Universidad Zaragoza
1542

Trabajo Fin de Grado

Aplicación de algoritmos de Machine Learning a la clasificación y reconocimiento de partículas en fluidos multifásicos

Application of Machine Learning to classification and recognition of particles in multiphase fluids

Autor

Jorge Condor Lacambra

Directores

Francisco José Torcal Milla

Ana María López Torres

GRADO EN INGENIERÍA ELECTRÓNICA Y AUTOMÁTICA

ESCUELA DE INGENIERÍA Y ARQUITECTURA

2020

Aplicación de algoritmos de Machine Learning a la clasificación y reconocimiento de partículas en fluidos multifásicos

RESUMEN

La holografía digital ha emergido como una posible herramienta útil en aplicaciones de biomedicina, aunque ciertas dificultades se interponen para que alcance todo su potencial. En particular, la tarea de localizar y clasificar partículas de diferentes tamaños presentes en fluidos, a partir de la información tridimensional que proporciona registrarlas en un holograma, es especialmente complicada cuando la densidad de partículas es elevada y/o los volúmenes analizados son de gran tamaño. Esta tarea es el primer paso de cara al desarrollo de nuevas herramientas contra el cáncer y análisis tridimensionales del riego sanguíneo. En este trabajo se propone una solución basada en machine learning para dicho problema. Se eligen los datos de entrada, generados a partir de un algoritmo sintetizador de hologramas, y los planos de salida adecuados para cumplir este objetivo. Se adapta una arquitectura basada en U-Net, aportando varias novedades. Se escoge un método de entrenamiento adecuado para el modelo, y se entrena el mismo en numerosas ocasiones hasta conseguir los hiperparámetros más óptimos. Los resultados que ofrece este modelo son muy positivos, y se especula sobre la posibilidad de extrapolarlos a casos reales. Finalmente se sugieren ciertas vías por las que continuar mejorando el algoritmo y los retos que deberá superar aún la técnica para tener una utilidad directa en el campo de la biomedicina.

Índice

1. Introducción

El objetivo de este Trabajo de Fin de Grado es desarrollar una aplicación basada en machine learning capaz de clasificar partículas micrométricas en el contexto de canales milimétricos, a partir de hologramas digitales de dichos canales. En particular, esta herramienta podría resultar útil en la búsqueda de nuevos tratamientos contra el cáncer, al ser empleada en el análisis del comportamiento de determinadas partículas en los vasos sanguíneos. En la actualidad, este problema se resuelve mediante la utilización de filtros adaptados[1], pero técnicas de machine learning han comenzado a utilizarse en el campo de la holografía digital para el reconocimiento de partículas, células o microorganismos en un fluido [2][3][4], por lo que existe un creciente interés en poner esta herramienta en marcha para el problema descrito.

1.1. Trascendencia y Justificación del Trabajo

La localización y clasificación de partículas de diversos tamaños o naturaleza en suspensión en microcanales es una tarea de interés en el campo de la biomedicina, particularmente en el estudio de la sangre. Una aplicación de estas características está relacionada directamente con el estudio de las primeras etapas de la trombosis, causada por acumulación de células sanguíneas, en particular plaquetas[5]. De igual forma, investigaciones modernas en busca de mejores tratamientos contra el cáncer abogan por desarrollar técnicas menos intrusivas y reducir a mínimos los efectos secundarios que un determinado tratamiento sobre cierto cáncer pueda tener en el resto del cuerpo, como ocurre en las técnicas de quimioterapia y radioterapia modernas (caída de pelo, náuseas, daño a células sanas, fatiga, entre otros). Una de estas líneas de investigación gira en torno a atacar directamente a las células cancerosas, dirigiendo los medicamentos a través de la sangre usando campos magnéticos y activándolos únicamente una vez han alcanzado la masa cancerígena[6]. Para que una técnica de estas características funcione, primero es necesario entender la dinámica de estas partículas en los vasos sanguíneos, lo que implica previamente comprender la influencia de otros elementos, como células sanguíneas, en dichas partículas magnéticas. Se ha comenzado con experimentos sencillos [1], en los que se analiza un flujo con dos tipos de partículas que se diferencian únicamente en el tamaño (23 μm y 3 μm), en capilares rectangulares de unos pocos mm de grosor.

1.2. Detalles Técnicos y Problemática Inicial

La holografía permite el registro de la información de un objeto tridimensional mediante la captura en un medio fotosensible de la interferencia de la luz difractada por dicho objeto al ser iluminado por luz coherente (láser) y una onda de referencia. La reconstrucción de un holograma permite obtener la imagen tridimensional de dicho objeto a partir de las características de la forma de onda capturada (amplitud y fase) [7]. Esto amplía las características que pueden aportar información de entrada a una posible red convolucional que se utilice para modelar las características del objeto. En este trabajo, el objeto está formado por un conjunto de partículas en un fluido y el objetivo del sistema de machine learning es clasificar estas partículas y localizarlas en el espacio tridimensional.

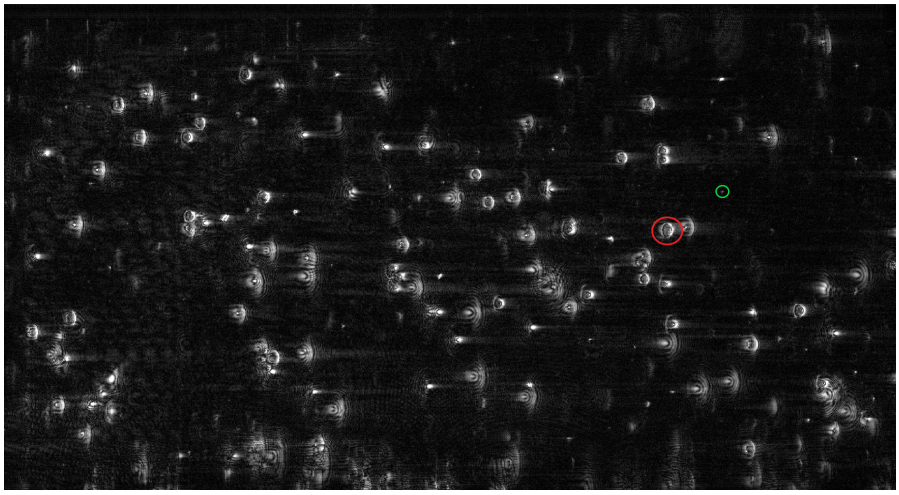

Figura 1: Proyección en Z=0 (centro del canal) del holograma. En rojo, un ejemplo de partícula blanca, y en verde, uno de magnética. Se puede apreciar también la presencia de ruido.

El Trabajo parte inicialmente con un único holograma digital. En éste se pueden apreciar dos clases distintas de partículas, a las que denominaremos partículas blancas y magnéticas. La diferencia entre ambas es el tamaño: las partículas blancas son considerablemente más grandes que las magnéticas, en torno a 8-10 veces más. En la figura 1 se puede observar la proyección de uno de los planos (la imagen 2D en una profundidad Z del canal dada, una sección del mismo) de dicho holograma. Una de las dificultades que entraña la localización, y en especial la clasificación, de las partículas, a partir de proyecciones de planos individuales del holograma, reside en las proyecciones de dichas partículas: lo que en un plano puede parecer una partícula magnética centrada (plano proyectado y la localización Z de la partícula son la misma) podría en realidad ser una partícula blanca observada en un plano desplazado (uno de sus bordes, por ejemplo). Por ello, se valora usar otras proyecciones que aporten más

información sobre las partículas, como las de intensidad y fase máximas y mínimas. Esta tarea es complicada para los algoritmos tradicionales, particularmente cuando la densidad de partículas es elevada y su tamaño pequeño, por lo que soluciones basadas en machine learning han empezado a ser consideradas.

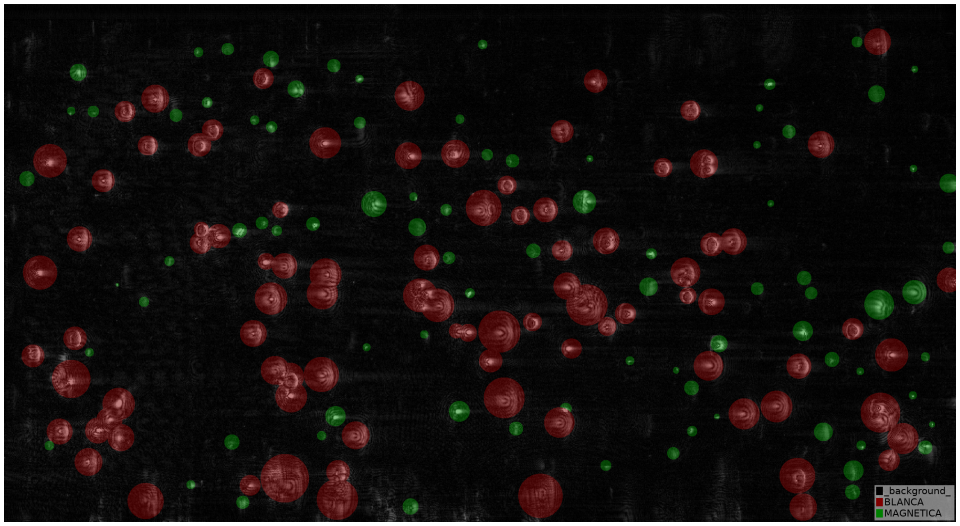

Figura 2: Proyección del holograma de la figura 1 segmentada, con los píxeles del mismo clasificados como partículas blancas, magnéticas o background (fondo). Creado con Labelme[8] (Anexo I)

El primer problema que presenta dicho enfoque es la construcción de un dataset: las redes convolucionales necesitan habitualmente de ingentes cantidades de datos sobre los que entrenarse, y registrar hologramas en laboratorio es una tarea costosa, en cuanto a tiempo se refiere. Además, construir un dataset basado en hologramas originales habría implicado clasificar y localizar a mano muchas de las partículas, debido a los ya mencionados problemas de los algoritmos tradicionales para realizar esta tarea. Aunque inicialmente comencé trabajando en esta línea y realicé un mapa segmentado de una de las proyecciones del holograma, como se puede ver en la figura 2, pronto vi la inviabilidad de escalar este proceso a un número razonable de hologramas, suficiente para entrenar la red. Para suplir esta lacra, se decidió que era necesario crear un sintetizador de hologramas, que acorde con las características de tamaño y densidad de partículas por holograma, entre otras, fuera capaz de generar tantos hologramas como fueran necesarios para entrenar el modelo, con el objeto final de entrenar el algoritmo de machine learning desarrollado en hologramas reales si los resultados en el entorno simulado fueran satisfactorios. Este sintetizador no produce representaciones exactas del holograma de referencia, pero estos hologramas sintéticos servirán para probar la validez de la técnica utilizada. El sintetizador fue proporcionado por los directores de este Trabajo.

1.3. Introducción a las Redes Neuronales Convolucionales

Información tomada de [9] y conocimientos previos

La estructura básica de cualquier red neuronal artificial es una serie de unidades (neuronas), conectadas entre sí mediante enlaces. Las neuronas están organizadas en capas, teniendo enlaces a neuronas de capas anteriores (entradas) y de capas posteriores (salidas). Neuronas y enlaces poseen un valor (peso). El proceso de aprendizaje de una red neuronal artificial es el siguiente: el valor de salida de cada neurona es multiplicado por el peso de los enlaces. Al llegar a una neurona de la capa siguiente, el valor resultante de sumar todos los valores de las entradas a dicha neurona se multiplica por el peso de la misma, y se procesa por una función (lineal o no), conocida como función de activación, que modifica el valor resultado o impone un límite que no se puede sobrepasar antes de propagarse a otra neurona. Al llegar a la capa de salida final de la red, una función conocida como función de pérdidas calculará el error cometido por la misma, evaluando la calidad con la que la red modela el set de datos dados. Un algoritmo calculará el gradiente del error (en el ejemplo clásico se utiliza el de retro-propagación o backpropagation, que lo calcula propagando hacia las capas anteriores el valor de pérdidas). Con este gradiente, una función conocida como optimizador será capaz de actualizar los pesos de la red. El tamaño del paso que el optimizador hará dar a la red en dirección contraria a la del gradiente es conocido como ritmo de aprendizaje o learning rate. Este proceso es iterativo y se produce el número de veces que se considere oportuno. El conjunto de datos que entra en la red, sobre los que se entrena, se conoce como dataset de entrenamiento. Cada cierto número de iteraciones (epoch o época), se calculan las pérdidas usando datos sobre los que la red no se entrena, el dataset de validación, para analizar el progreso del aprendizaje de la red durante su entrenamiento de forma independiente.

Las redes neuronales convolucionales (Convolutional Neural Network o CNN en adelante) son de especial interés para este Trabajo. A diferencia de las redes neuronales totalmente conectadas, las neuronas en una capa de una CNN no están conectadas con todas y cada una de las neuronas de la siguiente capa, sino simplemente a las neuronas de la siguiente capa más cercanas, al mismo tiempo que los pesos en cada localización se comparten, reduciendo de forma significativa el número de parámetros de la red. Esto permite obtener modelos más rápidos y ligeros.

La base de cualquier CNN son las capas de convolución. En éstas, se aplica la operación

matemática de convolución de uno o varios filtros (kernel) a lo largo y ancho de un tensor de entrada, resultando en un segundo tensor con un mayor número de canales, acorde al número de filtros aplicados al mismo. Estos canales contienen las distintas características del tensor (en la entrada a la red, la imagen original) de entrada, desde aquellas de bajo nivel (formas geométricas, detector de esquinas) hasta las de más alto nivel (formas complejas como pelo, ojos o zarpas en el caso de animales, por ejemplo). El tamaño de este kernel es el que determinará el número de parámetros entrenables que tendrá la red, persiguiendo un equilibrio entre demasiados parámetros (red muy pesada y lenta al hacer predicciones) y demasiado pocos (red demasiado ligera, incapaz de detectar suficientes características de la imagen de entrada para clasificarla o segmentarla adecuadamente).

1.4. Organización del Trabajo y Contenido de las Secciones

El Trabajo ha sido desarrollado en varias fases, que aquí presento en orden cronológico. Inicialmente se desarrolla una tarea de investigación del estado de la técnica, recopilando información sobre diversas arquitecturas de redes convolucionales. Una vez definidas las características del modelo en la fase de investigación, se diseña y genera el dataset más adecuado para entrenar dicho modelo. Posteriormente, el modelo es adaptado y programado. Después, se procede a diseñar el entrenamiento del modelo, elegir funciones de pérdidas para cada canal de salida y una serie de hiperparámetros que controlan las características de dicho entrenamiento. Esta es la fase que más tiempo requirió, necesitando de un exhaustivo ejercicio de experimentación. A continuación, los resultados obtenidos por el procedimiento son evaluados, acorde a distintos parámetros. Por último, aporto ciertas ideas sobre posibles áreas de mejora, que por falta de tiempo o recursos no han podido ser implementadas en el Trabajo definitivo. Concluyo aportando una valoración crítica del conjunto del trabajo realizado.

1.5. Anexos

Aporto dos anexos al trabajo. El primero (**Anexo I**) incluye una breve descripción del software y herramientas utilizadas, y el código de todos los algoritmos desarrollados. El segundo (**Anexo II**) incluye enlaces a carpetas en Google Drive donde se guardan los datasets y el archivo con el modelo entrenado y su historial de pérdidas.

2. Investigación Previa y Primeras Ideas

El objetivo del algoritmo es segmentar un volumen 3D, clasificando unas regiones como un tipo de partículas u otro. Por lo tanto, comencé estudiando las distintas arquitecturas que se usan actualmente para tareas de segmentación. En una primera fase de investigación del trabajo, recopilé información de diversas fuentes sobre distintas arquitecturas de machine learning aplicadas a la segmentación de imágenes y, en particular, en el contexto de la holografía, aquellas utilizadas en la clasificación de partículas. La mayoría de técnicas de vanguardia en este respecto actualmente abogan por usar FCN o Fully Convolutional Networks[10]. La característica más distintiva de estas redes es la eliminación de toda capa neuronal densamente conectada (densely connected layers), lo que permite entrenar los modelos, e inferir predicciones con ellos una vez entrenados, con imágenes de cualquier resolución, a diferencia de las redes neuronales tradicionales, que poseen, al menos en la salida, capas como las anteriormente mencionadas. Esto hace que la propia arquitectura del modelo limite la resolución y por ende el potencial del modelo para obtener mejores resultados. El problema de las FCN originales es su dificultad para predecir pequeños grupos o hacer segmentaciones con detalles finos y una buena resolución, problemática que se puede observar en la figura 3. Dado que aumentar el factor de reducción de la resolución de la imagen original en la capa más profunda aumenta la calidad de la extracción de características, lo cual es deseable, la cuestión que había que solucionar es mejorar las capacidades del decodificador. Esto es lo que consiguen los autores de [11], creando la arquitectura U-Net. La arquitectura utilizada por este Trabajo está basada en una U-Net, con varias modificaciones especialmente diseñadas para esta tarea, por lo que explicaré en detalle la misma en la sección 4. Numerosas versiones de U-Net, afinadas específicamente para las características de los datos que son necesarios segmentar en cada caso, han aparecido desde que irrumpió el estudio citado. Dos estudios llamaron especialmente mi atención, dada la naturaleza de este trabajo. En [12], los autores proponen una arquitectura basada en U-Net capaz de localizar en 3 dimensiones partículas en hologramas sintéticos. La utilidad de dicha arquitectura para la segmentación de partículas de distintos tamaños a partir de hologramas y distintas proyecciones provenientes de los mismos queda corroborada en [13], donde los autores consiguen resultados positivos usando esa misma arquitectura en tareas de segmentación por tamaños de partículas. Este modelo es similar al utilizado en la solución final de este Trabajo, y de nuevo será introducido en mayor profundidad en la sección 4.

