Universidad de Jaén

Escuela Politécnica Superior de Jaén

Métodos de aprendizaje profundo para la identificación de objetos en imágenes por satélite

Autor: Miguel Ángel López Reyes

Grado: Ingeniería Informática

Directores: Antonio Jesús Rivera Rivas y María Dolores Pérez Godoy
Departamento del director: Informática

Fecha: 09/09/2024

UNIVERSIDAD DE JAÉN

D./Dª Antonio Jesús Rivera Rivas y D./Dª María Dolores Pérez Godoy, tutor(es) del Trabajo Fin de Grado titulado: **Métodos de aprendizaje profundo para la identificación de objetos en imágenes por satélite**, que presenta Miguel Ángel López Reyes, autoriza(n) su presentación para defensa y evaluación en la Escuela Politécnica Superior de Jaén.

Jaén,

El estudiante

Los tutores

LOPEZ REYES MIGUEL ANGEL - 26528694B

Firmado digitalmente por LOPEZ REYES MIGUEL ANGEL - 26528694B
Fecha: 2024.09.09 18:15:27 +02'00'

Antonio Jesús Rivera Rivas

Fecha: 2024.09.09 17:10:26 +02'00'

PÉREZ GODOY MARÍA DOLORES

Fecha: 2024.09.09 17:16:46 +02'00'

Miguel Ángel López Reyes

Antonio Jesús Rivera Rivas

María Dolores Pérez Godoy

Agradecimientos

En primer lugar, quisiera expresar mi más sincero agradecimiento a mis tutores del Trabajo Fin de Grado, **Antonio Jesús Rivera Rivas** y **María Dolores Pérez Godoy**, por estar siempre atentos a los correos, la rapidez con la que han atendido mis consultas y por darme la oportunidad de llevar a cabo este trabajo. Su dedicación y apoyo han sido fundamentales para la realización de este proyecto.

Asimismo, me gustaría agradecer a mis compañeros de carrera, **Juan Llavero Company** e **Ismael Galán Machuca**, por su constante apoyo durante estos años. En especial, a **Juan**, por su amistad y compañía en los momentos más difíciles.

A mis padres, por estar siempre pendientes de mí, brindándome su apoyo incondicional a lo largo de todo este proceso.

Finalmente, agradezco la beca Ícaro al **CETEDEX** y a la **Universidad de Jaén** por la excelente formación que he recibido durante mi carrera.

Miguel Ángel López Reyes
09/09/2024

Tabla de contenidos

Lista de figuras

Lista de listados de código

Capítulo 1

Introducción

1.1. Contexto del problema y motivación

La detección de objetos es una técnica de visión por computador que se usa para localizar instancias de objetos en imágenes o vídeos [53]. Como resultado de la identificación de objetos, los algoritmos que se encargan de procesar la imagen, devuelven una serie de coordenadas de los objetos situados en la misma. Pero, estos modelos no solo localizan las coordenadas de los objetos en imagen, sino que además son capaces de clasificar los objetos diferenciándolos entre varias clases que el modelo ha aprendido, siempre, por supuesto, con cierta probabilidad de fallo [22].

Hoy en día, estos modelos tienen mucha importancia debido a la cantidad de ventajas que aportan a la hora de realizar ciertos trabajos.En varios de estos casos, se ha transformado en un factor crucial, impactando significativamente en áreas socioeconómicas, biofísicas, de monitoreo ambiental, entre otras.[8]. Podemos encontrarlo en muchos sitios, desde coches que disponen de detección de señales, pasando por aplicaciones móviles como Google Lens o Photomath e incluso en el reconocimiento facial en las calles de China [6].

Aunque hay muchos ámbitos en los que se puede usar las técnicas de detección de objetos como en la conducción autónoma o los sistemas de vigilancia, en este caso nos centraremos únicamente en la detección de objetos en imágenes por satélite mediante aprendizaje profundo.

El uso de técnicas de detección de objetos **ha adquirido una importancia crucial** en diversas aplicaciones, especialmente en la detección por satélite. Estas tecnologías permiten identificar y monitorear elementos clave en escenarios socioeconómicos y

ambientales, con especial relevancia en el **ámbito militar**, donde la vigilancia y el reconocimiento de objetivos juegan un papel esencial.

Este Trabajo de Fin de Grado se enmarca en una **línea de investigación propuesta por el CETEDEX**. Este es el Centro Tecnológico de Desarrollo y Experimentación, una institución dependiente del ministerio de defensa ubicada en Jaén, especializada en la investigación, desarrollo y experimentación de tecnologías avanzadas, especialmente en áreas relacionadas con la defensa y la seguridad, que busca impulsar proyectos de I+D que buscan mejorar las capacidades tecnológicas y operativas de las fuerzas de defensa y otros organismos de seguridad.

El aprendizaje profundo, una rama avanzada de la inteligencia artificial, ha mostrado resultados prometedores en el campo de la visión por computador. Gracias a su capacidad para procesar grandes cantidades de datos y aprender patrones complejos, los algoritmos basados en aprendizaje profundo han demostrado ser altamente eficaces en la detección de objetos.

1.2. Objetivos

1.2.1. Objetivo principal

El objetivo principal de este Trabajo de Fin de Grado es realizar una revisión bibliográfica en el ámbito de la detección de objetos para después implementar y evaluar modelos de detección de objetos en imágenes para el ámbito de imágenes satelitales utilizando técnicas de aprendizaje profundo. El modelo deberá ser capaz de identificar y clasificar diversos objetos presentes en las imágenes, demostrando una alta precisión y eficiencia en la tarea propuesta.

1.2.2. Objetivos específicos

Para alcanzar el objetivo principal, se establecen los siguientes objetivos específicos:

- **Explorar los conceptos clave de la ciencia de datos en general y del aprendizaje profundo en particular:** Este objetivo tiene como finalidad proporcionar una base sólida en ciencia de datos, abordando temas esenciales como la adquisición, procesamiento y análisis de datos. Se prestará especial atención a los principios fundamentales del aprendizaje profundo, incluyendo redes neuronales. Este estudio teórico es fundamental para comprender las técnicas que se aplicarán en la detección de objetos.

- **Investigar métodos de aprendizaje profundo enfocados en la identificación de objetos en imágenes:** Se estudiarán en detalle las principales técnicas de aprendizaje profundo para la detección de objetos en imágenes, como las redes neuronales convolucionales (CNN), YOLO (You Only Look Once), y Faster R-CNN, entre otros.

- **Analizar conjuntos de datos específicos para la identificación de objetos en imágenes satelitales:** En este objetivo, se examinarán diferentes conjuntos de datos disponibles para la detección de objetos en imágenes satelitales. La evaluación incluirá aspectos como la resolución de las imágenes, la calidad de los datos, y la diversidad de los objetos presentes. Este análisis permitirá seleccionar los conjuntos de datos más apropiados para entrenar y validar los modelos de aprendizaje profundo.

- **Adaptar y aplicar métodos de aprendizaje profundo para la identificación de objetos en imágenes satelitales:** Este objetivo se centrará en la adaptación de las técnicas estudiadas para que funcionen de manera efectiva en el contexto de imágenes satelitales. Se realizarán ajustes en las arquitecturas de redes neuronales, en los hiperparámetros, y se implementarán técnicas de preprocesamiento especializadas para este tipo de imágenes. Posteriormente, se procederá a ejecutar estos modelos, verificando su desempeño en la tarea propuesta.
- **Evaluar, comparar y analizar los resultados obtenidos:** Finalmente, se llevará a cabo un análisis de los resultados obtenidos tras la aplicación de los métodos de aprendizaje profundo. Se compararán los diferentes enfoques, evaluando su precisión, eficiencia y robustez, con el objetivo de identificar la mejor estrategia para la detección de objetos en imágenes satelitales.

1.3. Estructura de la Memoria

La memoria se estructura en varios capítulos que abordan de manera sistemática y detallada el desarrollo del proyecto. A continuación, se describe brevemente el contenido de cada capítulo:

1.3.1. Capítulo 1: Introducción

Este capítulo establece el contexto del problema y la motivación detrás del trabajo. Se presentan los objetivos generales y específicos del proyecto, y se proporciona una lista de acrónimos utilizados en la memoria.

1.3.2. Capítulo 2: Antecedentes

En este capítulo se realiza una revisión exhaustiva de la literatura relacionada con la detección de objetos y el aprendizaje profundo. Se cubren temas fundamentales como la inteligencia artificial, las redes neuronales (en particular las convolucionales) y se introducen los modelos de detección de objetos, incluyendo tanto métodos tradicionales como basados en aprendizaje profundo.

1.3.3. Capítulo 3: Materiales y Métodos

Este capítulo detalla los datos y las herramientas utilizadas para llevar a cabo el proyecto. Se describen los conjuntos de datos satelitales empleados, las técnicas de detección de objetos específicas para imágenes satelitales, así como el entorno de desarrollo utilizado, tanto en términos de hardware como de software.

1.3.4. Capítulo 4: Experimentación y Resultados

Aquí se presentan las experimentaciones realizadas con los modelos de detección de objetos seleccionados. Se incluyen descripciones detalladas sobre la preparación de los entornos de prueba, la preparación de los conjuntos de datos, y los resultados obtenidos, utilizando métricas como mAP50 y mAP50-95 para evaluar el rendimiento de los modelos.

1.3.5. Capítulo 5: Conclusiones

El capítulo final sintetiza el proyecto, realizando una comparación entre los modelos utilizados, destacando la importancia del contexto en la detección de objetos, y señalando las aplicaciones prácticas del trabajo. Además, se discuten los desafíos y oportunidades para futuras investigaciones.

1.3.6. Bibliografía

Finalmente, se proporciona una lista completa de las referencias bibliográficas utilizadas a lo largo de la memoria, siguiendo el formato apropiado según las normativas académicas.

1.4. Lista de Acrónimos

Se presenta a continuación la lista de acrónimos:

- **IA:** Inteligencia Artificial
- **ANN:** Artificial Neural Networks (Redes Neuronales Artificiales)
- **CNN:** Convolutional Neural Networks (Redes Neuronales Convolucionales)
- **RNN:** Recurrent Neural Networks (Redes Neuronales Recurrentes)
- **YOLO:** You Only Look Once
- **SSD:** Single Shot Multibox Detection
- **R-CNN:** Region-based Convolutional Neural Networks
- **mAP:** mean Average Precision
- **GPU:** Graphics Processing Unit
- **TPU:** Tensor Processing Unit
- **API:** Application Programming Interface
- **COCO:** Common Objects in Context
- **JSON:** JavaScript Object Notation
- **CSV:** Comma-Separated Values
- **HBB:** Horizontal Bounding Box
- **OBB:** Oriented Bounding Box
- **RoI:** Region of Interest
- **VOC:** Visual Object Classes
- **XML:** Extensible Markup Language
- **ILSVRC:** ImageNet Large Scale Visual Recognition Challenge
- **PoLSAR:** Polarimetric Synthetic Aperture Radar
- **LiDAR:** Light Detection and Ranging

- **AWS:** Amazon Web Services
- **GCP:** Google Cloud Platform
- **EC2:** Elastic Compute Cloud
- **NLP:** Natural Language Processing
- **CETEDEX:** Centro Tecnológico de Desarrollo y Experimentación

Capítulo 2

Antecedentes

En el campo de la detección de objetos en imágenes satelitales, los avances en inteligencia artificial (IA) han desempeñado un papel crucial. La IA, en particular a través de técnicas como el aprendizaje profundo, ha permitido el análisis y procesamiento de grandes volúmenes de datos visuales con una precisión sin precedentes.

Entre estas técnicas, las redes neuronales convolucionales (CNNs) han demostrado ser especialmente efectivas para la tarea de detección de objetos, ya que están diseñadas para identificar patrones y características visuales en imágenes de alta complejidad, como las que se obtienen mediante satélites. Estas CNNs se basan en la operación de convolución, que permite extraer y analizar características específicas de las imágenes, facilitando la detección de objetos de interés en un entorno tan variado como el terrestre.

Históricamente, la detección de objetos en imágenes se realizaba mediante métodos tradicionales, que requerían un extenso preprocesamiento y la creación de modelos a partir de un conjunto limitado de características manualmente diseñadas. Con la evolución de las redes neuronales y, en particular, de las CNNs, estos métodos han sido superados por enfoques más modernos basados en aprendizaje profundo, que automatizan la extracción de características y mejoran considerablemente la precisión y eficiencia del proceso de detección.

Modelos como YOLO, SSD, y las variantes de R-CNN han revolucionado este campo, permitiendo no solo la detección más rápida y precisa de objetos en imágenes satelitales, sino también la aplicación de estas técnicas a un espectro más amplio de problemas geoespaciales.

2.1. Contexto

2.1.1. Inteligencia Artificial

La **Inteligencia Artificial** [45], también conocida por sus siglas, **IA**, es una rama de la informática que se centra en la creación de sistemas capaces de realizar tareas que, cuando son realizadas por seres humanos, requieren de inteligencia. Estas tareas incluyen desde la resolución de problemas complejos y la toma de decisiones, hasta el reconocimiento de patrones y el aprendizaje a partir de datos.

En los últimos años, la IA ha **avanzado considerablemente** gracias al aumento en la capacidad de procesamiento de los ordenadores y la disponibilidad de grandes volúmenes de datos. Estos avances han permitido el desarrollo de técnicas de aprendizaje más complejas y precisas, que se utilizan en una variedad de aplicaciones, desde el diagnóstico médico hasta los vehículos autónomos.

En la figura 2.1 se puede ver de forma más detallada el camino de la Inteligencia Artificial que a lo largo de su corta historia, ha avanzado a paso agigantados.

En sus primeros años, la investigación en Inteligencia Artificial se centró en la creación de algoritmos basados en reglas explícitas y la lógica formal [45] para simular el razonamiento humano. Estos sistemas, conocidos como IA simbólica o IA basada en reglas, demostraron ser útiles en tareas específicas, como juegos de tablero o sistemas expertos para diagnóstico médico. Sin embargo, tenían limitaciones significativas debido a su incapacidad para manejar el conocimiento no estructurado o aprender de datos no etiquetados.

Con el tiempo, la Inteligencia Artificial evolucionó hacia enfoques más flexibles y adaptativos, influenciados por conceptos del aprendizaje automático y las redes neuronales [38]. Este cambio de paradigma permitió a la IA enfrentar problemas más complejos y menos estructurados, abriendo el camino a aplicaciones en áreas como el procesamiento del lenguaje natural, la visión por computadora, y la robótica.

En los últimos años, se ha producido un avance significativo en el desarrollo de modelos de IA de propósito general [5], como **ChatGPT** de OpenAI o **Gemini** de Google DeepMind. Estos modelos, basados en arquitecturas de aprendizaje profundo, utilizan grandes redes neuronales que han sido entrenadas en vastas cantidades de datos textuales para generar respuestas coherentes y relevantes en una variedad de contextos. A diferencia de los modelos de IA especializados, que se diseñan para realizar tareas específicas, los modelos de propósito general son altamente versátiles y pue-

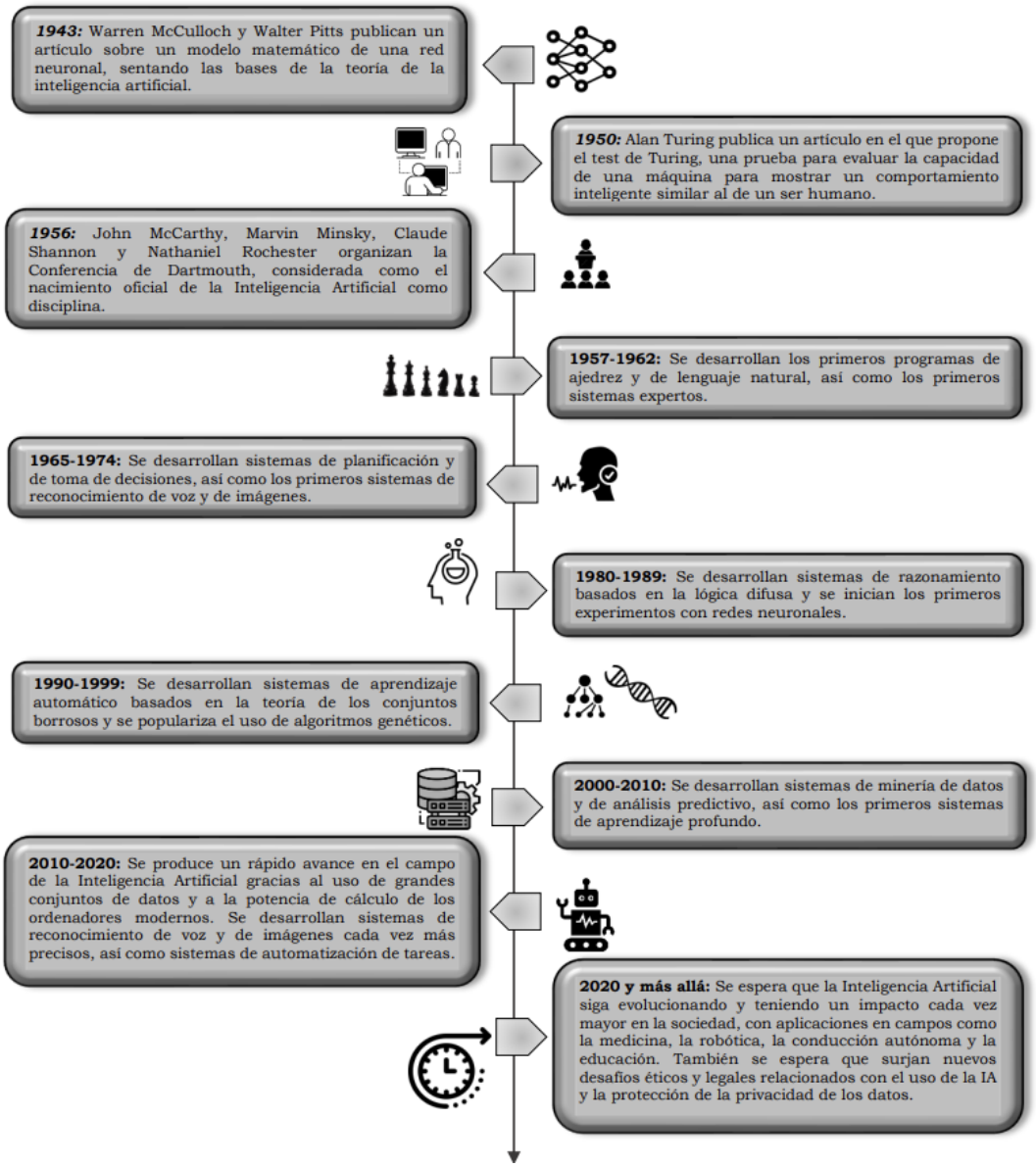

Figura 2.1: Evolución histórica de la IA (Fuente [4])

den adaptarse a múltiples aplicaciones, desde la generación de texto y la traducción automática hasta la asistencia virtual y el análisis de datos.

2.1.2. Aprendizaje Automático

El **aprendizaje automático** (o *machine learning*, en inglés) [45] es una subdisciplina de la inteligencia artificial que se centra en el desarrollo de algoritmos y modelos capaces de aprender y mejorar automáticamente a partir de la experiencia y los datos, sin ser programados de manera explícita para cada tarea [18].

A diferencia de los enfoques tradicionales de programación, donde el comportamiento del sistema es predefinido por reglas, el aprendizaje automático permite que las máquinas descubran patrones y relaciones en los datos, adaptándose a nuevos escenarios y mejorando su rendimiento con el tiempo.

El aprendizaje automático se divide en varias categorías principales, entre las que se incluyen el aprendizaje supervisado, no supervisado y por refuerzo.

En el **aprendizaje supervisado**, los algoritmos se entrenan utilizando un conjunto de datos etiquetados, es decir, datos en los que las respuestas correctas ya están definidas. Esto permite que el modelo aprenda a realizar predicciones o clasificaciones basadas en ejemplos anteriores mientras que en el **aprendizaje no supervisado**, en cambio, el modelo trabaja con datos no etiquetados, intentando descubrir estructuras ocultas o patrones inherentes sin ninguna guía previa.

Por último, el **aprendizaje por refuerzo** implica que el modelo aprende a tomar decisiones a través de la interacción con un entorno, recibiendo recompensas o penalizaciones basadas en las acciones que realiza.

El aprendizaje automático ha tenido un impacto significativo en una amplia gama de aplicaciones, como el reconocimiento de voz, la visión por computadora, la personalización de contenido y la predicción de tendencias. A medida que la disponibilidad de datos y la capacidad computacional continúan creciendo, las técnicas de aprendizaje automático se están volviendo cada vez más sofisticadas y esenciales para abordar problemas complejos en diversos campos.

2.1.3. Redes Neuronales

Las Redes Neuronales Artificiales, ANN (Artificial Neural Networks) están inspiradas en las redes neuronales biológicas del cerebro humano [38]. Están constituidas por elementos que se comportan de forma similar a la neurona biológica en sus funciones más comunes. Estos elementos están organizados de una forma parecida a la que presenta el cerebro humano.

Consisten en capas de nodos (o neuronas) conectados entre sí, donde cada conexión tiene un peso ajustable. Estas redes son capaces de aprender representaciones complejas de datos gracias a su capacidad para ajustar los pesos de las conexiones en función de los errores de predicción.

En una red neuronal, las neuronas o nodos de cada capa **realizan una operación**

clave conocida como la *función de activación*. Esta función toma como entrada una combinación lineal de los pesos y los valores de las entradas de la neurona, y produce una salida que se transmite a las siguientes capas.

Las funciones de activación permiten a la red modelar relaciones complejas y no lineales entre los datos. Entre las funciones más comunes se encuentran la función **sigmoide**, que convierte los valores en probabilidades entre 0 y 1, la función ***ReLU*** (Rectified Linear Unit), que es ampliamente utilizada debido a su capacidad para manejar problemas de desaparición del gradiente, y la función **tangente hiperbólica**, que normaliza las salidas entre -1 y 1.

Estas funciones permiten que las redes neuronales aprendan patrones complejos y mejoren su capacidad predictiva. En la figura 2.2 se puede apreciar la diferencia entre ReLU y sigmoid

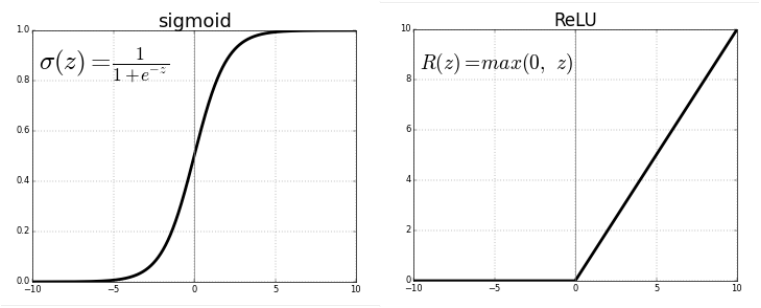

Figura 2.2: Funciones de activación ReLU y sigmoid

Estas redes neuronales se componen típicamente de una capa de entrada, una o más capas ocultas, y una capa de salida. Podemos ver una representación de las mismas en la figura 2.3. Las capas ocultas permiten a la red aprender características intermedias y complejas de los datos de entrada, mientras que la capa de salida genera la predicción final.

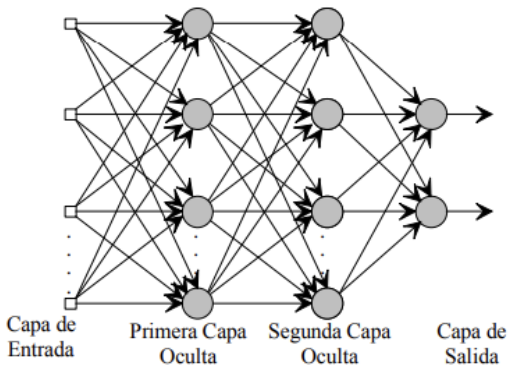

Figura 2.3: Esquema de una red neuronal (Fuente [38])

El **proceso de entrenamiento** de una red neuronal es un ejemplo de *aprendizaje automático supervisado*, en el cual el objetivo es ajustar los parámetros de la red, es decir, los pesos y los sesgos de las neuronas, para minimizar el error en las predicciones del modelo.

Este proceso comienza con un conjunto de datos etiquetados, que consiste en pares de entradas y salidas esperadas. Durante el entrenamiento, las entradas se alimentan a la red, que realiza una predicción basada en los valores iniciales de los pesos.

El error o *pérdida* se calcula comparando la predicción con la salida real a través de una función de pérdida, como el error cuadrático medio o la entropía cruzada. Para reducir este error, se utiliza un algoritmo de optimización, como el *descenso de gradiente*, que ajusta los pesos de la red en función de los gradientes de la función de pérdida con respecto a los pesos. Este proceso se repite a lo largo de múltiples iteraciones o *épocas*, con el objetivo de que la red aprenda a realizar mejores predicciones.

El ajuste de los pesos mediante el descenso de gradiente se mejora con técnicas como la retropropagación, que distribuye el error de la capa de salida hacia las capas anteriores de la red, permitiendo que cada neurona ajuste sus parámetros de forma adecuada.

A medida que avanza el entrenamiento, la red neuronal mejora su capacidad de generalización, es decir, su habilidad para realizar predicciones precisas con nuevos datos no vistos durante el entrenamiento.

Dejando a parte el proceso de entrenamiento, además de las arquitecturas básicas de redes neuronales clásicas, existen otros tipos de arquitecturas diseñadas para tareas específicas. Por ejemplo, las **Redes Neuronales Recurrentes** (RNNs) [18] se utilizan comúnmente para procesar datos secuenciales, como texto o series temporales, debido a su capacidad para mantener información a través de secuencias de entrada. Por otro lado, las **Redes Neuronales Convolucionales** (CNNs) son especialmente efectivas para tareas de visión por computadora, ya que utilizan capas convolucionales para detectar características locales en imágenes. Estas últimas serán sujeto de estudio en este trabajo y se explicará a continuación.

2.1.4. Redes Neuronales Convolucionales

Las Redes Neuronales Convolucionales (CNNs) son una clase de redes neuronales profundas que han demostrado un rendimiento notable en tareas de procesamiento de imágenes y visión por computadora [30]. La arquitectura de las CNNs está diseñada para reconocer patrones espaciales y de características jerárquicas en los datos, lo que las hace altamente eficaces para el reconocimiento de imágenes y otras tareas relacionadas.

En la figura 2.4 se puede observar la arquitectura general de una red neuronal convolucional en la que se distinguen las capas convolucionales, las capas de pooling, la capa de activación y normalización y por último las capas totalmente conectadas, a parte de la capa de salida que forma parte de estas últimas.

