UNIVERSIDAD POLITÉCNICA DE MADRID

E.T.S. DE INGENIERÍA DE SISTEMAS INFORMÁTICOS

PROYECTO FIN DE GRADO

GRADO EN INGENIERÍA DE COMPUTADORES

Del vóxel al diagnóstico: Segmentación 3D de la aurícula izquierda en resonancias magnéticas cardíacas

Desarrollado por: Rocío Frontaura Úbeda

Dirigido por: Edgar Talavera Muñoz

Madrid, 17 de julio de 2025

Del vóxel al diagnóstico: Segmentación 3D de la aurícula izquierda en resonancias magnéticas cardíacas
Desarrollado por: Rocío Frontaura Úbeda
Dirigido por: Edgar Talavera Muñoz
Proyecto Fin de Grado, 17 de julio de 2025

E.T.S. de Ingeniería de Sistemas Informáticos
Campus Sur UPM, Carretera de Valencia (A-3), km. 7
28031, Madrid, España

Si deseas citar este trabajo, la entrada completa en BibTeX es la siguiente:

```
@mastersthesis{citekey,
    title = {Del vóxel al diagnóstico: Segmentación 3D de la aurícula izquierda
en resonancias magnéticas cardíacas},
    type = {Bachelor's Thesis},
    author = {Frontaura Úbeda, R.},
    school = {E.T.S. de Ingeniería de Sistemas Informáticos},
    year = {2025},
    month = {7},
}
```

Agradecimientos

Quisiera dar las gracias a todas las personas que han contribuido, directa o indirectamente, a la realización de este Proyecto Fin de Grado.

Quiero agradecer en primer lugar a mi tutor Edgar por su dedicación, orientación y apoyo durante el desarrollo del proyecto. Su experiencia y disposición han sido fundamentales para superar cada una de las etapas de este proyecto.

Gracias a mi familia, por su apoyo incondicional en todo momento. Por su paciencia y ánimo en los momentos más difíciles. Por su comprensión y por ser un pilar constante a lo largo de mi vida. A mis abuelos, que a pesar de la diferencia generacional, siempre han mostrado interés y orgullo por mis estudios. A mis padres y a mis hermanos, Marcos y Noelia, que pese al mal humor y los nervios en muchos momentos han sabido comprenderme y tener paciencia.

También quiero dar las gracias a mis compañeros y compañeras, por compartir ideas, resolver dudas y pasar buenos momentos juntos.

A todos ellos, gracias.

Resumen

La segmentación automática de estructuras anatómicas en imágenes médicas se ha convertido en un área clave dentro del campo de la inteligencia artificial aplicada a la medicina. Este Proyecto de Fin de Grado se centra en la segmentación de la aurícula izquierda en imágenes por resonancia magnética cardíaca, una tarea de gran relevancia clínica por su implicación en el diagnóstico y tratamiento de patologías como la fibrilación auricular.

Para abordar este problema, se han entrenado y comparado distintos modelos de segmentación 3D, evaluando su rendimiento mediante experimentación sistemática. Los modelos estudiados son los siguientes: **UNet**, **AttentionUNet**, **SegResNet** y **DynUNet**. Se ha puesto especial atención en aspectos como la estabilidad durante el entrenamiento, la capacidad de generalización y la calidad de las segmentaciones obtenidas, tanto a nivel cuantitativo como cualitativo.

El desarrollo del proyecto ha incluido un pipeline de entrenamiento, validación y análisis, con técnicas de regularización, parada temprana y evaluación continua mediante la métrica de **Dice Coefficient**. Los resultados obtenidos muestran que modelos como **SegResNet** alcanzan altos niveles de precisión, con valores de **Dice** superiores al 0.90, destacando su potencial en aplicaciones clínicas.

Este trabajo contribuye a la evaluación comparativa de arquitecturas de segmentación 3D en un contexto clínico específico, y sienta las bases para futuras líneas de investigación.

Palabras clave: Imagen por Resonancia Magnética (MRI); Deep Learning; Segmentación 3D; Redes Neuronales Convolucionales (CNN); Aprendizaje Supervisado.

Abstract

The automatic segmentation of anatomical structures in medical imaging has become a key area within the field of artificial intelligence applied to healthcare. This Bachelor's Thesis focuses on the segmentation of the left atrium in cardiac magnetic resonance imaging (MRI), a task of high clinical relevance due to its role in the diagnosis and treatment of conditions such as atrial fibrillation.

To address this problem, several 3D segmentation models have been trained and compared, evaluating their performance through systematic experimentation. The models studied include **UNet**, **AttentionUNet**, **SegResNet** and **DynUNet**. Special attention has been given to training stability, generalization ability, and the quality of the resulting segmentations, both quantitatively and qualitatively.

The project development involved a complete pipeline for training, validation, and analysis, incorporating regularization techniques, early stopping, and continuous evaluation using the **Dice Coefficient**. The results show that models such as **SegResNet** achieve high levels of accuracy, with **Dice** values exceeding 0.90, highlighting their potential for clinical applications.

This work contributes to the comparative evaluation of 3D segmentation architectures in a specific clinical context and lays the groundwork for future research.

Keywords: Magnetic Resonance Imaging (MRI); Deep Learning; 3D Segmentation; Convolutional Neural Networks (CNN); Supervised Learning.

Índice general

Índice de figuras

Índice de tablas

Índice de listados

1. Introducción

Las enfermedades cardiovasculares representan uno de los mayores desafíos para la salud pública a nivel global. Según la Organización Mundial de la Salud (OMS), se estima que 17,9 millones de personas fallecen cada año como consecuencia de una enfermedad cardiovascular, lo que las convierte en la principal causa de defunción en el mundo [1]. La detección temprana es, por tanto, un factor clave para poder iniciar tratamientos que mejoren el pronóstico y la calidad de vida de los pacientes.

En este contexto, las técnicas de diagnóstico por imagen han demostrado ser herramientas invaluables. A diferencia de otras técnicas como los Rayos X (RX) o la Tomografía computarizada (TC), la Imagen por Resonancia Magnética (MRI) utiliza un fuerte campo magnético y ondas de radio para obtener imágenes detalladas y claras de los órganos y tejidos internos sin exponer al paciente a radiación ionizante [2]. Esta capacidad para capturar con gran precisión la morfología y estructura del corazón convierte a la MRI en una técnica adecuada para la evaluación de patologías cardiovasculares complejas, como cardiomiopatías, enfermedades congénitas y anomalías funcionales.

No obstante, la utilidad clínica de estas imágenes depende en gran medida de su correcta interpretación. La segmentación de imágenes es esencial en la imagenología médica, ya que permite identificar regiones de interés, facilitando su análisis y asistiendo a los profesionales de la salud en el diagnóstico, seguimiento y tratamiento de enfermedades. El desarrollo de modelos de segmentación puede, por tanto, reducir significativamente la carga de trabajo, mejorar la eficiencia diagnóstica y proporcionar atención médica rápida y personalizada. En cardiología, la segmentación precisa de estructuras como los ventrículos, las aurículas o el miocardio es crucial para evaluar parámetros funcionales como el volumen sistólico, la fracción de eyección o el grosor parietal, contribuyendo así a una valoración más objetiva y reproducible del estado del paciente.

En los últimos años, la segmentación de imágenes médicas ha evolucionado gracias a los avances en Visión por Computador (CV), Aprendizaje Automático (ML) e Inteligencia Artificial (IA), utilizando algoritmos que interpretan datos visuales complejos de manera precisa y eficiente. Su aplicación en el ámbito clínico ha contribuido a mejorar el análisis de imágenes, reducir los tiempos de procesamiento y aumentar la fiabilidad de los diagnósticos. Modelos basados en arquitecturas profundas, como las Redes Neu-

ronales Convolucionales (CNN), han demostrado un rendimiento sobresaliente en tareas de segmentación, superando en muchos casos a los métodos tradicionales tanto en exactitud como en robustez. Estas tecnologías no solo permiten automatizar procesos repetitivos, sino también detectar patrones sutiles que podrían pasar desapercibidos por el ojo humano, lo que abre nuevas posibilidades en la medicina personalizada y preventiva.

Este Proyecto Fin de Grado (PFG) busca investigar y comparar distintos modelos de segmentación aplicados a Resonancia Magnética Cardíaca (RMC). A través de la evaluación de su rendimiento, se busca explorar el potencial de la segmentación en el ámbito médico y sentar las bases para el desarrollo de herramientas asistenciales más eficaces y seguras. El análisis detallado de estos modelos permitirá comprender mejor sus fortalezas y limitaciones, así como su aplicabilidad en entornos clínicos reales. De este modo, se pretende contribuir al avance de soluciones tecnológicas que apoyen de forma directa el trabajo de los profesionales sanitarios y, en última instancia, mejoren la calidad de vida de los pacientes.

1.1. Motivación

La motivación de este proyecto surge de mi interés por la inteligencia artificial y su aplicación en ámbitos tan importantes como la tecnología médica. Durante mi formación académica, he comprobado el potencial de la visión por computador en la interpretación de datos visuales complejos. Estas técnicas no solo ofrecen soluciones avanzadas de ingeniería, sino que también emulan la capacidad del ser humano de reconocer patrones, una habilidad imprescindible en el diagnóstico por imagen. La IA es capaz de aprender de manera autónoma y adaptarse a nuevos datos reflejando, de manera sorprendente, el proceso mediante el cual un especialista adquiere y aplica su experiencia para interpretar imágenes.

Por otro lado, el carácter no invasivo y la capacidad de obtener imágenes detalladas del corazón hacen que la resonancia magnética cardíaca sea una herramienta diagnóstica de valor incalculable en el diagnóstico y tratamiento de enfermedades cardiovasculares. La integración de la IA en el análisis de estas imágenes ofrece una oportunidad única para asistir a los profesionales en la toma de decisiones, mejorando la precisión y eficiencia al automatizar tareas como la segmentación.

Al afrontar este desafío, aspiro a conectar ambos campos de estudio: aplicar las técnicas más avanzadas de visión por computador a la medicina. Este proyecto no solo me brinda la oportunidad de poner en práctica los conocimientos adquiridos durante la carrera, sino también realizar un análisis comparativo entre diferentes modelos de vanguardia, con el objetivo de identificar aquel que ofrezca los mejores resultados y, en consecuencia, el mayor potencial para mejorar el diagnóstico y la calidad de vida de los pacientes.

1.2. Objetivos

Este PFG tiene como objetivo desarrollar un sistema capaz de segmentar resonancias magnéticas cardíacas de manera precisa, fiable y eficiente. De este modo, se plantean los siguientes objetivos específicos:

- **Explorar en detalle la técnica de Imagen por Resonancia Magnética:** comprender sus fundamentos e historia, así como su importancia en el diagnóstico de enfermedades cardiovasculares.
- **Investigar y analizar las arquitecturas de segmentación 3D más relevantes:** profundizar en el estudio de los principales modelos utilizados en la segmentación de imágenes médicas, explicando su funcionamiento y características.
- **Desarrollar un pipeline para el procesamiento de los datos:** diseñar un flujo de trabajo que incluya la carga, preprocesamiento, normalización, aumento y división de los datos en conjuntos de entrenamiento, validación y prueba.
- **Entrenar y validar los modelos seleccionados:** implementar las arquitecturas estudiadas, ajustando sus hiperparámetros para optimizar el rendimiento de cada una de ellas.
- **Comparar el rendimiento de los distintos modelos:** evaluar objetivamente los resultados obtenidos, utilizando métricas tanto cuantitativas como cualitativas, con el fin de extraer conclusiones.

1.3. Estructura de la Memoria

El presente documento se estructura de la siguiente manera. En primer lugar, en la sección *Marco Teórico* se ofrece una visión general de los conceptos teóricos indispensables para comprender el proyecto en su totalidad, abarcando los fundamentos de la imagen por resonancia magnética, las bases de la segmentación y los principios del *Deep Learning*, así como las arquitecturas y paradigmas utilizados.

A continuación, en *Estado de la Cuestión* se realiza un análisis de la literatura actual y los trabajos más relevantes dentro de este ámbito. Más adelante, en el capítulo *Metodología* se explica detalladamente el flujo seguido para abordar los objetivos del PFG, y en la sección *Resultados* se exponen los hallazgos más importantes para derivar conclusiones pertinentes.

En la sección *Impacto del Proyecto* se discuten las consecuencias sociales, ambientales y profesionales del PFG. Finalmente, en *Conclusiones y Proyección a Futuro* se recogen las reflexiones finales y los pasos a seguir para la posible aplicación industrial del proyecto.

2. Estado de la Cuestión

La imagen por resonancia magnética ha revolucionado el campo de la medicina, transformando el diagnóstico y tratamiento de enfermedades al proporcionar imágenes detalladas de órganos y tejidos del cuerpo. Se ha convertido en una técnica de imagenología esencial con aplicaciones en especialidades como la neurología, cardiología y oncología, gracias a los avances tecnológicos y al trabajo constante de investigadores y profesionales que han perfeccionado su uso e impulsado su evolución.

2.1. Imagen por Resonancia Magnética (MRI)

Según el Instituto Nacional de Imágenes Biomédicas y Bioingeniería (NIBIB), la imagen por resonancia magnética es una técnica de imagenología no invasiva que produce imagénes anatómicas tridimensionales detalladas sin utilizar radiazación ionizante. Esta tecnología versátil se utiliza habitualmente para la detección, evaluación, tratamiento y seguimiento de enfermedades.

La MRI emplea imanes de gran potencia que producen un fuerte campo magnético que obliga a los protones del cuerpo a alinearse con dicho campo. A continuación, se emite un pulso de radiofrecuencia que perturba temporalmente esta alineación, estimulando a los protones para que giren fuera de equilibrio y luchen contra la fuerza del campo magnético. Cuando el pulso de radiofrecuencia cesa, los protones vuelven a alinearse con el campo magnético, liberando energía en el proceso. Los sensores del escáner detectan esta energía y miden el tiempo que tardan los protones en realinearse, variables que cambian dependiendo del entorno y la naturaleza química de las moléculas. Gracias a estas propiedades magnéticas, los médicos son capaces de distinguir entre diferentes tipos de tejidos y estructuras anatómicas.

Para obtener una imagen por MRI, se coloca al paciente en una camilla que se desliza dentro de un escáner cilíndrico (figura 2.1), donde se encuentran los imanes. El paciente debe permanecer inmóvil durante todo el procedimiento para que la imagen no salga borrosa. En algunos casos, se administra al paciente un medio de contraste por

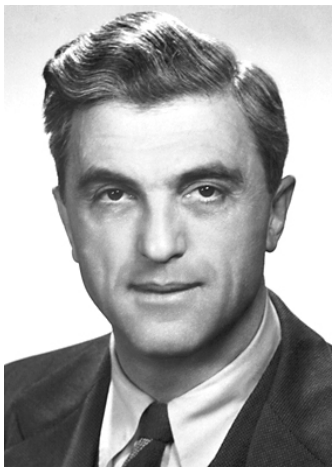
vía intravenosa, que generalmente contiene gadolinio, para modificar las propiedades magnéticas de los tejidos. Este medio aumenta la velocidad a la cual los protones se realinean con el campo magnético, lo que produce una imagen más nítida y brillante [3] [4].

Figura 2.1. Equipo de Resonancia Magnética [5]

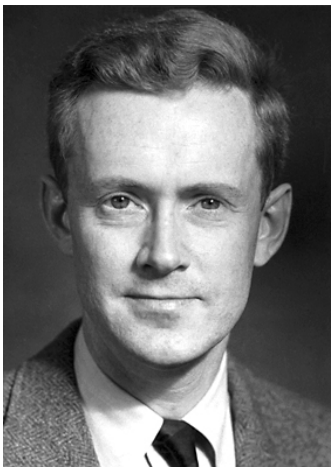
Las bases de la MRI se remontan al descubrimiento de la Resonancia Magnética Nuclear (NMR) en la década de los 40. Los físicos Felix Bloch (figura 2.2a) y Edward Purcell (figura 2.2b) descubrieron que el núcleo celular, sometido a un campo magnético, podía absorber y emitir energía proveniente de ondas de radiofrecuencia. Gracias a este descubrimiento ganaron el Premio Nobel de Física en 1952 y sentaron las bases de la NMR. La transición a la MRI comenzó a principios de los años 70, cuando los investigadores vieron el potencial de la NMR para obtener imágenes del cuerpo humano. El Dr. Raymond Damadian (figura 2.2c) fue uno de los primeros en proponer la idea de utilizar la NMR para detectar tejidos cancerosos. En 1971, publicó un artículo en el que demostraba que la NMR podía distinguir entre tejidos sanos y cancerosos, lo que despertó el interés por las aplicaciones médicas de esta tecnología.

En los años venideros, los avances en esta tecnología condujeron al desarrollo de los primeros escáneres de NMR. En 1973, el químico Paul Lauterbur (figura 2.2d) describió una técnica que utilizaba gradientes dentro del campo magnético para generar imágenes bidimensionales. El trabajo de Lauterbur, combinado con las aportaciones del físico Peter Mansfield (figura 2.2e), conocido por desarrollar técnicas para la obtención rápi-

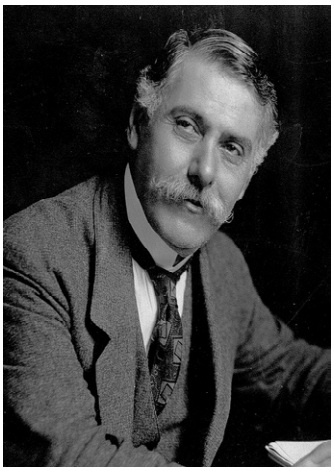
da de imágenes, culminó en la producción de las primeras imágenes por resonancia magnética. Ambos recibieron el Premio Nobel de Medicina en 2003 por su trabajo. A finales de los años 70 y principios de los 80 se construyeron los primeros escáneres de resonancia magnética capaces de obtener imágenes del cuerpo humano. En 1977, el Dr. Damadian y su equipo completaron la primera resonancia magnética de cuerpo completo de un ser humano, lo que marcó un hito importante en la obtención de imágenes médicas. Este logro demostró las aplicaciones prácticas de la MRI y allanó el camino para la adopción generalizada de esta tecnología en el ámbito clínico.



(a) Felix Bloch



(b) Edward Purcell



(c) Raymond Damadian

(d) Paul Lauterbur

(e) Peter Mansfield

Figura 2.2. Pioneros de la Imagen por Resonancia Magnética [6] [7] [8] [9] [10]

La década de los 80 marcó la comercialización y adopción clínica de esta tecnología. Los avances en informática y algoritmia mejoraron la calidad y velocidad de las explo-

raciones, aumentando su capacidad de diagnóstico. Desde su creación, la MRI no ha dejado de evolucionar gracias a la investigación y el desarrollo. Avances como los sistemas de alto campo y la introducción de la MRI funcional (fMRI) han hecho posible la aplicación de esta tecnología en campos como la neurología y cardiología [11] [12].

2.1.1. Aplicación Clínica en Cardiología

Tras su consolidación como técnica de imagen avanzada, la MRI se ha convertido en una herramienta de gran utilidad en áreas como la cardiología. La obtención de imágenes detalladas del corazón (figura 2.3) y los vasos sanguíneos es crucial para evaluar diferentes aspectos del sistema cardiovascular, incluyendo:

- El tamaño y la función de las cavidades del corazón.
- El grosor y el movimiento de las paredes del corazón.
- La extensión del daño causado por ataques cardíacos o enfermedades cardíacas.
- Los problemas estructurales en la aorta.
- La inflamación u obstrucción en los vasos sanguíneos. [13]

Dentro de las múltiples aplicaciones de la RMC, este proyecto se centrará en el análisis de la aurícula izquierda, una de las cuatro cavidades del corazón. Esta cavidad recibe sangre rica en oxígeno procedente de los pulmones y la vacía en el ventrículo izquierdo [14], que se encarga de bombearla al resto del cuerpo. La segmentación de la aurícula izquierda permitirá evaluar su tamaño, forma y función, facilitando el diagnóstico y la detección de patologías.

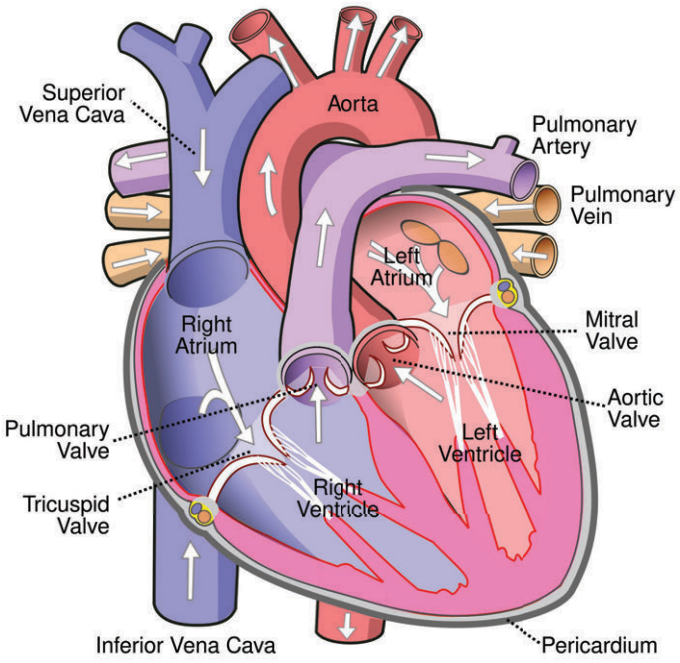


Figura 2.3. Anatomía del Corazón [15]

2.2. Deep Learning

A continuación, se definen los fundamentos del *Deep Learning* y las principales arquitecturas utilizadas en este PFG, con el objetivo de dar al lector el contexto necesario para comprender adecuadamente el trabajo realizado.

2.2.1. Fundamentos del Deep Learning

La IA es un campo de la informática que engloba a todos los sistemas capaces de realizar tareas que normalmente requieren inteligencia humana, como el aprendizaje, el razonamiento y la percepción [16]. Dentro de este campo se encuentra el *Machine Learning* (ML), una disciplina que se centra en el desarrollo de algoritmos que permiten a las máquinas aprender de los datos y tomar decisiones o hacer predicciones sin estar explícitamente programadas para ello [17].

El Deep Learning (DL) es una rama del ML que se basa en el uso de redes neuronales

profundas para modelar y resolver problemas complejos. A diferencia de los enfoques tradicionales de ML, el DL hace posible que las máquinas aprendan de manera autónoma, razonen y extraigan conclusiones, perfeccionando sus modelos a medida que procesan nuevos datos [18].

En la figura 2.4 se muestra dicha jerarquía.

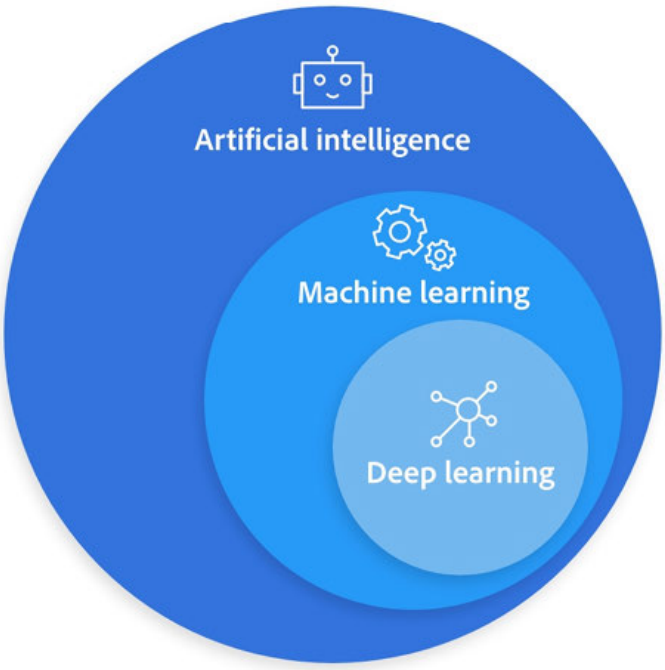


Figura 2.4. Relación entre Inteligencia Artificial, *Machine Learning* y *Deep Learning* [19]

Redes Neuronales

Las Redes Neuronales (NN), propuestas por McCulloch y Pitts en 1943 [20], son modelos matemáticos inspirados en la estructura y funcionamiento del cerebro humano. A diferencia de sus análogas biológicas, las neuronas artificiales no son entidades físicas, sino unidades de cálculo programadas que generan una salida (*output*) combinando un bias con los pesos de sus entradas (*inputs*) y aplicando una función de activación.

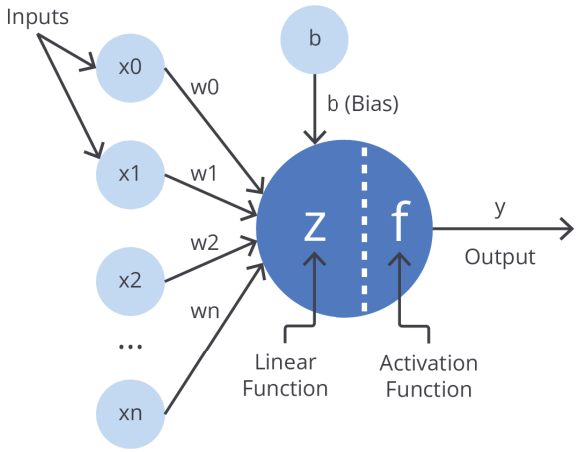


Figura 2.5. Neurona Artificial [21]

Los pesos son parámetros que se ajustan durante el proceso de entrenamiento de la red y determinan la importancia de cada entrada en el cálculo de la salida. Por su parte, el bias es un parámetro constante que aporta una capa adicional de flexibilidad al modelo al permitir ajustar la salida independientemente de los valores de entrada [22].

