Universidad de Valladolid

ESCUELA DE INGENIERÍA INFORMÁTICA

GRADO EN INGENIERÍA INFORMÁTICA
Mención en Computación

Estudio comparativo de clasificación de imágenes médicas, usando técnicas de Inteligencia Artificial basadas en Transformers, frente a Redes Convolucionales.

Alumno: Ismael Carbajo Valor

Tutor: Teodoro Calonge Cano

La obsesión vence al talento.

Agradecimientos

En primer lugar, quiero expresar mi más sincero agradecimiento a mi tutor, Teodoro, por su tiempo, dedicación y orientación a lo largo de este proceso. A mi familia, por creer en mí incondicionalmente y brindarme su apoyo constante. Y a mis amigos, por su compañía, ánimos y por compartir conmigo tantos momentos a lo largo de estos años.

Resumen

Resumen

La aplicación de modelos de Aprendizaje Automático en el ámbito de la Medicina, ha demostrado un gran potencial en tareas de diagnóstico y clasificación de imágenes. En este Trabajo de Fin de Grado, se ha explorado el uso de arquitecturas Vision Transformer (ViT), un enfoque relativamente reciente que ha mostrado resultados prometedores en Visión Artificial como alternativa a las tradicionales Redes Neuronales Convolucionales (CNN).

El objetivo principal ha sido desarrollar e implementar un sistema de clasificación de imágenes médicas basado en ViT, evaluando su rendimiento sobre tres conjuntos de datos distintos: radiografías de tórax, resonancias magnéticas cerebrales (MRI) y tomografías de coherencia óptica (OCT). Para ello, se han desarrollado desde cero diversas variantes de modelos ViT, incorporando diferentes técnicas. Cada uno de estos modelos cuenta con mapas de explicabilidad a través de ViT-ReciproCAM.

En cuanto a los resultados, se ha observado una mejora notable respecto a modelos previos en uno de los tres conjuntos de datos. Sin embargo, en los otros dos conjuntos, no se han obtenido resultados superiores a los logrados con enfoques basados en CNN, principalmente debido a las dificultades de generalización que presentan los ViT en situaciones de muestras limitadas.

Palabras clave: Aprendizaje Profundo, Vision Transformer, Clasificación de imágenes médicas (CXR, MRI, OCT), ViT-ReciproCAM.

Abstract

Machine Learning applied to Medicine has shown great potential in diagnosis and image classification tasks. In this Double Degree Thesis has been explored the use of Vision Transformer (ViT) architectures, a relatively recent approach that has demonstrated promising results in Computer Vision as an alternative to traditional Convolutional Neural Networks (CNNs).

The main goal of this work has been to develop and implement a medical image classification system based on ViT, evaluating its performance on three different datasets: chest X-rays, brain magnetic resonance imaging (MRI) and optical coherence tomography scans (OCT). To do that, several ViT model variants have been developed from scratch. Each of these models includes explainability maps using ViT-ReciproCAM.

Regarding the results, a significant improvement was observed in just one of the related datasets. However, for the other ones, the results are not bigger than those achieved with CNN-based approaches. It is mainly because ViT models with an insufficient number of samples present serious limitations due to a reduced power of generalization in practice.

Keywords: Deep Learning, Vision Transformer, Medical Image Classification (CXR, MRI, OCT), ViT-ReciproCAM.

Índice general

Lista de Figuras

Lista de Tablas

Capítulo 1

Introducción

1.1. Contexto

Durante los últimos años, la Inteligencia Artificial (IA) se ha ido haciendo paso como una de las tecnologías más innovadoras y revolucionarias de nuestra época. Su gran cantidad de aplicaciones abarca desde tareas tan cotidianas como recomendaciones de recetas, hasta la búsqueda de soluciones complejas en sectores como medicina, industria y logística. Esta capacidad de adaptación ha hecho que se consolide como una parte ya casi fundamental del mundo en el que vivimos.

Su integración en la sociedad ha transformado totalmente la forma en que trabajamos, nos comunicamos e incluso tomamos decisiones. Su capacidad de aprendizaje y evolución constante, combinada con la gran cantidad de datos que se generan actualmente, los cuales son cada vez mayores [18], ha provocado un cambio radical en muchos ámbitos, donde la precisión y eficiencia son esenciales.

Uno de estos sectores que se ha visto beneficiado, es la medicina. El uso e implementación de algoritmos avanzados ha permitido mejorar de manera significativa la rapidez y precisión de diagnósticos complejos. Aunque, evidentemente, no pueden llegar a sustituir a los profesionales de la salud, estas herramientas de apoyo cuentan con gran robustez y fiabilidad, facilitando su labor diaria a la hora de tomar decisiones.

Uno de los elementos más utilizados en el ámbito clínico, es el análisis de imágenes médicas. Estas, obtenidas mediante diversas técnicas como radiografías, resonancias magnéticas o tomografías, representan una fuente de información esencial para el diagnóstico y seguimiento de patologías. Por ello, técnicas de *Visión Artificial* cobran gran importancia.

1.2. Motivación

Dentro de los enfoques más efectivos y ampliamente utilizados en la actualidad se encuentra el Aprendizaje Profundo (*Deep Learning*), una rama del Aprendizaje Automático (*Machine Learning*), que usa un gran número de capas para intentar aprender diferentes niveles de abstracción de los datos. A diferencia de los métodos tradicionales, que requerían una extracción manual de características y un avanzado conocimiento del dominio, el *Deep Learning* permite a las máquinas aprender representaciones directamente a partir de los datos en bruto [19], como por ejemplo los píxeles en el caso de la clasificación de imágenes.

Una de las arquitecturas más utilizadas en el aprendizaje profundo de procesamiento de imágenes son las Redes Neuronales Convolucionales (*Convolutional Neural Networks, CNN*). Están diseñadas para trabajar con datos en varias dimensiones, como las imágenes. Se caracterizan por su capacidad de detectar patrones o relaciones locales mediante los núcleos o *kernels* que se aplican a la imagen. Gracias a esto, las CNN han demostrado tener un gran rendimiento en tareas de *Computer Vision* como clasificación de imágenes. Desde la aparición de sus primeros modelos como el AlexNet, que ganó el desafío de ImagenNet en 2012 con una amplia ventaja, han surgido una gran cantidad de variantes como VGGNet, ResNet o DenseNet, que han ido mejorando la precisión y eficiencia de los modelos [20].

A pesar de los grandes resultados obtenidos por este tipo de redes, se han seguido buscando nuevas alternativas que puedan superar algunas de sus limitaciones. Una de las más destacadas y con gran protagonismo estos últimos años, son los Transformers. Desde su aparición en 2017 con *Attention is all you need* [3], revolucionaron totalmente el campo del Procesamiento del Lenguaje Natural, llegando a nuestros días modelos a gran escala y con capacidades que se creían imposibles como ChatGPT, Copilot o DeepSeek entre otros.

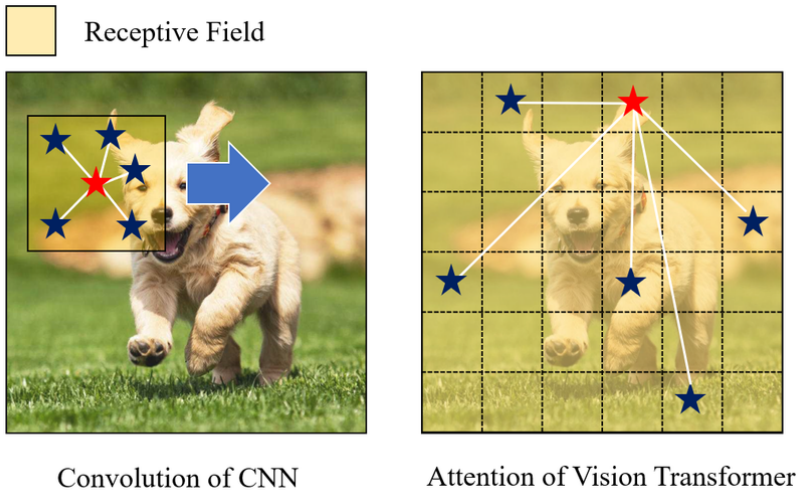


Figura 1.1: Comparativa entre la arquitectura CNN y ViT de [1].

Aunque estos modelos están inicialmente orientados al Procesamiento del Lenguaje Natural, su arquitectura fue adaptada para el ámbito de la visión mediante los llamados *Visual Transformers (ViT)*. Estos han demostrado ser capaces de igualar, e incluso superar en ciertos casos, el rendimiento de las CNN.

La principal ventaja de los ViT es su capacidad para ser altamente paralelizables, lo que permite entrenarlos de forma mucho más eficiente. Tratan toda la imagen en paralelo desde el inicio gracias al mecanismo de atención. Este mismo les otorga la capacidad de modelar relaciones globales entre distintas regiones de la imagen desde las primeras capas [6]. Esto contrasta con las CNN que, por su diseño, procesan la información de manera jerárquica y secuencial, donde se centran en relaciones locales, como se puede ver en la figura1.1.

Teniendo en cuenta el potencial de los Vision Transformers para superar a las Redes Convolucionales en tareas de clasificación de imágenes, este TFG se plantea como un estudio comparativo entre ambas arquitecturas. Para ello, se utilizarán tres conjuntos de imágenes médicas de TFGs anteriores [15, 16, 17], en los que se aplicaron Redes Convolucionales. Se pretende evaluar el rendimiento de los modelos ViT sobre estos mismos datasets y analizar si son capaces de igualar o mejorar los resultados obtenidos previamente y, si fuese posible, identificar en qué casos presentan ventajas respecto a las CNN.

1.3. Objetivos

Dado que este Trabajo de Fin de Grado se centra en la investigación de la adaptación de los Transformers a tareas de clasificación de imágenes (en este caso, médicas), así como su comparación con arquitecturas de Aprendizaje Profundo más clásicas como las CNN, se presentan los siguientes objetivos:

- Obtener y preparar adecuadamente los conjuntos de datos, asegurando su correcto uso tanto para realizar la clasificación, como para permitir la comparación objetiva entre los modelos.
- Construir prototipos basados en Transformers para cada conjunto de datos, con una tasa de acierto aceptable, e intentando mejorar los resultados obtenidos previamente con redes convolucionales.
- Analizar y comparar de manera rigurosa los resultados obtenidos por los modelos *Transformer* frente a los de arquitecturas CNN, considerando diferentes métricas en la medida de lo posible.
- Implementar técnicas de explicabilidad visual, con el fin de interpretar las decisiones del modelo y facilitar la comprensión de su funcionamiento interno.
- Desarrollar una aplicación web que integre todos los modelos construidos y que facilite la visualización de su rendimiento y resultados.

1.4. Estructura de la memoria

Este documento se estructura de la siguiente forma:

Capítulo 1 Introducción. Se presentan el contexto general del proyecto, su motivación, los objetivos principales y la estructura del documento.

Capítulo 2 Gestión del Proyecto. Se explica la metodología de trabajo utilizada, los entregables y la organización temporal del proyecto.

Capítulo 3 Fundamento teórico. Se desarrollan en detalle las bases teóricas necesarias para el proyecto, desde la estructura del Transformer original, hasta su adaptación a la visión por computador con los ViT, incluyendo también técnicas de explicabilidad visual aplicables a este tipo de modelos.

Capítulo 4 Marco de trabajo. Se detallan los recursos hardware y software utilizados a lo largo del desarrollo, justificando su elección.

Capítulo 5 Conjuntos de datos. Se describen los conjuntos de imágenes médicas empleados, su origen, estructura, preprocesamiento y transformaciones aplicadas, obtención y uso.

Capítulo 6 Construcción de los modelos. Se detalla el proceso de desarrollo, entrenamiento y validación de los modelos basados en *Transformers*, además de los criterios seguidos para su ajuste y evaluación.

Capítulo 7 Resultados. Se exponen los resultados obtenidos por los modelos en cada conjunto de datos y se realiza una comparación con los algoritmos basados en CNN correspondientes, analizando su rendimiento.

Capítulo 8 Aplicación. Se describe el desarrollo de la aplicación web que permite probar los modelos construidos de forma sencilla y sin conocimientos técnicos.

Capítulo 9 Conclusiones. Se resumen las aportaciones del trabajo, se reflexiona sobre los resultados obtenidos y se proponen posibles líneas de mejora y desarrollo futuro.

Capítulo 2

Gestión del Proyecto

Debido a su complejidad y extensión, este Trabajo de Fin de Grado se considera como un proyecto, y, por tanto, resulta necesario llevar a cabo una adecuada planificación y gestión del mismo que permita organizar las tareas, los recursos y, con mayor importancia, los tiempos de ejecución de manera eficiente. Esta forma garantiza la correcta organización a lo largo de las diferentes fases del proyecto.

2.1. Metodología

Para poder seleccionar aquella que se adecue a las necesidades de este trabajo, primero es imprescindible comprender el carácter experimental del mismo.

Si bien existen una gran cantidad de metodologías tradicionales ampliamente utilizadas para la gestión de proyectos, éstas están más orientadas al desarrollo del software. Sin embargo, al tratarse de un proyecto de Ciencias de Datos, no es correcto seguir este tipo de metodologías, pues sus ciclos de vida no se adecuan a las necesidades del trabajo.

Por tanto, se ha optado por utilizar la metodología *CRISP-DM* (Cross Industry Standard Process for Data Mining). Si bien existen otras como *KDD* (Knowledge Discovery in Databases) o *SEMMA* (Sample, Explore, Modify, Model, Assess), *CRISP-DM* es una de las más reconocidas en el ámbito de la Minería de Datos y el Aprendizaje Automático. Está específicamente pensada para proyectos de Análisis de Datos como el presente.

CRISP-DM proporciona un marco de trabajo flexible y bien definido, que se adapta al ciclo de vida de un proyecto de Ciencia de Datos, desde la comprensión del problema hasta la evaluación de los resultados [21]. El hecho de que este proyecto no cuente con un equipo de trabajo compuesto por varias personas, dificulta la aplicación de metodologías ágiles, dejando como opción más adecuada el desarrollo incremental como flujo de trabajo.

Figura 2.1: Ciclo de vida de *CRISP-DM* de [2].

Como se puede notar en la figura 2.1, consta de 6 etapas [22]:

1. **Comprensión del negocio (*Business Understanding*)** La fase inicial se centra en entender los objetivos y requisitos del proyecto desde una perspectiva empresarial, con el fin de traducirlos a un problema de Minería de Datos y, con ello, diseñar un proyecto preliminar capaz de alcanzar dichos objetivos.

2. **Comprensión de los datos (*Data Understanding*)** Comienza con la recolección de los datos. Prosigue con una serie de actividades para familiarizarse con los datos, evaluar su calidad y formular ciertas hipótesis útiles sobre la información que contienen. Esta fase está estrechamente relacionada con la anterior, ya que sin una comprensión adecuada de los datos, es prácticamente imposible formular un buen problema de Minería de Datos.

3. **Preparación de los datos (*Data Preparation*)** En esta fase se realizan todas las actividades para construir el conjunto de datos final a partir del original. Este tipo de

actividades como transformaciones, creación de nuevos atributos, limpieza o selección de atributos se suelen realizar en múltiples ocasiones y sin un orden estricto.

4. **Modelado (*Modeling*)** Se seleccionan y aplican técnicas de modelado adecuadas, optimizando los valores de sus parámetros. Dado que ciertas técnicas requieren formatos de datos específicos, y que durante esta fase pueden detectarse errores en los datos u obtener ideas para la creación de nuevos, está muy ligada con la anterior.
5. **Evaluación (*Evaluation*)** En este punto, se ha conseguido obtener uno o más modelos considerados de alta calidad. Antes de proceder a su despliegue, es de vital importancia evaluarlos y comprobar si se han cumplido todos los objetivos definidos. Al final de esta fase, se tiene que llegar a una decisión del uso de los resultados obtenidos.
6. **Despliegue (*Deployment*)** La creación del modelo generalmente no se considera el final del proyecto. La información obtenida debe organizarse y presentarse de forma que sea accesible y útil para el usuario final. Dependiendo de los requisitos, esta fase puede ser muy sencilla o verdaderamente compleja. En cualquier caso, es crucial comprender qué acciones debe realizarse para poder hacer un buen uso de los modelos creados.

2.2. Entregables

Con el desarrollo incremental como flujo de trabajo seleccionado, cada entregable deberá ser completamente funcional y dependerá de la finalización del anterior. De este modo, se garantiza un avance gradual y estructurado del proyecto, permitiendo revisar, evaluar, y mejorar cada fase antes de continuar con la siguiente.

- **Entregable 1:** obtención conjuntos de datos y su adecuada transformación.
- **Entregable 2:** desarrollo de un clasificador básico basado en Transformers, adaptable a cada conjunto de datos.
- **Entregable 3:** optimización específica de cada modelo para su correspondiente conjunto de datos.
- **Entregable 4:** generación de mapas de saliencia para interpretar las decisiones del modelo.
- **Entregable 5:** despliegue de aplicación web funcional que integre todos los modelos desarrollados.
- **Entregable 6:** redacción de la memoria del proyecto.

2.3. Planificación

Dado que este Trabajo de Fin de Grado cuenta con una carga de 12 créditos ECTS dentro del Grado en Ingeniería Informática de la Universidad de Valladolid, se estima una duración de 300 horas, considerando que un *European Credit Transfer System* equivale a 25 horas.

El desarrollo del proyecto comenzó el 17 de marzo de 2025, con una duración prevista de aproximadamente tres meses, finalizando a mediados de junio del mismo año.

2.3.1. Estimación inicial del coste

En la siguiente tabla se presenta una estimación del coste en horas para cada una de las fases de la metodología utilizada. Asimismo, a cada una de ellas se le ha asignado tareas concretas, recursos y tiempos estimados, con el objetivo de facilitar el desarrollo del trabajo y garantizar el cumplimiento de los objetivos planteados.

Fase CRISP-DM	**Tareas principales**	**Duración tarea (h)**	**Total fase (h)**
1. Comprensión del negocio	Contextualización con los TFGs anteriores	10	35
	Revisión del estado del arte	20	
	Definición de los objetivos	5	
2. Comprensión de los datos	Análisis de los datasets usados en los TFGs anteriores	5	5
3. Preparación de los datos	Descarga y estructuración	5	30
	Transformaciones adecuadas	10	
	Organización y almacenamiento para su uso en modelos	15	
4. Modelado	Implementación de ViT básico	20	85
	Estudio de mejoras y variantes para cada conjunto de datos	20	
	Entrenamiento y optimización	30	
	Regularización y optimización del entrenamiento	15	
5. Evaluación	Creación de mapas para explicabilidad	5	15
	Comparación métrica CNN vs ViT	5	
	Comparación de explicabilidad	2	
	Interpretación de resultados	3	
6. Despliegue	Desarrollo de la aplicación web	30	40
	Esquemas de diseño	10	
7. Documentación	Redacción de memoria	75	90
	Creación de gráficas, tablas y resultados	5	
	Revisión completa	10	
		Total	**300**

Tabla 2.1: Planificación temporal del proyecto según metodología CRISP-DM.

Aparte de la estimación de costes recogida en la tabla anterior, en la Figura 2.2 y la Figura 2.3 se representa visualmente el flujo temporal del proyecto a lo largo de sus tres meses de duración. Este diagrama tiene un carácter orientativo y no debe interpretarse como una secuencia estricta de ejecución.

En lugar de un desarrollo secuencial y rígido, el proyecto se ha estructurado siguiendo un enfoque más flexible. Algunas tareas, como la implementación de modelos o el ajuste de parámetros, se han solapado con otras fases, como el desarrollo de la aplicación web o la redacción del documento, tal y como se muestra en la Figura 2.4, donde se ve el flujo completo. Este solapamiento ha permitido aprovechar al máximo el tiempo disponible durante fases computacionalmente costosas, como el entrenamiento de modelos, lo que ha facilitado avanzar en paralelo con otras tareas.

Aunque la planificación establecida sirve como referencia para la organización del trabajo, en la práctica se ha adaptado constantemente en función del progreso real, de la aparición de nuevas ideas y del tiempo efectivo disponible semana a semana. Este enfoque ha permitido mantener un ritmo de trabajo constante, sin dejar de lado los objetivos marcados.

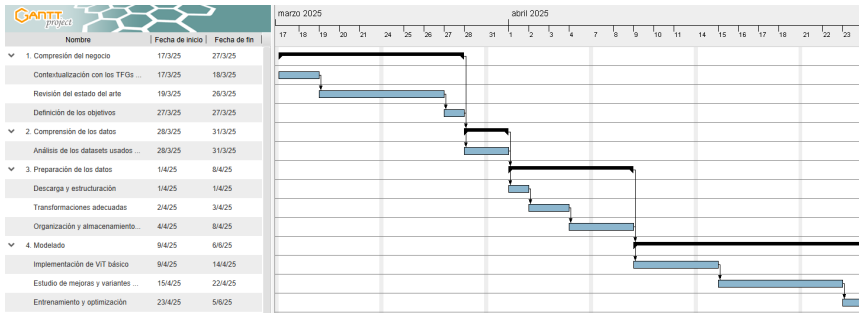


Figura 2.2: Primera parte del Diagrama de Gantt del proyecto.

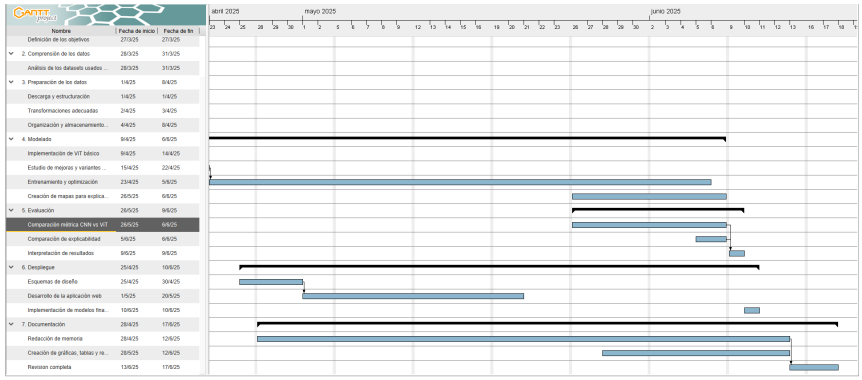


Figura 2.3: Segunda parte del Diagrama de Gantt del proyecto.

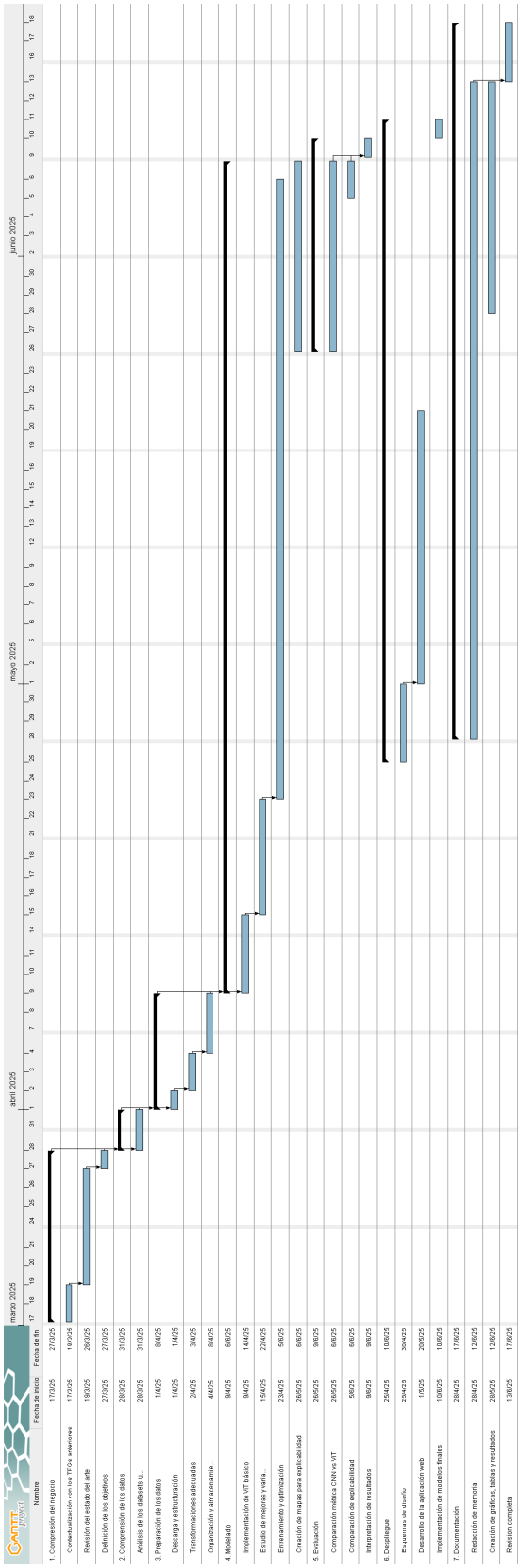


Figura 2.4: Diagrama de Gantt del proyecto al completo.

2.3.2. Variaciones en la planificación inicial

En la práctica, se han producido ciertas complicaciones respecto a los tiempos estimados. Estas variaciones se deben a tres principales razones:

En primer lugar, el desarrollo de este Trabajo de Fin de Grado ha implicado el entrenamiento de tres modelos independientes, cada uno adaptado a un conjunto de datos distinto. Esto ha supuesto un incremento considerable en los tiempos de ejecución, ya que cada modelo requiere múltiples iteraciones para alcanzar un rendimiento óptimo. Aunque se ha contado con dos máquinas para estas tareas, las limitaciones computacionales han seguido representando un cuello de botella significativo.

En segundo lugar, uno de los mayores retos encontrados ha sido el problema de generalización. A lo largo del proceso de entrenamiento, se ha observado un fuerte sobreaprendizaje, lo que ha exigido un gran trabajo de regularización, ajuste de hiperparámetros y pruebas experimentales. Aunque estas tareas estaban contempladas dentro del flujo de trabajo, su complejidad ha superado lo esperado, extendiendo la duración de esta fase.

Por otro lado, durante los dos primeros meses de desarrollo del proyecto, la carga de prácticas externas en empresa redujo la disponibilidad horaria. Esta situación redujo la dedicación semanal al proyecto, especialmente en fases iniciales clave como la preparación de datos y la primera implementación de los modelos base.

En conjunto, todos estos factores han llevado a una adaptación de la planificación, posponiendo ciertas tareas y alargando otras. El enfoque incremental seguido, junto con la flexibilidad de la metodología empleada, ha permitido reajustar la carga de trabajo en función del avance real, garantizando la finalización del proyecto.

Capítulo 3

Fundamento Teórico

Un Transformer es una arquitectura de Aprendizaje Profundo que, transforma una secuencia de entrada en una salida, optimizando la probabilidad de que esta sea coherente con los patrones aprendidos durante el entrenamiento. A diferencia de modelos anteriores, el Transformer permite procesar todos los elementos de la secuencia en paralelo, sin necesidad de mantener un orden explícito. Esta capacidad proviene de su componente principal: el mecanismo de atención, que hace posible que cada elemento de la entrada decida qué otros elementos son más relevantes para generar su salida.

Como ya se ha mencionado en la Introducción 1, los Transformers fueron presentados por primera vez en el artículo *Attention is All You Need* [3], desarrollado por investigadores de Google en 2017. La estructura original propuesta se enmarca como una instancia específica de los modelos *encoder-decoder*, en la que un codificador procesa la entrada y un decodificador genera la salida correspondiente. Desde entonces, han surgido numerosas variantes que permiten adaptar esta arquitectura a tareas muy diversas.

A continuación, se explicará de forma general la estructura base de los Transformers, con el objetivo de comprender sus fundamentos. Posteriormente, se profundizará en su adaptación al caso de los *Vision Transformers (ViT)*, diseñados específicamente para el procesamiento de imágenes.

3.1. Estructura principal

La estructura del Transformer se divide en dos grandes componentes: el codificador (*encoder*) y el decodificador (*decoder*), representados en las mitades izquierda y derecha de la Figura 3.1.

Antes de ser procesada por el codificador, la secuencia de símbolos de entrada se convierte en una secuencia de vectores densos mediante una capa de *embedding*, a los que se suma una

codificación posicional, pues el Transformer no tiene acceso explícito al orden de los elementos de la entrada al procesar los símbolos en paralelo.

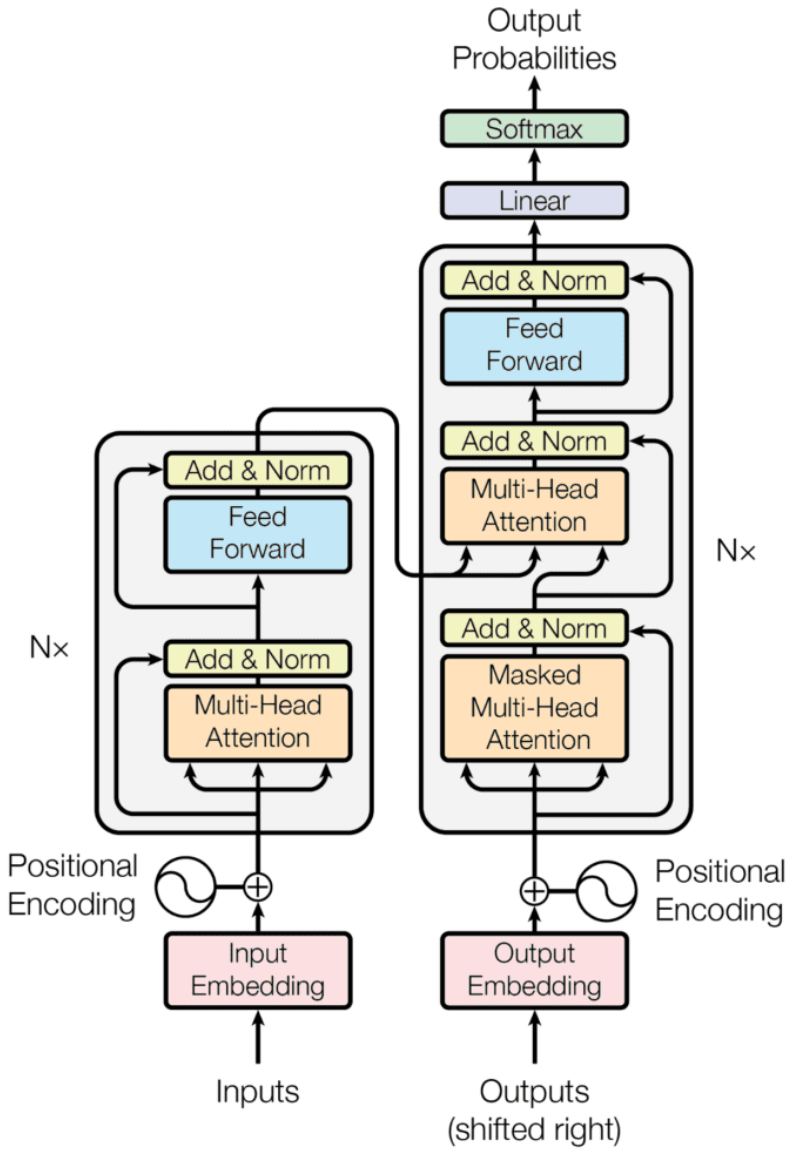


Figura 3.1: Estrucutra de un Transformer de [3].

Esta representación combinada $(x_1, ..., x_n)$ es la que recibe el codificador, el cual transforma en una nueva secuencia de representaciones numéricas continuas $\mathbf{z} = (z_1, ..., z_n)$. Esto almacena información contextual y semántica de toda la secuencia, permitiendo que cada z_i

incorpore no sólo el significado del símbolo correspondiente, sino también cómo se relaciona con el resto de elementos de la secuencia.

A partir de $\mathbf{z}$, el decodificador genera la secuencia de salida $(y_1, ..., y_n)$ de forma autorregresiva, produciendo un elemento en cada paso.

