Trabajo Fin de Máster

Máster Universitario en Ingeniería de Telecomunicación

Identificación del modelo de cámara mediante Redes Neuronales Convolucionales

Autora: Sara Domínguez Pavón

Tutor: Rubén Martín Clemente

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

Sevilla, 2019

Trabajo Fin de Máster
Máster Universitario en Ingeniería de Telecomunicación

Identificación del modelo de cámara mediante Redes Neuronales Convolucionales

Autora:

Sara Domínguez Pavón

Tutor:

Rubén Martín Clemente

Profesor titular

Dep. de Teoría de la Señal y Comunicaciones

Escuela Técnica Superior de Ingeniería

Universidad de Sevilla

Sevilla, 2019

Trabajo Fin de Máster: Identificación del modelo de cámara mediante Redes Neuronales Convolucionales

Autora: Sara Domínguez Pavón

Tutor: Rubén Martín Clemente

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

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2019

El Secretario del Tribunal

A mi familia

Agradecimientos

A mi familia, por darme fuerza, por su confianza y por enseñarme la virtud del esfuerzo y hacerme entender que con dedicación todo se consigue. A todos ellos, gracias por creer en mí.

A mi tutor, Rubén Martín, por volver a querer realizar un nuevo proyecto conmigo y por permitirme profundizar en este área de conocimiento que tanto nos llamaba la atención a los dos. A él, junto al resto de profesores que me han acompañado a lo largo de estos años, agradecer todo lo que he aprendido de cada uno de ellos.

A todos esos compañeros de clase con los que he coincidido durante todos estos años que se han acabado convirtiendo en grandes amigos, con los que he compartido grandes historias y junto a los que he crecido.

A Ángel, por su apoyo incondicional siempre que lo he necesitado. Gracias por seguir aquí.

Sara Domínguez Pavón

Sevilla, 2019

Resumen

La identificación del modelo de cámara siempre ha sido uno de los campos principales del análisis forense de imágenes, ya que es la base para resolver una amplia gama de problemas forenses. Dado que el Deep Learning ha logrado un gran progreso en las tareas de visión por computador, ha surgido un gran interés en la aplicación del aprendizaje profundo en imágenes forenses. En este documento, se propone un método de identificación de modelo de cámara basado en redes neuronales convolucionales profundas (CNNs). A diferencia de los métodos tradicionales, las CNNs pueden extraer características de forma automática y simultánea y aprender a clasificar durante el proceso de aprendizaje. En el presente trabajo se describe un enfoque de aprendizaje profundo para el problema de detección de cámara entre 3 modelos diferentes como parte del IEEE Signal Processing Cup 2018: Camera Model Identification organizado por IEEE Signal Processing Society. Los experimentos muestran que podemos detectar modelos de cámara desconocidos con una precisión de más del 90%.

Abstract

Source camera model identification has always been one of the main fields of digital image forensics since it is the foundation of solving a wide range of forensic problems. Several effective camera model identification algorithms have been developed for the practical necessity. However, they are mostly based on traditional machine learning methods. Since Deep Learning has made great progress in computer vision tasks, significant interest has arisen in applying Deep Learning in image forensics. In this paper, we propose a camera model identification method based on deep convolutional neural networks (CNNs). Unlike traditional methods, CNNs can automatically and simultaneously extract features and learn to classify during the learning process. In the current work, we describe our Deep Learning approach to the camera detection task of 3 cameras as a part of the IEEE Signal Processing Cup 2018: Camera Model Identification hosted by IEEE Signal Processing Society. Experiments show that we can detect unknown camera models with an accuracy greater than 90%.

Índice

ÍNDICE DE TABLAS

ÍNDICE DE FIGURAS

ÍNDICE DE CÓDIGOS

1 INTRODUCCIÓN

1.1 Motivación del trabajo

Debido a la creciente disponibilidad de dispositivos de adquisición de imágenes y al desarrollo de la tecnología de la información y redes sociales, las imágenes digitales se encuentran en todas partes en nuestra vida diaria y se están convirtiendo en el proveedor de información más generalizado. Además de desempeñar un papel importante en la difusión de información, las imágenes digitales, como un tipo de datos visuales, generalmente se consideran una certificación de verdad o evidencia ante un tribunal de justicia, ya que tradicionalmente creemos en la integridad de lo que vemos. Sin embargo, las personas pueden adquirir imágenes fácilmente con la popularidad de cámaras y dispositivos móviles y, lo que es más importante, también pueden manipularlas desde la información de origen a los contenidos e incluso crear las que deseen con el desarrollo de software de manipulación de imágenes y redes sociales a lo largo de los años. Por lo tanto, la situación resalta la necesidad de verificar la fuente y la autenticidad de las imágenes digitales. Esto es un trabajo clave en el campo de la imagen forense digital [1].

Entre los problemas abordados por la comunidad de investigadores en imágenes forenses, la determinación de la marca y el modelo de la cámara que capturó una imagen ha sido una de las áreas de investigación más importante en información forense durante más de una década (Figura 1-1). La información sobre qué tipo de cámara capturó una imagen puede usarse para ayudar a determinar o verificar el origen de una imagen, ayudando a resolver una amplia gama de problemas, desde la identificación de pruebas de delitos hasta la detección de manipulación de fotografías, pudiendo ser además una pieza muy importante en ciertos escenarios como por ejemplo en el análisis de imágenes involucradas en investigaciones de explotación infantil, escenas de actos terroristas, etc.

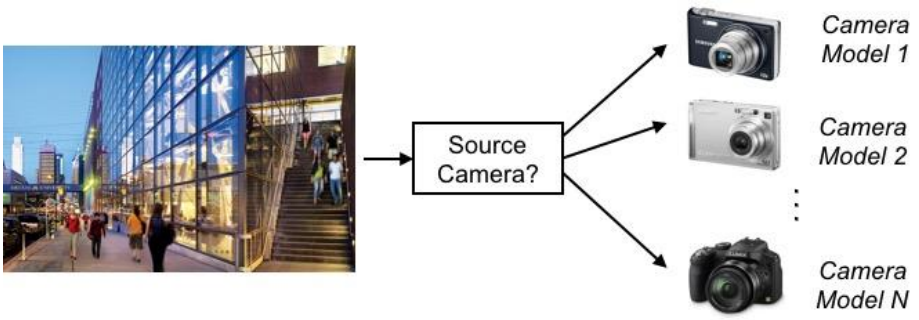


Figura 1-1. Identificación del modelo de cámara

Si bien es cierto que los metadatos pueden contener información sobre la cámara fuente de una imagen, a su vez, dichos metadatos son fáciles de falsificar y frecuentemente faltan en una imagen. Como se verá en el siguiente capítulo, el proceso de adquisición de imágenes implica varias etapas, cada una de las cuales se puede implementar de manera diferente en diferentes cámaras. En consecuencia, se introducen algunos rastros o huellas únicas en las imágenes finales que pueden actuar como activos para detectar el modelo de cámara utilizado.

Para esta tarea, los investigadores se han dedicado a estudiar el proceso de adquisición de imágenes y a explotar dichas trazas intrínsecas para capturar información de origen. Gracias a ello, se han propuesto una variedad de enfoques de identificación de cámaras a lo largo de los años como se verá más adelante, entre los cuales uno de ellos será el seleccionado para implementar en este trabajo.

IEEE Signal Processing Cup 2018 - Forensic Camera Model Identification [2] fue la competición propuesta el año pasado por IEEE Signal Processing Society [3], primera sociedad profesional del mundo para científicos y profesionales del procesamiento de señales desde 1948. Dicha sociedad lleva anualmente a cabo desde 2014 competiciones para alentar a los equipos de estudiantes e investigadores a trabajar juntos para resolver problemas del mundo real utilizando métodos y técnicas de procesamiento de señales. El objetivo del pasado concurso, como su nombre indica, era construir un sistema capaz de determinar la cámara que capturó una imagen digital sin depender de los metadatos. En al artículo [4] se encuentran los tres equipos ganadores, siendo la precisión conseguida por el equipo ganador igual a 99.5%.

Fue a partir de esta competición, vista en Kaggle [5], plataforma online donde se publican competiciones relacionadas con Machine Learning, donde surgió la idea y la motivación de implementar para este trabajo un sistema de identificación del modelo de cámara, una vez vista la gran importancia que tiene esta aplicación en el campo de la imagen forense digital.

1.2 Objetivos y enfoque

El objetivo principal de este trabajo es por tanto el estudio, implementación y evaluación de un sistema capaz de identificar la cámara con la que fue captada una imagen. Se estudiarán posibles métodos que han sido utilizados previamente para abarcar el problema, seleccionando uno de ellos para su posterior estudio en profundidad e implementación.

Dado que se trata de un desafío complejo, pues es propuesto por IEEE Signal Processing Society sobre todo para aquellos estudiantes e investigadores expertos en la materia y con recursos apropiados para poder llevarlo a cabo, no se buscará alcanzar los altos porcentajes de precisión que consiguieron los ganadores de la competición sobre la base de datos completa que el concurso proporcionaba, pues sería algo inviable por falta de recursos, dado que se necesita una gran capacidad de computación. Es por ello que se seleccionará un subconjunto de la base de datos con la que trabajar, donde se verá a lo largo de la memoria que los resultados obtenidos para dicho subconjunto son bastante altos.

1.3 Organización de la memoria

El presente documento se divide en los siguientes capítulos:

- **Capítulo 1: Introducción.** Se trata del capítulo presente, en el cual se exponen las razones que han motivado a la realización del trabajo así como los objetivos perseguidos con el mismo.
- **Capítulo 2: Estado del arte.** En primer lugar se realiza un repaso sobre el proceso de formación de una imagen digital, para después poder realizar un estudio de investigación sobre posibles métodos y técnicas de identificación de la fuente de una imagen.
- **Capítulo 3: Deep Learning para análisis de imágenes.** Tras dicha investigación, se decide emplear Deep Learning, debido a sus altas capacidades y número de posibilidades para el análisis de imágenes. Es por tanto en este capítulo donde se explican todos los conceptos teóricos referentes a ello y, en concreto, al modelo computacional que se va a implementar: redes neuronales.

- **Capítulo 4: Método propuesto.** Una vez conocida toda la teoría que hay detrás del Deep Learning y las redes neuronales, se procede a describir e implementar el método propuesto para el problema inicial presentado. También se expone el material que ha sido utilizado para ello, tanto a nivel de herramientas hardware y software, como la base de datos empleada.
- **Capítulo 5: Experimentos realizados y resultados.** Es en este capítulo donde se evalúa el rendimiento que ofrece el modelo que se ha implementado, realizando para ello una serie de experimentos con los que obtener diferentes métricas y resultados.
- **Capítulo 6: Conclusiones y líneas futuras.** En este capítulo final se resume el trabajo realizado en el proyecto extrayendo las conclusiones más importantes y se proponen líneas de trabajo futuras que hagan posible la continuación y mejora del trabajo aquí expuesto.

2 ESTADO DEL ARTE

En este capítulo se describen las principales técnicas de análisis forense de imágenes digitales haciendo énfasis en las técnicas de identificación de la fuente de la imagen, ya que es la rama del análisis forense en la que se centra este trabajo.

2.1 Formación de una imagen digital

Para comprender el análisis forense de las imágenes digitales lo primero que se requiere conocer es cómo está compuesta una cámara fotográfica y cuál es el procedimiento que realiza para generar una imagen (a menudo llamado *pipeline*) [6]. Las cámaras fotográficas se componen de un sistema de lentes, un grupo de filtros, una matriz de filtro de colores o CFA (*color filter array*), un sensor de imagen y un procesador de imagen. A pesar de que muchos de los detalles del *pipeline* se mantienen como información confidencial de los fabricantes este proceso es muy similar en la mayoría de las cámaras digitales. La estructura básica se muestra en la Figura 2-1:

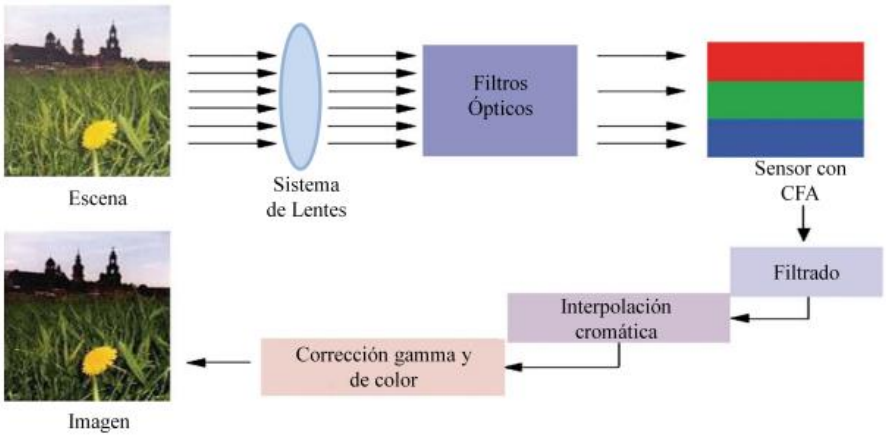


Figura 2-1. Proceso de adquisición de imágenes en cámaras digitales

Como primer paso para generar una imagen el sistema de lentes captura la luz de la escena controlando la exposición, el foco y la estabilización de la imagen. Después, la luz que entra en la cámara a través del sistema de lentes pasa por un grupo de filtros que mejora la calidad visual de la imagen. Este grupo incluye al menos un filtro infrarrojo y un filtro anti-aliasing. El filtro infrarrojo absorbe o refleja la luz permitiendo que sólo la parte visible del espectro pase a la siguiente fase, evitando que la radiación infrarroja ocasione pérdida de nitidez en la imagen. El filtro anti-aliasing se encarga de limpiar la señal produciendo imágenes con contornos más suaves.

A continuación la luz pasa al sensor de la imagen que es una matriz de elementos sensibles a la luz llamados píxeles. Cada elemento de esta matriz de píxeles integra la luz incidente y genera una señal analógica proporcional a la intensidad de la luz recibida. Esta señal se convierte en una señal digital y se transmite al procesador de imagen. Debido a que el sensor de la imagen es monocromático, para capturar una imagen a color se requieren diferentes sensores. Idealmente, un sensor para cada color. Sin embargo, debido al coste que esto implica, en la mayoría de las cámaras sólo se usa un sensor de imagen junto a una matriz de filtros de color que se coloca antes del sensor para producir los colores.

La matriz de filtros de color es una de las partes más importantes de la cadena de procesamiento para la generación de una imagen de las cámaras de un solo sensor. La CFA se encuentra sobre el sensor monocromo, y su función es adquirir la información del color de la escena. Cada celda del filtro de color deja pasar la luz de acuerdo a un rango de longitudes de onda, de tal manera que las intensidades filtradas separadas incluyen información sobre el color de la luz. Como se ilustra en la Figura 2-2, la intensidad de la luz que pasa por cada una de las celdas forma una imagen en escala de grises y, dependiendo de la configuración del filtro CFA, se interpreta como una imagen a color (considerando que cada píxel corresponde a un valor de intensidad).

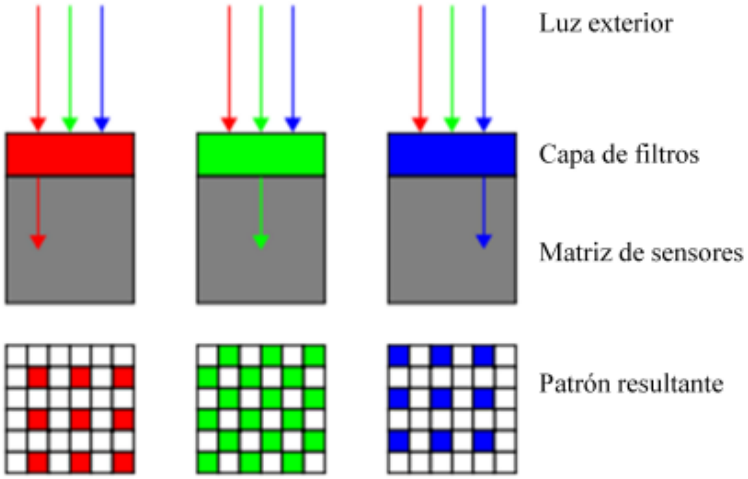


Figura 2-2. Matriz de filtros de color CFA

En este punto el proceso de interpolación cromática (conocida en inglés como *demosaicing*) se lleva a cabo para obtener los valores que faltan para cada uno de los colores del filtro CFA. Este proceso es el más complejo en cuanto a cómputo se refiere. Su algoritmo utiliza los valores de los píxeles vecinos para obtener todos los valores que no han sido medidos.

Antes de realizar la interpolación, una vez que el procesador de imagen recibe la señal digital generada por el sensor elimina el ruido y otras anomalías introducidas en las señales digitales (*artifacts*), con la finalidad de obtener una imagen visualmente agradable. Tras la interpolación, otro proceso al que se somete la imagen es el balanceo de blancos, que permite una reproducción más fiel del color, evitando que haya colores dominantes. Por último, el proceso de corrección gamma ajusta los valores de intensidad de la imagen. Aunque los algoritmos para llevar a cabo estos procesos están presentes en todas las cámaras, los detalles exactos de la forma de realizarlos pueden variar entre los diferentes fabricantes e, incluso, entre los modelos de un mismo fabricante.

Finalmente, la imagen generada por el procesador de imagen se comprime. En las cámaras digitales normalmente se utiliza el algoritmo *Joint Photographic Experts Group* (JPEG) para ahorrar espacio, almacenándose en la memoria del dispositivo junto con la información de la imagen (metadatos) en formato *Exchangeable Image File Format* (EXIF). Se podrían emplear estos metadatos como huellas de la cámara, sin embargo fácilmente pueden ser modificados o eliminados malintencionadamente. También puede ocurrir que no se disponga de los metadatos de la imagen, por ejemplo si ha sido obtenida de una red social. Es por ello que depender de los metadatos para identificar la fuente de una imagen no es una buena opción.

La investigación forense ha demostrado que muchos de los componentes que conforman el *pipeline* de procesamiento interno de una cámara introducen rastros o huellas estadísticas u otros artefactos en una imagen. Debido a que diferentes modelos de cámaras utilizan diferentes implementaciones de cada componente físico y algorítmico en su procesamiento interno, los rastros dejados en una imagen por cada componente pueden vincularse a la marca y modelo de la cámara que capturó la imagen.

2.2 Métodos de identificación de la fuente de una imagen

Una vez visto el proceso de adquisición de imágenes en cámaras digitales y entendido el concepto de rastros o huellas que dicho proceso introduce en las imágenes, ya se pueden presentar los distintos métodos que existen para identificar la fuente de una imagen. Muchos métodos de identificación de cámara han sido propuestos a lo largo de estos años en la literatura, los cuales pueden clasificarse en los dos siguientes grupos [7]:

2.2.1 Métodos basados en un modelo formal

Estos métodos requieren calcular en primer lugar un modelo analítico para después evaluar una proximidad estadística (correlación) entre dicho modelo y la imagen de prueba.

Como el sensor de imágenes digitales no es un dispositivo perfecto, la determinación del origen de la imagen basada en las imperfecciones inherentes del sensor se identifica como un método apropiado. Kurosawa et al. [8] (1999) abordaron inicialmente el problema de la identificación de la cámara de origen desarrollando un método que usa el coeficiente de correlación del ruido de patrón fijo (*Fixed Pattern Noise*, FPN). El FPN es causado por la corriente de oscuridad, que es una señal obtenida del sensor cuando no está expuesto a la luz. Otro enfoque propuesto por Geradts et al. [9] (2001) es el análisis de píxeles defectuosos. Los autores han demostrado que los *dead pixels* (píxeles muertos o defectuosos) podrían utilizarse para la identificación precisa de la cámara incluso a partir de imágenes comprimidas JPEG con pérdidas. Sin embargo, las cámaras recientes no contienen dichos píxeles defectuosos o es posible eliminarlos mediante el procesamiento posterior las imágenes. Lukas et al. [10] (2006) proponen un método para el problema de la identificación de cámaras digitales basado en el patrón de ruido del sensor (*Sensor Pattern Noise*, SPN). El método utiliza ruido de no uniformidad de píxeles, que es un componente estocástico del ruido para todos los sensores de imágenes digitales. Esto se determina promediando el ruido obtenido de múltiples imágenes tomadas por la misma cámara utilizando un filtro de eliminación de ruido.

Choi et al. [11] (2006) notaron que la mayoría de las cámaras digitales tienen lentes con superficies esféricas, cuyas distorsiones radiales inherentes sirven como huellas únicas en las imágenes. Extrajeron por tanto los parámetros de distorsión de las mediciones de la intensidad de los píxeles y las desviaciones, para luego emplear un clasificador para identificar la cámara fuente de una imagen.

