Facultad de Ciencias

Desarrollo y validación de algoritmos de Deep Learning para la clasificación de fundiciones de hierro

(Development and validation of Deep Learning algorithms for cast iron classification)

Trabajo de Fin de Grado
para acceder al

GRADO EN MATEMÁTICAS

Autor: Marta Bárcena Rodríguez

Director: Diego Ferreño Blanco

Co-Director: Lara Lloret Iglesias

Septiembre - 2022

Índice general

Agradecimientos

Me gustaría agradecer a Diego Ferreño y Lara Lloret por el tiempo dedicado a este trabajo y por ayudarme en todo lo que he necesitado durante estos meses. Ha sido un placer trabajar con vosotros.

Abstract

In the present work, we develop two algorithms based on deep learning techniques for the classification of samples of cast iron obtained by optical microscopy. The microstructure of cast irons determines their mechanical properties, so their correct classification is fundamental. As a starting point, the reference images provided by the standard *UNE-EN ISO 945-1* have been used to classify the images into 6 types of cast iron according to the shape of the graphite nodules. The work has been divided into two main objectives. The first one, is to train a CNN to classify an image with all graphite defects of the same type by identifying the nodule category. The second one, consists in generating images with a mix of nodules belonging to consecutive categories and training a CNN to classify each pixel of the image. For the development of both models it has been necessary to apply advanced techniques, such as transferred learning, as well as image processing techniques, such as data augmentation, to improve the performance of the algorithms.

Keywords: cast iron, deep learning, convolutional neural network, image classification

Resumen

En este trabajo se han desarrollado dos algoritmos basados en técnicas de *deep learning* para la clasificación de muestras de fundiciones de hierro obtenidas mediante microscopía óptica. La microestructura de las fundiciones determina sus propiedades mecánicas, por lo que su correcta clasificación es clave. Como punto de partida, se han empleado las imágenes de referencia proporcionadas por la norma *UNE-EN ISO 945-1* para clasificar las imágenes en 6 tipos de fundiciones según la forma de los nódulos de grafito. El trabajo se ha dividido en dos objetivos principales. El primer objetivo es entrenar una CNN tal que, dada una imagen con todos los defectos de grafito del mismo tipo, aprenda a clasificar a qué categoría pertenece la imagen. Para el segundo objetivo, se han generado imágenes en las que se mezclan manchas pertenecientes a categorías consecutivas y se ha entrenado una CNN que clasifica cada píxel de la imagen. Para el desarrollo de ambos modelos ha sido necesario aplicar técnicas avanzadas de *deep learning* como el aprendizaje transferido, así como técnicas de tratamiento de imágenes como el aumento de datos para mejorar el rendimiento de los algoritmos.

Palabras clave: fundición, aprendizaje profundo, red neuronal convolucional, clasificación de imágenes.

Capítulo 1

Introducción

Una fundición es una aleación de hierro con un contenido de carbono superior al 2 %. Generalmente, las fundiciones suelen tener un cierto porcentaje de silicio que hace que el carbono se convierta en grafito. La composición de la aleación y el tratamiento térmico determinan sus propiedades, por lo que el estudio de su microestructura y sus propiedades mecánicas es una importante rama de investigación.

Los tipos de fundiciones más comunes son: gris, esferoidal, blanca y maleable. En una fundición gris suele haber entre un 2,5 % y un 4,0 % de carbono y 1,0 % − 3,0 % de silicio. En estos casos, el grafito se presenta con forma de escamas o láminas. La fundición esferoidal es aquella en la que el grafito forma esferoides, mientras que en las fundiciones blancas el porcentaje de carbono es muy bajo ($< 3\,\%$) y suele aparecer en forma de cementita[1] en vez de grafito. Por último, si calentamos una fundición blanca con temperaturas en torno a los $800°C$, la cementita se descompone y forma grafito en forma de racimos o rosetas. Cada uno de estos tipos de fundiciones tiene distintas aplicaciones en la industria debido a las diferencias en su composición y, consecuentemente, en sus propiedades mecánicas. Por ejemplo, las fundiciones grises son bastante frágiles y poco resistentes a la tracción, pero tienen alta resistencia al desgaste y capacidad de amortiguamiento de vibraciones, por lo que se usan a menudo en piezas de maquinaria. En las fundiciones esferoidales, el grafito se presenta con forma esférica y con una distribución más uniforme que en las fundiciones grises, lo que hace que el material sea más resistente, tenaz y dúctil como el acero. Si examinamos una muestra de una fundición en el microscopio, lo que veremos es un fondo blanco con algunas manchas o defectos en color oscuro, que son lo que llamamos nódulos de grafito. Un ejemplo para cada tipo de fundición se puede observar en la imagen 1.1 [1-3].

El comportamiento mecánico del material viene determinado por la forma, distribución y tamaño de los nódulos, luego la clasificación del material según dichas propiedades es clave. En particular, en este trabajo nos centramos en la clasificación de los nódulos según su forma. Para ello, en el laboratorio se examina el material mediante un microscopio óptico y, comparando con una norma europea que proporciona imágenes de referencia, se realiza la clasificación [4]. Las muestras patrón que aparecen en la norma se muestran en la imagen 1.2. Como podemos observar, hay 6 tipos de muestras y se designan por números romanos del I al VI. La forma I es la que predomina en fundiciones grises, mientras que el resto pueden aparecer en fundiciones esferoidales y, además,

[1] La cementita (Fe_3C) es un compuesto formado por ferrita (α - Fe) y grafito (C)

las formas IV y V también se presentan en fundiciones maleables. El problema de este método es que la comparación visual llevada a cabo por un operario es susceptible al error humano, además de requerir experiencia y tiempo. En el Departamento de Ciencia e Ingeniería del Terreno y de los Materiales de la Universidad de Cantabria emplean un software para llevar a cabo esta clasificación, pero da resultados especialmente pobres para la forma V. En concreto, cuando analiza las imágenes de referencia de la norma, clasifica el 80 % de los nódulos de la forma V como forma IV.

En este trabajo proponemos una nueva solución a este problema: desarrollar algoritmos basados en técnicas de *deep learning* para la clasificación de imágenes. Se trata de algoritmos de aprendizaje automático que toman un conjunto de imágenes de entrada y, conociendo a qué clase pertenecen, se entrenan para aprender a identificar características o patrones para cada una de las clases. Es su gran capacidad de aprendizaje y generalización a partir de un conjunto de ejemplos representativo del problema original lo que justifica su elección en el ámbito de la clasificación de imágenes [5, 6]. Como veremos en el siguiente capítulo, las redes neuronales se inspiran en el proceso de aprendizaje humano, de manera que son capaces de simular nuestro comportamiento a la hora de extraer información a partir de un conjunto de datos y tomar una decisión fundamentada. De hecho, hemos podido encontrar trabajos donde aplican redes neuronales artificiales para clasificar fundiciones a partir de su morfología (distribución, homogeneidad, textura, etc.) [7-10] y a partir de la forma de los nódulos [11-13]. En este trabajo nos enfocamos también en la clasificación según la forma, pero vamos un paso más allá desarrollando dos modelos con un tipo específico de redes neuronales, las redes neuronales convolucionales (CNN).

Hemos estructurado este trabajo de la siguiente manera. En este primer capítulo comenzamos realizando una breve introducción a las redes neuronales a partir de la definición de inteligencia artificial y detallamos la motivación y objetivos de este trabajo. En el capítulo 2 se expone el desarrollo teórico de las redes neuronales convolucionales con su arquitectura y características, el proceso de entrenamiento y optimización del modelo y una serie de técnicas avanzadas y conceptos de *deep learning* que han sido necesarios a lo largo del desarrollo. En el capítulo 3 se detallan los pasos necesarios para crear el conjunto de datos de entrada de los modelos y se muestran ejemplos de las imágenes empleadas. El análisis de resultados se trata en el capítulo 4 y las conclusiones y perspectivas de trabajo futuro en el capítulo 5.

1.1. De la Inteligencia Artificial a las Redes Neuronales

La inteligencia artificial (AI) surge en los años 50 con el objetivo de automatizar las tareas que realizan los humanos. En sus primeros años, la inteligencia artificial se guiaba por lo que los humanos sabían programar, de manera que, basándose en reglas y lógica matemática, se definen el conjunto de características significativas para resolver el problema y se alimenta con estas reglas al algoritmo. Sin embargo, para muchas tareas es difícil saber qué características deben extraerse. Por ejemplo, si queremos desarrollar un algoritmo para detectar las imágenes en las que hay manzanas, podemos definir las formas y colores más comunes, pero es difícil describir exactamente el aspecto que tienen en términos de píxeles. Más aún, la tarea puede complicarse si aparecen sombras en la imagen, si la iluminación es distinta o si cambian los tamaños, entre otros. Precisamente, a principios de los 80, surge el campo del aprendizaje automático o *machine learning* para dar respuesta a la pregunta ¿puede un ordenador ir más allá de lo que sabemos programar y aprender por sí mismo a realizar una tarea? Los modelos de *machine learning*

necesitan que un programador los alimente con muchos ejemplos relevantes para dicha tarea y sus respuestas, pero, una vez dada la entrada, el modelo aprende automáticamente a reconocer las reglas y patrones que se derivan de ellos sin necesidad de que el programador los especifique. El campo del *machine learning* va de la mano con la estadística matemática, con la diferencia de que el primero suele tratar con conjuntos de datos grandes y complejos para los que el análisis estadístico clásico sería poco práctico. Más aún, es considerado una disciplina práctica en la que todas las ideas se fundamentan en teorías matemáticas pero la mayoría se demuestran empíricamente.

Una rama del *machine learning* es el aprendizaje profundo o *deep learning*. El término 'profundo' hace referencia a una nueva forma de aprendizaje de representaciones en los datos basada en capas. Los datos se procesan de forma jerárquica de manera que cada una de las capas obtiene representaciones cada vez más significativas. Llamamos representaciones a operaciones tales como cambios de coordenadas, proyecciones, traslaciones y, las más utilizadas, operaciones no lineales. De esta manera, las capas iniciales extraen conceptos más sencillos y, a partir de ellos, se definen conceptos cada vez más complejos y abstractos, logrando así una gran potencia y flexibilidad de aprendizaje. Es precisamente el número de capas existentes en un modelo lo que determina su profundidad. En la práctica podemos encontrar decenas o incluso cientos de capas de representaciones, lo que diferencia a los algoritmos de *deep learning* de otros tipos de aprendizaje, llamado aprendizaje superficial o *shallow learning* donde solamente hay una o dos capas. Con este enfoque, somos capaces de resolver problemas tales como la clasificación de imágenes o el reconocimiento por voz, tareas que los algoritmos tradicionales de *AI* no eran capaces de tratar.

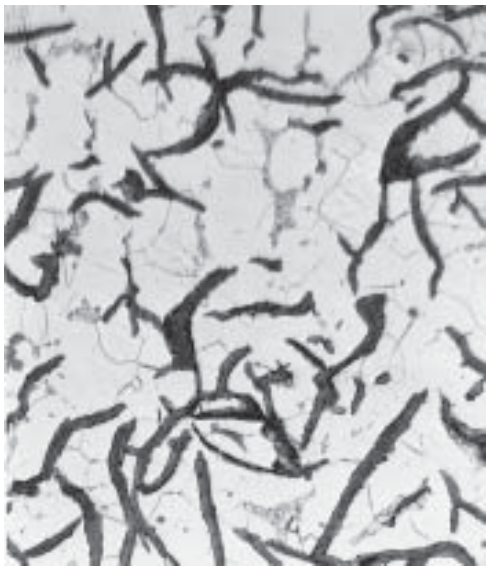
Generalmente, las representaciones en capas se aprenden mediante modelos de redes neuronales o *neural networks*, sobre las que hablaremos más en detalle en el siguiente capítulo. En la imagen 1.3 podemos ver un ejemplo en el que una red neuronal parte de una imagen original y va sucesivamente aplicando transformaciones en cada capa hasta llegar a un resultado final. Podemos pensar en estos modelos como una serie de filtros que toman la información original y la van desgranando en información cada vez más relevante y específica de la tarea que se quiere resolver, acercándonos al resultado esperado. La ventaja de las redes neuronales es que están mejor adaptadas a trabajar con datos de alta dimensionalidad, lo cual se ha convertido en el denominador común en la mayoría de las tareas computaciones de los últimos años debido al continuo aumento de la información [14].

1.2. Objetivos

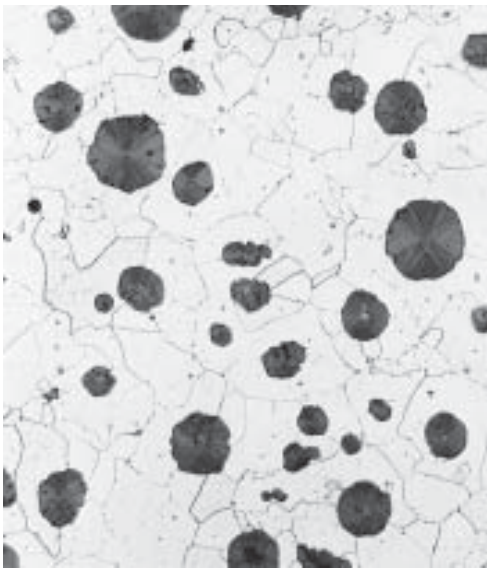
Partiendo de las imágenes de referencia que aparecen en la norma europea, en este trabajo proponemos la implementación y optimización de redes neuronales convolucionales para la clasificación de las principales formas del carbono en muestras de fundiciones. Se trata por lo tanto de resolver un problema de clasificación de imágenes en múltiples categorías donde a cada imagen se le asigna una única clase. Concretamente, el trabajo se ha divido en dos objetivos principales. El primer objetivo es entrenar una CNN tal que, dada una imagen con todos los defectos del mismo tipo, aprenda a clasificar a qué forma pertenece la muestra. En la segunda parte del trabajo, generamos imágenes en las que se mezclan manchas pertenecientes a categorías consecutivas, ya que es lo que generalmente esperamos encontrar en una muestra real en el laboratorio. La idea es que el algoritmo aprenda a identificar a qué categoría pertenece cada mancha, aunque, como detallaremos más adelante, hemos transformado esta tarea en un problema de segmentación

semántica, de manera que el algoritmo debe aprender a categorizar cada píxel de la imagen.

Al final de este trabajo queremos dar respuesta a las siguientes preguntas: ¿somos capaces de automatizar la clasificación de los nódulos mediante una red neuronal convolucional? ¿distingue esta red los diferentes tipos de formas con suficiente exactitud?

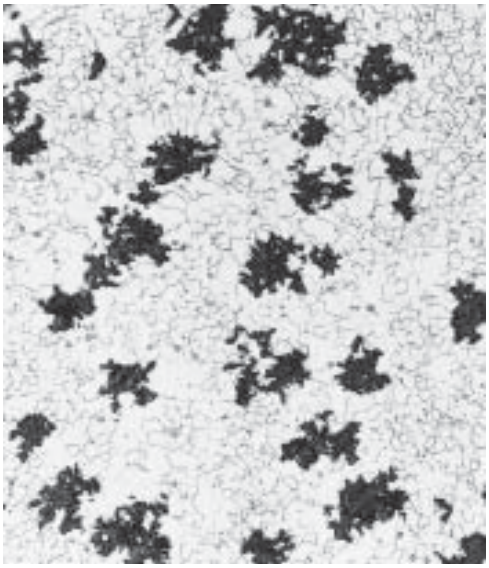


(a) Fundición gris



(b) Fundición esferoidal

(c) Fundición blanca



(d) Fundición maleable

Figura 1.1: Imágenes obtenidas con un microscopio óptico para cuatro tipos de muestras de fundiciones: gris, esferoidal, blanca y maleable [3].

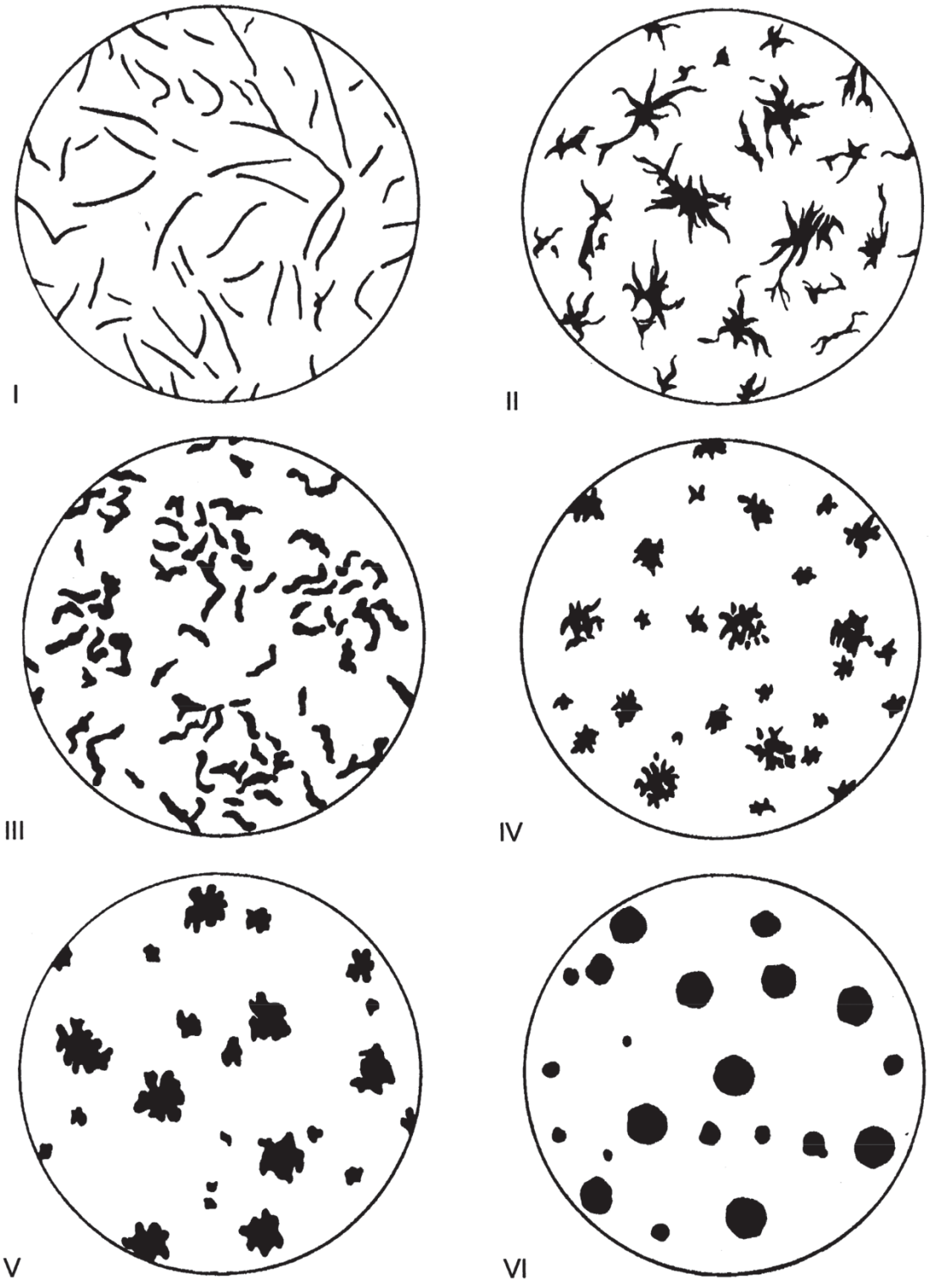


Figura 1.2: Imágenes de referencia para la clasificación de las principales formas del grafito en materiales de fundición [4].

