

Máster universitario en robótica y automatización
2023-2024

Trabajo Fin de Máster

Detección y clasificación de vehículos a
través de redes neuronales convolucionales

María Carmona Pastor

Tutor

José María Armingol Moreno
Leganés, septiembre de 2024

DETECCIÓN DEL PLAGIO

La Universidad utiliza el programa **Turnitin Feedback Studio** para comparar la originalidad del trabajo entregado por cada estudiante con millones de recursos electrónicos y detecta aquellas partes del texto copiadas y pegadas. Copiar o plagiar en un TFM es considerado una **Falta Grave**, y puede conllevar la expulsión definitiva de la Universidad.



Esta obra se encuentra sujeta a la licencia Creative Commons **Reconocimiento - No Comercial - Sin Obra Derivada**

RESUMEN

Se han aplicado redes neuronales convolucionales, y en concreto el modelo YOLOv8, a la resolución de un problema de detección y clasificación.

A partir de un *dataset* dado, se ha entrenado un modelo capaz de distinguir entre los diferentes tipos de vehículos. Si bien este problema ya se ha abordado antes (por ejemplo, en [1]), este trabajo se centra en la clasificación correcta de los vehículos tipo furgoneta en dos categorías ligera y pesada. No se ha encontrado literatura de esta casuística concreta.

El modelo obtenido se utilizará en un sistema de peaje inteligente, que entra dentro de los sistemas inteligentes de transporte.

Palabras clave

Redes neuronales convolucionales, aumento de datos, detección de vehículos, clasificación de vehículos, PSO, validación cruzada, Sistemas Inteligentes de Transporte, peajes inteligentes.

DEDICATORIA

A todas aquellas personas que me han apoyado y me han animado a seguir adelante.

ÍNDICE GENERAL

1. INTRODUCCIÓN	1
1.1. Sistemas inteligentes de transporte	1
1.2. Objetivos	3
2. ESTADO DEL ARTE	4
2.1. Sistemas de cobro electrónico de peajes	4
2.1.1. Sistemas basados en barreras	4
2.1.2. Sistemas de flujo libre (<i>Free-Flow</i>)	4
2.1.3. Sistemas híbridos.	5
2.1.4. Peajes urbanos y peajes por congestión	6
2.1.5. Tendencias futuras	6
2.2. Sistemas utilizados para el reconocimiento de vehículos	6
2.3. YOLOv8	7
3. MARCO TEÓRICO	8
3.1. Aprendizaje supervisado.	8
3.2. Redes neuronales convolucionales	8
3.3. Preprocesamiento de datos	9
3.3.1. Requisitos para la creación de un buen <i>dataset</i>	10
3.4. Hiperparámetros principales	10
3.5. Validación y métricas	11
4. METODOLOGÍA	12
4.1. Herramientas utilizadas	12
4.2. Creación y etiquetado del conjunto de datos.	12
4.2.1. Definición de clases	12
4.2.2. Proceso de etiquetado	15
4.3. Análisis de limitaciones	17
4.3.1. Problemas detectados	17
4.3.2. Obtención de información del <i>dataset</i>	18
4.3.3. Aumento de datos	22

4.4. Ajuste de hiperparámetros	23
4.4.1. Algoritmo de optimización por enjambre de partículas (PSO) [22]	24
5. RESULTADOS Y ANÁLISIS	28
6. CONCLUSIONES	32
6.1. Trabajo futuro	32
BIBLIOGRAFÍA	33

ÍNDICE DE FIGURAS

1.1	Costes anuales externos del transporte en España 2018[2].	2
2.1	Sistema <i>free-flow</i> que utiliza la tecnología RFID en Nueva York [9].	5
3.1	Estructura de una red neuronal convolucional [14].	9
4.1	Tipos de furgoneta por tamaño [18].	13
4.2	Fiat Fiorino Cargo [19].	14
4.3	Ford Ranger [20].	14
4.4	Citröen C15 [21].	14
4.5	Herramienta de etiquetado personalizada.	16
4.6	Redundancia de imágenes presente en el <i>dataset</i>	17
4.7	Funcionalidad añadida para apuntar más datos de la imagen.	19
4.8	Distribución de distancias en el dataset.	19
4.9	Distribución del sentido de la marcha de los vehículos.	20
4.10	Distribución de colores en el dataset completo.	20
4.11	Distribución de colores en el dataset reducido.	21
4.12	Distribución de clases de los vehículos.	21
4.13	Distribución de clases en el dataset aumentado.	24
4.14	Resultados del entrenamiento con el algoritmo de optimización Adam y valores de hiperparámetros por defecto. Tiempo empleado: 244.6s.	26
5.1	Métricas obtenidas en el entrenamiento final.	28
5.2	Matriz de confusión normalizada.	29
5.3	Curva F1.	29
5.4	Curva de precisión.	30
5.5	Curva de recall.	30
6.1	Resultados del entrenamiento con el algoritmo de optimización Adam. Tiempo empleado 244.6s.	

6.2	Resultados del entrenamiento con el algoritmo de optimización Adamax. Tiempo empleado 346.1s.
6.3	Resultados del entrenamiento con el algoritmo de optimización AdamW. Tiempo empleado 274.2s.
6.4	Resultados del entrenamiento con el algoritmo de optimización NAdam. Tiempo empleado 318.6s.
6.5	Resultados del entrenamiento con el algoritmo de optimización RAdam. Tiempo empleado 343.9s.
6.6	Resultados del entrenamiento con el algoritmo de optimización RSM- Prop. Tiempo empleado 316.3s.
6.7	Resultados del entrenamiento con el algoritmo de optimización SGD. Tiempo empleado 230.1s.

ÍNDICE DE TABLAS

1.1	Evolución de los sistemas inteligentes de transporte [5].	3
4.1	Distribución de imágenes del <i>dataset</i> de partida.	15
4.2	Atajos de teclado para la herramienta de etiquetado.	16

1. INTRODUCCIÓN

Cada día vemos más y más aplicaciones de la IA (Inteligencia Artificial) a todo tipo de problemas. Podría decirse que uno de los más señalados es el procesamiento de vídeo en tiempo real, y el uso de redes neuronales para la detección de objetos. Esto es útil en muchos contextos; específicamente, este trabajo se centra en su uso en gestión del tráfico.

Se entrenará una red neuronal que clasifique los vehículos presentes en una imagen según su tipo (moto, coche, furgoneta...). Esta información se utilizará en el desarrollo de un sistema de peaje inteligente.

Los peajes inteligentes evitan que los conductores tengan que parar para realizar el pago del peaje, aumentando la fluidez del tráfico. Estos sistemas funcionan utilizando cámaras y sensores cuya información se procesa mediante IA para obtener los datos de los vehículos que pasan por la carretera, como la matrícula y el tipo de vehículo. En lugar de los puntos de pago tradicionales, los vehículos contarán con un dispositivo electrónico que se detecta al pasar por cierto tramo de la carretera, y a través del cual se puede realizar el pago automáticamente.

Esto tiene varias ventajas evidentes:

- Comodidad, ya que elimina la necesidad de los conductores de reducir la velocidad, ahorrando tiempo y esfuerzo.
- Sostenibilidad, puesto que al evitar congestiones se reducen las emisiones.
- Seguridad, puesto que se eliminarían zonas en las que confluyen muchos vehículos.
- Rentabilidad, dado que los costes que generan los tres últimos puntos (figura 1.1) se reducirían considerablemente.

A la hora de fijar el precio a pagar en un peaje, influyen varios factores, como la distancia recorrida y el tipo o peso del vehículo. Este trabajo se centrará en la clasificación de furgonetas como “ligeras” o “pesadas”, lo cual tiene una aplicación directa a este problema.

Los peajes inteligentes entran dentro de los llamados sistemas inteligentes de transporte (SIT). A continuación, se ofrece un contexto histórico de cómo han ido evolucionando estos sistemas a lo largo de los años.

1.1. Sistemas inteligentes de transporte

Se trata de infraestructuras que utilizan las TIC para establecer comunicaciones entre todos los agentes que participan en la red de transportes.

Costes externos del transporte en España
(miles de millones de euros anuales)

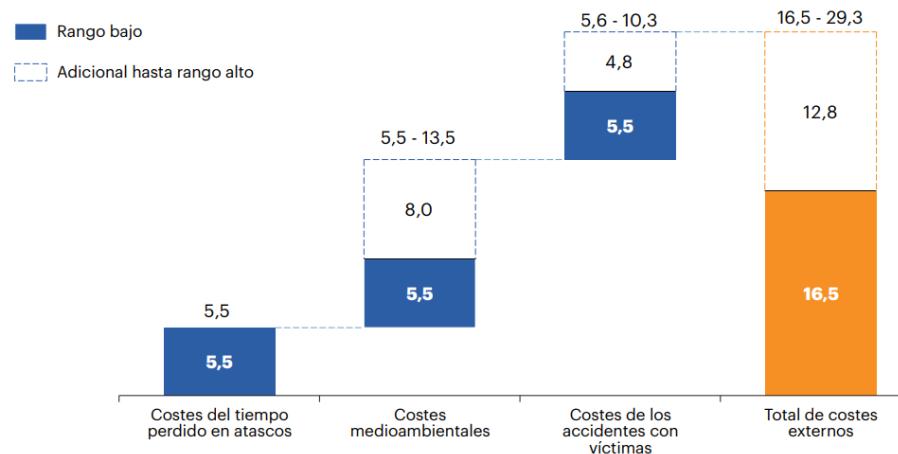


Fig. 1.1. Costes anuales externos del transporte en España 2018[2].

