TRABAJOS FIN DE GRADO

Universidad de Jaén

Escuela Politécnica Superior de Jaén

Técnica de Segmentación de Árboles en Fotografías Aéreas: Un Estudio para Aprendizaje Profundo

Autor: Alba Cano Lara

Grado: Ingeniería Informática

Director: Prof. D. Juan Roberto Jiménez Pérez y Prof. D. Juan Manuel Jurado Rodríguez

Departamento del director: Departamento de Informática

Fecha: 01/07/2024

Agradecimientos

Terminar este proyecto supone el fin de una etapa y el inicio de otra. Ha sido difícil llegar hasta aquí, superando todos los retos que han supuesto completar este Grado en Ingeniería Informática. Por ello, me gustaría expresar mi más sincero agradecimiento a aquellas personas que han hecho posible la realización de este trabajo.

En primer lugar, agradezco a mis tutores, D. Juan Roberto Jiménez Pérez y D. Juan Manuel Jurado Rodríguez por la orientación, sus consejos y ayuda en la realización de este trabajo.

Agradezco a mis amigos y compañeros de vida, en especial a Adrián, Sara, Alejandro y Lucía, que han compartido conmigo este viaje académico. Juntos hemos construido recuerdos y experiencias que me llevo a la siguiente aventura. A ellos les debo parte de mis logros.

Finalmente, me gustaría agradecer a mi familia el enorme esfuerzo y la comprensión que han tenido a lo largo de esta etapa. Me han permitido tomarme el tiempo necesario para conseguir mis metas y me han apoyado en mis decisiones, confiando en mí hasta el último momento.

Muchísimas gracias.

Índice

Índice de Figuras

Índice de Tablas

Índice de Fragmentos de Código

Índice de Algoritmos

Índice de Funciones

Glosario de Términos

Término	Siglas
Light Detection And Ranging	LiDAR
Aprendizaje Automático	AA
Aprendizaje Profundo	AP
Application Programming Interface	API
Artificial Intelligence	AI
Central Processing Unit	CPU
Graphic Processing Unit	GPU
Integrated Development Environment	IDE
Rectified Linear Unit	ReLU
Convolutional Neural Network	CNN
Region Based Convolutional Neural Network	R-CNN
Red De Propuestas De Regiones	RPN
Region Of Interest	RoI
You Only Look Once	YOLO
Feature Pyramid Network	FPN
Unmanned Aerial Vehicle	UAV
Field Of View	FOV
Computer-Aided Design	CAD
Generative Adversarial Network	GAN
Common Objects In Context	COCO

1. Prefacio

1.1 Motivación

En el estudio de entornos naturales, la identificación de árboles individuales es crucial para una caracterización detallada y la creación de modelos 3D basados en esa información, entre otras aplicaciones.

Junto con la incorporación de drones, se hace posible capturar información de parajes naturales utilizando cámaras RGB de alta resolución, hiperespectrales y sensores LiDAR (*siglas que se pueden ver en Glosario de Términos*). De esta forma, se puede recopilar una cantidad de datos e imágenes para el análisis del entorno.

El problema que se plantea es cómo llevar a cabo dicho análisis de una manera eficiente, automática y que presente conclusiones precisas para la posterior generación de los modelos. Además, se pone el foco en la resolución mediante el desarrollo de técnicas basadas en aprendizaje profundo.

Tras el auge en la última década de la inteligencia artificial y sus diferentes ramas, surgen nuevos enfoques que pretenden dar respuesta a tareas complejas aprovechando los recursos computacionales actuales y el acceso a bancos de datos. Debido a esto, se han ido abandonando técnicas tradicionales en la visión por computador para apostar por las prometedoras del aprendizaje profundo.

En definitiva, el motivo de la realización de este proyecto es el de aplicar técnicas de aprendizaje profundo orientadas a la detección, segmentación e instanciación de árboles en entornos naturales.

1.2 Objetivos

El propósito del proyecto es el de analizar el estado del arte de los modelos de segmentación e instanciación de imágenes y posteriormente estudiar los resultados que ofrecen para solucionar el problema de detección de árboles.

- Revisión del estado del arte de los modelos basados en redes neuronales para la detección de objetos.
- Desarrollo, selección de técnicas e implementación del preprocesamiento del conjunto de imágenes.

- Implementación de distintos modelos de instanciación de imágenes con el objetivo de realizar comparativas futuras.
- Recopilar resultados tras diversas ejecuciones y parámetros.
- Comportamiento de los modelos entrenados ante la detección de otro tipo de zonas arbóreas desconocidas.
- Comparación de los resultados obtenidos llegando a una conclusión acerca de qué modelo es el que mejor soluciona nuestro problema.

1.3 Metodología

Conforme a lo indicado anteriormente y dada la tipología experimental de este TFG, la metodología a seguir para su correspondiente desarrollo difiere de la común entre los TFG de ingeniería. Por ende, la metodología de este proyecto será:

- Se desarrollará un estudio teórico y una revisión del estado del arte de los modelos propuestos a utilizar.
- Se establecerán las tecnologías, herramientas y plataformas según las necesidades de cada modelo.
- Se implementará el código en las tecnologías software seleccionadas.
- Se realizará un análisis y una discusión de los resultados obtenidos.
- Se incluirá en la memoria del trabajo fin de grado todo el proceso seguido junto con los resultados finales, además de las conclusiones del estudio y las referencias bibliográficas utilizadas.

1.4 Resultados

- Código fuente de la solución contenedora de todas las implementaciones de los modelos.
- Manual de usuario de los modelos.
- Memoria.

1.5 Herramientas Utilizadas

Este apartado pretende dar una visión del marco de trabajo utilizado para el desarrollo e implementación de los modelos.

1.5.1 TensorFlow y Keras

Para la implementación del modelo de redes neuronales convolucionales, el framework escogido es **TensorFlow** junto con **Keras** puesto que ambos son dos componentes esenciales en el campo del *Aprendizaje Profundo* y *Automático*.

Tensorflow es una plataforma Open Source desarrollada por Google en 2015. Ofrece un amplio ecosistema de herramientas, librerías y recursos para construir aplicaciones basadas en *AA*. Se destaca por su capacidad para crear y entrenar modelos de *AP* y *AA* de manera eficiente.

Keras es la *API* de alto nivel integrada en TensorFlow que facilita la creación y el entrenamiento de modelos de aprendizaje profundo. Está diseñada para ser accesible y extensible, permitiendo a los desarrolladores enfocarse en diseñar modelos sin tener que lidiar con los aspectos técnicos más complejos. Keras funciona como una capa sobre TensorFlow, permitiendo definir modelos a través de su interfaz simplificada y luego ejecutarlos utilizando TensorFlow como motor subyacente.

1.5.2 Pytorch

Atendiendo a los requisitos del resto de modelos, otro de los *frameworks* utilizado será Pytorch. Posibilitará la implementación de éstos además de la adecuación del conjunto de datos para el posterior entrenamiento.

PyTorch es una biblioteca de aprendizaje automático basada en la biblioteca Torch, utilizada para aplicaciones como visión por computadora y procesamiento del lenguaje natural. Originalmente desarrollada por *Meta AI*, aunque ahora forma parte del Linux Foundation. Se reconoce como una de las dos bibliotecas de aprendizaje automático más populares junto a TensorFlow, ofreciendo software y código libre.

PyTorch se caracteriza por su excelente soporte para GPUs y su uso de diferenciación automática en modo inverso, lo que permite modificar gráficos de cálculo al instante. Esto lo convierte en una opción popular para experimentación rápida y prototipado. La biblioteca es conocida por su flexibilidad y facilidad de uso, gracias a su compatibilidad con el lenguaje de programación Python.

1.5.3 Visual Studio

Uno de los entornos de trabajo seleccionados para llevar a cabo el trabajo es Visual Studio.

Visual Studio es un entorno de desarrollo integrado (IDE) creado por Microsoft que ofrece un amplio abanico de herramientas para el desarrollo de software. Este IDE es compatible con múltiples lenguajes de programación, incluyendo Python.

Con él, se podrán implementar los modelos para su ejecución de manera local haciendo uso de los recursos del usuario.

1.5.4 Google Colab

Google Colaboratory, también conocido como Google Colab, es un servicio alojado de Jupyter Notebook que no requiere de ninguna configuración para su uso y ofrece una capa de acceso gratuito a recursos informáticos, incluidos GPUs y TPUs.

Una de las ventajas clave de Google Colab es la que proporciona acceso gratuito a infraestructura de cómputo intensiva, lo cual es crucial para realizar las pruebas y ejecuciones de código que necesitan grandes cantidades de GPU. Sin embargo, la capa gratuita sólo da acceso a un par de horas de conexión a entornos con aceleración gráfica y probablemente sea un impedimento en la fase de experimentación.

Existen planes de pago como Google Colab Pro en el que por 12€ aproximadamente al mes, te permite utilizar 100 unidades informáticas, GPUs más rápidas y potentes, con más memoria RAM en los equipos y una ampliación de las horas de conexión hasta 24 horas de conexión ininterrumpida.

1.5.5 Anaconda Navigator

Para gestionar paquetes y entornos de trabajo especializados será de gran utilidad Anaconda.

Anaconda es una distribución libre y de código abierto de los lenguajes de programación Python y R, diseñada específicamente para facilitar el trabajo en ciencia de datos, aprendizaje automático y análisis predictivo.

Uno de los aspectos más destacados de Anaconda es su sistema de gestión de paquetes llamado conda, que permite instalar, ejecutar y actualizar fácilmente software de ciencia de datos y aprendizaje automático como TensorFlow, Pytorch o Numpy entre otros. Conda es especialmente útil porque maneja las dependencias de manera eficiente, asegurando que las diferentes versiones de los paquetes sean compatibles entre sí.

1.5.6 Clúster de Computación Gráfica ADA

Para lograr mejores resultados en los distintos entrenamientos se hará uso de los recursos que provee el **clúster de ADA** (*Manual de usuario ADA*, s. f.) perteneciente al **CEATIC** (*Centro de Estudios Avanzados en Tecnologías de la Información y de la Comunicación*). El **CEATIC** es un espacio dedicado a la investigación interdisciplinar, en el que se unen investigadores y docentes de distintas disciplinas del ámbito computacional con el propósito de dar apoyo a la excelencia investigadora e impulsar la transferencia de conocimiento. Otro de sus objetivos es el ofrecer sus recursos y medios instrumentales punteros para tareas de investigación y desarrollo tecnológico mediante contratos con empresas e instituciones. Aunque, también permiten el uso de éstos a la comunidad de estudiantes de la universidad. Por ello, podremos acceder a dichos recursos para la realización de este trabajo.

Este clúster es un sistema de computación de altas prestaciones que da acceso a un procesador Intel Xeon Silver 4280 y a distintas tarjetas gráficas potentes como NVIDIA Tesla V100, NVIDIA Tesla A100 y NVIDIA GeForce RTX 2080Ti.

Al ser un sistema de archivos compartido, cada usuario dispone de una cuota de disco de unas 300 gigas, aunque se puede solicitar un aumento de estas durante un determinado tiempo. Sin embargo, para este estudio el espacio no es relevante, sino la capacidad de cómputo que ofrece.

2. Estado del Arte

2.1 Introducción

Para poner en contexto, se deben de definir los antecedentes del problema del proyecto, que es la identificación de árboles. De esta forma, se podrán entender los pasos dados para llegar a una resolución del problema y explicar qué técnica es la óptima.

2.2 Aprendizaje Profundo

El aprendizaje profundo es una rama de la inteligencia artificial que se basa en gran medida en las redes neuronales. Estas se inspiran en el comportamiento del cerebro humano, el cual se compone de muchas células o neuronas. Cada una realiza una operación sencilla e interactúa con el resto para tomar una decisión. Tras esto, el sistema va aprendiendo a través de las numerosas capas de la red neuronal consiguiendo una mejor tolerancia a fallos y una mayor adaptabilidad a los nuevos datos. En *Figura 1* se puede observar el esquema de una red en la cual las neuronas interactúan entre sí pasando la información.

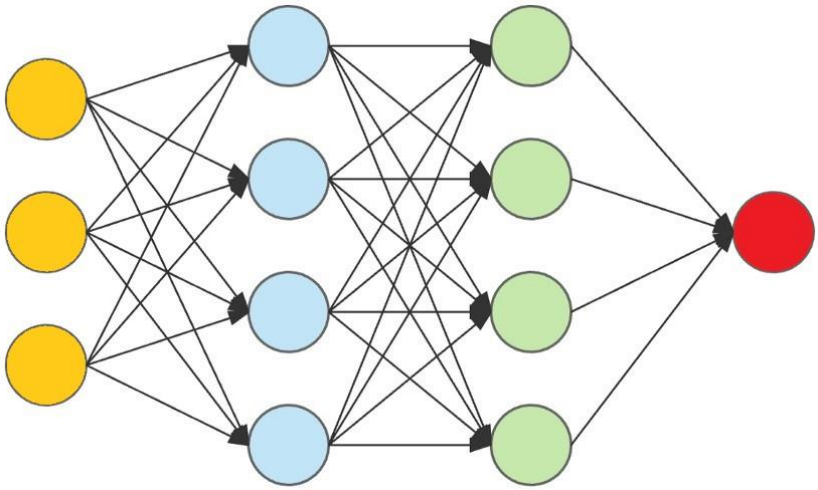


Figura 1. Estructura básica de una red neuronal (SVRAI, s. f.)

Algunos de los factores que contribuyeron al impulso del aprendizaje profundo fueron la aparición de grandes conjuntos de datos anotados de gran calidad disponibles públicamente, y el auge de la computación paralela en GPU que permitió la transición del entrenamiento basado en CPU al basado en GPU. Esto permitió una aceleración significativa en el entrenamiento de los modelos.

2.3 Redes Neuronales Aplicadas a la Visión por Computador

El aprendizaje profundo ha impulsado grandes avances en diversos problemas de visión por ordenador como la detección de objetos, el seguimiento de movimiento y la segmentación semántica gracias al uso de varios tipos de redes neuronales.

2.3.1 Redes Convolucionales (CNN)

Los modelos que están basados en las redes convolucionales resultan efectivos en tareas de la visión por computador y detección de patrones. Su estructura básica consiste en bloques o capas convolucionales, la cual se puede observar en la *Figura 2*, operadores de agrupamiento (*downsampling*), funciones de activación como la ReLU o la Sigmoide, normalización, combinación linear, y capas totalmente conectadas. Esta red se puede comprender como una secuencia de funciones como se describe en *Función 1*, que toman como entrada un vector *x* junto con unos parámetros *W*, y devuelve otro vector *x'*.

$$f(x) = f_L(\cdots f_2(f_1(x_1, W_1); W_2)\cdots), W_L)$$

Función 1. Función compuesta que representa una red CNN. (Ponti et al., 2017)

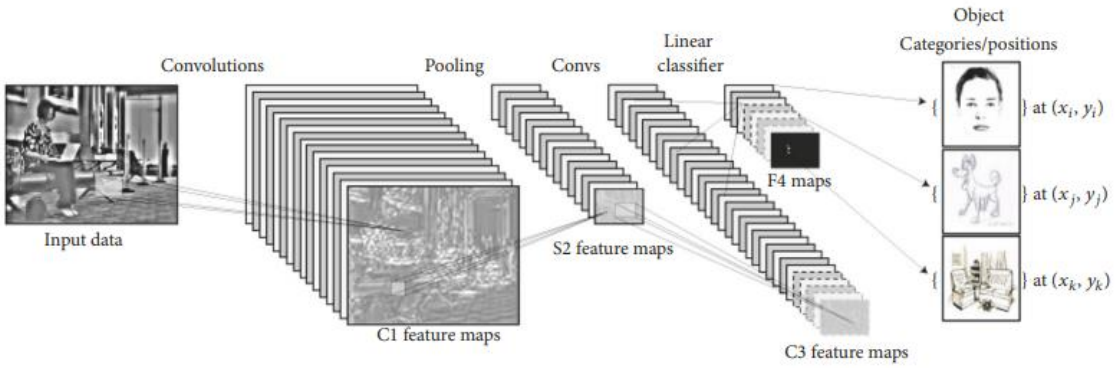


Figura 2. Ejemplo de la arquitectura de la CNN para la detección de objetos. (Voulodimos et al., 2018)

Existen diferentes tipos de bloques o capas convolucionales que determinarán el comportamiento y la eficiencia de la red según su posición y su combinación con el resto de ellas. A continuación, se explicarán los distintos bloques que suelen componer una red neuronal convolucional como Capa

Convolucional, Función de Activación, Capa Pooling, Capa de Dropout, Capa Softmax, Capa de Normalización y, por último, Capa Totalmente Conectada.

Capa Convolucional

Es una capa compuesta por un conjunto de filtros que se aplican directamente al vector de entrada. Estos filtros no son más que una matriz de pesos **n**x**n**. Cada peso es un parámetro del modelo que se irá modificando a medida que avance el aprendizaje de tal forma que sean capaces de detectar ciertas características.

La idea principal de esta capa es aplicar el filtro para comprobar si está la característica que se busca. Para ello, se lleva a cabo una operación de convolución calculando el producto del filtro con el área de entrada, a la cual se le superpone dicho filtro, como se puede observar en *Figura 3*.

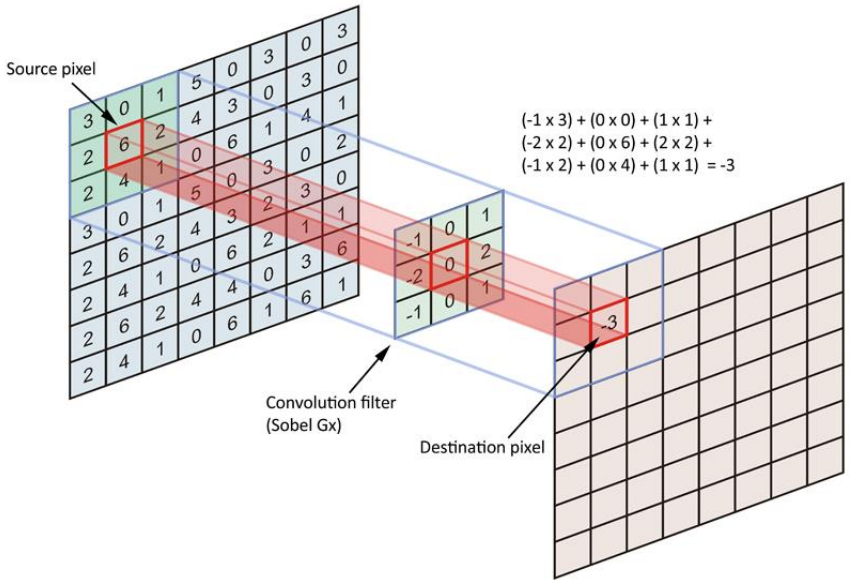


Figura 3. Ejemplo de cálculo convolucional con un filtro de Sobel (Navarro, s. f.)

Los filtros más comunes tienen unas dimensiones de 5x5x*d* o 3x3x*d*, donde *d* se refiere a la profundidad del tensor.

Función de Activación

Las funciones de activación suelen ser funciones no lineales como **Sigmoide**, **Tanh**, y **ReLU** (*Rectified Linear Unit*). Dependiendo de la naturaleza de los datos y las tareas de clasificación, unas funciones funcionarán mejor que

otras. Para ello se definirán brevemente y sus usos en el aprendizaje de las neuronas (Ponce, 2021).

La función **Sigmoide** mapea cualquier valor real a un rango comprendido entre [0, 1]. Se suele utilizar en la capa de salida de modelos de clasificación binaria, aunque es costosa computacionalmente al tener operaciones exponenciales.

A diferencia de la sigmoide, la función **tangente hiperbólica** o **tanh** mapea los valores en un rango entre [-1, 1]. Su uso se centra en las capas ocultas de redes neuronales puesto que sus valores centrados en cero facilitan el aprendizaje.

En último lugar, la función **ReLU** mapea cualquier valor real en un rango de [0, ∞]. Esta función simula la activación o no de las neuronas del cerebro, y gracias a esto se obtienen resultados favorables en tareas de reconocimiento de imágenes.

En términos matemáticos, que se pueden apreciar en *Figura 4*, y de velocidad de entrenamiento, conseguir una salida aplicando **f(x) = tanh(x)** es mucho más lento que **f(x) = max(0, x)** como es el caso de ReLU. Por ello, muchas de las arquitecturas conocidas de CNNs aplican la función de activación ReLU.

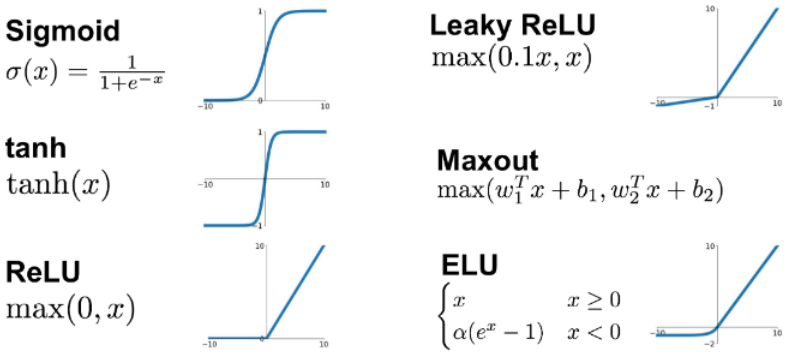


Figura 4. Gráficas de las funciones de activación más comunes. (Navarro, 2022)

Agrupación (Pooling)

Usualmente, después de algunas capas convolucionales se aplica la técnica de agrupación. Esta se encarga de reducir la muestra de la imagen, reduciendo la dimensión del vector.

