Universidad Zaragoza
1542

Trabajo Fin de Máster

Título del trabajo: Implementación de Redes Neuronales para la detección y diferenciación de estructuras anatómicas en ecografías

English title: Implementation of Neural Networks for the detection and differentiation of Anatomical Structures in ultrasound images

Autor/es

Mario Gutiérrez López

Director/es

Mario Morales Hernández
Miguel Malo Urriés

Ponente/es

José Manuel Ramírez Rodríguez

Titulación del autor

Máster en Ingeniería Biomédica

Escuela de Ingeniería y Arquitectura, Universidad de Zaragoza

2024/2025

Implementación de Redes Neuronales para la detección y diferenciación de estructuras anatómicas en ecografías

Resumen

El objetivo de este proyecto ha sido desarrollar un sistema basado en redes neuronales convolucionales (CNN) para la detección, segmentación y diagnóstico de la severidad del síndrome del túnel carpiano (STC) a partir de imágenes ecográficas del nervio mediano. El proyecto se divide en tres fases principales, cada una enfocada en una tarea específica dentro del flujo de trabajo.

Fase de Identificación. En esta fase se ha implementado un modelo basado en inteligencia artificial para filtrar las imágenes ecográficas. Este sistema evalúa si las imágenes poseen la calidad suficiente para ser analizadas en las fases posteriores. De esta manera, se asegura que solo se procesen imágenes con características óptimas para el diagnóstico.
Fase de Segmentación. La segunda etapa del proyecto emplea una red neuronal de tipo U-Net para realizar la segmentación de las estructuras anatómicas de interés: el nervio mediano, el borde del nervio, el ligamento y el hueso semilunar. Esta segmentación precisa es fundamental para delimitar las regiones relevantes y facilitar un análisis detallado en la siguiente fase.
Fase de Diagnóstico. En esta última fase, se ha diseñado e implementado un modelo de inteligencia artificial que analiza los nervios medianos segmentados y diagnostica la severidad del síndrome del túnel carpiano. Además, el sistema identifica si el paciente no presenta dicha condición, proporcionando un resultado clínico claro y fundamentado.

Para facilitar la interacción con el sistema, se ha desarrollado una interfaz gráfica destinada al personal médico. Esta interfaz permite visualizar las imágenes ecográficas de entrada, destacar las regiones segmentadas y mostrar el diagnóstico final de manera intuitiva. Su diseño se orienta a asistir a los profesionales de la salud, brindando soporte en la toma de decisiones clínicas y mejorando la eficiencia del diagnóstico.

Los códigos de este proyecto se encuentran disponibles en GitHub bajo el enlace: https://github.com/Mario-gl7/Neuron.git

Tabla de contenidos

1. Introducción 5
1.1. Síndrome del Túnel Carpiano (STC) 5
1.2. Elementos claves de las CNN 6
1.3. Overfitting 8
1.3.1. Dropout 8
1.3.2. Aumento de datos 8
1.3.3. Regularización 8
1.4. Estructura del proyecto 9
2. Metodología 9
2.1. Funciones de activación 10
2.2. Funciones de pérdida 11
2.3. Optimizador 11
3. Fase de identificación 11
3.1. Preprocesamiento 12
3.1.1. Preprocesamiento para Red_Trans_Long (Transversal o Longitudinal) 12
3.1.2. Preprocesamiento para Red_BGF_Trans y Red_BGF_Long (Bad, Good o Fair) 12
3.1.3. Preprocesamiento para Red_BGF_Trans_bin y Red_BGF_Long_bin (Binarias) 13
3.1.4. Preprocesamiento con filtros para Red_BGF_Trans_bin y Red_BGF_Long_bin 14
3.2. Arquitectura de los modelos 15
3.2.1. Red_Trans_Long 16
3.2.2. Red_BGF_Trans & Red_BGF_Long 16
3.2.3. Red_BGF_Trans_Bin & Red_BGF_Long_Bin 17
3.2.4. Red_BGF_Trans_Bin_Filtros & Red_BGF_Long_Bin_Filtros 18
3.3. Métricas 19
4. Fase de segmentación 19
4.1. Preprocesamiento 20
4.2. Arquitectura de los modelos 22
4.3. Métricas 24
5. Fase de Diagnóstico 25

5.1. Preprocesamiento 25
5.2. Arquitectura de los modelos 27
5.2.1. Diag_trans 28
5.2.2. Diag_Long 28
5.3. Métricas 29
6. Resultados 29
6.1. Fase de Identificación 29
6.1.1. Red de clasificación transversal o longitudinal (Red_Trans_Long) 29
6.1.2. Redes de clasificación multicategórica: “Bad”, “Good” o “Fair” 31
6.1.3. Redes de clasificación binaria: “Bad” o “Good & Fair” 33
6.1.4. Redes de clasificación binaria con filtros: “Bad” o “Good” y Fair” 35
6.2. Fase de Segmentación 37
6.3. Fase de Diagnostico 39
7. Conclusiones 41
7.1. Fase de identificación 41
7.2. Fase de Segmentación 41
7.3. Fase de diagnóstico 42
8. Interfaz clínica 42
9. Bibliografía 44

Anexos

A. Anexo 1: Bibliotecas usadas

B. Anexo 2: CNN

  a. Local receptive fields

  b. Pesos y bias compartidos

  c. Capas Pooling

  d. Capa Flatten

  e. Funciones de activación

  f. Funciones de pérdida

  g. Optimizador

  h. Overfitting

C. Anexo 3: Código para la extracción de frames

D. Anexo 4: Preprocesamientos fase de identificación

  a. Preprocesamiento para Red_Trans_Long (Transversal o Longitudinal)

  b. Preprocesamiento para Red_BGF_Trans y Red_BGF_Long (Bad, Good o Fair)

  c. Preprocesamiento para Red_BGF_Trans_bin y Red_BGF_Long_bin (Binarias)

E. Anexo 5: Arquitectura de los modelos de la fase de identificación

  a. Red_Trans_Long

  b. Red_BGF_Trans & Red_BGF_Long

  c. Red_BGF_Trans_Bin & Red_BGF_Long_Bin

  d. Red_BGF_Trans_Bin_Filtros & Red_BGF_Long_Bin_Filtros

F. Anexo 6: Programación de las métricas de la fase de identificación

G. Anexo 7: Preprocesamiento de la fase de segmentación

H. Anexo 8: Resumen de los modelos y parámetros de la fase de segmentación

I. Anexo 9: Programación de las métricas de la fase de segmentación

J. Anexo 10: Resumen de los modelos y parámetros de la fase de diagnóstico

K. Bibliografía

1. Introducción

La ecografía médica es una herramienta esencial en el ámbito clínico debido a su capacidad para proporcionar imágenes en tiempo real de manera no invasiva y a bajo coste. Esta técnica sólo requiere de un ecógrafo, ya sea móvil o fijo, y de un gel conductor que se aplica en la zona a analizar. Sin embargo, un desafío inherente a este procedimiento es la interpretación de las imágenes por parte del profesional. Las imágenes ecográficas son monocromáticas, es decir, en blanco y negro, y suelen presentar ruido y bajo contraste. Esto puede dificultar la identificación de zonas anatómicas clave, como nervios o tejidos específicos, representando un reto significativo.

En este proyecto, enmarcado en la línea de análisis de imágenes médicas, se propone desarrollar un sistema basado en técnicas de *"Machine Learning"*, en particular, Redes Neuronales, para la identificación y diferenciación de estructuras anatómicas en imágenes ecográficas y su posterior diagnóstico. Este sistema tiene como objetivo servir de apoyo en la detección, reducir la carga de trabajo de los profesionales y aumentar la precisión y rapidez de los diagnósticos.

Concretamente, las imágenes ecográficas usadas para este trabajo pertenecen a la región de la muñeca. Estas imágenes han sido obtenidas de pacientes con sospecha del Síndrome del Túnel Carpiano (STC) que se sometieron a electroneurografías en el Hospital Clínico Universitario Lozano Blesa. Estas ecografías incluyen tanto casos positivos como negativos, confirmados mediante neurofisiología. Los datos han sido completamente anonimizados para garantizar la protección de la privacidad de los pacientes y han sido almacenados en una base de datos segura, permitiendo así su consulta y utilización para la realización del proyecto.

Para la identificación de estructuras anatómicas específicas del túnel carpiano, como son el nervio mediano y los huesos del carpo se han utilizado Redes Neuronales Convolucionales (CNN) y una Red de Segmentación U-Net. Todo ello se ha programado en el lenguaje Python 3.11, con el uso de las bibliotecas Tensorflow 2.15, Sklearn, Seaborn, Matplotlib. La lista completa de bibliotecas se encuentra en el Anexo 1. Gracias al clúster del I3A, (Instituto de Investigación de Aragón), se ha podido usar una GPU3090 con un entorno Cuda 12.1 y CudNN 8.9 para la ejecución de los códigos. Este trabajo ha sido parcialmente financiado por el Programa de Becas y Ayudas del Instituto de Investigación en Ingeniería de Aragón (I3A).

Estos modelos predictivos desarrollados permiten un sistema de soporte al diagnóstico que asista a los médicos en la confirmación de la presencia del STC. A largo plazo se busca crear una metodología generalizable que pueda aplicarse a otras áreas anatómicas y patológicas.

1.1. Síndrome del Túnel Carpiano (STC)

El síndrome del túnel carpiano (STC) es la neuropatía por compresión más frecuente y supone elevados costes socioeconómicos. La etiología del STC puede estar relacionada con el trabajo, el estilo de vida, la predisposición genética o las lesiones. [1] El movimiento repetitivo se considera la causa más común del síndrome del túnel carpiano

(STC), ciertas enfermedades específicas, como la diabetes, la obesidad mórbida y situaciones fisiológicas como el embarazo, pueden estar asociadas con un mayor riesgo de desarrollar STC. Los síntomas comunes de la compresión del nervio mediano en la muñeca incluyen entumecimiento, hormigueo, parestesias y/o dolor neuropático de hormigueos y parestesias. Los pacientes a menudo informan que se les caen con frecuencia objetos sostenidos en la(s) mano(s) afectada(s) y tienen dificultad con tareas que requieren coordinación fina, como manipular botones en la ropa. [2]

Cuando existe incertidumbre en el diagnóstico o cuando es necesaria una medida objetiva para evaluar intervenciones invasivas como la cirugía, se requiere usar pruebas como electrodiagnósticos (EDX), estudios de conducción nerviosa (NCS) y electromiografía (EMG). [1] El primer paso en el manejo del STC consiste en determinar si el tratamiento debe ser quirúrgico o no. Para síntomas leves y discontinuos, se recomiendan medidas no quirúrgicas. No obstante, ante síntomas moderados a graves, se realizan pruebas adicionales como estudios de conducción nerviosa (NCS) o electromiografía con aguja (EMG) para evaluar si el STC es agudo o crónico y si hay evidencia significativa de daño axonal.

El canal del carpo está delimitado internamente por el pisiforme y la apófisis unciforme del hueso ganchoso. El suelo del canal está formado por los huesos de la primera fila del carpo. La pared externa del canal está constituida por el tubérculo del escafoides y el trapecio. A modo de puente, se encuentra el retináculo flexor, cuya parte central mide entre 2 y 4 mm de grosor, conocido como el ligamento carpiano transverso (LCT). [3] El túnel carpiano comienza en la superficie volar del pliegue de la muñeca y protege el nervio mediano y los nueve flexores de los dedos. El nervio mediano, la estructura más superficial del túnel, pasa por debajo del retináculo flexor antes de ramificarse en la palma hacia los músculos tenares y los dos lumbricales laterales. Finalmente, el nervio termina como tres nervios palmares comunes que se dividen en nervios digitales cutáneos que inervan la superficie volar del índice, el dedo medio y la mitad radial del anular. [1]

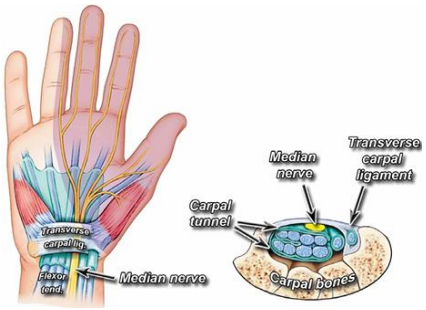


Fig. 1 Anatomía del túnel carpiano [1]

1.2. Elementos claves de las CNN

En el anexo 2 se encuentra detalladamente las siguientes partes pertenecientes a las CNN, en esta sección se realizará una explicación general para el entendimiento de los códigos del proyecto.

El auge de las técnicas de *"Machine Learning"* y, en particular, de *"deep learning"*, ha abierto nuevas posibilidades para el análisis automatizado de imágenes médicas. *"Machine Learning"* es una rama de la Inteligencia Artificial, que se basa en la

programación de algoritmos capaces de aprender de los inputs proporcionados. Estos algoritmos se conocen como redes neuronales, inspiradas en las neuronas biológicas [4]. Actualmente existen muchos tipos de redes neuronales, en este proyecto se van a utilizar las redes neuronales convolucionales (CNN), una variante que ha demostrado ser altamente efectiva en tareas complejas de procesamiento de imágenes biomédicas. En estas redes, los pesos determinan la importancia de cada entrada y el bias ajusta el resultado antes de aplicar una función de activación, la cual define la salida de cada neurona [4].
El objetivo de estos algoritmos es el autoaprendizaje, es decir, que a medida que se les administren inputs, determinen los pesos y bias correspondientes. [25] El cálculo de los pesos y bias se realiza mediante la función de pérdida o *"loss function",* que evalúa la precisión del modelo, y el algoritmo de *"backpropagation"*, que ajusta estos parámetros usando gradientes y técnicas de optimización. Durante el entrenamiento, pueden surgir problemas como el *"overfitting*" o la elección de hiperparámetros, que se abordan con técnicas como la regularización y el ajuste de la tasa de aprendizaje. [5]

Las CNN, debido a su estructura, consideran la distribución espacial de las imágenes, lo que resulta en un aprendizaje rápido [5]. Esto permite entrenar redes más profundas y con más capas, siendo especialmente eficaces para el reconocimiento de patrones en imágenes.
Las redes convolucionales usan una arquitectura especial basadas en tres ideas: *"local receptive fields"*, pesos y bias compartidos y *"pooling"*.

Local receptive fields
En una red convolucional las neuronas se dibujan como matrices de la forma Ancho x Alto de la misma forma que las imágenes de entrada, donde a cada píxel se le asigna una neurona. Una región de neuronas de la capa de entrada, conocido como kernel o *"Local receptive field"* recorre toda la imagen de entrada generando un mapa de características. [6]
Para cada "Local receptive field" hay una neurona oculta asignada en la primera capa oculta, Fig. 2.

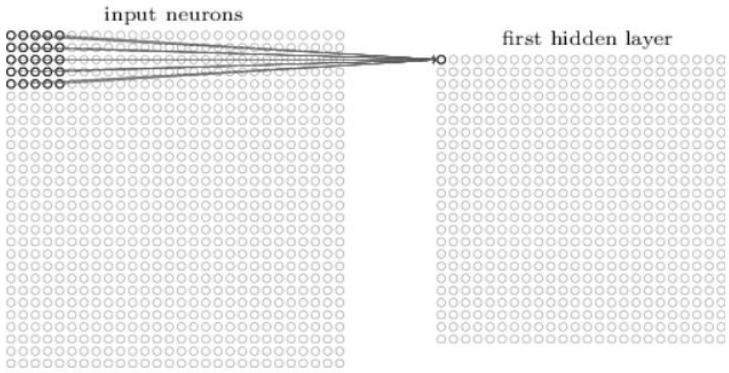


Fig. 2 Local receptive field [6]

Pesos y bias compartidos

Para capturar relaciones espaciales y aprender patrones, un kernel deslizante recorre la imagen aplicando el mismo peso en cada posición del mapa, reduciendo parámetros y permitiendo detectar patrones como bordes o texturas de forma consistente, independientemente de su ubicación [6].

Capas Pooling y flatten

Estas capas se encargan de simplificar la información de las capas convolucionales reduciendo la dimensionalidad espacial de las mismas (Fig. 3). En este proyecto se ha usado la capa *"Maxpooling"*.

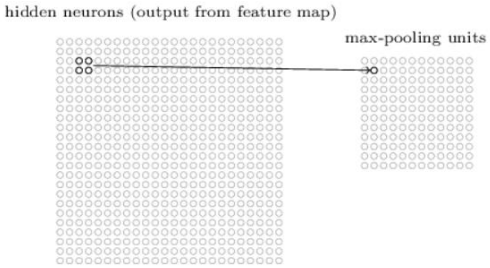


Fig. 3 Capa MaxPooling [6]

Como último paso, la capa "flatten" convierte un tensor tridimensional en un vector unidimensional. Sirve de conexión entre las capas convolucionales y las capas densas.

1.3. Overfitting

El *"overfitting"* o sobreajuste, es un comportamiento que ocurre cuando el modelo CNN está tan estrechamente alineado con los datos del conjunto de entrenamiento, que no es capaz de generalizar, es decir, se aprende de "memoria" los datos de entrenamiento y a la hora de validar, al no haber entrenado con esos datos, no es capaz de clasificarlos correctamente. [7]

1.3.1. Dropout

Esta técnica para reducir el sobreajuste conocida como *"dropout"* consiste en "apagar" aleatoriamente un porcentaje de las neuronas de una capa, de esta forma se previene que las neuronas se vuelvan dependientes de las salidas de las anteriores, fomentando así una red más robusta y con capacidad de generalizar. [8] Esto se aplica únicamente durante el entrenamiento.

1.3.2. Aumento de datos

Esta técnica consiste en ampliar artificialmente el conjunto de datos, aplicando de forma aleatoria transformaciones, rotaciones, volteos, etc.
Este aumento de datos únicamente se aplica al conjunto de entrenamiento para conseguir que el modelo aprenda a generalizar.

1.3.3. Regularización

En este proyecto se ha usado la regularización L2. Su función es agregar un término adicional a la función matemática de la pérdida, o *"loss function"*, y así fomentar pesos

más pequeños y uniformidad para ayudar al modelo a converger antes y reducir la función de pérdida.

1.4. Estructura del proyecto

El presente proyecto va a estar estructurado en cinco partes. Una parte de metodología donde se explicará en detalle la extracción del conjunto de datos, la obtención de las regiones de interés (ROIs) y configuraciones comunes que comparten todos los códigos que se han desarrollado. Posteriormente se explicarán las fases principales del proyecto, estructuradas en preprocesamiento, arquitectura de los modelos y métricas usadas, donde la primera fase, conocida como fase de identificación, desarrollará una serie de códigos en los que la inteligencia artificial se pondrá a prueba para clasificar la calidad de las imágenes. La segunda fase, conocida como fase de segmentación, explicará la red usada para segmentar las zonas anatómicas de interés presentadas en las imágenes con ROIs. Por último, la fase de diagnóstico se encargará de exponer los códigos para el diagnóstico del nervio mediano. Finalmente, el proyecto presentará un apartado de interfaz gráfica donde se mostrará con ejemplos visuales, los resultados obtenidos de cada fase aplicados al servicio clínico.

2. Metodología

Las fases de Identificación, Segmentación y Diagnóstico cubren desde un filtrado de imágenes para discriminar ecografías validas de las no validas, hasta una segmentación de diferentes partes anatómicas para un posterior diagnóstico, basándose en la forma, intensidad de píxeles y regiones tisulares.

