Universidad de Valladolid

Escuela de Ingeniería Informática

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática
(Mención en Ingeniería de Software)

REDES CONVOLUCIONALES 2D EN PYTORCH:
Clasificación de imágenes de TAC de retina (OCT)

Alumno: Mario Izquierdo Álvarez

Tutor: Teodoro Calonge Cano

Se lo dedico a mis padres, por confiar en mí y apoyarme siempre.

Agradecimientos

Le agradezco a mi familia, pareja y amigos todo el apoyo que me han brindado durante estos años. Igualmente, le agradezco a todos los profesores que han contribuido en mi formación académica por permitirme crecer como persona y como estudiante.

Resumen

En la actualidad, los progresos en el campo de la Inteligencia Artificial han destacado enormemente, abriendo la discusión sobre su impacto en diferentes sectores. Particularmente, el sanitario puede beneficiarse en gran medida de estas tecnologías, que tienen el potencial de facilitar la detección precoz de enfermedades, agilizar las tareas médicas y mejorar la precisión de los diagnósticos.

El presente Trabajo de Fin de Grado, explora el área de la Visión por Computador, proponiendo un sistema de diagnóstico de enfermedades retinianas a partir de imágenes de secciones transversales de Tomografías de Coherencia Óptica (OCT). Dentro de la Medicina Oftalmológica, las OCT permiten a los profesionales de la salud una visualización de alta resolución de las capas y estructura internas de la retina, lo cual facilita el diagnóstico de anomalías presentes. Sin embargo, esta tarea resulta especialmente laboriosa, y puede estar sujeta a errores humanos.

En este estudio, se presenta el marco teórico y práctico para el desarrollo de un clasificador de imágenes OCT, utilizando técnicas de Aprendizaje Profundo, específicamente, Redes Neuronales Convolucionales. El modelo propuesto alcanza una precisión superior al 96 % en la clasificación de imágenes OCT no presentadas previamente al sistema; no obstante, es importante destacar que los diagnósticos ofrecidos no han de considerarse definitivos, pues para obtener la máxima fiabilidad de estos, es necesaria la intervención humana. Para ello, el trabajo realizado se ha complementado con la implementación de Grad-CAM, un algoritmo que produce una visualización justificatoria del diagnóstico ofrecido por el sistema, lo cual resulta de gran utilidad para la interpretación de los resultados.

Como culmen de este proyecto, se desarrolla una aplicación web, que permite utilizar de forma sencilla e intuitiva el sistema de diagnóstico.

Abstract

Nowadays, Artificial Intelligence (AI) has experienced significant development increase, leading to the discussion of its potential across worldwide technologies. One of the most emerging fields AI has been applied to is Biomedicine, in order to facilitate early disease detection, obtain more efficient treatments and enhance diagnostic accuracy.

A Computer Vision system for retinal disease recognition using Optical Coherence Tomography (OCT) cross-sectional images is proposed. More precisely, a Convolutional Neural Network image classifier has been implemented. The model has been trained with a dataset composed of more than $85,000$ OCT examples distributed across four categories: Choroidal Neovascularization (CNV), Diabetic Macular Edema (DME), DRUSEN and NORMAL.

A Hold-Out methodology with validation set has been used during the training and evaluation of the proposed system. The final results show an impressive performance, with an accuracy surpassing $96\,\%$ in categorizing never presented OCT images. In addition, an explainability study has been performed with the incorporation of Grad-CAM. This algorithm generates attention maps which highlight the most relevant areas to achieve the classification task.

The culmination of this project is the creation of a user-friendly web application, specifically designed to allow the use of the diagnostic system in a simple and intuitive way.

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Introducción

1.1. Contexto social

En pleno 2023, los notables avances realizados en el campo de la Inteligencia Artificial, unidos a la gran popularidad mediática obtenida, han provocado que sea percibida por muchos, como la mayor revolución tecnológica desde la creación de internet. En esta época de cambio, abundan la expectación, y el temor, como dos sentimientos contrapuestos a una tecnología, que parece que va a cambiar nuestras vidas para siempre [1]. Como toda nueva tecnología, la Inteligencia Artificial, viene acompañada de una gran cantidad de incertidumbre. Sin embargo, puede ser muy beneficiosa y útil en ciertas áreas, una de las más destacadas es la Medicina, por su potencial en la ayuda al diagnóstico de numerosas enfermedades [2].

1.2. Origen del proyecto

Este Trabajo de Fin de Grado nace, en sus orígenes, de un reto Kaggle [3]. Consistía en la elaboración de un clasificador de imágenes bidimensionales de secciones transversales de Tomografías de Coherencia Óptica. Para este fin, se dispone un conjunto de imágenes etiquetadas según la anomalía presente. Sin embargo, una serie de motivos que se expondrán más adelante hacen que este TFG se aleje de su propuesta inicial, indagando más profundamente en otros aspectos de gran interés para un TFG de Ingeniería Informática.

1.3. Motivación

Dentro de la Oftalmología, una de las áreas que ha experimentado un notable progreso es la detección y diagnosis de enfermedades oculares mediante el uso de imágenes procesadas. En particular, la Tomografía de Coherencia Óptica (OCT, por sus siglas en inglés) es un claro ejemplo. Son imágenes de alta resolución de la estructura interna del ojo, permitiendo a los médicos visualizar con detalle las capas retinianas y detectar anomalías, que podrían ser indicativas de enfermedades oculares [4]. Sin embargo, el análisis manual de las imágenes de OCT puede ser laborioso, subjetivo y propenso a errores humanos. Es aquí donde una herramienta basada en Inteligencia Artificial, podría servir de ayuda para el análisis de las OCT, pudiendo impulsar tanto la velocidad, como la calidad de los diagnósticos.

El proyecto se centrará en el estudio y aplicación de las Redes Neuronales Convolucionales (CNN, por sus siglas en inglés) [5], para la ayuda al diagnóstico del Edema Macular Diabético [6], la Neovascularización Coroidea [7], y el Drusen [8].

Cuando se trata la Inteligencia Artificial, suelen aparecer en nuestros días con mucha frecuencia, el Aprendizaje Automático o Machine Learning (ML), y el Aprendizaje Profundo o Deep Learning (DP). En este TFG, con el objetivo de tratar imágenes OCT, se ha optado por la utilización de las Redes Neuronales Convolucionales, las cuales se encuentran en el marco del Aprendizaje Profundo. Merece la pena destacar cómo este campo de estudio, solo resulta ser un subconjunto de todo lo que abarca la Inteligencia Artificial, tal y como muestra la Figura 1.1

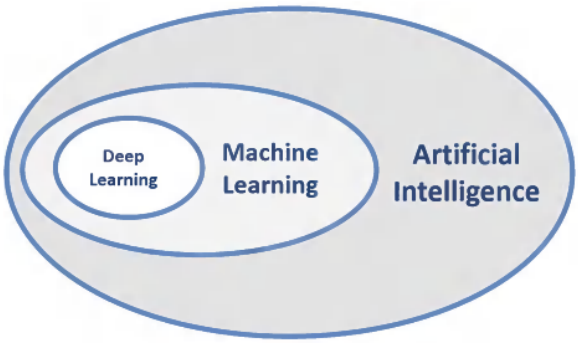


Figura 1.1: Relación entre IA, ML y DP. Fuente: [9]

De esta manera, este TFG pretende emprender un estudio en la posible aplicación de estas técnicas de Aprendizaje Profundo en la diagnosis de enfermedades oculares, aplicada de forma específica sobre imágenes OCT. Asimismo, se persigue explorar uno de los temas de mayor controversia en las Redes Neuronales: su explicabilidad.

Desde su concepción, las Redes Neuronales Artificiales, han sido criticadas por ser una caja negra [10], la cual esconde los procesos internos y los patrones que utiliza para tomar

decisiones. De esta manera, se pretende indagar en este campo, con el objetivo de mejorar la interpretabilidad del comportamiento de la red neuronal, frente a los diagnósticos obtenidos.

1.4. Alcance del proyecto

Este es un proyecto orientado al estudio, investigación y desarrollo. Con una carga teórica importante, se centra en el estudio de aplicación de Redes Neuronales Convolucionales para la diagnosis de enfermedades retinianas detectables, mediante Tomografías de Coherencia Óptica. El alcance de este trabajo contempla los siguientes objetivos:

- Análisis y tratamiento del conjunto de imágenes, adecuándolas a las necesidades del modelo por desarrollar.
- Diseño, implementación y evaluación de un clasificador, fundamentado en Redes Neuronales Convolucionales. Este debe, no solo cumplir con una alta tasa de clasificación correcta, sino también demostrar una robustez y adaptabilidad frente a las variaciones en los datos de entrada.
- Realizar una investigación en profundidad sobre la interpretabilidad del clasificador propuesto. Se pretende indagar en la 'caja negra' que suelen representar estas tecnologías, intentando extraer patrones subyacentes que guían su toma de decisiones.
- Finalmente, se contempla el diseño, desarrollo y despliegue de una aplicación web, que presente de manera accesible y comprensible el funcionamiento del clasificador desarrollado. Esta aplicación no tendrá que limitarse a ofrecer un diagnóstico basado en los datos de entrada, sino que también deberá proporcionar una visualización que facilite la interpretación de los fundamentos de dicho diagnóstico.

1.5. Estructura

Este trabajo de fin de grado está estructurado en los siguientes capítulos:

- **Capítulo 1: Introducción** - Se presentan el contexto, la motivación del proyecto y se definen los objetivos a alcanzar.
- **Capítulo 2: Gestión del proyecto** - Se describe la planificación y organización del trabajo, incluyendo la estimación de tiempo y gestión de riesgos.

- **Capítulo 3: Fundamento teórico** - Se detallan las bases teóricas necesarias para el entendimiento del proyecto, centrándose especialmente en las Redes Neuronales Convolucionales y su explicabilidad.
- **Capítulo 4: Plataforma de trabajo** - Se describen y justifican las herramientas software y hardware que han sido utilizadas durante el desarrollo del proyecto.
- **Capítulo 5: Conjunto de datos** - Se explica el origen, características y tratamiento del conjunto de datos empleado para el entrenamiento y evaluación del clasificador.
- **Capítulo 6: Construcción de la Red Neuronal Convolucional** - Se detalla el diseño e implementación de la Red Neuronal Convolucional desarrollada.
- **Capítulo 7: Resultados** - Se exponen los resultados obtenidos, incluyendo una evaluación de la efectividad y rendimiento del clasificador.
- **Capítulo 8: Aplicación** - Se describe el análisis y diseño de la aplicación web desarrollada, destacando sus funcionalidades y su integración con el clasificador.
- **Capítulo 9: Conclusiones** - Se reflexiona sobre los resultados obtenidos, se discuten las dificultades encontradas, las lecciones aprendidas y se sugieren líneas futuras de trabajo.

Capítulo 2

Gestión del proyecto

Dadas las dimensiones de este TFG, es necesario realizar una labor de gestión y planificación que se adecuen a los objetivos del trabajo. En este capítulo, se abordarán la metodología, planificación y gestión de riesgos del proyecto.

2.1. Metodología

Para seleccionar una metodología de trabajo, que se adapte a las necesidades de este proyecto, primero hay que identificar la naturaleza del mismo. Este TFG tiene un fuerte peso en el área de la Minería de Datos, pues el objetivo principal del mismo es aprovechar una tecnología, cuyo principal fin es la explotación y extracción de conocimiento mediante la introducción masiva de datos. De esta forma, se realiza una exploración para identificar posibles metodologías. Entre las consideradas, se encuentran SEMMA (Sample, Explore, Modify, Model, Assess)[11], TDSP (Team Data Science Process)[12] y CRISP-DM (Cross-Industry Standard Process for Data Mining)[13][14] siendo esta última la metodología finalmente aplicada en este proyecto.

CRISP-DM es una metodología flexible y muy popular en los proyectos de Minería de Datos, siendo considerada un estándar. Su estructura consta de seis fases principales:

1. **Comprensión del negocio** - La fase inicial del proyecto se centra en obtener un entendimiento de los objetivos y requisitos del proyecto, así como su motivación.
2. **Comprensión de los datos** - Aquí se busca el entendimiento de los datos, y su naturaleza, para que se puedan explotar al máximo.
3. **Preparación de los datos** - En este apartado, se describen todos aquellos procesos que transforman el conjunto de datos inicial en el final, que será usado para entrenar al modelo.

4. **Modelado** - Obtención del modelo con sus correspondientes hiperparámetros, que serán calibrados hasta encontrar los óptimos.

5. **Evaluación** - Antes de pasar a la fase final, es importante realizar una evaluación más exhaustiva del modelo obtenido de la fase anterior.

6. **Despliegue** - Para acabar, una vez que el modelo está listo, es necesario presentarlo de una forma utilizable para un usuario final.

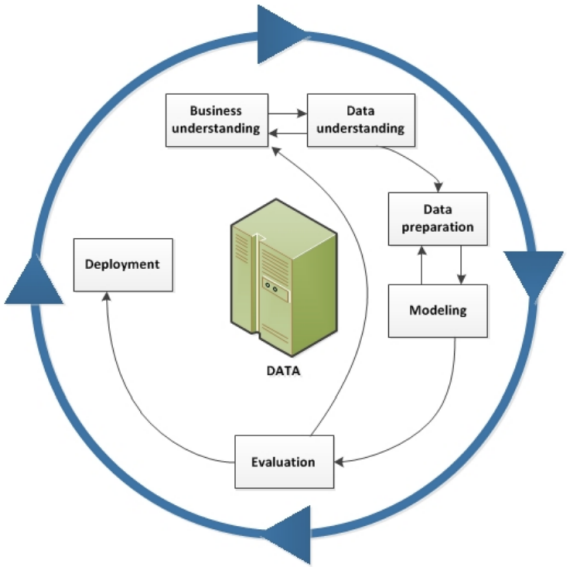


Figura 2.1: Ciclo de vida de un proyecto. Fuente: [13].

Como se puede apreciar en la Figura 2.1, una de las características más notables de CRISP-DM es su carácter cíclico, lo que significa, que el proceso puede repetirse tantas veces como sea necesario. Esto es algo fundamental, dado que la Minería de Datos, y más ampliamente la Ciencia de los Datos, son áreas que se benefician enormemente de la iteración y el refinamiento.

En lo relativo al flujo de trabajo, se considera la posibilidad de abordarlo mediante metodologías ágiles[15]. Sin embargo, debido a la inexistencia de un equipo con un mínimo de integrantes, se considera que este proyecto no cumple con los requisitos para explotar el potencial de estas metodologías, siendo más apropiado un enfoque tradicional. Se plantea, entonces, un desarrollo incremental, en el cual se proponen una serie de hitos; de esta manera, se puede organizar y estructurar el trabajo de forma efectiva y manejable.

El desarrollo incremental, a diferencia de los métodos ágiles, se basa en la entrega de piezas completas y funcionales del proyecto en cada fase, proporcionando una base sólida y consistente, sobre la que construir el trabajo final. Este enfoque se alinea de manera óptima con la naturaleza de este TFG, complementándose eficazmente con las etapas previstas en la metodología CRISP-DM.

2.2. Entregables

Siguiendo el enfoque de desarrollo incremental adoptado, se producirá un entregable completo y funcional al concluir cada hito. Cada nuevo entregable se construirá sobre la base de los anteriores, añadiendo características y funcionalidades adicionales. Este proceso continuará hasta que se complete el proyecto en su totalidad. Así, se garantiza un avance gradual y controlado, permitiendo la revisión, evaluación y mejora constantes del trabajo realizado.

- **Hito 1** - Comprensión de los datos, procesado y clasificador básico.
- **Hito 2** - Desarrollo de un clasificador óptimo.
- **Hito 3** - Exploración en la explicabilidad del clasificador.
- **Hito 4** - Despliegue del modelo desarrollado mediante el análisis, diseño y desarrollo de una aplicación web.
- **Hito 5** - Redacción de la memoria del Trabajo de Fin de Grado.

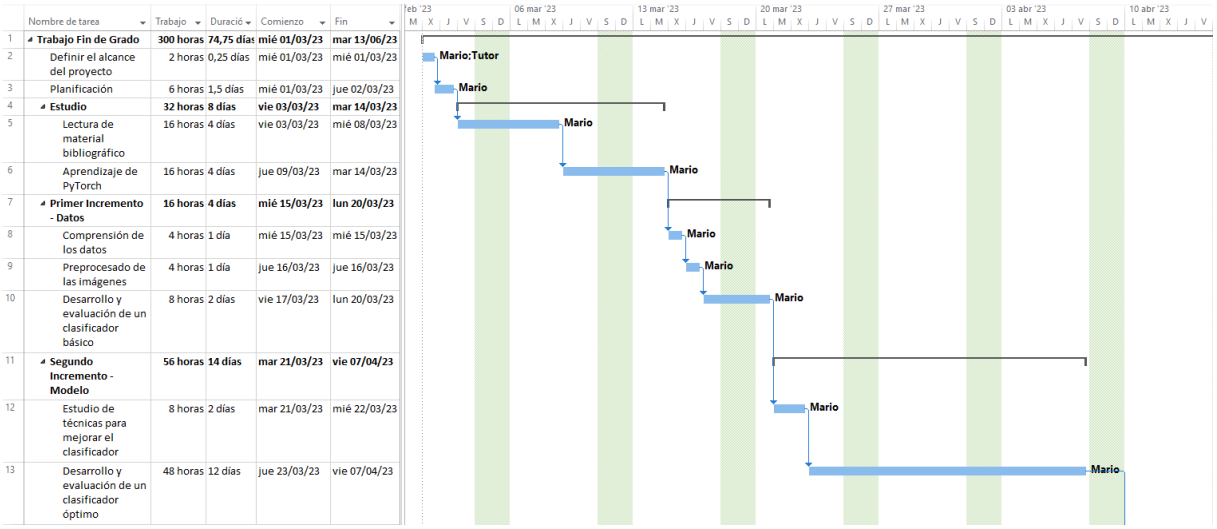
2.3. Planificación

En esta sección se dispone una planificación detallada de la realización del proyecto, incluyendo una estimación de coste temporal de sus tareas.

Inicialmente, se plantea un trabajo diario de cuatro horas que, si bien no ha podido ser respetado fielmente todos los días, se cumple en promedio. El diagrama de Gantt desglosado en las figuras 2.2, 2.3 y 2.4, muestra la consecución de tareas, así como el tiempo de realización estimado para cada una de ellas.

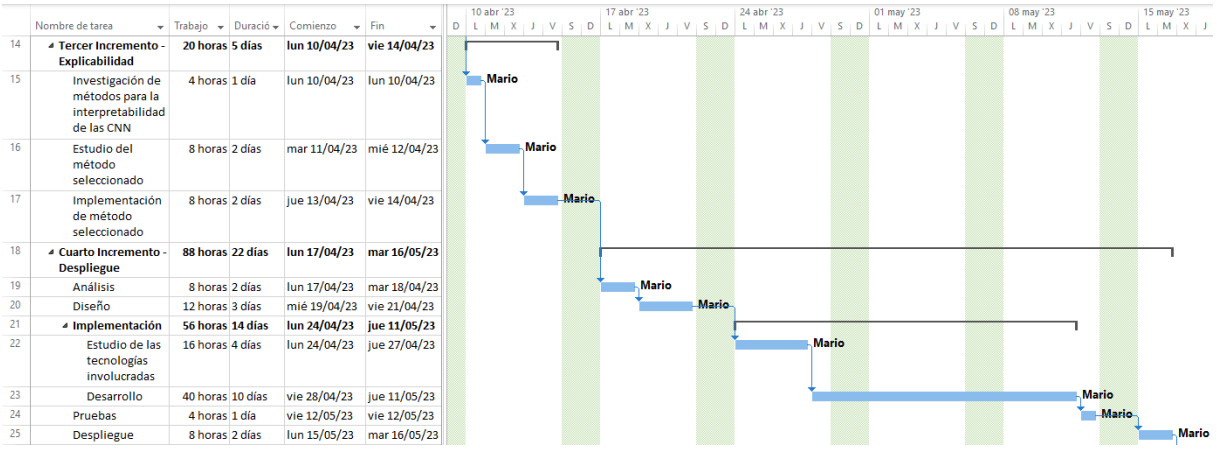
El desarrollo de este TFG da comienzo el 1 de marzo del 2023, con una finalización prevista para el 13 de junio del 2023. No obstante, existen una serie de factores que han propiciado leves retrasos en la estimación inicial:

- **Tiempo de entrenamiento** - El gran volumen del conjunto de datos, sumado a la falta de un hardware más adecuado, han propiciado entrenamientos de larga duración. Por ello, ha sido necesario una mejora del hardware disponible y un proceso de optimización mayor al previsto.
- **Falta de experiencia** - Debido a la novedad de frameworks como PyTorch o Flask, el tiempo dedicado a su aprendizaje, ha resultado mayor al estimado. Adicionalmente, la insuficiente experiencia previa con Redes Neuronales Convolucionales, provoca un aumento en el tiempo estimado para su estudio.



	Nombre de tarea	Trabajo	Duració	Comienzo	Fin
1	**Trabajo Fin de Grado**	**300 horas**	**74,75 días**	**mié 01/03/23**	**mar 13/06/23**
2	Definir el alcance del proyecto	2 horas	0,25 días	mié 01/03/23	mié 01/03/23
3	Planificación	6 horas	1,5 días	mié 01/03/23	jue 02/03/23
4	**Estudio**	**32 horas**	**8 días**	**vie 03/03/23**	**mar 14/03/23**
5	Lectura de material bibliográfico	16 horas	4 días	vie 03/03/23	mié 08/03/23
6	Aprendizaje de PyTorch	16 horas	4 días	jue 09/03/23	mar 14/03/23
7	**Primer Incremento - Datos**	**16 horas**	**4 días**	**mié 15/03/23**	**lun 20/03/23**
8	Comprensión de los datos	4 horas	1 día	mié 15/03/23	mié 15/03/23
9	Preprocesado de las imágenes	4 horas	1 día	jue 16/03/23	jue 16/03/23
10	Desarrollo y evaluación de un clasificador básico	8 horas	2 días	vie 17/03/23	lun 20/03/23
11	**Segundo Incremento - Modelo**	**56 horas**	**14 días**	**mar 21/03/23**	**vie 07/04/23**
12	Estudio de técnicas para mejorar el clasificador	8 horas	2 días	mar 21/03/23	mié 22/03/23
13	Desarrollo y evaluación de un clasificador óptimo	48 horas	12 días	jue 23/03/23	vie 07/04/23

Figura 2.2: Inicio del proyecto e incrementos 1 y 2.



	Nombre de tarea	Trabajo	Duració	Comienzo	Fin
14	**Tercer Incremento - Explicabilidad**	**20 horas**	**5 días**	**lun 10/04/23**	**vie 14/04/23**
15	Investigación de métodos para la interpretabilidad de las CNN	4 horas	1 día	lun 10/04/23	lun 10/04/23
16	Estudio del método seleccionado	8 horas	2 días	mar 11/04/23	mié 12/04/23
17	Implementación de método seleccionado	8 horas	2 días	jue 13/04/23	vie 14/04/23
18	**Cuarto Incremento - Despliegue**	**88 horas**	**22 días**	**lun 17/04/23**	**mar 16/05/23**
19	Análisis	8 horas	2 días	lun 17/04/23	mar 18/04/23
20	Diseño	12 horas	3 días	mié 19/04/23	vie 21/04/23
21	**Implementación**	**56 horas**	**14 días**	**lun 24/04/23**	**jue 11/05/23**
22	Estudio de las tecnologías involucradas	16 horas	4 días	lun 24/04/23	jue 27/04/23
23	Desarrollo	40 horas	10 días	vie 28/04/23	jue 11/05/23
24	Pruebas	4 horas	1 día	vie 12/05/23	vie 12/05/23
25	Despliegue	8 horas	2 días	lun 15/05/23	mar 16/05/23

Figura 2.3: Incrementos 3 y 4.

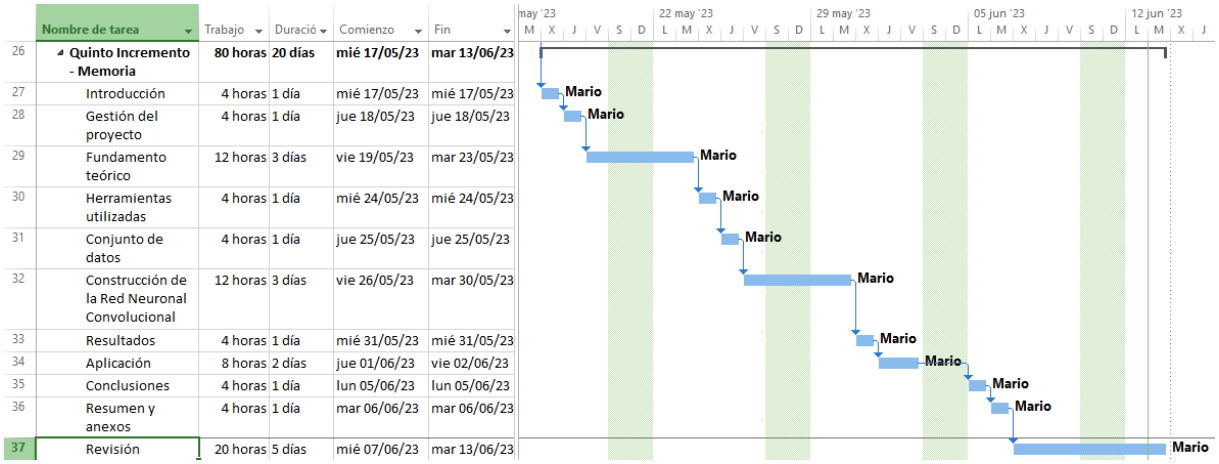


Figura 2.4: Incremento final.

Capítulo 3

Fundamento teórico

A lo largo de este capítulo, se establecerán las bases necesarias para comprender los desarrollos posteriores presentados en el TFG. La definición y comprensión de estos conceptos, proporcionará una sólida introducción teórica para un correcto seguimiento del trabajo experimental.

3.1. Introducción a las Redes Neuronales Artificiales

Comenzaremos introduciendo los conceptos fundamentales de las Redes Neuronales Artificiales. Si bien no nos adentraremos exhaustivamente, nos centraremos en aquellos que son relevantes para comprender las llamadas Convolucionales, que son ampliamente utilizadas en diversas aplicaciones de visión por computadora. Para comprenderlas, es crucial familiarizarse con algunos conceptos importantes que se describen a continuación.

3.1.1. El modelo de McCullot y Pitts

En su concepción inicial, las Redes Neuronales Artificiales tenían como objetivo proponer un modelo computacional inspirado en nuestro sistema nervioso. Esta idea se basa en la premisa de que el poder computacional no reside en el funcionamiento individual de una sola neurona, sino en la extensa conectividad que existe entre ellas. De esta manera, en 1943 Warren McCulloch y Walter Pitts presentan la Neurona Artificial.

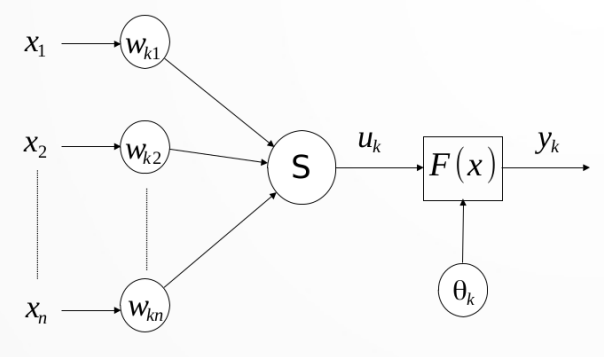


Figura 3.1: Neurona Artificial de McCullot y Pitts. Fuente: [16].

En la Figura 3.1 se describe la estructura de una Neurona Artificial. Se puede observar que la neurona está compuesta de varios elementos, pero el cálculo interno realizado en esta, no es más que una función que, ante una entrada, produce una determinada salida.

$$u_k = \sum_{j=0}^{n-1} w_{kj} x_j + w_{kn} \tag{3.1}$$

Esta operación se puede interpretar como una suma ponderada de cada uno de los elementos de la entrada, es decir, para la neurona k existen una lista de pesos w_{kn}, donde el peso w_{kj} multiplica al elemento x_j de la entrada. La variación de dicho peso permite asignarle una importancia a la entrada asociada. De esta manera, los parámetros w_{kn} serán ajustados de manera automática durante el proceso de entrenamiento de la Red Neuronal Artificial, hasta encontrar un valor óptimo.

Finalmente, falta describir el elemento típicamente conocido como función de activación, $F(x)$. La motivación de esta función es que el verdadero potencial de estas neuronas surge al combinarlas unas con otras. Sin embargo, si observamos detenidamente 3.1, se puede apreciar que el cálculo es el de una simple regresión lineal, y el efecto de concatenar múltiples funciones lineales de forma consecutiva da como resultado otra función lineal. Este hecho causa que, sin el añadido de $F(x)$, utilizar una concatenación de neuronas tendría el mismo resultado que aplicar una sola. De esta manera, se introduce $F(x)$, la cual aplica una transformación no lineal a u_k, aumentando de forma significativa la complejidad de los patrones detectados por la red al aumentar su profundidad.

