UNIVERSIDAD
DE MÁLAGA

ESCUELA DE
INGENIERÍAS
INDUSTRIALES

ESCUELA DE INGENIERÍAS INDUSTRIALES

TRABAJO FIN DE GRADO

Departamento de Lenguajes y Ciencias de la Computación

Grado en Ingeniería en Tecnologías Industriales

Mención en Itinerario de Electrónica

Detección automática con redes neuronales profundas de artefactos en superficies fabricadas mediante manufactura aditiva

---

Automated inspection of artifact surfaces fabricated by additive manufacturing images by deep convolutional neural networks

---

Realizado por
Martín Loring Bueno

Tutorizado por
Ezequiel López Rubio
Iván Gómez Gallego

MÁLAGA, JUNIO DE 2023

Declaración de originalidad

Yo, don Martín Loring Bueno, estudiante del Grado de Ingeniería en Tecnologías Industriales en la Escuela de Ingenierías Industriales de la Universidad de Málaga, declaro:

Ser autor del Trabajo de Fin de Grado titulado **"Detección automática con redes neuronales profundas de artefactos en superficies fabricadas mediante manufactura aditiva"**, tutorizado por Dr. Ezequiel López Rubio y Dr. Iván Gómez Gallego, y que no ha sido presentado con anterioridad, ni total ni parcialmente, para superar materias previamente cursadas en esta u otras titulaciones de la Universidad de Málaga o cualquier otra institución de educación superior u otro tipo de fin.

Así mismo, declaro no haber trasgredido ninguna norma universitaria con respecto al plagio ni a las leyes establecidas que protegen la propiedad intelectual, así como que las fuentes utilizadas han sido citadas adecuadamente.

Málaga, a 6 de junio de 2023

Fdo.:

Martín Loring Bueno

Resumen

La impresión 3D, o manufactura aditiva, se ha convertido en una de las revoluciones más importantes para la ingeniería en los últimos años, llegando a ser un pilar fundamental en la denominada 4ª Revolución Industrial. Este tipo de fabricación, que se basa en la creación de objetos tridimensionales a partir de un modelo digital, no solo ha provocado un gran cambio en la industria, sino que ha permitido que usuarios de todo el mundo tengan acceso a la fabricación de objetos complejos de calidad.

El modelado por deposición fundida (FDM), también conocido como fabricación por filamento fundido (FFF), es el tipo de tecnología de manufactura aditiva actualmente más usado a nivel de usuario. Sin embargo, esta tecnología sufre excesivamente de una falta de control de calidad y regulación durante el proceso, lo que se tratará de solventar en este trabajo mediante el aprendizaje profundo.

El aprendizaje profundo, por su parte, se ha convertido en la técnica dentro de la inteligencia artificial con un mayor respaldo, al ser capaz de obtener resultados que han sido todo un hito en este campo. Gran parte de la responsabilidad del ingeniero al trabajar con estos recursos es ser capaz de encontrar un problema donde las cualidades del aprendizaje profundo puedan brindar soluciones imposibles por otros métodos.

En este contexto, se tratará de aplicar el aprendizaje profundo a la detección de artefactos, o defectos, en capas impresas por fabricación por filamento fundido. En concreto, se entrenará una red neuronal convolucional con un conjunto de datos formado por imágenes de figuras fabricadas por este método.

La mayor parte de este trabajo se centrará en experimentos referidos a este entrenamiento, en los que se variarán diferentes parámetros y se probaran diferentes modelos buscando la mejor respuesta posible. Todos los conceptos teóricos aplicados se explican en este texto, así como la motivación para cada decisión tomada en el proceso.

Finalmente, se encontrará un modelo óptimo que sea capaz de detectar estos defectos en capas de figuras fabricadas por filamento fundido, evidenciando de esta forma la competencia de las redes neuronales profundas en este campo y el posible futuro de modelos de este tipo.

Palabras clave: Inteligencia Artificial, Aprendizaje Profundo, Redes Neuronales Convolucionales, Visión por Computador, Manufactura aditiva, Modelado por deposición fundida (FDM), Post Procesamiento, Detección de Defectos, PyTorch.

Abstract

3D printing, or additive manufacturing, has become one of the most significant revolutions in engineering in recent years, becoming a fundamental pillar in the so-called 4th industrial revolution. This type of manufacturing, which involves creating three-dimensional objects from a digital model, has not only brought about a major change in the industry but has also allowed users from around the world to access the production of complex and high-quality objects.

Fused Deposition Modeling (FDM), also known as fused filament fabrication (FFF), is currently the most widely used additive manufacturing technology at the user level. However, this technology suffers from a lack of quality control and regulation during the process, which will be addressed in this work through deep learning.

Deep learning, on the other hand, has become the most supported artificial intelligence technique after achieving groundbreaking results in this field. A significant responsibility for engineers working with these resources is the ability to find problems where the qualities of deep learning can provide solutions that are otherwise impossible to achieve.

In this context, the aim is to apply deep learning for the detection of artifacts, or defects, in layers printed using Fused Filament Fabrication. Specifically, a convolutional neural network will be trained using a dataset consisting of images of objects manufactured using this method.

The majority of this work will focus on experiments related to this training, where different parameters will be varied, and different models will be tested in search of the best possible performance. All the applied theoretical concepts are explained in this text, as well as the motivation behind each decision made in the process.

Finally, an optimal model will be found that is capable of detecting these defects in layers of objects manufactured using fused filament fabrication, thus demonstrating the competence of deep neural networks in this field and the potential future of such models.

Keywords: Artificial Intelligence, Deep Learning, Convolutional Neural Networks, Computer Vision, Additive Manufacturing, Fused Deposition Modeling (FDM), Post Processing, Detection of Defects, PyTorch.

ÍNDICE GENERAL

ÍNDICE DE FIGURAS

CAPÍTULO 1

INTRODUCCIÓN

1.1. Motivación y contexto

La manufactura aditiva, también conocida como impresión 3D, es un proceso de fabricación que construye objetos capa por capa, utilizando información digital. La tecnología de impresión 3D se ha convertido en una herramienta popular para la fabricación de componentes de alta calidad en diversos campos, incluyendo la aeroespacial, la automotriz y la biomédica. Sin embargo, la naturaleza de este proceso de fabricación también puede generar ciertos problemas, como la aparición de artefactos o defectos en la superficie de los objetos fabricados.

La detección automática de estos artefactos o defectos es una tarea importante, ya que puede afectar la calidad y la durabilidad del producto final. En los procesos de fabricación tradicionales, los operarios son responsables de inspeccionar visualmente los objetos para detectar cualquier anomalía. Sin embargo, esta tarea puede ser tediosa y propensa a errores. Por lo tanto, es necesario encontrar soluciones más eficientes y precisas para la detección automática de artefactos.

En los últimos años, las redes neuronales profundas se han utilizado con éxito en la detección de objetos y en la visión por computadora en general. Las redes neuronales profundas son una clase de algoritmos de aprendizaje automático que imitan el funcionamiento del cerebro humano para procesar información y extraer características relevantes de los datos de entrada. En particular, las redes neuronales convolucionales han demostrado ser muy efectivas en la detección de objetos en imágenes.

En este trabajo, se propone el uso de redes neuronales profundas para la detección automática de artefactos en superficies fabricadas mediante manufactura aditiva. Se utilizará un conjunto de datos de superficies fabricadas con diferentes parámetros de impresión para entrenar y evaluar el modelo propuesto. El objetivo es diseñar un modelo de detección de artefactos preciso y eficiente que pueda ser utilizado en la fabricación aditiva para mejorar la calidad de los productos finales.

1.2. Objetivos y recursos

El objetivo principal de este proyecto es desarrollar un sistema de detección automática de artefactos en la superficie de las piezas fabricadas mediante manufactura aditiva utilizando redes neuronales profundas. Específicamente, el proyecto se centrará en la detección de cuatro tipos de artefactos: *Under* (Falta de material), *Over* (Sobra material), *Empty* (Zona vacía) y *Ok* (Zona con el material suficiente). Además, se explorarán diferentes técnicas y algoritmos de aprendizaje profundo para mejorar la precisión de la detección y se realizará una evaluación exhaustiva del rendimiento del sistema propuesto.

Para lograr este objetivo, se utilizarán los siguientes recursos:

- Python: el programa se ha realizado en su totalidad en el lenguaje de programación Python.
  - PyTorch: se trata de una biblioteca de aprendizaje automático, que posee gran cantidad de recursos para la Inteligencia Artificial (IA).
  - Matplotlib: biblioteca que permite crear gráficos a partir de datos contenidos en listas o arrays con extensión de NumPy.
  - Scikit-learn: biblioteca que ofrece recursos para IA.
  - Time, OS y Copy: bibliotecas empleadas para diversas soluciones en Python.
  - NumPy: biblioteca matemática que permite crear vectores y matrices, junto con un gran número de funciones matemáticas.
- Google Colaboratory: permite la realización de código Python y su ejecución gracias a las GPUs de Google.
- GitHub: todos los códigos creados se pueden encontrar en esta plataforma.

1.3. Metodología

En este apartado se indican los pasos seguidos por el alumno para la realización de este trabajo; tanto previos, como durante los experimentos, y tras estos.

1. Aprendizaje previo: se ha realizado un aprendizaje del lenguaje Python por parte del alumno, así como de la biblioteca de inteligencia artificial PyTorch y de conceptos de inteligencia artificial y redes neuronales.
2. Planteamiento de los objetivos: se ha encontrado un problema en el cual la inteligencia artificial podría encontrar soluciones mejores, como es la detección de artefactos de superficies fabricadas mediante manufactura aditiva.
3. Investigación: se ha realizado un amplio análisis de la manufactura aditiva y la fabricación por filamento fundido, así como de las redes neuronales convolucionales para clasificación de imágenes.
4. Realización de experimentos: se han realizado numerosos experimentos, probando el rendimiento de diferentes modelos y con diferentes parámetros aplicados, hasta llegar a un resultado considerado aceptable.

5. Documentación: se ha redactado este documento, donde se reflejan los conceptos relevantes a la manufacturación aditiva y la inteligencia artificial. Debido a los contenidos técnicos de este trabajo, muchos no relacionados con los grados de la escuela de ingenierías industriales, se ha dedicado una gran parte del documento a la explicación de conceptos necesarios para este trabajo. Se ha explicado también el *dataset* utilizado, así como las técnicas aplicadas en el código. Por último, se han reflejado los resultados que han llevado a un modelo óptimo mediante diferentes gráficas, y se ha concluido con un resumen y posibles mejoras futuras.

1.4. Estructura de la memoria

El proyecto se divide en los siguientes apartados para lograr la mayor compresión y contextualización de las técnicas empleadas:

1. Introducción: Se comenzará mediante la introducción de los conceptos principales utilizados en este proyecto: manufactura aditiva y aprendizaje profundo, se ofrecerá la motivación y el contexto en el que ocurre este trabajo, explicando las partes en que se divide la memoria, así como la metodología y recursos utilizados.
2. Estado del arte.
   - Manufactura aditiva: Se detallarán los conceptos necesarios para el entendimiento de la manufactura aditiva, así como su relevancia actual, centralizando en los problemas existentes para la fabricación por filamento fundido y posibles soluciones.
   - Inteligencia artificial: Se diferenciarán y explicarán conceptos como inteligencia artificial, aprendizaje automático y redes neuronales, indagando en las redes neuronales convolucionales.
3. Datasets: Se expondrá el conjunto de datos empleado.
4. Metodología: Se explicarán todos los conceptos y técnicas utilizadas durante el trabajo, de forma que los experimentos, decisiones y conclusiones tengan una mayor comprensión.
5. Desarrollo de la solución y resultados obtenidos: Se expondrán los resultados obtenidos para los diferentes parámetros y modelos, explicando los conceptos tras los resultados obtenidos y las medidas tomadas.
6. Conclusiones y líneas futuras: Se resumirán los resultados obtenidos y su importancia. Tras lo que se expondrán posibles mejoras, así como otros posibles proyectos que sigan la línea de investigación de este.
7. Bibliografía y Apéndices: Finalmente, se detallará en la bibliografía los documentos consultados durante el desarrollo del trabajo para la compresión de conceptos y redacción del código. En el apéndice se mostrará parte del código empleado, y se referenciará al resto de los códigos disponibles en GitHub.

CAPÍTULO 2

MANUFACTURA ADITIVA

2.1. Introducción

La manufactura aditiva se define, según la Organización Internacional de Normalización (ISO) [1], como "el término general para referirse a todas las tecnologías que utilizan el proceso de juntar materiales para construir objetos físicos basándose en datos de modelos 3D generados por ordenador, al contrario que las metodologías de fabricación sustractiva y formativa".

Este término, conocido coloquialmente como impresión 3D, abarca las tecnologías que fabrican objetos en tres dimensiones, añadiendo sucesivamente las capas que forman el mismo. Los objetos son modelados mediante planos virtuales realizados por diseño asistido por ordenador (CAD) o por software de modelado y animación, que la máquina utiliza como guía para la impresión. [2] [3]

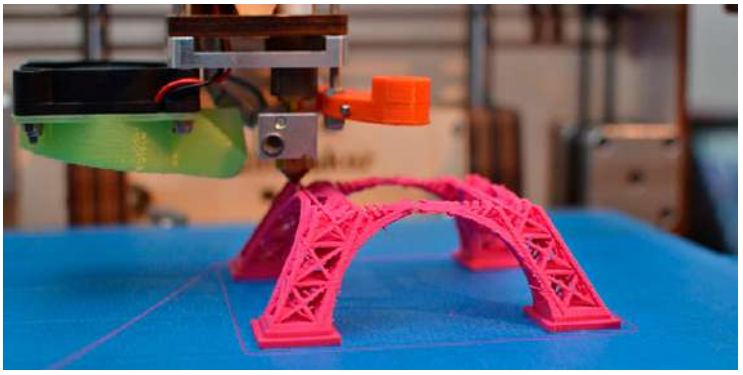


Figura 2.1: La manufactura aditiva o impresión 3D produce objetos capa por capa mediante modelos 3D generados por ordenador

Desde la introducción e implementación de la manufactura aditiva a mediados y finales de los 80 por parte de la empresa estadounidense 3D Systems [4], se han desarrollado numerosos avances. Esto ha permitido desarrollar diferentes técnicas y el uso de ellas en un amplio campo de modalidades, como son la industria automovilística [5] y la aeroespacial [6] o la medicina [7]. Asimismo, los avances más revolucionarios en los últimos años han traído resultados fascinantes, como la bioimpresión de diferentes tipos de tejidos y órganos [8], o la impresión 3D de aerogeles de grafeno, materiales con características óptimas para la ingeniería [9]. En la Figura 2.2 se puede encontrar otro ejemplo de las ventajas de esta tecnología fuera del ámbito industrial y académico. Se trata de un prototipo de balón sin necesidad de ser inflado, desarrollado por Wilson, proveedor oficial de la NBA [10].

Figura 2.2: Prototipo de balón utilizando manufactura aditiva [10]

La propia naturaleza de este tipo de fabricación conlleva una serie de ventajas frente a otros métodos tradicionales. La manufactura se puede llevar a cabo bajo demanda, sin depender de otras partes manufacturadas en otros lugares, ni existir la necesidad de un gran inventario. La producción se puede realizar en pequeños lotes, existiendo además un alto grado de personificación del producto, lo que es espacialmente relevante en campos como la medicina o la biología. La tecnología no necesita una gran inversión de recursos, generando pocos desechos, lo que facilita la posibilidad de manufactura entre usuarios no profesionales. [11]

Es considerada por gran parte de la comunidad investigadora como un avance revolucionario en el sector de la manufactura, al ser capaz de encontrar soluciones para problemas sin resolver por parte de las tecnologías convencionales [12]. Además, la impresión 3D es considerada uno de los componentes fundamentales de la 4ª Revolución Industrial o Industria 4.0, puesto que ambas conllevan autonomía e interconectividad entre empleados, máquinas, proveedores y consumidores. Por último, la impresión 3D supone un avance al ser un método de manufactura con mayor sostenibilidad, lo que podría mitigar los efectos medioambientales de la industrialización. [3]

2.2. Tipología

Entre las diferentes técnicas de manufactura aditiva, la Organización Internacional de Normalización (ISO), diferencia siete categorías principales [13]:

- Extrusión: en el cual se empuja un filamento de material termoplástico sólido a través de una boquilla calentada, derritiéndolo. Dentro de esta categoría, se encuentra el modelado por deposición fundida (FDM), también denominado fabricación con filamento fundido (FFF).
- Polimerización VAT: una resina de fotopolímero en una tina es curada selectivamente mediante una fuente de luz. Las dos formas más comunes de polimerización de depósitos son: SLA (estereolitografía) y DLP (Procesamiento de luz digital).
- Fusión en lecho de polvo: utiliza una fuente de calor (normalmente un láser) para sinterizar o fusionar las partículas de polvo atomizado. Algunos ejemplos son el sinterizado selectivo por láser (SLS) o la fusión por haz de electrones (EBM).
- Inyección: se inyectan capas de un fotopolímero líquido en una bandeja de impresión y las endurecen instantáneamente usando luz ultravioleta. Las principales tecnologías utilizadas son la inyección de material (MJ) o la DOD, en inglés *Drop on Demand* (gota a demanda).
- Inyección de aglutinante: conocida como *Binder Jetting* (BJ), en este proceso un cabezal de impresión a chorro se desplaza sobre un lecho de polvo, depositando selectivamente un líquido aglutinante, y repite este proceso hasta formar la pieza completa.
- Deposición de energía directa (DED): se introduce el material y se fusiona mediante una potente energía térmica aplicada al mismo tiempo que este se deposita. Normalmente, esta energía proviene de un láser o de un haz de electrones.
- Laminación: se apilan y laminan hojas de materiales mu finos para generar objetos. Estas capas se pueden fusionar utilizando diversos métodos, siendo el calor y el sonido los principales.

2.2.1. Modelado por deposición fundida (FDM)

Si bien el concepto de identificación de desperfectos desarrollado en este trabajo es aplicable a todos los tipos de manufactura aditiva, el conjunto de datos utilizado será de un proceso de fabricación utilizando FDM.

Como se expone en el apartado anterior, el modelado por deposición fundida (FDM) es un tipo de impresión 3D que utiliza la técnica de extrusión. Esta tecnología fue desarrollada y patentada por la empresa israelí-estadounidense Stratasys en 1989. En 2009, esta patente expiró, dando lugar al uso extendido de una tecnología muy similar conocida con el término de fabricación con filamento fundido (FFF). Tras esto, la impresión 3D ha sido accesible para una gran parte de la población mundial, provocando un gran interés mediático en 2014 [14]. Por esto, los términos de modelado por deposición fundida (FDM) y filamento fundido (FFF) se suelen intercalar y usar como sinónimos, si bien existe una pequeña diferencia entre estos, y es la falta de un entorno de impresión calentado en FFF. La cámara calentada de FDM ayuda a controlar la temperatura de la pieza y reducir las tensiones residuales en el producto terminado, mientras que la temperatura no controlada en las máquinas FFF hace que sus resultados sean menos precisos y más propensos a deformarse [15]. A lo largo de este texto, ambos términos se usarán con el mismo significado.

El modelado por deposición fundida (FDM), es la técnica más económica y puede usarse con una gran variedad de materiales, lo que le ha permitido ser la técnica más ampliamente utilizada y sobre la que existe una mayor investigación. Sin embargo, su principal obstáculo es su falta de precisión dimensional, al sufrir gravemente de una falta de evaluación de calidad en línea y ajuste de proceso [16].

La existencia de estas limitaciones en la impresión, junto con la gran comunidad creada en los últimos años de aficionados a esta tecnología, tanto a nivel de usuario como de investigadores, hace que este tipo de impresión sea óptima para el reconocimiento planteado en este trabajo.

Proceso de fabricación

El fenómeno físico que ocurre durante el modelado por deposición fundida (FDM) se puede dividir en tres fases [17]: el flujo de material pasa a través de la boquilla y se deposita en la cama de impresión o en otras capas de material ya depositado (extrusión), los gránulos depositados interaccionan creando una unión (fusión) y el material se enfría (solidificación).

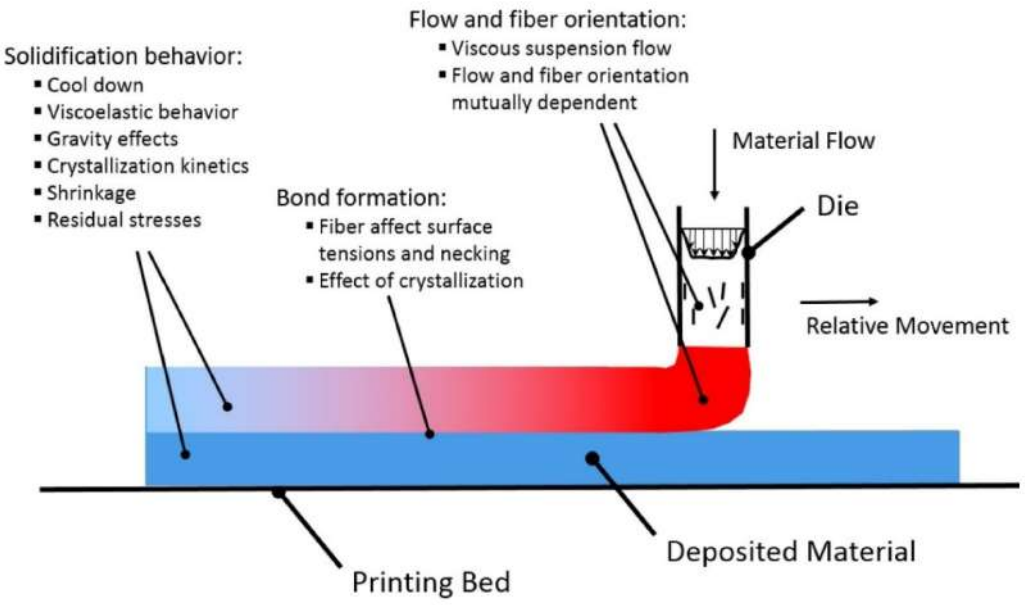


Figura 2.3: Diagrama esquemático del proceso de fabricación mediante FFF [17]

Limitaciones y mejoras

La principal desventaja del modelado FDM es su falta de precisión dimensional, al sufrir gravemente de una falta de evaluación de calidad en línea y ajuste de proceso. Esta falta de calidad en los acabados de las piezas puede ser solucionada con un postprocesamiento o acabado de las piezas.

El tipo de acabado necesario varía según el tipo de fabricación de la cual proceda la pieza. En el

caso del modelado por deposición fundida, es común el uso de lijado y el llenado de huecos [18]. La detección de defectos necesaria para tratar las piezas suele ser una tarea manual, que requiere una gran pérdida de tiempo y cuya complejidad puede provocar que el técnico que realiza este trabajo no lo haga con una gran efectividad.

Este trabajo propone solucionar esta cuestión utilizando la inteligencia artificial y, en concreto, mediante redes neuronales profundas (explicadas en el Apartado 3), que sean capaces de aprender a distinguir las zonas con defectos de un conjunto de imágenes. El conjunto de imágenes o *dataset*, que se explica en profundidad en el Apartado 4, está formado por imágenes de capas resultantes tras una impresión 3D. Estas imágenes han sido convertidas en imágenes 2D, convirtiendo la altura a diferentes colores. La red neuronal convolucional deberá ser capaz de distinguir las partes de la pieza que necesiten lijado o rellenado de huecos.

CAPÍTULO 3

INTELIGENCIA ARTIFICIAL (IA)

3.1. Breve contexto histórico

La inteligencia artificial (IA) es un campo interdisciplinario de la ciencia que se centra en la creación de sistemas que pueden realizar tareas que normalmente requerirían inteligencia humana, como el aprendizaje, la percepción, el razonamiento y la toma de decisiones. El término "'inteligencia artificial" fue acuñado por John McCarthy en 1956 durante la Conferencia de Dartmouth, donde se reunieron algunos de los primeros investigadores de IA [19].

Los inicios de la IA se remontan a la década de 1940, cuando se empezó a investigar cómo construir máquinas que pudieran realizar tareas que requerían inteligencia humana. En 1950 Alan Turing publicó el artículo *Computing Machinery and Intelligence* [20], en el que introducía el test de Turing para diferenciar un humano de una máquina. Uno de los primeros éxitos de la IA fue en este mismo año con el programa de ajedrez de Claude Shannon [21]. A partir de entonces, la IA experimentó un rápido crecimiento en la década de 1950 y 1960, con el desarrollo de los primeros programas de lógica simbólica y el descubrimiento del algoritmo de aprendizaje automático conocido como "perceptrón", considerada la primera red neuronal artificial [22]. En la Figura 3.1 se aprecia una un esquema perceptrón incluido en *"The Design of an Intelligent Automaton,"*, 1958 [23].

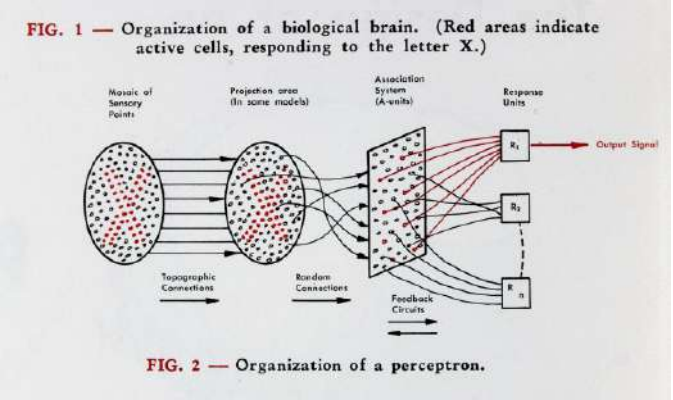


Figura 3.1: Esquema del perceptrón, primera red neuronal introducida por Frank Rosenblatt en 1958 [23]

Sin embargo, en la década de 1970 se produjo un ïnvierno de la IA", ya que los investigadores se dieron cuenta de que las técnicas existentes eran limitadas y no podían cumplir con las expectativas iniciales. Esta época fue especialmente provocada por la publicación del libro *"Perceptrons an introduction to computational geometry"* de Minsky y Papert [24], el cual explicaba los problemas para resolver los cálculos de las redes neuronales formadas por el "perceptrón". Durante este período, la financiación de la investigación en IA disminuyó significativamente.

En 1986, el interés a la inteligencia artificial vuelve tras la publicación del artículo *"Learning representations by back-propagating errors"* [25], en el cual se introduce el concepto de *back-propagation* como la técnica matemática que solucionaría los problemas de cálculo de las redes neuronales. En esta década, se introdujeron los "sistemas expertos", que agrupaban grandes datos de conocimiento para resolver diversos problemas como si se tratara de un humano. Sin embargo, la necesidad de un gran coste para el complejo hardware provoco un segundo "invierno" hasta entrados los 2000 [26].

El interés resurgió cuando en 1996, el ordenador de IBM Deep Blue venció al campeón del mundo de ajedrez Garry Kasparov [27], siendo televisado con gran expectación. Esta máquina utilizada "fuerza bruta" y no aprendía el juego como lo harían otras en el futuro.

En la década de los 2010, se produjo una gran revolución por parte de las redes neuronales, las cuales están formadas por neuronas, como ya imaginó Frank Rosenblatt. Estas redes son capaces de aprender en el sentido más humano de la palabra, a través de la exposición a un tipo de dato, extrayendo características por sí solas y clasificándolas. En 2012, Jeff Dean y Andrew Ng utilizaron los servidores de Google para construir una gran red neuronal que fue alimentada con 10 millones de imágenes de YouTube. La red fue programada para reconocer patrones y pudo extraer tres: una cara y un cuerpo humano, y un gato [28]. Más tarde, en 2016, los científicos de DeepMind consiguieron ganar al campeón del mundo del juego japonés de estrategia go con la IA AplhaGo [29]. Estos mismos científicos desarrollaron AlphaFold, otra IA capaz de resolver el problema de predecir la estructura 3D de una proteína a través de su secuencia de aminoácidos [30]. Este último supone un hito histórico que hizo ver la capacidad de la inteligencia artificial y el aprendizaje profundo en numerosos campos del conocimiento.

Figura 3.2: La red neuronal profunda AlphaGo gana 3-1 al campeón del mundo de go [31]

En la actualidad, la inteligencia artificial es un término de uso constante y cuya utilidad es aplicada en prácticamente todos los campos de la ciencia y la ingeniería. Con la reciente introducción de ChatGPT-3 en noviembre de 2022 por parte de la empresa OpenAI [32], la inteligencia artificial ha causado un gran ruido mediático, y un cambio revolucionario en la accesibilidad de estas herramientas al público general, siendo aún pronto para poder medir el impacto de estas herramientas en la sociedad.

3.2. Inteligencia artificial, *machine learning* y *deep learning*

Con el boom de la inteligencia artificial, es habitual escuchar mucha discusión al respecto y, en muchos casos, por parte de personas no entendidas en el campo. Esto provoca que términos que definen diferentes conceptos como inteligencia artificial, *machine learning*, o aprendizaje automático, y *deep learning*, o aprendizaje profundo, se usen como sinónimos y se intercambien.

En el apartado anterior se definió la inteligencia artificial como el campo interdisciplinario de la ciencia que se centra en la creación de sistemas que pueden realizar tareas que normalmente requerirían inteligencia humana, como el aprendizaje, la percepción, el razonamiento y la toma de decisiones. Es decir, que entrarían en este conjunto variedad de sistemas que realizan acciones inteligentes, como los que existen en robótica o aquellos que se utilizan para procesamiento del lenguaje.

El aprendizaje automático o *machine learning*, por otro lado, se trata de un campo dentro de la inteligencia artificial que se refiere al estudio de sistemas informáticos que aprenden y se adaptan automáticamente a partir de la experiencia sin ser programados explícitamente. Mientras que dentro de la inteligencia artificial pueden existir máquinas completamente programadas por parte de un humano para responder de una u otra forma, es decir, sin una verdadera capacidad para razonar o aprender, en el aprendizaje automático es el propio modelo el que, a través de un algoritmo, puede procesar grandes cantidades de datos y tomar la decisión por sí mismo. Un ejemplo de este campo sería la recomendación directa de música por parte de aplicaciones musicales como Spotify, que tras estudiar tus gustos y los de otras personas con gustos parecidos, es capaz de recomendar canciones que podrían gustar a la persona [33].

La técnica dentro del aprendizaje automático que ha tomado un papel protagonista en los últimos años por sus resultados son las redes neuronales o *neural networks*, las cuales son capaces de aprender de forma jerarquizadas, es decir, aprenden por niveles o capas. Las primeras capas, más cercanas a la información de entrada, aprenden conceptos básicos, como texturas, colores o formas. Por otro lado, las capas más cercanas a la salida aprenden conceptos más complejos al ir alejándose, hasta llegar a ser capaces de reconocer objetos altamente complicados, como coches, caras o animales [34].

Por último, el aprendizaje profundo o *deep learning* se trata de un subconjunto del aprendizaje automático formado por redes neuronales de 3 o más capas [35]. Mientras que los algoritmos de aprendizaje automático necesitan datos procesados anteriormente por humanos o la corrección ante el fallo, los algoritmos de aprendizaje profundo mejoran su aprendizaje mediante la repetición, sin intervención humana. Los algoritmos de aprendizaje profundo necesitan grandes cantidades de datos para poder realizar predicciones, pero a diferencia de otros aprendizajes automáticos, son capaces de "digerir" y procesar datos no estructurados, como texto e imágenes, y automatizan la extracción de características, eliminando parte de la dependencia de expertos humanos. Esta característica del *deep learning* será la que permita procesar las imágenes de este proyecto y realizar predicciones sobre otras.

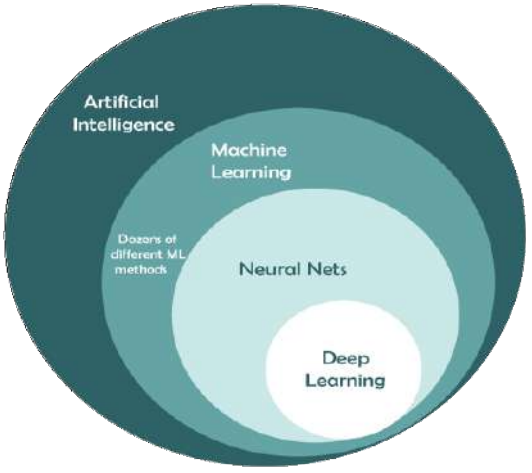


Figura 3.3: La inteligencia artificial, el aprendizaje automático y el aprendizaje profundo son distintos conceptos dentro de las ciencias de la computación

3.3. Redes neuronales

Como se explica en la Sección 3.1, las redes neuronales han permitido conseguir grandes logros a la inteligencia artificial, provocando que se sitúen como la el algoritmo más utilizado y con un mayor respaldo e investigación. A continuación, se abordan los principales conceptos que definen a una red neuronal, a una red neuronal profunda y a una red neuronal convolucional.

3.3.1. Neurona, funciones de activación, descenso del gradiente y backpropagation

Durante mediados del siglo XX, existía discusión entre los principales investigadores en el campo de la inteligencia artificial sobre si la forma de crear máquinas inteligentes sería con una estrategia *bottom-up* o *top-down*. Las primeras estrategias se basarían en recrear la biología humana de forma que a partir de neuronas o pequeñas unidades la máquina pudiera aprender, la segunda trataba de simular directamente el alto nivel de decisiones en que se basa el comportamiento humano.

Las redes neuronales, como su propio nombre indica, forman parte de los algoritmos *bottom-up*, puesto que están formadas por pequeñas unidades o **neuronas** que, al igual que en el sistema nervioso humano, se juntan para formar redes que definan su comportamiento.

La neurona artificial o unidad, que se puede ver en la Figura 3.4, es el bloque de construcción básico de todas las redes neuronales artificiales. Una unidad toma como entrada un vector x de D números reales y produce como salida un número real (un escalar) y. La unidad está parametrizada por un vector de pesos w de tamaño D y un término de sesgo (un escalar) b [36]. De esta forma, se aplica una función f a una entrada, resumiéndose este cálculo en la Ecuación 3.1.

$$y = f\left(b + \sum_{i=1}^{D} w_i x_i\right) = f\left(b + w^T x\right) \tag{3.1}$$

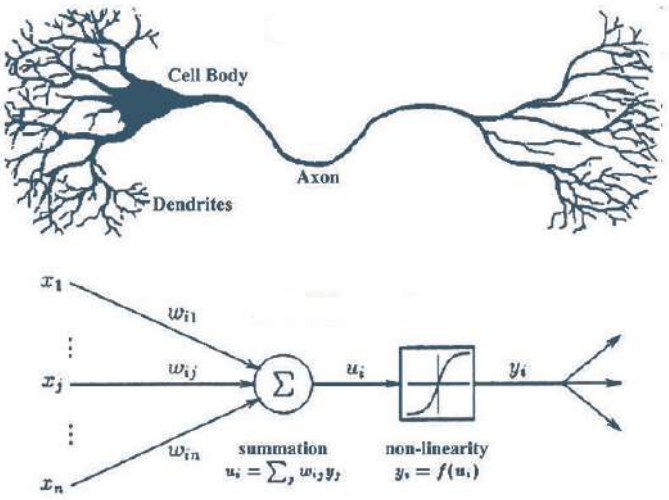


Figura 3.4: Diagrama de la arquitectura de una neurona artificial comparada con una neurona biológica

Las neuronas artificiales se juntan para formar **redes neuronales** artificiales, existiendo diversos tipos de redes según la forma en que estas neuronas forman conexiones. La forma más simple de conectar las neuronas es mediante el *feed forward* o prealimentación, en las cuales las redes neuronales forman capas secuenciales, existiendo un único flujo de información de las capas más cerca a la entrada hacia la salida. En la Figura 3.5 se aprecia una red neuronal de este tipo, existiendo capas de

entrada, de salida y capas intermedias, llamadas ocultas [36].

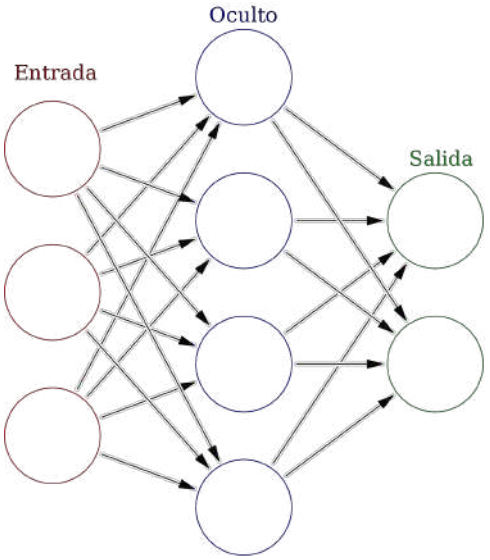


Figura 3.5: Diagrama de la arquitectura de una red neuronal

De esta forma, las redes neuronales están formadas por neuronas que aplican diferentes funciones a una entrada, dando una salida. Es importante hacer ver en este punto, que la suma de funciones lineales es una función lineal, por lo que si la función que aplican las neuronas a la entrada es lineal, una red neuronal formada por un gran número de neuronas se podría resumir en una única neurona aplicando una función lineal a la entrada y dando una salida. Dado que el objetivo de una red neuronal es conseguir resultados no lineales para problemas complejos, la función que se aplique a la entrada debe ser una función no lineal, siendo estas las **funciones de activación**. En la Figura 3.6 se aprecian las principales funciones de activación utilizadas: la sigmoide, RelU y la tangente [37].

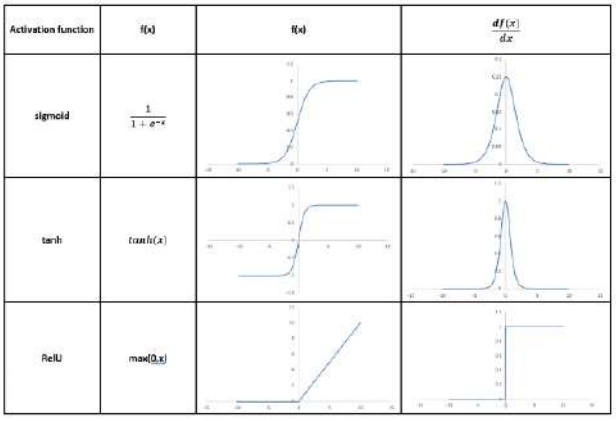


Figura 3.6: Principales funciones de activación utilizadas junto a sus derivadas [37]

Al hacer una recopilación de lo visto hasta hora, las redes neuronales son algoritmos formados

por unidades llamadas neuronas que aplican una función a la información de entrada, dando una salida. Estas neuronas tienen diferentes entradas y dan distinto peso o importancia a cada entrada, dando a su vez la red neuronal diferente peso a cada neurona. En este punto es donde se halla verdadero poder de una red neuronal, puesto que las redes neuronales "aprenden" al dar distintos pesos a las neuronas hasta llegar a la mejor solución. Para calcular qué pesos en cada neurona llevan a una mejor solución, se aplica un problema de optimización, buscando que modelo minimiza el error final, para lo que se utiliza el algoritmo del **descenso del gradiente**. En la Figura 3.7 se observa como para una función no lineal de dos parámetros, este algoritmo encuentra el mínimo en el error, siendo este el eje y. Es importante tener en cuenta que en una red neuronal las funciones están formadas por multitud de parámetros y no solo dos.