Siguiendo el ejemplo de la figura 2.5, la salida z de una neurona viene dada por la ecuación 2.1:

$$z = \sum_{i=1}^{n} w_i x_i + b \tag{2.1}$$

donde w son los pesos, x las entradas y b el bias.

Las neuronas se agrupan en capas para formar redes neuronales. Generalmente, una red neuronal está compuesta por una capa de entrada, encargada de recibir los datos; una o varias capas ocultas, responsables de procesar la información y extraer características a diferentes niveles de abstracción; y una capa de salida, que devuelve el resultado de la red (figura 2.6). En las redes densas, cada neurona se conecta con todas las neuronas de la capa siguiente, lo que permite a la red aprender y generalizar a partir de los datos de entrenamiento. Se dice que una red es profunda cuando tiene al menos tres capas ocultas [23].

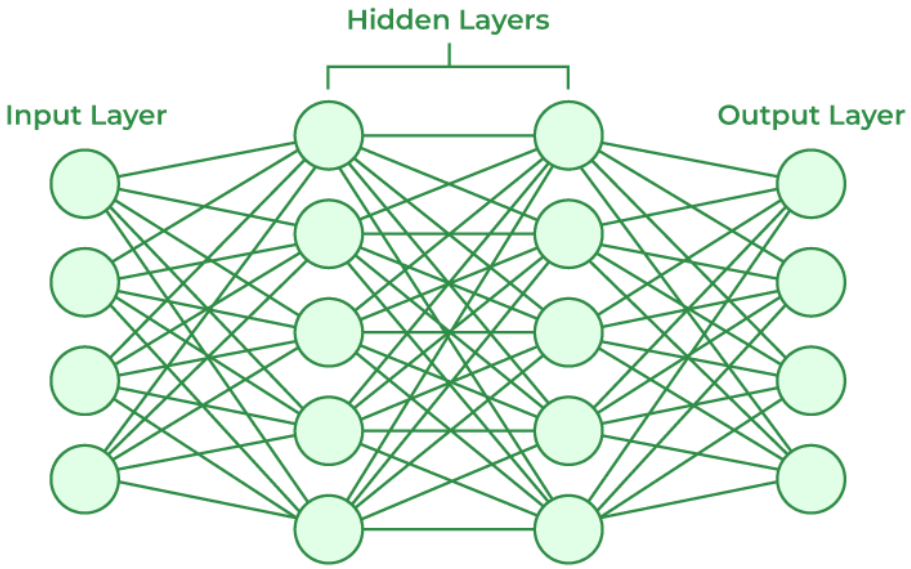


Figura 2.6. Arquitectura de una Red Neuronal [24]

Para que estas redes sean capaces de identificar patrones en conjuntos de datos complejos, es fundamental el uso de funciones de activación. Estas funciones introducen la no linealidad, permitiendo que la red aprenda relaciones complejas que van más allá de la combinación lineal de las entradas. En la figura 2.7 se muestran algunas de las funciones de activación más utilizadas. La función *ReLU* se utiliza en las capas ocultas por su eficiencia computacional y capacidad de acelerar el aprendizaje, mientras que la función *sigmoid* suele emplearse en las capas de salida, en tareas de clasificación binaria [25].

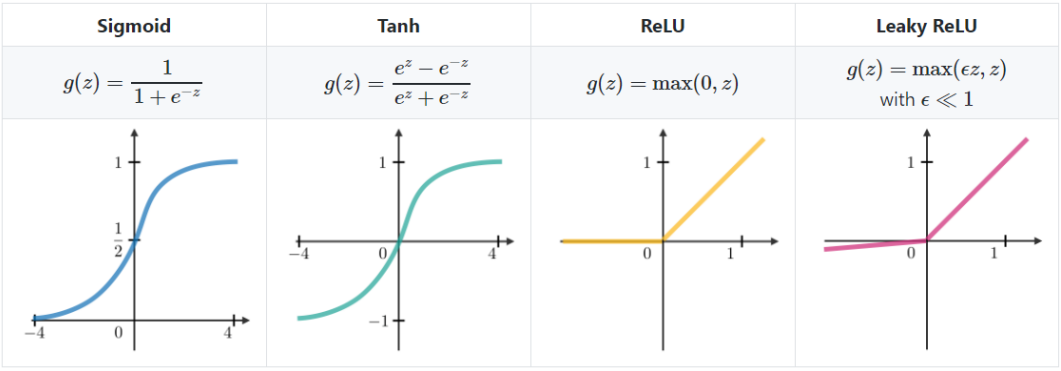


Figura 2.7. Funciones de Activación [26]

De esta forma, se define la ecuación 2.2:

$$a = f(z) \tag{2.2}$$

donde a es la activación y f la función de activación.

Aprendizaje y Optimización

El entrenamiento de una red neuronal consiste en ajustar sus parámetros para que pueda resolver un problema específico. Este ajuste se realiza de forma iterativa a lo largo de varias épocas (*epochs*) mediante un proceso de optimización que se basa en el algoritmo de descenso de gradiente. El objetivo de este algoritmo es minimizar la función de pérdida que cuantifica el error entre las predicciones del modelo y los valores reales esperados [23]. Cada iteración del proceso de entrenamiento (figura 2.8) consta de tres fases:

- ***Forward Propagation:*** En esta fase, las capas de la red neuronal procesan los datos de entrada. Cada neurona combina linealmente las entradas con sus pesos y bias, y luego aplica una función de activación no lineal. La salida de cada capa se convierte en la entrada de la siguiente, hasta obtener la predicción final (*output*) de la red. Este proceso puede expresarse matemáticamente con las ecuaciones 2.1 y 2.2.
- **Cálculo de la pérdida:** Se compara la predicción de la red, $\hat{y}$, con el valor real esperado, y, utilizando una función de pérdida $L(\hat{y}, y)$ que mide el error cometido. El objetivo del aprendizaje es minimizar este error. Entre las funciones de pérdida más utilizadas se encuentran: MAE (*Mean Absolute Error*, ecuación 2.3), MSE (*Mean Squared Error*, ecuación 2.4) y BCE (*Binary CrossEntropy*, ecuación 2.4):

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i| \tag{2.3}$$

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{2.4}$$

$$\text{BCE} = -\frac{1}{n}\sum_{i=1}^{n}[y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)] \tag{2.5}$$

- ***Back Propagation:*** Se calcula el gradiente de la función de pérdida respecto a cada parámetro de la red. Usando la regla de la cadena, estos gradientes se propagan

desde la capa de salida hasta la de entrada. Con ellos, se actualizan los parámetros de la red utilizando las siguientes fórmulas:

$$w = w - \alpha \frac{\partial L}{\partial w} \tag{2.6}$$

$$b = b - \alpha \frac{\partial L}{\partial b} \tag{2.7}$$

donde L es la función de pérdida, $\frac{\partial L}{\partial w}$ y $\frac{\partial L}{\partial b}$ los gradientes y α la tasa de aprendizaje [27].

Este proceso se repite hasta que el modelo converge, es decir, hasta que la función de pérdida se estabilice y se alcance una precisión aceptable en la tarea. Para agilizar la convergencia, es común utilizar optimizadores como Adam (*Adaptive Moment Estimation*) o RMSprop (*Root Mean Square Propagation*) [28].

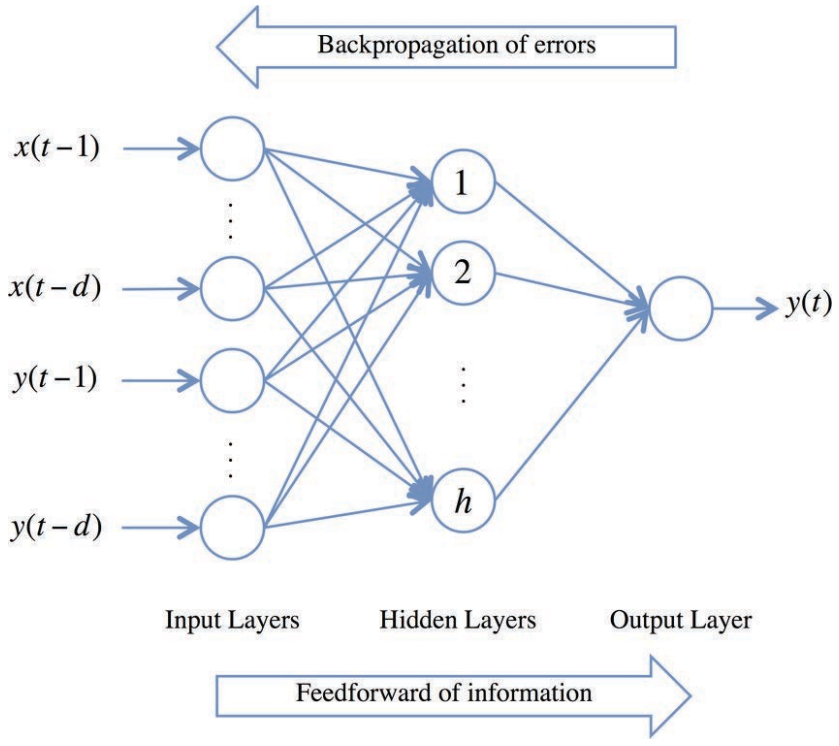


Figura 2.8. Proceso de Aprendizaje de una Red Neuronal [29]

División de los Datos, *Overfitting* y *Underfitting*

Una práctica fundamental en el desarrollo de modelos de aprendizaje automático es la división del conjunto de datos en tres subconjuntos: entrenamiento, validación y prue-

ba. Esta división permite evaluar la capacidad de generalización del modelo. El conjunto de entrenamiento, que normalmente abarca entre el 70 % y el 80 % de los datos, se usa para entrenar el modelo. Durante este proceso, el conjunto de validación se emplea para ajustar los hiperparámetros y supervisar el rendimiento del modelo. Una vez terminado el entrenamiento, el conjunto de prueba se utiliza para evaluar objetivamente la capacidad de generalización del modelo.

A raíz del rendimiento del modelo sobre estos conjuntos se estima el sesgo (*bias*) y la varianza (*variance*) de la red. El sesgo es el error sistemático entre las predicciones del modelo y los valores reales, mientras que la varianza refleja la sensibilidad del modelo a pequeñas variaciones en los datos de entrada. En la figura 2.9 se representa gráficamente la relación entre estos errores y la complejidad del modelo. Los modelos demasiado simples tienden a tener alto sesgo (señal de *underfitting*), mientras que los excesivamente complejos presentan alta varianza (señal de *overfitting*).

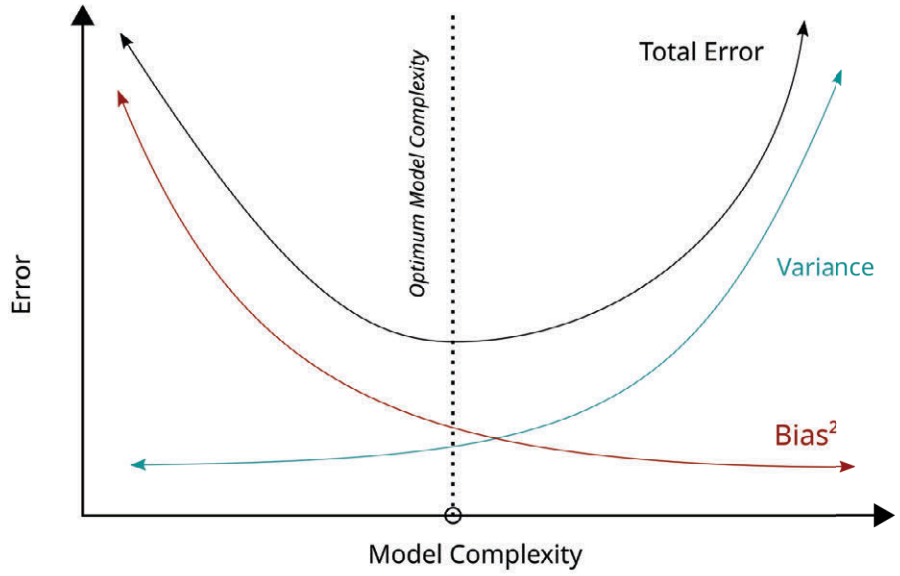


Figura 2.9. *Bias-Variance Tradeoff* [30]

El *overfitting* ocurre cuando la red neuronal es demasiado compleja y se ajusta en exceso a los datos de entrenamiento, capturando no solo los patrones subyacentes, sino también el ruido y las particularidades de esos datos. Esto impide que el modelo pueda generalizar, ya que ha "memorizado"los datos en vez de comprenderlos. Por el contrario, el *underfitting* ocurre cuando el modelo no es lo suficientemente complejo para capturar la estructura de los datos, lo que resulta en un rendimiento pobre tanto en los datos de entrenamiento como en los de prueba (figura 2.10) [31].

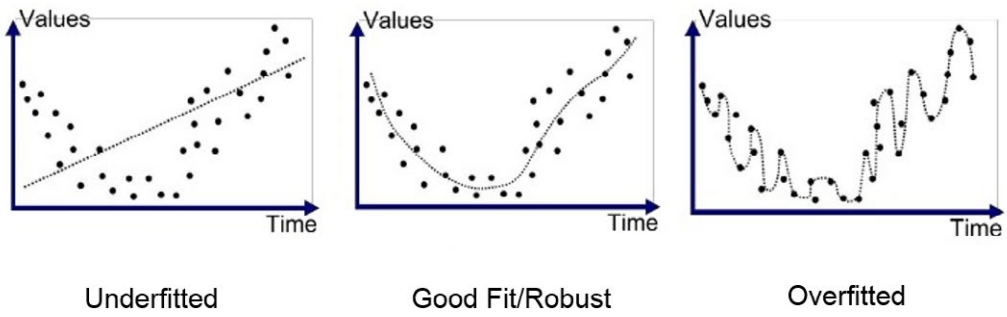


Figura 2.10. *Overfitting*, *Underfitting* y Ajuste Óptimo [32]

El objetivo final es alcanzar un equilibrio entre la complejidad del modelo y su capacidad de generalización.

Regularización

Para combatir el *overfitting* y mejorar la capacidad de generalización de los modelos, se emplean técnicas de regularización. Estas añaden términos de penalización a la función de pérdida para limitar la complejidad del modelo y evitar que sus parámetros tomen valores demasiado grandes, promoviendo así soluciones más robustas y simples. A continuación, se presentan las técnicas de regularización más utilizadas:

- **Regularización L1 y L2:** La regularización L1 o regularización de Lasso añade una penalización proporcional a la suma de los valores absolutos de los coeficientes del modelo. Esta técnica fuerza a algunos coeficientes a ser cero, lo que se traduce en una selección automática de variables. La regularización L2 o regularización de Ridge añade una penalización proporcional a la suma de los cuadrados de los coeficientes del modelo. Esta técnica reduce el valor de los coeficientes, pero sin llegar a anularlos.

  Matemáticamente, ambas técnicas de regularización se definen de la siguiente manera:

$$\mathcal{L}_{\text{L1}}(\theta) = \|\theta\|_1 = \sum_{i=1}^{p} |\theta_i| \tag{2.8}$$

$$\mathcal{L}_{\text{L2}}(\theta) = \|\theta\|_2^2 = \sum_{i=1}^{p} \theta_i^2 \tag{2.9}$$

donde θ es un parámetro de la red y p el número total de parámetros [33].

- ***Dropout:*** El *dropout* es una técnica de regularización, propuesta por Srivastava, Hinton, Krizhevsky, Sutskever y Salakhutdinov [34], que consiste en desactivar aleatoriamente un conjunto de neuronas durante el entrenamiento. Esto evita que las neuronas se especialicen demasiado y fomenta que contribuyan de manera más general a las predicciones del modelo.

  Durante el entrenamiento, cada activación intermedia a se reemplaza por una variable aleatoria a', de tal forma que:

  $$a' = a \cdot m, \quad m \sim \text{Bernoulli}(p) \tag{2.10}$$

  donde p es la probabilidad de que una neurona permanezca activa.

- ***Batch Normalization (BN):*** La normalización por lotes, propuesta por Ioffe y Szeged [35], busca acelerar y estabilizar el entrenamiento de redes neuronales profundas, normalizando la salida lineal de cada capa para que tenga una media de cero y una varianza de uno. Este proceso ayuda a mejorar la convergencia del modelo y a mitigar problemas como el desvanecimiento de gradiente.

  Dado un vector de activaciones z, la normalización por lotes se define del siguiente modo:

  $$\hat{z} = \frac{z - \mu}{\sigma^2 + \epsilon} \tag{2.11}$$

  $$z_{\text{norm}} = \gamma \cdot \hat{z} + \beta \tag{2.12}$$

  donde μ y σ^2 son la media y varianza del mini-lote, ϵ un valor pequeño para mantener la estabilidad numérica y γ y β parámetros que el modelo aprende durante el entrenamiento.

2.2.2. Redes Neuronales Convolucionales (CNN)

Las CNN (figura 2.11) son un tipo de red neuronal profunda especializada en el procesamiento de datos no estructurados, como imágenes, vídeos o señales de audio. A diferencia de las redes tradicionales, que procesan los datos de manera independiente, las CNN utilizan capas convolucionales. Estas capas aplican una serie de filtros (*kernels*) que se "deslizan" sobre la entrada para extraer características y generar mapas de activación (*feature maps*). En las capas iniciales, los filtros aprenden a reconocer patrones simples como texturas o colores, mientras que las capas más profundas combinan estas

características para identificar estructuras más complejas.

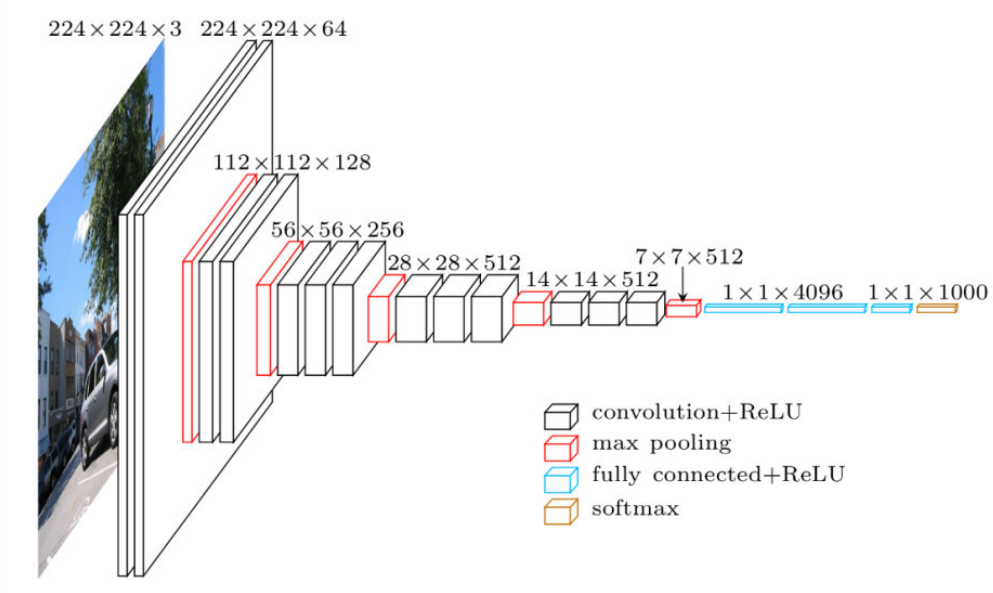


Figura 2.11. Arquitectura de una Red Neuronal Convolucional [36]

Para tareas como el análisis de imágenes, las CNN son mucho más eficientes que las redes densas. Una red densa requiere que la imagen se "aplane" en un vector unidimensional, lo que es computacionalmente costoso y supone la pérdida del contexto espacial. Las CNN ofrecen una solución mucho más eficiente al utilizar *kernels*, matrices de menor tamaño que la entrada, que se deslizan por la imagen con un *stride* determinado y un posible *padding* en los bordes (figura 2.12). Ajustando estos dos hiperparámetros y el número de filtros n_f, se pueden controlar las dimensiones de la salida de cada capa, lo que permite condensar la información en representaciones más profundas.

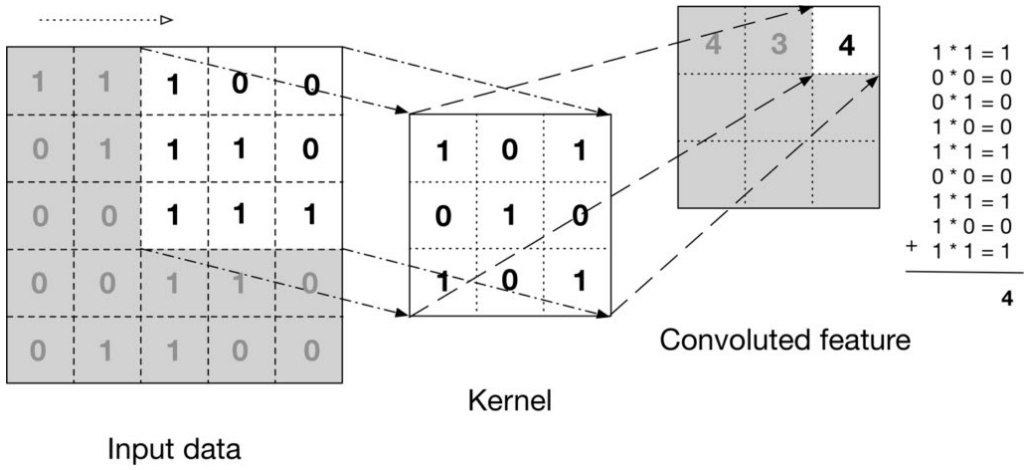


Figura 2.12. Operación de Convolución [37]

La relación entre las dimensiones de entrada (n, n, n_c) y las de salida viene dada por la expresión 2.13, donde el operando $*$ denota la operación de convolución y las dimensiones del *kernel* se representan como (f, f, n_c), siendo n_c el número de canales de entrada.

$$(n, n, n_c) * (f, f, n_c) = (\frac{n + 2p - f}{s} + 1, \frac{n + 2p - f}{s} + 1, n_f) \tag{2.13}$$

Para reducir progresivamente el tamaño de los mapas de activación y el coste computacional, las CNN utilizan capas de *pooling*. Estas capas compactan la información aplicando diferentes técnicas de reducción que no requieren parámetros entrenables. Gracias a esta compresión, se pueden añadir capas densas al final de la red para realizar tareas de clasificación de manera eficiente [38] [39].

2.2.3. Segmentación de Imágenes

La segmentación de imágenes es una técnica de visión por computador que consiste en dividir una imagen en segmentos o regiones con el objetivo de simplificar su representación y facilitar su análisis. A diferencia de la clasificación de imágenes, que asigna una única etiqueta a toda la imagen, la segmentación asigna una etiqueta a cada píxel. Esto permite no solo identificar la forma del objeto o región de interés, sino también la ubicación.

Existen tres tipos de segmentación (figura 2.13), cada una con un nivel de detalle diferente:

- **Segmentación semántica:** Asigna una etiqueta de clase a cada píxel de la imagen, sin diferenciar entre instancias del mismo objeto.
- **Segmentación de instancias:** No solo clasifica cada píxel en una categoría, sino que también diferencia entre distintas instancias de la misma clase. Este tipo de segmentación combina, por tanto, la detección de objetos con la segmentación.
- **Segmentación panóptica:** Es un paradigma que combina la segmentación semántica y de instancias. El objetivo es proporcionar una comprensión global y completa de la escena. Para ello, asigna a cada píxel de la imagen tanto una etiqueta de clase como un identificador de instancia [40] [41].

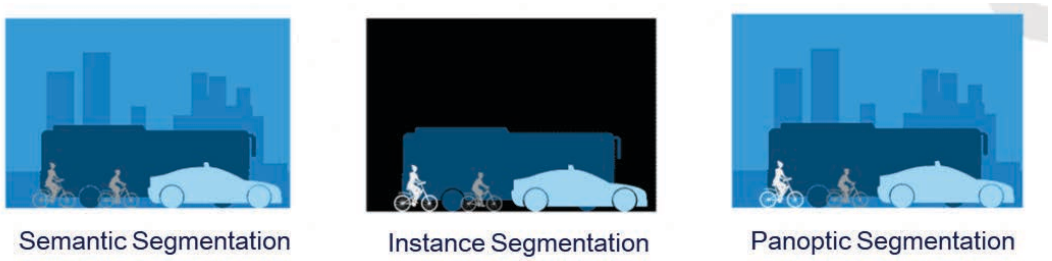


Figura 2.13. Tipos de Segmentación [42]

2.2.4. Arquitecturas de Segmentación

En los últimos años, se han propuesto varios modelos de segmentación semántica, basados en redes convolucionales, que incorporan mecanismos jerárquicos y contextuales para mejorar la precisión y robustez de las segmentaciones. A continuación, se exploran algunos de estos modelos.

UNet

UNet es una arquitectura de red neuronal convolucional diseñada específicamente para tareas de segmentación semántica en imágenes biomédicas. Propuesta por Ronneberger, Fischer y Brox en 2015 [43], esta arquitectura aborda las limitaciones de los métodos tradicionales de segmentación y de las redes convolucionales convencionales, resultando útil en situaciones en las que se dispone de pocos datos etiquetados y se requiere una alta precisión en la segmentación.

Su diseño en forma de "U" se debe a su arquitectura tipo encoder-decoder (figura 2.14). El encoder se encarga de extraer características a distintos niveles de abstracción mediante bloques de capas convolucionales, funciones de activación ReLU y operaciones de *max-pooling*, reduciendo progresivamente la resolución espacial de la imagen hasta generar una representación comprimida.

El decoder reconstruye la segmentación a partir de esa representación comprimida. Para ello, utiliza capas de convolución transpuesta (*upsampling*) que aumentan la resolución y *skip connections* que conectan las capas del encoder con sus contrapartes en el decoder. Estas conexiones ayudan a recuperar la información espacial que podría haberse perdido durante la compresión, mejorando significativamente la precisión en la segmentación [44].

