UNIVERSIDAD DE MÁLAGA

E.T.S. INGENIERÍA INFORMÁTICA
UNIVERSIDAD DE MÁLAGA

Graduado en Ingeniería Informática

# Aplicación de Modelos de Vision Transformer para la Segmentación de Imágenes en Esclerosis Múltiple

## Application of Vision Transformer Models for Image Segmentation in Multiple Sclerosis


Realizado por
Cynthia Guzmán Boceta

Tutorizado por
Rafael Marcos Luque Baena
Ezequiel López Rubio

Departamento
Lenguajes y Ciencias de la Computación


MÁLAGA, junio de 2025

UNIVERSIDAD DE MÁLAGA

E.T.S. INGENIERÍA INFORMÁTICA
UNIVERSIDAD DE MÁLAGA

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

GRADUADO EN INGENIERÍA INFORMÁTICA

# Aplicación de Modelos de Vision Transformer para la Segmentación de Imágenes en Esclerosis Múltiple

# Application of Vision Transformer Models for Image Segmentation in Multiple Sclerosis


Realizado por
**Cynthia Guzmán Boceta**

Tutorizado por
**Rafael Marcos Luque Baena**
**Ezequiel López Rubio**

Departamento
**Lenguajes y Ciencias de la Computación**

UNIVERSIDAD DE MÁLAGA
MÁLAGA, JUNIO DE 2025



Fecha defensa: septiembre de 2025

# Abstract

Multiple sclerosis is a chronic, neurodegenerative autoimmune disease that affects the central nervous system and can lead to a progressive loss of motor, sensory, and cognitive functions. Early detection and continuous monitoring of the disease are essential to improve patients' quality of life. Magnetic resonance imaging (MRI) is the primary tool for identifying characteristic lesions of this disease, although manual interpretation is complex, subjective, and prone to errors.

This Final Year Dissertation proposes an artificial intelligence–based solution, specifically deep learning, to automate the segmentation of MRI images of patients with this disease, enabling visualization of brain lesions. For this purpose, Vision Transformer (ViT) models have been used, a recent architecture that has demonstrated the ability to overcome some limitations of traditionally used convolutional neural networks. First, a multimodal MRI image dataset (T1, T2, FLAIR) was obtained and preprocessed to ensure homogeneity and suitability for training. Several ViT models were then trained and evaluated, selecting the one with the best performance. Additionally, an easy-to-use interactive web application was developed, which employs the best trained model to generate segmentations, allowing healthcare professionals to upload corresponding images and view the results.

Finally, a series of proposals for future improvements are included, along with an installation guide to deploy and use the application on any device.

**Keywords: segmentation, multiple sclerosis, deep learning, vision transformers.**

# Resumen

La esclerosis múltiple es una enfermedad autoinmune, neurodegenerativa y crónica que afecta al sistema nervioso central, y que puede provocar una pérdida progresiva de funciones motoras, sensoriales y cognitivas. Es esencial detectarla a tiempo y realizar un seguimiento de la enfermedad para mejorar la calidad de vida de los pacientes. La resonancia magnética (RM) es la herramienta principal para identificar lesiones características de esta enfermedad, aunque su interpretación manual es compleja, subjetiva y propensa a errores.

En este Trabajo de Fin de Grado se propone una solución basada en inteligencia artificial, específicamente en aprendizaje profundo (*deep learning*), para automatizar la segmentación de imágenes de resonancia magnética de pacientes con esta enfermedad, permitiendo visualizar las lesiones cerebrales. Para ello, se han utilizado modelos de tipo *Vision Transformer* (ViT), que es una arquitectura reciente que ha demostrado superar algunas limitaciones de las redes convolucionales que se habían usado tradicionalmente hasta ahora. Primero, se ha obtenido y preprocesado un conjunto de imágenes de RM multimodales (T1, T2, FLAIR) para asegurar que sean homogéneas y poder usarlas en el entrenamiento. Después se han entrenado y evaluado varios modelos ViT, escogiendo finalmente el de mejor rendimiento. Por otra parte, se ha construido una aplicación web interactiva y sencilla que utiliza el mejor modelo entrenado previamente para generar la segmentación, en la que el personal sanitario podrá subir las imágenes correspondientes y visualizar la segmentación.

Por último, se incluyen una serie de propuestas para mejoras en el futuro, así como una guía de instalación para desplegar y utilizar la aplicación en cualquier dispositivo.

**Palabras clave: segmentación, esclerosis múltiple, aprendizaje profundo, vision transformers.**

# Índice

# 1 Introducción

## 1.1. Motivación

La esclerosis múltiple (EM) es una enfermedad crónica y autoinmune que afecta al sistema nervioso central, compuesto por el cerebro, la médula espinal y los nervios ópticos. En esta condición, el sistema inmunitario ataca por error la mielina, la sustancia que recubre y protege las fibras nerviosas, lo que interfiere en la transmisión de los impulsos nerviosos. Esto puede provocar una pérdida progresiva de funciones motoras, sensoriales y cognitivas (Mayo Clinic, 2025). Es de vital importancia diagnosticarla a tiempo y realizar un seguimiento para mejorar la calidad de vida de los pacientes.

En este sentido, la resonancia magnética (RM) ha demostrado ser una herramienta esencial en la identificación de lesiones y la evaluación del daño en el sistema nervioso central de los pacientes con EM. La RM proporciona imágenes detalladas del cerebro, permitiendo a los médicos observar las áreas afectadas por la inflamación o la desmielinización (Roche Pacientes, 2024). Sin embargo, la interpretación de estas imágenes es un proceso complejo que requiere de experiencia clínica. Tradicionalmente, la segmentación de imágenes médicas, que consiste en identificar y clasificar las diferentes zonas relevantes, como las lesiones de esclerosis múltiple, ha sido realizada de manera manual por radiólogos y neurólogos. Esto es propenso a errores debido a la variabilidad en la interpretación entre distintos observadores. Además, la creciente cantidad de imágenes médicas que los profesionales de la salud deben procesar ha aumentado la necesidad de métodos automáticos para su segmentación.

Las técnicas de inteligencia artificial, especialmente el aprendizaje profundo (*deep learning*), han mostrado un gran potencial para automatizar este proceso. Aunque las redes neuronales convolucionales (CNN) han sido ampliamente utilizadas, los *Vision Transformers* (ViT) han mostrado un gran potencial para superar sus limitaciones, ya que pueden capturar relaciones a largo plazo entre los píxeles de la imagen, manejando de manera más eficiente las complejidades de las imágenes (Aburass et al., 2025; Raj, 2023). El uso de estos modelos puede mejorar significativamente la precisión y velocidad del diagnóstico, ayudando así a los profesionales de la salud en el análisis y seguimiento de esta enfermedad.

Este contexto resalta la necesidad de soluciones innovadoras que utilicen las últimas tecnologías en inteligencia artificial para apoyar el diagnóstico y el seguimiento de la esclerosis

múltiple.

## 1.2. Objetivos

El objetivo principal de este Trabajo de Fin de Grado es desarrollar una una aplicación web interactiva basada en inteligencia artificial que permita segmentar imágenes de resonancia magnética de pacientes con esclerosis múltiple, utilizando métodos de *deep learning* de tipo *Vision Transformer*. Debe ser fácil de utilizar para profesionales del ámbito clínico que no cuenten con experiencia técnica en *deep learning*.

Para ello, dicha aplicación se ha construido siguiendo una arquitectura cliente-servidor, donde el cliente permite al usuario cargar imágenes mediante una interfaz web intuitiva, mientras que el servidor se encarga del procesamiento y segmentación de las imágenes utilizando los modelos previamente entrenados.

A continuación, se resume a modo de esquema las funcionalidades a alto nivel:

1. **Carga de imágenes.** El usuario puede seleccionar y subir imágenes de resonancia magnética en formato .nii.gz correspondientes a las modalidades T1, T2 y FLAIR.

2. **Procesamiento e inferencia.** El servidor recibe las imágenes, realiza el preprocesamiento y ejecuta el modelo ViT para segmentar las lesiones.

3. **Visualización de la segmentación.** Se proporciona el resultado de la segmentación al usuario para su visualización.

4. **Interfaz de usuario sencilla e intuitiva.** La aplicación web tiene un diseño sencillo y funcional para facilitar su uso a profesionales de la salud sin conocimientos técnicos en programación o inteligencia artificial.

5. **Arquitectura cliente-servidor.** Separación entre la interfaz (cliente) y la lógica de procesamiento (servidor).

## 1.3. Metodología

Para llevar a cabo el desarrollo del presente Trabajo de Fin de Grado de forma ordenada, se ha seguido una metodología ágil, avanzando poco a poco (de forma iterativa) y añadiendo mejoras en cada paso (de forma incremental). Esto ha hecho posible adaptarse a los cambios, revisar lo que se iba haciendo y tomar decisiones basadas en lo que se ha ido observando en la práctica. Específicamente, el trabajo se ha dividido en las siguientes iteraciones o fases:

1. **Estudio del arte y selección de las diferentes tecnologías a utilizar.** El primer paso consistió en el estudio sobre el uso de la inteligencia artificial en la segmentación de

imágenes. Se analizaron modelos tradicionales basados en redes convolucionales (CNN), así como las nuevas arquitecturas de tipo Transformer. Además, se ha investigado qué tecnologías y herramientas pudieran ser útiles para el desarrollo del proyecto. De las escogidas, las más importantes son Python, PyTorch, MONAI y FastAPI para el lado del servidor (*backend*), y HTML, CSS y JavaScript para el lado del cliente (*frontend*).

2. **Obtención y preprocesamiento de imágenes de resonancia magnética.** Se obtuvieron imágenes de resonancia magnética de tipos T1, T2 y FLAIR de pacientes con esclerosis múltiple. Luego, se les aplicó un flujo de preprocesamiento para mejorar la homogeneidad de las imágenes.

3. **Entrenamiento del modelo.** Se entrenaron varios modelos *Vision Transformer* utilizando las imágenes preprocesadas. Para ello, se definieron los hiperparámetros de entrenamiento y se utilizaron métricas como el coeficiente de Dice para evaluar el rendimiento durante el entrenamiento. El proceso se llevó a cabo con aceleración por GPU.

4. **Evaluación.** Se evaluó el rendimiento de los modelos utilizando imágenes no vistas durante el entrenamiento. Se analizaron tanto métricas cuantitativas como resultados visuales, comparando la segmentación con las máscaras de referencia.

5. **Desarrollo de la aplicación web.** Se desarrolló una aplicación web con una arquitectura cliente-servidor. El servidor se encarga de recibir las imágenes, ejecutar el preprocesamiento y realizar la inferencia. En cambio, la interfaz web (el cliente), permite al usuario subir imágenes y descargar el resultado segmentado.

6. **Documentación.** Finalmente, se realizó la memoria en la que se explica su desarrollo y sus fases.

## 1.4. Estructura del documento

Este Trabajo de Fin de Grado se estructura en seis capítulos y un apéndice, que siguen el mismo orden en que se ha desarrollado el proyecto.

1. **Introducción**: Se introduce el problema que se quiere resolver, explicando la motivación, los objetivos y la metodología seguida, además de una breve presentación de las tecnologías empleadas.

2. **Estudio del estado del arte**: Se realiza un estudio sobre los conceptos más importantes sobre el proyecto, como la inteligencia artificial, el aprendizaje automático, el aprendizaje profundo, la visión artificial, las redes neuronales y los Vision Transformers, así como la segmentación de imágenes y las características de las resonancias magnéticas.

3. **Preprocesamiento**: Se describen en orden las transformaciones aplicadas a las imágenes (registro, remuestreo, normalización, etc.) para asegurar que sean homogéneas antes del entrenamiento.

4. **Entrenamiento**: Se explican los modelos Vision Transformer utilizados, la métrica de evaluación, la estrategia de entrenamiento y los experimentos realizados, incluyendo pruebas con distintas configuraciones y el proceso de selección del mejor modelo.

5. **Aplicación web**: Se explica el desarrollo de la aplicación web, explicando su arquitectura, el flujo de funcionamiento y cómo se integra el modelo entrenado para ofrecer segmentaciones automáticas al usuario.

6. **Conclusiones y Líneas Futuras**: Se recogen las principales conclusiones obtenidas y se proponen posibles líneas futuras de mejora o ampliación del proyecto.

Por último, en el Apéndice A, se presenta el manual de instalación, con una guía paso a paso para desplegar la aplicación en cualquier dispositivo compatible.

## 1.5. Tecnologías usadas

### 1.5.1. Python


Figura 1: Logo de Python.

Python es un lenguaje de programación de alto nivel, conocido por su sintaxis clara y legible, y es por esto que es fácil de usar y aprender. Fue creado por Guido van Rossum y lanzado en 1991, y desde entonces se ha consolidado como una herramienta esencial en múltiples áreas del desarrollo de software (Wikipedia, 2025b).

Entre sus características principales destacan su soporte para múltiples paradigmas de programación, como la programación orientada a objetos, imperativa y funcional (Wikipedia, 2025b). Python utiliza tipado dinámico y gestión automática de memoria, lo que significa que las variables no requieren declaración explícita de tipo, sino que se asigna automáticamente en tiempo de ejecución (SISE, 2024). Además, es un lenguaje interpretado, ya que no necesita compilarse antes de ejecutarse, y multiplataforma, ya que es compatible con varios sistemas operativos como Windows, macOS y Linux (Wikipedia, 2025b).

Python cuenta con una biblioteca extensa y una comunidad activa que contribuye constantemente con módulos y librerías que facilitan el desarrollo de aplicaciones en áreas como desarrollo web, ciencia de datos, inteligencia artificial y automatización, entre otras (SISE, 2024; Wikipedia, 2025b).

### 1.5.2. PyTorch

Figura 2: Logo de PyTorch.

PyTorch es un framework de *deep learning* de código abierto desarrollada por Facebook's AI Research Lab (ahora Meta AI). Está diseñada para la creación y entrenamiento de modelos de redes neuronales, combinando la biblioteca de *machine learning* Torch con una API de alto nivel basada en Python (Bergmann & Stryker, 2023; NVIDIA, 2025).

Una de las características principales de PyTorch es su sistema de diferenciación automática, que permite calcular gradientes de manera eficiente, facilitando la optimización de modelos de redes neuronales. Además, utiliza tensores, que son arrays multidimensionales similares a los de NumPy (una biblioteca de Python para realizar cálculos con arrays y matrices), pero con la capacidad de ejecutarse en unidades de procesamiento gráfico (GPU) para acelerar los cálculos intensivos que requieren los modelos de inteligencia artificial (NVIDIA, 2025).

Gracias a su flexibilidad y facilidad de uso, PyTorch se ha convertido en una herramienta importante para comunidades académicas y de investigación (Bergmann & Stryker, 2023).

### 1.5.3. MONAI

Figura 3: Logo de MONAI.

MONAI (*Medical Open Network for AI*) es un framework de código abierto, basado en PyTorch, diseñado específicamente para el desarrollo de modelos de *deep learning* para imágenes médicas. Fue creado en 2019 mediante una colaboración entre NVIDIA, los Institutos Nacionales de Salud (NIH) de EE. UU. y el King's College London (Wikipedia, 2025a). Sus objetivos son:

- Desarrollar una comunidad de investigadores académicos, industriales y clínicos que colaboren sobre una base común.
- Crear flujos de trabajo de entrenamiento para imágenes médicas.
- Proporcionar a los investigadores un estándar para crear y evaluar modelos de *Deep Learning* (MONAI, 2025a).

Entre sus principales características destacan:

- Preprocesamiento para datos de imágenes médicas multidimensionales.
- Herramientas y funciones especialmente diseñadas para trabajar con este tipo de imágenes, como redes neuronales, funciones de pérdida, métricas de evaluación, etc.
- Soporte para paralelismo de datos en múltiples GPUs y nodos (MONAI, 2025a).

### 1.5.4. Optuna


Figura 4: Logo de Optuna.

Optuna es un framework de Python presentado por primera vez en 2019, que sirve para automatizar la búsqueda de hiperparámetros en modelos de aprendizaje automático. Los hiperparámetros son valores que no se aprenden directamente durante el entrenamiento, pero que influyen en cómo se entrena el modelo. En lugar de tener que probar manualmente diferentes combinaciones, lo hace automáticamente mediante algoritmos inteligentes, como la

optimización bayesiana y otras estrategias basadas en búsqueda secuencial. Además, incluye una técnica llamada *pruning*, que "poda" (interrumpe) aquellas configuraciones que no están mostrando buen rendimiento antes de que terminen para ahorrar tiempo. Se suele utilizar en entornos experimentales donde el rendimiento del modelo depende de una buena configuración de estos hiperparámetros (Akiba et al., 2019).

Otro característica a destacar es su flexibilidad, ya que se integra fácilmente con cualquier framework de *deep learning*, como PyTorch, y no impone una estructura fija, es decir, es el usuario quien define el espacio de búsqueda y el objetivo a optimizar. Es por esto que es de gran utilidad en este TFG.

### 1.5.5. FastAPI


Figura 5: Logo de FastAPI.

FastAPI es un framework web de alto rendimiento para construir APIs con Python 3.8 o superior, utilizado por gigantes de la tecnología como Microsoft, Netflix y Uber. Hace que la creación de aplicaciones web sea mucho más fácil a través de distintas funcionalidades, como la validación automática de datos, la gestión de errores y la documentación interactiva de la API, además de ser fácil de utilizar y aprender (Avi, 2024; Ramírez, 2024).

Una de las características más destacadas de FastAPI es su velocidad y rendimiento, comparable al de Node.js y Go. También destaca por generar automáticamente documentación interactiva de la API utilizando los estándares OpenAPI y JSON Schema, es decir, crea una página web donde se puede probar la API sin escribir código (Avi, 2024).

### 1.5.6. HTML

HTML (*HyperText Markup Language*) es el lenguaje estándar utilizado para crear y estructurar páginas web. Es fácil de aprender y utilizar, incluso para principiantes en el desarrollo web. Incluye elementos como encabezados, párrafos, listas, enlaces, imágenes y otros recursos multimedia (Martínez Perandones, 2024).

Cada elemento se representa mediante etiquetas que indican al navegador cómo debe visualizarse el contenido. Por ejemplo, la etiqueta <p>define un párrafo, mientras que <img>se utiliza para insertar imágenes.


Figura 6: Logo de HTML5.

Aunque HTML proporciona la estructura básica de una página web, tiene limitaciones de diseño. Por ello, se suele usar junto a CSS.

### 1.5.7. CSS


Figura 7: Logo de CSS3.

CSS (*Cascading Style Sheets* o Hojas de Estilo en Cascada) es un lenguaje utilizado para definir la presentación visual de documentos estructurados en lenguajes de marcado como HTML o XML (MDN Web Docs, 2025).

Puede controlar aspectos como colores, tipografías, márgenes, tamaños y disposición de los elementos en una página. Por ejemplo, es posible cambiar el color de fondo de una sección, ajustar el tamaño de una fuente o definir la posición de una imagen sin modificar el contenido HTML subyacente.

### 1.5.8. JavaScript

JavaScript es un lenguaje de programación diseñado originalmente para ser ejecutado en el navegador web, es decir, del lado del cliente (aunque también se puede usar en el lado del servidor), que permite añadir interactividad a las páginas web. En el lado del cliente, permite actualizar partes del contenido sin recargar la página o crear efectos visuales dinámicos, entre otras cosas. En cambio, en el lado del servidor permite obtener datos de una base de datos o crear APIs, por ejemplo (GeeksforGeeks, 2025b).


Figura 8: Logo de JavaScript.

Tiene acceso al DOM (*Document Object Model*), que es la representación interna del contenido HTML. Esto le permite modificar la estructura y el estilo de la página en tiempo real. Además incluye programación asíncrona, que le permite realizar tareas en segundo plano, como cargar datos del servidor o procesar archivos, mientras la página sigue siendo usable (MDN Web Docs, 2025).

En el desarrollo de la aplicación web implementada en este Trabajo de Fin de Grado, JavaScript se encarga de la interacción del usuario con la interfaz, gestionando el envío de imágenes al servidor y la obtención del resultado segmentado tras el procesamiento del modelo.

Para mayor aclaración, se resume en forma de tabla las tecnologías usadas y su uso principal en el Cuadro 1.

| **Tecnología** | **Uso principal** |
|---|---|
| Python | Lenguaje de programación para la lógica del servidor |
| PyTorch | Construcción y entrenamiento de modelos *deep learning* |
| MONAI | Transformaciones de imágenes, métricas, modelos ViT, etc. |
| Optuna | Ajuste automático y optimización de hiperparámetros |
| FastAPI | Creación de API web del lado del servidor |
| HTML | Estructura de la interfaz web y formularios |
| CSS | Estilos visuales de la interfaz (colores, fuentes, ...) |
| JavaScript | Envío de imágenes al servidor y validaciones del cliente |

Cuadro 1: Tecnologías usadas y su uso principal.

# 2

# Estudio del Estado del Arte

## 2.1. Inteligencia Artificial

La inteligencia artificial (IA) es una rama de la informática que busca desarrollar sistemas capaces de realizar tareas que normalmente requieren inteligencia humana, como el aprendizaje, el razonamiento, la percepción y la toma de decisiones. Estos sistemas pueden analizar datos, identificar patrones y adaptarse a nuevas situaciones. Abarca desde algoritmos simples hasta redes neuronales complejas que imitan el funcionamiento del cerebro humano (Plan de Recuperación, Transformación y Resiliencia, 2023).

La inteligencia artificial emplea algoritmos y modelos matemáticos para analizar grandes volúmenes de datos y tomar decisiones basándose en patrones y reglas que se definen mediante el aprendizaje automático. Este aprendizaje permite que las máquinas puedan aprender de forma autónoma a partir de datos sin ser programada específicamente para hacerlo (Plan de Recuperación, Transformación y Resiliencia, 2023).