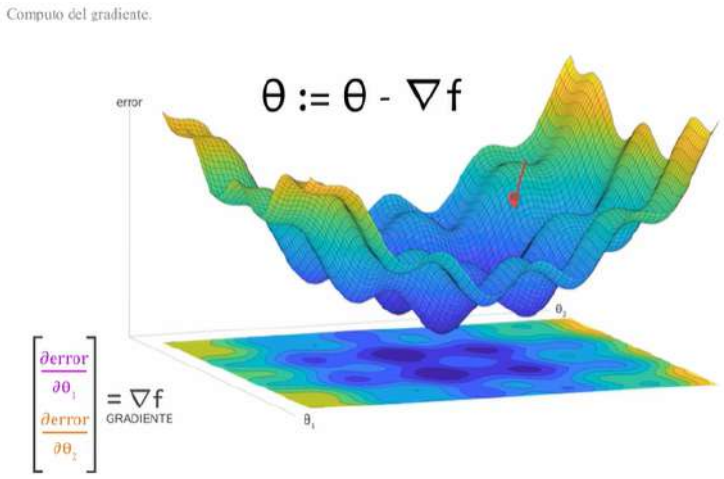
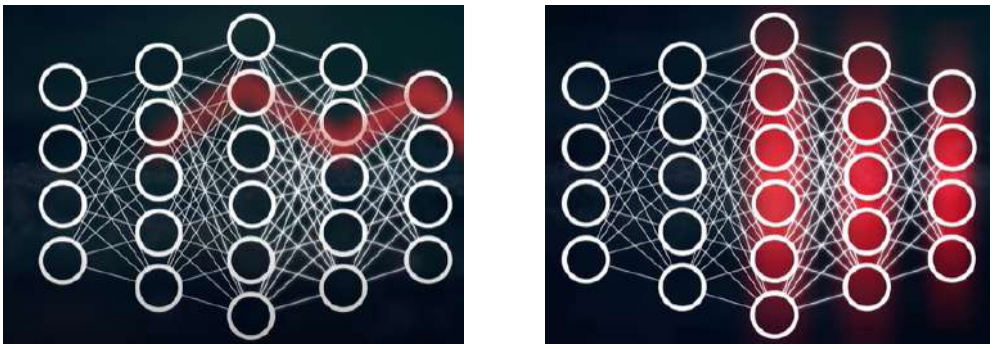


Figura 3.7: El algoritmo del descenso del gradiente calcula los parámetros para los que el modelo tiene un error mínimo [38]

En este punto, el cálculo del modelo que obtiene un error mínimo se complica, puesto que se hace necesario conocer como influye cada parámetro de cada neurona al error final, es decir, el aplicar el algoritmo del descenso del gradiente implica el cálculo de un conjunto muy alto de derivadas parciales altamente no triviales. Esta gran necesidad matemática fue la que provoco el primer invierno de la IA, hasta que el 1986 se introdujo el algoritmo del ***backpropagation*** o propagación hacia atrás [25]. Este algoritmo simplifica el cálculo de las derivadas parciales para el peso de una neurona concreta, puesto que toma en el cálculo la capa próxima como la última capa, sin necesidad de tomar en cuenta todas las neuronas que forman cada posible camino [39].

El proceso comienza calculando el error entre la salida final de la red neuronal y el valor deseado. Este error se propaga hacia atrás capa por capa, calculando las contribuciones de cada neurona y sus pesos. De esta forma, el cálculo de las derivadas asociadas al descenso del gradiente se simplifica y, gracias a la regla de la cadena, las derivadas solo tiene una capa de cálculo [37].

Gracias a estos conceptos, las redes neuronales actuales están formadas por cientos o miles de capas, por lo que se les considera con el término de **aprendizaje profundo** o ***deep learning***.



(a) Cálculo del gradiente sin *backpropagation*

(b) Cálculo del gradiente con *backpropagation*

Figura 3.8: Antes de existir el algoritmo del *backpropagation*, las pérdidas eran calculadas mediante algoritmos de "fuerza bruta" llamados perturbación aleatoria [39]

3.3.2. Redes neuronales convolucionales (CNN)

Dado que en este proyecto se centra en la clasificación de imágenes, se utilizaran las **redes neuronales convolucionales**, ya que son las principalmente utilizadas para la clasificación de objetos en imágenes naturales [40].

Capas de convolución

Mientras que una red neuronal simple o *vanilla* toma los píxeles de una imagen como valores independientes, una red neuronal convolucional saca partido de la posición espacial de los píxeles en la imagen. Esta pequeña diferencia supone un gran avance en la visión por ordenador, puesto que es de gran relevancia la posición espacial de los píxeles para comprender lo que la máquina está viendo [41]. Para poder captar esta información, las redes neuronales convolucionales cuentan con **capas convolucionales**, donde se realiza una **operación de convolución**.

Dado un vector A de entrada y un kernel K, una operación de convolución de una dimensión devuelve un vector B de la forma en que se aprecia en la ecuación 3.2, siendo la operación el símbolo $*$. En la ecuación 3.3 se aprecia la operación en dos dimensiones. En la Figura 3.9 se observa el ejemplo de una operación de convolución en dos dimensiones.

$$B(t) = (A * K)(t) = \sum_i A(t+i)K(i) \tag{3.2}$$

$$B(t) = (A * K)(t) = \sum_i A(t+i)K(i) \tag{3.3}$$

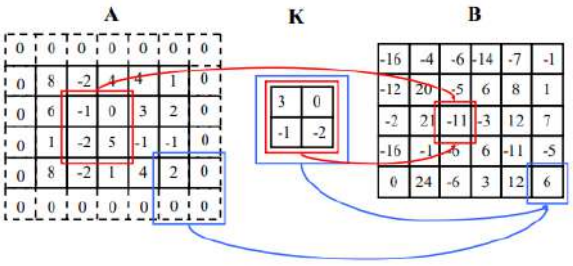
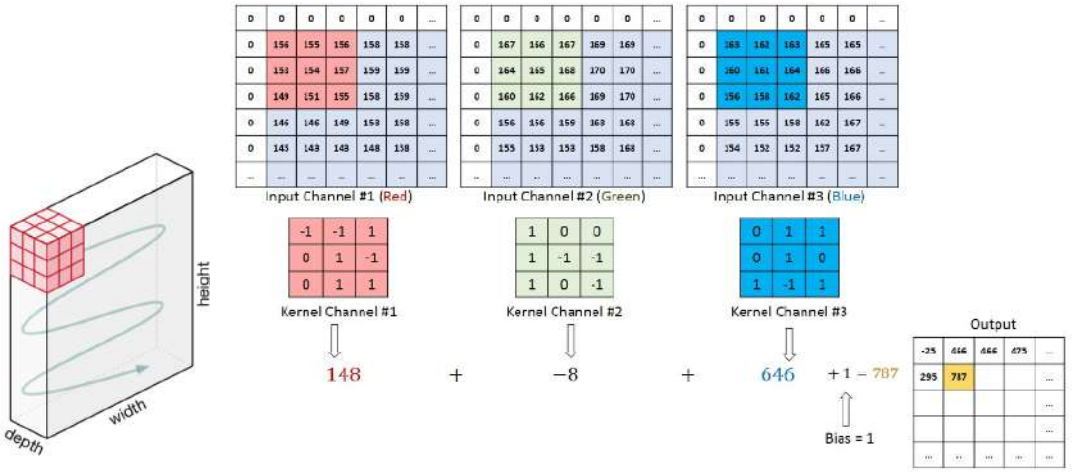


Figura 3.9: Ejemplo de operación de convolución 2D [40]

La entrada y la salida de la convolución tienen un tamaño parecido, mientras que el tamaño del kernel es bastante menor. El kernel recorre la entrada, dando como resultado valores grandes y positivos para secciones de entrada similares al propio kernel.

En el caso de imágenes del mundo real, estas están formadas por tres capas (rojo, verde y azul), por lo que la entrada tendrá un tamaño: largo (ej. 5 píxeles) x ancho (ej. 5 píxeles) x 3 (RGB), de forma que un kernel de 3x3x1 recorrerá la entrada, como aparece en la Figura 3.10. Como se comprueba en estos ejemplos de convolución, las matrices se completan con ceros con la técnica de rellenado o *valid padding* [42].

Los elementos del kernel K son los pesos de aprendizaje de las neuronas de la capa de convolución, seguidos de una matriz de activación. Cada canal aporta diferentes perspectivas de la imagen, dando importancia a diferentes cualidades significativas de la imagen.



(a) Diagrama convolución 3D

(b) Ejemplo cálculo convolución 3D

Figura 3.10: Las imágenes naturales están formadas por capas de colores y tamaño en píxeles [42]

Capas *pooling*

La operación *pooling* o sondeo se aplica a una sección de la entrada y produce un número real que represente esa sección. De esta forma, la operación reduce el tamaño de la entrada, disminuyendo la necesidad computacional, dado que las capas convolucionales mantienen el tamaño de la entrada, y extraen información de las características dominantes de la entrada. Esta operación es la realizada por las capas *pooling*, situadas tras las convolucionales.

Existen distintos tipos de *pooling*, siendo los principales *max pooling* y *average pooling*, representados con un ejemplo en la Figura 3.11. Normalmente, el método utilizado es *max pooling*, que además cumple la función de aislante ante el ruido [42].

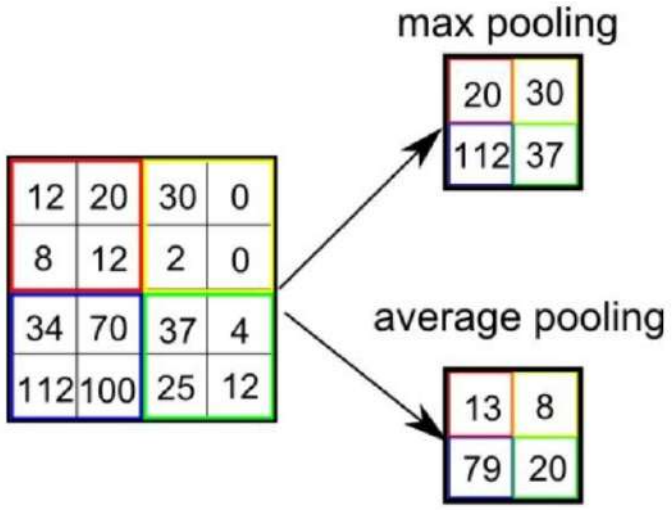


Figura 3.11: Ejemplo de operación de *max* y *average pooling* [42]

Capas *fully connected*

Tras las capas convolucionales y las *pooling*, la entrada se aplana y se alimenta a las capas ***fully connected*** o **completamente conectadas**. En estas capas, todas las neuronas están conectadas entre sí, lo que significa que cada neurona en una capa está conectada a todas las neuronas de la capa anterior. Esta estructura de conectividad completa permite que las capas totalmente conectadas realicen combinaciones lineales de las características de entrada y aprendan representaciones más complejas y abstractas.

Cada neurona en una capa totalmente conectada realiza una combinación lineal de las entradas que recibe, multiplicando cada entrada por su correspondiente peso y sumándolos junto con un término de sesgo (bias). Luego, se aplica una función de activación no lineal a la salida de la combinación lineal. Esta función de activación introduce no linealidad en la red y permite aprender relaciones no lineales entre las características.

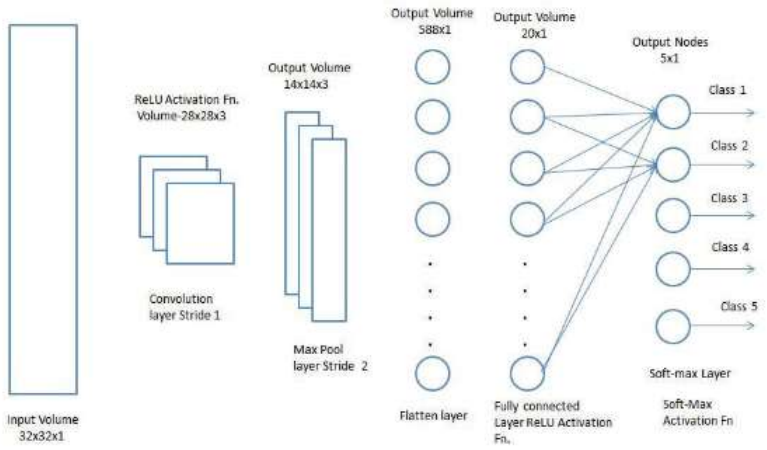


Figura 3.12: Tras las capas convolucionales y *pooling* las capas *fully connected* obtienen características más complejas y abstractas [42]

Capas *softmax*

Cuando las clases de clasificación del problema son más de dos, surge la necesidad de una capa que se ocupe de esta clasificación, siendo estas las **capas *softmax***.

Como se describe en la ecuación 3.4, la función de activación de las capas *softmax* es una generalización de la función logística. Esta función toma un vector x, cuyo tamaño es el número de clases C, diferente para cada unidad i, donde i es el índice de la clase.

$$f_i = \frac{exp(x_i)}{\sum_{j=1}^{C} exp(x_j)} \tag{3.4}$$

En la Figura 3.13 se demuestra un ejemplo de como las capas *softmax* generan probabilidades para cada clase, siendo la suma de estas 1.

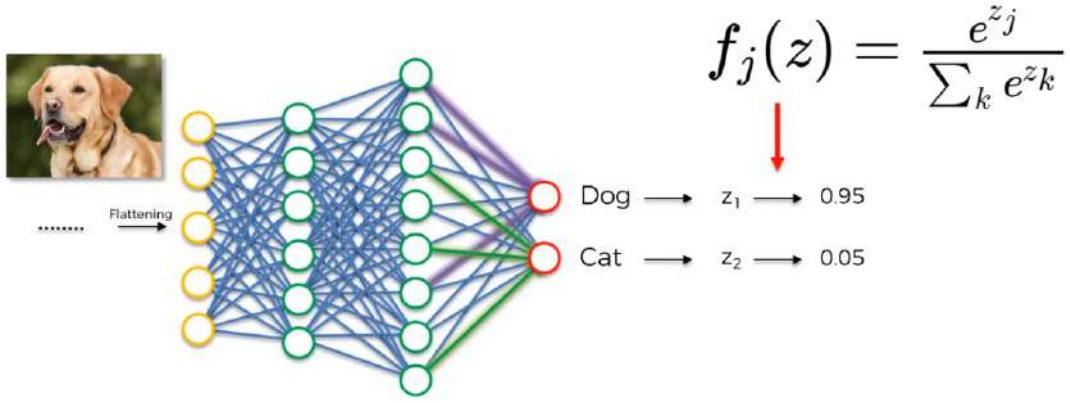


Figura 3.13: Diagrama resumen de la función de las capas *softmax* [43]

Red neuronal convolucional completa

Todas las capas que forman parte de una red neuronal convolucional se unen de manera secuencial, dando lugar a este tipo de red neuronal. Se puede observar en la Figura 3.14 un diagrama simplificado de una de estas redes, la cual es capaz de aprender las características de un tipo de clase y clasificar en esta todas las imágenes dentro de esta.

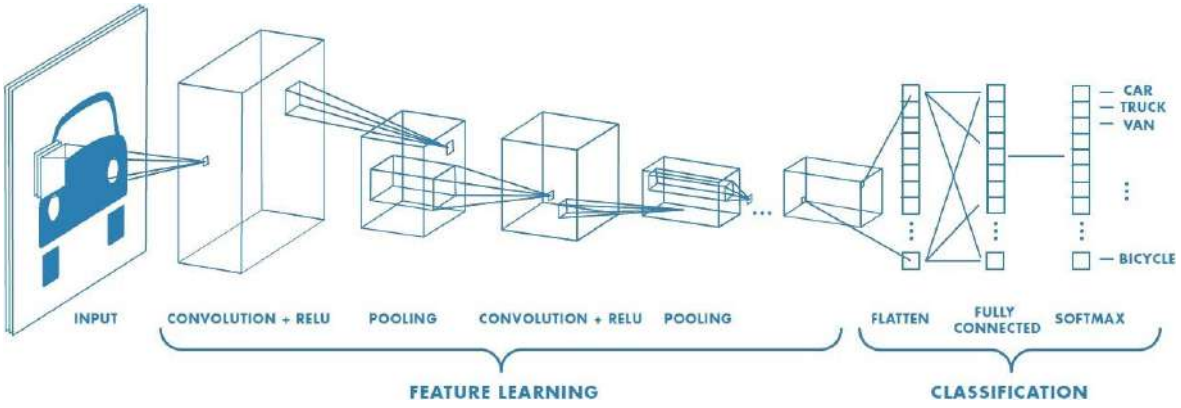


Figura 3.14: Diagrama resumen de una red CNN completa [42]

3.3.3. Conclusiones

La inteligencia artificial y su desarrollo son, sin duda, uno de los mayores hitos de la informática, la ciencia y la ingeniería en el comienzo del siglo XXI y en la historia de la humanidad. Su importancia es, además, que no solo es una tecnología aplicable a campos como la ciencia o la ingeniería, sino que ya es comprobable su uso en la vida diaria de cualquier persona.

En el punto en el que se encuentra la redacción de este texto, es aún pronto para conocer los posibles usos e influencia que llegaran a tener la inteligencia artificial y las redes neuronales. Sin embargo, la capacidad de usar el poder de las redes neuronales depende en gran parte de la capacidad innovadora para aplicarla a un problema que no tenga otra solución por parte de los ingenieros.

Este es el caso de este proyecto, en el que los problemas asociados a la impresión con filamento fundido, vistos en el Apartado 2, se pueden afrontar utilizando una inteligencia artificial y, en concreto, una red neuronal convolucional para el reconocimiento de las imágenes tras la impresión.

CAPÍTULO 4

CONJUNTO DE DATOS (DATASETS)

Para entrenar la red neuronal convolucional que detecte las distintas zonas en una pieza de impresión por manufactura aditiva, se ha utilizado un conjunto de imágenes desarrollado por el Instituto de Tecnología Stevens [16].

La obtención de los datos se llevó a cabo con un perfilador láser 2D de alta velocidad LJ-V7060 de Keyence equipado en lo alto de una máquina de impresión de fabricación por filamento fundido. Después de cada impresión de capa, el escáner láser obtiene la nube de puntos en 3D de la superficie superior. A continuación, la nube de puntos 3D se preprocesa para lograr la superficie objetivo de la pieza. Luego, la superficie superior se segmenta a partir de la nube de puntos y se convierte en una imagen de profundidad 2D, que serán las utilizadas para la clasificación. En la Figura 4.1 se presenta los recursos y su disposición para la obtención de las imágenes.

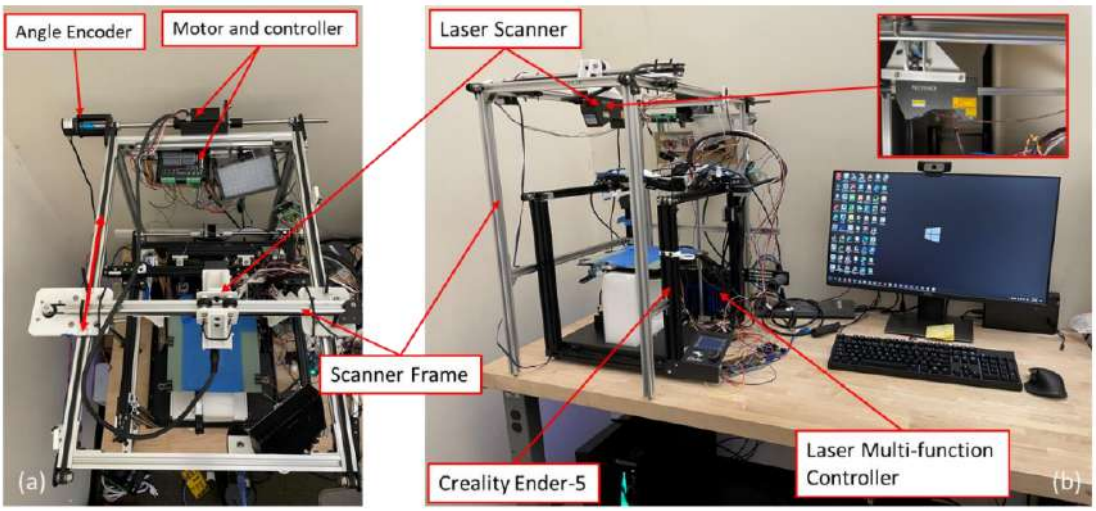


Figura 4.1: Configuración de los recursos para la obtención del conjunto de imágenes [16]

4.1. Descripción de los datos

Para la obtención del conjunto de datos se han impreso una serie de piezas y obtenido su representación 2D de profundidad, como se ve en el ejemplo de la Figura 4.2. Los colores verdes servirán para representar zonas de la pieza cuyo finalizado se puede considerar correcto. Las zonas de color azules no han tenido el suficiente material necesario, mientras que las zonas de color naranja, marrón y rojo cuentan con en exceso. Es importante también ver que la pieza puede tener, como en este caso, zonas blancas vacías que también se deben reconocer.

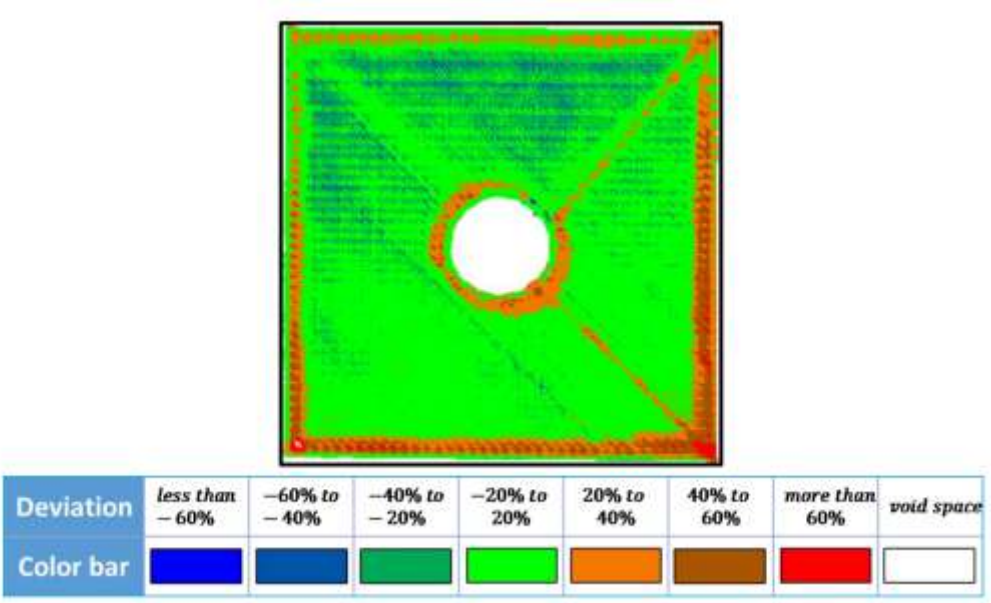


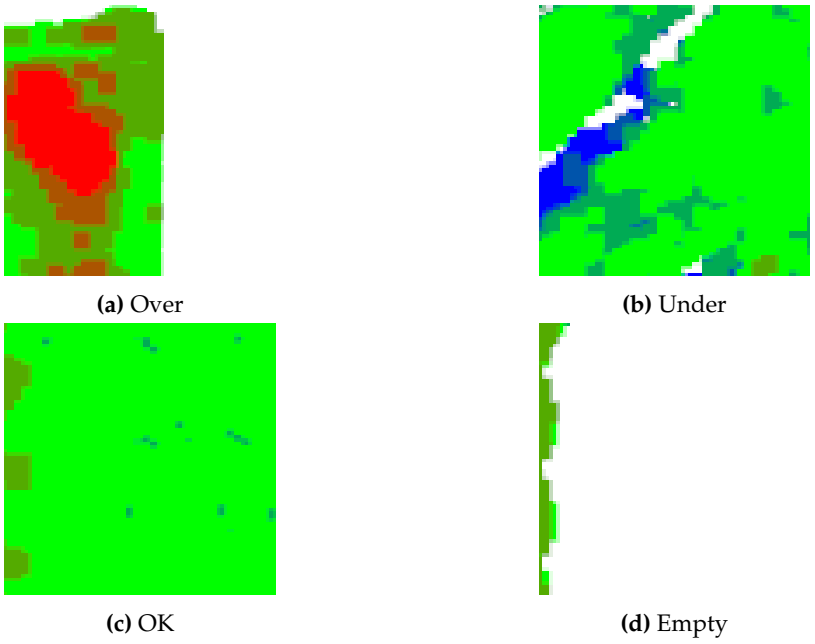
Figura 4.2: Imagen 2D para una pieza imprimida [16]

En la Figura 4.3 se observan las imágenes presentes en el *dataset*. Cada conjunto de datos del mapa de altura de una pieza ha sido dividido en una cuadrícula de 10x10 (100 segmentos en total), y cada segmento ha sido categorizado por profesionales en cuatro categorías: (a) Over: Situación de sobreimpresión, (b) Under: Situación de subimpresión, (c) OK: Situación de impresión normal, (d) Empty: Vacío.

Hay 434 escaneos de piezas, por lo que la división en 100 segmentos hace que el *dataset* esté formado por 43400 imágenes. En la Tabla 4.1 se encuentra resumida la clasificación de las imágenes.

Categoría	Nombre	Total
1	Over	12.268
2	OK	13.228
3	Under	14.726
4	Empty	3.179
		43.401

Tabla 4.1: Resumen del conjunto de datos



(a) Over **(b)** Under

(c) OK **(d)** Empty

Figura 4.3: Ejemplos de las cuatro clases del conjunto de datos [16]

CAPÍTULO 5

METODOLOGÍA

5.1. Python

Python se ha convertido en uno de los lenguajes de programación más utilizados a nivel mundial gracias, principalmente, a su sintaxis, al ser fácil de leer y asemejarse más al lenguaje natural, además de ser de código abierto, portable entre plataformas, y poseer amplias bibliotecas con una gran comunidad alrededor [44].

Estas ventajas han provocado que Python sea el principal lenguaje de programación para tareas complejas como son el *big data*, desarrollo web y de software, e inteligencia artificial. El propio creador de este lenguaje, Guido van Rossum, explica como en otros lenguajes la productividad se ve afectada si se emplea demasiado tiempo en la realización del código, dejándose de usar este tiempo para la comparación de soluciones [45].

De esta forma, el lenguaje Python, permitirá una sintaxis sencilla con la cual comparar resultados de diferentes entrenamientos con mayor facilidad.

5.2. PyTorch

PyTorch es una biblioteca para programas en Python que facilita la construcción de proyectos de aprendizaje profundo. Su énfasis se encuentra en la flexibilidad, permitiendo que los modelos de aprendizaje profundo se expresen en Python idiomático. Esta accesibilidad y facilidad de uso encontraron primeros usuarios en la comunidad de investigación, y en los años transcurridos desde su primer lanzamiento, se ha convertido en una de las herramientas de aprendizaje profundo más prominentes en una amplia gama de aplicaciones.

De la misma forma en que Python supone una introducción a la programación para personas inexpertas en este campo, debido a su simple sintaxis y legibilidad, PyTorch permite una fácil entra-

da al aprendizaje profundo y el entrenamiento de modelos para nuevos estudiantes de este campo (como el propio autor de este trabajo). [46]

5.2.1. Tensores

Los números de punto flotante son la forma en que una red maneja la información, por lo que se necesita una manera de codificar los datos del mundo real que se quieren procesar en algo digerible por una red, y luego decodificar la salida de nuevo en algo que una persona pueda entender y utilizar.

Para este fin, PyTorch introduce una estructura de datos fundamental: el tensor [46]. Este concepto de tensor no es el mismo que se utiliza en campos como matemáticas, física o ingeniería, donde el término tensor viene junto con la noción de espacios, sistemas de referencia y transformaciones entre ellos. En el contexto del aprendizaje profundo, los tensores se refieren a la generalización de vectores y matrices a un número arbitrario de dimensiones, como se puede apreciar en la Figura 5.1.

PyTorch no es la única biblioteca que se ocupa de matrices multidimensionales. NumPy es de lejos la biblioteca de matrices multidimensionales más popular, hasta el punto de que ahora se ha convertido en el lenguaje oficial de la ciencia de datos. PyTorch cuenta con una interoperabilidad perfecta con NumPy, lo que conlleva una integración de primera clase con el resto de las bibliotecas científicas en Python, como SciPy, Scikit-learn, y Pandas. Los tensores de PyTorch tienen algunas propiedades superiores a las matrices de NumPy, como la capacidad de realizar operaciones muy rápidas en unidades de procesamiento gráfico (GPU), distribuir operaciones en múltiples dispositivos o máquinas, y realizar un seguimiento del grafo de cálculos que los creó. Estas son características importantes cuando se implementa una biblioteca moderna de aprendizaje profundo.

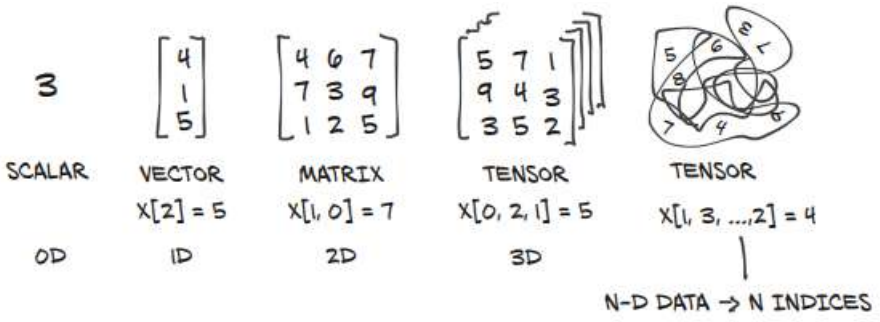


Figura 5.1: Boceto para el concepto de tensor en PyTorch [46]

5.3. Scikit-learn

Otras de las principales librerías de PyTorch que se utilizarán en este proyecto es Scikit-learn. Scikit-learn es una biblioteca para aprendizaje automático de código abierto para Python. Destaca por ser de gran ayuda en cuanto al procesamiento, la regresión, la clasificación y la selección de modelos. Es una de las librerías más populares de Python, ya que contiene gran variedad de algoritmos de aprendizaje automático y herramientas de procesamiento y análisis de datos. Está diseñada para

interoperar con las bibliotecas numéricas y científicas NumPy y SciPy. En este proyecto, se utilizará esta librería para la representación de datos, puesto que dispone de una sencilla interfaz para representar las curvas ROC y matrices de confusión [47].

5.4. Google Colab

El código en el que se basa este proyecto, se ha desarrollado a través de la herramienta gratuita de Google, Google Colab, la cual es un producto de Google Research. Este recurso permite a cualquier usuario escribir y ejecutar códigos de Python en el navegador, sin necesidad de instalar ningún software en su máquina. Es especialmente adecuado para tareas de aprendizaje automático, análisis de datos y educación.

Desde un punto de vista más técnico, Colab es un servicio de cuaderno alojado de Jupyter que no requiere configuración y que ofrece acceso sin coste adicional a recursos informáticos, como GPUs. El uso de esta herramienta permite la ejecución de código con coste computacional, como el de este proyecto, en tiempos reducidos en comparación con hacerlo en CPUs personales [48].

Figura 5.2: Python, PyTorch y Google Colab son los recursos principales para el desarrollo de este proyecto

5.5. *Transfer learning* o aprendizaje transferido

Transfer learning, o Aprendizaje por Transferencia, es una técnica de *Deep Learning* en la que se utiliza un modelo previamente entrenado para una tarea como punto de partida para entrenar otro modelo que realiza una tarea similar. Esta técnica facilita y reduce el tiempo utilizado en entrenamiento y se utiliza comúnmente en aplicaciones de reconocimiento de imágenes, detección de objetos o reconocimiento de voz, entre otras [49].

Las redes convolucionales son especialmente susceptibles a este tipo de técnicas, ya que las capas más bajas son independientes de la clasificación final, centrándose en diferenciar colores, esquinas, patrones o texturas. Son las últimas capas en una red neuronal las que se ocupan de clasificar y elegir el resultado de una imagen, especificando más en el problema del que se ocupa. Por ejemplo, si se

trata de un programa que decide si una imagen es un perro o un gato, las últimas capas se empezarán a fijar en las diferencias distintivas entre estos, como la cola, las orejas, o el hocico.

Existen diferentes estrategias basadas en el concepto de *transfer learning*. En este proyecto, se ha utilizado el *Fine tuning*, en el cual se mantienen la mayoría de pesos neuronales del modelo original, únicamente variando las capas finales necesarias para la nueva clasificación [50]

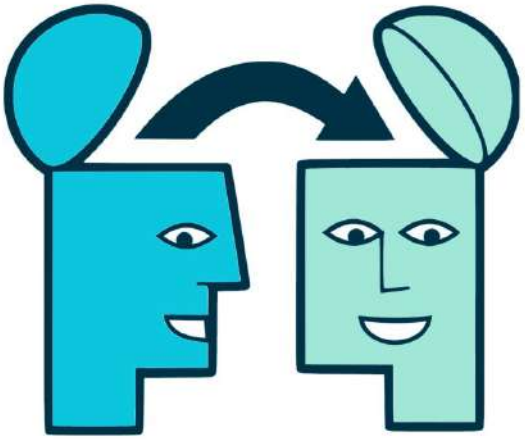


Figura 5.3: El *transfer learning* o aprendizaje transferido permite utilizar los conocimientos ya adquiridos por otro modelo

5.5.1. Principales modelos preentrenados

Se utilizarán seis de los principales modelos preentrenados que ofrece la librería PyTorch: AlexNet, VGG, ResNet, Inception, DenseNet, SqueezeNet. Todos ellos han sido entrenados con el conjunto de datos Imagenet, el cual está formado por millones de imágenes divididas en 1000 clases, por lo que se modificará el código para cambiar la clasificación a las 4 clases presentes en el conjunto de datos de este proyecto [51].

AlexNet

Esta red es conocida por ganar el Desafío de Reconocimiento Visual de Gran Escala ImageNet (ILSVRC)-2010 e ILSVRC 2012, con los mejores resultados reportados hasta la fecha, lo que impulsó el interés por el aprendizaje profundo para resolver este tipo de problemas [52]. En esta red, los autores utilizaron Unidades Lineales Rectificadas (ReLUs) como funciones de activación.

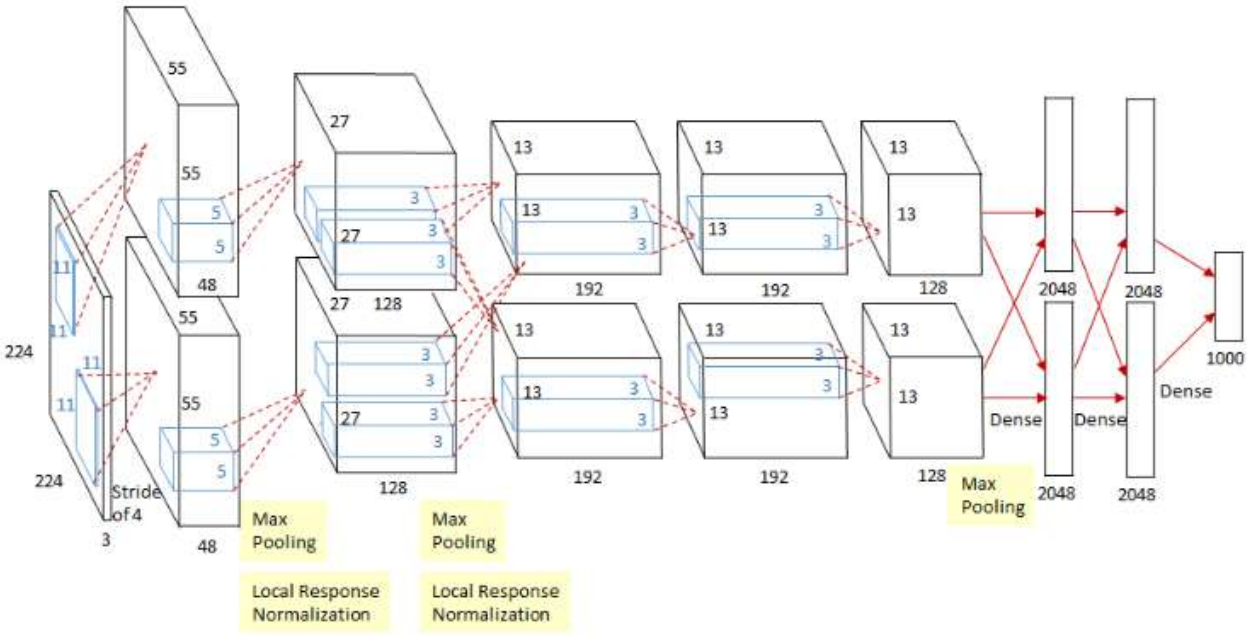


Figura 5.4: Arquitectura de la red AlexNet [52]

VGG

El grupo de geometría visual (VGG) de la Universidad de Oxford propuso redes CNN con 16 y 19 capas, conocidas popularmente como arquitecturas VGG-16 y VGG-19 [53]. El modelo VGG utiliza filtros convolucionales pequeños de 3x3 con *stride* de 1 píxel (por comparación, Alexnet tiene 11x11 con un *stride* de 4 píxeles), en lugar de un solo filtro grande, lo que mejora la función de decisión y acelera la convergencia del modelo. Además, el tamaño reducido del filtro reduce el sobreajuste del modelo durante el entrenamiento y permite capturar características espaciales de la imagen. La consistencia en el uso de filtros de 3x3 hace que el modelo sea fácil de manejar.

Se utilizará un modelo VGG11, el más pequeño de la familia disponible en PyTorch, siendo los otros de 13, 16 y 19 capas.

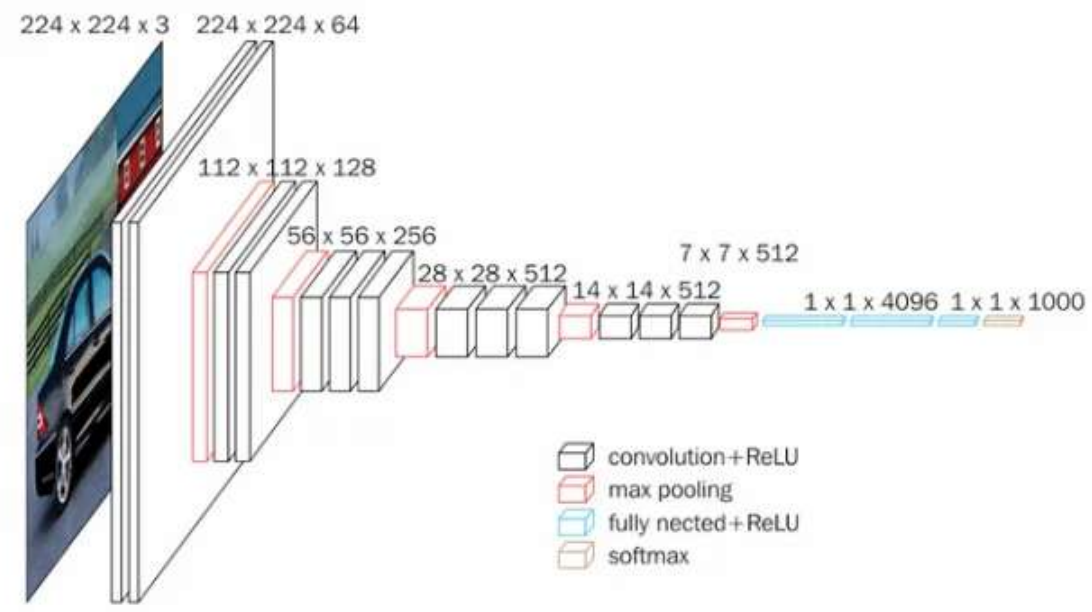


Figura 5.5: Arquitectura de la red VGG16 [40]

ResNet

El aumento de la profundidad de las redes, es decir, la adición de más capas a la red, como en VGG-16 y VGG-19, demostró que las redes podían aprender bien. Sin embargo, también expuso uno de los problemas importantes en el entrenamiento de redes más profundas: la degradación de la precisión del entrenamiento. Para superar este problema de degradación, se introdujo el marco de aprendizaje residual profundo [54]. Las redes residuales (ResNets) están inspiradas en las redes VGG, pero tienen menores complejidades.

