura de la red neuronal:

```
def ResNet50(input_shape, classes):

    X_input = tfl.Input(input_shape)

    X = tfl.ZeroPadding2D((3, 3))(X_input)

    X = tfl.Conv2D(64, (7, 7), strides = (2, 2), kernel_initializer = glorot_uniform(seed=0))(X)
    X = tfl.BatchNormalization(axis = 3)(X)
    X = tfl.Activation('relu')(X)
    X = tfl.MaxPooling2D((3, 3), strides=(2, 2))(X)

    X = convolutional_block(X, f = 3, filters = [64, 64, 256], s = 1)
    X = identity_block(X, 3, [64, 64, 256])
    X = identity_block(X, 3, [64, 64, 256])

    X = convolutional_block(X, f = 3, filters = [128, 128, 512], s = 2)

    X = identity_block(X, 3, [128, 128, 512])
    X = identity_block(X, 3, [128, 128, 512])
    X = identity_block(X, 3, [128, 128, 512])

    X = convolutional_block(X, f = 3, filters = [256, 256, 1024], s = 2)

    X = identity_block(X, 3, [256, 256, 1024])
    X = identity_block(X, 3, [256, 256, 1024])
    X = identity_block(X, 3, [256, 256, 1024])
    X = identity_block(X, 3, [256, 256, 1024])
    X = identity_block(X, 3, [256, 256, 1024])

    X = convolutional_block(X, f = 3, filters = [512, 512, 2048], s = 2)

    X = identity_block(X, 3, [512, 512, 2048])
    X = identity_block(X, 3, [512, 512, 2048])

    X = tfl.AveragePooling2D((2, 2))(X)
    X = tfl.Flatten()(X)
    X = tfl.Dropout(0.2)(X)
    X = tfl.Dense(classes, activation='softmax', kernel_initializer = glorot_uniform(seed=0))(X)

    model = tf.keras.Model(inputs = X_input, outputs = X)

    return model
```

Figura A-10. Arquitectura de ResNet50 con regularización

Anexo A.4. Arquitectura de VGG16 con regularización

```
def VGG16(input_shape, classes):

    X_input = tfl.Input(input_shape)

    X = tfl.Conv2D(64, (3, 3), padding = 'same')(X_input)
    X = tfl.Conv2D(64, (3, 3), padding = 'same')(X)
    X = tfl.BatchNormalization(axis = 3)(X)
    X = tfl.Activation('relu')(X)
    X = tfl.MaxPooling2D((2, 2), strides=(2, 2))(X)

    X = tfl.Conv2D(128, (3, 3), padding = 'same')(X)
    X = tfl.Conv2D(128, (3, 3), padding = 'same')(X)
    X = tfl.BatchNormalization(axis = 3)(X)
    X = tfl.Activation('relu')(X)
    X = tfl.MaxPooling2D((2, 2), strides=(2, 2))(X)

    X = tfl.Conv2D(256, (3, 3), padding = 'same')(X)
    X = tfl.Conv2D(256, (3, 3), padding = 'same')(X)
    X = tfl.Conv2D(256, (3, 3), padding = 'same')(X)
    X = tfl.BatchNormalization(axis = 3)(X)
    X = tfl.Activation('relu')(X)
    X = tfl.MaxPooling2D((2, 2), strides=(2, 2))(X)

    X = tfl.Conv2D(512, (3, 3), padding = 'same')(X)
    X = tfl.Conv2D(512, (3, 3), padding = 'same')(X)
    X = tfl.Conv2D(512, (3, 3), padding = 'same')(X)
    X = tfl.BatchNormalization(axis = 3)(X)
    X = tfl.Activation('relu')(X)
    X = tfl.MaxPooling2D((2, 2), strides=(2, 2))(X)

    X = tfl.Conv2D(512, (3, 3), padding = 'same')(X)
    X = tfl.Conv2D(512, (3, 3), padding = 'same')(X)
    X = tfl.Conv2D(512, (3, 3), padding = 'same')(X)
    X = tfl.BatchNormalization(axis = 3)(X)
    X = tfl.Activation('relu')(X)
    X = tfl.MaxPooling2D((2, 2), strides=(2, 2))(X)

    X = tfl.Flatten()(X)
    X = tfl.Dense(256, activation = 'relu')(X)
    X = tfl.Dense(128, activation = 'relu')(X)
    X = tfl.Dropout(0.2)(X)
    X = tfl.Dense(classes, activation='softmax')(X)

    model = tf.keras.Model(inputs = X_input, outputs = X)

    return model
```

Figura A-11. Arquitectura de VGG16 con regularización

Anexo A.5. Arquitectura de VGG16 sin regularización

```
def VGG16(input_shape, classes):

    X_input = tfl.Input(input_shape)

    X = tfl.Conv2D(64, (3, 3), padding = 'same')(X_input)
    X = tfl.Conv2D(64, (3, 3), padding = 'same')(X)
    X = tfl.BatchNormalization(axis = 3)(X)
    X = tfl.Activation('relu')(X)
    X = tfl.MaxPooling2D((2, 2), strides=(2, 2))(X)