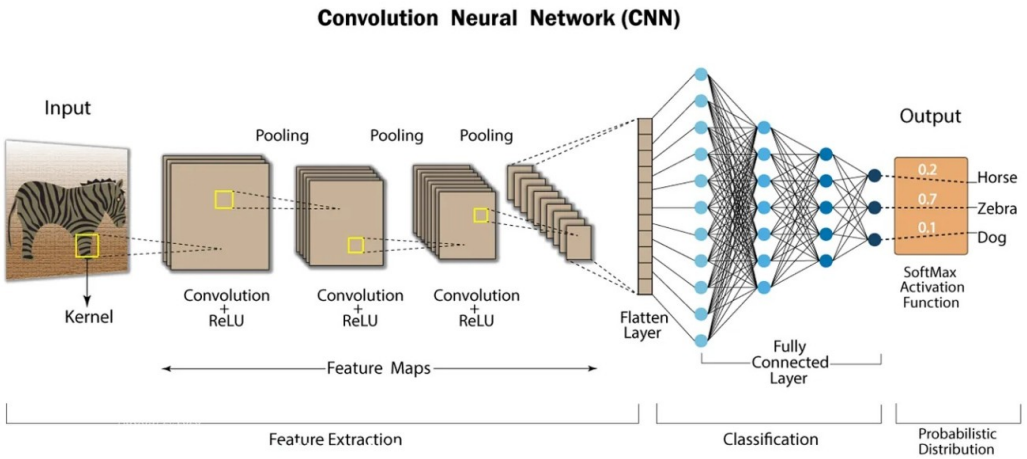

Figura 2.4: Arquitectura general de una red neuronal convolucional (Fuente [1])

2.1.4.1. Definición de Convolución

La **convolución** es una operación matemática fundamental que se utiliza extensamente en procesamiento de señales [31], análisis de imágenes y visión por computadora, especialmente en el contexto de las Redes Neuronales Convolucionales (CNNs). En términos generales, la convolución es un proceso que combina dos funciones para producir una tercera función que expresa cómo la forma de una se modifica por la otra. Esta operación permite resaltar características específicas de los datos de entrada, como bordes, texturas o patrones locales en imágenes.

Matemáticamente, la convolución de dos funciones discretas f y g se define como:

$$(f * g)(i,j) = \sum_{m=-\infty}^{\infty} \sum_{n=-\infty}^{\infty} f(m,n) \cdot g(i-m, j-n)$$

En el contexto de imágenes y CNNs, la función f representa la imagen de entrada y la función g representa el kernel o filtro que se aplica a la imagen. El resultado de la convolución es un nuevo mapa de características que resalta las propiedades detectadas por el filtro.

La **operación de convolución** se lleva a cabo mediante el desplazamiento del filtro sobre la imagen de entrada (o el mapa de características generado por capas previas) y calculando el producto escalar entre el filtro y la región de la imagen cubierta por el filtro en cada posición. Esto resulta en un mapa de características de salida que muestra la activación de ciertas características detectadas por el filtro en diferentes ubicaciones de la imagen de entrada.

De forma sencilla, la convolución en una CNN es como usar una ventana para escanear una imagen, haciendo pequeñas operaciones matemáticas en cada paso para resaltar ciertas características importantes que ayudan a la red a entender mejor la imagen.

2.1.4.2. Origen de las Redes Neuronales Convolucionales

El concepto de redes neuronales convolucionales surgió a partir de la necesidad de mejorar el reconocimiento de patrones en imágenes, inspirándose en la organización del córtex visual de los animales. El modelo LeNet [55], representado en la figura 2.5, desarrollado por Yann LeCun y sus colegas en la década de 1980, fue uno de los primeros modelos de CNN exitosos. LeNet se utilizó inicialmente para el reconocimiento de dígitos escritos a mano, logrando resultados impresionantes en tareas de clasificación de imágenes en un momento en que los modelos tradicionales tenían dificultades para manejar la variabilidad en los datos visuales.

El desarrollo de las CNNs experimentó un gran avance con la introducción de arquitecturas más profundas y complejas [30], como AlexNet, en 2012, que ganó la competencia ImageNet Large Scale Visual Recognition Challenge (ILSVRC). AlexNet demostró que una red neuronal con múltiples capas convolucionales podía superar significativamente a los modelos anteriores en precisión de clasificación de imágenes. A partir de ahí, se desarrollaron otras arquitecturas notables como VGG, ResNet e Inception, cada una de las cuales contribuyó con innovaciones adicionales, como el uso

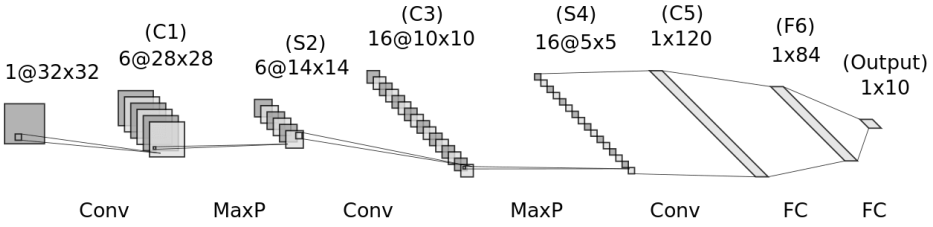

Figura 2.5: Arquitectura de LeNet (Fuente [32])

de convoluciones más pequeñas, bloques de residuales para facilitar el entrenamiento de redes más profundas, y arquitecturas de redes con múltiples caminos que permiten la modelización de patrones a diferentes escalas.

2.1.4.3. Características de las Redes Neuronales Convolucionales

Las CNNs se componen de varias capas especializadas diseñadas para extraer características de los datos de entrada de manera jerárquica. Las capas principales de una CNN incluyen:

- **Capa Convolucional:** Esta es la capa fundamental de las CNNs, donde se aplican filtros o kernels que convolucionan sobre la entrada para producir mapas de características (feature maps). Cada filtro es capaz de detectar características locales como bordes, texturas o formas básicas en la imagen, dependiendo de sus pesos que se entrenan durante el proceso de aprendizaje. La convolución ayuda a reducir el número de parámetros en comparación con una ANN totalmente conectada, lo que la hace más eficiente en términos de memoria y cálculo.
- **Capa de Pooling:** Esta capa se utiliza para reducir la dimensionalidad espacial de los mapas de características generados por la capa convolucional. El pooling más común es el *max pooling*, que toma el valor máximo en cada subregión de la entrada, ayudando a mantener las características más importantes mientras reduce la cantidad de datos y el costo computacional. Esta capa también introduce invarianza espacial, lo que significa que la red puede reconocer características independientemente de su ubicación exacta en la imagen.
- **Capas de Activación y Normalización:** Después de la convolución y el pooling, las activaciones no lineales, como ReLU (Rectified Linear Unit), se aplican para introducir no linealidad en el modelo, permitiendo a la red aprender una amplia

variedad de funciones. Además, las técnicas de normalización por lotes (*batch normalization*) ayudan a estabilizar y acelerar el proceso de entrenamiento.

- **Capas Completamente Conectadas:** Al final de una CNN, las capas completamente conectadas se utilizan para realizar la clasificación final. Estas capas toman las características extraídas por las capas anteriores y las utilizan para tomar decisiones sobre la clase de la entrada de acuerdo a los datos de entrenamiento.

2.2. Modelos de detección de objetos

Principalmente, tenemos dos maneras de dividir los modelos de detección de objetos. Por un lado tenemos la **detección de objetos basadas en métodos tradicionales** con métodos como DPM ("Deformable Parts Models") [13], HOG ("Histogram Of oriented Gradients") [7] o el método de clasificación de Cascadas de Haar [51] mientras que por otro lados tenemos los **modelos basados en aprendizaje profundo** que están basados en redes neuronales. Estas últimas son las que actualmente están teniendo más repercusión debido a su capacidad de aprendizaje, adaptación a diferentes situaciones y eficacia. Se entrará en detalle más adelante.

2.2.1. Modelos basados en métodos tradicionales

Los modelos de detección de objetos tradicionales se basan en algoritmos que deben de ser diseñados por un experto en el campo de la visión por computador. A diferencia de los modelos basados en aprendizaje profundo no hacen uso de una red neuronal. Un ejemplo de estos es el famoso **método de clasificación de Cascadas de Haar**.

El método de clasificación de Cascadas de Haar se basa en la concatenación de varios clasificadores débiles, cada uno analizando una porción diferente de una imagen o frame de vídeo. Se consideran débiles porque tienen alta probabilidad de dar falso positivo. Pero al juntar muchos clasificadores en cascada se pueden combinar para obtener resultados muy fiables.

Para crear los clasificadores se requiere de un diseño manual de características y un ajuste fino de los parámetros del clasificador aunque últimamente se ha creado software basado en redes neuronales para poder crear clasificadores de forma auto-

mática a través de datasets de imágenes positivas (imágenes en las que salgan los objetos que se buscan) y datasets de imágenes negativas [49].

Los modelos basados en métodos tradicionales incluyen una amplia variedad de algoritmos que, aunque no serán objeto de estudio en este trabajo, constituyen un tema relevante que cualquier persona interesada en la visión por computador debería explorar.

2.2.2. Modelos basados aprendizaje profundo

Dentro de los modelos basados en aprendizaje profundo podemos diferenciar entre los modelos de **detección de una etapa** o los de **detección en dos etapas**.

Las diferencias entre estos dos es la manera en la que afrontan el problema. En líneas generales, los **detectores de una etapa** usa una red neuronal convolucional (CNN) para localizar y clasificar los objetos resultando en una mayor velocidad de reconocimiento pero sacrificando la tasa de acierto mientras que por otro lado tenemos la **detección en dos etapas** que usa una red neuronal convolucional completa con características extras para poder usarlas posteriormente poder ubicar los objetos de destino entre la región candidata. Esto hace que estas tengan una capacidad de acierto mayor que las primeras, pero, reduciendo la velocidad.

Entrando en detalles, los detectores de una etapa realizan simultáneamente la localización de objetos y la clasificación de los mismos en la última capa de la red neuronal mientras que los detectores de dos etapas realizan la localización y la clasificación después de que un proponedor de regiones localice las zonas de las imágenes en las que podrían existir objetos como vemos en la Figura 2.6 [14].

En la Figura 2.7 [27], se presenta una clasificación de los algoritmos más utilizados actualmente para esta tarea.

En el siguiente apartado se estudiará cada uno de estos dos tipos de detectores y se entrará en detalle en cuales son sus características principales.

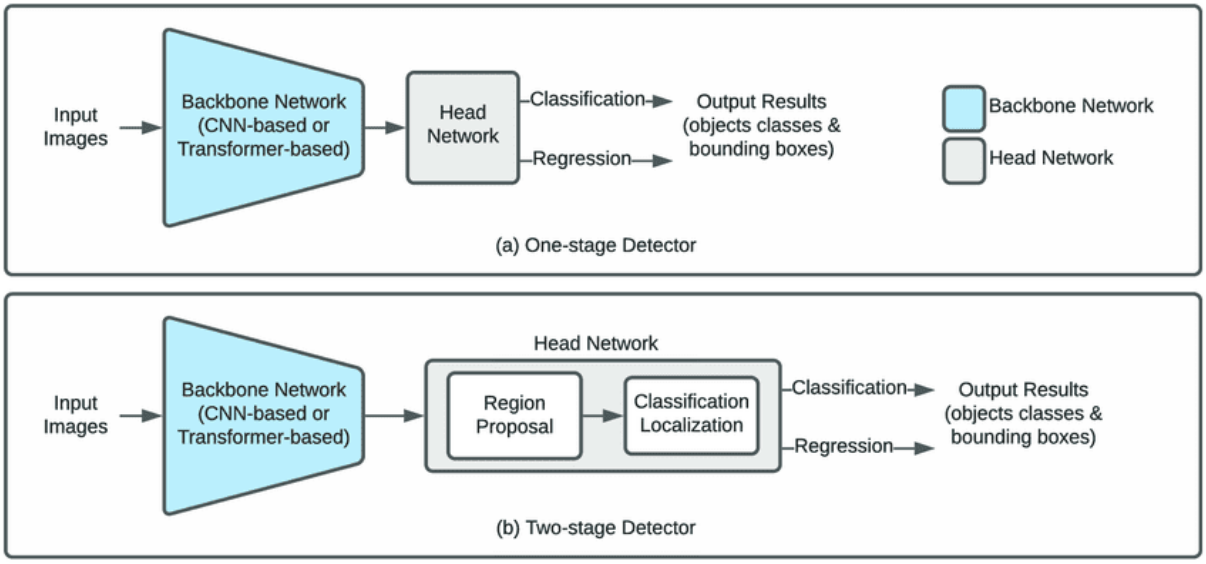

Figura 2.6: Diferenciación entre los detectores de una y dos etapas [14]

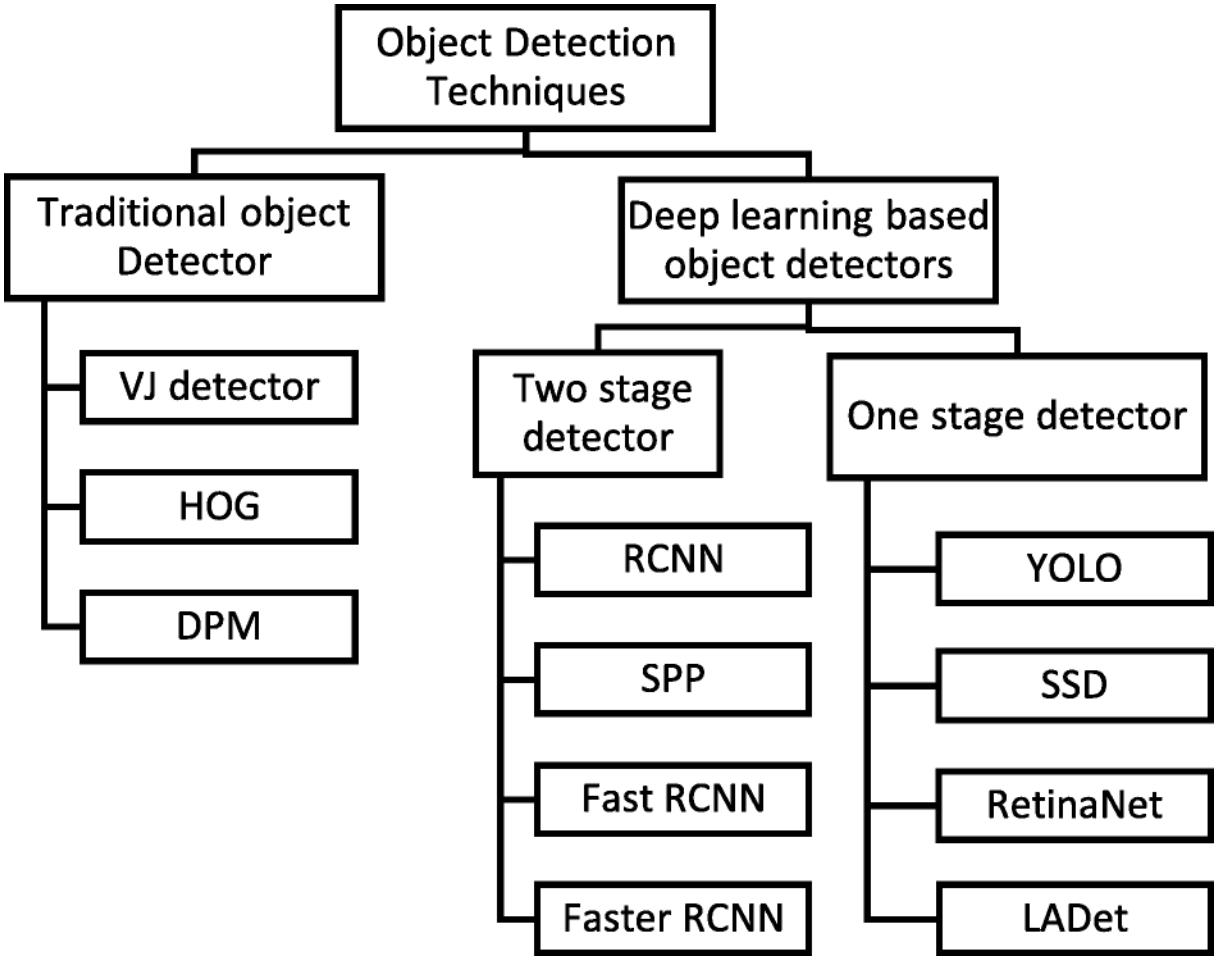

Figura 2.7: Clasificación de diversos modelos de detección de objetos [27]

2.2.3. Detectores de objetos de una etapa

En el punto anterior, hemos visto superficialmente las diferencias entre los detectores de una etapa y de dos. Un detector de una etapa requiere una pasada por la red neuronal y predice todos los recuadros delimitadores de una sola vez. Es mucho más rápido y adecuado para dispositivos con capacidad de procesamiento limitada o que requieren de una alta velocidad sacrificando algo de capacidad de acierto.

Dentro de este tipo de detectores, los más famosos son YOLO ("You Only Look Once"), SSD ("Single Shot MultiBox Detector"), RetinaNet y LADet.

De estos cuatro detectores, el más destacado es YOLO, ya que ha recibido actualizaciones continuas a lo largo del tiempo, logrando un excelente equilibrio entre eficacia y velocidad."

También veremos por encima el algoritmo SSD debido a su eficiencia en vídeo de tiempo real.

2.2.3.1. YOLO (You Only Look Once)

YOLO [33] es una arquitectura creada por Joseph Redmond capaz de detectar objetos de forma muy rápida y precisa. El hecho de estos dos últimos atributos la hacen perfecta para usarla en **detección de vídeo en tiempo real**. Consiste en una red neuronal convolucional (CNN) que predice simultáneamente múltiples cuadros delimitadores y las probabilidades de la clase objeto que delimitan dichos cuadros [44].

Esta detección es tan rápida porque como su propio nombre indica, solo tiene que analizar la imagen una sola vez.

Las beneficios de esta arquitectura son:

- La velocidad con la que se ejecuta debido a que el problema se reduce a un problema de regresión lineal, esto conlleva que no se requiera de un pipeline complejo.
- Esta arquitectura analiza la imagen completa, en vez de regiones específicas, por lo que hace que se limiten los errores a la hora de reconocer las clases de objetos.
- Aprende representaciones generalizadas de objetos, lo que hace que si se presentan un conjunto de datos nuevos, existe una mayor probabilidad de acierto.

De forma general, la imagen se divide en una rejilla de tamaño S x S de forma que, si un objeto cae en una celda de la cuadrícula, esa celda es la responsable de detectar el objeto. Por otra parte, cada celda de la cuadrilla predice N cuadros delimitadores (bounding boxes) asignando a cada uno de estos una puntuación de confianza (score). Si en dicha caja no hay objeto, el score será cercano o igual a 0 pudiendo establecer un límite para poder eliminar las cajas que no superen cierta cota. En la figura 2.8 podemos ver la división en rejillas del algoritmo.

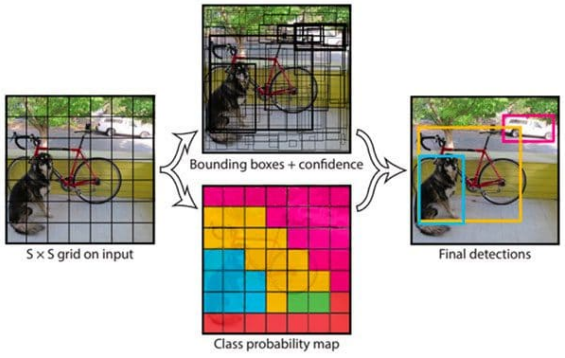

Figura 2.8: División de una imagen con YOLO calculando cierta probabilidad en cada casilla

A fecha de hoy, YOLO cuenta con ocho versiones, algunas de ellas como YOLO, YOLO9000, YOLOv2 o YOLOv3 desarrolladas por el mismo autor mientras que otras versiones están desarrolladas por académicos de la Taiwanesa Academia Sinica.

Este algoritmo es tan potente que en su versión v6 puede rendir a 1187 FPS con una probabilidad de acierto del 37,5 % [33]. La última versión lanzada de este algoritmo fue YOLOv8 lanzada en el año 2023.

2.2.3.2. SSD (Single Shot Multibox Detection)

SSD [34] presenta el primer detector de objetos basado en red profunda que no vuelve a muestrear píxeles o características para hipótesis de cuadros delimitadores. El resultado es una mejora significativa en la velocidad de la detección de alta precisión.

Las mejoras incluyen utilizar un pequeño filtro convolucional para predecir las categorías y offsets (compensaciones) de los objetos en las localizaciones de las cajas.

Estas se hicieron pensando en que los modelos hasta ese momento eran muy intensos computacionalmente hablando por lo que para la detección de objetos en tiem-

po real necesitaba de una optimización del pipeline tradicional que se estaba siguiendo tradicionalmente. Los resultados de las pruebas arrojaron que el sistema mejoraba significativamente a (59) FPS con un acierto del 74,3 % respecto a Faster R-CNN (7) FPS con mAP del 73,2 % o YOLO (45) FPS con un mAP del 63,4 % [34].

Véase que mAP es el promedio de las precisiones promedio de todas las clases de objetos presentes en el conjunto de datos.

2.2.4. Detectores de objetos de dos etapas

Como se ha comentado, los detectores de dos etapas se dividen ,como su propio nombre indica, en dos fases. La **primera fase** se encarga de proponer las regiones donde hay cierta probabilidad de poder localizar objetos y la **segunda fase** es una red neuronal convolucional que se encarga de obtener los objetos que existen dentro de las regiones propuestas por la primera fase.

Estos detectores de dos etapas han mostrado ser una opción muy fiable cuando nos interesa obtener resultados con un mAP alto, sin importar tanto la velocidad con la que se obtienen los resultados. Se suelen utilizar en campos en los que la precisión del detector sea de mucha importancia como en el caso de la medicina asistida por visión de computador.

En las siguientes secciones de este trabajo analizaremos en detalle algunos ejemplos de este tipo de detectores.

2.2.4.1. R-CNN, Fast R-CNN y Faster R-CNN

El primer detector que usó este tipo fue **R-CNN** [30]. Su autor Ross Girshick propuso un método para extraer las famosas *region proposals* a través de un algoritmo para posteriormente utilizar una red neuronal convencional para extraer las características y poder obtener los objetos de la imagen.

Este detector tenía algunos problemas como eran la lentitud del mismo, pudiendo llegar a tardar hasta 47 segundos para poder procesar una imagen de test, también se necesitaba mucho tiempo para poder entrenar la red neuronal y además la primera etapa de proposición de regiones era un algoritmo estático por lo que en esta etapa el algoritmo no aprendía nada.

El mismo autor volvió a publicar un nuevo paper en el que solventaba algunas de las desventajas que tenía el algoritmo anteriormente mencionado, llamado este nuevo detector **Fast R-CNN** [15].

El funcionamiento es similar al de R-CNN pero en vez de alimentar a la CNN con las region proposals, se alimentaba con la imagen para generar un mapa de características convolucional. Y una vez obtenido este mapa, se extraen las region proposals para envolverlas dentro de cuadros y usando una capa de agrupación de regiones de interés (RoI pooling), las redimensionamos a un tamaño fijo para que puedan ser introducidas en una capa completamente conectada.

Como podemos ver en la Figura 2.9 [15], el detector Fast R-CNN es mucho más rápido resolviendo la imagen y además tarda menos el entrenamiento del mismo.

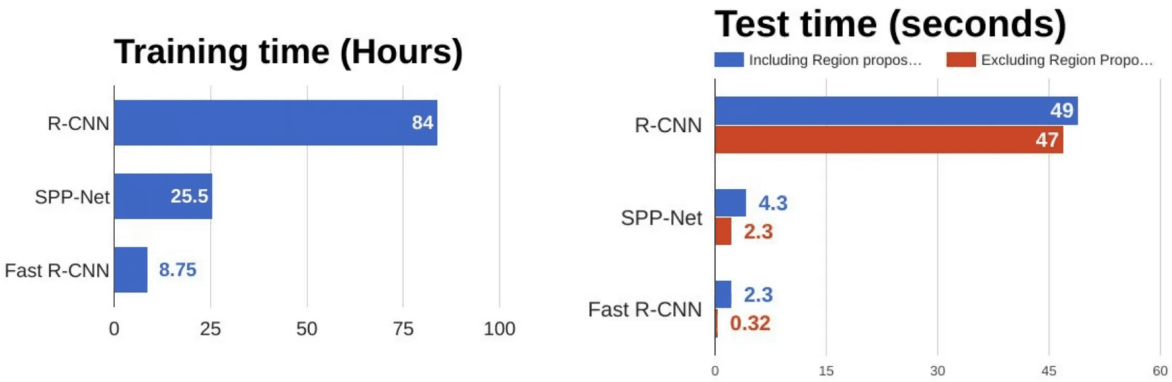

Figura 2.9: Diferencia de rendimiento entre R-CNN y Fast R-CNN [15]

La última evolución de este algoritmo fue cuando Shaoqing Ren, Kaiming He, Ross Girshick, y Jian Sun publicaron un nuevo artículo en el que introducian nuevas mejoras al detector, llamando a este último **Faster R-CNN** [15].

Hasta el momento la búsqueda de *region proposals* se había codificado mediante un algoritmo de búsqueda selectivo, este tipo de algoritmo era lento y consumía mucho tiempo. De esta forma, los autores mencionados anteriormente eliminaron este algoritmo y dejaron que la red aprendiera los region proposals.

De forma similar a Fast R-CNN tenía un mapa de características convolucional, pero en vez usar el algoritmo mencionado anteriormente una red neuronal separada era usada para predecirlas [42].

Este algoritmo mejoró tanto la velocidad de ejecución que se convirtió en un algoritmo de tiempo real como podemos ver en la Figura 2.10 [15].

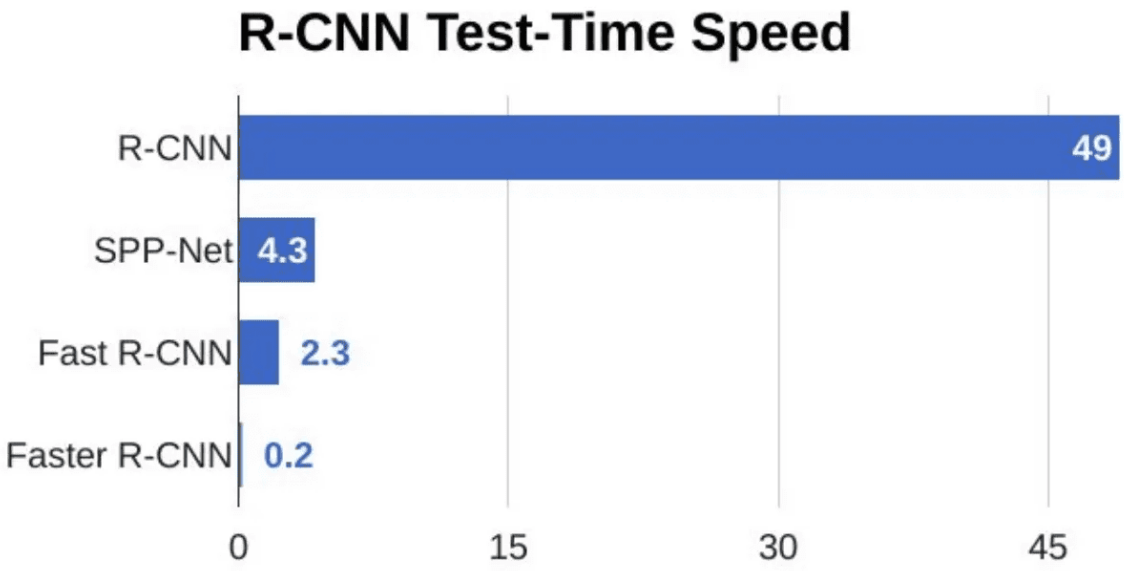

Figura 2.10: Diferencia de rendimiento entre Faster R-CNN y sus predecesoras [15]

Capítulo 3

Materiales y métodos

Anteriormente, se ha hecho alusión a la necesidad de tener conjuntos de datos extensos, flexibles y heterogéneos a la hora de entrenar modelos de detección de objetos. Por ello en este apartado se hará una investigación tanto en la búsqueda de conjunto de datos de imágenes satelitales como técnicas de implementaciones actuales que se pueden usar para la finalidad principal de este trabajo.

