VNIVERSITAT DE VALÈNCIA

Grado en Ingeniería
Electrónica de Telecomunicación

VNIVERSITAT DE VALÈNCIA

Trabajo Fin de Grado

Implementación de redes neuronales profundas para el reconocimiento de signos alfanuméricos manuscritos

Autor: Carles Serra Vendrell

Tutor: Emilio Soria Olivas

Septiembre 2020

VNIVERSITAT DE VALÈNCIA

Escola Tècnica Superior d'Enginyeria ETSE-UV

Trabajo Fin de Grado

Implementación de redes neuronales profundas para el reconocimiento de signos alfanuméricos manuscritos

Autor: Carles Serra Vendrell

Tutor: Emilio Soria Olivas

TRIBUNAL

PRESIDENTE/A:

VOCAL 1:

VOCAL 2:

FECHA DE DEFENSA:

CALIFICACIÓN:

Resumen:

El objetivo del trabajo de final de grado es el de implementar diferentes tipos de redes neuronales para el reconocimiento y clasificación de imágenes con signos alfanuméricos (letras y números) manuscritos. Se analizará el campo del aprendizaje profundo y, en concreto, las redes neuronales profundas. Una vez obtenidos los conocimientos sobre el tema se implementarán los modelos más extendidos para reconocer y clasificar imágenes de signos alfanuméricos manuscritos. Seguidamente se implementarán variaciones de estructuras/parámetros/algoritmos de aprendizaje de redes neuronales para mejorar el rendimiento y la precisión de los sistemas obtenidos.

Palabras claves: Redes Neuronales Convolucionales, Aprendizaje Profundo, Procesamiento de Imágenes, Reconocimiento de Imágenes, Clasificación de Imágenes.

Abstract:

The goal of the end of degree work is to implement different types of neural networks for the recognition and classification of images with alphanumeric signs (letters and numbers) in manuscripts. The field of deep learning and, in particular, deep neural networks will be analyzed. Once the knowledge on the subject is obtained, the most extended models to recognize and classify images of alphanumeric handwritten signs will be implemented. Then, variations of structures/-parameters/neural network learning algorithms will be implemented to improve the performance and accuracy of the obtained systems.

Keywords: Convolutional Neural Networks, Deep Learning, Image Processing, Image recognition, Image classification

Índice general

Índice de figuras

Capítulo 1

Introducción

1.1. Breve historia de las redes neuronales

Las redes neuronales son consideradas, por muchos, una nueva tecnología emergente, por la cantidad de avances y aplicaciones que han surgido en los últimos años. Multitud de sectores están aplicando esta tecnología para potenciar beneficios y solucionar problemas como la automatización de procesos, la optimización, el análisis y, un sin fin más de aplicaciones [10].

Las empresas están invirtiendo mucho dinero en el desarrollo de plataformas y aplicaciones [11] para solucionar problemas complejos de forma rápida y efectiva.

Estos algoritmos proporcionan tal valor para las empresas, que se espera una tasa de crecimiento anual compuesta del 43 % hasta 2025 [12]. La adopción de esta tecnología en las empresas actualmente se puede observar fácilmente en plataformas de empleo como LinkedIn. Buscando ofertas de empleo abiertas que requieren de conocimientos de esta tecnología, se puede analizar la demanda empresarial en este campo. Por ejemplo, si se buscan palabras clave como "*machine learning*" encontramos alrededor de 100.000 posiciones (Agosto, 2020) abiertas en todo el mundo. Actualmente la demanda de las empresas es elevada y, como se ha visto, aumentará en el futuro próximo.

La comunidad científica e investigadora ha jugado un papel importantísimo para el desarrollo del estado del arte de estos algoritmos. Nos encontramos en la época dorada de la investigación y del desarrollo de la inteligencia artificial. El número de publicaciones aumenta a un ritmo exponencial desde el año 2010 [1]. Esto se traslada de forma directa en los avances y descubrimientos sobre la materia de los últimos años. Como se verá más adelante, la multitud de campos donde se ha aplicado la inteligencia artificial con éxito es enorme. Algunos de los campos más predominantes serían la visión por ordenador, el procesamiento del lenguaje natural, la robótica, etc. Estos campos no serían lo que son en la actualidad sin la implementación de estas técnicas. No obstante, estamos lejos de alcanzar la inteligencia general, donde esta tecnología es capaz de realizar cualquier tarea. La inteligencia artificial todavía tiene ciertas limitaciones y existen industrias donde la tecnología todavía no es lo suficientemente madura para poder ser aplicada. Es por ello que tenemos que ser consecuentes y aplicar las virtudes de la inteligencia artificial en las áreas donde realmente es eficaz.

Las redes neuronales tienen una larga trayectoria. Esto sorprende a muchas personas por su reciente popularidad pero estos algoritmos datan de la época de los años cuarenta,

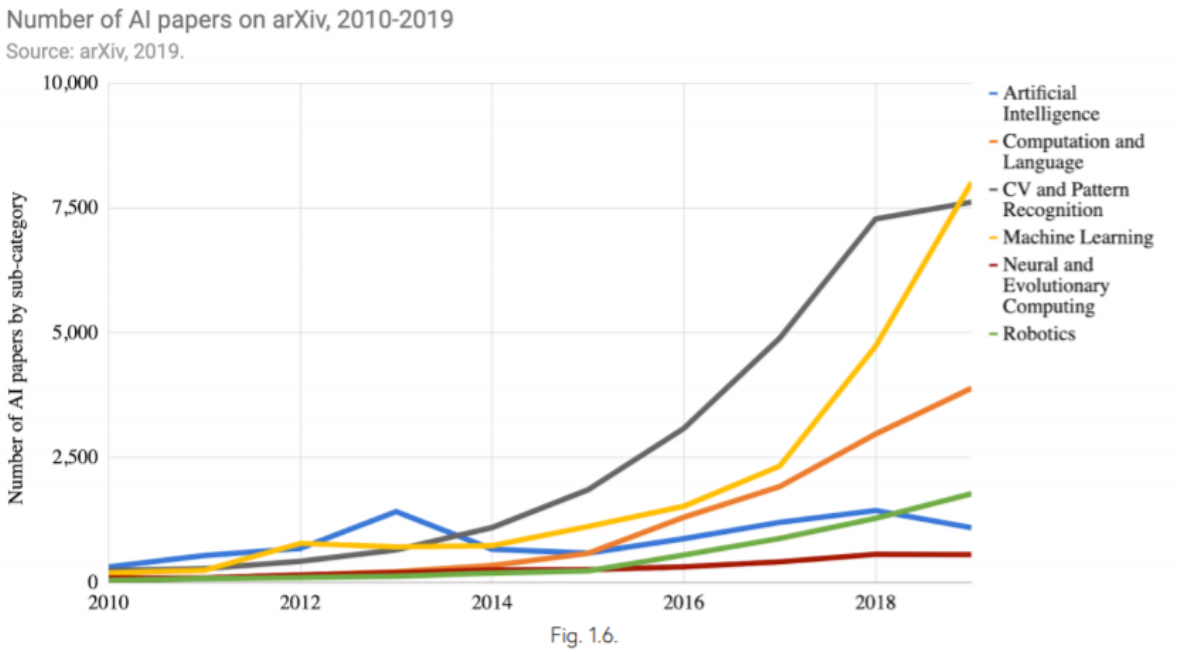

Figura 1.1: Número de publicaciones científicas sobre inteligencia artificial en arXiv, 2010-2019 [1].

cuando se empezaron a desarrollar los primeros modelos de redes neuronales. En 1943 se definió el primer modelo de red neuronal por parte de McCulloch y Pitts [13]. Desde aquel primer desarrollo, las redes neuronales han transitado tres ciclos principales de popularidad. La historia de las redes neuronales se entiende por oleadas, es decir, periodos donde el desarrollo se encontraba en su máximo apogeo y, otros periodos más oscuros donde las investigaciones se paralizan durante décadas. Estos períodos oscuros también son conocidos como inviernos de la inteligencia artificial, como analogía al nulo desarrollo.

Generalizando, se distinguen tres oleadas llamadas: *cybernetics*, *connectionism* y aprendizaje profundo. La primera etapa conocida como *cybernetics* surgió en los años cuarenta hasta los años sesenta. Tras esto siguió un invierno de veinte años hasta que empezó la segunda etapa, *connectionism*, que tuvo lugar entre los años ochenta hasta los años noventa. Al finalizar esta segunda etapa, al igual que en la primera, hubo otro período de inactividad científica.

Por último, surgió la última etapa, aprendizaje profundo, en el año 2006, la cual sigue presente en la actualidad [2]. En la figura 1.2 se observa cómo transcurrieron la primera y la segunda etapa en el tiempo según la frecuencia de estas palabras en publicaciones de libros científicos. Aparece cierto desfase en la línea temporal de las etapas debido al retraso de las publicaciones de los libros. Se suele tardar un par de años hasta la publicación de libros que reflejen el desarrollo científico. Para obtener mayor exactitud se deben utilizar las fechas de las publicaciones originales. Por tanto, el último periodo es demasiado reciente para que pueda aparecer en libros. No obstante, sí que se observa la tendencia negativa del *connectionism* la cual indica el inicio del aprendizaje profundo.

La frecuencia máxima de aparición aumenta conforme transcurren las etapas, indicando cómo etapa a etapa la tecnología madura e incrementa su relevancia. El incremento que existe entre la etapa *cybernetics* y la del *connectionism* es del doble. Por las tendencias que se han explicado anteriormente se predice que la etapa del aprendizaje profundo

continuará esta tendencia exponencial.

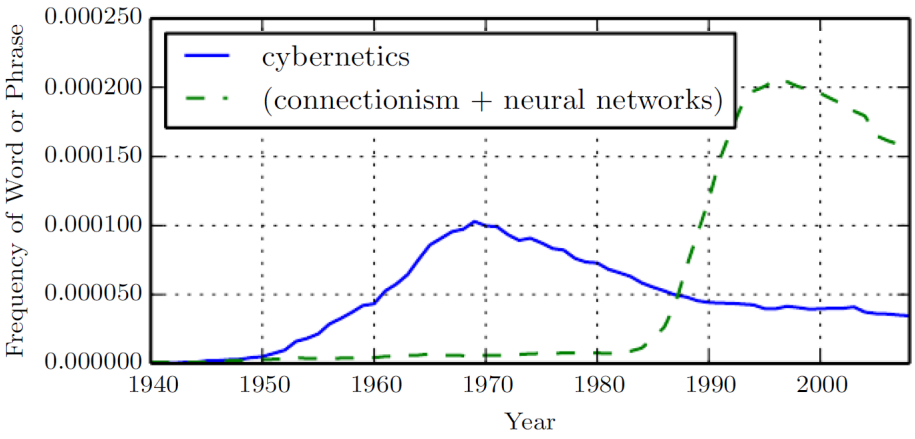

Figura 1.2: La figura muestra las olas de investigación según la frecuencia de las palabras *cybernetics*, *connectionism* y *neural networks* en *Google Book*. [2].

Los primeros sistemas creados en la década de los cuarenta estaban altamente inspirados en la biología del cerebro, en concreto, en las células nerviosas o neuronas, eran modelos que simulaban el comportamiento de las neuronas [14]. En la actualidad muchos medios han malinterpretado esto y, asocian las redes neuronales y el aprendizaje profundo con la neurociencia, cuando realmente son diferentes.

La primera red, pensada por McCulloch y Pitts, es un clasificador binario capaz de reconocer dos clases diferentes dada una entrada. Los coeficientes que determinan las clases se tenían que ajustar de forma manual. Esto suponía un problema ya que escala de manera inadecuada y, por lo tanto, no es viable de implementar. A finales de los años cincuenta, Roseblat mejoró el modelo anterior mediante el algoritmo de Perceptron [15] que dota el modelo de la capacidad de aprendizaje de los pesos de forma automática. Al proporcionar ejemplos de entradas de cada categoría al modelo este era capaz de aprender los pesos.

En el mismo período, Widrow y Hoff crearon otro algoritmo llamado elemento lineal adaptativo [16], también, capaz de optimizar los coeficientes mediante datos de entrada. Este algoritmo de aprendizaje es muy similar al empleado en filtros adaptativos (LMS), utilizado para adaptar la respuesta de un filtro en función de una señal de entrada, es decir, se optimizan los coeficientes según la señal de entrada.

Perceptron y *Adaline* son modelos lineales y, por tanto, poseen ciertas limitaciones. La más famosa y la que provocó la causa de su desaparición fue la limitación de aprender funciones no lineales. El trabajo de Minsky y Papert en 1969 [17] demostró que estos algoritmos eran clasificadores lineales incapaces de resolver problemas no lineales. Un ejemplo de este tipo de problemas, es la clasificación del conjunto de datos XOR. Como se observa en la figura 1.4, es imposible separar sus dos clases utilizando una simple recta y, por tanto, imposible de solucionar con estos algoritmos.

Así pues la etapa de *connectionism* terminó por las razones descritas anteriormente. Hasta la aparición del algoritmo *backpropagation* desarrollado por Rumelhart y Hinton [18] las redes neuronales no volvieron a aparecer. Con este último algoritmo se consigue poder entrenar en redes multicapa (redes neuronales con más de una capa). Esto implica

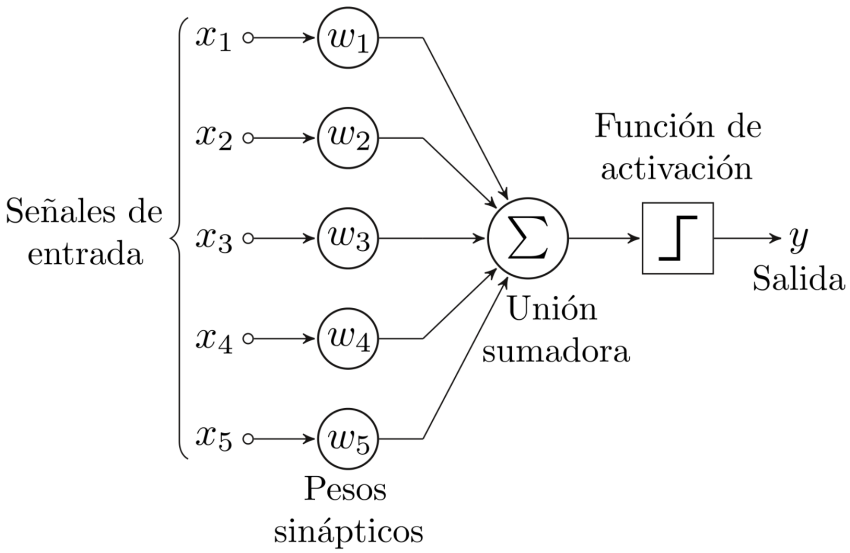

Figura 1.3: Esquema del algoritmo Perceptron [3].

construir redes más profundas y complejas y, por tanto, capaces de aproximar cualquier función continua tanto lineal como no lineal. Con estas propiedades se logra entrenar una red capaz de resolver problemas no lineales como el del conjunto de datos XOR. El nombre de *connectionism* o *parallel distributed processing* viene por la capacidad de las redes neuronales al combinar unidades computacionales simples (neuronas) para obtener un comportamiento inteligente.

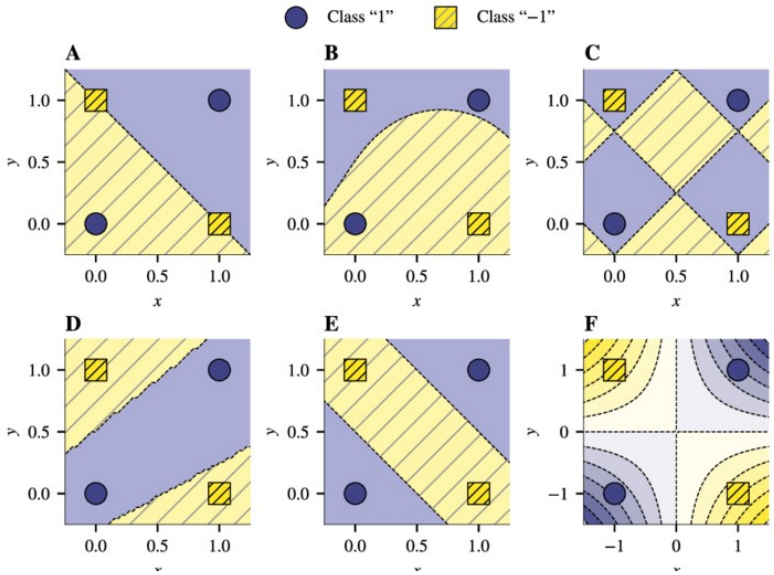

Figura 1.4: Resolución del problema XOR.

Con este nuevo algoritmo se consiguió solucionar uno de los problemas de las redes neuronales. No obstante, las redes multicapa no estaban preparadas para la tecnología y los recursos computacionales disponibles de aquella época. Los ordenadores no tenían suficiente potencia de cálculo, en comparación con los de la actualidad, y los conjuntos de datos etiquetados que tenían no eran suficientemente grandes para poder entrenar de forma efectiva los modelos.

Se sabía que estos algoritmos funcionaban bien, pero no fue hasta el año 2006, con el

inicio de la tercera oleada, cuando Geoffrey Hinton demostró que se podían entrenar de forma eficiente redes utilizando redes RBM [19]. Más tarde, el equipo de CIFAR consiguió generalizarlo para redes profundas [20, 21] al lograr entrenar redes más grandes y dominar estos modelos en las competiciones de inteligencia artificial frente a los algoritmos tradicionales. Desde entonces el término de aprendizaje profundo se empezó a popularizar y continúa hasta la actualidad, en la que nos encontramos en el máximo esplendor de esta tecnología. Además, hay que destacar que esta tendencia continúa y, cada vez los modelos y las bases de datos de etiquetas son más grandes, como se puede observar en modelos recientes como BERT, GPT-3, etc [22, 23] o bases de datos como ImageNet, Open Images, etc [6, 24]. Tecnologías como la computación en la nube y la tendencia actual de una sociedad digitalizada han sido claves para el desarrollo del aprendizaje profundo y lo continuarán siendo en el futuro.

Mencionar, durante los años 90, se desarrollaron avances en el modelado de secuencias mediante redes neuronales. Se plantearon las matemáticas para modelar la red LSTM (*Long Short-Term Memory*) [25] que hoy en día se utiliza para solucionar problemas del procesamiento natural del lenguaje. Sin esta tecnología, los asistentes personales virtuales no serían los mismos como los conocemos hoy en día.

Desde mi punto de vista, y el de muchos investigadores del sector como Andrew N.G, el aprendizaje profundo ha venido para quedarse. No creo que nos encontremos una burbuja como en las anteriores etapas, considero que se poseen los recursos necesarios para poder implementarla de forma efectiva en nuestra sociedad y cada año que pasa el valor proporcionado por inteligencia artificial aumenta, beneficiando a las empresas, gobiernos, instituciones, etc.

1.2. Principales tipos de aprendizaje y sus aplicaciones

El tipo de aprendizaje de los sistemas profundos se puede dividir en cuatro ramas principales: aprendizaje supervisado, aprendizaje no supervisado, aprendizaje autosupervisado y aprendizaje por refuerzo [4]:

- **Aprendizaje supervisado:** Consiste en mapear los datos de entrada a una salida conocida. Cuando se entrena un algoritmo de aprendizaje supervisado, los datos de entrenamiento consistirán en entradas emparejadas con las salidas correctas (también llamadas etiquetas). Normalmente las etiquetas son anotadas por humanos. Durante el entrenamiento, el algoritmo buscará patrones en los datos que se correlacionen con las salidas deseadas, es decir, el modelo aprenderá la función de mapeo capaz de asignar las clases asignadas a los datos de entrada. Por tanto el objetivo de un modelo de aprendizaje supervisado es predecir la etiqueta correcta para los datos de entrada recién presentados. Generalmente la mayoría de aplicaciones más importantes del aprendizaje profundo se encuentran en esta categoría. El aprendizaje supervisado se divide en dos subcategorías: clasificación que consiste en asignar clases a los datos de entrada y regresión en el que se determina un valor de la variable a predecir.

- **Aprendizaje no supervisado:** Se basa en encontrar agrupaciones de los datos de entrada. A diferencia del aprendizaje supervisado, no se dispone de las etiquetas junto con los datos de entrada. Se utiliza para entender las correlaciones presentes en

los datos de entrada, el modelo aprende estas relaciones autónomamente sin ningún tipo de supervisión. Algunos ejemplos son la compresión de datos, clustering, etc.

- **Aprendizaje autosupervisado:** Aprendizaje supervisado sin etiquetas. Las anotaciones o la creación de las etiquetas no viene del ser humano, sino del propio modelo utilizando los datos de entrada.