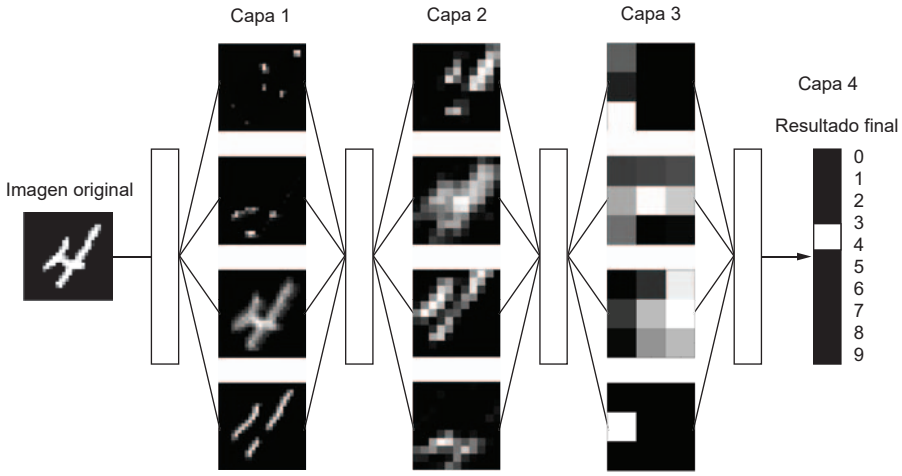


Figura 1.3: Esquema de un algoritmo de *deep learning* [6].

Capítulo 2

Redes neuronales para la clasificación de imágenes

Supongamos que queremos desarrollar un algoritmo que aprenda a determinar a cuál de las k clases pertenece un determinado conjunto de imágenes. Como veremos más adelante, si representamos las imágenes mediante matrices, la tarea se traduce en encontrar una función $f : \mathcal{M}_{n,m} \longrightarrow \{1, ..., k\}$ tal que $f(x) = y$ donde x representa una imagen (matriz de tamaño n x m) del conjunto de entrada e y es el entero que identifica la categoría que f asigna a x. La clasificación se llevará a cabo según las características de cada imagen tales como bordes, áreas, formas, intensidad, etc.

En este trabajo presentamos dos modelos de redes neuronales que resuelven el problema de clasificación de imágenes presentado en el capítulo anterior. Las redes neuronales son algoritmos que se han desarrollado tomando como ejemplo el proceso de aprendizaje que se produce en el cerebro humano. Están compuestas por una serie de funciones que permiten al algoritmo extraer información y aprender patrones a partir de un conjunto inicial al que llamamos conjunto de entrenamiento y, una vez que el algoritmo aprende a identificar las características más representativas de dicho conjunto, es capaz de hacer predicciones fundamentadas para otros datos con características similares.

Las funciones que constituyen una red neuronal se llaman neuronas artificiales y se agrupan formando capas alineadas. Hay distintos tipos de capas dentro de una red. La primera de ellas, que es la que recibe los datos de entrada, se llama capa visible. La capa final, que es la que retorna la solución del problema, se llama capa de salida y el resto de las capas intermedias se llaman capas ocultas. El algoritmo debe aprender a determinar qué patrones son útiles para llevar a cabo la tarea de clasificación y, para ello, se van encadenando las capas de manera que cada una de ellas actúa como un filtro cada vez más refinado. Las transformaciones que implementan las capas están parametrizadas por sus pesos y el objetivo del algoritmo es optimizar los valores de éstos de manera que las transformaciones que se aplican a los datos den predicciones lo más ajustadas posible. En la imagen 2.1 se muestra la representación gráfica de la arquitectura de una red neuronal con 8 neuronas en la capa inicial, 3 capas ocultas con 9 neuronas cada una y una capa final con 4. A modo de ejemplo, imaginemos un modelo en el que la primera capa aprende a reconocer los bordes de una imagen a partir de la comparación de píxeles vecinos. A

partir de la descripción de los bordes, una segunda capa es entrenada para distinguir esquinas a partir de conjuntos de bordes y la tercera para identificar determinados objetos asociados a conjuntos de esquinas y bordes. Esta descripción de la imagen en términos de los objetos que las capas anteriores han identificado hace que la capa final sea capaz de reconocer a qué clase pertenecen los objetos, y dicha clasificación será la salida del modelo. Se trata de un ejemplo muy simplificado, pero la idea principal es que las capas iniciales son las que extraen las características más generales (tales como líneas rectas, curvas, bordes, etc.) mientras que las intermedias y finales extraen los patrones más específicos de cada tarea (objetos).

Las neuronas son las unidades que forman las capas. Se conectan entre ellas y a través de dichas conexiones se transmiten información de unas a otras. Podemos representar cada neurona como una función tal que su conjunto de salida sea el conjunto de entrada de la siguiente neurona.

Definición 1 - *Neurona Artificial.* *Una neurona es una función que viene dada por la siguiente expresión:*

$$\phi : \mathbb{R}^n \longrightarrow \mathbb{R}^{n'}$$
$$x \longrightarrow \varphi(W^T x + b)$$

donde φ es una función de activación y $W \in \mathbb{R}^n$, $b \in \mathbb{R}$ son los pesos de la neurona. Al valor que devuelve la neurona se le llama señal de salida y actúa como valor de entrada de la siguiente neurona.

Definición 2 - *Red Neuronal.* *Dados $l, m \in \mathbb{N}$, una red neuronal se define como la composición de N funciones ϕ_i tal que:*

$$\phi : \mathbb{R}^l \longrightarrow \mathbb{R}^m$$
$$x \longrightarrow \phi_N \circ ... \circ \phi_2 \circ \phi_1(x)$$

donde cada función ϕ_i representa la $i-$ésima capa de la red.

En particular, en un problema de clasificación, el tamaño de salida m será igual al número de categorías.

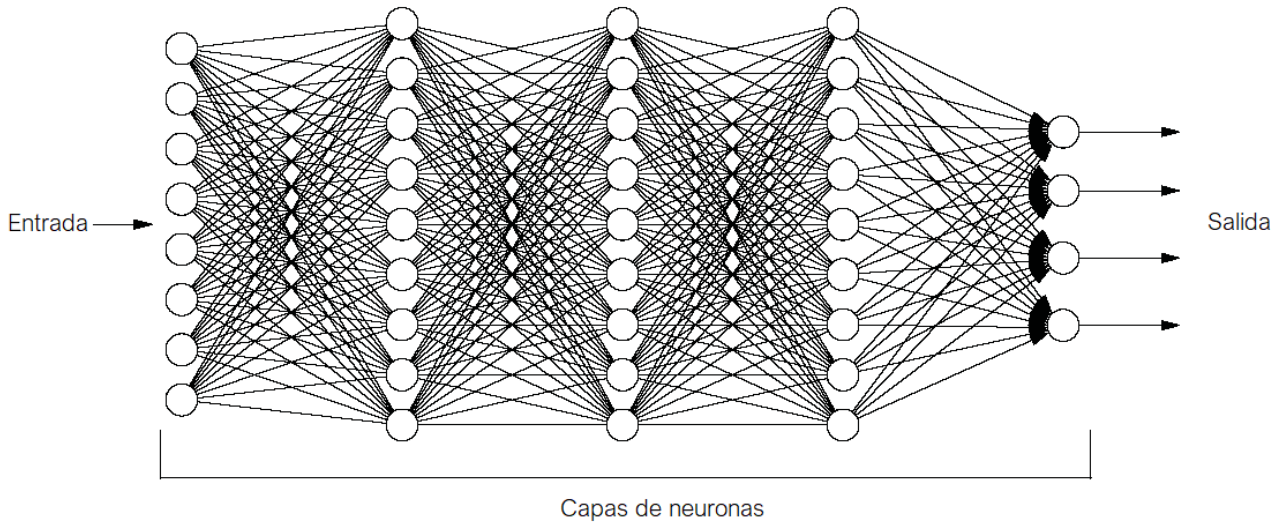


Figura 2.1: Representación gráfica de la arquitectura de una red neuronal [15].

2.1. Redes neuronales convolucionales

Las redes neuronales convolucionales (CNN) son un tipo de redes neuronales muy potentes. La principal diferencia entre una red convolucional y una tradicional es que la primera puede aprender a identificar patrones o características sin importar la estructura espacial, es decir, aprende a identificar patrones locales que luego pueden aplicarse al resto de la imagen. De esta manera, cuando una CNN aprende a distinguir un patrón en la esquina superior derecha de la imagen, es capaz de reconocerlo en cualquier otra parte, mientras que una red tradicional debería volver a ver el mismo patrón en otro punto de la imagen para ser capaz de reconocerlo. Para un problema como el nuestro en el que podemos encontrar manchas en diferentes posiciones y con distintas orientaciones y rotaciones, este método resulta muy eficiente ya que una CNN necesitará menos imágenes para aprender patrones generales. En cuanto a la arquitectura, en una red neuronal tradicional todas las neuronas de una capa están conectadas con todas las de la capa siguiente. Como veremos más adelante, un ordenador procesa imágenes en color como matrices de tamaño $n \times m \times 3$, donde n es el largo y m el ancho. Con imágenes de este tamaño, una sola neurona de la primera capa tendría $n \cdot m \cdot 3$ pesos, luego para una imagen de un tamaño relativamente pequeño como puede ser 100 x 100 x 3, esto supondría 30 mil pesos sólo en una neurona. Si queremos usar imágenes un poco más grandes, el coste computacional se vuelve excesivamente alto. Por otra parte, las redes convolucionales tienen una arquitectura diseñada para capturar la información más relevante y eliminar la que sea redundante, permitiendo reducir considerablemente el coste computacional. Estos modelos se denominan redes convolucionales ya que aplican la operación de convolución en, al menos, una de sus capas.

Definición 3 - *Producto de convolución.* *Dadas dos funciones g y f definidas en todo $\mathbb{R}$, se llama producto de convolución a la función $(g * f)$ tal que para cada $x \in \mathbb{R}$ se tiene:*

$$(g * f)(x) = \int_{-\infty}^{\infty} g(x - y) f(y) dy$$

y en el caso discreto:

$$(g * f)(x) = \sum_{y=-\infty}^{\infty} g(x - y) f(y)$$

Para este trabajo sólo consideramos la operación anterior en los casos para los que la integral está bien definida o, en el caso discreto, la serie es convergente. En el contexto de las redes convolucionales, se entiende la anterior operación como una media ponderada donde g actúa como peso o *kernel*, x es un elemento del conjunto de entrada y $(g * f)(x)$ es una matriz de salida. Cuando trabajamos con datos en un ordenador, resulta más conveniente usar la definición para el caso discreto.

2.1.1. Capas de una red neuronal convolucional

En una CNN hay principalmente tres tipos de capas: capas convolucionales, capas de agrupamiento o *pooling* y capas completamente conectadas o *fully connected*. Basta con que la red tenga una sola capa que realiza la operación de convolución para que sea considerada una red convolucional, pero en la práctica este tipo de redes aplican la operación de convolución en la mayoría de sus capas.

Comenzamos detallando la notación con un par de definiciones.

Definición 4 - *Matriz con coeficientes reales.* *Denotamos por $\mathcal{M}_{n,m}(\mathbb{R})$ al conjunto de todas las matrices de tamaño $n \times m$ cuyos coeficientes están en $\mathbb{R}$. Sea $M \in \mathcal{M}_{n,m}(\mathbb{R})$:*

$$M = \begin{pmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,m} \\ \vdots & \vdots & \vdots & \vdots \\ a_{n,1} & a_{n,2} & \dots & a_{n,m} \end{pmatrix}$$

y usamos la notación $a_{i,j}$ $\forall i, j \in \mathbb{N}$ tal que $1 \leq i \leq n, 1 \leq j \leq m$ para identificar cada uno de los elementos de la matriz.

Definición 5 - *Producto de Hadamard.* *Se define como multiplicación elemento a elemento o Producto de Hadamard para matrices reales la operación interna entre matrices del mismo tamaño definida de la siguiente manera:*

$$\bigodot : \mathcal{M}_{n,m}(\mathbb{R}) \times \mathcal{M}_{n,m}(\mathbb{R}) \longrightarrow \mathcal{M}_{n,m}(\mathbb{R})$$

tal que $A \bigodot B = M$, donde $m_{i,j} = a_{i,j} \cdot b_{i,j}$ $\forall i, j \in \mathbb{N}$ tal que $1 \leq i \leq n$, $1 \leq j \leq m$.

En una capa de convolución tenemos una matriz de entrada (una imagen) y un núcleo o *kernel*. El núcleo es una matriz cuadrada que suele ser de tamaño 1x1, 3x3, 5x5 o 7x7 con diferentes coeficientes también llamados pesos. Esta matriz se desplaza a lo largo de la matriz de entrada y aplica la operación de convolución a cada caja de la matriz, obteniendo como resultado una matriz de salida. En realidad, no se aplica un sólo núcleo, sino una concatenación de ellos a la que llamamos filtro. Por ejemplo, si nuestro conjunto de entrada son imágenes en color con 3 canales, los filtros serán matrices de 3 dimensiones de manera que cada núcleo actúa sobre uno de los canales de la imagen. El resultado es una matriz de salida a la que llamamos matriz de características o *feature map* (imagen 2.2). Cada una de estas matrices contiene información acerca de los *features* de cada píxel en relación con los píxeles adyacentes, por ejemplo, dónde hay bordes, líneas rectas, curvas, objetos, etc. [16, 17]. La arquitectura de una red convolucional está diseñada de manera que la imagen original se va comprimiendo en cada capa a la vez que el número de *feature maps* aumenta. Como hemos comentado anteriormente, una red neuronal va filtrando poco a poco la información, de manera que los primeros mapas contienen información más sencilla como líneas o formas y los mapas más próximos a la salida del modelo identifican objetos y patrones más complejos y específicos de las imágenes del problema.

En la imagen 2.3 se muestra gráficamente cómo se aplica la operación de convolución a una imagen. Partimos de la imagen de entrada de la izquierda y del núcleo (azul). El núcleo, de tamaño 3x3, se va desplazando por toda la matriz, de manera que se toman cajas de la matriz de entrada de tamaño 3x3 y se realiza la multiplicación elemento a elemento (definición 5) como se detalla en la imagen. El resultado de esta operación es un valor real (en amarillo). Se repite este proceso tomando cajas a lo largo y ancho de toda la imagen de entrada y el resultado final es una matriz de *features* del mismo tamaño que la inicial, donde cada elemento de salida son las sumas ponderadas de los elementos de entrada y los pesos son los valores del núcleo. En este ejemplo observamos que en la matriz de salida la red ha aprendido a reconocer los bordes a partir de los valores de los píxeles adyacentes.

Por otra parte, las capas de agrupamiento son capas sencillas que sirven para generalizar, consolidar y agrupar los *features* aprendidos por las capas de convolución. Si encadenamos convoluciones,

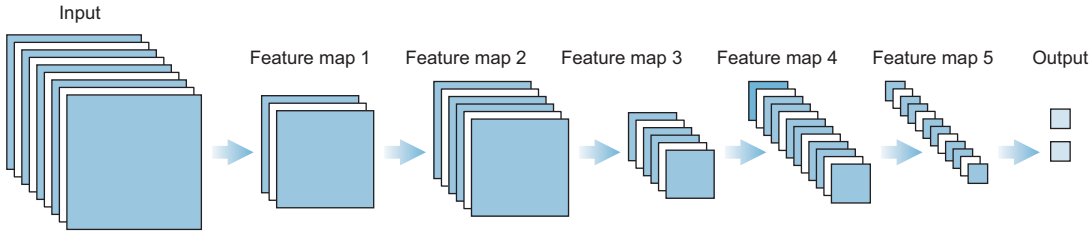


Figura 2.2: Representación gráfica de los *feature maps* extraídos por una CNN [15]. A medida que avanzamos en la red, los mapas aprenden patrones más complejos y específicos de la tarea que se está resolviendo, y los filtros capturan cada vez regiones más grandes de la imagen.

rápidamente tenemos un número de píxeles extremadamente alto que puede hacer que el procesamiento se vuelva lento y costoso. Para reducir la complejidad y el coste computacional, se intercalan capas de agrupamiento entre las capas convolucionales para eliminan los *features* más redundantes a la vez que mantienen los más significativos, haciendo así que las imágenes sean más fáciles de procesar. Esta transformación se realiza a través de un *kernel* que va seleccionando cajas de los mapas y, o bien toma la media de todos los valores de la caja (*average pooling*), o bien el máximo (*max pooling*), como se puede ver en la imagen 2.4. De acuerdo con la bibliografía, el segundo método funciona mejor, ya que los mapas codifican dónde están determinados patrones en la imagen original, por lo que los resultados son mejores cuando tomamos la información de dónde dichos patrones alcanzan el máximo, en vez de tomar el promedio de la información contenida en cada caja [5, 15]. En la imagen 2.2 podemos apreciar el efecto de capas de agrupamiento entre los mapas 2 y 3, donde se observa cómo las dimensiones a lo largo y ancho han disminuido.

Definición 6 - *Capa de agrupamiento max pooling.* *Dada una matriz de tamaño* $m \times n \times 3$ *y un kernel de tamaño* $h \times h \times 3$*, se define una capa de agrupamiento de tipo max pooling como una función* β *tal que:*

$$\beta : \mathcal{M}_{n,m,3}(\mathbb{R}) \longrightarrow \mathcal{M}_{n',m',3}(\mathbb{R})$$
$$M \longrightarrow M'$$

tal que

$$m'_{i,j,l} = \begin{cases} \max\{m_{a,b,c} : l = c, h(t-1)+1 \leq a, b \leq ht+1\} \ \ \forall i,j < h \\ \max\{m_{a,b,c} : l = c, h(t-1)+1 \leq a \leq n, h(t-1)+1 \leq b \leq m\} \ \ \forall i,j = h \end{cases}$$

donde $m'_{i,j,l}$ *los coeficientes de la matriz* M' *y* $m_{a,b,c}$ *los de* M*.*

Nótese que para las capas de tipo *average pooling* basta sustituir el máximo por la media de los coeficientes de la caja seleccionada.

Una vez que las capas convolucionales y de agrupación han extraído y combinado los patrones más representativos, se usan las capas completamente conectadas para realizar combinaciones no lineales de *features* y generar las predicciones de la red. Se llaman totalmente conectadas ya que cada neurona está conectada con todas las neuronas de la siguiente capa y solemos encontrar este tipo de capas en la parte final de la arquitectura de cualquier red tradicional actuando como capas de clasificación [19]. En el caso de CNN, estas capas reciben como entrada los *feature maps* y toman decisiones fundamentadas a partir de la información que estos contienen. También las encontramos en la bibliografía con el nombre de capas densas o *feed forward*.