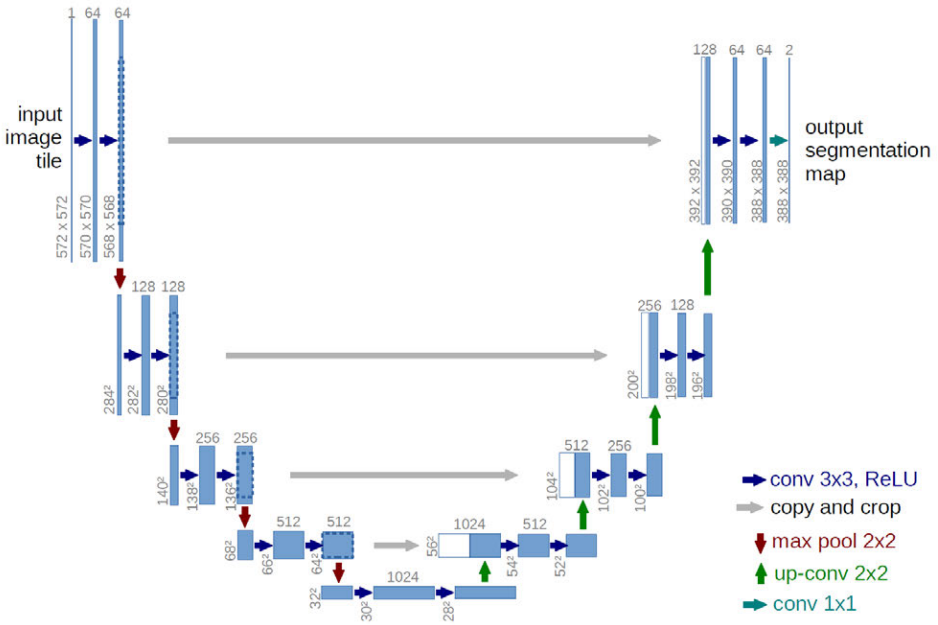


Figura 2.14. Arquitectura U-Net [43]

Una de las grandes ventajas de esta arquitectura es que puede entrenarse con conjuntos de datos relativamente pequeños. Esto se debe gracias al uso de las *skip connections*, la implementación de técnicas de *data augmentation* y la utilización de ventanas deslizantes. Además, UNet optimiza el proceso de segmentación utilizando una función de pérdida basada en la entropía cruzada a nivel de píxel.

Aunque originalmente fue concebida para trabajar con imágenes biomédicas bidimensionales, su diseño puede adaptarse al análisis de datos volumétricos. Para ello, basta

con reemplazar las operaciones 2D por sus equivalentes en 3D, lo que permite al modelo capturar el contexto espacial de los volúmenes [45].

DynUNet

DynUNet es una arquitectura de segmentación semántica diseñada para abordar los desafíos del análisis de imágenes tridimensionales. Esta variante de UNet es capaz de adaptarse automáticamente a diferentes resoluciones espaciales y a la diversidad anatómica presente en volúmenes médicos. Aunque conserva la estructura característica de encoder-decoder simétrico con *skip connections* de UNet, DynUNet incorpora mecanismos que le permiten ajustarse dinámicamente a diversos escenarios clínicos (figura 2.15), lo que mejora su eficiencia computacional, escalabilidad y capacidad de adaptación.

Esta arquitectura permite ajustar de manera dinámica parámetros como la profundidad de la red, el tamaño del *kernel* y el *stride*, basándose en las propiedades espaciales del volumen de entrada. Esta capacidad de adaptación a distintos tamaños y resoluciones de imagen es especialmente útil en el ámbito clínico debido a la variabilidad de los protocolos de adquisición y las regiones anatómicas de interés. En lugar de utilizar una estructura fija, DynUNet construye sus bloques de convolución y *upsampling* de forma dinámica. De este modo, ajusta la relación entre el detalle espacial y la profundidad semántica en función de la jerarquía de escalas que presenta el volumen.

DynUNet usa bloques residuales en lugar de capas convolucionales tradicionales. Estos bloques permiten una propagación del gradiente más estable durante el entrenamiento, lo que permite construir redes profundas sin comprometer el rendimiento. También utiliza *deep supervision*, una técnica que aplica la función de pérdida no solo al final del decoder, sino también en capas intermedias, lo que mejora la propagación del error y la segmentación de estructuras pequeñas o morfológicamente complejas [46].

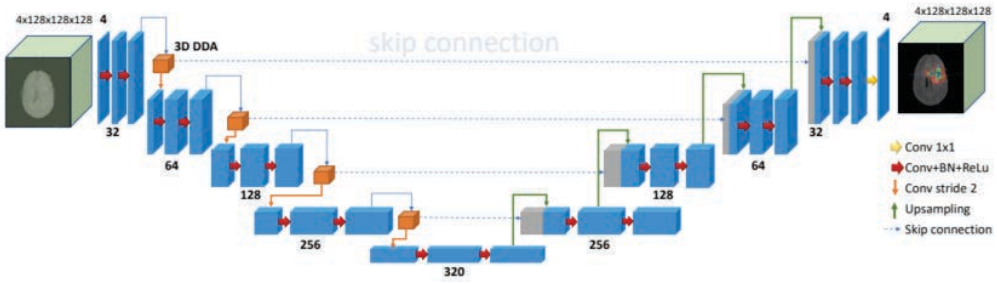


Figura 2.15. Arquitectura DynU-Net [47]

DynUNet fue concebida para trabajar con volúmenes tridimensionales, utilizando convoluciones, *pooling* y *upsampling* para capturar de forma efectiva las relaciones anatómicas en los tres ejes del espacio. Gracias a su diseño, DynUNet es una solución robusta para su integración en flujos de trabajo clínicos que requieren adaptarse a diferentes modalidades de imagen médica, niveles de resolución y estructuras anatómicas de interés.

AttentionUNet

AttentionUNet es una variante de UNet diseñada para mejorar la segmentación de imágenes biomédicas mediante la incorporación de mecanismos de atención. Propuesta por Oktay, Schlemper y Folgoc [48], su objetivo es ayudar al modelo a centrarse en las regiones anatómicas más relevantes, algo especialmente útil cuando las estructuras a segmentar son pequeñas, varían de forma o están rodeadas de elementos visuales complejos.

A nivel arquitectónico, AttentionUNet mantiene la topología en forma de "U" de UNet, con un encoder que comprime la información y un decoder que la reconstruye, conectados entre sí mediante *skip connections*. La diferencia está en que estas conexiones incorporan bloques de atención o Attention Gates (AG), que actúan como filtros capaces de reducir el ruido y destacar las características más útiles para la tarea de segmentación (figura 2.16). De este modo, el modelo puede centrarse en las zonas anatómicas importantes, sin que ello implique un aumento del coste computacional.

Cada bloque de atención, inspirado en técnicas utilizadas en modelos de lenguaje y visión artificial, recibe dos entradas: el mapa de características generado por el encoder y una señal contextual del decoder. A través de una serie de operaciones convolucionales, normalización y una activación *sigmoid*, el bloque genera un mapa de atención que mo-

dula la información que se transfiere. Esto permite que la red aprenda a identificar de forma dinámica las regiones más relevantes en cada fase del proceso de decodificación [49].

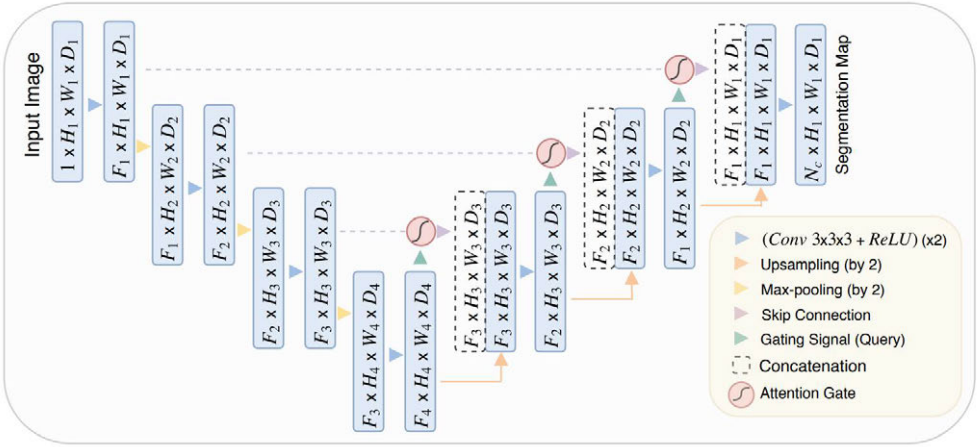


Figura 2.16. Arquitectura Attention U-Net [50]

La incorporación de módulos de atención en AttentionUNet mejora su capacidad para segmentar estructuras pequeñas o de bajo contraste, ya que permite al modelo distinguir con mayor precisión entre las regiones anatómicas de interés y el fondo. Al conservar la arquitectura de UNet, también mantiene su eficiencia computacional, lo que facilita su entrenamiento de extremo a extremo sin necesidad de modificar la función de pérdida ni el esquema de optimización. Esto lo convierte en una opción adecuada para entornos clínicos donde los recursos son limitados o se requiere procesamiento en tiempo real.

Aunque originalmente fue concebida para trabajar con imágenes biomédicas bidimensionales, su diseño puede adaptarse al análisis de datos volumétricos. Para ello, basta con reemplazar las operaciones 2D por sus equivalentes en 3D, lo que permite al modelo capturar el contexto espacial de los volúmenes.

SegResNet

SegResNet es un modelo de segmentación diseñado para trabajar con imágenes médicas tridimensionales de alta resolución. Su objetivo es ofrecer una solución robusta y eficiente capaz de abordar desafíos propios del entorno clínico, como la escasez de da-

tos o la presencia de ruido. Para ello, combina la estructura en forma de "U" de UNet con bloques residuales de redes ResNet para mejorar la estabilidad durante el entrenamiento y la capacidad de generalización del modelo.

Su arquitectura se organiza en torno a bloques convolucionales residuales, presentes tanto en el encoder como en el decoder. En el encoder, se utilizan convoluciones 3D, Group Normalization (GN) y funciones de activación ReLU para construir representaciones jerárquicas cada vez más abstractas. A diferencia de UNet, que emplea bloques convolucionales tradicionales, SegResNet incorpora conexiones residuales que suman la entrada y la salida de cada bloque. Esto facilita el paso de los gradientes durante el entrenamiento y reduce el riesgo de que se pierda información.

En el decoder, el modelo utiliza operaciones de *upsampling* seguidas de convoluciones 3D para recuperar la resolución espacial original. Al igual que en UNet, se conservan las *skip connections* entre las capas del encoder y el decoder para preservar la información que podría haberse perdido en el proceso de compresión (figura 2.17). Esta estrategia permite al modelo reconstruir con mayor precisión los detalles anatómicos, incluso cuando la calidad de la imagen es baja o hay ruido [51].

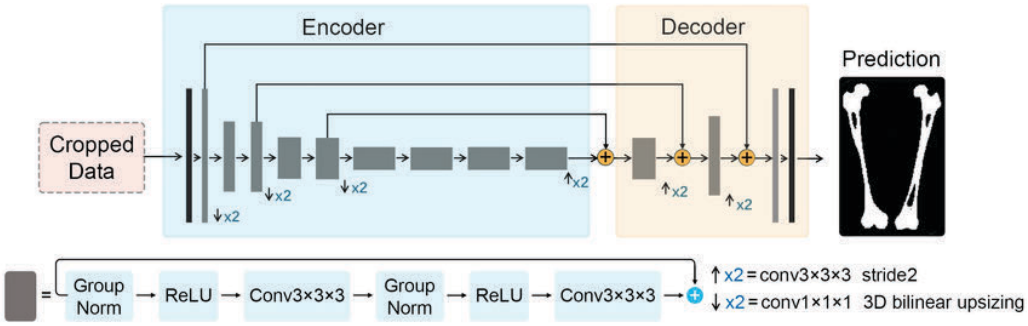


Figura 2.17. Arquitectura SegResNet [52]

SegResNet destaca por su eficiencia computacional. Su diseño optimiza el uso de memoria y reduce la complejidad del modelo, sin sacrificar la precisión de las predicciones. Por eso, resulta especialmente útil en entornos clínicos donde los recursos suelen ser limitados.

Desde su concepción, SegResNet ha sido diseñado para trabajar directamente con volúmenes tridimensionales. Opera de forma nativa sobre datos 3D, lo que le permite modelar relaciones espaciales a lo largo de los tres ejes, sin necesidad de convertir los volúmenes en cortes bidimensionales.

2.3. Trabajos Relacionados

En los últimos años, la segmentación de imágenes médicas ha avanzado gracias al desarrollo de arquitecturas especializadas. Entre ellas, UNet se ha consolidado como una de las más influyentes y utilizadas en este ámbito. Esta red convolucional en forma de "U" combina la extracción de características con la preservación de información espacial. Su eficacia, simplicidad y flexibilidad han sentado las bases para arquitecturas posteriores que han buscado superar sus limitaciones e incorporar nuevas estrategias de representación y aprendizaje.

A partir de esta arquitectura, han surgido modelos que exploran mecanismos como la atención, el ajuste dinámico de hiperparámetros o el uso de bloques residuales. A continuación, se presentan las publicaciones en las que se introdujeron formalmente estas arquitecturas:

- ***U-Net: Convolutional networks for biomedical image segmentation*, Olaf Ronneberger *et al.* (2015):** [43] **:** Presenta una arquitectura simétrica tipo encoder-decoder con *skip connections*, diseñada específicamente para la segmentación médica. Supuso un avance en tareas de segmentación con pocos datos etiquetados.
- ***nnU-Net: a self-configuring method for deep learning-based biomedical image segmentation*, Fabian Isensee *et al.* (2021)** [46] **:** Parte del *framework* nnU-Net, DynUNet automatiza la configuración del modelo, adaptándose a las características del conjunto de datos sin necesidad de intervenir manualmente.
- ***Attention u-net: Learning where to look for the pancreas*, Ozan Oktay *et al.* (2018)** [48] **:** Introduce mecanismos de atención que dirigen la atención del modelo hacia las regiones anatómicamente relevantes de la imagen, lo que resulta especialmente útil cuando las estructuras de interés son pequeñas o irregulares.
- ***Deep residual learning for image recognition*, Kaiming He *et al.* (2016)** [51] **:** Base del modelo SegResNet, este enfoque introduce bloques residuales que permiten entrenar redes más profundas y optimizar la propagación del gradiente.

La aparición de estas arquitecturas refleja una evolución constante en la búsqueda de modelos más precisos y robustos para la segmentación biomédica. Sin embargo, dado que cada modelo utiliza diferentes mecanismos para procesar y representar la informa-

ción espacial, su rendimiento puede variar en función del tipo de estructura anatómica estudiada o la modalidad de imagen utilizada.

El objetivo de este trabajo es comparar de forma empírica el rendimiento de estas arquitecturas en la segmentación de la aurícula izquierda del corazón en resonancias magnéticas cardíacas. Esta estructura, clave en el diagnóstico de patologías cardiovasculares, plantea un desafío clínico debido a su morfología variable y su proximidad a otras cavidades del corazón. Evaluar cómo se comportan estos modelos ayudará a identificar cuáles son los más adecuados para futuras aplicaciones clínicas asistidas por inteligencia artificial.

3. Metodología

En esta sección se presentan las herramientas y tecnologías utilizadas en el proyecto. Se describe el entorno de trabajo, los lenguajes y *frameworks* usados, así como las razones detrás de estas decisiones. El objetivo es proporcionar una visión general de los recursos que han hecho posible la implementación y entrenamiento de los modelos de segmentación.

3.1. Entorno de Desarrollo

El entrenamiento de redes neuronales para la segmentación de imágenes es una tarea altamente exigente en términos de capacidad de cómputo y almacenamiento. Ante las limitaciones del *hardware* local, sin acceso a unidades GPU y con espacio de disco reducido, se optó por trasladar el entorno de desarrollo a la nube, una alternativa que permite sortear estos obstáculos y trabajar con arquitecturas complejas y grandes volúmenes de datos de manera más eficiente.

Por esta razón, se eligió utilizar la plataforma Kaggle (figura 3.1a) [53], que ofrece un entorno de desarrollo basado en cuadernos Jupyter con acceso gratuito a recursos de alto rendimiento. Para este proyecto, se utilizaron dos GPUs NVIDIA Tesla T4, cada una con 16 GB de memoria, lo que proporcionó la capacidad de procesamiento necesaria para entrenar redes convolucionales tridimensionales de forma eficaz.

El lenguaje de programación utilizado ha sido Python (figura 3.1b) [54], ampliamente adoptado en el ámbito de la inteligencia artificial por su flexibilidad, claridad sintáctica y extenso ecosistema de librerías especializadas en aprendizaje automático y procesamiento de imágenes. Además, su comunidad activa y su abundante documentación la consolidan como una herramienta de referencia en proyectos de investigación en este campo.

En cuanto al framework utilizado, se ha trabajado con PyTorch (figura 3.1c) [55], una de las herramientas más consolidadas en el campo del *Deep Learning*. Su naturaleza dinámi-

ca y su diseño modular facilitan la implementación de modelos complejos y adaptables. Sobre esta base, se utilizó Medical Open Network for AI (MONAI) (figura 3.1d) [56], una extensión desarrollada específicamente para aplicaciones médicas. MONAI proporciona herramientas orientadas a los desafíos de este ámbito, como el preprocesamiento de volúmenes tridimensionales, la carga eficiente de datos clínicos, la integración de arquitecturas especializadas y la evaluación de modelos utilizando métricas relevantes en el ámbito sanitario.

(a) Kaggle (b) Python (c) PyTorch (d) MONAI

Figura 3.1. Herramientas de Desarrollo [53] [54] [55] [56]

El uso conjunto de estas herramientas ha permitido realizar experimentos de segmentación médica en 3D de manera eficiente y robusta, superando las limitaciones del *hardware* local y sin necesidad de recurrir a infraestructuras complejas o costosas de mantener.

3.2. Descripción del Dataset

Para el entrenamiento y evaluación de los modelos de segmentación se ha utilizado el dataset **Task02_Heart** del Medical Segmentation Decathlon (MSD) [57], una iniciativa internacional orientada a la comparación de algoritmos de segmentación automática en imágenes médicas. Esta tarea se centra en la segmentación de la aurícula izquierda del corazón a partir de volúmenes tridimensionales obtenidos mediante resonancia magnética. El conjunto de datos fue proporcionado por el King's College London y está disponible bajo licencia CC-BY-SA 4.0 [58], lo que permite su libre utilización siempre que se cite adecuadamente la fuente.

El objetivo de este conjunto de datos es la segmentación precisa de la aurícula izquierda,

una estructura clave en la planificación de procedimientos de ablación en pacientes con fibrilación auricular. Una segmentación fiable contribuye a una mejor comprensión de la anatomía individual del paciente, lo que puede traducirse en procedimientos más eficaces y seguros.

El conjunto se compone de 20 volúmenes de entrenamiento y 10 volúmenes de prueba. Cada volumen de entrenamiento incluye la imagen original y su correspondiente máscara de segmentación, que distingue entre dos clases: el fondo (clase 0) y la aurícula izquierda (clase 1). Las imágenes se almacenan en la carpeta `imagesTr` y las máscaras en `labelsTr`, mientras que los casos de prueba, sin anotaciones, se encuentran en `imagesTs`, simulando así un entorno de evaluación en condiciones reales. Todos los archivos están en formato NIfTI (.nii.gz), un estándar ampliamente utilizado en imagen médica que permite almacenar volúmenes 3D junto con metadatos espaciales.

Una de las grandes ventajas de este conjunto de datos radica en la calidad de sus anotaciones y en la homogeneidad de la modalidad de adquisición, aspectos que facilitan el entrenamiento de modelos supervisados tridimensionales. Sin embargo, el número limitado de muestras plantea retos importantes, especialmente en lo relativo al *overfitting*. Por esta razón, a lo largo del proyecto se han aplicado técnicas de *data augmentation* y estrategias de regularización para mitigar este problema y mejorar la capacidad de generalización de los modelos entrenados.

3.3. Pipeline de Procesamiento y Entrenamiento

A continuación, se describe el *pipeline* de procesamiento y entrenamiento diseñado para la tarea de segmentación correspondiente al **Task02_Heart** del MSD [57]. Esta sección detalla cada etapa del proceso, desde la organización inicial de los datos hasta la estrategia seguida para entrenar y evaluar los modelos. El objetivo es establecer una estructura clara, modular y reproducible que permita integrar distintas arquitecturas de segmentación, mantener la coherencia entre experimentos y optimizar el uso de recursos computacionales.

3.3.1. Carga y División de los Datos

Para iniciar el proceso de entrenamiento, es fundamental organizar adecuadamente los datos que se utilizarán. En este caso, se parte de una estructura de directorios donde las imágenes y sus correspondientes máscaras de segmentación se encuentran almacenadas por separado. En primer lugar, se recopilan las rutas de las imágenes y sus respectivas máscaras. Para ello, la función `get_image_and_label_paths` localiza todos los archivos con extensión `.nii` presentes en las carpetas `imagesTr` y `labelsTr`, emparejando cada imagen con su etiqueta. Estos pares imagen-etiqueta se organizan en una lista de diccionarios, lo que facilita su manipulación y la aplicación conjunta de transformaciones.

Una vez reunido los datos, la función `prepare_dataloaders` los divide en dos subconjuntos: entrenamiento y validación. En este caso, se destinan las últimas cuatro muestras al conjunto de validación, mientras que el resto se utiliza para entrenar los modelos. A partir de esta división, se construye un conjunto base (`base_train_ds`) aplicando únicamente las transformaciones esenciales. En caso de que se requiera aplicar técnicas de *data augmentation*, se genera un segundo conjunto (`augm_ds`) que incorpora tanto las transformaciones básicas como las de aumento. Posteriormente, ambos conjuntos se combinan para conformar el conjunto final de entrenamiento.

Cabe destacar que la clase `CacheDataset` de MONAI se emplea para almacenar en caché los datos ya transformados, lo que reduce significativamente los tiempos de carga durante el entrenamiento. Además, en caso de utilizar *data augmentation*, se recurre a `ConcatDataset` para unir de forma eficiente los conjuntos base y aumentado sin necesidad de duplicar el código de carga o redefinir estructuras adicionales.

Por último, se generan los `dataloaders` de entrenamiento y validación, que permiten recorrer los datos de forma eficiente durante el entrenamiento de los modelos. Estos `dataloaders` están diseñados para integrarse de manera directa en el ciclo de entrenamiento, proporcionando *batches* de datos ya transformados, listos para ser utilizados por los modelos.

3.3.2. Preprocesamiento y Aumento de Datos

El tratamiento previo de los datos es un paso fundamental en tareas de segmentación médica, ya que permite homogeneizar el conjunto de volúmenes y facilitar el entrena-

miento del modelo. El preprocesamiento de los datos se lleva a cabo mediante la función `get_transforms`, que define dos conjuntos de transformaciones: uno básico, aplicado en todos los casos, y otro adicional, destinado al aumento de datos para enriquecer el conjunto de entrenamiento.

Las transformaciones básicas tienen como objetivo preparar y estandarizar los volúmenes antes del entrenamiento. Este conjunto incluye la carga de los archivos (`LoadImaged`), la conversión al formato requerido por el modelo asegurando que el canal sea la primera dimensión (`EnsureChannelFirstd`), y el recorte automático del volumen para eliminar regiones sin información relevante (`CropForegroundd`). A continuación, se alinea cada volumen con el sistema de coordenadas estándar RAS mediante la transformación `Orientationd`, y se homogeniza la resolución espacial a través de `Spacingd`, asegurando un tamaño uniforme de los vóxeles. Finalmente, se aplica un relleno (padding) para que las dimensiones del volumen resultante sean múltiplos de 16, para garantizar la compatibilidad con arquitecturas convolucionales profundas, que suelen reducir la resolución espacial en potencias de dos.

Las transformaciones de *data augmentation* se aplican de forma aleatoria durante el entrenamiento con el fin de mejorar la capacidad de generalización del modelo. Entre las transfromaciones utilizadas se incluyen desplazamientos aleatorios (`RandAffined`), rotaciones (`RandRotated`) y la adición de ruido gaussiano (`RandGaussianNoised`). Estas variaciones simulan distintos escenarios anatómicos o condiciones de adquisición, permitiendo que el modelo se adapte mejor a datos no vistos durante la fase de entrenamiento.

Además, cabe destacar que, en el proceso de preparación de los datos, estas transformaciones se combinan mediante la clase `Compose`, lo que permite aplicarlas de manera secuencial sobre las mismas claves (`image` y `label`). En particular, cuando se activa el aumento de datos, las transformaciones básicas y las de aumento se concatenan en una misma cadena, generando un conjunto aumentado adicional que se suma al conjunto base, duplicando así el número de muestras vistas por el modelo durante el entrenamiento. Esta estrategia, implementada a través de la clase `ConcatDataset`, contribuye a mejorar la robustez del modelo sin necesidad de disponer de más datos reales.

3.3.3. Estrategia de Entrenamiento

Durante el entrenamiento de los modelos de segmentación 3D, se adopta una metodología estructurada y coherente, fundamentada en prácticas consolidadas dentro del aprendizaje profundo aplicado a imágenes médicas. Esta estrategia garantiza la reproducibilidad de los resultados y permite una evaluación comparativa rigurosa entre diferentes arquitecturas.