3.1. Conjunto de datos de imágenes satelitales

La detección de objetos se puede realizar mediante imágenes tomadas de distintas formas y con distintos dispositivos. Desde imágenes tomadas con una cámara de un smartphone hasta imágenes realizadas mediante cámaras montadas en satélites. Estas últimas son las que estudiaremos en este trabajo.

Uno de los retos que nos encontramos a la hora de entrenar un detector, es el conjunto de imágenes que necesitaremos para poder entrenar el modelo, es decir, necesitamos una conjunto con una gran cantidad de imágenes que no solo deben ser etiquetadas. No basta con solo tener las imágenes, sino que para la detección de objetos, tenemos que tener un **fichero de anotaciones** donde para cada imagen debe de estar descrita dentro de la misma los objetos que contiene junto a sus coordenadas.

Para poder anotar los objetos existentes, se pueden utilizar distintos tipos de archivos. Una de las extensiones que se utilizan es **CSV** (Comma-Separated Values) ya que nos permite de forma sencilla indicar en cada línea un objeto localizado en alguna de las imágenes que tiene nuestro conjunto de datos. Podemos ver un ejemplo a continuación.

```
id,image_id,geometry,class
1,4f833867-273e-4d73-8bc3-cb2d9ceb54ef.jpg,"[(135, 522), (245, 522), (245, 600), (135, 600), (135, 522)]",Airplane
2,4f833867-273e-4d73-8bc3-cb2d9ceb54ef.jpg,"[(1025, 284), (1125, 284), (1125, 384), (1025, 384), (1025, 284)]",Airplane
3,4f833867-273e-4d73-8bc3-cb2d9ceb54ef.jpg,"[(1058, 1503), (1130, 1503), (1130, 1568), (1058, 1568), (1058, 1503)]",Airplane
4,4f833867-273e-4d73-8bc3-cb2d9ceb54ef.jpg,"[(813, 1518), (885, 1518), (885, 1604), (813, 1604), (813, 1518)]",Airplane
5,4f833867-273e-4d73-8bc3-cb2d9ceb54ef.jpg,"[(594, 938), (657, 938), (657, 1012), (594, 1012), (594, 938)]",Airplane
6,4f833867-273e-4d73-8bc3-cb2d9ceb54ef.jpg,"[(451, 725), (524, 725), (524, 798), (451, 798), (451, 725)]",Airplane
7,4f833867-273e-4d73-8bc3-cb2d9ceb54ef.jpg,"[(1543, 1437), (1614, 1437), (1614, 1497), (1543, 1497), (1543, 1437)]",Airplane
8,4f833867-273e-4d73-8bc3-cb2d9ceb54ef.jpg,"[(1485, 1370), (1561, 1370), (1561, 1437), (1485, 1437), (1485, 1370)]",Airplane
9,4f833867-273e-4d73-8bc3-cb2d9ceb54ef.jpg,"[(1968, 1624), (2053, 1624), (2053, 1687), (1968, 1687), (1968, 1624)]",Airplane
```

Listado 3.1: Ejemplo de datos en archivo CSV

En cada línea del archivo CSV se encuentra un objeto encontrado en alguna de las imágenes. Cada coma representa una variable diferente. Podemos observar que la primera variables es el identificador del objeto, la segunda el identificador de la imagen que representa, la tercera variable es un conjunto con los vértices del objeto encontrado en la imagen y por último el nombre del objeto.

Cómo podemos ver, obtener un conjunto de datos no es algo sencillo, es más, gran parte del tiempo que se de dedica a la hora de construir un modelo, es obtener conjuntos de datos que nos permita el entrenamiento, y no, no vale cualquiera. Hay conjuntos de datos que vienen con un archivo de anotaciones preparado para alimentar a nuestro modelo de detección mientras que otros será incompatibles o necesitarán de un trabajo de adaptación para poder meterlos en nuestro modelo.

Debido a esto, una de las tareas antes de comenzar con la selección de la técnica de detección y la creación del modelo es la búsqueda de los conjuntos de datos que vamos a usar para el entrenamiento del mismo.

3.1.1. Tipos de sensores

En el mundo de la fotografía, no solo se puede tomar fotos con un sensor óptico, sino que se pueden tomar de diferentes formas y con diferentes sensores. Desde recoger imágenes que registren ondas del espectro no visibles para el ser humano hasta poder obtener imágenes que recogen la temperatura en cada punto de la imagen

simbolizándola con un color.

Lo común a la hora de tomar fotografías es tener un sensor **óptico** ya que son los más prácticos debido a que la mayoría de la gente desea realizar fotos de lo que ven y no de otras propiedades como podrían ser la temperatura o la profundidad. Pero con la evolución de la tecnología se puede encontrar sensores de múltiples tipos como veremos a continuación.

3.1.1.1. Multiespectral e Hiperespectral

Las fotos de este tipo de sensores, a parte de recoger la luz visible como los sensores fotográficos comunes, recogen más bandas del espectro.

En el caso de los sensores **multiespectrales**, recogen las bandas infrarrojas y las bandas infrarrojas de onda corta que son las que se sitúan entre la luz visible y las ondas infrarrojas. Mientras que los sensores **hiperespectrales** se encargan de dividir el espacio anterior en muchas bandas más corta haciendo que el mismo espacio de color se dividan en un número mayor de bandas por lo que la precisión del color al tener una continuidad mayor será mejor. En la Figura 3.1 se puede ver una representación de las distintas ondas que estos dos sensores son capaz de captar.

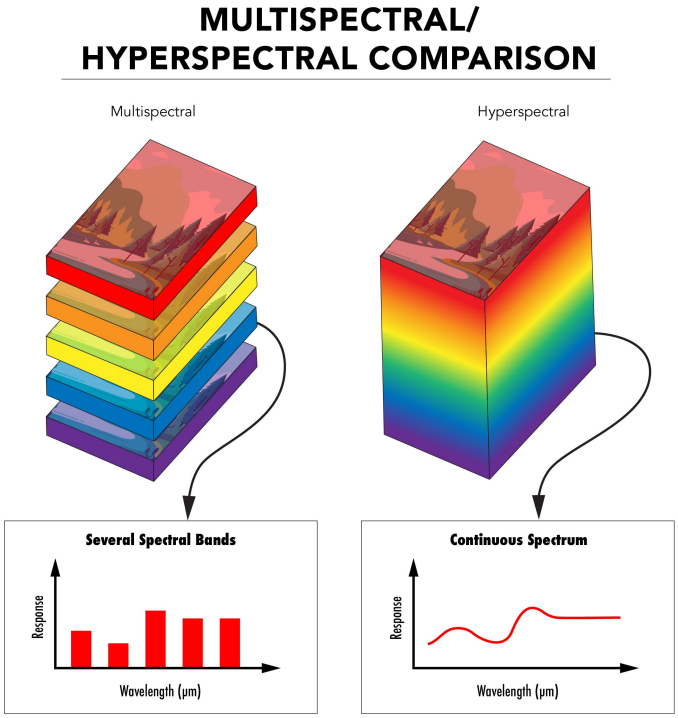

Figura 3.1: Comparación de las capas entre sensores multiespectrales e hiperespectrales [21]

3.1.1.2. Depth Map

Estos sensores se centran en recoger la información de la profundidad de los objetos con respecto a la cámara, una de las técnicas más usadas para poder obtener este tipo de imágenes es usando dos cámaras separadas horizontalmente por una cierta distancia. Gracias a esto, al igual que en la visión humana, se puede obtener con la diferencia entre las dos imágenes la profundidad del objeto.

Si el objeto se desplaza mucho entre una foto y su homóloga significa que el objeto está cerca mientras que si el objeto aparentemente no se mueve entre una foto y otra significa que el objeto se encuentra más lejos.

3.1.1.3. Thermal Map

Otro tipo bastante usados en procesos industriales, son los sensores de temperatura por imagen. Estos detectan en cada uno de los píxeles de la foto la temperatura aproximada de la superficie del objeto u objetos al que está apuntando, de esta forma, haciendo una imagen de un objeto se puede saber cuales de las zonas de la superficie tiene una mayor temperatura. Podemos ver un ejemplo de esta imagen en la Figura 3.2

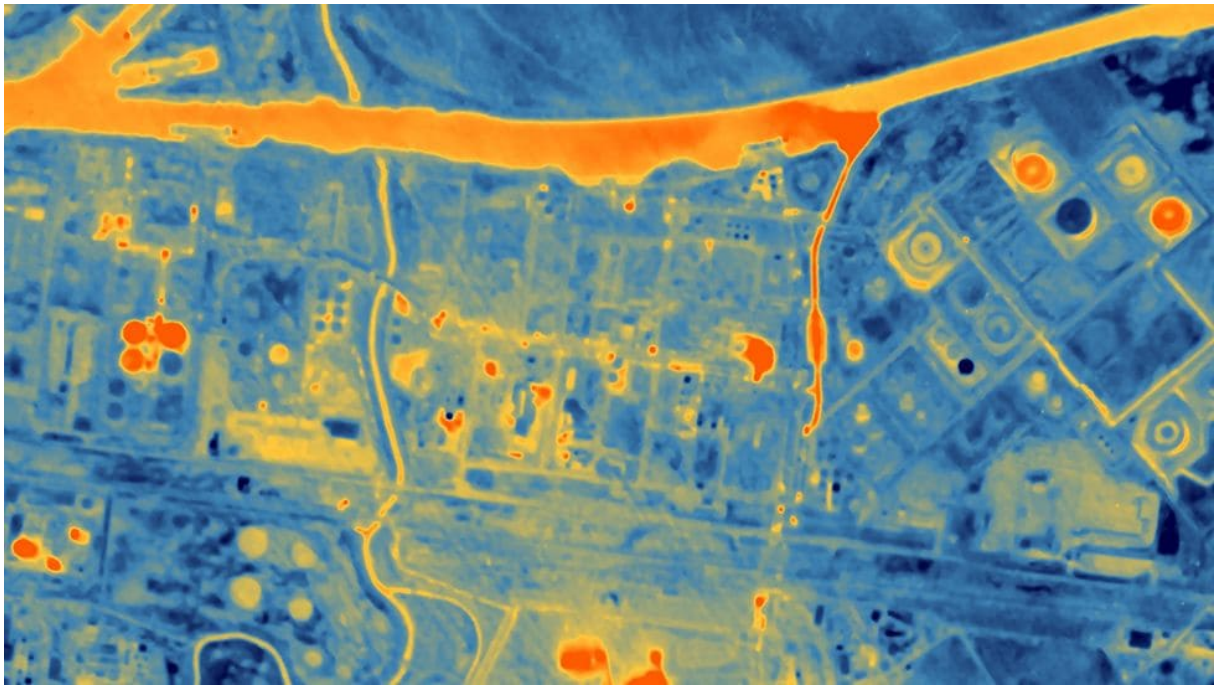

Figura 3.2: Imagen tomada con un sensor de temperatura.

3.1.2. Búsqueda de conjuntos de datos

Para realizar la búsqueda de conjuntos de datos siempre nos podemos apoyar en páginas que tengan un buscador integrado como **Google** o **Github** y en ellas intentar buscar con palabras clave los conjunto de datos, como podemos ver en la Figura 3.3.

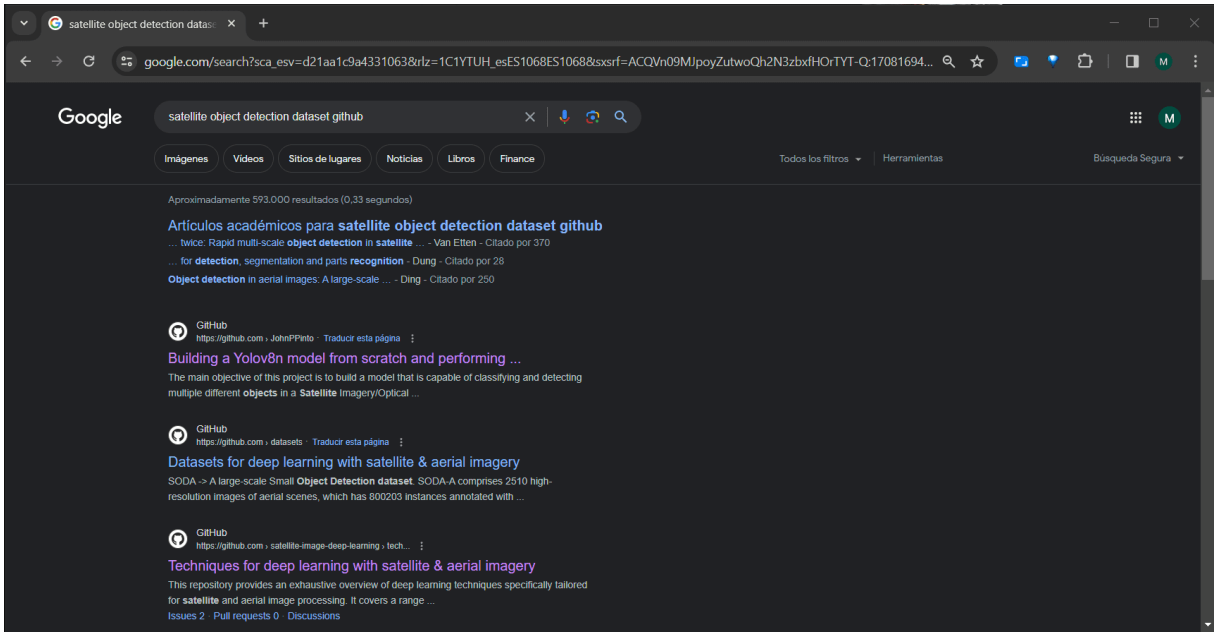

Figura 3.3: Búsqueda de conjuntos de datos en google

También podemos buscar en páginas especializadas en *machine learning* para encontrar conjuntos de datos, como Kaggle [26], o en este caso, para un tema más específico como la visión por satélite, en la página Earth Observation Database [9]. En esta última página, se hace mención a otros recursos de investigadores que se dedican a la creación de conjuntos de datos especializados.

Gracias a la expansión de internet, podemos tener acceso a múltiples fuentes de datos de donde extraer los conjuntos. Es cuestión de esfuerzo y tiempo encontrar buenos conjuntos que podamos utilizar para nuestro modelo, en los siguientes apartados se verán algunos de los conjuntos de datos que utilizaremos para el desarrollo de este proyecto.

3.1.2.1. Earth Obervation Database

En esta página [10] podemos encontrar datasets de imágenes tomadas por una gran cantidad de sensores y usados para resolver distintos problemas de la visión por

computador, desde detección de objetos, pasando por segmentación, reconstrucción 3D hasta detecciones de cambios.

Además de poder clasificar los datasets en función del objetivo que se tenga, se pueden clasificar por el tipo de sensor que usó a la hora de tomar las fotografías pudiendo escoger entre una gran cantidad incluyendo multiespectral, hyperspectral, PoLSAR, LiDAR..., etc.

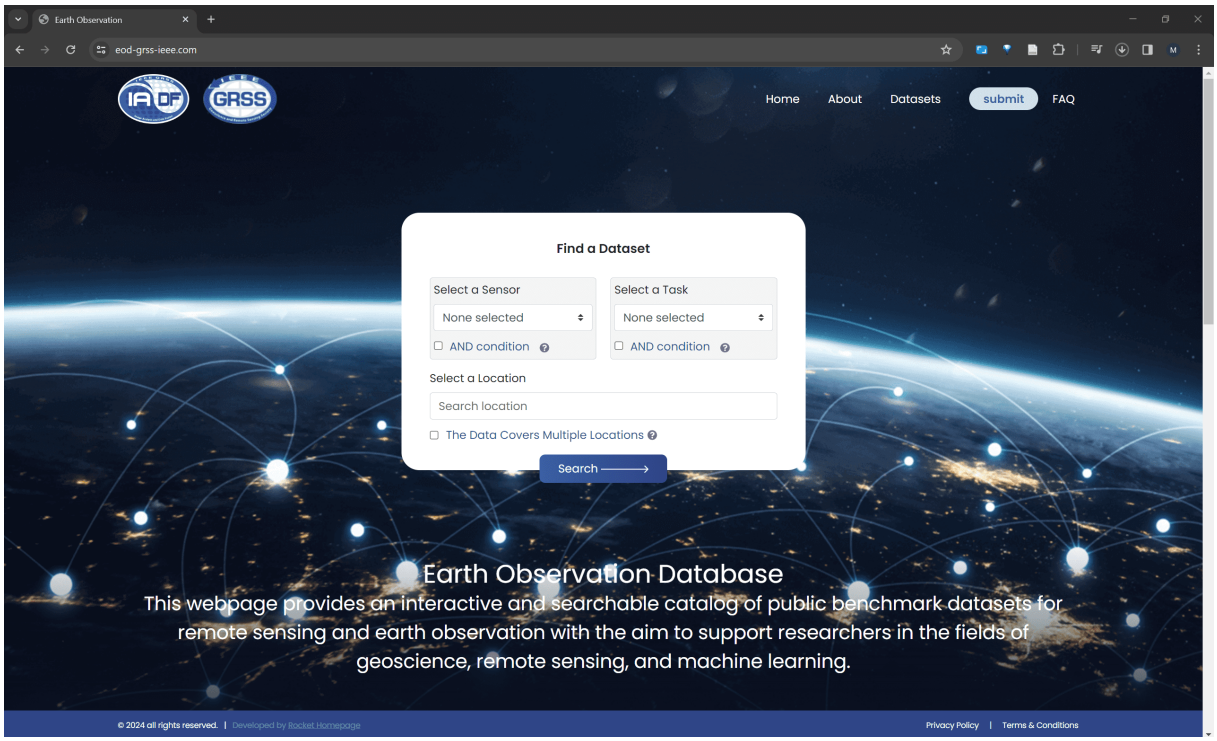

Figura 3.4: Earth Observation Databse

3.1.2.2. Kaggle

Kaggle es una página especializada en conjunto de datos de todo tipo para usarlos en algoritmos de inteligencia artificial. Podemos encontrar una variedad enorme de datos y en todos los formatos posibles. Uno de los subtipos de datos que opera esta página precisamente son los datos de imágenes y en concreto, los conjuntos de datos de imágenes tomadas por satélite.

Esta página a diferencia de la anterior, mantiene una comunidad activa de miembros que pueden puntuar los datasets con diferentes notas haciendo de este un filtro para poder ahorrarse tiempo a la hora de buscar conjuntos de datos. Además puedes filtrar no solo por palabras clave, sino por el tipo de archivos que contiene, las licencias que usan o características especiales.

3.1.2.3. Awesome Satellite Imagery Datasets

Awesome Satellite Imagery Datasets [43] es un GitHub dedicado a la recopilación de conjuntos de datos de imágenes por satelites usados con ténicas de aprendizaje profundo.

Nos adjuntan una lista bastante amplia de datasets enfocadas a diferentes tareas. Una de las tareas en las que se enfoca es en la detección de objetos, que es el caso que nos interesa aunque podremos encontrar conjuntos de datos enfocados a otros propósitos de la visión por computador.

3.1.3. Ficheros de anotaciones

Uno de los temas más importantes a considerar a la hora de buscar un conjuntos de datos, es el formato en el que viene el fichero de anotaciones. El fichero de anotaciones es el archivo que junto a las imágenes completan el conjunto de datos y sirve para poder identificar a los objetos existentes dentro de cada una de las imágenes que viene en el conjunto de datos.

COCO JSON (Microsoft's Common Objets) [53] es el más común a la hora de buscar conjuntos de datos. Frecuentemente es usado cuando se desarrolla un nuevo conjunto de datos de detección de objetos personalizado que como su propio nombre indica almacena la información en archivo **JSON** (JavaScript Object Notation).

Por otra parte tenemos **Pascal VOC** [53] que a diferencia de COCO JSON, este contiene la información almacenada en un archivo **XML** (Extensible Markup Language). Además en Pascal VOC tenemos un archivo de anotación para cada imagen mientras que en COCO tenemos un solo archivo para todo el conjunto de datos.

Otro formato ampliamente utilizado para la anotación de datos de imágenes es **YOLO** [40]. Aunque **YOLO** es conocido principalmente como una técnica de detección de objetos, también se emplea como un método de anotación de conjuntos de datos, específicamente en archivos TXT. Al igual que el formato Pascal VOC, **YOLO** genera un archivo de anotación para cada imagen en el conjunto de datos.

En el siguiente apartado se revisará algunas herramientas que sirve para crear conjuntos de datos etiquetados.

3.1.3.1. Herramientas de creación de conjuntos de datos

Existen herramientas para facilitar la creación de ficheros de anotaciones para crear conjuntos de datos de imágenes propios.

Algunas de estas herramientas podrían ser **Make Sense** [36] la cual nos permite crear estos archivos sin necesidad de instalación, únicamente a través del navegador. Además utiliza herramientas de inteligencia artificial para poder ayudarte en la creación de estos conjuntos de datos.

En contraposición tenemos **labelImg** [20], un proyecto de código abierto en Github que nos permite como en el caso anterior la creación de los conjuntos de datos, pero esta vez de una forma local y teniendo acceso a el código del software para poder

entender o modificar como funciona el mismo.

3.1.4. Algunos ejemplos de conjuntos de datos

Tenemos la suerte de vivir en un mundo informatizado y comunicado, por lo que podemos tener una gran cantidad de conjuntos de datos. Estos se pueden clasificar de múltiples maneras, pero haremos un análisis de algunos conjuntos de datos, indicando su nombre, resolución de las imágenes, distancia aproximada por cada píxel de la imagen, formato de las imágenes, cantidad de imágenes, obtener si el conjunto de datos tiene cajas orientadas o horizontales..., entre otros atributos.

En la siguiente tabla podemos ver los siguientes atributos:

- **Nombre**: Nombre por el que se conoce.
- **Resolución**: Viene dada en píxeles.
- **Formato**: Extensión de las imánges (PNG, JPG...).
- **Nº Imágenes**: Cantidad total de imágenes del conjunto de datos
- **Cajas**: **H** son conjuntos de datos con cajas .horizontales mientras que **O** son conjuntos con cajas orientadas.
- **Metros** / **pixel**: Cantidad de metros a la que corresponde de media cada píxel.
- **Anotaciones**: Cantidad de objetos dentro del conjunto total de datos.
- **Etiquetas**: Clases de objetos diferentes que podemos encontrar en el conjunto de datos.
- **Sensor**: Tipo de sensor con el que se capturó las imágenes.

Nombre	Resolución	Formato	Nº Im	Cajas	Metros / pixel	Anotaciones	Etiquetas	Sensor
Airbus Oil Storage	2560x2560	JPG	98	H	1,50m	13592	1	Óptico
Oil Storage Tanks	512x512	JPG	10.000	H	0,3m	19.942	1	Óptico
DIOR	800X800	JPG	23.463	H y O	0,5m - 3m	192.472	20	Óptico
HRPlanesv2	4800X2703	JPG	11.329	H	0,1m	50.225	1	Óptico
Airbus Aircraft	2560x2560	JPEG	103	H	0,50m	3425	2	Óptico
COWC	1024x1024	PNG		H	0,15m	32716	1	Óptico
FAIR1M	600x600 up to 10000-10000	TIFF	15266	O	0,3 - 0,8m	1020000	5 y 37	Óptico
NWPU VHR-10	1000x1000	JPG	800	H	0,5 - 2m	3775	10	Óptico
HRSC2016	300x300 up to 1500x1500	BMP	1070	O	0,4 - 2m	2976	1	Óptico
xView	2000x2000 up to 4000x4000	PNG	1127	H	0,3m	1000000	60	Óptico
DOTA 2.0	800x800 up to 4000x4000	TIFF	11268	O	-	1793658	18	Óptico
AIR-SARShip-2.0	800x800 up to 1500x1500	JPG	4368	H	1m	16463	1	SAR

Tabla. 3.1: Clasificación de algunos de los conjuntos de datos más importantes

A continuación veremos algunos de los ejemplo de los datasets para analizarlos un poco más en profundidad:

3.1.4.1. Urban Tree Count Detection

Este dataset llamado **Urban Tree Count Detection** [39], como su propio nombre indica, contiene imágenes de ciudades tomadas por satélite. En estas imágenes se encuentran árboles de distintos tipos y distintos tamaños. El objetivo de este dataset es contar el número de árboles que hay en las ciudades usando tecnología de visión por computador.

Una de las curiosidades de este dataset, es que las anotaciones de los objetos, está divididas en diferentes archivos. Un archivo por cada imagen donde en cada línea se encuentran las coordenadas de los árboles encontrados. Además, los ficheros de anotaciones en este caso son **TXT**, los valores están separados por espacios y los objetos por saltos de línea.

Además nos ofrece un archivo **IPYNB** (Interactive Python Notebook) que nos permite cargar los datos de forma sencilla en caso de que utilicemos Python como lenguaje de desarrollo del modelo. Este tipo de archivos los veremos más adelante cuando veamos las herramientas, frameworks y lenguaje de programación que usaremos para el desarrollo de los modelos.

Figura 3.5: Detección de árboles a través de imágenes satelitales [39]

3.1.4.2. Oil Storage Tanks

Los tanques de almacenamiento de petróleo juegan un papel importante en la economía global. El petróleo en crudo es guardado en múltiples tanques repartidos a lo

largo de la esfera terrestre dependiendo de donde se realice la extracción y la venta.

El volumen de petróleo en los tanques es un indicador económico indicando que naciones aumentan o disminuyen la producción. A esto hay que sumarle que la información nacional acerca de las reservas de petróleo no suele ser transparente.

Este repositorio llamado **Oil Storage Tanks** [37] se centra en buscar tanques de petróleo a través de imágenes satelitales, cuenta con un total de más de 100 imágenes de gran resolución divididas a su vez en trozos más pequeños para poder analizarlas. Es decir, un total de 10.000 imágenes.

Además tiene un fichero de anotaciones que es similar al que vimos en la introducción del tema, un archivo **CSV**, pero este no es el que contiene las coordenadas de los objetos, sino que en este caso es un archivo **JSON** en el que cada array del archivo representa una imagen, dentro de este, se encuentra otro array con los objetos encontrados en la imagen.

Figura 3.6: Parte de una imagen que contiene tanques de petróleo [37]

3.2. Técnicas de detección de objetos para imágenes satelitales

Anteriormente, los métodos de detección de objetos de objetos eran creados manualmente a través de descriptores por personas expertas en la visión por computador. Actualmente, conforme ha ido avanzando la capacidad de procesamiento de los componentes hardware y el desarrollo de las redes neuronales, se han podido crear modelos con capacidad de aprendizaje autónomo, por lo que crear métodos de detección de objetos de forma manual quedó totalmente obsoleto.