Otra arquitectura que podría dar resultados positivos para la tarea entre manos es

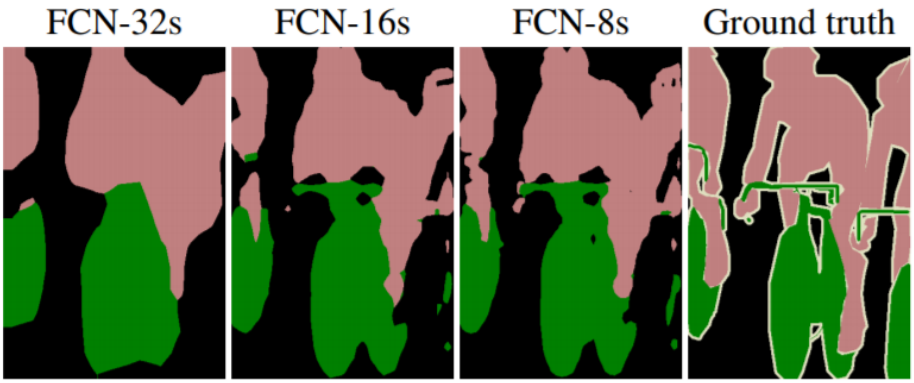

Figura 3: Comparativa de la calidad de las máscaras generadas por una FCN, en función del factor de reducción de resolución en su cuello de botella o capa más profunda. Origen [10]

la denominada GAN o Generative Adversarial Network, introducida en [14]. Las GAN se caracterizan por estar compuestas de un generador y un discriminador: el generador se encarga de generar, a partir de ruido, una imagen que se asemeje a la que introducimos como entrada o input; el discriminador, compara ambas imágenes, la generada y la original, y decide cuál es la real y cuál es la sintética. En este proceso, se producen 2 fuentes de pérdidas: las pérdidas del generador, que se utilizarán para entrenarlo y producir cada vez imágenes más realistas, y las del discriminador, que permiten que cada vez discrimine imágenes reales de sintéticas con mayor precisión. Esta estructura se puede observar en la figura 4.

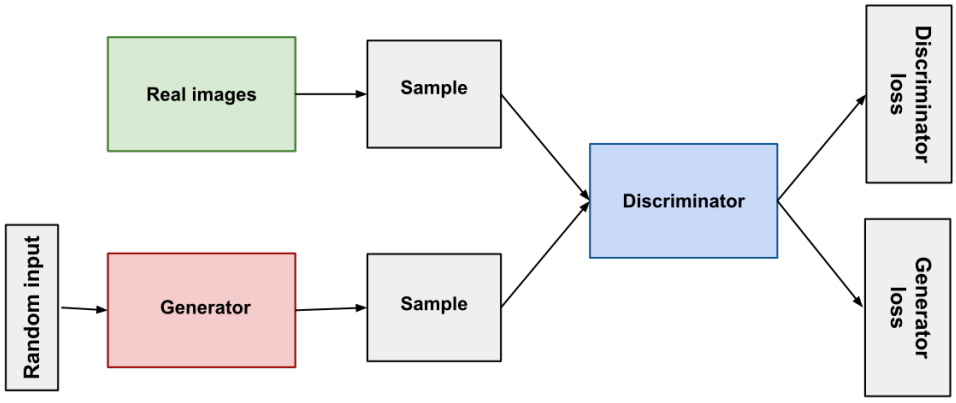

Figura 4: Esquema básico de una GAN. Origen [15]

Las arquitecturas basadas en GAN reciben mucha atención en la actualidad, y cada año se encuentran nuevas utilidades para las mismas. Algunos ejemplos son generación de caras sintéticas fotorrealistas [16], traducción imagen-imagen [17], traducción de imagen semántica a imagen fotorrealista [18] o incrementar la resolución de imágenes [19].

Una arquitectura basada en GAN en particular podría ser de especial interés en esta tarea, la CycleGAN [20]. A diferencia de la GAN original, CycleGAN incluye 2 generadores: uno parte de la imagen de entrada a la imagen objetivo, mientras que el otro traduce el objetivo de vuelta a la entrada. El discriminador compara ambas, imagen traducida e imagen de entrada. Esto permite obtener traductores imagen-imagen, que han sido demostrados como útiles en tareas de segmentación como la objeto de este Trabajo, tal y como demuestran estudios como [21]. La estructura descrita se puede observar en la figura 5.

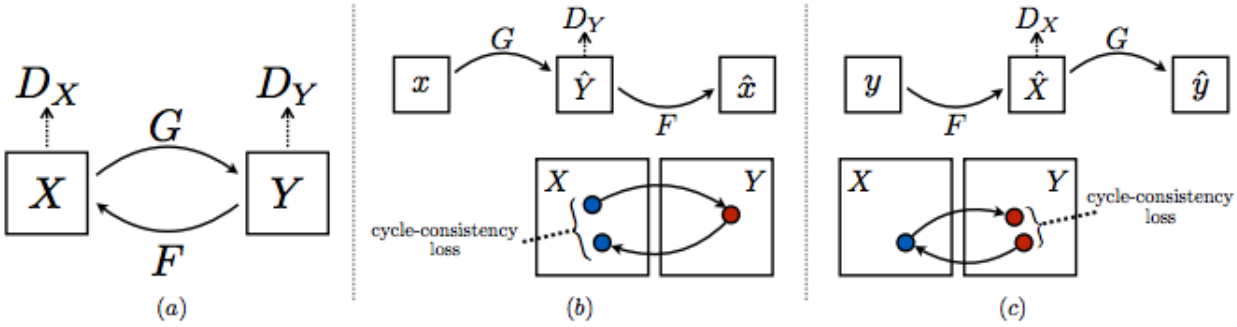

Figura 5: Esquema de una CycleGAN. Dx, Dy son los discriminadores; G y F, los generadores que intentan traducir una imagen X a Y, y de Y a X respectivamente. Además introducen el concepto de Cycle-consistent losses: intuitivamente, si se traduce de X-Y-X o Y-X-Y, el resultado debería ser la imagen original, por lo que las pérdidas se calculan comparando ambos. Origen [20]

Basándonos en esta arquitectura, se podría crear un modelo capaz de traducir el holograma original a una máscara, o imagen semántica, consiguiendo clasificar y localizar las partículas en ella presentes, de una forma similar a la descrita en el estudio [21] anteriormente mencionado. El primer generador crearía esta imagen en un primer lugar, a partir de los hologramas sintéticos; posteriormente, obtendríamos de ella las posiciones de las partículas y su clase, que serían enviadas al sintetizador de hologramas para generar un holograma con esas características. El discriminador discerniría sobre la naturaleza de ambas imágenes, holograma sintético original y el fruto del ciclo; por último, las pérdidas se utilizarían para mejorar el primer generador y el discriminador. En cualquier caso, descarté la idea ante la dificultad que entrañaría, y cantidad de tiempo que sería necesario para implementar un modelo de estas características. Las redes basadas en CycleGANs son conocidas por ser extremadamente sensibles a los hiperparámetros utilizados, y habría requerido de semanas de experimentación hallar los hiperparámetros más óptimos.

3. Dataset: características, generación y flujo de entrada a la red convolucional

Como en todo proyecto de Machine Learning, la confección de un set de datos sobre los que entrenar la red (dataset) es primordial. Como se ha mencionado anteriormente, el dataset estará compuesto por hologramas sintéticos y diversas proyecciones y mapas derivados obtenidos del mismo. La versión definitiva de este dataset está compuesta por 2000 sets, cada uno con 2 imágenes de entrada (input) y 3 de salida u objetivos de entrenamiento (output o training target). Se puede acceder a este dataset a través del enlace proporcionado en el Anexo II. Los algoritmo que generan estas imágenes están escritos en MATLAB, y se pueden encontrar en el Anexo I. Para la pipeline de entrada de datos, al igual que para la programación del modelo, entrenamiento y generador de resultados, se ha utilizado Python con las librerías de Tensorflow [22] y la API Keras [23], y también pueden encontrarse en el Anexo I.

3.1. Características

Se generaron hologramas sintéticos acorde a ciertos parámetros de ruido, número de partículas de cada clase en cada posición del capilar o canal, tamaño de dichas partículas y número de planos en los que se localizan partículas, definidos por la distancia elegida entre ellos. En la generación de este dataset se usaron los valores de dichos parámetros presentes en la tabla o cuadro 1. Se registra información del capilar cuyos límites se sitúan a una distancia Z entre $z = 1{,}3mm$ y $z = 2{,}7mm$ del plano del holograma, que es el plano de enfoque de la cámara, con partículas generándose en planos a una distancia entre los mismos de $dz = 0{,}1mm$ dentro de esos límites definidos, a razón de 4 partículas por plano. Acorde con los datos en la tabla, podremos encontrar 60 partículas en cada holograma (30 de cada tipo). La elección de densidad de partículas tiene su origen en la búsqueda de un valor similar al presente en el holograma real proporcionado. Por último cabe mencionar la elección de resolución de salida. Los hologramas son generados con una resolución de 2560x2160 píxeles, pero por constricciones de espacio de almacenamiento y memoria gráfica disponible, son redimensionadas a 256x256 antes de guardarse.

Parámetro	**Valor**	**Unidades**
Distancia entre planos (dz)	0.1	mm
Diámetro de part. Blancas	30	μm
Diámetro de part. Magnéticas	10	μm
Número de hologramas	2000	
Amplitud del ruido	0.001	
Número de part. Blancas/plano	2	
Número de part. Magnéticas/plano	2	

Cuadro 1: Tabla de valores de los parámetros de generación de los hologramas del dataset

3.1.1. Inputs

El holograma (figura 6) es la primera de las imágenes que componen el set de entrada. Aunque técnicamente podríamos obtener la localización y clasificación de las partículas únicamente del holograma, aportando información adicional al sistema le podemos evitar tener que aprender completamente los procesos físicos relacionados con la formación de los hologramas. En este caso decidí incluir una proyección de fase máxima (figura 7), lo que permite aportar información relativa a la posición tridimensional de las partículas [12]. La información contenida en el holograma permite obtener la amplitud compleja (amplitud y fase) del objeto a diferentes distancias z de éste. Se ha reconstruido el holograma a diferentes planos z y en cada posición transversal se ha seleccionado el valor máximo de la fase en el conjunto de planos.

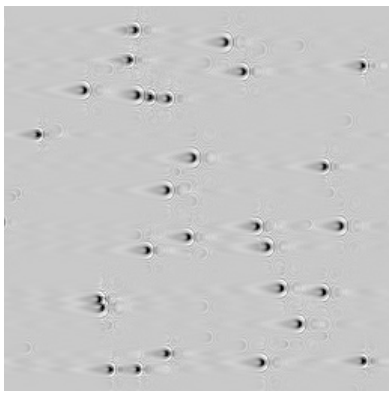

Figura 6: Holograma Sintético (amplitud compleja)

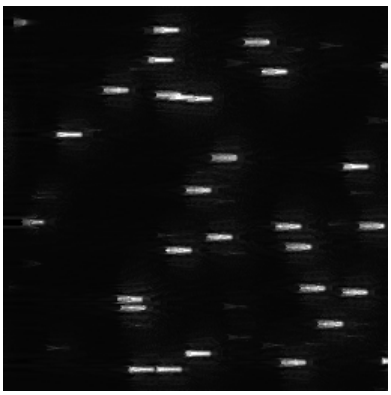

Figura 7: Proyección de Fase Máxima

3.1.2. Outputs

Basados en las localizaciones x,y,z de las partículas generadas, confeccioné otro algoritmo (Anexo I) que generó otros 2 mapas que codifican dicha posición 3D visualmente, adoptados de la solución en [12], y una máscara o imagen semántica que clasifica las partículas en magnéticas o blancas. El primero, un mapa de centroides que aporta la posición 2D de las partículas, en cuadrados de 2x2 píxeles (figura 8). El segundo, un mapa de localizaciones 3D de las mismas partículas, que aparte de codificar posicionalmente X e Y, codifica la posición Z en la intensidad de los píxeles, siendo las partículas de mayor Z blancas y proporcionalmente más oscuras cuanto más cerca están de la Z mínima (figura 9). Los cuadrados que representan cada partícula son de 4x4 píxeles, ya que usando cuadrados del tamaño utilizado para los centroides hacía que el modelo produjera predicciones sobre la intensidad de los píxeles poco precisas, e incluía mucho ruido de fondo. Por último, de nuevo basado en las localizaciones y tipo de partículas generadas, obtengo la máscara. Un ejemplo de la misma se puede observar en la figura 10, donde las circunferencias verdes representan partículas blancas y las rojas partículas magnéticas. Dado que la diferencia entre ambas es el tamaño, y por tanto la forma de sus proyecciones en distintas Z del holograma, decidí reflejarlo en la imagen semántica, para facilitar la tarea de segmentación. El tamaño de las partículas blancas y magnéticas en la imagen es proporcional a su tamaño real en el holograma sintético, que al ser uno de los parámetros de generación, es controlable. Estas 3 imágenes son usadas como output u objetivo de entrenamiento (training target) para la red, pudiendo extraer de las predicciones que surjan de la red la posición 3D y tipo de partícula de nuevo.

Figura 8: Mapa de Centroides 2D

Figura 9: Mapa de Localizaciones 3D

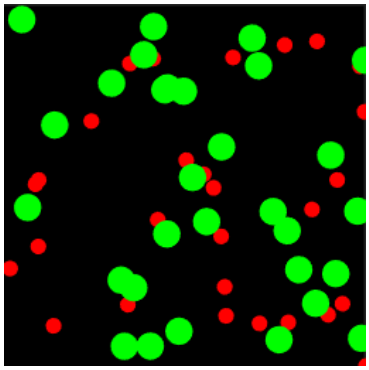

Figura 10: Imagen Semántica o Mapa Segmentado

3.2. Generación

El dataset definitivo está compuesto por 2000 sets como los descritos anteriormente. El código del generador del dataset se implementó en MATLAB, y ejecutado en un ordenador de sobremesa con un Intel Core i7 4790k@4.3GHz en 96h de ejecución ininterrumpida.

3.3. Flujo de datos de entrada a la red

Una vez generado, el dataset debe ser preparado para su introducción en la red. Originalmente intenté implementar esta pipeline de datos usando la API Tensorflow Datasets (tfds), pero al estar diseñada para los casos más sencillos (clasificadores imagen/etiqueta, y sistemas de 1 input y 1 output en general) era tremendamente inconveniente para la estructura de datos más compleja que tenemos aquí (2 inputs, 3 outputs, todos imágenes). Por tanto, decidí hacer uso de los objetos Generador que proporciona la API Keras.

Los ImageDataGenerator de Keras permiten cargar sets del directorio que contiene el dataset en grupos (batches) de un número definido por el usuario, aplicar el pre-procesamiento requerido y enviarlos directamente a la entrada de la red. Este pre-procesamiento tiene varias funciones:

3.3.1. Normalizar las imágenes

La normalización de las imágenes incluye reducir el tamaño de las mismas, en el caso de que usáramos un dataset con imágenes de mayor resolución a la que queremos utilizar de entrada, y cambiar el rango 0-255 de valor de sus píxeles a un float entre 0 y 1 capaz de ser procesado por la red.