Se basan en la comunicación de cámaras y otros sensores, redes de comunicación y sistemas de procesamiento de datos en tiempo real. Los datos obtenidos se integran y transmiten a los agentes interesados, ayudando a la toma de decisiones informadas o automáticas, según sea el caso, con el objetivo de mejorar el flujo del tráfico.

Existen muchos tipos de sistemas inteligentes de transporte, algunos ejemplos actuales son:

- Sistemas de gestión del tráfico, como semáforos inteligentes[3] o señales que informan del estado del tráfico en tiempo real.
- Sistemas de cobro electrónico de peajes o telepeaje (ETC¹). Profundizaremos sobre estos en el apartado 2.1.
- Sistemas de gestión del transporte público.
- Sistemas de gestión de flotas de vehículos.
- Sistemas cooperativos de transporte (C-ITS), que implementan comunicación V2V (vehículo a vehículo), V2I (vehículo a infraestructura), o incluso V2X (vehículo a todo) [4].

Los SIT han evolucionado significativamente en los últimos años, tal como ilustra la tabla 1.1. Desde el año 2000 han mejorado exponencialmente las infraestructuras y las tecnologías, permitiendo una gestión de los datos eficiente, en tiempo real. Esto no era posible antes, debido a que el volumen de datos a tratar suele ser muy elevado en estos sistemas, puesto que lo más común es recurrir al uso de cámaras, a parte de otros sensores.

Con la introducción del 5G, se están dando grandes avances en este campo [5].

¹Viene del inglés *Electronic Toll Collection*

Generación	Periodo	Características
1 ^a generación	2000	Comunicaciones unidireccionales.
2 ^a generación	2000-2003	Comunicaciones bidireccionales.
3 ^a generación	2004-2005	Introducción de vehículos autónomos; sistemas automáticos e interactivos; gestión de SIT.
4 ^a generación	2006-2019	Incorporación multimodal de dispositivos móviles personales y vehiculares; redes de infraestructura e información usadas en SIT; soluciones de movilidad contextual personal.
5 ^a generación	2020-	Sistemas cooperativos de transporte inteligente (C-ITS); mejora en la comunicación vehículo-infraestructura (V2I); seguridad peatonal; gamificación.

TABLA 1.1. EVOLUCIÓN DE LOS SISTEMAS INTELIGENTES DE TRANSPORTE [5].

1.2. Objetivos

El objetivo de este trabajo es obtener un modelo de red neuronal que clasifique los vehículos presentes en una imagen en las siguientes clases: moto (M), coche (C), furgoneta ligera (FL), furgoneta pesada (FP), autobús, (A), camión ligero (CL), camión pesado (CP) y camión pesado articulado (CPA).

Concretamente, el objetivo principal es que distinga entre furgoneta ligera y furgoneta pesada.

Para conseguirlo, nombramos los siguientes subobjetivos:

1. Creación de un *dataset* completo y bien balanceado mediante el uso de aumento de datos.
2. Ajuste de hiperparámetros y entrenamiento de la red.

2. ESTADO DEL ARTE

2.1. Sistemas de cobro electrónico de peajes

Como se ha comentado anteriormente, los ETC son sistemas automatizados que evitan que el conductor tenga que detenerse a pagar. Esta idea existe desde 1959, cuando William Vickrey propuso que un sistema según el cual cada vehículo llevaría un transpondedor cuya señal se leería cuando pasase por una intersección, pero no llegó a implantarse. No fué hasta 1975 cuando se implementó un ETC por primera vez en Singapur [6], y hoy en día es una industria que mueve más de \$8.5B [7].

El cobro electrónico de peajes (ETC) ha evolucionado significativamente en las últimas décadas, permitiendo una mayor eficiencia en la gestión del tráfico y la recaudación de fondos para el mantenimiento vial. Estas tecnologías han pasado de sistemas simples a infraestructuras avanzadas que ofrecen métodos de pago sin contacto y sin necesidad de detenerse. Actualmente, los sistemas ETC se dividen en varias categorías según el tipo de tecnología utilizada [8][7].

2.1.1. Sistemas basados en barreras

Estos sistemas requieren que los vehículos se detengan brevemente o reduzcan la velocidad al pasar por una cabina de peaje. La tecnología utilizada puede variar:

- **Identificación por Radiofrecuencia (RFID):** es una de las tecnologías más comunes. Los vehículos están equipados con una etiqueta RFID, que es leída por una antena cuando el vehículo se acerca a la barrera. La barrera se abre automáticamente y el cobro se realiza en una cuenta vinculada al usuario. Este sistema se utiliza ampliamente en países como India y Brasil. Esta tecnología también se puede usar en sistemas *free-flow*, como el de la figura 2.1.
- **Tarjetas de pago:** en algunos sistemas más simples, los conductores usan tarjetas para completar el pago.

Aunque eficaces, estos sistemas generan cierto nivel de congestión debido a la reducción de la velocidad o paradas en las barreras.

2.1.2. Sistemas de flujo libre (*Free-Flow*)

En los sistemas de peaje de flujo libre, los vehículos no necesitan detenerse ni reducir la velocidad, lo que mejora significativamente la eficiencia del tráfico. Estos sistemas emplean tecnologías avanzadas:



Fig. 2.1. Sistema *free-flow* que utiliza la tecnología RFID en Nueva York [9].

- **Sistema Global de Navegación por Satélite (GNSS):** utilizado en algunos países europeos, como Alemania, este sistema rastrea la ubicación de los vehículos mediante GPS y cobra en función de la distancia recorrida en áreas sujetas a peaje. Es particularmente efectivo para camiones y vehículos comerciales.
- **Comunicación Dedicada de Corto Alcance (DSRC):** similar a RFID, pero con mayor capacidad de intercambio de datos. DSRC se utiliza principalmente en Japón y algunas partes de EE. UU., donde permite la comunicación bidireccional entre el vehículo y la infraestructura vial. Esto facilita no solo el cobro, sino también la transmisión de datos sobre el tráfico y la seguridad.
- **Cámaras de Video (ANPR - Reconocimiento Automático de Matrículas):** este sistema usa cámaras para capturar imágenes de las matrículas y hacer coincidir la información con una base de datos de vehículos registrados. El uso de ANPR es común en Europa y Asia, y es ideal para zonas urbanas donde se busca minimizar la infraestructura física.
- **Móviles inteligentes:** estos sistemas todavía están en desarrollo, pero muestran un gran potencial. Aprovechan los avances con NFC y el aumento general de la cobertura. Sin embargo, todavía se presentan desafíos como la clasificación de vehículos y la protección de datos.

2.1.3. Sistemas híbridos

Algunos países han implementado sistemas híbridos que combinan tecnologías para maximizar la flexibilidad y adaptarse a diferentes tipos de vehículos y situaciones. Por ejemplo, en algunos peajes, los vehículos pueden optar por usar etiquetas RFID, GNSS, o simplemente pasar bajo cámaras de ANPR sin necesidad de equipamiento especial.

2.1.4. Peajes urbanos y peajes por congestión

Además de los peajes en autopistas, las tecnologías ETC también están siendo utilizadas en áreas urbanas para reducir la congestión y la contaminación. Londres, Singapur y Estocolmo son algunos de los ejemplos más avanzados, donde los conductores deben pagar una tarifa por entrar en zonas específicas de la ciudad. Estos sistemas se basan en cámaras de ANPR o etiquetas RFID y han demostrado ser eficaces para controlar el tráfico y mejorar la calidad del aire en zonas urbanas.

2.1.5. Tendencias futuras

El futuro de los sistemas de peaje electrónico está marcado por el uso de tecnologías emergentes como 5G y vehículos autónomos.

Además, se espera que los sistemas de peaje sean más personalizados, permitiendo el cobro en función del uso, el tiempo, las condiciones del tráfico y las emisiones del vehículo, pues existe el problema de cuánto cobrar a cada vehículo, o en base a qué. En [2] se menciona que los vehículos pesados podrían pagar hasta cinco veces más que los ligeros. También existen propuestas de esquemas de fijación de precios dinámicos como [10], que utilizan como input la situación del tráfico y otros factores, además del tipo de vehículo.