    X = tfl.Conv2D(128, (3, 3), padding = 'same')(X)
    X = tfl.Conv2D(128, (3, 3), padding = 'same')(X)
    X = tfl.BatchNormalization(axis = 3)(X)
    X = tfl.Activation('relu')(X)
    X = tfl.MaxPooling2D((2, 2), strides=(2, 2))(X)

    X = tfl.Conv2D(256, (3, 3), padding = 'same')(X)
    X = tfl.Conv2D(256, (3, 3), padding = 'same')(X)
    X = tfl.Conv2D(256, (3, 3), padding = 'same')(X)
    X = tfl.BatchNormalization(axis = 3)(X)
    X = tfl.Activation('relu')(X)
    X = tfl.MaxPooling2D((2, 2), strides=(2, 2))(X)

    X = tfl.Conv2D(512, (3, 3), padding = 'same')(X)
    X = tfl.Conv2D(512, (3, 3), padding = 'same')(X)
    X = tfl.Conv2D(512, (3, 3), padding = 'same')(X)
    X = tfl.BatchNormalization(axis = 3)(X)
    X = tfl.Activation('relu')(X)
    X = tfl.MaxPooling2D((2, 2), strides=(2, 2))(X)

    X = tfl.Conv2D(512, (3, 3), padding = 'same')(X)
    X = tfl.Conv2D(512, (3, 3), padding = 'same')(X)
    X = tfl.Conv2D(512, (3, 3), padding = 'same')(X)
    X = tfl.BatchNormalization(axis = 3)(X)
    X = tfl.Activation('relu')(X)
    X = tfl.MaxPooling2D((2, 2), strides=(2, 2))(X)

    X = tfl.Flatten()(X)
    X = tfl.Dense(256, activation = 'relu')(X)
    X = tfl.Dense(128, activation = 'relu')(X)
    X = tfl.Dense(classes, activation='softmax')(X)

    model = tf.keras.Model(inputs = X_input, outputs = X)

    return model
```

Figura A-12. Arquitectura de VGG16 sin regularización

Anexo B. Modelo de ResNet50 sin regularización

En este anexo, se muestra la arquitectura del modelo que resulta de ejecutar el código de la Figura 4-21:

```
Model: "model"
__________________________________________________________________________________________________
 Layer (type)                Output Shape              Param #   Connected to
==================================================================================================
 input_1 (InputLayer)        [(None, 64, 64, 3)]       0         []

 zero_padding2d (ZeroPaddin  (None, 70, 70, 3)         0         ['input_1[0][0]']
 g2D)

 conv2d (Conv2D)             (None, 32, 32, 64)        9472      ['zero_padding2d[0][0]']

 batch_normalization (Batch  (None, 32, 32, 64)        256       ['conv2d[0][0]']
 Normalization)

 activation (Activation)     (None, 32, 32, 64)        0         ['batch_normalization[0][0]']

 max_pooling2d (MaxPooling2  (None, 15, 15, 64)        0         ['activation[0][0]']
 D)

 conv2d_1 (Conv2D)           (None, 15, 15, 64)        4160      ['max_pooling2d[0][0]']

 batch_normalization_1 (Bat  (None, 15, 15, 64)        256       ['conv2d_1[0][0]']
 chNormalization)

 activation_1 (Activation)   (None, 15, 15, 64)        0         ['batch_normalization_1[0][0]'
                                                                 ]

 conv2d_2 (Conv2D)           (None, 15, 15, 64)        36928     ['activation_1[0][0]']

 batch_normalization_2 (Bat  (None, 15, 15, 64)        256       ['conv2d_2[0][0]']
 chNormalization)

 activation_2 (Activation)   (None, 15, 15, 64)        0         ['batch_normalization_2[0][0]'
                                                                 ]

 conv2d_3 (Conv2D)           (None, 15, 15, 256)       16640     ['activation_2[0][0]']

 conv2d_4 (Conv2D)           (None, 15, 15, 256)       16640     ['max_pooling2d[0][0]']

 batch_normalization_3 (Bat  (None, 15, 15, 256)       1024      ['conv2d_3[0][0]']
 chNormalization)

 batch_normalization_4 (Bat  (None, 15, 15, 256)       1024      ['conv2d_4[0][0]']
 chNormalization)

 add (Add)                   (None, 15, 15, 256)       0         ['batch_normalization_3[0][0]'
                                                                 , 'batch_normalization_4[0][0]
                                                                 ']