3.3.2. Aplicar técnicas de Data Augmentation

Un posible problema que podría originarse durante el entrenamiento de este modelo es el overfitting. Un modelo puede hacer overfit cuando se entrena demasiado sobre unos mismos datos, obteniendo resultados muy buenos de pérdidas, habiendo aprendido a la perfección cómo hacer predicciones certeras con datos de su propio dataset, pero mostrando

un rendimiento mucho peor al ser presentado con datos nuevos. Esto es común cuando se entrenan modelos con datasets relativamente pequeños (y comparativamente 2000 sets no son demasiados). Para prevenir que la red se especialice en exceso, es habitual aplicar ciertas técnicas (recortes, color-shift, zoom, giros, aporte de ruido...) aleatoriamente a las imágenes de entrada para añadir variedad en el dataset. Esto se denomina Data Augmentation. La principal ventaja de usar objetos generador es poder aplicar Data Augmentation on-the-fly (a medida que se introducen los grupos de datos en la red, lo que permite que cada iteración el modelo se entrene en datos ligeramente distintos). Qué técnicas utilizar en cada caso, cuáles son más útiles, etc. es a veces una cuestión de experimentar con distintas configuraciones y comparar resultados. En este caso, habiendo probado con giros horizontales y verticales sin resultado, decidí limitarlo a añadir cierta cantidad de ruido Gaussiano aleatoriamente (dentro de unos parámetros controlados) a cada imagen de input que entra en la red. Este ruido es particularmente interesante al tratar con hologramas, los cuales en la realidad presentan siempre cierta cantidad de ruido, por lo que entrenar el modelo con ruido aleatorio debería ser beneficioso para el modelo, haciéndolo más robusto, de cara a predecir partículas tanto en hologramas sintéticos como en reales.

3.3.3. Separación en sets de entrenamiento y de validación

En cada iteración, el modelo es entrenado en una serie de datos del set de entrenamiento, y los valores de pérdidas validados en una serie de datos del set de validación, en los que el algoritmo no se entrena. Es interesante para poder observar cuándo el modelo hace overfitting (se observaría un decrecimiento de las pérdidas de entrenamiento, mientras las de validación se mantendrían estables), cuándo se queda atascado en mínimos locales de la función de pérdidas, etc. y poder corregir los hiperparámetros para mejorar su rendimiento en el futuro. De forma estándar, se suele reservar un 20 % del dataset total (validation split) para este propósito. Siguiendo esta distribución, este dataset contiene 1600 sets de entrenamiento, y 400 de validación. El generador es también el encargado de hacer esta separación previamente al comienzo del entrenamiento.

4. Arquitectura del Modelo

Dado que el objetivo es una tarea primordialmente de segmentación (obtener una máscara o imagen semántica/segmentada que determine posición y tipo de partículas), la elección de arquitectura se decantó por una basada en U-Net [11], tal y como he presentado en la sección 2. A continuación entraré en detalle en las características de esta arquitectura, la variante en la que me he basado y las diferencias con ésta que he introducido.

4.1. La base del modelo: U-NET

U-Net es una arquitectura originalmente concebida para el campo de la biomedicina con el propósito de segmentar imágenes, aunque hoy en día diversas variaciones de la misma se utilizan para infinidad de aplicaciones distintas. Entre sus numerosos usos se incluye, por ejemplo, generar máscaras que separen en una imagen de tejido celular cuáles son cancerosas y cuáles no [24]. La arquitectura básica de una red U-Net se puede observar en la figura 11; está compuesta esencialmente por 3 cosas:

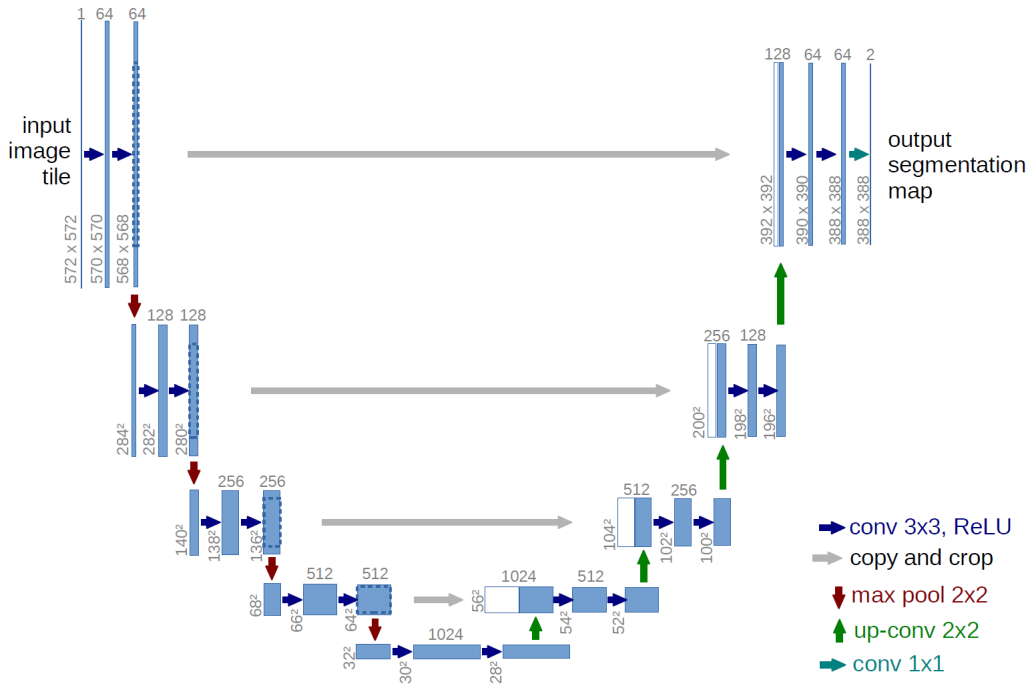

Figura 11: Arquitectura Básica del Modelo U-Net, según aparece en [11]

4.1.1. Capas

Como se ha mencionado en la Sección 1, en este tipo de redes la capa básica es la capa convolucional. En la arquitectura U-Net, se utilizan kernels de convolución de un tamaño 3x3 de forma constante a lo largo de la red (a diferencia de otras como AlexNet, que usa kernels de distintos tamaños, incrementalmente más pequeños a medida que se acerca a la salida[25]), variando únicamente el número de filtros aplicado. La razón por la que se usan kernels de 3x3 responde a cuestiones de tamaño de la red y rendimiento. Ofrecen unos resultados comparables y a la vez limitan el número de parámetros en la red, lo que se traduce en menor peso y entrenamiento e inferencia más rápidas. Cabe destacar que, al ser una red completamente convolucional (FCN o Fully Convolutional Network) no posee ninguna capa de neuronas densamente conectadas (densely, or fully connected layers), lo que implica que se pueden introducir como entrada imágenes de cualquier resolución, tan sólo cambiando el tamaño de la capa Input que espera el modelo. Esto es de extrema importancia en el campo de la biomedicina, ya que usar imágenes de gran resolución habitualmente posibilita obtener mejores resultados, aunque con un coste computacional y de memoria gráfica elevado.

Otros tipos de capas son también utilizadas:
1) Maximum Pooling: capa que calcula el valor máximo en cada parche del canal o mapa de características (feature map) sobre el que se aplica, permitiendo resaltar la característica más distintiva de cada uno, mientras reduce la altura y anchura (dimensionalidad) del mismo.
2) Up-Convolution: se trata de una convolución transpuesta, consiguiendo aumentar la dimensionalidad de los mapas de características sobre los que se aplica.

4.1.2. Estructura Encoder-Decoder

La clave de la arquitectura U-Net es su estructura de encoders-decoders. Los primeros van reduciendo paulatinamente la resolución de las imágenes (capas de maximum pooling) e incrementando el número de canales (capas de convolución). Esto consigue que el modelo sea capaz de detectar características incrementalmente más abstractas y localizadas. El decoder realiza el proceso contrario, reduciendo el número de canales (de nuevo capas de convolución) y aumentando la resolución (capas de up-convolution). Esto permite una excelente extracción de características y reduce los problemas de descompresión de las mismas de los que sufría AlexNet [25].

4.1.3. Copiar y Adjuntar

El proceso de adjuntar las salidas de un encoder a la entrada del decoder opuesto es a lo que se conoce como copiar y adjuntar (copy and append). Esto mejora enormemente la absorción de las características de la imagen de entrada, ya que en el proceso de aprendizaje algunos gradientes pueden degradarse y desaparecer al propagarse a través de todas las capas durante el entrenamiento.

4.2. Descripción, Características y Representación Gráfica

La variante de U-Net utilizada en este trabajo fue propuesta por [12]. Las diferencias entre esta arquitectura y la U-Net original son las siguientes:

4.2.1. Funciones de Activación

En vez de utilizar la estándar ReLU como función de activación entre capas, utiliza la función de activación Swish. Se ha demostrado que Swish ofrece mejores resultados que ReLU en modelos muy profundos como éste[26]. La capa de salida, en cambio, utiliza la sigmoidea como función de activación, ya que Swish puede dar como resultado valores negativos, incompatibles con el objetivo de entrenamiento que tenemos. La sigmoidea, en cambio, ofrece resultados en un rango de 0 a 1. Una comparativa directa entre ambas y otras funciones de activación utilizadas habitualmente se puede observar en la figura 12.

4.2.2. Batch Normalization

Consiste en normalizar los valores de salida de una capa anterior, restando la media de valores del batch o grupo de datos siendo procesados y dividiendo por la desviación estándar del batch. Como se explica en el estudio que introdujo la técnica[28], esta capa permite aumentar la velocidad, el rendimiento y la estabilidad de la red neuronal, aunque las razones por las cuáles lo consigue están aún bajo discusión[28][29][30].

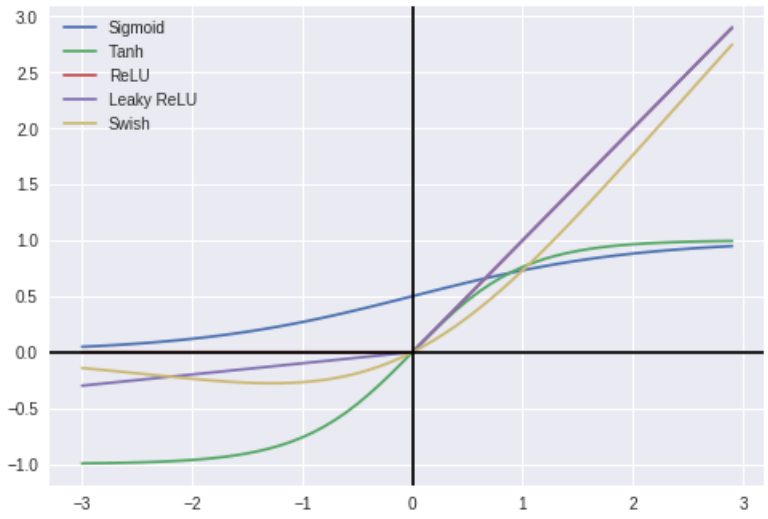

Figura 12: Representación de varias funciones de activación habitualmente utilizadas. Origen [27]

4.2.3. Conexiones residuales

Las conexiones residuales consisten en sumar los valores de entrada y salida de cada bloque encoder o decoder para aumentar la velocidad de entrenamiento, y reducir la probabilidad de que el entrenamiento se quede atrapado en un mínimo local[31].

4.3. Aportaciones a la Arquitectura

La arquitectura definitiva se puede observar en la figura 13. Dadas las diferencias entre inputs, outputs y características del dataset en general con el estudio citado anteriormente, introduje varios cambios. En primer lugar, el más evidente y de mayor impacto es el aumento de profundidad de la arquitectura: añadiendo un encoder y decoder enfrentados extra consigo que el canal de clasificación obtenga mejores resultados, especialmente en la clasificación de partículas magnéticas, al absorber características de más alto nivel. Además, al hacer uso de estructuras de copy & append y conexiones residuales el riesgo del desvanecimiento de gradientes (problema por el cual redes muy profundas pueden dejar de aprender al alcanzar el gradiente de la función de pérdidas valores muy pequeños, impidiendo actualizar correctamente los pesos) se reduce en gran medida. En segundo lugar, el tamaño de las partículas en el dataset utilizado por el estudio antes citado es varias veces mayor que el de las partículas magnéticas, por lo que aumenté la resolución de las imágenes de 128x128 a 256x256 para compensar parcialmente esta desventaja. Esto limitó, por otra

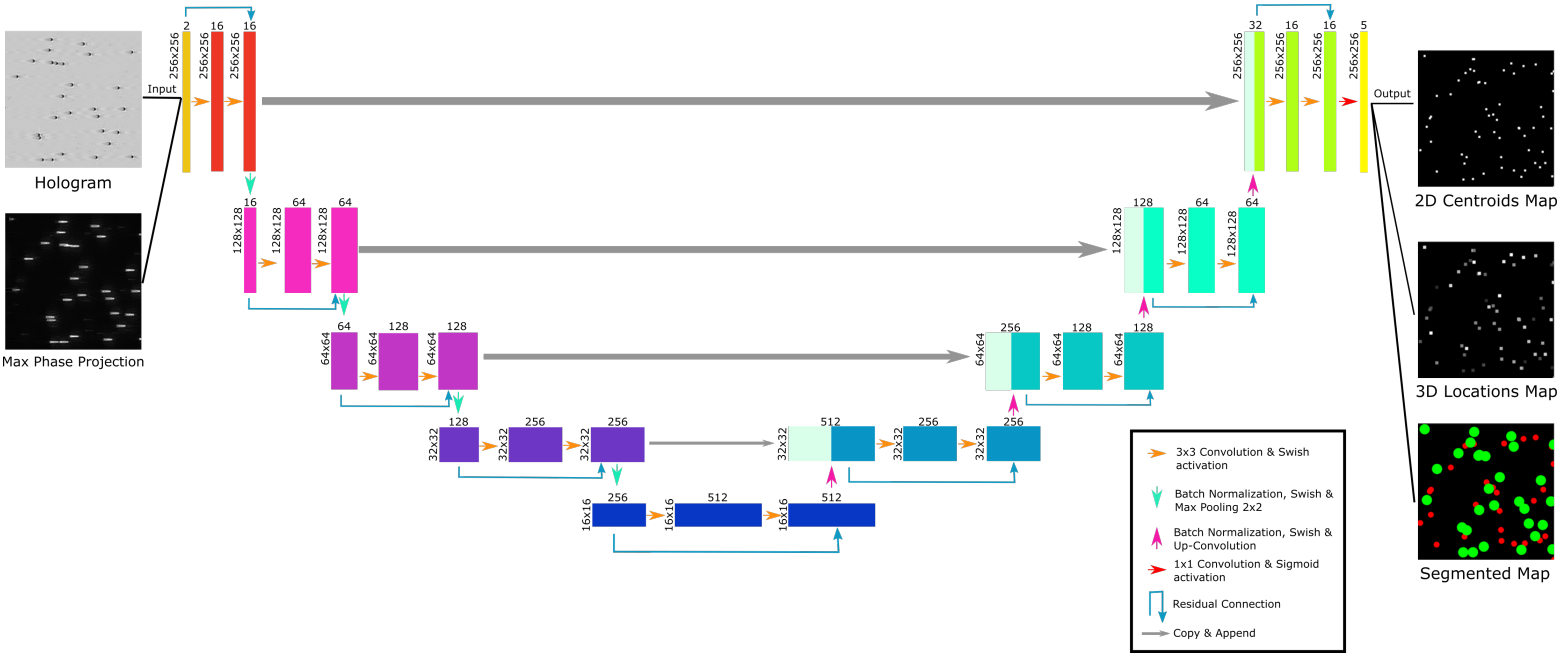

Figura 13: Arquitectura del Modelo. Dibujada con InkScape (Anexo I)

parte, el tamaño de los batches (grupos de imágenes que entran en la red al mismo tiempo durante el entrenamiento), por constricciones de memoria gráfica, pero los beneficios del aumento de resolución fueron mayores. Por último, elegí utilizar filtros de convolución de tamaño 1x1 para las salidas, frente a los 3x3 usados por el estudio antes citado. Este tipo de capas de convolución fueron propuestas como el equivalente a la fully conected layer para las redes FCN [10]. Permiten reducir la dimensionalidad (reducir el número de canales) usando el mínimo número de parámetros. Esto supuso una reducción del tiempo de procesamiento de cada época de entrenamiento y del peso de la red, sin coste en precisión aparente. Cabe mencionar que el número de canales en la salida del modelo es 5 debido a que el mapa segmentado utiliza 3 canales (al estar codificado en color), mientras que el de centroides y el de localizaciones, uno solo (al ser mapas en blanco y negro).

5. Entrenamiento de la Red: optimizador, funciones de pérdidas y selección de hiperparámetros

Una vez diseñado e implementado el modelo, y el dataset generado, es momento de entrenarlo. El entrenamiento de este modelo se ha realizado en su totalidad usando los recursos GPU gratuitos de Google Colab [32] (Anexo I). El modelo entrenado y el historial de pérdidas a lo largo de su entrenamiento se pueden encontrar en los enlaces del Anexo II. A continuación hablaré sobre todos los parámetros de interés, elecciones tomadas y las razones detrás de cada una de ellas.

