

VISIÓN POR COMPUTADOR
GRADO EN INGENIERÍA INFORMÁTICA

RECONOCIMIENTO DE NÚCLEOS MEDIANTE MASK R-CNN

Realizado por:
Alba Casillas Rodríguez
Jose Manuel Osuna Luque

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍAS INFORMÁTICA Y DE
TELECOMUNICACIÓN

CURSO 2021-2022



Índice

1. Introducción	2
2. Instalación	4
2.1. Equipo de ejecución	6
3. Mask R-CNN	7
3.1. Detalles de implementación	9
3.2. Métricas	10
4. Modelo Inicial	12
5. Mejoras realizadas	21
5.1. Aumentar el número de épocas	21
5.2. Data Augmentation	25
5.3. Ajuste fino de toda la red	29
5.4. Aumentar el tamaño de las imágenes	32
6. Predicciones	36
7. Propuestas de mejora	39
8. Bibliografía	41
Referencias	41

1. Introducción

El proyecto desarrollado se basa en el *2018 Data Science Bowl* publicado por **Kaggle** [1], cuyo objetivo es **detectar y segmentar núcleos celulares** en un conjunto de imágenes ópticas en distintas condiciones experimentales y con distintos protocolos de tinción sin la necesidad de ajustar manualmente los parámetros de la segmentación.

Se dispone de un total de 670 imágenes para el conjunto de entrenamiento, el cual es dividido posteriormente para obtener un conjunto de validación. Dicho conjunto se encuentra en la carpeta **stage1_train**, el cual está compuesto por numerosas carpetas tituladas con el *id* correspondiente a cada imagen y contendrán dos carpetas en su interior: *images*, con dicha imagen y *masks*, donde se encuentran el conjunto de máscaras asociadas a cada uno de los núcleos detectados en la imagen. Dichas máscaras serán utilizadas para realizar el entrenamiento.

Por otro lado, se hará uso del conjunto de test **stage1_test**, con 65 imágenes que, como es de esperar, no tienen máscaras asociadas.

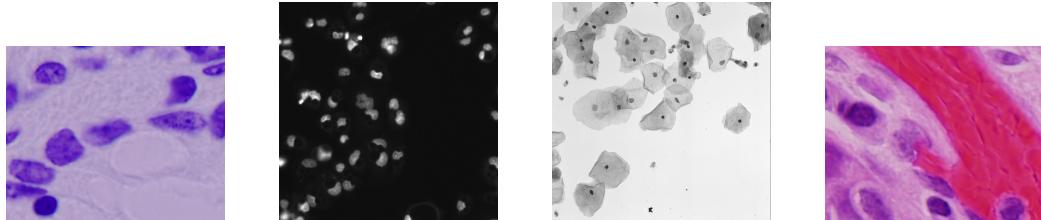


Figura 1: Ejemplo de imágenes obtenidas del conjunto de entrenamiento.

Se plantea abordar este problema mediante el uso de **Mask R-CNN** a partir de la implementación de código abierto de **matterport**, [2], basada en *Python 3*, *Keras* y *Tensorflow*. El modelo generará bounding boxes y máscaras de segmentación de las distintas instancias encontradas en una imagen.

Consideraciones a tener en cuenta: El código perteneciente a los entrenamientos del modelo quedará debidamente comentado en la práctica, pero siendo deshabilitada su ejecución, debido al gran tiempo de ejecución que consume cada uno de ellos. A su vez, serán proporcionados todos los pesos obtenidos y usados para la detección.

Tanto el **DATASET** como los **PESOS** (al superar la carga máxima permitida en Prado) se han subido a la **consigna** del alumno Jose Manuel Osuna Luque, cuyos enlaces son:

- **Dataset.** <https://consigna.ugr.es/f/y2GhV6dJn61NCAEH/dataset.zip>
- **Pesos_1.** https://consigna.ugr.es/f/KYGqZuy90kWoon34/Pesos_1.zip
- **Pesos_2.** https://consigna.ugr.es/f/Ik4uVW3YEHKkpxrP/Pesos_2.zip
- **Pesos_3.** https://consigna.ugr.es/f/WWLhZ81jz4LBlodq/Pesos_3.zip

Como la **consigna** solo permite añadir ficheros con un máximo de 500MB, se han realizado 3 zips para los pesos para poder subirlos. Cada fichero contiene 2 pesos entrenados. Será suficiente con descargar todos los zips y extraerlos en el mismo directorio donde se encuentre el fichero con el código fuente del proyecto. Estos ZIPs al extraerse deberán crear 3 carpetas, una llamada *stage1_train*, otra *stage1_test* y otra *Pesos*, y dentro de estas carpetas estarán los datos pertinentes.

***NOTA:** Es posible que de extraer estos ZIPs en MAC o UBUNTU, genere unas carpetas raíz extras con el nombre de los archivos comprimidos, de forma que quede una carpeta con el mismo nombre del ZIP que englobe las demás, en vez de las carpetas mencionadas. Queda por tanto mencionado que es necesario que al extraerse (tal y como está preparado) queden las carpetas mencionadas, y dentro de estas directamente los respectivos datos.

2. Instalación

Para el correcto funcionamiento de la práctica, se ha necesitado la instalación de un nuevo entorno virtual.

Por ello, el primero paso es la creación del mismo desde la *terminal de Anaconda*.

```
1 $ conda activate PROYECTOVC
```

Tras esto, se instala una versión más antigua de **Python** que será compatible con las versiones necesarias en el resto de herramientas.

```
1 $ pip install python==3.7
```

Se usará la versión 3.7 en vez de la 3.8 ya que es la última versión donde funciona la versión necesaria de *Tensorflow*.

Las versiones de **Tensorflow** más recientes que son compatibles con los módulos utilizados por Mask-RCNN son:

```
1 $ pip install tensorflow==1.13.1
2 $ pip install tensorflow-gpu==1.15.0
```

Se instala **CUDA** para poder hacer uso de la *GPU* para disminuir el tiempo de ejecución del proyecto. Atendiendo a la documentación [4], para que pueda ser compatible con la versión instalada de *Tensorflow*, será necesaria la versión 8, la cual se puede obtener a partir de [5].

NOTA: Si en el ordenador ya se encuentra una versión más reciente de CUDA, la más antigua no será reconocida en la ejecución. Por ello, una alternativa encontrada ha sido añadir a la carpeta *bin* de la versión más reciente de **CUDA** los archivos:

- **cudart64_100.dll**
- **nvcuda.dll**
- **cublas64_100.dll**
- **cufft64_100.dll**

- **curand64_100.dll**
- **cusolver64_100.dll**
- **cuparse64_100.dll**
- **cudnn64_7.dll**

En la carpeta *include* deberá estar el archivo:

- **cudnn.h**

y en *lib/x64* el archivo:

- **cudnn.lib**

La descarga de los tres últimos ficheros especificados se encuentra en [6], en el subapartado **Download cuDNN v7.6.5 [November 18th, 2019]** (la última versión 7 de cuDNN). Para ello, ha sido necesario registrarse en *Nvidia Developer*.

La siguiente instalación será la versión de **Keras** más nueva compatible con los módulos utilizados por el modelo.

```
1 $ pip install keras==2.2.5
```

Para la lectura y procesamiento de los pesos, los cuales portan la extensión *.h5* será necesario:

```
1 $ pip install h5py==2.10.0
```

Para evitar errores de dependencias (en *Spyder*) ha sido obligatorio realizar la instalación del módulo para control remoto **paramiko**.

```
1 $ pip install paramiko==2.4.0
```

Por último, se han instalado librerías utilizadas durante el desarrollo de la práctica:

```
1 $ pip install mrcnn
2 $ sudo install scikit-image
3 $ pip install imgaug
```

2.1. Equipo de ejecución

- **Sistema Operativo:** Windows 10.
- **IDE:** Spyder.
- **GPU:**
 - **NVIDIA GeForce GTX 1050 Ti (4GB).** Con esta gráfica se han encontrado problemas de memoria a la hora de realizar el entrenamiento usando una dimensiones de imágenes de 512x512.
 - **NVIDIA GeForce RTX 2060 SUPER (8GB).** No se han encontrado problemas de memoria al realizar ningún tipo de ejecución.

3. Mask R-CNN

El propósito de las ***R-CNN***, *Region Based Convolution Neural Network* es resolver el problema de la **detección de objetos**. Dada una determinada imagen, se quiere poder dibujar *bounding boxes* sobre todos los objetos. El proceso se puede dividir en dos componentes principales: extraer las propuestas de regiones y su clasificación,

Sin embargo, el objetivo de ***Mask R-CNN*** [3] es la **segmentación de instancias**. La segmentación de instancias resulta una tarea compleja ya que requiere de la correcta detección de todos los objetos mientras que, a su vez, se segmenta cada instancia; combinando de esta manera las tareas clásicas de *detección de objetos*, donde se clasifica y localiza cada objeto individual utilizando un *bounding box*, y la *segmentación semántica*, cuyo objetivo es clasificar cada píxel en un conjunto fijo de categorías sin diferenciar las instancias del objeto.

Mask R-CNN sigue la misma intuición que **Faster R-CNN** en una primera etapa de *Region Proposal Network*, *RPN*, donde se proponen *bounding boxes* candidatos. Sin embargo, extiende de la misma en una segunda etapa de extracción de características mediante los *bounding boxes* de cada *Región de Interés*, *RoI*, añadiendo una rama para predecir *máscaras de segmentación* en cada Región de Interés en paralelo con la rama ya existente para la clasificación y regresión de los *bounding boxes*.

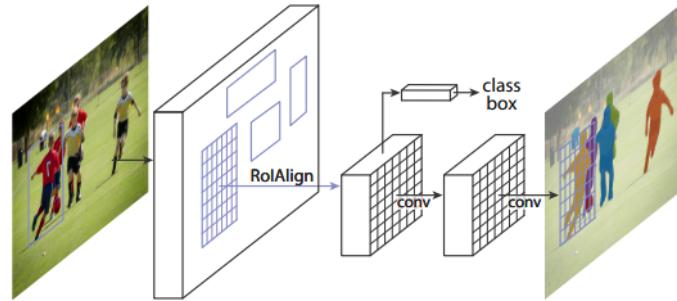


Figura 2: Mask R-CNN para segmentación de instancias, [3].

En cuanto a la **arquitectura** de Mask R-CNN, se consideran dos bloques fundamentales:

- **Backbone:** En muchas tareas de la Visión por Computador se hace uso de un *bakbone*, que suele estar previamente entrenado. De esta forma, la red troncal es usada como un extractor de características que brinda una representación de mapa de características para la entrada (para posteriormente aplicar una tarea de detección, segmentación, etc.). En el caso de Mask R-

CNN, se evalúan las redes *ResNet* y *ResNeXt* (con una profundidad de 50 y 101 capas) y una red de *Pirámide de Características* (*Feature Pyramid Network (FPN)*).