Existen numerosas funciones de activación, como la sigmoide, la *tanh*, *arctg*, Rectificador Lineal (ReLU), entre otras[16]. Con alguna excepción, estas suelen ser:

- **No lineales** - Pues esta es la propiedad que le confiere su utilidad, como ya se ha comentado.

- **Diferenciables** - Esto es necesario para el algoritmo de aprendizaje. En la práctica hay alguna excepción, como la función *ReLU*, sin embargo, existen técnicas para solventar este problema.
- **Acotadas** - Es una condición suficiente para garantizar la estabilidad numérica de los algoritmos de ajuste de pesos.

3.1.2. Rectificador Lineal

Dadas sus peculiaridades respecto a otras funciones de activación, y a su uso en este TFG como Función de Activación principal, dedico este apartado a la exposición de las características del Rectificador Lineal.

La función Rectificador Lineal o ReLU, definida como:

$$f(x) = \text{máx}(0, x) \tag{3.2}$$

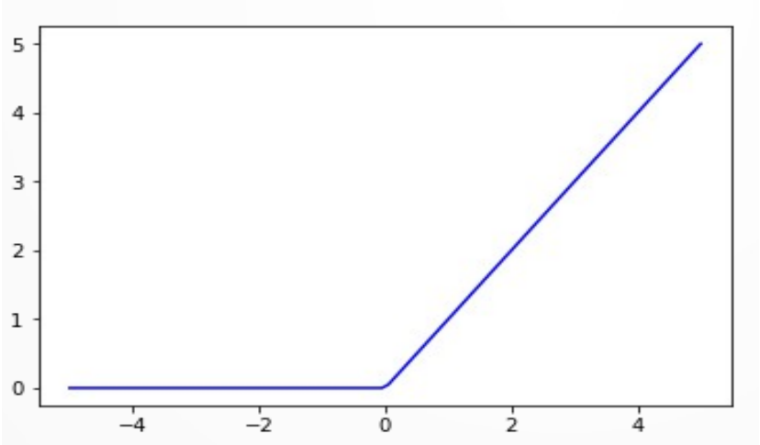


Figura 3.2: Rectificador Lineal. Fuente: [16].

La función ReLU es, desde el 2000, una de las funciones de activación más exitosas y ampliamente utilizadas [16][17]. Una sus grandes ventajas respecto a las demás funciones es que ayuda a paliar un problema asociado a las Redes Neuronales Artificiales, conocido como Evanescencia del Gradiente [16]. Adicionalmente, aquellas que implementan ReLU, son más fácilmente optimizables en comparación con la Sigmoide o la *tanh* [17]. Esto provoca una mayor velocidad en la convergencia de los modelos que implementan ReLU, lo cual es de gran utilidad, especialmente en el Aprendizaje Profundo.

Por otro lado, el Rectificador Lineal presenta algunos problemas. Uno de ellos es que no es derivable en $x = 0$. A pesar de esto, como ya se comentó, en la práctica existen técnicas que las librerías de Aprendizaje Automático implementan para paliar este problema. Otro

inconveniente de ReLU es que no está acotada en R^+, sin embargo, experimentalmente se demuestra que no da problemas cuando se mantienen las entradas acotadas en el intervalo $[0, 1]$ [16].

3.1.3. Aprendizaje Profundo

En el marco del Aprendizaje Automático, existen una gran cantidad de algoritmos ampliamente utilizados. Sin embargo, uno de los problemas típicos asociados a este tipo de algoritmos antes del surgimiento del Aprendizaje Profundo, es que son fuertemente dependientes de un proceso previo de "Ingeniería de Características". Esta es una disciplina que estudia el tratamiento previo de los datos, antes de ser introducido al modelo de Aprendizaje Automático. Su principal inconveniente es que resulta ser muy laboriosa, además de estar poco automatizada, requiriendo la intervención de un experto en la materia. El Aprendizaje Profundo, por contraparte, es una extensión de las Redes Neuronales Artificiales que ayuda a mitigar este problema. Como ya se expuso en la Figura 1.1, el Aprendizaje Profundo es un subconjunto de las técnicas de Aprendizaje Automático que está muy ligado a las Redes Neuronales. Su nombre se debe a la profundidad obtenida al interconectar capas de Neuronas Artificiales.

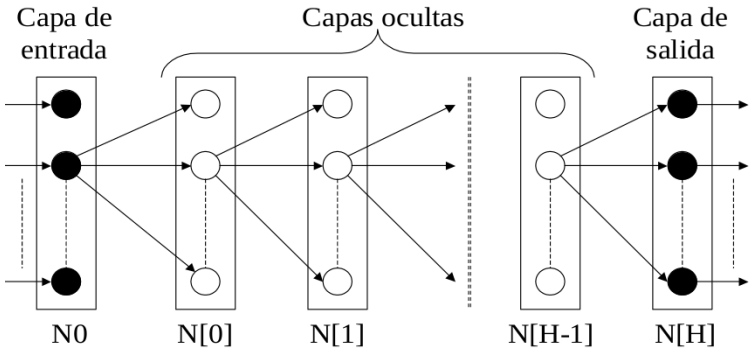


Figura 3.3: Perceptrón Multicapa. Fuente: [18].

La Figura 3.3 representa una red densamente conectada o Perceptrón Multicapa (MLP, por sus siglas en Inglés). La arquitectura del MLP define capas de neuronas, cuyas salidas están completamente conectadas con los nodos de la capa siguiente. El Perceptrón Multicapa está constituido por una capa de entrada, una capa de salida, y N capas ocultas con un número variable de neuronas. Conforme N aumenta, la profundidad y cantidad de conexiones de la red, la otorgan una mayor expresividad, esto es, mayor capacidad de modelar y aprender patrones complejos [19].

La característica distintiva del Aprendizaje Profundo es su notable capacidad para la abstracción, permitiendo al modelo manipular directamente los datos en bruto, sin procesar. Esta habilidad se fundamenta en su capacidad de aprender representaciones

en diversos niveles de abstracción. En las primeras capas de la red, el modelo aprende características de bajo nivel. A medida que se avanza hacia la salida, se van detectando patrones como combinación de las formas simples aprendidas en capas precedentes. De esta manera, las características se tornan cada vez más abstractas y específicas, ajustándose a la tarea dada de antemano.

Debido a esto, el Aprendizaje Profundo ofrece un ahorro significativo en tiempo y esfuerzo en comparación con las técnicas tradicionales de Aprendizaje Automático, eliminando la necesidad de una laboriosa Ingeniería de Características manual. Asimismo, los patrones que estos modelos aprenden automáticamente, suelen ser más efectivos y versátiles que los que se diseñan manualmente. Sin embargo, existen peligros en relegar esta tarea directamente al modelo. Como trataremos más adelante, si no se procede con cautela, podríamos terminar con un modelo fuertemente sesgado, que otorgue importancia desmedida a características que son escasamente relevantes para el problema planteado.

3.1.4. Clasificación

Este Trabajo Final de Grado se enfoca en un problema de clasificación. Si bien existe un amplio espectro de metodologías dentro del ámbito de las redes neuronales, este trabajo no buscará detallarlas todas. En su lugar, esta memoria se centrará en describir aquellos procesos y metodologías específicas, que son esenciales para entender y desarrollar el presente TFG.

Codificación OneHot

En un problema de multiclasificación, como el que se aborda en este TFG, la capa de salida de un Perceptrón Multicapa tendrá tantas neuronas como posibles categorías. De esta manera, si se tienen n clases posibles, la red, con n neuronas en su última capa, producirá como salida un vector de tamaño n. A este vector se le aplicará una codificación comúnmente conocida como *OneHot*. Consiste transformar, el vector de salida, en un vector binario, donde el componente máximo pasara a ser igual a uno y el resto de elementos serán remplazados por cero [18].

Entrenamiento Supervisado

La metodología de entrenamiento más ampliamente utilizada en el Aprendizaje Automático es el Entrenamiento Supervisado. Este enfoque se caracteriza en que, para cada entrada que recibe el modelo, existe una salida deseada conocida. El proceso consiste en mostrar una entrada al modelo, posteriormente comparar su salida obtenida con la deseada.

Para realizar esta comparación es necesario definir una *Función Objetivo* o *Función de Pérdida*, la cual mide el error entre la salida obtenida y la deseada [19]. Existen numerosas alternativas, una ampliamente extendida es el *Error Cuadrático Medio* [18]:

$$E^p = \frac{1}{2} \sum_{i=0}^{N[H]-1} (d_i^p - y_{Hi}^p)^2 \tag{3.3}$$

De esta manera, el entrenamiento no es más que un proceso iterativo, en el cual una muestra, o lote de ellas, es procesado por la red, proceso conocido como *propagación hacia adelante*. Posteriormente, con la salida obtenida y la deseada, se calcula la función de error. Finalmente, ocurre un proceso conocido como *propagación hacia atrás* o *Retropropagación*, el cual, en función del error obtenido, actualizará los pesos de cada neurona acercando el modelo a su configuración paramétrica óptima.

Propagación hacia adelante

Es el primer proceso que se lleva a cabo en cada iteración del entrenamiento de un MLP. Cuando se presenta una nueva muestra a la red, cada neurona procesa de manera progresiva la entrada, realizando los cálculos previamente descritos hasta obtener la salida final en forma de vector en la última capa de la red.

Retropropagación

Una vez el error ha sido calculado, es necesario modificar los parámetros de la red, incrementando o decrementado su valor, tratando de reducir el error para la próxima iteración. En definitiva, el proceso de aprendizaje es un problema de optimización de la *Función de Pérdida*. Para lograrlo, es necesario aplicar un algoritmo iterativo; un ejemplo de esto es el *Descenso Estocástico del Gradiente* o *SGD (Stochastic Gradient Descent).*

El Descenso Estocástico del Gradiente

En esencia, es un algoritmo de optimización iterativo, encargado de medir en qué grado hay que modificar los parámetros de la Red Neuronal Artificial. Para este fin, se apoya en el concepto del vector gradiente. Cuando se calcula en un punto, se obtiene un vector con la dirección y sentido de máximo crecimiento de la función en ese punto. En una Red Neuronal, lo que se trata de optimizar es la *Función de coste* (E). Para ello, en cada iteración, se calcula el coste o error asociado a la muestra procesada (E^p) y se computa el vector gradiente, para posteriormente, modificar los pesos en la dirección marcada por el gradiente. De esta manera, siendo w_{ij} el peso j asociado a la neurona i, tras procesar la

muestra p, sufrirá la variación indicada en 3.4 [19][20]. Dado que el objetivo es minimizar la función, se aplica el sentido contrario del gradiente, de ahí el signo negativo en 3.4.

$$\Delta_p w_{ij} = -\gamma \frac{\partial E^p}{\partial w_{ij}} \tag{3.4}$$

Con el fin de calcular estas derivadas parciales, se hace uso de la *Regla de la Cadena*. Siendo y_i^p la salida de la neurona i y F la Función de Activación:

$$y_i^p = \sum_{j=0}^{n-1} w_{ij} x_j^p + w_{in} \tag{3.5}$$

$$\frac{\partial E^p}{\partial w_{ij}} = \frac{\partial E^p}{\partial F_i} \frac{\partial F_i}{\partial y_i^p} \frac{\partial y_i^p}{\partial w_{ij}} \tag{3.6}$$

Por último, en 3.4 se puede observar un valor γ conocido como *Factor de Aprendizaje*. Este es un hiperparámetro que se multiplica por el vector gradiente, afectando en el grado en que se modifican los pesos en cada iteración.

3.1.5. Problemas de las Redes Neuronales Artificiales

Coste computacional

Como se ha expuesto anteriormente, el marco teórico que sustenta esta tecnología, lleva disponible desde 1943. Sin embargo, existe un problema intrínseco a la naturaleza de las redes neuronales, que ha provocado, que la máxima explotación de esta tecnología no se haya podido materializar de forma efectiva hasta tiempo después de su concepción.

Uno de los factores que destacan y contribuyen a este hecho, es el gran coste computacional asociado a los algoritmos que se encargan del entrenamiento de las Redes Neuronales Artificiales. Uno de los avances producidos en este milenio, que ha propulsado la evolución del Aprendizaje Profundo, ha sido la mejora y disponibilidad de hardware para el procesamiento en paralelo, en particular, las Unidades de Procesamiento Gráfico o GPU.

Estos sistemas, que son ampliamente utilizados en la industria de los videojuegos, un mercado enormemente competitivo, han experimentado una disminución en su precio gracias a la economía de escala. Además, estas tarjetas gráficas multiprocesador sobresalen en la realización de multiplicaciones rápidas de matrices y vectores, operaciones fundamentales no solo para la creación de realidades virtuales convincentes, sino también para el entrenamiento de las Redes Neuronales.

Este avance en la capacidad de procesamiento y la reducción de costes han sido vitales para el desarrollo del Aprendizaje Profundo. Gracias a las GPU y a sus implementaciones en Redes Neuronales, se han logrado importantes éxitos en áreas como el reconocimiento de patrones, la segmentación de imágenes o la detección de objetos. Esto demuestra cómo las mejoras en el hardware y su accesibilidad económica, han jugado un papel decisivo en la evolución del Deep Learning [21].

Explicabilidad

Las Redes Neuronales Artificiales de forma general, a medida que aumenta su profundidad, destacan sobre otros métodos tradicionales del Aprendizaje Automático, en la capacidad de abstracción y en la complejidad de los patrones que pueden aprender. Sin embargo, una de las mayores críticas a las que se enfrentan es que, una vez entrenadas, su funcionamiento o razonamiento es completamente opaco, esto es, son una "caja negra" [10].

Una vez disponemos de una Red Neuronal entrenada, si observamos su estructura y parámetros, la única información accesible es el valor numérico de millones de pesos ajustados. Lamentablemente, parece imposible utilizar estos valores para deducir la extracción de características complejas realizada de manera interna en la red. Esto es un problema, pues el poder predictivo de la red parece ser inversamente proporcional a explicabilidad de esta. La Figura 3.4 muestra un mapa de la relación entre el poder predictivo de la red y su explicabilidad. En ella, se enumeran una serie de técnicas disponibles para profundizar en la interpretabilidad de los distintos modelos. En particular, este TFG abordará en profundidad el algoritmo *Grad-CAM*, el cual resulta de gran utilidad en la interpretabilidad de las Redes Neuronales Convolucionales.

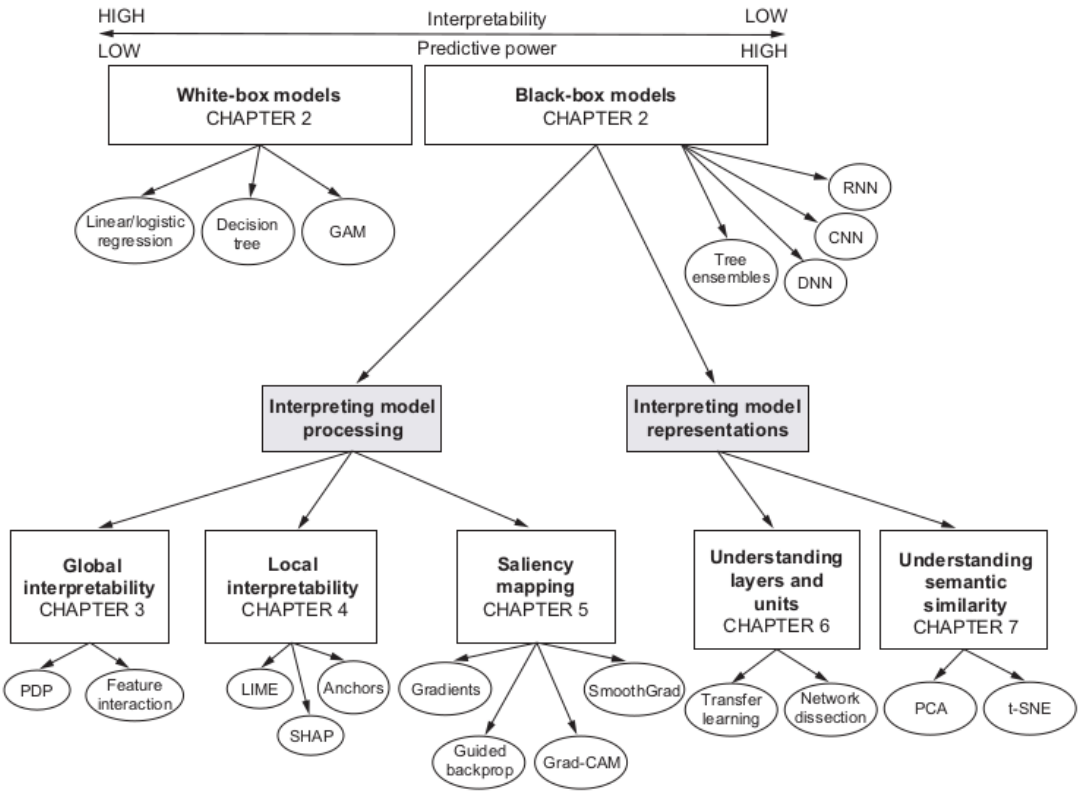


Figura 3.4: Mapa de interpretabilidad. Fuente: [22].

3.2. Redes Neuronales Convolucionales

Tras haber abordado las bases y conceptos principales sobre los que se sustentan las Redes Neuronales Artificiales, en esta sección se profundiza el caso particular de las Redes Neuronales Convolucionales, que son un tipo especial pertenecientes al marco del Aprendizaje Profundo. Se caracterizan por el uso de una operación matemática que otorga su nombre: la *Convolución*. Estas redes, resultan especialmente útiles para tratar datos con una estructura matricial o de rejilla [23]

3.2.1. Motivación

Para poder abordar un problema de clasificación de imágenes mediante un Perceptrón Multicapa, es imprescindible adaptar la imagen al formato de entrada requerido por este tipo de red. Este proceso, que se conoce como *aplanado*, se encarga de transformar la representación matricial de la imagen en un vector unidimensional. Sin embargo, una desventaja notable de esta metodología radica en que cada elemento del vector, que en general corresponde al valor asociado a un píxel, es tratado de manera aislada del resto.

En otras palabras, se pierde la información bidimensional de la imagen.

De esta manera, las Redes Neuronales Convolucionales surgen como una solución especialmente adecuada en el procesamiento de este tipo de información, basada en la localidad espacial.

3.2.2. La Convolución

La convolución, representada con un asterisco, es una operación matemática que realiza una transformación sobre dos funciones. Su aplicación en las Redes Neuronales es algo especial, pues se trata de un cálculo de naturaleza discreta.

Suponemos la convolución aplicada a una imagen representada como una matriz bidimensional I. Para ello, es necesario una segunda matriz bidimensional conocida de forma general como *Kernel* (K). Ambas matrices suelen conocerse como tensores. La convolución discreta aplicada a la imagen I y el kernel K da como resultado otra matriz F [23]:

$$F(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n) \tag{3.7}$$

Dado que el Kernel es generalmente de un tamaño menor que la imagen, para lograr un menor rango de los valores válidos de m y n se suele aplicar la propiedad conmutativa de la convolución.

$$F(i,j) = (K * I)(i,j) = \sum_m \sum_n I(i-m, j-n)K(m,n) \tag{3.8}$$

En muchas librerías de Aprendizaje Automático, aunque es llamada convolución, en su lugar se implementa una operación relacionada denominada *correlación-cruzada*, que resulta más sencilla de computar [23].

$$F(i,j) = (K * I)(i,j) = \sum_m \sum_n I(i+m, j+n)K(m,n) \tag{3.9}$$

En este sentido, en la expresión de correlación cruzada, el kernel puede ser concebido como una ventana móvil. En su operación, se desplaza de manera metódica de izquierda a derecha y de arriba a abajo a lo largo de los ejes de la imagen, multiplicando, en cada iteración, cada componente del kernel por el valor del pixel asociado y finalmente sumándolos todos. Cada posición adoptada por el kernel da lugar a un valor de la matriz resultado F, como se ilustra en la Figura 3.5. A partir de este momento, no se hará

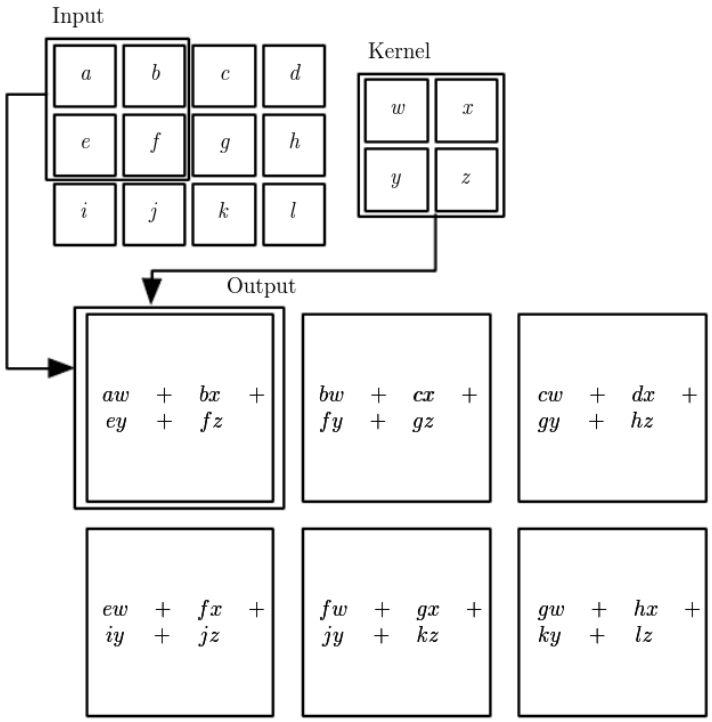


Figura 3.5: Acción del Kernel. Fuente: [23].

distinción entre la correlación cruzada y la convolución, pues a efectos prácticos en las CNN, pueden considerarse equivalentes.

En una Red Neuronal Convolucional, el tensor resultante al aplicar una convolución se denomina *Mapa de Características* o *Feature Map*. Esto se debe a que el efecto aplicar un kernel a una imagen es otra imagen, la cual, en función de la configuración del kernel, es decir, sus parámetros (En el caso de 3.5 *w, x, y* y *z*), puede enfatizar zonas de la imagen equivalentes a ciertas características. La intuición subyacente a estas redes es que, mediante la aplicación concatenada de convoluciones en una red profunda, se podrán extraer características complejas. Los mapas de las primeras capas extraerán características simples y, a medida que crece la profundidad de la red, las propiedades del Aprendizaje Profundo permiten la abstracción y reconocimiento de patrones más complejos. La Figura 3.6 ilustra esta idea. En ella se representa un gato y los distintos niveles de reconocimiento de características. Las primeras capas reconocen figuras simples, como líneas rectas, diagonales, etc. Finalmente, las capas más cercanas a la salida pueden aprender a detectar patrones más complejos, como un ojo o la nariz.

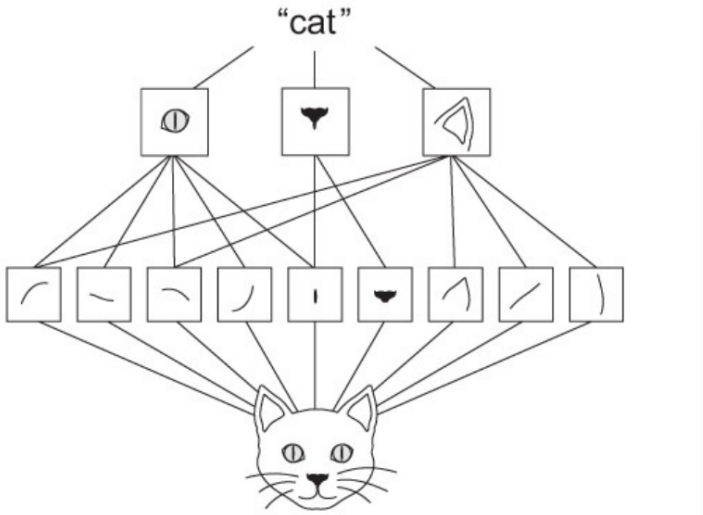


Figura 3.6: Extracción de características. Fuente: [24].

3.2.3. La Convolución aplicada

Propiedades de la Convolución

En su aplicación, las Redes Neuronales Convolucionales aprovechan algunas propiedades para mejorar su eficiencia. Gracias al principio de localidad, cuando un kernel se desplaza a través de una imagen para realizar la convolución, se centra en una pequeña región cada vez, considerando únicamente la información ahí localizada. Esto le permite detectar características como los bordes de una figura, a pesar de que se utilicen kernels de tamaño mucho menor que el de la imagen. Esto introduce el concepto de *Conectividad Dispersa*. Usualmente, dadas m entradas y n salidas, la multiplicación matricial requerirá $m \times n$ operaciones, lo cual conlleva el empleo de un algoritmo de complejidad $\mathcal{O}(m \times n)$. Sin embargo, dado que el tamaño del kernel, por lo general, es varios órdenes de magnitud menor que el tamaño de la imagen, esto se podría reducir a $\mathcal{O}(k \times n)$, siendo k mucho menor que m. La Figura 3.7 muestra de manera grafica una comparación en la conectividad cuando s es producida mediante una convolución con un kernel (3×3) y dos capas completamente conectadas [23].

Asimismo, el hecho del que el tamaño del kernel sea menor que el de la imagen, no solo ahorra complejidad computacional, sino que también disminuye la memoria necesaria para almacenar parámetros, sumándose esto a otra de las propiedades de este tipo de redes: los *Parámetros Compartidos*. Dado que un mismo kernel, con un tamaño menor al de la imagen, es utilizado sobre todas las regiones de la imagen, da lugar a la reutilización o compartición de los parámetros incluidos en el kernel. En una red convencional, cada uno de los parámetros se emplea una única vez, al ser multiplicado por su entrada asociada.

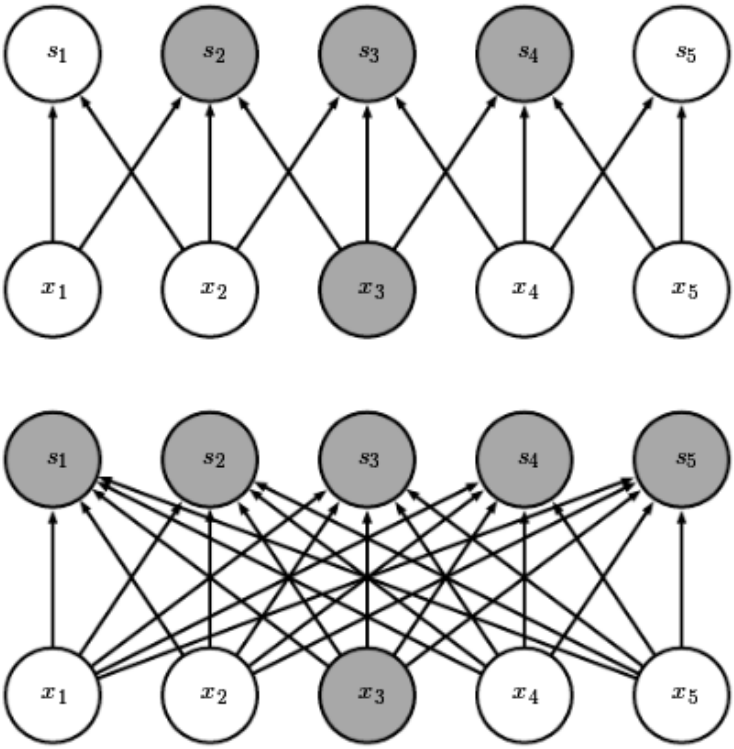


Figura 3.7: Conectividad Dispersa. La figura muestra las salidas en s afectadas por la entrada x resaltada. Fuente: [23].

Esto difiere de la utilización de un parámetro del kernel en las Redes Neuronales Convolucionales, ya que cada parámetro es aplicado sobre cada pixel de la imagen a excepción, en ocasiones, de los bordes. Esta propiedad no tiene efecto en términos de complejidad temporal, pero sí mejora los requisitos de memoria. La Figura 3.8 muestra, en su parte superior, un parámetro destacado, el cual se aplica a varias entradas, en contraposición al parámetro destacado de la parte inferior de la figura, que se utiliza únicamente una vez en una red completamente conectada.

Esta idea de compartir parámetros a lo largo de la imagen resulta intuitivamente útil. Un ejemplo de esto puede ser que, un kernel especializado en la detección de bordes, podrá hacerlo en las distintas partes de la imagen. En particular, la forma en la que se comparten los parámetros de una Red Neuronal Convolucional, da como resultado una propiedad adicional conocida como *Equivarianza a la Traslación*, es decir, si la entrada recibe una transformación de translación, la salida de la capa recibirá la misma traslación. A efectos prácticos, esto permite que si un objeto se traslada en una imagen de entrada, se desplazará de igual forma en la salida, siendo la detección de dicho objeto independiente. Sin embargo, existen otras transformaciones no equivariantes, como el cambio de escala o la rotación [23].