5.1. Criterios de Selección de Hiperparámetros, Optimizador y Funciones de Pérdidas

Un hiperparámetro es cualquier parámetro cuyo valor se utiliza para controlar el proceso de aprendizaje del modelo. La selección del valor de los mismos es un proceso delicado y en parte basado en la experimentación, ya que dependiendo de las características de nuestro dataset y modelo, los valores óptimos varían enormemente. Después del proceso de experimentación descrito, presento en la tabla o cuadro 2 los valores de los hiperparámetros del modelo con mejores resultados de todos los entrenados.

Parámetro	**Valor**
Image Size	256x256
Batch Size	20
Epochs	250
Validation Split	0.2
Steps per Epoch	30
Validation Steps	20
Learning Rate	0.0005
Huber δ (imagen semántica)	0.2
Huber δ (mapa de localizaciones 3D)	0.02
Ruido Gaussiano (sobre 255)	10

Cuadro 2: Valores de los Hiperparámetros utilizados en el modelo con mejor rendimiento de todos los entrenados. Utiliza una función de pérdidas de Huber en ambos canales con distintos valores Delta, e incluye un generador de ruido en la entrada de datos de entrenamiento, que incrementa la variabilidad del dataset y mejora el rendimiento del modelo frente a sets de datos externos a los entrenados.

5.1.1. Optimizador

Utilizo el optimizador Adam[33], con un ritmo de aprendizaje (learning rate, el tamaño del paso que da la red hacia la dirección de mayor gradiente en la función de pérdidas) inicial de 0.0005. Este optimizador es el por defecto en muchos casos actualmente, siendo el que ofrece mejor rendimiento en la mayoría de redes convolucionales y es sencillo de implementar. Se trata de un optimizador adaptativo, lo que indica que estudia ciertos parámetros de la red, como las pérdidas, para ir variando el ritmo de aprendizaje cuando juzga conveniente, en un intento de alcanzar mínimos absolutos de la función de pérdidas siempre. La elección de learning rate inicial es fruto de la experimentación con distintos valores; usualmente, valores más pequeños ofrecen entrenamientos más estables y mayor probabilidad de alcanzar mínimos absolutos, pero obligan a entrenar el modelo en un número de épocas mayor. Cabe destacar que el rendimiento de este modelo es muy sensible a este parámetro, y cambios del orden de 0.0001 han llegado a afectar el rendimiento del mismo considerablemente, incluso desestabilizar su entrenamiento.

5.1.2. Funciones de Pérdidas

Dado que cada canal de salida tiene unas características distintas, utilizo funciones de pérdidas distintas para cada uno. De igual forma, debido a las características específicas de cada dataset, unas funciones de pérdidas pueden otorgar mejores resultados que otras. En este caso particular, salidas como el mapa de localizaciones 3D y el de centroides, con una clase muy dominante sobre la otra (el vacío, color negro, es el componente mayoritario del mapa, con las partículas siendo una flagrante minoría), obligan a hacer uso de funciones que castiguen especialmente los errores de predicción, al necesitar de una precisión y un nivel de detalle elevados. Funciones como MSE, Huber o la Generalized Dice Loss suelen ser muy indicadas en estos casos. Experimentando con las 3, terminé eligiendo las siguientes:

Para el mapa de centroides, uso el error cuadrático medio como función de pérdidas (MSE o Mean Squared Error) (ecuación 1, donde y_true es la imagen de salida real (ground truth) y y_pred la predicción del modelo). MSE eleva al cuadrado el error cometido, lo que lo hace especialmente sensible a datos dispersos. Dado que el objetivo de entrenamiento es una serie de datos dispersos y pequeños (grupos de 2x2 píxeles), es especialmente indicado para este caso. Algunas fuentes como [12] y [13] recomiendan regularizar este error con la variación total, sirviendo como un ligero filtro para el ruido de salida. Por desgracia, las

características de este dataset hacían que esta regularización, incluso en aportes pequeños, generara unas pérdidas inmensas (valores del orden de cientos o miles), exacerbando el punto débil de la función MSE, que es precisamente éste, lo que creaba unos gradientes muy grandes al comienzo del entrenamiento, que tardaban mucho tiempo en propagarse por toda la red. Esto influía profundamente en el tiempo de entrenamiento, que se incrementaba notablemente, y en muchas ocasiones desestabilizaba el proceso de entrenamiento, debido a la disparidad de valores de pérdidas entre distintos canales, haciendo que los resultados fueran pésimos. Por tanto, decidí prescindir de la misma.

$$\text{Pérdidas (MSE)} = \frac{1}{\text{n}} \sum_{i=1}^{n} (\text{y\_true} - \text{y\_pred})^2 \tag{1}$$

Para el mapa de localizaciones 3D, utilizo la función de pérdidas de Huber[34](ecuación 2). Esta función de pérdidas actúa como una MSE cuando el error es menor que un parámetro δ, y como una MAE (Mean Absolute Error o Error Absoluto Medio) para valores de error mayores. Se la conoce también como una MAE suavizada, por la forma cuadrática que toma la función con valores pequeños de error, cuando la MAE estándar tiene forma de pico (lineal). El valor δ es otro hiperparámetro más de esta red, y como tal su elección no es sencilla y se ha ido variando experimentalmente. El criterio seguido para la elección de este parámetro atiende a la distribución de clases y la dispersión de los datos presente en el dataset. Para casos como éste, es recomendable usar valores muy pequeños de δ, reduciendo el impacto que tiene el uso de la MSE en las primeras épocas (gradientes de error elevados), al no ser totalmente necesario usarla en todo el proceso de entrenamiento (a diferencia del mapa de centroides, en el que la dispersión y el desequilibrio entre clases es extrema y se beneficia de usar MSE desde el comienzo).

$$\text{Pérdidas (Huber)} = \begin{cases} \frac{1}{2}(\text{y\_true} - \text{y\_pred})^2, & |\text{y\_true} - \text{y\_pred}| \leq \delta \\ \delta(|\text{y\_true} - \text{y\_pred}| - \frac{1}{2}\delta), & \text{otherwise} \end{cases} \tag{2}$$

Para el mapa segmentado, experimenté con la Generalized Dice Loss [35], pero los resultados no fueron satisfactorios, probablemente debido a que las clases en la imagen semántica están mejor equilibradas que en los demás casos. Terminé implementando de nuevo la función de pérdidas de Huber, con resultados muy positivos. El valor δ, por otra parte, es distinto al elegido para el mapa de localizaciones 3D, reflejando la disparidad entre ambos. Al poseer un equilibrio mejor entre clases, con bloques de tamaño razonable

representando cada una, la δ pudo usar valores órdenes de magnitud mayores que la utilizada en el caso anterior.

5.1.3. Memoria gráfica disponible

Al haber entrenado el modelo en los servidores gratuitos de Google Colab, la memoria gráfica disponible (VRAM) se establece en aproximadamente 12GB. Este número limita el tamaño de las imágenes, el número de filtros máximo en el cuello de botella (entre el último encoder y el primer decoder) y el tamaño de los batches de datos. Habiendo fijado las características del modelo, y la resolución de entrada de las imágenes a 256x256 píxeles, maximicé el tamaño de los batches al máximo posible antes de sufrir excepciones OOM (Out Of Memory), siendo éste número 20.

5.1.4. Tiempo máximo de entrenamiento

De nuevo, al estar usando Colab, el tiempo de ejecución de cualquier script tiene un límite de 12h. De todas formas, dada la necesidad de iterar rápidamente para refinar los valores de los hiperparámetros, intenté limitar el entrenamiento a sesiones de 3-4h. Entre los hiperparámetros que dependen de este criterio, están el número de épocas, pasos por época y los pasos de validación por época. En lo referente a éste último parámetro, es recomendable, si los recursos lo permiten, que en cada época (grupo de iteraciones, al final del cual se reportan valores de pérdidas para ser visualizados) se recorra el set de validación entero, para enfrentar al algoritmo con los mismos datos en todas las épocas, haciendo el valor de pérdidas de validación más estable y fiable. En cuanto al número de épocas, un número excesivo de las mismas tampoco es recomendable, ya que llegado a cierto punto el modelo deja de aprender y el único efecto que esto puede producir es overfitting.

5.1.5. Ruido

La cantidad de ruido introducido en las imágenes de entrada es un parámetro cuyo valor se debe íntegramente al proceso de experimentación. Aportando demasiado el rendimiento del sistema sufría al intentar hacer predicciones, especialmente de partículas magnéticas. Demasiado poco, y no era capaz de generalizar lo suficiente, además de probablemente afectar a su rendimiento en hologramas reales.

6. Resultados del proceso de entrenamiento y evaluación del modelo entrenado

En esta sección presento los resultados obtenidos del modelo con mayor rendimiento. Los criterios seguidos para evaluar el rendimiento de este modelo han sido las pérdidas y diferentes comparativas visuales. Todas las predicciones presentadas en esta sección han sido generadas tomando las imágenes de entrada de un dataset nuevo (Test Dataset) integrado por sets de imágenes de input y output nuevas, sobre las que por tanto el modelo ni se ha entrenado, ni se ha validado, conservando la independencia de los resultados. Este dataset está compuesto por 20 sets de imágenes, de los cuales 10 poseen una concentración de partículas equivalente a la del dataset de entrenamiento, mientras que las otras 10 poseen una concentración de partículas del doble. Esto servirá para evaluar la robustez del modelo ante la variable concentración de partículas presente en la realidad. Se puede acceder a este dataset por medio del enlace presente en el Anexo II. El algoritmo utilizado en esta sección se puede consultar en el Anexo I.

6.1. Resultados del Proceso de Entrenamiento

En primer lugar, analizo los resultados de pérdidas obtenidos a lo largo del entrenamiento. Éstas se pueden observar en la figura 14. A medida que el modelo se vuelve más preciso, el error cometido se minimiza, reduciéndose por tanto las pérdidas. Es una medida también del aprendizaje del modelo: si durante varias épocas las pérdidas se mantienen estables, es probable que el modelo haya dejado de aprender y se haya quedado estancado en un mínimo local de la función de pérdidas. Como se puede observar en este caso, el modelo ha ido reduciendo progresivamente sus pérdidas de entrenamiento hasta el final. Por contra, las pérdidas de validación se han mantenido casi invariables. Esto es un claro indicio de que el modelo ha incurrido en overfitting. Al sobreentrenar el modelo en un dataset específico, su capacidad para hacer predicciones sobre el mismo continúa aumentando, pero a costa de su capacidad de generalizar. Por lo que al ser presentado con datos nuevos, externos, su rendimiento puede ser mucho peor del esperado. En este caso, la similitud entre datos nuevos (validación y test) y de entrenamiento ha hecho que no se produzca una reducción de sus capacidades, sino que se haya simplemente estancado, pero sería algo a tener muy en cuenta de ser esta arquitectura utilizada en un caso real (como los vasos sanguíneos anteriormente mencionados, con un mayor número de variables y tipos de partículas).

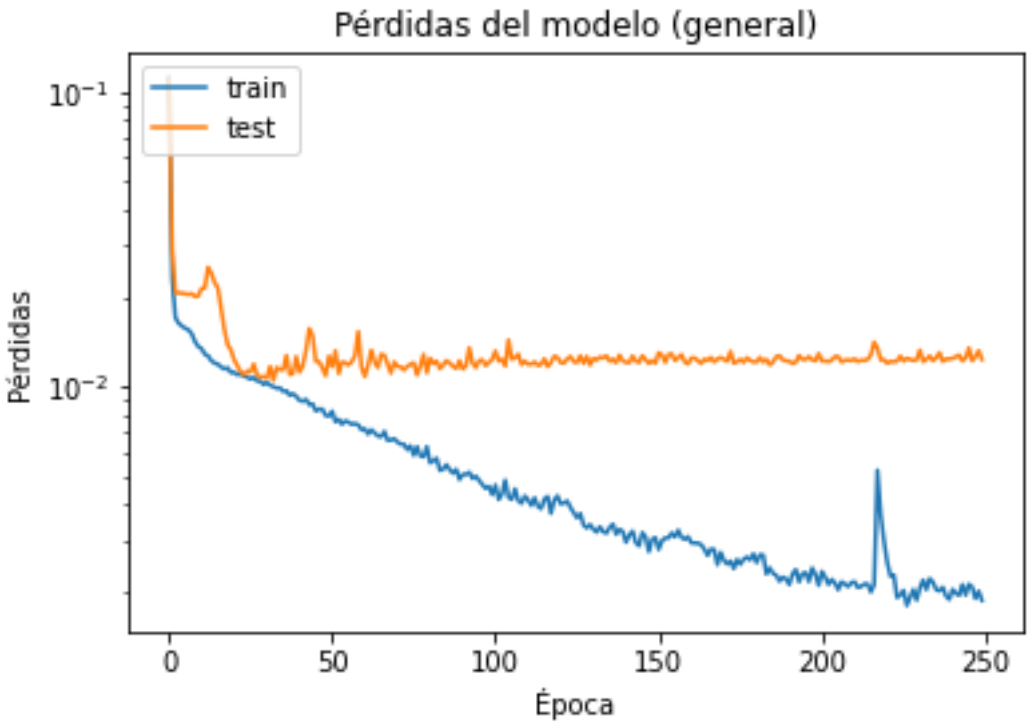

Figura 14: Historial de Pérdidas del Modelo durante su entrenamiento (250 épocas). Nótese que la escala es logarítmica, para visualizar mejor la evolución en las etapas finales del entrenamiento

Hay diversas formas de combatir el overfitting. Algunas ya han sido implementadas en este trabajo, como las técnicas de Data Augmentation (sección 3), aunque la más clara y probablemente efectiva sería obtener un dataset más variado (concentración de partículas variable) y de mayor tamaño.

6.2. Evaluación del Modelo

En segundo lugar, evaluamos visualmente la calidad de las predicciones al comparar directamente predicción con la ground truth. En la figura 15, se pueden observar las imágenes introducidas en el modelo (fila superior), mientras que las filas inferiores permiten una comparativa directa entre las predicciones que el modelo hace a partir del Input (derecha), con sus respectivas imágenes objetivo (ground truth, izquierda). El holograma y su correspondiente proyección de fase máxima se han tomado aleatoriamente del Test Dataset. Los resultados varían levemente de un holograma a otro, pero de forma clara se pueden sacar las siguientes conclusiones:

1) La clasificación de las partículas es muy buena, en especial de las partículas de mayor tamaño. Esto es un indicio muy positivo, ya que al extender este experimento a todas las partículas presentes en los microcanales estudiados (vasos sanguíneos, como se ejemplificaba en la introducción a este Trabajo) esta misma arquitectura podría clasificar dichas partículas de distintos tamaños con mucha precisión, habiendo logrado el objetivo principal de este Trabajo

2) La localización de las partículas es razonablemente precisa. En lo que se refiere a localización en el plano XY (mapa de centroides), la precisión es magnífica, aunque algunas partículas magnéticas terminan escapándose, especialmente si se encuentran muy próximas o sobreponiéndose a una blanca. En cuanto a la precisión en profundidad, en el eje Z (mapa de localizaciones 3D), es buena para partículas blancas pero pobre para magnéticas. La dificultad que presentan éstas es mucho mayor, debido a su pequeño tamaño, que hace que en muchos casos estas partículas sean confundidas por ruido. Incluso inspeccionando visualmente hologramas reales y sintéticos, son prácticamente invisibles al ojo humano, ocupando unos pocos píxeles y siendo confundidas fácilmente por artefactos espurios, como ruido. Los resultados en este sentido no son comparables con otros localizadores de partículas basados en Machine Learning como [12], ya que las partículas usadas en este trabajo son notablemente más pequeñas que las utilizadas por Shao et. al. en citado estudio.

3) El modelo es muy robusto, incluso cuando es presentado con hologramas de una concentración de partículas notablemente superior, aun no habiendo sido entrenado en concentraciones variables, como se puede observar en la figura 16. La localización y clasificación de partículas blancas sigue siendo muy buena, aunque se degrada más todavía su rendimiento en partículas magnéticas. Esto refuerza la idea de que el modelo tiene mucho potencial para generalizar resultados, y entrenado en casos reales o sintetizadores equivalentes a un caso real podría rendir a un nivel muy elevado.