 activation_3 (Activation)   (None, 15, 15, 256)       0         ['add[0][0]']

 conv2d_5 (Conv2D)           (None, 15, 15, 64)        16448     ['activation_3[0][0]']
```

```
batch_normalization_5 (Bat  (None, 15, 15, 64)          256       ['conv2d_5[0][0]']
chNormalization)

activation_4 (Activation)   (None, 15, 15, 64)          0         ['batch_normalization_5[0][0]'
                                                                  ]

conv2d_6 (Conv2D)           (None, 15, 15, 64)          36928     ['activation_4[0][0]']

batch_normalization_6 (Bat  (None, 15, 15, 64)          256       ['conv2d_6[0][0]']
chNormalization)

activation_5 (Activation)   (None, 15, 15, 64)          0         ['batch_normalization_6[0][0]'
                                                                  ]

conv2d_7 (Conv2D)           (None, 15, 15, 256)         16640     ['activation_5[0][0]']

batch_normalization_7 (Bat  (None, 15, 15, 256)         1024      ['conv2d_7[0][0]']
chNormalization)

add_1 (Add)                 (None, 15, 15, 256)         0         ['batch_normalization_7[0][0]'
                                                                  , 'activation_3[0][0]']

activation_6 (Activation)   (None, 15, 15, 256)         0         ['add_1[0][0]']

conv2d_8 (Conv2D)           (None, 15, 15, 64)          16448     ['activation_6[0][0]']

batch_normalization_8 (Bat  (None, 15, 15, 64)          256       ['conv2d_8[0][0]']
chNormalization)

activation_7 (Activation)   (None, 15, 15, 64)          0         ['batch_normalization_8[0][0]'
                                                                  ]

conv2d_9 (Conv2D)           (None, 15, 15, 64)          36928     ['activation_7[0][0]']

batch_normalization_9 (Bat  (None, 15, 15, 64)          256       ['conv2d_9[0][0]']
chNormalization)

activation_8 (Activation)   (None, 15, 15, 64)          0         ['batch_normalization_9[0][0]'
                                                                  ]

conv2d_10 (Conv2D)          (None, 15, 15, 256)         16640     ['activation_8[0][0]']

batch_normalization_10 (Ba  (None, 15, 15, 256)         1024      ['conv2d_10[0][0]']
tchNormalization)

add_2 (Add)                 (None, 15, 15, 256)         0         ['batch_normalization_10[0][0]
                                                                  ',
                                                                   'activation_6[0][0]']

activation_9 (Activation)   (None, 15, 15, 256)         0         ['add_2[0][0]']

conv2d_11 (Conv2D)          (None, 8, 8, 128)           32896     ['activation_9[0][0]']

batch_normalization_11 (Ba  (None, 8, 8, 128)           512       ['conv2d_11[0][0]']
tchNormalization)

activation_10 (Activation)  (None, 8, 8, 128)           0         ['batch_normalization_11[0][0]
                                                                  ']
```

```
conv2d_12 (Conv2D)          (None, 8, 8, 128)          147584    ['activation_10[0][0]']

batch_normalization_12 (Ba  (None, 8, 8, 128)          512       ['conv2d_12[0][0]']
tchNormalization)

activation_11 (Activation)  (None, 8, 8, 128)          0         ['batch_normalization_12[0][0]
                                                                 ']

conv2d_13 (Conv2D)          (None, 8, 8, 512)          66048     ['activation_11[0][0]']

conv2d_14 (Conv2D)          (None, 8, 8, 512)          131584    ['activation_9[0][0]']

batch_normalization_13 (Ba  (None, 8, 8, 512)          2048      ['conv2d_13[0][0]']
tchNormalization)

batch_normalization_14 (Ba  (None, 8, 8, 512)          2048      ['conv2d_14[0][0]']
tchNormalization)

add_3 (Add)                 (None, 8, 8, 512)          0         ['batch_normalization_13[0][0]
                                                                 ',
