UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería Industrial

DISEÑO, DESARROLLO Y VALIDACIÓN DE UN SISTEMA DE VISIÓN ARTIFICIAL PARA LA AUTOMATIZACIÓN DE ENSAYOS DE HEALTHSPAN PARA C. ELEGANS

Trabajo Fin de Máster

Máster Universitario en Ingeniería Industrial

AUTOR/A: Beato Pérez, Alejandro

Tutor/a: Sánchez Salmerón, Antonio José

CURSO ACADÉMICO: 2022/2023

AGRADECIMIENTOS

“Me gustaría agradecer a mi familia y amigos por su apoyo durante estos últimos meses, tanto en las buenas como en las malas, y por darme ese empujón cuando más lo necesitaba.

A Antonio, mi tutor. Gracias por tu paciencia y por responder a todas mis preguntas, sin importar cuántas veces te las hiciera. Has sido un gran mentor durante estos últimos años.

Y especialmente a Paula. Gracias por ser mi pilar diario y ayudarme con todo lo que hiciera falta para que pudiera sacar ese tiempo extra que siempre escasea, y por hacer este proyecto un poco más llevadero. Eres la mejor.

Muchas gracias a todos, de todo corazón. Os quiero mucho.”

RESUMEN

En el presente proyecto de final de máster se va a llevar a cabo el diseño, desarrollo y validación de un sistema de visión artificial para la automatización de ensayos de *Healthspan* para *C. elegans*.

El desarrollo del trabajo estará basado en arquitecturas de redes neuronales *"U-Net"* para resolver el complejo problema de la segmentación y en un procesamiento de imagen tradicional para extraer las características de movimiento a partir de las imágenes segmentadas. La elección de estos parámetros se realizará en función de qué tan bien caractericen el *Healthspan* de los *C. elegans.*

En el *ai2* se dispone de datasets con secuencias de imágenes de *C. elegans* de cepas con comportamientos diferentes. Este dataset se dividirá en tres subconjuntos: uno para el entrenamiento, otro para la validación y otro para test.

Para mejorar el entrenamiento de la U-Net, se van a utilizar diferentes metodologías avanzadas. Ajuste de hiperparámetros, como la tasa de aprendizaje, optimizador y la regularización de la validación. *Data Augmentation*, para aumentar el tamaño del dataset inicial. *Transfer Learning*, utilizando un modelo pre-entrenado como punto de partida, entre otras que se consideren convenientes.

En cuanto al desarrollo, se utilizará programación mediante el lenguaje Python a través de la plataforma de aprendizaje profundo *Pytorch*. Se utilizará el software *"Tierpsy Tracker"*, que realiza la extracción automática de muchos de los parámetros de movimiento de los *C. elegans* a partir de las imágenes segmentadas obtenidas con la U-Net.

Finalmente, se evaluará el trabajo desarrollado realizando una comparación entre los resultados obtenidos automáticamente y los datos etiquetados de validación. Esto se conseguirá mediante la combinación de diferentes indicadores de rendimiento, como la Matriz de Confusión, la Precisión, la Sensibilidad o el *F1-Score.*

Palabras clave: Automatización, Visión Artificial, Redes Neuronales, Deep Learning, Inteligencia Artificial, *C. elegans*, Ensayos Healthspan.

RESUM

En el present projecte de final de màster es durà a terme el disseny, desenvolupament i validació d'un sistema de visió artificial per a l'automatització d'assajos de *Healthspan* per a *C. elegan*s.

El desenvolupament del treball estarà basat en arquitectures de xarxes neuronals "*U-Net*" per a resoldre el complex problema de la segmentació i en un processament d'imatge tradicional per a extreure les característiques de moviment a partir de les imatges segmentades. La elecció d'aquests paràmetres es realitzarà en funció de com de bé caracteritzen el *Healthspan* dels *C. elegans*.

En *l'ai2* es disposa de datasets amb seqüències d'imatges de *C. elegans* de soques amb comportaments diferents. Aquest dataset es dividirà en tres subconjunts: un per a l'entrenament, un altre per a la validació i un altre per a la prova.

Per a millorar l'entrenament de la U-Net, s'utilitzaran diferents metodologies avançades. Ajust de hiperparàmetres, com la taxa d'aprenentatge, optimitzador i la regularització de la validació. *Data Augmentation*, per a augmentar la mida del dataset inicial. *Transfer Learning*, utilitzant un model pre-entrenat com a punt de partida, entre altres que es consideren convenients.

Quant al desenvolupament, s'utilitzarà programació mitjançant el llenguatge *Python* a través de la plataforma d'aprenentatge profund *Pytorch*. S'utilitzarà el programari *"Tierpsy Tracker"*, que realitza l'extracció automàtica de molts dels paràmetres de moviment dels *C. elegans* a partir de les imatges segmentades obtingudes amb la U-Net.

Finalment, s'avaluarà el treball desenvolupat realitzant una comparació entre els resultats obtinguts automàticament i les dades etiquetades de validació. Això s'aconseguirà mitjançant la combinació de diferents indicadors de rendiment, com la Matriu de Confusió, la Precisió, la Sensibilitat o el *F1-Score*.

Paraules clau: Automatització, Visió Artificial, Xarxes Neuronals, Deep Learning, Intel·ligència Artificial, *C. elegans*, Assajos Healthspan.

SUMMARY

In this master's final project, the design, development, and validation of a computer vision system for the automation of *Healthspan* tests for *C. elegans* will be carried out.

The development of the project will be based on *"U-Net"* neural networks architectures to solve the complex problem of segmentation and traditional image processing techniques to extract the motion characteristics from the segmented images. The choice of these parameters will be based on how well they characterize the *Healthspan* of *C. elegans*.

The *ai2* has at its disposal datasets with image sequences of *C. elegans* from different strains with different behaviours. This dataset will be divided into three subsets: one for training, one for validation and one for testing.

To improve the training of the U-Net*,* different advanced methodologies will be used. Hyperparameter tuning, such as learning rate, optimizer, and validation regularization, will be performed. *Data Augmentation* will be applied to increase the size of the initial dataset. Transfer learning will be utilized by using a pre-trained model as a starting point, among other relevant techniques considered.

For the development of the project, programming will be done using the *Python* language and the deep learning platform *Pytorch*. The software "*Tierpsy Tracker"* will be used, which automatically extracts many of the motion parameters of the *C. elegans* given the segmented images obtained with the U-Net*.*

Finally, the developed work will be evaluated by comparing the results obtained automatically with the labelled validation data. This will be accomplished by a combination of different performance indicators, such as Confusion Matrix, Accuracy, Sensitivity, or *F1-Score*.

Keywords: Automation, Computer Vision, Neural Networks, Deep Learning, Artificial Intelligence, *C. elegans*, Healthspan Test

ÍNDICE

TABLA DE ILUSTRACIONES

ÍNDICE DE TABLAS

ÍNDICE DE ECUACIONES

1. INTRODUCCIÓN

1.1. Descripción y motivación

En el presente Trabajo de Fin de Máster, se aborda el diseño, desarrollo y validación de un sistema de visión artificial destinado a la automatización de ensayos de *Healthspan* en *Caenorhabditis elegans (C. elegans)*. Para ello, el grueso del trabajo se centrará en diseñar un modelo de red neuronal profunda utilizando la arquitectura U-Net para realizar la segmentación binaria de imágenes que contienen a estos nematodos.

Una vez obtenidas las imágenes segmentadas, se implementarán algoritmos para la extracción de características de movimiento de los *C. elegans*. Este procedimiento se realizará mediante dos enfoques; uno desarrollado específicamente para este trabajo, y otro utilizando el software *Tierpsy Tracker*.

Finalmente, se evaluará la eficacia de la segmentación del modelo U-Net, mediante la comparación de sus predicciones con las máscaras originales. Además, se contrastarán los resultados de la extracción de las características de movilidad obtenida por ambos métodos.

Este proyecto surge de un interés combinado en la automatización de procesos, la visión artificial, la programación y, más recientemente, en la inteligencia artificial y las redes neuronales computacionales profundas. Estas áreas no solo son fascinantes desde una perspectiva personal, académica y profesional, sino que también ofrecen un enorme potencial para impulsar avances significativos en la ciencia, siendo en este caso concretamente en campos tan críticos como la investigación biomédica.

Adicionalmente, este proyecto representa una oportunidad para continuar y expandir la investigación iniciada en el Trabajo de Fin de Grado sobre la implementación de un control automático para la captura de imágenes de *C. elegans* mediante realimentación visual.

1.2. Objetivos

Se plantean como objetivos generales del trabajo el diseño, desarrollo y entrenamiento de un modelo de red neuronal con arquitectura U-Net para la segmentación automática de imágenes de *C. elegans*, así como la evaluación y extracción de características de movilidad de estos.

Para llevar a cabo los objetivos generales, se plantea resolver los siguientes objetivos específicos:

- El aprendizaje de terminología básica de redes neuronales, como convoluciones, "*Max Pooling*", entrenamiento, validación y funciones de pérdida.
- El estudio en profundidad de la estructura de red neuronal convolucional U-Net.
- La generación de los datasets de entrenamiento, validación y test a partir de los vídeos proporcionados como punto de partida. También a partir de estos la creación de *DataLoaders*.
- La implementación de técnicas tanto consolidadas como novedosas para mejorar el rendimiento del modelo a entrenar, tales como la *Data Augmentation*, Regularización y el método de super convergencia.
- La medición del rendimiento del modelo entrenado mediante métricas de evaluación objetivas, así como comparación visual de las predicciones con las máscaras reales.

- El análisis y optimización de los resultados obtenidos con diferentes modelos, hiperparámetros y funciones de pérdida, con el fin conseguir el mejor resultado posible.
- La comparación de los resultados de la extracción de características de movimiento de los *C. elegans* mediante los diferentes métodos utilizados.

1.3. Estado del arte y antecedentes

El *C. elegans* ha sido un organismo modelo extensamente utilizado en la investigación biomédica durante las últimas décadas [14]. Su importancia se debe a varias razones, como su biología simple y fácil de estudiar, así como las similitudes que comparte con los humanos en aspectos como genes, procesos celulares y vías metabólicas. Es por esto por lo que se han utilizado durante mucho tiempo en áreas de investigación como la genética, enfermedades neurodegenerativas, envejecimiento [15, 16] y cáncer [17, 18].

Los ensayos en *C. elegans* de *Healthspan [21]* (Calidad de vida), *Lifespan [19, 20]* (Longevidad) y *Quimiotaxis* [22, 23] (Respuesta a estímulos químicos), entre otros, pueden ofrecer información valiosa sobre la salud y enfermedades humanas, gracias a las similitudes genéticas y celulares entre ambos.

Anteriormente, estos ensayos se realizaban de una forma muy manual, lenta y tediosa por trabajadores especializados, lo que ralentizaba considerablemente el posible descubrimiento de avances. Pero en los últimos años, se han comenzado a utilizar técnicas de captura y procesamiento de imágenes mucho más automatizadas con técnicas de visión por computador [24, 26].

Hoy en día, se están abriendo nuevas vías de investigación para automatizar el procesamiento de las imágenes, como mediante el uso del aprendizaje profundo, que mejora considerablemente el rendimiento y la competitividad de este proceso. En concreto, se utilizan modelos derivados de la arquitectura U-Net, como se va a realizar en el presente proyecto.

El procesamiento de imágenes de *Caenorhabditis elegans* (*C. elegans*) mediante aprendizaje profundo es un campo en rápido crecimiento que ha demostrado ser fundamental para la investigación y análisis de este organismo modelo. En el Instituto de Automática e Informática Industrial (ai2-UPV), durante los últimos cinco años, se han desarrollado diversos sistemas de adquisición de imágenes [1, 3, 5, 8] y técnicas automáticas de procesamiento de imágenes [4, 7, 9, 10, 12, 13] que permiten automatizar los ensayos de *Lifespan* [2, 6, 11] o *Healthspan.*

1.4. Punto de partida

Para la realización del proyecto, se parte de un dataset que contiene, por una parte, diferentes vídeos de *C. elegans* individuales, pertenecientes a diferentes cepas y ensayos. Por otra parte, para cada vídeo se dispone un fichero de datos hdf5 que contiene toda la información en cuanto a características de movimiento calculadas para este, calculadas a cada fotograma, así como los valores medios de cada característica de todo el experimento. Además, el fichero incluye las coordenadas del esqueleto del gusano y sus contornos.

Toda esta información se obtuvo tras analizar con Tierpsy Tracker diferentes experimentos a partir de vídeos de la placa Petri completa. Es el propio software el que genera los vídeos y archivos hdf5 de los gusanos seleccionados.

Adicionalmente, se dispone del propio Software Tierpsy Tracker con todo su código fuente. Este Software será utilizado para calcular las características de movilidad de las imágenes segmentadas por la red neuronal al final del proyecto.

1.5. Software utilizado

Se ha utilizado el siguiente Software, cada uno de ellos esencial para diferentes aspectos proyecto:

Python

Se ha utilizado el lenguaje de programación de alto nivel Python para la totalidad del desarrollo del proyecto. Es la base para la implementación de los algoritmos y modelos de aprendizaje profundo.

Figura 1. Icono Python.

PyTorch

Biblioteca de aprendizaje profundo para Python, utilizada para diseñar, entrenar y validar el modelo de red neuronal basado en la arquitectura U-Net utilizado.

Figura 2. Icono PyTorch.

OpenCV

Biblioteca de visión artificial utilizada para el procesamiento de imágenes, la creación de los datasets y la extracción de las características de movimiento de los *C. elegans.*

Figura 3. Icono OpenCV.

Visual Studio Code (VS Code)

Editor de código fuente (IDE) desarrollado por Microsoft, utilizado para escribir, depurar y gestionar el código del proyecto.

Figura 4. Icono VSCode.

Google Colab y Kaggle

Ambas son plataformas para programación de ciencia de datos y aprendizaje automático con un entorno de *Jupyter Notebook* basado en la nube. Permite el uso de hardware especializado, como GPUs, esenciales para lograr el entrenamiento y validación del modelo de red neuronal en un tiempo razonable.

Figura 5. Iconos Google Colab y Kaggle.

Se han utilizado alternativamente para diseñar, entrenar y validar el modelo de U-Net diseñado, ya que no se disponía de una GPU compatible con *Cuda* en el ordenador personal. *Cuda* es una tecnología de computación en paralelo desarrollada por *NVIDIA* que permite utilizar la GPU para la aceleración de cálculos matemáticos complejos, como los requeridos para el entrenamiento del modelo y es compatible con la biblioteca *PyTorch*.

Ya que el acceso al hardware como las GPUs era limitado con la versión gratuita, se han tenido que utilizar ambas plataformas de manera alternativa para satisfacer los requerimientos computacionales necesarios para entrenar el modelo.

Tierpsy Tracker

Tierpsy Tracker es un software para el análisis de todo tipo de características de movilidad para pequeños gusanos como los *C. elegans*, desarrollado en el *Imperial College of London* [26].

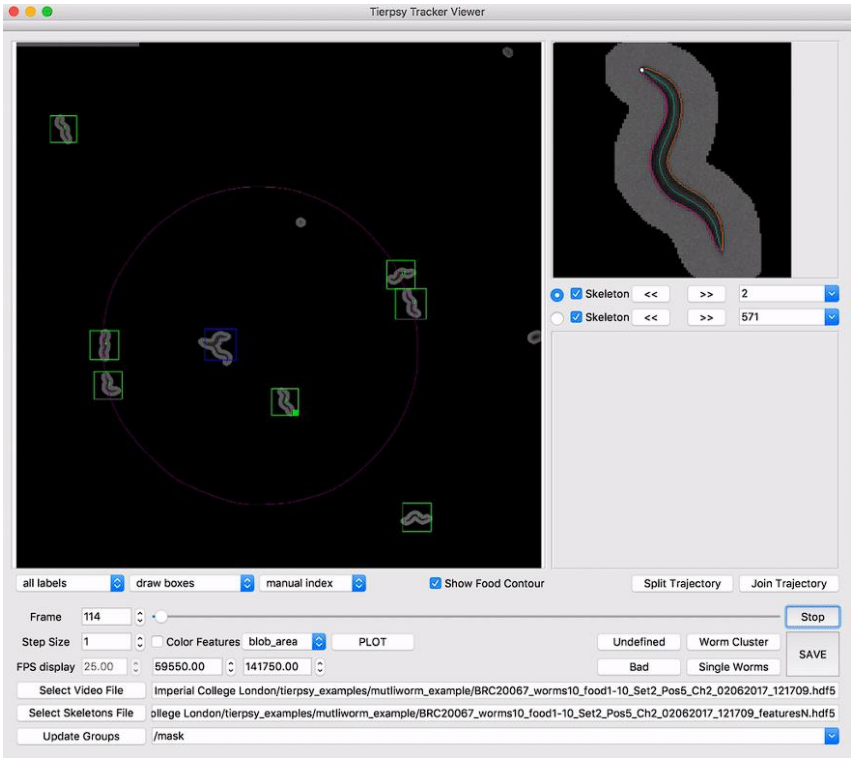

Figura 6. Interfaz gráfica Tierpsy Tracker. (https://github.com/Tierpsy/tierpsy-tracker)

Se ha utilizado para la extracción de características de movimiento de los gusanos del dataset inicial, para su análisis y comparación con los otros métodos implementados.

HDFView

Herramienta para visualizar archivos en formato HDF5, formato comúnmente utilizado en aplicaciones de ciencia de datos. Utilizada para inspeccionar los datos de salida tras el análisis con Tierpsy Tracker, de forma sencilla y visual.

Figura 7. Icono HDFView.

2. Descripción del proyecto

2.1. Ensayos de Healthspan en *C. elegans*

Los ensayos de *Healthspan* en *C. elegans* son estudios diseñados para evaluar el estado de salud de estos. Se mide cual es la calidad de vida y la longevidad de estos nematodos. Estos ensayos son especialmente importantes en la investigación biomédica para el desarrollo de fármacos ante enfermedades neurodegenerativas.

Los ensayos de *Healthspan* sobre *C. elegans* se realizan mediante la observación de estos a lo largo de su ciclo de vida bajo diferentes condiciones experimentales, como puede ser la exposición a sustancias químicas o fármacos, la alimentación u otros cambios en el entorno del experimento. Estos nematodos se colocan sobre placas Petri, y se observan utilizando técnicas de visión por computadora junto a microscopía automatizadas para evaluar su comportamiento.

2.2. Segmentación de las imágenes obtenidas

Una vez obtenidas las secuencias de imágenes o vídeos tras el ensayo de *Healthspan,* se debe proceder a la segmentación de estas, para poder separar el nematodo, que es el objeto de interés, del fondo de la placa Petri y otros restos de comida o fármacos que haya en la imagen.

Para esto, se van a utilizar distintas variaciones de modelos de redes neuronales profundas bajo la arquitectura U-Net, especializadas en la segmentación de imágenes. Se entrenarán los modelos bajo diferentes configuraciones de hiperparámetros hasta obtener un resultado óptimo.

2.3. Características de movilidad de los *C. elegans*

Una de las métricas más importantes en los ensayos de *Healthspan* es la movilidad del nematodo. La movilidad se relaciona directamente con el estado de salud y longevidad de los *C. elegans* de forma cuantificable y fiable [25].

Para evaluar la movilidad, se pueden calcular una serie de características, entre las que destacan por su utilidad las siguientes:

Velocidad de desplazamiento y giro

Se miden las velocidades de desplazamiento y giro en diferentes partes del cuerpo del *C. elegans.* Se utiliza el signo positivo si se está moviendo hacia delante y el negativo si lo hace hacia detrás. Una velocidad media y máxima elevada puede suponer un signo de buena salud, en cambio una velocidad lenta puede suponer un deterioro de la salud del nematodo o envejecimiento.

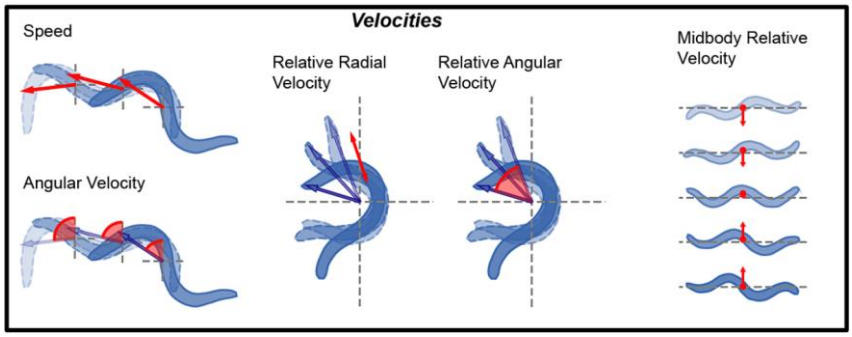

Figura 8. Análisis de velocidades de los C. elegans [25]

Análisis de trayectorias

El análisis de trayectorias sigue el movimiento que ha realizado el *C.elegans* durante el ensayo realizado. Este también puede proporcionar información valiosa sobre su comportamiento y estado de salud. Una trayectoria errática, con patrones de movimientos raros, exceso de movimientos marcha atrás, o movimientos circulares podrían ser indicativos de un deterioro del estado de salud de este.

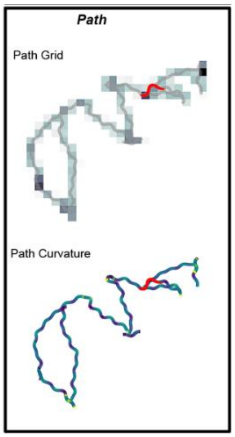

Figura 9. Análisis de trayectorias de los C. elegans [25]

Tiempo parado

El tiempo que un *C. elegans* pasa sin moverse también es un indicador importante. Una alta tasa de inactividad puede ser un signo de debilidad o enfermedad, especialmente si en etapas anteriores de su vida no actuaba de la misma manera.

Partes del *C. elegans* y Características morfológicas

El área, longitud y anchura del *C. elegans* pueden cambiar según el estado de salud o la edad de este. Por ejemplo, un aumento del área podría suponer hinchazón del cuerpo por problemas de salud, mientras que cambios en la longitud pueden indicar envejecimiento.

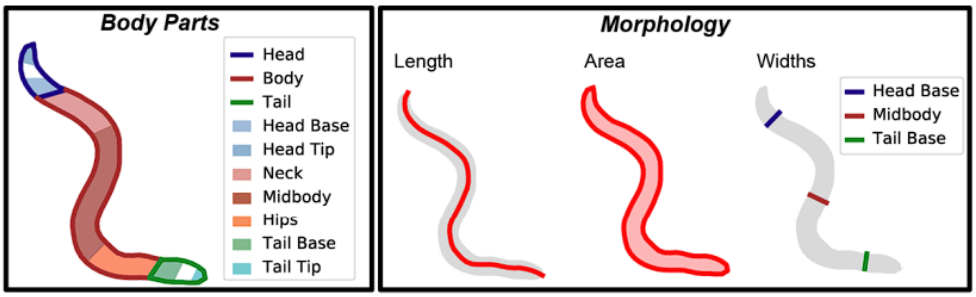

Figura 10. Morfología y partes de los C. elegans. [25]

Posturas

Las diferentes posturas que adopta el *C. elegans* también son importantes. Un elevado número de posturas en forma de "U" o "Omega" también pueden ser indicativos de su nivel de actividad, salud y edad de estos.

Una manera de medir la postura es con características como la "*Quirkiness*", dependiente de la relación entre el eje menor y mayor del rectángulo mínimo que encuadra al Nematodo.

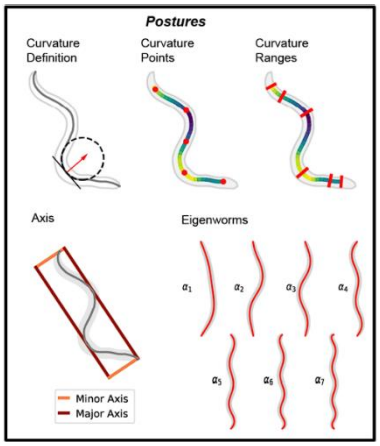

Figura 11. Análisis de posturas de los C. elegans. [25]

Amplitud y frecuencia de onda

Finalmente, la amplitud y la frecuencia de ondas que produce el C. elegans al moverse pueden dar signos de debilidad muscular o problemas neuromusculares, que puede ser común en nematodos envejecidos.

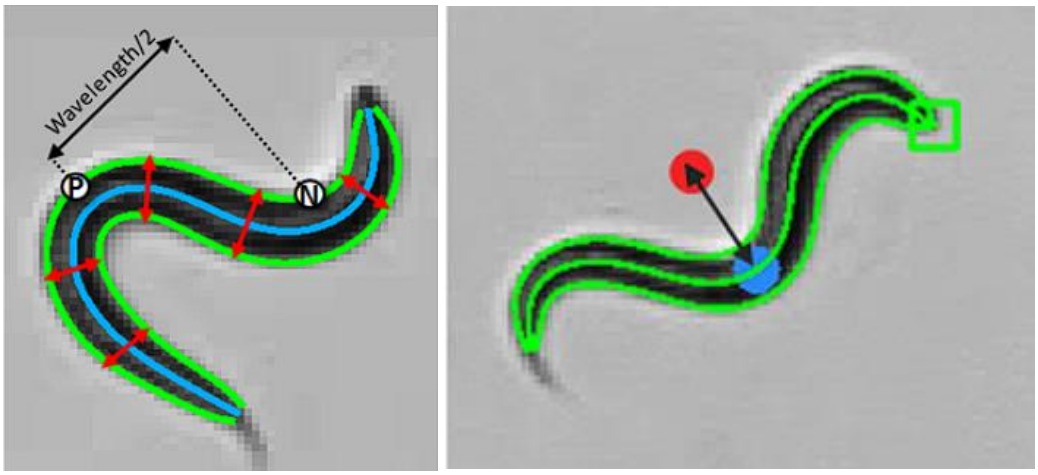

Figura 12. Longitud de Onda y Amplitud descrita por el movimiento de los C. elegans (https://www.mbfbioscience.com/help/wormlab/Content/Analyses/position%20&%20speed/trackSummary.htm)

Una vez calculadas estas características de movilidad a lo largo del ensayo, se pueden realizar análisis estadísticos para conocer con exactitud el estado de vida de los nematodos. Se aplican técnicas como la regresión lineal, el análisis por componentes principales (PCA), técnicas de aprendizaje automático y análisis de series temporales. Esto logra obtener una visión más completa y precisa del estado de Healthspan de los C. elegans, lo que permite obtener resultados significativos con más facilidad para la investigación, como en el descubrimiento de fármacos para enfermedades neurodegenerativas.

3. OBTENCIÓN DEL DATASET DE TRABAJO

La siguiente tarea a realizar es la obtención del dataset de trabajo, donde se almacenarán de manera ordenada todos los pares de imágenes originales y máscaras (o *"Ground Truth")* que contienen el *C. elegans* ya segmentado y separado del *background*.

Esta parte del proyecto ha sido decisiva, ya que dependiendo del enfoque desde el cual se llevará a cabo, podía cambiar la intención y finalidad del proyecto en gran medida.

Una vez obtenido, se han cargado los datos en *PyTorch* mediante un *DataLoader*, se han aplicado las transformaciones necesarias según las exigencias del modelo que se fuera a utilizar, y se ha aplicado el método de *Data Augmentation*.

3.1. Diferentes aproximaciones

Para realizar este dataset inicial, se tenían varias opciones en función del origen de datos a utilizar para generarlo. O bien a partir de los archivos *hdf5*, o bien a partir de los vídeos de los ensayos disponibles.

Uno de los métodos iniciales fue buscar de los datos en formato *hdf5* iniciales si había información suficiente para regenerar las máscaras de los *C. elegans*, o al menos los esqueletos a partir de su contenido. Para leer de una manera más visual la información que contenía, se utilizó el software *"HDFView"* (Figura 13)

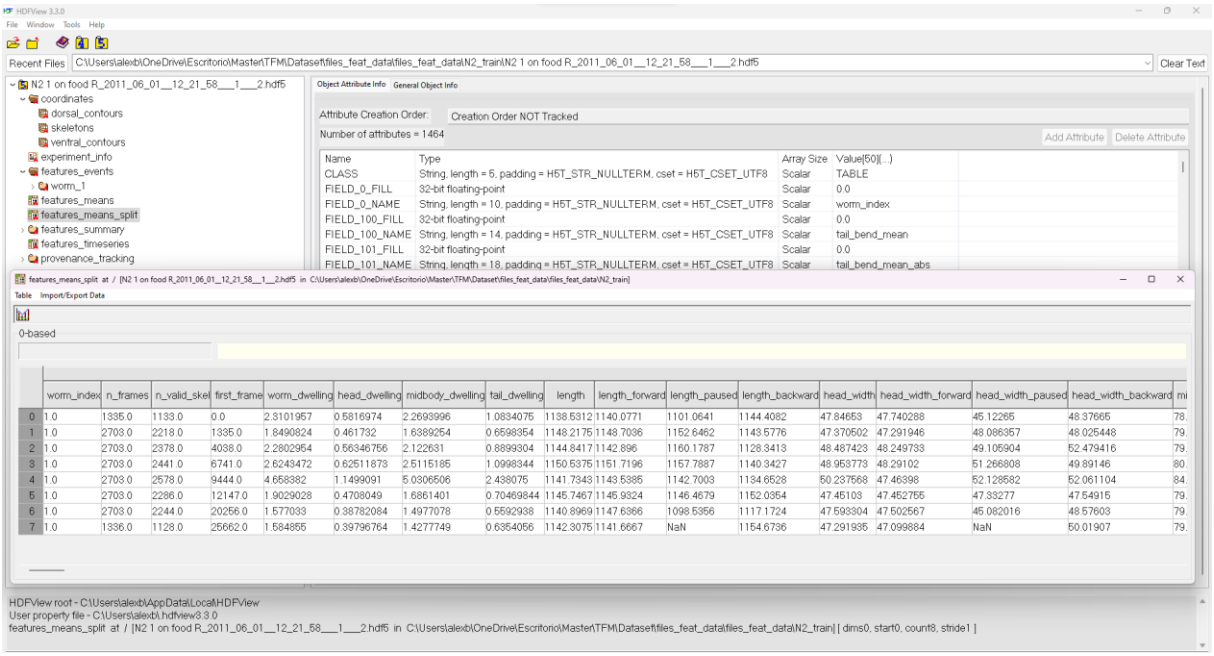

Figura 13. Vista del archivo hdf5 con HDFView.

Dentro no se encontró ninguna información relevante a los píxeles que formaban las máscaras, aunque sí que se encontraron las coordenadas de los esqueletos de los *C. elegans*. A partir de estas y mediante un script de *Python* fueron representadas (Figura 14).

