Universidad de Oviedo

EPM ESCUELA POLITÉCNICA DE MIERES

TRABAJO FIN DE GRADO

GRADO EN INGENIERÍA GEOMÁTICA

Una aplicación de la detección de objetos en imágenes UAV con Redes Neuronales Convolucionales (CNN)

Autor: **Pablo Álvarez Luengo**

Tutor: Silverio García Cortés

Cotutor: Agustín Menéndez Díaz

Febrero, 2024

Universidad de Oviedo

EPM
ESCUELA POLITÉCNICA DE MIERES

Declaración de Originalidad del Trabajo Fin de Grado

D. Pablo Álvarez Luengo, con DNI estudiante del Grado en Ingeniería Geomática de la Escuela Politécnica de Mieres de la Universidad de Oviedo, declaro bajo mi responsabilidad que:

El Trabajo de Fin de Grado aquí presentado con título “Una aplicación de la detección de objetos en imágenes UAV con Redes neuronales convolucionales (CNN)” ha sido realizado bajo mi autoría, es original y que todas las fuentes utilizadas para su realización han sido debidamente citadas en el mismo.

Para que así conste, firmo la presente declaración.

En Mieres, a 2 de febrero de 2024.

Pablo Álvarez Luengo

Relación del TFG con los Objetivos de Desarrollo Sostenible

ODS con los que se relaciona el TFG:

Breve justificación:

En el desarrollo de cualquier proyecto es importante tener en cuenta los objetivos que nos proponemos como sociedad y avanzar hacia su cumplimiento. En el contexto de este trabajo cabe destacar los Objetivos de Desarrollo Sostenible de la agenda 2030 de la ONU.

El proyecto que se desarrolla a continuación busca repercutir en los siguientes aspectos:

Objetivo 2, Hambre cero: Este proyecto busca avanzar hacia la automatización del mundo agrario, abaratando los costes y mejorando las producciones con sistemas versátiles y de bajo coste que puedan servir a grandes y pequeños productores.

Objetivo 9, Industria, Innovación e Infraestructura: Se busca modernizar la infraestructura y los métodos utilizados en la industria agraria, fomentando la investigación y la innovación.

Objetivo 12, Producción y consumo responsables: El objetivo fundamental de este trabajo, la automatización de la estimación de cosechas para sustituir a métodos visuales, permitirá estimar de manera más fiable las cantidades a recolectar, ayudando a reducir la generación de residuo alimentario en las primeras etapas de su producción.

CONTENIDO

1. INTRODUCCIÓN

1.1. CONTEXTO Y JUSTIFICACIÓN DEL PROYECTO

La visión artificial, una rama emergente de la inteligencia artificial, ha demostrado su eficacia para abordar tareas que, hasta hace poco, se consideraban exclusivas del ojo humano. Mediante el uso de algoritmos sofisticados y el poder del aprendizaje automático, la visión artificial puede procesar, analizar e interpretar imágenes del mundo real para generar datos significativos.

La automatización, impulsada por avances en inteligencia artificial y tecnologías de la información, está transformando rápidamente una amplia gama de industrias. En la agricultura, uno de los sectores más antiguos de la humanidad, la automatización tiene el potencial de revolucionar la forma en que trabajamos con la tierra. Desde la siembra hasta la cosecha, pasando por el riego y el control de plagas, todos los aspectos de la agricultura pueden beneficiarse de la automatización.

En este contexto, este proyecto surge de la necesidad de explorar cómo la visión artificial puede contribuir a la automatización en la agricultura. En concreto, el objetivo es desarrollar e implementar un sistema de detección de objetos para identificar y contar manzanas en los árboles. Esto permitiría a los agricultores evaluar con precisión el volumen de su cosecha, evitando así depender de valoraciones subjetivas que puedan subestimar su producción. En este sentido se pretende obtener una medida más realista que la simple estimación visual del producto en árbol y por tanto unas transacciones comerciales más justas. La detección de fruta en árbol es además un primer paso hacia la automatización de las tareas de cosechado.

Por lo tanto, la justificación de este proyecto radica en su potencial para llevar la automatización a la agricultura de una manera eficiente y rentable, lo que podría tener un impacto significativo en la economía agrícola y en la forma en que gestionamos nuestros recursos alimentarios.

1.2. OBJETIVOS DEL PROYECTO

El propósito principal de este proyecto es diseñar, implementar y validar un detector de objetos basado en visión artificial para identificar y contar manzanas en árboles. Para lograr este propósito general, el proyecto establece los siguientes objetivos específicos:

1. **Estudiar las diferentes arquitecturas de detección de objetos**: Se llevará a cabo una revisión de algunas de las diversas arquitecturas de detección de objetos disponibles, como R-CNN, Fast R-CNN, Faster R-CNN, SSD y YOLO. El objetivo es comprender las fortalezas y debilidades de cada arquitectura y determinar cuál podría ser más adecuada para la detección de frutas en árboles.

2. **Elección de la arquitectura más adecuada para la detección de frutas en árboles**: Basándose en los hallazgos del estudio anterior, se seleccionará la arquitectura de detección de objetos que mejor se adapte a las necesidades y restricciones del problema planteado.

3. **Creación de un conjunto de datos etiquetados para el entrenamiento**: Se recolectarán y etiquetarán imágenes de árboles de manzanas para crear un conjunto de datos que se utilizará para entrenar la arquitectura de detección de objetos seleccionada. Este conjunto de datos deberá ser lo suficientemente grande y diverso como para permitir que el modelo aprenda a detectar manzanas en diferentes condiciones de iluminación, orientaciones y etapas de madurez.

4. **Entrenamiento y validación del detector de objetos**: Una vez seleccionada la arquitectura y creado el conjunto de datos, el siguiente objetivo es entrenar el detector de objetos utilizando este conjunto de datos. Después del entrenamiento, el detector

será validado para evaluar su precisión en la detección de manzanas en imágenes no vistas durante el entrenamiento.

Estos objetivos, si se logran con éxito, permitirán la creación de un sistema de detección de objetos basado en visión artificial que pueda usarse para mejorar la eficiencia y la productividad en la industria de la manzana.

1.3. ESTRUCTURA DEL DOCUMENTO

Partiendo de todo lo anterior, este trabajo se estructura de la siguiente manera: en el Capítulo 2 se describen los fundamentos teóricos sobre los que se basa el trabajo. Una explicación sobre las redes neuronales convolucionales y los conceptos asociados, la visión artificial y las distintas tareas que engloba, así como un repaso a la historia reciente de la detección de objetos, comentando los principales avances y métodos desarrollados en los últimos años. En el Capítulo 3 se exponen las distintas herramientas y tecnologías utilizadas para el desarrollo del proyecto, separadas en hardware y software. En el Capítulo 4 se recogen las distintas etapas que comprende cualquier trabajo de *machine learning*, detallando las características de la arquitectura de red neuronal escogida. Estas etapas se desarrollarán para el caso concreto de este trabajo en el Capítulo 5, exponiendo los resultados que se han ido obteniendo y las problemáticas encontradas.

2. FUNDAMENTOS TEÓRICOS

Antes de comenzar el entrenamiento y trabajo con las herramientas de detección de objetos, es necesario explicar los conceptos fundamentales sobre los que se basan estas herramientas para poder comprenderlas, así como dar una contextualización de la situación de estas tecnologías en la actualidad.

2.1. Redes Neuronales Convolucionales

El Machine Learning o Aprendizaje máquina es actualmente una de las ramas más importantes de las ciencias de la computación, su objetivo es el diseño y entrenamiento de algoritmos capaces de la realización de tareas o toma de decisiones de forma automática, de una manera similar a cómo las haría un humano. Este campo engloba distintas técnicas y algoritmos entre los que destacan las Redes Neuronales.

Una Red Neuronal Artificial es un modelo computacional inspirado en la estructura y funcionamiento del cerebro humano. Su componente básico son las llamadas neuronas artificiales, clasificadores binarios que reciben distintos datos de entrada y generan una única salida.

El modelo original de neurona artificial es el Perceptrón. Fue concebido inicialmente por Rosenblatt[1] como un clasificador binario, recibe una serie de datos de entrada (características) $\{x_1, x_2, \dots, x_n\}$ a los que se les asignan unos pesos $\{w_1, w_2, \dots, w_n\}$ y con los que se realiza una suma ponderada, a la que se añade otro parámetro w_0 llamado sesgo. El resultado (z) de esta suma se utiliza como entrada de una función ϕ, en el caso del perceptrón es una función signo, que asignará valores iguales a -1 cuando la suma ponderada anterior sea menor que 0 y valores iguales a 1 cuando ésta sea mayor.

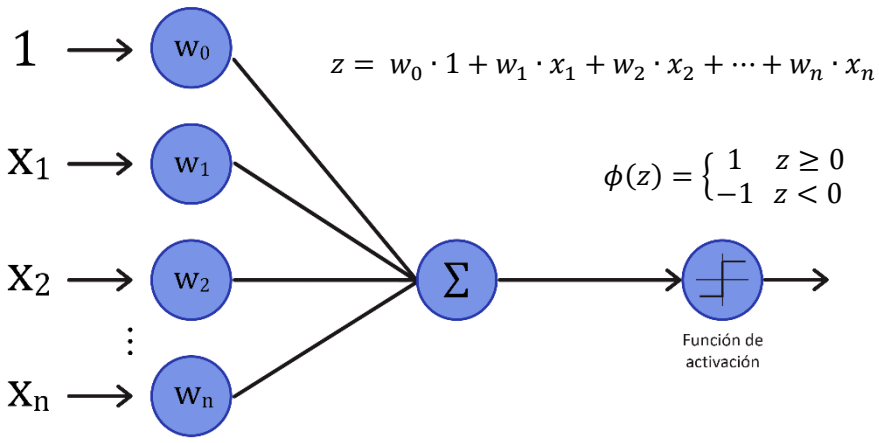


Figura 1: Diagrama del Perceptrón

La evolución de este modelo de neurona artificial fue el modelo *Adaline* (ADAptative LINear Element)[2]. La diferencia principal con su predecesor es el uso de una función de activación lineal, en lugar de una función escalón como es la función signo. Esta función de activación tendrá una salida continua, con lo que se podrá medir el grado de corrección de la predicción con respecto a la salida esperada. Esta medida, llamada función de pérdida, se utiliza durante el entrenamiento para valorar la manera y magnitud en la que deberán de modificarse los pesos.

Conectando estas neuronas entre sí y organizándolas jerárquicamente en capas se obtiene la estructura de red neuronal artificial más básica, el Perceptrón Multicapa o MLP (por sus siglas en inglés *Multi Layer Perceptron*), muy útil para la realización de diversas tareas aunque no es suficiente para el problema planteado en este trabajo ya que, en el caso de las imágenes, los niveles digitales de cada uno de los píxeles se consideran datos de entrada separados, generando un problema de dimensionalidad, es decir, se tienen muchos datos que individualmente aportan muy poca información.

Por esta razón, el modelo más apropiado para trabajar con imágenes son las Redes Neuronales Convolucionales (CNN por sus siglas en inglés *Convolutional Neural Network*).

Las redes neuronales convolucionales son similares a las redes neuronales artificiales tradicionales en el sentido de que están compuestas por neuronas cuyos pesos se optimizan durante el entrenamiento. Cada una de estas neuronas recibe una entrada, realiza una operación y genera una salida.

Las CNN están compuestas principalmente por tres tipos de capas: Convolucionales, de Pooling y Densas.

- Capas Convolucionales: A partir de la imagen de entrada y mediante la aplicación de filtros de convolución, generan una imagen de salida sobre la que se destacan determinadas características de la imagen: formas, contornos, etc. A medida que se

profundiza en la red, estas salidas conocidas como mapas de características o *feature maps* describirán formas y rasgos más complejos.

- Capas de Pooling: Su función es reducir la dimensionalidad de los mapas de características, con el objetivo de reducir el número de parámetros necesarios en las funciones de activación.
- Capas Densas: Son las capas finales de la red, a partir de los resultados obtenidos por las capas convolucionales generarán un vector de resultados como salida, cuyo significado y valores dependerá del trabajo a realizar. En ellas todas las neuronas están interconectadas entre sí, de ahí su nombre.

El número de parámetros con los que trabajan este tipo de redes es mucho menor que en el caso de los MLP o redes neuronales tradicionales, por lo que son mucho más eficientes manejando grandes cantidades de datos.