La IA tiene múltiples aplicaciones en diversos sectores, algunas son:

- **Salud**: Se utiliza para analizar imágenes médicas, diagnosticar enfermedades y desarrollar tratamientos personalizados.
- **Asistentes virtuales inteligentes**: Herramientas como Siri, Alexa o Google Assistant emplean IA para interactuar con los usuarios y realizar tareas mediante comandos de voz.
- **Reconocimiento facial y de imágenes**: Aplicaciones que identifican rostros o elementos en fotografías y videos, utilizadas en seguridad y redes sociales.
- **Sistemas de recomendación personalizados**: Plataformas como Netflix o Amazon analizan el comportamiento del usuario para sugerir contenidos o productos (DocuSign, 2025).

- **Automatización industrial**: Una aplicación destacada es el mantenimiento predictivo, donde sistemas basados en IA analizan datos en tiempo real de sensores instalados en maquinaria para anticipar fallos (LogicMelt, 2024).

Estas aplicaciones demuestran cómo la inteligencia artificial está transformando diversos aspectos de la sociedad y ofreciendo nuevas soluciones a problemas complejos.

## 2.2. Machine Learning

El aprendizaje automático (*machine learning*) es un subconjunto de la inteligencia artificial que permite a un sistema aprender directamente de los datos, sin ser programado explícitamente para realizar cada tarea. En lugar de seguir instrucciones fijas, los algoritmos de machine learning analizan grandes volúmenes de datos para identificar patrones y generalizar comportamientos (Google Cloud, 2025a).

El proceso típico de aprendizaje consiste en:

1. **Procesamiento de datos**: Se basa en recopilar y preparar la información necesaria para entrenar el modelo de aprendizaje automático.

2. **Desarrollo del modelo**: Se selecciona un algoritmo adecuado para el objetivo del proyecto, se entrena con los datos procesados y se ajusta mediante pruebas y validaciones.

3. **Despliegue del modelo**: Se integra el modelo en un entorno de producción, permitiendo su uso en aplicaciones reales (Belcic & Stryker, 2025).

Existen distintos tipos:

- **Aprendizaje supervisado**: Utiliza conjuntos de datos etiquetados, es decir, datos que ya tienen una salida o resultado conocido. El modelo aprende a relacionar las entradas con las salidas correctas durante la fase de entrenamiento, y luego puede predecir resultados para nuevos datos no vistos.

- **Aprendizaje no supervisado**: Se utiliza cuando los datos no están etiquetados. El objetivo es descubrir patrones, relaciones o estructuras ocultas dentro del conjunto de datos sin intervención humana.

- **Aprendizaje semisupervisado**: Combina una pequeña cantidad de datos etiquetados con una gran cantidad de datos no etiquetados. Se puede usar el aprendizaje no supervisado para identificar grupos de datos y el aprendizaje supervisado para etiquetar los grupos.

- **Aprendizaje por refuerzo**: Se basa en un proceso de prueba y error. Un agente interactúa con un entorno, toma decisiones y recibe recompensas o penalizaciones según su rendimiento (Algotive, 2022).

En la Figura 9 se pueden distinguir los tipos de aprendizaje automático.


Figura 9: Tipos de aprendizaje automático.

## 2.3. Deep Learning

El aprendizaje profundo (*deep learning*) es un subconjunto del *machine learning* que emplea redes neuronales de múltiples capas, denominadas redes profundas, para modelar relaciones complejas en los datos. Las capas iniciales captan patrones simples como bordes o texturas, mientras que capas superiores combinan esas características para reconocer objetos, conceptos o comportamientos más abstractos (Holdsworth & Scapicchio, 2024).

La mayor diferencia entre el *deep learning* y los modelos tradicionales de *machine learning* radica en la cantidad de capas utilizadas. Los modelos tradicionales emplean una o dos capas, mientras que las redes profundas pueden tener decenas o incluso cientos de ellas. Esto les permite aprender representaciones de datos más complejas (Google Cloud, 2025b). Además, durante el entrenamiento, estas redes ajustan miles o millones de parámetros mediante algoritmos de optimización y retropropagación (*backpropagation*), generalmente usando grandes conjuntos de datos (Holdsworth & Scapicchio, 2024).

Tanto el *machine learning* y el *deep learning* se pueden utilizar en el reconocimiento de imágenes o voz y el procesamiento del lenguaje natural. No obstante, el *deep learning* suele

ofrecer mejores resultados que el *machine learning* tradicional en tareas complejas como la clasificación de imágenes, la detección de objetos y la segmentación semántica, ya que es capaz de aprender representaciones jerárquicas de los datos (Google Cloud, 2025b). Es por este motivo que el *deep learning* es de vital importancia en la Visión Artificial (*Computer Vision*), y por tanto, en este TFG.

## 2.4. Visión Artificial

La Visión Artificial (*Computer Vision*) es una rama de la inteligencia artificial dedicada a desarrollar sistemas informáticos capaces de analizar y comprender imágenes y vídeos de forma automática. Específicamente, se trata de la detección de objetos, clasificación y segmentación de imágenes, reconocimiento facial y seguimiento de movimiento (GeeksforGeeks, 2025a). Sus etapas son: primero, la adquisición de imágenes o vídeos mediante dispositivos como cámaras o drones; luego, su procesamiento para extraer características relevantes como formas, texturas o patrones; y finalmente, la clasificación o interpretación de la información visual mediante modelos de aprendizaje automático y redes neuronales profundas, que permiten reconocer objetos o segmentar imágenes o vídeos (Ambika, 2023).

Tiene diversas aplicaciones, como:

- **Vehículos autónomos**, para identificar señales y obstáculos en tiempo real.
- **Industria**, para detectar defectos en productos de forma automatizada.
- **Agricultura**, para supervisar cultivos y detección de enfermedades.
- **Salud**, para analizar imágenes médicas y facilitar diagnósticos.
- **Seguridad**, mediante sistemas de reconocimiento facial y vigilancia (Ambika, 2023).

Véase la Figura 10 para una representación visual de las relaciones entre IA, *Machine Learning*, *Deep Learning* y *Computer Vision*.

### 2.4.1. Segmentación de imágenes

La segmentación de imágenes es una técnica utilizada en la Visión Artificial que consiste en dividir una imagen digital en múltiples regiones o segmentos homogéneos. Su principal objetivo es asignar una etiqueta de clase a cada píxel (Ultralytics, 2025). Esto significa que cada píxel de la imagen se clasifica según lo que representa, por ejemplo, en el contexto de este TFG, "lesión" o "tejido sano / no lesión".

Los algoritmos de segmentación examinan la imagen píxel por píxel agrupando los que comparten ciertas características en común como color, intensidad, textura o localización. Actualmente, se utilizan mayoritariamente algoritmos basados en *deep learning*, especialmente

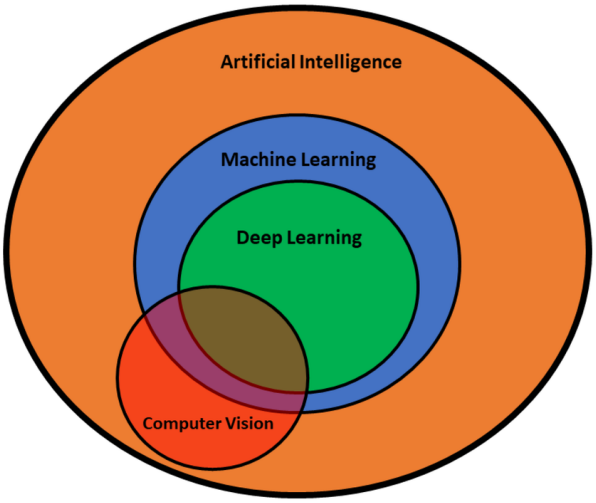


Figura 10: Diagrama de Venn de las relaciones entre IA, *Machine Learning*, *Deep Learning* y *Computer Vision*.

las redes neuronales convolucionales (CNNs), ya que aprenden representaciones jerárquicas para realizar clasificaciones píxel por píxel (Ultralytics, 2025). Así que el resultado suele ser una máscara de segmentación en la que cada píxel está etiquetado según su clase.

La segmentación se divide en 3 grupos:

- **Segmentación semántica**: Cada píxel de la imagen se clasifica según lo que representa, pero no se diferencia de otros objetos iguales. Siguiendo con el ejemplo de las lesiones, si hay varias, todas tienen la misma etiqueta.

- **Segmentación por instancias**: Distingue cada objeto por separado, incluso si pertenecen a la misma clase. Por ejemplo, si hay tres lesiones, las reconoce como lesión 1, lesión 2 y lesión 3.

- **Segmentación panóptica**: Combina las dos anteriores, clasificando cada píxel y distinguiendo cada objeto por separado (Ultralytics, 2025).

El presente TFG se centra en la segmentación semántica, ya que se etiqueta cada píxel con la clase "lesión" o "no lesión", sin distinguir entre distintos tipos ni contar lesiones individualmente.

## 2.5. Imágenes de resonancia magnética

Como se ha explicado en la sección 1.1, la esclerosis múltiple es una enfermedad en la que el sistema inmunológico ataca por error la mielina. Esto produce lesiones llamadas placas

desmielinizantes, que se caracterizan por un alto contenido en agua (Fundación Esclerosis Múltiple, 2023).

Las imágenes de resonancia magnética (RM) se utilizan para ver estas alteraciones. Existen varios tipos, donde cada uno resalta distintas propiedades del tejido cerebral (Nova, 2023). Para el presente TFG se han usado como conjunto de datos los tipos T1, T2 y FLAIR.

- **T1**: muestra mejor las estructuras con alto contenido de grasa. En esta secuencia el líquido cefalorraquídeo (LCR, o CSF en inglés) es de color negro, la sustancia gris es gris, la sustancia blanca es blanca, los huesos de color negro y el tejido adiposo de color blanco (Nova, 2023).

- **T2**: muestra mejor las estructuras con alto contenido de agua. Luego, el LCR se ve como blanco, la sustancia gris como gris, la sustancia blanca como un color gris más oscuro, los huesos como negro y el tejido adiposo como blanco (Nova, 2023).

- **FLAIR**: es una secuencia basada en T2, pero eliminando el LCR. De este modo, el LCR aparece como negro, mientras que las lesiones con alto contenido de agua son muy brillantes, como las producidas por la esclerosis múltiple, siendo así fáciles de detectar (MRI Master, 2023).

A continuación, se explican a modo de resumen las diferencias entre todas en el Cuadro 2.

| **Característica** | **T1** | **T2** | **FLAIR** |
|---|---|---|---|
| Visualización del LCR | Oscuro (negro) | Brillante (blanco) | Oscuro (suprimido) |
| Aspecto de la grasa | Brillante | Intermedia | Variable |
| Sustancia gris | Gris oscuro | Gris claro | Similar a T2 |
| Sustancia blanca | Gris claro | Gris más oscura | Mejor contraste que en T2 |
| Lesiones (como esclerosis múltiple) | Poco visibles o isointensas | Brillantes (hiperintensas) | Muy brillantes (mayor visibilidad que en T2) |
| Utilidad clínica | Anatomía detallada (estructuras cerebrales) | Detección de edemas (líquidos), inflamación, lesiones | Detección precisa de lesiones |

Cuadro 2: Comparación entre secuencias de RM T1, T2 y FLAIR.

Por último, para tener una visualización clara en mente de cada tipo se pueden observar las Figuras 11, 12 y 13, correspondientes a T1, T2 y FLAIR, respectivamente.

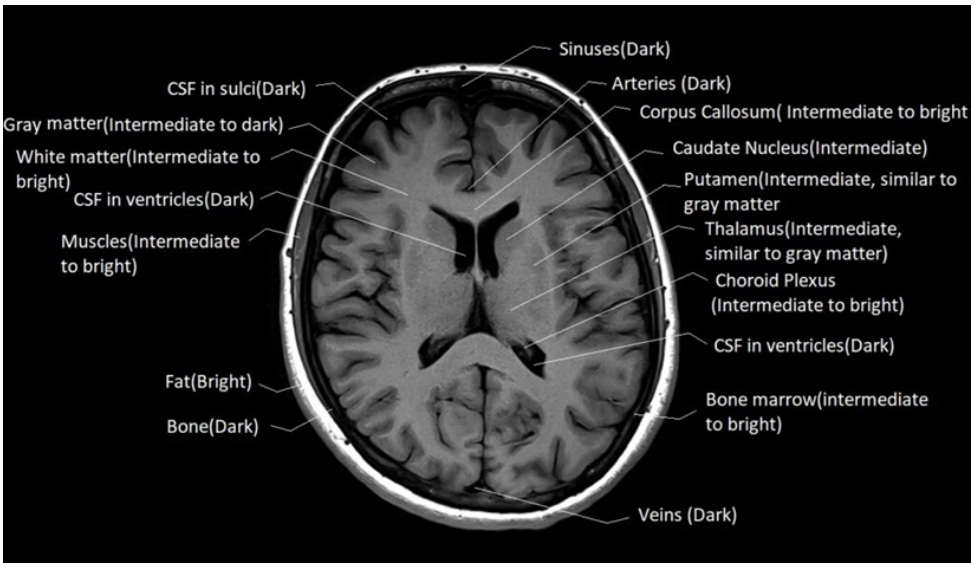


Figura 11: RM T1.

## 2.6. Redes Neuronales

Las redes neuronales son un subconjunto de *machine learning*, inspiradas en el funcionamiento del cerebro humano. Estas neuronas se agrupan en distintas capas, en las que cada neurona en una capa está conectada a neuronas de la capa siguiente, y cada conexión tiene un peso que determina la intensidad de la conexión. Estos pesos se ajustan a medida que la red aprende (IBM, 2021).

La estructura de la red está compuesta por:

- **Capa de entrada**: recibe los datos o características para que la red aprenda patrones.
- **Capas ocultas**: procesan y transforman los datos para detectar relaciones complejas.
- **Capa de salida**: produce la predicción o resultado final.

Es importante aclarar que "deep" en *deep learning* hace referencia a la cantidad de capas que tiene una red neuronal. Cuando una red cuenta con más de tres capas (incluyendo la capa de entrada y la de salida) se clasifica como un algoritmo de *deep learning*. En cambio, una red con solo dos o tres capas se considera una red neuronal simple (IBM, 2021).

Las redes neuronales aprenden en el entrenamiento, mediante:

- **Propagación hacia adelante (*forward propagation*)**: los datos se procesan capa por capa para generar una predicción.
- **Función de pérdida (*loss function*)**: se compara la predicción con el valor real para calcular el error o pérdida.

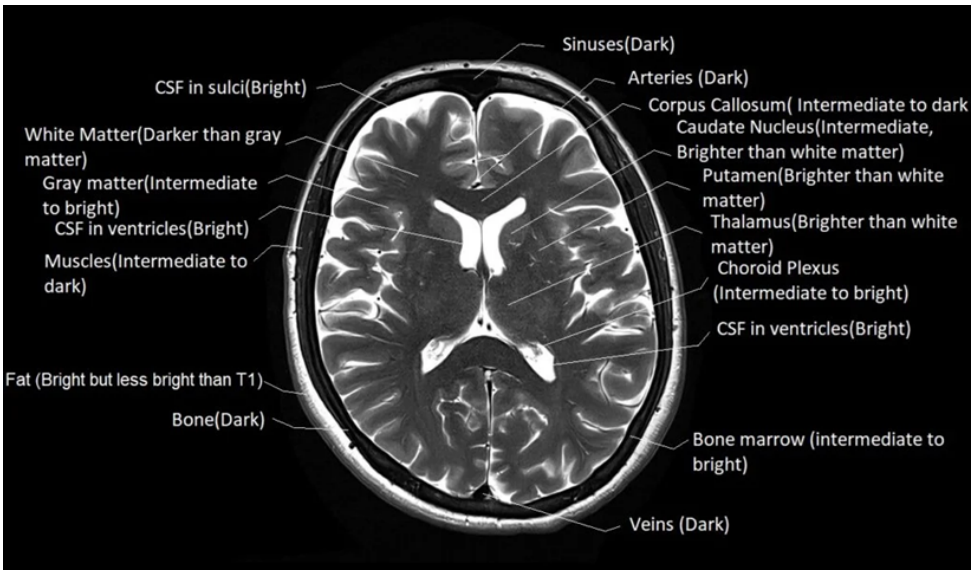


Figura 12: RM T2.

- **Algoritmos de optimización (*optimizer*)**: son fórmulas matemáticas que indican en qué dirección y cuánto se deben cambiar los pesos para que el modelo mejore.

- **Retropropagación (*backward propagation*)**: es un proceso que permite calcular cuán responsable fue cada peso del error cometido. Así, el modelo puede corregirse desde la salida hacia atrás, capa por capa (Karimian, 2024).

En la Figura 14 se puede visualizar el funcionamiento de una red neuronal.
Los principales tipos de redes neuronales son:

- **CNN (*Convolutional Neural Network*)**. Es un tipo de red que aprende a ver imágenes buscando patrones. Primero detecta cosas simples como bordes, luego formas más complejas, y al final reconoce lo que hay en la imagen. Funciona usando filtros que recorren la imagen y aprenden poco a poco cómo se ve cada objeto.

- **FNN (*Feedforward Neural Network*)**. Es el tipo más simple de red neuronal. La información va en una sola dirección, desde la entrada hasta la salida, pasando por capas ocultas. No hay ciclos ni retroalimentación, simplemente recibe datos, los procesa paso a paso y da un resultado.

- **RNN (*Recurrent Neural Network*)**. Es una red neuronal que recuerda información pasada. A diferencia de las redes normales, usa su salida anterior como parte de la entrada actual. Por esto, es ideal para trabajar con secuencias, como texto o audio. Así puede tener memoria de lo que ya ha visto (Karimian, 2024).

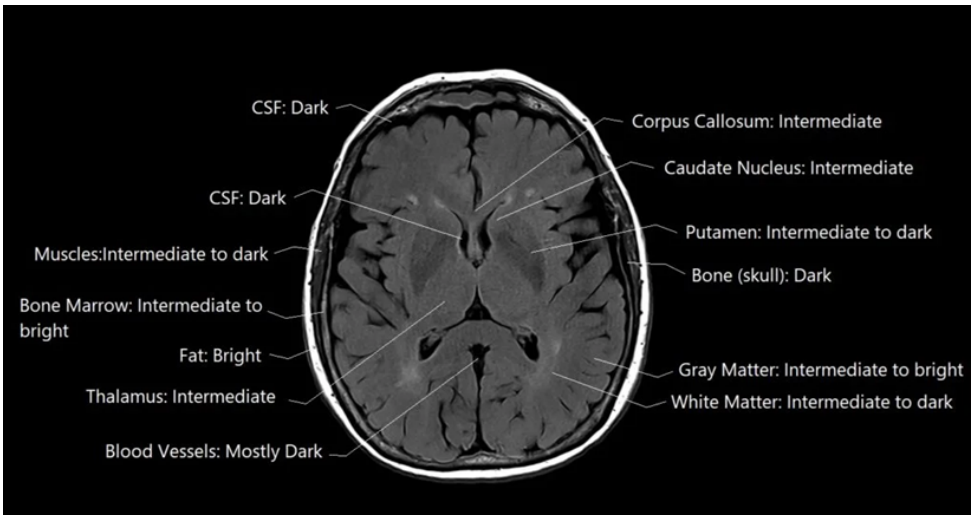


Figura 13: RM FLAIR.

En el presente TFG se trabaja con otro tipo arquitectura de redes neuronales, los *Vision Transformers* (ViT), que se explica en la sección 2.7.1.

## 2.7. Transformers

Un *transformer* es una arquitectura de red neuronal propuesta por Vaswani et al. en 2017, específicamente una arquitectura dentro del campo del *deep learning*. Se utiliza ampliamente en tareas de traducción automática y clasificación de texto.

A diferencia de modelos previos como los RNN, que procesan los datos de forma secuencial y dependen del orden para aprender relaciones entre elementos, el transformer procesa toda la secuencia de entrada de forma paralela, usando un mecanismo llamado "self-attention" para capturar relaciones entre todos los elementos de una secuencia de manera simultánea. Es decir, el transformer procesa toda la entrada simultáneamente y la hace pasar por varias capas compuestas por mecanismos de atención, que identifican qué partes son más relevantes entre sí, seguidas de pasos de normalización para estabilizar el aprendizaje, y pequeñas redes neuronales (*feedforward*) que permiten transformar la información y extraer patrones más complejos (Vaswani et al., 2017).

La atención es un mecanismo que ayuda al modelo a enfocarse en la información más relevante. Funciona comparando una pregunta (*query*) con una serie de claves (*keys*) y valores (*values*), todos representados como vectores. Calcula qué tan parecida es la query a cada key, asignando un peso a cada valor según esa similitud. Luego, combina los valores más importantes según cuánto se parecen las claves a la pregunta (Jurafsky & Martin, 2025; Vaswani et al., 2017).

El transformer fue propuesto con una arquitectura de tipo *encoder-decoder* (codificador-

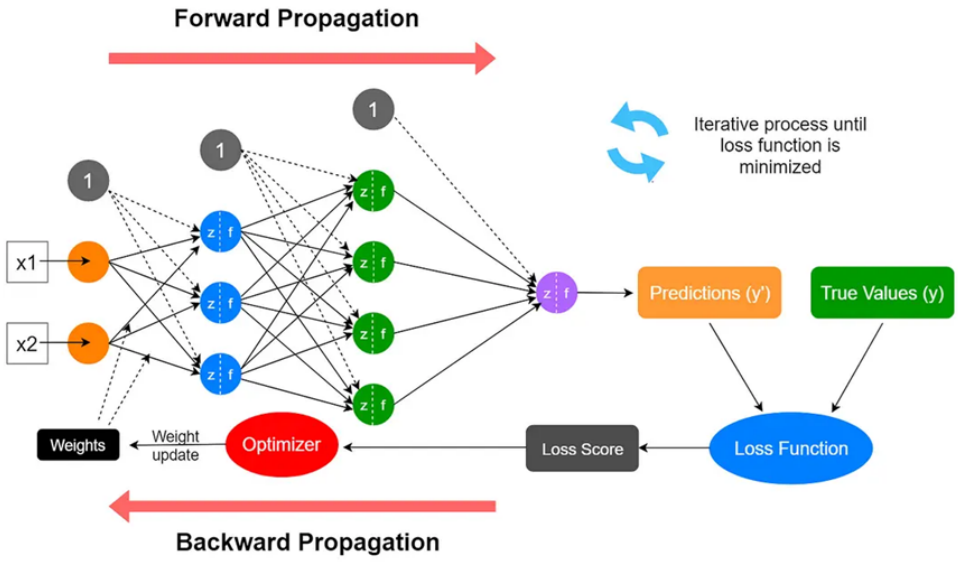


Figura 14: Funcionamiento de una red neuronal.

decodificador), especialmente pensada para tareas de traducción automática. En este esquema, el *encoder* codifica la información original en vectores que contienen contexto global, y el *decoder* genera la salida de manera autoregresiva, utilizando la información del *encoder* y lo que ha generado hasta el momento.