Un modelo autorregresivo genera cada elemento de la secuencia de salida condicionado a los elementos generados previamente. Es decir, en el paso t, el modelo produce y_t utilizando como entrada tanto la representación codificada de la entrada como la secuencia parcial $(y_1, ..., y_{t-1})$ generada hasta ese momento. De esta forma, se entrena al modelo para maximizar la probabilidad conjunta de la secuencia como producto de probabilidades condicionales:

$$P(y_1, y_2, ..., y_n \mid \mathbf{z}) = \prod_{t=1}^{n} P(y_t \mid y_1, ..., y_{t-1}, \mathbf{z})$$

Este enfoque permite al modelo construir salidas de manera coherente, ya que cada nuevo símbolo tiene en cuenta tanto el contexto global (extraído del codificador) como el contexto local (la salida generada hasta el momento). Durante el entrenamiento, el modelo recibe como entrada la secuencia real completa (técnica conocida como *teacher forcing*), mientras que en inferencia utiliza sus propias predicciones anteriores, lo que puede introducir errores acumulativos [23].

Este principio autorregresivo es de gran importancia y constituye una de las bases conceptuales del modelo original Transformer [3].

Estos símbolos (de aquí en adelante se denominarán *tokens*) corresponden a sub-palabras en la arquitectura original, ya que en un principio los Transformers estaban pensados para el Procesamiento de Lenguaje Natural. En el caso de los ViT, como se analizará más adelante, los tokens representan fragmentos o regiones de una imagen.

El Transformer sigue una arquitectura general basada en múltiples capas apiladas, donde cada capa consta de un mecanismo de atención (*attention*) y una capa de red neuronal completamente conectada (*feed-forward*). Cada uno de estos bloques se encuentra rodeado por una conexión residual seguida de una operación de normalización (*Add & Norm*). La característica fundamental es que cada token navega de manera paralela a través de las diferentes capas, siguiendo su propio camino, aunque cada uno depende directamente de todos los demás elementos de la secuencia.

Estas capas están conectadas punto a punto, tanto en el codificador como en el decodificador, como se muestra en la Figura 3.2, que representa la arquitectura original con seis capas de codificación.

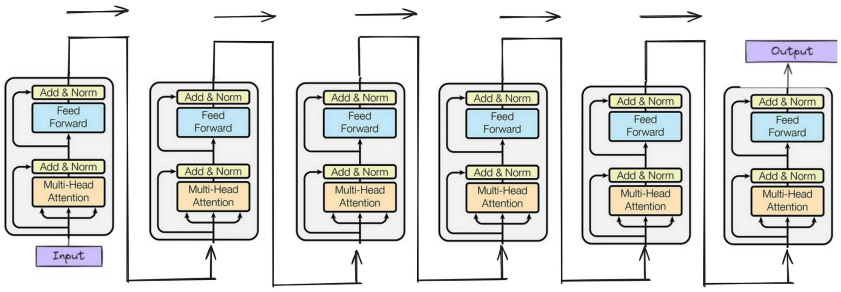


Figura 3.2: Navegación por capas del codificador de [4].

Finalmente, la representación generada en cada paso del decodificador se proyecta mediante una capa lineal sobre el espacio del vocabulario y pasa por una función *softmax*, que convierte ese vector en una distribución de probabilidad. Este proceso puede observarse en la parte derecha de la Figura 3.1, donde se muestra la conexión entre la salida del decodificador y la capa de predicción final.

Como se detallará más adelante, durante el entrenamiento, se suele emplear la pérdida de entropía cruzada para comparar esta distribución con el símbolo real esperado en cada paso de la secuencia.

3.1.1. Embedding

Antes de que un Transformer pueda procesar datos de entrada, estos deben transformarse en una representación numérica densa que el modelo sea capaz de manejar. Esta transformación se realiza mediante una capa de *embedding*, que convierte elementos discretos, como palabras o subpalabras, en vectores de dimensión fija dentro de un espacio continuo.

Un *embedding* es, por tanto, una técnica para representar tokens de forma densa y significativa. En lugar de trabajar con índices enteros que no contienen información semántica, se asigna a cada token un vector que captura relaciones de similitud y contexto. Por ejemplo, supongamos que queremos representar las palabras “perro”, “gato” y “coche” en un espacio de cuatro dimensiones. Un embedding podría asignarles los siguientes vectores:

$$\begin{aligned} \text{perro} &\rightarrow [0{,}35,\ 0{,}10,\ -0{,}22,\ 0{,}58] \\ \text{gato} &\rightarrow [0{,}33,\ 0{,}12,\ -0{,}20,\ 0{,}60] \\ \text{coche} &\rightarrow [-0{,}75,\ 0{,}90,\ 0{,}10,\ -0{,}30] \end{aligned}$$

En este espacio vectorial, la cercanía entre los vectores de “perro” y “gato” refleja su relación semántica, mientras que “coche” se encuentra más alejado, indicando una menor similitud con los anteriores. Aunque los valores de los vectores pueden parecer arbitrarios, estos son vectores entrenables que se ajustan durante el entrenamiento del modelo, permitiendo capturar relaciones complejas entre palabras o sub-palabras.

Otro ejemplo visual puede verse en la Figura 3.3, donde se representan distintos tokens correspondientes a animales y vehículos en un espacio bidimensional. La posición de los puntos refleja cómo los embeddings capturan las relaciones semánticas: los elementos del mismo grupo tienden a agruparse, lo que indica que el modelo ha aprendido a asociarlos por su significado (NO ME ACABA DE CONVENCER LA IMAGEN).

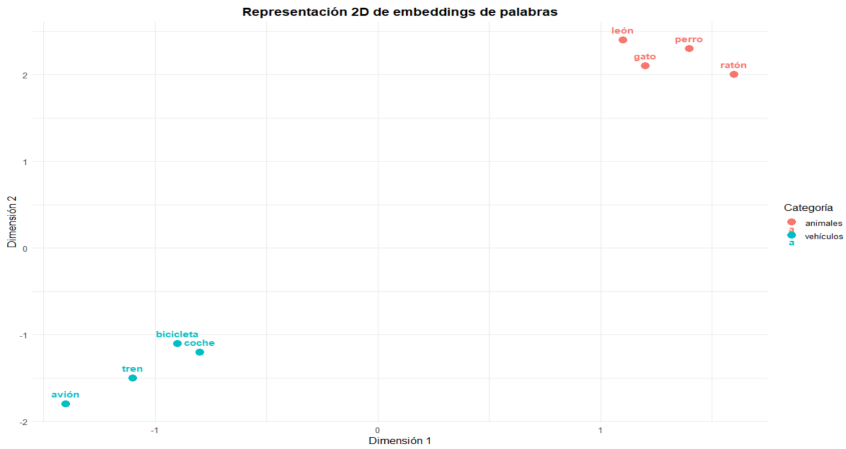


Figura 3.3: Representación bidimensional de embeddings de palabras correspondientes a dos grupos semánticos: animales y vehículos.

Codificación posicional

Una de las características del Transformer es que, al no tratase de una arquitectura secuencial como las CNN, no tiene conocimiento del orden en que aparecen los tokens. Para proporcionar esta información estructural, se añade a cada embedding de palabra un *positional encoding*, que codifica la posición del token en la secuencia.

En el trabajo original de Vaswani et al. [3], se propuso un esquema de codificación posicional fija, basado en funciones sinusoidales de diferentes frecuencias. Las fórmulas empleadas son las siguientes:

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{\frac{2i}{d_{\text{model}}}}}\right), \quad PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{\frac{2i}{d_{\text{model}}}}}\right)$$

donde pos es la posición del token y i es la dimensión dentro del vector de embedding. Estas funciones están diseñadas de tal forma, que el modelo pueda aprender fácilmente las posiciones relativas entre tokens. En concreto, permiten que un desplazamiento fijo k en la secuencia se represente mediante una combinación lineal de las codificaciones anteriores, lo que facilita el modelado de relaciones como la dependencia gramatical o sintáctica.

Estas codificaciones se visualizan claramente en la Figura 3.4, donde cada fila corresponde a una posición en la secuencia y cada columna representa una dimensión del embedding.

Las variaciones periódicas muestran cómo las funciones sinusoidales, al contar con múltiples frecuencias, marcan patrones posicionales desde el nivel local (ondulación rápida) hasta el global (ondulación suave)[5].

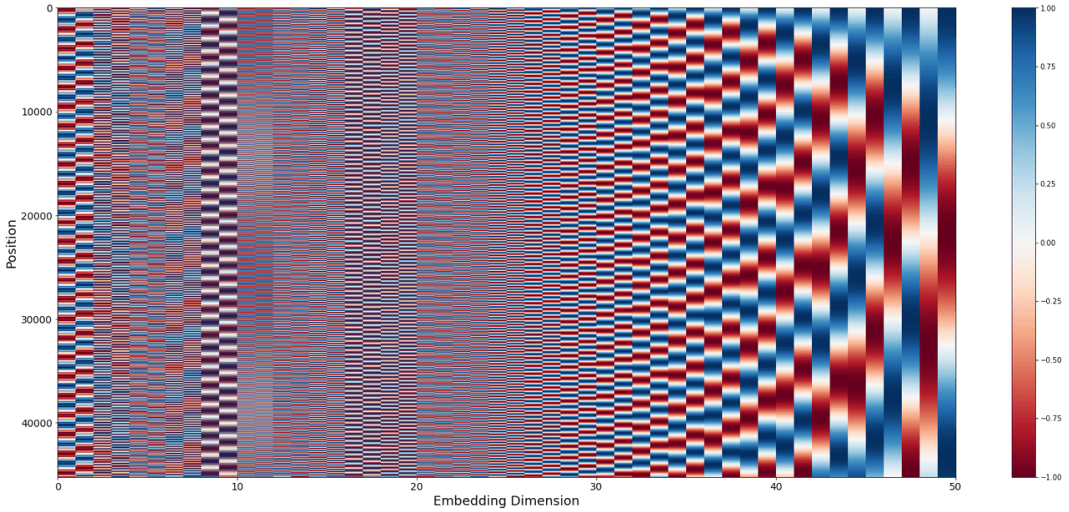


Figura 3.4: Codificación posicional sinusoidal: valores de sin y cos según la posición y dimensión del embedding de [5].

3.1.2. Mecanismo de Atención

Es el componente central de la arquitectura Transformer. Su objetivo es que el modelo, a la hora de generar las representaciones internas, pueda asignar diferentes niveles de importancia a cada elemento de la secuencia de entrada en función del contexto. Permite considerar simultáneamente todas las posiciones de la secuencia, lo que facilita la paralelización y una mejor captura de las dependencias más generales.

Una función de atención puede describirse como el mapeo entre una consulta (*query*) y un conjunto de pares clave-valor (*key-value*) a una salida, donde consultas, claves, valores y salida son todos vectores. Esta salida se obtiene como una combinación ponderada de los valores, en la que los pesos se calculan a partir de una función de compatibilidad entre la consulta y las claves.

En este contexto, los vectores utilizados se representan comúnmente de la siguiente forma:

- **q (*query*)**: representa la consulta que compara contra otros elementos.
- **k (*key*)**: representa las claves con las que se evalúa la similitud de cada consulta.
- **v (*value*)**: contiene la información asociada a cada clave y es lo que se combina (ponderadamente) para generar la salida.

La atención calcula una puntuación de similitud entre cada par consulta-clave, que luego se utiliza para obtener una combinación ponderada de los valores, dando lugar a una representación contextualizada para cada token de entrada.

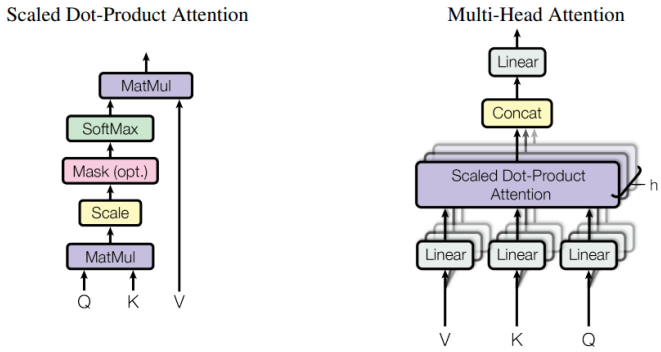


Figura 3.5: Mecanismo de Atención de [3]. (izq) Scaled Dot-Product Attention. (der) Multi-Head Attention .

Scaled Dot-Product Attention

Vaswani et al. [3] introdujeron un mecanismo de atención particular denominado *Scaled Dot-Product Attention* (Figura 3.5), que lleva al modelo a evaluar y a asignar diferentes niveles de importancia a cada elemento de la secuencia de entrada.

La entrada a este mecanismo consiste en vectores de consulta y clave de dimensión d_k, y vectores de valor de dimensión d_v. Para cada consulta, se calcula su producto escalar con todas las claves, se divide por $\sqrt{d_k}$, a lo que se aplica la función *softmax* para obtener los pesos con los que se combinarán los valores correspondientes.

Dado que en la práctica se requiere calcular la atención sobre múltiples consultas de forma simultánea, es más eficiente llevar a cabo los cálculos de manera matricial en lugar de vectorial. Para ello, se utilizan las matrices Q, K, V, que agrupan respectivamente todos los vectores de consulta, clave y valor. La fórmula completa del mecanismo es:

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V$$

Aunque actualmente, esta función es la más reconocida y empleada dentro de este ámbito, anteriormente las dos más usadas eran la atención aditiva y la multiplicativa sin escalado. Esta última es idéntica a la atención escalada, salvo por la ausencia del factor $\frac{1}{\sqrt{d_k}}$. Si bien ambas son equivalentes en complejidad computacional teórica, la multiplicativa es más eficiente en la práctica debido a su implementación mediante operaciones de multiplicación de matrices altamente optimizadas.

Para valores pequeños de d_k, ambas versiones se comportan de forma similar y se pueden usar indistintamente. Sin embargo, cuando d_k es grande, la atención aditiva supera a la multiplicativa sin escalado [24]. Esto se debe a que al realizar cálculos con dimensión de claves muy grande (d_k), el producto entre Q y K puede dar valores muy altos, lo que implica que la función *softmax* produzca gradientes extremadamente pequeños, es decir, devuelva casi ceros y unos. Por ello, para contrarrestar este efecto, se escalan los productos escalares dividiéndolos por $\sqrt{d_k}$.

Una manera de ver por qué se pueden llegar a dar valores altos al realizar los productos escalares es desde una perspectiva estadística: si se asume que los componentes q y k son variables independientes con media 0 y varianza 1, entonces su producto escalar $q \cdot k = \sum_{i=1}^{d_k} q_i k$, tendrá media 0 y varianza d_k. Por tanto, al dividir dicho producto por $\sqrt{d_k}$, la varianza pasa a ser 1, estabilizando los valores.

En trabajos recientes se han explorado otras variantes de atención con propiedades complementarias. Por ejemplo, las denominadas *symmetric attention* y *pairwise attention* proponen enfoques alternativos para el cálculo de similitudes entre tokens, a menudo con propiedades teóricas deseables como simetría, interpretabilidad o eficiencia computacional. Estas alternativas han sido estudiadas, entre otros, por Courtois et al. [25].

Multi-Head Attention

Como ya se ha visto, el mecanismo de atención permite establecer relaciones directas entre distintos tokens de una secuencia de entrada, modelando la importancia que tienen entre ellos [26]. Pero, Vaswani et al. [3] demostraron que es beneficioso emplear múltiples funciones de atención en paralelo, lo que da lugar al mecanismo conocido como *Multi-Head Attention*.

Este diseño permite que el modelo atienda a diferentes representaciones subespaciales de la información en paralelo, capturando así múltiples contextos o relaciones semánticas. En contraste, el uso de una única cabeza limitaría la capacidad del modelo para representar patrones diversos, al forzar una media aritmética sobre todas las interacciones.

Formalmente, se realizan proyecciones lineales sobre las matrices de consultas, claves y valores utilizando distintos pesos entrenables para cada cabeza de atención. En particular, para la i-ésima ($i \in \{1, ..., h\}$), se realizan las siguientes proyecciones:

$$Q_i = QW_i^Q, \quad K_i = KW_i^K, \quad V_i = VW_i^V$$

La salida de cada cabeza se calcula como:

$$\text{head}_i = \text{Attention}(Q_i, K_i, V_i)$$

Las salidas de todas ellas se concatenan y se proyectan con una matriz final:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, ..., \text{head}_h)W^O$$

donde $W_i^Q, W_i^K, W_i^V \in \mathbb{R}^{d_{\text{model}} \times d_k}$ y $W^O \in \mathbb{R}^{hd_v \times d_{\text{model}}}$ son parámetros entrenables.

Este proceso se representa en la figura 3.5, en la parte derecha del diagrama.

Es gracias a este diseño que los Transformers sean capaces de modelar muchas relaciones complejas entre tokens, tanto a corto como a largo alcance, e incluso centradas en distintos aspectos semánticos o espaciales (como sería el caso de este trabajo).

Self-Attention

El mecanismo de auto-atención (*self-attention*) es una forma particular de atención en la que las consultas (Q), las claves (K) y los valores (V) provienen de la misma secuencia de entrada. Es decir, cada token de la secuencia puede "atender" a todos los demás (incluyéndose a sí mismo), permitiendo calcular una representación contextualizada basada en toda la secuencia.

Esta capacidad resulta fundamental en los Transformers ya que, con ello, el modelo puede aprender relaciones entre cualquier par de elementos sin tener en cuenta su distancia relativa. Al ponderar la importancia de cada token en relación con los demás, se construyen representaciones capaces de integrar la comprensión global de la secuencia.

En una subcapa de auto-atención, la secuencia de entrada de donde provienen todas las claves, consultas y valores, es la salida de la capa anterior.

- **En el Codificador**: cada posición en la secuencia de entrada puede atender a todas las posiciones anteriores y posteriores, permitiendo una representación rica del contexto.
- **En el Decodificador**: similar al codificador, pero se aplica una máscara para evitar que la posición actual atienda a posiciones futuras, conservando la propiedad auto-regresiva. Esta máscara se aplica en el *Scaled Dot-Product Attention* asignando $-\infty$ a aquellas conexiones consideradas ilegales.
- **Entre Codificador y Decodificador**: el decodificador puede atender a todas las posiciones de la secuencia de entrada (provenientes del codificador), permitiendo que cada paso de generación se base en toda la información.

Ciertas ventajas relevantes de usar *self-attention* nombradas en [3] son:

- **Paralelización**: A diferencia de las RNN, que procesan secuencias de manera secuencial, el *self-attention* puede procesar todos los elementos simultáneamente, aprovechando mejor el cómputo.

- **Captura de Dependencias a Larga Distancia**: Como ya se ha comentado en la introducción, mientras que las RNN tienen dificultades para modelar relaciones entre elementos distantes en una secuencia, el *self-attention* puede capturar estas dependencias sin importar la distancia entre tokens.
- **Eficiencia Computacional**: Comparado con las CNN, el *self-attention* requiere menos operaciones para modelar relaciones entre todos los pares de elementos en una secuencia, especialmente en secuencias largas.

3.1.3. Redes Postion-Wise Feed-Forward

Adicionalmente, cada capa del codificador y del decodificador incorpora una subcapa de red neuronal *feed-forward* completamente conectada, que se aplica de manera independiente a cada token de la secuencia. Esta subcapa consta de dos transformaciones lineales separadas por una función de activación no lineal que, en la arquitectura original, es una $ReLU$ [3]. La operación que se realiza puede expresarse como:

$$\text{FFN}(x) = \text{máx}(0, xW_1 + b_1)W_2 + b_2$$

donde x es el vector de entrada, W_1 y W_2 son matrices de pesos, y b_1 y b_2 son vectores de sesgo.

Aunque esta operación se aplica de forma idéntica a cada token, los parámetros W_1, W_2, b_1 y b_2 son compartidos a lo largo de todos ellos dentro de una misma capa, pero varían de una a otra. Este diseño facilita la paralelización y la eficiencia computacional del modelo.

Investigaciones algo más recientes han mostrado, que estas subcapas *feed-forward* actúan como memorias de tipo clave-valor, en las que cada clave se asocia con patrones específicos y cada valor contribuye a la generación de la salida. Esto implica que las subcapas no sólo transforman las representaciones, sino que también son capaces de almacenar cierta información aprendida durante el entrenamiento [27].

3.1.4. Normalización y conexiones residuales

Cada subcapa del Transformer, tanto la de atención, como la de proyección feed-forward, va acompañada de una conexión residual seguida de una normalización por capas (*Layer Normalization*). Esta combinación se implementa de la siguiente manera:

$$\text{Output} = \text{LayerNorm}(x + \text{Sublayer}(x))$$

Este diseño tiene como objetivo mejorar la estabilidad del entrenamiento. Las conexiones residuales permiten que los gradientes se propaguen más fácilmente hacia capas anteriores, lo que ayuda a mitigar el problema del desvanecimiento del gradiente [28]. Este fenómeno,

conocido como *gradient vanishing*, ocurre cuando los gradientes se vuelven progresivamente más pequeños al retropropagarse, dificultando el aprendizaje en capas lejanas a la salida.

Por su parte, la normalización por capas estabiliza la activación de cada token normalizando sus dimensiones internas. A diferencia de la *Batch Normalization*, que se basa en estadísticas globales del lote (*batch*) y se aplica solo en entrenamiento, la *Layer Normalization* opera sobre cada instancia individual y esta presente tanto en entrenamiento, como en la fase de test. Esta propiedad la hace especialmente adecuada para tareas como las que aborda el Transformer [29].

Es importante destacar que existen alternativas principales respecto al punto en el que se aplica la normalización dentro del bloque Transformer:

- **Post-Normalización (Post-LN)**: La normalización se realiza después de la suma residual, como en la arquitectura original de Vaswani [3].
- **Pre-Normalización (Pre-LN)**: La normalización se aplica antes de la subcapa, lo que ha demostrado mejorar la estabilidad del entrenamiento para ciertos casos específicos.

3.2. Estructura ViT

Aunque los Transformers fueron creados en un primer momento con el objetivo de poder procesar el lenguaje natural, su éxito ha motivado su adaptación a otros dominios.

Los *Vision Transformer* (ViT) representan una adaptación de la arquitectura original al dominio de la Visión Por Computador. En Dosovitskiy et al. [6], se origina la idea de crear un Transformer que se pueda aplicar directamente a las imágenes, sin cambiar en exceso la estructura original.

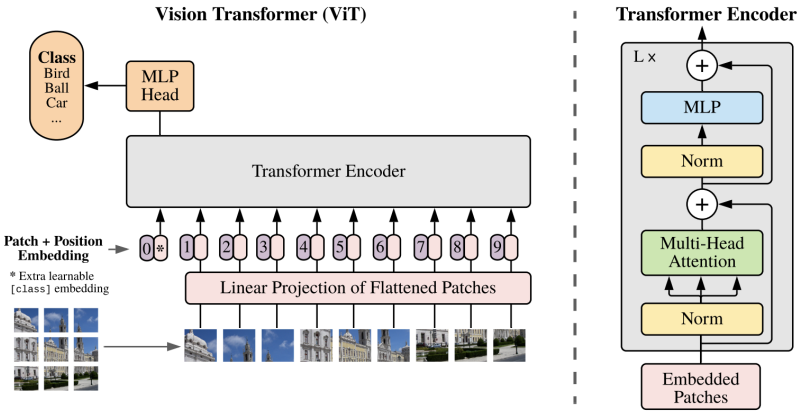


Figura 3.6: Estructura de un ViT de [6].

En ViT, se reemplaza las convoluciones tradicionales de las CNNs por los mecanismos de autoatención. Como se puede ver en la figura 3.6, en lugar de procesar la imagen de entrada como una cuadrícula de píxeles, se divide en una secuencia de *patches* de tamaño fijo (por ejemplo, 16×16 píxeles). Cada parte (*patch*) se aplana y se proyecta linealmente a un espacio de características, similar al proceso de tokenización en Procesamiento de Lenguaje Natural. A estos vectores se les añaden embeddings posicionales para conservar la información espacial. La secuencia resultante se introduce en las múltiples capas de codificador Transformer estándar.

Otra característica distintiva de los ViT, es la inclusión de un token de clasificación (`[CLS]`) al inicio de la secuencia. Este token es el encargado de recoger la información general de la imagen la cual, una vez se ha finalizado, se utiliza para realizar la predicción de clase.

Por tanto, de toda la estructura convencional del Transformer, únicamente se conserva la parte correspondiente al codificador, pero añadiendo a su salida la capa lineal para poder realizar la clasificación de las imágenes. Si se quisiese obtener las probabilidades de cada clase, también se puede añadir una función softmax tras esta capa lineal.

Las ecuaciones (1)–(4) resumen el flujo completo del Vision Transformer. En primer lugar, la imagen se convierte en una secuencia de tokens mediante proyección lineal de patches y adición de un token `[CLS]` junto con sus codificaciones posicionales (Ec.(1)). Esta secuencia atraviesa una pila de bloques Transformer compuestos por autoatención y capas MLP, ambas con normalización y conexiones residuales (Ec.(2)–(3)). Finalmente, se extrae y normaliza el token `[CLS]` para producir la predicción de salida (Ec.(4))[6].

$$z_0 = [x_{\text{class}}; x_p^1\mathbf{E}; \ldots; x_p^N\mathbf{E}] + \mathbf{E}_{\text{pos}}, \quad \mathbf{E} \in \mathbb{R}^{(P^2 \cdot C) \times D}, \quad \mathbf{E}_{\text{pos}} \in \mathbb{R}^{(N+1) \times D} \tag{1}$$

$$z'_\ell = \text{MSA}(\text{LN}(z_{\ell-1})) + z_{\ell-1}, \quad \ell = 1 \ldots L \tag{2}$$

$$z_\ell = \text{MLP}(\text{LN}(z'_\ell)) + z'_\ell, \quad \ell = 1 \ldots L \tag{3}$$

$$y = \text{LN}(z_L^0) \tag{4}$$

3.2.1. Embedding

Como ya se ha descrito antes, el proceso de *embedding* es esencial. En este caso, permite adaptar las imágenes a un formato en el que los ViT puedan aprovechar sus cualidades de paralelización. Este proceso consta de dos componentes principales: el *patch embedding* y el *positional encoding*.

Patch Embedding

El *patch embedding* consiste en dividir la imagen de entrada $x \in \mathbb{R}^{H \times W \times C}$ (altura, anchura y canales) en una cuadrícula de patches 2D no solapados $x_p \in \mathbb{R}^{N \times (P^2 \cdot C)}$, donde (H, W)

corresponde con la resolución de la imagen, C es el número de canales y (P, P) es la resolución de cada uno de los patches. Esto da lugar a $N = \frac{HW}{P^2}$ patches, valor que corresponde con la longitud de secuencia de entrada del Transformer.

Una vez se tiene la imagen dividida, cada patch se aplana en un vector de dimensión $P^2 \cdot C$. A continuación, se utiliza una capa lineal para realizar una proyección a un espacio de dimensión D (Ec. (1)). Este proceso transforma la imagen en una secuencia de vectores de características, similar a una secuencia de tokens en procesamiento de lenguaje natural.

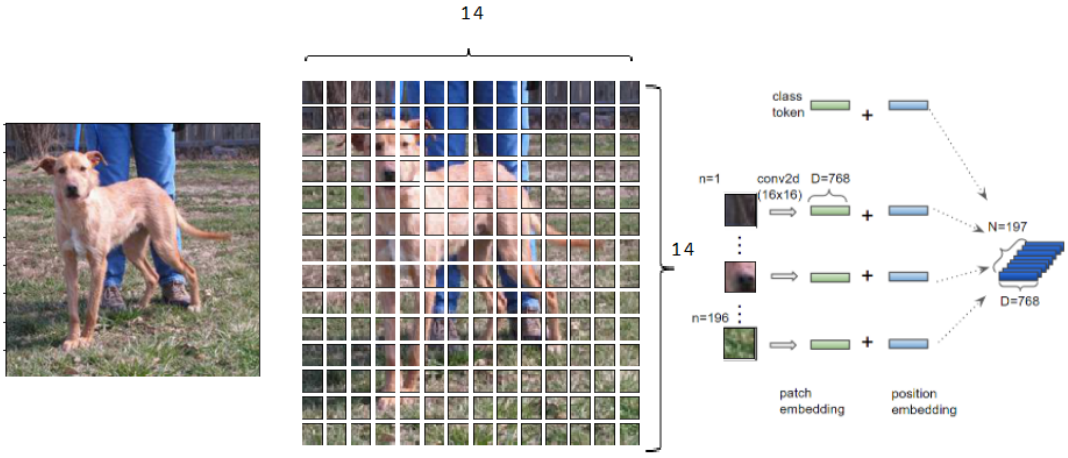


Figura 3.7: Patch embedding de [7].

Positional Encoding

Si bien, en la arquitectura original de los Transformer,s se emplea una codificación posicional determinista basada en funciones sinusoidales, en los Vision Transformers (ViT) se opta habitualmente por una estrategia diferente.

Se añade un *positional embedding* a cada vector de patch. En ViT, estos embeddings posicionales son vectores aprendibles de dimensión D, que se suman a los embeddings de los patches (parte de la derecha de figura 3.7). Esta suma permite al modelo distinguir la posición relativa de cada patch en la imagen, preservando la información espacial crítica.

Estos embeddings aprendibles, son vectores que se optimizan junto con los parámetros del modelo durante el entrenamiento. Esto ha demostrado ser efectivo en tareas de clasificación de imágenes, ya que permite al modelo aprender representaciones espaciales adaptadas a los datos concretos gracias a conocer las posiciones de cada uno de los patches. Sin embargo, a diferencia de las convoluciones, estos embeddings no son invariantes a transformaciones espaciales como la traslación, lo que puede limitar su generalización si no se dispone de una gran cantidad de datos.

Si bien los embeddings posicionales aprendibles suelen inicializarse con valores aleatorios,

existen enfoques alternativos que pueden mejorar la incorporación de la información espacial. Entre ellos se encuentran las codificaciones sinusoidales, los embeddings bidimensionales diseñados para preservar mejor la estructura de la imagen, o técnicas más avanzadas basadas en convoluciones o mecanismos de autoatención local [30]. Aunque cabe destacar que la inicialización específica en la práctica no es habitual, pues generalmente no conlleva mejoras significativas en el rendimiento.

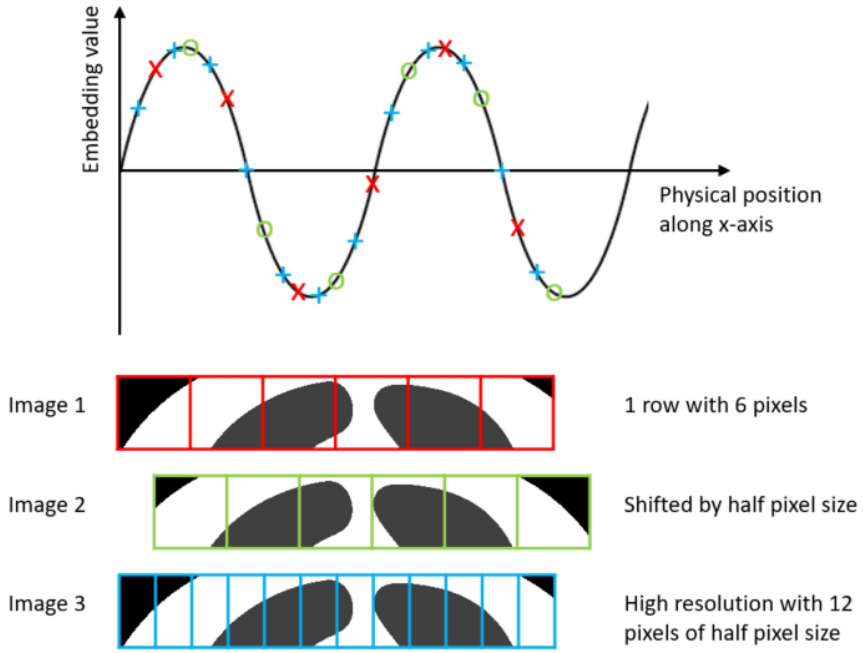


Figura 3.8: Ejemplo visual de distintas resoluciones en codificación posicional: (arriba) rejilla de baja densidad, (medio) desplazamiento fraccional, (abajo) rejilla de alta densidad de [8].