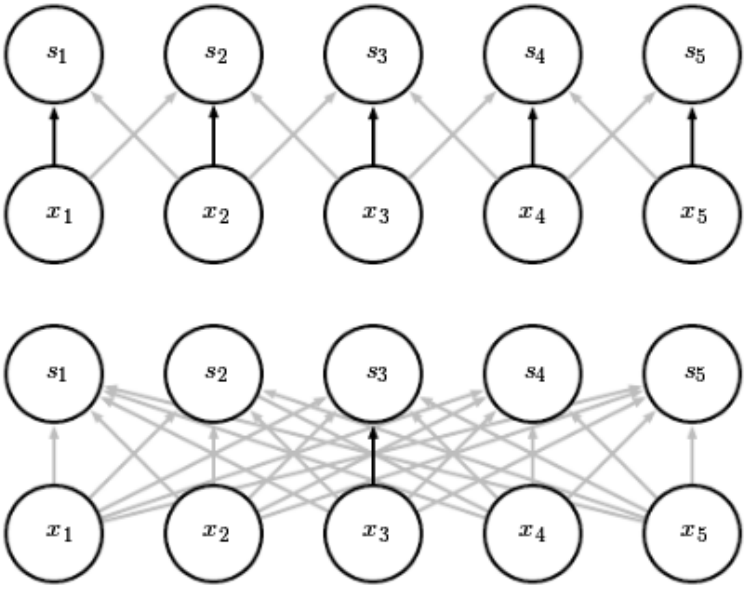


Figura 3.8: Parámetros Compartidos. Fuente: [23].

Convolución de imágenes a color

El proceso de convolución descrito hasta el momento, ejemplifica la operación aplicada a una imagen en escala de grises, suponiendo que cada pixel de una imagen es representado con un único valor numérico. Sin embargo, la representación de imágenes a color no sigue esta suposición; en su lugar, cada pixel tiene tres valores asociados, intensidad de rojo, de verde y de azul, conocido como código RGB. Dada esta representación, cada imagen estará conformada por tres matrices o *Canales Cromáticos*, uno para cada color, los cuales al ser superpuestos darán lugar a la imagen original. El esquema de funcionamiento de una convolución en una imagen RGB dispone tres kernels para cada imagen. Cada uno de ellos será aplicado sobre su canal cromático de la misma manera que se ha expuesto hasta ahora. Finalmente, el mapa de características resultante será la superposición o suma de estos. Este planteamiento se ejemplifica en la Figura 3.9

La operación de convolución previamente definida, puede ahora reformularse para ser aplicada a una imagen RGB 3.10. Siendo $F_{c,i,j}$ el valor del Feature Map de salida, en el canal cromático c, fila i y columna j; I la imagen de entrada, con el mismo formato que F y con el mismo número de canales cromáticos, y por último, consideramos K como un tensor tridimensional, igualmente con una dimensión para el canal, y dos dimensiones para la localización espacial del valor [23].

$$F_{c,i,j} = \sum_{m=1} \sum_{n=1} (I_{c,i+m-1,j+n-1} K_{c,m,n}) \tag{3.10}$$

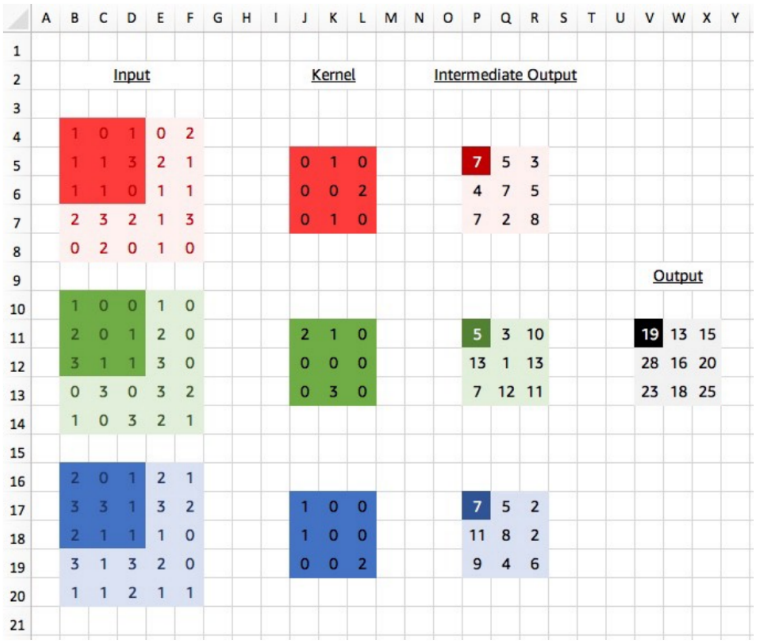


Figura 3.9: Convolución de imágenes a color con kernel 3×3. Fuente: [24].

3.2.4. Movimiento del kernel

Como ya se ha comentado anteriormente, el kernel convencionalmente puede describirse como una ventana móvil, que recorre la imagen bidimensional de izquierda a derecha y de arriba a abajo. Sin embargo, existen algunos hiperparámetros que pueden modificar su comportamiento.

Stride

Tal y como se ha descrito el movimiento del kernel hasta el momento, experimenta un desplazamiento de una unidad en cada iteración. En ocasiones, con el objetivo de reducir coste computacional, puede resultar interesante aumentar la magnitud del desplazamiento del kernel, como se muestra en la Figura 3.10, esto es conocido como *Stride* o *Paso*. Podemos modificar la expresión 3.10, incluyendo el Stride 3.11. Calcular el elemento $F_{c,i,j}$, implica que el kernel se ha desplazado $(i-1)$ veces a la derecha y $(j-1)$ veces hacia abajo, empezando el conteo en 1 [23].

$$F_{c,i,j} = \sum_{m=1} \sum_{n=1} (I_{c,(i-1)\times s+m,(j-1)\times s+n} K_{c,m,n}) \tag{3.11}$$

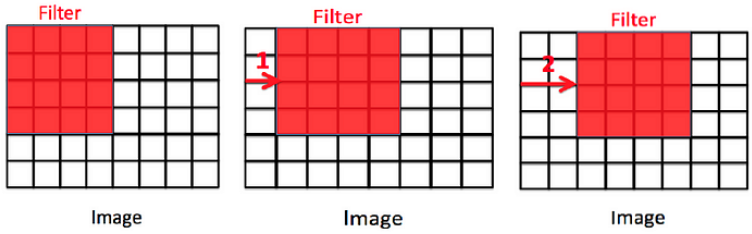


Figura 3.10: Representación de un Stride = 1 y un Stride = 2. Fuente: [25].

Padding

En la operación convolucional descrita, el kernel se desplazará a la derecha siempre que quede completamente contenido en los bordes de la imagen. Esto provoca que los píxeles de los bordes de la imagen no serán centrales en el kernel, por tanto, la información contenida en los límites de la imagen tendrá menos peso en el aprendizaje, lo cual puede ser un problema de haber información relevante. Adicionalmente, el proceso de convolución descrito hasta el momento genera un fenómeno conocido como *Submuestreo* o *Downsampling*. Este fenómeno implica que el tamaño del mapa de características resultante de la convolución, será estrictamente menor que el de la imagen de entrada. En particular, para un *stride* $= 1$, siendo m la anchura de imagen de entrada y k la anchura del kernel, el resultado de la convolución dará lugar a un mapa de características de anchura $m-k+1$. Esto significa, que si $k > 1$, en una cadena de convoluciones, eventualmente la salida será 1×1, siendo el tamaño del kernel un limitante del número de capas del que puede disponer la Red Neuronal Convolucional [23].

Como solución a estos problemas se presenta el *Padding*, una técnica que consiste en ampliar el tamaño de la imagen con píxeles transparentes, o ceros. De esta manera, se podrá desplazar el kernel hasta incluir la información de los bordes de la imagen y controlar la disminución de del tamaño del mapa de características resultante, pudiendo, incluso, lograr que este sea igual al tamaño de la imagen de entrada [23]. Siendo p el padding aplicado, la anchura del mapa de características ahora podrá calcularse cómo $m - k + 2p + 1$.

3.2.5. Submuestreo

Como se ha comentado en la sección anterior, las Redes Neuronales Convolucionales tienden a submuestrear, esto quiere decir que en cada convolución el tamaño de la salida es menor que el de la imagen de entrada, a excepción de que este comportamiento sea modificado con el padding. Las expresiones 3.12 y 3.13 definen la altura y anchura del mapa de características de salida $F_{h\times w}$ en función del de la entrada $I_{h\times w}$, el kernel $K_{h\times w}$,

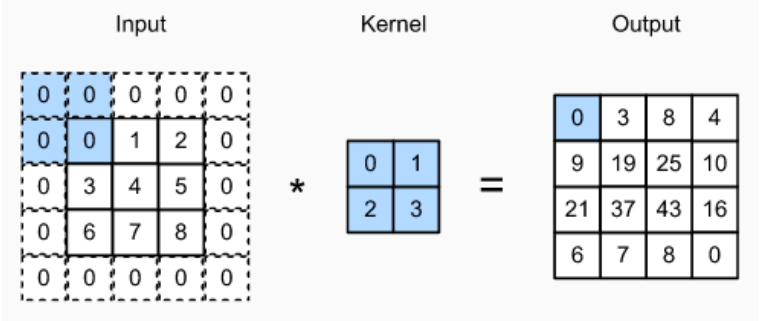


Figura 3.11: Convolución bidimensional con padding y kernel 2×2. Fuente: [26].

el padding p y el stride s.

$$F_h = \frac{I_h - K_h + 2p}{s} + 1 \tag{3.12}$$

$$F_w = \frac{I_w - K_w + 2p}{s} + 1 \tag{3.13}$$

El hecho de que los mapas de características reduzcan su tamaño tiene, una penalización en la expresividad de la Red Neuronal [23]. Sin embargo, esto provoca que en las capas más profundas de la red, el coste computacional de la convolución, así como la memoria requerida para almacenar los mapas de características, se vean fuertemente reducidos. Esto puede resultar una gran ventaja en términos computacionales. Por esta razón, es común aplicar en las Redes Neuronales Convolucionales una técnica que refuerza el submuestreo conocida como *Pooling*.

Pooling

Es una operación de submuestreo, que se aplica generalmente entre dos capas convolucionales. Reemplaza la salida de la red en una ubicación determinada con un resumen estadístico de las salidas cercanas [23]. Se puede entender como una ventana móvil similar al kernel, cuyo tamaño suele tomar valores pares, que se desplazará por el tensor de entrada efectuando una operación que agrupe píxeles vecinos. Dos de las funciones de pooling más utilizadas son el Max Pooling y el Average Pooling, los cuales calculan el valor máximo y la media aritmética de paso por el filtro respectivamente, tal como muestran la Figuras 3.12 y 3.13.

La aplicación del pooling tiene varias ventajas. La más inmediata es la mejora del coste computacional, ya que siendo n la anchura del filtro de pooling, la entrada de la capa siguiente será n veces menor que el tamaño de la salida de la capa anterior. Adicionalmente, ayuda a conferir a la red de una invarianza aproximada a pequeñas traslaciones de la entrada [23].

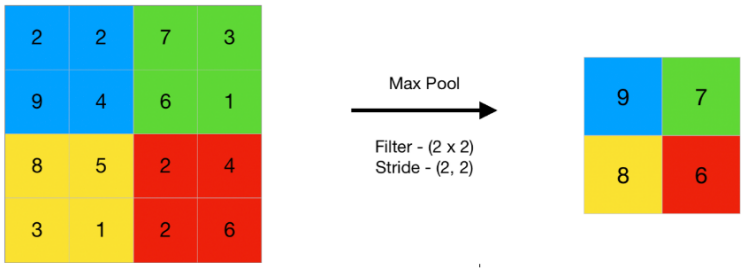


Figura 3.12: Max Pooling 2×2. Fuente: [27].

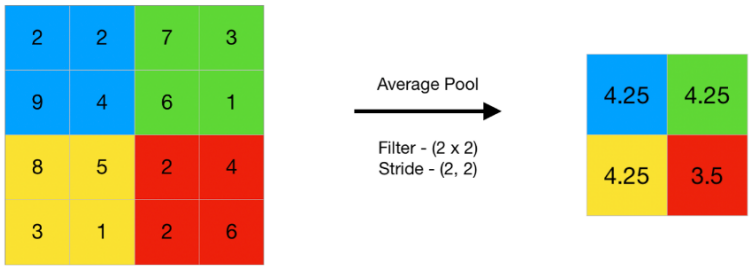


Figura 3.13: Average Pooling 2×2. Fuente: [27].

3.2.6. Arquitectura de las Redes Neuronales Convolucionales

Cuando se aplican las convoluciones en una Red Neuronal, cada capa no describe una única convolución, en su lugar, varias de ellas son computadas. Esto se debe a que una convolución, con un único kernel, solo es capaz de extraer una característica en varias localizaciones espaciales de la imagen. Generalmente, es necesario que cada capa pueda extraer varias características distintas en varias posiciones. Dada una imagen de entrada, se le aplicarán n convoluciones en una capa, lo cual generará n mapas de características, formando cada uno de estos parte de la entrada de la capa siguiente. Como se puede apreciar en este comportamiento, a medida que se profundiza en la red, el número de mapas de características crece de forma exponencial. Esto puede resultar un problema en términos de coste computacional; sin embargo, gracias al submuestreo intrínseco de estas redes y al pooling, los mapas de características son cada vez más pequeños, contribuyendo a la mitigación de este problema.

La estructura de una capa en una arquitectura general de una Red Neuronal Convolucional consiste en varias fases: una primera, en la que se aplican las convoluciones, seguido de una función de activación, que otorgue a la red la no linearidad ya comentada. Tras esto, una capa de submuestreo o Pooling suele ser empleada.

Como muestra la Figura 3.15, en una arquitectura general, cada capa convolucional se conecta con la siguiente hasta que, finalmente, los mapas de características de la última capa son aplanados y pasados a una red densamente conectada o Perceptrón Multicapa,

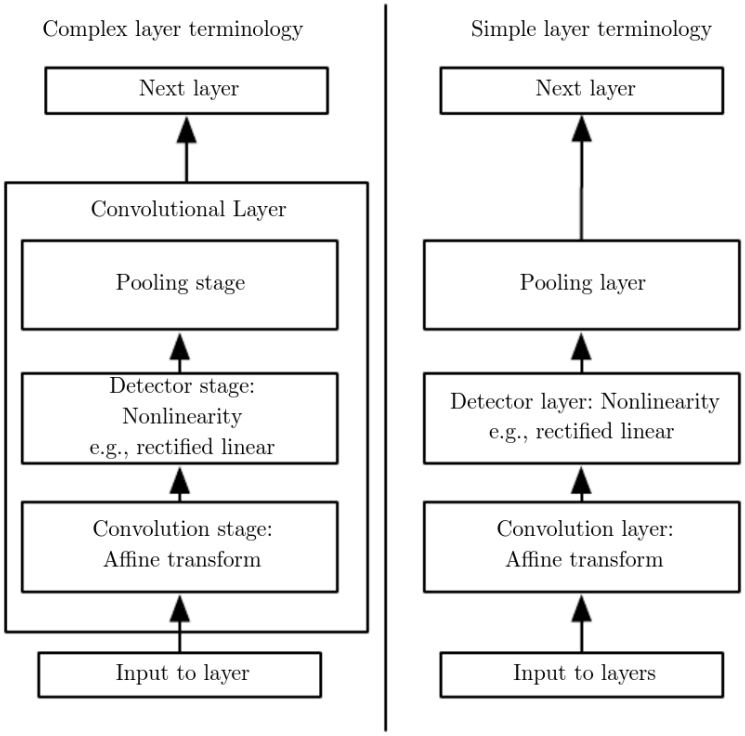


Figura 3.14: Estructura típica de una capa convolucional. Fuente: [23].

cuya capa de salida tendrá n neuronas, siendo n el número de clases, o una neurona en el caso de la clasificación binaria.

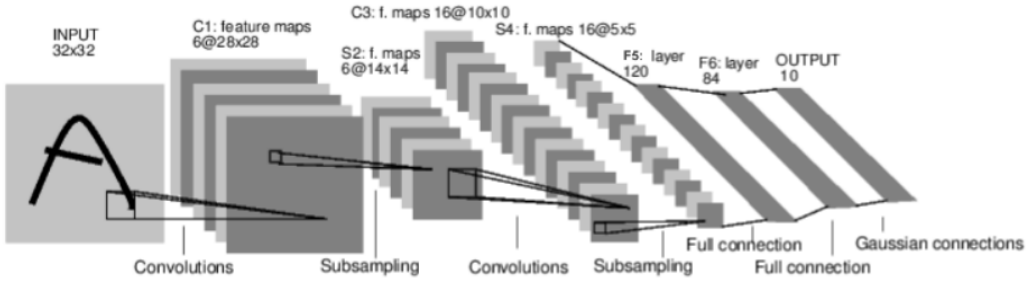


Figura 3.15: Arquitectura general de una Red Neuronal Convolucional. Fuente: [24].

Arquitectura VGG

Visual Geometry Group (VGG) es una de las arquitecturas de CNN más populares. Ha sido utilizada en una gran cantidad de modelos de reconocimiento de imágenes, dando sorprendentes resultados. En general, las más aplicadas son la VGG-16 y la VGG-19, que se corresponde con una profundidad de 16 capas y 19 capas respectivamente [28]. Las

capas convolucionales de la arquitectura VGG emplean un kernel de tamaño mínimo, un filtro 3×3 es el más pequeño capaz de visualizar las direcciones espaciales. Esto provoca que la reducción de tamaño tras la convolución sea mínima, lo cual se traduce en la capacidad de agregar más capas. VGG relega la expresividad de la red en su profundidad, permitiendo que los kernels de las primeras capas aprendan características sencillas, las cuales aumentan en complejidad a medida que se avanza hacia la salida de la red. Con el objetivo de conservar la resolución espacial tras cada convolución, el Stride está fijado a 1. La función de activación empleada en esta arquitectura es la ReLU 3.1.2, tanto en la red completamente conectada, como detrás de cada convolución [28]. Finalmente, la función Max Pooling 2×2 (3.2.5) se intercala entre las capas convolucionales tal y como se muestra en la Figura 3.16.

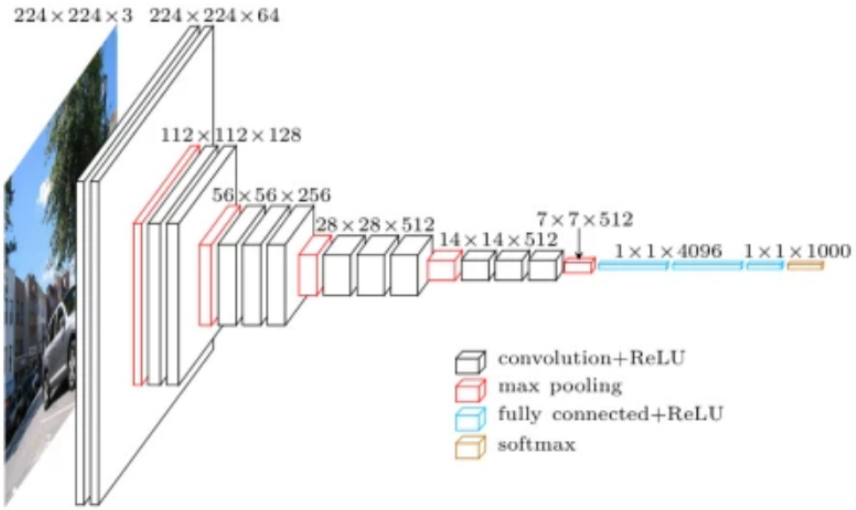


Figura 3.16: Arquitectura VGG. Fuente: [28].

Asimismo, esta arquitectura propone el aumento del número de convoluciones realizadas en paralelo en cada capa, a medida que aumenta la profundidad de la red, aprovechando la reducción de la complejidad computacional de las convoluciones debido al submuestreo. En particular, se proponen 64 convoluciones en la primera capa, aumentando en potencias de dos tras cada capa de *Pooling* hasta las 512 convoluciones por capa [24].

3.2.7. Estabilización y Regularización

Batch Normalization

Cuando los valores de entrada no están normalizados, puede dar lugar a componentes de gran magnitud o de un valor ínfimo. Esto puede perjudicar gravemente en la precisión

del modelo y su velocidad de convergencia. La normalización de los datos de entrada ayuda a mitigar estos problemas; sin embargo, podemos encontrar escenarios en los que este fenómeno surja entre capas ocultas [29]. Asimismo, debido a los cambios en los parámetros de la red durante el entrenamiento, la distribución de las activaciones de la red puede verse afectada. Esta situación es conocida como *Internal Co-variate Shift* [30].

Una técnica que surge para paliar estos problemas es la Normalización por Lote o *Batch Normalization*. Su funcionamiento reside en una normalización aplicada en cada capa del modelo sobre el lote actual. Esto ayuda a estabilizar el modelo reduciendo el *Internal Co-variate Shift*.

El primer paso es calcular el valor medio del lote y su varianza:

$$\mu_B = \frac{1}{m}\sum_{i=1}^{m} x_i \tag{3.14}$$

$$\sigma_B^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_B)^2 \tag{3.15}$$

Finalmente, se calcula la entrada normalizada:

$$\overline{x}_i = \frac{(x_i - \mu_B)}{\sqrt{\sigma_B^2 + \epsilon}} \tag{3.16}$$

$$BN_{\gamma,\beta}(x_i) \equiv \gamma\overline{x}_i + \beta \tag{3.17}$$

La expresión 3.16 es conocida como *Hard Normalization*. Al añadir los parámetros γ y β en 3.17, permitimos a la red identificar los mejores parámetros para la normalización [29].

Drop-Out

La regularización es una técnica utilizada en las redes neuronales empleada, en general, para evitar el Sobreajuste u *Overfitting*. Este es un problema común en el Aprendizaje Automático, donde un modelo se ajusta demasiado a los datos de entrenamiento y, como resultado, tiene un rendimiento pobre en los datos de prueba, esto es, los no mostrados al sistema.

Una forma típica de combatir el Overfitting es el *Bagging*, abreviatura de *Bootstrap Aggregating*, una técnica de ensamblado en Aprendizaje Automático. En Bagging, se crean múltiples modelos (por ejemplo, árboles de decisión, redes neuronales, etc.) en

subconjuntos de datos seleccionados al azar con reemplazo (*Bootstrap*). Posteriormente, sus predicciones son agregadas, normalmente tomando una votación mayoritaria, en el caso de la clasificación. Sin embargo, al tratar con Redes Neuronales Convolucionales, debido a su coste computacional, la creación y entrenamiento de varios modelos resulta altamente ineficiente [23].

El Dropout surge como una posible solución. Es una técnica de regularización en la que aleatoriamente se escoge un porcentaje especificado de activaciones, igualándolas a cero durante el entrenamiento, es decir, su contribución en la propagación hacia adelante y hacia atrás en la red se elimina durante una pasada. Cada neurona, por lo tanto, tiene la posibilidad de ser excluida del proceso de aprendizaje en cualquier iteración [29]. Esto implica que, en cada iteración, se está entrenando una *Red Submuestreada*.

Cuando se mira desde esta perspectiva, el Dropout puede verse como una forma de Bagging, en el que cada red submuestreada es un modelo. Al final del entrenamiento, todas estas redes submuestreadas están "ensambladas" para formar la red final. Sin embargo, a diferencia del Bagging tradicional, las redes submuestreadas comparten parámetros, lo que mantiene bajo el costo computacional, resultando de especial utilidad en las Redes Neuronales [23].

3.2.8. Gradient-weighted Class Activation Mapping

Tal y como se ha expuesto en las secciones anteriores, cuando una imagen es procesada por una Red Neuronal Convolucional, esta es expuesta a múltiples convoluciones y capas de Pooling para la extracción de características. Debido a esto, una CNN entrenada posee potencialmente millones de parámetros ajustados, y durante el procesado de una imagen, millones de operaciones tienen lugar. En consecuencia de esto, realizar un trazado de los motivos que han propiciado el resultado en la clasificación de una imagen resulta excesivamente complejo. Sin embargo, existen algunas técnicas que pueden ayudar en esta tarea.

Los *Métodos basados en Gradiente* son usados para visualizar el gradiente de la clase objetivo respecto a la imagen de entrada. La idea subyacente es alimentar el modelo entrenado con una imagen, obteniendo el resultado computado por la red. A continuación, se aplica el algoritmo de Retropropagación para calcular el gradiente de la salida respecto a la imagen de entrada. En este planteamiento, el gradiente es utilizado como una medida que pondera la influencia de cada pixel en el resultado obtenido. Cuando un gradiente es elevado, la acción de modificar levemente el valor del pixel asociado repercutirá fuertemente en el resultado de la red. De esta manera, aquellos píxeles con un gradiente asociado de mayor magnitud son considerados de gran relevancia en la obtención de la predicción [22].

Por otro lado, los *Métodos basados en Activaciones*, obtienen los mapas de caracte-

rísticas de la última capa convolucional, la cual recoge la mayor cantidad de patrones complejos. Estos son ponderados en importancia, utilizando como pesos el gradiente del resultado calculado con respecto a los mismos. Finalmente, se emplean para generar un mapa de calor, que representa la influencia de cada zona de la imagen en el cómputo de la predicción. La técnica descrita es conocida como *Gradient-weighted Class Activation Mapping* o *Grad-CAM* [22].

Una vez empleado el algoritmo de *Backpropagation* para calcular el gradiente de los mapas de características de la última capa convolucional, se obtiene un tensor 4D con la forma G_{i,j,K_h,K_w}, donde i es el número de canales de entrada, j el número de canales de salida y $K_h \times K_w$, el tamaño del kernel. Posteriormente, se computa la media para cada mapa de características, siendo W el vector de pesos unidimensional de tamaño j empleado para ponderar la importancia de cada uno de ellos. Con dicho vector W, se obtiene una media ponderada de los j mapas de características de la última capa. El resultado final es un tensor de dimensiones $(F_h \times F_w)$, siendo este el tamaño de los mapas de características de la última capa [29]. Finalmente, solo resta reescalar el tensor resultante al tamaño de la imagen original [29].

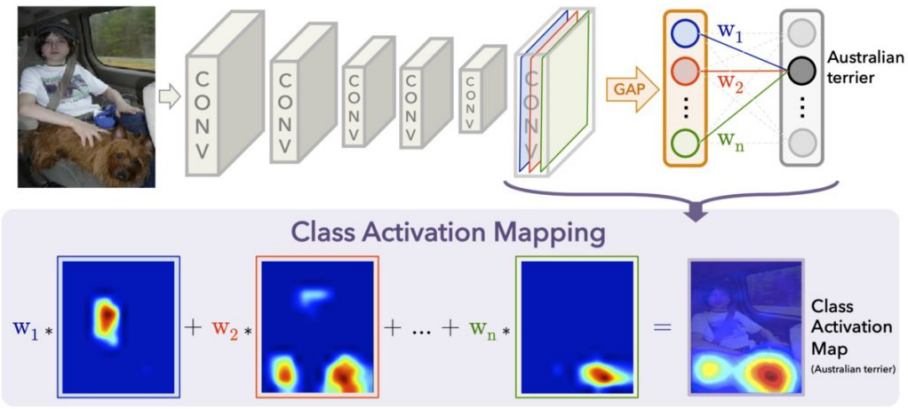


Figura 3.17: Ilustración del algoritmo Grad-CAM. Fuente

La afinidad de este algoritmo es limitada y, dado que solo son empleados los mapas de características de las últimas capas convolucionales, se considera un algoritmo de grano grueso. Sin embargo, Grad-CAM puede resultar de gran utilidad para la interpretabilidad del modelo, en particular para detectar sesgos, uno de los mayores problemas asociados a las Redes Neuronales Artificiales.

Un buen ejemplo de este fenómeno es el caso del conjunto de datos, o *dataset*, conocido como *doctor-nurse*. En la Figura 3.18, la fila superior ilustra un modelo que ha aprendido de manera sesgada, esto es evidente gracias a la visualización proporcionada por Grad-CAM, en los mapas de calor, donde se puede observar que la atención del modelo se centra en las caras. A pesar de que este tipo de información puede ser útil para clasificar el *dataset* en particular, no puede considerarse un factor determinante para la tarea asignada, de tal forma que este comportamiento pone de manifiesto un sesgo en el modelo. En

contraposición, en los mapas de calor de la fila inferior, se puede apreciar que la atención del modelo se dirige más hacia los atuendos y las herramientas profesionales, información que tiene una relación mucho más directa con la tarea que se está llevando a cabo.

Figura 3.18: Ejemplificación de Grad-CAM aplicado a un modelo sesgado y otro no sesgado. Fuente [31]

Capítulo 4

Plataforma de trabajo

A lo largo de este capítulo, se discutirán brevemente las herramientas tanto hardware como software, empleadas para el completo desarrollo de este Trabajo de Fin de Grado. Los proyectos de Aprendizaje Automático en general, y de Aprendizaje Profundo en particular, tienen ciertas características con implicaciones directas en la configuración de la plataforma de trabajo, como se verá a continuación.

4.1. Hardware

Como ya se comentó anteriormente en la Sección 3.1.5, el coste computacional asociado al entrenamiento de un modelo de Deep Learning y en particular, de una Red Neuronal Convolucional para el tratamiento de imágenes, resulta sustancialmente elevado. Para minimizar este inconveniente, es recomendable disponer de una unidad de procesamiento que realice cálculos matriciales a gran velocidad, siendo las Unidades de Procesamiento Gráfico ideales para esta tarea [21].