- El *encoder* está formado por 6 capas idénticas, cada una compuesta por dos subcapas principales: una capa de *multi-head self-attention*, que permite a cada *token* atender a todos los demás de la secuencia, y una red *feedforward* que refuerza la capacidad de representación. A estas subcapas se les añaden conexiones residuales y una normalización por capas para estabilizar el entrenamiento.

- El *decoder* también consta de 6 capas, pero incorpora una subcapa adicional de atención cruzada (*encoder-decoder attention*), que permite a cada posición de salida acceder a toda la representación generada por el *encoder*. Además, el *decoder* enmascara su mecanismo de atención propia para evitar mirar *tokens* futuros (Vaswani et al., 2017).

En la Figura 15 podemos ver dicha arquitectura, donde el *encoder* se encuentra en la izquierda y el *decoder* en la derecha.

### 2.7.1. Vision Transformers

El modelo *Vision Transformer* (ViT) fue introducido en 2020 como una adaptación del *transformer* en el campo de la Visión Artificial, aplicándose a tareas como la clasificación de imágenes, la detección de objetos y la segmentación semántica de imágenes y vídeos.

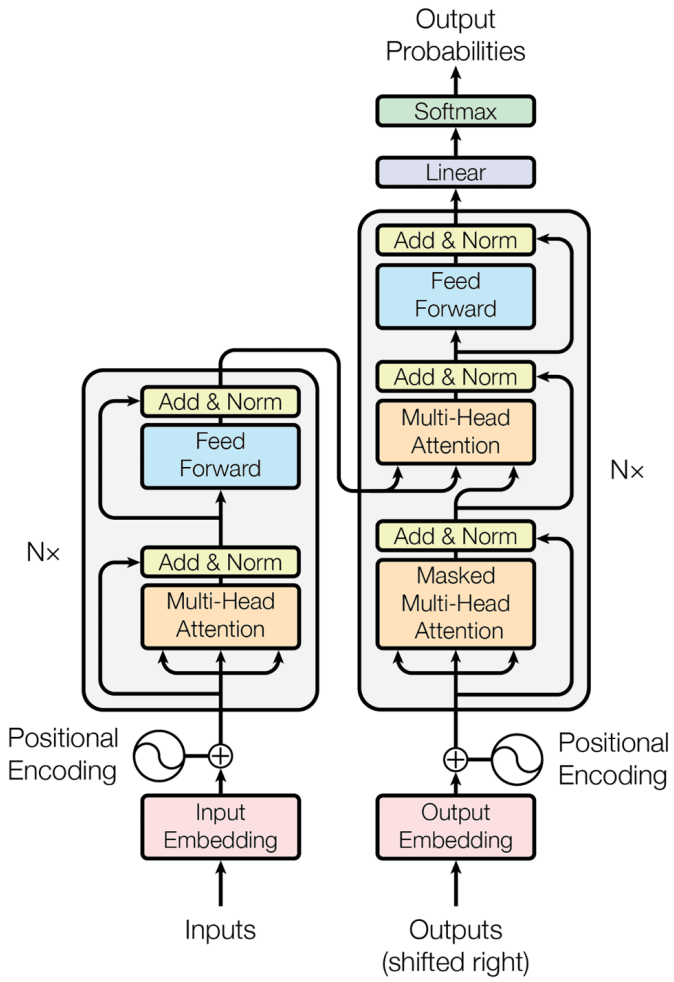


Figura 15: Arquitectura de un transformer, donde el *encoder* se encuentra en la izquierda y el *decoder* en la derecha.

Para ello, el ViT divide la imagen en pequeños parches (por ejemplo, de 16x16 píxeles), los convierte en vectores y los trata como si fueran *tokens*, de forma análoga a las palabras en el procesamiento de texto. A cada uno se le añade información de posición (*positional embedding*) para que el modelo sepa en qué parte de la imagen está cada uno. Luego, se introducen en un *transformer*, que usa los mecanismos de atención descritos en la sección 2.7 para capturar relaciones globales entre regiones de la imagen, sin necesidad de usar convoluciones (Dosovitskiy et al., 2021).

En la Figura 16 se puede observar una representación visual de la arquitectura ViT.

Existen dos grandes tipos:

1. **Transformers planos (*plain ViT*)**: eliminan por completo el uso de capas convolucionales. El concepto de "plain" se debe a que cuando la imagen es divida en parches, estos son aplanados y embebidos antes de pasar por los bloques de *self-attention* del

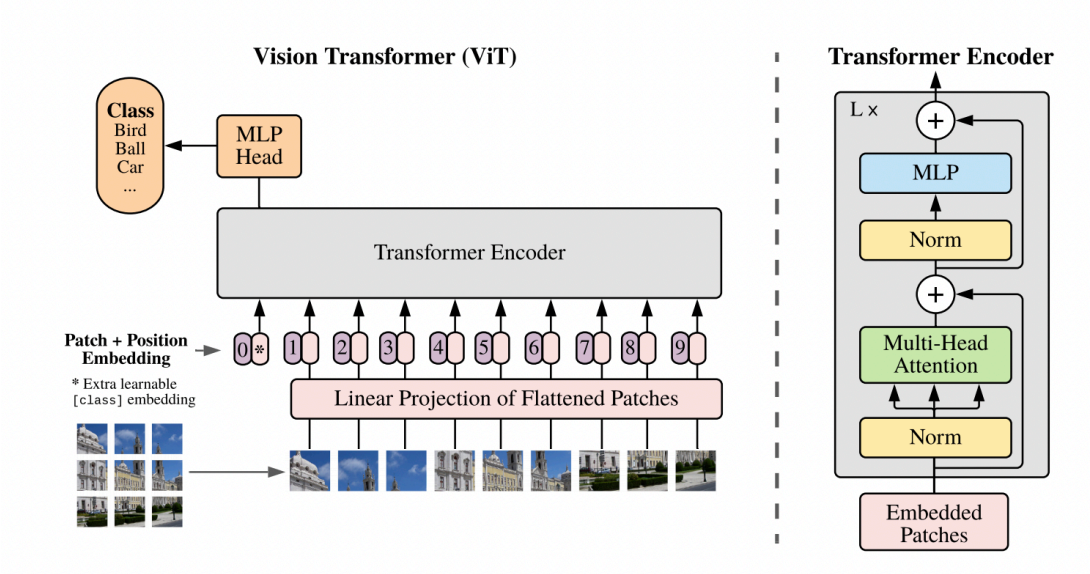


Figura 16: Arquitectura del *Vision Transformer*. El modelo divide una imagen en varios parches de tamaño fijo, los convierte en vectores y añade información de posición (izquierda). Luego, el resultado se introduce en el *encoder* del *transformer* (derecha).

*transformer*.

2. **Modelos híbridos**: integran una red convolucional para extraer características locales y la combinan con *transformers* que capturan relaciones globales en la imagen (Li et al., 2024).

El presente TFG se enfoca en los ViT híbridos, ya que un *plain ViT* solo da una etiqueta para toda la imagen porque está hecho para clasificar. Para segmentar cada píxel hace falta una "cabeza de segmentación", que es una parte extra del modelo que toma las salidas del *transformer* y las convierte en un mapa de píxeles (una máscara). Sin esa cabeza de segmentación, el ViT solo no puede segmentar (Thisanke et al., 2023).

Es importante destacar cómo la integración de *transformers* en modelos de segmentación ha permitido combinar los puntos fuertes de las CNN (capturando detalles locales) con la capacidad de los Transformers para modelar relaciones globales entre píxeles. Aunque al principio el rendimiento era inferior al de las CNN, con suficiente cantidad de datos y poder de cómputo, los ViT logran o incluso superan el rendimiento de los modelos convolucionales (Dosovitskiy et al., 2021; Thisanke et al., 2023).

# 3
# Preprocesamiento

Antes de entrenar el modelo, es necesario aplicar un preprocesamiento a las imágenes de resonancia magnética. El objetivo principal de este proceso es garantizar que todas las imágenes tengan un formato uniforme y sean homogéneas para introducirlas en el modelo de segmentación. Es decir, aplicar una serie de transformaciones, modificando las imágenes como sea necesario y convirtiéndolas en el formato correcto para cargarlas. Asimismo, este preprocesamiento se aplica también a las imágenes que los usuarios suben a la aplicación web para que sean compatibles con el modelo.

Por otra parte, es importante destacar que el proyecto ha sido desarrollado, preprocesado y entrenado en un equipo con un procesador AMD Ryzen 7 7840HS w/ Radeon 780M Graphics (8 núcleos), NVIDIA GeForce RTX 4060 Laptop GPU con 8GB VRAM y 32GB RAM.

## 3.1. Estructura de los datos

Cada paciente dispone de resonancias magnéticas obtenidas en tres modalidades distintas: T1, T2 y FLAIR, además de una máscara binaria que indica las regiones lesionadas. Las imágenes están almacenadas en formato .nii.gz (NIfTI), comúnmente utilizado en neuroimagen.

Los datos están organizados por carpetas, primero por paciente (por ejemplo, P1, P2, etc.), y dentro de cada paciente, por punto temporal (T1, T2, etc.), ya que pueden haberse tomado en diferentes fechas. Dentro de cada subcarpeta se encuentran los archivos correspondientes a las tres modalidades y la máscara.

## 3.2. Transformaciones

A continuación, se describen las transformaciones aplicadas a las imágenes. Cabe destacar que algunas de estas transformaciones no son necesarias, por ejemplo, si el conjunto de datos ya está preprocesado, normalizado y alineado adecuadamente, o si todas las imágenes tienen la misma orientación, resolución y formato que requiere el modelo. En el presente TFG, se proponen las más importantes en caso de utilizar cualquier otro conjunto de datos menos preprocesado.

### 3.2.1. Registro

El objetivo del registro (co-registration) es alinear imágenes de diferentes modalidades o momentos temporales a una referencia común de forma que las estructuras anatómicas coincidan exactamente en todas ellas. Este paso es muy importante porque las imágenes pueden no estar perfectamente superpuestas debido a pequeños movimientos o diferencias en el escaneo, por ejemplo, al ser obtenidas usando distintos protocolos.

Cada modalidad (T1, T2 y FLAIR) y la máscara han sido alineadas espacialmente respecto a la imagen T1 de cada paciente. Se ha elegido T1 como referencia porque es la imagen anatómica base estándar en estudios cerebrales, debido a su alta resolución anatómica (como se ha explicado en la sección 2.5).

El registro se realizó utilizando la librería SimpleITK, mediante la métrica *Mattes Mutual Information*. Esta métrica mide la dependencia estadística entre las intensidades de dos imágenes, que sirve para evaluar cuán bueno es el alineamiento entre las imágenes. Por eso se utiliza para el registro de imágenes de distintas secuencias (por ejemplo, T1 frente a T2 o FLAIR), ya que no exige que los valores sean iguales, solo que exista una relación consistente entre ellos. Una vez calculada la transformación que alinea la imagen móvil (T2, FLAIR o la máscara) con la fija (T1), se aplica mediante un proceso de interpolación. Los nuevos puntos resultantes no tienen por qué coincidir exactamente con los vóxeles originales, luego es necesario estimar los valores de intensidad en esas posiciones intermedias. Para ello, se utilizó interpolación lineal, que estima el valor de cada nuevo vóxel como una media ponderada de los vóxeles vecinos más cercanos. Este método se utiliza para imágenes continuas como T2 o FLAIR, ya que contienen intensidades que varían suavemente entre regiones. De este modo, se preserva esa continuidad, evitando la aparición de bordes artificiales o distorsiones que podrían alterar la anatomía representada. Sin embargo, en el caso de las máscaras de segmentación, que contienen solo valores discretos (0 y 1), se usa la interpolación de vecinos más cercanos (*nearest*), para conservar la naturaleza binaria de la imagen sin introducir valores no válidos (Climent Pardo, 2024).

### 3.2.2. Remuestreo de vóxeles

El remuestreo cambia el tamaño de los vóxeles (el equivalente a píxel pero en 3D) para que todas las imágenes y la máscara tengan la misma resolución, lo cual es útil cuando se trabaja con datos de distintas fuentes o cuando se necesita comparar imágenes entre sí. Este paso también es importante para realizar análisis que requieran que los vóxeles tengan un tamaño uniforme. Además, al hacer remuestreo, a veces se reduce la cantidad de datos, lo que hace que sea más fácil almacenarlos y procesarlos. Un tamaño común que se usa es 1x1x1 vóxeles, llamado tamaño isotrópico, (Climent Pardo, 2024), y es por este motivo que se ha remuestreado a este tamaño en este TFG.

Para llevar a cabo dicho remuestro, se ha utilizado la función *reslice* de DIPY, que aplica internamente una interpolación según el orden indicado. Como se ha explicado en la sección 3.2.1, para las imágenes continuas se utiliza la interpolación trilineal (orden 1), y para la máscara la interpolación por vecinos más cercanos (orden 0).

### 3.2.3. Corrección del campo de sesgo

En imágenes de resonancia magnética, el campo de sesgo (*bias field*) es una variación gradual y lenta en la intensidad de la imagen. No es ruido aleatorio ni errores, sino una variación suave que ocurre, por ejemplo, porque la bobina que capta la señal no tiene la misma sensibilidad en todo el volumen de la imagen.

La corrección del campo de sesgo ayuda a eliminar estas variaciones de intensidad que no tienen que ver con el tejido real, sino con los fallos del propio escáner. Para corregir esto, se ha usado el algoritmo N4, que modela el campo de sesgo como una distorsión suave y de baja frecuencia que afecta de forma multiplicativa a la imagen. Es así porque el sesgo no cambia bruscamente, sino de forma gradual, y no añade intensidad, sino que la aumenta o disminuye según la zona, como si aplicara un filtro de intensidad desigual sobre la imagen. Este algoritmo estima esa distorsión y la elimina, ajustando las intensidades de los vóxeles para obtener una imagen más homogénea y fiel a las características reales del tejido. Matemáticamente, el algoritmo trabaja con el logaritmo natural de las intensidades, ya que al transformar la multiplicación en una suma, resulta más sencillo estimar y eliminar esta distorsión suave. Así, N4 calcula esa variación en el dominio logarítmico y la elimina, para luego volver a la escala original de intensidades. De este modo, ajusta las intensidades de los vóxeles para obtener una imagen más homogénea y fiel a las características reales del tejido.

Esta corrección es especialmente útil en la segmentación o el análisis de la intensidad. Para ello, se ha tomado como referencia el ejemplo presentado en (Climent Pardo, 2024).

En la Figura 17 se puede observar un ejemplo de la correción del campo de sesgo, donde la imagen con campo de sesgo sin corregir está a la izquierda, y la corregida a la derecha.

### 3.2.4. Normalización

La normalización de intensidades tiene como objetivo ajustar los valores de intensidad de las imágenes de resonancia magnética para que sigan un mismo estándar o referencia. Esto hace que sean consistentes entre sí, incluso si provienen de diferentes fuentes, y se puedan realizar comparaciones, estudios y análisis estadísticos (Climent Pardo, 2024).

En este TFG, se han escalado las intensidades a un rango fijo [0, 1] para garantizar que todas las imágenes tengan una escala de valores homogénea, utilizando el método min-max, calculado como $\frac{x-\text{mín}(x)}{\text{máx}(x)-\text{mín}(x)}$. Aunque esta técnica sea sensible a valores a extremos, en este

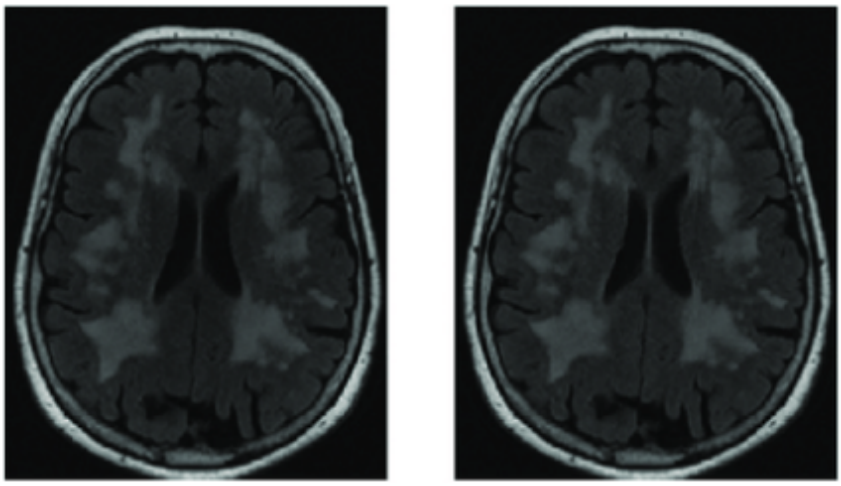

Figura 17: Imagen con campo de sesgo sin corregir y corregida, respectivamente.

caso funciona bien porque las intensidades de las imágenes tienen un rango parecido, es decir, no hay valores extremos.

### 3.2.5. Carga y conversión a tensores

Para cargar las imágenes y convertirlas a tensores se han utilizado las transformaciones ya definidas en MONAI.

- ***LoadImaged***: Esta transformación carga las imágenes (resonancias magnéticas) y etiquetas (máscara) desde sus archivos en formato NIfTI. Convierte los datos "crudos" almacenados en disco a una estructura en memoria para poder trabajar con ellos en el código. Específicamente, las guarda en un diccionario, que tiene como claves los nombres que se quieran definir, y los valores son los datos cargados correspondientes (en este caso, las imágenes y la máscara en formato de tensor).

- ***EnsureChannelFirstd***: Las imágenes a veces vienen con las dimensiones organizadas de distintas formas. Por ejemplo, el canal de la imagen, que representa diferentes modalidades o secuencias de imágenes (como en una resonancia magnética donde cada modalidad T1, T2, FLAIR es un canal diferente), puede estar al final en lugar de estar al principio.

  PyTorch necesita que el canal esté primero para poder procesar correctamente los datos. Esta transformación reorganiza las dimensiones del tensor para que quede en el orden correcto: [canal, alto, ancho, profundidad].

- ***EnsureTyped***: Los datos deben ser convertidos a tensores, ya que es el tipo de dato que las redes neuronales en Pytorch usan para hacer cálculos durante el entrenamiento o la

inferencia. Un tensor es una estructura de datos similar a un array multidimensional, pero optimizada para operaciones matemáticas en GPU y CPU. Esta transformación se encarga de convertir las imágenes y las etiquetas en tensores de Pytorch, asegurando que el tipo y la estructura de los datos sean correctos (MONAI, 2025b).

### 3.2.6. Corrección de la orientación

Cada dimensión del volumen de las imágenes corresponde a una dirección espacial distinta. Se definen mediante etiquetas, por ejemplo, una dimensión puede estar etiquetada como L (*Left*, izquierda) o R (*Right*, derecha); otra dimensión puede tener las etiquetas P (*Posterior*, parte trasera) o A (*Anterior*, parte frontal); y la tercera dimensión puede estar definida por I (*Inferior*, abajo) o S (*Superior*, arriba). Sin embargo, la forma en que estas dimensiones están alineadas puede variar entre diferentes imágenes si provienen de distintas fuentes, por lo que es necesario estandarizarlas para que todas tengan la misma orientación espacial. Para ello, se ha utilizado la transformación *Orientationd* de MONAI, ajustando las imágenes a la orientación RAS (*Right-Anterior-Superior*), una convención muy usada en neuroimagen (MONAI, 2025b).

En la Figura 18 se puede observar un ejemplo de la correción de la orientación a RPI, donde la imagen original está en la fila de arriba, y la corregida en la fila de abajo.

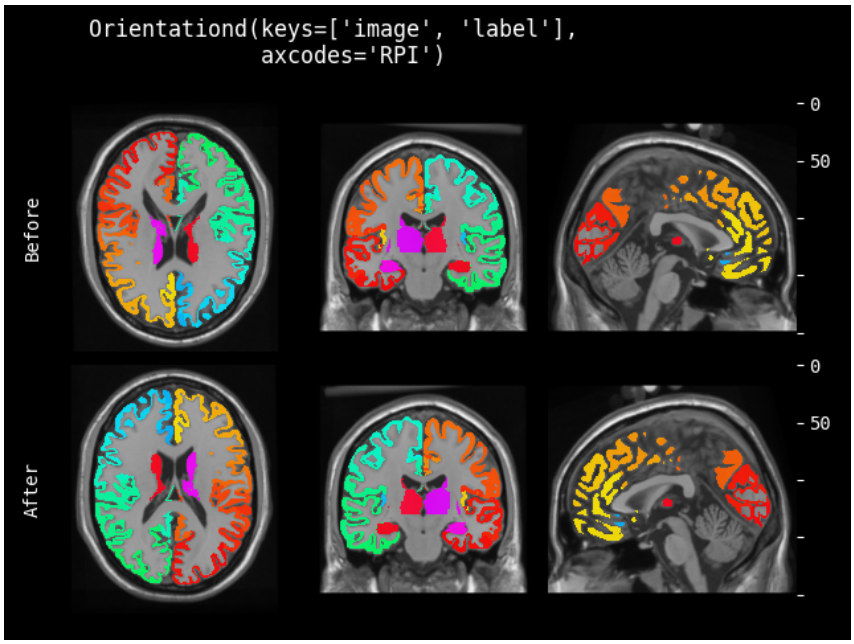


Figura 18: Imagen original en la fila de arriba y corregida con orientación RPI en la fila de abajo.