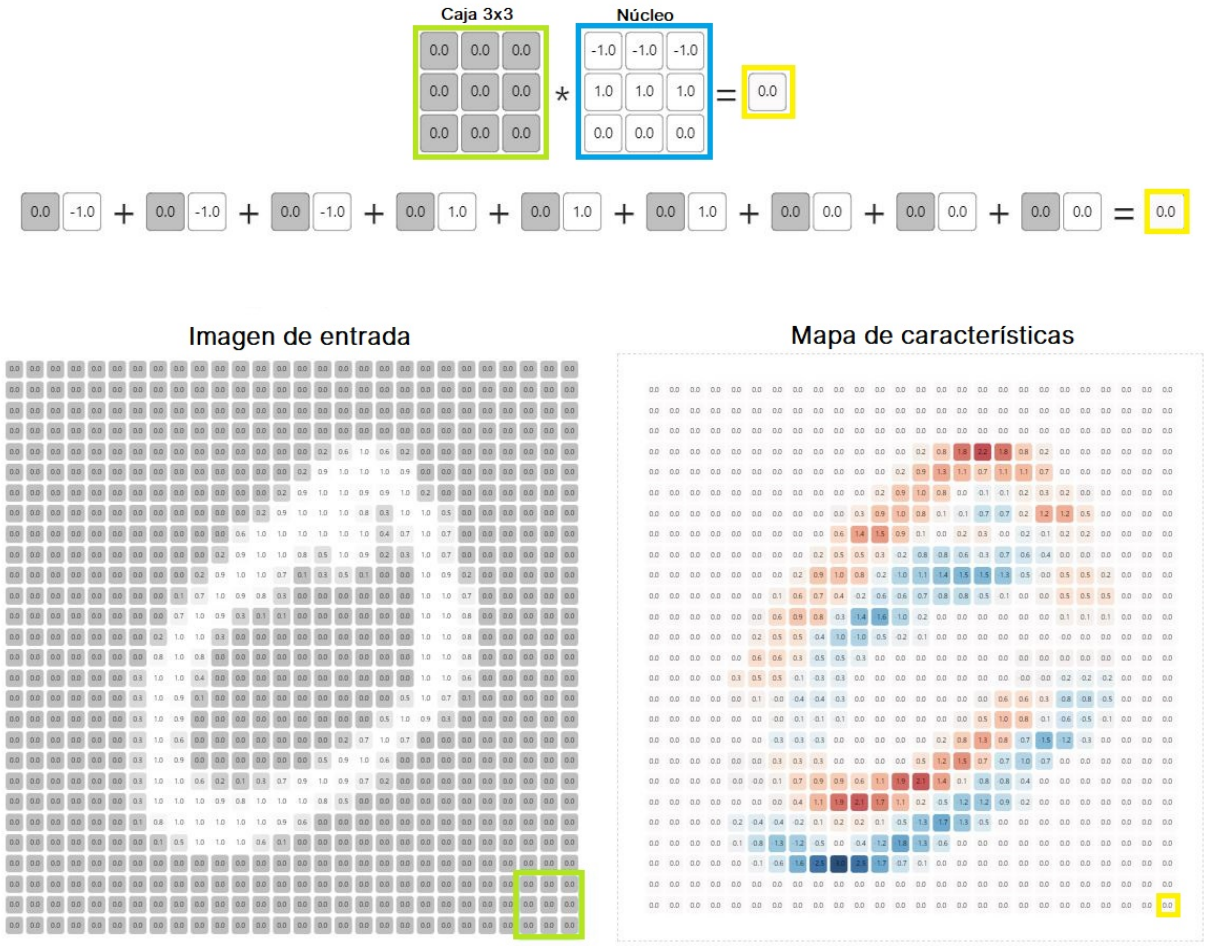


Figura 2.3: Representación visual de cómo funciona una capa de convolución aplicada a una imagen. A partir de una matriz de entrada de tamaño 28x28 y un kernel 3x3, el resultado es una matriz de *features* de tamaño 28x28 cuyos coeficientes (en amarillo) se calculan aplicando el Producto de Hadamard entre cada caja de la matriz de entrada (en verde) y el núcleo (en azul). En el *feature map* observamos que la capa ha identificado dónde se encuentran los bordes del objeto. Esta visualización ha sido creada con [18].

Definición 7 - *Capa completamente conectada.* *Dadas m neuronas, se define una capa completamente conectada como una función δ tal que:*

$$\delta : \mathbb{R}^m \longrightarrow \mathbb{R}^{m'}$$

$$x \longrightarrow \varphi(W^T x + b)$$

donde φ es la función de activación y $W \in \mathcal{M}_{m,m'}(\mathbb{R}), b \in \mathbb{R}^m$ son los pesos de las m neuronas.

Observando la definición anterior, vemos que el vector de entrada es $x \in \mathbb{R}^m$ pero las salidas de las capas de convolución y agrupamiento son matrices $\mathcal{M}_{m_1,m_2}(\mathbb{R})$. Para transformar una matriz de este tipo en un vector de tamaño $(m_1 \cdot m_2, 1)$ usamos capas de aplanamiento o *flatten*.

Definición 8 - *Capa de aplanamiento.* *Dada una matriz en $\mathcal{M}_{m_1,m_2}(\mathbb{R})$, se define una capa*

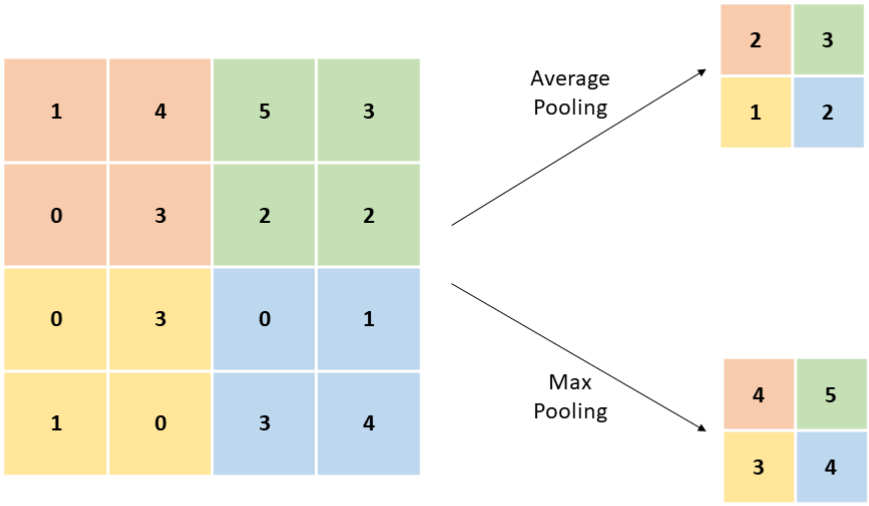


Figura 2.4: Representación gráfica de una capa de agrupamiento para un kernel 2x2 y los métodos *average pooling* arriba y *max pooling* abajo.

de aplanamiento como una función ρ *tal que:*

$$\rho : \mathcal{M}_{m_1,m_2}(\mathbb{R}) \longrightarrow \mathbb{R}^{m_1 m_2}$$
$$a_{i,j} \longrightarrow a_{j+m_1(i-1)} \quad \forall 0 \leq i \leq m_1, 0 \leq j \leq m_2$$

Estas capas por lo tanto no extraen información nueva, sino que definen una biyección entre matrices de salida y vectores de entrada para reestructurar la información de acuerdo a las necesidades de la arquitectura de la red.

Función de activación

Volviendo a la definición 1, vemos que la neurona primero realiza una suma ponderada dada por la expresión $W^T x + b = \sum_i W_i x_i + b$ y después aplica una función φ que llamamos función de activación, la cual determina si la neurona se activa o no. En el caso más sencillo, se trata de una función cuyo conjunto de entrada son los pesos de la neurona, pero, por lo general, se buscan funciones de activación que no sean lineales para transformar la suma ponderada de la expresión anterior en un resultado no lineal. Esto permite combinar los pesos de entrada de formas más complejas logrando así una mayor capacidad de aprendizaje que si solamente se pudiesen capturar relaciones lineales.

Hay varios tipos de funciones de activación y se suelen elegir según el tipo de problema y cuál sea el formato de la salida que se necesita. Aunque estas funciones se apliquen a las neuronas, se definen por capas, de manera que todas las neuronas de una capa tienen la misma función

de activación. La función de activación que más se usa para este tipo de redes neuronales es la función *ReLU* (*Rectified Linear Unit*):

Definición 9 - *Función ReLu.* *Se define la función ReLU (Rectified Linear Unit) de la siguiente manera:*

$$\begin{aligned} f : \mathbb{R} &\longrightarrow \mathbb{R} \\ x &\longrightarrow \text{máx}\{x, 0\} \end{aligned}$$

Según esto, si $f(x) = 0$, significa que el valor de salida de la neurona es 0, por lo que la siguiente neurona no se activa y se corta el flujo de información en esa rama de la arquitectura.

El motivo por el que esta función se usa a menudo es su diferenciabilidad. Como veremos más adelante, durante el proceso de entrenamiento de la red es necesario calcular las derivadas de las funciones. A pesar de que esta función no sea diferenciable en 0, su derivada es 1 en $\mathbb{R}^*$, lo que hace que computacionalmente sea muy eficiente.

Por otra parte, para un problema de clasificación en múltiples categorías como es el que se presenta en este trabajo, la función de activación que se usa en la capa final es *softmax*. Esta función convierte un vector de k números en un vector de k probabilidades, una para cada clase, de manera que la clase predicha será la que haya obtenido una mayor probabilidad. Para que esta función esté bien definida, la suma de todas las probabilidades debe ser 1. Definimos la función de la siguiente manera:

Definición 10 - *Función softmax.* *La función de activación softmax viene definida por la siguiente expresión:*

$$\begin{aligned} g : \mathbb{R}^k &\longrightarrow [0, 1]^k \\ g(x_i) &= \frac{e^{x_i}}{\sum_{j=1}^{k} e_k^x} \quad \forall i = 1, ..., k \end{aligned}$$

donde k es el número de categorías.

2.2. Entrenamiento y optimización de una red neuronal

Como hemos adelantado antes, en el proceso de entrenamiento de una red neuronal la información va pasando por las sucesivas capas, de manera que la salida de una capa es la entrada de la siguiente y la manera en la que esto ocurre depende de los pesos asignados a las neuronas. Cuando la información llega a la capa final, realiza predicciones sobre los datos de entrada y éstas se comparan con las etiquetas reales de dichos datos. En el caso de un problema de clasificación como el que detallamos aquí, la predicción consiste en asignarle una categoría a cada imagen y la etiqueta será la categoría real a la que pertenece. El objetivo es que el modelo aprenda a clasificar las imágenes con el menor error posible, es decir, queremos minimizar la distancia entre la predicción y la etiqueta. El entrenamiento se convierte por lo tanto en un problema de optimización.

El proceso de optimización del error en redes neuronales consiste en ajustar los pesos de cada neurona, a los que llamamos parámetros entrenables del modelo, hasta encontrar los valores

óptimos que minimizan el error. Este error viene determinado por la función de pérdida y el optimizador es el algoritmo que usa la función de pérdida para actualizar los pesos de la red. Habitualmente no podemos garantizar que el algoritmo vaya a alcanzar el mínimo global de la función de pérdida en un tiempo razonable, ya que esto puede suponer un coste computacional excesivamente alto, pero podemos conformarnos con encontrar un mínimo local o un valor de la función de pérdida lo suficientemente pequeño [20].

Función de pérdida

La función de pérdida se usa para medir cómo de bien el modelo predice los resultados. Hay muchas funciones de pérdida que podemos usar y su elección depende del problema que se quiera resolver. Para desarrollar la noción de función de pérdida comenzamos dando la definición de entropía.

Definición 11 - *Entropía.* *La entropía $E(F) \in \mathbb{R}$ es una medida de la incertidumbre asociada a una distribución $F(x)$. En el caso discreto para un conjunto de k puntos en $\mathbb{R}$ se calcula como:*

$$E(F) = -\sum_{i=1}^{k} F(x_i) \log F(x_i)$$

Si desconocemos la distribución real F, podemos aproximarla mediante una distribución G y definimos entropía cruzada (*cross-entropy*) de la siguiente manera:

Definición 12 - *Entropía cruzada.* *La entropía cruzada $E(F, G) \in \mathbb{R}$ es una medida de la incertidumbre asociada a una distribución $G(x)$ para un conjunto de k puntos en $\mathbb{R}$ cuya distribución real es $F(x)$:*

$$E(F, G) = -\sum_{i=1}^{k} F(x_i) \log G(x_i)$$

en el caso discreto para un conjunto de k puntos.

Si $G(x) = F(x)\ \forall x$, entonces las dos definiciones anteriores son equivalentes.

Definición 13 - *Divergencia Kullback-Leibler.* *La divergencia KL o divergencia Kullback-Leibler, también conocida como entropía relativa mide la diferencia entre la entropía y la entropía cruzada:*

$$D_{KL}(F, G) = E(F, G) - E(F) = \sum_{i=1}^{k} F(x_i)(\log F(x_i) - \log G(x_i))$$

De la definición anterior observamos que si $E(F, G) \longrightarrow E(F)$, o lo que es lo mismo, $F(x) \longrightarrow G(x)$ en cada punto, $D_{KL}(F, G) \longrightarrow 0$ y, puesto que la entropía de la distribución F es constante con respecto a G, minimizar $D_{KL}(F, G)$ es equivalente a minimizar $E(F, G)$. La tarea de optimización consiste por lo tanto en encontrar una función G que minimice la entropía cruzada $E(F, G)$.

La función de pérdida más común en problemas de clasificación donde se usa *softmax* como función de activación final es precisamente la entropía cruzada categórica (*categorical cross-entropy*) [15].

Definición 14 - *Función categorical cross-entropy.* *Sea k el número de clases del problema, se define la función categorial cross-entropy de la siguiente manera:*

$$\lambda : [0,1]^k \times [0,1]^k \longrightarrow \mathbb{R}$$

$$(y, y_{pred}) \longrightarrow -\sum_{i=1}^{k} y^{(i)} \log y_{pred}^{(i)}$$

donde $y^{(i)}$ representa la probabilidad asociada a la etiqueta real y $y_{pred}^{(i)}$ es la probabilidad dada por la función softmax para la $i-$ésima clase. Nótese que existe un único i tal que $y^{(i)} = 1$, por ser el valor objetivo, luego el sumatorio anterior sólo tiene un término distinto de cero.

La elección de la función de pérdida es una parte muy importante del desarrollo de un modelo ya que su comportamiento juega un papel clave en el proceso de optimización del algoritmo. Cada función de pérdida penaliza de distinta manera los errores que comete el modelo al realizar las predicciones y el problema que tiene la función anterior es que penaliza de la misma manera todas las categorías. Cuando hay un desequilibrio de clases, es decir, hay más ejemplos de una clase que de otra, se corre el riesgo de que el modelo aprenda a predecir con mucha exactitud la clase mayoritaria y no de importancia a los errores en el resto de clases con lo que estaremos entrenando un modelo sesgado. Una mejor alternativa en estos casos es la función de pérdida focal o *focal loss*. Esta función parte de las probabilidades que define la función *softmax* para cada clase y penaliza aquellas con menor probabilidad, es decir, las que el modelo tiene más dificultades en identificar, restando importancia a las clases que predice con seguridad. Con la ponderación descendiente o *down weighting* aplicada mediante un factor γ, conseguimos que el modelo se enfoque en mejorar la certeza con la que predice las clases minoritarias [21, 22].

Definición 15 - *Función de pérdida focal.* *Sea k el número de clases del problema, se define la función focal de la siguiente manera:*

$$\lambda : [0,1]^k \times [0,1]^k \longrightarrow \mathbb{R}$$

$$(y, y_{pred}) \longrightarrow -\sum_{i=1}^{k} y^{(i)} \cdot \alpha_i \cdot (1 - y_{pred}^{(i)})^{\gamma} \log y_{pred}^{(i)}$$

donde $y^{(i)}$ representa la probabilidad asociada a la etiqueta real y $y_{pred}^{(i)}$ es la probabilidad dada por la función softmax para la $i-$ésima clase. $\alpha_{i,j} \in (0,1)$ es un parámetro que se define según la frecuencia con la que aparece cada clase en el conjunto de entrada y $\gamma \in \mathbb{R}^+$ es el coeficiente que define la ponderación descendiente sobre el factor $(1 - y_{pred}^{(i)})$.

Según la expresión anterior, el factor $(1 - y_{pred}^{(i)})^{\gamma}$ sirve para reducir la función de pérdida, de manera que si una imagen se clasifica mal y su probabilidad asociada es pequeña, $(1-y_{pred}^{(i)})^{\gamma} \longrightarrow 1$. En cambio, si el modelo realiza una predicción con una confianza alta, $y_{pred}^{(i)} \longrightarrow 1$, luego $(1-y_{pred}^{(i)})^{\gamma} \longrightarrow 0$ y este ejemplo, por estar bien predicho, ponderará poco. En el trabajo original en el que se define esta función se recomienda utilizar $\gamma = 2$, ya que experimentalmente da muy

buenos resultados [22]. El parámetro α se usa para equilibrar la función y se puede configurar o bien como hiperparámetro[2] o bien como la inversa de la frecuencia de cada clase. Nótese que la función *categorical cross-entropy* es un caso particular de la función focal para $\gamma = 0$ y $\alpha = 1$.

Optimizador

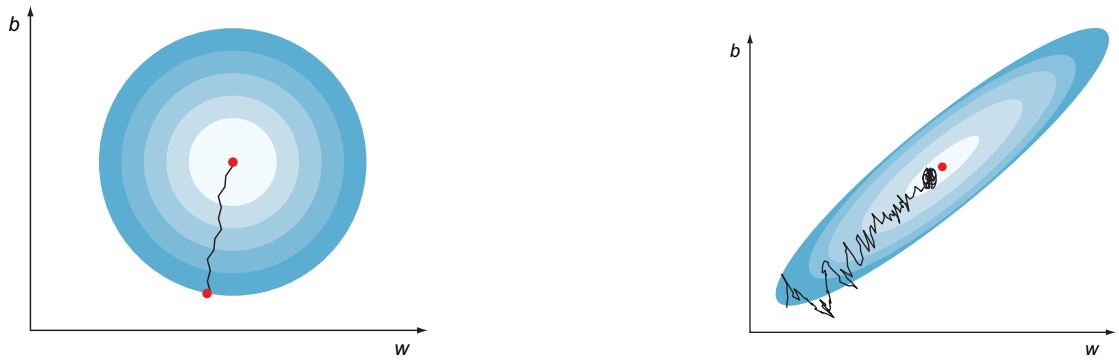
Durante el proceso de optimización, la red neuronal realiza un determinado número de iteraciones sobre el mismo conjunto de datos, a las que llamamos épocas. Los pesos se inicializan con valores aleatorios para cada neurona y, al final de cada época, se calcula la función de pérdida y se actualizan los pesos de manera que el error disminuya en la siguiente iteración. El proceso iterativo acaba cuando se minimiza la función de pérdida, aunque, para evitar problemas de convergencia, siempre se selecciona un número máximo de épocas.

El optimizador es el algoritmo que determina de qué manera debemos actualizar los pesos en cada iteración y, en el caso de redes neuronales, se suelen usar algoritmos que se basan en la técnica del gradiente descendiente. Sea f la función de pérdida, y w un peso, el gradiente $\nabla_w f(w_0)$ representa la curvatura de la función de pérdida con respecto al peso en w_0. Los algoritmos de gradiente descendiente se basan mover los pesos en la dirección opuesta al gradiente para reducir el valor de la función. Sea $w_{i+1} = w_i - \alpha \nabla_w f(w_i)$, tenemos que $f(w_{i+1}) < f(w_i)$, $\forall \alpha \in \mathbb{R}^+, i \in \mathbb{N}$. Para utilizar este método debemos por lo tanto requerir que las funciones de pérdida que se usen en la red sean funciones diferenciables. Al parámetro α, que indica el paso que debemos tomar para actualizar los pesos, se le llama ratio de aprendizaje o *learning rate*. Se trata de un parámetro que podemos configurar en nuestro modelo, teniendo en cuenta que un valor muy pequeño supondrá un mayor coste computacional ya que aumentará el tiempo de computación para alcanzar una solución óptima, pero un valor muy grande puede hacer que tengamos problemas de convergencia.

El problema que tiene este algoritmo es que calcula el error en cada punto del conjunto de entrenamiento, lo que computacionalmente resulta muy costoso además de requerir mucha memoria. El descenso de gradiente estocástico (*stochastic gradient descent*) soluciona este problema al actualizar los pesos tras calcular el gradiente en un único punto. Se basa en la idea de que, si el conjunto de datos es redundante, el gradiente en un punto será muy similar al gradiente en otro punto, por lo que no hace falta calcularlo en todo el espacio, reduciendo considerablemente el coste computacional. En la imagen 2.5 se muestra la comparación de ambos algoritmos en la superficie que definen dos de sus pesos. Como podemos observar, para el caso del gradiente descendiente se llega al mínimo prácticamente en una línea recta, mientras que en el gradiente estocástico vemos un comportamiento con forma de zigzag, como resultado de que en cada iteración se ajusta a un punto determinado del espacio. No se garantiza que en cada paso vaya a estar más cerca del mínimo, pero generalmente se consigue disminuir mucho la función de pérdida, que por lo general suele ser suficiente para la mayoría de problemas [5, 14].

