ULPGC
Universidad de
Las Palmas de
Gran Canaria

ESCUELA DE INGENIERÍA DE TELECOMUNICACIÓN Y ELECTRÓNICA

TRABAJO FIN DE GRADO

Detección y segmentación de fracturas y fisuras óseas mediante imágenes ecográficas con IA

Titulación: Grado en Ingeniería en Tecnologías de la Telecomunicación

Mención: Sonido e Imagen

Autor/a: Néstor Manuel Acosta Rivero

Tutor/a: Juan Bautista Ruiz Alzola

Cotutor/a: David García Mato

Fecha: Julio de 2023

TABLA DE CONTENIDO

TABLA DE FIGURAS

TABLAS

TABLA DE ECUACIONES

LISTA DE ACRÓNIMOS

- **IA** Inteligencia Artificial
- **US** Ultrasonido
- **RM** Resonancia Magnética
- **TC** Tomografía Computarizada
- **IU** Interfaz de Usuario
- **GUI** Interfaz Gráfica de Usuario
- **ML** Machine Learning
- **ROC** Receiver Operating Characteristic (Característica Operativa del Receptor)
- **AUC** Area Under the Curve (Área Bajo la Curva)
- **CAM** Class Activation Map (Mapa de Activación de Clases)
- **PoCUS** Ecografía en el Punto de Atención (Point-of-Care Ultrasound)
- **CNN** Convolutional Neural Network (Red Neuronal Convolucional)
- **CPU** Central Processing Unit (Unidad Central de Procesamiento)

- **GPU** Graphics Processing Unit
  (Unidad de Procesamiento de Gráficos)

- **COITT** Colegio Oficial de Ingenieros Técnicos en Telecomunicación

1 INTRODUCCIÓN

1.1 FRACTURAS Y FISURAS.

1.1.1 ¿Qué son las fracturas y las fisuras?

Se conoce como fractura ósea a la discontinuidad que se produce en los dos extremos de un hueso cuando este se ha roto. Por el contrario, se les llama fisura a aquellas fracturas incompletas, es decir, aquellas que no han terminado por dividir el hueso en dos porciones. Numerosos autores definen esta última como un tipo de fractura, e incluso la llegan a describir como una antesala a la misma [1].

1.1.2 Tipos de fracturas.

Actualmente, tal y como se debate en [2], no existe un consenso lo suficientemente objetivo entre observadores como para clasificar las fracturas de una forma determinada. El objetivo principal que persiguen este tipo de clasificaciones es lograr, lo que análogamente podríamos llamar, un "manual de tratamiento" para cualquier tipo de fractura. Actualmente, una de las bases fundamentales a nivel teórico, que ha sido validada por múltiples expertos es lo que se conoce como el Manual de Schneider. En él se enuncia lo siguiente: "todas las fracturas deben ser inmovilizadas, pero, si los fragmentos están desplazados, previamente deben ser reducidos" (véase Fig. 1.1). Lo que quiere decir F. Richard Schneider con esta expresión, se puede comprender en el siguiente gráfico extraído de su propio manual [3]. Básicamente consiste en que los fragmentos desplazados sean llevados, en la medida de lo posible, a su posición previa a la factura para que consolide adecuadamente.

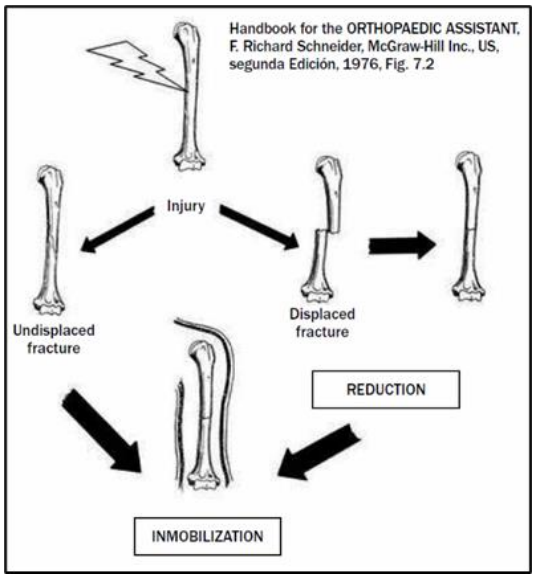


Fig. 1.1: Representación gráfica que muestra el procedimiento básico a seguir para la curación de fracturas según lo recogido en el manual de F. Richard Schneider.

Entiéndase mejor ahora, a qué se hacía alusión previamente cuando se mencionaba el término de "reducción".

En base a este manual, en el año 1981 se hace una primera clasificación de fracturas para huesos largos (véase Fig. 1.2). En esta clasificación se separan las fracturas en cuatro grupos.

Grupo 1. Simples: Este tipo de fracturas están compuesta por una discontinuidad principal, la cual puede ser transversa (lisa o dentellada), oblicua (corta o larga), espiroidea o transverso-oblicua. Estas son fáciles de reducir e inmovilizar empleando distintos tipos de técnicas.

Grupo 2. A tercer fragmento: Estas aumentan la complejidad con respecto al anterior grupo, puesto que ya no se dispone de una fractura principal que separa el hueso en dos partes, sino que estas van acompañadas de un tercer fragmento, de ahí su nombre. En comparación con lo ya visto, el proceso de reducción e inmovilización es más complejo para este tipo de casos, en el que, si se llevara una mala praxis en su tratamiento, podría derivar en una necrosis avascular (retardo de la consolidación o no unión).

Grupo 3. Segmentarias: Las fracturas denominadas segmentarias, son aquellas las cuales disponen de dos niveles de trazos, cada cual con su propia morfología. Esto hace

complicado el procedimiento de reducción, siendo este proceso estrictamente necesario para la correcta curación de la fractura. En caso contrario, suponiendo que no se toman las medidas necesarias para la recuperación del hueso, existe la gran posibilidad de que esta derive en una necrosis avascular.

Grupo 4. Conminutas: Estas son aquellas fracturas que, por la naturaleza de su causa, están compuestas por multitud de fragmentos de diferentes tamaños. Las fracturas pertenecientes a este grupo se pueden dividir en dos subgrupos.

4.1-. Moderadas Conminutas: Las que están formadas por unos pocos fragmentos de gran tamaño, en cuyo proceso de reducción se deben agrupar dichos pedazos formando nuevamente el hueso en su totalidad.

4.2-. Gran Conminutas: Estas son las compuestas de un gran número de fragmentos con distinta morfología y escala. Comparado con el anterior subgrupo, en este caso es imposible de llevar a cabo un proceso de reducción.

Cabe mencionar que, para ambos subgrupos, existe el riesgo de que un mal tratamiento puede derivar en una necrosis avascular, al igual que lo visto en anteriores casos.

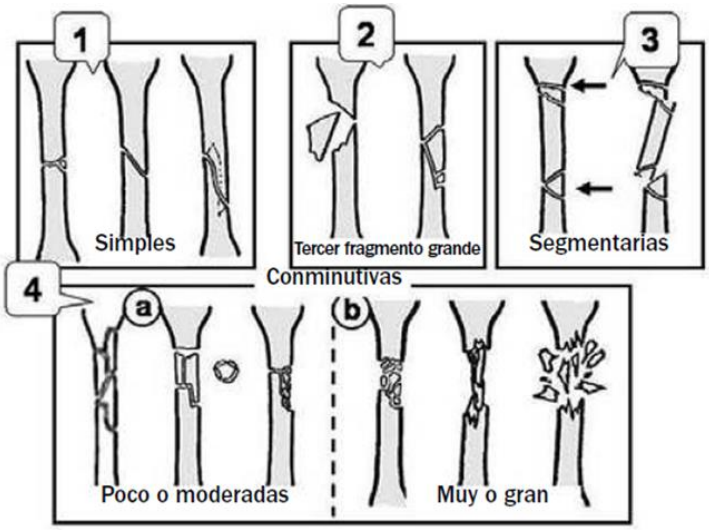


Fig. 1.2: Ilustración que muestra los tipos de fracturas de huesos largos de forma gradual según la dificultad de su tratamiento, siendo el (1) el más sencillo de tratar y el (4) el más complicado [2].

Seguidamente, una vez vista la clasificación de fracturas para huesos largos, cabe destacar aquellas fracturas que se producen en las grandes articulaciones, las cuales ponen el riesgo

la integridad del cartílago hialino (tejido situado en todas las articulaciones del cuerpo, cuyo objetivo absorber golpes y prevenir lesiones debido a movimientos bruscos). La clasificación de fracturas articulares se separa en únicamente dos grupos (véase Fig. 1.3).

Grupo 1. Simples: Aquellas fracturas que disponen de un número reducido de fragmentos con trazos bien definidos, susceptibles de ser fácilmente reducibles e inmovilizados usando implantes en caso de que fuera necesario.

Grupo 2. Conminutas: Al igual que en las fracturas en huesos largos, se divide en dos subgrupos con el mismo nombre:

2.1-. Moderadas Conminutas: Formada por unos pocos fragmentos generalmente grandes, y cuyas probabilidades de reducción e inmovilización son elevadas.

2.2-. Gran Conminutas: En este tipo de fracturas articulares se encuentran múltiples pedazos, generalmente pequeños, y en donde en mucho de los casos se llega a producir pérdida ósea. En este caso, al igual que se expresó en la clasificación para fracturas en huesos largos, el proceso de reducción es imposible de llevarse a cabo.

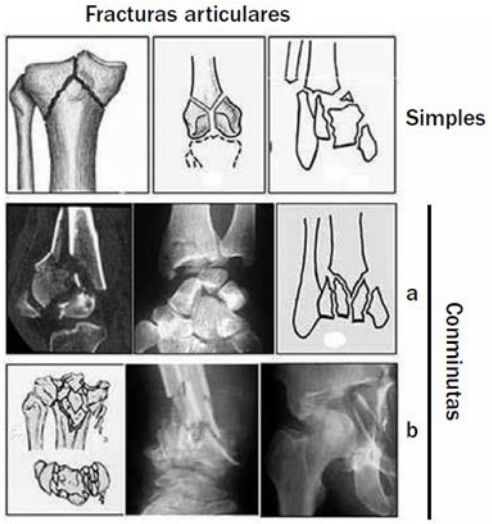


Fig. 1.3: Representación que muestra los dos tipos de fracturas articulares de forma gradual según la complejidad de su tratamiento, siendo las "simples" las más sencillas de tratar y las "cominutas – b" las más complejas [2].

Finalmente, sin entrar en profundidad, ya que no es objeto de este proyecto abordar este tipo de fracturas, es necesario mencionar las llamadas "fracturas abiertas"; son las que

quedan expuestas al aire y suponen un daño severo a aquellos tejidos blandos que las rodean. Este tipo de fracturas se clasifican según su dimensión y características.

- ➔ **Primer grado:** Son generalmente pequeñas, llegan a ser de hasta 1 centímetro.
- ➔ **Segundo grado:** Ocupan alrededor de un 33% de la zona afectada.
- ➔ **Tercer grado:** Aquellas que sobrepasan de largo el 33% de la región comprometida.

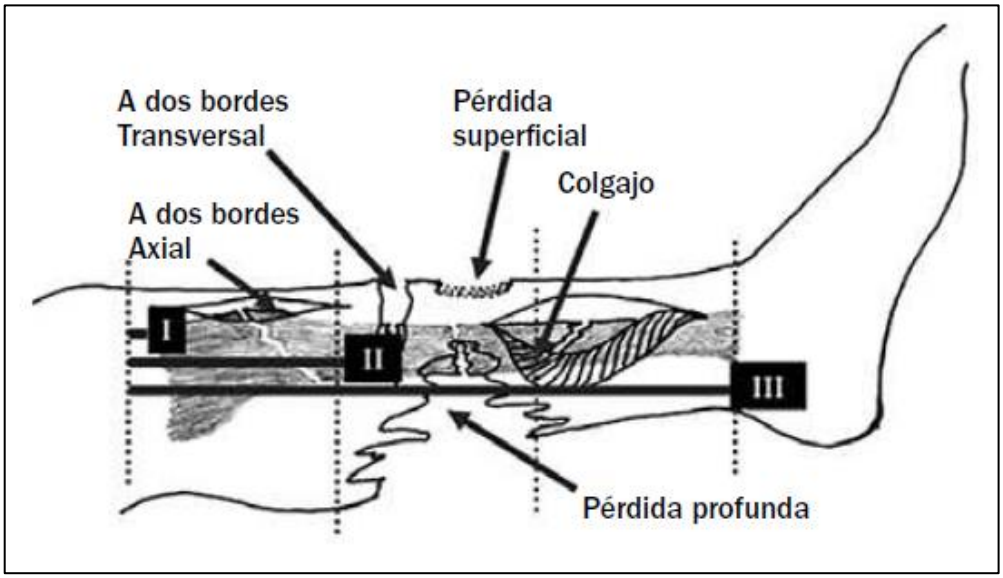


Fig. 1.4: Imagen que muestra la clasificación de las fracturas abiertas según su grado. También se muestran algunos de los nombres acuñados al tipo de daño que pueden llegar a producir estas fracturas en la piel [2].

Es importante llevar una clasificación estricta para este tipo de fracturas ya que estas incurren directamente en la integridad de los distintos tejidos blandos que la rodean.

1.1.3 Causas de las fracturas.

La manera más fiable de conformar un manual que sirva de guía para generalizar las causas de fracturas sería hacer una revisión profunda de diversos historiales de pacientes, extrayendo de estos aquellos puntos comunes que hagan derivar en el trágico escenario de una fractura ósea. Aunque a grandes rasgos, se podrían generalizar distintas causas que desencadenen en el ya mencionado traumatismo, tales como: caídas, golpes, accidentes de tráfico, movimientos bruscos, accidentes domésticos, accidentes laborales, etc.

Otro de los marcos que se deben contemplar, son aquellos pacientes que posean algún tipo de enfermedad que afecte directamente a la integridad de la estructura ósea, como es el caso de la osteogénesis imperfecta, vulgarmente conocida como “huesos de cristal”. Tal y como se comenta en [4], este síndrome hereditario tiene su mayor incidencia en la infancia, teniéndose una probabilidad de padecerlo de 1/30000. Para diagnosticar este tipo de patologías, se realizan inicialmente con las manifestaciones clínicas y las imágenes radiográficas.

En el documento citado, se data el caso de un paciente neonato que, debido a dificultades en el parto por cesárea, presentaba dificultades respiratorias severas y una fractura diafisaria de húmeros y fémur derecho (véase Fig. 1.5 y Fig. 1.6). Aparentemente, según los datos aportados en el propio artículo, los antecedentes prenatales están dentro de los parámetros normales, siendo el tiempo de gestación de 36, 1 semanas, sin presencia de anemia, ni tampoco de infecciones.

Fig. 1.5: Imagen de fractura diafisaria en el fémur derecho del paciente [4].

Fig. 1.6: Imagen de fractura diafisaria en el húmero derecho e izquierdo del paciente [4].

Este caso amplía el abanico de posibles causas de fracturas, tomando en consideración aquellas personas que por motivos meramente casuísticos han heredado una patología como la que se ha comentado durante el caso.

1.2 DIAGNÓSTICO DE FRACTURAS Y FISURAS.

1.2.1 Métodos más usados para la detección de fracturas y fisuras.

Hoy en día, para la detección de fracturas y fisuras, los métodos más comúnmente extendidos son: la radiografía (mediante el uso de rayos X), la tomografía computarizada (TC), la resonancia magnética (RM) y finalmente la ecografía (US) [5]. Pero si algo hay que objetar con respecto a las ya mencionadas, es que el método más extendido y usado hoy en día es la radiografía. Es por ello, que el objetivo principal que persigue este subapartado es esbozar de forma somera el funcionamiento de cada uno de los sistemas mencionados, exceptuando el US debido a que posteriormente se hablará de él con mayor profundidad.

La Radiografía: Este sistema se basa en la creación de imágenes mediante el uso de una fuente ionizante, en este caso de rayos X, que pasa a través del sujeto, proyectando posteriormente sobre una película o sobre un detector digital la imagen resultante. El resultado obtenido se debe al proceso físico de atenuación que sufre dicha radiación al atravesar los tejidos blandos, atenuándose de forma notoria en los huesos, debido a que

estos son capaces de absorber la radiación. Debido a esto último, los huesos se ven perfectamente representados por áreas blancas en la imagen radiográfica. Sin embargo, los tejidos blandos, como los músculos y los órganos, son susceptibles a que la radiación los atraviese con mayor facilidad, es por ellos que aparecen como áreas más oscuras en la imagen [6].

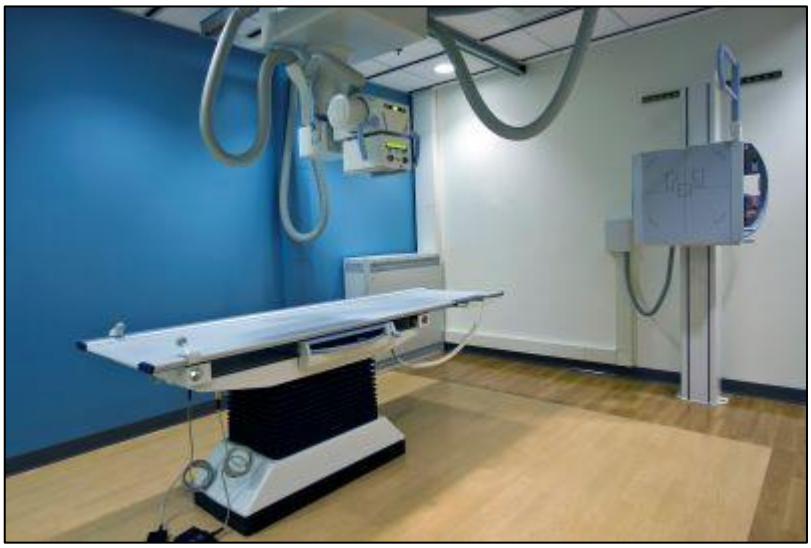


Fig. 1.7: Imagen de un sistema de rayos X.

La Tomografía Computarizada: Esta técnica, al igual que la radiografía, emplea rayos X para realizar la obtención de imagen. Lo que a diferencia del caso visto anteriormente, este genera una imagen tridimensional combinando múltiples imágenes adquiridas en distintas posiciones y perspectivas del sujeto. El sistema consta de un anillo que contiene una fuente de rayos X que gira alrededor del sujeto, y unos detectores que, incorporados a su vez en el anillo, interpretan la imagen recibida del sujeto. Una vez se han recibido todas las imágenes por parte del sistema captador, estas son procesadas mediante un algoritmo que reconstruye la zona anatómica de interés [7].

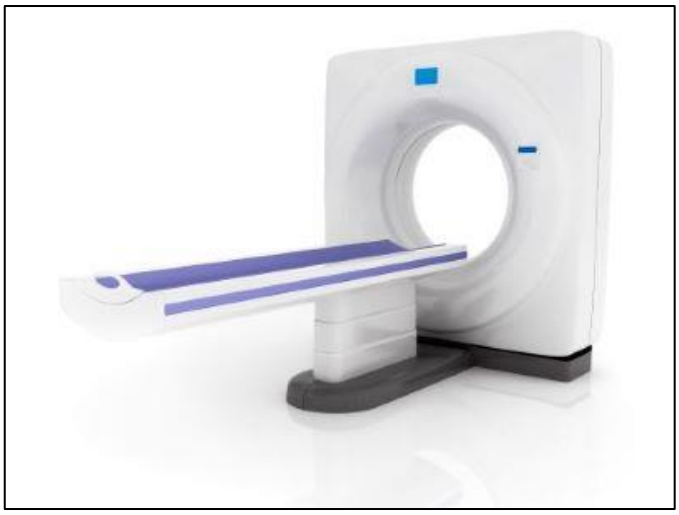


Fig. 1.8: Imagen de un sistema utilizado de tomografía computarizada.

Resonancia Magnética: Este sistema tiene una morfología cilíndrica en la cual, en el centro de este, se sitúa el sujeto bajo estudio. Dicho sistema, mediante un potente imán, crea un campo magnético alrededor del sujeto, produciendo que los átomos de hidrógeno presentes en el cuerpo se alineen en un sentido determinado. Seguidamente, se emiten pulsos de ondas de radio provocando que los átomos de hidrógeno del cuerpo emitan señales. Estas señales son recogidas con antenas especiales y son enviadas a un sistema de procesado destinado a crear imágenes detalladas de las estructuras internas del cuerpo [8].

Fig. 1.9: Imagen de un sistema de resonancia magnética.

1.2.2 Limitaciones de los sistemas

Este apartado tiene como objetivo mostrar algunas de las limitaciones y desventajas que afrontan estos sistemas en su uso diario, ya sea en clínicas u hospitales.

La primera de las limitaciones que debe ser mencionada, es aquella que implica una causa económica; pues sistemas como la resonancia magnética o la tomografía computarizada suponen un alto coste de adquisición y mantenimiento. Esto se podría decir que es una de las desventajas más severas, puesto que en lugares rurales o con una economía insuficientemente desarrollada no se tendrá acceso a algunas de estas modalidades de imagen médica que facilitan el diagnóstico.

Por otro lado, en el caso de la radiografía y la tomografía computarizada el uso de radiación ionizante hace que estas sean un problema a considerar, poniendo así limitaciones en su uso. En este aspecto hay varios artículos de múltiples autores que ponen en tesitura la integridad del usuario después de haber estado expuesto a la toma de imágenes de estos sistemas. Con respecto a este tema, en [9] se describe la forma detallada la historia de cómo han ido evolucionando estas tecnologías en base al tipo de uso que le dan a los rayos X. En el documento mencionado se muestran gráficos que reflejan como con el paso del tiempo, a medida que se ha integrado este tipo de tecnología en nuestra sociedad, la dosis efectiva de radiación per cápita ha aumentado considerablemente en el ámbito de la medicina. A modo de ilustración, se muestra un gráfico extraído del ya referenciado, que hace alusión a lo mencionado.

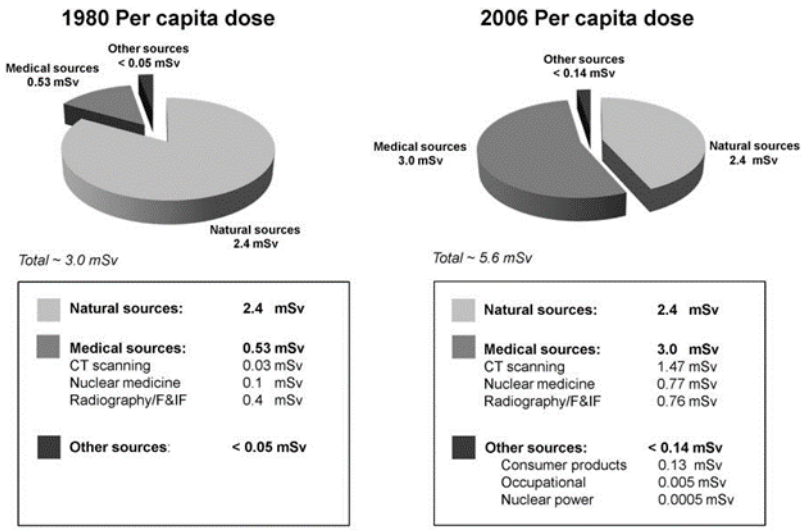


Fig. 1.10: Gráfico representativo de la dosis efectiva de radiación per cápita del año 1980 y 2006. Se muestra un crecimiento notorio en la radiación efectiva total, debiéndose dicho incremento, en su gran mayoría las fuentes médicas (Effective Radiation Dose in Millisieverts (mSv)) [9].

Cabe destacar que la unidad de medida utilizada para la caracterización de estos dos gráficos son los milisieverts (mSv), la cual es una unidad de medida utilizada para expresar la dosis de radiación ionizante absorbida por un ser humano o un objeto expuesto a la radiación. Un millisievert representa una milésima parte del Sievert (Sv), que es la unidad estándar utilizada para medir la dosis equivalente de radiación. Este tipo de medida es utilizada comúnmente para establecer límites de dosis seguros para los trabajadores que están expuestos regularmente a la radiación ionizante.

Una vez mencionado el caso particular que sufren las radiografías y las tomografías, es necesario mencionar una de las limitaciones que posee la resonancia magnética, que siendo de menor gravedad que el caso expuesto anteriormente, tiene una limitación fundamental que deriva del campo magnético creado por el propio sistema. Es de conocimiento general que los campos magnéticos son altamente sensibles a la presencia de metales, dependiendo del tipo de metal que se trate este puede ser capaz de deformar el campo magnético que le rodea. También hay que considerar el caso de los metales que ofrecen reluctancia magnética, los cuales ofrecen resistencia al paso de un flujo magnético, produciéndose así un calentamiento del propio metal. Teniendo en cuenta algunos de los efectos mencionados, y otros más que no se han comentado, es entendible que para los expertos sea un aspecto importante a considerar, sobre todo en aquellas personas que

tienen implantes. Aunque por lo general, en la mayoría de los casos, los implantes suelen ser de titanio, acero inoxidable, aleaciones de cobalto-cromo y aleaciones de níquel-titanio, los cuales son seguros para la realización de una resonancia magnética. Estos metales no son atraídos por el campo magnético y no se calientan durante el examen. Pero hay otro tipo de implantes como los clips de aneurisma cerebral o ciertos tipos de prótesis de oído, que pueden interferir con la calidad de la imagen de la resonancia magnética. Además, si el implante se encuentra cerca de un órgano importante o un área del cuerpo que se está examinando, puede haber preocupaciones adicionales sobre los efectos biológicos de la exposición a la radiación electromagnética [8].

1.3 IMAGEN ULTRASONIDOS

1.3.1 Funcionamiento del Ultrasonido para la captación de imágenes médicas.

Para entender el concepto de ultrasonido, es conveniente conocer primeramente el concepto de sonido en sí mismo. El sonido no es más que una onda mecánica la cual requiere un medio para propagarse, por lo general este medio suele ser el aire, pero el uso de esta onda, trasladada a otros medios, es lo que permitirá abordar el sistema del cual se pretende en este apartado abordar su funcionamiento.

El oído humano, en rasgos generales, es capaz de percibir esta onda mecánica, producto de una perturbación en el medio, en un rango de frecuencias que va desde los 20Hz hasta los 20.000Hz (a esto se le conoce como rango audible). A los sonidos situados en frecuencias superiores a los 20.000Hz es a los que se les conoce por el nombre de ultrasonidos. En un ecógrafo (sistema que utiliza los ultrasonidos para generar imágenes médicas) se generan ultrasonidos mediante el fenómeno físico acuñado como “efecto piezoeléctrico”. Este se basa en la compresión de un material determinado, que, al verse afectado de dicha manera, genera una diferencia de potencial en su superficie, dando lugar a lo que conocemos como corriente eléctrica. Este fenómeno es de carácter reversible, o, dicho de otra forma, si a este material se le aplicara una corriente alterna se consigue que este se comprima y se expanda la frecuencia determinada por la corriente, produciendo así una perturbación en el medio debido a las vibraciones del material. Este fenómeno es aprovechado en multitud de altavoces y micrófonos, pero en este caso, a este material se le aplicarán corrientes de alta frecuencia que acabarán resultando en ultrasonidos.

En el sistema que atañe a este apartado, el material piezoeléctrico se encuentra situado en el cabezal, encargado de realizar, por tanto, la generación de ondas ultrasónicas, como de recibirlas al reflejarse estas en los tejidos que tienen diferente impedancia acústica. Dicha recepción lleva consigo la conversión de dicha mecánica en corriente eléctrica. Finalmente, dicha señal eléctrica es interpretada por el sistema, de tal forma que este es capaz de generar una imagen en escala de grises mediante ellos.

Los elementos por los que está formado este sistema son [10].

Fuente de corriente o Generador: Este es el encargado de enviar pulsos de corriente eléctrica directamente al transductor, el cual, tal y como se comentó anteriormente, se trata de un material piezoeléctrico que al ser estimulado por corriente eléctrica se originarán movimientos de contracción y dilatación en el mismo, produciendo así las ondas de ultrasonido.

Transductor: El transductor está formado por materiales piezoeléctricos, los cuales son cristales (cristales de cuarzo, rubidio, algunas cerámicas…), que al ser excitados mediante corriente eléctrica producen las, ya conocidas, ondas ultrasonidos. Los ultrasonidos reflejados vuelven a llegar nuevamente al cabezal donde están situados estos transductores, y al ser estimulados mecánicamente, generan una señal eléctrica variable. En este apartado podemos encontrar cuatro tipos de transductores utilizados en ecografías, clasificados en base a la disposición de los cristales.

- Sectoriales: Este genera una imagen en forma de triángulo, con una base de emisión de ultrasonidos relativamente pequeña. Esta es comúnmente utilizada para ver estructuras profundas. Su rango de trabajo oscila entre los 3,5 a los 5 MHz.
- Convexos: El cabezal de estos ecógrafos posee una forma curva, y genera una imagen trapezoidal. Comúnmente usado en exploración abdominal. Su rango de trabajo, al igual que en el anterior caso, oscila entre los 3,5 a los 5 MHz.
- Lineales: Como su propio nombre indica, los cristales en este tipo de ecógrafos se encuentran dispuestos de forma lineal en el cabezal del mismo, formando por tanto una imagen rectangular. El ámbito de aplicación para este tipo de sondas suele ser para ver es estructuras y tejidos que se encuentran más cercanos a la superficie como pueden ser: los músculos, los tendones, algunos vasos superficiales, etc. Estos

son los que trabajan dentro de un rango de frecuencias que suelen ser entre los 7,5 y los 13 MHz, pero hay fabricantes que desarrollan transductores de este tipo cuya frecuencia máxima llegan hasta los 20 MHz.

- Intracavitarios: Este tipo de sondas tienen cabezales lineales o convexos, pero lo que realmente los caracteriza es por ser utilizadas para ver estructuras de forma interna como puede ser el caso de las exploraciones intrarrectales o intravaginales.

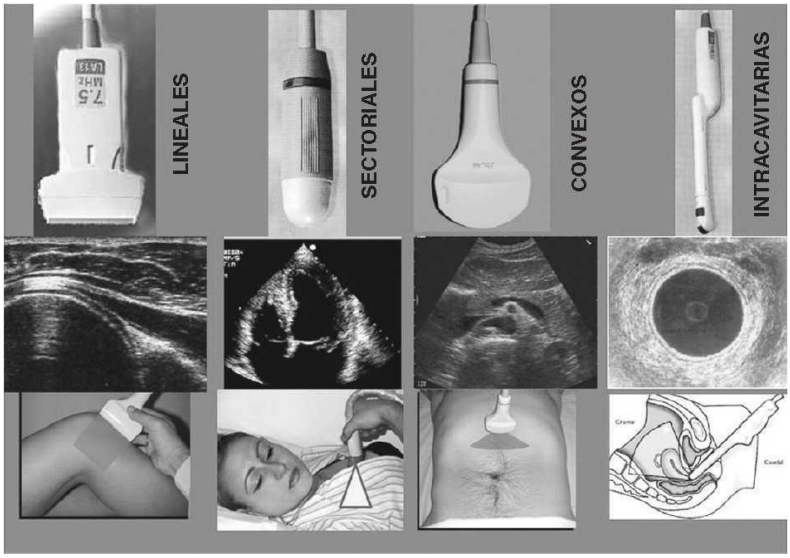


Fig. 1.11: Imagen extraída de, donde se muestran los distintos tipos de ecógrafos con sus respectivos formatos de imagen.

Convertidor analógico-digital: Este se encarga de digitalizar la señal recibida, convirtiendo la misma en información binaria (unos y ceros) para que pueda ser interpretada por el sistema.

Memoria Gráfica: La principal encargada de ordenar toda la información para, una vez recibida, mostrar la misma en una escala de grises a través del monitor.

Registro Gráfico: El registro gráfico es la representación visual resultante de haberse llevado a cabo un análisis al paciente mediante ultrasonidos. Este además proporciona información adicional sobre los distintos tejidos y órganos internos que no se pueden ver en la imagen en tiempo real. También se puede utilizar para medir la amplitud y la duración de las ondas sonoras, lo que puede indicar la densidad o la elasticidad de los tejidos.

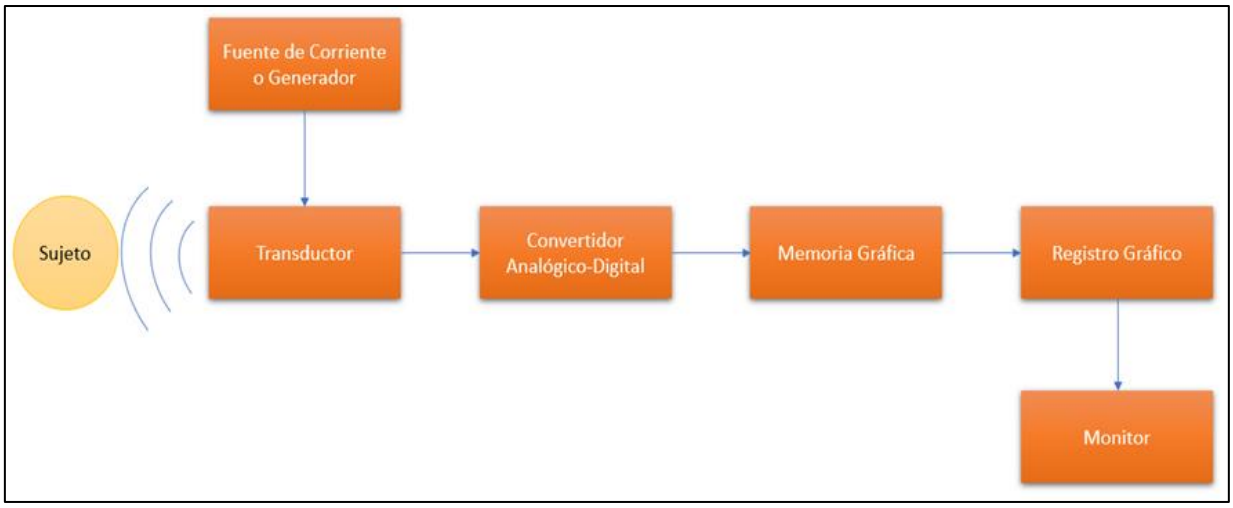


Fig. 1.12: Diagrama de bloques básico de un ecógrafo. En él se muestran todos los sistemas involucrados en el mismo, e incluso al sujeto bajo estudio.

Una vez se ha entendido la mecánica principal de este sistema con los elementos que lo conforman, es necesario explicar la teoría básica que explica cómo se forma el haz de ultrasonido. Y es que realmente, la función del transductor no es la de mandar ultrasonidos de forma continua, sino más bien la de mandar grupos de ciclos de ultrasonidos, conocidos como pulsos. La mecánica principal que domina al haz de ultrasonidos se basa en la alternación entre dos fases: La primera fase en la cual se emiten los ultrasonidos, y la segunda fase, en la cual se esperan recibir los ecos generados por el rebote de las ondas en los distintos tejidos que va encontrando a su paso. Un término a tener en cuenta en este proceso es el de frecuencia de repetición de pulsos (PRF), esta es la frecuencia con la que el generador produce impulsos eléctricos cada segundo. La ya nombrada PRF es la que a fin de cuentas determina de qué forma se reparten los intervalos de tiempo de emisión y recepción. Estos intervalos deben estar ajustados de tal manera de que las ondas de ultrasonido tengan el tiempo suficiente para llegar a la zona de interés y seguidamente volver hasta el cabezal de la sonda antes de que esta emita el siguiente conjunto de pulsos. Una vez llega la señal de vuelta al transductor, esta señal es digitalizada y pasada a la memoria gráfica, de tal manera que después de haber ordenado la información, esta es presentada en pantalla a una tasa de al menos 20 fotogramas por segundo, transmitiendo así la información a tiempo real.

1.3.2 Ventajas y desventajas del uso de sistemas de Ultrasonidos.

Como ya se ha comentado, a priori el funcionamiento se basa en el aprovechamiento de un principio físico básico que, a grandes rasgos, es bastante robusto y fiable en cuanto a su comportamiento. Pero esta caracterización no implica que el sistema sea completamente infalible, es por ello objeto de este apartado comentar las desventajas y ventajas que tiene la utilización de este tipo de sistemas.

➔ *Ventajas*

Entre las ventajas que poseen estos tipos de sistemas podemos encontrar algunas de las destacadas por [10].

a. La no utilización de radiación ionizante hace de este un sistema no invasivo. Por otro lado, está comprobado a nivel científico que la utilización de esta tecnología con bajas intensidades no tiene efectos biológicos dañinos sobre el paciente, como por el contrario sí podría suceder con la tomografía.
b. En concordancia con lo anteriormente comentado, este sistema permite efectuar controles en el paciente repetidas veces ya que, tal y como se comentó en el apartado anterior, el uso repetido de esta no supone efectos nocivos en el individuo bajo estudio.
c. A nivel económico se puede destacar que este equipo no requiere de un recinto construido especialmente para su utilización, como puede ser el caso de la resonancia magnética, de la máquina de rayos X, o la tomografía. En consecuencia, esto hace que su coste sea mucho menor. Otros aspectos que lo hacen más versátil en este ámbito, tanto el mantenimiento que precisa, como el coste del equipo en si mismo es mucho menor que el de los ya mencionados.
d. Su portabilidad es otro de los aspectos a tener en cuenta en este sistema, es decir, permite ser llevado hasta donde esté ubicado el paciente. Actualmente existen ecógrafos que permiten ser conectados a dispositivos móviles para su uso.
e. La visualización de imágenes a tiempo real es algo de lo que pocos sistemas destinados a la obtención de imagen médica pueden presumir, es por ello que esta peculiar característica cobra un gran valor dentro del marco de la medicina. Es por ello que es comúnmente utilizada para realizar punciones dirigidas.

➔ *Desventajas*

Entre las desventajas o limitaciones de este tipo de sistemas, se pueden destacar:

a. La ecografía es comúnmente utilizada para detectar lesiones, pero puede suponer una dificultad a la hora de clasificar y diferenciar el tipo de lesión bajo estudio mediante el uso de esta tecnología. Este problema, tal y como se comenta en el ya citado artículo, es muy común a la hora de llevar a cabo la diferenciación de tumores.
b. La escasez de profesionales capaces de realizar interpretaciones en las imágenes tomadas por ultrasonidos, es uno de los problemas latentes, puesto que no todos los profesionales en el sector médico son capaces de efectuar la interpretación conveniente para poder detectar cualquier tipo de anomalía en el paciente.

1.3.3 Aplicaciones actuales

En el marco actual, por lo que mayoritariamente se conocen estos sistemas de ultrasonidos entre el público no especializado, es por ser comúnmente utilizados en la práctica de realizar seguimientos en los embarazos, pero realmente esta es utilizada para realizar multitud de diagnósticos en distintas zonas del cuerpo, algunas de estas pueden ser: ecografía del corazón (conocida también como ecocardiografía), ecografía de la cavidad abdominal (una de las más comunes), ecografía del tracto urinario, etc. Aún así, esta tecnología está por evolucionar aún más, y utilizarse en diagnósticos donde comúnmente se han utilizado otros sistemas para ello. En concreto, en concordancia con el tema de este proyecto, esta tecnología tal y como se menciona en varios de los artículos ya citados, es utilizada también para el diagnóstico de fracturas óseas. Su aplicación no es muy común debido a que otros sistemas como los sistemas de imagen radiográfica, de resonancia magnética o de tomografía computarizada, ofrecen una mayor resolución y una mejor interpretación para el profesional encargado de llevar el estudio. No obstante, tal y como se redacta en el estudio estadístico realizado en [11], la ecografía tiene un alto rendimiento a la hora de detectar fracturas, llegándose así a la conclusión de que este sistema es susceptible de ser utilizado como primera fase de evaluación en estos casos.

1.4 INTELIGENCIA ARTIFICIAL

1.4.1 Fundamentos básicos de redes neuronales

Muchos fueron los investigadores que se adelantaron en el desarrollo de modelos matemáticos de neuronas y Redes Neuronales allá por los años 40 y 50 [12], entre los cuales podemos destacar algunos como McCulloch y Pitts (1943), Householder y Landahl (1945), Kleene (1956), Von Neumann (1956), Culbertson (1956), entre otros. Pero realmente fueron Farley y Clark (1954) y Rochester, Holland, Haibt y Duda (1956) los que desarrollaron modelos de simulación en computadora de neuronas y redes neuronales, o dicho en otras palabras, pusieron en práctica todo lo que los predecesores en este ámbito habían teorizado en su día, llevándolo por tanto al ámbito en el que actualmente se desarrolla.

Un gran hito en el desarrollo de redes neuronales vino de la mano de Frank Rosenblatt en el año 1958, con el desarrollo del **perceptrón**. Según se enuncia en [12], esta fue la primera red neuronal artificial especificada con precisión y orientada computacionalmente. Frank Rosenblatt cambio el paradigma en su desarrollo del perceptrón, pues desechó el punto de vista de anteriores investigadores, ya que estos veían al cerebro como un sistema de computación lógica, mientras que el mismo Rosenblatt lo atribuyó con el comportamiento de un clasificador, basado en la asociación de respuesta de clasificación a estímulos específicos.

Para entender cómo funciona el perceptrón primeramente se debe entender el modelo matemático que explica el funcionamiento de lo que se conoce como neurona artificial. Para ello, se pretende hacer una comparación con una neurona real, asemejando sus distintas partes con la de la neurona artificial. Se muestra a continuación la imagen de una neurona.

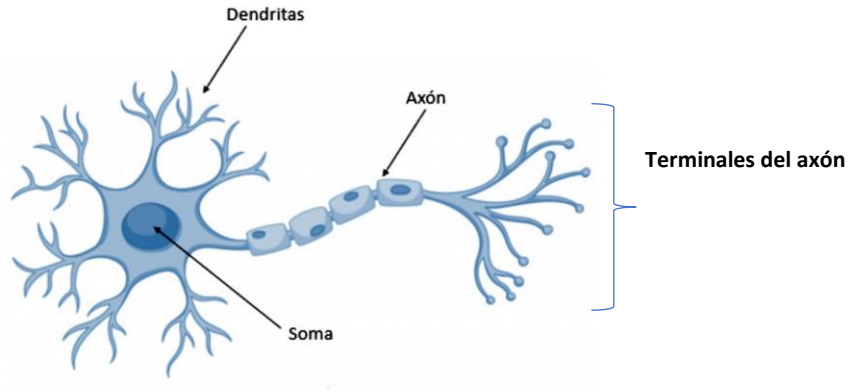


Fig. 1.13: Esquema con las partes de una neurona.

Las partes mostradas son suficientes para entender el funcionamiento de este tipo de células.

Primeramente, se tienen las dendritas, las cuales conforman los receptores de la neurona, permitiendo recibir mediante este los estímulos transmitidos por otras neuronas. En el soma, las sinapsis actuadoras disparan un impulso de salida que es conducido a través del axón, llegando así el extremo opuesto de la neurona, el cual, está dividido en múltiples fibras que le permite comunicarse con otras neuronas, o también con órganos efectores o motores. Visto esto, se muestra a continuación una imagen, extraída de [12], que muestra de forma teórico-conceptual cómo es una neurona artificial.