### 3.2.7. Redimensionado

La transformación *Resized* de MONAI cambiar el tamaño espacial de las imágenes, es decir, el número de vóxeles en cada eje (alto, ancho y profundidad). Internamente, MONAI interpola los valores de la imagen para adaptarla al nuevo tamaño, utilizando interpolación trilineal para imágenes continuas (como T1, T2 o FLAIR) y por vecinos más cercanos en el caso de las máscaras, como se ha explicado en numerosas ocasiones (MONAI, 2025b).

En este caso se ha elegido un tamaño de 96x96x96 vóxeles, debido al elevado coste computacional requerido. Sin embargo, se podría haber elegido 128x128x128 vóxeles, mejorando la resolución espacial y la precisión en la segmentación, ya que el modelo contaría con más detalles para aprender. No obstante, este aumento en la resolución implicaría un mayor consumo de memoria y tiempo de entrenamiento. Por ello, el tamaño de $96^3$ se considera un buen compromiso entre calidad y eficiencia computacional.

### 3.2.8. Concatenación de imágenes

Como se ha mencionado en múltiples ocasiones, en este TFG se han utilizado tres secuencias distintas de resonancia magnética por paciente: T1, T2 y FLAIR. Aunque estas imágenes se procesan de forma independiente durante las primeras etapas del preprocesamiento, es decir, durante el registro, el remuestreo, la corrección del campo de intensidad y la normalización, el modelo necesita recibirlas como una única entrada multicanal. Aquí es donde entra en juego la transformación *ConcatItemsd* de MONAI, que permite combinar múltiples imágenes en un solo tensor.

Específicamente, toma la lista de imágenes asociadas a la clave *image*, que en este caso contiene las tres modalidades ya alineadas y redimensionadas, y las concatena a lo largo del eje del canal, es decir, crea un único tensor con forma [3, H, W, D], donde cada canal corresponde a una de las secuencias (T1, T2 o FLAIR) (MONAI, 2025b).

A continuación, se presenta a modo de resumen el cuadro 3.2.8 con todas las transformaciones realizadas.

| Transformación | Descripción | Herramienta utilizada | Aplicado a |
|---|---|---|---|
| **Registro** | Alinea las secuencias T2, FLAIR y la máscara respecto a la imagen T1 de cada paciente. | *Set Metric As Mattes Mutual Information* (de SimpleITK) | Imagen y máscara |
| **Remuestreo de vóxeles** | Cambia la resolución a tamaño isotrópico 1x1x1 mm. | *reslice* (de DIPY) | Imagen y máscara |
| **Corrección del campo de sesgo** | Elimina variaciones suaves de intensidad causadas por el escáner. | *N4 Bias Field Correction Image Filter* (de SimpleITK) | Imagen |
| **Normalización** | Escala las intensidades al rango [0, 1] para homogeneizar los datos. | Técnica min-max | Imagen |
| **Carga de datos** | Carga las imágenes y máscaras desde archivos NIfTI. | *LoadImaged* (de MONAI) | Imagen y máscara |
| **Canal primero** | Reordena las dimensiones al formato [C, H, W, D] requerido por PyTorch. | *Ensure Channel Firstd* (de MONAI) | Imagen y máscara |
| **Conversión a tensor** | Convierte los datos a tensores compatibles con PyTorch. | *EnsureTyped* (de MONAI) | Imagen y máscara |
| **Corrección de orientación** | Ajusta las imágenes a la orientación RAS (*Right-Anterior-Superior*). | *Orientationd* (de MONAI) | Imagen y máscara |
| **Redimensionado** | Ajusta el tamaño de todas las imágenes a 96x96x96 vóxeles. | *Resized* (de MONAI) | Imagen y máscara |
| **Concatenación** | Une T1, T2 y FLAIR en un único tensor multicanal de forma [3, H, W, D]. | *ConcatItemsd* (de MONAI) | Imagen |

Cuadro 3: Transformaciones aplicadas durante el preprocesamiento.

# 4
# Entrenamiento

En este capítulo se explica el proceso completo de entrenamiento seguido para los modelos basados en Vision Transformers, desde la elección de las arquitecturas hasta la selección del modelo final. Primero, se describen los modelos ViT utilizados y la métrica de evaluación usada para evaluar su rendimiento. Luego, se detalla la estrategia de entrenamiento del modelo y su optimización, ajustando sus hiperparámetros y el uso de técnicas de *data augmentations* para que el modelo sea más robusto y aprenda a generalizar mejor.

## 4.1. Modelos ViT utilizados

### 4.1.1. SwinUNETR

Swin UNEt TRansformers (Swin UNETR) es una arquitectura propuesta por Hatamizadeh et al. (2022) que utiliza una red tipo U-Net (un modelo usado en segmentación médica con una estructura con forma de U, donde primero contrae la información para extraer características profundas y luego la expande para recuperar la resolución original) con un Swin Transformer como *encoder* y lo conecta a un *decoder* basado en CNN en distintas resoluciones mediante conexiones tipo *skip*. Estas conexiones permiten que el *decoder* utilice directamente las características extraídas por el *encoder*, de modo que el modelo pueda integrar tanto la información global como los detalles locales, y así obtener una segmentación más precisa.

Para ello, el modelo recibe imágenes de resonancias magnéticas 3D multimodales con cuatro canales. Se dividen en pequeños parches 3D sin que se superpongan entre ellos. Cada parche se lleva a un espacio embebido para que sea más fácil trabajar con la información. En el *encoder*, el Swin Transformer agrupa los parches en ventanas pequeñas para captar detalles locales, y desliza esas ventanas entre capas, de modo que en cada capa se consideran agrupaciones diferentes. Además, en cada etapa del *encoder* se aplica un *patch merging*, que reduce la resolución espacial a la vez que aumenta la cantidad de canales. Por su parte, el *decoder* reconstruye la imagen paso a paso mediante bloques convolucionales 3D, aumentando la resolución a través de capas deconvolucionales y combinando las características del *encoder* recibidas a través de las conexiones *skip*. Luego, hace una convolución 1x1x1, que es un filtro para combinar la información de todas las características extraídas en una sola salida. Por último, usa una

función que transforma esos valores en probabilidades entre 0 y 1, para que el modelo pueda decidir qué partes del volumen corresponden a la región segmentada. Así se obtiene el mapa de segmentación final, que indica qué vóxeles pertenecen a cada clase o región.

En la Figura 19 se puede observar el funcionamiento de esta arquitectura, y en la Figura 20, el mecanismo de las ventanas deslizantes.

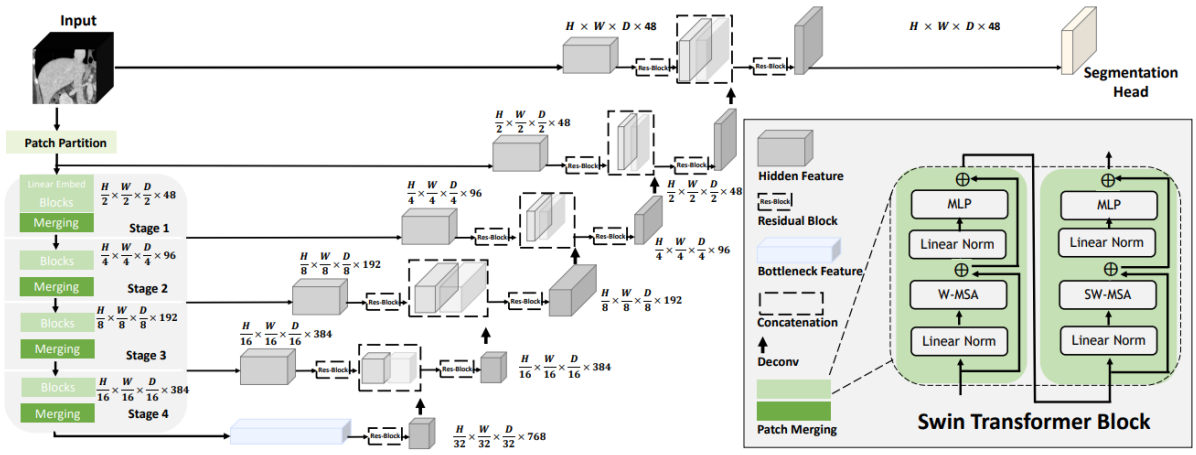


Figura 19: Arquitectura SwinUNETR. El modelo recibe imágenes de resonancia magnética 3D multimodales con 4 canales. Luego, divide la imagen en parches no superpuestos y utiliza una capa que los organiza en ventanas del tamaño adecuado para calcular la atención. Las representaciones que obtiene el transformer en el *encoder* se pasan al *decoder* CNN mediante conexiones tipo *skip*, para generar finalmente el mapa de la segmentación.

### 4.1.2. 3D-EffitViTCaps

3D-EffiViTCaps es una arquitectura de segmentación 3D que combina Transformers eficientes (EfficientViT) con redes de cápsulas para aprender mejor cómo las distintas partes de una estructura se conectan entre sí. Su diseño sigue una estructura tipo U-Net, donde el codificador reduce la resolución mientras captura tanto detalles locales como contextos globales, y utiliza cápsulas para aprender relaciones complejas entre partes de la imagen. En el decodificador, se usan tanto estos bloques EfficientViT como convoluciones tradicionales en 3D para crear la segmentación, para que la arquitectura no sea pesada.

Su funcionamiento es el siguiente: primero, el modelo recibe la imagen 3D, que pasa por un bloque que extrae características y aumenta el número de canales para que el modelo pueda trabajar mejor con la información. Después, las características se procesan mediante bloques 3D EfficientViT y bloques cápsula. Los bloques EfficientViT ayudan a entender tanto detalles pequeños como información más global. En el encoder, los bloques *3D Patch Merging* reducen la resolución espacial (*downsampling*) mientras aumentan los canales, permitiendo resumir

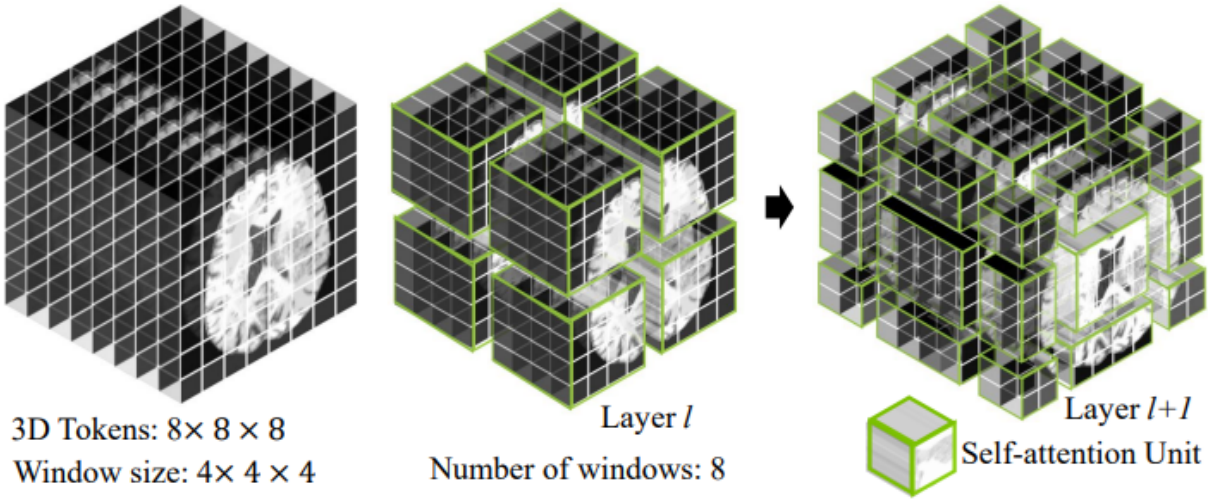


Figura 20: Mecanismo de las ventanas deslizantes en SwinUNETR, con tokens 3D de tamaño 8x8x8 y ventanas de 4x4x4.

la información en niveles más abstractos sin perder detalles clave. Además, en las capas más profundas del codificador y en la parte central, se usan bloques cápsula para aprender mejor cómo las distintas partes de una estructura se conectan entre sí. Finalmente, el decodificador genera la segmentación usando bloques EfficientViT y convoluciones tradicionales en 3D (Gan et al., 2024).

En la Figura 21 se puede observar su funcionamiento.

### 4.1.3. SegFormer3D

Segformer3D es un modelo Transformer diseñado para segmentar imágenes médicas en 3D. Su funcionamiento se basa en analizar la imagen a distintas escalas, es decir, captura información tanto de las partes más grandes y generales como de los detalles más pequeños. Así puede comprender mejor la estructura completa del volumen.

A diferencia de otros modelos, utiliza un decodificador sencillo compuesto solo por capas MLP (perceptrones multicapa), que son redes neuronales en las que cada neurona está conectada con todas las de la siguiente capa, y su función es integrar la información global y local que ha extraído el Transformer para producir máscaras de segmentación precisas sin necesidad de una arquitectura pesada.

En la Figura 22 se puede observar su funcionamiento.

Esta propuesta fue presentada por Perera et al. (2024) en el taller DEF-AI-MIA durante la conferencia CVPR 2024, y destaca por su equilibrio entre simplicidad y eficiencia.

## 4.2. Métrica de evaluación utilizada

En segmentación de imágenes médicas, una de las métricas más utilizadas es el Coeficiente de Dice, también llamado índice de Dice-Sørensen o Coeficiente de Similitud de Dice (DSC). Esta métrica mide la superposición entre la segmentación generada por el modelo y la segmentación de referencia (*ground truth*), y es especialmente adecuada para problemas donde el objetivo es delimitar regiones anatómicas o patológicas en imágenes médicas.

Matemáticamente, el coeficiente de Dice se define como:

$$\text{Dice}(A, B) = \frac{2\,|A \cap B|}{|A| + |B|}$$

donde:

- $|A|$ es número de vóxeles segmentados por el modelo,
- $|B|$ el número de vóxeles en el *ground truth* y
- $|A \cap B|$ es el número de vóxeles en los que ambos coinciden.

**Interpretación:**

La métrica toma valores entre 0 y 1, donde un valor cercano a 1 indica que la segmentación del modelo es muy similar a la segmentación manual. En cambio, un valor cercano a 0 indica que hay muy poca o ninguna coincidencia entre la predicción y el *ground truth*.

Por otra parte, cuando se aplica a datos booleanos, utilizando la definición de verdadero positivo (TP), falso positivo (FP) y falso negativo (FN), se puede expresar como:

$$DSC = \frac{2\,TP}{2\,TP + FP + FN}$$

donde:

- **Verdaderos positivos (TP)** es el número de vóxeles que el modelo clasifica correctamente como pertenecientes a la región de interés (ROI).
- **Falsos positivos (FP)** es el número de vóxeles que el modelo clasifica como parte de la ROI, cuando en realidad, no lo son.
- **Falsos negativos (FN)** es el número de vóxeles que el modelo clasifica como fondo cuando realmente pertenecen a la ROI.
- **Verdaderos negativos (TN)** es el número de vóxeles que el modelo clasifica correctamente como fondo, coincidiendo con la segmentación de referencia.

### 4.2.1. ¿Por qué esta métrica?

En segmentación médica es habitual que la región de interés (ROI) sea pequeña y el desequilibrio entre el número de vóxeles de la ROI y del fondo sea muy alto. En otras palabras, una imagen médica segmentada suele contener solo un pequeño porcentaje de vóxeles que pertenecen a las ROIs respecto del volumen total. Una segmentación que marque todo como fondo (como la métrica de *accuracy*) siempre dará como resultado una puntuación artificialmente alta, ya que incluye los verdaderos negativos. En cambio, Dice mide qué proporción de la ROI ha sido segmentada correctamente y no depende tanto del número de vóxeles que sean solo fondo.

Además, Dice combina precisión y exhaustividad (*recall*), muy útil cuando hay desequilibrio entre clases, ya que logra un equilibrio entre no segmentar áreas que no corresponden a la lesión (pocos falsos positivos) y, a la vez, captar la mayor parte posible de la región afectada (pocos falsos negativos). De este modo, el coeficiente de Dice es capaz de reflejar mejor la calidad global de la segmentación que métricas separadas como la precisión o el *recall* por sí solas (Huynh, 2023).

Por otra parte, aunque es parecida a la métrica de Jaccord (IoU), Dice da más importancia a la intersección entre la segmentación detectada y el *ground truth*, lo que hace que sea más sensible a pequeñas diferencias cuando las regiones son pequeñas, por lo que es ideal en este TFG. En cambio, IoU penaliza más estrictamente los errores, pero puede hacer que el aprendizaje sea más lento o menos estable en estos casos (Neville, 2023).

| **Métrica** | **Fórmula** | **Ventaja** | **Desventaja** |
|---|---|---|---|
| Dice | $\frac{2TP}{2TP+FP+FN}$ | Mayor sensibilidad en regiones pequeñas | Puede sobrevalorar áreas pequeñas |
| IoU | $\frac{TP}{TP+FP+FN}$ | Penaliza más errores | Penalización más severa que puede dificultar el aprendizaje |
| Precisión | $\frac{TP}{TP+FP}$ | Reduce falsos positivos | No considera falsos negativos |
| Recall | $\frac{TP}{TP+FN}$ | Reduce falsos negativos | No considera falsos positivos |
| Accuracy | $\frac{TP+TN}{TP+FP+FN+TN}$ | Fácil de interpretar | Sensible al desequilibrio de clases |

Cuadro 4: Resumen comparativo de métricas de evaluación.

## 4.3. Estrategia de entrenamiento y optimización

### 4.3.1. División de los datos

Primero, una vez habiendo realizado la carga y el preprocesamiento de las imágenes, se dividen en conjuntos de entrenamiento, validación y test.

- **Entrenamiento (70 %)**: Este conjunto se usa para que el modelo aprenda. El algoritmo ajusta automáticamente sus parámetros (los pesos y sesgos internos de la red) utilizando estas imágenes y máscaras para mejorar su capacidad de segmentación. Además, durante esta fase se establecen hiperparámetros (como la tasa de aprendizaje, número de épocas o tamaño del lote), que son configuraciones definidas antes del entrenamiento y que influyen en el proceso de aprendizaje, pero que el modelo no aprende por sí solo.

- **Validación (15 %)**: Durante el entrenamiento, este conjunto se utiliza para evaluar periódicamente cómo está funcionando el modelo con datos que no ha visto antes. Así podemos comprobar si el modelo está aprendiendo correctamente o si empieza a sobreajustarse (es decir, memorizando los datos de entrenamiento en lugar de generalizar). También sirve para ajustar los hiperparámetros y elegir el momento adecuado para detener el entrenamiento (*early stopping*).

- **Test (15 %)**: Finalmente, una vez que el entrenamiento ha terminado, este conjunto se utiliza para hacer una evaluación definitiva y objetiva del rendimiento del modelo. Es importante que estos datos no se hayan utilizado en ninguna fase anterior, para que la métrica final sea una medida realista de su rendimiento en un escenario real.

Se han elegido estas proporciones para que el modelo tenga suficientes muestras para aprender de forma eficaz, sin dejar de reservar una parte para poder validar correctamente su comportamiento durante el entrenamiento y otra para comprobar su rendimiento final.

### 4.3.2. Entrenamiento del modelo y *data augmentations*

El entrenamiento se estructura en varias épocas, que son ciclos completos en los que el modelo revisa todo el conjunto de datos una vez. Repetir este proceso es importante porque le permite ir aprendiendo poco a poco y reconocer los patrones que hay en los ejemplos. Para mejorar la capacidad de generalización y evitar que el modelo memorice los datos originales, durante el entrenamiento se utilizarán *data augmentations*, un conjunto de transformaciones aleatorias (como rotaciones, traslaciones, cambios de contraste, etc.) que se aplican a las imágenes en tiempo real antes de que sean procesadas por el modelo. Así, cada vez que el modelo vea una imagen, es probable que tenga ligeras variaciones que lo obliguen a reconocer los rasgos más importantes sin depender de los detalles específicos de cada imagen.

Dentro de cada época, el conjunto de entrenamiento se divide en pequeños grupos llamados lotes para que el aprendizaje sea más eficiente. El modelo procesa cada lote y genera una predicción, la compara con la etiqueta real y calcula la diferencia o pérdida. Esta pérdida indica qué tan lejos ha estado de acertar, y se utiliza para ajustar automáticamente los parámetros internos del modelo mediante retropropagación, mejorando su rendimiento a medida que procesa lote tras lote.

### 4.3.3. Validación y *early stopping*

Al finalizar cada época, el modelo cambia a modo evaluación para hacer predicciones sobre un conjunto de datos que nunca ha visto antes, el conjunto de validación. Aquí no se aprende, solo se mide el rendimiento utilizando una métrica, en este caso, el coeficiente de Dice, que mide la precisión con la que el modelo segmenta las áreas correctas en las imágenes. Esto permite evaluar la capacidad del modelo para generalizar más allá de los datos de entrenamiento.

Tras la validación, se compara el coeficiente de Dice obtenido con el mejor resultado registrado hasta ese momento. Si mejora, el modelo se guarda automáticamente como la mejor versión actual. Además, se implementa una estrategia de "parada temprana" (*early stopping*) para evitar entrenar más tiempo del necesario. Si el modelo no mejora tras un número determinado de épocas, el entrenamiento se detiene para evitar que el modelo se sobreajuste a los datos.

### 4.3.4. Ajuste de hiperparámetros y *pruning*

Para automatizar la búsqueda de los mejores hiperparámetros se utiliza Optuna, un *framework* diseñado para ello. Mediante una función objetivo, se realizan múltiples entrenamientos cortos con diferentes configuraciones propuestas por Optuna, evaluando cada configuración en el conjunto de validación con la métrica Dice. Así es más fácil descubrir qué combinaciones funcionan mejor y generalizan bien. Optuna repite este proceso muchas veces. Después de cada entrenamiento corto, obtiene la puntuación y utiliza esa información para escoger nuevos hiperparámetros en los siguientes intentos, explorando automáticamente el espacio de búsqueda. Optuna también cuenta con un mecanismo llamado "poda" (*pruning*) que evalúa el progreso del entrenamiento a mitad del proceso y, si detecta que un intento no está mejorando o es claramente peor que los demás, lo interrumpe antes de tiempo para pasar directamente al siguiente. De este modo, ahorra tiempo y se centra solo en los intentos más prometedores.