- **Cabecera.** Esta red cabecera realizará la tarea de reconocimiento de bounding boxes (clasificación y regresión), extendiendo de las cabeceras utilizadas por *Faster R-CNN* y mediante la adición de una rama de predicción de la máscara convolucional.

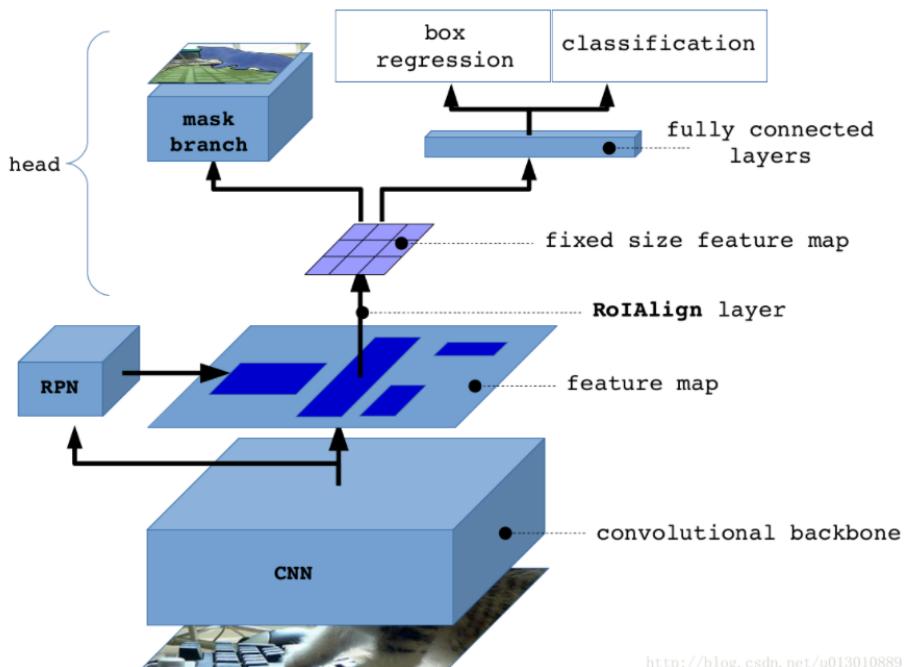


Figura 3: Arquitectura de Mask R-CNN.

Otra diferencia en comparación con *Faster R-CNN* subyace en la capa ***RoIA-lign***, debido a que la *RoIPool*, proveniente de la implementación original de Faster R-CNN, produce un redondeo en los píxeles causando un ligero desajuste en las regiones del mapa de características detectadas. Con RoIAlign se calcula una localización espacial exacta evitando cualquier cuantificación mediante el uso de la *interpolación bilineal* para calcular los valores exactos de las características de entrada.

3.1. Detalles de implementación

La configuración de los hiperparámetros correspondiente a la Mask R-CNN usada en la implementación de *Matterport* [2] es:

Configuración	Valor
Backbone	ResNet101
Backbone Strides	[4, 8, 16, 32, 64]
Batch Size	2
Bbox Std Dev	[0.1 0.1 0.2 0.2]
Compute Backbone shape	None
Detection Max Instances	100
Detection Min Confidence	0.7
Detection NMS Threshold	0.3
FPN Classif FC Layers Size	1024
GPU Count	1
Gradient Clip Norm	5.0
Images per GPU	2
Image Max Dim	1024
Image Meta Size	13
Image Min Dim	800
Image Min Scale	0
Image Resize Mode	square
Image Shape	[1024 1024 3]
Learning Momentum	0.9
Learning Rate	0.001
Mask Pool Size	14
Mask Shape	[28, 28]
Max GT Instances	100
Mean Pixel	[123.7, 116.8, 103.9]
Mini Mask Shape	[56, 56]
Name	None
Num Classes	1
Pool Size	7
Post NMS Rois Inference	1000
Post NMS Rois Training	2000
RoI Positive Ratio	0.33
RPN Anchor Ratios	[0.5, 1, 2]
RPN Anchor Scales	(32, 64, 128, 256, 512)
RPN Anchor Stride	1
RPN Bbox Std Dev	[0.1 0.1 0.2 0.2]
RPN NMS Threshold	0.7

Configuración	Valor
RPN Train Anchors per Image	256
Steps per Epoch	1000
Top Down Pyramid Size	256
Train BN	False
Train RoIs per Image	200
Use Mini Masks	True
Use RPN RoIs	True
Validation Steps	50
Weight Decay	0.0001

Cuadro 1: Configuración base de los hiperparámetros de Mask R-CNN.

Estos parámetros han sido configurados de acuerdo al paper original [3], aunque se difiere en algunos de ellos, como el *learning rate*, el cual originalmente es 0.02, y es considerado un valor muy alto que hacía que los pesos crecieran exponencialmente; es por ello que se establece a 0.001, valor que generalmente genera una convergencia más rápida y mejores resultados.

De igual manera, posteriormente se modificarán algunos de estos hiperparámetros para adaptar el modelo a nuestro problema.

3.2. Métricas

La métrica utilizada es **Average Precision, AP**, derivada de la *precision* y el *recall*. Average Precision es usualmente evaluado calculándose para cada categoría de objetos por separado. Para comparar el rendimiento sobre todas las categorías se utiliza **mean AP, mAP**, siendo el promedio sobre todas las categorías de objetos.

La estrategia de evaluación tiene en cuenta si una caja predicha es correcta. Para calcular una correspondencia entre la predicción y el *ground truth* (la verdadera localización de la máscara), se calcula la métrica **Intersección sobre la Unión (Intersection-over-Union, IoU)** de forma que:

$$IoU = \frac{|A \cap B|}{|A \cup B|}$$

donde A y B son dos objetos y el operador \cap mide el área; es decir, se calcula la proporción del área que forma la intersección de las dos cajas frente al área de las dos cajas unidas.

Se elige un umbral mínimo, t , para identificar objetos correctamente segmentados y cualquier otra máscara de segmentación prevista por debajo del umbral será considerada un error. Considerando *TP* como los verdaderos positivos, *FP*

los falsos positivos, TN los verdaderos negativos y FN los falsos negativos, los criterios de evaluación usados para calcular el mean Average son:

- **Precision:** Mide el porcentaje de predicciones correctas, es decir, en las cajas predichas donde hay realmente un objeto de la clase.

$$\text{Precision}(t) = \frac{\text{TP}(t)}{\text{TP}(t) + \text{FP}(t)}$$

- **Recall:** Mide el porcentaje de casos positivos encontrados, es decir, la proporción de objetos detectados frente al total de objetos a detectar.

$$\text{Recall}(t) = \frac{\text{TP}(t)}{\text{TP}(t) + \text{FN}(t)}$$

Una *precision alta* indica que muchas de las predicciones hechas son correctas, mientras que un *recall alto* significará que se han encontrado la mayoría de los objetos (es decir, existirá un bajo porcentaje de falsos negativos).

4. Modelo Inicial

Como primer paso, se crea la clase **NucleusDataset**, la cual es necesaria para poder utilizar la clase *Dataset* original que permitirá realizar el entrenamiento del modelo. En esencia, esta clase se adapta a nuestro problema a la hora de cargar los datos.

En ella se ha creado el método ***cargar_set***, cuya función es añadir el *id* y la *ruta* de cada una de las imágenes como atributos de la clase. Gracias a ello, se tendrá guardada toda la información necesaria para poder cargar y visualizar una imagen en cualquier momento.

De igual manera, se tiene el método ***load_mask***, el cual es una sobrecarga del método original. Se encarga de, dado el id de una imagen, buscar en los directorios y cargar como una matriz de booleanos todas las máscaras asociadas a una imagen. Se elige hacer la carga de las máscaras como una matriz de booleanos en vez de guardar sus valores [0-1] para agilizar el proceso. Además, se devuelve un array de 'unos' que indica el número de máscaras que hay para una única imagen.

Se prueba a visualizar varias imágenes aleatorias:

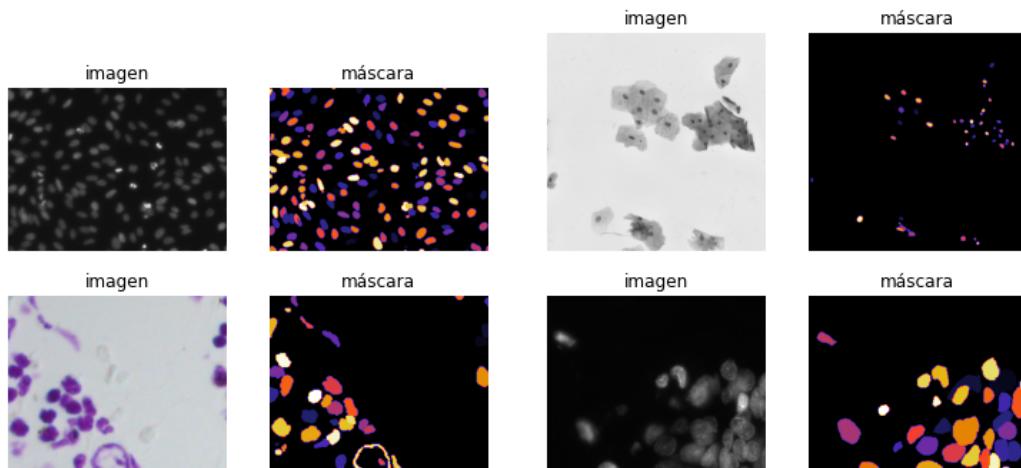


Figura 4: Ejemplo de imágenes cargadas junto a sus máscaras.

Para esta visualización, se ha decidido solapar todas las máscaras de una misma imagen en una sola. Para ello, se asigna a '-1' aquellas zonas donde no se encuentre máscara, mientras que se le asignará un valor numérico correspondiente a la clase encontrada (que, en nuestro caso, siempre será '1', ya que solamente se lidia con una única clase). Tras esto, se realiza la suma de todas las máscaras de la imagen para unirlas todas en una.