Sin embargo, lamentablemente no ha resultado posible disponer de una GPU para la realización de este proyecto. En su lugar, los cálculos que tienen lugar durante el entrenamiento se realizarán directamente con el procesador. Por otra parte, la unidad de procesamiento no es el único factor influyente en este proceso. La memoria disponible resulta de vital importancia, pues estos modelos, en general, son entrenados por lotes, y será imprescindible una memoria con la capacidad necesaria para cargar simultáneamente grandes grupos de imágenes, así como un modelo con una gran cantidad de parámetros.

En vista de estas consideraciones, para la realización de este proyecto, el Departamento de Informática de la Universidad de Valladolid ha proporcionado una máquina virtual con las siguientes especificaciones técnicas:

- **Memoria**: La máquina dispone de 32 GiB, lo cual facilitará la carga y procesamiento de grandes lotes de imágenes y permitirá manejar un modelo con un elevado número de parámetros. Esta cantidad de memoria también será útil para mejorar la eficiencia en el cálculo y la manipulación de grandes matrices y tensores, que son fundamentales en el entrenamiento de las redes neuronales.

- **Procesador**: El sistema contaba inicialmente con un procesador de 8 núcleos, el cual, debido a la alta demanda del sistema, fue ampliado a un procesador compuesto por 2 sockets y 6 núcleos cada uno, sumando un total de 12 núcleos. A pesar de no ser una GPU, esta estructura multicore permitirá una cierta paralelización de los cálculos matriciales, crucial en la ejecución eficiente de las redes neuronales.

- **Disco**: El sistema dispone de un disco de 50 GB, el cual resulta más que suficiente para el almacenamiento del conjunto de datos, así como los modelos y resultados generados durante el proceso de entrenamiento y evaluación.

Se debe resaltar que, aunque la disponibilidad de una GPU podría haber mejorado el rendimiento del entrenamiento de los modelos de Deep Learning, las especificaciones de hardware mencionadas anteriormente permiten la realización del proyecto de manera razonable.

4.2. Software

En esta sección se tratarán las herramientas software propuestas y empleadas para el desarrollo de este proyecto.

4.2.1. Sistema Operativo

Al momento de recibir la máquina virtual que se ha utilizado para este proyecto, contaba con Ubuntu 22.04.2 LTS instalado como sistema operativo, una popular distribución de Linux.

El uso de este sistema operativo para un proyecto de Deep Learning, tiene varias ventajas. En primer lugar, Linux es el preferido en el campo de la Ciencia de Datos y el Aprendizaje Automático debido a su robustez, eficiencia y mayor compatibilidad con los Frameworks más populares, como TensorFlow o PyTorch, los cuales tienen un soporte limitado en Windows.

4.2.2. Construcción del modelo

Para el desarrollo de modelos de Aprendizaje Automático y Deep Learning, existen diversas alternativas. Una de ellas es Weka [32], una plataforma que proporciona una interfaz de usuario gráfica, lo que simplifica el proceso de desarrollo y la evaluación de los modelos de Aprendizaje Automático. Sin embargo, al tratarse de un entorno basado en menús, puede no ser suficiente para los objetivos de este proyecto.

Otra alternativa relevante es MATLAB [33], una plataforma de cómputo numérico altamente reconocida. Esta ha sido, y sigue siendo, utilizada para ciertos tipos de análisis y tareas de Aprendizaje Automático, especialmente en el ámbito académico y de investigación. Su amplia gama de bibliotecas, junto con una sintaxis intuitiva orientada al cálculo matricial, hace de MATLAB una opción viable para cierto tipo de proyectos.

Adicionalmente, Julia [34] es una alternativa más reciente que ha ganado popularidad en el campo del Aprendizaje Automático y la Ciencia de Datos. Este lenguaje ha sido diseñado con el rendimiento en mente, y su creciente ecosistema de paquetes, junto con su habilidad para ejecutar cálculos numéricos de forma eficiente, lo convierten en una opción cada vez más atractiva para los profesionales en el campo. Por contraparte, y debido sobre todo a su novedad, la documentación de Julia todavía es escasa, lo cual puede resultar un problema en el desarrollo de un Trabajo de Fin de Grado con limitaciones temporales.

No obstante, a pesar de la disponibilidad de todas estas herramientas, en la Ciencia de Datos y el Aprendizaje Automático, Python [35] ha emergido como el lenguaje de programación predominante en este tipo de trabajos, en particular, en el entorno académico. Es conocido por su sintaxis sencilla y legible, y por su extenso ecosistema de bibliotecas y frameworks para el procesamiento de datos y el Aprendizaje Automático, lo que lo convierte en una elección muy acertada para este proyecto.

Para la consecución de los objetivos de este TFG, Python ofrece bibliotecas como Scikit-learn [36], que proporciona una gran cantidad de algoritmos de aprendizaje supervisado y no supervisado. Por otra parte, se dispone también de TensorFlow [37] o Keras [38], que son frameworks ampliamente utilizados para el desarrollo de modelos de Aprendizaje Profundo.

Sin embargo, para este proyecto se ha optado por utilizar PyTorch [39], que es un framework de Aprendizaje que, no solo proporciona una gran cantidad de funcionalidades para el desarrollo de redes neuronales profundas, sino que, adicionalmente, ofrece una gran flexibilidad y un control detallado sobre el proceso de construcción y entrenamiento de los modelos. Esto lo hace especialmente adecuado para proyectos de investigación y académicos como este Trabajo Fin de Grado.

Una posible desventaja de PyTorch es que tiene una curva de aprendizaje más pronunciada en comparación con alternativas como Keras, que está diseñado para ser fácil de usar. Por contra, tiene una mayor flexibilidad y control, lo que permite implementar

modelos más complejos y personalizados. Esto supone un reto mayor en el proyecto, pero también proporciona una valiosa experiencia de aprendizaje y una mayor comprensión de los procesos internos del Aprendizaje Profundo.

4.2.3. Construcción de la aplicación web

En este TFG, se plantea la construcción de una aplicación web sencilla, que sirva a modo de demostración del funcionamiento del modelo desarrollado. Para esta tarea, se plantean varias alternativas. La primera que surge, es desarrollar en Java, pues es el lenguaje de programación con el que se ha adquirido mayor familiaridad durante el transcurso del Grado en Ingeniería Informática y, en particular, en el desarrollo de aplicaciones web. Sin embargo, se acaba decidiendo que la utilización de un framework para desarrollar directamente en Python puede ser una mejor alternativa, pues la compatibilidad con el modelo propuesto será completa.

En este contexto, y debido a la falta de experiencia en desarrollo web con Python, Flask [40] surge como una alternativa viable. Es un microframework de desarrollo web en Python, el cual facilita las herramientas suficientes para implementar una aplicación que se ajuste a las necesidades de este TFG, manteniendo una curva de aprendizaje menos pronunciada que otras alternativas, lo cual resulta interesante debido a la limitación en el tiempo de desarrollo disponible.

Conjuntamente con Flask, se utilizó una combinación de HTML, CSS y JavaScript para desarrollar la interfaz de usuario de la aplicación web. Estas tecnologías son la base de la mayoría de las aplicaciones web, ya que permiten una gran flexibilidad en cuanto a la presentación y funcionalidad de la interfaz.

Para el despliegue de la aplicación en un entorno de producción, se utilizó Gunicorn [41], un servidor HTTP WSGI para Python, que proporciona una manera eficiente de manejar múltiples solicitudes simultáneas.

Finalmente, para garantizar una mayor portabilidad y facilidad de instalación, se empleó Docker [42]. Este permite empaquetar la aplicación y todas sus dependencias en un contenedor, que puede ser ejecutado en cualquier sistema que tenga Docker instalado, eliminando así los problemas asociados con las dependencias y las diferencias entre los entornos de desarrollo y producción, añadiendo una capa adicional de robustez.

Capítulo 5

Conjunto de datos

Aquí se presenta el conjunto de datos o *Dataset* utilizado para entrenar y probar el modelo. Se comentan las características subyacentes a los datos, su naturaleza, origen y las transformaciones aplicadas para su uso.

5.1. Descripción del conjunto de datos

El conjunto de datos utilizado para el desarrollo de este TFG consiste en un total de 84484 imágenes de secciones transversales de Tomografías de Coherencia Óptica en formato JPEG [4]. Estas imágenes están etiquetadas de acuerdo a la presencia de algunas anomalías retinianas, como son el Edema Macular Diabético [6], la Neovascularización Coroidea [7] y el Drusen [8].

5.1.1. Origen

El Dataset, en su totalidad, ha sido extraído de un reto Kaggle [3], que es una plataforma en línea que proporciona un entorno, donde los científicos de datos y los entusiastas del Aprendizaje Automático pueden participar en competiciones, explorar y construir modelos en un entorno colaborativo, así como compartir sus hallazgos y modelos con la comunidad.

5.1.2. Formato

El conjunto de datos, en su origen, consta de una jerarquía de directorios. En el interior del directorio padre (*OCT2017*), los datos vienen inicialmente separados en tres carpetas

(*test, train y val*) con el objetivo de diferenciar los de entrenamiento de los de prueba en distintos subconjuntos. Cada uno de estos últimos contiene cuatro carpetas, una para cada anomalía (*CNV, DME, DRUSEN y NORMAL*). En su interior, se encuentran todas las imágenes que han sido catalogadas con dicha anomalía. La Figura 5.1 muestra un ejemplo de cada tipo de imagen.

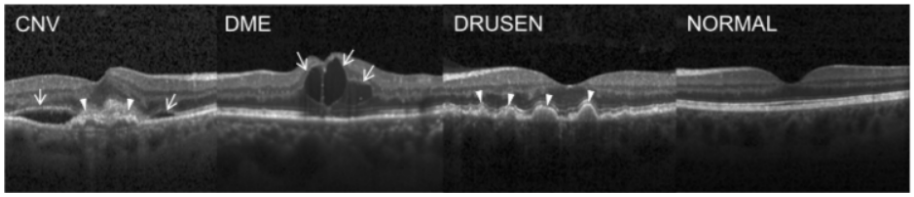


Figura 5.1: Anomalías presentes en el Dataset. Fuente: [3]

5.1.3. Etiquetado

Las imágenes de Tomografía de Coherencia Óptica fueron seleccionadas de cohortes retrospectivas de pacientes adultos de diversas instituciones.

Para el etiquetado, cada imagen fue sometida a un sistema de clasificación en capas con varios niveles de verificadores en creciente experiencia. Cada imagen comenzó con la etiqueta correspondiente al diagnóstico más reciente del paciente. El primer nivel de verificadores estaba compuesto por estudiantes de pregrado y de Medicina que habían aprobado un curso de interpretación de OCT. Estas personas realizaron el control de calidad inicial, excluyendo aquellas imágenes con alteraciones o reducciones significativas de resolución. El segundo nivel consistía en cuatro oftalmólogos, que evaluaron de manera independiente cada imagen que había pasado el primer nivel. Se registró la presencia o ausencia de neovascularización coroidea, edema macular y drusen. Finalmente, un tercer nivel de dos especialistas retinianos sénior, cada uno con más de 20 años de experiencia clínica, verificaron las etiquetas de cada imagen [3].

5.1.4. Distribución

Subconjuntos de datos

Se realiza un análisis para visualizar la distribución de imágenes ofrecida por defecto en los conjuntos de entrenamiento, prueba y validación.

En la Figura 5.2 se puede apreciar una gran diferencia en la proporción de imágenes entre los conjuntos de prueba y entrenamiento.

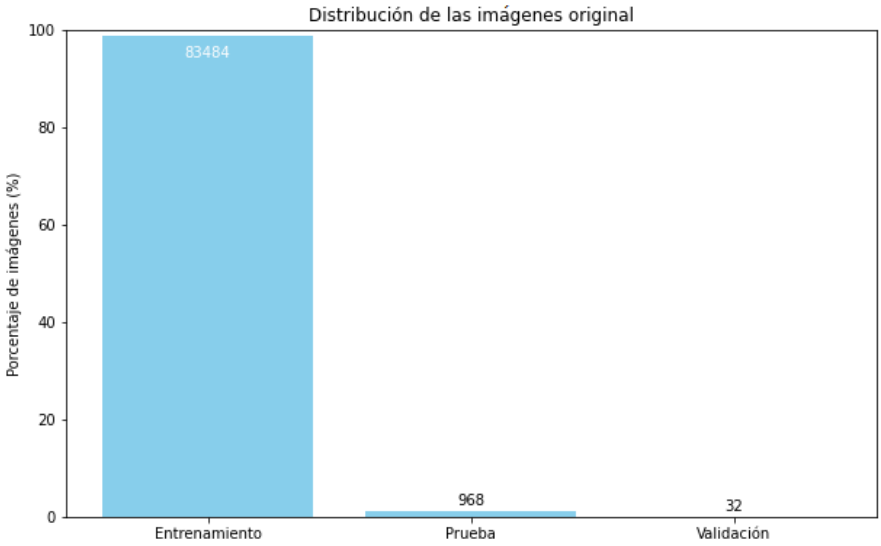


Figura 5.2: Distribución de imágenes original en los conjuntos de prueba, test y validación.

Distribución de clases

Se realiza una exploración del conjunto de datos con el objetivo de visualizar la distribución de clases subyacente. Adicionalmente, se estudia la distribución por separado de cada uno de los subconjuntos y, como se puede observar en la Figura 5.4, el de entrenamiento respeta la distribución de clases global. Sin embargo, la Figura 5.5 muestra una representatividad equivalente para cada clase en los datos de prueba y validación, alterando la distribución del conjunto completo.

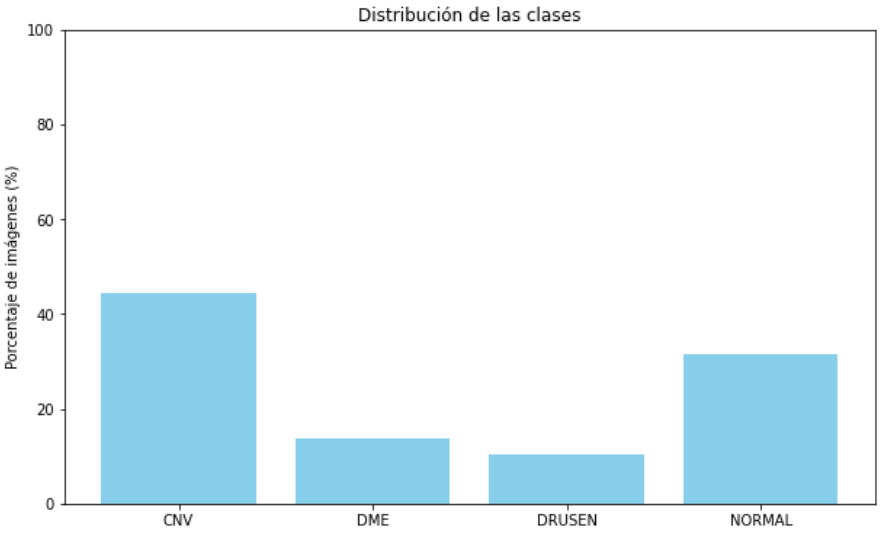


Figura 5.3: Distribución de clases en el conjunto de datos completo.

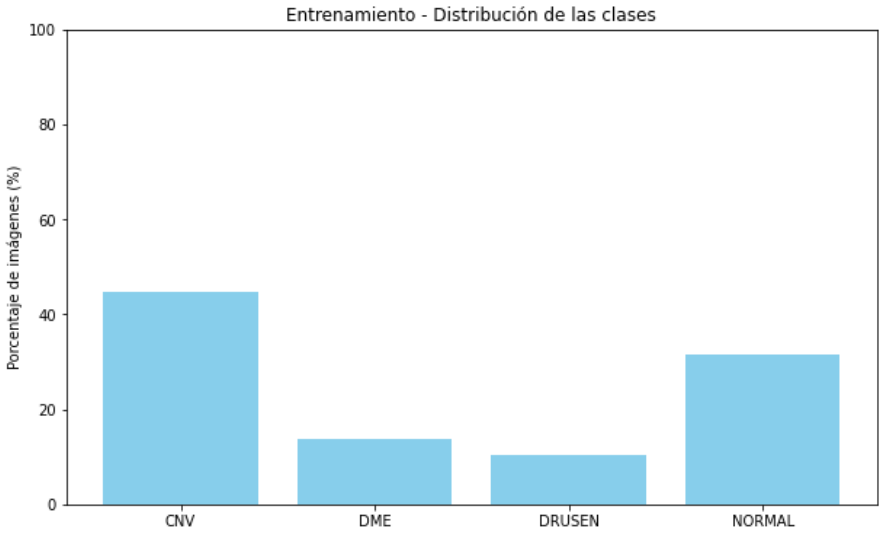


Figura 5.4: Distribución de clases en el conjunto de entrenamiento original.

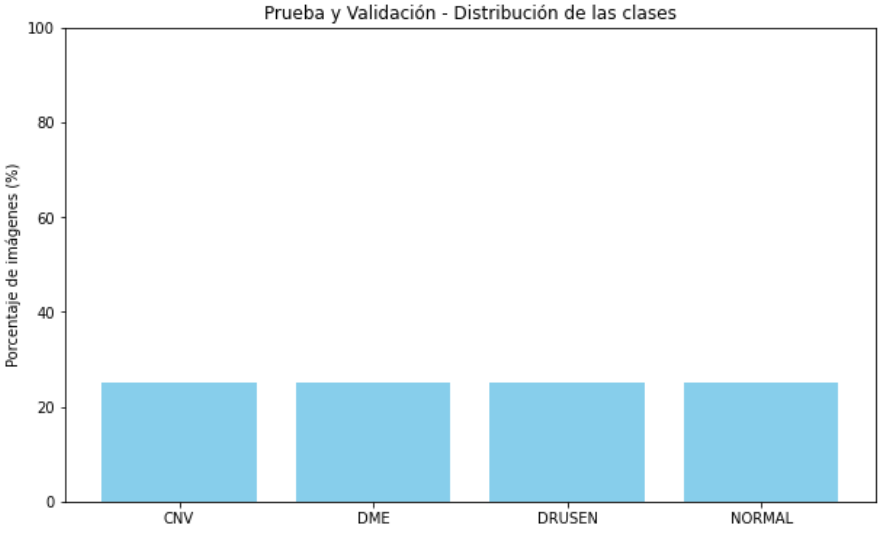


Figura 5.5: Distribución de clases en los conjuntos originales de prueba y validación.

5.2. Tratamiento del conjunto de datos

5.2.1. Redistribución

Tras advertir la descompensación en la distribución de imágenes de los conjuntos de entrenamiento, prueba y validación ilustrada en la Figura 5.2, la primera medida adoptada fue redistribuir las imágenes. Un conjunto de validación de únicamente 32 imágenes puede ser de utilidad para hacer pequeñas pruebas del rendimiento del modelo durante el entrenamiento, mejorándolo en el caso de hacer *Early Stopping* [43]. Sin embargo, incluso en este escenario, considero que el conjunto de validación es excesivamente pequeño y no es representativo, por lo cual las estimaciones de rendimiento obtenidas con él pueden ser poco fiables.

Tras valorar otras alternativas de división de los datos, como aplicar la *Validación Cruzada* o el *Holdout*, finalmente, debido a las limitaciones temporales de desarrollo, se opta por una división estratificada en conjuntos de entrenamiento, prueba y validación [44]. El conjunto de entrenamiento se utiliza para la generación de la hipótesis, mientras que los datos de validación se emplean en un ciclo iterativo, con el propósito de afinar los hiperparámetros y la estructura del modelo. Finalmente, los de prueba se destinan a evaluar el desempeño final del modelo ya entrenado.

La lógica intrínseca a este método, radica en que el conjunto de validación proporciona una estimación razonable de cómo el modelo generaliza frente a nuevos datos, no vistos en la etapa de entrenamiento. Sin embargo, esta estimación no puede considerarse definitiva, ya que su uso para ajustar los hiperparámetros, podría distorsionarla de manera optimista. Por ende, para realizar una evaluación final más objetiva, se reserva un conjunto de prueba, que no se presenta al modelo, hasta la conclusión del proceso de entrenamiento y ajuste.

En este contexto, el proceso seguido fue el siguiente:

1. Unifico todas las imágenes de los distintos conjuntos bajo un único directorio.

2. Creo un objeto `Dataset` mediante el uso de `torchvision.datasets.ImageFolder()`, calculo su tamaño y con él, el tamaño de los subconjuntos.

3. A continuación, mediante el uso de `sklearn.model_selection.train_test_split`, obtengo en dos pasos la lista de índices correspondientes a cada subconjunto, primero dividiendo en entrenamiento y prueba, y finalmente, extrayendo el de validación del conjunto de prueba. Como siempre, se ha tenido la precaución de que los tres conjuntos sean disjuntos.

4. Con el objetivo de que la división de datos sea replicable en posteriores modelos, se hace uso de una semilla.

```
split_random_state = 87
dataset = ImageFolder(path, transform=trans)
train_size = int(0.7 * len(dataset))
test_size = int(0.2 * len(dataset))
val_size = len(dataset) - train_size - test_size

##Obtengo las clases de las imágenes
labels = np.array([label for _, label in dataset.imgs])

#Obtengo los índices de los conjuntos de entrenamiento,
#prueba y validacion estratificados.
train_idx, test_idx, _, _ = train_test_split(
    np.arange(len(labels)), #evita cargar el conjunto de datos.
    stratify=labels,
    test_size=test_size + val_size,
    random_state= split_random_state
)
#Vuelvo a usar train_test_split para dividir test
#De esta manera obtengo test y validación
test_idx, val_idx, _, _ = train_test_split(
    test_idx,
    stratify=labels[train_idx],
    test_size=val_size, #esto son las n muestras de validacion
    random_state= split_random_state
)
```

Figura 5.6: División estratificada del conjunto de datos

Como ilustra la Figura 5.7 distribución de los datos es de un 70 % para el conjunto de entrenamiento, un 20 % para el de prueba y un 10 % para validación, siendo el menor de los tres para optimizar el early stopping del entrenamiento, pero ganando representatividad.

La distribución de clases de los tres conjuntos se corresponde a la distribución global, dado que la división de datos ha sido estratificada. La Figura 5.8 ejemplifica esto con el conjunto de prueba.

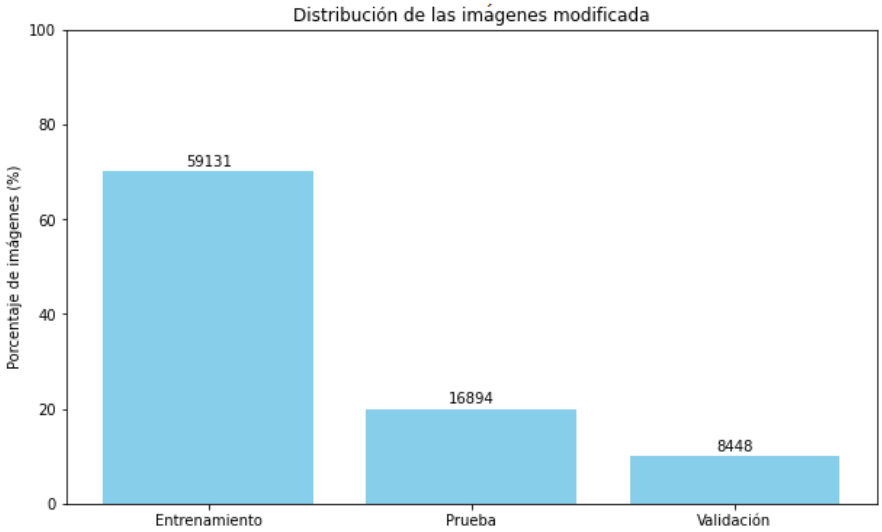


Figura 5.7: Distribución modificada de datos en entrenamiento, prueba y validación.

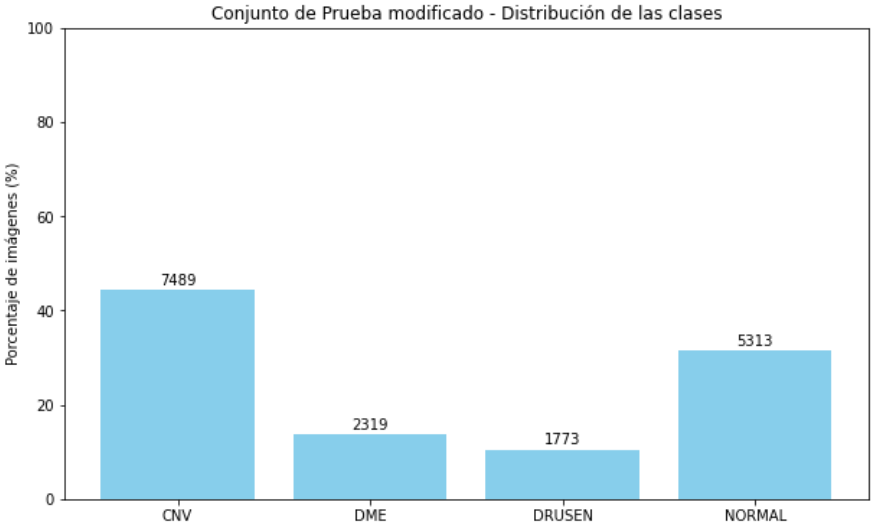


Figura 5.8: Distribución clases en el conjunto de prueba modificado.

5.2.2. Transformaciones

Para mejorar el tratamiento de datos se hacen una serie de transformaciones a las imágenes JPEG originales. Todo esto es llevado a cabo mediante PyTorch [39]:

1. Las imágenes iniciales no tienen una resolución común, por lo que es necesario unificarlas antes de ser procesadas por el modelo. La mayoría de ellas poseen una resolución de (512×512). Lo ideal sería mantener el máximo de resolución posible. Sin embargo, a medida que crece la resolución de las imágenes, aumenta el coste computacional del entrenamiento. Debido a esto, ha sido necesario reducir y unificar las imágenes a una resolución de (256×256), que ha proporcionado buenos resultados, a la par que un mejor rendimiento en el entrenamiento de la red. Para ello, se ha utilizado la transformación `torchvision.transforms.Resize()`

2. El conjunto de datos proporciona imágenes en escala de grises, sin embargo, debido al formato de las imágenes, al utilizar el método `torchvision.datasets.ImageFolder()`, por defecto, se cargan tres canales cromáticos. En este caso, el exceso de canales cromáticos puede suponer una optimización deficiente del algoritmo de entrenamiento, aportando información de poco valor. Por ello, se aplica una transformación que elimina los canales cromáticos innecesarios, unificándolos en un único canal en escala de grises. La transformación de PyTorch utilizada ha sido `torchvision.transforms.Grayscale`.

3. Para mejorar la estabilidad y rendimiento del modelo, tras cada capa de la red, se aplica la técnica de *Batch Normalization* descrita en la sección 3.2.7.

4. Finalmente, se emplea la transformación `torchvision.transforms.toTensor()`, la cual permite operar con las imágenes como tensores.

5.2.3. Carga de datos

Data Loaders

Para que el modelo pueda recibir las imágenes, estas han de ser cargadas del disco a la memoria. Dado que los conjuntos de datos son, en general, muy pesados, no es usual que puedan ser cargados en memoria simultáneamente en su totalidad. Una técnica recurrente es el uso de *Data Loaders*, los cuales cargan los datos en memoria por lotes.

En PyTorch, un `DataLoader` es un objeto iterable que encapsula la lógica necesaria para la carga eficiente de datos en lotes durante el entrenamiento de un modelo. Proporciona un mecanismo altamente flexible y eficiente para automatizar el proceso de carga y preprocesamiento de datos. Un `DataLoader` está diseñado para trabajar sobre un objeto `Dataset`, que se encarga de gestionar un conjunto de datos y proporcionar una interfaz

para su acceso. El `DataLoader`, por otro lado, toma el objeto Dataset y genera lotes de datos. Igualmente, ofrece funcionalidades adicionales como la mezcla de los datos, la carga de datos en paralelo utilizando múltiples procesos y la asignación de una cadena de tranformaciones, que serán aplicadas a cada lote de datos cargado.

Para hacer uso de los `DataLoader`, realizo la división de datos ya comentada, obteniendo tres listas de índices del dataset original, correspondientes a los subconjuntos de entrenamiento, prueba y validación. Posteriormente, estas listas son utilizadas en conjunto con el método `torch.utils.data.Subset()` para generar los subconjuntos que serán pasados a los `DataLoader`.