### 4.3.5. Entrenamiento final del mejor modelo

Con los hiperparámetros óptimos determinados, se realiza un entrenamiento completo del modelo final usando el conjunto de entrenamiento y aplicando las mismas técnicas que en las

fases anteriores, como *early stopping* y *data augmentation*. Así se obtiene la mejor versión del modelo para su posterior evaluación.

### 4.3.6. Evaluación final en el conjunto de *test*

Cuando finaliza el entrenamiento, el modelo final se evalúa con el conjunto de test, que no se usó en ninguna fase anterior. La evaluación se realiza utilizando la métrica Dice y mide la capacidad del modelo para reconocer patrones en datos que nunca ha visto antes. Así se puede comprobar de forma objetiva qué tan bien funciona el modelo en situaciones reales.

## 4.4. Análisis de resultados

En esta sección se interpretan y analizan las métricas obtenidas, y se revisan las curvas de entrenamiento y validación, que muestran cómo han cambiado la pérdida y el coeficiente de Dice a lo largo del entrenamiento. La curva de pérdida ayuda a ver si el modelo está aprendiendo bien, y al comparar las curvas de entrenamiento y validación se puede saber si el modelo está memorizando los datos (*overfitting*) o si realmente está aprendiendo a generalizar.

Por otra parte, para reducir el espacio de búsqueda con Optuna y acotar el rango de los hiperparámetros a valores más prometedores, se han realizado previamente varias pruebas manuales. Es decir, esto sirve para tener una idea inicial de qué valores funcionan mejor y así acotar los rangos antes de realizar la optimización automática. De este modo, Optuna puede centrarse en las configuraciones más prometedoras, ahorrando tiempo y recursos, evitando que explore aquellas que, por experiencia, se sabe que no van a dar buenos resultados.

### 4.4.1. Pruebas de modelos base con configuración estándar

En esta sección se presentan los resultados obtenidos al entrenar los tres modelos ViT descritos en la sección 4.1, utilizando una configuración común para todos con el objetivo de determinar cuál tiene un mejor rendimiento en el contexto de este TFG.

Para ello, se han elegido los siguientes hiperparámetros común a todos los modelos para su entrenamiento:

- Hiperparámetros de la arquitectura de los modelos:
  - Canales de entrada = 3 (en este caso, las secuencias T1, T2 y FLAIR).
  - Canales de salida = 2 (en este caso, “lesión” y “no lesión”).
- Hiperparámetros del entrenamiento:
  - *Batch size* (número de muestras por lote) = 1. Se ha elegido este número por el elevado coste de memoria GPU que requeriría un tamaño mayor.

- Épocas (número máximo de épocas de entrenamiento) = 200.
- Paciencia (número de épocas sin mejora antes de aplicar *early stopping*) = 20.
- Optimizador (algoritmo que ajusta los pesos del modelo durante el entrenamiento) = AdamW.
- *Learning rate* = 1e-4 (tasa de aprendizaje para el optimizador).
- *Weight decay* (coeficiente de regularización que penaliza grandes valores en los pesos para evitar sobreajuste) = 1e-4.
- *Scheduler* (reduce automáticamente la tasa de aprendizaje cuando la métrica de validación deja de mejorar) = *ReduceLROnPlateau*.
- Función de pérdida (función que calcula el error entre la predicción y la etiqueta real) = DiceCELoss.
- Métrica de validación (medida para evaluar el rendimiento del modelo) = Dice.
- *Sliding window inference* (tamaño de la ventana deslizante usada para inferir) = (96,96,96).

La elección del optimizador AdamW se debe a que es una versión mejorada de Adam, un optimizador muy usado en segmentación. AdamW corrige la regularización por decaimiento de pesos (*weight decay*), que ayuda a evitar que el modelo se sobreajuste y generalice mejor (Yassin, 2024). Además, hace que el entrenamiento sea más estable, muy útil en segmentación médica donde se trabaja con datos complejos y volúmenes 3D de resonancia magnética, como en este caso.

Además, se ha usado un *scheduler*, que sirve para reducir automáticamente la tasa de aprendizaje cuando la métrica de validación deja de mejorar durante varias épocas. Específicamente, el *ReduceLROnPlateau*, ya que a diferencia de otros *schedulers* que bajan el *learning rate* según la época, este lo hace cuando la métrica de validación deja de mejorar. Esto es útil en segmentación médica porque el progreso no siempre es lineal.

Como resultado, SwinUNETR ha tenido el mejor rendimiento. En el conjunto de validación ha alcanzado un valor Dice de 0.6016 tras 49 épocas, y en la evaluación final sobre el conjunto de test un valor medio de Dice de 0.6180 con una desviación típica de 0.2007.

3D-EffitViTCaps también ha obtenido un rendimiento competitivo. Tras 84 épocas de entrenamiento, ha obtenido un valor Dice de 0.5518 en validación. En cambio, en el conjunto de test un Dice medio de 0.5940 ± 0.2056, situándose ligeramente por debajo de SwinUNETR. Además, ha necesitado más épocas en el entrenamiento para obtener buenos resultados, por lo que es menos eficiente que SwinUNETR en ese aspecto.

SegFormer3D ha sido el modelo con el peor rendimiento. A pesar de completar 105 épocas, el valor Dice en validación ha sido tan solo de 0.3271. En el conjunto de test, ha obtenido

| Modelo | Épocas | Dice *test* (media ± std) | Observaciones principales |
|---|---|---|---|
| **SwinUNETR** | 49 | 0.6180 ± 0.2007 | Mejor rendimiento general. Mayor precisión y menor número de épocas. |
| **3D-EffitViTCaps** | 84 | 0.5940 ± 0.2056 | Resultados competitivos, pero ligeramente inferiores. Necesita más épocas. |
| **SegFormer3D** | 105 | 0.4014 ± 0.1867 | Peor rendimiento. Arquitectura eficiente pero demasiado simple para este caso. |

Cuadro 5: Resumen comparativo de los modelos ViT evaluados con configuración estándar.

un Dice medio de 0.4014 ± 0.1867. Estos resultados indican que, aunque la arquitectura de SegFormer está pensada para ser eficiente, es demasiado simple para lograr una segmentación precisa de las lesiones en este caso.

Es por esto que se ha elegido SwinUNETR como modelo principal.

En las Figuras 24, 25 y 26 se pueden observar las curvas de entrenamiento y validación de las arquitecturas SwinUNETR, 3D-EffiViTCaps y SegFormer3D, respectivamente, que muestran cómo han cambiado la pérdida (*loss*) y el coeficiente de Dice a lo largo del entrenamiento. A medida que la pérdida baja (es decir, el modelo aprende a segmentar mejor), el valor de Dice sube, indicando una mejora en la precisión de la segmentación. En las últimas épocas la pérdida sigue bajando muy lentamente pero el Dice no sube o es incluso más bajo, así que el entrenamiento se para (*early stopping*) para que no se sobreajuste a los datos.

### 4.4.2. Pruebas con distintas funciones de pérdida

Como se ha explicado al principio de este capítulo, se han realizado algunas pruebas manuales previas a la optimización con Optuna para reducir el espacio de búsqueda y así pueda centrarse en las configuraciones más prometedoras, descartando las que se sabe que no van a proporcionar un buen resultado.

Por otra parte, para evaluar el impacto de diferentes funciones de pérdida en el rendimiento del modelo, se ha entrenado SwinUNETR utilizando varias configuraciones, manteniendo iguales el resto de hiperparámetros. Para ello, se han utilizado funciones de pérdida especialmente para segmentación médica de MONAI. La función base utilizada es *DiceCELoss*, una combinación de Dice y entropía cruzada (CE). Por un lado, la pérdida de Dice mide la superposición entre las regiones segmentadas por el modelo y las regiones reales (*ground truth*), y por otro lado, la entropía cruzada actúa a nivel de vóxel, penalizando las predicciones inco-

| Función de pérdida | Dice *test* (media ± std) | Descripción |
|---|---|---|
| ***DiceCELoss*** | 0.6180 ± 0.2007 | Combinación de Dice y entropía cruzada (CE). Equilibra sensibilidad y precisión. |
| ***TverskyLoss* ($\alpha$=0.7, $\beta$=0.3)** | 0.6101 ± 0.1984 | Penaliza más los falsos negativos (más sensibilidad). |
| ***TverskyLoss* ($\alpha$=0.3, $\beta$=0.7)** | 0.6069 ± 0.1988 | Penaliza más los falsos positivos (más precisión). |
| ***DiceFocalLoss* ($\lambda_{dice}$=1.5, $\lambda_{focal}$=0.5)** | 0.6158 ± 0.2068 | Se centra más en la segmentación general de la lesión. |
| ***DiceFocalLoss* ($\lambda_{dice}$=0.5, $\lambda_{focal}$=1.5)** | 0.6063 ± 0.2008 | Se centra más en las regiones más difíciles. |

Cuadro 6: Resumen comparativo de funciones de pérdida utilizadas con SwinUNETR y su rendimiento en el conjunto de test.

rrectas en cada punto. Con esta función se ha obtenido el mejor resultado, con un valor medio de Dice en el conjunto de test de 0.6180.

También se han probado variantes de la función *TverskyLoss*, que permite ajustar el peso de los falsos positivos y falsos negativos mediante los parámetros $\alpha$ y $\beta$. En la configuración con $\alpha$=0.7 y $\beta$=0.3, que penaliza más los falsos negativos y, por tanto, favorece la sensibilidad, el modelo ha alcanzado un Dice medio de 0.6101. En cambio, con $\alpha$=0.3 y $\beta$=0.7, que penaliza más los falsos positivos, el valor de Dice ha sido de 0.6069. Ambos resultados han sido similares al obtenido con *DiceCELoss*, aunque ligeramente inferiores.

Por otro lado, se ha evaluado la función *DiceFocalLoss*, que combina la pérdida de Dice con la Focal Loss, usada para mejorar el aprendizaje en regiones difíciles. Al dar mayor peso a la parte Dice ($\lambda_{dice}$=1.5), el modelo ha alcanzado un Dice en test de 0.6158, mientras que al dar mayor peso a la parte Focal ($\lambda_{focal}$=1.5), el resultado ha sido ligeramente inferior, con un Dice de 0.6063. Por lo que la variante con más peso en Dice también ha obtenido un rendimiento competitivo respecto a *DiceCELoss*.

Los resultados indican que todas las funciones de pérdida evaluadas han sido capaces de entrenar modelos con un buen rendimiento, siendo *DiceCELoss* y la versión de *DiceFocalLoss* con mayor peso en la parte Dice las que han obtenido los mejores resultados.

### 4.4.3. Pruebas con *data augmentations*

En esta sección se explican las pruebas realizadas al aplicar *data augmentations*, que consisten en aplicar transformaciones aleatorias sobre las imágenes durante el entrenamiento. De

esta forma, mejoran la capacidad de generalización del modelo, reducen el riesgo de sobreajuste y simulan variaciones reales en las imágenes, por lo que el modelo aprende a ser más robusto frente a condiciones variables que pueden encontrarse en la práctica.

Primero, se han realizado algunas augmentations realistas que alteran la geometría espacial para evitar sobreajuste, además de cambios en la intensidad para simular la variabilidad del escáner de las RM. Para ello, se han utilizado las funciones *RandFlipd*, *RandScaleIntensityd* y *RandShiftIntensityd* de MONAI, todas con con probabilidad 0.1 (10 %).

- ***RandFlipd***: realiza "volteos" aleatorios de las imágenes y sus máscaras a lo largo de los ejes espaciales para ayudar al modelo a aprender independientemente de la orientación del paciente en el escáner. Se ha aplicado a los tres ejes x, y, z.
- ***RandScaleIntensityd***: modifica aleatoriamente el contraste de la imagen multiplicando sus valores de intensidad por un factor. Permite simular diferencias de contraste entre distintos escáneres o protocolos. Se ha elegido un factor de 0.1 para que no sea muy agresivo.
- ***RandShiftIntensityd***: Desplaza los valores de intensidad sumando un valor aleatorio, simulando cambios en el brillo de la imagen o en la calibración del escáner. Se ha elegido un *offset* (suma) de 0.1 para que no sea muy agresivo.

En las Figuras 31, 32 y 33 se pueden observar ejemplos del efecto de estas transformaciones.

Como resultado, se ha obtenido el mejor rendimiento hasta el momento, con un Dice en test de 0.6252 ± 0.1933.

Sin embargo, si se aumentan las probabilidades para que ocurran con más frecuencia, el rendimiento del modelo baja considerablemente. En este caso, se ha probado con una probabilidad de 0.5 (50 %) y el valor de Dice en el conjunto de *test* ha bajado a 0.4992 ± 0.2076.

| **Transformaciones** | **Probabilidad** | **Dice (*test*)** |
|---|---|---|
| *RandFlipd*, *RandScaleIntensityd*, *RandShiftIntensityd* | 0.1 (10 %) | **0.6252 ± 0.1933** |
| *RandFlipd*, *RandScaleIntensityd*, *RandShiftIntensityd* | 0.5 (50 %) | 0.4992 ± 0.2076 |

Cuadro 7: Resumen de resultados al aplicar *data augmentations* con distintas probabilidades.

### 4.4.4. Ajuste de hiperparámetros con Optuna

Para ajustar y optimizar los hiperparámetros automáticamente se ha usado Optuna, que busca la mejor combinación de hiperparámetros que maximice la métrica Dice. Para ello, se ha considerado el espacio de búsqueda reducido con las pruebas manuales previas, y se han aplicado además las técnicas de *data augmentations*. Específicamente, se ha ajustado el *weight*

*decay*, *learning rate* y la función de pérdida. El resto de hiperparámetros, como el tamaño del batch, se mantiene en 1 por su elevado coste de memoria GPU, o el optimizador, que también se mantiene igual, ya que se sabe que AdamW es mejor que Adam cuando se entrena con datos complejos. Los rangos considerados para el *weight decay* y *learning rate* son [1e-4, 1e-3], y además las dos funciones de pérdida con mejor resultados en las pruebas manuales (*DiceCELoss* y *DiceFocalLoss* con mayor peso en la parte Dice). Un valor ligeramente alto de *weight decay* puede ayudar a prevenir el sobreajuste en modelos grandes y complejos, mientras que modelos más pequeños necesitan menos regularización, por lo que se ha elegido un rango intermedio. En cambio, un valor alto de *learning rate* puede provocar que el entrenamiento sea inestable y uno muy bajo puede hacer que aprenda muy lentamente o no aprenda nada. Es por esto que se ha elegido también un rango intermedio.

Por otra parte, para evitar que los *trials* (intentos) sean demasiado largos se ha reducido el número de épocas máximo a 30 con una paciencia de 5 épocas. Así se pueden realizar más *trials*, probando nuevas configuraciones para obtener la más óptima.

Además, se ha usado TPE (*Tree-structured Parzen Estimator*), que es el algoritmo de muestreo que sugiere nuevos conjuntos de hiperparámetros, con una semilla fija para que los experimentos sean reproducibles. Este sampler es capaz de aprender qué rangos de hiperparámetros funcionan mejor a partir de los resultados anteriores, por lo que es más efectivo que probar configuraciones aleatorias. Por último, se ha usado el *pruner* (poda) *MedianPruner*, que permite detener automáticamente los *trials* que van mal antes de que terminen, para ahorrar tiempo. Este pruner compara el rendimiento actual de un *trial* con la mediana de resultados de todos los *trials* anteriores en el mismo paso (por ejemplo, la misma época). Si va peor que la mediana, el trial se puede detener antes de terminar. Antes de empezar *pruning*, espera a tener un número mínimo de *trials* (5 en este caso), es decir, deja completar esos *trials* para que el estudio tenga una base suficiente para calcular la mediana. Además, dentro de cada trial, no se hace *pruning* hasta que haya pasado un número mínimo de épocas (5 en este caso), así evita podar *trials* que tienen un inicio lento pero podrían mejorar.

El estudio se ha limitado a 100 *trials* y 10 horas, es decir, ha realizado todos los *trials* que ha podido en 10 horas, con límite 100. Como resultado, ha habido un total de 35 *trials*, de los cuales se han podado 27. Los mejores hiperparámetros obtenidos son: *lr* = 0.0001432249371823026, *weight decay* = 0.00014321698289111514, *loss* = *DiceFocalLoss*. El entrenamiento final del modelo con estos hiperparámetros proporcionó un Dice en el conjunto de *test* de 0.6119 ± 0.1786.

Como se han podado muchos *trials*, se ha hecho un nuevo estudio en el que se ha aumentado el número máximo de épocas a 40 con una paciencia de 10, y el número mínimo de épocas antes de que se pueda podar a 10. Además, como se han tenido que probar muchas configuraciones de hiperparámetros, solo se consideran en este estudio el *learning rate* y el *weight decay*, y se toma la mejor función de pérdida según las pruebas manuales (*DiceCELoss*).

En este caso, solo han habido 11 *trials*, donde el mejor tenía los siguientes hiperparámetros: *lr* = 0.0005395030966670229, *weight decay* = 0.00039687933304443713. El entrenamiento final del modelo con estos hiperparámetros proporcionó un Dice en *test* de 0.6086 ± 0.1880. Con esto se concluye que se necesitan muchos más *trials* para encontrar los mejores hiperparámetros, pero requeriría una gran cantidad de tiempo.

| **Estudio** | ***Trials*** | ***Learning rate*** | ***Weight decay*** | ***Loss function*** | **Dice (*test*)** |
|---|---|---|---|---|---|
| 1 | 35 | 0.00014322 | 0.00014321 | *DiceFocalLoss* | 0.6119 ± 0.1786 |
| 2 | 11 | 0.00053950 | 0.00039688 | *DiceCELoss* | 0.6086 ± 0.1880 |

Cuadro 8: Resumen de los dos estudios de optimización de hiperparámetros con Optuna. Conforme se realizan más *trials*, *learning rate* y *weight decay* se acercan más a 1e-4, considerado en un primer momento con un buen resultado.

### 4.4.5. Selección del mejor modelo

Tras todas las pruebas realizadas, se ha elegido el modelo con el mejor rendimiento, que ha sido el obtenido tras aplicar las *data augmentations* explicadas en la sección 4.4.3 con una probabilidad del 10 %, con la arquitectua SwinUNETR y los siguientes hiperparámetros:

- *Batch size* = 1.
- Épocas = 200.
- Paciencia = 20.
- Optimizador = AdamW.
- *Learning rate* = 1e-4.
- *Weight decay* = 1e-4.
- *Scheduler* = *ReduceLROnPlateau*.
- Función de pérdida = *DiceCELoss*.
- Métrica de validación = Dice.
- *Sliding window inference* = (96,96,96). Se ha elegido este tamaño porque la arquitectura del modelo requiere que sean múltiplos de 32, y con un tamaño de (64,64,64) la ventana es demasiado pequeña para capturar suficiente contexto espacial, obteniendo un Dice más bajo.

Este modelo obtuvo un Dice en *test* de **0.6252 ± 0.1933**.

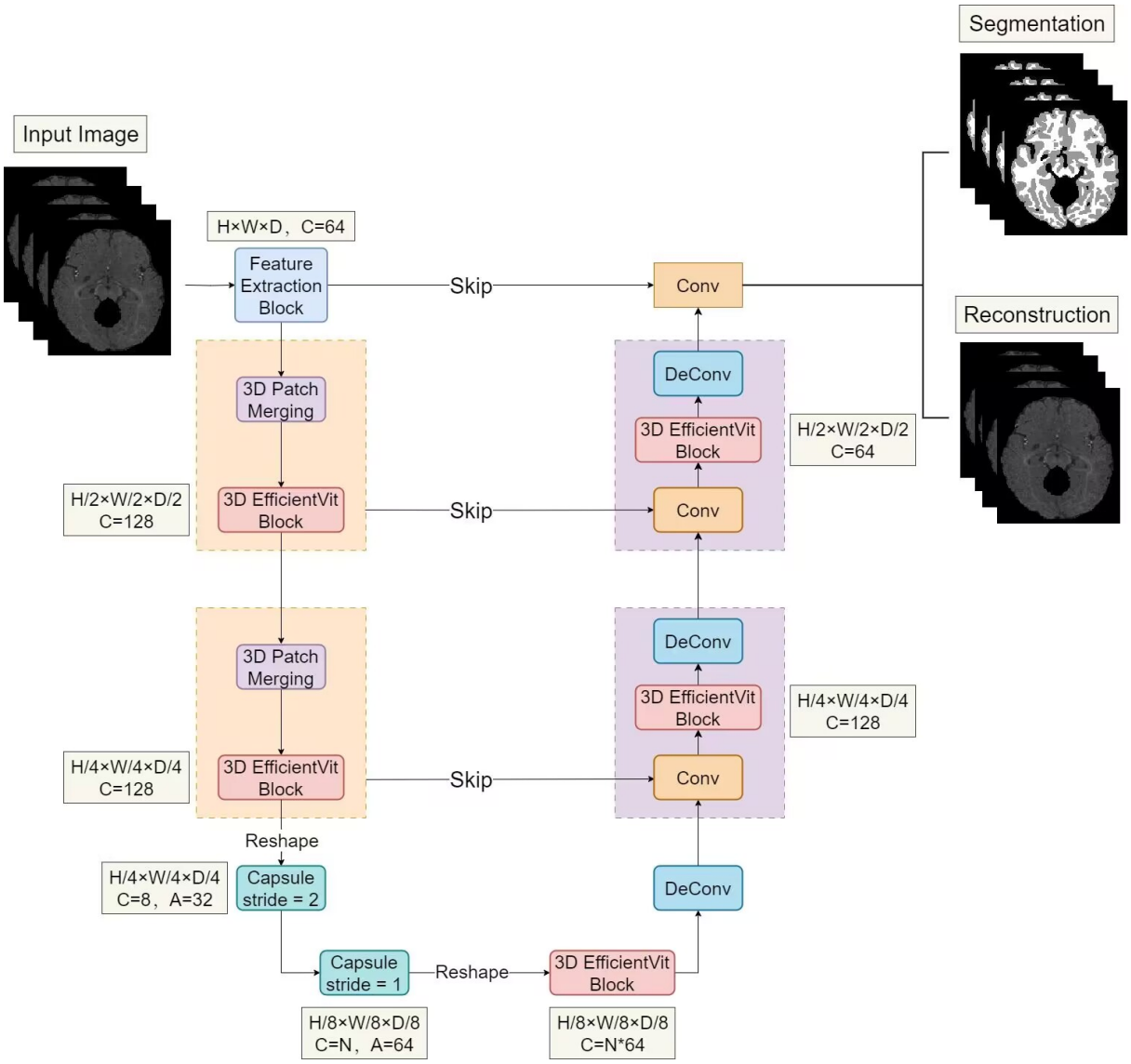


Figura 21: Arquitectura 3D-EffitViTCaps. El modelo recibe como entrada una imagen 3D que pasa por un bloque para extraer características y aumentar canales. Luego, se procesan con bloques 3D EfficientViT, que capturan detalles locales y globales, y bloques cápsula ubicados en las partes profundas del *encoder* y el centro para entender relaciones entre partes. El encoder también usa bloques *3D Patch Merging* para reducir resolución y aumentar canales, resumiendo la información. En el decoder, se combinan bloques EfficientViT con convoluciones 3D tradicionales para crear la segmentación.