2.2. Sistemas utilizados para el reconocimiento de vehículos

La detección de vehículos no se limita a imágenes o videos; existen métodos alternativos que también pueden ser efectivos. Entre estos métodos se encuentran [11]:

- Sensores magnéticos: se instalan en el suelo y detectan cambios en el campo magnético causados por el paso de vehículos.
- Radares: utilizan ondas electromagnéticas para medir la velocidad y la presencia de vehículos.
- LiDAR: generan nubes de puntos que crean un modelo tridimensional del entorno.
- Sensores infrarrojos: captan el calor emitido por los vehículos.
- Comunicación WiFi: se puede detectar la presencia de vehículos a través de esta comunicación inalámbrica.

Sin embargo, en cuanto a coste y rendimiento, el tratamiento de imágenes y videos sigue siendo el método más sencillo y utilizado para la detección de vehículos, puesto que la tecnología de procesamiento de imágenes ofrece una solución económica y versátil.

Después de la obtención de la información mediante sensores, hay que tratarla para conseguir información estructurada y útil. Dentro del campo del aprendizaje profundo, las redes neuronales convolucionales (CNN) se utilizan ampliamente para clasificar vehículos a partir de imágenes. Estas redes pueden estar basadas en diferentes enfoques de detección. Los algoritmos de detección en dos fases, como los que generan primero la *bounding box* y luego hacen la predicción, incluyen métodos como R-CNN y sus variantes (Fast R-CNN y Faster R-CNN). Estos enfoques, aunque precisos, tienden a ser más lentos debido a la necesidad de procesar cada imagen en múltiples etapas.

En contraste, los algoritmos de detección en una sola fase, como YOLO (You Only Look Once) y RetinaNet, realizan la detección y la clasificación en un solo paso, lo que permite una mayor rapidez en el procesamiento.

Además, también están las redes neuronales convolucionales basadas en regiones (RCNN), que siguen siendo una opción viable para escenarios donde la precisión detallada es más importante que la velocidad.

Cada uno de estos enfoques tiene sus ventajas y limitaciones, y la elección del algoritmo adecuado depende de los requisitos específicos de la aplicación y el entorno en el que se implementará [12].

2.3. YOLOv8

En este trabajo, utilizaremos la arquitectura YOLOv8, de la serie You Only Look Once. Esta arquitectura se utiliza para detectar objetos en imágenes en tiempo real, y destaca por su velocidad. Es capaz de realizar tareas de detección, clasificación, segmentación y detección de poses [13]. Su nombre se debe a que utiliza una única red para inferir sus predicciones.

La estructura de YOLOv8 se puede dividir en tres partes: el *backbone*, el cuello, y la cabeza. La función del *backbone* es la extracción de características. La del cuello, la fusión y refinamiento de estas características, y la de la cabeza, la generación de predicciones. El *backbone* utiliza una serie de capas convolucionales para extraer características jerárquicas de las imágenes, mientras que el cuello combina estas características a diferentes escalas para mejorar la detección de objetos en distintos tamaños. Finalmente, la cabeza de YOLOv8 predice simultáneamente las *bounding boxes*, las clases y las confianzas de los objetos en una sola pasada, lo que permite una detección rápida y precisa.

Por su velocidad, es una solución ideal para sistemas de vigilancia, vehículos autónomos y otras aplicaciones que requieren procesamiento en tiempo real. En comparación con versiones anteriores, la precisión promedio (mAP) mejora significativamente, y puede funcionar en dispositivos con recursos limitados, por lo que se puede implementar en una variedad de plataformas. Además, se destaca por su adaptabilidad, con modelos ligeros o estándar según las necesidades del usuario. En este trabajo utilizaremos la versión ligera YOLOv8n.

3. MARCO TEÓRICO

3.1. Aprendizaje supervisado

Existen numerosas técnicas de aprendizaje profundo o deep learning. Estas se pueden dividir en aprendizaje supervisado o no supervisado. La característica definitoria del aprendizaje supervisado es que recibe como entrada dataset que llevan asociados una salida esperada. El modelo debe aprender los patrones necesarios para que sus predicciones se ajusten a dicha salida esperada. Por el contrario, en el aprendizaje no supervisado, no se dispone de una salida esperada, si no que el modelo aprende patrones y características presentes en los datos, pero estos patrones no están “predefinidos”, los datos no están etiquetados [14].

Muchas veces, las etiquetas asociadas a los datos en el aprendizaje supervisado se refieren a ciertas clases en las que se quieren clasificar los datos de entrada. Esto se denominan problemas de clasificación, como es el caso del nuestro.

Utilizaremos una red neuronal convolucional (CNN, por sus siglas en inglés), que entra dentro de la categoría del aprendizaje supervisado.

3.2. Redes neuronales convolucionales

Este tipo de modelos tiene un rendimiento superior a otros a la hora de procesar imágenes o entradas de audio [15], motivo por el cual lo hemos escogido para esta tarea.

Las redes neuronales se componen de capas de nodos que están conectados unos a otros. La red recibe como entrada los valores numéricos asociados a cada píxel en una imagen. Las CNNs tienen al menos una capa convolucional y una capa totalmente conectada como capa final, pudiendo tener más capas convolucionales intermedias, así como capas de agrupación [15]. Podemos verlo gráficamente en la figura 3.1.

- **Capa convolucional:** se trata de un filtro o kernel que recorre partes de la imagen, y determina si una característica (línea, curva, textura...) existe o no en ese subconjunto de píxeles. El filtro es una matriz, y dependiendo de los valores o pesos que tenga, detectará una característica u otra. El resultado de pasar la imagen por este filtro es una matriz que representa ciertas características de la imagen. En la primera capa se detectarán elementos pequeños y, con cada capa adicional, la red será capaz de detectar características mayores o más abstractas.
- **Capa de agrupación:** su función es reducir la dimensionalidad del problema. No siempre están presentes. También utiliza un kernel, como la capa convolucional,

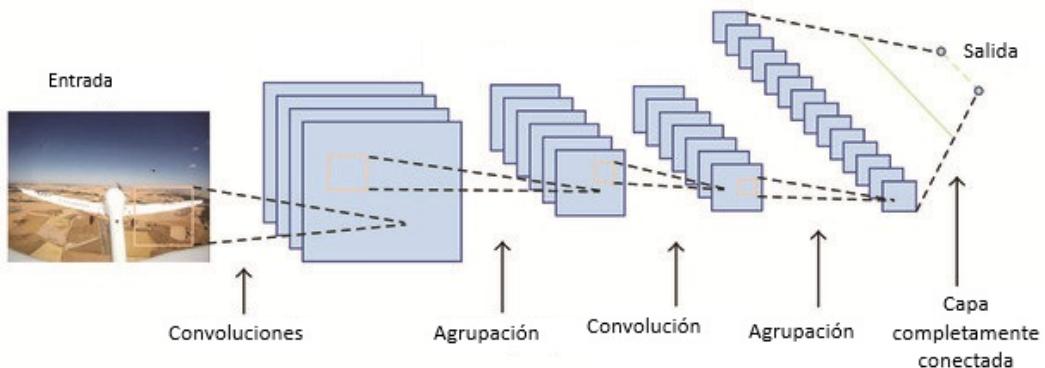


Fig. 3.1. Estructura de una red neuronal convolucional [14].

pero en este caso no tiene pesos asociados, si no que representa una función de agregación; por ejemplo, seleccionar el píxel con el valor más alto.

- **Capa totalmente conectada:** la última capa siempre estará totalmente conectada; es decir, cada nodo de esta capa está directamente conectado a un nodo de la capa anterior. De esta forma la salida de la red tendrá en cuenta todas las características extraídas en las capas anteriores.

3.3. Preprocesamiento de datos

El preprocesamiento de datos se refiere a la fase previa al entrenamiento de la red. En esta fase se decide qué datos de entrada se utilizarán en el entrenamiento. También es recomendable realizar un análisis de los datos. Los datos incluyen las imágenes y sus etiquetas. Dentro del preprocesamiento de datos entran el perfilado, limpieza, reducción, transformación, enriquecimiento y validación de los datos [16]:

- El **perfilado** de datos se refiere a la obtención de estadísticas que ayudan a comprender la estructura y la calidad de los datos.
- La **limpieza** de datos consiste en corregir o eliminar datos incorrectos, inexactos, duplicados o corruptos que podrían introducir ruido en el modelo.
- La **reducción** de datos consiste en la eliminación controlada de datos con el objetivo de reducir el volumen, pero manteniendo resultados suficientemente parecidos.
- La **transformación** de datos consiste en modificar o extraer nuevos datos que pueden ser beneficiosos según el problema.
- En el **enriquecimiento** se añaden datos provenientes de otras fuentes, o bien derivados de los datos originales. El aumento de datos entraría en esta fase.