El operador *maxpooling* es el más utilizado. Este tipo de operación tiene dos objetivos: reducir el tamaño de los datos y reducir el tamaño de la imagen con lo que es posible obtener bancos de filtros multiresolución, procesando la entrada en diferentes espacios de escala. Un ejemplo básico del funcionamiento de esta capa se puede ver en *Figura 5*.

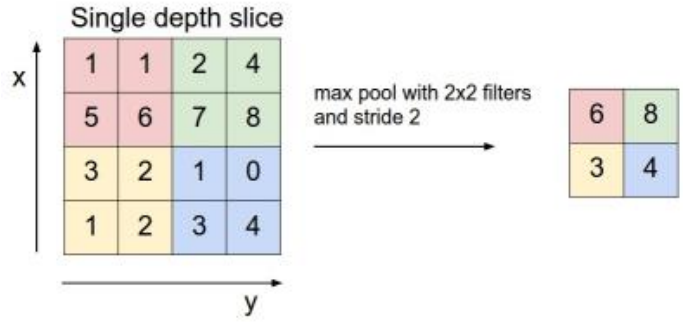


Figura 5. Ejemplo de MaxPooling con un filtro de 2x2. («Redes Neuronales Convolucionales en Profundidad», 2018)

Aun así, con la llegada de los modelos generativos, este tipo de capas puede complicar el entrenamiento y por tanto es probable que haya una tendencia en reducir su uso en futuras arquitecturas.

Capa de Dropout

En esta capa, se anulan aleatoriamente una serie de neuronas al entrenar cada ejemplo. En *Figura 6* se muestra un ejemplo del resultado tras eliminar ciertas conexiones entre neuronas aplicando el concepto de *dropout*. Esto hace que las neuronas aprendan de forma independiente y se reduzca el tiempo de entrenamiento.

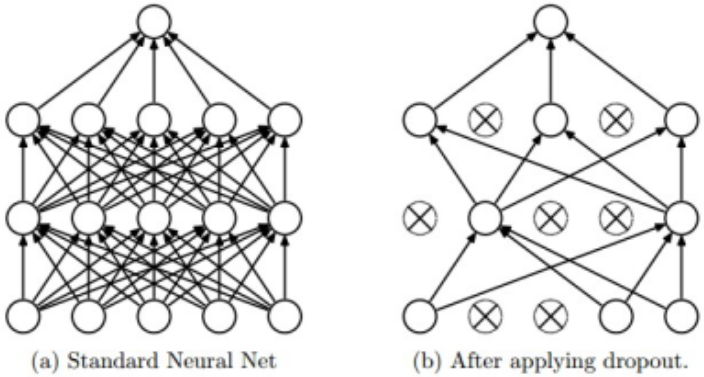


Figura 6. Comparación entre una red estándar, y otra red tras aplicar dropout. (Sharma, 2022)

Capa Softmax

La función *softmax* o función exponencial normalizada en el ámbito del aprendizaje por refuerzo se utiliza para convertir valores en probabilidades para llevar a cabo determinadas acciones. Se basa en el concepto de la **recompensa esperada**, una estimación de la ganancia o beneficio, que se obtendría al realizar una cierta acción en un estado específico. La función *softmax* transforma estas recompensas esperadas en probabilidades, de tal manera que acciones con mayor recompensa esperada tienen una mayor probabilidad de ser elegidas. En *Función 2* se calcula la probabilidad de escoger una acción **z** con una recompensa esperada $\mathbf{z}_j$.

$$\sigma(z)_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}}$$

Función 2. Función Softmax para el cálculo de la probabilidad de una acción.

Suele ser una de las últimas capas de las CNN, puesto que facilita la interpretación de las salidas de la red y las normaliza. Esto resulta en que la clase con la mayor probabilidad es la predicción de la red para una imagen en cuestión. También es importante en la parte del entrenamiento, debido a que la salida de la función *softmax* se puede usar junto con la función de *pérdida de entropía cruzada* para comparar las predicciones de la red con las anotaciones reales de la imagen.

Capa de Normalización

Tras las capas de convolución, también se suele aplicar el proceso de normalización de los datos. La técnica más conocida es *Batch Normalization*, o la normalización de lotes.

A todos los datos de un lote $\boldsymbol{x_i}$, se les resta la media y se dividen por su desviación típica, obteniendo $\boldsymbol{x'_i}$. La salida de esta capa sería $\mathbf{f(}\boldsymbol{x_i}\mathbf{) = \lambda}\boldsymbol{x'_i}\mathbf{ + \beta}$, donde los parámetros **λ** y **β** son aprendidos por la red neuronal.

Capa Totalmente Conectada

Por último, esta capa examina el vector de entrada completo, dando como resultado un único valor escalar. Un ejemplo de las conexiones de esta capa se

puede ver en *Figura 7*. Para ello, toma como entrada la versión redimensionada de los datos procedentes de la última capa. Normalmente, las conversiones de dimensiones que se producen en esta capa suelen pasar de ser bidimensionales procedentes de la capa anterior a tener tan sólo una dimensión. Por ejemplo, con un tensor o vector de la capa anterior a la totalmente conectada tiene como dimensión **4x4x40**, al redimensionarlo a una dimensión, quedaría de la siguiente forma: **1 x (4x4x40) = 1 x 640**.

Por tanto, cada neurona de esta capa, la totalmente conectada estará asociada a 640 pesos, produciendo una combinación lineal del vector.

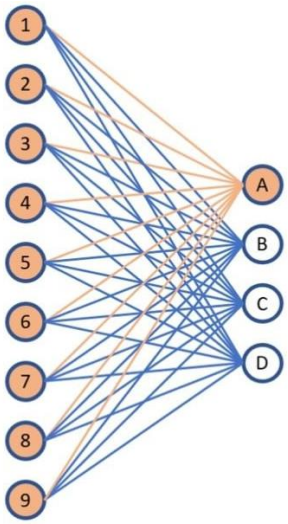


Figura 7. Ejemplo de una capa totalmente conectada. (Mudadla, 2023)

2.3.2 Arquitecturas de Redes CNN

En la década anterior, a partir de 2010, surgieron numerosas arquitecturas basadas en las redes neuronales convolucionales dado su éxito en el ámbito de la visión por ordenador y tomando como referencia la que propuso *LeCun* veinte años antes.

LeNet

Esta red fue diseñada por Yann LeCun (Lecun et al., 1998) y su equipo en la década de los noventa. El propósito de esta red era reconocer caracteres escritos a mano mediante el análisis de imágenes. En la *Figura 8*, se muestra un esquema de la estructura de la red que realiza ese reconocimiento de un carácter escrito a mano.

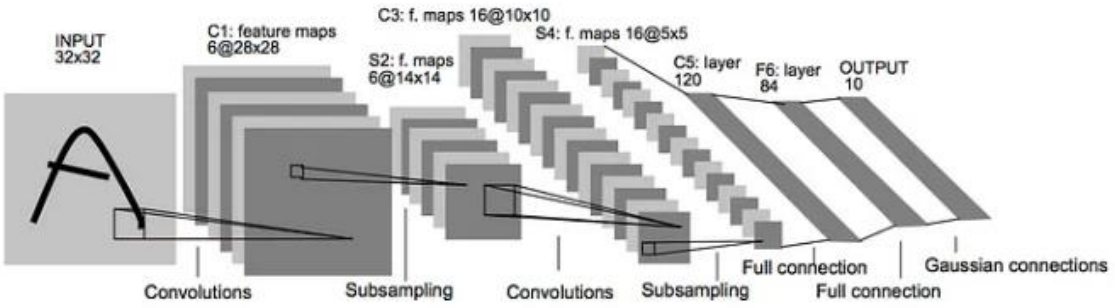


Figura 8. Arquitectura de la red LeNet-5. (Lecun et al., 1998)

Esta propuesta estaba compuesta de siete capas, dos convolucionales, dos de agrupamiento, y tres últimas totalmente conectadas. No se consiguieron unos buenos resultados al depender mucho de las condiciones y formatos de los datos, pero sentó la base de las futuras estructuras.

AlexNet

Fue expuesta por Alex Krizhevsky (Krizhevsky et al., 2012). La *Figura 9* expone su arquitectura. Consiste en cinco capas convolucionales, cada una seguida por funciones de activación ReLU, y finalmente tres capas completamente conectadas. Por último, se añaden dos capas más, una de *dropout* y el resultado se pasa a una capa *softmax* que produce una distribución de entre mil clases posibles para la clasificación.

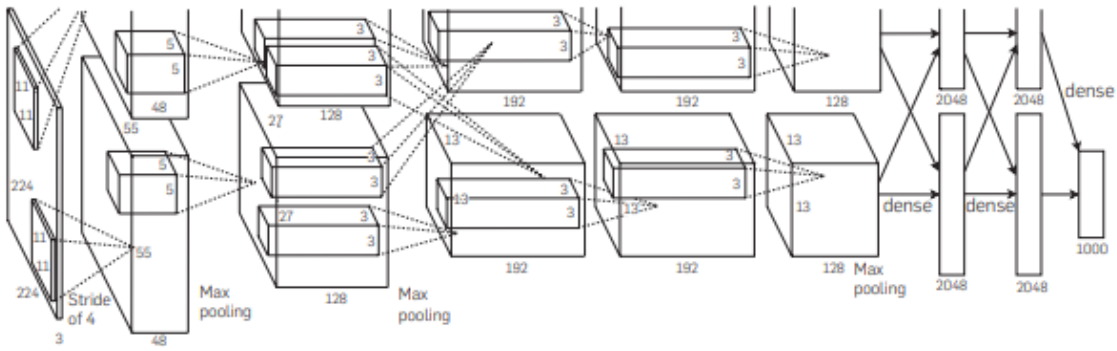


Figura 9. Arquitectura de AlexNet. (Krizhevsky et al., 2012)

Esta arquitectura fue presentada en 2012 y ganó fama tras competir en el *ImageNet Large Scale Visual Recognition Challenge* y lograr unos resultados significativos en comparación con otros participantes.

AlexNet fue el primer modelo convolucional que aprovechó el rendimiento de las unidades de procesamiento gráfico (GPUs) durante el entrenamiento, lo

que ayudó a entrenar modelos más profundos y complejos con una cantidad ingente de datos.

VGG-Net

Esta red fue publicada por Simonyan y Zisserman en 2015 (Simonyan & Zisserman, 2015). Su idea era aportar más profundidad a la red añadiendo capas convolucionales con filtros más pequeños de 3x3, dejando el resto de los parámetros fijos. Ellos argumentaban, que varios filtros pequeños consecutivos conseguían resultados más efectivos y tenían la ventaja de rectificar más entre capas. Esta consecución se puede apreciar en la *Figura 10*. De esta forma la función de decisión era más discriminativa. Además, se diferenciaban entre otros modelos coetáneos en el número de parámetros de unos 500 millones, mientras que AlexNet tenía 200 millones. Lo cual perjudicaba bastante en el tiempo de entrenamiento.

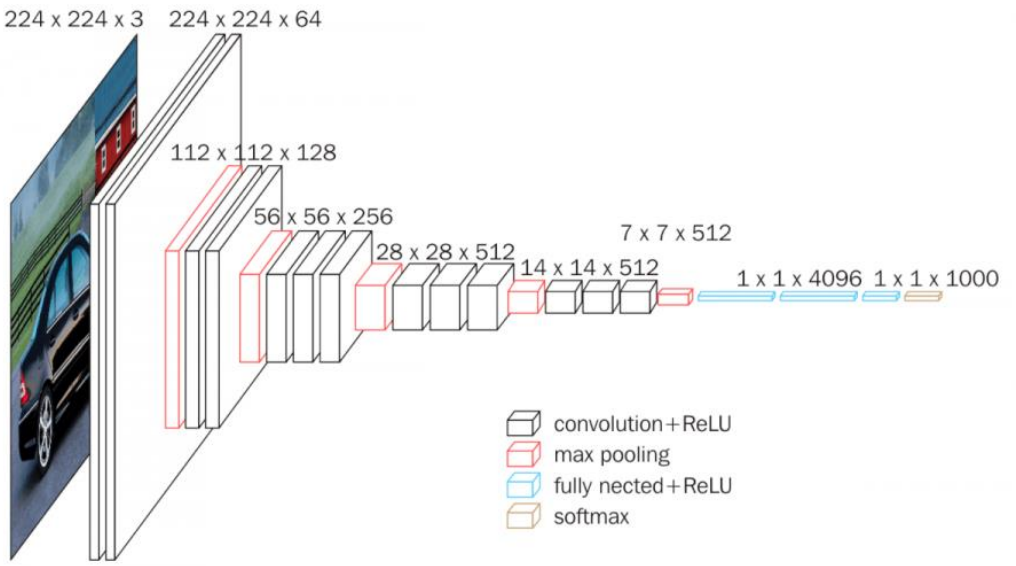


Figura 10. Arquitectura de VGG-16. (Papers with Code - VGG Explained, s. f.)

ResNet

Con la publicación de esta arquitectura, surgió el dilema de si construir estructuras más profundas y con más capas significaría mejores modelos. Así que, en 2016 (He et al., 2016) establecieron que aprender una función residual relativa a la entrada a las capas era más eficiente que aprender los parámetros de estas sin tenerlas en cuenta. Vemos un ejemplo de la aplicación del aprendizaje residual en la *Figura 11*.

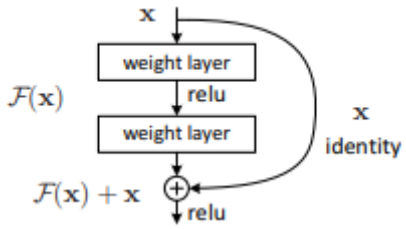


Figura 11. Aprendizaje residual. (He et al., 2016)

Propusieron una red residual llamada **ResNet** con 152 capas, ocho veces más profunda que **VGG**. Esta red utilizaba múltiples capas de parámetros, como se presenta en la *Figura 12*, para aprender la representación de los residuos entre la entrada y la salida. Al aumentar las conexiones directas, se refuerza la propagación de características, la reutilización de estas, y la reducción sustancial del número de parámetros. A pesar de esto, su desventaja principal es el exceso de ajuste.

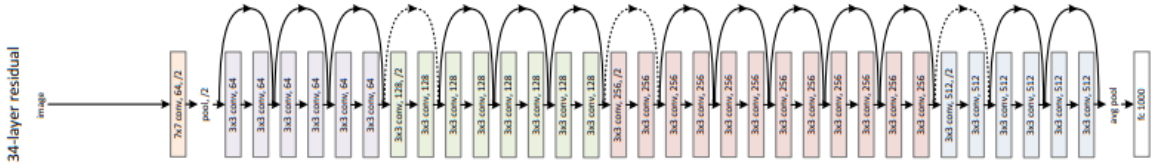


Figura 12. Arquitectura ResNet. (He et al., 2016)

2.3.3 Redes Convolucionales basadas en Regiones (R-CNN)

Recientemente, las técnicas de aprendizaje profundo han avanzado, especialmente aquellas redes convolucionales basadas en regiones que tomaron el nombre de **R-CNN**. Con su evolución han conseguido éxitos notables en las tareas de detección de objetos, clasificación y segmentación de imágenes. Esto se debe a que las CNN convencionales solo eran capaces de identificar la clase del objeto, pero no su ubicación en la imagen, y menos aun cuando aparecen varios objetos.

La R-CNN se encarga de identificar las regiones de interés denominadas **RoI** (*Region of Interest*), mediante una búsqueda selectiva utilizando como base una CNN. La *Figura 13* ejemplifica el funcionamiento interno de este tipo de redes. El algoritmo se divide en cuatro etapas.

1. **Red de Propuestas de Regiones (RPN):** En esta fase, el modelo genera un conjunto de propuestas de regiones que probablemente contengan objetos.
2. **Agrupación de Regiones de Interés (RoI):** Con el conjunto de propuestas de regiones, cada una se recortará de la imagen y se redimensionará a un tamaño determinado.
3. **Extracción de Características:** Las regiones preprocesadas pasarán a la red convolucional previamente entrenada para extraer características de cada región.
4. **Clasificación de objetos y Regresión de Cajas Delimitadoras:** Las características extraídas de cada región se utilizan para la clasificación de los objetos y delimitarlos dentro de una caja delimitadora. La clasificación consistirá en determinar la clase del objeto dentro de la región, y la delimitación define las coordenadas de la caja alrededor del objeto.

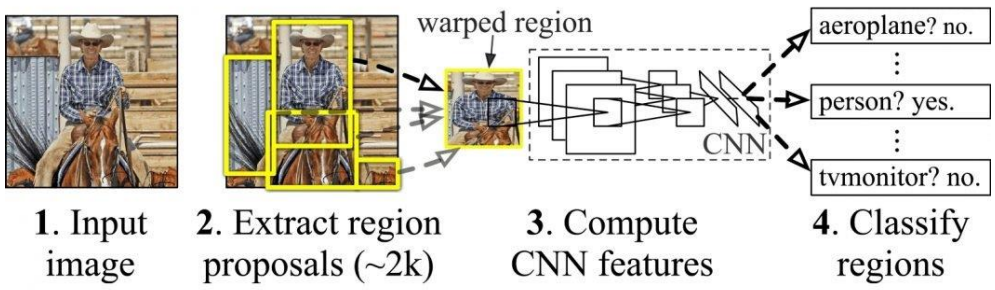


Figura 13. Ejemplo funcionamiento interno de una R-CNN. (Wang et al., 2021)

Sin embargo, este tipo de entrenamiento es muy costoso y lento, por tanto, surgieron modificaciones que pretendían acelerar el entrenamiento y mejorar la precisión como **Fast R-CNN**, **Faster R-CNN**, **YOLO** (*You Only Look Once*) o **Mask R-CNN**.

2.4 Modelos de Segmentación de Instancias

Aunque para el problema del trabajo, los modelos adecuados son aquellos que segmentan la imagen en instancias de objetos, en nuestro caso, en árboles. El estudio teórico realizado y según (Firoze et al., 2023; Safonova et al., 2021) fundamentan la teoría de que técnicas orientadas a la segmentación basadas en **Mask R-CNN** son las idóneas para la resolución de ese problema.

2.4.1 Mask R-CNN

El método **Mask R-CNN** introducido por (He et al., 2017) es una extensión de **Faster R-CNN** (Ren et al., 2017). El objetivo principal de este modelo es predecir una máscara binaria por cada región de interés, además de la etiqueta de clasificación y la caja delimitadora (*bounding box*) que da como resultado **Faster R-CNN** por cada objeto candidato detectado. La *Figura 14* esquematiza el proceso de segmentación con Mask R-CNN.

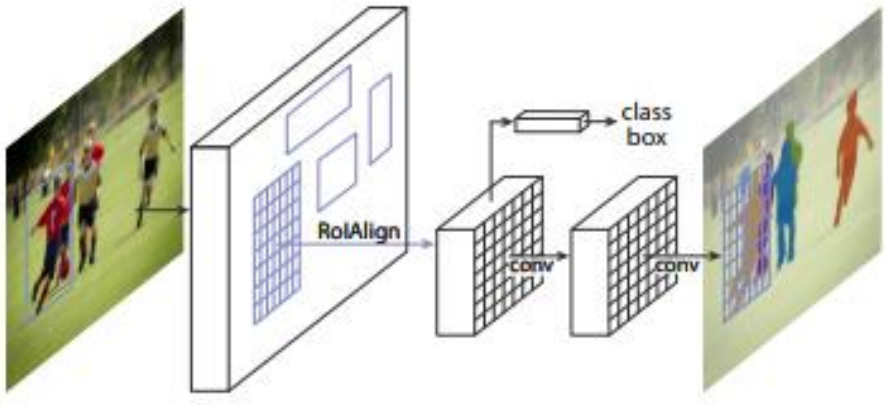


Figura 14. Esquema de segmentación con Mask R-CNN. (He et al., 2017)

La arquitectura de esta red se fundamenta en la arquitectura previamente mencionada **ResNet** (He et al., 2016), y se puede dividir en dos partes:

1) **Backbone** – arquitectura convolucional que se utiliza como base para la extracción de las características de la imagen completa. Fusionan la parte troncal de la red **ResNet** junto con **FPN** (*Feature Pyramid Network*) (Lin et al., 2017) para lograr excelentes resultados tanto en precisión como en velocidad.
2) **Head** – arquitectura de red convolucional totalmente conectada encargada del reconocimiento de las cajas delimitadoras mediante clasificación y regresión. A esta parte, añaden la rama de predicción de la máscara con una estructura sencilla.