                                                                  'batch_normalization_14[0][0]
                                                                 ']

activation_12 (Activation)  (None, 8, 8, 512)          0         ['add_3[0][0]']

conv2d_15 (Conv2D)          (None, 8, 8, 128)          65664     ['activation_12[0][0]']

batch_normalization_15 (Ba  (None, 8, 8, 128)          512       ['conv2d_15[0][0]']
tchNormalization)

activation_13 (Activation)  (None, 8, 8, 128)          0         ['batch_normalization_15[0][0]
                                                                 ']

conv2d_16 (Conv2D)          (None, 8, 8, 128)          147584    ['activation_13[0][0]']

batch_normalization_16 (Ba  (None, 8, 8, 128)          512       ['conv2d_16[0][0]']
tchNormalization)

activation_14 (Activation)  (None, 8, 8, 128)          0         ['batch_normalization_16[0][0]
                                                                 ']

conv2d_17 (Conv2D)          (None, 8, 8, 512)          66048     ['activation_14[0][0]']

batch_normalization_17 (Ba  (None, 8, 8, 512)          2048      ['conv2d_17[0][0]']
tchNormalization)

add_4 (Add)                 (None, 8, 8, 512)          0         ['batch_normalization_17[0][0]
                                                                 ',
                                                                  'activation_12[0][0]']

activation_15 (Activation)  (None, 8, 8, 512)          0         ['add_4[0][0]']

conv2d_18 (Conv2D)          (None, 8, 8, 128)          65664     ['activation_15[0][0]']

batch_normalization_18 (Ba  (None, 8, 8, 128)          512       ['conv2d_18[0][0]']
tchNormalization)
```

```
activation_16 (Activation)  (None, 8, 8, 128)          0         ['batch_normalization_18[0][0]
                                                                 ']

conv2d_19 (Conv2D)          (None, 8, 8, 128)          147584    ['activation_16[0][0]']

batch_normalization_19 (Ba  (None, 8, 8, 128)          512       ['conv2d_19[0][0]']
tchNormalization)

activation_17 (Activation)  (None, 8, 8, 128)          0         ['batch_normalization_19[0][0]
                                                                 ']

conv2d_20 (Conv2D)          (None, 8, 8, 512)          66048     ['activation_17[0][0]']

batch_normalization_20 (Ba  (None, 8, 8, 512)          2048      ['conv2d_20[0][0]']
tchNormalization)

add_5 (Add)                 (None, 8, 8, 512)          0         ['batch_normalization_20[0][0]
                                                                 ',
                                                                  'activation_15[0][0]']

activation_18 (Activation)  (None, 8, 8, 512)          0         ['add_5[0][0]']

conv2d_21 (Conv2D)          (None, 8, 8, 128)          65664     ['activation_18[0][0]']

batch_normalization_21 (Ba  (None, 8, 8, 128)          512       ['conv2d_21[0][0]']
tchNormalization)

activation_19 (Activation)  (None, 8, 8, 128)          0         ['batch_normalization_21[0][0]
                                                                 ']

conv2d_22 (Conv2D)          (None, 8, 8, 128)          147584    ['activation_19[0][0]']

batch_normalization_22 (Ba  (None, 8, 8, 128)          512       ['conv2d_22[0][0]']
tchNormalization)

activation_20 (Activation)  (None, 8, 8, 128)          0         ['batch_normalization_22[0][0]
                                                                 ']

conv2d_23 (Conv2D)          (None, 8, 8, 512)          66048     ['activation_20[0][0]']

batch_normalization_23 (Ba  (None, 8, 8, 512)          2048      ['conv2d_23[0][0]']
tchNormalization)

add_6 (Add)                 (None, 8, 8, 512)          0         ['batch_normalization_23[0][0]