- La **validación** comprueba que los datos cumplen ciertos criterios específicos, garantizando que sean completos, precisos y confiables.

El preprocesamiento es una fase crucial en el desarrollo de modelos de aprendizaje automático, y tiene un impacto muy alto en la calidad y precisión de los resultados finales.

3.3.1. Requisitos para la creación de un buen *dataset*

A continuación, se describen las características básicas que debe tener un dataset para obtener buenos resultados en el entrenamiento:

- **Diversidad y representatividad:** deben existir datos que representen las máximas situaciones posibles que podrían darse. Por ejemplo, si es posible que el modelo reciba imágenes tomadas de día y de noche, debe haber suficientes ejemplos de los dos casos en el dataset.
- **Tamaño adecuado:** debe haber suficientes datos. Un número aceptable suele ser entre 1000 y 10.000 imágenes.
- **Equilibrio de clases:** cada clase debe tener un número similar de ejemplos.
- **Etiquetado preciso:** las etiquetas asociadas a cada imagen deben ser precisas y consistentes.
- **Buena calidad de imagen:** la resolución de la imagen debe ser aceptablemente alta.

3.4. Hiperparámetros principales

Existe una gran cantidad de hiperparámetros que podemos modificar antes del entrenamiento de nuestra red, los que se explican a continuación son los que suelen influir más en el resultado:

- **Número de épocas:** cada época representa una pasada completa por todas las imágenes del *dataset*. Un número bajo de épocas puede llevar a un modelo subajustado, que se refiere a que no ha tenido tiempo de aprender ningún patrón, y por lo tanto no hará predicciones correctas ni siquiera en las imágenes del conjunto de entrenamiento. Por otro lado, es posible que demasiadas épocas conduzcan al fenómeno contrario, sobreajuste. Esto ocurre cuando el modelo empieza a aprender patrones demasiado específicos, propios de las imágenes presentes en el conjunto de entrenamiento/validación, y cuando se le presenta con imágenes de fuera de ese conjunto, falla.

- **Tamaño de *batch*:** Se refiere al número de imágenes que se procesan antes de actualizar los pesos de la red. Un tamaño grande tiende a ser más estable, pero puede conducir a una mala generalización.
- **Tasa de aprendizaje:** controla el tamaño de los ajustes que se le hacen a los pesos en cada iteración del aprendizaje. Si es demasiado alta, es posible que el modelo no llegue a converger o se salte el mínimo óptimo; si es demasiado baja, el entrenamiento puede tardar mucho y quedarse estancado en mínimos locales.

3.5. Validación y métricas

Para realizar la validación del modelo necesitamos mirar los valores de algunas métricas, que nos dirán cómo de bien se comporta el modelo. Algunas de las más usadas son[17]:

- **Precisión:** Dentro de los objetos detectados, mide la cantidad que ha sido clasificada correctamente; es decir mide los resultados positivos verdaderos. Una precisión alta indica que el modelo reconoce correctamente todos los objetos detectados, pero no tiene en cuenta si se han detectado o no todos los objetos presentes en la imagen.
- **Recuperación o *recall*:** Mide cuántos objetos se han detectado en la imagen respecto a los que verdaderamente hay. Un resultado bajo indica que se deja objetos sin reconocer, y uno demasiado alto indica que es posible que se estén encontrando falsos positivos. Mientras que la precisión mide la calidad de las detecciones, el *recall* mide la cantidad.
- **FP1:** Es la media armónica entre la precisión y el *recall*, de modo que esta métrica tiene en cuenta tanto falsos positivos como falsos negativos.
- **mAP50:** También combina los valores de precisión y *recall*, pero en este caso, lo hace calculando el área bajo la curva precisión-*recall* correspondiente a cada una de las clases, y haciendo la media de los valores obtenidos. El 50 se refiere a que se considerarán detecciones sólo a las *bounding boxes* que se solapen con la *bounding box* verdadera en más del 50 %; es decir, se establece el umbral IoU en 0.5.
- **mAP50-95 :** parecida a mAP50, pero obtiene los valores para umbrales IoU de 50 a 95, en intervalos de 5 en 5, obteniendo más información sobre la precisión de las medidas.

4. METODOLOGÍA

4.1. Herramientas utilizadas

Label Studio: programa destinado al etiquetado de datos, se ha utilizado para crear las etiquetas de las imágenes que no las tenían.

Tkinter: interfaz en python para el kit de herramientas de GUI Tk, que se utiliza para crear aplicaciones que corren en Windows, Mac OS X y Linux. En este caso, se ha utilizado para crear una herramienta personalizada que permite cambiar las clases de los vehículos en pocos *clicks*, a partir de las *bounding boxes* ya existentes. También se ha añadido funcionalidad para guardar otras estadísticas de los vehículos, no sólo la clase.

Albumentations: librería de python para el aumento de datos.

Ultralytics: herramienta con la que entrenaremos sobre el modelo YOLOv8 para obtener nuestra red.

4.2. Creación y etiquetado del conjunto de datos

4.2.1. Definición de clases

Partimos con ventaja en este problema, dado que en un trabajo anterior, se realizaron experimentos para obtener un modelo capaz de clasificar vehículos en las siguientes categorías, y por lo tanto, muchas imágenes tenían ya casi todas las etiquetas correctas.

0: Moto (M)

1: Coche (C)

2: Furgoneta (F)

3: Autobús (A)

4: Camión ligero (CL)

5: Camión pesado (CP)

6: Camión pesado articulado (CPA)

Como se puede observar, cada clase debe llevar un identificador (número entero) asociado, y cada una tiene su abreviatura.

En este trabajo, añadiremos una nueva categoría a la clasificación, dividiendo la clase “Furgoneta” en “Furgoneta ligera” y “Furgoneta pesada”.

Como se puede observar, cada clase debe llevar un identificador (número entero) asociado, y cada una tiene su abreviatura.

En este trabajo, añadiremos una nueva categoría a la clasificación, dividiendo la clase “Furgoneta” en “Furgoneta ligera” y “Furgoneta pesada”.

0: Moto (M)

1: Coche (C)

2: Furgoneta ligera (FL)

3: Furgoneta pesada (FP)

4: Autobús (A)

5: Camión ligero (CL)

6: Camión pesado (CP)

7: Camión pesado articulado (CPA)

A continuación, se establecen los criterios para diferenciar entre FL y FP.



Fig. 4.1. Tipos de furgoneta por tamaño [18].

En la Figura 4.1, vemos una clasificación de furgonetas por tamaño. Decimos que una furgoneta es ligera si se encuentra por debajo del umbral de longitud del tipo L1H1, cualquier cosa por encima será pesada. Si un vehículo es más largo que L1, pero más bajo que H1, se considerará pesado. Si es más alto que H1, pero más corto que L1, se considerará ligero. Es decir, la longitud es el factor predominante, mientras que la altura se utiliza para “desempatar”.

En la práctica, es difícil medir las distancias con exactitud en una foto, de modo que, si no lo vemos claro, se pueden seguir las siguientes reglas:

- Si podemos ver el lateral del vehículo, y vemos que caben 4 ruedas o más entre sus ruedas, es pesada, si no, es ligera.
- Si no podemos ver bien el lateral, discriminamos por altura. Si es difícil determinar si es más o menos alta que el umbral establecido, tomaremos como referencia algún coche que aparezca a una distancia similar en el mismo escenario.

En las Figuras 4.2 a 4.4 se muestran ejemplos de furgonetas ligeras.



Fig. 4.2. Fiat Fiorino Cargo [19].



Fig. 4.3. Ford Ranger [20].



Fig. 4.4. Citroën C15 [21].

Una vez tenemos las clases bien definidas, para entrenar un modelo con YOLOv8, es necesario etiquetar las imágenes con un formato determinado. Cada archivo de imagen debe tener un archivo “.txt” asociado, con el mismo nombre que la imagen. Dicho archivo tendrá una línea por cada vehículo presente en la imagen, con el siguiente formato:

id_clase x_centro y_centro anchura altura

Donde id_clase se refiere al identificador de clase, (x_centro, y_centro) se refiere a las coordenadas, en píxeles, de la *bounding box* del vehículo, contadas desde la esquina superior izquierda de la imagen, y anchura y altura se refieren a las dimensiones de la *bounding box*, también en píxeles.

En el conjunto inicial de datos, tenemos algunas imágenes sin etiquetar, y otras muchas que están etiquetadas, pero estas no incluyen las clases FL y FP, si no que todas las

furgonetas están marcadas como FL. El primer paso en la creación del dataset será añadir las etiquetas que faltan, y modificar el resto.

4.2.2. Proceso de etiquetado

Partimos de tres conjuntos de datos, organizados según la tabla 4.1. Podemos ver que la mayor parte de las imágenes (conjunto furgonetas_27_06_2023) ya tenían un archivo o etiqueta, pero no con las clases que necesitamos.