Una cuestión relevante en la codificación posicional es la resolución espacial con la que se representan las posiciones. La Figura 3.8 ilustra cómo distintas configuraciones pueden variar la densidad de puntos que codifican la posición en la imagen. Una mayor resolución permite capturar relaciones locales más precisas entre regiones cercanas, pero también implica un mayor coste computacional. Esta decisión de diseño puede afectar al tipo de información espacial que el modelo puede aprender y generalizar.

3.2.2. Token de Clasificación

El token de clasificación, denotado como `[CLS]`, es una parte esencial que permite al modelo generar una representación global de la imagen para tareas de clasificación. Este token, introducido inicialmente en modelos de procesamiento de lenguaje natural como BERT [31], se adapta en los ViT para resumir la información de todos los patches de la imagen. Se define un vector entrenable de dimensión D que se inserta al inicio de la secuencia de embeddings

de patches. A medida que la secuencia pasa por las capas del codificador Transformer, el token `[CLS]` interactúa con los demás gracias a los mecanismos de autoatención, guardando información del contexto de toda la imagen. Al final del proceso, la representación del token `[CLS]` contiene las características globales de la imagen para poder realizar su clasificación.

Con este planteamiento, el ViT puede realizar predicciones precisas sin necesidad de estructuras adicionales como capas de agrupamiento global, comunes en las redes neuronales convolucionales (CNNs). Estudios recientes han explorado variantes del token `[CLS]`, como el uso de múltiples tokens de clasificación o la modificación de su dimensionalidad para mejorar el rendimiento en tareas específicas [32].

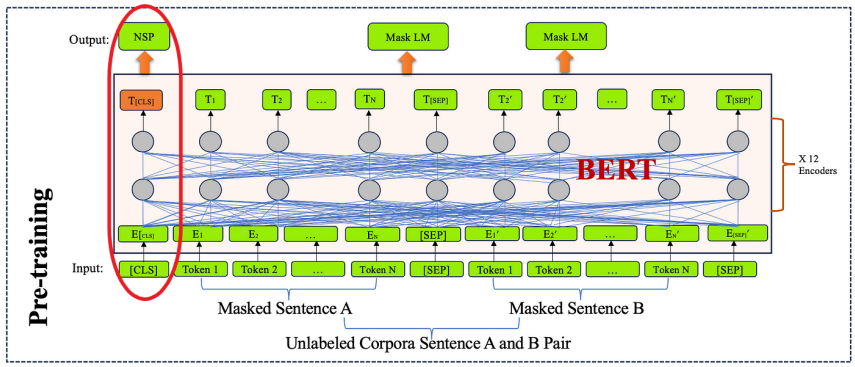


Figura 3.9: Token `[CLS]` en BERT de [9].

Como se puede ver en la figura 3.9 para el caso de BERT, el token se añade al inicio de la secuencia. Se trata como todos los demás durante el entrenamiento, y, tras finalizar, se utiliza para poder realizar la clasificación.

3.2.3. Mean Pooling

Otra estrategia común para obtener una representación global de la imagen es el denominado *mean pooling*. Esta técnica consiste en calcular la media de las salidas de todos los tokens del patch generados por el codificador. Formalmente, si $Z = [z_1, z_2, ..., z_N]$ representa las salidas de los N tokens o patches, la representación global z_{mean} se obtiene como:

$$z_{mean} = \frac{1}{N} \sum_{i=1}^{N} z_i$$

Esta representación z_{mean} se utiliza posteriormente para la clasificación mediante una capa lineal seguida, opcionalmente, de una función softmax para obtener las probabilidades de cada clase, al igual que ocurría con el `[CLS]`.

El *mean pooling* presenta varias ventajas en ciertos contextos. En primer lugar, al promediar las representaciones de todos los patches, se obtiene una visión más global de la imagen, lo que puede mejorar la generalización en tareas donde la información relevante está repartida en diferentes regiones. Esta técnica es invariante a la traslación, ya que no depende de la posición específica de los patches, lo que la hace robusta frente a desplazamientos en la imagen [33].

Sin embargo, también existen desventajas. Al tratar todos los patches con igual importancia, el *mean pooling* puede tener problemas con características importantes que sólo estén presentes en regiones muy específicas de la imagen. Esto es muy importante en casos donde ciertos detalles locales son cruciales para la clasificación. En cambio, el token `[CLS]` puede aprender a enfocarse en estas regiones discriminantes durante el entrenamiento [34].

Debido a estas desventajas, se han explorado variantes del *mean pooling*. Por ejemplo, el *Group Generalized Mean Pooling* (GGeM) divide los canales en grupos y aplica una media generalizada dentro de cada grupo, permitiendo más flexibilidad y adaptación a la información [33].

3.3. Funciones de activación

En los modelos basados en Transformers, las funciones de activación juegan un papel crucial en las redes neuronales *feed-forward*, tanto en la arquitectura original como en sus variantes. Estas funciones introducen no linealidad al modelo, lo que posibilita representar relaciones complejas entre los datos.

Aunque existen numerosas funciones de activación, en el contexto de este trabajo se abordarán las dos más utilizadas: *ReLU*, empleada en la arquitectura original de Vaswani et. al, y *GELU*, adoptada posteriormente en variantes como BERT o los Vision Transformers (ViT). A continuación se describen sus características, comportamiento y diferencias principales.

3.3.1. ReLU (Rectified Linear Unit)

La función *ReLU* es una de las más extendidas en redes neuronales profundas por su simplicidad y eficiencia computacional. Fue introducida por Nair y Hinton en 2010 [35]. Se define como:

$$\text{ReLU}(x) = \text{máx}(0, x)$$

Como puede verse, anula todos los valores negativos y deja pasar los positivos, lo que conlleva una activación dispersa. Su bajo coste computacional y facilidad para mitigar el problema del desvanecimiento del gradiente, han contribuido a su éxito.

En el Transformer original, la función *ReLU* es utilizada en las subcapas *feed-forward* tras la primera proyección lineal. En la Figura 3.10 se puede ver su comportamiento gráfico, donde se observa la activación nula para entradas negativas y lineal para las positivas.

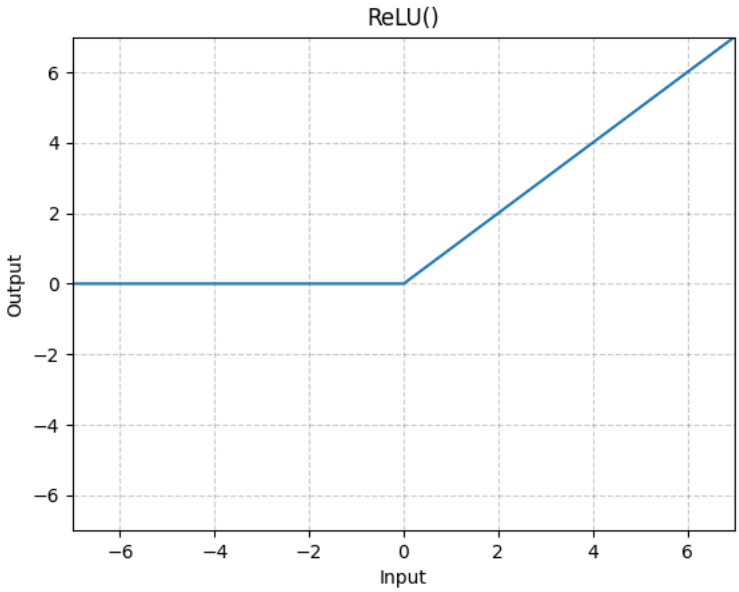


Figura 3.10: Representación gráfica de la función ReLU de [10].

3.3.2. GELU (Gaussian Error Linear Unit)

La función *GELU* ha sido propuesta como alternativa a *ReLU*, especialmente en modelos modernos como BERT y ViT, debido a su suavidad y mejor comportamiento empírico en tareas complejas. Fue introducida por Hendrycks y Gimpel en 2016 [36]. Se define como:

$$\text{GELU}(x) = x \cdot \Phi(x) \approx 0{,}5x \left(1 + \tanh\left[\sqrt{\frac{2}{\pi}}(x + 0{,}044715x^3)\right]\right)$$

donde $\Phi(x)$ representa la función de distribución acumulada de una normal estándar.

A diferencia de *ReLU*, la activación *GELU* atenúa gradualmente los valores negativos en lugar de anularlos por completo, lo que permite una mayor sensibilidad en la propagación del gradiente. Esto puede traducirse en una mejor capacidad de aprendizaje, especialmente en tareas con relaciones no lineales más sutiles.

En ViT, *GELU* se emplea en las subcapas *feed-forward* por su capacidad para ofrecer un mejor ajuste al aprendizaje durante el entrenamiento.

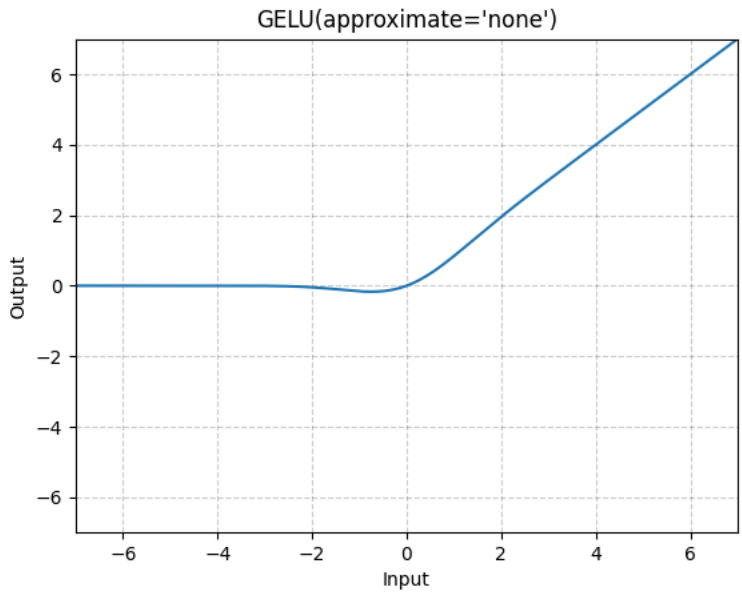


Figura 3.11: Representación gráfica de la función GELU de [11].

3.3.3. Comparativa general

Ambas funciones introducen no linealidad, pero lo hacen de forma distinta. Mientras que *ReLU* es abrupta, propensa a anular ciertos gradientes y no diferenciable, *GELU* proporciona una transición más suave, lo que puede traducirse en mejores resultados en ciertas tareas.

- **ReLU:** simple, rápida y eficaz, ideal para arquitecturas profundas tradicionales.
- **GELU:** suave, probabilística y más precisa en entornos con relaciones complejas.

El uso de una u otra depende en gran medida del tipo de tarea y del modelo. ViT, como arquitectura moderna basada en Transformers, se beneficia de las propiedades de *GELU* para mejorar la capacidad de aprendizaje y la estabilidad del entrenamiento, aunque como se verá más adelante, dependerá del caso específico.

3.3.4. Técnicas de Clasificación y Optimización

El entrenamiento de modelos de clasificación basados en Transformers requiere seleccionar adecuadamente tanto los algoritmos de optimización, como la función de pérdida y los esquemas de ajuste del *learning rate*. Esta sección describe los elementos utilizados en este proyecto.

Optimizadores: Adam y AdamW

Para la actualización de pesos durante el entrenamiento se han empleado dos variantes del optimizador basado en gradiente estocástico: *Adam* y *AdamW*.

El optimizador Adam (*Adaptive Moment Estimation*) combina los beneficios de *Momentum* y *RMSProp*, adaptando el *learning rate* de cada parámetro individualmente a partir de los primeros y segundos momentos del gradiente [37].

Por otro lado, AdamW es una modificación propuesta específicamente para mejorar el rendimiento en modelos de Transformers, introduciendo una descomposición explícita de la regularización L2, lo que lleva a un mejor control del peso del decaimiento [12]. Este optimizador es el utilizado por defecto en muchas implementaciones modernas como Hugging Face.

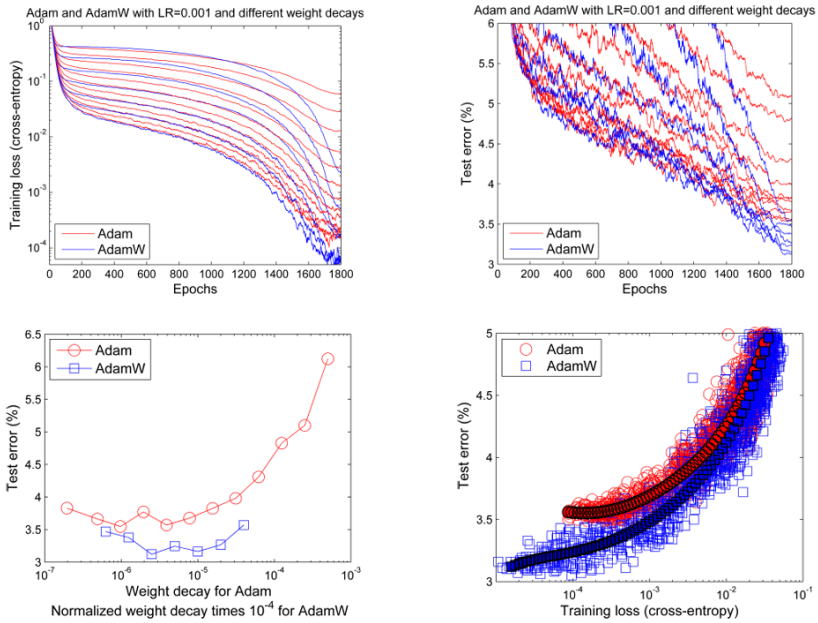


Figura 3.12: Curvas de aprendizaje y generalización obtenidas con una ResNet-26 entrenada en CIFAR-10 usando Adam y AdamW, comparando distintos valores de weight decay y su efecto sobre la pérdida y el error de test de [12].

Como se observa en la Figura 3.12, AdamW consigue una menor pérdida y error en test en comparación con Adam cuando se utiliza un *weight decay* adecuado. Esto demuestra cómo la regularización desacoplada mejora la capacidad de generalización del modelo.

Función de pérdida: Entropía cruzada con pesos

En tareas de clasificación multiclase, la función de pérdida utilizada habitualmente es la entropía cruzada (*Cross Entropy Loss*), debido a su capacidad para medir la diferencia entre dos distribuciones de probabilidad: la predicha por el modelo y la verdadera. En redes neuronales, la salida del modelo se interpreta como una distribución de probabilidad mediante la función *softmax*, y la entropía cruzada penaliza aquellas predicciones que asignan baja probabilidad a la clase correcta.

Dado un vector de probabilidades predicho $\hat{y} = (\hat{y}_1, \hat{y}_2, ..., \hat{y}_C)$ y una etiqueta verdadera codificada como *one-hot* $y = (0, ..., 1, ..., 0)$, la entropía cruzada se define como:

$$\mathcal{L}_{\text{CE}} = -\sum_{i=1}^{C} y_i \log(\hat{y}_i)$$

donde C es el número de clases. Esta fórmula se reduce a $-\log(\hat{y}_k)$ si la clase correcta es la k-ésima.

En contextos con clases desbalanceadas, la entropía cruzada tiende a favorecer las clases mayoritarias. Para contrarrestar este efecto, se usan pesos de clase que aumentan el impacto de los errores cometidos sobre clases minoritarias. Así, la pérdida ponderada se define como:

$$\mathcal{L}_{\text{CE}} = -\sum_{i=1}^{C} w_i y_i \log(\hat{y}_i)$$

donde w_i es el peso asociado a la clase i. Esta estrategia mejora la sensibilidad del modelo frente a clases con poca representación, contribuyendo a un entrenamiento más equilibrado.

Otra técnica es el *label smoothing*, que actúa como regularizador. En lugar de utilizar una codificación *one-hot* estricta (donde la clase correcta tiene probabilidad 1 y el resto 0), se asigna una pequeña parte de probabilidad a las clases incorrectas. Esto evita que el modelo se vuelva confiado de manera excesiva y favorece una mayor generalización.

Formalmente, la etiqueta suavizada para la clase correcta k se expresa como:

$$y_i^{\text{smooth}} = \begin{cases} 1 - \varepsilon & \text{si } i = k \\ \frac{\varepsilon}{C-1} & \text{si } i \neq k \end{cases}$$

donde $\varepsilon \in [0, 1]$ es el parámetro de suavizado y C es el número total de clases. Al repartir parte de la probabilidad objetivo entre las demás clases, se reduce el sobreajuste y se mejora la robustez del modelo ante ruido o ambigüedad en los datos.

Ajuste del Learning Rate: Cosine Scheduling

Otro aspecto fundamental para lograr una convergencia estable es la gestión dinámica del *learning rate*. En este trabajo, se han utilizado dos esquemas principales:

- **CosineAnnealingLR**: reduce el *learning rate* siguiendo una curva coseno decreciente, hasta llegar a un valor mínimo al final del entrenamiento. Mejora la estabilidad y evita oscilaciones tardías.
- **get_cosine_schedule_with_warmup**: scheduler de la librería `transformers` de Hugging Face. Añade una fase inicial de *warm-up* (crecimiento progresivo del *learning rate*) antes de aplicar la curva coseno, lo que facilita una adaptación suave en los primeros pasos del entrenamiento.

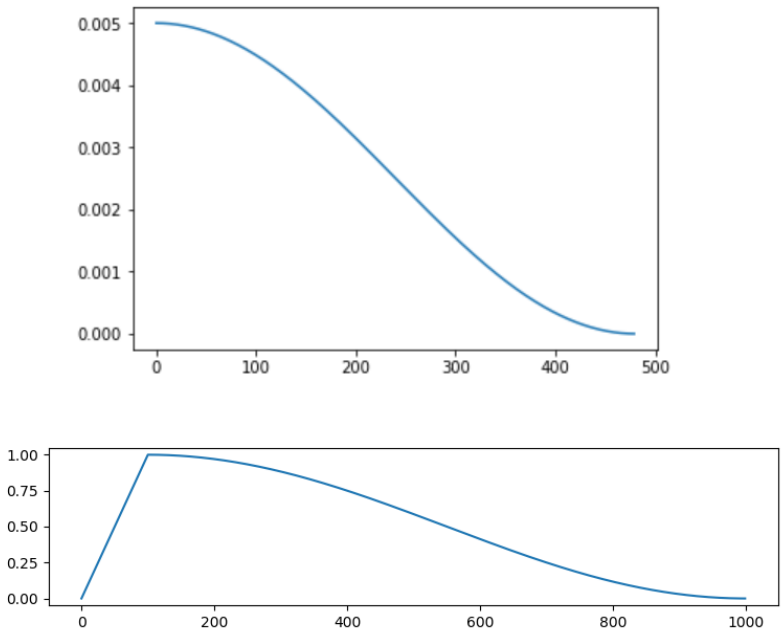


Figura 3.13: Curvas de aprendizaje típicas: Cosine Scheduler y Cosine con Warmup de [13].

3.4. Técnicas de explicabilidad visual

Si bien es fundamental comprender la arquitectura y funcionamiento interno de los Vision Transformers, hoy en día va desarrollándose la explicabilidad de la decisión del modelo. Esto adquiere su importancia en contextos críticos como el diagnóstico médico. En este tipo de

ámbitos, la mera precisión del modelo basta, pero sería deseable conocer por qué se ha tomado una decisión. Esto ayuda en mayor medida al experto, detectar errores y a validar que el modelo no esté aprendiendo patrones irrelevantes.

A lo largo de los años, se han desarrollado una gran cantidad de técnicas de explicabilidad visual para Redes Convolucionales, como *Class Activation Mapping* (CAM)[38], *Grad-CAM*[39], *Score-CAM*[40], o *Recipro-CAM*[41]. Todas ellas han demostrado su eficacia para localizar visualmente las regiones que más contribuyen a una predicción. Sin embargo, cuando se intenta aplicar estos enfoques sobre arquitecturas ViT, esta eficacia se ve empeorada seriamente, pues su naturaleza paralelizable complica la interpretación directa de sus gradientes.

3.4.1. Limitaciones del uso de gradientes en ViT

Las técnicas de explicabilidad basadas en gradientes, como *Grad-CAM*[39], calculan derivadas del score de la clase respecto a las activaciones internas del modelo. Aunque han demostrado buenos resultados en redes convolucionales, su aplicabilidad a arquitecturas Transformer es limitada por múltiples motivos estructurales:

- **Relaciones complejas entre tokens**: a diferencia de las CNN, donde la activación está directamente relacionada con una posición espacial local, en los ViT cada token puede atender a cualquier otro. Estas relaciones se propagan a lo largo de muchas capas de atención, lo que dificulta que una activación temprana o su gradiente refleje una contribución clara a la predicción final.
- **Acumulación de capas y proyecciones**: cada capa del Transformer aplica una combinación de atención *multi-head* y bloques *feed-forward*. Esta acumulación de transformaciones lineales y no lineales provoca que los gradientes de tokens específicos puedan degradarse o ser difíciles de interpretar (problema relacionado con *gradient saturation* o *vanishing gradients*).
- **Los gradientes no garantizan causalidad**: los gradientes muestran sensibilidad, no causalidad. Un gradiente alto no implica que esa región haya sido decisiva en la predicción, sino que una pequeña perturbación podría haber afectado el resultado. Esto puede crear correlaciones espurias[42], es decir, que se genere una relación entre cierta región de la imagen con una clase la cual no es real o causal, generando mapas engañosos.

Estas limitaciones reducen la efectividad de los gradientes como herramienta de interpretación, especialmente cuando se requieren explicaciones robustas y específicas de clase.

3.4.2. Limitaciones de métodos tradicionales en ViT

Debido a los problemas que presenta la arquitectura particular de los ViT, se han propuesto técnicas como *Attention Rollout*[43] o *Relevance*[44] para generar mapas de saliencia.

Aunque logran resultados visuales aceptables, presentan serias limitaciones:

- **No son específicas de clase**: los métodos basados en atención no están diseñados para reflejar la importancia de una clase concreta, lo que reduce su utilidad en tareas multiclase, como es el caso de los datasets de este trabajo.
- **Requieren acceso interno al modelo**: necesitan extraer y procesar todas las matrices de atención internas, lo que implica una alta dependencia de la arquitectura.
- **No se pueden aplicar en entornos sin acceso a gradientes**: gran cantidad de modelos no permiten calcular retropropagación.

Por todo lo anterior, el uso de gradientes y los métodos tradicionales presentan problemas tanto prácticos como teóricos. Estas limitaciones han motivado el desarrollo de enfoques alternativos que no requieran ni gradientes ni acceso a matrices de atención.

3.4.3. ViT-ReciproCAM

En 2023, Byun y Lee[14] propusieron una técnica de explicabilidad visual específicamente diseñada para ViT, denominada *ViT-ReciproCAM*. Se centra en determinar qué regiones de la imagen son realmente determinantes en la predicción del modelo. Para ello, se generan versiones modificadas de la entrada original mediante el enmascaramiento selectivo de patches, observando cómo cambia la confianza del modelo en la clase predicha, como se ve en la Figura 3.14.

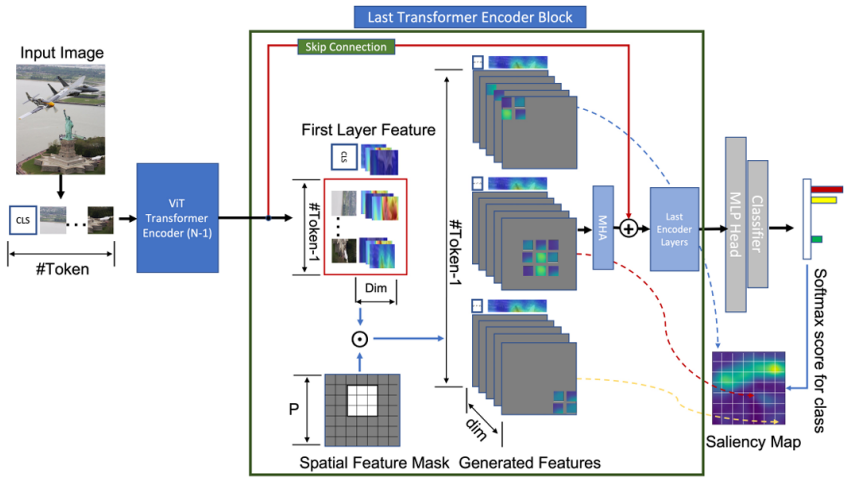


Figura 3.14: Arquitectura del ViT-ReciproCAM de [14].

El método construye un mapa de saliencia midiendo, para cada patch de la imagen, cuánto disminuye la probabilidad de la clase objetivo al eliminarlo. De este modo, se obtiene una

representación interpretable, que indica la contribución relativa de cada región sin necesidad de inspeccionar la arquitectura interna del modelo. ViT-ReciproCAM destaca por su simplicidad conceptual, su aplicabilidad en escenarios reales de inferencia y su capacidad para generar mapas de saliencia más localizados que otras técnicas comparables.

Se extrae un mapa de características con dimensiones $(H \times T \times D)$ a partir de la primera capa *LayerNorm* del último bloque del codificador del Transformer, donde H representa el número de cabezas (en la primera capa, $H = 1$ ya que todas están concatenadas), T es el número de tokens, y, D, la dimensión del codificador.

A partir de este mapa, se genera $(T - 1)$ máscaras espaciales, cada una de las cuales corresponde a un nuevo mapa de entrada, que se usará en las capas posteriores. Para cada posición espacial (x, y), definida como el centro de una máscara espacial Gaussiana, el método mide el score de predicción de una clase específica usando únicamente el token de característica correspondiente.

Cabe destacar que el método ignora la dimensión de batch para simplificar. La eficacia de ViT-ReciproCAM ha sido evaluada por Byun et. al.[14] sobre el conjunto de validación de ImageNet, mostrando un rendimiento superior frente a otros métodos de referencia del estado del arte.

Generación de máscaras espaciales y características

La máscara espacial de tokens M tiene dimensiones $(N \times T)$, donde $N = (T - 1)$ y T representa el token `[CLS]` más el número de patches, es decir, $T = P^2 + 1$, pues la imagen está dividida en $P \times P$ patches. Para cada $n \in [0, \ldots, N - 1]$, se genera una máscara que activa únicamente una región 3×3 de tokens espaciales, centrada en la posición correspondiente del token enmascarado. Los demás tokens se fijan a cero, excepto el token de clase que se mantiene constante.

A partir del mapa de características original F_k de la primera capa *LayerNorm* del último bloque codificador del Transformer, se pueden generar nuevos mapas de entrada F_k^n mediante multiplicación elemento a elemento (producto de Hadamard $\odot$) con las máscaras espaciales M^n:

$$\tilde{F}_k^n = F_k \odot M^n$$

Cada nuevo mapa de entrada contiene las características originales de una región de la imagen, reescaladas por un kernel Gaussiano 3×3. Estas versiones modificadas son utilizadas para estimar la importancia del token central observando cómo varía la confianza de la red en la clase objetivo.

Si bien el uso del kernel Gaussiano es el enfoque por defecto, denominado *ViT-ReciproCAM[3×3]*, cabe señalar que también es posible aplicar una máscara basada en el enmascaramiento de un único token, lo que ofrece una alternativa más sencilla, el *ViT-ReciproCAM*.

Para generar el mapa de saliencia, se divide el modelo en dos partes: la primera parte ($\mathcal{G}$) corresponde a las capas hasta la extracción de características (LayerNorm) y la segunda parte ($\mathcal{H}$) representa el resto del modelo. Al alimentar un lote de N mapas modificados a las capas $\mathcal{H}$, se obtienen puntuaciones de predicción y_c^n para una clase c.

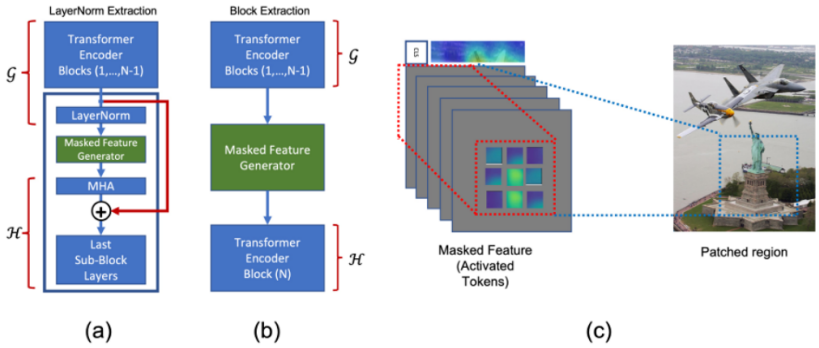


Figura 3.15: (a) Extracción de características desde la primera capa *LayerNorm* del último bloque codificador, (b) extracción de características desde la salida completa del bloque, (c) los tokens enmascarados cubren el área delimitada por la línea azul discontinua en la imagen de entrada de [14].

Estas puntuaciones permiten calcular la importancia relativa de cada token enmascarado. El mapa de saliencia final para la clase se obtiene normalizando y reestructurando los scores:

$$S^c = \text{reshape}\left[\frac{Y_c - \text{mín}(Y_c)}{\text{máx}(Y_c) - \text{mín}(Y_c)}, (P, P)\right]$$

donde $Y_c = [y_c^1, \ldots, y_c^N]^T$ es el vector de scores para la clase c, y cada y_c^n es calculado como:

$$y_c^n = \text{softmax}\left(\mathcal{H}\left(\mathcal{G}(I) \odot M^n\right)\right)_c$$

La operación reshape$[P, P]$ reorganiza los valores escalares unidimensionales obtenidos para cada patch en una matriz bidimensional del mismo tamaño que el grid de patches de entrada [14].

Comparación de métodos

Varios ejemplos de la eficacia de esta técnica son representados en la figura 3.16. Muestra una comparación visual entre distintos métodos de explicabilidad aplicados a ViTs: *Attention Rollout*, *Relevance*, *ViT-ReciproCAM* y su variante *ViT-ReciproCAM [3×3]*. Se presentan

tres escenarios representativos: (i) un objeto simple (Mantis), (ii) múltiples objetos idénticos (Yachts) y (iii) imágenes con múltiples clases (Zebra y Elephant).

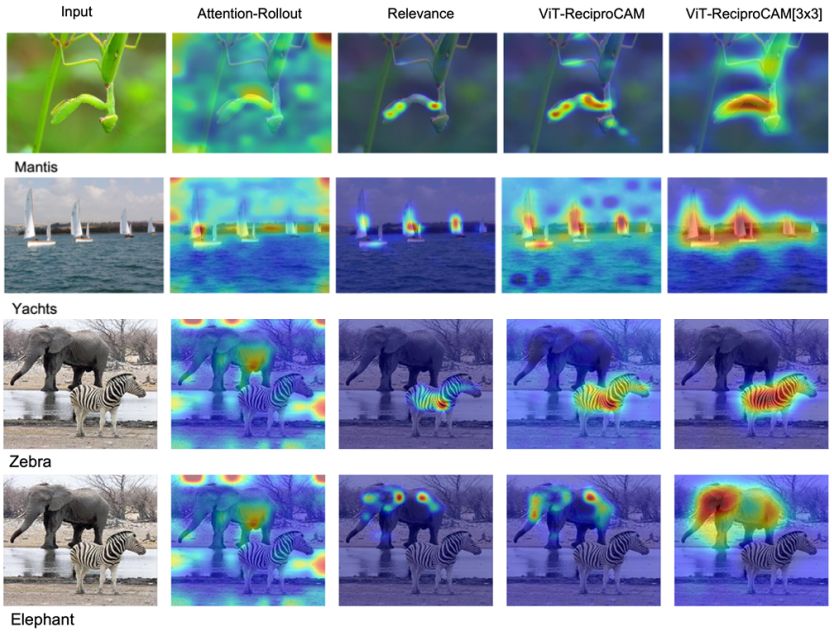


Figura 3.16: Resultados de objeto simple (Mantis), varios objetos iguales (Yachts) y múltiples clases (Elephant y Zebra). Adaptación de varias figuras de [14].