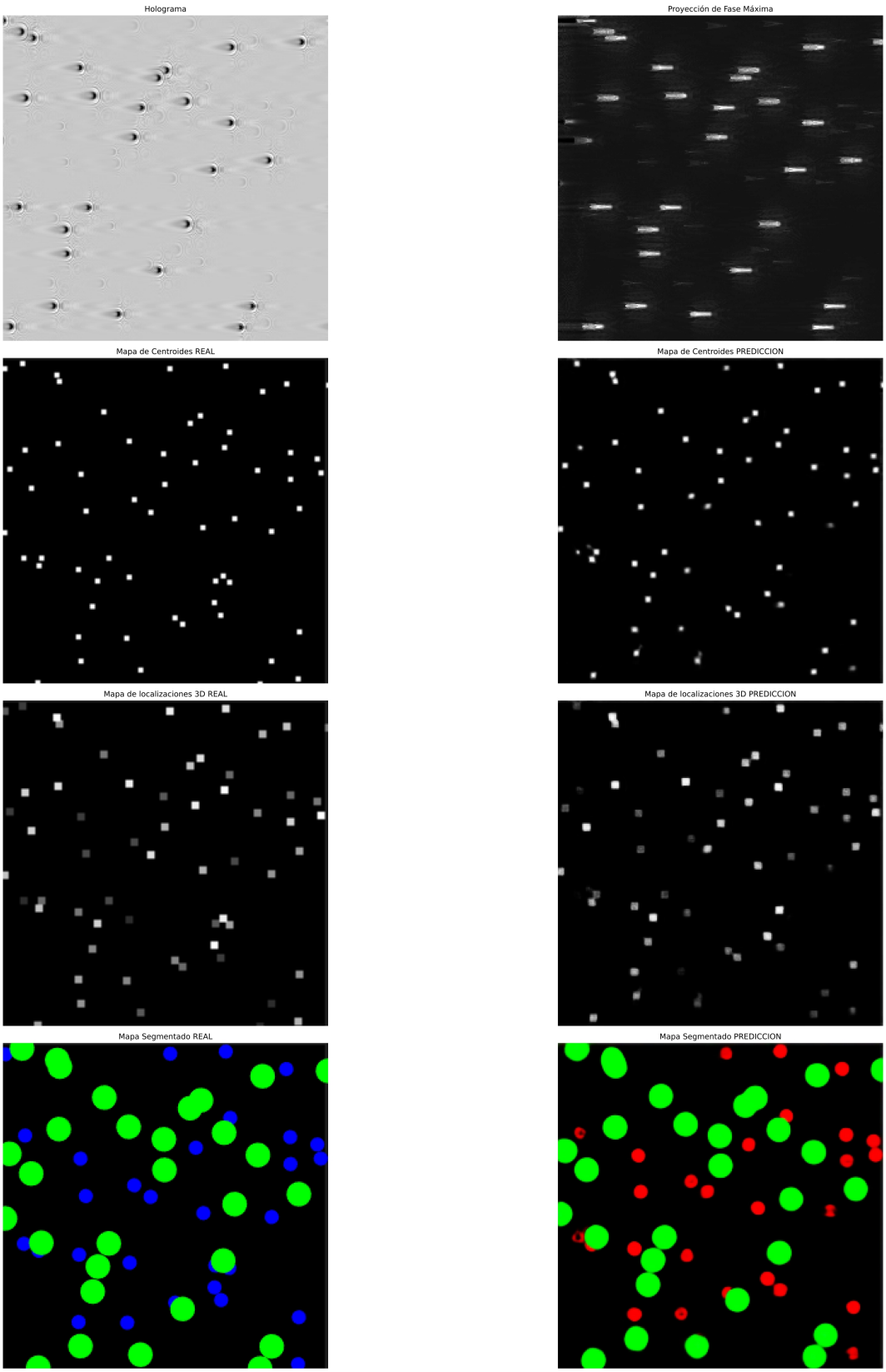

Figura 15: Predicciones del modelo entrenado en 250 épocas. Concentración de 60 partículas/holograma

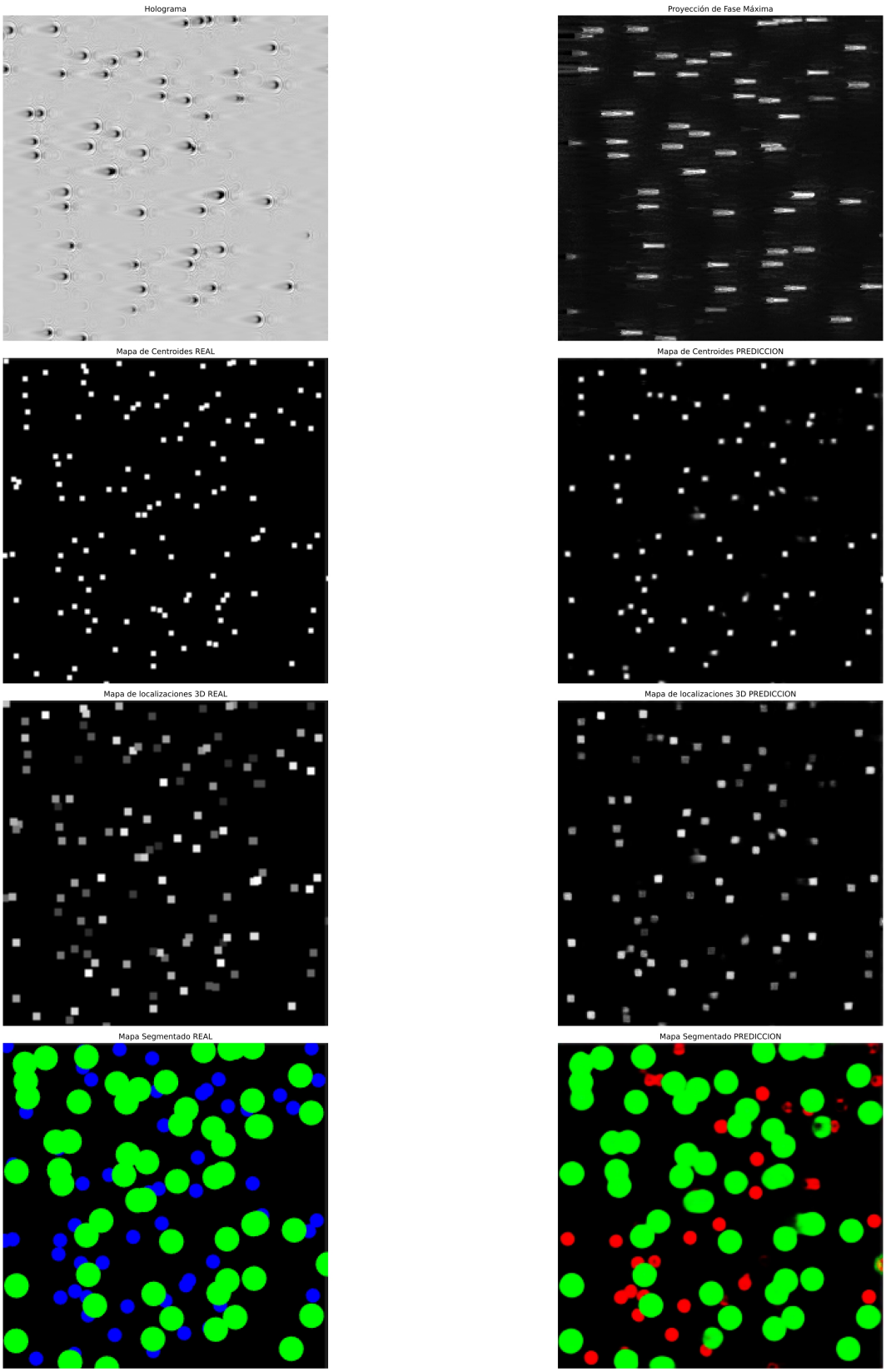

Figura 16: Predicciones del modelo entrenado en 250 épocas. Concentración de 120 partículas/holograma

7. Áreas de mejora y alternativas

Hay ciertas áreas en las que es probable que el algoritmo pueda obtener mejoras de rendimiento. La mayoría de ellas a menudo conllevan mayores costes computacionales o una inversión de tiempo mayor en el desarrollo del mismo. Menciono algunas a continuación.

7.1. Dataset de mayor tamaño

Como he demostrado en la sección 6, se ha demostrado que el modelo incurre en overfitting relativamente rápido en el proceso de entrenamiento. Por lo tanto, es probable que disponiendo de un dataset de mayor tamaño se pueda conseguir un modelo más robusto. Otra forma de compensar esta falta de datos es consiguiendo implementar con éxito más técnicas de data augmentation que permitan añadir más variedad al dataset.

7.2. Entrenamiento con datasets de diversa concentración de partículas

Entrenar reiteradamente el mismo modelo en datasets con distinto nivel de concentración de partículas, como sugiere [12], habría mejorado el rendimiento del modelo ante hologramas de densidad de partículas variable, como serán los utilizados en la realidad. Esto era inviable dentro del alcance de este Trabajo, ya que cada dataset (2000 hologramas), con los recursos disponibles, necesita de 96h para generarse. El estudio citado utiliza hasta 13 datasets de variados niveles de concentración de partículas.

7.3. Testeo de distintos valores para los hiperparámetros

Los valores escogidos de hiperparámetros como el δ de Huber que utilizan los canales de localización 3D y la imagen semántica, han sido fruto en gran parte de la experimentación. Es plausible que existan valores más óptimos de los mismos, y que con ciclos más cortos de entrenamiento se hubieran podido descubrir. Más razonable habría sido implementar funciones de pérdidas con parámetros variables, como la función de Huber adaptativa propuesta en [36], que es capaz de optimizarse por sí misma durante el entrenamiento del modelo. Es un algoritmo relativamente complejo y no está implementado en Keras de forma

nativa, por lo que habría requerido de bastante tiempo adaptarlo.

En cuanto a otros parámetros como el tamaño de los batches, la resolución de las imágenes de entrada y salida y la profundidad de la red, son dependientes de la cantidad de memoria gráfica disponible; haber tenido una cantidad mayor de VRAM a mi disposición habría permitido aumentar estos parámetros, con probables beneficios a la calidad de las predicciones como resultado.

7.4. Optimización del tiempo de entrenamiento e inferencia con el Profiler

La herramienta Profiler de Tensorflow permite controlar aspectos de la ejecución del algoritmo como el momento de carga o preprocessing de un batch o los recursos utilizados por cada tarea, entre otros. Esto haría los procesos de entrenamiento más eficientes, pero es un proceso que requiere de mucho tiempo de refinamiento por lo que se ha descartado.

7.5. Alternativas al modelo implementado: CycleGAN

La alternativa descartada en la fase de investigación que habría empleado una versión modificada de una arquitectura CycleGAN podría haberse perseguido, siendo ésta una técnica más novedosa y con posiblemente mayor potencial. Es posible que mejores resultados se hubieran podido obtener de haber sido desarrollada, dados sus recientes éxitos en otras tareas de segmentación como [21], donde consiguieron al mismo tiempo reducir la necesidad de datasets inmensos, pudiendo obtener resultados equiparables con una fracción de los datos. No obstante, las demandas de recursos y tiempo necesarios para conseguir alcanzar resultados razonables habrían sido muy elevadas, y es por lo que, aun reconociendo su potencial, se ha descartado para este Trabajo.

8. Conclusiones

A lo largo de estas páginas se ha analizado la posibilidad de usar técnicas de machine learning para la localización y clasificación de partículas de diferentes tamaños, suspendidas en fluidos presentes en microcanales, a partir de hologramas digitales. Se han introducido con éxito variaciones sobre modelos establecidos en el campo de la segmentación de objetos en general, ampliando su utilidad para cubrir los requisitos de la tarea propuesta. Entre las más notables se encuentran el uso de una red de mayor profundidad, imágenes de mayor resolución, una capa de salida más eficiente, y un esquema de inputs y outputs personalizado. Con esto se ha conseguido crear un modelo capaz de, al mismo tiempo, localizar y clasificar partículas, mientras que esfuerzos similares como los de [12] y [13] o sólo localizaban (los primeros), o sólo clasificaban (los segundos). Además, ha sido conseguido con un reducido número de entradas (un plano menos de input que [12], y sin tener que utilizar proyecciones en distintas Z como [13]), con la consiguiente reducción de tamaño y tiempo de entrenamiento necesarios para el modelo. Más aún, es capaz de localizar y clasificar partículas más pequeñas que las utilizadas en los citados estudios, aunque con un nivel de precisión mucho más modesto.

Tal y como he descrito en la Sección 6, el modelo ha sido capaz de clasificar de forma muy precisa ambos tipos de partículas, y localizarlas con una precisión muy razonable (especialmente en el caso de partículas blancas). Además, se ha demostrado la robustez del modelo ante hologramas de variable densidad de partículas, más representativos del caso real, incluso sin haber sido entrenado en un dataset específico que incorporara dicha densidad de partículas variable.

Los resultados de este TFG son, por tanto, muy positivos, habiendo cumplido todos los objetivos propuestos, y demuestran el potencial que tienen los algoritmos de machine learning en el área de la holografía digital aplicada a la biomedicina. Dichos resultados confirman que las redes neuronales son una buena alternativa a los algoritmos tradicionales en este campo. La arquitectura desarrollada, y la tendencia que ésta representa (de nuevo hacia modelos más profundos, habiendo compensado sus debilidades iniciales por medio del uso de técnicas como el copy & crop/append y las conexiones residuales) podrían servir de base a futuros modelos más complejos y específicos, a medida que nos acercáramos a la realidad presente en, por ejemplo, vasos sanguíneos, siendo ésta una de las aplicaciones de más impacto que podría tener.

Las vías de mejora de este algoritmo, para incrementar el rendimiento en esta aplicación en concreto, han sido estudiadas en la sección 7, donde también he propuesto una novedosa arquitectura alternativa, basada en las redes CycleGAN, para afrontar este problema. En cuanto a los siguientes pasos a dar para convertir el modelo presentado en este trabajo en una aplicación útil en la biomedicina real, el primero sería entrenar la red con un dataset proveniente de un sintetizador que modelara la realidad en la que se quiere aplicar de forma mucho más precisa, con el objetivo de ser probado en hologramas reales para corroborar su potencial. Es entonces cuando se podrían trazar comparativas directas con el resto de técnicas (basadas en machine learning o no) que se utilizan o podrían utilizarse para esta aplicación.

En una nota más personal, he encontrado el trabajo altamente satisfactorio. Me ha otorgado una visión mucho más completa y profunda de todo lo relacionado con las redes convolucionales, especialmente al haberme permitido desarrollar una aplicación en su totalidad, desde la elección y generación de un dataset, hasta la interpretación de los resultados de un proceso de entrenamiento. Mis habilidades en MATLAB, Python, Tensorflow y LaTeX se han incrementado exponencialmente; habilidades que estoy seguro me serán de utilidad en mi futuro académico y laboral, dado el altísimo nivel de transferencia que estos conocimientos tienen a la industria, donde aplicaciones basadas en machine learning no paran de aparecer día a día. No obstante, he de admitir que el desarrollo de este trabajo ha significado una lucha constante contra la falta de recursos computacionales, convirtiéndose en el mayor criterio de elección entre unos parámetros o unas estrategias y otras; y de igual forma, una carrera contra el tiempo, dada la complejidad de muchos de los temas tratados, la cantidad de fuentes consultadas y las largas esperas entre entrenamientos en la fase de experimentación. De cualquier forma, quiero recalcar el impacto positivo que este trabajo ha tenido en mi formación y mi satisfacción ante los resultados obtenidos.

9. Bibliografía

[1] Marina Climente, Julia Salazar, Virginia Palero-Díaz, and M. Grandes. Matched filter applied to discriminate particles with different sizes in biological flows. page 12, 06 2019. doi: 10.1117/12.2525833.

[2] Vittorio Bianco, P. Memmolo, Pierluigi Carcagnì, Francesco Merola, Melania Paturzo, Cosimo Distante, and Pietro Ferraro. Microplastic identification via holographic imaging and machine learning. *Advanced Intelligent Systems*, 2, 12 2019. doi: 10.1002/aisy.201900153.

[3] YoungJu Jo, Sangjin Park, JaeHwang Jung, Jonghee Yoon, Hosung Joo, Min-hyeok Kim, Suk-Jo Kang, Myung Choi, Sang Lee, and YongKeun Park. Holographic deep learning for rapid optical screening of anthrax spores. *Science Advances*, 3:e1700606, 08 2017. doi: 10.1126/sciadv.1700606.

[4] Faliu Yi, Inkyu Moon, and Bahram Javidi. Automated red blood cells extraction from holographic images using fully convolutional neural networks. *Biomedical Optics Express*, 8:4466, 10 2017. doi: 10.1364/BOE.8.004466.

[5] Tarek Nafee, C. Michael Gibson, Ryan Travis, Megan K. Yee, Mathieu Kerneis, Gerald Chi, Fahad Alkhalfan, Adrián Fernández Hernández, Russell D. Hull, Ander T. Cohen, Robert A. Harrington, and Samuel Z. Goldhaber. Machine learning to predict venous thrombosis in acutely ill medical patients. *Research and Practice in Thrombosis and Haemostasis*, 4:230 – 237, 2020.

[6] Somoshree Sengupta and Vamsi K. Balla. A review on the use of magnetic fields and ultrasound for non-invasive cancer treatment. *Journal of Advanced Research*, 14:97 – 111, 2018. ISSN 2090-1232. doi: https://doi.org/10.1016/j.jare.2018.06.003. URL `http://www.sciencedirect.com/science/article/pii/S2090123218300778`.