Dirik et al. [12] (2008) han investigado un método para la identificación de cámaras DSLR (*Digital Single Lens Reflex*) basado en la detección y correspondencia de las características *dust-spot* (partículas de polvo) del sensor. Debido a que las cámaras DSLR sufren este tipo de problemas en los sensores por sus lentes intercambiables, las manchas de polvo en la imagen se pueden detectar para formar el patrón de polvo de la cámara DSLR. Como los autores mencionaron, el problema con este enfoque es que para las aperturas anchas las manchas de polvo se vuelven casi invisibles y la detección de puntos de polvo en regiones no lisas y complejas es una tarea muy difícil.

Chang-Tsun Li ha investigado la limitación en la extracción del patrón de ruido del sensor. Los SPNs extraídos de las imágenes pueden estar severamente contaminados por los detalles de las escenas, por lo que Chang-Tsun Li [13] (2009) sugirió que se puede obtener una huella digital mejorada asignando factores de ponderación de acuerdo con la magnitud de los detalles de la escena para eliminar la influencia del contenido de la imagen. Para ello planteó la hipótesis de que cuanto más fuerte sea un componente del patrón de ruido del sensor, menos fiable deberá ser el componente y por lo tanto debe atenuarse. Los experimentos confirmaron la solidez de la hipótesis realizada, mejorando la tasa de identificación. Demostraron por tanto que se puede obtener una huella digital mejorada asignando factores de ponderación inversamente proporcionales a la magnitud de las componentes de la señal.

Amerini et al. [14] (2010) proponen un método alternativo para extraer el ruido de No-Uniformidad de Respuesta de Píxel a Píxel (*Pixel-to-pixel Response Non-Uniformity*, PRNU) para la identificación de la cámara de origen. Para la extracción de este tipo de ruido se introduce el filtro de Argenti, basado en un

modelo de ruido dependiente de la señal. También han demostrado que el modelo de ruido de filtro propuesto captura la imagen con mayor precisión y, por lo tanto, es útil en la identificación de cámaras. Caldelli et al. [15] (2010) han propuesto una nueva técnica para distinguir entre las imágenes digitales adquiridas por un escáner y las fotos tomadas por una cámara digital. Debido a la estructura del sensor CCD, el patrón de ruido (PRNU) que queda sobre una imagen digital tendrá una distribución completamente diferente en cada caso. Por lo tanto, la periodicidad del patrón de ruido del sensor a lo largo de la dirección de barrido se comprueba para su clasificación mediante un análisis de frecuencia.

Tomioka et al. [16] (2013) propuso un método basado en las relaciones de pares de grupos de píxeles para suprimir los efectos de la contaminación por ruido. Recientemente, Li et al. [17] propuso el uso del análisis de componentes principales (PCA) para formular una representación compacta de SPN. Además, también se propuso un procedimiento de construcción de conjuntos de entrenamiento para mejorar el efecto de eliminación de ruido en [17].

2.2.2 Métodos basados en aprendizaje automático o “Machine Learning”

Basados en extracción de características y clasificación

Estos métodos son aquellos que, para identificar un modelo de cámara, construyen un clasificador basado en Machine Learning para evaluar la proximidad (distancia) entre un modelo previamente aprendido y el vector de características de la imagen de prueba.

Kharrazi et al. [18] (2004) extrajeron 34 características (características de color, métricas de calidad de imagen (IQM) y estadísticas de dominio de wavelet) y las utilizaron para realizar la identificación del modelo de cámara. Celiktutan et al. [19] (2008) utilizaron un subconjunto del conjunto de características de Kharrazi además de características de medidas de similitud binarias para identificar la cámara de origen. Gloe et al. [20] (2012) usaron el conjunto de características de Kharrazi con características extendidas de color para identificar los modelos de cámara.

Bayram et al. [21] (2006) propusieron un método de identificación de cámara determinando las estructuras de correlación presentadas en cada banda de color en relación con la interpolación de la matriz de filtro de color CFA. Swaminathan et al. [22] (2007) también construyeron un identificador de cámara eficiente a través de la estimación de los coeficientes de interpolación CFA.

Filler et al. [23] (2008) introdujeron un método de identificación del modelo de cámara a partir de las características de los momentos estadísticos y las correlaciones del patrón lineal PRNU. Wahab et al. [24] (2012) usaron la probabilidad condicional como un conjunto único de características para clasificar los modelos de cámara.

Xu y Shi [25] (2012) también propusieron la identificación de la cámara utilizando el aprendizaje automático a través de patrones binarios locales (local binary patterns, LBP) como características. A diferencia de [25], el trabajo [26] (2016) también investigó la capacidad discriminativa de la cuantificación de fase local (local phase quantization, LPQ), un descriptor de textura tipo LBP, para distinguir modelos de cámaras. Las características de textura combinadas de LBP y LPQ dieron como resultado una mayor precisión de identificación en comparación con [25].

Hu et al. [27] (2012) desarrollaron un algoritmo mejorado que utiliza trazas de interpolación cromática inter-canal (inter-channel demosaicing traces) para la identificación del modelo de cámara.

Chen et al. [28] (2015) construyeron un modelo de características de 1372 dimensiones para identificar el modelo de la cámara fuente de una imagen. Utilizaron matrices de co-ocurrencia, las cuales muestran un resumen de la forma en que los valores de los pixeles ocurren al lado de otro valor en una pequeña ventana, para capturar el error reconstruido entre la imagen original y la versión reconstruida.

Marra et al. [29] (2015) investigan sobre el uso de características basadas en el análisis de residuos de imágenes (partes brillantes de la imagen adquirida anteriormente son visibles como imagen débil y borrosa en las siguientes exposiciones). En particular, las características las extrajeron localmente en base a matrices de co-ocurrencia de vecinos seleccionados y luego las usaron para entrenar un clasificador SVM (Support Vector Machine).

Tuama et al. [30] (2015) presentaron un nuevo enfoque de identificación de modelo de cámara utilizando el residuo de ruido extraído de una imagen mediante la aplicación de un filtro de eliminación de ruido basado en *wavelets* en un marco de aprendizaje automático. Se refieren a este ruido residual como el ruido contaminado (POL-PRNU), ya que contiene una señal PRNU contaminada con otros tipos de ruido, como el contenido de la imagen. Extrajeron por tanto estadísticas de alto orden de POL-PRNU mediante el cálculo de la matriz de co-ocurrencia, además de características relacionadas con la interpolación de CFA. Estos dos conjuntos de características alimentan a un clasificador SVM para realizar una identificación de modelo de cámara. Estos mismos autores también desarrollaron en [31] (2016) un método para la identificación del modelo de cámara digital mediante la extracción de tres conjuntos de características: matrices de co-ocurrencia, rasgos de dependencias de color relacionadas con la interpolación de CFA y estadísticas de probabilidad condicional, utilizando posteriormente también un clasificador SVM.

Basados en Redes Neuronales Convolucionales

Todos los métodos mencionados anteriormente se basan en procedimientos definidos manualmente para exponer las huellas que caracterizan diferentes modelos de cámaras. Es por ello por lo que surgen estos nuevos métodos revirtiendo el paradigma al resolver un problema de identificación de cámara utilizando una metodología *data-driven*. Esto significa que el objetivo es aprender las características que caracterizan las imágenes tomadas con diferentes cámaras directamente desde las imágenes, en lugar de imponer cualquier modelo o receta de características hechas a mano.

Los algoritmos de aprendizaje profundo (Deep Learning) son uno de los campos de investigación prometedores en la extracción automatizada de características a altos niveles de abstracción. En concreto las redes neuronales convolucionales (*convolutional neural networks*, CNNs), potentes en aprendizaje de características de forma automática, han tenido un gran éxito en tareas de visión artificial y se han desarrollado rápidamente desde 2012 [32, 33, 34, 35]. Estos logros han atraído la atención de la comunidad de investigadores forenses de imágenes digitales y se han realizado varios trabajos aplicando CNNs para resolver problemas forenses.

Chen et al. [36] (2015) agregaron una capa de preprocesamiento antes de las CNNs para el análisis forense. Este preprocesamiento logró un aumento significativo en el rendimiento. Bayer et al. [37] (2016) propusieron una nueva capa convolucional, que es similar a una capa de preprocesamiento como parte de las CNNs. Tuama et al. [38] (2016) presentaron una estructura CNN equipada con una capa de filtro paso alto (HPF) para hacer frente a la identificación del modelo de cámara. Sus resultados experimentales mostraron el importante papel que desempeña la función de preprocesamiento en la precisión de la clasificación.

Baroffio et al. [39] (2017) aplicaron las CNNs para identificar las cámaras de origen, pero el bajo rendimiento indicó que las CNNs diseñadas no se podían adaptar directamente a la identificación de la cámara. Para mejorarlo, sobre esa base propusieron un nuevo enfoque para tratar con la identificación de la cámara. Lo que hicieron fue emplear las CNNs solamente para extraer características y combinarlas después con clasificadores SVM para completar las tareas de clasificación.

Emplear redes neuronales convolucionales para afrontar el problema de identificación de cámara es la técnica que se implementará en este trabajo. Es por ello por lo que se dedica el capítulo siguiente a explicar toda la teoría que hay detrás de las CNNs, además de explicar con más detalle conceptos que han sido nombrados anteriormente como *Machine Learning* y *Deep Learning.*

3 DEEP LEARNING PARA ANÁLISIS DE IMÁGENES

Deep Learning [40] es el núcleo de este trabajo, elegido por sus altas capacidades y número de posibilidades para el análisis de imágenes. Como se verá a lo largo del capítulo, se trata de una clase de algoritmos de Machine Learning basados en redes neuronales que se caracterizan por un procesamiento de los datos en cascada.

Pero antes de continuar y exponer toda la teoría necesaria para comprender en qué consiste el Deep Learning, se aclararán los conceptos de Inteligencia Artificial y Machine Learning, pues dan lugar a este tipo de *aprendizaje profundo*.

La siguiente figura resume visualmente la idea intuitiva de que Deep Learning es solo una parte de la inteligencia artificial, aunque en estos momentos quizás es la más dinámica y la que está haciendo realmente vibrar a la comunidad científica.

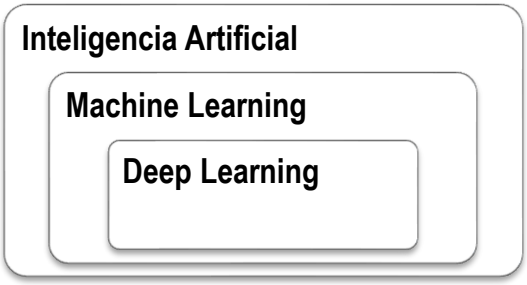


Figura 3-1. Deep Learning como parte de Machine Learning e Inteligencia Artificial

Inteligencia Artificial

Una extensa y precisa definición se encuentra en el libro *Artificial Intelligence: A Modern Approach* [41], el texto sobre inteligencia artificial más popular en el mundo universitario. Pero intentando hacer una aproximación más generalista, se podría aceptar una definición simple en la que por inteligencia artificial nos referimos a aquella inteligencia que muestran las máquinas, en contraste con la inteligencia natural de los humanos. En este sentido, una posible definición concisa y general de inteligencia artificial podría ser el esfuerzo para automatizar tareas intelectuales normalmente realizadas por humanos.

Machine Learning

El aprendizaje automático o "Machine Learning" [42] es en sí mismo un gran campo de investigación y desarrollo. En concreto, Machine Learning se podría definir como el subcampo de la inteligencia artificial que proporciona a los ordenadores la capacidad de aprender automáticamente sin ser explícitamente programados, es decir, sin que necesiten que el programador indique las reglas que debe seguir para lograr su tarea.

Generalizando, se puede decir que Machine Learning consiste en desarrollar para cada problema un algoritmo de predicción para un caso de uso particular. Estos algoritmos aprenden de grandes volúmenes de datos con el fin de encontrar patrones o tendencias para comprender qué nos dicen los datos y de esta manera construir un modelo para predecir y clasificar los elementos.

Dada la madurez del área de investigación en Machine Learning, existen muchos enfoques bien establecidos para el aprendizaje automático por parte de máquinas. Cada uno de ellos utiliza una estructura algorítmica diferente para optimizar las predicciones basadas en los datos recibidos. Machine Learning es un amplio campo con una compleja taxonomía de algoritmos que se agrupan, en general, en dos categorías: aprendizaje supervisado y no supervisado.

- Nos referimos a que el **"aprendizaje es supervisado"** cuando los datos que usamos para el entrenamiento incluyen la solución deseada, llamada "etiqueta". Algunos de los algoritmos más populares de Machine Learning en esta categoría son la regresión lineal, la regresión logística, *Support Vector Machines* (SVM), *decision trees, random forest* y redes neuronales.
- En cambio, cuando nos referimos a un **"aprendizaje no supervisado"** los datos de entrenamiento no incluyen las etiquetas, y será el algoritmo el que intentará clasificar la información por sí mismo. Algunos de los algoritmos más conocidos de esta categoría son *clustering* (*K-means*) o *Principal Component Analysis* (PCA). Este tipo de redes están aún en desarrollo pero apuntan a jugar un papel principal en las ciencias de la computación al no depender de intervención humana suministrando datos.

3.1 Redes Neuronales Artificiales

Como se ha comentado anteriormente, un caso especial de algoritmos de Machine Learning son las redes neuronales artificiales (ANNs) [43].

3.1.1 Arquitectura

En el caso concreto de Deep Learning, las estructuras algorítmicas antes mencionadas permiten modelos que están compuestos de múltiples capas de procesamiento para aprender representaciones de datos, con múltiples niveles de abstracción que realizan una serie de transformaciones lineales y no lineales que a partir de los datos de entrada generen una salida próxima a la esperada. El aprendizaje supervisado, en este caso, consiste en obtener los parámetros de esas transformaciones, y consigue que esas transformaciones sean óptimas, es decir, que la salida producida y la esperada difieran muy poco.

Una aproximación gráfica simple a una red neuronal Deep Learning es la siguiente:

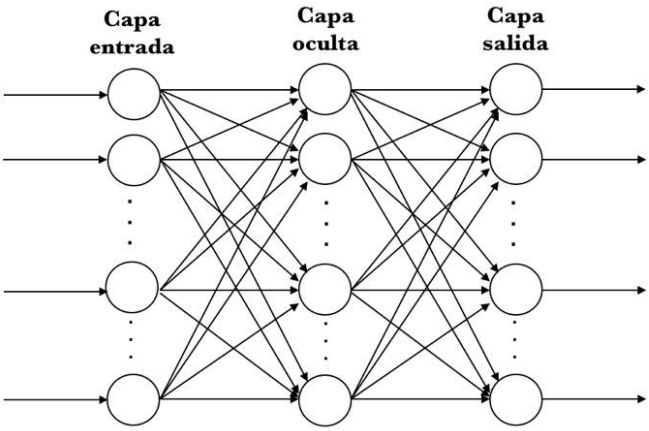


Figura 3-2. Ejemplo de red neuronal simple

Aquí se representa una red neuronal artificial con 3 capas de neuronas: una de entrada (*input layer*) que recibe los datos de entrada y una de salida (*output layer*) que devuelve la predicción realizada. Las capas de en medio se llaman capas ocultas (*hidden layers*) y puede haber tantas como se desee, cada una

con distinta cantidad de neuronas. Normalmente todas las neuronas de cada capa tienen una conexión con cada neurona de la siguiente capa, como se representa en el diagrama (capas densamente conectadas).

En general, hoy en día se manejan redes neuronales artificiales con muchísimas capas, que literalmente están apiladas una encima de la otra; de aquí el concepto de deep (profundidad de la red), donde cada una de ellas está a su vez compuesta por muchísimas neuronas, cada una con sus parámetros que se detallarán más adelante que, a su vez, realizan una transformación simple de los datos que reciben de neuronas de la capa anterior para pasarlos a las de la capa posterior. La unión de todas permite descubrir patrones complejos.

En cuanto al tipo de conexión, son de especial importancia las redes unidireccionales, cuyos enlaces no forman bucles. Son comunes al facilitar el estudio de las mismas y evitan además comportamientos caóticos, por lo que en aplicaciones finales son las más utilizadas.

Para comprender mejor todo esto, es mejor centrarse en el tipo de red más simple (una sola capa de una neurona, con varias entradas y una salida), también comprendida como la unidad básica funcional de una red neuronal, llamada perceptrón:

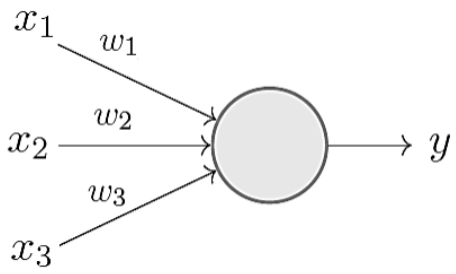


Figura 3-3. Esquema simple de un perceptrón

Como se aprecia en la figura anterior, a través de las entradas la neurona recibe valores (x_i), los cuales tienen un peso (w_i). El concepto de peso es de suma importancia, pues son las variables a encontrar en el entrenamiento de la red y su objetivo es ponderar las entradas de la neurona. De esta manera, la neurona se encarga de sumar todas sus entradas multiplicadas por sus respectivos pesos.

3.1.2 Función de activación

Hasta el momento, la operación que realiza la neurona es sencilla (productos y sumas). Hay otra operación que realizan todas las neuronas, se trata de la función de activación. Esta función recibe como entrada el sumatorio anterior y transforma el valor mediante una fórmula, produciendo un nuevo número.

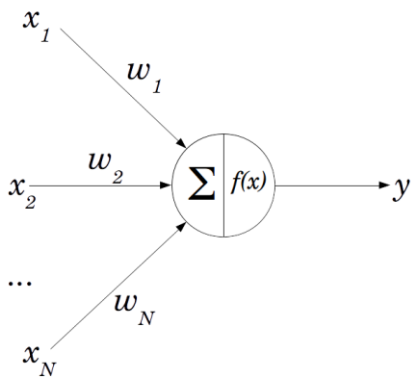


Figura 3-4. Esquema completo de un perceptrón

Es importante mencionar que si se elige una función de activación lineal, no merece la pena utilizar capas ocultas en absoluto, porque se puede comprobar que la potencia de la red será la misma por muchas capas que pongamos. Además, si la función de activación es lineal, la red estará limitada a resolver problemas lineales (muy simples). La potencia de la red sólo aumenta con el número de capas para funciones de activación no lineales, como por ejemplo la sigmoide.

Otro de los objetivos de la función de activación es mantener los números producidos por cada neurona dentro de un rango razonable (sigmoide: números reales entre 0 y 1).