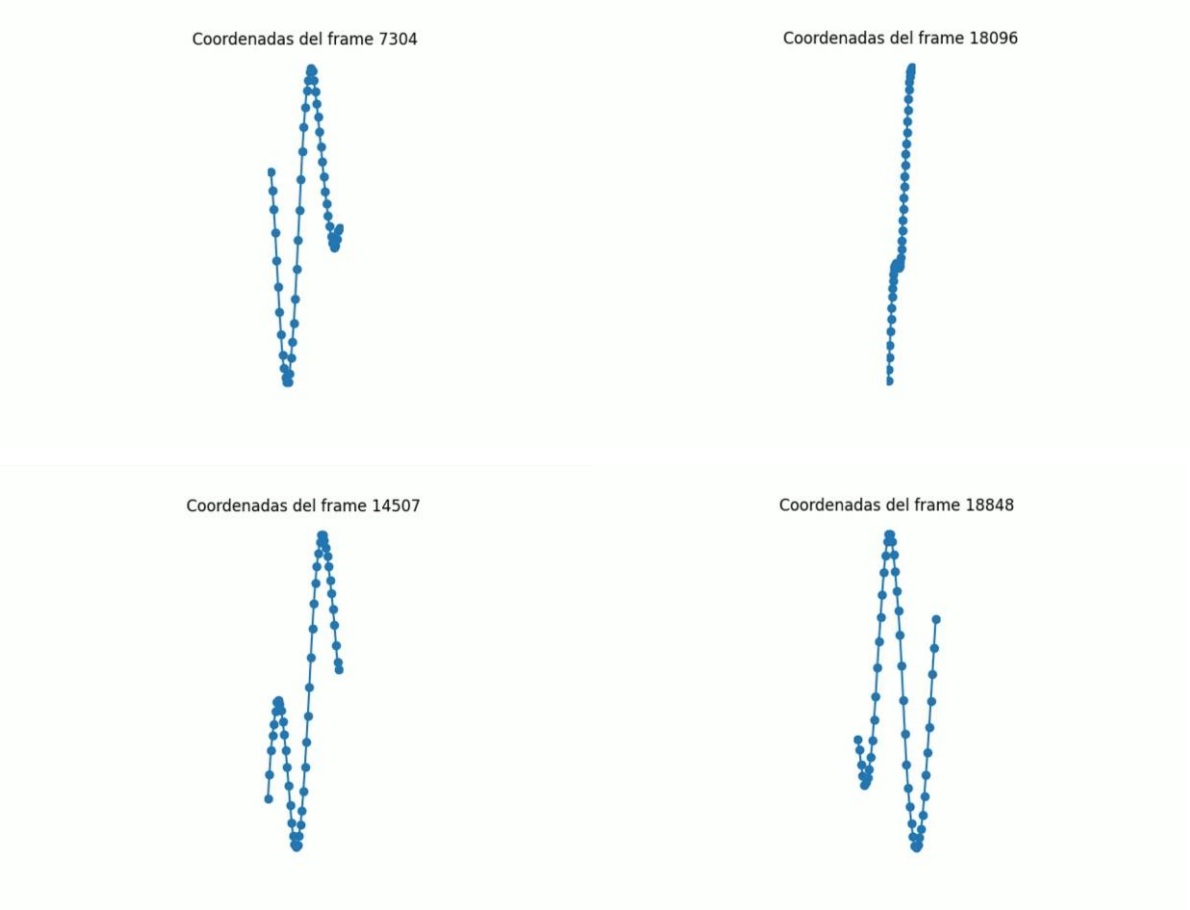

Figura 14. Coordenadas de los esqueletos representadas con Matplotlib

A continuación, se buscó la posibilidad de reconstruir el *C. elegans* completo mediante el esqueleto y operaciones de dilatación consecutivas. Pero finalmente, se descartó la idea por ser demasiado compleja y por no ser una solución aproximada a la forma que tendría un *C. elegans* en la realidad, lo que llevaría a entrenar al modelo de forma errónea, y, por lo tanto, a obtener resultados erróneos con datos reales.

3.2. Segmentación mediante métodos de visión artificial clásicos

Es por esto por lo que se decidió tomar la segunda opción, que es obtener las máscaras de los *C. elegans* mediante métodos de visión artificial convencionales utilizando la librería de *OpenCV*. Partiendo de cada uno de los frames de los videos proporcionados como punto de partida (Figura 15), se utilizaron diferentes metodologías para separar el *background* del objeto de interés

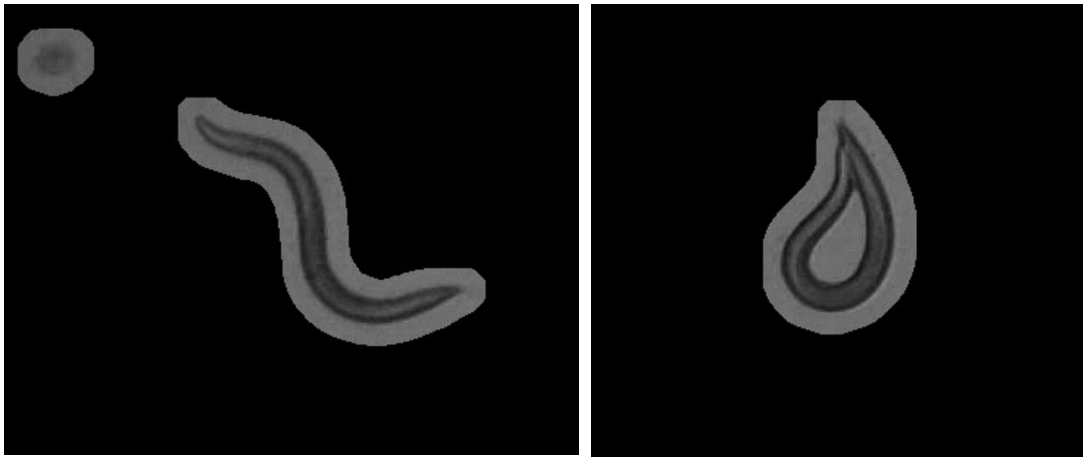

Figura 15. Frames de los videos de partida originales.

Para obtener la máscara del objeto de interés, se aplicó un *threshold* fijo a la imagen para eliminar tanto el fondo negro como el propio *C. elegans.* Podría parecer que este no es el camino correcto y que se debería realizar el *threshold* inverso para eliminar la zona iluminada de la imagen, pero hacerlo de esta manera es importante para solucionar uno de los problemas que surgen a continuación.

En segundo lugar, se realizó la búsqueda de contornos de la imagen resultante (Figura 16), generándose además una jerarquía con los sucesivos contornos internos de la imagen. Es con estos contornos internos con los que, al rellenarlos, generamos la máscara final deseada (Máscara 1).

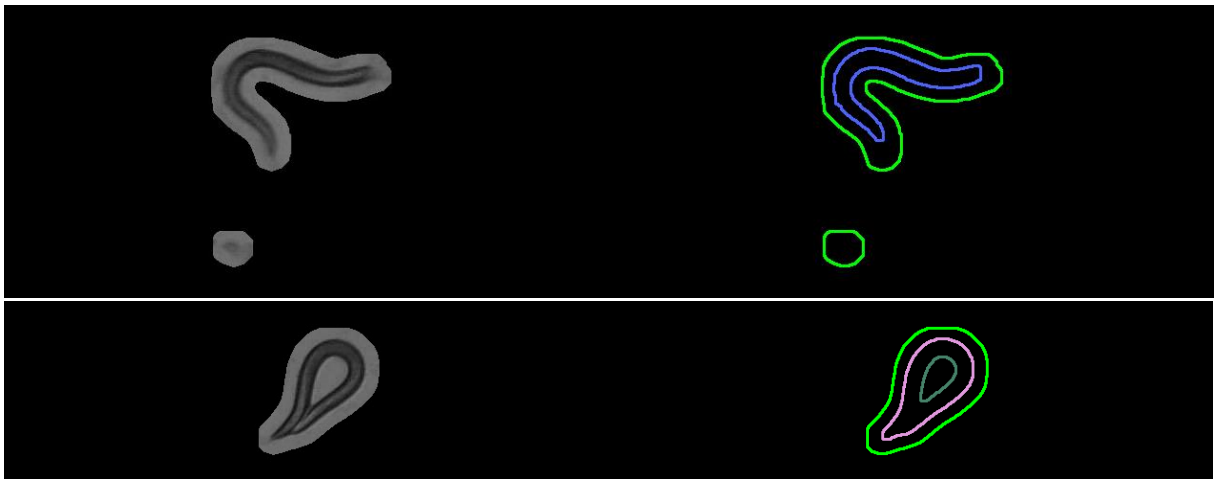

Figura 16. Obtención de contornos del C. elegans

Surgió un problema a la hora de segmentar un *C. elegans* que se encuentra cerrado formando un círculo, ya que no se estaba retirando el hueco central. Es por esto por lo que se tuvo que generar otra máscara del hueco interior resultante (Máscara2) y restarla a la anterior para obtener siempre un buen resultado (Figura 17). Este problema es el que hizo que se tuviera que eliminar inicialmente el *C. elegans* para regenerar la máscara a partir del hueco que dejó en la imagen, y no al revés.

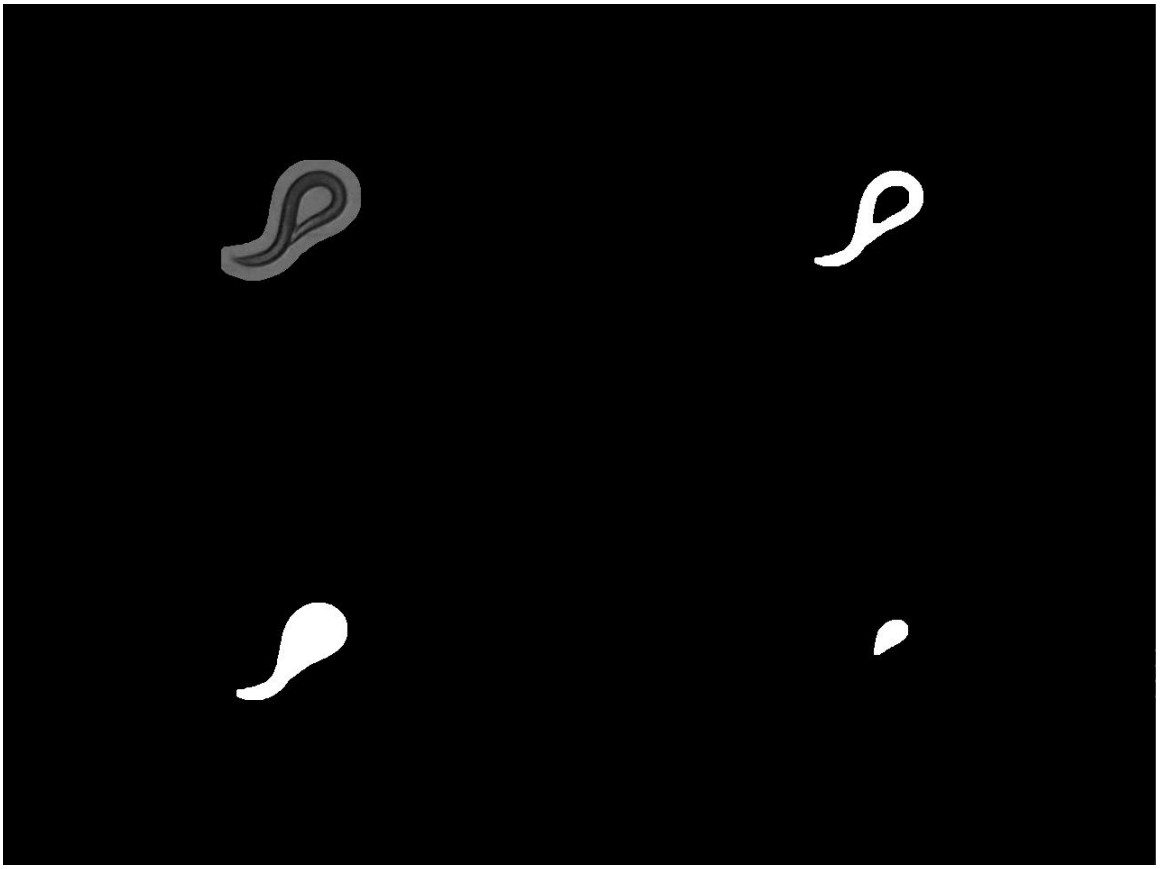

Figura 17. Máscaras según jerarquía de contornos. (Original – Máscara final - Máscara1 - Máscara2).

Finalmente, aplicando unas operaciones consecutivas de dilatación y erosión se consigue limpiar por completo los últimos restos de background que pudieran haber quedado en la imagen, y se obtiene finalmente el resultado final que se utilizará para crear el dataset.

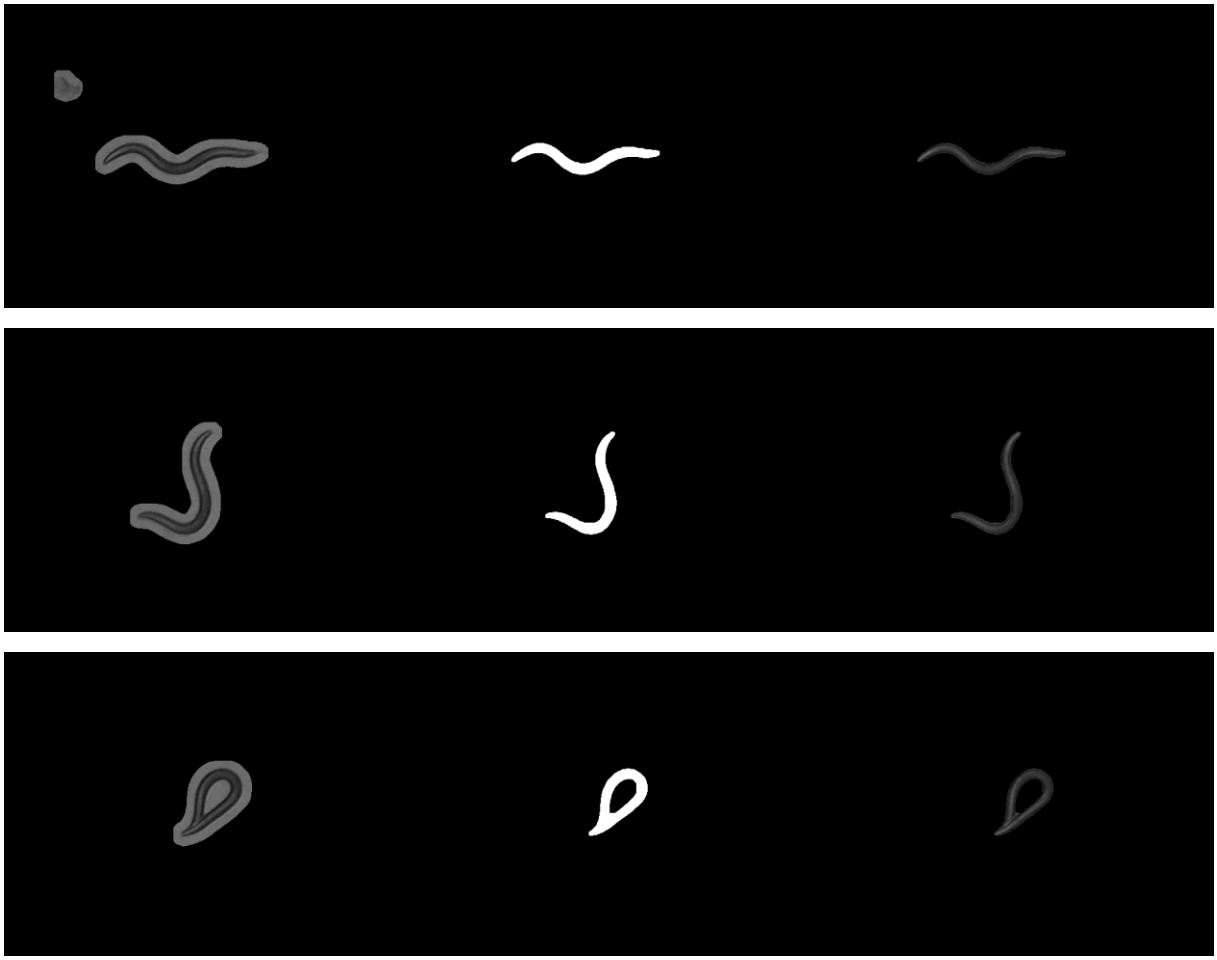

Figura 18. Resultado final de la segmentación (Original - Máscara - Máscara sobre Original).

Aquí quizás se pueda pensar que si se ha conseguido segmentar con técnicas de visión artificial clásicas el *C. elegans*, el uso de una red neuronal para hacer la misma tarea podría considerarse redundante. Esto se debe a que el hecho de que estos van a ser los datos de entrenamiento del modelo *U-Net*, hasta este punto es hasta donde va a llegar el rendimiento de la red neuronal como máximo.

Sin embargo, el objetivo principal de este proyecto no es solo la segmentación efectiva, sino también la adquisición de un conocimiento profundo sobre la tecnología de redes neuronales, con vistas a su aplicación en futuros proyectos.

Aunque algo que sí que podemos hacer es hacer la red neuronal más robusta que el script anteriormente explicado. El *script* original funciona bien ya que las condiciones en las que se han tomado los videos inicialmente han sido en un entorno con condiciones de iluminación muy controladas, y el problema se ha podido solucionar utilizando un *threshold* fijo.

Pero aplicando modificaciones aleatorias sobre las imágenes como desenfoques, cambios de intensidad o de la iluminación antes de alimentárselas a la red neuronal durante la fase de entrenamiento, se podrá lograr generar un modelo mucho más adaptable a diversas condiciones y que pueda lograr resultados que no se pudieran conseguir de otra manera. Estas modificaciones aleatorias, junto a otras más se explorarán en detalle más adelante en el apartado de "*Data Augmentation".*

3.3. Almacenamiento de imágenes y máscaras

Una vez clara la segmentación de los *C. elegans*, se puede comenzar a generar el dataset. Para esto, se van a extraer fotogramas cada un cierto intervalo de los videos, ya que al coger frames consecutivos donde la posición del *C. elegans* no cambia demasiado puede hacer que el modelo no generalice bien y obtenga un sobreajuste (o "*overfitting*") con facilidad. Por esto se ha escogido un intervalo de 50 frames, entre los cuales la posición del *C. elegans* ha cambiado lo suficiente como para que no haya problemas más adelante.

Cada fotograma se va a segmentar de la misma manera que se ha explicado anteriormente, y los pares de imágenes antes y después de procesar se van a almacenar en dos carpetas diferentes, una llamada "*Images*" con las imágenes originales, y otra llamada "*Masks*" con las máscaras, o "*Ground Truth*" extraídas. Ambas tendrán el mismo nombre para poder utilizarlas como un par inseparable en el futuro.

Finalmente se extrajeron un total de 50.000 pares de imágenes-máscaras, de los cuales 2.000 se separaron aleatoriamente para utilizar en la fase final de prueba y comprobar la efectividad del modelo entrenado, y el resto serán usadas para la fase de entrenamiento y validación.

Se ha tenido que realizar una revisión y limpieza de los datos, ya que por alguna razón algunas de las máscaras (unas 200 de los 50.000 totales) estaban completamente en negro, y generaban errores *NaN* durante el entrenamiento al calcular los criterios de pérdida. Pero se ha podido solucionar fácilmente con un script de *Python* que ha automatizado el trabajo.

3.4. DataLoader

En el siguiente paso, se carga el dataset a *Python* y se genera un *DataLoader*, con la ayuda de dos clases de *Pytorch*. Se genera una función que hereda de la clase "*Dataset*" para cargar los pares de imágenes y máscaras en *Python*, a la vez que aplica las transformaciones pertinentes a los datos, previamente al entrenamiento.

Este dataset se divide aleatoriamente en un dataset de entrenamiento y otro de validación, con un ratio del 70% y el 30%, respectivamente.

Por otro lado, se utiliza la clase "*DataLoader*" para envolver al dataset con un iterable, que sirve para alimentar los datos en la fase de entrenamiento en lotes de un tamaño establecido *"Batch Size"*, y permite añadir funcionalidad extra como barajar aleatoriamente los datos.

Finalmente, se han realizado comprobaciones para comprobar que los pares imagen-máscara se han cargado de forma correcta (Figura 19). Por un lado, en el tamaño de los tensores, para un *Batch Size* de 12 se obtienen respectivamente lo siguiente:

```
# Comprobamos tamaños del par imagen-mascara
imgs, masks = next(iter(train_loader))
print(imgs.shape, masks.shape)
```

```
torch.Size([12, 3, 224, 224]) torch.Size([12, 1, 224, 224])
```

Figura 19. Comprobación del tamaño de los tensores de Imagen y Máscara, respectivamente.

Por otro lado, se visualiza un lote de 12 pares de imágenes y se superpone la máscara correspondiente con una transparencia del 50% encima, y se observa cómo no hay ninguna discrepancia y coinciden a la perfección.

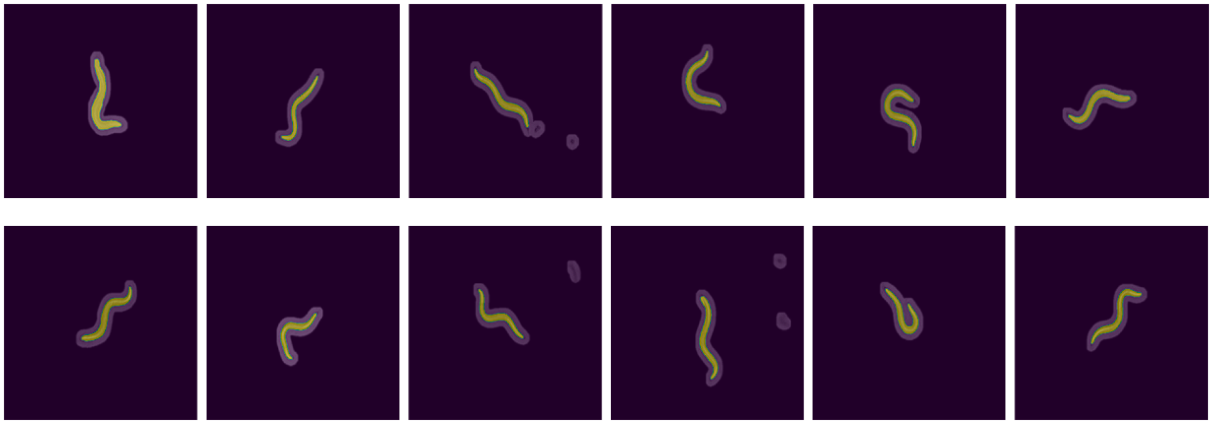

Figura 20. Comprobación del DataLoader. Lote de par imágenes-máscara.

3.5. Transformaciones y Data Augmentation

Antes de proceder con el entrenamiento del modelo, es crucial aplicar una serie de transformaciones a las imágenes. Estas transformaciones incluyen la transformación de imagen a tensor, escalado de las imágenes para variar las dimensiones espaciales, inversiones verticales y horizontales, rotaciones, recortes y cambios de iluminación. Estas etapas son fundamentales para preparar los datos para el proceso de aprendizaje y asegurar que este pueda obtener buenos resultados.

La única transformación estrictamente necesarias es transformar la imagen a tensor, para que este pueda ser procesado con una tarjeta gráfica, cosa que no se podría hacer con un "*ndarray"* de *Numpy* convencional.

Por otro lado, se emplea la técnica esencial del Aumento de Datos (o "*Data Augmentation*"). Esta técnica de preprocesamiento de datos se centra en aumentar la cantidad y diversidad del conjunto de datos y es especialmente útil cuando se dispone de pocos datos para el entrenamiento. Pese a que en este caso no es necesario del todo ya que se tienen imágenes de sobra (alrededor de 50.000), es una buena práctica ya que ayuda a que el modelo sea más robusto y mejore su capacidad de generalizar a partir de los datos de entrenamiento, para predecir mejor datos nunca vistos. Esto se consigue por aumentar la diversidad de datos al añadir estas variaciones aleatorias en las imágenes y por actuar como una forma de regularización implícita, ya que estas transformaciones se aplican de forma aleatoria.

De esta manera se evita también lo que se conoce como "*overfitting*" y que pase lo explicado antes, que el modelo falle en generalizar y aprenda demasiado bien los datos de entrenamiento, pero sea incapaz de predecir el resto.

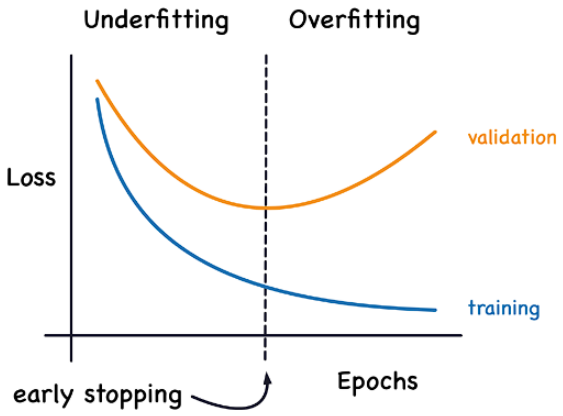

Figura 21. Ejemplo gráfico de Overfitting durante la fase de entrenamiento. (https://www.kaggle.com/code/ryanholbrook/overfitting-and-underfitting)

En este proyecto se están aplicando aleatoriamente inversiones verticales y horizontales, recortado (*crop*), zoom y cambios en el brillo de forma aleatoria en las imágenes, como se puede observar en las siguientes imágenes comparativas.

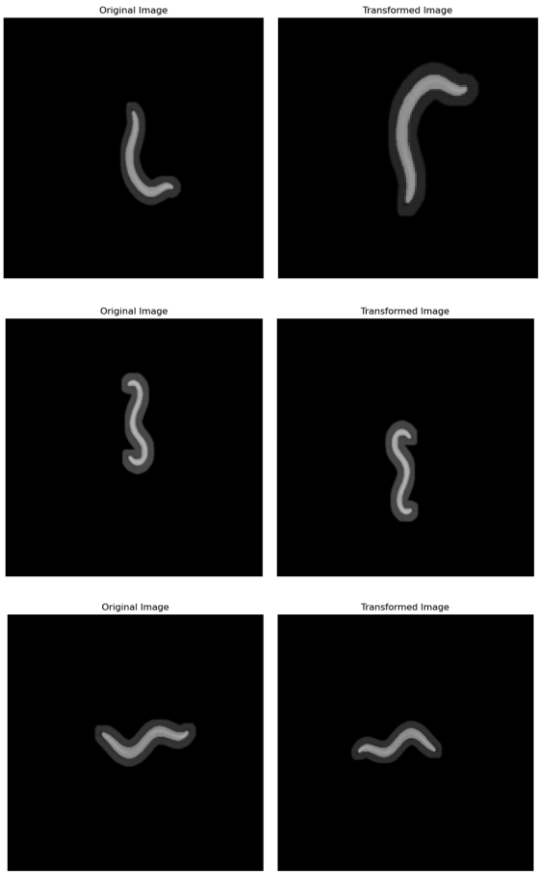

Figura 22. Pares Imágenes-Máscara antes y después de aplicar las transformaciones para Data Augmentation.

4. MODELADO DE LA RED NEURONAL

Se procede al modelado de una red neuronal convolucional (CNN) para la segmentación semántica de imágenes. Para ello, se entrenarán y compararán diferentes modelos de la arquitectura U-NET.

La segmentación semántica es una tarea de visión artificial que consiste en etiquetar cada uno de los píxeles con la probabilidad de que pertenezca a un determinado objeto o clase. No solo delimita la zona alrededor del objeto como un detector o clasificador de objetos, sino que proporciona una comprensión más detallada a cada píxel individual. Es especialmente de utilidad en el ámbito del proyecto, que está relacionado con el análisis de imágenes biomédicas.

En este caso, se va a realizar en concreto una segmentación semántica binara, ya que solo hay dos clases a las que se pueden asignar los píxeles de la imagen, el objeto de interés y el fondo. Por lo tanto, la salida de la red neuronal será una imagen donde cada píxel adquiere un valor de entre 0 y 1, siendo esto la probabilidad de que el píxel pertenezca a un *C. elegans*.

4.1. U-NET clásica

El primer modelo que se va a desarrollar es una red neuronal basada en la arquitectura U-Net presentada en 2015 para la segmentación de imágenes biomédicas [27].

Tiene una arquitectura simétrica que consta de dos componentes principales, una primera fase de contracción o "*downsampling*", y una fase de expansión o "*upsampling*", que juntas forman la característica forma de "U" por la que recibe el nombre U-net esta arquitectura (Figura 23).

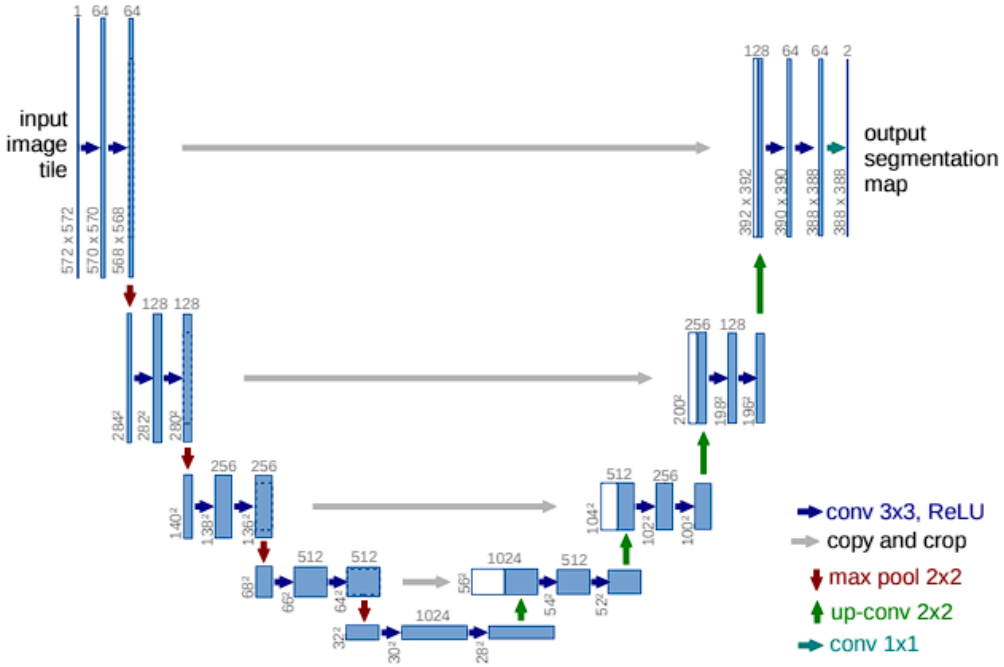

Figura 23. Estructura de la arquitectura U-Net clásica [27].

La fase de *downsampling*, también conocida como codificador, emplea una serie de capas convolucionales y de *max pooling* para extraer las características de alto nivel de la imagen. En concreto, cada capa convolucional está seguida de una función de activación no lineal *ReLu* y una capa de *max pooling* para reducir la dimensionalidad de las imágenes.

La fase de *upsampling*, o decodificador utiliza una combinación de operaciones *up-convolution* (o *transposed convolution*) y convoluciones genéricas que sirven para determinar la información espacial de las imágenes.

Lo que hace tan fuerte y diferencia a esta arquitectura son las conexiones *Skip* entre las capas iguales de codificador y decodificador, las cuales permiten que la red utilice la información contextual de alto nivel de la parte del codificador, y las combina con la información espacial de bajo nivel de la parte del decodificador para mejorar considerablemente la precisión de la segmentación

Finalmente, en la capa de salida se emplea una convolución de 1x1 seguida de una función de activación, como la *sigmoide* o la *softmax*, para asignar así las probabilidades de pertenencia de cada clase a cada píxel de la imagen. Dependiendo de los filtros utilizados para esta última capa, podemos determinar el número de clases que se quieren segmentar.

Esta arquitectura y sus derivadas son actualmente de las más potentes conocidas para realizar todo tipo de tareas de segmentación semántica en diferentes campos de aplicación.

Implementación Propia de la U-Net

Para la realización de este proyecto, se ha programado mediante *PyTorch* una implementación propia de un modelo con arquitectura U-Net, con algunos cambios mínimos respecto a la original.