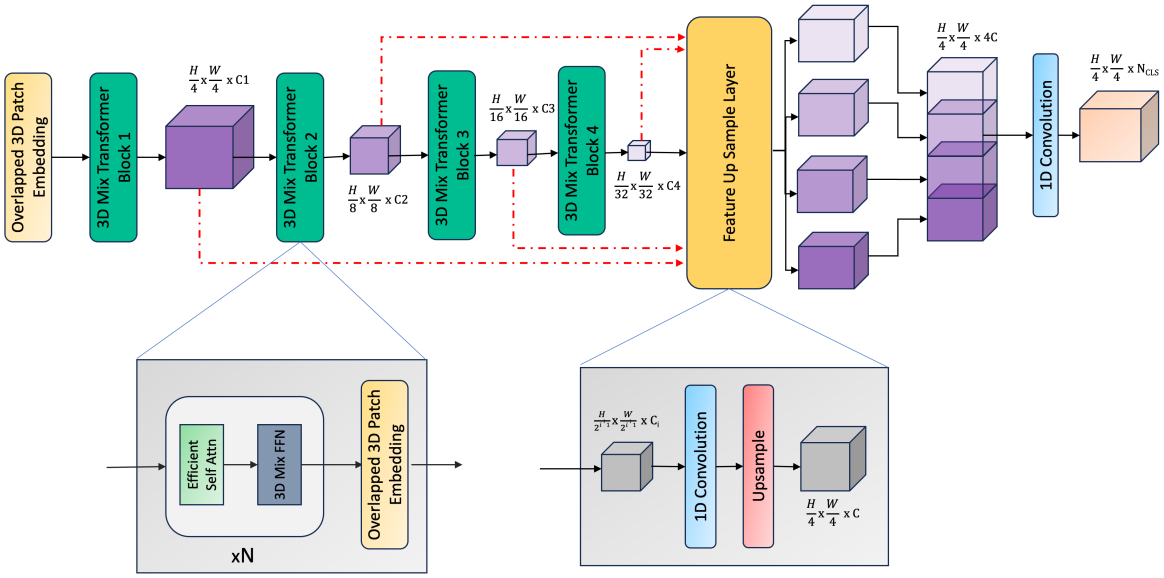


Figura 22: Arquitectura SegFormer3D. El modelo recibe como entrada un volumen 3D con dimensiones de profundidad, canales, altura y ancho. Primero, usa un Transformer en cuatro etapas para extraer características importantes a diferentes escalas dentro del volumen. Después, un decodificador sencillo, formado solo por capas MLP, aumenta la resolución y combina la información local y global que obtuvo el Transformer para crear la segmentación final.

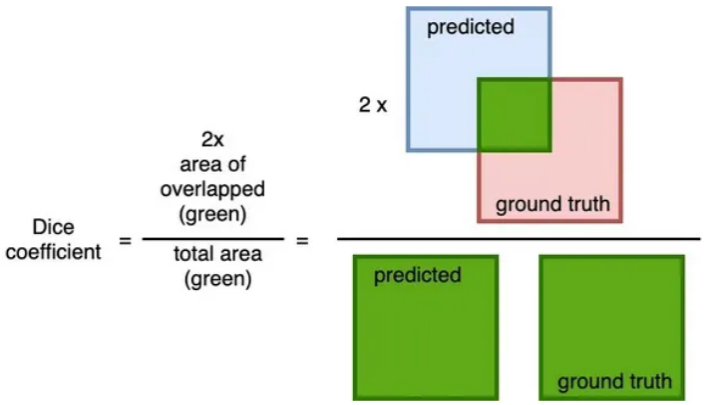


Figura 23: Coeficiente de Dice.

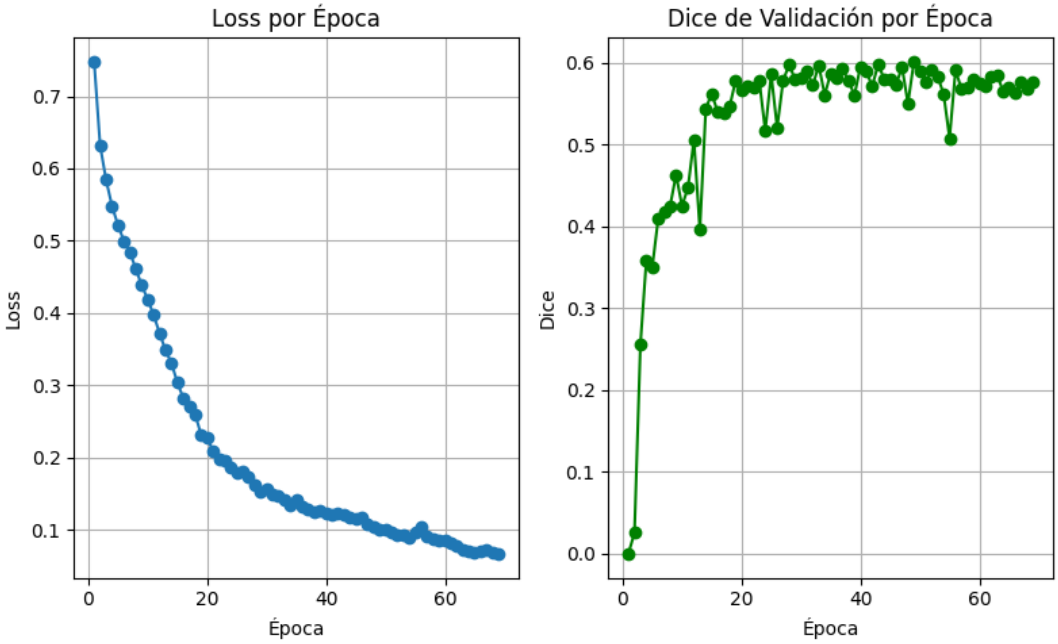


Figura 24: Curvas de entrenamiento y validación de la arquitectura SwinUNETR con configuración estándar.

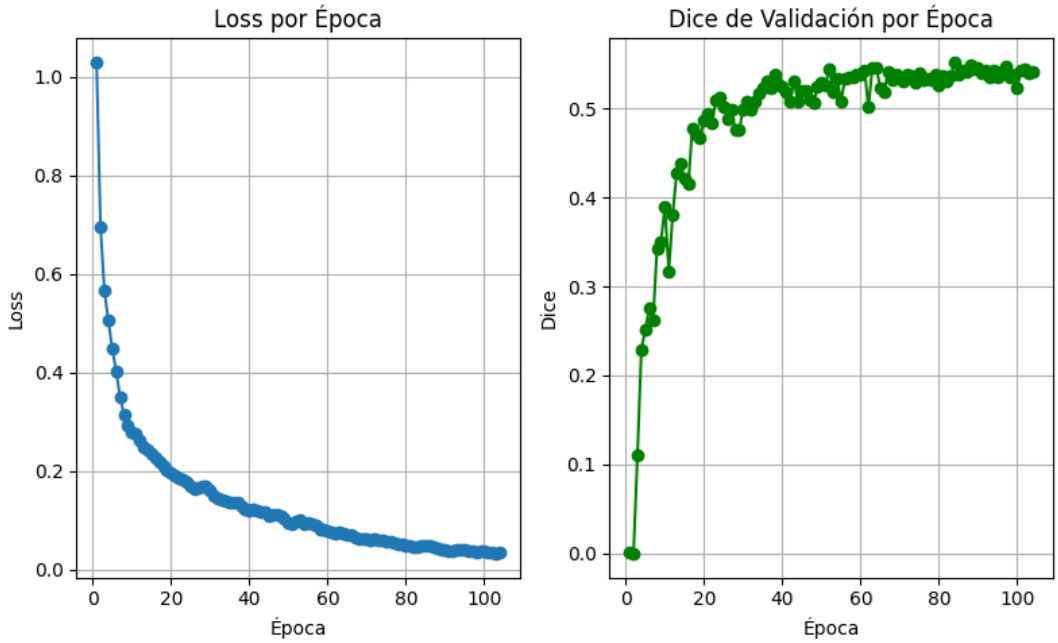


Figura 25: Curvas de entrenamiento y validación de la arquitectura 3D-EffiViTCaps con configuración estándar.

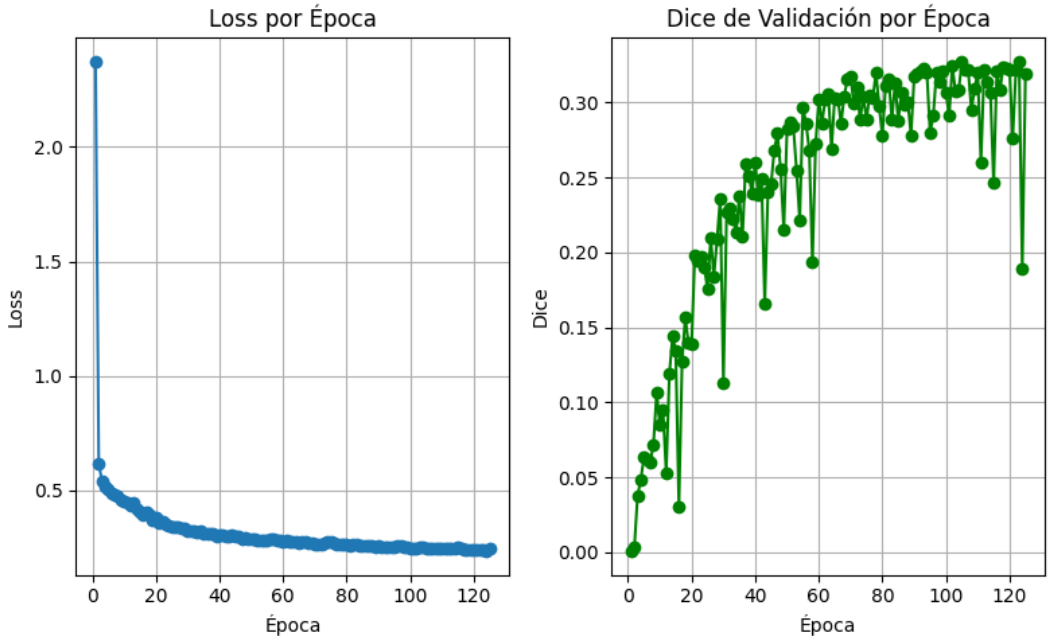


Figura 26: Curvas de entrenamiento y validación de la arquitectura SegFormer3D con configuración estándar.

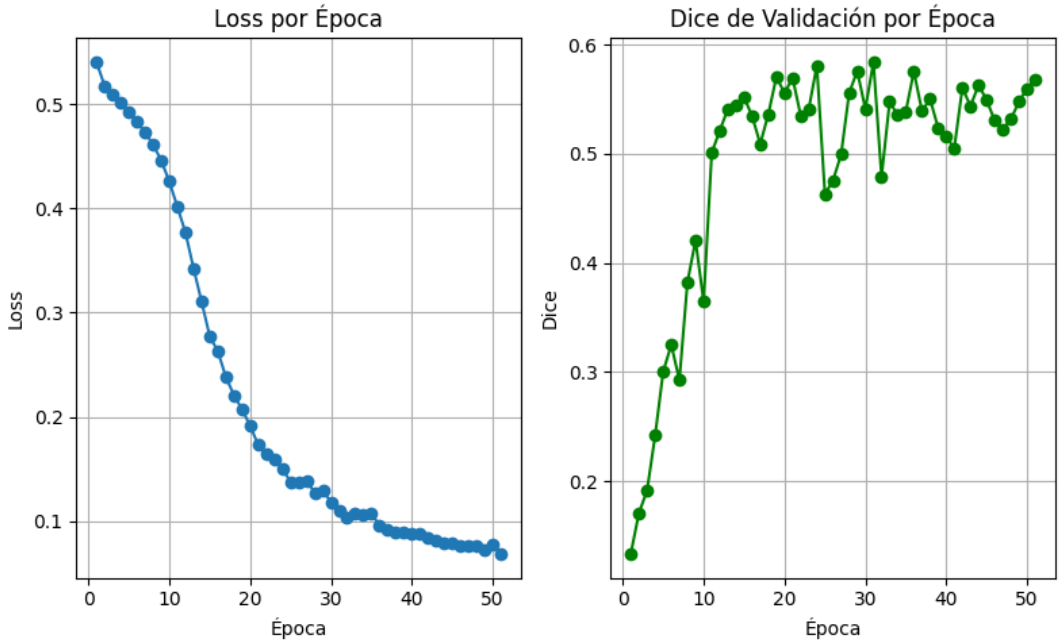


Figura 27: Curvas de entrenamiento y validación de la arquitectura SwinUNETR con la función de pérdida *TverskyLoss* ($\alpha$=0.7, $\beta$=0.3).

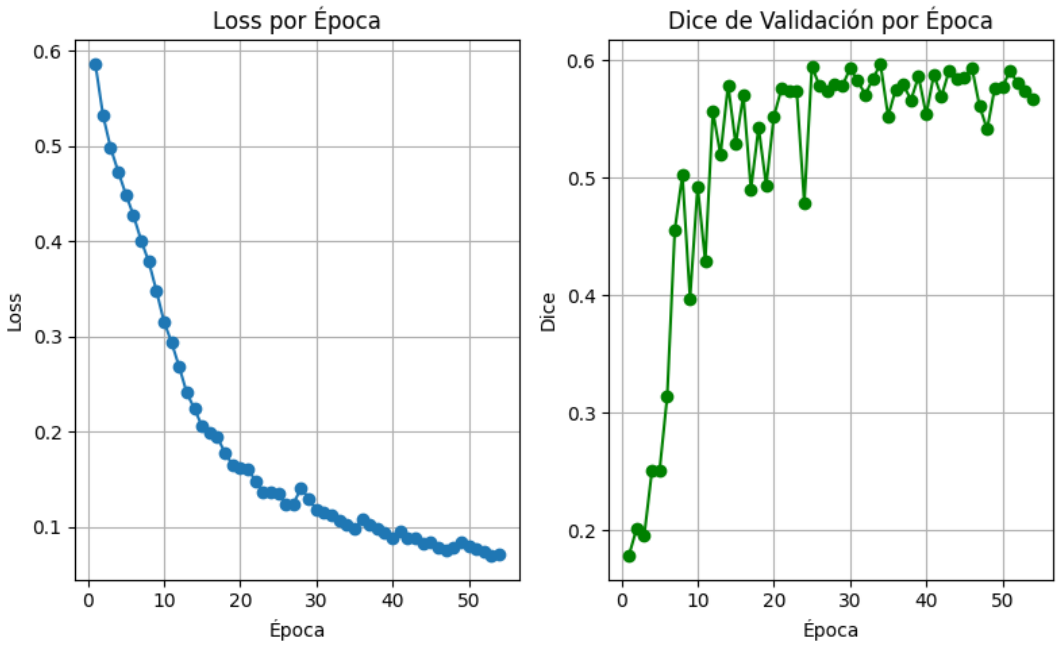


Figura 28: Curvas de entrenamiento y validación de la arquitectura SwinUNETR con la función de pérdida *TverskyLoss* ($\alpha$=0.3, $\beta$=0.7).

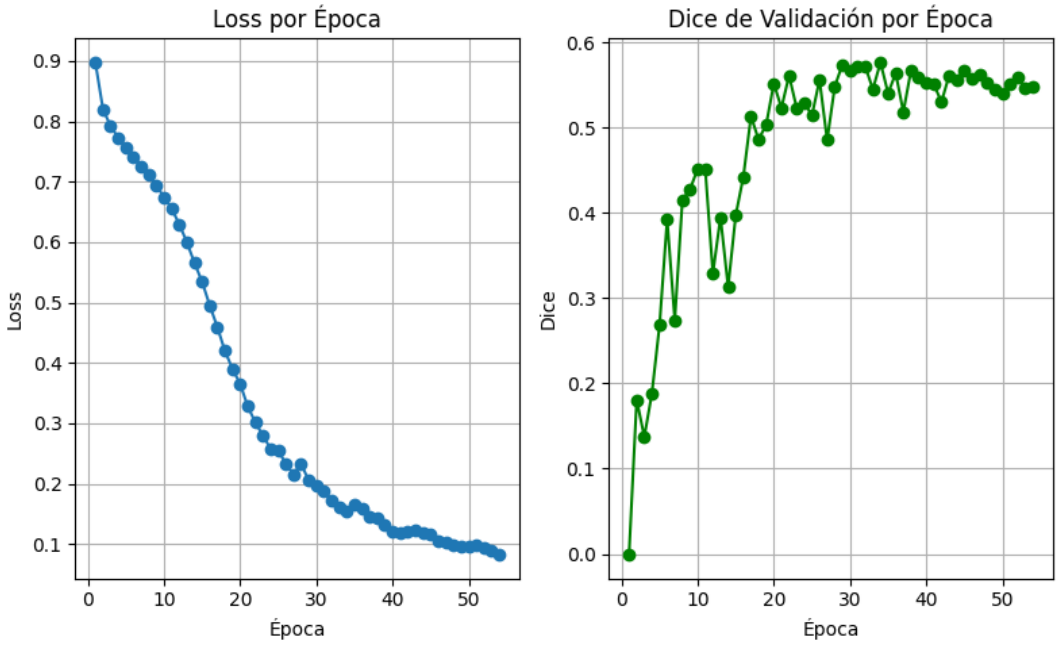


Figura 29: Curvas de entrenamiento y validación de la arquitectura SwinUNETR con la función de pérdida *DiceFocalLoss* ($\lambda_{dice}$=1.5, $\lambda_{focal}$=0.5).

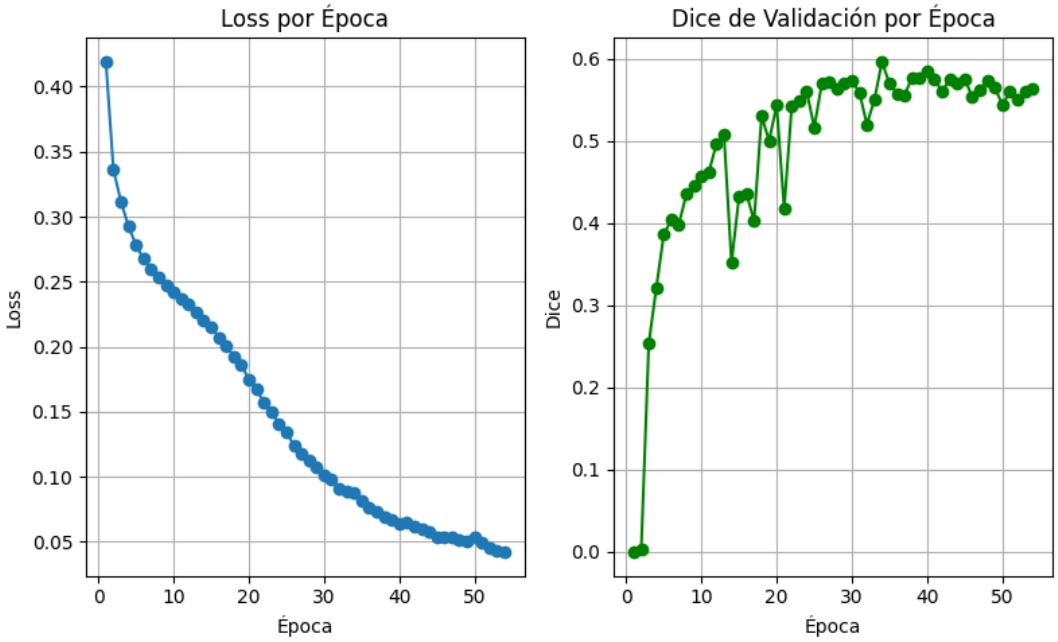


Figura 30: Curvas de entrenamiento y validación de la arquitectura SwinUNETR con la función de pérdida *DiceFocalLoss* ($\lambda_{dice}$=0.5, $\lambda_{focal}$=1.5).