Para entrenar los diferentes modelos de segmentación, se sigue un bucle de entrenamiento estándar por épocas (*epochs*). En cada una de ellas, el modelo recorre el conjunto de entrenamiento, actualizando sus pesos en función del error cometido. Al finalizar cada *epoch*, se evalúa su rendimiento sobre un conjunto de validación. Este enfoque permite comparar los modelos bajo condiciones homogéneas y controladas.

Como función de pérdida se utiliza DiceCELoss [59], una combinación de Dice Loss y Cross Entropy Loss (ecuación 2.5). Esta elección es especialmente útil en segmentación médica, donde las clases suelen estar desbalanceadas. La Dice Loss mide la coincidencia entre la segmentación producida por el modelo y la referencia anotada [60], poniendo énfasis en la forma y localización de las regiones. Su fórmula es la siguiente:

$$\text{Dice Loss} = 1 - \frac{2\sum_i \hat{y}_i y_i + \epsilon}{\sum_i \hat{y}_i + \sum_i y_i + \epsilon} \tag{3.1}$$

donde $\hat{y}$ y y representan, respectivamente, los valores predichos y reales de cada vóxel y ϵ es un valor pequeño para mantener la estabilidad numérica. Por su parte, la Cross Entropy Loss penaliza los errores de clasificación vóxel a vóxel [61], reforzando el aprendizaje a nivel local. Al combinar ambas funciones, se obtiene una penalización más completa que mejora la calidad de la segmentación tanto global como local.

Como optimizador se ha utilizado AdamW, una variante del algoritmo Adam que incorpora una estrategia más efectiva de regularización mediante *weight decay* [62] [63]. Esta técnica contribuye a reducir el riesgo de *overfitting* al penalizar valores de peso excesivamente grandes. AdamW adapta de forma automática la tasa de aprendizaje para cada parámetro utilizando promedios móviles del gradiente y de su cuadrado. Su fórmula de actualización es:

$$\theta_{t+1} = \theta_t - \eta \left(\frac{m_t}{\sqrt{v_t} + \epsilon} + \lambda \theta_t \right) \tag{3.2}$$

donde m_t y v_t son los promedios móviles del gradiente y su cuadrado, λ es el coeficiente

de regularización, y η es la tasa de aprendizaje.

A lo largo del entrenamiento, después de cada época, se realiza una validación en la que se calcula la pérdida y se evalúa la métrica de Dice. Esta se calcula tras aplicar una etapa de post-procesamiento que discretiza las predicciones del modelo mediante la función `AsDiscrete`, convirtiéndolas en mapas binarios con `argmax` y codificación *one-hot*. Este procedimiento garantiza una comparación coherente con las máscaras anotadas, que también se transforman de forma equivalente. Si se detecta una mejora, el modelo correspondiente se guarda como el mejor hasta ese momento.

Para evitar entrenar más de lo necesario, se incorpora una lógica de parada temprana mediante una clase llamada `EarlyStopper`. Esta herramienta monitoriza la pérdida en validación y detiene el entrenamiento si no se detectan mejoras significativas tras un número determinado de *epochs* (parámetro *patience*). Su funcionamiento es simple: si la pérdida mejora más allá de una cierta diferencia mínima (`min_delta`), se reinicia el contador. Si no lo hace, el contador avanza, y al alcanzar el límite fijado, se interrumpe el proceso de entrenamiento.

Además de los mecanismos de control del entrenamiento, se registran y almacenan tanto la pérdida media por época como los valores de la métrica de validación, lo que permite trazar curvas de aprendizaje e identificar patrones de sobreajuste o convergencia. Cuando se alcanza un nuevo máximo en la métrica de validación, el modelo se guarda automáticamente en disco, asegurando la conservación del mejor estado observado.

En conjunto, esta estrategia permite identificar el modelo más eficaz sin incurrir en *overfitting*, combinando evaluaciones periódicas sobre validación con funciones de pérdida adaptadas al problema, y técnicas modernas de optimización y regularización.

3.4. Modelos Implementados

En esta sección se detallan las configuraciones utilizadas para cada una de las arquitecturas implementadas. Se describen los parámetros que se han definido al instanciar cada modelo, con el objetivo de documentar las decisiones de diseño y facilitar la reproducibilidad de los experimentos.

3.4.1. UNet

UNet se configura de la siguiente manera:

- **spatial_dims=3:** Número de dimensiones espaciales. Se trabaja con datos tridimensionales.
- **in_channels=1:** Número de canales de entrada. Se procesan imágenes en escala de grises.
- **out_channels=2:** Número de canales de salida. Equivale al número de clases.
- **channels=(16, 32, 64, 128, 256):** Número de filtros en cada capa del encoder y decoder.
- **strides=(2, 2, 2, 2):** Factor de *downsampling* en cada capa del encoder.
- **num_res_units=2:** Número de bloques residuales por capa.
- **dropout=0.2:** Probabilidad de aplicar *dropout*.
- **norm=Norm.BATCH:** Tipo de normalización utilizada en las capas convolucionales.

UNet opera sobre volúmenes tridimensionales (`spatial_dims=3`) y ha sido ajustado para recibir un solo canal de entrada (`in_channels=1`), lo que es habitual en imágenes médicas. La salida consta de dos canales (`out_channels=2`), representando dos clases distintas en el proceso de segmentación.

El número de filtros por capa en la red se define con la tupla `channels=(16, 32, 64, 128, 256)`. Esto implica que en cada etapa del encoder se duplican los canales, permitiendo a la red capturar características de mayor complejidad a medida que se profundiza en la arquitectura. Las operaciones de *downsampling* entre capas se realizan mediante convoluciones de *stride* 2 (`strides=(2, 2, 2, 2)`), lo que reduce progresivamente la resolución espacial del volumen mientras incrementa la abstracción semántica.

Cada bloque convolucional incluye dos unidades residuales (`num_res_units=2`), lo que facilita el entrenamiento de redes profundas al permitir el paso directo de la información a través de las *skip connections*. Este diseño no solo favorece una mejor propagación del gradiente, sino que también preserva detalles relevantes de la imagen de

entrada. Asimismo, se incorpora una regularización mediante *dropout* con una probabilidad del 20 % (`dropout=0.2`), contribuyendo a mitigar el *overfitting* durante el entrenamiento. La normalización utilizada es *Batch Normalization* (`norm=Norm.BATCH`), lo que estabiliza y acelera el proceso de optimización.

En resumen, esta configuración de UNet ofrece un equilibrio entre profundidad, capacidad de generalización y eficiencia computacional, resultando adecuada para afrontar tareas complejas de segmentación 3D en entornos clínicos.

Detalles Arquitectónicos de UNet

La arquitectura UNet utilizada en este trabajo (figura 3.2, listado A.1) implementa bloques residuales y técnicas de normalización para mejorar la estabilidad y el rendimiento del entrenamiento. La red sigue el patrón de encoder-decoder característico de UNet, compuesto por múltiples niveles de profundidad y *skip connections*. Este diseño favorece tanto el flujo de gradientes como la recuperación precisa de la información espacial.

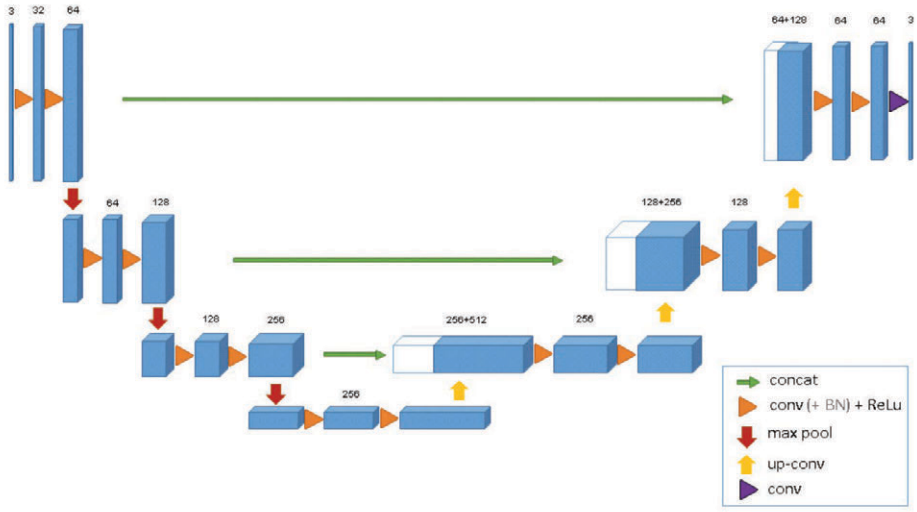


Figura 3.2. Arquitectura U-Net 3D [45]

Los bloques principales de la arquitectura se organizan en tres componentes: encoder, decoder y *skip connections*. Cada bloque se construye a partir de unidades residuales (`ResidualUnit`) que constan de dos convoluciones 3D consecutivas, cada una seguida de una normalización por lotes (`BatchNorm3d`), una activación no lineal de tipo

PReLU y una capa de *dropout* con probabilidad $p = 0{,}2$. La presencia de una rama residual que suma directamente la entrada al resultado procesado facilita el aprendizaje de funciones identidad, mitigando así el problema del desvanecimiento del gradiente en redes profundas.

El encoder está compuesto por una serie de bloques `ResidualUnit` que aplican convoluciones con *stride*=2 para reducir la resolución espacial del volumen de entrada, al mismo tiempo que se incrementa progresivamente la profundidad de canal. Las transformaciones siguen el siguiente patrón: 1→16, 16→32, 32→64, 64→128 y finalmente 128→256 canales, utilizando en cada caso *kernels* de 3×3×3 con *padding*=1 para preservar la alineación espacial.

El decoder revierte este proceso mediante bloques `ConvTranspose3d`, que realizan operaciones de *upsampling* para recuperar la resolución espacial perdida. Tras cada *upsampling*, se añade un bloque `ResidualUnit` que permite refinar la salida. En este camino de reconstrucción, los canales se decodifican de forma inversa: 256→128, 128→64, 64→32, 32→16, 16→2, adaptando así el volumen de salida al número de clases objetivo.

Las conexiones de salto (`SkipConnections`) desempeñan un papel fundamental en esta arquitectura, ya que permiten transferir directamente características de baja y media frecuencia desde el encoder al decoder. Estas conexiones combinan las salidas del encoder con las entradas correspondientes del decoder mediante bloques residuales o secuencias de estos, asegurando la preservación de detalles relevantes a distintas escalas.

La última capa consiste en una transpuesta de convolución (`ConvTranspose3d`) que ajusta el número de canales de salida a dos (aurícula izquierda y fondo), seguida de un bloque `ResidualUnit` que refina aún más la predicción final.

Finalmente, en cuanto a los componentes adicionales, la activación utilizada es PReLU, una variante paramétrica de ReLU que permite adaptarse dinámicamente a los datos durante el entrenamiento, ofreciendo así mayor flexibilidad y capacidad de modelado. Además, todas las capas convolucionales se normalizan con `BatchNorm3d`, lo que contribuye a estabilizar y acelerar el proceso de optimización. La regularización mediante *dropout* está integrada en todos los bloques, reforzando la generalización del modelo y reduciendo el riesgo de *overfitting*.

3.4.2. DynUNet

DynUNet se configura de la siguiente manera:

- **spatial_dims=3:** Número de dimensiones espaciales. Se trabaja con datos tridimensionales.
- **in_channels=1:** Número de canales de entrada. Se procesan imágenes en escala de grises.
- **out_channels=2:** Número de canales de salida. Equivale al número de clases.
- **kernel_size=[3, 3, 3, 3]:** Tamaño de los *kernels* utilizados en cada capa de la red.
- **strides=[1, 2, 2, 2]:** Factor de *downsampling* en cada capa del encoder.
- **upsample_kernel_size=[2, 2, 2]:** Tamaño de los *kernels* utilizados en el decoder.
- **filters=[16, 32, 64, 128]:** Número de filtros en cada capa del encoder y decoder.
- **dropout=0.2:** Probabilidad de aplicar *dropout*.

DynUNet ha sido definido para operar sobre datos tridimensionales (`spatial_dims=3`). El modelo recibe imágenes con un único canal de entrada (`in_channels=1`) y genera mapas de salida con dos canales (`out_channels=2`), que corresponden a las clases que se desean segmentar.

La estructura interna de la red se basa en una jerarquía de bloques convolucionales cuyas características están definidas explícitamente. En cada nivel, se emplean filtros con *kernels* de tamaño `[3, 3, 3, 3]`, lo que garantiza una cobertura espacial suficiente en cada etapa de la red. Las operaciones de *downsampling* se realizan mediante *strides* definidos como `[1, 2, 2, 2]`, lo que implica que la primera capa mantiene la resolución original del volumen mientras que las siguientes van reduciéndola progresivamente, permitiendo así una representación jerárquica y eficiente de las características espaciales.

En cuanto al número de filtros por capa, la red sigue una progresión ascendente con `[16, 32, 64, 128]`, lo que asegura una capacidad creciente para capturar patrones complejos conforme se profundiza en la red. Durante el proceso de reconstrucción en el decoder, se utilizan *kernels* de *upsampling* de tamaño `[2, 2, 2]`, lo que permite

recuperar gradualmente la resolución espacial original del volumen, combinando eficazmente la información semántica con los detalles estructurales conservados a través de las *skip connections*.

Además, se ha incorporado una tasa de *dropout* del 20 % (`dropout=0.2`) como mecanismo de regularización, lo que contribuye a reducir el riesgo de *overfitting*, especialmente relevante en escenarios con datos limitados o clases desbalanceadas.

En conjunto, DynUNet ofrece una arquitectura compacta pero potente, adaptada para equilibrar profundidad, precisión espacial y eficiencia. Gracias a su diseño dinámico, facilita la experimentación y el ajuste fino (*fine tuning*), lo que resulta especialmente útil en tareas exigentes como la segmentación 3D de imágenes médicas.

Detalles Arquitectónicos de DynUNet

DynUNet (figura 3.3, listado A.2) implementa una arquitectura U-Net tridimensional dinámica, cuya disposición simétrica y jerárquica facilita la segmentación precisa de volúmenes médicos complejos. A continuación, se describe la composición funcional de cada uno de sus bloques.

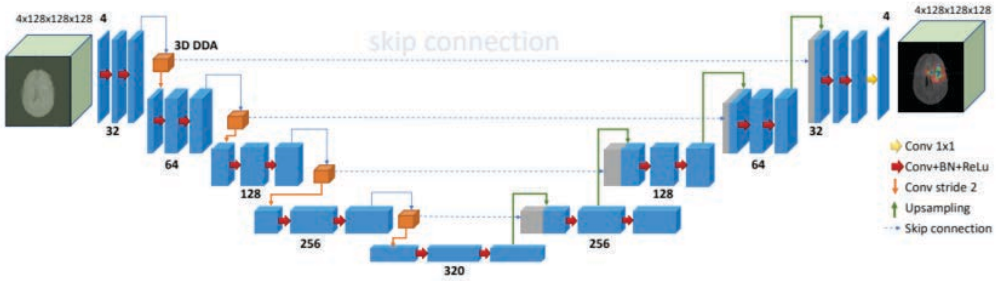


Figura 3.3. Arquitectura DynU-Net 3D [47]

El bloque de entrada (`Input Block`) se encarga de procesar los volúmenes tridimensionales iniciales mediante dos convoluciones 3D consecutivas que transforman los canales de entrada de 1 a 16. Este bloque incorpora además una normalización por instancias (`InstanceNorm3d`) seguida de una función de activación LeakyReLU, lo que favorece una activación más robusta frente a valores negativos. Para mejorar la generalización del modelo, se aplica también una capa de *dropout* con una probabilidad de 0.2.

El encoder (`Downsampling Path`) se compone de tres niveles jerárquicos en los que el número de canales aumenta progresivamente de 16 a 32, luego a 64, y finalmente a 128. En cada nivel, se aplica un *stride* de (2,2,2) en las convoluciones para reducir la resolución espacial, lo que permite al modelo capturar características globales. Cada uno de estos bloques de codificación incluye dos capas convolucionales 3D, seguidas de normalización por instancia, activación LeakyReLU y *dropout*.

El cuello de botella (`Bottleneck`), situado entre el encoder y el decoder, actúa como una capa de representación intermedia de alta capacidad. Este bloque transforma las características de 64 a 128 canales y comparte una estructura similar a la de los bloques del encoder, aunque orientada a una mayor abstracción de la información espacial y semántica contenida en el volumen.

En el decoder (`Upsampling Path`), el modelo hace uso de convoluciones transpuestas (`ConvTranspose3D`) para recuperar progresivamente la resolución espacial perdida. En cada nivel de este camino ascendente, se concatenan las características recuperadas con aquellas extraídas por las capas correspondientes del encoder a través de las *skip connections*. Esta combinación facilita una integración eficaz entre el contexto global y los detalles locales. Los canales se reducen de manera simétrica a como se incrementaron en el encoder: de 128 a 64, luego a 32 y finalmente a 16.

El bloque de salida (`Output Block`) consiste en una convolución 3D con un *kernel* de tamaño 1×1×1 que transforma los 16 canales finales en 2, correspondientes a las clases objetivo. Adicionalmente, se incluye una capa de *dropout* extra para reforzar la regularización en la etapa final.

Cabe destacar que las *skip connections* se implementan con bloques `DynUNetSkipLayer`, los cuales permiten transmitir características entre las capas del encoder y el decoder. Estos bloques están compuestos por convoluciones y normalización, lo que asegura una integración eficaz de las representaciones aprendidas y favorece una segmentación más precisa al preservar detalles relevantes.

3.4.3. AttentionUNet

AttentionUNet se configura de la siguiente manera:

- **spatial_dims=3:** Número de dimensiones espaciales. Se trabaja con datos tridimensionales.

- **in_channels=1:** Número de canales de entrada. Se procesan imágenes en escala de grises.
- **out_channels=2:** Número de canales de salida. Equivale al número de clases.
- **channels=(16, 32, 64, 128, 256):** Número de filtros en cada capa del encoder y decoder.
- **strides=(2, 2, 2, 2):** Factor de *downsampling* en cada capa del encoder.

AttentionUnet se ha configurado para operar sobre datos tridimensionales (`spatial_dims=3`), como es habitual en tareas de segmentación volumétrica, y se ha ajustado para recibir un único canal de entrada (`in_channels=1`). La salida consta de dos canales (`out_channels=2`), permitiendo segmentar entre dos clases, la región anatómica de interés y el fondo.

La arquitectura sigue una estructura encoder-decoder con *skip connections*, típica de las UNet, pero incorpora módulos de atención en las rutas de conexión entre las capas del encoder y las correspondientes del decoder. Estos módulos permiten al modelo ponderar la información proveniente de las capas de menor resolución según su relevancia para la reconstrucción final, descartando detalles irrelevantes o ruido, y enfocándose en las regiones más significativas del volumen.

El número de canales en cada nivel de la red se define con la tupla `channels=(16, 32, 64, 128, 256)`, lo que permite aumentar progresivamente la capacidad del modelo para representar patrones complejos a medida que se profundiza en la red. Por su parte, las operaciones de *downsampling* se realizan mediante convoluciones de *stride* 2 (`strides=(2, 2, 2, 2)`), lo que permite reducir la resolución espacial en cada etapa del encoder y extraer representaciones más abstractas del contenido.

Gracias al mecanismo de atención integrado, el modelo es capaz de recuperar de forma más precisa la información espacial durante la fase de *upsampling*, mejorando la calidad de la segmentación especialmente en estructuras pequeñas o con bordes difusos. Esta capacidad de centrarse dinámicamente en las regiones relevantes convierte a AttentionUNet en una herramienta muy eficaz cuando se requiere un alto nivel de sensibilidad y precisión.

En conjunto, esta configuración de AttentionUNet ofrece una solución robusta y refinada para la segmentación 3D de imágenes médicas, combinando la fuerza del diseño UNet con la inteligencia adaptativa de los mecanismos de atención.

Detalles Arquitectónicos de AttentionUNet

La arquitectura de AttentionUNet utilizada en este trabajo (figura 3.4, listado A.3) se basa en el uso de convoluciones tridimensionales (`Conv3d`), mecanismos de atención espacial (mediante bloques de tipo `AttentionBlock`), técnicas de normalización (`BatchNorm3d` e `InstanceNorm3d`) y capas de *upsampling* implementadas con convoluciones transpuestas (`ConvTranspose3d`). Esta combinación de componentes permite capturar tanto la información contextual global como los detalles espaciales locales necesarios para una segmentación precisa en volúmenes 3D.

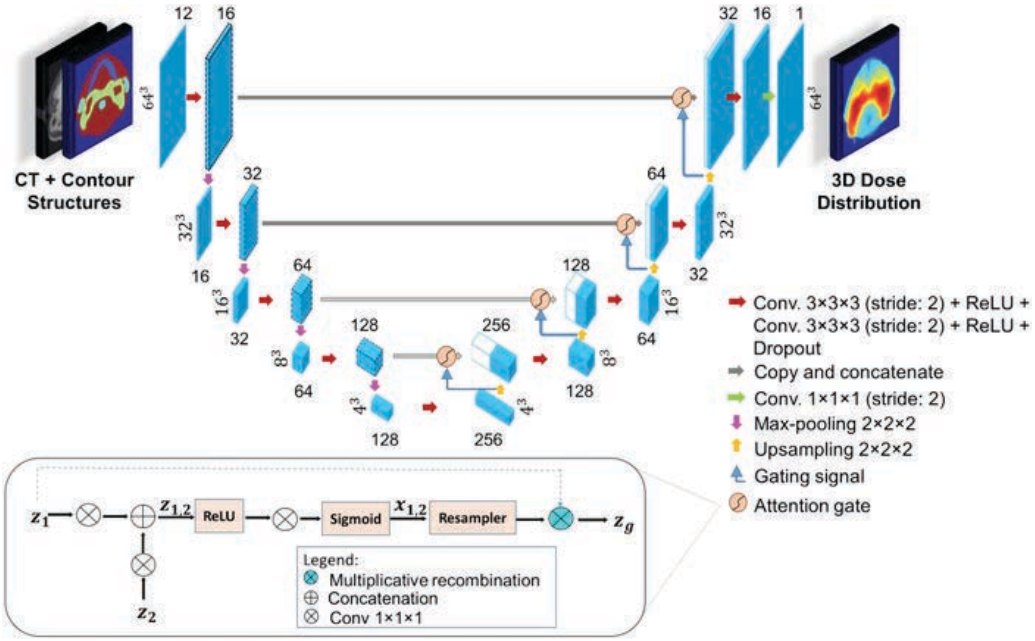


Figura 3.4. Arquitectura Attention U-Net 3D [64]

El modelo adopta una estructura tipo encoder-decoder, con *skip connections* entre capas simétricas moduladas mediante bloques de atención. Esta disposición jerárquica y recursiva se repite a lo largo de la arquitectura, lo que permite una integración progresiva de información multiescala. Cada nivel del encoder está compuesto por bloques `ConvBlock`, encargados de extraer características mediante dos capas `Conv3d` con *stride* y *padding* igual a 1, seguidas de normalización con `BatchNorm3d` y activación ReLU. En ciertos casos, se incluye además una capa de *dropout* para mitigar el *overfitting*.

Durante la fase de decodificación, se introducen los módulos `AttentionLayer`, que desempeñan un papel clave en la recuperación de información espacial relevante. Dentro de estos módulos, el `AttentionBlock` aplica atención espacial sobre las caracte-

rísticas extraídas del encoder, utilizando una señal de *gating* proveniente del decoder. La atención se calcula mediante proyecciones lineales, aplicadas sobre ambas señales y seguidas de normalización `BatchNorm3d`, una función ReLU, una proyección hacia un solo canal, y una función *softmax* final. Esta operación genera un mapa de atención que pondera las características del encoder antes de fusionarlas con las del decoder.

El *upsampling* se realiza a través de capas `UpConv` basadas en `ConvTranspose3d`, las cuales restauran la resolución espacial de forma progresiva. Posteriormente, se aplica una operación de *merge* entre la salida del *upsampling* y la señal modulada del encoder, utilizando una convolución 3D seguida de `InstanceNorm3d` y activación PReLU. Cada bloque de decodificación puede incluir submódulos adicionales de tipo `ConvBlock` + `AttentionLayer`, lo que refuerza la capacidad jerárquica y recursiva del modelo.

Cabe destacar que el modelo emplea diferentes estrategias de normalización, utilizando `BatchNorm3d` en las capas convolucionales estándar e `InstanceNorm3d` en las fusiones del decoder, lo que permite una adaptación más precisa a las distribuciones locales. Las funciones de activación combinan ReLU y PReLU en distintos niveles, optimizando así la no linealidad del flujo de información. Además, debido a la estructura jerárquica y recursiva del diseño, se implementa una supervisión profunda (*deep supervision*) implícita, que favorece una retropropagación más eficaz y una mejor discriminación espacial en distintos niveles de resolución.

3.4.4. SegResNet

SegResNet se configura de la siguiente manera:

- **spatial_dims=3:** Número de dimensiones espaciales. Se trabaja con datos tridimensionales.
- **in_channels=1:** Número de canales de entrada. Se procesan imágenes en escala de grises.
- **out_channels=2:** Número de canales de salida. Equivale al número de clases.
- **init_filters=16:** Número inicial de filtros en la primera capa convolucional.
- **blocks_down=[1, 2, 2, 4]:** Número de bloques residuales en cada capa del encoder.
- **blocks_up=[1, 1, 1]:** Número de bloques residuales en cada capa del decoder.

- **dropout=0.2:** Probabilidad de aplicar dropout.

SegResNet opera sobre datos volumétricos (`spatial_dims=3`) y está preparado para recibir un único canal de entrada (`in_channels=1`). La salida está compuesta por dos canales (`out_channels=2`), lo que permite la clasificación de cada vóxel en una de dos clases.