Para la obtención de la base de datos se han utilizado videos ecográficos de la región de la muñeca tanto derecha como izquierda de 52 pacientes con sospecha del Síndrome del Túnel Carpiano (STC). Estos videos ecográficos se han obtenido con la colaboración del Hospital Clínico Universitario Lozano Blesa y abarcan casos tanto positivos como negativos, confirmados a través de estudios neurofisiológicos, además, se cuentan con dos tipos de vistas ecográficas, transversal y longitudinal. Todos los datos han sido debidamente anonimizados para garantizar la privacidad de los pacientes.

De los videos ecográficos se han extraído 5 frames/segundo con un simple código en Python (Anexo 3), obteniendo un total de 11.895 imágenes ecográficas. Sin embargo, estas no son el total de imágenes con las que se ha contado para la realización del proyecto.

Con la herramienta online UZ qTool [9][24], nuestro médico experto en Fisiología, ha sido capaz de, primeramente, de imágenes aleatorias tanto transversales como longitudinales del conjunto de datos total, discriminar entre imágenes *"Bad", "Good" y "Fair"* basándose en la información propia de la imagen y seguidamente, de las imágenes clasificadas como *"Good" y "Fair",* identificar las Rois, por lo que finalmente se ha contado con un lote de datos de 1360 imágenes ecográficas transversales y 1158 imágenes ecográficas longitudinales.

La principal diferencia entre las imágenes transversales (Trans) y las imágenes longitudinales (Long) es la posición del ecógrafo a la hora de inspeccionar la región.

Para las imágenes transversales (Trans), el ecógrafo se coloca de forma perpendicular al antebrazo obteniendo una imagen clara del nervio mediano, los huesos escafoides, semilunar y pisiforme, además del ligamento del carpo que cubre el nervio mediano. Con la herramienta online, las ROIs se han identificado con diferentes colores, en Fig. 4 se observa el nervio mediano en azul, el borde del nervio en fucsia rodeando al nervio, el ligamento del carpo en rosa claro por encima del nervio y el hueso semilunar en amarillo.

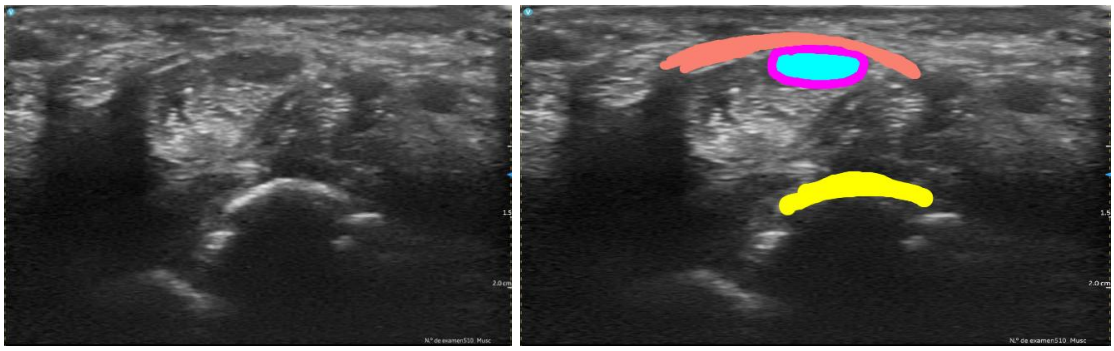


Fig. 4 a) Imagen ecográfica transversal, b) Imagen ecográfica transversal con ROI

Las imágenes longitudinales (Long) se extraen colocando el ecógrafo de forma paralela al antebrazo, de modo que se obtiene una región longitudinal del nervio. Para estas imágenes no se tiene en cuenta del ligamento del Carpo, de modo que en Fig. 5 encontramos el nervio Mediano en azul, el borde del nervio en fucsia por encima y debajo de este, y en amarillo el hueso semilunar.

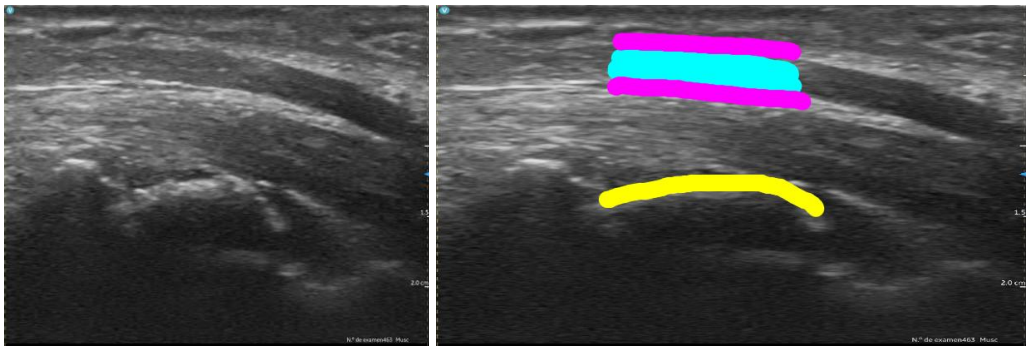


Fig. 5 a) Imagen ecográfica longitudinal, b) Imagen ecográfica longitudinal con ROI

Para cada imagen transversal y longitudinal se cuenta con un archivo json donde se encuentra referenciada cada imagen original con su imagen con ROI.
A continuación, se explicarán las características comunes de las redes usadas a lo largo de todos los códigos, se encuentra una explicación más detallada en el anexo 2.

2.1. Funciones de activación

Una función de activación es una función matemática que se la aplica a la salida de una neurona, es decir, estas funcionen deciden si una neurona se activa o no. [10]
Existen muchas funciones de activación como Sigmoid, Tanh y muchas combinaciones con funciones de pérdida y optimizadores. Por ello, se ha elegido para este proyecto una consistencia en el uso de estas funciones, siendo así las que se han utilizado: Función de activación ReLu, Sigmoid y Softmax.

2.2. Funciones de pérdida

Las funciones de pérdida miden en qué medida las predicciones que realiza el modelo coinciden con los resultados reales, es decir, proporcionan una métrica cuantitativa de la precisión de las predicciones que realiza el modelo. Estas funciones se usan para guiar el entrenamiento y a los algoritmos de optimización para ajustar los parámetros con el fin de reducir estas funciones. [11]
A lo largo del proyecto se van a usar las funciones: *"Binary Cross Entropy"*, y *"Categorical Cross Entropy"*.

2.3. Optimizador

Los optimizadores son algoritmos que ajustan los parámetros durante el entrenamiento con el objetivo de minimizar la función de pérdida. Mejoran el rendimiento del modelo al determinar la combinación de pesos y bias de la red que darán las mejores predicciones. El optimizador usado en este proyecto es Adam. [12]

A continuación, las diferentes fases del proyecto van a desarrollar el preprocesamiento realizado para la obtención de imágenes, su tratamiento, los modelos de redes neuronales ya optimizados y las métricas usadas para evaluar el rendimiento de los modelos.
Para poder realizar una comparación justa entre los entrenamientos y llegar a un modelo optimizado es necesario que exista repetibilidad en los códigos, de esta forma se pueden validar los resultados e identificar errores, para ello, a lo largo de todos los códigos se ha fijado la semilla *"seed=42"*. Esto no es más que una referencia para que los algoritmos que generan números aleatorios, como la inicialización de pesos o la división de datos, produzcan siempre la misma secuencia de números.

3. Fase de identificación

La primera fase del proyecto se conoce como identificación. En esta sección se ha desarrollado una red neuronal para diferenciar entre imágenes transversales y longitudinales, que se conoce como "Red_Trans_Long", una red para categorizar el tipo de imagen en las transversales, "Red_BGF_Trans", y otra red con el mismo objetivo para las longitudinales, "Red_BGF_Long", estas dos últimas redes se han entrenado con su banco de datos propio para que pueda clasificar si una imagen es válida o no para pasar a la siguiente fase del proyecto.

Las imágenes están categorizadas como *"Bad", "Good" y "Fair",* es decir, se trata de una clasificación categórica. Para saber cuál es la correspondiente a cada imagen, nuestro experto en fisiología se ha basado en el tipo de información que nos ofrecen las imágenes. Las calificadas como *"Good",* son ecografías en las que se puede ver sin dificultad, el nervio mediano, el ligamento del carpo y el hueso semilunar, utilizado como referencia para localizar el nervio. Las imágenes *"Bad",* son aquellas que o bien se encuentran borrosas, o no se encuentran aún en la zona de la muñeca, pues al obtenerlas de un video, algunas imágenes se localizan en el antebrazo o no se puede identificar en ellas el nervio correctamente. Por consiguiente, las fotografías correspondientes a la última categoría

son aquellas que se encuentran en la zona correcta, pero existe alguna dificultad para visualizar las zonas anatómicas claves que por el contrario se aprecian en las *"Good"*.
Este procedimiento de evaluación es subjetivo de cada profesional médico, pudiendo existir diferencias entre distintos expertos para discriminar entre imágenes *"Good"* y *"Fair"* así como un mismo profesional, en este caso nuestro experto, puede clasificar de forma distinta una misma imagen.

3.1. Preprocesamiento

Se organizaron en tres carpetas: imágenes originales, imágenes transversales y longitudinales con ROI pintadas, Fig. 4 y Fig. 5. Un archivo JSON vincula los nombres de las imágenes originales con las ROIs, su categoría *("Bad", "Good" o "Fair")* y su tipo ("Transversal" o "Longitudinal").
Para facilitar la lectura de los códigos a lo largo del proyecto y entre los diferentes códigos, se les ha asignado el mismo nombre a las variables. Sin embargo, en ningún momento los códigos se entrelazan, de forma que cada código cuenta son su propia información dentro de las variables pese a tener el mismo nombre.
Se puede encontrar una descripción más detallada de los preprocesamientos en el anexo 4, a continuación, se expondrá un resumen con las ideas clave.

3.1.1. Preprocesamiento para Red_Trans_Long (Transversal o Longitudinal)

Esta red se va a encargar de aprender a calificar si una imagen es transversal o longitudinal, de modo que, para guardar las etiquetas correspondientes a cada imagen, se ha creado un diccionario binario, como se ve en Fig. 6. donde se le otorga a cada categoría de la imagen un valor, de forma que las etiquetas son binarias.

```
# Inicializar listas para las imágenes y etiquetas
X_imagenes = []
Y_etiquetas = []

# Definir el mapeo de etiquetas a índices
etiquetas_unicas = {'transversal': 0, 'longitudinal': 1}
```

Fig. 6 Código de variables y etiquetas

Las imágenes originales transversales y longitudinales son de dimensiones 984 x 696 píxeles. Para garantizar uniformidad total de dimensiones y optimización, se han escalado a 256 x 256 píxeles, se ha añadido un canal extra (escala de grises) para garantizar la compatibilidad con las capas de convolución y se han normalizado los píxeles para mejorar la estabilidad, convergencia y uniformidad. De modo que el conjunto de datos final tiene la forma (2518, 256, 256, 1) con los píxeles de valor [0 1].

3.1.2. Preprocesamiento para Red_BGF_Trans y Red_BGF_Long (Bad, Good o Fair)

Para estas dos redes, en Red_BGF_Trans se han cargado únicamente las imágenes transversales y en Red_BGF_Long se han cargado las longitudinales.

Estas nuevas redes cuentan con tres categorías: *"Bad", "Good" y "Fair".* De modo que la variable de etiquetas y el diccionario han de ser modificado a las correspondientes categorías asignándole un valor a cada categoría para almacenarlo, Fig. 7.

```python
# Inicializar listas para las imágenes y etiquetas
X_trans = []
Y_etiquetas = []

# Definir el mapeo de etiquetas a índices
etiquetas_unicas = {'bad': 0, 'good': 1, 'fair': 2}
```

```python
# Inicializar listas para las imágenes y etiquetas
X_long = []
Y_etiquetas = []

# Definir el mapeo de etiquetas a índices
etiquetas_unicas = {'bad': 0, 'good': 1, 'fair': 2}
```

Fig. 7 Etiquetas para código transversal y longitudinal

Las imágenes se procesan en escala de grises, se ajustan a 256x256 píxeles, se les agrega un canal extra y se normalizan en el rango [0, 1]. Esto genera un lote de datos de (1360, 256, 256, 1) para Red_BGF_Trans y (1158, 256, 256, 1) para Red_BGF_Long.
Estas dos redes cuentan con 3 categorías, por ello es importante manejar correctamente las etiquetas o categorías de cada imagen. Dado que ya no es un tensor binario, es necesario realizar lo que se conoce como *"One-Hot encoding"*, Fig. 8. [13]

```python
Y_etiquetas = to_categorical(Y_etiquetas, num_classes=len(etiquetas_unicas))
```

Fig. 8 Función de "One-Hot Encoding"

"One-Hot encoding" convierte las etiquetas categóricas en un formato interpretable por las redes neuronales. Esta función le asigna un valor binario a cada variable del diccionario de categorías, así, la categoría *"Bad":* 0 se convierte en [1, 0, 0], *"Good":* 1 se convierte en [0, 1, 0] y *"Fair"*: 2 se convierte en [0, 0, 1].
Esta forma de encriptar las etiquetas evita que las redes interpreten relaciones ordinales, de orden o magnitud entre las diferentes categorías y garantiza compatibilidad con las funciones de pérdida.

3.1.3. Preprocesamiento para Red_BGF_Trans_bin y Red_BGF_Long_bin (Binarias)

Un problema de las categorías múltiple es que se incrementan las probabilidades de fallo para la predicción de las redes. Cuantas más opciones de resultados haya, más cálculos van a ser necesarios, menor eficiencia computacional, mayor riesgo de confusión entre clases y menor precisión. Por ello, las redes BGF_Trans y BGF_Long se han programado a parte con categorías binarias para que aprendan a distinguir entre fotos *"Bad" y "Good".*

Dado que las imágenes *"Fair"* son imágenes válidas para su estudio, se ha considerado asignarle el mismo valor categórico en el diccionario que a las "Good", Fig. 9.

```python
# Inicializar listas para las imágenes y etiquetas
X_imagenes = []
Y_etiquetas = []

# Definir el mapeo de etiquetas a índices
etiquetas_unicas = {'bad': 0, 'good': 1, 'fair': 1}
```

Fig. 9 Etiquetas código binario

Red_BGF_Trans_bin: (1360, 256, 256, 1) y Red_BGF_Long_bin: (1158, 256, 256, 1) ambos con píxeles [0 1].

3.1.4. Preprocesamiento con filtros para Red_BGF_Trans_bin y Red_BGF_Long_bin

Para ayudar a las redes binarias a aprender y calificar correctamente, se han realizado dos nuevos códigos, iguales a Red_BGF_Trans_bin y Red_BGF_Long_bin, con el mismo procesamiento de imágenes con lotes: Red_BGF_Trans_bin_filtros: (1360, 256, 256, 1) y Red_BGF_Long_bin_filtros: (1158, 256, 256, 1) ambos con píxeles [0 1].

A estos dos nuevos códigos se les ha aplicado una serie de filtros para resaltar las zonas anatómicas claves a la vez que reducir el ruido propio de las imágenes ecográficas, Fig. 10. [26]
Primero se aplica al conjunto de imágenes un filtro de contraste adaptativo (Clahe) [14], esta técnica mejora el contraste dado que se ajusta de forma dinámica a las características locales de cada imagen además de considerar variaciones de contraste en diferentes regiones dentro de una misma foto. No todas las fotos cuentan con la misma intensidad de píxeles, brillo, ruido o contraste entre zonas, mediante la aplicación de este filtro se consigue aplicar un contraste entre las diferentes zonas claves para mejorar los detalles en regiones oscuras o claras. Dado que nos interesa la región oscura del nervio, se han invertido los blancos y negros para trabajar mejor con la zona del nervio. Para este filtro se ha elegido un *"clip_limit"* de 2.0 y un tamaño de malla de 8x8 para el contraste local de regiones, es decir, que realiza el contraste local en una región de 8x8 píxeles.
El segundo filtro que se aplica es un filtro mediano, esta técnica reduce el ruido a la vez que preserva los bordes de las zonas donde hay un cambio en la intensidad de los píxeles. Este filtro es especialmente útil para eliminar el ruido conocido como "Sal y pimienta" que son cambios abruptos en la intensidad de píxeles que se observan como puntos brillantes en regiones oscuras y viceversa. Este filtro usa una malla conocida como kernel, de 3x3, donde se estudia el valor de intensidad del píxel central y los vecinos, se ordena los valores en orden ascendente y se calcula la mediana de los valores. El valor central se sustituye por la mediana calculada. Esto se aplica a cada píxel de la imagen procesando los kernel de forma deslizante. [15]
El tercer filtro a aplicar es una umbralización otsu, este filtro convierte una imagen en escala de grises en una imagen binaria de blancos y negros calculando un umbral óptimo para cada imagen ecográfica. Primero realiza un histograma de la imagen donde se estudia la frecuencia de cada nivel de intensidad, luego se dividen las intensidades en dos clases y se busca minimizar la varianza entre las clases calculando un umbral óptimo. Con esta técnica se intenta segmentar o resaltar el nervio y hueso en cada imagen. [16]
Finalmente se realiza una inversión de blancos y negros para devolver cada zona clave, nervio y hueso a su color correspondiente.
Con esta serie de filtros se consiguen imágenes binarias donde se encuentran segmentados y diferenciados el nervio mediano y hueso semilunar.
Como paso final, se ha multiplicado la imagen filtrada por la imagen original, de esta forma las zonas oscuras, calificadas como 0 en la imagen filtrada debido a la binarización

Otsu, oscurecen o apagan los píxeles oscuros de la imagen original al estar multiplicado su valor por 0, mientras que los píxeles claros de la imagen original se ven intactos al estar multiplicados por 1. Se consigue así una imagen ecográfica con regiones bien diferenciadas, Fig. 11.

```python
# Filtro de Contraste Adaptativo (CLAHE)
def filtro_contraste_adaptativo(X_trans, clip_limit=2.0, tile_grid_size=(8, 8)):
    clahe = cv2.createCLAHE(clipLimit=clip_limit, tileGridSize=tile_grid_size)
    X_trans = np.array([clahe.apply(np.uint8(img * 255)) for img in X_trans])
    return X_trans[..., np.newaxis]

# Filtrado Mediano
def aplicar_filtro_mediano(X_trans, kernel_size=3):
    X_trans = np.array([cv2.medianBlur(img, kernel_size) for img in X_trans])
    return X_trans[..., np.newaxis]

# Umbralización de Otsu
def umbral_otsu(imagen):
    imagen_uint8 = np.uint8(imagen * 255) if imagen.dtype != np.uint8 else imagen
    _, imagen_umbralizada = cv2.threshold(imagen_uint8, 0, 255, cv2.THRESH_BINARY + cv2.THRESH_OTSU)
    return imagen_umbralizada
```

```python
# Aplicar CLAHE
X_contraste = filtro_contraste_adaptativo(X_imagenes)
X_contraste = X_contraste.reshape(X_contraste.shape[0], 256, 256)
print('Shape X_contraste: ', X_contraste.shape, X_contraste.dtype)
print('Shape X_contraste: ', np.unique(X_contraste))

X_mediano = aplicar_filtro_mediano(X_contraste)
X_mediano = X_mediano.reshape(X_mediano.shape[0], 256, 256)
print('Shape X_mediano: ', X_mediano.shape, X_mediano.dtype)
print('Shape X_mediano: ', np.unique(X_mediano))

# Aplicar el método de Otsu a las imágenes
X_otsu = np.array([umbral_otsu(img) for img in X_mediano])
X_otsu = X_otsu.reshape(X_otsu.shape[0], 256, 256, 1)
print('Shape X_otsu: ', X_otsu.shape, X_otsu.dtype)
print('Shape X_otsu: ', np.unique(X_otsu))

# Invertir las imágenes binarizadas
X_inv = np.array([cv2.bitwise_not(img) for img in X_otsu])
X_inv = X_inv.reshape(X_inv.shape[0], 256, 256, 1)
print('Shape X_inv: ', X_inv.shape, X_inv.dtype)
print('Shape X_inv: ', np.unique(X_inv))

X_final = X_imagenes * X_inv
print('Shape X_final: ', X_final.shape, X_final.dtype)
print('Shape X_final: ', np.unique(X_final))
```

Fig. 10 Código de filtros e implementación

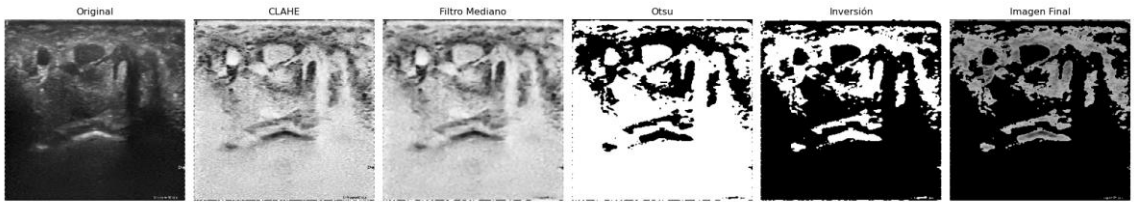


Fig. 11 Ejemplo de progresión de filtros, a) Original, b) CLAHE, c) Mediano, d) Otsu, e) Inversión, f) Imagen final

Resumen de Códigos

Tabla 1 Resumen de Códigos de la fase de Identificación

Categorías	***Transversal + Longitudinal***	***Transversal***	***Longitudinal***
Multicategórico	-	Red_BGF_Trans	Red_BGF_Long
Binario	Red_Trans_Long	Red_BGF_Trans_Bin	Red_BGF_Long_Bin
Binario + Filtros	-	BGF_Trans_Bin_Filtros	BGF_Long_Bin_Filtros

3.2. Arquitectura de los modelos

La arquitectura de cada modelo se encuentra detallada en el anexo 5. Sin embargo, en esta sección se incluye un resumen de los modelos.