En la práctica se usa un algoritmo que está en un punto medio entre el gradiente descendiente y el estocástico, llamado gradiente descendiente con *mini-batch*. Este algoritmo sigue el mismo esquema que el estocástico, pero en vez de calcular el gradiente en un punto y actualizar los pesos, toma un parámetro, llamado tamaño de lote (*mini-batch*), que indica el número de puntos en los que calcular el gradiente antes de actualizar los pesos. Dado que los conjuntos de datos

[2]Los hiperparámetros son las variables de configuración que son externas al modelo y cuyos valores no pueden ser entrenados, sino que son especificados por el programador.



(a) Gradiente descendiente

(b) Gradiente descendiente estocástico

Figura 2.5: Comparación de la evolución de la función de pérdida a lo largo de las épocas en el plano que definen dos de los pesos [15].

suelen ser considerablemente grandes, este algoritmo resulta muy útil, ya que mejora la eficiencia computacional y consigue una convergencia mucho más rápida [14]. El hiperparámetro que define el tamaño de lote se suele definir a partir de las restricciones de memoria que podamos tener.

En este trabajo hemos usado dos algoritmos optimizadores, *Adam* y *RMSProp*, ya que son dos de los más usados en modelos de redes neuronales, aunque hay muchos más. *RMSProp* o *Root Mean Squared Propagation* se basa en el descenso de gradiente y está diseñado para acelerar el proceso de optimización mediante la implementación de una tasa de aprendizaje diferente para cada parámetro. El algoritmo *Adam* combina el método del gradiente descendiente estocástico y *RMSProp*, tomando del primero el método de actualizar los pesos según el tamaño del lote del entrenamiento y del *RMSProp* la tasa de aprendizaje variable por parámetro. El resultado es una mejora del rendimiento del modelo que hace que actualmente sea el optimizador más usado en modelos de redes neuronales por su rapidez de convergencia [23].

Actualización de pesos

Como acabamos de ver, el proceso de entrenamiento de una red neuronal consta de tres pasos principales:

1. A partir de las sumas ponderadas que definen las neuronas y la función de activación para extraer relaciones no lineales, el algoritmo aprende a identificar los *features* del conjunto de datos de entrada. En este punto, la red procesa la entrada en la primera capa y va transmitiendo los pesos 'hacia adelante' (*forward pass*), activando las neuronas de capas superiores hasta llegar a la última capa.

2. Se realizan predicciones y se calcula el valor de la función de pérdida.

3. Mediante un algoritmo basado en el gradiente descendente, se calcula el valor de los pesos que minimizan la función de pérdida.

Los nuevos pesos se calculan según la expresión $w_{i+1} = w_i - \alpha \nabla_w f(w_i)$ donde f es la función

de pérdida, α la tasa de aprendizaje y w_i, w_{i+1} los pesos actual y actualizado. Para actualizar cada peso de la red neuronal, se usa el algoritmo de retropropagación o *backpropagation*, que consiste en pasar la información de la última capa, donde se ha realizado la predicción, hacia atrás, propagándola a todas las neuronas de la red hasta llegar a la primera capa (conjunto de entrada). La información que se propaga es precisamente $\frac{\partial f}{\partial w_i}$, de manera que cada derivada parcial indica cuánto contribuye el peso asociado al error global. Supongamos que tenemos n neuronas encadenadas, de manera que la que recibe la entrada es la neurona 1 y la final es la n, $w_{i,i+1}$ es el peso que define la neurona i con el que alimenta a la neurona $i+1$ y f es la función de pérdida. Aplicando entonces la regla de la cadena para calcular las derivadas parciales a lo largo de la red, se define el gradiente en la primera capa de acuerdo con la siguiente expresión:

$$\frac{\partial f}{\partial w_{1,2}} = \frac{\partial f}{\partial w_{n-1,n}} \cdot \frac{\partial w_{n-1,n}}{\partial w_{n-2,n-1}} \cdots \frac{\partial w_{2,3}}{\partial w_{1,2}}$$

Una vez actualizados todos los pesos, se vuelve al paso 1 y la nueva predicción que realice el modelo con estos nuevos valores dará una función de pérdida menor que la anterior.

En el caso de las CNN, los pesos que deben entrenarse son los siguientes. En las capas convolucionales los parámetros son los valores de los *kernels*, luego el número de pesos es el número de entradas de la matriz más un peso de sesgo. Las capas de agrupamiento no tienen parámetros y las capas completamente conectadas deben actualizar los valores de W y b según la expresión que veíamos en la definición 7. Como hemos comentado antes, al haber muchas menos conexiones entre neuronas en una red convolucional en comparación con una tradicional, el número de pesos que deben entrenarse también es significativamente menor.

2.3. Validación cruzada K-fold

Una vez que el modelo está entrenado para todo el conjunto de datos de entrada, podríamos evaluar su rendimiento sobre el mismo set de datos sobre el que ha sido entrenado, pero este resultado estaría sesgado ya que el modelo ha aprendido a distinguir patrones en el mismo conjunto. Lo que nos interesa es saber cómo de bien funciona un modelo en datos de entrada nuevos, es decir, datos que no ha visto en el proceso de entrenamiento.

En cualquier modelo de *deep learning* se suele dividir el conjunto de datos en 3 subconjuntos disjuntos:

- **Entrenamiento.** Conjunto sobre el que se entrena el modelo.
- **Validación.** Se usa para evaluar el rendimiento del modelo en cada época sobre un conjunto de datos que no ha visto. Permite configurar los hiperparámetros del modelo durante el entrenamiento.
- **Test.** Conjunto de datos que el modelo no ha visto durante el entrenamiento y que usamos para evaluar su rendimiento.

Queremos que los tres sets de datos sean representativos del conjunto total, es decir, queremos

evitar que en el test de entrenamiento tengamos, por ejemplo, todas las imágenes de las formas I, II, III y IV y que en el test sólo tengamos V y VI. Por este motivo la forma más eficaz de definir los subconjuntos es hacerlo de forma aleatoria. Nótese además que es imprescindible que los conjuntos anteriores sean disjuntos para evitar redundancia.

Más aún, en el caso de las redes neuronales, los resultados de un modelo pueden variar dependiendo de la estocástica del algoritmo y del conjunto de datos de entrada escogido, por lo que suele ser recomendable ejecutar el modelo varias veces y comparar el rendimiento en cada caso. La técnica más usada para evaluar un modelo como el nuestro es la validación cruzada *K-fold*. Consiste en dividir el conjunto de entrenamiento en K subconjuntos, de manera que cada vez que entrenamos el modelo cogemos $K-1$ subconjuntos de manera aleatoria para el set de entrenamiento y el subconjunto restante se usa como set de validación. Se repite este proceso hasta que cada uno de los subconjuntos haya servido de validación en uno de los modelos, como se puede observar en la imagen 2.6 para 5 subconjuntos. Como es lógico, el coste computacional incrementa ya que tenemos que entrenar el mismo modelo 5 veces, pero el resultado que obtenemos es más fiable ya que el rendimiento total del modelo será el promedio de todas las ejecuciones. Otra ventaja es que nos permite dar un intervalo de error a partir de la desviación típica.

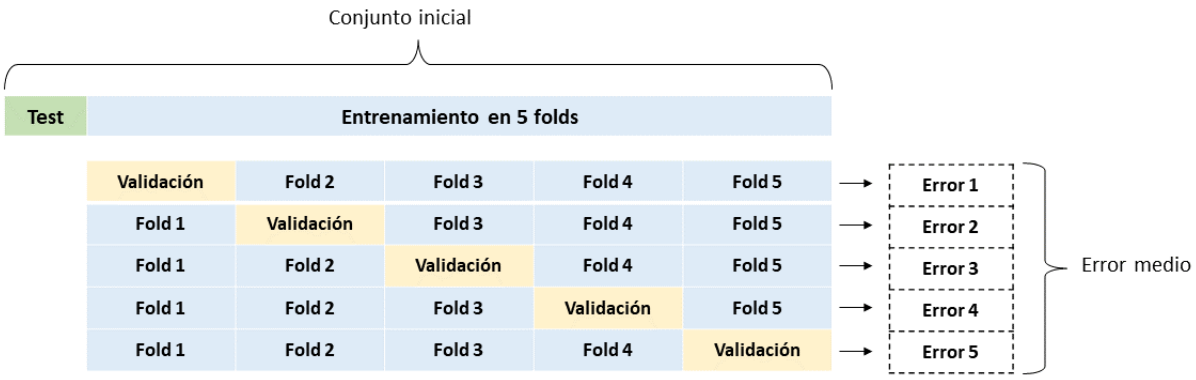


Figura 2.6: Reparto de los conjuntos de entrenamiento, test y validación para un *K-fold* con 5 subconjuntos. El set de test se usa para evaluar el modelo, mientras que el de entrenamiento se divide en 5 subconjuntos de manera que en cada partición se usa un set de validación distinto.

2.4. Generalización, overfitting y underfitting

Se define como generalización a la habilidad del modelo para producir buenos resultados en conjuntos de datos que no ha visto. Cuando el modelo no obtiene buenos resultados en el set de entrenamiento se dice que tiene *underfitting*, es decir, el modelo no está optimizado. En esta situación también podemos decir que tiene poca capacidad de generalización, ya que todavía quedan *features* por aprender. En cambio, cuando obtenemos buenos resultados en el set de entrenamiento, pero no en el set de validación, el modelo tiene *overfitting* o poca capacidad de generalización. Esto significa que el modelo está demasiado optimizado para el conjunto de entrenamiento, es decir, ha aprendido a identificar patrones en este set que no se observan en el set de validación.

Resulta importante por lo tanto encontrar un equilibrio entre generalización y optimización. Ne-

cesitamos una red lo suficientemente grande como para que aprenda todos los patrones necesarios para realizar las predicciones. A su vez, una red demasiado grande puede aprender a distinguir patrones que no son lo suficientemente relevantes, por lo que reduciendo el tamaño forzamos a que aprenda los más representativos. Para encontrar este punto medio entre *overfitting* y *underfitting*, generalmente se empieza por desarrollar un modelo que esté sobreentrenado, de manera que garantizamos que es capaz de identificar patrones en las imágenes de entrenamiento y después podremos ajustarlo usando técnicas de reducción de *overfitting*, también llamadas técnicas de regularización. Algunas de las más usadas son las siguientes:

- Generar más imágenes. La idea es que, si el modelo estuviese expuesto a un número infinito de imágenes de entrenamiento, sería capaz de aprender cualquier patrón existente por lo que nunca tendría *overfitting*. El problema de esta técnica es que, si las imágenes resultan redundantes para el algoritmo, los resultados no van a mejorar aunque sigamos añadiendo datos al conjunto de entrada.

- Reducir el tamaño del modelo. Si disminuimos el número de parámetros entrenables (número de capas o de neuronas), disminuimos la capacidad de aprendizaje del modelo.

- Probar diferentes arquitecturas. Añadir o eliminar capas o probar con otro optimizador.

- Añadir capas de *dropout*. Consiste en añadir capas que de manera aleatoria desactivan neuronas, lo que obliga a las neuronas cercanas a no depender tanto de las neuronas desactivadas [24]. Estas capas se configuran de manera que toman como variable el porcentaje de neuronas que se quieren excluir de la red.

- Regularización de pesos L1 o L2. Se trata de técnicas que restringen los valores que toman los pesos, con lo que se limita la complejidad de la red. En particular, lo que se hace es añadir a la función de pérdida un coste asociado a tener pesos grandes. Sea λ una función de pérdida como la de la definición 14, podemos modificar esta función de las siguientes maneras:

  Regularización L1. $\lambda_{L1} = \lambda + \beta \sum_{i=1}^{N} |w_i|$ para $\beta > 0$.

  Regularización L2. $\lambda_{L2} = \lambda + \beta \sum_{i=1}^{N} (w_i)^2$ para $\beta > 0$.

2.5. Aprendizaje Transferido o Transfer Learning

La ventaja de las redes neuronales convolucionales, como ya hemos visto, es su capacidad para aprender a distinguir patrones aislados sin importar en qué parte de la imagen estén situados. Esta característica hace que una CNN ya entrenada pueda ser fácilmente reutilizable para otro problema. La idea del aprendizaje transferido es aprovechar el aprendizaje de modelos ya entrenados para aplicarlo a una nueva tarea.

Para problemas de visión artificial hay muchas arquitecturas open source que podemos usar para entrenar nuestros propios modelos [15, 25]. En general, se suelen usar modelos que han sido

entrenados para tareas de clasificación a gran escala donde el conjunto de datos inicial es lo suficientemente grande y general, de manera que el modelo aprende a identificar representaciones visuales genéricas. De esta manera, los patrones aprendidos pueden aplicarse a problemas más específicos incluso si las clases de estos nuevos problemas son completamente distintas a la original. El entrenamiento sobre el nuevo conjunto se centra en aprender los *features* más específicos del nuevo set, lo que significa que no necesitamos entrenar el modelo de cero, sino que basta con realizar pequeños cambios sobre el modelo base. En un caso como el nuestro en el que el conjunto de datos es pequeño y las imágenes son muy distintas al conjunto original con el que ha sido entrenado el conjunto base, lo que se hace es 'congelar' las capas iniciales para que no se modifiquen sus pesos durante el entrenamiento, y sólo permitimos que se actualicen los de las capas finales. Dado que el nuevo conjunto no tiene mucha similitud con el original, es importante entrenar y particularizar las capas superiores del modelo de acuerdo con la tarea que queremos resolver, lo que llamamos ajuste fino.

La técnica del aprendizaje transferido suele ser recomendable incluso cuando el set de datos es grande. Cuando entrenamos un modelo desde cero, los pesos se inician aleatoriamente y el modelo tiene que pasar por todo el proceso de entrenamiento. Lo que conseguimos con esta técnica es una disminuir considerablemente el tiempo de computación, ya que el número de parámetros entrenables se reduce significativamente y la red converge más rápidamente.

En nuestro problema, las principales diferencias entre manchas de una clase y de otra son formas geométricas. Por ejemplo, la forma I tiene formas alargadas y curvas mientras que las formas V y VI son más redondeadas. A su vez, la V se diferencia de la VI en que sus bordes son más irregulares y se asemejan a pétalos. Este tipo de contornos se aprenden a clasificar en las primeras capas de la red, por lo que cualquier modelo pre-entrenado ya sabe detectar estos *features* y nos basta con entrenar el clasificador para que identifique a qué clase corresponde cada tipo.

2.6. Etiquetado

Podemos clasificar los algoritmos de *deep learning* en dos grupos: algoritmos de aprendizaje supervisado y no supervisado. Los primeros algoritmos son aquellos en los que el conjunto de entrenamiento debe estar etiquetado para que el algoritmo aprenda a identificar qué patrones pertenecen a cada categoría. Por otra parte, el aprendizaje no supervisado es aquel en el que los datos de entrada no están etiquetados, por lo que el modelo aprende a detectar patrones y relaciones existentes en los datos, pero sin tener la información de cuál es la clase asociada. Estos algoritmos se suelen usar en problemas de agrupación. Los dos modelos que se han desarrollado en este trabajo entran dentro del grupo del aprendizaje supervisado, y hemos etiquetado las imágenes de dos maneras distintas.

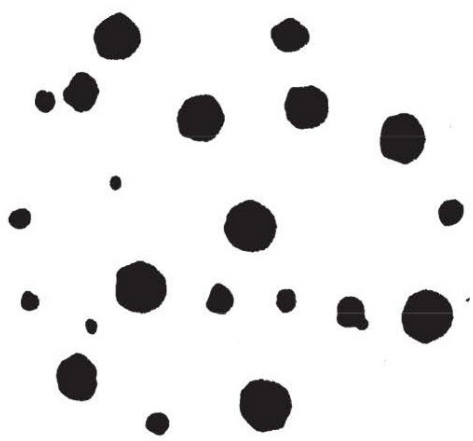
Para el primer modelo, puesto que cada imagen pertenece a una categoría, los nombres de las carpetas en las que se almacenan las imágenes de cada clase sirven como etiquetas para el modelo. El segundo modelo, puesto que mezclamos manchas de distintos tipos en una misma imagen, consiste en clasificar cada mancha de manera individual, para lo que necesitamos etiquetar cada una de ellas.

Segmentación Semántica

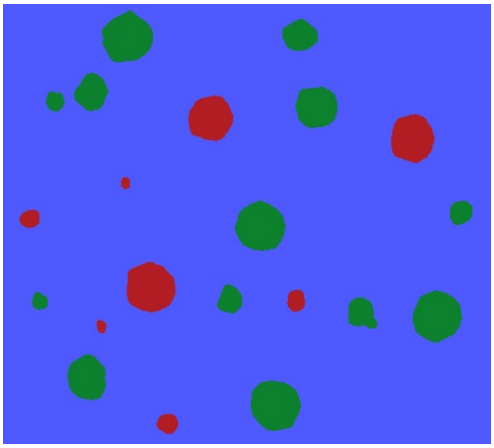
Para una red como las que hemos descrito, no es trivial identificar lo que es una mancha, es decir, puede detectar los bordes, pero no sabría identificar si en el caso de superposición de manchas tendríamos una o más. Si solamente queremos saber si existen manchas de x tipo en la imagen, podríamos resolver el problema como en el primer modelo, pero esto no sería efectivo si queremos saber cuántas manchas de cada tipo hay en la imagen. Por este motivo, hemos recurrido a la segmentación semántica, que consiste en asignar a cada píxel una categoría, de manera que el algoritmo se entrena para realizar predicciones para cada píxel.

En un problema de este tipo, las imágenes se etiquetan mediante máscaras, que son copias de la imagen original con el mismo tamaño y número de píxeles, de manera que cada píxel está coloreado según la categoría a la que pertenece. Las máscaras se transforman a su vez en etiquetas semánticas, en las que cada píxel tiene asignada una etiqueta según la clase a la que pertenece, y cada clase se representa con un número entero. El algoritmo recibe como input la imagen original en blanco y negro junto con su correspondiente etiqueta semántica y la salida es una nueva etiqueta que contiene la información de las predicciones que ha realizado el modelo para cada píxel. En la imagen 2.7 se muestra la diferencia entre máscara y etiqueta semántica para una de las imágenes de nuestro conjunto de datos. Nótese que por simplicidad la etiqueta representa un menor número de píxeles, pero en realidad debe tener la misma resolución que la imagen original.

La arquitectura de una red convolucional aplicada a este tipo de problemas es similar a la de cualquier red de este tipo con la diferencia de que la salida será una imagen codificada según la categoría que se haya predicho para cada píxel de la imagen. Como hemos visto anteriormente, en una red convolucional se disminuye progresivamente la resolución mediante capas de *pooling* para reducir el coste computacional, de manera que perdemos la información espacial. En una tarea de clasificación de imágenes sólo nos importan los elementos y patrones que tiene la imagen, independientemente de dónde están situados, por lo que no importa que la resolución de la imagen disminuya. En una tarea de segmentación semántica necesitamos mantener la misma resolución para poder identificar la categoría a la que pertenece cada píxel, y esto se consigue mediante convoluciones transpuestas y una arquitectura de red dividida en dos partes. La primera parte, llamada contracción, tiene el mismo tipo de arquitectura que encontraríamos en cualquier red neuronal convolucional. En esta parte es donde se reduce la resolución espacial y se generan los *feature maps*. La segunda parte de la arquitectura, llamada expansión, usa capas con convoluciones traspuestas para recuperar la información espacial de la imagen original. Como hemos explicado, una capa convolucional realiza el producto elemento a elemento de dos matrices, generando un único valor para cada píxel. Una convolución de trasposición hace justo lo contrario, es decir, toma cada valor del *feature map* generado con una baja resolución y se multiplica cada peso del *kernel* por dicho valor (imagen 2.8), generando un nuevo *feature map* con el resultado de estas multiplicaciones [26, 27]. Los *kernels* de estas capas pasan a ser también parámetros entrenables del modelo.