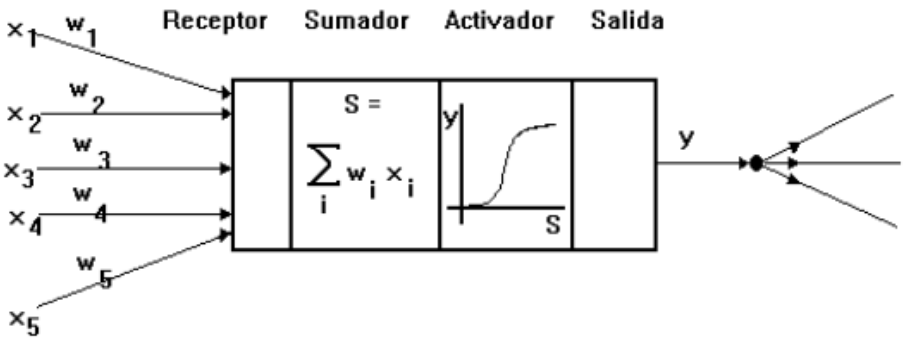


Fig. 1.14: Ilustración conceptual de las partes de una neurona artificial.

De forma análoga a las dendritas se tiene la parte receptora de la neurona artificial, es decir, aquella que tiene como entradas x_1, x_2, x_3, …, x_n. Estas entradas son amplificadas o atenuadas por un factor w_i, al cual se le conoce por el nombre de "pesos". Seguidamente encontramos lo que podríamos llamar "el soma de la neurona artificial", en el que podemos distinguir dos partes: el sumador en el que efectúa la suma algebraica ponderada de las señales de entrada, ponderándolas de acuerdo con su peso, y la función activadora, la cual vendría a simular el proceso de sinapsis. La función activadora aplica una función no lineal con un umbral a la salida del sumador, en el que se decide si se dispara una salida o no. Finalmente, se tiene un elemento a la salida el cual es pasado a la neurona o neuronas inmediatamente consecutivas.

Habiendo entendido esto, se puede profundizar de mejor forma en el funcionamiento del perceptrón en sí mismo. Este se basa en una neurona procesadora la cual tiene como entrada distintas señales x_i, donde i toma valores de 0 a 'n'. Dichas entradas son ponderadas con su peso correspondiente e inmediatamente sumadas en el propio sumador, tal y como se explicó anteriormente. Una vez se ha efectuado la suma ponderada, el activador emplea una función escalón umbral en el que si la suma ponderada es igual o mayor a **U** esta tendrá por consecuente una salida **y**, siendo la siguiente de la manera que se muestra.

$$Para\ \boldsymbol{S >= U} \rightarrow \boldsymbol{y = 1}$$
$$Para\ \boldsymbol{S < U} \rightarrow \boldsymbol{y = 0}$$

En base a esta idea del perceptrón, se originó lo que se conoce como el modelo de perceptrón multicapa. Este es un tipo de red neuronal artificial compuesta por múltiples capas de neuronas interconectadas.

El perceptrón multicapa surgió en la década de 1980 como solución a las limitaciones del perceptrón simple, que solo puede aprender funciones linealmente separables. En comparación al perceptrón simple, el perceptrón multicapa es capaz de aprender funciones no lineales y resolver problemas más complejos.

La estructura del perceptrón multicapa está formada de al menos tres capas: una capa de entrada, una o más capas ocultas y una capa de salida. Cada capa está compuesta por

unidades llamadas neuronas o nodos, y cada neurona está conectada con neuronas en las capas adyacentes mediante conexiones ponderadas. Además, cada neurona tiene una función de activación no lineal que introduce la no linealidad en la red.

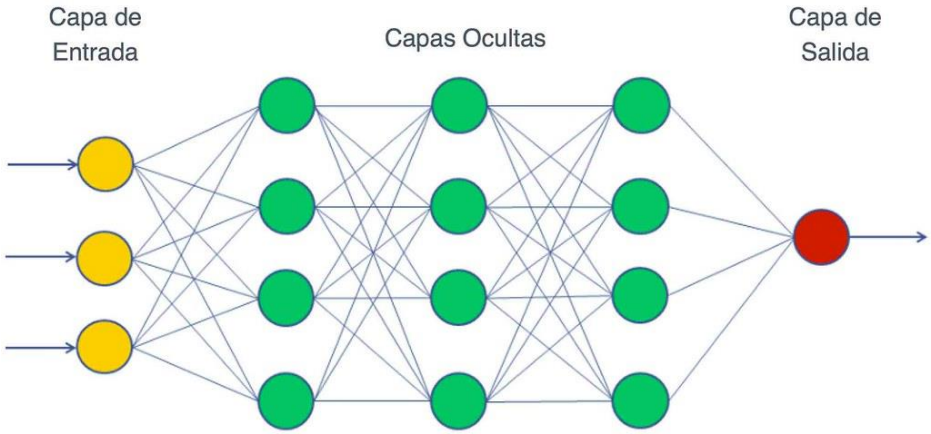


Fig. 1.15: Estructura del perceptrón multicapa [13].

Otra de las cosas a tener en cuenta son lo que se conoce como “la ley de aprendizaje”. Para entender lo que hay detrás de este término, se partirá de una red de perceptrones simples, la cual tan solo tiene dos capas, tal y como se muestra.

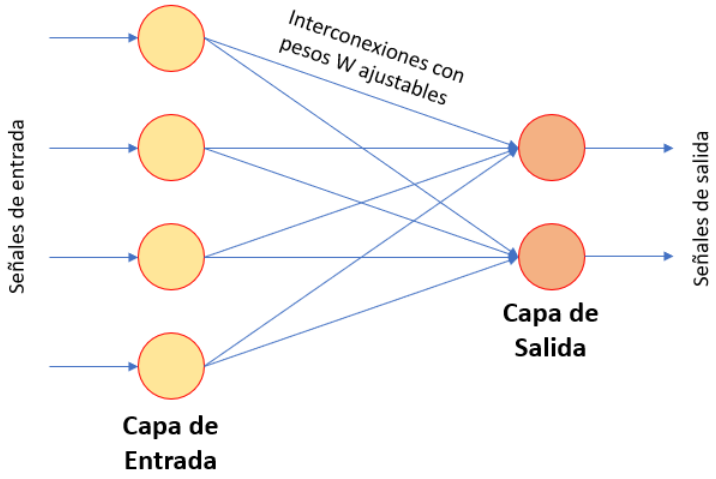


Fig. 1.16: Red simple de perceptrones.

Esta red está formada de tan solo una capa de entrada, y una capa de salida. El conjunto de neuronas que se encuentra en la capa de entrada está formado por neuronas simplemente reproducen la entrada que reciben, mientras que las dos neuronas que

conforman la capa de salida son dos perceptrones como los que se ha descrito en este documento posteriormente. La aplicación de la Ley de aprendizaje en esta red es la encargada de ir ajustando los pesos, de tal manera que se obtenga la salida deseada con respecto a un determinado conjunto de entradas. Para ello se llevaba a cabo un patrón de entrenamiento para el perceptrón, el cual se basa en un vector de entrada **x** con un vector **y** que representa la salida deseable. En base a la diferencia que se encuentre con respecto a la salida deseada y a la obtenida se reajustarán los pesos para así lograr el objetivo previamente asumido. Una de las reglas más usadas para hacer este reajuste, y a la par una de las más sencillas, es la Regla Delta de Widrow y Hoff. A continuación, se muestra la ecuación que define dicha regla.

$$w_1 = w_0 + \sigma(t - y)$$

Ec. 1.1: Regla Delta de Widrow y Hoff.

Esta ecuación expresa que el nuevo peso W_1 será igual al peso anterior W_0, más la multiplicación de la razón de aprendizaje σ con la diferencia entre la salida deseada t y la salida obtenida y. De la anterior se tiene una fórmula derivada que, en caso de tener a la entrada del perceptrón un vector de unos y ceros, el peso solo se modificará en caso de que la entrada valga 1.

$$w_1 = w_0 + \boldsymbol{x}.\sigma(t - y)$$

Ec. 1.2: Ecuación derivada de la Regla Delta de Widrow y Hoff.

Cabe destacar que estos pesos son fijados aleatoriamente antes de empezar el entrenamiento, por lo que estos se van modificando a medida que avanza el entrenamiento.

Durante la historia, el perceptrón no ha sido el único modelo de neurona que ha existido, posterior a este han ido surgiendo nuevos modelos, como puede ser el caso de Adaline (Adapative Linear Neuron) desarrollado por Bernard Widrow y Marcian E. Hoff(Widrow &

Hoff 1960). Algunas de las diferencias más notorias se pueden observar en el esquema del modelo extraído de [12].

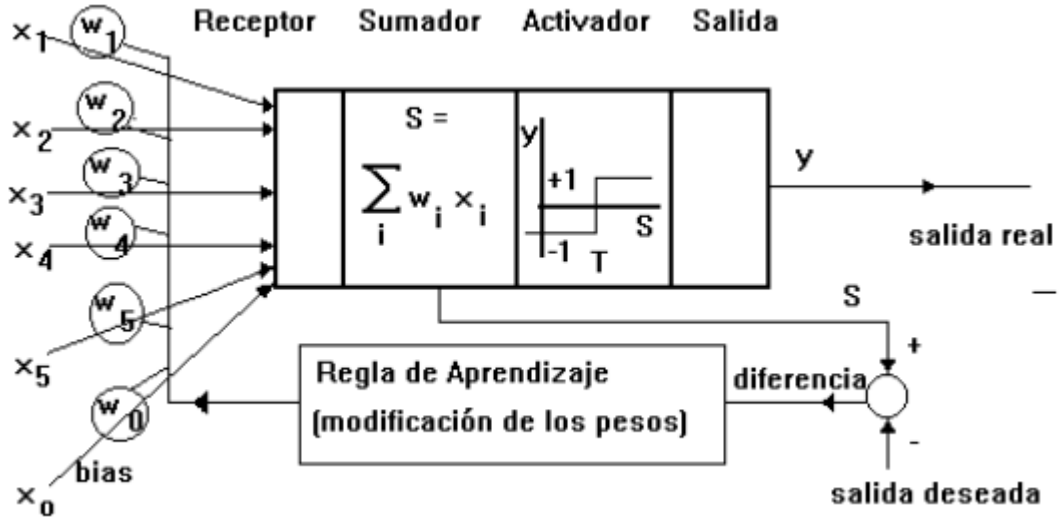


Fig. 1.17: Esquema de modelo de neurona Adaline.

Algunas de las diferencias más relevantes es la entrada x_0 la cual tiene un valor constante que por lo general suele ser igual a 1. Esta entrada x_0 modulada por su peso tiene una estrecha relación. Otra de las cosas a destacar, es que la señal de salida es +1 o -1, según si la suma ponderada **S** es mayor o menor al umbral de disparo, además también cambia la regla de aprendizaje, en la cual se mira principalmente la diferencia entre la salida de la sumatoria ponderada **S** y la salida deseada dada por el vector que previamente se mencionó para el entrenamiento. Se podría entrar más en detalle con respecto a la ecuación que domina el comportamiento de esta regla de aprendizaje, pero no es objetivo de este texto llegar a ello. Posteriormente a Adeline y al perceptrón, han ido surgiendo más modelos de neuronas en pro del desarrollo de este campo, pero el fundamento de funcionamiento y entrenamiento de redes neuronales no deja de ser el mismo para todos y cada uno de los caso, siendo evidente de que puedan haber ligeras variaciones.

Antes de darse por finalizado este punto, es necesario entender los tipos de aprendizajes de las redes neuronales, los cuales son fundamentalmente dos tipos.

- Aprendizaje supervisado: Se requiere la presencia de una persona que conoce la respuesta correcta ante cada vector de entrada y, con la respuesta real de la red, genera un vector de error que retroalimenta a ésta. Finalmente, a medida que se actualizan estos pesos el error tiende a desaparecer.

- Aprendizaje no supervisado: Este tipo de entrenamiento no requiere tutor externo, ni tampoco requiere de datos etiquetados para su entrenamiento. Simplemente se limita a intentar descifrar patrones o estructuras ocultas en los datos. Es decir, su objetivo se basa en encontrar estructuras o relaciones intrínsecas de datos sin conocimiento previo. Los enfoques que se pueden llegar a adquirir en estos tipos de entrenamientos son: el *clustering* o agrupamiento (agrupar los datos en grupos o categorías según su similitud), generación de datos (aprender de la distribución de los datos y generar ejemplos que se parezcan a los originales) y reducción de dimensionalidad (encontrar una representación compacta de los datos al reducir el número de dimensiones de entrada) los cuales se utilizan para proyectar datos en un espacio menor.

1.4.2 Frameworks y librerías para el desarrollo y entrenamiento de redes neuronales.

En este apartado, se pretende detallar primeramente los tres *frameworks* más utilizados para el desarrollo de redes neuronales y, seguidamente, se hablarán de algunos otros menos conocidos pero que tienen cierta relevancia en algunos ámbitos [14]. Se procede a continuación a hablar de ellos.

KERAS

Keras [15] es una biblioteca de código abierto de *deep learning* basada fundamentalmente en Python. Esta funciona como una API de alto nivel destinada al desarrollo de redes neuronales de forma relativamente sencilla. Su simplicidad recae en el hecho de disponer una interfaz de programación de aplicaciones (API, como ya previamente se mencionó) que permite a los usuarios interactuar con ella mediante el uso del lenguaje de programación Python, ofreciendo a su vez un alto nivel de abstracción. También cabe mencionar que se dispone de múltiples *back-ends* (parte del software que realiza los cálculos o procesos y devuelve los resultados) con propósitos meramente computacionales, es decir, permite al usuario elegir entre diferentes opciones para realizar los cálculos, lo que puede afectar la velocidad, la eficiencia y otros factores del procesamiento. Algunos de los *back-ends* que soporta Keras son: TensorFlow, Theano, CNTK (Microsoft Cognitive Toolkit), PlaidML, entre otros.

De estos cinco *frameworks*, TensorFlow ha adoptado Keras como su API oficial de alto nivel. Algunas de las ventajas de Keras se pueden resumir brevemente en los siguientes puntos.

- Fue desarrollada con el propósito de ser fácilmente usable por los usuarios. Reduce líneas de código, además de ofrecer una detección de errores sumamente clara de entender.
- El tiempo de desarrollo en Keras es mucho menor. También proporciona una variedad de opciones de implementación según las necesidades del usuario.
- Los lenguajes con un alto nivel de abstracción y funciones integradas suelen ser lentos y crear funciones personalizadas puede ser una tarea difícil, pero Keras se ejecuta sobre TensorFlow y es relativamente rápido.
- Dispone de una comunidad amplia con todo tipo de ayudas útiles para nuevos usuarios y para usuarios enmarcados dentro del ámbito de la investigación.

Pero, tanto como se habla de las ventajas de lo fácil e intuitivo que es usar Keras, se debe mencionar que este tiene algunas limitaciones que hacen que Keras no sea el *framework* predilecto a la hora de trabajar con redes neuronales. Una de las desventajas más destacadas es que Keras tiene muchas dificultades a la hora de programar a bajo nivel, es decir, es susceptibles de sufrir errores continuados si queremos programar operaciones o funciones que impliquen una programación más cercana al hardware del sistema que se esté utilizando. Y es que Keras no fue diseñada para realizar este tipo de funciones. Otro de los grandes problemas de los cuales sufre este famoso *framework*, es que necesita desarrollar mejoras en el preprocesamiento de datos, ya que comparado con otras librerías como puede ser el caso de scikit-learn, esta no dispone de las suficientes herramientas como para que la experiencia del usuario se completa y satisfactoria.

TENSORFLOW

TensorFlow [16], desarrollado por Google en 2015, es un *framework* de código abierto, el cual es utilizado particularmente en el ámbito de IAs y aprendizaje automático. Este ofrece las herramientas necesarias para que el usuario desarrolle modelos de *deep learning* (aprendizaje profundo), para posteriormente entrenarlos usando grandes conjuntos de datos, utilizando técnicas tales como redes neuronales convolucionales, redes neuronales recurrentes y aprendizaje por refuerzo. La fama de TensorFlow no solo reside en el

desarrollo de modelos de *deep learning* utilizando redes neuronales, sino que este, también ofrece herramientas de visualización y análisis de los resultados del modelo.

Según mencionan en su propia web, esta plataforma ofrece varios niveles de abstracción que se adaptan a la necesidad del usuario ofertándose como una plataforma *end-to-end* (en español conocida como "de principio a fin"). Esto significa que las herramientas provistas por este *framework* cubren todo el ciclo de desarrollo de una aplicación. En el caso de TensorFlow estas herramientas son las que se citan a continuación.

- **Preprocesamiento de datos:** Esta etapa se centra en la carga y transformación de datos en formatos que puedan ser utilizados por los modelos de aprendizaje automático.
- **Desarrollo y entrenamiento de modelos:** Tal y como se mencionó anteriormente, TensorFlow proporciona una amplia variedad de funciones y herramientas para desarrollar y entrenar modelos de aprendizaje automático, incluyendo funciones de optimización, capas de redes neuronales, y algoritmos de entrenamiento. Para desempeñar esta función, Tensorflow ha incorporado dentro de su amplia biblioteca una API de alto nivel de Keras en pro de ayudar a que los primeros pasos dentro de este paquete de librerías sean lo más sencillo posible.
- **Evaluación y ajuste de modelos:** La plataforma permite evaluar y ajustar modelos de aprendizaje automático, incluyendo funciones de evaluación, validación cruzada, y herramientas para detectar y corregir problemas de sobreajuste (*overfitting*).
- **Despliegue de modelos:** Uno de los puntos fuertes que ofrece TensorFlow, es que este permite desplegar sus modelos de aprendizaje en una amplia variedad de plataformas, incluyendo dispositivos móviles y la nube.

Aunque sea de sumo interés, conocer únicamente las herramientas no es suficiente como para definir objetivamente cómo es este *framework*. Es por ello que a continuación se resumen algunas de sus ventajas en los siguientes puntos.

- Este no está limitado para un tipo específico de dispositivo. TensorFlow está pensado de tal manera que pueda ser eficiente en cualquier tipo de sistema, ya sea un dispositivo móvil o cualquier otro tipo de sistema más complejo.

- Uno de los puntos fuertes de TensorFlow, es que dispone de un potente sistema de visualización, que permite incluso, mediante grafos y funciones, visualizar todo el flujo de datos, aportando la mayor información posible al usuario.
- TensorFlow, al igual que otros muchos *stacks* de librerías, permite al usuario elegir entre realizar todo el procesamiento de datos en la CPU o en la GPU.
- Otra de las cosas por la que destaca este *framework*, es que es compatible con diversos lenguajes de programación como Python, C++, JavaScript, entre otros. Esto permite a los usuarios trabajar en un entorno en el que se sienten cómodos.

Aunque las ventajas de utilizar TensorFlow son las suficientes como para tenerlo como biblioteca predilecta para el desarrollo de redes neuronales, cabe destacar algunos de sus puntos más débiles. Entre estos puntos, se puede destacar que TensorFlow dispone de un conjunto de características muy limitado para los usuarios que utilizan el sistema operativo de Windows, por lo general, este tiene su totalidad para aquellos usuarios que utilicen el mismo en el sistema operativo Linux. Otra de los puntos negativos a destacar de TensorFlow, es que solo dispone de soporte para las GPUs de Nvidia, por lo que aquellos usuarios que dispongan de una GPU de la línea de AMD Radeon, no pueden gozar de la capacidad de procesamiento de la que dispone sus GPU para procesar y ejecutar su red neuronal.

PYTORCH

Pytorch [17] es un *framework* de *machine learning* (ML) desarrollado por Facebook, basado en la librería Torch. Este *framework* a diferencia de TensorFlow, tiene un enfoque más cercano a la forma de programar en Python y utiliza un paradigma de programación orientado a objetos, consiguiendo de esta manera que sea relativamente fácil de aprender y usar para la mayoría de los desarrolladores de aprendizaje automático. PyTorch utiliza las estructuras de datos y las operaciones que ya están presentes en Python, lo que lo hace más intuitivo y fácil de entender para los programadores de Python. Además, se utilizan clases y objetos para representar redes neuronales y tensores, lo que permite una mayor modularidad y reutilización de código. Esto permite a los usuarios definir sus propias clases y métodos para personalizar su flujo de trabajo y realizar tareas específicas de manera más eficiente.

Tal y como informa la propia web de Nvidia: “PyTorch se distingue por su excelente soporte para GPU y su uso de diferenciación automática de modo inverso, que permite modificar los gráficos de cálculo sobre la marcha”. Para entender bien lo que quiere decir esto, debemos saber que quiere decir la diferenciación automática de modo inverso: es un método utilizado en el aprendizaje automático para ajustar los parámetros de un modelo de forma que se minimice la función de pérdida, que mide la diferencia entre las predicciones del modelo y los valores reales de los datos de entrenamiento. Esto implica calcular el gradiente de la función de pérdida con respecto a cada uno de los parámetros del modelo. La forma de realizar esto es utilizando la regla de la cadena para calcular las derivadas parciales de la función de pérdida con respecto a cada una de las capas del modelo. Es decir, a diferencia de Keras, nos permite inferir directamente en el comportamiento de la GPU a la hora de procesar los datos, algo que en Keras, como bien se explicó en este mismo documento, no ocurre debido a su poco desarrollo en el ámbito del bajo nivel. Esta diferenciación automática brinda a Pytorch la ventaja de que sea relativamente simple de depurar y se adapte bien a ciertas aplicaciones, como las redes neuronales dinámicas. Algunas de las ventajas más destacables de Pytorch se pueden resumir a continuación en los siguientes puntos.

- Ofrece una estructura de código sencilla y fácil de aprender basada en Python.
- Tal y como se comentó previamente Pytorch dispone una estructura de código fácilmente depurable con multitud de herramientas para facilitar esta labor.
- Ofrece escalabilidad, además de ser fácilmente soportable en plataformas en la nube.
- Tiene soporte gráfico computacional en tiempo de ejecución.

Por lo general, se asemeja bastante a lo ya visto con TensorFlow, por no decir que, actualmente, estos dos *frameworks* son los dos más utilizados a la hora de desarrollar redes neuronales. Por otro lado, algunas de las desventajas que se podrían destacar de este *stack* de librerías son algunas como su poco desarrollo comunitario en comparación con TensorFlow. Cabe destacar que esto es debido a que Pytorch, tal y como se conoce actualmente, es relativamente nuevo, por lo que aún está en proceso de seguir mejorando y creciendo. No obstante, Pytorch ha servido como base para otros *frameworks* de

desarrollo de redes neuronales como es el caso de MONAI, del cual se procede a hablar a continuación.

MONAI

MONAI [18](*Medical Open Network for AI*) es un *framework*, que como bien se explicó de forma escueta en el anterior apartado, tiene como base principal el *stack* de librerías de Pytorch. Este es un *framework* o marco de trabajo de código abierto específico para la creación de aplicaciones de inteligencia artificial en el campo de la medicina y la salud. MONAI proporciona una amplia variedad de herramientas y componentes para la creación de algoritmos de aprendizaje profundo (*deep learning*) y el procesamiento de imágenes médicas, como la segmentación de estructuras anatómicas, la clasificación de patologías, la regeneración de imágenes o la visualización 3D. Como bien se puede intuir, este se encuentra desarrollado en Python, y cuenta con una amplia documentación y una comunidad activa que contribuye al desarrollo y mejora continua de MONAI.

MONAI se beneficia de las ventajas que ofrece PyTorch, como la posibilidad de ejecutar operaciones en GPU para acelerar el procesamiento, un motor de autodiferenciación para calcular automáticamente los gradientes y la posibilidad de construir modelos complejos utilizando las redes neuronales como bloques de construcción fundamentales. Además, MONAI agrega sus propias funcionalidades y herramientas específicas para el procesamiento de imágenes médicas, lo que lo convierte en un *framework* muy completo y útil para la comunidad de la salud.

Este al igual que TensorFlow, considera de vital importancia disponer de las herramientas suficientes para cubrir lo que se conoce como el "*end-to-end workflow*", tal y como se explicó en el apartado destinado a TensorFlow. Es por este motivo que MONAI, tal y como exponen en su propia web, han desarrollado tres herramientas para cubrir todo este ciclo, conocidas como MONAI Label, MONAI Core y MONAI Deploy. Cada una de ella desempeña distintas funciones.

- **MONAI Label:** Es una herramienta que permite la creación de etiquetas o anotaciones para imágenes médicas. Esta proporciona una interfaz de usuario gráfica (GUI) para la creación de etiquetas o anotaciones de manera fácil y rápida, utilizando diversas herramientas de dibujo y edición. Además, MONAI Label es

altamente configurable, permitiendo adaptar la herramienta a diferentes tareas y necesidades específicas de cada proyecto. Esta además dispone de un asistente basado en IA con el objetivo de reducir tiempo y esfuerzo en la anotación de nuevas bases de datos.

- **MONAI Core:** Este es lo que se conoce como el núcleo de MONAI. Este proporciona una amplia variedad de herramientas y componentes para la creación de algoritmos de aprendizaje profundo (*deep learning*) y el procesamiento de imágenes médicas, como la segmentación de estructuras anatómicas, la clasificación de patologías, la regeneración de imágenes o la visualización 3D. Siendo específicos, sus funcionalidades principales se demarcan dentro de la lectura y escritura de imágenes médicas, la transformación y normalización de datos de imágenes, la creación de modelos de aprendizaje profundo, la evaluación de los modelos, la implementación de algoritmos de segmentación y otras tareas comunes en el procesamiento de imágenes médicas.
- **MONAI Deploy:** Por lo general, la implementación de modelos de aprendizaje profundo en producción puede ser un proceso complejo y desafiante, que implica aspectos como la selección de infraestructura, la gestión de recursos, la escalabilidad, el monitoreo y la seguridad. Es por ello por lo que MONAI Deploy proporciona una solución integral para simplificar este proceso y acelerar la puesta en marcha de aplicaciones de inteligencia artificial en el campo de la salud. Esta además de lo mencionado, proporciona una interfaz de usuario intuitiva para la configuración y monitoreo de los modelos en producción.

Theano

Este es un *framework* de aprendizaje profundo [14] de código abierto desarrollado por el Laboratorio de Ciencias de la Computación e Inteligencia Artificial (CSAIL) del MIT (Instituto Tecnológico de Massachusetts), destacando por ser uno de los pioneros en este ámbito. Theano permite a los usuarios definir, optimizar y evaluar expresiones matemáticas que involucran arreglos multidimensionales, con una sintaxis similar a NumPy. En particular, se especializa en la creación de redes neuronales, ya sea para tareas de clasificación, regresión, reconocimiento de imágenes o procesamiento de lenguaje natural, entre otras

aplicaciones de aprendizaje profundo. Aunque a priori parece ser una herramienta muy potente para el desarrollo y trabajo sobre redes neuronales esta ya no está en desarrollo activo desde septiembre de 2017, pero muchos de sus conceptos y técnicas se han incorporado en otros *frameworks* de aprendizaje profundo, como TensorFlow y PyTorch.

1.4.3 Mapas de activación en redes neuronales

Un mapa de activación, también conocido como mapa de calor (en inglés conocido como *activation map* o *heatmap*), es una herramienta estadística y visual para representar datos. O, dicho de otra manera, a cada valor de un conjunto total de muestras se le asigna un color específico, para que así se pueda evaluar visualmente su distribución en un plano 2D o 3D. Esta técnica es comúnmente utilizada para poder estudiar de forma relativamente sencilla patrones y tendencias en conjuntos grandes de datos, siendo particularmente útiles para resaltar áreas de alta o baja actividad en un conjunto de datos. Los mapas de activación también se utilizan en diversas aplicaciones, como el análisis de tráfico web, el análisis de redes sociales y la visualización de datos climáticos [19].

Aplicado en el ámbito en el que se desarrolla este apartado, los mapas de activación no solo permiten realizar una valoración estadística de los resultados obtenidos por la red neuronal, sino que también sirve de gran apoyo para evaluar el rendimiento y entender el funcionamiento del sistema en todo momento. En el caso de redes neuronales destinas a clasificación de imágenes, es interesante en qué áreas de la imagen, se está fijando la red para dictaminar la clasificación del mismo. Estos mapas de activación pueden ser de interés por parte del desarrollador del sistema, como por parte del usuario. También cabe mencionar que no existe un mapa de activación concreto para la representación visual de datos, es decir, podemos encontrar mapas de activación que juegan con transiciones de colores en toda su gama, depende del desarrollador utilizar uno u otro.

1.5 CLASIFICACIÓN DE IMÁGENES CON IA

La clasificación de imágenes con inteligencia artificial es un campo que ha experimentado un grandísimo avance en estos últimos años, siendo tal su repercusión que ya podemos encontrar en la red multitud de modelos entrenados orientados a la clasificación en distintas áreas, siendo el ámbito médico uno de estos. Para desempeñar esta labor se suelen emplear redes neuronales convolucionales (CNN) para su entrenamiento. Las CNN

son un tipo de arquitectura de redes neuronales destinadas esencialmente al procesamiento de imágenes, siendo altamente efectivas, no solo en clasificación de imágenes, sino también en detección de objetos y segmentación semántica. Según se explica en [20], las CNN basan su funcionamiento en una distribución de capas ocultas con una jerarquía concreta. Esta jerarquía se distribuye de tal forma que se pueden apreciar dos fases bien diferenciadas.

- En primer lugar, se encuentran una serie de capas destinadas a detectar las propiedades básicas de imagen sujeta a análisis.
- En segundo lugar, se encuentran capas especializadas en el reconocimiento de formas más complejas, como pueden ser siluetas, objetos cotidianos, rostros, etc.

Para poder llevar a cabo esta funcionalidad que tanto caracteriza a las CNN, estas necesitan trabajar con imágenes normalizadas. Las imágenes 2D se representan matemáticamente como un conjunto de matrices que toman valores entre 0 y 255. Suponiendo que se tiene una imagen RGB, se representaría mediante un conjunto de 3 matrices correspondiente a cada uno de los canales, la normalización de esta imagen correspondería a la conversión de ese rango, de tal manera que su valor mínimo y su valor máximo sean 0 y 1, respectivamente. Una vez se ha llevado a cabo este paso propio, se ejecuta la operación que caracteriza a este tipo de redes neuronales, estas no son ni más ni menos que las convoluciones. Esta convolución consiste en coger grupos de píxeles cercanos de la imagen de entrada, e ir realizando la convolución con una matriz más pequeña a la que se conoce como "*kernel*", el cual realiza una función de filtro. Este kernel recorre toda la matriz de entrada que conforman las neuronas de entrada realizando la convolución, y generando por tanto una nueva matriz de salida (Fig. 13). Esta matriz de salida será por tanto la nueva capa de neuronas ocultas. Como condición se estable que la convolución solo se llevará a cabo si y solo si el kernel es real y simétrico. Tal y como se puntualiza en [20], realmente no se utiliza un solo filtro para realizar las convoluciones, sino que se utiliza un conjunto de estos. Aunque parezca redundante, es necesario mencionar que al aplicarse un conjunto de filtros como resultado se tiene un conjunto de matrices de salida igual al número de filtros utilizados. A este conjunto de filtros de salida se les conoce como *feature mapping*.

Una vez ya se le ha aplicado la convolución a la imagen de entrada se aplica la función de activación, la cual según el valor que a esta se le proporcione, se obtendrá un valor en un

rango determinado como (0, 1) o (-1, 1), tal y como se explicó cuando en este mismo documento se habló del funcionamiento básico del perceptrón.

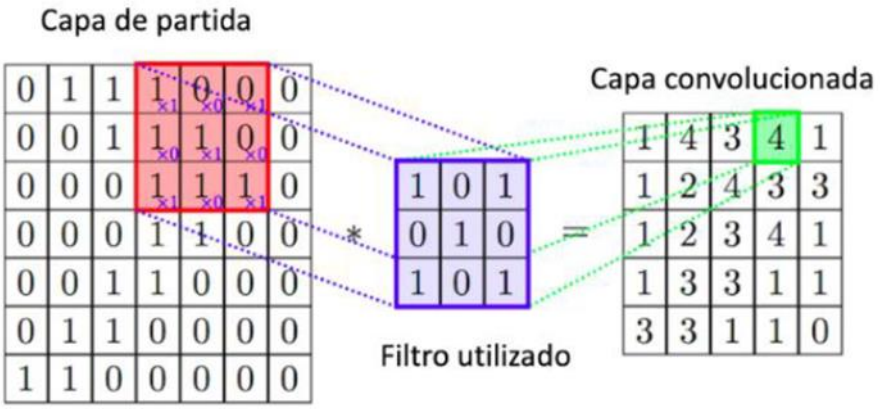


Fig. 1.18: Representación del trabajo realizado por el kernel para realizar la convolución [20].

2 MOTIVACIÓN Y OBJETIVOS

La **motivación** de este proyecto es desarrollar un sistema que permita la detección de fracturas de forma temprana, mediante la utilización de un ecógrafo y una IA entrenada a modo de ayuda para el profesional que esté empleando esta técnica de imagen médica.

Esta motivación proviene de varias necesidades latentes en la atención primaria en la mayoría de los sistemas sanitarios. Entre estas necesidades se enmarca la carencia de un sistema que permita realizar una primera valoración del estado de un paciente que haya sufrido un traumatismo óseo, evitando si es posible desplazamientos innecesarios de larga distancia y pudiendo tomar medidas óptimas en el menor tiempo posible.

Una de las grandes causas que justifican la necesidad de un sistema como el que se plantea en este proyecto, es el hecho de que las radiografías y las tomografías son métodos que utilizan radiación ionizante para la adquisición de imágenes; haciendo que desde el punto de vista de muchos expertos en el ámbito médico pueda llegar a considerarse como un método invasivo de atención al paciente. Algunos de los pacientes más susceptibles a este tipo de radiación ionizante son las embarazadas y los niños.

Por el contrario, la imagen de ultrasonidos cuenta con múltiples ventajas para la realización de esta tarea. La primera de ellas, y la más importante, es que a diferencia de las radiografías convencionales y la tomografía, los sistemas ecográficos no utilizan radiación ionizante, la cual en dosis grandes puede suponer un problema de salud al paciente. También es destacable por parte de los sistemas ecográficos el hecho de la portabilidad, es decir, estos sistemas actualmente pueden ser transportados con suma facilidad hasta el punto donde se encuentre el paciente. Por el contrario, los sistemas de rayos X, tomografía o resonancia magnética, no son portátiles; de hecho llevan un mantenimiento y un cuidado exhaustivo del entorno en donde se sitúan.

Debido a los motivos ya expuestos, una de las vistas a futuro de este proyecto, es conseguir que el sistema desarrollado sea usado como la primera herramienta de valoración ante posibles fracturas.

Desde el punto de vista comercial, este sistema se puede llegar a utilizar incluso en eventos deportivos profesionales de toda índole, propiciando de un diagnóstico inmediato y eficaz

al usuario. Esto es posible ya que se aprovecha una de las virtudes más grandes que tienen los sistemas de ultrasonidos, la cual es su portabilidad, tal y como se ha comentado anteriormente.

Para satisfacer esta motivación, el proyecto debe satisfacer el objetivo general de clasificar imágenes ecográficas según presenten, o no lo hagan, fracturas (esto es, realizar una detección de fracturas en la imagen) y, para el caso de que haya fracturas hacer una localización aproximada de las mismas.

Para ello abordaremos los siguientes objetivos específicos:

- **O1.** Análisis y elección de un *framework* para la implementación de la IA, utilizando aprendizaje profundo con redes neuronales.
- **O2.** Conseguir realizar una reconstrucción 3D de la zona afectada a partir de la secuencia de imágenes ecográficas 2D.
- **O3.** Selección y entrenamiento de la red neuronal destinada a la detección y localización de fracturas.
- **O4.** Integración de la IA en un software con interfaz gráfica que permita la conexión en tiempo real con el ecógrafo y la clasificación y visualización de imágenes en tiempo real para que pueda ser utilizada por el usuario.
- **O5.** Evaluar el sistema desarrollado en modelo de simulación, y evaluar los resultados obtenidos en el mismo.

En este documento, se pretende explicar de forma detallada cómo se ha abordado cada uno de los objetivos expuestos, a fin de mostrar total transparencia en la ejecución del mismo.

3 MATERIALES

3.1 SOFTWARE UTILIZADO

Antes de entrar en materia del desarrollo de este proyecto, es necesario identificar los materiales utilizados para el desarrollo del mismo, es por ello que este apartado tiene como misión evaluar el software utilizado, dando detalles del porqué de su utilización.

3.1.1 PyTorch

Anteriormente se evaluaron los distintos *frameworks* destinados al entrenamiento y desarrollo de redes neuronales. Y de los expuestos, se ha decidido escoger para llevar a cabo la tarea que se pretende PyTorch.

Entre muchas de las características favorables que tiene PyTorch, se ha escogido ya que está orientado esencialmente a ser usado con lenguaje de programación Python. Este es un lenguaje de programación de alto nivel orientado a objetos, cuya popularidad recae principalmente en la gran cantidad de bibliotecas y *frameworks*; además de ser un lenguaje multiplataforma, es decir, es compatible con los principales sistemas operativos, como Windows, macOS y Linux. Otro de los puntos a favor que tiene PyTorch es la excelente compatibilidad que tiene con las GPUs de Nvidia, siendo de vital relevancia en este proyecto, debido a que en el sistema donde se ha llevado a cabo el entrenamiento utiliza una tarjeta gráfica de Nvidia.

Este soporte es esencial, ya que los cálculos realizados para entrenar una red neuronal no son ni más ni menos que operaciones con matrices. Precisamente la GPU está especialmente diseñada para realizar este tipo de operaciones matriciales de forma eficaz, comparado con la CPU. Tal y como explica la propia Nvidia en [21], la GPU está orientada a ejecutar aquella parte de las aplicaciones donde la computación es más intensiva, mientras que lo demás se ejecuta en la CPU.

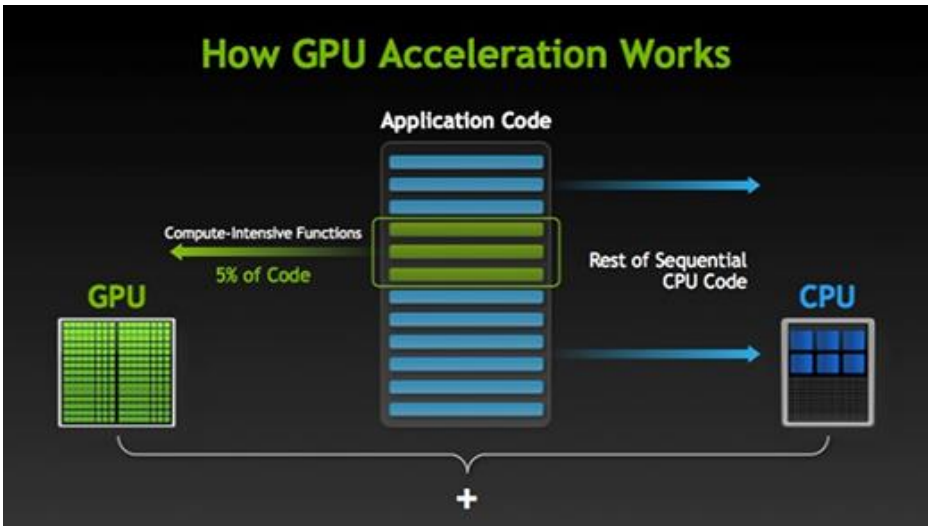


Fig. 3.1: Distribución del código entre la GPU y la CPU [22].

Aparte de lo ya mencionado, otro de los puntos a favor de PyTorch es la existencia de otras librerías complementarias que facilitan el entrenamiento de las redes neuronales, como puede ser el caso de PyTorch-Lightning el cual es clave en el desarrollo de este proyecto, y se explica con profundidad en el siguiente punto.

3.1.2 PyTorch-Lightning

PyTorch-Lightning es un *framework* de *deep learning*, por encima de PyTorch, que tiene como objetivo simplificar y acelerar el proceso de entrenamiento y desarrollo de modelos de aprendizaje automático utilizando. Dicho con otras palabras, esta biblioteca proporciona una capa de abstracción sobre Pytorch, consiguiendo por tanto que el código resultante sea más limpio, modular y legible. De esta forma se consigue eludir el tener que desarrollar bucles para llevar a cabo el entrenamiento de la red neuronal, por tanto, desde el punto de vista del programador, no hay que preocuparse de la lógica del entrenamiento, sino más bien de los detalles de implementación relacionados con el entrenamiento, la administración de las GPUs, la monitorización de métricas (las cuales se llevan a cabo mediante '*callbacks*'), el registro de resultados y otras tareas comunes. Este *framework* ofrece una facilidad increíble a la hora de entrenar modelos sobre GPU, o múltiples GPUs de forma distribuida, e incluso TPUs. Actualmente Pytorch Lightning se encuentra en su versión 2.0, que es la versión que se ha utilizado para realizar el entrenamiento de la red neuronal orientada a la detección de fracturas.

3.1.3 Jupyter Notebooks

Jupyter Notebook es una aplicación web de código abierto cuya finalidad reside en crear y compartir documentos interactivos que contienen código, texto explicativo, gráficos y animaciones. Su uso es muy intuitivo, y está orientado a la ejecución de código en diferentes lenguajes de programación, pero por lo general, es popularmente conocido por ser la opción predilecta para aquellos que escriben su código en Python. Pero, como se ha comentado, también admite lenguajes como R, Julia, Scala, entre otros, consiguiendo que sea una herramienta versátil para diferentes áreas y aplicaciones. A parte de las funcionalidades ya comentadas, Jupyter permite hacer procesado de texto mediante Markdown. Este último es especialmente utilizado a la hora de combinar explicaciones, ecuaciones matemáticas, imágenes y visualizaciones con el código necesario para realizar cálculos, manipular datos y generar resultados.

La forma de trabajar con Jupyter Notebooks es mediante las celdas, las cuales compilarán el código que se haya escrito en ellas. Para que este código sea compilado adecuadamente, es necesario especificar el *kernel* que se va a utilizar, ya que de este depende que el código sea ejecutado correctamente. También es necesario ejecutar los notebooks en un entorno que tenga las librerías que se pretenden utilizar instaladas en el mismo, para dedicar posteriormente una celda a la importación de las mismas. De no haber realizado este último paso, el código no sería capaz de ser compilado.

Se ha seleccionado Jupyter Notebooks para el entrenamiento de las redes neuronales debido a su gran versatilidad a la hora de poder trasladar los notebooks de un equipo a otro en caso de que fuera necesario. Otro de los puntos por los que se ha elegido Jupyter Notebooks, es debido a su gran desempeño a la hora de ejecutar Python.

3.1.4 3D Slicer

3D Slicer [23], [24] es una plataforma de software gratuita de código abierto para la computación de imágenes médicas, además es usado como medio para desarrollar software útil para investigaciones en el ámbito clínico y biomédico. Esta dispone de una de una comunidad *open-source* (código abierto), que mantiene la plataforma actualizada con los algoritmos más potentes del mercado para procesamiento de imágenes médicas y ayuda a desarrollar módulos (extensiones) que permiten añadir funcionalidades al programa. Además, 3D Slicer proporciona un entorno de trabajo especialmente enfocado

en la investigación y aplicación clínica en imágenes médicas. También, permite la visualización, manipulación, segmentación y registro de datos de imágenes en 3D obtenidos de diversas modalidades, como tomografía computarizada (CT), resonancia magnética (MRI), ecografía y otras.