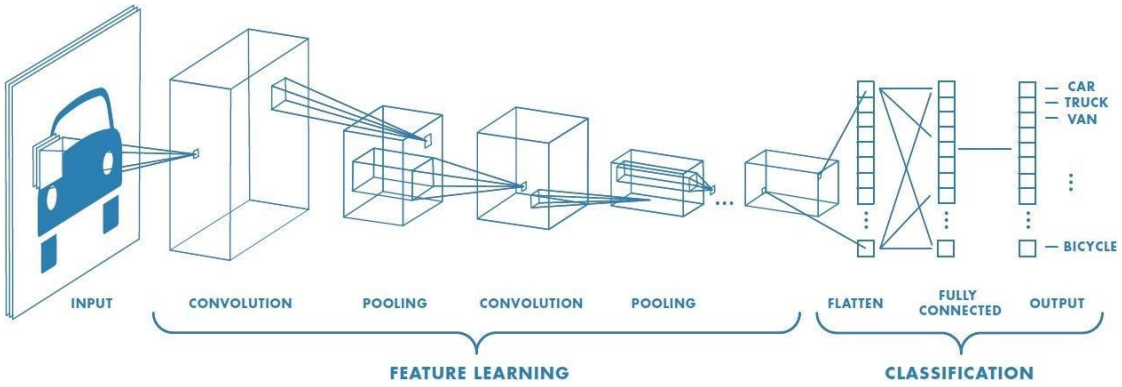


Figura 2: Arquitectura básica de una Red Neuronal Convolucional. Fuente:[3]

2.2. VISIÓN POR ORDENADOR Y DETECCIÓN DE OBJETOS

La Visión Artificial o Visión por Ordenador es una disciplina de la informática que se ocupa de procesar, analizar y comprender imágenes y vídeos, con la intención de extraer información de ellos.

Las tareas básicas que se trabajan en Visión Artificial son:

- Clasificación de imágenes: El objetivo es identificar y clasificar la temática general de la imagen o el objeto protagonista.
  Los algoritmos de este tipo reciben como entrada una imagen con un solo objeto o una escena y producen como salida un único identificador de clase.

- Localización de objetos: El objetivo es determinar la posición de los objetos destacables que aparecen en la imagen, marcándolos con un rectángulo conocido como *bounding box*.
  El algoritmo recibe una imagen con uno o más objetos y genera como salida las *bounding boxes* de los objetos, expresadas en función de las coordenadas imagen de sus esquinas o mediante las coordenadas del centro de la caja, su altura y su anchura.

- Detección de objetos: Es una combinación de las dos técnicas anteriores, se determina la posición de los distintos objetos de la imagen y se le asigna la clase correspondiente a cada uno de ellos.
  El algoritmo recibe como entrada una imagen con uno o más objetos y genera como salida las coordenadas de las *bounding boxes* y los identificadores de clase de cada uno de ellos.
  Este será el problema objetivo de este trabajo.

- Segmentación de objetos: Va un paso más allá de la detección. La segmentación de objetos, también llamada segmentación de escenas o segmentación semántica se encarga de marcar exactamente los píxeles de la imagen que ocupan cada uno de los objetos identificados, no limitándose únicamente a las *bounding boxes*.

Figura 3: Tareas básicas de la Visión Artificial

2.3. DETECTORES DE OBJETOS: UNA VISIÓN GENERAL

2.3.1. Introducción

El problema de la detección de objetos no es un tema nuevo, se ha estado trabajando en ello desde comienzos de los años 2000, con técnicas directas convencionales de análisis de imágenes como son el uso de filtros y gradientes para la detección de las zonas más interesantes de la imagen. Algunos ejemplos de estos métodos son los algoritmos de Viola-Jones[4], basado en *Haar-like features*, o los basados en HOG (*Histograms of Oriented Gradients*) [5], basado en histogramas de gradientes para identificar líneas y formas.

Este tipo de algoritmos, aunque revolucionarios para la época, no eran especialmente robustos ni versátiles, dado que principalmente se utilizaban para detecciones de una sola clase. Además, su implementación es compleja y los tiempos de ejecución demasiado extensos para la mayoría de las aplicaciones.

2.3.2. Métodos de dos etapas

No fue hasta la llegada de las Redes Neuronales Convolucionales (CNN) y el *Deep Learning* en la década de 2010 que se volvió a trabajar en la búsqueda de algoritmos eficientes para la detección de objetos en imágenes, tras ver que este tipo de arquitecturas daban buenos

resultados en las tareas de clasificación de imágenes [6]. En este periodo destacan los métodos de dos etapas (detección de objetos y clasificación), especialmente R-CNN y sus sucesores:

R-CNN: El trabajo de detección y clasificación de objetos se realiza en dos etapas separadas. La primera la realiza un algoritmo de *Selective Search*[7], que propone 2000 regiones de interés basándose en colores, intensidades y texturas de las distintas zonas de la imagen.

Estas regiones propuestas se extraen de la imagen, se estiran a un tamaño uniforme y se utilizan como entrada de la Red Neuronal Convolucional, que se encarga de clasificar los objetos presentes en cada una de ellas si los hubiera.

Finalmente, para cada uno de los objetos se aplica una supresión no máxima en función de la IoU (*Intersection over Union*) de las regiones, para elegir la *bounding box* que mejor se adapte al elemento detectado. [8]

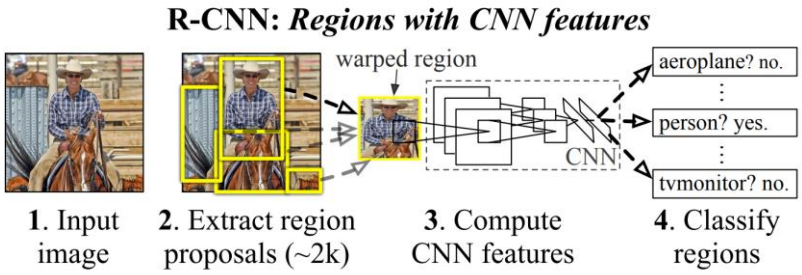


Figura 4: Esquema básico de R-CNN [8]

El algoritmo R-CNN consiguió obtener unos resultados de rendimiento muy altos para la época, especialmente en comparación con sus predecesores. En el test con el conjunto de datos PASCAL VOC 2010-12[9] conseguía una mAP (*mean Average Precision* o precisión media. Véase sección 2.5) del 53.7%, una diferencia amplia respecto a algoritmos anteriores como UVA o DPM v5, con resultados del 35.1% y 33.4% respectivamente. También los tiempos de ejecución eran mucho menores que los de sus predecesores, aunque aún insuficientes para un trabajo en tiempo real, requiriendo de media algo menos de un minuto por cada imagen.

Fast R-CNN: Este nuevo método surge como una mejora del R-CNN, en términos de precisión, pero especialmente de tiempos, tanto de entrenamiento como de ejecución.

Entre las mejoras que propone destacan:

- El entrenamiento es un proceso de una única etapa. Hasta ahora algoritmos como R-CNN entrenaban por separado el detector de objetos, la red neuronal y los regresores para las *bounding boxes*. En Fast R-CNN se propone una función de pérdida conjunta para las distintas etapas, con el fin de realizar el entrenamiento sobre todas a la vez.
- El entrenamiento afecta a todas las capas convolucionales. Aunque esto no era un problema en R-CNN, sí que se daba en alguno de sus principales competidores, como es el caso de SPPnet.
- Se ahorra espacio en disco. Para las fases de entrenamiento de R-CNN, las *features* de cada región propuesta para cada imagen se almacenaban en la memoria del equipo, ocupando incluso cientos de gigabytes en el disco duro.

A diferencia de R-CNN, este algoritmo recibe como entrada la imagen y las propuestas de regiones de interés en forma de *bounding boxes* realizadas por *Selective Search*. El algoritmo pasa la imagen original a través de una serie de capas convolucionales y de *max-pooling* con el objetivo de generar un *feature map* de la imagen.

Con las *bounding boxes* generadas por *Selective Search*, extrae los *features* del *feature map* en forma de vector. Cada uno de estos vectores se introduce en una red densa que realiza la clasificación de los objetos y el refinamiento de las *bounding boxes* en paralelo.

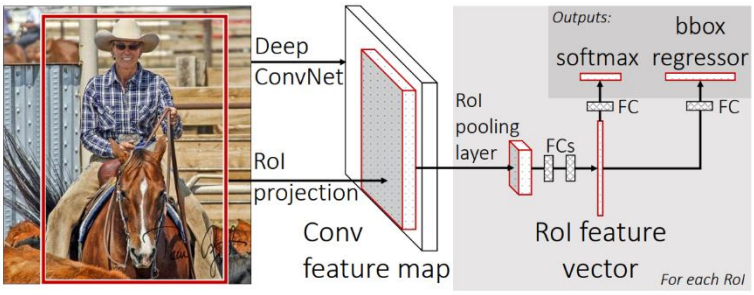


Figura 5: Esquema de Fast R-CNN [10]

Empleando la misma arquitectura VGG-16 (red neuronal convolucional de 16 capas) y el mismo conjunto de entrenamiento VOC 2007[11], Fast R-CNN consigue ser 146 veces más rápido que R-CNN en detección de objetos (sin tener en cuenta la propuesta de regiones de interés), y 9 veces más rápido en el entrenamiento de la red, manteniendo unos valores de mAP iguales o incluso ligeramente superiores.[10]

Faster R-CNN: Con el método anterior, se reducen enormemente los tiempos de clasificación de los objetos detectados, pero el cuello de botella se encuentra ahora en la propuesta de las regiones de interés (RoI, *Region of Interest*), que puede llevar hasta 2 segundos por imagen con *Selective search*, por 0.2 segundos que tarda Fast R-CNN en realizar el resto del proceso. Faster R-CNN propone deshacerse del algoritmo *Selective Search* y utilizar una Red Neuronal Convolucional de propuesta de regiones de interés en su lugar, llamada *Region Proposal Network* (RPN). Este cambio de arquitectura, combinado con la posibilidad de ejecutarla en GPU en lugar de en CPU, reduce los tiempos de propuesta de regiones hasta los 10 milisegundos por imagen.

De esta forma la RPN genera un *feature map* a partir del cual realiza la propuesta de regiones de interés, y con esto alimenta la segunda parte de la red de la misma forma que Fast R-CNN.

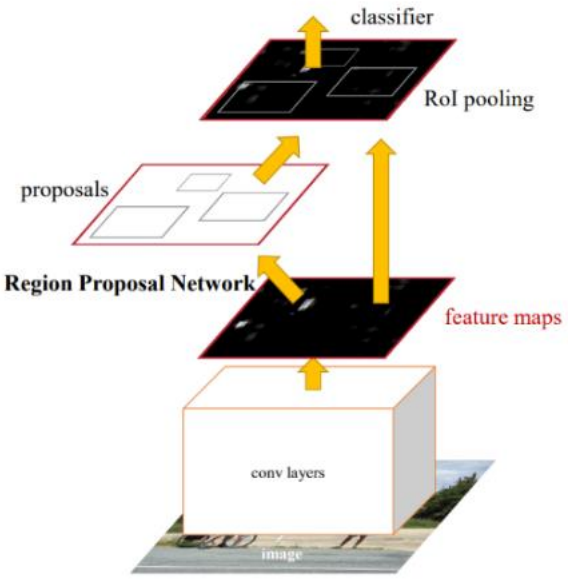


Figura 6: Esquema de Faster R-CNN [12]

Utilizando el *dataset* VOC 2007, la misma arquitectura VGG-16 y ejecutando en la misma GPU (modelo K40, aunque *Selective Search* se ejecuta en la CPU), Fast R-CNN con *Selective Search* (SS) tarda 1830 milisegundos en realizar la detección de objetos, mientras que utilizando RPN en lugar de SS ese tiempo se reduce hasta los 198 milisegundos, consiguiendo así una velocidad de 5 fotogramas por segundo, lo cual permite una implementación del método en tiempo real. Además de la mejora en los tiempos, también se consigue un ligero incremento en la precisión de las detecciones, aumentando el mAP para VOC 2007 hasta un 69.9%, mientras que Fast R-CNN obtenía un mAP del 65.7%. [12]

2.3.3. Métodos de una etapa

La siguiente evolución en los algoritmos de detección de objetos con el objetivo de hacerlos más simples y por tanto más rápidos fue fusionar ambas etapas (Detección y clasificación) en una misma red. Estos métodos son los conocidos como métodos de una etapa y son los utilizados en la actualidad, entre ellos cabe destacar:

Single Shot Detector (SSD): En el algoritmo anterior, se utilizaba un remuestreo de la imagen a distintas escalas para la detección de objetos de distintos tamaños, lo cual es un proceso costoso y que además se realiza aparte del resto de la ejecución.

SSD ataja esta debilidad realizando la detección de objetos durante el proceso de clasificación. Simplificando la explicación, SSD realiza la detección de objetos sobre los *feature maps* extraídos al final de cada capa de la CNN de clasificación. Dado que estos son progresivamente más finos en su descripción de la imagen, consigue el efecto de la detección a distintas escalas sin necesidad de añadir etapas de remuestreo de la imagen.