En un inicio, la detección de objetos se hacía de forma en la que las imágenes que cogía la red neuronal eran tomadas de forma horizontal con respecto a los objetos, por lo que los modelos no tenían que preocuparse por la rotación de los mismos. En caso de las imágenes satelitales, estas están tomadas de forma que los objetos la mayoría de veces no están alineados horizontalmente ya que se toman las imágenes desde satélites.

Por ello, los detectores de objetos horizontales, los cuales valga la redundancia detectaban cajas horizontales, sufren de los siguientes problemas:

- Las **HBB** (Horizontal Bounded Boxes) tienden a contener antecedentes, es decir, demasiado fondo, lo que hace que la redes neuronales apenas capte características discriminatorias vitales para las tareas de regresión y clasificación.
- La intersección sobre Unión (IoU) entre una HBB y las HBB adyacentes puede ser muy grande en escenarios de disposición densa, especialmente para objetos con relaciones de aspecto extremadamente grandes.

La IoU (Intersección sobre Unión) [23] es una métrica comúnmente utilizada para evaluar el rendimiento de algoritmos de detección de objetos, especialmente en tareas de visión por computadora. El IoU se define como la proporción del área de superposición entre las cajas delimitadoras predichas y las verdaderas delimitadoras al total de sus áreas de unión.

Debido a esto, las OBB (Oriented Bounded Boxes) son más apropiadas para la detección de objetos en imágenes satelitales. Además, debido a que las OBB se han hecho tan populares existen una gran cantidad de conjuntos de datos que vienen con fichero de anotaciones especialmente pensados para cajas orientadas.

En la figura 3.7 se puede ver una distinción visual entre las Oriented Bounded

Boxes y las Horizontal Bounded Boxes.

En los últimos años se ha podido ver una evolución en este campo, llegando incluso a poder formar una taxonomía centrada únicamente en la detección de objetos en imágenes satelitales.

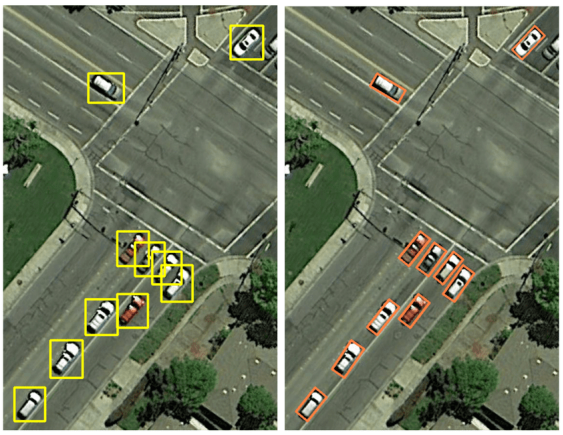

Figura 3.7: A la izquierda se aprecian OBB mientras que a la derecha HBB [52]

3.2.1. Taxonomía de la detección de objetos en imágenes satelitales

En la figura 3.8 se presenta la taxonomía de la detección de objetos en imágenes satelitales propuesta en el artículo Remote Sensing Object Detection Meets Deep Learning: A metareview of challenges and advances [56]. En este, se clasifica las distintas áreas de la detección de objetos en imágenes satelitales en las siguientes categorías:

- **Detección de objetos multiescala**: Se refiere a la capacidad de un sistema de detección para identificar objetos en diferentes escalas o tamaños dentro de una imagen, permitiendo reconocer tanto objetos grandes como pequeños sin perder precisión.
- **Detección de objetos rotados**: Implica la habilidad de detectar objetos que están orientados en ángulos no convencionales o rotados con respecto a su alineación típica, manteniendo la precisión en la detección a pesar de la rotación.
- **Detección de objetos débiles**: Consiste en identificar objetos que tienen características sutiles o poco contrastadas dentro de la imagen, como sombras o reflejos, que hacen difícil su detección con métodos tradicionales.
- **Detección de objetos pequeños**: Se refiere a la capacidad de detectar objetos muy pequeños dentro de una imagen, que suelen estar representados por un número limitado de píxeles y, por lo tanto, son difíciles de identificar con precisión.

Figura 3.8: Taxonomía de la detección de objetos en imágenes satelitales [56]

3.2.1.1. Detección de objetos multiescala

Los objetos a detectar en una imagen pueden variar de tamaño entre ellos, es decir, no tiene el mismo tamaño un árbol que un parque. La detección de objetos con múltiples escalas es un reto notorio. En este problema también juega un papel crucial la **resolución espacial**, esta es el tamaño del objeto dentro de la fotografía, esto quiere decir que dependiendo del sensor que se esté utilizando y la distancia del satélite con la superficie terrestre, el objeto puede variar en varias órdenes de magnitud el tamaño del mismo en la fotografía.

Este problema afronta la detección de distintos objetos en la misma foto teniendo diferentes clasificaciones de objetos. Un ejemplo de esto sería detectar un pequeño vehículo en un aeropuerto que tendría una resolución espacial de 10 píxeles y a la vez un aeropuerto que su resolución espacial podrían rondar los 100.000 píxeles en la misma foto.

Un ejemplo claro de multiescala lo podemos observar en la figura 3.9, en esta vemos dos aviones que tienen diferente tamaño, ambos son aviones pero tienen diferente escala por lo que se trataría de un problema dentro de la misma categoría, mientras que vemos que al lado hay un vehículo que pertenece a otra categoría y la diferencia de tamaño con el avión es bastante por lo que esto podría ser un ejemplo de problemas de escala entre categorías.

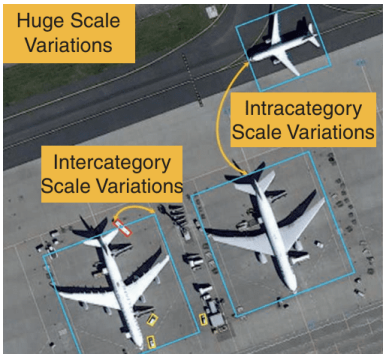

Figura 3.9: Ejemplo de objetos multiescala [56]

Para abordar este problema, los estudios actuales están divididos entre el aumento de datos, la representación de características a múltiples escalas y la generación de anclas a múltiples escalas. A continuación se explica en que consiste cada uno de ellos:

- **Aumento de datos**: Se trata de un conjunto de técnicas utilizadas para incrementar la cantidad y diversidad del conjunto de datos de entrenamiento sin real-

mente recolectar nuevos datos. Esto se logra aplicando diversas transformaciones a las imágenes existentes para crear nuevas versiones de ellas.

- **Representación de características a múltiples escalas**: Los estudios iniciales de la detección de objetos se basaban en un único mapa de características, sin embargo, esta predicción con un único mapa de características a una sola escala limita la capacidad del detector para manejar objetos con una amplia gama de escalas. Actualmente se han seguido tres filosofías distintas para abordar este problema, las cuales son la integración de características a múltiples escalas, jerarquía de características piramidales y redes de pirámides de características.
- **Generación de anclas a múltiple escala**: Un ancla en la detección de objetos es una caja delimitadora inicial de referencia que se utiliza como partida para predecir la ubicación y el tamaño de los objetos en la imagen, pues, la generación de anclas a múltiple escala lo que hace es modificar las configuraciones de las anclas en la detección de objetos común para cubrir mejor las escalas de los objetos.

3.2.1.2. Detección de objetos rotados

Otro reto adicional en la detección de objetos a través de imágenes satelitales, es la rotación de los objetos en la propia imagen. Al principio de este punto hablábamos de las Oriented Bounded Boxes y las Horizontal Bounded Boxes viendo que las segundas no representaban con suficiente precisión los objetos y podían contener antecedentes. Recordemos que nos referimos a antecedentes en este ámbito al concepto de contener demasiado fondo, lo que hace que la redes neuronales apenas capte características discriminatorias vitales para las tareas de regresión y clasificación.

Debido a esto, muchos investigadores se centraron en la detección de objetos con rotaciones arbitrarias para obtener unos resultados precisos. Una vez más podemos ver un ejemplo de objetos rotados en la figura 3.10.

Los detectores de objetos más recientes puedes entender los objetos rotados de varias formas, entre ellas, las dos más importantes, son las siguientes:

1. **Representación con cinco parámetros**: Esta representación es la más utilizada y las más famosa debido a su sencillez ya que únicamente añade un ángulo θ, al final se obtienen con cinco parámetros (x,y,w,h,θ). Los parámetros x e y son las coordenadas dentro de la imagen mientras que los parámetros w y h son las dimensiones de la caja.

Figura 3.10: Ejemplo de objetos con rotación arbitraria [56]

2. **Representación con ocho parámetros**: En contraposición al modelo de cinco parámetros, este funciona indicando en cuatro pares de coordenadas los vértices del objetos encontrado (a_x,a_y), (b_x,b_y), (c_x,c_y), (d_x,d_y).

3.2.1.3. Detección de objetos débiles

Los objetos de interés en las imágenes satelitales suelen estar en escenas complejas con patrones espaciales llenos de objetos y grandes cantidades de ruido de fondo. El contexto complejo y el ruido de fondo dañan la representación de las características de los objetos de interés, lo que resulta en respuestas de características débiles para los objetos. Por lo tanto, muchos trabajos se han enfocado en mejorar la representación de las características de los objetos de interés, que se pueden agrupar en dos corrientes: la supresión del ruido de fondo y la extracción de información contextual relacionada.

Un ejemplo de objetos débiles lo podemos apreciar en la figura 3.11 en el que vemos que al detectar camiones, pueden existir en esta imagen dos problemas. El primero que tenga **apariencia similar** a otros objetos de la zona, como podemos ver, similares a las protuberancias del almacén que está al lado y el segundo es que puedan estar en un **contexto complejo** como es este caso y alguno de los camiones queden parcialmente cambiados por la sombra.

Una posible solución para la detección de objetos débiles consiste en **mejorar la sensibilidad del modelo** mediante técnicas de mejora de contraste y el uso de redes neuronales profundas con mecanismos de atención.

Estas redes pueden concentrarse en las áreas de interés de una imagen, permitiendo identificar características sutiles que son difíciles de detectar con métodos

tradicionales. Además, el uso de técnicas de aumento de datos que introduzcan variaciones en las condiciones de iluminación y ruido también puede ayudar a entrenar modelos más robustos ante objetos poco contrastados o con bordes difusos.

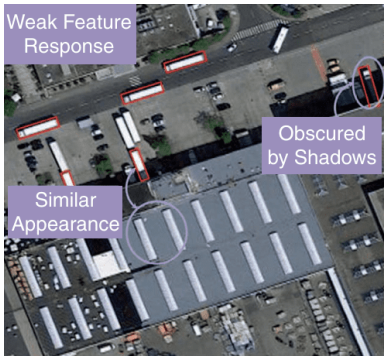

Figura 3.11: Ejemplo de objetos débiles [56]

3.2.1.4. Detección de objetos pequeños

El GSD (Ground Sample Distance) típico para las imágenes de sensores remotos es de 1 a 3 metros de media como vimos en la tabla de los conjuntos de datos anteriormente, lo que significa que incluso los objetos grandes (por ejemplo, aviones, barcos y tanques de almacenamiento) pueden ocupar menos de 16x16 píxeles. Además, incluso en imágenes satelitales de alta resolución con un GSD de 0.25 metros, un vehículo con una dimensión de 3x1.5 m^2 cubre solo 72 píxeles (12x6). Esta prevalencia de objetos pequeños en las imágenes satelitales aumenta aún más la dificultad de la detección de objetos pequeños.

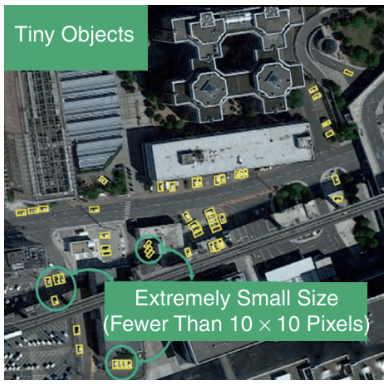

Figura 3.12: Ejemplo de objetos pequeños [56]

En la figura 3.12 en el que podemos ver detección de vehículos dentro del aeropuerto, los objetos dentro de la imagen son extremadamente pequeños, más chicos que 10 píxeles por 10 píxeles.

Este problema se puede tratar actualmente a través de 3 formas, mediante **aprendizaje de características discriminativas**, **métricas de detección mejoradas** utilizando métricas de detección bien diseñadas para objetos pequeños y **métodos basados en superresolución** que se encargan de ampliar la calidad de la imagen haciendo que esta tenga más píxeles.

3.2.2. Repositorios de métodos para la detección de objetos

Al igual que para la búsqueda de conjuntos de datos, existen sitios y repositorios en los que podemos encontrar modelos codificados que podemos usarlos teniendo el software requerido para ejecutarlo.

En sitios como Github podemos encontrar modelos que han sido construidos para tareas en específico. Cada uno de ellos se basa en una técnica y además la ventaja que tienen estos es la implementación. Esto quiere decir que el modelo ya viene codificado en su mayor parte. La mayoría de estos, vienen programados en python con frameworks como *PyTorch* o *Tensorflow* que son ampliamente utilizados en el ámbito del deep learning.

Una de las gran ventajas que nos aportan estos modelos ya implementados, es que han sido testeados por lo que en la descripción del modelo suele venir un resumen con el funcionamiento general del mismo y además los campos en los que suele destacar.

Además de ser diferentes modelos, también suelen venir entrenados para algunas tareas en específico. Por ejemplo, pueden venir entrenados para el reconocimiento de barcos, paneles solares, edificios, vehículo, trenes y un largo etcétera.

3.2.2.1. Satellite-image-deep-learning

Uno de los repositorios en Github que nos encontramos es *Satellite-image-deep-learning* [46], este es uno de los repositorios más grandes públicos en el que se recogen detectores testeados de imágenes satelitales.

Este repositorio alberga una gran cantidad de técnicas especializadas en la visión por computador. Es decir, no solo se centra en el reconocimiento de objetos sino que también posee otro tipo de objetivos como la clasificación, el detector de cambios..., etc.

En este caso, para la detección de objetos, dispone de detectores entrenados para situaciones diferentes. En concreto tiene varios apartados con tareas bastantes específicas como pueden ser la detección de animales o la detección de aviones y barcos, pero, también dispone de tareas más generales como técnicas especializados en el tracking de objetos en vídeo.

A continuación se puede ver una clasificación de las distintas técnicas que recopila este repositorio:

- Tracking de objetos en vídeo.
- Detección de objetos con cajas rotadas.
- Detección de objetos mejorada por super resolución.
- Detección de objetos destacados.
- Detección de objetos: Edificios, techos y paneles solares.
- Detección de objetos: Barcos y embarcaciones.
- Detección de objetos: Coches, vehículos y trenes.
- Detección de objetos: Aviones y aeronaves.
- Detección de objetos: Infraestructura y servicios públicos.
- Detección de tanques de almacenamiento de petróleo.
- Detección de animales.
- Detección de objetos diversos.

Cada uno de estos apartados contiene decenas de técnicas que corren en diferentes lenguajes junto a diferentes frameworks de programación. Por ello, es importante revisar los requisitos de cada una de estas técnicas. Además hay que añadir que estas técnicas no vienen con el conjunto de datos, por lo que un consejo antes de probar alguna técnicas, es comprobar antes de descargarlo, que el conjunto de datos con el que ha sido entrenado está disponible para poder testearlo.

3.2.3. Algunos ejemplos de técnicas

A continuación se presentarán una técnica para cada una de las categorías anteriores extraídas del repositorio *Satellite-image-deep-learning* [46]. Para obtener más información relativa a cada una de las siguientes técnicas se puede consultar el repositorio de Github.

3.2.3.1. Objetos con cajas rotadas: Mmrotate

Como su propio nombre indica, en este apartado se pueden encontrar técnicas especializadas para la detección de objetos con cajas rotadas, es decir, con Oriented Bounded Boxes.

Mmrotate [17] es un kit de herramientas de código abierto para la detección de objetos rotados basado en PyTorch. Es una parte del proyecto de OpenMMLab.

La mayoría de detectores de objetos rotados están basados en detectores de objetos clásicos generales, por ellos, las cajas horizontales no han sabido satisfacer las necesidades de los investigadores. Lo llaman detección de objetos giratorios al redefinir la representación del objeto y aumentar el número de grados de libertad de regresión para lograr la detección de rectángulos, cuadriláteros e incluso formas arbitrarias giradas

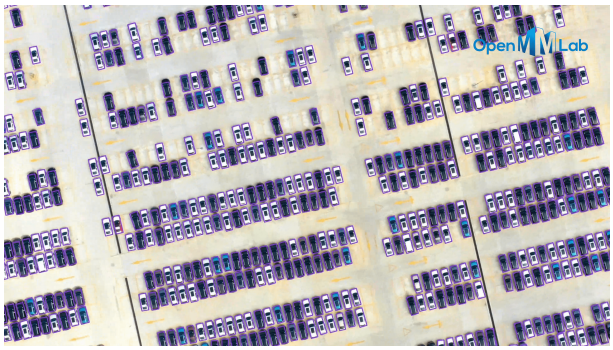

Figura 3.13: Ejemplo del funcionamiento de Mmrotate con imágenes rotadas

3.2.3.2. Edificios, tejados y paneles solares: Building Detection in the Spacenet Satellite Imagery

Otros proyectos que podemos encontrar de detección de objetos son los relacionados con la detección de edificios, es decir, todo tipo de infraestructuras como por ejemplo, centrales térmicas, centrales hidráulicas o incluso, por qué no, bases militares.

El proyecto **Building Detection in the Spacenet Satellite Imagery** [16] busca detectar edificios haciendo uso de un detector SSD, por lo tanto, una de las ventajas que tendrá este proyecto es la rapidez con la que obtendrá los resultados.

En la figura 3.14 vemos un ejemplo de SSD aplicado a el conjunto de datos de Spacenet en el que se puede ver como el algoritmo predice los posibles edificios.

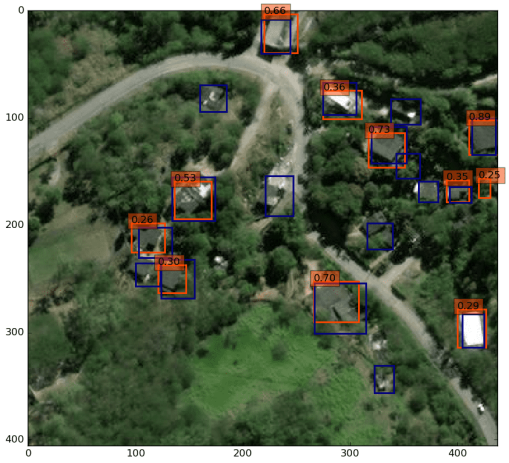

Figura 3.14: Detección de edificios [16]

3.2.3.3. Detección de objetos mejorada por superresolución: EESRGAN

Uno de los problemas que veíamos cuando analizábamos la taxonomía de la detección de objetos en imágenes satelitales es la **detección de objetos pequeños**, y una de las posibles soluciones a este problema eran los métodos basados en superresolución, por ello esta técnica trabaja con un modelo basado en una red generativa adversarial (GAN) llamado ESRGAN (Enhanced Super-Resolution GAN) mostrando un notable rendimiento en la mejora de imágenes.

Este repositorio llamado **EESRGAN** [47] ofrece una técnica, en la que antes de pasar a la detección de objetos hay una capa intermediaria que se encarga de aplicar técnicas de superresolución para poder ofrecer una detección de objetos mejorada.

Como podemos ver en la figura 3.15, aplicando una técnica de superresolución previa a la detección conseguimos la detección de un vehículo adicional que previamente el modelo no había detectado. En la primera columna se puede apreciar la imagen previa a aplicar la técnica de superresolución, en la segunda columna una vez ya aplicada y la tercera muestra la imagen verdadera.

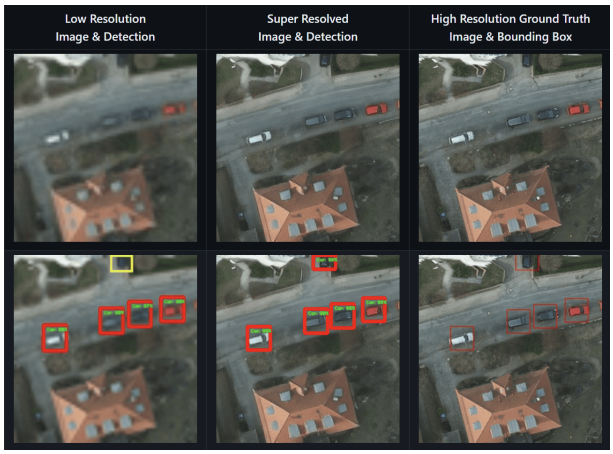

Figura 3.15: Funcionamiento de EESRGAN [47]

3.2.3.4. Barcos y embarcaciones: Ship Detection from Satellite Images using YOLOV4

Ship Detection from Satellite Images using YOLOV4 [48] como su propio nombre indica es un detector de barcos por imágenes satélite que está implementado en YOLO versión 4 usando uno de los conjuntos de datos que podemos encontrar en Kaggle llamado *Airbus Ship Detection*.

Este ejemplo es bastante interesante porque es lo contrario al caso anterior. El apartado anterior se centraba en la investigación de técnicas para mejorar la detección de objetos rotados mientras que este repositorio hace una implementación específica. Además es interesante ver como algunos de los barcos que no están alineados ni vertical ni horizontalmente, la caja que los envuelve no se adapta a la forma del barco.

Figura 3.16: Imagen resultado del procesamiento del modelo *Ship Detection from Satellite Images using YOLOV4*

3.2.3.5. Coches, vehículos y trenes: COWC Car Counting

A diferencia de las otras técnicas, **COWC Car Counting** [29] además de detectar los vehículos, cuenta los vehículos que hay en cada parcela de las imágenes. Esta técnica hace uso de uno de los datasets previamente vistos en este trabajo llamado COWC.

Este repositorio contiene una serie de scripts de python y hojas de jupyter notebook para entrenar y evaluar redes neuronales convolucionales.

En la figura 3.17 se puede observar un ejemplo del funcionamiento de esta técnica en la que se puede ver la imagen divididas en parcelas, estas conteniendo un número que representa los coches encontrados dentro de esas parcelas.

Figura 3.17: Ejemplo de la técnica COWC Car Counting [29]

3.2.3.6. Aviones y Aeronaves: Deep Learning for Aircraft Recognition

Hoy en día, muchas ramas militares de todo el mundo están invirtiendo en Visión por Computador para Inteligencia, Vigilancia y Reconocimiento (ISR). Algunas aplicaciones relacionadas con este campo incluyen el reconocimiento de vehículos militares en las redes sociales [19], el reconocimiento de objetivos militares camuflados y la detección de objetos en el campo de batalla.

En el proyecto **Deep Learning for Aircraft Recognition** [17] se utiliza una red neuronal convolucional (CNN) para clasificar una serie de aeronaves militares a través de imágenes satelitales. El objetivo del proyecto es construir una red neuronal convolucional capaz de distinguir y clasificar con precisión aeronaves a través de una instantánea de imágenes de satélite.

Este proyecto es capaz de distinguir entre los siguientes tipos de aviones: B-1, B-2, B-52, C-17, C-130, E-3, C-135, KC-10 y C-5. A continuación en la figura se puede ver una porción de imagen en la que se detectan cuatro aviones C-130.

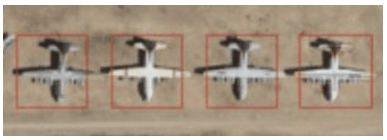

Figura 3.18: Identificación de aviones C-130 [17]

3.2.3.7. Infraestructura y servicios públicos: Wind Turbine Object Detection

Wind Turbine Object Detection [3] contiene un cuaderno de Jupyter y archivos de apoyo para entrenar un detector de objetos de turbinas eólicas utilizando la API de Detección de Objetos de TensorFlow.

En este repositorio se detectan turbinas eólicas a partir de imágenes aéreas tomadas sobre el centro-oeste de Iowa. El pipeline completo, desde el entrenamiento hasta la inferencia, está contenido en el cuaderno con explicaciones detalladas para cada paso del proceso. Esto puede servir como un tutorial para aquellos interesados en entrenar su propio modelo de detección de objetos personalizado.

En la figura 3.19 se puede ver un ejemplo del funcionamientode esta técnica.

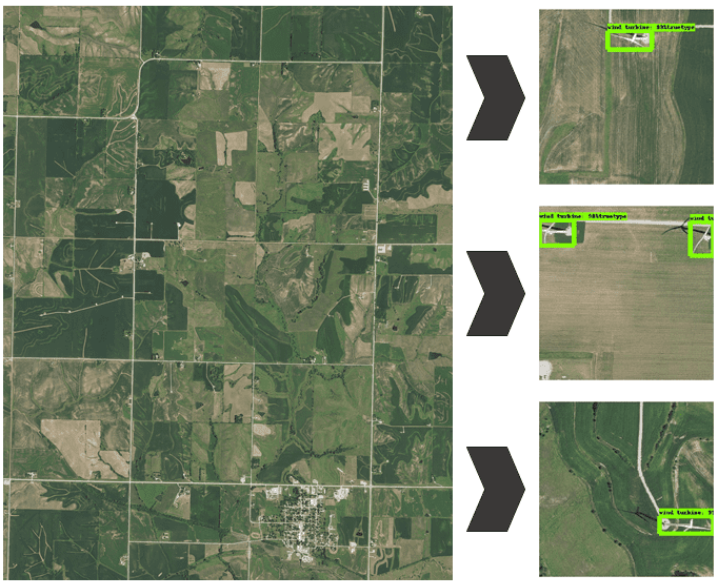

Figura 3.19: Detección de aerogeneradores usando el repositorio Wind Turbine Object Detection [3]

3.2.3.8. Almacenamiento de petróleo: Oil Storage Detection on Airbus Imagery with YOLOX

Como comenté anteriormente, las reservas de petróleo de un país son un marcador importante del estado económico del mismo, por ello, una forma de saber las reservas de petróleo, es contando los bidones. Para esta tarea se puede usar detección de objetos y contar el número de bidones totales.

Oil Storage Detection on Airbus Imagery with YOLOX [12] es un artículo en el que se entrena un modelo de YOLOX para poder hacer la tarea anteriormente mencionada, este se pública después de que Kaggle junto a la empresa Airbus haga una competición adjuntando un dataset.

En la figura 3.20 podemos ver una imagen de conjunto de datos ofrecido por Kaggle que ha sido procesada por el modelo y como se puede observar, los bidones de tanques son remarcados por un cuadro rojo.