Puede observarse que los métodos basados en atención como *Attention Rollout* generan mapas más difusos y poco específicos. El método *Relevance* mejora la focalización en regiones relevantes, pero aún presenta activaciones espurias. En contraste, *ViT-ReciproCAM* y especialmente su versión $[3 \times 3]$ producen mapas más localizados, que capturan con mayor precisión las regiones responsables de la predicción del modelo, incluso en presencia de múltiples objetos o clases.

Estos resultados demuestran el potencial de *ViT-ReciproCAM* como herramienta eficaz de explicabilidad visual, especialmente en entornos donde se requiere interpretar la decisión del modelo de forma localizada y centrada en una clase específica. No obstante, como se aprecia en los ejemplos, su variante $[3 \times 3]$ ofrece ciertas ventajas adicionales en términos de continuidad y precisión visual, generando mapas de saliencia más suaves, coherentes y centrados en las regiones relevantes. Esto se debe a que el enmascaramiento realizado en bloques proporciona un mayor contexto espacial, lo cual reduce la fragmentación del mapa y mejora en gran medida la interpretabilidad visual. Por esta razón, dicha variante ha sido la seleccionada para aplicarse en este trabajo.

Capítulo 4

Marco de trabajo

En este capítulo, se detallarán las diferentes recursos físicos y tecnológicos empleados durante el desarrollo del proyecto. Se describen tanto los aspectos relacionados con el entorno software, como el hardware sobre el que se ha trabajado y ejecutado el entrenamiento de los modelos. Se justifican sus decisiones exponiendo sus ventajas y limitaciones. De esta manera, se da a entender las condiciones bajos las cuales se han obtenido los resultados experimentales.

4.1. Hardware

Se han empleado dos dispositivos para llevar a cabo el desarrollo del proyecto:

- **Maquina virtual:** prestada por el departamento de Informática de la Universidad de Valladolid. Esta máquina dispone de un procesador *Intel® Xeon® Gold 6326* a 2.90 GHz con 32 núcleos, 64 GB de memoria RAM y 50 GB de espacio en disco.
- **Ordenador personal:** cuenta con un procesador *Intel® Core™ i5-8600K* a 3.60 GHz (hasta 4.30 GHz) con 6 núcleos, 16 GB de memoria RAM y más de 3TB de espacio en disco. Cuenta con una tarjeta gráfica *NVIDIA GeForce GTX 1060* con 6GB DDR5 de VRAM, 1506MHz (hasta 1708 MHz) y 1280 CUDA cores.

El uso combinado de ambos dispositivos ha resultado especialmente útil para el desarrollo del proyecto. La máquina virtual, gracias a su gran capacidad de procesamiento y sus 64GB de memoria RAM, ha sido la encargada de entrenar modelos con conjuntos de datos de gran tamaño, donde el consumo de recursos es considerable. La que se puede considerar su mejor ventaja es su disponibilidad constante, pues permite dejar modelos en entrenamiento durante largos periodos de tiempo sin preocupación.

Por otro lado, el ordenador personal también ha resultado imprescindible, especialmente para conjuntos de menor tamaño, ya que su tarjeta gráfica dedicada de 6GB de VRAM permite acelerar el entrenamiento en gran medida.

Esta combinación ha hecho posible ejecutar de forma paralela varios entrenamientos, optimizando los tiempos y aprovechando al máximo los recursos disponibles, cosa de especial importancia en un proyecto como este, donde el objetivo no es obtener un modelo sino varios.

4.2. Software

4.2.1. Sistema operativo

La maquina virtual cuenta con un sistema operativo *Debian GNU/Linux 12 (Bookworm)*, mientras que el ordenador personal cuenta con un *Windows 10 Pro*.

4.2.2. Lenguajes y herramientas

Python

Esta elección no se ha basado únicamente en sus propias ventajas, sino también en su comparación con otras alternativas comunes en el ámbito del Análisis de Datos y el Aprendizaje Automático, como *R* y *Julia*, las cuales no encajan con el presente proyecto.

R es una herramienta muy potente para análisis estadístico y visualización de datos. Sin embargo, su ecosistema orientado principalmente al análisis exploratorio y no tanto al desarrollo de sistemas complejos de aprendizaje profundo. Esto, junto con las limitadas librerías que ofrece para este tipo de proyectos, lo hace menos adecuado.

Julia, por otro lado, presenta ventajas en cuanto a rendimiento computacional, ya que ha sido diseñado específicamente para ello. No obstante, su comunidad y ecosistema son reducidos en comparación con *Python*, lo cual supone una limitación significativa a la hora de encontrar bibliotecas potentes y ejemplos prácticos.

Por tanto, *Python* es el lenguaje restante para realizar el proyecto. Este se ha consolidado como el de referencia en Aprendizaje Automático y Ciencia de Datos, debido a su gran comunidad, abundancia de bibliotecas especializadas (como *NumPy*, *Pandas*, *scikit-learn*, *PyTorch*, entre muchas otras) y su compatibilidad con entornos de trabajo como *Jupyter Notebooks*. Su sintaxis sencilla y legibilidad han favorecido un desarrollo ágil durante todas las fases del proyecto, desde el preprocesamiento de datos hasta el despliegue de los modelos.

Pytorch

Dado que se utiliza *Python* como lenguaje de desarrollo, para la implementación de los modelos de aprendizaje profundo, se ha optado por el framework *PyTorch*. Aunque existen otras alternativas como *Keras* (frecuentemente utilizado a través de *TensorFlow*), la elección de *PyTorch* se debe a varios factores.

En primer lugar, *PyTorch* ofrece mayor flexibilidad y control en la creación de modelos y bucles de entrenamiento. Su enfoque basado en gráficos dinámicos (*define-by-run*) permite una depuración más sencilla, lo cual ha resultado útil para la implementación de arquitecturas complejas como los *Vision Transformers* (ViT).

Aunque *Keras* destaca por su simplicidad y curva de aprendizaje más suave, esta abstracción puede ser una limitación, cuando se desea personalizar el comportamiento interno de los modelos.

PyTorch también cuenta con una comunidad muy activa, siendo el framework preferido en la mayoría de publicaciones recientes sobre aprendizaje profundo. Esto facilita el acceso a una gran cantidad de ejemplos reales, facilitando el aprendizaje.

En definitiva, *PyTorch* ha sido elegido por su equilibrio entre potencia, flexibilidad y soporte comunitario, lo que lo convierte en una herramienta especialmente adecuada para proyectos de investigación como este Trabajo de Fin de Grado.

Jupyter y Anaconda

El entorno de desarrollo utilizado ha sido *JupyterLab*, gestionado a través de la distribución *Anaconda*. Estas elecciones permiten trabajar de forma modular, visualizando resultados paso a paso y permitiendo una gestión más sencilla de los distintos códigos. Cuenta con una gran cantidad de bibliotecas para el desarrollo de proyectos de Ciencias de Datos, entre las que se encuentra *PyTorch*.

HTML, CSS y JavaScript

Para la parte de despliegue web se han utilizado tecnologías estándar del desarrollo frontend como *HTML*, *CSS* y *JavaScript*, que permiten estructurar la interfaz, aplicar estilos visuales y proporcionar interacción.

Flask

La biblioteca *Flask* ha sido utilizada como microframework para el backend de la aplicación web. Su simplicidad y compatibilidad con Python y Pytorch lo convierten en la mejor opción para desplegar los modelos entrenados en una página web sencilla.

Astah

Para la elaboración de diagramas UML (como casos de uso, clases o actividades), se ha utilizado la herramienta *Astah*. Esta aplicación facilita la representación estructurada de los componentes del sistema, ayudando a comunicar de forma visual la arquitectura de la aplicación. Cuenta con una versión de pago, *Astah Profesional*, la cual ha estado disponible gracias a la licencia que proporciona por la Universidad de Valladolid, lo que ha favorecido su elección.

TexStudio

La memoria del proyecto ha sido redactada íntegramente en LaTeX, empleando el editor *TeXstudio*. Este entorno permite gestionar documentos de forma profesional, garantizando un formato uniforme y la correcta inserción de fórmulas, figuras, tablas y referencias bibliográficas. Se ha proporcionado una plantilla base por parte de la Escuela, que incluye la estructura general del documento, así como los paquetes y configuraciones necesarios.

Si bien es cierto que otras opciones como *OverLeaf* cuentan con funcionalidades en la nube para asegurar el control de versiones, el dominio previo de esta herramienta y su versatilidad en entornos locales, se han considerado como factores de mayor peso a la hora de realizar la elección.

GanttProject

Para la planificación temporal del proyecto se ha utilizado la herramienta *GanttProject*. Esta aplicación de código abierto permite gestionar tareas, asignar recursos, definir dependencias entre actividades y visualizar el progreso mediante un diagrama de Gantt. Gracias a su interfaz intuitiva, ha sido posible estructurar las distintas fases del proyecto, desde la investigación inicial hasta el despliegue final, de forma clara y eficiente.

GanttProject facilita la exportación de los diagramas a distintos formatos como PDF o PNG, lo cual ha resultado útil para la inclusión de la planificación dentro de la documentación del proyecto.

Capítulo 5

Conjuntos de datos

Aunque este trabajo se enfoque en la comparación objetiva entre las arquitecturas ViT y CNN en el ámbito de la salud, el centro de atención se basa en la adaptación de los Transformers para el caso de la clasificación de imágenes. Por ello, no sólo es necesario tener unos datasets adecuados y representativos para la comparación en este ámbito, sino que también tengan estudios de calidad relacionados con su clasificación a partir de CNNs.

Para que se pueda cumplir con las condiciones, se han seleccionado tres conjuntos de datos utilizados en Trabajos de Fin de Grado anteriores. En cada uno de ellos, se han mantenido las mismas transformaciones y preprocesamientos aplicados originalmente, permitiendo así una comparación directa entre ambas arquitecturas. A continuación, se describen estos datasets.

5.1. Radiografías de tórax (CXR)

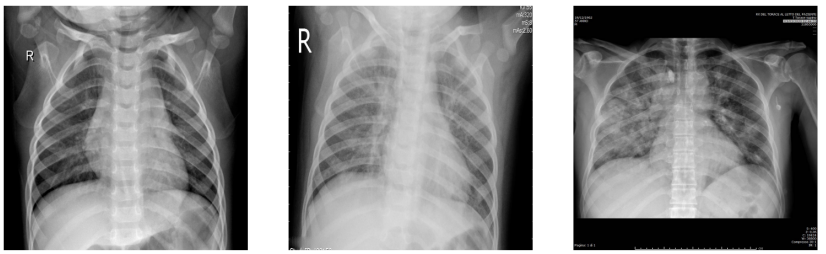
5.1.1. Descripción

El conjunto de imágenes corresponde con radiografías de tórax y proviene de una competición de Kaggle [45]. Cuenta con un total de 2905 imágenes con una resolución de 1024×1024 píxeles y repartidas en tres categorías diferentes como se puede ver en la figura 5.1:

- **Normal:** pacientes sanos que no presentan enfermedad.
- **Neumonía vírica:** pacientes que presentan neumonía pero no COVID-19.
- **COVID-19:** pacientes con COVID-19.

Según comenta Toquero [15], las imágenes correspondientes a las dos primeras clases provienen de la base de datos de Kaggle de Paul Moore, *Chest X-Ray Images (Pneumonia)*

[46]. Por otro lado, las imágenes de COVID-19 provienen de diferentes fuentes abiertas: la base de datos COVID-19 de la Sociedad Italiana de Radiología Médica e Intervencionista, *Società Italiana di Radiologia Medica e Interventistica (SIRM)*, del conjunto de datos *Novel Corona Virus 2019 (nCOVID-19)* de Joseph Paul Cohen, Paul Morrison y Lan Dao, y de otras 43 publicaciones diferentes [47].



(a) Normal (b) Pneumonía vírica (c) Covid-19

Figura 5.1: Ejemplos de cada clase de [15].

5.1.2. Transformaciones

Como se puede apreciar en la figura 5.1, las tres clases son imágenes en escala de grises (y por tanto de un solo canal). Sin embargo, Toquero [15] detalla que sólo *Normal* y *Neumonía vírica* se encuentran de manera natural así, mientras que las imágenes con los positivos en *COVID-19* tienen formato RGB, es decir, tres canales. Por tanto, transforma las imágenes de color a escala de grises para poder tratarlas de manera uniforme al realizar el modelo.

Dado que los valores de los píxeles se encuentran en el rango [0,255], las imágenes se someten a un proceso de normalización con el fin de escalar dichos valores a [0,1]:

$$X_i = \frac{X_i - X_{\min}}{X_{\max} - X_{\min}} \tag{5.1}$$

siendo X_i valor del píxel i, $X_{\max}$ y $X_{\min}$ corresponden a los valores máximo y mínimo posibles respectivamente. Para este caso particular, el cálculo es equivalente a dividir todos los valores de los píxeles por 255.

El conjunto de datos se encuentra distribuido en dos carpetas: entrenamiento y test. La división está hecha de manera estratificada, es decir, se mantiene la proporción de número de muestras de cada clase. Esto es de gran importancia, cuando el número de observaciones por clase está desequilibrado. Siendo éste el caso, pues el número de muestras de la clase *COVID-19* es bastante inferior al de las otras dos (figura 5.2).

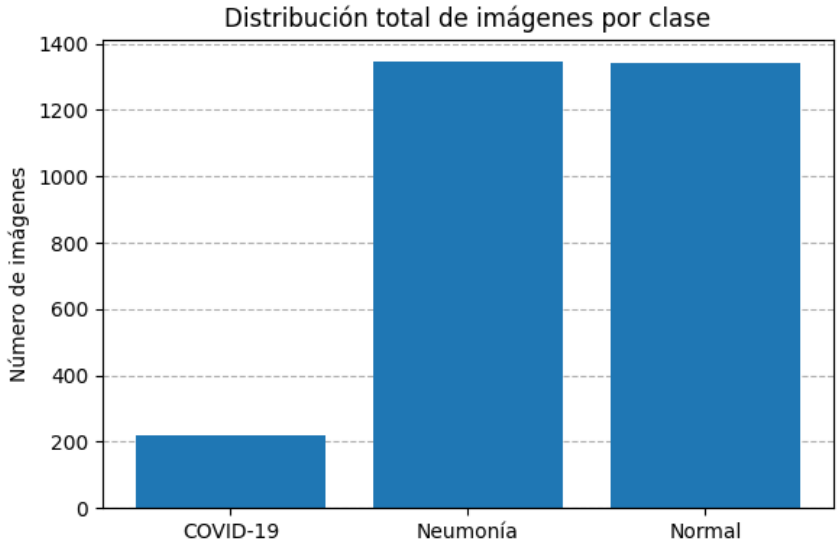


Figura 5.2: Gráfica de la distribución de clases de radiografías de tórax.

Clases	Conjunto de entrenamiento	Conjunto de test	Total
Covid-19	146	73	**219**
Neumonía	896	449	**1345**
Normal	894	447	**1341**
Total	**1936**	**969**	**2905**

Tabla 5.1: Distribución de clases de radiografías de tórax.

Esto tipo de partición denominada *Hold Out*, evita realizar una estimación del error optimista, que ocurre cuando se utilizan los mismos datos con los que ha sido entrenado, haciendo que se sobreestime la verdadera capacidad de generalización del modelo. En cambio, utilizando otros datos diferentes, se consigue una estimación justa y se puede comprobar si el modelo es capaz de generalizar correctamente.

5.1.3. Obtención y uso

Los datos provienen de un Trabajo de Fin de Grado [15] dirigido por el mismo tutor que el presente trabajo. Gracias a ello, se ha podido contar desde un principio con los datos ya recopilados, organizados y preprocesados, exceptuando la normalización. No obstante, esto no supone un problema, pues dicha normalización se realiza de forma automática al transformar las imágenes a tensores [48].

Como se detallará más adelante, el elevado tamaño original de las imágenes implica ciertos problemas computacionales, por lo que se ha decidido redimensionarlas a un tamaño

más manejable de 256×256 píxeles.

5.2. Resonancias magnéticas de cerebro (MRI)

5.2.1. Descripción

El conjunto de imágenes utilizado por Arranz [16] corresponde con resonancias magnéticas de cerebro y proviene directamente de la base de datos de Kaggle [49]. Aunque sería recomendable utilizar otras versiones más recientes y con un mayor número de muestras, se utiliza el original para realizar la comparación de la manera más objetiva y directa posible.

Cuenta con un total de 3264 imágenes de diferente resolución, aunque la mayoría de ellas tienen un tamaño de 512×512 píxeles. Están repartidas en cuatro categorías diferentes como se puede ver en la figura 5.3:

- **No tumor:** cerebros sanos sin presencia de tumores.
- **Glioma:** tumores cerebrales del tipo glioma, que se originan en las células gliales [50].
- **Meningioma:** tumores que se desarrollan en las meninges, las membranas que rodean el cerebro y la médula espinal [50].
- **Pituitaria:** tumores en la glándula pituitaria, también conocida como hipófisis [50].

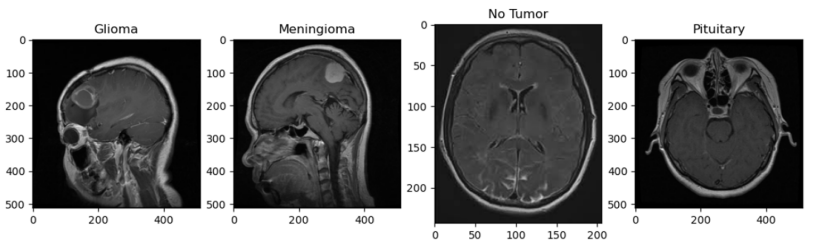


Figura 5.3: Ejemplos de cada clase de [16].

Si bien es cierto que sería posible ampliar el conjunto con imágenes de otras fuentes, Arranz señala en [16] que el desconocimiento sobre la procedencia de las imágenes dificulta esta tarea, ya que existe el riesgo de incluir imágenes duplicadas, empeorando así el aprendizaje.

5.2.2. Transformaciones

El conjunto de datos se encuentra originalmente divido en dos carpetas: entrenamiento y test. Sin embargo, esta partición presenta un problema importante: no ha sido realizada de forma estratificada, como se puede observar en la Tabla 5.2. Por este motivo, Arranz [16] considera combinar ambas carpetas y realizar la división de manera dinámica en el propio código, aunque esto solo lo realiza durante la adaptación del modelo definitivo, pues durante los modelos iniciales opta por utilizar la división original.

En cuanto a la distribución de clases, el conjunto no presenta un desbalance excesivo, con la excepción de la clase correspondiente a pacientes sin tumores, que contiene aproximadamente la mitad de muestras en comparación con las clases con tumores, como se muestra en la Figura 5.4. Aunque la diferencia no es tan pronunciada como en los otros conjuntos de imágenes, puede ser un factor a tener en cuenta durante el entrenamiento.

Clases	**Entrenamiento (%)**	**Test (%)**	**Total (%)**
Glioma	28.78 %	25.38 %	28.37 %
Meningioma	28.64 %	29.19 %	28.70 %
Pituitaria	28.82 %	18.78 %	27.60 %
No-tumor	13.76 %	26.65 %	15.31 %
Total	**100 %**	**100 %**	**100 %**

Tabla 5.2: Porcentaje de representación de clases de resonancias magnéticas cerebrales.

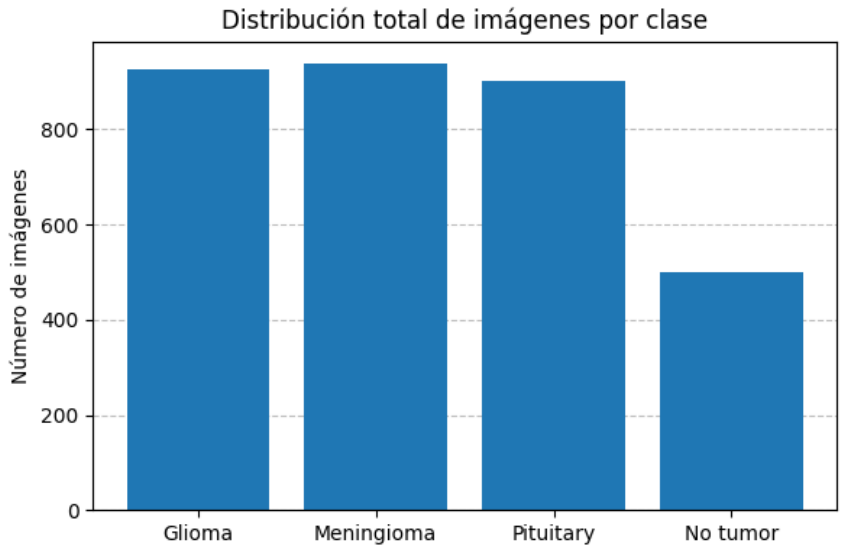


Figura 5.4: Gráfica de la distribución de clases de resonancias magnéticas cerebrales.

Clases	Conjunto de entrenamiento	Conjunto de test	Total
Glioma	826	100	**926**
Meningioma	822	115	**937**
Pituitaria	827	74	**901**
No-tumor	395	105	**500**
Total	**2870**	**394**	**3264**

Tabla 5.3: Distribución de clases en el conjunto de resonancias magnéticas cerebrales.

Dejando de un lado la distribución de las muestras, hay que tener en cuenta la heterogeneidad de las imágenes. En primer lugar, no todas las tomografías se han tomado desde el mismo ángulo, como se ve en la figura 5.3. Como ya se ha comentado, las imágenes presentan diferentes resoluciones, introduciendo variabilidad adicional.

En relación con estos aspectos, Arranz [16] argumenta que la variación en el ángulo de captura no supone un inconveniente significativo, ya que el modelo debe ser capaz de generalizar independientemente de la orientación de las imágenes, pues no siempre se necesita el mismo ángulo. No obstante, sí resulta necesario homogeneizar su tamaño. Para ello, aplica transformaciones de *resize* sobre las imágenes, probando dos diferentes dimensiones: 128×128 y 256×256, con el objetivo de facilitar la computación con un tamaño no excesivamente grande.

Con el fin de evitar el sobreajuste del modelo, también se realizan transformaciones aleatorias tanto de manera horizontal como vertical.

Por último, al igual que el anterior conjunto, se necesita realizar un proceso de normalización de los valores de los píxeles del rango original [0,255] al intervalo [0,1], lo cual se puede llevar a cabo con la ecuación 5.1.

5.2.3. Obtención y uso

Al igual que el anterior conjunto, éste proviene de un Trabajo de Fin de Grado [16] dirigido por el mismo tutor que el presente trabajo. Gracias a ello, se tiene una situación semejante, con solo la partición dinámica.

En cuanto a las transformaciones requeridas, se utilizan ambas dimensiones comentadas para las pruebas. La normalización vuelve a realizarse de forma automática al transformar las imágenes a tensores [48].

5.3. Secciones transversales de tomografías de coherencia óptica (OCT)

5.3.1. Descripción

El conjunto de imágenes estudiado por Izquierdo [17] corresponde con secciones transversales de Tomografías de Coherencia Óptica, que proviene de una competición de Kaggle [51]. Cuenta con un total de 84484 imágenes con diferentes resoluciones como 512×512, 512×496 o 768×496 entre las más concurrentes. Se reparten en cuatro categorías diferentes como se puede ver en la figura 5.5:

- **Normal:** retina sin patologías.
- **CNV:** neovascularización coroidea (*Choroidal Neovascularization*), se desarrollan vasos sanguíneos anómalos debajo de la retina, común en enfermedades como la degeneración macular asociada a la edad [52].
- **DME:** enfermos con edema macular diabético (*Diabetic Macular Edema*), una complicación de la retinopatía diabética que provoca acumulación de líquido en la mácula [52].
- **Drusen:** imágenes que muestran depósitos amarillentos (drusas) bajo la retina, típicos en fases tempranas de la degeneración macular asociada a la edad [52].

Figura 5.5: Ejemplos de cada clase de [17].

Izquierdo [17] también señala que las imágenes fueron seleccionadas de cohortes retrospectivas de pacientes adultos de diversas instituciones. La clasificación de las mismas se llevó a cabo por diferentes niveles de experiencia: en primer lugar, estudiantes; posteriormente, oftalmólogos; y finalmente, dos especialistas en retina con más de 20 años de experiencia clínica [51].

5.3.2. Transformaciones

Dado que este conjunto cuenta con una gran cantidad de muestras, este se encuentra dividido en las carpetas de entrenamiento, test y validación. Sin embargo, como indica Izquierdo

[17] y se puede ver en la figura 5.6, hay demasiada diferencia en la proporción entre dichas particiones. La división original realizada no se hizo de forma estratificada: mientras que la distribución original de las clases es la que se muestra en el gráfico 5.7, el conjunto de test contenía el mismo número de muestras para cada clase.

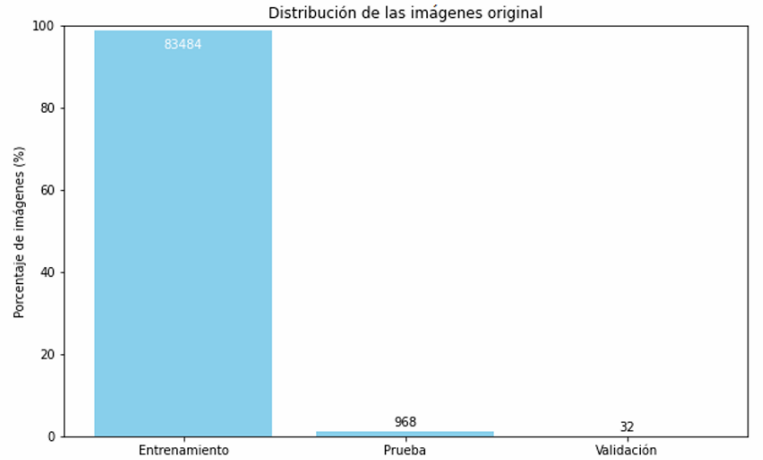


Figura 5.6: Gráfica de la distribución original de [17].

Debido a los problemas derivados de la distribución original, Izquierdo combina todas las muestras y realiza la división de manera dinámica en el código. Esta misma trata de un 70 % para el conjunto de entrenamiento, 20 % para el de test y el 10 % restante para el de validación, aunque ahora con la consecuente estratificación de las clases.

Aunque los otros conjuntos detallados anteriormente cuentan con un volumen apto para su uso, no es suficiente para realizar una tercera partición para validación. En cambio, este conjunto sí permite este tipo de distribución. Ésta resulta especialmente útil cuando se trabaja con metadatos adicionales o se requiere un ajuste muy fino del modelo, pues se dispone de un conjunto que no ha sido utilizado para calibrar los diferentes valores de los metadatos, dando una mejor estimación aún que el error estimado por test.

Clases	**Total**
CNV	37444
DME	11598
Drunsen	8866
Normal	26565
Total	**84473**

Tabla 5.4: Distribución de clases en el conjunto de tomografías de coherencia óptica.

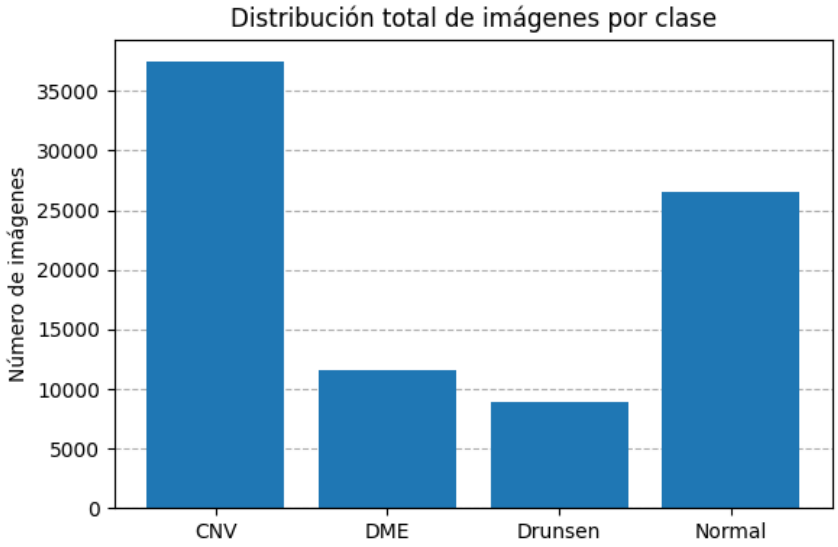


Figura 5.7: Gráfica de la distribución de clases de tomografías de coherencia óptica.

Aparte del problema de la distribución, también hay que lidiar con el diferente tamaño de las imágenes. Para poder paliar el problema, Izquierdo opta por realizar un *resize* a todas las imágenes a un tamaño de 256×256. También comenta la opción de 512×512 pero lo considera de demasiado coste computacional.

Por último, al igual que los anteriores, se necesita realizar un proceso de normalización de los valores de los píxeles del rango original [0,255] al intervalo [0,1], lo cual se puede llevar a cabo con la ecuación 5.1.

5.3.3. Obtención y uso

Al igual que los otros conjuntos, este proviene de un Trabajo de Fin de Grado [17] dirigido por el mismo tutor que el presente trabajo, por tanto los datos vuelven a ser proporcionados de manera directa.

En cuanto a las transformaciones requeridas, se utiliza la misma dimensión, y la normalización vuelve a realizarse de forma automática al transformar las imágenes a tensores [48].

Capítulo 6

Construcción de los modelos

Tras contextualizar el problema, el siguiente paso ha sido la construcción y entrenamiento de los modelos específicos utilizados en este trabajo.

Dado que no todos los conjuntos de datos presentan las mismas características ni plantean los mismos retos, se ha optado por emplear distintos tipos de Vision Transformers (ViT), adaptando su configuración en función de las particularidades de cada caso. Esto ha requerido ajustar tanto los parámetros del modelo como las estrategias de entrenamiento, incluyendo el preprocesamiento de imágenes, la selección de funciones de pérdida y los esquemas de optimización más adecuados.

Se han aplicado diversas técnicas para mejorar el rendimiento y la generalización de los modelos, evaluando de forma sistemática su comportamiento a lo largo del proceso de entrenamiento.

En este capítulo se describen en detalle las decisiones tomadas durante esta fase, así como la metodología seguida para construir, entrenar y validar los distintos modelos utilizados a lo largo del proyecto.

6.1. Planteamiento inicial

A pesar de que cada uno de los modelos implementados presenta variaciones en su funcionamiento, todos comparten una arquitectura base común que sigue el esquema fundamental de un Vision Transformer (ViT). Por tanto, se plantea una estructura modular compuesta por tres componentes principales, definidos como clases independientes con el objetivo de facilitar su reutilización y la experimentación:

- **PatchEmbedding**: se encarga de dividir la imagen en patches no solapados, aplanarlos y organizarlos como una secuencia de tokens.

- **TransformerBlock**: corresponde al bloque principal del codificador Transformer.
- **Modelo ViT**: representa la arquitectura global, integrando las módulos anteriores y generando la salida final.

De esta manera, se intenta comparar diferentes estrategias sin modificar la lógica interna del modelo base, manteniendo así la coherencia estructural entre las distintas variantes desarrolladas.