- **Aprendizaje por refuerzo:** Se trata de encontrar las acciones que maximizan la señal de recompensa. Los agentes reciben información del entorno y aprenden a tomar decisiones que maximicen la recompensa. Por ejemplo, en un videojuego donde se quiere obtener la máxima puntuación, un agente aprenderá a jugar al videojuego de forma autónoma a base de prueba y error, en ningún momento se proporciona información de cómo tiene que jugar el videojuego. Poco a poco aprende las acciones que mejoren la puntuación (mejoran las señal de recompensa) y con el tiempo se aprenderá qué acciones maximizan la puntuación. Se trata de una metodología relativamente reciente, sin embargo, ya se pueden encontrar muchas aplicaciones que demuestran su potencial, por ejemplo: en robótica, en videojuegos, en coches autónomos, etc [26].

Multitud de sectores se benefician del aprendizaje profundo para poder desarrollarse. La naturaleza de esta tecnología permite la automatización de procesos de forma muy efectiva. Muchos campos aplican la tecnología para automatizar procesos que antes se tenían que hacer de forma manual o mediante *feature engineering*, con el cual se utiliza el conocimiento específico en un área para extraer información de los datos. Este enfoque tiene ciertas desventajas, ya que se tiene que codificar a mano, muchas veces siendo inviable y, por tanto, no escala de forma óptima El aprendizaje profundo sustituye estas técnicas mediante un modelo que es capaz de aprender de forma autónoma. Muchos campos han recurrido a esta tecnología para la automatización de procesos, los cuales no serían viables sin el aprendizaje profundo. Por ejemplo, campos como la robótica, el procesamiento del lenguaje natural, el audio, la medicina, etc.

Una de las áreas con mayor repercusión, gracias a la inteligencia artificial, ha sido la visión por ordenador y el reconocimiento de patrones. Siendo el área con mayor tendencia de crecimiento en cuanto a publicaciones científicas se refiere en la última década [10]. Este hecho tiene una implicación directa con el número de aplicaciones del campo. Si se busca en las principales revistas o repositorios de publicaciones científicas enfocadas en la inteligencia artificial, se observa las diferentes subclases en las que se puede dividir el campo [27, 28]. La magnitud es tal, que el número de subcategorías es de alrededor de 700. Por nombrar algunas de ellas: segmentación, clasificación, detección de objetos, generación de imágenes [29], super resolución de imágenes, colorización y un largo etcétera.

De todas las subclases del campo de la visión por ordenador y el reconocimiento de patrones, el presente trabajo se centrará en la clasificación de imágenes, más específicamente en la clasificación multiclase y por tanto, un problema de tipo supervisado. El objetivo perseguido es el de construir un modelo capaz de asignar a las imágenes de entrada una clase determinada. En concreto, las imágenes de entrada corresponderá con dígitos manuscritos del número cero hasta el nueve. Por tanto, se tendrá un total de diez clases, una por dígito manuscrito. En resumen, el modelo predecirá a qué clase (número de dígitos 0-9) pertenecen las imágenes de dígitos manuscritos de entrada.

Capítulo 2

Redes convolucionales

2.1. Tipos de capas

Las redes neuronales convolucionales (CNN) o *convnets* son un tipo de modelo de aprendizaje profundo que se utilizan, especialmente, en aplicaciones de visión por ordenador. El nombre "red neuronal convolucional" viene de la utilización de la operación de convolución por la red, en lugar de la multiplicación matricial tradicional de las redes neuronales [2].

Como se ha visto en el capítulo anterior, existen multitud de aplicaciones prácticas para la visión por ordenador y, por lo general, todas utilizan las redes convolucionales. Por tanto, nosotros utilizaremos este tipo de redes para la clasificación de imágenes manuscritas, puesto que se obtendrá mejores resultados que frente a los algoritmos tradicionales, como las redes neuronales. Además, como cada aplicación necesita de estructuras y técnicas diferentes para aplicar las *convnets*, el proyecto se centrará solo en las estructuras típicas para la clasificación de imágenes, puesto que es el objetivo perseguido.

En el presente capítulo se describirá en qué consiste la operación y los parámetros de la convolución, se explicarán las principales capas utilizadas para la creación de modelos convolucionales enfocados en la clasificación de imágenes y se plantearán las ventajas de utilizar *convnets* frente a las redes neuronales tradicionales. Después, se especificarán los algoritmos de aprendizaje más importantes para que el modelo aprenda los parámetros de forma autónoma y se finalizará introduciendo las técnicas de regularización.

2.1.1. Capa convolucional

La operación de la convolución es denotada mediante el operador asterisco. Esta operación posee varios términos: el primer argumento x corresponde con la entrada, el segundo argumento w serán los núcleos o filtros. Y, la salida se llamará el mapa de características. La operación de convolución se implementará en el dominio discreto, pues es como se representa la información en un ordenador, donde x y w solo estarán definidos cuando la dimensión t sea entera:

$$S(n) = \sum_{k=-\infty}^{+\infty} x(k)w(n-k) \tag{2.1}$$

Los modelos normalmente utilizan arreglos de entrada multidimensionales (también llamados tensores). Por ejemplo, las imágenes en escala de grises son implementadas mediante tensores de dos dimensiones. Como la entrada es multidimensional, el núcleo también tendrá que serlo. Por tanto, si utilizamos matrices de dos dimensiones para la imagen de entrada, I, y el núcleo, K, obtenemos la siguiente expresión para la convolución [2]:

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n K(i-m, j-n) I(m,n) \tag{2.2}$$

A diferencia de la expresión anterior se ha implementado la convolución mediante sumatorios finitos sobre los elementos de la matriz. Para ello, se asume que los valores que no se encuentran almacenados en las matrices son cero. Con esto se consigue una implementación mucho más práctica.

La capa convolucional es la encargada de extraer las características de la imagen de entrada. Esta se compone por multitud de filtros (*kernels*) que mediante la operación de la convolución van extrayendo las diferentes características de la imagen de entrada. El número de filtros es un parámetro que se tiene que especificar y corresponderá con el número de mapas de salida. (También se suele conocer como la profundidad del mapa del mapa de características). Es decir a la salida se obtiene la presencia de los patrones de los filtros en diferentes localizaciones de la entrada. Y se tendrá tantos mapas de características como filtros.

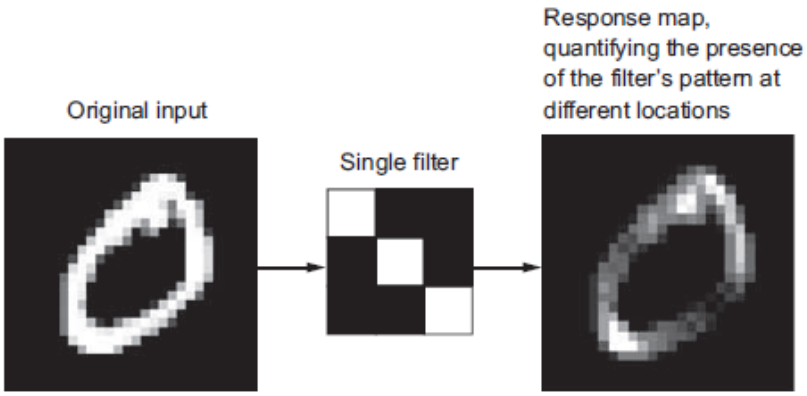

Figura 2.1: Concepto de mapa de características, que representa la presencia del filtro en diferentes localizaciones de la entrada [4].

La operación de convolución se realizará cada vez que se situe el centro de coordenadas del *kernel* en una localización concreta de la imagen. Se aplica ésta entre la región de la matriz de entrada y el *kernel*. La expresión 2.3 ejemplariza la operación de convolución ilustrada en la figura 2.2.

$$O(i,j) = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 93 & 139 & 101 \\ 26 & 252 & 186 \\ 135 & 230 & 48 \end{bmatrix} = \sum \begin{bmatrix} -93 & 0 & 101 \\ 52 & 0 & 372 \\ -135 & 0 & 48 \end{bmatrix} = 717 \tag{2.3}$$

El movimiento de los filtros sobre las diferentes localizaciones de las imágenes se determina mediante la operación conocida como *stride* [5]. Se fundamenta en mover el centro

del filtro sobre las diferentes coordenadas de la entrada, primero sobre el eje x (columnas) y después sobre el eje y (filas). Como ilustra la figura 2.2. La dimensión de los filtros tendrá que ser un número impar por obligación, para que exista un centro de coordenadas y poder aplicar el movimiento de *stride*. Por esto, las dimensiones más utilizadas suelen ser 3x3, 5x5, 7x7, etc. La dimensión variará en función de los patrones que se desean extraer y según las dimensiones de la entrada. Además, se suelen usar matrices cuadradas para de esta forma poder sacar el máximo partido de las librerías de álgebra lineal que están optimizadas para utilizar este tipo de matrices.

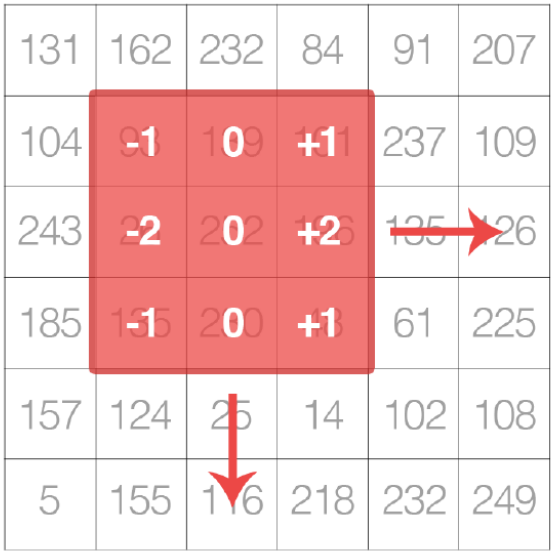

Figura 2.2: Movimiento de *stride* del *kernel* sobre una matriz de entrada [5].

El parámetro de *stride* en función de su valor determinan el movimiento de los filtros sobre las coordenadas de la entrada. Es decir, si el movimiento es continuo o si realiza saltos de coordenadas. Por ejemplo, un *stride* de valor dos provocaría que los filtros se desplazaran con saltos de dos en dos coordenadas de la entrada. Pero ojo, al utilizar *stride* con este valor se reduce a la mitad las dimensiones de mapa de caracteríticas de salida, pues se deja de calcular la operación de convolución en las coordenadas saltadas. Aunque se suele utilizar la capa de *pooling* (se explicará en el apartado 2.1.4) para esto, algunas arquitecturas utilizan el parámetro de *stride* para reducir la dimensionalidad. Como todo dependerá del tipo de problema nos beneficiarán de forma distinta cada técnica.

El efecto de borde provoca que las dimensiones de la entrada no coincidan con las del mapa de características de salida. El movimiento de *stride* se inicia en la primera coordenada válida en la que se pueda aplicar la convolución entre la región de la entrada y el *kernel*. Como se observa en la figura 2.3 no se puede centrar el *kernel* en la primera coordenada de la matriz y, por tanto, se utilizará la primera coordenada valida. Esto provocará que se reduzcan las dimensiones a la salida pues no se realiza la convolución en toda la entrada.

Para evitar este efecto se emplean técnicas de *padding*, que consisten en introducir las suficiente filas y columnas en la entrada para que cuando se centre el centro del *kernel* y se realice la convolución sin excluir ninguna celda de la entrada.

Aunque existen muchas técnicas de *padding*, se suele utilizar *zero-padding*. Esta consiste en rellenar con las filas y columnas mediante ceros, ya que no se añade información extra y el coste computacional es muy bajo.

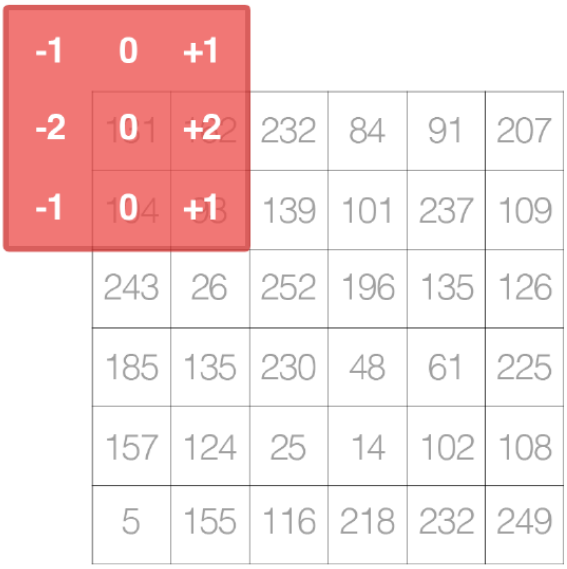

Figura 2.3: El *kernel* de dimensiones 3x3 sobresale de la matriz de entrada y, por tanto, no se puede calcular la convolución [5].

Una vez conocidos los principios básicos de las redes convolucionales, se mencionan las ventajas de éstas frente a las redes neuronales. Mientras que las redes neuronales clásicas aprenden patrones globales, las *convnets* son capaces aprender patrones locales. Esto tiene como consecuencia las siguientes ventajas a la hora de afrontar problemas de visión por ordenador [2, 4] :

- **Patrones invariantes:** Cuando aprenden un patrón en una localización concreta de una imagen, luego son capaces de reconocerlo en cualquier ubicación. Las redes neuronales, por el contrario, tienen que aprender el mismo patrón en cada localización haciéndolas menos eficientes. Como el mundo real sigue este tipo de naturaleza, las *convnets* con menos datos obtendrán mejores resultados frente las redes clásicas.
- **Jerarquías espaciales:** Son capaces de combinar los patrones locales aprendidos y crear jerarquías para aprender patrones más complejos. Siguen un orden de menos a más, combinando pequeños patrones para aprender abstracciones más complejas. Como se observa en la figura 2.4, primero se aprende patrones locales, como bordes y luego, combinándolos, se adquieren representaciones más complejas como una oreja, un ojo, etc.

2.1.2. Función de activación

Tras realizar la operación de la convolución se aplica una función de activación no lineal. Como la convolución es una operación lineal es necesario aplicar una función no lineal a la salida. De esta forma el modelo puede llegar a aprender representaciones más complejas. Con un modelo lineal no se puede sacar el máximo partido a la estructura de varias capas de representación porque, por muchas capas que se le añaden al modelo, solamente contendrán operaciones lineales. Una combinación lineal de modelos lineales es un modelo lineal.

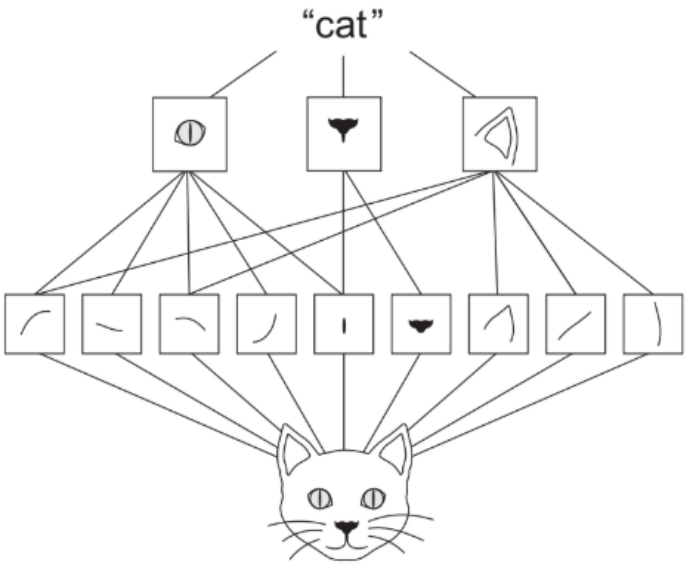

Figura 2.4: Ejemplo de jerarquía espacial de un gato [4].

Las funciones de activación tienen que ser continuas y derivables para poder aplicar los algoritmos de aprendizaje que se verán en el apartado 2.3. Muchas justificaciones de las funciones de activación necesitan de conocimientos de algoritmos de optimización. En este apartado se mencionan las justificaciones y supondrá el conocimiento de estos algoritmos.

Históricamente se ha utilizado la función activación *sigmoide* principalmente. Comprime los valores de entrada en los rangos de 0 a 1. Su principal ventaja es que suaviza el gradiente haciendo que sea más fácil de calcular el algoritmo de aprendizaje. Por contra, la salida no está centrada en el cero y es saturable y, por tanto, provoca el desvanecimiento del gradiente [30]. Los valores muy grandes o pequeños son saturados de forma asintótica causando que el gradiente sea muy pequeño en estos casos, donde el modelo es incapaz de aprender. Un caso particular que podría ser muy peligroso sería, por ejemplo, que el modelo se iniciara con parámetros muy grandes, provocando la parálisis de muchas neuronas desde un inicio.

Además se trata de una función que requiere de un coste computacional elevado por la exponencial que se tiene que calcular. Es una función poco recomendable de utilizar en la actualidad y, raramente, es implementada en modelos. Aunnque en las capas de salida de muchas redes es necesaría, por ejemplo, en redes clasificadoras.

$$t = \sum_{i=1}^{n} w_i x_i \tag{2.4}$$

$$\sigma(t) = \frac{1}{1 + e^{-t}} \tag{2.5}$$

Otra función muy utilizada es la *tangente hiperbólica*. Esta fue muy utilizada en la década de 1990. La función comprime los valores de entrada entre los rangos -1 y 1. En esencia tiene las mismas propiedades que la función sigmoide pero la salida está centrada en cero, no obstante, sí que es saturable. El coste computacional es muy similar también. Por esta capacidad extra se recomienda utilizar *tanh* frente a la tradicional *sigmoide*.

$$tanh(t) = \frac{(e^t - e^{-t})}{e^t + e^{-t}} \tag{2.6}$$

Como se observa se trata de una versión escalada de la *sigmoide*:

$$tanh(t) = 2\sigma(2t) - 1 \tag{2.7}$$

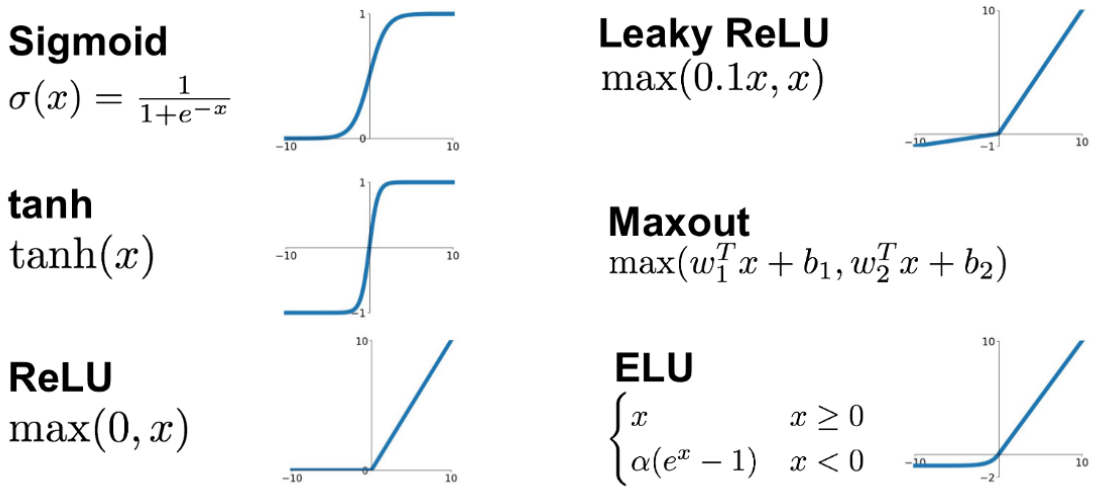

Figura 2.5: Representación gráfica de diversas funciones de activación.

En la actualidad existen mejores alternativas a las tradicionales que se acaban de ver. Una de las funciones de activación más utilizadas es la *ReLU* (*Rectified Linear Unit*) [31]. Está función a la salida es cero si la entrada es negativa y luego aumenta linealmente con la entrada positiva con una pendiente de valor 1 (Ecuación 2.8). Se trata de una función extremadamente simple y muy fácil de implementar. A diferencia de las anteriores no requiere de operaciones complejas como la exponencial, solo se necesita calcular un umbral. Es por esto que su coste computacional es mucho menor que sus antecesores. Además de su eficiencia, la función *ReLU* proporciona mejores resultados empíricos de hasta 6 veces [6], aunque dependerá del problema planteado. La figura 2.6 muestra la comparativa de la función *ReLU* frente a *tanh*, donde claramente converge mucho más rápido. Por contra esta función cuando la salida es cero el gradiente también es cero. Esto provoca que algunas neuronas durante el entrenamiento dejen de aprender reduciendo la capacidad total de aprendizaje del modelo.

$$f(x) = max(0, x) \tag{2.8}$$

Existen algunas variantes de la función *Leaky ReLU* [32] o *PReLU* [33] que consiguen solucionar este problema de la parálisis de neuronas:

- ***Leaky ReLU:*** Tiene una pequeña pendiente negativa cuando la entrada es menor que cero. De esta forma consigue tener un pequeño gradiente cuando la unidad no está activa. La pendiente se determina mediante el coeficiente de fuga α que se tendrá que parametrizar.

$$f(x) = \begin{cases} x & x >= 0 \\ \alpha x & x < 0 \end{cases} \tag{2.9}$$

- ***PReLU:*** Conocida también con el nombre de "ReLU paramétrica" por su capacidad de aprender el parámetro α de forma autónoma cada vez que se realiza una activación de la función.