Algunas de las funciones de activación más utilizadas, son las siguientes:

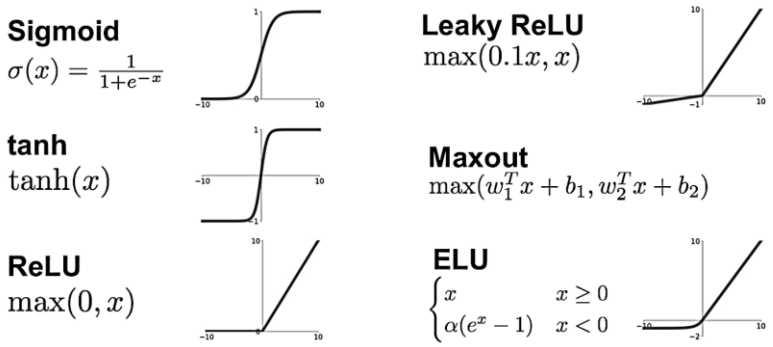


Figura 3-5. Ejemplos de funciones de activación

Las ANNs son a menudo usadas para clasificación, y en concreto cuando las clases son exclusivas, como por ejemplo en la clasificación de imágenes de dígitos (en clases de 0 hasta 9), la capa de salida es una función ***softmax*** en la que la salida de cada neurona corresponde a la probabilidad estimada de la clase correspondiente. Visualmente se podría representar de la siguiente forma:

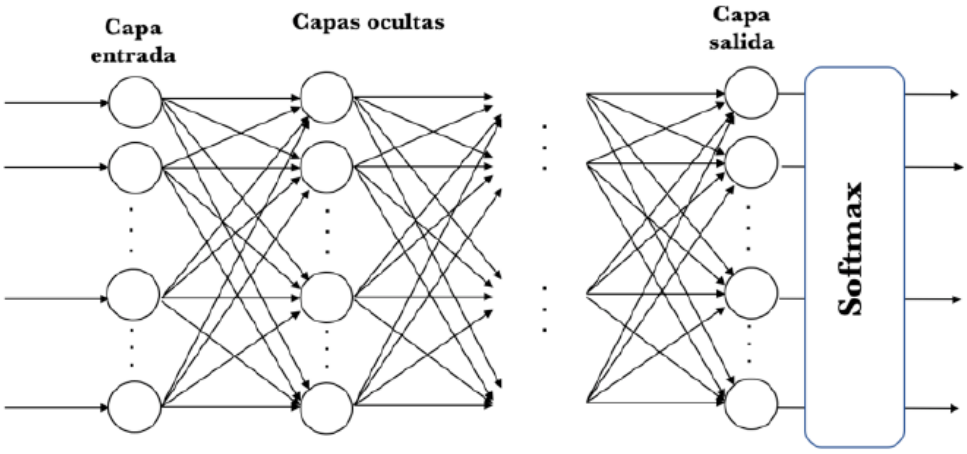


Figura 3-6. Red neuronal terminada con función de activación *Softmax*

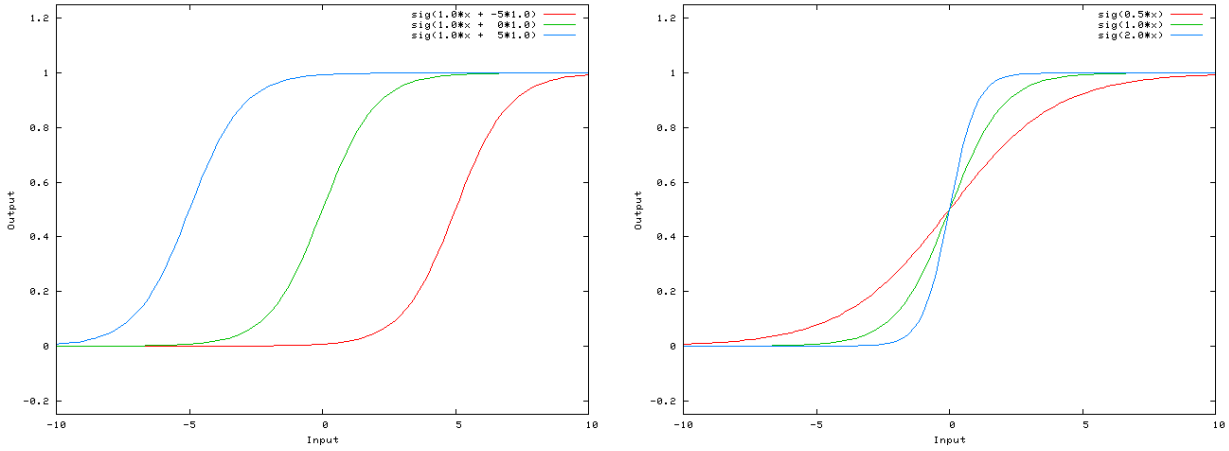
La función *softmax* (ver ecuación 3.1) se basa en calcular "las evidencias" de que una determinada imagen pertenece a una clase en particular y luego se convierten estas evidencias en probabilidades de que pertenezca a cada una de las posibles clases. Para medir la evidencia de que una determinada imagen pertenece a una clase en particular, una aproximación consiste en realizar una suma ponderada de la evidencia de pertenencia de cada uno de sus píxeles a esa clase. Lo interesante de esta función es que una buena predicción devolverá un vector con solo un valor cercano a uno, mientras que las entradas restantes estarán cerca de cero. La expresión matemática para calcular dicha función es la siguiente:

$$f_{softmax}(x_i) = \frac{e(x_i)}{\sum_{j=0}^{k} e(x_j)} \quad , \qquad i = 0,1,2 \dots k \tag{3.1}$$

Como se ha comentado anteriormente, la principal ventaja de usar softmax es el rango de probabilidades de salida. El rango será de 0 a 1 y la suma de todas las probabilidades será igual a uno. De esta forma, si la función es utilizada para clasificación multiclase, devolverá las probabilidades de pertenencia a cada una de las clases. Por ejemplo dado un vector de entrada $[1, 2, 3, 4, 1, 2, 3]$, la función softmax devolverá $[0.024, 0.064, 0.175, 0.475, 0.024, 0.064, 0.175]$. Una vez se obtiene este vector es muy fácil decir que la imagen pertenece a la clase 3 (si suponemos que van desde la 0 hasta la 6) con una probabilidad de 47.5%.

3.1.3 Sesgo

Justo antes de aplicar la función de activación, cada neurona añade a la suma de productos un nuevo término constante, llamado habitualmente sesgo (b). Una forma típica de implementar este término consiste en imaginarse que se extiende la capa anterior con una falsa neurona que siempre toma como valor un 1.0, e incorporar los pesos correspondientes a dicha falsa neurona a la matriz de pesos. Al introducir dicho término se consigue desplazar la función de activación hacia la izquierda o hacia la derecha (Figura 3-7 a), lo que puede ser crítico para el aprendizaje exitoso, pues con la variación de los pesos (w_i) solo se puede modificar la "inclinación" de la función (Figura 3-7 b).



a) b)

Figura 3-7. a) Variación del sesgo. b) Variación de los pesos

Finalmente, el esquema del perceptrón completo sería el siguiente:

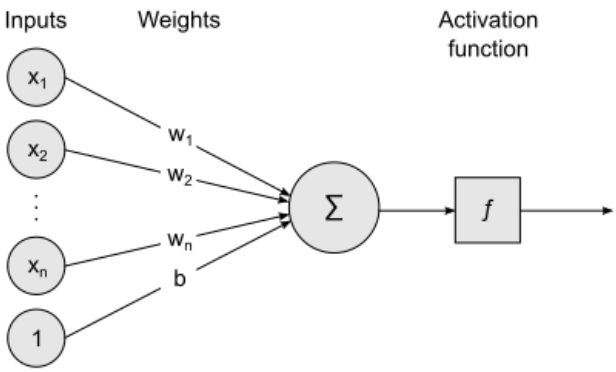


Figura 3-8. Esquema completo de un perceptrón incluyendo el sesgo

Es importante mencionar que aunque el perceptrón es un ejemplo de red como tal, su alcance está limitado a realizar operaciones básicas booleanas (pues solo consta de una neurona). Conforme aumenta la complejidad de un problema, lo hace del mismo modo la red necesaria para resolverlo. Es un reto importante el diseño de una red, ya que se necesita determinar el número de neuronas necesarias y cómo deben ser las conexiones que existen entre ellas. Surge en este momento el concepto de capa. Al aumentar el número de capas de neuronas, se aumenta el número de operaciones que la red puede realizar y la capacidad de ajustarse al problema.

Llegados a este punto, se puede observar que las redes son herramientas de gran utilidad para procesar valores a partir de un proceso de entrenamiento y que generan un resultado que trata de ajustarse a lo aprendido de la experiencia. Sin embargo, el poder de las redes neuronales va más allá, sobre todo gracias a la aparición de arquitecturas de redes neuronales convolucionales (también conocidas como CNNs), las cuales no se limitan a recibir cantidades de entrada sino que permiten trabajar con imágenes y están provocando importantes avances en los campos de la visión artificial y el procesamiento del audio.

3.2 Redes Neuronales Convolucionales

Hay aplicaciones en las que el diseño tradicional de las redes neuronales artificiales no funciona adecuadamente, como es el caso de la clasificación de imágenes:

→ Por ejemplo para una pequeña imagen de entrada a color de tamaño 32x32 (3 valores por píxel), una sola neurona de la primera capa necesitaría $32 * 32 * 3 + 1 = 3.073$ parámetros.

→ Si la imagen es algo más normal, 200x200 por ejemplo, una sola neurona de la primera capa necesitaría $200 * 200 * 3 + 1 = 120.001$ parámetros.

Como vemos, el número de parámetros requeridos es muy elevado, aún más cuanto más aumente el número de capas ocultas y de neuronas en ellas (alto coste computacional).

En estos casos sería una ventaja tener en cuenta la disposición espacial de las diferentes dimensiones de los datos de entrada. Las redes neuronales convolucionales (CNNs) [44] son redes neuronales artificiales: están formadas por neuronas que tienen parámetros en forma de pesos y sesgos que se pueden aprender. Pero un rasgo diferencial de las CNNs es que hacen la suposición explícita de que las entradas son imágenes, cosa que permite codificar ciertas propiedades en la arquitectura.

Su principal ventaja es que cada parte de la red es entrenada para realizar una tarea, aprendiendo diferentes niveles de abstracción, por lo que se reduce significativamente el número de capas ocultas y el entrenamiento es más rápido. Las CNNs son muy potentes para todo lo que tiene que ver con el análisis de imágenes, debido a que son capaces de detectar características simples como por ejemplo detección de bordes, líneas, etc. y componer en características más complejas hasta detectar lo que se busca.

Como se verá a continuación, los componentes básicos de una red neuronal son las operaciones de convolución y *pooling*.

3.2.1 Convolución

La diferencia fundamental entre una capa densamente conectada (como las capas de las ANNs) y una capa especializada en la operación de convolución, llamada capa convolucional, es que la capa densa aprende patrones globales en su espacio global de entrada, mientras que las capas convolucionales aprenden patrones locales en pequeñas ventanas de dos dimensiones.

De manera intuitiva, se podría decir que el propósito principal de una capa convolucional es detectar características o rasgos visuales en las imágenes como aristas, líneas, etc. Esta es una propiedad muy interesante, porque una vez aprendida una característica en un punto concreto de la imagen la puede reconocer después en cualquier parte de la misma. En cambio, en una red neuronal densamente conectada tiene que aprender el patrón nuevamente si este aparece en una nueva localización de la imagen.

Otra característica importante es que las capas convolucionales pueden aprender jerarquías espaciales de patrones preservando relaciones espaciales. Por ejemplo, una primera capa convolucional puede aprender elementos básicos como aristas, y una segunda capa convolucional puede aprender patrones compuestos de elementos básicos aprendidos en la capa anterior. Y así sucesivamente hasta ir aprendiendo patrones muy complejos. Esto permite que las redes neuronales convolucionales aprendan eficientemente conceptos visuales cada vez más complejos y abstractos.

El funcionamiento de dicha operación de convolución se explicará mediante un ejemplo. Como entrada en la red neuronal se tendrá una imagen en escala de grises de 28x28, por lo que se tendrá un espacio de neuronas en la capa de entrada de tantas neuronas como píxeles tiene la imagen (cada píxel se considera una neurona). Una primera capa de neuronas ocultas conectadas a las neuronas de la capa de entrada realizará las operaciones convolucionales. Pero como se ha explicado con anterioridad, no se conectan todas las neuronas de entrada con todas las neuronas de este primer nivel de neuronas ocultas, como en el caso de las redes neuronales densamente conectadas; solo se hace por pequeñas zonas localizadas del espacio de las neuronas de entrada que almacenan los píxeles de la imagen.

Siguiendo con el ejemplo, cada neurona de la capa oculta será conectada a una pequeña región de 5×5 neuronas (es decir 25 neuronas) de la capa de entrada (de 28×28). Intuitivamente, se puede pensar en una ventana del tamaño de 5×5 que va recorriendo toda la capa de 28×28 de entrada que contiene la imagen. Esta ventana va deslizándose (de izquierda-derecha, de arriba-abajo) a lo largo de toda la capa de neuronas. Por cada posición de la ventana hay una neurona en la capa oculta que procesa esta información.

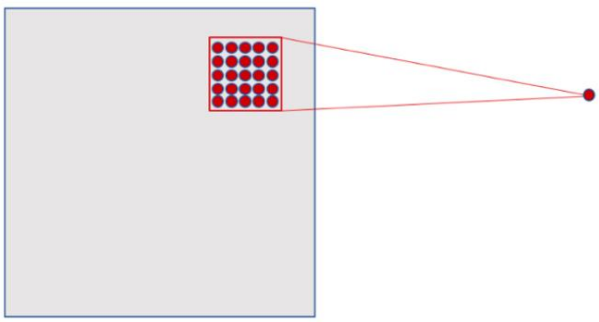


Figura 3-9. Operación de convolución

Analizando un poco este caso concreto, se observa que si se tiene una entrada de 28×28 píxeles y una ventana de 5×5, esto nos define un espacio de 24×24 neuronas en la primera capa del oculta, debido a que solo se puede mover la ventana 23 neuronas hacia la derecha y 23 hacia abajo. El paso de avance de la ventana (parámetro llamado *stride*) no tiene por qué ser siempre de 1, sino que es un valor ajustable como se verá en el próximo capítulo. En las CNNs también se puede aplicar una técnica de relleno de ceros alrededor del margen de la imagen para mejorar el barrido que se realiza con la ventana que se va deslizando. El parámetro para definir este relleno recibe el nombre de *padding*. En el siguiente capítulo se detallará el valor que se le asigna a dicho parámetro y la implicación que ello supone.

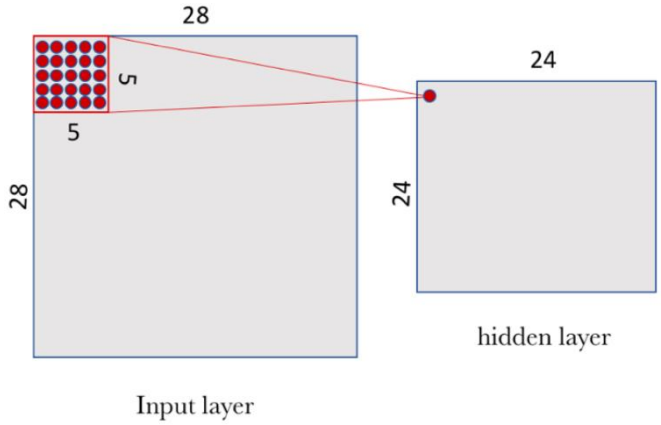


Figura 3-10. Ejemplo concreto de convolución

Para conectar cada neurona de la capa oculta con las 25 neuronas que le corresponden de la capa de entrada se usa de la misma forma que en las ANNs un valor de sesgo *b* y una matriz de pesos *W* de tamaño 5×5 llamada filtro o *kernel*. El valor de cada punto de la capa oculta corresponde al producto escalar entre el filtro y el conjunto de 25 neuronas (5×5) de la capa de entrada.

Ahora bien, lo particular y muy importante de las CNNs es que se usa el mismo filtro (la misma matriz *W* de pesos y el mismo sesgo *b*) para todas las neuronas de la capa oculta: en este caso para las 24×24 neuronas (576 neuronas en total) de la primera capa. Se puede ver por tanto que esta compartición reduce de manera drástica el número de parámetros que tendría una red neuronal si no la hiciéramos: pasa de 14.400 parámetros que tendrían que ser ajustados (5×5×24×24) a 25 (5×5) parámetros más los sesgos.

Hay que destacar que un filtro definido por una matriz *W* y un sesgo *b* solo permiten detectar una característica concreta en una imagen; por tanto, para poder realizar el reconocimiento de imágenes se propone usar varios filtros a la vez, uno para cada característica que queramos detectar. Por eso una capa convolucional completa en una CNN incluye varios filtros.

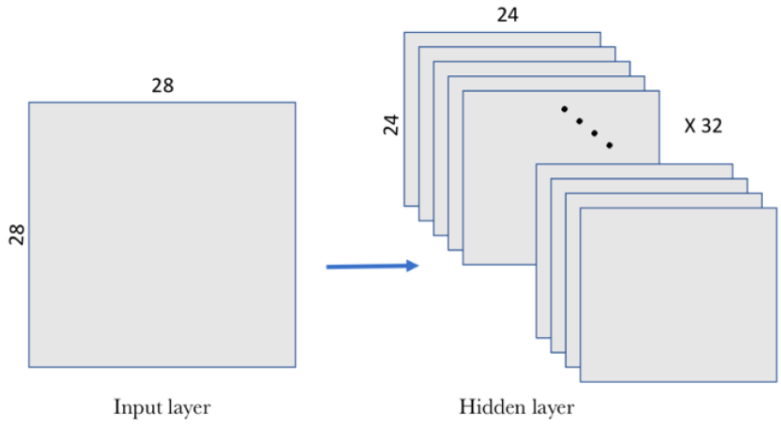


Figura 3-11. Capa convolución completa compuesta por varios filtros

Una manera habitual de representar visualmente esta capa convolucional es la que se muestra en la figura anterior, donde el nivel de capas ocultas está compuesto por 32 filtros (cada uno de ellos con una matriz *W* de pesos compartida de 5×5 y un sesgo *b*).

Finalmente, la primera capa convolucional recibe una entrada de tamaño (28, 28, 1) y genera una salida de tamaño (24, 24, 32), que contiene las 32 salidas de 24×24 píxel resultado de computar los 32 filtros sobre la entrada.

3.2.2 Pooling

Además de las capas convolucionales que se acaban de describir, las redes neuronales convolucionales acompañan a la capa de convolución con unas capas de *pooling*, que suelen ser aplicadas inmediatamente después de las capas convolucionales. Una primera aproximación para entender para qué sirven estas capas es ver que las capas de pooling hacen una simplificación de la información recogida por la capa convolucional y crean una versión condensada de la información contenida en estas. Con ello se consigue reducir la cantidad de neuronas antes de hacer una nueva convolución, ya que si se hiciera una nueva convolución a partir de la capa oculta obtenida, el número de neuronas de la próxima capa seguiría siendo muy elevado (y ello implica mayor procesamiento).

Para reducir el tamaño de la próxima capa de neuronas haremos un proceso de *subsampling* en el que reduciremos el tamaño de nuestras imágenes filtradas pero manteniendo las características más importantes que detectó cada filtro.

Siguiendo el ejemplo anterior, se va a escoger una ventana de 2×2 de la capa convolucional sintetizando la información en un punto en la capa de *pooling*. Visualmente, se puede expresar de la siguiente manera:

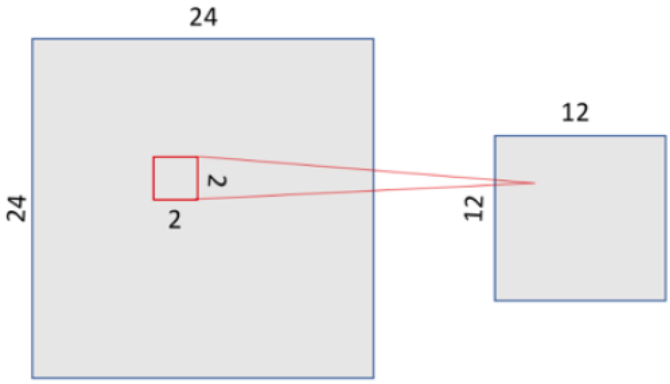


Figura 3-12. Ejemplo de pooling

Hay varias maneras de condensar la información, pero una habitual y que usaremos en nuestro ejemplo es la conocida como *max-pooling*, que como su nombre indica se queda con el valor máximo de los que había en la ventana de entrada de 2×2 en nuestro caso. Con ello se consigue dividir por 4 el tamaño de la salida de la capa de *pooling*, quedando una imagen de 12×12.

También se puede utilizar *average-pooling* en lugar *max-pooling*, donde cada grupo de puntos de entrada se transforma en el valor promedio del grupo de puntos en vez de su valor máximo. Pero en general el *max-pooling* tiende a funcionar mejor que las soluciones alternativas.

Tal como se ha mencionado anteriormente, la capa convolucional alberga más de un filtro, y por tanto, como se aplica el *max-pooling* a cada uno de ellos separadamente, la capa de *pooling* contendrá tantos filtros de *pooling* como de filtros convolucionales:

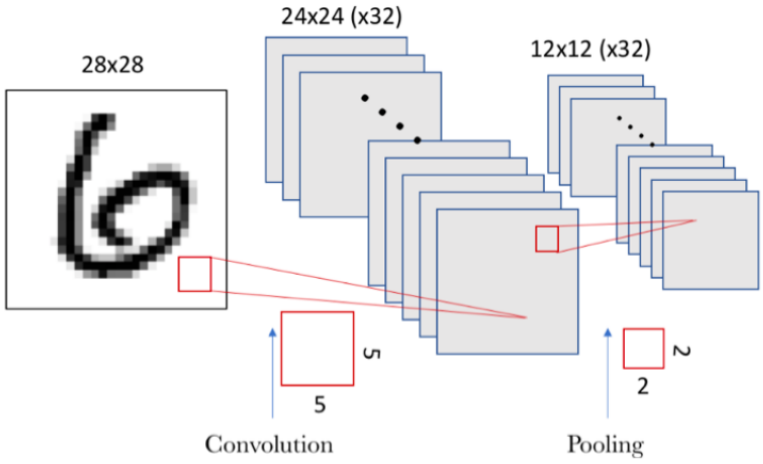


Figura 3-13. Ejemplo de capa de convolución seguida de capa de pooling

El resultado es, dado que se tenía un espacio de 24×24 neuronas en cada filtro convolucional, después de hacer el *pooling* se tendrá 12×12 neuronas que corresponde a las 12×12 regiones de tamaño 2×2 que aparecen al dividir el espacio de neuronas del espacio del filtro de la capa convolucional.