Nombre de la carpeta	Preetiquetadas	Nº de imágenes
FURGONETAS LIGERAS	No	97
FURGONETAS PESADAS	No	47
furgonetas_27_06_2023	Si	2254

TABLA 4.1. DISTRIBUCIÓN DE IMÁGENES DEL DATASET DE PARTIDA.

Dado que los vehículos ya estaban correctamente etiquetados, lo único que habría que hacer sería cambiar la clase de las furgonetas pesadas, las etiquetas del resto de vehículos no hace falta tocarlas. Se consideró que sería más eficiente crear un programa personalizado para hacer exactamente eso en un sólo click, en lugar de utilizar Label Studio para esta tarea también. La interfaz de esta herramienta se muestra en la figura 4.5.

Muestra dos imágenes, una con la *bounding box* y otra sin ella, lo cual resulta útil en imágenes donde hay mucho solapamiento de vehículos, y la *bounding box* puede tapar otras, o en casos donde el vehículo es muy pequeño, y la presencia de la *bounding box* hace más difícil la tarea de clasificación. Cada vehículo lleva un identificador numérico asociado, que se corresponde con el número que aparece al lado de cada casilla, justo debajo de la imagen.

Aparecerán habilitadas las casillas correspondientes a las furgonetas. Si la marcamos, es FP, si no, es FL. Para cambiar de imagen podemos utilizar las flechas de debajo, o una serie de atajos de teclado (tabla 4.2). También podemos introducir el número de la imagen en el cuadro superior, teniendo en cuenta que las imágenes se ordenan alfabéticamente².

Cada vez que se pasa de una imagen a otra, se guarda la información de clase en el archivo “.txt” con las etiquetas de esa imagen. Para lanzar la herramienta, basta con ejecutar el script `clasify_vans.py`, disponible en el Anexo A.

Recibe cuatro directorios como parámetro:

- `image_folder`: contiene las imágenes.

²Windows y python no ordenan alfabéticamente de la misma manera cuando se trata de números. Si se van a estar mirando las imágenes en el explorador de archivos, merece la pena renombrar las imágenes al principio para que este orden coincida.

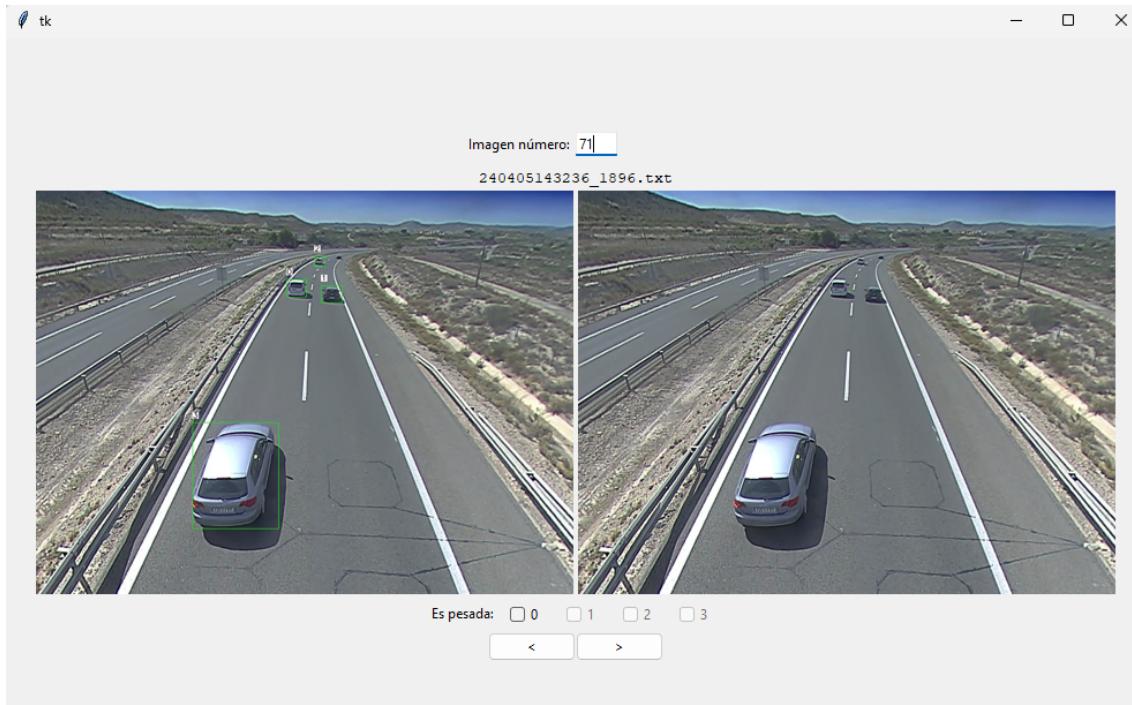


Fig. 4.5. Herramienta de etiquetado personalizada.

Atajo de teclado	Acción
Flecha dcha o click dcho	Pasar a imagen siguiente
Flecha izda o click con la rueda del ratón	Pasar a imagen anterior
Letra "L"	Poner el <i>focus</i> en la casilla siguiente
Letra "K"	Poner el <i>focus</i> en la casilla anterior
Carácter ":"	Seleccionar o deseleccionar casilla

TABLA 4.2. ATAJOS DE TECLADO PARA LA HERRAMIENTA DE ETIQUETADO.

- bb_image_folder: contiene las imágenes con bounding boxes.
- label_folder: contiene las etiquetas originales.
- new_label_folder: carpeta donde se quieren guardar las etiquetas modificadas.

En cada uno de estos directorios (excepto en el que se guardan las etiquetas modificadas), debe haber el mismo número de archivos, con los mismos nombres (excepto la extensión). Una vez se ha recorrido todo el dataset, se habrán guardado todas las nuevas etiquetas en new_label_folder, y ya tendremos nuestro dataset bien etiquetado.

Finalmente, se utilizó Label Studio para etiquetar las imágenes de las carpetas FURGONETAS LIGERAS y FURGONETAS PESADAS. Una vez terminado, obtenemos un “.json” con los datos de las imágenes, que habrá que transformar en archivos “.txt” para poderlos usar en el entrenamiento con YOLOv8. Esto se puede hacer fácilmente con el script LS2yolo.py, disponible en el Anexo B.



Fig. 4.6. Redundancia de imágenes presente en el *dataset*.

4.3. Análisis de limitaciones

4.3.1. Problemas detectados

Durante el etiquetado del dataset, se observaron las siguientes limitaciones:

- Redundancia:

Hay imágenes que corresponden al mismo vídeo de una cámara de tráfico, y muchas van fotograma a fotograma. Esto significa que hay muchas imágenes muy parecidas entre sí, como puede observarse en la figura 4.6. Este tipo de imágenes no aportarán apenas nada al modelo, si no que sólo conseguirán ralentizar el proceso de entrenamiento. Recordemos que, para obtener un buen dataset, necesitamos imágenes diversas. Desgraciadamente, una cantidad considerable de imágenes del conjunto de datos de partida es redundante.

- Ambigüedad:

A pesar de haber definido unos criterios claros, la realidad es que hay muchas imágenes en las que es casi imposible decir con certeza si una furgoneta cae en la categoría de pesadas o de ligeras. Muchas se ven de espaldas y a lo lejos, lo cual complica las cosas cuando se intenta determinar su tamaño. Otras simplemente están demasiado lejos como para distinguirlas, aunque este caso es menos preocupante. Hubo que basarse en el contexto en muchas ocasiones durante la fase de etiquetado; es decir, mirar las imágenes anteriores o siguientes en las cuales el vehículo se viese mejor, y anotar en consecuencia. Esto nos indica que es probable que al modelo le cueste alcanzar una precisión muy alta, dado que ni siquiera los humanos somos capaces de distinguir muchos casos.

- Variación de color:

La mayor parte de las furgonetas son blancas. Esto puede causar que el modelo

haga la generalización de que los vehículos blancos son furgonetas, o de que las furgonetas de otros colores no lo son.

- Variedad de localizaciones:

El dataset se compone de imágenes tomadas de cámaras de tráfico que provienen de, a lo sumo, 6 o 7 cámaras diferentes, prácticamente todas en carreteras o autopistas, muy pocas en un entorno urbano.

- Variación en el estado del tráfico:

En todo el conjunto no se encuentran imágenes con más de 10 vehículos, la mayoría tiene sólo 2 o 3, esto significa que el modelo podría no detectar todos los objetos en una carretera con atasco, ya que también son escasas las imágenes con occlusiones.

- Variación en la perspectiva con que se ven los vehículos:

Relacionado con la poca variación de localizaciones, los ángulos desde los que se ven los vehículos también son limitados.

- Variación de las condiciones de visibilidad:

Todas las imágenes son de días soleados, y hay muy pocas imágenes tomadas de noche, o con visibilidad baja o diferentes condiciones de luminosidad.