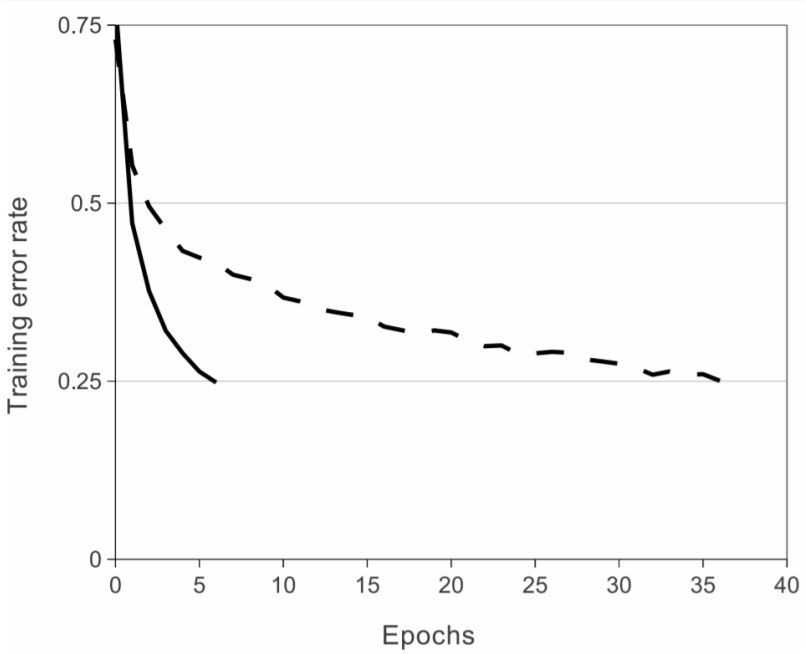

Figura 2.6: Una red convolucional de cuatro capas con *ReLU* (Línea Sólida) alcanza una tasa de error del 25 % en el conjunto de datos CIFAR-10, siendo seis veces más rápido que una red equivalente con *tanh* (Línea discontinua) [6].

Añadir que existe más funciones de activación como *Maxout*, *ELU*, etc. De forma general, se suele recomendar utilizar ReLU para implementar modelos ya que funciona muy bien en multitud de situaciones [5]. Como base de partida se debe utilizar *ReLU* para implementar modelos puesto que garantiza buenos resultados. Cuando el modelo esté definido y tenga cierta solidez se puede experimentar con otras funciones más avanzadas.

2.1.3. Capa totalmente conectada

La capa totalmente conectada es, básicamente, el algoritmo del perceptrón multicapa. El perceptrón multicapa es un sistema de neuronas interconectadas. Una neurona se define como una unidad computacional determinada por la siguiente operación:

$$f\left(\sum_{i=0}^{n} w_i x_i + b\right) \tag{2.10}$$

Se calcula el producto escalar entre la entrada x_i y los pesos w_i. A continuación, se suma el vector de tendencia b y, por último, se aplica la función de activación f a este resultado. Los coeficientes, o pesos, son los parámetros que se optimizan mediante el algoritmo de aprendizaje y, de esta manera, se obtiene la máxima precisión.

La función de activación como se ha mencionado anteriormente es necesaria para implementar no linealidades a la salida del producto escalar y poder modelar problemas más

complejos. El vector de tendencia es empleado para compensar el resultado del producto escalar. Por ejemplo, si tenemos una entrada de 0 y se necesita 2 a la salida. El producto escalar de la entrada con cualquier peso siempre será 0 pero, al añadir la tendencia se puede compensar este resultado, logrando la salida deseada. Como los pesos, es un parámetro que se optimiza por el algoritmo de aprendizaje y, por tanto, al dotar a las neuronas de una tendencia se logra aumentar la flexibilidad y mejorar la generalización de la red.

Una vez definida el funcionamiento de una neurona, se explicará el algoritmo del perceptrón multicapa que describe la estructura de interconexión de las neuronas. Las neuronas son colocadas en columnas o capas [17]. Todas las neuronas de una capa están conectadas a todas las neuronas de capas tanto previas como posteriores. La primera capa es la de entrada y no contiene ningún cálculo en ella, es una simple representación del vector de entrada. Las siguientes capas son conocidas como *hidden layers*. Por último, la última capa es la de salida donde se realizará la clasificación, puesto que será el objetivo que perseguimos, determinará a qué categoría corresponderá la entrada. El número de neuronas de la capa de salida corresponderá con el número de clases que contiene el problema. Por ejemplo, si se quiere clasificar imágenes de gatos frente a las de perros serán necesarias dos neuronas en la capa de salida.

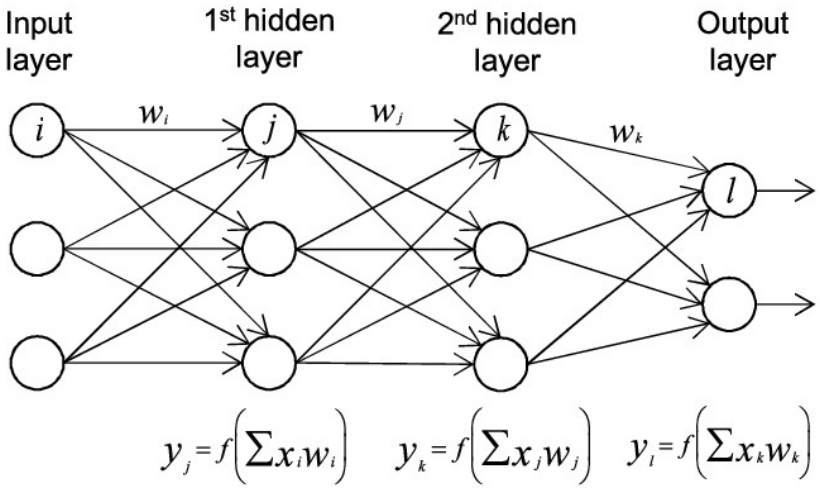

Figura 2.7: Algoritmo percetrón multicapa.

A diferencia con las neuronas de las capas intermedias en las que se recomienda la función *ReLU*. La última capa se utilizará la función *sigmoide* o *softmax* para la tarea de clasificación. *Sigmoide* es utilizada en problemas de clasificación binaria mientras que *softmax* se utiliza en problemas multiclasificación. Además, esta última, devuelve la posibilidad de la entrada de pertenecer a cada clase y selecciona la clase donde la probabilidad es más alta. En el apartado 2.3.1 se explicará de forma detallada todo esto.

2.1.4. Capa de pooling

Uno de los principales problemas de las convolucionales es el coste computacional. Si además le añadimos el número de *kernels* para cada entrada se tendrá una convolución con los K filtros diferentes. Por ejemplo, si tenemos una entrada con dimensiones 28x28x1 y el número de *kernels* es 64, el mapa de características de salida será 28x28x64 (Utilizando *zero-padding*). Por cada muestra de entrada tendremos 50.176 coeficientes. Si se intenta conectar a una capa totalmente conectada con 512 neuronas para obtener la clasificación

a la salida se tendría sobre los 25 millones de coeficientes en esa capa. Esto sería inviable ya que se necesitaría un poder computacional demasiado grande para entrenar una red tan grande. Por esto se utilizan técnicas para reducir el número de coeficientes del mapa de características.

Computacionalmente es importante reducir el número de coeficientes pero no es la única ventaja. También es fundamental para establecer una jerarquía espacial en la red. Es decir, al reducir el mapa de características capa a capa se logra que el modelo sea capaz aprender patrones globales más complejos. Las primeras capas se centran en los pequeños detalles como los ejes, contornos, etc. Y conforme se reduce la dimensionalidad del mapa de características se obtiene la totalidad de la información de la entrada.

Existen multitud de procedimientos para reducir el mapa de características. Una técnica sería utilizar la propiedad de la red convencional del *stride* para reducir el mapa de características de salida como se ha explicado en el apartado anterior. Otra técnica consistiría en la de *pooling*. Dicha técnica se fundamenta en extraer ventanas de las características de la entrada. Es muy similar a la convolución puesto que extrae regiones locales de la entrada, pero no lo hace mediante la operacional lineal de la convolución con un *kernel* sino mediante otra operación. No necesita de filtros simplemente se aplica una operación de forma local en la entrada. Principalmente existen dos variantes de *pooling*, según la operación que aplican podemos llamar *average pooling* o *max-pooling* [4]:

- ***Average pooling:*** calcula la media aritmética de la ventana de entrada.
- ***Max-pooling:*** calcula el máximo valor de la ventana de entrada.

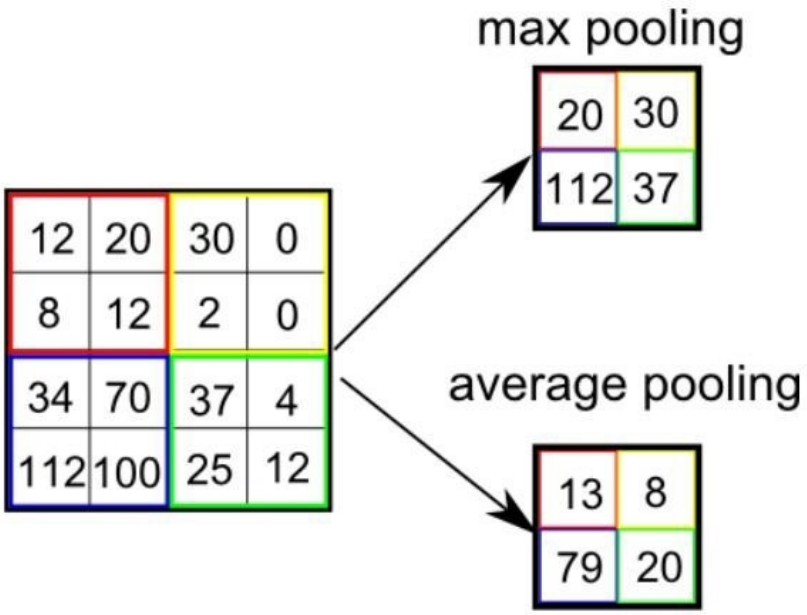

Figura 2.8: Ejemplo de cada técnica utilizando ventanas de 2x2 y stride 2.

Como se ilustra en la figura 2.8, es el mismo movimiento de ventana que una convolución pero calculando el máximo valor de una ventana o calculando la media aritmética.

Se suele utilizar *max-pooling* por sus mejores resultados frente a *average pooling* y la propiedad *stride*. El mapa de características tiende a codificar la presencia de patrones y, por tanto, es más informativo obtener la máxima presencia que la media aritmética de las

diferentes características. Tampoco es óptimo utilizar convoluciones con el parámetro de *stride*, porque de esta forma el movimiento de la convolución no es continuo y se pierde información que podría ser relevante. Por último, añadir que al utilizar los patrones más destacados de alguna manera esta operación actúa como supresor del ruido. La forma en la que se recomienda trabajar es utilizar convoluciones sin parámetro de *stride* y, seguidamente, emplear *max-pooling* para reducir la dimensionalidad.

2.2. Estructura de una red de clasificación

Tras analizar los principales bloques que componen un modelo de clasificación mediante redes convolucionales nos centraremos en la combinación de estos bloques para lograr construir las principales combinaciones de patrones de bloques utilizados en el área de la clasificación.

En la figuraa 2.9, se representa una de las estructuras más utilizadas. Consiste en una etapa inicial que tiene la función de extraer las características de las muestras de entrada y otra etapa final que realizará la tarea de clasificación.

La arquitectura típica de las redes convolucionales consiste en apilar capas convolucionales junto con la capa de *ReLU*, seguido de una operación de *pooling*. Este proceso se repite hasta reducir la dimensionalidad del mapa de características lo suficiente para que pueda ser conectado con el clasificador, que consistirá en una, o varias, capas totalmente conectadas con una función de activación *ReLU* y donde la última capa será una función *softmax* o *sigmoide*. Generalmente, utilizamos arquitecturas más profundas (con mayor número de capas *convnet*) cuando se tienen muchos datos con etiquetas para alimentar la red o cuando el problema es suficientemente complicado. Apilar múltiples capas de convoluciones junto con la función de activación antes de la capa de *poolling* permite a la red desarrollar mapas de características más complejas antes de que se reduzca la información tras la operación de *pooling*. Por último, para conectar los diferentes bloques utilizamos la capa de aplanamiento que convierte en mapa de características en un vector de entrada para capa totalmente conectada.

Algunos ejemplos de arquitecturas que utilizan este tipo de patrones como estructura son *AlexNet*, *VGGNet*, *GoogLeNet*, *ResNet*, etc [6, 34, 35, 36]. Como se comprueba en las publicaciones de los autores de éstas, se obtienen buenos resultados siguiendo este tipo de principios para la construcción de modelos.

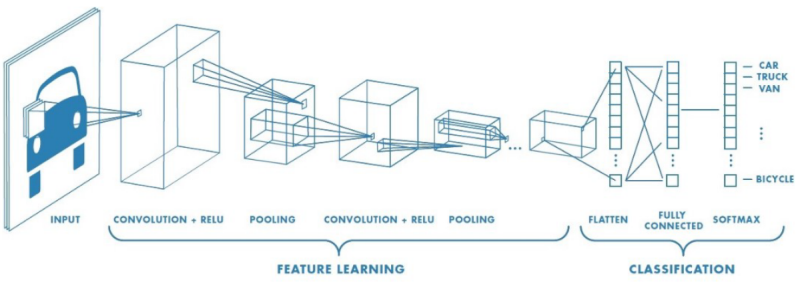

Figura 2.9: Estructura de una red convolucional clasificadora [7].

2.3. Algoritmos de aprendizaje

Una vez explicados los bloques que comprenden un modelo, se definirá el proceso de aprendizaje de la red. Existen dos mecanismos principales llamados la propagación hacia delante y la retropropagación [18], que son el motor del funcionamiento interno de las redes neuronales, permitiendo los procesos de clasificación y aprendizaje sean posibles [2].

La propagación hacia delante consiste en la generación de predicciones a la salida dados unos datos de entrada. En nuestro caso la red convolucional tiene como entrada una imagen que es procesada por todas las operaciones de las diferentes capas que conforman la red, para obtener a la salida la predicción de la categoría a la que pertenece la entrada. La información se transmite de forma secuencial entre las diferentes capas, hasta que se llega a la capa de salida donde se clasifica la entrada.

El mecanismo de retropropagación es el que realmente proporciona las cualidades de predicción de los modelos. Los modelos tienen que aprender a mapear las entradas para obtener las predicciones a la salida. Para ello, se tiene que encontrar los parámetros (pesos, kernel, vector de pendiente, etc) de la red adecuados para que a la salida obtener las predicciones deseadas. Lógicamente, cuanto más precisas las predicciones mejor funcionará el modelo, por tanto, el objetivo será el de optimizar los parámetros del modelo que nos garanticen unas predicciones lo más exactas posibles.

La retropropagación es el algoritmo encargado del proceso de optimización de los parámetros, en el que los parámetros óptimos serán aquellos en el que se minimice la función de pérdidas. La función de pérdidas se calcula comparando las predicciones del modelo con las etiquetas reales de los datos de entrada. El modelo de esta manera es capaz de saber si las predicciones son correctas o no lo son. Los algoritmos de optimización se encargarán de la minimización y actualización de los pesos, que serán transmitidos hacia toda la red utilizando la retropropagación. Para poder aplicarlo, las funciones de todo el modelo tienen que ser diferenciables puesto que es necesario poder aplicar la regla de la cadena. La regla de la cadena se emplea sobre el gradiente de la función de pérdidas y se propaga hacia atrás desde la función de pérdidas hasta llegar final de la red. Se calcula la contribución de cada parámetro ha tenido en la función de pérdidas.

Una red sigue las siguientes operaciones para su funcionamiento. Primeramente, se inicializan los parámetros de la red, se tiene que establecer un modelo de partida inicial. Los métodos de inicialización se pueden clasificar en dos: distribución normal y distribución uniforme. Algunos métodos de inicialización son la inicialización aleatoria tanto normal como uniforme, la inicialización de *Xavier* o *Glorot* [37] y la inicialización *Kaiming He* [38] siendo estos dos últimos los más recomendados generalmente. En ningún caso se debería utilizar la inicialización con ceros. Tras inicializar los parámetros de la red, el vector de entrada se propaga por la red hasta llegar a la última capa donde se obtiene la predicción del modelo, se ha aplicado el mecanismo de propagación hacia delante. Después, las predicciones se comparan con las etiquetas de los datos de entrada y se calculará la función de pérdidas. El error de la función de pérdidas se propaga hacia atrás mediante la retropropagación y se actualizan los parámetros con los algoritmos de optimización, para minimizar el valor de la función de pérdidas. Se repetirán estos pasos hasta encontrar un mínimo global o un mínimo local que proporcione la exactitud del modelo deseado.

A continuación, en los siguientes apartados se describirán los algoritmos de optimización más empleados para la búsqueda de mínimos de la función de pérdidas y la actualización de los parámetros de la red. Además se explicará de forma detallada los diferentes

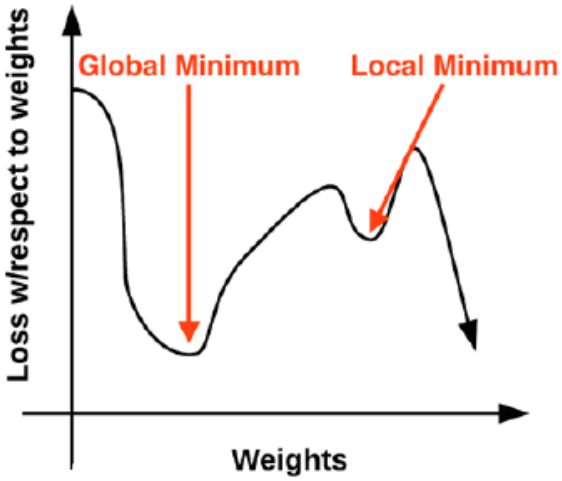

Figura 2.10: Función de pérdidas de dos dimensiones.

tipos y características de la función de pérdidas.

2.3.1. Función de pérdidas y coste

Como parte del algoritmo de optimización el error del estado actual del modelo tiene que estimarse de forma repetida. La función de pérdidas tiene un trabajo muy importante, puesto que resume el comportamiento del modelo en un solo número. Elegir la función de perdidas tiene una gran relevancia en el diseño del modelo, ya que permitirá al algoritmo de optimización encontrar un mínimo absoluto o un mínimo local lo suficientemente bueno.El tipo de función de pérdidas depende del problema a resolver y de la función de activación utilizada en la capa de salida [2].

En problemas de regresión la función de perdidas más utilizada es la del error medio al cuadrado (MSE) (2.11). Como capa de salida se tiene una sola neurona con la función de activación de unidad lineal. Se calcula el cuadrado de la diferencia entre el valor predicho $\hat{y}_i$ y el verdadero valor y_i.

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2 \tag{2.11}$$

Para problemas de clasificación la función más utilizada es la función de perdidas de entropía cruzada (2.12), donde t_i son las etiquetas verdaderas y s_i es la predicción del modelo para cada clase en C.

$$CE = -\sum_{i}^{C} t_i log(s_i) \tag{2.12}$$

Según el tipo de clasificación se implementará una función de activación diferente. En el fondo todos los problemas de clasificación utilizan la función de perdidas de cross-entropy, pero en función si es binaria o multiclase existen diferentes variantes. La figura 2.11, resume justo esto.

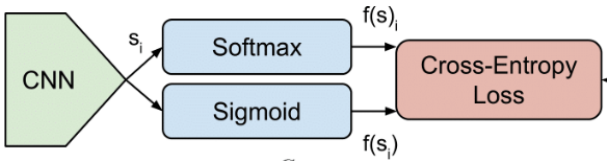

Figura 2.11: Diagrama variantes de la función de perdidas *cross-entropy*.