Una vez cargados los datos, se debe crear el propio modelo. Para ello, se realizan varios **cambios en los hiperparámetros** de la configuración descrita en el apartado 3.1 de la práctica:

- **Num_classes:** Este parámetro depende del problema a afrontar. En nuestro caso, al solo querer detectar núcleos, el valor correspondiente será '2', englobando a la clase '*núcleo*' y el '*background*', es decir, aquello que *no* pertenece a la clase.
- **Backbone.** Aunque originalmente la arquitectura de la red está establecida a *ResNet101*, se elige utilizar *Resnet50*, estudiada y utilizada en prácticas, con el objetivo de ofrecer la posibilidad de realizar el entrenamiento con toda la red sin añadir una carga excesiva de trabajo.
- **Batch_size.** A pesar de no ser un parámetro explícito, se calcula multiplicando el valor de *images_per_gpu* con *gpu_count*. A mayor batch_size, un mayor número de muestras serán propagadas en la red neuronal, lo que requiere de cálculos intermedios más grandes, los cuales deben almacenarse en GPU. Es por ello que se decide mantener ambos valores a '1' para no encontrar problemas de memoria.
- **Steps_per_epoch:** Número de pasos de entrenamiento por época. Se ha decidido seguir la misma filosofía que la realizada en prácticas anteriores, de dividir el conjunto de datos de entrenamiento en un 90 % de las imágenes para el entrenamiento y el 10 % restante para la validación. Por ello, se realizará $\text{len}(x_{\text{valid}}) * 0.9 / \text{batch_size}$ pasos.
- **Validation_steps:** Número de pasos de validación a ejecutar al final de cada época de entrenamiento. Se considera el 10 % de imágenes aleatorias restantes seleccionadas para validación y se realizará $\text{len}(x_{\text{train}}) * 0.1 / \text{batch_size}$ pasos.
- **Image_resize_mode:** De entre las opciones de redimensión (square, pad64 o crop), se elige dejar todas las imágenes a un tamaño estándar y, por ello, se elige '*square*' como modo de redimensión. De esta manera, el lado más pequeño de las imágenes son redimensionados a *Image_min_dim*, asegurándose que el lado más grande no sea mayor que *Image_max_dim* y, tras esto, la imagen se rellena con ceros para convertirla en un cuadrado.

- **Image_min_dim e Image_max_dim:** En la bibliografía correspondiente [3], se menciona que el tamaño estándar es de 800 x 800; sin embargo, a la hora de realizar la implementación de nuestra configuración, se ha necesitado que la dimensión sea divisible entre dos. Por otra parte, unas dimensiones de 800x800 han causado problemas de memoria a la hora de realizar el entrenamiento, por lo que se ha decidido hacer uso de imágenes más pequeñas (128 x 128).
- **RPN_anchor_scales:** Es la lista de dimensiones (en píxeles) del *Region Proposal Network* que localizará los distintos núcleos. Como los valores deben ser divisibles entre las dimensiones de la imagen, se han elegido valores que no superen la mitad de la dimensión de la imagen (ya que en ningún caso se encontrarán núcleos tan grandes como para ocupar tanto espacio en la imagen): (8, 16, 32, 64).
- **Post_NMS_RoIs_training y Post_NMS_RoIs_inference:** Se establecen a 1000 y 2000 respectivamente. Esta configuración limita la cantidad de *Regiones de Interés, RoIs* tras realizar la *No supresión máxima*. De estos RoIs, se elige un subconjunto de *train_RoIs_per_image* para entrenar en la siguiente etapa, por lo que, aunque nos interesa un valor alto, se tiene en cuenta que cuanto mayor sea, más memoria necesitará.
- **RPN_train_anchors_per_image:** Para ser posible entrenar el RPN, lo primero es generar un número de cajas delimitadoras (Bounding Box) de forma que se devuelva una caja que contenga la zona del objeto detectado (en este caso, un núcleo). El mecanismo que se usa para delimitar esta zona se denomina *anchors boxes*. Cada pixel de la imagen se considera un *anchor*. En este parámetro se elige el conjunto de *Anchors* que rodearía la zona de interés. Las pruebas han sido con imágenes de 512x512, y en una imagen pueden aparecer varios núcleos mucho más pequeños que este tamaño, por lo que en este caso el número elegido es la mitad de la dimensión de la imagen.
- **Mean_pixel:** Hace referencia a la media de los píxeles en RGB. Para realizar este cálculo, se ha calculado para cada imagen la proporción de cada uno de los tres canales (R,G,B) y se ha realizado la media con todas las imágenes (del conjunto de entrenamiento-validación). Los resultados obtenidos han sido:

MEAN PIXELS: [43.51683223 39.54457796 48.20850446]

- **Train_RoIs_per_image:** Número de RoIs por imagen que se utiliza para la clasificación. Se establece a *256*, un valor mayor que el de la implementación base (200) pero menor que el indicado por el paper (512). Se disminuye el valor con respecto al paper porque se indica que el RPN de Mask R-CNN no suele generar suficientes propuestas positivas como para que sea necesario utilizar un valor tan alto.
- **Detection_Max_Instances y Max_GT_Instances:** En la literatura se establecen a 100, pero para nuestro problema estos valores han sido incrementados a *500* ya que, sin poder saberlo con total seguridad, varias de las imágenes parecen superar los 100 núcleos, por lo que se incrementa el valor para que no queden instancias sin detectar.
- **RPN_NMS_Threshold:** Este parámetro se usa para reducir el número de cajas delimitadoras generadas en el reconocimiento de una zona de interés que han quedado superpuestas unas a otras. Es el umbral del método denominado *Non-Maximum Suppression*, que elimina aquellas cajas que se colocan encima de otras cajas que tienen un mejor resultado, dejando por tanto las opciones deseadas. Se establecerá a 0.9 para que, en caso de que le resulte costoso al modelo encontrar los núcleos, sea seguro que encuentre bounding boxes en la zona de interés.

Una vez realizada la configuración, se debe de crear el modelo:

```
modelo = modellib.MaskRCNN(mode, config, model_dir)
```

donde:

- **mode** indica el modo en que se va a crear el modelo, pudiendo ser '*training*' para el entrenamiento o '*inference*' para la detección.
- **config** guardará la configuración descrita anteriormente.
- **model.dir** indica el directorio donde se van a guardar los pesos y distintos logs durante la ejecución. **NOTA:** No se ha conseguido encontrar una manera para no tener que crear este archivo, por lo que la única alternativa disponible es, al final de la ejecución, realizar un borrado de todo su contenido y el mismo.

Se usarán como pesos iniciales unos pesos pre-entrenados de **COCO**. Para poder cargarlos, se deben excluir las últimas capas: *mrcnn_class_logits*, *mrcnn_bbox_fc*, *mrcnn_bbox* y *mrcnn_mask* para poder tener un número de clases que coincida (Para nuestro problema se tiene una única clase 'núcleo' mientras que COCO posee

80 clases distintas).

Aunque se partan de estos pesos, se ha decidido realizar un entrenamiento simple de las cabeceras de la red durante *una época* para adaptarlos a nuestro problema. Tras el entrenamiento, se utilizan los pesos obtenidos para generar el modelo en modo *inferencia*, manteniendo la configuración, con el objetivo de realizar la **detección** de los objetos.

La idea es realizar las detecciones sobre las imágenes del conjunto de validación, sin embargo, se ha intentado obtener los resultados del mismo conjunto de imágenes para poder ver las diferencias entre resultados. Cada una de las imágenes es cargada mediante el método *load_image_gt*, proporcionado por la librería **modellib**:

```
image, dimensions, gt_class_id, gt_bbox, gt_mask =  
modellib.load_image_gt(dataset, config, image_id, use_mini_mask)
```

donde los argumentos:

- **dataset** es el conjunto de datos proporcionado (validación).
- **config** es la configuración del modelo.
- **image_id** es el id de la imagen a cargar.
- **use_mini_mask** se establece a False para que se devuelvan las máscaras con el mismo tamaño de la imagen.

y devuelve:

- **image** es la imagen a cargar.
- **dimensions** son los metadatos (dimensiones) de la imagen a cargar.
- **gt_class_id** son los ids de las clases de las máscaras (en nuestro caso será un vector de 'unos').
- **gt_bbox** son los bounding boxes formados por [número de instancia, (y1,x1,y2,y2)].
- **gt_mask** hace referencia a [height, width, número de instancia] de la máscara.

La imagen cargada y sus dimensiones son los parámetros necesarios para llamar al método *detect_molded* que realizará la detección de los objetos. Como resultado, se obtiene un vector que contiene los id de las clases, bounding boxes y dimensiones de las máscaras que ha detectado. Estos datos, junto a estos mismos parámetros obtenidos de la carga de la imagen y sus máscaras, servirán para mostar los resultados:

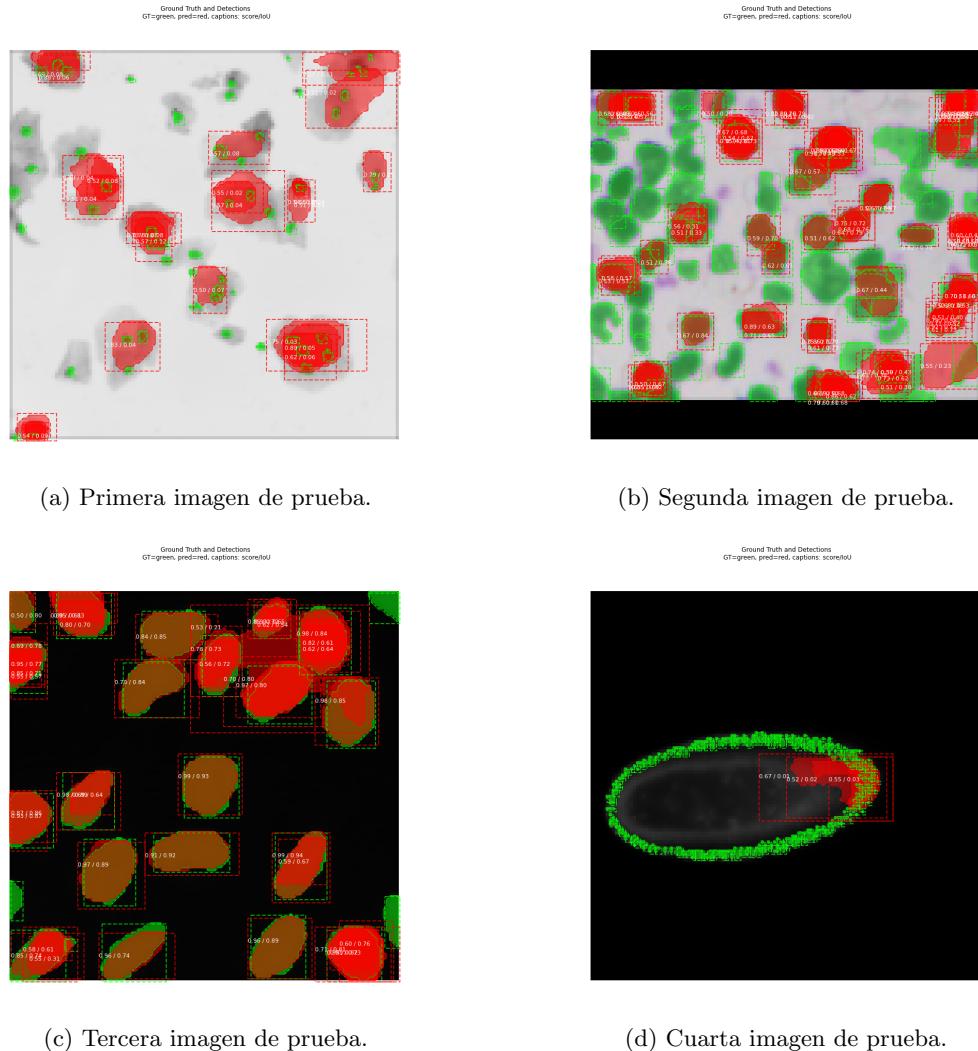


Figura 5: Resultados obtenidos tras entrenar las cabeceras de la red durante una época.