El código de la Figura 5.9 muestra la creación de los `DataLoader`, donde cabe destacar:

- `batch_size` - Tamaño del lote en cada iteración. Se fija a 200 en el modelo final.
- `num_workers` - Número de procesos empleados en la carga de datos a memoria. Se establece a doce tras el aumento de cores de la máquina virtual.
- `suffle=True` - Permite que los datos no se carguen en orden. En este caso, ante la ausencia de este parámetro, no se mostraría ningún ejemplo de una clase al modelo hasta haber procesado por completo la clase anterior, ya que suelen aparecer seguidas todas las muestras de una misma clase. Este comportamiento es muy perjudicial para el entrenamiento, y este parámetro lo soluciona.

```
train_dataset = torch.utils.data.Subset(dataset, train_idx)
val_dataset = torch.utils.data.Subset(dataset, val_idx)
test_dataset = torch.utils.data.Subset(dataset, test_idx)

train_loader = DataLoader(train_dataset, batch_size=batch,
                          shuffle=True, num_workers = workers)
test_loader = DataLoader(test_dataset, batch_size=batch,
                         shuffle=True, num_workers = workers)
val_loader = DataLoader(val_dataset, batch_size=batch,
                        shuffle=True, num_workers = workers)
```

Figura 5.9: Creación de los `DataLoader`

En el desarrollo de este proyecto se emplea un lote de 200 muestras. Esto no es lo ideal, pues es preferible usar tamaños de batch más pequeños. Sin embargo, debido a la gran cantidad de imágenes, a los recursos computacionales y a las limitaciones temporales del TFG, un lote mayor permite un entrenamiento más rápido. Se comprueba experimentalmente que el lote de 200 conlleva un entrenamiento con un tiempo de ejecución asumible, a la par que demuestra buenas tasas de aciertos.

Carga total

Otra alternativa al uso de data loaders, es la posibilidad cargar todos los datos en memoria de forma simultánea. De esta manera, al evitar la lectura de datos en disco, la velocidad del entrenamiento podría incrementarse de manera considerable. Sin embargo, el conjunto de datos no es el único recurso que consume memoria. Durante el entrenamiento, se almacenan en memoria todos los parámetros del modelo, así como un grafo computacional con las cadenas de operaciones y gradientes de la red. Todo esto, sumado a otros procesos que se están ejecutando simultáneamente, como el navegador, provoca que incluso con el uso de data loaders, si no se es cuidadoso con el tamaño del lote, el proceso de entrenamiento puede llegar a detenerse por problemas de memoria. En consecuencia, la carga total de datos en memoria no resulta viable y, por tanto, se procede con el uso de lotes mediante data loaders.

Capítulo 6

Construcción de la Red Neuronal Convolucional

6.1. Construcción

6.1.1. Diferenciación automática y Autograd

Un aspecto clave de PyTorch es su sistema de diferenciación automática, denominado Autograd, que es esencial para el entrenamiento de modelos de Aprendizaje Profundo.

La diferenciación automática es el proceso de calcular automáticamente los gradientes o derivadas de las funciones, lo cual es crucial para el proceso de optimización empleado en el entrenamiento de las Redes Neuronales, explicado en la Sección 3.1.4.

Autograd en PyTorch funciona mediante el mantenimiento de un grafo de cómputo de todas las operaciones realizadas en los tensores que requieren un gradiente (DAG, Directed Acyclic Graph). Cada tensor tiene un atributo llamado `grad_fn` que referencia a la función que ha creado dicho tensor. Cuando se invoca la función `backward()` en un tensor, PyTorch calcula automáticamente los gradientes para todas las operaciones que llevaron a la generación de ese tensor. Posteriormente, estos gradientes son almacenados en el atributo `grad` de todos los tensores con la propiedad `requires_grad=True` [39].

6.1.2. torch.nn.Module

La clase `torch.nn.Module` es el componente principal de casi todos los modelos de redes neuronales en PyTorch. Proporciona una forma coherente y flexible de construir y manejar estas redes.

Los modelos personalizados de redes neuronales en PyTorch se definen creando una nueva clase que hereda de `torch.nn.Module`, que ha de implementar, como mínimo, los métodos `__init__()` y `forward()`.

init

El método `__init__()` es el utilizado como constructor en Python. En el caso de definir una red neuronal en PyTorch que hereda de **torch.nn.Module**, este método se utiliza para definir las capas y los parámetros del modelo.

Cada capa de la red neuronal (convolución, pooling, normalización, Adaline, etc.) es una instancia de una clase, que también hereda de `torch.nn.Module`.

forward

El método `forward()` define cómo se calcula la salida del modelo a partir de la entrada. Aquí se especifica cómo deben propagarse las entradas a través de las capas definidas en `__init__()`.

6.1.3. torch.nn.Conv2d

La clase `Conv2d`, definida en el módulo `torch.nn`, aplica una convolución 2D sobre una entrada compuesta de varios canales. La salida de la capa con una entrada de tamaño (N, C_{in}, H, W) y salida de tamaño $(N, C_{out}, H_{out}, W_{out})$ puede ser descrita de forma precisa de la siguiente manera:

$$out(N_i, C_{out_j}) = bias(C_{out_j}) + \sum_{k=0}^{C_{in}-1} weight(C_{out}, k) * input(N_i, k) \tag{6.1}$$

Siendo N el tamaño del lote, C el número de canales, H la altura en píxeles de los canales de entrada, W la anchura en píxeles y $*$ la operación de *correlación cruzada* 2D, explicada en la Sección 3.2.2 [39].

6.1.4. torch.nn.Linear

La clase `Linear` aplica una transformación lineal a los datos de entrada de la forma [39]:

$$y = xA^T + b \tag{6.2}$$

En la Red Neuronal Convolucional creada, esta clase es utilizada para la construcción del Perceptrón Multicapa al final de la red.

6.1.5. torch.nn.BatchNorm2d

Las clases `BatchNorm2d` y `BatchNorm1d`, se encargan de la normalización por lote explicada en la Sección 3.2.7. En particular, `BatchNorm2d` realiza una normalización sobre un tensor 4D (un lote de entradas bidimensionales, con una dimensión para los canales).

$$\overline{x}_i = \frac{(x_i - \mu_B)}{\sqrt{\sigma_B^2 + \epsilon}}\gamma + \beta \tag{6.3}$$

Siendo γ y β parámetros ajustables durante el entrenamiento. Adicionalmente, y por defecto, durante el entrenamiento se mantiene una estimación de la media y varianza del conjunto completo, la cual es utilizada en el proceso de normalización durante la evaluación del modelo [39].

6.1.6. torch.nn.Dropout

Como se explicó en la Sección 3.2.7, el dropout es una forma efectiva de regularización, la cual ayuda a prevenir la co-adaptación de las neuronas.

Durante el entrenamiento, mediante la anulación de algunos de los elementos del tensor de entrada, se dejan invariantes los pesos de determinadas neuronas escogidas aleatoriamente con probabilidad p [39]. Sin embargo, este efecto no tiene lugar al poner el modelo en modo evaluación (`model.eval()`), pues resultaría perjudicial, ya que una vez entrenado el modelo ha de comportarse de manera determinista.

6.1.7. Arquitectura de la red

La arquitectura empleada para el modelo se inspira en algunos de los principios de la arquitectura VGG, explicada en la Sección 3.2.6. Como ya se comentó, el modelo recibe como entrada imágenes en escala de grises con una resolución de 256×256. Como en la arquitectura VGG, para la construcción del modelo se emplea un kernel 3×3, un $stride = 1$ y no se usa padding. Entre las distintas capas convolucionales se intercala la función $MaxPooling(2 \times 2)$, explicada en la Sección 3.12.

La primera capa convolucional produce 32 planos, frente a los 64 que se utilizan en general en la arquitectura VGG. Esto se debe a que, en la primera capa, el tamaño de la

entrada es mayor que en las demás, por tanto, el coste computacional del entrenamiento es menor al aplicar menos convoluciones. Esto resulta adecuado dadas las restricciones temporales del TFG. Para finalizar, como medidas de estabilización y regularización, se aplica normalización por lotes tras cada capa y dropout con una probabilidad del 25 % únicamente en la red completamente conectada (MLP). Estas medidas se explican en detalle en la Sección 3.2.7.

Con la finalidad de visualizar mejor la configuración del modelo, la función `summary()` del módulo `torchsummary` ofrece un resumen de las capas y parámetros de la red, como se puede apreciar en la Figura 6.1.

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1         [-1, 32, 254, 254]             320
       BatchNorm2d-2         [-1, 32, 254, 254]              64
            Conv2d-3         [-1, 64, 125, 125]          18,496
       BatchNorm2d-4         [-1, 64, 125, 125]             128
            Conv2d-5          [-1, 128, 60, 60]          73,856
       BatchNorm2d-6          [-1, 128, 60, 60]             256
            Conv2d-7          [-1, 256, 28, 28]         295,168
       BatchNorm2d-8          [-1, 256, 28, 28]             512
            Conv2d-9          [-1, 256, 12, 12]         590,080
      BatchNorm2d-10          [-1, 256, 12, 12]             512
           Conv2d-11          [-1, 512, 10, 10]       1,180,160
      BatchNorm2d-12          [-1, 512, 10, 10]           1,024
           Linear-13                 [-1, 256]      13,107,456
      BatchNorm1d-14                 [-1, 256]             512
          Dropout-15                 [-1, 256]               0
           Linear-16                 [-1, 256]          65,792
      BatchNorm1d-17                 [-1, 256]             512
           Linear-18                   [-1, 4]           1,028
================================================================
Total params: 15,335,876
Trainable params: 15,335,876
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.25
Forward/backward pass size (MB): 58.21
Params size (MB): 58.50
Estimated Total Size (MB): 116.96
----------------------------------------------------------------
```

Figura 6.1: Resumen paramétrico del modelo

6.1.8. Implementación de la red

En la Figura 6.2, se puede ver la definición de la clase `CNN(nn.Module)`. En el constructor se han definido las capas con parámetros entrenables. Esto incluye las capas convolucionales, las capas lineales, la capas de normalización por lote y, por claridad, el dropout. La Figura 6.3 ilustra la función `forward()`, la cual describe la forma en que las capas se conectan, aplicando la función $MaxPooling(2\times2)$, mediante `torch.nn.functional.max_pool2d()`, y ReLU como función de activación, usando la clase `torch.nn.functional.relu()`. Ambas funciones han sido explicadas en las secciones 3.2.5 y 3.1.2 respectivamente.

La última capa convolucional del modelo se conecta con la de entrada del Perceptrón Multicapa. Para este fin, se hace uso de la función *torch.flatten(x, 1)* para aplanar el tensor x, resultante de la última capa convolucional. Adicionalmente, es necesario calcular el tamaño de la entrada aplanada, que ha de coincidir con el tamaño de la capa de entrada del Perceptrón Multicapa. Para ello, se hace uso de la función `flatten_size_after_conv()` representada en la Figura 6.4, la cual aplica la expresión indicada en la Sección 3.2.5.

```
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        # 1 input image channel(escala de grises),
        # 64 output channels, 3x3 square convolution kernel
        self.conv1 = nn.Conv2d(1, 32, 3)
        self.bn1 = nn.BatchNorm2d(32)
        self.conv2 = nn.Conv2d(32, 64, 3)
        self.bn2 = nn.BatchNorm2d(64)
        self.conv3 = nn.Conv2d(64, 128, 3)
        self.bn3 = nn.BatchNorm2d(128)
        self.conv4 = nn.Conv2d(128, 256, 3)
        self.bn4 = nn.BatchNorm2d(256)
        self.conv5 = nn.Conv2d(256, 256, 3)
        self.bn5 = nn.BatchNorm2d(256)
        self.conv6 = nn.Conv2d(256, 512, 3)
        self.bn6 = nn.BatchNorm2d(512)

        # calculo el tamaño de la imagen aplanada
        flat_img_size = self.flatten_size_after_conv(
            height=img_height,
            width=img_width,
            n_conv=6, kernel_dim=3,
            pooling_kernel=2,
            n_pooling=4)

        self.fc1 = nn.Linear(512 * flat_img_size, 256)
        self.bn7 = nn.BatchNorm1d(256)
        self.dropout1 = nn.Dropout(0.25)
        self.fc2 = nn.Linear(256, 256)
        self.bn8 = nn.BatchNorm1d(256)
        self.fc3 = nn.Linear(256, 4)
```

Figura 6.2: Constructor de la clase `CNN(nn.Module)`

```
    def forward(self, x):
        # Max pooling over a (2, 2) window
        x = F.max_pool2d(F.relu(self.bn1(self.conv1(x))), (2, 2))
        x = F.max_pool2d(F.relu(self.bn2(self.conv2(x))), 2)
        x = F.max_pool2d(F.relu(self.bn3(self.conv3(x))), 2)
        x = F.max_pool2d(F.relu(self.bn4(self.conv4(x))), 2)
        x = F.relu(self.bn5(self.conv5(x)))
        x = F.relu(self.bn6(self.conv6(x)))
        # flatten all dimensions except the batch dimension
        x = torch.flatten(x, 1)
        x = F.relu(self.bn7(self.fc1(x)))
        x = self.dropout1(x)
        x = F.relu(self.bn8(self.fc2(x)))
        x = self.fc3(x)
        return x
```

Figura 6.3: Función `forward()` de la clase `CNN(nn.Module)`

```
    def flatten_size_after_conv(self, height, width, kernel_dim, n_conv=1,
                                padding=0, stride=1, pooling_kernel=1,
                                n_pooling=0):

        #returns the size of a flatted image
        h_size = height
        w_size = width
        for i in range(n_conv):
             #Utilizo floor para redondear, y que no haya decimales
            h_size = math.floor((h_size + 2 * padding - kernel_dim)
            / stride) + 1
            w_size = math.floor((w_size + 2 * padding - kernel_dim)
            / stride) + 1
            if(n_pooling != 0):
                h_size = math.floor(h_size/pooling_kernel)
                w_size = math.floor(w_size/pooling_kernel)
                n_pooling -= 1
        return int(h_size*w_size)
```

Figura 6.4: Función de la clase `CNN(nn.Module)` para calcular el tamaño de la imagen aplanada tras el paso hacia adelante.

6.2. Entrenamiento

6.2.1. torch.nn.CrossEntropyLoss

La clase utilizada como función de pérdida es `CrossEntropyLoss`, que es ampliamente utilizada en problemas de multiclasificación. En esencia, combina la función `LogSoftmax` y la pérdida de entropía cruzada en una sola clase, lo que la hace más conveniente y eficiente [39].

Generalmente, en problemas de clasificación, la última capa del modelo suele ser una función Softmax, aplicada en PyTorch mediante la clase `torch.nn.Softmax`, ya que gracias a esta función, el vector resultante es el de las probabilidades de pertenencia a cada clase [18]. Es importante mencionar que en el modelo creado no emplea `Softmax` como última capa de la red, ya que `CrossEntropyLoss` ya incluye una operación de este tipo. Aplicar otra capa `Softmax` puede llevar a una distribución de probabilidad incorrecta y reducir la eficacia de la red. Por otro lado, en modo evaluación resulta innecesaria, pues para calcular la tasa de aciertos solo se precisa la función `torch.max`.

6.2.2. torch.optim.Adam

El optimizador `Adam` es una extensión del descenso estocástico del gradiente explicado en la Sección 3.4. Se utiliza a menudo en el entrenamiento de redes neuronales profundas. En la práctica, Adam ha demostrado buenos resultados en la velocidad de convergencia al compararse con otros métodos basados en el descenso estocástico del gradiente [45, 39].

6.2.3. torch.optim.lr_scheduler.ReduceLROnPlateau

Para afinar el ajuste del modelo se considera interesante la aplicación de un *Learning Rate Scheduler*. En este caso, la clase `ReduceLROnPlateau` del módulo `torch.optim.lr_scheduler`, que se encarga de reducir la tasa de aprendizaje cuando la métrica de rendimiento ha dejado de mejorar. A menudo, los modelos se benefician de la reducción de la tasa de aprendizaje en un factor de 2 a 10, una vez que el aprendizaje se estanca [39]. El uso de este planificador implica que, tras un número de épocas denominado `patience` sin mejora, la tasa de aprendizaje se reduce.

Esta clase ha sido probada para varios grados de `patience`, sin embargo, no se ha conseguido detonar el planificador. Esto se debe a que se ha empleado *Early Stopping* [43] durante el entrenamiento del modelo, y este siempre se ha activado antes. Esto se debe a que la métrica empleada por el early stopping es la precisión en el conjunto de validación, mientras que el planificador utiliza el valor de pérdida.

6.2.4. Bucle de entrenamiento

Previo al proceso de entrenamiento, se instancia el modelo y se definen las variables necesarias para este; entre ellas están la función de perdida (`CrossEntropyLoss`), el optimizador (`Adam`), el learning rate scheduler, variables para el early stopping, así como unas listas para almacenar datos relativos a la precisión del modelo. Esto será de utilidad para la generación de gráficos, que ayuden a la visualización del entrenamiento. Esto se muestra en la Figura 6.5, donde vale la pena destacar:

- `device`: En esta variable se almacena el dispositivo disponible, dando preferencia a la GPU, pues el entrenamiento es más eficiente. La implementación tiene en cuenta la posibilidad de utilizar una GPU, sin embargo, debido a que, como se dijo en el Capítulo 4 no se dispone de una, la funcionalidad relativa a la ejecución en GPU no ha podido ser testeada.
- `limit`: Se usa como límite para el número de épocas sin mejora en el Early Stopping. Si durante el entrenamiento, la precisión en el conjunto de validación no aumenta durante un número de épocas consecutivas igual a `limit`, entonces se detiene el entrenamiento. En otras palabras, se utiliza como una condición de parada temprana para prevenir el sobreajuste. En este caso, el límite se ha establecido en 8, lo que significa que si no hay mejora en la precisión de validación durante 8 épocas consecutivas, el entrenamiento se detendrá.
- `count`: Esta variable lleva un conteo de las iteraciones en el entrenamiento, será útil para graficar la perdida frente a las iteraciones durante el entrenamiento.

```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

max_epochs = 30
limit = 8
count = 0
patience = 3
# Listas para visualizar la perdida y la precisión
loss_list = []
epoch_list = []
accuracy_list = []
iteration_list = []

# Listas para ver la precision por clase
predictions_list = []
labels_list = []

model = CNN()
model.to(device) #si hay GPU pasamos el modelo a la GPU

loss_criterion = nn.CrossEntropyLoss()

#Se inicializa el optimizador con los parámetros del modelo
#y el learning rate
learning_rate = 0.001
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(
    optimizer,
    patience=patience,
    verbose=True)
```

Figura 6.5: Instanciación e inicialización de clases y variables necesarias para el entrenamiento.

La función `train_one_epoch`, ilustrada en la Figura 6.6, está diseñada para ejecutar una época de entrenamiento, donde se puede destacar que:

- Se inicia poniendo el modelo en modo de entrenamiento con `model.train()`; de esta manera, el dropout será efectivo y la normalización por lote no se ejecutará en su modalidad de evaluación.
- Para cada lote de imágenes y etiquetas se realiza lo siguiente:
  1. Son enviadas al dispositivo de cálculo `device`.
  2. El modelo realiza una predicción en las imágenes, así como el cálculo de la pérdida entre las etiquetas predichas y las esperadas mediante `loss = loss_criterion(outputs, labels)`.
  3. Se borran los gradientes acumulados en el optimizador (`optimizer.zero_grad()`).
  4. Al ejecutar `loss.backward()`, como se explicó en la Sección 6.1.1, se realiza la retropropagación del error para calcular los gradientes.
  5. Se actualizan los pesos de la red con el paso del optimizador.
  6. Se libera la memoria de las imágenes y etiquetas que ya se han procesado.
  7. Se registra la pérdida del lote en una lista de pérdidas de la época.
- Finalmente, la función devuelve la lista de pérdidas de la época.

Adicionalmente, se definen dos funciones, que se pueden apreciar en las figuras 6.7 y 6.8. Se encargan de procesar el conjunto de validación, calculando el número de predicciones correctas y la precisión en porcentaje respectivamente.

Finalmente, todas estas funciones son ejecutadas en conjunto para entrenar el modelo, como se muestra en la Figura 6.9. Para ello, hay que tener en cuenta que:

- Se emplea un número máximo de épocas, para que el entrenamiento no se eternice. En general, este máximo no es alcanzado, pues el Early Stopping se detona previamente.
- Con el objetivo de mostrar una barra de progreso, que ofrezca información del estado del entrenamiento, se emplea la clase `tqdm`. La información ofrecida es la siguiente:
  1. Tiempo de entrenamiento transcurrido.
  2. Estimación del tiempo máximo de entrenamiento, considerando que todas las épocas indicadas en `max_epochs` son ejecutadas.
  3. Épocas ejecutadas frente a las máximas épocas posibles.
  4. Épocas consecutivas sin mejora en validación frente al límite definido de épocas sin mejora `limit`.

- La precisión obtenida, al procesar el conjunto de validación, es empleada como una métrica de la capacidad de generalización del modelo. Por tanto, se mantiene un registro de la mejor puntuación obtenida hasta el momento, y una vez superado ese valor, se utiliza el método `torch.save()` para hacer una copia persistente del estado del modelo. Este registro mantendrá siempre el estado del modelo que ha demostrado una mayor capacidad de generalización en validación, y será la configuración escogida para su evaluación posterior. Puede verse como una manera de anticiparse al sobreajuste.
- Se calcula el tiempo total de ejecución y se mantienen registros de la precisión y perdida, con el objetivo de poder graficar la evolución del modelo a lo largo del entrenamiento.

```python
def train_one_epoch(model, train_loader, optimizer,
                    loss_criterion, device):
    epoch_losses = []
    # Activo el modo entrenamiento
    # (Capas dropout y batch normalization)
    model.train()
    for images, labels in train_loader:
        images, labels = images.to(device), labels.to(device)

        outputs = model(images)
        loss = loss_criterion(outputs, labels)  # Vector de error

        optimizer.zero_grad()

        # Backpropagation para propagar el error
        # actualizando los gradientes
        loss.backward()
        optimizer.step()  # Actualización de los pesos

        del images, labels  # Liberación de memoria

        epoch_losses.append(loss.item())

    return epoch_losses
```

Figura 6.6: Función que se encarga del entrenamiento de una época.

```python
def validate(model, loader, device):
    model.eval()    # Cambio el comportamiento del DropOut
                    # y el batchNormalization
    correct = 0
    total = 0
    for images, labels in loader:
        images, labels = images.to(device), labels.to(device)
        outputs = model(images)
        predictions = torch.max(outputs, 1)[1].to(device)
        correct += (predictions == labels).sum()
        total += len(labels)
        del images, labels

    return correct, total
```

Figura 6.7: Función que calcula el número de predicciones correctas para un modelo en un conjunto dado.

```python
def calculate_accuracy(correct, total):
    return correct * 100 / total
```

Figura 6.8: Función que calcula el porcentaje de precisión dado el número de predicciones correctas y el total de predicciones dadas.

```
start = time.time()

for epoch in range(max_epochs):
    loop = tqdm(val_loader, leave=False)
    epoch_losses = train_one_epoch(model, loop, optimizer,
                                   loss_criterion, device)

    mean_loss = sum(epoch_losses) / len(epoch_losses)
    scheduler.step(mean_loss)
    correct, total = validate(model, val_loader, device)
    acc_val = calculate_accuracy(correct, total)

    loop.set_description(
        (
            f"Epocas [{epoch}/{max_epochs}]"
            f" | Epocas sin mejora [{epochs_sin_mejora}/{limit}]"
        )
    )

    loop.set_postfix(loss=mean_loss, acc=best_acc)

    if acc_val > best_acc:
        best_acc = acc_val
        epochs_sin_mejora = 0
        torch.save(model, 'model.pth')
        print("### - Guardando modelo - "
              f"Validation accuracy = {acc_val}%")

    else:
        epochs_sin_mejora += 1

    if epochs_sin_mejora > limit:
        print(f"Deteniendo entrenamiento tras "
              f"{limit} épocas sin mejora")
        break  # salimos de este bucle

    epoch_list.append(epoch)
    accuracy_list.append(acc_val)

end = time.time()
total_time = end - start
```

Figura 6.9: Bucle de entrenamiento.

6.3. Implementación de Grad-CAM

Como se comentó en la Sección 3.2.8, Gradient-weighted Class Activation Mapping es un método de gran utilidad para indagar en la interpretabilidad de un modelo. Por ello, se considera de gran interés la aplicación de este algoritmo en la red desarrollada. Para lograr este fin, surgen diferentes alternativas: la búsqueda de un módulo ya implementado que permita su aplicación al modelo o, por otra parte, la implementación manual del algoritmo. En este TFG finalmente se realizan ambas.

6.3.1. M3d-CAM

Medcam es una librería de Python que facilita la interpretación y visualización de modelos de Redes Neuronales Convolucionales, la cual está principalmente enfocada a imágenes médicas, permitiendo no solo la visualización de mapas de calor en imágenes bidimensionales, sino que también puede aplicarse a Redes Convolucionales 3D [46].

El código Python de la Figura 6.10 muestra cómo se puede utilizar Medcam para visualizar un mapa de atención en una imagen procesada.

El código comienza poniendo el modelo en modo de evaluación con `model.eval()`. Posteriormente, Medcam se "inyecta" en el modelo con `medcam.inject()`. Esta función modifica el modelo para que produzca mapas de atención en el directorio indicado, además de su salida normal. Asimismo, es necesario indicar algunos parámetros:

- El modelo sobre el que se inyectará `medcam`.
- El directorio donde se guardarán los mapas de atención.
- Que el algoritmo sea Grad-Cam (`backend='gcam'`)
- La capa sobre la que se realizará la ponderación de los mapas de características, en este caso, la última capa convolucional de la red (`layer='conv6'`).
- Respecto a qué etiqueta aplicar en el algoritmo, se usará la ganadora en la predicción (`label='best'`).
- La forma de la entrada, que incluye el tamaño del lote, número de canales y resolución de la imagen (`data_shape=(1,1,256,256)`)

A continuación, se carga una imagen y se le añade una dimensión de batch utilizando el método `image.unsqueeze(0)`. La imagen es procesada mediante `model.forward(image)`, lo que produce tanto la salida normal del modelo, como un mapa de atención que será guardado en el directorio indicado.

Para la visualización del mapa de atención sobre la imagen procesada, este se carga desde el disco haciendo uso de `PIL` [47] con el método `Image.open(map_path)`. Luego, se redimensiona al mismo tamaño que la imagen original con `transforms.Resize()`. Finalmente, se utiliza `plt.imshow()` para visualizar tanto la imagen original, como el mapa de atención superpuesto. Esto proporciona una visualización de las áreas de la imagen, que el modelo considera más importantes para hacer su predicción.

Un detalle adicional es el uso de `img_np = imagen.permute(1, 2, 0).numpy()`, porque la forma de los tensores de PyTorch es, en general, (Canales, Altura, Anchura). Por otro lado, `imshow` espera un vector numpy de la forma (Altura, Anchura, Canales); de esta manera, se utiliza `permute` para reorganizar las dimensiones de la imagen.

En la Figura 6.11, se puede observar el resultado visual de la aplicación de esta técnica a una imagen que muestra síntomas de Neovascularización Coroidea.

```
model.eval()
model = medcam.inject(
    model, output_dir='heat_maps',
    backend='gcam', layer='conv6',
    label='best',
    data_shape=(1,1,256,256),
    save_maps=True)

# Obtengo una imagen cualquiera del conjunto de test
# En este caso la novena imagen
image, label = test_dataset.__getitem__(9)

image = image.to(device)
output = model.forward(image)
map_path = 'heat_maps/conv6/attention_map_0_0_0.png'

heatmap = Image.open(map_path)
# Redimensionar el mapa de calor a 256x256
resize_transform = transforms.Resize(size=(256, 256))
heatmap_resized = resize_transform(heatmap)
# Mostrar la imagen y el mapa de calor
img_np = image.permute(1, 2, 0).numpy()
plt.imshow(img_np, cmap='gray')
plt.imshow(heatmap_resized, alpha=0.2, cmap='jet')
plt.show()
```

Figura 6.10: Aplicación de `medcam` sobre una imagen del conjunto de prueba.

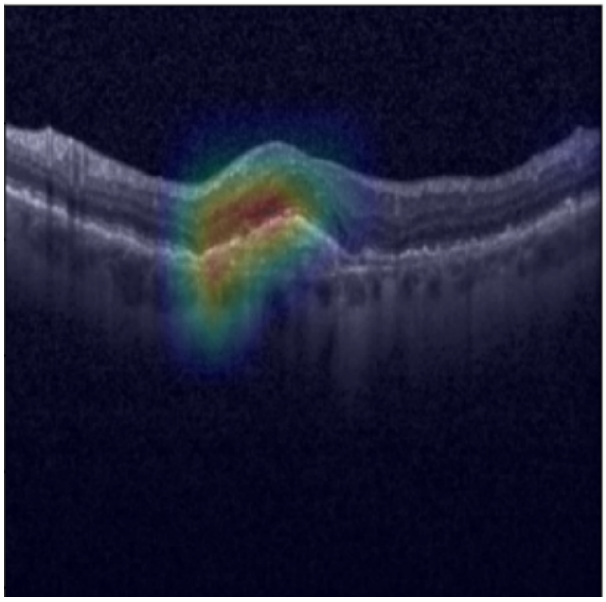


Figura 6.11: Visualización del mapa de atención generado por `medcam` en una imagen con presencia de Neovascularización Coroidea.

6.3.2. Implementación manual

Para lograr una mayor flexibilidad y entendimiento del algoritmo Grad-CAM, dado que se trata de un proyecto académico, se opta por realizar una implementación manual del algoritmo, consultando para este fin el libro *Modern Computer Vision with PyTorch* [29].

Para calcular los mapas de atención superpuestos a las imágenes, se crean dos funciones: `im2gradCam()` y `upsampleHeatmap()`.

La primera, expuesta en la Figura 6.13, se utiliza para calcular el mapa de calor que corresponde a una imagen y un modelo dado. Esta función aplica en detalle el algoritmo descrito en la Sección 3.2.8 y recibe los siguientes argumentos:

- `tensor` - Imagen en forma de tensor para la que obtener el mapa de atención.
- `model` - Modelo que procesará la imagen.
- `im2fmap` - Se trata de una instancia de `nn.Sequential`, la cual define la secuencia de capas del modelo hasta la última capa convolucional. Este argumento se emplea para obtener los mapas de características de la última capa convolucional.
- `device` - Dispositivo sobre el que se han de realizar los cálculos. Por defecto: `cpu`

Esta función retorna un tensor que representa el mapa de atención y la predicción del modelo.

Por otro lado, la función `upsampleHeatmap()`, cuya implementación se muestra en la Figura 6.12, recibe el mapa de atención generado por `im2gradCam`, la imagen original y las dimensiones a la resolución a la que se desea la imagen final. Con ello, realiza un escalado del mapa de calor y de la imagen, los cuales superpone para retornar un mapa de calor fácilmente interpretable.