                                                                 ',
                                                                  'activation_18[0][0]']

activation_21 (Activation)  (None, 8, 8, 512)          0         ['add_6[0][0]']

conv2d_24 (Conv2D)          (None, 4, 4, 256)          131328    ['activation_21[0][0]']

batch_normalization_24 (Ba  (None, 4, 4, 256)          1024      ['conv2d_24[0][0]']
tchNormalization)

activation_22 (Activation)  (None, 4, 4, 256)          0         ['batch_normalization_24[0][0]
                                                                 ']
```

```
conv2d_25 (Conv2D)          (None, 4, 4, 256)          590080    ['activation_22[0][0]']

batch_normalization_25 (Ba  (None, 4, 4, 256)          1024      ['conv2d_25[0][0]']
tchNormalization)

activation_23 (Activation)  (None, 4, 4, 256)          0         ['batch_normalization_25[0][0]
                                                                 ']

conv2d_26 (Conv2D)          (None, 4, 4, 1024)         263168    ['activation_23[0][0]']

conv2d_27 (Conv2D)          (None, 4, 4, 1024)         525312    ['activation_21[0][0]']

batch_normalization_26 (Ba  (None, 4, 4, 1024)         4096      ['conv2d_26[0][0]']
tchNormalization)

batch_normalization_27 (Ba  (None, 4, 4, 1024)         4096      ['conv2d_27[0][0]']
tchNormalization)

add_7 (Add)                 (None, 4, 4, 1024)         0         ['batch_normalization_26[0][0]
                                                                 ',
                                                                  'batch_normalization_27[0][0]
                                                                 ']

activation_24 (Activation)  (None, 4, 4, 1024)         0         ['add_7[0][0]']

conv2d_28 (Conv2D)          (None, 4, 4, 256)          262400    ['activation_24[0][0]']

batch_normalization_28 (Ba  (None, 4, 4, 256)          1024      ['conv2d_28[0][0]']
tchNormalization)

activation_25 (Activation)  (None, 4, 4, 256)          0         ['batch_normalization_28[0][0]
                                                                 ']

conv2d_29 (Conv2D)          (None, 4, 4, 256)          590080    ['activation_25[0][0]']

batch_normalization_29 (Ba  (None, 4, 4, 256)          1024      ['conv2d_29[0][0]']
tchNormalization)

activation_26 (Activation)  (None, 4, 4, 256)          0         ['batch_normalization_29[0][0]
                                                                 ']

conv2d_30 (Conv2D)          (None, 4, 4, 1024)         263168    ['activation_26[0][0]']

batch_normalization_30 (Ba  (None, 4, 4, 1024)         4096      ['conv2d_30[0][0]']
tchNormalization)

add_8 (Add)                 (None, 4, 4, 1024)         0         ['batch_normalization_30[0][0]
                                                                 ',
                                                                  'activation_24[0][0]']

activation_27 (Activation)  (None, 4, 4, 1024)         0         ['add_8[0][0]']

conv2d_31 (Conv2D)          (None, 4, 4, 256)          262400    ['activation_27[0][0]']

batch_normalization_31 (Ba  (None, 4, 4, 256)          1024      ['conv2d_31[0][0]']
tchNormalization)
```

```
activation_28 (Activation)  (None, 4, 4, 256)          0         ['batch_normalization_31[0][0]
                                                                 ']

conv2d_32 (Conv2D)          (None, 4, 4, 256)          590080    ['activation_28[0][0]']

batch_normalization_32 (Ba  (None, 4, 4, 256)          1024      ['conv2d_32[0][0]']
tchNormalization)

activation_29 (Activation)  (None, 4, 4, 256)          0         ['batch_normalization_32[0][0]
                                                                 ']