Implementar módulos en 3D Slicer es relativamente sencillo, pues el propio programa ofrece una plantilla escrita en lenguaje de programación Python (como bien se mencionó en anteriores apartados) diseñada para facilitar la labor a la persona que programa dicho módulo. Programáticamente la plantilla se divide en bloques de código siguiendo la jerarquía mostrada a continuación.

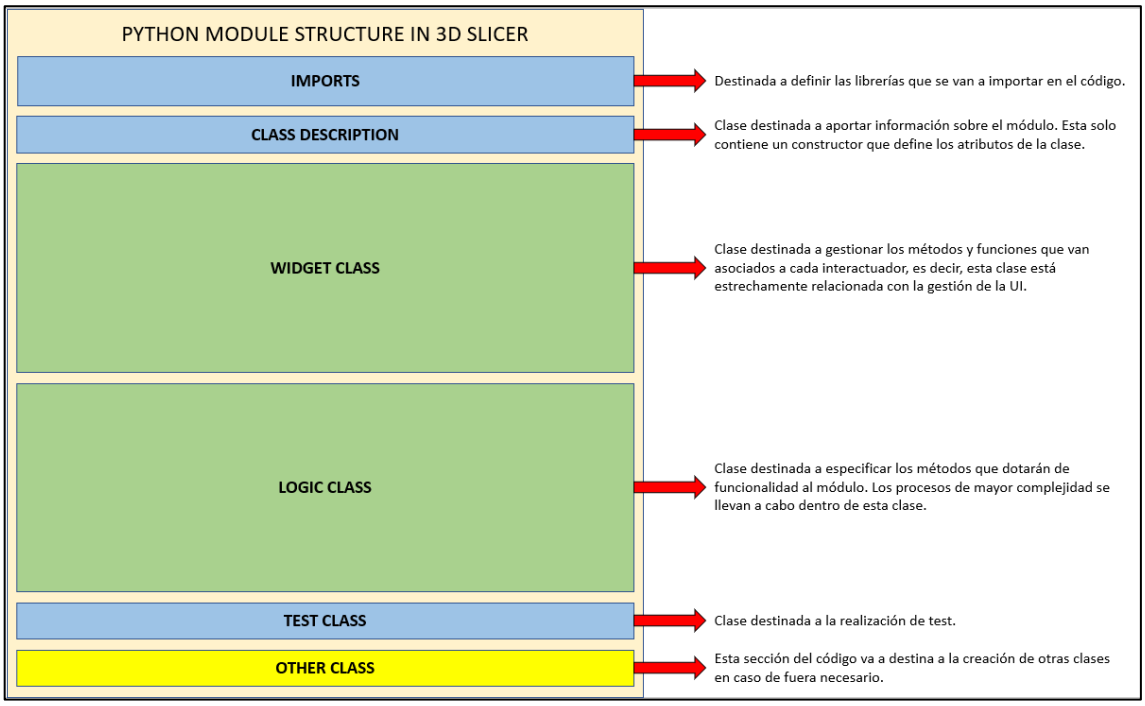


Fig. 3.2: Esquema conceptual de la distribución del código en un módulo de 3D Slicer.

Un dato curioso, es que 3D Slicer también te ayuda en la creación de IUs mediante una aplicación dependiente de 3D Slicer llamada Qt Designer. Esta aplicación está orientada al desarrollo de IUs de una forma relativamente intuitiva, ya que tan solo te tienes que preocupar de colocar los interactuadores (botones) de forma visual, sin necesidad de programar nada. Una vez hecho esto, Qt Designer genera un archivo ".ui" que tiene definidos los botones de la interfaz creada. Posteriormente en el código se importará el

archivo ".ui" creado, para conectar los botones con las funcionalidades correspondientes en el código.

3.1.5 PLUS Toolkit

PLUS Toolkit [25] es una aplicación de código abierto externa de 3D Slicer, la cual adquiere la información de los dispositivos hardware conectado a un equipo (como ecógrafos o sistemas de posicionamiento) y envía esta información mediante un protocolo TCP/IP; es por este motivo que cuando se quiere combinar el uso de 3D Slicer con PLUS, es necesario conectarse a PLUS por un puerto desde 3D Slicer.

PLUS Toolkit fue diseñada para servir de interface entre el hardware y el software orientado a hacer un tratamiento de los datos transferidos por esta. O tal y como se define en [26], esta es un conjunto de herramientas para la adquisición, preprocesamiento y calibración de datos para intervenciones guiadas por imágenes navegadas. Aunque PLUS Toolkit puede ser utilizada también con otros propósitos que se verán más adelante en este mismo documento. Para poder combinar las funcionalidades que ofrece PLUS Toolkit con 3D Slicer, es necesario utilizar OpenIGTLink [27], la cual es un protocolo de comunicación que permite el envío de información como transformadas, imágenes, y otros datos de interés para el campo de la cirugía guiada por imagen.

3.1.6 IGT – Volume Reconstruction

Dentro del paquete de módulos que ofrece la extensión SlicerIGT [28] encontramos *Volume Reconstruction*. Este módulo está diseñado para ser utilizado con imagen de ultrasonidos, ofreciendo la posibilidad de generar una reconstrucción del hueso o, en el caso de este proyecto, de la zona afectada. Este permite seleccionar si se quiere generar el volumen mediante una secuencia de imagen de ultrasonidos grabada o, por el contrario, mediante imagen ultrasonido en tiempo real.

Para que el módulo pueda realizar esta tarea es necesario especificarle el volumen que se va a tener como entrada al sistema, en base a este generará la reconstrucción 3D del volumen de entrada en un volumen nuevo o en uno ya existente. Debido a las propiedades del tejido óseo, la identificación de éste durante la reconstrucción volumétrica es bastante sencilla y no requiere de algoritmos de IA para su identificación.

3.2 HARDWARE UTILIZADO

Este apartado tiene como misión evaluar el hardware utilizado, dando detalles del porqué de su utilización.

3.2.1 Fantoma para simulación de hueso fracturado

Un fantoma es un modelo destinado a imitar las propiedades del tejido humano. Este es el pilar fundamental de este proyecto, ya que en base al fantoma se realizarán los ensayos pertinentes. El fantoma desarrollado para la ejecución de este proyecto está compuesto por dos huesos fracturados hechos con arcilla, y por una mezcla en estado sólido de compuestos químicos a modo de imitación de la piel humana (tejido blando).

La mezcla realizada para la elaboración de este tejido blando simulado es: 250 gramos de vinilo líquido, 30 gramos de celulosa y 65 gramos de ablandador. La forma en la que se ha realizado está mezcla se explica con más detalle en el apartado 4 de este documento.

La fabricación del fantoma requiere realizar la mezcla a alta temperatura. Por ello los huesos se han fabricado con arcilla, y no se han fabricado huesos mediante impresoras 3D para esta labor, ya que los materiales utilizados para la impresión son muy sensibles al calor y tienden a deformarse. Por el contrario, la arcilla resiste muy bien la alta temperatura, y además ofrece propiedades acústicas adecuadas para su visualización en las imágenes ecográficas.

3.2.2 Ecógrafo Telemed

El ecógrafo utilizado para llevar a cabo el desarrollo de este proyecto es la sonda para ecografía lineal L12-5L40S-3 del fabricante Telemed Medical Systems. Esta sonda, tal y como se ha indicado en su propio nombre, es de tipo lineal y abarca un rango en frecuencia que va desde los 5 – 12 MHz, haciéndola ideal para aplicaciones de toma de imagen ecográfica vascular, para partes pequeñas, pediátrica y musculoesquelética [29]. Para su correcto uso experimental, ha sido necesario el uso de un adaptador del propio Telemed, llamado MicrUs EXT-1H. Este es capaz de gestionar la señal que llega directamente desde la sonda, enviándola vía USB al dispositivo al que se quiera conectar. Este además, está preparado para soportar imagen ecográfica en modo B, M, B + M, 4B, PW Doppler, PW Doppler + B (duplex).

Fig. 3.3: Imagen del L12-5L40S-3 de Telemed.

Fig. 3.4: Imagen del MicrUs EXT-1H de Telemed.

Type	Frequency (MHz)	Scanning Method	Field of View Degree/mm	Applications
Convex				
C5-2R60S-3	2.0-5.0	Convex R60	65	Abdominal, Obstetrics, Pediatrics
MC4-2R20S-3	2.0-4.0	Convex R20	104	Abdominal, Cardiac, Veterinary
MC10-5R10S-3	5.0-10.0	Convex R10	147	Pediatrics, Small Parts, Vascular, Veterinary
Linear				
L12-5L40S-3	5.0-12.0	Linear 40 mm	39	Pediatrics, Small Parts, Vascular, Anesthesia, Veterinary
L15-6L25S-3	6.0-15.0	Linear 25 mm	24	Pediatrics, Small Parts, Vascular, Anesthesia, Veterinary
Endocavity				
MCV9-5R10S-3	5.0-9.0	Convex R10	147	Transvaginal
Veterinary				
LV8-4L65S-3	4.0-8.0	Linear 65 mm	64	Veterinary

Fig. 3.5: Especificaciones técnicas de la sonda L12-5L40S-3 en comparación con otras sondas del mismo fabricante.

Una vez llegaba la señal al sistema utilizado para visualizar la imagen, es necesario ajustar los parámetros haciendo uso de la aplicación Echo Wave II en su versión 4.1.1, proporcionada por el fabricante, el cual permitía utilizar parámetros tales como la frecuencia de trabajo, la profundidad, el punto focal, etc.

3.2.3 Sistema de posicionamiento electromagnético: 3D Guidance trakStar

El sistema 3D Guidance TrakStar [30] es un sistema de seguimiento de posición y orientación en 3D empleado en multitud de aplicaciones en el ámbito médico. Este, mediante la utilización de un campo electromagnético generado por el mismo sistema, es capaz de detectar con precisión la posición y la orientación de objetos o dispositivos en el espacio tridimensional. Este utiliza sensores 6DOF[1] para su detección en el espacio, lo que genera una detección sumamente óptima y precisa de los sensores. Estos sensores están equipados con bobinas que captan las señales del campo electromagnético emitido por la estación base [31]. Estas señales se utilizan para determinar la posición y la orientación de los sensores en relación con la estación base.

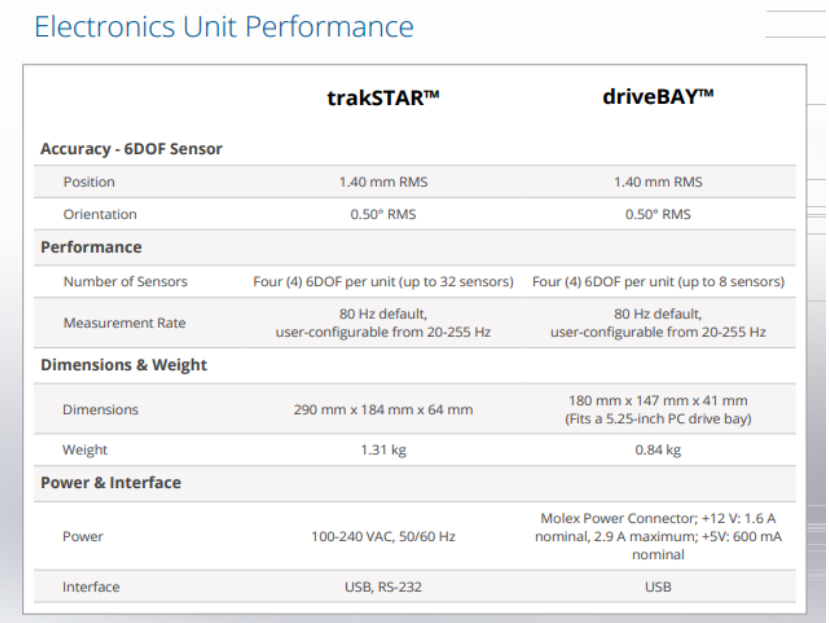
[1] 6DOF: Las siglas DOF se refieren al número de parámetros independientes que definen la posición y orientación de un objeto en el espacio tridimensional. En el caso de los sensores 6DOF, se están rastreando seis parámetros: tres para la posición (eje X, eje Y, eje Z) y tres para la orientación (giro alrededor del eje X, giro alrededor del eje Y, giro alrededor del eje Z).

Una de las características más destacables del sistema TrakSTAR es su baja latencia, lo que significa que proporciona mediciones en tiempo real. Esto lo hace adecuado para aplicaciones que requieren seguimiento preciso, como cirugía asistida por computadora, simulaciones médicas, análisis del movimiento humano, investigación biomédica, entre otros.

Fig. 3.6: Imagen que muestra el sistema 3D Guidance en su versión driveBay y trakSTAR.

En la Fig. 3.6, se muestra la imagen relativa a los dos sistemas que ofrece este fabricante para el posicionamiento de sensores 6DOF. El primero de ellos, el más pequeño, está pensado para ser utilizado en el dentro del compartimento de unidad (Drive Bay) de una computadora. En este caso, cuando se menciona el Drive Bay, se hace referencia a un espacio o compartimento en un chasis de computadora que está diseñado para alojar unidades de almacenamiento, como discos duros o unidades de estado sólido. Estos compartimentos suelen ser espacios estándar que cumplen con ciertas dimensiones y conexiones para permitir la instalación de diferentes dispositivos de almacenamiento.

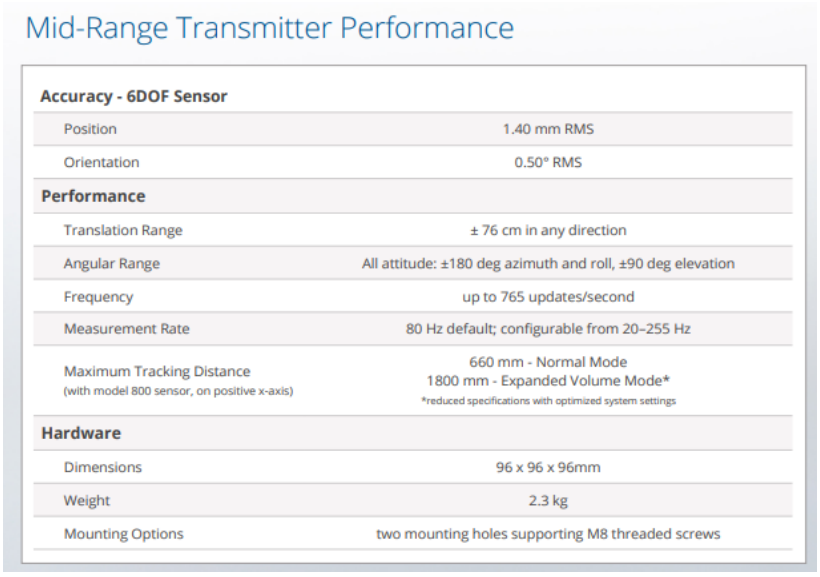
Por otro lado, el trakSTAR (que es el que se ha usado en este caso) es un dispositivo portátil preparado para ser conectado directamente a una toma de corriente. A continuación, se muestran las especificaciones técnicas del trakSTAR donde se especifican sus características.



Electronics Unit Performance

	trakSTAR™	driveBAY™
Accuracy - 6DOF Sensor		
Position	1.40 mm RMS	1.40 mm RMS
Orientation	0.50° RMS	0.50° RMS
Performance		
Number of Sensors	Four (4) 6DOF per unit (up to 32 sensors)	Four (4) 6DOF per unit (up to 8 sensors)
Measurement Rate	80 Hz default, user-configurable from 20-255 Hz	80 Hz default, user-configurable from 20-255 Hz
Dimensions & Weight		
Dimensions	290 mm x 184 mm x 64 mm	180 mm x 147 mm x 41 mm (Fits a 5.25-inch PC drive bay)
Weight	1.31 kg	0.84 kg
Power & Interface		
Power	100-240 VAC, 50/60 Hz	Molex Power Connector; +12 V: 1.6 A nominal, 2.9 A maximum; +5V: 600 mA nominal
Interface	USB, RS-232	USB

Fig. 3.7: Especificaciones técnicas del TrakSTAR.



Mid-Range Transmitter Performance

Accuracy - 6DOF Sensor	
Position	1.40 mm RMS
Orientation	0.50° RMS
Performance	
Translation Range	± 76 cm in any direction
Angular Range	All attitude: ±180 deg azimuth and roll, ±90 deg elevation
Frequency	up to 765 updates/second
Measurement Rate	80 Hz default; configurable from 20–255 Hz
Maximum Tracking Distance (with model 800 sensor, on positive x-axis)	660 mm - Normal Mode 1800 mm - Expanded Volume Mode* *reduced specifications with optimized system settings
Hardware	
Dimensions	96 x 96 x 96mm
Weight	2.3 kg
Mounting Options	two mounting holes supporting M8 threaded screws

Fig. 3.8: Especificaciones técnicas del generador de campo.

Tal y como se ve reflejado en la Fig. 3.8, se ha incluido las especificaciones técnicas del transmisor electromagnético encargado de generar el campo en el cual se efectuará el posicionamiento de cada uno de los sensores que vayan conectados al sistema central.

4 MÉTODOS

Debido a la inexistencia de una base de datos pública que contenga imágenes de huesos fracturados tomados con imagen ecográfica, se ha optado por conformar una base de datos propia, utilizando para ello dos técnicas diferentes. La primera de ellas consiste en utilizar un software que permita simular imágenes de fracturas obtenidas mediante una sonda ecográfica, y la segunda trata de el desarrollo de un fantoma de hueso fracturado que permita la obtención para posteriormente obtener imágenes ecográficas del mismo. Tras ejecutar estos dos procesos se tendrían dos bases de datos de naturaleza distinta, ya que una es puramente virtual y la otra es obtenida de un fantoma con un ecógrafo real. Posteriormente, en este mismo proyecto, se detallará el procedimiento a seguir con estas dos bases de datos.

Tras la breve explicación del planteamiento, se exponen los apartados que conformarán este capítulo de la memoria. Primeramente, en los puntos 4.1 y 4.2 se detallará cómo se han generado escenarios, virtuales y reales, para la generación de imágenes de fracturas. En el 4.3 se detallarán como se han generado las bases de datos de imágenes a partir de los escenarios ya vistos. Posteriormente en el punto 4.4, se detallará el proceso de desarrollo de los algoritmos de inteligencia artificial; siendo objeto de este entender, no solo la cadena de procesos seguida para el desarrollo de estos algoritmos, sino también algunas de las técnicas empleadas para cumplir los objetivos propuestos con éxito. Finalmente, en los puntos 4.5 y 4.6, se explicará el desarrollo de una aplicación con interfaz gráfica para llevar a cabo de la evaluación del sistema en un escenario realista.

4.1 GENERACIÓN DE ESCENA VIRTUAL DE FRACTURAS

El objetivo de este apartado es explicar de forma detallada cómo se ha generado la escena virtual para la obtención de imagen ecográfica de fracturas. Como ya se ha dicho, el motivo de la utilización de este sistema en el proyecto, se debe a la falta de bases de datos públicas que contengan imágenes de fracturas tomadas con sondas ecográficas que podamos utilizar para entrenar redes neuronales.

Es necesario definir algunas de las características del sistema que se ha empleado para la obtención de las imágenes ecográficas simuladas.

Para ello se ha utilizado en 3D Slicer un módulo que permite llevar a cabo, junto con PLUS Toolkit, la adquisición imágenes ecográficas simuladas [32] (nombre del módulo: *Fracture Ultrasound Simulator*). Esta es una adaptación de otro módulo previamente desarrollado por el cotutor de este TFG, David García Mato. Originalmente el módulo estaba destinado, en un principio, a la simulación de imagen ecográfica en mujeres embarazadas, es por ello que se ha aprovechado esta funcionalidad del mismo, para implementar sobre él una escena que permita adquirir imágenes ecográficas simuladas con distintos tipos de huesos fracturados.

Antes de poner en marcha este módulo para la adquisición de imágenes, se debe realizar previamente la creación de los modelos (todos ellos en formato STL) destinados a ser utilizados en 3D Slicer. A continuación, se muestran imágenes de los modelos empleados en para la adquisición de imágenes (Modelos adquiridos en base a la modificación de un modelo de esqueleto humano utilizando Autodesk Mesh Mixer).

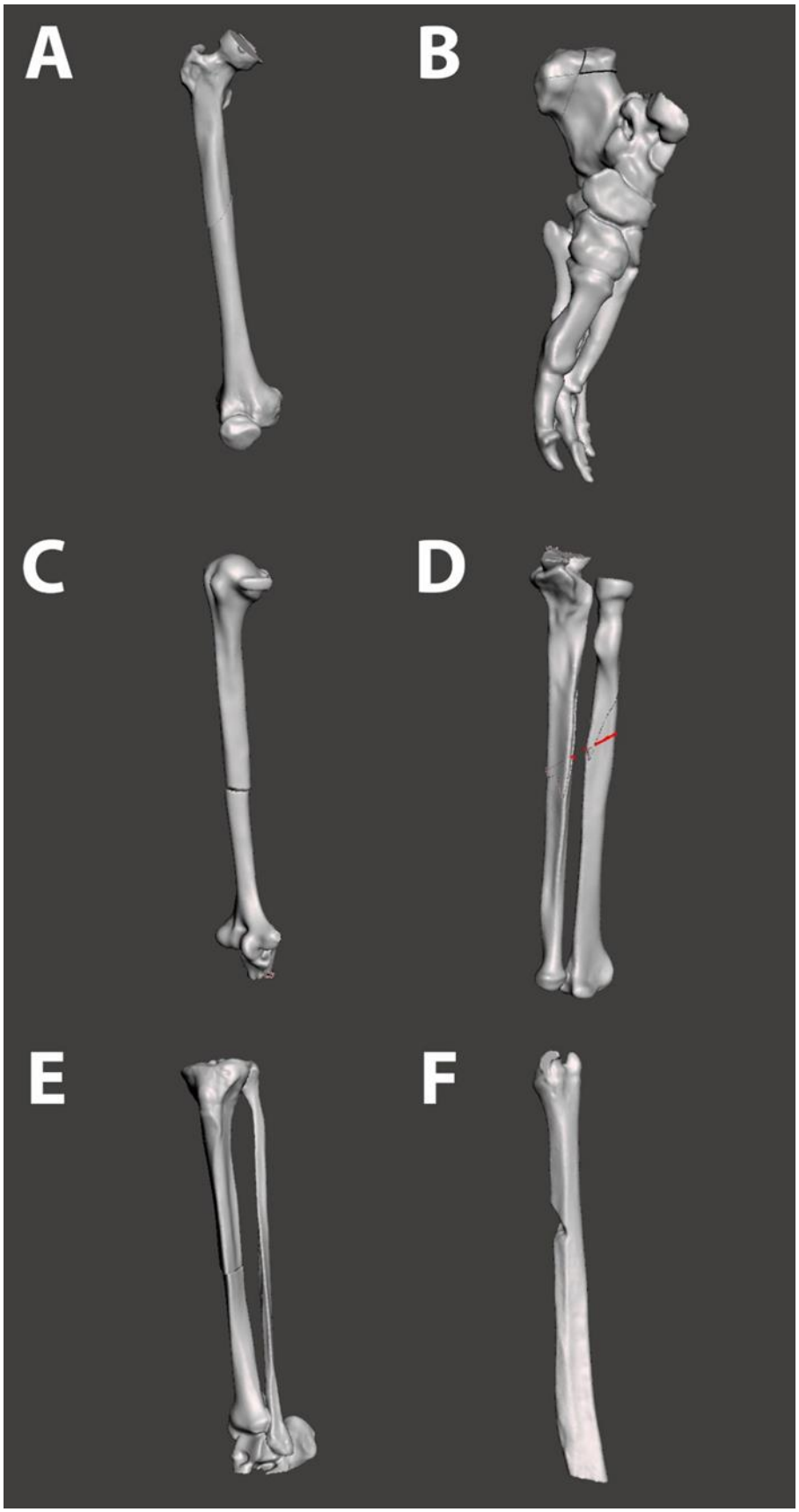


Fig. 4.1: Previsualización de los modelos STL. (A) Fractura diafisaria simple de fémur, (B) Fractura a tercer fragmento en talón, (C) Fractura diafisaria simple de húmero, (D) Fractura a tercer fragmento

de cúbito y radio, (E) Fractura diafisaria simple de tibia, (F) Fractura a tercer fragmento de cúbito con ausencia de fragmento.

Tal y como se pueden apreciar en la imagen aportada, el tipo de fracturas que se está tratando en su mayoría, son fracturas simples; aunque también podemos encontrar fracturas a tercer fragmento, como puede ser el caso de B, D y F. Una vez desarrollados los modelos en Autodesk Meshmixer, se procedió a acondicionar un entorno de trabajo que ofrece el módulo de 3D Slicer para llevar a cabo una adquisición de imágenes satisfactoria.

Primeramente, se definieron las posiciones de los distintos modelos en el espacio, aplicándoles a cada uno de ellos las transformaciones correspondientes. Estas transformaciones correspondientes a cada uno de los objetos de la escena son transferidos a PLUS, el cual, tiene definidas las propiedades acústicas de cada uno de los elementos que conforman la escena virtual. Estas propiedades han sido definidas en un archivo XML necesario para que PLUS lleve a cabo su funcionalidad correctamente.

En segundo lugar, es necesario mencionar que el módulo original no disponía de botones que permitieran guardar las imágenes en un directorio deseado, por ello se implementaron dos botones que, dependiendo del tipo de imagen que estemos tomando (imagen de hueso fracturado o imagen de hueso sano) se utilizará el botón correspondiente, véase en la Fig. 4.2.

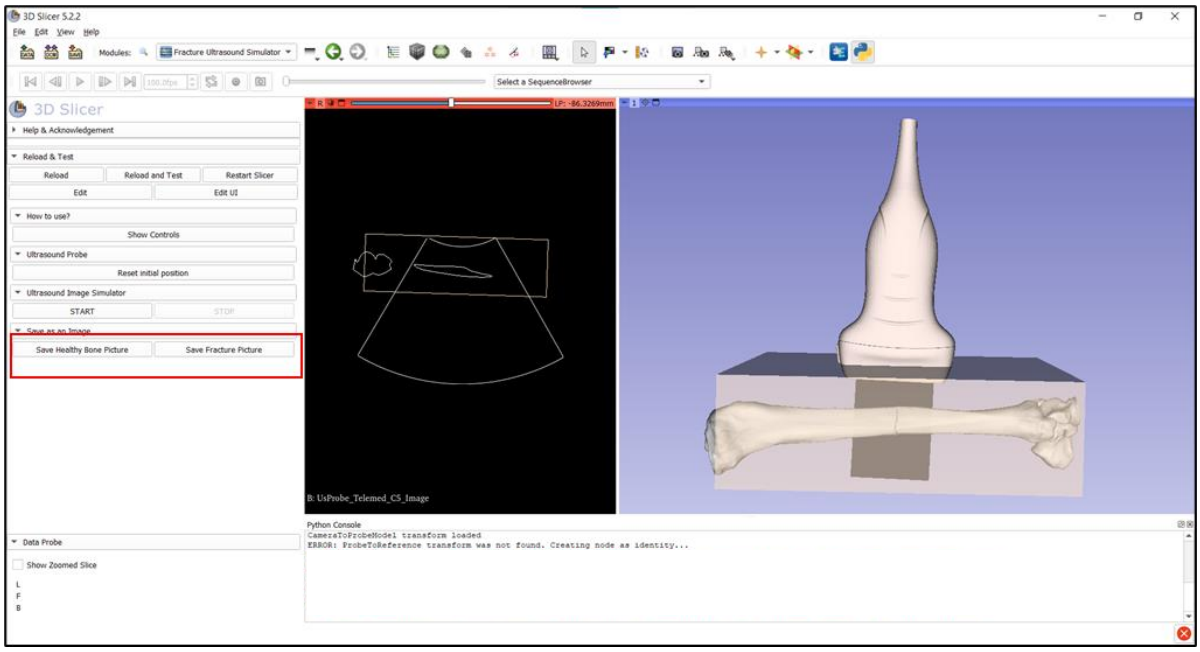


Fig. 4.2: Pulsador "Save Healthy Bone Picture": guarda la imagen en una carpeta llamada "HealthyBonesImages" donde están aquellas imágenes tomadas a huesos sanos. Pulsador "Save Fracture Picture": guarda la imagen en una carpeta llamada "FractureUsImages" donde están aquellas imágenes tomadas a huesos fracturados.

Una vez se ha explicado cómo se ha adaptado el entorno de forma adecuada, se procede a comentar de forma detallada cómo funciona exactamente el módulo del que se ha venido hablando a lo largo de este documento.

El módulo funciona con PLUS de forma complementaria para poder realizar la simulación de imagen de ultrasonido. El funcionamiento de este sistema se rige bajo el modelo cliente-servidor, tal y como se ilustra en Fig. 4.3. El propio PLUS recibe la posición de la sonda en el módulo de 3D Slicer, es por ello que, en función de esta, PLUS devolverá la imagen que debería recibir la sonda en dicha posición.

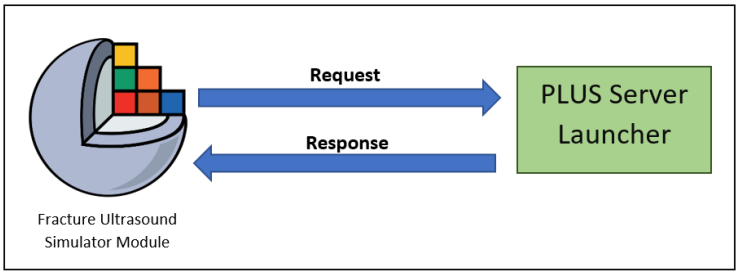


Fig. 4.3: Modelo cliente-servidor entre 3D Slicer (cliente) y PLUS (servidor).

Este trasiego de información representado en la Fig. 4.3, se realiza de forma local en el equipo desde el cual se está trabajando. Para llevar a cabo esto, se ha de configurar tanto el módulo como el servidor respectivamente. En el caso del módulo, esta configuración se realiza automáticamente, esta se refleja en un módulo complementario llamado OpenIGTLinkIF, Fig. 4.4.

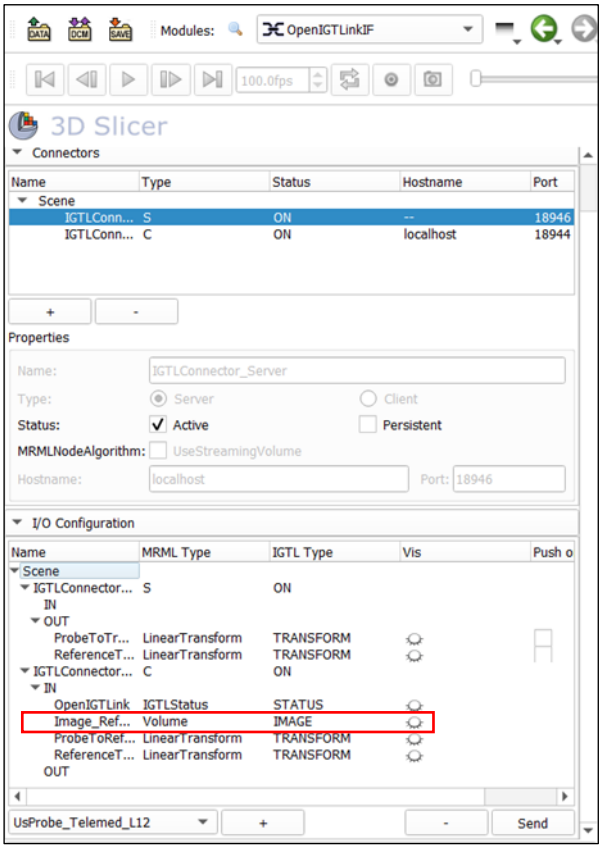


Fig. 4.4: Configuración realizada por el módulo Fracture Ultrasound Simulator sobre OpenIGTLinkIF para facilitar la comunicación con el servidor.

Tal y como se ve en la imagen, en el apartado *I/O Configuration*, se puede ver como la propia aplicación cliente envía las transformaciones de la sonda (*OUT*) y recibe, entre otras cosas, el volumen correspondiente a la imagen simulada (*IN*). Por otro lado, desde el punto de vista del servidor, es necesario realizar una configuración desde un archivo XML. Este archivo es intrínseco al propio PLUS, es decir, este es necesario siempre que se quiera utilizar PLUS como servidor para un módulo en 3D Slicer. A continuación, se muestra una parte de este archivo de configuración, Fig. 4.5.

(A) (B)

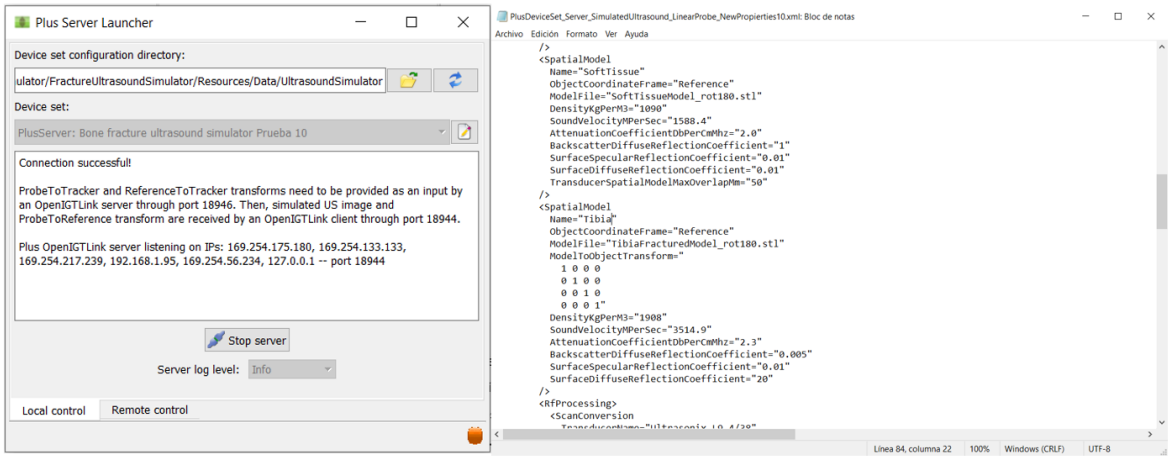


Fig. 4.5: (A) Interfaz inicial de PLUS Server Launcher. (B) Archivo de configuración de PLUS.

Es de destacar que en el archivo de configuración se detallan, no solo los sistemas que están involucrados en el proceso, sino también las características físicas de los modelos involucrados en la escena (tal y como se muestra en la imagen (B), Fig. 4.5), las cuales, la gran mayoría fueron sacadas de un archivo Excel que disponía de multitud de parámetros físicos relativos a los distintos tipos de tejidos que conforman la estructura anatómica humana. Esto es lo que consigue generar la imagen simulada que se busca en esta parte del proyecto, esta es enviada al puerto 18944 del cliente, el cual ha sido previamente configurado tal y como se vio anteriormente.

Es necesario mencionar que 3D Slicer y PLUS utilizan distintos sistemas de coordenadas, por ello, para poder efectuar la simulación en el entorno virtual, es necesario realizar una transformación a los modelos. Una vez visto todos los puntos fuertes en el desarrollo de esta parte del proyecto, es necesario puntualizar que se han hecho modificaciones en el archivo de configuración XML para conseguir una imagen fidedigna a lo que se obtendría utilizando el modelo de ecógrafo Telemed que se va a emplear para el desarrollo del proyecto. Es por ello que junto con unos de los supervisores de este proyecto, se ha llevado una comparativa de los resultados obtenidos en la simulación, con el objetivo de encontrar aquella imagen que mejor se adapte a lo que se quiere buscar. Para ello, se desarrolló una presentación en la que se mostraban todas las imágenes tal y como se muestra en Fig. 4.6.

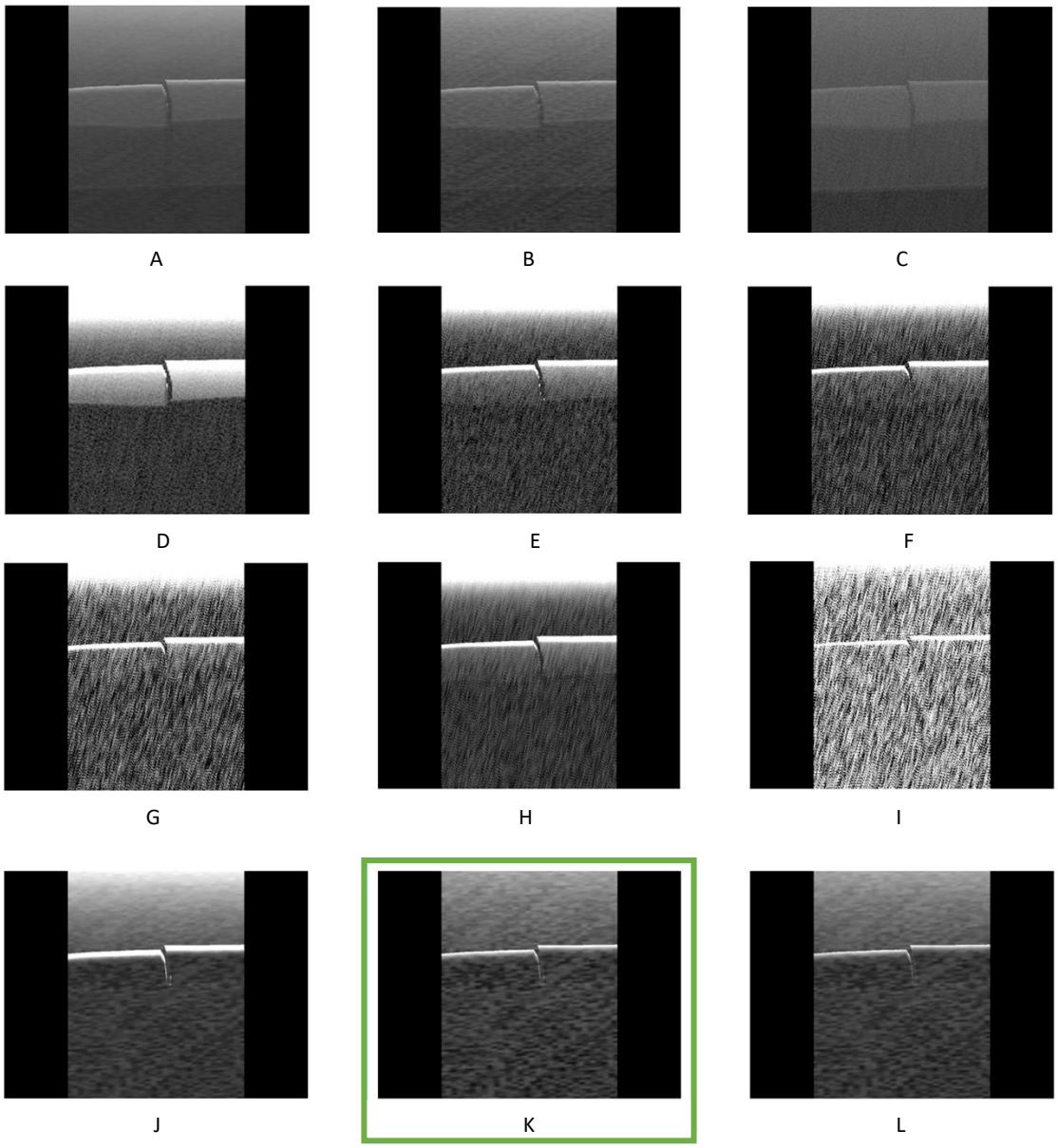


Fig. 4.6: Comparativa de todas las imágenes obtenidas en el proceso de simulación. Se muestra enmarcada en verde la imagen finalmente seleccionada.

4.2 DISEÑO Y FABRICACIÓN DE FANTOMA

Es misión de este apartado redactar de qué forma se ha elaborado el fantoma de hueso fracturado. Ya se ha dicho que el motivo por el que se ha llevado a cabo este proceso de fabricación, se debe a que, ante la ausencia de una base de datos de imágenes ecográficas de fracturas reales, se pretende obtener una base de datos lo suficientemente realista como para simular la actuación del modelo de red neuronal en un entorno real.

Para la elaboración del hueso fracturado se ha utilizado arcilla blanca, y con esta se han hecho dos huesos con fractura simple, véase Fig. 4.7.

Fig. 4.7: Huesos hechos de arcilla representando fractura simple; únicamente divide el hueso en dos mitades.

A diferencia de lo que figura en el anteproyecto, se ha antepuesto la opción de utilizar arcilla, debido a que los materiales que comúnmente son utilizados para realizar impresiones 3D, no son capaces de soportar las altas temperaturas que se requieren para la fabricación del fantoma.

Tal y como se menciona en el apartado 3.2.1, para la fabricación del fantoma se ha requerido de los siguientes materiales: 250 gramos de vinilo líquido, 30 gramos de celulosa microcristalina y 65 gramos de ablandador. Además, se ha requerido de una báscula y algunos recipientes para poder preparar la mezcla correctamente; uno de ellos se acondicionó incluso para que sirviera como molde. En la Fig. 4.8 se muestra imagen de todos los elementos utilizados.

Fig. 4.8: Materiales empleados para la fabricación del fantoma.

Como ya se conocen los elementos que forman parte de este proceso, se requiere mencionar los pasos a seguir para el desarrollo del fantoma. En primer lugar, se calientan los 250 gramos de vinilo líquido (cuyo color es blanquecino) en el microondas, entre 3 minutos o más a la potencia 3 del microondas. Una vez transcurrido ese tiempo, el vinilo pasa de ser color blanco a tener un color transparente; este es un indicio de que el vinilo está listo para ser mezclado con el ablandador y la celulosa.

Fig. 4.9: Antes y después del vinilo tras haber sido introducido en el microondas. Como se aprecia, tras sacarlo del microondas se torna semitransparente.

En el proceso de mezcla de ablandador y celulosa en el vinilo, se debe de verter primeramente el ablandador, y seguidamente la celulosa; esta última se debe ir espolvoreando poco a poco encima de la mezcla a la par que se remueve lentamente. El proceso de mezcla de la celulosa es muy delicado, ya que se debe ir removiendo despacio para no generar burbujas, ni tampoco grumos. Una vez se ha hecho la mezcla, se procede a meter la misma en el microondas a potencia 1 durante un minuto o dos; esto se hace con el afán de mantener la mezcla caliente antes de verterla.

Cuando ya se tiene la mezcla lista para verter en el molde, antes de poner los huesos, es necesario generar una pequeña capa en el fondo del molde, con el fin de que los huesos queden perfectamente confinados dentro del fantoma, véase Fig. 4.10 y Fig. 4.11.

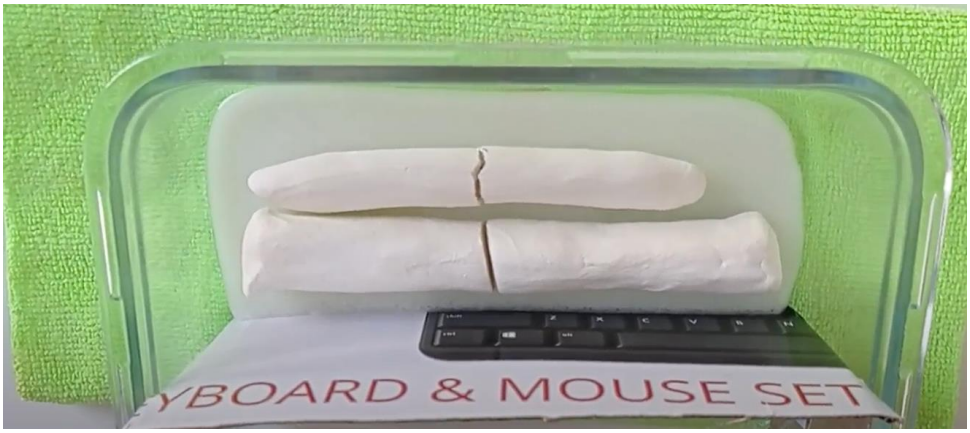


Fig. 4.10: Una vez se ha colocado el fondo, encima del mismo se colocan los huesos fracturados.