Por cada imagen se encuentran algunas zonas rodeadas de color verde y otras de color rojo. Como se puede deducir, las **zonas verdes**, las cuales marcan mucha exactitud, rodean los núcleos de cada una de las imágenes, esto es, el ground truth. Por otro lado, las **manchas rojas** son las predicciones obtenidas de la detección realizada por el modelo.

Como se puede observar, en general no se ha obtenido buenos resultados. A priori, es evidente que el modelo funciona mejor con aquellos núcleos de mayor tamaño (como se puede ver en la tercera imagen). Sin embargo, los núcleos pequeños y aquellos que se encuentran en las esquinas no son detectados. Este hecho se ve

claramente en la primera y cuarta imágenes, donde el modelo realiza predicciones totalmente incorrectas, marcando como un único núcleo zonas donde se encuentran varios (o ninguno) de ellos.

Para estudiar los resultados con mayor exactitud, se decide calcular la métrica **Average Precision**, explicada en el segundo apartado. Para ello, se ha creado el método *calcularAveragePrecision*, el cual hace uso de un método proporcionado por la librería **utils** llamado *compute_matches* para calcular las correspondencias entre las verdaderas localizaciones de la máscara y las coordenadas predichas por el método *detect*:

```
gt_match, pred_match, overlaps = utils.compute_matches(gt_bbox,
    gt_class_id, gt_mask, pred_boxes, pred_class_ids, pred_scores,
    pred_masks, iou_threshold)
```

el método devuelve:

- **gt_match** vector unidimensional que, para cada caja de ground truth, indica el índice de la caja de predicción con la que ha hecho correspondencia.
- **pred_match** vector unidimensional que, para cada caja de predicción, indica el índice del ground truth con la que ha hecho correspondencia.
- **overlaps** son los solapamientos de la *Intersección sobre la Unión (IoU)*.

Con estas correspondencias, se hace posible calcular los valores de precisión y recall de manera que, como se han indicado en las fórmulas del segundo apartado, se realizará, para ambos criterios de evaluación , la *suma acumulativa* de aquellos valores de *pred_match* con los que se hayan encontrado correspondencia (es decir, aquellos que tienen un valor mayor que '-1') (verdaderos positivos) y, tras esto, el resultado obtenido se dividirá entre el número de predicciones, y el número de ground truths, para precision y recall, respectivamente.

Una vez calculados ambos valores, para calcular el valor de *Average Precision* se recorren los valores de precision en orden decreciente y se buscan aquellos puntos donde el recall cambia de valor. De esta manera, y atendiendo definición expuesta en *VOOC (2007)* [7], se atendería a la fórmula:

$$AP = \frac{1}{11} \sum_{r \in (0,0,1,\dots,1)} P_{interpr}(r)$$

donde:

$$P_{interpr}(r) = \max_{\tilde{r}: \tilde{r} \geq r} p(r)$$

Los valores obtenidos para las imágenes de la Figura 5 son:

Umbral (IoU)	Average Precision
0.50	0.000
0.55	0.000
0.60	0.000
0.65	0.000
0.70	0.000
0.75	0.000
0.80	0.000
0.85	0.000
0.90	0.000
0.95	0.000
mAP [0.50-0.95]	0.000

Cuadro 2: Average Precision para varios umbrales de IoU para la **Primera imagen**.

Umbral (IoU)	Average Precision
0.50	0.171
0.55	0.171
0.60	0.153
0.65	0.101
0.70	0.059
0.75	0.023
0.80	0.014
0.85	0.000
0.90	0.000
0.95	0.000
mAP [0.50-0.95]	0.069

Cuadro 3: Average Precision para varios umbrales de IoU para la **Segunda imagen**.

Umbral (IoU)	Average Precision
0.50	0.779
0.55	0.779
0.60	0.779
0.65	0.765
0.70	0.765
0.75	0.589
0.80	0.505
0.85	0.244
0.90	0.092
0.95	0.000
mAP [0.50-0.95]	0.530

Cuadro 4: Average Precision para varios umbrales de IoU para la **Tercera imagen**.

Umbral (IoU)	Average Precision
0.50	0.000
0.55	0.000
0.60	0.000
0.65	0.000
0.70	0.000
0.75	0.000
0.80	0.000
0.85	0.000
0.90	0.000
0.95	0.000
mAP [0.50-0.95]	0.000

Cuadro 5: Average Precision para varios umbrales de IoU para la **Cuarta imagen**.

Atendiendo a las gráficas, se puede observar como tanto la primera como cuarta imagen tienen un *mAP* del 0 %. Esto es, porque todo lo que ha detectado en la imagen han sido falsos positivos, es decir, nada de lo detectado es un núcleo en la imagen. Por otro lado, aunque para la segunda y tercera imagen parece tomar valores algo mejores, siguen siendo muy deficientes. Esto se puede observar claramente en la imagen de la segunda imagen en la Figura 5: aunque haya núcleos detectados correctamente, sigue existiendo un alto número de falsos negativos, es decir, núcleos existentes que no han sido detectados.

Estos resultados son una indicación de que entrenar partiendo de los pesos de **COCO** es una buena aproximación, sin embargo, aún se necesita un mayor entrenamiento o un refinamiento distinto de los hiperparámetros para poder afirmar que se obtiene un buen resultado.

5. Mejoras realizadas

5.1. Aumentar el número de épocas

Dados los resultados iniciales, se considera que una única época de entrenamiento es insuficiente como para poder adaptarse correctamente a nuestro problema. Es por ello que se decide volver a entrenar las **cabeceras** de la red, esta vez durante 20 épocas.

Además, al entrenar durante varias épocas, resulta interesante poder ver la evolución de los valores de la función de pérdida para el conjunto de entrenamiento y validación. Atendiendo a la literatura [3], formalmente la función de pérdida L , se define como:

$$L = L_{cls} + L_{box} + L_{mask}$$

donde L_{cls} es la pérdida de clasificación, L_{box} es la pérdida en bounding box y L_{mask} es definida como la pérdida promedio de la *entropía cruzada binaria* [9]:

$$H_p(q) = -\frac{1}{N} \sum_{i=1}^N y_i \cdot \log(p(y_i)) + (1 - y_i) \cdot \log(1 - p(y_i))$$

La evolución de estos valores son calculados durante el entrenamiento. Debido a la implementación de Mask R-CNN, estos valores no pueden guardarse como variables del programa. Sin embargo, en el directorio indicado en la creación del modelo se guardan todos los pesos calculados durante cada una de las épocas, así como un archivo de *eventos*. Gracias a esto, se ha podido hacer uso de la herramienta **tensorboard** para mostrar las gráficas de la evolución de la función de pérdida mediante la importación de los pesos y eventos.

NOTA: Esta herramienta funciona para el entorno de *Colab*. Debido a que la realización del proyecto ha sido realizada en *Spyder*, la ejecución de tensorboard **estará comentada en el código**. Para la obtención de las gráficas, se ha tenido que crear un entorno de colab que únicamente leyese la información necesaria almacenada en Drive; por ende, en la ejecución del programa **no se visualizarán este tipo de gráficas**.

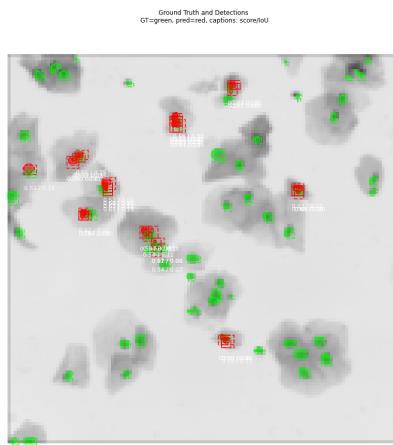
La ejecución de la herramienta se realiza mediante el código:

```

1 %load_ext tensorboard # Carga la extension de tensorboard
2 !kill 729
3 %tensorboard --logdir="ruta/"
```

`!kill 729` es necesario, ya que tras el reiterado uso de la herramienta, a veces se generan problemas al mostrar los resultados. El mensaje de error de %tensorboard hace referencia a que el proceso actual (en nuestro caso, 729, pero este número depende del proceso que indique Colab que está ocupado y sea necesario finalizarlo) está ocupado; por lo que para poder evitar este problema se termina la instancia antes de la ejecución. Ambas sentencias deben estar en la misma celda de Colab.

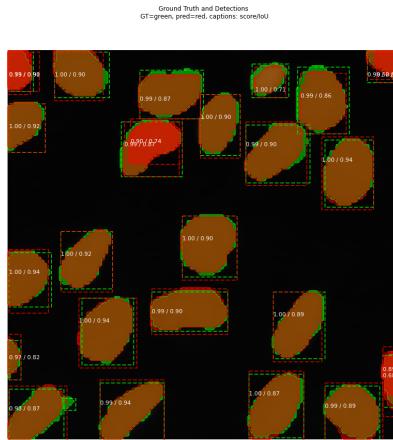
Los resultados obtenidos son:



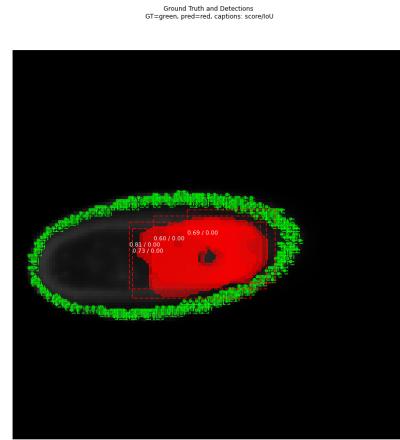
(a) Primera imagen de prueba.



(b) Segunda imagen de prueba.

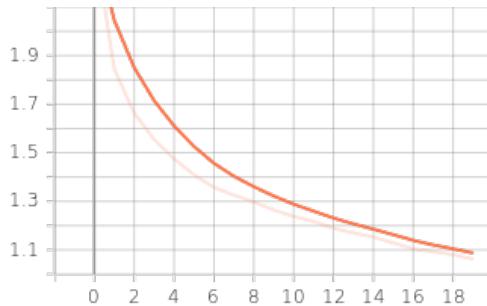


(c) Tercera imagen de prueba.