Para este proyecto, se ha utilizado Resnet18, una red neuronal convolucional con 18 capas de profundidad, aunque existen otras mayores; Resnet34, Resnet50, Resnet101 y Resnet152.

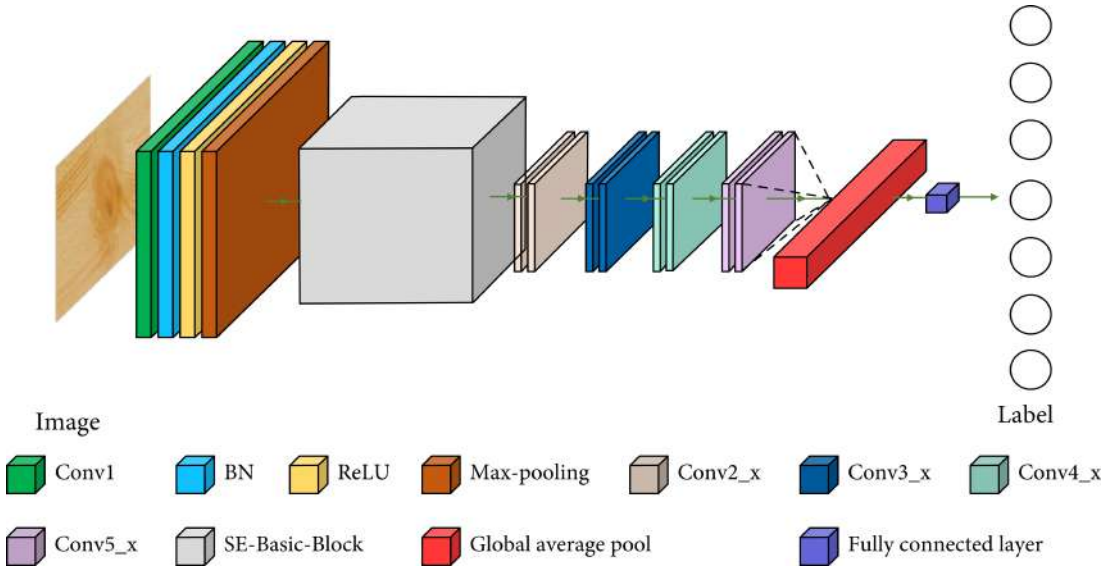


Figura 5.6: Arquitectura de la red ResNet [55]

Inception

Inception v3 fue descrita por primera vez en el artículo *"Rethinking the inception architecture for computer vision"* [56]. Esta red es única porque tiene dos capas de salida durante el entrenamiento. La segunda salida se conoce como una salida auxiliar y se encuentra en la parte de AuxLogits de la red. La salida principal es una capa lineal al final de la red. Es importante destacar que, durante las pruebas, solo consideramos la salida principal.

Se utilizará Inception-v3, la cual utiliza la idea de factorizar filtros espaciales más grandes en filtros más pequeños, y reemplazar un filtro convolucional espacial simétrico con múltiples filtros asimétricos. Inception-v3 tiene 42 capas y cuesta más del doble que GoogLeNet, pero menos que las redes VGG [57].

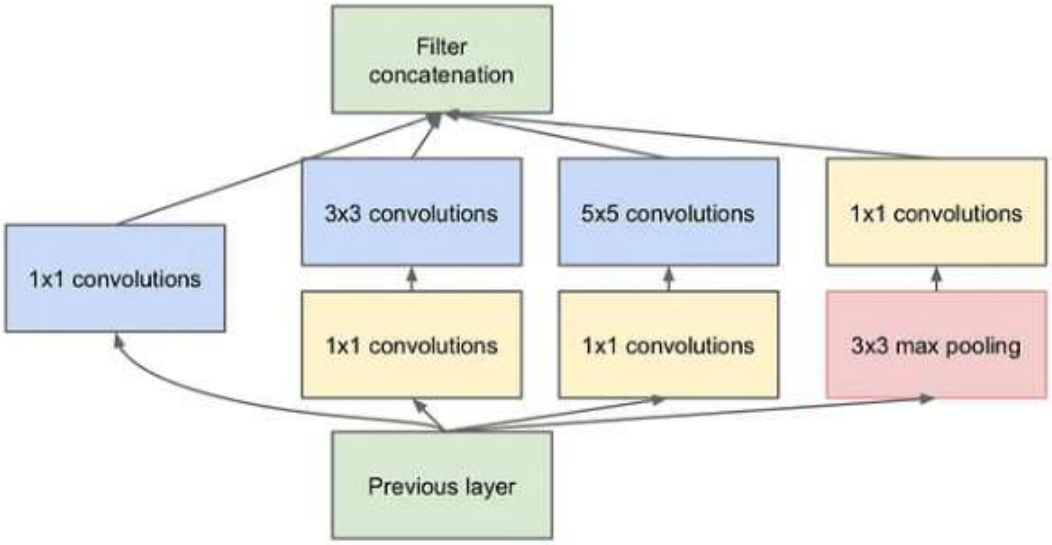


Figura 5.7: Arquitectura de la red Inception [56]

DenseNet

Las redes convolucionales tradicionales tienen L conexiones entre capas, mientras que DenseNet conecta cada capa con todas las demás capas en un enfoque de alimentación directa, lo que resulta en menos parámetros que las redes convolucionales tradicionales. Además, DenseNet combina características mediante concatenación, lo que permite un flujo mejorado de información y gradientes en toda la red, ayudando a entrenar redes más profundas. DenseNet también tiene efectos de regularización en conjuntos de entrenamiento pequeños, lo que reduce el sobreajuste durante el entrenamiento. Esta red fue introducida con el artículo *"Densely Connected Convolutional Networks"* [58].

PyTorch tiene 4 tipos de esta arquitectura, pero en este trabajo se utilizará DenseNet-121.

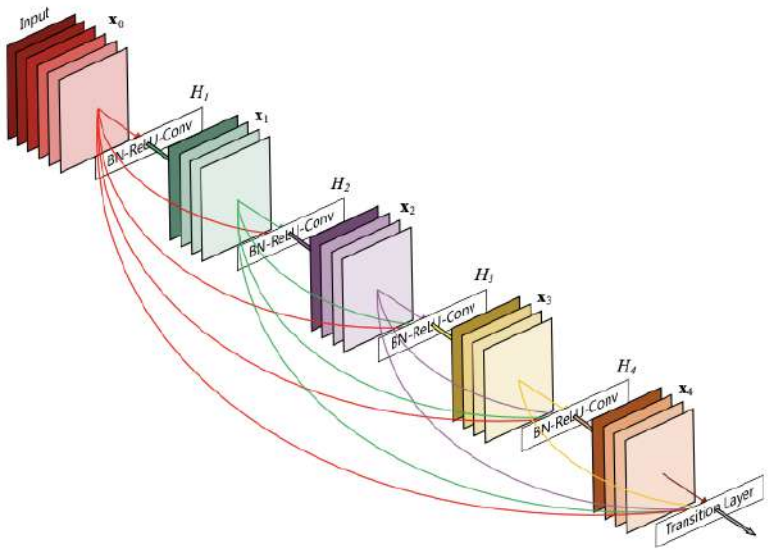


Figura 5.8: Arquitectura de la red DenseNet [58]

SqueezeNet

Mediante el artículo *"SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and ¡0.5MB model size"* [59] se pretendió crear una red que reemplazara los filtros de 3x3 por filtros de 1x1, disminuyera el número de canales de entrada a los filtros de 3x3 y realizará el submuestreo (downsampling) tarde en la red, de modo que las capas de convolución tengan mapas de activación grandes. De esta forma se crea, como indica el propio título del artículo, una red con mayor precisión y menos tamaño que la clásica AlexNet.

Comparación empírica de modelos preentrenados

Para una idea visual de las características de cada modelo, se muestra en la Figura 5.9 los resultados obtenidos para otro conjunto de datos publicados en un artículo [57], en el que también se utilizan algunos de los utilizados en este trabajo. Aunque el comportamiento para este conjunto de datos no tiene por qué ajustarse a estos resultados, se puede obtener una primera impresión de las cualidades de cada uno.

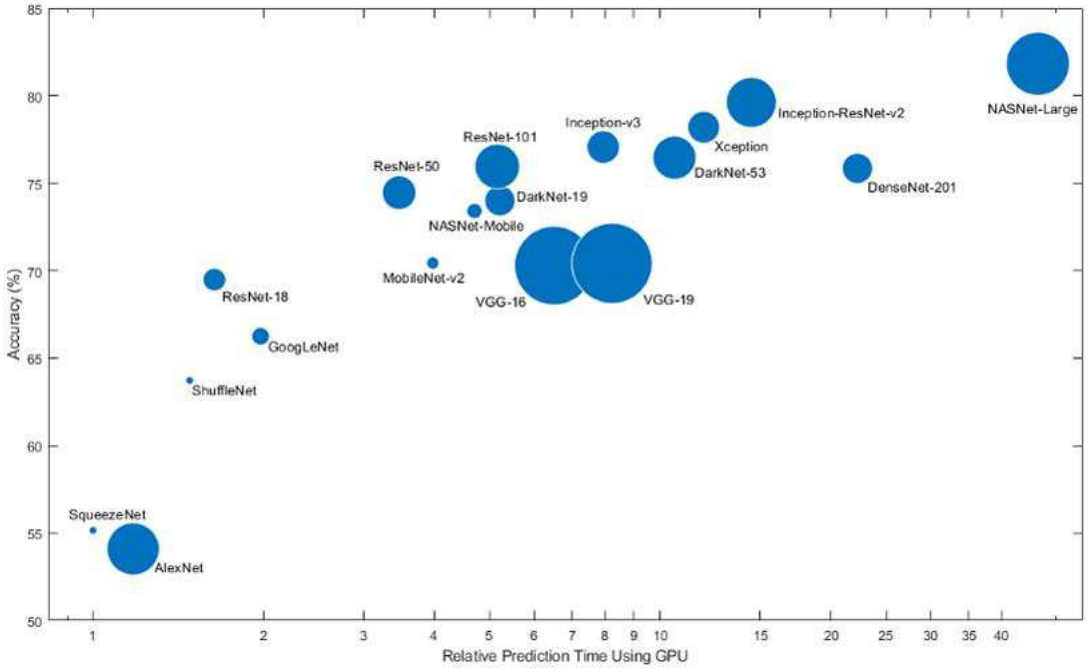


Figura 5.9: Comparación empírica de los diferentes modelos para el entrenamiento sobre un conjunto de datos concreto [57]

5.6. Aumentación de datos

La aumentación de datos, o *data augmentation* en inglés, es una técnica comúnmente utilizada en el campo de la inteligencia artificial para mejorar el rendimiento de los modelos de aprendizaje automático. Esta técnica se basa en generar nuevas muestras de datos a partir de los datos originales, pero con pequeñas modificaciones.

Las modificaciones que se aplican pueden ser rotaciones, zooms, desplazamientos o reflejos, entre otros, como se puede ver en la Figura 5.10. Al generar estas nuevas muestras de datos, se aumenta el tamaño del conjunto de entrenamiento y se evita el sobreajuste en los modelos de aprendizaje automático, lo que significa que el modelo aprende a generalizar mejor para nuevos datos en lugar de memorizar los datos de entrenamiento [40] [60].

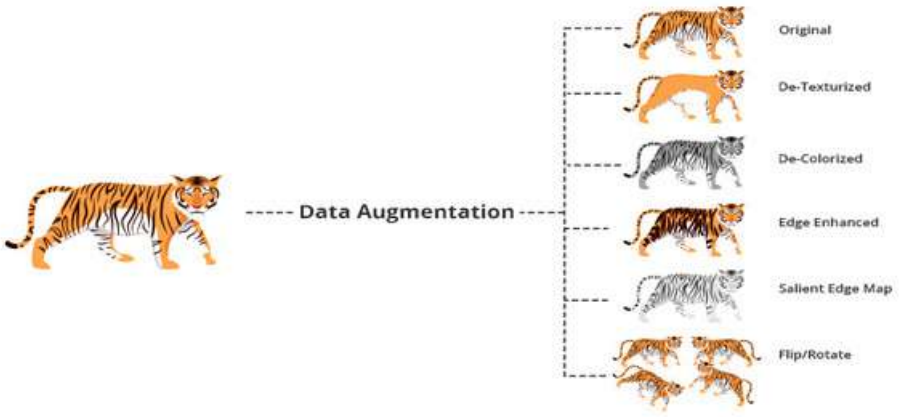


Figura 5.10: Visualización gráfica de algunos procesos utilizados para la aumentación de datos [60]

La aumentación de datos se aplica comúnmente en el procesamiento de imágenes, pero también se puede utilizar en otros tipos de datos, como texto y audio. Esta técnica ha demostrado ser efectiva para mejorar el rendimiento de los modelos de aprendizaje automático en diversas tareas, como la clasificación de imágenes, detección de objetos y reconocimiento de voz.

Aunque en este problema la amplia variedad de la muestra hace que no sea necesario aumentar el tamaño del dataset, se empleará esta técnica para evaluar si se mejora el rendimiento de nuestro modelo.

5.6.1. Transformaciones biblioteca PyTorch

La propia biblioteca de aprendizaje profundo PyTorch cuenta con transformaciones aplicables a imágenes para la aumentación del conjunto de datos. Algunos ejemplos de estas transformaciones se encuentran en la Figura 5.11. Las transformaciones utilizadas serán las siguientes:

- Resize: cambiar el tamaño de la imagen.
- ColorJitter: cambia el brillo, contraste, saturación y tono de una imagen.

- RandomRotation: se gira la imagen un número de grados.
- GaussianBlur: desenfoque gaussiano.

Figura 5.11: Ejemplos de transformaciones a imágenes proporcionadas por la biblioteca PyTorch [61]

5.6.2. Transformaciones biblioteca imgaug

Además de las transformaciones de la biblioteca PyTorch, se han utilizado las transformaciones de la biblioteca imgaug, la cual se utiliza únicamente para este tipo de operaciones, y cuenta con transformaciones algo más complejas. Además, las operaciones de esta biblioteca y las de PyTorch son compatibles y puede combinarse al utilizarse en las imágenes. Algunas de las transformaciones utilizadas serán:

- Sometimes(GaussianBlur(sigma=())): provoca un desenfoque gaussiano en algunas imágenes.
- Sometimes(Dropout()): provoca que en algunas imágenes haya píxeles eliminados.

En la Figura 5.12 se aprecian algunas de las complejas transformaciones que se pueden realizar fácilmente con la biblioteca imgaug.

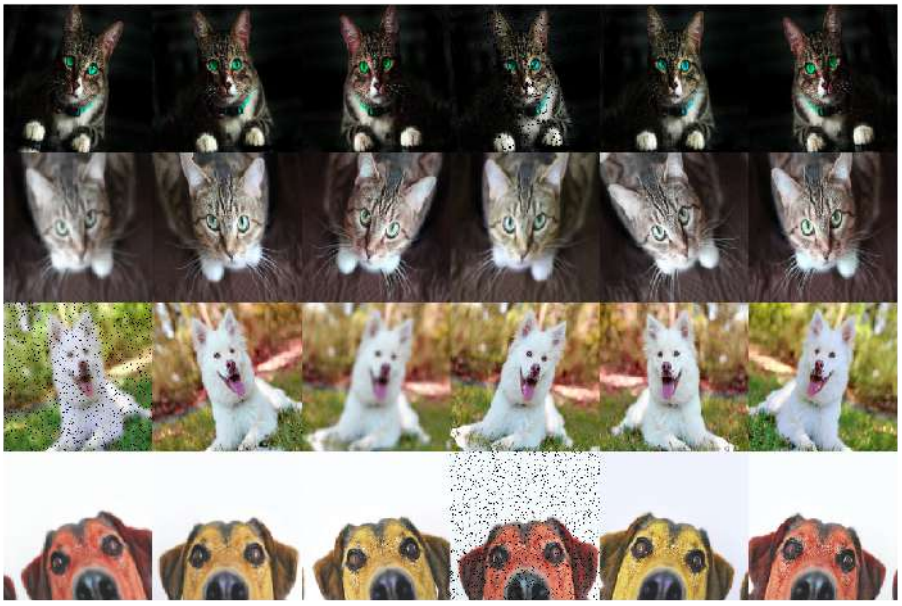


Figura 5.12: Ejemplos de transformaciones a imágenes proporcionadas por la biblioteca imgaug [61]

5.7. Hiperparámetros

Los hiperparámetros son parámetros cuyo valor controla el proceso de aprendizaje de un modelo y determinan cómo de bien un algoritmo de aprendizaje profundo llega a entrenarse sobre un conjunto de datos. Existen otros parámetros, como el peso de cada neurona, que se consideran simplemente parámetro, pero el prefijo "híper" implica que estos hiperparámetros están a alto nivel [62].

Estos valores son externos al modelo, puesto que estos no varían durante el entrenamiento del modelo, sino que es el propio diseñador del algoritmo quien debe estudiar y elegir los más adecuados.

En definitiva, cualquier parámetro que defina un modelo y que no se pueda cambiar durante el entrenamiento se considera un hiperparámetro. Ejemplos de esto son la tasa de aprendizaje de la función de optimización, el valor de la función de optimización, el número de épocas, el tamaño de dataset, y muchos otros.

Para este proyecto, el número de épocas y el tamaño del dataset serán lo primero que se definirá tras varios experimentos. Más tarde, se decidirá la mejor función de optimización y tasa de aprendizaje.

5.7.1. Función de optimización

Durante la sección 3.3.2 del apartado sobre inteligencia artificial, se explicaron las bases conceptuales y matemáticas de un modelo de red neuronal. En este, se explicó el método del descenso del gradiente como algoritmo de optimización que se utiliza para calcular los pesos de las neuronas. Existen, sin embargo, otros algoritmos de optimización, aunque el descenso del gradiente sea el más

utilizado.

Los **optimizadores** son algoritmos que se utilizan para actualizar los parámetros (pesos y bias) de las neuronas del modelo. Su objetivo es encontrar aquellos para los que el modelo tiene una mejor respuesta. Si se vuelve a tomar el ejemplo del algoritmo del descenso del gradiente, este está definido según la ecuación 5.1, donde x es un punto, α la tasa de aprendizaje y f la función a optimizar.

$$x_1 = x_0 - \alpha \nabla f(x_0) \tag{5.1}$$

Dentro de la optimización, las **funciones de pérdida** reflejan la diferencia entre la salida predicha por la red neuronal y el valor real de a salida. Existen diferentes técnicas para el cálculo de esta perdida, aunque en este proyecto tan solo se utilizará la función de **entropía cruzada** o *cross entropy loss*. Esta función de pérdida está definida según lo establecido en la ecuación 5.2, donde x es la entrada, w es su peso, y es el objetivo y C es el número de clases.

$$loss = -w_y \cdot log \frac{exp(x_y)}{\sum_{C=1}^{C} exp(x_C)} \tag{5.2}$$

Se probarán y compararán cuatro de los principales métodos de optimización, que se explican a continuación, siendo todos ellos funciones en PyTorch cuyos valores de entrada son los parámetros a optimizar y la tasa de aprendizaje.

Descenso del gradiente estocástico o *Stochastic gradient descent* (SGD)

Se trata de una versión sencilla de aplicación del descenso del gradiente. SGD solo estima el gradiente para la pérdida a partir de una pequeña submuestra de puntos de datos, que se eligen estocásticamente (al azar), lo que le permite ejecutarse mucho más rápido a través de las iteraciones. Teóricamente, la función de pérdida no se minimiza tan bien como con BGD (descenso de gradiente por lotes). Sin embargo, en la práctica, la aproximación cercana que se obtiene en SGD para los valores de los parámetros puede ser lo suficientemente buena en muchos casos. Además, los procesos estocásticos son una forma de regularización, por lo que las redes suelen generalizar mejor.

Algoritmo de Gradiente Adaptativo o *Adaptative Gradient Algorithm* (Adagrad)

Es una variación del algoritmo SGD, introducido en 2011 [63] en la que se utilizan distintas tasas de aprendizaje para cada variable, teniendo en cuenta el gradiente acumulado en cada iteración, haciendo que a aquellas que tienen un gradiente acumulado mayor se les aplica una tasa de aprendizaje inferior y viceversa.

Este algoritmo de optimización es uno de los más utilizados, aunque un posible problema con él se encuentra en que en ocasiones la tasa de aprendizaje para una variable decrece demasiado rápido debido a la acumulación de valores altos del gradiente al comenzar el entrenamiento, pudiendo provocar que el modelo no sea capaz de aproximarse al mínimo en la dimensión. [64].

Adadelta

Este algoritmo se introduce en 2012 por Matthew Zeiler [65]. El método Adadelta se adapta dinámicamente a lo largo del tiempo, utilizando solo información de primer orden y con un costo computacional mínimo más allá del descenso del gradiente estocástico estándar. El método no requiere ajustes manuales de una tasa de aprendizaje y es robusto frente al ruido en la información de gradiente, diferentes opciones de arquitectura de modelo, diversas modalidades de datos y selección de hiperparámetros.

Adadelta es una extensión más robusta de Adagrad que adapta las tasas de aprendizaje en función de una ventana móvil de actualizaciones de gradientes, en lugar de acumular todos los gradientes pasados. De esta manera, Adadelta continúa aprendiendo incluso cuando se han realizado muchas actualizaciones. En comparación con Adagrad, en la versión original de Adadelta no es necesario establecer una tasa de aprendizaje inicial. En esta versión, se puede establecer una tasa de aprendizaje inicial, al igual que en la mayoría de los otros.

Estimación Adaptativa de Momentos o *Adaptative movement estimation* (Adam)

Se introdujo en 2014 [66] como un algoritmo para la optimización de funciones objetivo estocásticas basado en estimaciones adaptativas de momentos de orden inferior. El método es fácil de implementar, eficiente computacionalmente, requiere poca memoria, es invariante a la reescala diagonal de los gradientes y es adecuado para problemas que son grandes en términos de datos y/o parámetros. El método también es apropiado para objetivos no estacionarios y problemas con gradientes muy ruidosos y/o dispersos.

La intuición detrás de este algoritmo es similar a la de SGD (descenso de gradiente estocástico). La diferencia principal es que los solucionadores Adam son notificadores adaptativos. Adam también ajusta la tasa de aprendizaje basándose en la magnitud de los gradientes utilizando la propagación de la media cuadrática de las raíces (RMSProp). Esto sigue una lógica similar al uso de momentum y amortiguamiento para SGD. Esto lo hace robusto para el paisaje de optimización no convexo de las redes neuronales [67].

5.7.2. Tasa de aprendizaje

Si la optimización consiste en encontrar un mínimo para una función concreta, se puede imaginar como si se buscará el lugar de menor altitud para una cordillera. Al hacer esto, se utiliza el gradiente para calcular cuanto se está bajando en cada dirección y encontrar el mínimo. En este símil, la tasa de aprendizaje o *learning rate* sería el tamaño de la zancada a la hora de desplazarse por la cordillera y buscar el mínimo, de forma que dependiendo de la función, un paso mayor o menor nos hará encontrarlo con mayor facilidad.

Este concepto se explica también en la Figura 5.13, donde una tasa de aprendizaje demasiado pequeña llegará al mínimo, pero lo hará demasiado lento, y una tasa muy grande no será capaz de encontrarlo.

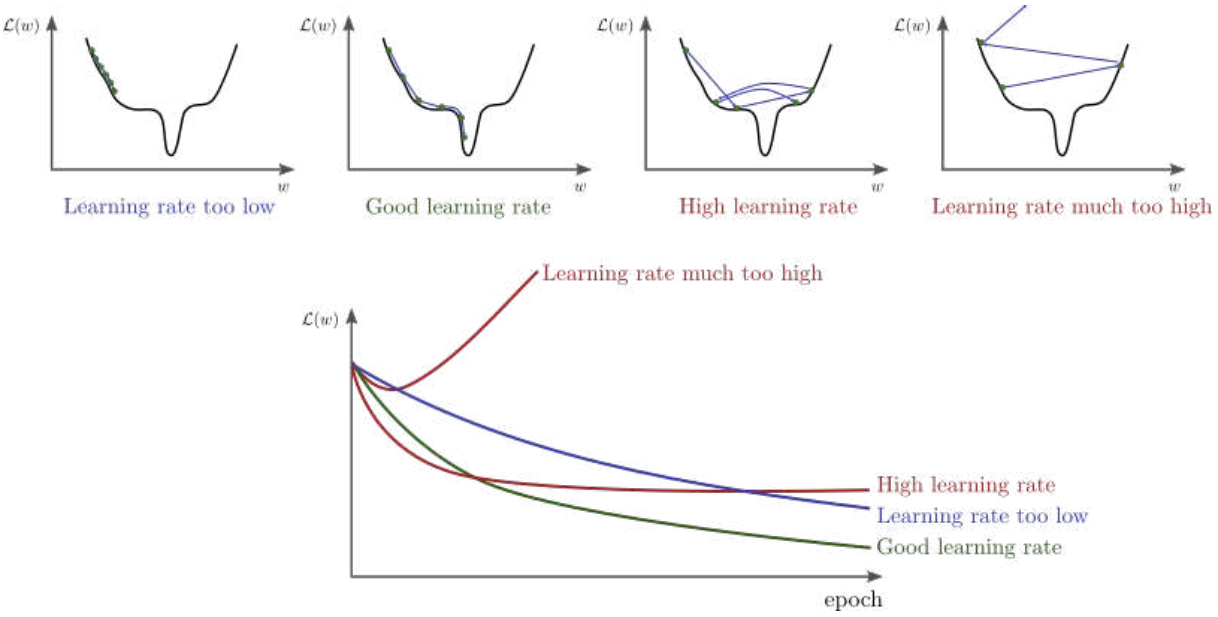


Figura 5.13: Una tasa de aprendizaje adecuada es fundamental para encontrar la solución óptima a cualquier problema [68]

Existen dos aspectos que optimizar con respecto a la tasa de aprendizaje:

- La tasa de aprendizaje inicial: marca cuál va a ser la velocidad global a la que se van a modificar los parámetros de la red. Sí, es demasiado pequeña, la red tarda demasiado en aprender. Si es demasiado grande, la red va saltando entre configuraciones muy alejadas entre sí, y no converge a una configuración óptima.
- La variación de la tasa de aprendizaje durante el entrenamiento: La tasa de aprendizaje se podría dejar constante (gamma=1), pero habitualmente esto no es lo ideal porque conforme se va avanzando en el entrenamiento se procura acercarse poco a poco a una configuración óptima. Ahora bien, la velocidad a la que decrece la tasa de aprendizaje también hay que determinarla experimentalmente. Si gamma es demasiado alto, la red tarda mucho en converger. Si gamma es demasiado bajo, es posible que la red se quede atascada en una configuración no óptima.

PyTorch cuanta con diversas funciones que permiten variar la tasa de aprendizaje mientras el modelo se entrena, de las cuales se utilizaran dos de las más usadas, que se exponen a continuación.

StepLR

Esta función se utiliza para reducir el valor de la tasa de aprendizaje cada cierto número de épocas. De esta forma, la función acepta dos parámetros principales:

- Tamaño del paso: se indica el número N de épocas que deben pasar para cada reducción de la tasa.
- γ: un factor multiplicativo por el cual la tasa de aprendizaje se reduce. Por ejemplo, si la tasa de aprendizaje es 1000 y gamma es 0.5, la nueva tasa de aprendizaje será 1000 x 0.5 = 500.

En la Figura 5.14 se representa un ejemplo de la función StepLR.

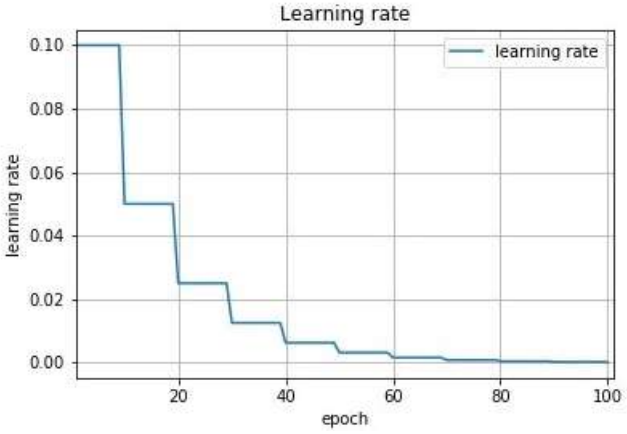


Figura 5.14: Ejemplo de función StepLR para reducción de la tasa de aprendizaje [69]

ExpotentialLR

Parecida a la anterior, esta función genera una bajada en la tasa de aprendizaje con el paso de las épocas, en este caso exponencialmente, como se representa en la Figura 5.15. Esta función tan solo necesita un término γ para reducir la tasa cada época, tal como indica la ecuación 5.3.

$$Tasa\,de\,aprendizaje_1 = Tasa\,de\,aprendizaje_0 \left(1 - \frac{\gamma}{100}\right)^{nepoca} \tag{5.3}$$

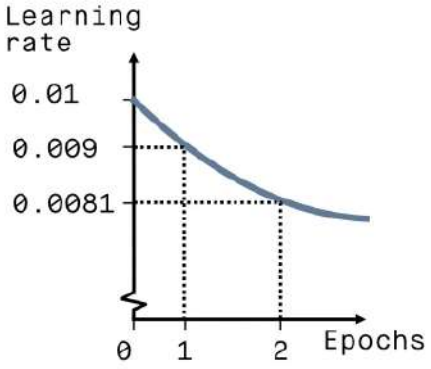


Figura 5.15: Ejemplo de función ExpotentialLR para reducción de la tasa de aprendizaje [70]

5.8. Optimización de hiperparámetros

Si bien en este proyecto se utilizará una optimización de los parámetros e hiperparámetros a partir de los resultados de los experimentos, existen otras técnicas para mejorar estos parámetros, muchas de ellas descubiertas en los últimos años y otras aún en desarrollo. Debido a la relevancia de estos conceptos en el campo de la inteligencia artificial, se ha considerado oportuno explicar estas ideas, aunque no se utilizarán empíricamente, existe la posibilidad de hacerlo en líneas futuras.

La elección de valores en la arquitectura de una red neuronal con base en intuición y un poco de prueba y error no es lo más apropiado, aun siendo lo más usado [71], ya que no ayuda a saber como de eficiente es realmente nuestro modelo y si existe alguna combinación de arquitectura que pueda ayudarnos a obtener resultados generalizados mejores [72]. Algunos de los hiperparámetros más relevantes son:

- Número de capas: un número muy alto puede causar problemas como sobreajuste, mientras que un número bajo puede hacer que el modelo tenga sesgo alto y un potencial bajo.
- Número de unidades ocultas por capa: encontrar un equilibrio entre sesgo alto y varianza.
- Función de activación: Las opciones populares son ReLU, Sigmoid y Tanh.
- Optimizador: es el algoritmo utilizado por el modelo para actualizar los pesos de cada capa después de cada iteración. Las opciones populares son SGD, RMSProp y Adam.
- Tasa de aprendizaje: es responsable de la característica central de aprendizaje y debe elegirse de tal manera que no sea demasiado alta, lo que dificultaría la convergencia al mínimo, ni demasiado baja, lo que dificultaría acelerar el proceso de aprendizaje.
- Dropout: consiste en eliminar ciertas conexiones en cada iteración, de modo que las unidades ocultas no dependan demasiado de ninguna característica en particular.
- Regularización L1/L2: actúa como otro regularizador en el que se controlan los valores de peso muy altos para evitar que el modelo dependa de una única característica

Además, para redes neuronales convolucionales, como la que se emplea en este proyecto, existen otros como:

- Tamaño del Kernel/Filtro: como se explicó en el apartado 3, un kernel es una matriz de pesos con la que se aplica la convolución a la entrada. Si se considera que se necesitan muchos píxeles para que la red reconozca el objeto, se usan filtros grandes (como 11x11 o 9x9). Si lo que diferencia a los objetos son algunas características pequeñas y locales, se usan filtros pequeños (3x3 o 5x5).
- Padding: se utiliza generalmente para agregar columnas y filas de ceros para mantener las dimensiones espaciales constantes después de la convolución. Hacer esto puede mejorar el rendimiento, ya que se conserva la información en los bordes.
- Stride: es el número de píxeles que se saltan al atravesar la entrada horizontal y verticalmente durante la convolución después de cada multiplicación de elementos de los pesos de entrada con los del filtro. Se utiliza para disminuir considerablemente el tamaño de la imagen de entrada.
- Parámetros de Capas de Pooling: las capas de pooling también tienen los mismos parámetros que una capa de convolución. La opción más comúnmente utilizada es el Max-Pooling.

Algunos de estos parámetros ya se han expuesto en secciones anteriores, como la tasa de aprendizaje o la función de optimación, y se trataran de optimizar mediante experimentos en la sección de resultados, aunque otros como el número de capas o el *dropout* no se trataran en este proyecto. La optimización de hiperparámetros o HPO (*Hyper-parameter optimization*) es un proceso sistemático que ayuda a encontrar los valores de hiperparámetros que permitan que el modelo ofrezca los mejores resultados.

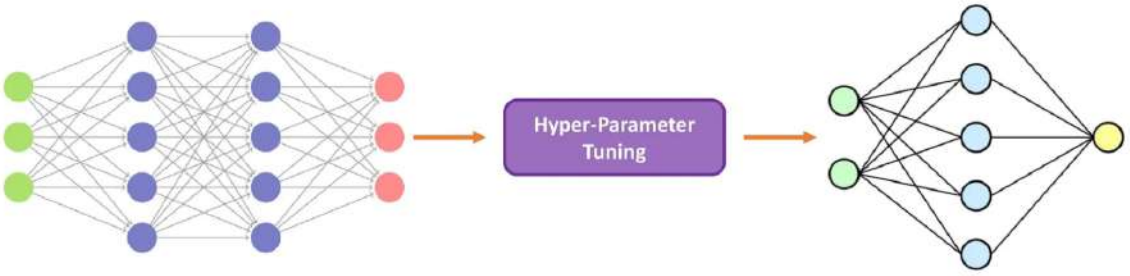


Figura 5.16: La optimización de hiperparámetros es un proceso sistemático que permite encontrar los mejores valores de hiperparámetros para el modelo [72]

Dentro de estos procesos, los principales son ***Grid search***, ***Ramdon search*** y **búsqueda por optimización bayesiana**.

- ***Grid search***: es una de las técnicas de optimización más sencillas, la cual funciona por medio de la comparación: se toman dos valores de un conjunto pequeño de valores para dos o más parámetros, se evalúan todas las combinaciones posibles y con ellas se forma una cuadrícula de valores (traducción de *grid search*) [73]. Los principales pasos del método se pueden resumir de la siguiente manera: 1. determinar un conjunto de valores de hiperparámetros, 2. combinar los valores seleccionados y crear un modelo para cada combinación, y 3. probar cada combinación utilizando una técnica de validación. La principal ventaja de la búsqueda en cuadrícula es la posibilidad de realizar las diferentes pruebas en paralelo.

  Una de sus principales limitaciones es que el método solo se puede utilizar en el caso de un espacio de hiperparámetros de baja dimensionalidad, es decir, 1-D, 2-D, etc. El método es lento para un mayor número de parámetros. Además, no se puede aplicar para la selección de modelos, ya que solo se puede utilizar para ajustar un solo modelo [74].

- ***Ramdon search***: se introdujo en 2012 [75] como una alternativa a la clásica técnica **grid search**. En este caso, los hiperparámetros se seleccionan al azar, de forma independiente de otras elecciones. El método es simple de implementar y es adecuado para funciones libres de gradiente. En comparación con *grid search*, este método converge más rápido, al buscar de manera efectiva en un espacio de hiperparámetros más grande y menos prometedor.

  Algunas de sus limitaciones es que es un método que consume mucho tiempo, ya que la evaluación de la función se vuelve costosa. El método no tiene la capacidad de selección de modelos, ya que solo se utiliza con un solo modelo. En comparación con la optimización bayesiana, este método no aprovecha el conocimiento de un espacio de búsqueda que funciona bien.

- **Búsqueda bayesiana**: introducida en 2012 [76], esta técnica se diferencia de la búsqueda aleatoria o en cuadrícula, en que llevan un registro de los resultados de evaluación pasados que utilizan para formar un modelo probabilístico que relaciona los hiperparámetros con la probabilidad de obtener un puntaje en la función objetivo [77].

  En resumen, el proceso de optimización bayesiano se resume en:

  1. Construir un modelo de probabilidad sustituto de la función objetivo. 2. Encontrar los hiperparámetros que funcionen mejor en el modelo sustituto. 3. Aplicar estos hiperparámetros a la verdadera función objetivo. 4. Actualizar el modelo sustituto incorporando los nuevos resultados. 5. Repetir los pasos 2-4 hasta alcanzar el número máximo de iteraciones o el tiempo establecido.

  A nivel general, los métodos de optimización bayesianos son eficientes porque eligen los próximos hiperparámetros de manera informada. La idea básica es invertir un poco más de tiempo en seleccionar los próximos hiperparámetros para realizar menos llamadas a la función objetivo. En la práctica, el tiempo invertido en seleccionar los próximos hiperparámetros es insignificante en comparación con el tiempo dedicado a la función objetivo. Al evaluar hiperparámetros que parecen más prometedores según los resultados pasados, los métodos bayesianos pueden encontrar mejores configuraciones de modelo que la búsqueda aleatoria en menos iteraciones.

  Los métodos basados en modelos bayesianos pueden encontrar mejores hiperparámetros en menos tiempo porque razonan acerca del mejor conjunto de hiperparámetros a evaluar según las pruebas pasadas.

- Existen otras técnicas para la optimización de hiperparámetros [78] [79], aunque estas son las más utilizadas. Cabe destacar también que el mejor tipo de optimización para el entrenamiento de un modelo concreto sobre un conjunto de datos, dependerá de las características del problema, eligiendo uno u otro viendo que características son más adecuadas [80].

Para el uso de estas técnicas en Python, existen bibliotecas diseñadas para su implementación, como Optuna o Hyperopt [81].

En la Figura 5.17 se ilustra como se genera el espacio de hiperparámetros (sobre dos hiperparámetros) mediante diferentes esquemas de búsqueda. Los colores de los puntos, desde negro a "amarillento" a naranja, indica el orden de la búsqueda. En *grid search*, los valores de los hiperparámetros están predefinidos sistemáticamente. En *random search*, el espacio de hiperparámetros se crea al azar. En la Optimización Bayesiana, los valores de los hiperparámetros elegidos se optimizan progresivamente para aproximar un mínimo. En este ejemplo ilustrativo, el hiperparámetro 1 tiene una influencia mayor que el hiperparámetro 2, hecho que se refleja en la profundidad de los mínimos de la función de pérdida.

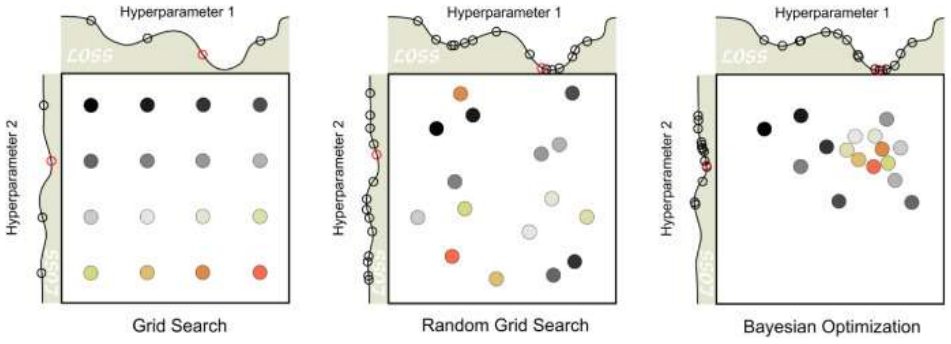


Figura 5.17: Resultados de la optimización de hiperparámetros con las principales técnicas [82]

5.9. Validación

Entre los conceptos explicados hasta este punto, falta uno de los detalles fundamentales relacionados con el entrenamiento de modelos basados en redes neuronales. Si bien se han explicado diferentes técnicas y métodos para el entrenamiento de un conjunto de datos, no se han introducido los diferentes grupos en los que se divide un *datase*: entrenamiento (*training*), validación (*validation*) y prueba (*test*).