Tanto para la fase de *downsampling* y *upsampling*, se van a utilizar un total de 4 niveles de convolución, más el nivel más bajo intermedio. Las convoluciones serán de tamaño 3x3, *stride* 1 y *padding* 1, manteniendo así el tamaño de la imagen antes y después de la convolución, y reduciendo la complejidad de entendimiento del modelo.

Tras cada convolución se va a utilizar una capa de *Batch Normalization* (explicado a continuación) y finalmente una capa de activación tipo *ReLU*. Después se incluye un *Max Pooling* de tamaño de *kernel* 2x2. Con esto se completa cada uno de los niveles del *encoder*.

Para el *decoder*, en vez de realizar una *up-convolution* como indica el *paper* original, se ha decido realizar un *up-sample* seguido de una convolución 1x1, de manera que en vez de agrandar la imagen y rellenarla con píxeles con valor cero, se están rellenando esos píxeles con una interpolación bicúbica, obteniendo un mejor resultado y evitando posibles fenómenos como los “Checkerboard Artifacts” en la imagen de salida [28].

Se programan *las skip connections* de tal manera que se concatenen las imágenes del codificador y del decodificador como indica en el paper.

Finalmente, para la última convolución 1x1 de salida, se ha utilizado una única dimensión y con una capa de activación sigmoide, por lo que el resultado va a ser un tensor donde cada valor representa la probabilidad de que ese píxel pertenezca a la clase *C. elegans.* Se puede observar el esquema de como se ha implementado a continuación (Figura 24).

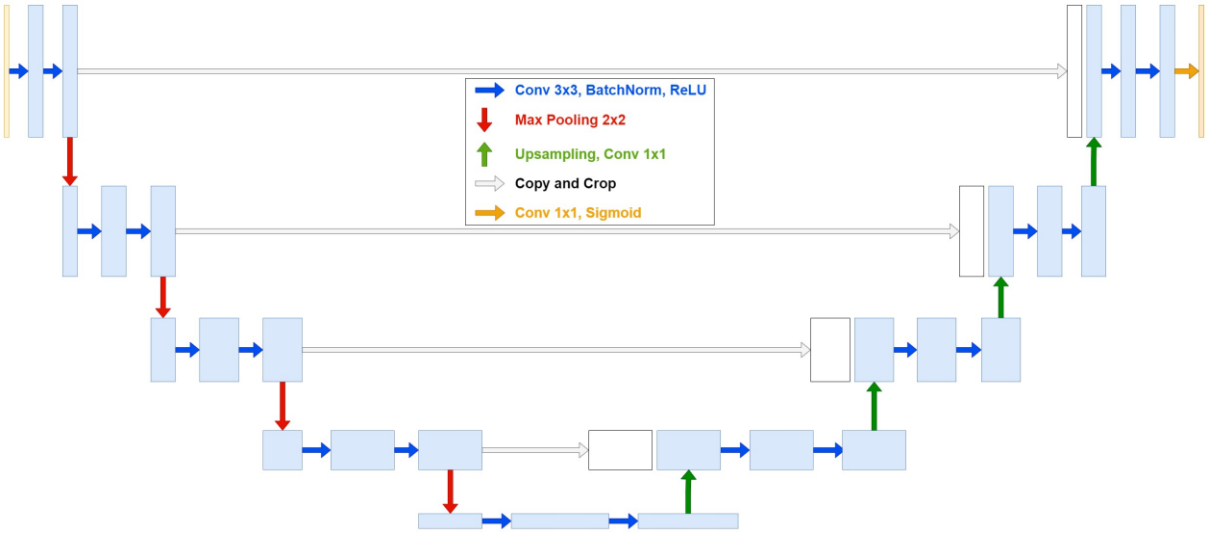

Figura 24. Implementación propia de U-Net.

Aunque no se utiliza en la U-Net original, ya que el concepto fue introducido después de su descubrimiento, se ha añadido una capa de *Batch normalization* [29] detrás de cada convolución. Esta técnica aporta varios beneficios que mejoran tanto la eficiencia como la efectividad del modelo. En cada mini lote, se normalizan las activaciones de las neuronas en cada capa. Se resta la media y se divide por la desviación estándar del mini-lote, y se aplican parámetros aprendibles gamma y beta que permiten que la red deshaga la normalización si lo desea (Ecuación 1).

$$BN(x) = \gamma\left(\frac{x-\mu}{\sqrt{\sigma^2+\epsilon}}\right)+\beta$$

Ecuación 1. Fórmula de Batch Normalization.

Donde μ es la media del mini-lote, σ^2 es la varianza del mini-lote, γ es una escala aprendible, β es un *bias* aprendible, y ϵ es un pequeño número para evitar la división por cero.

Con esto se consigue acelerar la convergencia, reduciendo el "*covariate shift*", referido al cambio en la distribución de las activaciones internas de una red neuronal. Al normalizar, se reduce la cantidad en la que se tienen que actualizar las activaciones de las capas posteriores del modelo debido a un cambio en capas anteriores, lo que también permite que se utilicen tasas de aprendizaje más altas y actúa como regularizador implícito, ya que añade cierto nivel de ruido durante el entrenamiento, lo que por lo tanto mejora a la vez la capacidad de generalización del modelo.

4.2. U-NET++

Con el tiempo, han surgido varias modificaciones sobre la arquitectura U-Net original para mejorar su rendimiento y adaptabilidad. Una de las más notables que han surgido recientemente es la U-Net++. La U-Net++ [30] busca abordar algunas de las limitaciones de la U-Net original, como la captura de detalles a diferentes escalas y la optimización del rendimiento de la segmentación de los objetos. Las características principales que incluye este modelo son las conexiones anidadas, mediante las cuales se introducen nuevas capas de convolución anidadas y densas entre las dos ramas principales de codificador y decodificador (Figura 25), lo que permite una mejor propagación del gradiente y captura de características a múltiples resoluciones diferentes.

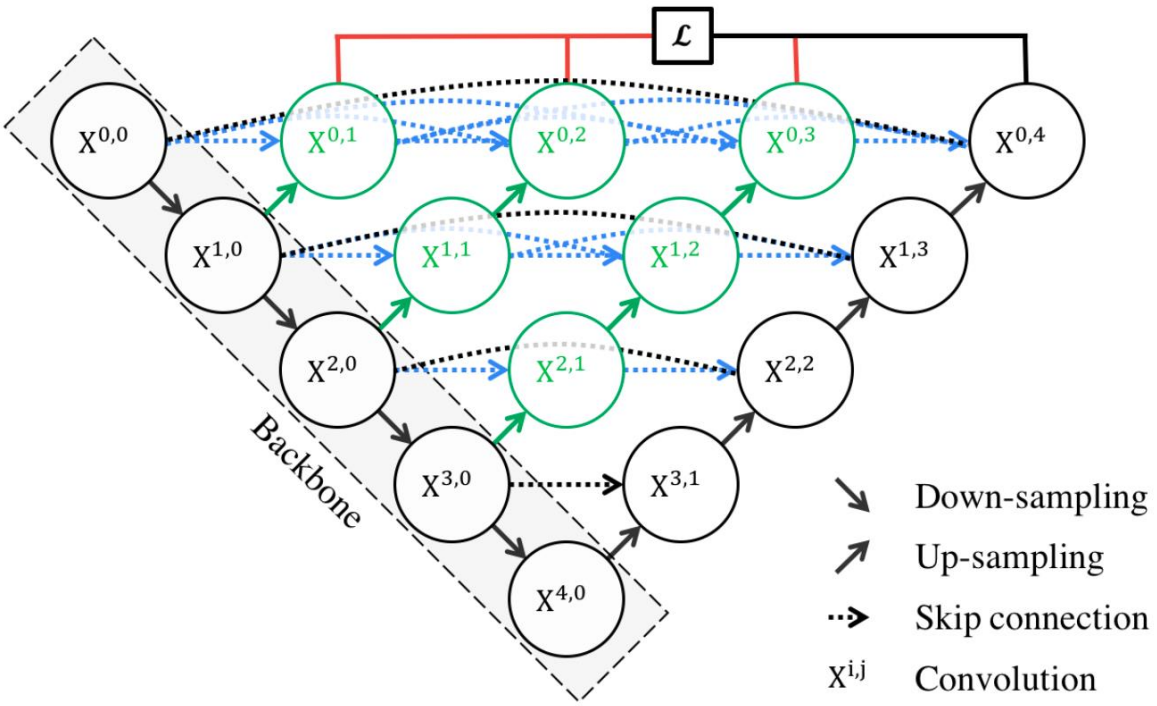

Figura 25. Estructura de la U-Net++ [30]

Con esto se consigue una ligera mejora en cuanto a la precisión en la segmentación para determinados casos de estudio, al capturar los detalles a diferentes niveles de resolución, así como un entrenamiento más rápido por la mejora en la eficiencia de propagación de los gradientes.

Esta variante se utilizará para comparar el rendimiento del entrenamiento con diferentes modelos de U-Net, y ver como de útiles pueden ser la utilización de estas nuevas variantes en el caso de estudio del proyecto.

5. ENTRENAMIENTO Y VALIDACIÓN DEL MODELO

Entre las fases que destacan en el proceso de desarrollo de un modelo aprendizaje profundo, tenemos las fases de entrenamiento y de validación. Estas son cruciales para el desarrollo de un modelo robusto y con capacidad de generalizar bien a datos nunca vistos.

Para llevar a cabo estas fases y obtener un resultado satisfactorio, será necesaria la selección y ajuste de los hiperparámetros del modelo, que son aquellos que se definen por el usuario con anterioridad, es decir, que no los aprende automáticamente el modelo.

A continuación, se definirán las métricas de evaluación, para medir de manera objetiva el rendimiento del modelo ante varias combinaciones de modelo e hiperparámetros.

Después, se expondrán las funciones de pérdida elegidas, las cuales el modelo busca minimizar para mejorar sus predicciones.

Finalmente, se explicarán los algoritmos de entrenamiento y validación creados.

5.1. Ajuste de Hiperparámetros

La elección y ajuste de hiperparámetros tiene un papel crucial a la hora de obtener un buen rendimiento en el modelo [33]. Estos son variables que definen la estructura del modelo y la forma en que se entrena. Como se mencionó anteriormente, estos parámetros se configuran previamente al aprendizaje.

A continuación, se van a describir los hiperparámetros escogidos y cómo se han ajustado para optimizar el rendimiento.

Tasa de Aprendizaje (Learning Rate)

Este es uno de los hiperparámetros más importante a la hora de entrenar un modelo. Controla cuánto se actualizan los pesos del modelo en cada iteración del algoritmo de optimización.

Una tasa de aprendizaje demasiado alta puede causar que el modelo nunca llegue a converger, además que puede llevar al *overfitting* con facilidad, especialmente si no se utiliza ninguna técnica de regularización. Y también puede llegar a hacer el entrenamiento inestable, por lo que los pesos y la función de pérdida del modelo fluctuarían constantemente.

Por otro lado, una tasa de aprendizaje demasiado baja puede ralentizar el entrenamiento considerablemente hasta llegar al mínimo en la función de pérdida, requiriendo muchas más iteraciones. También puede provocar infra ajuste (o "*underfitting")*, donde el modelo no aprende lo suficiente de los datos de entrenamiento, lo que no le permite generalizar correctamente. El entrenamiento del modelo será mucho más estable y disminuirá la función de pérdida de manera constante, pero es posible que se quede bloqueado en un mínimo local y no encuentre el mínimo global.

Por ello, es crucial seleccionar una tasa de aprendizaje óptima, ni muy alta ni muy baja. Para esto existen diversas técnicas que ayudan a encontrar estas tasas de aprendizaje óptimas antes de empezar con la propia fase de entrenamiento.

Búsqueda de la tasa de aprendizaje óptima

En este proyecto se ha utilizado el concepto de *"Learning Rate Range Test"* [31]. Este consiste en partir de una tasa de aprendizaje muy baja, e ir incrementándola exponencialmente en cada iteración, hasta recorrer todo el dataset una vez, es decir, completar un *Epoch*. Se registra la función de pérdida en cada iteración y se grafica en función de la tasa de aprendizaje (Figura 26).

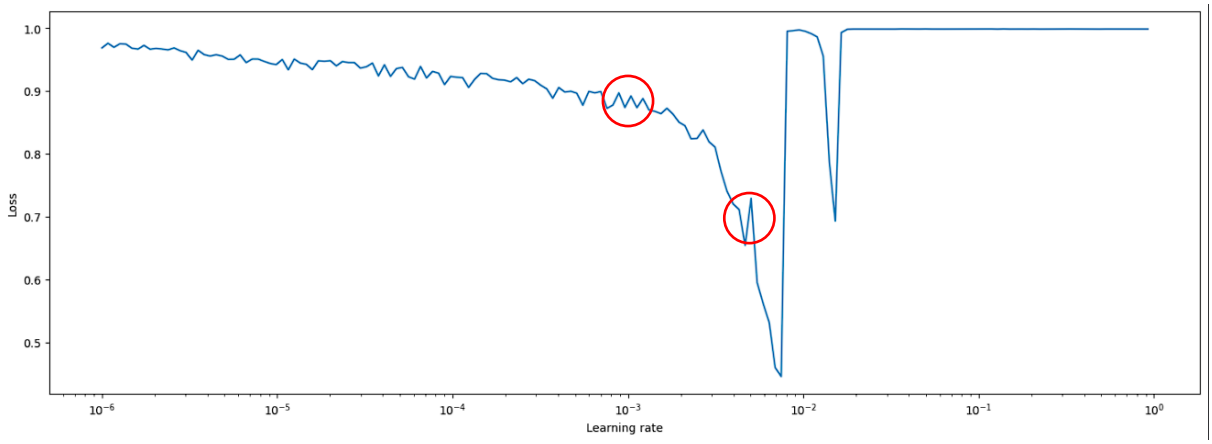

Figura 26. Tasa de aprendizaje vs Pérdida para encontrar el LR óptimo.

Esta técnica explica que la tasa de aprendizaje óptima máxima se encuentra poco antes del punto de mínima pérdida, y la óptima mínima se encuentra en el punto donde la pérdida comienza a disminuir rápidamente, o bien se utiliza una regla aproximada de entre 1/3 y 1/10 de la tasa máxima. En el caso de este modelo, se ha escogido como máximo 0.004 y como mínimo 0.001.

Ahora se podría utilizar una tasa fija durante toda la fase de entrenamiento, o bien implementar un *Scheduler* que la haga variar a lo largo de este, para acelerar la convergencia del modelo y mejorar el rendimiento.

"One-Cycle Policy". Método de Super-Convergencia

Es por esto que se va a utilizar otro método introducido por el mismo autor, llamado "*One-Cycle Policy*", o "*1cycle"* [32] en su paper sobre la Super-Convergencia, la cual se basa en utilizar tasas de aprendizaje muy altas junto a "1cycle" para entrenar los modelos de manera muy rápida.

El método "1cycle" es una variante del concepto de las tasas de aprendizaje cíclicas "CLR", cuya idea consiste en variar la tasa de aprendizaje entre un máximo y un mínimo en lugar de mantenerla constante, lo que permite salir de mínimos locales y converger a mínimos globales óptimos, acelerando la convergencia.

En “1cycle”, lo que cambia es que la variación de la tasa de aprendizaje desde el mínimo hasta el máximo, y luego de nuevo al mínimo se realiza una única vez durante todo el entrenamiento y todos los Epochs (Figura 27). Este método ha resultado ser muy efectivo para entrenar modelos rápidamente y con un buen rendimiento

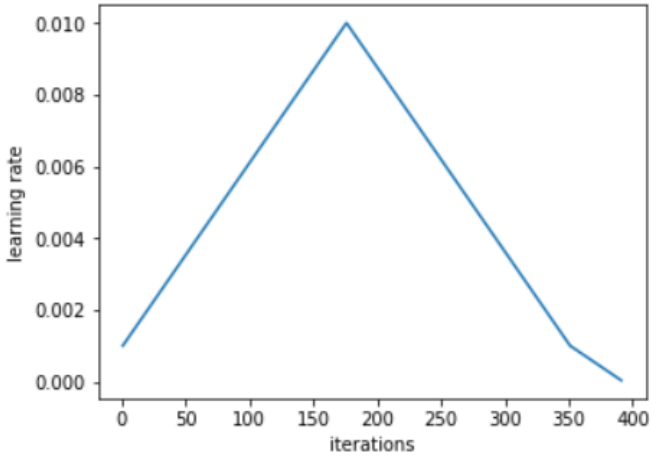

Figura 27. Evolución de la tasa de aprendizaje durante el entrenamiento completo según el método "1cycle" (https://sgugger.github.io/the-1cycle-policy.html)

En resumen, un ajuste cuidadoso de los hiperparámetros, respaldado por técnicas validadas, es fundamental para el desarrollo de modelos de aprendizaje automático robustos y eficientes.

Tamaño del Lote (Batch Size) y Tamaño de las Imágenes

El tamaño del lote y tamaño de las imágenes son otros hiperparámetros importantes a la hora de entrenar los modelos, los cuales tienen un gran impacto en cuanto a la eficiencia computacional y al rendimiento del modelo. Estos están relacionados y principalmente limitados con la capacidad de memoria de la GPU.

El tamaño de lote define el número de imágenes que se utilizarán en cada iteración para actualizar los pesos de este. Es recomendable utilizar el tamaño de lote más grande posible [33], ya que se reduce el número de iteraciones a realizar durante el entrenamiento, y por lo tanto la posibilidad de que haya *overfitting*. Pero al tener lotes más grandes se tendrá que aumentar el número de *Epochs* a entrenar de forma acorde para obtener un desempeño similar a un tamaño de lote menor. Se ha escogido finalmente un *Batch Size* de 12.

El tamaño de la imagen tiene que ver con la resolución que se toma en esta antes de introducirla al modelo. En tareas de segmentación es un parámetro crítico, ya que cuanto mayor la resolución de la imagen, mejores detalles se obtendrán en el entrenamiento. Se ha utilizado un tamaño de 224x224 píxeles, con el cual se consigue un resultado bastante bueno para el caso de estudio.

El problema es que ambos parámetros vienen limitados por la capacidad de memoria de la GPU. Tanto aumentar el tamaño de la imagen, como aumentar el tamaño del lote, aumenta las necesidades de memoria durante el entrenamiento.

Es por ello por lo que el ajuste de estos tamaños es un acto de equilibrio entre la eficiencia computacional y el rendimiento del modelo que se debe ajustar meticulosamente en función de las limitaciones del hardware disponible y de la tarea a realizar.

Número de Epochs

El número de Epochs es otro parámetro indispensable a la hora de entrenar un modelo. Cada Epoch indica una pasada completa a través de todo el dataset de datos de entrenamiento. Es importante utilizar un número de Epochs adecuado, ya que un número insuficiente puede resultar en *underfitting*, mientras que un número excesivo puede resultar en overfitting.

Es por eso por lo que se utilizan las funciones de pérdida para monitorear el modelo durante el entrenamiento, y obtener conclusiones sobre el número de Epochs adecuado a utilizar.

Siempre se puede poner un número elevado de Epochs, para asegurarse de que el modelo no se quede corto, y utilizar técnicas como el "Early Stopping", que detiene el entrenamiento automáticamente si detecta que el modelo no ha mejorado nada durante un tiempo o si ha comenzado a empeorar.

Optimizador

El optimizador es un algoritmo que ajusta los pesos de un modelo para minimizar la función de pérdida. También es el encargado de modificar la tasa de aprendizaje durante este tiempo si se configura un *scheduler*, como el de "*1cycle*" explicado anteriormente. El objetivo es buscar el mínimo global de la función de pérdida, para que el modelo funcione de la mejor manera posible.

El optimizador utiliza la técnica de "*backpropagation*" que, durante el entrenamiento, calcula los gradientes de la función de pérdida. Los gradientes son vectores de derivadas parciales que indican hacia donde crece más rápidamente la función de pérdida. Tras cada iteración el optimizador actualiza los pesos para que se opongan a esta dirección de los gradientes, para minimizar así la función de pérdida. Antes de calcular los gradientes en cada iteración, se ponen a cero "*Zero Grad*" para evitar que se acumulen con los anteriores.

El optimizador utilizado en el entrenamiento del modelo es el de "Adam" (*Adaptive Moment Estimation*), uno de los más utilizados actualmente por su mejor velocidad de actualización y precisión.

5.2. Coeficientes de rendimiento

En las tareas de segmentación de imágenes es crucial evaluar el rendimiento de los modelos de una manera precisa y eficaz. Los coeficientes de rendimiento o métricas de evaluación sirven para obtener una calificación objetiva de la calidad del modelo. Esto facilita la medición de su desempeño y comparar diferentes modelos y enfoques.

Algunas de las métricas más comunes en la segmentación de imágenes, las cuales se van a utilizar son la Precisión (Accuracy), el Índice de Jaccard (IoU) y el coeficiente DICE (Dice Coefficient). Cada una de estas métricas tiene sus propias ventajas y desventajas, es por ello por lo que a menudo se utilizan de manera conjunta para obtener una evaluación más completa del rendimiento del modelo.

Precisión (Accuracy)

La precisión es la métrica más simple y fácil de entender, y se define como la proporción de píxeles correctamente clasificados respecto al total de píxeles de la imagen. Sin embargo, la precisión puede llegar a ser una métrica muy engañosa cuando las clases están muy desbalanceadas, por ejemplo, en casos donde el objeto de interés es muy pequeño en comparación con el fondo. Para que se entienda mejor, si el objeto de interés solo representa el 5% de los píxeles de la imagen, un modelo que proporcionaría una precisión del 95% o superior podría estar completamente equivocado al analizar el rendimiento de forma cualitativa.

$$\text{Precisión} = \frac{\text{Verdaderos Positivos} + \text{Verdaderos Negativos}}{\text{Total de Píxeles}}$$

Ecuación 2. Fórmula de la Precisión

Índice de Jaccard (IoU)

El Índice de Jaccard, también conocido como *IoU* (*Intersection over Union*), es una métrica más robusta que mide la superposición entre la predicción del modelo y la segmentación verdadera. Se calcula como la intersección de los dos conjuntos dividida por su unión. Es especialmente útil para evaluar la calidad de la segmentación en cuanto a tamaño y forma del objeto

$$\text{IOU} = \frac{\text{Intersección (Predicción, Verdadero)}}{\text{Unión (Predicción, Verdadero)}}$$

Ecuación 3. Fórmula del Índice de Jaccard (IoU).

Coeficiente DICE (Dice Coefficient)

El coeficiente DICE es una métrica similar al IoU, pero le da más importancia a la intersección de los dos conjuntos. Esto lo hace poderoso para aplicaciones donde pequeñas discrepancias son importantes, pero a la vez sensible a las pequeñas diferencias entre la predicción y la segmentación esperada. La fórmula del coeficiente DICE es igual a dos veces la intersección entre las imágenes dividido por la unión entre ellas.

$$DICE(A, B) = \frac{2 \times |\, A \cap B \,| + 1}{|\, A \,| + |\, B \,| + 1}$$

Ecuación 4. Fórmula coeficiente DICE

Se utiliza un parámetro "*smooth*", con un valor bajo arbitrario, que en este caso es 1 para evitar posibles divisiones entre cero.

Este coeficiente va a ser uno de los más importantes a la hora de cuantificar el rendimiento que proporciona el modelo, junto al coeficiente de *Jaccard*.

5.3. Funciones de pérdida

Para que el modelo sea capaz de aprender en cada iteración y mejorar el resultado que proporciona, es necesario indicarle una función de pérdida o "*loss function*", para que actualice sus pesos para minimizar su valor. Esta función de pérdida cuantifica como de bien está realizando un modelo su tarea, que en este caso es la segmentación semántica. En otras palabras, representa el error entre la salida predicha por el modelo, y la máscara o "*ground truth*" que era la salida real esperada. El objetivo es que se minimice esta función.

Hay multitud de funciones de pérdida, que funcionan mejor o peor dependiendo de la tarea a realizar por el modelo. Para la segmentación 2D, dos de las que funcionan mejor y que se van a utilizar son la pérdida según el coeficiente "*DICE*", y la pérdida de entropía cruzada (la modalidad binaria en este caso) "*BCE Loss*", o una combinación de estas.

DICE Loss

Como se desea minimizar la pérdida (y maximizar el coeficiente DICE), una manera sencilla de implementarla a partir de este es utilizar la siguiente fórmula, que deriva de la fórmula de DICE (Ecuación 4) calculada anteriormente.

$$\text{DICE Loss} = 1 - DICE$$

Ecuación 5. Fórmula del coeficiente de pérdida DICE.

BCE Loss

La función de perdida de entropía binaria se utiliza para medir el error entre el *Ground Truth* y las predicciones probabilísticas realizadas por el modelo.

La fórmula para el BCE Loss simplificada es la siguiente, la cual deberíamos aplicar píxel por píxel a las predicciones.

$$BCE\ Loss(y, \hat{y}) = -\frac{1}{N}\sum_{i=1}^{N}[y_i \cdot \log(\hat{y}_i) + (1 - y_i) \cdot \log(1 - \hat{y}_i)]$$

Ecuación 6. Fórmula Pérdida de Entropía Binaria Cruzada BCE Loss.

Donde:

- $y\ son\ las\ máscaras\ verdaderas, donde\ cada\ píxel\ toma\ un\ valor\ 0\ o\ 1$
- $\hat{y}\ son\ las\ predicciones\ probabilísticas\ de\ cada\ píxel\ del\ modelo, tras\ aplicar\ la\ función\ de\ activación\ sigmoide$
- $N\ es\ el\ número\ de\ imágenes\ en\ cada\ mini\ batch$
- $i\ es\ el\ índice\ de\ la\ imagen\ en\ ese\ mini\ batch$

Es importante tener en cuenta que este modelo espera que las probabilidades de las predicciones estén en el rango [0, 1], lo cual se ha logrado aplicando una función de activación sigmoide en la última capa de salida del modelo

DICE y BCE Loss combinados

Ya que ambas de las funciones de pérdida expuestas anteriormente son muy potentes para problemas de segmentación semántica, y cada una de ellas tiene sus puntos fuertes y débiles, una práctica común es crear una función de pérdida que combine ambas, para intentar conseguir un mejor resultado durante la fase de entrenamiento.

Se ha demostrado [34] que DICE Loss es más eficaz con la detección de bordes y es una función mucho más especializada, en cambio BCE Loss es mejor con la clasificación píxel por píxel y es más genérica y robusta. Se espera obtener un mejor resultado con DICE Loss que con BCE Loss por separado, pero el primero es mucho más susceptible a fallos por perturbaciones, lo cual se quiere evitar a toda costa, y más en segmentación de imágenes biomédicas donde un fallo puede salir caro. Es por esto por lo que combinar ambas puede ser la mejor opción para entrenar el modelo.

En este caso, se ha utilizado una combinación simplemente sumando ambas funciones de pérdida, según la siguiente fórmula:

$$Combined\ Loss = \text{BCE Loss} + \text{DICE Loss}$$

Ecuación 7. Ecuación de función de pérdidas combinada. BCE + DICE.

5.4. Funciones de entrenamiento y validación

La función de entrenamiento y validación es el corazón del proceso de aprendizaje de un modelo de red neuronal profunda. Una vez definidos todos los hiperparámetros expuestos anteriormente, estas funciones se encargan de varias tareas críticas. Primero, iteran a través de los conjuntos de datos de entrenamiento y validación. Seguido de esto, en cada iteración se calculan las predicciones del modelo basadas en las imágenes de entrada. Luego, se calculan las funciones de pérdida y se ajustan los pesos del modelo comparando la predicción realizada con la etiqueta real correspondiente, ajustando con el optimizador los pesos del modelo para minimizar estas pérdidas.

Finalmente, se obtienen métricas de evolución, como la precisión, el coeficiente DICE o el IoU, para analizar y comparar el rendimiento del modelo.

Durante la fase de entrenamiento, se guardan periódicamente los pesos calculados para poder reutilizarlos en un futuro sin tener que entrenar el modelo de nuevo, o bien para continuar con el entrenamiento si se interrumpe por algún motivo.

La fase de validación es especialmente crucial. En esta etapa no se realiza la *backpropagation*, por lo que el modelo no aprende durante esta fase. El modelo se expone a datos nuevos que no han sido utilizados para el entrenamiento, por lo que se puede medir si ha conseguido realmente generalizar más allá de los datos de entrenamiento y si se ha obtenido el rendimiento deseado. Para esto, se comparan las funciones de pérdida y las métricas de evaluación obtenidas en ambas fases.

6. ANÁLISIS DE RESULTADOS

A continuación, se van a presentar los resultados obtenidos tras la fase de entrenamiento, y se van a comparar varios hiperparámetros y modelos para escoger el que mejor rendimiento, tanto cuantitativo como cualitativo consiga antes de proseguir con las siguientes fases del proyecto.

6.1. Consideraciones previas al entrenamiento

En un primer momento, se entrenó la red neuronal con datasets formados a partir de fotogramas de diferentes vídeos, todos estos repartidos aleatoriamente (70% para entrenamiento y 30% para validación), por lo que los datasets estaban equilibrados con relación al número de muestras de cada vídeo. Pero esta aproximación, que es la que se utiliza normalmente a la hora de entrenar un modelo, resultó ser demasiado fácil para el modelo y alcanzaba el mínimo global sin problema, por lo que las mejoras que se implementaron, así como la regularización estaban siendo redundantes.

Es por esto por lo que se decidió ponerle el trabajo más complicado al modelo y utilizar vídeos diferentes para cada uno de los datasets, cuyos *C. elegans* estaban diferenciados claramente en tamaño y color (Figura 28).

Figura 28. Diferencia de clases de C.elegans. Entrenamiento - Validación – Test

La primera clase mostrada en la figura anterior es la más diferente de todas, y también la que ha demostrado ser la más complicada para el modelo. Es por esto por lo que se ha decidido usar como clase para el entrenamiento, ya que la U-Net es capaz de generalizar a las otras dos clases, pero no tan bien al revés.

Para el entrenamiento del modelo, finalmente no han sido necesarias demasiadas imágenes, en concreto solamente se ha utilizado un vídeo de cada clase, generando más tarde datasets del siguiente tamaño:

DATASET	*TAMAÑO*
Entrenamiento	*1270*
Validación	*500*
Test	*200*

Tabla 1. Tamaño de los datasets.

Se definen a continuación los hiperparámetros utilizados para realizar los ensayos:

HIPERPARÁMETRO	*VALOR*
Tasa de aprendizaje máxima	*0.004*
Tasa de aprendizaje mínima	*0.001*
Scheduler	*1cycle*
Optimizador	*Adam*
Nº Epochs	*5-20*
Función de Pérdida	*DICE, BCE, Ambas combinadas*
Tamaño de Lote	*12*
Input Size (píxeles)	*224x224*

Tabla 2. Definición de hiperparámetros utilizados.

Las métricas objetivas utilizadas para medir el rendimiento del entrenamiento y validación son las expuestas en el apartado 5.2., *DICE*, *Accuracy* y *Jaccard*. Durante este tiempo, también se prestará especial atención a la evolución de las funciones de pérdida.

Finalmente se evaluará de manera cualitativa comparando la predicción del modelo con la etiqueta verdadera.

6.2. Importancia de la Regularización

A lo largo del proyecto, se han introducido diferentes técnicas de regularización para el modelo, cuya finalidad es aumentar la generalización a datos desconocidos y disminuir el *overfitting*. Una de las técnicas que más afecta es la "*Data Augmentation".* Para comprobar el efecto de este en el modelo U-Net clásica, se ha entrenado dos veces, una de ellas utilizando "*Data Augmentation*" y otra de ellas sin utilizarla. Se ha entrenado durante 20 epochs para forzar el modelo a sobreentrenar y favorecer la aparición de *overfitting*, y la función de pérdida utilizada ha sido *DICE*.