La arquitectura está organizada en bloques residuales tanto en el encoder como en el decoder. El número inicial de filtros se ha establecido en 16 (`init_filters=16`), y estos se van duplicando progresivamente a medida que la red profundiza. En el encoder, la red cuenta con una secuencia de bloques definida por `blocks_down=[1, 2, 2, 4]`, lo que indica que cada nivel contiene un número diferente de bloques residuales, aumentando en profundidad a medida que se reduce la resolución espacial. Esto permite una extracción de características jerárquica y rica en contexto.

Por su parte, el decoder está compuesto por `blocks_up=[1, 1, 1]`, una estructura más liviana pero suficiente para reconstruir con precisión la segmentación a partir de las representaciones aprendidas. Gracias al uso de conexiones residuales a lo largo de la red, tanto en el encoder como en el decoder, el modelo es capaz de mitigar el problema del desvanecimiento del gradiente y facilitar un entrenamiento más estable.

Además, se ha incluido una tasa de *dropout* del 20 % (`dropout_prob=0.2`) como mecanismo de regularización, ayudando a mejorar la generalización del modelo y reducir el *overfitting*, algo especialmente importante en entornos clínicos donde los conjuntos de datos suelen ser limitados.

En conjunto, SegResNet ofrece un equilibrio entre profundidad, estabilidad y capacidad de generalización, convirtiéndose en una arquitectura fiable y potente para la segmentación precisa de estructuras anatómicas en imágenes médicas 3D.

Detalles Arquitectónicos de SegResNet

La arquitectura de SegResNet utilizada en este trabajo (figura 3.5, listado A.4) se compone de varios bloques funcionales, cada uno con un propósito específico dentro del proceso de segmentación.

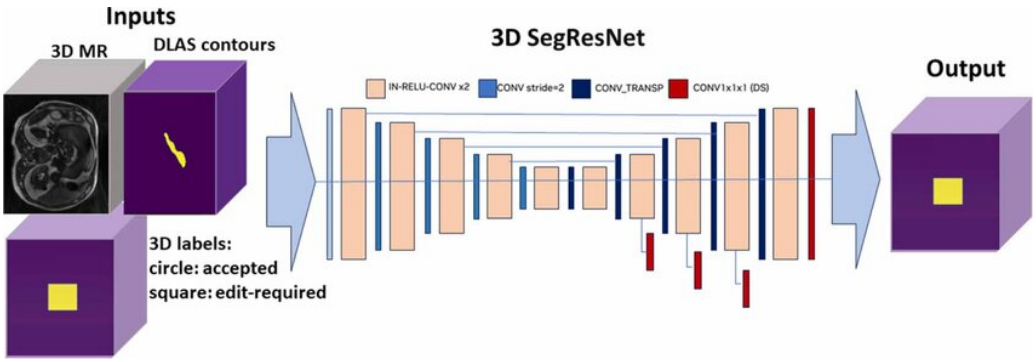


Figura 3.5. Arquitectura SegResNet 3D [65]

En primer lugar, se encuentra el bloque `convInit`, una convolución 3D inicial que transforma el canal de entrada en 16 canales, estableciendo así la base para una representación de mayor dimensionalidad. A partir de esta capa, hay una serie de bloques descendentes o `down_layers` que combinan convoluciones y reducción progresiva de la resolución espacial, mientras se incrementa la profundidad de las características aprendidas. El número de canales se duplica en cada etapa, siguiendo la secuencia: 16→32→64→128.

Cada una de estas capas descendentes contiene bloques residuales (`ResBlocks`) compuestos por convoluciones 3D sin sesgo (`bias=False`), activación ReLU y normalización de grupo (`GroupNorm`). Esta última ha sido elegida en lugar de `BatchNorm` debido a su estabilidad en escenarios donde el tamaño del *batch* es pequeño. La activación ReLU proporciona no linealidades eficientes y ayuda a mantener la estabilidad del entrenamiento. Dentro de cada bloque, las convoluciones utilizan un *padding* de 1 para conservar la resolución espacial, mientras que el uso de un *stride* de 2 en los bloques descendentes permite realizar un *downsampling* efectivo.

En el decoder, se emplean componentes de *upsampling* junto con los `up_layers`, bloques que restauran progresivamente la resolución espacial mientras reducen la profundidad de los canales: 128→64→32→16. Cada `up_layer` contiene un `ResBlock`, con una estructura análoga a la utilizada en el encoder, lo que asegura una transición coherente entre las etapas de codificación y decodificación. Esta simetría contribuye a preservar la información semántica relevante para una segmentación precisa.

La inclusión de esta estructura modular y jerárquica, apoyada en bloques residuales y operaciones cuidadosamente diseñadas, dota al modelo de una gran capacidad para capturar tanto detalles locales como contexto global, esenciales en tareas de segmentación médica tridimensional.

4. Resultados

Para evaluar el rendimiento de los modelos de segmentación, se ha utilizado el Dice Coefficient, una métrica común en el ámbito de la segmentación médica por su capacidad para medir la superposición entre la predicción del modelo y la segmentación de referencia. Este coeficiente se define matemáticamente como:

$$\text{Dice} = \frac{2|X \cap Y|}{|X| + |Y|} \tag{4.1}$$

donde X representa el conjunto de píxeles que el modelo ha clasificado como positivos, e Y corresponde a los píxeles positivos reales. Un valor de Dice próximo a 1 indica una alta concordancia entre la segmentación predicha y la real.

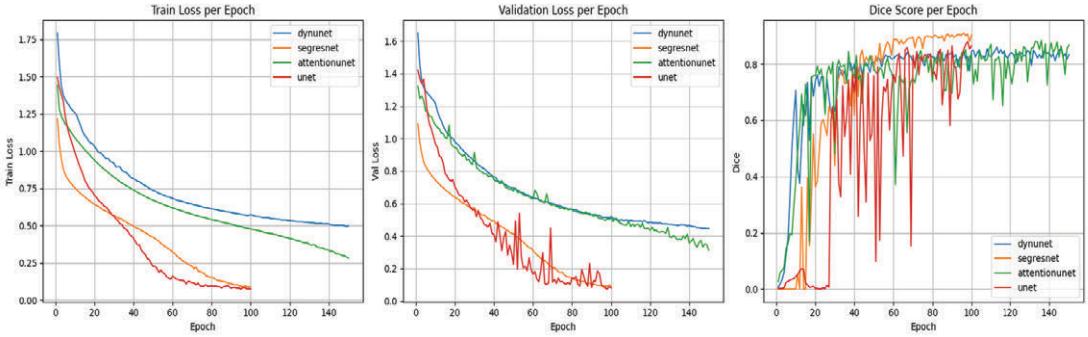


Figura 4.1. Curvas de Aprendizaje de UNet, DynUNet, AttentionUNet y SegResNet

En términos generales, todos los modelos evaluados lograron valores en torno a 0.85 en el conjunto de validación, lo que refleja un buen desempeño global (figura 4.1). No obstante, al analizar en mayor detalle tanto las métricas como los comportamientos durante el entrenamiento y la evaluación en el conjunto de prueba, se observaron diferencias relevantes entre los distintos enfoques arquitectónicos. Se han comparado las segmentaciones generadas por los distintos modelos sobre una misma muestra tanto del conjunto de validación como del conjunto de prueba (anexo B), con el fin de observar las diferencias en sus predicciones:

- **UNet**, a pesar de su arquitectura sencilla, obtuvo resultados notables, con un Dice de 0.8643 y una *Validation Loss* de 0.0818. Fue el modelo que más rápido redujo su pérdida, aunque presentó cierta inestabilidad durante el entrenamiento, reflejada en las oscilaciones del valor de Dice. Esta variabilidad puede deberse a su limitada capacidad de representación en comparación con arquitecturas más profundas o especializadas. No obstante, en términos cualitativos, UNet logró delinear correctamente la aurícula izquierda en la mayoría de los casos, tanto en el conjunto de validación (figura B.1) como en el de prueba (figura B.2).

- **DynUNet** presentó los peores resultados, con un Dice de 0.8326, y también las mayores pérdidas durante el entrenamiento. No obstante, mostró una evolución estable, lo que indica que, con un mayor número de *epochs*, podría mejorar considerablemente sus resultados. En el conjunto de validación segmentó correctamente la aurícula izquierda (figura B.3), y también lo hizo en el conjunto de prueba (figura B.4); sin embargo, en algunos casos detectó estructuras fuera de la región de interés, lo que podría estar asociado a una generalización insuficiente, posiblemente derivada de un entrenamiento incompleto o de una sensibilidad elevada a estructuras anatómicas cercanas.

- **Attention UNet** alcanzó un Dice de 0.8664, aunque mantuvo valores de pérdida relativamente altos (*Train Loss* de 0.2840 y *Validation Loss* de 0.3131). Su evolución fluctuó, lo que sugiere que requeriría más entrenamiento para consolidar su rendimiento. Al igual que DynUNet, este modelo presentó casos en los que, si bien segmentaba correctamente la aurícula izquierda tanto en el conjunto de validación (figura B.5) como en el de prueba (figura B.6), en este último, incluía también regiones ajenas al corazón en sus predicciones. Esto podría deberse a la incorporación de mecanismos de atención que, si bien permiten focalizar regiones relevantes, también podrían amplificar señales ruidosas si no han sido adecuadamente entrenados.

- **SegResNet** fue el modelo con mejor rendimiento cuantitativo, alcanzando un Dice de 0.9023 con una *Train Loss* de 0.0890 y una *Validation Loss* de 0.0887. Sus curvas de pérdida y precisión muestran una evolución regular y sostenida, lo que sugiere una buena capacidad de generalización. Tanto en el conjunto de validación (figura B.7) como en el de prueba (figura B.8), SegResNet segmentó la aurícula izquierda de forma precisa, lo que refuerza su fiabilidad clínica en escenarios reales. Este comportamiento podría atribuirse a la arquitectura residual que caracteriza a este modelo, la cual facilita una propagación más eficiente del gradiente durante el entrenamiento y mejora la capacidad para capturar características espaciales

relevantes.

Modelo	Dice Metric	Train Loss	Validation Loss
UNet	0.8643	0.0751	0.0818
DynUNet	0.8326	0.4980	0.4464
Attention UNet	0.8664	0.2840	0.3131
SegResNet	0.9023	0.0890	0.0887

Tabla 4.1. Comparativa de Métricas de los Modelos Evaluados

En cuanto a la estabilidad durante el entrenamiento, tanto SegResNet como DynUNet mostraron curvas de pérdida y métricas consistentes y bien definidas, mientras que UNet y Attention UNet fueron más irregulares. En lo que respecta a la precisión de las segmentaciones, UNet y SegResNet lograron delinear con claridad el contorno de la aurícula izquierda. Por el contrario, DynUNet y Attention UNet cometieron errores esporádicos, identificando regiones fuera del corazón, probablemente debido a un entrenamiento incompleto. Es razonable pensar que, con más tiempo de entrenamiento, estos modelos podrían refinar sus predicciones.

En resumen, los resultados sugieren que, si bien todos los modelos tienen un desempeño aceptable, SegResNet se posiciona como la opción más robusta y precisa en términos tanto cuantitativos como cualitativos. AttentionUNet y DynUNet muestran potencial, pero su desempeño actual sugiere la necesidad de entrenamientos más prolongados y posiblemente un ajuste más fino de sus hiperparámetros.

Los resultados cuantitativos de cada modelo, en términos de métrica de Dice, *Train Loss* y *Validation Loss*, se recogen en la Tabla 4.1, lo que permite comparar su rendimiento de forma más directa. Por otro lado, ejemplos de las segmentaciones realizadas por los modelos se incluyen en el anexo B, tanto para el conjunto de validación como para el conjunto de prueba, con el fin de complementar el análisis con una evaluación cualitativa.

5. Impacto del Proyecto

5.1. Impacto Social y Responsabilidad Ética

El desarrollo de herramientas basadas en inteligencia artificial para la segmentación automática de la aurícula izquierda en resonancias magnéticas cardíacas tiene un gran impacto social en la medicina, sobre todo en el diagnóstico y seguimiento de enfermedades cardiovasculares como la fibrilación auricular. La integración de este tipo de tecnologías permite mejorar de forma notable la precisión y la eficiencia del proceso diagnóstico, facilitando al personal médico especializado una evaluación más rápida, objetiva y consistente de la morfología auricular.

Este tipo de soluciones no solo contribuye a optimizar el uso de los recursos hospitalarios y a aliviar la carga de trabajo de radiólogos y cardiólogos, sino que también permite acortar los tiempos de respuesta, algo especialmente valioso en contextos con alta presión asistencial. Además, al reducir la variabilidad entre observadores que conlleva la segmentación manual, se mejora la reproducibilidad y consistencia tanto en estudios clínicos como en la toma de decisiones.

Desde una perspectiva de equidad, herramientas como las desarrolladas en este proyecto pueden ser útiles en centros con recursos limitados o con escasez de especialistas en imagen médica. Su implementación contribuiría a democratizar el acceso a tecnologías de diagnóstico avanzadas, garantizando que un mayor número de pacientes, independientemente de su ubicación geográfica o situación socio-económica, puedan beneficiarse de diagnósticos más precisos y tratamientos personalizados.

No obstante, la adopción de modelos de inteligencia artificial en el ámbito médico plantea retos éticos importantes. La fiabilidad de las segmentaciones generadas de forma automática debe ser cuidadosamente supervisada para evitar errores que comprometan la vida del paciente. Asimismo, es fundamental identificar y mitigar posibles sesgos en los datos de entrenamiento, que podrían afectar negativamente a ciertos subgrupos de la población si no se corrigen de manera adecuada.

Por último, el cumplimiento estricto de las normativas de protección de datos, como el Reglamento General de Protección de Datos (RGPD) en Europa, es imprescindible para garantizar la privacidad de la información clínica utilizada en el desarrollo y validación de estos modelos. Promover buenas prácticas en el manejo de datos sensibles y asegurar la trazabilidad de las decisiones generadas por los algoritmos son pilares fundamentales para fomentar la confianza, la transparencia y la aceptación ética de estas tecnologías en el entorno sanitario.

5.2. Impacto Económico y Empresarial

Desde el punto de vista económico, la implementación de tecnologías de segmentación automática basadas en aprendizaje profundo, como las desarrolladas en este PFG, representa una oportunidad perfecta para mejorar tanto la eficiencia del sistema sanitario como el crecimiento del sector industrial vinculado a la tecnología médica.

Por un lado, automatizar el proceso de segmentación de la aurícula izquierda permite reducir significativamente los costes operativos en hospitales y centros sanitarios. Al minimizar el tiempo que los especialistas dedican al análisis de áreas de interés (creación manual de máscaras) y disminuir el riesgo de errores e inconsistencias que podrían requerir repetir estudios, se optimiza el uso de recursos humanos y técnicos. Esta mejora en la eficiencia se traduce en una atención médica más ágil y en una gestión más sostenible de los recursos sanitarios, como ya se ha comentado previamente.

Por otro lado, la integración de modelos de IA en la práctica clínica está contribuyendo al crecimiento de un ecosistema empresarial orientado al desarrollo de soluciones médicas inteligentes. Esto no solo favorece la aparición de start-ups y empresas especializadas en software médico, imagenología avanzada y herramientas de apoyo al diagnóstico, sino que también genera nuevas oportunidades de negocio y empleo en el ámbito de la salud digital. Al mismo tiempo, la creciente demanda de profesionales con conocimientos en inteligencia artificial aplicada a la medicina impulsa la formación de perfiles técnicos especializados, fortaleciendo los lazos entre el sector sanitario y el tecnológico.

En definitiva, la inteligencia artificial no solo está transformando la forma en que se presta atención médica, mejorando su calidad y eficiencia, sino que también está actuando como un motor de cambio económico, promoviendo la innovación, el emprendimiento y el desarrollo sostenible en la industria de la salud.

6. Conclusiones y Proyección a Futuro

En base a los resultados obtenidos, se puede afirmar que se han alcanzado todos los objetivos de este proyecto. El análisis comparativo de diferentes arquitecturas de segmentación ha permitido evaluar su rendimiento en la segmentación de la aurícula izquierda en imágenes por resonancia magnética. La estrategia de entrenamiento, basada en la función de pérdida DiceCELoss, el optimizador AdamW y un mecanismo de *early stopping*, ha demostrado ser clave para estabilizar el proceso de aprendizaje y obtener segmentaciones consistentes y precisas en la mayoría de los casos.

Cada uno de los modelos evaluados ha demostrado ventajas en cuanto a la velocidad de entrenamiento, la rapidez de convergencia y el rendimiento sobre el conjunto de validación. No obstante, también se han detectado ciertas limitaciones que condicionan los resultados obtenidos. Por un lado, el tamaño reducido del conjunto de datos ha dificultado el aprendizaje de patrones generalizables, sobre todo en el caso de las clases menos representadas. Por otro lado, la alta variabilidad en las imágenes ha supuesto un desafío adicional a la hora de segmentar. A esto se suman las restricciones computacionales, que han limitado la posibilidad de realizar una búsqueda exhaustiva de hiperparámetros o el entrenamiento de modelos más complejos.

A pesar de estas limitaciones, los resultados obtenidos respaldan la viabilidad de los enfoques propuestos y abren la puerta a futuras líneas de investigación:

- Una dirección interesante sería explorar técnicas de aprendizaje auto-supervisado, que han mostrado un gran potencial en entornos médicos con escasez de datos etiquetados. Esta estrategia permitiría aprovechar las imágenes no anotadas, disminuyendo la necesidad de intervención manual.
- También resultaría útil estudiar la adaptación de los modelos desarrollados a otras modalidades de imagen médica, como las tomografías computarizadas o las radiografías. Esto permitiría analizar su capacidad de generalización a dominios diferentes al utilizado durante el entrenamiento.

- Por último, un paso lógico en la evolución de este trabajo sería su validación en entornos clínicos reales. Este tipo de evaluación sería clave para valorar su utilidad práctica como herramienta de apoyo en tareas médicas complejas.

En resumen, este proyecto sienta las bases para seguir avanzando en el desarrollo de modelos de segmentación aplicados al ámbito médico. Su implementación y validación en escenarios reales podría suponer una mejora significativa en la calidad del diagnóstico y tratamiento de enfermedades, beneficiando tanto a los profesionales de la salud como a los pacientes.

A. Arquitecturas de los Modelos Implementados

Listado A.1. Arquitectura del Modelo UNet

```
UNet(
  (model): Sequential(
    (0): ResidualUnit(
      (conv): Sequential(
        (unit0): Convolution(
          (conv): Conv3d(1, 16, kernel_size=(3, 3, 3), stride=(2, 2, 2), padding=(1, 1, 1))
          (adn): ADN(
            (N): BatchNorm3d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
            (D): Dropout(p=0.2, inplace=False)
            (A): PReLU(num_parameters=1)
          )
        )
        (unit1): Convolution(
          (conv): Conv3d(16, 16, kernel_size=(3, 3, 3), stride=(1, 1, 1), padding=(1, 1, 1))
          (adn): ADN(
            (N): BatchNorm3d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
            (D): Dropout(p=0.2, inplace=False)
            (A): PReLU(num_parameters=1)
          )
        )
      )
      (residual): Conv3d(1, 16, kernel_size=(3, 3, 3), stride=(2, 2, 2), padding=(1, 1, 1))
    )
    (1): SkipConnection(
      (submodule): Sequential(
        (0): ResidualUnit(
          (conv): Sequential(
            (unit0): Convolution(
              (conv): Conv3d(16, 32, kernel_size=(3, 3, 3), stride=(2, 2, 2), padding=(1, 1, 1))
              (adn): ADN(
                (N): BatchNorm3d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
                (D): Dropout(p=0.2, inplace=False)
                (A): PReLU(num_parameters=1)
              )
            )
            (unit1): Convolution(
              (conv): Conv3d(32, 32, kernel_size=(3, 3, 3), stride=(1, 1, 1), padding=(1, 1, 1))
              (adn): ADN(
                (N): BatchNorm3d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
                (D): Dropout(p=0.2, inplace=False)
                (A): PReLU(num_parameters=1)
              )
            )
          )
          (residual): Conv3d(16, 32, kernel_size=(3, 3, 3), stride=(2, 2, 2), padding=(1, 1, 1))
        )
        (1): SkipConnection(
          (submodule): Sequential(
            (0): ResidualUnit(
              (conv): Sequential(
                (unit0): Convolution(
                  (conv): Conv3d(32, 64, kernel_size=(3, 3, 3), stride=(2, 2, 2), padding=(1, 1, 1))
                  (adn): ADN(
                    (N): BatchNorm3d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
                    (D): Dropout(p=0.2, inplace=False)
                    (A): PReLU(num_parameters=1)
                  )
                )
                (unit1): Convolution(
                  (conv): Conv3d(64, 64, kernel_size=(3, 3, 3), stride=(1, 1, 1), padding=(1, 1, 1))
                  (adn): ADN(
                    (N): BatchNorm3d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
                    (D): Dropout(p=0.2, inplace=False)
                    (A): PReLU(num_parameters=1)
                  )
                )
              )
              (residual): Conv3d(32, 64, kernel_size=(3, 3, 3), stride=(2, 2, 2), padding=(1, 1, 1))
            )
            (1): SkipConnection(
              (submodule): Sequential(
                (0): ResidualUnit(
                  (conv): Sequential(
                    (unit0): Convolution(
                      (conv): Conv3d(64, 128, kernel_size=(3, 3, 3), stride=(2, 2, 2), padding=(1, 1, 1))
```

```
              (adn): ADN(
                (N): BatchNorm3d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
                (D): Dropout(p=0.2, inplace=False)
                (A): PReLU(num_parameters=1)
              )
            )
            (unit1): Convolution(
              (conv): Conv3d(128, 128, kernel_size=(3, 3, 3), stride=(1, 1, 1), padding=(1, 1, 1))
              (adn): ADN(
                (N): BatchNorm3d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
                (D): Dropout(p=0.2, inplace=False)
                (A): PReLU(num_parameters=1)
              )
            )
          )
          (residual): Conv3d(64, 128, kernel_size=(3, 3, 3), stride=(2, 2, 2), padding=(1, 1, 1))
        )
        (1): SkipConnection(
          (submodule): ResidualUnit(
            (conv): Sequential(
              (unit0): Convolution(
                (conv): Conv3d(128, 256, kernel_size=(3, 3, 3), stride=(1, 1, 1), padding=(1, 1, 1))
                (adn): ADN(
                  (N): BatchNorm3d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
                  (D): Dropout(p=0.2, inplace=False)
                  (A): PReLU(num_parameters=1)
                )
              )
              (unit1): Convolution(
                (conv): Conv3d(256, 256, kernel_size=(3, 3, 3), stride=(1, 1, 1), padding=(1, 1, 1))
                (adn): ADN(
                  (N): BatchNorm3d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
                  (D): Dropout(p=0.2, inplace=False)
                  (A): PReLU(num_parameters=1)
                )
              )
            )
            (residual): Conv3d(128, 256, kernel_size=(1, 1, 1), stride=(1, 1, 1))
          )
        )
        (2): Sequential(
          (0): Convolution(
            (conv): ConvTranspose3d(384, 64, kernel_size=(3, 3, 3), stride=(2, 2, 2), padding=(1, 1, 1),
                output_padding=(1, 1, 1))
            (adn): ADN(
              (N): BatchNorm3d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
              (D): Dropout(p=0.2, inplace=False)
              (A): PReLU(num_parameters=1)
            )
          )
          (1): ResidualUnit(
            (conv): Sequential(
              (unit0): Convolution(
                (conv): Conv3d(64, 64, kernel_size=(3, 3, 3), stride=(1, 1, 1), padding=(1, 1, 1))
                (adn): ADN(
                  (N): BatchNorm3d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
                  (D): Dropout(p=0.2, inplace=False)
                  (A): PReLU(num_parameters=1)
                )
              )
            )
            (residual): Identity()
          )
        )
      )
    )
    (2): Sequential(
      (0): Convolution(
        (conv): ConvTranspose3d(128, 32, kernel_size=(3, 3, 3), stride=(2, 2, 2), padding=(1, 1, 1), output_padding
            =(1, 1, 1))
        (adn): ADN(
          (N): BatchNorm3d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
          (D): Dropout(p=0.2, inplace=False)
          (A): PReLU(num_parameters=1)
        )
      )
      (1): ResidualUnit(
        (conv): Sequential(
          (unit0): Convolution(
            (conv): Conv3d(32, 32, kernel_size=(3, 3, 3), stride=(1, 1, 1), padding=(1, 1, 1))
            (adn): ADN(
              (N): BatchNorm3d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
              (D): Dropout(p=0.2, inplace=False)
              (A): PReLU(num_parameters=1)
            )
          )
        )
        (residual): Identity()
      )
    )
  )
)
(2): Sequential(
  (0): Convolution(
    (conv): ConvTranspose3d(64, 16, kernel_size=(3, 3, 3), stride=(2, 2, 2), padding=(1, 1, 1), output_padding=(1, 1,
        1))
    (adn): ADN(
      (N): BatchNorm3d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (D): Dropout(p=0.2, inplace=False)
      (A): PReLU(num_parameters=1)
    )
  )
  (1): ResidualUnit(
    (conv): Sequential(
```

```
            (unit0): Convolution(
              (conv): Conv3d(16, 16, kernel_size=(3, 3, 3), stride=(1, 1, 1), padding=(1, 1, 1))
              (adn): ADN(
                (N): BatchNorm3d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
                (D): Dropout(p=0.2, inplace=False)
                (A): PReLU(num_parameters=1)
              )
            )
          )
          (residual): Identity()
        )
      )
    )
    (2): Sequential(
      (0): Convolution(
        (conv): ConvTranspose3d(32, 2, kernel_size=(3, 3, 3), stride=(2, 2, 2), padding=(1, 1, 1), output_padding=(1, 1, 1))
        (adn): ADN(
          (N): BatchNorm3d(2, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
          (D): Dropout(p=0.2, inplace=False)
          (A): PReLU(num_parameters=1)
        )
      )
      (1): ResidualUnit(
        (conv): Sequential(
          (unit0): Convolution(
            (conv): Conv3d(2, 2, kernel_size=(3, 3, 3), stride=(1, 1, 1), padding=(1, 1, 1))
          )
        )
        (residual): Identity()
      )
    )
  )
)
```