Una vez añadidas todas las capas de convolución y *pooling* deseadas, el último paso consiste en añadir una capa densamente conectada, que servirá para alimentar una capa final de *softmax* como la introducida en el apartado anterior para hacer la clasificación:

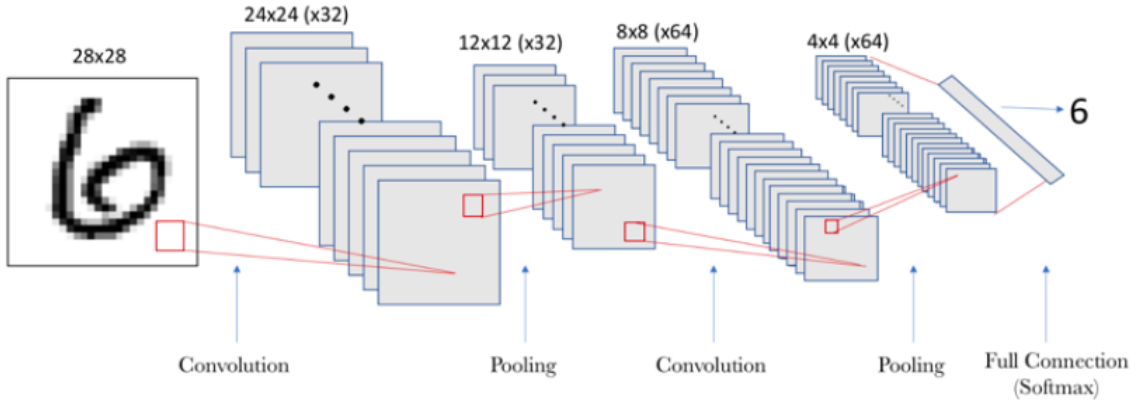


Figura 3-14. Conjunto de capas convolución + pooling terminada en una densamente conectada

3.3 Entrenamiento de una red neuronal

Conocidos los resultados de un determinado problema y los datos de partida, el proceso de entrenamiento consiste en hallar los pesos que permiten que con los valores de entrada, se obtenga la salida esperada. Recordemos que una red neuronal está formada de neuronas conectadas entre ellas; a su vez, cada conexión de nuestra red neuronal está asociada a un peso que dictamina la importancia que tendrá esa relación en la neurona al multiplicarse por el valor de entrada.

Como se puede observar, para entrenar una red que consiga predecir un resultado a partir de unos datos, es necesario previamente tener como entradas ciertos valores de un problema y sus respectivas salidas (entrenamiento supervisado). De esta forma, la red ajustará poco a poco sus pesos de modo que intente que ante todas las entradas, la salida sea la correcta.

3.3.1 Proceso de aprendizaje

Entrenar una red neuronal es la parte más genuina de Deep Learning. Este proceso de aprendizaje se puede ver como un proceso iterativo de "ir y venir" por las capas de neuronas: *forwardpropagation* y *backpropagation* [45].

La primera fase *forwardpropagation* se da cuando se expone la red a los datos de entrenamiento y estos cruzan toda la red neuronal para ser calculadas sus predicciones. Es decir, se pasan los datos de entrada a través de la red de tal manera que todas las neuronas apliquen su transformación a la información que reciben de las neuronas de la capa anterior y la envíen a las neuronas de la capa siguiente. Cuando los datos hayan cruzado todas las capas, y todas sus neuronas han realizado sus cálculos, se llegará a la capa final con un resultado de predicción para aquellos ejemplos de entrada.

A continuación usaremos una función de pérdida (*loss*) para estimar el error y para comparar y medir cuán bueno/malo fue nuestro resultado de la predicción en relación con el resultado correcto (recordemos que estamos en un entorno de aprendizaje supervisado y disponemos de la etiqueta que nos indica el valor esperado). Idealmente, queremos que nuestro error sea cero, es decir, sin divergencia entre valor estimado y el esperado. Por eso a medida que se entrena el modelo se irán ajustando los pesos de las interconexiones de las neuronas de manera automática hasta obtener buenas predicciones.

Una vez se tiene calculado el error, se propaga hacia atrás esta información. De ahí su nombre, retro-propagación, en inglés *backpropagation*. Partiendo de la capa de salida, esa información de pérdida se propaga hacia todas las neuronas de la capa oculta que contribuyen directamente a la salida. Sin embargo las neuronas de la capa oculta solo reciben una fracción de la señal total del error, basándose aproximadamente en la contribución relativa que haya aportado cada neurona a la salida original. Este proceso se repite, capa por capa, hasta que todas las neuronas de la red hayan recibido una señal de pérdida que describa su contribución relativa a la pérdida total.

Visualmente, se puede resumir el proceso detallado anteriormente con este esquema visual:

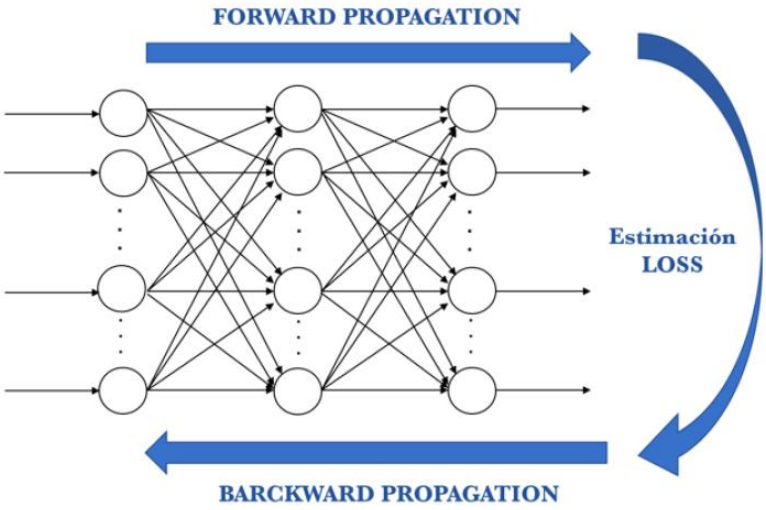


Figura 3-15. Proceso de aprendizaje de una red neuronal

Una vez propagada hacia atrás esta información, se pueden ajustar los pesos de las conexiones entre neuronas. Lo que se pretende conseguir con todo esto es que el error se aproxime lo más posible a cero la próxima vez que se vuelva a usar la red para una predicción. Para ello se usa una técnica llamada *gradient descent*. Esta técnica va cambiando los pesos en pequeños incrementos (*learning rate*) con la ayuda del cálculo de la derivada (o gradiente) de la función de pérdida, cosa que permite ver en qué dirección "descender" hacia el mínimo global; esto lo va haciendo en general en lotes de datos (*batches*) en las sucesivas iteraciones (*epochs*) del conjunto de todos los datos que se le pasan a la red en cada iteración.

Se presentan a continuación en más detalle cada uno de estos últimos conceptos mencionados.

3.3.2 Método del descenso del gradiente (Gradient Descend)

El optimizador *gradient descend* [46] es la base de muchos otros y uno de los algoritmos de optimización más comunes en Machine Learning y Deep Learning.

Gradient descent usa la primera derivada (gradiente) de la función de pérdida cuando realiza la actualización en los parámetros. Recordemos que el gradiente nos da la pendiente de una función en ese punto. El proceso consiste en encadenar las derivadas de la función de pérdida de cada capa oculta a partir de las derivadas de la función de su capa superior, incorporando su función de activación en el cálculo (por eso las funciones de activación deben ser derivables). En cada iteración, una vez todas las neuronas disponen del valor del gradiente de la función de pérdida que les corresponde, se actualizan los valores de los parámetros en el sentido contrario a la que indica el gradiente. El gradiente, en realidad, siempre apunta en el sentido en el que se incrementa el valor de la función. Por tanto, si se usa el negativo del gradiente podemos se puede conseguir el sentido en que se tiende a reducir la función de pérdida.

De manera visual, el proceso sería el siguiente, donde línea representa los valores que toma la función de pérdida para cada posible parámetro y en el punto inicial indicado el negativo del gradiente se indica por la flecha:

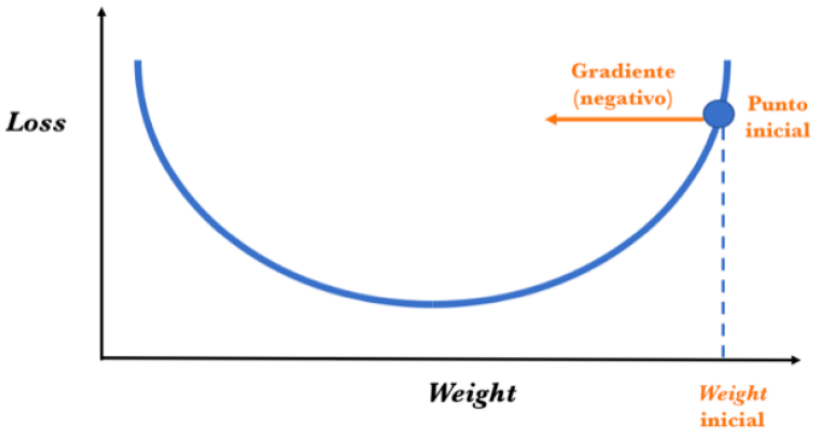


Figura 3-16. Punto inicial del optimizador *gradient descend*

Para determinar el siguiente valor para el parámetro (para simplificar la explicación, se considerará solo el peso/*weight*), el algoritmo de *gradient descent* modifica el valor del peso inicial para ir en sentido contrario al del gradiente (ya que este apunta en el sentido en que crece la pérdida y queremos reducirla), añadiendo una cantidad proporcional a este. La magnitud de este cambio está determinada por el valor del gradiente y por un hiperparámetro *learning rate* (que se presentará en breve) que podemos especificar. Por lo tanto, conceptualmente es como si se siguiera la dirección de la pendiente cuesta abajo hasta alcanzar un mínimo local:

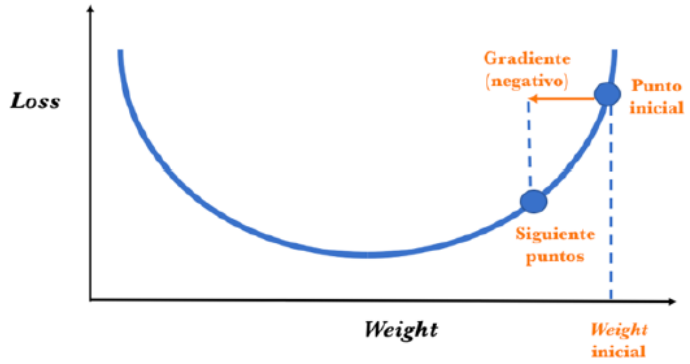


Figura 3-17. Dirección a seguir del optimizador *gradient descend*

El algoritmo *gradient descent* repite este proceso acercándose cada vez más al mínimo hasta que el valor del parámetro llega a un punto más allá del cual no puede disminuir la función de pérdida:

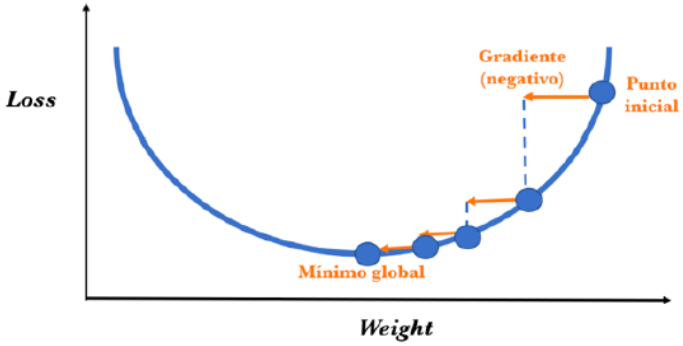


Figura 3-18. Proceso de optimización del *gradient descend*

3.3.3 Hiperparámetros

En primer lugar, se van a explicar las diferencias entre "parámetro" e "hiperparámetro". En general, consideramos un parámetro del modelo como una variable de configuración que es interna al modelo y cuyo valor puede ser estimado a partir de los datos. En cambio, por hiperparámetro nos referimos a variables de configuración que son externas al modelo en sí mismo y cuyo valor en general no puede ser estimado a partir de los datos, y son especificados por el programador para ajustar los algoritmos de aprendizaje.

Existen hiperparámetros tanto a nivel de estructura y topología de la red neuronal (número de capas, número de neuronas, sus funciones de activación, etc) ya comentados anteriormente, como hiperparámetros a nivel de algoritmo de aprendizaje (*learning rate, momentum, epochs, batch size,* etc).

- **Epochs:** indica cada una de las veces en las que todos los datos de entrenamiento pasarán por la red neuronal en el proceso de entrenamiento. Como se presentará más adelante en el siguiente capítulo, una buena pista es incrementar el número de *epochs* hasta que el porcentaje de acierto con los datos de validación empieza a decrecer.

- **Bach size:** se pueden particionar los datos de entrenamiento en mini lotes para pasarlos por la red. Este hiperparámetro indica el tamaño que se usará de estos lotes en cada iteración del entrenamiento para actualizar el gradiente. El tamaño óptimo dependerá de muchos factores, entre ellos de la capacidad de memoria del computador que usemos para hacer los cálculos.

- **Learning rate:** el vector de gradiente tiene una dirección y una magnitud. Los algoritmos de *gradient descent* multiplican la magnitud del gradiente por un escalar conocido como *learning rate* (también denominado a veces *step size*) para determinar el siguiente punto, el cual indica el paso que el algoritmo realiza en el entrenamiento de una red en cada iteración.

  El valor adecuado de este hiperparámetro es muy dependiente del problema en cuestión, pero en general, si este es demasiado grande, se están dando pasos enormes, lo que podría ser bueno para ir rápido en el proceso de aprendizaje, pero es posible que se salte el mínimo y dificultar así que el proceso de aprendizaje se detenga. Contrariamente, si el learning rate es pequeño, se harán avances constantes y pequeños, teniéndose una mejor oportunidad de llegar a un mínimo local, pero esto puede provocar que el proceso de aprendizaje sea muy lento.

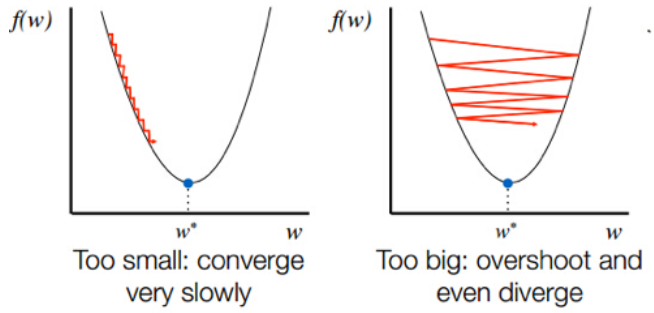


Figura 3-19. Problemas del *learning rate* demasiado pequeño o grande

3.3.4 Resultados

Un entrenamiento puede finalizar con varios resultados posibles:

- **No convergencia:** el algoritmo es incapaz de encontrar una solución y continúa iterando sin fin.
- **Convergencia:** se encuentra una solución al problema, con tres posibilidades, representadas además en la Figura 3-20.
  - **Subajuste (*underfitting*):** se produce cuando el conjunto de entrenamiento no es lo suficientemente representativo de la tarea a aprender. Esto implica que el sistema tenga un error de entrenamiento grande y no que no sea capaz de generalizar en ningún caso.
  - **Sobreajuste (*overfitting*):** se produce cuando la complejidad del sistema de aprendizaje es claramente superior al necesario por el sistema o cuando se intenta minimizar demasiado el error de entrenamiento. En este caso las predicciones sobre la base de datos son muy buenas, pero la capacidad de generalización de la red es pobre, haciendo que ante nuevos datos con los que no ha sido entrenada sea incapaz de realizar una clasificación acertada.
  - **Buen entrenamiento:** la red genera predicciones con mucha precisión pero a su vez ha desarrollado una capacidad de generalización que le permite predecir correctamente ante nuevos datos de entrada.

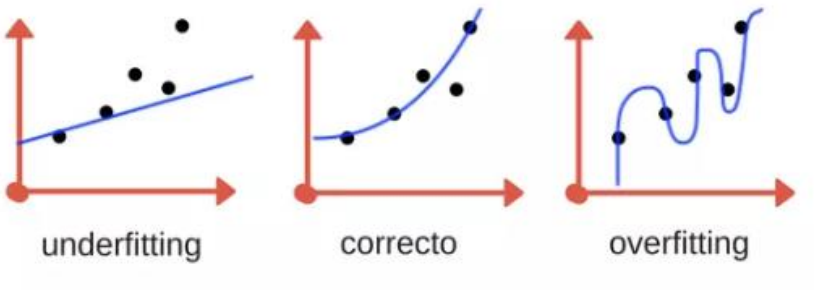


Figura 3-20. Posibilidades de convergencia en un problema

Existen diversas técnicas para evitar las patologías típicas de un mal entrenamiento. Sin embargo, con una correcta elección de parámetros y una base de datos equilibrada, los algoritmos actuales son capaces de alcanzar buenas soluciones.

3.4 Datos para alimentar una red neuronal

Una correcta elección de una base de datos para entrenamiento es vital para llegar hasta los resultados deseados. La red debe conocer de igual manera todos aquellos elementos que se quieren predecir, por lo que los prototipos de entrenamiento deben ser equilibrados para todas las clases a detectar y además se debe disponer de una enorme cantidad de datos. Todo ello conlleva, aunque a costa de un entrenamiento más largo, obtener soluciones aceptables.

3.4.1 Data Augmentation

Dado que es muy importante la construcción de una buena base de datos, existen algunas técnicas para mejorar una base de datos dada y *Data Augmentation* [47] es una de ellas. Consiste en aumentar la base de datos a partir de los prototipos conocidos, modificándolos ligeramente para generar nuevos a partir de ellos. En el ejemplo de las redes neuronales aplicadas al reconocimiento en imágenes, es común generar variaciones de los prototipos realizando rotaciones, traslaciones, aplicando aumentos, variando resolución y ruido y modificando el espacio de color para crear "nuevas" imágenes con las que entrenar la red.

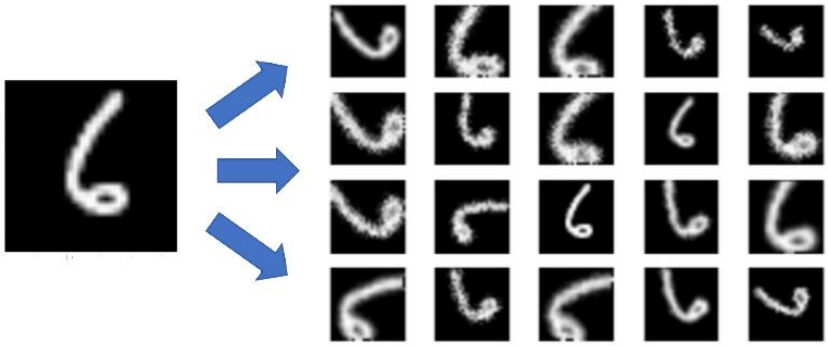


Figura 3-21. Ejemplo de *data augmentation* para una imagen de una base de datos

La eficacia de esta técnica reside en la forma en que las redes neuronales entienden las imágenes y sus características. Si una imagen es modificada ligeramente es percibida por la red como una imagen completamente distinta perteneciente a la misma clase. Esto disminuye las probabilidades de que la red se centre en orientaciones o posiciones mientras mantiene la relevancia de las características deseadas como puede ser el patrón de la cola en este ejemplo.

3.4.2 Datos de entrenamiento y validación

Por último, generalmente los algoritmos de entrenamiento necesitan realizar una separación en la base de datos, usando una parte de ella para el entrenamiento en sí y otra para validar. Es difícil generar una regla para elegir el número de muestras en cada uno de los dos conjuntos, pero es conveniente que el conjunto de datos de entrenamiento sea mayor que el conjunto de test, por ejemplo son típicas las agrupaciones 80-20% o 70-30%:

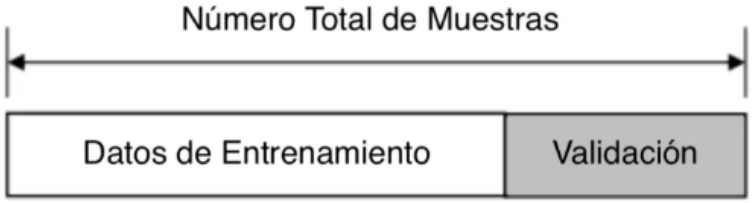


Figura 3-22. Separación de la base de datos en datos de entrenamiento y de validación

La parte de validación no es usada para entrenamiento y sirve para verificar si la red realiza predicciones correctas. De igual forma, tras el proceso se predice sobre la base de datos de validación, para comparar el comportamiento frente a datos que la red "conoce" y datos que nunca ha "visto". De esta comparativa se obtienen conclusiones como la precisión de la red o si existe subajuste o sobreajuste.

4 MÉTODO PROPUESTO

Una vez conocida toda la teoría que hay detrás del Deep Learning, se procede a describir e implementar el método propuesto para el problema inicial presentado. Pero antes, en primer lugar, se detallará todo el material que ha sido utilizado para ello, tanto a nivel de herramientas hardware y software, como la base de datos empleada.

4.1 Material

4.1.1 Hardware

Como se ha comentado anteriormente, el Deep Learning conlleva una gran carga computacional, pues las CNNs suelen estar compuestas por un gran número de capas y neuronas, además del hecho de trabajar con grandes cantidades de imágenes, sobre todo si son de alta resolución. Es por ello que dependiendo del hardware que tengamos, la experiencia cambia considerablemente.