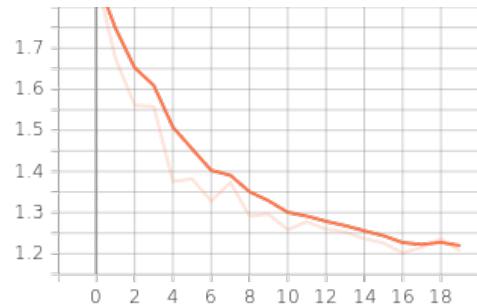


(d) Cuarta imagen de prueba.

Figura 6: Resultados obtenidos tras entrenar aumentar a 20 el número de épocas.



(a) Evaluación de la función de pérdida en entrenamiento.



(b) Evaluación de la función de pérdida en validación.

Figura 7: Evaluación de la función de pérdida a lo largo de las épocas (Tensorboard).

y las métricas de evaluación obtenidas son:

Umbral (IoU)	Average Precision
0.50	0.014
0.55	0.000
0.60	0.000
0.65	0.000
0.70	0.000
0.75	0.000
0.80	0.000
0.85	0.000
0.90	0.000
0.95	0.000
mAP [0.50-0.95]	0.002

Cuadro 6: Average Precision para varios umbrales de IoU para la **Primera imagen**.

Umbral (IoU)	Average Precision
0.50	0.501
0.55	0.415
0.60	0.373
0.65	0.253
0.70	0.182
0.75	0.063
0.80	0.016
0.85	0.001
0.90	0.000
0.95	0.000
mAP [0.50-0.95]	0.180

Cuadro 7: Average Precision para varios umbrales de IoU para la **Segunda imagen**.

Umbral (IoU)	Average Precision
0.50	0.957
0.55	0.957
0.60	0.957
0.65	0.957
0.70	0.957
0.75	0.881
0.80	0.881
0.85	0.802
0.90	0.436
0.95	0.000
mAP [0.50-0.95]	0.778

Cuadro 8: Average Precision para varios umbrales de IoU para la **Tercera imagen**.

Umbral (IoU)	Average Precision
0.50	0.000
0.55	0.000
0.60	0.000
0.65	0.000
0.70	0.000
0.75	0.000
0.80	0.000
0.85	0.000
0.90	0.000
0.95	0.000
mAP [0.50-0.95]	0.000

Cuadro 9: Average Precision para varios umbrales de IoU para la **Cuarta imagen**.

Los resultados muestran cómo aumentar el número de épocas mejora, como era de esperar, el desempeño del modelo. Como observación positiva, a parte de la evidencia de que aquellos núcleos de gran tamaño son generalmente bien detectados, se puede observar (sobre todo en la Tercera imagen), como a diferencia de los resultados anteriores, ahora detecta aquellos núcleos que se encuentran en las esquinas. A pesar de ello, aún se sigue encontrando un gran número de falsos negativos, sobre todo cuando se trata de núcleos de menor tamaño; es por eso que para la primera y cuarta imagen se sigue obteniendo un mAP de (o muy aproximado a) cero.

Cabe destacar, visualizando las gráficas que muestran la evolución de los valores de pérdida, como tiene una tendencia descendente y no parece generarse *overfitting*. Se puede observar que, por cada una de ellas, aparecen dos líneas en la misma gráfica: ambas hacen referencia al mismo valor, sin embargo, en nuestro caso siempre se hará referencia a la línea de color más claro, la cual representa las fluctuaciones reales del entrenamiento. La línea de color más oscuro es la curva con un factor de suavizado, para aquellos que únicamente quieran ver la tendencia de los valores.

5.2. Data Augmentation

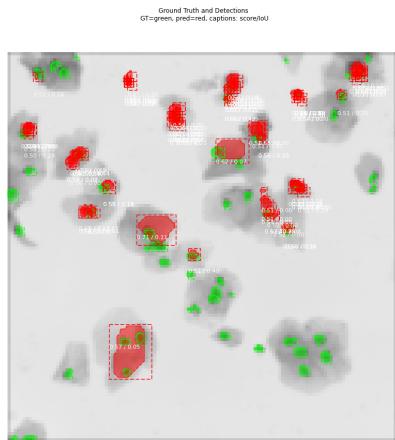
En base a mejorar los resultados anteriores, la mejora planteada consiste en aumentar los datos. Esto conlleva un efecto importante: aumenta la ***diversidad*** de los datos, lo que significa que en cada etapa el modelo se encontrará con una versión diferente y artificial de las imágenes originales. Se cree que este aumento de datos es necesario y que, con bastante seguridad, añadirá una mejora extra a los resultados; ya que una mayor diversidad en los datos implica la varianza del modelo al mejorar la ***generalización***.

Inicialmente, se intentó utilizar el método *ImageDataGenerator*, de igual manera que se utilizó en prácticas, pero la implementación de Mask R-CNN solo permite utilizar la clase **imagaug** [8]. Se añaden las siguientes transformaciones:

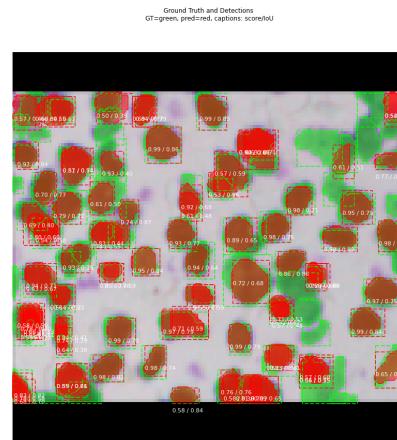
- **Volteos:** Como su nombre indica, básicamente volteá tanto las filas como las columnas de una imagen. Se aplican tanto volteos *horizontales* como *verticales*, ya que teniendo en cuenta que se trabaja con imágenes celulares, donde la gran mayoría de formas encontradas son redondeadas o elípticas, no supondrá gran diferencia voltearlas. Para ello, se utiliza *Fliplr(0.5)* y *Flipud(0.5)*.
- **Multiplicación de píxeles:** Se utiliza el método *Multiply* para multiplicar todos los píxeles de una imagen con un valor aleatorio muestreado una vez por imagen. En nuestro caso, se utilizará el rango *[0.8, 1.5]*. Se ha decidido utilizar esta transformación ya que suele utilizarse para añadir/quitar luminosidad en las imágenes, siendo una aplicación interesante sobre imágenes celulares, las cuales pueden ser fotografiadas en distintas condiciones similares a esta.
- **Rotaciones:** Al igual que ocurre con los volteos, las imágenes del conjunto de datos siguen siendo interpretables si se les aplica rotaciones. Aunque es posible que no sea una transformación demasiado significativa, pues los núcleos no tienen formas demasiado distintivas, estos pueden encontrarse agrupados o esparcidos por cualquier parte de la imagen, por lo que el uso de rotaciones puede ser útil. Se realizarán rotaciones de *90, 180 y 270 grados*.
- **Desenfoque Gaussiano:** Se realiza desenfoques Gaussianos con sigmas que varían de entre *0* y *5*. La importancia de realizar esta transformación reside en que la introducción deliberada de imperfecciones en el conjunto de datos es esencial para hacer que los modelos sean más resistentes, ya que en aplicaciones de la vida real, es normal que las fotos a tratar no hayan sido hechas en condiciones ideales, y la aparición de ruido o desenfoque en ellas son algunas de las imperfecciones más perjudiciales.

El enfoque utilizado para aplicar estas transformaciones se ha basado en utilizar el utilizar el método de imgaug *SomeOf(0,2)* para aplicar entre 0 y 2 de las

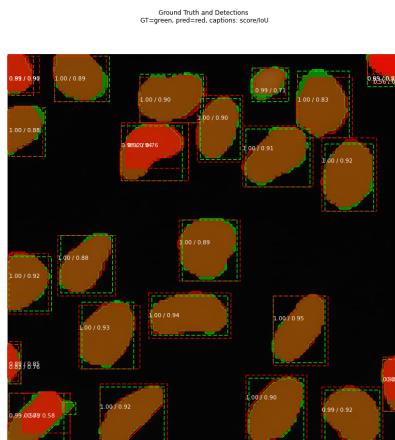
transformaciones nombradas aleatoriamente a la vez, y el método *OneOf* para aplicar una única rotación cada vez (ya que no tiene sentido aplicar dos rotaciones distintas sobre una misma imagen).



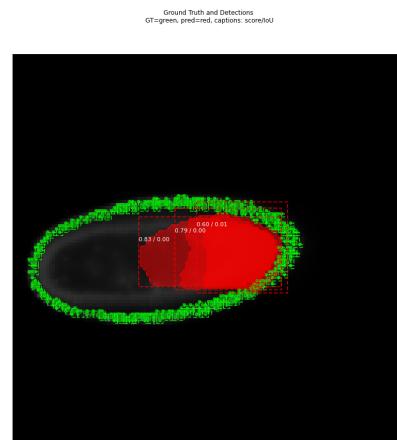
(a) Primera imagen de prueba.



(b) Segunda imagen de prueba.

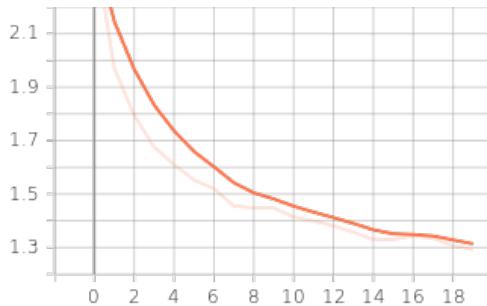


(c) Tercera imagen de prueba.

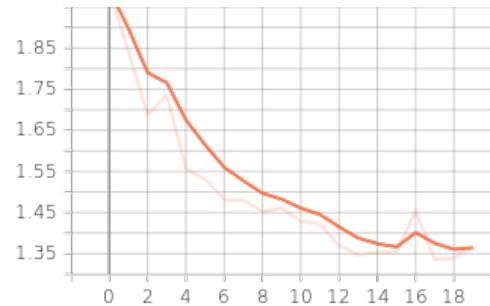


(d) Cuarta imagen de prueba.

Figura 8: Resultados obtenidos tras añadir Data Augmentation.



(a) Evaluación de la función de pérdida en entrenamiento.



(b) Evaluación de la función de pérdida en validación.

Figura 9: Evaluación de la función de pérdida a lo largo de las épocas (Tensorboard).

Métricas de evaluación:

Umbral (IoU)	Average Precision
0.50	0.025
0.55	0.006
0.60	0.003
0.65	0.002
0.70	0.001
0.75	0.001
0.80	0.001
0.85	0.001
0.90	0.001
0.95	0.001
mAP [0.50-0.95]	0.004

Cuadro 10: Average Precision para varios umbrales de IoU para la **Primera imagen**.