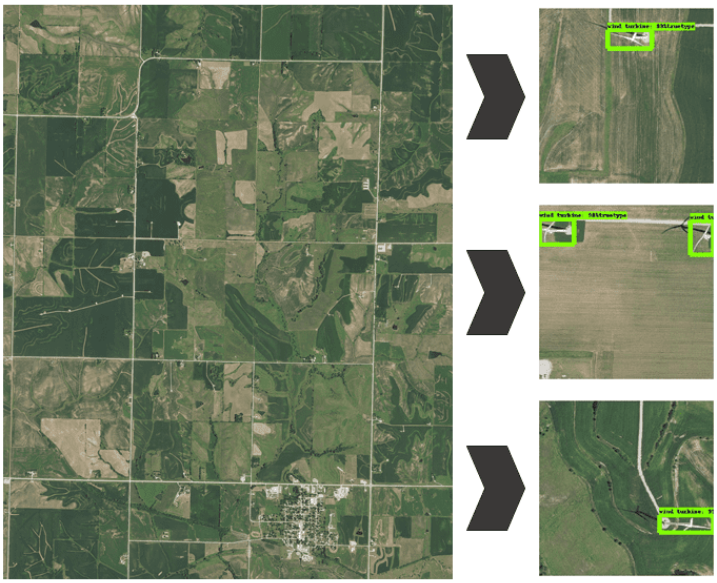

Figura 3.20: Detección de tanques usando YOLOX [12]

3.2.3.9. Animales: Elephant Detection

Existen multitud de usos en la detección de objetos con imágenes satelitales, pero, todos o la mayoría están relacionados con infraestructuras o elementos creados por los humanos. En este caso, se puede ver un uso algo distinto a los demás apartados.

Elephant Detection [28] es un ejemplo de uso no habitual en el que haciendo uso de modelos de detección de objetos como RetinaNet se puede detectar animales y hacer un conteo para poder obtener los ejemplares exactos existentes o al menos un número aproximado.

Además este proyecto se realiza mediante condiciones reales, es decir, los elefantes como es lógico, no siempre están en explanadas en las que los sensores de los satélites pueden tomarle fotos completas, si no, que están en sitios con vegetación, por lo que esto dificulta la tarea. A continuación en la figura 3.21 se puede ver un ejemplo de esto mismo.

Figura 3.21: Detección de elefantes usando RetinaNet [28]

3.3. Entorno de las pruebas prácticas

3.3.1. Introducción

A la hora de implementar las técnicas vistas en este trabajo, hay que tener experiencia en el ámbito de la informática para poder realizar y llevar a cabo las pruebas pertinentes de forma exitosa.

Para ello requerimos de software y hardware específico que sea capaz de ejecutar de forma eficaz los algoritmos.

En cuanto al hardware, uno de los componentes clave para realizar estas pruebas son las tarjetas gráficas, comúnmente conocidas como GPUs. Estas son esenciales, ya que se encargan de llevar a cabo las operaciones matemáticas necesarias para calcular los pesos de la red neuronal que se esté utilizando.

En cuanto al software, utilizaremos lenguajes de programación que, gracias a sus librerías, frameworks y facilidad de uso, facilitan el desarrollo y mantenimiento de la implementación de este tipo de técnicas. En este caso, Python es el claro vencedor debido a la comunidad tan grande que respalda el desarrollo de algoritmos de aprendizaje profundo para este lenguaje.

3.3.2. Hardware

3.3.2.1. Entorno usado en este trabajo

Como he mencionado anteriormente, es necesario tener un buen equipo para poder ejecutar las pruebas, sobretodo, en cuanto a tiempo se refiere. Cualquier ordenador con un procesador moderno, podría en teoría ejecutar estos algoritmos, pero el tiempo necesario para su ejecución sería demasiado elevado.

Para este proyecto se ha utilizado un servidor dedicado prestado por el grupo de investigación SIMIDAT de la Universidad de Jaén. En este caso este equipo, un NVIDIA DGX-1 cuenta con las siguientes especificaciones:

- **Procesadores**: 4 procesadores Intel(R) Xeon(R) CPU E5-2698 v4 @ 2.20GHz
- **Tarjetas gráficas**: 8 tarjetas gráficas dedicadas Nvidia Tesla V100-SXM2-32GB
- **Memoria RAM**: 512 Gigabytes.

Como se puede apreciar, no todo el mundo puede hacer uso de este tipo de equipos porque el hardware que monta este servidor en conjunto es una inversión que solo se realizaría en caso de tener que necesitarlo.

Adicionalmente, cabe recalcar, que no todo los componentes de este servidor están siendo utilizados debido a que los recursos se comparten con más usuarios y el software no puede aprovechar la totalidad de los recursos del servidor.

Debido a que es una inversión considerable, existen otras alternativas para poder ejecutar estos algoritmos sin tener que comprar el hardware. Se verá en el siguiente apartado.

3.3.2.2. Alternativa: Google Colab

Como hemos visto, una de las opciones que se pueden considerar para ejecutar algoritmos de detección de objetos es tener nuestro propio hardware, pero no siempre es así, por ello, existen alternativas que son gratuitas o de pago para poder alquilar por cierto tiempo los recursos hardware necesarios.

Google Colab es una plataforma **gratuita** que ofrece recursos computacionales en la nube, permitiendo a los usuarios ejecutar código Python sin necesidad de un

hardware potente propio. Esta herramienta es especialmente útil para aquellos que desean experimentar con algoritmos de aprendizaje profundo o ejecutar modelos de inteligencia artificial sin preocuparse por las limitaciones de su equipo personal.

Figura 3.22: Logo de Google Colab

Proporciona acceso a tarjetas gráficas y TPUs (Tensor Processing Unit), lo que resulta ideal para acelerar el entrenamiento de modelos de redes neuronales profundas.

Otra ventaja destacada de Google Colab es su simplicidad de uso y la facilidad para compartir proyectos. Los notebooks de Colab pueden ser compartidos de manera sencilla, permitiendo la colaboración entre varios usuarios en tiempo real.

Sin embargo, Google Colab tiene algunas limitaciones. Al ser un servicio gratuito, el acceso a los recursos de hardware no está garantizado y puede estar limitado por períodos de tiempo. Los usuarios pueden ser desconectados después de ciertos períodos de inactividad, y los tiempos de uso de GPU y TPU están sujetos a cuotas, especialmente para usuarios no suscritos a la versión Colab Pro, que ofrece recursos de hardware de mayor calidad y tiempos de uso más prolongados.

3.3.2.3. Otras alternativas

Además de Google Colab, existen otras opciones de computación en la nube que permiten alquilar recursos computacionales bajo demanda. Algunos de los proveedores más conocidos incluyen **Amazon Web Services** (AWS), **Microsoft Azure**, y **Google Cloud Platform** (GCP). Estos servicios ofrecen una variedad de opciones de hardware, incluyendo instancias con GPU de última generación, que son ideales para tareas de aprendizaje profundo.

Amazon Web Services (AWS), por ejemplo, ofrece instancias EC2 con GPU que pueden ser configuradas para satisfacer las necesidades específicas de cualquier proyecto. AWS también proporciona el servicio SageMaker [35], que está diseñado específicamente para desarrollar, entrenar y desplegar modelos de aprendizaje automático a gran escala.

Microsoft Azure ofrece una variedad de máquinas virtuales con capacidades de GPU, además de servicios gestionados como Azure Machine Learning [2], que permiten a los usuarios construir e implementar modelos de aprendizaje automático de manera más eficiente. Azure también ofrece instancias especializadas con hardware específico para cargas de trabajo de inteligencia artificial.

Google Cloud Platform (GCP) ofrece instancias de Compute Engine con GPU y TPU, que están diseñadas para cargas de trabajo de alta computación, como el entrenamiento de modelos de aprendizaje profundo. GCP también incluye herramientas como AI Platform [41], que facilita la creación, el entrenamiento y la implementación de modelos de aprendizaje automático.

Al optar por una de estas plataformas de computación en la nube, es posible personalizar los recursos según las necesidades del proyecto, lo que permite escalar los recursos de manera flexible y eficiente. Sin embargo, a diferencia de Google Colab, estos servicios suelen tener un costo asociado basado en el uso de recursos, lo que puede ser un factor a considerar para proyectos de largo plazo o con presupuestos limitados.

3.3.2.4. Conclusión sobre las alternativas hardware

La elección de la plataforma de hardware depende de diversos factores como el presupuesto, los requerimientos específicos del proyecto, y el nivel de experiencia del equipo de trabajo. Mientras que Google Colab es una excelente opción para proyectos más pequeños, educativos o experimentales, las plataformas de computación en la nube como AWS, Azure, y GCP son más adecuadas para proyectos más grandes o que requieren escalabilidad y recursos dedicados. En última instancia, la decisión dependerá de las necesidades particulares de cada usuario o equipo de investigación.

3.3.3. Software

Uno de los inconvenientes que se presenta cuando manejamos algoritmos de aprendizaje profundo es el manejo de las dependencias y ecosistema necesario. Esto quiere decir que para poder ejecutar una determinada técnica como podría ser una implementación en Python de YOLO, necesitamos un sistema operativo capaz de ejecutar estas tareas, con drivers específicos para hacer uso de la GPU, una versión específica de un lenguaje de programación junto a una serie de librerías que también

por supuesto necesitan una versión específica, esto es muy importante, debido a que el problema de incompatibilidades entre versiones está a la orden del día.

Para poder manejar este tipo de situaciones, veremos punto por punto el software necesario para poder manejar el entorno de forma relativamente sencilla.

Cabe recalcar que todo el software que se ha seleccionado para el trabajo es gratuito, pero algunos de los software mencionados puede tener restricciones para uso comercial.

El software escogido ha sido determinado en su mayoría por el software preinstalado en la máquina DGX-1, esto debido a que el software existente en la máquina era totalmente compatible con los objetivos del trabajo y por tanto no es necesario cambiarlo.

3.3.3.1. Ubuntu 22.04.4 LTS

El sistema operativo principal escogido es Ubuntu 22.04.4 debido a que en las instalaciones de servidores de GPU de alto rendimiento de la Universidad de Jaén, se hace uso de este sistema operativo.

Figura 3.23: Logo de Ubuntu

Ubuntu 22.04.4 LTS es una distribución de Linux que ofrece un entorno estable y bien soportado, ideal para entornos de desarrollo de alto rendimiento como el que se utiliza en este trabajo. Este sistema operativo es compatible con una amplia variedad de herramientas y bibliotecas necesarias para el desarrollo de algoritmos de aprendizaje profundo, incluyendo soporte nativo para CUDA y las últimas versiones de controladores de NVIDIA.

3.3.3.2. Docker 24.07

En este proyecto, Docker 24.07 se ha utilizado para crear contenedores que permiten ejecutar el software en un entorno controlado y replicable. Docker facilita la portabilidad del entorno de desarrollo, permitiendo que los mismos contenedores se ejecuten

en diferentes sistemas operativos sin problemas de compatibilidad.

Esto es especialmente útil para gestionar complejas dependencias de software y para colaborar en equipos distribuidos.

Además es especialmente útil para poder tener **ejecuciones paralelas** debido a que se puede asignar una tarjeta gráfica distinta para cada contenedor.

Figura 3.24: Logo de Docker

Otra ventaja adicional es poder **compartir almacenamiento entre los diferentes contenedores** facilitando el trabajo para poder manejar los datos necesarios para los algoritmos, como por ejemplo los conjuntos de datos.

3.3.3.3. NVIDIA Cuda compiler driver 11.5

Para aprovechar el poder de las GPU en el servidor utilizado, se necesita el controlador NVIDIA CUDA. CUDA es una plataforma de computación paralela y un modelo de programación que permite aumentar el rendimiento computacional, aprovechando la potencia de las GPU de NVIDIA.

Esta versión es especialmente adecuada para los últimos modelos de tarjetas gráficas como las Nvidia Tesla V100-SXM2-32GB utilizadas en el servidor.

3.3.3.4. Python

Python se ha elegido como el lenguaje de programación principal para este trabajo debido a su popularidad en el ámbito del aprendizaje profundo y la gran cantidad de bibliotecas y herramientas disponibles. Proporciona un equilibrio adecuado entre estabilidad y compatibilidad con las librerías modernas de aprendizaje automático y ciencia de datos, como TensorFlow, PyTorch y Scikit-learn.

Figura 3.25: Logo de Python

3.3.3.5. Jupyter Notebook

Jupyter Notebook ha sido una herramienta esencial para este proyecto. Se ha empleado para desarrollar, ejecutar y documentar el código de manera interactiva, lo que facilita el proceso de desarrollo y prueba de modelos de aprendizaje profundo. Su capacidad para integrar código, texto y visualizaciones en un solo documento ha mejorado la colaboración y la comunicación dentro del equipo.

3.3.3.6. Github

Github como plataforma de control de versiones para gestionar el código fuente del proyecto ofrece un entorno colaborativo donde los miembros del equipo pueden revisar cambios, reportar errores y gestionar la evolución del software. Además, facilita la integración con herramientas de automatización que ayudan a mantener la calidad del código y simplifican el despliegue.

En este caso, para este trabajo, se ha utilizado para poder descargar el software necesario para realizar las pruebas.

3.3.3.7. Google drive

Para la gestión de datos, Google Drive ha sido la herramienta elegida. Proporciona un almacenamiento en la nube accesible y seguro, permitiendo compartir fácilmente conjuntos de datos grandes y otros recursos entre los miembros del equipo.

3.3.3.8. Software adicional

- **Unzip**: Gestor de paquetes comprimidos .zip
- **Gdown**: Librería de python utilizada para descargar archivos de google drive

- **Nano**: Editor de texto para sistemas Unix basado en curses

Capítulo 4

Experimentación y resultados

En esta sección se presentan los **experimentos realizados** y los **resultados** obtenidos en el marco de este trabajo, enfocado en la detección de objetos mediante el uso de técnicas de *deep learning* en imágenes satelitales. El objetivo de esta fase es **evaluar el rendimiento** de los distintos modelos seleccionados, comparando su capacidad de detección en conjuntos de datos específicos.

La experimentación se ha estructurado en varias etapas clave. Primero, se **describen las técnicas empleadas** para la detección de objetos, proporcionando una visión general de los algoritmos y herramientas utilizados. Posteriormente, se detalla el proceso de preparación del entorno de trabajo, incluyendo la instalación del software y la configuración de los contenedores para los modelos YOLOv5, YOLOv8 y Detectron2.

En la siguiente fase, se **discute la preparación de los conjuntos de datos**, especificando los formatos utilizados (YOLO y COCO) y los scripts desarrollados para su conversión y procesamiento. También se incluyen los detalles de los conjuntos de datos empleados en los experimentos, como *HRPlanesv2*, *OilStorageTanks* y *DIOR*.

A continuación, se describen las métricas de evaluación empleadas, como *mAP50*, *mAP50-95* y *F1 Score*, que permiten cuantificar el rendimiento de los modelos. Se ofrece una explicación sobre la configuración del entrenamiento de los modelos, detallando los hiperparámetros ajustados para YOLOv5, YOLOv8 y Faster R-CNN en Detectron2.

Finalmente, se **presentan los resultados** obtenidos para cada uno de los conjuntos de datos, analizando el desempeño de los modelos en cada caso y comparando su efectividad en función de las métricas establecidas.

4.1. Técnicas usadas de detección de objetos

Se han estudiado diferentes técnicas de detección de objetos a lo largo de este trabajo, pero en nuestro caso, escogeremos para las pruebas 3 de las mencionadas.

- **YoloV5** [24]: Detector de objetos de una etapa, destacado por su alto rendimiento en detección de objetos. Se ejecutará mediante su implementación en Pytorch en Python que podemos encontrar en su repositorio público
- **YoloV8** [25]: Detector de objetos de última generación de una etapa, destacado por su velocidad de entrenamiento y su alta precisión. Al igual que su hermano, su implementación se basa en PyTorch.
- **Faster R-CNN**: [54] Al contrario que YoloV5, este detector es de dos etapas. Su implementación se realiza mediante el repositorio desarrollado por investigadores de Facebook llamado Detectron2 [11].

Las técnicas escogidas para los experimentos de este trabajo han sido seleccionadas por los motivos descritos a continuación.

YoloV8 ha sido escogido por ser el detector de objetos de una etapa más importante en la actualidad por encima de los demás vistos en este trabajo. Mientras que **Yolov5** ha sido principalmente escogido por realizar una comparación de rendimiento entre este y su hermano más actualizado. Por otro lado, **FasterRCNN** ha sido seleccionado por ser el detector de objetos de dos etapas más usado.

4.2. Preparación del entorno

4.2.1. Instalación del software del sistema operativo principal

Hay que comenzar instalando el sistema operativo necesario en nuestra máquina, es decir, en este caso **Ubuntu 22.04**.

Una vez instalado el sistema operativo principal hay que instalar los **drivers CUDA de Nvidia**, esto es **importante** debido a que si no instalamos los drivers de la tarjeta gráfica, al ejecutar el código de entrenamiento, el programa de Python correrá sobre el procesador, por lo que obtendremos tiempo de entrenamiento eternos.

Posteriormente, hay que instalar **Docker** para poder ejecutar contenedores, estos contenedores son imprescindibles para tener entre otras ventajas, aislamiento del entorno, manejo de dependencias independiente entre contenedores, portabilidad...

Una vez instalado lo anteriormente mencionado, crearemos en el servidor una carpeta que será compartida entre los contenedores. Esta carpeta almacenará los conjuntos de datos necesarios para no tener que estar descargándolos cada vez que ejecutemos el contenedor de Docker.

Además esto facilita la recolección de resultados, debido a que podemos guardar todos los resultados en la carpeta compartida y no tener que estar pasándolos desde el interior del Docker hacia fuera. La crearemos en un directorio permanente, en Ubuntu típicamente es en /home/username, pero en mi caso es en /raid/malr0016. La llamaré datasets.

El siguiente código, nos permite ejecutar uno de los contenedores de esta prueba.

```
1 docker run --gpus all -it -d --shm-size=8gb --env="DISPLAY"
  ↪  --volume="/raid/malr0016/datasets:/tmp/datasets:rw"
  ↪  --name=detectron2 detectron2:v0cat
```

La explicación de los parámetros son los siguientes:

- **–gpus all**: Utiliza todas las GPUs disponibles en el host para el contenedor. Este parámetro es necesario para aplicaciones que requieren aceleración por GPU.
- **-it**: Ejecuta el contenedor en modo interactivo y asigna una terminal para la entrada y salida.

- **-d**: Ejecuta el contenedor en segundo plano (detached mode).
- **–shm-size=8gb**: Asigna 8 GB de espacio para el sistema de memoria compartida del contenedor, lo que puede ser útil para procesos que requieren grandes cantidades de memoria compartida.
- **–env="DISPLAY"**: Pasa la variable de entorno 'DISPLAY' al contenedor, lo que permite que el contenedor acceda a la pantalla del host para aplicaciones gráficas.
- **–volume=/raid/malr0016/datasets:/tmp/datasets:rw"**: Monta el directorio '/raid/malr0016/datasets' del host en '/tmp/datasets' dentro del contenedor con permisos de lectura y escritura.
- **–name=detectron2**: Asigna el nombre 'detectron2' al contenedor para facilitar su identificación y gestión.
- **detectron2:v0cat**: Especifica la imagen del contenedor que se va a ejecutar, en este caso, 'detectron2' con la etiqueta 'v0cat'.

Con esto ya tendríamos el entorno principal configurado, lo siguiente es preparar las imágenes para cada uno de los entornos.

4.2.2. Contenedor para YOLOv5

En mi caso, he decidido usar de base la imagen **nvcr.io/nvidia/tensorflow:20.12-tf2-py3**. He escogido esta porque una de las ventajas que obtenemos a parte de tener los drivers de Nvidia instalados, es tener un entorno configurado para el desarrollo de modelos de aprendiza profundo, algunos ejemplos serían librerías de Python.

Necesitamos tener algunas librerías con versiones específicas instaladas que vienen descritas en los requerimientos de YoloV5, por ello, he creado una imagen personalizada para poner duplicar el entorno de forma rápida. El Dockerfile creado es el siguiente:

```
1 # Utilizar la imagen base de TensorFlow de NVIDIA
2 FROM nvcr.io/nvidia/tensorflow:20.12-tf2-py3
3
4 # Establecer la variable de entorno para evitar problemas interactivos
5 ENV DEBIAN_FRONTEND=noninteractive
6
7 # Instalar las dependencias del sistema
8 RUN apt-get update && \
```

```
9       apt-get install -y git && \
10      apt-get clean && \
11      rm -rf /var/lib/apt/lists/*
12
13  RUN apt-get update && apt-get install -y libgl1-mesa-glx libglib2.0-0 && \
14      rm -rf /var/lib/apt/lists/*
15
16  # Instalar las dependencias de YOLOv5
17  RUN pip install --no-cache-dir \
18      gitpython>=3.1.30 \
19      matplotlib>=3.3 \
20      numpy==1.23.5 \
21      opencv-python>=4.1.1 \
22      pillow>=10.3.0 \
23      psutil \
24      PyYAML>=5.3.1 \
25      requests>=2.32.0 \
26      scipy>=1.4.1 \
27      thop>=0.1.1 \
28      torch>=1.8.0 \
29      torchvision>=0.9.0 \
30      tqdm>=4.64.0 \
31      ultralytics>=8.2.34 \
32      pandas>=1.1.4 \
33      seaborn>=0.11.0 \
34      setuptools>=70.0.0
35
36  # Instalar ONNX, SciPy y Shapely con versiones específicas
37  RUN pip install --no-cache-dir \
38      onnx==1.8.1 \
39      scipy \
40      setuptools==59.5.0 \
41      shapely
42
43  # Ejecutar el script y otros comandos necesarios al iniciar el contenedor
44  CMD ["bash", "-c", "pip install onnx==1.8.1 scipy setuptools==59.5.0 shapely"]
45  CMD ["bash"]
```

Listado 4.1: Contenedor para YoloV5

Para ejecutar esta imagen ejecutamos el siguiente comando que a parte de los parámetros anteriormente mencionados, también consta del parámetro **-p 8888:8888** que redirecciona el puerto 8888 interno del contenedor hacia el 8888 externo, esto lo utilizamos para poder acceder de forma externa a la libreta de Jupyter Notebook:

```
1  docker run --gpus all -it -d --rm --name=yolov5
   ↪  --volume="/raid/malr0016/datasets:/tmp/datasets:rw" -p 8888:8888
   ↪  yolov5
```

En este caso, como la primera que vez que ejecutemos este comando no tendremos la imagen descargada en nuestro sistema local, realizará la descarga desde el repositorio público de Docker.

Una vez creado el contenedor para acceder a él debemos de obtener el id del mismo, esto lo podemos obtener escribiendo en la terminal **docker ps**, obtendremos algo similar a la figura 4.1. Con este identificador, el cual corresponde a la primera columna, podemos acceder al comando con el siguiente comando:

```
docker exec -it 5a9e775c7754 /bin/bash
```

Figura 4.1: Ejecución del comando docker ps

Una vez dentro del contenedor, ejecutaremos los tres siguientes comandos que nos permitirá descargar el repositorio público de YoloV5 e instalar todas las dependencias necesarias de YoloV5.

```
git clone https://github.com/ultralytics/yolov5.git
cd yolov5
pip install -r requirements.txt
```

Una vez ejecutado, ya tendremos el **modelo preparado** para correr, únicamente habría que llamar a las líneas de código necesarias, pero esto, lo veremos más adelante.

4.2.3. Contenedor para YOLOv8

Para la configuración de YOLOv8, he decidido usar la imagen base **ultralytics/ultralytics**. Esta imagen está optimizada para el desarrollo y la ejecución de modelos YOLO de última generación, ya que incluye todas las dependencias necesarias, como PyTorch, CUDA y otros componentes esenciales para el aprendizaje profundo, lo que simplifica enormemente el proceso de instalación y configuración.

Para ejecutar esta imagen, utilizamos el siguiente comando.

```
docker run --gpus all -it -d --shm-size=16gb --env="DISPLAY"
  --volume="/raid/malr0016/datasets:/tmp/datasets:rw" --name=yolov8
  ultralytics/ultralytics:latest
```

Si es la primera vez que ejecutamos este comando, la imagen se descargará automáticamente desde el repositorio público de Docker, ya que no estará presente en nuestro sistema local.

Una vez que el contenedor ha sido creado y está en ejecución, para acceder a él, debemos obtener el ID del contenedor. Esto se puede hacer ejecutando el comando **docker ps** en la terminal, al igual que en apartado anterior. Con el identificador del contenedor, que se encuentra en la primera columna, podemos acceder al contenedor utilizando el siguiente comando:

```
docker exec -it <ID_del_contenedor> /bin/bash
```

Dentro del contenedor, ya tendremos todas las dependencias necesarias para YOLOv8 instaladas. Para comenzar a usar YOLOv8, simplemente navegamos al directorio correspondiente y ejecutamos los scripts o comandos necesarios para el entrenamiento o la inferencia con el modelo YOLOv8:

```
yolo train data=/tmp/datasets/aircrafts/aircrafts.yaml model=yolov8s.pt
  epochs=1 imgsz=512 batch=16
```

El comando anterior es un ejemplo de cómo iniciar el entrenamiento de un modelo YOLOv8 utilizando el modelo small. Una vez ejecutado, ya tendremos el **modelo entrenado y listo para la inferencia o evaluación**, y podremos proceder con el uso o análisis de los resultados obtenidos.

4.2.4. Contenedor para Detectron2

Detectron2 es una biblioteca de código abierto desarrollada por Facebook AI Research (FAIR) para la detección de objetos, segmentación de instancias y otras tareas de visión por computadora. Es una implementación avanzada en PyTorch que ofrece

modelos preentrenados y una arquitectura flexible para la investigación y el desarrollo en visión por computadora. Detectron2 es ampliamente utilizado en aplicaciones que requieren análisis de imágenes y videos.

El contenedor utilizado es la siguiente imagen que hay que construir:

```
FROM nvidia/cuda:11.1.1-cudnn8-devel-ubuntu20.04

ENV DEBIAN_FRONTEND noninteractive
RUN apt-get update && apt-get install -y \
    python3-opencv ca-certificates python3-dev git wget sudo ninja-build
RUN ln -sv /usr/bin/python3 /usr/bin/python

# create a non-root user
ARG USER_ID=1000
RUN useradd -m --no-log-init --system  --uid ${USER_ID} appuser -g sudo
RUN echo '%sudo ALL=(ALL) NOPASSWD:ALL' >> /etc/sudoers
USER appuser
WORKDIR /home/appuser

ENV PATH="/home/appuser/.local/bin:${PATH}"
RUN wget https://bootstrap.pypa.io/pip/3.6/get-pip.py && \
    python3 get-pip.py --user && \
    rm get-pip.py

# install dependencies
# See https://pytorch.org/ for other options if you use a different version of CUDA
RUN pip install --user tensorboard cmake onnx   # cmake from apt-get is too old
RUN pip install --user torch==1.10 torchvision==0.11.1 -f https://download.pytorch.org
    /whl/cu111/torch_stable.html

RUN pip install --user 'git+https://github.com/facebookresearch/fvcore'
# install detectron2
RUN git clone https://github.com/facebookresearch/detectron2 detectron2_repo

ENV FORCE_CUDA="1"

ARG TORCH_CUDA_ARCH_LIST="Kepler;Kepler+Tesla;Maxwell;Maxwell+Tegra;Pascal;Volta;
    Turing"
ENV TORCH_CUDA_ARCH_LIST="${TORCH_CUDA_ARCH_LIST}"

RUN pip install --user -e detectron2_repo

# Set a fixed model cache directory.
ENV FVCORE_CACHE="/tmp"
WORKDIR /home/appuser/detectron2_repo
```

Listado 4.2: Contenedor para Detectron2

Para construir el contenedor de Detectron2, primero necesitamos un archivo llamado Dockerfile que contiene las instrucciones necesarias para crear la imagen. El contenido del Dockerfile es el que se ha mostrado anteriormente.