Sin Data Augmentation

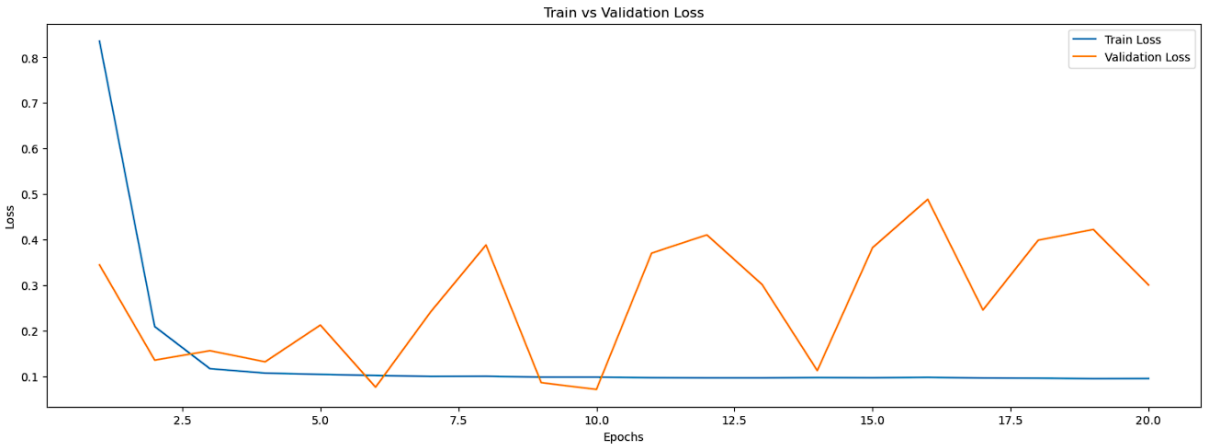

Figura 29. Train Loss vs Validation Loss sin Data Augmentation.

Se observa claramente como ocurre el fenómeno de *overfitting*, por lo que el modelo no puede generalizar para predecir imágenes que no ha visto durante el entrenamiento. Es por eso por lo que la pérdida de entrenamiento es capaz de converger a un valor muy bajo, pero la pérdida de validación es inestable ya que el modelo es incapaz de predecir correctamente estas imágenes.

Esto se observa más fácilmente al comparar la máscara predicha frente a la segmentación original. La predicción para los datos de entrenamiento es muy buena (Figura 30), pero para los datos de validación no (Figura 31).

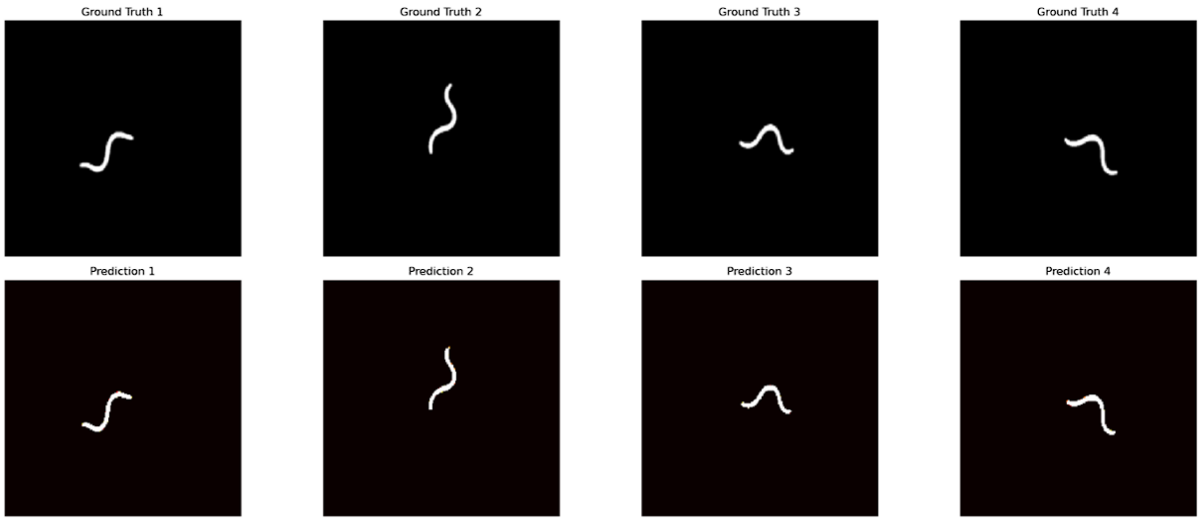

Figura 30. Comparación entre predicciones del modelo y Ground Truth sin Data Augmentation (Dataset de entrenamiento)

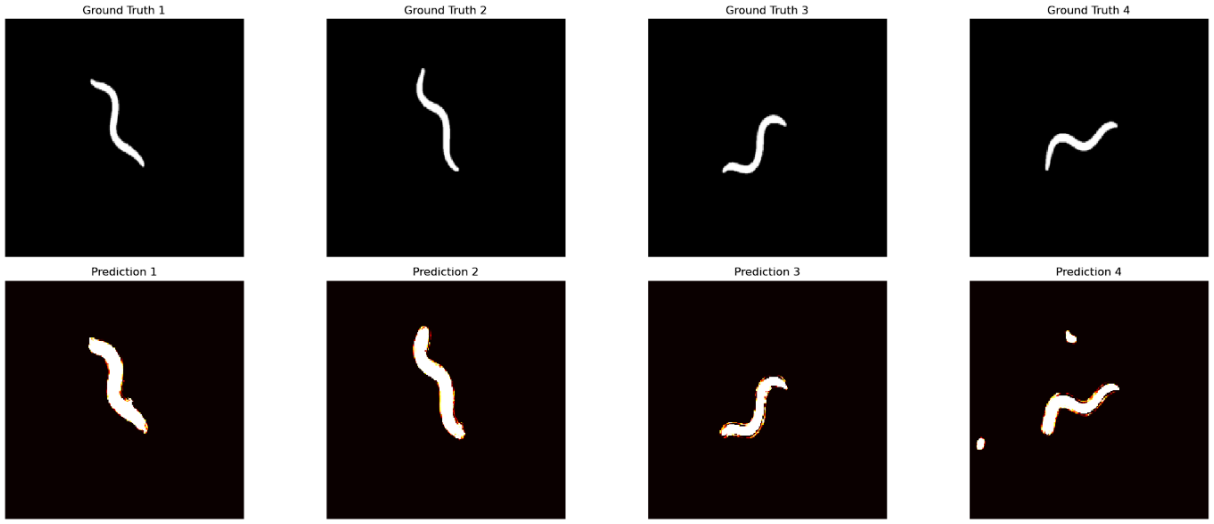

Figura 31. Comparación entre predicciones del modelo y Ground Truth sin Data Augmentation (Dataset de validación)

Con Data Augmentation

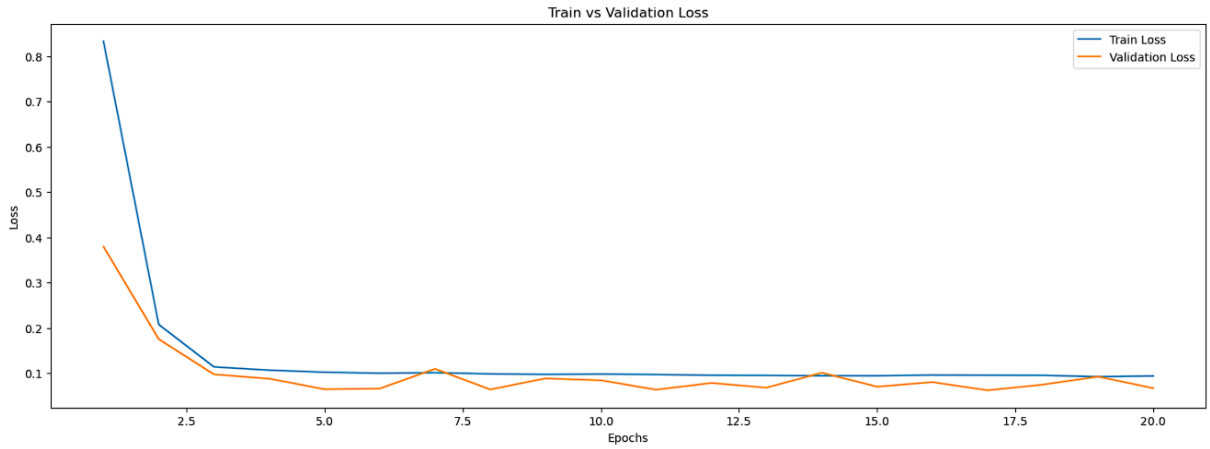

Figura 32. Train Loss vs Validation Loss con Data Augmentation.

Ahora sí que se observa el poder de la regularización. El modelo ha sido capaz de converger sin problema, y no se percibe ningún *overfitting*, ni siquiera al sobreentrenar el modelo durante más *Epochs* de los necesarios. El modelo ha generalizado correctamente a los datos no vistos.

En la gráfica de las pérdidas, también se puede ver algo que pueda resultar extraño al principio, y es que se consigue una pérdida menor para la validación que para el entrenamiento. Esto se explica fácilmente, ya que como se ha dicho antes se ha utilizado el tipo de *C. elegans* más diferente y complicado como dataset de entrenamiento, entonces como los datos de validación le resultan más fáciles al modelo y ha conseguido generalizar correctamente, es capaz de predecirlos con mayor precisión. Por otro lado, se han detectado un par de errores en las "*Ground Truth*" del dataset de entrenamiento, que pueden estar generando este levemente peor rendimiento.

Pero despreciando esta diferencia, es capaz de predecir ambos tipos con un gran nivel de detalle:

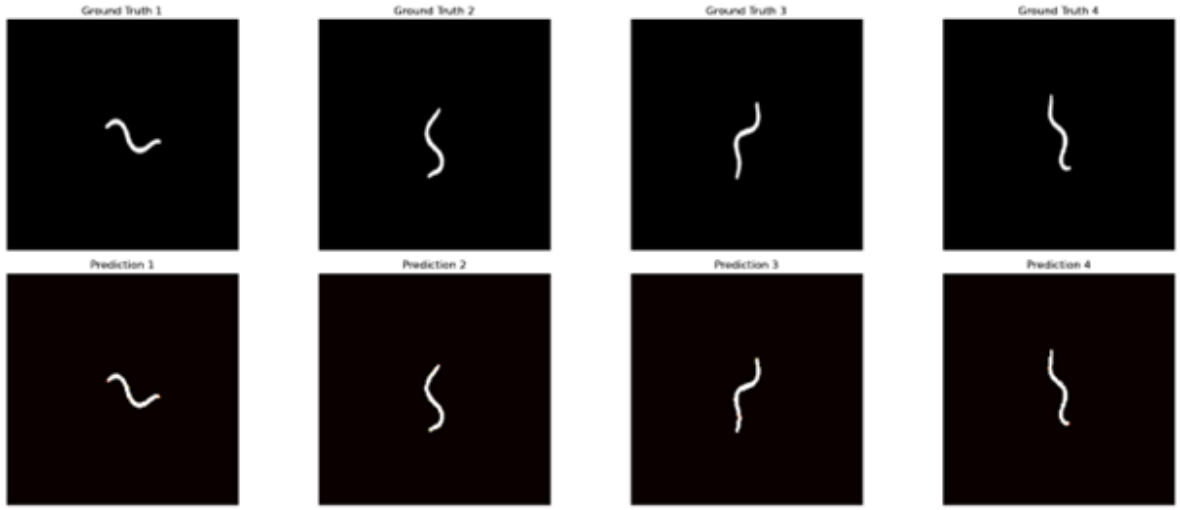

Figura 33. Comparación entre predicciones del modelo y Ground Truth con Data Augmentation (Dataset de entrenamiento)

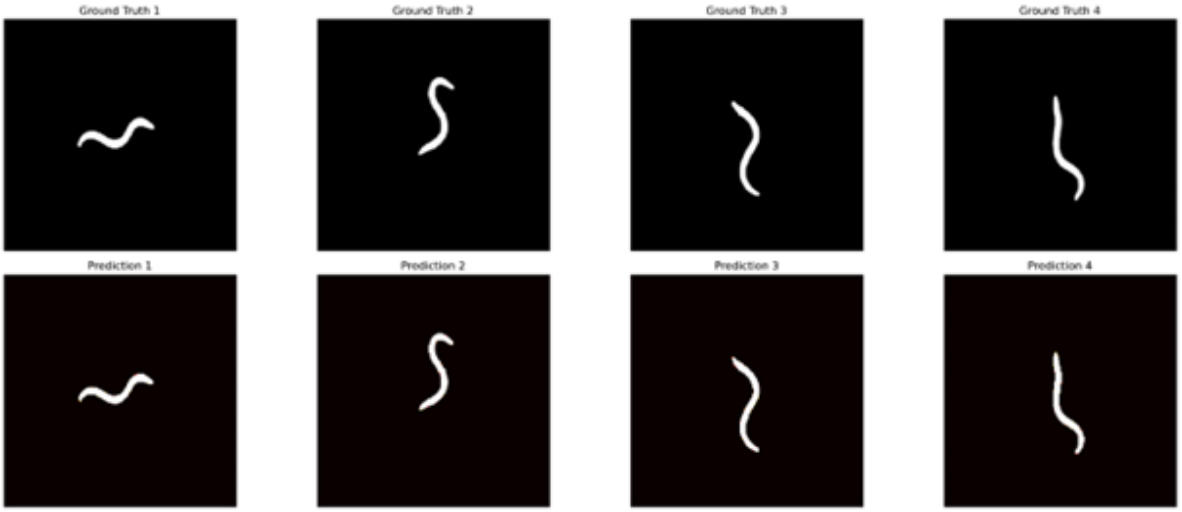

Figura 34. Comparación entre predicciones del modelo y Ground Truth con Data Augmentation (Dataset de validación)

Cuantitativamente, también se observa claramente la diferencia de rendimiento para la validación entre ambos modelos, y se aprecia como la métrica de precisión (Accuracy) puede llegar a conclusiones erróneas de ser usada por ella misma.

Modelo	***Loss***	***DICE***	***IoU***	***Accuracy***
U-Net sin Data Augmentation (Train)	*0.0953*	*0.9724*	*0.9464*	*0.9996*
U-Net sin Data Augmentation (Val)	*0.3006*	*0.6808*	*0.5161*	*0.9876*
U-Net con Data Augmentation (Train)	*0.0941*	*0.9550*	*0.9138*	*0.9990*
U-Net con Data Augmentation (Val)	*0.0670*	*0.9682*	*0.9385*	*0.9992*

Tabla 3. Comparación de métricas de rendimiento, ensayo de Data Augmentation.

Con esto se ha demostrado la importancia del *Data Augmentation* y de la regularización en general, que ayuda mucho para el entrenamiento de modelos donde hay un desbalance de clases (forzado a propósito para ponérselo difícil al modelo) en los diferentes datasets utilizados, así como puede ayudar a aumentar sintéticamente el número de imágenes para el entrenamiento en casos donde no se dispongan demasiadas, u obtenerlas sea muy caro, como algunos casos biomédicos.

Es por esto por lo que a partir de ahora se va a utilizar *Data Augmentation* en todos los experimentos a realizar.

6.3. Elección de la función de pérdida

Como se ha expuesto antes (Apartado 5.3), se han planteado tres modelos diferentes de función de pérdida para utilizar, así como sus ventajas y desventajas teóricas. Estos tres modelos son; DICE, BCE y el combinado de ambos. Se va a realizar el entrenamiento y validación del modelo de U-Net clásica con cada una de ellas y se van a comparar cuantitativa y cualitativamente.

Con todos los modelos se obtuvieron resultados objetivos muy similares, y se llegó a la convergencia del modelo con facilidad:

Modelo	***Loss***	***DICE***	***IoU***	***Accuracy***
U-Net con DICE (Train)	*0.0941*	*0.9550*	*0.9138*	*0.9990*
U-Net con DICE (Val)	*0.0670*	*0.9682*	*0.9385*	*0.9992*
U-Net con BCE (Train)	*0.0043*	*0.9694*	*0.9405*	*0.9994*
U-Net con BCE (Val)	*0. 0.0047*	*0.9766*	*0.9544*	*0.9994*
U-Net con DICE+BCE (Train)	*0.1027*	*0.9749*	*0.9510*	*0.9996*
U-Net con DICE+BCE (Val)	*0.0810*	*0.9674*	*0.9369*	*0.9992*

Tabla 4. Comparación de métricas de rendimiento, ensayo de Funciones de Pérdida.

Sin embargo, al comparar las predicciones con las máscaras verdaderas, sí que se pueden vislumbrar las fortalezas y desventajas de cada tipo de función de pérdida.

DICE funciona extremadamente bien en general, con resultados muy precisos, pero es más susceptible a dar un resultado erróneo en casos límite que el modelo no haya aprendido correctamente. Estos errores se visualizan como huecos en el interior de las máscaras, pese a que los bordes están muy bien definidos.

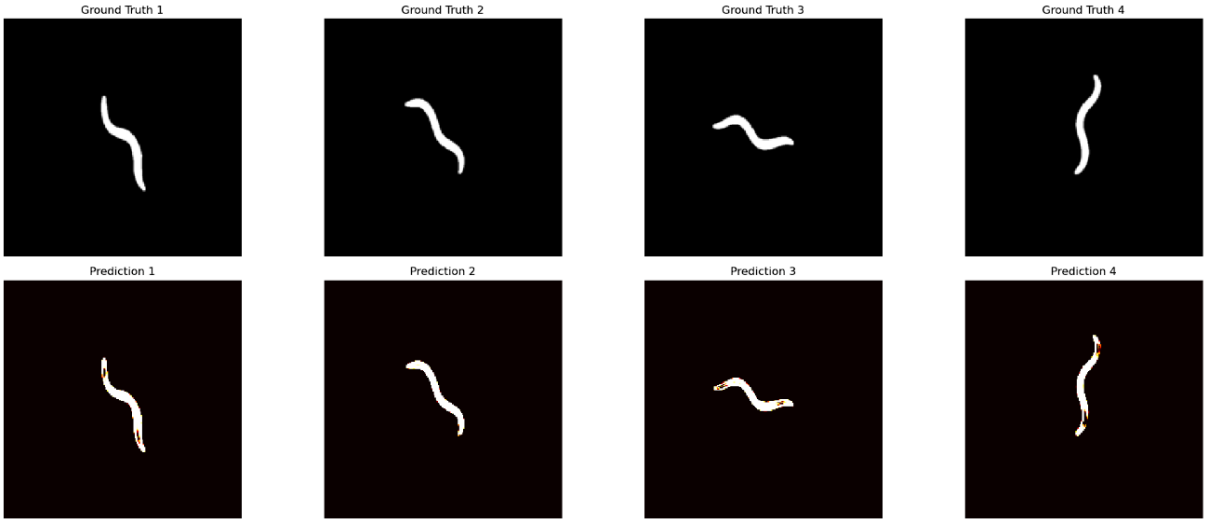

Figura 35. Comparación entre predicciones del modelo y Ground Truth con DICE Loss (Dataset de test).

BCE es mucho más robusto, y rara vez da un resultado erróneo, aunque tiene más problemas a la hora de segmentar los bordes de los objetos de interés y tiene un peor rendimiento.

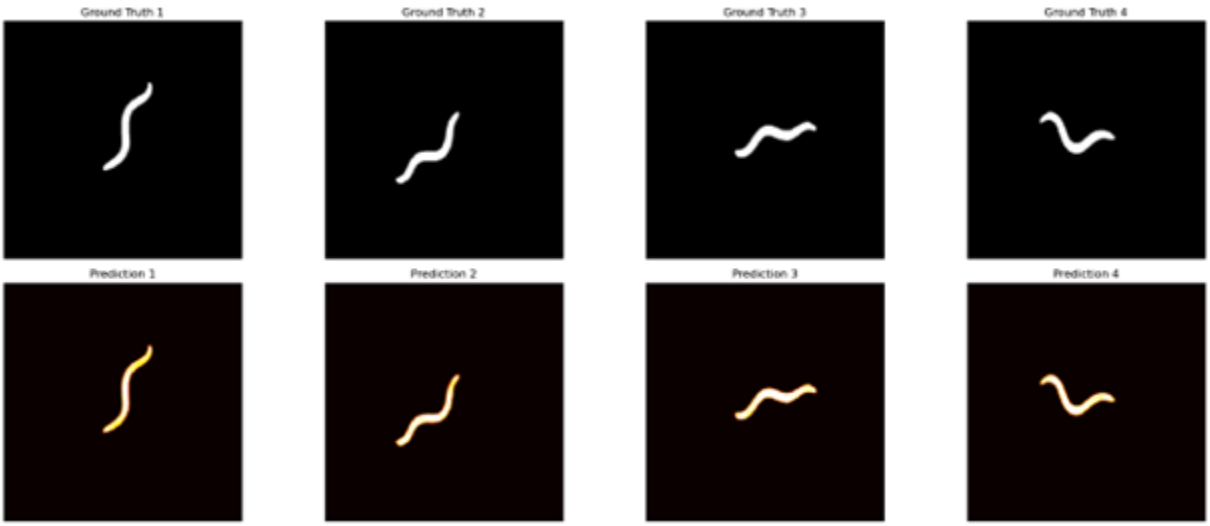

Figura 36. Comparación entre predicciones del modelo y Ground Truth con BCE Loss (Dataset de test).

La función combinada de DICE + BCE combina las fortalezas de ambos y consigue un resultado tanto robusto como preciso, y es por esto por lo que se utilizará a partir de ahora para el resto de los entrenamientos.

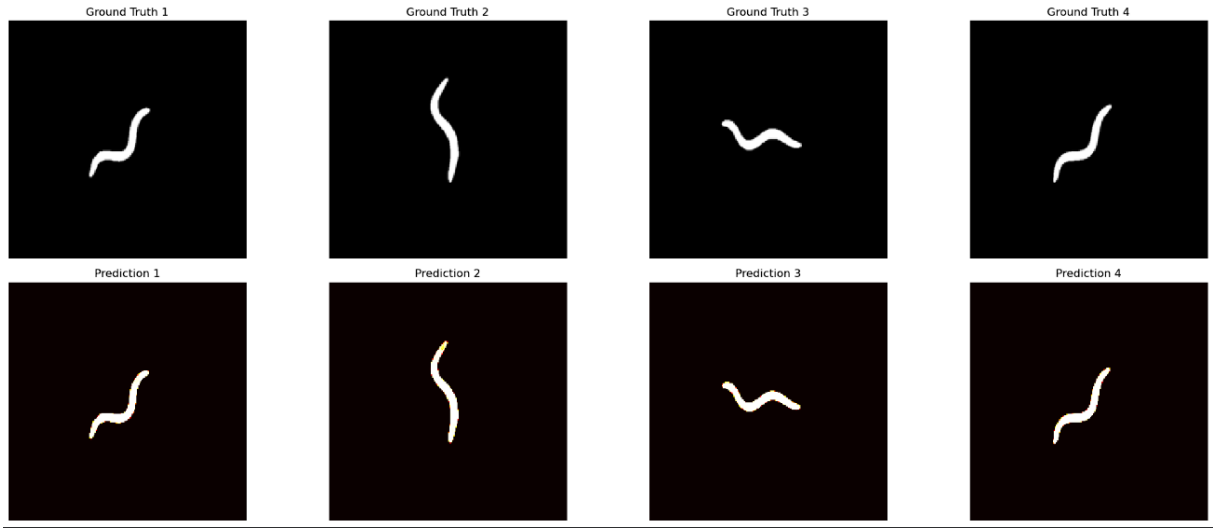

Figura 37. Comparación entre predicciones del modelo y Ground Truth con DICE+BCE Loss (Dataset de test).

6.4. Comparativa entre U-Net y U-Net++

Una de las formas más efectivas de evaluar la eficiencia de una arquitectura, es compararla con otras que buscan resolver el mismo problema. Es por esto por lo que se va a realizar una comparativa entre el modelo de U-Net clásico y su variante más moderna, la U-Net++. Ambos modelos deberían resolver el problema de la segmentación sin problema, pero la U-Net++ viene con implementaciones novedosas que prometen, no solo mejorar la calidad de la segmentación, sino también optimizar la velocidad de entrenamiento y la eficiencia.

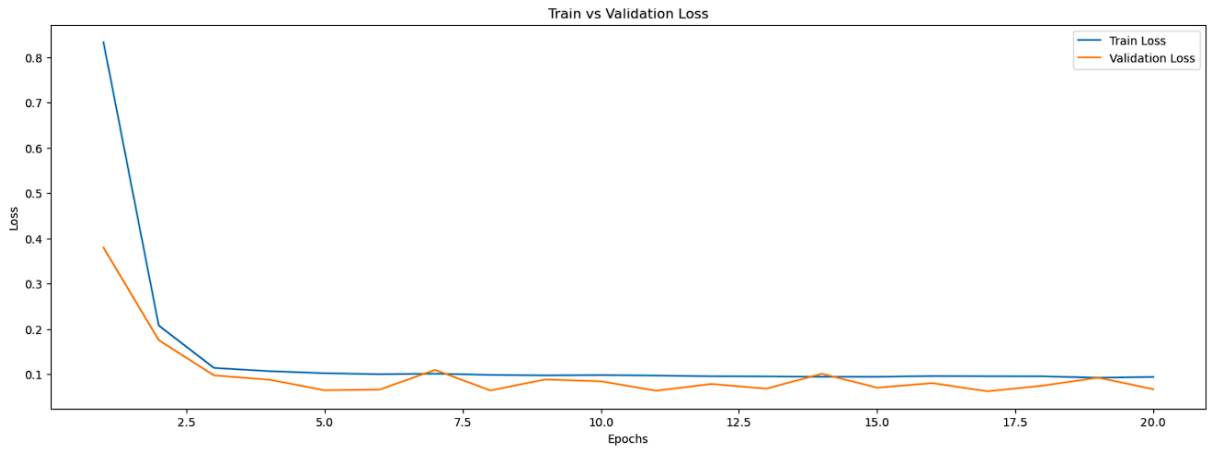

Figura 38. Train Loss vs Validation Loss U-Net convencional.

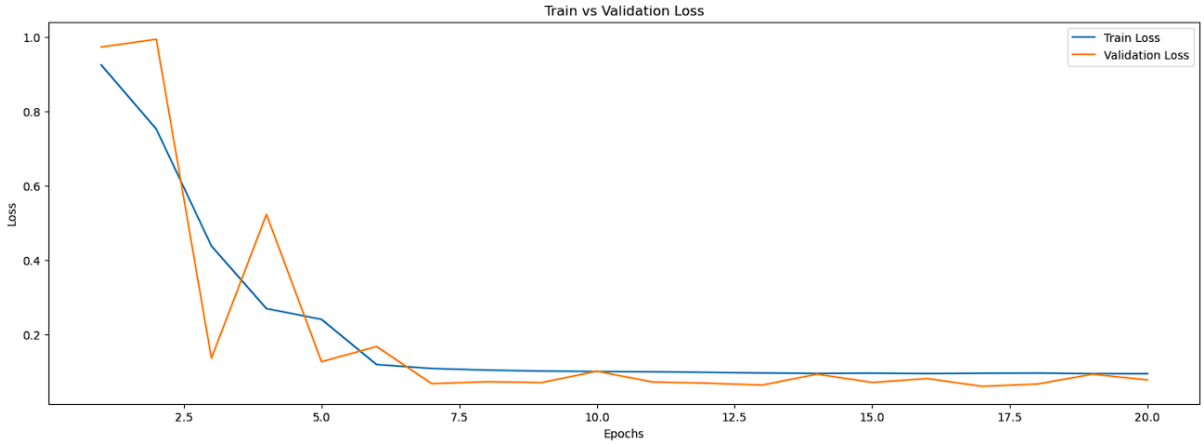

Figura 39 . Train Loss vs Validation Loss U-Net++.

Viendo las funciones de pérdida, para el caso de estudio del proyecto, la U-Net++ tiene más problemas a la hora de encontrar el mínimo global durante el entrenamiento, tardando hasta tres veces más que el modelo convencional en converger. Por lo que, respecto a la velocidad de entrenamiento, no proporciona ninguna mejora significativa.

En cuanto a las métricas de evaluación, en ambos modelos se obtienen rendimientos objetivos muy similares, lo que indica que ambas arquitecturas son efectivas para realizar la segmentación de imágenes. Se observan datos en las métricas de rendimiento muy similares para ambos modelos, con una diferencia despreciable entre ellos.

Modelo	***Loss***	***DICE***	***IoU***	***Accuracy***
U-Net convencional (Train)	*0.0941*	*0.9550*	*0.9138*	*0.9990*
U-Net convencional (Val)	*0.0670*	*0.9682*	*0.9385*	*0.9992*
U-Net++ (Train)	*0.0954*	*0.9598*	*0.9228*	*0.9992*
U-Net++ (Val)	*0.0786*	*0.9602*	*0.9235*	*0.9989*

Tabla 5. Comparación de métricas de rendimiento, U-Net vs U-Net++.

Al comparar visualmente las predicciones generadas por ambos modelos, la diferencia en la calidad de la segmentación es mínima. Ambos modelos logran capturar con precisión las características de los objetos de interés en las imágenes.

Sin embargo, se observa una ligera ventaja en el caso de la U-Net++, aunque casi despreciable. Esta pequeña mejora podría atribuirse a la capacidad de poder capturar detalles en múltiples resoluciones gracias a las conexiones anidadas.

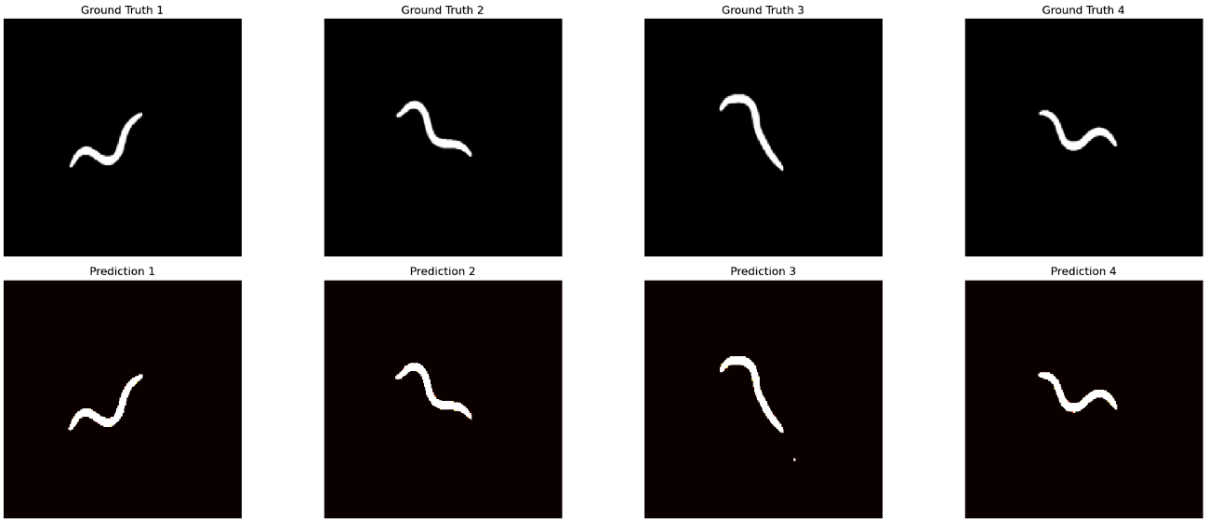

Figura 40. Comparación entre predicciones del modelo y Ground Truth U-Net++ (Dataset de test)

En conclusión, es probable que el modelo U-Net++ consiga mejores rendimientos para un problema de segmentación más complicado, donde aumentar la complejidad y profundidad del modelo sea necesario. Pero para este caso de estudio no tiene una mejora lo suficientemente significativa como para utilizarlo frente a la arquitectura clásica, la cual es mucho más sencilla.