SSD es capaz de conseguir velocidades y precisiones mucho más altas que Faster R-CNN, llegando a alcanzar los 59 fotogramas por segundo con una mAP del 74,3%, ejecutándose en el mismo equipo (Nvidia Titan X) y utilizando la misma arquitectura VGG-16 y el *dataset* VOC 2007. [13]

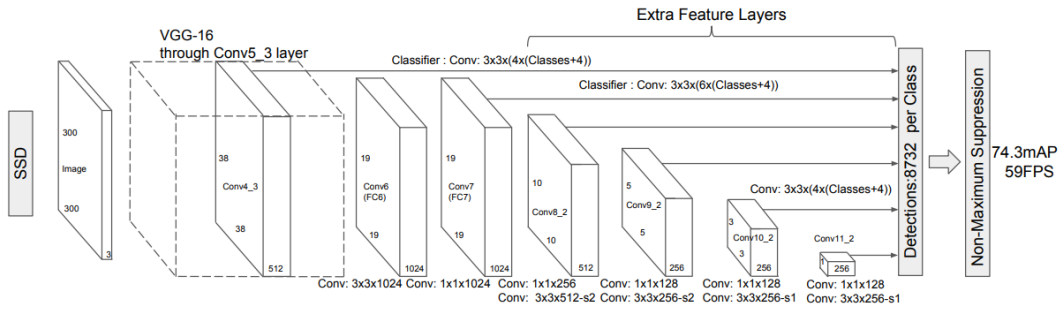


Figura 7: Esquema de la arquitectura de SSD [13]

YOLO: Al igual que SSD, YOLO emplea una única red neuronal convolucional para realizar las distintas etapas del proceso de detección de objetos. Su funcionamiento es simple:

En primera instancia, divide la imagen en una cuadrícula de tamaño S x S. Si el centro de un objeto de la imagen está dentro de una de las celdas de la cuadrícula, esa será la responsable de detectarlo.

En cada celda se predicen B *bounding boxes* de distintos tamaños y relaciones de aspecto, además de la confianza asociada a cada una de ellas. Esta confianza engloba la confianza que tiene el modelo de que esa caja contiene un objeto y la precisión de su definición. Se calcula como $\Pr(\text{Objeto}) * \text{IoU}_{pred}^{truth}$, donde el segundo término representa la IoU entre la caja predicha y la real.

Cada *bounding box* cuenta con 5 parámetros: x, y, w, h y la confianza. Las coordenadas (x,y) representan el centro de la caja con respecto a la celda en la que se encuentre, y (w,h) el ancho y alto de la misma.

Para cada celda en las que se subdivide la imagen también se calculan las probabilidades condicionadas de cada clase, condicionadas por la existencia o no de un objeto en la celda. Sólo se calcula un set de probabilidades por cada celda, independientemente del número de *bounding boxes* que contenga.

Finalmente se combinan estos valores para dar como salida de la red un tensor de dimensión $S \times S \times (B * 5 + C)$, con las coordenadas de las cajas y las probabilidades de cada clase de objeto asociadas.

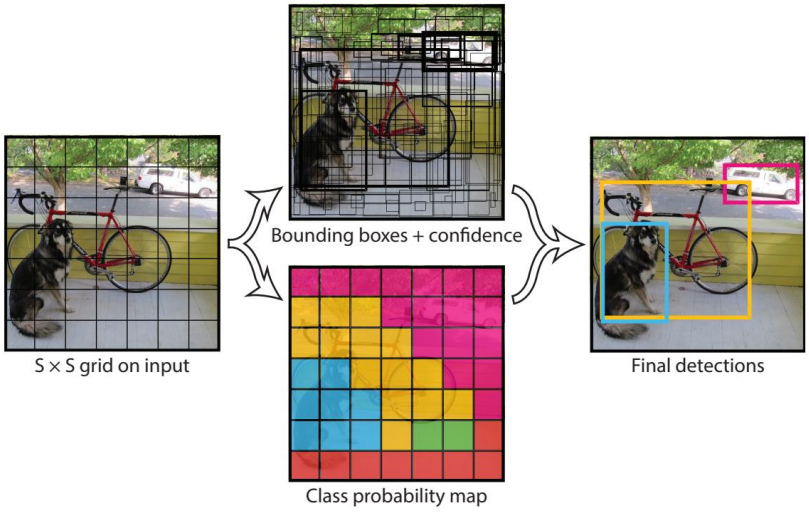


Figura 8: Esquema de funcionamiento de YOLO [14]

A diferencia de los algoritmos expuestos previamente, La versión original de YOLO se implementa sobre una arquitectura similar a la *GoogLeNet* [15], con 24 capas convolucionales seguidas de 2 capas densas. La última capa utiliza una función de activación lineal, y todas las anteriores una *Leaky ReLU* con α=0.1.

$$f(x) = \begin{cases} x \; si \; x \geq 0 \\ \alpha \cdot x \; si \; x < 0 \end{cases}$$

La función de pérdida empleada en el entrenamiento combina las diferencias entre los valores predichos y reales para todos los parámetros de la predicción (x_i, y_i, w_i, h_i, C_i, p_i) para cada celda i de la imagen.

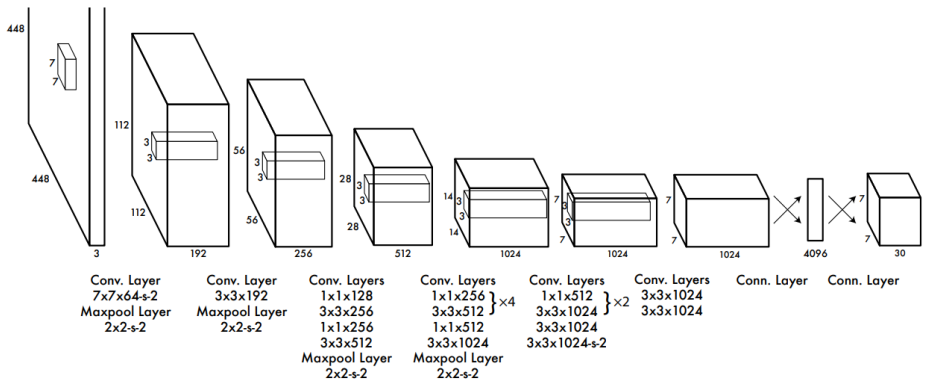


Figura 9: Esquema de la arquitectura de YOLO [14]

Trabajando sobre el *dataset* VOC 2007, YOLO es capaz de conseguir una mAP de 63.4% a 45 FPS. [14]

Estos valores son objetivamente peores que los obtenidos con el modelo SSD, publicado el mismo año, pero a diferencia de este, YOLO ha recibido constantes actualizaciones y nuevas versiones con el paso de los años, mejorando significativamente su rendimiento y sobre todo su versatilidad. [16], [17]

- YOLOv2: Lanzado en 2016, mejoró el modelo incorporando *batch normalization* (normalización de lotes), *anchor boxes* y *clusters* de dimensión, además de cambiar la arquitectura base a una red de 19 capas convolucionales, conocida como Darknet-19. Con esto se consigue mejorar los resultados hasta el 78.6% de mAP a 40 FPS, o 77.8% a 59FPS utilizando imágenes de entrada más pequeñas. [18]

- YOLOv3: Presentado en 2018, cambia la arquitectura de la red por una más grande y eficiente, la Darknet-53, de 53 capas convolucionales. También añade *anchors múltiples* y un *pooling* de pirámides espaciales, lo que le permite predecir *bounding boxes* a 3 escalas distintas, mejorando la precisión en la detección de objetos pequeños. [19]

- YOLOv4: Dado que YOLO es un proyecto *open source*, otros investigadores se han unido a la causa aportando sus mejoras para construir nuevas iteraciones del algoritmo. En este caso la versión 4 fue desarrollada y publicada en 2020 por el Instituto de Ciencias de la Información Academia Sinica, de Taiwan. Implementa mejoras como el uso de mosaicos en el entrenamiento, y algunas modificaciones en la arquitectura y funciones de coste para su mejor implementación en tarjetas gráficas (GPU). [20]
- YOLOv5: Fue desarrollado por Ultralytics, la empresa que se encarga principalmente del desarrollo de YOLO en la actualidad. Esta versión introduce mejoras de rendimiento, además de nuevas características como la optimización de hiperparámetros. Uno de los objetivos principales de esta actualización, además de las mejoras introducidas, es mejorar la accesibilidad y la utilización del algoritmo, integrándolo con PyTorch e implementándolo con entornos de desarrollo como Roboflow, ClearML, Comet o Neural Magic. [21]

- YOLOv6: Creado por el departamento de IA y Visión de la empresa china Meituan. Implementan ligeras mejoras, con el objetivo de adecuarlo para tareas industriales dentro de su empresa. [22]

- YOLOv7: Una versión actualizada de YOLO creada por los mismos autores que YOLOv4 [23]

2.4. YOLO v8

La versión más reciente de YOLO es la conocida como YOLOv8, un modelo de última generación lanzado en enero de 2023 que construye sobre las versiones anteriores del algoritmo e introduce nuevas mejoras, tanto de rendimiento como de versatilidad. Tan solo aplicando ligeras modificaciones en el entrenamiento y en las capas de salida, es posible utilizar la misma red para realizar tareas de Detección y seguimiento de objetos (incluso con *bounding boxes* orientadas), Clasificación de imágenes, Segmentación semántica y Estimación de posturas mediante *keypoints*.

Existen distintas arquitecturas de YOLOv8, de mayor o menor complejidad, que se deberán elegir en función de las características de la tarea a realizar, por ejemplo, si se quieren detectar muchas clases distintas de objetos a la vez o utilizar imágenes de entrada más grandes, será necesario emplear una arquitectura más compleja, que permita almacenar más información acerca de cada una de esas clases:

Modelo	tamaño (pixeles)	mAP^{val} 50-95	Velocidad CPU ONNX (ms)	Velocidad A100 TensorRT (ms)	parámetros (M)	FLOPs (B)
YOLOv8n	640	37.3	80.4	0.99	3.2	8.7
YOLOv8s	640	44.9	128.4	1.20	11.2	28.6
YOLOv8m	640	50.2	234.7	1.83	25.9	78.9

Modelo	tamaño (pixeles)	mAP^{val} 50-95	Velocidad CPU ONNX (ms)	Velocidad A100 TensorRT (ms)	parámetros (M)	FLOPs (B)
YOLOv8l	640	52.9	375.2	2.39	43.7	165.2
YOLOv8x	640	53.9	479.1	3.53	68.2	257.8

Figura 10: Tabla de arquitecturas de YOLOv8 [24]

Todas ellas comparten no obstante la misma estructura, condicionada por unos parámetros en función de la versión elegida. El modelo está dividido en 2 partes diferenciadas, cada una compuesta a su vez por distintos bloques de capas:

Backbone: Es la parte principal de la red. Alterna bloques Conv y C2f:

- Conv: Compuesto por una capa convolucional 2D seguida de una capa de *Batch Normalization 2D.* La función de activación que emplea es *SiLU*, $f(x) = x \cdot \sigma(x)$, donde $\sigma(x)$ es la función sigmoide.
- C2f: Contiene una capa convolucional, seguida de una o varias capas *Bottleneck*[25] que reducen la dimensionalidad del *feature map* extraído.
- SPPF: Combina capas convolucionales con capas de MaxPooling2D. Se sitúa al final del *backbone* como último bloque.

Head: Recibe las salidas de distintas etapas del *backbone* y las combina con una versión reescalada de la última salida. Con esto se consigue un trabajo simultáneo de los bloques de detección sobre tres escalas distintas.

Es la parte del modelo que se sustituye en función del trabajo a realizar. En este caso es la etapa en la que se realiza la detección de objetos. Para cada escala se realiza por separado la clasificación del objeto y la estimación de la *bounding box* correspondiente.

En el Anexo I se incluye un gráfico que detalla la arquitectura de YOLOv8.

Para la tarea de Tracking, que es complementaria a todas las anteriores excepto la de clasificación de imágenes, YOLO dispone de implementaciones de dos de los algoritmos más avanzados para llevar a cabo este trabajo:

- ByteTrack: Trata de recopilar información acerca del movimiento relativo entre la cámara y el objeto y con ello predice la posición de la *bounding box* en los siguientes fotogramas. Cuando llega el siguiente fotograma, calcula la IoU de la *bounding box* predicha respecto a la *bounding box* más cercana y si es lo suficientemente alto identifica ambas como detecciones del mismo objeto.