(a) Imagen de entrada



(b) Máscara

(c) Etiqueta semántica

Figura 2.7: Dada la imagen de entrada (a) en blanco y negro, se muestra su máscara en (b) y la correspondiente etiqueta semántica en (c). Observamos que la imagen tiene 3 clases: el fondo en azul (clase 0), un tipo de mancha en verde (clase 1) y otro tipo en rojo (clase 2).

2.7. Métricas de validación

Una parte importante del desarrollo de un modelo es entender cómo de bien predice cada una de las clases. Para ello, como ya mencionamos anteriormente, usamos un conjunto de datos que el modelo no ha visto durante el proceso de entrenamiento. Dependiendo de cómo se comporte el modelo al ver estos nuevos datos podemos extraer conclusiones sobre su capacidad de generalización.

Las métricas también nos sirven para comparar modelos entre sí, y su elección depende del problema que estemos resolviendo. Comenzamos definiendo los siguientes términos:

- (TP) Positivos verdaderos. La imagen corresponde a la categoría X, y es correctamente clasificada.

- (FP) Positivos falsos. La imagen se clasifica dentro de la categoría X pero pertenece a otra

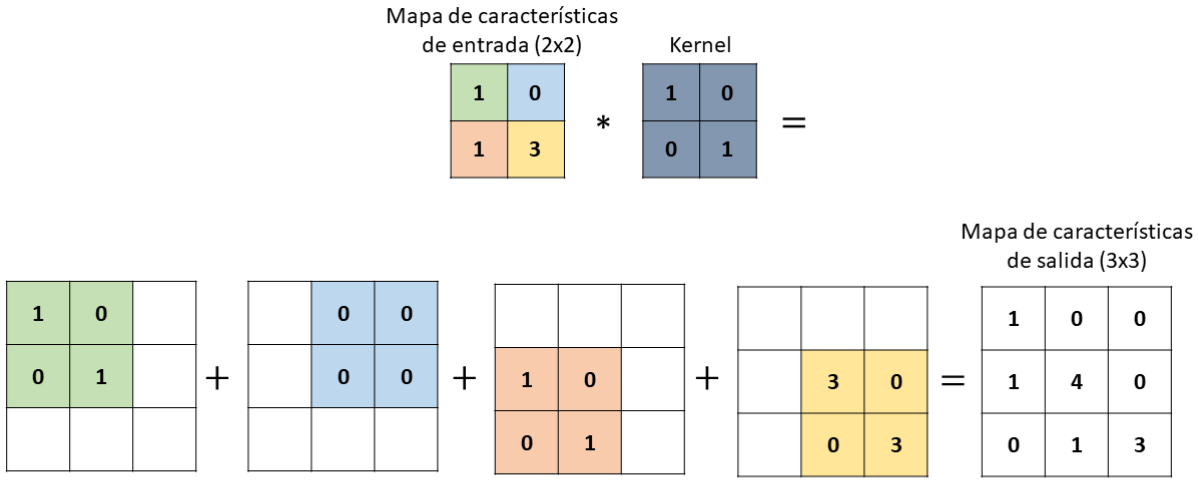


Figura 2.8: Representación gráfica de una capa de convolución traspuesta aplicada a una imagen. A partir de una matriz de entrada de tamaño 2x2 y un kernel de tamaño 2x2, el resultado es una matriz de *features* de tamaño 3x3 cuyos coeficientes se calculan aplicando el Producto de Hadamard.

categoría.

- (TN) Negativos verdaderos. La imagen no pertenece a la categoría X y tampoco se ha clasificado dentro de dicha categoría.

- (FN) Negativos falsos. La imagen pertenece a la categoría X pero se ha clasificado dentro de otra categoría.

Nótese que $TP+FP+TN+FN = \#$ imágenes evaluadas. Para resolver un problema de segmentación semántica donde clasificamos píxeles también usamos estos mismos términos sustituyendo 'imágenes clasificadas' por 'píxeles clasificados'.

A partir de estos cuatro términos se definen las métricas de validación que se han usado a lo largo de este trabajo.

Accuracy o exactitud

Es la métrica de validación más usada para problemas de clasificación y, multiplicada por 100, da el porcentaje de imágenes correctamente clasificadas respecto del total de imágenes a clasificar:

$$\frac{TP + TN}{TP + FP + TN + FN} \tag{2.1}$$

Precisión y sensibilidad

La precisión es el número de positivos verdaderos entre el número total de positivos que se han predicho:

$$\frac{TP}{TP+FP} \tag{2.2}$$

es decir, responde a la pregunta 'Cuando predecimos que es verdadero, ¿con qué frecuencia es correcto?". La sensibilidad, en cambio, responde a la pregunta 'Cuando es verdadero, ¿con qué frecuencia lo predice de manera correcta?' y se define de la siguiente manera:

$$\frac{TP}{TP+FN} \tag{2.3}$$

La precisión es una métrica útil cuando queremos minimizar los falsos positivos, mientras que la sensibilidad es útil para penalizar los falsos negativos [28]. Un ejemplo sencillo es el siguiente. Si realizamos un estudio sobre un conjunto de pacientes para ver si tienen una determinada enfermedad, nuestro objetivo principal sería reducir el número de falsos negativos, es decir, el número de personas que tienen la enfermedad pero que hemos predicho que no la tienen. En cambio, predecir que un paciente presenta una enfermedad que en realidad no tiene sería menos problemático. Por lo tanto, en un problema como éste donde queremos predecir el mayor número de positivos posibles, la sensibilidad sería la métrica más adecuada. En nuestro problema no tenemos preferencia sobre falsos negativos o positivos, ya que queremos penalizar de la misma manera ambos. Por este motivo, resulta de utilidad la métrica valor-F ya que nos permite combinar ambos resultados.

Valor-F

$$F_\beta = (1+\beta^2)\cdot\frac{\text{precisión}\cdot\text{sensibilidad}}{(\beta^2\cdot\text{precisión})+\text{sensibilidad}} = \frac{(1+\beta^2)\cdot TP}{(1+\beta^2)\cdot TP+\beta^2 FN+FP} \tag{2.4}$$

Si $\beta > 1$ le damos más importancia a los falsos negativos y si $\beta < 1$ a los falsos positivos. En este trabajo usamos el valor $\beta = 1$, de manera que le damos la misma importancia a ambos [29-31].

Intersección sobre unión (IoU)

Una métrica similar a la anterior es la intersección sobre unión. Esta métrica cuantifica cómo se solapan las regiones de verdad y predicción, entendiendo por región de verdad las etiquetas reales y región de predicción las etiquetas que asigna el algoritmo al hacer su predicción. Lo que buscamos con esta métrica es penalizar las clasificaciones incorrectas (FP y FN). Esta métrica se usa a menudo en problemas de segmentación semántica.

$$\frac{TP}{TP+FP+FN} \tag{2.5}$$

Para la primera parte, puesto que las clases están balanceadas, usaremos la exactitud para evaluar el modelo. En la segunda parte sí tenemos un problema con el desbalanceo de las categorías, por lo que el valor-F con $\beta = 1$ es una métrica más acertada en este caso.

Capítulo 3

Conjunto de datos

En un ordenador, cualquier imagen digital en 2D se representa mediante un mapa de bits. Esto significa que cada imagen de tamaño (n x m) se lee como si fuese una matriz con n filas y m columnas. Siguiendo la notación presentada en el capítulo anterior, si la imagen es de tamaño n x m, significa que tiene n x m píxeles y cada elemento de la matriz $a_{i,j}$ $\forall 1 \leq i \leq n, 1 \leq j \leq m$ es un píxel.

Si la imagen es en blanco y negro, cada píxel es un escalar. En cambio, si la imagen es en color, cada píxel es un vector de tres coordenadas, donde la primera coordenada representa la intensidad del color rojo, la segunda la del color verde y la tercera la del color azul. A este tipo de representación se le llama RGB (red-green-blue), y la idea es que las diferentes combinaciones de estos tres colores pueden producir todas las diferentes tonalidades que detecta el ojo humano. Para representar la intensidad de cada color se usa la escala de 0 a 2^N bits, llamada representación de N-bits. Comúnmente se emplea la escala de 8-bits, lo que significa que la escala es $[0, 255]$. De esta manera, una imagen en escala de grises tiene cada píxel asociado a un valor entre 0 (negro) y 255 (blanco), mientras que una imagen en color tendrá cada píxel de la forma (a, b, c) con $a, b, c \in [0, 255]$. Por ejemplo, el color negro en una imagen en color se representa como $[0, 0, 0]$, el blanco como $[255, 255, 255]$, el rojo como $[255, 0, 0]$ y así sucesivamente. Según esto, cada píxel de una imagen RGB puede tener 256^3 valores diferentes y decimos que tiene 3 canales, ya que la información se guarda en tensores de tamaño $(n, m, 3)$ donde n y m representan el número de píxeles a lo alto y a lo ancho.

3.1. Aumento de datos

Para cualquier modelo de *deep learning* es imprescindible que el algoritmo procese un número lo suficientemente alto de imágenes como para poder extraer los *features* más relevantes de cada una de ellas. Puesto que nuestro conjunto de partida no es más que las 6 muestras que observamos en la imagen 1.2, es decir, una sola imagen por cada categoría, necesitamos generar un set de imágenes sintéticas para cada tipo. Las técnicas que se usan en este tipo de modelos para aumentar de manera artificial el tamaño del conjunto de datos mediante transformaciones sobre el conjunto inicial se llama aumento de datos o *data augmentation*.

Para este trabajo, el proceso de generación de imágenes sintéticas ha consistido en extraer cada una de las manchas de las imágenes anteriores, aplicar una serie de transformaciones, y pegar estas manchas en una nueva imagen. Además, se ha generado el mismo número de imágenes para cada una de las clases, evitando así caer en el frecuente problema del desbalanceo de datos. Como hemos adelantado en el capítulo anterior, el desbalanceo ocurre cuando tenemos más imágenes en una categoría que en otras, lo cual puede provocar que el modelo esté sesgado y de predicciones pobres para las clases minoritarias.

3.1.1. Extracción de defectos

En primer lugar, se ha extraído cada una de las manchas de cada imagen. En este paso es importante eliminar las manchas de los bordes, ya que su forma curva es una característica muy representativa y la red podría enfocarse en aprender a detectarlo. Esta forma se debe al ocular del microscopio por lo que no nos interesa que el algoritmo se centre en aprenderlo. En la imagen 3.1a, señalados en azul, se pueden ver algunos ejemplos de defectos que hemos eliminado de nuestro conjunto por estar en los bordes de la imagen.

El set de manchas extraídas consta de 42 manchas de la forma I, 22 de la forma II, 52 de la forma III, 27 de la forma IV, 19 de la forma V y 21 de la forma VI.

3.1.2. Transformaciones

A continuación, partiendo de las manchas originales, aplicamos una serie de transformaciones a cada una de ellas para aumentar el tamaño del conjunto. Las transformaciones que se han aplicado son las detalladas en la tabla 3.1.

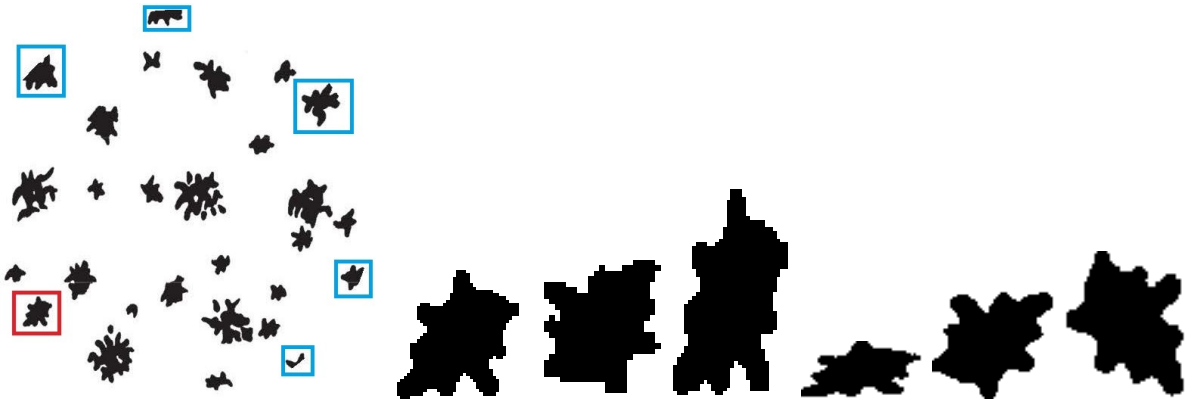
Transformación	**Rango**
Ancho de la imagen	[0.5, 1.5]
Alto de la imagen	[0.5, 1.5]
Inversión sobre el eje horizontal	True/False
Inversión sobre el eje vertical	True/False
Rotación	(0, 2π]
Inclinación	[0, 0.4]

Tabla 3.1: Transformaciones aplicadas a los defectos para la generación de imágenes sintéticas. Cada transformación se aplica mediante un coeficiente que toma un valor aleatorio dentro del rango especificado en cada caso.

Dichas transformaciones y sus rangos de aplicación han sido elegidos tras realizar una serie de pruebas sobre las imágenes. El objetivo es ser capaces de aumentar el rendimiento y la capacidad de generalización del modelo con imágenes que tengan características creíbles. Por ejemplo, no estamos considerando coeficientes de inclinación mayores que 0.4 ya que se observó que valores

muy altos distorsionan demasiado la imagen. Estas transformaciones se aplicaron al set de manchas estableciendo de manera aleatoria el valor del coeficiente aplicado dentro de cada rango. Al final de este paso hemos aumentado x10 el conjunto de manchas original.

Un ejemplo de cómo se ve la mancha transformada se muestra en la imagen 3.1. La imagen de la izquierda muestra la imagen representativa de la categoría IV tal cual aparece en la norma y en rojo la mancha que tomamos como ejemplo. En (b) vemos esta misma mancha extraída de la norma sin aplicar ninguna transformación más que la de binarización para eliminar los píxeles en escala de grises. El resto de las manchas son el resultado de aplicar las transformaciones anteriores de manera aleatoria.



(a) Norma. Categoría IV (b) Mancha original (c) (d) (e) (f) (g)

Figura 3.1: Ejemplo de las transformaciones aplicadas a una mancha perteneciente a la categoría IV. En (a) se muestra la imagen de referencia que aparece en la norma para la categoría IV y en rojo señalada la mancha que queremos transformar. En azul se muestran las manchas de los bordes que hemos eliminado. En (b) muestra la mancha original y en (c)-(g) las manchas transformadas.

3.1.3. Generación de imágenes sintéticas

Antes de pegar las manchas para generar las imágenes, es importante realizar un paso previo de binarización. Las imágenes originales están en escala de grises, pero aquí necesitamos que todos los píxeles de la mancha sean negros y el fondo transparente. Para esto, basta con seleccionar un umbral de manera que todos los píxeles en escala de grises pasan a ser negros y los píxeles blancos transparentes. Esta técnica se le llama *thresholding* y la idea es que recorre cada píxel $(a_{i,j}, b_{i,j}, c_{i,j})$ de la imagen y lo compara con el valor asignado del umbral $(\epsilon_a, \epsilon_b, \epsilon_c)$[3]:

$$\begin{cases} \text{si } a_{i,j} > \epsilon_a \text{ y } b_{i,j} > \epsilon_b \text{ y } c_{i,j} > \epsilon_c \Longrightarrow (a_{i,j}, b_{i,j}, c_{i,j}) = (255, 255, 255, 0) \text{ transparente} \\ \text{en otro caso} \Longrightarrow (a_{i,j}, b_{i,j}, c_{i,j}) = (0, 0, 0) \text{ negro} \end{cases}$$

Por último, dada una imagen en blanco de (600×600) píxeles, se ha elegido que cada imagen tenga un número aleatorio de manchas entre 10 y 60. Valorando cómo se ven las muestras reales

[3]Nótese que un píxel transparente es un vector 4-dimensional.

de fundiciones, se ha permitido la superposición de manchas, por lo que se ha empleado en este punto una función tal que, dado un número fijo de puntos que pasamos como parámetro, los distribuye por una cuadrícula manteniendo una distancia mínima determinada entre ellos. De esta manera, podemos controlar la superposición evitando que varias manchas caigan en las mismas coordenadas.

3.1.4. Conjuntos de entrenamiento, validación y test

El conjunto de entrada para la primera parte del trabajo son imágenes con todas las manchas de un mismo tipo y las creamos seleccionando manchas de manera aleatoria a partir del conjunto de las manchas transformadas. Un ejemplo de imagen sintética generada para cada categoría se puede ver en la imagen 3.2. Para la segunda parte del trabajo, se ha generado un set de imágenes que combina manchas de dos formas consecutivas (I y II, II y III, etc.) y toma como valor aleatorio el porcentaje de manchas de cada tipo, como se puede ver en la imagen 3.3. Nótese que las imágenes en color son las máscaras, pero la entrada del modelo es en blanco y negro.

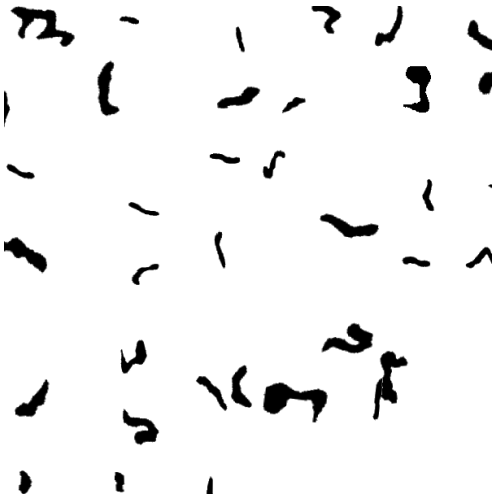
En la tabla 3.2 se detalla el tamaño de los sets de entrenamiento, validación y test para cada uno de los problemas. En ambos casos hemos tenido que reducir las dimensiones de las imágenes sintéticas para reducir el coste computacional. Esta reducción se aplica sobre toda la imagen, por lo que, al guardar la información en menos píxeles, las manchas pueden verse ligeramente distorsionadas. Esto no supone un problema porque las manchas de cada tipo siguen siendo reconocibles, pero si no estuviésemos limitados por los recursos computacionales sería conveniente mantener el tamaño original para lograr mejores resultados.

	Entrenamiento	Validación	Test	Total	Dimensión
Modelo 1	1800 (55 %)	300 (9 %)	1200 (36 %)	3300	(224 x 224)
Modelo 2	1250 (74 %)	200 (12 %)	250 (15 %)	1700	(128 x 128)

Tabla 3.2: Tamaños de los sets finales de entrenamiento, validación y test para cada una de las partes. Se muestra el número de imágenes de cada set junto con el porcentaje respecto del número total de imágenes. La última columna indica las dimensiones de las imágenes en cada caso.