Una GPU (*Graphics Processing Unit*) [48] es un coprocesador dedicado al procesamiento de imágenes y operaciones de coma flotante. Básicamente, al ser otro procesador añadido, su función es la de liberar de carga a la CPU, aumentando el rendimiento de nuestro ordenador. En realidad, los procesadores principales ya tienen una GPU integrada, pero de potencia reducida. Son las tarjetas gráficas las que tienen unas GPU potentes.

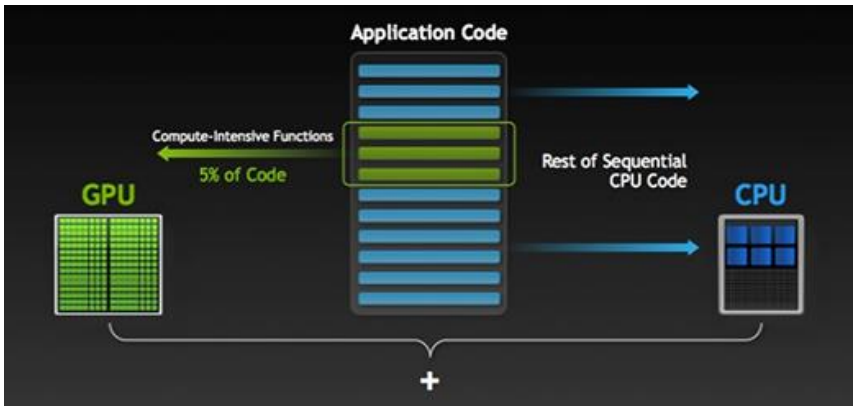


Figura 4-1. Cómo la GPU acelera el trabajo

Tener una buena GPU es fundamental para poder entrenar modelos. Y es que, sin una GPU instalada, entrenar una red neuronal puede llevar días e incluso meses. Con una GPU potente, en cambio, el tiempo de entrenamiento disminuye considerablemente, pudiendo ser de horas en lugar de días, o días en lugar de meses.

Otra alternativa es la computación en la nube, gracias a la aparición de Amazon Web Services (AWS) [49] y de las Tensor Processing Unit (TPU) de Google Cloud Platform [50]. Estos productos ofrecen potencia de computación, permitiendo contratar por tiempo limitado máquinas con soporte en la nube personalizadas. Existen numerosas alternativas de procesador, memoria y GPUs con precios razonables.

Para este trabajo se ha optado por la opción de emplear una GPU, pues se disponía de este componente en el equipo empleado. Las especificaciones técnicas del equipo final donde se desarrolló la totalidad del proyecto, incluyendo el entrenamiento de redes neuronales y su implementación vía software son las siguientes:

Componente	Valor
CPU	Intel Core i7-7700HQ @ 2.80GHz
GPU	NVIDIA GeForce GTX 1070
RAM	16 GB
HHD	1 TB
SSD	128 GB
OS	Windows 10

Tabla 4-1. Especificaciones del equipo

NVIDIA [51] es el líder absoluto en la venta de GPU, pues sus librerías estándar permiten trabajar con CUDA, una plataforma de computación paralela muy potente. Al ser la primera vez que se emplea el equipo para trabajar en este ámbito del Deep Learning, ha sido necesaria la preparación de la GPU [52], instalando tanto CUDA, como su librería especializada en Deep Learning: NVIDIA CUDA Deep Neural Network library (cuDNN). Una guía para descargar tanto CUDA como cuDNN así como para instalar ambas se puede encontrar en la siguiente documentación de NVIDIA [53].

4.1.2 Software

Lenguaje de programación

Python es actualmente el lenguaje de programación de más rápido crecimiento, y la tendencia es que siga por ese camino en un futuro cercano. Tiene una serie de características que lo hacen muy particular y que, sin duda, le aportan muchas ventajas y están en la raíz de su uso tan extendido [54].

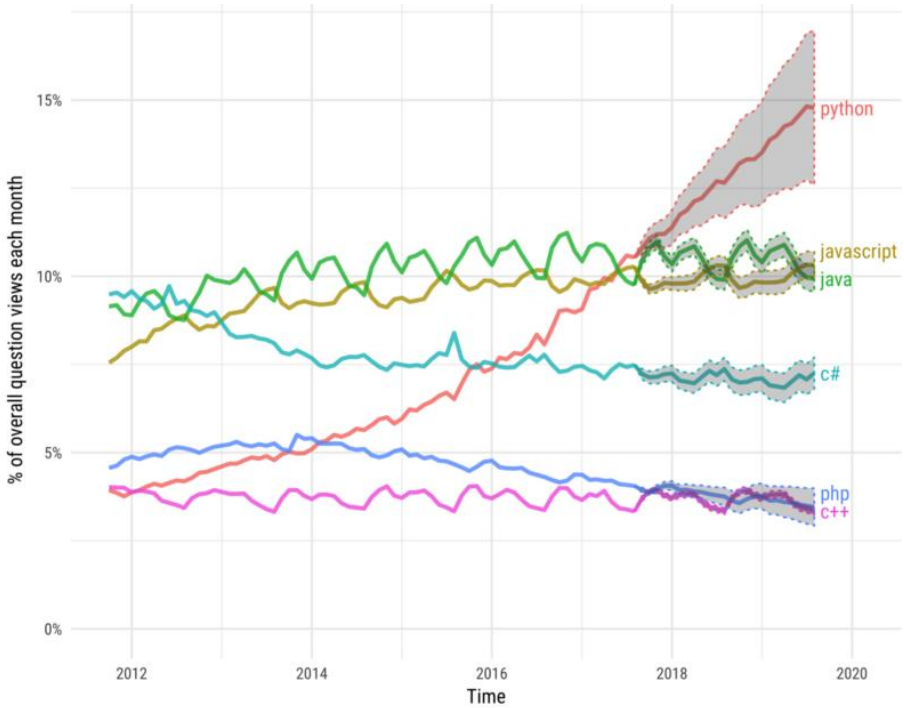


Figura 4-2. Proyección para los próximos años [Fuente: Stackoverflow]

El crecimiento en el uso del lenguaje está siendo espectacular gracias, fundamentalmente, a las nuevas tecnologías de Inteligencia Artificial, Machine Learning y Deep Learning, donde junto con el lenguaje R es el rey. Sin embargo, R es un lenguaje más de nicho que proviene del mundo de la estadística. Python, por otro lado, es un lenguaje de propósito general y su uso está mucho más extendido. Es por ello por lo que Python ha sido el lenguaje seleccionado para este trabajo, además de porque ya estaba previamente familiarizada con él.

Destacar que aunque la última versión de Python es la 3.7, se ha empleado la versión anterior 3.6, al ser la última versión más estable para trabajar con Keras y TensorFlow, que serán presentados a continuación.

Entorno de desarrollo y librerías

El entorno de trabajo utilizado es Spyder [55], un entorno de programación para Python que incluye la distribución multiplataforma Anaconda [56] desarrollada por Continuum Analytics. Esta distribución se caracteriza por ser libre y por incluir una gran colección de paquetes y librerías para análisis de datos, computación científica e ingeniería. Los paquetes y librerías más destacables empleados en el trabajo son los siguientes:

- Numpy [57]: paquete para Python de computación científica. Sus posibilidades son innumerables, permitiendo trabajar con matrices N-dimensionales además de implementar álgebra lineal, transformada de Fourier y muchas funciones relacionadas con estadística.
- Pillow [58]: librería para el manejo de todo tipo de imágenes.
- Matplotlib [59]: librería que permite representación 2D y 3D de funciones e imágenes.
- OS [60]: permite acceder a funcionalidades dependientes del Sistema Operativo. Sobre todo, aquellas que refieren información sobre el entorno del mismo y permiten manipular la estructura de directorios.

Framework empleado para Deep Learning

Los *frameworks* predominantes en el mercado para la implementación de redes neuronales son: TensorFlow [61], Theano [62], Keras [63], Caffe [64] y Pythorch [65].

Entre todos ellos, Keras ha sido el elegido para este trabajo. Es el framework recomendado para principiantes, puesto que su curva de aprendizaje es muy suave en comparación con otras, a la vez que es, sin duda, una de las herramientas para implementar redes neuronales de mayor popularidad en el momento después de TensorFlow. Aporta una sintaxis homogénea y una interfaz sencilla, modular y ampliable para la creación de redes neurales.

Keras se puede usar como una librería de Python que proporciona, de una manera sencilla, la creación de una gran gama de modelos de Deep Learning usando como *backend* otras librerías como TensorFlow o Theano, siendo Tensorflow el elegido para trabajar bajo Keras al ser la librería más popular en Deep Learning.

Por último también hablar de la librería Scikit-learn [66], pues es muy importante en Machine Learning e incluye multitud de funciones, como por ejemplo la matriz de confusión o la validación cruzada, que serán empleadas en este trabajo.

Figura 4-3. Framework empleado para trabajar en Deep Learning

4.2 Base de datos

La base de datos de imágenes que se ha utilizado durante la implementación de este trabajo es la misma que se proporcionó para el IEEE Signal Processing Cup 2018 - Forensic Camera Model Identification, la cual se puede encontrar en [5]. La base de datos cuenta con imágenes capturadas por diez modelos de cámaras diferentes (Tabla 4-2), en concreto un conjunto de 275 imágenes para cada uno de los diez modelos. En total, la base de datos está formada por un total de 2750 imágenes con un peso de 10GB.

Sony NEX-7	Apple iPhone 6
Motorola Moto X	Apple iPhone 4s
Motorola Nexus 6	HTC One M7
Motorola DROID MAXX	Samsung Galaxy S4
LG Nexus 5x	Samsung Galaxy Note 3

Tabla 4-2. Modelos de cámara de la base de datos

Todas las imágenes han sido tomadas y almacenadas en formato JPEG usando los ajustes predeterminados de cada cámara y no han sido postprocesadas de ninguna forma. Han sido además capturadas en situaciones cotidianas (interiores de una casa, calles, carreteras, coches, objetos, etc) sin estar preparadas de ninguna forma.

Se muestra en la siguiente figura alguna de las imágenes de un modelo en concreto, por ejemplo del Motorola-X. En la parte derecha de la figura se muestran las características de las imágenes tomadas por este modelo en cuestión y también de la propia cámara, donde se puede apreciar que son fotos de alta resolución y calidad.

Figura 4-4. Parte de las imágenes del Motorola-X y sus características

Como se ha comentado en varias ocasiones, trabajar con Deep Learning y bases de datos de imágenes grandes como la que se ha presentado, requiere un alto coste computacional y unas muy buenas prestaciones de potencia del equipo a usar. En este trabajo, aun contando con un buen equipo incluso con GPU incluida, se ha visto la necesidad de reducir la base de datos de 10 a 3 modelos de cámaras, pues de la otra forma el entrenamiento de la red neuronal se hacía totalmente inviable al conllevar tiempos demasiado elevados. Por tanto, la base de datos con la que se trabajará estará formada por un total de 825 imágenes con un peso de 3GB: 275 imágenes por cada uno de los 3 modelos de cámaras seleccionados.

4.3 Implementación

Una vez presentado todo el marco teórico, así como el material y la base de datos a utilizar, ya se puede proceder a hablar de la implementación de la red neuronal para conseguir el objetivo del presente trabajo. Se desarrollará todo el proceso seguido y las reflexiones y decisiones realizadas. Este capítulo se puede considerar el más importante del trabajo, porque es aquel en el que se crea, ajusta y entrena el modelo de la red neuronal. Cuanto mejor se haga toda esta implementación, mejor serán los resultados de validación que se mostrarán en el siguiente capítulo.

4.3.1 Punto de partida

Comentar que como punto de partida se tendrán dos carpetas: *train* y *test*. A su vez, dentro de cada una de ellas se tendrán 3 subcarpetas, cuyos nombres serán los 3 modelos de cámaras seleccionados. Inicialmente, dichas subcarpetas de *test* estarán vacías, mientras que las subcarpetas de *train* almacenarán las 275 imágenes de cada una de las cámaras.

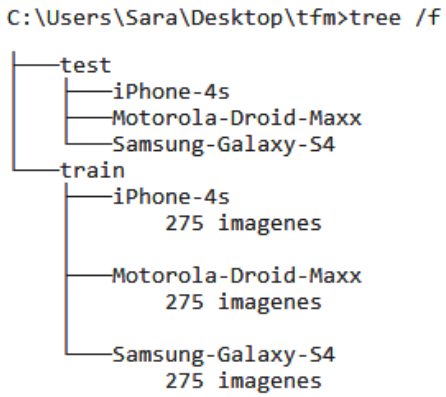


Figura 4-5. Punto de partida de las carpetas train y test

Las librerías que se van a utilizar a lo largo de toda la implementación son las siguientes:

```
import matplotlib.image as mpimg
from PIL import Image
import os, re, shutil, random, keras
from keras.preprocessing.image import ImageDataGenerator
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation, Flatten
from keras.layers import Convolution2D, MaxPooling2D
from keras.preprocessing.image import array_to_img, img_to_array,
load_img
import numpy as np
from sklearn.metrics import confusion matrix
```

Código 4-1. Librerías utilizadas

4.3.2 Separación imágenes de entrenamiento y validación

El primer paso consiste en hacer la separación de los datos en imágenes de entrenamiento y validación. Dado que se dispone de 275 para cada uno de los modelos de cámaras, se ha optado por utilizar 200 imágenes para entrenar y 75 para validar, pues cumple con las separaciones típicas 80-20% / 70-30% ya comentadas.

Para que el sistema sea lo más generalista y real posible, la separación se hace completamente aleatoria. Además, en el siguiente capítulo de experimentos se someterá al sistema a la técnica de "validación cruzada", donde se podrá comprobar si el sistema depende o no de las imágenes aleatorias que tome como entrenamiento y validación.

```
# Separacion de las imagenes de entrenamiento (200) y validacion (75)

for phone in os.listdir("train"):
    path = "train/" + phone
    files = os.listdir(path)
    r = random.sample(range(275), 75)
    files = [files[x] for x in r]
    for file in files:
        source = path + "/" + file
        destination = "test/" + phone + "/" + file
        os.rename(source, destination)
```

Código 4-2. Separación del conjunto en entrenamiento y validación

4.3.3 División y recorte de las imágenes

En un principio se comenzó a trabajar con las imágenes sin ningún tipo de preparación ni preprocesado (en cualquiera de los modelos de las cámaras la resolución de dichas imágenes se encuentra en torno a 3000x4000 px). Esto a la larga supuso un gran problema, pues se estaba trabajando con imágenes muy pesadas con un gran número de píxeles, lo que ralentizaba mucho el trabajo suponiendo un alto coste computacional.

Es por ello que, como se presenta en [39], es bueno dividir las imágenes en parches más pequeños con los que trabajar. En este trabajo se ha optado por recortar las imágenes en trozos de 64x64 píxeles, pues se ha visto que son unas dimensiones muy utilizadas en este tipo de aplicaciones. Gracias a esto, y a pesar de que ahora el número de imágenes crece considerablemente, el coste computacional que esto conlleva es mucho menor.

```
# Recorte de las imagenes en parches de 64x64

for phone in os.listdir("train"):

    path = "train/" + phone
    files = os.listdir(path)
    for file in files:
        crop_picture(path + "/" + file)
        os.remove(path + "/" + file)

    path = "test/" + phone
    files = os.listdir(path)
    for file in files:
        crop_picture(path + "/" + file)
        os.remove(path + "/" + file)
```

Código 4-3. Recorte de las imágenes en trozos de 64 x 64 píxeles

4.3.4 Data Augmentation

Como se comentó en el capítulo anterior, una buena técnica para mejorar y ampliar la base de datos de forma que sea lo más generalista posible consiste en aplicar *Data Augmentation.* Gracias a la opción escogida de los generadores (se detallará en el siguiente paso) para trabajar con la red neuronal es viable plantearse un aumento de la base de datos mediante esta técnica.

Existen numerosas modificaciones [67] que pueden aplicarse a las imágenes con el fin de crear otras nuevas, pero hay que tener algo muy importante en cuenta debido al problema que se está tratando. Como ya sabemos, el objetivo de este trabajo consiste en detectar el modelo de cámara que captó una cierta imagen dada, por tanto es evidente pensar que las imágenes con las que se entrene la red neuronal no pueden perder la huella de la cámara con la que fue captada. Con el planteamiento de esta hipótesis se decide por tanto que las modificaciones que se van a aplicar serán rotaciones, giros, volteos, etc., sin cambiar las características o propiedades de la imagen que sean propias de la cámara con la que se captó y que ayuden a la identificación.

```
# Data Augmentation sobre las imágenes de entrenamiento

for phone in os.listdir("train"):
    path = "train/" + phone
    files = os.listdir(path)
    for file in files:
        input_path = path + "/" + file
        output_path = path + "/" + "{}" + file
        count = 0

        gen = ImageDataGenerator(
            rotation_range=20,
            width_shift_range=0.2,
            height_shift_range=0.2,
            horizontal_flip=True
        )

        image = img_to_array(load_img(input_path))
        image = image.reshape((1,) + image.shape)

        images_flow = gen.flow(image, batch_size=1)
        for i, new_images in enumerate(images_flow):
            new_image = array_to_img(new_images[0], scale=True)
            new_image.save(output_path.format(i + 1))
            if i >= count:
                break
```

Código 4-4. Data Augmentation sobre las imágenes del conjunto de entrenamiento

4.3.5 Generators

Se ha visto como el dividir y recortar las imágenes pesadas en trozos de 64x64 es una ayuda considerable a la hora de entrenar una red neuronal, pues ahora las imágenes con las que se trabajarán son mucho más ligeras. Sin embargo, sigue habiendo un problema también muy común en el mundo del Deep Learning que aún no se ha solucionado: el conjunto total de las imágenes es demasiado grande como para almacenarlo en memoria.

Se decidió por tanto que la forma de trabajo típica de almacenar los datos en variables (y por tanto en memoria) era algo inviable para este proyecto, pues habría que cargar y almacenar en una variable un total de 3GB de imágenes para que posteriormente la red neuronal pudiera trabajar y entrenar con ella.

Antes de comenzar este trabajo ya se tenía un cierto manejo de Keras en este ámbito del Deep Learning, pero siempre trabajando con bases de datos pequeñas y muy manejables, por lo que a priori no se conocía ninguna posible solución. Se investigó por tanto en la documentación de Keras [68], descubriendo que hay dos formas posibles de abarcar el entrenamiento de una red neuronal:

- En primer lugar estaría la forma ya conocida de almacenar todo el conjunto de datos en una variable, descartada para este trabajo por el problema ya comentado. Esto es recomendable cuando las bases de datos son pequeñas y manejables, no habiendo problemas de almacenamiento en memoria.

  Otra desventaja de este método es que no permite realizar *Data Augmentation* de forma automática sobre la marcha, sino que habría que realizar ese aumento de los datos a mano, imagen por imagen, almacenando también todas esas 'nuevas imágenes' en memoria, por lo que el problema crecería aún más.

- La otra opción, que ha sido la seleccionada para este trabajo, consiste en trabajar con los llamados *generators* de Python. Un generador es una función que se comporta como un iterador, recorriendo los elementos de un objeto, como por ejemplo los elementos en una lista. Las ventajas que ofrece esta herramienta son las siguientes:

  - ✓ No almacena todo el conjunto de las imágenes en memoria, sino que lo va haciendo por 'lotes' recorriendo todas las imágenes. De esta forma, en memoria solo estará almacenado el lote actual de las imágenes que será pasado en ese momento a la red neuronal durante el entrenamiento.

  - ✓ Se ejecuta en paralelo con el modelo, proporcionando una mayor eficiencia. Por ejemplo, esto permite hacer un aumento de datos en tiempo real en imágenes en la CPU en paralelo con el entrenamiento del modelo en la GPU, haciendo que cada nuevo lote de datos se escoja aleatoriamente teniendo también en cuenta las 'nuevas imágenes' generadas con el aumento de datos.

```
# Creación de los generadores de entrenamiento y validacion

batch_size = 1024

train_datagen = ImageDataGenerator(rescale=1./255)
test_datagen = ImageDataGenerator(rescale=1./255)

train_generator = train_datagen.flow_from_directory(
        'train',
        target_size=(64, 64),
        batch_size=batch_size,
        class_mode='categorical',
        shuffle=False)

validation_generator = test_datagen.flow_from_directory(
        'test',
        target_size=(64, 64),
        batch_size=batch_size,
        class_mode='categorical',
        shuffle=False)
```

Código 4-5. Creación de los generadores para entrenamiento y validacion

Recordemos que con el hiperparámetro *batch_size* se indica el número de datos que se usarán en cada actualización de los parámetros del modelo.

4.3.6 Modelo de capas

Llegados a este punto, ya está todo preparado para comenzar a crear la red neuronal convolucional siguiendo los conocimientos explicados en el capítulo 3.

Keras ofrece varios modelos preentrenados [69] como son ResNet, MobileNet, DenseNet, NASNet, etc., empleando la base de datos ImageNet [70]. El hecho que sean preentrenados es muy valioso en casos donde no se dispone de suficientes datos para entrenarlas. Sin embargo, al tratarse de modelos de capas muy profundos se decidió, basándose en esos modelos, crear un modelo propio y específico para el problema aquí tratado.