El grupo de entrenamiento es el más sencillo, puesto que tan solo se trata del conjunto de datos que se utiliza para que el modelo aprenda las características de cada imagen. Por otro lado, para comprobar el rendimiento del modelo, no podemos hacerlo con las mismas imágenes con las que se está entrenando, puesto que el modelo podría memorizar estas y, aunque ofreciera una alta precisión, no sería una precisión real. Es por esto que para hacer el *feedback* o realimentación que nos indique como está funcionando nuestro modelo, se crea otro grupo con imágenes que no se utilicen durante el entrenamiento, llamado conjunto de validación.

Si bien con el conjunto de validación se pueden definir los parámetros que hagan que el modelo aprenda mejor, el tener buenos resultados ante la validación no asegurara que el modelo tenga un buen rendimiento ante otras imágenes del mundo real, que es para lo que un modelo debe servir. Es por esto que existe un grupo de prueba con el que se comprueba el modelo una vez ha sido configurado con los conjuntos de entrenamiento y validación, siendo este el conjunto de prueba, *testing* o testeo.

En la Figura 5.18 se resumen estos tres conjuntos y su función.

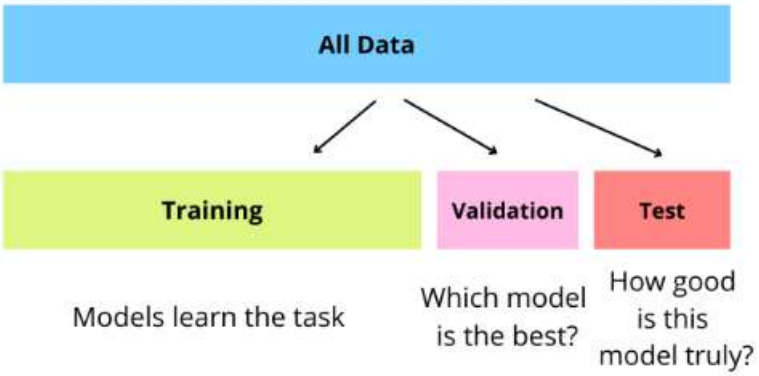


Figura 5.18: División del conjunto de datos para las diferentes etapas en la comprobación de un modelo

La manera más sencilla de crear estos conjuntos, es mediante una división al azar, eligiendo un porcentaje determinado para cada conjunto sobre el *dataset*. Algunos porcentajes comúnmente usados a la hora de dividir el conjunto de datos en el contexto del aprendizaje profundo son del 80 % para el conjunto de entrenamiento, 10 % para la validación y otro 10 % para el testeo [83]. Al ser este un trabajo simple y de introducción a este campo, tan solo se han utilizado los conjuntos de entrenamiento y validación, siendo suficientes para el proceso de elección y optimización del modelo de reconocimiento de las imágenes. Si bien, sería oportuno que en líneas futuras de este trabajo se continuará realizando pruebas con las imágenes no usadas para entrenamiento y validación.

5.9.1. Validación cruzada o *cross validation*

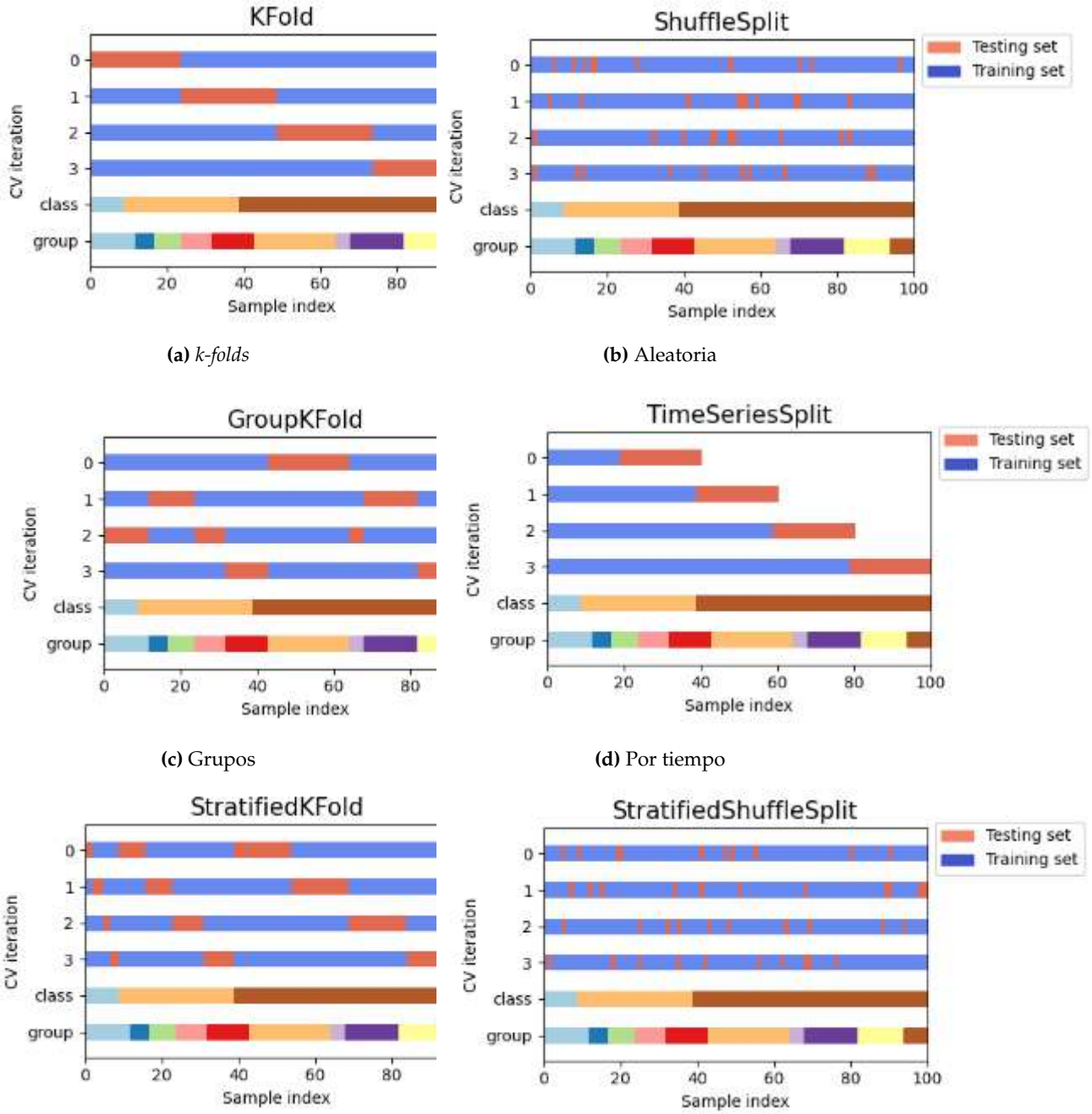
Aunque se ha explicado que la forma de entrenar un modelo es con conjuntos de entrenamiento y validación fijos creados al azar a partir del *dataset*, esto se trata de una manera básica de entrenar y validar un modelo, conocida como validación ***hold out*** [84]. Esta será la manera en la que se valide el modelo durante los primeros entrenamientos en este trabajo: para comparar el modelo preentrenado frente al creado desde cero, comparar los diferentes modelos preentrenados, la aumentación de datos y la optimización de hiperparámetros.

Sin embargo, existe un fallo en la utilización de un entrenamiento con conjunto de datos de entrenamiento, validación y prueba fijos. Si bien el modelo se está entrenando, validando y probando, lo está haciendo siempre con las mismas imágenes para cada conjunto, por lo que es posible que no termine de aprender todo lo que pueda sobre el *dataset*. Es por esto que en aprendizaje profundo se utiliza la **validación cruzada** o *cross validation*, la cual se centra en crear diferentes entrenamientos, variando los conjuntos de entrenamiento y validación, de forma que le modelo pueda aprender de todas las imágenes.

Existen diversas formas de aplicar la validación cruzada [85], algunas de ellas se muestran en la Figura 5.19. Se trata de diferentes maneras de variar los conjuntos de entrenamiento y validación (aunque se muestre en la imagen como *testing*, los conceptos de validación y prueba son a veces intercambiados. Estas imágenes se refieren a los conjuntos de datos para validar el modelo, es decir, conjunto de validación).

La manera más básica de elegir el conjunto de datos con validación cruzada sería la validación repetida por submuestreo aleatorio (5.19b), donde simplemente se repite durante varias iteraciones la validación *hold out*. El siguiente ejemplo es una de las formas más utilizadas de validación cruzada, la validación cruzada por *k-folds* (5.19a), donde se dividen los datos de forma aleatoria en k grupos de aproximadamente el mismo tamaño. Similar a esta, la validación cruzada por grupos 5.19c se utiliza si el conjunto de datos está formado por diferentes grupos además de las clases, escogiéndose un conjunto de grupo para validación en cada iteración. Por último, otra forma de realizar la validación cruzada es mediante la validación cruzada creciente con series de tiempo (5.19d), donde se divide el dataset como para validación *k-folds*, pero empezando por uno de entrenamiento y uno de validación e ir aumentando en uno los subconjuntos de entrenamiento, hasta llegar al total del *dataset*.

Además, en las Figuras 5.19e y 5.19f se muestra otra técnica común en la validación cruzada, la estratificación de la validación cruzada, de forma que además de repetir el entrenamiento con diferentes validaciones, se asegura que en el caso de que el *dataset* no este equilibrado, el número de datos de cada clase sea porcentual al número de muestras en la clase total.



(a) *k-folds*

(b) Aleatoria

(c) Grupos

(d) Por tiempo

(e) Estratificada por grupos

(f) Estratificada aleatoria

Figura 5.19: Diferentes tipos de validación cruzada explicados [86]

De entre las posibles técnicas de validación cruzada, se utilizarán el *k-fold*, aleatorio y temporal. Si bien se podrían haber utilizado más ejemplos, el conjunto de datos utilizado en este trabajo no está formado por grupos y la estratificación no es necesaria dado que las clases están igualadas en cuantos a las muestras en cada una.

5.10. Representación de la información

La inteligencia artificial y el deep learning son utilizadas para tratar con grandes cantidades de datos, el conocido como *Big Data*. En casos como el presente en este proyecto, la comparación de resultados se hace complicada con algunos métodos gráficos clásicos. Es por ello que también se han utilizado otros métodos para la representación de resultados para este tipo de técnicas, utilizándose en este trabajo las dos más extendidas.

5.10.1. Precisión y perdida

Aunque estos términos son algo triviales y no requieren una extensa explicación, gran parte de las comparaciones entre modelos se harán según la **precisión**, o *accuracy*, y la **pérdida**, o *loss*, que estos modelos ofrezcan, por los que se aclararan a continuación.

La precisión es un método para medir el rendimiento de un modelo de clasificación que se expresa típicamente como un porcentaje. Este valor es el recuento de predicciones en las que el valor predicho es igual al valor real. La precisión a menudo se representa gráficamente y se monitorea durante la fase de entrenamiento, aunque el valor suele asociarse con la precisión general o final del modelo. El valor de la precisión durante validación tiene una mayor importancia a la hora de escoger un modelo, puesto que muestra el rendimiento de ese modelo para reconocer imágenes fuera de su conjunto de entrenamiento. Esto implica que el modelo tendrá también una mayor precisión clasificando nuevas imágenes.

Una función de pérdida tiene en cuenta las probabilidades o la incertidumbre de una predicción en función de cuánto varía la predicción respecto al valor real. Esto brinda una visión más detallada de cómo está funcionando el modelo. A diferencia de la precisión, la pérdida no es un porcentaje, es una suma de los errores cometidos para cada muestra en los conjuntos de entrenamiento o validación. Por esto, la precisión tiene una mayor facilidad para interpretarse frente a la pérdida.

En la Figura 5.20 se ha representado un ejemplo de la precisión y pérdidas para un modelo cualquiera.

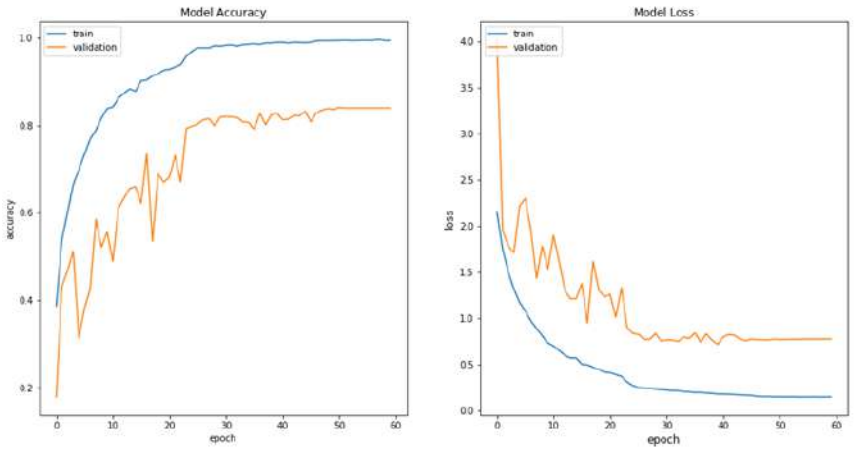


Figura 5.20: Gráficas para la precisión y pérdidas en entrenamiento y validación de un modelo

5.10.2. *Overfitting*

Si bien este concepto se podría haber explicado junto a las redes neuronales convolucionales, se ha decidido redactar en este apartado puesto que es un concepto fundamental a la hora de entender los resultados que un modelo ofrece.

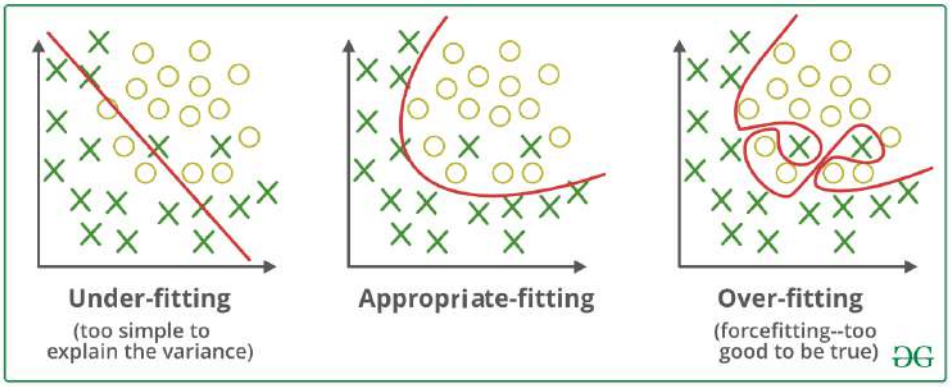


Figura 5.21: *Overfitting* y *underfitting* explicado [87]

El *overfitting* o sobreajuste es un error que se produce cuando una función está demasiado alineada con un conjunto limitado de datos. El resultado es un modelo que es útil solo cuando se refiere a su conjunto de datos inicial y no a cualquier otro conjunto de datos. La intuición con respecto a por qué se da este error se puede explicar como que el modelo aprende "de memoria" los datos dentro de un conjunto en lugar de aprender las características que definen cada clase de datos. De esta forma, como se explica en la Figura 5.21 el modelo no servirá para lo que interesa: tener un buen rendimiento con un conjunto de datos nuevos [88].

El *overfitting* se puede reconocer fácilmente si un modelo no está generando buenos datos de validación y se observa que la precisión en entrenamiento continua aumentando. Puede ocurrir que el modelo genere buenos datos de entrenamiento y validación hasta que llegue un punto donde el entrenamiento mejora y la validación no, como se observa en la Figura 5.22, ante lo cual la solución sería cortar el entrenamiento antes. Otras soluciones a la hora de entrenar modelos con *overfitting* pueden ser la validación cruzada, la aumentación de datos o el *pruning*, donde se eliminan algunos parámetros del modelo.

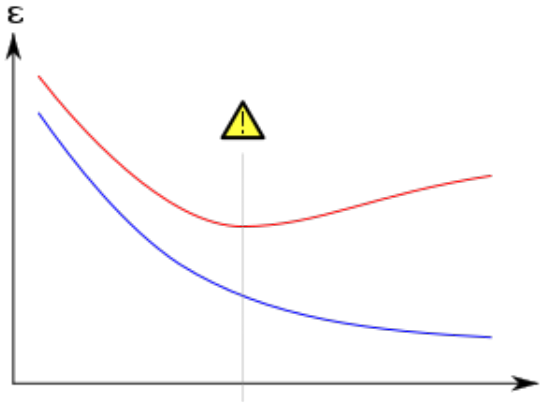


Figura 5.22: El *overfitting* puede comenzar en un punto donde la validación y el entrenamiento no avanzan igual

5.10.3. Matriz de confusión

La representación de la precisión o *accuracy*. así como de la pérdida o *loss*, del modelo en sus diferentes etapas permite evaluar fácilmente si el modelo cumple o no con nuestras necesidades. Sin embargo, estos parámetros tan solo aportan información sobre el resultado del modelo y no de lo que ocurre en su interior y cómo de bien está clasificando las diferentes clases. Ante esto, la matriz de confusión es la principal respuesta para entender el comportamiento interior del modelo.

Una matriz de confusión es una representación visual del rendimiento de un modelo de aprendizaje automático, mediante el resumen de los valores correctos y predichos de la clasificación. Los resultados se aprecian divididos según la clasificación real y la que ha predicho el modelo, pudiendo distinguir en qué tipo de datos ha fallado el modelo.

Como se observa en la Figura 5.23, un modelo de calcificación binaria cuyas dos salidas sean verdadero o positivo pueden tener cuatro tipos de resultados [89]:

- *True positive* o verdaderos positivos (TP): el modelo supone verdadero y el valor es verdadero.
- *False positive* o falsos positivos (FP): el modelo supone verdadero y el valor es negativo.
- *True negative* o verdaderos negativos (TN): el modelo supone negativo y el valor es negativo.
- *False negative* o falsos negativos (FN): el modelo supone negativo y el valor es verdadero.

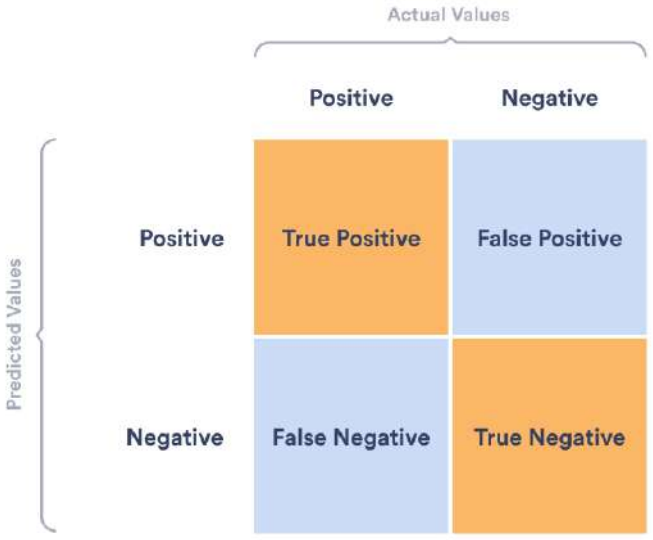


Figura 5.23: Posibles salidas de una matriz de confusión binaria [89]

Esta gráfica aporta una información que se hizo evidente durante la pandemia de COVID-19 para la mayor parte de la población mundial, y es que algunos fallos por parte de un modelo tienen más importancia que otros. De igual forma que un falso positivo de COVID u otra enfermedad es preferible a uno falso negativo, un modelo de aprendizaje profundo que al reconocer imágenes de ecografías para detectar cáncer de mama deberá centrarse en no clasificar un resultado positivo como negativo.

El modelo utilizado en este proyecto no es un modelo binario, sino que clasifica las imágenes entre cuatro posibilidades, por lo que se usará un modelo basado en el de la Figura 5.24, el cual

clasifica imágenes de diferentes tipos de flores. Al igual que en el ejemplo médico, la clasificación errónea de una clase en el modelo de este proyecto tendrá una mayor importancia según la clase con la que es confundida. Es sencillo visualizar que un parte de la pieza donde sobra material, siendo confundida con una en la que falta, supondría un mayor problema, puesto que se estaría añadiendo material en un lugar donde ya existe demasiado.

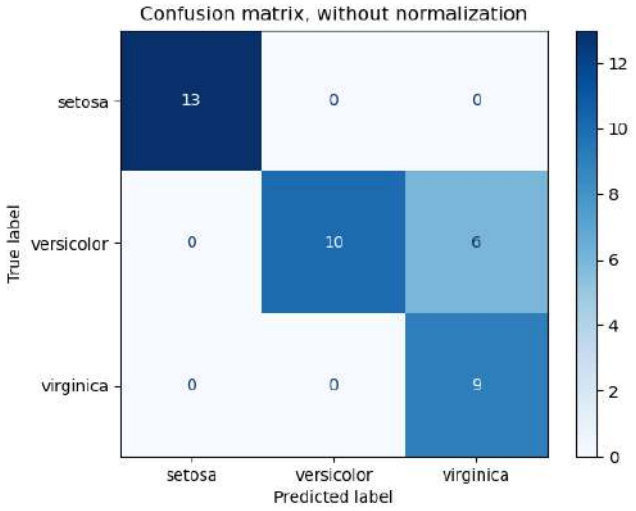


Figura 5.24: Ejemplo de una matriz de confusión no binaria para clasificación de flores [90]

5.10.4. Curva ROC

La curva ROC (*receiver operating characteristic*), al igual que la matriz de confusión, aporta información sobre la clasificación interna que hace el modelo, con la diferencia de que la curva ROC permite representar esta información gráficamente. Esta curva se genera calculando la tasa de verdaderos positivos (TPR), expresada en la ecuación 5.4 contra la tasa de falsos positivos (FPR), expresada en la ecuación 5.5 [91], para cada posible límite en el porcentaje de elección de una clase, que es un porcentaje entre 0 y 100 % o entre 0 y 1, como aparece en la gráfica que se utilizará.

$$TPR = Sensibility(Sensibilidad) = \frac{TP}{TP + FN} \tag{5.4}$$

$$FPR = (1 - Specifity(Especifidad)) = \frac{FP}{TN + FP} \tag{5.5}$$

Por tanto, esta gráfica permite representar como de bien clasifica un modelo una clase frente al resto de clases.

En la Figura 5.25 se observa una gráfica con ejemplos para entender intuitivamente el concepto tras la curva ROC. Un modelo perfecto será aquel que para cualquier límite en el porcentaje de clasificación de una clase tiene una sensibilidad perfecta, es decir, no comete errores y la curva tiene un valor constante igual a 1. Por otro lado, un modelo que tenga una curva ROC igual a la bisectriz del primer cuadrante (y = x) se tratará de un modelo que siempre tiene una tasa TPR igual a su tasa

FPR, es decir, que acierta lo mismo que falla y tiene un 50 % de éxito, no siendo mejor que clasificar las clases al azar. Entre estos dos extremos se hallan resto de modelos que se acercarán más a una u otra curva según su comportamiento.

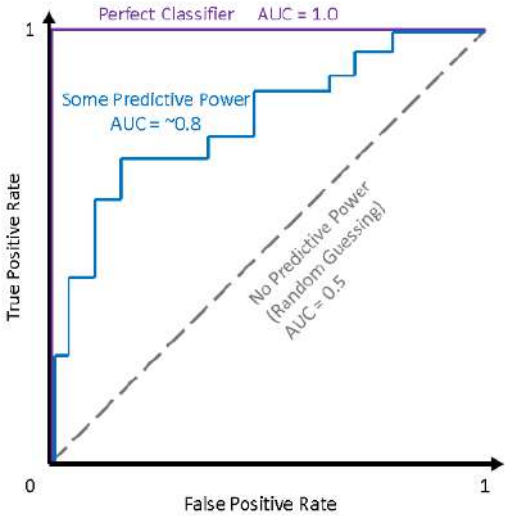
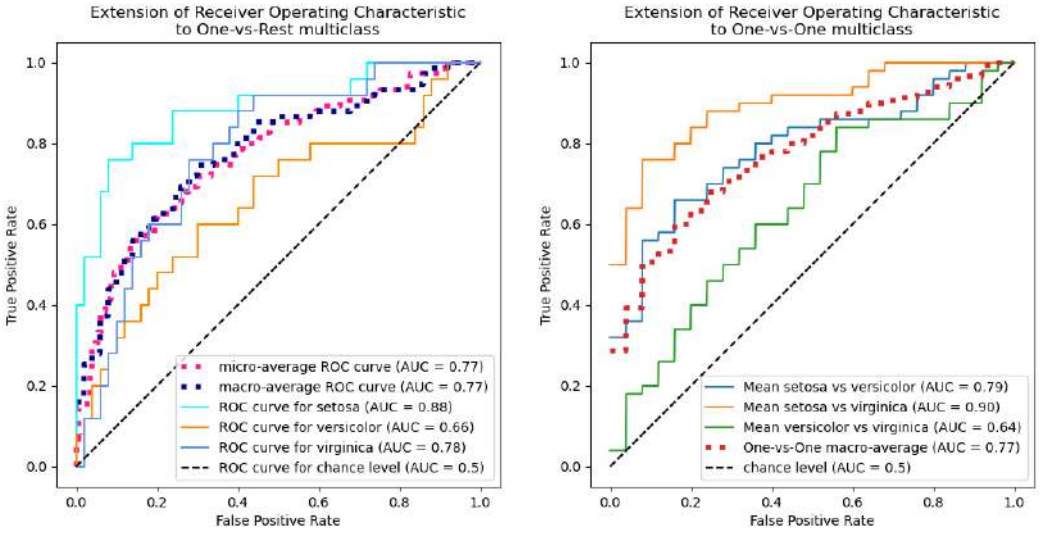


Figura 5.25: Ejemplo teórico de los diferentes tipos de modelos representados con una curva ROC [91]

En la figura anterior también se introduce el término AUC (*area under ROC*), un valor que permite tener un valor numérico para reflejar el comportamiento del modelo, además de la interpretación visual de la curva ROC. Como indican sus siglas en inglés, este valor es el área de la curva bajo la curva ROC. Una curva para el modelo que anteriormente se explicaba como perfecto, tendría una AUC igual a 1, mientras que el modelo que se comporta como el azar, tendrá una AUC de 0.5.

La curva ROC se define a través de la clasificación de una clase con respecto a otra, sin embargo, es posible hacer cálculos para representar una clase frente al resto en un problema multiclase. El código necesario para representar estas curvas se basará en el explicado en el ejemplo de Scikit-learn [92]. Este ejemplo continúa con el modelo detector de diferentes plantas ya visto en el apartado 5.10.3 y calcula la curva ROC tanto para una clase frente al resto u *One vs. All* (OvA), representado en la Figura 5.26a, como para una clase frente a otra, *One vs. One* (OvO), Figura 5.26b. Además, en este ejemplo se calculan los valores con promedio micro y macro, cuya diferencia es que el promedio macro da igual importancia a cada peso dentro de cada categoría, mientras que el promedio micro da igual importancia a cada peso dentro de cada muestra. No es relevante el tipo de cálculo según el promedio, puesto que ambos ofrecen el mismo resultado [93].



(a) Todas las curvas ROC OvR

(b) Todas las curvas ROC OvO

Figura 5.26: Ejemplos de curvas ROC para un modelo multiclase clasificador de flores [92]

CAPÍTULO 6

ANÁLISIS DE RESULTADOS

En este apartado se muestran los resultados obtenidos durante las etapas de entrenamiento de la red neuronal, comparando diferentes modelos hasta llegar a uno óptimo. Para ello se han comparado los modelos, utilizando diferentes técnicas y cambiados parámetros, siguiendo un proceso en el cual se ha continuado con los modelos con mejores resultados, hasta llegar a uno con el mejor comportamiento.

Se ha comenzado mediante la comparación de un modelo preentrenado frente a uno desde cero o *scratch*, variando el tamaño del *dataset* y de las épocas, con la intención de explicar de esta forma las cualidades de los modelos preentrenados.

Tras esto, se han comparado los principales modelos preentrenados utilizado en la biblioteca PyTorch, los cuales se describieron teóricamente en la Sección 5.5.

Se ha elegido el modelo preentrenado con una mejor respuesta frente a los datos de entrada, buscando un equilibrio entre exactitud de la salida y tiempo empleado. Con este modelo, se ha buscado una posible mejora mediante dos técnicas; la aumentación de datos y la variación de hiperparámetros.

Durante estas comparaciones, se han utilizado principalmente 4 gráficas. Se ha comenzado graficando la exactitud, o *accuracy*, y las pérdidas, o *loss*, para observar la respuesta del modelo tanto en entrenamiento como en validación. Después, se han utilizado las gráficas de la matriz de confusión y curva ROC, las cuales aportan un mayor entendimiento del comportamiento interior del modelo.

Es importante resaltar, antes de comparar los resultados obtenidos, que tipo de precisión se considera buena. Una precisión del 25 % se trataría de un modelo que es capaz de clasificar el modelo con el mismo resultado que una clasificación al azar, dado que el conjunto de datos cuenta con cuatro clases, mientras que una clasificación al 100 % sería perfecta. Entre estos dos valores, dependerá del tipo de dataset y de cómo de importante sea la clasificación, puesto que no es lo mismo un *dataset* balanceado que uno con la mayor parte de imágenes en una categoría.

En nuestro caso, siendo este un *dataset* balanceado (mismo número de imágenes para cada data-

set), se podrían usar los siguientes límites [94]:

- Mayor a 90 % - Muy bueno
- Entre 70 % y 90 % - Bueno
- Entre 60 % y 70 % - Normal
- Menor a 60 % - Malo

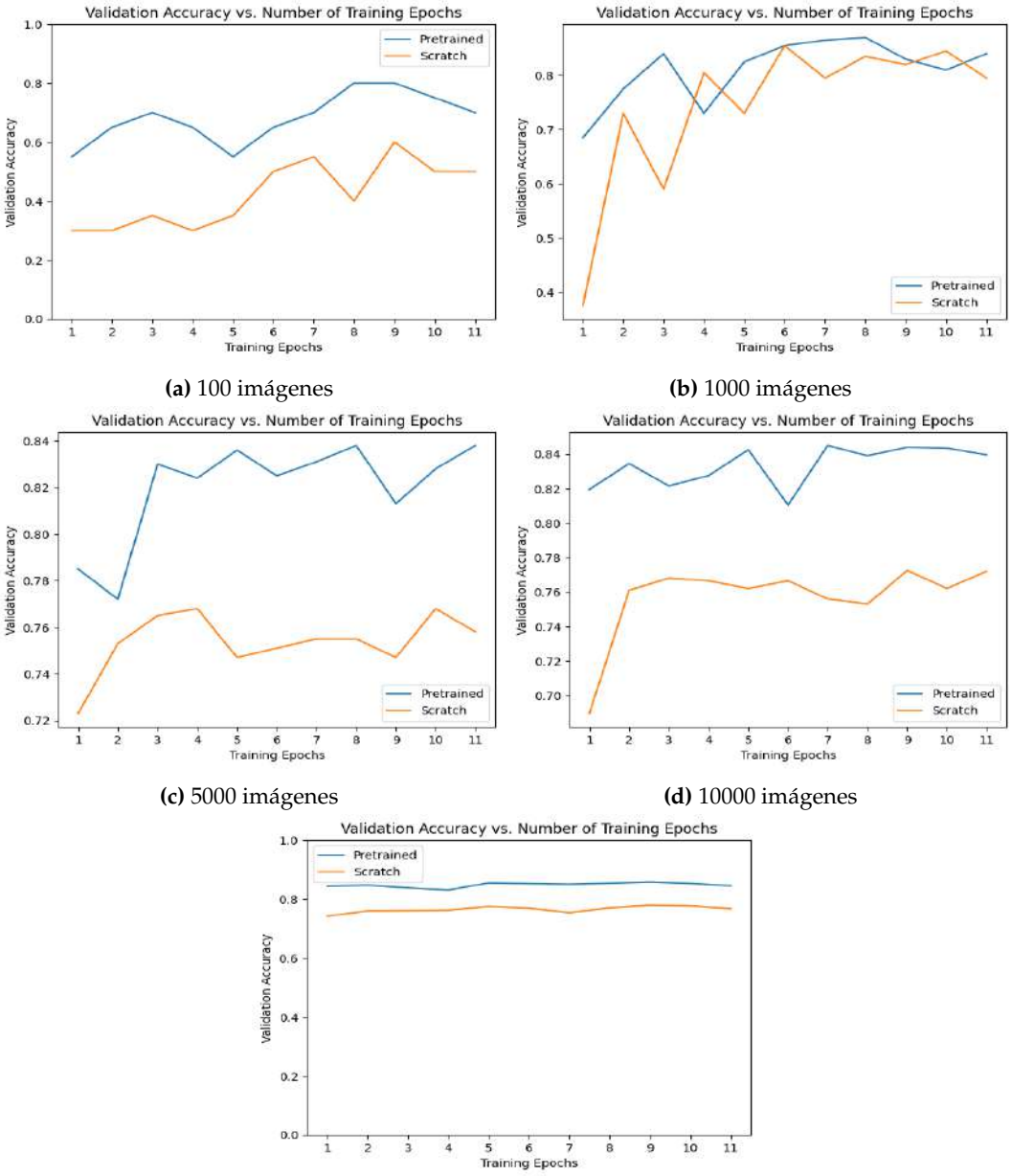
Por último, se utilizarán diversos ejemplos de validación cruzada explicados en la Sección 5.9.1, donde se pondrán en práctica estos conceptos y se comprobarán sus beneficios.

6.1. Red preentrenada vs. *from scratch*

Para una mayor comprensión de las diferencias entre estos modelos, se ha variado el tamaño del dataset, así como el número de épocas, cambiando la duración del entreno. Para esta comparación, se ha utilizado el modelo preentrenado Squeezenet, ya que se trata de uno de los modelos más simples y con mejor respuesta, a la vez que rápido [59] [95]. Se ha representado la precisión de validación y entrenamiento para *datasets* de 100, 1000, 5000, 10000 y 15000 imágenes durante once y cincuenta épocas.

En las Figuras 6.1 y 6.2 se han representado la precisión de validación y entrenamiento de los modelos preentrenado y desde cero para diferentes tamaños de datasets y once épocas. Se puede observar como para tamaños pequeños los modelos desde cero o *scratch* no son capaces de conseguir un rendimiento demasiado bueno, mientras que los modelos preentrenados utilizando el *fine tunning* llegan a resultados buenos. Esto es debido a que con esta técnica, se están utilizando los pesos predeterminados con los que cuenta el modelo SqueezeNet, el cual ha sido previamente entrenado con millones de imágenes de todo tipo. Estos pesos pueden ser un buen punto de partida para entrenar otro conjunto de imágenes, como ocurre para este *dataset*.

Si se hubiera dado el caso en el que el número de imágenes fuera limitado (50, 100 o 200 imágenes), la técnica de *fine tunning* hubiera permitido encontrar un modelo con un rendimiento bastante bueno. Sin embargo, al tener un dataset con miles de imágenes y en el cual las cuatro clases se diferencian principalmente por colores, sin demasiadas formas complejas, un modelo *scratch* es capaz de entrenarse y obtener resultados parecidos a los de los modelos preentrenados. Es también relevante resaltar que, aunque el modelo desde cero consigue resultado buenos, lo hace con un mayor tiempo de procesamiento empleado, como se refleja en la Tabla 6.1, y no llega a tener una precisión como tan alta como el preentrenado. Además, para el caso del entrenamiento con 15000 imágenes desde cero con 50 épocas, el entrenamiento desde cero ha tardado demasiado como para poder terminarlo y representarlo, teniendo errores durante diversos intentos de simulación.