Umbral (IoU)	Average Precision
0.50	0.563
0.55	0.525
0.60	0.499
0.65	0.408
0.70	0.358
0.75	0.214
0.80	0.092
0.85	0.013
0.90	0.000
0.95	0.000
mAP [0.50-0.95]	0.267

Cuadro 11: Average Precision para varios umbrales de IoU para la **Segunda imagen**.

Umbral (IoU)	Average Precision
0.50	0.955
0.55	0.955
0.60	0.955
0.65	0.955
0.70	0.955
0.75	0.906
0.80	0.906
0.85	0.797
0.90	0.331
0.95	0.000
mAP [0.50-0.95]	0.771

Cuadro 12: Average Precision para varios umbrales de IoU para la **Tercera imagen**.

Umbral (IoU)	Average Precision
0.50	0.000
0.55	0.000
0.60	0.000
0.65	0.000
0.70	0.000
0.75	0.000
0.80	0.000
0.85	0.000
0.90	0.000
0.95	0.000
mAP [0.50-0.95]	0.000

Cuadro 13: Average Precision para varios umbrales de IoU para la **Cuarta imagen**.

La adición de un aumento de la diversidad de los datos ha conllevado una ligera mejoría, en general, en las métricas obtenidas. Aunque no ha sido suficiente como para encontrar ningún núcleo en la cuarta imagen, se observa como detecta algo mejor los pequeños núcleos de la Primera fotografía. Aunque, por el contrario, las formas de las detecciones no llegan a encajar muy bien con las máscaras verdaderas. Aún así, debido a la mejora en los criterios numéricos, se decide mantener esta mejora.

5.3. Ajuste fino de toda la red

El siguiente modelo se entrena bajo las mismas condiciones de configuración que las pruebas anteriores. El entrenamiento se realiza durante 20 épocas y con el generador de aumento de datos creado. Sin embargo, se decide realizar un *ajuste fino*, también llamado *fine-tuning*. Aplicar fine tuning permite utilizar redes preentrenadas para reconocer clases con las que no fueron pre-entrenadas originalmente. Por tanto, se reentrenarán *todas* las capas de la red para adaptarla completamente a nuestro problema, generando los siguientes resultados:

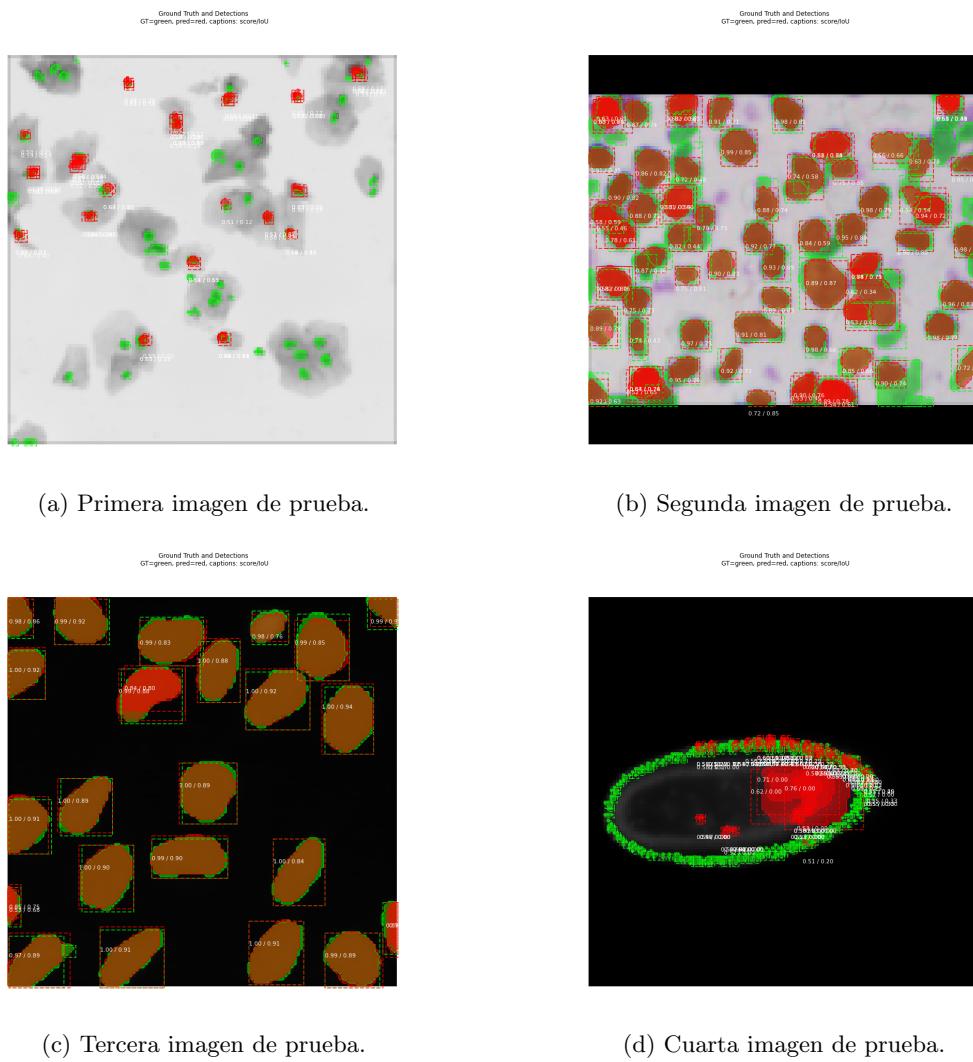
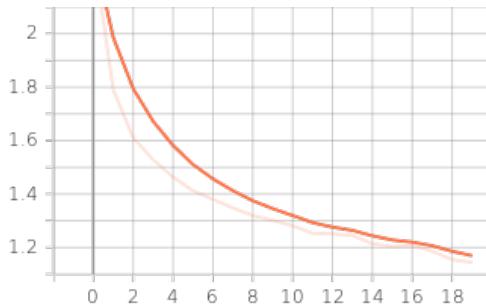
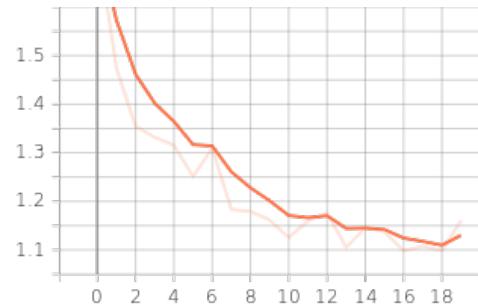


Figura 10: Resultados obtenidos aplicando ajuste fino de la red.



(a) Evaluación de la función de pérdida en entrenamiento.



(b) Evaluación de la función de pérdida en validación.

Figura 11: Evaluación de la función de pérdida a lo largo de las épocas (Tensorboard).

Se obtienen los siguientes valores para las métricas:

Umbral (IoU)	Average Precision
0.50	0.046
0.55	0.019
0.60	0.013
0.65	0.007
0.70	0.001
0.75	0.000
0.80	0.000
0.85	0.000
0.90	0.000
0.95	0.000
mAP [0.50-0.95]	0.009

Cuadro 14: Average Precision para varios umbrales de IoU para la **Primera imagen**.

Umbral (IoU)	Average Precision
0.50	0.635
0.55	0.635
0.60	0.584
0.65	0.510
0.70	0.393
0.75	0.236
0.80	0.106
0.85	0.005
0.90	0.000
0.95	0.000
mAP [0.50-0.95]	0.310

Cuadro 15: Average Precision para varios umbrales de IoU para la **Segunda imagen**.

Umbral (IoU)	Average Precision
0.50	0.958
0.55	0.958
0.60	0.958
0.65	0.958
0.70	0.915
0.75	0.875
0.80	0.831
0.85	0.683
0.90	0.310
0.95	0.000
mAP [0.50-0.95]	0.745

Cuadro 16: Average Precision para varios umbrales de IoU para la **Tercera imagen**.

Umbral (IoU)	Average Precision
0.50	0.009
0.55	0.000
0.60	0.000
0.65	0.000
0.70	0.000
0.75	0.000
0.80	0.000
0.85	0.000
0.90	0.000
0.95	0.000
mAP [0.50-0.95]	0.001

Cuadro 17: Average Precision para varios umbrales de IoU para la **Cuarta imagen**.

Atendiendo a los valores de *mean Average Precision* notamos cierta mejoría en los resultados, a pesar de que se esperaba una diferencia algo mayor. Sin embargo, se puede observar algunos detalles bastante positivos:

Aunque de la sensación de que para la Primera imagen se hayan detectado menos núcleos en comparación al entrenamiento de las cabeceras, esta vez las detecciones realizadas tienen bastante precisión, es decir, se adecúan casi perfectamente a las dimensiones y forma de la máscara verdadera.

Además, para la Cuarta imagen, se ha conseguido que empiece a detectar verdaderos positivos, a la vez que el número de falsos positivos también ha disminuido, aunque aún detecte una gran cantidad de núcleos no existentes.

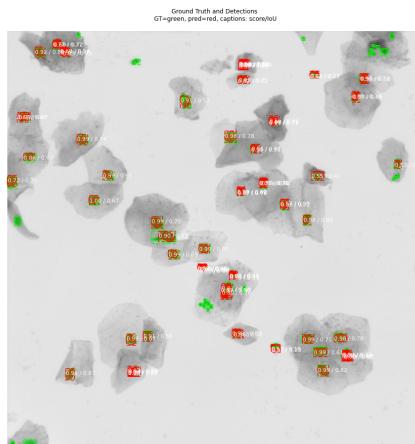
5.4. Aumentar el tamaño de las imágenes

Aunque los resultados del apartado anterior hayan supuesto cierta mejora, se elige abordar el problema desde otra perspectiva: cambiando la configuración previamente hecha y aumentando las dimensiones de la imagen. Este enfoque ha sido un elemento clave en la búsqueda de una mejora en los resultados del problema.

Hasta ahora, se ha configurado el modelo para que las imágenes fuesen de tamaño 128x128. La Red realiza internamente una serie de convoluciones, las cuales eliminan información de **altas frecuencias**, por lo que usar un tamaño tan pequeño en las imágenes puede conllevar que la representación de la imagen sea mucho menos específica y que, por ende, se obtenga un peor desempeño en las predicciones. Por esta misma razón se ha decidido aumentar el tamaño de las imágenes a 512x512.

El modelo se ha mantenido igual, a excepción de que se ha modificado el hiperparámetro *RPN_ANCHOR_SCALES* añadiéndole una escala más, pasando a (8, 16, 32, 64, 128), ya que al aumentar las dimensiones, lógicamente, el tamaño de los núcleos se verá a su vez afectado, resultando en que haya núcleos mayores a tamaño 64.