6.2. Estructuras desarrolladas

Con el objetivo de evitar repeticiones innecesarias y mejorar la claridad de la documentación, en esta sección se describen en detalle las principales estructuras y componentes desarrollados para la implementación de los modelos. Cada fragmento de código referenciado corresponde a partes relevantes del proyecto, acompañado de una explicación detallada sobre su propósito, funcionamiento y relación con la arquitectura global.

6.2.1. Patch Embedding

Para implementar la etapa de división de la imagen en patches, se han considerado varias alternativas disponibles en *PyTorch*. Aunque existen funciones directas como *nn.Unfold* para extraer regiones de una imagen de manera vectorizada, o realizar convoluciones a través de *nn.Conv2d*, se ha optado finalmente por una implementación basada en la biblioteca *einops*, concretamente mediante el uso de la función *Rearrange*[53]. Esta opción es de gran utilidad en la creación de modelos tipo ViT, como se observa en muchas implementaciones en la comunidad [54] [55], ya que proporciona una sintaxis muy legible y expresiva para definir las transformaciones de tensores de manera intuitiva.

Permite especificar transformaciones mediante una notación basada en patrones de ejes. Su funcionamiento se basa en definir explícitamente cómo se reorganizan las dimensiones de un tensor de entrada para obtener una nueva disposición, utilizando una sintaxis tipo `<entrada> -> <salida>`. Esta forma de expresión es especialmente útil para operaciones que implican reestructurar datos sin necesidad de manipular manualmente índices o tamaños.

Por ejemplo, la expresión utilizada en la clase PatchEmbedding:

```
1    Rearrange('b c (h p1) (w p2) -> b (h w) (p1 p2 c)')
```

Listing 6.1: Ejemplo de uso de einops.Rearrange

indica que partimos de un tensor de entrada con forma `(batch, channels, height, width)`, donde la altura y la anchura pueden dividirse en bloques de tamaño `p1` y `p2`, respectivamente. La transformación reordena el tensor dividiendo la imagen en bloques no solapados

de tamaño `(p1, p2)` (cuyos valores se definen explícitamente) y reorganiza el resultado para que cada *patch* aplanado ocupe una posición en una nueva secuencia de forma `(batch, n_patches, patch_dim)`.

En esta notación:

- `b` representa el tamaño del *batch*;
- `c` el número de canales;
- `h` y `w` son los factores que resultan de dividir la altura y la anchura entre el tamaño de *patch*;
- `p1` y `p2` son las dimensiones del *patch*;
- la flecha `->` define la forma deseada tras la transformación.

Un ejemplo de su funcionamiento es el representado en la figura 6.1, donde se utiliza una bandeja de una sola imagen con 3 canales (RGB).

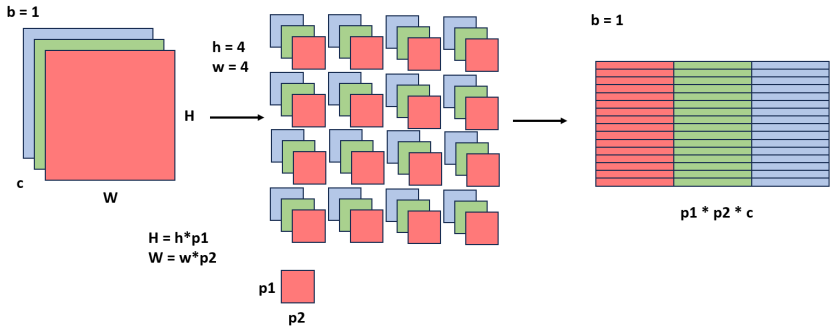


Figura 6.1: Funcionamiento de la función Rearrange de einops.

Durante el desarrollo, se han planteado dos opciones principales. La primera consiste en dividir la imagen en patches según se ha explicado. A continuación, aplica una proyección lineal a una dimensión fija, seguida de una normalización. Esta opción transforma los patches al espacio dimensional requerido por el modelo y añade cierta capacidad de aprendizaje desde el principio del flujo de datos.

La segunda opción, más simple, realiza únicamente la división en patches sin aplicar ninguna proyección adicional. Esta variante es útil cuando se desea mantener la dimensionalidad original del patch y utilizar los valores del tensor directamente.

Ambas alternativas se han implementado dentro de una misma clase A.1, lo que permite alternar entre ellas fácilmente durante las pruebas y comparativas.

La decisión entre utilizar, o no, la proyección depende del diseño general del modelo y del tamaño de entrada esperado por las capas posteriores, además de la naturaleza propia de las

imágenes. Por ejemplo, si se desea mantener una mayor cantidad de información en la etapa inicial o realizar la proyección más adelante en la arquitectura, puede ser preferible omitir la proyección inicial.

En resumen, esta etapa transforma una imagen de tamaño *(C, H, W)* en una secuencia de vectores de tamaño fijo *(N, D)*, donde N es el número de patches y D es la dimensión del embedding.

6.2.2. Transformer Block

Dado que en los modelos ViT se utilizan múltiples capas de codificador con una estructura idéntica, se ha diseñado una clase modular que representa un bloque Transformer, la cual puede integrarse fácilmente mediante el uso de la función *nn.Sequential* para construir modelos tan profundos como se quiera.

Como ya se comentó anteriormente, cada bloque está compuesto por dos componentes principales: una capa de *self-attention* multi-cabeza y una red *feed-forward*, ambas acompañadas por normalización por capas (*Layer Normalization*) y conexiones residuales. Esta estructura es directamente la presentada en el diseño original de Vaswani et al. [3].

En este proyecto se han implementado dos variantes de la clase *TransformerBlock*. La primera versión sigue fielmente la arquitectura Transformer estándar, aplicando conexiones residuales tanto en la atención como en el bloque *feed-forward* A.2.

El primer componente del bloque es la capa de atención multi-cabeza (*MultiheadAttention*), que se encarga de aplicar el mecanismo de atención sobre los embeddings de entrada. Esta operación lleva al modelo a enfocar distintas posiciones de la secuencia simultáneamente, identificando relaciones entre diferentes *tokens* o *patches*, como se detalla en la Sección 3.1.2.

Internamente, esta capa se trata de la clase *nn.MultiheadAttention* de PyTorch [56]. Crea internamente los vectores `query`, `key` y `value` a partir de la entrada mediante parámetros entrenables (*nn.Parameter*), realiza la división en múltiples cabezas de atención, aplica la atención por cabeza de forma paralela y, finalmente, concatena y proyecta el resultado de vuelta al espacio original. Por ello, no es necesario realizar ninguna proyección manual previa; basta con pasar el tensor de entrada como argumento en las tres posiciones (`query`, `key`, `value`).

Dado que, en este caso, se trata de una operación de *self-attention*, los tres tensores son idénticos: corresponden directamente a la salida del bloque anterior. Esto es lo que permite que cada elemento de la secuencia, es decir, que cada patch tenga acceso a todos los demás, aprendiendo qué partes de la imagen debe prestar atención.

La clase también permite aplicar `dropout` de forma integrada sobre los pesos de atención, lo cual contribuye a una mejor regularización durante el entrenamiento.

La salida de esta atención se suma al tensor original mediante una conexión residual, que facilita el flujo de gradientes durante el entrenamiento y previene la desaparición de la

información relevante. Esta suma se normaliza inmediatamente con una capa de *Layer Normalization*, teniendo entonces la variante *post-norm*, es decir, normalizar después de aplicar la atención.

La capa *LayerNorm*, a diferencia de la normalización por lotes (*BatchNorm*), que depende del tamaño del batch y de la estadística global, normaliza cada muestra de forma independiente, utilizando la media y varianza de cada vector de embedding. Esto conduce a un comportamiento más consistente, especialmente útil en tareas como esta donde el tamaño de batch es reducido.

Tras la atención, se aplica un bloque *feed-forward* compuesto por dos capas lineales separadas por una función de activación *ReLU*. La primera capa aumenta la dimensionalidad hasta *mlp_dim*, permitiendo al modelo capturar representaciones más abstractas, mientras que la segunda proyecta de nuevo al espacio de dimensión de embedding original. Se incluye una capa de *Dropout* como técnica de regularización para reducir el sobreajuste. El alcance del `dropout` utilizado en este bloque es el mismo que el empleado en la capa de atención.

Finalmente, se añade una segunda conexión residual seguida de una nueva normalización por capas, completando así la estructura del bloque.

No obstante, en escenarios con conjuntos de datos reducidos, como es este trabajo, se ha observado que mantener conexiones residuales en bloques con baja complejidad (como un MLP de solo dos capas) puede inducir cierto sobreajuste o a veces no afectar verdaderamente al entrenamiento. Por ello, se ha explorado una segunda variante más simplificada, en la que se elimina la conexión residual en el bloque *feed-forward*. De esta manera, el método *forward* queda definido como se muestra en A.3.

Este enfoque busca reducir la capacidad de la red en etapas tempranas del entrenamiento, mejorando la regularización cuando se dispone de un volumen de datos limitado. Ambas variantes se han integrado en el flujo de pruebas del proyecto, permitiendo seleccionar de manera simple la versión más adecuada en función de la tarea y del conjunto de datos utilizado.

En resumen, el *TransformerBlock* implementa la unidad funcional principal de la arquitectura ViT, capaz de replicarse las veces que sean necesarias.

6.2.3. Vit con CLS

En esta sección se presenta la clase principal del modelo *Vision Transformer*, basada en la arquitectura ViT original [6]. Esta variante, además de implementar los dos módulos ya explicados, utiliza un token especial denominado `[CLS]` como representación global de la imagen, que será utilizado para realizar la clasificación final. El desarrollo de esta clase se puede ver en A.4.

El modelo comienza aplicando la etapa de *patch embedding* a la imagen de entrada, transformándola en una secuencia de vectores de las dimensiones requeridas. A esta secuencia se le añade un token especial, denominado `[CLS]`, que se inicializa como un parámetro entrenable

con valores iniciales aleatorios, gracias a *nn.Parameter*. Este token se encargará de obtener la información global de la imagen contenida en los demás tokens, es decir, en los patches.

El token `[CLS]` se concatena a la secuencia antes de introducir la codificación posicional, la cual también se define como un parámetro entrenable y valores iniciales aleatorios. Este vector es el encargado de conservar información sobre el orden espacial de los patches, compensando el hecho de que el Transformer no posee estructura espacial implícita.

A continuación, la secuencia completa (incluyendo el token `[CLS]`) se procesa a través de un conjunto de bloques Transformer idénticos gracias a *nn.Sequential*, definidos anteriormente mediante la clase `TransformerBlock`. El número de bloques se especifica mediante el parámetro `num_layers`, permitiendo ajustar la profundidad del modelo.

Una vez pasada la secuencia por la pila de bloques, se extrae únicamente el token `[CLS]`, que se supone contiene la representación global de la imagen. Este vector se normaliza mediante una capa de `LayerNorm` y finalmente se proyecta a las clases posibles mediante una capa lineal, dando lugar a la predicción final del modelo.

6.2.4. Vit con Mean Pooling

A parte de desarrollar la arquitectura original basada en el uso de un token `[CLS]`, se ha implementado una variante del modelo ViT que utiliza una estrategia alternativa: el *mean pooling*. La implementación de esta estrategia tiene como objetivo mitigar posibles problemas de generalización asociados al uso del token `[CLS]`, especialmente en contextos con pocos datos o alta variabilidad en las imágenes, donde dicho token puede no capturar de manera correcta la representación global de la imagen.

La estructura general de esta clase es similar a la anterior, con la diferencia principal de que no se utiliza el token `[CLS]`. En su lugar, tras aplicar la codificación posicional sobre los embeddings de los patches, se pasa directamente al conjunto de bloques codificador. Una vez procesados, se aplica una operación de *mean pooling* sobre la secuencia de salida para obtener una única representación global, sobre la que se realizan la normalización y la clasificación final.

De esta manera, se utiliza toda la secuencia de tokens para generar la representación global, en lugar de confiar en un único vector aprendido, ayudando en casos donde la información puede estar más dispersa.

El resto de componentes del modelo se mantienen sin cambios: se utilizan los mismos bloques Transformer, la misma codificación posicional aprendida y una capa de `LayerNorm` previa a la proyección final por la capa lineal de clasificación. Esta consistencia facilita la comparación entre ambas variantes y permite analizar el impacto que tiene esta alternativa. La implementación completa puede consultarse en A.5.

6.3. Entrenamiento

6.3.1. Dataset para HDF5

Dataset

Inicialmente, los datos se cargan utilizando la función *torchvision.datasets.ImageFolder*, que permite leer directamente imágenes almacenadas en una estructura de carpetas organizada por clases. Esta función asume que cada subcarpeta representa una clase distinta, y asigna como etiqueta de clase el nombre de dicha subcarpeta a todas las imágenes contenidas en ella. El resultado es un conjunto de datos en formato tensorial ya preparado para ser utilizado en PyTorch, lo que facilita enormemente la carga inicial de los datos. Se pueden utilizar las transformaciones que se deseen, desde *transforms.toTensor()* para convertir a tensores hasta transformaciones utilizadas para realizar *Data Augmentation*, como por ejemplo *transforms.ColorJitter()*.

No obstante, el uso directo de `ImageFolder` en los bucles de entrenamiento puede resultar poco óptimo en términos de rendimiento. Para mejorar la eficiencia en la carga de datos, especialmente en configuraciones donde el cuello de botella se encuentra en la lectura desde disco, se ha optado por transformar los conjuntos de datos a formato *HDF5* (Hierarchical Data Format version 5).

El formato HDF5 permite almacenar grandes volúmenes de datos estructurados de forma jerárquica, lo que no sólo reduce el número de accesos a disco, sino que también permite organizar imágenes y etiquetas con las transformaciones aplicadas y descomprimidas. Aunque los archivos HDF5 suelen ocupar más espacio en disco que otros formatos como JPEG o PNG, presentan una clara ventaja en términos de velocidad de lectura y acceso secuencial a los datos, lo cual es fundamental durante el entrenamiento de modelos de aprendizaje profundo [57].

Como se puede ver en A.6, la clase `HDF5Dataset` esta desarrollada directamente desde PyTorch, integrándose con facilidad en los *DataLoader* para entrenamiento y validación.

Cada muestra es devuelta como un par de tensores (`image`, `label`) listos para ser usados en el modelo. La conversión explícita a tensores se realiza en cada llamada a *___getitem___*. Se incluye un método auxiliar *close()* para cerrar el archivo HDF5 de forma segura al finalizar su uso.

Dataset v2

Con el objetivo de aplicar técnicas de *Data Augmentation* dinámicas en vez de estáticas durante el entrenamiento, se ha desarrollado una versión extendida de la clase, denominada `HDF5Datasetv2`. Esta nueva variante también incluye soporte para las tres divisiones distintas (entrenamiento, validación y prueba), así como transformaciones que se aplican en tiempo real (véase A.7).

Esta implementación resulta especialmente útil para aumentar la variabilidad del conjunto de entrenamiento sin necesidad de almacenar múltiples versiones de las mismas imágenes. Así, se mejora la generalización del modelo, mientras se mantiene una estructura de datos eficiente.

Generación del archivo hdf5

Para poder utilizar esta clase, es necesario generar previamente un archivo en formato HDF5 que contenga las imágenes y sus correspondientes etiquetas. Para ello, se parte de los datos organizados en carpetas por clase y se utiliza la función `torchvision.datasets.ImageFolder`, aplicando las transformaciones deseadas sobre cada imagen.

Una vez cargados, los datos se recorren secuencialmente y se almacenan los vectores en formato *numpy* en un archivo tipo *.hdf* a través de la librería *h5py*, como se puede ver en A.8.

6.3.2. Bucle de entrenamiento

El fragmento A.9 muestra la rutina de entrenamiento y evaluación utilizada en todos los experimentos realizados. El entrenamiento se desarrolla durante un número definido de épocas (`num_epochs`). En cada una de ellas, se alterna una fase de entrenamiento (*model.train()*) con una de evaluación (*model.eval()*).

Durante el aprendizaje, se mide la función de pérdida y el porcentaje de aciertos sobre el conjunto de entrenamiento. Posteriormente, en modo evaluación, sin calcular gradientes (mediante *torch.no_grad()*), se evalúa el rendimiento del modelo sobre los datos de prueba.

Se almacenan las métricas (`loss`, `accuracy`, `learning rate`) en listas para su posterior análisis y representación gráfica. El planificador de tasa de aprendizaje (`scheduler`) se puede actualizar en diferentes puntos estratégicos según su tipo, en este caso, para cada lote hay una actualización.

Una vez se han obtenido las configuraciones óptimas para los modelos, se ejecuta de nuevo el bucle con nuevo código. Al final de cada época se añade el fragmento A.10.

Este código es el encargado de guardar el estado del mejor modelo en términos de precisión sobre el conjunto de test. Para evitar almacenamientos prematuros, la evaluación comienza a partir de la época 70 (modificable según el modelo). Si la precisión del modelo en la época actual supera respecto al mejor valor anterior, se actualiza el estado del modelo y se guarda mediante `torch.save`, indicando en pantalla la información relevante sobre dicha mejora.

Asimismo, se implementa un mecanismo de *early stopping* que interrumpe automáticamente el entrenamiento, si no se observa ninguna mejora en un número determinado de épocas consecutivas, definido por la variable `early_stop_patience`.

6.4. Modelos implementados

Una vez explicadas en detalle las distintas estructuras desarrolladas y cómo se realiza el entrenamiento, en esta sección se comenta cuáles han sido las configuraciones utilizadas para cada uno de los conjuntos de datos. Cada elección se ha realizado considerando las características propias de cada experimento, información adicional que se proporciona en los Trabajos de Fin de Grado utilizados como referencia, así como el rendimiento observado durante la fase de entrenamiento y validación.

6.5. Radiografías de tórax (CXR)

Este conjunto de datos presenta ciertas particularidades que, si bien no dificultan tanto el entrenamiento como otros, sí requieren aplicar medidas específicas de preprocesamiento y regularización. Las imágenes son más homogéneas que las resonancias, aunque presentan diferencias en contraste y nitidez entre clases, lo que puede inducir sesgos en el modelo, si no se abordan adecuadamente.

Para mejorar la robustez y aumentar la diversidad del conjunto de entrenamiento, se ha optado por utilizar transformaciones de *Data Augmentation* dinámicas. Estas transformaciones se aplican en tiempo real, esto es, durante el entrenamiento mediante la versión extendida del dataset HDF5 descrita previamente. Las operaciones empleadas han sido:

- `RandomHorizontalFlip()`
- `RandomAdjustSharpness(sharpness_factor=1.5, p=0.3)`
- `RandomAutocontrast(p=0.2)`
- `RandomRotation(degrees=5)`
- `RandomPerspective(distortion_scale=0.1, p=0.3)`

En cuanto a la función de pérdida, se ha mantenido la entropía cruzada ponderada por clases y se ha aplicado label smoothing al 5 %. Esta configuración contribuye a paliar los efectos del desbalanceo entre clases y a mejorar la calibración de las predicciones, evitando la sobreconfianza.

El optimizador seleccionado ha sido `AdamW`, con un *learning rate* inicial de $5 \cdot 10^{-4}$ y un *weight decay* de $5 \cdot 10^{-3}$. Como política de ajuste del ritmo de aprendizaje, se ha empleado una planificación cosenoidal con warm-up y con un 5 % de los pasos de entrenamiento dedicados al calentamiento.

Respecto a la arquitectura del modelo, se ha optado por una variante del ViT con *mean pooling*, en este caso con una estructura más profunda (4 bloques Transformer). La entrada consiste en imágenes RGB de 256×256 píxeles, que se dividen en patches de 8×8, generando

1024 tokens por imagen. A diferencia del caso anterior, aquí sí se ha utilizado una proyección lineal para transformar cada patch en un vector de dimensión 192.

El modelo incluye 8 cabezas de atención por bloque, una red *feed-forward* de dimensión 576 y una tasa de `dropout` de 0.1. Asimismo, se ha mantenido la conexión residual dentro de los bloques *feed-forward* dado que, gracias a la gran cantidad de data augmentation, un valor mayor empeoraba los resultados.

```
model = ViTWithMeanPooling(img_size=256, patch_size=8, in_channels
    =3, emb_dim=8*8*3, num_heads=8, mlp_dim=8*8*3*3, num_layers=4,
    num_classes=3, dropout=0.1)
```

Listing 6.2: Modelo.

```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
ViTWithMeanPooling                       [1, 3]                    196,608
├─PatchEmbeddingv3: 1-1                  [1, 1024, 192]            --
│    └─Sequential: 2-1                   [1, 1024, 192]            --
│    │    └─Rearrange: 3-1               [1, 1024, 192]            --
│    │    └─LayerNorm: 3-2               [1, 1024, 192]            384
│    │    └─Linear: 3-3                  [1, 1024, 192]            37,056
│    │    └─LayerNorm: 3-4               [1, 1024, 192]            384
├─Sequential: 1-2                        [1, 1024, 192]            --
│    └─TransformerBlock: 2-2             [1, 1024, 192]            --
│    │    └─MultiheadAttention: 3-5      [1, 1024, 192]            148,224
│    │    └─LayerNorm: 3-6               [1, 1024, 192]            384
│    │    └─Sequential: 3-7              [1, 1024, 192]            221,952
│    │    └─LayerNorm: 3-8               [1, 1024, 192]            384
│    └─TransformerBlock: 2-3             [1, 1024, 192]            --
│    │    └─MultiheadAttention: 3-9      [1, 1024, 192]            148,224
│    │    └─LayerNorm: 3-10              [1, 1024, 192]            384
│    │    └─Sequential: 3-11             [1, 1024, 192]            221,952
│    │    └─LayerNorm: 3-12              [1, 1024, 192]            384
│    └─TransformerBlock: 2-4             [1, 1024, 192]            --
│    │    └─MultiheadAttention: 3-13     [1, 1024, 192]            148,224
│    │    └─LayerNorm: 3-14              [1, 1024, 192]            384
│    │    └─Sequential: 3-15             [1, 1024, 192]            221,952
│    │    └─LayerNorm: 3-16              [1, 1024, 192]            384
│    └─TransformerBlock: 2-5             [1, 1024, 192]            --
│    │    └─MultiheadAttention: 3-17     [1, 1024, 192]            148,224
│    │    └─LayerNorm: 3-18              [1, 1024, 192]            384
│    │    └─Sequential: 3-19             [1, 1024, 192]            221,952
│    │    └─LayerNorm: 3-20              [1, 1024, 192]            384
├─LayerNorm: 1-3                         [1, 192]                  384
├─Linear: 1-4                            [1, 3]                    579
==========================================================================================
Total params: 1,719,171
Trainable params: 1,719,171
Non-trainable params: 0
Total mult-adds (Units.MEGABYTES): 0.93
==========================================================================================
Input size (MB): 0.79
Forward/backward pass size (MB): 42.47
Params size (MB): 3.72
Estimated Total Size (MB): 46.97
==========================================================================================
```

Figura 6.2: Resumen del modelo.

6.6. Resonancias magnéticas de cerebro (MRI)

Este conjunto de datos presenta varios retos que dificultan el entrenamiento. En primer lugar, se tratan de imágenes con un alto nivel de ruido, lo cual complica la extracción de características relevantes. Existe una gran variabilidad en cuanto al ángulo de toma de las imágenes, encontrándose cortes axiales, sagitales y coronales. Esta diversidad añade un componente de complejidad que solo podría abordarse de forma efectiva con una gran cantidad de datos, algo que no se cumple en este caso, ya que el conjunto disponible apenas supera las 2000 muestras.

Para intentar solucionar los problemas mencionados, se han incorporado diversas estrategias. En primer lugar, se ha utilizado únicamente la transformación `transforms.RandomRotation(5)` de manera estática como técnica de *Data Augmentation*, pues los datos cuentan con gran variabilidad. Aunque se han probado múltiples combinaciones, que incluían volteos horizontales y verticales, jitter de color y otras transformaciones geométricas, no han llegado a mejorar los resultados.

Luego, se ha utilizado la función de pérdida `CrossEntropyLoss`, incluyendo tanto una ponderación por clase como un suavizado de etiquetas (*label smoothing*) del 10 %. Esta configuración permite reducir el impacto del desbalanceo entre clases y disminuir la sobreconfianza del modelo en sus predicciones.

Como optimizador se ha empleado `Adam` con una tasa de aprendizaje inicial de 0.001. Para regular el ritmo de aprendizaje a lo largo del entrenamiento, se ha utilizado una política basada en planificación cosenoidal con fase de *warm-up*. Se ha definido un 5 % del total de pasos de entrenamiento como periodo de *warm-up*, lo que posibilita iniciar el entrenamiento de manera más estable antes de iniciar el decaimiento progresivo del *learning rate*.

En cuanto a la arquitectura del modelo, se ha optado por la variante del Vision Transformer con *mean pooling*. El modelo recibe imágenes de entrada de 128×128 píxeles con tres canales que divide en patches de 8×8, generando un total de 256 tokens por imagen. Cada patch se aplanó y se usó directamente como vector de entrada, sin aplicar ninguna proyección lineal adicional, por tanto, la dimensión del embedding ha sido de 192. El número de cabezas de atención se fijó en 8, y la red *feed-forward* interna de cada bloque Transformer fue de tamaño 384. En total, se apilaron 3 bloques Transformer y se fijó una tasa de `dropout` de 0.1. La salida final del modelo se proyecta a 4 clases correspondientes a las categorías del conjunto de datos.

```
1    model = ViTWithMeanPooling(img_size=128, patch_size=8, in_channels
         =3, emb_dim=3*8*8, num_heads=8, mlp_dim=3*8*8*2, num_layers=3,
         num_classes=4, dropout=0.1)
```

Listing 6.3: Modelo.

```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
ViTWithMeanPooling                       [1, 4]                    49,152
├─PatchEmbedding: 1-1                    [1, 256, 192]             37,440
│    └─Rearrange: 2-1                    [1, 256, 192]             --
├─Sequential: 1-2                        [1, 256, 192]             --
│    └─TransformerBlock: 2-2             [1, 256, 192]             --
│    │    └─MultiheadAttention: 3-1      [1, 256, 192]             148,224
│    │    └─LayerNorm: 3-2               [1, 256, 192]             384
│    │    └─Sequential: 3-3              [1, 256, 192]             148,032
│    │    └─LayerNorm: 3-4               [1, 256, 192]             384
│    └─TransformerBlock: 2-3             [1, 256, 192]             --
│    │    └─MultiheadAttention: 3-5      [1, 256, 192]             148,224
│    │    └─LayerNorm: 3-6               [1, 256, 192]             384
│    │    └─Sequential: 3-7              [1, 256, 192]             148,032
│    │    └─LayerNorm: 3-8               [1, 256, 192]             384
│    └─TransformerBlock: 2-4             [1, 256, 192]             --
│    │    └─MultiheadAttention: 3-9      [1, 256, 192]             148,224
│    │    └─LayerNorm: 3-10              [1, 256, 192]             384
│    │    └─Sequential: 3-11             [1, 256, 192]             148,032
│    │    └─LayerNorm: 3-12              [1, 256, 192]             384
├─LayerNorm: 1-3                         [1, 192]                  384
├─Linear: 1-4                            [1, 4]                    772
==========================================================================================
Total params: 978,820
Trainable params: 978,820
Non-trainable params: 0
Total mult-adds (Units.MEGABYTES): 0.45
==========================================================================================
Input size (MB): 0.20
Forward/backward pass size (MB): 5.90
Params size (MB): 1.79
Estimated Total Size (MB): 7.89
==========================================================================================
```

Figura 6.3: Resumen del modelo.

6.7. Secciones transversales de tomografías de coherencia óptica (OCT)

Este conjunto se caracteriza por contener imágenes en escala de grises. Presentan una gran nitidez estructural, pero también una elevada similitud entre clases, lo que complica la separación entre categorías.

A diferencia de los conjuntos anteriores, estas imágenes tienen un solo canal de entrada, cosa que se ha tenido en cuenta en la arquitectura. Igualmente, se ha utilizado el mismo *Data Augmentation* dinámico comentado para el primer conjunto (CXR), pues esta configuración también ha presentado grandes mejoras de regularización para este caso.

Para mitigar el sobreajuste y mejorar la generalización del modelo, se ha empleado label smoothing con un valor del 5 %, así como una ponderación por clases en la función de pérdida `CrossEntropyLoss`. En cuanto al optimizador, se ha utilizado `Adam` con una tasa de aprendizaje inicial de 0.001 y un `weight decay` de 10^{-3}.

Como política de ajuste dinámico del ritmo de aprendizaje, se ha utilizado una planificación cosenoidal con *warm-up* del 5 % de los pasos totales. Esta estrategia permite un

inicio suave en el entrenamiento y un descenso progresivo del *learning rate*, estabilizando la convergencia.

Respecto al modelo, se ha utilizado una arquitectura basada en Vision Transformer estándar, con token `[CLS]` en vez de *mean pooling*. La entrada se divide en patches de 8×8, lo que genera 1024 tokens por imagen. A pesar de tratarse de imágenes en escala de grises, se ha proyectado cada patch a un vector de dimensión 192 (como si tuviera 3 canales) para mantener la coherencia con las arquitecturas previas y aprovechar configuraciones ya probadas.

El modelo está compuesto por 3 bloques Transformer con 8 cabezas de atención, una red *feed-forward* de dimensión 384 y una tasa de `dropout` de 0.2, ligeramente superior a los anteriores para contrarrestar la menor variabilidad del dataset.

```
model = ViT(img_size=256, patch_size=8, in_channels=1, emb_dim
    =8*8*3, num_heads = 8, mlp_dim=8*8*2*2, num_layers=3,
    num_classes=4, dropout=0.2).to(device)
```

Listing 6.4: Modelo.

```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
ViT                                      [1, 4]                    196,992
├─PatchEmbeddingv3: 1-1                  [1, 1024, 192]            --
│    └─Sequential: 2-1                   [1, 1024, 192]            --
│    │    └─Rearrange: 3-1               [1, 1024, 64]             --
│    │    └─Linear: 3-2                  [1, 1024, 192]            12,480
│    │    └─LayerNorm: 3-3               [1, 1024, 192]            384
├─Sequential: 1-2                        [1, 1025, 192]            --
│    └─TransformerBlock: 2-2             [1, 1025, 192]            --
│    │    └─MultiheadAttention: 3-4      [1, 1025, 192]            148,224
│    │    └─LayerNorm: 3-5               [1, 1025, 192]            384
│    │    └─Sequential: 3-6              [1, 1025, 192]            98,752
│    │    └─LayerNorm: 3-7               [1, 1025, 192]            384
│    └─TransformerBlock: 2-3             [1, 1025, 192]            --
│    │    └─MultiheadAttention: 3-8      [1, 1025, 192]            148,224
│    │    └─LayerNorm: 3-9               [1, 1025, 192]            384
│    │    └─Sequential: 3-10             [1, 1025, 192]            98,752
│    │    └─LayerNorm: 3-11              [1, 1025, 192]            384
│    └─TransformerBlock: 2-4             [1, 1025, 192]            --
│    │    └─MultiheadAttention: 3-12     [1, 1025, 192]            148,224
│    │    └─LayerNorm: 3-13              [1, 1025, 192]            384
│    │    └─Sequential: 3-14             [1, 1025, 192]            98,752
│    │    └─LayerNorm: 3-15              [1, 1025, 192]            384
├─LayerNorm: 1-3                         [1, 192]                  384
├─Linear: 1-4                            [1, 4]                    772
==========================================================================================
Total params: 954,244
Trainable params: 954,244
Non-trainable params: 0
Total mult-adds (Units.MEGABYTES): 0.31
==========================================================================================
Input size (MB): 0.26
Forward/backward pass size (MB): 23.61
Params size (MB): 1.25
Estimated Total Size (MB): 25.13
==========================================================================================
```

Figura 6.4: Resumen del modelo.

6.8. Explicabilidad

Como ya se ha comentado, además de realizar el diseño y entrenamiento de los modelos, se ha incorporado una técnica de explicabilidad, el *ViT-ReciproCAM*, con el objetivo de generar mapas de saliencia que ayuden en la interpretación.