[7] U. Schnars and W. Jueptner. Digital holography: Digital hologram recording, numerical reconstruction, and related techniques. *Digital Holography: Digital Hologram Recording, Numerical Reconstruction, and Related Techniques*, pages 1–5, 01 2005. doi: 10.1007/b138284.

[8] Labelme, graphical image anotation tool. URL `https://github.com/wkentaro/labelme`.

[9] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[10] Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully Convolutional Networks for Semantic Segmentation. *arXiv e-prints*, art. arXiv:1411.4038, November 2014.

[11] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-Net: Convolutional Networks for Biomedical Image Segmentation. *arXiv e-prints*, art. arXiv:1505.04597, May 2015.

[12] Siyao Shao, Kevin Mallery, S. Santosh Kumar, and Jiarong Hong. Machine learning holography for 3D particle field imaging. *Optics Express*, 28(3):2987, February 2020. doi: 10.1364/OE.379480.

[13] Siyao Shao, Kevin Mallery, and Jiarong Hong. Machine learning holography for measuring 3D particle size distribution. *arXiv e-prints*, art. arXiv:1912.13036, December 2019.

[14] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative Adversarial Networks. *arXiv e-prints*, art. arXiv:1406.2661, June 2014.

[15] Overview of gan structure.(accessed: 10.03.2020). URL https://developers.google.com/machine-learning/gan/gan_structure?hl=es-419.

[16] Tero Karras, Timo Aila, Samuli Laine, and Jaakko Lehtinen. Progressive Growing of GANs for Improved Quality, Stability, and Variation. *arXiv e-prints*, art. arXiv:1710.10196, October 2017.

[17] Phillip Isola, Jun-Yan Zhu, Tinghui Zhou, and Alexei A. Efros. Image-to-Image Translation with Conditional Adversarial Networks. *arXiv e-prints*, art. arXiv:1611.07004, November 2016.

[18] Ting-Chun Wang, Ming-Yu Liu, Jun-Yan Zhu, Andrew Tao, Jan Kautz, and Bryan Catanzaro. High-Resolution Image Synthesis and Semantic Manipulation with Conditional GANs. *arXiv e-prints*, art. arXiv:1711.11585, November 2017.

[19] Christian Ledig, Lucas Theis, Ferenc Huszar, Jose Caballero, Andrew Cunningham, Alejandro Acosta, Andrew Aitken, Alykhan Tejani, Johannes Totz, Zehan Wang, and Wenzhe Shi. Photo-Realistic Single Image Super-Resolution Using a Generative Adversarial Network. *arXiv e-prints*, art. arXiv:1609.04802, September 2016.

[20] Jun-Yan Zhu, Taesung Park, Phillip Isola, and Alexei A. Efros. Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks. *arXiv e-prints*, art. arXiv:1703.10593, March 2017.

[21] Arnab Kumar Mondal, Aniket Agarwal, Jose Dolz, and Christian Desrosiers. Revisiting CycleGAN for semi-supervised segmentation. *arXiv e-prints*, art. arXiv:1908.11569, August 2019.

[22] Tensorflow, an end-to-end open source machine learning platform. URL `https://www.tensorflow.org/`.

[23] Keras, the python deep learning api. URL `https://keras.io/`.

[24] Zhemin Zhuang, Nan Li, Alex Noel Joseph Raj, Vijayalakshmi Mahesh, and Shunmin Qiu. An rdau-net model for lesion segmentation in breast ultrasound images. *PLOS ONE*, 14: e0221535, 08 2019. doi: 10.1371/journal.pone.0221535.

[25] Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton. Imagenet classification with deep convolutional neural networks. *Neural Information Processing Systems*, 25, 01 2012. doi: 10.1145/3065386.

[26] Prajit Ramachandran, Barret Zoph, and Quoc V. Le. Searching for Activation Functions. *arXiv e-prints*, art. arXiv:1710.05941, October 2017.

[27] Comparison of activation functions for deep neural networks (accessed: 24.06.2020). URL `https://towardsdatascience.com/comparison-of-activation-functions-for-deep-neural-networks-706ac4284c8a`.

[28] Sergey Ioffe and Christian Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. *arXiv e-prints*, art. arXiv:1502.03167, February 2015.

[29] Shibani Santurkar, Dimitris Tsipras, Andrew Ilyas, and Aleksander Madry. How Does Batch Normalization Help Optimization? *arXiv e-prints*, art. arXiv:1805.11604, May 2018.

[30] Jonas Kohler, Hadi Daneshmand, Aurelien Lucchi, Ming Zhou, Klaus Neymeyr, and Thomas Hofmann. Exponential convergence rates for Batch Normalization: The power of length-direction decoupling in non-convex optimization. *arXiv e-prints*, art. arXiv:1805.10694, May 2018.

[31] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. *arXiv e-prints*, art. arXiv:1512.03385, December 2015.

[32] Google colab. URL `https://colab.research.google.com/`.

[33] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. *arXiv e-prints*, art. arXiv:1412.6980, December 2014.

[34] Peter J. Huber. Robust estimation of a location parameter. *Ann. Math. Statist.*, 35(1): 73–101, 03 1964. doi: 10.1214/aoms/1177703732. URL `https://doi.org/10.1214/aoms/1177703732`.

[35] William Crum, Oscar Camara, and Derek Hill. Generalized overlap measures for evaluation and validation in medical image analysis. *IEEE transactions on medical imaging*, 25:1451–61, 12 2006. doi: 10.1109/TMI.2006.880587.

[36] Jacopo Cavazza and Vittorio Murino. Active Regression with Adaptive Huber Loss. *arXiv e-prints*, art. arXiv:1606.01568, June 2016.

Anexo I: Software, herramientas utilizadas y algoritmos desarrollados

Jorge Condor Lacambra

25 de Junio 2020

Grado en Ingeniería Electrónica y Automática

I.1. Herramientas

Las herramientas elegidas en este trabajo son las siguientes:

1) Generación del Dataset. El dataset fue generado usando MATLAB, el lenguaje en el que estaba implementado el sintetizador de hologramas. Confeccioné dos algoritmos distintos: uno que utiliza el sintetizador para generar los hologramas, proyectarlos en los planos, calcular la proyección de fase máxima y guardar la información de posición y tipo de partículas en una tabla Excel; y el otro que toma esos datos de la tabla en Excel y genera los distintos mapas para localizar y clasificar las partículas decritos en la Sección 3.

2) Mapa segmentado del holograma original. Inicialmente creé un mapa semántico de una de las proyecciones del holograma original, el cual se puede observar superpuesto a dicha proyección en la Sección 1. Para ello utilicé una herramienta llamada Labelme (https://github.com/wkentaro/labelme).

3) Algoritmos de Machine Learning. Todo lo referente a los distintos algoritmos utilizados en el procesado de los datos, el código de la arquitectura, entrenamiento de la red, y obtención de resultados de pérdidas y predicciones fueron programados en Python, usando las librerías de Tensorflow y su frontend API Keras. Se eligió Python por permitir iterar muy rápidamente, y por la gran variedad de información y recursos disponibles, al ser el lenguaje de programación más popular en la actualidad. En cuanto a la decisión de utilizar Tensorflow y no otras librerías de machine learning como PyTorch o Caffe2, simplemente se debió a preferencia personal, ya que poseía experiencia previa con la librería. Además implementar modelos customizados es una tarea muy cómoda con Keras, stackeando objetos del tipo layer uno sobre otro. Aparte, también he usado otras librerías de Python comunes como matplotlib.pyplot (plots), numpy (cálculos numéricos) y pickle (guardar/cargar datos, usado para guardar y cargar el historial de pérdidas), entre otras.

4) Esquema de la arquitectura. Para realizar el esquema visual de la arquitectura utilicé InkScape, una herramienta de dibujo vectorial fácil de usar y open source.

5) Entrenamiento de la red. El entrenamiento se ha realizado íntegramente usando los servicios gratuitos de Google Colab, los cuales incluyen el uso de una GPU para acelerar el entrenamiento de redes neuronales, con ciertas limitaciones. Los scripts no pueden estar más de 12h funcionando, y te ofrecen hasta 12GB de RAM y una cantidad más o menos equivalente de VRAM. Es éste último parámetro el que hizo que me decantara por usar Colab, al poseer 3 veces más memoria gráfica que los servicios que tenía a mi disposición.

6) Memoria. La memoria ha sido escrita en su totalidad en LaTeX, usando la plataforma online gratuita Overleaf. LaTeX es muy conveniente a la hora de escribir textos científicos por las

facilidades que ofrece para numerar figuras, gestionar bibliografía, incluir índices, formateo de ecuaciones...

I.2. Algoritmos Desarrollados

Incluyo a continuación los algoritmos desarrollados durante este Trabajo. Adicionalmente, se puede acceder a los archivos originales de los mismos con el siguiente enlace: `https://drive.google.com/drive/folders/1FjtmXAgqPwMYaHCScK8CGXAoSPw0FmHq?usp=sharing`.

El orden de aparición de los mismos es el siguiente:

I.2.1. Generador del Dataset: Hologramas y Proyecciones de Fase Máxima

I.2.2. Generador del Dataset: Mapa de Centroides, Mapa de Localizaciones 3D y Mapa Segmentado