6.5. Entrenamiento del modelo definitivo

Finalmente, tras una exhaustiva comparación y análisis de diferentes arquitecturas y configuraciones, se ha decidido utilizar el modelo U-Net convencional para entrenar el modelo definitivo. Este modelo es preciso, robusto y eficiente para las necesidades específicas de este proyecto.

Como se ha demostrado en análisis previos, se incorporarán técnicas de *Data Augmentation* para aumentar la capacidad de generalización del modelo y evitar el *overfitting*, para que el modelo sea más versátil frente a datos nunca antes vistos.

En cuanto a la función de pérdida, se va a utilizar una combinación de DICE Loss y BCE Loss. De esta manera, se busca aprovechar las fortalezas individuales de cada una de estas funciones de pérdida. La eficacia de DICE en la detección de bordes, y la robustez de BCE en la clasificación píxel a píxel.

Para obtener un mejor resultado esta vez, y ya que solo se va a entrenar una única vez y no hacer diferentes pruebas que podrían llevar mucho tiempo para compararlas, se va a aumentar la resolución de las imágenes al tamaño original (480x640 píxeles) para realizar el entrenamiento, a costa de reducir el tamaño de lote a dos imágenes por lote por limitaciones de memoria en la GPU. Esto, pese a que aumente considerablemente el tiempo requerido de entrenamiento, aumentará el rendimiento en gran mesura.

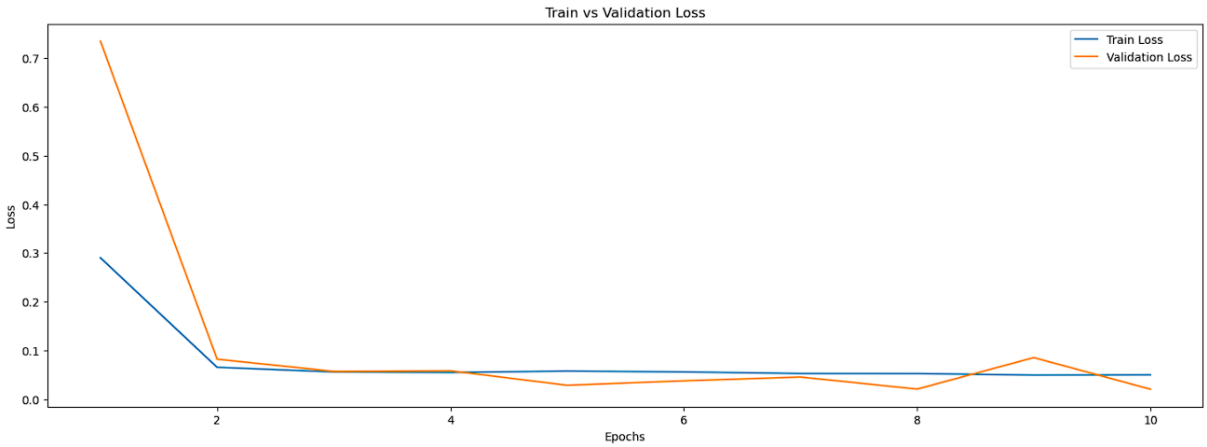

Figura 41. Evolución de las funciones de pérdida del modelo definitivo.

Se observa como ambas funciones de pérdida convergen a un valor muy pequeño y similar (figura anterior), por lo que se puede afirmar que el modelo ha sido capaz de generalizar para predecir correctamente datos nunca vistos. Si bien la pérdida de validación puede ser ligeramente menor que la de entrenamiento, esto se debe a que las imágenes del dataset de validación son mucho más fáciles para el modelo una vez ha conseguido generalizar, y a algunos fallos despreciables en las máscaras verdaderas de entrenamiento.

Modelo	***Loss***	***DICE***	***IoU***	***Accuracy***
U-Net Fase Entrenamiento	0.0498	0.9820	0.9647	0.9998
U-Net Fase Validación	0.0204	0.9902	0.9806	0.9997

Tabla 6. Métricas de rendimiento al final del entrenamiento del modelo definitivo.

En cuanto a las métricas de rendimiento al final del entrenamiento (Tabla 6), se observan unos valores muy elevados, tanto para entrenamiento como validación, indicando así la validez del modelo para la realización de la tarea de segmentación.

También se visualiza la evolución de estas métricas a lo largo de las diferentes iteraciones es las siguientes gráficas, tanto para el ensayo de entrenamiento como el de validación.

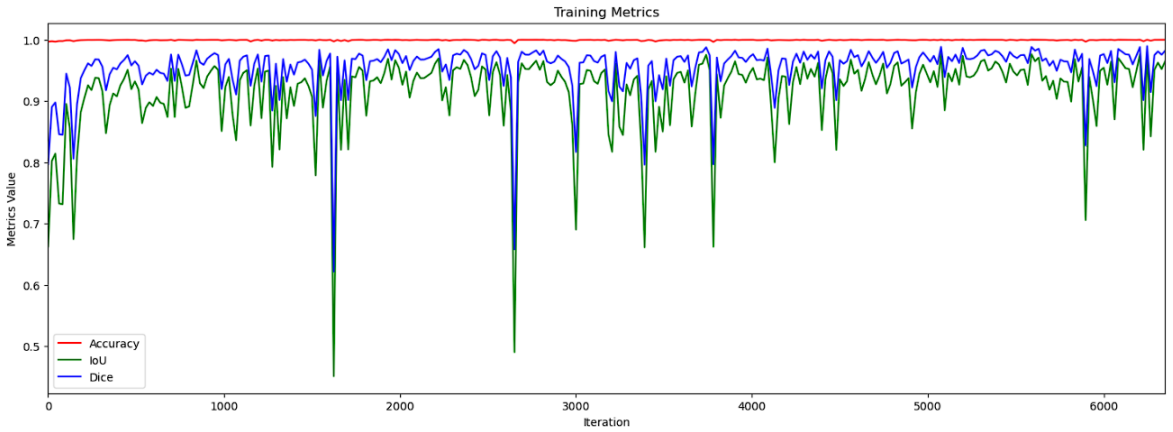

Figura 42. Evolución de las métricas de rendimiento en el modelo definitivo. Ensayo de entrenamiento.

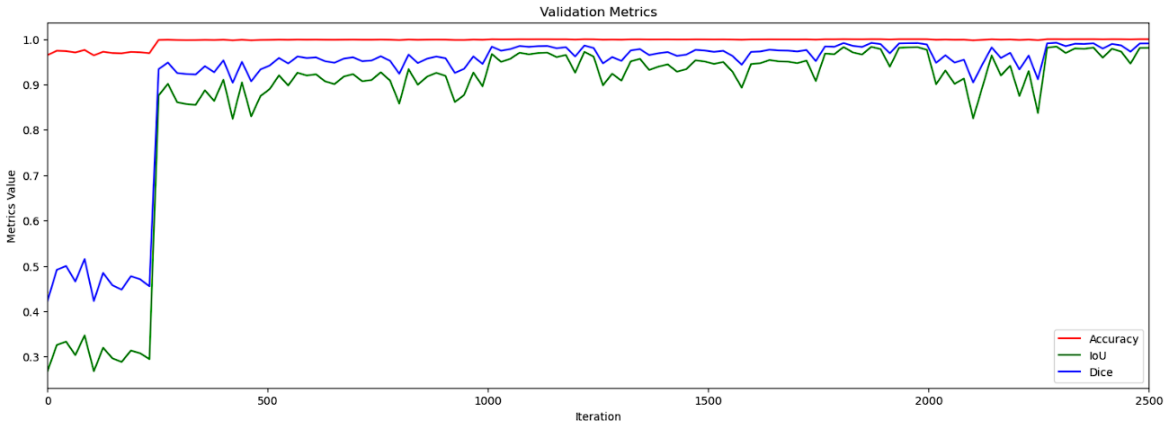

Figura 43. Evolución de las métricas de rendimiento en el modelo definitivo. Ensayo de validación.

Se observa como las métricas de *IoU* y *DICE* aumentan y disminuyen a la par que lo hacía la función de pérdida correspondiente, comportándose por lo tanto de la manera esperada, y estabilizándose al final del entrenamiento.

También se observa que la métrica de Accuracy resulta poco útil en este caso debido al desequilibrio significativo entre los píxeles ocupados por el *C. elegans* y el fondo. Este desequilibrio hace que la métrica presente valores elevados y poco variables, lo que la hace poco informativa.

En la evolución de las métricas de entrenamiento, se pueden asociar esos picos de caída de rendimiento en algunos *batches* en los que unas de las máscaras de "Ground Truth" puede tener errores, y al tener tantos *minibatches* por ser de menor tamaño (dos imágenes por *minibatch*), se muestran de manera más pronunciada. De todas maneras, no debería ser inconveniente ya que pese a esos errores el modelo es capaz de generalizar correctamente, y si se comprueban los resultados predichos de entrenamiento se ve como segmentan correctamente, aunque la segmentación verdadera fuera errónea.

A continuación, se visualizan las máscaras predichas por el modelo, colocadas encima de la imagen original para observar su grado de superposición y rendimiento de manera cualitativa, para cada uno de los datasets utilizados.

Train Dataset

Como se esperaba, se muestra una alta similitud entre las máscaras predichas y las reales en el conjunto de entrenamiento.

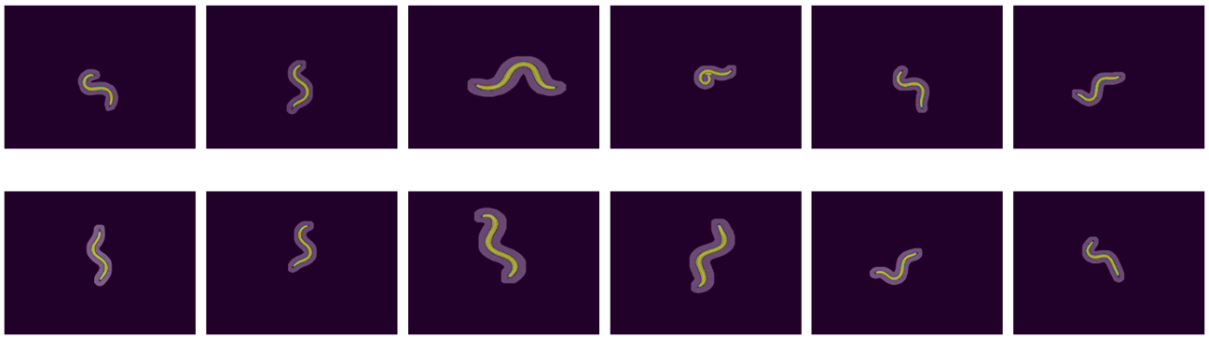

Figura 44. Máscaras predichas por el modelo sobre las imágenes originales. Datos de entrenamiento.

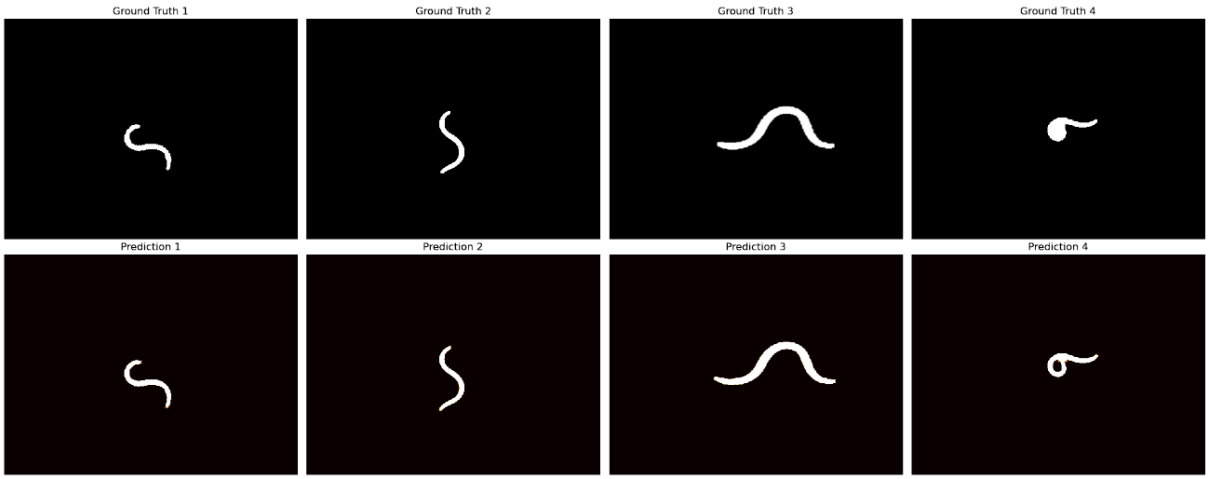

Figura 45. Comparativa entre Ground Truths y máscaras predichas por el modelo. Datos de entrenamiento.

Lo bueno del modelo entrenado, es que es capaz de generalizar y entender en gran detalle las características de las imágenes a procesar, de tal manera que obtiene en algunos casos un resultado correcto respecto a la realidad, cuando la máscara que se utilizó para el entrenamiento y que se generó al crear el dataset era claramente errónea (Figura 45).

Esto demuestra la potencia que tiene para segmentar imágenes con inconsistencias de iluminación u otros casos límite, que el método de creación del dataset por técnicas de visión artificial convencionales no fue capaz de resolver.

Validation Dataset

Al igual que para el dataset de entrenamiento, y como se ha observado en las métricas de rendimiento, el resultado esperado iba a ser muy positivo. El modelo consigue predecir la máscara del gusano con muy buena precisión.

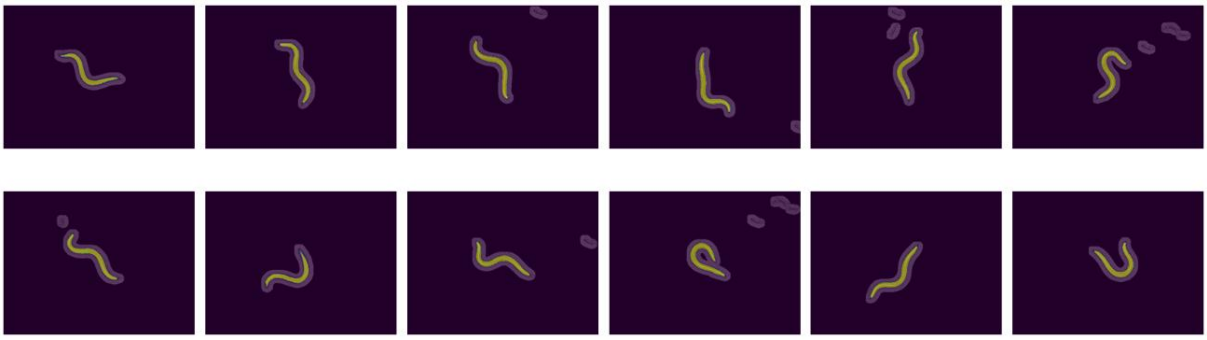

Figura 46. Máscaras predichas por el modelo sobre las imágenes originales. Datos de validación.

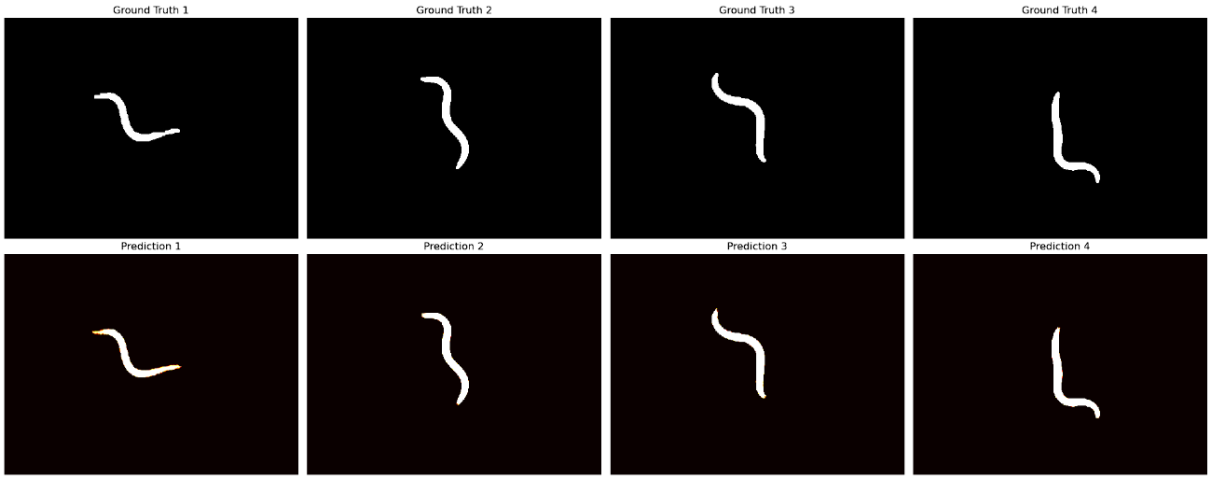

Figura 47. Comparativa entre Ground Truths y máscaras predichas por el modelo. Datos de validación.

Test Dataset

En el dataset de test, se obtiene un coeficiente DICE de 0.9907, por lo que se vuelve a demostrar de nuevo el rendimiento del modelo y su capacidad de generalizar.

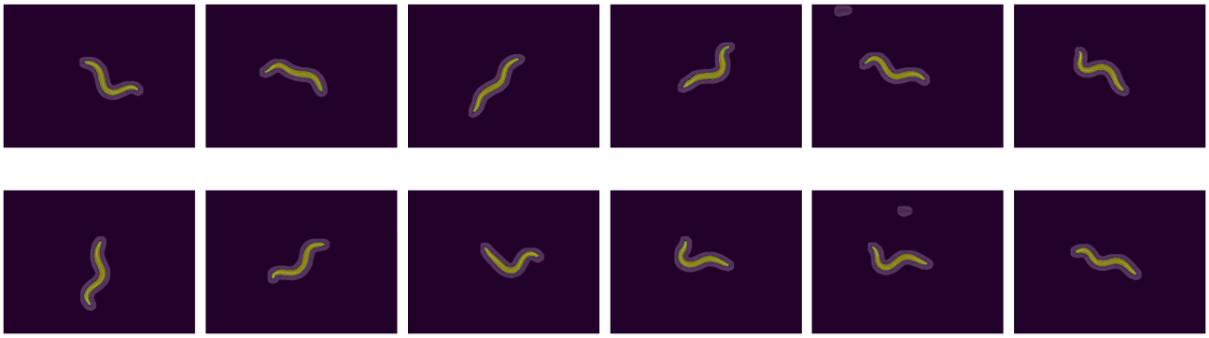

Figura 48. Máscaras predichas por el modelo sobre las imágenes originales. Datos de test.

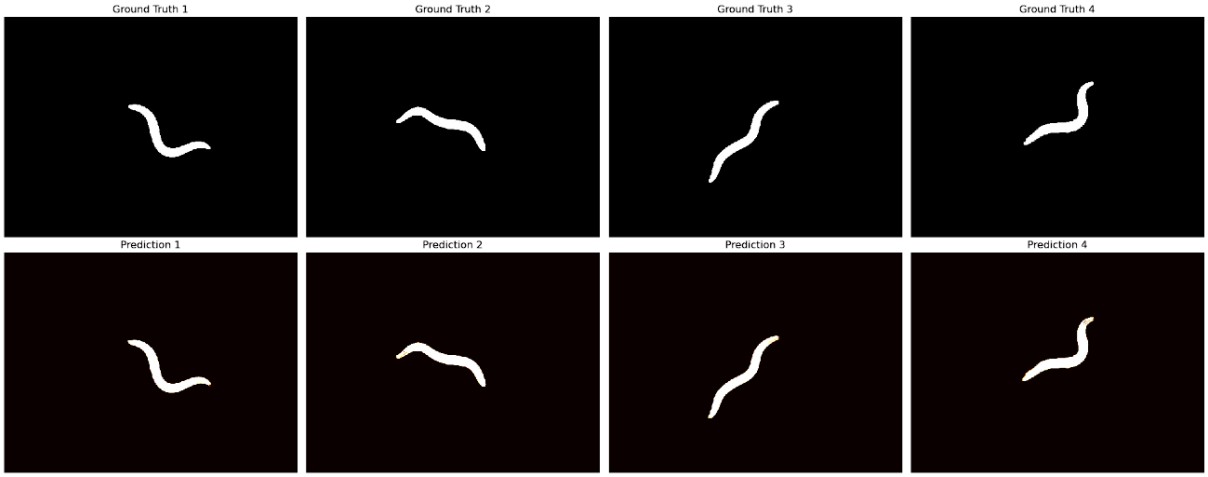

Figura 49. Comparativa entre Ground Truths y máscaras predichas por el modelo. Datos de test.

En resumen, se demuestra la eficacia del modelo U-Net convencional en la tarea de segmentación de imágenes de *C. elegans*. Este resultado se ha conseguido a través de una cuidadosa selección y ajuste de hiperparámetros, así como la incorporación de diversas técnicas de regularización como *Data Augmentation* y una combinación de diferentes funciones de pérdida. Sin embargo, se identificó que la métrica de Accuracy no es adecuada para este caso específico debido al desequilibrio en la distribución de píxeles.

7. CÁLCULO DE LAS CARACTERÍSTICAS DE MOVILIDAD

Se realizarán cálculos de algunas de las características de movilidad del *C. elegans*, tanto utilizando el Tierpsy Tracker como por un script personalizado para el proyecto. El objetivo es comparar los resultados para evaluar la efectividad de la segmentación del modelo U-Net, así como los cálculos realizados sobre las imágenes segmentadas.

Para lograr esta comparación, se utilizará el mismo vídeo original para ambos métodos. Los resultados del Tierpsy Tracker en archivos de formato *hdf5* ya están disponibles como se explicó al principio de la memoria (Apartado 1.4). Los resultados del script personalizado se calculan sobre el vídeo con la máscara predicha por el modelo.

Mientras que las características del Tierpsy Tracker se calcularon a partir del vídeo original que contenía la totalidad de la placa Petri con todos los *C. elegans*, sí que se han podido calcular características como velocidades y trayectorias. En cambio, como para la segmentación con U-Net solo se disponen de los vídeos de los nematodos individuales siendo seguidos por la cámara microscópica, no se tienen las referencias suficientes para calcular estas velocidades o trayectorias, aunque sí que se calcularán y compararán características morfológicas y de posición.

7.1. POST PROCESAMIENTO DE LOS RESULTADOS

Una vez entrenado el modelo definitivo, se podrá utilizar para segmentar con precisión la secuencia de imágenes o vídeos de *C. elegans*, obteniendo las máscaras predichas. Esto se consigue de forma sencilla realizando inferencia sobre los fotogramas de forma individual, y agregando los resultados en un vídeo al terminar.

En este caso, se va a realizar la segmentación de un video de una cepa no perteneciente a la que se usó para el entrenamiento, para evaluar aún más la capacidad de generalización del modelo.

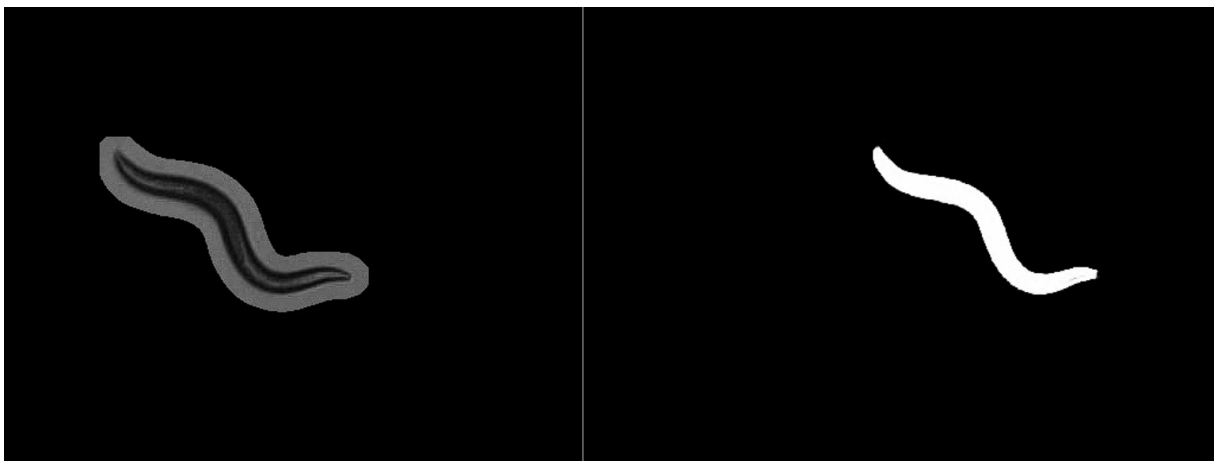

Ilustración 1. Video original vs video con las predicciones del modelo

Una vez realizada la inferencia, se realiza un breve post procesamiento de las máscaras predichas para subsanar posibles errores ocurridos, como huecos dentro del nematodo, o bien cuerpos flotantes que no pertenezcan al objeto de interés, como trozos de comida. Esto se realiza con operaciones de apertura y cierre, y filtrando además los objetos de interés detectados por área.

7.2. Tierpsy Tracker

En primer lugar, se han calculado mediante el Tierpsy Tracker de manera automatizada muchas de las características de movilidad de los *C. elegans*, a partir del video original. Es el propio programa el que se encarga de segmentar a su manera por técnicas avanzadas de *thresholding*, y realiza los cálculos sobre las máscaras obtenidas.

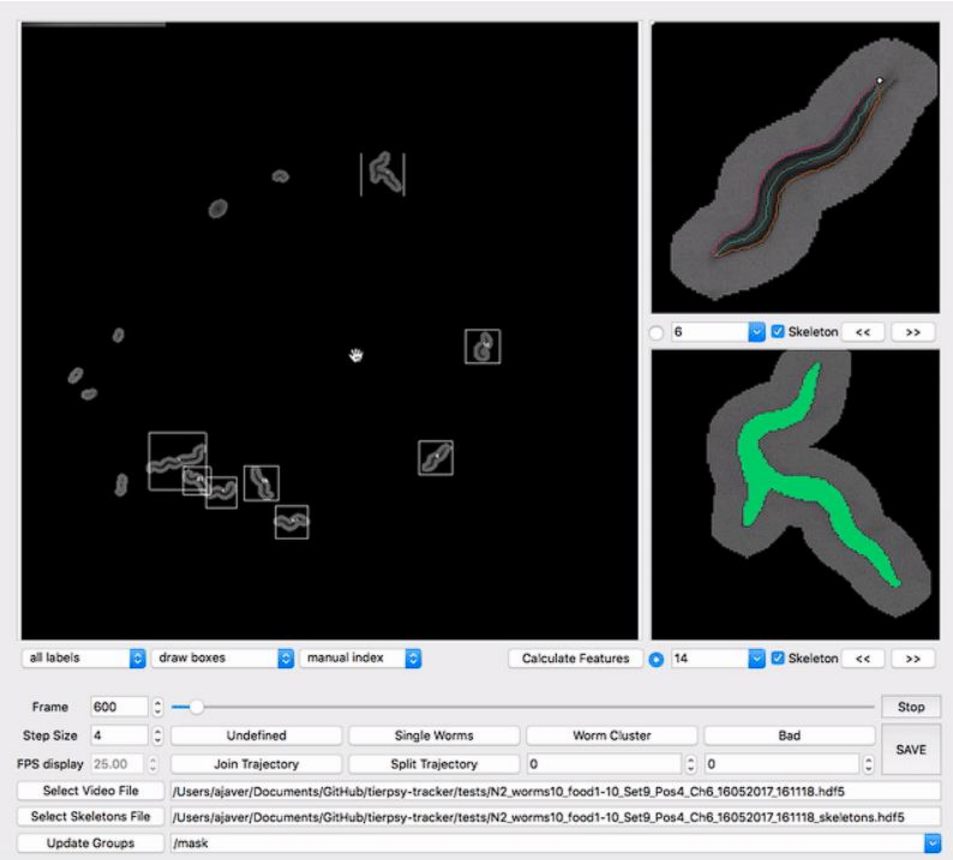

Figura 50. Interfaz gráfica de Tierpsy Tracker.
(https://github.com/Tierpsy/tierpsy-tracker/blob/development/docs/HOWTO.md)

Tras analizar el vídeo con Tierpsy Tracker, se pueden seleccionar clicando en la imagen principal los diferentes *C. elegans* detectados. Una vez seleccionado, se pueden indicar que se quieren calcular las características sobre él, y se obtienen los archivos *hdf5* con todas las características de movimiento calculadas.

Dentro de estos archivos *hdf5* se encuentran estructuradas diferentes tablas, que contienen el valor de las coordenadas de los esqueletos del nematodo, características de movimiento, ya sean a lo largo del tiempo o la media de todo el vídeo y otros metadatos de interés.

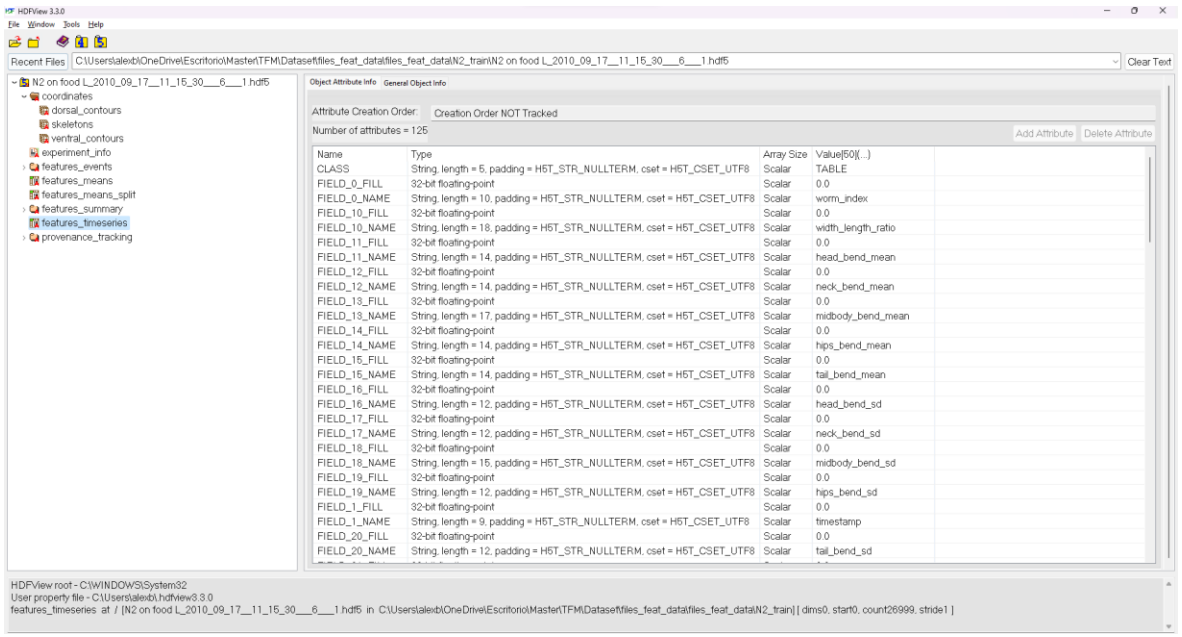

Figura 51. Estructura de datos dentro del archivo hdf5.