3.2.1. Red_Trans_Long

Se dividieron los datos en 80% para entrenamiento y 20% para validación y se aplicó un aumento de datos moderado para evitar sobreajuste, se recuerda que únicamente se aplica al entrenamiento. El modelo comienza con 32 capas convolucionales de 3x3 para captar características generales, seguidas de *"BatchNormalization"* para estabilizar el entrenamiento y *"MaxPooling2D"* para reducir dimensiones, hasta las 128 capas, pasando de 256x256 a 32x32 píxeles. La capa *"Flatten"* convierte la imagen en un vector unidimensional que alimenta capas densas, iniciando con 256 neuronas y un *"Dropout"* de 0.4. La última capa con una neurona y activación *"Sigmoid"* (clasificación binaria), clasifica imágenes como transversales o longitudinales, utilizando *"Binary_CrossEntropy"* (clasificación binaria) como función de pérdida, Fig. 12.

```
modelo = tf.keras.Sequential([

    tf.keras.layers.Conv2D(32, (3, 3), activation='relu', padding='same', kernel_initializer='he_normal', input_shape=
    (256, 256, 1)),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.MaxPooling2D(2,2),

    tf.keras.layers.Conv2D(64, (3, 3), activation='relu',  padding='same', kernel_initializer='he_normal'),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.MaxPooling2D(2,2),

    tf.keras.layers.Conv2D(128, (3, 3), activation='relu',  padding='same', kernel_initializer='he_normal'),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.MaxPooling2D(2,2),

    tf.keras.layers.Flatten(),

    tf.keras.layers.Dense(256, activation='relu' , kernel_initializer='he_normal'),
    tf.keras.layers.Dropout(0.4),
    tf.keras.layers.Dense(1, activation = 'sigmoid')
])

modelo.compile(
    optimizer = 'adam',
    loss = 'binary_crossentropy',
    metrics=['accuracy']
)
```

Fig. 12 Arquitectura modelo Red_Trans_Long

El tamaño de lote utilizado es de 15 imágenes, y se ha entrenado durante 100 épocas. En cada época, el conjunto de entrenamiento se recorre, recalculando los pesos y bias cada 15 imágenes.

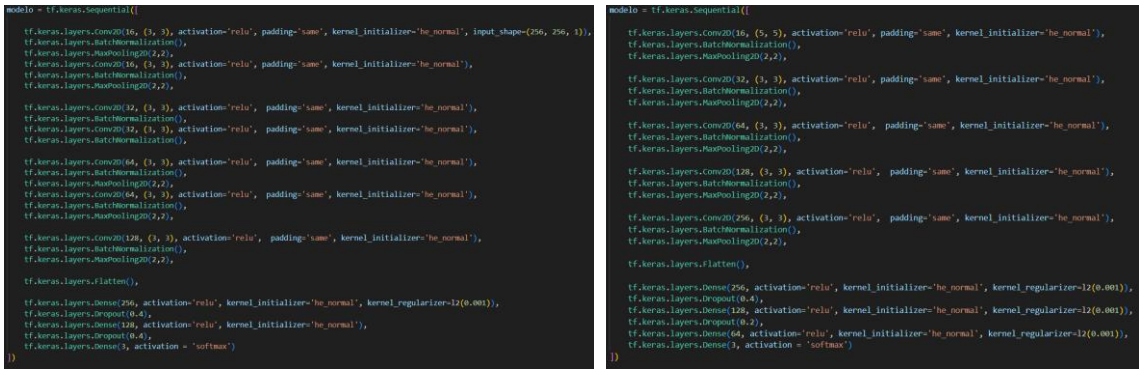
3.2.2. Red_BGF_Trans & Red_BGF_Long

Con un conjunto de datos de (1360, 256, 256, 1) para las imágenes transversales y (1158, 256, 256, 1) para las longitudinales se ha decidido por una división del 75% para el entrenamiento y 25% para la validación.

Para la red de imágenes transversales, fig. 13 a), se ha empleado una extracción de características profunda, comenzando con dos capas escalonadas de 16 filtros con activación *"relu"* y *"he_normal"*, para mejorar la capacidad de aprendizaje y reducir los parámetros. Las capas escalonadas alcanzan 64 filtros, con una capa final de 128, seguida de una capa densa con 256 neuronas de entrada, activación *"relu"*, regularización L2 y una capa oculta de 128 neuronas. La capa de salida tiene 3 neuronas para las 3 categorías,

con activación *"softmax"*. El tamaño de lote es de 25 unidades y se entrena durante 150 épocas.

Para el modelo longitudinal, Fig. 13 b), no se ha utilizado una estructura escalonada, sino que se ha comenzado con 16 filtros de tamaño 5x5, activación *"relu"* y *"he_normal"*, y se ha incrementado hasta 256 filtros. En las capas densas, con activación *"relu"* y *"he_normal"*, la entrada tiene 256 neuronas, la salida tiene 3 neuronas y las capas ocultas tienen 128 y 64 neuronas. Se ha aplicado un *"dropout"* de 0.4 en la primera capa y 0.2 en las ocultas, con regularización L2 en todas las capas densas y una activación *"softmax"* en la salida.



```
modelo = tf.keras.Sequential([

    tf.keras.layers.Conv2D(16, (3, 3), activation='relu', padding='same', kernel_initializer='he_normal', input_shape=(256, 256, 1)),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.MaxPooling2D(2,2),
    tf.keras.layers.Conv2D(16, (3, 3), activation='relu', padding='same', kernel_initializer='he_normal'),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.MaxPooling2D(2,2),

    tf.keras.layers.Conv2D(32, (3, 3), activation='relu',  padding='same', kernel_initializer='he_normal'),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Conv2D(32, (3, 3), activation='relu',  padding='same', kernel_initializer='he_normal'),
    tf.keras.layers.BatchNormalization(),

    tf.keras.layers.Conv2D(64, (3, 3), activation='relu',  padding='same', kernel_initializer='he_normal'),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.MaxPooling2D(2,2),
    tf.keras.layers.Conv2D(64, (3, 3), activation='relu',  padding='same', kernel_initializer='he_normal'),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.MaxPooling2D(2,2),

    tf.keras.layers.Conv2D(128, (3, 3), activation='relu',  padding='same', kernel_initializer='he_normal'),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.MaxPooling2D(2,2),

    tf.keras.layers.Flatten(),

    tf.keras.layers.Dense(256, activation='relu', kernel_initializer='he_normal', kernel_regularizer=l2(0.001)),
    tf.keras.layers.Dropout(0.4),
    tf.keras.layers.Dense(128, activation='relu', kernel_initializer='he_normal'),
    tf.keras.layers.Dropout(0.4),
    tf.keras.layers.Dense(3, activation = 'softmax')
])
```

```
modelo = tf.keras.Sequential([

    tf.keras.layers.Conv2D(16, (5, 5), activation='relu', padding='same', kernel_initializer='he_normal'),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.MaxPooling2D(2,2),

    tf.keras.layers.Conv2D(32, (3, 3), activation='relu', padding='same', kernel_initializer='he_normal'),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.MaxPooling2D(2,2),

    tf.keras.layers.Conv2D(64, (3, 3), activation='relu',  padding='same', kernel_initializer='he_normal'),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.MaxPooling2D(2,2),

    tf.keras.layers.Conv2D(128, (3, 3), activation='relu',  padding='same', kernel_initializer='he_normal'),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.MaxPooling2D(2,2),

    tf.keras.layers.Conv2D(256, (3, 3), activation='relu',  padding='same', kernel_initializer='he_normal'),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.MaxPooling2D(2,2),

    tf.keras.layers.Flatten(),

    tf.keras.layers.Dense(256, activation='relu', kernel_initializer='he_normal', kernel_regularizer=l2(0.001)),
    tf.keras.layers.Dropout(0.4),
    tf.keras.layers.Dense(128, activation='relu', kernel_initializer='he_normal', kernel_regularizer=l2(0.001)),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Dense(64, activation='relu', kernel_initializer='he_normal', kernel_regularizer=l2(0.001)),
    tf.keras.layers.Dense(3, activation = 'softmax')
])
```

Fig. 13 a) Arquitectura Red_BGF_Trans, b) Arquitectura Red_BGF_Long

3.2.3. Red_BGF_Trans_Bin & Red_BGF_Long_Bin

Se dividieron los datos en un 75% para entrenamiento y un 25% para validación en ambos modelos. Se optó por capas convolucionales iniciales más grandes para capturar mejor las relaciones espaciales en las imágenes, comenzando con 8 filtros de kernel 7x7 y luego 16 capas de kernel 5x5, seguidas de *"MaxPooling"*. La última capa convolucional tiene 64 filtros, y las capas densas comienzan con 128 neuronas, con intermedias de 64 y 16, y finalizan con 1 neurona para la salida binaria, con un *"dropout"* del 0.3 para evitar sobreajuste.

```
modelo = tf.keras.Sequential([

    tf.keras.layers.Conv2D(8, (7, 7), activation='relu', padding='same',
                           kernel_initializer='he_normal'),
    tf.keras.layers.MaxPooling2D(2,2),

    tf.keras.layers.Conv2D(16, (5, 5), activation='relu', padding='same',
                           kernel_initializer='he_normal'),
    tf.keras.layers.MaxPooling2D(2,2),

    tf.keras.layers.Conv2D(32, (3, 3), activation='relu', padding='same',
                           kernel_initializer='he_normal'),
    tf.keras.layers.MaxPooling2D(2,2),
    tf.keras.layers.Conv2D(64, (3, 3), activation='relu', padding='same',
                           kernel_initializer='he_normal'),
    tf.keras.layers.MaxPooling2D(2,2),

    tf.keras.layers.Flatten(),

    tf.keras.layers.Dense(128, activation='relu', kernel_initializer='he_normal'),
    tf.keras.layers.Dropout(0.3),
    tf.keras.layers.Dense(64, activation='relu', kernel_initializer='he_normal'),
    tf.keras.layers.Dropout(0.3),
    tf.keras.layers.Dense(16, activation='relu', kernel_initializer='he_normal'),
    tf.keras.layers.Dropout(0.3),
    tf.keras.layers.Dense(1, activation = 'sigmoid')
])
```

```
modelo = tf.keras.Sequential([

    tf.keras.layers.Conv2D(8, (7, 7), activation='relu', padding='same',
                           kernel_initializer='he_normal', input_shape = (256, 256, 1)),
    tf.keras.layers.MaxPooling2D(2,2),

    tf.keras.layers.Conv2D(16, (5, 5), activation='relu', padding='same',
                           kernel_initializer='he_normal'),
    tf.keras.layers.MaxPooling2D(2,2),

    tf.keras.layers.Conv2D(32, (3, 3), activation='relu', padding='same',
                           kernel_initializer='he_normal'),
    tf.keras.layers.MaxPooling2D(2,2),
    tf.keras.layers.Conv2D(64, (3, 3), activation='relu', padding='same',
                           kernel_initializer='he_normal'),
    tf.keras.layers.MaxPooling2D(2,2),

    tf.keras.layers.Flatten(),

    tf.keras.layers.Dense(128, activation='relu', kernel_initializer='he_normal'),
    tf.keras.layers.Dropout(0.3),
    tf.keras.layers.Dense(64, activation='relu', kernel_initializer='he_normal'),
    tf.keras.layers.Dropout(0.3),
    tf.keras.layers.Dense(16, activation='relu', kernel_initializer='he_normal'),
    tf.keras.layers.Dropout(0.3),
    tf.keras.layers.Dense(1, activation = 'sigmoid')
])
```

Fig. 14 a) arquitectura Red_BGF_Trans_Bin, b) Arquitectura Red_BGF_Long_Bin

Se ha aplicado un agresivo aumento de datos para prevenir el sobreajuste. Se utilizó el optimizador *"Adam"* y la función de pérdida *"Binary_CrossEntropy"* para clasificación binaria. En el modelo transversal, se aplicó *"ReduceLROnPlateau"* para ajustar la tasa de aprendizaje si la precisión no mejoraba, Fig. 15, con un tamaño de lote de 15 para el modelo transversal y 10 para el longitudinal, durante 300 épocas.

```
modelo.compile(
    optimizer = 'adam',
    loss = 'binary_crossentropy',
    metrics=['accuracy']
)


callbacks = [
    tf.keras.callbacks.ReduceLROnPlateau(monitor='val_accuracy',
                                         factor=0.1, patience=5, min_lr=1e-6)
   ]

datagen = ImageDataGenerator(
    rotation_range=10,         # Rotación aleatoria en grados
    width_shift_range=0.1,     # Desplazamiento horizontal
    height_shift_range=0.1,    # Desplazamiento vertical
    zoom_range=0.1,            # Zoom aleatorio
    horizontal_flip=True,      # Inversión horizontal
    fill_mode='nearest'        # Relleno de píxeles al aplicar transformaciones
)

datagen.fit(X_train)
```

Fig. 15 Callback de Red_BGF_Trans_Bin

3.2.4. Red_BGF_Trans_Bin_Filtros & Red_BGF_Long_Bin_Filtros

Para estos dos últimos códigos se ha vuelto a usar una división de datos del 75% para entrenamiento y 25% para validación.

Las configuraciones para los dos modelos son en esta ocasión diferentes, teniendo diferente distribución de kernels en las capas iniciales de 7 x 7 para el modelo transversal y 5 x 5 para el longitudinal comenzando en 8 filtros hasta los 64, con activación *"relu"* y *"he_normal"*. Se introduce de nuevo para ambos modelos en las capas densas, el *"kernel_regularizer"*, se recuerda que este parámetro se utiliza para reducir el sobreajuste de datos y aumentar la generalización del modelo. Con una capa densa de entrada de 128 neuronas y una salida de 1.

```
modelo = tf.keras.Sequential([

    tf.keras.layers.Conv2D(8, (7, 7), activation='relu', padding='same',
                           kernel_initializer='he_normal', input_shape=(256, 256, 1)),
    tf.keras.layers.MaxPooling2D(2,2),

    tf.keras.layers.Conv2D(16, (5, 5), activation='relu', padding='same',
                           kernel_initializer='he_normal'),
    tf.keras.layers.MaxPooling2D(2,2),

    tf.keras.layers.Conv2D(32, (3, 3), activation='relu', padding='same',
                           kernel_initializer='he_normal'),
    tf.keras.layers.MaxPooling2D(2,2),
    tf.keras.layers.Conv2D(64, (3, 3), activation='relu', padding='same',
                           kernel_initializer='he_normal'),
    tf.keras.layers.MaxPooling2D(2,2),

    tf.keras.layers.Flatten(),

    tf.keras.layers.Dense(128, activation='relu', kernel_regularizer=l2(0.001),
                          kernel_initializer='he_normal'),
    tf.keras.layers.Dropout(0.1),
    tf.keras.layers.Dense(64, activation='relu', kernel_regularizer=l2(0.001),
                          kernel_initializer='he_normal'),
    tf.keras.layers.Dropout(0.1),
    tf.keras.layers.Dense(16, activation='relu', kernel_regularizer=l2(0.001),
                          kernel_initializer='he_normal'),
    tf.keras.layers.Dropout(0.1),
    tf.keras.layers.Dense(1, activation = 'sigmoid')
])
```

```
modelo = tf.keras.Sequential([

    tf.keras.layers.Conv2D(8, (5, 5), activation='relu', padding='same',
                           kernel_initializer='he_normal', input_shape = (256, 256, 1)),
    tf.keras.layers.MaxPooling2D(2,2),

    tf.keras.layers.Conv2D(16, (5, 5), activation='relu', padding='same',
                           kernel_initializer='he_normal'),
    tf.keras.layers.MaxPooling2D(2,2),

    tf.keras.layers.Conv2D(32, (3, 3), activation='relu', padding='same',
                           kernel_initializer='he_normal'),
    tf.keras.layers.MaxPooling2D(2,2),
    tf.keras.layers.Conv2D(64, (3, 3), activation='relu', padding='same',
                           kernel_initializer='he_normal'),
    tf.keras.layers.MaxPooling2D(2,2),

    tf.keras.layers.Flatten(),

    tf.keras.layers.Dense(128, activation='relu', kernel_regularizer=l2(0.01),
                          kernel_initializer='he_normal'),
    tf.keras.layers.Dropout(0.1),
    tf.keras.layers.Dense(64, activation='relu', kernel_regularizer=l2(0.01),
                          kernel_initializer='he_normal'),
    tf.keras.layers.Dropout(0.1),
    tf.keras.layers.Dense(16, activation='relu', kernel_regularizer=l2(0.01),
                          kernel_initializer='he_normal'),
    tf.keras.layers.Dropout(0.1),
    tf.keras.layers.Dense(1, activation = 'sigmoid')
])
```

Fig. 16 a) Arquitectura Red_BGF_Trans_Bin_Filtros, b) Arquitectura Red_BGF_Long_Bin_Filtros

3.3. Métricas

Se han usado las mismas métricas para monitorear el rendimiento de todos modelos en cada época. Estas son: "accuracy" o exactitud y "loss function" o función de pérdida.
La precisión mide la proporción de predicciones correctas sobre el total de ejemplos evaluados mientras que la función de pérdida cuantifica el error entre las predicciones y las etiquetas reales. Esta función sirve como guía al modelo ya que el objetivo es minimizarla.