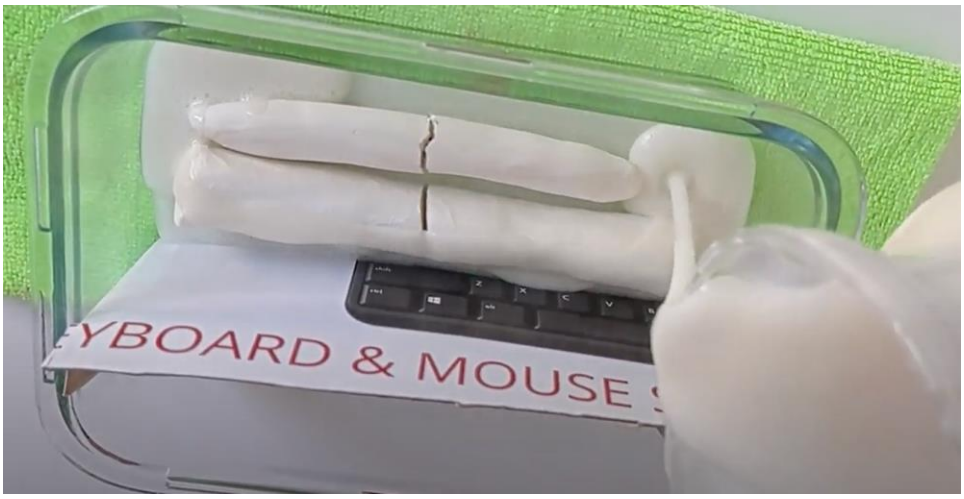


Fig. 4.11: Se vierte la mezcla encima de ambos modelos. El resultado de la mezcla, tal y como se ve, es de un color blanco bastante denso.

Teniendo la primera capa cubriendo de forma homogénea el fondo del molde, ya se puede proceder a colocar ambos huesos de arcilla fracturados. Una vez han sido colocados se vierten encima de estos el resto de la mezcla y se deja fraguar el tiempo que se estime necesario.

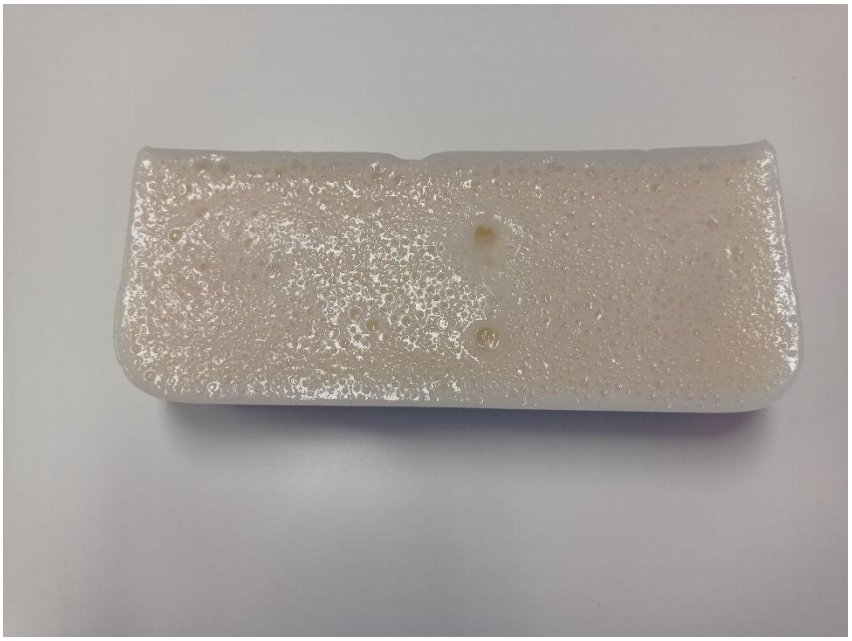


Fig. 4.12: Resultado final del fantoma tras el fraguado de la mezcla.

Como se puede apreciar, en el fraguado han salido una serie de burbujas a la superficie las cuales dificultarían una buena calidad de imagen con la sonda; para solucionar este

problema, se ha optado por extraer una lámina de la parte superior del mismo para eliminar la gran mayoría de las burbujas que se observan. Una de las soluciones hubiera sido hacer otro fantoma, pero no había vinilo líquido suficiente; aparte de que actualmente no hay proveedores que suministren vinilo líquido a Canarias.

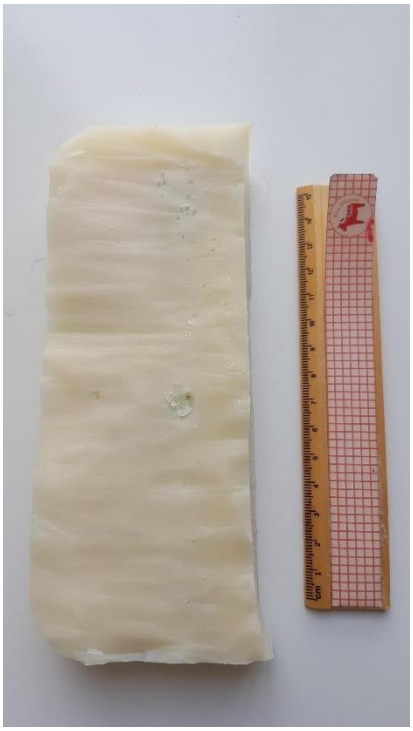


Fig. 4.13: Resultado de la extracción de burbujas de la superficie del fantoma.

Tal y como se puede ver en la última figura, se han extraído las burbujas que se han formado en el proceso del fantoma.

4.3 CREACIÓN DE BASES DE DATOS

4.3.1 Imágenes simuladas

Una vez ha quedado todo perfectamente adecuado, se lleva a cabo la toma de imagen ecográfica simulada utilizando los controles que se detallan en la Fig. 4.14:

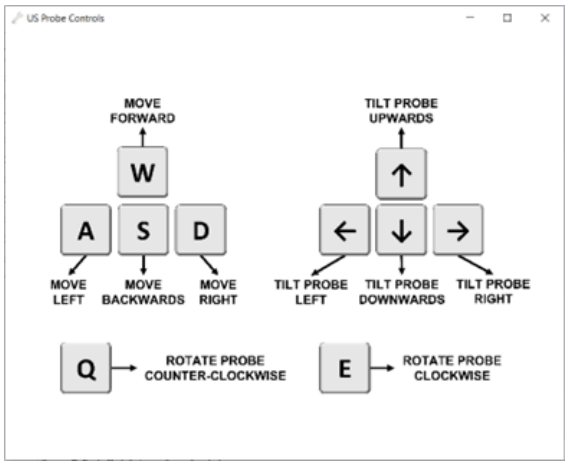


Fig. 4.14: Controles para manejar la sonda en el entorno de 3D Slicer. Adicionalmente se han integrado las teclas "Z" y "X" para acercar y alejar la sonda en su eje longitudinal.

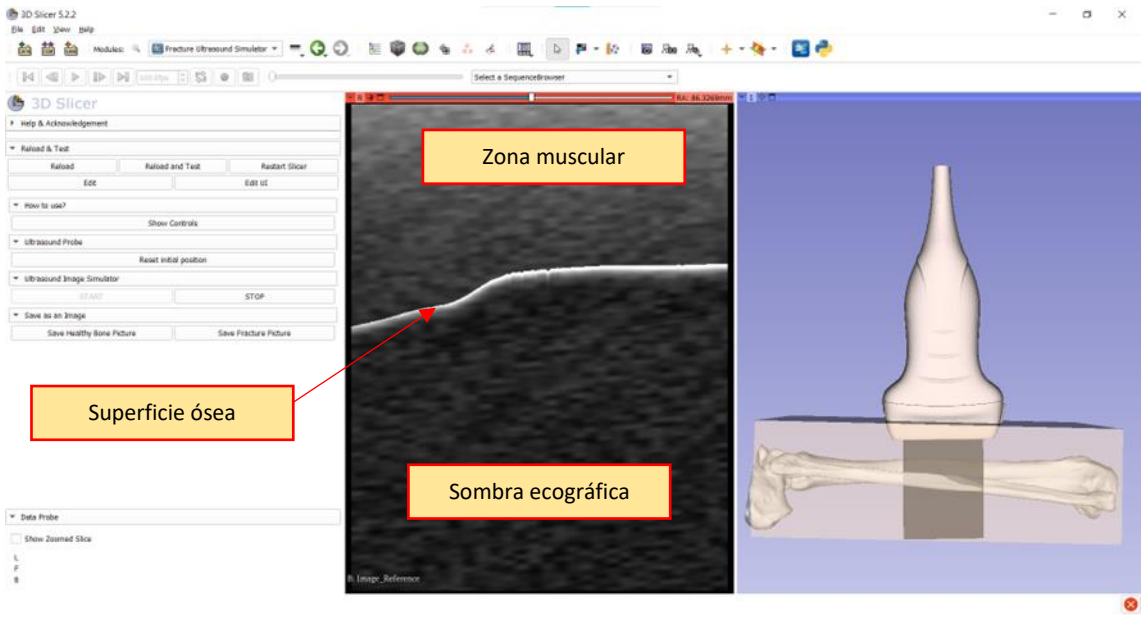


Fig. 4.15: Sistema de toma de imágenes simuladas en funcionamiento. En la ventana de la izquierda (Image_Reference) se muestra la imagen simulada, y en la ventana derecha se muestra el entorno con los modelos utilizados.

Durante el proceso se toman imágenes alternativamente para ir creando la base de datos de imagen ecográfica simulada, la cual ha sido objeto de esta parte del proyecto. La base de datos está formada por 50 imágenes de huesos sanos y 50 imágenes de imágenes de huesos fracturados. Para cada una de las carpetas se ha creado un README con la intención

de que quede constancia de los modelos de huesos que han sido empleados para esta base de datos. Se muestra un ejemplo en la Tabla 4.1:

Tabla 4.1: Tipo de fractura según el número de imagen en la base de datos de imagen ecográfica simulada.

Tipo de fractura según el número de imagen	
Número de imagen:	Tipo de fractura
0 – 9	Fractura diafisaria simple de húmero
10 – 19	Fractura a tercer fragmento de cúbito con ausencia de fragmento
20 – 29	Fractura diafisaria simple de tibia
30 – 39	Fractura diafisaria simple de fémur
40 – 49	Fractura a tercer fragmento de cúbito y radio

4.3.2 Imágenes reales en fantoma

Para realizar la creación de esta base de datos, es necesario primeramente tener en cuenta cómo se va a abordar la toma de imágenes, es decir, es necesario tener en cuenta todo el material hardware y software que se va a utilizar. En este caso se volverá a utilizar el módulo de Fracture Ultrasound Simulator, en el cual se tienen dos botones para guardar las imágenes en dos carpetas según el hueso este sano o fracturado.

Para este proceso de obtención de imágenes no basta con el módulo dentro de 3D Slicer; es necesario utilizar PLUS para que este haga de interfaz entre 3D Slicer y la sonda. Para ello, lo único que se ha hecho es abrir PLUS y se le ha otorgado el archivo de configuración que viene por defecto en el programa para el uso de la sonda Telemed que se va a emplear. También fue necesario tener abierto Echo Wave II, ya que como se pudo comprobar, este determina el comportamiento de la sonda a la hora de captar imágenes, y según como se configure se verá de dicha manera en 3D Slicer. Durante el proceso, de obtención de imágenes se ha ido jugando con el rango de frecuencias, puesto que según se cita en [33], es necesario tener en cuenta que a mayor frecuencia se trabaje mejor se detectarán las pequeñas fracturas o fisuras. Aunque debido a la densidad del material con el que se está

trabajando, y la profundidad a la que se encuentran los huesos en el fantoma se han utilizado frecuencias situadas entre los 5 y 10 MHz.

Ya conectada la zona al sistema, con el Echo Wave II configurado correctamente y 3D Slicer recibiendo la imagen que le otorga PLUS, solo queda abrir el módulo previamente mencionado, e ir tomando imágenes utilizando los botones configurados. Antes empezar a pasar la sonda encima del fantoma es necesario lubricar de forma correcta el fantoma con gel para ecografías. Este gel sirve para reducir la impedancia que pueda haber entre la superficie del fantoma y el cabezal de la sonda.

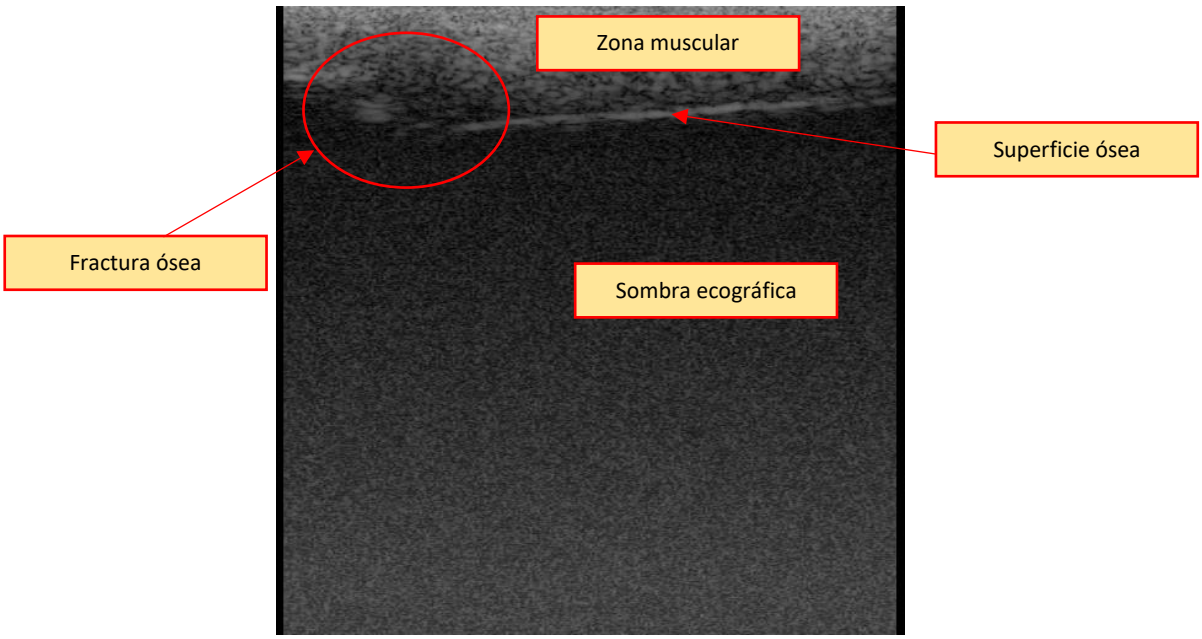


Fig. 4.16: Ejemplo de resultado obtenido en el fantoma.

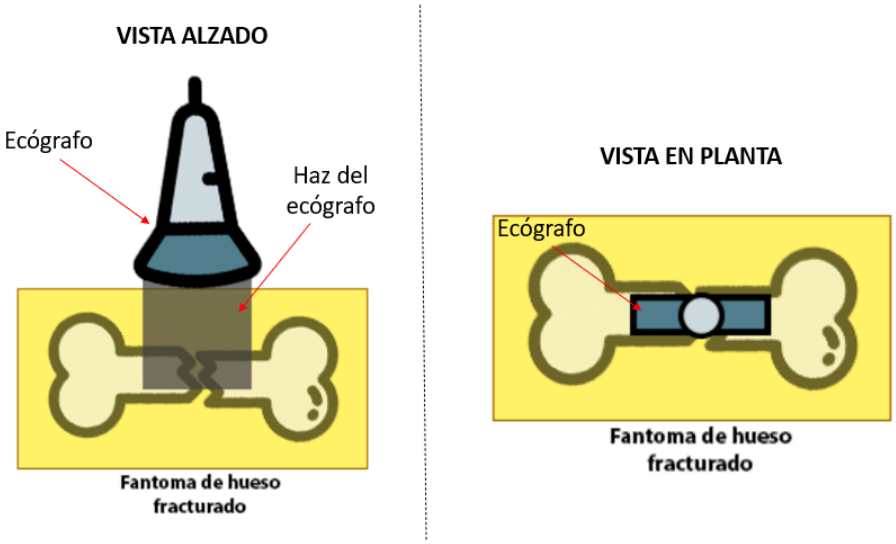


Fig. 4.17: Posición del ecógrafo en la que fue tomada la imagen de la figura anterior.

4.4 ENTRENAMIENTO DE LA RED NEURONAL Y DESARROLLO DEL ALGORITMO

Ya que el objetivo principal que sigue este apartado es efectuar la localización de fracturas tomadas con imagen de US, se parte de una clasificación binaria (es decir, reconocer si la imagen aportada representa una fractura o no) que posteriormente, tras realizarse algunas modificaciones en el código, se consigue que aporte facilidad a la hora de poder localizar la fractura en cuestión. Es por ello que el entrenamiento de la red neuronal estará enfocado en reconocer fracturas, generando posteriormente un algoritmo que permita ver visualmente en qué se está fijando la propia red para clasificar una imagen como "fracturada" o "no fracturada". La visualización de lo anteriormente comentado es lo que aportará al usuario la estimación de la zona que la red ha creído oportuna para diagnosticar dicha anomalía.

También, se ha creído oportuno aportar un porcentaje de certeza que indica la seguridad de la propia IA en clasificar la imagen como "fracturada", teniendo como objetivo aportar la información suficiente como para aportar un posterior diagnóstico del paciente. Como bien se ha comentado a lo largo de este proyecto, la intención de esta herramienta no es sustituir, sino complementar a modo de ayuda. Este apoyo va orientado, sobre todo, a aquellos profesionales en el ámbito de la salud que no estén muy familiarizado con el uso

de ecógrafos, ya que, como se ha comentado, la interpretación de dichas imágenes suelen ser un problema para la mayoría de estos.

Para el desarrollo del sistema que se ha creído oportuno seguir un flujo de trabajo dividido en cuatro bloques.

- Etiquetado de imágenes y preparación de las bases de datos sujetas a ser usados en el entrenamiento.
- Entrenamiento de la red neuronal. En este caso se utilizará la arquitectura de red ResNet-18, comúnmente utilizado en clasificación, pero en este caso la red ha sido adaptada a solucionar el problema en cuestión.
- Inferencia de imágenes en la red neuronal. Este apartado se hace a modo de prueba para comprobar la accesibilidad a la red entrenada, y comprobar si los datos aportados por la misma son correctos; para ello se hace inferencia con imágenes que están fuera de la base de datos de entrenamiento.
- Implementación en el módulo de 3D Slicer.

Siguiendo este flujo, se empezará a explicar de qué manera se han etiquetado las imágenes y cómo se ha llevado a cabo organización de la base de datos. Cabe destacar que, exceptuando la implementación del módulo, el resto de los apartados se han llevado a cabo en notebooks de Jupyter.

Para entender todo el procedimiento que se sigue en este apartado, es necesario tener en consideración cual es la lógica que sigue un entrenamiento de redes neuronales. Para poder ser entrenada necesita unos datos que se deben estructurar en:

- **Proceso de entrenamiento y conjunto de datos de entrenamiento:** Estos corresponden a la porción de datos que se utilizará para entrenar el modelo. En el entrenamiento se ajustan los pesos y los sesgos de las neuronas que conforman la red neuronal.
- **Proceso de validación y conjunto de datos de validación:** Esta muestra de datos, tal y como su propio nombre indica, se utilizan en el proceso de validación del modelo. En este proceso se proporciona una evaluación de los ajustes realizados al modelo en el proceso de entrenamiento, a la par que se ajustan los

hiperparámetros del mismo. Estos hiperparámetros pueden ser: la tasa de aprendizaje, el tamaño del lote, las épocas, entre otros.

- **Proceso de evaluación y conjunto de datos de evaluación:** Este conjunto de datos es utilizado para proporcionar una evaluación imparcial del ajuste de modelo final en el conjunto de datos de entrenamiento, dicho con otras palabras, este solo se usa una vez que un modelo está completamente entrenado.

Una vez entendida esta lógica, se aporta un resumen de la estructuración, y la cantidad de datos recogidos para cada una de las dos bases de datos empleando las dos técnicas previamente explicadas en el anterior apartado:

Base de datos con imagen ecográfica simulada (BD Simulada): Conformado por un total de 100 imágenes simuladas mediante el uso de PLUS y 3D Slicer, posteriormente subdividida en 70 imágenes para la base de datos de entrenamiento, 30 para la base de datos de validación.

Base de datos de imágenes obtenidas mediante un fantoma fabricado (BD Fantoma): Conformado por un total de 128 imágenes, posteriormente subdividida en 96 imágenes para la base de datos de entrenamiento, y 32 para la base de datos de validación.

Una vez definidas las dos bases de datos que se van a utilizar para entrenar al modelo, se definen a continuación dos bancos de imágenes que se van a utilizar para el proceso de evaluación de los modelos; uno de los bancos de datos servirá para evaluar al modelo que ha sido entrenado con imagen ecográfica simulada, y el otro para evaluar al modelo entrenado con las imágenes obtenidas del fantoma de hueso fracturado. Estos bancos de datos están conformados por un total de 30 imágenes cada uno.

Para decidir la distribución de datos entre el base de entrenamiento y validación, es común, en el marco de entrenamiento de redes neuronales, destinar entre un 20% y un 30% a la base de datos de validación, quedando el resto dentro de la de entrenamiento.

4.4.1 Etiquetado de la base de datos

Para realizar el correspondiente etiquetado, se ha elaborado un algoritmo que, de forma automática, ha creado una organización de carpetas de la siguiente forma:

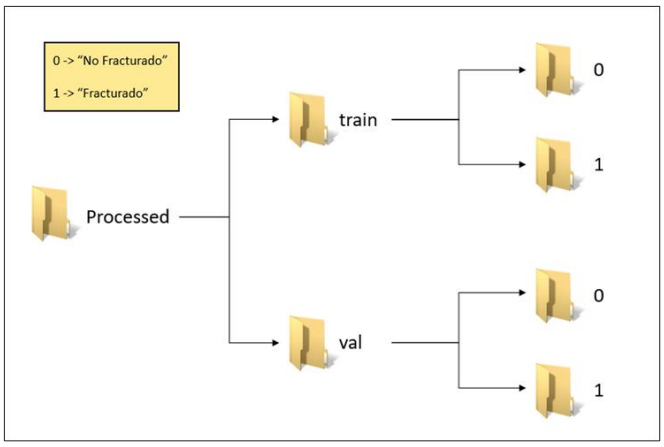


Fig. 4.18: Organigrama de carpetas realizado por el algoritmo mencionado.

Esta organización de carpetas es necesaria ya que posteriormente se realizará el etiquetado de imágenes en la base de datos mediante un objeto de Python que está diseñado para fijarse en el nombre de la carpeta que contiene la imagen para posteriormente etiquetarla. Esta información irá incluida en los metadatos de la imagen, las cuales han sido guardadas en formato ". npy". Cabe destacar que, tal y como se muestra en la Fig. 4.18, las imágenes han sido clasificadas en dos clases, las cuales son '1' para aquellas imágenes correspondientes a fracturas, y '0' para aquellas imágenes correspondientes a huesos sanos. Por otro lado, es importante comentar que las carpetas 'train' y 'val' corresponderán a las bases de datos de entrenamiento y validación correspondientemente. El bloque de código que se muestra en (anexo 10.1, Código 10.1) es el encargado de realizar la repartición de los datos según lo visto en apartados anteriores, es decir, suponiendo el caso de la base de datos de imagen ecográfica simulada, se tiene 100 imágenes en total, de las cuales 70 serán de la base de datos de entrenamiento (35 imágenes de fracturas y 35 de huesos sanos), y los 30 restantes serán de la base de datos de validación (15 imágenes de fracturas y 15 de huesos sanos).

4.4.2 Transformaciones y *dataloaders*

Tal y como se explica a inicios del apartado 4.4, es necesario realizar un proceso de entrenamiento y validación de una red neuronal para que pueda realizar tareas específicas, todo ello sin olvidar que estos dos deben de ir seguidos de un proceso de evaluación encargado de verificar que el modelo cumple con la tarea que le ha sido asignada correctamente.

Sin entrar en prolegómenos, el objetivo de este apartado es visibilizar que, para poder llevarse a cabo el proceso de entrenamiento y validación, es necesario definir unas transformaciones previas que detallan la forma en la que se infieren los datos en la propia red neuronal. Tanto el proceso de entrenamiento, como de validación, tienen sus propias transformaciones, las cuales, por lo general, no suelen ser iguales; todo dependerá del problema que se pretenda resolver. Un ejemplo de esto se puede ver en el anexo 10.1, Código 10.2.

Por otro lado, los *dataloaders* son componentes que se utilizan para facilitar la tarea de cargar y preparar los datos necesarios (ordenación de los datos) para el entrenamiento de una red neuronal. Entre las tareas relevantes que estos desempeñan destacan: Carga de datos, preprocesamiento, división en lotes o *batching*, y mezcla de datos o *shuffling*.

Se puede ver un ejemplo de configuración de *dataloader* en (anexo 10.1, Código 10.2). Tal y como se aprecia en la celda de código referenciada, se definen las instancias del “train_loader” y el “val_loader”. También, se ha aprovechado para mostrar la distribución de los datos de cara al entrenamiento.

4.4.3 Red neuronal convolucional ResNet-18

A la hora de hacer el entrenamiento se ha utilizado una red neuronal convolucional ResNet-18 para realizar una clasificación binaria de estas imágenes y detectar fracturas (véase Fig. 4.19). Tal y como se puede observar en la figura referenciada, esta arquitectura no deja de ser un símil a lo visto en el apartado 1.4.1, donde se explicaba el perceptrón multicapa.

El motivo por el que se ha elegido una arquitectura de red ResNet-18 se basa en varias consideraciones fundamentales. En primer lugar, ResNet-18 es capaz de realizar tareas de visión por computadora demostrando un rendimiento excepcional, incluida la clasificación de imágenes médicas. Además, ResNet-18 utiliza una técnica de aprendizaje profundo conocida como "conexiones residuales ". Las conexiones residuales es una técnica utilizada en redes neuronales profundas para abordar el problema de la degradación del rendimiento a medida que se incrementa la profundidad de la red. Dicho con otras palabras; si se dispone de una red con muchas capas, es muy probable que a medida que se van aplicando transformaciones no lineales y la red se vuelve más profunda el rendimiento de la misma se pueda ver afectado. La presencia de conexiones residuales

ayudaría a evitar la desaparición del gradiente (parámetro que indica la dirección y la magnitud en la que los parámetros de la red deben ajustarse para minimizar la función de pérdida) y facilitaría el entrenamiento eficiente de la red, lo cual es esencial en el caso de conjuntos de datos limitados.

Al tener una arquitectura relativamente simple comparado con otros tipos de redes con arquitecturas más grandes, hace que esta disponga de una eficiencia computacional que, para el tipo de sistemas que se están utilizando para el entrenamiento, se puedan realizar los entrenamientos sin ningún tipo de problemas.

Una arquitectura ciertamente similar en cuanto a simpleza y eficiencia computacional es la arquitectura VGG-11. Esta arquitectura, ya ha sido utilizada para detectar fracturas con imagen ecográfica, demostrando resultados bastante buenos en pacientes reales [31]. Es por ello, que a modo de prueba se ha seleccionado otra red semejante en cuanto a rendimiento y arquitectura.

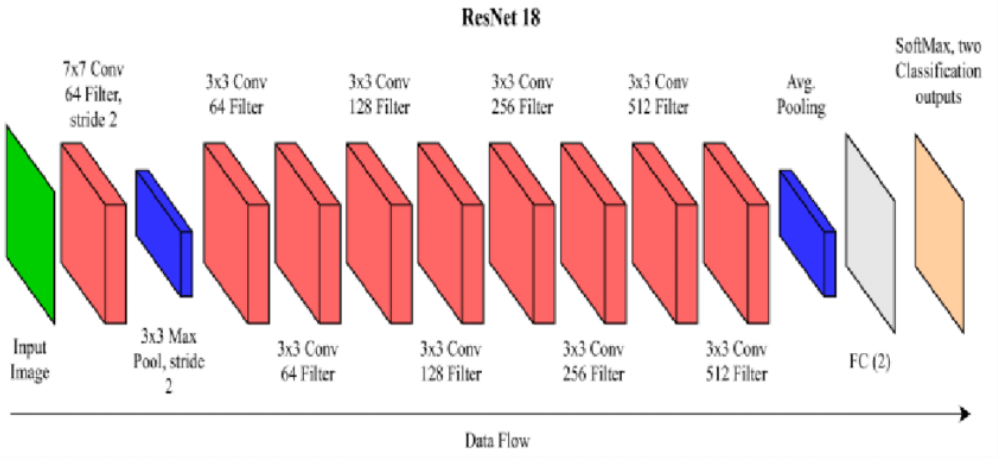


Fig. 4.19: Arquitectura del modelo de red neuronal ResNet-18, imagen extraída de [33].

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

Fig. 4.20: Comparación de la arquitectura ResNet-18 con otras arquitecturas ResNet, imagen extraída de [33].

4.4.4 Entrenamiento de la red neuronal

Hemos entrenado la red de dos formas distintas, comparando los resultados

A) Entrenamiento completo en una sola fase:

Este entrenamiento consiste en entrenar a la red neuronal desde cero, directamente con las imágenes obtenidas del fantoma de hueso fracturado. De esta se obtendrá un modelo entrenado que será sujeto a evaluación (véase Fig. 4.21).

B) Entrenamiento con aprendizaje transferido (*transfer learning*) en dos fases:

Este entrenamiento se ejecuta en dos fases. En la primera de ellas se entrena al modelo desde cero con las imágenes ecográficas simuladas, de esta se obtendrá un modelo entrenado para clasificar ese tipo de imágenes. La segunda fase consiste en entrenar ese modelo previamente entrenado, pero esta vez utilizando las imágenes obtenidas del fantoma de hueso fracturado (véase Fig. 4.21).

Todo este proceso con el fin de comparar y evaluar los resultados. En la siguiente figura se aprecia de forma gráfica el proceso seguido.

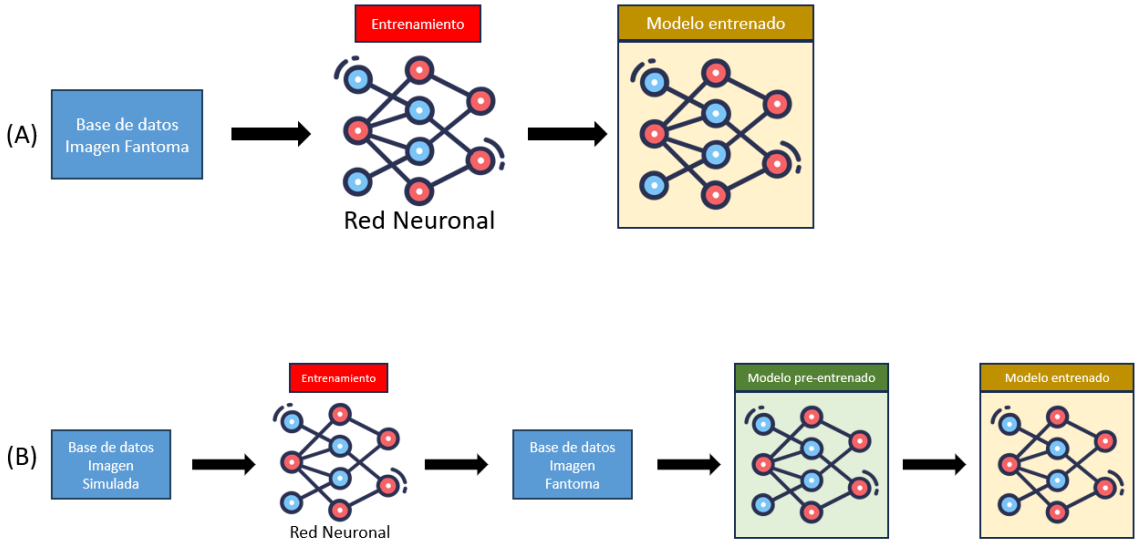


Fig. 4.21: Dos procesos de entrenamiento con la red neuronal convolucional ResNet-18.

Primeramente, antes de poder entrenar la red neuronal elegida, se debe editar la arquitectura de la red, en concreto los datos que espera como "*inputs*" y los datos que devuelve como "*output*". Para ello se muestra la arquitectura de ResNet-18 que nos otorga los modelos incluidos en la librería de torchvision.

```
torchvision.models.resnet18()

ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
  (layer1): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
    (1): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )

  (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
  (fc): Linear(in_features=512, out_features=1000, bias=True)
)
```

Fig. 4.22: Imagen la que se muestra como está estructurada la primera y última capa del modelo. Mostrándose rodeado en rojo lo que se debe cambiar para su correcto funcionamiento.

Tal y como se muestra en la anterior celda de código, esta espera recibir tres imágenes, para así devolver un total de mil resultados. Para solucionar esto, se ha creado una clase que modifica la capa de entrada, y la capa completamente conectada a la salida (*fuly connected layer*), de tal forma que la red espere recibir un solo canal de entrada, y devuelva un solo dato a la salida, en (anexo 10.1, *Código 10.3*) se muestra como se ha llevado a cabo esta modificación en la estructura de la red. Una vez definida la clase, lo único que queda por hacer es crear una instancia del modelo, donde además se ha comprobado que la arquitectura de la ResNet-18 ha cambiado a la deseada.

Algo que es necesario comentar, si se fijan en (anexo 10.1, *Código 10.3*) donde se define la clase que hereda de 'pl.LightningModule', es que en todo momento a la hora de hacer la inferencia del dato en el modelo es necesario ejecutar una función de activación sigmoide. La función sigmoide acota los valores devueltos por la red entre 0 y 1. Si nos fijamos en la Fig. 4.23, vemos como la función tiende de manera asintótica tanto a 1 como a 0, esto quiere decir que a medida que los valores de entrada aumentan (hacia infinito), la función sigmoide se acerca cada vez más a 1 pero nunca alcanza ese valor exactamente. Del mismo modo, a medida que los valores de entrada disminuyen (hacia menos infinito), la función sigmoide se acerca cada vez más a 0 pero nunca llega a ser 0 exactamente. Pero obviamente, si la red devuelve un valor muy alto, y tras pasar por la función sigmoide retorna un resultado de 0.9999, el sistema va a tender a redondearlo este resultado final a 1.

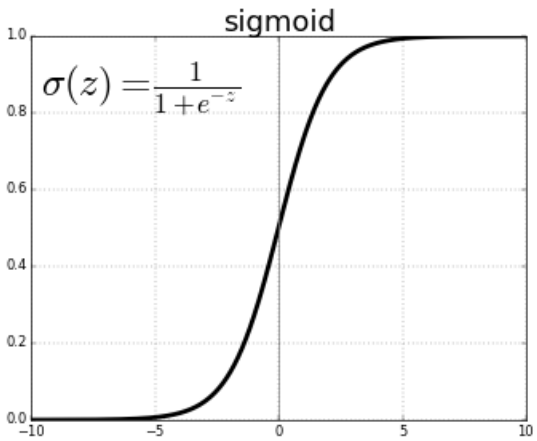


Fig. 4.23: Representación gráfica de la función sigmoide [34].

4.4.5 Checkpoint y trainer en Pytorch-Lightning

Ya una vez efectuado este último paso, solo queda definir el "*checkpoint*" y el "*trainer*", este último se utiliza para definir los parámetros para el entrenamiento, de los cuales, uno de ellos es el *checkpoint*. El *checkpoint* se utiliza para hacer un seguimiento de uno de los parámetros de la clase previamente definida, con el objetivo de guardar los estados del modelo que mejor se adapten a lo que buscamos, véase (anexo 10.1, Código 10.4). En el caso específico de este proyecto, se pretende guardar el modelo con mayor valor del 'val_accuracy'.

Siguiendo el flujo anteriormente comentado, solo queda definir el trainer, el cual se muestra en (anexo 10.1, Código 10.4).

El Trainer es una clase que se utiliza para controlar y facilitar el proceso de entrenamiento y evaluación de modelos de aprendizaje profundo. Proporciona una interfaz simplificada y potente para entrenar modelos en PyTorch, abstrayendo muchas de las tareas repetitivas y complejas asociadas con el entrenamiento de redes neuronales. Este se encarga de gestionar: el ciclo de entrenamiento, el ciclo de evaluación, los dispositivos en donde se ejecutará el entrenamiento, el guardado y registro del modelo mediante lo que se conoce como "*loggers*", y los *callbacks* o *checkpoints*.

Al objeto Trainer se le pasarán todos aquellos parámetros que se han definido anteriormente. Entre ellos destacan el parámetro que define en qué procesador se va a llevar a cabo el entrenamiento (en este caso en la GPU mediante CUDA), el *logger*, el cual en nuestro caso guardará todos los resultados obtenidos en cada '*step*' en un archivo CSV; el *callback*, en el cual se define el *checkpoint*; y finalmente las épocas del entrenamiento. Otros parámetros como el Learning Rate (lr) y la Loss Function fueron previamente definidas en la clase "FractureModel", en cuyo caso, se utilizó una función Adam de lr = 1e-3 y una Loss Function Binary Cross Entropy con un peso de 1, lo cual quiere decir que hay una distribución de clases equitativa. Una vez todo ha sido definido se podrá comenzar el entrenamiento. Para ello, lo único que se debe hacer es llamar a la función '*fit*' del *trainer*, la cual espera por parámetros: el modelo a entrenar, el 'train_loader' y el 'val_loader'; véase (anexo 10.1, Código 10.4).

Una vez el entrenamiento se ha ejecutado de forma satisfactoria, lo que procede es cargar el modelo que se ha detectado con mayor valor de 'val_accuracy' para, en base a este sacar los parámetros estadísticos que indican cuan fiable es el modelo y cómo de bien ejecuta su trabajo. También se extraen los datos del CSV, para ver cómo ha sido el aprendizaje el modelo y si este ha sido sobreentrenado. Para detectar esto, se extrae la función de pérdida o *loss function,* y la función que define el progreso de la exactitud o lo que se conoce también como *accuracy*.

4.4.6 Guardado del modelo

Finalmente, dando por bueno el resultado obtenido, guardar el modelo en formato PTH, para ello se ejecutan las dos instrucciones que se muestran en (anexo 10.1, Código 10.5). Para ser concretos, los pesos del modelo se guardan en un diccionario[2].

4.4.7 Uso de la técnica CAM (*Class Activation Map*) para obtener el mapa de activación

Antes de hablar sobre la inferencia a partir de imágenes en el modelo entrenado, es necesario hablar sobre la técnica que se usará para obtener el mapa de activación que ayudará al usuario en la interpretación de la imagen ecográfica tomada. La técnica utilizada es conocida como CAM (*Class Activation Map*), tal y como se enuncia en el título de este apartado.

Esta es una técnica de visualización que permite identificar las regiones más relevantes de una imagen para la predicción de una clase específica en un modelo de clasificación basado en redes neuronales convolucionales (CNN).

Cuando se entrena una CNN para la clasificación de imágenes, la red aprende a reconocer y extraer características relevantes de las imágenes en diferentes capas convolucionales. Estas capas convolucionales capturan características en diferentes niveles de abstracción, desde características simples como bordes hasta características más complejas como formas y objetos. Es por ello por lo que el CAM se basa en la observación de la última capa convolucional de una CNN, en la cual se tienen activaciones que reflejan las características aprendidas por la red para la clasificación. Por lo tanto, si se pudiera determinar qué partes

[2] Diccionario (Python): estructura de datos que permite almacenar y organizar elementos en pares clave-valor. Es una colección mutable, no ordenada y que no admite duplicados.

de las activaciones de la última capa convolucional contribuyen más a una clase en particular, se podría identificar las regiones importantes en la imagen para esa clase.

Trasladado al caso de este proyecto, lo que se pretende hacer en este modelo de ResNet-18 es multiplicar la salida de la última capa convolucional (BasicBlock 1 de la capa 4) $\boldsymbol{A_k}$ (compuesta por k canales) con los parámetros **ω** de la capa completamente conectada siguiente para calcular un mapa de activación **M**, de tal forma que, en términos simplificados se tendría lo siguiente [35].

$$M = \sum_{k} \omega_k A_k$$

Ec. 4.1: Ecuación que define el mapa de activación M.

Dicho con otras palabras, y a modo de que se entienda mejor este método, si se quisiera replicar este método en cualquier red neuronal, habría que localizar primeramente la última capa convolucional de la red. Una vez hecho esto, hay que ver la dimensión de los datos que devuelva esta última capa convolucional y comprobar que coincide con el número de pesos de entrada que tiene la neurona de la clase sujeta bajo análisis en la *fuly connected layer*. Es decir, suponiendo que la última capa convolucional devuelve una estructura de datos (224, 5, 5), esta nos estará indicando que esta capa nos está devolviendo 224 canales o matrices de 5x5; por lo tanto, atendiendo a la Fig. 4.24, el número de pesos de entrada que debería tener la neurona comentada previamente es de 224 pesos.

A continuación, se aporta un ejemplo para proporcionar al lector una idea visual de lo que realiza esta técnica.

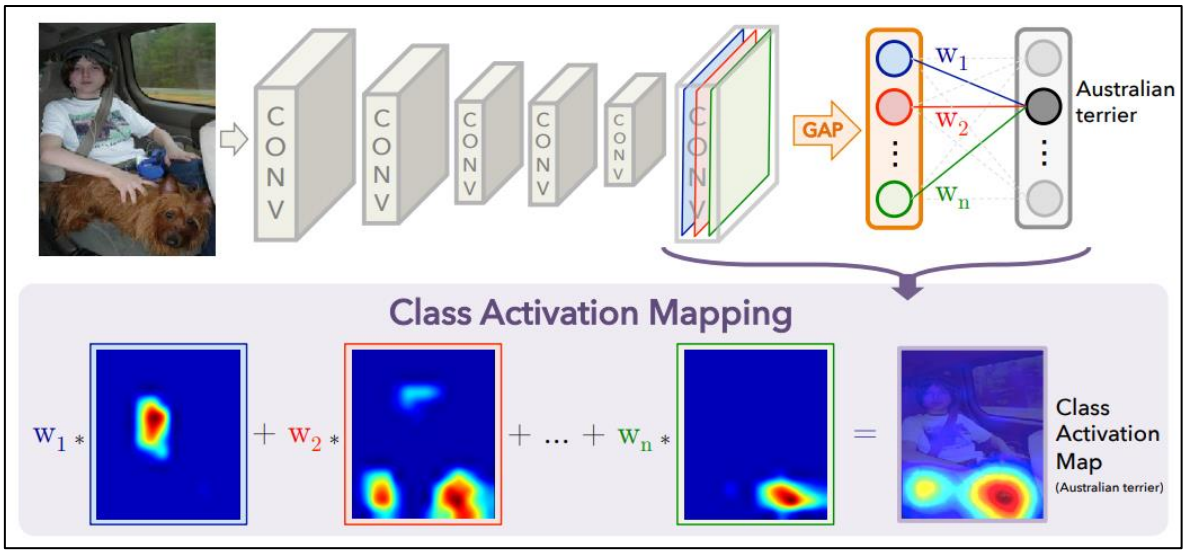


Fig. 4.24: En este proceso, el puntaje de clasificación predicho se asigna nuevamente a la capa convolucional anterior para generar los mapas de activación de clase [35].