conv2d_33 (Conv2D)          (None, 4, 4, 1024)         263168    ['activation_29[0][0]']

batch_normalization_33 (Ba  (None, 4, 4, 1024)         4096      ['conv2d_33[0][0]']
tchNormalization)

add_9 (Add)                 (None, 4, 4, 1024)         0         ['batch_normalization_33[0][0]
                                                                 ',
                                                                  'activation_27[0][0]']

activation_30 (Activation)  (None, 4, 4, 1024)         0         ['add_9[0][0]']

conv2d_34 (Conv2D)          (None, 4, 4, 256)          262400    ['activation_30[0][0]']

batch_normalization_34 (Ba  (None, 4, 4, 256)          1024      ['conv2d_34[0][0]']
tchNormalization)

activation_31 (Activation)  (None, 4, 4, 256)          0         ['batch_normalization_34[0][0]
                                                                 ']

conv2d_35 (Conv2D)          (None, 4, 4, 256)          590080    ['activation_31[0][0]']

batch_normalization_35 (Ba  (None, 4, 4, 256)          1024      ['conv2d_35[0][0]']
tchNormalization)

activation_32 (Activation)  (None, 4, 4, 256)          0         ['batch_normalization_35[0][0]
                                                                 ']

conv2d_36 (Conv2D)          (None, 4, 4, 1024)         263168    ['activation_32[0][0]']

batch_normalization_36 (Ba  (None, 4, 4, 1024)         4096      ['conv2d_36[0][0]']
tchNormalization)

add_10 (Add)                (None, 4, 4, 1024)         0         ['batch_normalization_36[0][0]
                                                                 ',
                                                                  'activation_30[0][0]']

activation_33 (Activation)  (None, 4, 4, 1024)         0         ['add_10[0][0]']

conv2d_37 (Conv2D)          (None, 4, 4, 256)          262400    ['activation_33[0][0]']

batch_normalization_37 (Ba  (None, 4, 4, 256)          1024      ['conv2d_37[0][0]']
tchNormalization)

activation_34 (Activation)  (None, 4, 4, 256)          0         ['batch_normalization_37[0][0]
                                                                 ']

conv2d_38 (Conv2D)          (None, 4, 4, 256)          590080    ['activation_34[0][0]']
```

```
batch_normalization_38 (Ba  (None, 4, 4, 256)          1024      ['conv2d_38[0][0]']
tchNormalization)

activation_35 (Activation)  (None, 4, 4, 256)          0         ['batch_normalization_38[0][0]
                                                                 ']

conv2d_39 (Conv2D)          (None, 4, 4, 1024)         263168    ['activation_35[0][0]']

batch_normalization_39 (Ba  (None, 4, 4, 1024)         4096      ['conv2d_39[0][0]']
tchNormalization)

add_11 (Add)                (None, 4, 4, 1024)         0         ['batch_normalization_39[0][0]
                                                                 ',
                                                                  'activation_33[0][0]']

activation_36 (Activation)  (None, 4, 4, 1024)         0         ['add_11[0][0]']

conv2d_40 (Conv2D)          (None, 4, 4, 256)          262400    ['activation_36[0][0]']

batch_normalization_40 (Ba  (None, 4, 4, 256)          1024      ['conv2d_40[0][0]']
tchNormalization)

activation_37 (Activation)  (None, 4, 4, 256)          0         ['batch_normalization_40[0][0]
                                                                 ']

conv2d_41 (Conv2D)          (None, 4, 4, 256)          590080    ['activation_37[0][0]']

batch_normalization_41 (Ba  (None, 4, 4, 256)          1024      ['conv2d_41[0][0]']
tchNormalization)

activation_38 (Activation)  (None, 4, 4, 256)          0         ['batch_normalization_41[0][0]
                                                                 ']