Con esto, consiguieron resultados bastante buenos como por ejemplo la *Figura 15*, y que con un entrenamiento adecuado podríamos obtener una instanciación de zonas arbóreas prometedoras.

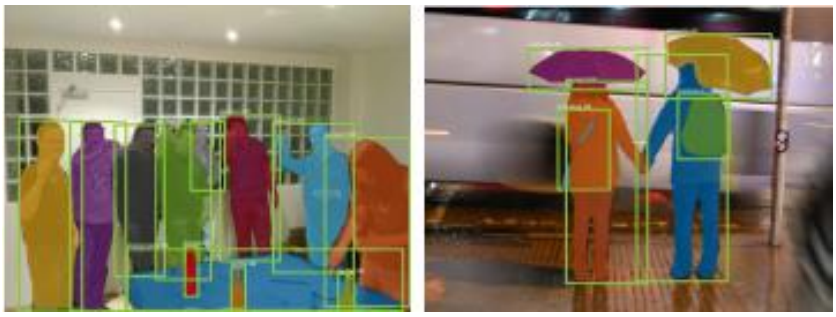


Figura 15. Ejemplo de resultados de Mask R-CNN sobre imágenes del conjunto COCO. (He et al., 2017)

2.4.2 YOLOv8

YOLO, "*You Only Look Once*" (Redmon et al., 2016), es un algoritmo de detección de objetos que se caracteriza por su velocidad y eficiencia en la identificación de objetos en imágenes y videos. Este algoritmo revolucionó el campo de la detección de objetos al permitir la detección de múltiples objetos en una única pasada por la red neuronal, a diferencia de los métodos previos que requerían múltiples pasos y análisis secuenciales.

La arquitectura de **YOLO** se basa en redes neuronales convolucionales y divide la imagen de entrada en una cuadrícula de celdas como se muestra en *Figura 16*. Para cada celda, la red produce una lista de posibles cuadros delimitadores y clases de objetos. Estos cuadros delimitadores contienen las coordenadas de las esquinas del cuadro y las probabilidades asociadas a cada clase de objeto. La detección de objetos se realiza mediante la regresión de las coordenadas de los cuadros delimitadores y la clasificación de los objetos basada en las probabilidades predichas.

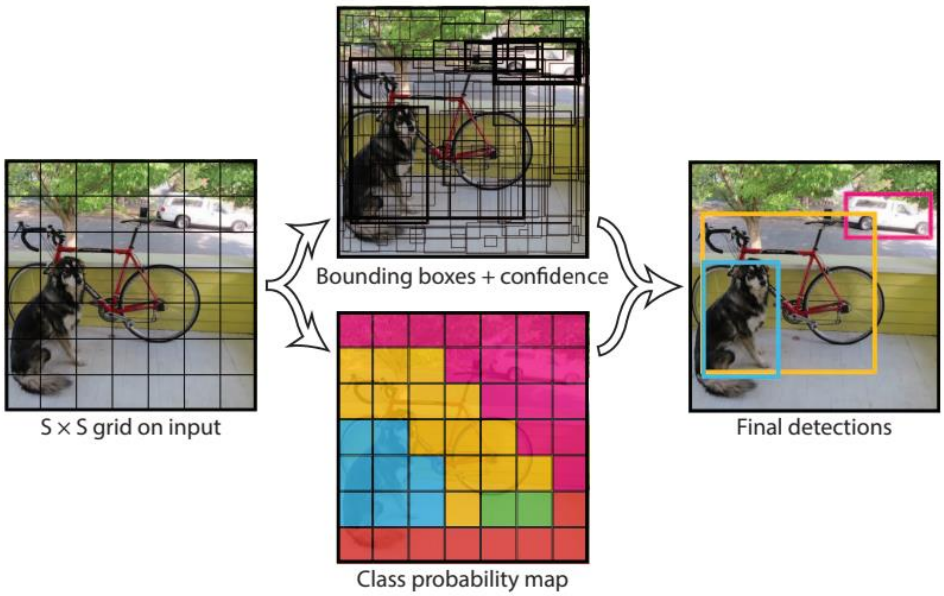


Figura 16. Detección del modelo YOLO. (Redmon et al., 2016)

La versión seleccionada ha sido **YOLOv8**, que fue lanzada en enero de 2023 por la empresa *Ultralytics* (Jocher et al., 2022/2023). El motivo fue su soporte para tareas de detección de objetos y segmentación. **YOLOv8** ofrece un modelo de segmentación semántica denominado **YOLOv8-Seg** (Terven & Cordova-Esparza, 2023), la extensión escogida para realizar los experimentos de instanciación de árboles.

2.4.3 Detectron2

Otro modelo a destacar es **Detectron2** desarrollado por *Facebook AI Research* (*facebookresearch/detectron2*, 2019/2024). Es un proyecto de código abierto que ofrece una solución avanzada para tareas de detección de objetos, la segmentación de instancias y semántica. Es un modelo personalizable y modular que permite a los usuarios modificar sus componentes como el núcleo, el cuello y la cabeza. La *Figura 17* presenta el esquema de estos módulos. Además, ofrece diferentes versiones de modelos pre-entrenados que permiten realizar entrenamientos y ajustes a medida según el conjunto de datos o tareas de detección deseadas.

Su estructura no difiere mucho de los modelos basados en **Mask R-CNN**, en la que se destacan:

1) **Parte troncal (Backbone)** que extrae mapas de las características a diferentes escalas.
2) **RPN (Region Proposal Network)** regiones del objeto detectadas a partir de características multiescala.
3) **ROI Heads** procesan los mapas de características a partir de las regiones seleccionadas anteriormente. Se refinan las posiciones de las cajas delimitadoras mediante capas de neuronas totalmente conectadas.

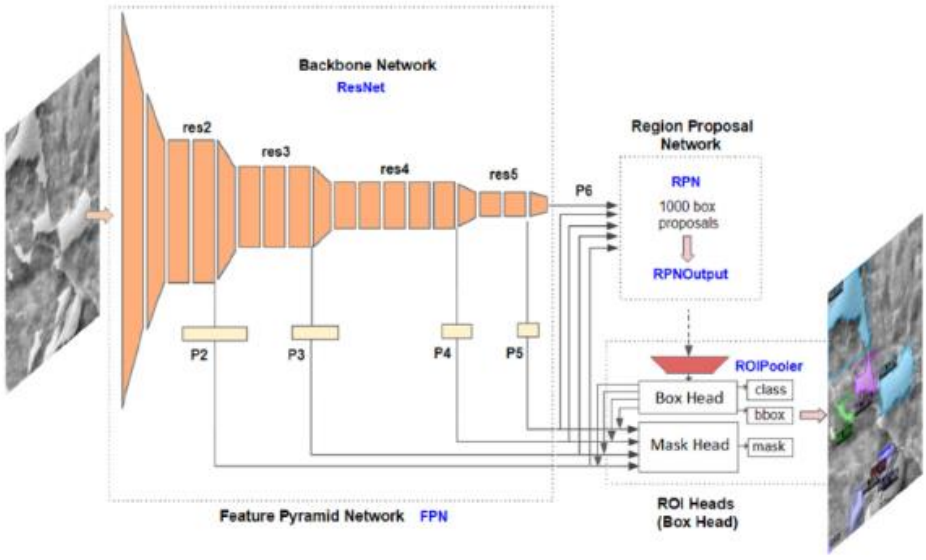


Figura 17. Esquema de la arquitectura de Detectron2. (Ackermann et al., 2022)

2.5 Toma de Imágenes con Dron

Las plataformas aéreas tradicionales, como aviones y satélites, no son adecuadas para estudios como éste debido a su baja resolución espacial y temporal. Por ello, la introducción de vehículos voladores no tripulados, comúnmente conocidos como drones, en aplicaciones del análisis y procesamiento de imágenes permiten la recolección de éstas de los diferentes terrenos con más facilidad y frecuencia. Además, los **UAVs** pueden suministrar imágenes incluso en días nublados, y el tiempo necesario para preparar e iniciar el vuelo es reducido, lo que permite una mayor flexibilidad a la hora de programar la adquisición de imágenes. Otras ventajas de los **UAVs** son su menor coste y su gran flexibilidad de configuración en comparación con las aeronaves

pilotadas, lo que permite utilizar y probar sensores de bajo coste, como las cámaras digitales convencionales. (Torres-Sánchez et al., 2014)

Las cámaras en primera persona que llevan equipadas estos vehículos han impactado significativamente en el control remoto de las actividades aéreas de reconocimiento y recolección de datos. Pueden llegar a transmitir señales de vídeo en directo con baja latencia y alta resolución. Algunos modelos incorporan giroscopios para conseguir imágenes estables y vídeos fluidos.

Los últimos avances se centran en mejorar la calidad de imagen y la facilidad de uso, haciendo hincapié en factores como el campo de visión (*FOV*) y la distancia focal del objetivo. Con distancias focales más cortas se consiguen *FOV* más amplios, que ofrecen vistas claras en todas las direcciones sin necesidad de maniobras de guiñada (Mokhtar et al., 2023). Los parámetros que influyen en estas distancias focales se pueden observar de forma visual en *Figura 18*.

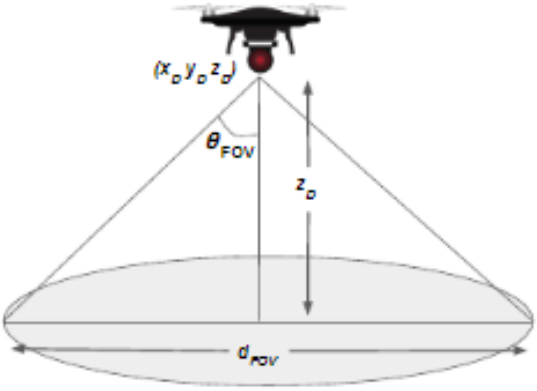


Figura 18. Campo de Visión de un UAV. (Bhagat & Sujit, 2020)

El sensor de imagen es una de las partes más importantes de la cámara de los **UAVs**. El mecanismo consiste en que estos sensores absorben la luz creando una carga que posteriormente se convierte en señal de tensión proporcional a su iluminación. Existen dos tipos de sensores de imagen a resaltar, los sensores **CCD** y los **CMOS**, presentados en *Figura 19*. Ambos sensores tienen sus ventajas. El sensor **CCD** muestra un buen rendimiento con escasa luz, un buen contraste de la imagen y ésta resulta más natural con poco ruido. Por otra parte, el sensor **CMOS** necesita un bajo consumo de energía,

produce una imagen de alta resolución y es más barato en comparación. (December 17 & 2019, s. f.)

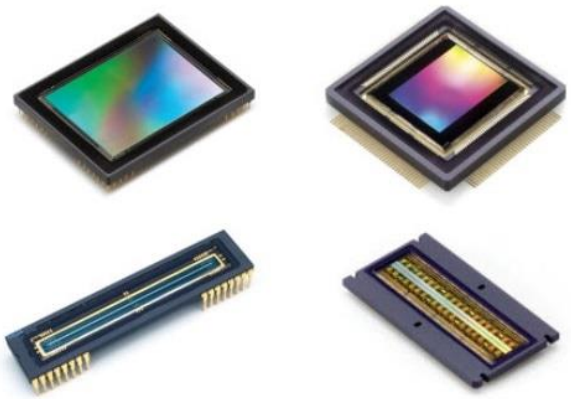


Figura 19. Sensores CCD y CMOS. (CCD vs CMOS – Which Is Better?, 2018)

En definitiva, la utilización de estos dispositivos en el ámbito de la monitorización agrícola y forestal es casi imprescindible.

2.6 Generación de Imágenes Sintéticas

Un problema común en el aprendizaje supervisado en técnicas de aprendizaje profundo es el requerimiento de una gran cantidad de datos para la fase del entrenamiento. En este caso, se necesita un número grande de imágenes junto con sus anotaciones de copas de árboles. Debido a esto, se suele recurrir a la generación de imágenes artificiales mediante varios métodos.

El método utilizado en (Firoze et al., 2023) hace referencia al estudio donde proponen usar brochas de generación procedural de ramas, y editar el resultado con operaciones como poda o curva de las ramas para lograr una imagen realista dentro del marco sintético (Palubicki et al., 2009).

Otros método popular es la generación de datos sintéticos con la ayuda de software **CAD** (*Computer-Aided Design*) donde se puede hacer uso de programas como **Blender** o **Unity** junto con representaciones digitales de los objetos a detectar (Arcidiacono, 2018). Se puede optar por recrear escenas lo más fieles posibles a la realidad pero requieren de mucho tiempo, por lo que la solución pasa por aleatorizar esta tarea de cierta manera para conseguir resultados similares sin tener que dedicar mucho esfuerzo (Tremblay et al., 2018). También existe el método en el cual se copia y pega la instancia a detectar

en otros entornos y fondos con el objetivo de que el algoritmo de aprendizaje sea capaz de localizar el elemento sin importar lo que le rodea.

Los últimos avances en las tareas de generación de datos e imágenes sintéticos tratan de introducir el uso de **Redes Adversarias Generativas** (*GANs*). Son modelos compuestos por dos redes neuronales, una red generativa y otra discriminativa (Islam & Zhang, 2020). Un esquema de una red GAN se puede apreciar en *Figura 20*. Durante la fase de entrenamiento, la parte generativa se encarga de generar imágenes falsas mientras que la parte discriminativa aprende a diferenciar cuáles son imágenes reales y cuáles son falsas. Al finalizar el entrenamiento, la red generativa puede utilizarse eficazmente para generar imágenes con un alto grado de realismo.

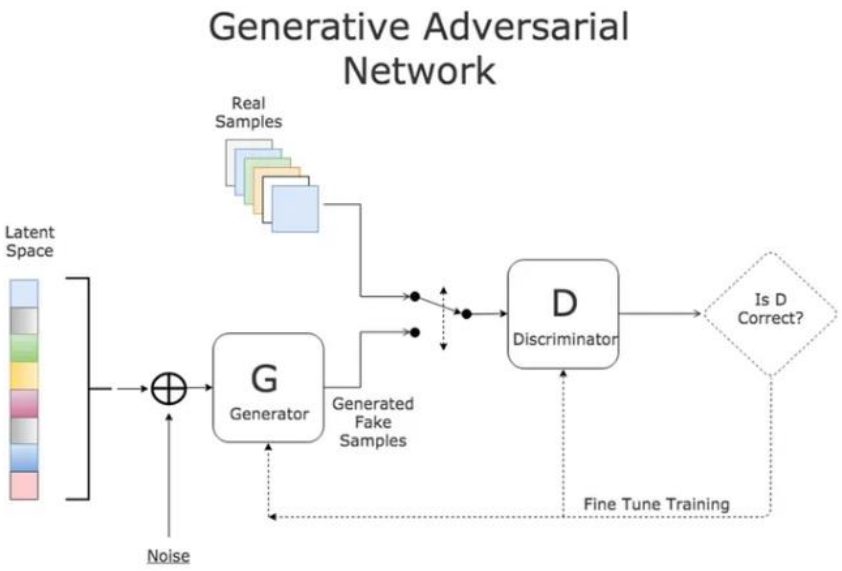


Figura 20. Esquema de la Arquitectura de una red GAN. (Ganz, 2020)

3. Implementación

Este capítulo está centrado en esclarecer el proceso llevado a cabo para la implementación y uso de los modelos de aprendizaje profundo escogidos, junto con los pasos previos en el tratamiento de las imágenes y las anotaciones.

Primeramente, se explica el preprocesamiento del conjunto de imágenes en la sección *3.1 Preprocesamiento del Conjunto de Datos*. Ahí se relatarán las técnicas seguidas para normalizar las imágenes, la obtención de las máscaras binarias de cada una de ellas, la ampliación del número de imágenes y finalmente la estandarización de las anotaciones en formatos aceptados por los modelos escogidos.

A continuación, en la sección *3.2 Métodos*, se detalla la puesta en funcionamiento de los modelos **Mask R-CNN**, **YOLOv8**, y **Detectron2**. Además, se muestran las pruebas, errores, soluciones tomadas y los posteriores resultados de sus ejecuciones. Cabe puntualizar que estos resultados no serán los definitivos para analizar el comportamiento real de los modelos.

Finalmente, en la sección *3.3 Equipo Usado para la Experimentación*, se describe brevemente las características del equipo utilizado en este proyecto.

3.1 Preprocesamiento del Conjunto de Datos

El conjunto de datos principal con el que se han realizado los experimentos con los distintos métodos seleccionados ha sido el proporcionado por (Firoze et al., 2023) en su repositorio de *GitHub*. Desde un primer momento se realizaron las pruebas con estas imágenes a esperas de saber si podríamos introducir otras obtenidas por el **Grupo de Investigación de Gráficos y Geomática de Jaén**.

La estructura del *dataset* viene dada por dos tipos de imágenes, unas reales, *Figura 21*, y otras sintéticas, *Figura 22*.

Figura 21. Imagen real.

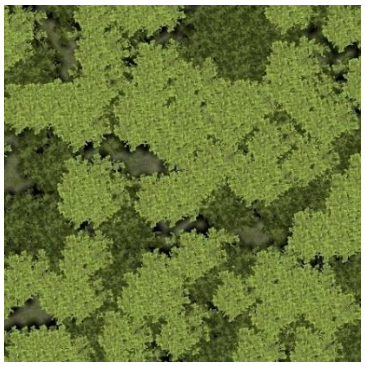


Figura 22. Imagen sintética.

Para cada subconjunto, se incluyen distintas anotaciones para cada imagen. Una imagen tiene asociado un archivo *JSON* y una carpeta con otros archivos e imágenes anotando las formas de las copas de los árboles. El *Código 1,* muestra un ejemplo del contenido de un archivo con las anotaciones de una imagen.

```
{
  "version": "5.0.1",
  "flags": {},
  "shapes": [
    {
      "label": "Tree",
      "points": [
        [
          2401.75,
          838.125
        ],
        [
          2348.625,
          819.375
        ],
        [
          2273.625,
          778.75
        ], . . .
```

Código 1. Ejemplo del contenido de las anotaciones JSON.

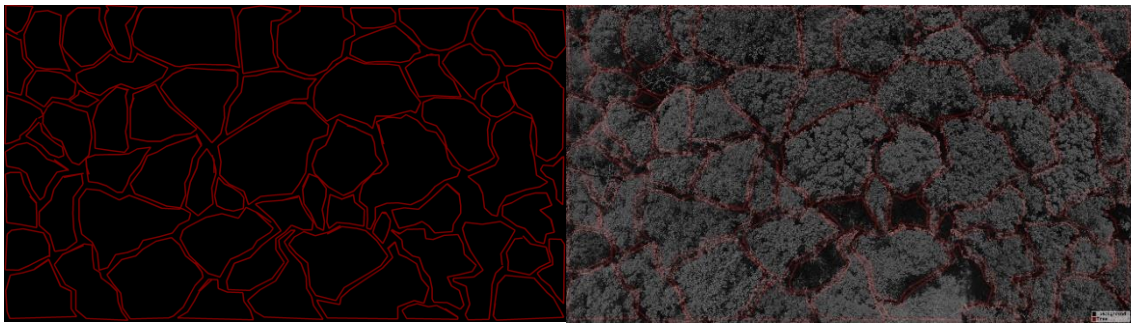


Figura 23. Ejemplo de anotaciones para una imagen del conjunto real.

En *Figura 23*, se pueden observar las diferentes formas geométricas indicando las delimitaciones de cada copa de árbol. Sin embargo, este tipo de anotaciones no son adecuadas para el entrenamiento de los modelos que se estudian en este trabajo. Esto se debe a que son muy particulares de este conjunto de datos y muy probablemente fueron útiles para los algoritmos personalizados que utilizaron los dueños de estas imágenes. Por ello, el primer paso a dar fue la adecuación de las anotaciones junto con un preprocesamiento de las imágenes.

```
función preprocesar_imagen(dir_imagen, nombre_imagen,
dir_imagen_preproc):

    imagen = redimensionar_imagen(leer_imagen(dir_imagen), 224, 224)
    imagen = normalizar_pixeles_imagen(imagen)

    crear_directorio_si_no_existe(dir_imagen_preproc + '/imagenes')

    guardar_imagen_numpy(imagen,
              dir_imagen_preproc + '/imagenes/' + nombre_imagen)
    guardar_imagen_png(imagen,
              dir_imagen_preproc + '/imagenes/imgpng/' + nombre_imagen +
'.png')

    mostrar_imagen(imagen)

    retornar imagen
```

Algoritmo 1. Pseudocódigo del preprocesamiento de una imagen.