La clasificación binaria emplea la función *sigmoide*, que se calcula justo antes de la función de perdidas. Como se tiene $C' = 2$ y se asume que las dos clases C_1 y C_2 tienen forma $t_1 \in [0, 1]$. Esta variante se llama *binary cross-entropy* y tiene la siguiente expresión:

$$CE = -\sum_{i}^{C'=2} t_i log(s_i) = -t_1 log(s_1) - (1 - t_1) log(1 - s_1) \tag{2.13}$$

La clasificación multiclase emplea la función *softmax* (2.14), que se calcula justo antes de la función de perdidas (2.12). *Softmax* comprime el vector s en el rango $[0, 1]$ y el resultado de la suma de los elementos de s es uno. No puede ser implementada de forma independiente para cada s_i (depende de todos los elementos de s), por tanto dada una clase s_i la función *softmax* es:

$$f(s)_i = \frac{e^{s_i}}{\sum_C^j e^{s_j}} \tag{2.14}$$

Donde s_j corresponde con la predicción de la red para cada clase en C. Tras calcular la función *softmax* se aplica la función de perdidas *cross-entropy* (2.11), se obtiene la variante *categorical cross-entropy*:

$$CE = -\sum_{i}^{C} t_i log(\frac{e^{s_i}}{\sum_C^j e^{s_j}}) \tag{2.15}$$

De forma general, las etiquetas utilizadas suelen estar codificadas como *one-hot* (se explica en detalle en 3.4.1), de esta forma todos los elementos del vector de etiquetas t serán cero excepto el elemento $t_i = t_p$, por lo que solamente la clase positiva C_p será incluida en la función de perdidas. Si se descartan todos los elementos de la suma que son cero debido este formato de las etiquetas, se puede escribir la función de perdidas *categorical cross-entropy* como:

$$CE = -log(\frac{e^{s_p}}{\sum_j^C e^{s_j}}) \tag{2.16}$$

2.3.2. Descenso del gradiente

El gradiente descendiente es un método iterativo de optimización sobre la función de pérdidas. Es uno de los algoritmos de aprendizaje más utilizado en las redes neuronales. El principio de funcionamiento consiste en descender la pendiente de la superficie de la función de pérdidas (superficie de optimización). Con cada *epoch* (etapa del conjunto de entrenamiento) la función de pérdidas se minimizará y se actualizarán los pesos en

consecuencia. Este proceso se repite hasta encontrar el mínimo absoluto, o un mínimo local, que pueda considerarse como buen resultado.

La ecuación fundamental del gradiente descendiente se muestra en la ecuación 2.17. Los parámetros de modelo entero se denotan como θ. La tasa de aprendizaje η determina el escalón de cada iteración del movimiento hacia el mínimo. Es uno de los parámetros más importantes y se tiene que configurar de forma adecuada. Una tasa de aprendizaje demasiado elevada provoca que se salten los mínimos y una tasa demasiado pequeña que se tarde demasiado en converger. El gradiente ∇_θ es la derivada de la función de pérdidas y, por tanto, determina la dirección y el cambio de la función de pérdidas J. En función del sentido del gradiente, los parámetros θ serán actualizados para converger hacia el mínimo $\nabla_\theta J(\theta) = 0$.

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta J(\theta) \tag{2.17}$$

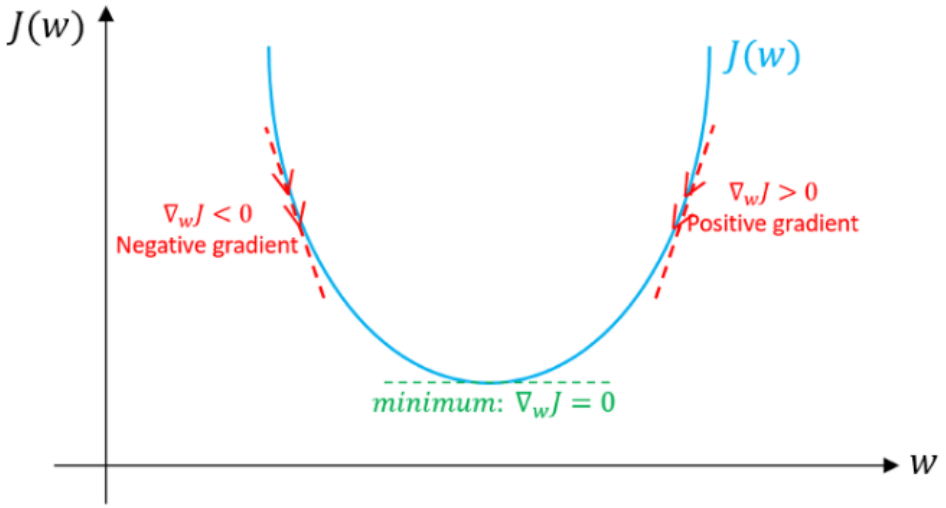

Figura 2.12: Función de perdidas en dos dimensiones. Se ejemplariza las direcciones de los gradientes para converger hacia el mínimo de la función.

2.3.3. Descenso del gradiente estocástico

El gradiente descendente tradicional calcula en cada iteración el gradiente de la función de pérdidas en todo el conjunto de datos. Al realizarse en todo el conjunto de datos es muy preciso pero muy costoso computacionalmente. La razón de su coste elevado es que requiere calcular las predicciones de todos los datos de entrenamiento antes de actualizar la matriz de parámetros. Otro problema de implementación es que los conjunto de datos voluminosos no caben en memoria, por ejemplo ImageNet [6] con más de 1.2 millones de imágenes sería imposible de implementar utilizando el algoritmo. Para evitar este problema existe una modificación del gradiente descendente más popular llamada descenso del gradiente estocástico (SGD) [2]. Esta modificación funciona de la misma forma que el gradiente descendente pero, el cálculo de $\nabla_\theta J(\theta)$ y la actualización de los parámetros θ se realiza en cada muestra de entrenamiento. Además otra versión del SGD trabaja con lotes pequeños llamada *mini-batch stochastic gradient descent*. Sigue el mismo sistema pero el proceso de cálculo y actualización se realiza para cada lote de datos (*mini-batch*). Un *mini-batch* es una pequeña división del conjunto de entrenamiento; las divisiones más

usuales son 32, 64, 128, 254 muestras por *batch*, estos valores suelen proporcionar buenos resultados [4]. Este método tiene menor coste computacional, mejor convergencia y, por lo general, mejores resultados que el método del gradiente descendente tradicional. Aunque, también tiene sus limitaciones, la tasa de aprendizaje η es fundamental ajustarse de forma apropiada. Utilizar un valor no adecuado puede provocar una convergencia muy lenta, la divergencia a un mínimo óptimo, estancarnos en un mínimo local no óptimo, etc. Para evitar estos problemas existen extensiones del SGD y otros algoritmos de optimización más avanzados que se crean con la finalidad de evitar estos problemas [2].

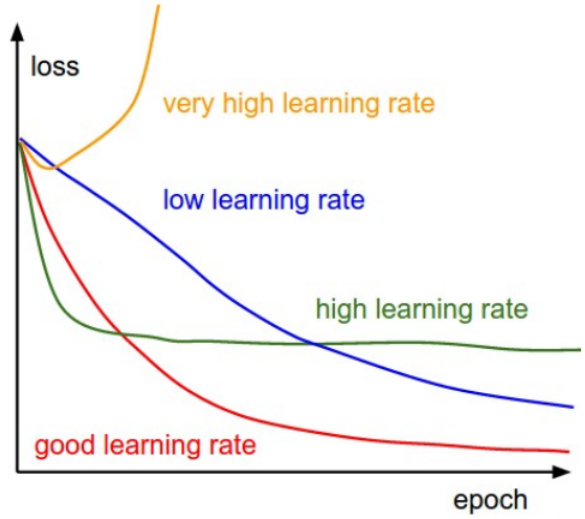

Figura 2.13: Comportamiento de la función de perdidas según la tasa de aprendizaje.

2.3.4. Extensiones del descenso del gradiente estocástico

- ***Step Decay:*** La tasa de entrenamiento empieza con valores más grandes y conforme se converge disminuye el valor de la tasa de aprendizaje. Se suele implementar de forma exponencial la disminución.
- ***Momentum:*** Ayuda al algoritmo SGD a acelerar en la dirección correcta disminuyendo oscilaciones no relevantes. Aumenta la fuerza de los pesos actualizados en los gradientes que mantiene la dirección y los disminuye cuando los gradientes cambian de dirección. Este fenómeno se consigue añadiendo una fracción γ del vector pasado a la actualización del vector presente[39].

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta) \tag{2.18}$$

$$\theta_{t+1} = \theta_t - v_t \tag{2.19}$$

- **Aceleración de *Nesterov:*** Se puede conceptualizar como una actualización correctiva del momentum que nos permite obtener una idea aproximada de donde estarán los parámetros después de ser actualizados. El gradiente de la aceleración de *Nesterov* contiene información sobre las posiciones futuras y, por tanto, incorpora un factor corrector para evitar converger demasiado rápido en un mínimo local [40].

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta - \gamma v_{t-1}) \tag{2.20}$$

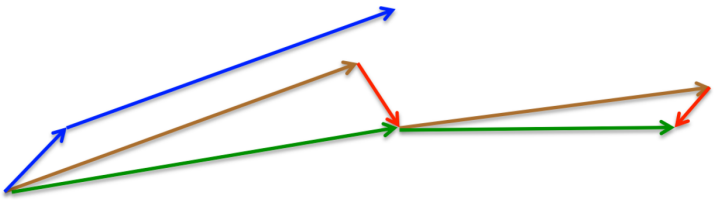

Figura 2.14: Comparación gráfica de los algoritmos *Momentum* y la aceleración de *Nesterov*. El momentum tradicional calcula el gradiente (pequeño vector azul) y luego tomamos un gran salto en la dirección del gradiente (gran vector azul). La aceleración de *Nesterov*, primero realiza un gran salto en la dirección del gradiente anterior, seguidamente mide el gradiente donde se tendría que haber ido y realiza la corrección hacia ese gradiente [8].

2.3.5. Algoritmos con tasa de aprendizaje adaptativa

A fin de lograr una convergencia más rápida, evitar las oscilaciones y estancarse en mínimos locales indeseables, la tasa de aprendizaje suele variar durante el entrenamiento, ya sea de acuerdo con un decaimiento progresivo de la tasa de aprendizaje o utilizando una tasa de aprendizaje adaptativa. Algoritmos como *AdaGrad*, *RMSProp* y *Adam* [41, 8, 42], son capaces de adaptar la tasa de aprendizaje de las siguientes formas:

- ***Adagrad:*** Adapta la tasa de aprendizaje en función del valor de los parámetros. Realiza pequeñas actualizaciones (pequeña tasa de aprendizaje) de parámetros con características frecuentes y actualizaciones grandes (tasa de aprendizaje grande) para parámetros con menos frecuencia. El algoritmo *Adagrad* utiliza una tasa de aprendizaje η diferente para cada parámetro θ_i y el gradiente g_t en cada iteración t. Por tanto, $g_{t,i}$ es la derivada parcial de la función objetivo para un parámetro θ_i en t:

$$g_{t,i} = \nabla_\theta J(\theta_{t,i}) \tag{2.21}$$

  El gradiente descendente para cada θ_i en cada paso de t sigue la siguiente ecuación:

$$\theta_{t+1,i} = \theta_{t,i} - \eta \cdot g_{t,i} \tag{2.22}$$

  La tasa de aprendizaje se modificará en cada paso t para cada parámetro θ_i basándose en el gradiente calculado anteriormente sobre el mismo parámetro θ_i. La matriz diagonal $G_{t,ii}$ es la suma de los gradientes al cuadrado. El término ϵ sirve para evitar dividir por cero siendo un valor muy pequeño.

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii+\epsilon}}} \cdot g_{t,i} \tag{2.23}$$

- ***RMSprop:*** Es una modificación del algoritmo *Adagrad*. Decrementa la tasa de aprendizaje de forma monótona. Además, en vez de acumular todos los gradientes al cuadrado pasados, restringe el número acumulado con un tamaño fijo w y realiza un promedio exponencial de todos los gradientes cuadrados pasados $E[g^2]_t$. Al no tener que acumular todos los gradientes se hace mucho más eficiente. Incorpora γ como la fracción igual que se hacía en el algoritmo de *momentum*. Los autores recomiendan utilizar valores de $\gamma = 0.9$ y $\eta = 0.001$.

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma) g_t^2 \tag{2.24}$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t \tag{2.25}$$

- ***Adam:*** Adapta la tasa de aprendizaje de cada parámetro utilizando el termino v_t y la función m_t de forma similar al *momentum*. Se calculan los promedios de decaimiento de los gradientes pasados m_t y el promedio de los gradientes pasados al cuadrado v_t con las siguientes ecuaciones:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t \tag{2.26}$$

$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \tag{2.27}$$

  Como los vectores m_t y v_t se inicializan con ceros, puede que, en instantes iniciales, estos tiendan a cero. Esto provocara que en las fases iniciales las predicciones no sean del todo precisas. Para evitar este sesgo los autores añadieron estas ecuaciones para contrarrestarlo:

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t} \tag{2.28}$$

$$\hat{v}_t = \frac{v_t}{1-\beta_2^t} \tag{2.29}$$

  Finalmente la actualización de parámetros se realiza como los algoritmos *RMSprop* y *Adagrad* :

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t + \epsilon}} \hat{m}_t \tag{2.30}$$

En la actualidad no existe un consenso a la hora de elegir el algoritmo de optimización. El estudio de Schaul [43] testeó diferentes algoritmos sobre una gran variedad de problemas; como resultado los algoritmos con tasa de aprendizaje adaptativa destacaron sobre los demás algoritmos aunque no obtuvieron resultados concluyentes sobre ninguno. Por tanto, se recomienda la utilización de cualquier algoritmo de aprendizaje como *AdaGrad*, *RMSProp*, *Adam*, pero dependerá del problema en cuestión y se utilizará el más óptimo en cada caso.

2.4. Métodos de generalización

El problema fundamental del aprendizaje de los modelos es la disputa entre la optimización y la generalización. Como se ha visto la optimización es el proceso por el que buscamos el mayor rendimiento con los datos de entrenamiento. Por el contrario, la capacidad de generalización, se define como el rendimiento del modelo en datos desconocidos para el modelo.

Al principio del proceso de entrenamiento estas métricas están correlacionadas, puesto que el modelo aprende los principales patrones del conjunto de entrenamiento. Todavía

existe un margen de mejora en la capacidad de aprendizaje del modelo mientras la función de pérdidas va disminuyendo, es decir, todavía quedan patrones relevantes que el modelo puede aprender del conjunto de datos de entrenamiento. Pero, a medida que la función de pérdidas se estanca y deja de disminuir, el modelo empieza a sobresaturar y, con ello, deja de aprender patrones más generales para centrarse en los específicos del conjunto de entrenamiento. Lo que no es recomendable puesto que perdemos capacidad de generalización. Se tiene que buscar el equilibrio perfecto entre optimización y capacidad del modelo de generalizar. Por tanto, si se quiere evitar el sobreentrenamiento, la mejor solución es proporcionar al modelo un conjunto de datos mucho mayor ya que cuantos más ejemplos vea el modelo más fácil será que generalice. Cuando no sea posible esto tenemos que aplicar técnicas y métodos para reducir la capacidad de memorización de la red. De esta forma al reducir el número de patrones que ésta es capaz de almacenar, la red se centrará en aprender los patrones más relevantes para poder reducir la función de pérdidas. Por tanto, si reducimos el número de capas de nuestra red y, con ello, el número de parámetros de ésta, se conseguirá una mayor generalización. Pero, por el contrario, si la capacidad de la red se reduce demasiado, la red no aprenderá los patrones necesarios. Tiene que existir un balance entre la dimensión del conjunto de datos y la profundidad del modelo.

En los próximos apartados se explicarán otros métodos más avanzados para lograr mejores resultados y que los modelos generalice de forma más efectiva. Estas técnicas también son conocidas como métodos de regularización.

2.4.1. Aumentación de datos

La mejor forma de aumentar la generalización del modelo es exponiéndolo a un número mayor de muestras. Pero el problema, muchas veces, viene por la falta de suficientes datos para poder entrenar el modelo de forma efectiva. La aumentación de datos es una técnica muy poderosa puesto que permite generar más datos de entrenamiento mediante datos ya existentes. Se realizan transformaciones de los datos, por ejemplo, en imágenes se realizan: rotaciones, transformaciones geométricas, traslaciones, etc [44]. De esta forma logramos que el modelo no vea los mismos datos varias veces. Se trata de una técnica computacionalmente exigente, puesto que al propio coste del entrenamiento de la red se tiene que sumar el procesado de las transformaciones. Esto aumenta de forma significativa el tiempo de entrenamiento y, por tanto, se debe tener en cuenta a la hora de implementarse.

Esta técnica funciona muy bien cuando el conjunto de datos que disponemos es muy limitado e imposible de entrenar con éste. Aunque como todo esta técnica tiene sus limitaciones, las imágenes que se crean son altamente dependientes de las que se utilizan y, por tanto, en cierto modo estamos limitados por esto. Esta técnica no podrá sustituir nunca a la de obtener nuevos datos reales. Este campo ha avanzado mucho y, en la actualidad, existen técnicas avanzadas para la generación de datos utilizando redes adversariales generativas (GAN) muy prometedoras [45].

2.4.2. Dropout

Es un tipo de regularización que previene el sobreentrenamiento incrementando la precisión del modelo en el conjunto de datos de prueba pero sacrificando la precisión del conjunto de entrenamiento. Es una de las técnicas de regularización más utilizadas y efectivas. Fue desarrollada por Geoff Hinton [46] y consiste en desconectar de forma

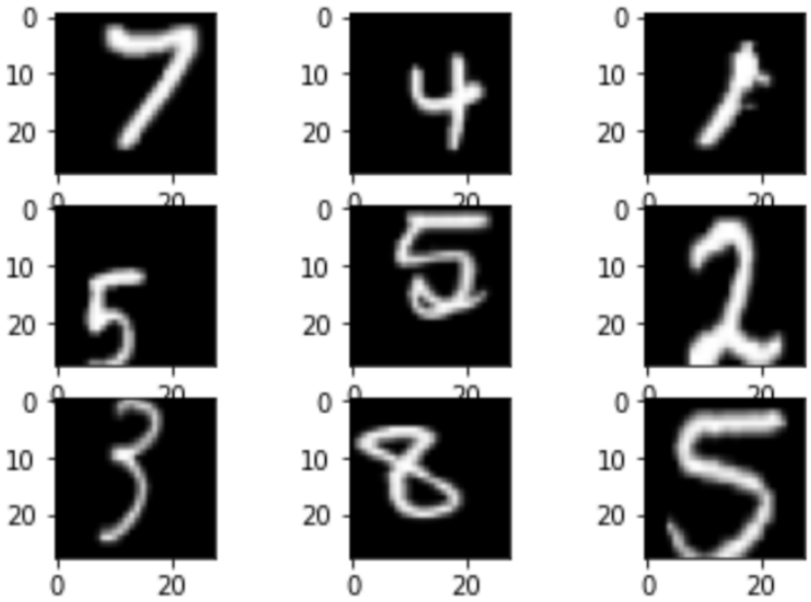

Figura 2.15: Ejemplo de procesado de imágenes. Se ha implementado rotaciones, traslaciones, escalado e inversión de forma aleatoria.

aleatoria neuronas durante la fase de entrenamiento. De esta forma, en cada época las neuronas tienen una probabilidad p de quedarse conectadas o $1-p$ de desconectarse. Esta técnica es muy efectiva por las siguientes razones: se reduce de forma aleatoria el número de neuronas activas y, por tanto, se está reduciendo la capacidad de la red [47]. Además, se evita el fenómeno de coadaptación al disminuir la dependencia entre neuronas. Como se desconectan las neuronas de forma aleatoria dejan de cooperar entre ellas y se evitan dependencias en unidades superiores. Y, por último, se fuerza a las neuronas a tomar más o menos responsabilidades de forma aleatoria. Como el número de neuronas activas en cada momento es aleatorio y, por tanto, en cada momento el rol de las neuronas cambia.

Por norma general se suele utilizar entre las capas totalmente conectadas en el bloque de clasificación de la arquitectura, con una probabilidad de alrededor $p = 0.5$. Aunque también utilizado, pero con menos frecuencia, entre las capas de convolución y de *pooling* a veces se suele utilizar *dropout* con una probabilidad de $p = [0.1-0.25]$. Por la conectividad local de la capa convolucional esto suele ser menos efectivo [5].

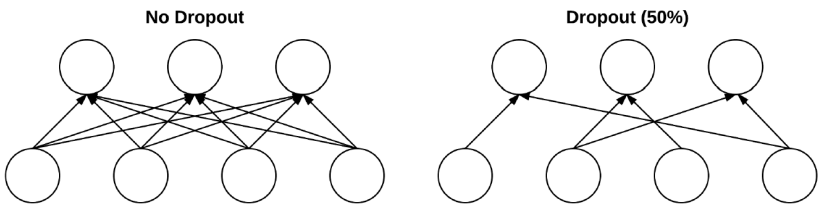

Figura 2.16: Se tiene una red totalmente conectada formada por dos capas de neuronas. A la **izquierda** la red sin *dropout* y a la **derecha** con *dropout* del 50 %.