I.2.3. Algoritmos de Machine Learning: Input Pipeline, Modelo, Entrenamiento

I.2.4. Algoritmos de Machine Learning: Resultados de Pérdidas y Predicciones

```
%Generador del dataset para posterior entrenamiento
NumHolo = 2000; %número de hologramas generados
dz = 0.1; %distancia entre planos
Ruido = 0.001; %amplitud del ruido

Diam1 = 30; %diámetro de blancas
Diam2 = 10; %diámetro de magnéticas

NumPart1 = 2;
NumPart2 = 2;
NumPartTotal=NumPart1+NumPart2;

output_size = [256 256]; %tamaño de las figuras de salida
resolution = 256; %dpi de la imagen

for H = 1:NumHolo

    close all
    figure;

    set(gca,'visible','off')
    set(gcf,'paperunits','inches','paperposition',[0 0 output_size/resolution]);
    %genero el holograma
    [Holograma, Posiciones1,Posiciones2, R, dz] = Generar_holograma(Diam1,↙
NumPart1,Diam2,NumPart2,dz,Ruido);
    nombreHolo=strcat('Hologram',num2str(H));
    pathHolo='F:\Universidad\Universidad\TFG\TFG Jorge Condor↙
Lacambra\Dataset\Hologram\Hologram';
    title('');
    set(gca,'visible','off')
    grid off
    box off
    axis off
    set(gcf,'NumberTitle','off'); %elimina el título del plot
    set(gca,'color','k'); %color de background negro
    set(gca,'xtick',[]); %elimina eje X
    set(gca,'ytick',[]); %elimina eje Y
    set(gca,'position',[0 0 1 1],'units','normalized') %elimina bordes en el↙
plot
    print(fullfile(pathHolo,nombreHolo),'-dpng',['-r' num2str(resolution)]);

    [ny,nx]=size(Holograma);
    z=(2-0.7):dz:(2+0.7);

    for ind=1:length(z)
        if (ind==1)
            Bx = -Posiciones1(1,:,1)+nx/2;
            By = -Posiciones1(1,:,2)+ny/2;
            Bz = z(1)*ones(1,NumPart1);
        else
        Bx = cat(2,Bx,-Posiciones1(ind,:,1)+nx/2);
        By = cat(2,By,-Posiciones1(ind,:,2)+ny/2);
        Bz = cat(2,Bz,z(ind)*ones(1,NumPart1));
        end
```

```
 52      end
 53
 54      for ind=1:length(z)
 55          if (ind==1)
 56              Mx = -Posiciones2(1,:,1)+nx/2;
 57              My = -Posiciones2(1,:,2)+ny/2;
 58              Mz = z(1)*ones(1,NumPart2);
 59          else
 60          Mx = cat(2,Mx,-Posiciones2(ind,:,1)+nx/2);
 61          My = cat(2,My,-Posiciones2(ind,:,2)+ny/2);
 62          Mz = cat(2,Mz,z(ind)*ones(1,NumPart2));
 63          end
 64      end
 65
 66      BSize = [Diam1*ones(1,NumPart1*length(z))];
 67      MSize = [Diam2*ones(1,NumPart2*length(z))];
 68      %propago el holograma
 69      [PlanoMax, PlanoMin, PlanosTotal] = PropagarHolograma(Holograma,R,↙
Posiciones1,Posiciones2,dz);
 70      %obtengo el mapa de profundidad
 71      print(fullfile(pathProf,nombreProf),'-dpng',['-r' num2str(resolution)]);
 72      %obtengo la proyección de fase máxima
 73      faseMAX = squeeze(max(angle(PlanosTotal)));
 74      figure(2);
 75      grid off
 76      box off
 77      axis off
 78      set(gcf,'paperunits','inches','paperposition',[0 0 output_size/resolution]);
 79      imagesc(escala(faseMAX,0.999,0.001));colormap gray
 80      title('');
 81      set(gca,'visible','off')
 82      nombreFase=strcat('MaxPhase',num2str(H));
 83      pathFase='F:\Universidad\Universidad\TFG\TFG Jorge Condor↙
Lacambra\Dataset\MaxPhase\MaxPhase';
 84      set(gcf,'NumberTitle','off'); %elimina el título del plot
 85      set(gca,'color','k'); %color de background negro
 86      set(gca,'xtick',[]); %elimina eje X
 87      set(gca,'ytick',[]); %elimina eje Y
 88      set(gca,'position',[0 0 1 1],'units','normalized') %elimina bordes en el↙
plot
 89       print(fullfile(pathFase,nombreFase),'-dpng',['-r' num2str(resolution)]);
 90       %guardo los datos de posiciones en variables para luego exportar a un
 91       %excel
 92       if(H==1) HologramasBlancas=H*ones(length(z)*NumPart1,1); else
 93          HologramasBlancas=[HologramasBlancas;H*ones(length(z)*NumPart1,1)];
 94       end
 95       if(H==1) HologramasMagneticas=H*ones(length(z)*NumPart2,1); else
 96          HologramasMagneticas=[HologramasMagneticas;H*ones(length(z)*NumPart2,↙
1)];
 97       end
 98       if(H==1) PosicionesBlancasTotal=[Bx(:) By(:) Bz(:)]; else
 99          PosicionesBlancasTotal=[PosicionesBlancasTotal;Bx(:) By(:) Bz(:)];
100       end
101       if(H==1) PosicionesMagneticasTotal=[Mx(:) My(:) Mz(:)]; else
```

```
102          PosicionesMagneticasTotal=[PosicionesMagneticasTotal;Mx(:) My(:) Mz(:)];
103       end
104  end
105      %por último guardo los datos de la posición de cada partícula en cada
106       %holograma en un excel
107       TablaBlancas = table(HologramasBlancas,PosicionesBlancasTotal(:,1),↙
PosicionesBlancasTotal(:,2),PosicionesBlancasTotal(:,3),'VariableNames',↙
{'Hologram','X','Y','Z'});
108       nombreArchivoB='ParticlePositionsWHITE.xlsx';
109       pathArchivo='F:\Universidad\Universidad\TFG\TFG Jorge Condor↙
Lacambra\Dataset\ParticlePositionData';
110       writetable(TablaBlancas,fullfile(pathArchivo,nombreArchivoB),'Sheet',1);
111       TablaMagneticas = table(HologramasMagneticas,PosicionesMagneticasTotal(:,↙
1),PosicionesMagneticasTotal(:,2),PosicionesMagneticasTotal(:,3),'VariableNames',↙
{'Hologram','X','Y','Z'});
112       nombreArchivoM='ParticlePositionsMAGNETIC.xlsx';
113       writetable(TablaMagneticas,fullfile(pathArchivo,nombreArchivoM),'Sheet',1);
114
```

```
%Crea los train targets
% - Render 2D de la posición 3D de las partículas
% - Mapa de centroides 2D de las partículas
% - Imagen Semántica clasificando las partículas
%AVISO partes del código requerirían ser cambiadas si hubiera un número
%diferente de partículas blancas y magnéticas por plano

%datos iniciales; se podrían sacar directamente de las tablas de datos,
%esto es por facilitar el código de momento

TotalHolo   = 2000; %total de hologramas
BlancasHolo = 2; %blancas por plano
MagnetHolo  = 2; %magnéticas por plano
dz = 0.1; %distancia entre planos

%ahora cargo los archivos con los datos generados anteriormente en tablas

pathDATOS = 'F:\Universidad\Universidad\TFG\TFG Jorge Condor↙
Lacambra\DatasetV2\ParticlePositionData';
nombreArcMAG = 'ParticlePositionsMAGNETIC.xlsx';
datosMAG = fullfile(pathDATOS,nombreArcMAG);

tablaMAG = readtable(datosMAG);

NumHologramaMAG = table2array(tablaMAG(:,1));
XposMAG         = table2array(tablaMAG(:,2));
YposMAG         = table2array(tablaMAG(:,3));
ZposMAG         = table2array(tablaMAG(:,4));

nombreArcBLANCA = 'ParticlePositionsWHITE.xlsx';
datosBLANCA = fullfile(pathDATOS,nombreArcBLANCA);

tablaBLANCA = readtable(datosBLANCA);

NumHologramaBLANCA = table2array(tablaBLANCA(:,1));
XposBLANCA         = table2array(tablaBLANCA(:,2));
YposBLANCA         = table2array(tablaBLANCA(:,3));
ZposBLANCA         = table2array(tablaBLANCA(:,4));

%ahora procedo a hacer el plot de los targets


minZMAG   = min(ZposMAG);
maxZMAG   = max(ZposMAG);
minZBLA   = min(ZposBLANCA);
maxZBLA   = max(ZposBLANCA);


Ny=2160; %tamaño imagen original
Nx=2560;
partCont  = 0;
TOTALcont = 1;
output_size = [256 256]; %tamaño de las figuras de salida, coincidirá con el↙
usado en el modelo IA
```

```
 53 resolution = 256; %dpi de la imagen
 54
 55
 56
 57 for H = 1:TotalHolo
 58     close all
 59     figure;
 60     grid off
 61     box off
 62     axis off
 63     set(gcf,'paperunits','inches','paperposition',[0 0 output_size/resolution]);
 64     nombreCENT=strcat('Centroids',num2str(H));
 65     pathCENT='F:\Universidad\Universidad\TFG\TFG Jorge Condor↙
Lacambra\DatasetV2\Centroids\Centroids';
 66     partCont = TOTALcont;
 67     while (partCont-TOTALcont)<((1.5/dz)*MagnetHolo)
 68         scatter(-XposMAG(partCont),YposMAG(partCont),2,'w','filled','s');
 69         hold on;
 70         partCont=partCont+1;
 71     end
 72
 73     partCont = TOTALcont;
 74
 75     while (partCont-TOTALcont)<((1.5/dz)*BlancasHolo)
 76         scatter(-XposBLANCA(partCont),YposBLANCA(partCont),2,'w','filled','s');
 77         hold on;
 78         partCont=partCont+1;
 79     end
 80
 81     hold off;
 82     title('');
 83     set(gcf,'NumberTitle','off'); %elimina el título del plot
 84     set(gca,'color','k'); %color de background negro
 85     set(gca,'xtick',[]); %elimina eje X
 86     set(gca,'ytick',[]); %elimina eje Y
 87     set(gca,'position',[0 0 1 1],'units','normalized') %elimina bordes en el↙
plot
 88     set(gcf,'InvertHardcopy','off'); %mantiene color negro de background al↙
guardar el plot
 89     camroll(-180)
 90     print(fullfile(pathCENT,nombreCENT),'-dpng',['-r' num2str(resolution)]);
 91
 92
 93     %ahora procedo a generar el mapa de localizaciones 3D de las partículas
 94
 95     figure;
 96     grid off
 97     box off
 98     axis off
 99     set(gcf,'paperunits','inches','paperposition',[0 0 output_size/resolution]);
100     nombreMAP=strcat('LocationMap',num2str(H));
101     pathMAP='F:\Universidad\Universidad\TFG\TFG Jorge Condor↙
Lacambra\DatasetV2\LocationMap\LocationMap';
102     partCont = TOTALcont;
```

```
103     while (partCont-TOTALcont)<((1.5/dz)*MagnetHolo)
104         colorMAG = (ZposMAG(partCont)-1)/1.7; %la intensidad del cuadrado es↙
proporcional a la posición en z de la partículaç
105         if (colorMAG>1)
106             colorMAG = 1;
107         end
108         scatter(-XposMAG(partCont),YposMAG(partCont),4,[colorMAG colorMAG↙
colorMAG],'filled','s');
109         hold on;
110         partCont=partCont+1;
111     end
112 
113     partCont = TOTALcont;
114 
115     while (partCont-TOTALcont)<((1.5/dz)*BlancasHolo)
116         colorBLANCA = (ZposBLANCA(partCont)-1)/1.7;
117         if (colorBLANCA>1)
118             colorBLANCA = 1;
119         end
120         scatter(-XposBLANCA(partCont),YposBLANCA(partCont),4,[colorBLANCA↙
colorBLANCA colorBLANCA],'filled','s');
121         hold on;
122         partCont=partCont+1;
123     end
124 
125 
126     hold off;
127     title('');
128     set(gcf,'NumberTitle','off'); %elimina el título del plot
129     set(gca,'color','k'); %color de background negro
130     set(gca,'xtick',[]); %elimina eje X
131     set(gca,'ytick',[]); %elimina eje Y
132     set(gca,'position',[0 0 1 1],'units','normalized') %elimina bordes en el↙
plot
133     set(gcf,'InvertHardcopy','off'); %mantiene color negro de background al↙
guardar el plot
134     camroll(-180)
135     print(fullfile(pathMAP,nombreMAP),'-dpng',['-r' num2str(resolution)]);
136 
137     %ahora genero el mapa segmentado
138 
139     close all
140     figure;
141     grid off
142     box off
143     axis off
144     set(gcf,'paperunits','inches','paperposition',[0 0 output_size/resolution]);
145     nombreCENT=strcat('SegmentedMap',num2str(H));
146     pathCENT='F:\Universidad\Universidad\TFG\TFG Jorge Condor↙
Lacambra\DatasetV2\SegmentedMap\SegmentedMap';
147     partCont = TOTALcont;
148     while (partCont-TOTALcont)<((1.5/dz)*MagnetHolo)
149         scatter(-XposMAG(partCont),YposMAG(partCont),10,[1 0 0],'filled','o');
150         hold on;
```

```
151         partCont=partCont+1;
152     end
153 
154     partCont = TOTALcont;
155 
156     while (partCont-TOTALcont)<((1.5/dz)*BlancasHolo)
157         scatter(-XposBLANCA(partCont),YposBLANCA(partCont),30,[0 1↙
0],'filled','o');
158         hold on;
159         partCont=partCont+1;
160     end
161 
162 
163     TOTALcont = TOTALcont+30;
164 
165     hold off;
166     title('');
167     set(gcf,'NumberTitle','off'); %elimina el título del plot
168     set(gca,'color','k'); %color de background negro
169     set(gca,'xtick',[]); %elimina eje X
170     set(gca,'ytick',[]); %elimina eje Y
171     set(gca,'position',[0 0 1 1],'units','normalized') %elimina bordes en el↙
plot
172     set(gcf,'InvertHardcopy','off'); %mantiene color negro de background al↙
guardar el plot
173     camroll(-180)
174     print(fullfile(pathCENT,nombreCENT),'-dpng',['-r' num2str(resolution)]);
175 end
176 
177 
178 
179 
```

```python
# -*- coding: utf-8 -*-
"""Versión definitiva del código: contiene la pipeline de entrada de datos del
dataset, el modelo y el proceso de entrenamiento. También guarda los
resultados y el modelo entrenado"""

"""Importo las librerías necesarias"""

import tensorflow as tf
from tensorflow.keras.backend import sigmoid
from tensorflow.keras import layers
import os
import numpy as np
import random
import pickle
from tensorflow.keras.utils import get_custom_objects


"""Compruebo la versión de tensorflow y si está activado el eager executing mode"""
"""Esto hace más sencillo hacer debugging del modelo al evaluar las operaciones
   inmediatamente en vez de contruir un grafo y ejecutarlas más tarde"""

print("Version: ", tf.__version__)
print("Eager mode: ", tf.executing_eagerly())

"""datos de interés sobre el dataset"""

DATASET_SIZE = 2000

"""hiperparámetros"""

IMG_SIZE = 256 #las imágenes que se van a introducir a la red neuronal tendrán un tamaño IMG_SIZExIMG_SIZE
BATCH_SIZE = 20 #tamaño de cada batch de imágenes del dataset
VALIDATION_SPLIT = 0.2 #determina el ratio train/test (0.2= 20% test, 80% train)
STEPS_PER_EPOCH = 30 #numero de veces que se actualizará el modelo por cada epoch
VALIDATION_STEPS = 20
EPOCHS = 250 #numero de epochs que se va a entrenar el modelo.
LEARNING_RATE = 0.0005 #tamaño del paso que se toma cada iteración para hallar el mínimo global y minimizar las pérdidas.
HUBER_DELTA_LOC = 0.02 #parámetro delta de la función de pérdidas de Huber que utilizo para mapa 3D de localización de partículas
HUBER_DELTA_SEG = 0.2  #parámetro delta para la imagen semántica
GAUSSIAN_NOISE = 10 #valor/255 será el límite superior de ruido aplicado

"""localizo las imágenes del dataset"""

holograms_path_dir = os.path.join(r'/content/drive/My Drive/Universidad/TFG/TFG Jorge Condor Lacambra/Dataset/Hologram')
maxphase_path_dir =  os.path.join(r'/content/drive/My Drive/Universidad/TFG/TFG Jorge Condor Lacambra/Dataset/MaxPhase')
locationmap_path_dir = os.path.join(r'/content/drive/My Drive/Universidad/TFG/TFG Jorge Condor Lacambra/Dataset/LocationMap')
centroids_path_dir = os.path.join(r'/content/drive/My Drive/Universidad/TFG/TFG Jorge Condor Lacambra/Dataset/Centroids')
segmented_path_dir = os.path.join(r'/content/drive/My Drive/Universidad/TFG/TFG Jorge Condor Lacambra/Dataset/SegmentedMap')

"""funciones para preprocesar las imágenes (normalizar, redimensionar y data augmentation)
 y convertirlas en batches. He preferido utilizar generadores para evitar tener en memoria el dataset completo,
 debido a limitaciones de memoria gráfica"""

"""Empiezo definiendo el aumentador, que añadirá variedad a mi dataset, lo dividirá en test y train y lo escalará adecuadamente"""
seed=1 #tengo que asegurar que cada set de hologramas, proyecciones y mapas de localización reciben el mismo tratamiento de data augmentation

def ruido_gaussiano(img):
    variabilidad = GAUSSIAN_NOISE
    desviacion = variabilidad*random.random()
    ruido = np.random.normal(0, desviacion, img.shape)
    img += ruido
    np.clip(img, 0., 255.)
    return img

argumentos_aumentador_input = dict(validation_split=VALIDATION_SPLIT,
                                   preprocessing_function=ruido_gaussiano,
                                   rescale=1./255)

argumentos_aumentador_output = dict(validation_split=VALIDATION_SPLIT,
                                    rescale=1./255)

argumentos_aumentador_test = dict(validation_split=VALIDATION_SPLIT,
                                  rescale=1./255)

aumentador_input = tf.keras.preprocessing.image.ImageDataGenerator(**argumentos_aumentador_input)
aumentador_output = tf.keras.preprocessing.image.ImageDataGenerator(**argumentos_aumentador_output)
aumentador_test = tf.keras.preprocessing.image.ImageDataGenerator(**argumentos_aumentador_test)

"""generadores del training dataset"""

argumentos_generador_train = dict(batch_size=BATCH_SIZE,
                            shuffle=False,
                            target_size=(IMG_SIZE,IMG_SIZE),
                            color_mode='grayscale',
                            class_mode=None,
                            seed=seed,
                            subset="training")
argumentos_generador_train_segmented = dict(batch_size=BATCH_SIZE,
                            shuffle=False,
                            target_size=(IMG_SIZE,IMG_SIZE),
                            color_mode='rgb',
```

```
                            class_mode=None,
                            seed=seed,
                            subset="training")

generador_holograms_train = aumentador_input.flow_from_directory(directory=str(holograms_path_dir),**argumentos_generador_train)
generador_maxphase_train = aumentador_input.flow_from_directory(directory=str(maxphase_path_dir),**argumentos_generador_train)
generador_locationmaps_train = aumentador_output.flow_from_directory(directory=str(locationmap_path_dir),**argumentos_generador_train)
generador_centroids_train = aumentador_output.flow_from_directory(directory=str(centroids_path_dir),**argumentos_generador_train)
generador_segmented_train = aumentador_output.flow_from_directory(directory=str(segmented_path_dir),**argumentos_generador_train_segmented)

"""generadores del test o validation dataset. De nuevo, primero defino el aumentador de datos, luego el generador en sí"""

argumentos_generador_test = dict(batch_size=BATCH_SIZE,
                            shuffle=False,
                            target_size=(IMG_SIZE,IMG_SIZE),
                            color_mode='grayscale',
                            class_mode=None,
                            seed=seed,
                            subset="validation")
argumentos_generador_test_segmented = dict(batch_size=BATCH_SIZE,
                            shuffle=False,
                            target_size=(IMG_SIZE,IMG_SIZE),
                            color_mode='rgb',
                            class_mode=None,
                            seed=seed,
                            subset="validation")

generador_holograms_test = aumentador_test.flow_from_directory(directory=str(holograms_path_dir),**argumentos_generador_test)
generador_maxphase_test = aumentador_test.flow_from_directory(directory=str(maxphase_path_dir),**argumentos_generador_test)
generador_locationmaps_test = aumentador_test.flow_from_directory(directory=str(locationmap_path_dir),**argumentos_generador_test)
generador_centroids_test = aumentador_test.flow_from_directory(directory=str(centroids_path_dir),**argumentos_generador_test)
generador_segmented_test = aumentador_test.flow_from_directory(directory=str(segmented_path_dir),**argumentos_generador_test_segmented)

"""ahora junto los generadores de imágenes de input en uno solo, y los de output o target en otro"""

train_input_generator = (pair for pair in zip(generador_holograms_train, generador_maxphase_train))
train_target_generator = (pair for pair in zip(generador_locationmaps_train, generador_centroids_train, generador_segmented_train))
train_gen = (pair for pair in zip(train_input_generator, train_target_generator))

test_input_generator = (pair for pair in zip(generador_holograms_test, generador_maxphase_test))
test_target_generator = (pair for pair in zip(generador_locationmaps_test, generador_centroids_test, generador_segmented_test))
test_gen = (pair for pair in zip(test_input_generator, test_target_generator))

"""defino la función de activación que usaré entre las capas del modelo"""

def swish(x, beta = 1):
        return (x*sigmoid(beta*x))

get_custom_objects().update({'swish': swish})

"""defino la función de pérdidas MSE regularizada con la TV (Variación Total). La TV ayuda a eliminar el ruido de salida."""
"""
def MSE_TVreg(y_true,y_pred):
    pixel_dif1 = y_pred[:, 1:, :, :] - y_pred[:, :-1, :, :]
    pixel_dif2 = y_pred[:, :, 1:, :] - y_pred[:, :, :-1, :]
    sum_axis = [1,2,3]
    TV = (K.sqrt(
        (math_ops.reduce_sum(math_ops.abs(pixel_dif1), axis=sum_axis))**2 +
        (math_ops.reduce_sum(math_ops.abs(pixel_dif2), axis=sum_axis))**2))

    loss_value = (1-TV_ALPHA)*(tf.reduce_mean(tf.math.squared_difference(y_pred, y_true)))+TV_ALPHA*K.square(TV)
    return loss_value

get_custom_objects().update({'MSE_TVreg': MSE_TVreg})
"""

"""defino el modelo"""

class HoloNet:
    def build(IMG_SIZE):
        hologram_input = tf.keras.Input(shape=(IMG_SIZE,IMG_SIZE,1), name='hologram_input')
        maxphase_input = tf.keras.Input(shape=(IMG_SIZE,IMG_SIZE,1), name='maxphase_input')


        input_layer = layers.concatenate([hologram_input, maxphase_input])

        #bloque encoder 1
        eb1_1 = layers.Conv2D(16,(3,3),padding='same',activation='swish')(input_layer)
        eb1_2 = layers.Conv2D(16,(3,3),padding='same',activation='swish')(eb1_1)
        input_layer = tf.pad(input_layer,[[0, 0], [0, 0], [0, 0], [7, 7]])
        eb1_2 = layers.Add()([input_layer, eb1_2])
        #bloque encoder 2
        batch_norm_eb1 = layers.BatchNormalization()(eb1_2)
        activation_eb1 = layers.Activation('swish')(batch_norm_eb1)
        max_pool_eb1   = layers.MaxPool2D(pool_size=(2,2),padding='same')(activation_eb1)
        eb2_1 = layers.Conv2D(64,(3,3),padding='same',activation='swish')(max_pool_eb1)
        eb2_2 = layers.Conv2D(64,(3,3),padding='same',activation='swish')(eb2_1)
        max_pool_eb1 = tf.pad(max_pool_eb1,[[0, 0], [0, 0], [0, 0], [24, 24]])
        eb2_2 = layers.Add()([max_pool_eb1, eb2_2])
        #bloque encoder 3
        batch_norm_eb2 = layers.BatchNormalization()(eb2_2)
```

```
        activation_eb2 = layers.Activation('swish')(batch_norm_eb2)
        max_pool_eb2   = layers.MaxPool2D(pool_size=(2,2),padding='same')(activation_eb2)
        eb3_1 = layers.Conv2D(128,(3,3),padding='same',activation='swish')(max_pool_eb2)
        eb3_2 = layers.Conv2D(128,(3,3),padding='same',activation='swish')(eb3_1)
        max_pool_eb2 = tf.pad(max_pool_eb2,[[0, 0], [0, 0], [0, 0], [32, 32]])
        eb3_2 = layers.Add()([max_pool_eb2, eb3_2])
        #bloque encoder 4
        batch_norm_eb3 = layers.BatchNormalization()(eb3_2)
        activation_eb3 = layers.Activation('swish')(batch_norm_eb3)
        max_pool_eb3   = layers.MaxPool2D(pool_size=(2,2),padding='same')(activation_eb3)
        eb4_1 = layers.Conv2D(256,(3,3),padding='same',activation='swish')(max_pool_eb3)
        eb4_2 = layers.Conv2D(256,(3,3),padding='same',activation='swish')(eb4_1)
        max_pool_eb3 = tf.pad(max_pool_eb3,[[0, 0], [0, 0], [0, 0], [64, 64]])
        eb4_2 = layers.Add()([max_pool_eb3, eb4_2])
        #bloque encoder 5
        batch_norm_eb4 = layers.BatchNormalization()(eb4_2)
        activation_eb4 = layers.Activation('swish')(batch_norm_eb4)
        max_pool_eb4   = layers.MaxPool2D(pool_size=(2,2),padding='same')(activation_eb4)
        eb5_1 = layers.Conv2D(512,(3,3),padding='same',activation='swish')(max_pool_eb4)
        eb5_2 = layers.Conv2D(512,(3,3),padding='same',activation='swish')(eb5_1)
        max_pool_eb4 = tf.pad(max_pool_eb4,[[0, 0], [0, 0], [0, 0], [128, 128]])
        eb5_2 = layers.Add()([max_pool_eb4, eb5_2])
        #bloque decoder 1
        batch_norm_db1 = layers.BatchNormalization()(eb5_2)
        activation_db1 = layers.Activation('swish')(batch_norm_db1)
        upsampler_db1  = layers.UpSampling2D(size=(2, 2))(activation_db1)
        upconv_db1     = layers.Conv2DTranspose(256,(3,3),padding='same')(upsampler_db1)
        concat_db1 = layers.concatenate([eb4_2, upconv_db1], axis = 3)
        db1_1 = layers.Conv2D(256,(3,3),strides=(1, 1),padding='same',activation='swish')(concat_db1)
        db1_2 = layers.Conv2D(256,(3,3),strides=(1, 1),padding='same',activation='swish')(db1_1)
        db1_2 = layers.Add()([upconv_db1, db1_2])
        #bloque decoder 2
        batch_norm_db2 = layers.BatchNormalization()(db1_2)
        activation_db2 = layers.Activation('swish')(batch_norm_db2)
        upsampler_db2  = layers.UpSampling2D(size=(2, 2))(activation_db2)
        upconv_db2     = layers.Conv2DTranspose(128,(3,3),padding='same')(upsampler_db2)
        concat_db2 = layers.concatenate([eb3_2, upconv_db2], axis = 3)
        db2_1 = layers.Conv2D(128,(3,3),strides=(1, 1),padding='same',activation='swish')(concat_db2)
        db2_2 = layers.Conv2D(128,(3,3),strides=(1, 1),padding='same',activation='swish')(db2_1)
        db2_2 = layers.Add()([upconv_db2, db2_2])
        #bloque decoder 3
        batch_norm_db3 = layers.BatchNormalization()(db2_2)
        activation_db3 = layers.Activation('swish')(batch_norm_db3)
        upsampler_db3  = layers.UpSampling2D(size=(2, 2))(activation_db3)
        upconv_db3     = layers.Conv2DTranspose(64,(3,3),padding='same')(upsampler_db3)
        concat_db3 = layers.concatenate([eb2_2, upconv_db3], axis = 3)
        db3_1 = layers.Conv2D(64,(3,3),strides=(1, 1),padding='same',activation='swish')(concat_db3)
        db3_2 = layers.Conv2D(64,(3,3),strides=(1, 1),padding='same',activation='swish')(db3_1)
        db3_2 = layers.Add()([upconv_db3, db3_2])
        #bloque decoder 4
        batch_norm_db4 = layers.BatchNormalization()(db3_2)
        activation_db4 = layers.Activation('swish')(batch_norm_db4)
        upsampler_db4  = layers.UpSampling2D(size=(2, 2))(activation_db4)
        upconv_db4     = layers.Conv2DTranspose(16,(3,3),strides=(1, 1),padding='same')(upsampler_db4)
        concat_db4 = layers.concatenate([eb1_2,upconv_db4], axis = 3)
        db4_1 = layers.Conv2D(16,(3,3),strides=(1,1),padding='same',activation='swish')(concat_db4)
        db4_2 = layers.Conv2D(16,(3,3),padding='same',activation='swish')(db4_1)
        db4_2 = layers.Add()([upconv_db4, db4_2])
        #capa de salida
        locationmap_output = layers.Conv2D(1,(1,1),padding='same',activation='sigmoid',name='locationmap_output')(db4_2)
        centroids_output = layers.Conv2D(1,(1,1),padding='same',activation='sigmoid',name='centroids_output')(db4_2)
        segmented_output = layers.Conv2D(3,(1,1),padding='same',activation='sigmoid',name='segmented_output')(db4_2)

        inputs = [hologram_input, maxphase_input]
        outputs = [locationmap_output, centroids_output, segmented_output]

        HoloNet = tf.keras.Model(inputs = inputs,outputs = outputs)

        return HoloNet

"""construyo el modelo"""

HoloNet = HoloNet.build(IMG_SIZE)

"""Creo un plot para comprobar que el modelo se ha generado correctamente y la estructura es la deseada"""

resultados_dir = os.path.join(r'/content/drive/My Drive/Universidad/TFG/TFG Jorge Condor Lacambra/Resultados')

tf.keras.utils.plot_model(HoloNet, resultados_dir+'/Esquema del Modelo/HoloNet.png', show_shapes=True)

"""defino el optimizador a usar en el proceso de training"""
optimizador = tf.keras.optimizers.Adam(learning_rate=LEARNING_RATE)

"""Compilo el modelo, cada salida tiene una función de pérdidas distinta"""
HoloNet.compile(optimizer=optimizador,
                loss={'locationmap_output':tf.keras.losses.Huber(delta=HUBER_DELTA_LOC),
                    'centroids_output': 'MSE',
                    'segmented_output': tf.keras.losses.Huber(delta=HUBER_DELTA_SEG)})

"""inicio el training"""
```

```
historial = HoloNet.fit(train_gen,
                        steps_per_epoch=STEPS_PER_EPOCH,
                        epochs=EPOCHS,
                        verbose=1,
                        validation_data=test_gen,
                        validation_steps=VALIDATION_STEPS)

"""guardo el modelo entrenado y su historial de pérdidas"""

HoloNet.save(resultados_dir+'/Modelo Entrenado/HoloNet.h5')

archivo_historial = open(resultados_dir+'/Modelo Entrenado/Historial', 'wb')
pickle.dump(historial.history, archivo_historial)
archivo_historial.close()
```

```python
# -*- coding: utf-8 -*-
"""Versión definitiva del código de evaluación: carga el modelo entrenado y sets de input del Test
Dataset, y los introduce en el modelo para hacer predicciones. Imprime
en pantalla y guarda los resultados (pérdidas, comparativa visual)"""

import tensorflow as tf
import os
import numpy as np
import cv2
import pickle
import matplotlib.pyplot as plt

"algunas variables que sigo utilizando"

TEST_DATASET_SIZE = 20
IMG_SIZE = 256

"""directorios que utilizo"""
holograms_path_dir = os.path.join(r'/content/drive/My Drive/Universidad/TFG/TFG Jorge Condor Lacambra/TestDataset/Hologram')
maxphase_path_dir =  os.path.join(r'/content/drive/My Drive/Universidad/TFG/TFG Jorge Condor Lacambra/TestDataset/MaxPhase')
locationmap_path_dir = os.path.join(r'/content/drive/My Drive/Universidad/TFG/TFG Jorge Condor Lacambra/TestDataset/LocationMap')
centroids_path_dir = os.path.join(r'/content/drive/My Drive/Universidad/TFG/TFG Jorge Condor Lacambra/TestDataset/Centroids')
segmented_path_dir = os.path.join(r'/content/drive/My Drive/Universidad/TFG/TFG Jorge Condor Lacambra/TestDataset/SegmentedMap')

resultados_dir = os.path.join(r'/content/drive/My Drive/Universidad/TFG/TFG Jorge Condor Lacambra/Resultados')


"""cargo el modelo ya entrenado y el historial generado"""

HoloNet = tf.keras.models.load_model(resultados_dir+'/Modelo Entrenado/HoloNet.h5')

archivo_historial = open(resultados_dir+'/Modelo Entrenado/Historial', 'rb')
historial = pickle.load(archivo_historial)
archivo_historial.close()

"""plot de las gráficas de resultados. Pérdidas sobre época, para las 3 funciones de pérdidas"""

plt.plot(historial['locationmap_output_loss'])
plt.plot(historial['val_locationmap_output_loss'])
plt.title('Pérdidas del modelo (Mapa localización 3D de las partículas)')
plt.yscale("log")
plt.ylabel('Pérdidas')
plt.xlabel('Época')
plt.legend(['train', 'test'], loc='upper left')
plt.savefig(resultados_dir+'/Model Losses/locationmap_loss.png')
plt.show()

plt.plot(historial['centroids_output_loss'])
plt.plot(historial['val_centroids_output_loss'])
plt.title('Pérdidas del modelo (Mapa de centroides o localización 2D)')
plt.yscale("log")
plt.ylabel('Pérdidas')
plt.xlabel('Época')
plt.legend(['train', 'test'], loc='upper left')
plt.savefig(resultados_dir+'/Model Losses/centroids_loss.png')
plt.show()

plt.plot(historial['segmented_output_loss'])
plt.plot(historial['val_segmented_output_loss'])
plt.title('Pérdidas del modelo (Mapa Segmentado)')
plt.yscale("log")
plt.ylabel('Pérdidas')
plt.xlabel('Época')
plt.legend(['train', 'test'], loc='upper left')
plt.savefig(resultados_dir+'/Model Losses/segmented_loss.png')
plt.show()

"""y ahora los resultados generales"""
plt.plot(historial['loss'])
plt.plot(historial['val_loss'])
plt.title('Pérdidas del modelo (general)')
plt.yscale("log")
plt.ylabel('Pérdidas')
plt.xlabel('Época')
plt.legend(['train', 'test'], loc='upper left')
plt.savefig(resultados_dir+'/Model Losses/overall_loss.png')
plt.show()


"""hago algunas predicciones para evaluar visualmente los resultados"""
"""funciones para generar y mostrar predicciones del modelo una vez entrenado"""

def muestra_aleatoria():

    num = np.random.randint(TEST_DATASET_SIZE)

    holograma_random = cv2.imread(holograms_path_dir+'/Hologram/Hologram'+str(num)+'.png',1)
    maxphase_random = cv2.imread(maxphase_path_dir+'/MaxPhase/MaxPhase'+str(num)+'.png',1)
    locationmap_random = cv2.imread(locationmap_path_dir+'/LocationMap/LocationMap'+str(num)+'.png',1)
    centroids_random = cv2.imread(centroids_path_dir+'/Centroids/Centroids'+str(num)+'.png',1)
    segmented_random = cv2.imread(segmented_path_dir+'/SegmentedMap/SegmentedMap'+str(num)+'.png',1)
```

```
    holograma_random = cv2.normalize(holograma_random, None, alpha=0, beta=1, norm_type=cv2.NORM_MINMAX, dtype=cv2.CV_32F)
    maxphase_random = cv2.normalize(maxphase_random, None, alpha=0, beta=1, norm_type=cv2.NORM_MINMAX, dtype=cv2.CV_32F)
    locationmap_random = cv2.normalize(locationmap_random, None, alpha=0, beta=1, norm_type=cv2.NORM_MINMAX, dtype=cv2.CV_32F)
    centroids_random = cv2.normalize(centroids_random, None, alpha=0, beta=1, norm_type=cv2.NORM_MINMAX, dtype=cv2.CV_32F)
    segmented_random = cv2.normalize(segmented_random, None, alpha=0, beta=1, norm_type=cv2.NORM_MINMAX, dtype=cv2.CV_32F)

    holograma_random = cv2.resize(holograma_random,dsize=(IMG_SIZE,IMG_SIZE))
    maxphase_random = cv2.resize(maxphase_random,dsize=(IMG_SIZE,IMG_SIZE))
    locationmap_random = cv2.resize(locationmap_random,dsize=(IMG_SIZE,IMG_SIZE))
    centroids_random = cv2.resize(centroids_random,dsize=(IMG_SIZE,IMG_SIZE))
    segmented_random = cv2.resize(segmented_random,dsize=(IMG_SIZE,IMG_SIZE))

    holograma_random = np.asarray(holograma_random)
    maxphase_random = np.asarray(maxphase_random)
    locationmap_random = np.asarray(locationmap_random)
    centroids_random = np.asarray(centroids_random)
    segmented_random = np.asarray(segmented_random)

    holograma_random = np.expand_dims(holograma_random,axis=0)
    maxphase_random = np.expand_dims(maxphase_random,axis=0)
    locationmap_random = np.expand_dims(locationmap_random,axis=0)
    centroids_random = np.expand_dims(centroids_random,axis=0)
    segmented_random = np.expand_dims(segmented_random,axis=0)

    holograma_random = tf.image.rgb_to_grayscale(holograma_random)
    maxphase_random = tf.image.rgb_to_grayscale(maxphase_random)
    locationmap_random = tf.image.rgb_to_grayscale(locationmap_random)
    centroids_random = tf.image.rgb_to_grayscale(centroids_random)

    input_data = [holograma_random, maxphase_random]
    output_data = [locationmap_random, centroids_random, segmented_random]

    return input_data, output_data

def mostrar(lista_muestras):
  plt.figure(figsize=(30, 30))
  title = ['Holograma','Proyección de Fase Máxima', 'Mapa de Centroides REAL', 'Mapa de Centroides PREDICCION', 'Mapa de localizaciones 3D R

  for i in range(len(lista_muestras)):
    plt.subplot(4, 2, i+1)
    plt.title(title[i])
    plt.imshow(tf.keras.preprocessing.image.array_to_img(np.squeeze(lista_muestras[i],axis=0)), cmap='gray')
    plt.axis('off')
  plt.tight_layout()
  plt.savefig(resultados_dir+'/Predicciones/pred_aleatoria.jpeg', dpi=1000)
  plt.show()

"""genero una predicción con las funciones definidas"""

input_data, output_data = muestra_aleatoria()

prediccion = HoloNet(input_data, training=False)

location_pred = prediccion[0]
centroids_pred = prediccion[1]
segmented_pred = prediccion[2]

mostrar([input_data[0], input_data[1], output_data[1], centroids_pred, output_data[0], location_pred, output_data[2], segmented_pred])
```