(a) 100 imágenes

(b) 1000 imágenes

(c) 5000 imágenes

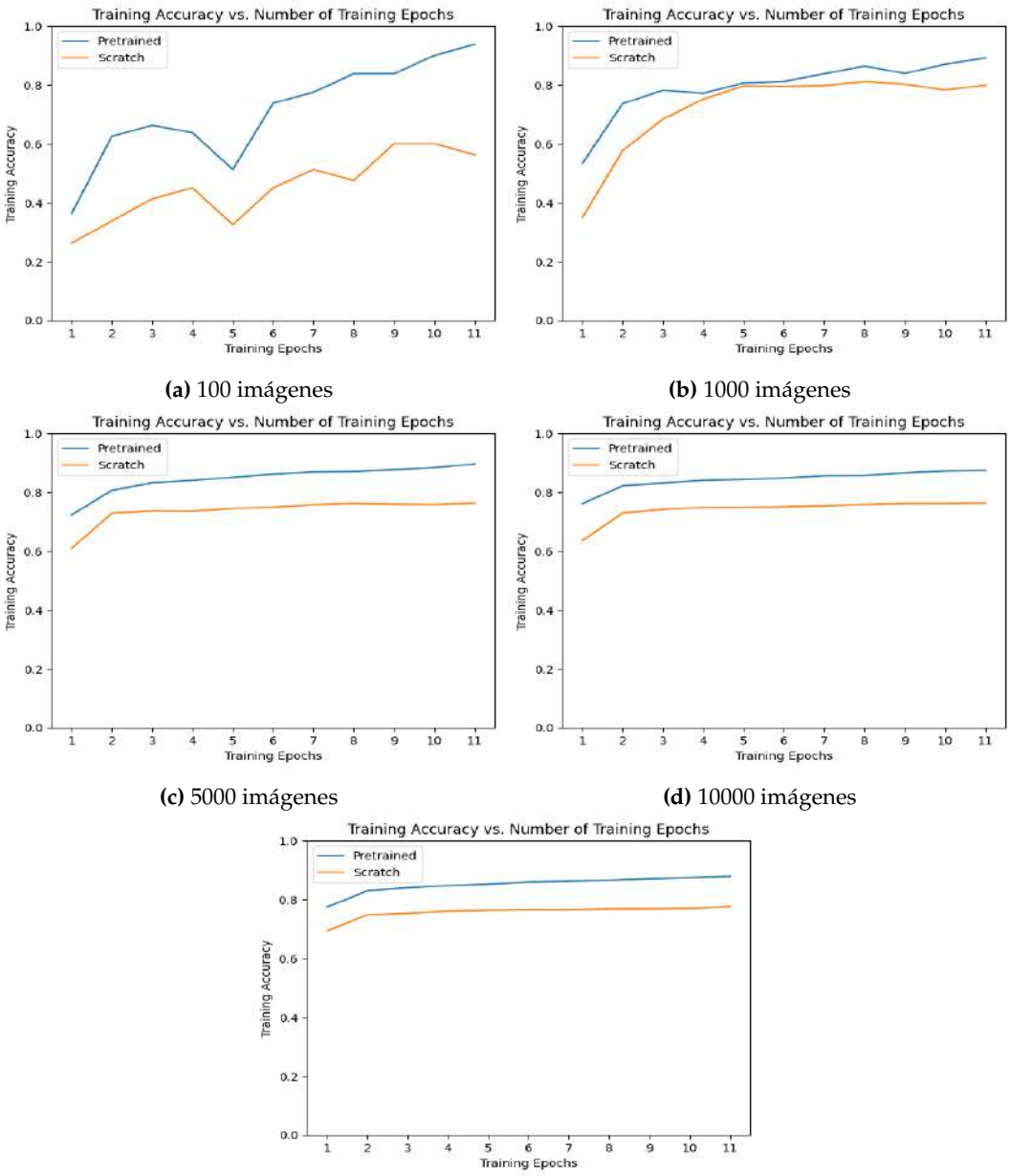
(d) 10000 imágenes

(e) 15000 imágenes

Figura 6.1: Precisión de validación con modelos preentrenado y desde cero para diferentes tamaños de *dataset* y once épocas

Tamaño dataset	Preentrenado	*Scratch*
100	0′ 45″	0′ 53″
1000	2′ 28″	3′ 34″
5000	11′ 04″	11′34″
10000	27′ 58″	28′ 58″
15000	44′ 26″	46′ 58″

Tabla 6.1: Tiempo de entrenamiento con modelos preentrenado y desde cero para diferentes tamaños de *dataset* y once épocas



(a) 100 imágenes

(b) 1000 imágenes

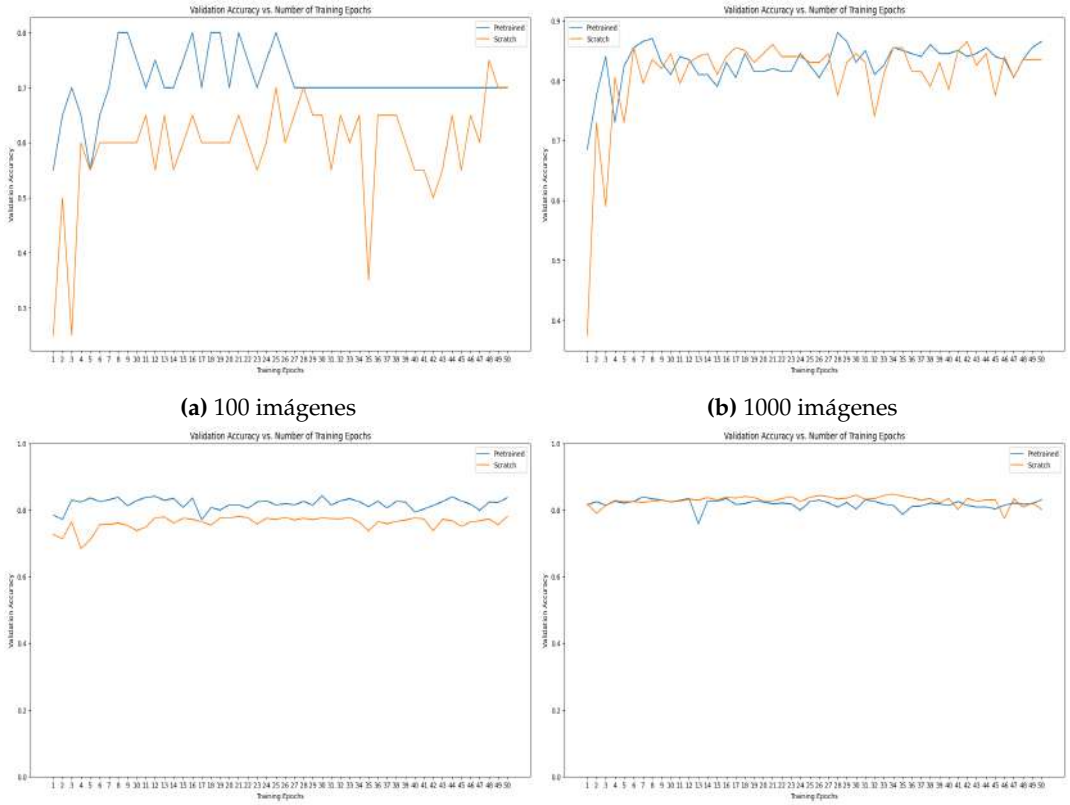
(c) 5000 imágenes

(d) 10000 imágenes

(e) 15000 imágenes

Figura 6.2: Precisión de entrenamiento con modelos preentrenado y desde cero para diferentes tamaños de *dataset* y once épocas

Tras esto, se han vuelto a realizar los mismos entrenamientos para un tiempo de entrenamiento de cincuenta épocas, representados en las Figuras 6.3 y 6.4 para validación y entrenamiento, respectivamente. En estos entrenamientos, se confirman los conceptos ya vistos para once épocas. El aumento de épocas supone un aumento de tiempo de procesamiento para el entrenamiento, pero no una subida considerable en precisión, por lo que se continuará con las once épocas para el resto de apartados. Además, se continuará con un dataset de 10000 imágenes, puesto que se aprecia con estos resultados que es suficiente para conseguir buenos resultados sin consumir un tiempo exagerado en entrenamiento.



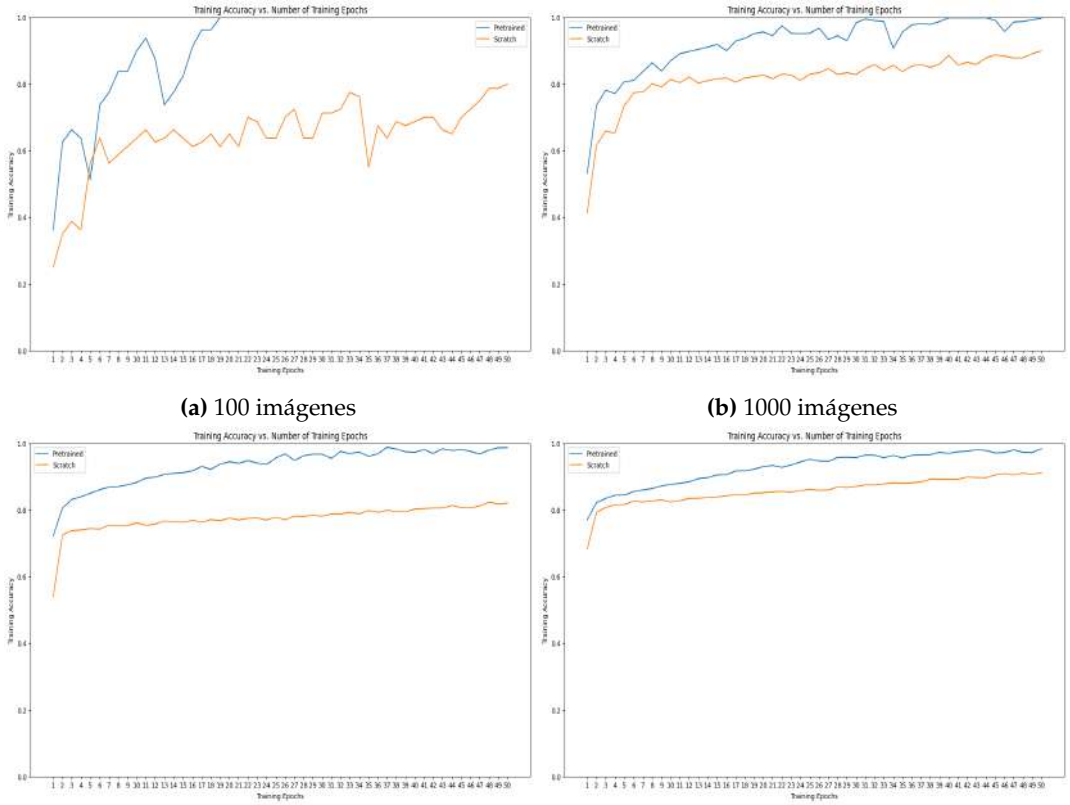
(a) 100 imágenes **(b)** 1000 imágenes

(c) 5000 imágenes **(d)** 10000 imágenes

Figura 6.3: Precisión de validación con modelos preentrenado y desde cero para diferentes tamaños de *dataset* y cincuenta épocas

Tamaño dataset	Preentrenado	*Scratch*
100	0′ 45″	0′ 53″
1000	4′ 3″	4′ 33″
5000	26′ 29″	27′ 5″
10000	36′ 53″	46′ 18″
20000	53′ 4″	No conseguido

Tabla 6.2: Tiempo de entrenamiento con modelos preentrenado y desde cero para diferentes tamaños de *dataset* y cincuenta épocas



(a) 100 imágenes **(b)** 1000 imágenes

(c) 5000 imágenes **(d)** 10000 imágenes

Figura 6.4: Precisión de entrenamiento con modelos preentrenado y desde cero para diferentes tamaños de *dataset* y cincuenta épocas

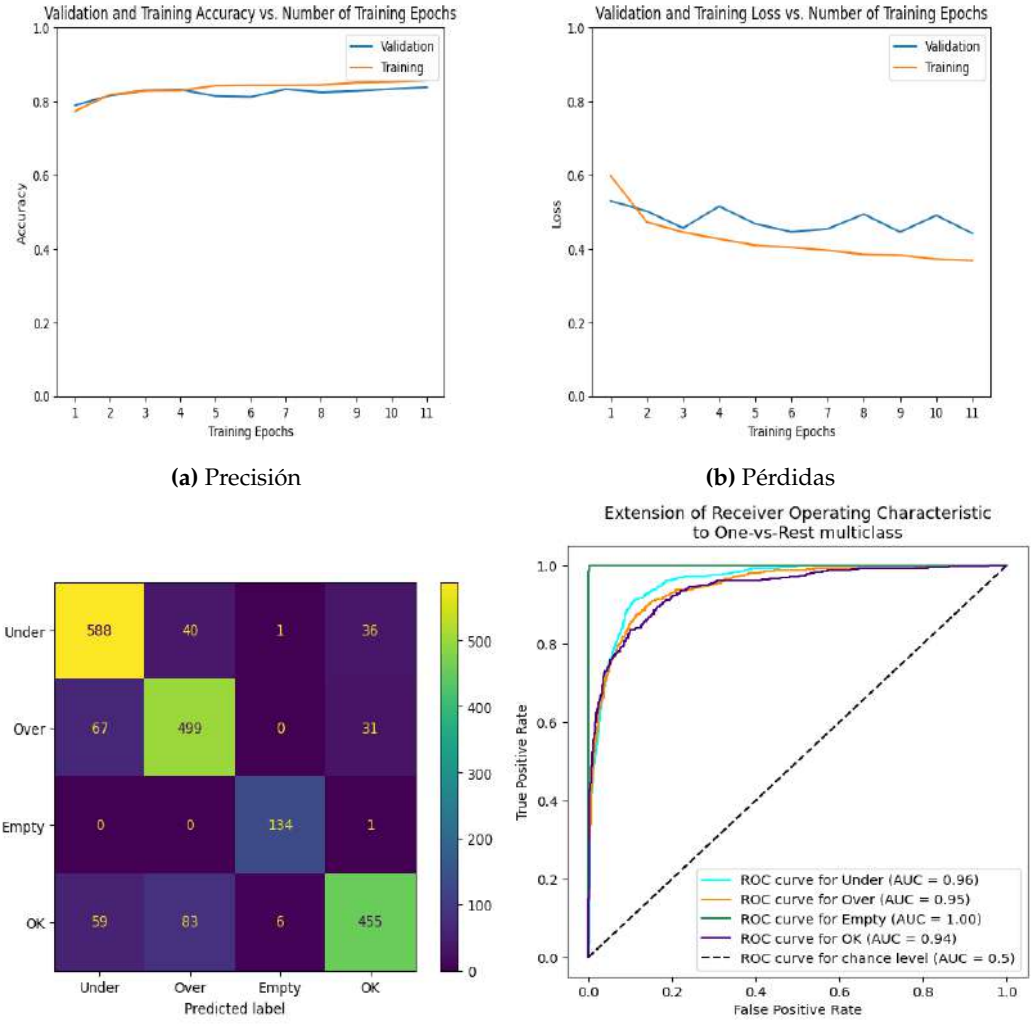
6.1.1. Conclusiones

Se continuará con un modelo que entrene durante once épocas y 10000 imágenes en el conjunto de datos. En la Figura 6.5 se observan los resultados que ofrece este modelo, tanto su precisión y pérdida de validación y entrenamiento como su Curva ROC y matriz de confusión.

Se aprecia tanto en la matriz de confusión como en la Curva ROC que el modelo consigue reconocer la clase *Empty* con una precisión de casi el 100 %, lo cual es lógico, puesto que estas imágenes están vacías y son fáciles de reconocer con respecto al resto. Sin embargo, es importante que esta clase se reconozca con facilidad para que la máquina no imprima en zonas vacías en las que no tendría sentido y pudiera provocar problemas en la impresión física de las piezas.

Por otro lado, es importante que el modelo sea capaz de diferenciar con precisión las imágenes *Under* de las *Over*, puesto que el error de clasificación más grave del modelo ocurriría cuando el modelo asigne el valor *Under* a una imagen *Over* y al contrario. Esto es así, puesto que si se utiliza un rellenado de huecos o lijado tras la impresión de la capa, se estaría dando una orden de rellenar huecos a una parte de la pieza que ya tiene demasiado material o lijando una parte con falta de material. Dentro de estos fallos, el lijado de material no daría problemas, ya que lijar una zona vacía no supone un problema mayor que el que ya existía de falta de material. Sin embargo, el rellenado de huecos de una parte con demasiado material estaría provocando un error aún mayor al que se tenía, imprimiendo más material en una zona donde ya existe demasiado, incluso podría suponer

fallos físicos de la máquina al chocar con una zona donde se ha supuesto que no había material. Es por esto que este tipo de fallos se deben evitar con prioridad.



(a) Precisión

(b) Pérdidas

(c) Matriz de confusión para validación

(d) Curva ROC para validación OvR

Figura 6.5: Resultados del modelo preentrenado

6.2. Comparación de modelos preentrenados

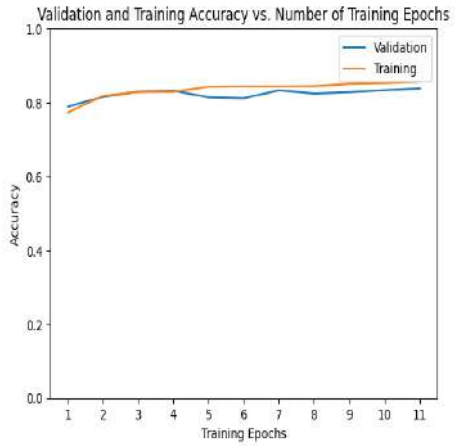
Se continúa la elección del modelo óptimo comparando los resultados de los principales modelos preentrenados vistos en la Sección 5.5. Para ello, se ha graficado la precisión en validación y entrenamiento, así como la pérdida del modelo. Además, también se han representado la matriz de confusión y las curva ROC de las clases frente al resto (curva ROC OvR) para tener un mejor entendimiento del funcionamiento de los modelos. Se han entrenado los modelos durante 11 épocas y con 10000 imágenes tras ver que estos valores eran suficientes en el apartado anterior.

6.2.1. SqueezeNet

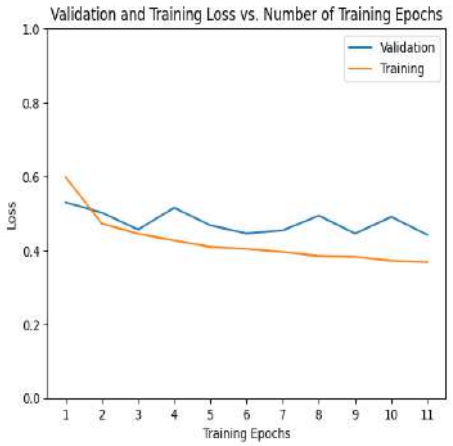
En la Figura 6.6 se encuentran graficados los resultados para el entrenamiento del modelo SqueezeNet con diez mil imágenes y once épocas. En la Tabla 6.3 se encuentran resumidos los resultados, así como el tiempo concurrido durante el entrenamiento.

Modelo	SqueezeNet
Max Acc Val	82.25 %
Max Acc Train	82.09 %
Min Loss Val	0.4617
Min Loss Train	0.3630
Time	26′ 58″

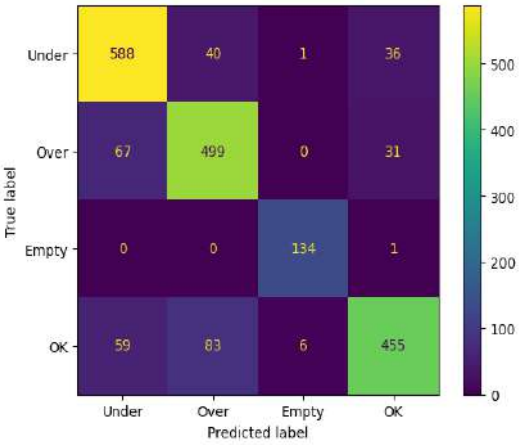
Tabla 6.3: Resultados SqueezeNet



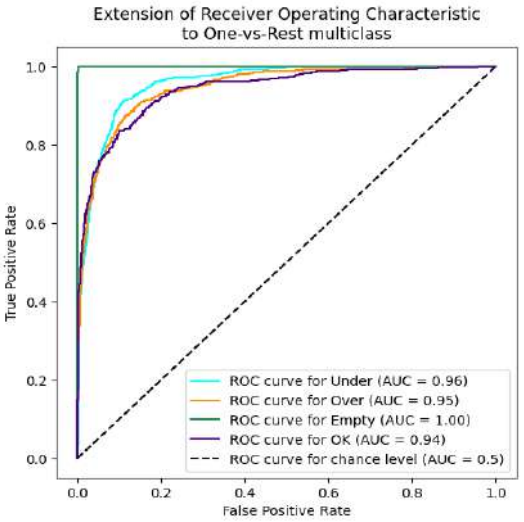
(a) Precisión



(b) Pérdidas



(c) Matriz de confusión para validación



(d) Curva ROC para validación

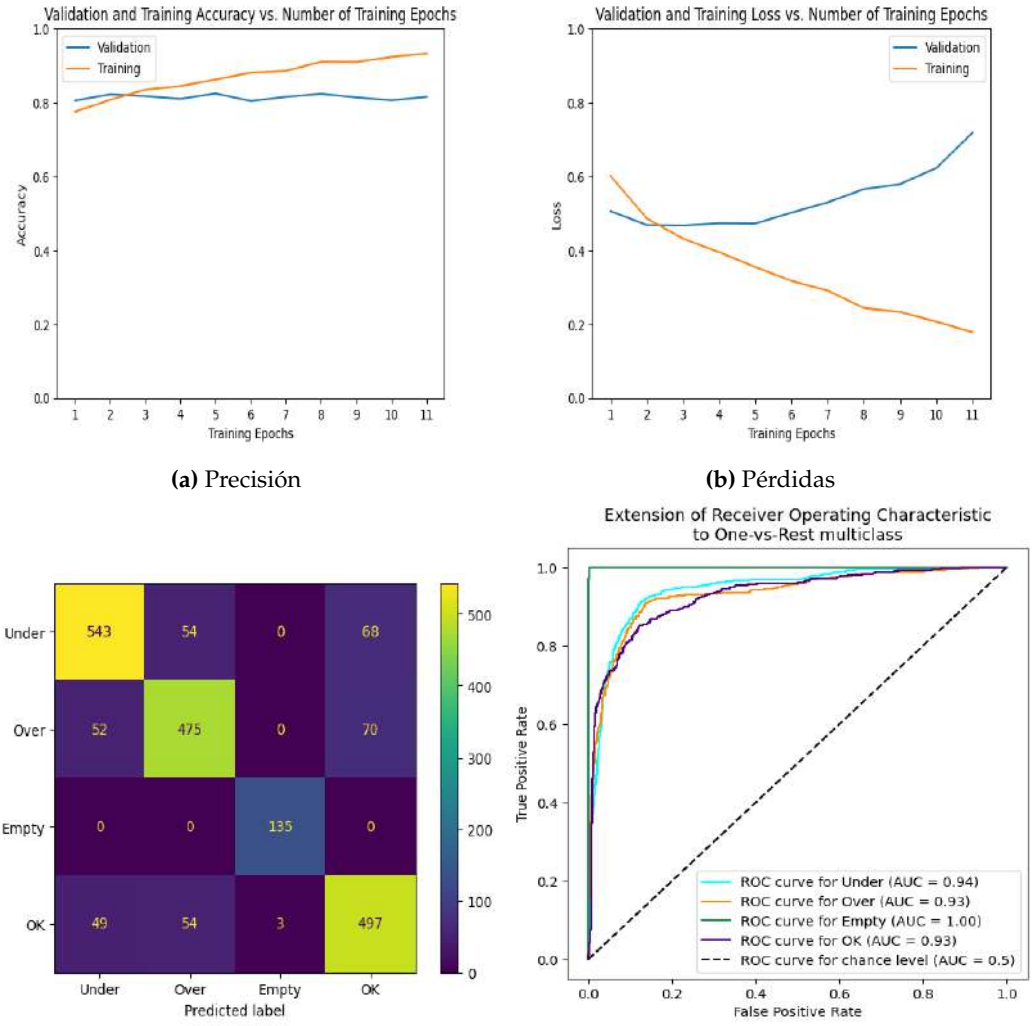
Figura 6.6: Resultados del entrenamiento con SqueezeNet

6.2.2. AlexNet

En la Figura 6.7 se encuentran graficados los resultados para el entrenamiento del modelo AlexNet con diez mil imágenes y once épocas. En la Tabla 6.4 se encuentran resumidos los resultados, así como el tiempo concurrido durante el entrenamiento.

Modelo	AlexNet
Max Acc Val	82.5 %
Max Acc Train	92.55 %
Min Loss Val	0.5150
Min Loss Train	0.2026
Time	24′ 36″

Tabla 6.4: Resultados AlexNet



(a) Precisión

(b) Pérdidas

(c) Matriz de confusión para validación

(d) Curva ROC para validación

Figura 6.7: Resultados del entrenamiento con AlexNet

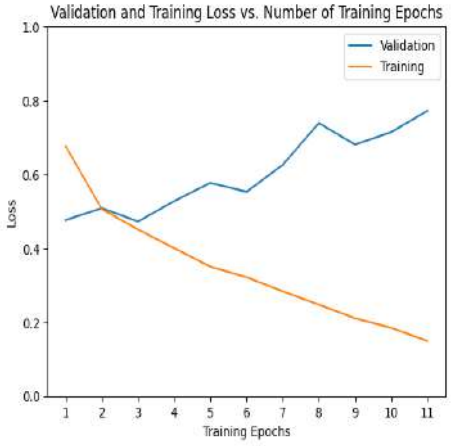
6.2.3. ResNet

En la Figura 6.8 se encuentran graficados los resultados para el entrenamiento del modelo ResNet con diez mil imágenes y once épocas. En la Tabla 6.5 se encuentran resumidos los resultados, así como el tiempo concurrido durante el entrenamiento.

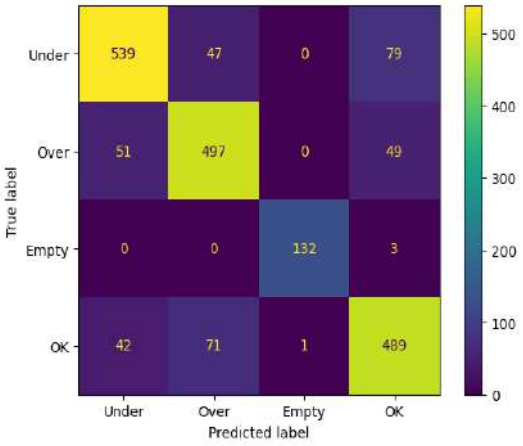
Modelo	ResNet
Max Acc Val	82.65 %
Max Acc Train	95.03 %
Min Loss Val	0.4807
Min Loss Train	0.1419
Time	17′ 59″

Tabla 6.5: Resultados ResNet

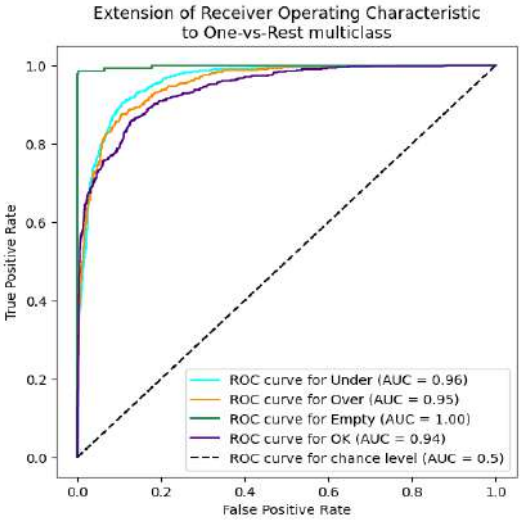
(a) Precisión



(b) Pérdidas



(c) Matriz de confusión para validación



(d) Curva ROC para validación

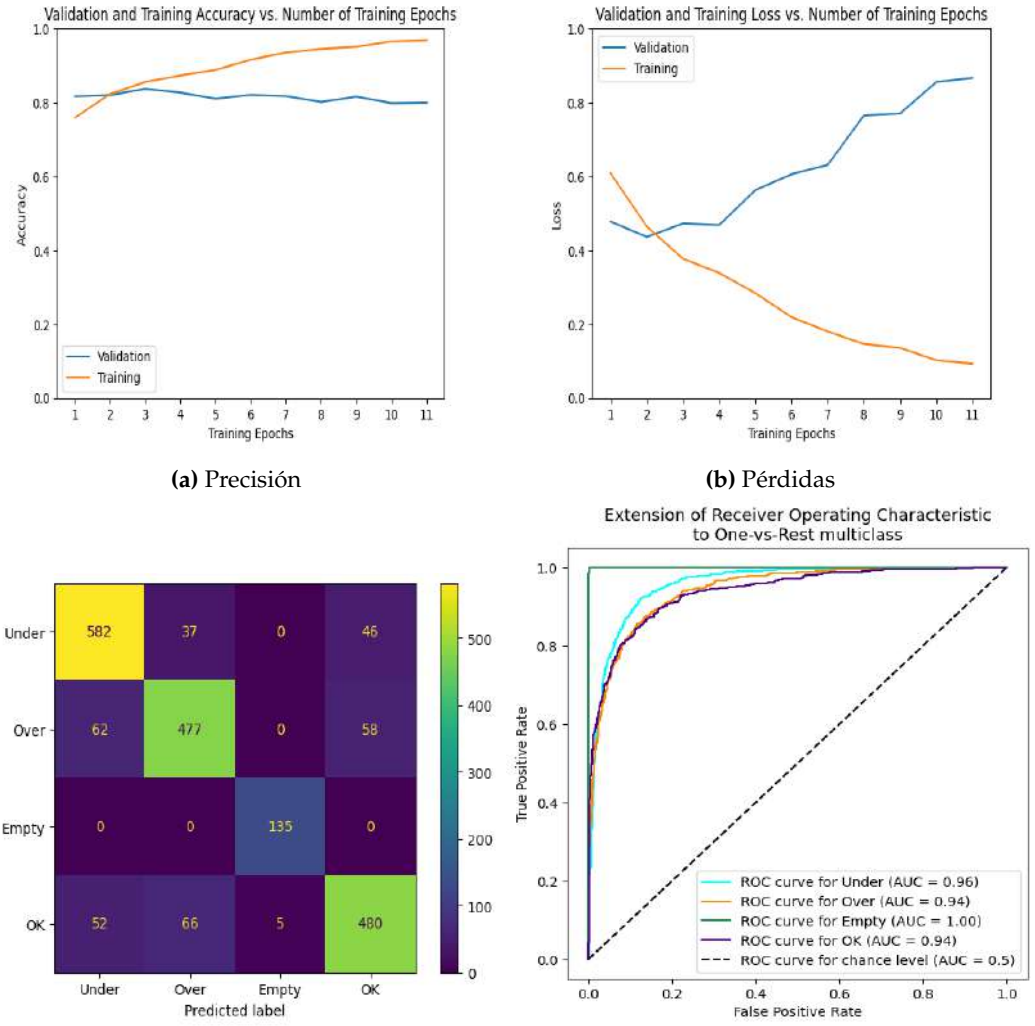
Figura 6.8: Resultados del entrenamiento con ResNet

6.2.4. VGG

En la Figura 6.9 se encuentran graficados los resultados para el entrenamiento del modelo VGG con diez mil imágenes y once épocas. En la Tabla 6.6 se encuentran resumidos los resultados, así como el tiempo concurrido durante el entrenamiento.

Modelo	VGG
Max Acc Val	82.75 %
Max Acc Train	97.10 %
Min Loss Val	0.4523
Min Loss Train	0.088
Time	71′ 06″

Tabla 6.6: Resultados VGG



(a) Precisión

(b) Pérdidas

(c) Matriz de confusión para validación

(d) Curva ROC para validación

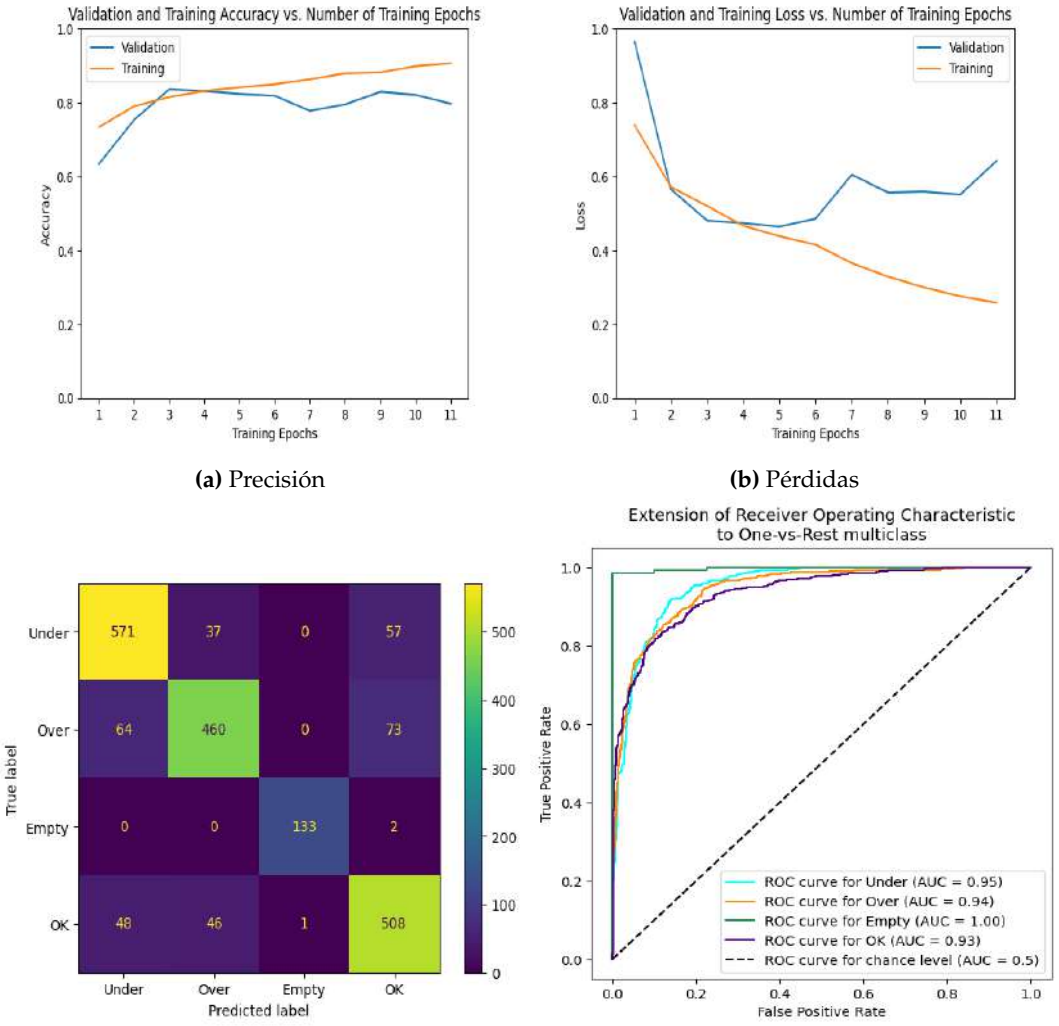
Figura 6.9: Resultados del entrenamiento con VGG

6.2.5. DenseNet

En la Figura 6.10 se encuentran graficados los resultados para el entrenamiento del modelo DenseNet con diez mil imágenes y once épocas. En la Tabla 6.7 se encuentran resumidos los resultados, así como el tiempo concurrido durante el entrenamiento.

Modelo	DenseNet
Max Acc Val	81.80 %
Max Acc Train	90.21 %
Min Loss Val	0.4859
Min Loss Train	0.2686
Time	65'39''

Tabla 6.7: Resultados DenseNet



(a) Precisión

(b) Pérdidas

(c) Matriz de confusión para validación

(d) Curva ROC para validación

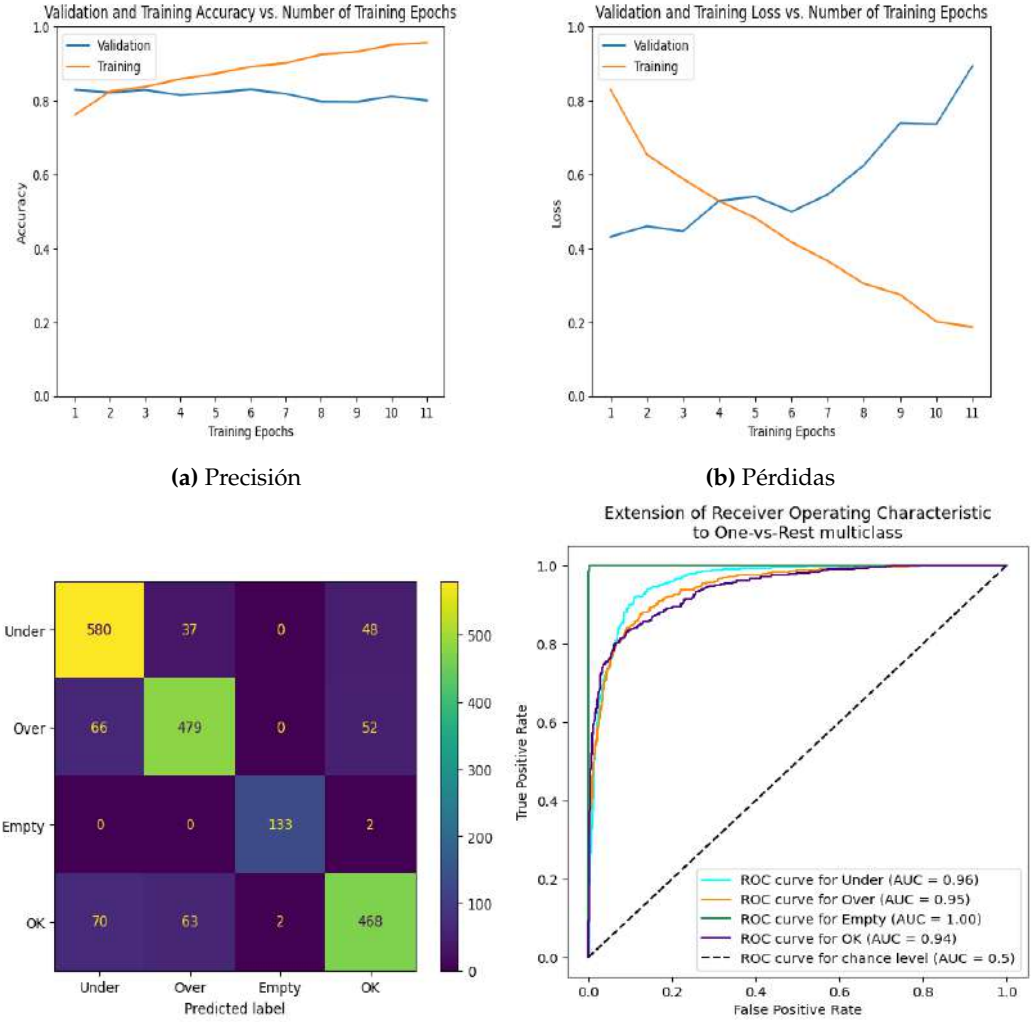
Figura 6.10: Resultados del entrenamiento con DenseNet

6.2.6. Inception

En la Figura 6.11 se encuentran graficados los resultados para el entrenamiento del modelo Inception con diez mil imágenes y once épocas. En la Tabla 6.8 se encuentran resumidos los resultados, así como el tiempo concurrido durante el entrenamiento.

Modelo	Inception
Max Acc Val	83.85 %
Max Acc Train	94.89 %
Min Loss Val	0.4786
Min Loss Train	0.2021
Time	42′21″

Tabla 6.8: Resultados Inception



(a) Precisión

(b) Pérdidas

(c) Matriz de confusión para validación

(d) Curva ROC para validación

Figura 6.11: Resultados del entrenamiento con Inception

6.2.7. Resumen de resultados y conclusiones

Tras estudiar estos resultados, se confirman parcialmente las predicciones vistas en la teoría relativa a los diferentes modelos preentrenados. Modelos como Inception, VGG y DenseNet son capaces de obtener una mayor precisión en entrenamiento, aunque su mayor complejidad también lleva consigo un mayor tiempo de entrenamiento. Es por esto que para este dataset, modelos menos complejos suponen una mejor elección, puesto que estos son suficientes para clasificar imágenes sencillas como las presentes en el *datset*.

Entre los modelos más simples, ResNet consigue precisiones de validación y de entrenamiento altas, manteniendo el tiempo de ejecución más bajo. Además, la curva ROC muestra una muy buena clasificación, especialmente para las clases *Over* y *Under*. Prestando especial atención a las imágenes con *Over* clasificadas como *Under* (este es el peor de los fallos posibles), se observa en la matriz de confusión que ResNet hace un muy buen trabajo en esta clasificación y cuenta con el menor número de falsos positivos de este tipo.

Por estas razones, se ha continuado con el modelo ResNet para el resto de experimentos. Cabe destacar también que todos los modelos sufren algo de *overfitting*, lo que se puede detectar tanto en las curvas de precisión, al ver como la precisión de entrenamiento mejora, pero la de validación no. Este fallo se tratará de solucionar con las siguientes modificaciones.

6.3. Aumentación de datos

Se aplicarán diferentes transformaciones a las imágenes, según lo visto en el apartado 5.6, para comprobar si se corrige el *overfitting* del modelo y se mejora la validación, especialmente la clasificación entre clases que no deben ser confundidas. Para ello, se han utilizado transformaciones disponibles en PyTorch, así como en la librería imgaug, la cual se especializa en transformaciones en imágenes. Cabe destacar que para estudiar los cambios provocador por estas transformaciones, se ha representado la matriz de confusión y la curva ROC para la clase *Under* sobre *Over*, ya que este es el peor de los errores posibles y el que se pretende solucionar con las transformaciones.

6.3.1. Transformación en tamaño y giro

Se comienza con un pequeño giro de 20º y un aumento de las imágenes. En la Figura 6.12 se han representado las imágenes transformadas para comprobar los efectos del cambio, y en las Figuras 6.13a y 6.13b se han representado la matriz de confusión y curva ROC OvO para la clase Under vs. Over, respectivamente.