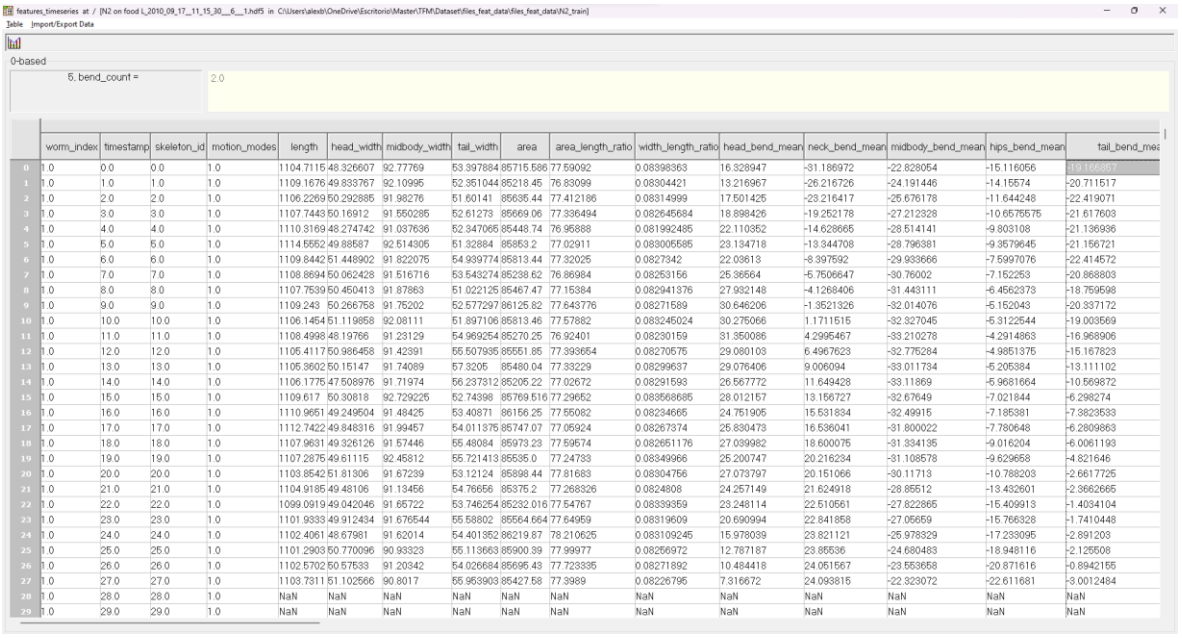

Figura 52. Tabla dentro del archivo hdf5 con características de movilidad calculadas por cada fotograma.

Otro dato importante a extraer para utilizar el mismo en los cálculos con OpenCV es la relación de micras por píxel, que depende de la cámara con la que se tomaron las imágenes en un primer momento. Este dato también se puede encontrar dentro del dataset *hdf5*.

fps	64-bit floating-point	Scalar	30.030030030032737
is_light_background	64-bit integer	Scalar	1
microns_per_pixel	64-bit floating-point	Scalar	4.458167496324821
time_units	String, length = 7, padding = H5T_STR_NULLTERM, cset = H5T_CSET_UTF8	Scalar	seconds
worm_index_type	String, length = 17, padding = H5T_STR_NULLTERM, cset = H5T_CSET_UTF8	Scalar	worm_index_joined
xy_units	String, length = 11, padding = H5T_STR_NULLTERM, cset = H5T_CSET_UTF8	Scalar	micrometers

Figura 53. Dato de micras por píxel extraído del hdf5 resultado del Tierpsy Tracker.

A continuación, se muestran un par de gráficas obtenidas a partir de los datos del Tierpsy Tracker:

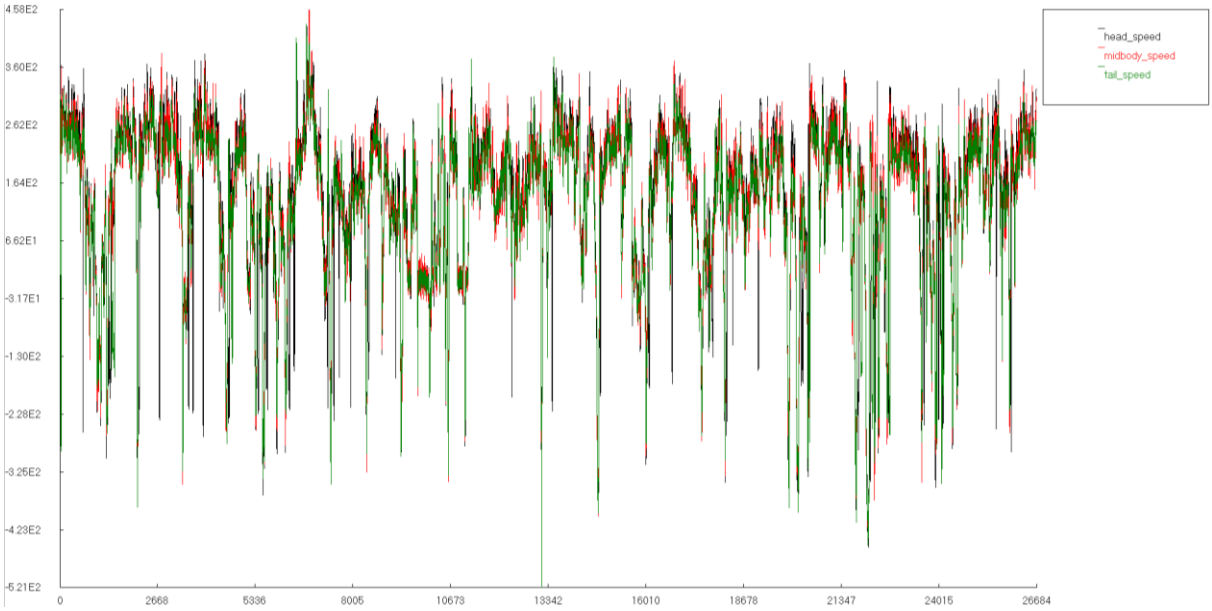

Figura 54. Velocidad (micras/s) de la cabeza, cola y cuerpo del C. elegans a lo largo del tiempo.

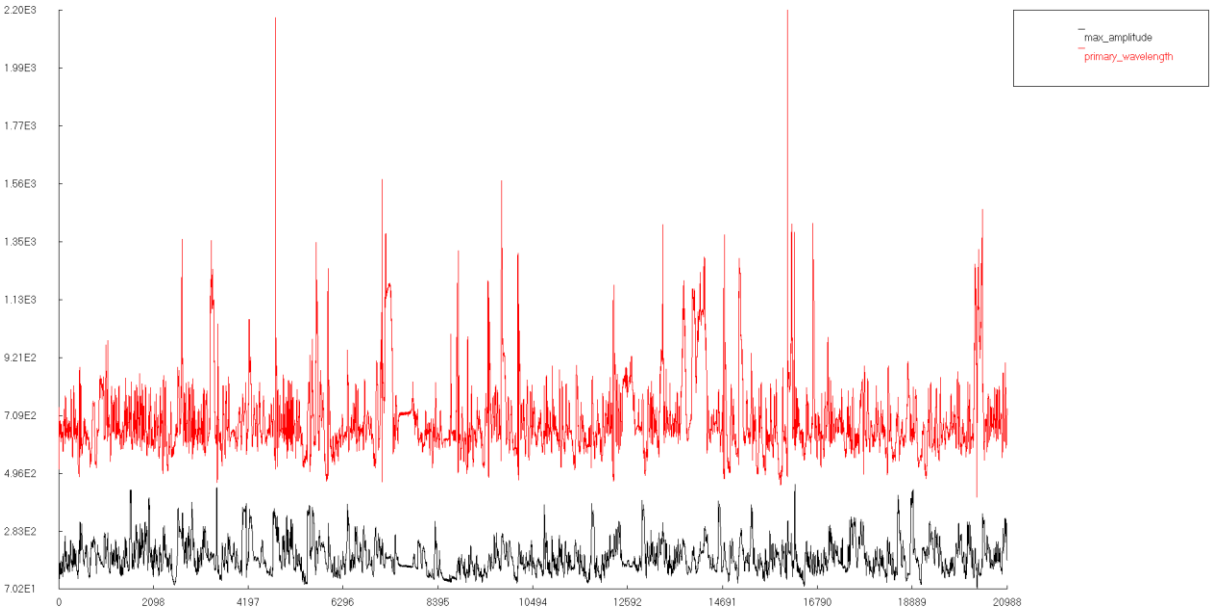

Figura 55. Amplitud máxima y longitud de onda (micras) del C. elegans a lo largo del tiempo.

7.3. Cálculo de características de movilidad con OpenCV

En este apartado se van a calcular algunas de las características de movilidad del *C. elegans* para comparar con los resultados anteriores del Tierpsy Tracker. Para ello se va a hacer uso principalmente de algunas de las funciones incluídas en la librería de visión por computador OpenCV.

Características Morfológicas y posturas

Lo primero que se va a calcular del nematodo son sus características morfológicas. En primer lugar, se obtiene el contorno del *C. elegans* segmentado, sobre el cual se aplican diferentes operaciones como la esqueletonización.

Sobre el esqueleto del *C. elegans*, se puede calcular la longitud total del gusano, sumando la distancia entre todos los puntos que forman el esqueleto, así como el punto del centro del cuerpo (centro del esqueleto) y los extremos, correspondientes a la cabeza y la cola.

Sobre el contorno se calcula también el área del *C. elegans* y el mínimo rectángulo delimitador.

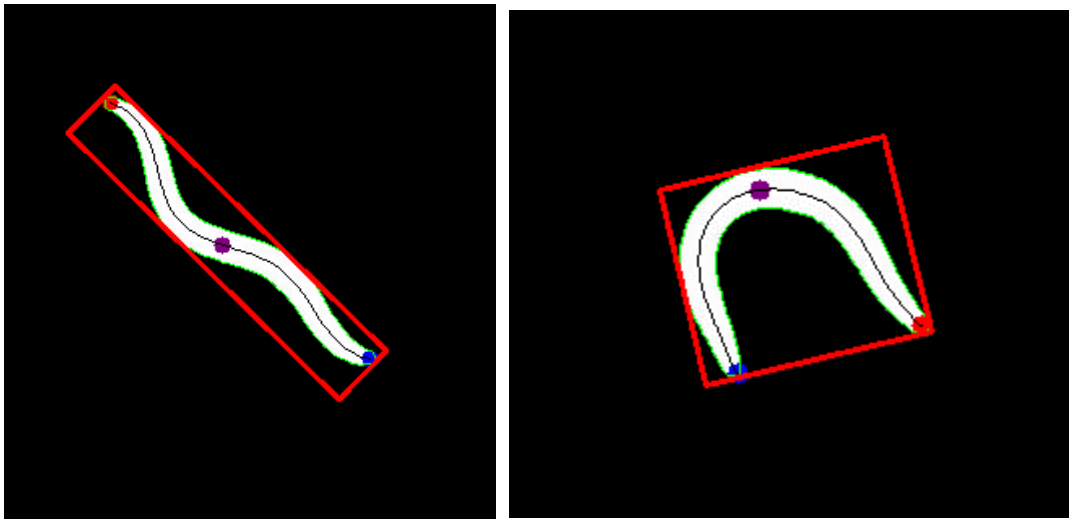

Figura 56. Fotogramas de C. elegans segmentados (blanco) sobre los cuales se ha obtenido el contorno (verde), el mínimo rectángulo delimitador (rojo), el esqueleto (negro), los puntos de cabeza y cola (rojo y azul) y el punto central del cuerpo (morado).

Con este último rectángulo delimitador mínimo, se extrae el eje máximo y mínimo sobre el cual se puede encuadrar al *C. elegans*, y con estos ejes se puede calcular la característica de "*Quirkiness*", donde "a" es el eje menor, y "A" el eje mayor

$$Q = \sqrt{1 - \left(\frac{a^2}{A^2}\right)}$$

Ecuación 8. Ecuación de Quirkiness.

Se representa esta variable a lo largo del tiempo para el vídeo analizado, y se obtiene el siguiente resultado:

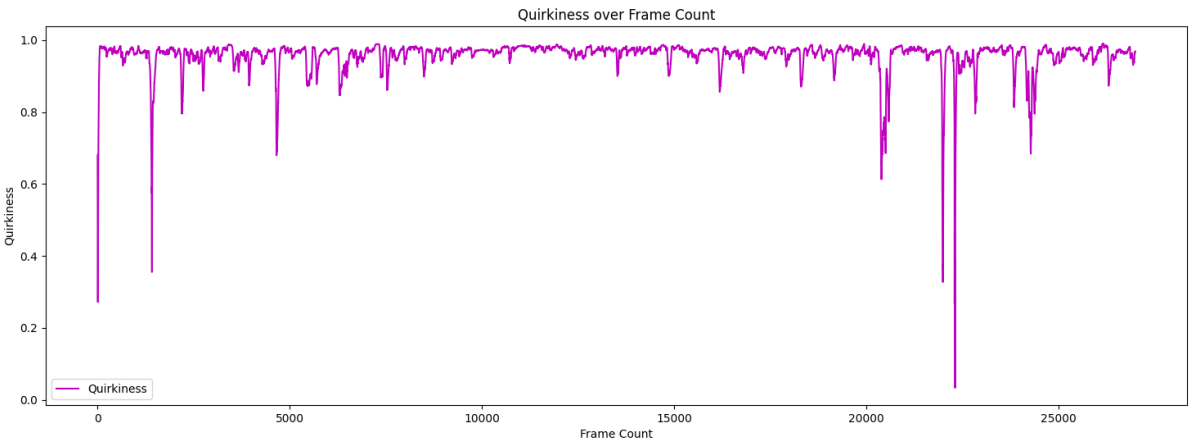

Figura 57. Valor de Quirkiness a lo largo del tiempo.

Los picos en los que decrece de manera significativa este valor se corresponden a los fotogramas en los que el nematodo no se encuentra avanzando en línea recta, sino que está en medio de un giro, y se encuentra en forma de “U” o similar.

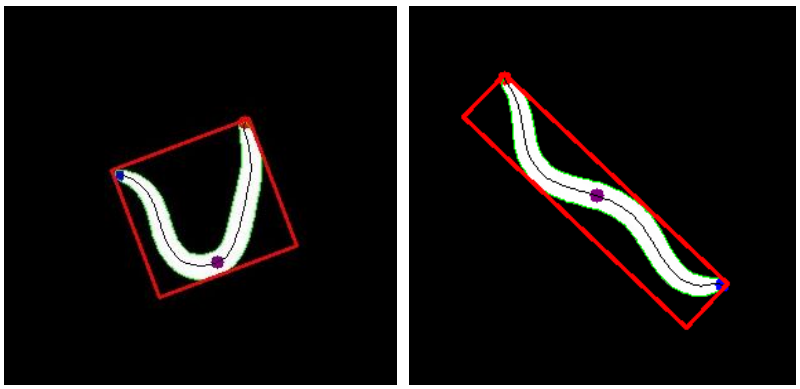

Figura 58. Captura de Quirkiness muy bajo, C elegans en forma de “U” vs captura con Quirkiness promedio (cerca de 0.95)

7.4. Comparación de características

A continuación, se compararán las características calculadas manualmente con aquellas extraídas con el *Tierpsy Tracker*.

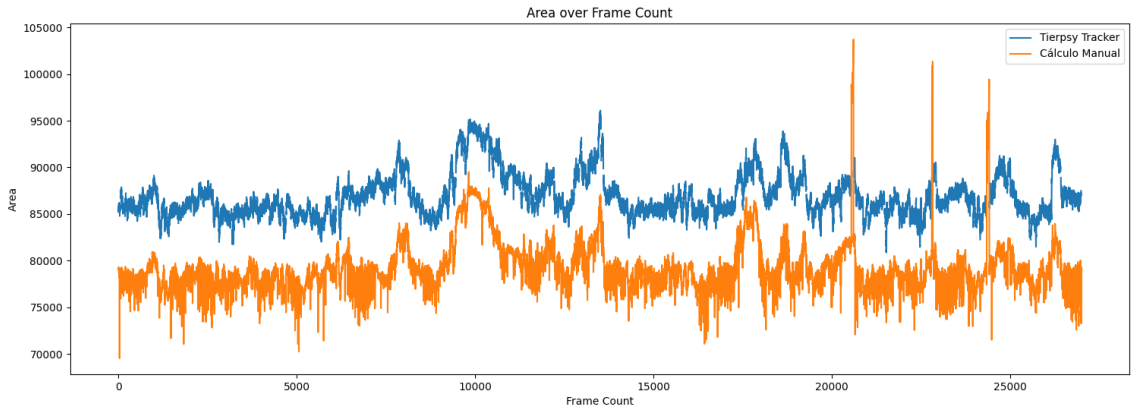

Figura 59. Comparación del Área de C. elegans calculada Manualmente vs calculada por Tierpsy Tracker.

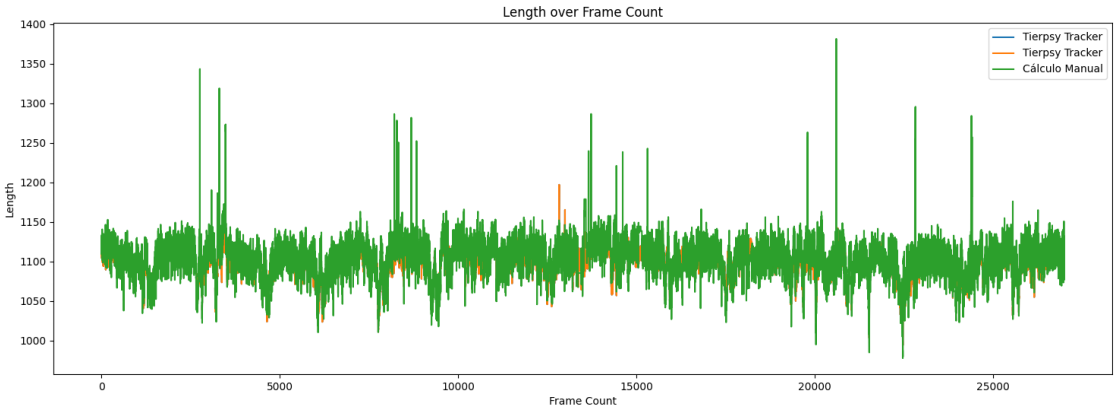

Figura 60. Comparación de la Longitud de C. elegans calculada Manualmente vs calculada por Tierpsy Tracker.

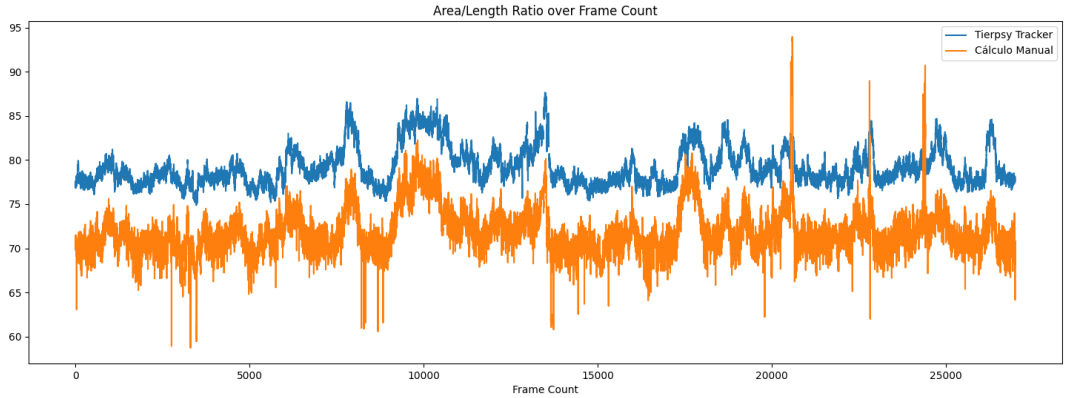

Figura 61. Comparación del Ratio Área/Longitud de C. elegans calculada Manualmente vs calculada por Tierpsy Tracker.

En general, se obtienen resultados razonablemente similares para las características comparadas. Los datos de longitud son iguales, y se observa cierta diferencia en cuanto al área del *C. elegans*, la cual se ve reflejada más tarde en las diferencias entre los ratios Área/Longitud.

Esta diferencia puede deberse a la manera de segmentar y obtener los contornos por parte del Tierpsy Tracker, que en general tienda a obtener un *C. elegans* ligeramente más ancho, lo que produce esta diferencia del 10% aproximadamente.

En cuanto a los picos que se ven en los datos calculados manualmente, se deben generalmente a posiciones especiales de los *C. elegans*, como por ejemplo aumenta considerablemente el área de estos cuando se encuentran en posición “Omega” (figura).

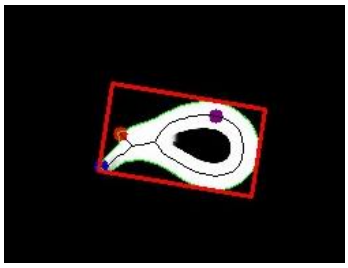

Figura 62. C. elegans en posición "Omega" que ocupa mayor área.

8. CONCLUSIÓN Y TRABAJOS FUTUROS

En resumen, este estudio demuestra la eficacia del modelo U-Net convencional en la tarea de segmentación de imágenes de *C. elegans*. A través de una cuidadosa selección y ajuste de hiperparámetros, así como la incorporación de técnicas de *Data Augmentation* y una combinación de funciones de pérdida, se logró un rendimiento elevado en las métricas evaluadas. Sin embargo, se identificó que la métrica de Accuracy no es adecuada para este caso específico debido al desequilibrio en la distribución de píxeles en las clases.

En relación con los resultados obtenidos tras el ensayo de *Healthspan*, se observa una alta correlación entre los datos calculados a partir de las máscaras predichas por el modelo, y los resultados obtenidos por el software especializado Tierpsy Tracker. Esto reafirma de nuevo la validez tanto de los cálculos de características de movimiento como del modelo entrenado para automatizar este tipo de ensayos.

La realización de este Trabajo de Fin de Máster ha servido como una introducción significativa en el ámbito de las redes neuronales y los ensayos de *Healthspan* en *C. elegans*. El proyecto ha permitido un enfoque centrado en la comprensión exhaustiva de diferentes hiperparámetros, así como la exploración de avances recientes en arquitecturas como la U-Net. Además, se han estudiado multitud de técnicas de regularización y otros conceptos avanzados como el método de la super convergencia. Esto ha permitido un enriquecimiento del conocimiento en el campo.

En futuras investigaciones se podría estudiar la viabilidad del modelo sobre cómo puede funcionar para segmentar *C. elegans* con un sistema de captura de imágenes diferente al utilizado para este proyecto, o bien para segmentar otros objetos de interés completamente diferentes gracias a todos los conocimientos adquiridos.

Debido a la escasez de recursos de hardware como GPUs para entrenar el modelo, se ha visto limitado ligeramente el número de entrenamientos y comparaciones que se podrían haber realizado, por eso se sugiere la idea de continuar mejorando el modelo en trabajos futuros.

Por limitaciones del dataset disponible y la complejidad inherente a la tarea, no se han calculado algunas de las características de movilidad del *C. elegans*. El grueso de este proyecto se ha centrado en el modelo de red neuronal, y se considera que el cálculo preciso de todas las variables es una tarea compleja que excede el alcance del proyecto actual. Esta funcionalidad podría implementarse como continuación del proyecto o mediante la reutilización de programas ya existentes, como el Tierpsy Tracker a partir de las máscaras generadas por el modelo desarrollado.

BIBLIOGRAFÍA

Kaggle: https://www.kaggle.com/

Google Colab: https://colab.research.google.com/?hl=es

Documentación Pytorch: https://pytorch.org/docs/stable/index.html

HDFView: https://www.hdfgroup.org/downloads/hdfview/

Wormlab: https://www.mbfbioscience.com/help/wormlab/Content/Analyses/position%20&%20speed/trackSummary.htm

Archivo de artículos científicos: https://arxiv.org/

Tierpsy Tracker: https://github.com/Tierpsy/tierpsy-tracker/tree/development

Documentación Python: https://docs.python.org/es/3.11/

Documentación OpenCV: https://docs.opencv.org/4.x/

REFERENCIAS

[1] Puchalt, J. C., Sánchez-Salmerón, A. J., Martorell Guerola, P., & Genovés Martínez, S. (2019). Active backlight for automating visual monitoring: An analysis of a lighting control technique for Caenorhabditis elegans cultured on standard Petri plates. PloS one, 14(4), e0215548. https://doi.org/10.1371/journal.pone.0215548

[2] Puchalt, J. C., Sánchez-Salmerón, A. J., Ivorra, E., Genovés Martínez, S., Martínez, R., & Martorell Guerola, P. (2020). Improving lifespan automation for Caenorhabditis elegans by using image processing and a post-processing adaptive data filter. Scientific reports,10(1), 8729. https://doi.org/10.1038/s41598-020-65619-4

[3] Puchalt, J. C., Layana Castro, P. E., & Sánchez-Salmerón, A. J. (2020). Reducing results variance in lifespan machines: an analysis of the influence of vibrotaxis on wild-type Caenorhabditis elegans for the death criterion. Sensors, 20(21), 5981. https://doi.org/10.3390/s20215981

[4] Layana Castro, P. E., Puchalt, J. C., & Sánchez-Salmerón, A. J. (2020). Improving skeleton algorithm for helping Caenorhabditis elegans trackers. Scientific Reports, 10(1), 22247. https://doi.org/10.1038/s41598-020-79430-8

[5] Puchalt, J. C., Sánchez-Salmerón, A. J., Ivorra, E., Llopis, S., Martínez, R., & Martorell, P. (2021). Small flexible automated system for monitoring Caenorhabditis elegans lifespan based on active vision and image processing techniques. Scientific reports, 11(1), 12289. https://doi.org/10.1038/s41598-021-91898-6

[6] García Garví, A., Puchalt, J. C., Layana Castro, P. E., Navarro Moya, F., & Sánchez-Salmerón, A. J. (2021). Towards lifespan automation for Caenorhabditis elegans based on deep learning: analysing convolutional and recurrent neural networks for dead or live classification. Sensors, 21(14), 4943. https://doi.org/10.3390/s21144943

[7] Layana Castro, P. E., Puchalt, J. C., García Garví, A., & Sánchez-Salmerón, A. J. (2021). Caenorhabditis elegans multi-tracker based on a modified skeleton algorithm. Sensors, 21(16), 5622. https://doi.org/10.3390/s21165622

[8] Puchalt, J. C., Gonzalez-Rojo, J. F., Gómez-Escribano, A. P., Vázquez-Manrique, R. P., & Sánchez-Salmerón, A. J. (2022). Multiview motion tracking based on a cartesian robot to monitor Caenorhabditis elegans in standard Petri dishes. Scientific reports, 12(1), 1767. https://doi.org/10.1038/s41598-022-05823-6

[9] Navarro Moya, F., Puchalt, J. C., Layana Castro, P. E., García Garví, A., & Sánchez-Salmerón, A. J. (2022). A new training strategy for spatial transform networks (STN's). Neural Computing and Applications, 34(12), 10081-10092. https://doi.org/10.1007/s00521-022-06993-0

[10] Rico-Guardiola, E. J., Layana-Castro, P. E., García-Garví, A., & Sánchez-Salmerón, A. J. (2022, October). Caenorhabditis elegans detection using yolov5 and faster r-cnn networks. In International Conference on Optimization, Learning Algorithms and Applications (pp. 776-787). Cham: Springer International Publishing. https://doi.org/10.1007/978-3-031-23236-7_53

[11] García-Garví, A., Layana-Castro, P. E., & Sánchez-Salmerón, A. J. (2023). Analysis of a C. elegans lifespan prediction method based on a bimodal neural network and uncertainty estimation. Computational and Structural Biotechnology Journal, 21, 655-664. https://doi.org/10.1016/j.csbj.2022.12.033

[12] Castro, P. E. L., Garví, A. G., & Sánchez-Salmerón, A. J. (2023). Automatic segmentation of Caenorhabditis elegans skeletons in worm aggregations using improved U-Net in low-resolution image sequences. Heliyon, 9(4). https://doi.org/10.1016/j.heliyon.2023.e14715

[13] Layana Castro, P. E., García Garví, A., Navarro Moya, F., & Sánchez-Salmerón, A. J. (2023). Skeletonizing Caenorhabditis elegans Based on U-Net Architectures Trained with a Multi-worm Low-Resolution Synthetic Dataset. International Journal of Computer Vision, 1-17. https://doi.org/10.1007/s11263-023-01818-6

[14] Riddle, D. L., Blumenthal, T., Meyer, B. J., & Priess, J. R. (1997). C. elegans ii. https://pubmed.ncbi.nlm.nih.gov/21413221/

[15] Kenyon, C., Chang, J., Gensch, E., Rudner, A., & Tabtiang, R. (1993). A C. elegans mutant that lives twice as long as wild type. Nature, 366(6454), 461-464. https://doi.org/10.1038/366461a0

[16] Tissenbaum, H. A. (2015). Using C. elegans for aging research. Invertebrate reproduction & development, 59(sup1), 59-63. https://doi.org/10.1080/07924259.2014.940470

[17] Kato, M., & Slack, F. J. (2008). microRNAs: small molecules with big roles–C. elegans to human cancer. Biology of the Cell, 100(2), 71-81. https://doi.org/10.1042/BC20070078

[18] Potts, M. B., & Cameron, S. (2011). Cell lineage and cell death: Caenorhabditis elegans and cancer research. Nature Reviews Cancer, 11(1), 50-58. https://doi.org/10.1038/nrc2984

[19] Tissenbaum, H. A. (2012). Genetics, life span, health span, and the aging process in Caenorhabditis elegans. Journals of Gerontology Series A: Biomedical Sciences and Medical Sciences, 67(5), 503-510. https://doi.org/10.1093/gerona/gls088

[20] Bansal, A., Zhu, L. J., Yen, K., & Tissenbaum, H. A. (2015). Uncoupling lifespan and healthspan in Caenorhabditis elegans longevity mutants. Proceedings of the National Academy of Sciences, 112(3), E277-E286. https://doi.org/10.1073/pnas.1412192112

[21] Hahm, J. H., Kim, S., DiLoreto, R., Shi, C., Lee, S. J. V., Murphy, C. T., & Nam, H. G. (2015). C. elegans maximum velocity correlates with healthspan and is maintained in worms with an insulin receptor mutation. Nature communications, 6(1), 8919. https://doi.org/10.1038/ncomms9919

[22] Pierce-Shimomura, J. T., Morse, T. M., & Lockery, S. R. (1999). The fundamental role of pirouettes in Caenorhabditis elegans chemotaxis. Journal of Neuroscience, 19(21), 9557-9569. https://doi.org/10.1523/JNEUROSCI.19-21-09557.1999

[23] Ward, S. (1973). Chemotaxis by the nematode Caenorhabditis elegans: identification of attractants and analysis of the response by use of mutants. Proceedings of the National Academy of Sciences, 70(3), 817-821. https://doi.org/10.1073/pnas.70.3.817

[24] Geng, W., Cosman, P., Berry, C. C., Feng, Z., & Schafer, W. R. (2004). Automatic tracking, feature extraction and classification of C. elegans phenotypes. IEEE transactions on biomedical engineering, 51(10), 1811-1820. https://doi.org/10.1109/TBME.2004.831532

[25] Javer, A., Ripoll-Sánchez, L., & Brown, A. E. (2018). Powerful and interpretable behavioural features for quantitative phenotyping of Caenorhabditis elegans. Philosophical Transactions of the Royal Society B: Biological Sciences, 373(1758), 20170375. https://doi.org/10.1098/rstb.2017.0375

[26] Javer, A., Currie, M., Lee, C. W., Hokanson, J., Li, K., Martineau, C. N., ... & Brown, A. E. (2018). An open-source platform for analyzing and sharing worm-behavior data. Nature methods, 15(9), 645-646. https://doi.org/10.1038/s41592-018-0112-1. https://doi.org/10.1038/s41592-018-0112-1

[27] Ronneberger, O., Fischer, P., & Brox, T. (2015). U-net: Convolutional networks for biomedical image segmentation. In Medical Image Computing and Computer-Assisted Intervention–MICCAI 2015: 18th International Conference, Munich, Germany, October 5-9, 2015, Proceedings, Part III 18 (pp. 234-241). Springer International Publishing. https://doi.org/10.48550/arXiv.1505.04597

[28] Odena, A., Dumoulin, V., & Olah, C. (2016). Deconvolution and checkerboard artifacts. Distill, 1(10), e3. https://doi.org/10.23915/distill.00003

[29] Ioffe, S., & Szegedy, C. (2015). Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. In Proceedings of the 32nd International Conference on Machine Learning (Vol. 37). https://doi.org/10.48550/arXiv.1502.03167

[30] Zhou, Z., Rahman Siddiquee, M. M., Tajbakhsh, N., & Liang, J. (2018). Unet++: A nested u-net architecture for medical image segmentation. In Deep Learning in Medical Image Analysis and Multimodal Learning for Clinical Decision Support: 4th International Workshop, DLMIA 2018, and 8th International Workshop, ML-CDS 2018, Held in Conjunction with MICCAI 2018, Granada, Spain, September 20, 2018, Proceedings 4 (pp. 3-11). Springer International Publishing. https://doi.org/10.48550/arXiv.1807.10165

[31] Smith, Leslie N. "Cyclical Learning Rates for Training Neural Networks." ArXiv:1506.01186 [Cs], April 4, 2017. http://arxiv.org/abs/1506.01186.