Este *Algoritmo 1*, trata de preprocesar la imagen de tal forma que se redimensiona a un tamaño establecido de 224x224 píxeles. Después, se normalizan los valores RGB en un intervalo de [0, 255] y finalmente se guarda la nueva imagen en el directorio deseado de salida.

```
función preprocesar_etiqueta(dir_etiq, nombre_etiq, dir_etiq_preproc):

    data = leer_json(dir_etiq)

    lista_puntos = []

    Para cada forma en lista[formas]:

        Agregar puntos de forma a lista_puntos

    coords_normalizadas= []

    Para cada poligono en lista_puntos:

        coords = []

        Para cada coordenada en poligono:

            normalizar_coordenada(coordenada, coords)

        Agregar coords a coords_normalizadas

    mask = crear_mascara_vacia(224, 224)

    Para cada poligono en coords_normalizadas:

        dibujar_poligono_mascara(mask, poligono)

    crear_directorio_si_no_existe(dir_etiq_preproc + '/mascaras')

    mask_normalizada = normalizar_mascara(mask)

    guardar_mascara_numpy(mask, dir_etiq_preproc + '/mascaras/' +
nombre_etiq)

    guardar_mascara_png(normalized_mask, dir_etiq_preproc +
'/mascaras/maskpng/' + nombre_etiq + '.png')

    mostrar_mascara(mask)

    mostrar_mascara(mask_normalizada)

    devolver normalized_mask
```

Algoritmo 2. Pseudocódigo del preprocesamiento de las anotaciones.

El *Algoritmo 2* anterior, realiza la lectura de las anotaciones del *dataset* que vienen en formato JSON. Así que para cada una se ha considerado que forme una lista de polígonos, los cuales a su vez contienen una lista de coordenadas. Se determina que cada polígono está formado por los puntos contenidos en cada aparición de la etiqueta "*Tree*" en el archivo de anotaciones.

Además, dichas coordenadas se corresponden con la nueva dimensión de la imagen aplicada previamente. Por último, se hace uso de la función *polylines()* perteneciente al módulo de **OpenCV** para que dibuje esos polígonos en una máscara binaria normalizada en un intervalo de [*0, 1*], que se convertirá en la etiqueta válida para el modelo.

Tras aplicar el preprocesamiento a un conjunto de *40* imágenes reales, salieron como resultado las siguientes figuras: *Figura 24*, *Figura 25* y *Figura 26*.

Figura 24. Imagen Original antes de aplicar el preprocesamiento.

Figura 25. Imagen Preprocesada

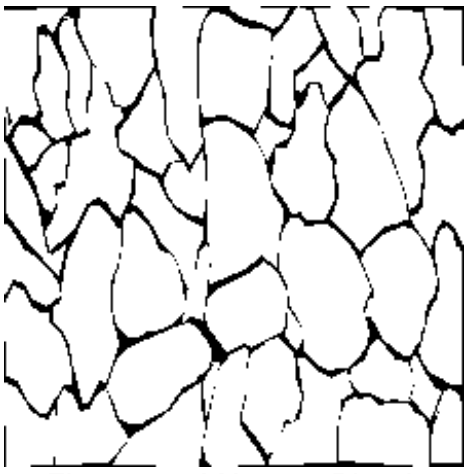


Figura 26. Máscara Binaria

Aplicando este mismo procedimiento al conjunto de 33 imágenes sintéticas, éstas quedarían de la siguiente forma en las figuras: *Figura 27*, *Figura 28* y *Figura 29*.

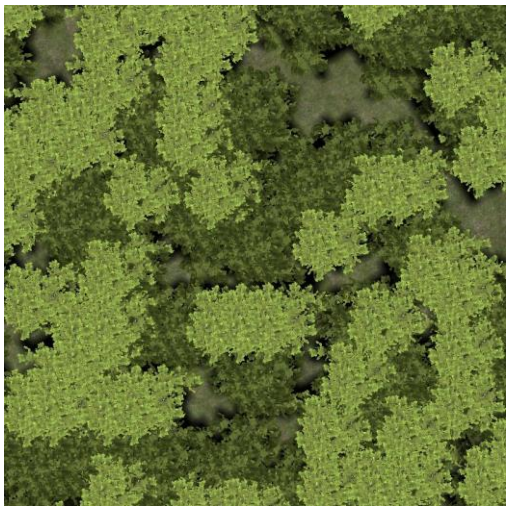


Figura 27. Imagen Sintética antes del preprocesamiento.

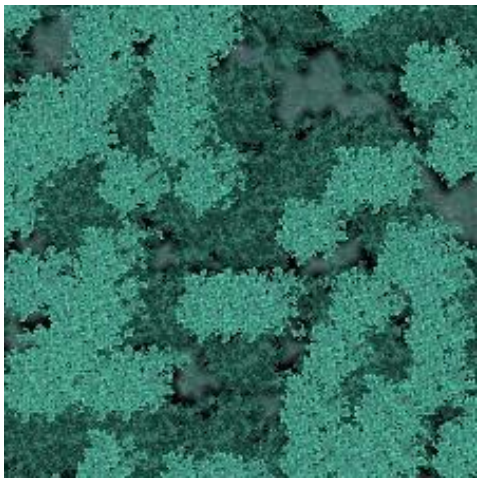


Figura 28. Imagen Preprocesada

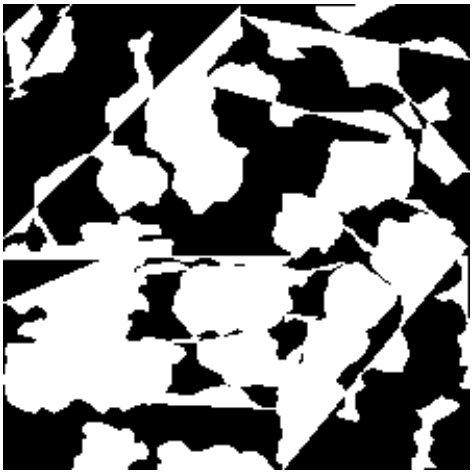


Figura 29. Máscara Binaria

Antes de proceder con la división de las imágenes en subconjuntos, es recomendable aplicar el aumento de las imágenes mediante la técnica sencilla de rotar las imágenes 90, 180 y 270 grados. Con esto, se pretende mejorar la precisión y robustez de los modelos al tener más casos y variedad de cierta forma artificial. En la *Tabla 1*, se muestra el número de imágenes previo a la aplicación de la rotación, y la cantidad resultante.

Conjunto de Imágenes	***Número Imágenes Originales***	***Número Imágenes con* Data Augmentation**
Imágenes Reales	50	200
Imágenes Sintéticas	33	132

Tabla 1. Cantidad de imágenes tras aplicar rotación.

Posteriormente, tras el aumento de imágenes, se procede a organizar el conjunto de éstas en dos subconjuntos, entrenamiento y validación. La subdivisión ha de realizarse de tal manera que exista cierta aleatoriedad en los datos, manteniendo también la proporción entre las imágenes de cada clase que compone cada subdivisión. En *Figura 30* se presenta la proporción escogida.

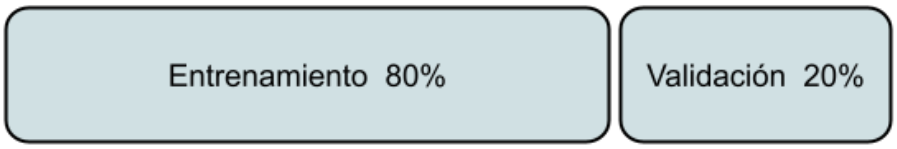


Figura 30. Distribución de las imágenes para el desarrollo de los experimentos.

En cuanto al conjunto de imágenes de test, se ha acordado de que será independiente al conjunto utilizado para el entrenamiento y comprobar si el modelo es capaz de detectar cualquier entidad arbórea. Por ello, no se ha tenido en cuenta para esta separación.

Una vez divididas las imágenes en los subconjuntos de entrenamiento y validación, es posible la conversión a los formatos aceptados por los modelos **YOLO** y **Detectron**. Éstos requieren formatos como el propio de *YOLO*, y *COCO JSON*, utilizado también por otros métodos similares en el estado del arte de la detección de objetos.

El primer paso, es la transformación de las máscaras binarias en un solo archivo **JSON** con la estructura *COCO* (*COCO - Common Objects in Context*, s. f.).

```
función anotación_imagenes(dir_mask):
    global image_id, annotation_id
    annotations = []
    images = []
    para cada categoría en category_ids.keys():
        para cada máscara en (dir_mask/Tree):
            original_file_name = máscara[0].png
            mask_image_open = cv2.imread(máscara)
            height, width, _ = máscara.dimension()

            # Creación de las anotaciones de cada imagen
            si original_file_name no está anotada:
```

```
                image = {
                    "id": image_id + 1,
                    "width": width,
                    "height": height,
                    "file_name": original_file_name,
                }
                images.añadir(image)
                image_id += 1
            si no:
                image = [element for element in images if
element['file_name'] == original_file_name][0]

            # Obtención de los contornos de las máscaras binarias
            gray = cv2.cvtColor(mask_image_open, cv2.COLOR_BGR2GRAY)
            _, thresh = cv2.umbral(gray, 0, 255, cv2.THRESH_BINARY +
cv2.THRESH_OTSU)
            contours = cv2.encontrarContorno(thresh, cv2.RETR_TREE,
cv2.CHAIN_APPROX_SIMPLE)[0]

            # Creación de las anotaciones para cada contorno
            para cada contorno en contornos:
                bbox = cv2.boundingRect(contour)
                area = cv2.contourArea(contour)
                segmentation = contour.flatten().tolist()

                annotation = {
                    "iscrowd": 0,
                    "id": annotation_id,
                    "image_id": image['id'],
                    "category_id": category_ids[category],
                    "bbox": bbox,
                    "area": area,
                    "segmentation": [segmentation],
                }
                si area > 0:
                    annotations.append(annotation)
                    annotation_id += 1
```

```
    retornar images, annotations, annotation_id
```

Algoritmo 3. Anotación de las máscaras de cada imagen.

```
función procesar_máscaras(dir_mask, dest_json):
    global image_id, annotation_id
    image_id = 0
    annotation_id = 0

    # Inicialización del archivo COCO JSON con sus categorías
    coco_format = {
        "info": {},
        "licenses": [],
        "images": [],
        "categories": [{"id": value, "name": key, "supercategory": key}
for key, value in category_ids.items()],
        "annotations": [],
    }

    # Creación de las secciones para las imágenes y sus anotaciones
    coco_format["images"], coco_format["annotations"], annotation_cnt =
anotación_imágenes(dir_mask)

    # Guardar el archivo COCO JSON
    guardarJSON(coco_format, dest_json)
```

Algoritmo 4. Creación del archivo JSON y procesamiento de las máscaras.

Los *Algoritmo 3* y *Algoritmo 4* anteriores se encargan de convertir las máscaras binarias del conjunto de imágenes en anotaciones *COCO JSON*, obteniendo los contornos de los polígonos correspondientes a cada árbol mediante métodos propios de **OpenCV**. Con estos contornos, pueden transformarse en coordenadas y también calcular las cajas delimitadoras, las cuales serán cruciales en la fase de entrenamiento de los modelos. Finalmente, con todas las anotaciones de las imágenes recolectadas, se crea un archivo **.json** con el contenido de la segmentación de los árboles junto con las categorías que en nuestro caso será tan solo "*Tree*". Un ejemplo del contenido de estos archivos se puede ver en *Código 2*.

```
{
    "annotations": [
        {
            "area": 1633.5,
            "bbox": [
                0,
                184,
                81,
                40
            ],
            "category_id": 1,
            "id": 0,
            "image_id": 1,
            "iscrowd": 0,
            "segmentation": [
                [
                    69,
                    184,
                    68,
                    185,
                    64,
      . . .
],
    "categories": [
        {
            "id": 1,
            "name": "Tree",
            "supercategory": "Tree"
        }
    ],
    "images": [
        {
            "file_name": "imagen54.png",
            "height": 224,
            "id": 1,
            "width": 224
        }, . . .
```


Código 2. Ejemplo resultado de las anotaciones transformadas a COCO JSON.

El siguiente paso es utilizar este formato *COCO JSON* para transformar las anotaciones en el formato requerido por los modelos **YOLO**. Estos modelos, necesitan de un único archivo **.yaml**, como se muestra en *Código 3*, que contiene los directorios de los subconjuntos de entrenamiento y validación, el número de categorías y sus nombres.

```
names:
- Tree
nc: 1
test: ''
train: "/.../annotated_forest_dataset/yolo_dataset/train/images"
val: "/.../annotated_forest_dataset/yolo_dataset/valid/images"
```

Código 3. Contenido del archivo data.yaml.

```
función convertir_a_yolo(input_images_path, input_json_path,
output_images_path, output_labels_path):

    # Abrir el archivo JSON que contiene las imágenes anotadas
    data = json.abrir(input_json_path)

    # Guardar el nombre de las imágenes del conjunto en una lista
    file_names = []
    para cada archivo en (input_images_path):
        si archivo termina en ".png":
            source = unir(input_images_path, archivo)
            destination = unir(output_images_path, archivo)
            copiar(source, destination)
            file_names.añadir(archivo)

    # Iterar sobre cada archivo y procesar cada imagen
    para cada archivo en file_names:
        img = obtenerImagen(archivo)
        img_id = img['id']
        img_w = img['width']
        img_h = img['height']
        img_ann = obtenerAnotacionImagen(img_id, data)

        # Registrar el polígono normalizado en el archivo de texto
```

```
        si existe img_ann:
            crearArchivo(unir(output_labels_path, (archivo)[0].txt"):
                para cada anotación en img_ann:
                    current_category = anotación['category_id'] - 1
                    polygon = anotación['segmentation'][0]
                    normalized_polygon = normalizar(coord) para cada c,
coord in polígonos
                    registrar((current_category) + (normalized_polygon))
```

Algoritmo 5. Creación de las anotaciones en formato YOLO.

Esta vez, en *Algoritmo 5*, las anotaciones se guardan en archivos individuales correspondientes a cada foto en vez de contenerse en uno solo como en el caso de *COCO JSON*. Así que, para cada imagen, existe un archivo de texto con las coordenadas de los polígonos que representan las copas de los árboles. En *Código 4* se observa un ejemplo de estas anotaciones, donde el primer dígito indica el número de la categoría, y el resto son los que forman el polígono.

```
0 0.000000 0.875000 0.000000 0.897321 0.004464 0.901786 0.004464 0.924107
0.017857 0.910714 0.022321 0.910714 0.026786 0.906250 0.035714 0.906250
0.040179 0.910714 0.049107 0.910714 0.053571 0.915179 0.058036 0.915179
0.062500 0.919643 0.071429 0.910714 0.080357 0.910714 0.084821 . . .
```

Código 4. Contenido de las anotaciones de las imágenes en formato YOLO.

Tras tener todas las imágenes con diferentes formatos de anotaciones como máscaras binarias, *COCO JSON* y *YOLO*, se puede proceder al entrenamiento de los modelos y comprobar qué resultados se obtienen.

3.2 Métodos

A lo largo del desarrollo de este trabajo, se han probado diferentes técnicas hasta encontrar aquellas que han dado resultados con el conjunto de datos que se dispone. En esta sección, se detallan los pasos seguidos, las pruebas, problemas encontrados a lo largo de los experimentos y las soluciones adoptadas.

3.2.1 Mask R-CNN de Tensorflow

Las primeras semanas de investigación se centraron en explorar material, documentación y código sobre cómo implementar una red CNN del tipo **Mask R-CNN**. Se encontró un repositorio en *GitHub* bastante reconocido por la comunidad debido a su buena implementación de dicha red utilizando el framework **TensorFlow**, aunque solo es compatible con las versiones **1.x** de éste. (*matterport/Mask_RCNN*, 2017/2024).

Se intentó utilizar en diferentes equipos de manera local siguiendo las instrucciones de instalación creando un entorno dedicado para los requisitos con el gesto de paquetes **Anaconda**, y también en entornos virtuales como los que ofrece **Google Colab**. En todos los intentos dio fallos de compatibilidad, puesto que en la ejecución no encontraba los módulos necesarios aun estando instalados. Por otro lado, partes del código propio de la implementación estaban obsoletas, las cuales impedían que el sistema tanto local como virtual las reconociese.

Cabe destacar, que a partir de las últimas versiones de **Google Colab** desde mayo de 2023, cambiaron diferentes especificaciones como por ejemplo el tiempo máximo de aceleración GPU y requerimientos mínimos de versiones de diferentes librerías como las de **Keras**. Por tanto, fue otro factor influyente al no poder llevar a cabo los experimentos con esta implementación.

Aun así, se siguió buscando otras implementaciones más actuales y con versiones de los *frameworks* y librerías más recientes como la presentada por (Gad, 2020/2024). Este repositorio tiene el código actualizado para **TensorFlow 2.2.0** y los requerimientos de bibliotecas también con versiones más recientes, lo cual aparentemente no debería de dar ningún problema. Pero los hubo y tampoco hubo éxito instalándola. Había módulos del código utilizando paquetes de Keras obsoletos como *keras.engine* y que actualmente pertenecen a otras dependencias con nombres distintos. No era un fallo catastrófico, pero refactorizar todo el código de la implementación requeriría mucho tiempo y se decidió no invertirlo e intentar otras opciones.

3.2.2 Red Neuronal Convolucional

La siguiente opción fue enfocar el estudio en búsquedas de técnicas más sencillas y básicas, para partir de ahí como base. Por tanto, se utilizó el libro (*Deep Learning with Python, Second Edition*, s. f.) como referencia. En él se proponía una estructura de red convolucional utilizando TensorFlow y Keras que se presenta en *Código 5*.

```
función definición_modelo(tam_img, num_clases):

    inputs = keras.Input(shape=tam_img + (3,))

    x = layers.Rescaling(1./255)(inputs)

    x = layers.Conv2D(64, 3, activation="relu")(x)

    x = layers.Conv2D(64, 3, activation="relu")(x)

    x = layers.Conv2D(128, 3, activation="relu")(x)

    x = layers.Conv2D(128, 3, activation="relu")(x)

    x = layers.Conv2D(256, 3, activation="relu")(x)

    x = layers.Conv2D(256, 3, activation="relu")(x)

    x = layers.Conv2DTranspose(256, 3, activation="relu")(x)

    x = layers.Conv2DTranspose(256, 3, activation="relu")(x)

    x = layers.Conv2DTranspose(128, 3, activation="relu")(x)

    x = layers.Conv2DTranspose(128, 3, activation="relu")(x)

    x = layers.Conv2DTranspose(64, 3, activation="relu")(x)

    x = layers.Conv2DTranspose(64, 3, activation="relu")(x)

    outputs = layers.Conv2D(num_clases, 3, activation="softmax")(x)

    modelo = keras.Model(inputs, outputs)

    return modelo
```

Código 5. Esquema de la red convolucional del libro Deep Learning with Python. (Deep Learning with Python, Second Edition, s. f.)

Para entrenar esta red con el conjunto de datos se hicieron algunas modificaciones a las funciones que proporcionan para adaptar la implementación

a este conjunto, como por ejemplo la lectura de imágenes y anotaciones. Supuestamente parecía que iba a funcionar, pero tras varios intentos de entrenamiento no dio el resultado esperado. Devolvía unas imágenes, *Figura 31*, con unas máscaras que no se deberían de corresponder con lo que supuestamente ha aprendido. Se probó a cambiar los parámetros de entrenamiento por si necesitaba dar más iteraciones, se añadieron más imágenes y anotaciones al conjunto de datos, y se alteró ligeramente el modelo que ofrecía el libro. Sin embargo, no se logró una segmentación de ninguna entidad en las imágenes.

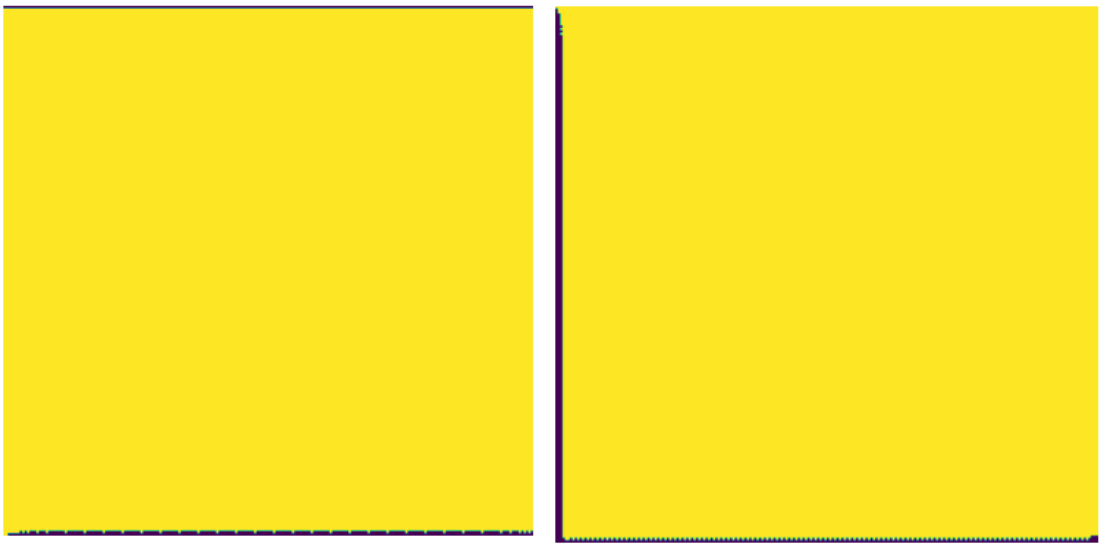


Figura 31. Ejemplos de las "detecciones" de los árboles tras el entrenamiento con la red convolucional.