conv2d_42 (Conv2D)          (None, 4, 4, 1024)         263168    ['activation_38[0][0]']

batch_normalization_42 (Ba  (None, 4, 4, 1024)         4096      ['conv2d_42[0][0]']
tchNormalization)

add_12 (Add)                (None, 4, 4, 1024)         0         ['batch_normalization_42[0][0]
                                                                 ',
                                                                  'activation_36[0][0]']

activation_39 (Activation)  (None, 4, 4, 1024)         0         ['add_12[0][0]']

conv2d_43 (Conv2D)          (None, 2, 2, 512)          524800    ['activation_39[0][0]']

batch_normalization_43 (Ba  (None, 2, 2, 512)          2048      ['conv2d_43[0][0]']
tchNormalization)

activation_40 (Activation)  (None, 2, 2, 512)          0         ['batch_normalization_43[0][0]
                                                                 ']

conv2d_44 (Conv2D)          (None, 2, 2, 512)          2359808   ['activation_40[0][0]']

batch_normalization_44 (Ba  (None, 2, 2, 512)          2048      ['conv2d_44[0][0]']
tchNormalization)
```

```
activation_41 (Activation)  (None, 2, 2, 512)          0         ['batch_normalization_44[0][0]
                                                                 ']

conv2d_45 (Conv2D)          (None, 2, 2, 2048)         1050624   ['activation_41[0][0]']

conv2d_46 (Conv2D)          (None, 2, 2, 2048)         2099200   ['activation_39[0][0]']

batch_normalization_45 (Ba  (None, 2, 2, 2048)         8192      ['conv2d_45[0][0]']
tchNormalization)

batch_normalization_46 (Ba  (None, 2, 2, 2048)         8192      ['conv2d_46[0][0]']
tchNormalization)

add_13 (Add)                (None, 2, 2, 2048)         0         ['batch_normalization_45[0][0]
                                                                 ',
                                                                  'batch_normalization_46[0][0]
                                                                 ']

activation_42 (Activation)  (None, 2, 2, 2048)         0         ['add_13[0][0]']

conv2d_47 (Conv2D)          (None, 2, 2, 512)          1049088   ['activation_42[0][0]']

batch_normalization_47 (Ba  (None, 2, 2, 512)          2048      ['conv2d_47[0][0]']
tchNormalization)

activation_43 (Activation)  (None, 2, 2, 512)          0         ['batch_normalization_47[0][0]
                                                                 ']

conv2d_48 (Conv2D)          (None, 2, 2, 512)          2359808   ['activation_43[0][0]']

batch_normalization_48 (Ba  (None, 2, 2, 512)          2048      ['conv2d_48[0][0]']
tchNormalization)

activation_44 (Activation)  (None, 2, 2, 512)          0         ['batch_normalization_48[0][0]
                                                                 ']

conv2d_49 (Conv2D)          (None, 2, 2, 2048)         1050624   ['activation_44[0][0]']

batch_normalization_49 (Ba  (None, 2, 2, 2048)         8192      ['conv2d_49[0][0]']
tchNormalization)

add_14 (Add)                (None, 2, 2, 2048)         0         ['batch_normalization_49[0][0]
                                                                 ',
                                                                  'activation_42[0][0]']

activation_45 (Activation)  (None, 2, 2, 2048)         0         ['add_14[0][0]']

conv2d_50 (Conv2D)          (None, 2, 2, 512)          1049088   ['activation_45[0][0]']

batch_normalization_50 (Ba  (None, 2, 2, 512)          2048      ['conv2d_50[0][0]']
tchNormalization)

activation_46 (Activation)  (None, 2, 2, 512)          0         ['batch_normalization_50[0][0]
                                                                 ']

conv2d_51 (Conv2D)          (None, 2, 2, 512)          2359808   ['activation_46[0][0]']
```

```