Tabla 2 Tipos de clases en las métricas

	Predicted Class	
	Positive	***Negative***
Positive	True Positive (TP)	False Negative (FN)
Negative	False Positive (FP)	True Negative (TN)

Accuracy (Exactitud)
Esta métrica mide el ratio de predicciones correctas sobre el total de elementos según la Tabla 2. Evalúa la proporción de aciertos totales. [17]

$$Accuracy = \frac{TP-TN}{TP+TN+FP+FN} \quad [17]$$

Función de pérdida o "Loss Function"
Ambas métricas se monitorearon durante cada época para el conjunto de entrenamiento y el de validación, así se puede detectar si existe sobreajuste, convergencia o discrepancias entre el entrenamiento y la validación.
Además, se ha realizado una matriz de confusión para cuantificar que clase es la menos acertada, saber cuáles son las precisiones de cada clase y/o evaluar problemas de sesgos que no se pueden identificar con las métricas. Esta matriz se realiza en el conjunto de validación y en lugar de obtener una predicción como su propio nombre indica, lo que se obtiene es una salida que consiste en probabilidades para cada clase. Esta probabilidad viene dada por la función de activación de la capa de salida.

En modelos binarios, se aplica un umbral de 0.5: predicciones mayores se asignan a la clase 1, y las menores, a la clase 0 y en modelos multicategóricos, dado que se ha realizado el *"One-Hot Encoding"*, es necesario revertir esta codificación para extraer la clase, por lo que, en lugar de usar un umbral, la salida es la clase con mayor probabilidad en la predicción. En el anexo 6 se encuentra la programación de las métricas.

4. Fase de segmentación

Para la siguiente fase del proyecto se han realizado dos modelos U-Net, que se les conocerá como Unet_trans para el código encargado de segmentar las zonas claves en las imágenes transversales y Unet_long para código de las imágenes longitudinales.

4.1. Preprocesamiento

En esta fase, se han desarrollado dos códigos: Unet_trans para imágenes transversales y Unet_long para longitudinales. Ambos comparten el mismo procesamiento de imágenes, pero difieren en el tipo de datos utilizado. Para las redes de segmentación, se ha realizado un procesamiento más intensivo debido a las mayores necesidades de entrada. Solo se han incluido imágenes categorizadas como *"Good" y "Fair"* para enfocar la segmentación, excluyendo las *"Bad"* mediante un filtro en el archivo JSON. Esto deja un conjunto de datos de tamaño (862, 256, 256, 1) para Unet_trans y (932, 256, 256, 1) para Unet_long. En el anexo 7 encontramos una explicación más detallada del procesamiento en la fase de segmentación.

Se han utilizado imágenes con ROIs pintadas para crear máscaras que servirán como entradas en lugar de etiquetas. Las imágenes originales y las pintadas se cargan de manera secuencial usando referencias del archivo JSON, asegurando que los índices coincidan con funciones *"assert"*. Las máscaras son imágenes en blanco y negro que reflejan las ROIs sobre un fondo negro. Cada zona de interés se pintó con colores RGB fijos y se creó un mapa de colores con una tolerancia de intensidad de ±10, Fig. 17.

```
# Definir los colores en formato RGB
color_nervio_bajo_rgb = [0-10, 255-10, 255-10]  # Rango inferior para nervio (RGB)
color_nervio_alto_rgb = [0+10, 255+10, 255+10]  # Rango superior para nervio (RGB)

color_borde_bajo_rgb = [255-10, 0-10, 255-10]  # Rango inferior para borde (RGB)
color_borde_alto_rgb = [255+10, 0+10, 255+10]  # Rango superior para borde (RGB)

color_ligamento_bajo_rgb = [250-10, 128-10, 114-10]  # Rango inferior para ligamento (RGB)
color_ligamento_alto_rgb = [250+10, 128+10, 114+10]  # Rango superior para ligamento (RGB)

color_hueso_bajo_rgb = [255-10, 255-10, 0-10]  # Rango inferior para hueso (RGB)
color_hueso_alto_rgb = [255+10, 255+10, 0+10]  # Rango superior para hueso (RGB)
```

Fig. 17 Rango de colores de las ROIs

Se ha creado una función para generar máscaras que convierte las imágenes con ROIs a formato RGB y detecta píxeles dentro del rango de colores definido en el mapa, Fig. 18. Esto permite crear imágenes binarias para cada zona de interés, asignando valor 1 a las áreas relevantes y 0 al fondo. Luego, todas las clases se combinan en una sola máscara iniciando una matriz de ceros con dimensiones iguales a las entradas, Fig. 19 y Fig. 20. Cada clase recibe un valor único, formando un mapa de clases con rangos [0, 1, 2, 3, 4] para máscaras transversales y [0, 1, 2, 3] para longitudinales ya que en estas no se pinta el ligamento, Fig. 18.

```
# Función para crear la máscara de colores
def crear_mascara(color_bajo, color_alto, image_rgb):

    mask = np.all(np.logical_and(image_rgb >= color_bajo, image_rgb <= color_alto), axis=-1)
    return mask

for image in X_roi:
    if image is None:
        print("Error al cargar una imagen.")
        continue

    image_rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

    mask_nervio = crear_mascara(color_nervio_bajo_rgb, color_nervio_alto_rgb, image_rgb)
    mask_borde = crear_mascara(color_borde_bajo_rgb, color_borde_alto_rgb, image_rgb)
    mask_ligamento = crear_mascara(color_ligamento_bajo_rgb, color_ligamento_alto_rgb, image_rgb)
    mask_hueso = crear_mascara(color_hueso_bajo_rgb, color_hueso_alto_rgb, image_rgb)

    image_combined = np.zeros(image_rgb.shape[:2], dtype=np.uint8)  # Inicializamos con fondo (clase 0)

    image_combined[mask_nervio] = 1       # Nervio (clase 1)
    image_combined[mask_borde] = 2        # Borde (clase 2)
    image_combined[mask_ligamento] = 3    # Ligamento (clase 3)
    image_combined[mask_hueso] = 4        # Hueso (clase 4)

    X_mask.append(image_combined)
```

Fig. 18 Función para crear máscaras en segmentación

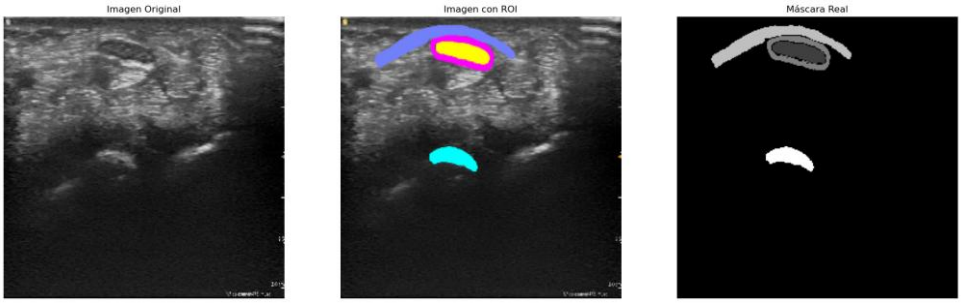


Fig. 19 Ejemplo de las ROIs y máscara en imagen transversal

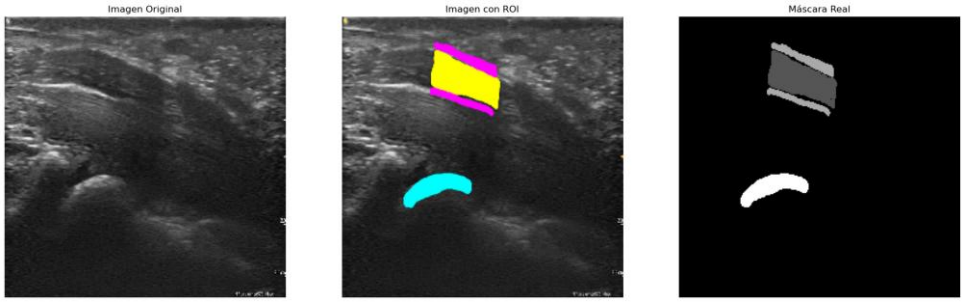


Fig. 20 Ejemplo de las ROIs y máscara en imagen longitudinal

Una vez se realizan las máscaras se ha realizado un balance de clases. Dado que el fondo (clase 0) domina significativamente, se ha utilizado una función para calcular automáticamente los pesos de cada clase, asignando mayor importancia a las clases minoritarias, Fig. 21. Como resultado se obtiene el diccionario:
{0: 0.21055961538414947, 1: 24.357625551088162, 2: 21.59386418410506, 3: 10.193446944647189, 4: 15.317546894573299}
Al igual que ocurre con las multicategorías, como se cuenta con varias clases, se realiza el *"One-Hot encoding"* al mapa de clases.

```
# Ponderamos las clases
X_mask_flatt = X_mask.flatten()
y = X_mask_flatt
classes=np.unique(X_mask_flatt)
class_weights = class_weight.compute_class_weight('balanced', classes=classes, y=X_mask_flatt)
class_weights_dict = {i: class_weights[i] for i in classes}
print(class_weights_dict)

X_masks_one_hot = to_categorical(X_mask, num_classes=5)
```

Fig. 21 Código de ponderación de clases en segmentación

Como último paso se ha creado una función de pérdida personalizada basada en "*Categorical_CrossEntropy"*, adaptada para incorporar los pesos previamente calculados. Estos pesos permanecen constantes durante todo el entrenamiento, y la función se centra en ajustar los bias para minimizar la pérdida, Fig. 22. Esto asegura que

el modelo preste mayor atención a las clases minoritarias y mejore su rendimiento en la segmentación.

```python
def weighted_loss(y_true, y_pred):

    class_indices = tf.argmax(y_true, axis=-1)

    class_weights = np.array([class_weights_dict.get(i, 1.0) for i in range(len(class_weights_dict))])
    class_weights_tensor = tf.constant(class_weights, dtype=tf.float32)

    sample_weights = tf.gather(class_weights_tensor, class_indices)

    loss = tf.keras.losses.CategoricalCrossentropy(from_logits=False)(y_true, y_pred)

    return loss * sample_weights
```

Fig. 22 Función personalizada de clases ponderadas en segmentación

4.2. Arquitectura de los modelos

Para la segmentación se ha optado por usar una red U-Net, este tipo de redes fueron originalmente diseñadas para la segmentación de imágenes biomédicas y son popularmente conocidas por su alto éxito en tareas de segmentación. [18]
La principal característica de estas redes junto con las *"Skip Connections"* es su arquitectura. Esta está dividida en dos partes, una inicial llamada "Encoder" o *"Contracting path"* y una segunda llamada *"Decoder"* o *"Expansive path"*. El *"Encoder"* se encarga de extraer características y reducir la dimensión espacial de las imágenes, de forma que se obtiene una alta cantidad de detalles y relaciones espaciales y el *"Decoder"* se encarga de recuperar la resolución original de la imagen de entrada y generar una máscara de segmentación.
Las *"Skip Connections"* o conexiones de salto son, como su propio nombre indica, conexiones entre el *"Encoder"* y el *"Decoder"*. A medida que la imagen va pasando por el *"Encoder"*, esta va perdiendo resolución debido a las capas *"Maxpooling"* que reducen las dimensiones espaciales de la imagen, esto conlleva una pérdida de información, sin embargo, gracias a estas *"Skip Connections"*, cuando la imagen comienza a recuperar su resolución durante el *"Decoder"* se utiliza información del *"Encoder"* para combinar características superficiales (antes de reducir la dimensión) y detalles profundos de las capas más bajas para proporcionar información y recuperar detalles finos, de esta forma se mejora la precisión. Esto también tiene un impacto en la evasión del *"overfitting"*, en el aumento de la robustez y en la mejora del flujo de gradientes durante el entrenamiento.

La división de entrenamiento y validación se ha realizado al 25% para el conjunto de validación. Se ha optado por usar la misma arquitectura de red para ambos modelos ya que ambas imágenes comparten similitudes de ruido y así se ha visto reflejado en los resultados.

La estructura del modelo es una estructura clásica de U-Net comenzando con un *"Encoder"* de dos convoluciones de kernel 3x3 de 16 filtros, con activación *"Relu"* y activación de kernel "h2_normal" hasta llegar a los 256 filtros, pasando de una imagen de 256x256 píxeles a 16x16 píxeles, el parámetro *"padding"* asegura estabilidad en las

dimensiones espaciales. En la parte del *"encoder"* se ha usado un *"dropout"* de valor 0.4 entre las convoluciones de cada bloque de filtros.

En la parte del *"Decoder"* se introducen unas capas "Conv2DTranspose", conocidas como deconvoluciones, que son las encargadas de aumentar la resolución espacial de las características capturadas realizando la operación inversa (transpuesta). Todas cuentan con un kernel de 2x2 y un parámetro llamado *"stride"* de 2x2 el cual aumenta la imagen el doble del tamaño, es decir, revierte las capas *"maxpooling"*.
Las capas *"concatenate()"* son las definidas *"Skip Connection"* entre el *"encoder"* y *"decoder"*. Junto con las transpuestas se definen las capas convolucionales que han de tener el mismo número de filtros que las capas transpuestas. En la parde del *"Decoder"* se ha usado un valor de *"dropout"* de 0.3.
Finalmente se acaba con una capa convolucional de 5 filtros y kernel de 1x1, con activación *"softmax"* al ser la entrada un mapa de clases, Fig. 23. En la Fig. 24, se encuentra una representación visual de la red.

```
# Arquitectura
inputs = tf.keras.layers.Input((256, 256, 1))
s = tf.keras.layers.Lambda(lambda x: x/255)(inputs)

# Contracting path (encoder)
c1 = tf.keras.layers.Conv2D(16, (3,3), activation='relu', kernel_initializer='he_normal',
                            padding='same')(s)
c1 = tf.keras.layers.Dropout(0.4)(c1)
c1 = tf.keras.layers.Conv2D(16, (3,3), activation='relu', kernel_initializer='he_normal',
                            padding='same')(c1)
p1 = tf.keras.layers.MaxPooling2D((2,2))(c1)

c2 = tf.keras.layers.Conv2D(32, (3,3), activation='relu',kernel_initializer='he_normal',
                            padding='same') (p1)
c2 = tf.keras.layers.Dropout(0.4)(c2)
c2 = tf.keras.layers.Conv2D(32, (3,3), activation='relu', kernel_initializer='he_normal',
                            padding='same')(c2)
p2 = tf.keras.layers.MaxPool2D((2,2))(c2)

c3 = tf.keras.layers.Conv2D(64, (3,3), activation='relu',kernel_initializer='he_normal',
                            padding='same') (p2)
c3 = tf.keras.layers.Dropout(0.4)(c3)
c3 = tf.keras.layers.Conv2D(64, (3,3), activation='relu', kernel_initializer='he_normal',
                            padding='same')(c3)
p3 = tf.keras.layers.MaxPool2D((2,2))(c3)

c4 = tf.keras.layers.Conv2D(128, (3,3), activation='relu',kernel_initializer='he_normal',
                            padding='same') (p3)
c4 = tf.keras.layers.Dropout(0.4)(c4)
c4 = tf.keras.layers.Conv2D(128, (3,3), activation='relu', kernel_initializer='he_normal',
                            padding='same')(c4)
p4 = tf.keras.layers.MaxPool2D((2,2))(c4)

c5 = tf.keras.layers.Conv2D(256, (3,3), activation='relu',kernel_initializer='he_normal',
                            padding='same') (p4)
c5 = tf.keras.layers.Dropout(0.4)(c5)
c5 = tf.keras.layers.Conv2D(256, (3,3), activation='relu', kernel_initializer='he_normal',
                            padding='same')(c5)
```

```
# Expansive path (decoder)
u6 = tf.keras.layers.Conv2DTranspose(128, (2,2), strides=(2,2), padding='same')(c5)
u6 = tf.keras.layers.concatenate([u6,c4])
c6 = tf.keras.layers.Conv2D(128, (3,3), activation='relu', kernel_initializer='he_normal',
                            padding='same')(u6)
c6 = tf.keras.layers.Dropout(0.3)(c6)
c6 = tf.keras.layers.Conv2D(128, (3,3), activation='relu', kernel_initializer='he_normal',
                            padding='same')(c6)

u7 = tf.keras.layers.Conv2DTranspose(64, (2,2), strides=(2,2), padding='same')(c6)
u7 = tf.keras.layers.concatenate([u7,c3])
c7 = tf.keras.layers.Conv2D(64, (3,3), activation='relu', kernel_initializer='he_normal',
                            padding='same')(u7)
c7 = tf.keras.layers.Dropout(0.3)(c7)
c7 = tf.keras.layers.Conv2D(64, (3,3), activation='relu', kernel_initializer='he_normal',
                            padding='same')(c7)

u8 = tf.keras.layers.Conv2DTranspose(32, (2,2), strides=(2,2), padding='same')(c7)
u8 = tf.keras.layers.concatenate([u8,c2])
c8 = tf.keras.layers.Conv2D(32, (3,3), activation='relu', kernel_initializer='he_normal',
                            padding='same')(u8)
c8 = tf.keras.layers.Dropout(0.3)(c8)
c8 = tf.keras.layers.Conv2D(32, (3,3), activation='relu', kernel_initializer='he_normal',
                            padding='same')(c8)

u9 = tf.keras.layers.Conv2DTranspose(16, (2,2), strides=(2,2), padding='same')(c8)
u9 = tf.keras.layers.concatenate([u9,c1])
c9 = tf.keras.layers.Conv2D(16, (3,3), activation='relu', kernel_initializer='he_normal',
                            padding='same')(u9)
c9 = tf.keras.layers.Dropout(0.3)(c9)
c9 = tf.keras.layers.Conv2D(16, (3,3), activation='relu', kernel_initializer='he_normal',
                            padding='same')(c9)

outputs = tf.keras.layers.Conv2D(5, (1,1), activation='softmax')(c9)
```

Fig. 23 Arquitectura encoder y decoder para modelos Unet_Trans y Unet_Long

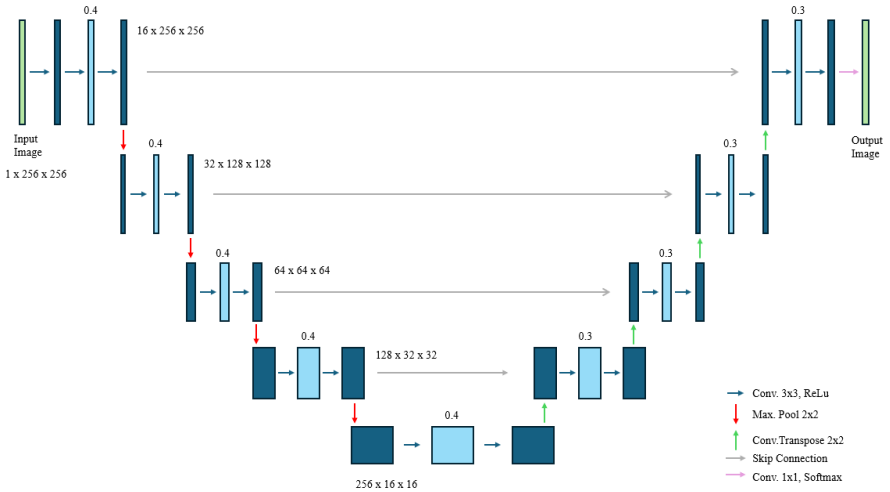


Fig. 24 Representación visual U-Net

Se configuró el *"callback": "EarlyStopping"* de Keras para monitorizar la función de pérdida en la validación. Si no mejora tras 3 épocas consecutivas (parámetro "*patience"*), el entrenamiento se detiene automáticamente, evitando entrenamientos innecesarios.
El modelo se compiló con el optimizador *"Adam"*, la función de pérdida definida previamente en el preprocesamiento, y las métricas de precisión y *"recall"*, que se detallarán en el siguiente apartado.
Para Unet_trans se ha usado un tamaño de lote de 25 imágenes con 100 épocas de entrenamiento mientras que para Unet_long se han necesitado un tamaño de lote de 35 imágenes con 200 épocas. En el anexo 8 se encuentra un resumen de los modelos.