Después, se refactorizó parte del código para poder ejecutarlo en el **clúster de ADA** que tendría los recursos suficientes para poder ejecutar el entrenamiento de esta red con el conjunto de datos de mascotas que utilizan en el libro inicialmente puesto que en Google Colab la memoria *RAM* no era capaz de procesar las más de 7000 imágenes (*Visual Geometry Group - University of Oxford*, s. f.). Una vez pasados los archivos al clúster mediante *FileZilla*, se pudo ejecutar el script y tardó unas 3 horas en terminar el entrenamiento con las 7390 imágenes y dar el siguiente resultado mostrado en la *Figura 32*. En esta figura, se puede intuir el contorno y forma de una mascota, pero aún con algunos fallos en la segmentación.

Figura 32. Resultado de segmentación de una mascota con la red convolucional del libro Deep Learning with Python.

Dados estos resultados se concluyó que la red convolucional necesitaba de miles de imágenes para el entrenamiento para dar unos resultados decentes, y quizás no es adecuada para entidades como los árboles al estar muy juntos unos con otros y no haber una diferencia clara entre ellos.

3.2.3 Mask R-CNN de Pytorch

De forma paralela también se comenzaron experimentos con la implementación del modelo **Mask R-CNN** que ofrece *Pytorch* (*Mask R-CNN — Torchvision main documentation*, s. f.). Se realizó la ejecución con el *dataset* de ejemplo, con alrededor de unas 170 imágenes, el cual se asimila en dimensión al que disponemos. Las anotaciones que se le pasan son las máscaras binarias que contienen 0 o 1. Después de un entrenamiento con unas 10 iteraciones, el resultado se puede comprobar en las figuras: *Figura 33* y *Figura 34.*

Figura 33. Imagen test para la detección de peatones.

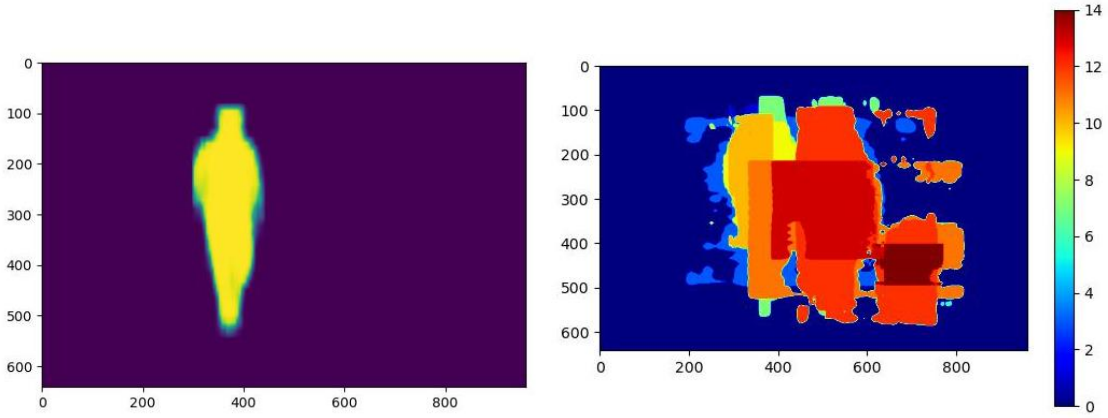


Figura 34. Segmentación de un peatón, e instanciación de todos los peatones en la imagen.

La modificación añadida para los resultados fue la fusión de máscaras, puesto que como ficheros resultantes dibujaba la segmentación de las instancias en imágenes separadas como se puede ver en la imagen izquierda de la *Figura 34.* De esta forma, en la imagen de la derecha de la *Figura 34* se puede apreciar mejor en conjunto cómo ha segmentado las personas en este caso.

No obstante, en el momento de realizar las pruebas con nuestro conjunto de datos dio errores en el momento de ejecución y con el tiempo se abandonó.

3.2.4 Modelos YOLOv8 y Detectron2

En esta sección se presentan las implementaciones de las últimas técnicas probadas en este trabajo. El motivo de estas elecciones fue principalmente la sencillez de su implementación y los modelos preentrenados que ofrecen. Estas redes tienen unos pesos y son capaces de detectar objetos en las imágenes sin necesidad de realizar un entrenamiento desde cero. Por lo cual, solo necesitan un entrenamiento enfocado al ajuste de la identificación y segmentación de nuestras clases particulares.

Para el entrenamiento de los modelos **YOLOv8** y **YOLOv9**, se instancian los pesos de los modelos preentrenados específicos para las tareas de segmentación como se realiza en *Código 6*.

```
Modelv8 = YOLO('yolov8n-seg.pt')

Modelv9 = YOLO('yolov9e-seg.pt')
```

Código 6. Instanciación de los modelos YOLOv8 y YOLOv9.

Una vez instanciados, se pueden configurar diferentes aspectos como la carpeta con los resultados tras el entrenamiento y algunos parámetros relacionados con éste mostrados en *Código 7*.

```
results = Modelv8.train(data='/.../yolo_dataset/data.yaml',
                        project=/results/yolov8,
                        name=200_epochs-,
                        epochs=200,
                        patience=0,
                        batch=4,
                        imgsz=224)
```

Código 7. Ejemplo de configuración de entrenamiento para el modelo YOLOv8.

Después de unas iteraciones de prueba para comprobar que funciona correctamente y sin errores, da unos resultados prometedores utilizando tan solo el conjunto de imágenes sintéticas en el entrenamiento. Un ejemplo de estos resultados se presenta en la *Figura 35*.

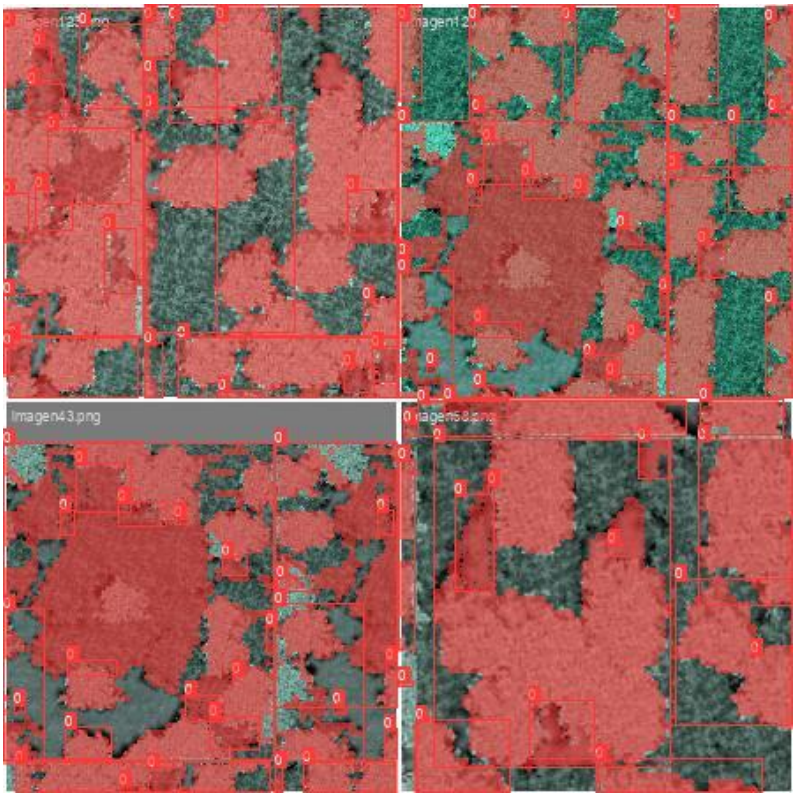


Figura 35. Resultado de segmentación tras 200 iteraciones con YOLOv8.

Por otro lado, para el modelo **Detectron2** solo hace falta instalar el repositorio correspondiente de *GitHub* (*facebookresearch/detectron2*, 2019/2024). Tienen diferentes modelos para elegir, y en este estudio se ha elegido el propio **Mask R-CNN** con un *backbone* de **ResNet50** y **FPN** como *head* mostrado en el *Código 8*.

```
cfg.MODEL.WEIGHTS = model_zoo.get_checkpoint_url("COCO-
InstanceSegmentation/mask_rcnn_R_50_FPN_3x.yaml")
```

Código 8. Instanciación del modelo Detectron2 con Mask R-CNN.

Algunos aspectos de la configuración de este modelo difieren ligeramente de **YOLO**, pero en esencia son los mismos parámetros. Un ejemplo de configuración se puede ver en *Código 9*.

```
cfg.OUTPUT_DIR = f"{path_dir}/Detectron2_models"

cfg.merge_from_file(model_zoo.get_config_file("COCO-
InstanceSegmentation/mask_rcnn_R_50_FPN_3x.yaml"))

cfg.DATALOADER.NUM_WORKERS = 2

cfg.MODEL.WEIGHTS = model_zoo.get_checkpoint_url("COCO-
InstanceSegmentation/mask_rcnn_R_50_FPN_3x.yaml")

cfg.SOLVER.IMS_PER_BATCH = 2

cfg.SOLVER.BASE_LR = 0.00025

cfg.SOLVER.MAX_ITER = 1000

cfg.MODEL.ROI_HEADS.BATCH_SIZE_PER_IMAGE = 256

cfg.MODEL.ROI_HEADS.NUM_CLASSES = 1 # Solo hay una clase, Tree
```

Código 9. Ejemplo de configuración de entrenamiento de Detectron2.

Después de las mil iteraciones para comprobar su funcionamiento, se obtienen resultados que segmentan y detectan los árboles en la *Figura 36*.

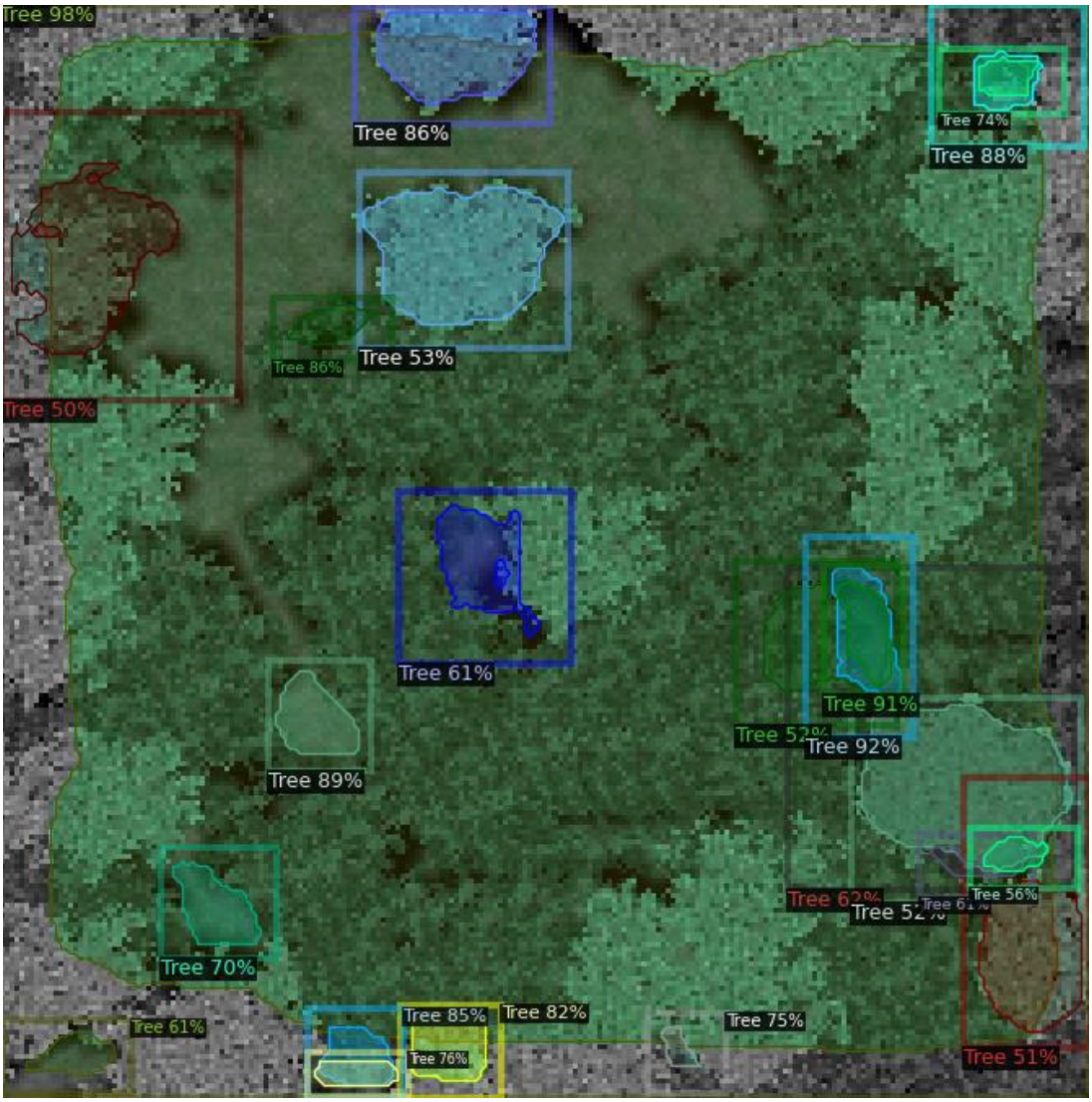


Figura 36. Resultado de segmentación tras 1000 iteraciones con Detectron2.

No son del todo precisas, pero es un objetivo conseguido tras las innumerables pruebas con las técnicas previamente mencionadas. Por tanto, los modelos **YOLOv8**, **YOLOv9** y **Detectron2** han sido las técnicas de segmentación elegidas para el desarrollo del estudio. En el siguiente capítulo se presentan los resultados obtenidos con diferentes configuraciones y qué tal se comportan realizando la inferencia sobre imágenes de zonas arbóreas desconocidas en un principio por éstos.

3.3 Equipo Usado para la Experimentación

Dado que el objetivo principal de este proyecto es identificar el modelo óptimo considerando el trabajo y tiempo invertidos, es fundamental describir los componentes del equipo utilizado en los experimentos para proporcionar un

contexto adecuado a todas las pruebas realizadas. En la *Tabla 2* a continuación se enumeran todos los componentes relevantes teniendo en cuenta la naturaleza de este experimento.

Componente	***Modelo***
CPU	Intel® Xeon® CPU 2.00GHz
Memoria RAM	12.7 GB
Disco Duro	78.2 GB
GPU	Tesla T4 15 GB

Tabla 2. Especificaciones del Equipo para los Experimentos.

4. Resultados

En este capítulo se presentan los resultados obtenidos tras la implementación de cada uno de los métodos seleccionados, **YOLOv8**, **YOLOv9** y **Detectron2**.

En primer lugar, en la sección *4.1 Métricas de Evaluación*, se describen las métricas de evaluación para medir el rendimiento de los modelos después del entrenamiento. Seguidamente, se detallan las métricas resultantes de cada modelo junto con un análisis cualitativo sobre las imágenes del conjunto de datos utilizado para el entrenamiento y validación en las secciones, *4.2 Resultados YOLOv8*, *4.3 Resultados YOLOv9* y *4.4 Resultados Detectron2*. Además, se realizarán las pruebas de segmentación con las imágenes del conjunto de test.

Finalmente, en la sección *4.5 Comparación General*, se mostrará una comparación general con todos los métodos, se desarrollarán cuáles son las fortalezas y debilidades de cada uno en el contexto de la segmentación de copas de árboles, y se determinará cuál de ellos es el adecuado para este estudio.

4.1 Métricas de Evaluación

Las métricas de evaluación son fundamentales para valorar el rendimiento y la eficacia de los modelos de aprendizaje automático y profundo. Proporcionan información sobre el rendimiento de un modelo y ayudan a comparar diferentes modelos o algoritmos. Son de gran ayuda para realizar mejoras, cambios en los modelos y discernir cuál de ellos es el idóneo y eficiente para llevar a cabo la tarea en cuestión.

En nuestro caso, las métricas principales escogidas son **Precisión** (*precision*), **Sensibilidad** (*recall*), **Intersección sobre Unión** (*IoU*), **mAP50** (*mean average precision IoU=0.5*) y **mAP50-95** (*mean average precision IoU [0.5, 0.95]*). Antes de explicar cómo calcular las métricas anteriores, es necesario introducir una serie de conceptos:

- **Verdaderos Positivos (TP):** número de instancias clasificadas correctamente por la red como pertenecientes a la clase objetivo. *Ejemplo*: Si el modelo detecta un árbol correctamente se cuenta como un verdadero positivo.

- **Verdaderos Negativos (TN):** número de instancias clasificadas correctamente por la red como no pertenecientes a la clase objetivo. *Ejemplo*: Si el modelo correctamente no detecta un objeto en una región donde no hay un árbol, se cuenta como un verdadero negativo.
- **Falsos Positivos (FP):** número de instancias clasificadas incorrectamente como pertenecientes a la clase objetivo cuando no lo son.

  Ejemplo: Si el modelo detecta algo como un árbol, pero en realidad no hay un árbol en esa posición, la predicción es incorrecta, se cuenta como un falso positivo.
- **Falsos Negativos (FN):** número de instancias que pertenecen a la clase objetivo, pero no fueron clasificadas correctamente. *Ejemplo*: Si el modelo no detecta un árbol que realmente está presente en la imagen, falla en la detección, se cuenta como un falso negativo.

Normalmente, la forma de presentación de estos conceptos es en una matriz de confusión, como se puede ver en *Figura 37*.

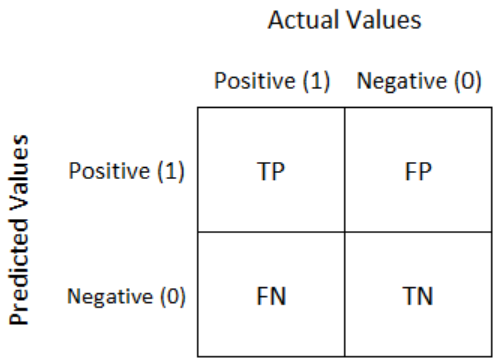


Figura 37. Esquema de matriz de confusión. (Izco, s. f.)

Una vez definidos los términos anteriores, se puede comprender mejor el cálculo de las siguientes métricas:

- **Precisión:** La precisión mide la proporción de detecciones correctas sobre el total de detecciones realizadas por el modelo. La *Función 3*

describe esta medida. Una alta precisión indica que el modelo comete pocos errores de detección.

$$Precision = \frac{TP}{TP + FP}$$

Función 3. Cálculo de la precisión. (Batallas et al., 2020)

- **Recall:** mide la proporción de instancias positivas correctamente detectadas sobre el total de instancias positivas reales. Un alto valor de esta medida indica que el modelo detecta la mayoría de las instancias positivas. En *Función 4* se puede comprender cómo se calcula el recall.

$$Recall = \frac{TP}{TP + FN}$$

Función 4. Cálculo del recall. (Batallas et al., 2020)

- **IoU:** mide la superposición entre el cuadro delimitador predicho y el cuadro delimitador verdadero. En la *Función 5* se ejemplifican varias formas de cómo obtener esta métrica.
  - **IoU = 0**: No hay superposición.
  - **IoU = 1:** Predicción perfecta (superposición completa).
  - **0 < IoU < 1:** Existe cierta superposición entre los cuadros delimitadores.

$$\text{IoU} = \frac{|X \cap Y|}{|X \cup Y|} \in [0, 1] \qquad \text{IoU} = \frac{\text{TP}}{\text{TP} + \text{FP} + \text{FN}} \in [0, 1]$$

Función 5. Cálculo de la Intersección sobre Unión. (Rainio et al., 2024)

- **AP:** es el área bajo la curva de precisión-recall. Se calcula integrando la precisión a diferentes niveles de recall como se puede ver en la *Función 6*.

$$\mathrm{AP} = \int_0^1 p(r)dr$$

Función 6. Integral para calcular la precisión media AP. (Average Precision - Testing with Kolena, s. f.)

También hay otra forma de calcular AP sin la necesidad de recurrir a integrales como propone la *Función 7*.

$$\boldsymbol{AP} = \sum_{\boldsymbol{k=0}}^{\boldsymbol{k=n-1}} [\boldsymbol{Recalls}(\boldsymbol{k}) - \boldsymbol{Recalls}(\boldsymbol{k+1})] * \boldsymbol{Precisions}(\boldsymbol{k})$$
$$\boldsymbol{Recalls}(\boldsymbol{n}) = \boldsymbol{0}, \boldsymbol{Precisions}(\boldsymbol{n}) = \boldsymbol{1}$$
$$\boldsymbol{n} = \boldsymbol{Number\ of\ thresholds}.$$

Función 7. Fórmula para el cálculo de la precisión media AP. (Mean Average Precision (mAP) Explained, s. f.)

- **mAP:** es el promedio de las precisiones medias (**AP**) calculadas para todas las clases en el conjunto de datos y, en diferentes umbrales de IoU. En *Función 8* se muestra su obtención matemáticamente.

$$\mathrm{mAP} = \frac{1}{N} \sum_{i=1}^{N} \mathrm{AP}_i$$

Función 8. Cálculo de la precisión media según el número de clases a detectar. (Mean Average Precision (mAP) Explained, s. f.)