Para implementar este enfoque, es necesario dividir el modelo ViT en dos bloques funcionales diferenciados: $\mathcal{G}$ y $\mathcal{H}$. Según se ha explicado en la Sección 3.4.3, la parte $\mathcal{G}$ corresponde a las capas del modelo que generan los tokens de representación a partir de la imagen, incluyendo el patch embedding, la codificación posicional y los bloques Transformer. La normalización final y la cabeza de clasificación forman parte de $\mathcal{H}$, ya que se aplican únicamente tras la agregación de los tokens mediante *mean pooling* o el uso del token `[CLS]`.

Esta separación se implementa añadiendo dos métodos adicionales en las dos clases que definen las dos versiones del modelo ViT implementadas A.11 A.12. Uno de ellos se encarga de obtener los *tokens* intermedios justo al final de $\mathcal{G}$, y el otro toma dichos tokens como entrada para procesarlos a través de $\mathcal{H}$. Esta división no interfiere con el entrenamiento, ni con la carga de pesos desde archivos `.pth`, por lo que puede añadirse tras el entrenamiento sin afectar al comportamiento del modelo.

Aunque para ambas versiones, las funciones realizan la misma divisón teórica, hay diferencia en la dimensión de la salida del método `forward_features`. En el caso de utilizar *mean pooling*, la salida es un tensor de dimensiones `[B, T, D]`, correspondiente a los T tokens generados a partir de los *patches* de la imagen. En cambio, en la versión original del ViT que utiliza el token `[CLS]`, la salida es de dimensiones `[B, T+1, D]`, ya que se incluye un token adicional al principio de la secuencia. Esto se tiene que tener en cuenta a la hora de realizar el enmascaramiento.

Una vez extraídos los tokens, se realiza un enmascaramiento local sobre ellos con el objetivo de observar cómo varía la puntuación del modelo al eliminar información de zonas específicas. Para ello se utiliza una función que enmascara un bloque de 3×3 tokens centrado en una posición determinada A.13.

La función toma como entrada los tokens generados por la función *forward_features* correspondiente. Para cada celda central especificada por fila y columna, se enmascara sus vecinos inmediatos (formando un bloque 3×3), estableciendo su valor a cero (o cualquier otro valor definido por el parámetro `fill_value`). Existe también una variante que salta el primer token, en caso de que el modelo utilice un token `[CLS]`, la cual se basa en sumar 1 para pasar por alto el token sin modificare:

```
1     idx = 1 + r * num_patches + c  # +1 para saltar el [CLS]
```

Listing 6.5: Linea diferente para versión CLS.

A partir de estas modificaciones, se procede a generar el mapa de saliencia. Para cada posición del grid, se calcula la diferencia entre la puntuación original del modelo y la obtenida tras enmascarar dicha región. Este proceso se repite para todos los patches de la imagen. Un ejemplo de cómo se aplica esta técnica tras obtener la predicción del modelo esta en A.14.

Este procedimiento permite obtener un mapa que indica, de forma visual, qué zonas afectan más a la predicción del modelo. La transparencia del mapa (`alpha`) se puede ajustar para facilitar la interpretación. Para ello, se utiliza una interpolación bilineal para adaptar la resolución del mapa al tamaño original de la imagen. El resultado es un mapa de saliencia que resalta las regiones clave utilizadas por el modelo para tomar su decisión.

Capítulo 7

Resultados

En este capítulo, se resumen los datos numéricos finales, como los resultados obtenidos durante el proceso de evaluación. El análisis se centra en el rendimiento alcanzado por cada modelo sobre su correspondiente conjunto de datos, acompañando la comparación con las redes convolucionales desarrolladas en los trabajos previos.

Para ello, se presentan diversas métricas de rendimiento para evaluar y comparar los modelos, teniendo en cuenta los resultados explicados por los antiguos compañeros.

Asimismo, se incluyen visualizaciones de mapas de saliencia generados con el enfoque desarrollado en este proyecto, los cuales se contrastan con los mapas basados en gradientes utilizados en los trabajos anteriores.

7.1. Resonancias magnéticas de cerebro (MRI)

En la Tabla 7.1 se muestran los resultados obtenidos por Arranz en el TFG anterior [15] para distintos modelos basados en arquitecturas CNN. Se observan cinco versiones distintas que emplean combinaciones de tamaño de imagen, transformaciones, número de filtros, tasa de *dropout* y número de capas. Aunque algunas versiones alcanzan altas precisiones en el conjunto de entrenamiento (v2 y v3 superan el 90 %), aunque la precisión en el conjunto de test no supera el 60 %, lo que indica un gran problema de sobreajuste.

Para este conjunto se ha empleado una arquitectura ViT con *mean pooling* y estrategias de regularización como *data augmentation*, ponderación de clases y *label smoothing*, lo que ha permitido mejorar sustancialmente la capacidad de generalización.

La Figura 7.2 muestra la matriz de confusión absoluta en el conjunto de test, mientras que la Figura 7.3 muestra la versión normalizada por filas. Se observa que el modelo consigue una clasificación precisa en todas las clases, con un rendimiento especialmente alto en la clase 3, siendo más difusos los resultados en las demás clases.

Versión	Parámetros	Capas	AccTrain	AccTest
v1	img_size : 56, transform: NO channels_ini : 3, Initial_filters : 32, batch_size : 64, f_perdidas : nn.CrossEntropyLoss(), Learning_rate: 0.005	[3, 32] [32, 64]	0.8087	0.4036
v2	img_size : 128, trasform: RandomVertical + RandomHorizontal Channels_ini : 3, initial_filters : 8, batch_size : 64, dropout_rate : 0.5, f_perdidas : nn.CrossEntropyLoss(), Learning_rate: 0.005	[3, 8] [8, 16] [16, 32] [32, 64]	0.8902	0.5964
v3	img_size : 256, transform: NO channels_ini : 3, initial_filters : 8, batch_size": 64, dropout_rate : 0.25, f_perdidas: nn.CrossEntropyLoss(), learning_rate: 0.005	[3, 16] [16, 32] [32, 64]	0.9129	0.5812
v4	img_size: 128, transform: RandomVertial + AdjustSharpness + RandomHorizontal channels_ini : 3, initial_filters : 32, batch_size : 64, dropout_rate : 0.25, f_perdidas : nn.CrossEntropyLoss(), learning_rate: 0.005	[3, 32] [32, 64]	0.7063	0.4289
v5	img_size : 64, transform: NO channels_ini : 3, initial_filters : 16, batch_size : 64, dropout_rat : 0.25, f_perdidas: nn.CrossEntropyLoss(), learning_rate: 0.0001	[3, 16] [16, 32]	0.4648	0.3173

Figura 7.1: Resultados de distintas versiones CNN implementadas en el TFG anterior [15].

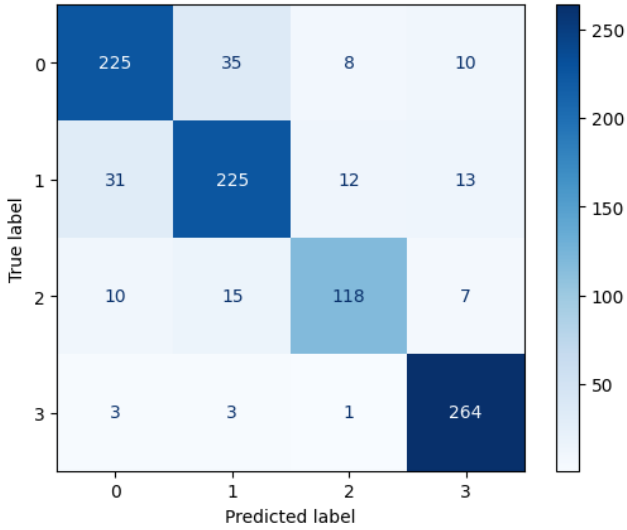


Figura 7.2: Matriz de confusión sobre el conjunto de test del modelo seleccionado.

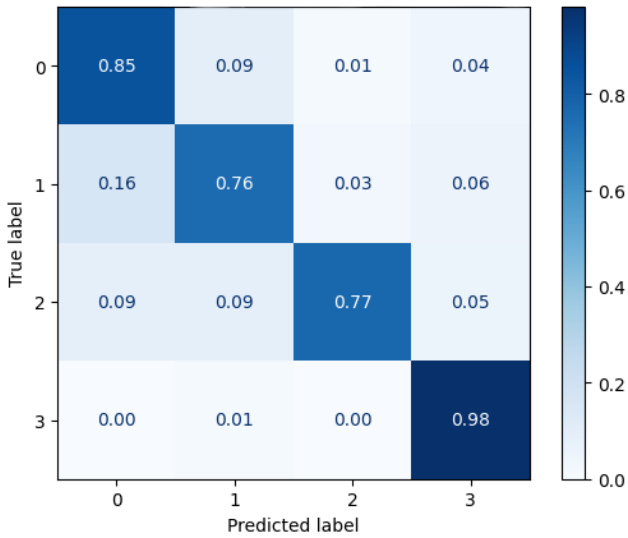


Figura 7.3: Matriz de confusión (frecuencias) sobre el conjunto de test del modelo seleccionado.

La evolución del entrenamiento se resume en la Figura 7.4. El modelo alcanza un valor mínimo de pérdida de test cercano a 0.72 y una precisión del 84.9 % en el conjunto de test durante la época 84, momento en que se guardó el modelo final. La pérdida de entrenamiento continúa disminuyendo mientras que la de test se estabiliza, lo que indica un entrenamiento

con cierto sobreajuste una vez pasadas las 40 épocas.

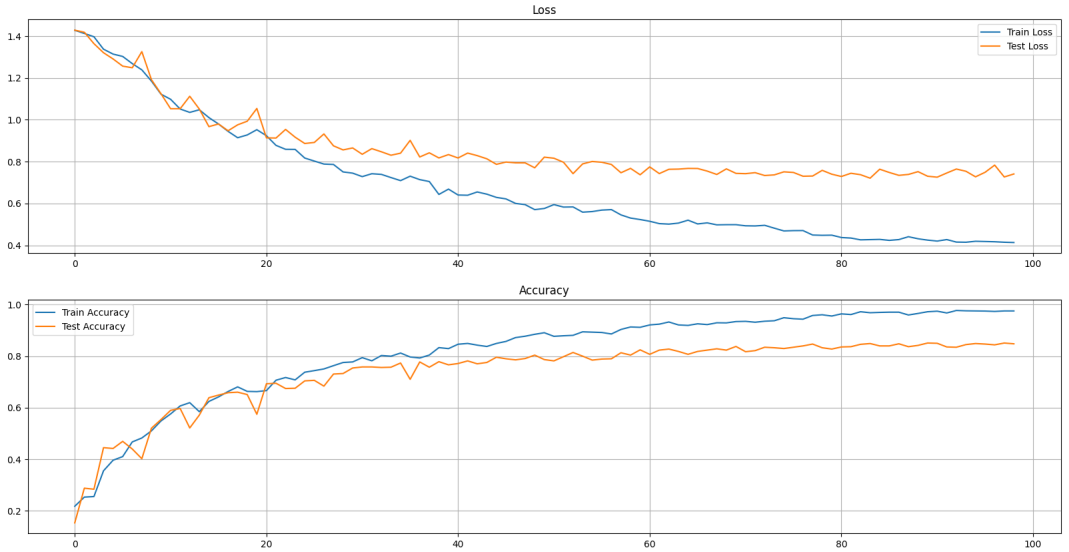


Figura 7.4: Evolución de la función de pérdida (arriba) y la precisión (abajo) durante el entrenamiento.

De esta manera, si bien todavía se mantiene dicho sobreajuste, se ha logrado una gran mejora respecto a las arquitecturas CNN del trabajo previo, incrementando en casi un 30 % la precisión sobre el conjunto de test.

Datos completos de la mejor época:

- Epoch: 84/150
- Train Loss: 0.42777 Train Accuracy: 0.96848
- Test Loss: 0.72113 Test Accuracy: 0.84898

7.2. Radiografías de tórax (RXC)

A pesar de no alcanzar las tasas de clasificación mencionadas en el trabajo anterior (con una precisión en test del 96.18 %), el modelo Vision Transformer ha mostrado un rendimiento notable, especialmente teniendo en cuenta las diferencias arquitectónicas y de enfoque.

Como se observa en la Figura 7.5, la evolución de las métricas durante el entrenamiento muestra un comportamiento estable, con un test loss que se mantiene en niveles relativamente bajos y sin indicios claros de sobreajuste severo, a pesar de la complejidad del conjunto de datos.

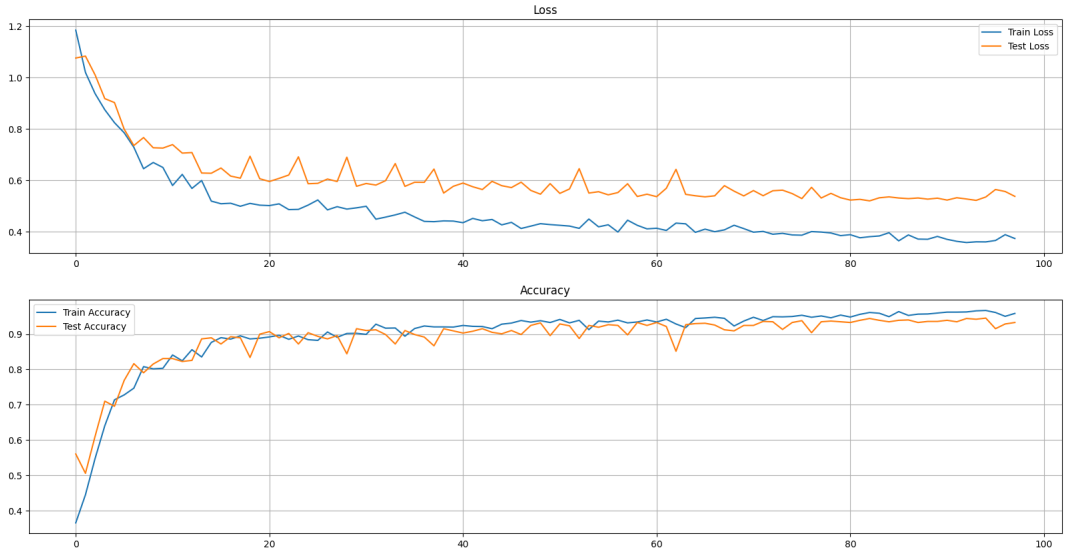


Figura 7.5: Evolución de la función de pérdida (arriba) y la precisión (abajo) durante el entrenamiento.

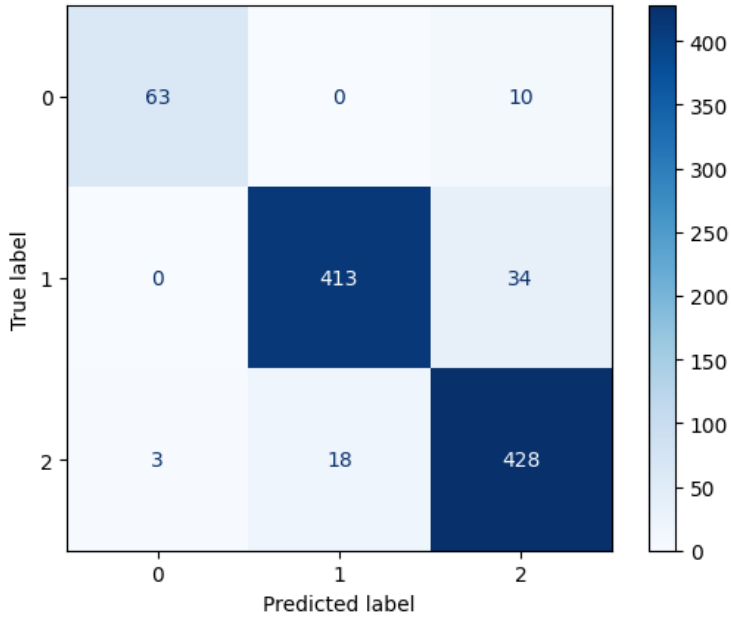


Figura 7.6: Matriz de confusión sobre el conjunto de test del modelo seleccionado.

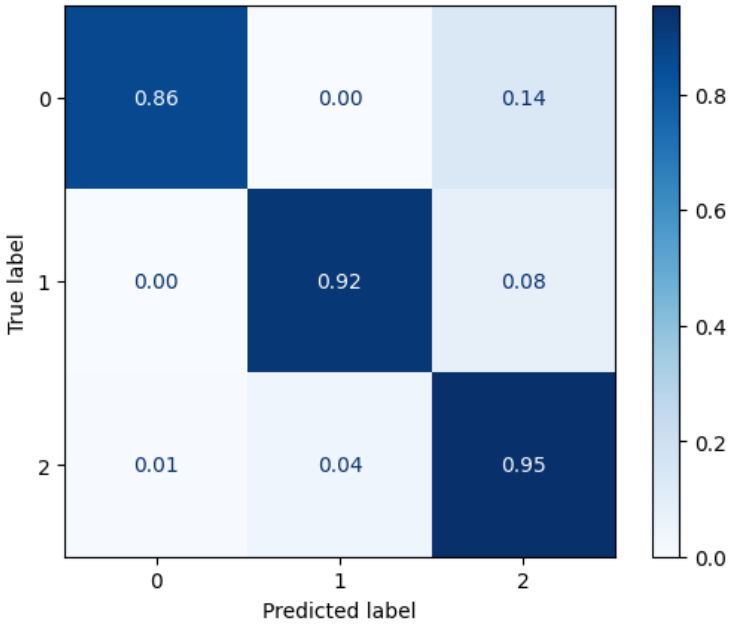


Figura 7.7: Matriz de confusión (frecuencias) sobre el conjunto de test del modelo seleccionado.

En cuanto a la distribución de errores, las matrices de confusión 7.6 y 7.7 revelan una precisión elevada en todas las clases, lo que indica un buen rendimiento general del modelo. No obstante, se observa cierta confusión entre las clases 'Normal' (1) y 'Neumonía vírica' (2), fenómeno que también ocurre en el trabajo anterior. Por otro lado, la clase 'COVID-19' (0), si bien presenta un rendimiento algo inferior al de las otras dos clases, mejora los resultados obtenidos del trabajo anterior, a pesar de mostrar cierta confusión con la clase 2.

Si bien el modelo anterior presentaba métricas globales superiores, estaba más centrado en optimizar la predicción de las clases 1 y 2. En cambio, el modelo actual busca un rendimiento equilibrado entre todas las clases, incluyendo la clase 0, que parece ser más difícil de clasificar.

En conjunto, aunque los resultados obtenidos no superan a los del trabajo anterior, se sitúan en un rango competitivo, demostrando la viabilidad del uso de modelos basados en atención en tareas de clasificación médica incluso con un número limitado de ejemplos.

Datos completos de la mejor época:

- Epoch: 83/150
- Train Loss: 0.38122 Train Accuracy: 0.96126
- Test Loss: 0.52052 Test Accuracy: 0.94427

7.3. Secciones transversales de tomografías de coherencia óptica (OTC)

En este caso, el conjunto de datos presenta un reto particular debido a su gran tamaño, implicando un coste computacional elevado. El entrenamiento con un número elevado de imágenes alarga significativamente las épocas, dificultando la exploración con arquitecturas más complejas y agrandando el problema de disponibilidad de recursos.

En la Figura 7.8 se presenta la evolución de la función de pérdida y precisión a lo largo del entrenamiento. El modelo alcanzó una precisión del 90.1 % en el conjunto de test en la época 72, con una pérdida de validación estabilizada y sin signos evidentes de sobreajuste.

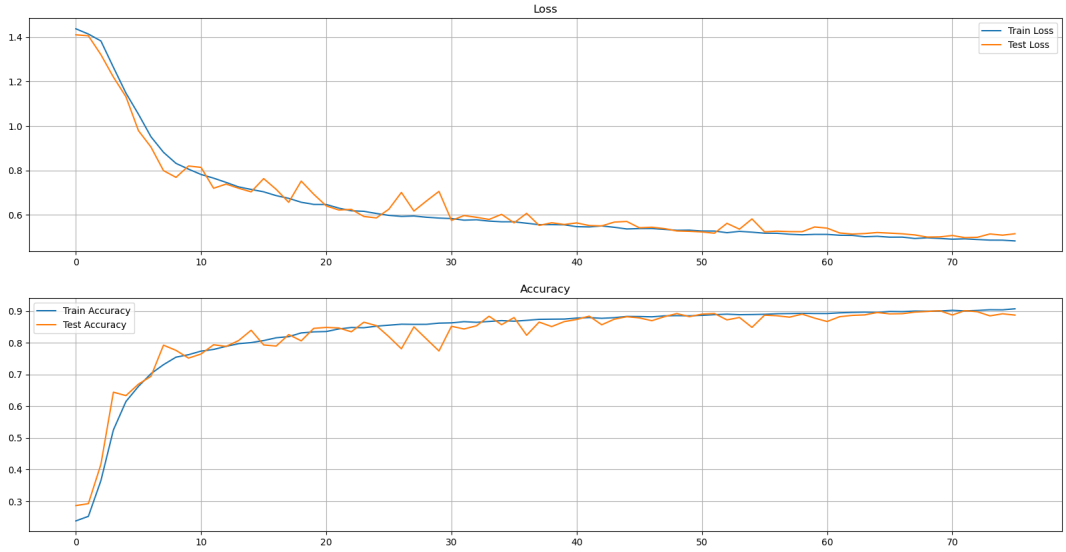


Figura 7.8: Evolución de la función de pérdida (arriba) y la precisión (abajo) durante el entrenamiento.

La evaluación del modelo se presenta en las Figuras 7.9 y 7.10, donde se muestran las matrices de confusión absoluta y normalizada, respectivamente. Se observa un comportamiento muy consistente en todas las clases, con una diagonal claramente destacada, lo que indica una excelente capacidad de generalización. No obstante, se aprecia cierta confusión entre la clase 0 y la clase 2, con un número significativo de muestras de la clase 0 clasificadas erróneamente como clase 2, lo cual podría deberse a similitudes visuales entre ambas clases.

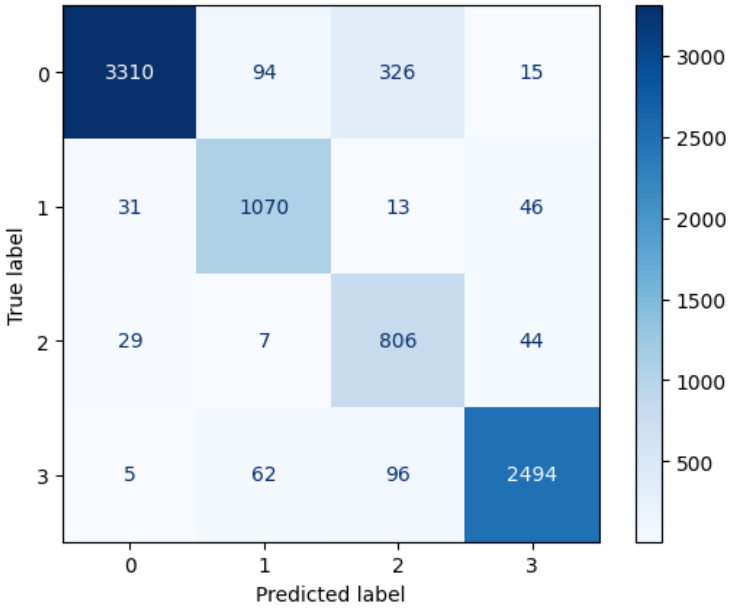


Figura 7.9: Matriz de confusión sobre el conjunto de validación del modelo seleccionado.

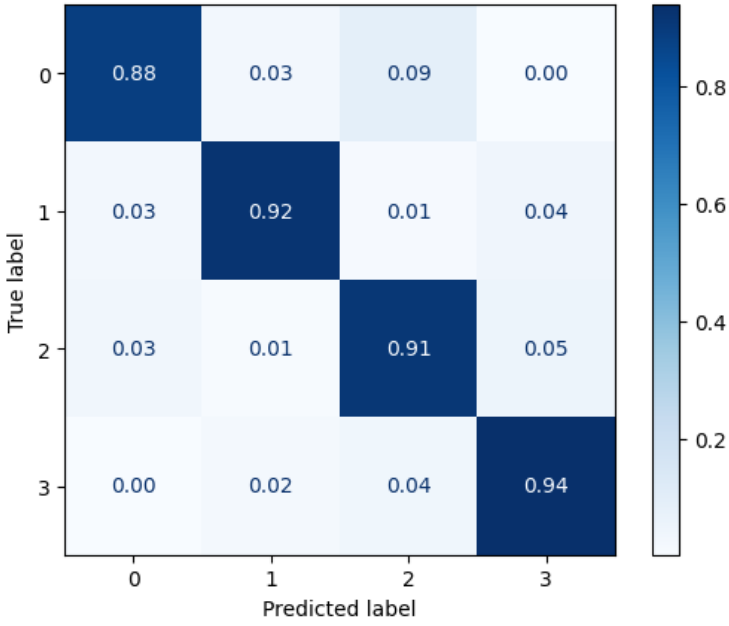


Figura 7.10: Matriz de confusión (frecuencias) sobre el conjunto de validación del modelo seleccionado.

Partición	Trabajo anterior (CNN)	Trabajo actual (ViT)
Entrenamiento	98,841 %	90,042 %
Test	96,531 %	90,116 %
Validación	96,366 %	90,110 %

Tabla 7.1: Comparativa de precisión entre el trabajo anterior [15] y el modelo ViT actual.

Tal como se recoge en la Tabla 7.1, los modelos CNN implementados en el trabajo anterior obtuvieron mejores resultados en todas las particiones. No obstante, los resultados alcanzados por el modelo ViT siguen siendo muy competitivos. Dado que se observó una buena estabilidad del modelo y una tasa de acierto elevada incluso con arquitecturas ligeras, es razonable pensar que la diferencia en rendimiento se debe más a las limitaciones computacionales (que han impedido usar arquitecturas ViT más profundas o entrenar durante más épocas) que a una incapacidad del modelo como tal. En este sentido, es probable que el uso de mayor capacidad computacional o modelos más grandes permita disminuir esta diferencia, sino incluso ponerla a su favor.

Datos completos de la mejor época:

- Epoch: 72
- Train Loss: 0.49257 Train Accuracy: 0.90042
- Test Loss: 0.49790 Test Accuracy: 0.90116

Capítulo 8

Aplicación

Tras haber obtenido y evaluado los modelos necesarios, se ha desarrollado una aplicación web con el objetivo de facilitar su uso y demostrar de forma interactiva su funcionamiento.

Esta herramienta permite al usuario cargar imágenes y obtener predicciones para cada uno de los conjuntos sin necesidad de conocimientos técnicos, ni experiencia previa con entornos de desarrollo *Python*, notebooks o bibliotecas específicas como *Pytorch* o *Pandas*.

Si bien el desarrollo de esta aplicación no es el objetivo principal de este Trabajo de Fin de Grado, su implementación aporta el valor de la construcción de un programa en producción. Sirve como una forma accesible y práctica de mostrar el rendimiento real de los modelos entrenados, sobre todo en contextos como este, donde la facilidad de uso y la interpretación de los resultados son de gran importancia para el diagnóstico médico final.

Por tanto, durante este capítulo se detallará el proceso de análisis, diseño e implementación de dicha aplicación.

8.1. Tecnologías y herramientas utilizadas

Aunque se ha detallado en el capítulo 4 las diferentes tecnologías para todo el proyecto, específicamente para el desarrollo de la aplicación se han empleado las siguientes:

8.1.1. Frontend

El lado cliente de la aplicación se ha construido utilizando tecnologías web estándar:

- **HTML5**: lenguaje de marcado utilizado para estructurar el contenido de las páginas web.

- **JavaScript**: lenguaje de programación que permite implementar funcionalidades dinámicas e interactivas en el navegador.
- **Tailwind CSS**: framework de utilidades CSS que permite construir interfaces modernas y responsivas de forma rápida y eficiente en el propio archivo HTML.

8.1.2. Backend y modelado

En el lado servidor, se han utilizado tecnologías basadas en Python:

- **Python**: lenguaje principal empleado tanto para el backend como para el desarrollo y entrenamiento de los modelos.
- **PyTorch**: biblioteca utilizada para cargar los pesos en los modelos.

8.2. Análisis

8.2.1. Requisitos

Requisitos funcionales

ID	Nombre	Descripción
RF-1	Seleccionar modelo	El sistema debe permitir al usuario seleccionar el modelo a utilizar entre los 3 disponibles.
RF-2	Subir imagen	El sistema debe permitir al usuario subir una imagen.
RF-3	Mostrar imagen	El sistema debe ser capaz de mostrar la imagen cargada por el usuario.
RF-4	Procesar imagen	El sistema debe ser capaz de procesar la imagen cargada por el usuario.
RF-5	Realizar diagnóstico	El sistema debe ser capaz de realizar el diagnóstico (clasificación) de la imagen cargada por el usuario.
RF-6	Mostrar resultado	El sistema debe ser capaz de mostrar el resultado del diagnóstico.
RF-7	Actualizar historial imagen	El sistema debe ser capaz de actualizar el historial cada vez que se realiza un diagnóstico.
RF-8	Usos consecutivos	El sistema debe permitir realizar múltiples diagnósticos consecutivos, cada uno con su propia imagen y modelo seleccionado.

Tabla 8.1: Tabla de requisitos funcionales.

Requisitos no funcionales

ID	Nombre	Descripción
RNF-1	Tiempo de procesamiento	El sistema debe tarmer menos de 10 segundos en mostrar un diagnóstico tras su petición.
RNF-2	Formato de imagen	El sistema debe permitir diferentes formatos de imagen: .jpg, .png y .jpeg.
RNF-3	Usabilidad	Un usuario con conocimientos básicos sobre navegación por la red, debe ser capaz de realizar su primer diagnóstico en menos de un minuto.
RNF-4	Lenguaje de programación	El sistema debe ser programado en Python para facilitar el uso de los modelo creados con Pytorch.

Tabla 8.2: Tabla de requisitos no funcionales.

Requisitos de información

ID	Nombre	Descripción
RI-1	Modelos disponibles	El sistema debe permitir el acceso a los modelos de diagnóstico previamente definidos y almacenados en archivos locales (.pth).
RI-2	Imagen cargada	El sistema debe almacenar temporalmente la imagen subida por el usuario durante el proceso de diagnóstico.
RI-3	Resultados de diagnóstico	El sistema debe mantener los resultados (predicción, confianza y modelo usado) tras cada ejecución, disponibles para su visualización inmediata.
RI-4	Historial en sesión	El sistema debe conservar en memoria el historial de diagnósticos realizados durante la sesión activa del usuario, incluyendo fecha, imagen y predicción.

Tabla 8.3: Tabla de requisitos de información.

8.2.2. Casos de uso

Diagrama

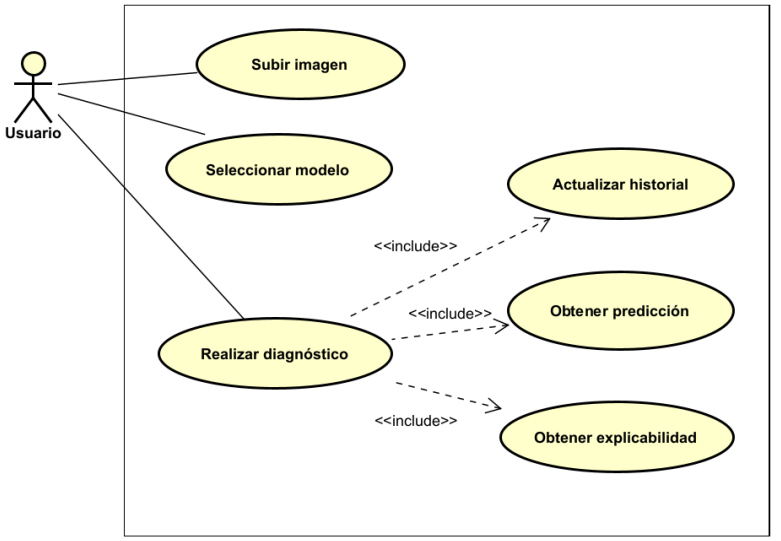


Figura 8.1: Diagrama de casos de uso.