  ByteTrack destaca respecto a sus predecesores en situaciones en las que se producen oclusiones parciales o totales de los objetos a detectar, ya que no descarta las *bounding boxes* con índices de confianza bajos (que podrían deberse tanto a falsos positivos en la detección como a verdaderos positivos de objetos parcialmente ocluidos) si tiene una confianza alta de que estas pueden estar asociadas a objetos detectados en fotogramas anteriores.[26]

- BoT-SORT: Introduce mejoras sobre ByteTrack, especialmente para la tarea de seguimiento de personas. Entre otras, además de utilizar la predicción de la posición para relacionar objetos entre fotogramas, emplea un modelo de apariencias que realiza una reidentificación del objeto, confirmando que se trata del mismo que se detectó en el fotograma anterior.[27]

Al igual que YOLOv5, YOLOv8 está integrado en plataformas de desarrollo de proyectos de Machine Learning como Roboflow, ClearML, Comet, Neural Magic u OpenVINO. [24]

2.5. MÉTRICAS UTILIZADAS

En todo proceso de aprendizaje máquina en el que se construye un modelo a partir del entrenamiento con un *dataset*, es necesario, al final del mismo estimar la fiabilidad de las predicciones que ese modelo es capaz de realizar. Esta etapa denominada validación del modelo, se lleva a cabo mediante el cálculo de diferentes métricas que se adaptan a la categoría de problema que resuelve el modelo (regresión, clasificación, localización, detección, etc). En este apartado se mencionarán distintas métricas empleadas para la evaluación y comparación de los modelos empleados en detección de objetos. Estas métricas son:

- *Intersection over Union*: Es una métrica que representa la similitud entre *bounding boxes*, generalmente se utiliza para comparar las predichas por el modelo con las reales. Esta métrica se define como el cociente entre el área de intersección de las dos *bounding boxes* a comparar y el área de su unión.

$$IoU = \frac{Área\ de\ la\ intersección}{Área\ de\ la\ unión}$$

- Precisión: Hace referencia a la precisión en la clasificación de los objetos de una determinada clase. Se define como el cociente entre los verdaderos positivos de las detecciones del modelo entre el total de objetos detectados (verdaderos positivos y falsos positivos).

$$P = \frac{VP}{VP + FP}$$

- Exhaustividad o *Recall:* Se define como el cociente entre el número de detecciones correctas (verdaderos positivos) y el número real de objetos a detectar (verdaderos positivos y falsos negativos).

$$R = \frac{VP}{VP + FN}$$

- *Mean Average Precision:* A partir de las métricas anteriores podemos calcular la precisión media, o AP (*Average Precision*) para cada una de las clases.
  Se calcula la precisión asociada a la predicción de cada uno de los objetos de la clase, y se toma la media de todos ellos.
  Finalmente, tomando la media de estas AP de cada clase se obtiene la mAP.

$$mAP = \frac{1}{N}\sum_{i=1}^{N} AP_i$$

  Donde $i = 1, \dots, N$ hacen referencia a cada una de las clases.

  También existen dos versiones muy utilizadas de esta métrica, son mAP^{50} y $mAP^{50\text{-}95}$. Estas representan el *mean average precision* de los objetos detectados cuyas *bounding box* tengan un IoU > 0.50 y 0.50 < IOU < 0.95 respectivamente.

3. HERRAMIENTAS Y TECNOLOGÍAS

3.1. LENGUAJE DE PROGRAMACIÓN PYTHON

Python es un lenguaje de programación interpretado de alto nivel. Fue creado en 1991 por Guido van Rossum y uno de sus objetivos principales es la legibilidad, claridad y la utilización de una sintaxis simple.

Es un lenguaje que sirve tanto a programadores debutantes como expertos, gracias a su simplicidad y versatilidad. Existen multitud de librerías creadas para este lenguaje con objetivos que abarcan prácticamente todos los ámbitos de las ciencias de la computación. Manejo de bases de datos, creación de videojuegos, cálculos matemáticos complejos o, como es el caso de las utilizadas en este proyecto, tratamiento de imágenes y *Machine Learning*.

Por todo esto y por ser el lenguaje en el que está implementado YOLOv8 es por lo que se ha elegido para el desarrollo de la parte de programación de este proyecto.

En este proyecto se utilizará la versión 3.10.13 de Python en conjunto con las siguientes librerías:

- Ultralytics 8.1.5: Es la librería que contiene la implementación en Python de YOLOv8, además de algunas de las funciones auxiliares que se emplean en el proyecto, como son las de conteo de objetos.
- OpenCV 4.6.0: Librería de visión por ordenador y tratamiento de imágenes. Se utilizan las funciones cv2.VideoCapture() para la importación del archivo de video, cv2.imshow() para mostrar las imágenes/fotogramas por pantalla y cv2.VideoWriter(), para guardar los vídeos resultantes en la memoria del equipo.
- NumPy 1.26.2: Librería de operaciones matemáticas.
- PyTorch 2.1.1+ cu121: Librería de Machine Learning que permite el trabajo tanto en CPU como en GPU.
- Patchify 0.2.3: Librería que permite segmentar imágenes en cuadrículas con o sin superposición, permitiendo actuar sobre las distintas teselas de manera independiente.

- Multiprocessing: Es un paquete base de Python que permite la ejecución de varias funciones o trozos de código en paralelo, permitiendo acelerar en gran medida algunos procesos.

3.2. LIBRERÍAS DE PROCESAMIENTO EN GPU

A pesar de que la mayoría de los programas pueden implementarse sin problemas sobre una CPU, las GPU ofrecen una arquitectura más apropiada para el desarrollo de redes neuronales profundas y el trabajo con imágenes. Su gran cantidad de núcleos e hilos permiten la realización de tareas en paralelo, acelerando así los tiempos de ejecución.

Para la implementación de programas sobre tarjetas gráficas Nvidia, como es la empleada este proyecto, se utilizan las siguientes librerías:

- CUDA 11.2: *Compute Unified Device Architecture* es una plataforma de computación en paralelo que permite la programación y codificación de algoritmos para su ejecución en. tarjetas gráficas (GPU) de Nvidia.
- cuDNN 8.1.0: Es una librería de primitivas con aceleración por GPU para redes neuronales profundas (*Deep Neural Networks*). Incluye implementaciones de algunos de los procesos básicos necesarios para el entrenamiento y desarrollo de redes neuronales, como convoluciones, *pooling* o normalización.

Sobre estos dos recursos trabaja la versión específica para GPU de PyTorch, la librería de Python mediante la cual se implementa YOLOv8.

Figura 11: Software empleado: Python, Ultralytics, OpenCV, NumPy, PyTorch, CUDA

3.3. HARDWARE EMPLEADO

Para la realización del proyecto se han utilizado los siguientes equipos:

Captura de datos:

- DJI Mini 3 Pro: Dron de clase C0

  Especificaciones de la cámara:

  - Sensor CMOS de 1/1.3” de 48 MP
  - Campo de visión 82.1º, apertura f/1.7
  - Rango ISO: 100-6400
  - Resolución de vídeo:
    - 4K: 3840×2160 a 24/25/30/48/50/60 fps
    - 2.7K: 2720×1530 a 24/25/30/48/50/60 fps
    - FHD: 1920×1080 a 24/25/30/48/50/60 fps
    - Slow Motion: 1920×1080 a 120 fps

Figura 12: Dron utilizado: DJI Mini 3 Pro. Fuente: store.dji.com

Entrenamiento del modelo y detección:

- HP ENVY TE02-0000ns

  Especificaciones:

  - Procesador: Intel i5-12400F (2.50GHz)
  - Memoria RAM: 16GB 3200MHz DDR4 (2x8GB)
  - GPU: NVIDIA GeForce RTX 3060 12GB

4. DISEÑO DEL DETECTOR DE OBJETOS

4.1. ELECCIÓN DEL MÉTODO DE DETECCIÓN DE OBJETOS

Para este proyecto se ha elegido el modelo YOLOv8, el más versátil de los existentes, empleando la arquitectura YOLOv8n, la más ligera de las que ofrece, ya que sólo se necesitará detectar una única clase de objetos, manzanas, y permite la implementación del modelo en equipos menos potentes.

Se utilizará la versión de *Track*, que permite la identificación y seguimiento individual de cada una de las manzanas, asignándoles identificadores individuales, lo cual permite el conteo. Se utilizará el algoritmo ByteTrack, puesto que en el caso concreto de las manzanas son frecuentes las oclusiones con hojas, ramas u otras manzanas y los algoritmos de reidentificación por apariencia como el que utiliza BoT-SORT son poco útiles, dada la similitud del aspecto de las distintas manzanas.

4.2. PREPROCESAMIENTO DE DATOS

El primer paso en cualquier trabajo de Machine Learning supervisado es la creación de un conjunto de datos etiquetados, que servirá para el entrenamiento y validación del modelo.

Una vez se hayan obtenido los datos de partida se debe realizar una etapa de preprocesamiento, en la que se adaptarán a las necesidades del trabajo y del modelo utilizado.

Dado que la captura y etiquetado de estos datos es generalmente un proceso costoso en tiempo y recursos, se suele introducir una etapa de aumento de datos (*data augmentation*) en el preprocesamiento, multiplicando de forma sintética el tamaño del conjunto de datos.

El aumento del conjunto de datos, en el caso concreto de un *dataset* de imágenes, consiste en duplicar algunas de estas imágenes y aplicar ciertas modificaciones y transformaciones, como

pueden ser aumentos o reducciones en contraste, brillo, saturación o tono; reflejado y rotación de las imágenes, recortes o generación de oclusiones parciales, composición de mosaicos a partir de distintas imágenes.

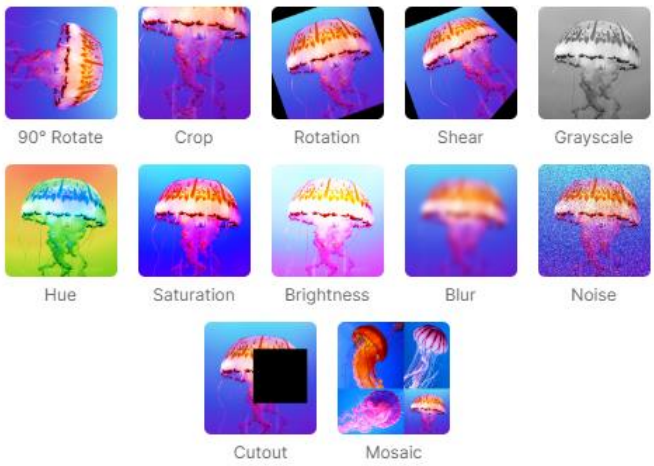


Figura 13: Aumentados posibles mediante Roboflow [28]

4.3. ARQUITECTURA DEL MODELO DE RED NEURONAL

Dado que en este proyecto solamente se requiere la detección de una única clase y se pretende la mayor eficiencia posible del modelo, priorizando su posible utilidad en tiempo real ante su precisión en la detección, la arquitectura elegida para este proyecto de detección de manzanas será la YOLOv8n, la más pequeña de las que ofrece la librería.

Esta versión está compuesta por 225 capas, con un total de 3.011.043 parámetros.

```
                 from  n    params  module                                       arguments
  0                -1  1       464  ultralytics.nn.modules.conv.Conv             [3, 16, 3, 2]
  1                -1  1      4672  ultralytics.nn.modules.conv.Conv             [16, 32, 3, 2]
  2                -1  1      7360  ultralytics.nn.modules.block.C2f             [32, 32, 1, True]
  3                -1  1     18560  ultralytics.nn.modules.conv.Conv             [32, 64, 3, 2]
  4                -1  2     49664  ultralytics.nn.modules.block.C2f             [64, 64, 2, True]
  5                -1  1     73984  ultralytics.nn.modules.conv.Conv             [64, 128, 3, 2]
  6                -1  2    197632  ultralytics.nn.modules.block.C2f             [128, 128, 2, True]
  7                -1  1    295424  ultralytics.nn.modules.conv.Conv             [128, 256, 3, 2]
  8                -1  1    460288  ultralytics.nn.modules.block.C2f             [256, 256, 1, True]
  9                -1  1    164608  ultralytics.nn.modules.block.SPPF            [256, 256, 5]
 10                -1  1         0  torch.nn.modules.upsampling.Upsample         [None, 2, 'nearest']
 11            [-1, 6]  1         0  ultralytics.nn.modules.conv.Concat           [1]
 12                -1  1    148224  ultralytics.nn.modules.block.C2f             [384, 128, 1]
 13                -1  1         0  torch.nn.modules.upsampling.Upsample         [None, 2, 'nearest']
 14            [-1, 4]  1         0  ultralytics.nn.modules.conv.Concat           [1]
 15                -1  1     37248  ultralytics.nn.modules.block.C2f             [192, 64, 1]
 16                -1  1     36992  ultralytics.nn.modules.conv.Conv             [64, 64, 3, 2]
 17           [-1, 12]  1         0  ultralytics.nn.modules.conv.Concat           [1]
 18                -1  1    123648  ultralytics.nn.modules.block.C2f             [192, 128, 1]
 19                -1  1    147712  ultralytics.nn.modules.conv.Conv             [128, 128, 3, 2]
 20            [-1, 9]  1         0  ultralytics.nn.modules.conv.Concat           [1]
 21                -1  1    493056  ultralytics.nn.modules.block.C2f             [384, 256, 1]
 22      [15, 18, 21]  1    751507  ultralytics.nn.modules.head.Detect           [1, [64, 128, 256]]
Model summary: 225 layers, 3011043 parameters, 3011027 gradients, 8.2 GFLOPs
```

Figura 14: Arquitectura de YOLOv8n

4.4. ELECCIÓN DE PARÁMETROS Y CONFIGURACIÓN DEL ENTRENAMIENTO

Para el entrenamiento del modelo se partirá de un archivo de pesos preentrenados para detección de objetos sobre el *dataset* COCO (cocodataset.org), disponible en la web de Ultralytics.