(a) I

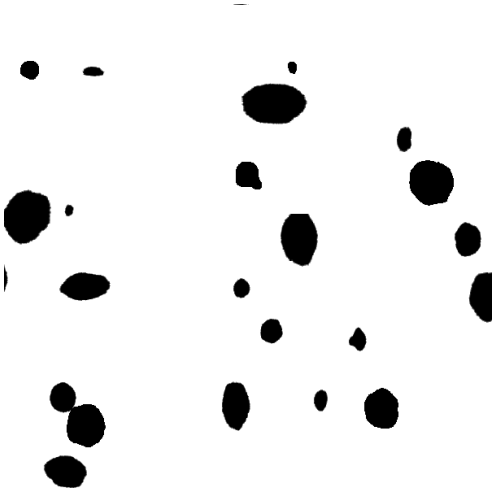
(b) II



(c) III

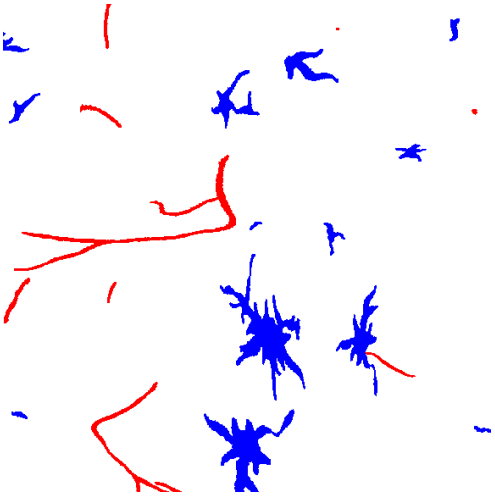
(d) IV

(e) V

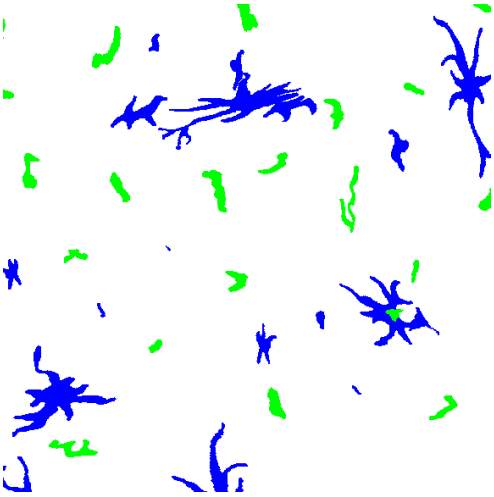


(f) VI

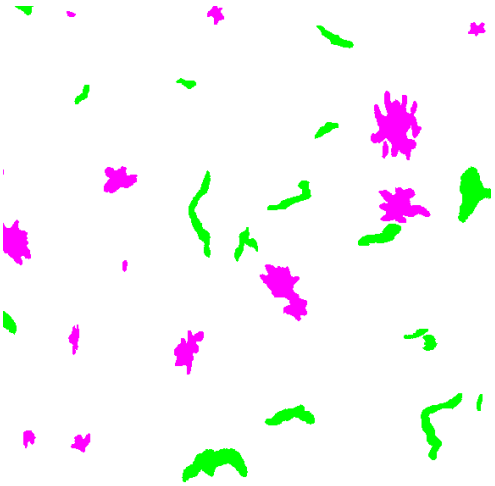
Figura 3.2: Imágenes sintéticas generadas con manchas pertenecientes a una única categoría.



(a) I y II

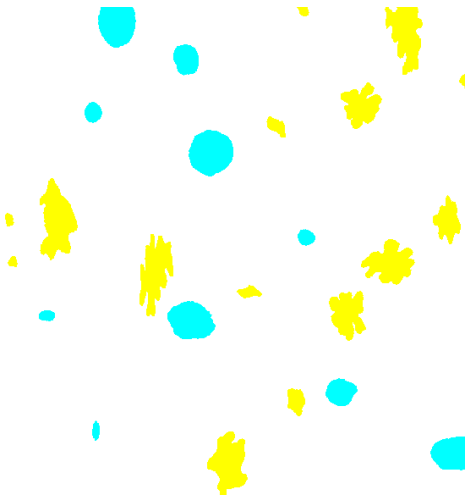


(b) II y III



(c) III y IV

(d) IV y V



(e) V y VI

Figura 3.3: Imágenes sintéticas generadas con manchas pertenecientes a categorías consecutivas. Cada categoría se representa por un color: I-rojo, II-azul, III-verde, IV-rosa, V-amarillo, VI-cian.

Capítulo 4

Resultados y análisis

Para configurar un modelo de *deep learning* debemos tomar una serie de decisiones. Por una parte, debemos elegir el tipo de arquitectura: número, tamaño y tipo de capas. Por otra parte, las funciones de pérdida, el optimizador, las funciones de activación y número de épocas, entre otros. Para definir estos parámetros hacemos uso en primer lugar de la bibliografía, entendiendo cada parte de la red y decidiendo lo que mejor se aplica al problema que queremos resolver. En ocasiones no existen reglas generales que podamos aplicar, por lo que nos basamos en ensayos de prueba y error para analizar qué configuraciones funcionan mejor.

Como se ha detallado en la introducción, en este trabajo se han implementado y optimizado dos modelos de redes neuronales convolucionales para la clasificación de las principales formas del carbono en muestras de fundiciones. El primer modelo toma una imagen con todos los defectos del mismo tipo como las que se muestran en la imagen 3.2 y el modelo es entrenado para aprender a identificar a qué categoría pertenece la imagen. En la segunda parte, se han generado imágenes pertenecientes a categorías consecutivas como las de la imagen 3.3 y, transformando la tarea de clasificación de imágenes en un problema de segmentación semántica, se ha entrenado la red para clasificar cada píxel de la imagen. En este capítulo presentamos el análisis de cada uno de los modelos, explicando las decisiones que se han tomado y los resultados obtenidos.

Todo el código que se ha implementado en este trabajo así como los conjuntos de datos usados se pueden encontrar en un repositorio de GitHub en [32]. El lenguaje de programación usado ha sido *Python* ya que dispone de las librerías *Keras* y *TensorFlow* [33, 34]. *TensorFlow* es una librería *open source* desarrollada por *Google* que permite desarrollar modelos de *machine learning* debido a su rapidez de computación. *Keras* se centra en la implementación de modelos de *deep learning* usando *TensorFlow* como *backend*. La ventaja de *Keras*, a parte de su extensa documentación y simplicidad, es que permite desarrollar modelos desde su configuración inicial hasta la evaluación y generación de predicciones. En cuanto al equipo usado, se trata de un CPU *Intel(R) Core(TM) i7-8565* y 8GB de memoria RAM.

4.1. Clasificación de imágenes pertenecientes a una única categoría

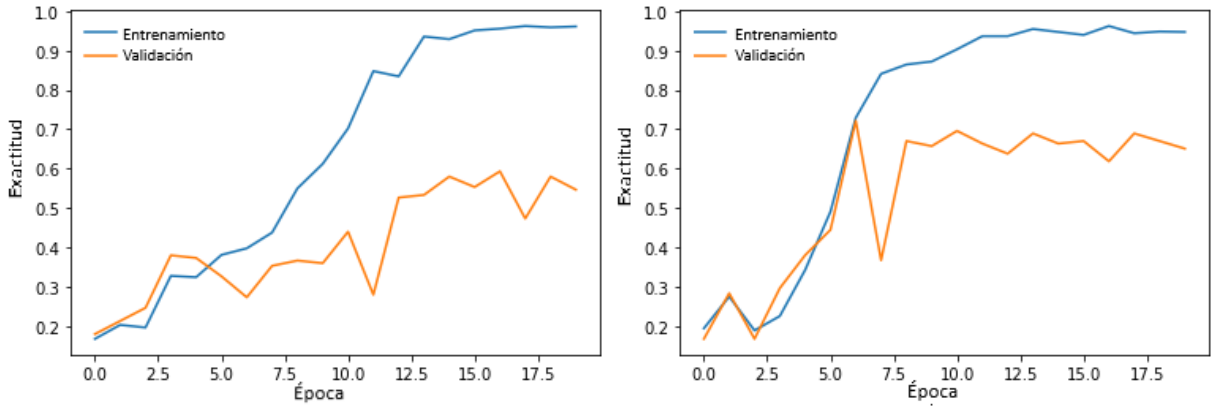
Partiendo de las imágenes sintéticas generadas (imagen 3.2), comenzamos implementando un modelo sencillo de clasificación de imágenes. Para esta primera parte, los nombres de las carpetas en las que se almacenan las imágenes de cada categoría sirven como etiquetas para el modelo.

Se comenzó tomando como referencia la arquitectura presentada en [35], que consta de capas de tipo convolucional, *max pooling*, capas de aplanamiento y completamente conectadas. Como función de pérdida se ha usado *categorical cross-entropy* y el optimizador *RMSProp*. Todas las capas usan *ReLU* como función de activación a excepción de la última que usa *softmax*. Puesto que tenemos 6 categorías, el resultado del modelo será un vector 6-dimensional cuyas entradas tomarán valores en el rango $[0, 1]$ según la probabilidad de que la imagen analizada pertenezca a la i-ésima categoría dado el valor de entrada x. El número de épocas es 19 y el tamaño de lote 32. Para el reparto de los conjuntos de entrenamiento y validación se ha usado la técnica *K-fold* con 5 subconjuntos.

El principal problema con este modelo ha sido el *overfitting*. La red aprende muy bien las características de las imágenes del set de entrenamiento, pero falla considerablemente en el set de validación. La exactitud de este primer modelo es 55 % para el set de validación y la evolución a lo largo de las épocas se puede ver en la imagen 4.1a. Como vemos, el *overfitting* comienza a partir de la época 5 que es cuando las curvas de los sets de entrenamiento y validación comienzan a separarse. Se ha decidido entonces tratar de mitigar el sobreajuste mediante técnicas de regularización como las que se han presentado en la sección 2.4. En particular, añadir capas *dropout* ha permitido mejorar ligeramente los resultados, ya que conseguimos una exactitud de 67 % (imagen 4.1b). En esta gráfica vemos que las curvas se separan a partir de la época 6, y que el error de generalización, definido como la distancia entre las curvas de entrenamiento y validación, se ha reducido con respecto al modelo anterior. También se ha observado que el modelo funciona mejor con el optimizador *Adam* que con el *RMSProp*, pero los resultados son todavía algo pobres.

El hecho de que ninguna técnica de optimización ni regularización haya mejorado lo suficiente los resultados hace pensar que el problema está en la generación de las imágenes. Puesto que se han generado todas las imágenes a partir de un número muy reducido de defectos originales, muchas de ellas resultan redundantes para el algoritmo, lo que dificulta la tarea de clasificación. Además, tienen muy alta correlación entre ellas ya que no generamos nueva información para ayudar a que el modelo generalice, sino que transformamos y mezclamos la información de la que disponemos en cada una de las imágenes originales. Como se puede observar en la imagen 4.1b, la exactitud del modelo se mantiene bastante estable en las últimas épocas, por lo que no parece que haciendo más iteraciones vayamos a conseguir una mejora significativa.

La exactitud alcanzada en este punto no es suficiente para los objetivos marcados, por lo que se ha optado por recurrir a una red pre-entrenada. De acuerdo con la bibliografía, esta técnica funciona especialmente bien cuando se trata de conjuntos de datos no lo suficientemente grandes, como es nuestro caso. Se ha usado la red convolucional pre-entrenada *MobileNet* [36, 37], la cual se puede usar en multitud de contextos, ya que ha sido entrenada usando el dataset COCO[38] que consta de más de 200K imágenes de objetos cotidianos tales como animales, comida, ropa, vehículos, etc. y 81 categorías de salida. Además, tiene una arquitectura ligera que reduce significativamente el tamaño del modelo y el tiempo de computación. Se ha tomado la salida de este modelo congelando



(a) Resultados con overfitting

(b) Resultados tras aplicar capas de *dropout*

Figura 4.1: Comparación de los resultados obtenidos para la exactitud de nuestro modelo a lo largo de las épocas cuando el modelo presenta *overfitting*. Se muestran los valores obtenidos para el set de entrenamiento en azul y para el set de validación en naranja. En (b) podemos ver cómo la exactitud ha aumentado ligeramente tras aplicar técnicas de regularización.

todas las capas para evitar que los pesos se actualicen durante el entrenamiento y solamente hemos añadido dos capas entrenables: una capa *max pooling* y otra completamente conectada con 6 neuronas y activación *softmax* que actúa como clasificador. El número de parámetros totales del modelo es 3 millones, pero entrenables sólo hay 6 mil. En el proceso de entrenamiento se ha usado un set con 3300 imágenes (550 por cada clase) de las cuales 300 (50) se han usado en el set de validación. El número de épocas establecido es 7 y el optimizador es *Adam* con tasa de aprendizaje 10^{-4}. Como vimos en el capítulo 2, este algoritmo modifica la tasa de aprendizaje en cada época, luego lo que hacemos es fijar α_0 de manera que $\alpha_i = \frac{\alpha_0}{\sqrt{i}}$ para la $i-$ésima época. En la imagen 4.2 se puede ver cómo evolucionan la exactitud y la función de pérdida para cada uno de los *folds* cuando los evaluamos sobre el set de validación. Observamos que la exactitud ya es relativamente alta en las primeras épocas, debido al hecho de que estamos usando un modelo pre-entrenado que ya sabe distinguir patrones en las imágenes. A partir de la época 4 vemos que la función de pérdida ya alcanza valores muy bajos, aunque sigue disminuyendo hasta la época 6. Podríamos entrenar el modelo durante más épocas, pero la tendencia de la función de pérdida indica que por cada época que añadamos al modelo, cada vez vamos a obtener mejoras menos significativas. En la imagen 4.3 podemos observar la comparación de la exactitud obtenida en los sets de entrenamiento y validación. Estos resultados se han obtenido a partir de la media de la exactitud obtenida en los 5 *folds* para cada época. De esta gráfica destacamos que las curvas comienzan ligeramente separadas, pero rápidamente se solapan a partir de la segunda época, por lo que podemos garantizar que nuestro modelo no ha generado *overfitting*.

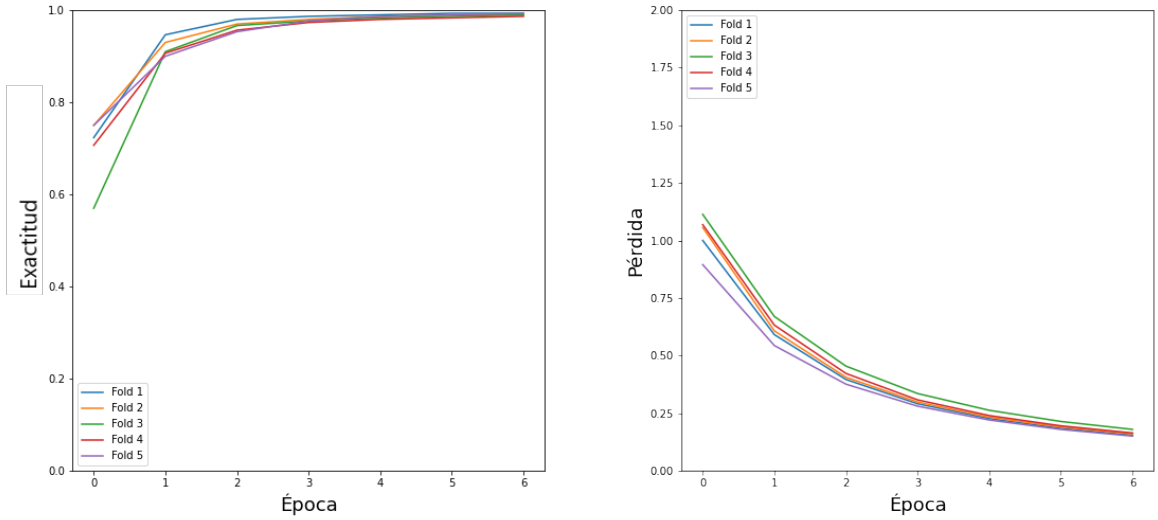


Figura 4.2: Evolución de la exactitud (izquierda) y la función de pérdida (derecha) con respecto al número de épocas cuando se evalúa el modelo sobre el set de validación para cada uno de los *folds*.

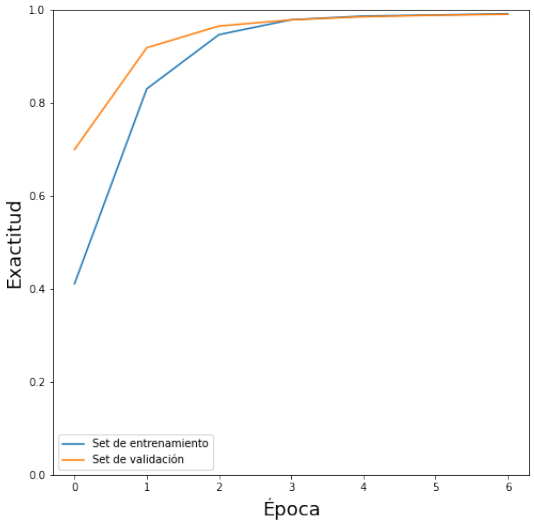


Figura 4.3: Evolución de la exactitud con respecto al número de épocas para los sets de entrenamiento y validación. En ambos casos la exactitud es la media obtenida en cada época para los 5 *folds*.

Para el modelo final se ha obtenido una exactitud de 98,9 %±0,4 cuando evaluamos los resultados predichos para un set de test con 200 imágenes por categoría mezcladas entre sí. Este set de imágenes ha sido generado de la misma manera que los conjuntos de entrenamiento y validación, pero son imágenes que el modelo no ha visto en el proceso de entrenamiento. La tabla 4.1 muestra los resultados de la exactitud por categoría, mientras que en la tabla 4.4 podemos ver el detalle

del porcentaje de imágenes que pertenecen a la categoría X y el algoritmo las clasifica en la categoría Y. A partir de estos resultados, observamos que las categorías I y VI son las que mejor identifica. Las categorías IV y V son las que peor se predicen, ya que son manchas similares y el modelo confunde unas con otras. En la tabla 4.1 también se muestra el detalle de la probabilidad media con la que el algoritmo predice cada clase. Es interesante observar que la clase II se predice correctamente menos veces que la clase III pero la certeza con la que se predice es más alta. Lo mismo ocurre con las clases V y IV. Asimismo, destacamos el hecho de que la probabilidad media más alta que se obtiene cuando la predicción es incorrecta es 15 % y ocurre cuando se predice la clase IV pero la imagen pertenece a la clase V. Esto significa que, aunque la certeza pueda no ser excesivamente alta en algunas clases, siempre hay mucha diferencia entre la clase con la probabilidad más alta y las demás.

	I	II	III	IV	V	VI
Exactitud	$100\,\% \pm 0$	$99{,}0\,\% \pm 0{,}4$	$99{,}4\,\% \pm 0{,}4$	$97{,}9\,\% \pm 0{,}4$	$97\,\% \pm 1$	$99{,}7\,\% \pm 0{,}2$
Certeza	$92{,}8\,\% \pm 0{,}7$	$87{,}6\,\% \pm 0{,}9$	$83{,}6\,\% \pm 0{,}7$	$76\,\% \pm 2$	$79\,\% \pm 2$	$95{,}0\,\% \pm 0{,}6$

Tabla 4.1: Exactitud y probabilidad media obtenidas usando el modelo final para predecir a qué categoría pertenecen 200 imágenes de cada tipo. Se muestran los resultados obtenidos para 5 *folds* y el error (desviación típica) en cada caso. La certeza indica la probabilidad media con la que el algoritmo predice cada clase.