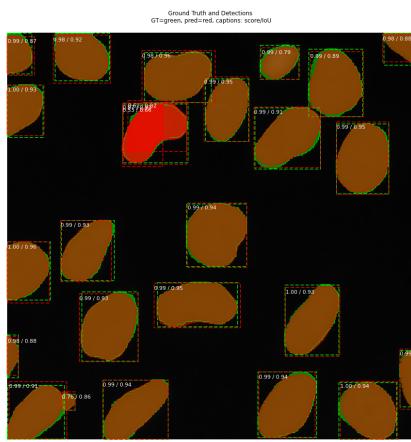
Sin embargo, se advierte correcto indicar que esta decisión en gran medida depende del problema en cuestión. Además, como desventaja, el aumento de tamaño de las imágenes con las que se trabaja puede acarrear problemas de memoria. En nuestro caso, no fue posible entrenar con la primera GPU indicada en la memoria cuando el tamaño de las imágenes pasó a ser de 512x512 ya que se desbordaba la memoria; motivo por el cual no se ha decidido probar con imágenes de dimensiones mayores.



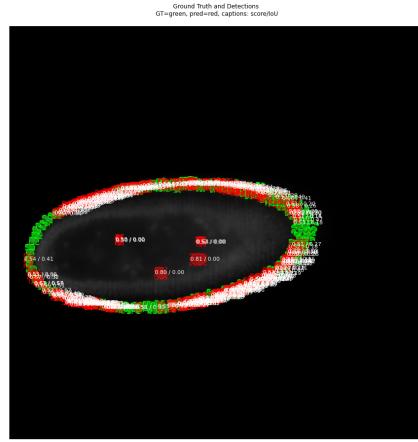
(a) Primera imagen de prueba.



(b) Segunda imagen de prueba.

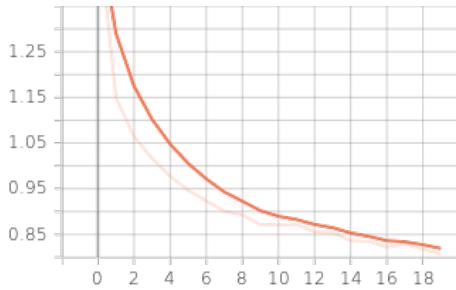


(c) Tercera imagen de prueba.

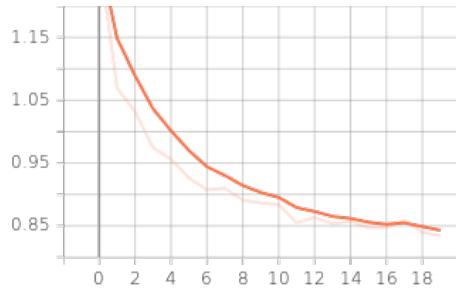


(d) Cuarta imagen de prueba.

Figura 11: Resultados obtenidos al entrenar las cabeceras aumentando a 512x512 las dimensiones de las imágenes.



(e) Evaluación de la función de pérdida en entrenamiento.



(f) Evaluación de la función de pérdida en validación.

Figura 12: Evaluación de la función de pérdida a lo largo de las épocas (Tensorboard).

y las métricas de evaluación obtenidas son:

Umbral (IoU)	Average Precision
0.50	0,459
0.55	0,445
0.60	0,439
0.65	0,364
0.70	0,189
0.75	0,059
0.80	0,014
0.85	0,002
0.90	0,000
0.95	0,000
mAP [0.50-0.95]	0,197

Cuadro 18: Average Precision para varios umbrales de IoU para la **Primera imagen**.

Umbral (IoU)	Average Precision
0.50	0,879
0.55	0,833
0.60	0,812
0.65	0,721
0.70	0,634
0.75	0,466
0.80	0,309
0.85	0,062
0.90	0,002
0.95	0,000
mAP [0.50-0.95]	0,472

Cuadro 19: Average Precision para varios umbrales de IoU para la **Segunda imagen**.

Umbral (IoU)	Average Precision
0.50	1
0.55	1
0.60	1
0.65	1
0.70	1
0.75	1
0.80	0,927
0.85	0,927
0.90	0,689
0.95	0,064
mAP [0.50-0.95]	0,861

Cuadro 20: Average Precision para varios umbrales de IoU para la **Tercera imagen**.

Umbral (IoU)	Average Precision
0.50	0,062
0.55	0,029
0.60	0,019
0.65	0,005
0.70	0,002
0.75	0,000
0.80	0,000
0.85	0,000
0.90	0,000
0.95	0,000
mAP [0.50-0.95]	0,012

Cuadro 21: Average Precision para varios umbrales de IoU para la **Cuarta imagen**.

En este caso, como se esperaba, se puede apreciar una mejora bastante notoria en las imágenes de muestra. A nivel visual, es posible apreciar como ha detectado mucho mejor los núcleos, en especial en las imágenes donde estos salen más pe-

queños. Con esta nueva configuración, el modelo se adapta a los distintos tamaños y formas de los núcleos, generando detecciones que cubren prácticamente de manera idéntica las máscaras verdaderas. Además, el número de falsos positivos ha decrecido en gran medida; aunque por desgracia aún encuentre algunos (pocos) en la Cuarta imagen.

Esto se debe a lo comentado anteriormente, los núcleos pequeños dentro de una imagen no se pierden tan fácilmente en comparación a cuando las imágenes eran de un tamaño menor. Se mantiene una mayor cantidad de **altas frecuencias** con las que es posible identificar esos elementos más pequeños, generando un entrenamiento mucho más preciso y con unos resultados mucho mejores. La mayoría de los núcleos son representados en todas las imágenes.

Puede apreciarse como en la cuarta imagen, el resultado sigue siendo pobre. De las cuatro imágenes es la más compleja ya que contiene todos los núcleos muy próximos entre sí y muy pequeños. El hecho de que un conjunto de núcleos estén estrechamente agrupados añade una dificultad extra al problema, ya que un único n úcleo es fácil de segmentar, pero en el caso de las agrupaciones, la red debe aprender a separar las diferentes instancias.

6. Predicciones

Una vez comentadas las modificaciones realizadas a lo largo del proyecto, se ha considerado buena idea mostrar los resultados de *Average Precision* y *Mean Average Precision* en Excel para mostrar visualmente los resultados obtenidos:

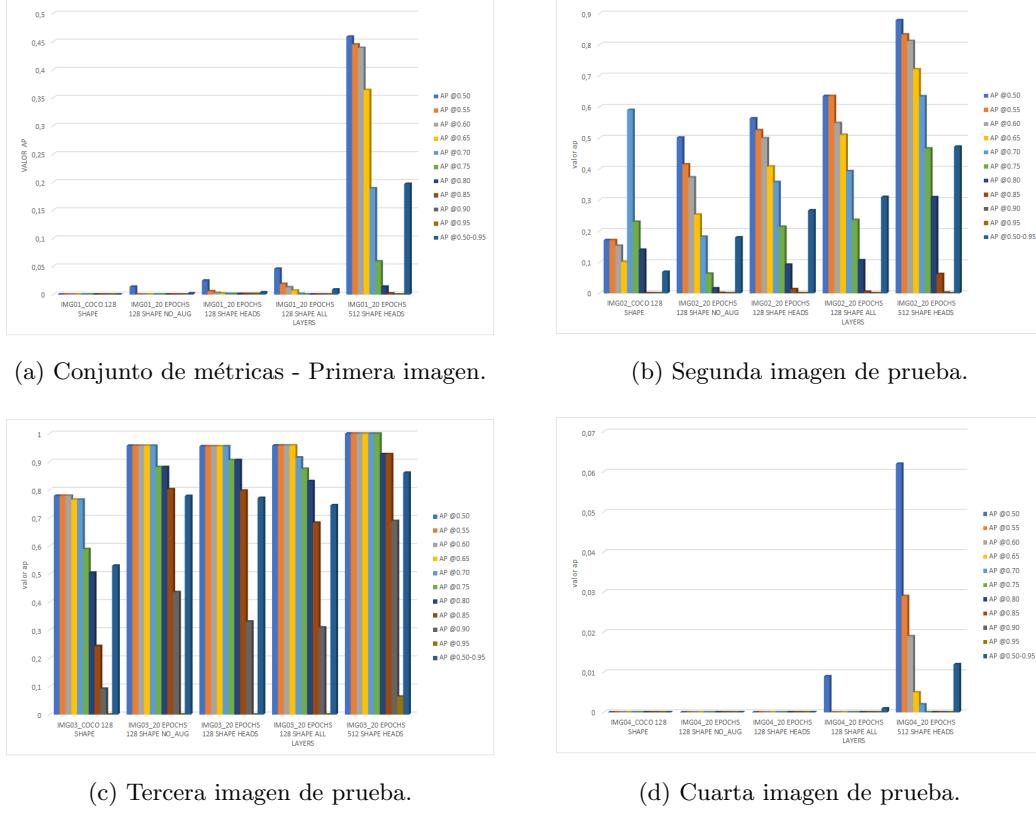


Figura 13: Visualización de las métricas obtenidas a lo largo del proyecto.

Atendiendo a los valores obtenidos en las métricas, y como se ha ido comentando a lo largo de la memoria, no hay duda alguna de que la última mejora realizada proporciona unos resultados más precisos, por tanto se ha pasado a realizar una serie de predicciones sobre imágenes aleatorias del *conjunto de test*, las cuales ya no tienen máscaras con las qué poder comprobar. Se ha obtenido el ID de las imágenes aleatorias para poder seleccionar la imagen original y realizar la comprobación de una forma más detallada sobre la precisión de la predicción:

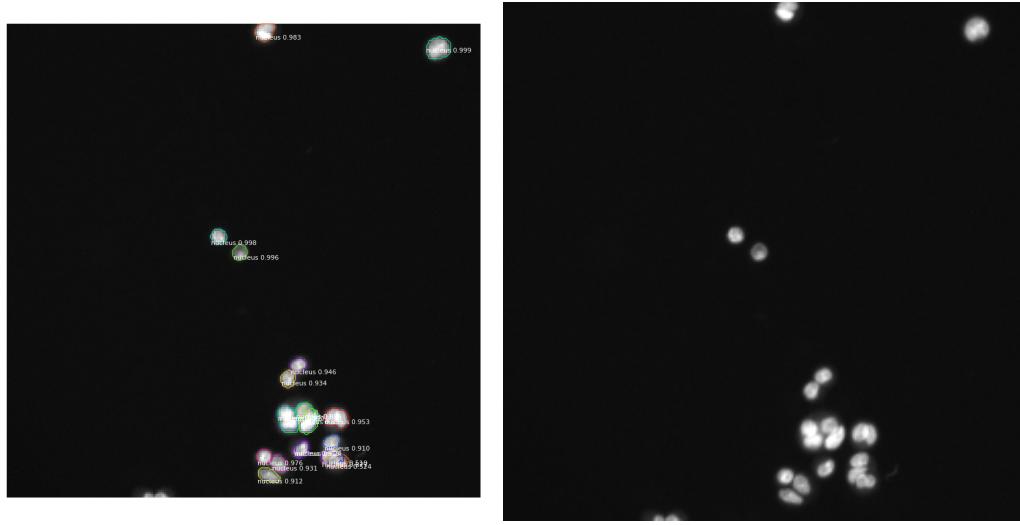


Figura 14: Predicción sobre una imagen aleatoria elegida del conjunto de test.