- Desequilibrio de clases:

Las imágenes proporcionadas para el entrenamiento son únicamente de furgonetas, lo cual, teniendo en cuenta que, de forma ideal, el modelo debería clasificar correctamente en las 7 clases proporcionadas, es un problema. Sin embargo, esto no es excesivamente preocupante, dado que el objetivo de esta red se centra más en distinguir entre tipos de furgonetas, y no tanto en clasificar correctamente el resto de vehículos.

4.3.2. Obtención de información del *dataset*

Con el objetivo de obtener información más precisa que nos ayude a la hora de escoger una estrategia de aumento de datos, se volvió a recorrer el *dataset*, esta vez anotando otras características de las imágenes, a parte de la clase. Esta segunda vuelta también sirve para corregir posibles errores. Para conseguir esto, se añadió funcionalidad a la herramienta de etiquetado, como podemos ver en la figura 4.7. Permite anotar el color de las furgonetas, así como la distancia a la que se encuentran, y si van de frente o de espaldas. Adicionalmente, también se anotaron las imágenes que eran redundantes, con el objetivo de obtener una versión reducida del *dataset*, que será muy útil para reducir el tiempo de entrenamiento del modelo. Eliminando estas imágenes obtenemos el *dataset* reducido con el que realizaremos el ajuste de hiperparámetros.



Fig. 4.7. Funcionalidad añadida para apuntar más datos de la imagen.

Los datos se guardan en ficheros “.txt” con el mismo nombre de la imagen, pero en un directorio diferente al de las etiquetas, que debe especificarse. Una vez recolectada la información, podemos ver los resultados obtenidos en las figuras 4.8 a 4.11.

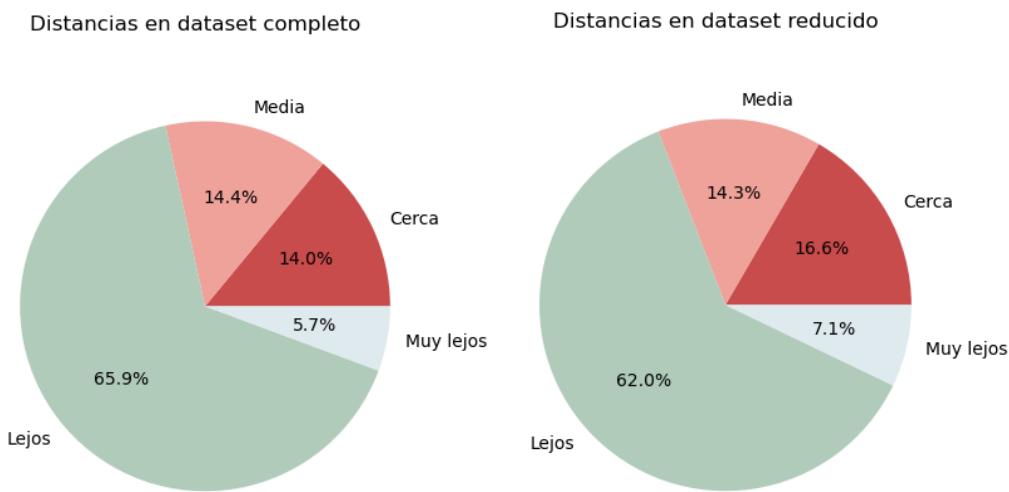


Fig. 4.8. Distribución de distancias en el dataset.

En la figura 4.8, comprobamos que la mayoría de furgonetas están “lejos”; es decir, no se aprecian demasiado bien las características del vehículo. Si bien es cierto que no hay un porcentaje muy alto de “muy lejos”, que implica que sólo se ven unos pocos píxeles.

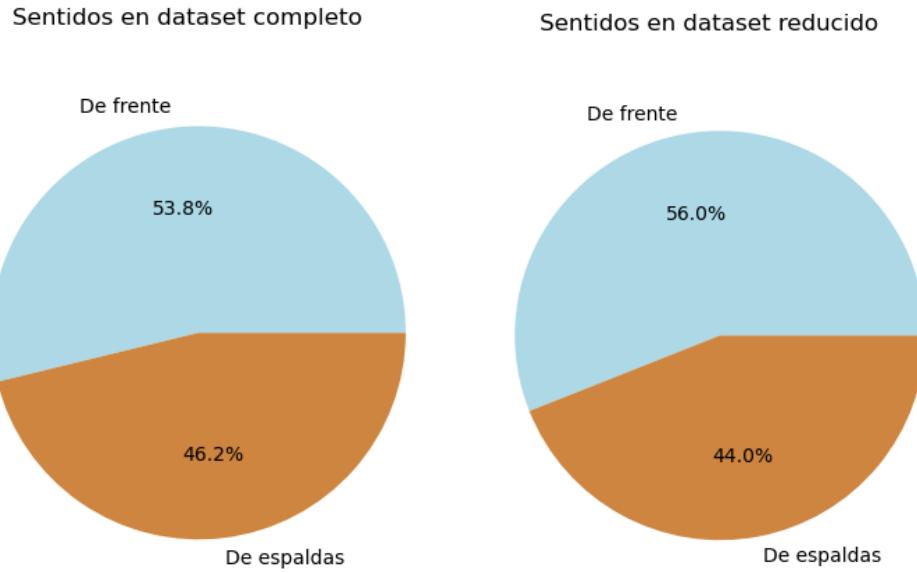


Fig. 4.9. Distribución del sentido de la marcha de los vehículos.

En la figura 4.9 vemos que la distribución del sentido de los vehículos está bastante equilibrada, y por tanto no será necesario realizar aumento de datos en este sentido.

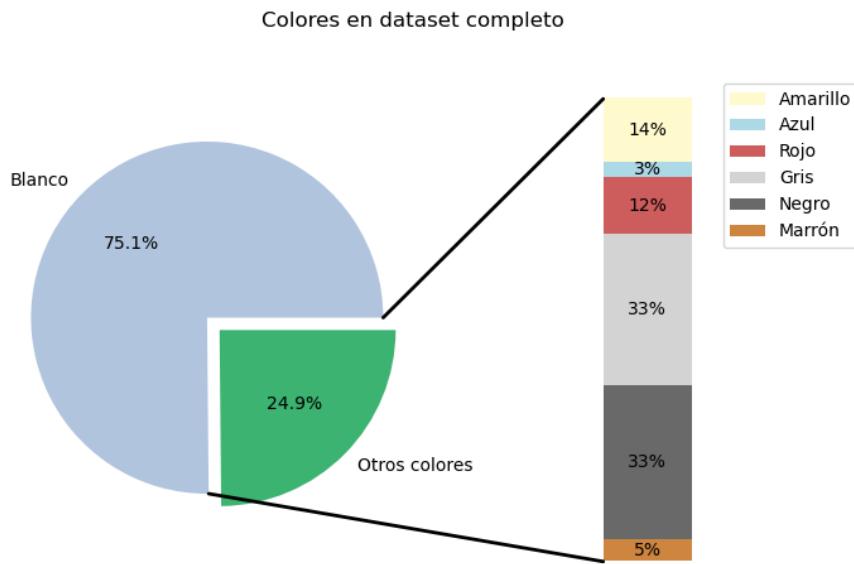


Fig. 4.10. Distribución de colores en el dataset completo.

En las figuras 4.10 y 4.11, por simplicidad de visualización, se muestran los colores como si cada furgoneta fuese enteramente de un color; sin embargo, aproximadamente un cuarto de las furgonetas que no eran blancas presentaban varios colores principales, lo cual es positivo, pues introduce más variación, pero también significa que hay muy pocos casos de cada color. Vemos que, aparte del blanco, los colores principales son el gris, el negro y el rojo.

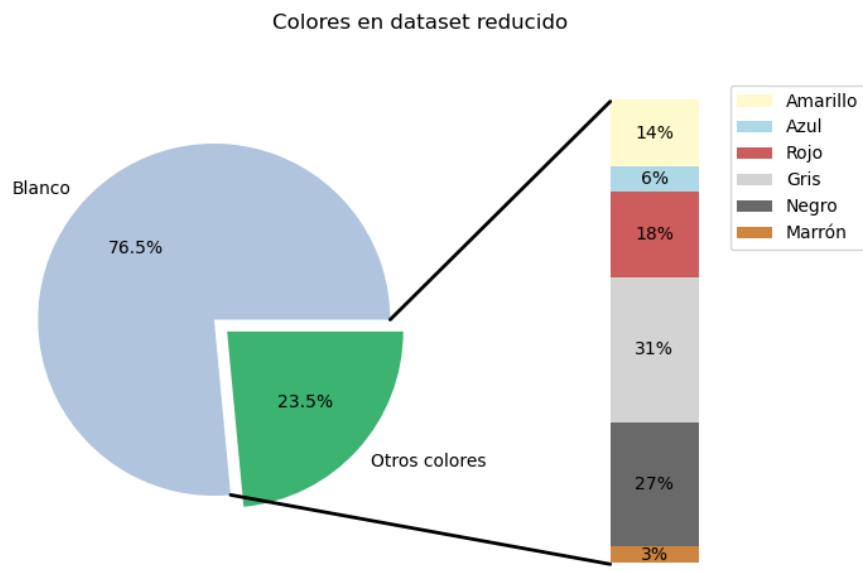


Fig. 4.11. Distribución de colores en el dataset reducido.