Comenzar el entrenamiento basándose en un modelo preentrenado sobre un conjunto de datos más grande y genérico suele ser beneficioso, ya que da un punto de partida más avanzado que si se inicializara con pesos aleatorios. La red preentrenada ya es capaz de reconocer ciertos objetos y patrones, y sólo hace falta "educarla" para que reconozca los del caso concreto en el que se quiera aplicar, los ajustes que se deben hacer a los pesos son menos y más finos y por tanto es menos costoso computacionalmente. Esto reduce los tiempos de entrenamiento, así como la dimensión del conjunto de datos necesario para entrenar la red. Este proceso es habitual en el uso de las redes neuronales y se denomina de forma general "Transfer Learning".

5. IMPLEMENTACIÓN Y ENTRENAMIENTO DEL MODELO

5.1. RECOPILACIÓN Y ETIQUETADO DEL CONJUNTO DE DATOS

El conjunto de datos utilizado ha sido creado específicamente para este proyecto.

Para ello se han tomado distintos videos y fotografías en un manzanal tradicional asturiano (pumarada) situada en el concejo de Siero utilizando un dron DJI Mini 3 Pro. Se capturaron una serie de secuencias de video y fotografías en las que se muestran distintos árboles, en formato apaisado y vertical y a distintas resoluciones.

Figura 15: Captura de imágenes en campo con dron

En estos videos se trató de capturar la mayor variedad posible de condiciones, tanto de iluminación como de distribución de manzanas en el árbol, color de las manzanas, etc., con el fin de conseguir un *dataset* que permita entrenar un modelo robusto a todas estas variables.

Utilizando la herramienta online Roboflow, se extrajeron varios fotogramas de cada vídeo, eligiéndose los más representativos y comprobando que fueran lo más distintos posible entre sí.

El siguiente paso es la definición de las clases de objetos que aparecerán en el *dataset*, en este caso solamente existe una, Manzanas, puesto que el objetivo fundamental es la detección y conteo del número de estos frutos en el propio árbol.

El etiquetado de las imágenes consiste en la identificación manual de los objetos en cada uno de los fotogramas, trazando para cada uno de ellos una *bounding box* que lo contenga. Es importante no dejar objetos de la clase que se quiera detectar sin marcar, ya que esto confundirá a la red durante el entrenamiento.

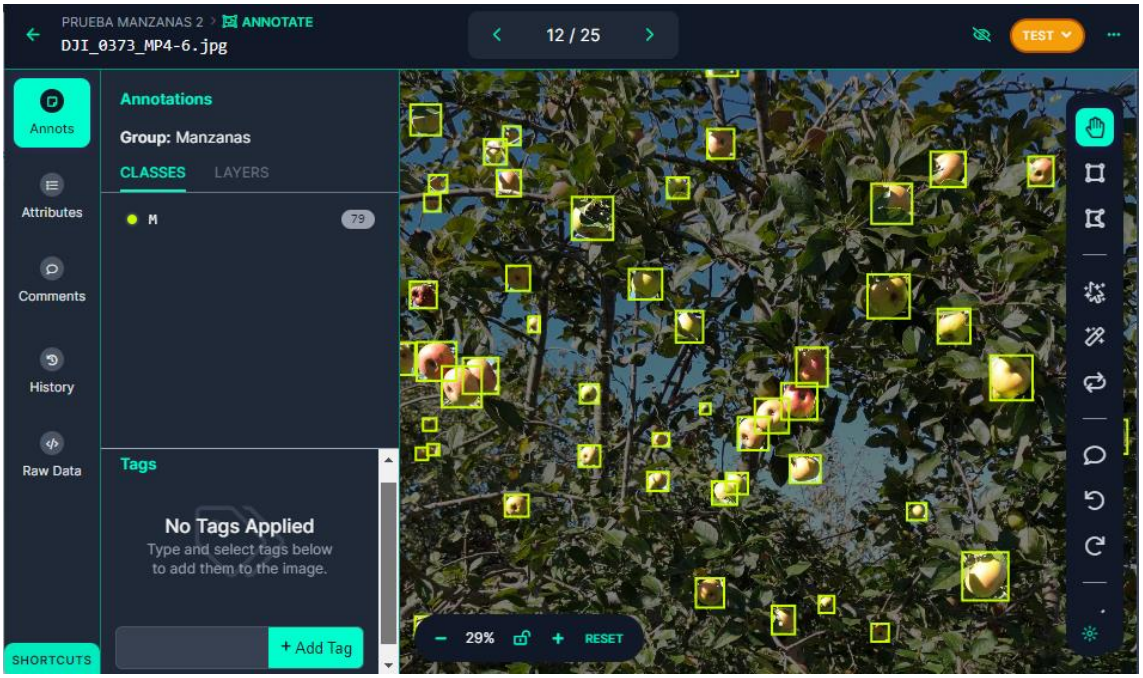


Figura 16: Interfaz de anotación de Roboflow

5.2. PROCESAMIENTO Y AUMENTADO DE LOS DATOS

El etiquetado de las imágenes es un proceso largo y tedioso, ya que se pretende conseguir un conjunto de imágenes lo más amplio posible y es un trabajo que se debe realizar manualmente.

Como ya se mencionó en el apartado 4.2, mediante procesos de aumentado podemos multiplicar el tamaño de del *dataset*, haciéndolo incluso más robusto a la variación de las condiciones y aspecto de las imágenes.

Previo a este paso se hace también un preprocesamiento de los datos, para adaptarlos a los formatos que utiliza de forma nativa el modelo y conseguir un conjunto de datos apropiado para este tipo de trabajos.

En el caso de este proyecto se han utilizado tres videos como fuente, dado el tamaño de los fotogramas, de 3840x2160 píxeles en uno de los videos y de 2688x1512 en los otros, se han dividido siguiendo una cuadrícula de 6x3 y 4x2 respectivamente, de modo que de cada fotograma del primer video se extraen 18 imágenes de 640x720, y de cada uno de los segundos, 8 imágenes de 672x756. Estas imágenes se reescalan a 640x640 ya que es el tamaño de entrada que utiliza la arquitectura de YOLOv8.

También se eliminan algunas de estas imágenes que correspondan a *background*, es decir, que no contengan ningún objeto de los que se pretende detectar. Es conveniente de todos modos dejar algún ejemplo de este tipo para que el modelo también aprenda las características del fondo de las escenas y pueda distinguirlo de los objetos que se pretenden detectar. En este caso se ha configurado para que al menos el 66% de las imágenes del conjunto de datos contengan algún objeto.

En el conjunto de datos que se utiliza en este proyecto, se han aplicado los siguientes aumentos:

- Rotación entre -10º y +10º
- Exposición entre -10% y +10%
- Difuminado gaussiano con radio hasta 2.5 píxeles

Tras realizar estos pasos, partiendo de un conjunto de 32 fotogramas etiquetados se ha obtenido un *dataset* con un total de 1142 imágenes.

El conjunto de datos se ha dividido en los tres lotes o *splits* tradicionales propios de los trabajos de *machine learning*: Entrenamiento, Validación y Test, destinando 1044 imágenes al primer conjunto, y 62 y 36 respectivamente a los otros dos.

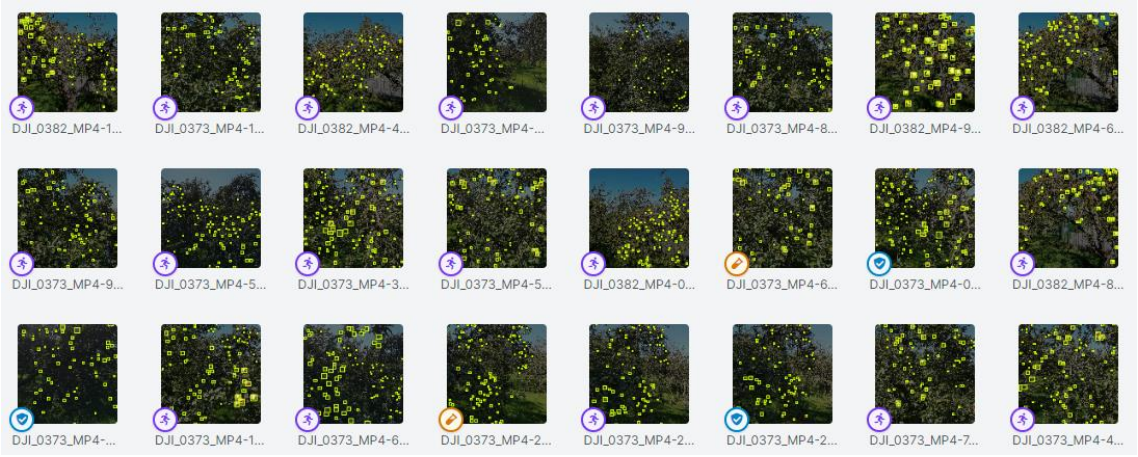


Figura 17: Separación de las imágenes en conjuntos de Entrenamiento, Validación y Test

El último paso es exportar el *dataset* para poder cargarlo en el script de entrenamiento. La plataforma de Roboflow permite seleccionar el formato de anotación preferido para los datos. En el caso de este trabajo se seleccionará el formato YOLOv8, compatible con el modelo utilizado. Este formato genera tres carpetas, para los conjuntos de entrenamiento (*train*), validación (*valid*) y test, dos archivos README con información acerca del conjunto de datos, como los parámetros de aumentado empleados, y un archivo *data.yaml*, que es el que se debe introducir en el script para que la función de entrenamiento cargue los archivos.

En cada una de las carpetas de entrenamiento, validación y test se encuentran dos subcarpetas: *images* y *labels*, que como sus nombres indican, contienen respectivamente las imágenes en formato .jpg y las etiquetas asociadas a cada una de ellas. Estas etiquetas se presentan en archivos de formato .txt, que contienen una tabla con cinco columnas. La primera hace referencia al identificador de clase, y las siguientes a las coordenadas (*x,y*) del centro de la *bounding box* y a los parámetros (*w, h*), ancho y alto de las mismas. Estos parámetros y coordenadas están expresados en coordenadas imagen normalizadas.

```
0       0.20859375      0.34921875      0.1015625       0.0609375
0       0.55625         0.4546875       0.065625        0.071875
0       0.22578125      0.6359375       0.10625         0.1109375
0       0.17578125      0.728125        0.1234375       0.1171875
0       0.3765625       0.5703125       0.0703125       0.0828125
0       0.32890625      0.540625        0.0640625       0.08125
0       0.7546875       0.6625          0.11875         0.115625
0       0.85546875      0.58046875      0.090625        0.1015625
```

Figura 18: Contenido del archivo "labels" asociado a una imagen.

Col1: id_clase, col2: "x_centro", col3: "y_centro", col4: "h_box", col5: "w_box"

5.3. ENTRENAMIENTO DEL MODELO DE DETECCIÓN DE OBJETOS

El entrenamiento se ha realizado empleando la función integrada de la clase YOLO de Ultralytics. Se han utilizado los siguientes parámetros:

- Nº de épocas: 300
- Paciencia: 50 (número de épocas que debe esperar la función de entrenamiento sin detectar mejoras significativas en las métricas para detener de forma temprana el entrenamiento)

El resto de los parámetros se han dejado en sus valores por defecto, ya que proporcionaron resultados suficientemente satisfactorios. Algunos de estos parámetros y valores son:

Parámetro	**Valor**	**Descripción**
batch	16	Number of images per batch
lr0	0.01	initial learning rate
lrf	0.01	final learning rate (lr0 * lrf)
momentum	0.937	SGD momentum/Adam beta1
weight_decay	0.0005	optimizer weight decay
warmup_epochs	3.0	warmup epochs

warmup_momentum	0.8	warmup initial momentum
warmup_bias_lr	0.1	warmup initial bias lr
box	7.5	box loss gain
cls	0.5	cls loss gain (scale with pixels)
dfl	1.5	dfl loss gain
label_smoothing	0.0	label smoothing
nbs	64	nominal batch size

Figura 19: Parámetros YOLOv8 [24]

Durante el entrenamiento, la función utiliza los conjuntos de validación y entrenamiento para realizar pruebas de detección y determinar el estado del modelo.

Para ello utiliza las métricas: [29]

- Box Loss: Representa la diferencia, expresada en términos de la IoU, entre las *bounding box* predichas por el modelo y las reales.
- Classification Loss: Representa la diferencia entre la clase predicha por el modelo y la real de cada objeto (en este caso manzana o fondo).
- Distribution Focal Loss: Es otra función de pérdida que hace referencia a la posición de las Bounding boxes, complementa a la *Box Loss.*
- Precision: Proporción de detecciones correctas respecto al total de detecciones.
- Recall: Proporción de detecciones de objetos respecto a los objetos reales que existen.
- mAP50: *Mean Average Precision* de los objetos detectados con IoU > 0.50.
- mAP50-95: *Mean Average Precision* de los objetos detectados con 0.5 < IoU < 0.95.