Tablas

CU-1	**Subir imagen**
Actor	Usuario
Descripción	El usuario selecciona y sube una imagen para su posterior procesado.
Precondiciones	—
Postcondiciones	El sistema ha almacenado la imagen subida por el usuario
Flujo normal	1. El usuario sube una imagen al sistema. 2. El sistema verifica el formato de la imagen. 3. El sistema almacena la imagen. 4. El sistema muestra la imagen cargada.
Flujo alternativo	2a. El formato no es adecuado, vuelve a 1. 4a. El usuario decide subir otra imagen, vuelve a 1.

Tabla 8.4: Descripción del caso de uso 1: Subir imagen.

CU-2	**Seleccionar modelo**
Actor	Usuario
Descripción	El usuario selecciona uno de los modelos disponibles.
Precondiciones	—
Postcondiciones	El sistema ha almacenado el modelo si no lo estaba ya.
Flujo normal	1. El usuario selecciona uno de los modelos disponibles. 2. El sistema almacena el nombre del modelo para su posterior uso.
Flujo alternativo	2a. El nombre seleccionado no es válido, vuelve a 1.

Tabla 8.5: Descripción del caso de uso 2: Seleccionar modelo.

CU-3	**Realizar diagnóstico**
Actor	Usuario
Descripción	El sistema presenta procesa la imagen y presenta un diagnóstico.
Precondiciones	1. La imagen ha sido cargada en el sistema. 2. El usuario ha seleccionado un modelo.
Postcondiciones	1. El sistema presenta los resultados del diagnóstico. 2. El historial se actualiza con el nuevo diagnóstico realizado.
Flujo normal	1. El usuario solicita el diagnóstico de la imagen. 2. El sistema procesa la imagen para el modelo seleccionado. 3. El sistema carga el modelo seleccionado. 4. El sistema obtiene la predicción del modelo. 5. El sistema realiza el CU-6 - Obtener explicabilidad. 6. El sistema realiza el CU-4 - Actualizar historial. 7. El sistema prepara los resultados para su presentación. 8. El sistema presenta los resultados.
Flujo alternativo	4a. Se produce un error, se vuelve a 1. 5a. Se produce un error, el sistema muestra un mensaje, se vuelve a 1. 6a. Se produce un error, el sistema muestra un mensaje, se vuelve a 1. 8a. El usuario decide realizar otro diagnóstico, se vuelve a 1.

Tabla 8.6: Descripción del caso de uso 3: Realizar diagnóstico.

CU-4	**Actualizar historial**
Actor	Usuario
Descripción	El sistema actualiza el historial añadiendo una nueva entrada con la predicción realizada.
Precondiciones	1. El modelo seleccionado esta cargado en el sistema. 2. La imagen ha sido cargada en el sistema. 3. El sistema ha realizado la predicción de la imagen cargada.
Postcondiciones	El historial contiene una nueva entrada.
Flujo normal	1. El sistema prepara los resultados para la nueva entrada. 2. El sistema crea una nueva entrada con los nuevos resultados.
Flujo alternativo	

Tabla 8.7: Descripción del caso de uso 4: Actualizar historial.

CU-5	**Obtener explicabilidad**
Actor	Usuario
Descripción	El sistema crea un mapa de saliencia para la imagen cargada.
Precondiciones	1. El modelo seleccionado esta cargado en el sistema. 2. La imagen ha sido cargada en el sistema.
Postcondiciones	1. El mapa de saliencia queda cargado en el sistema.
Flujo normal	1. El sistema calcula el mapa de saliencia para la imagen cargada. 2. El sistema almacena el mapa de saliencia.
Flujo alternativo	1a. Se produce un error, el sistema muestra un mensaje.

Tabla 8.8: Descripción del caso de uso 5: Obtener explicabilidad.

8.3. Diseño

8.4. Patrones de Diseño Aplicados

En el desarrollo de la aplicación se han utilizado varios patrones de diseño que facilitan la organización modular, la escalabilidad y la reutilización del código. A continuación, se explican con mayor detalle los patrones implementados.

8.4.1. Singleton

El patrón *Singleton* restringe la creación de objetos pertenecientes a una clase a una sola instancia. Se accede a dicha instancia mediante un punto de acceso global. Esto permite el control centralizado de ciertos recursos evitando duplicar objetos innecesariamente.

Justificación en la aplicación: Los modelos pueden ocupar una gran cantidad de memoria debido a la gran cantidad de pesos que se necesitan almacenar, y tardan cierto tiempo en inicializarse. Para evitar que se cargue múltiples veces innecesariamente cada modelos, se usa el patrón Singleton. De esta manera, al cargar una vez un modelo, todas las peticiones futuras referentes al mismo reutilizan la misma instancia ya cargada en memoria (RAM), optimizando el rendimiento y reduciendo la latencia de respuesta.

8.4.2. Factory

El patrón *Factory* centraliza y abstrae la lógica de creación de objetos, permitiendo que el código cliente no conozca detalles concretos de implementación. Se basa en delegar la responsabilidad de instanciación a una clase dedicada (fábrica).

Justificación en la aplicación: Se utiliza una fábrica de modelos para construir dinámicamente los distintos modelos de cada conjunto de datos. El usuario selecciona un modelo mediante la interfaz, y la *Factory* se encarga de devolver la instancia correspondiente. Esto permite una alta extensibilidad del sistema: agregar un nuevo modelo no requiere cambios en el controlador, solo registrar un nuevo constructor en la fábrica.

8.4.3. Adapter

El patrón *Adapter* convierte la interfaz de una clase en otra que el sistema espera. Se utiliza comúnmente para integrar componentes que no fueron diseñados para trabajar juntos.

Justificación en la aplicación: El objeto que representa la imagen subida por el usuario a través del navegador no es directamente compatible con el modelo de PyTorch. Se implementa un adaptador que convierte el archivo recibido en una imagen transformada y normalizada, lista para ser procesada por el modelo correspondiente. Esto desacopla la lógica del servidor HTTP de los detalles internos de procesamiento de datos.

8.4.4. MVC (Modelo-Vista-Controlador)

El patrón *Modelo-Vista-Controlador* divide la aplicación en tres componentes interconectados que separan las responsabilidades:

- **Modelo:** Contiene la lógica del negocio, en este caso, los diferentes ViT, su inicialización y ejecución. También abarca la lógica de interpretabilidad (saliency maps) y el procesamiento de imágenes.
- **Vista:** Es la parte de la aplicación que interactúa con el usuario. En esta aplicación corresponde al conjunto de archivos HTML, CSS (Tailwind) y JavaScript (incluyendo Chart.js), que forman la interfaz gráfica y recogen las acciones del usuario.

- **Controlador:** Gestiona las peticiones del cliente (usuario) y coordina las acciones del modelo y la vista. Las rutas de Flask actúan como controladores, recibiendo las imágenes, seleccionando el modelo adecuado, generando predicciones, y devolviendo los resultados al cliente.

Justificación en la aplicación: Gracias al patrón MVC, la lógica de negocio (modelo) y la presentación (vista) están claramente separadas, lo que facilita la mantenibilidad, pruebas unitarias y escalabilidad. Además, permite que distintas vistas se conecten a los mismos modelos, reutilizando código de forma eficiente.

8.4.5. Diagramas

En esta sección se detalla la arquitectura de la aplicación a través de los diferentes diagramas.

Esquema general

En este primer diagrama, presentado en la Figura 8.2, se muestra el diagrama general de tipo *Uses Style* de la aplicación. Como puede observarse, la arquitectura está dividida en tres componentes principales, correspondientes a las capas del patrón MVC: **Model**, **View** y **Controller**.

Dentro de la capa *View*, se realiza una subdivisión adicional en las carpetas *Static* y *Templates*, para tener mejor división entre el HTML y la parte de JavaScript.

Puede apreciarse que la arquitectura sigue un enfoque de capas estrictas, en el que cada capa únicamente tiene conocimiento de la capa inmediatamente inferior. Esta restricción contribuye a mejorar la mantenibilidad, escalabilidad y claridad del sistema. Cabe decir que esta relación no se encuentra en los estereotipos del diagrama con el fin de no sobrecargarlo.

Clases detalladas

A continuación, se presentan los diagramas de clases detallados correspondientes a cada una de las capas del patrón MVC. Cada diagrama muestra las clases principales, sus métodos y relaciones.

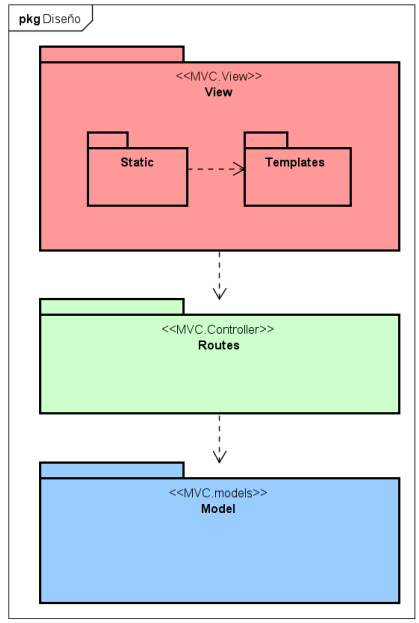


Figura 8.2: Diagrama Uses Style general.

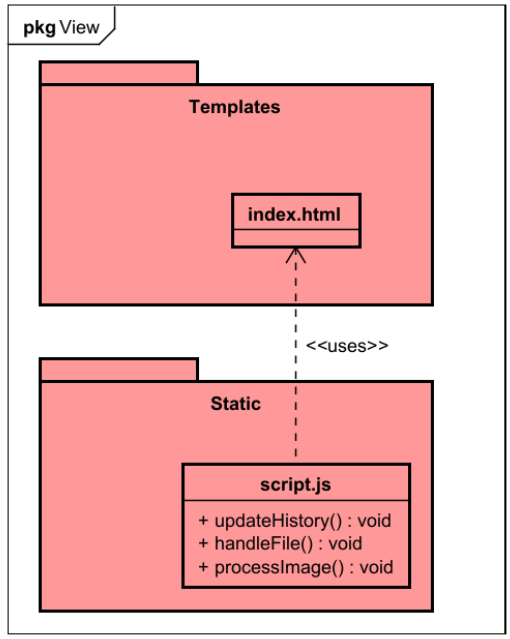


Figura 8.3: Diagrama de clases detallado de View.

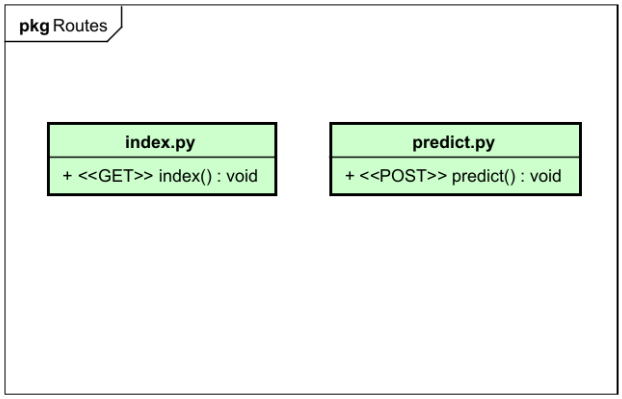


Figura 8.4: Diagrama de clases detallado de Routes.

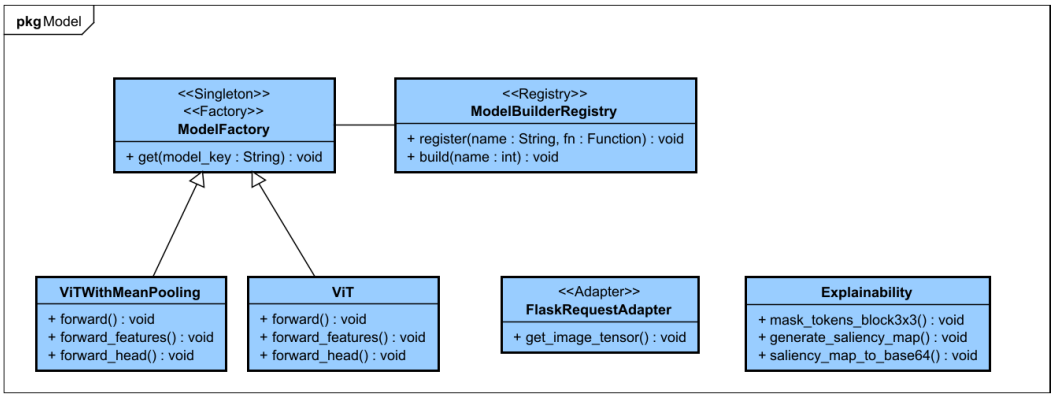


Figura 8.5: Diagrama de clases detallado de Model.

Caso de uso principal

Dado que el caso de uso **CU-3 Realizar diagnóstico**, no solo es el principal sino también el más complejo, se detallan tanto el diagrama *Uses Style* como el de secuencia. La primera Figura 8.6 corresponde al diagrama *Uses Style*, el que se observa todas las relaciones entre las capas. Por otro lado, en la Figura 8.7 se representa el diagrama de secuencia asociado, en el que se describe la interacción entre los distintos componentes del sistema durante la ejecución de este caso de uso clave, a parte de referenciar a otros casos de uso contenidos.

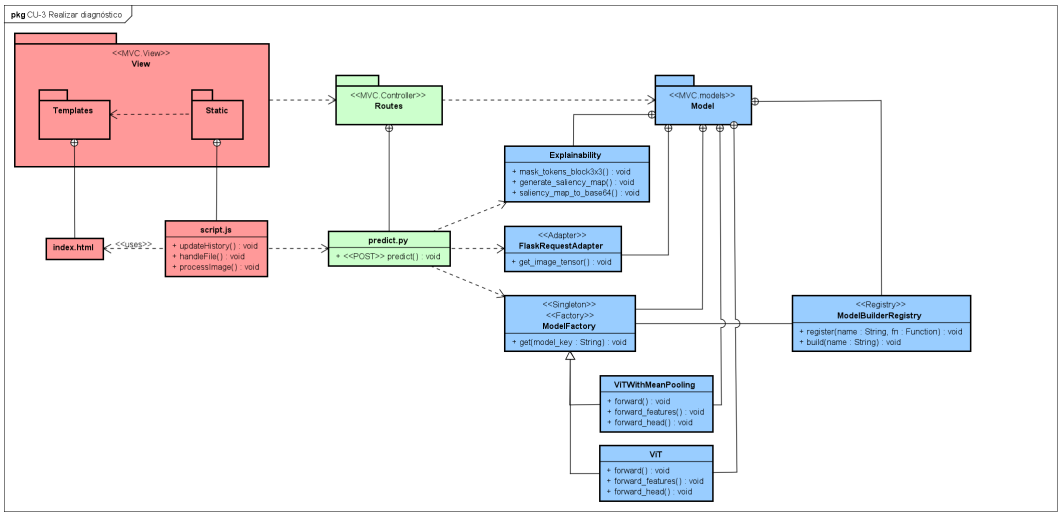


Figura 8.6: Diagrama Uses Style del CU-3 Realizar diagnóstico.

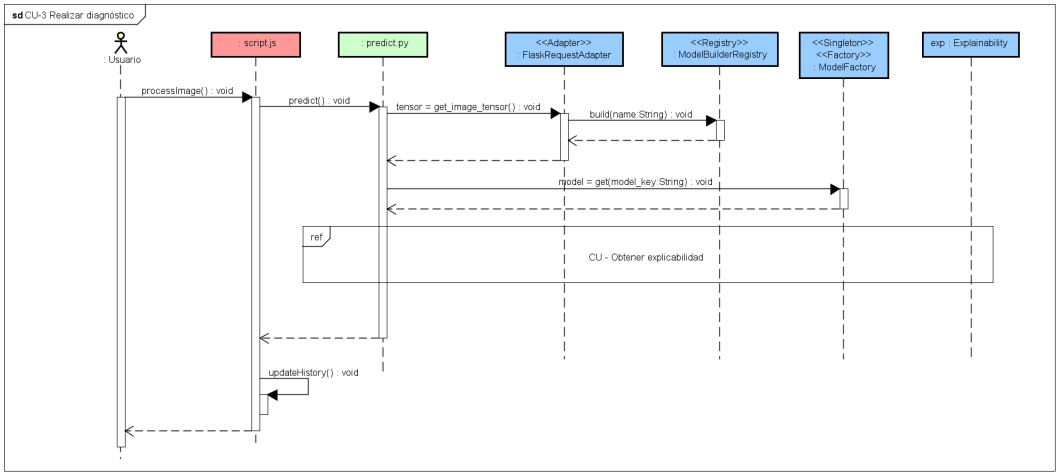


Figura 8.7: Diagrama de secuencia del CU-3 Realizar diagnóstico.

Otros casos de uso

A continuación, se presentan los diagramas de secuencia de otros casos de uso relevantes para la funcionalidad general de la aplicación.

Se han incluido únicamente los diagramas correspondientes a los casos de uso CU-1 y CU-5, ya que los casos CU-2 y CU-4 no tienen la complejidad suficiente como para elaborar un diagrama específico.

El CU-2 consiste en una operación simple de almacenamiento de una clave a partir de un elemento HTML, mientras que el CU-4 se reduce a una única llamada a una función JavaScript, la cual ya se encuentra representada en el paso final del diagrama de la Figura 8.8.

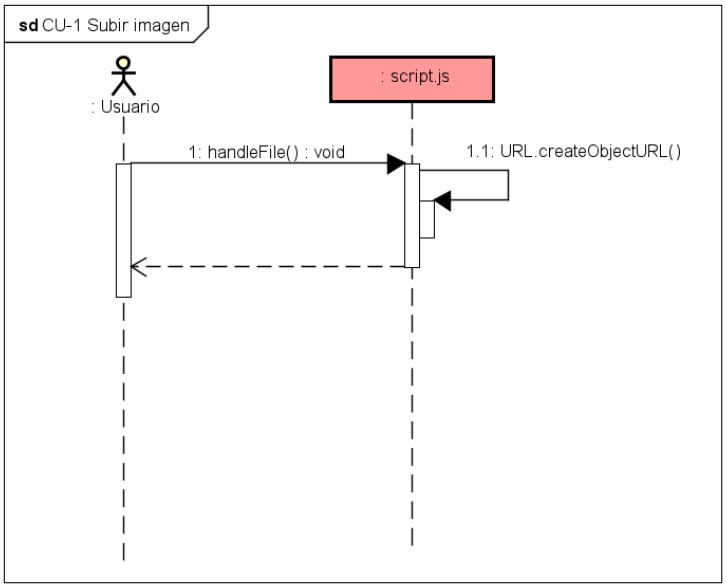


Figura 8.8: Diagrama de secuencia del CU-1 Subir imagen.

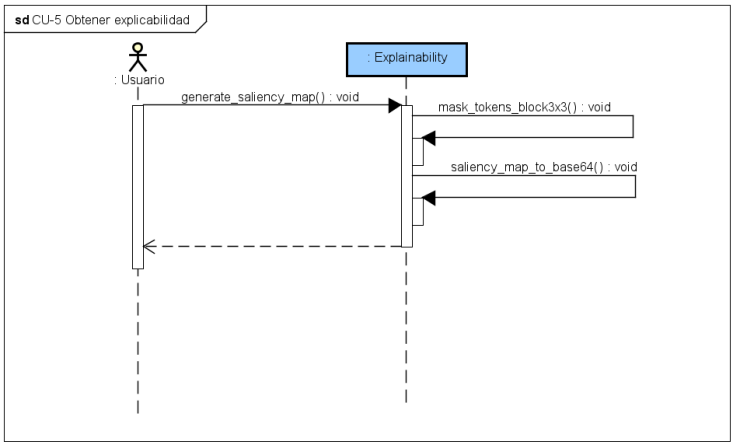


Figura 8.9: Diagrama de secuencia del CU-5 Obtener explicabilidad.

Capítulo 9

Conclusiones

A lo largo de este Trabajo de Fin de Grado se han aprendido y aplicado diversas herramientas, técnicas y metodologías de la Ingeniería Informática. Desde la planificación y diseño de un proyecto desde cero, hasta el desarrollo del sistema, el uso de librerías especializadas para Aprendizaje Automático y el uso de entornos de experimentación.

En particular, se han utilizado aspectos avanzados del campo del Aprendizaje Profundo, centrados en los modelos basados en *Transformers*. Esto ha permitido ahondar en mecanismos como la atención múltiple, la codificación posicional y las técnicas de agregación de información, adaptando dichas ideas al dominio de la Visión por Computador mediante el uso de la arquitectura Vision Transformer.

Este proyecto ha permitido explorar las posibilidades de este tipo de arquitecturas en un entorno médico real, en comparación con enfoques tradicionales basados en Redes Neuronales Convolucionales, ampliamente utilizadas en este tipo de tareas.

Los resultados obtenidos han sido heterogéneos. En el conjunto de datos de radiografías cerebrales, se ha conseguido mejorar de forma notable la precisión sobre el conjunto de test, aumentando en casi un 30 % respecto a los modelos CNN empleados en el trabajo original. Este resultado pone de manifiesto el potencial de los ViT en entornos, donde las relaciones espaciales globales sean especialmente relevantes o donde las CNN puedan tener dificultades para generalizar.

Sin embargo, en los conjuntos de ojos y tórax, los modelos ViT no han logrado superar los resultados obtenidos previamente con arquitecturas convolucionales. Aunque en ambos casos se alcanzaron valores de precisión considerablemente altos, los modelos CNN se mantuvieron por encima. Esto podría explicarse por varios factores, como la dificultad en el ajuste de hiperparámetros específicos para ViT en situaciones de baja cantidad de datos, o el hecho de que en ciertos contextos locales (donde patrones espaciales específicos son determinantes) las CNN siguen siendo más eficientes.

Esta comparación de resultados es de gran importancia, ya que complementa una de las

principales conclusiones del trabajo: los ViT no deben considerarse un reemplazo directo de las CNN, sino una alternativa con fortalezas y debilidades propias, que deben valorarse en función del contexto. En conjuntos donde la cantidad de datos es limitada o donde el detalle local predomina sobre la estructura global, los ViT pueden no resultar tan efectivos como se esperaba. Por tanto, una evaluación caso a caso sigue siendo necesaria.

Se ha implementado y analizado una técnica complementaria basada en ViT-ReciproCAM, con el fin de poder dar explicaciones visuales sobre el funcionamiento interno de los modelos. Esta herramienta, adaptada específicamente para modelos basados en atención, ha permitido generar mapas de saliencia interpretables, que ayudan a identificar las regiones de la imagen más relevantes para la decisión del modelo. Si bien su aplicación ha sido exploratoria, ha servido para reforzar la comprensión del comportamiento del sistema.

Se han abordado diversas dificultades relacionadas con el ajuste de los modelos, destacando especialmente los fenómenos de *overfitting* e *underfitting*. Para hacerles frente, se han implementado múltiples estrategias de regularización y ajuste estructural, incluyendo variaciones en la profundidad de la arquitectura, modificaciones en las tasas de dropout o ajustes en la función de pérdida. Estos cambios han requerido un proceso iterativo de experimentación de larga duración.

A pesar de estas limitaciones, el trabajo ha permitido explorar en profundidad el funcionamiento de los Vision Transformers de manera específica, abordando aspectos como la división en patches, el uso de posiciones embebidas y su estructura basada en el Transformer original. Igualmente, se ha llevado a cabo un análisis detallado sobre configuraciones estructurales, como el uso de bloques residuales o la selección del vector de salida (`[CLS]` vs. *mean pooling*), de gran utilidad a la hora de refinar cada modelo.

En términos generales, este proyecto ha contribuido a consolidar competencias adquiridas a lo largo del grado, familiarizarse con arquitecturas ciertamente modernas en el campo del *Deep Learning*.

9.1. Líneas de trabajo futuras

Existen diversos caminos para ampliar el alcance de este trabajo:

- **Optimización de hiperparámetros**: aunque se han obtenido resultados satisfactorios mediante ajustes manuales, la aplicación de técnicas sistemáticas podría permitir identificar configuraciones más eficientes, sobre todo en los casos donde el rendimiento ha sido más limitado o inestable.
- **Extensión del conjunto de datos**: dado que los modelos ViT tienden a beneficiarse de grandes cantidades de datos, la incorporación de nuevas muestras mediante la recolección directa, o el uso de conjuntos públicos adicionales, podría mejorar la capacidad de generalización de los modelos.
- **Análisis en profundidad de un conjunto específico**: una de las rarezas del proyecto ha sido trabajar simultáneamente con tres conjuntos de datos distintos. Un enfoque

alternativo sería centrarse en uno sólo de ellos y realizar un análisis más profundo y específico, explorando configuraciones más avanzadas y ajustadas de manera particular.

- **Mejora de la aplicación**: actualmente la aplicación cumple su función, pero se podrían añadir aspectos para perfeccionar la interfaz de usuario, seguridad y en la simultaneidad. Esto se haría con un servidor, creando una buena configuración a través de la red y adaptando el sistema para permitir un uso más eficiente y escalable.
- **Escalabilidad mediante hardware avanzado**: muchas de las limitaciones experimentadas durante el desarrollo han estado condicionadas por la capacidad computacional disponible. Contar con un hardware más potente (como GPUs de gama alta) permitiría entrenar modelos más profundos y complejos, así como realizar experimentaciones en tiempos más razonables.

Bibliografía

[1] Ritesh Vedpathak, Krishna Chaitanya, Anirban Chakraborty, and Arjun Jain. Visual transformer meets cutmix for improved accuracy, communication efficiency, and data privacy in split learning. *arXiv preprint arXiv:2207.00234*, 2022.

[2] Kenneth Jensen. Crisp-dm process diagram. `https://commons.wikimedia.org/wiki/File:CRISP-DM_Process_Diagram.png`, 2013. Trabajo propio basado en documentación de IBM SPSS. Licencia CC BY-SA 3.0.

[3] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *arXiv preprint arXiv:1706.03762*, 2017.

[4] Dhanush Srinivasa. Transformer architecture: Part 1 — a deep dive. `https://pub.towardsai.net/transformer-architecture-part-1-d157b54315e6`, 2023. Accessed: 2025-04-30.

[5] Kemal Erdem. Understanding positional encoding in transformers. *https://erdem.pl*, May 2021.

[6] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale. In *International Conference on Learning Representations (ICLR)*, 2021.

[7] Aishwarya Kamath. Vision transformer explained. `https://towardsdatascience.com/vision-transformer-explained-5f6c701e8b23`, 2021.

[8] B. K. Das, G. Zhao, S. Islam, T. J. Re, D. Comaniciu, A. Maier, and E. Gibson. Coordinate-based positional embedding that captures resolution to enhance transformer's performance in medical image analysis. *Scientific Reports*, 14(1):9380, 2024.

[9] Aditya. Understanding the [cls] token in bert – a comprehensive guide. `https://aditya007.medium.com/understanding-the-cls-token-in-bert-a-comprehensive-guide-a62b3b94a941`, 2023.

[10] torch.nn.relu — pytorch 2.3 documentation. `https://docs.pytorch.org/docs/stable/generated/torch.nn.ReLU.html`. Accedido el 15 de junio de 2025.

[11] torch.nn.gelu — pytorch 2.3 documentation. `https://docs.pytorch.org/docs/stable/generated/torch.nn.GELU.html`. Accedido el 15 de junio de 2025.

[12] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. In *International Conference on Learning Representations (ICLR)*, 2019. arXiv preprint arXiv:1711.05101.

[13] Hugging Face. Transformers: Learning rate schedulers, 2023.

[14] Seok-Yong Byun and Wonju Lee. Vit-reciprocam: Gradient and attention-free visual explanations for vision transformer. *arXiv preprint arXiv:2310.02588*, 2023.

[15] Miguel Toquero Barón. Clasificación de imágenes médicas de rayos-x mediante redes neuronales convolucionales. Trabajo Fin de Grado, Universidad de Valladolid, 2021.

[16] Jorge Arranz Simón. Clasificación de tumores cerebrales a partir de imágenes médicas mediante aprendizaje profundo. Trabajo Fin de Grado, Universidad de Valladolid, 2023.

[17] Mario Izquierdo Álvarez. Redes convolucionales 2d en pytorch : clasificación de imágenes de tac de retina (oct). Trabajo Fin de Grado, Universidad de Valladolid, 2023.

[18] Fundación Bankinter. En 2025 el volumen de datos en el mundo será 175 veces más que en 2011. `https://www.fundacionbankinter.org/noticias/en-2025-el-volumen-de-datos-en-el-mundo-sera-175-veces-mas-que-en-2011/?_adin=11551547647`.

[19] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[20] Geert Litjens, Thijs Kooi, Babak Ehteshami Bejnordi, Arnaud Arindra Adiyoso Setio, Francesco Ciompi, Mohsen Ghafoorian, Jeroen A. W. M. van der Laak, Bram van Ginneken, and Clara I. Sánchez. A survey on deep learning in medical image analysis. *Medical Image Analysis*, 42:60–88, 2017.

[21] IBM Corporation. Guía del proceso crisp-dm en ibm spss modeler. `https://www.ibm.com/docs/es/SS3RA7_18.4.0/pdf/ModelerCRISPDM.pdf`, 2020.

[22] P. Chapman, J. Clinton, R. Kerber, T. Khabaza, T. Reinartz, C. Shearer, and R. Wirth. Crisp-dm 1.0: Step-by-step data mining guide. `https://cs.unibo.it/~danilo.montesi/CBD/Beatriz/10.1.1.198.5133.pdf`, 2000. Developed by the CRISP-DM consortium.

[23] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[24] Denny Britz, Anna Goldie, Minh-Thang Luong, and Quoc V. Le. Massive exploration of neural machine translation architectures. *CoRR*, abs/1703.03906, 2017.

[25] Martin Courtois, Malte Ostendorff, Leonhard Hennig, and Georg Rehm. Symmetric dot-product attention for efficient training of bert language models. *arXiv preprint arXiv:2406.06366*, 2024.

[26] Shui-Hua Wang, Yuanhao Zheng, Yu-Dong Zhang, and Yudong Zhang. Residual dense blocks and convolutional neural network for the diagnosis of covid-19 using x-ray images. *Neurocomputing*, 439:1–10, 2021.

[27] Mor Geva, Roei Schuster, Jonathan Berant, and Omer Levy. Transformer feed-forward layers are key-value memories. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 5484–5495. Association for Computational Linguistics, 2021.

[28] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016.

[29] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

[30] Yunpeng Chen, Zihang Zhang, Yuandong Tian, Bing Xiao, Lu Wang, Li Yuan, and Lei Zhang. Do we really need positional encodings in vision transformers? *arXiv preprint arXiv:2102.10882*, 2021.

[31] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[32] Anthony Fuller, Yousef Yassin, Daniel G. Kyrollos, Evan Shelhamer, and James R. Green. Simpler fast vision transformers with a jumbo cls token. *arXiv preprint arXiv:2502.15021*, 2025.

[33] Byungsoo Ko, Han-Gyu Kim, Byeongho Heo, Sangdoo Yun, Sanghyuk Chun, Geonmo Gu, and Wonjae Kim. Group generalized mean pooling for vision transformer. *arXiv preprint arXiv:2212.04114*, 2022.

[34] Zizheng Pan, Bohan Zhuang, Jing Liu, Haoyu He, and Jianfei Cai. Scalable vision transformers with hierarchical pooling. *arXiv preprint arXiv:2103.10619*, 2021.

[35] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on Machine Learning (ICML)*, pages 807–814, 2010.