[32] Smith, Leslie N., and Nicholay Topin. "Super-Convergence: Very Fast Training of Neural Networks Using Large Learning Rates." ArXiv:1708.07120 [Cs, Stat], May 17, 2018. http://arxiv.org/abs/1708.07120.

[33] Smith, Leslie N. "A Disciplined Approach to Neural Network Hyper-Parameters: Part 1 -- Learning Rate, Batch Size, Momentum, and Weight Decay." ArXiv:1803.09820 [Cs, Stat], April 24, 2018. http://arxiv.org/abs/1803.09820.

[34] Rajput, V. (2021). Robustness of different loss functions and their impact on networks learning capability. https://doi.org/10.48550/arXiv.2110.08322

ANEXO I. PRESUPUESTO

En este anexo, se lleva a cabo el cálculo económico del proyecto realizado.

Se va a realizar un presupuesto basado en el coste de la mano de obra, tanto del ingeniero estudiante de máster, como del tutor del proyecto. También se va a incluir el coste del hardware y del software utilizado para la realización de este, teniendo en cuenta la amortización del precio según el uso dado.

PARTIDA DE MANO DE OBRA				
Nombre	*Unidades*	*Cantidad*	*Coste Unitario (€)*	*Importe (€)*
Alumno de Máster de Ingeniería Industrial. Encargado del proyecto	*h*	*300*	*25*	*7.500*
Tutor del Trabajo de Fin de Grado	*h*	*25*	*40*	*1.000*
			Importe Total Mano de Obra:	***8.500***

PARTIDA DE HARDWARE					
Nombre	*Precio (€)*	*Vida Útil (meses)*	*Coste Amortizado (€/mes)*	*Tiempo de uso (meses)*	*Importe (€)*
Ordenador de sobremesa (Incluye todos los componentes y sistema operativo)	*1.500*	*60*	*25*	*4*	*100*
				Importe Total Hardware:	***100***

PARTIDA DE SOFTWARE				
Todo el software utilizado ha sido de código abierto o bien se han utilizado versiones gratuitas				
Nombre	*Unidades*	*Cantidad*	*Coste Unitario (€)*	*Importe (€)*
Python	*Ud.*	*1*	*0 (Software Libre)*	*0*
Pytorch	*Ud.*	*1*	*0 (Software Libre)*	*0*
Tierpsy Tracker	*Ud.*	*1*	*0 (Software Libre)*	*0*
Kaggle	*Ud.*	*1*	*0 (Software Libre)*	*0*
Google Colab	*Ud.*	*1*	*0 (Versión gratuita)*	*0*
			Importe Total Software:	***0***

PRESUPUESTO TOTAL	
Concepto	*Importe (€)*
PRESUPUESTO DE EJECUCIÓN MATERIAL	***8.600***
12% Gastos Generales	*1.032*
6% Beneficio Industrial	*516*
PRESUPUESTO DE EJECUCIÓN POR CONTRATA	***10.148***
21% IVA	*2.131,08*
PRESUPUESTO TOTAL	***12.279,08***

Con esto, el presupuesto total del proyecto asciende a un valor de **DOCE MIL DOSCIENTOS SETENTA Y NUEVE EUROS CON OCHO CÉNTIMOS (*12.279,08 €*)**

ANEXO II. CÓDIGO FUENTE

Se adjunta todo el código fuente creado para el Trabajo de Fin de Máster. Para mejor visualización se recomienda copiar el código y pegarlo en un IDE cualquiera.

1. Obtención de esqueletos a partir de coordenadas HDF5

```
import h5py
import numpy as np
import matplotlib.pyplot as plt

def read_coordinates_from_hdf5(hdf5_filename):
    # Leer el archivo HDF5
    with h5py.File(hdf5_filename, 'r') as hdf5_file:
        # Obtener un vector de coordenadas de todos los frames
        xy_coordinates = hdf5_file['/coordinates/skeletons'][:, :]
        print(xy_coordinates[0])
        return xy_coordinates

def draw_coordinates(xy_coordinates):
    # Crear la figura y el lienzo para dibujar
    fig, ax = plt.subplots()

    # Dibujar las coordenadas X
    ax.plot(xy_coordinates[:, 0], marker='o', linestyle='-')

    # Configurar los ejes
    ax.set_aspect('equal', adjustable='box')
    ax.set_xlabel('Índice del punto')
    ax.set_ylabel('Coordenada X')
    ax.set_title('Coordenadas X del frame 0')
    plt.show()


if __name__ == "__main__":
    dataset_dir =
"C:/Users/alexb/OneDrive/Escritorio/Master/TFM/Dataset/files_feat_data/files_feat_data/N2_tr
ain"
    hdf5_filename = dataset_dir + "/N2 1 on food R_2011_06_01__12_21_58___1___2.hdf5"
    output_image_filename = "coordenadas.png"

    # Leer las coordenadas X e Y del hdf5
    xy_coordinates = read_coordinates_from_hdf5(hdf5_filename)

    for frame in range(len(xy_coordinates)):
        # Dibujar las coordenadas
        draw_coordinates(xy_coordinates[frame])
```

2. Segmentación manual *C. elegans*

```
import cv2
import numpy as np
import time


def segment_object(video_path, threshold_value):
    cap = cv2.VideoCapture(video_path)

    while cap.isOpened():
        ret, frame = cap.read()

        if not ret:
            break

        # Convertir a escala de grises y aplicar Threshold
        gray_frame = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
        _, thresholded_frame = cv2.threshold(gray_frame, threshold_value, 255,
cv2.THRESH_BINARY)

        # Operaciones de opening y closing
        kernel = np.ones((7, 7), np.uint8)
        opening = cv2.morphologyEx(thresholded_frame, cv2.MORPH_OPEN, kernel)
        closing = cv2.morphologyEx(opening, cv2.MORPH_CLOSE, kernel)

        # Buscar contornos
        contours, hierarchy = cv2.findContours(closing, cv2.RETR_TREE,
cv2.CHAIN_APPROX_SIMPLE)

        # Inicializar máscaras
        mask_green = np.zeros_like(gray_frame)
        mask_blue = np.zeros_like(gray_frame)

        for i, contour in enumerate(contours):
            color = (0, 0, 255)  # Color rojo para contorno exterior (nivel 0)

            if hierarchy[0][i][3] != -1:
                # Area del contorno exterior
                area = cv2.contourArea(contour)

                if area < 500:
                    continue  # Ignorar contornos externos con area menor a X
pixeles. Así se descartan zonas con comida que aparecen en el video

                if hierarchy[0][hierarchy[0][i][3]][3] != -1:
                    color = (255, 0, 0)  # Color azul para contorno interior
(nivel 2)
                    cv2.drawContours(mask_blue, [contour], -1, 1,
thickness=cv2.FILLED)  # Rellenar la máscara azul
                else:
                    color = (0, 255, 0)  # Color verde para contorno intermedio
(nivel 1)
                    cv2.drawContours(mask_green, [contour], -1, 1,
thickness=cv2.FILLED)  # Rellenar la máscara verde
```

```
        # Restar la máscara azul de la máscara verde
        mask_final = mask_green - mask_blue

        # Erosionar la máscara para quitar el pixel de contorno de placa petri
del c elegans
        kernel_erosion = np.ones((3, 3), np.uint8)
        mask_final = cv2.erode(mask_final, kernel_erosion, iterations=1)
        segmented_frame = cv2.bitwise_and(frame, frame, mask=mask_final)

        mask_final = mask_final*255
        mask_green = mask_green*255
        mask_blue = mask_blue*255

        # Visualizar las diferentes imagenes, antes y después de segmentar
        combined = np.hstack((frame, cv2.cvtColor(mask_final,
cv2.COLOR_GRAY2BGR), segmented_frame))
        combined1 = np.hstack((frame, cv2.cvtColor(mask_final,
cv2.COLOR_GRAY2BGR)))
        combined2 = np.hstack((cv2.cvtColor(mask_green, cv2.COLOR_GRAY2BGR),
cv2.cvtColor(mask_blue, cv2.COLOR_GRAY2BGR)))
        combined3 = np.vstack((combined1, combined2))
        cv2.imshow('Imagen antes y despues de segmentar', combined)

        if cv2.waitKey(1) & 0xFF == ord('q'):
            break

    cap.release()
    cv2.destroyAllWindows()

if __name__ == "__main__":
    folder_path =
'C:/Users/alexb/OneDrive/Escritorio/Master/TFM/Dataset/datasetN2vsUnc/datasetN2vs
Unc/N2_train/'
    file_path = 'N2 1 on food R_2011_06_01__12_21_58___1___2fps_30nf_26998.avi'
    video_path = folder_path + file_path
    threshold_value = 70 # Este valor hay que cambiarlo depende el video del
experimento, para mejores resultados (entre 80-110)
    segment_object(video_path, threshold_value)
```

3. Creación de datasets a partir de vídeos

Se reutiliza el código de segmentación previo y se añaden funcionalidades para seleccionar el valor de *threshold* para cada vídeo antes de segmentar.

```
import os
import cv2
import numpy as np


def create_dataset_folders(dataset_path):
    # Crear las carpetas necesarias para el dataset
    train_folder = os.path.join(dataset_path, 'train')

    train_images_folder = os.path.join(train_folder, 'images')
    train_masks_folder = os.path.join(train_folder, 'masks')


    os.makedirs(train_images_folder, exist_ok=True)
    os.makedirs(train_masks_folder, exist_ok=True)

    return train_images_folder, train_masks_folder


def segment_object(video_path, threshold_value, i_video, train_images_folder,
train_masks_folder, save_interval=20):

    cap = cv2.VideoCapture(video_path)

    current_frame = 0

    while cap.isOpened():
        ret, frame = cap.read()

        if not ret:
            break

        if current_frame % save_interval == 0:
            gray_frame = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)

            _, thresholded_frame = cv2.threshold(gray_frame, threshold_value,
255, cv2.THRESH_BINARY)

            kernel = np.ones((7, 7), np.uint8)
            opening = cv2.morphologyEx(thresholded_frame, cv2.MORPH_OPEN, kernel)
            closing = cv2.morphologyEx(opening, cv2.MORPH_CLOSE, kernel)

            contours, hierarchy = cv2.findContours(closing, cv2.RETR_TREE,
cv2.CHAIN_APPROX_SIMPLE)

            mask_green = np.zeros_like(gray_frame)
            mask_blue = np.zeros_like(gray_frame)

            for i, contour in enumerate(contours):
```

```
                color = (0, 0, 255)  # Color rojo para contorno exterior (nivel
0)

                if hierarchy[0][i][3] != -1:
                    # Calculate the area of the external contour
                    area = cv2.contourArea(contour)

                    if area < 500:
                        continue  # Ignorar contornos externos con area menor a X
pixeles.

                    if hierarchy[0][hierarchy[0][i][3]][3] != -1:
                        color = (255, 0, 0)  # Color azul para contorno interior
(nivel 2)
                        cv2.drawContours(mask_blue, [contour], -1, 1,
thickness=cv2.FILLED)  # Rellenar la máscara azul
                    else:
                        color = (0, 255, 0)  # Color verde para contorno
intermedio (nivel 1)
                        cv2.drawContours(mask_green, [contour], -1, 1,
thickness=cv2.FILLED)  # Rellenar la máscara verde

            # Restar la máscara azul de la máscara verde
            mask_final = mask_green - mask_blue

            # Erosionar la máscara para quitar el pixel de contorno de placa
petri del c elegans
            kernel_erosion = np.ones((3, 3), np.uint8)
            mask_final = cv2.erode(mask_final, kernel_erosion, iterations=1)

            mask_final = mask_final * 255

            image_name = f"{i_video}_{current_frame:05d}.jpg"
            mask_name = f"{i_video}_{current_frame:05d}.jpg"
            cv2.imwrite(os.path.join(train_images_folder, image_name), frame)
            cv2.imwrite(os.path.join(train_masks_folder, mask_name), mask_final)

        current_frame += 1

    cap.release()
    cv2.destroyAllWindows()




def view_frames(video_path, threshold_value):
    cap = cv2.VideoCapture(video_path)

    cv2.namedWindow("Ajuste de Threshold")

    def update_threshold(val):
        nonlocal threshold_value
        threshold_value = val
```

```
    cv2.createTrackbar("Threshold", "Ajuste de Threshold", threshold_value, 255,
update_threshold)

    while cap.isOpened():
        ret, frame = cap.read()

        if not ret:
            break

        gray_frame = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
        _, thresholded_frame = cv2.threshold(gray_frame, threshold_value, 255,
cv2.THRESH_BINARY)

        kernel = np.ones((7, 7), np.uint8)
        opening = cv2.morphologyEx(thresholded_frame, cv2.MORPH_OPEN, kernel)
        closing = cv2.morphologyEx(opening, cv2.MORPH_CLOSE, kernel)

        contours, hierarchy = cv2.findContours(closing, cv2.RETR_TREE,
cv2.CHAIN_APPROX_SIMPLE)

        mask_green = np.zeros_like(gray_frame)
        mask_blue = np.zeros_like(gray_frame)

        for i, contour in enumerate(contours):
            color = (0, 0, 255)  # Color rojo para contorno exterior (nivel 0)

            if hierarchy[0][i][3] != -1:
                area = cv2.contourArea(contour)

                if area < 500:
                    continue  # Ignorar contornos externos con area menor a X
pixeles. Así se descartan zonas con comida que aparecen en el video

                if hierarchy[0][hierarchy[0][i][3]][3] != -1:
                    color = (255, 0, 0)  # Color azul para contorno interior
(nivel 2)
                    cv2.drawContours(mask_blue, [contour], -1, 1,
thickness=cv2.FILLED)  # Rellenar la máscara azul
                else:
                    color = (0, 255, 0)  # Color verde para contorno intermedio
(nivel 1)
                    cv2.drawContours(mask_green, [contour], -1, 1,
thickness=cv2.FILLED)  # Rellenar la máscara verde

        # Restar la máscara azul de la máscara verde
        mask_final = mask_green - mask_blue

        # Erosionar la máscara para quitar el pixel de contorno de placa petri
del c elegans
        kernel_erosion = np.ones((3, 3), np.uint8)
        mask_final = cv2.erode(mask_final, kernel_erosion, iterations=1)

        mask_final = mask_final * 255
```

```
        segmented_frame = cv2.bitwise_and(frame, frame, mask=mask_final)


        combined_2 = np.hstack((segmented_frame, cv2.cvtColor(mask_final,
cv2.COLOR_GRAY2BGR)))
        cv2.imshow('Ajuste de Threshold', combined_2)

        key = cv2.waitKey(1) & 0xFF
        if key == ord('q'):
            break

    cap.release()
    cv2.destroyAllWindows()
    return threshold_value


if __name__ == "__main__":
    train_folder_path =
'C:/Users/alexb/OneDrive/Escritorio/Master/TFM/neural_net/umf3/videos_diferentes/
test'

    train_videos = os.listdir(train_folder_path)

    # Ruta del dataset a crear
    dataset_path = 'dataset_diferentes/test'
    train_images_folder, train_masks_folder =
create_dataset_folders(dataset_path)

    i_train = 1


    for video_file in train_videos:
        video_path = os.path.join(train_folder_path, video_file)
        threshold = 70  # Threshold inicial

        # Ventana para ajuste de threshold previo a la segmentación de cada vídeo
        threshold_value = view_frames(video_path, threshold)
        segment_object(video_path, threshold_value, i_train, train_images_folder,
train_masks_folder, save_interval=20)
        print(f"finalizado el video {video_file}")
        i_train += 1

    print("Proceso completado")
```

4. Limpieza de los datasets generados

Se limpian los datasets para eliminar fotogramas mal segmentados.

```
import os
from PIL import Image

# Paths a las carpetas de imagenes y mascaras
mask_folder =
'c:/Users/alexb/OneDrive/Escritorio/Master/TFM/neural_net/umf3/dataset_diferentes
/test/masks'
image_folder =
'C:/Users/alexb/OneDrive/Escritorio/Master/TFM/neural_net/umf3/dataset_diferentes
/test/images'

mask_files = [f for f in os.listdir(mask_folder) if
os.path.isfile(os.path.join(mask_folder, f))]

for mask_file in mask_files:
    mask_path = os.path.join(mask_folder, mask_file)

    mask = Image.open(mask_path)

    # Convertir a escala de grises
    mask_gray = mask.convert('L')

    # Se comprueba si la máscara es completamente negra. No existe C. elegans
segmentado
    if not mask_gray.getextrema()[1]:  # getextrema() devuelve máximos y mínimos
valores de píxeles en la imagen.
        # Si el máximo es 0, la máscara es toda negra y se elimina
        print(f"Eliminando máscara vacía: {mask_path}")
        os.remove(mask_path)

        # Se elimina la imagen correspondiente
        image_path = os.path.join(image_folder, mask_file)
        if os.path.exists(image_path):
            print(f"Eliminando la imagen correspondiente a la máscara:
{image_path}")
            os.remove(image_path)
        else:
            print(f"Error. No se ha encontrado imagen relacionada con la máscara
{mask_file}")

print("Proceso completado")
```

5. DataLoaders, modelos U-Net, entrenamiento, validación, test e inferencia

Aquí se adjunta el grueso del código utilizado para generar los *DataLoaders* en *Python*, los modelos de U-Net, la fase de entrenamiento, validación, test y la inferencia de imágenes. En este caso el código tiene formato de *Jupyter Notebook*.

```
# %% [code]
import numpy as np
import matplotlib.pyplot as plt
import os
import random
import torch
from torch import nn, optim
from torchvision.transforms import functional as F
import torch.nn.functional as Fu
from torchvision import transforms as T
from torch.utils.data import DataLoader, Dataset, random_split, Subset
import PIL
from PIL import Image, ImageEnhance

# %% [code]
PATH = '/kaggle/input/dataset-v3/dataset_v2'
TRAIN_PATH = '/kaggle/input/dataset-
diferentes/dataset_diferentes/dataset_diferentes/test/images'
TRAIN_MASKS_PATH = '/kaggle/input/dataset-
diferentes/dataset_diferentes/dataset_diferentes/test/masks'
VAL_PATH = '/kaggle/input/dataset-
diferentes/dataset_diferentes/dataset_diferentes/train/images'
VAL_MASKS_PATH = '/kaggle/input/dataset-
diferentes/dataset_diferentes/dataset_diferentes/train/masks'
TEST_PATH = '/kaggle/input/dataset-
diferentes/dataset_diferentes/dataset_diferentes/val/images'
TEST_MASKS_PATH = '/kaggle/input/dataset-
diferentes/dataset_diferentes/dataset_diferentes/val/masks'

# %% [markdown]
# # DATASET + DATALOADERS + DATA AUGMENTATION

# %% [code]
class Celegan_Dataset(Dataset):
    def __init__(self, images, masks=None, data_augmentation=False):
        self.train_images = images
        self.train_masks = masks
        self.data_augmentation = data_augmentation

        # Importante ordenar, para que los pares imagen-mascara se seleccionen
correctamente (aunque en este caso se llaman igual)
        self.images = sorted(os.listdir(self.train_images))
        self.masks = sorted(os.listdir(self.train_masks))

    def transform(self, image, mask):
        # Random vertical flipping
        if random.random() > 0.75:
            image = F.vflip(image)
```

```
            mask = F.vflip(mask)

        # Random horizontal flipping
        if random.random() > 0.75:
            image = F.hflip(image)
            mask = F.hflip(mask)

        # Random rotation
        if random.random() > 0.75:
            angle = random.uniform(-20, 20)
            image = F.rotate(image, angle)
            mask = F.rotate(mask, angle)

        # Random brightness change (luminance)
        if random.random() > 0.8:
            enhancer = ImageEnhance.Brightness(image)
            brightness_factor = random.uniform(0.7, 1.3)  # generate random
brightness factor
            image = enhancer.enhance(brightness_factor)

        # Center crop
        if random.random() > 0.8:
            var = int(random.random() * 100)
            image = F.center_crop(image, (300 + var, 300 + var))
            mask = F.center_crop(mask, (300 + var, 300 + var))

        # Resize
        image = F.resize(image, (480, 640))
        mask = F.resize(mask, (480, 640))

        # Convert to tensor
        image = F.to_tensor(image)
        mask = F.to_tensor(mask)

        return image, mask

    def __len__(self):
        if self.train_masks is not None:
            assert len(self.images)==len(self.masks), 'not the same number of
images and masks'
        return len(self.images)

    def __getitem__(self, idx):
        image_name = os.path.join(self.train_images, self.images[idx])
        img = Image.open(image_name)
        mask_name = os.path.join(self.train_masks, self.masks[idx])
        mask = Image.open(mask_name)

        # default transform, if no other declared
        trans = T.Compose([
                T.Resize([480, 640]),
                T.ToTensor()] )
```

```
        if self.data_augmentation:
            img, mask = self.transform(img, mask)
        else:
            img =trans(img)
            mask = trans(mask)

        #Normalizamos la máscara entre 0 y 1
        mask_max = mask.max().item()
        mask /= mask_max

        return img, mask

# %% [markdown]
# ## Creamos los datasets y seleccionamos el tamaño

# %% [code]
# Generar los datasets
full_train_dataset = Celegan_Dataset(TRAIN_PATH, TRAIN_MASKS_PATH,
data_augmentation=True)
full_val_dataset = Celegan_Dataset(VAL_PATH, VAL_MASKS_PATH,
data_augmentation=False)
full_test_dataset = Celegan_Dataset(TEST_PATH, TEST_MASKS_PATH,
data_augmentation=False)
print(len(full_train_dataset), len(full_val_dataset), len(full_test_dataset))

# %% [code]
# Numero de imagenes de cada dataset a utilizar
num_train_images = 1270
num_val_images = 500
num_test_images = 500

train_dataset = Subset(full_train_dataset, indices=range(num_train_images))
val_dataset = Subset(full_val_dataset, indices=range(num_val_images))
test_dataset = Subset(full_test_dataset, indices=range(num_test_images))
print(len(train_dataset), len(val_dataset), len(test_dataset))

# %% [code]
# Crear los dataloaders para train, val y test
BATCH_SIZE = 2
train_loader = DataLoader(train_dataset, batch_size=BATCH_SIZE, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=BATCH_SIZE, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=BATCH_SIZE, shuffle=True)

# %% [code]
# Comprobamos tamaños del par imagen-mascara
imgs, masks = next(iter(train_loader))
print("train:   ", imgs.shape, masks.shape)

imgs, masks = next(iter(val_loader))
print("val:     ", imgs.shape, masks.shape)

imgs, masks = next(iter(test_loader))
print("test:    ", imgs.shape, masks.shape)
```

```
# %% [markdown]
# ## VISUALIZAMOS DATASET

# %% [code]
def plot_mini_batch(imgs, masks=None, alpha=False):
    plt.figure(figsize=(17,6))
    for i in range(BATCH_SIZE):
        plt.subplot(2, 6, i+1)
        img=imgs[i,...].permute(1,2,0).numpy()

        if masks is not None:
            mask = masks[i, ...].permute(1,2,0).numpy()

        plt.imshow(img)

        if alpha:
            plt.imshow(mask, alpha=0.5)  # Superponer la máscara por encima

        plt.axis('Off')
    plt.tight_layout()
    plt.show()


def plot_heatmaps(ground_truth, predictions, alpha=0.5):
    plt.figure(figsize=(17, 4))

    for i in range(len(ground_truth)):
        # Plot ground truth
        plt.subplot(2, len(ground_truth), i+1)
        gt = ground_truth[i, ...].squeeze().cpu().numpy()
        plt.imshow(gt, cmap='gray', interpolation='nearest')
        plt.title(f'Ground Truth {i+1}')
        plt.axis('off')

        # Plot predictions
        plt.subplot(2, len(predictions), len(ground_truth) + i + 1)
        pred = predictions[i, ...].squeeze().cpu().detach().numpy()
        plt.imshow(pred, cmap='hot', interpolation='nearest')
        plt.title(f'Prediction {i+1}')
        plt.axis('off')

    plt.tight_layout()
    plt.show()


def plot_heatmaps_big(ground_truth, predictions, alpha=0.5):
    plt.figure(figsize=(20, 8))

    for i in range(4):
        # Plot ground truth
        plt.subplot(2, 4, i+1)
        gt = ground_truth[i, ...].squeeze().cpu().numpy()
        plt.imshow(gt, cmap='gray', interpolation='nearest')
```

```
        plt.title(f'Ground Truth {i+1}')
        plt.axis('off')

        # Plot predictions
        plt.subplot(2, 4, 4 + i + 1)
        pred = predictions[i, ...].squeeze().cpu().detach().numpy()
        plt.imshow(pred, cmap='hot', interpolation='nearest')
        plt.title(f'Prediction {i+1}')
        plt.axis('off')

    plt.tight_layout()
    plt.show()

# %% [code]
# Plotear un minibatch de cada dataset para comprobar que funciona bien
print("Train images")
imgs, masks = next(iter(train_loader))
plot_mini_batch(imgs, masks, alpha=True)

print("\n\n\nValidation images")
imgs, masks = next(iter(val_loader))
plot_mini_batch(imgs, masks, alpha=True)

print("\n\n\nTest images")
imgs, masks = next(iter(test_loader))
plot_mini_batch(imgs, masks, alpha=True)

# %% [markdown]
# # MODELO U-NET

# %% [code]
class Conv_3_k(nn.Module):
    def __init__(self, channels_in, channels_out):
        super().__init__()
        self.conv1 = nn.Conv2d(channels_in, channels_out, kernel_size=3,
stride=1, padding=1)
    def forward(self, x):
        return self.conv1(x)

class Double_Conv(nn.Module):
    '''
    Double convolution block for U-Net
    '''
    def __init__(self, channels_in, channels_out):
        super().__init__()
        self.double_conv = nn.Sequential(
                           Conv_3_k(channels_in, channels_out),
                           nn.BatchNorm2d(channels_out),
                           nn.ReLU(),

                           Conv_3_k(channels_out, channels_out),
                           nn.BatchNorm2d(channels_out),
                           nn.ReLU(),
                            )
```

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

class Down_Conv(nn.Module):
    '''
    Down convolution part
    '''
    def __init__(self, channels_in, channels_out):
        super().__init__()
        self.encoder = nn.Sequential(
                        nn.MaxPool2d(2,2),
                        Double_Conv(channels_in, channels_out)
                        )
    def forward(self, x):
        return self.encoder(x)

class Up_Conv(nn.Module):
    '''
    Up convolution part
    '''
    def __init__(self,channels_in, channels_out):
        super().__init__()
        self.upsample_layer = nn.Sequential(
                        nn.Upsample(scale_factor=2, mode='bicubic'),
                        nn.Conv2d(channels_in, channels_in//2, kernel_size=1,
stride=1)
                        )
        self.decoder = Double_Conv(channels_in, channels_out)

    def forward(self, x1, x2):
        '''
        x1 - upsampled volume
        x2 - volume from down sample to concatenate
        '''
        x1 = self.upsample_layer(x1)
        x = torch.cat([x2, x1],dim=1)
        return self.decoder(x)

class UNET(nn.Module):
    '''
    UNET model
    '''
    def __init__(self, channels_in, channels, num_classes):
        super().__init__()
        self.first_conv = Double_Conv(channels_in, channels) #64, 224, 224
        self.down_conv1 = Down_Conv(channels, 2*channels) # 128, 112, 112
        self.down_conv2 = Down_Conv(2*channels, 4*channels) # 256, 56, 56
        self.down_conv3 = Down_Conv(4*channels, 8*channels) # 512, 28, 28

        self.middle_conv = Down_Conv(8*channels, 16*channels) # 1024, 14, 14

        self.up_conv1 = Up_Conv(16*channels, 8*channels)
        self.up_conv2 = Up_Conv(8*channels, 4*channels)
        self.up_conv3 = Up_Conv(4*channels, 2*channels)
```

```
        self.up_conv4 = Up_Conv(2*channels, channels)

        self.last_conv = nn.Conv2d(channels, num_classes, kernel_size=1,
stride=1)

    def forward(self, x):
        x1 = self.first_conv(x)
        x2 = self.down_conv1(x1)
        x3 = self.down_conv2(x2)
        x4 = self.down_conv3(x3)

        x5 = self.middle_conv(x4)

        u1 = self.up_conv1(x5, x4)
        u2 = self.up_conv2(u1, x3)
        u3 = self.up_conv3(u2, x2)
        u4 = self.up_conv4(u3, x1)

        '''ACABA CON UNA ACTIVACIÓN SIGMOIDE PARA QUE EL OUPUT SEAN DATOS ENTRE 0
Y 1'''
        return Fu.sigmoid(self.last_conv(u4))