2.4.3. Batch normalization

Fue desarrollada por Loffe y Szgedy en su publicación *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift* [48]. Este método consiste en normalizar la salida de la capa anterior restando la media del lote (*batch*) y dividiendo por la desviación estándar del lote. Si consideramos x_i como el lote, entonces se calcula $\hat{x}_i$ mediante la siguiente ecuación:

$$\hat{x}_i = \frac{x_i - \mu_\beta}{\sqrt{\sigma_\beta^2 + \varepsilon}} \tag{2.31}$$

El parámetro ε es necesario para evitar realizar la raíz cuadrada de zero. Se debe establecer como un valor positivo muy pequeño, por ejemplo *1e-6*. Durante la fase de entrenamiento se calculan μ_β y σ_β sobre un lote β mediante las siguientes ecuaciones:

$$\mu_\beta = \frac{1}{m}\sum_{i=1}^{m} x_i \tag{2.32}$$

$$\sigma_\beta^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_\beta)^2 \tag{2.33}$$

Como se observa μ_β será la media del lote y σ_β la desviación estándar del lote también. Además, en la fase de testeo del modelo (utilizamos el conjunto de datos específico para esto) se reemplaza μ_β y σ_β con la media de éstas calculada durante la fase de entrenamiento. De esta forma nos aseguramos obtener resultados precisos sin depender de los valores de μ_β y σ_β calculados en el último lote de la fase de entrenamiento.

Los principales beneficios de utilizar *batch normalization* es que reduce el número de veces que la red tiene que ver el conjunto de datos de entrenamiento de forma completa para entrenarla. También estabiliza el proceso de entrenamiento, admitiendo mayor variedad de valores a la tasa de aprendizaje que es fundamental establecer de forma adecuada. Esto se debe a que la función de pérdidas deja de tener zonas abruptas y se suaviza. De forma general facilita ajustar los parámetros de forma óptima más rápidamente. Por ejemplo, la red es menos dependiente del algoritmo de inicialización de los pesos, siendo menos dependiente a comienzos no óptimos. Actúa como método de regularización reduciendo el sobreentrenamiento y mejorando la precisión de la red. Por contra, el coste computacional de la fase de entrenamiento aumenta de forma notable, aunque se reduce el número de muestras necesarias para alimentar al modelo. Esto es debido a que se añade el coste de las operaciones de normalización que es mayor que el ahorrado con la reducción de las muestras.

Implementar el método de normalización de los lotes es un tema polémico. La publicación original Loffe y Szeged [48], recomienda aplicar la normalización justo antes de la función de activación. Pero si se ha analizado desde el punto de vista estadístico, tiene más sentido que se implemente justo después de la no linealidad. La justificación es simple, imaginemos que se implementa la normalización justo a las características de salida de una red convolucional, algunas de esas características tienen valor negativo, la no linealidad como *ReLU* las convertirá en valores iguales a cero, por lo tanto, se añaden valores negativos a la normalización que, cuando se procesen en función de la activación, serán eliminados. Estamos influyendo la normalización con valores negativos que no van a pasar

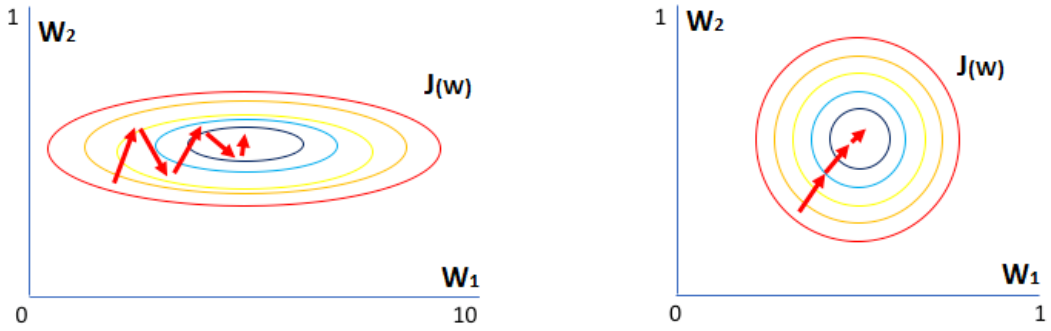

Figura 2.17: Representación de una fución de perdidas. A la **izquierda** no se ha aplicado *Batch Normalization* y en la **derecha** sí. Al aplicar *Batch Normalization* se observa como el salto entre zonas es uniforme y también cómo se reduce la influencia del valor inicial de los pesos, la distancia es más o menos la misma.

a la siguiente capa de convolución y con esto se pierde el beneficio de la normalización, al tener el conjunto de entradas centradas en cero. Aunque todavía existe debate en esto, muchos autores como *Fraçois Chollet* recomiendan utilizar la capa de *batch normalization* después de la función de activación [49, 50].

Capítulo 3

Metodologia

En el presente capítulo se describirán las herramientas y las metodologías utilizadas para el desarrollo de la parte experimental del trabajo. Primeramente se presentarán las herramientas utilizadas, en concreto, el *hardware* y el *software* necesario. Seguidamente, se explicará el conjunto de datos de imágenes que se emplearán para la resolución del problema junto a la metodología de evaluación de los resultados y el protocolo de división del total de los datos para poder llevarla a cabo. Y, finalmente, se procesarán los datos y se expondrá detalladamente la implementación del modelo junto con los resultados obtenidos y el análisis de estos.

3.1. Herramientas y entorno de desarrollo.

El proyecto se ha llevado a cabo en un ordenador con un procesador *Itel*(R) *Core*(TM) i7-4510U *CPU* @ 2.00 Ghz 2.60 Ghz con *turboboost* hasta los 3.10 Ghz, con 16 *GBytes* de memoria *RAM DDR3*. La tarjeta gráfica utilizada ha sido *NVIDIA GeForce* 840M con 2 *GBytes* de memoria *RAM* virtual. Como sistema operativo se ha usado *Windows* 10 de 64 *bit*.

El lenguaje de programación utilizado ha sido *Python* 3.7. El principal motivo de la utilización de este lenguaje es por el gran número de librerías enfocadas al aprendizaje profundo. En concreto, se implementará el modelo por medio del *framework* de *Keras*, como se demuestra en la figura 3.1. Se trata de una *API* de alto nivel que nos permite poner en funcionamiento modelos de aprendizaje profundo de forma rápida y sencilla. *Keras* trabaja con bloques de nivel alto, pero no realiza las operaciones. Para eso recurre a librerías optimizadas para la operación tensores y, de todas las compatibles con keras nosotros emplearemos *Tensorflow* por ser la más utilizada de todas las compatibles con *Keras*. Las librerías de tensores se pueden ejecutar tanto en la *CPU* o en la *GPU*, siempre es más recomendable ejecutar los modelos de aprendizaje profundo sobre la *GPU* frente a la *CPU*. Como se ha visto en el capítulo 2, la mayoría de operaciones que tiene que realizar las redes profundas son operaciones de álgebra lineal, ejecutadas mucho más rápido con la *GPU*. Pero para qué tensorflow se pueda comunicar con la *GPU*, es necesario instalar *NVIDIA CUDA Deep Network library* (*cuDNN*) y *Comute Unified Device Architecture* (*CUDA*). Además, se han utilizado otras librerías como *Numpy* para calculos cientificos y *matplotlib* para la generación de gráficas y visualizaciones del proyecto.

Para la instalación de las librerías se ha optado por la utilización de un entorno virtual

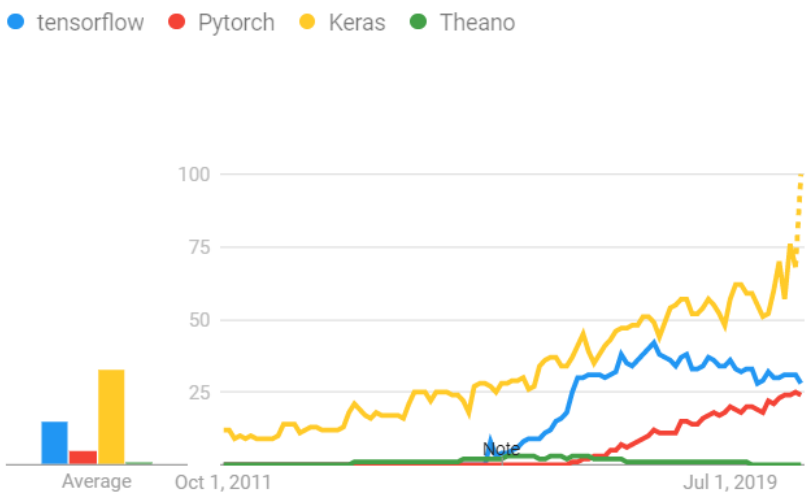

Figura 3.1: Frecuencia de aparición de las palabras *Tensorflow*, *Pytorch*, *Keras*, *Theano* en *Google Analytics*.

frente a una instalación local. Puesto que, para instalar *Keras* y *Tensorflow* son necesarias la instalación de otros módulos. Los paquetes necesarios normalmente requieren de versiones específicas. Por ejemplo, para instalar *Tensorflow* 1.15.0 es necesario haber instalado *Numpy* 1.18.2. Pero qué ocurre si en la misma instalación de *Python* necesitamos en otra aplicación utilizar la versión más reciente de *Numpy*. Se tendría que estar cambiando de la instalación del paquete cada vez que cambiáramos de aplicación, lo que es inviable puesto que, puede llevar multitud de dependencias. Este problema se soluciona rápidamente realizando una instalación de las librerías de cada proyecto en un entorno virtual diferente, aislando las instalaciones del resto.

Con el módulo interno de *Python venv*, se crea y se activa un entorno virtual. Una vez activado, se pueden realizar instalaciones y desinstalaciones de paquetes de forma tradicional en *Python* mediante *pip*. Además, la creación de un archivo de texto con los requerimientos de librerías del proyecto es muy recomendable. De esta forma, se puede compartir el código e instalar los paquetes necesarios para ejecutarlo de forma rápida y efectiva.

Como entorno de desarrollo para el desarrollo y mantenimiento del código se ha utilizado la siguiente configuración:

- ***Sublime text 3:*** Como editor de textos (*IDE*) se ha elegido por su simplicidad y por la gran versatilidad que proporciona para poder configurarlo a nuestro gusto. Mediante la instalación de paquetes es posible aumentar las funcionalidades del entorno. En nuestro caso en concreto hemos instalado los paquetes de *Anaconda* y *Virtualenv*. El primero de los paquetes proporciona las ventajas como el autocompletado del código, la coloración del código, etc. Las librerías que se emplean están soportadas por *Anaconda* lo que es muy útil puesto que facilita nuestra labor de programación. Además, proporciona autocompletado con la herramienta *PEP8*, extremadamente necesario para estructurar el código de forma adecuada y siguiendo unos estándares recomendados por la comunidad de programadores de *Python*. El otro paquete, tiene como función principal ejecutar nuestro entorno virtual en la propia consola del *IDE* y, de esta forma, no ejecutamos los *scripts* de *Python* en una consola externa. Al poder ejecutar en el propio *IDE* se automatiza el proceso

de ejecución y con el atajo de teclado *Ctrl+Mayús+B* se ejecutan los programas. Lo que nos proporciona mucha comodidad.

- ***Git:*** Para el control de versiones del desarrollo se ha utilizado "*Git*". Es el más utilizado por la comunidad de desarrolladores y, por tanto, se implementa en cualquier sistema. Como servicio de almacenamiento se ha utilizado la plataforma de *GitHub*, de esta forma se tiene un servidor en remoto en el que se pueden ir almacenando las diferentes versiones del desarrollo. De esta forma, si fuese necesario se puede colaborar de forma fácil y efectiva con otros programadores y tenemos un historial del proyecto entero en la nube. EL *link* del servidor de proyecto se puede encontrar en el siguiente enlace: https://github.com/carseven/Convolutional-Neural-Network-MNIST.

- ***Git Extension:*** Aunque se puede controlar "*Git*" mediante comandos de consola, por comodidad se utilizará la interfaz gráfica de "*Git Extension*" que permite administrar el control de versiones de nuestro código de una forma más intuitiva.

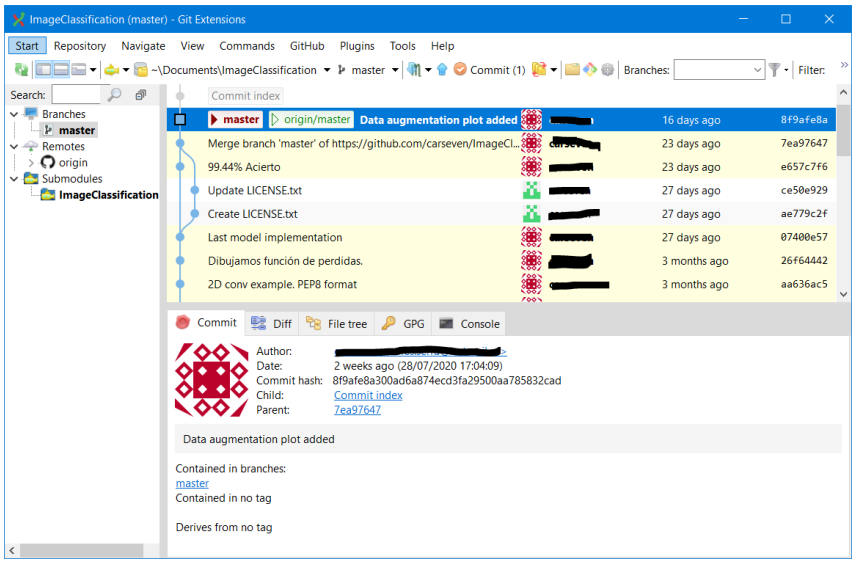

Figura 3.2: Interfaz gráfica del software *Git Extension*.

3.2. Descripción y análisis del conjunto de datos

El conjunto de datos que se emplea para el desarrollo del trabajo será el del *MNIST* [51]. *Yann LeCun* y *Corinna Cortes* tienen los derechos de autor de este conjunto de datos y este se puede utilizar bajo las condiciones de la licencia *Creative Commons Attribution-Share Alike 3.0*. Se trata de una versión modificada del conjunto de datos original *NIST* (*National Institute of Standards and Technology*). *NIST* fue dividido en un conjunto de entrenamiento y otro de prueba, el problema era que el conjunto de entrenamiento era mucho más fácil comparado con el de prueba. Los datos de entrenamiento fueron recolectados de entre los empleados de una oficina de censo, sin embargo, el conjunto de prueba se recolectó de entre los estudiantes de secundaria. Esto provocó que la dificultad de los datos no estuviese relacionada. Por estos motivos, fue necesario crear un nuevo conjunto a partir del mezclando de los datos originales de *NIST*. Además, las imágenes se han preprocesado para que sea más fácil su implementación, por ejemplo, se ha normalizado el tamaño, se ha establecido un tamaño fijo, etc.

La base de datos de *MNIST* tiene una importancia histórica en el área del aprendizaje profundo, se suele utilizar a la hora de realizar pruebas experimentales, prueba de nuevas técnicas, para comparar el comportamiento de diferentes modelos, etc [2].

El conjunto de datos está formado por dígitos manuscritos, en concreto tenemos un total de diez clases que van desde el número cero hasta el nueve. Está formado por 60.000 imágenes de entrenamiento y por 10.000 imágenes de prueba. Cada vector de características tiene dimensión 784, correspondiente a imágenes en escala de grises 28x28x1. La intensidad de los pixeles tiene valores con un rango de 0 a 255 y, por tanto, se almacenan como enteros sin signo (*uint8* en *Python*). Todos los dígitos son representados en blanco, las sombras se trazan mediante colores grises y todas ellas se muestran sobre un fondo de color negro.

El objetivo del modelo a implementar será clasificar a qué clase pertenecen los dígitos manuscritos, se trata de un problema de clasificación multiclase. Pero antes de diseñar el modelo como se verá en los próximos apartados, se analizarán los datos para conocer mejor la naturaleza de estos.

Para obtener los datos se puede hacer desde la página web oficial [51], donde se pueden descargar las imágenes y las etiquetas necesarias para la clasificación. Aunque la librería *Keras* nos proporciona la función "*mnist.loaddata()*" que nos devuelve una tupla con arrays de *Numpy* con las 60.000 imágenes de entrenamiento con array de etiquetas correspondientes y con las 10.000 imágenes de prueba también con sus array de etiquetas. Estos datos se almacenarán en caché de forma local. Se hace uso de esta función por comodidad al cargar los datos y no se tiene que emplear otra librería.

Tras cargar los datos, se analizarán los datos. Primeramente, se hará uso de las librerías de *Matplotlib*, *Numpy* y la librería propia de *Python Random*, para generar un gráfico de muestras aleatorias de cada clase. Se generarán dos gráficos, uno para los datos de entrenamiento y otro para los datos de prueba. Las muestras de ambas figuras confirman que ambos grupos de datos son muy parecidos en cuanto dificultad se refiere, lo que es fundamental para que sea válido el proceso de entrenamiento frente al de prueba. Destacar la diferente variedad de muestras en una misma clase y siendo alguna de ellas de bastante más dificultad que otras, esto podría ser debido a los dos grupos utilizados para recolectar las muestras. Además, también se aprecia similitud de muestras entre clases, que puede suponer un problema para el modelo, porque producirá falsas predicciones al poder confundir entre clases. Por ejemplo, un nueve puede llegar a confundirse con un cuatro si no está bien dibujados.

Con las mismas librerías se han creado las gráficas de frecuencia de las diferentes clases. En la figura 3.5, se visualiza la frecuencia de entrenamiento por clase del conjunto de entrenamiento. A primera vista, se observa que existe una ligera descompensación entre clases, no están perfectamente balanceadas. La clase del número uno es la que más muestras contiene en total, siendo el 11.2 % del total de muestras de entrenamiento. Por el contrario, la clase del número cinco es la que menos muestras tiene, con un 9 % del total.

En el conjunto de prueba también ocurre esto, como se ve en la figura 3.4. Siendo el desequilibrio muy parecido al del conjunto de entrenamiento. Es importante que para que el modelo no aprenda de forma sesgada, la frecuencia de datos entre clases sea muy parecida. En el caso del dato *MNIST*, el desequilibrio que presenta es mínimo y no supondrá ningún problema de discriminación de clases. Además, otro aspecto fundamental, es que la distribución de las clases sea la misma para que sean coherentes entre ellos y

garantizar que el modelo aprende de la misma naturaleza con los que será testado. No tendría sentido que el modelo aprendiera sobre datos donde luego no será testado; los resultados no serían concluyentes [4].

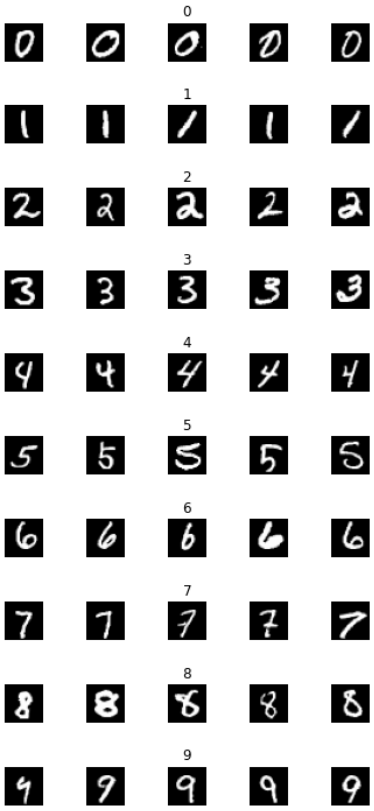

Figura 3.3: Muestras aleatorias de cada clase del conjunto de datos de entrenamiento.

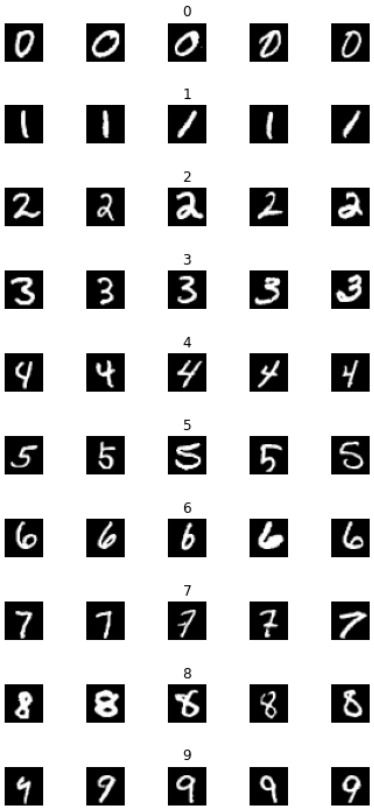

Figura 3.4: Muestras aleatorias de cada clase del conjunto de datos de *test*.