A continuación, se detallan las dos variantes que se van a utilizar:

  - **mAP50:** precisión media calculada con un umbral de intersección sobre unión (**IoU**) mayor o igual a 0.50. Es una medida de la precisión del modelo considerando sólo las detecciones "fáciles".
  - **mAP50-95:** Media de la precisión media calculada con distintos umbrales de **IoU**, que van de 0.50 a 0.95. Ofrece una visión global del rendimiento del modelo en distintos niveles de dificultad de detección. (*YOLO Performance Metrics - Ultralytics YOLO Docs*, s. f.).

4.2 Resultados YOLOv8

Los primeros resultados de este estudio se obtienen tras la implementación y ejecuciones varias de **YOLOv8**. En dichas ejecuciones se modificaron parámetros como las iteraciones o *epochs*, el *dropout*, las iteraciones de *calentamiento* o *warmup epochs*, y por último la *paciencia*. Todo esto, para mejorar la arquitectura del modelo para nuestro conjunto de datos y no caer en el sobreajuste. Este término, **sobreajuste**, se define como la adaptación del modelo demasiado bien a los datos de entrenamiento, hasta el punto de captar el ruido y peculiaridades, significando un empeoramiento de los resultados con datos desconocidos.

Antes de pasar a la interpretación de los datos, es necesario dar unas breves introducciones a los conceptos ligados a los parámetros modificados.

Los **epochs** se asemejan a cuántas veces el modelo pasará por todo el conjunto de datos completo durante el entrenamiento. Mientras que el **dropout**, definido anteriormente en *Capa de Dropout*, se refiere a la técnica de regularización en la que se desactivan aleatoriamente un porcentaje de neuronas en una capa durante el entrenamiento para evitar el sobreajuste. Por otro lado, las **iteraciones de calentamiento** son aquellas primeras épocas del entrenamiento donde se utiliza una tasa de aprendizaje más baja, permitiendo así una estabilización previa al comienzo del entrenamiento con la tasa normal. Finalmente, el concepto de **paciencia** o **early stopping**, define el número de épocas adicionales que el modelo debe entrenar sin mejorar en la métrica de validación antes de detener el entrenamiento. Con este último parámetro, nos ha permitido conocer el punto óptimo local cuando el modelo no ha mejorado sus métricas en *100* iteraciones. En nuestro caso, esto ocurrió en la iteración *350* aproximadamente.

Si bien es cierto que, en la *Tabla 3,* no se aprecia el sobreajuste en la iteración *400* sino una mejoría. Pero sí que entre *400* y *600* se aprecia una ligera bajada. Esto puede deberse a que las métricas estiman una media de entre todas las iteraciones, y aún en la época *400* no había una clara disminución de los valores que las afectasen. Por tanto, esto nos ayuda a confirmar que, para este trabajo, y con los datos que disponemos, la configuración destacada de las

iteraciones es de [*350, 400*] y que no es aconsejable pasar de estas cifras. Así, obtenemos los mejores valores, ahorramos tiempo de ejecución y recursos computacionales.

Epochs	***Precisión***	***Recall***	***mAP50***	***mAP50-95***
200	0.129	0.064	0.032	0.008
400	0.153	0.070	0.036	0.010
600	0.132	0.068	0.032	0.009

Tabla 3. Evaluación de Segmentación de YOLOv8.

En la *Figura 38*, se presenta la predicción de segmentación realizada por el mejor modelo **YOLOv8** a las *350* iteraciones. Se trata de cuatro imágenes pertenecientes al conjunto de entrenamiento. En las máscaras resultantes, podemos apreciar una segmentación precisa en algunas de ellas como las dos imágenes superiores. Mientras que, en las inferiores, se puede intuir cierta inexactitud al dibujar las formas de los árboles detectados.

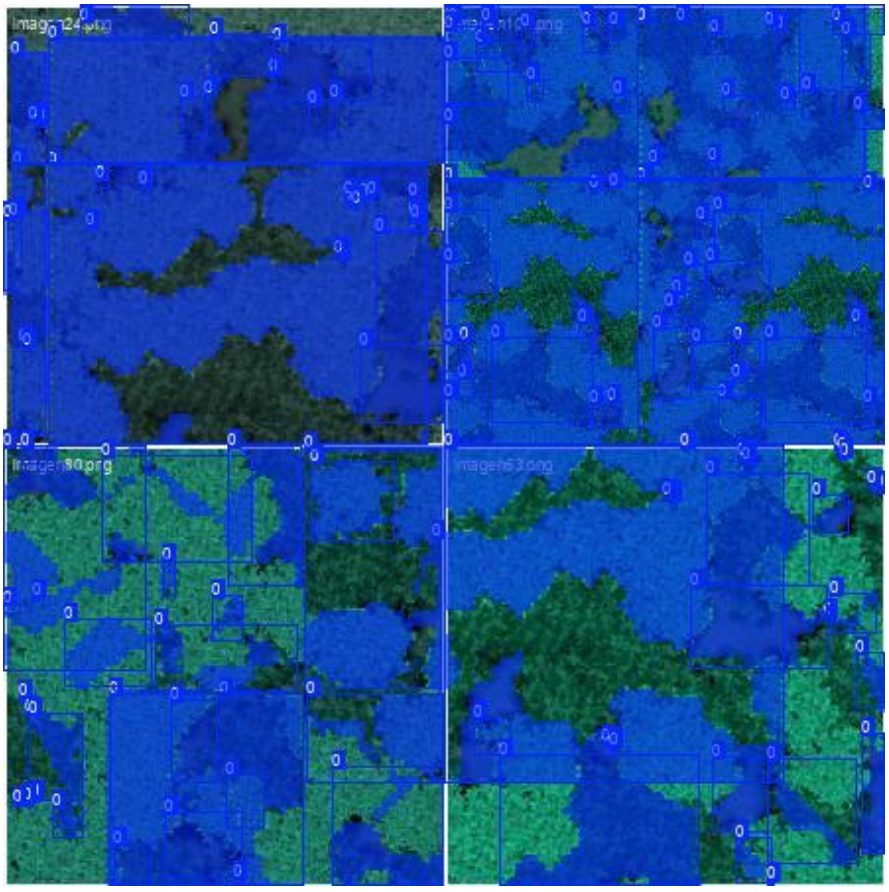


Figura 38. Ejemplo de segmentación con imágenes del conjunto del entrenamiento.

Una vez entrenado el modelo **YOLOv8**, probamos su eficiencia realizando la inferencia con el conjunto de imágenes de test completamente desconocidas

por este modelo. En la *Tabla 4* se muestra una comparativa con las imágenes originales a la izquierda, y las predicciones de éstas a la derecha. Se puede ver que, en algunos casos, sí que es capaz de identificar y segmentar árboles de manera individual. Sin embargo, se repite bastante el caso en el que engloba masas de árboles bajo una misma segmentación más amplia.

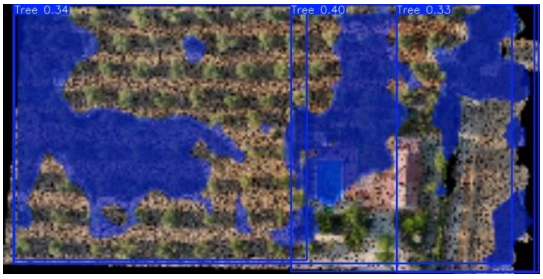
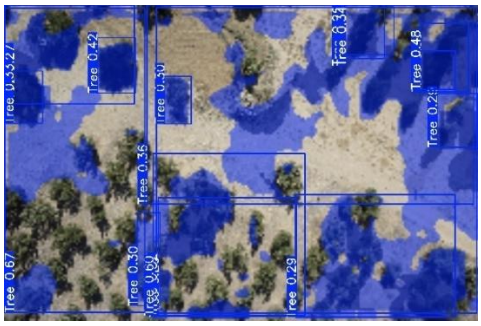
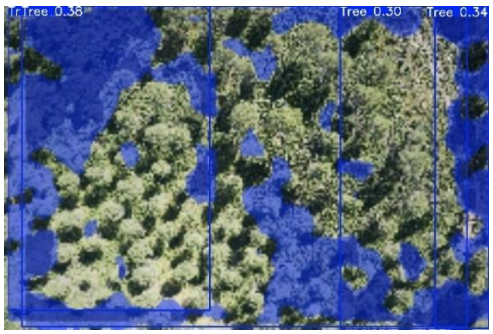
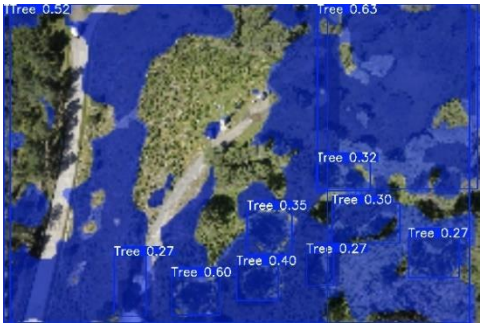
Imagen Original	Imagen Segmentada
	
	
	
	

Tabla 4. Inferencia de YOLOv8 sobre imágenes de test.

4.3 Resultados YOLOv9

Para la obtención de los resultados de **YOLOv9** procedemos con la misma configuración anterior, repitiendo el entrenamiento con diferentes iteraciones con el objetivo de comprobar cuál es la óptima.

El modelo prometía mejores métricas según su superioridad de parámetros internos sobre el modelo anterior, **YOLOv8**. Sin embargo, en la *Tabla 5* se puede comprobar el mal rendimiento que ha tenido. Esto nos da la indicación de que no es el modelo más adecuado para nuestro problema en particular, o que el conjunto de datos no era el mejor para sacar el máximo de esta arquitectura.

Epochs	***Precisión***	***Recall***	***mAP50***	***mAP50-95***
200	0.061	0.041	0.014	0.003
400	0.109	0.058	0.025	0.007
600	0.113	0.059	0.026	0.007

Tabla 5. Evaluación de Segmentación de YOLOv9.

Entre las iteraciones *400* y *600*, han trascurrido un par de horas más para la diferencia del entrenamiento, y no parecen demostrar un incremento notable como para justificar esas horas como productivas. Mediante una inspección visual de su comportamiento ante las imágenes de test en .

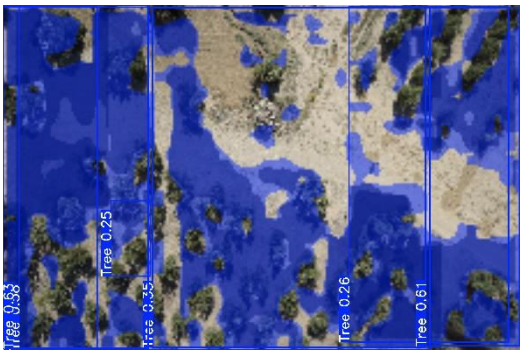
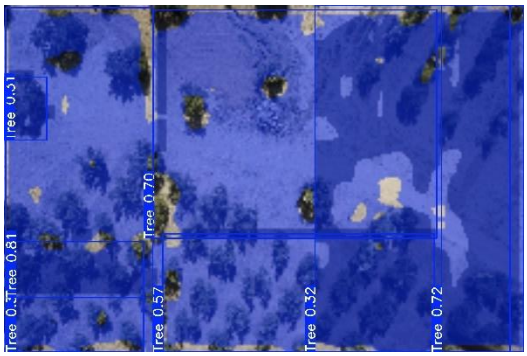
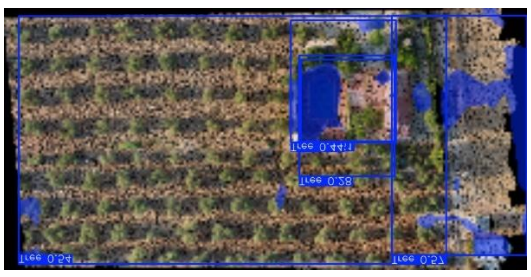
400 Epochs	**600 Epochs**
	
	

Tabla 6. Comparación del comportamiento de YOLOv9 con 400 y 600 epochs.

Por tanto, se tomará la configuración con *400* iteraciones como referencia en la posterior comparación general. Algunas imágenes más pertenecientes al conjunto de test a las que se les ha realizado predicciones se pueden ver en las *Figura 39* y *Figura 40*. Se puede observar que las detecciones de este modelo tratan de abarcar en una misma máscara varias instancias de árboles como si perteneciesen a uno solo.

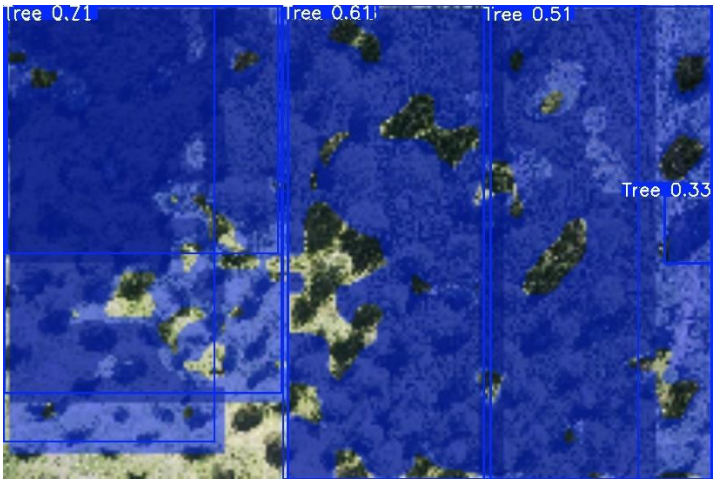


Figura 39. Segmentación de YOLOv9 sobre zona de bosque.

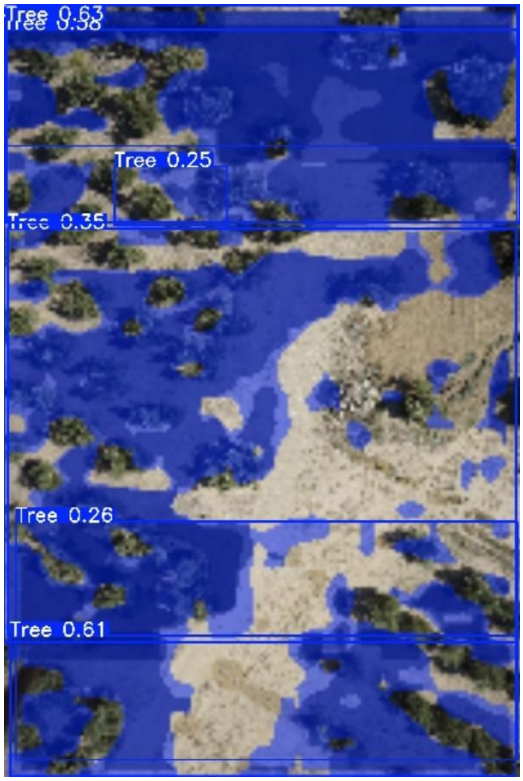


Figura 40. Segmentación de YOLOv9 sobre zona de olivar.

4.4 Resultados Detectron2

Los resultados tras varias ejecuciones de Detectron2 muestran las diferencias en la estructura interna con respecto a los modelos anteriores. En la Tabla 7 se presentan los resultados obtenidos después de *200*, *400* y *600* iteraciones. Las métricas de este modelo tienen una particularidad, puesto que presentan una diferenciación del cálculo de la precisión media según el tamaño de los objetos presentes en las imágenes. Estas métricas son **APs**, **APm** y **APl**, que se corresponden a la precisión media para objetos pequeños, precisión media para objetos medianos y la precisión media para objetos grandes respectivamente.

Epochs	***AP***	***AP50***	***AP75***	***APs***	***APm***	***APl***
200	0.780	2.583	0.330	0.032	0.347	15.023
400	0.836	2.821	0.509	0.050	0.327	16.520
600	1.092	3.037	0.266	0.078	0.600	17.693

Tabla 7. Evaluación de Segmentación de Detectron2.

Se puede observar una mejora asociada al incremento lineal de las épocas, salvo en la precisión media con un IoU mayor o igual que 0.75. Esto nos indica que el modelo se comporta mejor a mayor tiempo de entrenamiento, pero habría que tener cuidado con un posible empeoramiento de la métrica de **AP75**.

Cabe destacar una reducción sustancial en el tiempo de ejecución de dichas iteraciones, llegando a transcurrir un par de minutos para *200* iteraciones, hasta menos de diez minutos en el caso de *600* iteraciones. Es una clara diferencia con los modelos **YOLO** que llegaban a tardar un par de horas o más en los distintos entrenamientos. Por esta razón, se decidió aumentar el número de iteraciones, diez veces más que las originales, con el objetivo de comprobar si seguía manteniéndose una mejora.

Epochs	***AP***	***AP50***	***AP75***	***APs***	***APm***	***APl***
2000	0.994	3.175	0.111	0.117	0.350	17.361
4000	0.981	3.350	0.394	0.117	0.722	17.650
6000	1.138	3.169	0.208	0.198	1.371	19.028

Tabla 8. Evaluación de Segmentación de Detectron2 con más iteraciones.

En la *Tabla 8* vemos un estancamiento y un decrecimiento en las métricas en comparación con la *Tabla 7*. La media de precisión **AP** en *6000* iteraciones sí que es más alta que con *600* iteraciones, pero no es una subida sustancial.

Atendiendo a las medidas adicionales, sí que nos indican un mayor acierto en la iteración *6000* que en la *600*, detectando y segmentando árboles de distintos tamaños, tanto pequeños como medianos y grandes. Determinar qué configuración es la óptima para esta tarea de segmentación podría pasar por un análisis cualitativo y visual de cómo se comportan estas configuraciones ante las imágenes de test.

Figura 41. Imagen de test segmentada por Detectron2 tras 600 epochs.

Figura 42. Imagen de test segmentada por Detectron2 tras 6000 epochs.

Comparando ambas figuras, *Figura 41* y *Figura 42*, se nota unas ligeras diferencias en qué configuración de iteraciones es la mejor. Quizás un punto clave entre ellas es fijarse en el número de instancias de árboles que son

capaces de detectar, y no identificar otros objetos que no son de la clase objetivo como la piscina que se puede ver, o incluso las sombras. Por lo cual, podríamos determinar que la configuración de *6000 epochs* es la que mejor realiza la tarea, puesto que identifica bastante masa arbórea individual sin agrupar las zonas de sombra de estos árboles a su máscara a pesar de identificar la piscina como un árbol. En resumen, la inversión de unos minutos más merece la pena en este caso puesto que puede aportar más beneficios en la instanciación.

Aunque en algunas imágenes no ha sido capaz de realizar buenas predicciones sobre las de test. En *Figura 43* y *Figura 44* se puede ver que sí ha identificado árboles individuales y algunos sobre masas más compactas. Pero se puede apreciar un problema de *sobresegmentación* y el dibujo de unas máscaras parecidas a las de **YOLOv9** abarcando grandes zonas de árboles.

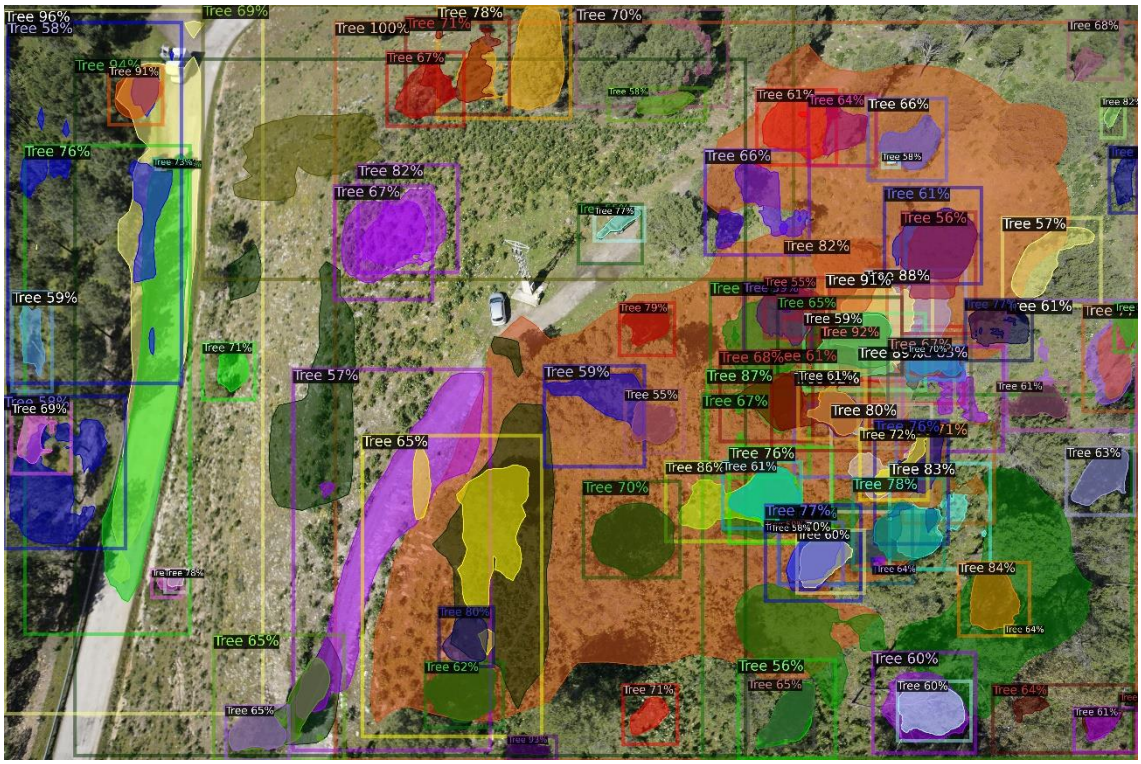


Figura 43. Predicción de segmentación de Detectron2 en zona de bosque.