# %% [markdown]
# # Modelo UNET++

# %% [code]
class NestedUNet(nn.Module):

    def __init__(self, num_classes, input_channels=3, deep_supervision=False,
**kwargs):
        super().__init__()

        nb_filter = [32, 64, 128, 256, 512]

        self.deep_supervision = deep_supervision

        self.pool = nn.MaxPool2d(2, 2)
        self.up = nn.Upsample(scale_factor=2, mode='bilinear',
align_corners=True)

        self.conv0_0 = VGGBlock(input_channels, nb_filter[0], nb_filter[0])
        self.conv1_0 = VGGBlock(nb_filter[0], nb_filter[1], nb_filter[1])
        self.conv2_0 = VGGBlock(nb_filter[1], nb_filter[2], nb_filter[2])
        self.conv3_0 = VGGBlock(nb_filter[2], nb_filter[3], nb_filter[3])
        self.conv4_0 = VGGBlock(nb_filter[3], nb_filter[4], nb_filter[4])

        self.conv0_1 = VGGBlock(nb_filter[0]+nb_filter[1], nb_filter[0],
nb_filter[0])
        self.conv1_1 = VGGBlock(nb_filter[1]+nb_filter[2], nb_filter[1],
nb_filter[1])
        self.conv2_1 = VGGBlock(nb_filter[2]+nb_filter[3], nb_filter[2],
nb_filter[2])
        self.conv3_1 = VGGBlock(nb_filter[3]+nb_filter[4], nb_filter[3],
nb_filter[3])
```

```
        self.conv0_2 = VGGBlock(nb_filter[0]*2+nb_filter[1], nb_filter[0],
nb_filter[0])
        self.conv1_2 = VGGBlock(nb_filter[1]*2+nb_filter[2], nb_filter[1],
nb_filter[1])
        self.conv2_2 = VGGBlock(nb_filter[2]*2+nb_filter[3], nb_filter[2],
nb_filter[2])

        self.conv0_3 = VGGBlock(nb_filter[0]*3+nb_filter[1], nb_filter[0],
nb_filter[0])
        self.conv1_3 = VGGBlock(nb_filter[1]*3+nb_filter[2], nb_filter[1],
nb_filter[1])

        self.conv0_4 = VGGBlock(nb_filter[0]*4+nb_filter[1], nb_filter[0],
nb_filter[0])

        if self.deep_supervision:
            self.final1 = nn.Conv2d(nb_filter[0], num_classes, kernel_size=1)
            self.final2 = nn.Conv2d(nb_filter[0], num_classes, kernel_size=1)
            self.final3 = nn.Conv2d(nb_filter[0], num_classes, kernel_size=1)
            self.final4 = nn.Conv2d(nb_filter[0], num_classes, kernel_size=1)
        else:
            self.final = nn.Conv2d(nb_filter[0], num_classes, kernel_size=1)

    def forward(self, input):
        x0_0 = self.conv0_0(input)
        x1_0 = self.conv1_0(self.pool(x0_0))
        x0_1 = self.conv0_1(torch.cat([x0_0, self.up(x1_0)], 1))

        x2_0 = self.conv2_0(self.pool(x1_0))
        x1_1 = self.conv1_1(torch.cat([x1_0, self.up(x2_0)], 1))
        x0_2 = self.conv0_2(torch.cat([x0_0, x0_1, self.up(x1_1)], 1))

        x3_0 = self.conv3_0(self.pool(x2_0))
        x2_1 = self.conv2_1(torch.cat([x2_0, self.up(x3_0)], 1))
        x1_2 = self.conv1_2(torch.cat([x1_0, x1_1, self.up(x2_1)], 1))
        x0_3 = self.conv0_3(torch.cat([x0_0, x0_1, x0_2, self.up(x1_2)], 1))

        x4_0 = self.conv4_0(self.pool(x3_0))
        x3_1 = self.conv3_1(torch.cat([x3_0, self.up(x4_0)], 1))
        x2_2 = self.conv2_2(torch.cat([x2_0, x2_1, self.up(x3_1)], 1))
        x1_3 = self.conv1_3(torch.cat([x1_0, x1_1, x1_2, self.up(x2_2)], 1))
        x0_4 = self.conv0_4(torch.cat([x0_0, x0_1, x0_2, x0_3, self.up(x1_3)],
1))

        if self.deep_supervision:
            output1 = self.final1(x0_1)
            output2 = self.final2(x0_2)
            output3 = self.final3(x0_3)
            output4 = self.final4(x0_4)
            return [output1, output2, output3, output4]

        else:
            output = self.final(x0_4)
```

```
            return Fu.sigmoid(output)


class VGGBlock(nn.Module):

    def __init__(self, in_channels, middle_channels, out_channels):
        super().__init__()
        self.relu = nn.ReLU(inplace=True)
        self.conv1 = nn.Conv2d(in_channels, middle_channels, 3, padding=1)
        self.bn1 = nn.BatchNorm2d(middle_channels)
        self.conv2 = nn.Conv2d(middle_channels, out_channels, 3, padding=1)
        self.bn2 = nn.BatchNorm2d(out_channels)
        self.SE = Squeeze_Excite(out_channels,8)

    def forward(self,x):
        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)

        out = self.conv2(out)
        out = self.bn2(out)
        out = self.relu(out)
        out = self.SE(out)

        return(out)


class Squeeze_Excite(nn.Module):

    def __init__(self,channel,reduction):
        super().__init__()
        self.avgpool = nn.AdaptiveAvgPool2d(1)
        self.fc = nn.Sequential(
            nn.Linear(channel, channel // reduction, bias=False),
            nn.ReLU(inplace=True),
            nn.Linear(channel // reduction, channel, bias=False),
            nn.Sigmoid()
        )

    def forward(self,x):
        b, c, _, _ = x.size()
        y = self.avgpool(x).view(b, c)
        y = self.fc(y).view(b, c, 1, 1)
        return x * y.expand_as(x)


# %% [markdown]
# ## Testeamos el modelo para ver si hay errores y ver que las dimensiones de
salida son las esperadas

# %% [code]
def test():
    x = torch.randn((12, 3, 224, 224))
    model = UNET(3, 64, 1)
```

```
    #model = NestedUNet(1,3)
    return model(x)

preds = test()
print(preds.shape)

# %% [markdown]
# # FASE DE ENTRENAMIENTO

# %% [markdown]
# ## BÚSQUEDA DE LR ÓPTIMO

# %% [code]
def train_batch(batch, model, optimizer, lr=None):
    imgs, masks = batch
    imgs, masks = imgs.to(device), masks.to(device)

    # Set learning rate if specified
    if lr is not None:
        for param_group in optimizer.param_groups:
            param_group['lr'] = lr

    # Zero the parameter gradients
    optimizer.zero_grad()

    # Forward pass
    outputs = model(imgs)

    # Compute loss
    loss = dice_loss(outputs, masks)

    acc = calculate_accuracy(outputs, masks)
    iou = calculate_iou(outputs, masks)
    dice = calculate_dice(outputs, masks)

    # Backward pass and optimization
    loss.backward()
    optimizer.step()

    print(f'lr: {lr}, Loss: {loss.item():.4f}, Acc: {acc:.4f}, IoU: {iou:.4f},
Dice: {dice:.4f}')

    return loss

# %% [code]
device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')
print(device)

model = UNET(3, 64, 1).to(device) # Adjust the parameters as per your needs
optimizer = optim.Adam(model.parameters(), lr=1e-7)  # Initialize with a very low
learning rate

lr_finder = []  # to store the loss and learning rate
```

```
lr_mult = (1 / 1e-6) ** (1/len(train_loader))  # we will increase the learning
rate at each step
lr = 1e-6

for batch in train_loader:
    loss = train_batch(batch, model, optimizer, lr)  # assumes this function
performs a training pass and returns the loss
    lr_finder.append((lr, loss.item()))
    lr *= lr_mult
    for param_group in optimizer.param_groups:
        param_group['lr'] = lr




lrs, losses = zip(*lr_finder)
plt.figure(figsize=(18, 6))
plt.plot(lrs, losses)
plt.xscale('log')
plt.xlabel('Learning rate')
plt.ylabel('Loss')
plt.show()

# %% [markdown]
# ## Definición de hiperparámetros

# %% [code]
# Hyperparameters
learning_rate = 0.001
epochs = 10

device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')
print(device)

#model = UNET_noBN(3, 64, 1).to(device) #3 inputs, 1 output class, 64 starting
channels
#model = UNET(3, 64, 1).to(device) #3 inputs, 1 output class, 64 starting
channels
#model = UNet33(3, 1, bilinear=True).to(device)
model = NestedUNet(1,3).to(device)

#optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
optimizer = optim.Adam(model.parameters(), lr=learning_rate)

# %% [markdown]
# ## LOSS FUNCTION Y MÉTRICAS DE RENDIMIENTO

# %% [code]
def dice_loss(pred, target, smooth=1.):
    intersection = (pred * target).sum(dim=2).sum(dim=2)

    dice = (2. * intersection + smooth) / (pred.sum(dim=2).sum(dim=2) +
target.sum(dim=2).sum(dim=2) + smooth)
```

```
    return 1 - dice.mean()


def combined_loss(pred, target, smooth=1.):
    # BCE Loss
    bce_loss = Fu.binary_cross_entropy(pred, target)

    # Dice Loss
    intersection = (pred * target).sum(dim=2).sum(dim=2)
    dice_coeff = (2. * intersection + smooth) / (pred.sum(dim=2).sum(dim=2) +
target.sum(dim=2).sum(dim=2) + smooth)
    dice_loss = 1 - dice_coeff.mean()

    # Combined Loss = BCE + Dice Loss
    combined = bce_loss + dice_loss

    return combined


def calculate_accuracy(pred, target, threshold=0.5):
    pred = pred > threshold
    target = target > threshold
    correct = (pred == target).sum().item()
    total = pred.numel()
    return correct / total

def calculate_iou(pred, target, threshold=0.5):
    pred = pred > threshold
    target = target > threshold
    intersection = (pred & target).sum().item()
    union = (pred | target).sum().item()
    return intersection / union

def calculate_dice(pred, target, smooth=1e-6, threshold=0.5):
    pred = pred > threshold
    target = target > threshold
    intersection = (pred & target).sum().item()
    return (2. * intersection + smooth) / (pred.sum().item() +
target.sum().item() + smooth)

# %% [markdown]
# ## TRAINING LOOP

# %% [code]
from torch.optim.lr_scheduler import OneCycleLR

epochs = 10
max_lr = 0.004
min_lr = 0.001
steps_per_epoch = len(train_loader)

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

```
model = UNET(3, 64,1).to(device)

optimizer = optim.Adam(model.parameters(), lr=max_lr)

# Initialize 1cycle scheduler
scheduler = OneCycleLR(optimizer, max_lr=max_lr, epochs=epochs,
steps_per_epoch=steps_per_epoch,
                       anneal_strategy='linear', div_factor=(max_lr / min_lr),
final_div_factor=1e4)


import csv

def train_model_scheduler(model, optimizer, scheduler, epochs,
csv_file_name="/kaggle/working/CSV_UNET_Lr004_AdamOptim_1cycle_CombinedLoss_DataA
ug.csv"):
    train_loss_list = []
    val_loss_list = []
    lr_list = []

    # Open CSV file for writing
    with open(csv_file_name, 'w', newline='') as csvfile:
        csvwriter = csv.writer(csvfile)

        # Write the header
        csvwriter.writerow(['Epoch', 'Step', 'Phase', 'Loss', 'Accuracy', 'IoU',
'Dice'])

        for epoch in range(epochs):
            # Training loop
            model.train()
            train_loss = 0.0
            for i, (imgs, masks) in enumerate(train_loader):
                imgs, masks = imgs.to(device), masks.to(device)

                optimizer.zero_grad()
                outputs = model(imgs)
                loss = combined_loss(outputs, masks)

                loss.backward()
                optimizer.step()

                # Step the scheduler
                scheduler.step()
                current_lr = scheduler.get_last_lr()[0]
                lr_list.append(current_lr)

                train_loss += loss.item()

                if (i+1) % 20 == 0:
                    acc = calculate_accuracy(outputs, masks)
                    iou = calculate_iou(outputs, masks)
                    dice = calculate_dice(outputs, masks)
```

```
                    print(f"[TRAIN] Epoch [{epoch+1}/{epochs}], Step
[{i+1}/{len(train_loader)}], Loss: {loss.item():.4f}, Acc: {acc:.4f}, IoU:
{iou:.4f}, Dice: {dice:.4f}")

                    # Save metrics to CSV
                    csvwriter.writerow([epoch+1, i+1, 'TRAIN', loss.item(), acc,
iou, dice])

            train_loss /= len(train_loader)
            train_loss_list.append(train_loss)

            # Validation loop
            model.eval()
            val_loss = 0.0
            with torch.no_grad():
                for i, (imgs, masks) in enumerate(val_loader):
                    imgs, masks = imgs.to(device), masks.to(device)

                    outputs = model(imgs)
                    loss = combined_loss(outputs, masks)

                    val_loss += loss.item()

                    if (i+1) % 20 == 0:
                        acc = calculate_accuracy(outputs, masks)
                        iou = calculate_iou(outputs, masks)
                        dice = calculate_dice(outputs, masks)

                        print(f"[VAL] Epoch [{epoch+1}/{epochs}], Step
[{i+1}/{len(val_loader)}], Loss: {loss.item():.4f}, Acc: {acc:.4f}, IoU:
{iou:.4f}, Dice: {dice:.4f}")

                        # Save metrics to CSV
                        csvwriter.writerow([epoch+1, i+1, 'VAL', loss.item(),
acc, iou, dice])

            val_loss /= len(val_loader)
            val_loss_list.append(val_loss)

            # Save model weights
            torch.save(model.state_dict(),
f"/kaggle/working/PTH_UNET_Lr004_AdamOptim_1cycle_CombinedLoss_DataAugepoch_{epoc
h+1}.pth")

            print(f"Epoch [{epoch+1}/{epochs}], Train Loss: {train_loss:.4f}, Val
Loss: {val_loss:.4f}")

    # Plotting train and validation loss
    plt.figure(figsize=(18, 6))
    plt.subplot(1, 2, 1)
    plt.plot(range(1, epochs+1), train_loss_list, label='Train Loss')
    plt.plot(range(1, epochs+1), val_loss_list, label='Validation Loss')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
```

```
    plt.title('Train vs Validation Loss')
    plt.legend()

    # Plotting learning rates
    plt.subplot(1, 2, 2)
    plt.plot(lr_list)
    plt.xlabel('Iteration')
    plt.ylabel('Learning Rate')
    plt.title('Learning Rate Schedule')

    plt.show()

# %% [markdown]
# # A ENTRENAR

# %% [code]
import time

start_time = time.time()  # Record the start time

train_model_scheduler(model, optimizer, scheduler, epochs)

end_time = time.time()

elapsed_time = end_time - start_time
hours, rem = divmod(elapsed_time, 3600)
minutes, seconds = divmod(rem, 60)

print(f"Total time taken: {int(hours)}:{int(minutes)}:{int(seconds)}")

# %% [markdown]
# ## ANÁLISIS MÉTRICAS DE RENDIMIENTO

# %% [code]
import pandas as pd
import matplotlib.pyplot as plt

# Read the CSV file into a DataFrame
df =
pd.read_csv('/kaggle/working/CSV_UNET_Lr001_AdamOptim_CombinedLoss_DataAug_highre
s.csv')

# Filtering the DataFrame for the "TRAIN" phase
df_train = df[df['Phase'] == 'TRAIN']

# Filtering the DataFrame for the "VAL" phase
df_val = df[df['Phase'] == 'VAL']

# Creating tables for the "TRAIN" phase
train_table = df_train[['Epoch', 'Step', 'Accuracy', 'IoU', 'Dice']]

# Creating tables for the "VAL" phase
val_table = df_val[['Epoch', 'Step', 'Accuracy', 'IoU', 'Dice']]
```

```
print("Metrics for TRAIN phase:")
print(train_table)

print("\nMetrics for VAL phase:")
print(val_table)


# Create new x-axis arrays for training and validation
x_train_new = np.linspace(0, 6350, len(df_train))
x_val_new = np.linspace(0, 2500, len(df_val))

# Plotting for the training phase
plt.figure(figsize=(18, 6))
plt.title('Training Metrics')
plt.ylabel('Metrics Value')
plt.xlabel('Iteration')
plt.xlim(0, 6350)  # Set x-axis range for training
plt.plot(x_train_new, df_train['Accuracy'].values, label='Accuracy', color='r')
plt.plot(x_train_new, df_train['IoU'].values, label='IoU', color='g')
plt.plot(x_train_new, df_train['Dice'].values, label='Dice', color='b')
plt.legend()
plt.show()

# Plotting for the validation phase
plt.figure(figsize=(18, 6))
plt.title('Validation Metrics')
plt.ylabel('Metrics Value')
plt.xlabel('Iteration')
plt.xlim(0, 2500)  # Set x-axis range for validation
plt.plot(x_val_new, df_val['Accuracy'].values, label='Accuracy', color='r')
plt.plot(x_val_new, df_val['IoU'].values, label='IoU', color='g')
plt.plot(x_val_new, df_val['Dice'].values, label='Dice', color='b')
plt.legend()
plt.show()

# %% [markdown]
# # VALIDAR RESULTADOS CON PREDICCIONES

# %% [code]
#model = UNET(3, 64, 1).to(device)
#model_path = '/kaggle/working/unet_epoch_5.pth'
#model_dict = torch.load(model_path)
#model.load_state_dict(model_dict)

# Assuming small_data contains your small batch of images and masks
imgs, masks = next(iter(test_loader))
imgs, masks = imgs.to(device), masks.to(device)

# Forward pass to get the model's predictions
with torch.no_grad():
    outputs = model(imgs)

# Calculate Dice scores for the batch
dice_scores = calculate_dice(outputs, masks)
```

```
# Move to CPU and convert to NumPy for visualization
imgs = imgs.cpu()
masks = masks.cpu()
outputs = outputs.cpu()


# Plot original images with ground truth
print("Original Images with Ground Truth Masks")
plot_mini_batch(imgs, masks, alpha=True)

# Plot original images with predicted masks
print("Original Images with Predicted Masks")
plot_mini_batch(imgs, outputs, alpha=True)


# Plot masks with predicted masks
print("Ground Truths with Predicted Masks")
plot_mini_batch(masks, outputs, alpha=True)

# Plot original images with predicted masks
print("Ground truth vs predictions")
plot_mini_batch(masks)
plot_mini_batch(outputs)


plot_heatmaps_big(masks, outputs)

# Print Dice scores
print("Dice Scores:", dice_scores)

# %% [markdown]
# # POST PROCESADO

# %% [code]
import cv2
import torch
import numpy as np
from PIL import Image
from torchvision import transforms as T
from torch.utils.data import DataLoader
from torchvision.transforms import functional as F


device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
#model = UNET(3, 64, 1).to(device)
#model.load_state_dict(torch.load("/kaggle/working/UNET_UNET_Lr001_AdamOptim_Comb
inedLoss_DataAug_highres_epoch_10.pth"))
model = U

model.load_state_dict(torch.load("/kaggle/working/UNET_test_epoch_1.pth"))

model.eval()
```

```
# %% [code]
def preprocess_frame(frame):
    # Convert to PIL Image
    img = Image.fromarray(cv2.cvtColor(frame, cv2.COLOR_BGR2RGB))

    # Apply transformations
    trans = T.Compose([
        T.ToTensor()
    ])
    img = trans(img)

    # Add batch dimension
    img = img.unsqueeze(0)

    return img

def postprocess_output(output):
    output = output.squeeze().cpu().numpy()
    output = (output * 255).astype(np.uint8)
    return output

# %% [markdown]
# ## GENERAR VIDEO DE PREDICCIONES CON INFERENCIA

# %% [code]
from IPython.display import clear_output

# Initialize VideoCapture and VideoWriter
cap = cv2.VideoCapture("/kaggle/input/dataset-diferentes/N2 on food
L_2010_09_17__11_15_30___6___1fps_30nf_26999.avi")
fourcc = cv2.VideoWriter_fourcc(*'XVID')
# Obtener la tasa de fotogramas del video original
fps = int(cap.get(cv2.CAP_PROP_FPS))
out = cv2.VideoWriter('segmented_output_highres_v4.avi', fourcc, fps, (640, 480),
isColor=False)



while(cap.isOpened()):
    ret, frame = cap.read()

    if ret:
        # Preprocess frame
        input_tensor = preprocess_frame(frame).to(device)

        # Perform inference
        with torch.no_grad():
            output = model(input_tensor)

        # Post-process output
        segmented_frame = postprocess_output(output)

        # Ensure it's uint8 and single channel
        segmented_frame = segmented_frame.astype(np.uint8)
```

```
        if len(segmented_frame.shape) == 3:
            segmented_frame = cv2.cvtColor(segmented_frame, cv2.COLOR_BGR2GRAY)


        # Write the frame
        out.write(segmented_frame)

        plt.imshow(segmented_frame, cmap='gray')
        plt.title('Segmented Frame')
        plt.axis('off')
        plt.show()

        # Clear the output to overwrite the plot
        clear_output(wait=True)

    else:
        break

# Release VideoCapture and VideoWriter
cap.release()
out.release()
print("ended")
```

6. Cálculo de características de *Healthspan*

Se calculan algunas de las características de Healthspan de los *C. elegans* y se almacenan los resultados en un .csv

```
import cv2
import numpy as np
from skimage.morphology import skeletonize
import matplotlib.pyplot as plt
import math
import pandas as pd

def get_skeleton_extremes(skeleton):
    # Encuentra los puntos del esqueleto
    y, x = np.where(skeleton == 1)

    extremes = []
    for i in range(len(x)):
        xi, yi = x[i], y[i]

        # Cuenta los vecinos del punto (xi, yi)
        neighbors = np.sum(skeleton[yi-1:yi+2, xi-1:xi+2]) - 1  # Resta 1 para
excluir el punto mismo

        # Si tiene solo un vecino, es un extremo
        if neighbors == 1:
            extremes.append((xi, yi))

    return extremes

#############################################################################

# Función para caminar a lo largo del esqueleto y ordenar los puntos
def order_skeleton_points(points, start_point):
    ordered_points = [start_point]
    remaining_points = set(points)
    remaining_points.remove(start_point)

    current_point = start_point

    while remaining_points:
        next_point = min(remaining_points, key=lambda p: (p[0] -
current_point[0])**2 + (p[1] - current_point[1])**2)
        ordered_points.append(next_point)
        remaining_points.remove(next_point)
        current_point = next_point

    return ordered_points

def find_middle_point_length(ordered_points):
    # Initialize variables
    total_distance = 0
    cumulative_distances = [0]

    # Calculate the total distance and cumulative distances
```

```
    for i in range(1, len(ordered_points)):
        distance = np.linalg.norm(np.array(ordered_points[i]) -
np.array(ordered_points[i-1]))
        total_distance += distance
        cumulative_distances.append(total_distance)

    # Find the middle point
    half_distance = total_distance / 2
    middle_point_index = np.argmin(np.abs(np.array(cumulative_distances) -
half_distance))

    return ordered_points[middle_point_index], middle_point_index, total_distance

#########################################################################

def process_frame(frame, good_frames):
    # Inicializar variables
    length_microns = np.nan
    area_length_ratio = np.nan

    # Aplicar un umbral para binarizar la imagen
    _, thresh = cv2.threshold(frame, 127, 255, cv2.THRESH_BINARY)

    # Convertir la imagen en escala de grises a color
    color_frame = cv2.cvtColor(frame, cv2.COLOR_GRAY2BGR)
    microns_per_pixel = 4.458167496324821

    # Encontrar contornos
    contours, _ = cv2.findContours(thresh, cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_SIMPLE)

    # Suponemos que el contorno más grande es el del C. elegans
    c = max(contours, key=cv2.contourArea)

    # Calculamos características morfológicas y de postura del gusano
    area = cv2.contourArea(c)
    area_microns = area * (microns_per_pixel ** 2)
    print(f"The total area of the worm is: {area_microns}")

    # El eje menor y mayor se pueden obtener del rectángulo delimitador
    rect = cv2.minAreaRect(c)
    _, (a, b), _ = rect
    major_axis = max(a, b)
    minor_axis = min(a, b)

    # Calcula la quirkiness
    quirkiness = math.sqrt(1 - (minor_axis ** 2) / (major_axis ** 2))
    print("quirkiness  ", quirkiness)

    # Esqueletonización
    skeleton = skeletonize(thresh // 255)
    y, x = np.where(skeleton == 1)
    skeleton_points = list(zip(x, y))
```

```
    # Obtenemos los extremos del esqueleto
    extremes = get_skeleton_extremes(skeleton)

    # Solo si se cumple que hay exactamente 2 extremos se calcula el centro,
longitud y demás
    if len(extremes) == 2:
        extremo1, extremo2 = extremes

        # Dibujamos puntos extremos
        cv2.circle(color_frame, extremo1, 5, (0, 0, 255), -1)  # Punto extremo
izquierdo en rojo
        cv2.circle(color_frame, extremo2, 5, (255, 0, 0), -1)  # Punto extremo
derecho en azul

        # Ordenamos los puntos del esqueleto para que sean consecutivos
        ordered_points = order_skeleton_points(skeleton_points, extremo2)

        if len(ordered_points) != len(skeleton_points):
            raise ValueError("No mismo numero de puntos")

        # Buscamos el punto central y calculamos la longitud
        middle_point, middle_point_index, total_distance =
find_middle_point_length(ordered_points)
        #print(f"The middle point is at {middle_point} with index
{middle_point_index}")

        # Pasamos de pixeles a micras los valores obtenidos
        length_microns = total_distance * microns_per_pixel
        print(f"The total length of the worm is: {length_microns}")
        area_length_ratio = area_microns / length_microns
        print(f"The area/length ratio is : {area_length_ratio}")

        # Dibujamos punto medio
        cv2.circle(color_frame, middle_point, 5, (128, 0, 128), -1)  # Punto
medio en morado

        good_frames = good_frames + 1

    else:
        print("Se han encontrado más de dos extremos. Saltando cálculos para este
frame.")



    # Dibujar en la imagen el contorno
    cv2.drawContours(color_frame, [c], -1, (0, 255, 0), 1)  # Contorno en verde

    # Para dibujar el rectangulo
    box = cv2.boxPoints(rect)
    box = np.intp(box)
    cv2.drawContours(color_frame, [box], 0, (0, 0, 255), 2)

    # Superponer el esqueleto en la imagen original
    skeleton = (skeleton * 255).astype(np.uint8)
```

```
    color_frame[skeleton == 255] = [0, 0, 0]  # Esqueleto en blanco

    return color_frame, good_frames, area_microns, length_microns,
area_length_ratio, quirkiness



if __name__ == "__main__":
    video_path = "C:/Users/alexb/Downloads/segmented_output_raw.avi"
    cap = cv2.VideoCapture(video_path)
    frame_cnt = 0
    good_frames = 0

    # Inicializar un dataframe vacio
    df = pd.DataFrame(columns=['frame_cnt', 'length', 'area',
'area_length_ratio', 'quirkiness'])

    while True:
        ret, frame = cap.read()

        if not ret:
            print("Failed to grab frame or end of video reached")
            break

        frame_cnt = frame_cnt  + 1

        frame_gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)

        kernel_size = 3
        kernel = np.ones((kernel_size, kernel_size), np.uint8)

        # Apply opening (erosion followed by dilation)
        frame_open = cv2.morphologyEx(frame_gray, cv2.MORPH_OPEN, kernel)

        # Apply closing (dilation followed by erosion)
        frame_close = cv2.morphologyEx(frame_open, cv2.MORPH_CLOSE, kernel)

        # Procesamos el frame actual
        processed_frame, good_frames, area, length, area_length_ratio, quirkiness
= process_frame(frame_close, good_frames)

        # Esto para ver que forma tienen las imagenes con datos anómalos
        if quirkiness < 0.4:
            cv2.imwrite(f"./basura/quirkiness/{frame_cnt}.jpg", processed_frame)

        if length > 1900:
            cv2.imwrite(f"./basura/length/{frame_cnt}.jpg", processed_frame)

        if area > 89000:
            cv2.imwrite(f"./basura/area/{frame_cnt}.jpg", processed_frame)

        # Append the results to the DataFrame using pandas.concat
        new_row = pd.DataFrame({
            'frame_cnt': [frame_cnt],
```

```
            'length': [length],
            'area': [area],
            'area_length_ratio': [area_length_ratio],
            'quirkiness': [quirkiness]
        })
        df = pd.concat([df, new_row]).reset_index(drop=True)

        cv2.imshow('Processed Frame', processed_frame)
        if cv2.waitKey(10) & 0xFF == ord('q'):
            break

    cap.release()
    cv2.destroyAllWindows()
    print("Total frames", frame_cnt)
    print("Good frames", good_frames)

    # Save the DataFrame to a CSV file
    df.to_csv('./results3.csv', index=False)
```

7. Comparación de características de movilidad

Se comparan y grafican las características de movilidad obtenidas con Tierpsy Tracker contra las calculadas manualmente.

```
import pandas as pd
import matplotlib.pyplot as plt
import h5py


# Leer datos del archivo HDF5
with
h5py.File('C:/Users/alexb/OneDrive/Escritorio/Master/TFM/Dataset/files_feat_data/
files_feat_data/N2_train/N2 on food L_2010_09_17__11_15_30___6___1.hdf5', 'r') as
f:
    data = f['features_timeseries'][:]
    df_hdf5 = pd.DataFrame(data)

area = df_hdf5['area']
length = df_hdf5['length']
area_length_ratio = df_hdf5['area_length_ratio']

# Cargar los datos del CSV en un dataframe
df = pd.read_csv('./results3.csv')

# Filtrar valores anómalos de longitud
df.loc[df['length'] > 1400, 'length'] = None

# Forward Fill. Rellenar valores con NaN con el ultimo valor anterior no NaN
df.fillna(method='ffill', inplace=True)
df['area_length_ratio2'] = df['area'] / df['length']


# Graficar los valores segun metodo propio
plt.figure(figsize=(18, 6))
plt.plot(df['frame_cnt'], df['length'], label='Length', color='b')
plt.xlabel('Frame Count')
plt.ylabel('Length')
plt.title('Length over Frame Count')
plt.legend()
plt.show()

plt.figure(figsize=(18, 6))
plt.plot(df['frame_cnt'], df['area'], label='Area', color='r')
plt.xlabel('Frame Count')
plt.ylabel('Area')
plt.title('Area over Frame Count')
plt.legend()
plt.show()

plt.figure(figsize=(18, 6))
plt.plot(df['frame_cnt'], df['area_length_ratio2'], label='Area/Length Ratio',
color='g')
plt.xlabel('Frame Count')
plt.ylabel('Area/Length Ratio')
```

```
plt.title('Area/Length Ratio over Frame Count')
plt.legend()
plt.show()

# Plot quirkiness over frame count
plt.figure(figsize=(18, 6))
plt.plot(df['frame_cnt'], df['quirkiness'], label='Quirkiness', color='m')
plt.xlabel('Frame Count')
plt.ylabel('Quirkiness')
plt.title('Quirkiness over Frame Count')
plt.legend()
plt.show()


# Graficar las comparaciones entre metodo propio y Tierpsy
plt.figure(figsize=(18, 6))
plt.plot(area, label='Tierpsy Tracker')
plt.plot(df['area'], label='Cálculo Manual')
plt.xlabel('Frame Count')
plt.ylabel('Area')
plt.title('Area over Frame Count')
plt.legend()
plt.show()

plt.figure(figsize=(18, 6))
plt.plot(length, label='Tierpsy Tracker')
plt.plot(df_hdf5['length'], label='Tierpsy Tracker')
plt.plot(df['length'], label='Cálculo Manual')
plt.xlabel('Frame Count')
plt.ylabel('Length')
plt.title('Length over Frame Count')
plt.legend()
plt.show()


plt.figure(figsize=(18, 6))
plt.plot(area_length_ratio, label='Tierpsy Tracker')
plt.plot(df['area_length_ratio2'], label='Cálculo Manual')
plt.title('Area/Length Ratio')
plt.xlabel('Frame Count')
plt.title('Area/Length Ratio over Frame Count')
plt.legend()
plt.show()
```