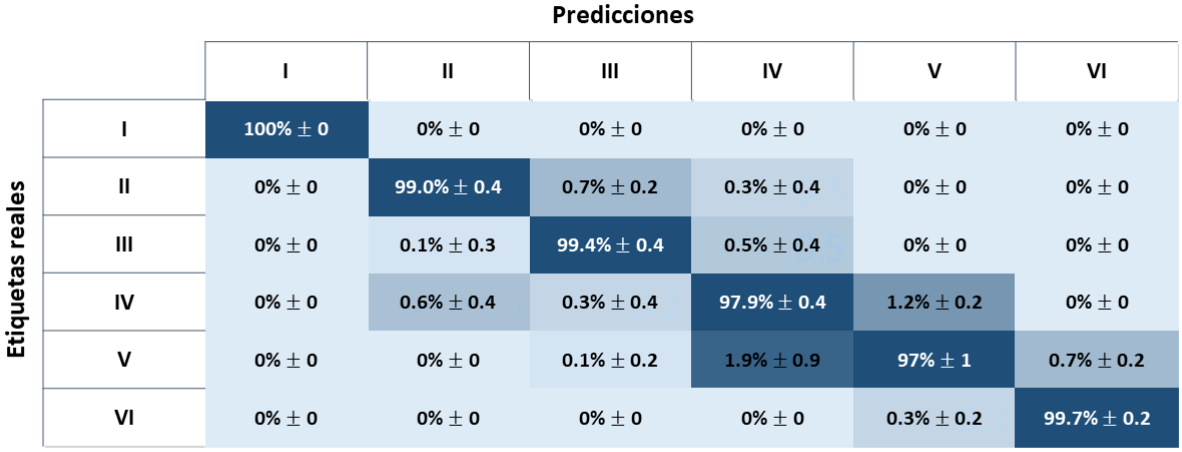


Figura 4.4: Porcentaje de imágenes pertenecientes a la categoría X (filas) que han sido clasificadas como categoría Y (columnas). Estos resultados han sido obtenidos con el modelo final para un set de 200 imágenes de test por categoría. Los errores que se muestran son la desviación típica en cada caso.

4.2. Clasificación de defectos pertenecientes a categorías consecutivas

Para la segunda parte de este trabajo se usaron imágenes de partida como las que se muestran en la imagen 3.3 pero en blanco y negro. El etiquetado de las imágenes se realiza coloreando los píxeles según la categoría a la que pertenecen: los píxeles de color blanco tienen asignado el valor 0 (fondo), los rojos el 1 (forma I), los azules el 2 (forma II) etc. Esto significa que nuestro problema tiene ahora 7 categorías, 6 pertenecientes a las distintas manchas y el fondo (clase 0), mientras que nuestras imágenes son de 5 tipos (I y II, II y III, III y IV, IV y V, V y VI). Una vez el modelo esté entrenado y reciba una imagen nueva, devolverá una imagen con la misma resolución que la imagen original en forma de máscara coloreada según la predicción que se haya asignado a cada píxel. Un ejemplo se muestra en la imagen 4.5.

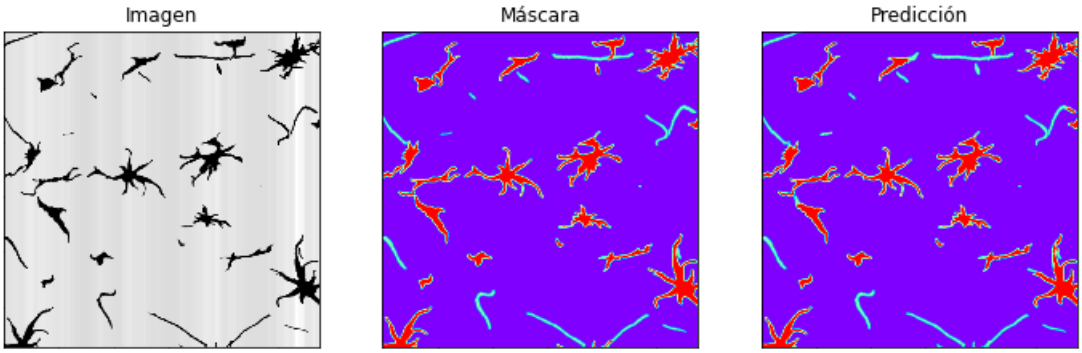


Figura 4.5: Comparación de una imagen de test en su versión original (izquierda), máscara (centro) y la predicción que realiza el modelo (derecha).

Se ha optado de nuevo por recurrir a un modelo pre-entrenado. Para un problema de segmentación semántica como éste, una red que es muy potente es la arquitectura *U-Net*, diseñada en un principio para resolver problemas de segmentación de imágenes biomédicas pero que ha demostrado dar muy buenos resultados para casi cualquier otro tipo de imágenes [39]. *U-Net* resuelve el problema de la complejidad computacional que supone clasificar cada píxel de la imagen dividiendo la arquitectura en una parte de contracción y otra de expansión. La parte de contracción, también llamada codificador, está formada por capas convolucionales con activación *ReLU* y capas *max pooling* intercaladas con capas de *dropout*. El bloque de expansión, o decodificador, está formado por convoluciones traspuestas que reconstruyen la imagen original [40]. Se han cargado los pesos pre-entrenados de la red *U-Net* entrenada con el conjunto de datos *ImageNet*, uno de los conjuntos de imágenes de referencia para modelos de computación visual ya que cuenta con más de 14 millones de imágenes y más de 20 mil categorías [41].

El principal problema en esta parte del trabajo ha sido el desbalanceo de clases. Nuestras imágenes tienen un porcentaje extremadamente elevado de píxeles pertenecientes al fondo, en torno al 93 % del total de píxeles. Aunque no nos interesa que el algoritmo aprenda a clasificar estos píxeles, en un problema de segmentación semántica es importante que todos los píxeles de la imagen estén asociados a una categoría, es decir, no pueden quedar píxeles sin etiqueta. Esto supone que, desde las primeras épocas, el modelo aprende a predecir correctamente dichos píxeles y la exactitud supera el 95 % muy rápido. Sin embargo, este resultado no significa que nuestro modelo

esté resolviendo bien el problema que nos ocupa. En la tabla 4.2 se muestra cuál es el porcentaje de píxeles de cada tipo para nuestro set de test, junto con la descomposición de la exactitud y el valor-F por clase para uno de los primeros modelos que se probaron con función de pérdida *categorical cross-entropy*. Si observamos el valor-F, podemos observar que el fondo (clase 0) y la clase VI se predicen muy bien, pero el algoritmo no clasifica con suficiente exactitud el resto de categorías. El problema que supone tener las clases desbalanceadas es que los píxeles del fondo, que es la clase mayoritaria, son muy fáciles de predecir y dominan el error del algoritmo restando importancia al error que comete en el resto de las clases. Para resolver este problema, en [21] proponen usar la función de pérdida focal que definimos en el capítulo 2. En la siguiente tabla se muestran los resultados obtenidos con el mismo modelo simplemente cambiando la función de pérdida.

		0	I	II	III	IV	V	VI
	% de píxeles	93,1 %	0,3 %	1,7 %	0,9 %	1,5 %	1,6 %	0,8 %
Categorical cross-entropy	Exactitud	99 %	70 %	66 %	47 %	48 %	63 %	87 %
	Valor-F	0,99	0,76	0,81	0,81	0,81	0,16	0,88
Función focal	Exactitud	100 %	84 %	93 %	89 %	91 %	86 %	90 %
	Valor-F	1,00	0,84	0,93	0,89	0,90	0,86	0,90

Tabla 4.2: Dado un conjunto de 250 imágenes de test, se muestra el porcentaje de píxeles pertenecientes a cada categoría y una comparación de la exactitud y valor-F obtenidos cuando se usan la función de pérdida *categorical cross-entropy* y la función focal.

El modelo final que se ha dado por válido para esta parte tiene las siguientes características. Hemos partido del modelo pre-entrenado *U-Net* para la clasificación de imágenes RGB de tamaño 128 x 128 y 7 categorías (fondo y 6 clases de manchas). Todas las capas usan $ReLU$ como función de activación a excepción de la última que usa *softmax*, de manera que nuestro output es un vector que nos devuelve 7 probabilidades y tomamos como predicción para cada píxel la clase con mayor probabilidad. En el proceso de entrenamiento se ha usado un set con 1250 imágenes de las cuales 200 se han usado en el set de validación. En este caso, hemos entrenado el algoritmo durante 100 épocas y el tamaño de lote elegido es 8, ya que tomar conjuntos más grandes supone un coste de memoria excesivamente alto que el equipo usado no es capaz de soportar. El algoritmo de optimización usado es *Adam* con tasa de aprendizaje 10^{-4}, ya que en el desarrollo anterior observamos que daba mejores resultados que *RMSProp*. Para el reparto de los conjuntos de entrenamiento y validación se ha usado la técnica *K-fold* con 5 subconjuntos.

En la imagen 4.6 se puede ver cómo evolucionan la exactitud y la función de pérdida a lo largo de las épocas. Observamos que la exactitud crece en las primeras épocas y a partir de la época 15 se mantiene bastante estable. Aunque en la imagen no se aprecie por la escala usada, la exactitud sigue creciendo ligeramente en las sucesivas épocas a medida que el modelo aprende a reconocer patrones pertenecientes a las clases minoritarias. Necesitamos que el modelo siga aprendiendo estos patrones ya que es importante para nuestro problema que aprenda a distinguir bien unas manchas de otras, aunque esta mejora en el aprendizaje apenas afecta a la exactitud total del modelo. Observamos que, de la misma manera, la función de pérdida se vuelve insignificante a partir de la época 15. Cabe destacar que desde las primeras épocas la exactitud ya es muy alta, ya que, por una parte, estamos usando un modelo pre-entrenado que ya sabe cómo reconocer ciertos patrones en las imágenes y, por otra parte, tenemos un número extremadamente alto de píxeles pertenecientes al fondo, los cuales son muy fáciles de identificar para una red convolucional. En cuanto al problema del *overfitting*, observamos que el modelo da resultados muy similares para los conjuntos de validación y entrenamiento en todo momento, por lo que no ha sido necesario aplicar ninguna técnica de regularización en este caso.

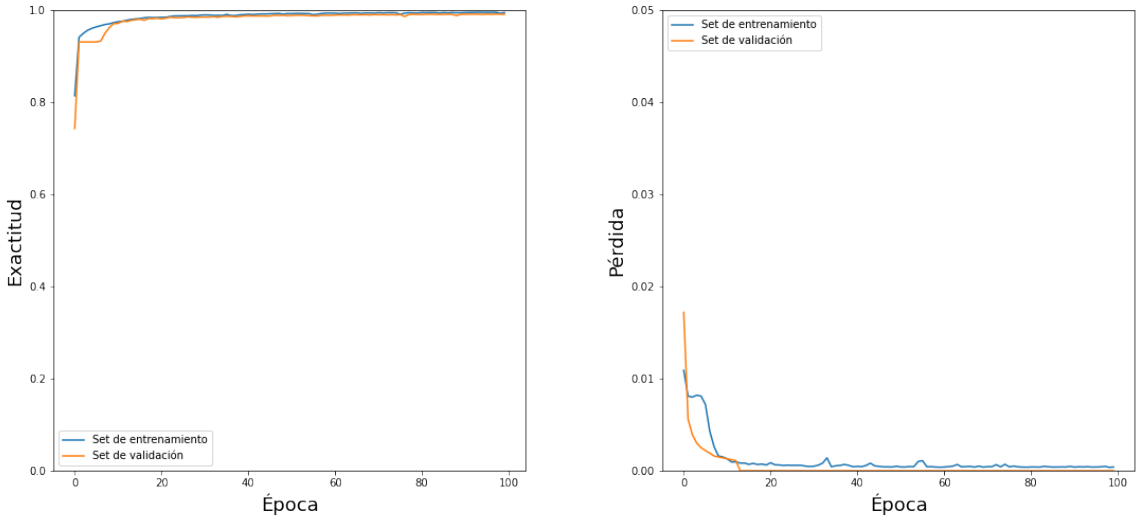


Figura 4.6: Evolución de la exactitud (izquierda) y la función de pérdida focal (derecha) con respecto al número de épocas para los sets de entrenamiento y validación. Tanto la exactitud como la pérdida son la media obtenida en cada época para los 5 *folds*.

Con el modelo final se ha obtenido una exactitud de $99{,}21\,\% \pm 0{,}04$ cuando evaluamos los resultados predichos para un set de test con 250 imágenes. Para este mismo conjunto, obtenemos una exactitud de $91\,\% \pm 1$ sin tener en cuenta los píxeles del fondo. En la tabla 4.7 podemos ver el detalle del porcentaje de píxeles que pertenecen a la categoría X y el algoritmo clasifica en la categoría Y. A partir de esta tabla observamos que, sin contar el fondo, las categorías II, IV y VI son las que mejor clasifica. Las categorías I y V son las que peor se predicen, ya que principalmente se confunden con las categorías II y IV, respectivamente. Es interesante destacar también el hecho de que todas las categorías tienen un pequeño porcentaje de píxeles clasificados como clase 0. Esto es debido a que el algoritmo falla a menudo a la hora de distinguir los bordes de las manchas. En el caso práctico este error no tiene importancia, ya que basta con que identifique en qué parte de la imagen hay una mancha y de qué tipo es, sin importar demasiado donde empieza y acaba.

En la imagen 4.8 se muestra una imagen de test para cada tipo de forma comparando la imagen original, su máscara y la predicción que realiza nuestro modelo. Observamos que en general los resultados son muy buenos, a excepción de pequeños errores que observamos principalmente cuando las manchas se solapan, ya que al modelo le cuesta diferenciar dónde acaba y empieza cada una.

Benchmarking

Para este desarrollo se han generado imágenes con manchas pertenecientes a categorías consecutivas (I y II, II y III, III y IV, IV y V, V y VI) y con el porcentaje de manchas de cada tipo definido: $20\,\% - 80\,\%$, $40\,\% - 60\,\%$, $60\,\% - 40\,\%$, $80\,\% - 20\,\%$. Nótese que los porcentajes en este caso se aplican al número de manchas, no al número de píxeles. La idea de este desarrollo es

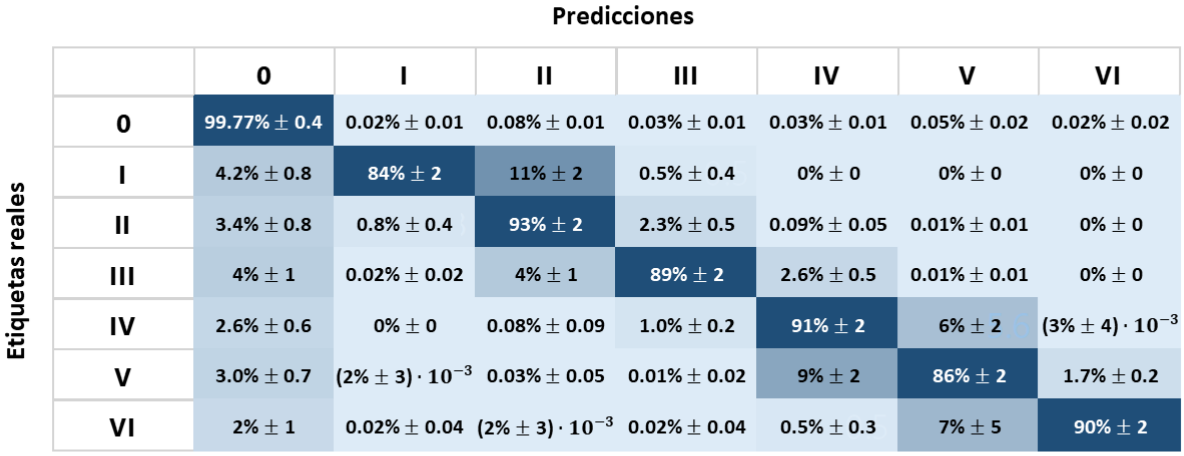


Figura 4.7: Porcentaje de imágenes pertenecientes a la categoría X (filas) que han sido clasificadas como categoría Y (columnas). Estos resultados han sido obtenidos con el modelo final para un set de 50 imágenes de test por categoría (I y II, II y III, etc.). Los errores que se muestran son la desviación típica en cada caso.

analizar cómo varían las predicciones que hace el modelo según el porcentaje de manchas de cada tipo que están presentes en las imágenes de test. Como podemos observar, todas las categorías dan muy buenos resultados, quedando ligeramente por debajo de la media las de las categorías IV-V y V-VI.