Listado A.2. Arquitectura del Modelo DynUNet

```
DynUNet(
  (input_block): UnetBasicBlock(
    (conv1): Convolution(
      (conv): Conv3d(1, 16, kernel_size=(3, 3, 3), stride=(1, 1, 1), padding=(1, 1, 1), bias=False)
      (adn): ADN(
        (D): Dropout(p=0.2, inplace=False)
      )
    )
    (conv2): Convolution(
      (conv): Conv3d(16, 16, kernel_size=(3, 3, 3), stride=(1, 1, 1), padding=(1, 1, 1), bias=False)
      (adn): ADN(
        (D): Dropout(p=0.2, inplace=False)
      )
    )
    (lrelu): LeakyReLU(negative_slope=0.01, inplace=True)
    (norm1): InstanceNorm3d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=False)
    (norm2): InstanceNorm3d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=False)
  )
  (downsamples): ModuleList(
    (0): UnetBasicBlock(
      (conv1): Convolution(
        (conv): Conv3d(16, 32, kernel_size=(3, 3, 3), stride=(2, 2, 2), padding=(1, 1, 1), bias=False)
        (adn): ADN(
          (D): Dropout(p=0.2, inplace=False)
        )
      )
      (conv2): Convolution(
        (conv): Conv3d(32, 32, kernel_size=(3, 3, 3), stride=(1, 1, 1), padding=(1, 1, 1), bias=False)
        (adn): ADN(
          (D): Dropout(p=0.2, inplace=False)
        )
      )
      (lrelu): LeakyReLU(negative_slope=0.01, inplace=True)
      (norm1): InstanceNorm3d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=False)
      (norm2): InstanceNorm3d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=False)
    )
    (1): UnetBasicBlock(
      (conv1): Convolution(
        (conv): Conv3d(32, 64, kernel_size=(3, 3, 3), stride=(2, 2, 2), padding=(1, 1, 1), bias=False)
        (adn): ADN(
          (D): Dropout(p=0.2, inplace=False)
        )
      )
      (conv2): Convolution(
        (conv): Conv3d(64, 64, kernel_size=(3, 3, 3), stride=(1, 1, 1), padding=(1, 1, 1), bias=False)
        (adn): ADN(
          (D): Dropout(p=0.2, inplace=False)
        )
      )
      (lrelu): LeakyReLU(negative_slope=0.01, inplace=True)
      (norm1): InstanceNorm3d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=False)
      (norm2): InstanceNorm3d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=False)
    )
  )
  (bottleneck): UnetBasicBlock(
    (conv1): Convolution(
      (conv): Conv3d(64, 128, kernel_size=(3, 3, 3), stride=(2, 2, 2), padding=(1, 1, 1), bias=False)
      (adn): ADN(
        (D): Dropout(p=0.2, inplace=False)
      )
```

```
    )
    (conv2): Convolution(
      (conv): Conv3d(128, 128, kernel_size=(3, 3, 3), stride=(1, 1, 1), padding=(1, 1, 1), bias=False)
      (adn): ADN(
        (D): Dropout(p=0.2, inplace=False)
      )
    )
    (lrelu): LeakyReLU(negative_slope=0.01, inplace=True)
    (norm1): InstanceNorm3d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=False)
    (norm2): InstanceNorm3d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=False)
  )
  (upsamples): ModuleList(
    (0): UnetUpBlock(
      (transp_conv): Convolution(
        (conv): ConvTranspose3d(128, 64, kernel_size=(2, 2, 2), stride=(2, 2, 2), bias=False)
        (adn): ADN(
          (D): Dropout(p=0.2, inplace=False)
        )
      )
      (conv_block): UnetBasicBlock(
        (conv1): Convolution(
          (conv): Conv3d(128, 64, kernel_size=(3, 3, 3), stride=(1, 1, 1), padding=(1, 1, 1), bias=False)
          (adn): ADN(
            (D): Dropout(p=0.2, inplace=False)
          )
        )
        (conv2): Convolution(
          (conv): Conv3d(64, 64, kernel_size=(3, 3, 3), stride=(1, 1, 1), padding=(1, 1, 1), bias=False)
          (adn): ADN(
            (D): Dropout(p=0.2, inplace=False)
          )
        )
        (lrelu): LeakyReLU(negative_slope=0.01, inplace=True)
        (norm1): InstanceNorm3d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=False)
        (norm2): InstanceNorm3d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=False)
      )
    )
    (1): UnetUpBlock(
      (transp_conv): Convolution(
        (conv): ConvTranspose3d(64, 32, kernel_size=(2, 2, 2), stride=(2, 2, 2), bias=False)
        (adn): ADN(
          (D): Dropout(p=0.2, inplace=False)
        )
      )
      (conv_block): UnetBasicBlock(
        (conv1): Convolution(
          (conv): Conv3d(64, 32, kernel_size=(3, 3, 3), stride=(1, 1, 1), padding=(1, 1, 1), bias=False)
          (adn): ADN(
            (D): Dropout(p=0.2, inplace=False)
          )
        )
        (conv2): Convolution(
          (conv): Conv3d(32, 32, kernel_size=(3, 3, 3), stride=(1, 1, 1), padding=(1, 1, 1), bias=False)
          (adn): ADN(
            (D): Dropout(p=0.2, inplace=False)
          )
        )
        (lrelu): LeakyReLU(negative_slope=0.01, inplace=True)
        (norm1): InstanceNorm3d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=False)
        (norm2): InstanceNorm3d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=False)
      )
    )
    (2): UnetUpBlock(
      (transp_conv): Convolution(
        (conv): ConvTranspose3d(32, 16, kernel_size=(2, 2, 2), stride=(2, 2, 2), bias=False)
        (adn): ADN(
          (D): Dropout(p=0.2, inplace=False)
        )
      )
      (conv_block): UnetBasicBlock(
        (conv1): Convolution(
          (conv): Conv3d(32, 16, kernel_size=(3, 3, 3), stride=(1, 1, 1), padding=(1, 1, 1), bias=False)
          (adn): ADN(
            (D): Dropout(p=0.2, inplace=False)
          )
        )
        (conv2): Convolution(
          (conv): Conv3d(16, 16, kernel_size=(3, 3, 3), stride=(1, 1, 1), padding=(1, 1, 1), bias=False)
          (adn): ADN(
            (D): Dropout(p=0.2, inplace=False)
          )
        )
        (lrelu): LeakyReLU(negative_slope=0.01, inplace=True)
        (norm1): InstanceNorm3d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=False)
        (norm2): InstanceNorm3d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=False)
      )
    )
  )
  (output_block): UnetOutBlock(
    (conv): Convolution(
      (conv): Conv3d(16, 2, kernel_size=(1, 1, 1), stride=(1, 1, 1))
      (adn): ADN(
        (D): Dropout(p=0.2, inplace=False)
      )
    )
  )
  (skip_layers): DynUNetSkipLayer(
    (downsample): UnetBasicBlock(
      (conv1): Convolution(
        (conv): Conv3d(1, 16, kernel_size=(3, 3, 3), stride=(1, 1, 1), padding=(1, 1, 1), bias=False)
        (adn): ADN(
          (D): Dropout(p=0.2, inplace=False)
        )
      )
    )
```

```
    (conv2): Convolution(
      (conv): Conv3d(16, 16, kernel_size=(3, 3, 3), stride=(1, 1, 1), padding=(1, 1, 1), bias=False)
      (adn): ADN(
        (D): Dropout(p=0.2, inplace=False)
      )
    )
    (lrelu): LeakyReLU(negative_slope=0.01, inplace=True)
    (norm1): InstanceNorm3d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=False)
    (norm2): InstanceNorm3d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=False)
  )
  (next_layer): DynUNetSkipLayer(
    (downsample): UnetBasicBlock(
      (conv1): Convolution(
        (conv): Conv3d(16, 32, kernel_size=(3, 3, 3), stride=(2, 2, 2), padding=(1, 1, 1), bias=False)
        (adn): ADN(
          (D): Dropout(p=0.2, inplace=False)
        )
      )
      (conv2): Convolution(
        (conv): Conv3d(32, 32, kernel_size=(3, 3, 3), stride=(1, 1, 1), padding=(1, 1, 1), bias=False)
        (adn): ADN(
          (D): Dropout(p=0.2, inplace=False)
        )
      )
      (lrelu): LeakyReLU(negative_slope=0.01, inplace=True)
      (norm1): InstanceNorm3d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=False)
      (norm2): InstanceNorm3d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=False)
    )
    (next_layer): DynUNetSkipLayer(
      (downsample): UnetBasicBlock(
        (conv1): Convolution(
          (conv): Conv3d(32, 64, kernel_size=(3, 3, 3), stride=(2, 2, 2), padding=(1, 1, 1), bias=False)
          (adn): ADN(
            (D): Dropout(p=0.2, inplace=False)
          )
        )
        (conv2): Convolution(
          (conv): Conv3d(64, 64, kernel_size=(3, 3, 3), stride=(1, 1, 1), padding=(1, 1, 1), bias=False)
          (adn): ADN(
            (D): Dropout(p=0.2, inplace=False)
          )
        )
        (lrelu): LeakyReLU(negative_slope=0.01, inplace=True)
        (norm1): InstanceNorm3d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=False)
        (norm2): InstanceNorm3d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=False)
      )
      (next_layer): UnetBasicBlock(
        (conv1): Convolution(
          (conv): Conv3d(64, 128, kernel_size=(3, 3, 3), stride=(2, 2, 2), padding=(1, 1, 1), bias=False)
          (adn): ADN(
            (D): Dropout(p=0.2, inplace=False)
          )
        )
        (conv2): Convolution(
          (conv): Conv3d(128, 128, kernel_size=(3, 3, 3), stride=(1, 1, 1), padding=(1, 1, 1), bias=False)
          (adn): ADN(
            (D): Dropout(p=0.2, inplace=False)
          )
        )
        (lrelu): LeakyReLU(negative_slope=0.01, inplace=True)
        (norm1): InstanceNorm3d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=False)
        (norm2): InstanceNorm3d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=False)
      )
      (upsample): UnetUpBlock(
        (transp_conv): Convolution(
          (conv): ConvTranspose3d(128, 64, kernel_size=(2, 2, 2), stride=(2, 2, 2), bias=False)
          (adn): ADN(
            (D): Dropout(p=0.2, inplace=False)
          )
        )
        (conv_block): UnetBasicBlock(
          (conv1): Convolution(
            (conv): Conv3d(128, 64, kernel_size=(3, 3, 3), stride=(1, 1, 1), padding=(1, 1, 1), bias=False)
            (adn): ADN(
              (D): Dropout(p=0.2, inplace=False)
            )
          )
          (conv2): Convolution(
            (conv): Conv3d(64, 64, kernel_size=(3, 3, 3), stride=(1, 1, 1), padding=(1, 1, 1), bias=False)
            (adn): ADN(
              (D): Dropout(p=0.2, inplace=False)
            )
          )
          (lrelu): LeakyReLU(negative_slope=0.01, inplace=True)
          (norm1): InstanceNorm3d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=False)
          (norm2): InstanceNorm3d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=False)
        )
      )
    )
    (upsample): UnetUpBlock(
      (transp_conv): Convolution(
        (conv): ConvTranspose3d(64, 32, kernel_size=(2, 2, 2), stride=(2, 2, 2), bias=False)
        (adn): ADN(
          (D): Dropout(p=0.2, inplace=False)
        )
      )
      (conv_block): UnetBasicBlock(
        (conv1): Convolution(
          (conv): Conv3d(64, 32, kernel_size=(3, 3, 3), stride=(1, 1, 1), padding=(1, 1, 1), bias=False)
          (adn): ADN(
            (D): Dropout(p=0.2, inplace=False)
          )
        )
        (conv2): Convolution(
```

```
            (conv): Conv3d(32, 32, kernel_size=(3, 3, 3), stride=(1, 1, 1), padding=(1, 1, 1), bias=False)
            (adn): ADN(
              (D): Dropout(p=0.2, inplace=False)
            )
          )
          (lrelu): LeakyReLU(negative_slope=0.01, inplace=True)
          (norm1): InstanceNorm3d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=False)
          (norm2): InstanceNorm3d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=False)
        )
      )
    )
    (upsample): UnetUpBlock(
      (transp_conv): Convolution(
        (conv): ConvTranspose3d(32, 16, kernel_size=(2, 2, 2), stride=(2, 2, 2), bias=False)
        (adn): ADN(
          (D): Dropout(p=0.2, inplace=False)
        )
      )
      (conv_block): UnetBasicBlock(
        (conv1): Convolution(
          (conv): Conv3d(32, 16, kernel_size=(3, 3, 3), stride=(1, 1, 1), padding=(1, 1, 1), bias=False)
          (adn): ADN(
            (D): Dropout(p=0.2, inplace=False)
          )
        )
        (conv2): Convolution(
          (conv): Conv3d(16, 16, kernel_size=(3, 3, 3), stride=(1, 1, 1), padding=(1, 1, 1), bias=False)
          (adn): ADN(
            (D): Dropout(p=0.2, inplace=False)
          )
        )
        (lrelu): LeakyReLU(negative_slope=0.01, inplace=True)
        (norm1): InstanceNorm3d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=False)
        (norm2): InstanceNorm3d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=False)
      )
    )
  )
)
```

Listado A.3. Arquitectura del Modelo AttentionUNet

```
AttentionUnet(
  (model): Sequential(
    (0): ConvBlock(
      (conv): Sequential(
        (0): Convolution(
          (conv): Conv3d(1, 16, kernel_size=(3, 3, 3), stride=(1, 1, 1), padding=(1, 1, 1))
          (adn): ADN(
            (N): BatchNorm3d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
            (D): Dropout(p=0.0, inplace=False)
            (A): ReLU()
          )
        )
        (1): Convolution(
          (conv): Conv3d(16, 16, kernel_size=(3, 3, 3), stride=(1, 1, 1), padding=(1, 1, 1))
          (adn): ADN(
            (N): BatchNorm3d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
            (D): Dropout(p=0.0, inplace=False)
            (A): ReLU()
          )
        )
      )
    )
    (1): AttentionLayer(
      (attention): AttentionBlock(
        (W_g): Sequential(
          (0): Convolution(
            (conv): Conv3d(16, 8, kernel_size=(1, 1, 1), stride=(1, 1, 1))
          )
          (1): BatchNorm3d(8, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        )
        (W_x): Sequential(
          (0): Convolution(
            (conv): Conv3d(16, 8, kernel_size=(1, 1, 1), stride=(1, 1, 1))
          )
          (1): BatchNorm3d(8, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        )
        (psi): Sequential(
          (0): Convolution(
            (conv): Conv3d(8, 1, kernel_size=(1, 1, 1), stride=(1, 1, 1))
          )
          (1): BatchNorm3d(1, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
          (2): Sigmoid()
        )
        (relu): ReLU()
      )
      (upconv): UpConv(
        (up): Convolution(
          (conv): ConvTranspose3d(32, 16, kernel_size=(3, 3, 3), stride=(2, 2, 2), padding=(1, 1, 1), output_padding=(1, 1, 1)
          )
          (adn): ADN(
            (N): BatchNorm3d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
            (D): Dropout(p=0.0, inplace=False)
            (A): ReLU()
          )
        )
      )
```

```
(merge): Convolution(
  (conv): Conv3d(32, 16, kernel_size=(3, 3, 3), stride=(1, 1, 1), padding=(1, 1, 1))
  (adn): ADN(
    (N): InstanceNorm3d(16, eps=1e-05, momentum=0.1, affine=False, track_running_stats=False)
    (D): Dropout(p=0.0, inplace=False)
    (A): PReLU(num_parameters=1)
  )
)
(submodule): Sequential(
  (0): ConvBlock(
    (conv): Sequential(
      (0): Convolution(
        (conv): Conv3d(16, 32, kernel_size=(3, 3, 3), stride=(2, 2, 2), padding=(1, 1, 1))
        (adn): ADN(
          (N): BatchNorm3d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
          (D): Dropout(p=0.0, inplace=False)
          (A): ReLU()
        )
      )
      (1): Convolution(
        (conv): Conv3d(32, 32, kernel_size=(3, 3, 3), stride=(1, 1, 1), padding=(1, 1, 1))
        (adn): ADN(
          (N): BatchNorm3d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
          (D): Dropout(p=0.0, inplace=False)
          (A): ReLU()
        )
      )
    )
  )
  (1): AttentionLayer(
    (attention): AttentionBlock(
      (W_g): Sequential(
        (0): Convolution(
          (conv): Conv3d(32, 16, kernel_size=(1, 1, 1), stride=(1, 1, 1))
        )
        (1): BatchNorm3d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
      (W_x): Sequential(
        (0): Convolution(
          (conv): Conv3d(32, 16, kernel_size=(1, 1, 1), stride=(1, 1, 1))
        )
        (1): BatchNorm3d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
      (psi): Sequential(
        (0): Convolution(
          (conv): Conv3d(16, 1, kernel_size=(1, 1, 1), stride=(1, 1, 1))
        )
        (1): BatchNorm3d(1, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): Sigmoid()
      )
      (relu): ReLU()
    )
    (upconv): UpConv(
      (up): Convolution(
        (conv): ConvTranspose3d(64, 32, kernel_size=(3, 3, 3), stride=(2, 2, 2), padding=(1, 1, 1), output_padding=(1,
              1, 1))
        (adn): ADN(
          (N): BatchNorm3d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
          (D): Dropout(p=0.0, inplace=False)
          (A): ReLU()
        )
      )
    )
    (merge): Convolution(
      (conv): Conv3d(64, 32, kernel_size=(3, 3, 3), stride=(1, 1, 1), padding=(1, 1, 1))
      (adn): ADN(
        (N): InstanceNorm3d(32, eps=1e-05, momentum=0.1, affine=False, track_running_stats=False)
        (D): Dropout(p=0.0, inplace=False)
        (A): PReLU(num_parameters=1)
      )
    )
    (submodule): Sequential(
      (0): ConvBlock(
        (conv): Sequential(
          (0): Convolution(
            (conv): Conv3d(32, 64, kernel_size=(3, 3, 3), stride=(2, 2, 2), padding=(1, 1, 1))
            (adn): ADN(
              (N): BatchNorm3d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
              (D): Dropout(p=0.0, inplace=False)
              (A): ReLU()
            )
          )
          (1): Convolution(
            (conv): Conv3d(64, 64, kernel_size=(3, 3, 3), stride=(1, 1, 1), padding=(1, 1, 1))
            (adn): ADN(
              (N): BatchNorm3d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
              (D): Dropout(p=0.0, inplace=False)
              (A): ReLU()
            )
          )
        )
      )
      (1): AttentionLayer(
        (attention): AttentionBlock(
          (W_g): Sequential(
            (0): Convolution(
              (conv): Conv3d(64, 32, kernel_size=(1, 1, 1), stride=(1, 1, 1))
            )
            (1): BatchNorm3d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
          )
          (W_x): Sequential(
            (0): Convolution(
              (conv): Conv3d(64, 32, kernel_size=(1, 1, 1), stride=(1, 1, 1))
            )
```

```
      (1): BatchNorm3d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
    (psi): Sequential(
      (0): Convolution(
        (conv): Conv3d(32, 1, kernel_size=(1, 1, 1), stride=(1, 1, 1))
      )
      (1): BatchNorm3d(1, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): Sigmoid()
    )
    (relu): ReLU()
  )
  (upconv): UpConv(
    (up): Convolution(
      (conv): ConvTranspose3d(128, 64, kernel_size=(3, 3, 3), stride=(2, 2, 2), padding=(1, 1, 1), output_padding
          =(1, 1, 1))
      (adn): ADN(
        (N): BatchNorm3d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (D): Dropout(p=0.0, inplace=False)
        (A): ReLU()
      )
    )
  )
  (merge): Convolution(
    (conv): Conv3d(128, 64, kernel_size=(3, 3, 3), stride=(1, 1, 1), padding=(1, 1, 1))
    (adn): ADN(
      (N): InstanceNorm3d(64, eps=1e-05, momentum=0.1, affine=False, track_running_stats=False)
      (D): Dropout(p=0.0, inplace=False)
      (A): PReLU(num_parameters=1)
    )
  )
  (submodule): Sequential(
    (0): ConvBlock(
      (conv): Sequential(
        (0): Convolution(
          (conv): Conv3d(64, 128, kernel_size=(3, 3, 3), stride=(2, 2, 2), padding=(1, 1, 1))
          (adn): ADN(
            (N): BatchNorm3d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
            (D): Dropout(p=0.0, inplace=False)
            (A): ReLU()
          )
        )
        (1): Convolution(
          (conv): Conv3d(128, 128, kernel_size=(3, 3, 3), stride=(1, 1, 1), padding=(1, 1, 1))
          (adn): ADN(
            (N): BatchNorm3d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
            (D): Dropout(p=0.0, inplace=False)
            (A): ReLU()
          )
        )
      )
    )
    (1): AttentionLayer(
      (attention): AttentionBlock(
        (W_g): Sequential(
          (0): Convolution(
            (conv): Conv3d(128, 64, kernel_size=(1, 1, 1), stride=(1, 1, 1))
          )
          (1): BatchNorm3d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        )
        (W_x): Sequential(
          (0): Convolution(
            (conv): Conv3d(128, 64, kernel_size=(1, 1, 1), stride=(1, 1, 1))
          )
          (1): BatchNorm3d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        )
        (psi): Sequential(
          (0): Convolution(
            (conv): Conv3d(64, 1, kernel_size=(1, 1, 1), stride=(1, 1, 1))
          )
          (1): BatchNorm3d(1, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
          (2): Sigmoid()
        )
        (relu): ReLU()
      )
      (upconv): UpConv(
        (up): Convolution(
          (conv): ConvTranspose3d(256, 128, kernel_size=(3, 3, 3), stride=(2, 2, 2), padding=(1, 1, 1),
              output_padding=(1, 1, 1))
          (adn): ADN(
            (N): BatchNorm3d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
            (D): Dropout(p=0.0, inplace=False)
            (A): ReLU()
          )
        )
      )
      (merge): Convolution(
        (conv): Conv3d(256, 128, kernel_size=(3, 3, 3), stride=(1, 1, 1), padding=(1, 1, 1))
        (adn): ADN(
          (N): InstanceNorm3d(128, eps=1e-05, momentum=0.1, affine=False, track_running_stats=False)
          (D): Dropout(p=0.0, inplace=False)
          (A): PReLU(num_parameters=1)
        )
      )
      (submodule): ConvBlock(
        (conv): Sequential(
          (0): Convolution(
            (conv): Conv3d(128, 256, kernel_size=(3, 3, 3), stride=(2, 2, 2), padding=(1, 1, 1))
            (adn): ADN(
              (N): BatchNorm3d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
              (D): Dropout(p=0.0, inplace=False)
              (A): ReLU()
            )
          )
          (1): Convolution(
```

```
                        (conv): Conv3d(256, 256, kernel_size=(3, 3, 3), stride=(1, 1, 1), padding=(1, 1, 1))
                        (adn): ADN(
                          (N): BatchNorm3d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
                          (D): Dropout(p=0.0, inplace=False)
                          (A): ReLU()
                        )
                      )
                    )
                  )
                )
              )
            )
          )
        )
      )
    )
    (2): Convolution(
      (conv): Conv3d(16, 2, kernel_size=(1, 1, 1), stride=(1, 1, 1))
    )
  )
)
```