En Keras a partir del modelo `Sequential` se pueden definir las capas de manera sencilla con el método `add()`, como se presenta a continuación:

```
# Modelo de capas

model = Sequential()

model.add(Convolution2D(filters=32, kernel_size=(4,4), input_shape
=(64,64,3), strides=(1,1)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2,2), strides=2, padding='same'))
model.add(Dropout(0.25))

model.add(Convolution2D(filters=48, kernel_size=(5,5), strides=(1,1)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2,2), strides=2, padding='same'))

model.add(Convolution2D(filters=64, kernel_size=(5,5), strides=(1,1)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2,2), strides=2, padding='same'))
model.add(Dropout(0.25))

model.add(Convolution2D(filters=128, kernel_size=(5,5), strides=(1,1)))
model.add(Flatten())
model.add(Activation('relu'))
model.add(Dropout(0.5))
model.add(Dense(3, activation='softmax'))
```

Código 4-6. Modelo de capas empleado para la red neuronal convolucional

A grandes rasgos se puede apreciar que la construcción de las capas sigue lo ya explicado en la teoría de las redes neuronales: capa convolucional seguida de una función de activación, continuando con una capa *pooling* y así sucesivamente, terminando con una capa densamente conectada con la función de activación *softmax*.

Centrándonos más en los detalles, se observa que aparecen parámetros también ya explicados, como son el número de filtros, el tamaño del kernel y del *pool*, el paso de avance de las ventanas (*stride*) y el *padding*. De este último parámetro decir que asignarle el valor "same" indica que se añadirán tantas filas y columnas de ceros como sea necesario para que la salida tenga la misma dimensión que la entrada con el fin de mejorar el barrido que se realiza con la ventana que se va deslizando.

Por último, aún falta algo por comentar: la capa *Dropout*. Es una técnica de regularización para redes neuronales propuesta por [71] en la que ciertas neuronas seleccionadas de forma aleatoria son ignoradas durante el entrenamiento. Se ha considerado que sería bueno implementar estas capas, pues el efecto que se consigue es que la red se vuelve menos sensible a los pesos específicos de las neuronas, dando como resultado una red cuya capacidad de generalización es mayor (evita el sobreajuste).

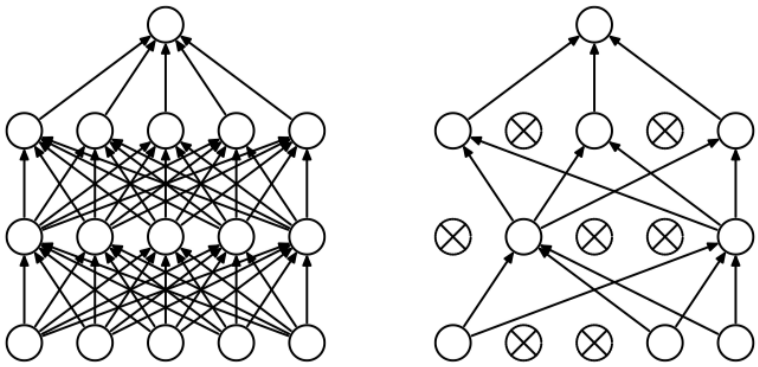


Figura 4-6. Funcionamiento del dropout

4.3.7 Configuración del proceso de aprendizaje

Una vez se tiene el modelo definido se puede configurar cómo será su proceso de aprendizaje con el método `compile()`, con el que se pueden especificar algunas propiedades a través de argumentos del método.

```
# Configuración del proceso de aprendizaje

model.compile(loss='categorical_crossentropy',
              optimizer='sgd', #stocastic gradient descent
              metrics=['accuracy'])
```

Código 4-7. Configuración del proceso de aprendizaje

El primero de estos argumentos es la función de pérdidas que se usará para evaluar el grado de error entre las salidas calculadas y las salidas deseadas de los datos de entrenamiento. En la página del manual de Keras [68] se pueden encontrar todos los tipos de funciones de pérdidas disponibles. En el problema que estamos tratando, se usará la función `categorical_crossentropy`, pues la salida debe ser en formato categórico, es decir, que la variable de salida debe tomar un valor entre 3 posibles (recordemos que queremos clasificar una imagen dada entre 3 modelos de cámaras posibles). La elección de la mejor función de pérdidas reside en entender qué tipo de error es o no es aceptable para el problema en concreto.

El optimizador es otro de los argumentos que se requieren en el método de `compile()`. Keras dispone en estos momentos de diferentes optimizadores que pueden usarse: SGD, RMSprop, Adagrad, Adadelta, Adam, Adamax, Nadam. Se puede encontrar más detalle de cada uno de ellos en la documentación de Keras [68]. De forma general, se puede ver el proceso de aprendizaje como un problema de optimización global donde los parámetros (los pesos y los sesgos) se deben ajustar de tal manera que la función de pérdida presentada anteriormente se minimice. En la mayoría de los casos, estos parámetros no se pueden resolver analíticamente, pero en general se pueden aproximar bien con algoritmos de optimización iterativos u optimizadores, como los mencionados anteriormente.

Finalmente hay que indicar la métrica que se usará para monitorizar el proceso de aprendizaje (y prueba) de la red neuronal. Lo más típico y común es tener en cuenta la *accuracy* (fracción de imágenes que son correctamente clasificadas).

4.3.8 Entrenamiento y validación del modelo

Llegado a este punto ya estaría todo listo para la parte final de la implementación: el entrenamiento y la validación. Un buen preprocesado de las imágenes de la base de datos, una buena construcción del modelo de capas y una buena configuración de proceso del aprendizaje es algo crucial para que todo vaya bien en el entrenamiento y en la validación.

Para ello podemos entrenar o "ajustar" el modelo a los datos de entrenamiento invocando al método `fit_genertor()`, pues estamos trabajando con generadores:

```
# Entrenamiento del modelo

model.fit_generator(
        train_generator,
        epochs=100,
        validation_data=validation_generator)
```

Código 4-8. Entrenamiento del modelo

Como se puede apreciar, dos de los argumentos que recibe este método son los *generadores* de entrenamiento y validación ya creados y explicados anteriormente, donde en ellos va implícitamente el valor del hiperparámetro *batch_size.* Para finalizar, lo último que quedaría para terminar de configurar el entrenamiento sería indicar el hiperparámetro *epochs*, que recordemos que indica el número de veces en las que todos los datos de entrenamiento pasarán por la red neuronal en el proceso de entrenamiento.

Destacar que en ningún momento hay que indicarle al programa que utilice la GPU del ordenador, pues si están instaladas previamente las librerías CUDA y cuDNN, como bien se indicó en la Figura 4-1 automáticamente la CPU se liberará de aquellas funciones más pesadas que requieran un alto coste encargándose de ellas la GPU.

En la siguiente figura puede verse el rendimiento del ordenador durante la fase de entrenamiento, mostrando en más detalle el uso de la GPU:

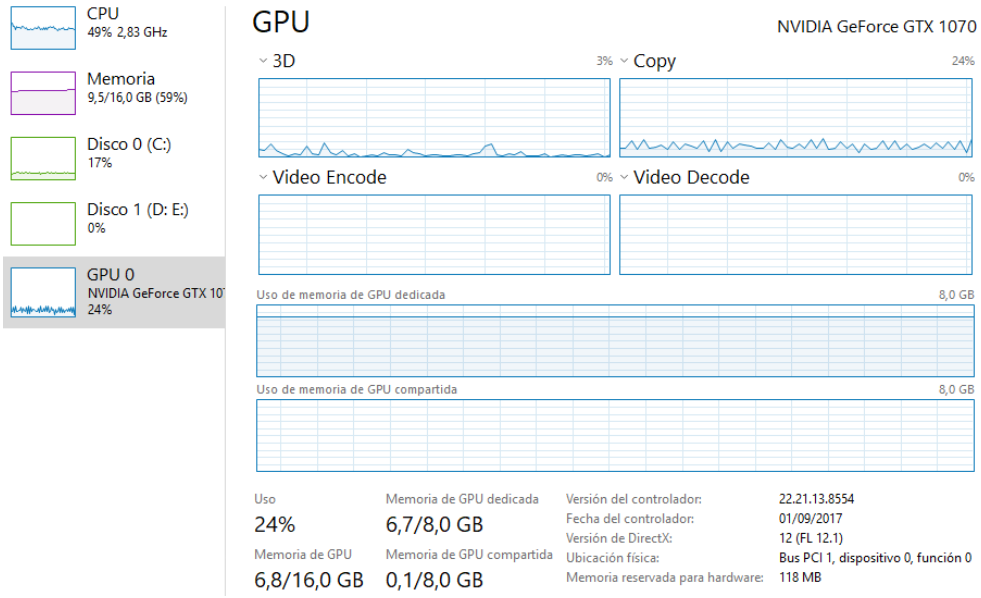


Figura 4-7. Rendimiento del ordenador durante la fase de entrenamiento

5 EXPERIMENTOS REALIZADOS Y RESULTADOS

Como se ha comentado anteriormente, la parte de implementación es crucial para obtener unos buenos resultados de validación. Es en este capítulo donde se evaluará el rendimiento que ofrece el modelo que se ha implementado.

Ya se ha visto que por motivos de rendimiento y tiempo, no se trabajará con los 10 modelos de cámaras que dispone la base de datos del concurso de IEEE Signal Processing Cup sino con 3 de esos modelos. A continuación se muestran los experimentos que se han realizado para evaluar la precisión del modelo implementado.

5.1 Experimento 1: modelos iPhone 4S, Motorola Droid Maxx y Samsung Galaxy S4

En primer lugar se seleccionan los 3 modelos de cámaras con los que se va a trabajar. Como indica el título del apartado, las cámaras seleccionadas son: **iPhone 4S, Motorola Droid Maxx y Samsung Galaxy S4.** El motivo principal de esta selección ha sido escoger cámaras cuya resolución sea parecida, de esta forma al recortar las imágenes en parches de 64x64 px todas las cámaras obtendrán una cantidad aproximadamente igual de imágenes con las que entrenar y validar.

Un resumen de los datos que se van a utilizar para entrenar y validar el modelo en este primer experimento se muestra en la siguiente tabla:

Modelo	Resolución	Imágenes	Entrenamiento	Validación	Entrenamiento (recortes 64x64)	Validación (recortes 64x64)
iPhone	3264 x 2448	275	200	75	99.960	49.980
Motorola	4320 x 2432	275	200	75	153.360	71.170
Samsung	4128 x 2322	275	200	75	119.040	59.520
					372.360	180.670

Tabla 5-1. Datos utilizados para el experimento 1

5.1.1 Entrenamiento y validación

Tras haber llevado a cabo todos los pasos previos al entrenamiento vistos en el capítulo anterior, el último paso consiste en entrenar el modelo y validarlo. Para ello se ejecuta el método `fit_generator()` como ya se ha visto.

El resultado se muestra a continuación. Por comodidad solamente se muestran las 5 primeras y últimas epochs:

```
Epoch 1/100
364/364 [==============================] - 528s - loss: 1.1601 - acc: 0.3954 - val_loss: 1.0678 - val_acc: 0.4151
Epoch 2/100
364/364 [==============================] - 426s - loss: 1.1129 - acc: 0.4174 - val_loss: 1.0392 - val_acc: 0.4100
Epoch 3/100
364/364 [==============================] - 424s - loss: 1.0395 - acc: 0.4655 - val_loss: 1.0050 - val_acc: 0.5151
Epoch 4/100
364/364 [==============================] - 424s - loss: 1.0529 - acc: 0.5002 - val_loss: 1.0125 - val_acc: 0.5003
Epoch 5/100
364/364 [==============================] - 423s - loss: 1.0178 - acc: 0.5201 - val_loss: 1.0403 - val_acc: 0.4421
...

Epoch 96/100
364/364 [==============================] - 413s - loss: 0.5622 - acc: 0.7841 - val_loss: 0.5609 - val_acc: 0.7928
Epoch 97/100
364/364 [==============================] - 415s - loss: 0.5404 - acc: 0.7890 - val_loss: 0.6235 - val_acc: 0.7912
Epoch 98/100
364/364 [==============================] - 418s - loss: 0.5374 - acc: 0.7885 - val_loss: 0.5355 - val_acc: 0.7978
Epoch 99/100
364/364 [==============================] - 411s - loss: 0.5489 - acc: 0.7882 - val_loss: 0.5058 - val_acc: 0.8063
Epoch 100/100
364/364 [==============================] - 410s - loss: 0.5470 - acc: 0.7815 - val_loss: 0.5449 - val_acc: 0.7872
```

Figura 5-1. Resultados de la primera tanda de 100 epochs del experimento 1

Se puede observar que tras 100 epochs, la precisión alcanza un valor máximo de **val_acc = 80.63%**. Nos fijamos en este valor porque es el que verdaderamente importa, la precisión que se obtiene al pasar por el modelo aquellas fotos de validación que separamos al principio y que no han sido vistas nunca por el modelo a la hora del entrenamiento. También se puede apreciar como las pérdidas val_loss van disminuyendo a medida que avanzan las iteraciones.

Otro detalle importante a destacar es el tiempo que ha tardado el modelo en realizar el entrenamiento y la validación, que se puede calcular viendo el tiempo que ha empleado en cada epoch, dando un total de 10 horas y media aproximadamente. Además de esto también habría que sumar el tiempo que conlleva los primeros pasos de la implementación: recorte de las fotos y data augmentation, pues al tratar con una gran cantidad de fotos pesadas estos pasos también llevan su tiempo. Por tanto, el tiempo completo se podría decir que ha sido aproximadamente de 11 horas.

A continuación se decide volver a iterar unas 100 epochs más, para ver si sigue aumentando dicho porcentaje de precisión o si por el contrario ya ha alcanzado su máxima valor. El resultado es el siguiente:

```
Epoch 1/100
364/364 [==============================] - 419s - loss: 0.5332 - acc: 0.7862 - val_loss: 0.6480 - val_acc: 0.8005
Epoch 2/100
364/364 [==============================] - 414s - loss: 0.5399 - acc: 0.7847 - val_loss: 0.5203 - val_acc: 0.7975
Epoch 3/100
364/364 [==============================] - 414s - loss: 0.5335 - acc: 0.7924 - val_loss: 0.5132 - val_acc: 0.8019
Epoch 4/100
364/364 [==============================] - 414s - loss: 0.5477 - acc: 0.7859 - val_loss: 0.5199 - val_acc: 0.7954
Epoch 5/100
364/364 [==============================] - 411s - loss: 0.5292 - acc: 0.7901 - val_loss: 0.4997 - val_acc: 0.8091
...

Epoch 96/100
364/364 [==============================] - 413s - loss: 0.4320 - acc: 0.8340 - val_loss: 0.4349 - val_acc: 0.8379
Epoch 97/100
364/364 [==============================] - 413s - loss: 0.4230 - acc: 0.8363 - val_loss: 0.5151 - val_acc: 0.8459
Epoch 98/100
364/364 [==============================] - 415s - loss: 0.4139 - acc: 0.8396 - val_loss: 0.4903 - val_acc: 0.8273
Epoch 99/100
364/364 [==============================] - 413s - loss: 0.4368 - acc: 0.8314 - val_loss: 0.3949 - val_acc: 0.8517
Epoch 100/100
364/364 [==============================] - 474s - loss: 0.4351 - acc: 0.8313 - val_loss: 0.4419 - val_acc: 0.8463
```

Figura 5-2. Resultados de la segunda tanda de 100 epochs del experimento 1

Se observa que el valor máximo alcanzado es de **val_acc = 85.17%**, consiguiendo con esta segunda tanda de 100 epochs un incremento solamente de un 5% en la precisión. Se probó a iterar unas 50 epochs más y se comprobó que dicho valor no aumentaba más, por lo que finalmente la precisión del modelo es de un 85.17%.

5.1.2 Matriz de confusión

De momento para evaluar el modelo solo nos hemos centrado en su precisión, es decir la proporción entre las predicciones correctas que ha hecho el modelo y el total de predicciones. Sin embargo, aunque en ocasiones resulta suficiente, otras veces es necesario profundizar un poco más y tener en cuenta los tipos de predicciones correctas e incorrectas que realiza el modelo en cada una de sus categorías.

En el mundo de Machine Learning una herramienta para evaluar modelos es la matriz de confusión (confusion matrix) [72], una tabla con filas y columnas que contabilizan las predicciones en comparación con los valores reales. Esta tabla se usa para entender mejor cómo el modelo se comporta de bien y es muy útil para mostrar de forma explícita cuando una clase es confundida con otra. Una matriz de confusión para un ejemplo de clasificador binario tiene la siguiente estructura:

		Predicción	
		Positivos	**Negativos**
Observación	**Positivos**	Verdaderos Positivos (VP)	Falsos Negativos (FN)
	Negativos	Falsos Positivos (FP)	Verdaderos Negativos (VN)

Tabla 5-2. Matriz de confusión para clasificador binario

En la que:

- VP es la cantidad de positivos que fueron clasificados correctamente como positivos.
- VN es la cantidad de negativos que fueron clasificados correctamente como negativos.
- FN es la cantidad de positivos que fueron clasificados incorrectamente como negativos.
- FP es la cantidad de negativos que fueron clasificados incorrectamente como positivos.

Con esta matriz de confusión, la precisión se puede calcular sumando los valores de la diagonal y dividiendo por el total (ver ecuación 5.1), por lo que podemos llegar al mismo valor de precisión pero dando además más información adicional.

Ahora bien, la precisión puede ser engañosa en la calidad del modelo pues al medirla para un modelo concreto no se distingue entre los errores de tipo falso positivo y falso negativo, como si ambos tuvieran la misma importancia. Es por ello que otra métrica importante que se suele sacar de esta matriz es la *sensibilidad* (ver ecuación 5.2), que indica cómo de bien el modelo evita los falsos negativos, es decir, del total de observaciones positivas cuantas detecta el modelo.

$$Precisión = \frac{VP + VN}{VP + FP + FN + VN} \qquad (5.1)$$

$$Sensibilidad = \frac{VP}{VP + FN} \qquad (5.2)$$

Dicha matriz de confusión se ha calculado mediante el método `confusion_matrix()` [73] que ofrece la librería `sklearn.metrics`. Este método únicamente necesita que se le pase como argumentos las etiquetas reales y predichas de cada una de las imágenes. Tras ejecutarlo, se procede a pintar la matriz con diferentes tonalidades como muestra la siguiente imagen para que quede mejor visualmente:

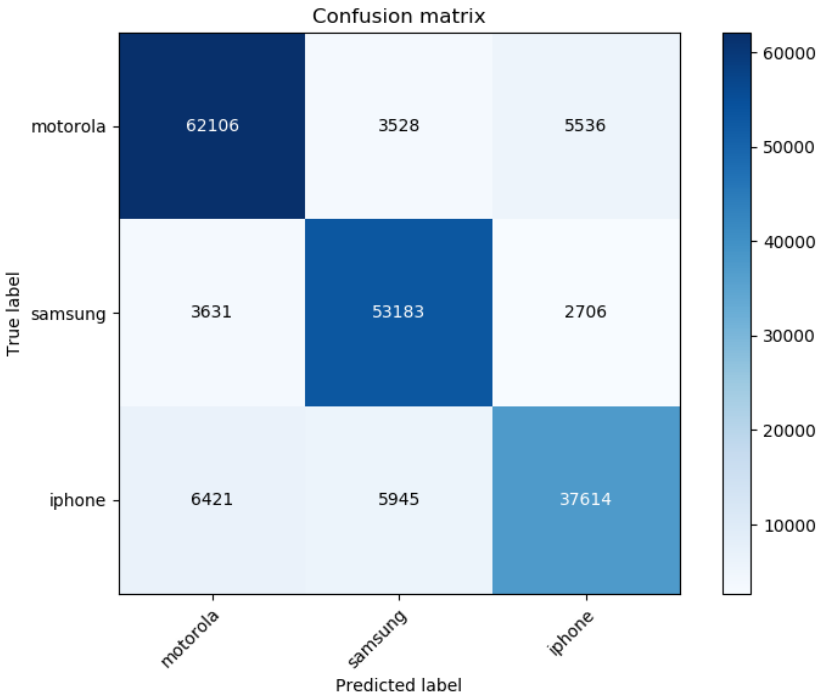


Figura 5-3. Matriz de confusión en cantidad de imágenes del experimento 1

Representada en términos de tanto por uno quedaría de la siguiente manera:

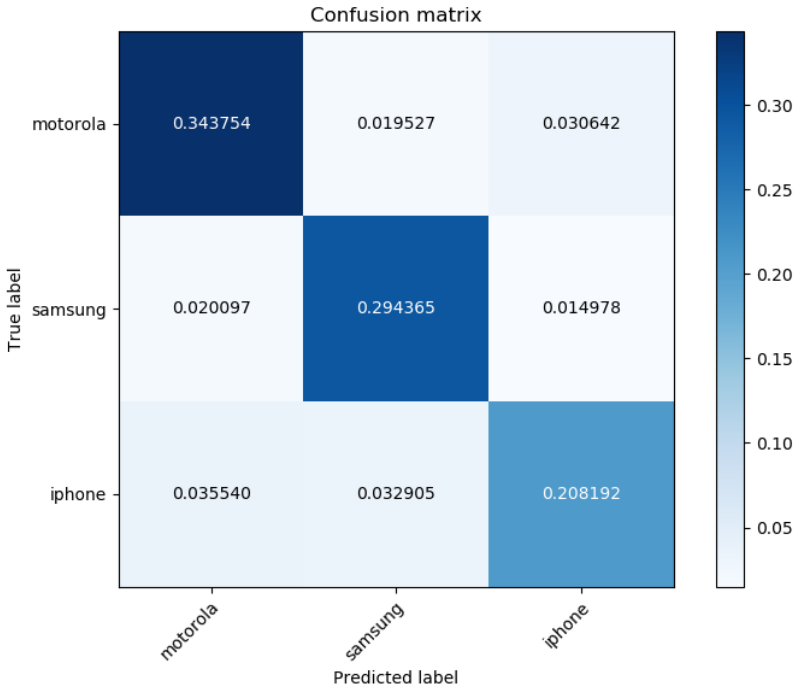


Figura 5-4. Matriz de confusión en tanto por uno del experimento 1

Se aprecia que la clase que mejor predice es *motorola*, seguida de *samsung* y finalmente *iphone*. Es por ello que se decide en un segundo experimento sustituir la clase que obtiene peores resultados (*iphone*) por alguno de los otros modelos de cámaras, para observar qué sucede en ese nuevo caso y así probar con otro grupo de móviles.