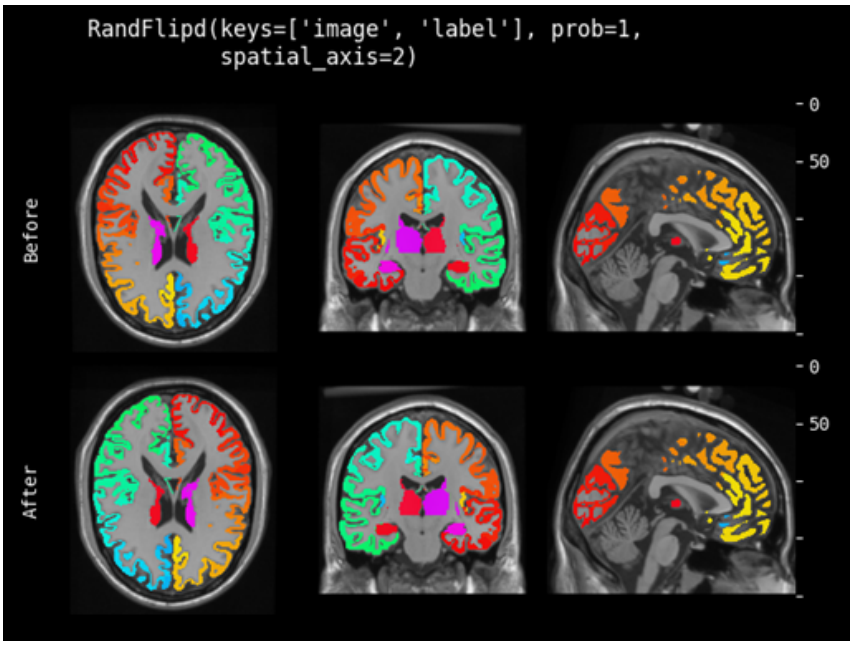


Figura 31: Ejemplo de transformación con *RandFlipd*.

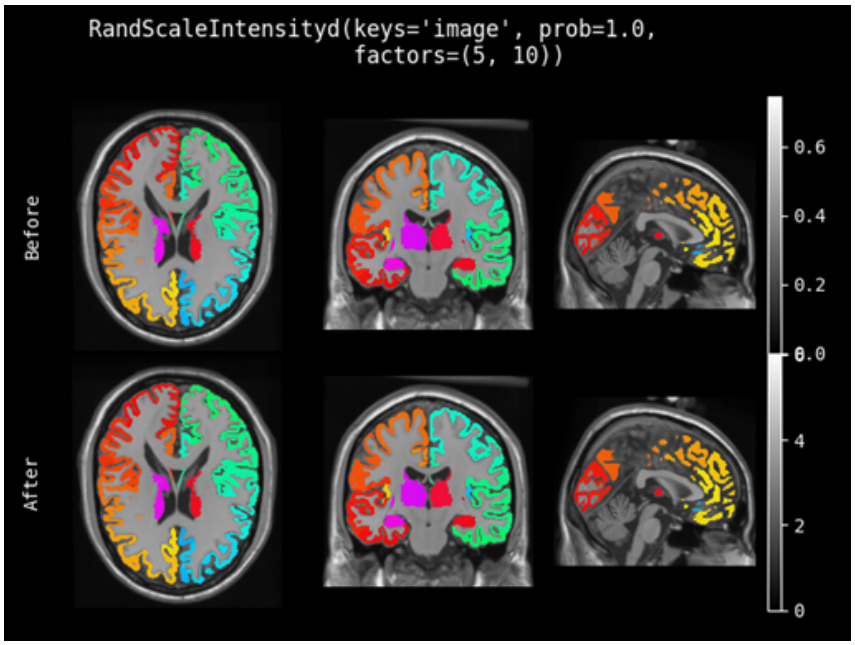


Figura 32: Ejemplo de transformación con *RandScaleIntensityd*.

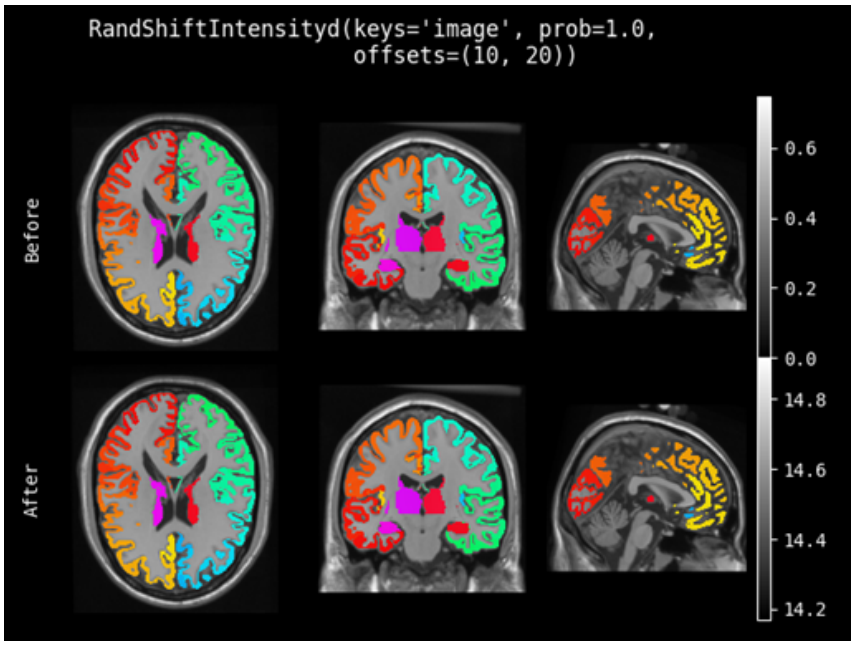


Figura 33: Ejemplo de transformación con *RandShiftIntensityd*.

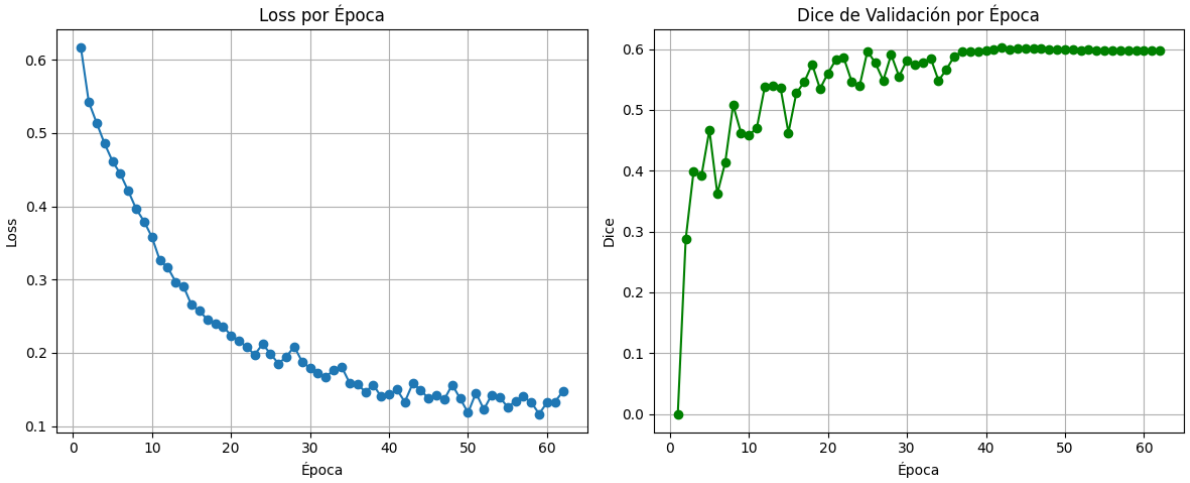


Figura 34: Curvas de entrenamiento y validación de la arquitectura SwinUNETR aplicando *data augmentation* con *RandFlipd*, *RandScaleIntensityd* y *RandShiftIntensityd*, todas con una probabilidad del 10 %.

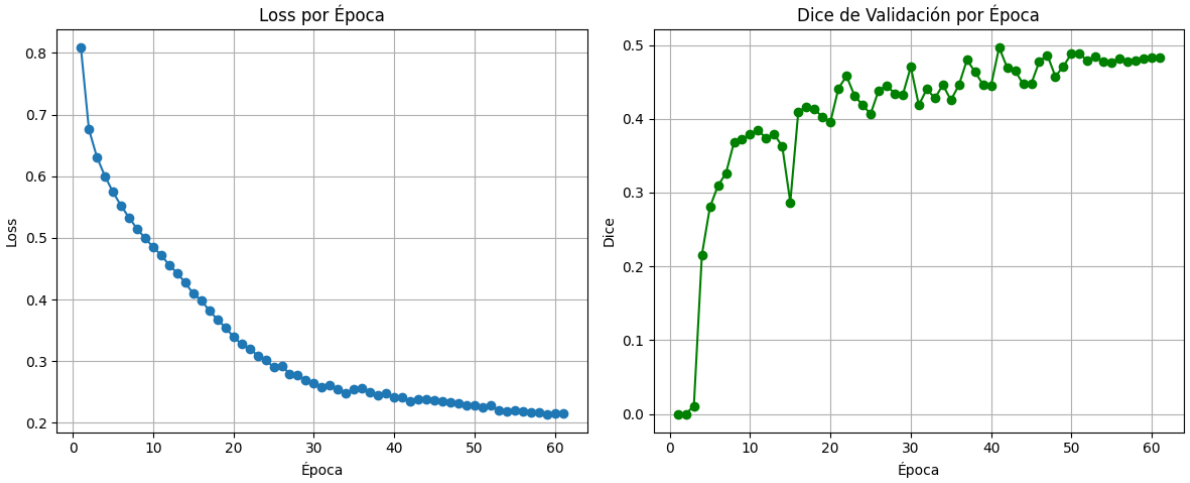


Figura 35: Curvas de entrenamiento y validación de la arquitectura SwinUNETR aplicando *data augmentation* con *RandFlipd*, *RandScaleIntensityd* y *RandShiftIntensityd*, todas con una probabilidad del 50 %.

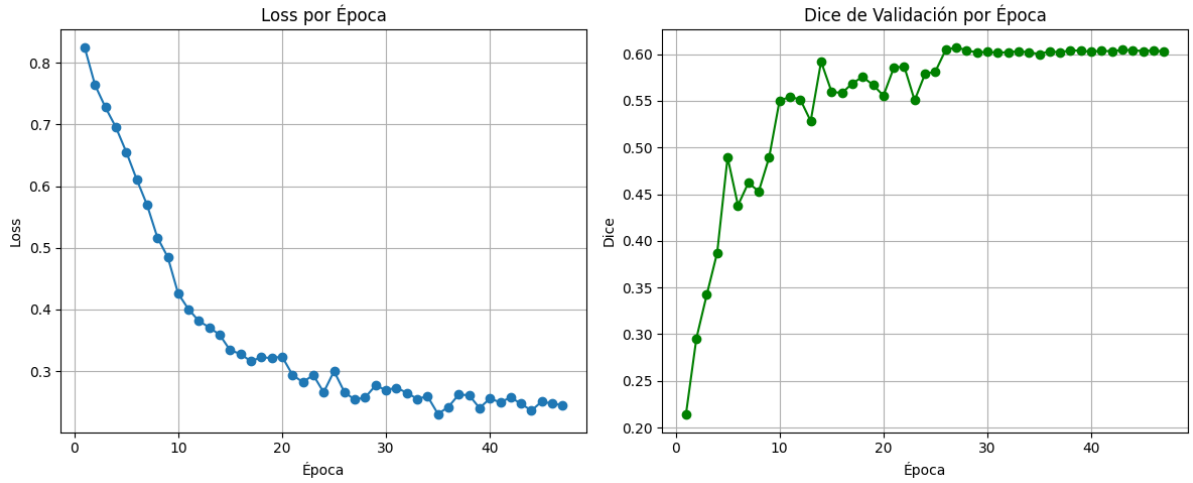


Figura 36: Curvas de entrenamiento y validación de la arquitectura SwinUNETR después del primer estudio con Optuna (35 *trials*).

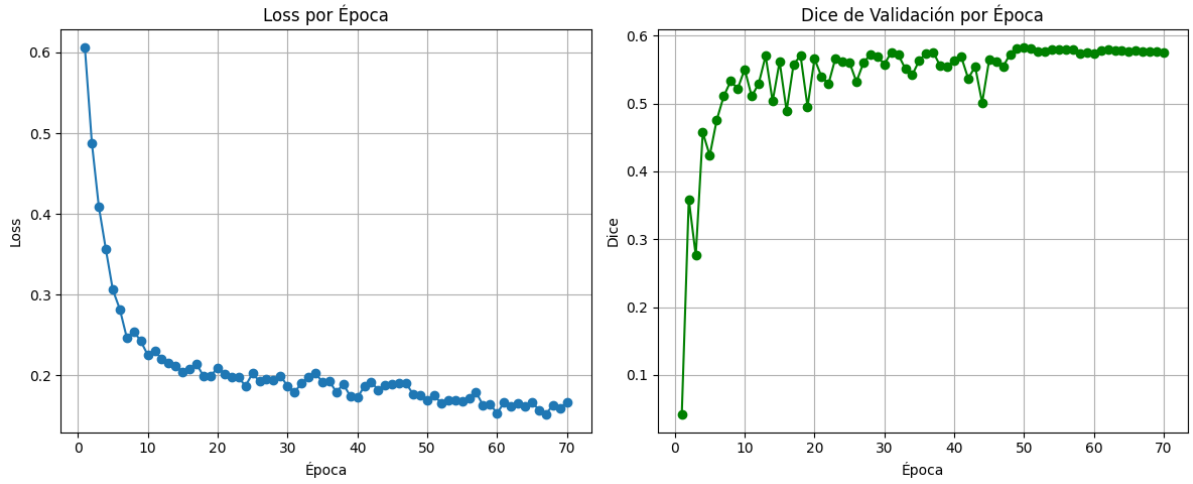


Figura 37: Curvas de entrenamiento y validación de la arquitectura SwinUNETR después del primer estudio con Optuna (11 *trials*).

# 5
# Aplicación Web

## 5.1. Arquitectura

La aplicación está construida con una arquitectura cliente-servidor, donde el cliente (la interfaz web) interactúa con el servidor que ejecuta el procesamiento de las imágenes y realiza la inferencia utilizando el modelo entrenado.

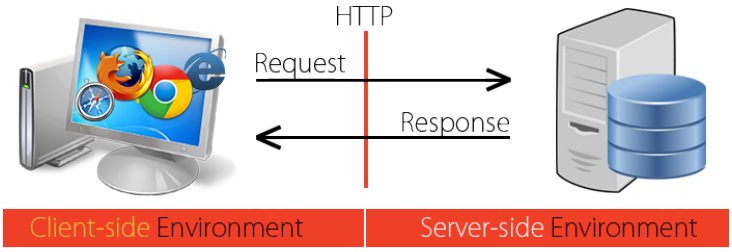


Figura 38: Arquitectura cliente-servidor.

Esta separación permite que cada parte se ocupe de distintas tareas: el cliente gestiona la parte visual y la interacción con el usuario, mientras que el servidor se encarga de la lógica y el procesamiento. Esto también hace que sea más fácil actualizar el servidor sin tener que modificar la interfaz, y permite que varios usuarios accedan al sistema al mismo tiempo.

La comunicación entre cliente y servidor se lleva a cabo mediante una API (*Application Programming Interface*), que es un conjunto de reglas que define cómo los distintos componentes del sistema intercambian datos entre sí. En este TFG se ha implementado una API RESTful con FastAPI, que sigue los principios REST (*Representational State Transfer*). Esto significa que los recursos del servidor se organizan en direcciones específicas llamadas *endpoints*, que son URLs a las que el cliente puede enviar solicitudes usando los métodos estándar de HTTP para realizar diferentes acciones, como subir imágenes o visualizar datos. Cada *endpoint* representa una función concreta que el servidor pone a disposición para que el cliente pueda trabajar con los datos. Además, REST es sin estado (*stateless*), lo que quiere decir que cada vez que el cliente hace una petición al servidor, debe incluir toda la información necesaria para que el servidor la entienda y pueda responder correctamente, sin que el servidor tenga que recordar nada de peticiones anteriores.

Para comunicarse con estos *endpoints*, el cliente utiliza métodos del protocolo HTTP, que son comandos que indican al servidor qué acción debe realizar con la información recibida. Específicamente, se utilizarán los métodos GET y POST.

A continuación se describen los *endpoints* usados:

- **GET/**: Este endpoint representa la interfaz web del sistema. Entrega la página principal de la aplicación cuando el usuario accede a la raíz del servidor. Esta página contiene un título y una breve descripción del servicio, un formulario para cargar las imágenes T1, T2 y FLAIR con un botón "Segmentar" para enviar el formulario y lanzar así la segmentación, y finalmente un botón "Empezar de nuevo".

  Antes de enviar el formulario, verifica que los nombres de los archivos contienen correctamente "T1", "T2" y "FLAIR", y que tienen la extensión .nii.gz. Si el tipo de archivo es incorrecto, se muestra un mensaje de error al usuario.

  Tras elegir los archivos, cuando el usuario hace clic en el botón "Segmentar", muestra el texto "Segmentando..." y una rueda de carga (*spinner*). Además, se desactiva el botón para evitar múltiples envíos.

  Por otra parte, usa *XMLHttpRequest* (un objeto propio de JavaScript que permite a las páginas web comunicarse con servidores web a través de peticiones HTTP sin necesidad de recargar la página completa) para enviar los archivos al *endpoint* POST/segment/, actualizar la página para mostrar el resultado segmentado una vez que el servidor responda con la URL de la visualización, ocultar el formulario y mostrar el botón "Empezar de nuevo".

  Cuando el usuario hace clic en el botón "Empezar de nuevo", vuelve a mostrar la página como era inicialmente, mostrando otra vez el formulario.

- **POST/segment/**: Este endpoint recibe las imágenes que envía el cliente mediante el formulario. Para ello sigue los siguientes pasos:

  1. Crea una carpeta temporal para almacenar los archivos en disco mientras se procesan.
  2. Aplica el preprocesamiento descrito en la sección 3.2.
  3. Realiza la inferencia con el modelo entrenado, obteniendo primero un mapa de probabilidades de segmentación para cada vóxel. A continuación, convierte ese mapa en etiquetas discretas (0 o 1 por vóxel), donde 1 indica lesión y 0 tejido sano, obteniendo de esta forma la máscara segmentada. Para hacer la inferencia se ha usado la función *sliding window inference* de MONAI, que permite procesar imágenes volumétricas grandes dividiéndolas en ventanas más pequeñas, pasando cada ventana

a través del modelo de segmentación, y después recombinando las predicciones de todas esas ventanas para obtener la segmentación completa. De esta forma, ayuda a procesar imágenes grandes sin sobrecargar la memoria GPU.

4. Realiza el postprocesado, donde redimensiona la máscara segmentada para que coincida con la dimensiones de la imagen T1 preprocesada. Esto se hace para asegurar que la máscara se pueda superponer correctamente.
5. Crea una visualización de la segmentación superpuesta a la imagen T1 preprocesada. Se ha elegido T1 porque es la modalidad de resonancia magnética que habitualmente sirve como referencia anatómica principal, como se ha explicado en la sección 3.2.1.
6. Elimina los archivos temporales y devuelve la URL para visualizar la imagen segmentada.

- **GET/visualization/filename**: Este endpoint se utiliza para obtener la imagen segmentada generada por el servidor. Cuando el cliente accede a esta URL con el nombre del fichero (*filename*), el servidor busca la imagen en su directorio de resultados y la envía para que pueda ser visualizada. Si el archivo no existe, se responde con un error 404 indicando que no se ha encontrado.

## 5.2. Flujo de funcionamiento

El funcionamiento de la aplicación se puede resumir en los siguientes pasos:

1. El usuario accede a la aplicación desde el navegador (figura 39). El cliente realiza una petición GET que devuelve la interfaz web.
2. A través de un formulario, el usuario selecciona las imágenes a segmentar (figura 40) y se comprueba que el nombre y formato de cada archivo sean correctos. Si el tipo de archivo es incorrecto, se muestra un mensaje de error al usuario (figura 41).
3. Al pulsar el botón “Segmentar”, se envían las imágenes al servidor mediante la petición POST/segment/.
4. El servidor recibe las imágenes, aplica el preprocesamiento y realiza la inferencia con el modelo ViT entrenado. Mientras tanto, en el lado del cliente, se muestra al usuario una rueda de carga para indicar que se está realizando la segmentación (figura 42).
5. Una vez obtenido el resultado de la segmentación, se realiza su postprocesado y se devuelve al cliente para que el usuario pueda verla.


Figura 39: Formulario para subir las imágenes.

6. El cliente realiza la petición GET/visualization/filename. El usuario puede visualizar el resultado de la segmentación (figura 43), y si desea volver a segmentar, pulsa “Empezar de nuevo” y se muestra el formulario nuevamente.

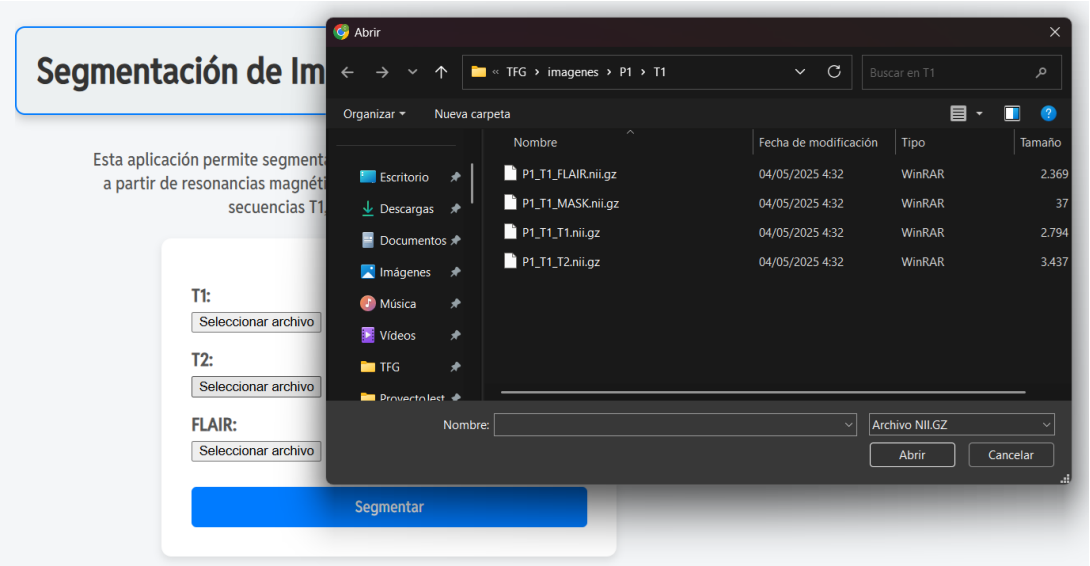


Figura 40: Selección de archivos.