Es interesante comprobar que las distribuciones de datos no cambian significativamente entre el dataset completo y el reducido, lo cual podría indicar que había una proporción más o menos constante de imágenes redundantes a lo largo de todo el dataset, y que no hemos disminuido la cantidad de información disponible.

En la figura 4.12 vemos la distribución de clases en los datasets completo y reducido.

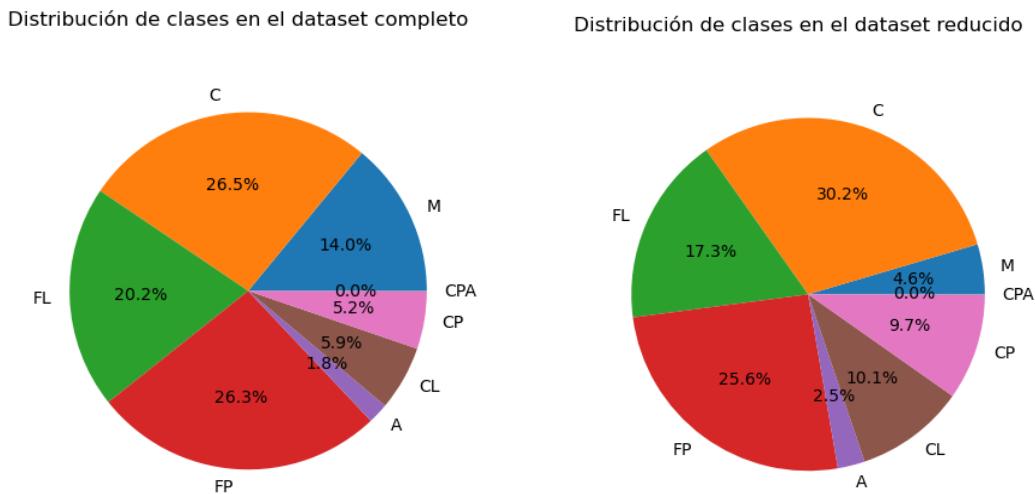


Fig. 4.12. Distribución de clases de los vehículos.

Se puede observar un desequilibrio de clases considerable, sobre todo en lo que respecta a los camiones pesados articulados, de los cuales hay apenas 3 instancias, los autobuses y las motos. Vemos que al eliminar las imágenes redundantes, se ha reducido un poco la diferencia entre algunas clases, pero también hemos perdido muchas instancias

de vehículos que ya de por si estaban pobemente representados. Este dataset reducido puede ser interesante para realizar el ajuste de hiperparámetros, y puede que ayude a evitar el sobreajuste al conjunto de entrenamiento, pero no lo utilizaremos tal cual en el entrenamiento final del modelo, si no que habrá que realizar aumento de datos.

4.3.3. Aumento de datos

La mayor limitación del dataset es la falta de diversidad de los datos. Podemos intentar paliar esto introduciendo las siguientes técnicas de aumento de datos:

- Editando los valores de matiz, saturación y brillo de las imágenes, para simular diferentes condiciones de iluminación, y para introducir más colores de furgonetas.
- Añadiendo un filtro de lluvia o niebla, para simular diferentes condiciones meteorológicas.
- Se puede paliar la falta de variación de perspectivas y localizaciones girando las imágenes horizontalmente, o rotándolas algunos grados.
- Podemos tratar el desequilibrio de clases dando una probabilidad mayor de ser aumentadas a las imágenes con vehículos de clases subrepresentadas.

No realizaremos el aumento de datos sobre el *dataset* completo, dato que volveríamos a introducir las imágenes redundantes, pero, dado que ya las tenemos, al recorrer cada imagen, si se decide que se va a realizar aumento de datos sobre la misma, este se realizará sobre una de las imágenes muy similares que habíamos eliminado previamente, para hacer uso de toda la variación posible, por mínima que sea.

Para decidir si se realizará aumento a una imagen o no, se asigna a cada una una probabilidad que depende de qué tipos de vehículos hay en la imagen, y si están cerca o lejos (se dará más prioridad a imágenes con una proporción alta de vehículos de clases poco representadas, y a furgonetas que estén cerca, puesto que son las que más información pueden aportar). Esta probabilidad se calcula de la siguiente forma:

$$P_{\text{aumento}}(\text{imagen}) = \left(\frac{1}{n_{\text{img}}} \sum_{c \in v_{\text{img}}} \frac{n_{\text{ideal}} - v_{\text{totales}}[c]}{n_{\text{ideal}}} + \min(v_{\text{totales}}) \right) \times d_{\text{factor}}$$

Donde:

- n_{ideal} el número de vehículos de la clase más representada,
- v_{totales} es una lista que representa el número de vehículos totales por cada clase,
- v_{img} es una lista en la cual hay un elemento por vehículo presente en la imagen, que representa su clase.

- n_{img} es el número de vehículos en esa imagen,
- d_{factor} se refiere a la distancia, será 0,6 si es cerca, y 0,3 si es lejos.

Para realizar el aumento de datos mencionado, utilizaremos la librería Albumentations, que cuenta con numerosas funciones para realizar diferentes técnicas de aumento de datos.

Para simular diferentes espacios y perspectivas, utilizaremos las siguientes transformaciones:

- Girar la imagen horizontalmente.
- Añadir sombras aleatorias.
- Rotar la imagen unos grados.
- Escoger un recorte de la imagen.
- Modificar el brillo y contraste.
- Modificar los valores hsv.

Para simular diferentes colores y condiciones lumínicas: modificar el brillo, contraste y valores de hsv.

Para simular diferentes condiciones meteorológicas: añadir filtros de lluvia, niebla y destellos solares.

Para añadir occlusiones, simulando que haya más tráfico: eliminar ciertos rectángulos de la imagen, poniéndolos a negro (*CoarseDropout*).

Se puede ver el proceso completo en el Anexo C.

En la figura 4.13 podemos ver la distribución de clases del *dataset* aumentado, donde algunas de las clases están más equilibradas.

4.4. Ajuste de hiperparámetros

Para realizar el ajuste de hiperparámetros, se utilizó el algoritmo PSO (Particle Swarm Optimization). Este es un algoritmo que forma parte de los algoritmos biológicos, consiste en un número de partículas, de las cuales cada una representa una configuración de hiperparámetros.

Cada partícula tiene una posición inicial y una velocidad asociada, que se irán actualizando conforme al valor de una función de coste que obtiene la partícula, y que depende de la configuración que tenga en ese momento. Un componente de la velocidad empuja a la partícula en la dirección de la mejor solución que ha encontrado, otro componente, la

Distribución de clases en el dataset aumentado

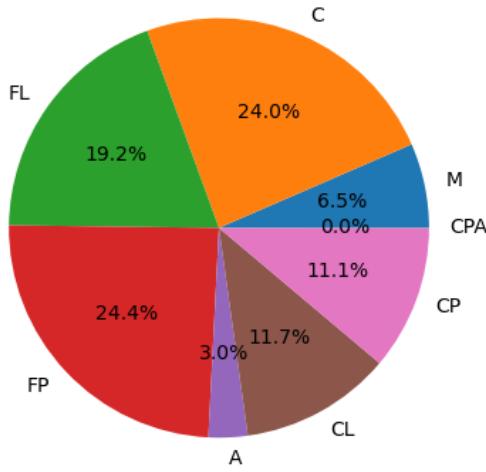


Fig. 4.13. Distribución de clases en el dataset aumentado.

inercia, la empuja a seguir la dirección que ya llevaba, y un último componente, el factor social, la empuja a moverse hacia el punto en el que se haya encontrado una mejor solución hasta el momento entre todas las partículas. A continuación, se detalla la formulación matemática del algoritmo.