5.2 Experimento 2: cambiando iPhone 4S por LG Nexux 5X

Se decide en este segundo experimento cambiar el modelo de cámara Iphone 4S por el **LG Nexux 5X.** Se decide este modelo por los mismos motivos que en el experimento anterior, para que las relaciones entre las cámaras sean parecidas de forma que ambas tengan finalmente el mismo número aproximado de imágenes recortadas.

Un resumen de los datos que se han utilizado para entrenar y validar el modelo en este experimento se muestra en la siguiente tabla:

Modelo	Resolución	Imágenes	Entrenamiento	Validación	Entrenamiento (recortes 64x64)	Validación (recortes 64x64)
LG	4032 x 3024	275	200	75	152.374	75.435
Motorola	4320 x 2432	275	200	75	148.952	71.170
Samsung	4128 x 2322	275	200	75	119.040	59.520
					420.366	206.125

Tabla 5-3. Datos utilizados para el experimento 2

5.2.1 Entrenamiento y validación

Al igual que en el experimento anterior, se muestran a continuación las 5 primeras y últimas epochs de una primera tanda de 100 epochs:

```
Epoch 1/100
411/411 [==============================] - 520s - loss: 1.1618 - acc: 0.3740 - val_loss: 1.0830 - val_acc: 0.4143
Epoch 2/100
411/411 [==============================] - 402s - loss: 1.1070 - acc: 0.3934 - val_loss: 1.0554 - val_acc: 0.4214
Epoch 3/100
411/411 [==============================] - 442s - loss: 1.0666 - acc: 0.4281 - val_loss: 1.0561 - val_acc: 0.4331
Epoch 4/100
411/411 [==============================] - 459s - loss: 1.0772 - acc: 0.4715 - val_loss: 1.0159 - val_acc: 0.4724
Epoch 5/100
411/411 [==============================] - 428s - loss: 1.0433 - acc: 0.4928 - val_loss: 0.9946 - val_acc: 0.4896
...

Epoch 96/100
411/411 [==============================] - 496s - loss: 0.6228 - acc: 0.7284 - val_loss: 0.6589 - val_acc: 0.6883
Epoch 97/100
411/411 [==============================] - 495s - loss: 0.6246 - acc: 0.7333 - val_loss: 0.6340 - val_acc: 0.7151
Epoch 98/100
411/411 [==============================] - 496s - loss: 0.6166 - acc: 0.7354 - val_loss: 0.6824 - val_acc: 0.6953
Epoch 99/100
411/411 [==============================] - 497s - loss: 0.6267 - acc: 0.7360 - val_loss: 0.6777 - val_acc: 0.7004
Epoch 100/100
411/411 [==============================] - 497s - loss: 0.6102 - acc: 0.7462 - val_loss: 0.6661 - val_acc: 0.7157
```

Figura 5-5. Resultados de la primera tanda de 100 epochs del experimento 2

Se observa tras estas primeras 100 epochs que el valor máximo alcanzado de presión es **val_acc = 71.57%,** algo menor que en el experimento anterior, por lo que a priori parece que con este cambio en los modelos de las cámaras se va a alcanzar una menor precisión.

Al igual que antes se procedió a realizar otras 100 epochs para ver si la precisión seguía aumentando o si por el contrario se mantenía estable sin seguir creciendo:

```
Epoch 1/100
411/411 [==============================] - 405s - loss: 0.6104 - acc: 0.7473 - val_loss: 0.6937 - val_acc: 0.6985
Epoch 2/100
411/411 [==============================] - 401s - loss: 0.6205 - acc: 0.7362 - val_loss: 0.6238 - val_acc: 0.7148
Epoch 3/100
411/411 [==============================] - 401s - loss: 0.6103 - acc: 0.7388 - val_loss: 0.8030 - val_acc: 0.7060
Epoch 4/100
411/411 [==============================] - 400s - loss: 0.6170 - acc: 0.7414 - val_loss: 0.6131 - val_acc: 0.7169
Epoch 5/100
411/411 [==============================] - 400s - loss: 0.6207 - acc: 0.7321 - val_loss: 0.6251 - val_acc: 0.7188

Epoch 96/100
411/411 [==============================] - 498s - loss: 0.5195 - acc: 0.7843 - val_loss: 0.5952 - val_acc: 0.7477
Epoch 97/100
411/411 [==============================] - 408s - loss: 0.5151 - acc: 0.7925 - val_loss: 0.6093 - val_acc: 0.7503
Epoch 98/100
411/411 [==============================] - 418s - loss: 0.5159 - acc: 0.7869 - val_loss: 0.5327 - val_acc: 0.7697
Epoch 99/100
411/411 [==============================] - 423s - loss: 0.5167 - acc: 0.7851 - val_loss: 0.5598 - val_acc: 0.7615
Epoch 100/100
411/411 [==============================] - 423s - loss: 0.5050 - acc: 0.7966 - val_loss: 0.5876 - val_acc: 0.7533
```

Figura 5-6. Resultados de la segunda tanda de 100 epochs del experimento 2

El valor máximo de precisión ahora alcanzado es de un **val_acc = 76.97%,** volviendo a crecer un 5% en esta segunda tanda de 100 epochs al igual que ocurría en el experimento anterior. Igualmente se probó a iterar unas 50 epochs más y se volvió a comprobar que dicho valor no aumentaba prácticamente más, por lo que finalmente la precisión del modelo con esta nueva elección de cámaras es de un 76.97%, casi un 10% menor que el primer experimento cuando se empleaba el *iphone* en lugar del *lg*.

5.2.2 Matriz de confusión

Se muestra a continuación la matriz de confusión para este segundo experimento, tanto de forma normal expresando los resultados en número de imágenes como en tanto por uno:

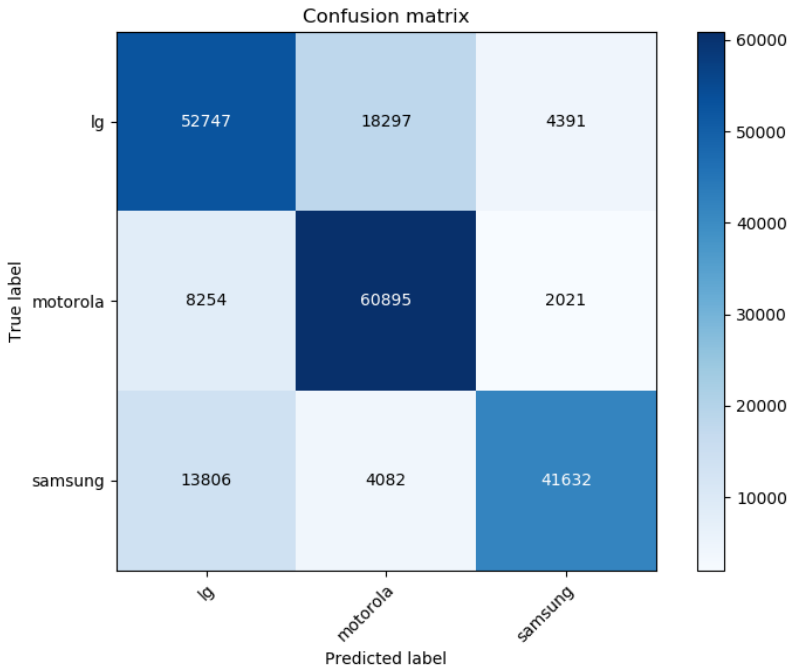


Figura 5-7. Matriz de confusión en cantidad de imágenes del experimento 2

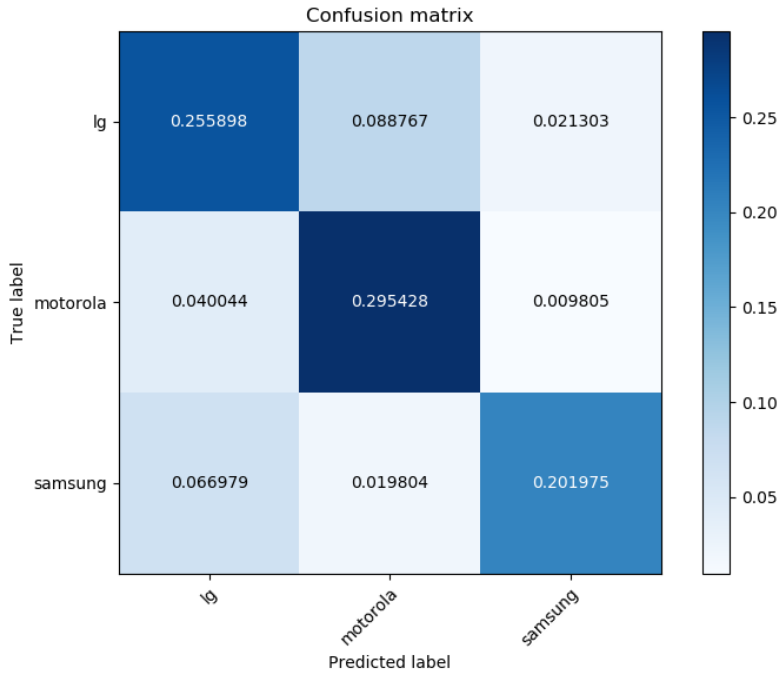


Figura 5-8. Matriz de confusión en tanto por uno del experimento 2

Es este caso se observa que los resultados son algo menos desequilibrados que en el primer experimento, siendo ahora la clase con mayor presión *motorola*, seguida de *lg*, seguida de *samsung*.

5.3 Experimento 3: validación cruzada

En el capítulo anterior de la implementación del modelo, se ha visto como el primer paso consistía en hacer la separación de las imágenes que se utilizarían para entrenar y para validar, dividiendo el conjunto de 275 imágenes de cada modelo en 200 y 75 imágenes respectivamente. Aunque dicha separación se haga totalmente de forma aleatoria, puede dar la casualidad de que esos dos conjuntos salgan 'buenos' o 'malos', obteniendo una precisión alta o baja dependiente de esa separación.

La validación cruzada (*cross-validation*) [74] es un método muy utilizado en Machine Learning para garantizar que los resultados que obtengamos sean independientes de la partición entre datos de entrenamiento y datos de validación, y por eso se usa mucho en para validar modelos generados en proyectos de Inteligencia Artificial. Consiste en calcular la precisión del modelo sobre diferentes particiones del conjunto de datos, como se muestra en la siguiente imagen:

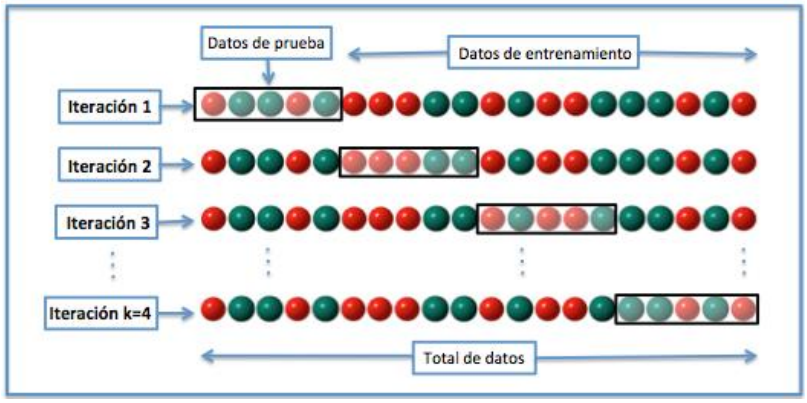


Figura 5-9. Validación cruzada

Validando el modelo de esta forma empleando cada vez un subconjunto distinto de entrenamiento y test, permite observar si las precisiones obtenidas difieren mucho entre ellas indicando que el modelo no es generalista (sobreajuste) o si por el contrario las precisiones salen muy parecidas entre ellas indicando que el modelo generaliza muy bien no dependiendo de dicha separación.

Cuando se emplea validación cruzada, lo ideal es iterar entre 5 y 10 veces. En este trabajo, y en general cuando se trabaja con redes neuronales convolucionales, los altos tiempos que requieren el entrenamiento y la validación al trabajar con imágenes pueden hacer que esto se convierta en un proceso muy pesado. Es por ello que se ha decidido solamente iterar 3 veces, fraccionando el conjunto de datos en 3 particiones y procediendo a hacer validaciones cruzadas como se mostraba en la imagen anterior.

Se van a emplear los modelos de cámaras que se seleccionaron en el experimento 1: iPhone 4S, Motorola Droid Maxx y Samsung Galaxy S4, donde se obtuvo en esa primera iteración una precisión máxima del 85.17%. A continuación se van a realizar dos experimentos más en los cuales se emplean distintas separaciones del conjunto de entrenamiento y test. Los resultados de la validación cruzada en estos dos experimentos son los siguientes:

```
Epoch 96/100
281/281 [==============================] - 371s - loss: 0.4276 - acc: 0.8298 - val_loss: 0.5409 - val_acc: 0.7939
Epoch 97/100
281/281 [==============================] - 371s - loss: 0.4355 - acc: 0.8238 - val_loss: 0.5110 - val_acc: 0.8159
Epoch 98/100
281/281 [==============================] - 377s - loss: 0.4318 - acc: 0.8266 - val_loss: 0.4703 - val_acc: 0.8311
Epoch 99/100
281/281 [==============================] - 370s - loss: 0.4246 - acc: 0.8313 - val_loss: 0.6299 - val_acc: 0.8130
Epoch 100/100
281/281 [==============================] - 369s - loss: 0.4312 - acc: 0.8281 - val_loss: 0.5185 - val_acc: 0.8142
```

Figura 5-10. Validación cruzada, resultados de la segunda iteración

```
Epoch 96/100
313/313 [==============================] - 411s - loss: 0.4320 - acc: 0.8340 - val_loss: 0.5109 - val_acc: 0.8139
Epoch 97/100
313/313 [==============================] - 411s - loss: 0.4253 - acc: 0.8388 - val_loss: 0.5410 - val_acc: 0.8155
Epoch 98/100
313/313 [==============================] - 417s - loss: 0.4139 - acc: 0.8396 - val_loss: 0.4433 - val_acc: 0.8212
Epoch 99/100
313/313 [==============================] - 410s - loss: 0.4213 - acc: 0.8293 - val_loss: 0.5199 - val_acc: 0.8379
Epoch 100/100
313/313 [==============================] - 470s - loss: 0.4114 - acc: 0.8361 - val_loss: 0.4349 - val_acc: 0.8459
```

Figura 5-11. Validación cruzada, resultados de la segunda iteración

Se comprueba por tanto que las precisiones son muy parecidas, variando muy poco entre ellas, por lo que se concluye que el modelo de red neuronal implementado es generalista al no depender de las separaciones entre entrenamiento y test, por lo que no está sobreajustado.

Debido al alto tiempo de ejecución que requiere este experimento de validación cruzada, pues consiste en entrenar y validar el modelo una serie de veces, no se va a realizar validación cruzada para el caso del experimento 2 donde se cambiaba el modelo del iPhone por el del LG, ya que es muy probable que al igual que en este caso se obtengan predicciones muy parecidas a la que se obtuvo en el experimento 2, pues se ha comprobado que el modelo de red neuronal implementado no está sobreajustado.

5.4 Experimento 4: predicciones finales

Para finalizar este capítulo de experimentos y resultados, se va a proceder a simular predicciones reales que por ejemplo se podrían realizar en análisis forense de imágenes.

Recordemos que en los experimentos anteriores, las métricas de precisión y matrices de confusión obtenidas son calculadas mediante el método `fit_genertor()` al modelo implementado. Esto quiere decir que todo el tiempo se está trabajando con los recortes de las imágenes en sí, y no con las imágenes originales. Si pensamos en un problema real, se debe poder introducir al sistema una imagen de test real y completa, es decir la original, y que el modelo prediga de qué modelo de cámara se trata. De esta forma, el recorte de dicha imagen y el paso de cada trozo a la red neuronal sería transparente hacia el usuario, únicamente viendo a qué modelo pertenece la imagen original en cuestión.

Para ello se crea una función como la siguiente:

```
def predict(path,file,classes,ctrl):

    if os.path.exists(path + "cuts"):
        shutil.rmtree(path + "cuts")
    os.mkdir(path + "cuts")

    crop_picture2(path, file)

    images=[]
    for root, dirnames, filenames in os.walk(path + "cuts"):
        for filename in filenames:
            if re.search("\.(jpg|jpeg|png|bmp|tiff)$", filename):
                filepath = os.path.join(root, filename)
                image = plt.imread(filepath)
                images.append(image)

    img = np.array(images, dtype=np.uint8) #convierto de lista a numpy
    img = img.astype('float32')
    img = img / 255.

    predictions = model.predict(img)
    pred = np.argmax(predictions,axis=1)

    pred2 = []
    for i in range(len(classes)):
        pred2.append(list(pred).count(i))

    if ctrl==1:
        print('La imagen '+file+' pertenece a la clase: ' +
                classes[pred2.index(max(pred2))])
        for i in range(len(classes)):
            print('   ' + classes[i] + ': ' +
                    str(round((pred2[i]/len(pred) * 100),2)) + '%')
```

Código 5-1. Función para predecir el modelo de cámara que captó una imagen

En primer lugar se recorta la imagen de test a predecir en trozos de 64x64, que serán las imágenes que se introduzcan en la red neuronal. Viendo los experimentos anteriores, las precisiones salieron bastantes altas, pero lógicamente sin alcanzar el 100%, pues no es perfecto. Esto quiere decir que de todos los recortes de la imagen de test, habrá una cierta cantidad (alta) en la que se acierte prediciéndolos y otra en la que se falle (baja). Es por ello que una vez sean predichos todos esos recortes que forman la imagen, basta con ver

cuál ha sido el modelo de los tres posibles qué más ha salido en las predicciones y esa será la predicción final de la imagen original, que al fin y al cabo es el único dato que le interesa al usuario.

5.4.1 Imágenes de test individuales

Para comenzar se va a probar el funcionamiento de la función anterior sobre varias imágenes de test de forma individual. Se realiza el procedimiento que se ha descrito anteriormente, calculando el porcentaje de recortes que se predicen como cada una de las cámaras y quedándonos con aquella que haya conseguido un mayor porcentaje.

Caso del experimento 1: iPhone 4S, Motorola Droid Maxx y Samsung Galaxy S4

```
La imagen (iP4s)64.jpg pertenece a la clase: iphone
  motorola: 5.83%
  samsung: 22.81%
  iphone: 71.36%
```

Figura 5-12. Predicción de una imagen de test aleatoria perteneciente a la clase *iphone*

```
La imagen (MotoMax)47.jpg pertenece a la clase: motorola
  motorola: 86.96%
  samsung: 6.64%
  iphone: 6.4%
```

Figura 5-13. Predicción de una imagen de test aleatoria perteneciente a la clase *motorola*

```
La imagen (GalaxyS4)27.jpg pertenece a la clase: samsung
  motorola: 13.93%
  samsung: 76.39%
  iphone: 9.68%
```

Figura 5-14. Predicción de una imagen de test aleatoria perteneciente a la clase *samsung*

Caso del experimento 2: LG Nexux 5X, Motorola Droid Maxx y Samsung Galaxy S4

```
La imagen (LG5x)11.jpg pertenece a la clase: lg
  lg: 86.32%
  motorola: 10.3%
  samsung: 3.38%
```

Figura 5-15. Predicción de una imagen de test aleatoria perteneciente a la clase *lg*

```
La imagen (MotoMax)35.jpg pertenece a la clase: motorola
  lg: 6.72%
  motorola: 91.87%
  samsung: 1.41%
```

Figura 5-16. Predicción de una imagen de test aleatoria perteneciente a la clase *motorola*

```
La imagen (GalaxyS4)65.jpg pertenece a la clase: samsung
  lg: 28.56%
  motorola: 1.69%
  samsung: 69.75%
```

Figura 5-17. Predicción de una imagen de test aleatoria perteneciente a la clase *Samsung*

Se observa que efectivamente, aunque las precisiones que se consiguieron inicialmente sobre los trozos no fueron del 100%, el hecho de que fueran porcentajes altos ayuda ahora notablemente a que estas predicciones de las imágenes completas y originales sean correctas.