Una vez que tenemos el Dockerfile listo, podemos construir la imagen con el si-

guiente comando:

```
docker build --build-arg USER_ID=$UID -t detectron2:v0 .
```

Este comando le indica a Docker que **construya una imagen** usando el Dockerfile en el directorio actual, asignándole la etiqueta detectron2:v0.

Después de que la imagen se haya construido correctamente, podemos ejecutar un contenedor basado en esta imagen utilizando el siguiente comando:

```
docker run --gpus all -it -d --shm-size=8gb --env="DISPLAY"
↪  --volume="/raid/malr0016/datasets:/tmp/datasets:rw"
↪  --name=detectron2 detectron2:v0
```

Por último, para dejar el entorno totalmente listo, y evitar algunos incompatibilidades que han surgido con las dependencias de python, se ejecutan los dos siguientes comandos:

```
./datasets/prepare_for_tests.sh
pip install onnx==1.8.1 scipy setuptools==59.5.0 shapely
```

Una vez hecho esto, ya tenemos el entorno totalmente configurado para poder usar las librerías de Python de Detectron2.

4.3. Preparación de los conjuntos de datos

Antes de poder utilizar las técnicas de detección de objetos, necesitamos tener los conjuntos de datos preparados para alimentarlas. Debido a esto, cada vez que descarguemos un conjunto de imágenes, debemos comprobar entre otras cosas, el formato del fichero de anotaciones que viene junto a él.

En la mayoría de casos, el fichero de anotaciones no viene preparado para alimentar las técnicas, realmente hay un problema grave en este ámbito debido a que no se sigue una estandarización, incluso, algunos conjuntos vienen con formatos muy poco conocidos o propietarios.

Para las técnicas de detección de objetos se usan principalmente dos tipos de fichero de anotaciones, YOLO y COCO, vistos anteriormente. En este apartado vamos a ver de forma más exhaustiva como funciona cada una de estas formas de anotar objetos.

4.3.1. Formatos utilizados

Los dos formatos utilizados en las prácticas son YOLO y COCO, esto debido a que los algoritmos que hemos utilizado, hacen uso de estos formatos. A continuación se entrará en detalles de como funciona internamente cada uno de estos dos.

4.3.1.1. YOLO

El formato **YOLO** (You Only Look Once) es uno de los más populares y utilizados en la detección de objetos en tiempo real. Este formato es conocido por su simplicidad y eficiencia. Las anotaciones en YOLO se almacenan en archivos de texto con extensión .txt, donde cada línea del archivo corresponde a un objeto detectado en la imagen.

Un ejemplo de un fichero sería el siguiente:

```
0 0.13500150937504812 0.25950912028896894 0.0718933447026269 0.13376668159878422
0 0.40166815031602354 0.22204073879947356 0.032174968064660636 0.04651119251638025
0 0.5023944295554807 0.2639103593148315 0.038344637525919434 0.05384648847793086
```

Listado 4.3: Ejemplo de formato YOLO

Cada línea de una anotación YOLO sigue una estructura específica que incluye cinco componentes básicos:

- **Clase del objeto**: Un número entero que representa la categoría del objeto detectado. El índice de la clase suele estar basado en un archivo separado que lista todas las clases posibles.
- **Coordenada x del centro del cuadro delimitador (bounding box)**: Este valor es relativo al ancho de la imagen y está normalizado entre 0 y 1.
- **Coordenada y del centro del cuadro delimitador**: Similar a la coordenada x, pero en relación con la altura de la imagen.
- **Ancho del cuadro delimitador**: También normalizado entre 0 y 1, este valor representa el ancho
- **Altura del cuadro delimitador**: Representa la altura del cuadro delimitador, igualmente normalizada.

Como es obvio, el formato YOLO, es usado por todas las técnicas YOLO, incluyendo todas las versiones.

4.3.1.2. COCO

COCO (Common Objects in Context) es otro formato ampliamente utilizado en la detección de objetos, especialmente en aplicaciones donde la precisión es crítica y se requiere manejar anotaciones complejas. El formato de anotaciones COCO se basa en archivos JSON que contienen una estructura de datos jerárquica.

Las anotaciones en COCO incluyen información no solo sobre los cuadros delimitadores de los objetos, sino también sobre segmentos (máscaras), puntos clave y jerarquías de las clases. Las principales características de un archivo de anotación COCO son:

- **Imagenes** : Una lista de todas las imágenes en el conjunto de datos, donde cada entrada incluye el ID de la imagen, su tamaño y su nombre de archivo.
- **Anotaciones** : Una lista detallada de todos los objetos anotados en las imágenes. Cada anotación incluye el ID de la imagen asociada, la categoría del objeto, el cuadro delimitador (en formato [x, y, width, height]), y, opcionalmente, la máscara del objeto.

- **Categorías** : Una lista de todas las categorías de objetos que se pueden detectar, donde cada categoría tiene un nombre y un ID.
- **Puntos clave y segmentación** : COCO permite la anotación de puntos clave en los objetos (por ejemplo, las posiciones de las articulaciones en seres humanos) y segmentaciones detalladas que describen la forma exacta del objeto en la imagen.

A continuación se muestra un ejemplo simplificado de un archivo de anotaciones COCO, que describe una imagen con un objeto anotado:

```
{
  "images": [
    {
      "id": 1,
      "width": 640,
      "height": 480,
      "file_name": "imagen1.jpg"
    }
  ],
  "annotations": [
    {
      "id": 1,
      "image_id": 1,
      "category_id": 1,
      "bbox": [100, 150, 200, 300],
      "segmentation": [[120, 160, 140, 180, 160, 200, 120, 160]],
      "area": 60000,
      "iscrowd": 0
    }
  ],
  "categories": [
    {
      "id": 1,
      "name": "persona",
      "supercategory": "humano"
    }
  ]
}
```

Listado 4.4: Ejemplo de formato COCO

Analizemos cada uno de los índices del JSON:

- **images**: Contiene la información de la imagen, como su ID, tamaño y nombre de archivo.
- **annotations**: Incluye detalles de los objetos detectados, como el ID de la imagen, la categoría, el cuadro delimitador (`bbox`) y la segmentación.

- **categories**: Lista de las categorías posibles, en este caso, solo una categoría llamada `persona`.

El formato COCO es más complejo que YOLO y está diseñado para tareas donde se requiere una anotación rica y precisa, como la segmentación de instancias, detección de poses humanas, y otras aplicaciones avanzadas en visión por computadora.

4.3.2. Scripts útiles para el procesamiento de conjuntos de datos

En este apartado veremos algunos de los scripts que se utilizan para poder hacer las pruebas entre distintos modelos de detección de objetos, entre ellos podemos encontrar convertidores de formatos, redimensionadores de imágenes a parches que servirán para alimentar los algoritmos o scripts para completar información restante.

4.3.2.1. Conversión de YOLO a COCO

Una herramienta necesaria para trabajar con fichero de anotaciones es poder transformar las anotaciones de un formato a otro para poder alimentar las técnicas de detección de objetos.

A continuación se presenta un código programado en Python para transformar las anotaciones de YOLO a COCO:

```
import os
import json
from PIL import Image

def yolo_to_coco(image_dir, label_dir, output_json_path, categories):
    """
    Args:
    - image_dir (str): Directorio que contiene las imágenes.
    - label_dir (str): Directorio que contiene las anotaciones YOLO.
    - output_json_path (str): Ruta del archivo de salida JSON en formato COCO.
    - categories (list): Lista de categorías en formato COCO.
    """

    # Inicializar estructura COCO
    coco_format = {
        "images": [],
        "annotations": [],
        "categories": categories
    }

    annotation_id = 0
    image_id = 0
```

```
    # Recorrer todas las imágenes en el directorio de imágenes
    for filename in os.listdir(image_dir):
        if filename.endswith('.jpg'):
            # Ruta completa de la imagen
            image_path = os.path.join(image_dir, filename)

            # Obtener el tamaño de la imagen
            with Image.open(image_path) as img:
                width, height = img.size

            # Añadir entrada de la imagen al formato COCO
            image_info = {
                "id": image_id,
                "file_name": filename,
                "width": width,
                "height": height
            }
            coco_format["images"].append(image_info)

            # Nombre base del archivo (sin extensión)
            base_name = os.path.splitext(filename)[0]
            label_path = os.path.join(label_dir, base_name + '.txt')

            # Procesar el archivo de anotación correspondiente si existe
            if os.path.exists(label_path):
                with open(label_path, 'r') as file:
                    lines = file.readlines()

                for line in lines:
                    class_id, x_center, y_center, yolo_width, yolo_height = map(float,
                        line.split())

                    # Convertir coordenadas YOLO a COCO
                    abs_x_center = x_center * width
                    abs_y_center = y_center * height
                    abs_width = yolo_width * width
                    abs_height = yolo_height * height

                    x_min = abs_x_center - abs_width / 2
                    y_min = abs_y_center - abs_height / 2

                    # Crear la anotación en formato COCO
                    annotation_info = {
                        "id": annotation_id,
                        "image_id": image_id,
                        "category_id": int(class_id),
                        "bbox": [x_min, y_min, abs_width, abs_height],
                        "area": abs_width * abs_height,
                        "iscrowd": 0
                    }
                    coco_format["annotations"].append(annotation_info)
                    annotation_id += 1

            # Incrementar el ID de imagen
            image_id += 1

    # Guardar el archivo JSON en formato COCO
    with open(output_json_path, 'w') as json_file:
        json.dump(coco_format, json_file, indent=4)
```

```
81
82      print(f"Conversión completada. Archivo guardado en: {output_json_path}")
83
84  def main():
85      # Directorios de entrada y archivo de salida
86      image_dir = '/tmp/datasets/aircrafts/new/val/images'
87      label_dir = '/tmp/datasets/aircrafts/new/val/labels'
88      output_json_path = '/tmp/datasets/aircrafts/new/val/coco.json'
89
90      # Definir categorías
91      categories = [
92          {"id": 1, "name": "aircraft"}
93      ]
94
95      yolo_to_coco(image_dir, label_dir, output_json_path, categories)
96
97  if __name__ == "__main__":
98      main()
```

Listado 4.5: Conversión de YOLO a COCO

Este script recorre un conjunto de imágenes y sus correspondientes anotaciones en formato YOLO, calcula las coordenadas y dimensiones absolutas de los cuadros delimitadores, y finalmente genera un archivo JSON en formato COCO que contiene las imágenes, anotaciones y categorías especificadas. Esta conversión es esencial para utilizar modelos de detección de objetos que requieren anotaciones en formato COCO, facilitando así la integración con diversas herramientas de visión por computadora.

4.3.2.2. Conversión de COCO a YOLO

Igual que creamos una herramienta para convertir de YOLO a COCO, habrán conjunto de datos que sean COCO y tengamos que pasarlo a YOLO, por ello es útil el siguiente script creado para esta situación.

A continuación se presenta un código en Python para esta funcionalidad:

```
1  import json
2  import os
3  import shutil
4
5  # Función para convertir bbox de COCO a YOLO
6  def convert_bbox_to_yolo(size, bbox):
7      dw = 1.0 / size[0]
8      dh = 1.0 / size[1]
9      x = bbox[0] + bbox[2] / 2.0
10     y = bbox[1] + bbox[3] / 2.0
11     w = bbox[2]
12     h = bbox[3]
```

```
    x = x * dw
    w = w * dw
    y = y * dh
    h = h * dh
    return (x, y, w, h)

with open('labels_coco.json', 'r') as f:
    data = json.load(f)

# Crear carpetas para las imágenes y las etiquetas
os.makedirs('output/images', exist_ok=True)
os.makedirs('output/labels', exist_ok=True)

# Obtener todas las imágenes y anotaciones
images = data['images']
annotations = data['annotations']

# Procesar todas las imágenes
for img in images:
    img_id = img['id']
    file_name = img['file_name']
    width = img['width']
    height = img['height']

    # Copiar imagen al directorio de salida
    shutil.copy(os.path.join('images', file_name), os.path.join('output/images',
        file_name))

    # Crear el archivo de etiquetas YOLO
    label_file_path = os.path.join('output/labels', f"{os.path.splitext(file_name)
        [0]}.txt")
    with open(label_file_path, 'w') as label_file:
        for ann in annotations:
            if ann['image_id'] == img_id:
                category_id = ann['category_id'] - 1  # YOLO usa índices de categoría que
                    empiezan en 0
                bbox = convert_bbox_to_yolo((width, height), ann['bbox'])
                label_file.write(f"{category_id} {bbox[0]} {bbox[1]} {bbox[2]} {bbox
                    [3]}\n")

print("Conversión completada con éxito!")
```

Listado 4.6: Conversión de COCO a YOLO

Este código convierte anotaciones de un formato COCO a YOLO y guarda todas las imágenes y etiquetas en un solo conjunto. Los pasos del algoritmo son los siguientes:

- **Conversión de Bbox**: Se define la función convert_bbox_to_yolo que convierte las coordenadas de los cuadros delimitadores del formato COCO al formato YOLO .

- **Carga de datos**: Se carga un archivo JSON llamado labels_coco.json que contiene las anotaciones en formato COCO.

- **Creación de directorios**: Se crean carpetas output/images y output/labels para almacenar todas las imágenes y sus correspondientes etiquetas en formato YOLO.

- **Procesamiento de imágenes**: Se itera sobre todas las imágenes en los datos cargados. Se copia cada imagen a la carpeta output/images y por último crea un archivo de etiquetas en formato YOLO para cada imagen en output/labels, incluyendo todas las anotaciones correspondientes a esa imagen.

4.3.2.3. División de imágenes de alta resolución en parches

Algunos datasets vienen anotados con imágenes de una resolución muy alta, incluso llegando a tener hasta 4.000 píxeles por lado. Esto es normal debido a la naturaleza de las imágenes, por ello, para alimentar a los algoritmos, es necesario hacer un script que convierta las imágenes en pequeños parches.

Los parches que se usan para alimentar los modelos, suelen tener desde 512 píxeles por lado hasta 640 píxeles por lado, esto debido a que los modelos necesitan de pequeñas imágenes para poder procesarlas, en caso contrario, las tarjetas gráficas no tendrán memoria suficiente para albergar y procesar estas imágenes.

A continuación se presenta un script programado para poder redimensionar las imágenes a parches de 640 píxeles, este script en específico está programado para YOLO.

```
import os
import cv2

def create_fixed_size_patches(image_path, label_path, output_image_dir,
    output_label_dir, patch_size=640, max_no_annotation_ratio=0.25):
    # Leer la imagen
    image = cv2.imread(image_path)
    img_height, img_width, _ = image.shape

    # Leer las anotaciones
    with open(label_path, 'r') as file:
        labels = file.readlines()

    # Calcular el número total de parches
    num_patches_x = (img_width + patch_size - 1) // patch_size
    num_patches_y = (img_height + patch_size - 1) // patch_size
    total_patches = num_patches_x * num_patches_y

    # Contador para mostrar el progreso
    patch_count = 0
    no_annotation_patches = 0

```

```
    # Calcular el número de parches que caben en cada dimensión
    for i in range(0, img_height, patch_size):
        for j in range(0, img_width, patch_size):
            # Definir las coordenadas del parche
            x_min = j
            y_min = i
            x_max = min(j + patch_size, img_width)
            y_max = min(i + patch_size, img_height)

            # Recortar la imagen
            patch = image[y_min:y_max, x_min:x_max]

            # Solo guardar parches que tengan el tamaño deseado
            if patch.shape[0] != patch_size or patch.shape[1] != patch_size:
                continue

            # Definir nombre del parche
            base_name = os.path.splitext(os.path.basename(image_path))[0]
            patch_filename = f"{base_name}_{i}_{j}"
            patch_image_path = os.path.join(output_image_dir, patch_filename + ".jpg")
            patch_label_path = os.path.join(output_label_dir, patch_filename + ".txt")

            # Ajustar las etiquetas y guardarlas
            patch_labels = []
            for label in labels:
                class_id, x_center, y_center, width, height = map(float, label.split()
                    )

                # Convertir las coordenadas relativas al espacio absoluto de la imagen original
                abs_x_center = x_center * img_width
                abs_y_center = y_center * img_height
                abs_width = width * img_width
                abs_height = height * img_height

                # Calcular los límites absolutos del bounding box
                abs_x_min = abs_x_center - abs_width / 2
                abs_x_max = abs_x_center + abs_width / 2
                abs_y_min = abs_y_center - abs_height / 2
                abs_y_max = abs_y_center + abs_height / 2

                # Verificar si cualquier parte del bounding box está dentro del parche actual
                if (abs_x_max > x_min) and (abs_x_min < x_max) and (abs_y_max > y_min)
                     and (abs_y_min < y_max):
                    # Recortar las coordenadas del bounding box al tamaño del parche
                    clipped_x_min = max(abs_x_min, x_min)
                    clipped_x_max = min(abs_x_max, x_max)
                    clipped_y_min = max(abs_y_min, y_min)
                    clipped_y_max = min(abs_y_max, y_max)

                    # Calcular el nuevo centro, ancho y alto del bounding box recortado
                    clipped_width = clipped_x_max - clipped_x_min
                    clipped_height = clipped_y_max - clipped_y_min
                    clipped_x_center = (clipped_x_min + clipped_x_max) / 2
                    clipped_y_center = (clipped_y_min + clipped_y_max) / 2

                    # Convertir coordenadas absolutas al espacio relativo del parche
                    rel_x_center = (clipped_x_center - x_min) / patch_size
                    rel_y_center = (clipped_y_center - y_min) / patch_size
                    rel_width = clipped_width / patch_size
```

```
79                      rel_height = clipped_height / patch_size
80
81                      # Calcular el área del objeto dentro del parche en comparación con su área
                            original
82                      if (abs_width * abs_height) > 0:
83                          area_in_patch = (clipped_width * clipped_height) / (abs_width
                                * abs_height)
84
85                          if area_in_patch >= 0.25:
86                              patch_labels.append(f"{class_id} {rel_x_center} {
                                    rel_y_center} {rel_width} {rel_height}")
87
88              # Si no hay etiquetas y la proporción de imágenes sin anotaciones está por encima del
                    umbral, no guardar el parche
89              if not patch_labels:
90                  no_annotation_patches += 1
91                  if no_annotation_patches > max_no_annotation_ratio * total_patches:
92                      continue
93
94              # Guardar el parche de imagen
95              cv2.imwrite(patch_image_path, patch)
96
97              # Guardar las etiquetas del parche si hay alguna
98              if patch_labels:
99                  with open(patch_label_path, 'w') as f:
100                     f.write("\n".join(patch_labels))
101
102             # Actualizar y mostrar el progreso
103             patch_count += 1
104             print(f"Procesando parche {patch_count} de {total_patches} para {os.path.
                    basename(image_path)}")
105
106 def main():
107     # Directorio de entrada y salida
108     input_dir = '/tmp/datasets/aircrafts/test'  # Cambia esto a tu directorio de imágenes y
            etiquetas
109     output_image_dir = '/tmp/datasets/aircrafts/new2/test/images'  # Directorio donde se
            guardarán las imágenes de salida
110     output_label_dir = '/tmp/datasets/aircrafts/new2/test/labels'  # Directorio donde se
            guardarán las etiquetas de salida
111
112     # Crear los directorios de salida si no existen
113     os.makedirs(output_image_dir, exist_ok=True)
114     os.makedirs(output_label_dir, exist_ok=True)
115
116     # Recorrer todas las imágenes y etiquetas
117     for filename in os.listdir(input_dir):
118         if filename.endswith('.jpg'):
119             image_path = os.path.join(input_dir, filename)
120             label_path = os.path.join(input_dir, os.path.splitext(filename)[0] + '.txt
                    ')
121             if os.path.exists(label_path):
122                 create_fixed_size_patches(image_path, label_path, output_image_dir,
                        output_label_dir)
123
124     input_dir = '/tmp/datasets/aircrafts/train'  # Cambia esto a tu directorio de imágenes y
            etiquetas
125     output_image_dir = '/tmp/datasets/aircrafts/new2/train/images'  # Directorio donde se
            guardarán las imágenes de salida
126     output_label_dir = '/tmp/datasets/aircrafts/new2/train/labels'  # Directorio donde se
            guardarán las etiquetas de salida
127
128     # Crear los directorios de salida si no existen
```

```
    os.makedirs(output_image_dir, exist_ok=True)
    os.makedirs(output_label_dir, exist_ok=True)

    # Recorrer todas las imágenes y etiquetas
    for filename in os.listdir(input_dir):
        if filename.endswith('.jpg'):
            image_path = os.path.join(input_dir, filename)
            label_path = os.path.join(input_dir, os.path.splitext(filename)[0] + '.txt')
            if os.path.exists(label_path):
                create_fixed_size_patches(image_path, label_path, output_image_dir, output_label_dir)

    input_dir = '/tmp/datasets/aircrafts/val'  # Cambia esto a tu directorio de imágenes y etiquetas
    output_image_dir = '/tmp/datasets/aircrafts/new2/val/images'  # Directorio donde se guardarán las imágenes de salida
    output_label_dir = '/tmp/datasets/aircrafts/new2/val/labels'  # Directorio donde se guardarán las etiquetas de salida

    # Crear los directorios de salida si no existen
    os.makedirs(output_image_dir, exist_ok=True)
    os.makedirs(output_label_dir, exist_ok=True)

    # Recorrer todas las imágenes y etiquetas
    for filename in os.listdir(input_dir):
        if filename.endswith('.jpg'):
            image_path = os.path.join(input_dir, filename)
            label_path = os.path.join(input_dir, os.path.splitext(filename)[0] + '.txt')
            if os.path.exists(label_path):
                create_fixed_size_patches(image_path, label_path, output_image_dir, output_label_dir)

    # Repetir para otros conjuntos de datos como test y val...

if __name__ == "__main__":
    main()
```

Listado 4.7: Redimensionador de imágenes

Algunas de las características más importantes de este script, es que además de realizar el trabajo anteriormente mencionado, intenta **mantener una relación de parches** entre los que tienen objetos y los que no tienen en torno a un 75 % y 25 % respectivamente.

Además, solo anota los objetos que se encuentran en la imagen en al menos un 25 %, esto lo he realizado para que haya suficiente información característica de los objetos.

Este script está diseñado para permitir la selección de la resolución de los parches mediante la modificación de una variable, lo que facilita su ajuste a los requisitos específicos del modelo.

Se presenta a continuación una previsualización en la figura 4.2 de algunos de los parches generados en un conjunto de datos utilizado más adelante.

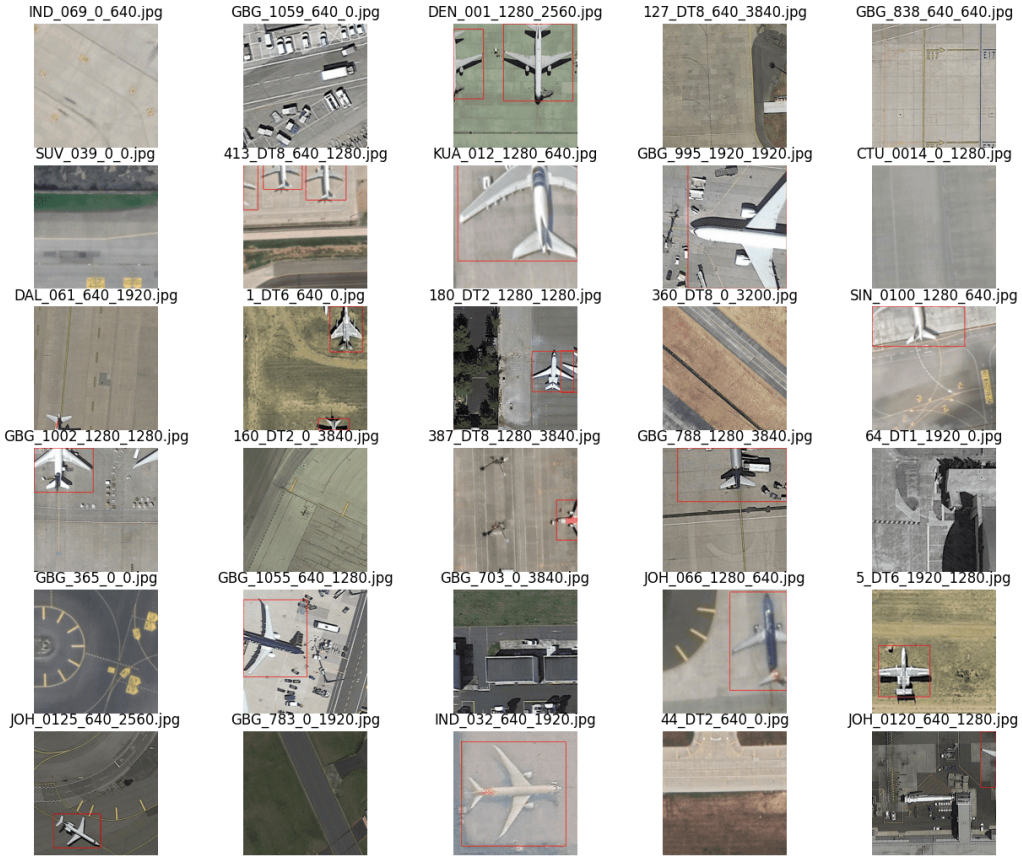

Figura 4.2: Parches del conjunto de datos HRPlanesv2 [50]

4.4. Conjunto de datos utilizados y su procesamiento

En este apartado, se explicará como se han preparado los conjuntos de datos para la experimentación con las diferentes técnicas de detección de objetos.

4.4.1. HRPlanesv2

Este conjunto llamado **HRPlanesv2**, publicado el 1 de febrero de 2022, identifica mediante imágenes recolectadas de Google Maps una gran cantidad de imágenes que contienen aviones de diferentes tipos, desde aviones comerciales como el Boeing 747 hasta avionetas Cesna, pasando por aviones militares de combate.

Las imágenes de este conjunto de datos tienen una resolución de 4800 píxeles por 2703 píxeles. Está conformado por un total de 14.335 imágenes.

El conjunto de datos en principio describía que existen 3 etiquetas, aviones comerciales, civiles y militares, pero el conjunto final solo están etiquetados todos los aviones como un tipo único.

En la figura 4.3 podemos ver un ejemplo de las imágenes del conjunto de datos.

Figura 4.3: Imagen extraída del conjunto de datos HRPlanesv2 [50]

El procesamiento necesario que se ha realizado en este conjunto de datos es aplicar un script de redimensión en pequeños parches de 640 píxeles como el visto en

la sección anterior. En la figura 4.4 podemos observar el resultado de uno de estos parches.

Figura 4.4: Imagen parcheada extraída del conjunto de datos HRPlanesv2 [50]

Además también se ha aplicado una transformación del formato YOLO A COCO para poder ejecutar este conjunto en las redes convolucionales de Detectron2.