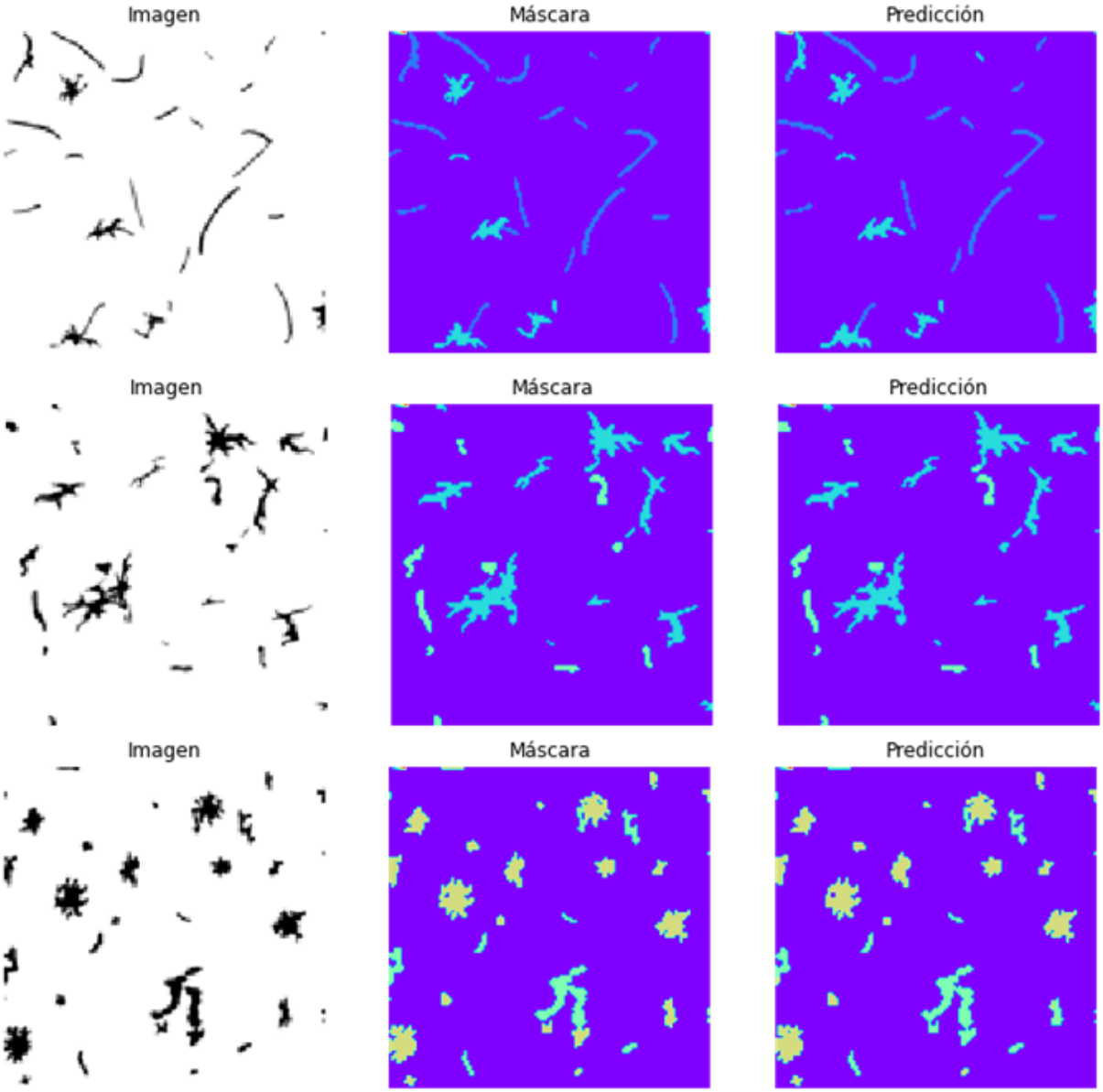


Imagen
Máscara
Predicción
Imagen
Máscara
Predicción
Imagen
Máscara
Predicción

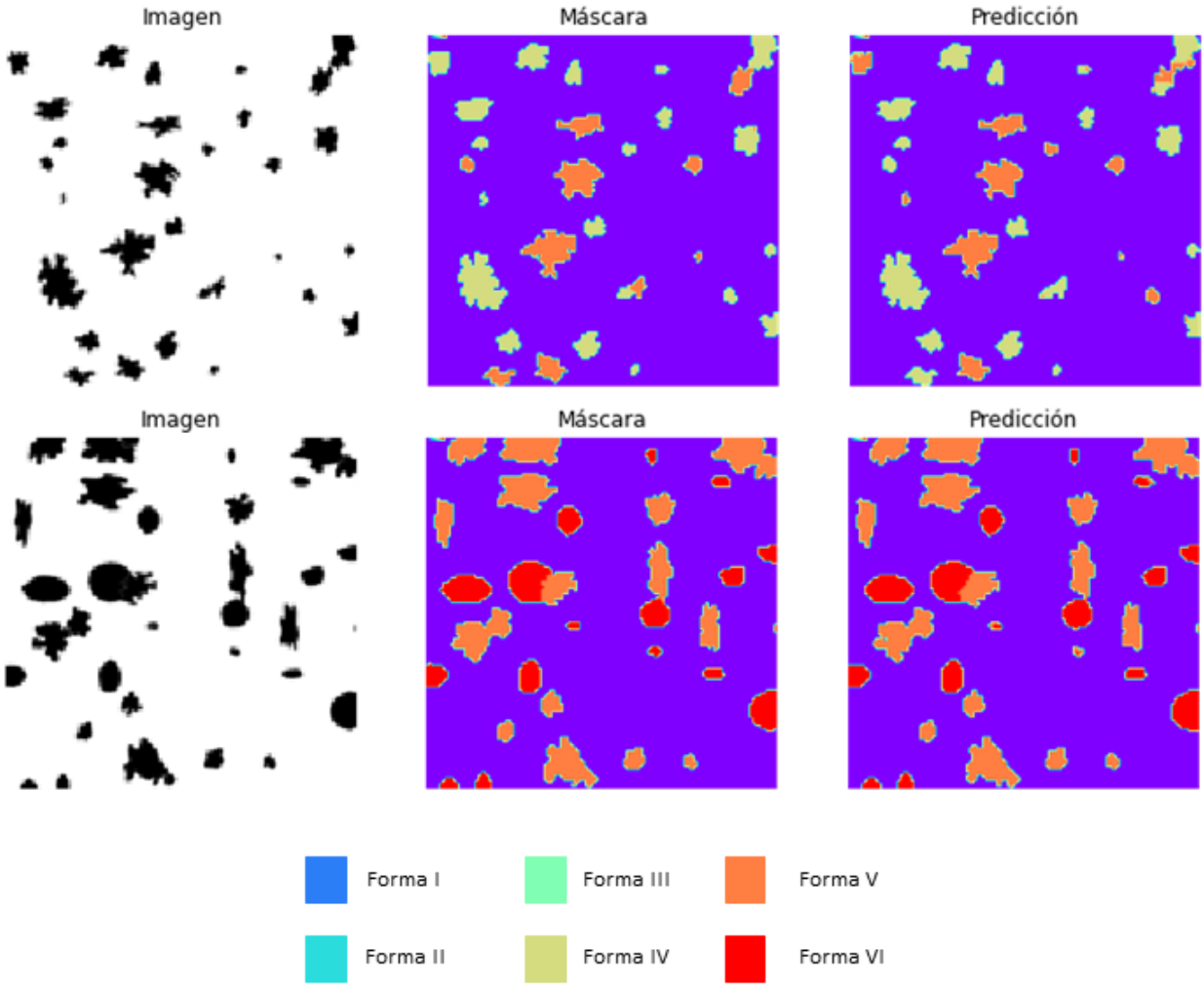


Figura 4.8: Comparación de una imagen de test en su versión original (izquierda), máscara (centro) y predicción que realiza el modelo (derecha) para cada tipo de imagen.

	20 %-80 %	**40 %-60 %**	**60 %-40 %**	**80 %-20 %**
I y II	99,24 % ± 0,05	99,29 % ± 0,05	99,30 % ± 0,03	99,34 % ± 0,06
II y III	99,16 % ± 0,08	99,10 % ± 0,05	99,12 % ± 0,05	99,14 % ± 0,04
III y IV	99,1 % ± 0,2	99,37 % ± 0,03	99,35 % ± 0,05	99,3 % ± 0,1
IV y V	98,7 % ± 0,3	99,0 % ± 0,1	98,7 % ± 0,1	98,4 % ± 0,2
V y VI	99 % ± 1	98,9 % ± 0,7	99,0 % ± 0,2	98,6 % ± 0,1

Tabla 4.3: Resultados de la exactitud obtenidos cuando el porcentaje de manchas de cada tipo está definido, para un set de test de 1000 imágenes (50 de cada tipo). Los resultados se muestran acompañados de su error (desviación típica) para 5 *folds*.

Capítulo 5

Conclusiones y trabajo futuro

En el desarrollo de este trabajo hemos visto que las técnicas de *deep learning* aplicadas a la clasificación de imágenes se basan en modelos que toman conjuntos de imágenes de entrada y, conociendo a qué clase pertenecen, se entrenan para aprender a identificar características o patrones para cada una de las clases. En particular, las CNN han supuesto una revolución en el campo del reconocimiento de imágenes debido a su gran eficiencia en comparación con las redes neuronales tradicionales y otros algoritmos de clasificación de imágenes. El motivo por el que son tan eficaces es porque son capaces de identificar un patrón en un punto de la imagen y reconocerlo incluso si cambia su posición. Además, disminuyen significativamente el coste computacional al reducir el número de conexiones entre neuronas.

En este trabajo hemos implementado y optimizado dos algoritmos de CNN para la clasificación de las principales formas del grafito en muestras de fundiciones. En el primer modelo, hemos clasificado imágenes con manchas pertenecientes a una única categoría. En este caso hemos usado la función de pérdida *categorical cross-entropy* que cuantifica la diferencia entre dos distribuciones de probabilidad y hemos obtenido una exactitud de $98{,}9\,\% \pm 0{,}4$ para un set de validación de 200 imágenes. Hemos visto que las clases que peor se predicen son la IV y la V, como cabría esperar, ya que son las formas más similares entre ellas y las que peor diferenciaba el software comercial. También se ha analizado la probabilidad con la que se predicen las clases y se ha comprobado que, cuando el algoritmo predice correctamente una clase, lo hace con una probabilidad media de más del $75\,\%$.

En el segundo modelo, la clasificación se ha hecho píxel a píxel para imágenes que mezclan manchas pertenecientes a categorías consecutivas, por lo que se ha desarrollado un modelo basado en una tarea de segmentación semántica. La principal diferencia entre los dos modelos es que en el primero no importa la localización, por lo que podemos permitir que el modelo reduzca en cada capa el tamaño de la imagen original mientras extrae los patrones más representativos. Sin embargo, en el segundo modelo nos importa tanto la extracción de los patrones como su localización, por lo que necesitamos que la salida del modelo tenga las mismas dimensiones que la entrada, aumentando considerablemente los recursos computacionales necesarios para entrenar el modelo. Definiendo como función de pérdida la función *categorical cross-entropy* obtenemos un $97{,}1\,\% \pm 0{,}3$ de exactitud, mientras que con la función focal hemos conseguido aumentarla en un $2\,\%$ consiguiendo $99{,}21\,\% \pm 0{,}04$. Un $97\,\%$ de exactitud parece un buen resultado, pero, como vimos en el capítulo anterior, este valor está sesgado ya que en realidad el algoritmo no

clasifica con suficiente exactitud las manchas. Hemos lidiado con el problema del desbalanceo de clases causado por el gran porcentaje de píxeles del fondo y hemos comprobado cómo cambiando únicamente la función de pérdida conseguimos mejorar significativamente los resultados. En este caso, para un set de 250 imágenes de test, las clases peor predichas son la I y la V. Por otra parte, el modelo comete bastantes errores al clasificar manchas que se solapan, ya que le cuesta diferenciar dónde empieza y acaba cada una de ellas.

En ambos casos hemos recurrido a la técnica del aprendizaje transferido, y hemos podido comprobar cómo no sólo mejoran los resultados con respecto a un algoritmo entrenado desde cero, si no que se reduce considerablemente el tiempo de computación. En particular, hemos tomado los modelos pre-entrenados *MobileNet* y *U-Net*, ambos diseñados para resolver la tarea de la clasificación de imágenes. Nos ha bastado con realizar un ajuste fino congelando las capas iniciales para no modificar los pesos durante el entrenamiento y entrenando solamente las capas superiores que actúan como clasificador. En ambos casos hemos implementado la técnica del *K-fold* con 5 subconjuntos, aumentando así el tiempo de computación, pero obteniendo resultados más fiables. Cabe destacar la capacidad de generalización de los modelos, ya que los resultados en ambos casos cuando se evalúa sobre el set de test son muy similares a los obtenidos en el set de entrenamiento. Esto significa que ninguno de los modelos presenta *overfitting* y el principal motivo ha sido el uso de la técnica del aprendizaje transferido. Cuando introducimos un nuevo conjunto de datos en un modelo ya entrenado, estamos pidiendo que el modelo tenga un buen rendimiento en datos que están relacionados con la tarea para la que fue entrenado, pero que no son exactamente iguales, con lo que implícitamente conseguimos evitar que el modelo genere *overfitting*. En cuanto al número de épocas, el primer algoritmo realiza solamente 7 iteraciones, mientras que necesitamos que el segundo realice más de 50 para empezar a clasificar con suficiente eficiencia las manchas de las imágenes. Si en un futuro quisiéramos que el modelo clasifique imágenes de mayor tamaño sería necesario usar un equipo más potente y, probablemente, aumentar el número de épocas.

Uno de los principales problemas que observamos es la disponibilidad de datos, la cual limita la complejidad de la red. Puesto que la norma que hemos usado de referencia sólo muestra una imagen por cada categoría, hemos tenido que recurrir a la creación de imágenes sintéticas mediante la técnica del aumento de datos. Todas las manchas que se han creado provienen de la transformación de un conjunto reducido de manchas originales, lo que hace que, en ocasiones, los datos resultan redundantes para el algoritmo.

En el primer capítulo nos planteábamos las siguientes preguntas: ¿somos capaces de automatizar la clasificación de los nódulos mediante una red convolucional? ¿distingue esta red los diferentes tipos de formas con suficiente exactitud? Podemos concluir que la respuesta a ambas preguntas es afirmativa. En ambos modelos, partiendo de un conjunto de imágenes etiquetadas, hemos desarrollado dos algoritmos de CNN que automatizan la extracción de patrones de las imágenes originales y son capaces de realizar predicciones sobre conjuntos de datos nuevos. En cuanto a la exactitud de los modelos, en ambos casos hemos obtenido resultados bastante precisos, si bien en el segundo modelo nos hubiese gustado aumentar un poco la exactitud de las formas I y V, en general podemos estar satisfechos con los resultados obtenidos.

Como se ha comentado anteriormente, el software usado en el Departamento de Ciencia e Ingeniería del Terreno y de los Materiales de la Universidad de Cantabria no clasifica correctamente las formas del carbono en muestras de fundiciones, ya que clasifica un 80 % de las manchas de la forma V como forma IV. En nuestro caso, hemos logrado que la exactitud sea $86\,\% \pm 2$ para

el número de píxeles de la categoría V, con lo que hemos conseguido mejorar significativamente los resultados. Otra ventaja que tiene nuestro este método, y que es especialmente importante para este trabajo, es poder dar unos resultados de la exactitud junto con su error, el cual hemos calculado como la desviación típica del conjunto de resultados obtenido para los 5 *folds*.

A continuación damos una perspectiva del trabajo futuro que queda por hacer en esta dirección. En primer lugar, para lograr una mejora y poder aplicar estos modelos en el laboratorio, deberíamos volver a entrenarlos con imágenes reales obtenidas con el microscopio óptico. Este desarrollo supone varios desafíos ya que es importante tener en cuenta que la calidad de los resultados depende altamente de la calidad del set de imágenes usado para el entrenamiento. Si usamos imágenes reales como las que se mostraron en el primer capítulo, tenemos que asegurarnos de que la iluminación, contraste y la superposición de manchas, entre otros, son los adecuados.

Por otra parte, en la norma [4] también se aportan referencias para la clasificación del grafito según su distribución para la forma y tamaño. Una vez tenemos un algoritmo que predice el tipo de forma con suficiente exactitud, podríamos desarrollar un nuevo modelo de *deep learning* tal que, dada una mancha y la categoría a la que pertenece, sea entrenado para identificar su tamaño y distribución.

Bibliografía

[1] Shackelford J.F. *Introduction to Materials Science for Engineers*. 1.ª ed. Pearson Prentice Hall, 2014. ISBN: 0133826651.

[2] Smith W.F. *Foundations of Materials Science and Engineering*. 5.ª ed. McGraw Hill, 2009. ISBN: 9780073529240.

[3] Callister W.D. y Rethwisch D.G. *Fundamentals of Materials Science and Engineering. An Integrated Approach*. 3.ª ed. John Wiley y Sons Inc., 2008. ISBN: 9780470125373.

[4] *Microestructura de la fundición de hierro. Parte 1: Clasificación del grafito por análisis visual (ISO 945-1:2019)*. 2020.

[5] Chollet F. *Deep Learning with Python*. 1.ª ed. Manning Publications, 2017. ISBN: 9781617294433.

[6] Brownlee J. *Deep Learning With Python: Develop Deep Learning Models on Theano and TensorFlow Using Keras*. 1.20. Machine Learning Mastery, 2020.

[7] Hiremath P. S. Prakash P. et al. *Characterization of Flake Graphite Forms in Digital Microstructure Images of Gray Cast Iron using Digital Image Analysis Techniques*. 2011.

[8] Yiyong T. Hong J. et al. «Auto-analysis system for graphite morphology of grey cast iron». En: *Journal of automated methods and management in chemistry* 25 (2003), págs. 87-92. DOI: 10.1155/S1463924603000154.

[9] Khaled A. y Mostafa A. «Grey Cast Iron Categorization using Artificial Neural Network». En: *International Journal of Scientific and Engineering Research* 5 (2014).

[10] Tavares J. Albuquerque V. y Cortez P. «Quantification of the Microstructures of Hypoeutectic White Cast Iron using Mathematical Morphology and an Artificial Neuronal Network». En: *International Journal of Microstructure and Materials Properties* 5 (2010), págs. 52-64. DOI: 10.1504/IJMMP.2010.032501.

[11] Cortez P. Albuquerque V. et al. «A New Solution for Automatic Microstructures Analysis from Images Based on a Backpropagation Artificial Neural Network». En: *Nondestructive Testing and Evaluation* 23 (2008). DOI: 10.1080/10589750802258986.

[12] Gusev A. Sivkova T. y Syropyatov A. «Technology for Cast Iron Microstructure Analysis in SIAMS Software Using Neural Networks». En: *Proceedings of the 31th International Conference on Computer Graphics and Vision* 2 (2021).

[13] Cortez P.C. Albuquerque V. et al. «Evaluation of multilayer perceptron and self-organizing map neural network topologies applied on microstructure segmentation from metallographic images». En: *NDT and E International* 42 (2009), págs. 644-651. DOI: 10.1016/j.ndteint.2009.05.002.

[14] Y. Goodfellow I. Bengio y Courville A. *Deep Learning*. 1.20. MIT Press, 2016. ISBN: 9780262035613.

[15] Elgendy M. *Deep Learning for Vision Systems*. Manning Publications, 2020. ISBN: 9781617296192.

[16] *A Comprehensive Guide to Convolutional Neural Networks, the ELI5 way*. URL: https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53.

[17] *Types of Convolution Kernels : Simplified*. URL: https://towardsdatascience.com/types-of-convolution-kernels-simplified-f040cb307c37.

[18] *Convolution Operation Demo*. URL: https://deeplizard.com/resource/pavq7noze2.

[19] Dumoulin V. y Francesco V. *A guide to convolution arithmetic for deep learning*. 2016. DOI: 10.48550. arXiv: 1603.07285. URL: https://arxiv.org/abs/1603.07285.

[20] Bertsekas D y Tsitsiklis J. «Gradient Convergence In Gradient Methods With Errors». En: *SIAM Journal on Optimization* 10 (1999). DOI: 10.1137/S1052623497331063.

[21] Shruti J. *A survey of loss functions for semantic segmentation*. 2020. DOI: 10.1109. arXiv: 2006.14822. URL: https://doi.org/10.1109%5C%2Fcibcb48159.2020.9277638.

[22] Goyal P. Lin T. et al. *Focal Loss for Dense Object Detection*. 2017. DOI: 10.48550. arXiv: 1708.02002. URL: https://arxiv.org/abs/1708.02002.

[23] Kingma D.P. y Ba J. *Adam: A Method for Stochastic Optimization*. 2014. DOI: 10.48550/ARXIV.1412.698. arXiv: 1412.6980. URL: https://arxiv.org/abs/1412.6980.

[24] Hinton G. Srivastava N. et al. «Dropout: A Simple Way to Prevent Neural Networks from Overfitting». En: *The Journal of Machine Learning Research* 15.1 (2014), págs. 1929-1958.

[25] *Transfer Learning on Greyscale Images: How to Fine-Tune Pretrained Models on Black-and-White Datasets*. URL: https://towardsdatascience.com/transfer-learning-on-greyscale-images-how-to-fine-tune-pretrained-models-on-black-and-white-9a5150755c7a.

[26] *Image Segmentation with Python*. URL: https://www.sergilehkyi.com/es/image-segmentation-with-python/.

[27] Donahue J. Girshick R. et al. *Rich feature hierarchies for accurate object detection and semantic segmentation*. 2013. DOI: 10.48550. arXiv: 1311.2524. URL: https://arxiv.org/abs/1311.2524.

[28] *Precision and recall a simplified view*. URL: https://towardsdatascience.com/precision-and-recall-a-simplified-view-bc25978d81e6.

[29] *Understanding Confusion Matrix*. URL: https://towardsdatascience.com/understanding-confusion-matrix-a9ad42dcfd62.

[30] *Accuracy vs. F1-Score*. URL: https://medium.com/analytics-vidhya/accuracy-vs-f1-score-6258237beca2.

[31] *Confusion Matrix for Your Multi-Class Machine Learning Model*. URL: https://towardsdatascience.com/confusion-matrix-for-your-multi-class-machine-learning-model-ff9aa3bf7826.

[32] URL: https://github.com/martabarcena/ML-clasificacion-fundiciones.

[33] *TensorFlow*. URL: https://www.tensorflow.org/.

[34] *Keras*. URL: https://keras.io/.

[35] *MultiClass Image Classification*. URL: https://www.kaggle.com/code/prateek0x/multiclass-image-classification-using-keras/notebook.

[36] *MobileNet, MobileNetV2, and MobileNetV3*. URL: https://keras.io/api/applications/mobilenet/.

[37] Menglong Z. Howard A.G. et al. *MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications*. 2017. DOI: 10.48550. arXiv: 1704.04861. URL: https://arxiv.org/abs/1704.04861.

[38] Maire M. Tsung-Yi L. et al. *Microsoft COCO: Common Objects in Context*. 2014. DOI: 10.48550. arXiv: 1405.0312. URL: https://arxiv.org/abs/1405.0312.

[39] Fischer P. Ronneberger O. et al. *U-Net: Convolutional Networks for Biomedical Image Segmentation*. 2015. DOI: 10.48550. arXiv: 1505.04597. URL: https://arxiv.org/abs/1505.04597.

[40] *U-Net Architecture For Image Segmentation*. URL: https://blog.paperspace.com/unet-architecture-image-segmentation/.

[41] *ImageNet*. URL: https://www.image-net.org/.