4.3. Métricas

Para las tareas de segmentación, a diferencia de las redes de clasificación, es necesario usar más métricas ya que no hay una métrica que te mida directamente el desempeño del modelo.
Para ello, se han usado 6 tipos de métricas con diferentes enfoques cuya programación se encuentra en el anexo 9.

Accuracy (exactitud) y Loss Function
Utilizadas y explicadas en la fase de identificación.

Precision Metric (precisión)
La precisión mide cuantas de las predicciones positivas son realmente correctas. Es el ratio entre TP y todos los elementos calificados como positivos. [17]

$$Precision = \frac{TP}{TP+FP} \qquad [17]$$

Recall (sensibilidad)
Esta métrica parecida a la precisión mide cuantos casos positivos reales son correctamente identificados, a diferencia de la precisión, no tiene en cuenta los falsos negativos.

$$Recall = \frac{TP}{TP+ FN} \qquad [17]$$

IOU (Intersection Over Union)
El IoU o índice de Jaccard evalúa la superposición píxel a píxel entre una predicción, en este caso la segmentación predicha, y el área real. [19]
IoU = 0: no hay superposición, IoU=1: Predicción perfecta

$$IOU = \frac{|A \cap B|}{|A \cup B|} = \frac{|\mathrm{I}|}{|U|} \qquad [19]$$

F1-Score

El F1-score combina la precisión y el recall en un único valor armónico, es decir, mide el balance entre ambos. Se utiliza cuando las clases se encuentran desbalanceadas. [17]

$$F1 - Score = 2\ x\ \frac{Precision\ x\ Recall}{Precision + Recall} \qquad [17]$$

5. Fase de Diagnóstico

En esta última fase del proyecto se van a realizar dos códigos, uno para cada tipo de imagen, encargados de diagnosticar y detectar si el paciente de las imágenes sufre del STC y de ser así, evaluar su severidad. Los códigos se llamarán Diag_trans y Diag_long. La clasificación al diagnosticar será: Normal, Leve, Moderado o Severo

5.1. Preprocesamiento

El preprocesamiento, que será igual para ambos códigos, comienza como en los anteriores códigos, cargando los conjuntos de datos. En esta ocasión es necesario extraer más información del json, para ello se van a crear variables para guardar: las fotos originales, las etiquetas de calidad de las imágenes: *{'bad': 0, 'good': 1, 'fair': 2}* y los diagnósticos pertenecientes a cada imagen: *{'normal': 0, 'leve': 1, 'moderado': 2, 'severo': 3}*, a este último diccionario se le aplicará el *"One-Hot encoding"* al tratarse de una red multicategórica.

Se han creado dos diccionarios, uno para definir las etiquetas de la calidad de las imágenes, el cual se usará para filtrar únicamente las imágenes *"Good" y "Fair",* ya que al igual que en los anteriores códigos se cuenta con un conjunto de datos tanto para transversal como para longitudinal relativamente pequeño.

Una vez cargadas las imágenes, es necesario cargar las imágenes con ROI para posteriormente realizar las máscaras. Una vez realizadas las máscaras, se calculan nuevamente los pesos ponderados para luego definir la función de pérdida definida en la fase de segmentación. Hasta ahora el proceso ha sido el mismo que el preprocesamiento de las redes de segmentación, a excepción de la variable de diagnóstico.

El objetivo de realizar el mismo procesamiento de nuevo es para utilizar la red se segmentación como parte del procesamiento nuevo. Se va a utilizar esta red para localizar en cada foto el nervio y posteriormente realizar un recorte a cada imagen para alimentar a Diag_trans y Diag_long con imágenes recortadas de los nervios. Así, la red se centra únicamente en estudiar la morfología del nervio y se reduce la cantidad de ruido con información irrelevante.

De esta forma, con la función de tensorflow *".load_model()"* cargamos el modelo con mejores resultados de segmentación y se proceden a definir las funciones para recortar las imágenes.

Es importante destacar que la red de segmentación ha sido entrenada con el mismo conjunto de datos que se le va a pasar para localizar el nervio. Esto da como resultado una segmentación precisa ya que la red se "conoce" las imágenes, pero no se está

evaluando el rendimiento de esta red, si no que se está usando como una herramienta de procesamiento.
Una vez, cargado el código, se crea una función donde se van a almacenar las máscaras predichas por la red. Sin embargo, estas máscaras solo van a ser de la clase nervio, es decir, son únicamente máscaras de los nervios, Fig. 25.

```python
# Función para realizar la predicción y guardar las máscaras
def predecir_y_guardar(model, X_imagenes, output_dir):
    os.makedirs(output_dir, exist_ok=True)
    mask_pred = []
    mask_nervios = []

    for i, imagen in enumerate(X_imagenes):

        imagen_input = np.expand_dims(imagen, axis=-1)  # Añadir la dimensión del canal
        imagen_input = np.expand_dims(imagen_input, axis=0)  # Añadir la dimensión del batch

        # Realizar la predicción
        prediccion = model.predict(imagen_input, verbose=0)

        # Convertir la predicción a una máscara con la clase de mayor probabilidad
        prediccion_clase = np.argmax(prediccion, axis=-1)  # Forma (1, 128, 128)
        prediccion_clase = np.squeeze(prediccion_clase)  # Eliminar dimensiones de tamaño 1
        mascara_clase_1 = np.uint8(prediccion_clase == 1) * 255

        # Añadir la máscara a mask_pred
        mask_pred.append(prediccion_clase)
        mask_nervios.append(mascara_clase_1)

    return mask_nervios
```

Fig. 25 Función para la predicción de máscaras en la fase de diagnóstico

El siguiente paso es crear la función para realizar los recortes, Fig. 26. Primero se han usado las máscaras de los nervios, que recordemos que son máscaras binarias de dimensiones exactas a las imágenes originales. Estas máscaras al ser binarias, los píxeles distintos a 0 son el nervio predicho, de modo que se ha localizado dichos píxeles y se ha calculado un *"bounding box"* que contenga a todos los píxeles que pertenecen al nervio. Para garantizar que todos los píxeles que pertenecen al nervio en las imágenes originales se encuentran dentro del recorte, el *"bounding box"* de las máscaras se ha agrandado a un tamaño fijo de 70x70 donde el nervio predicho se encuentra centrado. De esta forma también se garantiza uniformidad de dimensiones en el conjunto de datos y no es necesario realizar interpolaciones para agrandar o reducir las imágenes. La función también está programada por si hay algún caso en el que la segmentación es más grande que el *"bounding box"*, en cuyo caso el recorte será el del *"bounding box"* y se notificará que imagen no cumpla con las dimensiones.
Una vez se tiene el *"bounding box"* de 70x70 en las máscaras, se calculan las coordenadas individuales de cada *"box"* y se realiza un recorte en dichas coordenadas en las correspondientes imágenes originales. Se resalta así una vez más la importancia de cargar las variables de forma correcta y secuencial.
Finalmente, los recortes se guardan en un array y obtenemos un conjunto de datos para Diag_trans de (817, 70, 70, 1) y (851, 70, 70, 1) para Diag_Long. Ligeramente inferior al conjunto de datos de los anteriores códigos debido a que no se contaba con el diagnóstico de todas las imágenes.
Los datos, como en todos los códigos, son normalizados para la entrada a la red.

```python
nervios = []
tamaño_fijo = 70 # Tamaño fijo del recorte

for i, (mascara, imagen) in enumerate(zip(mask_nervios, X_imagenes_long)):
    # Coordenadas del bounding box a partir de la máscara
    coords = np.column_stack(np.where(mascara == 255))  # Índices de píxeles de clase 1
    if coords.size > 0:
        # Bounding box inicial
        x_min, y_min = coords.min(axis=0)
        x_max, y_max = coords.max(axis=0)

        # Centro del bounding box
        x_centro = (x_min + x_max) // 2
        y_centro = (y_min + y_max) // 2

        # Calcular los límites del nuevo bounding box (70x70)
        x_min_nuevo = max(0, x_centro - tamaño_fijo // 2)
        x_max_nuevo = min(imagen.shape[1], x_centro + tamaño_fijo // 2)
        y_min_nuevo = max(0, y_centro - tamaño_fijo // 2)
        y_max_nuevo = min(imagen.shape[0], y_centro + tamaño_fijo // 2)

        # Ajustar el tamaño para que sea exactamente 70x70 (si toca los bordes)
        if x_max_nuevo - x_min_nuevo < tamaño_fijo:
            if x_min_nuevo == 0:
                x_max_nuevo = tamaño_fijo
            elif x_max_nuevo == imagen.shape[1]:
                x_min_nuevo = imagen.shape[1] - tamaño_fijo

        if y_max_nuevo - y_min_nuevo < tamaño_fijo:
            if y_min_nuevo == 0:
                y_max_nuevo = tamaño_fijo
            elif y_max_nuevo == imagen.shape[0]:
                y_min_nuevo = imagen.shape[0] - tamaño_fijo

        # Recortar la región del nervio con el bounding box ajustado
        recorte = imagen[x_min_nuevo:x_max_nuevo, y_min_nuevo:y_max_nuevo]
```

Fig. 26 Función para realizar "bounding box"

Para finalizar el procesamiento de imágenes se han aplicado una serie de filtros a las imágenes recortadas. Los filtros son los mismos que los usado en las redes BGF_Trans_Bin_filtros y BGF_Long_Bin_filtros pero en un orden distinto ya que ahora los filtros se van a centrar en oscurecer el nervio y resaltar los bordes del mismo.

En esta ocasión se ha aplicado primero un filtro mediano para suavizar el ruido de la imagen ya que el borde del nervio y los tejidos circundantes aún se encuentran en las imágenes pese a los recortes. Luego se ha aplicado un filtro de contraste adaptativo Clahe, esta vez con un *"clip limit"* de 0.5 y un kernel de 2x2, parámetros más restrictivos ya que se quiere una buena diferenciación borde-nervio y en lugar de aplicar un filtro OTSU, se ha aplicado un contraste adicional que resalta aún más la diferencia borde-nervio. Es importante a la hora de diagnosticar que la red aprenda no solo la morfología del nervio si no también la del borde, por ello los filtros que binarizan no se han aplicado en esta ocasión, para no perder detalles finos de las imágenes.

5.2. Arquitectura de los modelos

En este apartado se explica la arquitectura de los modelos, se puede encontrar un resumen de los mismos con sus parámetros en el anexo 10.

La división del conjunto de entrenamiento y validación para la etapa final ha sido del 75% para entrenamiento y 25% para validación. Al ser una red multicategórica es necesario obtener muchos ejemplos de validación de cada clase para que la red aprenda a generalizar.

En esta ocasión las redes, aunque diferentes para ambos códigos, cuentan con una extracción de características menor ya que los mismos recortes realizados fuerzan a las redes a centrarse en lo importante de las imágenes.

Para ambos códigos se ha utilizado aumento de datos para evitar el sobreajuste. Se ha utilizado en esta fase para aumentar las imágenes, una rotación y un volteo horizontal.
La razón de no usar un aumento agresivo reside en no querer modificar de forma excesiva las imágenes. Al realizar los recortes, se centra la atención en toda la imagen al ser esta el nervio, de este modo, se quieren captar detalles finos característicos de cada categoría y al usar algún aumento de datos que requiera estirar las imágenes o realizar zooms podemos perder los detalles difíciles de captar.

5.2.1. Diag_trans

La red Diag_trans sigue una estructura simple comenzando con 16 filtros convolucionales de 3x3, función de activación: *"relu"* e iniciador kernel: *"he_normal",* capas *"maxpooling"* de kernel 2x2 y *"batch normalization"* después de las convoluciones, hasta llegar a los 64 filtros convolucionales.
En cuanto a las capas densas, se comienza con una capa de 256 neuronas con capas intermedias de 72 y 16 neuronas, y una capa final de 4 neuronas, cada una correspondiente a las 4 posibilidades de diagnóstico. Se ha aplicado un fuerte *"dropout"* de 0.6 entre cada capa densa, Fig. 27.

```
modelo = tf.keras.Sequential([

    tf.keras.layers.Conv2D(16, (3, 3), activation='relu', padding='same', kernel_initializer='he_normal', input_shape=
    (70, 70, 1)),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.MaxPooling2D(2,2),

    tf.keras.layers.Conv2D(32, (3, 3), activation='relu',  padding='same', kernel_initializer='he_normal'),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.MaxPooling2D(2,2),

    tf.keras.layers.Conv2D(64, (3, 3), activation='relu',  padding='same', kernel_initializer='he_normal'),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.MaxPooling2D(2,2),

    tf.keras.layers.Flatten(),

    tf.keras.layers.Dense(256, activation='relu', kernel_initializer='he_normal'),
    tf.keras.layers.Dropout(0.6),
    tf.keras.layers.Dense(72, activation='relu', kernel_initializer='he_normal'),
    tf.keras.layers.Dropout(0.6),
    tf.keras.layers.Dense(16, activation='relu', kernel_initializer='he_normal'),
    tf.keras.layers.Dense(4, activation = 'softmax')
])
```

Fig. 27 Arquitectura modelo Diag_Trans

En cuanto al tamaño de lote, se ha necesitado uno de 30 unidades con 500 épocas.

5.2.2. Diag_Long

Muy similar a la red Diag_trans, esta red comienza con 16 capas convolucionales de kernel 3x3 con activación: *"relu"* y *"kernel_initializer": "he_normal"* hasta los 32 filtros. Esta red cuenta con una menor extracción de características ya que son imágenes con menos información a analizar para las capas.
Las capas densas también se han visto reducidas, comenzando con 256 capas, y finalizando con 4 capas densas, una por cada categoría. Esta vez no se han utilizado capas intermedias ya que se ha reducido la extracción de categorías, y por lo tanto hay menos

parámetros a analizar. Se ha usado un fuerte *"dropout"* de 0.6 en las capas densas para evitar el sobreajuste, Fig. 28.
Para el tamaño de lote se ha optado por 20 unidades y 400 épocas.

```
modelo = tf.keras.Sequential([

    tf.keras.layers.Conv2D(16, (3, 3), activation='relu', padding='same', kernel_initializer='he_normal',
                           input_shape=(70, 70, 1)),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.MaxPooling2D(2,2),

    tf.keras.layers.Conv2D(32, (3, 3), activation='relu',  padding='same', kernel_initializer='he_normal'),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.MaxPooling2D(2,2),

    tf.keras.layers.Flatten(),

    tf.keras.layers.Dense(256, activation='relu', kernel_initializer='he_normal'),
    tf.keras.layers.Dropout(0.6),
    tf.keras.layers.Dense(4, activation = 'softmax')
])
```

Fig. 28 Arquitectura modelo Diag_Long

5.3. Métricas

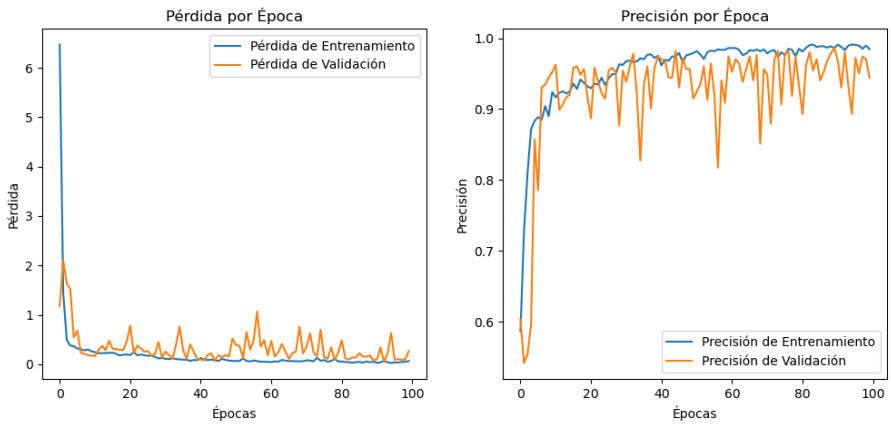
Dado que ambos códigos consisten en redes de clasificación multicategórica, las métricas utilizadas para evaluar el rendimiento del modelo han sido las mismas que las usadas en la fase de identificación.
Estas son las gráficas de la función de pérdida y la precisión (del entrenamiento y validación) en función del número de épocas y una matriz de confusión para evaluar la precisión de cada clase.

6. Resultados

En esta sección se detallan los resultados de las métricas recogidas en cada modelo. Para ello se presentarán gráficas con las curvas de entrenamiento en azul y las curvas de validación en naranja. También se presentarán matrices de confusión equivalentes a los mejores modelos, es decir, la mejor época realizada por cada modelo. Dichos modelos son los que finalmente representan el trabajo realizado. En cada matriz se verá una leyenda conde se encuentran identificadas las clases.

6.1. Fase de Identificación

6.1.1. Red de clasificación transversal o longitudinal (Red_Trans_Long)



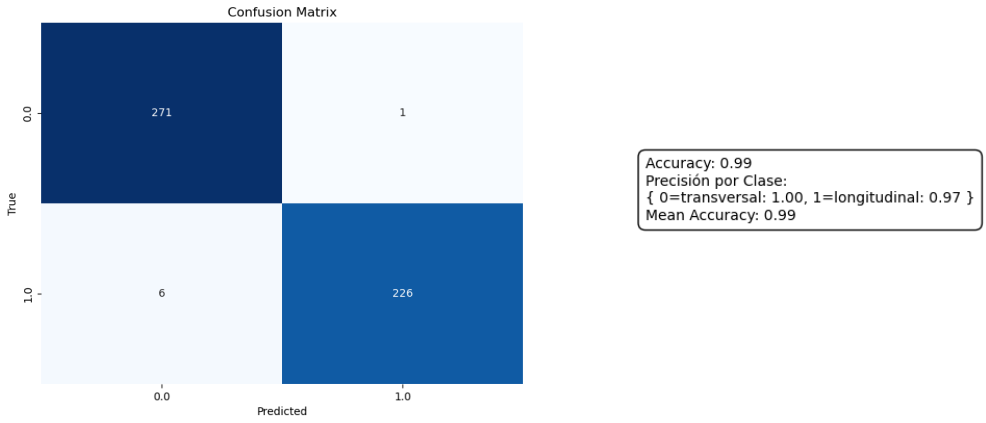


Fig. 29 Métricas Red_Trans_Long

Para la función de pérdida se puede observar en Fig. 29, que no hay sobreajuste en el modelo ya que ambas curvas comparten una misma tendencia descendente, sin embargo, podemos ver que la curva del entrenamiento es más uniforme que la de validación ya que esta tiene oscilaciones. Estas oscilaciones se encuentran dentro del rango [0 1] por lo que no son alarmantes y son parte de la tendencia general de la curva. Principalmente se deben a que el modelo no es capaz de alcanzar una convergencia, muy posiblemente por el tipo de imágenes que está tratando, al ser imágenes ecográficas la cantidad de ruido presente es bastante notable lo que dificulta su clasificación.