El motivo por el que se ha utilizado esta técnica en vez de utilizar una red neuronal orientada a segmentación (como puede ser el caso de U-Net), es debido a que para realizar la localización de la fractura, no se necesita ser preciso en el contorno del área que se quiere marcar, ya que con localizar la zona en la que se encuentre la fractura es suficiente.

4.4.8 Implementación del sistema

En este apartado se relata de qué manera se ha llevado a capo la implementación del sistema destinado a clasificación. En base a este sistema, se extraerá un mapa de activación mediante el uso de la técnica CAM previamente comentada.

Para llevar a cabo esta implementación, se ha utilizado un *notebook* de Jupyter a modo de prueba para comprobar que el modelo realiza la tarea de clasificación satisfactoriamente.

Inferencia de imágenes en el modelo

Hay que tener en cuenta que en la práctica el modelo va a recibir imágenes en formato PNG y que este ha sido previamente entrenado con imágenes en formato NPY. Es por este motivo que se pretende comprobar que la clasificación de imágenes en formato PNG sean idénticas a la clasificación de imágenes en formato NPY. Para realizar este proceso se seguirán los dos pasos que a continuación se exponen:

- Inferencia con un objeto DatasetFolder, tal y como se hizo en el entrenamiento para la validación.

- Inferencia de imágenes individuales.

Cabe destacar que para realizar este proceso de comprobación se utilizarán las imágenes utilizadas en la validación del modelo, por tanto lo que se procede a hacer no es una evaluación o un test del modelo, sino más bien una simple comprobación de que este, a pesar de que el formato de imagen sea distinto, realiza correctamente la clasificación. Para un mejor entendimiento del proceso que se ha seguido para este apartado se expone la Fig. 4.25:

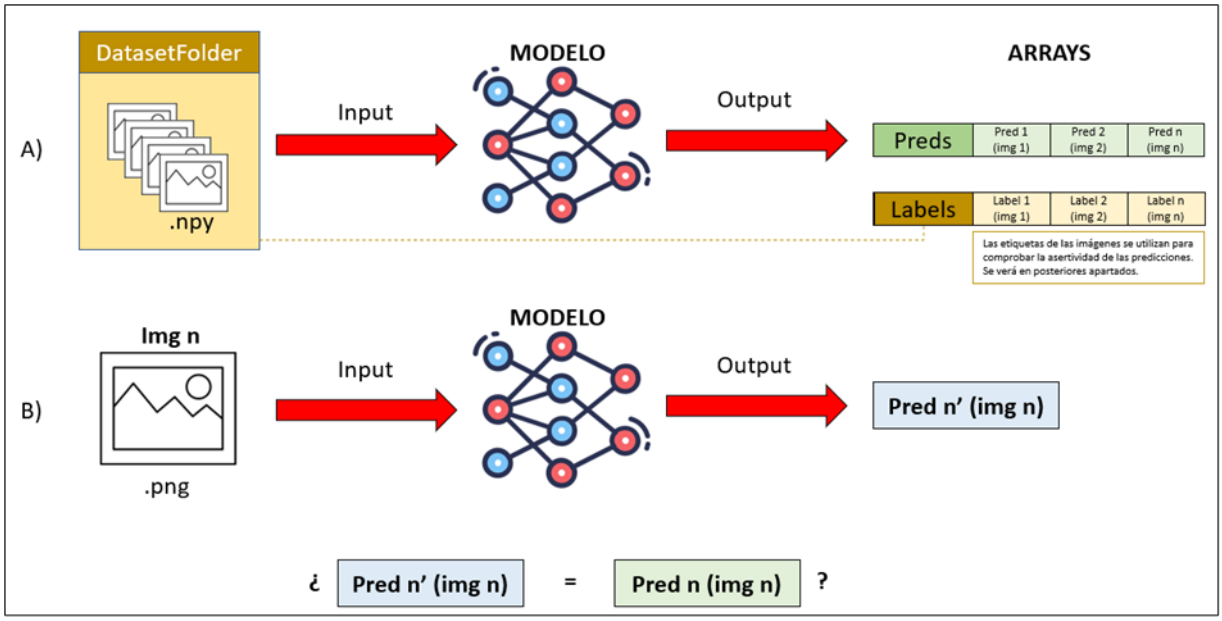


Fig. 4.25: A) Subproceso de inferencia con el DatasetFolder. B) Subproceso de inferencia con una imagen ".png", este subproceso se asemeja a la forma en la que trabajará el modelo en el módulo de 3D Slicer.

Primero se realiza la inferencia con un objeto DatasetFolder ya que esta inferencia se podría decir que es "segura", es decir, a esta se le pasa una base de datos de X imágenes, que son las mismas que se han utilizado durante el proceso de validación del modelo. Su resultado corresponde al que se obtuvo en el proceso de validación durante el entrenamiento en la *epoch* correspondiente a la del archivo que contiene los pesos de la red neuronal previamente guardada. Estas imágenes son pasadas al modelo en formato NPY.

Un detalle a tener en cuenta a la hora de realizar este tipo de inferencias, es que al modelo siempre habrá que especificarle que está en modo evaluación, mediante la instrucción 'model.eval()'. Esto implica que se desactiven ciertas capas o funcionalidades específicas que se utilizan durante el entrenamiento, como la regularización o la normalización por lotes. Esto asegura obtener predicciones precisas y estables en lugar de realizar ajustes en los pesos del modelo.

De vuelta con el tema principal, a la hora de definir la red fuera del notebook de entrenamiento, no se utilizará Pytorch Lightning ya que hay muchos sistemas que todavía no lo soportan, es por este motivo que, siempre que se quiera cargar el modelo, se debe crear una clase que pueda sostener al mismo, para ello se utilizará una clase que herede de nn.Module. Una vez definida la clase, se creará la instancia del modelo para poder realizar la inferencia en el mismo.

Una vez se ha creado la clase del modelo, se procederá a cargar el modelo con los pesos del modelo entrenado, y se trasladará al mismo a la CPU o a la GPU, según el valor que contenga la variable '*device*', véase (anexo 10.2, Código 10.7). Seguidamente, ya hechos los dos pasos anteriores, solo queda definir las transformaciones que se le van a aplicar al DatasetFolder para pasar a hacer la inferencia. Nótese que las transformaciones a utilizar son las mismas que se definieron para los datos de validación en el notebook destinado al entrenamiento, siendo las dos únicas transformaciones 'transforms.ToTensor()', la cual transforma a tensor el array de numpy correspondiente a la imagen; y finalmente 'transforms.Normalize(mean, std)', utilizado comúnmente para normalizar los valores de los datos de una imagen o un tensor en un rango específico. Los valores que toman los parámetros que recibe la función previamente mencionada corresponden al resultado de la media y la desviación típica del conjunto de imágenes de entrenamiento. Aunque, se ha probado a utilizar el valor de la media y la desviación típica correspondiente a una sola imagen, y el resultado apenas varía en una décima. En (anexo 10.2, Código 10.9) se ve diferenciado claramente el proceso seguido a la hora de realizar el proceso A) y el B) referenciados en la Fig. 4.25.

Implementación del *Class Activation Map*

Para llevar a cabo esta implementación es necesario tener en cuenta el siguiente bloque de código referenciado (anexo 10.2, Código 10.10), pues se hará alusión a este para entender el proceso que se ha seguido.

Para poder abordar esta implementación se deben implementar dos puntos:

1) Crear una clase en la que poder cargar los pesos del modelo previamente entrenado, y que en base a este se pueda extraer el mapa de activación.
2) Una función que, en base a un objeto de la clase creada anteriormente, permita realizar la operación explicada en el apartado 4.4.7 para obtener el mapa de activación.

Si nos fijamos en la clase creada, en el constructor de la propia clase hay definidas cuatro instancias. La primera instancia define el tipo de red neuronal que se va a utilizar (en este caso es la ResNet-18), y las otras tres instancias definen las modificaciones que se le aplican a la arquitectura que tiene por defecto la red neuronal seleccionada.

La última de las instancias, define una modificación del modelo de red neuronal que consiste en quitarle al modelo las dos últimas capas (capa *average pool* y *fuly connected layer*) y quedarse con un nuevo modelo que solo llegaría hasta la última capa convolucional del mismo. Es a esta última modificación a la que realmente se le hará la inferencia, obteniendo de este un tensor de características de la imagen de tamaño (512, 7, 7), lo que corresponde a 512 canales, formado por matrices de características de 7x7. Si echamos la vista atrás, esto vendría a ser un análogo a lo visto en la Fig. 4.24, en la cual se tienen unas láminas de colores dentro de la última capa convolucional del modelo, que viene a representar los 512 canales de matrices 7x7 que serán multiplicados por los pesos que tiene la *fuly connected layer*.

En la función desarrollada que implementa lo visto en la Ec. 4.1 se realizan una serie de arreglos matemáticos para, mediante el uso de una sola multiplicación matricial, poder obtener el mapa de activación deseado. Para ello, hay que tener en cuenta que la matriz de pesos es una matriz fila de 512 pesos **ω**, es decir, se tiene una matriz cuyo tamaño es (1, 512). Según lo visto en la ecuación ya mencionada, se deberían multiplicar cada uno de estos 512 pesos por las 512 matrices de 7x7, para finalmente sumarlas todas. Pero para

solucionar esto de una manera práctica se hace un reajuste, en el que se utiliza la función *reshape* para hacer del conjunto de matrices (512, 7, 7) una matriz semejante cuyo tamaño es (512, 49).

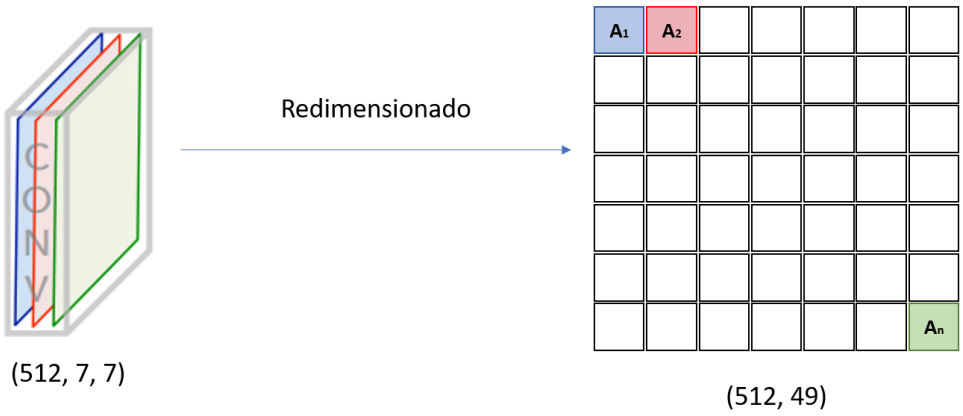


Fig. 4.26: Redimensionamiento del paquete de 512 matrices de 7x7 a una nueva matriz de (512, 49)

Esta ya no es un conjunto de 512 matrices, sino es una única matriz de 512 filas por 49 columnas. Finalmente, se hace la multiplicación matricial de la matriz de pesos (1, 512) por la nueva matriz única cuyas dimensiones son (512, 49). Obteniendo de esta una matriz fila de 49 valores, que haciendo el reajuste adecuado con la función *reshape* se puede transformar en una única matriz de 7x7 que corresponde al mapa de activación.

Una vez se tiene el mapa de activación, al tener un tamaño de 7x7, se le hace un cambio de escala, en el que se le pasan el ancho y el alto de la imagen que ha sido sujeta a análisis.

El resultado final de la función 'heatmap_creation' viene a ser algo como lo que se muestra en la Fig. 4.27:

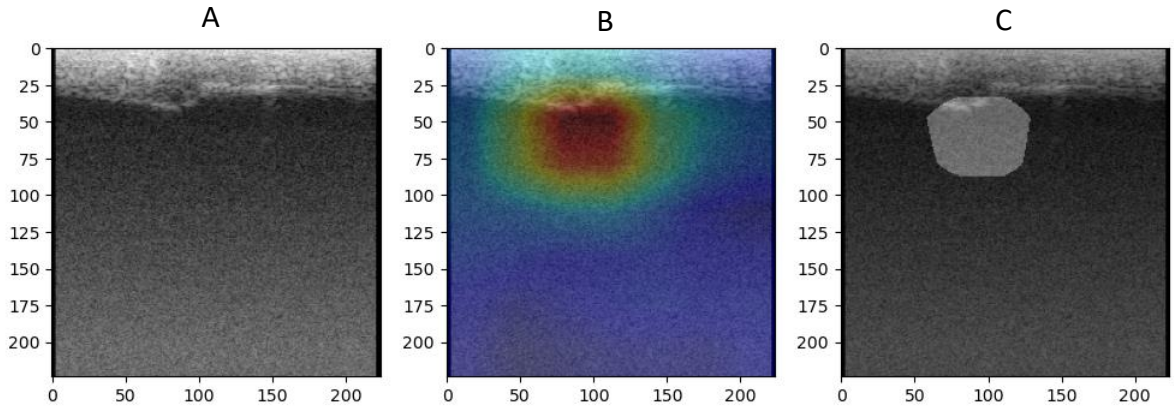


Fig. 4.27: (A) Imagen original la cual se pretende clasificar. (B) Mapa de Activación. (C) Segmentación del mapa de activación.

En esta última imagen, se muestra el resultado que ofrece la red neuronal, tras haber computado el mapa de activación en su totalidad. En resumen, tras ver el procedimiento que se ha seguido, el lector se puede dar cuenta que lo que se ha hecho es utilizar una inteligencia artificial orientada a clasificación, y haciendo una serie de ajustes, se ha conseguido reutilizar esta misma red neuronal entrenada a la misión de localizar fracturas en huesos con un mapa de activación, y en base a este, hacer una segmentación de la zona afectada. Todo esto que se ha comentado, se verá detallado de mejor forma en el siguiente apartado, en el cual se tratará cómo se ha desarrollado el módulo que implementa todas las funcionalidades, marcadas como objetivo de este proyecto.

4.4.9 Métricas utilizadas para la evaluación de los modelos

El rendimiento de los modelos de clasificación desarrollados se va a evaluar mediante el cálculo de las siguientes métricas: exactitud (*accuracy*), precisión, valor F1, sensibilidad (*recall*) y la especificidad. Además, también se han analizado la matriz de confusión, la función de pérdida y la función de exactitud.

A continuación, se muestran las definiciones y expresiones matemáticas de cada uno de los parámetros utilizados:

Exactitud

La exactitud, matemáticamente se define como el número de predicciones correctas dividido por el número total de predicciones realizadas. Esta es capaz de proporcionar una medida general de qué tan bien el modelo clasifica correctamente las instancias.

$$Exactitud = \frac{Verdaderos\ Positivos\ +\ Verdaderos\ Negativos}{Verdaderos\ Positivos\ +\ Verdaderos\ Negativos\ +\ Falsos\ Positivos\ +\ Falsos\ Negativos}$$

Ec. 4.2: Ecuación de exactitud.

Precisión

La precisión es un parámetro que mide la proporción de instancias positivas correctamente identificadas con respecto al número total de instancias clasificadas como positivas. A nivel conceptual, la precisión se suele ver como la dispersión del conjunto de valores obtenidos a partir de mediciones repetidas de una magnitud, obteniéndose mayor precisión cuanto menor es la dispersión.

$$Precisión = \frac{Verdaderos\ Positivos}{Verdaderos\ Positivos\ +\ Falsos\ Positivos}$$

Ec. 4.3: Ecuación de precisión.

Sensibilidad

La sensibilidad mide la proporción de instancias positivas correctamente identificadas con respecto al número total de instancias positivas. En otras palabras, esta es la capacidad del modelo para encontrar todas las instancias positivas.

$$Sensibilidad = \frac{Verdaderos\ Positivos}{Verdaderos\ Positivos\ +\ Falsos\ Negativos}$$

Ec. 4.4: Ecuación de sensibilidad.

Valor F1

El valor F1 se emplea para resumir la precisión y la sensibilidad en una sola métrica. Esta es de gran utilidad cuando la distribución entre las clases es desigual.

$$Valor\ F1 = 2 \times \frac{Precisión\ \times Sensibilidad}{Precisión + Sensibilidad}$$

Ec. 4.5: Ecuación de valor F1.

Matriz de Confusión

La matriz de confusión es una tabla que muestra la cantidad de predicciones correctas e incorrectas realizadas por un modelo de clasificación en cada clase. En una matriz de confusión típica para un problema de clasificación binaria, se muestran cuatro valores: verdaderos positivos, falsos positivos, verdaderos negativos y falsos negativos. Estos valores permiten analizar el rendimiento del modelo en términos de los diferentes tipos de errores cometidos.

$$Matriz\ de\ Confusión = \begin{pmatrix} Verdaderos\ Positivos & Falsos\ Positivos \\ Falsos\ Negativos & Verdaderos\ Negativos \end{pmatrix}$$

Ec. 4.6: Matriz de confusión.

Especificidad

La especificidad aplicada al ámbito médico, se puede ver como la capacidad de una prueba para identificar correctamente a las personas que no tienen una determinada condición o enfermedad.

$$Especificidad = \frac{Verdaderos\ Negativos}{Verdaderos\ Negativos\ +\ Falsos\ Positivos}$$

Ec. 4.7: Ecuación de especificidad.

A parte de estos parámetros estadísticos, se hará a uso también de la curva ROC (Receiver Operating Characteristic) y el AUC (Area Under the Curve) para así ofrecer más información de las características del modelo.

Curva ROC

Este es un gráfico que representa la tasa de verdaderos positivos (Sensibilidad) en el eje Y y la tasa de falsos positivos (1-Especificidad) en el eje X. Cada punto en la curva ROC representa un umbral de clasificación diferente. El umbral determina el punto de corte para decidir si una instancia se clasifica como positiva o negativa. Al variar el umbral, se obtienen diferentes tasas de verdaderos positivos y falsos positivos, lo que permite trazar la curva.

AUC

El AUC es el área bajo la curva ROC. Es una medida numérica que proporciona una medida de la capacidad de discriminación del clasificador. El AUC varía entre 0 y 1, donde 0 indica un clasificador que siempre se equivoca y 1 indica un clasificador perfecto. Cuanto mayor sea el AUC, mejor será la capacidad de clasificación del modelo.

4.5 Desarrollo del módulo de 3D Slicer

Para comentar el procedimiento que se ha llevado a cabo para implementar el módulo de 3D Slicer, se deben asimilar previamente varios conceptos. Primeramente, es necesario mencionar cómo se diseña la interfaz de usuario (IU) de un módulo de 3D Slicer. Este es un sistema bastante intuitivo en el que se le asignan a los botones, u otros interactuadores, nombres de variables que posteriormente serán utilizadas en el propio código del módulo para definir la funcionalidad que tiene cada uno de estos "objetos" (entiéndase por qué se ha escrito entre comillas, haciéndose alusión a que se utilizará Python para programar dicho módulo, y este es un lenguaje de programación orientado a objetos). El diseño de estos interactuadores, previamente mencionados, se llevan a cabo en un programa complementario llamado "Qt Designer", del cual a continuación se muestra una imagen.

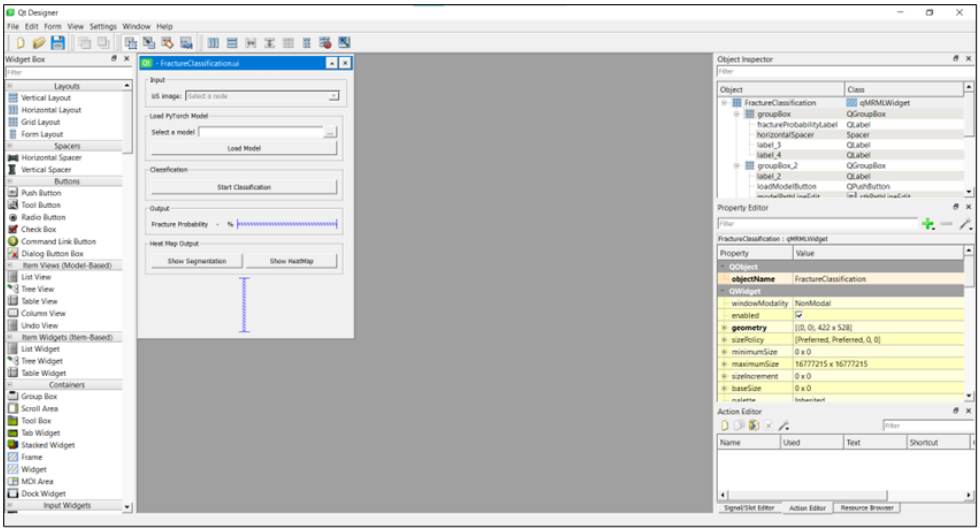


Fig. 4.28: Entorno de desarrollo de Qt Designer.

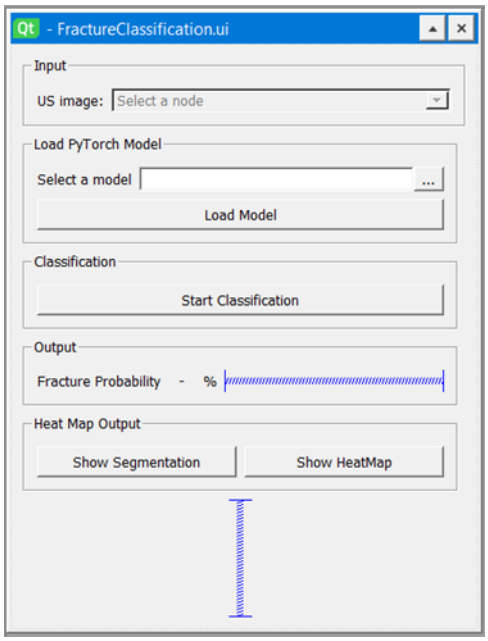


Fig. 4.29: Previsualización de la IU que se está diseñando, la cual se muestra dentro del entorno de desarrollo.

En las imágenes posteriormente mostradas se pueden ver el entorno donde se lleva a cabo este diseño y, además, también se aprecia el diseño que se ha hecho para nuestro módulo de 3D Slicer. A cada uno de estos botones que conforman la IU del módulo, se les ha asignado un nombre como interactuador, dicho nombre será posteriormente utilizado para programar la funcionalidad que llevaran asignados cada uno de ellos. A continuación, se adjunta un ejemplo en el que se muestra en que parte del entorno de desarrollo de la IU se define el nombre del objeto:

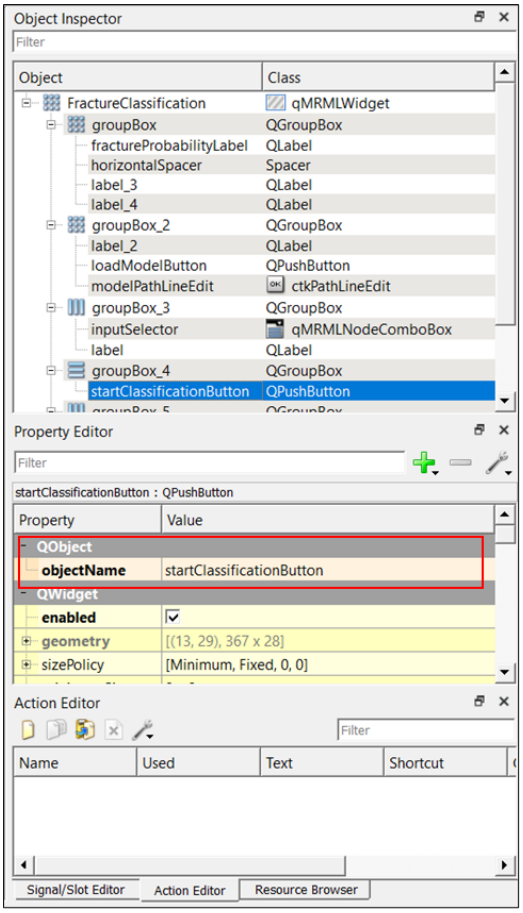


Fig. 4.30: Ejemplo de los parámetros correspondientes a cada botón. En este ejemplo se resalta el nombre del objeto.

En esta parte del entorno de desarrollo de la IU, se muestra la jerarquía de objetos interactuadores que se despliegan en la propia IU, además, se marca en rojo el área donde se define el nombre del interactuador, donde en la parte inmediatamente inferior, se detallan algunas de las características de este.

Una vez entendido el proceso que rige el diseño de una IU con Qt Designer, se procede a explicar el cómo se ha abordado la programación del algoritmo que rige el comportamiento del sistema.

Primeramente, para entender cómo funciona la programación de un módulo de 3D Slicer hay que entender de qué manera se estructura este código. Para ello, es necesario recurrir a la Fig. 3.2, que define de forma sencilla los diferentes grandes apartados en los que se divide el código escrito en Python de 3D Slicer. Es por ello que siguiendo dicho esquema se explica por bloques lo que se ha hecho en cada uno de los apartados. Para entender

correctamente las explicaciones, es necesario revisar cada uno de los anexos al código que ser referencian en cada uno de los bloques.

4.5.1 Importaciones necesarias para el módulo

Si vemos el código aportado en (anexo 10.3, **¡Error! No se encuentra el origen de la referencia.**), en las 6 primeras líneas se define lo que sería el paquete básico de librerías que se suele usar para programar un módulo en 3D Slicer. Es de ver que no todas estas librerías están siendo utilizadas, pero aun así, estas se dejan por si acaso de cara a un futuro se le quisiera hacer una modificación al módulo. Por otro lado, de la línea 9 a la 16, se tiene un '*try*' (declaración que se utiliza para manejar excepciones) que importa el paquete de librerías de PyTorch, las cuales serán objeto principal de este proyecto. También se ha utilizado un bloque de código parecido al utilizado anteriormente, para importar la biblioteca de CV2 de OpenCV (Open Source Computer Vision Library), esta se suele utilizar en aplicaciones de visión artificial, como reconocimiento de objetos, detección de rostros, procesamiento de imágenes médicas, entre otras. Su aplicación dentro de este proyecto se verá posteriormente a medida que se vaya exponiendo el código utilizado para elaborar este módulo.

4.5.2 Class Description

Lo que se muestra en (anexo 10.3, **¡Error! No se encuentra el origen de la referencia.**), sería el bloque de código al que, por conveniencia en el informe de este proyecto, se le ha designado como "Class Description". En esta clase, tal y como se ha podido apreciar, hay información tan relevante como: el nombre del propio módulo, la categoría a la que pertenece dentro del conjunto de módulos en el que se le ha enmarcado (se verá posteriormente), las personas que han contribuido en el desarrollo de este proyecto, etc.

4.5.3 Widget Class

En este bloque de código, tal y como se explica en la Fig. 3.2, se gestionarán los métodos y funciones que van asociados a cada uno de los interactuadores que utilizará el usuario. En la referencia aportada (anexo 10.3, *Código 10.13*), se puede observar el inicio de la Widget Class; empezando por el constructor de la clase.

La estructura que sigue un módulo de 3D Slicer a la hora de ser programado, viene definido por el "esqueleto de código" (por llamarlo de alguna manera) que ofrece la propia

aplicación para que cualquier usuario que quiera desarrollar su módulo dentro de 3D Slicer pueda hacerlo; es por ello que hay líneas como la 53 y 54 (véase nuevamente la referencia (anexo 10.3, *Código 10.13*)) que ya vienen definidas dentro del propio "esqueleto de código" para que el usuario no se centre en hacer que el módulo funcione desde cero, sino que este invierta su esfuerzo en procurar que las funcionalidades de las que se van a dotar al módulo funcionen correctamente. Es por ello que no es objeto de este proyecto comentar cada una de las líneas o bloques de código que ya el propio 3D Slicer te ofrece, sino que se limitará a comentar aquello que ha sido cosecha propia del autor. En este caso, dentro del constructor de la clase 'FractureClassificationWidget' se han definido unos *flags* los cuales posteriormente se podrá ver su uso, y también se ha definido un atributo que crea un objeto de la clase 'FractureClassificationLogic' la cual corresponde a lo que en la Fig. 3.2 se ha definido como Logic Class. Por último, y no menos importante, en la línea 63 de código se ha hecho un llamamiento a la propia clase 'FractureClassificationWidget' con la biblioteca de 'slicer' para que el autor de este proyecto pudiera hacer pruebas dentro del propio 3D Slicer en la Python Console.

Algunas de las funciones que se encuentran después del constructor de la clase vienen definidas por el propio código predefinido por 3D Slicer; algunas de estas funciones predefinas, las cuales no se han tocado para nada son: la función 'setup', 'clean', 'enter', 'excit' y 'setupUi'. Esta última, establece una conexión entre el archivo ".ui" que genera el "Qt Designer" después de haber definido la IU, y las funcionalidades que se definen en el propio módulo de 3D Slicer.

Inmediatamente después del 'setupUi' encontramos las funciones 'setupConnections' y 'disconnect'. Estas están definidas dentro del "esqueleto de código" que el propio 3D Slicer te proporciona, pero evidentemente estas están vacías por defecto, ya que aquí es donde utilizamos los nombres asignados a los distintos objetos interactuadores, que se definieron dentro del Qt Designer. En el caso de la función 'setupConnections', se utiliza el 'self.ui', para llamar a los distintos interactuadores. Una vez hecho esto, conectamos estos interactuadores a las distintas funciones del 'FractureClassificationWidget' que utilizarán las funciones de lo que sería la Logic Class para que se desempeñe la función deseada. Otra función de vital importancia es 'updateGUIFromMRML', la cual será encargada de actualizar el estado de los interactuadores según se vayan ejecutando las distintas

funciones, o simplemente, se haya hecho algún cambio en la escena. En este último método es donde se han utilizado como apoyo los *flags* previamente definidos en el constructor de la clase para actualizar el estado de los interactuadores.

De forma seguida a lo anteriormente mostrado, se establecen dos funciones. La primera de ellas 'onInputSelectorChanged' la cual lo único que hace es llamar a la función 'updateGUIFromMRML' para asegurar de que hay un nodo detectado por el interactuador 'inputSelector'. Una vez se ha detectado dicho nodo, se procede a mostrarlo en las diferentes vistas. Por otro lado, la función 'onLoadModelButton' adquiere la dirección al directorio que se le ha pasado a través del interactuador 'modelPathLineEdit'. Una vez adquirida la dirección se le pasa por parámetros a la función 'loadModel'; la cual pertenece a la Logic Class del propio código. Una vez hecho esto, es común en todas las funciones volver a acudir a la función 'updateGUIFromMRML', donde se revisan *flags* y otras premisas para habilitar o deshabilitar los botones en sus correspondientes casos.

Siguiendo el flujo explicativo establecido se tiene la función 'onStartClassificationButton'. La función principal de este método se centra en devolver dos parámetros, los cuales serán: la probabilidad que tiene un paciente de sufrir una fractura en base a la imagen suministrada, y el mapa de calor de la fractura, o mejor conocido como el mapa de activación. Una vez hecho esto, se mostrará en la IU la probabilidad de fractura, se obtendrá el volumen del mapa de activación y se obtendrá la segmentación de la fractura en base a este último.

Las funciones 'onShowMapa de activación' y 'onShowSegmentation' hacen básicamente la misma función, pero estas están referidas a distintos nodos. En 'onShowHeatmap' se regulará la opacidad del mapa de activación, el cual ha sido colocado en el *foreground* de la escena, para así poder visualizarlo. Esta regulación vendrá dada en base a los *flags* vistos anteriormente en el constructor de la clase. Una vez la opacidad ha sido regulada, lo último que se hace es cambiar el nombre del botón de la IU que regula esta funcionalidad. Por otro lado, con la función 'onShowSegmentation' se hace lo análogo a lo ya visto. Lo único que cambia en este, es que no se regula la opacidad del nodo, sino que directamente se regula la visibilidad del mismo.

Una vez visto todo lo referente a la Widget Class, se pasa a analizar de la misma manera la Logic Class, en la cual vienen definidas todas las funciones que actúan de forma directa sobre los nodos de la escena.

4.5.4 Logic Class

De igual manera que se ha visto en la Widget Class, se pasa a analizar el constructor de la Logic Class, la cual realmente tiene por nombre 'FractureClassificationLogic'. Para seguir la explicación es necesario tener en cuenta que continuamente se hará alusión al código (anexo 10.3, Código 10.20).

Tal y como se puede ver en el constructor de la clase, se han definido muchas variables vacías, las cuales, a medida que el usuario va ejecutando las distintas funciones, irán tomando valores. Entre las variables vacías caben destacar las nombradas como 'classificationModel' y 'segmentationModel', las cuales son encargadas de alojar en ellas el modelo que llevará a cabo la clasificación y la segmentación de la imagen ecográfica respectivamente. Otra de las variables más destacables dentro de este bloque de código es la nombrada como 'transformations'. Esta variable se utiliza en el momento que se le tengan que hacer las transformaciones pertinentes a un nodo de la escena, o dicho en otras palabras, para poder hacer la inferencia correctamente (explicado anteriormente en el apartado 4.4.2) se deberán hacer una serie de transformaciones que permitirán que el modelo realice las predicciones. Por último, no se puede olvidar mencionar la variable 'redSliceLogic', la cual es una instancia encargada de alojar como objeto la '*red slice*' que corresponde con la vista axial dentro de 3D Slicer.

El método inmediatamente posterior al constructor de la clase es 'displayVolumeInSliceView', en la cual se ve como infiere directamente sobre esta *red slice* para mostrar en esta, la imagen que el propio usuario habrá arrastrado sobre la escena. Esta función fue utilizada en 'updateGUIFromMRML' definida en la Widget Class. A esta función se le ha de pasar el nodo sujeto a análisis por parámetros para que este pueda ser mostrado correctamente.

Por otro lado, se tiene la función 'loadModel', la cual se encarga de cargar los dos modelos explicados previamente. Para ello se ha procedido de igual manera que lo visto en el apartado 4.4.8, en el que se explica cómo se hace el proceso de inferencia de las imágenes

para poner a prueba el modelo. Con respecto a esto último, la única variación que se puede apreciar en el código, es el parámetro 'map_location' que recibe por parámetros la función 'torch.load'. El propósito de usar 'map_location' es especificar el dispositivo en el que se cargarán los datos del modelo guardado. Esto se hace ya que puede ser que el modelo se haya guardado originalmente en un dispositivo diferente al que se está utilizando para cargarlo.

Dejando de lado el parámetro 'map_location' dentro del 'torch.load', cabe destacar la funcionalidad del *try* cada vez que se intenta cargar uno de los modelos. Esto se hace con afán de gestionar las excepciones, y que notifique al usuario qué es lo que está fallando a la hora de cargar el modelo, en caso de que este estuviera dando algún error.

Si seguimos comentando el código, después de 'loadModel', nos encontramos con 'convertHeatMapToVolume'. Tal y como se detalla paso a paso en el código aportado en el anexo de este proyecto, el método recibe el mapa de activación por parámetros y lo convierte en un volumen o nodo, colocando este en el foreground del 'red slice' como se puede ver en las líneas que van de la 331 a la 334, definiendo por defecto la opacidad de este foreground a 0. Por último, se le aplica un "lookup table" que será clave para la interpretación de este mapa de activación.

Inmediatamente después a 'convertHeatMapToVolume' encontramos el método 'heatMapSegmentation'. Como se puede verse en el código, esta segmentación se hace en base a lo que se conoce como una "*threshold segmentation*". Este tipo de segmentación se lleva a cabo mediante una operación lógica AND, en la que solo se cogen los valores que sean iguales o superiores a 0.65 en el mapa de activación, haciendo que estos adquieran el valor de 1 y el resto de valores tomen el valor 0. Seguidamente se aplica un *try* que borrará los nodos de segmentación de la escena en caso de que estos ya existan. Una vez hecho esto, en base al array de numpy de 0 y 1 creado por la *threshold segmentation*, se añadirá dicho segmento a la escena mediante función 'addSegmentFromNumpyArray', definida posteriormente.

En este punto, llegamos al método que más peso tiene en el código, el cual es 'startClassification'. En el primer bloque de este método, se realizan las transformaciones pertinentes para hacer la inferencia tal y como se ha visto en los notebooks. Seguidamente,

en el segundo bloque de esta función se lleva a cabo una comprobación que asegura si los modelos en los que se van a hacer las inferencias realmente existen o no. Una vez comprobada la existencia de estos modelos, se hace la inferencia, en cuyo caso se utiliza la sentencia “with torch.no_grad” para llevar a cabo esta tarea, ahorrando mediante esta instrucción memoria y aumentando la velocidad de cálculo. Una vez hecho esto, se hace el procesado de los resultados devueltos por la red, y se actualizan los flags necesarios.

El último método de esta lista sería “addSegmentFromNumpyArray”, en la cual lo que se hace es crear un “objeto” segmento vacío, al cual se le asigna un nombre y un color, los cuales han sido previamente pasado por parámetros. Seguidamente, dicho segmento vacío se añade al nodo de segmento, modificando a su vez la visibilidad del mismo. Una vez hecho todo esto, se utiliza una de las herramientas administradas por la propia librería ‘slicer’ para añadir la segmentación en la escena, siendo este propósito el objetivo principal del método.

4.5.5 Test Class & Other Class

En este caso, lo que hemos asignado como Test Class no se ha tocado, es decir, se ha dejado por defecto. Son embargo, el bloque al que se le ha llamado Other Class se ha aprovechado para definir las clases de los modelos a utilizar. Ambas ya han sido comentadas previamente en este mismo documento, es por ello que no se comentarán, pero si se adjuntará el código correspondiente (anexo 10.3, Código 10.27 y Código 10.28) que muestra cómo se han definido dentro del código. Estas se definen fuera para que puedan ser accesibles por cada una de las otras clases, siendo en este caso la “Logic Class” la que en su mayoría hace uso de estas.

4.5.6 Diagrama de flujo del módulo *Fracture Classification*

Con el afán de proporcionar un mejor entendimiento del funcionamiento y la estructura del módulo, se muestra a continuación el diagrama de flujo que lo caracteriza:

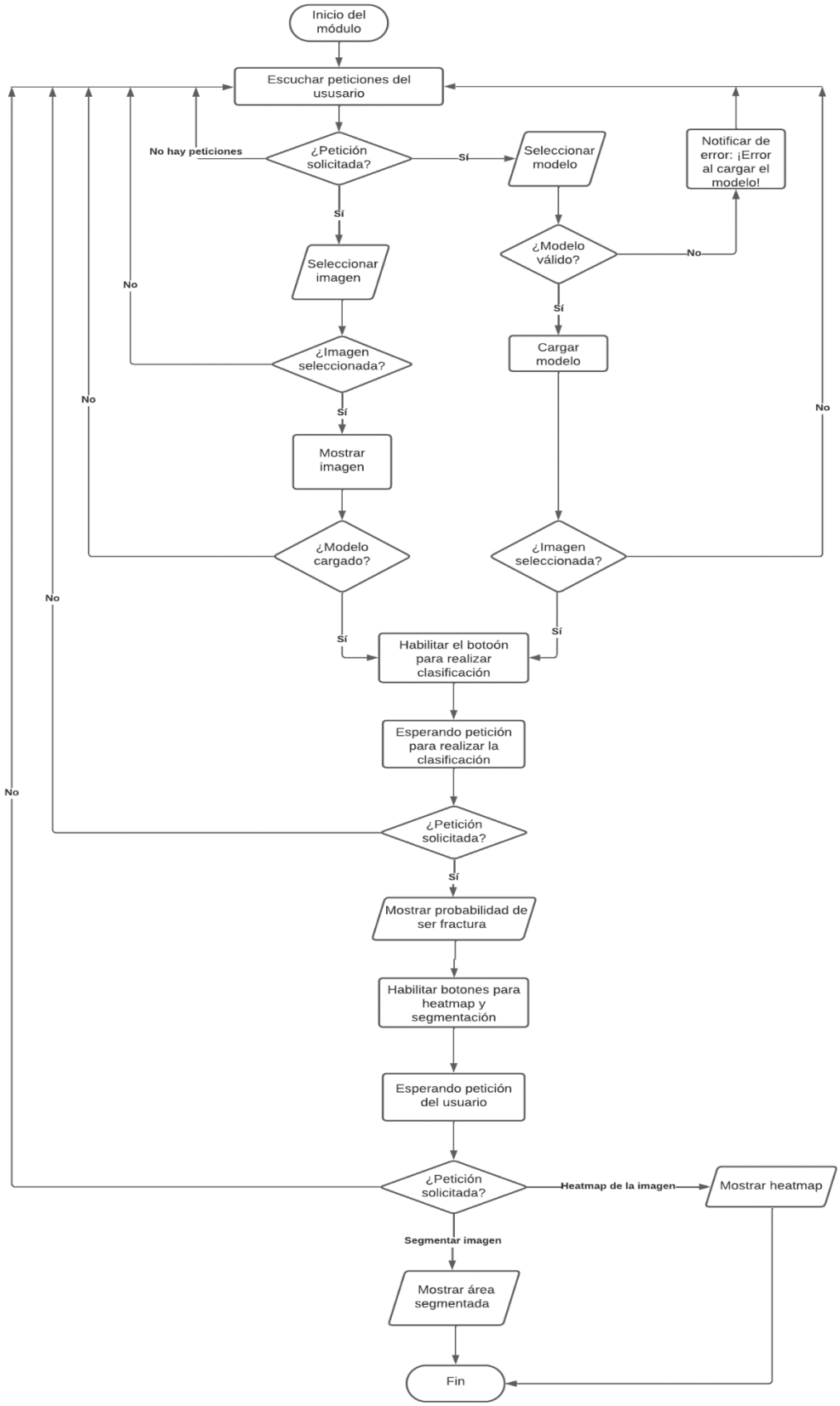


Fig. 4.31: Diagrama de flujo del módulo Fracture Classification.

4.6 ENSAYO Y PUESTA EN ACCIÓN

Para el ensayo se ha seguido el esquema aportado. En dicho esquema se incorpora el seguimiento del posicionamiento de la aguja, y el posicionamiento del modelo en cuestión, que en este caso no hará falta.

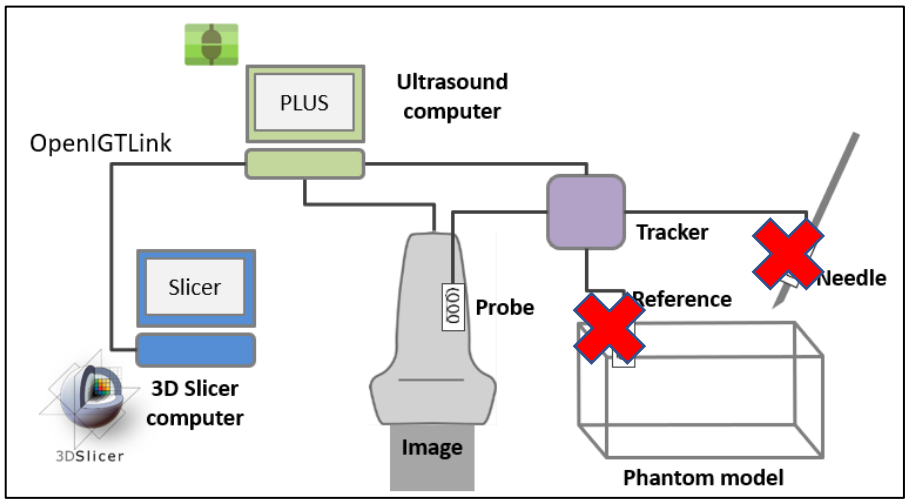


Fig. 4.32: Esquema extraido del PerkLab Bootcamp, en el que se detalla el montaje del sistema de ensayo.