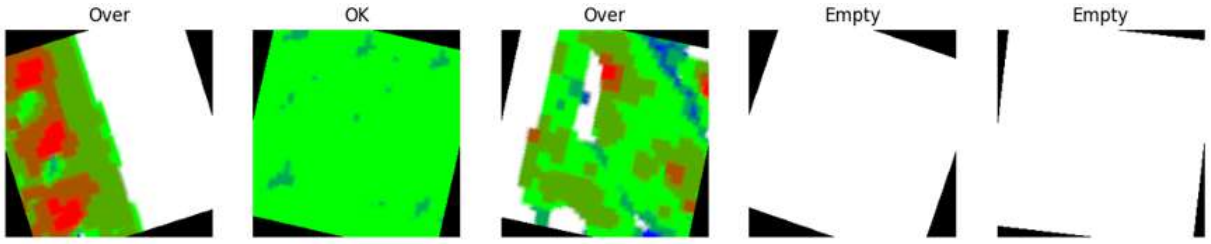
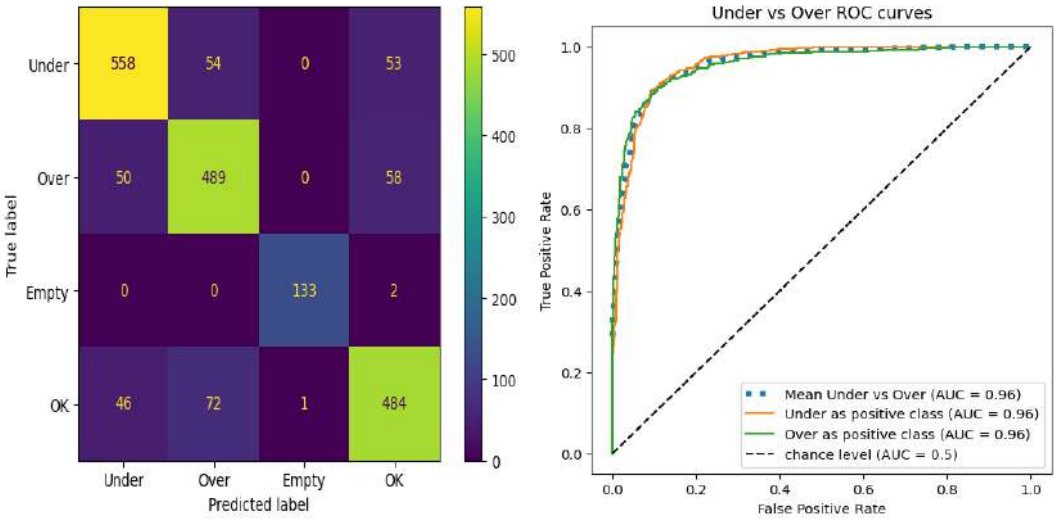


Figura 6.12: Imágenes transformadas con algunas transformaciones simples de PyTorch



(a) Matriz de confusión

(b) Curva ROC

Figura 6.13: Resultados del entrenamiento con algunas transformaciones simple

6.3.2. Transformaciones complejas PyTorch

Tras esto, se han aumentado las transformaciones, añadiendo algunas más complejas. Se han utilizado transformaciones para cambiar el brillo, contraste, saturación, tono y tamaño de la imagen. Se ha aplicado también un giro horizontal a algunas imágenes y un giro de 45º, además de un desenfoque gaussiano. En la Figura 6.14 se han representado los efectos de estos cambios aplicados en las imágenes y los resultados en las Figuras 6.15a y 6.13b.

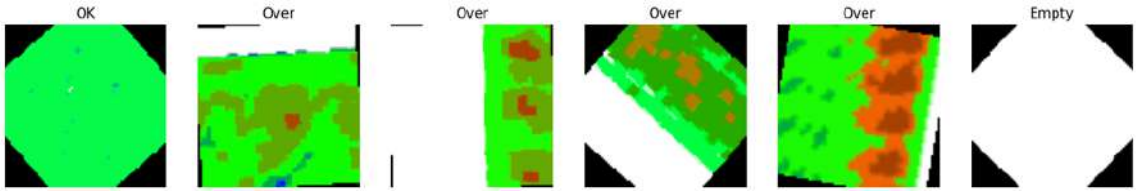
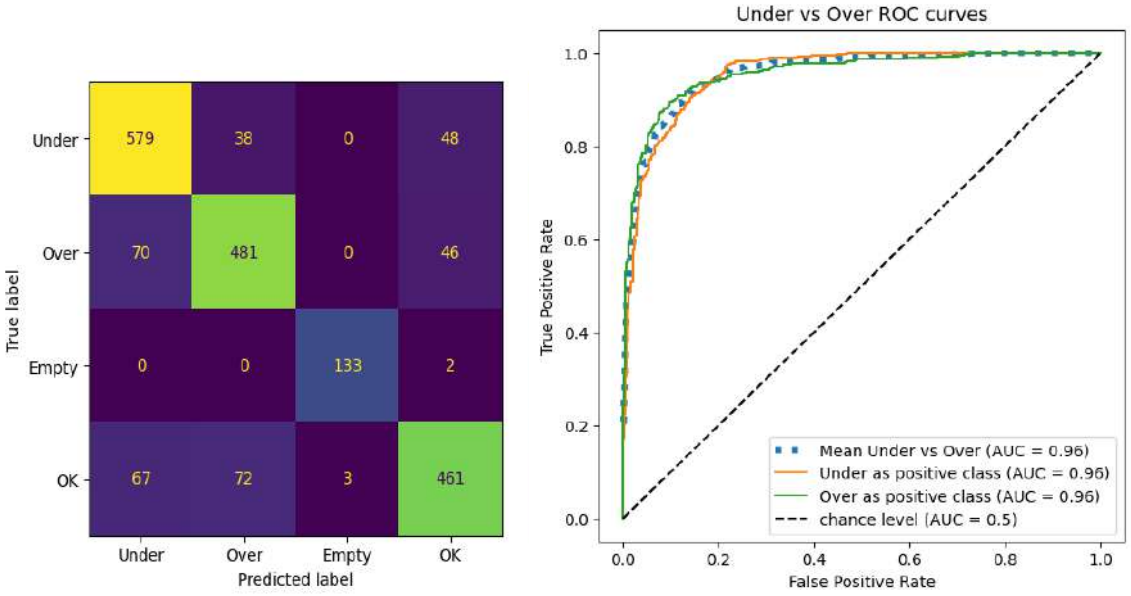


Figura 6.14: Imágenes transformadas con algunas transformaciones complejas de PyTorch



(a) Matriz de confusión

(b) Curva ROC

Figura 6.15: Resultados del entrenamiento con algunas transformaciones complejas de PyTorch

6.3.3. Transformaciones complejas librería imgaug

Por último, se ha recurrido a la biblioteca pública Imgaug, la cual permite transformaciones más complejas de las disponibles en PyTorch. Se han hecho algunas modificaciones como el desenfoque gausiano, rotación, eliminar píxeles o saturar la imagen. Además, esta biblioteca permite algunos comportamientos, como el comando "sometimes" para solo afectar a algunas imágenes del dataset con la probabilidad deseada.

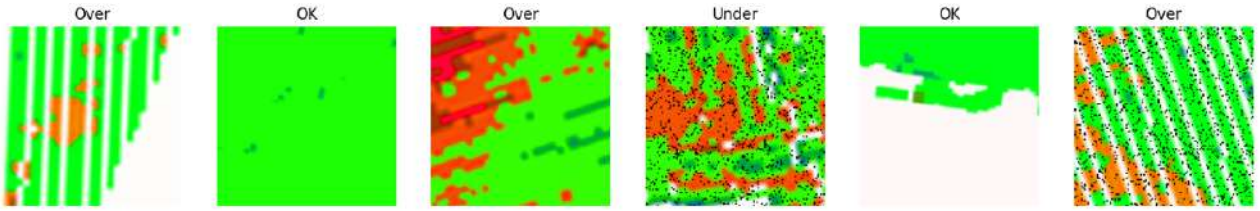
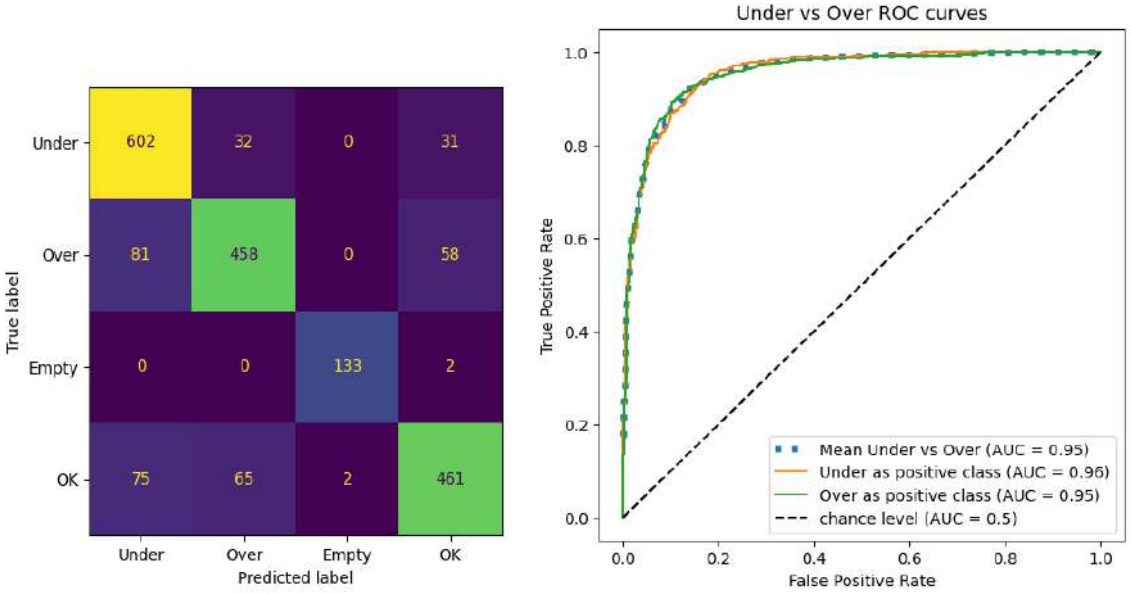


Figura 6.16: Imágenes transformadas con transformaciones de la biblioteca Imgaug



(a) Matriz de confusión **(b)** Curva ROC

Figura 6.17: Resultados del entrenamiento con algunas transformaciones de la biblioteca Imgaug

6.3.4. Resumen de resultados y conclusiones

En la Tabla 6.9 se resume el rendimiento con las diferentes transformaciones a imágenes. Al estudiar esta tabla, así como los resultados y gráficas anteriores, se puede concluir que la aumentación de datos no supone una mejora relevante para este modelo, ya que los modelos con mayores transformaciones no han solucionado problemas de clasificación ni de rendimiento general del modelo. Se puede razonar la razón tras estos hechos, ya que el *dataset* con el que se trabaja ya cuenta con suficientes imágenes como para ofrecer el buen resultado con el que contábamos de casi un 83 % en validación. Además, imágenes como las que se encuentran en este conjunto de datos ya son lo suficientemente simples, puesto que la clasificación se basa en detectar los colores verde, rojo y azul. Otros conjuntos de datos se podrían beneficiar de este tipo de complejas transformaciones si las imágenes cuentan con demasiados detalles que el modelo no es capaz de detectar, como imágenes reales, u otras en las que debe detectar elementos muy particulares, como detección de tumores.

Transformaciones	Max Acc Val	Max Acc Train	Tiempo
Trans. simples PyTorch	83.25 %	85.85 %	24’28”
Trans. complejas PyTorch	83.15 %	84.41 %	42’12”
Trans. Imgaug	83.00 %	86.1 %	63’56”

Tabla 6.9: Resultados entrenamientos para aumentación del conjunto de datos

En definitiva, se escogerán las transformaciones simples de PyTorch, puesto que estas no han supuesto una pérdida aún mayor de tiempo de entrenamiento como las otras dos y han dado el mejor resultado.

6.4. Variación de hiperparámetros

Como se ha explicado en la sección 5.7 existen dos parámetros a optimizar en este apartado; el valor de la tasa de aprendizaje inicial y la variación de la tasa de aprendizaje en el entrenamiento. Se ha comenzado entrenando el modelo con diferentes funciones de optimización y diferentes valores de tasa de aprendizaje (lr) para asegurar con qué valor funciona mejor el modelo y empezar con este. Tras esto, se han utilizado las funciones StepLR y ExponentialLR para variar la tasa de aprendizaje y comprobar con qué valores de variación el modelo aprende mejor. Cabe destacar que no se ha considerado el tiempo de ejecución a la hora de considerar el mejor optimizador, ya que todos han ofrecido resultados temporales parecidos durante las simulaciones.

6.4.1. Tasa de aprendizaje inicial

Se recomienda utilizar una tasa de aprendizaje de entre 0.1 y 0.001 como comienzo y comprobar si el problema mejora o no con una menor [96], por lo que se ha comenzado con una tasa de aprendizaje de 0.1 y se ha disminuido hasta comprobar que el modelo no mejoraba.

SGD

En la Figura 6.18 se han utilizado las tasas de aprendizaje 0.1, 0.05, 0.005 y 0.0005, hasta comprobar que el modelo no mejoraba. La mejor precisión de validación se ha conseguido con una tasa de 0.005, que será la que se utilice en el próximo apartado.

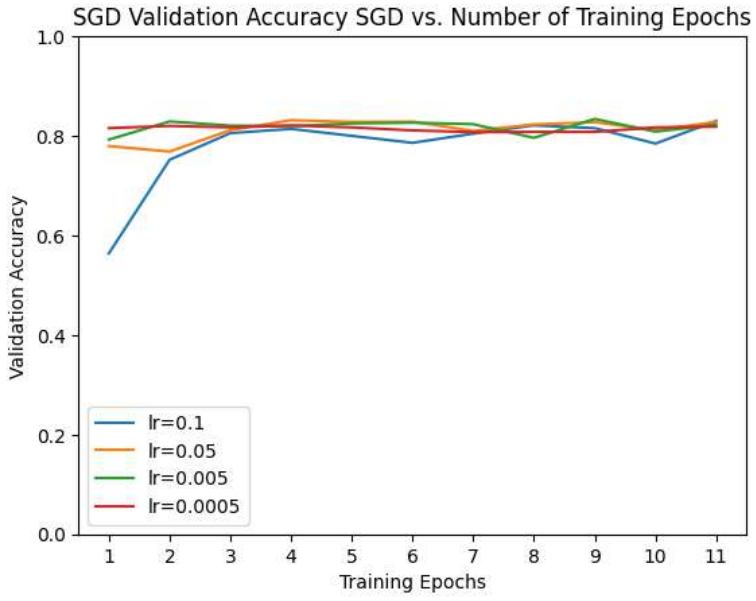


Figura 6.18: Precisión de validación con el optimizador SGD y diferentes tasas de aprendizaje

Adam

Se han utilizado las tasas de aprendizaje 0.1, 0.05 y 0.005, finalizando en este valor, ya que, como se comprueba en la Figura 6.19, se consiguen los mejores resultados con un valor de 0.1.

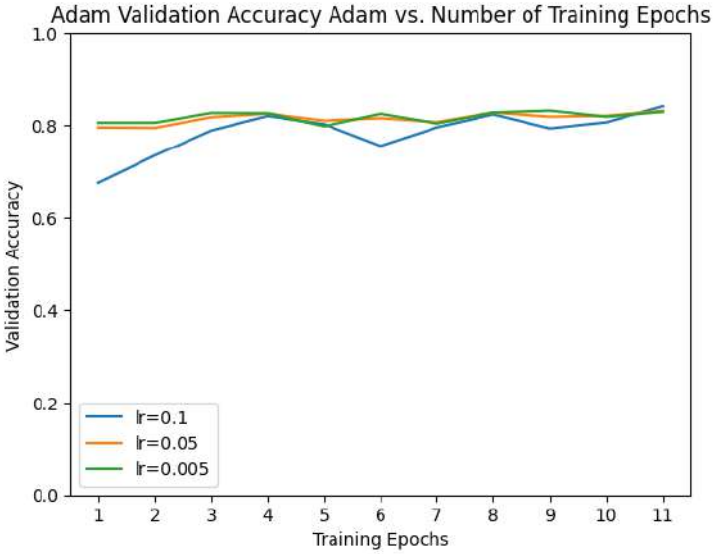


Figura 6.19: Precisión de validación con el optimizador Adam y diferentes tasas de aprendizaje

Adagrad

Al igual que con el optimizador Adam, se ha conseguido la mejor respuesta con $lr = 0{,}05$, habiendo hecho pruebas con 0.1, 0.05, 0.005, 0.0005, graficados en la Figura 6.20.

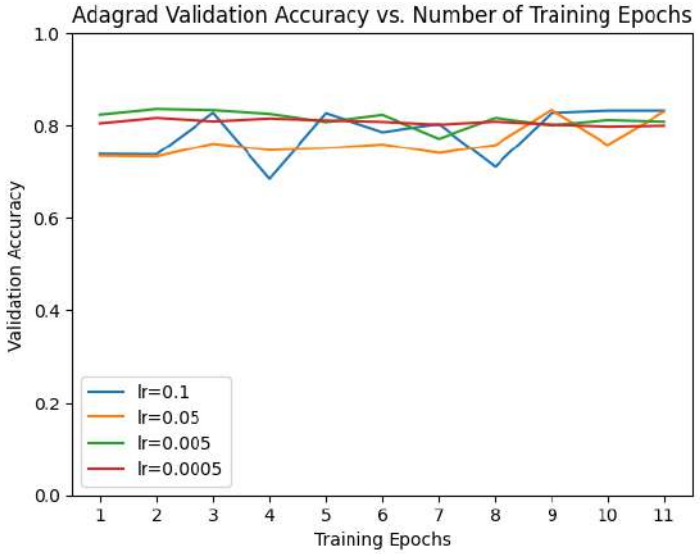


Figura 6.20: Precisión de validación con el optimizador Adagrad y diferentes tasas de aprendizaje

Adadelta

En la Figura 6.21 se observa que se han utilizado las tasas de aprendizaje 0.1, 0.05, y 0.005, hasta comprobar que el modelo no mejoraba, siendo 0.05 la mejor. Además, debido a otros proyectos donde Adadelta tiene buenos resultados con tasas de aprendizajes altas, se han intentado con $lr = 10$, aunque la respuesta no ha mejorado.

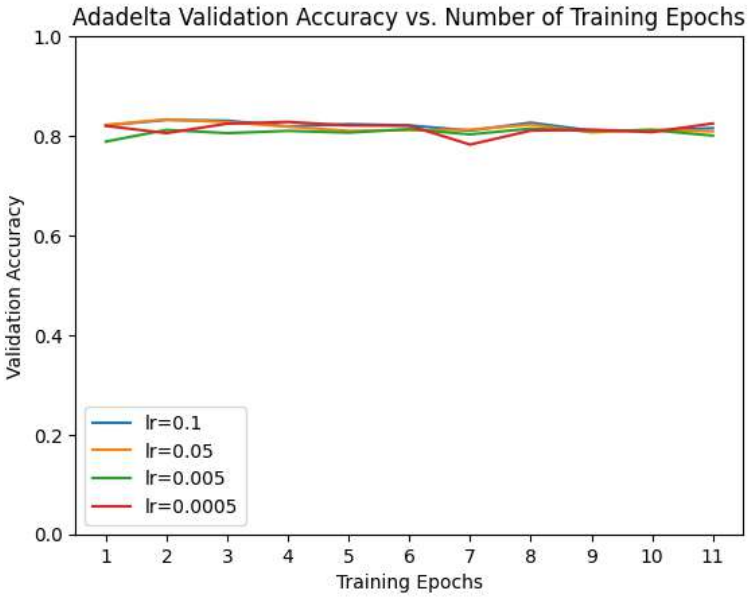


Figura 6.21: Precisión de validación con el optimizador Adadelta y diferentes tasas de aprendizaje

Resumen de resultados

En la Tabla 6.10 se observan resumidos los mejores resultados de precisión de validación obtenidos por cada modelo.

Func. Optimización	lr = 0.1	lr = 0.05	lr = 0.005	lr = 0.0005	lr = 10
SGD	83.00 %	83.15 %	83.40 %	82.10 %	-
Adam	84.05 %	83.05 %	82.85 %	- %	-
Adagrad	83.10 %	83.30 %	83.45 %	81.55 %	-
Adadelta	83.20 %	83.30 %	81.45 %	-	82.80 %

Tabla 6.10: Resultados del entrenamiento para variación de la tasa de aprendizaje con diferentes optimizadores

6.4.2. Variación de tasa de aprendizaje

A continuación, se han resumido los resultados obtenidos para cada optimizador con diferentes variaciones de tasa de aprendizaje. Para todos los casos se han utilizado las dos funciones de variación de tasa de aprendizaje StepLR y ExponentialLR vistos en la Sección 5.7.2, utilizando como tasa

de aprendizaje inicial la que haya ofrecido un mejor resultado en el apartadon anterior y variándola de diferente forma. En estas simulaciones, se ha cambiado el número de épocas a treinta para poder comprobar con mayor facilidad si el modelo mejora con la variación de tasa de aprendizaje.

SGD

Se ha comenzado utilizando la función StepLR con un valor inicial de tasa de aprendizaje de 0.005 y de 0.3 para gamma (γ), reduciendo la tasa de aprendizaje cada dos épocas, y manteniendo el $momentum$ constante en 0.9 para esta y todas las simulaciones. En esta primera simulación, el modelo demuestra que los cambios son demasiados grandes como para mejorar, por lo que se ha vuelto a probar con una tasa de aprendizaje de 0.005. Se ha cambiado la variación a cada 3 épocas, y se ha aumentado γ a 0.6, reduciendo así la variación de la tasa de aprendizaje. Este modelo tampoco aporta mejoras con respecto a mantener la tasa constante.

Func. Varianza	γ	N.º épocas para cambio	Tasa de aprendizaje	Val. Acc.
StepLR	0.3	2	0.005	82.15 %
StepLR	0.6	3	0.005	83.20 %

Tabla 6.11: Resultados para SGD con varianza de la tasa de aprendizaje escalada

A continuación, se ha utilizado la función ExponentialLR, variando la tasa de aprendizaje exponencialmente con $\gamma = 0{,}6$. De nuevo, el resultado es parecido al conseguido con la tasa de aprendizaje constante, sin mejorarlo,

Func. Varianza	γ	Tasa de aprendizaje	Val. Acc.
ExponentialLR	0.6	0.005	83.35 %

Tabla 6.12: Resultados para SGD con varianza de la tasa de aprendizaje exponencial

Adam

En este caso, se ha comenzado con un valor de 0.1 para la tasa de aprendizaje, puesto que era el valor que ofrecía una mejor precisión en el apartado anterior. Se han utilizado descensos con StepLR de $\gamma = 0{,}1$ y $\gamma = 0{,}8$ cada 3 y 5 épocas, de forma que se tenga un descenso más rápido y otro más lento. Por último, también se prueba con una tasa de aprendizaje inicial de 0.005, ya que esta tenía buenos resultados. Se le aplica un descenso escalonado con $\gamma = 0{,}1$ cada 3 épocas y se consigue una precisión superior a la constante con esta tasa de aprendizaje, pero no supera a los resultados ofrecidos con la de tasa de aprendizaje constante en 0.1.

Func. Varianza	γ	N.º épocas para cambio	Tasa de aprendizaje	Val. Acc.
StepLR	0.1	3	0.1	83.70 %
StepLR	0.8	5	0.1	83.10 %
StepLR	0.1	3	0.005	83.90 %

Tabla 6.13: Resultados para Adam con varianza de la tasa de aprendizaje escalada

En ninguno de los casos anteriores se consigue una precisión superior a la conseguida con la tasa

de aprendizaje constante, probando además con una bajada exponencial con $\gamma = 0{,}1$.

Func. Varianza	γ	Tasa de aprendizaje	Val. Acc.
ExponentialLR	0.1	0.005	83.40 %

Tabla 6.14: Resultados para Adam con varianza de la tasa de aprendizaje exponencial

Adagrad

Se han utilizado dos tasas de aprendizaje de partida, 0.05 y 0.005, puesto que ambas tenían resultados parecidos en el apartado anterior, aplicando para ambas un descenso cada 5 épocas de $\gamma = 0{,}1$. En ambos casos, se ha conseguido mejorar la precisión que tenía el modelo con tasa de aprendizaje fija, aunque levemente.

Func. Varianza	γ	N.º épocas para cambio	Tasa de aprendizaje	Val. Acc.
StepLR	0.1	5	0.05	83.50 %
StepLR	0.1	5	0.005	83.75

Tabla 6.15: Resultados para Adagrad con varianza de la tasa de aprendizaje escalada

Además, se ha utilizado una bajada exponencial con $\gamma = 0{,}9$ y tasa de aprendizaje inicial 0.005, consiguiendo la mejor precisión para el optimizador Adagrad hasta ahora, 83.95 %.

Func. Varianza	γ	Tasa de aprendizaje	Val. Acc.
ExponentialLR	0.1	0.005	83.95 %

Tabla 6.16: Resultados para Adagrad con varianza de la tasa de aprendizaje exponencial

Adadelta

Se ha comenzado utilizando una tasa de aprendizaje inicial de 0.05, puesto que es la que ofrece una mejor respuesta, con $\gamma = 0{,}1$ cada 3 épocas, consiguiendo una precisión algo superior a la vista con la tasa fija. Al comprobar que el modelo mejoraba con menores tasas, se ha vuelto a probar con una tasa de aprendizaje de 0.005, en este caso variando la tasa cada 3 épocas con $\gamma = 0{,}6$, consiguiendo la mejor precisión con el optimizador Adadelta.

Func. Varianza	γ	N.º épocas para cambio	Tasa de aprendizaje	Val. Acc.
StepLR	0.1	3	0.05	83.35 %
StepLR	0.6	3	0.005	83.75

Tabla 6.17: Resultados para Adadelta con varianza de la tasa de aprendizaje escalada

También se ha probado una bajada exponencial con $\gamma = 0{,}1$ y tasa de aprendizaje inicial 0.05.

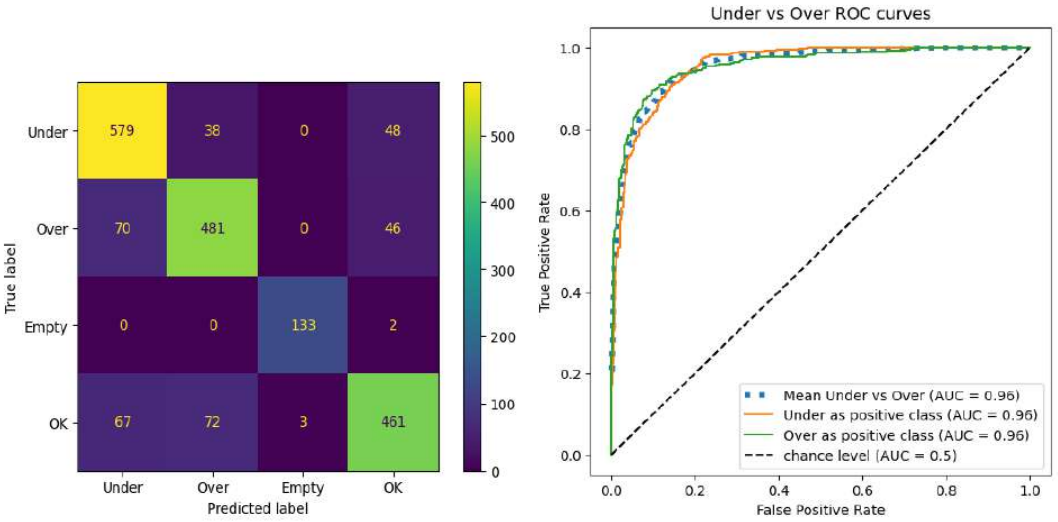
Func. Varianza	γ	Tasa de aprendizaje	Val. Acc.
ExponentialLR	0.1	0.05	83.38 %

Tabla 6.18: Resultados para Adadelta con varianza de la tasa de aprendizaje exponencial

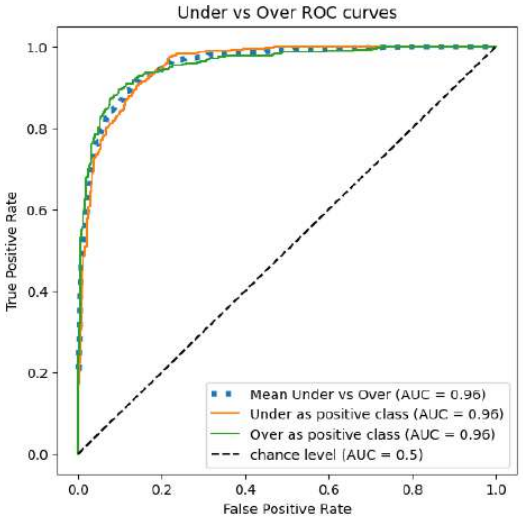
6.4.3. Conclusiones

Se puede observar que en la mayoría de los casos anteriores, una variación de la tasa de aprendizaje durante el entrenamiento mejora la respuesta del modelo, aunque en estos casos, esta mejora no es de un gran nivel. Se puede concluir que el modelo ya se encuentra en un punto en el que es complicado ofrecer mejoras, con lo que aunque las mejoras sean pequeñas, estas pequeñas subidas de precisión no son triviales. Para mejorar la respuesta del modelo se debería recurrir a otras estrategias como aumentar y mejorar el dataset o la validación cruzada.

Se ha decidido considerar como óptimo el modelo con el optimizador Adagrad y una bajada de la tasa de aprendizaje exponencial, puesto que consigue una buena precisión en todos sus resultados, llegando a ofrecer casi un 84 %. Además, se han graficado sus curvas ROC y matriz de confusión (ver Figura 6.22), donde se observa que cumple en reconocer la clase *Under* frente a la clase *Over*.



(a) Matriz de confusión **(b)** Curva ROC OvR



(c) Curva OvO para *Under* frente a *Over*

Figura 6.22: Matriz de confusión y curvas ROC para modelo óptimo tras ajuste de hiperparámetros

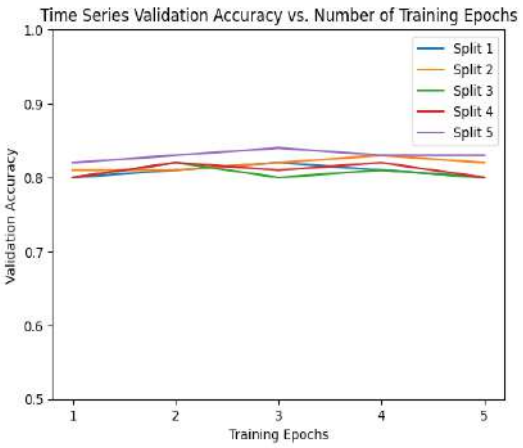
6.5. Validación cruzada

Como se ha explicado en la sección 5.9.1 se utilizarán las técnicas de validación cruzada aleatoria repetida, temporal y *k-folds*. Los códigos utilizados son parecidos a los usados en los apartados anteriores, consultando en ejemplos de GitHub para añadir la validación cruzada [97]. Además, los códigos están explicados en el Anexo de este texto.

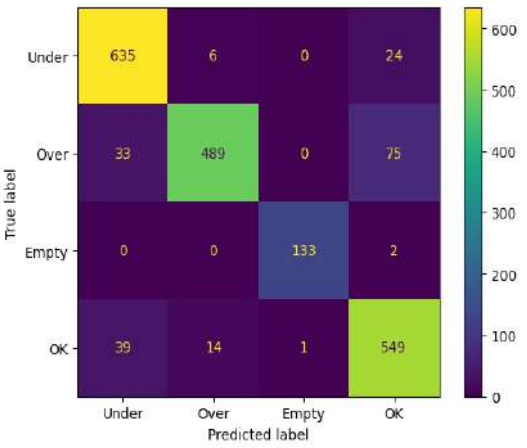
6.5.1. Aleatoria

Se comienza por la validación cruzada aleatoria, puesto que es la más simple, no siendo más que la validación *hold out* repetida durante diversas iteraciones, eligiendo diferentes imágenes del conjunto de datos al azar en cada una. La simpleza de este método se aprecia en la Figura 6.23 donde se pueden ver los resultados mediante la matriz de confusión y curvas ROC para OvR (*One vs. Rest*) y OvO (*One vs. One*) para la situación crítica de *Under vs. Over*. Además, se ha añadido una gráfica donde se dibujan las precisiones durante las épocas de cada división, iteración o *split* del modelo.

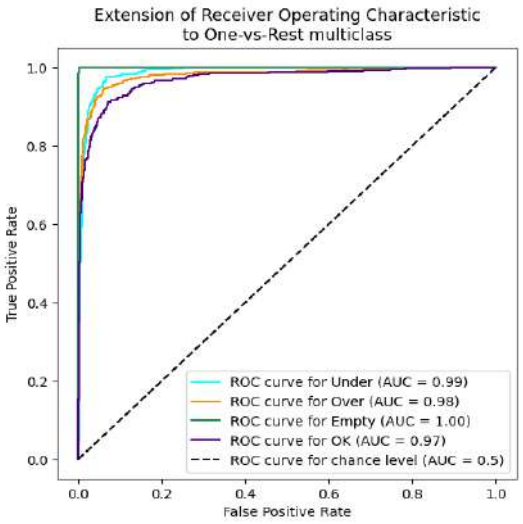
Se observa que el modelo tiene buenos resultados, aunque no consigue superar la precisión que ya ofrecían los modelos sin validación cruzada. Sin embargo, en las curvas ROC se aprecia como el modelo es capaz de mejorar la clasificación de la situación crítica (*Under vs. Over*), lo cual puede ser debido a que en este caso el modelo está viendo un mayor número de imágenes, con lo que es probable que consiga diferenciar las características de estas clases con una mayor facilidad. También se puede ver que las divisiones tienen prácticamente las mismas precisiones, lo cual es lógico dado que las divisiones son aleatorias.



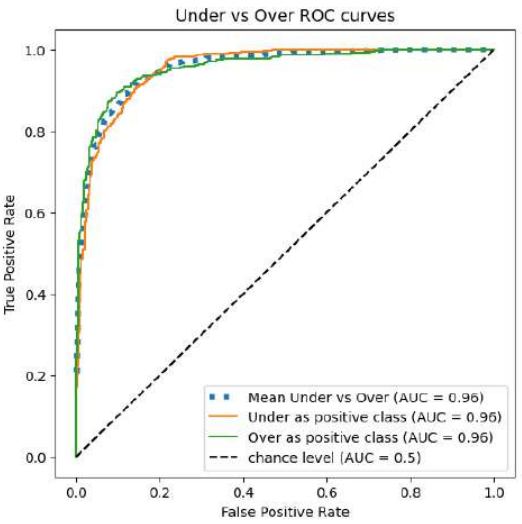
(a) Precisión en las iteraciones



(b) Matriz de confusión



(c) Curva ROC OvR



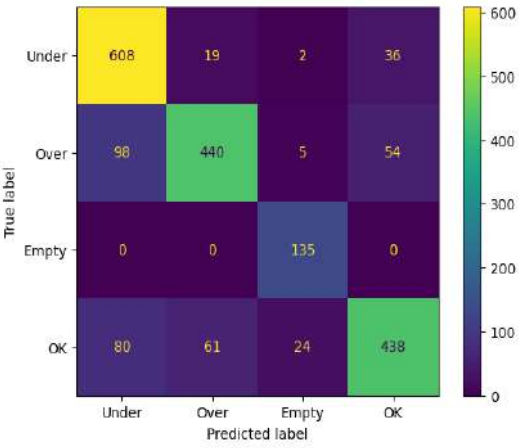
(d) Curva OvO para *Under* frente a *Over*

Figura 6.23: Precisión, matriz de confusión y curvas ROC para modelo con validación cruzada aleatoria

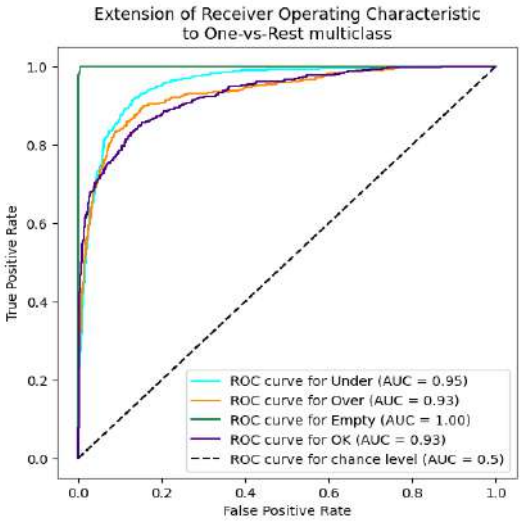
6.5.2. Temporal

A continuación, se ha utilizado la validación cruzada mediante series temporales, donde aumenta el tamaño del *dataset* utilizado, tanto en entrenamiento como en validación, para cada iteración. En la Figura 6.24 se aprecia como en las primeras iteraciones la precisión es algo menor a lo esperado, lo cual tiene sentido, dado que durante estas iteraciones el modelo se comporta como si se estuviera entrenando con un conjunto de datos menor. Si bien el modelo consigue una buena precisión, es visible tanto en la matriz de confusión como en las curvas ROC que el modelo no consigue clasificar las imágenes tan bien como la validación cruzada aleatoria. Este tipo de validación cruzada tiene un menor coste de computación y puede ser aplicable con buenos resultados para otro tipo de modelo y conjunto de datos, pero en el caso actual no es una buena solución. Sin embargo, sí que demuestra que el modelo no necesita de un conjunto de datos tan grande como el que se está utilizando para conseguir una buena precisión, si se utiliza validación cruzada, dado que en la segunda o tercera iteración, donde el *dataset* es menor, ya se consigue una buena precisión.

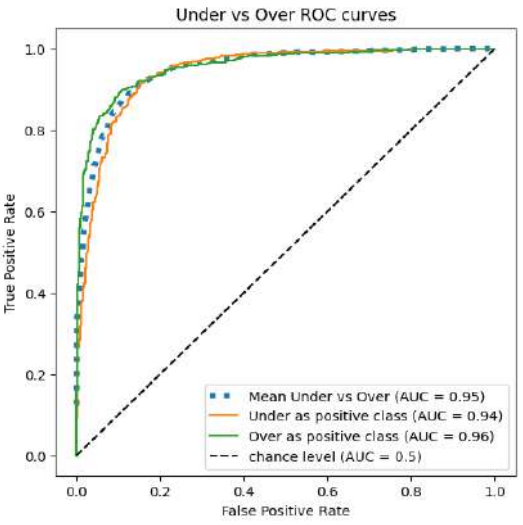
(a) Precisión en las iteraciones



(b) Matriz de confusión



(c) Curva ROC OvR

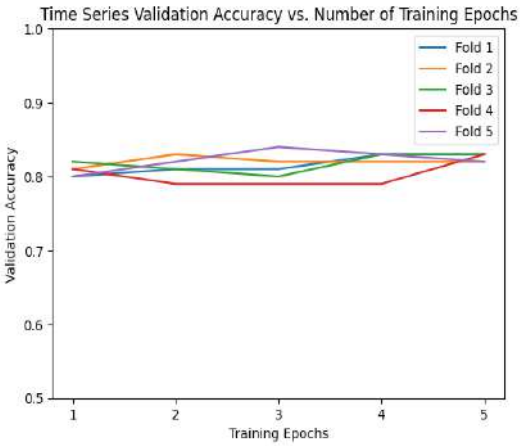


(d) Curva OvO para *Under* frente a *Over*

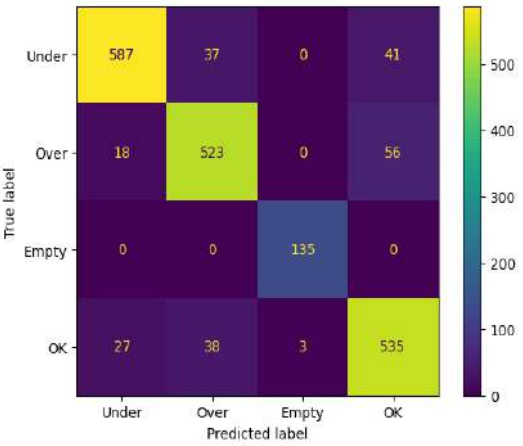
Figura 6.24: Precisión, matriz de confusión y curvas ROC para modelo con validación cruzada con series de tiempo

6.5.3. *K-folds*

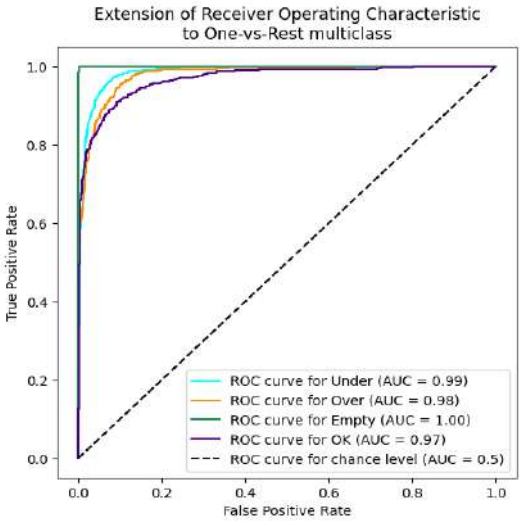
Por último, se ha utilizado la validación cruzada mediante *k-folds*, donde se divide el conjunto de entrenamiento en las divisiones deseadas y cada una de ellas actuará como conjunto de validación en una iteración. En la Figura 6.25 se aprecia como la precisión es muy buena, pero parecida a las conseguidas anteriormente. Sin embargo, donde esta validación se diferencia al resto como el mejor modelo hasta ahora, es en la capacidad de clasificación que se demuestra en la matriz de confusión y curvas ROC. Como se puede apreciar en estas, se ha conseguido el mejor resultado hasta ahora para la clase *Under* frente a *Over*, que es precisamente uno de los principales objetivos que se buscan. Al igual que con la validación aleatoria, esta mejora se puede deber a que ahora se están utilizando más imágenes en validación y el modelo puede aprender las características de cada clase mejor. De esta forma, se comprueban las ventajas de la validación cruzada, y en concreto, de la validación cruzada con *k-folds*.