Los picos de la curva de validación se ven reflejados en la curva de precisión, donde cada vez que la precisión disminuye, la función de pérdida se aleja de la curva de entrenamiento. A pesar de encontrar una alta oscilación en las precisiones iniciales sobre las épocas 40 y 70, se puede ver que al igual que la curva de validación en la función de pérdida disminuye las oscilaciones, la precisión aumenta y consigue reducir su rango de oscilaciones en el 90% de precisión. Esto se ve reflejado en la matriz de confusión, que se puede ver que para la clase 0: transversal se ha predicho bien 271 imágenes, confundiendo 1 de ellas, y para la clase 1: longitudinal solo se ha equivocado en 6. Esto son precisiones superiores al 96% por lo que el modelo es válido para su uso como clasificador.

6.1.2. Redes de clasificación multicategórica: “Bad”, “Good” o “Fair”

Red_BGF_Trans

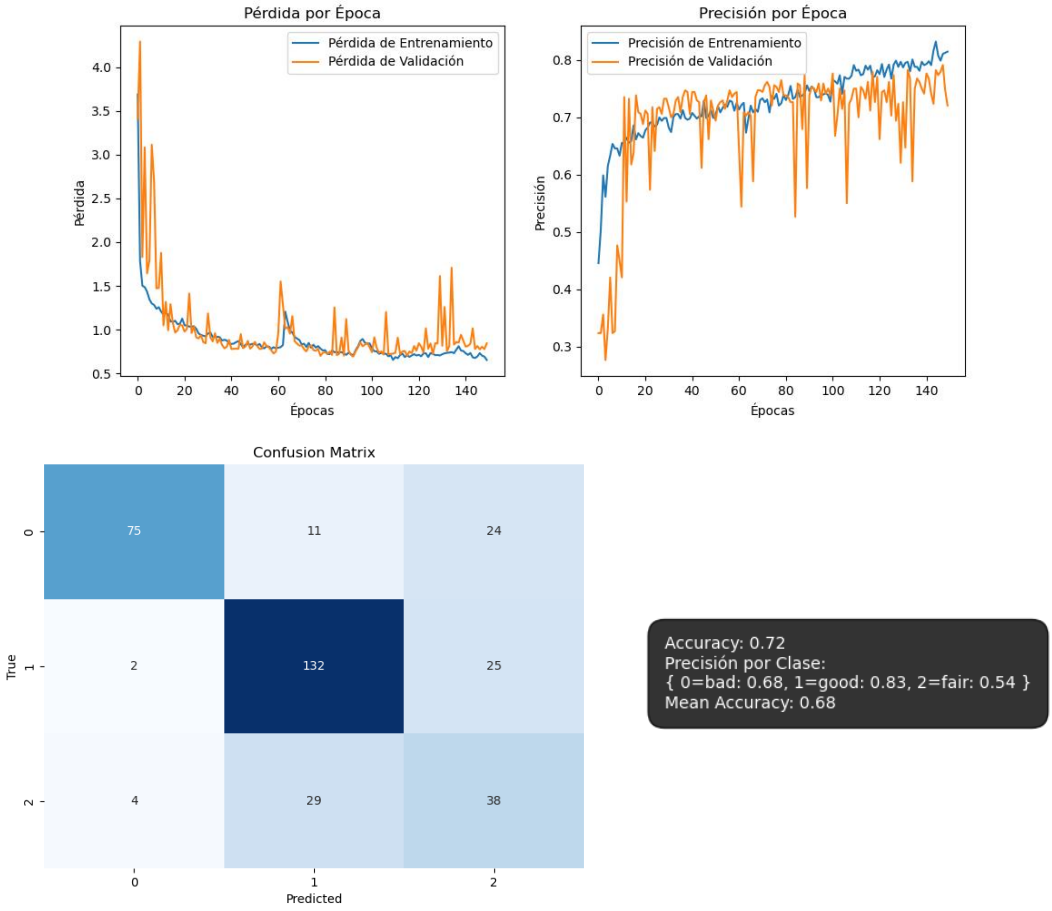


Fig. 30 Métricas Red_BGF_Trans

Comenzando por la red de imágenes transversales, se puede ver en la gráfica de pérdida por época en Fig. 30, que tampoco se observa un sobreajuste, teniendo una curva muy similar la curva del entrenamiento y la de validación. Se vuelven a observar oscilaciones, esta vez con una mayor frecuencia tanto en la pérdida como en la precisión. Se puede ver que la curva de precisión en el entrenamiento tiene una tendencia ascendente mientras que la curva de validación hasta la época 100 se adecúa al entrenamiento, a partir de esta época se estabilizan las oscilaciones en torno al 68% de precisión. Estas oscilaciones tienen una amplitud grande ya que le cuesta al modelo diferenciar las tres clases, y así se ve reflejado en la matriz de precisión, donde la clase con mayor precisión es *“Good”,* seguida por *“Bad” y “Fair”.*

La diferencia entre las tres clases es subjetiva, en cierto modo, es decir, una imagen se califica como *“Bad”* cuando no se encuentra el nervio mediano, el hueso semilunar o esta se encuentra borrosa o desplazada, y una imagen *“Fair”* es similar a una imagen *“Good”* salvo por alguna zona que no se verá con la nitidez de las *“Good”.* A efectos de la red, es difícil que estas sutilezas las detecte por muchas extracciones de características se realice, se tratan de relaciones bastante complejas tanto espaciales como de zonas anatómicas muy particulares y fácilmente confundible dado el ruido de las imágenes. A esto se le añade la poca cantidad de ejemplos, contamos con 1360 imágenes de las cuales

1020 (75%) son para realizar el entrenamiento y 340 son para validar, teniendo en cuenta que se quieren diferenciar tres categorías dentro de este conjunto, no se cuentan con datos suficientes como para realizar un modelo multicategórico robusto y preciso.

Red_BGF_Long

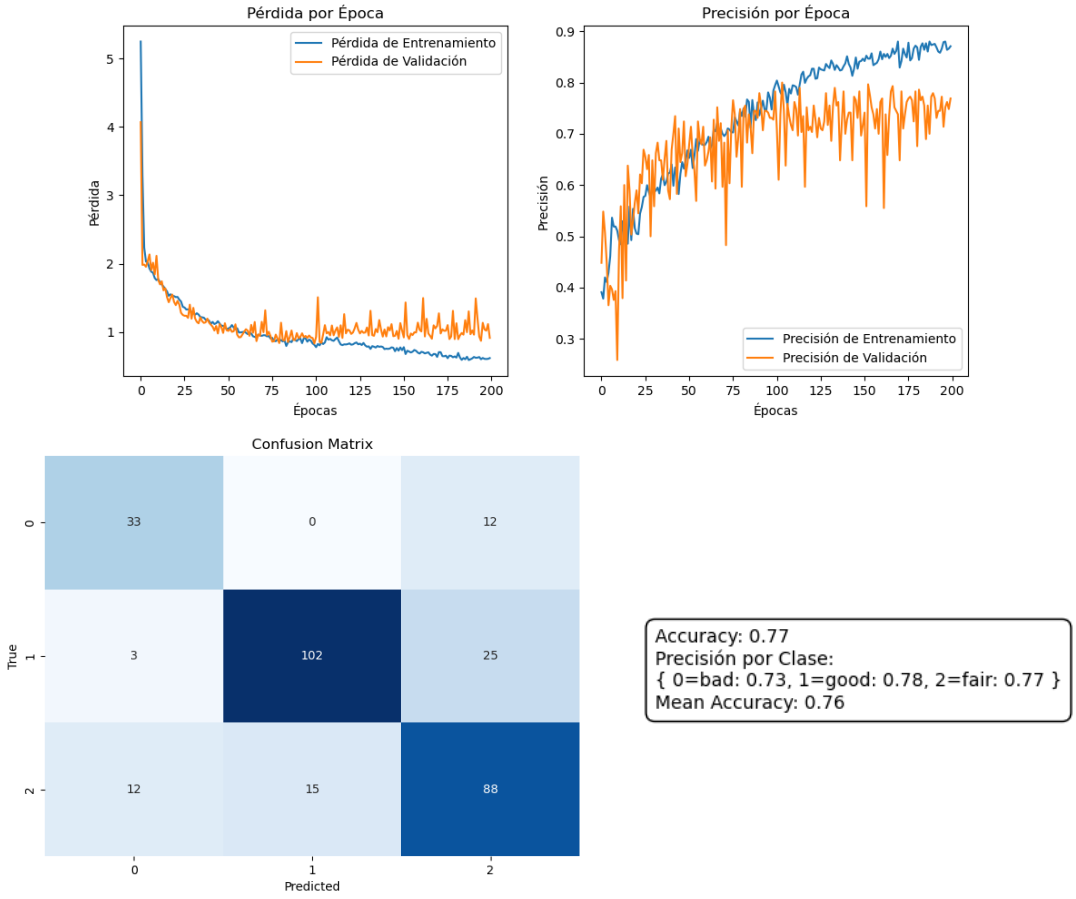


Fig. 31 Métricas Red_BGF_Long

A diferencia de la red transversal, encontramos para este modelo, Fig. 31, una mayor frecuencia de oscilaciones en la gráfica de pérdida, pero con menor amplitud, esto quiere decir que los datos o imágenes longitudinales resultan más fáciles de clasificar, de modo que el conjunto de validación es representativo del conjunto de entrenamiento. Sin embargo, este buen comportamiento se da hasta la época 100, donde se puede observar un gran pico a partir del cual la curva de validación se estabiliza en el rango [1 1.5] mientras que la curva de entrenamiento sigue su curso reduciéndose. Esto es un claro ejemplo de sobreajuste, que, aunque no sea preocupante debido a que la diferencia entre ambas curvas no es exagerada, sí que se ve reflejado en la gráfica de precisión. En esta, a partir de la época 100, la curva de validación se separa de la del entrenamiento, estabilizándose en una precisión del 70%. Por su parte la curva de entrenamiento tiene una tendencia ascendente, demostrando que a partir de esas épocas el modelo ha dejado de aprender y esta "memorizando" las imágenes de entrenamiento y no es capaz de generalizar a las de validación.

Por ello se ha elegido el mejor modelo en cuanto a precisión, el cual se observa en la gráfica que se encuentra en la época 100. Se ha graficado una matriz de confusión donde se puede ver que las tres clases se encuentran bastante balanceadas en cuanto a precisión, destacando nuevamente la clase *"Good"*.

Estos modelos, pese a que tienen una precisión de en torno al 70%, teniendo una mayor el modelo longitudinal, no es suficiente, ya que lo que interesa es que haya una alta precisión entre las imágenes *"Bad"* y el resto, de modo que las imágenes *"Fair"* se han calificado como *"Good"* para realizar un modelo binario que se centre únicamente en dos clases, facilitando así su clasificación. Esta decisión se puede tomar debido a que las imágenes *"Fair"* son imágenes válidas para su posterior segmentación y evaluación en las siguientes fases del proyecto.

6.1.3. Redes de clasificación binaria: "Bad" o "Good & Fair"

Red_BGF_Trans_Bin

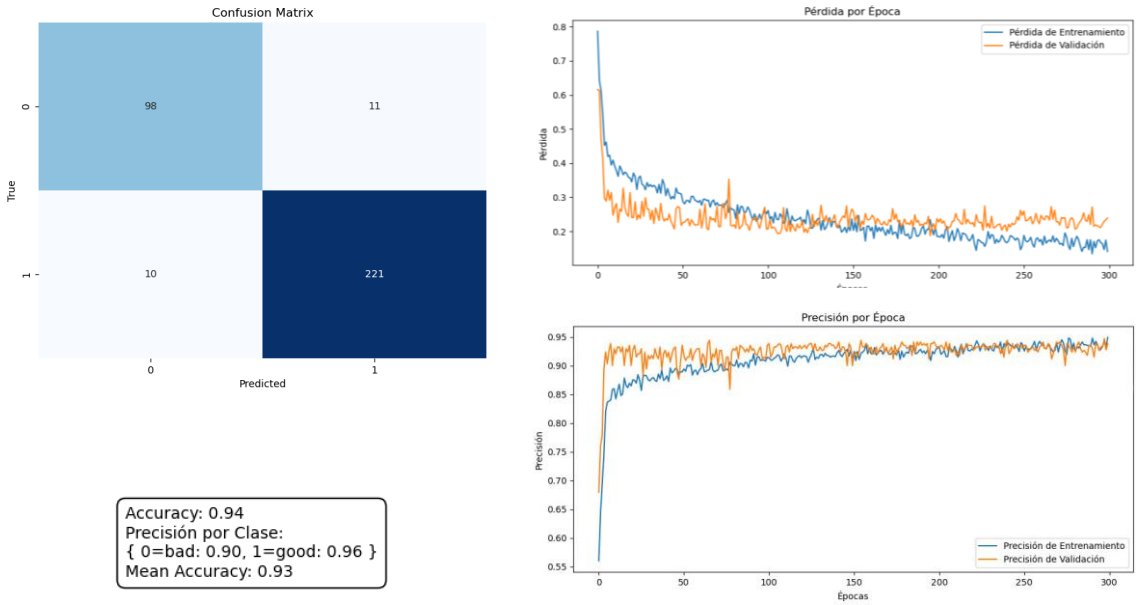


Fig. 32 Métricas Red_BGF_Trans_Bin

Se puede apreciar que la decisión de convertir la clasificación en binaria ha sido ciertamente acertada, se puede observar una notable mejoría, no solo en las curvas de entrenamiento si no en la precisión, Fig. 32.

Comenzando con las curvas de la función de pérdida, encontramos oscilaciones naturales de un entrenamiento ya que tienen una amplitud muy pequeña, y la frecuencia no es alta. Destaca también que la curva de validación en un inicio, hasta la época 125 aproximadamente, tiene menores valores de pérdida que el entrenamiento para posteriormente estabilizarse en una pérdida del 0.3 y a partir de la época 150 la curva de entrenamiento ya se separa de la validación.

La razón por la que la validación se encuentra por debajo del entrenamiento se da por diversas razones: Primero encontramos que el modelo tiene un aumento de datos, esto genera modificaciones en las imágenes lo que les añade ruido, pero únicamente se realiza a los datos del entrenamiento, por lo que hasta que el modelo aprende, los datos de entrenamiento no son representativos, es decir, es más difícil clasificar las imágenes de entrenamiento que las de validación por el ruido introducido y la variabilidad. Segundo, en las primeras etapas del entrenamiento, el modelo es incapaz de capturar patrones complejos en los datos de entrenamiento, lo que lleva a un mayor error. Sin embargo, puede ajustarse mejor a los patrones más simples presentes en el conjunto de validación desde el principio.

Por eso a partir de la época 100 ambas curvas ya se encuentran en la misma tendencia, cuando el modelo ha aprendido los patrones claves y generaliza, a partir de entonces y hasta la época 150 ambas curvas se encuentran en la misma línea, pero a partir de las 150 la curva de entrenamiento disminuye con respecto a la validación, aparece entonces, aunque aceptable, un pequeño sobreajuste, la tendencia de la validación no aparenta reducirse, por lo que el entrenamiento se corta.

En las curvas de precisión se puede observar un comportamiento similar, ya que en las primeras etapas la validación presenta mayor precisión que el entrenamiento hasta la época 150 para posteriormente juntarse y estabilizarse en una precisión del 91%.

Se ha elegido el modelo con mayor precisión entre las épocas 100 y 150 y se puede ver en la matriz de confusión la alta precisión que tiene, contando con un 90% de precisión para las imágenes *"Bad"* y 96% para las *"Good y Fair"*.

Red_BGF_Long_Bin

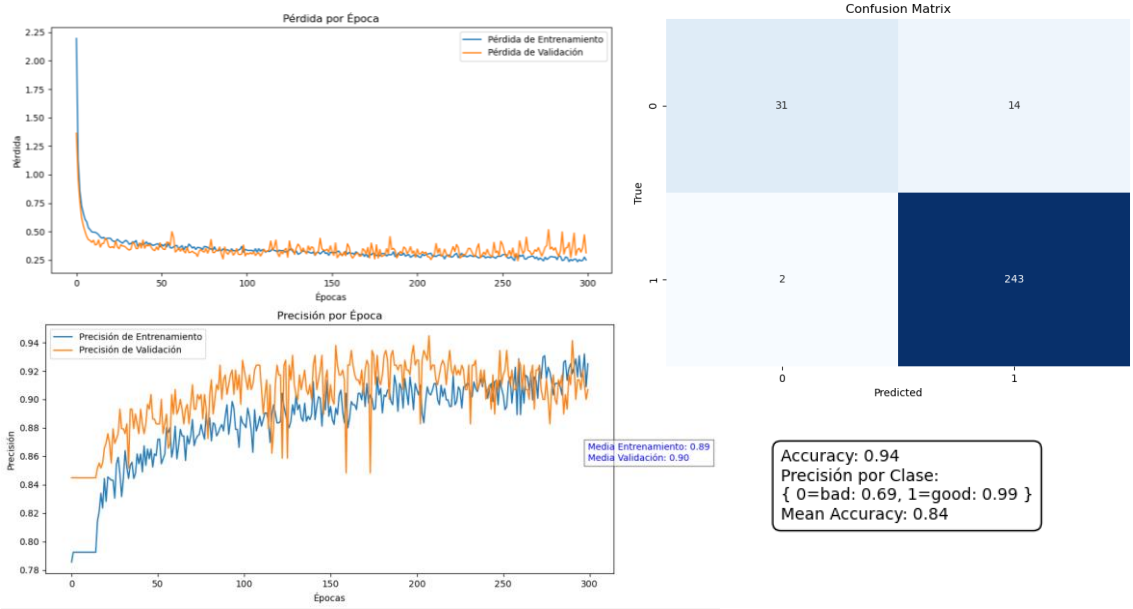


Fig. 33 Métricas Red_BGF_Long_Bin

Para el modelo longitudinal se puede observar una mejor gráfica de pérdida pues ambas curvas siguen la misma tendencia, no se encuentra sobreajuste y las oscilaciones han

reducido su frecuencia y amplitud, sin embargo, en cuanto a la precisión no se observa el mismo comportamiento, Fig. 33.

En las primeras etapas del entrenamiento, en la precisión, la curva de validación se encuentra por encima de la de entrenamiento, esto concuerda con el inicio de la gráfica de pérdida ya que se encuentra la curva de validación ligeramente por debajo de la del entrenamiento, sin embargo, una vez se han igualado ambas gráficas en la pérdida, la precisión de validación sigue por encima de la del entrenamiento. Estudiando la curva de precisión del entrenamiento se puede observar amplias oscilaciones, esto se debe a las técnicas de aumento de datos y el ruido que introducen, a esto se le añade el bajo conjunto de datos.
Pese a ello, las oscilaciones se encuentran en una precisión bastante alta, por lo que el modelo elegido cuenta con una precisión media del 85%. Teniendo una precisión del 99% para las imágenes *"Good y Fair"* (tan alta debido al desbalance) y una precisión del 69% para las imágenes *"Bad"*.