Anexo II: Datasets, Modelo Entrenado e Historial de Pérdidas

Jorge Condor Lacambra

25 de Junio 2020

Grado en Ingeniería Electrónica y Automática

A continuación publico enlaces para poder consultar o descargar distintos archivos de interés generados durante el Trabajo. Ante cualquier problema para acceder a los mismos por favor no dude en dirigir un correo a `jorge.condorlacambra@gmail.com`.

II.1. Datasets

Dataset completo utilizado en el Trabajo como entrenamiento y validación:

`https://drive.google.com/drive/folders/1fC3fmKDr7SON5wnJR0frKnoQYPorKaXv?usp=sharing`

Dataset de Test utilizado para la evaluación del modelo:

`https://drive.google.com/drive/folders/1rQHE4UbT18P24uXyGfItgH_OpyG1NqY-?usp=sharing`

II.2. Modelo Entrenado

Archivo H5 del Modelo Entrenado:

`https://drive.google.com/file/d/1x97MipmOdexCsikBBDrbj-q5v6WPegaA/view?usp=sharing`

II.3. Historial de Pérdidas

Archivo del Historial de Pérdidas (necesita extraerse en un script de Python con el comando unpickle):

`https://drive.google.com/file/d/1haKFWpZyFvfcKj3Dp1M7jdlIBoPT8ap8/view?usp=sharing`