```
def upsampleHeatmap(hmap, img, h, w):
    m,M = hmap.min(), hmap.max()
    hmap = 255 * ((hmap-m)/(M-m)) #Normalizamos el heatmap
    hmap = np.uint8(hmap)
    hmap = cv2.resize(hmap, (w,h))
    hmap = cv2.applyColorMap(255-hmap, cv2.COLORMAP_JET)
    hmap = np.uint8(hmap)
    # Convertimos la imagen a un array de NumPy
    # y ajustamos su forma
    trans = transforms.Resize((h, w))
    img = trans(img)
    img_np = img.cpu().numpy()
    img_np = np.transpose(img_np, (1, 2, 0)) * 255
    img_np = np.uint8(img_np)

    # Combinamos las imágenes del heatmap
    # y la imagen original
    hmap = np.uint8(hmap * 0.3 + img_np * 0.7)
    return hmap
```

Figura 6.12: Función para obtener el mapa de atención superpuesto a la imagen original.

Mediante el uso de estas funciones, se pueden extraer los mapas de atención para cada predicción realizada por la red. Un ejemplo de estos mapas puede verse en la Figura 6.14 donde cada fila representa una clase verdadera, mostrando cinco imágenes de entrada y sus respectivos mapa de atención. Cada par de imágenes, posee una etiqueta sobre el, representando la predicción ofrecida por el modelo.

```python
def im2gradCam(tensor, model, im2fmap, device=torch.device("cpu")):
    model.eval()  # Modelo en modo evaluación
    # Asegurar que el modelo esté en el dispositivo correcto
    model.to(device)
    # Asegurar que el tensor esté en el dispositivo correcto
    tensor = tensor.to(device)
    outs = model(tensor)
    activations = im2fmap(tensor)  # Obtengo los mapas de características
    pred = outs.max(1)[-1]  # Obtengo la clase predicha
    model.zero_grad()

    # Computo los gradientes respecto a la clase predicha
    outs[0, pred].backward(
        retain_graph=True
    )

    # Obtengo la última capa convolucional del modelo
    last_conv2d = None
    for name, module in reversed(model._modules.items()):
        if isinstance(module, torch.nn.Conv2d):
            last_conv2d = module
            break

    # Calculo la media de los gradientes
    # respecto a las dimensiones:
    # (entrada, altura kernel, anchura kernel)
    # dejando tantos pesos como feature maps en la última capa
    weights = last_conv2d.weight.grad.data.mean((1, 2, 3))

    # Pondero cada feature map
    for i in range(activations.shape[1]):
        activations[:, i, :, :] *= weights[i]

    # Calculo el heatmap como la media ponderada
    # de los mapas de características
    heatmap = (
        torch.mean(activations, dim=1)[0].cpu().detach()
    )
    return heatmap, pred.item()
```

Figura 6.13: Función para obtener el mapa de atención de una imagen y un modelo dados.

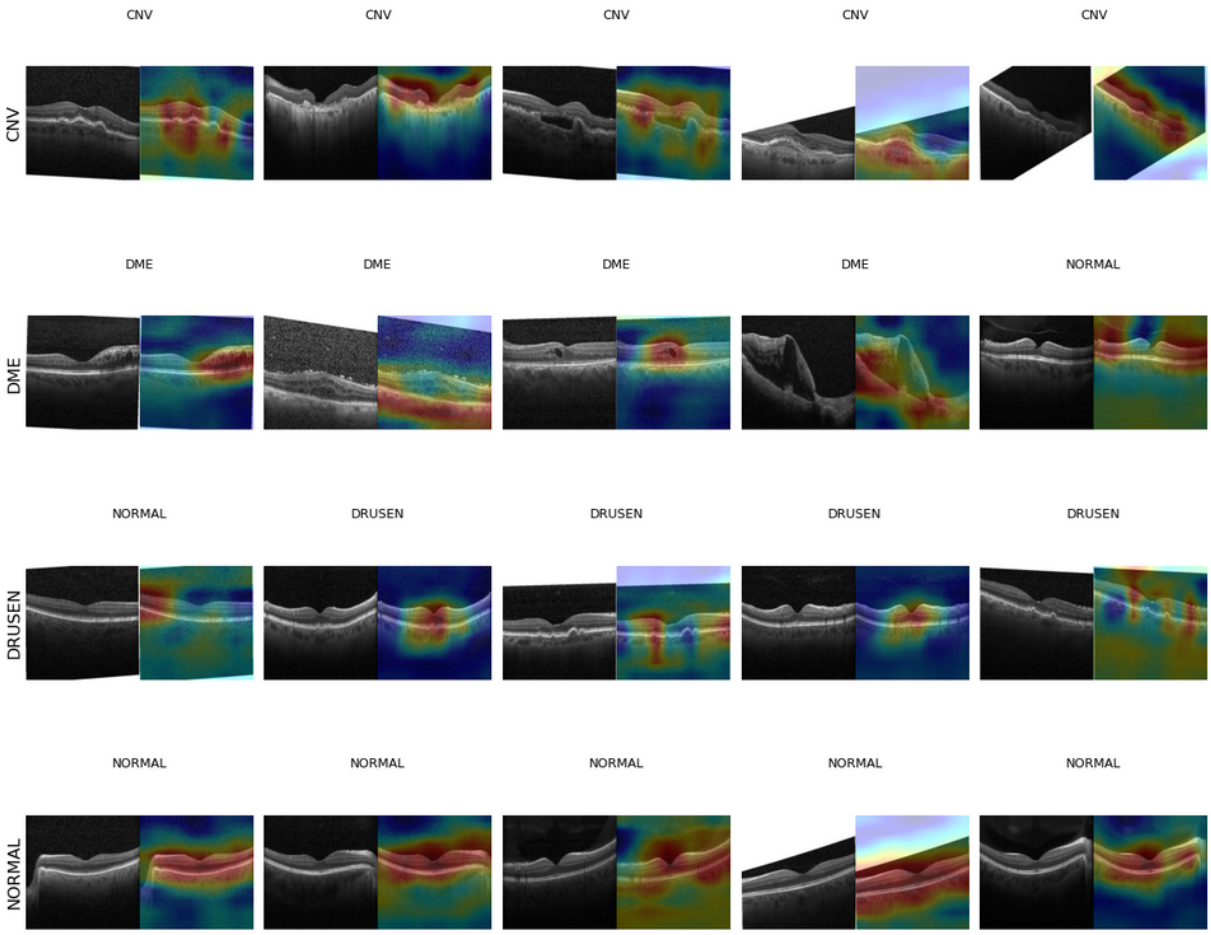


Figura 6.14: Ejemplos de mapas de atención en la implementación manual.

Capítulo 7

Resultados

A lo largo del desarrollo del modelo, se han probado distintas variaciones de los hiperparámetros, dando lugar a varios modelos. Como ya se comentó, durante el proceso de desarrollo se hace uso de la técnica de Parada Anticipada o *Early Stopping* [43] sobre el conjunto de validación. Igualmente, con él se ha ido calculando la precisión a fin de evaluar la capacidad de generalización del modelo. En función de estos resultados, los hiperparámetros del modelo se han ido ajustando tratando de maximizar las prestaciones del sistema. Tras el proceso de entrenamiento, el modelo elegido para su evaluación ha sido aquel que ha obtenido una mayor tasa de acierto en el conjunto de validación.

Este capítulo presenta los resultados obtenidos tras la evaluación e interpretación del modelo seleccionado. Esto incluye su evolución a lo largo del entrenamiento, la precisión final obtenida y un análisis cuyo objetivo es la interpretación de los métodos de diagnóstico empleados por el modelo.

7.1. Evolución del modelo

Como ya se presentó previamente en el Capítulo 6, para garantizar el almacenamiento de la configuración paramétrica del modelo que obtiene una mejor precisión, al final de cada época se realiza una evaluación sobre el conjunto de validación. Los parámetros del modelo serán almacenados si demuestran un mejor rendimiento que el mejor obtenido hasta el momento durante el entrenamiento. Este comportamiento se muestra en la Figura 7.1, en la que se pueden observar los sucesivos aumentos en la precisión obtenida por el modelo. Finalmente, el entrenamiento se detiene debido a la falta de mejora en un número de épocas superior al límite fijado, en este caso ocho.

La Figura 7.1 muestra únicamente los picos más altos en las precisiones obtenidas durante el entrenamiento. Para una mejor visualización de la evaluación del modelo, las