4.4.2. Oil Storage Tanks

El dataset **Oil Storage Tanks** fue publicado en noviembre de 2019. Este dataset fue creado para la investigación y el desarrollo de modelos de detección de objetos, enfocándose específicamente en la identificación de tanques de almacenamiento de petróleo en imágenes de alta resolución.

El proceso de anotación original del dataset categoriza todos los tanques como una única clase denominada "tanque de almacenamiento de petróleo". Esta consistencia en la clasificación facilita su uso para la formación de modelos de detección de objetos específicos de dominio.

En la figura 4.5, se muestra un ejemplo de las imágenes contenidas en este conjunto de datos.

En este caso, el conjunto de datos venía en formato COCO y además venía dividido directamente en parches, por lo que lo único que hacía que hacer en este caso es pasarlo a YOLO, esto lo vimos en el apartado anterior.

Además de esto, había que hacer un **split randomizado** para dividir el conjunto de

Figura 4.5: Imagen extraída del conjunto de datos Oil Storage Tanks

datos de forma aleatoria entre 70 % el conjunto de datos de entrenamiento, 15 % el de validación y 15 % el de testeo.

Un parche extraído del conjunto de datos lo podemos contemplar en la figura 4.6

4.4.3. DIOR

El conjunto de datos **DIOR** (Dataset for Object dectIon in Optical Remote sensing images) fue publicado en 2019. Este conjunto de datos es diverso y complejo, ya que incluye imágenes de diferentes resoluciones espaciales y condiciones ambientales.

En la figura 4.7 podemos ver una distribución de las clases de este conjunto de datos. Vemos resaltado que hay una gran cantidad de barcos, coches y tanques de almacenamiento, aunque hay 20 distintos tipos de categorías.

Las imágenes vienen en parches de 800 píxeles por 800 píxeles, además vienen anotadas en XML, por lo que hay que hacer una conversión de XML a YOLO y posteriormente a COCO.

Como realizamos en el anterior conjunto de datos, hay que realizar un **split rando-**

Figura 4.6: Imagen parcheada extraída del conjunto de datos Oil Storage Tanks

mizado para dividir el conjunto de datos de igual forma que en el conjunto anterior.

Por último en la figura 4.8 podemos ver una imagen extraída del conjunto de datos DIOR.

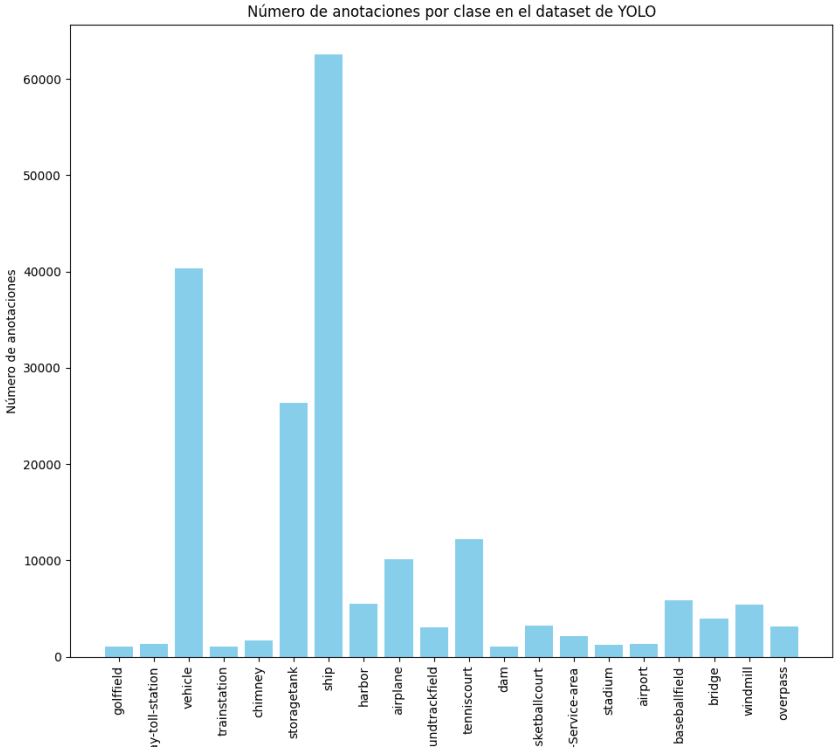

Figura 4.7: Distribución de las clases en el conjunto de datos DIOR (Fuente propia)

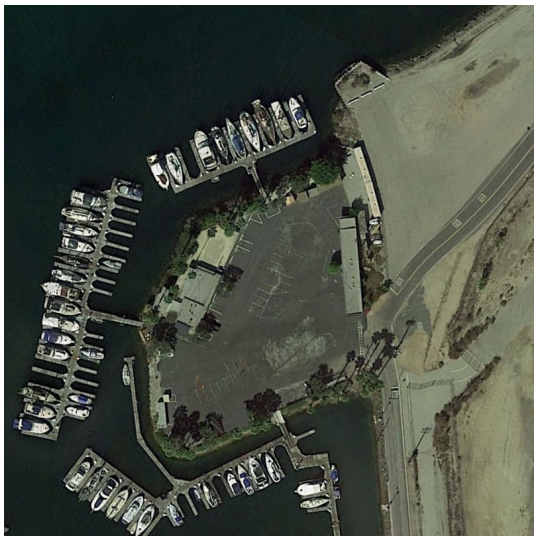

Figura 4.8: Imagen extraída del conjunto de datos DIOR

4.5. Métricas

En este capítulo, se presentan los resultados obtenidos a partir de la experimentación con los modelos YOLOv5 y YOLOv8 en diferentes conjuntos de datos. La precisión de los modelos se evaluó utilizando métricas comunes en la detección de objetos, como mAP50 y mAP50-95, y se llevaron a cabo múltiples experimentos variando el número de épocas para entender cómo afecta el entrenamiento a la capacidad de detección del modelo.

4.5.1. Métricas de evaluación

Para evaluar el rendimiento de los modelos de detección de objetos, se utilizan las métricas **mAP50**, **mAP50-95** y F1 Score, se presentan junto a las fórmulas utilizadas para calcular cada una de estas métricas:

- **mAP50** (Mean Average Precision at 50 % IoU): Es la precisión media calculada cuando el umbral de la Intersección sobre la Unión (IoU, por sus siglas en inglés) es del 50 %. Esto significa que se considera una detección como correcta si el área de intersección entre la predicción del modelo y el objeto real cubre al menos el 50 % del área combinada de ambos.

$$\text{mAP@50} = \frac{1}{N} \sum_{i=1}^{N} \text{AP}_{50}(i)$$

  - N: El número total de clases o categorías en el dataset.
  - $\text{AP}_{50}(i)$: La precisión promedio (*Average Precision*) de la clase i calculada con un umbral de *IoU* (Intersection over Union) del 50 %.

- **mAP50-95**: Es la media de las precisiones promedio (AP) calculadas en múltiples umbrales de IoU, desde el 50 % hasta el 95 %, con incrementos de 5 %. Esta métrica proporciona una evaluación más completa de la precisión del modelo, ya que tiene en cuenta cómo varía la precisión a diferentes niveles de exigencia en la superposición de las predicciones con los objetos reales.

$$\text{mAP@50-95} = \frac{1}{N} \sum_{i=1}^{N} \left(\frac{1}{T} \sum_{t=50}^{95} \text{AP}_t(i) \right)$$

  - N: El número total de clases o categorías en el dataset.

- T: El número total de umbrales de *IoU* considerados, que suelen ser desde 50 % hasta 95 %, aumentando en incrementos de 5 %.
- $\text{AP}_t(i)$: La precisión promedio para la clase i calculada con un umbral de *IoU* del $t\,\%$, donde t varía entre 50 y 95.

- **F1 Score:** Es una métrica utilizada en el ámbito del aprendizaje automático y la estadística para evaluar el rendimiento de un modelo de clasificación. Es especialmente útil cuando las clases están desbalanceadas (es decir, cuando el número de ejemplos de una clase es mucho mayor que el de otra clase).

$$F_1 = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

- $\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}}$
- $\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}}$

4.6. Configuración del entrenamiento de los modelos

4.6.1. YOLOv5

Para ejecutar los entrenamientos en YOLOv5, hace falta descargar el repositorio, como anteriormente se menciona, una vez dentro del repositorio se podría ejecutar la siguiente línea bash:

```
python train.py --img 512 --batch 16 --epochs 50 --data
↪ /tmp/datasets/oiltanks/yolo/oiltank.yaml --weights yolov5x.pt
↪ --device 0
```

A continuación se detallan los hiperparámetros utilizados en este comando:

- **–img 512**: Este hiperparámetros especifica el tamaño de las imágenes de entrada al modelo. En este caso, las imágenes se redimensionan a 512x512 píxeles antes de ser procesadas, es recomendable que tengan el tamaño nativo, por ello es importante separar en patches los conjuntos de datos.

- **–batch 16**: Define el tamaño del lote (batch size) durante el entrenamiento. En este caso, se procesan 16 imágenes en paralelo en cada iteración del entrenamiento. El tamaño del lote afecta la memoria requerida por la GPU y puede influir en la estabilidad y la velocidad del entrenamiento.
- **–epochs 50**: Indica el número total de épocas o pasadas completas a través del conjunto de datos durante el entrenamiento. Un mayor número de épocas permite al modelo aprender más de los datos, pero también aumenta el riesgo de sobreajuste.
- **–data /tmp/datasets/oiltanks/yolo/oiltank.yaml**: Especifica la ruta al archivo de configuración del conjunto de datos. Este archivo YAML contiene la información sobre las rutas de los datos de entrenamiento y validación, así como las clases y otras configuraciones relacionadas con el conjunto de datos.
- **–weights yolov5x.pt**: Indica la ruta al archivo de pesos preentrenados del modelo YOLOv5.
- **–device 0**: Especifica el dispositivo en el que se debe realizar el entrenamiento. En este caso, el número 0 se refiere a la primera GPU disponible en el sistema.

Al final del entrenamiento se crean varias carpetas y archivos en el directorio de salida especificado (o en el directorio por defecto si no se ha especificado). A continuación, se detalla la estructura de directorios y los archivos que puedes encontrar:

- **output**/: Este es el directorio principal donde se guardan todos los resultados del entrenamiento. Si no se especifica un directorio de salida con la opción `-project`, YOLOv5 creará una carpeta llamada `runs/train/exp` por defecto.
  - **weights**/: Dentro del directorio `output/`, se crea una subcarpeta llamada `weights/` donde se almacenan los archivos de pesos del modelo guardados durante el entrenamiento. Los archivos de pesos suelen incluir:
    - `best.pt`: El archivo de pesos del modelo que obtuvo el mejor rendimiento en el conjunto de validación, basado en la métrica de evaluación especificada (como la precisión media promedio).
    - `last.pt`: El archivo de pesos del modelo al final de la última época de entrenamiento.
  - **results.txt**: Este archivo contiene un resumen de los resultados del entrenamiento. Incluye métricas de rendimiento como la pérdida total (loss), la precisión media promedio (mAP), y otros indicadores clave de rendimiento para cada época.

  - **events**/: Este directorio contiene archivos relacionados con el registro de eventos del entrenamiento, como los registros generados por TensorBoard. Permite realizar un seguimiento visual del entrenamiento a través de TensorBoard.
  - **labels**/: Si se realiza la conversión de etiquetas durante el entrenamiento, las etiquetas generadas para los datos de validación y entrenamiento se almacenan en esta carpeta. Estas etiquetas suelen estar en el formato YOLO, con un archivo por imagen.
  - ***.jpg y *.png**: Durante el entrenamiento, YOLOv5 puede generar imágenes de ejemplos y visualizaciones, como imágenes de detección con anotaciones. Estas imágenes suelen estar guardadas en formato JPG o PNG.

- **runs/train/exp**/: Si no se especifica una carpeta de salida personalizada, YOLOv5 crea un directorio llamado `exp/` dentro de `runs/train/`, donde `exp` es un índice que se incrementa con cada ejecución de entrenamiento. Este directorio contiene subdirectorios y archivos similares a los descritos anteriormente.

En resumen, la estructura de directorios y archivos generada por YOLOv5 durante el entrenamiento incluye carpetas para pesos del modelo, resultados, registros de eventos y etiquetas, así como archivos para la visualización de los resultados. Estos archivos permiten analizar el rendimiento del modelo y revisar el progreso del entrenamiento.

4.6.2. YOLOv8

El comando utilizado para entrenar el modelo YOLOv8 es muy similar al de YoloV5 y se usan exáctamente los mismo parámetros pero escritos de otra manera. Un ejemplo del comando es el siguiente:

```
yolo train data=/tmp/datasets/aircrafts/aircrafts.yaml model=yolov8s.pt
↪  epochs=1 imgsz=512 batch=16 device=0,1,2,3,4,5,6,7
```

A continuación se detallan los parámetros utilizados en este comando:

- **data=/tmp/datasets/aircrafts/aircrafts.yaml**: Es el equivalente en YoloV5 a –data

- **model=yolov8s.pt**: Es el equivalente en YoloV5 a –weights
- **epochs=1**: Es el equivalente en YoloV5 a –epochs
- **imgsz=512**: Es el equivalente en YoloV5 a –img
- **batch=16**: Es el equivalente en YoloV5 a –batch
- **device=0,1,2,3,4,5,6,7**: Es el equivalente en YoloV5 a –device, en este caso el entrenamiento se ejecutará en las 8 GPU a la vez.

Las carpetas de salida son exactamente iguales que las de YoloV5.

4.6.3. FasterRCNN

Para entrenar un modelo con esta técnica, se ha tenido que programar un script en Python, ya que para usarla se utiliza mediante librerías escritas en dicho lenguaje.

Para ello, se ha escrito el siguiente código que se encarga tanto de entrenar como de evaluar el modelo, además se ha programado para que se lance con las 8 gráficas dedicadas disponibles en el servidor.

A continuación, se explica el código paso a paso, destacando las partes clave y los parámetros importantes:

1. **Registro de Datasets:**
   - Se utilizan las funciones de Detectron2 para registrar los conjuntos de datos en formato COCO. Esto incluye los datasets de entrenamiento, validación y prueba.
   - El comando `register_coco_instances` toma el nombre del dataset, la ruta al archivo JSON de anotaciones en formato COCO y la ruta a las imágenes correspondientes.

```
1  register_coco_instances("dior_train", {},
   ↪ f"{dataset_path}/train/coco.json",
   ↪ f"{dataset_path}/train/images")
2  register_coco_instances("dior_val", {},
   ↪ f"{dataset_path}/val/coco.json",
   ↪ f"{dataset_path}/val/images")
```

```
register_coco_instances("dior_test", {},
↪ f"{dataset_path}/test/coco.json",
↪ f"{dataset_path}/test/images")
```

2. **Configuración del Modelo (`setup_cfg`):**

   - En esta función, se configura el modelo y los parámetros de entrenamiento. Algunos de los aspectos clave incluyen:
     - **Archivo de configuración del modelo:** Se carga el archivo de configuración de Faster R-CNN predefinido. En este caso, se utiliza la configuración para Faster R-CNN con una ResNet-50 como backbone y Feature Pyramid Network (FPN).
     - **Dataset:** Se especifican los datasets para el entrenamiento y la validación mediante `cfg.DATASETS.TRAIN` y `cfg.DATASETS.TEST`.
     - **Tamaño del lote:** El tamaño del lote por GPU se ajusta con otra variable llamada `cfg.SOLVER.IMS_PER_BATCH`.
     - **Learning Rate:** Se define la tasa de aprendizaje inicial mediante la variable de configuración `cfg.SOLVER.BASE_LR`.
     - **Número de Iteraciones:** El número de iteraciones se calcula en función del tamaño del conjunto de datos y el número total de imágenes procesadas por iteración.
     - **Número de clases:** Se establece el número de clases del modelo con `cfg.MODEL.ROI_HEADS.NUM_CLASSES`. En este caso, el modelo está configurado para detectar 20 clases.
     - **Directorio de salida:** Se especifica la carpeta donde se guardarán los resultados y modelos entrenados.

```
def setup_cfg(num_epochs, ims_per_batch):
    cfg = get_cfg()
    cfg.merge_from_file(model_zoo.get_config_file("COCO-
    Detection/faster_rcnn_R_50_FPN_3x.yaml"))
    cfg.DATASETS.TRAIN = ("dior_train",)
    cfg.DATASETS.TEST = ("dior_val",)
    cfg.DATALOADER.NUM_WORKERS = 40
    cfg.SOLVER.IMS_PER_BATCH = ims_per_batch
    cfg.SOLVER.BASE_LR = 0.002
    num_images = 16424
    total_batch_size = ims_per_batch * 8
    iterations_per_epoch = num_images / total_batch_size
```

```
        cfg.SOLVER.MAX_ITER = int(iterations_per_epoch *
        ↪ num_epochs)
        cfg.SOLVER.STEPS = []
        cfg.MODEL.ROI_HEADS.BATCH_SIZE_PER_IMAGE = 512
        cfg.MODEL.ROI_HEADS.NUM_CLASSES = 20
        cfg.TEST.EVAL_PERIOD = int(iterations_per_epoch * 50)
        cfg.OUTPUT_DIR = "./output"
        os.makedirs(cfg.OUTPUT_DIR, exist_ok=True)
        return cfg
```

3. **Clase personalizada para el entrenamiento (`MyTrainer`):**

   - Se define una clase personalizada `MyTrainer`, esta clase extiende la clase `DefaultTrainer` de Detectron2. En esta clase, se implementa un evaluador basado en el protocolo COCO para evaluar el rendimiento del modelo durante el entrenamiento.
   - La función `build_evaluator` construye el evaluador COCO, mientras que la función `test` realiza la evaluación utilizando los evaluadores configurados.

```
class MyTrainer(DefaultTrainer):
    @classmethod
    def build_evaluator(cls, cfg, dataset_name):
        return COCOEvaluator(dataset_name, cfg, True,
        ↪ output_dir=cfg.OUTPUT_DIR)
    @classmethod
    def test(cls, cfg, model, evaluators=None):
        if evaluators is None:
            evaluators = [cls.build_evaluator(cfg, name) for
            ↪ name in cfg.DATASETS.TEST]
        return super().test(cfg, model, evaluators)
```

4. **Entrenamiento con múltiples GPUs (`main`):**

   - La función `main` configura y lanza el entrenamiento. Si se especifica la opción de evaluación (`-eval_only`), el modelo se carga y se evalúa en el conjunto de validación.
   - En caso contrario, se crea una instancia de `MyTrainer` y se inicia el proceso de entrenamiento.

- El script permite realizar el entrenamiento en múltiples GPUs mediante la función `launch`, que gestiona la distribución del entrenamiento en todas las GPUs disponibles.

```
def main(args):
    cfg = setup_cfg(args.epochs, args.ims_per_batch)
    if args.eval_only:
        model = MyTrainer.build_model(cfg)
        model.load_state_dict(torch.load(cfg.MODEL.WEIGHTS))
        res = MyTrainer.test(cfg, model)
        return res
    trainer = MyTrainer(cfg)
    trainer.resume_or_load(resume=False)
    return trainer.train()

if __name__ == "__main__":
    parser = default_argument_parser()
    parser.add_argument("--epochs", type=int, default=50,
    ↪  help="Número de épocas para entrenar")
    parser.add_argument("--ims_per_batch", type=int, default=4,
    ↪  help="Número de imágenes por lote por GPU")
    args = parser.parse_args()
    launch(
        main,
        args.num_gpus,
        num_machines=1,
        machine_rank=0,
        dist_url="tcp://127.0.0.1:{}".format(29500),
        args=(args,),
    )
```

4.6.4. Hiperparámetros modificados

Para cada modelo, se han probado diferentes configuraciones de hiperparámetros. En cada uno de los experimentos, se modificaron los hiperparámetros más significativos para observar su impacto en el rendimiento del modelo.

4.6.4.1. Hiperparámetros de YOLOv5 y YOLOv8

En los experimentos realizados, se modificaron dos hiperparámetros principales: el número de **épocas** y la elección del **modelo** (versiones *s* y *x*). A continuación, se detalla qué representa cada hiperparámetros y cómo varía para los modelos YOLOv5 y YOLOv8:

- **Épocas**: El número de épocas se refiere a la cantidad de veces que el modelo completa una pasada por todo el conjunto de datos de entrenamiento. Un mayor número de épocas permite al modelo aprender más de los datos, pero también aumenta el riesgo de sobreajuste (*overfitting*). En nuestros experimentos, se utilizaron 10 y 50 épocas para evaluar cómo afecta el número de iteraciones al rendimiento del modelo.
- **Modelos s y x**:
  - **Modelo *s* (small)**: Esta versión es una configuración más ligera del modelo, con menos capas e hiperparámetros. Está diseñada para ser más rápida y menos exigente en términos de recursos computacionales, lo que la hace adecuada para dispositivos con menos capacidad de procesamiento o para aplicaciones donde la velocidad es crucial. Tanto en YOLOv5 como en YOLOv8, el modelo *s* ofrece una solución equilibrada entre precisión y velocidad.
  - **Modelo *x* (extra-large)**: Esta versión es la más grande y compleja del modelo, con más capas e hiperparámetros. Está diseñada para maximizar la precisión de la detección, aprovechando arquitecturas más profundas y sofisticadas. El modelo *x* es más adecuado para aplicaciones donde se dispone de recursos computacionales significativos y donde la precisión es más importante que la velocidad de inferencia. En ambos YOLOv5 y YOLOv8, el modelo *x* tiende a ofrecer mejores resultados en términos de mAP, especialmente en tareas complejas.

En resumen, al cambiar las **épocas**, se estudia el impacto del tiempo de entrenamiento en la capacidad del modelo para generalizar. Al variar entre los modelos **s** y **x**, se analiza cómo la arquitectura del modelo afecta el equilibrio entre velocidad y precisión.

4.6.4.2. Hiperparámetros de Faster R-CNN en Detectron2

En los experimentos realizados con el modelo Faster R-CNN en Detectron2, se modificaron dos hiperparámetros principales: el número de **épocas** y el **tamaño del lote** (batch size). A continuación, se detalla qué representa cada hiperparámetro y cómo varía en el contexto de Faster R-CNN:

- **Épocas**: El número de épocas se refiere a la cantidad de veces que el modelo completa una pasada por todo el conjunto de datos de entrenamiento, explicado anteriormente.
- **Tamaño del lote (Batch Size)**:
  - **Tamaño del lote por GPU**: Este hiperparámetros indica cuántas imágenes se procesan en paralelo en cada GPU durante una iteración del entrenamiento. En los experimentos, se ajustó el tamaño del lote para encontrar un equilibrio entre la utilización eficiente de la memoria de la GPU y la estabilidad del entrenamiento.
  - **Tamaño total del lote**: Es el número total de imágenes procesadas en una iteración, que se calcula multiplicando el tamaño del lote por GPU por el número de GPUs.

En FasterRCNN, el único hiperparámetro cambiado para realizar los pruebas en cada conjunto de datos son las **épocas**. El tamaño del lote es el mismo siempre para un determinado conjunto de datos. Este se modifica dependiendo del conjunto de datos debido a que si es demasiado elevado pueden sobrecargarse las memorias de las GPUs, por lo que hay que afinar este parámetro intentando poner el mayor número posible que pueda soportar las gráficas.

4.7. Resultados

En este apartado se detallan los resultados obtenidos con las técnicas utilizadas para los distintos conjuntos de datos. Se han obtenido usando exáctamente los mismos conjuntos de datos con las mismas imágenes para todos los entrenamientos.

Para cada modelo se han variado los parámetros anteriormente descritos, en caso de YOLO el número de épocas y el modelo en concreto a utilizar y en caso de FasterRCNN se ha usado una configuración de 300 épocas y otra de 150.

En caso de las pruebas de YOLO, para cada conjunto de datos se ha realizado 8 pruebas distintas, es decir, se ha llevado a cabo **24** entrenamientos diferentes en total, es decir, 8 para cada conjunto de datos.

Se ha trabajado con YOLO de forma más extensa en este trabajo debido a diferentes factores entre los que incluye la constante evolución y desarrollo de esta técnica por parte de sus contribuidores.

En total, entre FasterRCNN y YOLO, se han llevado a cabo **30 entrenamientos diferentes**.

Se ha trabajado menos en FasterRCNN debido a que en las pruebas prelimilinares se observó una tasa de acierto bastante inferior a la de YOLO por lo que decidí enfocarme más en la técnica de Ultralytics.

4.7.1. DIOR

DIOR, presentando anteriormente, al ser el conjunto de datos con más categorías de este estudio, se puede interpretar que puede ser con el que menos porcentaje de mAP obtenga, y así es, aún así, se consiguen muy buenos resultados. El porcentaje de mAP varía mucho dependiendo del modelo que se utilice.

Los resultados obtenidos con los modelos YOLO son los siguientes:

mAP50	mAP50-95	Épocas	Modelo
0.804	0.584	10	YOLOv8x
0.91102	**0.71919**	**50**	**YOLOv8x**
0.811	0.574	10	YOLOv8s
0.896	0.686	50	YOLOv8s
0.77384	0.48475	10	YOLOv5x
0.86934	0.61707	50	YOLOv5x
0.86567	0.60835	10	YOLOv5s
0.902	0.702	50	YOLOv5s
0.73123	0.42898	150	FasterRCNN
0.78123	0.48044	300	FasterRCNN

Tabla. 4.1: Resultados de los modelos en el conjunto de datos DIOR. (Fuente propia)

En la figura 4.9 podemos ver el mejor modelo, en este caso **YoloV8x** prediciendo los objetos dentro de la imagen con una alta probabilidad en cada uno de los objetos.

En este caso además es interesante ver el **F1 Score** resultados de las predicciones de este conjunto de datos, ya que al tener 20 clases y estar desbalanceadas podemos tener una idea cuales son aquellas clases que tiene menos probabilidad de ser localizadas.

Como vemos en la figura 4.10, la clase con mejor puntuación son las chimeneas, es bastante curioso, ya que se podría deber a la homogeneidad de la formas de ellas, o por el humo expulsado debido a que es muy distintivo, mientas que la clase con menos porcentaje de acierto son los puentes y los puertos, esto se podría deberse a la heterogeneidad de estas infraestructuras ya que suelen tener formas muy diferentes entre ellas.