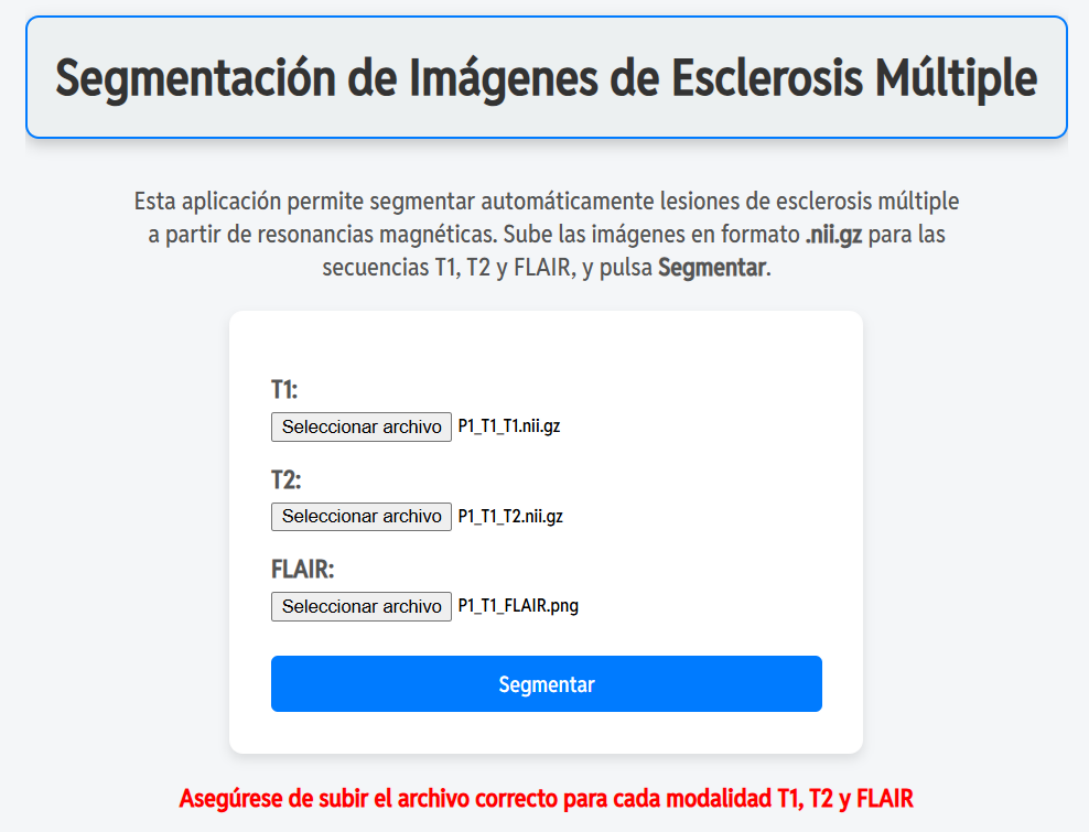


Figura 41: Mensaje de error si uno de los archivos no es correcto.


Figura 42: Procesamiento de la segmentación.

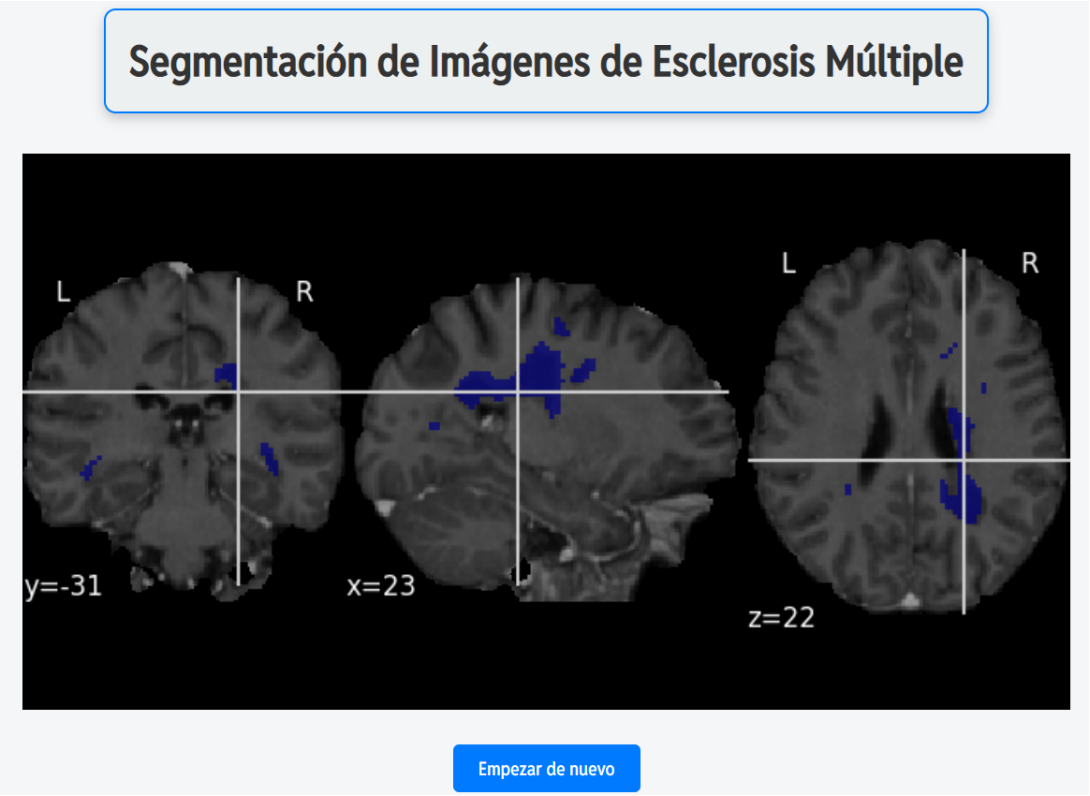


Figura 43: Visualización del resultado de la segmentación.

# 6

# Conclusiones y Líneas Futuras

## 6.1. Conclusiones

Aunque han existido limitaciones como tener un dataset más amplio o una GPU más potente, el proyecto ha cumplido los objetivos principales: se ha entrenado un modelo Vision Transformer con datos reales de esclerosis múltiple, se ha construido un *pipeline* de preprocesamiento y se ha implementado una aplicación web sencilla. Además, ha demostrado que los modelos Vision Transformer tienen potencial en el ámbito clínico. Esta aplicación puede servir como base para futuras herramientas de diagnóstico, y puede ayudar a los profesionales de la salud a detectar lesiones de forma más rápida.

Uno de los retos más importantes ha sido encontrar arquitecturas Vision Transformer que se pudieran aplicar para este caso. Algunas requerían más canales de entrada aparte de los tres usados (T1, T2 y FLAIR), o menos, perdiéndose mucha información; muchas otras estaban diseñadas para clasificación de imágenes y no para segmentación, etc. Otra dificultad ha sido el elevado coste computacional de los modelos 3D, que limitaba tanto el tamaño de las imágenes como la complejidad del entrenamiento. Se han tenido que tomar decisiones para encontrar un equilibrio entre rendimiento y recursos, por ejemplo, reduciendo el tamaño de las imágenes a 96x96x96 vóxeles.

Por una parte, el preprocesamiento ha sido muy importante para homogeneizar las imágenes y poder realizar el entrenamiento. Durante el entrenamiento ha sido necesario realizar muchas pruebas con diferentes combinaciones de hiperparámetros, como funciones de pérdida, además de aplicar técnicas como *data augmentations*, por lo que todo el proceso llevó mucho tiempo. Para evaluar la calidad de las segmentaciones de estos modelos y poder compararlos se ha utilizado la métrica Dice, ya que al tener mayor sensibilidad detectando zonas pequeñas es ideal para este caso. Gracias a todo este trabajo, se ha conseguido seleccionar un modelo que proporciona segmentaciones fiables, con un Dice de $0.6237 \pm 0.1811$.

Uno de los logros ha sido construir una aplicación web que cualquier profesional de la salud puede usar sin tener que preocuparse por la parte técnica, simplemente sube las imágenes y

visualiza la segmentación. Eso es lo que hace que el proyecto tenga sentido, porque demuestra que esta tecnología no se queda solo en lo experimental, sino que puede usarse en la práctica.

Este trabajo ha sido mucho más que entrenar un modelo, ya que ha implicado resolver problemas técnicos complejos, adaptarse constantemente y traducir una idea de inteligencia artificial en una aplicación real y útil. Esa experiencia es, sin duda, uno de los mayores logros del proyecto.

## 6.2. Líneas Futuras

Existen varias líneas de mejora y expansión del proyecto para trabajos posteriores:

- **Entrenamiento en un entorno más potente**: Ejecutar el proyecto en un equipo con mayor capacidad de procesamiento, especialmente con una GPU más potente y con más memoria VRAM. Así se podría aumentar el tamaño de las imágenes (por ejemplo, de $96^3$ a $128^3$), utilizar un tamaño de lotes más grande y acelerar tanto el entrenamiento como la inferencia del modelo.
- **Ampliación del conjunto de datos**: Incluir imágenes de más pacientes y hospitales, para mejorar la generalización del modelo y evaluar su rendimiento en otros escenarios.
- **Estadísticas y evolución**: Incluir estadísticas como el volumen total de lesiones, número de lesiones y una comparativa con otros pacientes o con valores normales. También podría mostrar cómo han crecido o disminuido las lesiones, comparando segmentaciones anteriores y actuales.

# Referencias


Aburass, S., M. Dorgham, O., Al Shaqsi, J., Abu Rumman, M., & S. Al-Kadi, O. (2025). Vision Transformers in Medical Imaging: a Comprehensive Review of Advancements and Applications Across Multiple Diseases. https://doi.org/10.1007/s10278-025-01481-y

Akiba, T., Sano, S., Yanase, T., Ohta, T., & Koyama, M. (2019). Optuna: A Next-generation Hyperparameter Optimization Framework. *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*.

Algotive. (2022). *Machine Learning: ¿Qué es el aprendizaje automático y cómo funciona?* https://www.algotive.ai/es-mx/blog/machine-learning-que-es-el-aprendizaje-autom%C3%A1tico-y-c%C3%B3mo-funciona

Ambika. (2023). *What is Computer Vision? (History, Applications, Challenges)*. https://medium.com/@ambika199820/what-is-computer-vision-history-applications-challenges-13f5759b48a5

Avi. (2024). *Explicación de FastAPI en 5 minutos o menos*. https://geekflare.com/es/fastapi-explained/

Belcic, I., & Stryker, C. (2025). *What is an ML pipeline?* https://www.ibm.com/think/topics/machine-learning-pipeline

Bergmann, D., & Stryker, C. (2023). *What is PyTorch?* https://www.ibm.com/think/topics/pytorch

Climent Pardo, J. (2024, 18 de agosto). *AI in Medical Imaging for Beginners: III MRI Preprocessing*. https://medium.com/@jc.climentpardo/ai-in-medical-imaging-iii-mri-preprocessing-440625e55968

DocuSign. (2025). *5 aplicaciones prácticas de inteligencia artificial*. https://www.docusign.com/es-es/blog/aplicaciones-inteligencia-artificial

Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, D., Zhai, X., Unterthiner, T., Dehghani, M., Minderer, M., Heigold, G., Gelly, S., Uszkoreit, J., & Houlsby, N. (2021). An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale. *Proceedings of the International Conference on Learning Representations (ICLR)*. https://doi.org/10.48550/arXiv.2010.11929

Fundación Esclerosis Múltiple. (2023). *Lesiones desmielinizantes cerebrales: en qué consisten*. https://www.fem.es/es/lesiones-desmielinizantes-cerebrales-consisten/

Gan, D., Chang, M., & Chen, J. (2024). 3D-EffiViTCaps: 3D Efficient Vision Transformer with Capsule for Medical Image Segmentation. *arXiv preprint arXiv:2403.16350*. https://doi.org/10.48550/arXiv.2403.16350

GeeksforGeeks. (2025a). *Computer Vision Tutorial*. https://www.geeksforgeeks.org/computer-vision/

GeeksforGeeks. (2025b). *Introduction to JavaScript*. https://www.geeksforgeeks.org/javascript/introduction-to-javascript/

Google Cloud. (2025a). *¿Qué es el aprendizaje automático (AA)?* https://cloud.google.com/learn/what-is-machine-learning?hl=es-419

Google Cloud. (2025b). *¿Qué es el aprendizaje profundo?* https://cloud.google.com/discover/what-is-deep-learning

Hatamizadeh, A., Nath, V., Tang, Y., Yang, D., Roth, H., & Xu, D. (2022). Swin UNETR: Swin Transformers for Semantic Segmentation of Brain Tumors in MRI Images. *arXiv preprint arXiv:2201.01266*. https://doi.org/10.48550/arXiv.2201.01266

Holdsworth, J., & Scapicchio, M. (2024). *What is deep learning?* https://www.ibm.com/think/topics/deep-learning

Huynh, N. (2023). Understanding Evaluation Metrics in Medical Image Segmentation. https://medium.com/@nghihuynh_37300/understanding-evaluation-metrics-in-medical-image-segmentation-d289a373a3f

IBM. (2021). *¿Qué son las redes neuronales?* https://www.ibm.com/es-es/think/topics/neural-networks

Jurafsky, D., & Martin, J. H. (2025). Speech and Language Processing. https://web.stanford.edu/~jurafsky/slp3/9.pdf

Karimian, K. (2024). *What Is a Neural Network? A Simple Explanation*. https://www.alation.com/blog/what-is-a-neural-network/

Li, X., Ding, H., Yuan, H., Zhang, W., Pang, J., Cheng, G., Chen, K., Liu, Z., & Loy, C. (2024). Transformer-Based Visual Segmentation: A Survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence*. https://doi.org/10.48550/arXiv.2304.09854

LogicMelt. (2024). *Inteligencia artificial para la automatización industrial*. https://logicmelt.com/es/blog/inteligencia-artificial-para-la-automatizacion-industrial/

Martínez Perandones, S. (2024). *Qué es html y para qué sirve*. https://www.cursosfemxa.es/blog/estudiar-html

Mayo Clinic. (2025). Esclerosis múltiple – Descripción general. https://www.mayoclinic.org/es/diseases-conditions/multiple-sclerosis/symptoms-causes/syc-20350269

MDN Web Docs. (2025). *CSS*. https://developer.mozilla.org/es/docs/Web/CSS

MONAI. (2025a). MONAI: Medical Open Network for AI. https://github.com/Project-MONAI/MONAI

MONAI. (2025b). *MONAI: Transforms Documentation*. https://docs.monai.io/en/stable/transforms.html

MRI Master. (2023). *T1 vs T2 vs PD vs FLAIR MRI: Physics and Image Comparison.* https://mrimaster.com/t1-vs-t2-vs-pd-vs-flair-mri/

Neville. (2023). Dice Loss in Medical Image Segmentation. https://cvinvolution.medium.com/dice-loss-in-medical-image-segmentation-d0e476eb486

Nova, S. (2023). *Resonancia magnética cerebral normal.* https://www.kenhub.com/es/library/anatomia-es/rm-cerebral-normal

NVIDIA. (2025). *PyTorch.* https://www.nvidia.com/en-us/glossary/pytorch/

Perera, S., Navard, P., & Yilmaz, A. (2024). SegFormer3D: an Efficient Transformer for 3D Medical Image Segmentation. *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW).* https://doi.org/10.48550/arXiv.2404.10156

Plan de Recuperación, Transformación y Resiliencia. (2023). *Qué es la Inteligencia Artificial.* https://planderecuperacion.gob.es/noticias/que-es-inteligencia-artificial-ia-prtr

Raj, A. (2023). Image Segmentation Using Vision Transformers (ViT). https://medium.com/@ankitrajsh/image-segmentation-using-vision-transformers-vit-a-deep-dive-with-cityscapes-and-camvid-datasets-fc1ccdca295b

Ramírez, S. (2024). *FastAPI.* https://fastapi.tiangolo.com/es/

Roche Pacientes. (2024). Resonancia magnética y esclerosis múltiple. https://rochepacientes.es/esclerosis-multiple/resonancia-magnetica.html

SISE. (2024). *¿Cuáles son las características del lenguaje de Python?* https://www.sise.edu.pe/blog/caracteristicas-principales-python

Thisanke, H., Deshan, C., Chamith, K., Seneviratne, S., Vidanaarachchi, R., & Herath, D. (2023). Semantic Segmentation using Vision Transformers: A Survey. *Engineering Applications of Artificial Intelligence.* https://doi.org/10.48550/arXiv.2305.03273

Ultralytics. (2025). *Segmentación de imágenes.* https://www.ultralytics.com/es/glossary/image-segmentation

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., & Polosukhin, I. (2017). Attention Is All You Need. *Proceedings of the 31st International Conference on Neural Information Processing Systems (NIPS).* https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf

Wikipedia. (2025a). *Medical Open Network for AI.* https://en.wikipedia.org/wiki/Medical_open_network_for_AI

Wikipedia. (2025b). *Python (programming language).* https://es.wikipedia.org/wiki/Python

Yassin, A. (2024). Adam vs. AdamW: Understanding Weight Decay and Its Impact on Model Performance. https://yassin01.medium.com/adam-vs-adamw-understanding-weight-decay-and-its-impact-on-model-performance-b7414f0af8a1

# Apéndice A
# Manual de Instalación

## A.1. Requisitos previos

Antes de comenzar la instalación, es necesario tener instalados los siguientes componentes:

- **Python 3.11**: Descargar desde https://www.python.org/downloads/release/python-3118/
- **CUDA Toolkit 12.8**: si se dispone de una tarjeta gráfica NVIDIA, se puede instalar CUDA para aprovechar la aceleración por GPU. Descargar desde https://developer.nvidia.com/cuda-12-8-0-download-archive

## A.2. Descarga del proyecto

Primero, se debe descargar y descomprimir el archivo .zip de la entrega en un directorio. Debe tener la estructura mostrada en la Figura 44. Cabe destacar que las carpetas se podrán consultar mediante el enlace proporcionado debido a su gran tamaño, junto con el resto de los archivos.

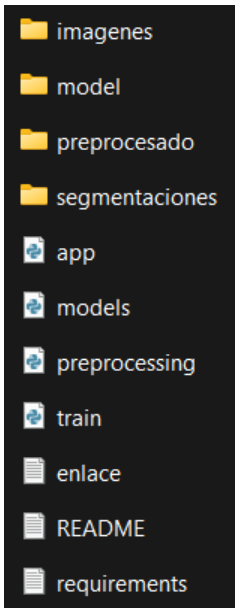


Figura 44: Estructura del proyecto.

## A.3. Creación del entorno virtual

Para evitar conflictos entre versiones de librerías y *frameworks*, se recomienda crear un entorno virtual. Para ello, abrir la consola de comandos y ejecutar:

```
cd "directorio"
```

donde "directorio" debe reemplazarse por la ruta donde se encuentra la carpeta del proyecto. Una vez dentro del directorio del proyecto, crear el entorno virtual y luego activarlo con los siguientes comandos:

```
python -m venv venv
venv\Scripts\activate
```

Una vez activado, el nombre del entorno (en este caso, `(venv)`) aparecerá al principio de la línea en la consola, como se muestra en la Figura 45.

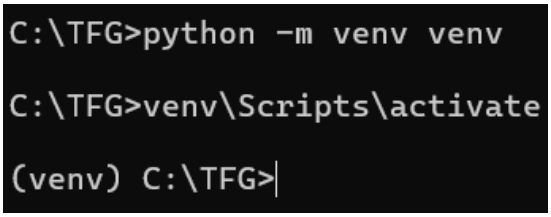


Figura 45: Entorno virtual activado.

## A.4. Instalación de PyTorch con CUDA

Si el equipo dispone de una GPU compatible con CUDA 12.8, ejecutar el siguiente comando para instalar la versión adecuada de PyTorch:

```
pip3 install torch torchvision torchaudio --index-url https://download.pytorch.org/whl/cu128
```

**Nota:** Si no se dispone de GPU o no se ha instalado CUDA, se puede instalar una versión de PyTorch sin soporte CUDA:

```
pip install torch torchvision torchaudio
```

## A.5. Instalación de dependencias

Con el entorno virtual activo, ejecutar:

```
pip install -r requirements.txt
```

Esto instalará las librerías y *frameworks* necesarios. El archivo "requirements.txt" debe contener:

```
monai==1.4.0
fastapi==0.115.12
uvicorn==0.34.2
scikit-learn==1.6.1
nibabel==5.3.2
SimpleITK==2.5.0
dipy==1.11.0
nilearn==0.11.1
matplotlib==3.10.3
optuna==4.3.0
einops==0.8.1
pytorch-lightning==2.5.1.post0
timm==0.9.2
python-multipart==0.0.20
```

## A.6. Ejecución de la aplicación

Una vez completadas las instalaciones, se puede lanzar la API desde la consola dentro del entorno virtual creado. Para ello, se debe ejecutar el siguiente comando:

```
uvicorn app:app --reload
```

donde `app:app` hace referencia al archivo "app.py" y a la instancia `app` de FastAPI en su interior. La opción `--reload` permite recargar automáticamente la aplicación si se hacen cambios. Esto iniciará el servidor localmente.

Para acceder a la interfaz interactiva de la API, abrir un navegador web y entrar a la siguiente dirección:

```
http://127.0.0.1:8000
```

Desde esta interfaz es posible subir imágenes, realizar pruebas de segmentación y visualizar las respuestas de la API.

UNIVERSIDAD DE MÁLAGA | uma.es

E.T.S de Ingeniería Informática
Bulevar Louis Pasteur, 35
Campus de Teatinos
29071 Málaga

E.T.S. DE INGENIERÍA INFORMÁTICA