```
### - Guardando modelo - Validation accuracy = 86.13873291015625%

### - Guardando modelo - Validation accuracy = 92.8977279663086%

### - Guardando modelo - Validation accuracy = 93.4422378540039%

### - Guardando modelo - Validation accuracy = 95.18228912353516%

### - Guardando modelo - Validation accuracy = 95.34801483154297%

### - Guardando modelo - Validation accuracy = 96.09375%

### - Guardando modelo - Validation accuracy = 96.36600494384766%

Deteniendo entrenamiento tras 8 épocas sin mejora
```

Figura 7.1: Ejemplo de Parada Anticipada en el entrenamiento del modelo seleccionado.

Figuras 7.2 y 7.3 muestran la evolución de la precisión y de la pérdida a lo largo del entrenamiento, respectivamente. Se puede observar que la granularidad de la Figura 7.3 es mucho más fina, esto se debe a que la pérdida se calcula en cada iteración, mientras que la precisión solo es calculada tras la finalización de cada época, evitando así ralentizar en exceso el entrenamiento.

Figura 7.2: Evolución de la precisión sobre el conjunto de validación a lo largo del entrenamiento.

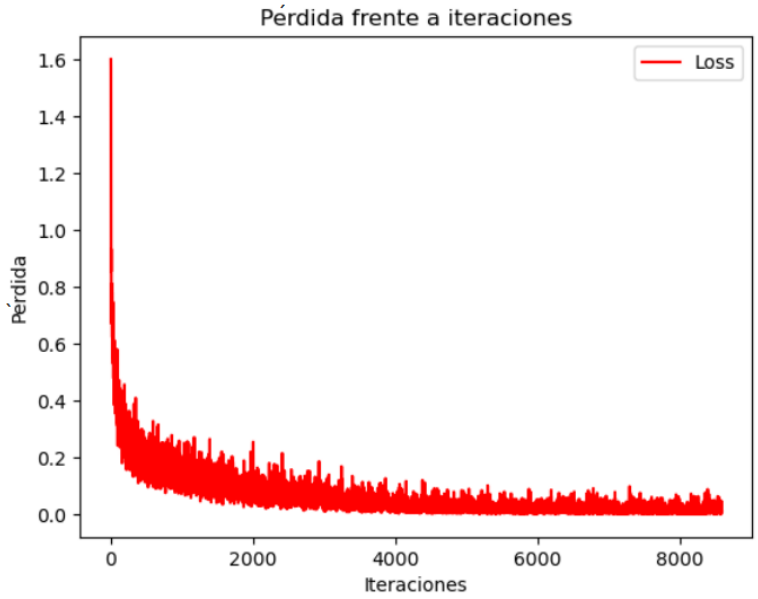


Figura 7.3: Evolución de la pérdida a lo largo del entrenamiento.

7.2. Evaluación

Una vez seleccionado el modelo, se hace uso por primera vez del conjunto de prueba, esto evita que el proceso iterativo de ajuste de hiperparámetros tenga efectos nocivos en la calidad de la estimación del rendimiento, pues gracias al empleo del conjunto de validación para este fin, la prueba resulta completamente independiente del proceso de entrenamiento.

En la Tabla 7.1 se pueden observar los porcentajes de precisión obtenidos para cada una de las particiones del conjunto de datos.

Partición	**Precisión obtenida**
Entrenamiento	98, 841 %
Validación	96, 366 %
Test	96, 531 %

Cuadro 7.1: Porcentaje de precisión para los conjuntos de datos

Asimismo, en la Figura 7.4 se puede ver la matriz de confusión del modelo en el conjunto de prueba. Esta resulta de gran utilidad, pues permite visualizar en su diagonal el número de muestras etiquetadas correctamente para cada clase y, además, fuera de la diagonal se observan las etiquetas asignadas erróneamente. Sin embargo, debido a la descompensación en el volumen de muestras para cada clase, para poder realizar una

comparación más directa, una matriz de confusión normalizada puede resultar de mayor utilidad.

En la Figura 7.5 se muestra una matriz de confusión en la que cada celda de una fila, ha sido dividida por el número total de muestras en dicha fila. Este tipo de matriz, permite observar en su diagonal la tasa de acierto del modelo para cada clase. Es en esta diagonal, donde mejor se puede apreciar una alta tasa de acierto obtenida para cada una de las clases, siendo DRUSEN la que ofrece un peor rendimiento, con una tasa de acierto de $0,87$.

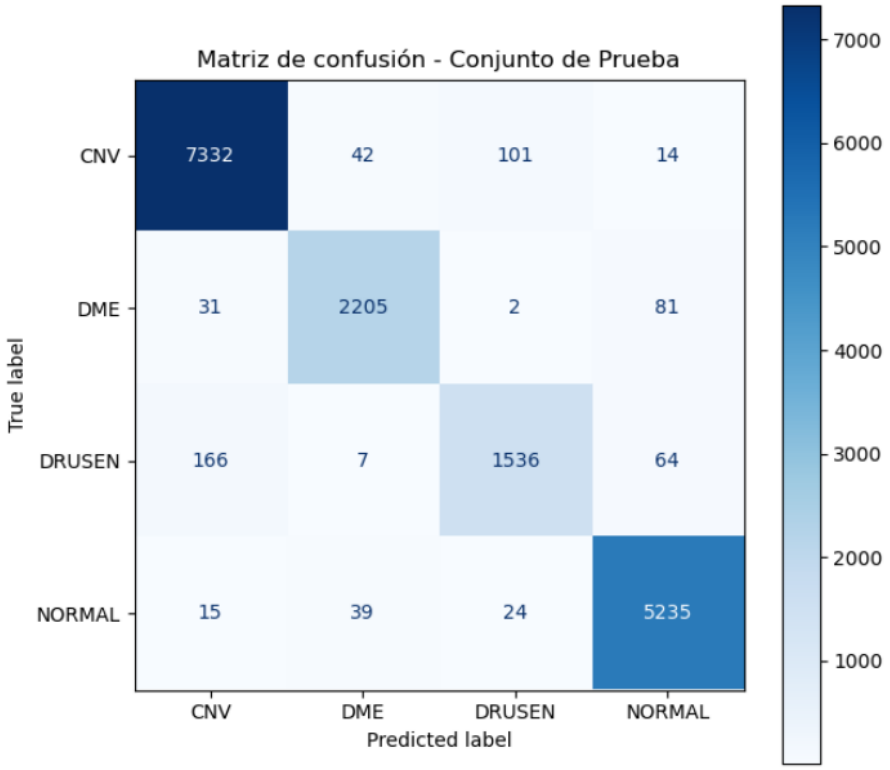


Figura 7.4: Matriz de confusión en el conjunto de prueba.

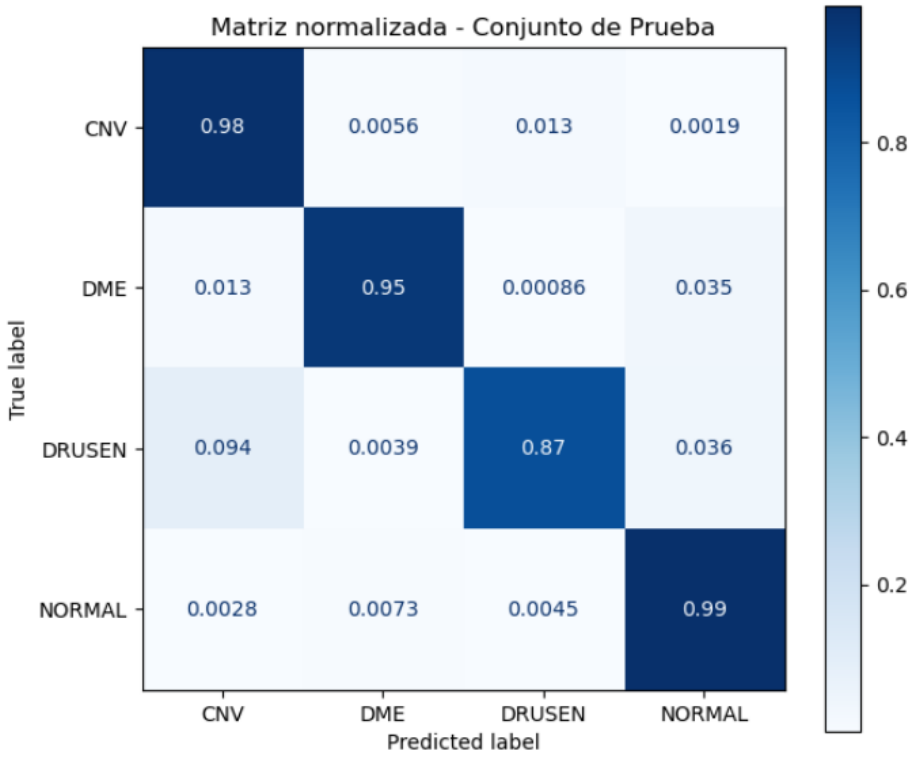


Figura 7.5: Matriz de confusión normalizada en el conjunto de prueba.

7.3. Interpretación

Para tratar de explicar el modo en el que los diagnósticos son producidos por el modelo, se hace uso de los mapas de atención computados por el algoritmo Grad-CAM desarrollado, el cual se explica en detalle en la Sección 3.2.8. Gracias a estos mapas, para cada predicción de diagnóstico obtenida, se pueden observar de forma visual las zonas de la imagen de entrada que han resultado de mayor relevancia para el modelo.

En la Figura 7.6 se pueden observar tres ejemplos de imágenes correctamente etiquetadas para cada clase, junto a sus respectivos mapas de atención. En ellos, las zonas con colores más cálidos, como el rojo, son aquellas con mayor importancia para el diagnóstico, mientras que las zonas con colores fríos, como el azul, resultan poco relevantes.

Debido a la falta de un conocimiento experto en la interpretación de imágenes OCT, no es posible analizar ni comparar los focos de atención producidos por el modelo respecto a las observaciones realizadas por profesionales oftalmólogos. Sin embargo, es apreciable en la Figura 7.6, como las zonas más calientes se centran en secciones de la imagen de aparente relevancia. En las OCT mostradas existen zonas cuya relevancia, a simple vista, es poca o incluso nula: por ejemplo, los fondos negros o, como en el caso de la segunda fila y tercera columna, los bordes cortados en la parte superior de la imagen. Asimismo, se

puede observar como el modelo da poca relevancia a esas zonas, enfocándose en otras que aportan información más valiosa. Sin embargo, para concluir una interpretación definitiva, sería recomendable contar con apoyo experto en la materia.

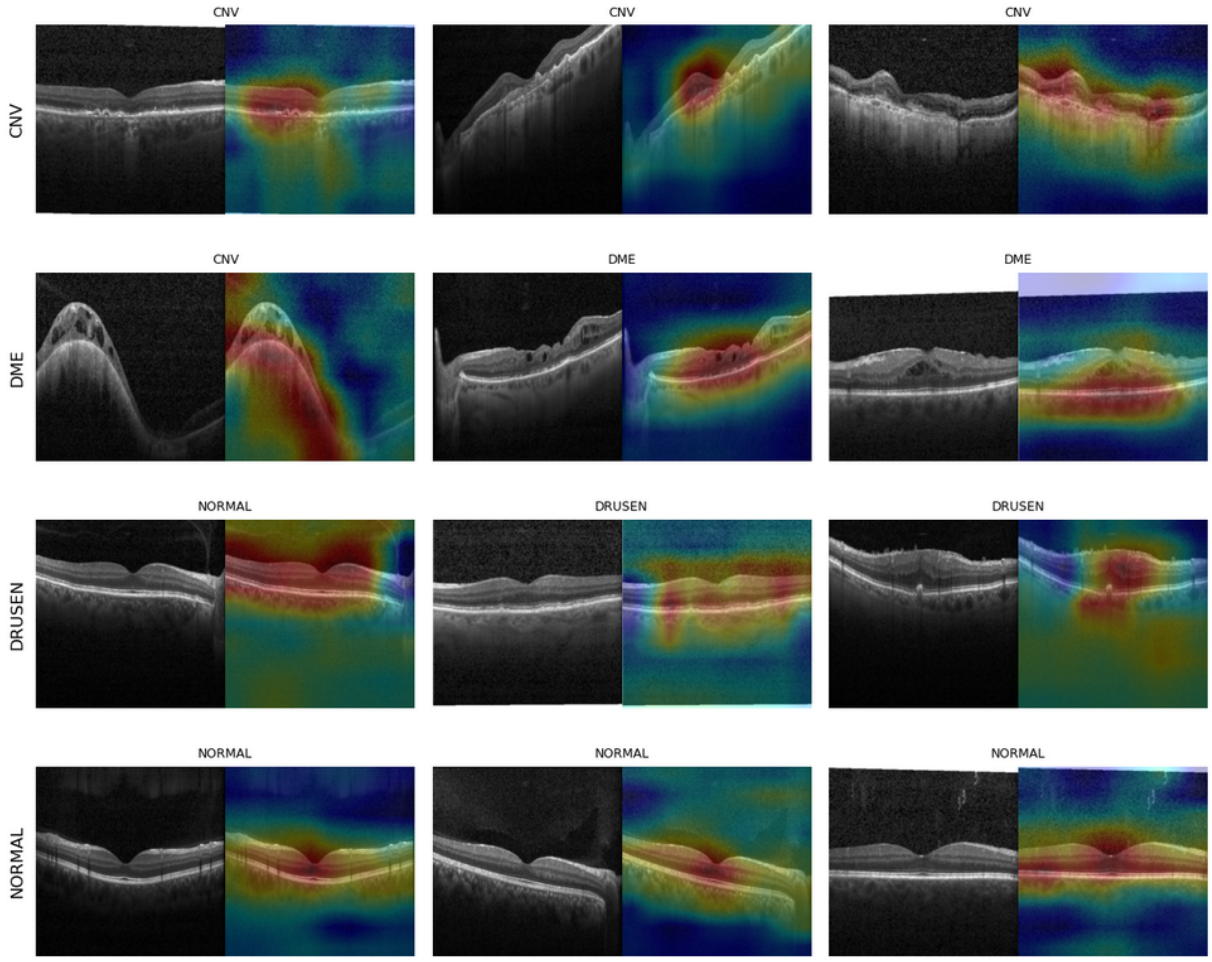


Figura 7.6: Mapas de atención para cada clase.

Capítulo 8

Aplicación

Este capítulo presenta un pequeño proyecto de desarrollo software, centrado en la creación de una herramienta con una interfaz de usuario intuitiva. Dicha herramienta, permitirá el uso del modelo desarrollado para realizar diagnósticos de Edema Macular Diabético, Neovascularización Coroidea y Drusen, utilizando imágenes de secciones transversales provenientes de la Tomografía de Coherencia Óptica. Como complemento, la herramienta proporcionará una justificación detallada del diagnóstico a través de un mapa de calor, el cual resaltará las áreas de la imagen que han resultado relevantes para el diagnóstico.

El presente TFG, aborda como temática principal el estudio y aplicación de Redes Neuronales Convolucionales para la diagnosis de enfermedades oculares, de este modo, la herramienta software desarrollada surge como un complemento que ofrece un resultado tangible del modelo.

Este capítulo cubre principalmente las fases de Análisis y Diseño de un proyecto de Ingeniería de Software, ofreciendo una serie de modelos que faciliten la comprensión de la herramienta desarrollada.

8.1. Análisis

8.1.1. Requisitos

Este capítulo detalla los requisitos elicitados para abordar el desarrollo de la herramienta.

Requisitos funcionales

ID	Nombre	Descripción
RF-01	Cargar imagen	El sistema debe permitir que el usuario cargue una imagen.
RF-02	Diagnosticar	El sistema debe ofrecer diagnósticos sobre el Edema Macular Diabético, la Neovascularización Coroidea y el Drusen a partir de una imagen.
RF-03	Justificación	El sistema debe ofrecer una justificación del diagnóstico ofrecido.
RF-04	Visualización	El sistema debe permitir la visualización del diagnóstico y la justificación.

Cuadro 8.1: Tabla de Requisitos Funcionales

Requisitos no funcionales

ID	Nombre	Descripción
RNF-01	Usabilidad	El sistema debe permitir que un usuario con conocimientos para navegar por internet y subir archivos, obtenga su primer diagnóstico sin ayuda y sin atascarse más de un minuto en el proceso.
RNF-02	Tiempo	El sistema debe tardar menos de un minuto en ofrecer un diagnóstico una vez se ha realizado la petición.
RNF-03	Concurrencia	El sistema debe soportar la concurrencia de múltiples usuarios.
RNF-04	Accesibilidad	El sistema debe permitir su acceso a través de un navegador web.
RNF-05	Portabilidad	El sistema debe ser portable a cualquier entorno con una instalación de Docker.
RNF-06	Formatos de imagen	El sistema debe permitir el uso de imágenes PGN, JPG.
RNF-07	Compatibilidad con PyTorch	El sistema debe ser desarrollado en un lenguaje de programación que permita una compatibilidad completa con PyTorch.

Cuadro 8.2: Tabla de Requisitos no Funcionales

Requisitos de información

ID	Nombre	Descripción
RINF-01	Modelo	El sistema debe almacenar el modelo de diagnóstico.
RNF-02	Diagnostico	El sistema debe almacenar el diagnóstico ofrecido hasta el final de la visualización.
RNF-03	Justificación	El sistema debe almacenar la justificación ofrecida hasta el final de la visualización.

Cuadro 8.3: Tabla de Requisitos Funcionales de Información

8.1.2. Casos de Uso

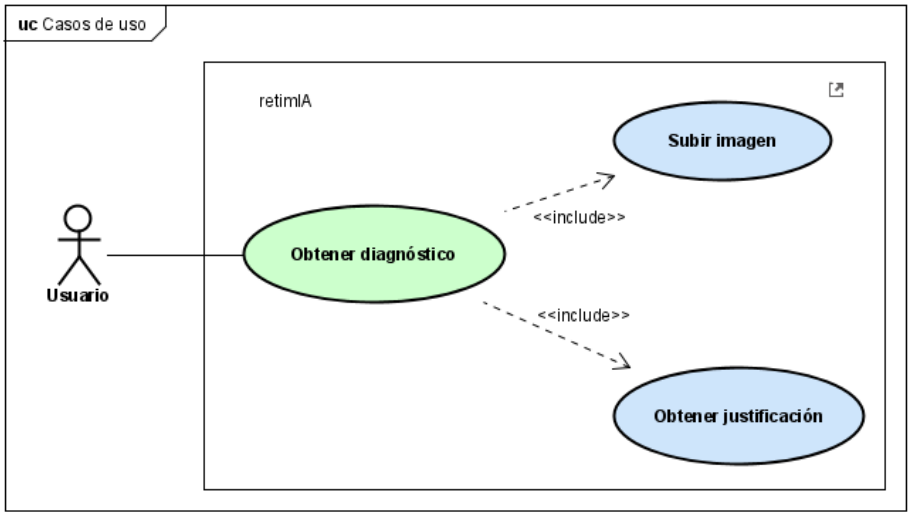


Figura 8.1: Diagrama de Casos de Uso

CU-01	**Subir imagen**
Actor	Usuario
Descripción	El usuario sube una imagen para su posterior diagnóstico.
Precondiciones	1. El usuario ha iniciado el sistema y está en la interfaz de subida de imágenes.
Postcondiciones	1. El sistema ha almacenado la imagen subida por el usuario.
Flujo principal	1. El usuario carga una imagen en el sistema. 2. El sistema verifica el formato de la imagen. 3. El sistema almacena la imagen. 4. El sistema muestra una previsualización de la imagen cargada.
Flujo alternativo	1.a. El usuario carga múltiples archivos simultáneamente. El sistema muestra un mensaje de error y vuelve al paso 1. 2.a. El sistema detecta un formato de imagen no compatible, muestra un mensaje de error y vuelve al paso 1. 4.a. El usuario desea cambiar la imagen seleccionada, repite el paso 1.

Cuadro 8.4: Descripción del caso de uso CU-01: Subir imagen

CU-02	**Obtener diagnóstico**
Actor	Usuario
Descripción	El sistema ofrece un diagnóstico a partir de la imagen cargada.
Precondiciones	1. El usuario ha cargado en el sistema la imagen a diagnosticar.
Postcondiciones	1. El sistema ofrece una visualización de los resultados del diagnóstico
Flujo principal	1. El usuario solicita el diagnóstico de la imagen cargada. 2. El sistema preprocesa la imagen. 3. El sistema carga el modelo de diagnóstico. 4. El sistema obtiene el diagnóstico. 5. El sistema realiza el caso de uso <Obtener justificación>. 6. El sistema ofrece una visualización de los resultados del diagnóstico.
Flujo alternativo	2.a. Se produce un error en el preprocesado de la imagen. El sistema muestra un mensaje de error. 3.a. Se produce un error durante la carga del modelo. El sistema muestra un mensaje de error. 4.a. Se produce un error durante la obtención del diagnóstico. El sistema muestra un mensaje de error. 6.a. El usuario desea obtener otro diagnóstico. Se realiza el caso de uso <Subir imagen>.

Cuadro 8.5: Descripción del caso de uso CU-02: Obtener diagnóstico

CU-03	**Obtener justificación**
Actor	Usuario
Descripción	El sistema produce una justificación del diagnóstico calculado.
Precondiciones	1. El modelo de diagnóstico está cargado en el sistema. 2. La imagen está cargada en el sistema. 3. El sistema ha calculado un diagnóstico para la imagen cargada.
Postcondiciones	1. El sistema almacena la justificación a la predicción calculada.
Flujo principal	1. El sistema calcula la justificación para la predicción a partir del modelo y la imagen. 2. El sistema almacena la justificación.
Flujo alternativo	1.a. Se produce un error durante el cálculo de la justificación. El sistema muestra un mensaje de error. 2.a. Se produce un error durante el almacenamiento de la justificación. El sistema muestra un mensaje de error.

Cuadro 8.6: Descripción del caso de uso CU-03: Obtener justificación

8.1.3. Diagrama de clases inicial

Como idea inicial, se plantea la creación de una clase `GradCam`, la cual a partir de un modelo será capaz de calcular la justificación a sus predicciones. La clase GradCam depende de una abstracción de la clase `CNN`, que representa el modelo desarrollado en este TFG. De esta manera, se aplica el principio de *Inversión de dependencia* [48], procurando una mayor flexibilidad en el sistema, que permitirá una posterior agregación de nuevos modelos, a los cuales, `GradCam` dará soporte.

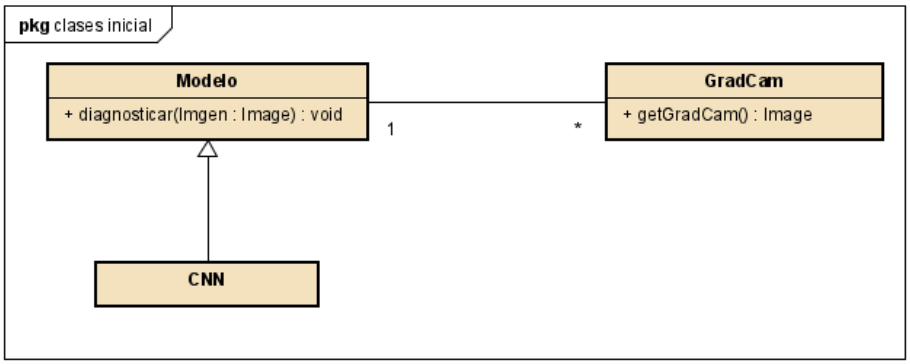


Figura 8.2: Diagrama de Clases inicial

8.2. Diseño

8.2.1. Decisiones de implementación

Una vez realizado el análisis del sistema, atendiendo a los requisitos recogidos, se propone el desarrollo de una aplicación web. Esto cubre eficazmente el requisito no funcional de accesibilidad RNF-04 y ayuda en cierta medida al cumplimiento del requisito de usabilidad RNF-01.

Para cumplir eficazmente con el requisito de compatibilidad con PyTorch RNF-07, se plantea el desarrollo en Python. De este modo, como se expuso en el Capítulo 4, debido a la poca experiencia en proyectos de desarrollo web en Python, se opta por la utilización de Flask [40]. Flask es un microframework para el desarrollo web en Python que permite manejar peticiones HTTP, sesiones de usuario, etc. Flask destaca frente a otras alternativas por su sencillez y su curva de aprendizaje poco pronunciada, resultando estas características de especial utilidad debido a las restricciones temporales para el desarrollo de la aplicación.

Con el objetivo de cumplir el requisito no funcional de concurrencia RNF-03, se opta por utilizar Gunicorn para el despliegue en un entorno de producción. Gunicorn es un

servidor HTTP WSGI para Python, el cual proporciona una manera eficiente de manejar múltiples solicitudes simultáneas [41].

Atendiendo al requisito de usabilidad RNF-01, se implementará una interfaz de usuario intuitiva. Para el desarrollo de la interfaz, se hará uso de código HTML, JavaScript y CSS. Las interfaces desarrolladas están inspiradas en el trabajo [49].

Finalmente, como cobertura al requisito de portabilidad RNF-05, se utiliza Docker para contenerizar todas las dependencias de la aplicación, de esta manera, el sistema será portable a cualquier entorno con una instalación de Docker [42].

8.2.2. Patrones de diseño

Patrón MVT

El patrón Modelo-Vista-Plantilla (MVT), es un patrón comúnmente utilizado en frameworks de desarrollo web como Flask y Django, con el objetivo de organizar el código en componentes separados con responsabilidades específicas, promoviendo así la modularidad y facilitando el mantenimiento y la escalabilidad. El patrón MVT guarda varias similitudes con el conocido patrón Modelo-Vista-Controlador (MVC), pero tiene algunas características que merece la pena destacar [50].

La arquitectura MVT divide la aplicación en tres componentes principales: Modelo, Vista y Plantilla.

- **Modelo** - Este componente se encarga de la gestión de los datos. Define la estructura de los datos, maneja las consultas de la base de datos y aplica las reglas de negocio o lógica de la aplicación. En este sentido, es muy similar al Modelo en el patrón MVC.

- **Vista** - La Vista en MVT, asume parte de las responsabilidades que tendría el Controlador en MVC. En lugar de simplemente presentar los datos al usuario como se haría en una Vista tradicional, en MVT, la Vista también maneja la lógica de control de la aplicación. Se encarga de la recepción de las solicitudes HTTP e interactúa con el Modelo para recuperar o manipular los datos y, posteriormente envía estos datos a la Plantilla para su presentación.

- **Plantilla** - La Plantilla es el componente que se encarga de presentar los datos al usuario. Se corresponde con la Vista en MVC. Utiliza los datos proporcionados por la Vista y los presenta en un formato adecuado para el usuario, generalmente en forma de HTML para las aplicaciones web.

Patrón fachada

El patrón de diseño Fachada o Facade, define una interfaz de alto nivel que facilita el uso de un subsistema, ocultando la complejidad subyacente al mismo [51].

En el caso particular de esta aplicación, se ha creado una clase `Service` que actúa como una Fachada del Modelo, encapsulando la complejidad de sus operaciones. Cuando la Vista recibe una solicitud de diagnóstico del usuario, delega esta solicitud a la clase `Service`. La función correspondiente en la clase interactúa entonces con el modelo para obtener el diagnóstico.

La Fachada no solo simplifica la interacción entre la Vista y el Modelo, sino que también ayuda a reducir el acoplamiento entre el Modelo y la Vista, lo que facilita la evolución independiente de ambos, proporcionando una arquitectura más robusta, flexible y mantenible.

Patrón Factoría

El patrón Factoría define una clase "Factoría" que se encarga de la creación de los objetos de una clase. En este caso concreto, se emplea una Factoría para la instanciación de la aplicación Flask, de esta manera se pueden crear instancias con una configuración de desarrollo, con los modos de depuración activados, o de producción.

Este enfoque tiene una ventaja significativa en comparación con la creación estándar de la aplicación. Cuando se instancia la aplicación sin el uso de una Factoría, no existe la posibilidad de aplicar cambios de configuración de manera dinámica. Cuando el script de inicialización está en ejecución, la instancia de la aplicación ya ha sido creada, por lo que es demasiado tarde para realizar cambios en la configuración. Esto es especialmente relevante para las pruebas unitarias, ya que en ocasiones es necesario ejecutar la aplicación con diferentes ajustes de configuración para lograr una mejor cobertura de las pruebas [50].

8.2.3. Arquitectura

En la presente sección, se detalla la arquitectura de la aplicación desarrollada. Para ello, se hace uso de modelos *Uses Style* a distintos niveles y se presentan diagramas de clases con un nivel de detalle superior al contemplado en la fase de análisis, previamente discutida en la Sección 8.1. De esta manera, se procura proporcionar una visión detallada del diseño y la organización estructural de la aplicación.

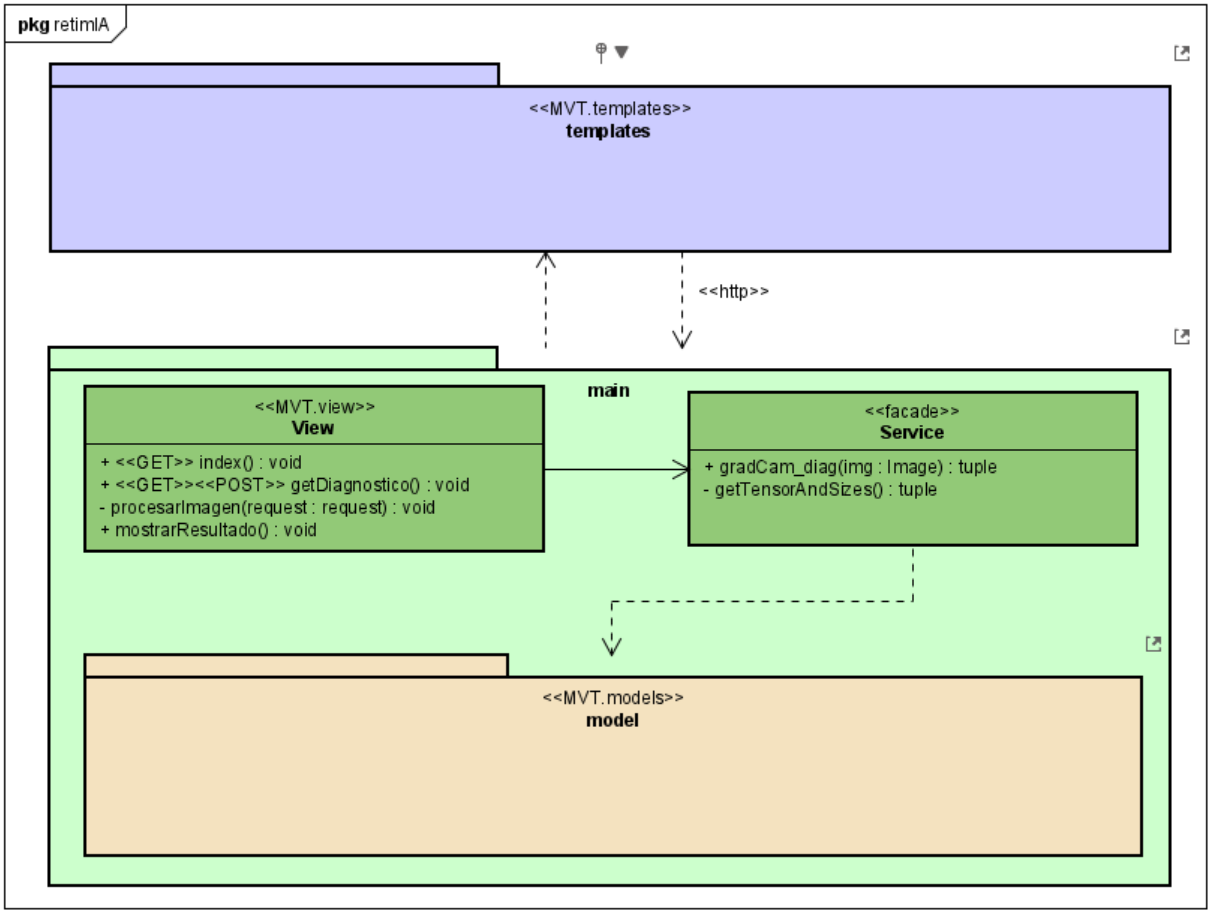


Figura 8.3: Diagrama Uses Style General.

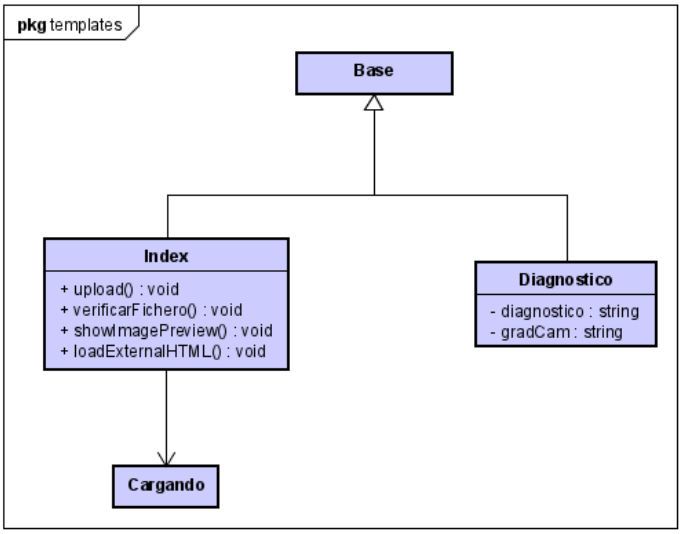


Figura 8.4: Diagrama de Clases del paquete templates.

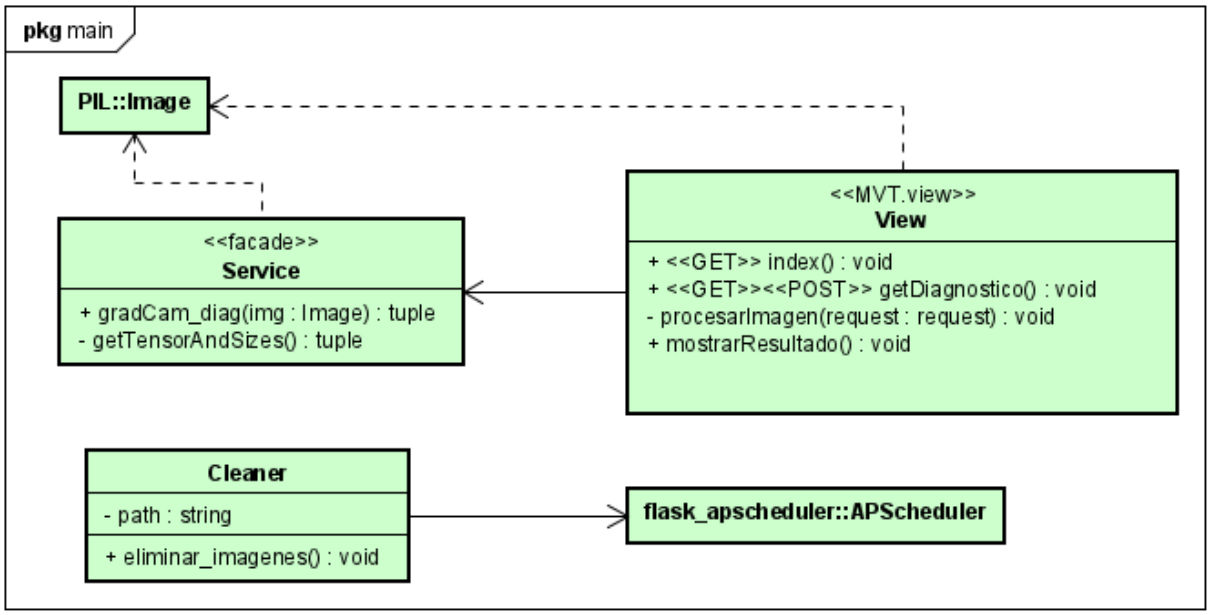


Figura 8.5: Diagrama de Clases del paquete main.

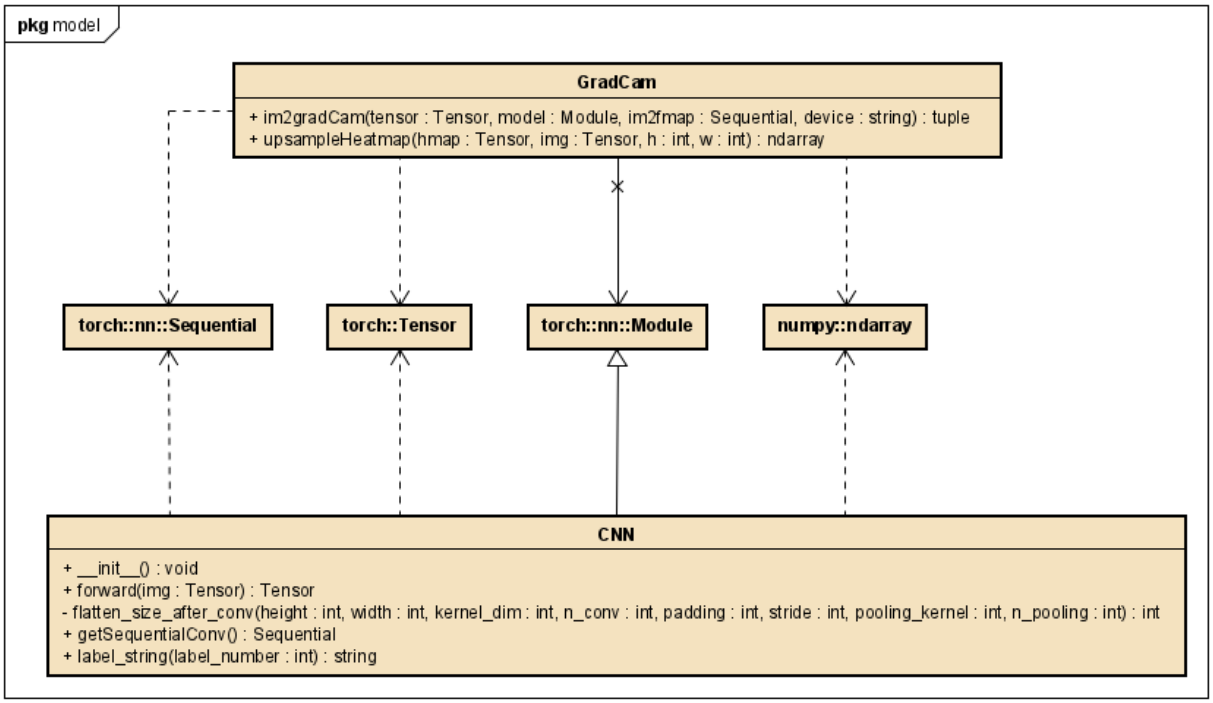


Figura 8.6: Diagrama de Clases del paquete modelo.

8.2.4. Diagramas de secuencia

Esta sección presenta los diagramas de secuencia de los casos de uso detallados en la Sección 8.1.2. Adicionalmente, para detallar las dependencias entre las capas de la arquitectura, en la Figura 8.7 se presenta el diagrama de *Uses Style* del caso de uso principal (Obtener diagnóstico, CU-02).

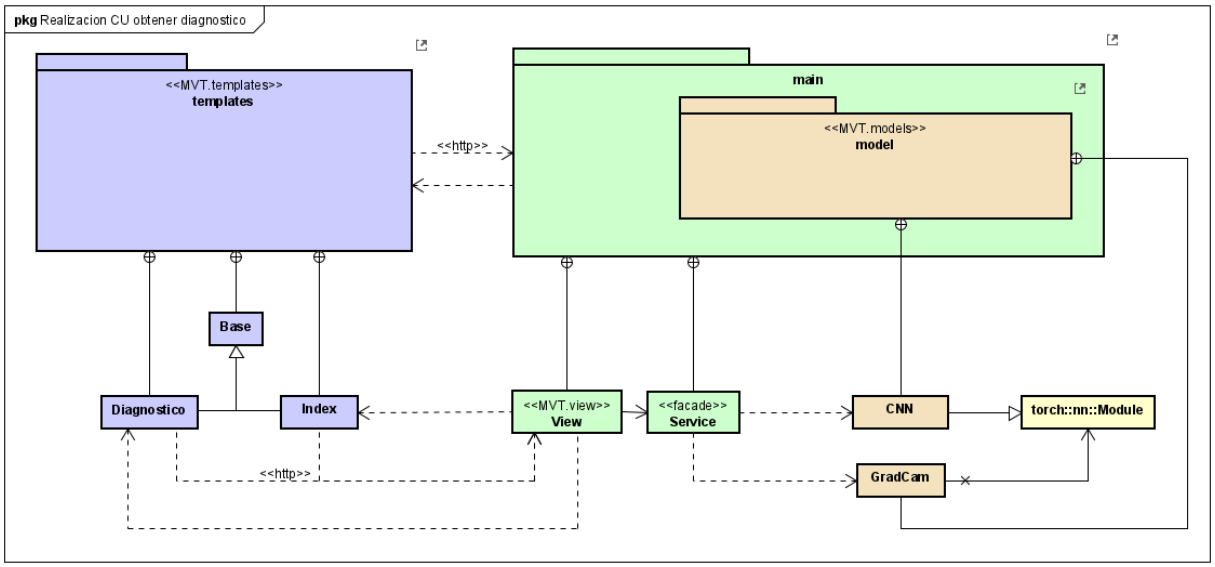


Figura 8.7: Uses Style CU-02, Obtener diagnóstico.

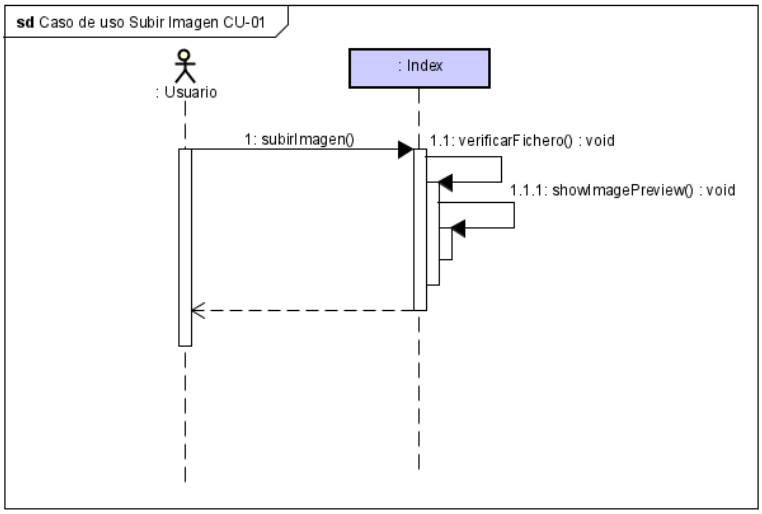


Figura 8.8: Diagrama de secuencia del CU-01, Subir Imagen.

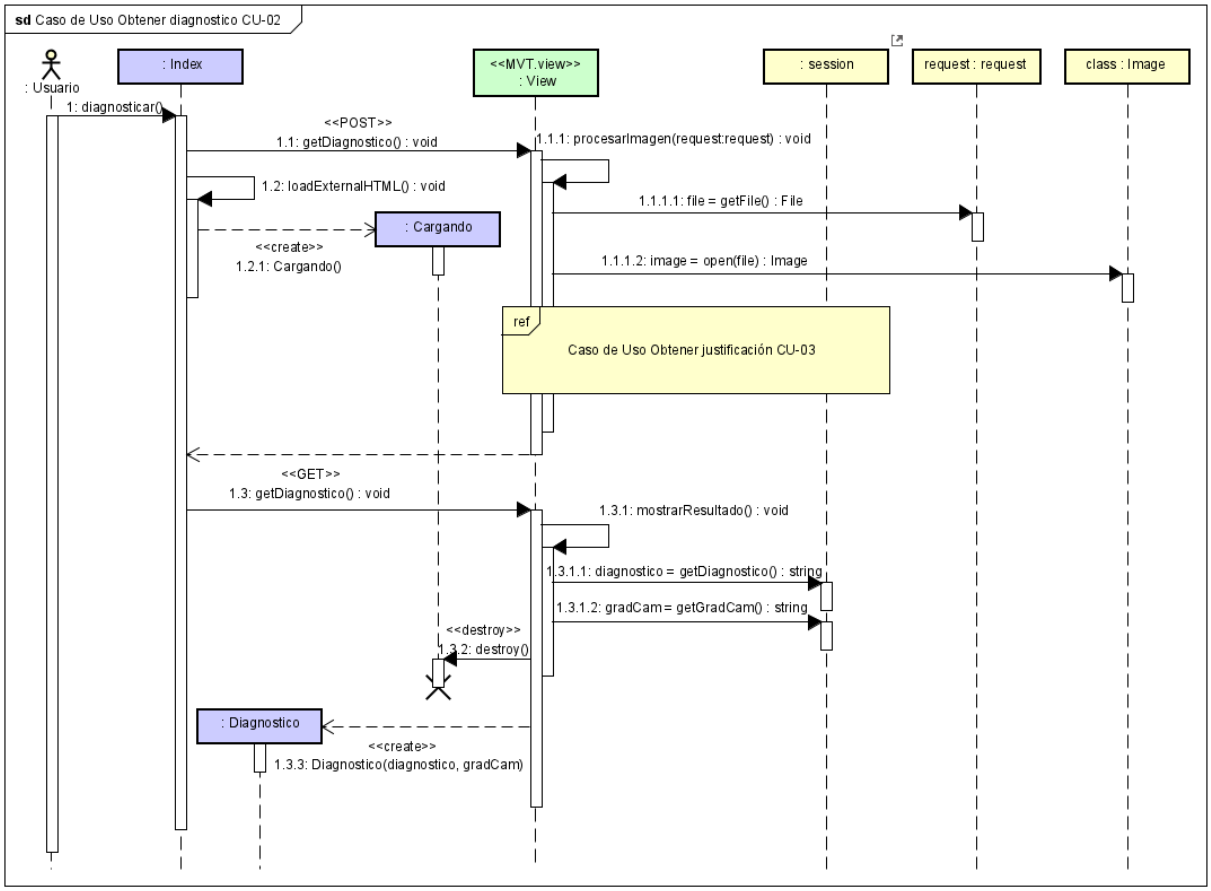


Figura 8.9: Diagrama de secuencia del CU-02, Obtener diagnóstico.

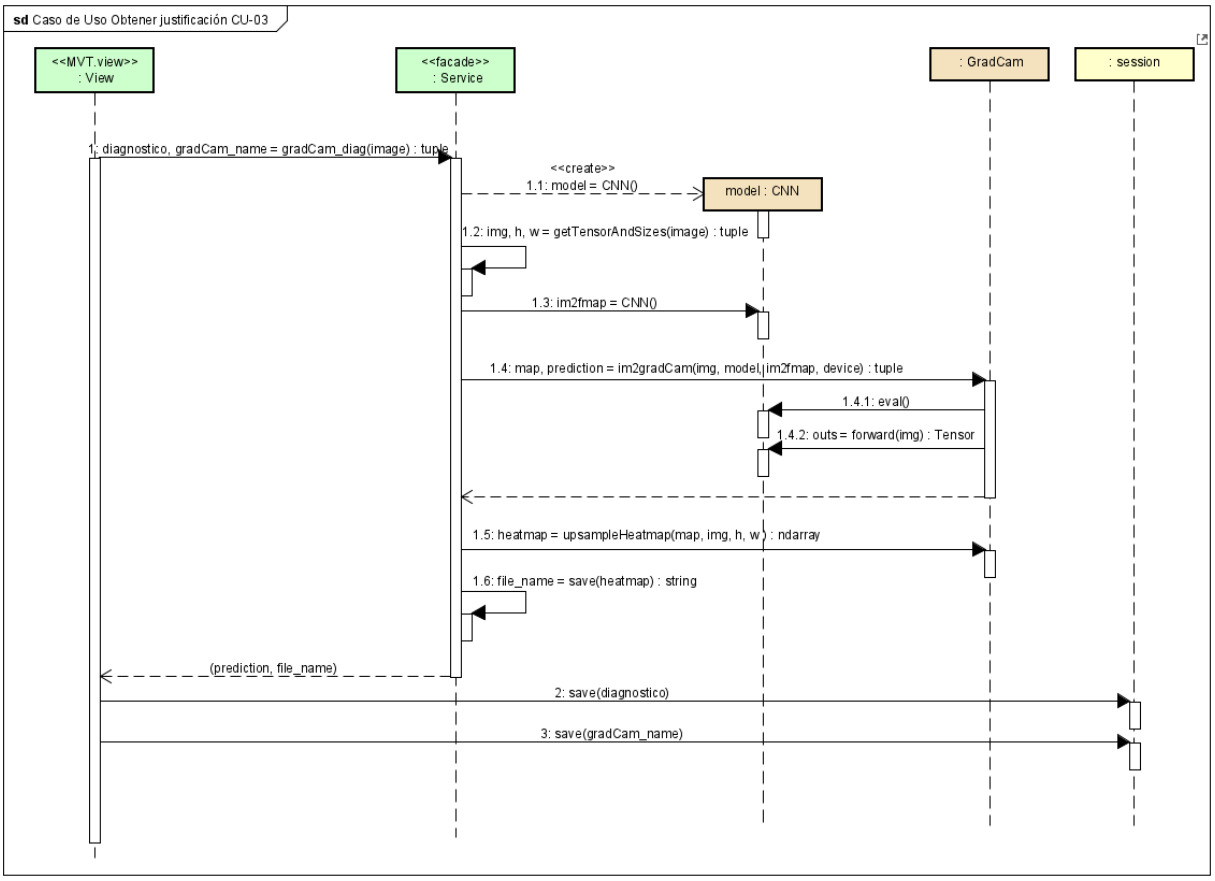


Figura 8.10: Diagrama de secuencia del CU-03, Obtener justificación.

8.2.5. Implementación

Se presenta en el CD entregado el código fuente completo para la ejecución y despliegue de la aplicación. La implementación de esta aplicación toma como referencia [49], y una serie de buenas prácticas para el desarrollo mantenible y escalable de aplicaciones web en Flask [50].

8.2.6. Pruebas

Debido a las restricciones temporales del proyecto, no se ha desarrollado una batería de pruebas unitarias. No obstante, se han probado mediante el uso de la interfaz, los flujos principales y alternativos descritos en la Sección 8.1.2.

8.2.7. Seguridad

En general, dotar de medidas de seguridad a las aplicaciones web es una práctica altamente recomendable. Sin embargo, al tratarse esta de una aplicación simple cuyo objetivo es la demostración del funcionamiento del modelo de diagnóstico desarrollado, la implementación de medidas de seguridad robustas quedan fuera del alcance del proyecto. No obstante, una medida de seguridad aplicada es la previa comprobación de las extensiones de los archivos que se cargan al sistema.

8.2.8. Despliegue

Como se ha mencionado con anterioridad, con el objetivo de disponer de una aplicación con gran portabilidad, se ha realizado la configuración de Docker [42]. De esta manera, se podrá desplegar la aplicación mediante la ejecución del comando `docker compose up -d` o `docker-compose up -d` dependiendo de la versión de Docker instalada.

La configuración actual del Dockerfile, desplegará la aplicación en un entorno de producción en Gunicorn [41], siendo esta accesible en `http://localhost:5000/`.

Capítulo 9

Conclusiones

9.1. Aprendizaje percibido

El desarrollo del presente TFG ha supuesto un gran reto y avance en materia de aprendizaje. El contenido principal es tratado en mayor medida en las asignaturas de *Técnicas de Aprendizaje Automático* y *Minería de Datos*, las cuales han aportado una base fundamental sobre la que construir este trabajo. No obstante, su consecución se apoya fuertemente en múltiples disciplinas transversales a la Ingeniería Informática, resultando especialmente relevantes:

- **Programación Orientada a Objetos** - Necesaria para la implementación de los modelos y la aplicación web.
- **Servicios y Sistemas Web** - Resultando especialmente útil en el desarrollo de la aplicación web.
- **Estadística** - Materia transversal a todo lo relacionado con el tratamiento de datos, con una fuerte presencia en la creación y evaluación de Redes Neuronales Artificiales.
- **Modelado y Diseño de Software** - A pesar de la sencillez de la aplicación desarrollada, se han tratado de incluir patrones de diseño y modelos relativos a la Ingeniería de Software.
- **Desarrollo Basado en Componentes Software** - El uso de Docker en esta asignatura ha facilitado su configuración para la aplicación creada.
- **Planificación y Gestión de Proyectos** - Ha resultado de especial utilidad para afrontar un proyecto de estas dimensiones.

- **Profesión y Sociedad** - El peso y formato de los trabajos escritos que se han redactado en esta asignatura, han contribuido positivamente a la correcta redacción de esta memoria.

De igual forma, la gran mayoría de asignaturas cursadas no mencionadas han contribuido directamente en la consolidación de unas bases teóricas y prácticas, que han permitido abordar este TFG.

Resulta interesante, como la realización de este TFG ha ofrecido la posibilidad de explorar y profundizar en el Aprendizaje Profundo, así como en la Visión por Computador. Esto ha supuesto un desafío, pues es un tema escasamente tratado en la carrera. En particular, las Redes Neuronales Convolucionales han sido brevemente abordadas en el último tema de la asignatura Minería de Datos. No obstante, se trabajaron mediante el uso de Keras, el cual fue pensado para su sencillez. Por tanto, el uso de PyTorch ha sido completamente novedoso, pero su dificultad y flexibilidad adicional han promovido un mayor entendimiento del funcionamiento interno de estas redes.

Adicionalmente, explorar técnicas para mejorar la interpretabilidad de las CNN, es algo completamente nuevo y no estudiado en la carrera. Esto ha brindado la oportunidad de conocer métodos como Grad-CAM, que resulta de gran utilidad para mejorar la explicabilidad de estos modelos predictivos, contribuyendo en gran medida a la detección de sesgos.

Por último, vale la pena mencionar el aprendizaje que ha supuesto la integración del modelo propuesto en una aplicación de usuario. Esta experiencia ha permitido familiarizarse con Flask, como plataforma de despliegue para este tipo modelos.

9.2. Consecución de objetivos

En su origen, este TFG plantea una serie de objetivos principales que, en su mayoría, son alcanzados satisfactoriamente.

El principal de ellos es la creación de un clasificador de imágenes OCT con una alta tasa de acierto. Tras el preprocesado de los datos, el ajuste iterativo de hiperparámetros y el entrenamiento del modelo, se ha logrado obtener un sistema de diagnóstico con una precisión del $96,531\,\%$ en el conjunto de prueba. Este resultado, en principio, parece bastante aceptable, especialmente debido a que, gracias al conjunto de validación, el de prueba no ha intervenido directa ni indirectamente en el proceso de entrenamiento. No obstante, para obtener una estimación más robusta de la precisión del modelo, sería recomendable el uso de validación cruzada, técnica que ha sido descartada para este TFG, pues supondría un tiempo de cómputo no asumible.

Otro de los objetivos centrales alcanzados es el estudio de la interpretabilidad del modelo propuesto. Mediante el uso de Grad-CAM, se ha expuesto una serie de ilustraciones que muestran las zonas más relevantes en la clasificación de las imágenes procesadas. Una inspección visual de estos, parece exponer un comportamiento coherente en el modelo. Sin embargo, para una mayor comprensión del sistema, sería recomendable el estudio de un mayor número de imágenes, preferentemente asistido por personal médico cualificado.

Finalmente, se plantea el despliegue del sistema de diagnóstico desarrollado mediante la creación de una aplicación de usuario. Como se ha visto, se ha conseguido crear una herramienta sencilla, capaz no solo de clasificar una imagen OCT, sino de generar una justificación visual de la misma. Cabe destacar, que habría sido preferible la creación de una herramienta más completa, dado que la que ha sido finalmente desarrollada, por su sencillez, no cumple los requisitos para ser utilizada en un entorno real. No obstante, su utilidad reside en una demostración tangible de las capacidades del modelo para un usuario no especializado.

Por lo que se concluye que, a pesar de existir múltiples caminos de mejora, el trabajo desarrollado cumple exitosamente los objetivos principales propuestos.

9.3. Líneas futuras

Como se ha comentado previamente, a partir de la realización de este TFG se abren varios caminos para mejorar y complementar el trabajo desarrollado.

El más inmediato es la creación de una herramienta más completa, que pueda ser utilizada en un entorno real. Un software que permita la gestión de pacientes, así como el procesado de múltiples imágenes de forma simultánea, podría ayudar a agilizar en gran medida el trabajo de los oftalmólogos en la detección de enfermedades retinianas mediante imágenes OCT.

Con respecto al modelo propuesto, hay varios aspectos que se podrían plantear:

- El uso de hardware especializado, permitiría no solo el ajuste de un mayor número de parámetros en un tiempo razonable, sino también el empleo de una mayor resolución en las imágenes de entrada y un tamaño de lote menor. Si bien un modelo más complejo no siempre da mejores resultados, estos cambios tienen el potencial de incrementar la complejidad de las características detectadas, y merece la pena estudiarlos.

- Utilizar la técnica de validación cruzada que, como ya se comentó, aumentaría la robustez y fiabilidad de la estimación de precisión.

- Explorar en detalle los mapas de atención generados al procesar un mayor número de imágenes no presentadas en el entrenamiento. En particular, sería especialmente interesante contrastar las áreas relevantes para el modelo con las usadas por los oftalmólogos para diagnosticar estas enfermedades. A su vez, puede resultar de gran interés medir y observar como se comporta el sistema de diagnóstico en un conjunto adicional con imágenes similares.
- Por último, podría s la construcción de un sistema basado en expertos, donde existen modelos especializados para cada una de las clases. Tener sistemas de diagnóstico enfocados en la detección de una única patología, podría ofrecer mejores resultados, y además, no se limitaría a un único diagnóstico por imagen. Lamentablemente, las limitaciones en el tiempo disponible no han permitido este desarrollo en el presente TFG.

Apéndice A

Manual de instalación y administración

A.1. Instalación

Gracias al uso de Docker, desplegar la aplicación en un entorno de producción resulta bastante sencillo. Para ello, es necesario disponer de un sistema con una instalación de Docker y Docker-compose, la cual se puede hacer siguiendo la Documentación Oficial.

Cabe destacar que esta guía expone el proceso seguido para la instalación en Ubuntu 22.04.2 LTS, y puede no ser funcional en otras distribuciones o sistemas operativos. Sin embargo, el uso de Docker debería permitir su instalación en múltiples plataformas. Las versiones concretas de Docker y Docker-compose utilizadas son la 23.0.1 y la v2.12.1 respectivamente.

Una vez se cumplen los requisitos de instalación, habrá que navegar al directorio `retimIA/` incluido en el CD entregado, en el cual se encuentra el archivo `docker-compose.yml` y ejecutar `docker-compose build` o en su defecto `docker compose build`, dependiendo de la versión de Docker-compose instalada.

Para concluir, solo será necesario ejecutar el comando `docker-compose up` o `docker compose up` para levantar el contenedor. Tras esto, la aplicación será accesible por defecto en `http://localhost:5000/`, siendo la IP y el puerto modificables mediante su configuración en el archivo `docker-compose.yml`.

A.2. Administración

Para arrancar la aplicación en un entorno de desarrollo, se facilita el entorno virtual `venv` en el CD entregado, cuya estructura se discute en el Anexo C. La configuración descrita es operativa en Ubuntu 22.04.2 LTS y puede ser aplicable a otros sistemas operativos GNU/Linux.

Con una terminal abierta, situar `CD-ROM/APP` como directorio de trabajo y ejecutar los siguientes comandos:

```
1 $ export APP_PATH=$(pwd)
2 $ source venv/bin/active
3 $ echo $VIRTUAL_ENV
```

Tras esto, se debería mostrar por pantalla una ruta absoluta al directorio `venv`. De ser así, se procederá de la siguiente manera:

```
1 $ export FLASK_APP=retimIA.py
2 $ cd retimIA
3 $ flask run
```

Una vez se ha concluido, la aplicación habrá sido desplegada en un entorno de desarrollo y será accesible en `https://localhost:5000`.

Apéndice B

Manual de usuario

Este manual dispone una guía de uso de la aplicación de diagnóstico desarrollada.

Una vez efectuado el acceso a la aplicación, se presenta en pantalla la ventana principal, ilustrada en la Figura B.1. La forma de proceder para obtener un diagnóstico es la siguiente:

- Arrastrar una imagen al cuadro de Drag & Drop o bien seleccionarla mediante el botón "Examinar", el cual abrirá un explorador de archivos para la selección de la imagen. Una vez la imagen ha sido seleccionada, se mostrará en pantalla una previsualización de la misma, como muestra la Figura B.2.
- A continuación se presiona el botón "Diagnosticar", que activará la pantalla de carga de la Figura B.3 y se mantendrá en pantalla mientras se procesa el diagnóstico.
- Finalmente, una vez listos, los resultados se mostrarán en pantalla como se expone en la Figura B.4.

Cabe destacar que un uso incorrecto de la herramienta puede desencadenar en los errores mostrados en las figuras B.5, B.6, B.7.

Figura B.1: Página de inicio.

Figura B.2: Previsualización de la imagen cargada.

Figura B.3: Pantalla de carga.

Figura B.4: Visualización de resultados.

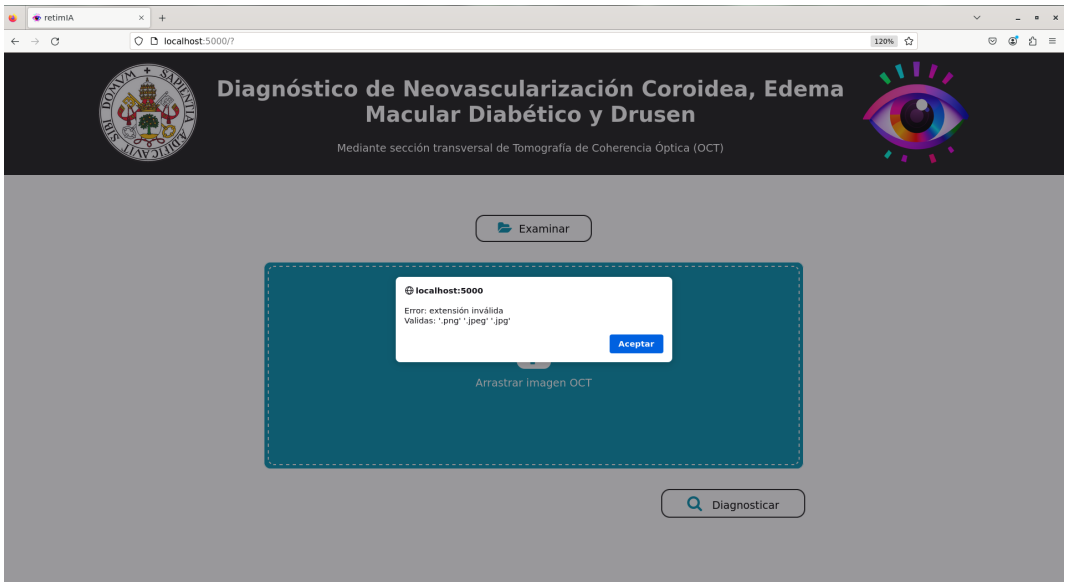


Figura B.5: Archivo de formato no válido.

Figura B.6: No se ha introducido ninguna imagen.

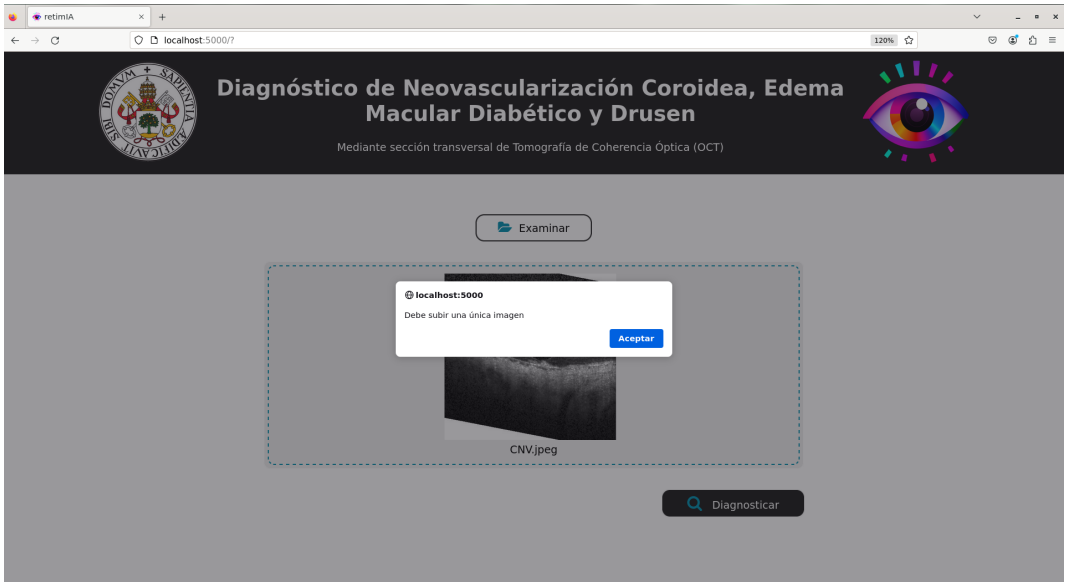


Figura B.7: Se han introducido varios archivos.

Apéndice C

Contenidos del CD-ROM

La finalidad de este anexo es exponer brevemente la estructura y contenidos del CD entregado, para ello, la Figura C.1 muestra una visión superficial de elementos disponibles.

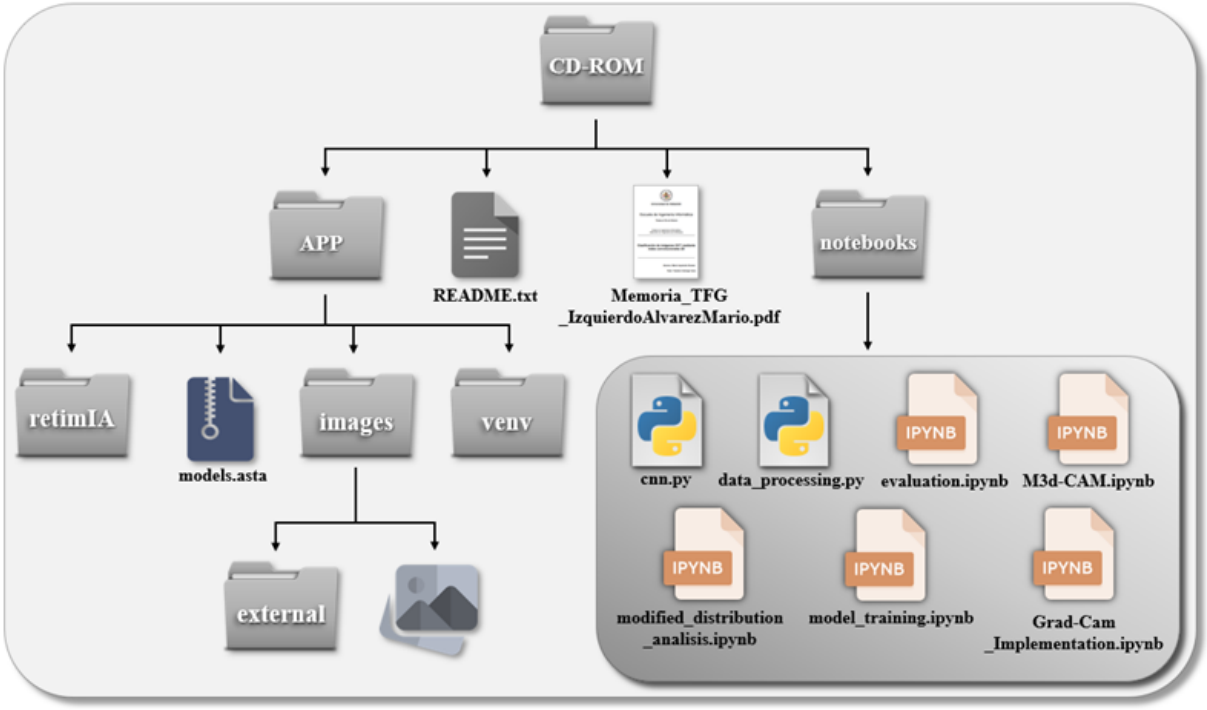


Figura C.1: Estructura del CD entregado.

Como se puede observar, el CD contiene esta memoria, un README con una breve descripción de los contenidos y, los directorios `notebooks/` y `APP/`.

El primero de ellos almacena tanto los Jupyter Notebooks utilizados para la creación,

entrenamiento y evaluación del modelo propuesto, como los usados para la visualización de mapas de atención (`M3d-CAM.ipynb` y `Grad-CAM_implementation.ipynb`). Adicionalmente, en `cnn.py` y `data_processing.py` se incluyen dos clases para la definición del modelo y el procesamiento de datos respectivamente, las cuales son utilizadas en todos los notebooks entregados. Por último, en `modified_distribution_analisis.ipynb` se muestra un estudio de la distribución del conjunto resultante tras aplicar los procedimientos de preprocesado.

Por otro lado, el directorio `APP` contiene lo relativo a la aplicación desarrollada, incluyendo tanto los modelos, como todo lo necesario para la instalación y prueba de la misma:

- Bajo el directorio `retimIA/` se encuentra todo el código fuente de la aplicación, ficheros de configuración y despliegue.
- En `images/`, con el fin de facilitar una rápida comprobación del desempeño de la aplicación, se dispone una imagen de cada posible clase. Además, se ha añadido un ejemplo de cada anomalía en `images/external`, los cuales han sido extraídos de fuentes externas al conjunto de datos empleado en este trabajo.
- Para permitir la administración de la aplicación, se incluye el entorno virtual `venv/`, configurado con las dependencias necesarias para la correcta ejecución de la aplicación.
- Finalmente, el archivo `models.asta` contiene los modelos desarrollados para el análisis y diseño de la aplicación.

Bibliografía

[1] Martin Krause. ¿La Inteligencia Artificial va a cambiar nuestras vidas para siempre? Parece que sí. https://bazar.ufm.edu/la-inteligencia-artificial-va-cambiar-nuestras-vidas-siempre-parece/, 13 MAY 2023. Accedido el 19 de mayo de 2023.

[2] IBM. Inteligencia Artificial en la Medicina. https://www.ibm.com/es-es/topics/artificial-intelligence-medicine. Accedido el 19 de mayo de 2023.

[3] Paul Mooney. Retinal OCT Images (optical coherence tomography). https://www.kaggle.com/datasets/paultimothymooney/kermany2018, 2018. Accedido el 19 de mayo de 2023.

[4] J. G. Fujimoto, C. Pitris, S. A. Boppart, and M. E. Brezinski. Optical coherence tomography: an emerging technology for biomedical imaging and optical biopsy. *Neoplasia (New York, N.Y.)*, 2(1-2):9–25, 2000.

[5] Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *Nature*, 521:436–444, 2015.

[6] O. Musat, C. Cernat, M. Labib, A. Gheorghe, O. Toma, M. Zamfir, and A. M. Boureanu. Diabetic Macular Edema. *Romanian Journal of Ophthalmology*, 59(3):133–136, 2015.

[7] N. J. Y. Yeo, E. J. J. Chan, and C. Cheung. Choroidal Neovascularization: Mechanisms of Endothelial Dysfunction. *Frontiers in Pharmacology*, 10, 2019.

[8] A. M. VanDenLangenberg and M. P. Carson. Drusen Bodies. https://www.ncbi.nlm.nih.gov/books/NBK559087/, 2022. Accedido el 19 de mayo de 2023.

[9] A. Holzinger, P. Kieseberg, E. Weippl, and A.M. Tjoa. Current Advances, Trends and Challenges of Machine Learning and Knowledge Extraction: From Machine Learning to Explainable AI. In *Machine Learning and Knowledge Extraction*, pages 1–8. Springer International Publishing, 2018.

[10] W. Samek, T. Wiegand, and K.-R. Müller. Explainable Artificial Intelligence: Understanding, Visualizing and Interpreting Deep Learning Models. https://arxiv.org/abs/1708.08296, 2017.

[11] SAS® Help Center. Introduction to SEMMA. `http://documentation.sas.com/doc/en/emref/14.3/n061bzurmej4j3n1jnj8bbjjm1a2.htm`, Aug. 30, 2017. Accedido el 19 de mayo de 2023.

[12] Microsoft. What is the Team Data Science Process? - Azure Architecture Center. `https://learn.microsoft.com/en-us/azure/architecture/data-science-process/overview?view=azureml-api-2`. Accedido el 19 de mayo de 2023.

[13] IBM. Guía de CRISP-DM de IBM SPSS Modeler. `https://www.ibm.com/docs/es/SS3RA7_18.4.0/pdf/ModelerCRISPDM.pdf`. Accedido el 19 de mayo de 2023.

[14] Peter Chapman, Janet Clinton, Randy Kerber, Tom Khabaza, Thomas P. Reinartz, Colin Shearer, and Richard Wirth. CRISP-DM 1.0: Step-by-step data mining guide. `https://www.semanticscholar.org/paper/CRISP-DM-1.0%3A-Step-by-step-data-mining-guide-Chapman-Clinton/54bad20bbc7938991bf34f86dde0babfbd2d5a72`, 2000.

[15] Principios del manifiesto ágil. `https://agilemanifesto.org/iso/es/principles.html`. Accedido el 19 de mayo de 2023.

[16] T. Calonge Cano. REDES NEURONALES ARTIFICIALES: Técnicas de Aprendizaje Automático, 2021-2022. Departamento de Informática, Universidad de Valladolid.

[17] Prajit Ramachandran, Barret Zoph, and Quoc V. Le. Búsqueda de Funciones de Activación. 2017.

[18] T. Calonge Cano. PERCEPTRÓN MULTICAPA: Técnicas de Aprendizaje Automático, 2021-2022. Departamento de Informática, Universidad de Valladolid.

[19] Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *Nature*, 521:436–444, 2015.

[20] T. Calonge Cano. PERCEPTRÓN SIMPLE Y ADALINE: Técnicas de Aprendizaje Automático. 2021-2022. Departamento de Informática, Universidad de Valladolid.

[21] J. Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61:85–117, 2015.

[22] A. Thampi. *Building Explainable Machine Learning Systems*. Manning Publications Co., 2022.

[23] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016.

[24] T. Calonge Cano. REDES NEURONALES CONVOLUTIVAS: Minería de datos. 2022-2023. Departamento de Informática, Universidad de Valladolid.

[25] Tashmit. Convolution layer, Padding, Stride, and Pooling in CNN. Mar 2023.

[26] A. Zhang, Z. C. Lipton, M. Li, and A. J. Smola. *Dive into Deep Learning*. d2l.ai.

[27] S. Khosla. CNN | Introduction to Pooling Layer. `https://www.geeksforgeeks.org/cnn-introduction-to-pooling-layer/`. Accedido el 27 de mayo de 2023.

[28] G. Boesch. VGG Very Deep Convolutional Networks (VGGNet) – What you need to know. `https://viso.ai/deep-learning/vgg-very-deep-convolutional-networks/`. Accedido el 27 de mayo de 2023.

[29] V. Kishore, A. Ayyadevara, and Y. Reddy. *Modern Computer Vision with PyTorch*. Birmingham, 2020.

[30] S. Ioffe and C. Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. `https://arxiv.org/abs/1502.03167`, 2015. Accedido el 28 de mayo de 2023.

[31] S. Tong and L. Kagal. Investigating Bias in Image Classification using Model Explanations. `https://arxiv.org/abs/2012.05463`, 2020. Accedido el 28 de mayo de 2023.

[32] Eibe Frank, Mark A. Hall, and Ian H. Witten. The WEKA Workbench. Online Appendix for "Data Mining: Practical Machine Learning Tools and Techniques", Morgan Kaufmann, Fourth Edition, 2016.

[33] Matlab documentation. `https://www.mathworks.com/help/matlab/`. Accedido el 29 de mayo de 2023.

[34] Julia 1.9 documentation. `https://docs.julialang.org/`. Accedido el 29 de mayo de 2023.

[35] Python Documentation. `https://docs.python.org/`. Accedido el 29 de mayo de 2023.

[36] Scikit-learn Documentation. `https://scikit-learn.org/stable/`. Accedido el 29 de mayo de 2023.

[37] TensorFlow Documentation. `https://www.tensorflow.org/api_docs`. Accedido el 29 de mayo de 2023.

[38] Keras Documentation. `https://keras.io/`. Accedido el 29 de mayo de 2023.

[39] PyTorch Documentation. `https://pytorch.org/docs/stable/index.html`. Accedido el 29 de mayo de 2023.

[40] Flask Documentation. `https://flask.palletsprojects.com/`. Accedido el 29 de mayo de 2023.

[41] Gunicorn Documentation. `https://docs.gunicorn.org/en/stable/`. Accedido el 29 de mayo de 2023.

[42] Docker Documentation. `https://docs.docker.com/`. Accedido el 29 de mayo de 2023.

[43] Lutz Prechelt. *Early Stopping - But When?*, pages 53–67. 2012.

[44] C.A. González. Metodología experimental para la creación y evaluación de hipótesis. Revisión y ampliación, 2021-2022. Grupo de Sistemas Inteligentes, Departamento de Informática, Universidad de Valladolid.

[45] D. P. Kingma and J. Ba. Adam: A Method for Stochastic Optimization. `https://arxiv.org/abs/1412.6980`, 2017. Recuperado el 3 de junio de 2023.

[46] Medcam Documentation. `https://meclabtuda.github.io/M3d-Cam/medcam/medcam_inject.html`, 2023. Recuperado el 4 de junio de 2023.

[47] PIL (Python Imaging Library) Documentation. `https://pillow.readthedocs.io/en/stable/`, 2023. Recuperado el 4 de junio de 2023.

[48] R. C. Martin. *Design Principles and Design Patterns*. 2000.

[49] S. Duque. *Estudio y Aplicación de Redes Convolucionales a la Clasificación de Imágenes Estáticas*. 2019.

[50] M. Grinberg. *Flask Web Development*. O'Reilly Media, Inc., 2 edition, 2018.

[51] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.