El ensayo se ha dividido como tal en dos etapas. La primera etapa consiste en asegurar el correcto funcionamiento del módulo; para ello se harán inferencias en dicho modulo tomando imágenes a tiempo real del fantoma. La idea en esta primera etapa, es clasificar 15 imágenes de las fracturas encontradas en el fantoma, y otras 15 de las partes sanas del mismo. Los resultados obtenidos por el modelo se apuntarán en un archivo TXT para su posterior análisis estadístico. Después de este, se utilizará un fantoma comercial que contiene costillas que se reconstruirán utilizando el módulo de *Volume Reconstruction* del paquete de módulos de IGT.

Durante el proceso el proceso la posición de la sonda variará según lo que se esté haciendo, es decir, para realizar la clasificación el haz de la sonda debe ser paralelo al hueso bajo estudio, sin embargo, en el *Volume Reconstruction* este debe ser perpendicular. A continuación, en la parte inferior se exponen dos imágenes extraídas de un video que se ha grabado realizando estos dos procedimientos.

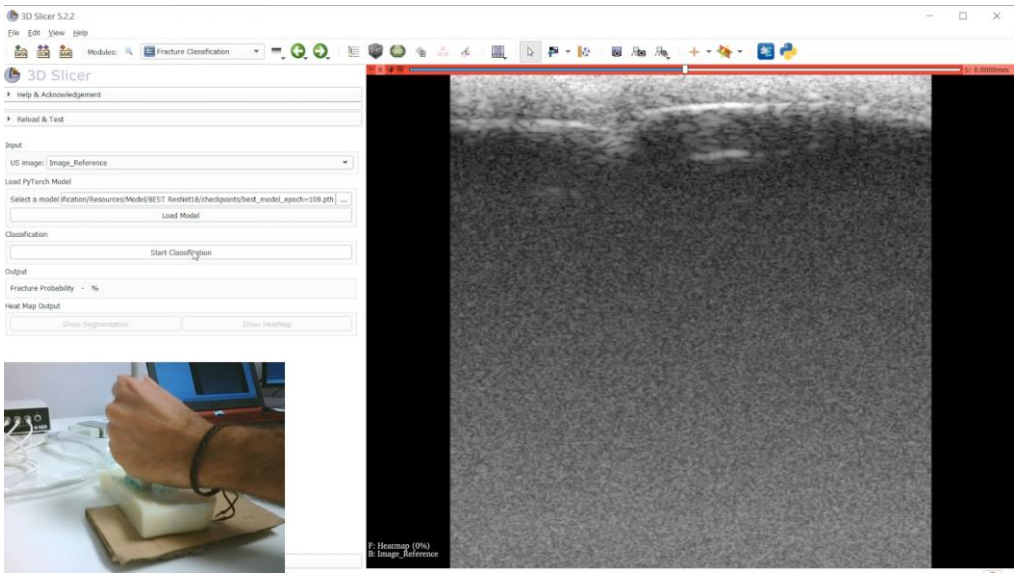


Fig. 4.33: Imagen correspondiente a la etapa destinada a la evaluación de la clasificación y localización de la fractura.

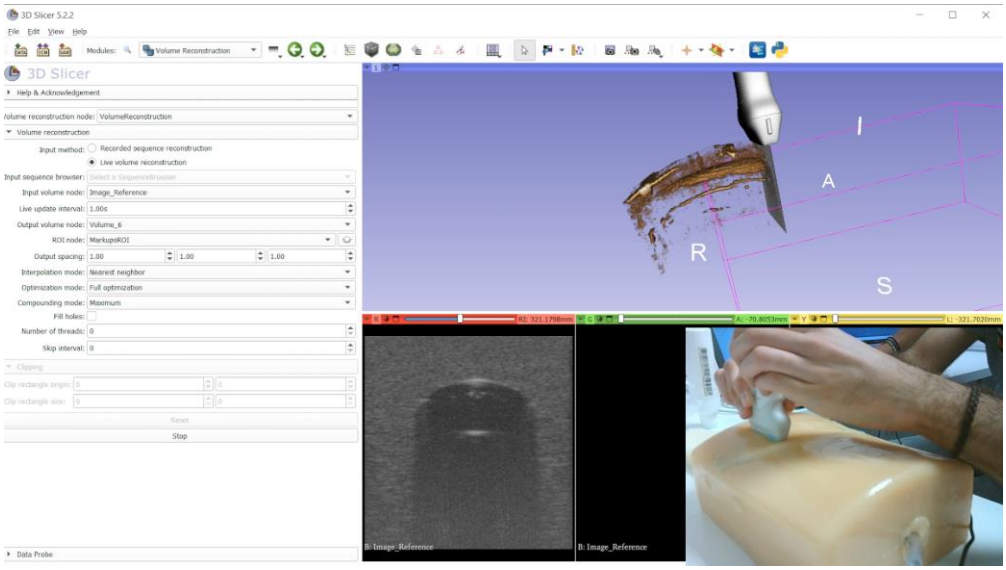


Fig. 4.34: Imagen correspondiente a la etapa destinada a la evaluación de resultados por parte del módulo Volume Reconstruction.

5 RESULTADOS Y DISCUSIÓN

Este apartado estará divido en tres apartados principales. El primero de ellos, el apartado 5.1 tiene como misión dar una valoración objetiva de las imágenes que conforman la base de datos. El segundo apartado (apartado 5.2) pondrá en valor los resultados obtenidos durante el entrenamiento, mostrando para ello gráficos que muestren la función de pérdida y la función de exactitud a medida que avanzan las épocas de entramiento. Por último, en el apartado 5.3, se valorarán los resultados de los modelos entrenados mediante las dos técnicas ya comentadas, utilizando para ello las métricas estadísticas especificadas en el apartado 4.4.9.

5.1 DISCUSIÓN Y VALORACIÓN DE LAS BASES DE DATOS

Dada cuenta de que disponemos de dos tipos de base de datos (vistas en el capítulo 4, apartado 4.4), se cree oportuno analizar cada una de estas por separado para así profundizar mejor en las características de cada una de estas.

5.1.1 Base de datos de imagen ecográfica simulada

Un ejemplo del resultado obtenido de cara a realizar el proceso de simulación de imagen ecográfica es el que a continuación se muestra en la Fig. 5.1.

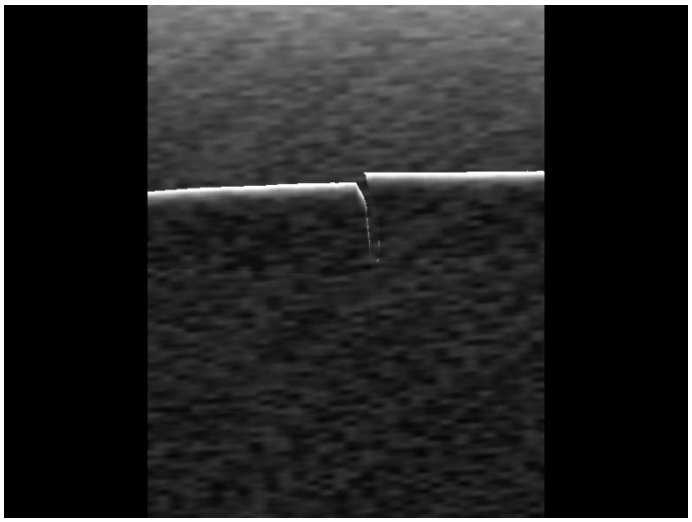


Fig. 5.1: Fractura ósea obtenida mediante imagen simulada en 3D Slicer.

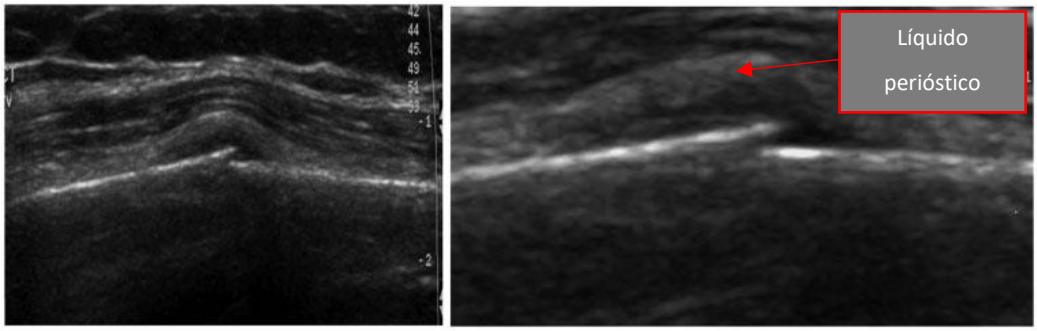


Fig. 5.2: Imagen de fractura diafisaria en la que se aprecia el líquido perióstico. Esta imagen ha sido obtenida mediante PoCUS[3] [36].

Para poder hacer una buena valoración del resultado obtenido empleando la técnica de imagen simulada, se ha añadido una imagen de una fractura real obtenida mediante imagen ecográfica (Fig. 5.2), con el fin de tener una referencia visual de cómo sería una fractura real con imagen ecográfica.

Primeramente, tal y como se puede observar en la imagen simulada tenemos una definición de la corteza ósea que en el caso real es casi inexistente. Esto se debe a que el tipo de ruido que se encuentra en la imagen real no solo afecta al medio en el que se propagan las ondas sonoras, sino que también afecta a las interfaces en las que se reflejan, es decir, los tejidos o la corteza ósea. Es por ello que podría ser oportuno en el caso de la obtención de imagen ecográfica simulada, a modo de conseguir una imagen mucho más real, poder dotar de ruido a la parte de la imagen relativa a la corteza ósea de este. La forma de conseguir esto se obtiene mediante la modificación de los parámetros especificados en el XML ya visto en el apartado 4.1, donde se habla con más detalle de la lógica que hay tras el sistema empleado para la simulación de las imágenes de fracturas.

Otra de las diferencias aparentemente más drásticas en el caso de la imagen simulada, es la ausencia de tejido alrededor de la fractura. La presencia de estos tipos de tejidos en las imágenes de entrenamiento resultaría de vital importancia para que la red supiera distinguir en qué casos se está observando la corteza ósea, y en qué casos se trata de vasos o tejidos musculares.

[3] PoCUS (Point-of-Care Ultrasound): Se acuña este término a aquellos sistemas ecográficos portátiles que pueden trasladarse fácilmente a los puntos de asistencia. Entre los diagnósticos que se suelen realizar con este tipo de tecnología destacan la detección de fracturas.

Una característica importante a la hora de detectar fracturas en PoCUS es la presencia del líquido perióstico en la zona afectada. El líquido perióstico se trata de un que se encuentra entre el periostio y la capa subyacente del hueso. El periostio es básicamente una capa que rodea el hueso, y es la encargada del crecimiento, reparación y nutrición de este; es por eso que cuando se produce una fractura, el periostio se rompe, provocando la segregación de este líquido que se encuentra bajo su capa. Es por ello que resultaría de interés poder replicarlo para que así, la red sujeta al entrenamiento bajo este tipo de imágenes localice estas anomalías.

5.1.2 Base de datos de fantoma de hueso fracturado

En el caso de las imágenes de esta base de datos, vemos como los resultados que se obtienen son en gran medida semejantes al caso real; sobre todo en cuanto a ruido se refiere. Además, se puede ver que, a diferencia del caso anterior, encontramos una definición de la corteza ósea más realista. Sin embargo, una de las grandes pegas es que nuevamente no tenemos la presencia de tejidos superiores a la superficie del hueso. Debido a esto último, tampoco se tiene la oportunidad de ver cómo reaccionaría la red ante la presencia de líquido perióstico debido a la fractura.

Cabe comentar, que otro de los puntos débiles encontrados a la hora de tomar imágenes del fantoma, es que, debido al poco grosor del tejido blando del mismo, es prácticamente imposible obtener una imagen de fractura centrada correctamente en el eje vertical, como si veíamos en el caso de las fracturas mediante imagen ecográfica simulada.

5.2 DISCUSIÓN DE LOS RESULTADOS OBTENIDOS EN EL ENTRENAMIENTO

En este apartado, se utilizarán los parámetros estadísticos ya definidos en el apartado 4.4.9 para poder realizar una evaluación de los modelos precisa.

5.2.1 Entrenamiento de la red con imagen ecográfica simulada

En este primer caso, los resultados de validación obtenidos del modelo entrenado son los que se muestran en la Tabla 5.1: Resultados de red con imagen ecográfica simulada, correspondientes a la época 28 utilizando la base de datos de validación. 5.1.

Tabla 5.1: Resultados de red con imagen ecográfica simulada, correspondientes a la época 28 utilizando la base de datos de validación.

RESULTADOS CONJUNTO DE DATOS DE VALIDACIÓN	
Exactitud	80%
Precisión	71,42%
Sensibilidad	100%
Valor F1	83,32%
Matriz de Confusión	$\begin{pmatrix} 15 & 6 \\ 0 & 9 \end{pmatrix}$

Los resultados obtenidos de la red neuronal entrenada con imágenes de fracturas tomadas mediante imagen ecográfica simulada, tal y como se puede apreciar en la tabla anterior, son bastante prometedores.

Primeramente, tenemos una exactitud del 80%. Esto indica que el modelo ha clasificado correctamente la mayoría de las imágenes, tanto de fracturas como de no fracturas, en el conjunto de validación.

En segundo lugar, disponemos de la precisión, la cual ha sido de un 71,42%. Esto indica que el modelo ha clasificado incorrectamente 6 imágenes como fractura cuando en realidad no lo eran; siendo este resultado bastante mejorable. Al disponer de una base de datos con las clases equilibradas, tener una precisión del 71,42%, hace que el resultado sea valorado como bueno para llevar a cabo este tipo de clasificación, puesto que, tal y como se volverá a mencionar más adelante, en un sistema de detección dentro del ámbito médico es preferible minimizar los falsos negativos frente a los falsos positivos, y en este caso se ha conseguido con éxito.

Se ha obtenido la sensibilidad, ya que este es capaz de indicar la proporción de casos positivos que el modelo ha identificado correctamente en relación con el total de casos positivos presentes en el conjunto de datos. En este caso, se ha obtenido una sensibilidad del 100% viene a significar que el modelo ha identificado correctamente el 100% de las instancias de la clase positiva (en este caso, las fracturas). Una sensibilidad tan alta es algo muy favorable tratándose de un caso de clasificación binaria, ya que muestra que el modelo tiene una buena capacidad para detectar los casos positivos. Sin embargo, también es

importante tener en cuenta el contexto clínico y las implicaciones de los casos de falsos negativos (casos de fracturas que el modelo no ha identificado). Se valora tener minimizar al máximo este último parámetro, ya que en este ámbito muchas veces es preferible diagnosticar una patología con un falso positivo que con un falso negativo. Para observar esto, se procede a analizar la matriz de confusión.

Para terminar, se ha obtenido un valor F1 dado que este parámetro estadístico es capaz de combinar la precisión y la sensibilidad en un solo valor que refleja tanto la capacidad del modelo para clasificar correctamente ejemplos positivos como su capacidad para evitar falsos negativos. En este caso un valor F1 del 83,32% sugiere que el modelo tiene un buen equilibrio entre la capacidad de clasificar correctamente tanto los ejemplos positivos como los negativos; teniendo sobre todo muy en cuenta tanto la capacidad de clasificar correctamente los ejemplos positivos como la capacidad de evitar falsos negativos.

Tal y como se ha podido observar en la Tabla 5.1, la matriz de confusión tiene un tamaño de 2x2 y muestra la cantidad de muestras que han sido clasificadas en cada una de las cuatro categorías posibles: 15 verdaderos positivos, 9 verdaderos negativos, 6 falsos positivos y 0 falsos negativos. Los resultados obtenidos son muy favorables, ya que como se explicó antes, si fijamos la vista en los falsos positivos y falsos negativos, tenemos seis falsos positivos (hueso sano diagnosticado como fractura) y ningún falso negativo, algo que, como bien se ha ido comentando, es lo que se busca en este tipo de casos. Es decir, se ha cumplido con la premisa de minimizar lo más posible los falsos negativos.

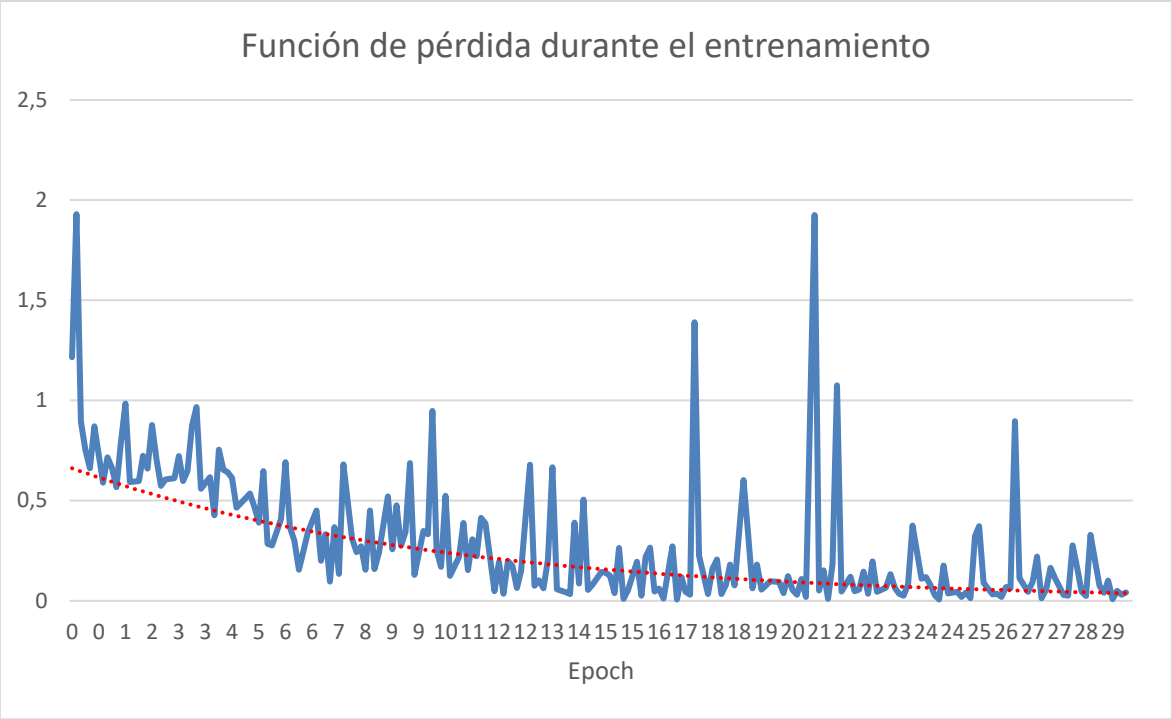


Fig. 5.3: Función de pérdida durante el entrenamiento con imagen ecográfica simulada.

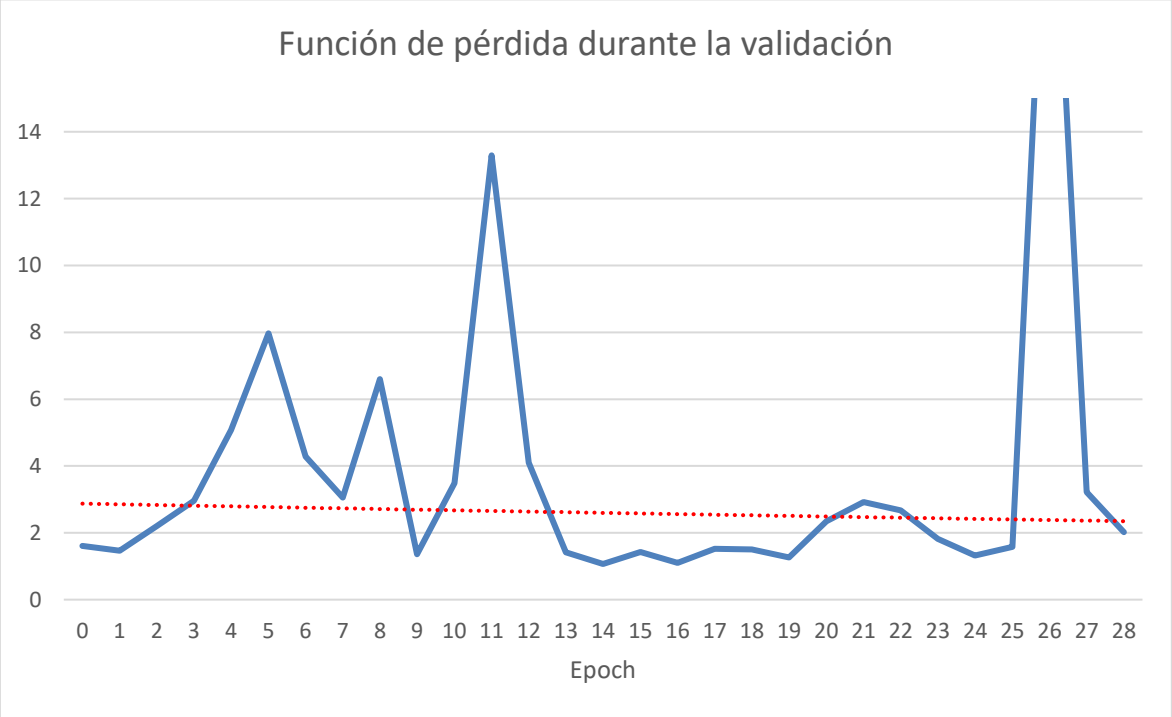


Fig. 5.4: Función de pérdida durante la validación con imagen ecográfica simulada.

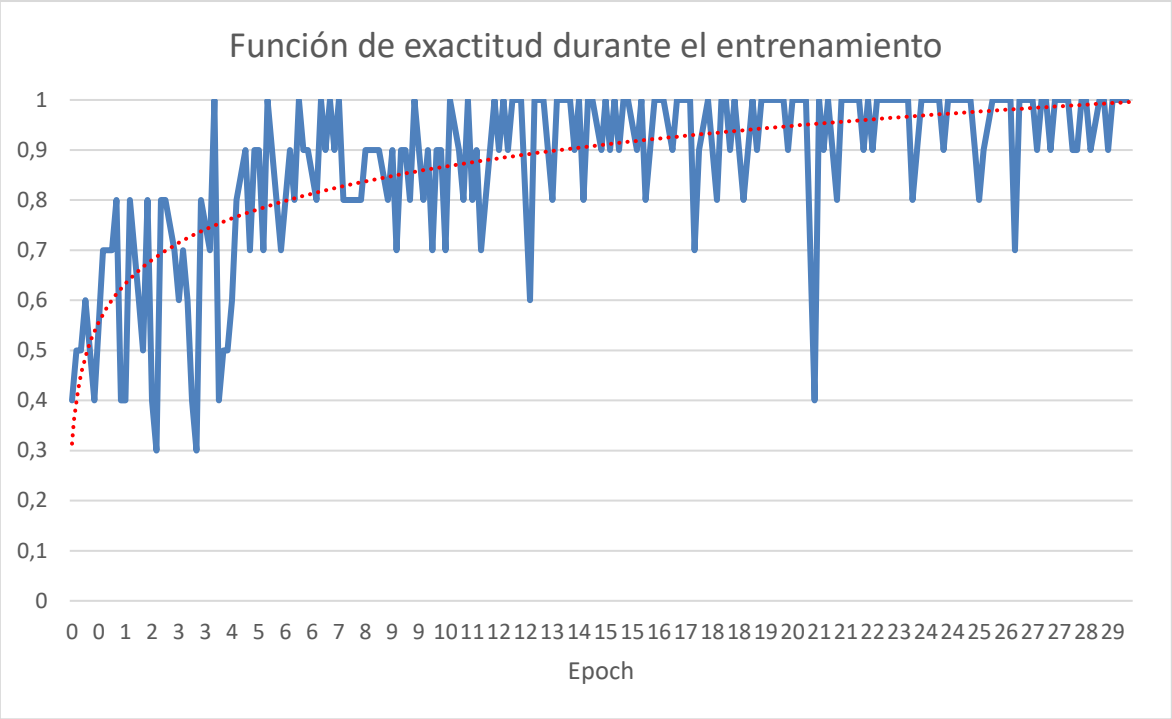


Fig. 5.5: Función de exactitud durante el entrenamiento con imagen ecográfica simulada.

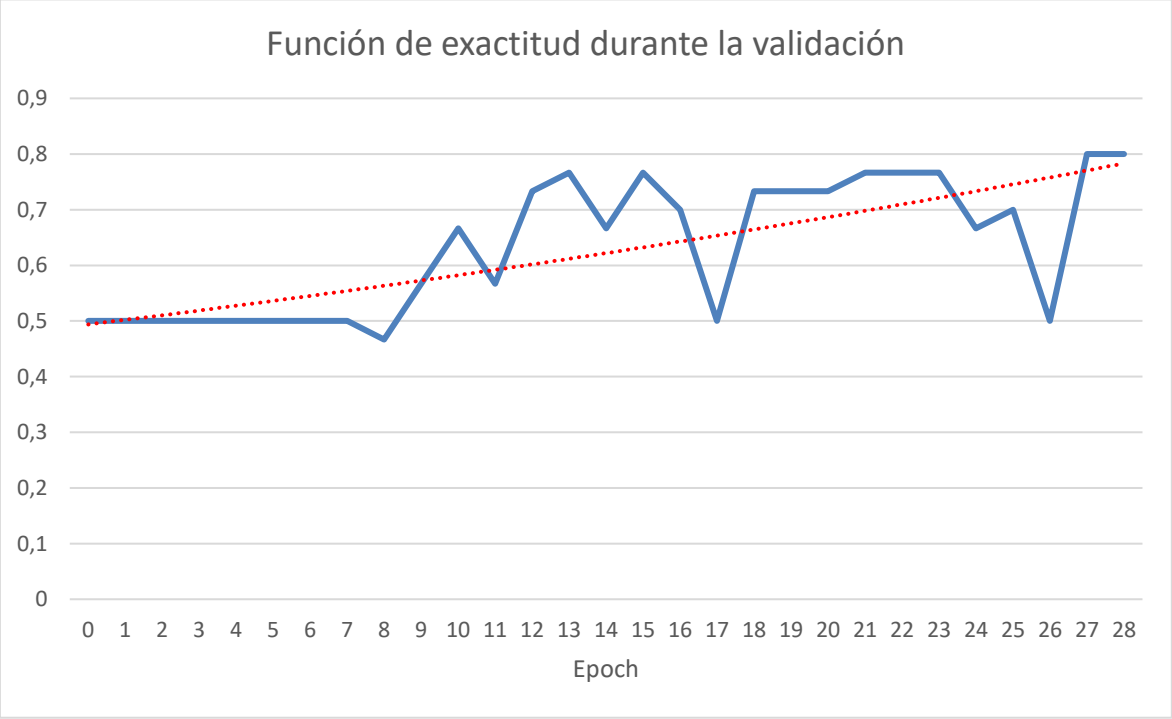


Fig. 5.6: Función de exactitud durante la validación con imagen ecográfica simulada.

A parte de los dos gráficos ya mostrados, se han extraído otra serie de parámetros estadísticos que ayudarán a terminar de definir la calidad de este sistema a la hora de diferenciar entre un hueso fracturado, y uno sano. Los parámetros insinuados son la curva ROC y el AUC, para entender mejor la información que aportan estos estadísticos véase el apartado 4.4.9.

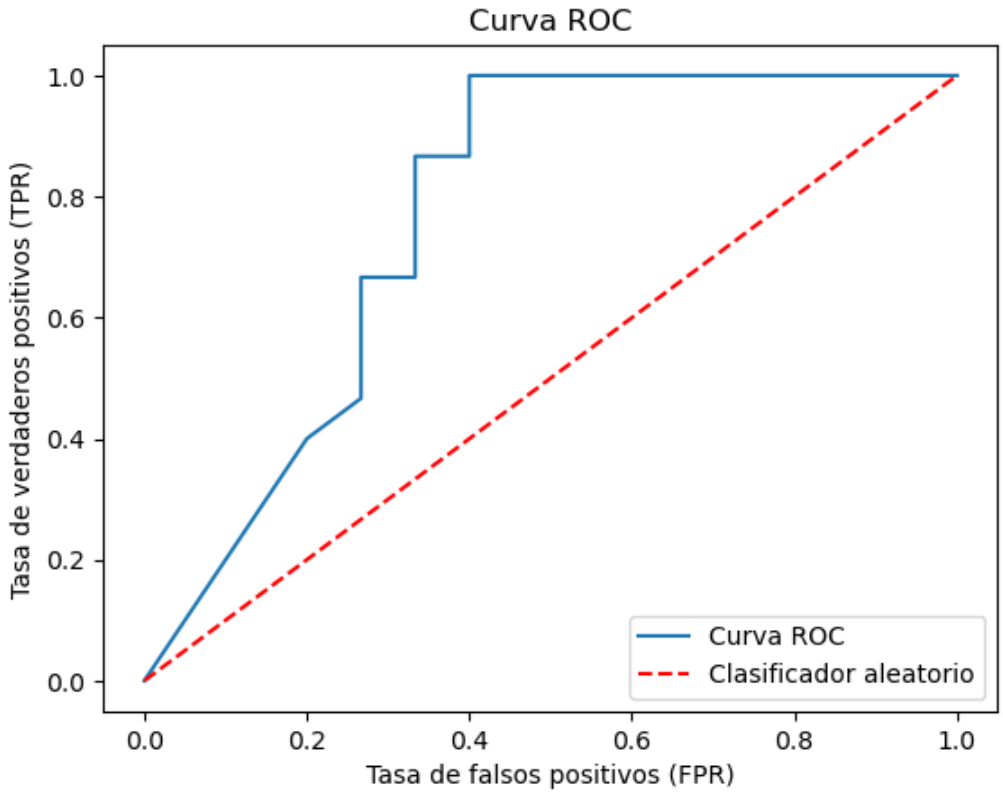


Fig. 5.7: Curva ROC extraída del análisis de los datos de validación.

Tabla 5.2: Valor del AUC extraída del análisis de los datos de validación, con imagen ecográfica simulada.

PARÁMETROS EXTRAIDOS DEL CÁLCULO DE LA CURVA ROC	
AUC	0,77

Con estos parámetros podemos definir de manera precisa si el modelo es bueno o no. En este caso, viendo la curva ROC y el AUC podemos deducir que el modelo es capaz de diferenciar con de forma excelente los huesos fracturados de los no fracturados. Un valor del 0,77 de AUC es un resultado que viene a indicar que el modelo tiene una capacidad de

clasificación superior al azar. Esto significa que el modelo es capaz de tomar decisiones más precisas que una clasificación al azar en la tarea de separar las dos clases objetivo.

Ya vistos los parámetros estadísticos del anterior párrafo, casi se pasa por alto la Fig. 5.3 y Fig. 5.4, que representa como el modelo se ha ido adaptando a los datos de entrenamiento a medida que ha transcurrido el mismo. Es evidente que uno de los mayores problemas que asechan tras todo entrenamiento de una red neuronal es el sobreajuste (*overfitting*), es decir, cuando se da el caso de que el modelo ha memorizado los datos de entrenamiento en lugar de aprender patrones generales. En este caso, el modelo puede tener dificultades para generalizar correctamente a nuevos datos. La selección de los pesos óptimos del modelo se basa en los pesos correspondientes al modelo que alcanzó la mayor exactitud en el periodo de validación; en este caso los pesos óptimos están sacados de la época (*epoch*) 28.

Todos los datos expuestos hasta ahora corresponden a datos extraídos del entrenamiento, o del propio conjunto de validación. Es por ello por lo que se ha procedido a evaluar al modelo con datos completamente nuevos, de manera que se pretende evaluar estadísticamente el comportamiento del mismo ante una situación completamente nueva, y así comprobar si el entrenamiento realizado hasta el momento ha sido bueno o no.

Para llevar a cabo esto se ha hecho una inferencia empleando una base de datos de imágenes ecográficas simuladas de evaluación, conformada por un total de 30 imágenes (15 fracturas y 15 huesos sanos). Los porcentajes obtenidos en la clasificación han sido los siguientes.

Tabla 5.3: Resultados obtenidos en la evaluación del modelo de imagen ecográfica simulada.

Probabilidad (%) de ser una fractura	
Fracturas	No fracturas
100,00	0,01
99,99	0,01
99,93	0,02
99,89	0,02
99,97	0,02
59,41	0,02
86,59	0,01
97,98	0,01
55,65	0,01
85,37	0,01
33,46	0,01
0,12	0,02
100,00	00,02
100,00	0,01
99,97	0,10

Si se hace el mismo análisis extrayendo los parámetros estadísticos ya vistos, se obtienen los resultados mostrados en la Tabla 5.4.

Tabla 5.4: Resultados de red con imagen ecográfica simulada, correspondientes a la época 28 utilizando la base de datos de evaluación.

RESULTADOS CONJUNTO DE DATOS DE EVALUACIÓN	
Exactitud	93,33%
Precisión	100%
Sensibilidad	86,67%
Valor F1	92,85%
Matriz de Confusión	$\begin{pmatrix} 13 & 0 \\ 2 & 15 \end{pmatrix}$

Tal y como se aprecia en la tabla anteriormente mostrada, los resultados en la evaluación han sido mucho mejores a los vistos en la validación del modelo. Este incremento en los valores de los parámetros estadísticos se debe a que el conjunto de imágenes de validación está compuesto por imágenes de fracturas complejas para el análisis de la red neuronal, es por ello por lo que los resultados obtenidos en la evaluación del modelo son mejores,

puesto que la complejidad de estas imágenes es menor. Haciendo una puntualización de estos nuevos datos, se puede ver que aquí no se cumple la premisa en la que los falsos positivos son mayores a los falsos negativos. Esto es quizás el punto débil de estos parámetros estadísticamente hablando, no obstante, esto se compensa con la curva ROC y AUC perfecta que a continuación se muestra.

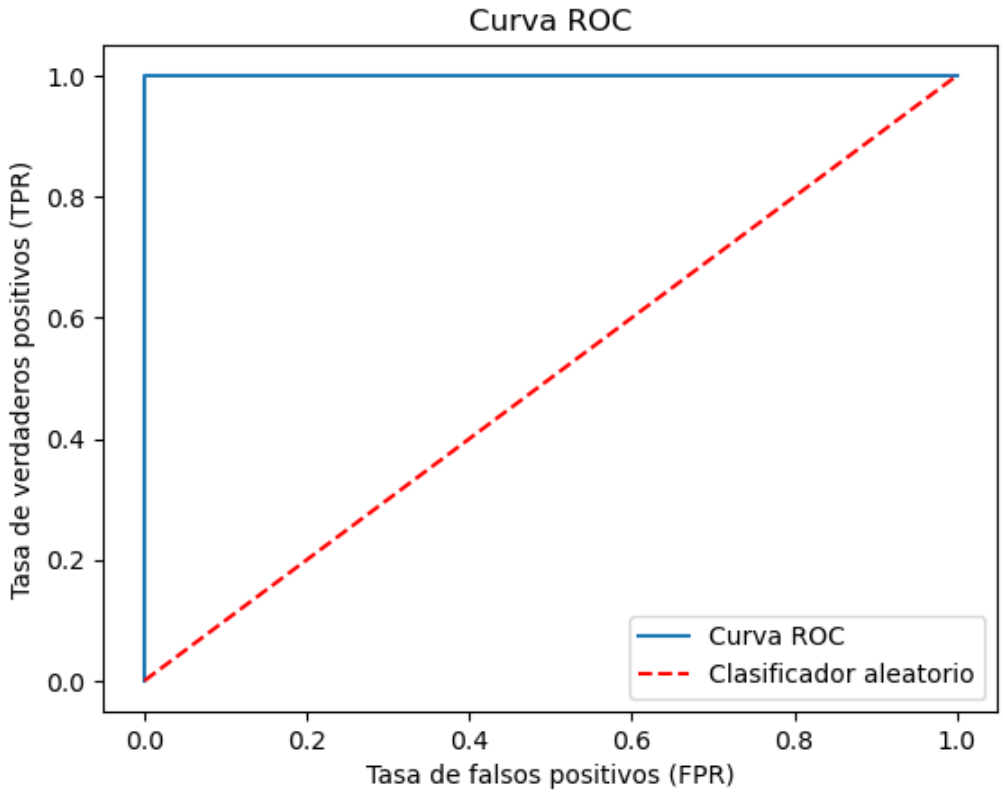


Fig. 5.8: Curva ROC extraída del análisis de los datos de evaluación.

Tabla 5.5: Valor del AUC extraída del análisis de los datos de evaluación, con imagen ecográfica simulada.

PARÁMETROS EXTRAIDOS DEL CÁLCULO DE LA CURVA ROC	
AUC	1,00

5.2.2 Entrenamiento de la red con imagen ecográfica obtenida del fantoma

En este segundo caso, se han obtenido los siguientes resultados haciendo inferencias con la base de datos de validación, véase la Tabla 5.6.

Tabla 5.6: Resultados de red con imágenes extraídas del fantoma mediante sonda ecográfica, correspondientes a la época 109.

RESULTADOS CONJUNTO DE DATOS DE VALIDACIÓN	
Exactitud	93,75%
Precisión	93,75%
Sensibilidad	93,75%
Valor F1	93,75%
Matriz de Confusión	$\begin{pmatrix} 15 & 1 \\ 1 & 15 \end{pmatrix}$

Nuevamente en este análisis se tiene en primer lugar la exactitud, la cual esta vez es de un 93.75%. Nos encontramos ante un modelo que ha clasificado correctamente un total de 30 imágenes, tanto de fracturas como de no fracturas, en el conjunto de validación. Dado que las clases están equilibradas (mismo número de imágenes fracturadas que de no fracturadas), esta alta exactitud sugiere que el modelo es capaz de discernir entre fracturas y no fracturas con éxito; análisis el cual es muy parecido al caso anterior.

En segundo lugar, disponemos de la precisión, la cual ha tenido el mismo resultado que la exactitud, 93.75%. Esto indica que el modelo ha clasificado incorrectamente una imagen como fractura cuando en realidad no lo era. Al disponer de una base de datos con las clases equilibradas, tener una precisión del 93.73% es un resultado favorable, aunque se puede ver como defecto el hecho de que se tiene el mismo número falsos positivos y de falsos negativos; en este caso se pretende tener menor número de falsos negativos que de falsos positivos.

En este caso, una sensibilidad del 93.73% viene a significar que el modelo ha identificado correctamente el 93.73% de las instancias de la clase positiva (en este caso, las fracturas) en relación con todas las instancias de fracturas presentes en el conjunto de validación. Tal y como se comentó anteriormente, una sensibilidad tan alta es algo muy favorable tratándose de un caso de clasificación binaria, ya que muestra que el modelo tiene una buena capacidad para detectar los casos positivos.

Tal y como se ha podido observar en Tabla 5.6, la matriz de confusión tiene un tamaño de 2x2 y muestra la cantidad de muestras que han sido clasificadas en cada una de las cuatro

categorías posibles: 15 verdaderos positivos, 15 verdaderos negativos, 1 falso positivo y 1 falsos negativos. En este caso, no se ha cumplido con la premisa de minimizar lo más posible los falsos negativos, pero, aunque no se haya cumplido con esto, no exime el hecho de que el modelo no sea bueno, simplemente nos viene a indicar que existe prácticamente las mismas probabilidades de que devuelva un falso positivo o un falso negativo.

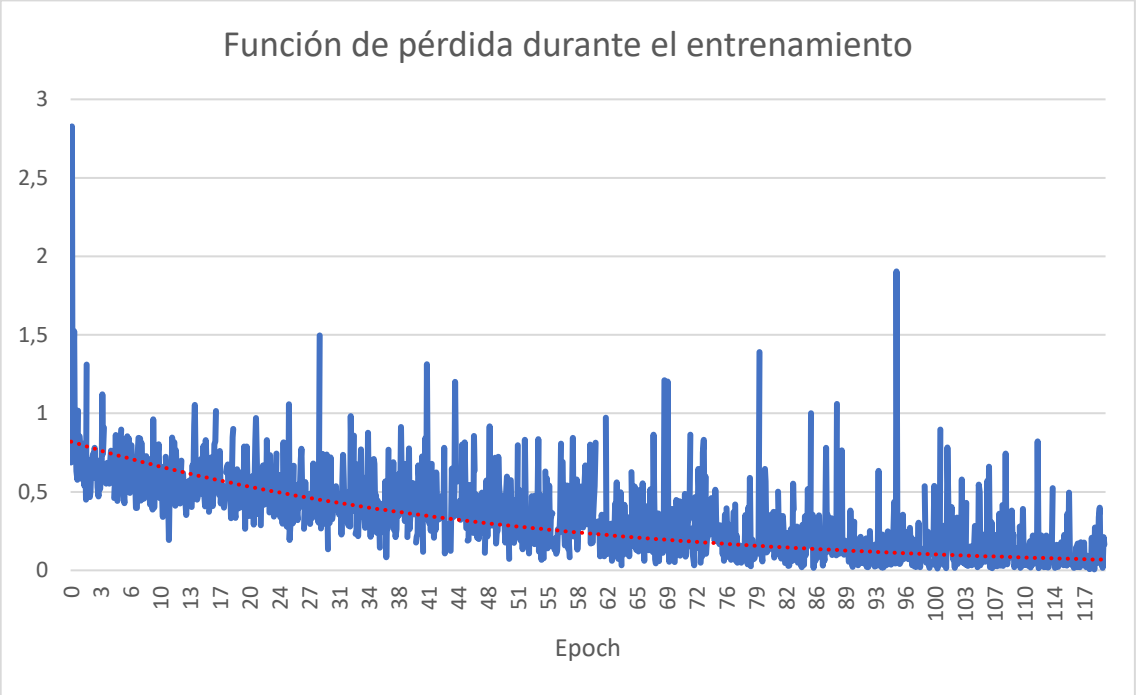


Fig. 5.9: Función de pérdida durante el entrenamiento con las imágenes del fantoma mediante sonda ecográfica.