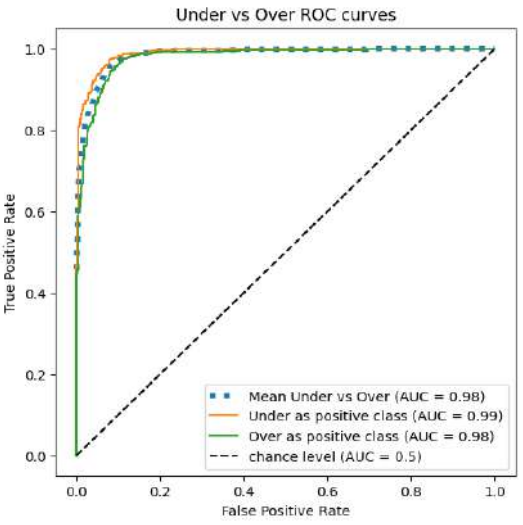
(a) Precisión en las iteraciones



(b) Matriz de confusión



(c) Curva ROC OvR



(d) Curva OvO para *Under* frente a *Over*

Figura 6.25: Precisión, matriz de confusión y curvas ROC para modelo con validación cruzada *k-folds*

6.6. Discusión

En este último apartado, se discutirán los diferentes modelos probados, recapitulando las técnicas utilizadas y resultados obtenidos. Además, se recogerán y mostrarán algunos ejemplos de imágenes mal clasificadas por parte del modelo, de forma que se puedan encontrar qué tipo de imágenes el modelo no es capaz de reconocer para intentar buscar maneras de mejorarlo.

Se comenzó comparando modelos desde cero o *scratch* frente a modelos preentrenados, donde se comprobó que los modelos preentrenados actuaban mejor que los *scratch* para un menor número de imágenes. Además, los modelos preentrenados también conseguían mejores resultados, especialmente en cuanto a velocidad de entrenamiento frente a los *Scratch*, gracias al concepto de aprendizaje transferido. Para estas simulaciones se utilizó el modelo SqueezeNet, puesto que era uno de los más simples de los modelos preentrenados.

Tras esto, se compararon los principales modelos preentrenados basados en redes neuronales convolucionales: AlexNet, ResNet, SqueezeNet, VGG, Inception y DenseNet. Se realizaron simulaciones con todos ellos y se decidió que el modelo preentrenado ResNet era el que ofrecía mejores resultados, por lo que se continuó utilizándolo durante el resto de secciones. De estos experimentos se puede concluir que los modelos simples ofrecían resultados suficientemente buenos debido a que las imágenes del *dataset* son bastante sencillas, diferenciándose entre clases principalmente por colores.

En el siguiente apartado se realizaron experimentos basándose en la aumentación de datos, para lo que se sometieron las imágenes a tres tipos de transformaciones. Se comenzó por algunas simples propias de la librería PyTorch, como giros o recortes, seguidas de otras más complejas de la librería PyTorch, como desenfoques gaussianos o cambios en el brillo, contraste o saturación. Por último, se utilizaron otras disponibles en la librería Imgaug, como la eliminación de píxeles al azar. En este caso, se comprobó que con las transformaciones simples de PyTorch era como el modelo conseguía un mejor resultado, probablemente debido de nuevo a la simpleza de las imágenes, que no necesitan grandes transformaciones para que el modelo reconozco las características de cada clase.

A continuación, se utilizaron diferentes modelos con diferentes valores de tasa de aprendizaje y funciones de optimización. En este caso, se comprobó como los cambios de estos valores pueden provocar que el modelo consiga mejores o peores resultados, si bien no se consiguió mejorar el resultado en una gran medida, dado que ya se estaba entrenando al modelo con valores de tasa de aprendizaje y función de optimización que ofrecían una gran precisión.

Por último, se han utilizado los tipos de validación cruzada: aleatoria repetida, series temporales y *k-folds*. Se ha comprobado como la validación cruzada puede dar mejores resultados que una validación de tipo *hold out*, al exponer al modelo a más imágenes, consiguiendo los mejores resultados de todos los entrenamientos con la validación cruzada *k-folds*.

6.6.1. Imágenes mal clasificadas

Para terminar con esta sección dedicada a los resultados, se ha decidido que es de interés guardar las imágenes mal clasificadas por el modelo y mostrarlas, de modo que se pueda tener una mejor idea de qué tipo de imágenes el modelo tiene problemas para reconocer y así poder mejorar la precisión. De esta forma, en las Figuras 6.26, 6.27 y 6.28 se pueden ver las imágenes mal clasificadas cuyo valor real es *Ok*, *Under* y *Over*, respectivamente.

Comenzando por las imágenes *Ok* en la Figura 6.26, se puede ver que la mayoría de estos fallos se encuentran en la forma de clasificar las imágenes del *dataset*, es decir, en las etiquetas verdaderas dadas por los profesionales que han creado el *dataset*. Muchas de estas figuras están bastante cerca de lo que se podría considerar *Under* u *Over*, lo que se reconoce por los colores, pero la imagen ha sido clasificada como *Ok*. En este caso, no supone un problema que el modelo tenga otro umbral diferente al establecido al realizar la clasificación, puesto que el modelo estaría estableciendo como *Over* o sobreimpresas partes de la figura que, aunque no establecidas como tal, no serían afectadas por un postprocesado (incluso beneficiándose).

Otro tipo de fallos que también se verán más adelante son los fallos debido a imágenes complicadas de clasificar. Un ejemplo de esta es la primera subfigura de la Figura 6.26, donde el modelo la ha clasificado como *Under* debido a una clara zona donde se necesita más material. Sin embargo, la superficie cuenta también con una parte sobreimpresa. En este caso, este tipo de imágenes con superficies tan irregulares, debería dividirse en partes más pequeñas donde el proceso que necesiten sea claro. No es demasiado preocupante si las imágenes se clasifican bien por parte del modelo, puesto que se le podría aplicar un postprocesado y, en una segunda revisión, otro.

En el caso de esta primera subfigura de la Figura 6.26 también existe otro problema y es que la etiqueta verdadera dada por parte del *dataset* es *Ok*, cuando la Figura claramente no dispone de un buen acabado. En este caso, se comprueba que hay casos en el que las etiquetas verdaderas dadas por los profesionales no son correctas, lo cual es comprensible, teniendo en cuenta que se trata de un conjunto de más de 40 mil imágenes, pero se debe tener en cuenta al ver los resultados de precisión del modelo.

Por último, en estas imágenes también se comprueba un ejemplo de una imagen supuesta por el modelo como *Empty*, cuando realmente es *Ok* debido a una pequeña zona donde existe superficie. Este tipo de fallos, no suponen ningún tipo de relevancia a la hora de usar este modelo para encontrar defectos en superficies, pero debe tenerse en cuenta que pueden existir al contar los fallos dados por el modelo.

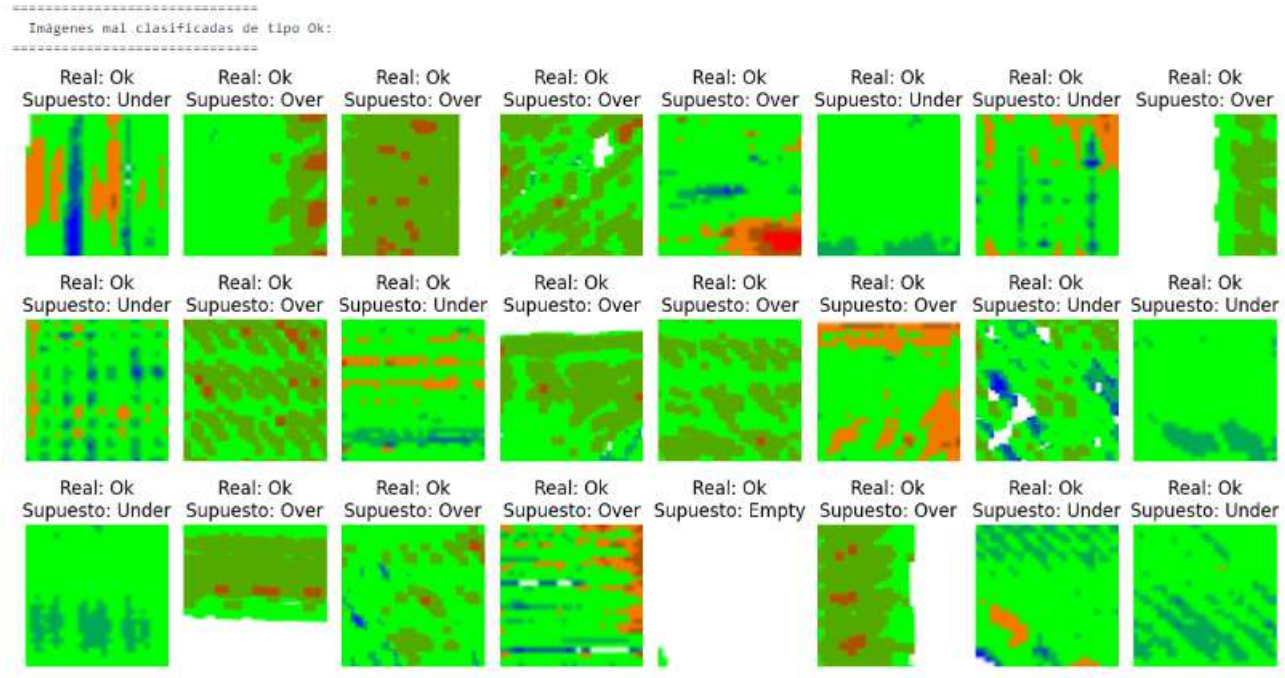


Figura 6.26: Imágenes *Ok* mal clasificadas

Se continúa con imágenes mal clasificadas del tipo *Under*, es decir, en las que falta impresión y necesitarían un llenado, mostradas en la Figura 6.27. Al igual que en los casos anteriores, se puede ver como la mayoría de imágenes son complicadas de clasificar, puesto que son muy irregulares, con partes con falta de material y otras con una sobreimpresión muy elevada. En estos casos, el modelo clasifica como *Over* cuando la etiqueta verdadera es *Under*, aunque ambos son correctos y se necesitaría una división de estas imágenes en otras menores con más regularidad en la superficie. De cualquier manera, en estos casos se podría realizar el postprocesado según lo que ha reconocido el modelo y tras esto hacer un segundo reconocimiento donde el modelo reconocerá la otra irregularidad, realizando un segundo postprocesamiento.

Al igual que con las imágenes de tipo *Ok* mal clasificadas, también existen casos en estas imágenes donde el límite para clasificarlas de una u otra forma es la causa del error. En estos casos no supondría un problema, ya que el modelo clasifica como *Ok* imágenes que son *Under*, pero cuya falta de material es muy baja.

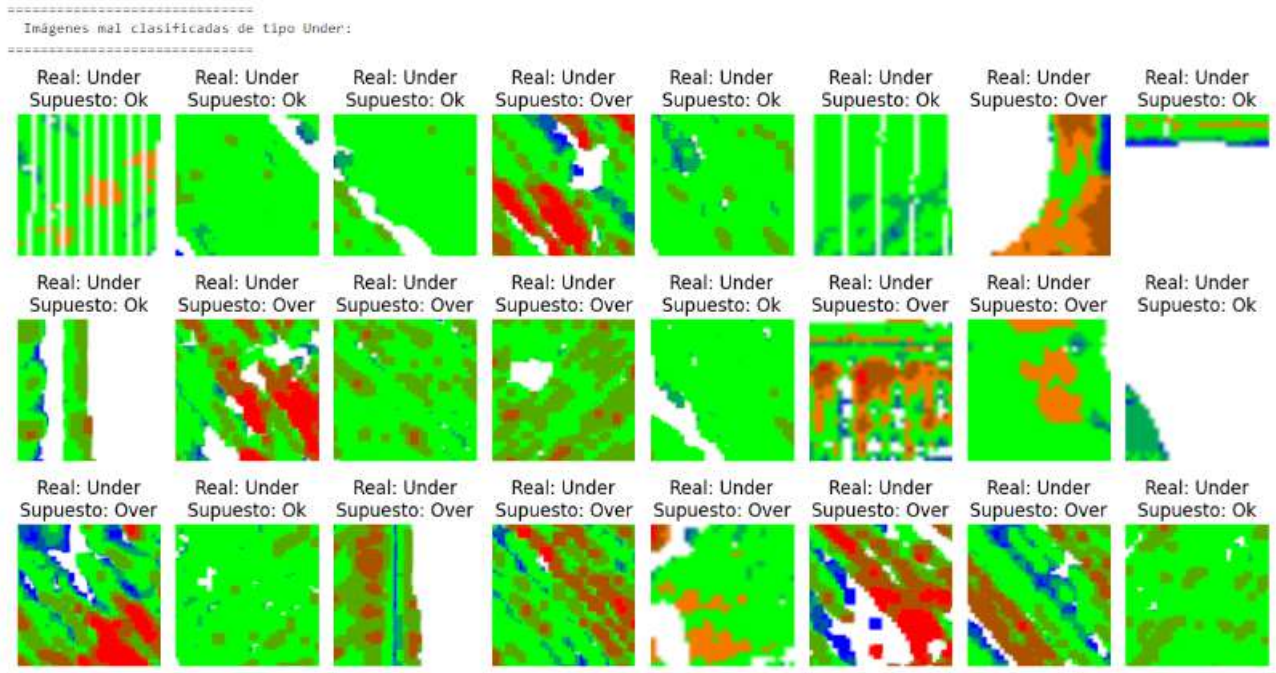


Figura 6.27: Imágenes *Under* mal clasificadas

Por último, en la Figura 6.28, se pueden ver imágenes de tipo *Over* mal clasificadas. Cabe destacar en este punto que no se han mostrado imágenes de tipo *Empty* mal clasificadas, puesto que, como se ha visto en los resultados del modelo, la clasificación de esta es prácticamente perfecta. Tan solo existe problema con la clasificación *Empty* cuando el modelo clasifica como *Empty* imágenes donde la mayor parte de la superficie está vacía con algo de superficie en los bordes. En estos casos, se trata de un problema de elegir la etiqueta para estas imágenes en un principio y no de clasificación del modelo.

Volviendo a las imágenes de la clase *Over* mal clasificadas, se puede ver que ocurren las mismas dos casuísticas que en las clases anteriores: existen imágenes complicadas para el modelo debido a la irregularidad, con zonas de sobreimpresión y falta de material, y existen imágenes al límite entre clases que el modelo clasifica como una y es otra, no siendo esto un problema debido a que es una cuestión del umbral a la hora de elegir una u otra clase.

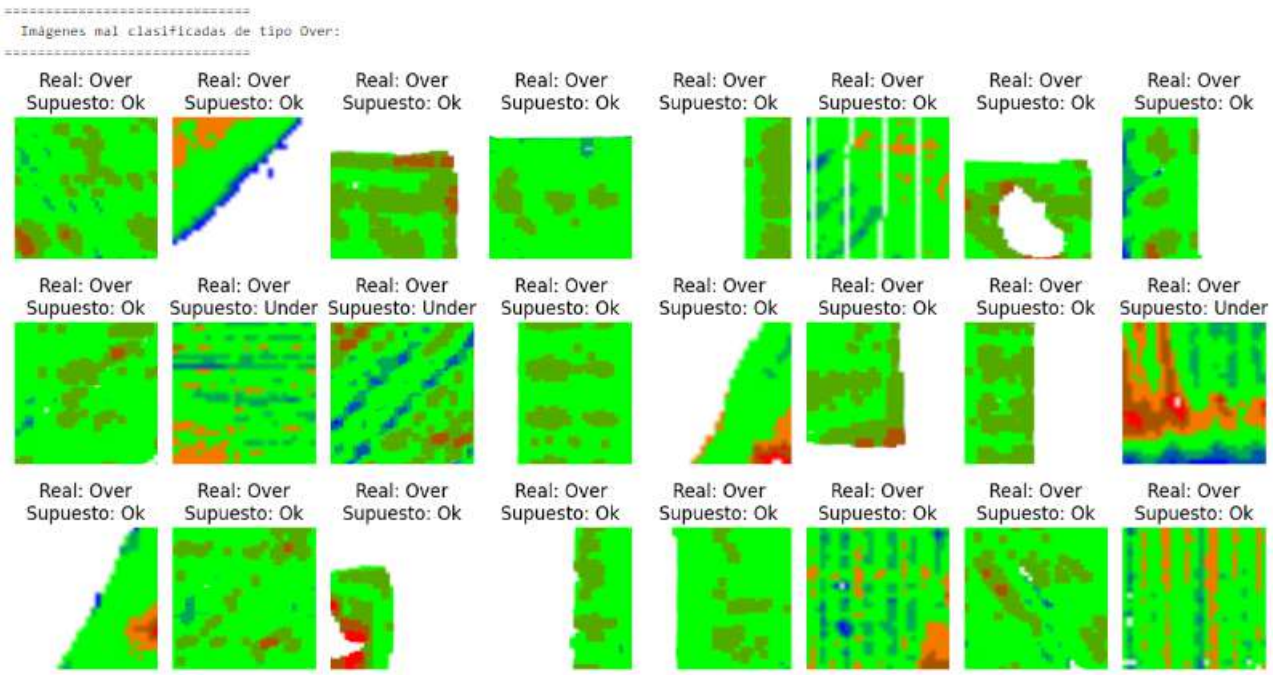


Figura 6.28: Imágenes *Over* mal clasificadas

Como se han visto en los casos anteriores, el modelo comete errores, sobre todo en imágenes irregulares con zonas de diferentes tipos y que el modelo elige como una clase equivocada, viendo un ejemplo en la Figura 6.29. En estos casos, el error se solucionaría dividiendo la superficie total impresa en zonas más pequeñas, de modo que estos cambios drásticos se solucionen. Por otra parte, si no se puede cambiar el dataset, se podría tomar una solución a la hora de realizar el postprocesamiento, realizando dos etapas de clasificación de la superficie y postprocesado, asegurando que en el caso de imágenes con dos tipos de zonas, ambas se puedan tratar.

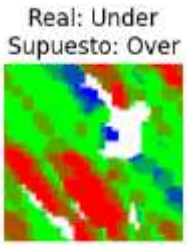


Figura 6.29: Imagen mal clasificada del tipo zona irregular

En la Figura 6.30 se puede ver otro tipo de errores, aquellos en los que la imagen se encuentra en el límite entre dos clases, clasificando el modelo la imagen en una clase que no es la propia. Este tipo de errores no son necesariamente preocupantes, puesto que el modelo estaría clasificando como *Over* o *Under* imágenes que son *Ok* o al revés, pero que podrían estar en una u otra clase según el límite puesto entre estas al crear el *dataset*. El resultado de este error consistiría en someter a un postprocesamiento a superficies que no lo necesitan del todo, pero que igualmente se podrían beneficiar de ello, o en ignorar en este postprocesamiento algunas imágenes que lo necesitarían, pero que no supondrían un gran fallo en el acabado de la superficie. En este caso, dependería de cada objeto que se quiera manufacturar, según cuál es el límite que se desea imponer, de forma

que si la exactitud del acabado es primordial, se reduzca el número de imágenes *Ok* en el *dataset*, es decir, que en caso de duda entre dos clases, se decida por las clases *Under* u *Over* en vez de *Ok*, puesto que se estará dando al modelo una mayor libertad para reconocer como estas clases, las cuales aseguran que exista un postprocesado. Al contrario, si lo que se está primando es la velocidad, se puede crear el *dataset* con un mayor número de imágenes *Ok* y que tan solo se procesen algunas superficies claramente necesitadas de ello.

(a) *Over* **(b)** *Under*

Figura 6.30: Imagen mal clasificada del tipo límite entre clases

Otro error encontrado son las imágenes mal clasificadas, como es el caso de las de la Figura 6.31. En estos casos, la solución es mejorar el *dataset* y asegurar que las imágenes están bien clasificadas, puesto que estos errores suponen un inconveniente al obtener los resultados del modelo.

(a) Imagen claramente no *Ok* **(b)** Imagen claramente no *Under*

Figura 6.31: Imagen mal clasificada del tipo etiqueta mal elegida

Por último, también se han visto otro tipo de errores, como el que se ve en la Figura 6.32, donde una imagen prácticamente vacía se clasifica como *Empty*, aunque al tener algo de superficie su etiqueta verdadera es otra. En este caso, la solución sería arreglar las etiquetas del *dataset* como en el anterior, si bien este tipo de imagen se ha decidido meter en otro conjunto, puesto que imágenes de este tipo no supondrían ningún problema reales en el postprocesamiento, aunque sí afectan al resultado de la precisión del modelo.

Figura 6.32: Imagen mal clasificada de otro tipo

CAPÍTULO 7

CONCLUSIONES

En este proyecto se ha abordado el problema de la detección automática de artefactos o defectos en superficies fabricadas mediante manufactura aditiva utilizando redes neuronales profundas. A lo largo del estudio, se han llevado a cabo diversas etapas que han permitido obtener resultados significativos y concluyentes.

En primer lugar, se realizó una exhaustiva revisión de la literatura existente sobre la manufactura aditiva, así como de la inteligencia artificial y, en concreto, sobre las redes neuronales convolucionales. Esto proporcionó una base sólida para comprender los desafíos y enfoques previos en el campo de la detección de artefactos en la manufactura aditiva.

Posteriormente, se utilizó un conjunto de datos que fue recopilado y procesado, de forma que fuera representativo e incluyera muestras de superficies fabricadas mediante manufactura aditiva. Este conjunto de datos y su compresión fue fundamental para entender el proceso de entrenamiento de los modelos y las posibles mejoras.

Se diseñaron y entrenaron diferentes arquitecturas de redes neuronales. Se utilizaron redes neuronales preentrenadas, entendiendo sus ventajas frente a utilizar una arquitectura desde cero. También se explicaron y utilizaron conceptos para la mejora del rendimiento, como la aumentación de datos y la afinación de hiperparámetros. Estos modelos consiguieron un alto rendimiento en la detección de artefactos, demostrando las posibilidades de las redes neuronales en este campo.

Finalmente, se ha conseguido un modelo capaz de clasificar las diferentes secciones dentro de una pieza fabricada con manufactura aditiva con casi un 90 % de precisión. De esta forma, un modelo entrenado en apenas 40 minutos es capaz de establecer si las diferentes secciones de una pieza necesitan un postprocesamiento de lijado o rellenado de huecos o no.

En conclusión, este Trabajo de Fin de Grado ha demostrado la viabilidad y eficacia de utilizar redes neuronales profundas para la detección automática de artefactos en superficies fabricadas mediante manufactura aditiva. Los resultados obtenidos abren nuevas perspectivas en el campo de la calidad y control de la fabricación aditiva, brindando oportunidades para mejorar la eficiencia y precisión de los procesos de producción en esta industria.

CAPÍTULO 8

LÍNEAS FUTURAS

A pesar de los resultados logrados en este proyecto en la detección automática de artefactos utilizando redes neuronales profundas, existen aún varias áreas de investigación y desarrollo que pueden ser exploradas en futuros estudios. A continuación, se presentan algunas posibles líneas futuras de investigación:

- Aumento de entrenamientos: debido a que una gran parte del tiempo en este trabajo ha sido empleado en aprender conceptos de programación y redes neuronales por parte del alumno, la posibilidad de realizar entrenamientos y más comprobaciones se ha reducido. Además, la posibilidad de entrenar los modelos se ha visto supeditada a la disponibilidad de GPUs por parte de la plataforma Google Colab. Un mayor tiempo con la posibilidad de conectarse a procesadores propios permitiría realizar más entrenamientos y encontrar el modelo óptimo con una mayor seguridad.
- Mejora del *dataset*: si bien en este proyecto se ha hecho todo lo posible por conseguir modelos con la máxima precisión, se ha encontrado difícil de mejorar la precisión de validación llegado un límite. Un campo que, sin embargo, no se ha podido mejorar es el del conjunto de imágenes empleado. Si bien este conjunto de imágenes es amplio y ha sido clasificado con profesionalidad, no ha sido clasificado con total precisión, por lo que el modelo encuentra imágenes que no es capaz de entender. Llevar a cabo la construcción del dataset puede permitir mejorarlo y asegurar cómo son las imágenes de las que aprende el modelo.
- Mejora de la precisión: aunque los modelos de redes neuronales profundas desarrollados han demostrado un alto rendimiento en la detección de artefactos, siempre existe margen para mejorar la precisión. Se pueden explorar técnicas avanzadas de preprocesamiento de datos, arquitecturas de redes neuronales más complejas y algoritmos de aprendizaje más sofisticados para mejorar la capacidad de detección y reducir los falsos positivos y falsos negativos Además, se podrían utilizar otras técnicas de optimización explicadas en la memoria.
- Integración de técnicas de realimentación: actualmente, los modelos de redes neuronales profundas se entrenan utilizando conjuntos de datos estáticos. Sin embargo, en entornos de fabricación aditiva en tiempo real, las condiciones pueden cambiar y los artefactos pueden evolucionar durante el proceso de fabricación. Se pueden explorar técnicas de realimentación en línea que permitan actualizar y ajustar continuamente los modelos en función de los datos en tiempo real, mejorando así la capacidad de adaptación a las variaciones del proceso de fabricación.

- Validación experimental: aunque este estudio ha utilizado conjuntos de datos representativos para entrenar y evaluar los modelos, es importante realizar una validación experimental en entornos de fabricación aditiva reales. Esto implicaría la implementación de los modelos en sistemas de inspección en línea y la comparación de los resultados obtenidos con inspecciones manuales y métodos de control de calidad tradicionales.

Las líneas futuras de investigación pueden incluir la mejora de la precisión de detección con un mayor número de entrenamientos, la integración de técnicas de realimentación, la mejora del conjunto de datos, la validación experimental en entornos de fabricación aditiva reales, así como el uso de otras técnicas de optimización. Estas investigaciones adicionales contribuirán a avanzar en el campo de la detección automática de artefactos y mejorar la calidad y eficiencia de los procesos de fabricación aditiva.

Dado que este proyecto ha sufrido de un tiempo limitado y de un alumno con conocimientos previos limitados, los resultados obtenidos son simples aunque buenos. El aplicar estas líneas expuestas supondría la posibilidad de encontrar resultados aún mejores que demuestren el claro beneficio del reconocimiento de imágenes mediante redes neuronales profundas para detectar defectos en capas de manufactura aditiva.

APÉNDICE A

ANEXO I

A.1. Códigos python

Todos los códigos y sus resultados están disponibles en https://github.com/MrMartinMLB/TFG, así como la memoria de este proyecto, desarrollada con LaTeX, un sistema de preparación de documentos utilizado para la comunicación y publicación de documentos científicos. A continuación se exponen algunas partes fundamentales del código.

A.1.1. Librerías

```
#Se importan todas las librerias relevantes: torch, os, time, copy, random
from __future__ import print_function
from __future__ import division
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np
import torchvision
from torchvision import datasets, models, transforms
import matplotlib.pyplot as plt
import time
import os
import copy
from PIL import Image
import random
from torch.utils.data import random_split
from torch.utils.data import DataLoader
import torch.nn.functional as F

#Se ajusta una semilla para asegurar todos los resultados son iguales
torch.manual_seed(123)
torch.backends.cudnn.benchmark = False
torch.use_deterministic_algorithms(True)
random.seed(123)

#Se comprueba la version de Python y PyTorch
print("PyTorch Version: ",torch.__version__)
print("Torchvision Version: ",torchvision.__version__)
```

```
#Se importan las librerias necesarias para las representaciones graficas complejas
import pandas as pd
import seaborn as sns
from scipy import stats
from sklearn.metrics import confusion_matrix
from sklearn.metrics import roc_auc_score
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import RocCurveDisplay
from itertools import cycle
```

```
#Asegurar que trabajamos en la GPU
device = (torch.device('cuda') if torch.cuda.is_available()
          else torch.device('cpu'))
print(f"Training on device {device}.")
```

A.1.2. Creación de dataset

```
# Definir transformaciones para las imágenes
transform = transforms.Compose([
    transforms.Resize((78, 78)), #Se asegura todas las imagenes tienen la misma forma con un ajuste de tamaño
    transforms.ToTensor(),  # Convertir a tensor de PyTorch
])



# Crear dataset personalizado de PyTorch
class CustomDataset(torch.utils.data.Dataset):
    def __init__(self, root_dir):
        self.root_dir = root_dir
        self.classes = ['Under', 'Over', 'Empty', 'OK']
        self.imgs = []

        for i, cls in enumerate(self.classes):
            cls_path = os.path.join(self.root_dir, cls)
            for img_name in os.listdir(cls_path):
                img_path = os.path.join(cls_path, img_name)
                self.imgs.append((img_path, i))  # Añadir ruta de imagen y su label

        # Mezclar las imágenes antes de dividir el dataset
        random.shuffle(self.imgs)

    def __len__(self):
        return len(self.imgs)

    def __getitem__(self, idx):
        img_path, label = self.imgs[idx]
        img = Image.open(img_path).convert('RGB')
        img = transform(img)
        return img, label
```

```
#Se traen los archivos de Drive donde están las imágenes
from google.colab import drive
drive.mount('/content/drive')
```

```
dataset_1 = torch.utils.data.Subset(dataset, range(10000)) # Se crea un dataset del tamaño deseado apartir del total

# Definir tamaño de los conjuntos de entrenamiento y validación (80%-20%)
train_size = int(0.8 * len(dataset_1))
val_size = len(dataset_1) - train_size

# Dividir el dataset en conjuntos de entrenamiento y validación de manera aleatoria
train_data, val_data = random_split(dataset_1, [train_size, val_size])
```

```
# Creamos diccionario for training and validation
image_datasets = {'train': train_data, 'val': val_data}

# Creamos dataloaders for training and validation
batch_size = 8
dataloaders_dict = {x: torch.utils.data.DataLoader(image_datasets[x], batch_size=batch_size, shuffle=True, num_workers=2) for x in ['train', 'val']}
```

A.1.3. Comprobación dataset

```
# Imprimir el tamaño de los batches para los conjuntos de entrenamiento y validación
for phase in ['train', 'val']:
    dataloader = dataloaders_dict[phase]
    print(f'Tamaño de los batches para el conjunto de {phase}: {dataloader.batch_size}')
    print(f'Número de batches para el conjunto de {phase}: {len(dataloader)}')
    print(f'Número total de imágenes para el conjunto de {phase}: {len(dataloader.dataset)}')
    print('Clases:', dataset.classes)
    print()
```

```
#Asegurar el dataset está equilibrado para todas las clases
dataloader = torch.utils.data.DataLoader(val_data, batch_size=64,
                                         shuffle=True, num_workers=1)


label_count = {'OK': 0, 'Under': 0, 'Empty': 0, 'Over': 0}

for _, labels in dataloader:
    for label in labels:
        label_str = ['OK', 'Under', 'Empty', 'Over'][label]
        label_count[label_str] += 1

print(label_count)
```

```
# Obtener un batch de datos
images, labels = next(iter(dataloader))

# Visualizar las imágenes y etiquetas
fig, axs = plt.subplots(1, 20, figsize=(15, 3))
for i in range(20):
    axs[i].imshow(images[i].permute(1, 2, 0))
    axs[i].set_title(dataset.classes[labels[i]])
    axs[i].axis('off')
plt.show()
```

A.1.4. Definir el modelo

```
# Elegimos el modelo [resnet, alexnet, vgg, squeezenet, densenet, inception]
model_name = "squeezenet"

# Numero de clases en el dataset
num_classes = 4

# Tamaño del Batch (cambiar segun el tamaño de la memoria)
batch_size = 8

# Numero de epocas por las que se entrena
num_epochs = 50

# Booleano para indicar si se hace feature extraction (true) o fine tunning (false)
#   cuando es true solo se cambian algunos parametros al final
feature_extract = False
```

```python
#Se crea una clase para entrenar el modelo
def train_model(model, dataloaders, criterion, optimizer, num_epochs=25, is_inception=False):
    since = time.time() #Se comienza el cronometro

    #Se definen las variables para guardar los resultados del entrenamiento
    val_acc_history = []
    val_loss_history = []
    train_acc_history = []
    train_loss_history = []

    best_model_wts = copy.deepcopy(model.state_dict())
    best_acc = 0.0

    #Se crea un bucle para cada epoca
    for epoch in range(num_epochs):
        print('Epoch {}/{}'.format(epoch, num_epochs - 1))
        print('-' * 10)

        # Cada epoca tiene entrenamiento y validación
        for phase in ['train', 'val']:
            if phase == 'train':
                model.train()  # El modelo en entrenamiento
            else:
                model.eval()   # El modelo en validación

            #Se ajusta la perdida a 0 para cada epoca
            running_loss = 0.0
            running_corrects = 0

            # Dentro de cada epoca, se recorren los dataloaders.
            for inputs, labels in dataloaders[phase]:
                #Se envian los valores a la GPU
                inputs = inputs.to(device)
                labels = labels.to(device)

                # zero the parameter gradients
                optimizer.zero_grad()

                # forward
                # track history if only in train
                with torch.set_grad_enabled(phase == 'train'):
                    # Get model outputs and calculate loss
                    # Special case for inception because in training it has an auxiliary output. In train
                    #   mode we calculate the loss by summing the final output and the auxiliary output
                    #   but in testing we only consider the final output.
                    if is_inception and phase == 'train':
                        # From https://discuss.pytorch.org/t/how-to-optimize-inception-model-with-auxiliary-classifiers/7958
                        outputs, aux_outputs = model(inputs)
                        loss1 = criterion(outputs, labels)
                        loss2 = criterion(aux_outputs, labels)
                        loss = loss1 + 0.4*loss2
                    else:
                        outputs = model(inputs)
                        loss = criterion(outputs, labels)

                    _, preds = torch.max(outputs, 1)
```

```
                # Solo si se esta en entrenamiento se hace el calculo de optimizacion y backpropagation
                if phase == 'train':
                    loss.backward()
                    optimizer.step()

            # Estadisticas
            running_loss += loss.item() * inputs.size(0)
            running_corrects += torch.sum(preds == labels.data)

        epoch_loss = running_loss / len(dataloaders[phase].dataset)
        epoch_acc = running_corrects.double() / len(dataloaders[phase].dataset)

        print('{} Loss: {:.4f} Acc: {:.4f}'.format(phase, epoch_loss, epoch_acc)) #Se imprimen las estadisticas de la epoca

        # Se copia el mejor mejor modelo y se crean los arrays con los valores de cada epoca
        if phase == 'val' and epoch_acc > best_acc:
            best_acc = epoch_acc
            best_model_wts = copy.deepcopy(model.state_dict())
        if phase == 'val':
            val_acc_history.append(epoch_acc)
            val_loss_history.append(epoch_loss)
        if phase == 'train':
            train_acc_history.append(epoch_acc)
            train_loss_history.append(epoch_loss)

    print()

#Se imprimen los valores finales
time_elapsed = time.time() - since
print('Training complete in {:.0f}m {:.0f}s'.format(time_elapsed // 60, time_elapsed % 60))
print('Best val Acc: {:4f}'.format(best_acc))

# Se guardan los mejores valores y se devuelve la historia
model.load_state_dict(best_model_wts)
return model, val_acc_history,val_loss_history,train_acc_history,train_loss_history
```

```
#Se definen los parámetros a optimizar
def set_parameter_requires_grad(model, feature_extracting):
    if feature_extracting:
        for param in model.parameters():
            param.requires_grad = False
```

```python
#Se definen los diferentes modelos disponibles
def initialize_model(model_name, num_classes, feature_extract, use_pretrained=True):
    # Initialize these variables which will be set in this if statement. Each of these
    #   variables is model specific.
    model_ft = None
    input_size = 0

    if model_name == "resnet":
        """ Resnet18
        """
        model_ft = models.resnet18(pretrained=use_pretrained)
        set_parameter_requires_grad(model_ft, feature_extract)
        num_ftrs = model_ft.fc.in_features
        model_ft.fc = nn.Linear(num_ftrs, num_classes)
        input_size = 224

    elif model_name == "alexnet":
        """ Alexnet
        """
        model_ft = models.alexnet(pretrained=use_pretrained)
        set_parameter_requires_grad(model_ft, feature_extract)
        num_ftrs = model_ft.classifier[6].in_features
        model_ft.classifier[6] = nn.Linear(num_ftrs,num_classes)
        input_size = 224

    elif model_name == "vgg":
        """ VGG11_bn
        """
        model_ft = models.vgg11_bn(pretrained=use_pretrained)
        set_parameter_requires_grad(model_ft, feature_extract)
        num_ftrs = model_ft.classifier[6].in_features
        model_ft.classifier[6] = nn.Linear(num_ftrs,num_classes)
        input_size = 224

    elif model_name == "squeezenet":
        """ Squeezenet
        """
        model_ft = models.squeezenet1_0(pretrained=use_pretrained)
        set_parameter_requires_grad(model_ft, feature_extract)
        model_ft.classifier[1] = nn.Conv2d(512, num_classes, kernel_size=(1,1), stride=(1,1))
        model_ft.num_classes = num_classes
        input_size = 224

    elif model_name == "densenet":
        """ Densenet
        """
        model_ft = models.densenet121(pretrained=use_pretrained)
        set_parameter_requires_grad(model_ft, feature_extract)
        num_ftrs = model_ft.classifier.in_features
        model_ft.classifier = nn.Linear(num_ftrs, num_classes)
        input_size = 224
```

```
    elif model_name == "inception":
        """ Inception v3
        Be careful, expects (299,299) sized images and has auxiliary output
        """
        model_ft = models.inception_v3(pretrained=use_pretrained)
        set_parameter_requires_grad(model_ft, feature_extract)
        # Handle the auxilary net
        num_ftrs = model_ft.AuxLogits.fc.in_features
        model_ft.AuxLogits.fc = nn.Linear(num_ftrs, num_classes)
        # Handle the primary net
        num_ftrs = model_ft.fc.in_features
        model_ft.fc = nn.Linear(num_ftrs,num_classes)
        input_size = 299

    else:
        print("Invalid model name, exiting...")
        exit()

    return model_ft, input_size

# Initialize the model for this run
model_ft, input_size = initialize_model(model_name, num_classes, feature_extract, use_pretrained=True)

# Print the model we just instantiated
print(model_ft)
```

```
# Send the model to GPU
model_ft = model_ft.to(device)

# Gather the parameters to be optimized/updated in this run. If we are
#  finetuning we will be updating all parameters. However, if we are
#  doing feature extract method, we will only update the parameters
#  that we have just initialized, i.e. the parameters with requires_grad
#  is True.
params_to_update = model_ft.parameters()
print("Params to learn:")
if feature_extract:
    params_to_update = []
    for name,param in model_ft.named_parameters():
        if param.requires_grad == True:
            params_to_update.append(param)
            print("\t",name)
else:
    for name,param in model_ft.named_parameters():
        if param.requires_grad == True:
            print("\t",name)

# Observe that all parameters are being optimized
optimizer_ft = optim.SGD(params_to_update, lr=0.001, momentum=0.9)
```