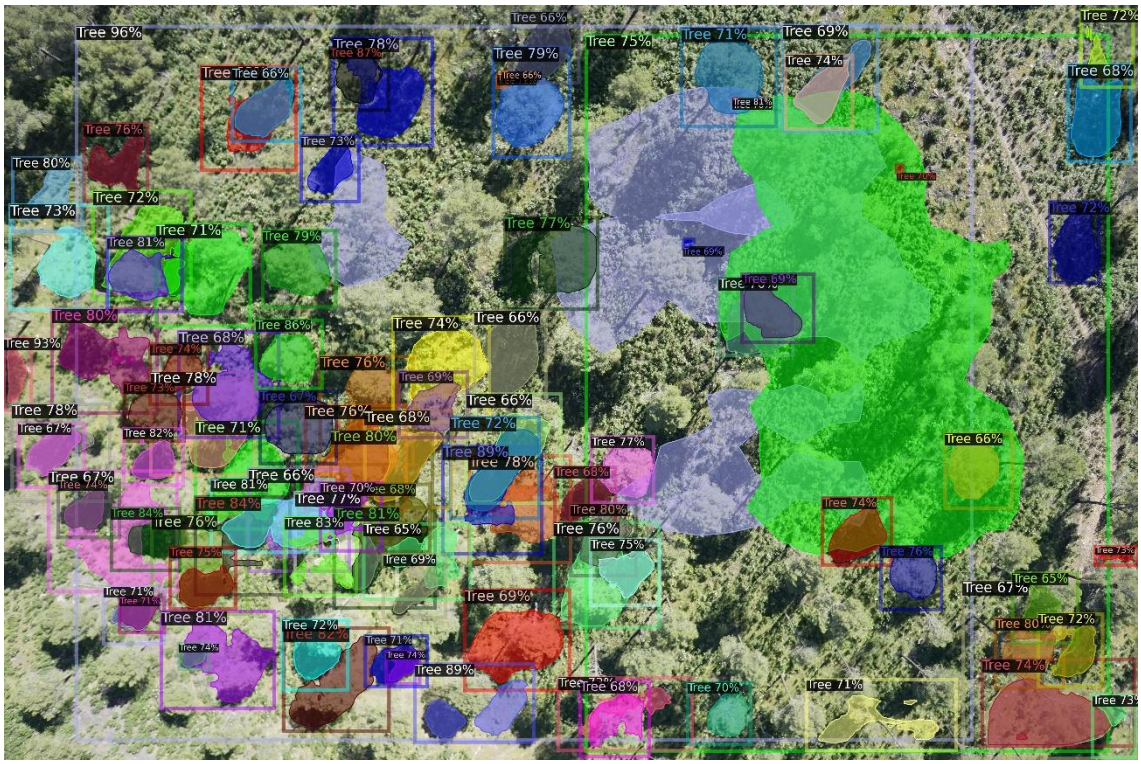


Figura 44. Otra predicción de segmentación de Detectron2 en zona de bosque.

4.5 Comparación General

Finalmente, tras la obtención de todos los resultados, y quedándonos con las mejores configuraciones de cada modelo, se realiza una comparación entre ellos y de tal forma que nos permita averiguar cuál es el óptimo para nuestra tarea de segmentación de árboles.

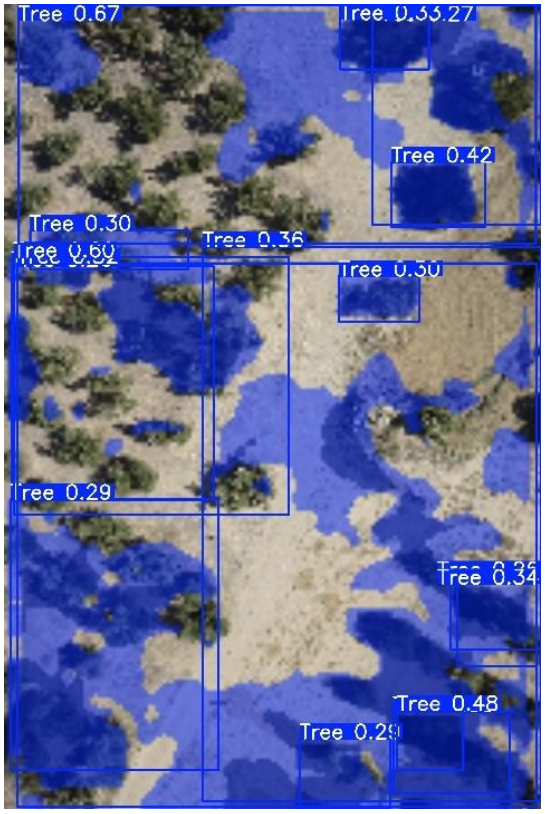
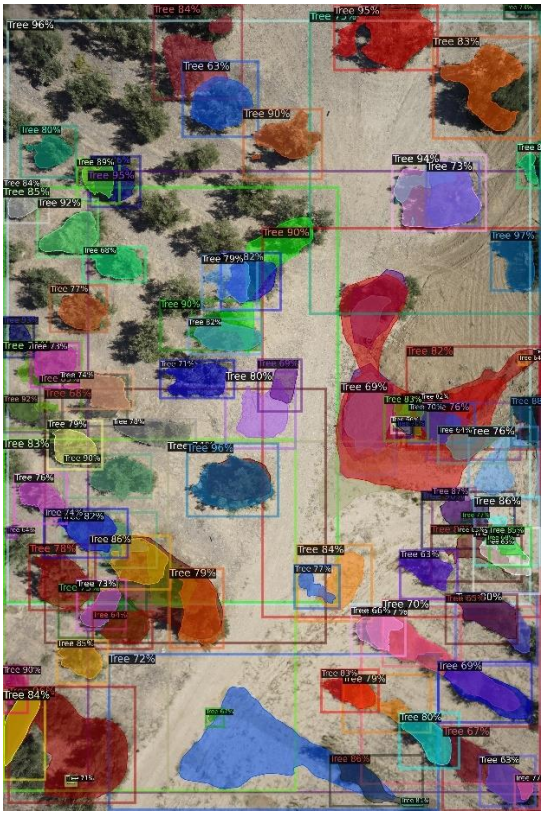
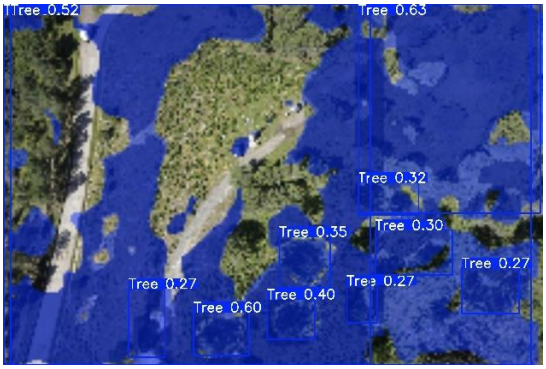
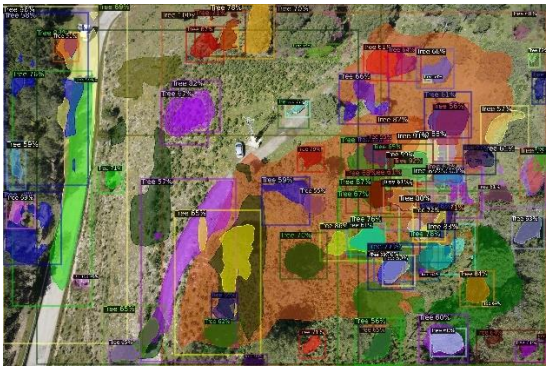
Para ello, se presenta una comparativa con dos métricas, **AP** y **AP50**. Esto se debe a que los modelos nos ofrecen distintas medidas tras las ejecuciones y como se habrá podido ver en las secciones anteriores, las métricas entre los modelos YOLO y Detectron2 no comparten ciertas medidas de evaluación. Así que, se han escogido **AP** y **AP50** porque son las métricas comunes entre ambos.

Epochs	***AP***	***AP50***
YOLOv8	0.153	0.070
YOLOv9	0.109	0.058
Detectron2	1.138	3.169

Tabla 9. Comparación general de los modelos YOLOv8, YOLOv9 y Detectron2.

En *Tabla 9* se puede ver un claro ganador en todas las métricas, **Detectron2**. En la media general de la precisión, ha superado por *siete* veces el resultado de **YOLOv8**, y en *setenta* la media de la precisión con un IoU mayor o igual a *0.5*.

Aun así, en una inspección visual, **Detectron2** sólo se nota que es superior en todas las imágenes y no hay mucha competición con **YOLOv8** en cuál de ellos es más capaz de segmentar instancias.

YOLOv8	Detectron2
	
	

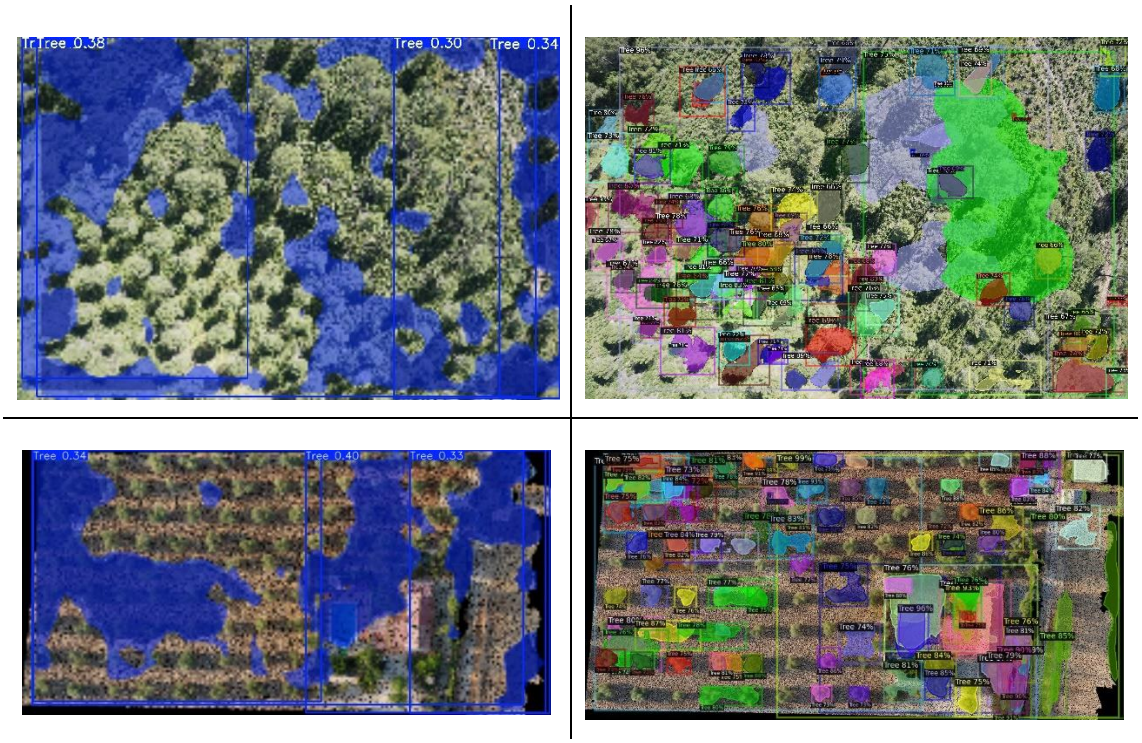


Tabla 10. Tabla comparativa de predicciones de segmentación entre los modelos YOLOv8 y Detectron2.

En la *Tabla 10* se muestran diferentes predicciones de segmentación para algunas imágenes ya expuestas a lo largo del análisis. Se pueden ver los errores y los aciertos de cada modelo. En la primera fila se evidencia la diferencia de segmentación por parte de **Detectron2** delimitando mejor los contornos de los árboles con respecto a **YOLOv8**. En las siguientes imágenes se demuestra la superioridad de **Detectron2** sobre **YOLOv8** en la detección. En cuanto a cantidad de árboles individuales, es un claro ganador, y también en no generar máscaras grandes englobando muchos de esos árboles.

Atendiendo también a la velocidad y el rendimiento, **Detectron2** es el escogido como el más adecuado en esta tarea. Su rápido entrenamiento y sus métricas mucho más altas que los modelos **YOLO** nos asegura que puede segmentar decentemente zonas arbóreas. No obstante, se debe recalcar el enfoque inicial de entrenar estos modelos con solo imágenes sintéticas y utilizar unas imágenes de diferentes zonas reales para comprobar si estos modelos eran capaces de adaptarse. Y así ha sido, con mejor o peor resultado, los modelos han logrado identificar árboles nunca vistos.

5. Conclusiones

5.1 Estudio Realizado

El objetivo de este trabajo, encontrar técnicas de segmentación de imágenes para detectar masas arbóreas comenzó con la investigación en el aprendizaje profundo y los diferentes tipos redes neuronales. Pasando desde las redes convolucionales, sus arquitecturas y capas, hasta llegar a las redes convolucionales basadas en regiones y su extensión orientada a la segmentación, **Mask R-CNN**. Después se procedió a buscar diferentes implementaciones y los estados del arte para esta red, encontrando los modelos **Mask R-CNN** de Tensorflow y Pytorch, **YOLO** y **Detectron**.

Una vez elegidos los métodos, continuamos con el preprocesamiento de las imágenes. Aplicando diferentes técnicas de normalización, redimensión a las imágenes y sus máscaras. Adecuando también las anotaciones del conjunto de imágenes para el posterior entrenamiento de los modelos. Tras esto se detallaron los problemas encontrados durante las implementaciones y la forma en la que se resolvieron llegando por fin a los primeros resultados. Asimismo, se debe destacar la complejidad que aportaban las imágenes del conjunto utilizado, por la poca precisión de las anotaciones, sus distintos formatos y unas copas de árboles casi indiferenciables unas de otras a ojo humano.

Por último, se realizaron varias configuraciones de los métodos finalmente escogidos, para su evaluación y comparación general a partir de las métricas de la sección *4.1 Métricas de Evaluación*. Con este estudio, finalmente se determinó que **Detectron2** es el modelo con mejor rendimiento y eficiencia que es capaz de llevar a cabo la tarea propuesta en este trabajo.

5.2 Conclusiones

En conclusión, podemos terminar el proyecto afirmando que se han cumplido los propósitos esperados. Se han estudiado las diferentes técnicas de segmentación y se ha desarrollado una implementación para cada una de ellas. Si bien es cierto que no se han obtenido los resultados esperados y unas detecciones perfectas, debido a la dificultad al trabajar con las imágenes. A pesar de ello, se ha podido identificar masas arbóreas y con ello podemos expresar cierta satisfacción con el trabajo realizado.

5.3 Propuestas Futuras

Brevemente se aportarán algunas mejoras para una posible continuación de este proyecto.

5.3.1 Conjunto de Datos

Como primera propuesta, se podría utilizar un conjunto de datos de entrenamiento distinto o al menos ampliado con mejores fotografías y anotaciones. Creo que es un factor clave para conseguir un buen entrenamiento de los modelos y, en consecuencia, un mejor rendimiento en la tarea de segmentación que se ha propuesto a lo largo del estudio.

Además, se podrían aplicar diversas técnicas de aumento de los datos aparte de la rotación usada en el trabajo. Pueden ir desde el volteo, escalado, recorte de cada imagen, hasta la implementación de redes adversarias generativas mencionadas en la sección *2.6 Generación de Imágenes Sintéticas*.

5.3.2 Segmentación Semi-Automática

Se puede dar el caso que los nuevos datos introducidos no traigan consigo sus anotaciones correspondientes y haya que anotarlas de forma manual. Esto puede significar horas de trabajo que podrían ser cruciales en el desarrollo de un proyecto. Por ello, a este problema, se plantea una segmentación *semi-automática* utilizando una conjunción de dos modelos **SAM** (*Segment Anything | Meta AI*, s. f.) y **Grounding DINO** (*IDEA-Research/GroundingDINO*, 2023/2024). Un ejemplo sencillo para la implementación de este enfoque personalizado a nuestro caso puede verse en (*Zero-Shot Image Annotation with Grounding DINO and SAM - A Notebook Tutorial*, 2023).

5.3.3 Ajuste de Parámetros

Otra propuesta para el proyecto sería el uso de técnicas de optimización como *Grid Search, Random Search* o la *optimización Bayesiana* en el ajuste de los hiper-parámetros de los modelos (*Optimización bayesiana de hiperparámetros*, s. f.). De tal forma que se encuentren aquellos que mejoren su rendimiento y su posterior funcionamiento en las tareas de segmentación.

5.3.4 Transformers para la Visión por Computador

Por último, se podría probar a introducir en este trabajo nuevas técnicas recientes en el ámbito de la visión por computador como son los **transformers**. Son una arquitectura neuronal mucho más simple, están basados en mecanismos de atención, y prescinden por completo de la recurrencia y la convolución (Vaswani et al., 2017). Sería muy interesante ver qué resultados se obtendrían con este enfoque.

Unos ejemplos de Transformers orientados a la visión por computador y a la segmentación de imágenes son *Mask2Former* (Cheng et al., 2022), *SegFormer* (Xie et al., 2021) y *SwinFormer* (Liu et al., 2021).

5.4 Valoración Personal

Al inicio del trabajo, tenía pequeñas nociones sobre el ámbito del aprendizaje profundo y sus aplicaciones en la visión por computador. Gracias a asignaturas cursadas previamente en el grado, me fueron de gran ayuda para enfocar este proyecto y empezar la búsqueda de técnicas adecuadas para el problema planteado.

Fueron innumerables horas de investigación, de prueba y error, que me sumergieron en sentimientos negativos y desesperación al no ver avances. Pero gracias al apoyo de mi entorno y mi perseverancia, pude encontrar la manera de obtener resultados fructíferos.

De igual forma, la comprensión y aplicación de modelos de aprendizaje profundo, junto con el preprocesamiento del conjunto de datos para su implementación, han sido un proceso enriquecedor para mi formación. Además, no se ha tratado solo de resolver un problema técnico, sino también de aprender cómo estructurar un proyecto de tal envergadura.

Todo el conocimiento que me ha aportado el desarrollo de este TFG supone para mí un avance en el ámbito profesional y también en el personal, puesto que me brinda una visión de las distintas aplicaciones de estas técnicas en diversas disciplinas que pueden ofrecer beneficios incalculables a la sociedad.

6. Apéndice

6.1 Manual de Usuario

En este apartado se explica el funcionamiento de los diferentes módulos de la solución entregada como código fuente.

Es importante aclarar que son archivos o cuadernos de Jupyter, entonces se recomienda ejecutarlos con **Jupyter** o **Google Colab**. Además, para las ejecuciones de aquellos módulos de las implementaciones de los modelos se aconseja tener acceso a recursos de computación gráfica ya sea de manera local al utilizar Jupyter, o configurando el entorno con GPU o TPU en el caso de Google Colab como se muestra en *Figura 45*.

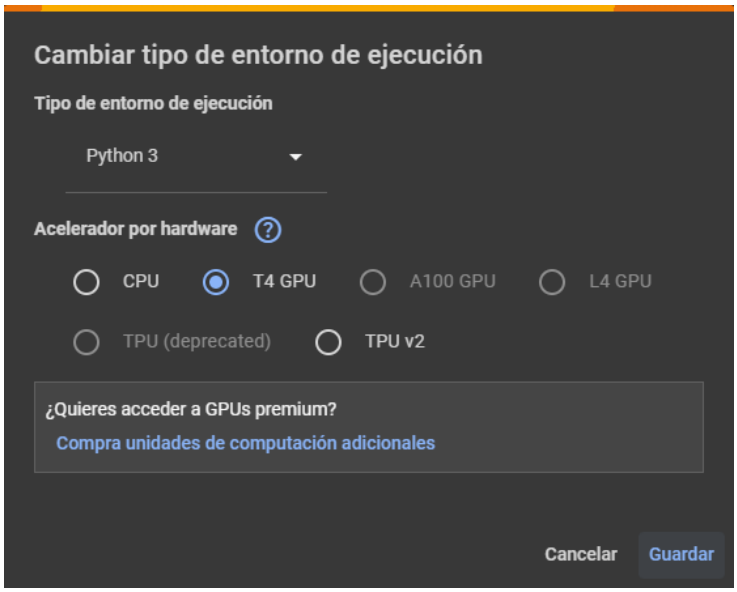


Figura 45. Cambio de entorno de ejecución en Google Colab.