En esta fase se han entrenado distintas arquitecturas con los mismos parámetros y sobre el mismo *dataset*. Los resultados obtenidos en la validación de los modelos han sido los siguientes:

Modelo	Épocas	Precisión	Recall	mAP50	mAP50-95
YOLOv8n	55	0.784	0.802	0.857	0.576
YOLOv8s	55	0.791	0.808	0.859	0.592
YOLOv8l	41	0.816	0.766	0.852	0.594

Figura 20: Resultados del entrenamiento de diferentes arquitecturas de YOLOv8

Dado que los resultados no muestran una mejora significativa en los valores de las métricas para modelos más complejos, se elige el modelo YOLOv8n, el más sencillo de todos, ya que será también más rápido en la inferencia. El entrenamiento se realiza mediante la siguiente función:

```
    model = YOLO('yolov8n.pt')

    results = model.train(data='Dataset/data.yaml', epochs=300, imgsz=640, patience=50,
verbose=True, device='cuda', plots=True)
```

Se puede observar la progresión de los valores de las métricas a lo largo del entrenamiento en las gráficas que genera la propia función de entrenamiento:

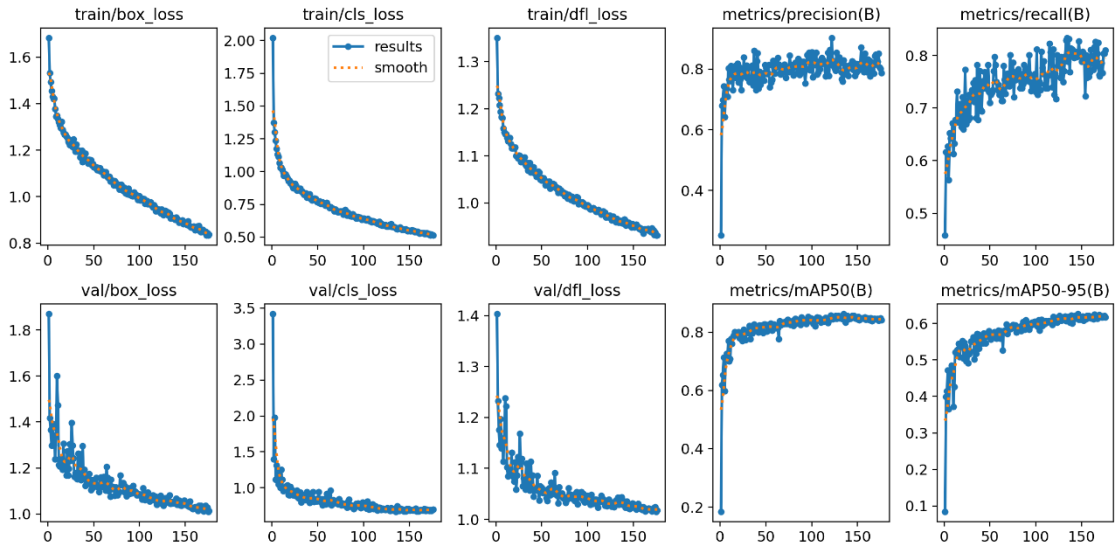


Figura 21: Gráficas de las métricas del entrenamiento para YOLOv8n.

box_loss: "Box loss", cls_loss: "Classification loss", dfl_loss: "Distribution Focal Loss"

También genera una serie de gráficas que dan una idea de las relaciones entre precisión, *recall* y confianza:

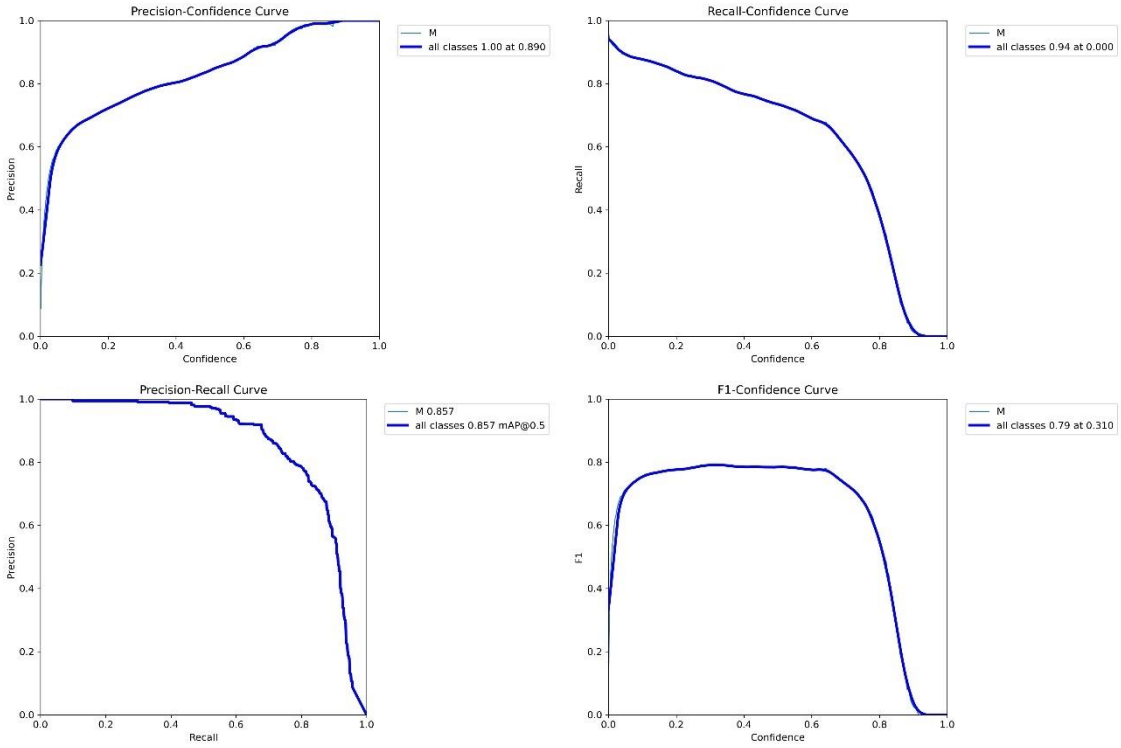


Figura 22: Gráficas de Precision, Recall y F1-score

Quizás la curva más interesante de las mostradas es la del F1-score – Confianza. F1-score es una métrica que resume en un solo valor la precisión y *recall*. Combinada con la Confianza da una idea de la confianza mínima a partir de la cual deberíamos filtrar los elementos detectados para obtener un mejor balance de objetos detectados vs objetos existentes.

El F1-score se calcula de la siguiente manera:

$$F_1 = 2 \cdot \frac{precision \cdot recall}{precision + recall}$$

La función de entrenamiento genera también un archivo de pesos en formato *.pt*, que será el que se cargue en el script de inferencia para utilizar el modelo entrenado.

5.4. IMPLEMENTACIÓN, DETECCIÓN Y CONTEO

Con el modelo ya entrenado, el siguiente paso es la implementación para su uso en la inferencia sobre datos reales.

En este proyecto se realizará una implementación en Python en forma de una función, cuentaManzanas, que incorpora dos sistemas de conteo. La función recibe como entrada un video, la dirección en la que se deben contar las manzanas en el primero de los métodos y el tamaño de las teselas en las que se desea dividir.

Esta función abre el archivo de video que se le haya indicado en la ruta introducida y lee el primer fotograma, a partir del cual obtiene la resolución del video. En función de las dimensiones y el tamaño de tesela que se haya introducido, calcula el número de teselas en las que dividirá el video, utilizando las funciones de la librería *patchify*.

```
def cuentaManzanas(video_path, direccion="H", tamanyo_tesela=640):

    # Abre el archivo de video

    cap = cv2.VideoCapture(video_path)

    # Lee el primer fotograma del video, con él determina la resolución y por tanto el
numero de teselas en los que se debería dividir el video para su procesamiento

    success, frame1 = cap.read()

    cap.release()

    # Si el tamaño de tesela introducido es mayor que las dimensiones del video, no se
subdivide. En caso contrario, se dividirá en teselas del tamaño especificado

    tamanyo_parche = min(tamanyo_tesela, max(frame1.shape))

    print('tamaño del frame: '+ str(frame1.shape))

    parches_inicio = patchify(frame1, (tamanyo_parche, tamanyo_parche,3),
step=tamanyo_parche)

    print('tamanio grupo de parches: '+str(parches_inicio.shape))
```

El primer método de conteo será mediante una franja en el medio de la tesela. Cada vez que una *bounding box* de una manzana cruce esta línea será contabilizada. Para ello el primer paso será determinar los puntos que definen la línea que servirá como frontera para el conteo.

Por defecto esta línea cruza el *frame* de arriba abajo por la mitad del mismo (de derecha a izquierda si la dirección de conteo seleccionada es la vertical).

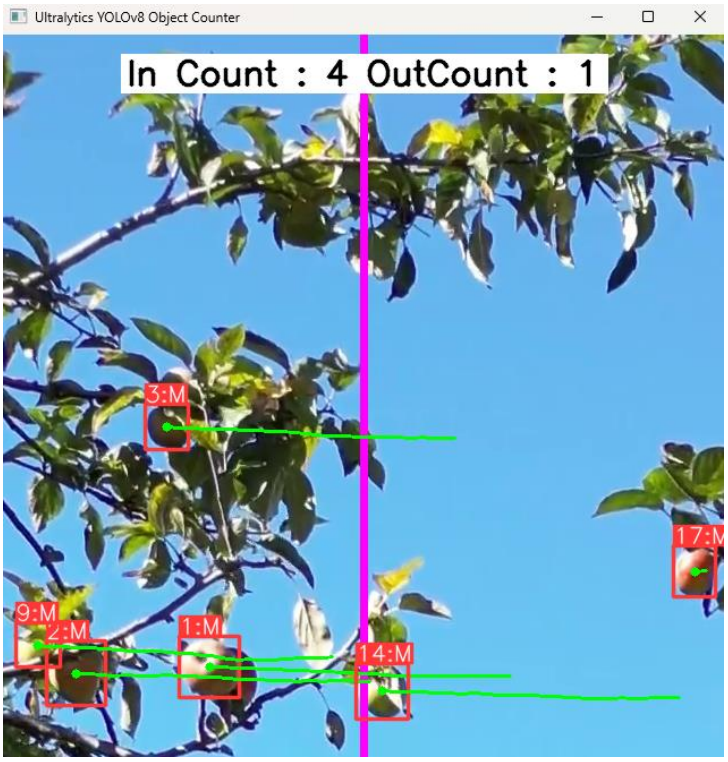


Figura 23: Ventana de visualización con el contador de manzanas

El segundo método se basará en los ID que genera el método *Track* del modelo YOLO. Este funciona asignando un identificador único a cada nuevo elemento, en este caso manzana, que aparezca en pantalla. De esta forma se pueden contabilizar las manzanas sin necesidad de que estén siendo detectadas en el momento de pasar por la línea, haciéndolo más robusto frente a problemas de oclusiones. En condiciones ideales ambos métodos de conteo deberían dar el mismo resultado.

Para ejecutar el programa de forma más eficiente aprovechando mejor los recursos disponibles del equipo, se realizará la detección en cada frame por separado, en procesos paralelos.