[36] Dan Hendrycks and Kevin Gimpel. Gaussian error linear units (gelus). *arXiv preprint arXiv:1606.08415*, 2016.

[37] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[38] Bolei Zhou, Aditya Khosla, Agata Lapedriza, Aude Oliva, and Antonio Torralba. Learning deep features for discriminative localization. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2921–2929, 2016.

[39] Ramprasaath R. Selvaraju et al. Grad-cam: Visual explanations from deep networks via gradient-based localization. *arXiv preprint arXiv:1610.02391*, 2016.

[40] Haofan Wang et al. Score-cam: Score-weighted visual explanations for convolutional neural networks. *CVPRW*, 2020.

[41] Seok-Yong Byun and Wonju Lee. Recipro-cam: Fast gradient-free visual explanations for convolutional neural networks. *arXiv preprint arXiv:2209.14074*, 2022.

[42] Runjin Fu, Qiang Hu, Xin Dong, Yanjun Guo, Yu Gao, and Bo Li. Axiom-based grad-cam: Towards accurate visualization and explanation of cnns. *BMVC*, 2020.

[43] Samira Abnar and Willem Zuidema. Quantifying attention flow in transformers. *arXiv preprint arXiv:2005.00928*, 2020.

[44] Hila Chefer, Shir Gur, and Lior Wolf. Transformer interpretability beyond attention visualization. In *CVPR*, pages 782–791, 2021.

[45] Kaggle. Kaggle: Your machine learning and data science community. `https://www.kaggle.com/`, 2025.

[46] Muhammad E. H. Chowdhury and et al. Can ai help in screening viral and covid-19 pneumonia? *IEEE Access*, 8:132665–132676, 2020.

[47] Shadman Sakib and et al. Detection of covid-19 disease from chest x-ray images: A deep transfer learning framework. *medRxiv*, 2020. `https://doi.org/10.1101/2020.04.13.20063461`.

[48] PyTorch. torchvision.transforms.totensor — torchvision main documentation. `https://docs.pytorch.org/vision/master/generated/torchvision.transforms.ToTensor.html`, 2024.

[49] Sartaj Bhuvaji. Brain tumor classification (mri). `https://www.kaggle.com/datasets/sartajbhuvaji/brain-tumor-classification-mri`, 2020.

[50] Mayo Clinic. Brain tumor - symptoms and causes. `https://www.mayoclinic.org/diseases-conditions/brain-tumor/symptoms-causes/syc-20350084`, 2024.

[51] Paul Mooney. Kermany2018: Labeled optical coherence tomography (oct) and chest x-ray images for classification. `https://www.nei.nih.gov/learn-about-eye-health/eye-conditions-and-diseases`, 2018.

[52] National Eye Institute. Eye conditions and diseases. `https://www.nei.nih.gov/learn-about-eye-health/eye-conditions-and-diseases`, 2024.

[53] Arseny Kravchenko. Einops: Flexible and powerful tensor operations for readable and reliable code, 2020.

[54] Phil Wang. vit-pytorch: Vision transformer - pytorch. `https://github.com/lucidrains/vit-pytorch`, 2020.

[55] Bahadır Akdemir. Vision transformer (vit): How it works and how to build it in pytorch, 2023.

[56] PyTorch Team. torch.nn.multiheadattention — pytorch documentation, 2024.

[57] The HDF Group. Hdf5 file format specification, 2023.

Apéndice A

Código

```
    class PatchEmbedding(nn.Module):
    def __init__(self, img_size, patch_size, in_channels, emb_dim):
    super().__init__()

    patch_dim = patch_size * patch_size * in_channels
    self.n_patches = (img_size // patch_size) ** 2

    # First option: embedding with proyection
    self.to_patch_embedding = nn.Sequential(
    Rearrange('b c (h p1) (w p2) -> b (h w) (p1 p2 c)', p1=patch_size,
        p2=patch_size),
    nn.Linear(patch_dim, emb_dim),
    nn.LayerNorm(emb_dim)
    )

    # Second option: embedding without proyection
    self.to_patch_embedding_solo = Rearrange('b c (h p1) (w p2) -> b (
        h w) (p1 p2 c)', p1=patch_size, p2=patch_size)

    def forward(self, x):
    return self.to_patch_embedding_solo(x)
```

Listing A.1: Clase del Patch Embedding (segunda opción activada)

```
    class TransformerBlock(nn.Module):
    def __init__(self, emb_dim, num_heads, mlp_dim, dropout):
    super().__init__()

    # Normalization layers
    self.norm1 = nn.LayerNorm(emb_dim)
    self.norm2 = nn.LayerNorm(emb_dim)

```

```
        # Multihead attention
        self.attn = nn.MultiheadAttention(embed_dim=emb_dim, num_heads=
            num_heads, dropout=dropout, batch_first=True)

        # Feed-forward
        self.mlp = nn.Sequential(
        nn.Linear(emb_dim, mlp_dim),
        nn.ReLU(),
        nn.Linear(mlp_dim, emb_dim),
        nn.Dropout(dropout)
        )

        def forward(self, x):
        # First block: multi-head self-attention with residual connection
            and layer normalization
        attn_output, _ = self.attn(x, x, x, need_weights=False)
        x = self.norm1(x + attn_output)

        # Second block: feed-forward network with residual connection and
            layer normalization
        ff_output = self.mlp(x)
        x = self.norm2(x + ff_output)

        return x
```

Listing A.2: Clase del bloque del codificador Transformer

```
        def forward(self, x):
        # First block: multi-head self-attention with residual connection
            and layer normalization
        attn_output, _ = self.attn(x, x, x, need_weights=False)
        x = self.norm1(x + attn_output)

        # Second block: feed-forward network followed by layer
            normalization (no residual connection here)
        ff_output = self.mlp(x)
        x = self.norm2(ff_output)

        return x
```

Listing A.3: Segunda versión parte feed-forward

```
    class ViT(nn.Module):
    def __init__(self, img_size, patch_size, in_channels, emb_dim,
        num_heads, mlp_dim, num_layers, num_classes, dropout):
    super().__init__()

    # Converts the input image into a sequence of flattened patches (
        tokens)
```

```
        self.patch_embed = PatchEmbedding(img_size, patch_size,
            in_channels, emb_dim)

        # Learnable [CLS] token used as a global representation
        self.cls_token = nn.Parameter(torch.randn(1, 1, emb_dim, device=
            device))

        # Learnable positional encoding added to patch embeddings and the
            [CLS] token
        self.pos_embedding = nn.Parameter(torch.randn(1, 1 + self.
            patch_embed.n_patches, emb_dim, device=device))

        # Stack of Transformer encoder blocks
        self.transformer_blocks = nn.Sequential(
        *[TransformerBlock(emb_dim, num_heads, mlp_dim, dropout) for _ in
            range(num_layers)]
        )

        # Layer normalization before classification
        self.norm = nn.LayerNorm(emb_dim)

        # Final linear classification head
        self.mlp_head = nn.Linear(emb_dim, num_classes)

        def forward(self, x):
        # Batch size
        B = x.shape[0]

        # Patch embedding of the input image
        x = self.patch_embed(x)

        # Expand and concatenate [CLS] token with patch tokens
        cls_tokens = self.cls_token.expand(B, -1, -1)
        x = torch.cat([cls_tokens, x], dim=1)

        # Add positional encoding
        x = x + self.pos_embedding[:, :x.size(1), :]

        # Pass through Transformer encoder blocks
        x = self.transformer_blocks(x)

        # Extract and normalize the [CLS] token and classification
        x = self.norm(x[:, 0])
        return self.mlp_head(x)
```

Listing A.4: Clase principial del ViT con CLS

```
    class ViTWithMeanPooling(nn.Module):
    def __init__(self, img_size, patch_size, in_channels, emb_dim,
        num_heads, mlp_dim, num_layers, num_classes, dropout):
    super().__init__()
```

```
4
5       # Converts the input image into a sequence of flattened patches (
            tokens)
6       self.patch_embed = PatchEmbedding(img_size, patch_size,
            in_channels, emb_dim)
7
8       # Learnable positional encoding added to the patch embeddings
9       self.pos_embedding = nn.Parameter(torch.randn(1, self.patch_embed.
            n_patches, emb_dim, device=device))
10
11      # Stack of Transformer encoder blocks
12      self.transformer_blocks = nn.Sequential(
13      *[TransformerBlock(emb_dim, num_heads, mlp_dim, dropout) for _ in
            range(num_layers)]
14      )
15
16      # Layer normalization before classification
17      self.norm = nn.LayerNorm(emb_dim)
18
19      # Final linear classification head
20      self.mlp_head = nn.Linear(emb_dim, num_classes)
21
22      def forward(self, x):
23      # Patch embedding with positional encoding
24      x = self.patch_embed(x)
25      x = x + self.pos_embedding[:, :x.size(1), :]
26
27      # Pass through Transformer encoder blocks
28      x = self.transformer_blocks(x)
29
30      # Global mean pooling over all token embeddings (instead of using
            a [CLS] token)
31      x = x.mean(dim=1)
32
33      # Final normalization and classification
34      x = self.norm(x)
35      return self.mlp_head(x)
```

Listing A.5: Clase principial del ViT con Mean Pooling

```
1       class HDF5Datasetv2(Dataset):
2           def __init__(self, h5_file_path, train=True, validate=False):
3               self.h5_file = h5py.File(h5_file_path, "r")
4               if validate:
5                   self.images = self.h5_file['val_images']
6                   self.labels = self.h5_file['val_labels']
7                   self.split = "val"
8               else:
9                   if train:
10                      self.images = self.h5_file['train_images']
11                      self.labels = self.h5_file['train_labels']
```

```
                    self.split = "train"
                else:
                    self.images = self.h5_file['test_images']
                    self.labels = self.h5_file['test_labels']
                    self.split = "test"


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

        def __getitem__(self, idx):
            image = torch.tensor(self.images[idx], dtype=torch.float32
                )  # (1, 256, 256)
            label = torch.tensor(self.labels[idx], dtype=torch.long)
            return image, label

        def close(self):
            self.h5_file.close()
```

Listing A.6: Clase del dataset para archivos HDF5.

```
    class HDF5Datasetv2(Dataset):
        def __init__(self, h5_file_path, train=True, validate=False):
            self.h5_file = h5py.File(h5_file_path, "r")
            if validate:
                self.images = self.h5_file['val_images']
                self.labels = self.h5_file['val_labels']
                self.split = "val"
            else:
                if train:
                    self.images = self.h5_file['train_images']
                    self.labels = self.h5_file['train_labels']
                    self.split = "train"
                else:
                    self.images = self.h5_file['test_images']
                    self.labels = self.h5_file['test_labels']
                    self.split = "test"

            # Transforms
            if self.split == "train":
                self.transform = v2.Compose([
                    v2.RandomHorizontalFlip(),
                    v2.RandomAdjustSharpness(sharpness_factor=1.5, p
                        =0.3),
                    v2.RandomAutocontrast(p=0.2),
                    v2.RandomRotation(degrees=5),
                    v2.RandomPerspective(distortion_scale=0.1, p=0.3)
                ])

        def __len__(self):
            return len(self.labels)
```

```
30
31          def __getitem__(self, idx):
32              image = torch.tensor(self.images[idx], dtype=torch.float32
                    )  # (1, 256, 256)
33              if self.split == "train":
34                  image = self.transform(image)
35              label = torch.tensor(self.labels[idx], dtype=torch.long)
36              return image, label
37
38          def close(self):
39              self.h5_file.close()
```

Listing A.7: Clase del dataset para archivos HDF5 version 2.

```
1       import h5py
2       import numpy as np
3       from torchvision import datasets, transforms
4       from tqdm import tqdm
5
6       train_dataset_path = "../data/torax/train"
7       test_dataset_path = "../data/torax/test"
8       output_file = "../data/torax_dataset_modif_rot_5.h5"
9
10      tam = 256
11      train_transform = transforms.Compose([
12      transforms.Resize((tam,tam)),
13      #transforms.RandomHorizontalFlip(),
14      transforms.RandomRotation(5),
15      #transforms.ColorJitter(0.4, 0.4, 0.4, 0.1),
16      transforms.ToTensor(),
17      #transforms.Normalize(mean=[0.5]*3, std=[0.5]*3)
18      ])
19
20      test_transform = transforms.Compose([
21      transforms.Resize((tam,tam)),
22      transforms.ToTensor(),
23      #transforms.Normalize(mean=[0.5]*3, std=[0.5]*3)
24      ])
25
26
27      train_dataset = datasets.ImageFolder(root=train_dataset_path,
            transform=train_transform)
28      test_dataset = datasets.ImageFolder(root=test_dataset_path,
            transform=test_transform)
29
30      with h5py.File(output_file, "w") as hdf:
31      train_images = hdf.create_dataset("train_images", shape=(len(
            train_dataset), 3, tam, tam), dtype=np.float32)
32      train_labels = hdf.create_dataset("train_labels", shape=(len(
            train_dataset),), dtype=np.int64)
```

```
    test_images = hdf.create_dataset("test_images", shape=(len(
        test_dataset), 3, tam, tam), dtype=np.float32)
    test_labels = hdf.create_dataset("test_labels", shape=(len(
        test_dataset),), dtype=np.int64)

    class_names = train_dataset.classes  # Asumimos que las clases son
         las mismas en ambos
    hdf.attrs["class_names"] = [name.encode("utf-8") for name in
        class_names]

    # Entrenamiento
    for i, (img, label) in tqdm(enumerate(train_dataset), total=len(
        train_dataset), desc="Procesando entrenamiento"):
    train_images[i] = img.numpy().astype(np.float32)
    train_labels[i] = label

    # Test
    for i, (img, label) in tqdm(enumerate(test_dataset), total=len(
        test_dataset), desc="Procesando prueba"):
    test_images[i] = img.numpy().astype(np.float32)
    test_labels[i] = label

    print(f"Archivo HDF5 creado!")
    print(f"Imagenes train: {len(train_dataset)}")
    print(f"Imagenes test: {len(test_dataset)}")
    print(f"Clases: {class_names}")
```

Listing A.8: Creación de archivos HDF5, caso tórax.

```
    for epoch in range(num_epochs):
        print(f"Epoch: {epoch + 1}/{num_epochs}")

        # Training
        model.train()
        running_loss, correct, total = 0.0, 0, 0

        for images, targets in tqdm(train_loader):
            images, targets = images.to(device, non_blocking=True),
                targets.to(device, non_blocking=True)
            optimizer.zero_grad()
            y_pred = model(images)
            loss = criterion(y_pred, targets.long())
            loss.backward()
            optimizer.step()

            scheduler.step()

            running_loss += loss.item()
            correct += (y_pred.argmax(dim=1) == targets).sum().item()
            total += targets.size(0)

```

```
        # scheduler.step()

        train_loss.append(running_loss / len(train_loader))
        train_accuracy.append(correct / total)
        print(f'Train Loss: {train_loss[-1]:.5f}\tTrain Accuracy: {
            train_accuracy[-1]:.5f}')

        # Testing
        model.eval()
        correct, total, running_loss = 0, 0, 0.0

        with torch.no_grad():
            for images, targets in test_loader:
                images, targets = images.to(device, non_blocking=True)
                    , targets.to(device, non_blocking=True)
                y_test_pred = model(images)
                loss = criterion(y_test_pred, targets.long())

                running_loss += loss.item()
                correct += (y_test_pred.argmax(dim=1) == targets).sum
                    ().item()
                total += targets.size(0)

        test_loss.append(running_loss / len(test_loader))
        test_accuracy.append(correct / total)
        learning_rate.append(optimizer.param_groups[0]['lr'])
        print(f'Test Loss: {test_loss[-1]:.5f}\tTest Accuracy: {
            test_accuracy[-1]:.5f}\tLearning Rate: {learning_rate[-1]}'
            )

        # scheduler.step()
```

Listing A.9: Bucle de entrenamiento.

```
    # Best model saving
    if epoch > 70:
    if test_accuracy[-1] < best_test_acc:
    best_test_acc = test_accuracy[-1]
    best_model_state = model.state_dict()
    torch.save(best_model_state, "modelo_brain.pth")
    print(f"Model saved (epoch {epoch+1}) with Test Loss: {test_loss
        [-1]:.5f} and Accuracy: {test_accuracy[-1]:.5f}")
    epochs_without_improvement = 0
    else:
    epochs_without_improvement += 1

    # Early stopping condition
    if epochs_without_improvement >= early_stop_patience:
    print("Early stopping triggered. No improvement in the last 10
        epochs.")
    break
```

Listing A.10: Selección de mejor modelo.

```
    def forward_features(self, x):
    x = self.patch_embed(x)
    x = x + self.pos_embedding[:, :x.size(1), :]
    x = self.transformer_blocks(x)
    return x # [B, T, D]

    def forward_head(self, tokens):
    x = tokens.mean(dim=1)
    x = self.norm(x)
    return self.mlp_head(x)
```

Listing A.11: Métodos adicionales para ViT con Mean Pooling.

```
    def forward_features(self, x):
    B = x.shape[0]
    x = self.patch_embed(x)
    cls_tokens = self.cls_token.expand(B, -1, -1)
    x = torch.cat([cls_tokens, x], dim=1)
    x = x + self.pos_embedding[:, :x.size(1), :]
    x = self.transformer_blocks(x)
    return x  # [B, T+1, D]

    def forward_head(self, tokens):
    cls_token = self.norm(tokens[:, 0])
    return self.mlp_head(cls_token)
```

Listing A.12: Métodos adicionales para ViT original (CLS).

```
    def mask_tokens_block_3x3(tokens, patch_size=8, center_row=0,
        center_col=0, fill_value=0.0):

    tokens = tokens.clone()
    num_patches = int((tokens.shape[1]) ** 0.5)  # se asume cuadrIcula
         P x P
    D = tokens.shape[2]

    for dr in [-1, 0, 1]:
    for dc in [-1, 0, 1]:
    r = center_row + dr
    c = center_col + dc
    if 0 <= r < num_patches and 0 <= c < num_patches:
    idx = r * num_patches + c
    tokens[0, idx, :] = fill_value
    return tokens
```

Listing A.13: Función de enmascaramiento 3x3.

```
    model.eval()
    image = image.to(device).unsqueeze(0)  # [1, 3, 128, 128]

    with torch.no_grad():
    tokens = model.forward_features(image)
    logits = model.forward_head(tokens)
    pred_class = logits.argmax(dim=1).item()
    original_score = logits[0, pred_class].item()

    import matplotlib.pyplot as plt
    import torchvision.transforms as T
    import torch.nn.functional as F

    print(f"Clase verdadera: {label}")
    print(f"Clase predicha: {pred_class}")

    # Upsample the saliency map to match image size
    upsampled_map = F.interpolate(
    saliency_map.unsqueeze(0).unsqueeze(0),
    size=(128, 128),
    mode='bilinear',
    align_corners=False
    ).squeeze().cpu().numpy()

    import numpy as np

    low, high = np.percentile(upsampled_map, [5, 95])  # recorta los
        extremos
    norm_map = (upsampled_map - low) / (high - low + 1e-8)
    norm_map = norm_map.clip(0, 1)


    # Visualizacion
    plt.imshow(T.ToPILImage()(image.squeeze().cpu()))
    plt.imshow(norm_map, cmap='turbo', alpha=0.4) # turbo, jet, plasma
        , viridis
    plt.colorbar(label="Importancia relativa")
    plt.title("ReciproCAM-normalized saliency map")
    plt.axis('off')
    plt.show()
```

Listing A.14: Ejemplo de creación de mapa de saliencia.

Apéndice B

Manual de instalación

Este apéndice detalla el procedimiento necesario para clonar, construir y ejecutar la aplicación desarrollada en este Trabajo de Fin de Grado. Se proporciona una guía paso a paso para asegurar su correcta instalación mediante Docker.

1. Clonación del repositorio

Para obtener una copia local del proyecto, es necesario clonar el repositorio desde GitLab (o GitHub) mediante el siguiente comando en la terminal:

```
git clone https://gitlab.inf.uva.es/usuario/repositorio.git
```

Reemplace la URL por la correspondiente al repositorio real.

2. Requisitos previos

Para ejecutar el proyecto mediante contenedores, es necesario tener instalado:

- **Docker Desktop**, disponible en: `https://www.docker.com/products/docker-desktop`
- Acceso a una terminal de comandos (CMD, PowerShell o WSL en Windows)

3. Construcción de la imagen Docker

Desde la raíz del repositorio, acceda a la carpeta principal del proyecto (por ejemplo, `projecto-app`):

```
cd repositorio/projecto-app
```

Ejecute el siguiente comando para construir la imagen Docker:

```
docker build -t mi-app-tfg .
```

Este proceso descargará la imagen base de Python, copiará el contenido del proyecto y ejecutará la instalación de las dependencias indicadas en el archivo `requirements.txt`.

4. Ejecución de la aplicación

Una vez construida la imagen, puede iniciarse el contenedor mediante:

```
docker run -p 5000:5000 mi-app-tfg
```

Este comando expone el puerto 5000 del contenedor al mismo puerto en el host, permitiendo acceder a la aplicación desde un navegador.

5. Acceso a la aplicación

Con el contenedor en ejecución, la aplicación se encuentra disponible en el navegador en la siguiente dirección:

`http://localhost:5000`

6. Notas adicionales

- La aplicación está desarrollada con Python 3.13 y el microframework Flask.
- El contenedor utiliza como base una imagen oficial de Python optimizada (slim).
- Este despliegue corresponde al entorno de desarrollo. Para producción se recomienda utilizar un servidor WSGI.

Apéndice C

Manual de usuario

Este apéndice describe el funcionamiento de la aplicación web desarrollada, desde la carga de una imagen hasta la obtención de resultados de predicción utilizando distintos modelos de clasificación. Se incluye una guía visual que ilustra cada paso del proceso.

1. Página de inicio

Al acceder a la dirección `http://localhost:5000`, se carga la página principal de la aplicación, que permite al usuario subir la imagen que desee.

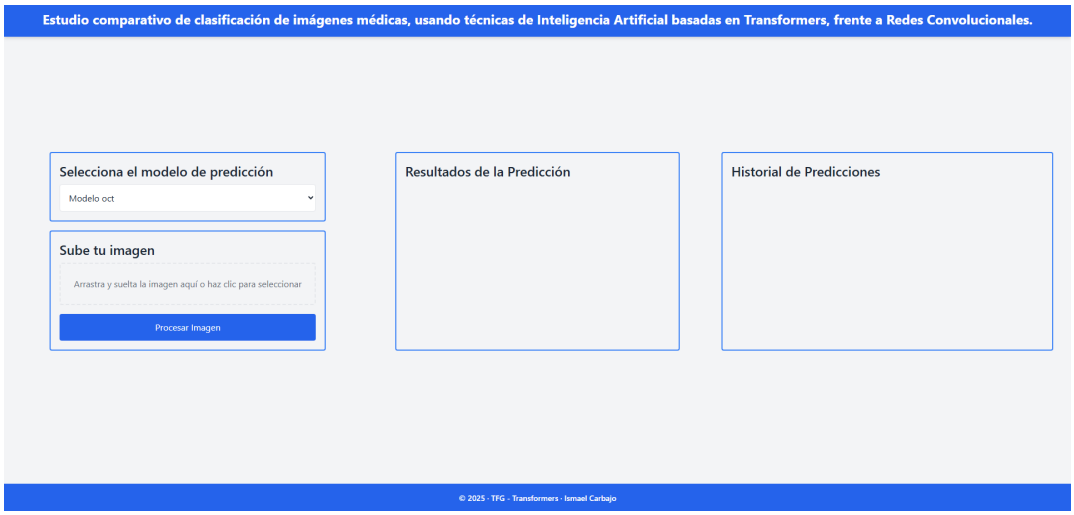


Figura C.1: Pagina inicial.

2. Carga de imagen

El usuario debe seleccionar desde su sistema local una imagen compatible para el análisis (formato `.jpg`, `.png`, etc.). Una vez cargada, la imagen se previsualiza en la misma página.

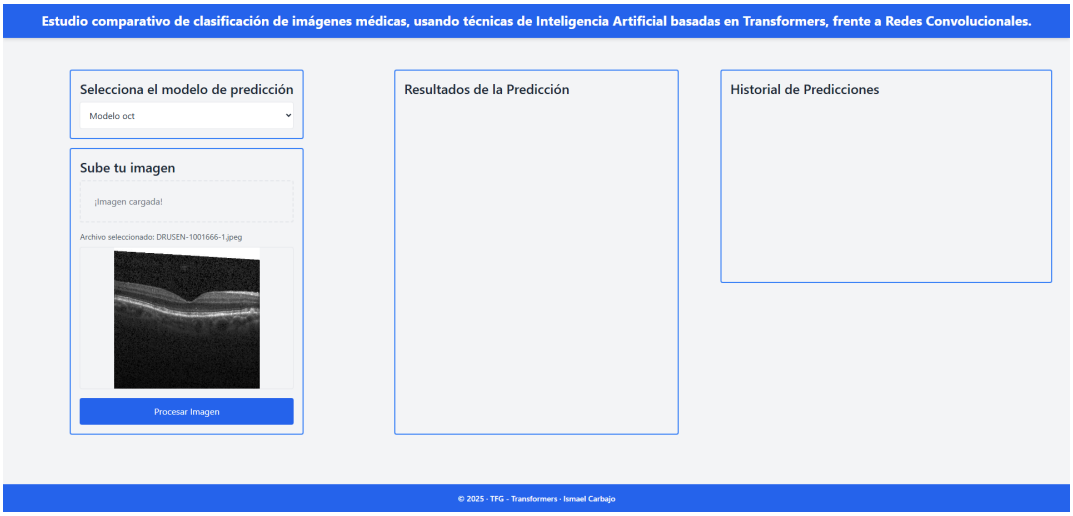


Figura C.2: Pagina con imagen cargada.

3. Selección del modelo

La interfaz permite seleccionar uno de los modelos disponibles para realizar la predicción. Esta selección se hace a través de un menú desplegable.

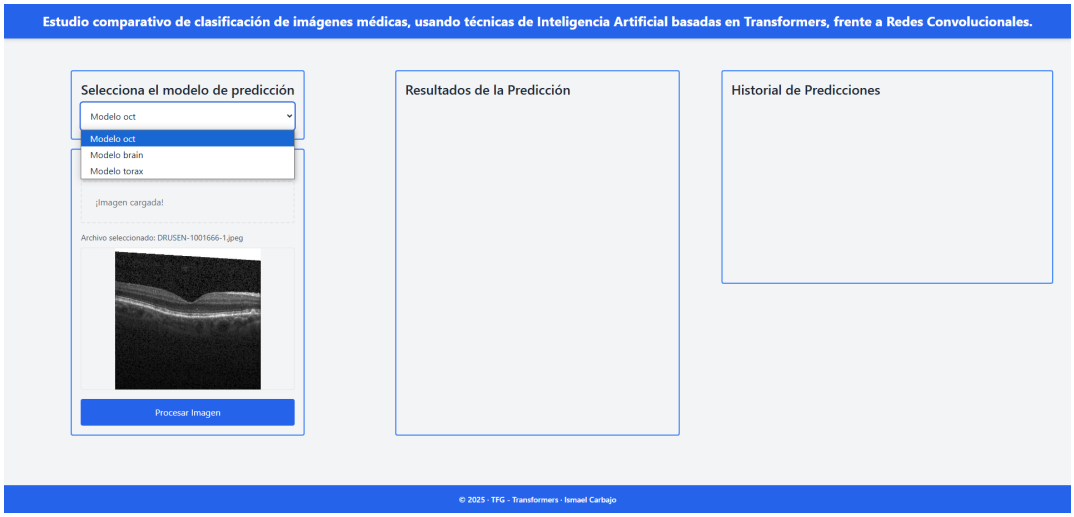


Figura C.3: Pagina seleccionando modelo.

4. Ejecución de la predicción

Al pulsar el botón correspondiente, la aplicación ejecuta el modelo seleccionado sobre la imagen cargada, procesándola internamente. Durante este proceso, puede mostrarse un indicador de carga.

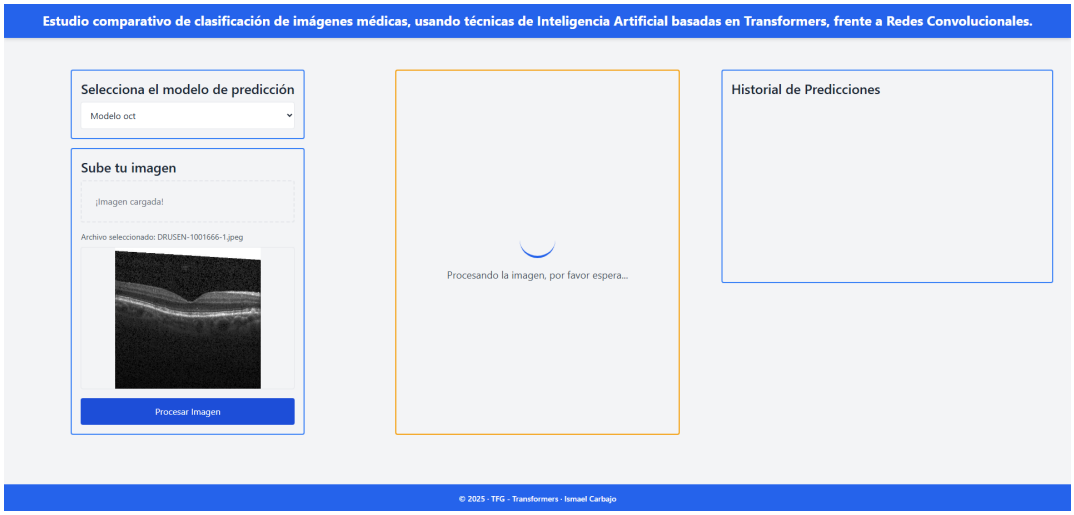


Figura C.4: Pagina realizando la predicción.

5. Visualización de resultados

Finalizada la predicción, se muestran en pantalla los resultados obtenidos: la probabilidad de cada clase, la clase predicha, el mapa de saliencia explicativo y la nueva entrada en el historial de predicciones.

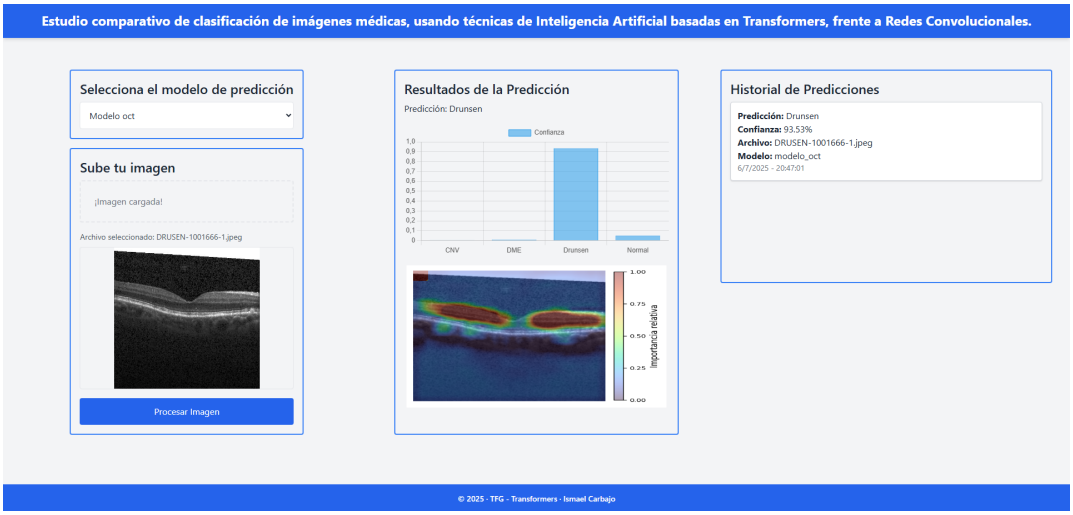


Figura C.5: Pagina con todos los resultados.