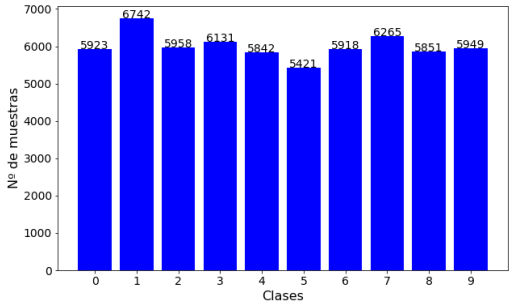

Figura 3.5: Número de muestras por clase del conjunto de entrenamiento.

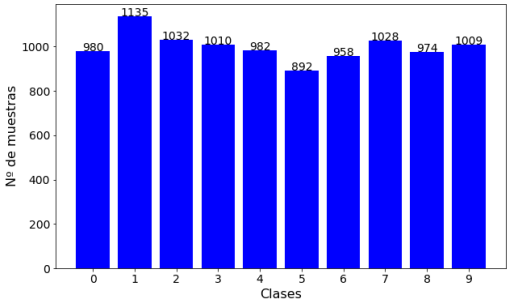

Figura 3.6: Número de muestras por clase del conjunto de *test*.

3.3. Metodología de la medición de los resultados

En el apartado 3.5 se presentarán los resultados obtenidos con el modelo de clasificación propuesto. Por tanto, para cuantificar los resultados y poder controlar cómo de bien funciona el modelo, primero hay que especificar qué significa que el comportamiento del modelo es el adecuado para nuestra meta.

Existen cuatro posibles resultados por clase:

- **Positivo verdadero (TP):** El clasificador predice correctamente la clase positiva como positiva.
- **Negativo verdadero (FP):** El clasificador predice correctamente la clase negativa como negativa.
- **Falso positivo (FP):** El clasificador predice incorrectamente la clase negativa como positiva.
- **Falso negativo (FN):** El clasificador predice incorrectamente la clase positiva como negativa.

Como criterio de evaluación del modelo se utilizará la matriz de confusión [52]. Es una herramienta muy útil puesto que nos permite visualizar el funcionamiento del modelo de forma eficaz. En las columnas se representan las predicciones de cada clase y en las filas la clase real. En la figura 3.7, se observa como la matriz de confusión ilustra de forma muy efectiva las diferentes soluciones. En concreto, una matriz enfocada a un problema de clasificación multiclase.

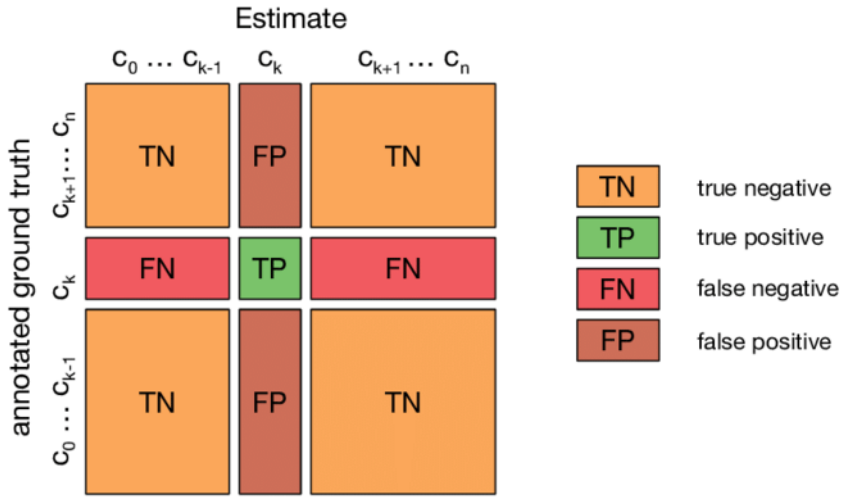

Figura 3.7: Matriz de confusión de una clasificaciín multi-clase [9].

De la matriz de confusión se puede utilizar las siguientes métricas para la medición del rendimiento del modelo [53]:

- **Exactitud:** Es la métrica más utilizada por su sencillez. Se calcula el cociente entre las predicciones clasificadas correctamente con el total de predicciones. Como es lógico cuanto mayor es mejor rendimiento tendrá el modelo. Por contra, mencionar que proporciona información limitada sobre el comportamiento, es demasiado general y no proporciona información específica de cada clase. En la clasificación

multiclase, esta métrica se calcula para cada clase y luego se realiza la media de todas las clases para obtener la exactitud total del modelo.

$$E = \frac{TP + TN}{TP + FP + TN + FN} \tag{3.1}$$

También se puede ver como la inexactitud:

$$I = (1 - E) = \frac{FP + FN}{TP + FP + TN + FN} \tag{3.2}$$

- **Precisión:** Se centra en el rendimiento del modelo en clasificar correctamente una clase particular. Mide la proporción de predicciones positivas (TP) frente al total de las predicciones de la clase (TP+FP), incluyendo las predicciones de la clase erróneas (FP). Cuanto mayor sea este valor mejor rendimiento del modelo en esa clase, en concreto la unidad es el valor mayor precisión y ocurre cuando el número de falso positivos es cero.

$$P = \frac{TP}{TP + FP} \tag{3.3}$$

- **Probabilidad de detección o *Recall*:** Cuantifica el número de predicciones correctas sobre el total de las predicciones realizadas para esa clase. Indica las predicciones positivas frente al total que pertenecía a la clase.

$$R = \frac{TP}{TP + FN} \tag{3.4}$$

- ***F1-Score*:** Combina las métricas de precisión y *recall* mediante la media armónica. De esta forma, se logra una combinación de las dos en una sola métrica y se compara más intuitivamente el rendimiento de varios modelos.

$$F1 = 2 \cdot \frac{Recall \cdot Precision}{Recall + Precision} \tag{3.5}$$

Existen multitud más de métricas, por nombrar algunas más: *ROC curve*, *AUC* (*Area Under Curve*), etc. Se utilizarán las descritas anteriormente más adelante en el apartado 3.5.

3.3.1. Protocolo de evaluación

El procedimiento tradicional es la división del total de los datos en dos conjuntos, pues sería impropio utilizar los mismos datos para entrenar el modelo y cuantificar su desempeño. Por lo tanto, se reserva un conjunto independiente (*test*) para baremar el éxito del modelo de forma no sesgada. Junto con la fase de entrenamiento y testeo, el desarrollo del modelo siempre implica un proceso de configuración de parámetros, por ejemplo, el número de capas del modelo, el algoritmo de aprendizaje, la función de activación, etc. Estos parámetros tienen el nombre de hiperparámetros y no se tienen que confundir con los parámetros que aprende el modelo. Para ajustar los hiperparámetros se emplean los datos de validación. Principalmente, se encuentran las siguientes aproximaciones o enfoques para dividir el conjunto de datos en entrenamiento, validación y testeo: *hold-out*, *k-fold*

y *k-fold* iterativa. El método empleado elegido ha sido el de validación *hold-out* por su sencillez y fácil implementación. Este enfoque consiste en la división tradicional explicada antes y utilizar el conjunto de testeo para la validación y prueba del modelo. Esta técnica es muy utilizada y extendida tiene sus limitaciones, al compartir la fase de testeo con la de prueba presentará fugas de información. Esto es debido a que se está ajustando el modelo utilizando con los datos que luego se testea, exponiendo el modelo a menos datos no vistos y, con esto, reduciendo la capacidad de generalización. Cuando se dispone de una cantidad de datos elevados como es nuestro caso, este efecto se reduce, aunque se tiene que tener en cuenta igualmente. Añadir que, para obtener mejores rendimientos, se recomienda las demás técnicas mencionadas, puesto que independizan los tres procesos con datos independientes.

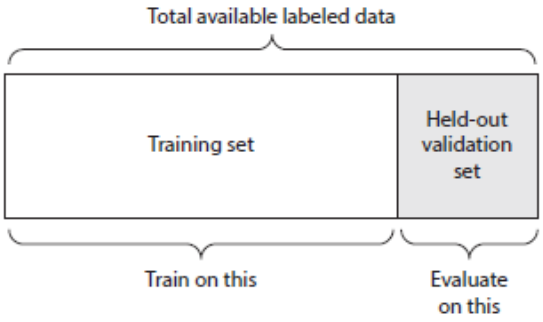

Figura 3.8: Esquema de división *Hold-out* [4].

3.4. Modelo de clasificación propuesto

En este apartado se describirán de forma detallada los modelos desarrollados para atacar el problema de clasificación multiclase del conjunto de datos *MNSIT*. Primero se mencionara el procesado realizado a los datos para que los modelos puedan trabajar con ellos. Seguidamente, se implementarán los modelos de redes convolucionales y se explicará el proceso de optimización realizado para mejorar el rendimiento o la capacidad de generalizar de estos.

3.4.1. Preprocesado de los datos

Tras analizar la naturaleza de los datos, como se ha visto en el apartado 3.2, se procesarán y se formatean para poder ser introducidos en el modelo y además, mejorar la eficiencia del modelo.

El procesado que ha sido aplicado es muy básico y típico en redes de estas características. Se ha aplicado tanto a las imágenes como a las etiquetas. La información de entrada de las redes tiene que estar vectorizada, en concreto, en forma de tensores. Las imágenes de *MNIST* son matrices y, por tanto, ya tiene forma vectorizada de tensores de dos dimensiones (28,28). Aunque para poder trabajar con la capa convolucional (*conv2d*) de la librería de *Keras*, se tiene que transformar a un tensor de tres dimensiones (28,28,1). Además, el tipo de variable de las imágenes se ha transformado de *uint8* a *float32*, debido a que se ha reducido el valor de las intensidades de los píxeles con un factor de 255,

consiguiendo que esta intensidad tenga un rango entre 1 y 0. El beneficio del escalado es agilizar y facilitar al modelo el aprendizaje mediante algoritmos como la retropropagación [54]. Destacar que como todas las imágenes presentan el mismo rango (al estar codificadas en escala de grises), son homogéneas y por tanto, no ha sido necesario ajustar las desviaciones estándar de estas.

Las etiquetas han tenido que procesarse para poder ser introducidas en el modelo. Para ello, se han vectorizado. Existen dos técnicas principales para codificar etiquetas en forma de vector: *label encoding* y codificación de categóricas o *one-hot encoding*, donde se transforma cada clase mediante unos y ceros de forma binaria, como se muestra en la siguiente ecuación:

$$\begin{bmatrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \end{bmatrix} \mapsto \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.6}$$

Se podría pensar que utilizar un vector con las clases de cada dígito de forma directa, es más intuitivo que "*one-hot encoding*", pero sería un error. Si se vectorizan las etiquetas directamente, el modelo podría confundirse pensando que existe algún orden en los datos y no sería beneficioso.

3.4.2. Implementación del modelo

La estructura del modelo inicial está ilustrada en la figura 3.9. Es una red convolucional poco profunda configurada desde cero que está formada de tres bloques convolucionales. Cada bloque contiene una capa convencional y una capa de *max-pooling*. Menos la última que sólo contiene una sola capa convolucional. Todas las capas convolucionales se han configurado igual, con filtros de de 3x3 dimensiones, *stride* 1x1, función de activación $ReLU$ y con *padding* para mantener la dimensionalidad del mapa de características, puesto que se utiliza la capa de *max-pooling* con filtros 2x2 para reducir la dimensionalidad. El número de filtros del primer bloque es de 32 y los dos últimos bloques se aumentó a 64. El método de inicialización es el de *Xavier* para todos los bloques [37]. Los bloques de convoluciones están conectados a una red multicapa de tipo perceptrón que, clasificará las características extraídas de éstos. Primero, se aplica una capa de allanamiento para convertir las dimensiones del mapa de características en un vector y poder ser clasificado mediante las dos capas totalmente conectadas. La primera capa totalmente conectada contiene 64 neuronas y una función de activación $ReLU$, la última capa es la encargada de la clasificación y, por tanto, tiene 10 neuronas (una por clase) y función de activación *softmax*. El algoritmo de optimización elegido ha sido el de *Adam* [42] con una tasa de aprendizaje de 0,001 y la función de pérdidas empleada ha sido *categorical cross-entropy* Además, se ha entrenado durante 20 épocas con un tamaño de lotes de 256. En total la red contiene 257.802 parámetros de los cuales 257.482 son entrenables y costó unos 10

minutos en completarse la fase de entrenamiento con la potencia computacional descrita en el apartado 3.1.

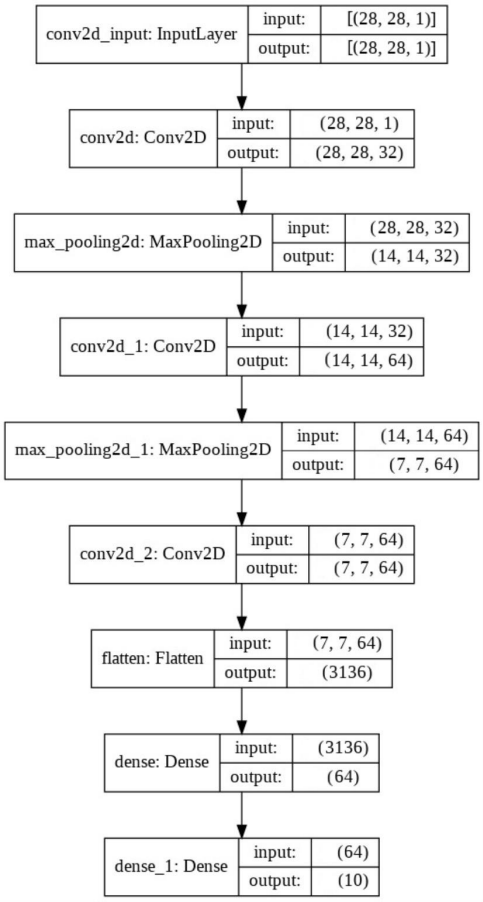

Figura 3.9: Estructura del primer modelo implementado.

La red se ha construido poco a poco aumentando el número de capas y parámetros para aumentar la capacidad de ésta de forma gradual. Al principio del entrenamiento, la optimización y la generalización están correlacionadas. Conforme disminuye la función de pérdidas en datos de entrenamiento también disminuye en los datos de *test*, como se observa en la figura 3.10 y 3.11. Hasta un poco antes de la 5ª época la relación se mantiene, pero cuando se sobrepasa este punto, la función de pérdidas del conjunto entrenamiento continúa reduciéndose, mientras que los datos de validación dejan de mejorar y se estancan. Este fenómeno es un claro ejemplo de sobreentrenamiento. El modelo aprende a optimizar el conjunto de entrenamiento, aprendiendo patrones específicos, provocando la reducción de la capacidad de generalización ante datos no vistos.

Para solucionar este problema, se incorporaron a la red diferentes técnicas de regularización (Apartado 2.4) para fomentar que la red generalice y evitar la optimización de patrones no relevantes. Para ello, se ha utilizado la técnica de aumentar los datos, donde se ha procesado mediante técnicas como desplazamientos, *zoom*, ligeras rotaciones de menos de 90º (ya que clases como el nueve se podría confundir con la clase del seis si la rotación es demasiado pronunciada) y añadir ruido gaussiano. Estas técnicas son las que mejores resultados han proporcionado puesto que otras dificultan demasiado el proceso de entrenamiento obteniendo peores resultados. Este proceso se aplica a todas las imágenes de forma aleatoria antes de introducirse en el modelo y, de esta forma, el modelo en fase de entrenamiento tiene datos que nunca verán los mismos datos dos veces. Lógicamente,

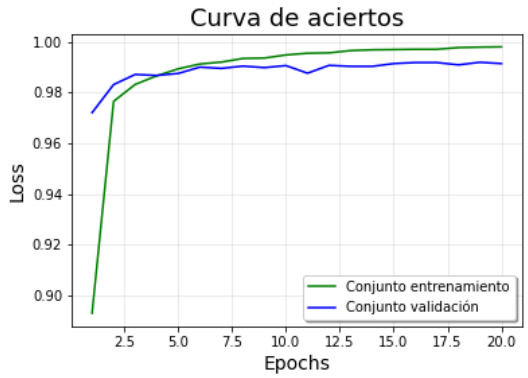

Figura 3.10: Exactitud en la fase de entrenamiento y validación.

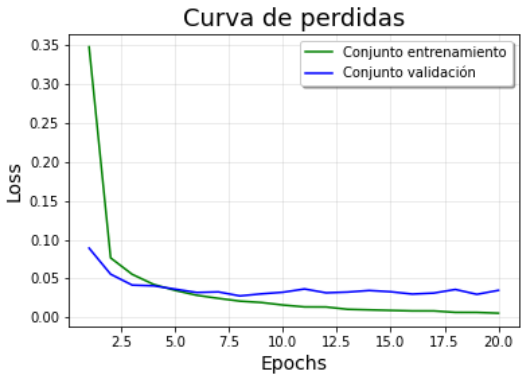

Figura 3.11: Función de pérdidas en la fase de entrenamiento y validación.

este procesado no se emplea en el conjunto de *test* y validación.

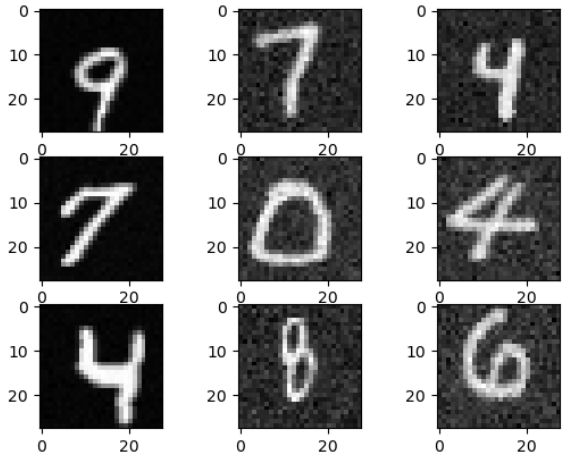

Figura 3.12: Muestras con el procesado empleado para aumentar los datos.

También se han añadido a los bloques de convolucionales dos capas más, una capa tradicional de normalización de lotes y *dropout*. *Batch normalization* se ha incorporado justo después de la función de activación de la capa convolucional [49, 50] y *dropout* a la salida de la capa de *max-pooling* y entre las capas totalmente conectadas. En el tercer bloque como no se dispone de capa de max pooling se ha añadido justo después de la capa de normalización de lote. Con una probabilidad de desconexión de $p = 0.25$. En la figura 3.13 se visualiza la estructura de modelo una vez aplicadas estas técnicas.

La red continúa teniendo el mismo número de parámetros entrenables, pero el tiempo de entrenamiento aumentó de forma realmente notable. Debido principalmente al procesado de las imágenes de entrenamiento que se realiza en tiempo real conforme las imágenes se introducen en el modelo y esto supone un coste computacional bastante elevado. La técnica de normalización por lotes aumenta el coste computacional al tener que aplicar esta operación [5]. Aplicar estas técnicas supuso un aumento del tiempo de cómputo del modelo. En este caso fueron necesarios dos horas para completar la fase de entrenamiento con los 40 *epochs* y 256 muestras por lote. Con los mismos 20 *epochs* el modelo previo hubiese sido necesario una hora para completar la fase de entrenamiento, significando un

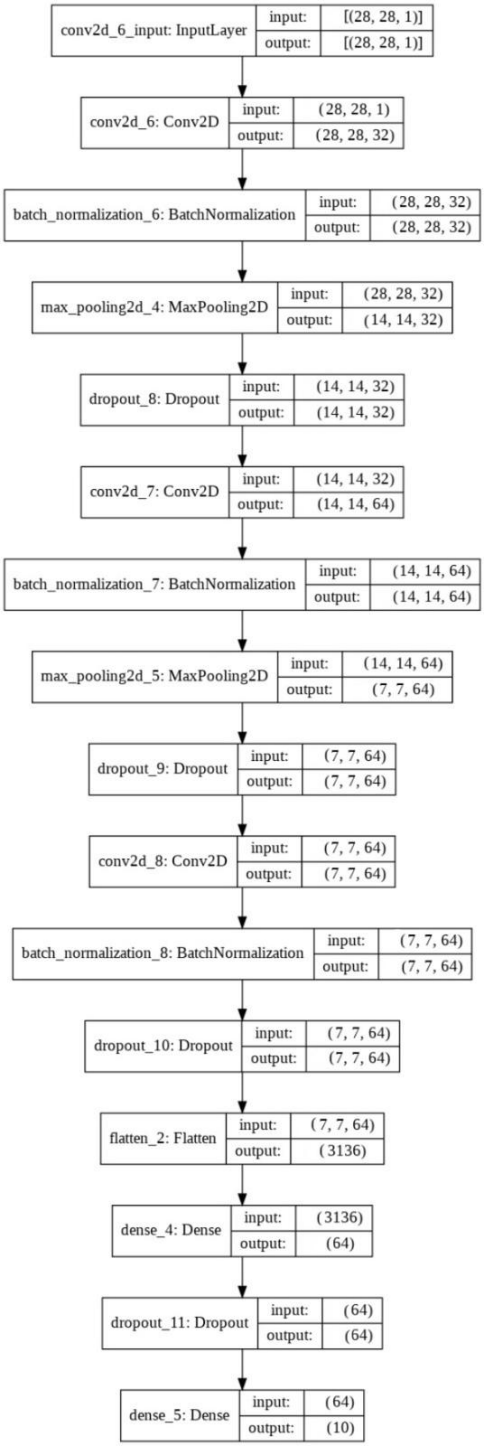

Figura 3.13: Estructura del segundo modelo implementado mediante técnicas de regularización.

coste computacional diez veces mayor al aplicar las técnicas de regularización.