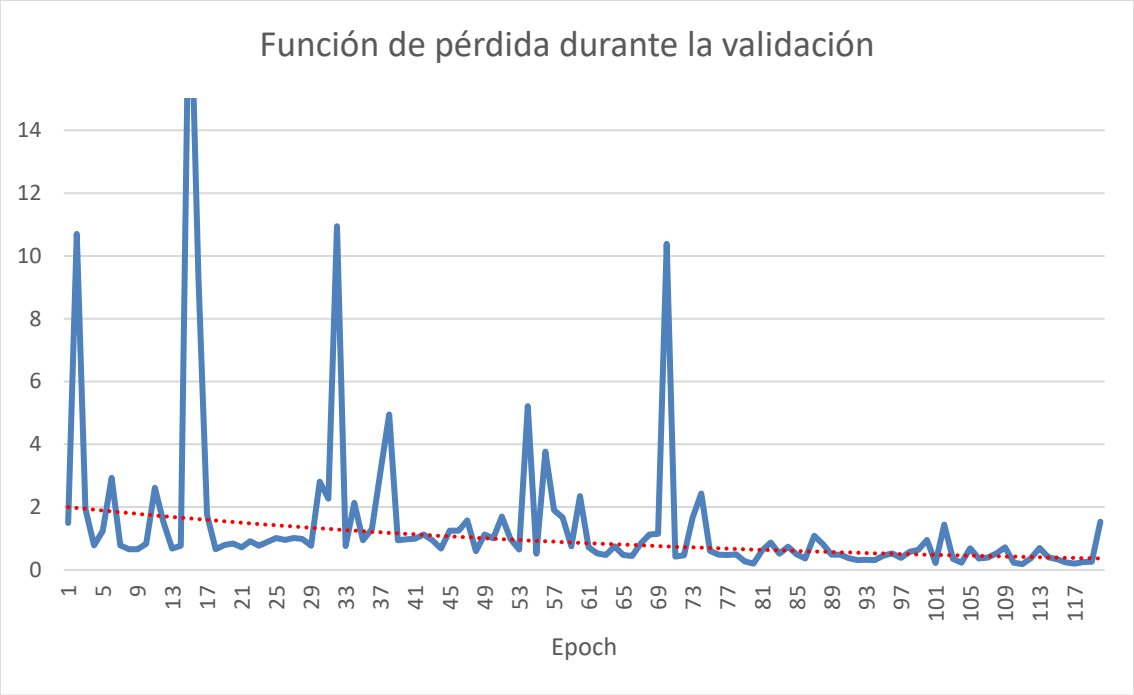


Fig. 5.10: Función de pérdida durante la validación con las imágenes del fantoma mediante sonda ecográfica.

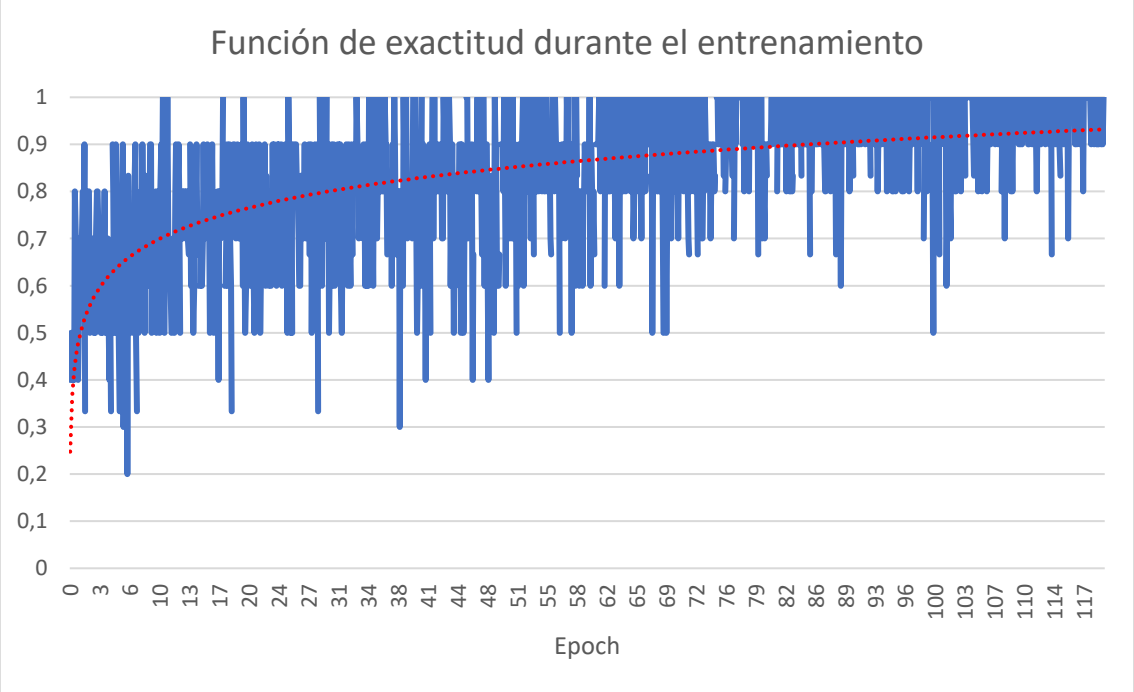


Fig. 5.11: Función de exactitud durante el entrenamiento con imagen extraída del fantoma mediante imagen ecográfica.

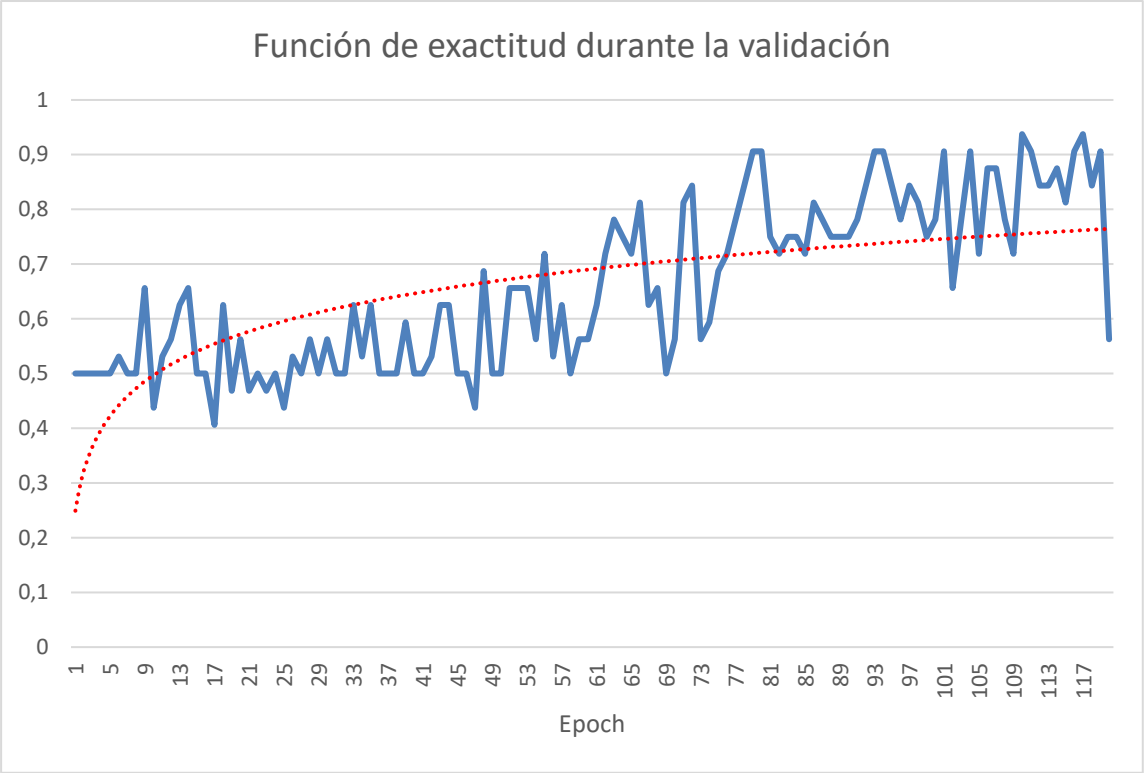


Fig. 5.12: Función de exactitud durante la validación con imagen extraída del fantoma mediante imagen ecográfica.

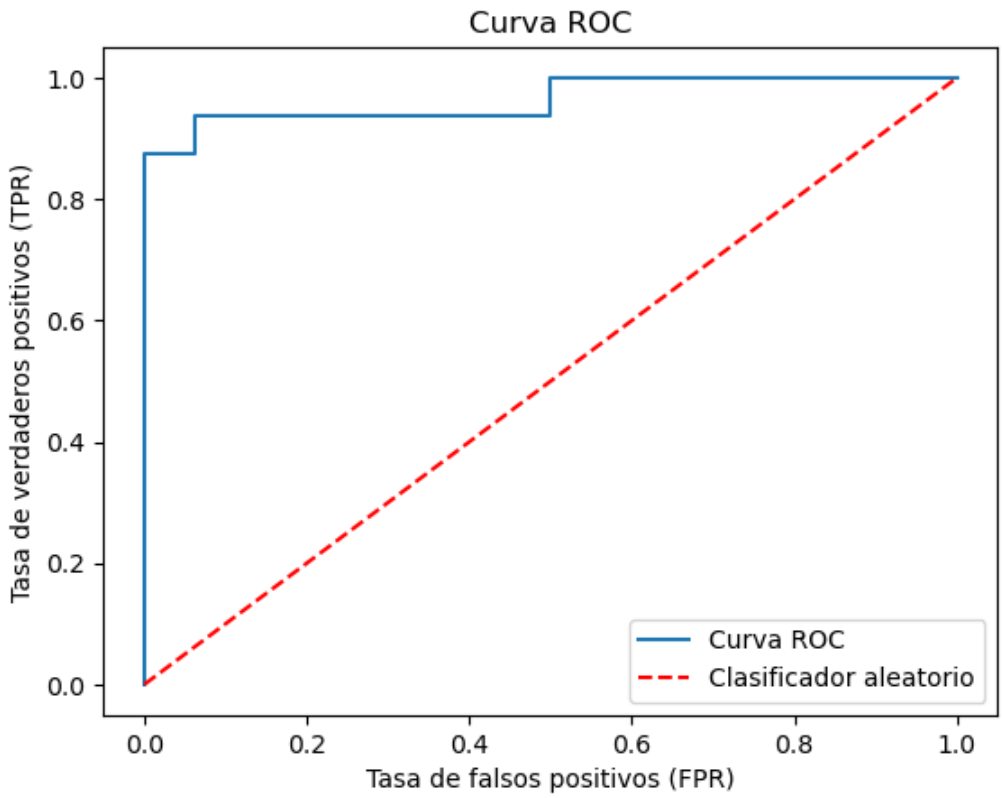


Fig. 5.13: Curva ROC extraída del análisis de los datos de validación.

Tabla 5.7: Valor del AUC extraída del análisis de los datos de validación, sin aprendizaje transferido.

PARÁMETROS EXTRAIDOS DEL CÁLCULO DE LA CURVA ROC	
AUC	0.96

Sin entrar en mucho detalle con el fin de no ser tan repetitivo con unos datos casi semejantes, es evidente que en este caso tanto el AUC como la curva ROC difieren poco del caso anterior. También es detener en cuenta que en el proceso de validación se han utilizado mayor número de imágenes que en el caso de la imagen simulada. Es de tener en cuenta este último punto, ya que a mayor volumen de datos se tienen, mejor se podrá caracterizar el sistema.

Ya, por último, un apunte que es necesario realizar en este caso, es que, debido a la complejidad de las imágenes sujetas bajo análisis, se puede apreciar que el número de épocas en este entrenamiento es sumamente mayor comparado con el caso anterior. Este entrenamiento en concreto fue realizado con un total de 120 épocas, y fue en la 109 cuando se obtuvo los mejores datos en el proceso de validación de la red.

5.2.3 Entrenamiento de la red con aprendizaje transferido de la imagen ecográfica simulada

Tabla 5.8: Resultados de red con aprendizaje transferido, correspondiente a la época 91.

RESULTADOS CONJUNTO DE DATOS DE VALIDACIÓN	
Exactitud	87,50%
Precisión	87,50%
Sensibilidad	87,50%
Valor F1	87,50%
Matriz de Confusión	$\begin{pmatrix} 14 & 2 \\ 2 & 14 \end{pmatrix}$

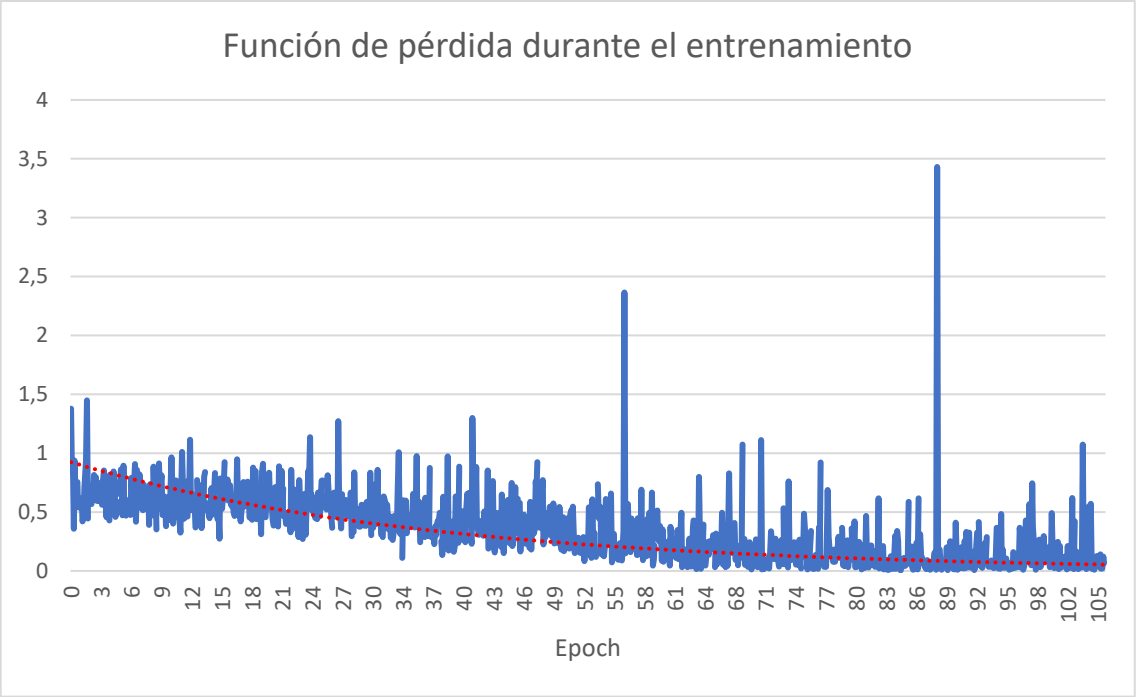


Fig. 5.14: Función de pérdida durante el entrenamiento con aprendizaje transferido.

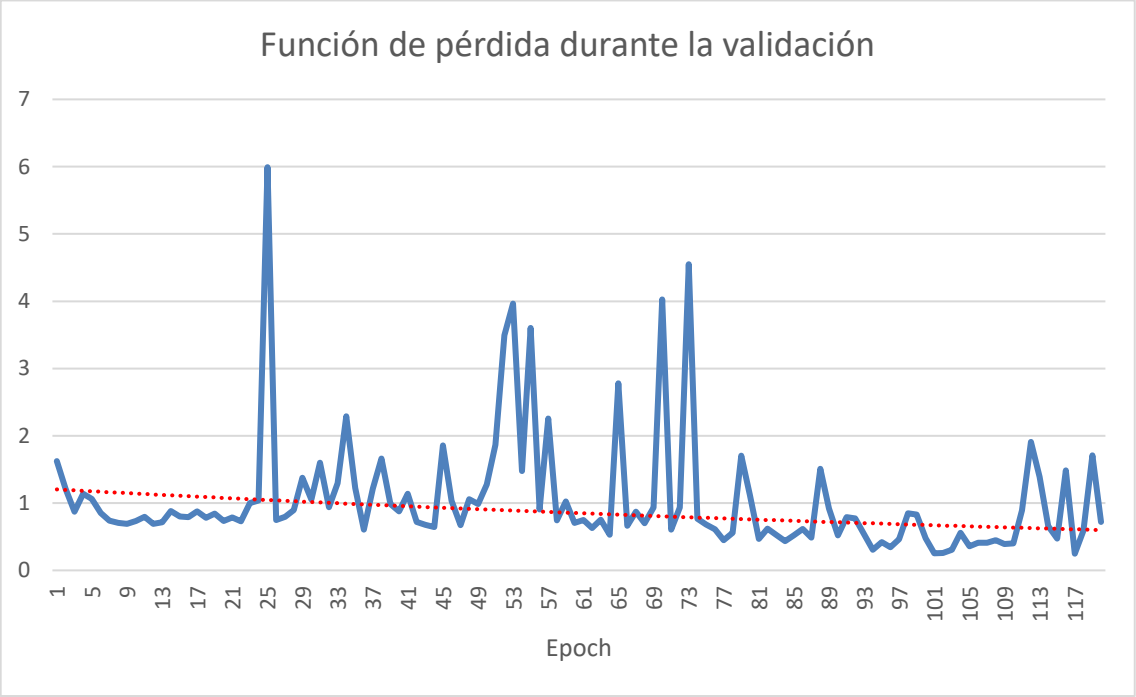


Fig. 5.15: Función de pérdida durante la validación con aprendizaje transferido.

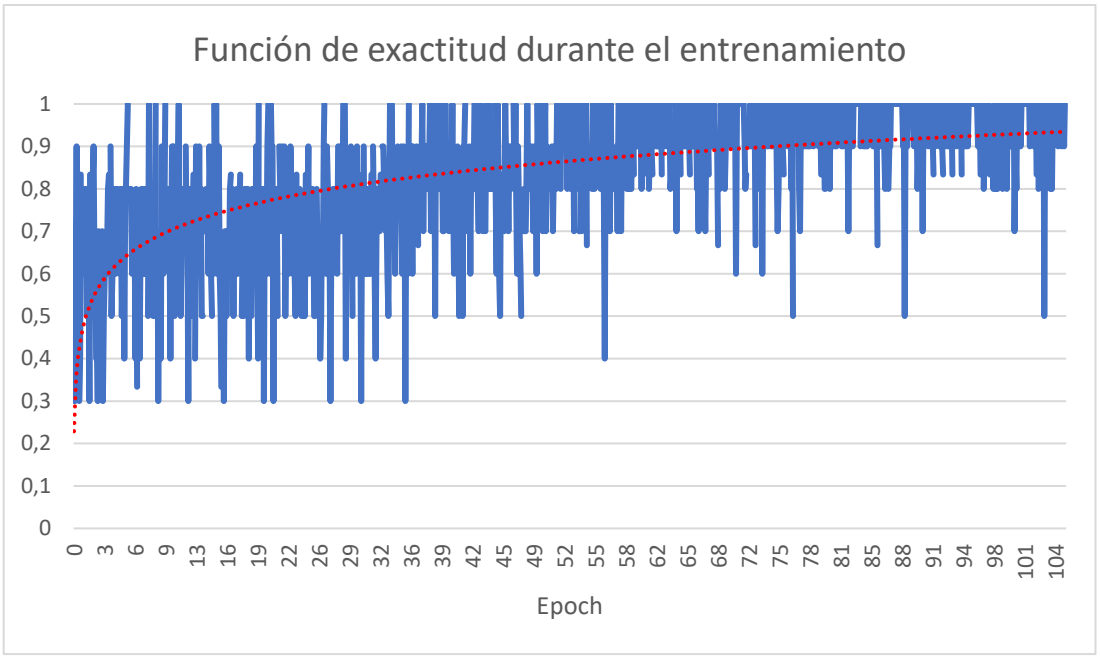


Fig. 5.16: Función de exactitud durante el entrenamiento con aprendizaje transferido.

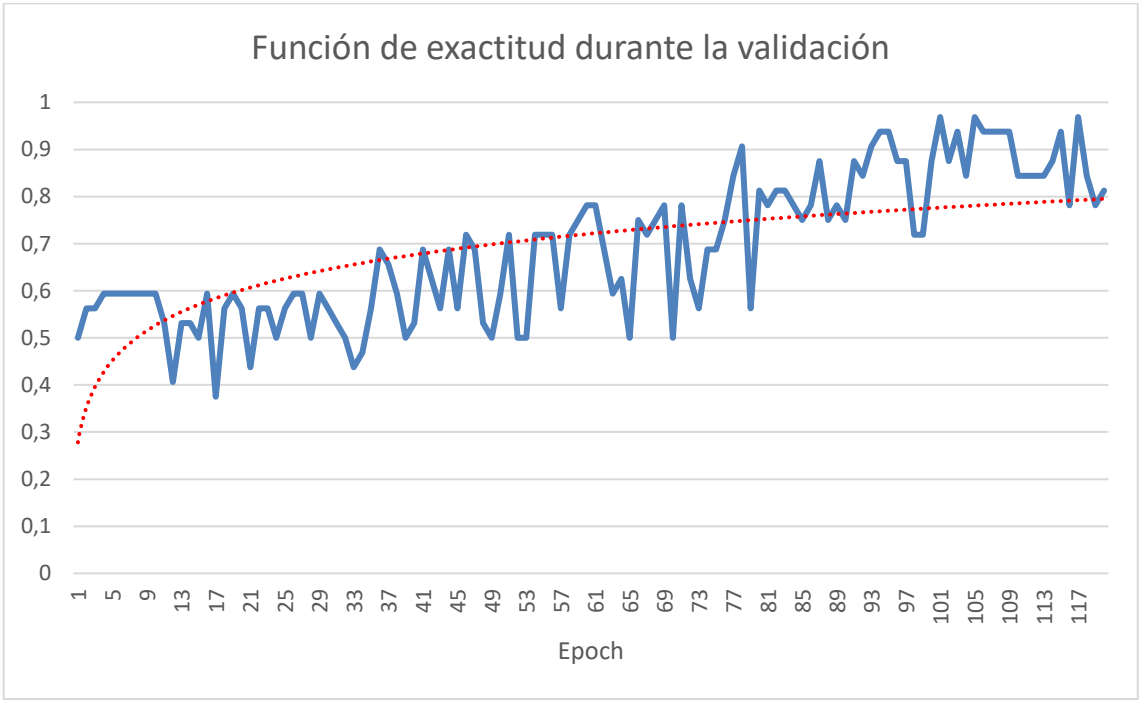


Fig. 5.17: Función de exactitud durante la validación con aprendizaje transferido.

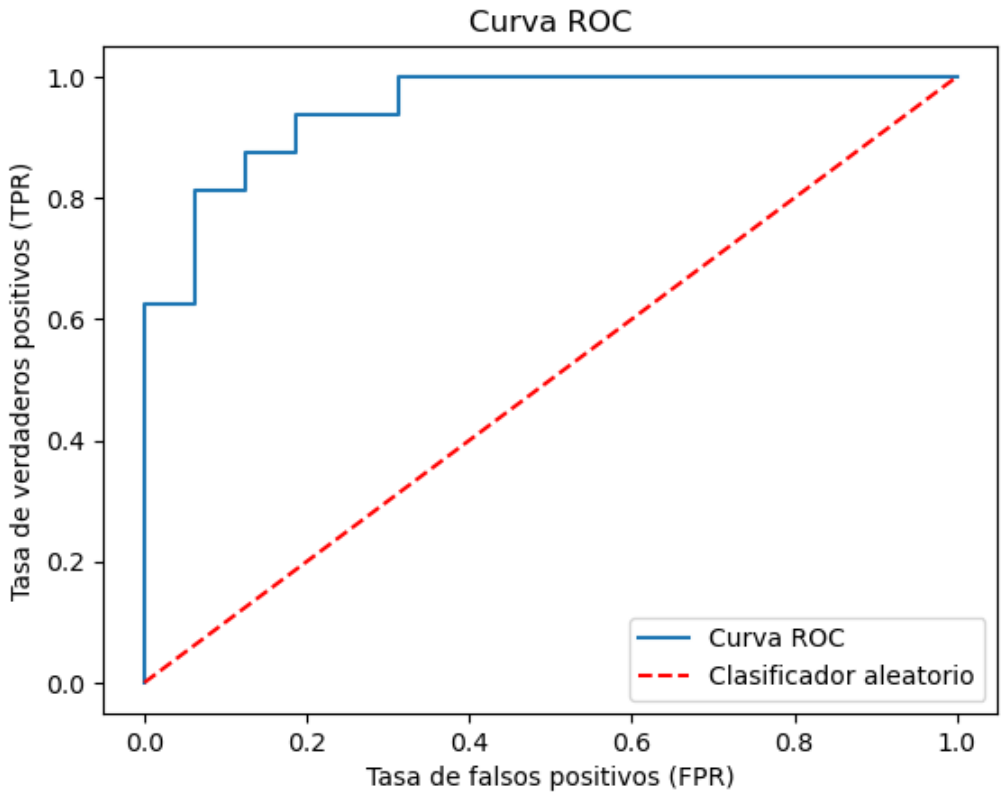


Fig. 5.18: Curva ROC (Receiver Operating Characteristic) extraída del análisis de los datos de validación.

Tabla 5.9: Valor del AUC extraída del análisis de los datos de validación, con aprendizaje transferido.

PARÁMETROS EXTRAIDOS DEL CÁLCULO DE LA CURVA ROC	
AUC	0,94

Como análisis rápido de estos últimos resultados, se puede llegar destacar que se tiene un AUC del 0,94; el cual es ligeramente menor al valor de AUC anteriormente obtenido en el apartado anterior en el que no se empleaba la técnica de aprendizaje transferido. Realmente, la diferencia se podría considerar despreciable, ya que ambos modelos desarrollan la tarea encomendada casi a la perfección. Lo destacable en este caso en el que se ha empleado esta técnica de aprendizaje transferido, es que se han conseguido resultados prácticamente iguales al caso anterior, entrenando al modelo casi 20 épocas menos, es decir, se ha logrado una eficiencia en el entrenamiento con resultados

favorables. No obstante, aun queda por comprobar estadísticamente, cuál es el comportamiento que efectúan estos dos modelos en el ensayo.

5.3 DISCUSIÓN DE LOS RESULTADOS OBTENIDOS EN EL ENSAYO

Para el siguiente análisis, se expondrán los resultados obtenidos en el ensayo, y seguidamente se pondrán los resultados de sus curvas ROC y AUC, además de la exactitud, precisión, sensibilidad, valor F1 y Matriz de confusión. Todas las inferencias que se han hecho en el modelo mediante el uso del módulo han sido a tiempo real, tal y como se ha explicado en el apartado 4.6 de este documento.

5.3.1 Datos del ensayo sin aprendizaje transferido

A continuación, se muestran los resultados referidos al ensayo con el modelo **sin aprendizaje transferido**; dichos resultados reflejan la probabilidad (%) de que la imagen inferida en el modelo sea una fractura:

Tabla 5.10: Resultados obtenidos del ensayo con imagen ecográfica del fantoma a tiempo real sin aprendizaje transferido.

Probabilidad (%) de ser una fractura	
Fracturados	No fracturados
91,14	15,84
41,08	8,44
86,87	9,25
99,46	8,08
39,05	8,94
75,18	8,33
95,20	10,68
64,75	21,55
54,13	42,82
62,05	15,77
83,16	26,37
59,82	1,63
52,85	33,58
81,00	16,50
95,73	8,72

Tabla 5.11: Resultados de la exactitud y la precisión.

RESULTADOS CONJUNTO DE DATOS DE EVALUACIÓN	
Exactitud	93,33%
Precisión	100%
Sensibilidad	86,67%
Valor F1	92,85%
Matriz de Confusión	$\begin{pmatrix} 13 & 0 \\ 2 & 15 \end{pmatrix}$

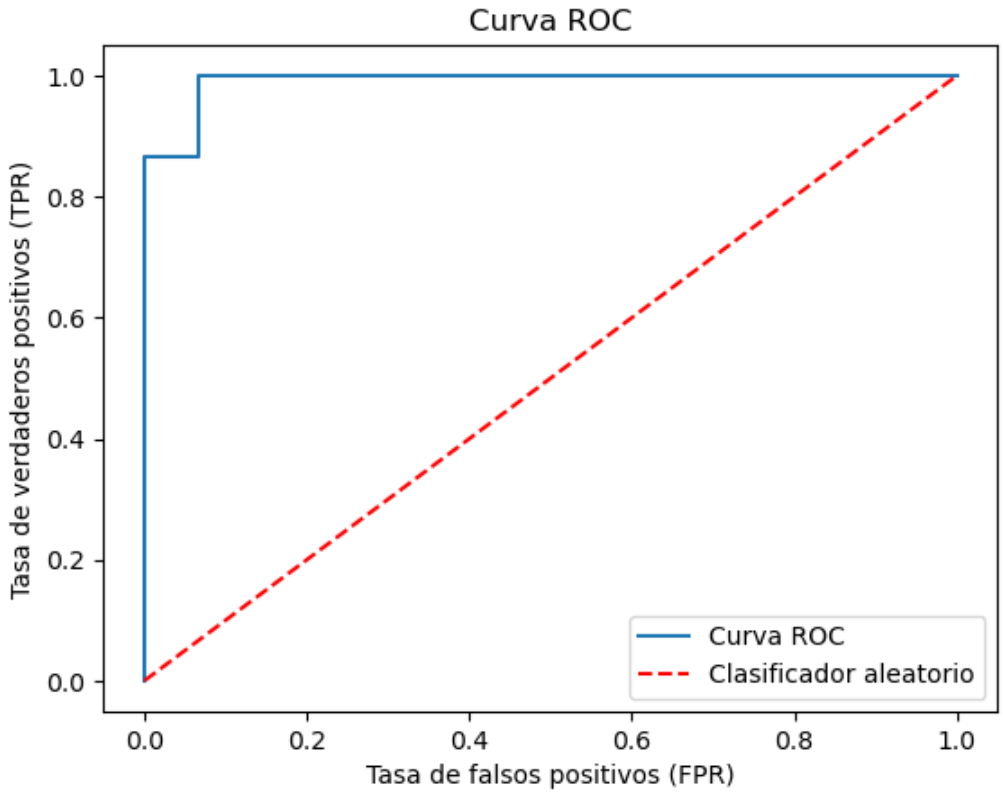


Fig. 5.19: Curva ROC del ensayo sin aprendizaje transferido.

Tabla 5.12: Valor del AUC extraída del análisis de los datos de evaluación, sin aprendizaje transferido.

PARÁMETROS EXTRAIDOS DEL CÁLCULO DE LA CURVA ROC	
AUC	0,99

Tal y como se puede observar, los resultados que ha demostrado tener la red neuronal en el ensayo han sido un fiel reflejo de lo visto anteriormente en los datos aportados en el proceso de validación. Aunque, es notorio que en este caso los datos son mucho más favorables que lo visto en el proceso de validación.

5.3.2 Datos del ensayo con aprendizaje transferido

Los datos que a continuación se muestran son los referidos al ensayo con el modelo **con aprendizaje transferido**, dichos resultados reflejan la probabilidad (%) de que la imagen inferida en el modelo sea una fractura.

Tabla 5.13: Resultados obtenidos del ensayo con imagen ecográfica del fantoma a tiempo real con aprendizaje transferido.

Probabilidad (%) de ser una fractura	
Fracturados	No fracturados
99,64	1,64
22,19	0,74
88,45	0,70
89,91	14,30
83,38	0,67
99,56	0,56
25,58	0,63
85,54	1,73
53,13	16,30
41,99	1,59
99,65	0,56
84,29	1,16
81,61	1,05
97,62	1,16
62,93	0,75

Tabla 5.14: Resultados de la exactitud y la precisión.

RESULTADOS CONJUNTO DE DATOS DE EVALUACIÓN	
Exactitud	89,99%
Precisión	100%
Sensibilidad	80,00%
Valor F1	88,89%
Matriz de Confusión	$\begin{pmatrix} 12 & 0 \\ 3 & 15 \end{pmatrix}$

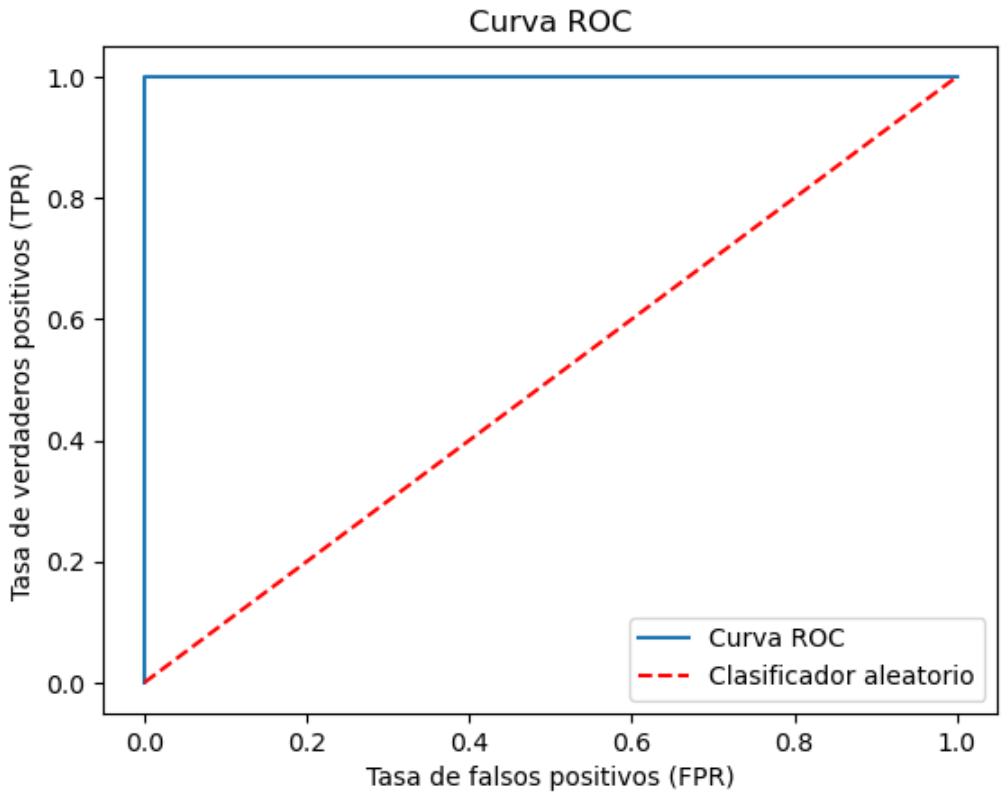


Fig. 5.20: Curva ROC del ensayo con aprendizaje transferido.

Tabla 5.15: Valor del AUC extraída del análisis de los datos de evaluación, con aprendizaje transferido.

PARÁMETROS EXTRAIDOS DEL CÁLCULO DE LA CURVA ROC	
AUC	1,00

El resultado obtenido no es nada más lejos de la realidad, ya que realmente disponemos de casi los mismos números que la inferencia realizada con los datos de validación. Lo único, es que ni en este caso, ni en el anterior, se han obtenido falsos positivos, haciendo por tanto que la precisión sea de 1. Seguidamente, si se observa la gráfica ROC y su AUC, parece ser que la red neuronal sabe hacer una distinción perfecta entre fracturas y huesos sanos. No obstante, hay algunos puntos que disertar al respecto, tal y como se comentó en el apartado 5.3.1.

6 CONCLUSIONES

Para la realización de este proyecto, se han seguido firmemente los objetivos planteados en el apartado 2 de este documento. Es por ello que a continuación se exponen los hitos que se han llevado a cabo para completar cada uno de los objetivos expuestos en el apartado ya mencionado.

- Se ha seleccionado un *framework* de trabajo (en este caso Pythorch) para el entrenamiento de una red neuronal (de la cual se ha hecho uso en este proyecto) en base a la información y comparación llevada a cabo en el apartado 1.4.2 de este documento.
- Se ha llevado a cabo una reconstrucción 3D de un hueso al completo, utilizando un módulo de 3D Slicer llamado *Volume Reconstruction*.
- Se ha entrenado un modelo de red neuronal ResNet-18 mediante dos métodos distintos y se ha combinado la actuación de este junto a la técnica *Class Activation Map* especificada en el apartado 4.4.7, todo ello para llevar a cabo detección y localización de fracturas que se ha pretendido desde un principio.
- El sistema se ha conseguido implementar en la plataforma 3D Slicer, pudiéndose realizar en este las inferencias a tiempo real en el modelo, viéndose en cada una de estas inferencias la localización de las fracturas.
- Se han evaluado los resultados obtenidos en un sistema de ensayo apropiado utilizando los parámetros estadísticos que mejor se adaptan al tipo de sistema que se ha desarrollado.

Atendiendo a lo comentado en el último punto, las estadísticas extraídas de los resultados obtenidos en el ensayo son sumamente favorables en ambos de los casos (modelo entrenado utilizando aprendizaje transferido y modelo entrenado desde cero). Dicho con otras palabras, el modelo es capaz de discernir perfectamente fracturas tomadas con un ecógrafo en tiempo real.

Sin duda alguna, este proyecto y los resultados obtenidos en él son una base importante de cara a la futura implementación de un sistema asistido con IA capaz de detectar fracturas empleando tan solo una sonda ecográfica. Esta propuesta no solo revolucionaría los sistemas de atención médica primaria en nuestra sociedad, sino que también lo haría

en sociedades menos avanzadas. También supondría un avance a nivel deportivo profesional, pues un con este sistema se podría llevar a cabo una evaluación y diagnóstico del atleta en cuestión, en donde quiera que se estuviere llevando a cabo el evento; pues como se ha comentado, una de las ventajas que ofrecen las sondas ecográficas es la portabilidad.

Las aportaciones del sistema desarrollado en este proyecto son favorables en múltiples campos. Primeramente, de cara a la comunidad científica es un avance totalmente novedoso, ya que actualmente no existe un sistema que iguale las características de las cuales este ha sido dotado. Evidentemente, la innovación y desarrollo evolutivo que se puede llevar a cabo sobre el sistema ya planteado es inminente, pero la base para que eso llegue a ser una realidad es sin duda alguna lo planteado hasta el momento en este proyecto. Por otro lado, de cara a la sociedad general, supone la posibilidad de cubrir una necesidad latente a nivel mundial, en la que cualquier persona, independientemente de la accesibilidad de la situación geográfica en la que se encuentre, siempre que haya un profesional sanitario con unas nociones básicas, podrá localizar y evaluar la posible fractura del paciente para así, posteriormente, efectuar el tratamiento que considere oportuno. Además de lo ya mencionado, la implementación de este sistema puede llegar a producir un efecto revolucionario en el mercado destinado al desarrollo tecnológico de ecógrafos, ya que la utilización de los mismos se puede llegar a convertir en una premisa indispensable en todo sistema de atención sanitaria, pues se trata de un traumatismo que, independientemente del estado de salud del paciente, siempre se puede producir en el momento menos esperado.

Un punto a tener en cuenta, teniendo en consideración el dilema moral que el uso de la inteligencia artificial en nuestra sociedad actual, este sistema no abole la necesidad de tener un experto en salud que tenga unas nociones básicas en el reconocimiento de imágenes ecográficas, pues su presencia es de vital importancia para que se pueda llevar a cabo un diagnóstico en el que el éxito de encontrar una posible fractura, sin la necesidad de utilizar un sistema de rayos X o una resonancia magnética, esté garantizado.

7 TRABAJOS FUTUROS Y POSIBLES MEJORAS

Dadas las conclusiones extraídas de este proyecto, se proponen algunas mejoras de este proyecto de cara un futuro.

- Entrenar a la red con una base de datos de imágenes adquiridas de pacientes, que cuente con fracturas y huesos de sanos de distintas zonas del cuerpo. También se valora que las imágenes estén en distintos formatos o que se hayan recogido con sondas ecográficas diferentes.
- Ofrecer resultados estadísticos de mayor precisión gracias a los resultados ofrecidos por una base de datos de validación lo suficientemente extenso.
- Hacer un análisis en redes neuronales de mayor profundidad, y comprobar si la degradación del rendimiento supone algún problema a la hora de abordar la clasificación con este tipo de imágenes. Se proponen algunas como ResNet-50, ResNet-101 y VGG-16.
- Se propone también preparar un fantoma de hueso fracturado con mayores dimensiones, y con diferentes tipos de fracturas, como lo pueden ser las fracturas segmentarias o las fracturas a tercer fragmento.
- Utilizar una U-Net para llevar a cabo la localización de la fractura en la imagen ecográfica, y comparar los resultados obtenidos con los que ofrece la técnica CAM; vista con detalle en el apartado 4.4.7.
- Montar un sistema de ensayo mucho más riguroso, en el que se tenga la opción de tener una persona ajena al proyecto para realizar la detección de fracturas utilizando el módulo.
- Intentar aunar en un mismo módulo la funcionalidad de la reconstrucción 3D de la zona afectada, junto con la funcionalidad de detección y localización de fracturas.

8 PRESUPUESTO

Para abordar este apartado del proyecto se atenderá a las indicaciones dadas por el Colegio Oficial de Ingenieros de Telecomunicaciones (COITT), quedando el precio final obtenido bajo la aprobación y aceptación del cliente. Este presupuesto quedará desglosado en varias secciones.

- Trabajo tarifado por el tiempo empleado.
- Amortización del inmovilizado material.
- Costes asociados a la redacción del documento.
- Derechos de visado del COITT.
- Gastos de tramitación y envío.
- Aplicación de impuestos.

8.1 TRABAJO TARIFADO POR EL TIEMPO EMPLEADO

Este apartado del presupuesto está destinado a contabilizar los gastos derivados de la mano de obra, según el salario correspondiente a la hora de trabajo de un Ingeniero Técnico de Telecomunicación. Para calcular estos honorarios por el tiempo dedicado, ya es un proyecto que se ha llevado a cabo bajo la tutela de la Universidad de Las Palmas de Gran Canaria, se ha acudido al documento "Clasificación y retribución del personal contratado con cargo a proyectos, programas, convenios y contratos"[37], en concreto a la modificación llevada a cabo en el año 2023. En este documento se especifica que el salario del personal técnico con titulación mínima de Grado asciende a 745,15€ mensuales, con una dedicación de 20 horas semanales; esto viene a ser un total de 24,84€. Sabiendo que la asignatura de Trabajo Fin de Grado (TFG) tiene una duración total de 300 horas, la tarifa por tiempo empleado vendrá dada por la siguiente ecuación.

$$Tarifa\ por\ tiempo\ empleado = 300 \times 24{,}84 = 7452€$$

Ec. 8.1: Ecuación de honorarios totales por el tiempo dedicado.

8.2 AMORTIZACIÓN DEL INMOVILIZADO MATERIAL

Para este apartado es necesario realizar los cálculos de amortización de los recursos *hardware* y *software* empleados para el desarrollo de este proyecto.

Para realizar este cálculo se presupone un sistema de amortización lineal en el que se supone que el inmovilizado material se desprecia de forma constante a lo largo de su vida útil. De lo explicado se deduce la siguiente ecuación que define el coste de amortización.

$$Coste\ de\ amortización = \frac{Valor\ de\ adquisición - Valor\ residual}{Años\ de\ vida\ útil}$$

Ec. 8.2: Ecuación que define el coste de amortización.

En este caso el valor residual es el valor que tendrá un bien después de su vida útil. Para el cálculo de este parámetro, se recomienda hacer uso de la tabla de coeficientes de amortización lineal de la Agencia Tributaria (AEAT) [38]. De esta tabla de coeficientes se ha extraído la siguiente información.

Tabla 8.1: Extracción de la tabla de coeficientes de amortización lineal de la Agencia Tributaria (AEAT).

Tipo de elemento	Coeficiente lineal máximo	Periodo de años máximo
Equipos médicos y asimilados	15%	14
Equipos para procesos de información	25%	8
Sistemas y programas informáticos	33%	6
Otros enseres	15%	14

8.2.1 Amortización del material hardware

La amortización del material hardware viene recogida en la tabla que se expone a continuación.

Tabla 8.2: Coste de amortización del hardware.