5.4.2 Conjuntos de test completos

Para finalizar, se va a emplear la función anterior sobre todas las imágenes de test de cada uno de los tres modelos de cámaras. Para ello se emplea el siguiente script:

```
classes = ['motorola','samsung','iphone']
path= 'pruebas'
pred = []
cant = []
for phone in os.listdir(path):
    i=0
    pathphone = path + "/" + phone
    if os.path.exists(pathphone + "/cuts"):
        shutil.rmtree(pathphone + "/cuts")
    files = os.listdir(pathphone)
    for file in files:
        pred.append(predict(pathphone + "/",file,classes,0))
        i=i+1
    cant.append(i)

predindiv = []
for i in range(len(classes)):
    predindiv.append(pred[i*cant[i]:cant[i]*(i+1)].count(classes[i]))
    print('Conjunto de test: '+classes[i])
    print('  ' + str(predindiv[i]) + '/' + str(cant[i])
          +' imágenes predichas correctamente')
    print('  Precisión: '+str(round(predindiv[i]/cant[i]*100,2)) + '%')
print('PRECISIÓN FINAL: '+str(round(sum(predindiv)/sum(cant)*100,2))
        +'%')
```

Código 5-2. Script para calcular la precisión final de todo el conjunto de imágenes de test

Caso del experimento 1: iPhone 4S, Motorola Droid Maxx y Samsung Galaxy S4

```
Conjunto de test: motorola
  73/75 imágenes predichas correctamente
  Precisión: 97.33%
Conjunto de test: samsung
  70/75 imágenes predichas correctamente
  Precisión: 93.33%
Conjunto de test: iphone
  74/75 imágenes predichas correctamente
  Precisión: 98.67%
PRECISIÓN FINAL: 96.44%
```

Figura 5-18. Precisión final para el caso del experimento 1

Se observa que se obtiene una precisión mayor que el 85.17% que se consiguió en el experimento 1. Como se ha explicado con anterioridad, esto es debido a que antes se calculaba dicha precisión validando únicamente el conjunto total de trozos que se obtenía al recortar todas las imágenes de cada modelo, sin hacer nada más. Ahora se está aprovechando ese '85.17% de acierto de cada trozo' para poder calcular la precisión de una imagen mediante las predicciones individuales de sus trozos. De esta forma gracias al 85.17% obtenido anteriormente ahora se consiguen muy buenas predicciones de las imágenes al completo.

Por tanto ahora sí se podría decir que esta nueva precisión del **96.44%** es la verdadera precisión final que se ha conseguido con el modelo de red neuronal implementado para estos 3 modelos de cámaras.

Caso del experimento 2: LG Nexux 5X, Motorola Droid Maxx y Samsung Galaxy S4

```
Conjunto de test: lg
  68/75 imágenes predichas correctamente
  Precisión: 90.67%
Conjunto de test: motorola
  70/75 imágenes predichas correctamente
  Precisión: 93.33%
Conjunto de test: samsung
  65/75 imágenes predichas correctamente
  Precisión: 86.67%
PRECISIÓN FINAL: 90.22%
```

Figura 5-19. Precisión final para el caso del experimento 2

En este caso ocurre lo mismo, mientras que antes en el experimento 2 se conseguía un '76.97% de precisión por trozo', ahora al medirla por imagen se mejora consiguiendo un **90.22%,** que sería la precisión final que se consigue con el modelo de red neuronal implementado para el caso de estos 3 modelos de cámaras.

6 CONCLUSIONES Y LÍNEAS FUTURAS

A lo largo de este trabajo se ha podido ver un estudio de las técnicas existentes más comunes para la identificación del modelo de cámara en el ámbito forense así como una clasificación de las mismas. También se ha visto toda la teoría que hay detrás de las redes neuronales y se han descrito los pasos para el diseño y desarrollo de una red de estas características preparada para la identificación de cámaras. Finalmente, se han evaluado los resultados del sistema implementado en una fase de pruebas. En este capítulo final se estudia el resultado obtenido para después proponer posibles mejoras y líneas de investigación futuras.

A pesar de que han ido surgiendo diferentes problemas, principalmente a lo largo de la implementación del sistema debido a los altos requerimientos que una red neuronal convolucional supone, se han conseguido solventar a tiempo. De esta forma, se ha conseguido una red final funcional que cumple con los objetivos que se habían propuesto durante las primeras fases del trabajo.

6.1 Conclusiones

Deep Learning es el núcleo de este trabajo, elegido por sus altas capacidades y número de posibilidades para el análisis de imágenes. Sin embargo, no todo son ventajas al programar algoritmos de Deep Learning y redes neuronales convolucionales: diferentes capas sirven para diferentes propósitos, y cada parámetro e hiperparámetro importa mucho en el resultado final. Esto lo hace extremadamente complicado a la hora de intentar afinar la programación de un modelo de red neuronal, pareciendo más un arte que una ciencia para los que se adentran por primera vez en el área pues se requiere mucha experiencia e intuición para encontrar los valores óptimos de estos hiperparámetros, que se deben especificar antes de iniciar el proceso de entrenamiento para que los modelos entrenen mejor y más rápidamente.

Otro aspecto muy importante a tener en cuenta, además de una buena elección de capas e hiperparámetros, es la base de datos a emplear. Una correcta elección de la base de datos para entrenamiento es vital para llegar hasta los resultados deseados. La red debe conocer de igual manera todos aquellos elementos que se quieren predecir, por lo que los prototipos de entrenamiento deben ser equilibrados para todas las clases a detectar. Por tanto, disponer de una enorme cantidad de datos conlleva, aunque un entrenamiento más largo, soluciones muy buenas. De esta forma, uno de los resultados erróneos más comunes a la hora del entrenamiento de una red neuronal como es el subajuste, producido cuando el subconjunto de la base de datos que se emplea para entrenar no es lo suficientemente representativo de la tarea a aprender, implicando que el sistema tenga un error de entrenamiento grande y no que no sea capaz de generalizar en ningún caso, quedaría solventado. Por otro lado, en el capítulo de experimentos se pudo ver gracias a la validación cruzada, que tanto el modelo de red neuronal implementado como la base de datos empleada son generalistas, sin mostrar sobreajuste al no depender de las separaciones entre entrenamiento y test, otro de los resultados erróneos más comunes a la hora del entrenamiento.

Como se ha comentado en numerosas ocasiones, trabajar con redes neuronales convolucionales y con grandes bases de datos requiere un alto coste computacional. Para este trabajo se ha tenido la suerte de poder contar con un equipo de altas prestaciones (GPU incluida) y aun así se vio la necesidad de reducir la base de datos de 10 a 3 cámaras. Hay que tener siempre en mente este requisito cuando se trabaja con CNNs, ya que si no se cuenta con un equipo de estas características el entrenamiento puede durar días, sobre todo si la profundidad de las capas y la base de datos son muy grandes. No obstante siempre se puede

considerar el hecho de trabajar con las imágenes recortadas como se ha hecho en este trabajo, lo cual mejoró aún más el tiempo de cómputo de entrenamiento.

En cuanto a los resultados de los experimentos, que es quizás la parte más importante de todo el trabajo, decir que han sido muy satisfactorios. Cuando se estaban llevando a cabo las pruebas, al principio solamente se tuvieron en cuenta los resultados de precisión que se obtenían al pasar por la red neuronal los recortes de las imágenes de test, que fueron el torno al 80%. El hecho de pensar en una aplicación real, en la que al usuario solamente le interesa saber la predicción de una imagen de test sin tener en cuenta los recortes y los pasos que haga la red neuronal internamente, dio lugar a que se realizara un último experimento. Gracias a ese experimento final se consiguieron unos resultados de precisión de las imágenes completas bastante altos, de más del 90%, debido a los buenos resultados que se obtuvieron en las predicciones de los recortes.

Por último destacar el buen funcionamiento del framework Keras/TensorFlow sobre Python para trabajar en Deep Learning dado su alto rendimiento, pero sobre todo gracias a su multitud de funciones y librerías que facilitan el trabajo en numerosas ocasiones, como se ha visto a lo largo de la implementación de la red neuronal.

6.2 Líneas futuras

Con el objetivo de continuar y mejorar lo estudiado en este proyecto, se proponen a continuación posibles líneas futuras de trabajo.

En primer lugar, si se contase con un equipo aún más potente, se podría probar el modelo de red neuronal implementado con los 10 modelos de cámara de la base de datos completa y, en caso de que los porcentajes de precisión bajasen, bastaría con modificar las capas e hiperparámetros del modelo ajustándolos a los nuevos datos.

Otras de las mejoras que se podrían añadir al trabajo sería el poder predecir imágenes postprocesadas. Hoy en día es muy común procesar las imágenes con algún software o aplicación, bien con el fin de manipular su información o contenido, o por el simple hecho de subirla a una red social, donde la imagen sufre ciertas alteraciones y compresiones por la misma web o aplicación donde es subida. Esto haría que la imagen dejase de ser original, perdiendo parte de las huellas características de la cámara con la que fue tomada. Por tanto a la hora de identificar dicha imagen, ese postprocesado haría el proceso aún más complejo.

Para finalizar, comentar que cuando se presentaron en el capítulo del estado de arte algunos de los artículos que utilizaban redes neuronales para solventar el problema de la identificación de cámara, se vio que muchos de ellos agregaban capas de preprocesamiento. Con ello notaron mejoras en los resultados cuando previamente no lo eran tan buenos. En este trabajo, al haber conseguido unos resultados satisfactorios desde el principio no se ha visto la necesidad de emplear dichas capas. Sin embargo como se ha comentado anteriormente, en el caso de probar este modelo con la base de datos completa de 10 cámaras, si se diese el caso de que los porcentajes de precisión bajasen una buena solución podría ser probar estas capas de preprocesamiento que se presentan en dichos artículos.

REFERENCIAS

[1] A. Piva, "An overview on image forensics", Isrn Signal Process, 2013.

[2] IEEE Signal Processing Cup, https://signalprocessingsociety.org/get-involved/signal-processing-cup.

[3] IEEE Signal Processing Society, https://signalprocessingsociety.org/.

[4] M. Stamm, P. Bestagini y L. Marcenaro, "Forensic Camera Model Identification", Highlights from the IEEE Signal Processing Cup 2018 Student Competition, 2018.

[5] Forensic Camera Model Identification Challenge, https://www.kaggle.com/c/sp-society-camera-model-identification.

[6] R. Ramanath, W. Snyder y Y. Yoo, "Color Image Processing Pipeline in Digital Still Cameras", IEEE Signal Processing Magazine, 2004, vol. 22, pp. 34-43.

[7] M. Kirchner y T. Gloe, "Forensic camera model identification", Handbook of Digital Forensics of Multimedia Data and Devices, 2015, pp. 329-374.

[8] K. Kurosawa, K. Kuroki y N. Saitoh, "CCD fingerprint method-identification of a video camera from videotaped images", Proceedings of IEEE International Conference on Image Processing, 1999, pp. 537-540.

[9] Z. Geradts, J. Bijhold, M. Kieft, K. Kurosawa, K. Kuroki y N. Saitoh, "Methods for identification of images acquired with digital cameras", Proceedings of the Enabling Technologies for Law Enforcement and Security, 2001, vol. 4232, pp. 505-512.

[10] J. Lukas, J. Fridrich y M. Goljan, "Digital camera identification from sensor pattern noise", IEEE Transactions on Information Forensics and Security, 2006, vol. 1 (2), pp. 205-214.

[11] K. Choi, E. Lam y K. Wong, "Source camera identification using footprints from lens aberration", Proc. SPIE, Digital Photography, 2006, pp. 172-179.

[12] A. Dirik, H. Sencar y N. Memon, "Digital single lens reflex camera identification from traces of sensor dust", IEEE Transactions on Information Forensics and Security, 2008, vol. 3 (3), pp. 539–552.

[13] C. Li, "Source camera identification using enhanced sensor pattern noise", IEEE Transactions on Information Forensics and Security, 2010, vol. 5 (2), pp. 280–287.

[14] I. Amerini, R. Caldelli, V. Cappellini, F. Picchioni y A. Piva, "Estimate of PRNU noise based on different noise models for source camera identification", International Journal of Digital Crime and Forensics, 2010, vol. 2 (2), pp. 21-33.

[15] I. Caldelli, I. Amerini y F. Picchioni, "A DFT-Based Analysis to Discern Between Camera and Scanned Images", International Journal of Digital Crime and Forensics, 2010, vol. 2 (1), pp. 21-29.

[16] Y. Tomioka, Y. Ito y H. Kitazawa, "Robust digital camera identification based on pairwise magnitude relations of clustered sensor pattern noise", IEEE Transactions on Information Forensics and Security, 2013, vol. 8 (12).

[17] R. Li, C. Li y Y. Guan, "Inference of a compact representation of sensor fingerprint for source camera identification", Pattern Recognition, 2018, vol. 74, pp. 556–567.

[18] M. Kharrazi, H. Sencar y N. Memon, "Blind source camera identification", IEEE International Conference on Image Processing ICIP, 2004, vol. 1, pp. 709-712.

[19] O. Celiktutan, B. Sankur y I. Avcibas, "Blind identification of source cell-phone model", IEEE Transactions on Information Forensics and Security, 2008, vol. 3 (3), pp. 553–566.

[20] T. Gloe, Q. Shi y S. Katzenbeisser, "Feature-based forensic camera model identification", Transactions on Data Hiding and Multimedia Security, 2012, vol. 7228, pp. 42–62.

[21] S. Bayram, H. Sencar y N. Memon, "Improvements on source camera model identification based on CFA interpolation", Advances in Digital Forensics II, IFIP International Conference on Digital Forensics, Orlando, 2006.

[22] A. Swaminathan, M. Wu y K. Liu, "Nonintrusive component forensics of visual sensors using output images", IEEE Transactions on Information Forensics and Security, 2007, vol. 2 (1), pp. 91–106.

[23] T. Filler, J. Fridrich y M. Goljan, "Using sensor pattern noise for camera model identification", Proc. of the 15th IEEE International Conference on Image Processing (ICIP), 2008, pp. 1296–1299.

[24] A. AbdulWahab, A. Ho y S. Li, "Inter camera model image source identification with conditional probability features", Proc. of the 3rd IIEEJ Image Electronics and Visual Computing Workshop, 2012.

[25] G. Xu y Y. Shi, "Camera model identification using local binary patterns", Proc. IEEE Int Conference on Multimedia and Expo (ICME), 2012, pp. 1296–1299.

[26] B. Xu, X. Wang, X. Zhou, J. Xi y S. Wang, "Source camera identification from image texture features", Neurocomputing, 2016, vol. 207, pp. 131–140.

[27] Y. Hu, C. Li, X. Lin y B. Liu, "An improved algorithm for camera model identification using inter-channel demosaicking traces", Proceedings of 20th International Conference on Intelligent Information Hiding and Multimedia Signal Processing, 2012, pp. 325–330.

[28] C. Chen y M. Stamm, "Camera model identification framework using an ensemble of demosaicing features", IEEE International Workshop on Information Forensics and Security (WIFS), 2015, pp. 1–6.

[29] F. Marra, G. Poggi, C. Sansone y L. Verdoliva, "Evaluation of residual-based local features for camera model identification", New Trends in Image Analysis and Processing - ICIAP Workshop, 2015, pp. 11–18.

[30] A. Tuama, F. Comby y M. Chaumont, "Source camera identification using features from contaminated sensor noise", IWDW 2015, The 14th International Workshop on Digital-forensics and Watermarking, Proceedings as Lecture Notes in Computer Science (LNCS), pp. 83-93.

[31] A. Tuama, F. Comby y M. Chaumont, "Camera model identification based machine learning approach with high order statistics features", Proceedings of the 24th European Signal Processing Conference, 2016, pp. 1183–1187.

[32] A. Krizhevsky, I. Sutskever y G. Hinton, "Imagenet classification with deep convolutional neural networks", Proceedings of Advances in Neural Information Processing Systems, 2012, pp. 1097–1105.

[33] O. D. J. Russakovsky, H. Su, J. Krause, S. Satheesh, S. Ma y Z. Huang, "Imagenet large scale visual recognition challenge", Int. J. Comput. Vis., 2015, vol. 115 (3), pp. 211–252.

[34] C. Szegedy, W. Liu, Y. Jia, P. Sermanet y S. Reed, "Going deeper with convolutions", Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2015, pp. 1–9.

[35] K. He, X. Zhang, S. Ren y J. Sun, "Deep residual learning for image recognition", Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2016, pp. 770–778.

[36] J. Chen, X. Kang, Y. Liu y Z. Wang, "Median filtering forensics based on convolutional neural networks", IEEE Signal Process, 2015, vol. 22 (11), pp. 1849–1853..

[37] B. Bayar y M. Stamm, "A deep learning approach to universal image manipulation detection using a new convolutional layer", Proceedings of the 4th ACM Workshop on Information Hiding and Multimedia Security, 2016, pp. 5–10.

[38] A. Tuama, F. Comby y M. Chaumont, "Camera model identification with the use of deep convolutional neural networks", Proceedings of IEEE International Workshop on Information Forensics and Security, 2016, pp. 6-13.

[39] L. Bondi, L. G. D. Baroffio y P. Bestagini, "First steps toward camera model identification with convolutional neural networks", IEEE Signal Process., vol. 24 (3), 2017, pp. 259–263.

[40] I. Goodfellow, Y. Bengio y A. Corville, "Deep Learning", MIT Press, 2016.

[41] S. Russell y P. Norvig, "Artificial Intelligence: A Modern Approach", Prentice hall, 2009.

[42] A. Smola y S. Vishwanathan, "Introduction to Machine Learning", Cambridge University, 2008.

[43] K. Gurney, "An introduction to neural networks", University of Sheffield, 1997.

[44] A. Saad y M. Tareq, "Understanding of a Convolutional Neural Network", The International Conference on Engineering and Technology (ICET), 2017.

[45] M. Cilimkovic, "Neural Networks and Back Propagation Algorithm", Institute of Technology Blanchardstown, Dublin, 2015.

[46] N. Cui, "Applying Gradient Descent in Convolutional Neural Networks", Electrical and Computer Engineering, University of Massachusetts Lowell, 2018.

[47] A. Mikolajczyk y M. Grochowski, "Data augmentation for improving deep learning in image classification problem", 2018 International Interdisciplinary PhD Workshop (IIPhDW).

[48] GPU for Deep Learning, https://timdettmers.com/2019/04/03/which-gpu-for-deep-learning/.

[49] Amazon Web Services, https://aws.amazon.com/es/.

[50] Google Cloud TPU, https://cloud.google.com/tpu/.

[51] NVIDIA, https://www.nvidia.com/es-es/.

[52] NVIDIA Deep Learning, https://docs.nvidia.com/deeplearning/sdk/cudnn-install/.

[53] cuDNN Installation Guide, https://docs.nvidia.com/deeplearning/sdk/cudnn-install/.

[54] ¿Es Python el lenguaje del futuro?, https://www.paradigmadigital.com/dev/es-python-el-lenguaje-del-futuro/.

[55] Spyder, https://anaconda.org/anaconda/spyder.

[56] Anaconda, https://www.anaconda.com/.

[57] Numpy, https://www.numpy.org/.

[58] Pyllow, https://python-pillow.org/.

[59] Matplotlib, https://matplotlib.org/.

[60] OS, https://docs.python.org/3/library/os.html.

[61] Tensorflow, https://www.tensorflow.org/.

[62] Theano, http://deeplearning.net/software/theano/.

[63] Keras, https://keras.io/.

[64] Caffe, https://caffe.berkeleyvision.org/.

[65] Pytorch, https://pytorch.org/.

[66] Scikit-learn, https://scikit-learn.org/stable/.

[67] Keras Image Preprocessing, https://keras.io/preprocessing/image/.

[68] Keras Documentation, https://keras.io/models/sequential/.

[69] Keras Models, https://keras.io/applications/.

[70] ImageNet Dataset, http://www.image-net.org/.

[71] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever y R. Salakhutdinov, "Dropout: A Simple Way to Prevent Neural Networks from Overfitting", Journal of Machine Learning Research, 2014.

[72] S. Narkhede, Understanding Confusion Matrix, https://towardsdatascience.com/understanding-confusion-matrix-a9ad42dcfd62.

[73] Confusion Matrix, https://scikit-learn.org/stable/modules/generated/sklearn.metrics.confusion_matrix.html.

[74] G. Drakos, Cross Validation, https://towardsdatascience.com/cross-validation-70289113a072.