batch_normalization_51 (Ba  (None, 2, 2, 512)          2048      ['conv2d_51[0][0]']
tchNormalization)

activation_47 (Activation)  (None, 2, 2, 512)          0         ['batch_normalization_51[0][0]
                                                                 ']

conv2d_52 (Conv2D)          (None, 2, 2, 2048)         1050624   ['activation_47[0][0]']

batch_normalization_52 (Ba  (None, 2, 2, 2048)         8192      ['conv2d_52[0][0]']
tchNormalization)

add_15 (Add)                (None, 2, 2, 2048)         0         ['batch_normalization_52[0][0]
                                                                 ',
                                                                  'activation_45[0][0]']

activation_48 (Activation)  (None, 2, 2, 2048)         0         ['add_15[0][0]']

average_pooling2d (Average  (None, 1, 1, 2048)         0         ['activation_48[0][0]']
Pooling2D)

flatten (Flatten)           (None, 2048)               0         ['average_pooling2d[0][0]']

dense (Dense)               (None, 43)                 88107     ['flatten[0][0]']

==================================================================================================
Total params: 23675819 (90.32 MB)
Trainable params: 23622699 (90.11 MB)
Non-trainable params: 53120 (207.50 KB)
__________________________________________________________________________________________________
None
```

Anexo C. Tabla de métricas

En este anexo, se adjunta la tabla de métricas de la matriz de confusión de la Figura 5-7 para las 43 clases:

Tabla C-1. Tabla de métricas

Clase	VP	FP	FN	VN	Exactitud	Precisión	Exhaustividad	Especificidad	F1
0	16	1	3	6295	0,9994	0,9412	0,8421	0,9998	0,8889
1	340	32	5	5938	0,9941	0,9140	0,9855	0,9946	0,9484
2	375	10	6	5924	0,9975	0,9740	0,9843	0,9983	0,9791
3	242	18	8	6047	0,9959	0,9308	0,9680	0,9970	0,9490
4	323	4	7	5981	0,9983	0,9878	0,9788	0,9993	0,9833
5	293	14	18	5990	0,9949	0,9544	0,9421	0,9977	0,9482
6	60	2	13	6240	0,9976	0,9677	0,8219	0,9997	0,8889
7	197	2	17	6099	0,9970	0,9899	0,9206	0,9997	0,9540
8	224	18	15	6058	0,9948	0,9256	0,9372	0,9970	0,9314
9	259	7	0	6049	0,9989	0,9737	1	0,9988	0,9867
10	312	1	4	5998	0,9992	0,9968	0,9873	0,9998	0,9921

11	205	1	5	6104	0,9990	0,9951	0,9762	0,9998	0,9856
12	322	6	6	5981	0,9981	0,9817	0,9817	0,9990	0,9817
13	349	11	7	5948	0,9971	0,9694	0,9803	0,9982	0,9749
14	130	0	0	6185	1	1	1	1	1
15	93	1	5	6216	0,9990	0,9894	0,9490	0,9998	0,9688
16	74	6	1	6234	0,9989	0,9250	0,9867	0,9990	0,9548
17	156	1	11	6147	0,9981	0,9936	0,9341	0,9998	0,9630
18	176	17	28	6094	0,9929	0,9119	0,8627	0,9972	0,8866
19	29	1	1	6284	0,9997	0,9667	0,9667	0,9998	0,9667
20	40	3	5	6267	0,9987	0,9302	0,8889	0,9995	0,9091
21	35	3	16	6261	0,9970	0,9211	0,6863	0,9995	0,7865
22	70	1	1	6243	0,9997	0,9859	0,9859	0,9998	0,9859
23	77	17	0	6221	0,9973	0,8191	1	0,9973	0,9006
24	52	11	3	6249	0,9978	0,8254	0,9455	0,9982	0,8814
25	215	24	19	6057	0,9932	0,8996	0,9188	0,9961	0,9091
26	83	4	15	6213	0,9970	0,9540	0,8469	0,9994	0,8973
27	15	4	11	6285	0,9976	0,7895	0,5769	0,9994	0,6667
28	68	13	2	6232	0,9976	0,8395	0,9714	0,9979	0,9007
29	36	2	2	6275	0,9994	0,9474	0,9474	0,9997	0,9474
30	56	1	17	6241	0,9971	0,9825	0,7671	0,9998	0,8615
31	140	6	2	6167	0,9987	0,9589	0,9859	0,9990	0,9722
32	25	11	8	6271	0,9970	0,6944	0,7576	0,9982	0,7246
33	110	4	4	6197	0,9987	0,9649	0,9649	0,9994	0,9649
34	52	0	1	6262	0,9998	1	0,9811	1	0,9905
35	186	12	6	6111	0,9971	0,9394	0,9688	0,9980	0,9538
36	60	0	2	6253	0,9997	1	0,9677	1	0,9836
37	34	2	0	6279	0,9997	0,9444	1	0,9997	0,9714
38	344	2	5	5964	0,9989	0,9942	0,9857	0,9997	0,9899
39	41	2	4	6268	0,9990	0,9535	0,9111	0,9997	0,9318
40	35	15	1	6264	0,9975	0,7000	0,9722	0,9976	0,8140
41	25	14	5	6271	0,9970	0,6410	0,8333	0,9978	0,7246
42	32	5	20	6258	0,9960	0,8649	0,6154	0,9992	0,7191