A.1.5. Entrenar el modelo

```
# Se elige el criterio del calculo de perdida
criterion = nn.CrossEntropyLoss()

# Se entrena el modelo
model_ft, hist_val_acc, hist_val_loss, hist_train_acc, hist_train_loss = train_model(model_ft, dataloaders_dict, criterion, optimizer_ft, num_epochs=num_epochs, is_inception=(model_name=="inception"))
```

```
# Se entrena el modelo desde cero
scratch_model,_ = initialize_model(model_name, num_classes, feature_extract=False, use_pretrained=False)
scratch_model = scratch_model.to(device)
scratch_optimizer = optim.SGD(scratch_model.parameters(), lr=0.001, momentum=0.9)
scratch_criterion = nn.CrossEntropyLoss()
_, scratch_hist_val_acc, scratch_hist_val_loss, scratch_hist_train_acc, scratch_hist_train_loss = train_model(scratch_model, dataloaders_dict, scratch_criterion, scratch_optimizer, num_epochs=num_epochs,
```

A.1.6. Gráficas

Gráficas de precisión y pérdidas

```
# Se grafican los valores historicos de precisión de validacion en scratch y preentrenado
ohist_val_acc = []
shist_val_acc = []

ohist_val_acc = [h.cpu().numpy() for h in hist_val_acc]
shist_val_acc = [h.cpu().numpy() for h in scratch_hist_val_acc]

plt.title("Validation Accuracy vs. Number of Training Epochs")
plt.xlabel("Training Epochs")
plt.ylabel("Validation Accuracy")
plt.plot(range(1,num_epochs+1),ohist_val_acc,label="Pretrained")
plt.plot(range(1,num_epochs+1),shist_val_acc,label="Scratch")
plt.ylim((0,1.))
plt.xticks(np.arange(1, num_epochs+1, 1.0))
plt.legend()
plt.show()
```

Matriz de confusión

```
#Se definen las clases del dataset
classes = ('Under', 'Over', 'Empty', 'OK')
```

```
#En entas líneas de guardan los valores reales y supuestos del dataset y los resultados para las representaciones
y_pred_val = []
y_true_val = []
yscore_val = []

for inputs, labels in dataloaders_dict['val']: #validación o entrenamiento
        inputs, labels = inputs.to(device), labels.to(device)

        output = model_ft(inputs) # Feed Network
        output = (torch.max(torch.exp(output), 1)[1]).data.cpu().numpy()
        y_pred_val.extend(output) # Save Prediction

        labels = labels.data.cpu().numpy()
        y_true_val.extend(labels) # Save Truth

        model_ft.eval()
        logits = model_ft(inputs)
        yscore = F.softmax(logits, dim=1).data.cpu().numpy() # assuming logits has the shape [batch_size, nb_classes]
        yscore_val.extend(yscore)

y_pred_val = np.array(y_pred_val)
y_true_val = np.array(y_true_val)
yscore_val = np.array(yscore_val)
```

```
# Mostrar matriz de confusion
cf_matrix = confusion_matrix(y_true_val, y_pred_val) #Validacion o entrenamiento
disp = ConfusionMatrixDisplay(confusion_matrix = cf_matrix, display_labels = classes)
disp.plot()
plt.show()
```

Curva ROC

```
class_dict = {0: 'Under', 1: 'Over', 2: 'Empty', 3: 'Ok'}
y_true_valnames = [(class_dict[label])
          for label in y_true_val
          if label in [0, 1, 2, 3]]
```

```
#Elegir la clase que queremos ver frente al resto
classes = ('Under', 'Over', 'Empty', 'OK')
class_of_interest = 'OK'
class_id = classes.index(class_of_interest)
not_class_of_interest = [clase for clase in classes if clase != class_of_interest]

#Representamos la curva
RocCurveDisplay.from_predictions(
    y_onehot_test[:, class_id],
    yscore_val[:, class_id],
    name=f"{class_of_interest} vs the rest",
    color="darkorange",
)
plt.plot([0, 1], [0, 1], "k--", label="chance level (AUC = 0.5)")
plt.axis("square")
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title(f"One-vs-Rest ROC curves:\n {class_of_interest} vs {not_class_of_interest}")
plt.legend()
plt.show()
```

```
from itertools import cycle

fig, ax = plt.subplots(figsize=(6, 6))

colors = cycle(["aqua", "darkorange", "seagreen","indigo"])
for class_id, color in zip(range(len(classes)), colors):
    RocCurveDisplay.from_predictions(
        y_onehot_test[:, class_id],
        yscore_val[:, class_id],
        name=f"ROC curve for {classes[class_id]}",
        color=color,
        ax=ax,
    )

plt.plot([0, 1], [0, 1], "k--", label="ROC curve for chance level (AUC = 0.5)")
plt.axis("square")
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("Extension of Receiver Operating Characteristic\nto One-vs-Rest multiclass")
plt.legend()
plt.show()
```

```
from sklearn.metrics import roc_curve, auc

pair_scores = []
mean_tpr = dict()
fpr_grid = np.linspace(0.0, 1.0, 1000)

for ix, (label_a, label_b) in enumerate(pair_list):

    a_mask = y_true_val == label_a
    b_mask = y_true_val == label_b
    ab_mask = np.logical_or(a_mask, b_mask)

    a_true = a_mask[ab_mask]
    b_true = b_mask[ab_mask]

    idx_a = np.flatnonzero(label_binarizer.classes_ == label_a)[0]
    idx_b = np.flatnonzero(label_binarizer.classes_ == label_b)[0]

    fpr_a, tpr_a, _ = roc_curve(a_true, yscore_val[ab_mask, idx_a])
    fpr_b, tpr_b, _ = roc_curve(b_true, yscore_val[ab_mask, idx_b])

    mean_tpr[ix] = np.zeros_like(fpr_grid)
    mean_tpr[ix] += np.interp(fpr_grid, fpr_a, tpr_a)
    mean_tpr[ix] += np.interp(fpr_grid, fpr_b, tpr_b)
    mean_tpr[ix] /= 2
    mean_score = auc(fpr_grid, mean_tpr[ix])
    pair_scores.append(mean_score)

    fig, ax = plt.subplots(figsize=(6, 6))
    plt.plot(
        fpr_grid,
        mean_tpr[ix],
        label=f"Mean {class_dict[label_a]} vs {class_dict[label_b]} (AUC = {mean_score :.2f})",
        linestyle=":",
        linewidth=4,
    )
    RocCurveDisplay.from_predictions(
        a_true,
        yscore_val[ab_mask, idx_a],
        ax=ax,
        name=f"{class_dict[label_a]} as positive class",
    )
    RocCurveDisplay.from_predictions(
        b_true,
        yscore_val[ab_mask, idx_b],
        ax=ax,
        name=f"{class_dict[label_b]} as positive class",
    )
    plt.plot([0, 1], [0, 1], "k--", label="chance level (AUC = 0.5)")
    plt.axis("square")
    plt.xlabel("False Positive Rate")
    plt.ylabel("True Positive Rate")
    plt.title(f"{classes[idx_a]} vs {class_dict[label_b]} ROC curves")
    plt.legend()
    plt.show()

print(f"Macro-averaged One-vs-One ROC AUC score:\n{np.average(pair_scores):.2f}")
```

A.1.7. Aumentación de datos

```
class ImgAugTransform:
  def __init__(self):
    self.aug = iaa.Sequential([
        iaa.Scale((224, 224)),
        iaa.Sometimes(0.25, iaa.GaussianBlur(sigma=(0, 3.0))),
        iaa.Fliplr(0.5),
        iaa.Affine(rotate=(-20, 20), mode='symmetric'),
        iaa.Sometimes(0.25,
                      iaa.OneOf([iaa.Dropout(p=(0, 0.1)),
                                 iaa.CoarseDropout(0.1, size_percent=0.5)])),
        iaa.AddToHueAndSaturation(value=(-10, 10), per_channel=True)
    ])

  def __call__(self, img):
    img = np.array(img)
    return self.aug.augment_image(img)
```

```
# Definir transformaciones para las imágenes
transform = transforms.Compose([
    transforms.Resize((224,224)),
    transforms.RandomRotation(20),
    transforms.ToTensor()
])

#transform = transforms.Compose([
#    ImgAugTransform(),
#    transforms.ToTensor()
#])
```

A.1.8. Hiperparámetros

Elección de *learning rate* inicial

```
# Observe that all parameters are being optimized
optimizer_ft = optim.SGD(params_to_update, lr=0.05, momentum=0.9) #Se varia lr

# Setup the loss fxn
criterion = nn.CrossEntropyLoss()

# Train and evaluate
model_ft, hist_val_acc, hist_val_loss, hist_train_acc, hist_train_loss = train_model(model_ft, dataloaders_dict, criterion, optimizer_ft, num_epochs=num_epochs, is_inception=(model_name=="inception"))
```

```
# Observe that all parameters are being optimized
optimizer_ft = optim.SGD(params_to_update, lr=0.05, momentum=0.9) #Se varia lr

# Setup the loss fxn
criterion = nn.CrossEntropyLoss()

# Train and evaluate
model_ft, hist_val_acc, hist_val_loss, hist_train_acc, hist_train_loss = train_model(model_ft, dataloaders_dict, criterion, optimizer_ft, num_epochs=num_epochs, is_inception=(model_name=="inception"))
```

Variación de *learning rate*

```
# Observe that all parameters are being optimized
optimizer_ft = optim.SGD(params_to_update, lr=0.05, momentum=0.9)

# Setup the loss fxn
criterion = nn.CrossEntropyLoss()

scheduler = optim.lr_scheduler.StepLR(optimizer_ft, epochs = 2, gamma=0.1, last_epoch=- 1, verbose=False) #gamma controla cuanto se baja y epochs cada cuantas epocas

# Train and evaluate
model_ft, hist_val_acc, hist_val_loss, hist_train_acc, hist_train_loss = train_model(model_ft, dataloaders_dict, criterion, optimizer_ft, scheduler, num_epochs=num_epochs, is_inception=(model_name=="inception"))
```

```
# Observe that all parameters are being optimized
optimizer_ft = optim.SGD(params_to_update, 0.005, momentum=0.9)

# Setup the loss fxn
criterion = nn.CrossEntropyLoss()

scheduler = optim.lr_scheduler.ExponentialLR(optimizer_ft, 0.1, last_epoch=- 1, verbose=False)

# Train and evaluate
model_ft, hist_val_acc, hist_val_loss, hist_train_acc, hist_train_loss = train_model(model_ft, dataloaders_dict, criterion, optimizer_ft, scheduler, num_epochs=num_epochs, is_inception=(model_name=="inception"))
```

```
            # statistics
            running_loss += loss.item() * inputs.size(0)
            running_corrects += torch.sum(preds == labels.data)

        if phase == 'train': #Se introduce para la variación de hiperparametros
          scheduler.step()
          print(optimizer_ft.state_dict()['param_groups'][0]['lr'],epoch,sep = "->epoch ")

        epoch_loss = running_loss / len(dataloaders[phase].dataset)
        epoch_acc = running_corrects.double() / len(dataloaders[phase].dataset)

        print('{} Loss: {:.4f} Acc: {:.4f}'.format(phase, epoch_loss, epoch_acc))

        # deep copy the model
        if phase == 'val' and epoch_acc > best_acc:
            best_acc = epoch_acc
            best_model_wts = copy.deepcopy(model.state_dict())
        if phase == 'val':
            val_acc_history.append(epoch_acc)
            val_loss_history.append(epoch_loss)
        if phase == 'train':
            train_acc_history.append(epoch_acc)
            train_loss_history.append(epoch_loss)

    print()

time_elapsed = time.time() - since
print('Training complete in {:.0f}m {:.0f}s'.format(time_elapsed // 60, time_elapsed % 60))
print('Best val Acc: {:4f}'.format(best_acc))

# load best model weights
model.load_state_dict(best_model_wts)
return model, val_acc_history,val_loss_history,train_acc_history,train_loss_history
```

A.1.9. Validación cruzada

Aleatoria

```
#Para la validación cruzada se ha creado esta función para reiniciar los pesos calculados del modelo
def reset_weights(m):
  '''
    Try resetting model weights to avoid
    weight leakage.
  '''
  for layer in m.children():
   if hasattr(layer, 'reset_parameters'):
    print(f'Reset trainable parameters of layer = {layer}')
    layer.reset_parameters()
```

```
from sklearn.model_selection import ShuffleSplit #Se importa ShufleSplit de la libreria de modelos de sklearn

n_splits = 5 # Número de divisiones
results = {} #Se guardan los resultados de cada validación cruzada
loss_function = nn.CrossEntropyLoss()


def train_model(model, dataset, num_epochs=25, is_inception=False):
    since = time.time()

    best_model_wts = copy.deepcopy(model.state_dict())
    best_acc = 0.0
    rs = ShuffleSplit(n_splits=n_splits, test_size=0.2, random_state=42) #Se crea el objeto para la division, con 5 validaciones del 20%


  #La principal diferencia al hacer la validación cruzada es el siguiente bucle for que se encarga de realizar tantas validaciones cruzadas como se indique.
  #El entrenamiento y validación son practicamente identicos
    for fold, (train_ids, test_ids) in enumerate(rs.split(dataset)):
      print(f"Split {fold + 1}")

      train_subsampler = torch.utils.data.SubsetRandomSampler(train_ids)
      test_subsampler = torch.utils.data.SubsetRandomSampler(test_ids)

      trainloader = torch.utils.data.DataLoader(
          dataset,
          batch_size=10,
          sampler=train_subsampler
      )
      testloader = torch.utils.data.DataLoader(
          dataset,
          batch_size=10,
          sampler=test_subsampler
      )

      network = model
      network.apply(reset_weights) #Se reinician los pesos calculados en cada validación cruzada

      optimizer = torch.optim.Adam(network.parameters(), lr=1e-4) # Se reinicia la función de optimización

      for epoch in range(num_epochs):
        print(f'Starting epoch {epoch+1}') #Print epoch
```

Series temporales

```
from sklearn.model_selection import TimeSeriesSplit #Se importa TimeSeriesSplit estratificado de la libreria de modelos de sklearn

n_splits = 5 # Número de divisiones
results = {}
loss_function = nn.CrossEntropyLoss()

def train_model(model, dataset, num_epochs=25, is_inception=False):
    since = time.time()



    best_model_wts = copy.deepcopy(model.state_dict())
    best_acc = 0.0
    TSP = TimeSeriesSplit(n_splits=n_splits) #En este caso tan solo se eligen las divisiones, dado que el tamaño del conjunto de entrenamiento
                                             # y validación son variables

    for fold, (train_ids, test_ids) in enumerate(TSP.split(dataset)):
      print(f"Split {fold + 1}")

      train_subsampler = torch.utils.data.SubsetRandomSampler(train_ids)
      test_subsampler = torch.utils.data.SubsetRandomSampler(test_ids)

      trainloader = torch.utils.data.DataLoader(
          dataset,
          batch_size=10,
          sampler=train_subsampler
      )
      testloader = torch.utils.data.DataLoader(
          dataset,
          batch_size=10,
          sampler=test_subsampler
      )

      network = model
      network.apply(reset_weights)

      optimizer = torch.optim.Adam(network.parameters(), lr=1e-4)

      for epoch in range(num_epochs):
        print(f'Starting epoch {epoch+1}') #Print epoch
```

K-folds

```
from sklearn.model_selection import KFold #Se importa KFold de la libreria de modelos de sklearn
k_folds = 5
results = {}

def train_model(model, dataset, num_epochs=25, is_inception=False):
    since = time.time()

    best_model_wts = copy.deepcopy(model.state_dict())
    best_acc = 0.0
    kfold = KFold(n_splits=k_folds, shuffle=True) #Se crea el objeto para la division, con 5 validaciones

    for fold, (train_ids, test_ids) in enumerate(kfold.split(dataset)):

      # Lineas para la Impresion
      print(f'FOLD {fold}')
      print('--------------------------------')

      train_subsampler = torch.utils.data.SubsetRandomSampler(train_ids)
      test_subsampler = torch.utils.data.SubsetRandomSampler(test_ids)

      trainloader = torch.utils.data.DataLoader(
                        dataset,
                        batch_size=10, sampler=train_subsampler)
      testloader = torch.utils.data.DataLoader(
                        dataset,
                        batch_size=10, sampler=test_subsampler)

      network = model
      network.apply(reset_weights)

      optimizer = torch.optim.Adam(network.parameters(), lr=1e-4)

      for epoch in range(num_epochs):
        print(f'Starting epoch {epoch+1}') #Print epoch

      # Set current loss value
        current_loss = 0.0

      # Iterate over the DataLoader for training data
        for i, data in enumerate(trainloader, 0):

          # Get inputs
          inputs, targets = data
```

A.1.10. Guardar y mostrar imágenes mal clasificadas

```
misclassified_images = [] #Se crea una variable para guardar las imagenes mal clasificadas

# Iterar sobre las imágenes del conjunto de validación
for images, labels in dataloaders_dict['val']:
    images = images.to(device)
    labels = labels.to(device)
    outputs = model_ft(images)
    _, predicted = torch.max(outputs, 1)
    incorrect_predictions = predicted != labels #Se crea una variable que indique si hay imagenes mal clasificadas
    misclassified_images.extend(zip(images[incorrect_predictions].cpu(), labels[incorrect_predictions].cpu(), predicted[incorrect_predictions].cpu())) #Se guardan

# Convertir las imágenes, etiquetas verdaderas y predicciones incorrectas en tensores
misclassified_images_tensors = [torch.stack(t) for t in zip(*misclassified_images)]

# Crear el TensorDataset a partir de los tensores
misclassified_images_dataset = torch.utils.data.TensorDataset(*misclassified_images_tensors)
```

```
# Crear un diccionario para almacenar las imágenes mal clasificadas de cada tipo
misclassified_dict = defaultdict(list)

# Recorrer todas las imágenes mal clasificadas y guardarlas en el diccionario
for image, true_label, predicted_label in misclassified_images_dataset:
    misclassified_dict[true_label.item()].append((image, true_label.item(), predicted_label.item()))

# Mostrar las imágenes mal clasificadas con sus etiquetas real y supuesta (No es necesario, comprobacion)
for true_label, misclassified_images in misclassified_dict.items():
    print(f"Imágenes mal clasificadas de tipo {true_label}:")
    for image, true_label, predicted_label in misclassified_images:
        print(f"Etiqueta real: {true_label}, Etiqueta supuesta: {predicted_label}")
```

```
# Recorrer las imágenes mal clasificadas y mostrarlas junto con su etiqueta real y supuesta
for true_label, misclassified_images in misclassified_dict.items():
    print(f"Imágenes mal clasificadas de tipo {true_label}:")

    # Crear una nueva figura para cada tipo de imagen
    fig = plt.figure(figsize=(12, 8))
    rows = 4
    cols = 8

    for i, (image, true_label, predicted_label) in enumerate(misclassified_images):
        ax = fig.add_subplot(rows, cols, i + 1)
        ax.axis('off')
        ax.set_title(f"Real: {true_label}\nSupuesto: {predicted_label}")
        ax.imshow(image.permute(1, 2, 0))

        # Salir del bucle si se alcanza el número máximo de subplots por figura
        if i == (rows * cols - 1):
            break

    # Ajustar los espacios entre subplots
    plt.tight_layout()

    # Mostrar la figura actual
    plt.show()
```

BIBLIOGRAFÍA

[1] International Organization for Standardization (ISO). *ISO/ASTM 52900:2021(en) Additive manufacturing — General principles - Fundamentals and vocabulary*. https://www.iso.org/standard/69669.html. 2021.

[2] Silvia Leal. *E-Renovarse o morir: 7 Tendencias tecnológicas para convertirte en un líder digital*. LID Editorial Empresarial, S.L., 2015.

[3] A. Jandyal et al. «3D printing – A review of processes, materials and applications in industry 4.0». En: *Sustainable Operations and Computers* 3 (2022). https://www.sciencedirect.com/science/article/pii/S2666412721000441, págs. 33-42.

[4] W.E. Masters. *Computer Automated Manufacturing Process and System*. U.S. Patent 4,665,492, 12 May 1987. https://patents.google.com/patent/US4665492/en.

[5] V. Sreehitha. «Impact of 3D printing in automotive industry.» En: *International Journal of Mechanical And Production Engineering* 5 (2017). https://www.iraj.in/journal/journal_file/journal_pdf/2-347-149260399391-94.pdf, págs. 91-94.

[6] S. Singamneni et al. «Additive Manufacturing for the Aircraft Industry: A Review.» En: *Journal of Aeronautics 'I&' Aerospace Engineering* 8 (2019). https://www.longdom.org/open-access/additive-manufacturing-for-the-aircraft-industry-a-review-18967.html, págs. 1-13.

[7] M.; Haleem Javaid. «Additive manufacturing applications in medical cases: A literature based review.» En: *Alexandria Journal of Medicine* 54 (2018). https://doi.org/10.1007/s00170-018-1601-1, págs. 411-422.

[8] N. Noor et al. «3D Printing of Personalized Thick and Perfusable Cardiac Patches and Hearts.» En: *Advanced Science* 6 (2019). https://onlinelibrary.wiley.com/doi/10.1002/advs.201900344, pág. 1900344.

[9] Q. Zhang et al. «3D Printing of Graphene Aerogels.» En: *Small* 12 (2016). https://onlinelibrary.wiley.com/doi/abs/10.1002/smll.201503524, págs. 1702-1708.

[10] John Loeffler. *Wilson's newest basketball prototype is 3D-printed and doesn't need air*. Interesting Engineering. https://interestingengineering.com/innovation/wilson-airless-basketball-prototype. 2023.

[11] P. Karayannis et al. «Facilitating Safe FFF 3D Printing: A Prototype Material Case Study». En: *Sustainability* 14.5 (2022). https://www.mdpi.com/2071-1050/14/5/3046. ISSN: 2071-1050.

[12] M. Simons. «Additive manufacturing—a revolution in progress? Insights from a multiple case study.» En: *Int J Adv Manuf Technol* 96 (2018). https://doi.org/10.1007/s00170-018-1601-1, págs. 735-749.

[13] ALL3DP. *The 7 Main Types of 3D Printing Technology*. https://all3dp.com/1/types-of-3d-printers-3d-printing-technology/. 2023.

[14] Digital manufacturing news y research. *FFF vs FDM: Difference and Best Printers*. https://top3dshop.com/blog/fff-vs-fdm-difference-and-best-printers. 2020.

[15] Xometry. *FDM vs. FFF: Differences and Comparison*. https://www.xometry.com/resources/3d-printing/fdm-vs-fff-3d-printing/#:~:text=The%20main%20difference%20between%20FFF,and%20more%20likely%20to%20warp.. 2022.

[16] Javid Akhavan y Jiaqi Lyu. «Image-based dataset of artifact surfaces fabricated by additive manufacturing with applications in machine learning». En: *Data in Brief* 41 (2022). https://www.sciencedirect.com/science/article/pii/S2352340922000646, pág. 107852.

[17] Ans Al Rashid y Muammer Koç. «Fused Filament Fabrication Process: A Review of Numerical Simulation Techniques». En: *Polymers* 13.20 (2021). https://www.mdpi.com/2073-4360/13/20/3534, pág. 3534.

[18] John Ryan C. Dizon et al. «Post-Processing of 3D-Printed Polymers». En: *Technologies* 9.3 (2021). https://doi.org/10.3390/technologies9030061, pág. 61.

[19] Webedia Brand Services. *Cómo se levanta uno e inventa la IA en 1956*. Xataka. https://ecosistemahuawei.xataka.com/como-se-levanta-uno-e-inventa-la-ia-en-1956/. 2018.

[20] A.M. Turing. «COMPUTING MACHINERY AND INTELLIGENCE». En: *Mind* 49 (1950). https://academic.oup.com/mind/article/LIX/236/433/986238, págs. 433-460.

[21] Chess-poster blog. *Claude E. Shannon*. Chess-poster. https://www.chess-poster.com/spanish/historia/claude_e_shannon_e.htm.

[22] Melanie Lefkowitz. *Professor's perceptron paved the way for AI – 60 years too soon*. Cornell Chronicle. https://news.cornell.edu/stories/2019/09/professors-perceptron-paved-way-ai-60-years-too-soon. 2019.

[23] Frank Rosenblatt. «The Design of an Intelligent Automaton». En: *Research Trends* 2.6 (1958). https://www.researchgate.net/publication/357680315_The_Gestalt_of_AI_Beyond_the_Holism-Atomism_Divide, págs. 1-7.

[24] Marvin Minsky y Seymour Papert. *Perceptrons: An Introduction to Computational Geometry*. https://direct.mit.edu/books/book/3132/PerceptronsAn-Introduction-to-Computational. Cambridge, MA, USA: MIT Press, 1969.

[25] David E Rumelhart, Geoffrey E Hinton y Ronald J Williams. «Learning representations by back-propagating errors». En: *nature* 323.6088 (1986). https://www.nature.com/articles/323533a0, págs. 533-536.

[26] Mark Sullivan. *The 10 most important moments in AI (so far)*. FastCompany. https://www.fastcompany.com/90402503/the-10-most-important-moments-in-ai-so-far. 2019.

[27] Garry Kasparov. *Man vs Machine*. Garry Kasparov page. https://www.kasparov.com/timeline-event/deep-blue/.

[28] Quoc V. Le et al. *Building high-level features using large scale unsupervised learning*. https://arxiv.org/abs/1112.6209. 2011.

[29] Koray Kavukcuoglu et al. «Mastering the Game of Go with Deep Neural Networks and Tree Search». En: *Nature* 529.7587 (2016). https://www.nature.com/articles/nature16961, págs. 484-489.

[30] Koray Kavukcuoglu et al. «Highly accurate protein structure prediction with AlphaFold». En: *Nature* 596.7873 (2021). https://www.nature.com/articles/s41586-021-03819-2, págs. 583-589.

[31] Alex Hern. *AlphaGo: its creator on the computer that learns by thinking*. The Guardian. https://www.theguardian.com/technology/2016/mar/15/alphago-what-does-google-advanced-software-go-next. 2016.

[32] OpenAI. *Introducing ChatGPT*. https://openai.com/blog/chatgpt. 2022.

[33] Coursera. *Deep Learning vs. Machine Learning: Beginner's Guide*. Coursera. https://www.coursera.org/articles/ai-vs-deep-learning-vs-machine-learning-beginners-guide.

[34] Carlos Santana Vega. *¿Qué es el Machine Learning?¿Y Deep Learning? Un mapa conceptual — DotCSV*. YouTube: canal Dot CSV. https://www.youtube.com/watch?v=KytW151dpqU&list=PL-Ogd76BhmcC_E2RjgIIJZd1DQdYHcVf0&index=1&pp=iAQB.

[35] IBM. *¿Qué es Deep Learning?* IBM.com. https://www.ibm.com/es-es/topics/deep-learning.

[36] Ezequiel Lopez Rubio. *Fundamentals of deep learning*. https://drive.google.com/drive/folders/1hp7NXQrCVkGgwjhYRFGtMuEZxvsj0A2E. 2020.

[37] Ritwick Roy. *Neural Networks: Forward pass and Backpropagation*. Towards Data Science. https://towardsdatascience.com/neural-networks-forward-pass-and-backpropagation-be3b75a1cfcc. 2022.

[38] Carlos Santana Vega. *¿Qué es una Red Neuronal? Parte 2 : La Red — DotCSV*. YouTube: canal Dot CSV. https://www.youtube.com/watch?v=uwbHOpp9xkc&list=PL-Ogd76BhmcC_E2RjgIIJZd1DQdYHcVf0&index=7&pp=iAQB.

[39] Carlos Santana Vega. *¿Qué es una Red Neuronal? Parte 3 : Backpropagation — DotCSV*. YouTube: canal Dot CSV. https://www.youtube.com/watch?v=eNIqz_noix8&list=PL-Ogd76BhmcC_E2RjgIIJZd1DQdYHcVf0&index=9.

[40] Ezequiel Lopez Rubio. *Convolutional Neural Networks*. https://drive.google.com/drive/folders/1hp7NXQrCVkGgwjhYRFGtMuEZxvsj0A2E. 2020.

[41] Carlos Santana Vega. *¡Redes Neuronales CONVOLUCIONALES! ¿Cómo funcionan?* YouTube: canal Dot CSV. https://www.youtube.com/watch?v=V8j1oENVz00&t=403s. 2020.

[42] Sumit Saha. *A Comprehensive Guide to Convolutional Neural Networks — the ELI5 way*. SaturnCloud. https://saturncloud.io/blog/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way/. 2018.

[43] Andrea Perlato. *CNN and Softmax*. andreaperlato.com. https://www.andreaperlato.com/aipost/cnn-and-softmax/. 2019.

[44] Rizel Scarlett. *Why Python keeps growing, explained*. GitHub. https://isip.piconepress.com/courses/temple/ece_4822/resources/books/Deep-Learning-with-PyTorch.pdf. 2022.

[45] Lex Fridman. «Guido van Rossum: Python — Lex Fridman Podcast Nº6». En: https://www.youtube.com/watch?v=ghwaIiE3Nd8. 2018.

[46] Eli Stevens, Luca Antiga y Thomas Viehmann. *Deep Learning with PyTorch*. 2020. URL: https://pytorch.org/assets/deep-learning/Deep-Learning-with-PyTorch.pdf.

[47] Tokio School. *Libreria scikit-learn python, aprende todo*. https://www.tokioschool.com/noticias/que-es-scikit-learn/. 2023.

[48] Google. *Colaboratory: Preguntas frecuentes*. https://research.google.com/colaboratory/intl/es/faq.html.

[49] MathWorks. *Transfer learning para entrenar modelos de Deep Learning*. MathWorks. https://es.mathworks.com/discovery/transfer-learning.html.

[50] Ezequiel Lopez Rubio. *Feature extraction*. https://drive.google.com/drive/folders/1hp7NXQrCVkGgwjhYRFGtMuEZxvsj0A2E. 2020.

[51] Nathan Inkawhich. *FINETUNING TORCHVISION MODELS*. PyTorch. https://pytorch.org/tutorials/beginner/finetuning_torchvision_models_tutorial.html. 2017.

[52] Alex Krizhevsky, Ilya Sutskever y Geoffrey E. Hinton. «ImageNet Classification with Deep Convolutional Neural Networks». En: *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*. https://proceedings.neurips.cc/paper_files/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf. Red Hook, NY, USA: Curran Associates Inc., 2012, págs. 1097-1105.

[53] Karen Simonyan y Andrew Zisserman. «Very deep convolutional networks for large-scale image recognition». En: *arXiv preprint arXiv:1409.1556* (2014). https://arxiv.org/abs/1409.1556.

[54] Gao Huang et al. «Deep Residual Learning for Image Recognition». En: *Proceedings of 2016 IEEE Conference on Computer Vision and Pattern Recognition* (2016). https://arxiv.org/abs/1512.03385.

[55] Mingyu Gao et al. «A Novel Deep Convolutional Neural Network Based on ResNet-18 and Transfer Learning for Detection of Wood Knot Defects». En: *Journal of Sensors* (2022). https://doi.org/10.1155/2021/4428964.

[56] Christian Szegedy et al. «Rethinking the Inception Architecture for Computer Vision». En: *CoRR* (2015). https://arxiv.org/abs/1512.00567.

[57] Aravinda S Rao, Tuan Nguyen y Tuan Ngo. «Vision-based automated crack detection using convolutional neural networks for condition assessment of infrastructure». En: *Structural Health Monitoring* 20 (4 2020). https://journals.sagepub.com/doi/abs/10.1177/1475921720965445, págs. 2124-2142.

[58] Gao Huang et al. «Densely Connected Convolutional Networks». En: *arXiv preprint arXiv:1608.06993* (2015). https://arxiv.org/abs/1608.06993.

[59] F. Iandola et al. «SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and ¡0.5MB model size». En: *arxiv:1602.07360Comment* (2016). https://arxiv.org/abs/1602.07360.

[60] Archna Oberoi. *What is Data Augmentation in Deep Learning?* Daffodil. https://insights.daffodilsw.com/blog/what-is-data-augmentation-in-deep-learning. 2022.

[61] Alexander Jung. *imgaug*. Github. https://github.com/aleju/imgaug. 2022.

[62] Kizito Nyuytiymbiy. *Parameters and Hyperparameters in Machine Learning and Deep Learning*. Towards Data Science. https://towardsdatascience.com/parameters-and-hyperparameters-aa609601a9ac. 2020.

[63] John Duchi, Elad Hazan y Yoram Singer. «Adaptive subgradient methods for online learning and stochastic optimization». En: *Journal of Machine Learning Research* 12.Jul (2011). https://www.jmlr.org/papers/volume12/duchi11a/duchi11a.pdf, págs. 2121-2159.

[64] InteractiveChaos. *AdaGrad*. InteractiveChaos.com. https://interactivechaos.com/es/manual/tutorial-de-machine-learning/adagrad.

[65] Matthew D Zeiler. «ADADELTA: An Adaptive Learning Rate Method». En: *CoRR* abs/1212.5701 (2012). http://dblp.uni-trier.de/db/journals/corr/corr1212.html#abs-1212-5701.

[66] Diederik P Kingma y Jimmy Ba. «Adam: A method for stochastic optimization». En: *arXiv preprint arXiv:1412.6980* (2014). https://arxiv.org/abs/1412.6980.

[67] Hasty. *Adam*. Hasty. https://hasty.ai/docs/mp-wiki/solvers-optimizers/adam.

[68] B. D. Hammel. *What learning rate should I use?* B. D. Hammel.com. http://www.bdhammel.com/learning-rates/. 2019.

[69] Hasty.ai. *StepLR*. Hasty.ai. https://hasty.ai/docs/mp-wiki/scheduler/steplr. 2023.

[70] Hasty.ai. *ExponentialLR*. Hasty.ai. https://hasty.ai/docs/mp-wiki/scheduler/exponentiallr. 2023.

[71] Lydia Bouzar-Benlabiod, Stuart H. Rubin y Amel Benaida. «Optimizing Deep Neural Network Architectures: an overview». En: *IEEE 22nd International Conference on Information Reuse and Integration for Data Science (IRI)* (2021). https://arxiv.org/abs/1808.05979.

[72] Shashank Ramesh. *A guide to an efficient way to build neural network architectures- Part I: Hyper-parameter selection and tuning for Dense Networks using Hyperas on Fashion-MNIST*. Medium. https://towardsdatascience.com/a-guide-to-an-efficient-way-to-build-neural-network-architectures-part-i-hyper-parameter-8129009f131b. 2018.

[73] Redacción KeepCoding. *Grid search en Python*. KeepCoding. https://keepcoding.io/blog/grid-search-en-python/. 2023.

[74] Shashank Shekhar, Adesh Bansode y Asif Salim. «A Comparative study of Hyper-Parameter Optimization Tools». En: *IEEE 22nd International Conference on Information Reuse and Integration for Data Science (IRI)* (2022). https://arxiv.org/abs/2201.06433.

[75] James Bergstra y Yoshua Bengio. «Random Search for Hyper-Parameter Optimization». En: *Journal of Machine Learning Research* 13 (2012), págs. 281-305. URL: http://www.jmlr.org/papers/v13/bergstra12a.html.

[76] Jasper Snoek, Hugo Larochelle y Ryan P. Adams. «Practical Bayesian Optimization of Machine Learning Algorithms.» En: (2012). URL: https://arxiv.org/abs/1206.2944.

[77] Will Koehrsen. *A Conceptual Explanation of Bayesian Hyperparameter Optimization for Machine Learning*. Medium. https://towardsdatascience.com/a-conceptual-explanation-of-bayesian-model-based-hyperparameter-optimization-for-machine-learning-b8172278050f. 2018.

[78] Yoshua Bengio. «Practical Recommendations for Gradient-Based Training of Deep Architectures». En: (2012). URL: https://arxiv.org/abs/1206.5533.

[79] Lisha Li et al. «Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization». En: (2018). URL: https://arxiv.org/abs/1603.06560.

[80] Aashish Nair. *Grid Search VS Random Search VS Bayesian Optimization*. Towards Data Science. https://towardsdatascience.com/grid-search-vs-random-search-vs-bayesian-optimization-2e68f57c3c46. 2022.

[81] Jakub Czakon. *Optuna vs Hyperopt: Which Hyperparameter Optimization Library Should You Choose?* neptune.ia. https://neptune.ai/blog/optuna-vs-hyperopt. 2023.

[82] Dário Passos y Puneet Mishra. «A tutorial on automatic hyperparameter tuning of deep spectral modelling for regression and classification tasks». En: *Chemometrics and Intelligent Laboratory Systems* 223 (2022), pág. 104520. URL: https://www.sciencedirect.com/science/article/pii/S0169743922000314.

[83] Pragati Baheti. *Train Test Validation Split: How To Best Practices*. V7 Labs. https://www.v7labs.com/blog/train-validation-test-set#:~:text=In%20general%2C%20putting%2080%25%20of,good%20split%20to%20start%20with.. 2023.

[84] Eijaz Allibhai. *Hold-out vs. Cross-validation in Machine Learning*. Medium. https://medium.com/@eijaz/holdout-vs-cross-validation-in-machine-learning-7637112d3f8f. 2018.

[85] Satyam Kumar. *Understanding 8 types of Cross-Validation*. https://towardsdatascience.com/understanding-8-types-of-cross-validation-80c935a4976d. 2020.

[86] Scikit Learn. *3.1. Cross-validation: evaluating estimator performance*. https://scikit-learn.org/stable/modules/cross_validation.html.

[87] Rubiales Alberto. *¿Qué es Underfitting y Overfitting?* Medium. https://rubialesalberto.medium.com/qu%C3%A9-es-underfitting-y-overfitting-c73d51ffd3f9.

[88] Keep Coding. *¿Qué es el overfitting?* Keep coding: Tech school.com. https://keepcoding.io/blog/que-es-el-overfitting/#:~:text=El%20overfitting%20o%20sobreajuste%20es,cualquier%20otro%20conjunto%20de%20datos..

[89] Lory Seraydarian. *What Is a Confusion Matrix in Machine Learning?* Plat.AI. https://plat.ai/blog/confusion-matrix-in-machine-learning/. 2022.

[90] Skicit learn. *Confusion matrix*. Skicit learn. https://scikit-learn.org/stable/auto_examples/model_selection/plot_confusion_matrix.html.

[91] Doug Steen. *Understanding the ROC Curve and AUC*. Towards Data Science. https://towardsdatascience.com/understanding-the-roc-curve-and-auc-dd4f9a192ecb. 2020.

[92] Skicit learn. *Multiclass Receiver Operating Characteristic (ROC)*. Skicit learn. https://scikit-learn.org/stable/auto_examples/model_selection/plot_roc.html#one-vs-rest-multiclass-roc.

[93] Arslan Tariq. *What is the difference between micro and macro averaging?)* Educative. https://www.educative.io/answers/what-is-the-difference-between-micro-and-macro-averaging.

[94] Stephen Allwright. *What is a good accuracy score in machine learning?* StephenAllwright.com. https://stephenallwright.com/good-accuracy-score/#:~:text=There%20is%20a%20general%20rule,60%25%20and%2070%25%20%2D%20OK. 2022.

[95] Anmol Malhotra et al. «SqueezeNet: The Key to Unlocking the Potential of Edge Computing». En: *SFU Professional Computer Science* (2023). https://medium.com/sfu-cspmp/squeezenet-the-key-to-unlocking-the-potential-of-edge-computing-c8b224d839ba.

[96] Jason Brownlee. *How to Configure the Learning Rate When Training Deep Learning Neural Networks*. Machine Learning Mastery. https://machinelearningmastery.com/learning-rate-for-deep-learning-neural-networks/#:~:text=A%20traditional%20default%20value%20for,starting%20point%20on%20your%20problem.. 2019.

[97] Christian Versloot. *How to use K-fold Cross Validation with PyTorch?* https://github.com/christianversloot/machine-learning-articles/blob/main/how-to-use-k-fold-cross-validation-with-pytorch.md.