Por cada tesela en el eje elegido se generará un proceso mediante la librería *multiprocessing* de Python. En cada uno de estos procesos se ejecutará la función run_tracker, que toma como entradas la ruta del video, el índice correspondiente a la tesela, los puntos de la frontera de conteo, el tamaño del parche, la cola de salidas, la dirección elegida y un valor booleano que indica si se quiere que se muestren los videos por pantalla o no. La salida de esta función será el número de manzanas que se hayan contado en la tesela sobre la que se está trabajando.

```
    # Se inicializan la lista de procesos y la cola de salidas que almacenará los
resultados de cada proceso

    procesos = []

    output_queue = Queue()

    mostrar=True

    start = time.time()

    if direccion == "H":

        # Se determina la línea que servirá como frontera para el contador, en función
de la dirección del video elegida, esta será horizontal o vertical

        region_points = [(int(tamanyo_parche/2), 0), (int(tamanyo_parche/2),
tamanyo_parche)]

        eje = 0

    else:

        region_points = [(0, int(tamanyo_parche/2)), (tamanyo_parche,
int(tamanyo_parche/2))]

        eje = 1

    # Se generará un proceso de ejecución paralela por cada tesela que haya a lo largo
del eje elegido

    for i in range(parches_inicio.shape[eje]):

        proceso = Process(target=run_tracker, args=(video_path, i, region_points,
tamanyo_parche, output_queue, eje, mostrar))

        # Se inicializan los procesos y se añaden a la lista de procesos en marcha

        proceso.start()

        print(f"Comenzado el thread {i}")

        procesos.append(proceso)
```

En la función run_tracker se importa el modelo previamente entrenado mediante el archivo de pesos, se inicializa el contador de objetos y se abre el video a través de la ruta indicada.

```
def run_tracker(video_path, indice, region_points, tamanyo_parche, output_queue, eje=0,
mostrar=False):

    # Carga del modelo de YOLOv8 y su correspondiente archivo de pesos

    model = YOLO('best_yolov8n.pt')

    if eje:

        j=indice

        indice = 0

    else:

        j=0

    # Se inicializa el contador de objetos

    counter = object_counter.ObjectCounter()
```

```
    counter.set_args(view_img=mostrar, reg_pts=region_points, classes_names=model.names,
draw_tracks=True)

    cap = cv2.VideoCapture(video_path)

    lista_id=[]
```

En cada fotograma del vídeo se realizarán los siguientes pasos:

```
while cap.isOpened():

        # Se extrae cada frame del video

        success, frame = cap.read()

        # Cuando se alcance el final del video, success será False y se terminará la
ejecución

        if success:
```

Se divide el fotograma en el número de teselas correspondiente según el tamaño de tesela introducido

```
            parches = patchify(frame, (tamanyo_parche,tamanyo_parche,3),
        step=tamanyo_parche)
```

Se almacena temporalmente en una variable la tesela que se encuentra en la posición correspondiente al proceso que se está ejecutando.

```
            parche_simple = parches[indice,j,0,:,:,:]
```

Se realiza la inferencia mediante la función model.track() y se almacenan los resultados.

```
            results = model.track(parche_simple, tracker="bytetrack.yaml",
            persist=True)

            if results[0].boxes.id != None:

                    boxes = results[0].boxes.xywh.cpu().numpy().astype(int)

                    track_ids = results[0].boxes.id.cpu().numpy().astype(int)

                    lista_id.append(track_ids)
```

Estos resultados junto con la imagen original se introducen a la función contadora para que haga su trabajo.

```
            frame = counter.start_counting(np.ascontiguousarray(parche_simple),
results)
```

Cuando la secuencia de vídeo termina se cierra la ventana de reproducción y se extrae de la clase *counter* el número de manzanas contadas. Para hallar el total se restan las leídas de izquierda a derecha de las leídas de derecha a izquierda (o al revés, dependiendo de la dirección del vídeo) para compensar las que se hayan contado dos veces.

El número de manzanas de cada método de conteo se añade a la cola de salidas *output_queue*.

```
        cap.release()
        cuenta_track = lista_id[-1].item(-1)
        in_c = counter.in_counts
        out_c = counter.out_counts
        cuenta_counter = in_c - out_c
        output_queue.put([cuenta_counter, cuenta_track])
```

Cuando todos los procesos han terminado se recogen los valores obtenidos de la cola de salidas y se suman para dar como resultado el número total de manzanas que se observaron en el vídeo.

(de nuevo en cuentaManzanas())

```
    indice_proceso= 0
    resultados = []
    for proceso in procesos:
        # Se espera a que finalicen los procesos
        proceso.join()
        print(f"Finalizado el thread {indice_proceso}")
        # Cuando termina un proceso se recogen los resultados de la cola de ejecucion
        #y se almacenan en la lista resultados
        resultado = output_queue.get()
        resultados.append(resultado)
        indice_proceso +=1
    # Se cierran todas las ventanas de video abiertas y se muestran los resultados por
pantalla
    cv2.destroyAllWindows()
    total_conteo = sum(item[0] for item in resultados)
    total_track = sum(item[1] for item in resultados)

    print("Resultados por tesela:", resultados)
    print("\n\nSuma conteo:", total_conteo)
    print("Suma track:", total_track)
```

```
    end = time.time()
    print(f"\nTiempo total: {end - start}")


    return total_track, total_conteo
```

Para un video de 3840x2160 píxeles y 16 segundos, utilizando teselas de 640 pixeles se ha tardado aproximadamente 40 segundos en completar la ejecución del programa. Esto no permite el trabajo en tiempo real pero sí un procesamiento a posteriori suficientemente rápido para la mayoría de las aplicaciones.

Para hacer test de los métodos de conteo de manzanas se han utilizado varios videos:

- Un video sintético de resolución 1920x1080 en el que fotografías de manzanas sin ningún tipo de oclusión se desplazan de derecha a izquierda por la pantalla.
- Un video grabado en la plantación de manzanos en el que el dron se desplaza lateralmente a una altura fija, capturando una sección horizontal de un árbol. Tiene una resolución de 3840x2160.

En el caso del primer video, ambos sistemas de conteo han indicado correctamente la cantidad de manzanas que aparecían a lo largo del video, 70.

En el caso del segundo video, se ha determinado por inspección visual la presencia de aproximadamente 190 manzanas a lo largo de la secuencia. Los métodos de conteo han ofrecido los siguientes resultados: Mediante tracking: 209 manzanas. Mediante contador: 51

A la vista de los resultados, se puede confirmar que el método de tracking es más robusto y preciso en el conteo de objetos.

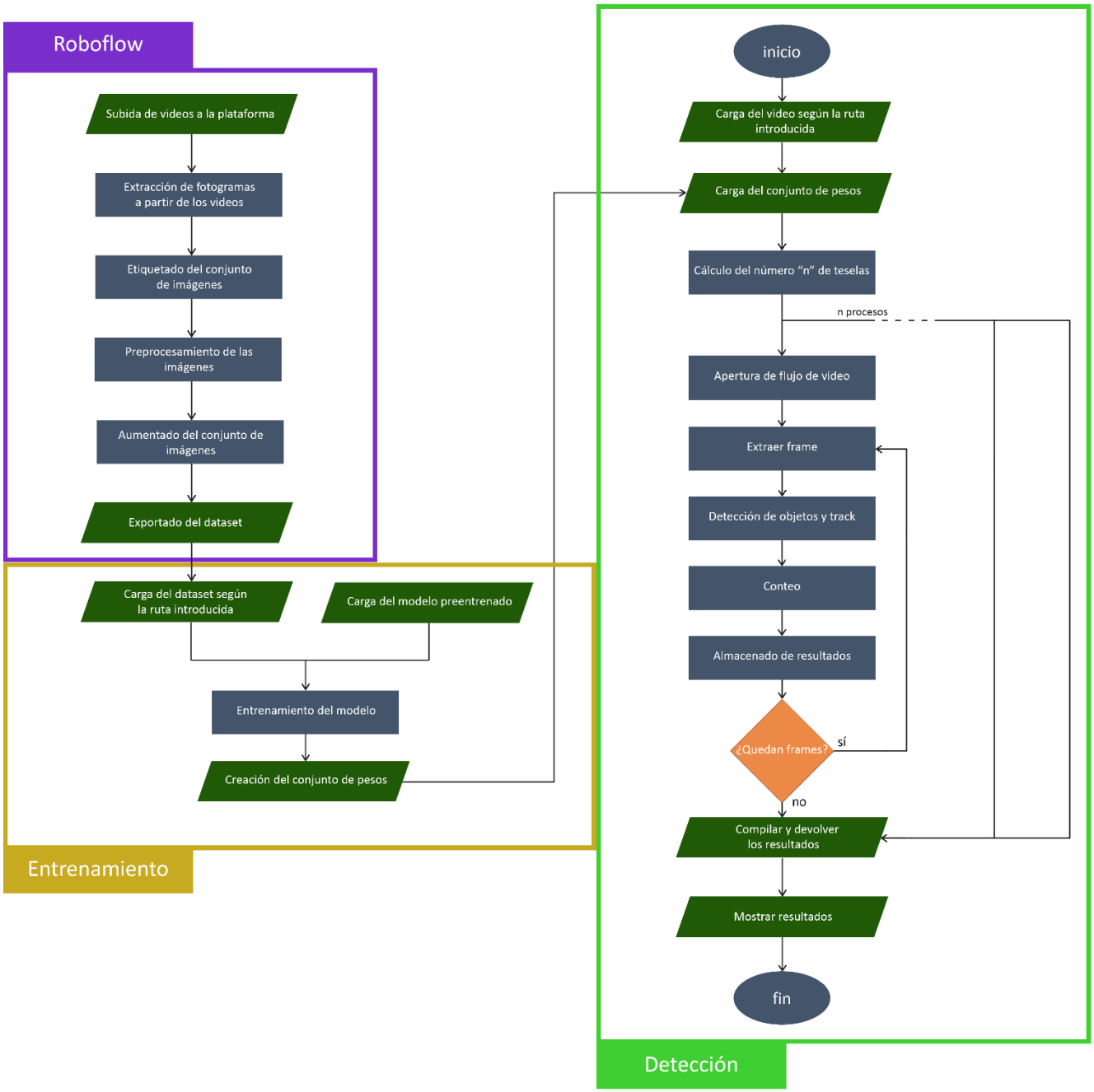


Figura 24: Diagrama de flujo del proyecto

6. CONCLUSIONES Y TRABAJO FUTURO

6.1. RESUMEN DE LOS HALLAZGOS Y LOGROS DEL PROYECTO

En este trabajo se han repasado los fundamentos y la historia de los detectores de objetos en imágenes, describiendo los avances realizados por distintos investigadores y comentando la situación actual de este conjunto de tecnologías. Se ha detallado la estructura y funcionamiento de la arquitectura YOLOv8, así como demostrado su versatilidad de cara a su aplicación en la realización de distintas tareas.

Con este proyecto se ha demostrado la viabilidad del desarrollo de una herramienta de detección de objetos en flujos de video basada Inteligencia Artificial para el aforo de cultivos de fruta en árbol. Esta herramienta utiliza un hardware comercializado públicamente y al alcance de la mayoría, por lo que supone un avance hacia la democratización de la tecnología, especialmente en un sector fundamental para la sociedad como es el de la producción agraria.

Si bien es verdad que la mayoría de los resultados obtenidos son positivos y prometedores de cara al futuro desarrollo e implementación de estas tecnologías, también se han encontrado una serie de limitaciones que se exponen en el apartado siguiente.

6.2. LIMITACIONES DEL ESTUDIO

A lo largo del proyecto se han identificado diferentes limitaciones y problemas, debidos tanto a la naturaleza de la problemática tratada como a la de los métodos utilizados.

En primera instancia, la plantación escogida para la toma de datos no es quizá la más apropiada para este tipo de soluciones, ya que es de estilo tradicional, presentando irregularidades en su topografía y distribución de los árboles sobre el terreno.

Árboles que a su vez también resultan problemáticos por sus grandes dimensiones, siendo difícil capturarlos en su totalidad en una sola toma de video. Además, debido a su gran radio, presentan manzanas a distintas distancias de la cámara, en muchas ocasiones muy lejanas o tapadas por hojas, ramas u otras manzanas. Esto es un problema ya que las manzanas más alejadas aparecerán más pequeñas en las imágenes capturadas y sufrirán oclusiones frecuentes, dificultando mucho su detección y seguimiento entre fotogramas.

Otras de las limitaciones encontradas son debidas al modelo empleado. A pesar de ser el más avanzado del mercado, sufre cuando hay una gran cantidad de objetos en cada imagen o estos son muy pequeños. Esto puede mitigarse reduciendo el tamaño de las teselas en las que se subdivide la imagen, pero genera problemas a la hora de detectar objetos grandes o en los bordes de las teselas.

Este tipo de sistemas funcionarían mejor en plantaciones más estructuradas, como son las que emplean técnicas como el cultivo en espalderas. En este tipo de plantaciones los árboles se disponen en hileras paralelas, creciendo enredadas en una especie de vallado que les sirve como guía. Con esto se consigue que la profundidad de las plantas sea mínima, centrándose todo el crecimiento de la planta en dos dimensiones: alto y ancho. En este tipo de cultivos las oclusiones por hojas y ramas serían mínimas, dejando ver con más claridad las manzanas y facilitando detección. También se facilitaría las pasadas de cámara lineales siguiendo estas hileras, un formato muy cómodo para la utilización de las estrategias de conteo desarrolladas en este proyecto ya que se reducen o incluso eliminan las dobles pasadas por una misma zona, desapareciendo así el problema de la duplicidad de las detecciones.

También es una limitación difícil de mitigar la gran potencia computacional que este tipo de algoritmos requieren para su funcionamiento. En un equipo como el indicado en el apartado 3.3 es posible realizar la inferencia en tiempo real sobre un vídeo de 640x640 píxeles, pero para trabajar con resoluciones de video mayores se deberían ignorar algunos fotogramas, dificultando e incluso llegando a imposibilitar el seguimiento de los objetos en las imágenes.

De todos modos, en un equipo como el mencionado se dispone de la suficiente potencia de cómputo como para poder realizar la detección a posteriori en un tiempo muy breve, suficiente para la mayoría de las aplicaciones.