Listado A.4. Arquitectura del Modelo SegResNet

```
SegResNet(
  (act_mod): ReLU(inplace=True)
  (convInit): Convolution(
    (conv): Conv3d(1, 16, kernel_size=(3, 3, 3), stride=(1, 1, 1), padding=(1, 1, 1), bias=False)
  )
  (down_layers): ModuleList(
    (0): Sequential(
      (0): Identity()
      (1): ResBlock(
        (norm1): GroupNorm(8, 16, eps=1e-05, affine=True)
        (norm2): GroupNorm(8, 16, eps=1e-05, affine=True)
        (act): ReLU(inplace=True)
        (conv1): Convolution(
          (conv): Conv3d(16, 16, kernel_size=(3, 3, 3), stride=(1, 1, 1), padding=(1, 1, 1), bias=False)
        )
        (conv2): Convolution(
          (conv): Conv3d(16, 16, kernel_size=(3, 3, 3), stride=(1, 1, 1), padding=(1, 1, 1), bias=False)
        )
      )
    )
    (1): Sequential(
      (0): Convolution(
        (conv): Conv3d(16, 32, kernel_size=(3, 3, 3), stride=(2, 2, 2), padding=(1, 1, 1), bias=False)
      )
      (1): ResBlock(
        (norm1): GroupNorm(8, 32, eps=1e-05, affine=True)
        (norm2): GroupNorm(8, 32, eps=1e-05, affine=True)
        (act): ReLU(inplace=True)
        (conv1): Convolution(
          (conv): Conv3d(32, 32, kernel_size=(3, 3, 3), stride=(1, 1, 1), padding=(1, 1, 1), bias=False)
        )
        (conv2): Convolution(
          (conv): Conv3d(32, 32, kernel_size=(3, 3, 3), stride=(1, 1, 1), padding=(1, 1, 1), bias=False)
        )
      )
      (2): ResBlock(
        (norm1): GroupNorm(8, 32, eps=1e-05, affine=True)
        (norm2): GroupNorm(8, 32, eps=1e-05, affine=True)
        (act): ReLU(inplace=True)
        (conv1): Convolution(
          (conv): Conv3d(32, 32, kernel_size=(3, 3, 3), stride=(1, 1, 1), padding=(1, 1, 1), bias=False)
        )
        (conv2): Convolution(
          (conv): Conv3d(32, 32, kernel_size=(3, 3, 3), stride=(1, 1, 1), padding=(1, 1, 1), bias=False)
        )
      )
    )
    (2): Sequential(
      (0): Convolution(
        (conv): Conv3d(32, 64, kernel_size=(3, 3, 3), stride=(2, 2, 2), padding=(1, 1, 1), bias=False)
      )
      (1): ResBlock(
        (norm1): GroupNorm(8, 64, eps=1e-05, affine=True)
        (norm2): GroupNorm(8, 64, eps=1e-05, affine=True)
        (act): ReLU(inplace=True)
        (conv1): Convolution(
          (conv): Conv3d(64, 64, kernel_size=(3, 3, 3), stride=(1, 1, 1), padding=(1, 1, 1), bias=False)
        )
        (conv2): Convolution(
          (conv): Conv3d(64, 64, kernel_size=(3, 3, 3), stride=(1, 1, 1), padding=(1, 1, 1), bias=False)
        )
      )
      (2): ResBlock(
        (norm1): GroupNorm(8, 64, eps=1e-05, affine=True)
        (norm2): GroupNorm(8, 64, eps=1e-05, affine=True)
        (act): ReLU(inplace=True)
        (conv1): Convolution(
          (conv): Conv3d(64, 64, kernel_size=(3, 3, 3), stride=(1, 1, 1), padding=(1, 1, 1), bias=False)
        )
        (conv2): Convolution(
          (conv): Conv3d(64, 64, kernel_size=(3, 3, 3), stride=(1, 1, 1), padding=(1, 1, 1), bias=False)
        )
      )
```

```
    )
    (3): Sequential(
      (0): Convolution(
        (conv): Conv3d(64, 128, kernel_size=(3, 3, 3), stride=(2, 2, 2), padding=(1, 1, 1), bias=False)
      )
      (1): ResBlock(
        (norm1): GroupNorm(8, 128, eps=1e-05, affine=True)
        (norm2): GroupNorm(8, 128, eps=1e-05, affine=True)
        (act): ReLU(inplace=True)
        (conv1): Convolution(
          (conv): Conv3d(128, 128, kernel_size=(3, 3, 3), stride=(1, 1, 1), padding=(1, 1, 1), bias=False)
        )
        (conv2): Convolution(
          (conv): Conv3d(128, 128, kernel_size=(3, 3, 3), stride=(1, 1, 1), padding=(1, 1, 1), bias=False)
        )
      )
      (2): ResBlock(
        (norm1): GroupNorm(8, 128, eps=1e-05, affine=True)
        (norm2): GroupNorm(8, 128, eps=1e-05, affine=True)
        (act): ReLU(inplace=True)
        (conv1): Convolution(
          (conv): Conv3d(128, 128, kernel_size=(3, 3, 3), stride=(1, 1, 1), padding=(1, 1, 1), bias=False)
        )
        (conv2): Convolution(
          (conv): Conv3d(128, 128, kernel_size=(3, 3, 3), stride=(1, 1, 1), padding=(1, 1, 1), bias=False)
        )
      )
      (3): ResBlock(
        (norm1): GroupNorm(8, 128, eps=1e-05, affine=True)
        (norm2): GroupNorm(8, 128, eps=1e-05, affine=True)
        (act): ReLU(inplace=True)
        (conv1): Convolution(
          (conv): Conv3d(128, 128, kernel_size=(3, 3, 3), stride=(1, 1, 1), padding=(1, 1, 1), bias=False)
        )
        (conv2): Convolution(
          (conv): Conv3d(128, 128, kernel_size=(3, 3, 3), stride=(1, 1, 1), padding=(1, 1, 1), bias=False)
        )
      )
      (4): ResBlock(
        (norm1): GroupNorm(8, 128, eps=1e-05, affine=True)
        (norm2): GroupNorm(8, 128, eps=1e-05, affine=True)
        (act): ReLU(inplace=True)
        (conv1): Convolution(
          (conv): Conv3d(128, 128, kernel_size=(3, 3, 3), stride=(1, 1, 1), padding=(1, 1, 1), bias=False)
        )
        (conv2): Convolution(
          (conv): Conv3d(128, 128, kernel_size=(3, 3, 3), stride=(1, 1, 1), padding=(1, 1, 1), bias=False)
        )
      )
    )
  )
  (up_layers): ModuleList(
    (0): Sequential(
      (0): ResBlock(
        (norm1): GroupNorm(8, 64, eps=1e-05, affine=True)
        (norm2): GroupNorm(8, 64, eps=1e-05, affine=True)
        (act): ReLU(inplace=True)
        (conv1): Convolution(
          (conv): Conv3d(64, 64, kernel_size=(3, 3, 3), stride=(1, 1, 1), padding=(1, 1, 1), bias=False)
        )
        (conv2): Convolution(
          (conv): Conv3d(64, 64, kernel_size=(3, 3, 3), stride=(1, 1, 1), padding=(1, 1, 1), bias=False)
        )
      )
    )
    (1): Sequential(
      (0): ResBlock(
        (norm1): GroupNorm(8, 32, eps=1e-05, affine=True)
        (norm2): GroupNorm(8, 32, eps=1e-05, affine=True)
        (act): ReLU(inplace=True)
        (conv1): Convolution(
          (conv): Conv3d(32, 32, kernel_size=(3, 3, 3), stride=(1, 1, 1), padding=(1, 1, 1), bias=False)
        )
        (conv2): Convolution(
          (conv): Conv3d(32, 32, kernel_size=(3, 3, 3), stride=(1, 1, 1), padding=(1, 1, 1), bias=False)
        )
      )
    )
    (2): Sequential(
      (0): ResBlock(
        (norm1): GroupNorm(8, 16, eps=1e-05, affine=True)
        (norm2): GroupNorm(8, 16, eps=1e-05, affine=True)
        (act): ReLU(inplace=True)
        (conv1): Convolution(
          (conv): Conv3d(16, 16, kernel_size=(3, 3, 3), stride=(1, 1, 1), padding=(1, 1, 1), bias=False)
        )
        (conv2): Convolution(
          (conv): Conv3d(16, 16, kernel_size=(3, 3, 3), stride=(1, 1, 1), padding=(1, 1, 1), bias=False)
        )
      )
    )
  )
  (up_samples): ModuleList(
    (0): Sequential(
      (0): Convolution(
        (conv): Conv3d(128, 64, kernel_size=(1, 1, 1), stride=(1, 1, 1), bias=False)
      )
      (1): UpSample(
        (upsample_non_trainable): Upsample(scale_factor=(2.0, 2.0, 2.0), mode='trilinear')
      )
    )
    (1): Sequential(
      (0): Convolution(
        (conv): Conv3d(64, 32, kernel_size=(1, 1, 1), stride=(1, 1, 1), bias=False)
```

```
      )
      (1): UpSample(
        (upsample_non_trainable): Upsample(scale_factor=(2.0, 2.0, 2.0), mode='trilinear')
      )
    )
    (2): Sequential(
      (0): Convolution(
        (conv): Conv3d(32, 16, kernel_size=(1, 1, 1), stride=(1, 1, 1), bias=False)
      )
      (1): UpSample(
        (upsample_non_trainable): Upsample(scale_factor=(2.0, 2.0, 2.0), mode='trilinear')
      )
    )
  )
  (conv_final): Sequential(
    (0): GroupNorm(8, 16, eps=1e-05, affine=True)
    (1): ReLU(inplace=True)
    (2): Convolution(
      (conv): Conv3d(16, 2, kernel_size=(1, 1, 1), stride=(1, 1, 1))
    )
  )
  (dropout): Dropout3d(p=0.2, inplace=False)
)
```

B. Ejemplos de Segmentaciones

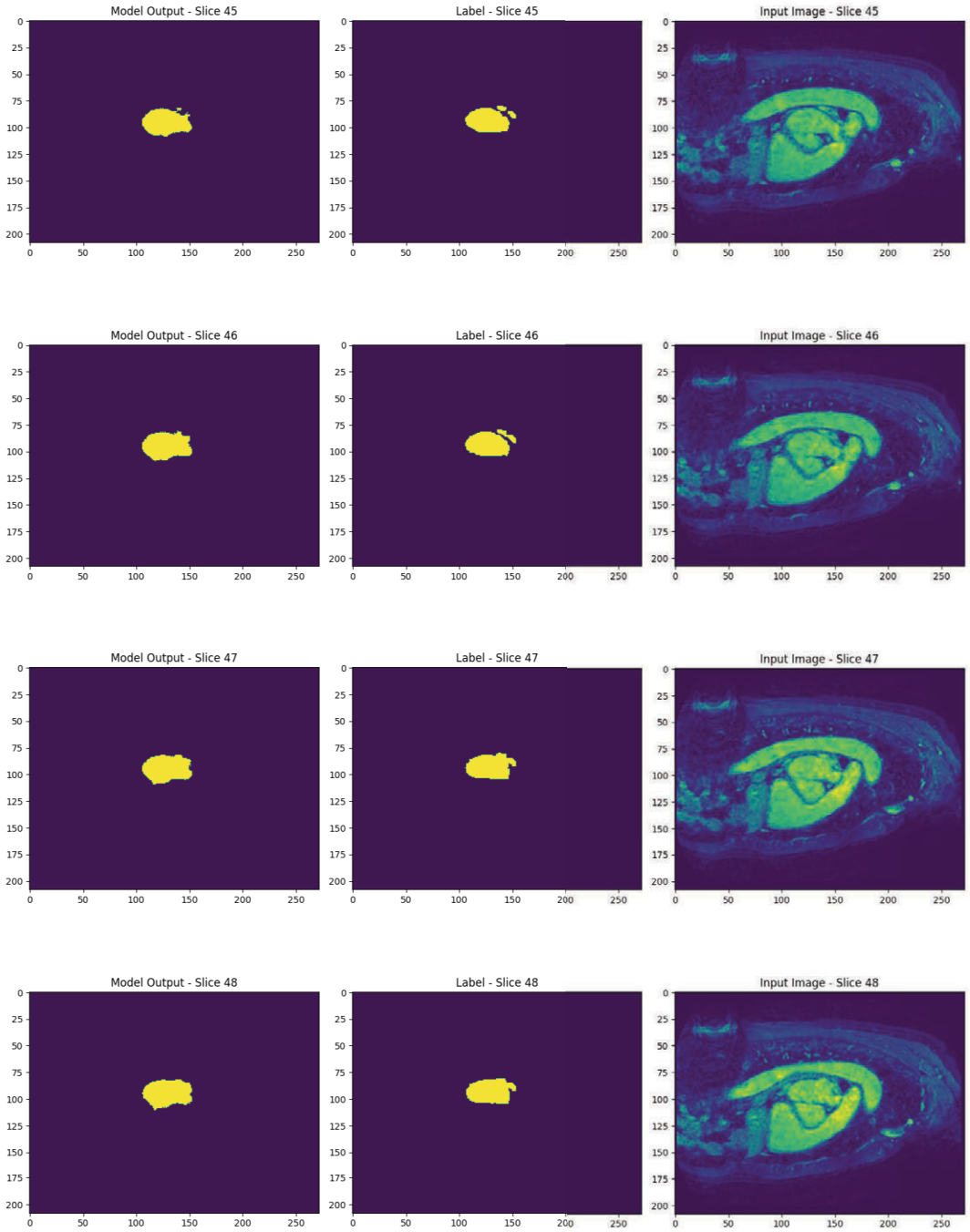


Figura B.1. Ejemplos de segmentación de UNet sobre el conjunto de validación

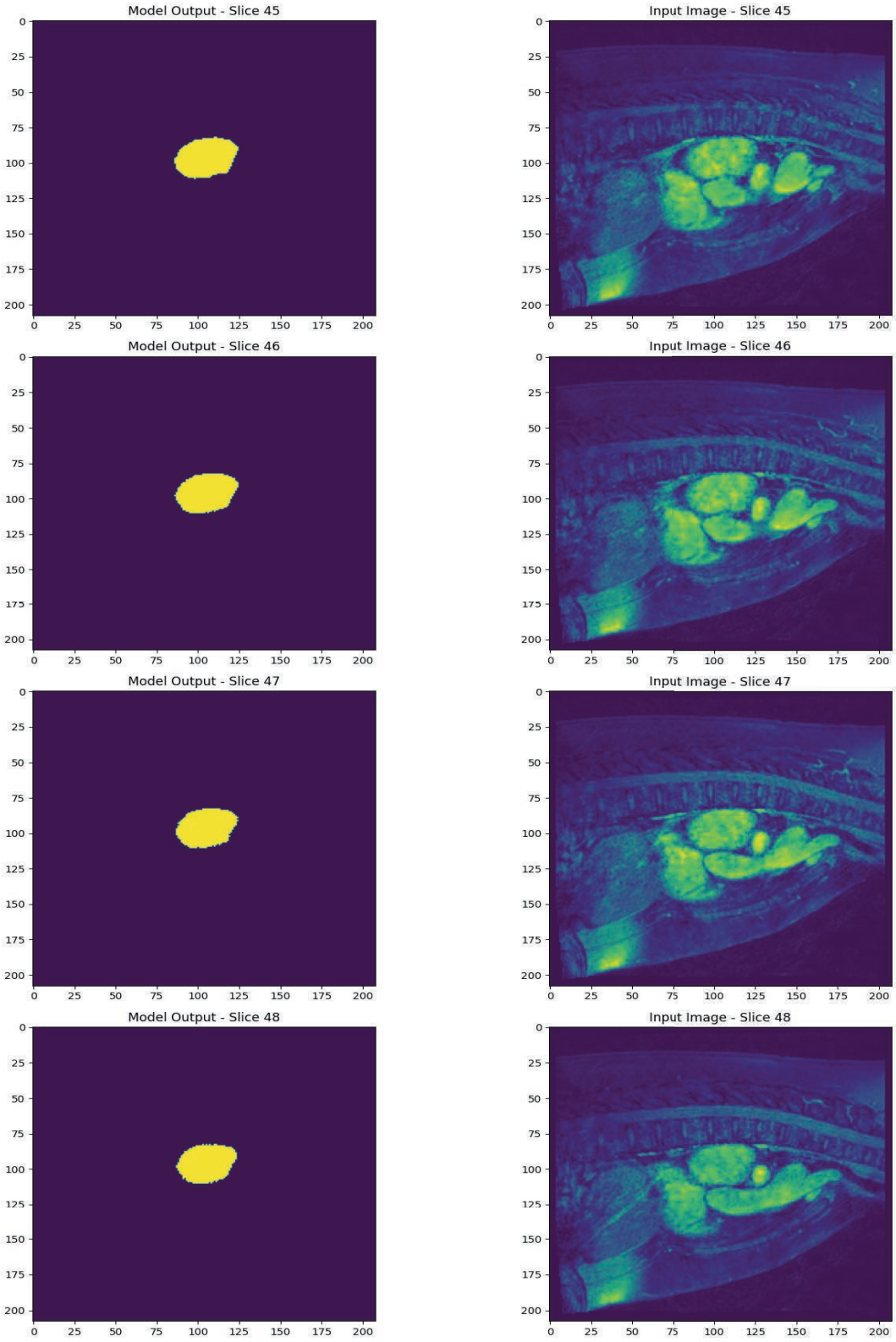


Figura B.2. Ejemplos de segmentación de UNet sobre el conjunto de prueba

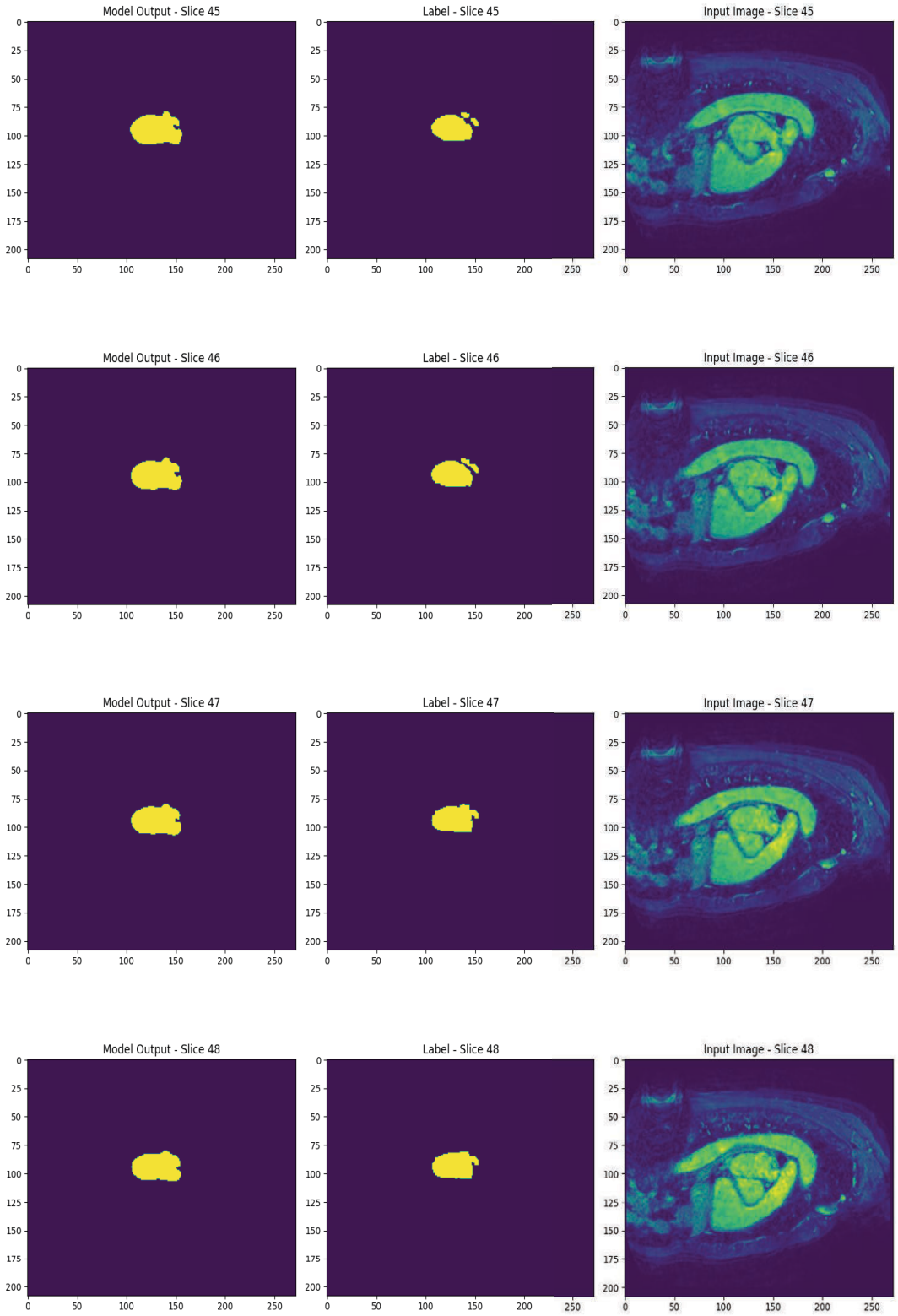


Figura B.3. Ejemplos de segmentación de DynUNet sobre el conjunto de validación

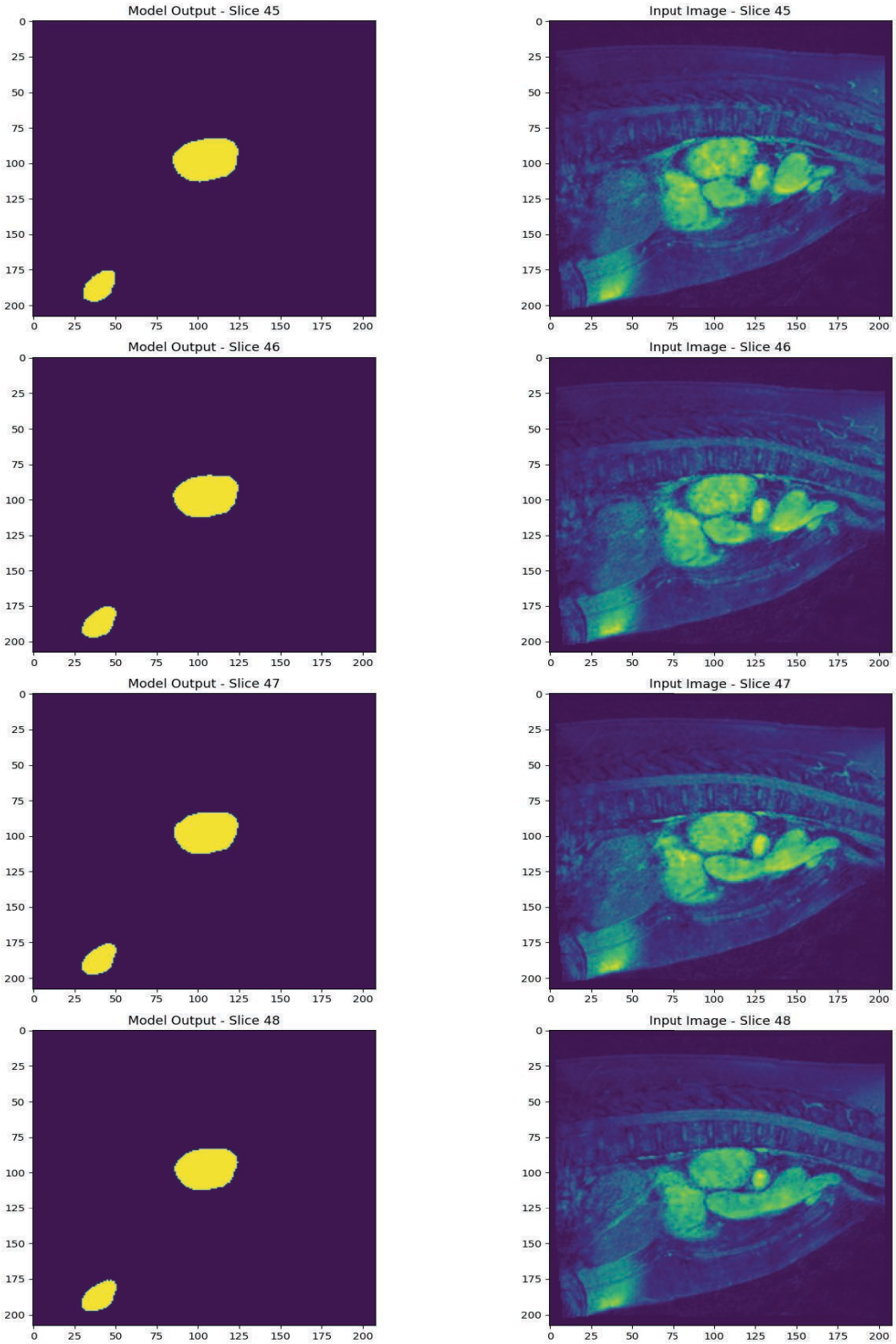


Figura B.4. Ejemplos de segmentación de DynUNet sobre el conjunto de prueba

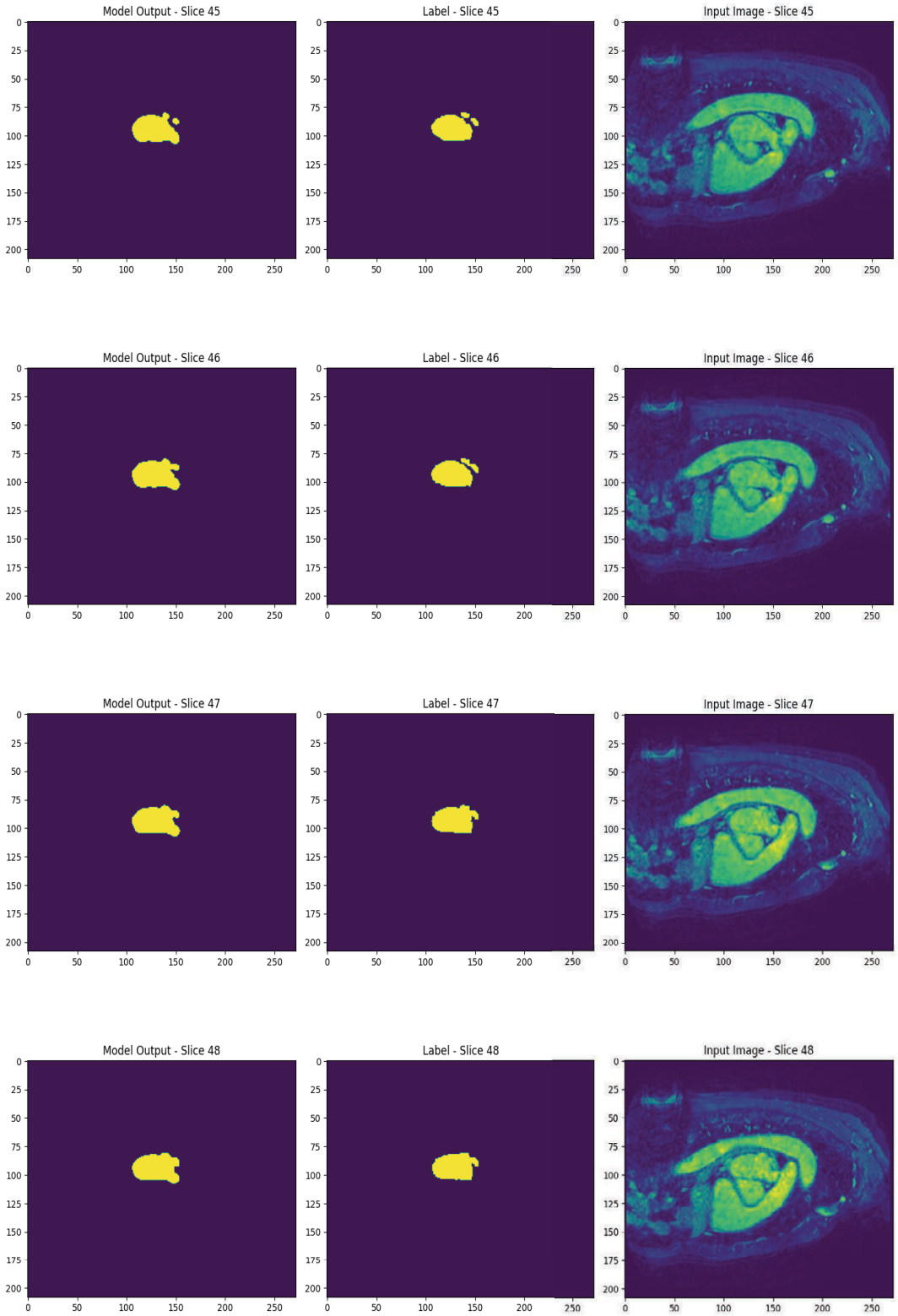


Figura B.5. Ejemplos de segmentación de AttentionUNet sobre el conjunto de validación