6.1.4. Redes de clasificación binaria con filtros: "Bad" o "Good" y Fair"

Red_BGF_Trans_Bin_Filtros

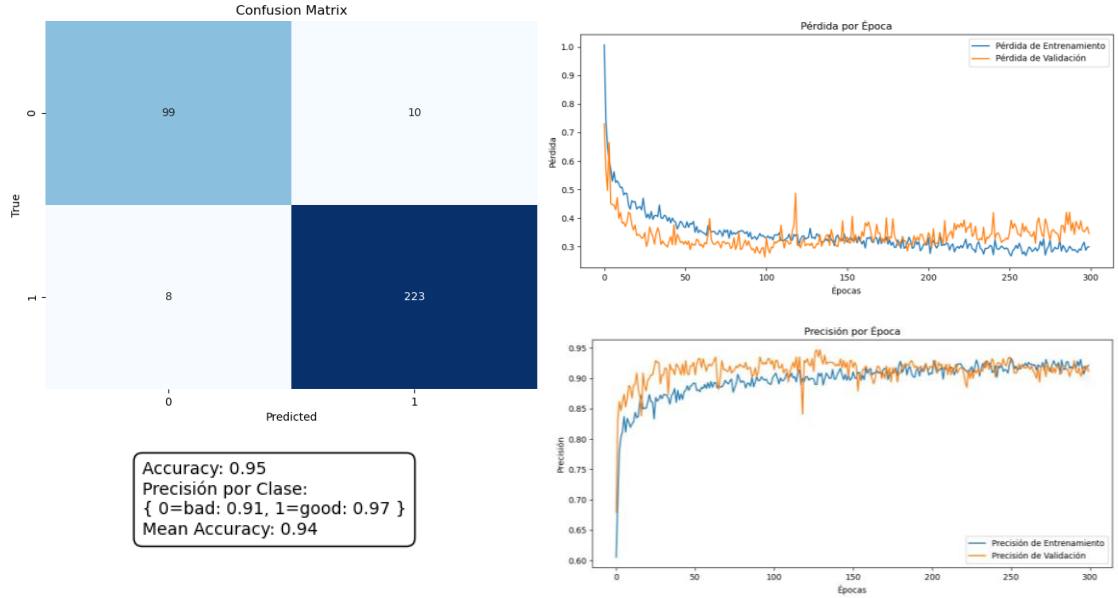


Fig. 34 Métricas Red_BGF_Trans_Bin_Filtros

Comenzando con las curvas de la función de pérdida, observamos nuevamente características oscilaciones naturales de un entrenamiento ya que tienen una amplitud muy pequeña, y la frecuencia no es alta. Se puede observar que con respecto a la pérdida del modelo transversal sin filtros (Red_BGF_Trans_Bin) ha mejorado en cuanto al sobreajuste y estabilidad, Fig. 34.
En las etapas iniciales la curva de validación es inferior a la de entrenamiento al igual que en el modelo Red_BGF_Trans_Bin, pero en esta ocasión, ambas curvas se juntan a partir de la época 100 hasta la época 200, esto implica que el modelo tarda más en sobre ajustarse, por lo que está más tiempo aprendiendo, pero al igual que ocurría con el modelo

sin filtros, el sobreajuste que aparece es muy pequeño por lo que no influye en el entrenamiento.
Este comportamiento se ve reflejado en la gráfica de precisión, donde se observa contrario a la gráfica de pérdida, que la validación en las etapas iniciales tiene mayor precisión que el entrenamiento, hasta la época 150 que comienzan a juntarse, en una precisión del 92% aproximadamente.
Se ha elegido el modelo con mayor precisión y se ha conseguido una precisión del 97% para la clase *"Good y Fair"* y 91% para la clase *"Bad",* mejorando ambas precisiones con respecto al modelo sin filtros.

Red_BGF_Long_Bin_Filtros

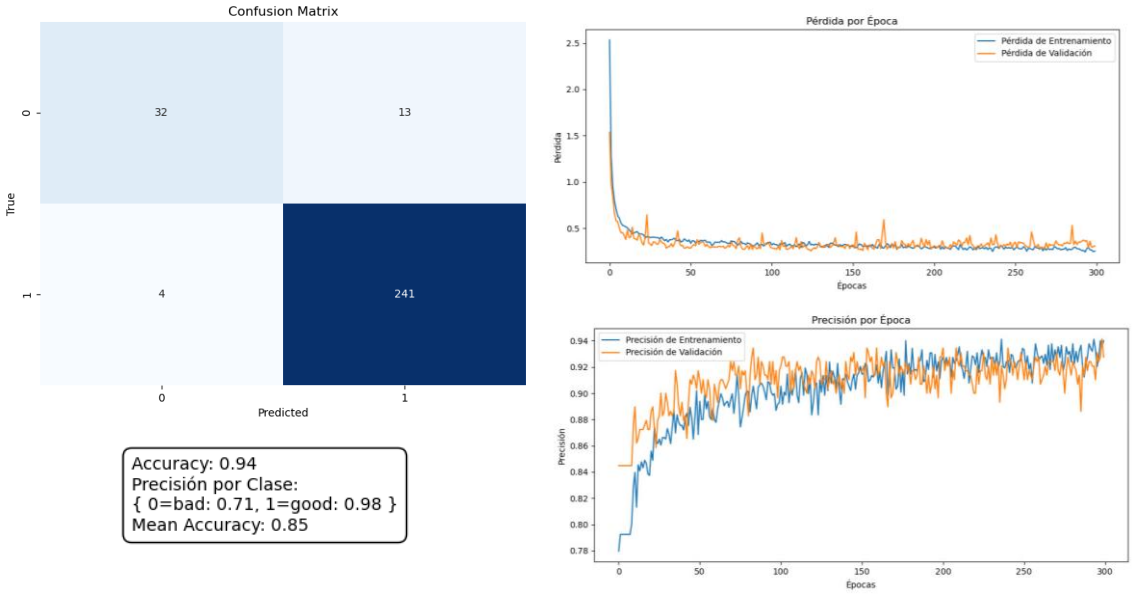


Fig. 35 Métricas Red_BGF_Long_Bin_Filtros

El modelo longitudinal con filtros muestra también una mejora con respecto al modelo sin filtros. Primero en la gráfica de función de pérdida encontramos unas curvas más estabilizadas con menos oscilaciones, tampoco se aprecia sobreajuste.
En la gráfica de precisión encontramos la misma frecuencia de oscilaciones que en la gráfica del modelo sin filtros, pero con menor amplitud, también las curvas de validación y entrenamiento se junta en épocas más tempranas hasta prácticamente el final del entrenamiento que destaca una bajada de precisión que coincide con un pico en la pérdida.

Las oscilaciones de la precisión se deben a las técnicas de aumento de datos y el ruido que introducen. A esto se le añade el bajo conjunto de datos, que al tener más ejemplos de imágenes *"Good y Fair"* que de las *"Bad",* se genera cierto sesgo y al modelo le cuesta aprender la diferencia por la baja cantidad de ejemplos. Este conjunto reducido de datos también genera que el conjunto de validación sea pequeño, por lo que el modelo tiene una mayor precisión que la del entrenamiento ya que puede clasificar las imágenes sencillas, pero cuando se introducen imágenes dudosas el modelo falla, de ahí las oscilaciones propias de la precisión de validación.

El modelo elegido, con respecto al modelo sin filtros ha reducido el sesgo, pues ha conseguido clasificar más imágenes de categoría *"Bad",* sin embargo, ha reducido su precisión en clase *"Good y Fair"* de un 99% a un 98%, un valor completamente aceptable.

6.2. Fase de Segmentación

Unet_Trans

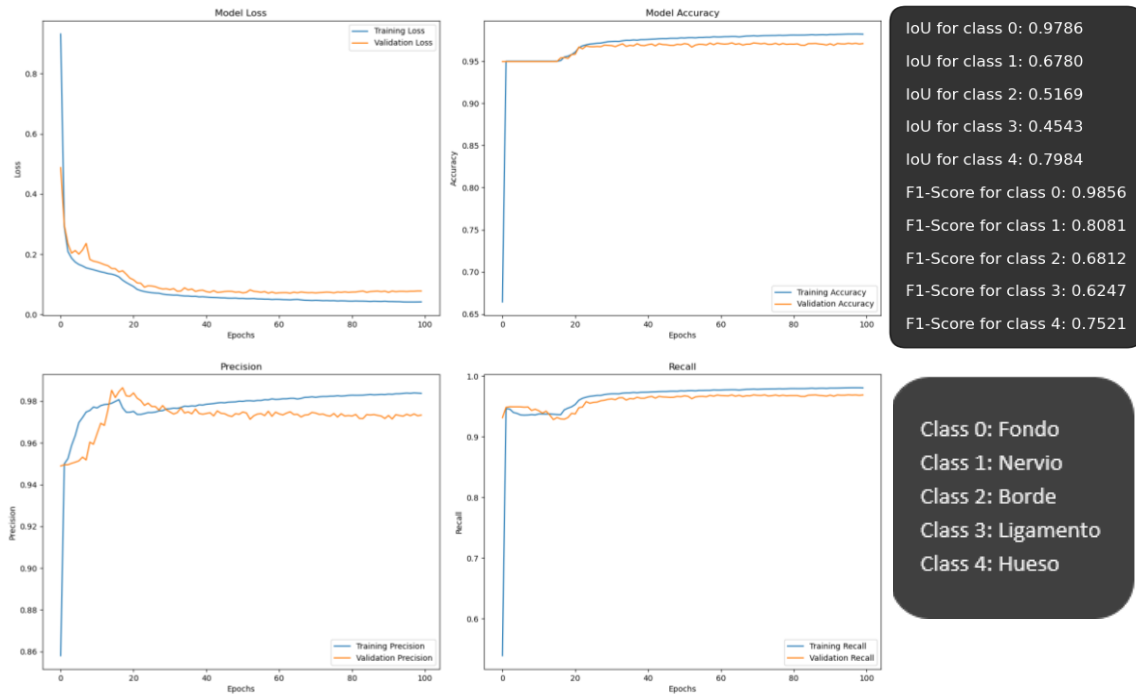


Fig. 36 Métricas Unet_Trans

Para esta fase de segmentación recordemos que se usan más gráficas debido a que se deben combinar para estudiar el rendimiento del modelo, Fig. 36.

Durante el entrenamiento, se observa que la función de pérdida disminuye consistentemente tanto para el conjunto de entrenamiento como de validación, lo que indica un aprendizaje progresivo y una buena generalización. Se puede ver también que la curva de validación no llega a juntarse con la de entrenamiento, esto no es necesariamente un signo de sobreajuste, ya que la tendencia descendente en ambas se perpetua a lo largo de las épocas, al final del entrenamiento en torno a la época 70 sí que se puede notar como la curva de validación se aleja y aumenta con respecto a la de entrenamiento, esto es un ligero sobreajuste que no interfiere en los resultados.

Las métricas *"accuracy" y "recall"* mostraron una tendencia ascendente, alcanzando cierta estabilización en valores altos por encima del 90%. Este comportamiento refleja la capacidad del modelo para clasificar correctamente los píxeles de cada clase de interés, minimizando falsos positivos y falsos negativos. Sin embargo, se identifica una ligera caída en la precisión junto con un conjunto de oscilaciones, esto es posiblemente asociado a la variabilidad intrínseca de los datos, es decir, la diferencia entre clases y junto con el sobreajuste encontrado, pese a todo la precisión encuentra valores superiores al 96%.

Dado que en esta fase existe un destacable desbalance de clases de interés en comparación con el fondo, se ha evaluado el IoU y el F1-score.
Destaca los resultados de la clase 0, con valores del 98% tanto para el IoU como el F1-score, sin embargo, esto no es un valor representativo del modelo ya que se trata del fondo y aquí encontramos el desbalance de píxeles. Para este proyecto, la clase más relevante es la 1 (nervio). El IoU obtenido es del 67%, lo que indica que el modelo identifica correctamente el nervio, aunque existe margen de mejora debido a posibles imprecisiones en la extracción de las ROI. Al realizarse a mano, y con un grosor constante para todas las ROI, existe un porcentaje de píxeles que realmente no pertenecen a la clase nervio, cosa que el modelo puede interpretar con valores inferiores de IoU. Por otra parte, encontramos un F1-score del 80%, un buen valor, lo que nos sugiere que el modelo tiene un buen balance entre precisión y *"recall"* para esta clase, es decir, el modelo es efectivo a la hora de identificar el nervio.
Para el resto de clases se tienen unos valores similares en cuento al IoU, destacando la clase 4 del hueso con un valor de 79%, para la clase 2 del borde del nervio encontramos un 51% y para la clase 3 del ligamento un 45%. No sorprende que esta clase sea la que menor valor obtenga ya que es la más difícil de detectar incluso para el ojo de nuestro experto. Además, su identificación como ROI, no es precisa y en ocasiones es necesario aproximar su localización ya que no se identifica en las imágenes. Para el F1-score, sin embargo, encontramos valores buenos para las tres clases, aunque inferiores a la clase nervio, con valores que indican un buen balance e identificación.

Unet_Long

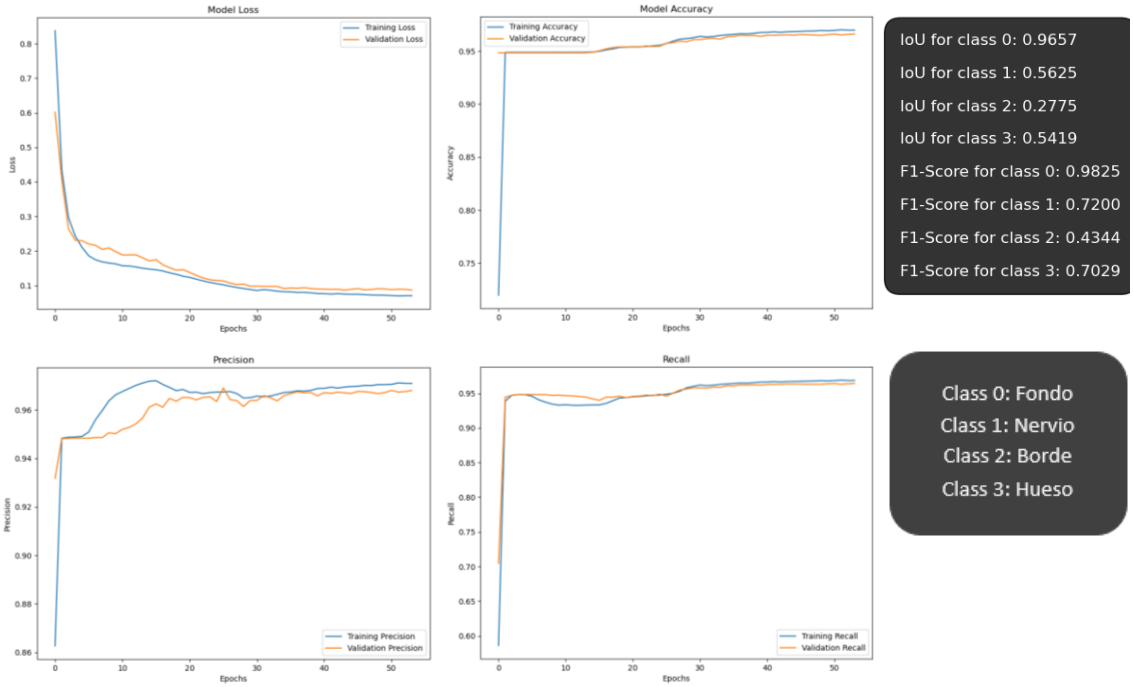


Fig. 37 Métricas Unet_Long

Para el modelo longitudinal se observa un comportamiento similar al de las imágenes transversales, ambas curvas tienen una tendencia descendente en la pérdida y por el

contrario una tendencia ascendente en el *"accuracy"* y el *"recall"*, Fig. 37. Para este modelo, el *"callback"* configurado ha detenido el entrenamiento a las 55 épocas ya que se detectó que el modelo no aprendía más, es decir, alcanzó su convergencia. Esto se ve reflejado en las gráficas ya que no se observan oscilaciones y todas se estabilizan en valores elevados. En esta ocasión no se observa ningún sobreajuste en la pérdida y por lo tanto tampoco en la precisión.
En cuanto al IoU y F1-score, nuevamente destacan los altos valores de la clase 0. Para la clase 1, del nervio tenemos un 56% de IoU y 72% de F1-score. Un valor aceptable para el IoU pero que muestra margen de mejora, esto sugiere que hay falsos positivos o falsos negativos que afectan a la métrica, sin embargo, el modelo muestra un buen balance entre precisión y *"recall"* por lo que identifica correctamente la clase nervio. La clase 2 muestra un IoU bajo debido a que, para las imágenes longitudinales, en algunas el borde del nervio está marcado enteramente y en otras el borde solo está marcado en la zona del nervio, estas diferencias se deben a que el borde es meramente orientativo y la clase importante es la del nervio.

6.3. Fase de Diagnostico

Diag_Trans

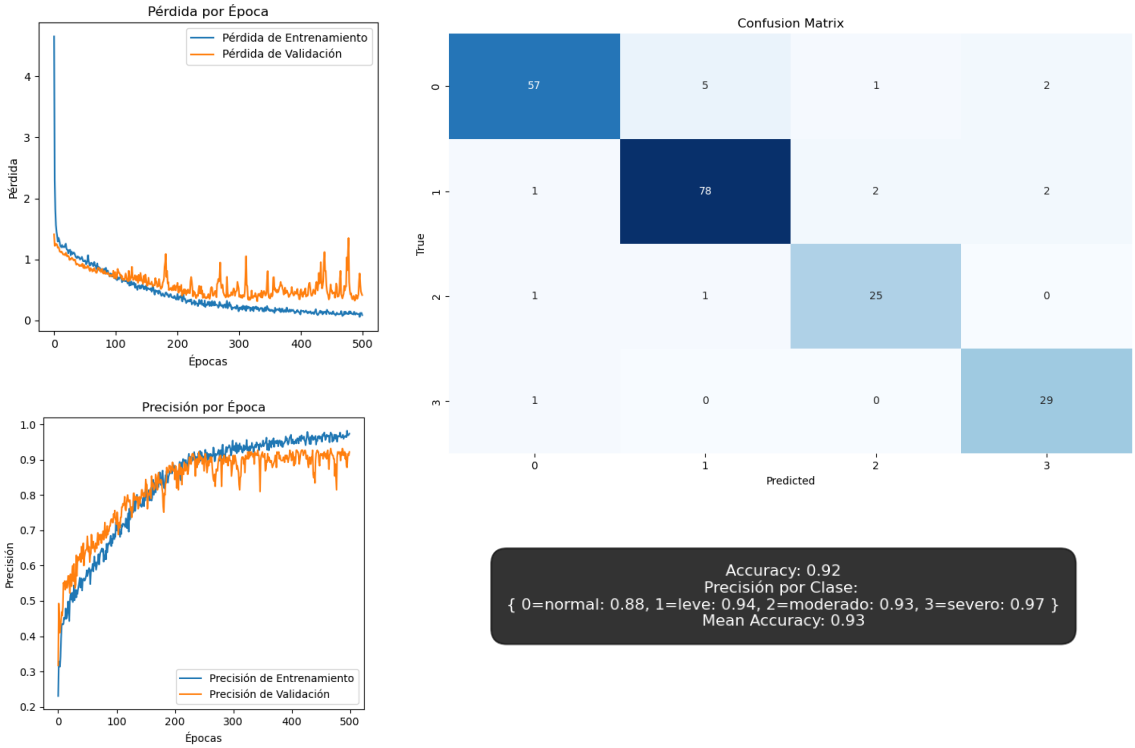


Fig. 38 Métricas Diag_Trans

En la última fase el proyecto, para el modelo transversal contamos con unas precisiones bastante altas aun teniendo en cuenta que el modelo es multicategórico.
La gráfica de pérdida en el conjunto de entrenamiento, Fig. 38, muestra una tendencia descendente constante a lo largo de todas las épocas, la validación por su parte muestra una tendencia descendente hasta la época 300 donde comienza a estabilizarse sin mostrar

mejoras, esto sugiere que el modelo ha alcanzado un punto en el que ya no generaliza para el conjunto de validación, pero sigue aprendiendo patrones simples, esto es un indicio de sobreajuste y una explicación de las oscilaciones, es decir, el modelo está aprendiendo pero tiene dificultades en algunos ejemplos para clasificarlos correctamente lo que limita su capacidad.