La función de pérdidas del conjunto de testeo ha mejorado frente al modelo previo tras implementar las técnicas de regularización. El error del conjunto de entrenamiento ahora es mayor, puesto que se ha dificultado la tarea de entrenamiento. Como consecuencia el modelo funciona mucho mejor en datos no vistos previamente y aumentando su capacidad de generalizar [4, 2].

Con el objetivo de conocer un poco mejor el comportamiento del modelo, se han crea-

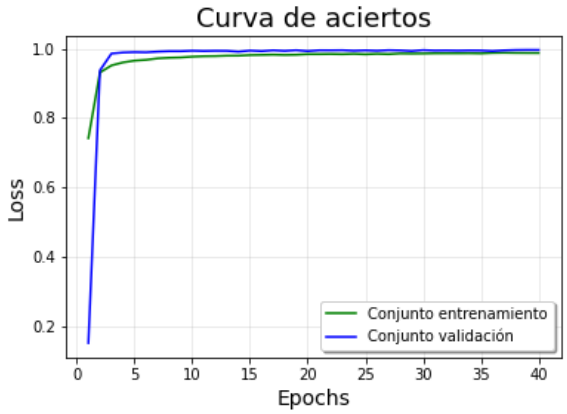

Figura 3.14: Exactitud en la fase de entrena-miento y validación.

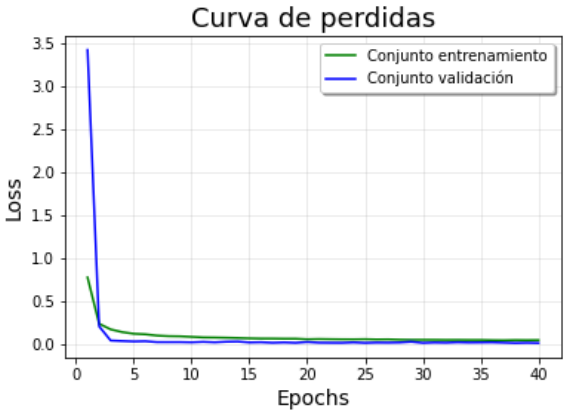

Figura 3.15: Función de pérdidas en la fase de entrenamiento y validación.

do las figuras del mapa de características a la salida de las capas convolucionales. De esta forma se puede observar la presencia de los filtros aprendidos por el modelo sobre las imágenes de entrada. En la figura 3.16, se visualizan los mapas de características de las diferentes de la salida de la primera convolución, como todavía no se han reducido las dimensiones se conserva casi toda la información de la imagen y, por tanto, los patrones extraídos son generales. Conforme se profundiza la red, se reduce la dimensionalidad de los mapas de características visualizándose activaciones más complejas y abstractas. Por ejemplo, en la figura 3.17, que corresponde con la capa intermedia, se aprecian contornos, ejes, etc. En la figura 3.18, se visualiza las activaciones de la última capa, donde la dimensionalidad se ha reducido al máximo. Las características que se observan son muy abstractas y ya no es posible su entendimiento. Puede que algunos filtros no se activen (aparecería representado en negro). Esto ocurre, sobre todo, conforme se profundiza en la red. Esto muestra que la capacidad de generalización del modelo está llegando al límite. En las figuras este efecto no se observa pero puede que el modelo con otras muestras diferente tenga una menor capacidad de generalizar.

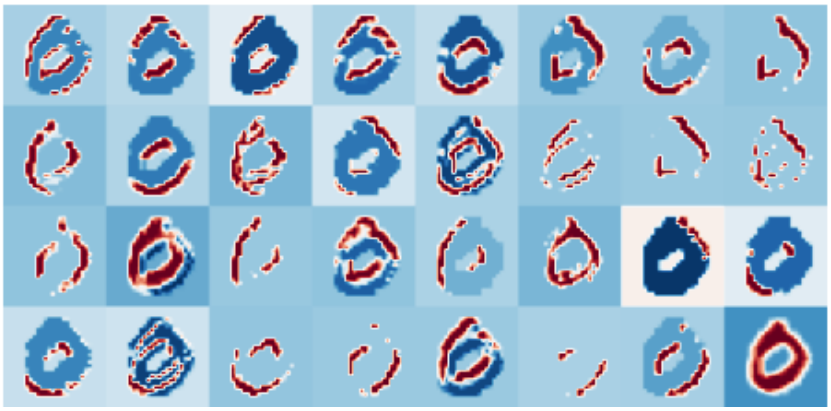

Figura 3.16: Mapas de características de la primera capa convolucional.

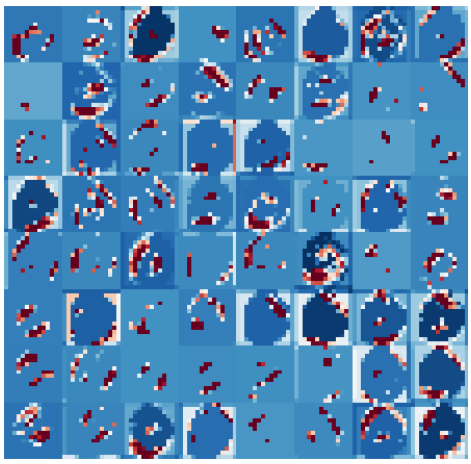

Figura 3.17: Mapas de características de la capa convolucional intermedia.

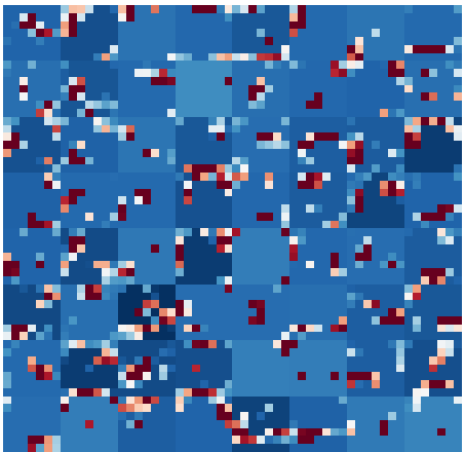

Figura 3.18: Mapas de características de la última capa convolucional.

3.5. Resultados

Una vez propuesto el modelo, se mostrarán los resultados obtenidos en la clasificación. Primeramente se detallarán los resultados mediante la matriz de confusión y las métricas de evaluación explicadas en el apartado 3.3. Finalmente se analizarán los resultados y se visualizarán las muestras predecidas incorrectamente por el modelo y los mapas de características, para poder comprender mejor el modelo.

La figura 3.19 muestra la matriz de confusión una vez terminada la fase de *test* mediante el conjunto de datos de entrenamiento. El eje x contiene las predicciones de cada clase y el eje y las etiquetas "*ground truth*" de las muestras predichas. Por tanto, en la diagonal de la matriz, se encontrará el número de muestras que han sido precedidas correctamente para cada clase. Las demás casillas contienen el número de muestras predichas incorrectamente de cada clase y, según el eje y, la clase a la que realmente pertenecía.

La exactitud del modelo en la fase de testeo ha sido de 99.59 %, de las 10.000 imágenes del conjunto de entrenamiento; sólo se ha predecido de forma errónea en 41 imágenes. Este resultado es muy bueno se ha logrado conseguir una tasa de error menor que la propia del ser humano. En la publicación *Efficient Pattern Recognition* [55] analiza el comportamiento del ser humana frente al *dataset NIST*, se obtuvo de una tasa de error ligeramente superior del 1 %. Además, si se comparan los resultados con la competición de *MNIST* de *Kaggle* [56], nuestro modelo se posicionaría alrededor del top 300 (Agosto, 2020).

La exactitud no permite analizar el resultado general del modelo pero no es muy informativa en cuanto al comportamiento específico de cada clase. En la tabla 3.1, se han calculado los resultados utilizando las métricas precisión, *Recall* y *F1-Score*. La clase con peores resultados es la del dígito nueve con diferencia, esto se debe a la cercanía con la clase del dígito cuatro, que es la segunda peor. En otras clases ocurre lo mismo pero de forma menos frecuente. Por ejemplo, la clase del dígito cinco es confundida con la clase del dígito tres. El trazado de la grafía provoca que el modelo tenga dificultades en el reconocimiento de los dígitos, ya que los patrones que aprende para las dos clases son muy similares.En la figura 3.20, se pueden observar las 41 muestras predecidas incorrectamente. Muchas muestras son realmente difíciles de predecir, puesto que presentan lo que se conoce como disgrafía caligráfica, es decir, una escritura defectuosa que resulta difícil

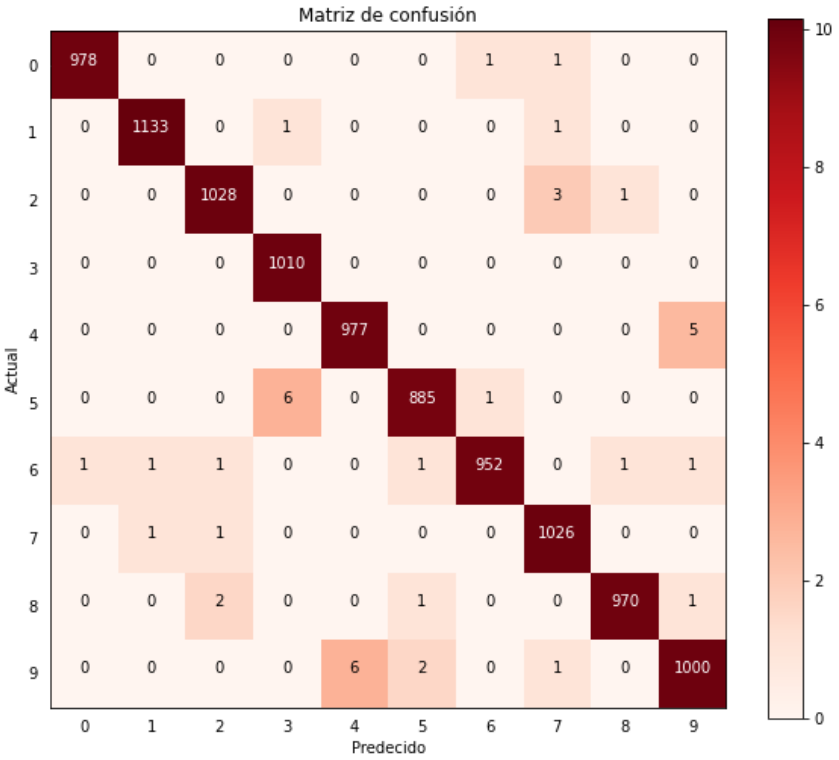

Figura 3.19: Matriz de confusión de las prediciones del modelo en fase de prueba.

de reconocer y puede provocar una confusión. Las muestras provocan confusión ya que contienen deformaciones en tamaño, forma y el trazado de los dígitos, etc. La calidad de la escritura en las muestras viene determinada por la rigidez de la escritura, impulsividad que produce letras difusas, etc.

Clase	Precisión	Recall	F1-Score
0	99,90 %	99,80 %	99,85 %
1	99,82 %	99,82 %	99,82 %
2	99,61 %	99,61 %	99,61 %
3	99,31 %	100,00 %	99,65 %
4	99,39 %	99,49 %	99,44 %
5	99,55 %	99,22 %	99,38 %
6	99,79 %	99,37 %	99,58 %
7	99,42 %	99,81 %	99,61 %
8	99,79 %	99,59 %	99,69 %
9	99,30 %	99,11 %	99,21 %
Total	99,59 %	99,58 %	99,59 %

Cuadro 3.1: Resultados de las metricas por clase del modelo.

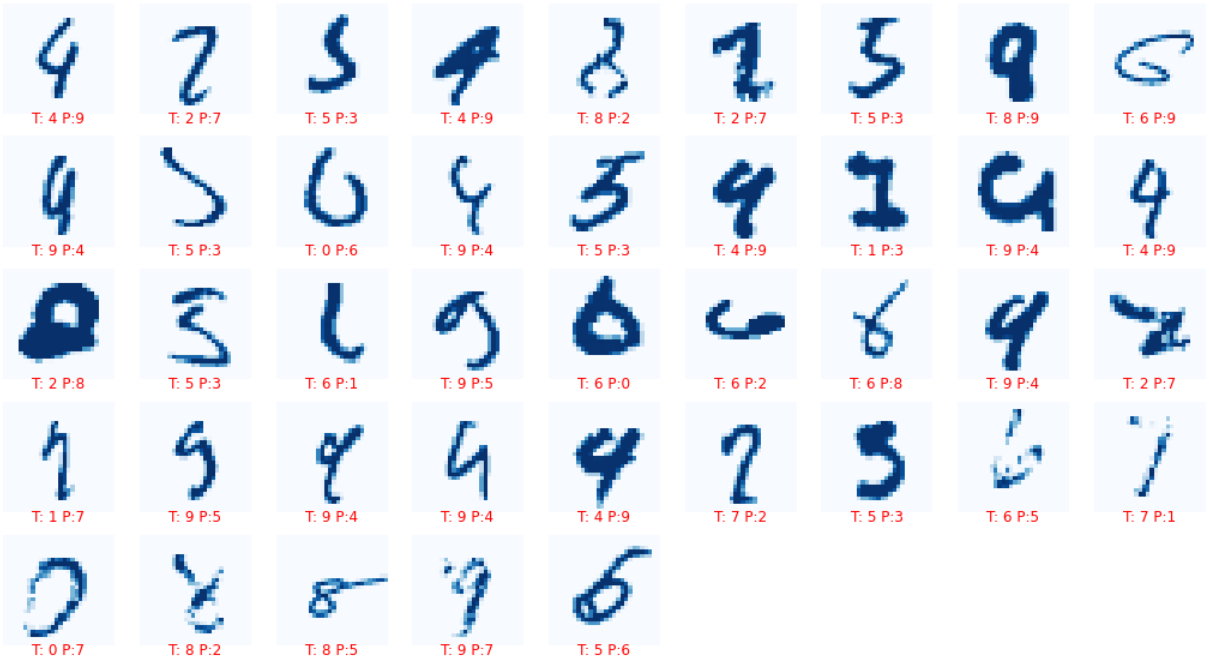

Figura 3.20: Visualización de las muestras predecidas incorrectamente.

Capítulo 4

Conclusiones

El principal objetivo de este proyecto ha sido entrenar un modelo capaz de clasificar dígitos manuscritos. Para ello se ha utilizado un modelo con redes neuronales convolucionales y se ha logrado obtener rendimiento muy satisfactorio.

Primeramente, se analizó el conjunto de datos de *MNIST* para conocer sus características y ha servido de base para el desarrollo de la metodología de evaluación y la implementación del modelo. A continuación, se propuso la metodología de evaluación utilizada para entrenar el modelo. Para el desarrollo del modelo se formuló un primer modelo cuyos resultados sirvieron de base para la elaboración de un segundo modelo idéntico al primero pero aplicando técnicas de regularización. Una vez desarrollado el modelo se han producido diversas gráficas y se han calculado diferentes métricas: exactitud, precisión, *Recall* y, *F1-Score* para un posterior análisis de los resultados y comprensión de este último modelo.

Respecto a los resultados obtenidos del proyecto estoy satisfecho ya que se han cumplido mis objetivos y, por tanto, mis expectativas. No obstante, como se trata de un tema inexistente durante mis estudios en el grado, ha sido necesaria una investigación profunda. He tenido que formarme a través de diferentes cursos, lectura de libros y publicaciones científicas. Además, para llevar a cabo la parte práctica del proyecto he tenido que aprender a programar en *Python* y las librerías como *Keras*, *Numpy*, *Matplotlib*, etc. También, he presentado diversas dificultades a causa de mi inexperiencia en programar y configurar o ajustar modelos, ya que ha supuesto un nuevo paradigma de programación para mi.

Aunque se han mostrado mejores resultados de los esperados, he de admitir que siempre hay margen de mejora. Por ejemplo, se hubiese podido utilizar otra metodología de evaluación más óptima, como es dividir el *dataset* en tres partes: entrenamiento, evaluación y testeo. De esta forma, se evitará un sobreentrenamiento al ajustar el modelo, puesto que de dicha forma la evaluación y el testeo poseen datos independientes. Otro ejemplo de mejora sería utilizar estructuras más avanzadas como redes residuales [36], convoluciones en paralelo , *transfer learning*, etc.

Este presente trabajo me ha servido como una iniciación en el campo del aprendizaje profundo. En un futuro me gustaría ampliar mis conocimientos sobre esta temática.