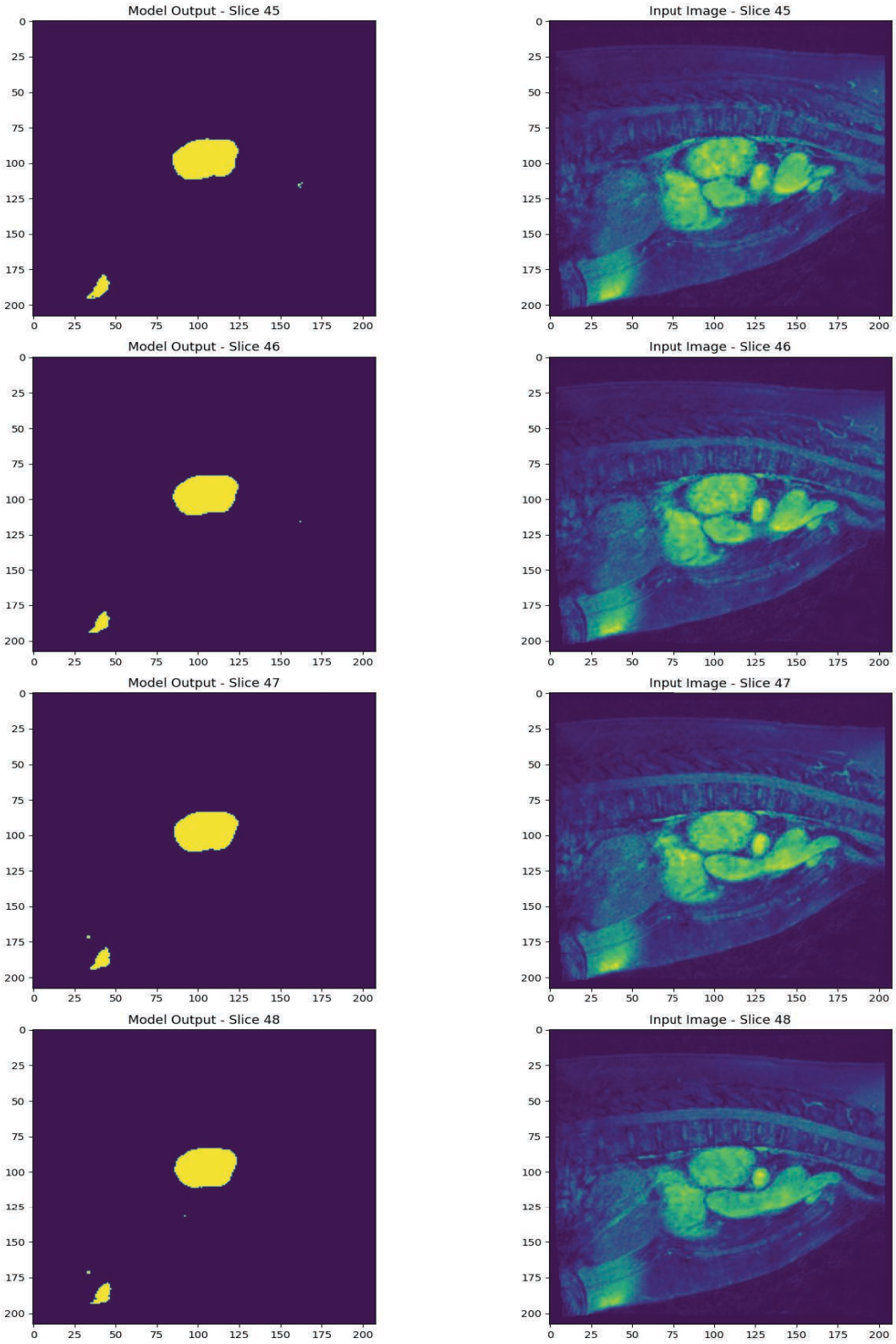


Figura B.6. Ejemplos de segmentación de AttentionUNet sobre el conjunto de prueba

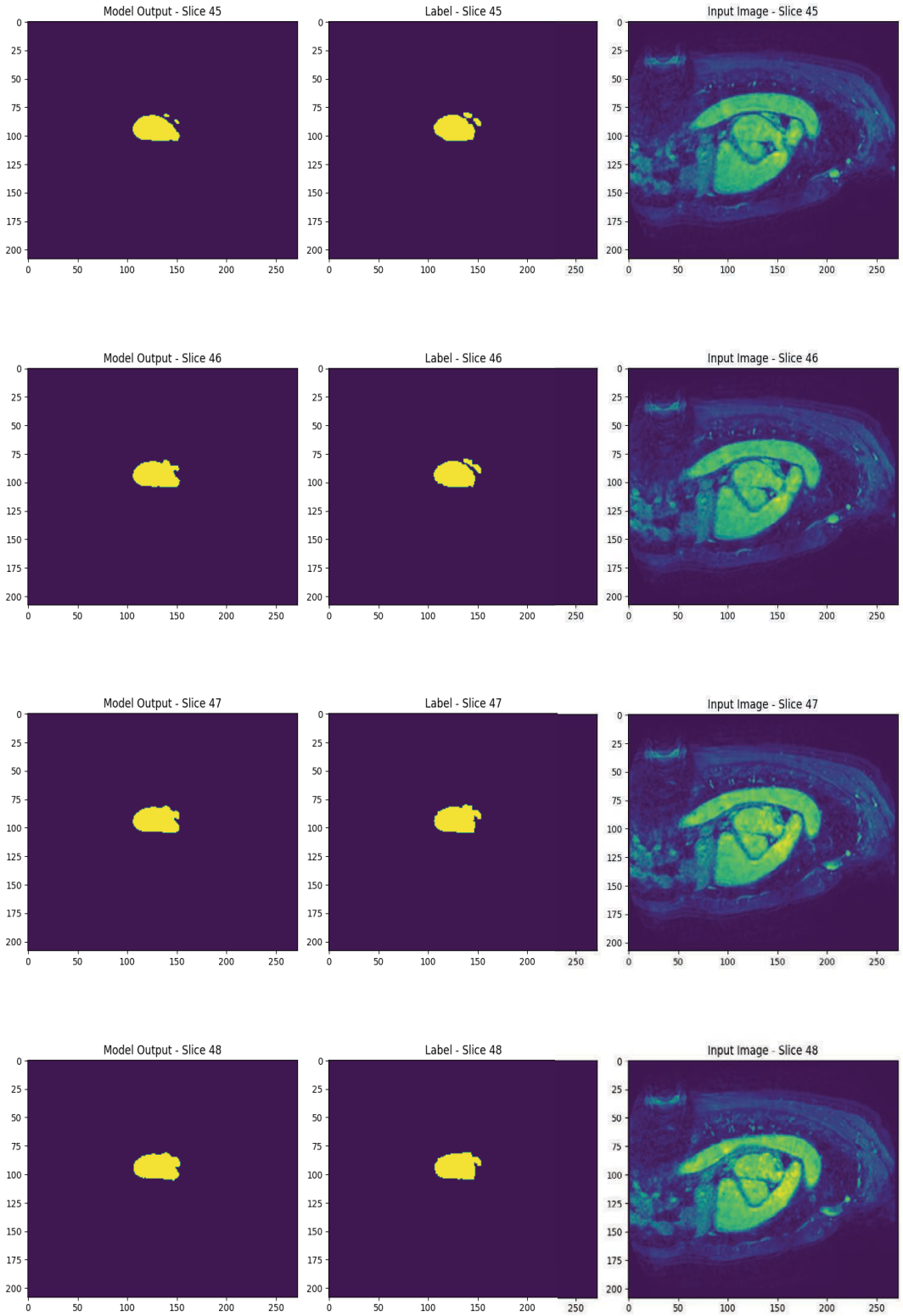


Figura B.7. Ejemplos de segmentación de SegResNet sobre el conjunto de validación

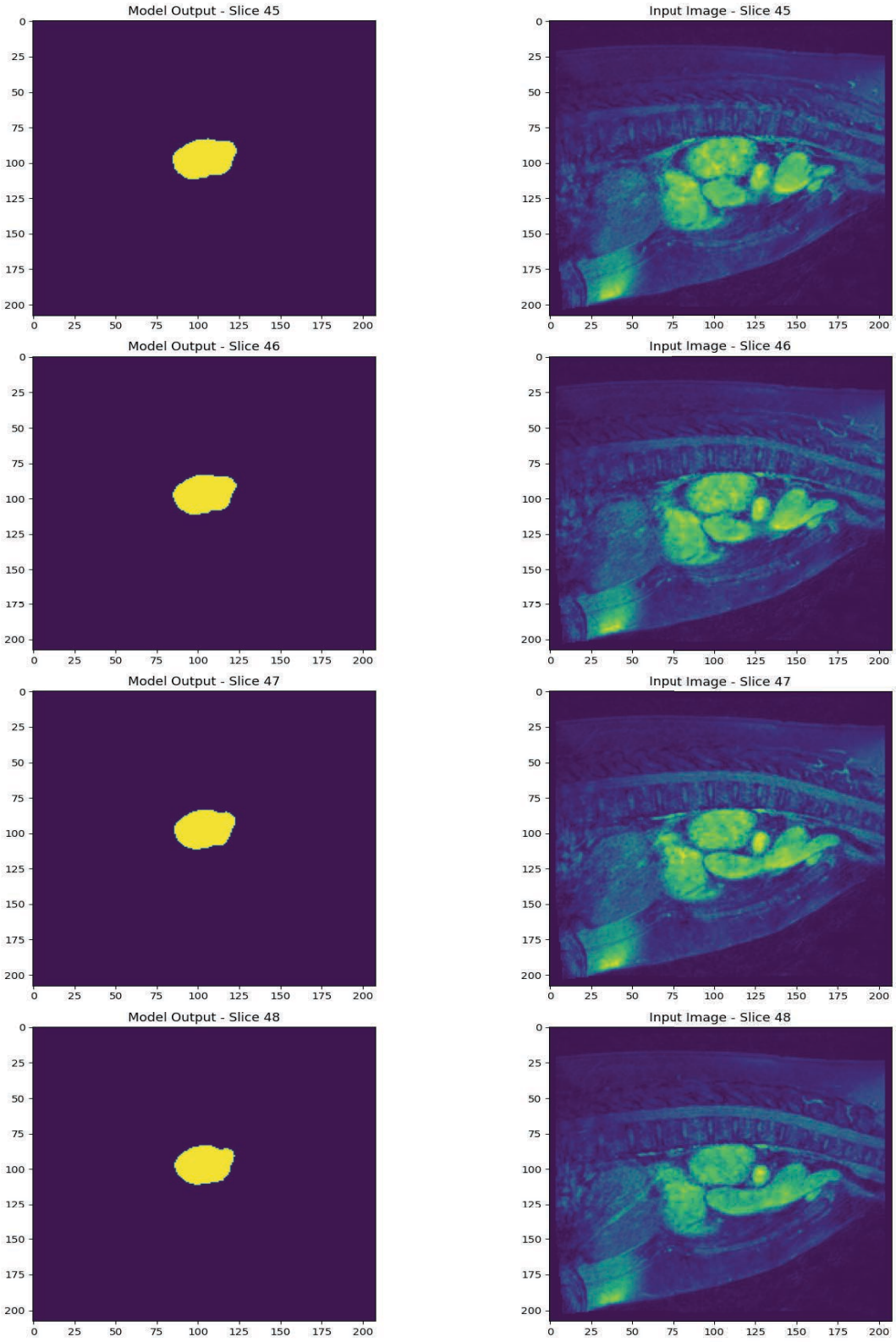


Figura B.8. Ejemplos de segmentación de SegResNet sobre el conjunto de prueba

Índice de términos

Siglas

RGPD Reglamento General de Protección de Datos. 50

RMC Resonancia Magnética Cardíaca. 2, 8

RX Rayos X. 1

TC Tomografía computarizada. 1

Referencias

[1] W. H. Organization, *Cardiovascular diseases (CVDs)*. dirección: https://www.who.int/es/health-topics/cardiovascular-diseases#tab=tab_1.

[2] George Washington University Hospital, *Magnetic Resonance Imaging (MRI)*. dirección: https://es.gwhospital.com/conditions-services/radiology/magnetic-resonance-imaging-mri.

[3] John's Hopkins Medicine, *Magnetic Resonance Imaging (IRM)*. dirección: https://www.hopkinsmedicine.org/health/treatment-tests-and-therapies/magnetic-resonance-imaging-mri.

[4] National Institute of Biomedical Imaging and Bioengineering, *Imagen por Resonancia Magnética (IRM)*. dirección: https://www.nibib.nih.gov/science-education/science-topics/magnetic-resonance-imaging-mri.

[5] Advanced Orthopaedic Associates, P.A., *MRI Frequently Asked Questions: Everything You Need to Know*. dirección: https://www.aoaortho.com/mri-frequently-asked-questions-everything-you-need-to-know/.

[6] Wikipedia, *Felix Bloch*. dirección: https://es.wikipedia.org/wiki/Felix_Bloch.

[7] Wikipedia, *Edward Mills Purcell*. dirección: https://es.wikipedia.org/wiki/Edward_Mills_Purcell.

[8] Armenian History, *The Thorny Road to Independence and Conclusion – Cilicia*. dirección: https://armenian-history.com/thorny-road-independence-conclusion-cilicia/.

[9] EcuRed, *Paul C. Lauterbur*. dirección: https://www.ecured.cu/Paul_C._Lauterbur.

[10] El Baúl Radiológico, *In Memoriam. Sir Peter Mansfield*. dirección: https://www.elbaulradiologico.com/2017/02/in-memoriam-sir-peter-mansfield-by-luis.html.

[11] Midwestern Career College, *The History of the MRI: The Development of Medical Resonance Imaging*. dirección: https://mccollege.edu/aas-in-magnetic-resonance-imaging-mri-technology/about-the-mri-technology-career/the-history-of-the-mri-development-of-medical-resonance-imaging/.

[12] M. Marcelo Gálvez, «Algunos hitos históricos en el desarrollo del diagnóstico médico por imágenes,» *Revista Médica Clínica Las Condes*, vol. 24, n.º 1, págs. 5-13, 2013. dirección: https://www.sciencedirect.com/science/article/pii/S0716864013701238.

[13] Mayo Clinic, *Resonancia Magnética*. dirección: https://www.mayoclinic.org/es/tests-procedures/mri/about/pac-20384768.

[14] Dayton Children's Hospital, *Dictionary: Left Atrium*. dirección: https://www.childrensdayton.org/kidshealth/a/dictionary-left-atrium/es.

[15] Wikimedia Commons, *Diagram of the Human Heart*. dirección: https://commons.wikimedia.org/wiki/File:Diagram_of_the_human_heart_(cropped).svg.

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

[17] DataCamp, *What is Machine Learning? Definition, Types, Tools and More*, 2024. dirección: https://www.datacamp.com/blog/what-is-machine-learning.

[18] HardZone, *IA, Machine Learning y Deep Learning. ¿Cuál es la Diferencia?* 2025. dirección: https://hardzone.es/tutoriales/rendimiento/diferencias-ia-deep-machine-learning/.

[19] Adobe for Business, *AI vs. Machine Learning*. dirección: https://business.adobe.com/au/products/real-time-customer-data-platform/ai-vs-machine-learning.html.

[20] W. S. McCulloch y W. Pitts, «A logical calculus of the ideas immanent in nervous activity,» *The bulletin of mathematical biophysics*, vol. 5, págs. 115-133, 1943.

[21] Medium, *Introduction To Artificial Intelligence — Neural Networks*, 2019. dirección: https://medium.com/data-science-collective/introduction-to-artificial-intelligence-neural-networks-5c7244f60425.

[22] GeeksforGeeks, *Weights and Bias in Neural Networks*, 2024. dirección: https://www.geeksforgeeks.org/deep-learning/the-role-of-weights-and-bias-in-neural-networks/.

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

[24] GeeksforGeeks, *Artificial Neural Networks and its Applications*, 2025. dirección: https://www.geeksforgeeks.org/artificial-intelligence/artificial-neural-networks-and-its-applications/.

[25] T. D. Science, *The Importance and Reasoning behind Activation Functions*, 2021. dirección: https://towardsdatascience.com/the-importance-and-reasoning-behind-activation-functions-4dc00e74db41/.

[26] Medium, *Activation Functions in Neural Networks: A Simplified Overview*, 2023. dirección: https://medium.com/@AI-Simplified/activation-functions-in-neural-networks-a-simplified-overview-ad012c03532a.

[27] D. E. Rumelhart, G. E. Hinton y R. J. Williams, «Learning representations by back-propagating errors,» *nature*, vol. 323, n.º 6088, págs. 533-536, 1986.

[28] T. D. Science, *Understanding Deep Learning Optimizers: Momentum, AdaGrad, RMSProp Adam*, 2023. dirección: https://towardsdatascience.com/understanding-deep-learning-optimizers-momentum-adagrad-rmsprop-adam-e311e377e9c2/.

[29] Medium, *Recurrent Neural Networks (RNNs) for Sequence Processing*, 2024. dirección: https://medium.com/@prasanNH/recurrent-neural-networks-rnns-for-sequence-processing-a851266e278f.

[30] Wikipedia, *Bias–variance Tradeoff*. dirección: https://en.wikipedia.org/wiki/Bias%E2%80%93variance_tradeoff.

[31] GeeksforGeeks, *ML | Underfitting and Overfitting*, 2025. dirección: https://www.geeksforgeeks.org/machine-learning/underfitting-and-overfitting-in-machine-learning/.

[32] Medium, *Overfitting vs. Underfitting: A Conceptual Explanation*, 2018. dirección: https://medium.com/data-science/overfitting-vs-underfitting-a-conceptual-explanation-d94ee20ca7f9.

[33] Medium, *Técnicas de Regularización Básicas para Redes Neuronales*, 2019. dirección: https://medium.com/metadatos/t%C3%A9cnicas-de-regularizaci%C3%B3n-b%C3%A1sicas-para-redes-neuronales-b48f396924d4.

[34] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever y R. Salakhutdinov, «Dropout: a simple way to prevent neural networks from overfitting,» *The journal of machine learning research*, vol. 15, n.º 1, págs. 1929-1958, 2014.

[35] S. Ioffe y C. Szegedy, «Batch normalization: Accelerating deep network training by reducing internal covariate shift,» en *International conference on machine learning*, pmlr, 2015, págs. 448-456.

[36] Medium, *Convolutional Neural Network*, 2019. dirección: https://medium.com/data-science/covolutional-neural-network-cb0883dd6529.

[37] Analytics Vidhya, *Convolutional Neural Networks (CNN) in Deep Learning*, 2025. dirección: https://www.analyticsvidhya.com/blog/2021/05/convolutional-neural-networks-cnn/.

[38] K. O'shea y R. Nash, «An Introduction to Convolutional Neural Networks,» *arXiv preprint arXiv:1511.08458*, 2015.

[39] P. Purwono, A. Ma'arif, W. Rahmaniar, H. I. K. Fathurrahman, A. Z. K. Frisky y Q. M. ul Haq, «Understanding of Convolutional Neural Network (CNN): A Review,» *International Journal of Robotics and Control Systems*, vol. 2, n.º 4, págs. 739-748, 2022.

[40] Medium, *Image Segmentation — A Beginner's Guide*, 2024. dirección: https://medium.com/@raj.pulapakura/image-segmentation-a-beginners-guide-0ede91052db7.

[41] Medium, *Mastering the Art of Image Segmentation: A Comprehensive Guide to Image Segmentation Techniques*, 2023. dirección: https://medium.com/@1marcusangella/mastering-the-art-of-image-segmentation-a-comprehensive-guide-to-image-segmentation-techniques-545a75010a35.

[42] PyImageSearch, *Semantic vs. Instance vs. Panoptic Segmentation*, 2022. dirección: https://pyimagesearch.com/2022/06/29/semantic-vs-instance-vs-panoptic-segmentation/.

[43] O. Ronneberger, P. Fischer y T. Brox, «U-net: Convolutional networks for biomedical image segmentation,» en *Medical image computing and computer-assisted intervention–MICCAI 2015: 18th international conference, Munich, Germany, October 5-9, 2015, proceedings, part III 18*, Springer, 2015, págs. 234-241.

[44] GeeksforGeeks, *U-Net Architecture Explained*, 2025. dirección: https://www.geeksforgeeks.org/machine-learning/u-net-architecture-explained/.

[45] Ö. Çiçek, A. Abdulkadir, S. S. Lienkamp, T. Brox y O. Ronneberger, «3D U-Net: learning dense volumetric segmentation from sparse annotation,» en *Medical Image Computing and Computer-Assisted Intervention–MICCAI 2016: 19th International Conference, Athens, Greece, October 17-21, 2016, Proceedings, Part II 19*, Springer, 2016, págs. 424-432.

[46] F. Isensee, P. F. Jaeger, S. A. Kohl, J. Petersen y K. H. Maier-Hein, «nnU-Net: a self-configuring method for deep learning-based biomedical image segmentation,» *Nature methods*, vol. 18, n.º 2, págs. 203-211, 2021.

[47] N.-T. Do, H.-S. Vo-Thanh, T.-T. Nguyen-Quynh y S.-H. Kim, «3D-DDA: 3D Dual-Domain Attention for Brain Tumor Segmentation,» en *2023 IEEE International Conference on Image Processing (ICIP)*, IEEE, 2023, págs. 3215-3219.

[48] O. Oktay, J. Schlemper, L. L. Folgoc et al., «Attention u-net: Learning where to look for the pancreas,» *arXiv preprint arXiv:1804.03999*, 2018.

[49] Medium, *A Detailed Explanation of the Attention U-Net*, 2020. dirección: https://medium.com/data-science/a-detailed-explanation-of-the-attention-u-net-b371a5590831.

[50] Idiot Developer, *Attention UNet and its Implementation in Tensorflow*, 2023. dirección: https://idiotdeveloper.com/attention-unet-and-its-implementation-in-tensorflow/.

[51] K. He, X. Zhang, S. Ren y J. Sun, «Deep residual learning for image recognition,» en *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, págs. 770-778.

[52] W. Xie, P. Chen, Z. Li et al., «A Two stage deep learning network for automated femoral segmentation in bilateral lower limb CT scans,» *Scientific Reports*, vol. 15, n.º 1, pág. 9198, 2025.

[53] A. Goldbloom, *Kaggle: Your Home for Data Science*. dirección: https://www.kaggle.com/.

[54] G. van Rossum, *Python*. dirección: https://www.python.org/.

[55] A. Paszke, S. Gross, S. Chintala y G. Chanan, *PyTorch*. dirección: https://pytorch.org/.

[56] NVIDIA and King's College London, *MONAI: Medical Open Network for AI*. dirección: https://monai.io/.

[57] M. Antonelli, A. Reinke, S. Bakas et al., «The Medical Segmentation Decathlon,» *Nature communications*, vol. 13, n.º 1, pág. 4128, 2022.

[58] A. L. Simpson, M. Antonelli, S. Bakas et al., «A large annotated medical image dataset for the development and evaluation of segmentation algorithms,» *arXiv preprint arXiv:1902.09063*, 2019.

[59] NVIDIA and King's College London, *MONAI: Loss Functions*. dirección: https://docs.monai.io/en/stable/losses.html.

[60] Medium, *Dice Loss in Medical Image Segmentation*. dirección: https://cvinvolution.medium.com/dice-loss-in-medical-image-segmentation-d0e476eb486.

[61] Adam Paszke and Sam Gross and Soumith Chintala and Gregory Chanan, *CrossEntropyLoss*. dirección: https://docs.pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.html.

[62] Adam Paszke and Sam Gross and Soumith Chintala and Gregory Chanan, *AdamW*. dirección: https://docs.pytorch.org/docs/stable/generated/torch.optim.AdamW.html.

[63] Medium, *Adam vs. AdamW: Understanding Weight Decay and Its Impact on Model Performance*. dirección: https://yassin01.medium.com/adam-vs-adamw-understanding-weight-decay-and-its-impact-on-model-performance-b7414f0af8a1.

[64] A. F. Osman y N. M. Tamam, «Attention-aware 3D U-Net convolutional neural network for knowledge-based planning 3D dose distribution prediction of head-and-neck cancer,» *Journal of applied clinical medical physics*, vol. 23, n.º 7, e13630, 2022.

[65] M. Zarenia, Y. Zhang, C. Sarosiek, R. Conlin, A. Amjad y E. Paulson, «Deep learning-based automatic contour quality assurance for auto-segmented abdominal MR-Linac contours,» *Physics in Medicine & Biology*, vol. 69, n.º 21, pág. 215 029, 2024.