En última instancia, se sugiere tener un directorio de trabajo dedicado a la utilización de los archivos, con las rutas del conjunto de datos bien establecidas y accesibles.

6.1.1 Requerimientos y Dependencias

Primeramente, estos archivos están enfocados para la ejecución en entornos de Google Colab. Por lo que se debe puntualizar ciertas instrucciones como en *Código 10* vienen precedidas por un signo de exclamación que solo es necesario para la ejecución en ese tipo de entornos. En el caso de un entorno local basta con quitar la exclamación y ejecutar en una terminal de comandos.

```
!pip install ultralytics
```

Código 10. Ejemplo de instalación de una biblioteca en Google Colab.

Otra ventaja de Google Colab, es que los entornos tienen la mayor parte de las dependencias ya instaladas. Por lo que hay que tener en cuenta si se quiere utilizar un entorno local. Para ello, se recomienda usar Anaconda, el gestor de paquetes mencionado en *1.5.5 Anaconda Navigator*.

Con esta herramienta, se facilita la creación de un nuevo entorno dedicado a la ejecución de este proyecto. La versión de **Python** debe ser igual o más reciente a la *3.9*. Si se van a utilizar recursos de aceleración gráfica, será necesario tener **CUDA Toolkits** instalada. La versión de **CUDA** deberá corresponderse con la tarjeta gráfica de su equipo, y por tanto influirá en las versiones de algunas dependencias como **Torch**.

El resto de los paquetes podrán instalarse con la interfaz gráfica de *Anaconda Navigator* o elaborando un archivo requirements.txt con las versiones de todas las dependencias e instalarlas con un comando como *Código 11*.

```
pip3 install -r requirements.txt
```

Código 11. Comando para la instalación de dependencias contenidas en un archivo .txt.

El contenido del archivo debe ser similar al mostrado en *Código 12*, pero puede dar el caso de que algunas dependencias necesiten ser versiones más recientes para garantizar la compatibilidad.

```
numpy>=1.25.2

pycocotools>=2.0.2

Pillow>=8.4.0

matplotlib>=3.7.1

torch>=1.8.0

torchvision>=0.9.0

pandas>=1.1.4

opencv-python==4.6.0

pyyaml>=5.3.1

scipy>=1.4.1
```

Código 12. Ejemplo de archivo requerimientos.txt.

En cuanto a los requerimientos hardware, se puede establecer la recomendación a partir de las características del equipo utilizado para el experimento mencionadas en *3.3 Equipo Usado para la Experimentación*. Sin embargo, se aconseja que las características sean al menos superiores en la **RAM** y la **GPU** dedicada con más de *8 gigas* cada una. Debido a que son los factores claves para un rendimiento eficiente y capaz de llevar a cabo el entrenamiento.

6.1.2 Preprocesamiento e Implementación Red Convolucional

En los cuadernos *RedConvolucionalTF_ConjuntoReal.ipynb* y *RedConvolucionalTF_ConjuntoReal.ipynb* contienen la parte del pre-procesamiento de imágenes, lectura de los archivos JSON propios del conjunto de datos, y su transformación en máscaras binarias. Justo después, se presenta la implementación de la red convolucional explicada en la sección *3.2.2 Red Neuronal Convolucional*, utilizando las imágenes ya procesadas junto con sus máscaras. Ambos archivos realizan la misma tarea, con la diferencia de que están especializados para el conjunto de imágenes reales o sintéticas, así que el usuario podría utilizar uno u otro indistintamente si su conjunto de datos no tiene una subdivisión.

Con un seguimiento y ejecución secuencial de los pasos, el usuario debe de ser capaz de preprocesar su conjunto de imágenes, modificar los parámetros a su caso particular, y finalmente entrenar la red convolucional con sus datos. Aun así, en los archivos el usuario puede encontrar comentarios aclaratorios en ciertos puntos del código.

6.1.3 Utilidades

Los archivos *DividirDataset.ipynb*, *RotacionImagenes.ipynb* y *RenombrarFotosDirectorio.ipynb*, tienen el objetivo de aportar diferentes utilidades que pueda necesitar el usuario durante el preprocesamiento de su conjunto de datos. Lo que ofrecen son una solución automática de tareas repetitivas como la división del conjunto de datos en las proporciones deseadas, la rotación de las imágenes y las máscaras binarias, y por último el renombre de archivos si lo consideran oportuno.

6.1.4 Conversión de Anotaciones a COCO JSON y YOLO

Los archivos encargados para las conversiones de las anotaciones explicados en *3.1 Preprocesamiento del Conjunto de Datos* son los cuadernos *ConversionMascaraBinaria-COCOJSON.ipynb* y *ConversionCOCOJSON-Yolov8.ipynb*. Estos scripts solo serán necesarios en el caso de que las anotaciones del conjunto de datos a utilizar no tengan los formatos requeridos por Detectron y YOLO.

El usuario deberá seguir secuencialmente los pasos modificando únicamente las rutas y directorios del dataset, tanto las de entrada como las de salida.

6.1.5 Implementación modelo YOLO

Para la implementación de los modelos **YOLOv8** y **YOLOv9**, se pueden seguir las instrucciones en los cuadernos *YOLOv8Training.ipynb* y *YOLOv9Training.ipynb*. Ahí se puede seguir paso por paso la configuración, entrenamiento y la posterior evaluación e inferencia del modelo. Aunque YOLO permite al usuario realizar el entrenamiento y la inferencia con recursos de la CPU, es recomendable hacer uso de GPU para acelerar el entrenamiento.

También es importante que se siga la estructura de directorios específica para estos modelos YOLO.

- DatasetYOLO/
  - train/
    - images/
    - labels/
  - valid/
    - images/
    - labels/
  - results/ # esta carpeta aparecerá una vez entrenado el modelo
    - 200_epochs-/
  - data.yaml

6.1.6 Implementación modelo Detectron2

Al igual que en los archivos de implementación de **YOLO**, para entrenar el modelo **Detectron2** se incluye el archivo *Detectron2Training.ipynb*. Indica todos los pasos a seguir de forma ordenada, con la configuración e instalación previa del modelo, las importaciones de las dependencias, la modificación de los hiperparámetros, el entrenamiento y finalmente la inferencia sobre imágenes de validación o test. Además, se incluyen comentarios aclaratorios en cada paso con información de qué se va a hacer y posibles modificaciones a gusto del usuario.

7. Referencias

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jozefowicz, R., Jia, Y., Kaiser, L., Kudlur, M., … Zheng, X. (2015). *TensorFlow, Large-scale machine learning on heterogeneous systems* [C++]. https://doi.org/10.5281/zenodo.4724125

AlexNet. (2024). En *Wikipedia*. https://en.wikipedia.org/w/index.php?title=AlexNet&oldid=1228721653

Anaconda (Python distribution). (2024). En *Wikipedia*. https://en.wikipedia.org/w/index.php?title=Anaconda_(Python_distribution)&oldid=1229055294

anandmeg. (2023, octubre 31). *¿Qué es el IDE de Visual Studio?* https://learn.microsoft.com/es-es/visualstudio/get-started/visual-studio-ide?view=vs-2022

Arcidiacono, C. S. (2018). *An empirical study on synthetic image generation techniques for object detectors*. https://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-235502

Average Precision—Testing with Kolena. (s. f.). Recuperado 25 de junio de 2024, de https://docs.kolena.com/metrics/average-precision/

Batallas, A. L., Bermeo Paucar, J., Paredes Quevedo, J., & Torres Ordoñez, H. (2020). Una revisión de las métricas aplicadas en el procesamiento de imágenes. *RECIMUNDO: Revista Científica de la Investigación y el Conocimiento*, *4*(3), 267-273.

Bharati, P., & Pramanik, A. (2020). Deep Learning Techniques—R-CNN to Mask R-CNN: A Survey. En A. K. Das, J. Nayak, B. Naik, S. K. Pati, & D. Pelusi (Eds.), *Computational Intelligence in Pattern Recognition* (pp. 657-668). Springer. https://doi.org/10.1007/978-981-13-9042-5_56

Chai, J., Zeng, H., Li, A., & Ngai, E. W. T. (2021). Deep learning in computer vision: A critical review of emerging techniques and application scenarios. *Machine Learning with Applications*, *6*, 100134. https://doi.org/10.1016/j.mlwa.2021.100134

Cheng, B., Misra, I., Schwing, A. G., Kirillov, A., & Girdhar, R. (2022). *Masked-attention Mask Transformer for Universal Image Segmentation* (arXiv:2112.01527). arXiv. https://doi.org/10.48550/arXiv.2112.01527

COCO - Common Objects in Context. (s. f.). Recuperado 22 de junio de 2024, de https://cocodataset.org/#format-data

December 17, P. on & 2019. (s. f.). *Difference Between CMOS & CCD and Why CMOS Sensors Are Preferred for Machine Vision Cameras*. Phase1 Vision. Recuperado 20 de junio de 2024, de https://www.phase1vision.com/blog/difference-between-cmos-and-ccd

Deep Learning with Python, Second Edition. (s. f.). Recuperado 23 de junio de 2024, de https://learning.oreilly.com/library/view/deep-learning-with/9781617296864/

Facebookresearch/detectron2. (2024). [Python]. Meta Research. https://github.com/facebookresearch/detectron2 (Obra original publicada en 2019)

Firoze, A., Wingren, C., Yeh, R. A., Benes, B., & Aliaga, D. (2023). *Tree Instance Segmentation With Temporal Contour Graph*. 2193-2202. https://openaccess.thecvf.com/content/CVPR2023/html/Firoze_Tree_Instance_Segmentation_With_Temporal_Contour_Graph_CVPR_2023_paper.html

Gad, A. (2024). *Ahmedfgad/Mask-RCNN-TF2* [Python]. https://github.com/ahmedfgad/Mask-RCNN-TF2 (Obra original publicada en 2020)

Getting Started with R-CNN, Fast R-CNN, and Faster R-CNN - MATLAB & Simulink—MathWorks España. (s. f.). Recuperado 15 de junio de 2024, de https://es.mathworks.com/help/vision/ug/getting-started-with-r-cnn-fast-r-cnn-and-faster-r-cnn.html

Google Colab. (s. f.). Recuperado 15 de junio de 2024, de https://research.google.com/colaboratory/faq.html

He, K., Gkioxari, G., Dollar, P., & Girshick, R. (2017). *Mask R-CNN*. 2961-2969. https://openaccess.thecvf.com/content_iccv_2017/html/He_Mask_R-CNN_ICCV_2017_paper.html

He, K., Zhang, X., Ren, S., & Sun, J. (2016). *Deep Residual Learning for Image Recognition*. 770-778. https://openaccess.thecvf.com/content_cvpr_2016/html/He_Deep_Residual_Learning_CVPR_2016_paper.html

Hernan_olmi.pdf. (s. f.). Recuperado 15 de junio de 2024, de https://revistaingenieriaaldia.ucentral.cl/rev_10/hernan_olmi.pdf

Hinton, G. E., Srivastava, N., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. R. (2012a). *Improving neural networks by preventing co-adaptation of feature detectors* (arXiv:1207.0580). arXiv. http://arxiv.org/abs/1207.0580

Hinton, G. E., Srivastava, N., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. R. (2012b). *Improving neural networks by preventing co-adaptation of feature detectors* (arXiv:1207.0580). arXiv. http://arxiv.org/abs/1207.0580

Hinton, G. E., Srivastava, N., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. R. (2012c). *Improving neural networks by preventing co-adaptation of feature detectors* (arXiv:1207.0580). arXiv. https://doi.org/10.48550/arXiv.1207.0580

IDEA-Research/GroundingDINO. (2024). [Python]. IDEA-Research. https://github.com/IDEA-Research/GroundingDINO (Obra original publicada en 2023)

Islam, J., & Zhang, Y. (2020). GAN-based synthetic brain PET image generation. *Brain Informatics*, *7*(1), 3. https://doi.org/10.1186/s40708-020-00104-2

Izco, F. (s. f.). *Base de datos corporativa de personas*. Recuperado 25 de junio de 2024, de https://bookdown.org/f_izco/BDC-POC/metricas.html

Jocher, G., Chaurasia, A., & Qiu, J. (2023). *Ultralytics YOLO* (8.0.0) [Python]. https://github.com/ultralytics/ultralytics (Obra original publicada en 2022)

Keras | TensorFlow Core. (s. f.). TensorFlow. Recuperado 15 de junio de 2024, de https://www.tensorflow.org/guide/keras?hl=es-419

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). ImageNet Classification with Deep Convolutional Neural Networks. *Advances in Neural Information Processing Systems*, *25*. https://proceedings.neurips.cc/paper_files/paper/2012/hash/c399862d3b9d6b76c8436e924a68c45b-Abstract.html

Lecun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, *86*(11), 2278-2324. https://doi.org/10.1109/5.726791

Lin, T.-Y., Dollár, P., Girshick, R., He, K., Hariharan, B., & Belongie, S. (2017). *Feature Pyramid Networks for Object Detection* (arXiv:1612.03144; Versión 2). arXiv. https://doi.org/10.48550/arXiv.1612.03144

Liu, Z., Lin, Y., Cao, Y., Hu, H., Wei, Y., Zhang, Z., Lin, S., & Guo, B. (2021). *Swin Transformer: Hierarchical Vision Transformer using Shifted Windows* (arXiv:2103.14030). arXiv. https://doi.org/10.48550/arXiv.2103.14030

Luo, H., Xiong, C., Fang, W., Love, P. E. D., Zhang, B., & Ouyang, X. (2018). Convolutional neural networks: Computer vision-based workforce activity assessment in construction. *Automation in Construction*, *94*, 282-289. https://doi.org/10.1016/j.autcon.2018.06.007

Manual de usuario ADA. (s. f.). Google Docs. Recuperado 15 de junio de 2024, de https://docs.google.com/document/u/1/d/19UEHJD-NT0bahERsPFOR_r3XBFFLikuNM2bvAhYdluE/edit?usp=sharing&usp=embed_facebook

Mask R-CNN — Torchvision main documentation. (s. f.). Recuperado 23 de junio de 2024, de https://pytorch.org/vision/main/models/mask_rcnn.html

Matterport/Mask_RCNN. (2024). [Python]. Matterport, Inc. https://github.com/matterport/Mask_RCNN (Obra original publicada en 2017)

Mean Average Precision (mAP) Explained: Everything You Need to Know. (s. f.). Recuperado 25 de junio de 2024, de https://www.v7labs.com/blog/mean-average-precision, https://www.v7labs.com/blog/mean-average-precision

Mokhtar, I., Hameed Sultan, M. T., Shahar, F., Nayak, S., Łukaszewicz, A., Nowakowski, M., Giernacki, W., & Holovatyy, A. (2023). *Image Processing Systems for Unmanned Aerial Vehicle: State-of-the-art*. https://doi.org/10.20944/preprints202308.1855.v1

O'Mahony, N., Campbell, S., Carvalho, A., Harapanahalli, S., Hernandez, G. V., Krpalkova, L., Riordan, D., & Walsh, J. (2020). Deep Learning vs. Traditional Computer Vision. En K. Arai & S. Kapoor (Eds.), *Advances in Computer Vision* (pp. 128-144). Springer International Publishing. https://doi.org/10.1007/978-3-030-17795-9_10

Optimización bayesiana de hiperparámetros. (s. f.). Recuperado 28 de junio de 2024, de https://cienciadedatos.net/documentos/62_optimizacion_bayesiana_hiperparametros#introducci%C3%B3n

Palubicki, W., Horel, K., Longay, S., Runions, A., Lane, B., Měch, R., & Prusinkiewicz, P. (2009). Self-organizing tree models for image synthesis. *ACM Transactions on Graphics*, *28*(3), 1-10. https://doi.org/10.1145/1531326.1531364

Ponce, J. (2021, julio 10). Conoce qué son las funciones de activación y cómo puedes crear tu función de activación usando Python, R y Tensorflow. *Jahaziel Ponce*. https://jahazielponce.com/funciones-de-activacion-y-como-puedes-crear-la-tuya-usando-python-r-y-tensorflow/

Ponti, M. A., Ribeiro, L. S. F., Nazare, T. S., Bui, T., & Collomosse, J. (2017). Everything You Wanted to Know about Deep Learning for Computer Vision but Were Afraid to Ask. *2017 30th SIBGRAPI Conference on Graphics, Patterns and Images Tutorials (SIBGRAPI-T)*, 17-41. https://doi.org/10.1109/SIBGRAPI-T.2017.12

¿Qué es Anaconda? -Escuela Internacional de Posgrados. (2022, febrero 4). https://eiposgrados.com/blog-python/que-es-anaconda/

Rainio, O., Teuho, J., & Klén, R. (2024). Evaluation metrics and statistical tests for machine learning. *Scientific Reports*, *14*(1), 6086. https://doi.org/10.1038/s41598-024-56706-x

RecFaces. (2021, mayo 24). *¿Qué es la visión por computadora? Sistemas de visión artificial y aplicaciones: ejemplos de soluciones*. RecFaces. https://recfaces.com/articles/vision-computador-soluciones

Redmon, J., Divvala, S., Girshick, R., & Farhadi, A. (2016). You Only Look Once: Unified, Real-Time Object Detection. *2016 IEEE Conference on Computer Vision*

and Pattern Recognition (CVPR), 779-788. https://doi.org/10.1109/CVPR.2016.91

Ren, S., He, K., Girshick, R., & Sun, J. (2017). Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *39*(6), 1137-1149. https://doi.org/10.1109/TPAMI.2016.2577031

Safonova, A., Guirado, E., Maglinets, Y., Alcaraz-Segura, D., & Tabik, S. (2021). Olive Tree Biovolume from UAV Multi-Resolution Image Segmentation with Mask R-CNN. *Sensors*, *21*(5), Article 5. https://doi.org/10.3390/s21051617

Segment Anything | Meta AI. (s. f.). Recuperado 27 de junio de 2024, de https://segment-anything.com/

Simonyan, K., & Zisserman, A. (2015). *Very Deep Convolutional Networks for Large-Scale Image Recognition* (arXiv:1409.1556). arXiv. https://doi.org/10.48550/arXiv.1409.1556

SVRAI. (s. f.). Recuperado 15 de junio de 2024, de https://www.cs.us.es/~fsancho/Cursos/SVRAI2122/Metaheuristicas2.md.html

Terven, J., & Cordova-Esparza, D. (2023). A Comprehensive Review of YOLO Architectures in Computer Vision: From YOLOv1 to YOLOv8 and YOLO-NAS. *Machine Learning and Knowledge Extraction*, *5*(4), 1680-1716. https://doi.org/10.3390/make5040083

Torch (machine learning). (2024). En *Wikipedia*. https://en.wikipedia.org/w/index.php?title=Torch_(machine_learning)&oldid=1228870223

Torres-Sánchez, J., Peña, J. M., de Castro, A. I., & López-Granados, F. (2014). Multi-temporal mapping of the vegetation fraction in early-season wheat fields using

images from UAV. *Computers and Electronics in Agriculture*, *103*, 104-113. https://doi.org/10.1016/j.compag.2014.02.009

Tremblay, J., Prakash, A., Acuna, D., Brophy, M., Jampani, V., Anil, C., To, T., Cameracci, E., Boochoon, S., & Birchfield, S. (2018). *Training Deep Networks With Synthetic Data: Bridging the Reality Gap by Domain Randomization*. 969-977. https://openaccess.thecvf.com/content_cvpr_2018_workshops/w14/html/Tremblay_Training_Deep_Networks_CVPR_2018_paper.html

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł. ukasz, & Polosukhin, I. (2017). Attention is All you Need. *Advances in Neural Information Processing Systems*, *30*. https://proceedings.neurips.cc/paper_files/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html

Visión por computadora—Libro online de IAAR. (s. f.). Recuperado 15 de junio de 2024, de https://iaarbook.github.io/vision-por-computadora/

Visual Geometry Group—University of Oxford. (s. f.). Recuperado 23 de junio de 2024, de https://www.robots.ox.ac.uk/~vgg/data/pets/

Voulodimos, A., Doulamis, N., Doulamis, A., & Protopapadakis, E. (2018). Deep Learning for Computer Vision: A Brief Review. *Computational Intelligence and Neuroscience*, *2018*(1), 7068349. https://doi.org/10.1155/2018/7068349

What is PyTorch? (s. f.). NVIDIA Data Science Glossary. Recuperado 15 de junio de 2024, de https://www.nvidia.com/en-us/glossary/pytorch/

Xie, E., Wang, W., Yu, Z., Anandkumar, A., Alvarez, J. M., & Luo, P. (2021). *SegFormer: Simple and Efficient Design for Semantic Segmentation with*

Transformers (arXiv:2105.15203). arXiv. https://doi.org/10.48550/arXiv.2105.15203

YOLO Performance Metrics—Ultralytics YOLO Docs. (s. f.). Recuperado 25 de junio de 2024, de https://docs.ultralytics.com/guides/yolo-performance-metrics/#class-wise-metrics

Zero-Shot Image Annotation with Grounding DINO and SAM - A Notebook Tutorial. (2023, abril 21). Roboflow Blog. https://blog.roboflow.com/enhance-image-annotation-with-grounding-dino-and-sam/