6.3. PROPUESTAS PARA TRABAJOS FUTUROS

Algunos trabajos futuros que podrían derivarse del presente podrían ser la expansión del modelo a otras variedades de manzana u otros tipos de fruta, entrenando el modelo para que sea capaz de diferenciarlos en distintas clases. También se podrían implementar clases para los distintos estados de madurez o salud de las frutas, información muy importante para los productores de cara a conocer el estado de sus plantaciones. Este tipo de trabajos requerirían la construcción de *datasets* mucho más amplios, que cubran suficientemente las distintas clases, ya que por ejemplo en los casos de variedades de la misma fruta, y estados de maduración o salud, las diferencias entre clases son mucho más sutiles y se requerirían unos modelos más complejos y potentes para llevar estas tareas a cabo.

También se podría implementar este tipo de modelos en sistemas de recolección o poda automática, identificando las frutas y ubicándolas en el espacio mediante la combinación con otras técnicas para recogerlas o esquivarlas respectivamente.

En el pasado se han realizado investigaciones y se ha tratado de desarrollar tecnologías similares a las comentadas, con mayor o menor éxito, aunque implementadas con modelos que se podrían considerar obsoletos por los estándares actuales. Algunos de estos trabajos son: [30], [31], [32], [33], [34], [35].

7. REFERENCIAS

[1] F. Rosenblatt, «The perceptron - A perceiving and recognizing automaton», Ithaca, New York, ene. 1957.

[2] B. Widrow, «An Adaptive `Adaline' Neuron Using Chemical `Memistors'», oct. 1960.

[3] R. Fernandez-Beltran, P. Carmona, y F. Pla, «Single-frame super-resolution in remote sensing: a practical overview», *Int J Remote Sens*, vol. 38, pp. 314-354, feb. 2017, doi: 10.1080/01431161.2016.1264027.

[4] P. Viola y M. Jones, «Rapid Object Detection using a Boosted Cascade of Simple Features», 2001.

[5] N. Dalal y B. Triggs, «Histograms of Oriented Gradients for Human Detection», 2005. [En línea]. Disponible en: http://lear.inrialpes.fr

[6] A. Krizhevsky, I. Sutskever, y G. E. Hinton, «ImageNet Classification with Deep Convolutional Neural Networks», 2012. [En línea]. Disponible en: http://code.google.com/p/cuda-convnet/

[7] J. R. R. Uijlings, K. E. A. Van De Sande, T. Gevers, y A. W. M. Smeulders, «Selective search for object recognition», *Int J Comput Vis*, vol. 104, n.º 2, pp. 154-171, sep. 2013, doi: 10.1007/s11263-013-0620-5.

[8] R. Girshick, J. Donahue, T. Darrell, y J. Malik, «Rich feature hierarchies for accurate object detection and semantic segmentation», nov. 2013, [En línea]. Disponible en: http://arxiv.org/abs/1311.2524

[9] «The PASCAL Visual Object Classes Challenge 2012 (VOC2012)». Accedido: 1 de febrero de 2024. [En línea]. Disponible en: http://host.robots.ox.ac.uk/pascal/VOC/voc2012/

[10] R. Girshick, «Fast R-CNN», abr. 2015, [En línea]. Disponible en: http://arxiv.org/abs/1504.08083

[11] «The PASCAL Visual Object Classes Challenge 2007 (VOC2007)». Accedido: 1 de febrero de 2024. [En línea]. Disponible en: http://host.robots.ox.ac.uk/pascal/VOC/voc2007/

[12] S. Ren, K. He, R. Girshick, y J. Sun, «Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks», jun. 2015, [En línea]. Disponible en: http://arxiv.org/abs/1506.01497

[13] W. Liu *et al.*, «SSD: Single Shot MultiBox Detector», dic. 2015, doi: 10.1007/978-3-319-46448-0_2.

[14] J. Redmon, S. Divvala, R. Girshick, y A. Farhadi, «You Only Look Once: Unified, Real-Time Object Detection», jun. 2015, [En línea]. Disponible en: http://arxiv.org/abs/1506.02640

[15] C. Szegedy *et al.*, «Going Deeper with Convolutions», sep. 2014, [En línea]. Disponible en: http://arxiv.org/abs/1409.4842

[16] J. Terven y D. Cordova-Esparza, «A Comprehensive Review of YOLO Architectures in Computer Vision: From YOLOv1 to YOLOv8 and YOLO-NAS», abr. 2023, doi: 10.3390/make5040083.

[17] P. Jiang, D. Ergu, F. Liu, Y. Cai, y B. Ma, «A Review of Yolo Algorithm Developments», en *Procedia Computer Science*, Elsevier B.V., 2021, pp. 1066-1073. doi: 10.1016/j.procs.2022.01.135.

[18] J. Redmon y A. Farhadi, «YOLO9000: Better, Faster, Stronger», dic. 2016, [En línea]. Disponible en: http://arxiv.org/abs/1612.08242

[19] J. Redmon y A. Farhadi, «YOLOv3: An Incremental Improvement», 2018. [En línea]. Disponible en: https://pjreddie.com/yolo/.

[20] A. Bochkovskiy, C.-Y. Wang, y H.-Y. M. Liao, «YOLOv4: Optimal Speed and Accuracy of Object Detection», abr. 2020, [En línea]. Disponible en: http://arxiv.org/abs/2004.10934

[21] G. Jocher y Ultralytics, «YOLOv5: A state-of-the-art real-time object detection system». 2020. Accedido: 26 de enero de 2024. [En línea]. Disponible en: https://github.com/ultralytics/yolov5

[22] C. Li *et al.*, «YOLOv6: A Single-Stage Object Detection Framework for Industrial Applications», sep. 2022, [En línea]. Disponible en: http://arxiv.org/abs/2209.02976

[23] C.-Y. Wang, A. Bochkovskiy, y H.-Y. M. Liao, «YOLOv7: Trainable bag-of-freebies sets new state-of-the-art for real-time object detectors», jul. 2022, [En línea]. Disponible en: http://arxiv.org/abs/2207.02696

[24] G. Jocher, A. Chaurasia, y J. Qiu, «Ultralytics YOLO». enero de 2023. [En línea]. Disponible en: https://github.com/ultralytics/ultralytics

[25] K. He, X. Zhang, S. Ren, y J. Sun, «Deep Residual Learning for Image Recognition», dic. 2015, [En línea]. Disponible en: http://arxiv.org/abs/1512.03385

[26] Y. Zhang *et al.*, «ByteTrack: Multi-Object Tracking by Associating Every Detection Box», oct. 2021, [En línea]. Disponible en: http://arxiv.org/abs/2110.06864

[27] N. Aharon, R. Orfaig, y B.-Z. Bobrovsky, «BoT-SORT: Robust Associations Multi-Pedestrian Tracking», jun. 2022, [En línea]. Disponible en: http://arxiv.org/abs/2206.14651

[28] «Roboflow». Accedido: 29 de enero de 2024. [En línea]. Disponible en: https://app.roboflow.com/

[29] G. Jocher, «GitHub thread on DFL_loss». Accedido: 29 de enero de 2024. [En línea]. Disponible en: https://github.com/ultralytics/ultralytics/issues/4219

[30] S. Lu, W. Chen, X. Zhang, y M. Karkee, «Canopy-attention-YOLOv4-based immature/mature apple fruit detection on dense-foliage tree architectures for early crop load estimation», *Comput Electron Agric*, vol. 193, p. 106696, feb. 2022, doi: 10.1016/J.COMPAG.2022.106696.

[31] P. Chu, Z. Li, K. Lammers, R. Lu, y X. Liu, «Deep learning-based apple detection using a suppression mask R-CNN», *Pattern Recognit Lett*, vol. 147, pp. 206-211, jul. 2021, doi: 10.1016/J.PATREC.2021.04.022.

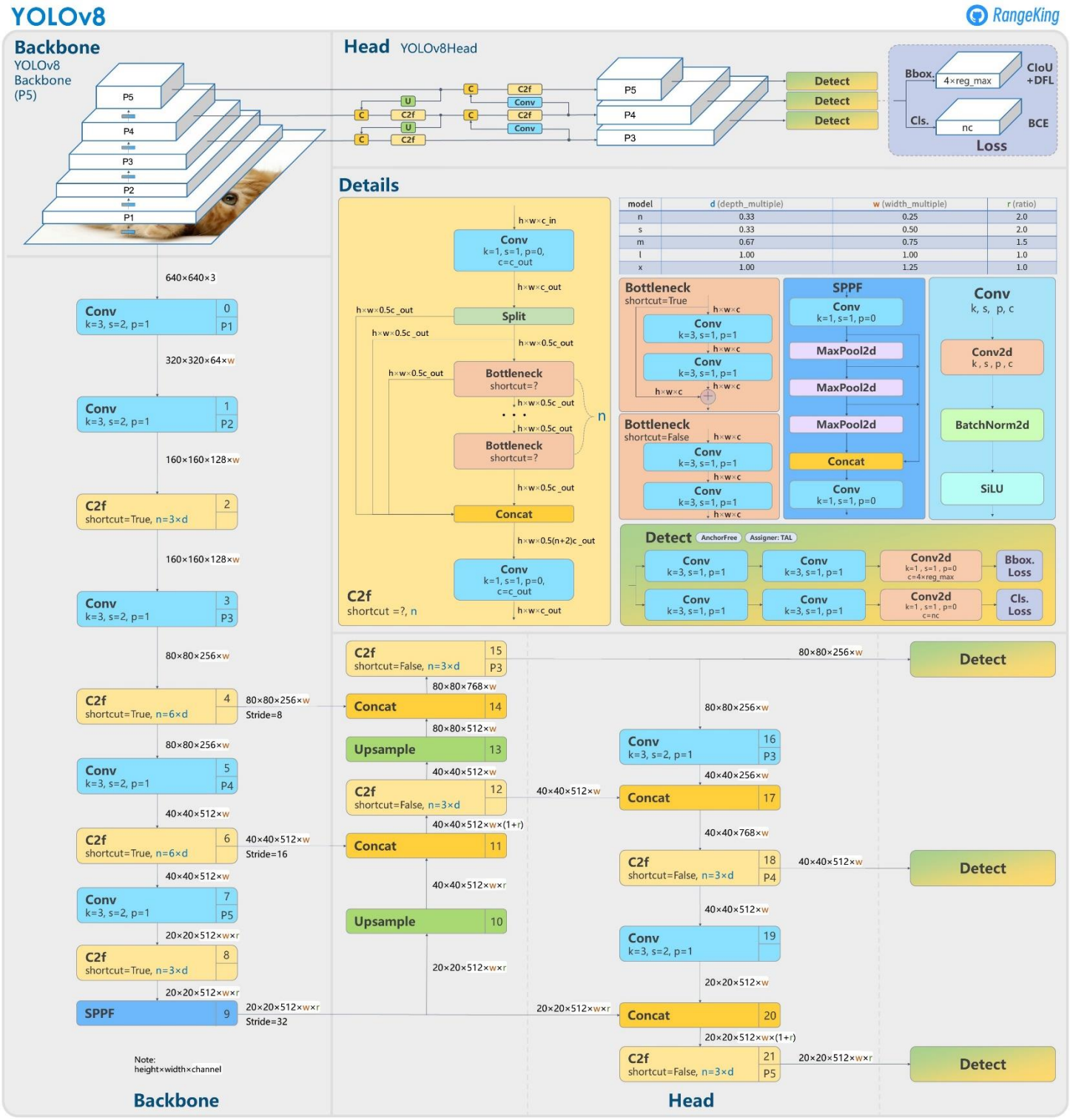
[32] P. Chu, Z. Li, K. Zhang, D. Chen, K. Lammers, y R. Lu, «O2RNet: Occluder-occludee relational network for robust apple detection in clustered orchard environments», *Smart Agricultural Technology*, vol. 5, p. 100284, oct. 2023, doi: 10.1016/J.ATECH.2023.100284.

[33] G. Zhao *et al.*, «Phenotyping of individual apple tree in modern orchard with novel smartphone-based heterogeneous binocular vision and YOLOv5s», *Comput Electron Agric*, vol. 209, p. 107814, jun. 2023, doi: 10.1016/J.COMPAG.2023.107814.

[34] X. Zhao, Y. Peng, Y. Li, Y. Wang, Y. Li, y Y. Chen, «Intelligent micro flight sensing system for detecting the internal and external quality of apples on the tree», *Comput Electron Agric*, vol. 204, p. 107571, ene. 2023, doi: 10.1016/J.COMPAG.2022.107571.

[35] K. Zhang, K. Lammers, P. Chu, Z. Li, y R. Lu, «An automated apple harvesting robot—From system design to field evaluation», *J Field Robot*, 2023, doi: 10.1002/ROB.22268.

ANEXO I. ESTRUCTURA DE YOLOV8



Fuente: https://github.com/RangeKing