Recurso Hardware	Valor de adquisición	Valor residual	Vida útil (años)	Coste de amortización
Portátil Hacer Nitro AN 515-51 **Procesador** Intel Core i7-7700HQ CPU @ 2.80GHz **RAM** 8GB DDR4 **Tarjeta Gráfica** NVIDIA GeForce GTX 1050 4 GB	998,00€	249,5€	4	187,13€
Telemed Sonda Lineal L12-5L40S-3	1800,00€	270,00€	9	170,00€
Telemed MicrUs EXT-1M KIT	4035,00€	605,25€	9	381,08€
3D Guidance trakSTAR (con todos sus componenetes)	5181,43€	777,21€	9	489,35
Fantoma de hueso fracturado: Vinilo líquido + Ablandador + Celulosa Microcristalina + Arcilla	8,67€	1,3€	1	9,90€
Microondas + Recipiente	64€	9,6€	9	6,04€
TOTAL				1243,50€

8.2.2 Amortización del material software

La amortización del material software viene recogida en la tabla que se expone a continuación.

Tabla 8.3: Coste de amortización del software.

Recurso Hardware	Coste de amortización
Anaconda	0,00€
Jupyter Notebook	0,00€
Visual Studio Code	0,00€
Pytorch	0,00€
3D Slicer	0,00€
Echo Wave II	0,00€
PLUS Server Launcher	0,00€
Microsoft Office 365	0,00€
TOTAL	0,00€

Prácticamente todas las herramientas software utilizadas son totalmente gratuitas, exceptuando al Microsoft Office 365 y el Echo Wave II. En el caso del Microsoft Office 365, se dispone de una licencia dada por la Universidad de Las Palmas de Gran Canaria que hace este software de carácter gratuito para los estudiantes. Por otro lado, el software Echo Wave II viene incluido con la compra de la sonda lineal de Telemed, por lo que viene incluido en el precio del mismo.

8.3 COSTES ASOCIADOS A LA REDACCIÓN DEL DOCUMENTO

Para determinar el coste asociado a la redacción del proyecto se empleará la siguiente ecuación.

$$R = 0{,}07 \times P \times C_n$$

Ec. 8.3: Ecuación de coste por redacción del proyecto.

, donde

- ***R***: Honorarios por la redacción del documento.
- ***P***: Presupuesto del proyecto.
- ***C_n***: Coeficiente de ponderación en función del presupuesto.

El cálculo de ***P*** se lleva a cabo mediante la suma del trabajo tarifado por el tiempo empleado y la amortización total del inmovilizado material. En la siguiente tabla se hace un desglose de esta suma.

Tabla 8.4: Cálculo de P derivada de la ecuación de coste por redacción.

Tipo de cargo	Coste
Amortización trabajo tarifado por el tiempo empleado	7452,00€
Amortización del material hardware	1243,50€
Amortización del material software	0,00€
TOTAL	8695,50€

Por otro lado, para el cálculo de $\mathbf{C_n}$ se tiene en cuenta lo estipulado por el COITT, en la que para presupuestos menores a 30.050,00€ se tiene un $\mathbf{C_n}$ = 1. Con estos datos se tiene lo siguiente.

$$R = 0{,}07 \times 8695{,}50 \times 1 = 608{,}69€$$

Ec. 8.4: Resultado de los honorarios por la redacción del proyecto.

Nuevamente, el coste hasta ahora calculado está totalmente libre de impuestos.

8.4 DERECHOS DE VISADO DEL COITT

Los gastos de visado del COITT se calculan atendiendo a la expresión que se muestra.

$$V = 0{,}006 \times P_1 \times C_1 + 0{,}003 \times P_2 \times C_2$$

Ec. 8.5: Ecuación para el cálculo de los gastos de visado del COITT.

, donde

- ***V***: Coste de visado del trabajo.
- ***P_1***: Prepuesto del proyecto.
- ***C_1***: Coeficiente reductor dado en función del presupuesto del trabajo.
- ***P_2***: Presupuesto de ejecución material correspondiente a la obra civil.
- ***C_2***: Coeficiente reductor dado en función de ***P_2***.

El valor de ***P_1*** vendrá dado por la suma del trabajo tarifado por tiempo empleado y la amortización del inmovilizado material (este valor ***P_1*** calculado anteriormente en la Tabla 7.5). El valor de ***C_1*** vuelve a ser 1 ya que se toma como premisa este valor para los proyectos con un presupuesto inferior a 30.050,00€. Por último, el valor de ***P_2*** será de 0,00€ ya se no se realiza ninguna obra. De esta forma, aplicando la ecuación anterior, el valor de ***V*** queda tal que así.

$$V = 0{,}006 \times 8695{,}50 \times 1 + 0{,}003 \times 0 \times 1 = 52{,}17€$$

Ec. 8.6: Resultado del cálculo de los gastos de visado del COITT.

8.5 GATOS DE TRAMITACIÓN Y ENVÍO

Los gastos de tramitación están estipulados en seis euros (6€) por cada documento enviado de forma telemática.

8.6 APLICACIÓN DE IMPUESTOS

Los costes totales de este proyecto están gravados por el Impuesto General Indirecto Canario (IGIC), que actualmente es de un siete por ciento (7%). En la siguiente tabla se muestra el presupuesto final con el impuesto correspondiente aplicado.

Tabla 8.5: Coste total del proyecto con los impuestos correspondientes aplicados.

Concepto	Coste
Trabajo tarifado por tiempo empleado	7452,00€
Amortización del material hardware	1243,50€
Amortización del material software	0,00€
Costes asociados a la redacción del documento	608,69€
Derechos de visado del COITT	52,17€
Gastos de tramitación y envío	6,00€
SUBTOTAL	9362,36€
IGIC (7%)	655,36€
TOTAL	10.017,72€

El presupuesto total del proyecto “Detección y segmentación de fracturas y fisuras óseas mediante imágenes ecográficas con IA” asciende a *diez mil diecisiete euros y setenta y dos céntimos* (10.017,72€).

9 BIBLIOGRAFÍA

[1] Clínica Martín Gómez Traumatólogos, «Diferencia entre fisura ósea y fractura de hueso». https://clinicamartingomez.es/diferencia-entre-fisura-osea-y-fractura-de-hueso/ (accedido 28 de abril de 2023).

[2] A. AYBAR MONTOYA, «Clasificaciones en fracturas», 2012.

[3] Schneider F.R., *Handbook for the ORTHOPAEDIC ASSISTANT*, Second Edition. McGraw-Hill Inc., 1976.

[4] Dr. Alexander Torres Molina, «Huesos de cristal. Presentación de un caso», *Revista Electrónica de las Ciencias Médicas en Cienfuegos*, 2010, Accedido: 2 de mayo de 2023. [En línea]. Disponible en: http://scielo.sld.cu/pdf/ms/v8n4/v8n4a1057.pdf

[5] Y. Bishitz *et al.*, «Noncontact optical sensor for bone fracture diagnostics», *Biomed Opt Express*, vol. 6, n.º 3, p. 651, mar. 2015, doi: 10.1364/boe.6.000651.

[6] «Rayos X». https://www.nibib.nih.gov/espanol/temas-cientificos/rayos-x (accedido 12 de junio de 2023).

[7] «Tomografía Computarizada (TC)». https://www.nibib.nih.gov/espanol/temas-cientificos/tomograf%C3%ADa-computarizada-tc (accedido 12 de junio de 2023).

[8] «Imagen por Resonancia Magnética (IRM)». https://www.nibib.nih.gov/espanol/temas-cientificos/imagen-por-resonancia-magn%C3%A9tica-irm (accedido 12 de junio de 2023).

[9] M. S. Linet *et al.*, «Cancer Risks Associated with External Radiation From Diagnostic Imaging Procedures», *CA Cancer J Clin*, vol. 62, n.º 2, p. 75, mar. 2012, doi: 10.3322/CAAC.21132.

[10] N. Díaz-Rodríguez, R. P. Garrido-Chamorro, y J. Castellano-Alarcón, «Metodología y técnicas. Ecografía: principios físicos, ecógrafos y lenguaje

ecográfico», *Medicina de Familia. SEMERGEN*, vol. 33, n.º 7, pp. 362-369, ago. 2007, doi: 10.1016/S1138-3593(07)73916-3.

[11] N. Champagne, L. Eadie, L. Regan, y P. Wilson, «The effectiveness of ultrasound in the detection of fractures in adults with suspected upper or lower limb injury: A systematic review and subgroup meta-analysis», *BMC Emergency Medicine*, vol. 19, n.º 1. BioMed Central Ltd., 28 de enero de 2019. doi: 10.1186/s12873-019-0226-5.

[12] F. Lara y R. Indice, «FUNDAMENTOS DE REDES NEURONALES ARTIFICIALES».

[13] «¿Qué es el Perceptrón? Perceptrón Simple y Multicapa». https://aprendeia.com/que-es-el-perceptron-simple-y-multicapa/ (accedido 8 de julio de 2023).

[14] «Pytorch Vs Tensorflow Vs Keras: Here are the Difference You Should Know». https://www.simplilearn.com/keras-vs-tensorflow-vs-pytorch-article#what_is_deep_learning (accedido 12 de junio de 2023).

[15] «Keras: Deep Learning for humans». https://keras.io/ (accedido 12 de junio de 2023).

[16] «TensorFlow». https://www.tensorflow.org/?hl=es-419 (accedido 12 de junio de 2023).

[17] «PyTorch». https://pytorch.org/ (accedido 12 de junio de 2023).

[18] «MONAI - Home». https://monai.io/ (accedido 12 de junio de 2023).

[19] «Mapas de calor: qué son y cómo utilizarlos para mejorar tu SEO [2023]». https://www.dosmedia.com/mapas-de-calor/ (accedido 12 de junio de 2023).

[20] Álvaro Artola Moreno, «Clasificación de imágenes usando redes neuronales convolucionales en Python». Accedido: 20 de mayo de 2023. [En línea]. Disponible en: https://idus.us.es/bitstream/handle/11441/89506/TFG-2402-ARTOLA.pdf?sequence=1&isAllowed=y

[21] «¿GPU vs. CPU? ¿Qué es la computación por GPU? | NVIDIA». https://www.nvidia.com/es-la/drivers/what-is-gpu-computing/ (accedido 22 de mayo de 2023).

[22] «¿GPU vs. CPU? ¿Qué es la computación por GPU? | NVIDIA». https://www.nvidia.com/es-la/drivers/what-is-gpu-computing/ (accedido 12 de junio de 2023).

[23] «3D Slicer image computing platform | 3D Slicer». https://www.slicer.org/ (accedido 14 de junio de 2023).

[24] A. Fedorov *et al.*, «3D Slicer as an Image Computing Platform for the Quantitative Imaging Network», *Magn Reson Imaging*, vol. 30, n.º 9, pp. 1323-1341, 2012, doi: 10.1016/j.mri.2012.05.001.

[25] A. Lasso, T. Heffter, A. Rankin, C. Pinter, T. Ungi, y G. Fichtinger, «PLUS: open-source toolkit for ultrasound-guided intervention systems», *IEEE Trans Biomed Eng*, vol. 61, n.º 10, pp. 2527-2537, oct. 2014, doi: 10.1109/TBME.2014.2322864.

[26] A. Lasso, T. Heffter, A. Rankin, C. Pinter, T. Ungi, y G. Fichtinger, «PLUS: Open-source toolkit for ultrasound-guided intervention systems», *IEEE Trans Biomed Eng*, vol. 61, n.º 10, pp. 2527-2537, oct. 2014, doi: 10.1109/TBME.2014.2322864.

[27] J. Tokuda *et al.*, «OpenIGTLink: an open network protocol for image-guided therapy environment», *Int J Med Robot*, vol. 5, n.º 4, p. 423, 2009, doi: 10.1002/RCS.274.

[28] T. Ungi, A. Lasso, y G. Fichtinger, «Open-source platforms for navigated image-guided interventions», *Med Image Anal*, vol. 33, pp. 181-186, oct. 2016, doi: 10.1016/J.MEDIA.2016.06.011.

[29] «Sonda para ecografía lineal - L12-5L40S-3 - Telemed Medical Systems - vascular / musculoesquelética / para partes pequeñas». https://www.medicalexpo.es/prod/telemed-medical-systems/product-70298-718260.html (accedido 24 de mayo de 2023).

[30] «3D Guidance - NDI». https://www.ndigital.com/electromagnetic-tracking-technology/3d-guidance/ (accedido 7 de junio de 2023).

[31] «3D Guidance Sensors - NDI». https://www.ndigital.com/electromagnetic-tracking-technology/3d-guidance/3d-guidance-sensors/ (accedido 7 de junio de 2023).

[32] L. Bartha, A. Lasso, C. Pinter, T. Ungi, Z. Keri, y G. Fichtinger, «Open-source surface mesh-based ultrasound-guided spinal intervention simulator», *Int J Comput Assist Radiol Surg*, vol. 8, n.º 6, pp. 1043-1051, 2013, doi: 10.1007/S11548-013-0901-Z.

[33] M. E. Atik y Z. Duran, «Deep Learning-Based 3D Face Recognition Using Derived Features from Point Cloud», en *Lecture Notes in Networks and Systems*, Springer Science and Business Media Deutschland GmbH, 2021, pp. 797-808. doi: 10.1007/978-3-030-66840-2_60.

[34] «Redes Neuronales». https://ml4a.github.io/ml4a/es/neural_networks/ (accedido 1 de junio de 2023).

[35] B. Zhou, A. Khosla, A. Lapedriza, A. Oliva, y A. Torralba, «Learning Deep Features for Discriminative Localization», Accedido: 12 de junio de 2023. [En línea]. Disponible en: http://cnnlocalization.csail.mit.edu

[36] H. Qadi, J. Davidson, M. Trauer, y R. Beese, «Ultrasound of bone fractures», *Ultrasound*, vol. 28, n.º 2, pp. 118-123, may 2020, doi: 10.1177/1742271X20901824.

[37] «MODIFICACIÓN DE LAS RETRIBUCIONES DEL PERSONAL INVESTIGADOR, DEL PERSONAL TÉCNICO Y DEL PERSONAL TÉCNICO DE APOYO CONTRATADO DE LA ULPGC CON CARGO A PROYECTOS, CONVENIOS Y CONTRATOS PARA EL AÑO 2023».

[38] «Tabla de coeficientes de amortización y años | SP Auditoría». https://www.spauditoria.com/tabla-de-amortizacion (accedido 8 de junio de 2023).

10 ANEXOS

10.1 EJEMPLO DE ENTRENAMIENTO

FOLDER ESTRUCTURATION

```
def load_file(path):
    return np.load(path).astype(np.float32)
root_path = "Ruta_donde_están_las_imágenes_tomadas"
save_path = "Ruta_de_guradado"
sums, sums_squared = 0, 0
cat = ["Healthy", "Fracture"]
a = 0
if not os.path.exists(save_path + "/" + "train"):
    for c, patient_status in enumerate(tqdm(cat)):

        path = root_path + "/" + patient_status
        i = len(os.listdir(path))

        for x in range(i):

            path = root_path + "/" + patient_status + "/" + patient_status + str(x) +
".png"

            img = Image.open(path)
            img_np = np.asarray(img)/255
            img_array = cv2.resize(img_np, (224, 224)).astype(np.float16)

            label = 0 if patient_status == "Healthy" else 1
            train_or_val = "train" if x <=47 else "val"

            current_save_path = save_path + "/" + train_or_val
            if not os.path.exists(current_save_path):
                os.mkdir(os.path.join(save_path, train_or_val))

            current_save_path = save_path + "/" + train_or_val + "/" + str(label)
            if not os.path.exists(current_save_path):
                current = save_path + "/" + train_or_val
                os.mkdir(os.path.join(current, str(label)))
```

```
            current_save_path = current_save_path + "/" + patient_status + str(x)
            np.save(current_save_path, img_array)

            normalizer = 224*224
            if train_or_val == "train":
                scale_factor = 100
                img_array_scaled = img_array / scale_factor
                sums += np.sum(img_array_scaled) / normalizer
                sums_squared += (img_array_scaled**2).sum() / normalizer
```

Código 10.1

DATASET AND DATALOADER CREATION

```
train_transforms = transforms.Compose([
                                transforms.ToTensor(),
                                transforms.Normalize(0.129, 0.119),
                                transforms.RandomHorizontalFlip(),
                                transforms.RandomRotation(15)
])
val_transforms = transforms.Compose([
                                transforms.ToTensor(),
                                transforms.Normalize(0.129, 0.119)
])
train_dataset = torchvision.datasets.DatasetFolder("Ruta_de_datos_de_entrenamiento",
loader=load_file, extensions="npy", transform=train_transforms)
val_dataset =
torchvision.datasets.DatasetFolder("Ruta_de_datos_de_validación",loader=load_file,
extensions="npy", transform=val_transforms)
batch_size=10
train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True )
val_loader = DataLoader(val_dataset, batch_size=batch_size, shuffle=False )
```

Código 10.2

TRAINING PREPARATION

```
class FractureModel(pl.LightningModule):
    def __init__(self, weight=1):
        super().__init__()
```

```
        self.model = torchvision.models.resnet18()
        # Se cambia conv1 de 3 a 1 canal de input.
        self.model.conv1 = torch.nn.Conv2d(1, 64, kernel_size=(7, 7), stride=(2, 2),
padding=(3, 3), bias=False)
        # Se cambia out_feature de el fully connected layer (fc en resnet18) de 1000 a 1.
        self.model.fc = torch.nn.Linear(in_features=512, out_features=1)

        self.optimizer = optim.Adam(self.model.parameters(), lr=0.001)
        self.loss = torch.nn.BCEWithLogitsLoss(torch.tensor([weight]))
        self.val_acc = torchmetrics.Accuracy("binary")

    def forward(self, data):
        pred = self.model(data)
        return pred # El modelo devuelve la predicción.

    def training_step(self, batch, batch_idx):
        us_image, label = batch
        label = label.float()  # Convierte la estiqueta en float (para obetener la loss
function).
        pred = self(us_image)[:,0]
        loss = self.loss(pred, label)  # Se obtiene la pérdida "loss".
        acc = self.val_acc(torch.sigmoid(pred), label.int())
        # Se hace un log de el loss y la acertividad.
        self.log('train_loss', loss)
        self.log('train_acc', acc)
        return loss

    def validation_step(self, batch, batch_idx):
        # Los mismos pasos seguidos que en el training_step.
        us_image, label = batch
        label = label.float()
        pred = self(us_image)[:,0]
        loss = self.loss(pred, label)
        acc = self.val_acc(torch.sigmoid(pred), label.int())
        # Se hace un log de el loss y la acertividad.
        self.log('val_loss', loss)
        self.log('val_acc', acc)

    def configure_optimizers(self):
        return [self.optimizer]
```

Código 10.3

```
model = FractureModel()


device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
model.to(device);


checkpoint = ModelCheckpoint(
    monitor='val_acc',
    mode='max',
    filename='best_model_{epoch:02d}',
    save_top_k=1,
    save_weights_only=True)


# Creación del trainer
trainer = pl.Trainer("cuda", logger=CSVLogger(save_dir="./logs"), log_every_n_steps=1,
                     callbacks= [checkpoint],
                     precision=16,
                     max_epochs=120)


trainer.fit(model, train_loader, val_loader)
```

Código 10.4

SAVING THE MODEL STATE

```
mod = FractureModel.load_from_checkpoint("Ruta_para_cargar_el_checkpoint_formato_CKPT")


ruta_guardado = 'Ruta_de_guardado_formato_PTH'
torch.save({'state_dict': mod.state_dict()}, ruta_guardado)
```

Código 10.5

10.2 EJEMPLO DE INFERENCIA

CLASS DEFINITION

```
class FractureModel(nn.Module):

    def __init__(self, weight=1):

        super().__init__()


        self.model = torchvision.models.resnet18()

        self.model.conv1 = torch.nn.Conv2d(1, 64, kernel_size=(7, 7), stride=(2, 2),
padding=(3, 3), bias=False)

        self.model.fc = torch.nn.Linear(in_features=512, out_features=1)

        self.loss = torch.nn.BCEWithLogitsLoss(torch.tensor([weight]))

    def forward(self, data):

        pred = self.model(data)

        return pred


model = FractureModel()
```

Código 10.6

LOADING TRAINED MODEL AND DEFINING THE TRANSFORMS

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

checkpoint = torch.load("Ruta_de_carga_archivo_guardado_en_PTH")

print(checkpoint.keys())

model.load_state_dict(checkpoint['state_dict'])

model.eval();

print(device)

model.to(device);


val_transforms = transforms.Compose([
                                transforms.ToTensor(),
                                transforms.Normalize(0.129, 0.119)
])
```

Código 10.7

INFERENCE

```
def load_file(path):

    return np.load(path).astype(np.float32)


# Se define el dataset.

val_dataset = torchvision.datasets.DatasetFolder("Ruta_de_los_datos_de_validación",
loader=load_file, extensions="npy", transform=val_transforms)


# Variables que van a ser rellenadas dentro del bucle for.

preds = []

labels = []

names = []

# Contador.

i = 0

for data, label in tqdm(val_dataset):

    data = data.to(device).float().unsqueeze(0) # Se adecua el dato indexado para pasarlo
por el modelo.

    pred = torch.sigmoid(model(data)) # Predicción dada por el modelo.

    preds.append(pred) # Se llena el array vacío preds.

    labels.append(label) # Se llena el array vacío label.

    filename = os.path.basename(val_dataset.samples[i][0]) + " " + "pos. array = " + str(i)
#Se obtiene el nombre del archivo

                                                                                    #
del cual se ha hecho una predicción.

    names.append(filename) # Se llena el array names con los nombres de los archivos
sujetos a predicción.

    i+=1 # Se incrementa el contador.


# Se convierten los arrays en tensores.

preds = torch.tensor(preds)

labels = torch.tensor(labels).int()
```

Código 10.8

```
img1 =
Image.open("D:/FractureUltrasoundSimulator/FractureUltrasoundSimulator/Resources/Data/Datas
et_Fantoma/Fracture/Fracture63.png")

img_np = np.asarray(img1)/255 # Se convierte la imagen de Pillow a un array de Numpy y se
normaliza.

img_array = cv2.resize(img_np, (224, 224)).astype(np.float16)

plt.imshow(img_array, cmap='gray')

mean = np.mean(img_array)

std = np.std(img_array)
```

```
img_t2 = val_transforms(img_array).to(device).unsqueeze(0).float() # Se le aplican las
transformaciones previas.

pred_img_t2 = torch.sigmoid(model(img_t2)) # Se pasa la imagen al modelo, y se guarda su
predicción.

round(pred_img_t2.item()*100, 4) # Se muestra la predicción

preds[31] * 100 # Se muestra la predicción que se guardón en el array "preds",
correspondiente al archivo que se ha utilizado para esta comprobación.
```

Código 10.9

```
class FractureModelMap(nn.Module):

    def __init__(self):

        super().__init__()



        self.model = torchvision.models.resnet18()

        self.model.conv1 = torch.nn.Conv2d(1, 64, kernel_size=(7, 7), stride=(2, 2),
padding=(3, 3), bias=False)

        self.model.fc = torch.nn.Linear(in_features=512, out_features=1)



        self.feature_map = torch.nn.Sequential(*list(self.model.children())[:-2])

        print(self.feature_map)



    def forward(self, data):



        feature_map = self.feature_map(data)

        return feature_map



temp_model = FractureModelMap()



temp_model.load_state_dict(checkpoint['state_dict'], strict = False)

temp_model.eval();

temp_model.to(device);



def heatmap_creation(model, img):

    with torch.no_grad(): # Deshabilita el cálculo del gradiente de la red neuronal. Se
hace para ahorrar memoria y acelerar el cálculo.

        features = model(img.unsqueeze(0))

    features = features.reshape((512, 49))

    weight_params = list(model.model.fc.parameters())[0] # Otención de la matriz de pesos
weight_params de la capa fc.

    weight = weight_params[0].detach() # Extrae la matriz de pesos weight de la variable
weight_params y la separa del grafo de cálculo con el método detach(.

    mult = torch.matmul(weight, features) # Multiplicación de matriz entre la matriz de
pesos weight y las características features.
```

```
    heatmap_img = mult.reshape(7, 7).cpu()
    return heatmap_img
```

Código 10.10

10.3 CÓDIGO DEL MÓDULO DE 3D SLICER

```
 1 import vtk, qt, ctk, slicer
 2 import os
 3 import numpy as np
 4 from slicer.ScriptedLoadableModule import *
 5 from slicer.util import VTKObservationMixin
 6 import logging
 7
 8 # Load PyTorch library
 9 try
10   import torch
11   import torch.nn as nn
12   import torchvision
13   from torchvision import transforms
14   DEVICE = 'cuda
15 except
16   logging.error('PyTorch is not installed. Please, install PyTorch...')
17
18 # Load OpenCV
19 try
20   import cv2
21 except
22   logging.error('OpenCV is not installed. Please, install OpenCV...')
23
```

Código 10.11

```
25 #------------------------------------------------------------------------------
26 #
27 # FractureClassification
28 #
29 #------------------------------------------------------------------------------
30 class FractureClassification(ScriptedLoadableModule)
31
32   def __init__(self, parent)
33     ScriptedLoadableModule.__init__(self, parent)
34     self.parent.title = "Fracture Classification"
35     self.parent.categories = ["Ultrasound AI"]
36     self.parent.dependencies = []
```

```
37      self.parent.contributors = ["Nestor Manuel Acosta Rivero (ULPGC), David Garcia Mato
    (Ebatinca)"]
38      self.parent.helpText = """ Module to classify and detect fractures in US images using
    deep learning. """
39      self.parent.acknowledgementText = """EBATINCA, S.L."""
40
```

Código 10.12

```
42  #------------------------------------------------------------------------------
43  #
44  # FractureClassificationWidget
45  #
46  #------------------------------------------------------------------------------
47  class FractureClassificationWidget(ScriptedLoadableModuleWidget, VTKObservationMixin)
48
49    def __init__(self, parent)
50      """
51      Called when the user opens the module the first time and the widget is initialized.
52      """
53      ScriptedLoadableModuleWidget.__init__(self, parent)
54      VTKObservationMixin.__init__(self)  # needed for parameter node observation
55      #Flags
56      self.showHeatMap = False
57      self.showSegmentation = False
58
59      # Create logic class
60      self.logic = FractureClassificationLogic(self)
61
62      # DEVELOPMENT
63      slicer.FractureClassificationWidget = self
64
```

Código 10.13

```
65    #------------------------------------------------------------------------------
66    def setup(self)
67      """
68      Called when the user opens the module the first time and the widget is initialized.
69      """
70      ScriptedLoadableModuleWidget.setup(self)
71
72      # Set up UI
73      self.setupUi()
74
75      # Setup connections
76      self.setupConnections()
```

```
 77
 78      # The parameter node had defaults at creation, propagate them to the GUI
 79      self.updateGUIFromMRML()
 80
 81    #------------------------------------------------------------------------------
 82    def cleanup(self)
 83      self.disconnect()
 84
 85    #------------------------------------------------------------------------------
 86    def enter(self)
 87      """
 88      Runs whenever the module is reopened
 89      """
 90
 91      # Update GUI
 92      self.updateGUIFromMRML()
 93
 94    #------------------------------------------------------------------------------
 95    def exit(self)
 96      """
 97      Runs when exiting the module.
 98      """
 99      pass
100
```

Código 10.14

```
101    #------------------------------------------------------------------------------
102    def setupUi(self)
103      # Load widget from .ui file (created by Qt Designer).
104      # Additional widgets can be instantiated manually and added to self.layout.
105      uiWidget = slicer.util.loadUI(self.resourcePath('UI/FractureClassification.ui'))
106      self.layout.addWidget(uiWidget)
107      self.ui = slicer.util.childWidgetVariables(uiWidget)
108
109      # Set scene in MRML widgets. Make sure that in Qt designer the top-level
qMRMLWidget's
110      # "mrmlSceneChanged(vtkMRMLScene*)" signal in is connected to each MRML widget's.
111      # "setMRMLScene(vtkMRMLScene*)" slot.
112      uiWidget.setMRMLScene(slicer.mrmlScene)
113
114      # Customize widgets
115      self.ui.modelPathLineEdit.currentPath = self.logic.defaultModelFilePath
116
117    #------------------------------------------------------------------------------
118    def setupConnections(self)
119      self.ui.inputSelector.currentNodeChanged.connect(self.onInputSelectorChanged)
```

```
120     self.ui.loadModelButton.connect('clicked(bool)', self.onLoadModelButton)
        self.ui.startClassificationButton.connect('clicked(bool)',
121 self.onStartClassificationButton)
122     self.ui.showHeatmapButton.connect('clicked(bool)', self.onShowHeatMap)
123     self.ui.showSegmentationButton.connect('clicked(bool)', self.onShowSegmentation)
124
125   #------------------------------------------------------------------------
126   def disconnect(self)
127     self.ui.inputSelector.currentNodeChanged.disconnect()
128     self.ui.loadModelButton.clicked.disconnect()
129     self.ui.startClassificationButton.clicked.disconnect()
130     self.ui.showHeatmapButton.clicked.disconnect()
131     self.ui.showSegmentationButton.clicked.disconnect()
132
```

Código 10.15

```
133   #------------------------------------------------------------------------
134   def updateGUIFromMRML(self, caller=None, event=None)
135     """
136     Set selections and other settings on the GUI based on the parameter node.
137
138     Calls the updateGUIFromMRML function of all tabs so that they can take care of their
    own GUI.
139     """
140     # Display selected volume in red slice view
141     inputVolume = self.ui.inputSelector.currentNode()
142     if inputVolume
143       self.logic.displayVolumeInSliceView(inputVolume)
144
145     # Activate buttons
146     self.ui.startClassificationButton.enabled = (self.ui.inputSelector.currentNode() !=
    None and self.logic.classificationModel != None)
147     self.ui.showHeatmapButton.enabled = (self.logic.classificationFlag == True)
148     self.ui.showSegmentationButton.enabled = (self.logic.classificationFlag == True)
149
150   #------------------------------------------------------------------------
151   def onInputSelectorChanged(self)
152     # Update GUI
153     self.updateGUIFromMRML()
154
155   #------------------------------------------------------------------------
156   def onLoadModelButton(self)
157     # Acquire path from the line in UI
158     modelFilePath = self.ui.modelPathLineEdit.currentPath
159
160     # Load model using the function in the logic section
161     self.logic.loadModel(modelFilePath)
162
```

```
163     # Update GUI
164     self.updateGUIFromMRML()
```

Código 10.16

```
166   #------------------------------------------------------------------------
167   def onStartClassificationButton(self)
168     # Get input volume
169     nodeName = self.ui.inputSelector.currentNode().GetName()
170
171     # Classification
172     [fracture_probability, heatMap] = self.logic.startClassification(nodeName)
173
174     # Display fracture probability in UI
175     self.ui.fractureProbabilityLabel.text = str(fracture_probability)
176
177     # Display heatmap
178     self.logic.convertHeatMapToVolume(heatMap)
179
180     # Segment heatmap and display segmentation
181     self.logic.heatMapSegmentation(heatMap)
182
183     # Update GUI
184     self.updateGUIFromMRML()
185
```

Código 10.17

```
186   #------------------------------------------------------------------------
187   def onShowHeatMap(self)
188     # Variable creation
189     layoutManager = slicer.app.layoutManager()
      sliceCompositeNode =
190 layoutManager.sliceWidget('Red').sliceLogic().GetSliceCompositeNode()
191
192     # Update visibility
193     if self.showHeatMap == False
194       self.showHeatMap = True # Flag change
195       sliceCompositeNode.SetForegroundOpacity(0.5)
196       self.ui.showHeatmapButton.text = 'Hide HeatMap'
197     else
198       self.showHeatMap = False # Flag change
199       sliceCompositeNode.SetForegroundOpacity(0)
200       self.ui.showHeatmapButton.text = 'Show HeatMap'
201
202     # Update GUI
```

```
203      self.updateGUIFromMRML()
204
```

Código 10.18

```
205    #------------------------------------------------------------------------
206    def onShowSegmentation(self)
207      # Variable creation
208      segmentation = slicer.util.getNode('Segmentation')
209      segmentId = segmentation.GetSegmentation().GetSegmentIdBySegmentName('Segmentation')
210
211      # Update visibility
212      if self.showSegmentation == False
213        self.showSegmentation = True # Flag change
214        segmentation.GetDisplayNode().SetSegmentVisibility(segmentId, True)
215        self.ui.showSegmentationButton.text = 'Hide Segmentation'
216      else
217        self.showSegmentation = False # Flag change
218        segmentation.GetDisplayNode().SetSegmentVisibility(segmentId, False)
219        self.ui.showSegmentationButton.text = 'Show Segmentation'
220
221      # Update GUI
222      self.updateGUIFromMRML()
```

Código 10.19

```
225 #------------------------------------------------------------------------------
226 #
227 # FractureClassificationLogic
228 #
229 #------------------------------------------------------------------------------
230 class FractureClassificationLogic(ScriptedLoadableModuleLogic, VTKObservationMixin)
231
232   def __init__(self, widgetInstance, parent=None)
233     """
      Called when the logic class is instantiated. Can be used for initializing member
234 variables.
235     """
236     ScriptedLoadableModuleLogic.__init__(self, parent)
237     VTKObservationMixin.__init__(self)
238     self.moduleWidget = widgetInstance
239
240     # Flag
241     self.classificationFlag = False
242
243     # Input image array and other properties
```

```
244     self.inputImage = None
245     self.numRows = None
246     self.numCols = None
247
248     # Classification model and HeatMap model
249     self.defaultModelFilePath = self.moduleWidget.resourcePath('Archivo.pth')
250     self.classificationModel = None
251     self.segmentationModel = None
252
253     # Definition of transformations applied to image.
254     self.transformations = transforms.Compose([
255                         transforms.ToTensor(),
256                         transforms.Normalize(0.12967301527195704, 0.11952487479741893)
257     ])
258     # Red slice
259     self.redSliceLogic = slicer.app.layoutManager().sliceWidget("Red").sliceLogic()
260
```

Código 10.20

```
261   #------------------------------------------------------------------------------
262   def displayVolumeInSliceView(self, volumeNode)
263     # Display input volume node in red slice background
264     self.redSliceLogic.GetSliceCompositeNode().SetBackgroundVolumeID(volumeNode.GetID())
265     self.redSliceLogic.FitSliceToAll()
266
```

Código 10.21

```
267   #------------------------------------------------------------------------------
268   def loadModel(self, modelFilePath)
269     """
270     Loads PyTorch model for classification
271     :param modelFilePath: path where the model file is saved
272     :return: True on success, False on error
273     """
274     print('Loading model...')
275     # Load classification model
276     try
277       self.classificationModel = FractureModel() # Model instance
278       checkpoint = torch.load(modelFilePath, map_location=torch.device(DEVICE))
279       self.classificationModel.load_state_dict(checkpoint['state_dict'])  # Loads the weights stored in the 'state_dict' key
280       self.classificationModel.to(DEVICE)
281       self.classificationModel.eval()
282     except Exception as e
```

```
283       self.classificationModel = None
284       logging.error("Failed to load classification model
285       return False
286
287     # Load segmentation model
288     try
289       self.segmentationModel = FractureModelMap() # HeatMap Model instance.
290       checkpoint = torch.load(modelFilePath, map_location=torch.device(DEVICE)
291       self.segmentationModel.load_state_dict(checkpoint['state_dict'], strict = False)
292       self.segmentationModel.to(DEVICE)
293       self.segmentationModel.eval()
294     except Exception as e
295       self.classificationModel = None
296       logging.error("Failed to load segmentation model
297       return False
298
299     return True
300
```

Código 10.22

```
301   #------------------------------------------------------------------------
302   def convertHeatMapToVolume(self, heatMap)
303     """
304     Generates heatmap visualization.
305     :param heatMap: heatmap tensor.
306     :return: True on success, False on error.
307     """
308     # Map transformations
309     heatMap = np.array(heatMap)
310     img_array = slicer.util.arrayFromVolume(self.inputImage)[0,
311
312     # Data acquisition
313     self.numRows = img_array.shape[0]
314     self.numCols = img_array.shape[1]
315     heatMap = cv2.resize(heatMap , (self.numCols, self.numRows)).astype(np.float16)
316
317     # Remove existing node if any
318     try
319       node = slicer.util.getNode('Heatmap')
320       slicer.mrmlScene.RemoveNode(node)
321     except
322       pass
323
324     # Clone the node
```

```
325     shNode = slicer.vtkMRMLSubjectHierarchyNode.GetSubjectHierarchyNode(slicer.mrmlScene)
326     itemIDToClone = shNode.GetItemByDataNode(self.inputImage)
        clonedItemID = slicer.modules.subjecthierarchy.logic().CloneSubjectHierarchyItem(shNode,
327 itemIDToClone)
328     clonedNode = shNode.GetItemDataNode(clonedItemID)
329     clonedNode.SetName('Heatmap')
330
331     # Updating Red slice with the HeatMap
332     slicer.util.updateVolumeFromArray(clonedNode, heatMap)
333     sliceCompositeNode = self.redSliceLogic.GetSliceCompositeNode()
334     sliceCompositeNode.SetForegroundVolumeID(clonedNode.GetID())
335     sliceCompositeNode.SetForegroundOpacity(0)
336
337     # Modify display properties
338     heatmapVolumeDisplayNode = clonedNode.GetDisplayNode()
    heatmapVolumeDisplayNode.SetAndObserveColorNodeID('vtkMRMLColorTableNodeFileColdToHotRainbow.txt')
339 # lookup table
340     heatmapVolumeDisplayNode.SetAutoThreshold(True)
341     heatmapVolumeDisplayNode.SetAutoWindowLevel(True)
342
```

Código 10.23

```
343   #------------------------------------------------------------------------------
344   def heatMapSegmentation(self, heatMap)
345     """
346     Generates heatmap segmentation and visualization.
347     :param heatMap: heatmap tensor.
348     :return: True on success, False on error.
349     """
350     # Map transformations
351     heatMap = np.array(heatMap)
352     img_array = slicer.util.arrayFromVolume(self.inputImage)[0,
353
354     # Data acquisition
355     self.numRows = img_array.shape[0]
356     self.numCols = img_array.shape[1]
357     heatMap = cv2.resize(heatMap , (self.numCols, self.numRows)).astype(np.float16)
358     max = heatMap.max()
359
360     # Threshold Segmentation
361     segmentationMask = np.logical_and(heatMap[
362
363     # Remove existing segmentation node if any
364     try
365       node = slicer.util.getNode('Segmentation')
366       slicer.mrmlScene.RemoveNode(node)
```

```
367     except
368       pass
369
370     # Create new segmentation node
371     segmentationNode = slicer.mrmlScene.AddNewNodeByClass("vtkMRMLSegmentationNode",
    "Segmentation")
372     segmentationNode.CreateDefaultDisplayNodes()
373
374     # Add segment to segmentation
375     self.addSegmentFromNumpyArray(segmentationNode, segmentationMask, 'Segmentation', (1, 0,
    0), False)
376
```

Código 10.24

```
377   #------------------------------------------------------------------------------
378   def startClassification(self, nodeName)
379     """
380     US Image Classification
381     :param nodeName: Name of the current node.
382     :return: percentage of prediction and heatmap.
383     """
384     # Node transformations
385     self.inputImage = slicer.util.getNode(nodeName)
386     img_array = slicer.util.arrayFromVolume(self.inputImage)[0,
387     if nodeName == 'Image_Reference'
388       img_array = np.rot90(np.rot90(img_array, k=1))
389     img_array = cv2.resize(img_array, (224, 224)).astype(np.float16)
390     image_trans = self.transformations(img_array).to(DEVICE).unsqueeze(0).float()
391
392     # Proving that models exist
393     print('Starting classification...')
394     if self.classificationModel is None or self.classificationModel is None
395      print("No model loaded")
396
397     # Models inference
398     with torch.no_grad()
399       pred = torch.sigmoid(self.classificationModel(image_trans))
400       feature_map = self.segmentationModel(image_trans)
401
402     # Classification prediction Transformation for interpretation
403     fractureProbability = round(pred.item()*100, 2)
404
405     # Flag Updating
406     self.classificationFlag = True
407
408     # Predicted HeatMap transformations
409     feature_map = feature_map.reshape((512, 49))
```

```
410     weight_params = list(self.segmentationModel.model.fc.parameters())[0]
411     weight = weight_params[0].detach()
412
413     # Predicted HeatMap Computing
414     heatMap = torch.matmul(weight, feature_map)
415     heatMap = heatMap.reshape(7, 7).cpu()
416     if nodeName == 'Image_Reference'
417       heatMap = np.rot90(np.rot90(heatMap, k=1))
418     # Output is the percentage of having a fracture located in the "percentage" instance and
    the computed heatmap
419     return fractureProbability, heatMap
420
```

Código 10.25

```
421   #------------------------------------------------------------------------------
      def addSegmentFromNumpyArray(self, outputSegmentation, input_np, segmentName, color,
422 visibility)
423     """
424     Adds a new segment to segmentation node from a numpy array.
425     :param outputSegmentation: segmentation node where the segment will be added
426     :param input_np: input array containing image intensity values
427     :param segmentName: name of the segment to be added
428     :param labelValue: pixel value corresponding to region of interest
429     :param inputVolume: reference volume node for segmentation
430     :param color: color to be assigned to segment for display
431     """
432     # Creation of the segment
433     emptySegment = slicer.vtkSegment()
434     emptySegment.SetName(segmentName)
435     emptySegment.SetColor(color)
436
437     # Adding the segment to the segmentation node
438     outputSegmentation.GetSegmentation().AddSegment(emptySegment)
439     segmentId = outputSegmentation.GetSegmentation().GetSegmentIdBySegmentName(segmentName)
440     outputSegmentation.GetDisplayNode().SetSegmentVisibility(segmentId, visibility)
441
442     # Update the scene
443     slicer.util.updateSegmentBinaryLabelmapFromArray(input_np, outputSegmentation,
    segmentId, self.inputImage)
444
```

Código 10.26

```
446 #------------------------------------------------------------------------------
447 #
448 # FractureClassificationTest
449 #
450 #------------------------------------------------------------------------------
451 class FractureClassificationTest(ScriptedLoadableModuleTest)
452   """This is the test case for your scripted module.
453   """
454   def runTest(self)
455     """Run as few or as many tests as needed here.
456     """
457     ScriptedLoadableModuleTest.runTest(self)
458
459
```

Código 10.27

```
460 #------------------------------------------------------------------------------
461 #
462 # FractureModel
463 #
464 #------------------------------------------------------------------------------
465 class FractureModel(nn.Module)
466     def __init__(self, weight=1)
467       super().__init__()
468
469       self.model = torchvision.models.resnet18()
470       self.model.conv1 = torch.nn.Conv2d(1, 64, kernel_size=(7, 7), stride=(2, 2),
    padding=(3, 3), bias=False)
471       self.model.fc = torch.nn.Linear(in_features=512, out_features=1)
472       self.loss = torch.nn.BCEWithLogitsLoss(torch.tensor([weight]))
473
474     def forward(self, data)
475       pred = self.model(data)
476       return pred
477
478
479 #------------------------------------------------------------------------------
480 #
481 # FractureModelMap
482 #
483 #------------------------------------------------------------------------------
484 class FractureModelMap(nn.Module)
485     def __init__(self)
486       super().__init__()
487
488       self.model = torchvision.models.resnet18()
```

```
489         self.model.conv1 = torch.nn.Conv2d(1, 64, kernel_size=(7, 7), stride=(2, 2),
    padding=(3, 3), bias=False)
490         self.model.fc = torch.nn.Linear(in_features=512, out_features=1)
491         self.feature_map = torch.nn.Sequential(*list(self.model.children())[
492
493       def forward(self, data)
494         feature_map = self.feature_map(data)
495         return feature_map
```

Código 10.28