Figura 4.9: Imágenes predichas de YoloV8x en el conjunto de datos DIOR con 50 épocas (Fuente propia)

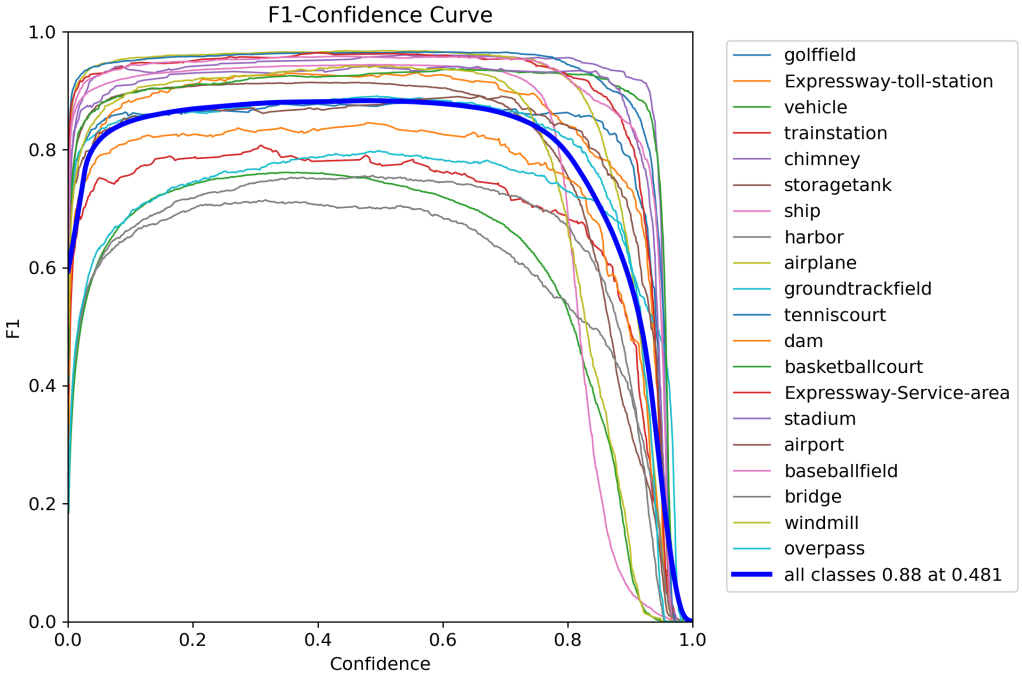

Figura 4.10: F1 Score DIOR con YoloV8x y 50 épocas (Fuente propia)

4.7.2. Oil Storage Tanks

A diferencia del anterior conjunto de datos, este conjunto al igual que el siguiente tiene una única categoría, en este caso, tanques de almacenamientos de petróleo. Pero a diferencia del siguiente, los tanques tienen una forma a priori bastante más sencilla por lo que los métodos podrían ser más propenso a cometer errores con otro tipo de estructuras circulares.

Los resultados podemos verlo en la tabla 4.2, donde en este caso no hay un claro vencedor ya que tanto **Yolov8x**, **Yolov8s** y **Yolov5x** son bastantes parecidos.

A pesar de ello, acabaría escogiendo como mejor opción el modelo YoloV8s, por dos motivos esenciales, el primero al ser un modelo mucho más **moderno** por lo que estará mejor mantenido y además por ser el modelo **small**, esto aumentaría la velocidad con la que detectaría los objetos respecto a YoloV5x

mAP50	mAP50-95	Épocas	Modelo
0.94848	0.78064	10	YOLOv5s
0.9776	0.91207	50	YOLOv5s
0.335	0.279	10	YOLOv8x
0.983	**0.922**	**50**	**YOLOv8x**
0.966	0.868	10	YOLOv8s
0.981	**0.933**	**50**	**YOLOv8s**
0.95344	0.80689	10	YOLOv5x
0.98219	**0.92943**	**50**	**YOLOv5x**
0.95129	0.82069	150	FasterRCNN
0.95690	0.81319	300	FasterRCNN

Tabla. 4.2: Resultados de los modelos en el conjunto de datos Oil Tank Storage. (Fuente propia)

En la figura 4.11 observamos que no todos los tanques están señalados. Por ello, es extraño en principio, pero si vemos las anotaciones originales de estas imágenes en la figura 4.12, vemos que no todos están señalados, esto podría ser un error del conjunto de datos, o, los tanques no señalados podrían no ser tanques de almacenamiento de petróleo.

En la matriz de confusión de la figura 4.13, vemos que la mayoría de anotaciones están en la diagonal, es decir, no ha habido muchos falsos positivos, aún así sigue habiendo pero esto es normal ya que ningún detector de objetos queda libre de error.

Figura 4.11: Predicción de YOLOV8s en el conjunto de datos Oil Tanks Storage con 50 épocas (Fuente propia)

Figura 4.12: Anotaciones de Oil Tanks Storage (Fuente propia)

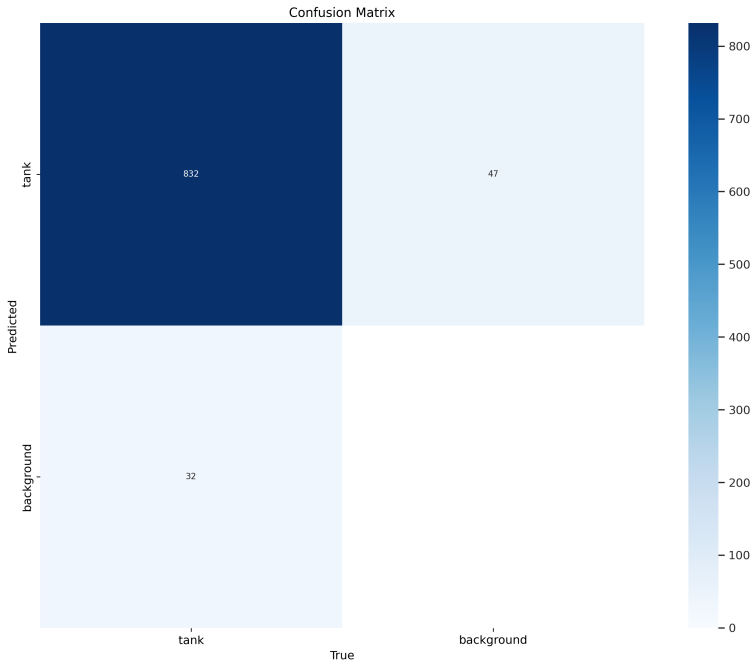

Figura 4.13: Matriz de confusión del modelo YOLOV8s en el conjunto de datos Oil Tanks Storage con 50 épocas (Fuente propia)

4.7.3. HRPlanesv2

Por último, el conjunto de datos HRPlanesv2 nos demuestra que el modelo YOLOV8s tiene un rendimiento mejor de lo esperado atendiendo a los resultados de la tabla 4.3.

Podría ser casualidad que en conjuntos de datos con una única categoría, el rendimiento de YOLOV8s fuese más aconsejable que los modelos más grandes. Le sigue muy de cerca su hermano mayor YOLOV8x, pero en mi opinión, el tiempo de entrenamiento y el tiempo de detección del modelo extra largo, debilita su posición como elección a escoger.

Aún así en este conjunto de datos, todos los modelos tienen buen rendimiento, incluyendo en esta lista FasterRCNN.

mAP50	mAP50-95	Épocas	Modelo
0.96454	0.77092	10	YOLOv8x
0.98848	0.87905	50	YOLOv8x
0.967	0.789	10	YOLOv8s
0.990	**0.880**	**50**	**YOLOv8s**
0.94696	0.72921	10	YOLOv5s
0.98101	0.82655	50	YOLOv5s
0.96487	0.76906	10	YOLOv5x
0.9867	0.85815	50	YOLOv5x
0.96934	0.74442	150	FasterRCNN
0.96568	0.71439	300	FasterRCNN

Tabla. 4.3: Resultados de los modelos en el conjunto de datos HRPlanesv2.

En este caso hay que resaltar que FasterRCNN parece sufrir un problema de sobreajuste debido a que el porcentaje de acierto disminuye en el entrenamiento con 300 épocas. Sería interesante cambiar otros hiperparámetros del modelo como el *learning rate* o modificado el conjunto de datos para intentar obtener una mejor tasa de acierto.

En la figura 4.14 podemos ver un ejemplo de las predicciones del modelo y vemos que son bastante acertadas, incluso detectando aviones que están parcialmente fuera de la imagen, esto debido al entrenamiento y preprocesamiento anteriormente realizado en este conjunto de datos.

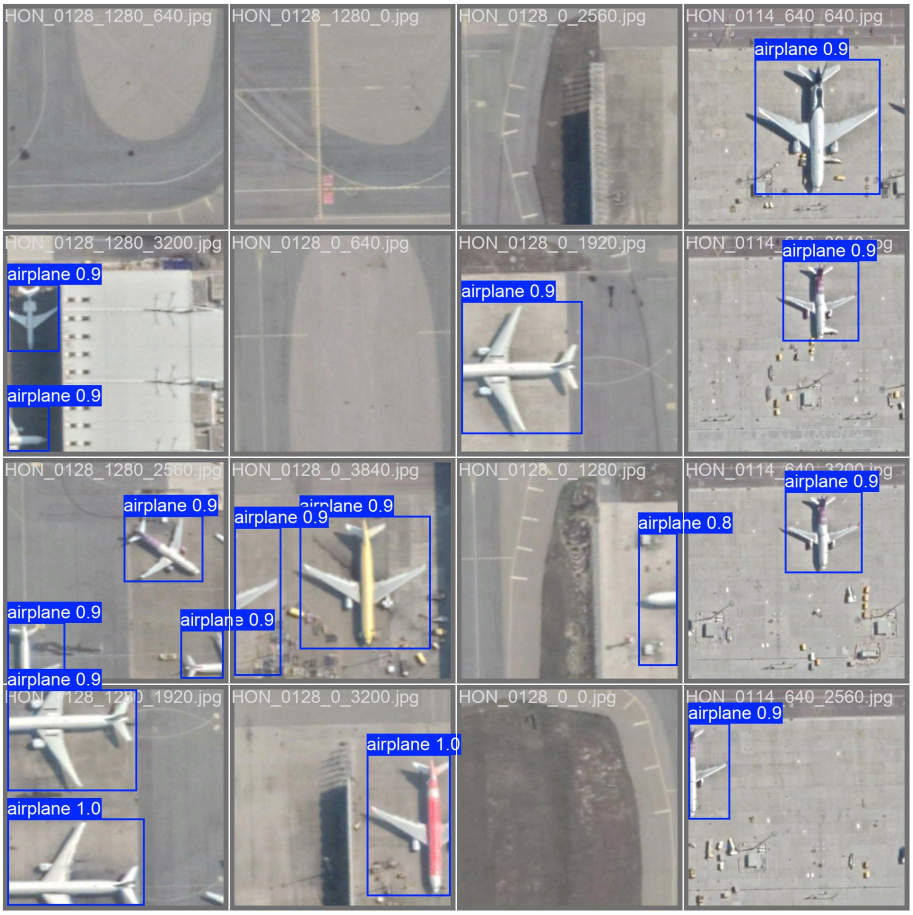

Figura 4.14: Predicción de YOLOV8s en el conjunto de datos HRPlanesv2 con 50 épocas (Fuente propia)

Capítulo 5

Conclusiones

Este trabajo de fin de grado se ha abordado el problema de la detección de objetos en imágenes satelitales utilizando técnicas avanzadas de aprendizaje profundo. A lo largo de este estudio, se han implementado y evaluado diversos modelos de redes neuronales convolucionales, centrándose en la identificación y clasificación de objetos en imágenes de alta complejidad. A continuación, se exponen las principales conclusiones obtenidas a partir de los resultados y análisis realizados:

5.1. Importancia del Contexto en la Detección de Objetos

Uno de los aspectos más destacados de este estudio es la importancia del contexto en la detección de objetos en imágenes satelitales. Las variaciones en la resolución, las condiciones climáticas y la diversidad de los objetos presentes en las imágenes presentan desafíos significativos para los modelos de detección.

En este sentido, el trabajo subraya la necesidad de aplicar técnicas de preprocesamiento de imágenes que mejoren la calidad de los datos antes de su ingreso a los modelos de detección. La elección adecuada de los parámetros de los modelos, así como la aplicación de técnicas de aumento de datos, contribuyeron a mejorar significativamente los resultados. Estas técnicas permitieron que los modelos generalizaran mejor, adaptándose a una amplia variedad de escenarios, lo que es crucial en el contexto de las imágenes satelitales, donde las condiciones pueden variar drásticamente.

5.2. Aplicaciones Prácticas

Las técnicas desarrolladas y evaluadas en este trabajo tienen aplicaciones que van más allá del ámbito académico. En particular, las capacidades de detección de objetos en imágenes satelitales tienen implicaciones directas en áreas como la gestión de desastres naturales, la monitorización ambiental, la vigilancia de infraestructuras críticas y la planificación urbana entre otras.

5.3. Comparación de Modelos

En este trabajo se han explorado diferentes arquitecturas de redes neuronales, entre las que destacan YOLOv5, YOLOv8 y Faster R-CNN. Estos modelos fueron seleccionados por su relevancia en el campo de la visión por computador y su capacidad para realizar detecciones de alta precisión en tiempo real.

- **YOLOv5** mostró ser una opción robusta en términos de velocidad, siendo capaz de realizar detecciones rápidas en imágenes con objetos de diferentes tamaños. Sin embargo, su desempeño en términos de precisión fue superado por modelos más recientes.
- **YOLOv8**, la versión más reciente de la familia YOLO, demostró ser un avance significativo en comparación con sus anteriores versiones.
- **Faster R-CNN**, aunque no tan rápido como los modelos YOLO, proporcionó también buena precisión en la detección de objetos en escenarios donde la velocidad no es crítica.

En conjunto, los resultados experimentales indicaron que cada modelo tiene sus fortalezas y debilidades, dependiendo del contexto de la aplicación. En resumen, YOLOv8 se posiciona como la mejor opción para aplicaciones que requieren un alto rendimiento en tiempo real.

5.4. Desafíos y Oportunidades Futuras

A pesar de los avances logrados, durante este trabajo se ha identificado una serie de desafíos que deben abordarse en investigaciones futuras. Uno de los principales

retos es la mejora en la detección de objetos en imágenes de baja resolución o en aquellos escenarios donde los objetos son pequeños y están parcialmente ocultos.

La detección de objetos débiles sigue siendo un problema, especialmente en imágenes satelitales donde la resolución y la claridad de los objetos pueden verse afectada. Además, la capacidad de los modelos para adaptarse a nuevas clases de objetos sin requerir grandes cantidades de datos etiquetados sigue siendo un problema a resolver para la comunidad investigadora.

En términos de oportunidades futuras, se sugiere la integración de técnicas de superresolución y aprendizaje semi-supervisado para abordar estos desafíos. La superresolución podría mejorar la calidad de las imágenes, facilitando la detección de objetos pequeños, mientras que el aprendizaje semi-supervisado permitiría a los modelos aprender de datos no etiquetados, reduciendo la dependencia de grandes conjuntos de datos anotados manualmente.

5.5. Conclusión Final

En conclusión, este trabajo ha demostrado que las técnicas de aprendizaje profundo aplicadas a la detección de objetos en imágenes satelitales son herramientas poderosas con un amplio rango de aplicaciones prácticas. Los avances logrados en este campo tienen el potencial de transformar la manera en la que interactuamos con los datos geoespaciales, mejorando la precisión y eficiencia de las tareas de detección. A medida que la tecnología sigue avanzando, es probable que veamos mejoras continuas en estos modelos, lo que abrirá nuevas posibilidades y aplicaciones en diversos sectores.

Bibliografía

[1] *(1) What is Convolutional Neural Network — CNN (Deep Learning) | LinkedIn*. URL: https://www.linkedin.com/pulse/what-convolutional-neural-network-cnn-deep-learning-nafiz-shahriar/ (visitado 03-09-2024).

[2] *Azure Machine Learning: ML como servicio | Microsoft Azure*. URL: https://azure.microsoft.com/es-es/products/machine-learning (visitado 25-08-2024).

[3] Luke Borkowski. *lbborkowski/wind-turbine-detector*. original-date: 2019-05-31T00:29:35Z. 4 de mayo de 2024. URL: https://github.com/lbborkowski/wind-turbine-detector (visitado 29-05-2024).

[4] Jarod David Pedraza Caro. *La Inteligencia Artificial en la sociedad:Explorando su Impacto Actual y los Desafíos Futuros*. Universidad Politécnica de Madrid, 2023.

[5] Oleksii Chalyi. «An Evaluation of General-Purpose AI Chatbots: A Comprehensive Comparative Analysis». En: *InfoScience Trends* 1.1 (1 de jun. de 2024), págs. 52-66. ISSN: 3041-9875. DOI: 10.61186/ist.202401.01.07. URL: https://www.isjtrend.com/article_197152.html (visitado 06-09-2024).

[6] Evelyn Cheng. *China releases plans to restrict facial recognition technology*. CNBC. 8 de ago. de 2023. URL: https://www.cnbc.com/2023/08/08/china-releases-plans-to-restrict-facial-recognition-technology.html (visitado 06-02-2024).

[7] E. R. Davies. «Advanced Methods and Deep Learning in Computer Vision». En: *Advanced Methods and Deep Learning in Computer Vision*. Ed. por E. R. Davies y Matthew A. Turk. Computer Vision and Pattern Recognition. Academic Press, 1 de ene. de 2022, págs. 1-91. ISBN: 9780128221099. DOI: 10.1016/B978-0-12-822109-9.00010-2. URL: https://www.sciencedirect.com/science/article/pii/B9780128221099000102 (visitado 07-02-2024).

[8] Serestina Viriri Dekanmi Adeyinka Adegun Jean Vincent Fonou Dombeu y John Odindi. «State-of-the-Art Deep Learning Methods for Objects Detection in Remote Sensing Satellite Images». En: *Sensors* (2023).

[9] *Earth Observation*. URL: https://eod-grss-ieee.com/ (visitado 20-03-2024).

[10] *Earth Observation*. URL: https://eod-grss-ieee.com/ (visitado 20-03-2024).

[11] *facebookresearch/detectron2*. original-date: 2019-09-05T21:30:20Z. 26 de ago. de 2024. URL: https://github.com/facebookresearch/detectron2 (visitado 26-08-2024).

[12] Jeff Faudi. *Oil Storage Detection on Airbus Imagery with YOLOX*. Artificialis. 16 de jun. de 2022. URL: https://medium.com/artificialis/oil-storage-detection-on-airbus-imagery-with-yolox-9e38eb6f7e62 (visitado 20-06-2024).

[13] Pedro F. Felzenszwalb et al. «Object Detection with Discriminatively Trained Part-Based Models». En: *IEEE Transactions on Pattern Analysis and Machine Intelligence*. Vol. 32. 9. 2010, págs. 1627-1645.

[14] *FIGURE 2. Basic deep learning-based one-stage vs. two-stage object...* ResearchGate. URL: https://www.researchgate.net/figure/Basic-deep-learning-based-one-stage-vs-two-stage-object-detection-model-architectures_fig2_358362847 (visitado 08-02-2024).

[15] Rohith Gandhi. *R-CNN, Fast R-CNN, Faster R-CNN, YOLO — Object Detection Algorithms*. Medium. 9 de jul. de 2018. URL: https://towardsdatascience.com/r-cnn-fast-r-cnn-faster-r-cnn-yolo-object-detection-algorithms-36d53571365e (visitado 10-02-2024).

[16] *GitHub - aurotripathy/ssd-spacenet: Detect buildings in the Spacenet satellite imagery dataset using Single Shot MultiBox Detector (SSD)*. URL: https://github.com/aurotripathy/ssd-spacenet?tab=readme-ov-file (visitado 20-06-2024).

[17] *GitHub - open-mmlab/mmrotate: OpenMMLab Rotated Object Detection Toolbox and Benchmark*. URL: https://github.com/open-mmlab/mmrotate (visitado 21-03-2024).

[18] Ian Goodfellow, Yoshua Bengio y Aaron Courville. *Deep Learning*. Cambridge, MA: MIT Press, 2016. ISBN: 978-0262035613.

[19] Tuomo Hiippala. «Recognizing military vehicles in social media images using deep learning». En: 1 de jul. de 2017. DOI: 10.1109/ISI.2017.8004875.

[20] *HumanSignal/labelImg*. original-date: 2015-09-17T01:33:59Z. 18 de feb. de 2024. URL: https://github.com/HumanSignal/labelImg (visitado 18-02-2024).

[21] *Hyperspectral and Multispectral Imaging*. URL: https://www.edmundoptics.ca/knowledge-center/application-notes/imaging/hyperspectral-and-multispectral-imaging/ (visitado 20-03-2024).

[22] Ida. *How object detectors learn*. Ambolt. 21 de feb. de 2023. URL: https://ambolt.io/en/how-object-detectors-learn/ (visitado 06-02-2024).

[23] *Intersección sobre la unión*. URL: https://digifarm.io/es/blog/intersection-over-union (visitado 10-05-2024).

[24] Glenn Jocher et al. *YOLOv5: You Only Look Once Version 5*. Accessed: 2024-09-06. 2020. URL: https://github.com/ultralytics/yolov5.

[25] Glenn Jocher et al. *YOLOv8: You Only Look Once Version 8*. Accessed: 2024-09-06. 2023. URL: https://github.com/ultralytics/ultralytics.

[26] *Kaggle: Your Machine Learning and Data Science Community*. URL: https://www.kaggle.com/ (visitado 06-09-2024).

[27] Jaskirat Kaur y Williamjeet Singh. «Tools, techniques, datasets and application areas for object detection in an image: a review». En: *Multimedia Tools and Applications* 81.27 (1 de nov. de 2022), págs. 38297-38351. ISSN: 1573-7721. DOI: 10.1007/s11042-022-13153-y. URL: https://doi.org/10.1007/s11042-022-13153-y (visitado 06-02-2024).

[28] Anastasia Kharina. *akharina/elephant_detection*. original-date: 2019-11-18T14:19:31Z. 25 de abr. de 2024. URL: https://github.com/akharina/elephant_detection (visitado 20-06-2024).

[29] Motoki Kimura. *motokimura/cowc_car_counting*. original-date: 2018-11-03T08:25:00Z. 26 de mar. de 2024. URL: https://github.com/motokimura/cowc_car_counting (visitado 29-05-2024).

[30] Moez Krichen. *Computers | Free Full-Text | Convolutional Neural Networks: A Survey*. 2023.

[31] Alex Krizhevsky, Ilya Sutskever y Geoffrey E Hinton. «Imagenet classification with deep convolutional neural networks». En: *Communications of the ACM* 60.6 (2017), págs. 84-90.

[32] *LeNet*. En: *Wikipedia*. Page Version ID: 1236713920. 26 de jul. de 2024. URL: https://en.wikipedia.org/w/index.php?title=LeNet&oldid=1236713920 (visitado 03-09-2024).

[33] Chuyi Li et al. *YOLOv6 v3.0: A Full-Scale Reloading*. 2023. arXiv: 2301.05586 [cs.CV].

[34] Wei Liu et al. «SSD: Single Shot MultiBox Detector». En: vol. 9905. 2016, págs. 21-37. arXiv: 1512.02325[cs]. URL: http://arxiv.org/abs/1512.02325 (visitado 10-02-2024).

[35] *Machine Learning: Amazon Web Services*. Amazon Web Services, Inc. URL: https://aws.amazon.com/es/sagemaker/ (visitado 25-08-2024).

[36] *Make Sense*. URL: https://www.makesense.ai/ (visitado 18-02-2024).

[37] *Oil Storage Tanks*. URL: https://www.kaggle.com/datasets/towardsentropy/oil-storage-tanks (visitado 18-02-2024).

[38] Xabier Basogain Olabe. *Redes neuronales artificiales y sus aplicaciones*. Escuela Superior de Ingeniería de Bilbao, EHU, 2020.

[39] pedrozamboni. *pedrozamboni/individual_urban_tree_crown_detection*. original-date: 2021-06-09T17:53:50Z. 6 de feb. de 2024. URL: https://github.com/pedrozamboni/individual_urban_tree_crown_detection (visitado 17-02-2024).

[40] Sabina Pokhrel. *Image Data Labelling and Annotation — Everything you need to know*. Medium. 11 de mar. de 2020. URL: https://towardsdatascience.com/image-data-labelling-and-annotation-everything-you-need-to-know-86ede6c684b1 (visitado 18-02-2024).

[41] *Productos y servicios de IA y de aprendizaje automático*. Google Cloud. URL: https://cloud.google.com/products/ai (visitado 25-08-2024).

[42] Shaoqing Ren et al. *Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks*. 6 de ene. de 2016. arXiv: 1506.01497[cs]. URL: http://arxiv.org/abs/1506.01497 (visitado 10-02-2024).

[43] Christoph Rieke. *chrieke/awesome-satellite-imagery-datasets*. original-date: 2018-05-01T22:13:17Z. 8 de abr. de 2024. URL: https://github.com/chrieke/awesome-satellite-imagery-datasets (visitado 10-04-2024).

[44] Saúl Rozada Raneros. «Estudio de la arquitectura YOLO para la detección de objetos mediante deep learning». En: (2021). URL: https://uvadoc.uva.es/handle/10324/45359 (visitado 08-02-2024).

[45] Stuart Russell y Peter Norvig. *Artificial Intelligence: A Modern Approach*. 4.ª ed. Hoboken, NJ: Pearson, 2020.

[46] *satellite-image-deep-learning/techniques*. original-date: 2018-04-16T08:42:09Z. 21 de mar. de 2024. URL: https://github.com/satellite-image-deep-learning/techniques (visitado 21-03-2024).

[47] Divyam Shah. *divyam96/EESRGAN*. original-date: 2020-11-03T09:12:51Z. 17 de jul. de 2023. URL: https://github.com/divyam96/EESRGAN (visitado 29-05-2024).

[48] *Ship-Detection-from-Satellite-Images-using-YOLOV4*. URL: https://github.com/debasis-dotcom/Ship-Detection-from-Satellite-Images-using-YOLOV4?tab=readme-ov-file (visitado 21-03-2024).

[49] soporteconfediorg. *Reconocimiento de objetos a través de la metodología Haar Cascades*. CONFEDI. URL: https://confedi.org.ar/reconocimiento-de-objetos-a-traves-de-la-metodologia-haar-cascades/ (visitado 07-02-2024).

[50] Dilsad Unsal. *HRPlanesv2 - High Resolution Satellite Imagery for Aircraft Detection*. 1 de feb. de 2022. DOI: 10.5281/zenodo.7331974. URL: https://zenodo.org/records/7331974 (visitado 28-08-2024).

[51] Paul Viola y Michael Jones. «Rapid Object Detection using a Boosted Cascade of Simple Features». En: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2001), págs. 511-518.

[52] Minghui Wang et al. «Highly Efficient Anchor-Free Oriented Small Object Detection for Remote Sensing Images via Periodic Pseudo-Domain». En: *Remote Sensing* 15 (3 de ago. de 2023), pág. 3854. DOI: 10.3390/rs15153854.

[53] *What Is Object Detection? - MATLAB & Simulink*. URL: https://www.mathworks.com/discovery/object-detection.html (visitado 06-02-2024).

[54] Yuxin Wu et al. *Detectron2*. Accessed: 2024-09-06. 2019. URL: https://github.com/facebookresearch/detectron2.

[55] Aston Zhang et al. *Dive into Deep Learning*. Capítulo 7.6: Convolutional Neural Networks (LeNet). Cambridge, New York, Port Melbourne, New Delhi, Singapore: Cambridge University Press, 2024. ISBN: 978-1-009-38943-3.

[56] Xiangrong Zhang et al. «Remote Sensing Object Detection Meets Deep Learning: A metareview of challenges and advances». En: *IEEE Geoscience and Remote Sensing Magazine* 11.4 (dic. de 2023), págs. 8-44. ISSN: 2168-6831. DOI: 10.1109/MGRS.2023.3312347. URL: https://ieeexplore.ieee.org/abstract/document/10292941 (visitado 10-05-2024).