En la precisión tanto en entrenamiento como la validación muestran un incremento hasta la época 300 donde la validación se estabiliza en precisiones medias del 90% mientras que el entrenamiento continúa incrementando ligeramente. Esto concuerda con los signos de sobreajuste encontrados en la pérdida.

El modelo elegido se encuentra sobre la época 200, donde no existe sobreajuste y la curva ya se encuentra estabilizada, dicho modelo ha conseguido tener una precisión para el conjunto de validación de: 88% para la clase normal y 94% para la clase leve, estas dos clases son muy similares entre sí, ya que al ser la gravedad leve, el nervio no muestra indicios visibles en imágenes ecográficas de su lesión, por lo que es normal que al modelo le cueste diferenciar ambas clases, por su parte para la clase moderada y severa se cuenta con una precisión del 93% y 97% respectivamente.

Se puede observar que pese a que existe una mayor cantidad de datos de la clase normal y leve, donde el modelo se confunde más, este es capaz de diferenciar las otras dos clases.

Diag_Long

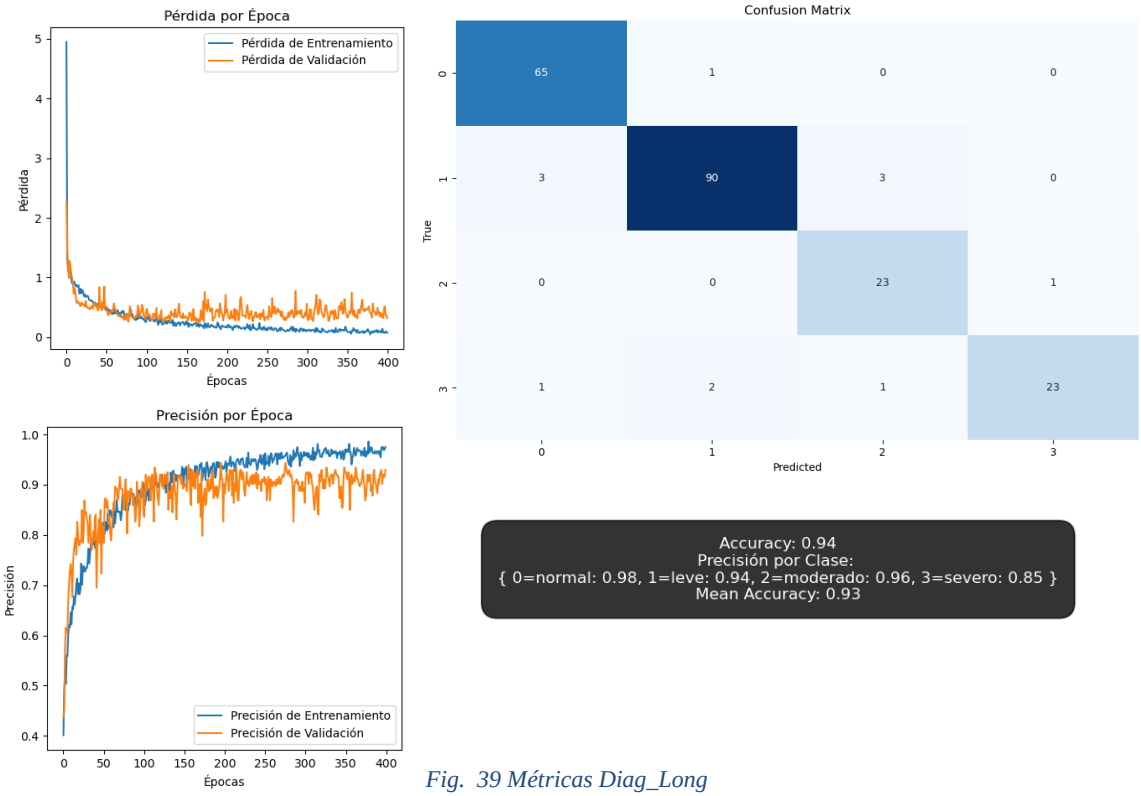


Fig. 39 Métricas Diag_Long

El modelo longitudinal por su parte muestra un comportamiento similar al modelo transversal, Fig. 39. Se muestras las características oscilaciones y las curvas descendentes en la función de pérdida, donde aproximadamente en la época 150 la función de pérdida del conjunto de validación se estabiliza mientras que la del entrenamiento continua

descendiendo para estabilizarse poco después, nuevamente encontramos un ligero sobreajuste que también se ve reflejado en la precisión donde la curva de validación se estabiliza tras unos intensos picos de caída en una precisión del 90% mientras que el entrenamiento sigue su tendencia de incremento. Se pueden observar oscilaciones más amplias en esta gráfica de precisión debido a la dificultad que muestra el modelo para clasificar las clases leve y severa.
Pese a que el modelo elegido muestra elevadas precisiones, se puede ver que la clase leve (1) y la clase severa (3) son las más difíciles de clasificar correctamente. Al igual que en el modelo anterior, las clases 2 y 3 cuentan con menos ejemplos que las clases 1 y 2 pero pese a ello el modelo consigue excelentes resultados con una precisión media del 93%.

7. Conclusiones

7.1. Fase de identificación

En esta fase se han desarrollado varios modelos, uno para clasificar el tipo de imágenes (Transversales o Longitudinales) y el resto para clasificar la calidad de las imágenes *("Bad", "Good" o "Fair").*
Para el primer caso se muestran buenos resultados con altas precisiones sobrepasando el 95% pero las gráficas muestran oscilaciones en ambas curvas, esto demuestra que el modelo tiene dificultades para clasificar cierto tipo de imágenes, esto se puede relacionar con varios motivos entre los que se encuentran el tipo de imagen ya que las imágenes *"Bad"* pueden generar errores ya que no se puede saber a qué categoría pertenece o el ruido introducido por el aumento de datos junto con el característico propio de las imágenes ecográficas.
Para el segundo tipo de códigos, se ha demostrado que la clasificación múltiple resulta más compleja y el modelo no cuenta con un conjunto suficiente de datos como para tener un modelo satisfactorio y robusto. Por ello, los modelos binarios destacan por sus buenos resultados de clasificación, aunque muestran cierto rango de mejora pese a sus excelentes resultados en la validación. Por su parte, la implementación de filtros en los modelos binarios ha ayudado a mejorar la clasificación, se demuestra así que el preprocesamiento de imágenes es un parte clave y fundamental para el entrenamiento de redes neuronales, al igual que el conjunto de datos, en este proyecto la cantidad de datos ha sido limitada por lo que para la obtención de un modelo más robusto y con mejores resultados es imprescindible un amplio conjunto de imágenes para que este pueda aprender correctamente y generalizar.

7.2. Fase de Segmentación

Esta fase ha mostrado buenos resultados para ambos modelos, aunque muestra también un rango de mejora, nuevamente con el conjunto de datos disponible se ha conseguido un robusto modelo, aunque es necesario ampliarlo.
El desbalance de clases es algo a tener en cuenta por lo que el uso de una función de pérdida con clases preponderadas ha sido clave para obtener buenos resultados de segmentación, pese a ello, la clase 3, perteneciente al ligamento ha sido la que peores

resultados ha obtenido debido a la dificultad que esta presenta. La clase del nervio, que en resumen es la más importante, ha mostrado buenos resultados de métricas e IoU por lo que su segmentación pese a que muestra margen de mejora es satisfactoria y válida para su posterior diagnóstico.

7.3. Fase de diagnóstico

Esta última fase muestra muy buenos resultados de clasificación. La red es capaz de diagnosticar 3 tipos de severidad del nervio además de aquellos que no son patológicos. El preprocesamiento en esta fase es una parte fundamental para la red y así se ha demostrado en los resultados.
Gracias a la anterior fase de segmentación se han podido realizar los recortes del nervio para entrenar la red. Pese a que los datos de segmentación no son perfectos, el procesamiento de la *"bounding box"* con un tamaño aumentado ha evitado depender completamente de la red de segmentación y la aplicación de los filtros ha ayudado a destacar zonas claves al igual que reducir el ruido restante.
Idealmente, para esta última fase lo correcto habría sido segmentar nervios de imágenes que el modelo nunca ha visto, es decir, realizar un modelo en cascada para utilizar el potencial completo de la red de segmentación, sin embargo, esto ha sido imposible con el conjunto de datos con el que se ha contado.

8. Interfaz clínica

Como fase final del proyecto, se ha diseñado e implementado una interfaz visual interactiva que permite a los profesionales no solo introducir imágenes para su análisis, sino también interpretar los resultados de manera clara e intuitiva. Esta herramienta se convierte en un puente entre los modelos desarrollados y su aplicación práctica en el diagnóstico asistido por inteligencia artificial.
Esta herramienta proporciona una visualización en tiempo real de las predicciones realizadas por el modelo, la precisión con la que ha predicho y realiza una localización del nervio. Esto facilita la interpretación clínica y ofrece un diagnóstico automatizado.
Una vez se introduzca una imagen en la interfaz, el código primeramente clasificará la imagen según sea transversal o longitudinal y posteriormente será clasificada según su calidad. Las imágenes clasificadas como *"Good" o "Fair"* son recuadradas en un marco verde junto con su predicción y precisión, mientras que las imágenes *"Bad"*, son tachadas en un recuadro rojo.

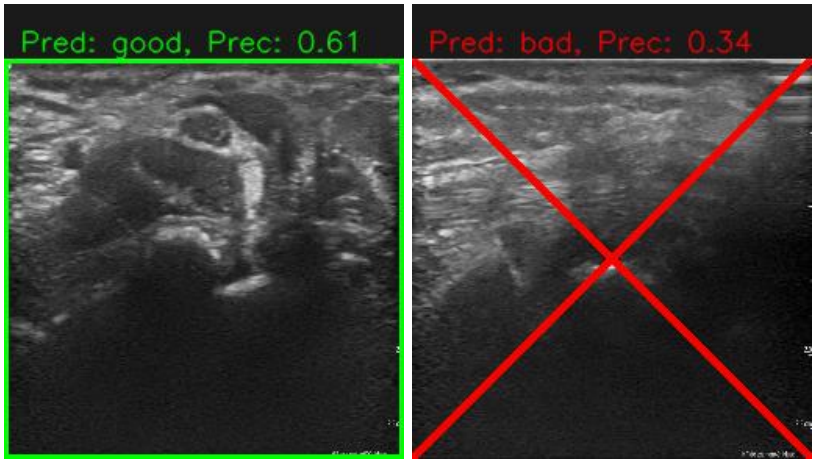


Fig. 40 Ejemplo de identificación de imágenes de la interfaz

Las imágenes que sean *"Good" o "Fair"* pasarán a ser segmentadas. La interfaz proporcionará una imagen progresiva de la segmentación realizada donde la máscara predicha será superpuesta sobre la imagen de entrada para comprobar que la segmentación es correcta.

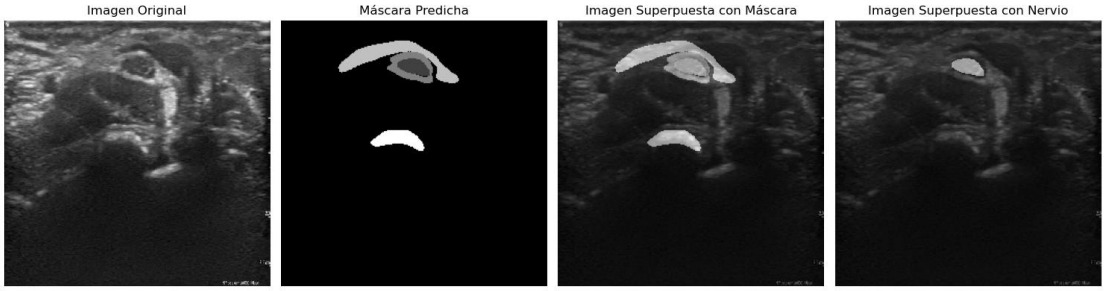


Fig. 41 Ejemplo de segmentación hecha por la interfaz

Como último paso, se proporcionará una imagen con el nervio localizado y su correspondiente diagnóstico y precisión de este.

Fig. 42 Ejemplo de diagnóstico proporcionado por la interfaz

9. Bibliografía

[1] K. Osiak *et al*, "Carpal tunnel syndrome: state-of-the-art review," *Folia Morphologica*, vol. 81, *(4)*, pp. 851–862, 2022. Available: https://journals.viamedica.pl/folia_morphologica/article/view/FM.a2021.0121. DOI: 10.5603/FM.a2021.0121.

[2] G. Goldberg *et al*, "Electrosonodiagnosis in CarpalTunnelSyndrome: A Proposed DiagnosticAlgorithm Based on an Analytic Literature Review," *PM&R*, vol. 8, *(5)*, 2016. Available: https://doi.org/10.1016/j.pmrj.2015.11.016. DOI: 10.1016/j.pmrj.2015.11.016.

[3] A. Rico Agudo, "El síndrome del túnel carpiano," *Revista Española De Cirugía Ortopédica Y Traumatología*, vol. 52, *(6)*, pp. 403–407, 2008. Available: https://www.elsevier.es/es-revista-revista-espanola-cirugia-ortopedica-traumatologia-129-articulo-el-sindrome-del-tunel-carpiano-13128704.

[4] C. Mayora Cebollero, "Deep Learning from a Mathematical Point of View," pp. 73, 2021. Available: https://deposita.unizar.es/TAZ/CIEN/2021/62721/TAZ-TFM-2021-458.pdf.

[5] MathWorks. *¿Qué son las redes neuronales convolucionales? | 3 cosas que debe saber*. Available: https://es.mathworks.com/discovery/convolutional-neural-network.html.

[6] M. A. Nielsen, "Neural Networks and Deep Learning," 2015. Available: http://neuralnetworksanddeeplearning.com.

[7] MathWorks. *Sobreajuste*. Available: https://es.mathworks.com/discovery/overfitting.html.

[8] A. Antona Castañares, "Aplicación Del Dropout a La Cuantificación De La Incertidumbre En Redes Neuronales." Escuela Técnica Superior de Ingenieros Industriales, 2020.

[9] *Implementación de una Herramienta Cuantitativa de Evaluación Ecográfica de Tendones (UltraZound quantitative Tool (UZ qTool)) en la Simulación Clínica para el Aprendizaje en el Máster de Fisioterapia Clínica Avanzada de la Universidad de Zaragoza. Available: https://simulacion-onehealth.org/ponencia/implementacion-de-una-herramienta-cuantitativa-de-evaluacion-ecografica-de-tendones-ultrazound-quantitative-tool-uz-qtool-en-la-simulacion-clinica-para-el-aprendizaje-en-el-master-de-fisioterapia/*.

[10] *Activation functions in Neural Networks*. Available: https://www.geeksforgeeks.org/activation-functions-neural-networks/.

[11] *Loss Functions in Deep Learning*. Available: https://www.geeksforgeeks.org/loss-functions-in-deep-learning/.

[12] *Optimization Rule in Deep Neural Networks*. Available: https://www.geeksforgeeks.org/optimization-rule-in-deep-neural-networks/.

[13] *tf.keras.utils.to_categorical*. Available: https://www.tensorflow.org/api_docs/python/tf/keras/utils/to_categorical.

[14] *OpenCV: Histograms - 2: Histogram Equalization*. Available: https://docs.opencv.org/4.x/d5/daf/tutorial_py_histogram_equalization.html#autotoc_md1395.

[15] Open CV. *OpenCV: Image Filtering*. Available: https://docs.opencv.org/4.x/d4/d86/group__imgproc__filter.html#ga564869aa33e58769b4469101aac458f9.

[16] Open CV. *OpenCV: Image Thresholding*. Available: https://docs.opencv.org/4.x/d7/d4d/tutorial_py_thresholding.html#autotoc_md1426

[17] Eduardo Fernandez-Moral1, Renato Martins1, Denis Wolf2, and Patrick Rives1, "A new metric for evaluating semantic segmentation:leveraging global and contour accuracy," Available: https://www.irisa.fr/lagadic/pdf/2017_ppniv_fernandezmoral.pdf.

[18] O. Ronneberger, P. Fischer and T. Brox, "U-Net: Convolutional Networks for Biomedical Image Segmentation," 2015. Available: http://arxiv.org/abs/1505.04597.

[19] Rezatofighi, Hamid and Tsoi, Nathan and Gwak, JunYoung and Sadeghian, Amir and Reid, Ian and Savarese, Silvio, "Generalized Intersection over Union," *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019. Available: https://giou.stanford.edu/.

[20] *Softmax Activation Function in Neural Networks*. Available: https://www.geeksforgeeks.org/the-role-of-softmax-in-neural-networks-detailed-explanation-and-applications/.

[21] D. Godoy, "Understanding binary cross-entropy / log loss: a visual explanation," 2022. Available: https://towardsdatascience.com/understanding-binary-cross-entropy-log-loss-a-visual-explanation-a3ac6025181a.

[22] M. Sorokin, "Categorical Cross-Entropy: Unraveling its Potentials in Multi-Class Classification," 2024. Available: https://medium.com/@vergotten/categorical-cross-entropy-unraveling-its-potentials-in-multi-class-classification-705129594a01.

[23] "The Math Behind the Adam Optimizer," Available: https://logongas.es/lib/exe/fetch.php?media=clase:iabd:pia:2eval:the_math_behind_adam_optimizer.pdf.

[24] I. Albarova-Corral *et al*, "A New Quantitative Tool for the Ultrasonographic Assessment of Tendons: A Reliability and Validity Study on the Patellar Tendon," *Diagnostics*, vol. 14, *(11)*, pp. 1067, 2024. Available: https://www.mdpi.com/2075-4418/14/11/1067. DOI: 10.3390/diagnostics14111067.

[25] F. Gu *et al*, "Research on Classification Method of Medical Ultrasound Image Processing Based on Neural Network," *Comput Intell and Neurosc*, vol. 2022, *(1)*, pp. 8912566, 2022. Available: https://doi.org/10.1155/2022/8912566. DOI: 10.1155/2022/8912566.

[26] Kumar Mohit, "A Survey on the Machine Learning Techniques for Automated Diagnosis from Ultrasound Images," *Current Medical Imaging*, vol. 20, 2024. Available: https://www.sciencedirect.com/org/science/article/pii/S1573405624000055. DOI: 10.2174/1573405620666230529112655.