Apéndice A

Código

```
# ------------------------------------------------------------------------
# Implementation of deep neural network structure for handwritten digits
# ------------------------------------------------------------------------
# YEAR 2019-2020           TFG - GIET - ETSE
# ------------------------------------------------------------------------
# AUTHOR: Carles Serra Vendrell
# ------------------------------------------------------------------------

from keras import layers
from keras import models
from keras.utils import to_categorical
from keras.datasets import mnist
from keras.preprocessing.image import ImageDataGenerator
from keras.models import Model
import matplotlib.pyplot as plt
import numpy as np
import random

#  CARGA DEL DATASET MNIST
#  Se utliza la función propia de keras para cargar los datos del dataset MNIST
#  Esta función nos devuelve una tupla con el conjunto de entrenamiento y el
#  connjunto de validación y test.

(train_images, train_labels), (test_images, test_labels) = mnist.load_data()

# Se muestra la división del dataset
print('============ Dataset Split ============')
print('Training set:', train_images.shape)
print('Test and Validation set:', test_images.shape)
print('=======================================')
print('')


#  Se guardan las etiquetas sin codificar para utilizarlas en graficar la matriz
#  de confusión.
raw_test_labels = test_labels
```

```
# GRÁFICOS DE MUESTRAS ALEATORIAS DE LAS CLASES (CONJUNTO ENTRENAMIENTO)
# Se grafican 5 muestras aleatorias por clase del conjunto de entrenamiento.

numero_muestras = []
numero_columnas = 5
numero_clases = 10

# Configuración de las dimensiones del gráfico
figura, axis = plt.subplots(nrows=numero_clases, ncols=numero_columnas,
                            figsize=(6, 11))
figura.tight_layout()

# Bucle aninado, que recorre las columnas y las clases
for i in range(numero_columnas):
    for j in range(numero_clases):

        # Se eliguen las imagenes que pertenecen a una clase
        x = train_images[train_labels == j]

        # Se representa una imagen aleatoriamente.
        axis[j][i].imshow(x[random.randint(0, len(x - 1)),
                            :, :],
                          cmap=plt.get_cmap('gray'))

        # Desactivar cuadricula
        axis[j][i].axis("off")

        # Titulo de la clase en la 2 columna y se guarda el número de muestras
        # de cada clase.
        if i == 2:
            axis[j][i].set_title(str(j))
            numero_muestras.append(len(x))

# Se guarda el grafico en formato .png
plt.savefig('training_samples.png')

# GRÁFICOS DE FRECUENCIA DE LAS CLASES (CONJUNTO ENTRENAMIENTO)
# Gráfico de barras representando el número de muestras por clase del conjunto
# entrenamiento.

# Configuración de las dimensiones del gráfico
plt.figure(figsize=(10, 6))
plt.bar(range(0, numero_clases), numero_muestras, width=0.7, color="blue")

# Creación de las barras del grafico, con el número de muestras por clase
for i in range(0, numero_clases):
    plt.text(i, numero_muestras[i], str(numero_muestras[i]),
             horizontalalignment='center', fontsize=14)

# Configuración de la apariencia del gráfico.
plt.tick_params(labelsize=14)
plt.xticks(range(0, numero_clases))
```

```
plt.xlabel("Clases", fontsize=16)
plt.ylabel("N de muestras", fontsize=16)
plt.title("Frecuencia del conjunto entrenamiento", fontsize=20)


#  GRÁFICOS DE MUESTRAS ALEATORIAS DE LAS CLASES (CONJUNTO TEST y VALIDACIÓN)
#  Se grafican 5 muestras aleatorias por clase del conjunto de test y validación

numero_muestras = []
numero_columnas = 5
numero_clases = 10

#  Configuración de las dimensiones del gráfico
figura, axis = plt.subplots(nrows=numero_clases, ncols=numero_columnas,
                            figsize=(5, 10))
figura.tight_layout()

#  Bucle aninado, que recorre las columnas y las clases
for i in range(numero_columnas):
    for j in range(numero_clases):

        #  Se eliguen las imagenes que pertenecen a una clase
        x = test_images[test_labels == j]

        #  Se representa una imagen aleatoriamente.
        axis[j][i].imshow(x[random.randint(0, len(x) - 1),
                            :, :],
                          cmap=plt.get_cmap('gray'))

        #  Desactivar cuadricula
        axis[j][i].axis("off")

        #  Titulo de la clase en la 2 columna y se guarda el número de muestras
        #  de cada clase.
        if i == 2:
            axis[j][i].set_title(str(j))
            numero_muestras.append(len(x))

# Se guarda el grafico en formato .png
plt.savefig('validation_samples.png')

#  GRÁFICOS DE FRECUENCIA DE LAS CLASES (CONJUNTO ENTRENAMIENTO)
#  Gráfico de barras representando el número de muestras por clase del conjunto
#  validación y test.

#  Configuración de las dimensiones del gráfico.
plt.figure(figsize=(10, 6))

#  Creación de las barras del grafico, con el número de muestras por clase
plt.bar(range(0, numero_clases), numero_muestras, width=0.8, color="blue")
for i in range(0, numero_clases):
    plt.text(i, numero_muestras[i], str(numero_muestras[i]),
```

```
            horizontalalignment='center', fontsize=14)

# Configuración de la apariencia del gráfico.
plt.tick_params(labelsize=14)
plt.xticks(range(0, numero_clases))
plt.xlabel("Clases", fontsize=16)
plt.ylabel("N de muestras", fontsize=16)
plt.title("Frecuencia del conjunto validación y test", fontsize=20)

# Se guarda elgráfico en formato .png
plt.savefig('digit_frequency_val.png')

# Se muestran todos los gráficos.
plt.show()

# PREPROCESADO DE LOS DATOS

# Las imagenes se convierten en tensores de 3 dimnsiones para poder ser
# con las conv2d de keras.
train_images = train_images.reshape((60000, 28, 28, 1))

# Se normalizan las imagenes en un factor 1/255 y se convierten en tipo float
train_images = train_images.astype('float32') / 255

# Las imagenes se convierten en tensores de 3 dimnsiones para poder ser
# con las conv2d de keras.
test_images = test_images.reshape((10000, 28, 28, 1))

# Se normalizan las imagenes en un factor 1/255 y se convierten en tipo float
test_images = test_images.astype('float32') / 255

# Se codifican las etiquetas como one-hot enconding
train_labels = to_categorical(train_labels)
test_labels = to_categorical(test_labels)


# AUMENTACIÓN DE LOS DATOS

# Función propia, ruido gaussiano
def ruido(imagen):
    varianza = 0.1
    desviacion = varianza * random.random()
    ruido = np.random.normal(0, desviacion, imagen.shape)
    imagen += ruido
    np.clip(imagen, 0., 255.)
    return imagen


# Configuración del generador de imagenes.
datagen = ImageDataGenerator(zoom_range=0.1,
                             width_shift_range=0.1,
                             height_shift_range=0.1,
```

```
                        preprocessing_function=ruido)

#  Solo utilizamos aumentación en el conjunto de entrenamiento. Se indica al
#  al generador que imagenes tiene que procesar
datagen.fit(train_images)

# Se grafican las primeras 9 muestras generadas por ImageDataGenerator
for x_batch, y_batch in datagen.flow(train_images, train_labels, batch_size=9):
    for i in range(0, 9):
        plt.subplot(330 + 1 + i)
        plt.imshow(x_batch[i].reshape(28, 28), cmap=plt.get_cmap('gray'))
    plt.show()
    break
plt.savefig('dataAugmentation.png')


#  MODELo

#  Se indica que es un modelo secuencial
model = models.Sequential()

#  Se añaden las capas al modelo

#  Bloque 1 CNN
model.add(layers.Conv2D(32, (3, 3),
                        activation='relu',
                        padding='same',
                        use_bias=True,
                        input_shape=(28, 28, 1)))
model.add(layers.BatchNormalization())
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Dropout(0.25))

#  Bloque 2 CNN
model.add(layers.Conv2D(64, (3, 3),
                        activation='relu',
                        padding='same',
                        use_bias=True))
model.add(layers.BatchNormalization())
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Dropout(0.25))

# Bloque 3 CNN
model.add(layers.Conv2D(64, (3, 3),
                        activation='relu',
                        padding='same',
                        use_bias=True))
model.add(layers.BatchNormalization())
model.add(layers.Dropout(0.25))

#  Bloque 4 FC
model.add(layers.Flatten())
```

```
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dropout(0.5))
model.add(layers.Dense(10, activation='softmax'))

#  Se configura la función de perdidas y el algoritmo de apredizaje.
model.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['accuracy'])

#  Visualización de los bloques y parametros del modelo implementado.
model.summary()


#  Se indica que datos alimentan al modelo en la fase de entrenamiento y en la
# de validación. En este caso los datos de entrenamiento viene generador tras
# procesar el conjunto de entrenamiento.
history = model.fit(datagen.flow(train_images, train_labels,
                                 batch_size=256),
                    steps_per_epoch=int(train_images.shape[0] / 256) + 1,
                    epochs=40,
                    validation_data=(test_images, test_labels))

#  TEST
#  Se testea la precisión del modelo en el conjunto de datos de testeo.
test_loss, test_acc = model.evaluate(test_images, test_labels)
print('Test accuracy:', test_acc)

#  GRÁFICOS DE LA FUNCIÓN DE PERDIDAS Y DE ACIERTOS

#  Se obtiene los datos la función de perdidas calculados por el modelos. Tanto
#  la de entrenamiento como la de validación.
loss = history.history['loss']

#  Se obtiene los datos la función de perdidas calculados por el modelos. Tanto
#  la de entrenamiento como la de validación.
val_loss = history.history['val_loss']

#  Generación del vector de épocas
epochs = range(1, len(loss) + 1)

#  Generamos el grafico de la función de perdidas en entrenamiento y validación
plt.plot(epochs, loss, 'g', label='Conjunto entrenamiento')
plt.plot(epochs, val_loss, 'b', label='Conjunto validación')

#  Se configura la apariencia del gráficos
plt.title('Curva de perdidas', fontsize=18)
plt.xlabel('Epochs', fontsize=14)
plt.ylabel('Loss', fontsize=14)
plt.grid(alpha=0.3)
plt.legend(loc='best', shadow=True)

#  Guardar en formato .png y mostrar el gráfico.
```

```
plt.savefig('loss.png')
plt.show()

#  Limpiar representación anterior.
plt.clf()


#  Se obtiene los datos la función de aciertos calculados por el modelos. Tanto
#  la de entrenamiento como la de validación.
acc = history.history['accuracy']
val_acc = history.history['val_accuracy']

#  Generamos el grafico de la función de perdidas en
#  entrenamiento y validación.
plt.plot(epochs, acc, 'g', label='Conjunto entrenamiento')
plt.plot(epochs, val_acc, 'b', label='Conjunto validación')
plt.title('Curva de aciertos', fontsize=18)
plt.xlabel('Epochs', fontsize=14)
plt.ylabel('Loss', fontsize=14)
plt.grid(alpha=0.3)
plt.legend(loc='best', shadow=True)

#  Guardar en formato .png y mostrar el gráfico.
plt.savefig('val.png')
plt.show()

#  MATRIZ DE CONFUSIÓN

#  Generación de la matriz de confusión.

#  Se obtienen las predicciones del modelo.
valores_predecidos = np.argmax(model.predict(test_images), axis=1)

fallos = 0

#  Rellenamos matriz con ceros.
matriz_confusion = np.zeros([10, 10])

#  Rellenamos la matriz con las predicciones y se cuentan el número de fallos.
for i in range(test_images.shape[0]):
    matriz_confusion[raw_test_labels[i], valores_predecidos[i]] += 1
    if raw_test_labels[i] != valores_predecidos[i]:
        fallos += 1

#  Configuramos las dimensiones de la figura
f = plt.figure(figsize=(11, 9))
f.add_subplot(111)

#  Configuramos el aspecto del gráfico
plt.imshow(np.log2(matriz_confusion + 1), cmap="Reds")
plt.colorbar()
plt.tick_params(size=5, color="white")
```

```
plt.xticks(np.arange(0, 10), np.arange(0, 10))
plt.yticks(np.arange(0, 10), np.arange(0, 10))
plt.xlabel("Predecido")
plt.ylabel("Actual")
plt.title("Matriz de confusión")

#  Se establece un umbral para la coloración.
threshold = matriz_confusion.max() / 2

#  Rellenamos el grafico mediante la matriz de confusión
for i in range(10):
    for j in range(10):
        plt.text(j,
                 i,
                 int(matriz_confusion[i, j]),
                 horizontalalignment="center",
                 color="white" if matriz_confusion[i, j] > threshold else "black")

#  Guardamos en formato .png y se muestra el gráfico.
plt.savefig("confusionMatrix.png")
plt.show()

#  FIGURA CON LAS MUESTRAS PREDICHAS DE FORMA INCORRECTA

#  Dimensiones de la figura.
rows = 9
cols = 9
f = plt.figure(figsize=(2 * cols, 2 * rows))

subplot = 1
for i in range(test_images.shape[0]):

    #  Si se ha predecido correctamente.
    if raw_test_labels[i] != valores_predecidos[i]:

        # Generamos subplot
        f.add_subplot(rows, cols, subplot)
        subplot += 1

        #  Mostramos imagen mal predecida
        plt.imshow(test_images[i].reshape([28, 28]), cmap="Blues")

        #  Desactivamos la cuadricula y generamos el titulo.
        plt.axis("off")
        plt.title(
            "T: " + str(raw_test_labels[i]) + " P:" + str(valores_predecidos[i]),
                y=-0.15, color="Red")

#  Guardamos en formato .png y mosatramos la figura.
plt.savefig("error_plots.png")
plt.show()
```

```
#  FIGURAS DE LOS MAPAS DE CARACTERÍTICAS DE CADA CAPA

#  Entrada que utilizaremos para generar los mapas de caracteristicas.
imagen = test_images[0].reshape(1, 28, 28, 1)

#  Obtenemos las capas de salida del modelo
capas_salida = [layer.output for layer in model.layers]

#  Obtenemos los mapas de caracteristicas de la imagen
actiaciones_del_modelo = Model(inputs=model.input, outputs=capas_salida)
mapas = actiaciones_del_modelo.predict(imagen)

#  Nombres de las capas
nombre_capas = []
for layer in model.layers[1:7]:
    nombre_capas.append(layer.name)

mapas_por_columnas = 8
#  Representar el mapa de caracteríticas de cada capa de salida
for nombre_capa, mapa_capa in zip(nombre_capas, mapas):

    # Numero de caracteristicas por mapa de caracteristicas
    numero = mapa_capa.shape[-1]

    # Tamaño del mapa de caracteristicas
    tamaño = mapa_capa.shape[1]

    #  Caracteristicas por columna
    num_columnas = numero // mapas_por_columnas

    #  Generamos matriz con zeros que rellenaremos luego.
    display_grid = np.zeros(
        (tamaño * num_columnas, mapas_por_columnas * tamaño))

    #  Rellenamos matriz con las activaciones porcesadas para visualizarse.
    for col in range(num_columnas):
        for fila in range(mapas_por_columnas):

            # Procesamos cada caracteristica
            canal_imagen = mapa_capa[0,
                                     :, :,
                                     col * mapas_por_columnas + fila]
            canal_imagen -= canal_imagen.mean()
            canal_imagen /= canal_imagen.std()
            canal_imagen *= 64
            canal_imagen += 128
            canal_imagen = np.clip(canal_imagen, 0, 255).astype('uint8')

            #  Rellenamos las caracteristicas tras ser procesada en la matriz
            display_grid[col * tamaño: (col + 1) * tamaño,
                         fila * tamaño: (fila + 1) * tamaño] = canal_imagen
```

```
    #  Configuramos el tamaño de la figura
    escala = 1. / tamaño
    plt.figure(figsize=(escala * display_grid.shape[1],
                        escala * display_grid.shape[0]))

    # Titulo de la figura
    plt.title(nombre_capa)

    #  Desativamos la cuadricula y los ejes
    plt.grid(False)
    plt.axis('off')

    #  Representamos en la figura la matriz generada
    plt.imshow(display_grid, aspect='auto', cmap='Greys')
```

Bibliografía

[1] Raymond Perrault, Yoav Shoham, Erik Brynjolfsson, Jack Clark, John Etchemendy, Barbara Grosz, Terah Lyons, James Manyika, Juan Carlos Niebles, and Saurabh Mishra. Artificial intelligence. Report, jan 2020. https://hai.stanford.edu/sites/default/files/ai_index_2019_report.pdf.

[2] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[3] Perceptrón, wikipedia. Web page, Abril 2020. https://es.wikipedia.org/wiki/Perceptr%C3%B3n#/media/Archivo:Perceptr%C3%B3n_5_unidades.svg.

[4] François Chollet. *Deep Learning with Python*. Manning, November 2017.

[5] Adrian Rosebrock. *Deep learning for Computer Vision with Python. Pyimagesearch*, 2017.

[6] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[7] Fei-Fei Li, Ranjay Krishna, and Danfei Xu. Cs231n: Convolutional neural networks for visual recognition. Course Lectures, may 2020.

[8] Geoffrey Hinton. Neural networks for machine learning. Course Lectures. https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf.

[9] Frank Krüger. *Activity, Context, and Plan Recognition with Computational Causal Behaviour Models*. PhD thesis, 12 2016.

[10] D. Kelnar. The state of ai divergence. *MMC Ventures*, page 151, 2019.

[11] S. Feldman. Machine learning tops dollars. *Statista*, 2019. https://www.statista.com/chart/17966/worldwide-artificial-intelligence-funding/.

[12] Market research future. Global machine learning market research report: by component (hardware, software), organization size (large enterprise, small and medium-sized enterprises) vertical (bfsi, media and entertainment, automotive, telecommunication, retail and e-commerce, education, healthcare, government and defense, others) and region (north america, europe, asia-pacific, rest of the world) - forecast to 2024. Report, 2019. https://www.marketresearchfuture.com/reports/machine-learning-market-2494.

[13] James A. Anderson and Edward Rosenfeld, editors. *Neurocomputing: Foundations of Research*. MIT Press, Cambridge, MA, USA, 1988.

[14] Geoffrey E. Hinton and Tim Shallice. Lesioning an attractor network: Investigations of volume = 98, year = 1991. *Psychological Review*, (1):74–95.

[15] F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, pages 65–386, 1958.

[16] Bernard Widrow, Hoff, and Marcian E. Adaptive switching circuits. In *1960 IRE WESCON Convention Record, Part 4*, pages 96–104. IRE, 1960.

[17] Marvin Minsky and Seymour Papert. *Perceptrons: An Introduction to Computational Geometry*. MIT Press, 1969.

[18] James A. Anderson and Edward Rosenfeld, editors. *Neurocomputing: Foundations of Research*. MIT Press, Cambridge, MA, USA, 1988.

[19] Geoffrey E. Hinton, Simon Osindero, and Yee-Whye Teh. A fast learning algorithm for deep belief nets. *Neural Comput.*, 18(7):1527–1554, July 2006.

[20] Yoshua Bengio, Pascal Lamblin, Dan Popovici, and Hugo Larochelle. Greedy layer-wise training of deep networks. In *Proceedings of the 19th International Conference on Neural Information Processing Systems*, NIPS'06, page 153–160, Cambridge, MA, USA, 2006. MIT Press.

[21] Marc' Aurelio Ranzato, Y-Lan Boureau, and Yann LeCun. Sparse feature learning for deep belief networks. In *Proceedings of the 20th International Conference on Neural Information Processing Systems*, NIPS'07, page 1185–1192, Red Hook, NY, USA, 2007. Curran Associates Inc.

[22] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCand lish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language Models are Few-Shot Learners. *arXiv e-prints*, page arXiv:2005.14165, May 2020.

[23] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *arXiv e-prints*, page arXiv:1810.04805, October 2018.

[24] Open images dataset. Web page. https://opensource.google/projects/open-images-dataset.

[25] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, November 1997.

[26] Derrick Mwiti. Real-life applications of reinforcement learning, *Neptune.ai*. Blog. https://neptune.ai/blog/reinforcement-learning-applications#:~:text=Applications%20in%20self%2Ddriving%20cars&text=Some%20of%20the%20autonomous%20driving,based%20learning%20policies%20for%20highways.

[27] Arxiv.org. Web Page. https://arxiv.org/.

[28] Papers with code. Web Page. https://paperswithcode.com/.

[29] Thanh Thi Nguyen, Cuong M. Nguyen, Dung Tien Nguyen, Duc Thanh Nguyen, and Saeid Nahavandi. Deep Learning for Deepfakes Creation and Detection: A Survey. *arXiv e-prints*, sep 2019.

[30] S. Hochreiter, Y. Bengio, P. Frasconi, and J. Schmidhuber. Gradient flow in recurrent nets: the difficulty of learning long-term dependencies. In S. C. Kremer and J. F. Kolen, editors, *A Field Guide to Dynamical Recurrent Neural Networks*. Field Guide to Dynamical, 2001.

[31] Richard H. R. Hahnloser and H. Sebastian Seung. Permitted and forbidden sets in symmetric threshold-linear networks. In *Proceedings of the 13th International Conference on Neural Information Processing Systems*, NIPS'00, page 199–205, Cambridge, MA, USA, 2000. MIT Press.

[32] Andrew L. Maas, Awni Y. Hannun, and Andrew Y. Ng. Rectifier nonlinearities improve neural network acoustic models. In *in ICML Workshop on Deep Learning for Audio, Speech and Language Processing*, 2013.

[33] Kaiming He et al. Deep residual learning for image recognition. *CoRR abs/1512.03385*, 2015.

[34] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[35] C. Szegedy, Wei Liu, Yangqing Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1–9, June 2015.

[36] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. 2015.

[37] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In Yee Whye Teh and D. Mike Titterington, editors, *AISTATS*, volume 9 of *JMLR Proceedings*, pages 249–256. JMLR.org, 2010.

[38] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the 2015 IEEE International Conference on Computer Vision (ICCV)*, ICCV '15, page 1026–1034, USA, 2015. IEEE Computer Society.

[39] Ning Qian. On the momentum term in gradient descent learning algorithms. *Neural Netw.*, 12(1):145–151, January 1999.

[40] Y. Nesterov. A method for unconstrained convex minimization problem with the rate of convergence o(1=k2). 1983.

[41] John C. Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. 12:2121–2159, 2011.

[42] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2014. cite arxiv:1412.6980Comment: Published as a conference paper at the 3rd International Conference for Learning Representations, San Diego, 2015.

[43] Tom Schaul, Ioannis Antonoglou, and David Silver. Unit Tests for Stochastic Optimization. *arXiv e-prints*, page arXiv:1312.6055, December 2013.

[44] C. Shorten and T. Khoshgoftaar. A survey on image data augmentation for deep learning. *Journal of Big Data*, 2019.

[45] Fabio Henrique Kiyoiti dos Santos Tanaka and Claus Aranha. Data Augmentation Using GANs. *arXiv e-prints*, page arXiv:1904.09135, April 2019.

[46] Nitish Srivastava, Geoffrey E. Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[47] We are the google brain team. we'd love to answer your questions about machine learning. Reddit thread. http://mng.bz/XrsS.

[48] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*, ICML'15, page 448–456. JMLR.org, 2015.

[49] Batch normalization before or after relu? Reddit thread. https://www.reddit.com/r/MachineLearning/comments/67gonq/d_batch_normalization_before_or_after_relu/.

[50] ducha-aiki/caffenet-benchmark. Github. https://github.com/ducha-aiki/caffenet-benchmark/blob/master/batchnorm.md.

[51] Mnist dataset. Official Web page. http://yann.lecun.com/exdb/mnist/.

[52] S. Visa, B. Ramsay, A. Ralescu, and E. Knaap. Confusion matrix-based feature selection. In *MAICS*, 2011.

[53] Joydwip Mohajon. Confusion matrix for your multi-class machine learning model. *Towards Data Science*, 2020. https://towardsdatascience.com/confusion-matrix-for-your-multi-class-machine-learning-model-ff9aa3bf7826.

[54] Yann LeCun, Léon Bottou, Genevieve B. Orr, and Klaus-Robert Müller. Efficient backprop. In *Neural Networks: Tricks of the Trade, This Book is an Outgrowth of a 1996 NIPS Workshop*, page 9–50, Berlin, Heidelberg, 1998. Springer-Verlag.

[55] Patrice Simard, Yann LeCun, and John S. Denker. Efficient pattern recognition using a new transformation distance. In *Advances in Neural Information Processing Systems 5, [NIPS Conference]*, page 50–58, San Francisco, CA, USA, 1992. Morgan Kaufmann Publishers Inc.

[56] Digit recognizer, learn computer vision fundamentals with the famous mnist data. Web page. https://www.kaggle.com/c/digit-recognizer/overview.