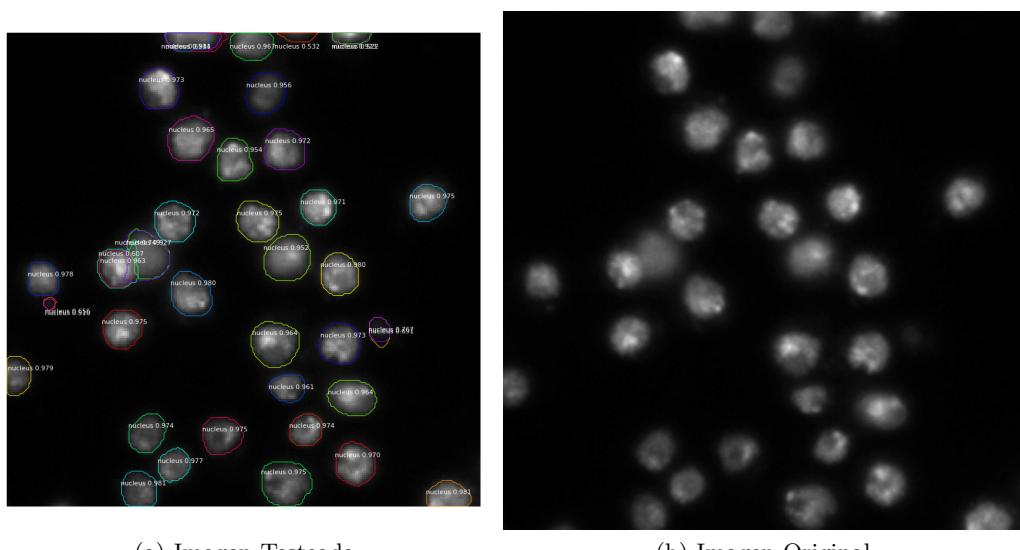


Figura 15: Predicción sobre una imagen aleatoria elegida del conjunto de test.

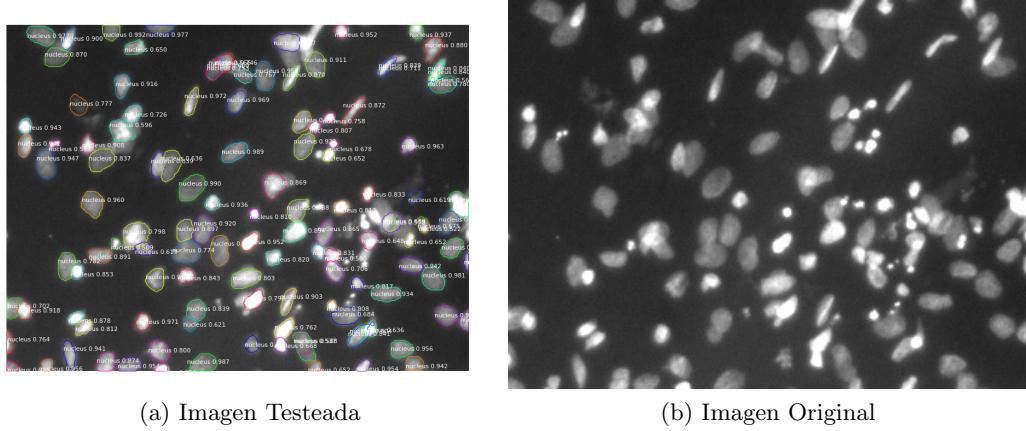


Figura 16: Predicción sobre una imagen aleatoria elegida del conjunto de test.

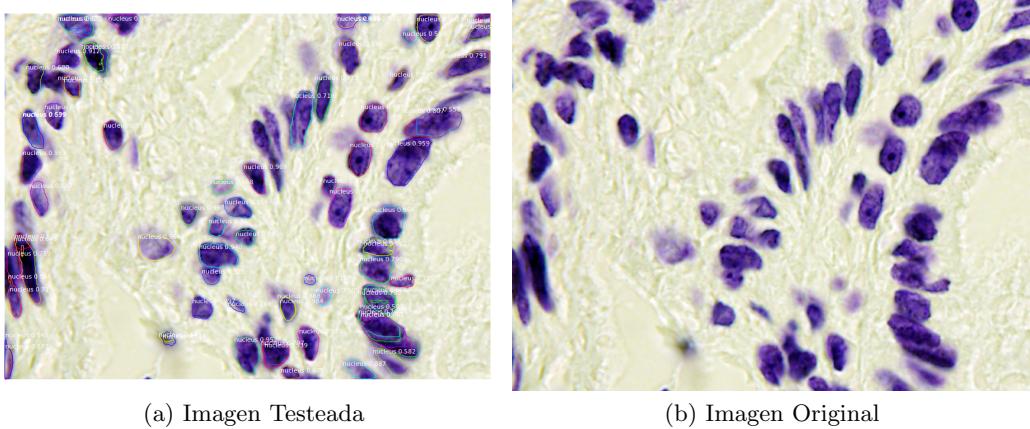


Figura 17: Predicción sobre una imagen aleatoria elegida del conjunto de test.

Puede observarse que la mayoría de núcleos son detectados en las imágenes, un resultado que no difiere del esperado. Además, las formas y dimensiones de las predicciones parecen adecuarse bien a las de los núcleos encontrados, al mismo tiempo que los valores de confianza con los que se han predicho los núcleos se encuentran, en su gran mayoría, por encima del 90 %. Es por ello que podemos concluir que nuestro último modelo actúa como un buen detector de núcleos en distintas imágenes celulares.

7. Propuestas de mejora

- Como primera propuesta de mejora, se propone un estudio más profundo de los hiperparámetros del problema. Una técnica como *cross-validation* para comparar un rango más amplio de distintos parámetros hubiese sido más idóneo. Sin embargo, el tiempo de ejecución de los entrenamientos, los cuales superaban las 15-20 horas; así como los problemas de desbordamiento de memoria encontrados, han impedido una mayor profundidad en esta experimentación.
- De igual manera que la primera propuesta, hubiese sido de interés especial utilizar como backbone *ResNet101*. Esto se debe a que las *Redes Residuales (ResNet)* se inspiran en el hecho de que algunas neuronas se conectan con otras neuronas en capas que no tienen por qué ser contiguas, saltando por ende a capas intermedias. Además, mediante estos *saltos*, se evita que los gradientes se desvanezcan, ya que un gradiente que comienza a base de dar saltos llegará hasta el final de la red sin ser modificado.

Es por ello que el paradigma de **ResNet** tuvo un gran impacto ya que su metodología permitió el entrenamiento de redes mucho más profundas. De hecho una de las principales características de las Redes Residuales es que: “*Con las ResNets puede ganarse fácilmente un mejor valor de la precisión (accuracy) incrementando la profundidad*”. Sin embargo, esta metodología tiene el ya conocido contra: a mayor profundidad, mayor tiempo de ejecución, y mayor necesidad de memoria. Hubiese sido un punto de vista interesante ver cómo mejora esta arquitectura de red en comparación a los resultados obtenidos para imágenes de 512x512 con ResNet50.

- Por último, haciendo referencia a la competición de 2018 en **Kaggle** [1] de la cual se basa nuestro proyecto, numerosas implementaciones que consiguieron un puesto alto en el ranking ganador utilizan **U-net** en vez de Mask R-CNN. Por un lado, Mask-RCNN resuelve el problema directamente, es decir, produce máscaras para cada objeto reconocido. Por el contrario, U-net puede producir solamente una máscara, por lo que se utiliza para predecir la unión de todas las máscaras, seguido de un post-procesamiento para dividir la máscara predicha en una máscara por cada objeto.

La implementación de la propia U-Net resulta más simple que Mask-RCNN, pero requiere de un post-procesado de mayor complejidad. Por otra parte, atendiendo a la literatura [10], se indica que U-Net proporciona un mejor rendimiento cuando los núcleos tienen formas elípticas, pues Mask R-CNN encuentra problemas para establecer los bounding boxes en estas situaciones.

De igual manera, se menciona como U-Net realiza mejores predicciones sobre aquellos núcleos que se encuentran en solitario; por lo que una propuesta interesante sería experimentar con esta Red con el objetivo de intentar mejorar nuestros propios resultados.

8. Bibliografia

Referencias

- [1] Kaggle. *2018 Data Science Bowl. Find Nuclei in divergent images to advance medical discovery.*
<https://www.kaggle.com/c/data-science-bowl-2018/overview>
- [2] Mask R-CNN Implementation. *Mask R-CNN for Object Detection and Segmentation.*
https://github.com/matterport/Mask_RCNN
- [3] Mask R-CNN. *Mask R-CNN.* Kaiming He, Georgia Gkioxari, Piotr Dollár, Ross Girshick
<https://arxiv.org/pdf/1703.06870.pdf>
- [4] CUDA Version. *Build from source. Choosing the correct CUDA Version*
<https://www.tensorflow.org/install/source#gpu>
- [5] CUDA Version. *Download CUDA Toolkit 8.0 - Feb 2017*
<https://developer.nvidia.com/cuda-80-ga2-download-archive>
- [6] cuDNN Version. *Download cuDNN Archive*
<https://developer.nvidia.com/rdp/cudnn-archive>
- [7] PASCAL VOC. *The PASCAL Visual Object Classes (VOC) Challenge. IJCV (2010).* Everingham, M., Van Gool, L., Williams, C.K.I., Winn, J., Zisserman, A.
<http://host.robots.ox.ac.uk/pascal/VOC/pubs/everingham10.pdf>
- [8] Imgaug. *imgaug library documentation*
<https://imgaug.readthedocs.io/en/latest/>
- [9] Binary Cross-Entropy. *Loss Function: Binary Cross-Entropy*
<https://towardsdatascience.com/understanding-binary-cross-entropy-/log-loss-a-visual-explanation-a3ac6025181a>
- [10] Mask R-CNN vs U-Net. *MASK-RCNN and U-NET ensembled for nuclei segmentation.* A.O. Vuola, S.U. Akram, J. Kannala.
<https://arxiv.org/pdf/1901.10170.pdf>