4.4.1. Algoritmo de optimización por enjambre de partículas (PSO) [22]

Dado un conjunto de partículas en un espacio de búsqueda, cada una con una posición \mathbf{x}_i y una velocidad \mathbf{v}_i , el objetivo del PSO es encontrar la mejor posición global \mathbf{x}^* que minimiza (o maximiza) una función objetivo $f(\mathbf{x})$. El algoritmo se puede describir mediante las siguientes ecuaciones:

Actualización de la velocidad

La velocidad de cada partícula \mathbf{v}_i se actualiza utilizando la siguiente ecuación:

$$\mathbf{v}_i^{(t+1)} = \omega \mathbf{v}_i^{(t)} + c_1 r_1 (\mathbf{p}_i - \mathbf{x}_i^{(t)}) + c_2 r_2 (\mathbf{g} - \mathbf{x}_i^{(t)})$$

donde:

- $\mathbf{v}_i^{(t)}$ es la velocidad de la partícula i en el tiempo t ,
- ω es el factor de inercia,
- c_1 y c_2 son los coeficientes de aceleración (o constantes de aprendizaje),

- r_1 y r_2 son números aleatorios uniformemente distribuidos en el intervalo $[0, 1]$,
- \mathbf{p}_i es la mejor posición conocida de la partícula i ,
- \mathbf{g} es la mejor posición global conocida.

Actualización de la posición

La posición de cada partícula \mathbf{x}_i se actualiza usando:

$$\mathbf{x}_i^{(t+1)} = \mathbf{x}_i^{(t)} + \mathbf{v}_i^{(t+1)}$$

Actualización de las mejores posiciones

La mejor posición conocida por cada partícula \mathbf{p}_i se actualiza si la nueva posición $\mathbf{x}_i^{(t+1)}$ es mejor que \mathbf{p}_i :

$$\mathbf{p}_i = \begin{cases} \mathbf{x}_i^{(t+1)} & \text{si } f(\mathbf{x}_i^{(t+1)}) < f(\mathbf{p}_i) \\ \mathbf{p}_i & \text{si no} \end{cases}$$

La mejor posición global \mathbf{g} se actualiza si alguna de las posiciones \mathbf{p}_i es mejor que \mathbf{g} :

$$\mathbf{g} = \begin{cases} \mathbf{p}_i & \text{si } f(\mathbf{p}_i) < f(\mathbf{g}) \\ \mathbf{g} & \text{si no} \end{cases}$$

Como parámetros de entrada para el algoritmo PSO, se utilizó $\omega = 0,7$ y $c_1 = c_2 = 1,7$, que entran dentro de los posibles rangos señalados en [22]. La función de coste definida es la métrica mAP50-95 obtenida después de entrenar la red con los siguientes parámetros:

```
def cost_func(x, n_iter: int, n_part: int) -> float:
    model = YOLO("yolov8n.pt")

    model.train(
        data=r"/home/maria/TFM/data/datasets/filtered_DATASET_v2/cfg.yaml",
        project=r"/home/maria/TFM/data/results/filtered_DATASET_v2/PSO_batch",
        name=f"{n_iter}_{n_part}_{x[0]}_{x[1]}_{x[2]}",
        epochs=40,
        patience=5,
        imgsz=608,
        device="cuda:0" if cuda.is_available() else "cpu",
        exist_ok=True,
        seed=4,
        optimizer="Adam",
```

```

        close_mosaic=0,
        hsv_h=0,
        hsv_s=0,
        hsv_v=0,
        translate=0,
        scale=0,
        fliplr=0,
        mosaic=0,
        verbose=False,
        cos_lr=True,
        batch=x[0],
        lr0=0.0010192339694602597,
        lrf=0.01,
        momentum=0.9171775316059347,
        weight_decay=x[1],
        cls=x[2],
        plots=False,
    )

    return model.val().box.map

```

Se decidió utilizar el optimizador “Adam” después de comprobar que era el que mejores resultados generaba de los que había disponibles, utilizando valores de hiperparámetros por defecto, y el *dataset* sin aumento de datos. Podemos ver los resultados de este en la figura 4.14. Los resultados obtenidos por el resto de optimizadores se encuentran en el Anexo 1.

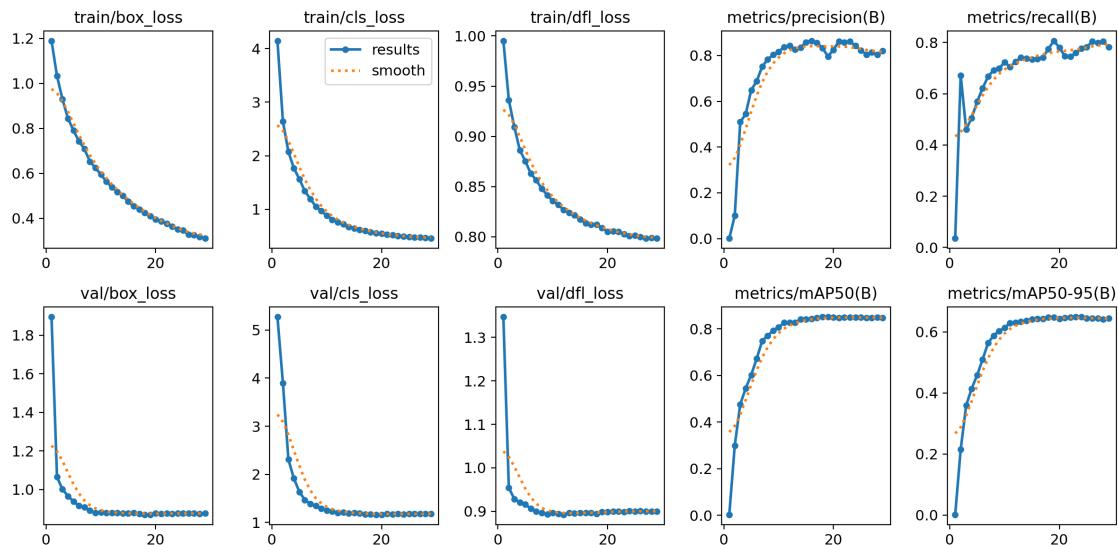


Fig. 4.14. Resultados del entrenamiento con el algoritmo de optimización Adam y valores de hiperparámetros por defecto. Tiempo empleado: 244.6s.

Observamos que para ser un primer entrenamiento, los resultados no son terribles.

Vemos que las gráficas de pérdidas disminuyen en una curva suave según avanzan las épocas, lo cual indica que el modelo continúa aprendiendo con cada iteración. Si es cierto que estas curvas son significativamente más bruscas en el conjunto de validación, lo que indica que probablemente haya un sobreajuste con los datos de entrenamiento, nada sorprendente teniendo en cuenta la falta de variedad en las imágenes del *dataset* sin aumento.

A parte de esto, la precisión y el *recall* se estabilizan en un valor de alrededor de 0.8, que de primeras es aceptable, pero debe mejorar. Esto nos indica que está reconociendo un 80 % de los vehículos de las imágenes (*recall*), y que de esos, clasifica correctamente otro 80 %. Observamos una oscilación importante en el *recall* en las primeras épocas, esto nos indica que sería conveniente ajustar la tasa de aprendizaje, cosa que hacemos en el siguiente paso.

Después de la elección del algoritmo de optimización, primero se ejecutó PSO con los parámetros por defecto (Anexo D), y eliminando el aumento de datos por defecto de YOLOv8, y se optimizaron la tasa de aprendizaje inicial (*lr0*) y final (*lrf*) y el momento (*momentum*). Después, con los mejores valores obtenidos, se hizo lo mismo para los parámetros del tamaño de *batch*, *weight_decay*³ y *cls*⁴.

El algoritmo se ejecutó las dos veces con 25 partículas y 20 iteraciones, y los resultados obtenidos fueron demasiado buenos, con un mAP50-95 de 0.713, siendo evidente que existía un sobreajuste muy grande al conjunto de test, puesto que este valor no es realista viendo las limitaciones presentes en el *dataset* de partida.

Así pues, para paliar el sobreajuste, se volvió a entrenar el modelo con los parámetros obtenidos con PSO, pero esta vez utilizando el *dataset* aumentado y validación cruzada. En el siguiente apartado se presentan los resultados.

³La disminución de peso es una técnica de regularización que provoca que los pesos de las neuronas no suban demasiado, lo cual ayuda a prevenir el sobreajuste.

⁴Un valor de *cls* alto indica al modelo que de más importancia a clasificar correctamente, a ser más preciso. Se refiere a la pérdida de clase.

5. RESULTADOS Y ANÁLISIS

Podemos ver los resultados obtenidos en las figuras 5.1 a 5.5.

En la figura 5.1, observamos que las pérdidas obtenidas, tanto en entrenamiento como en validación, van disminuyendo con cada época. En este caso, las curvas obtenidas con el conjunto de validación se parecen bastante más a las del conjunto de entrenamiento, lo que nos indica que hemos eliminado algo de sobreajuste, aunque es cierto que la pérdida de clase tiene una bajada muy brusca al principio, por lo que quizás sería conveniente realizar un mejor ajuste del hiperparámetro *cls*. También debemos tener en cuenta que las limitaciones de este *dataset*, aunque hayan disminuido con el aumento de datos, siguen ahí. Lo que sería necesario para obtener un mejor modelo es añadir imágenes más representativas al *dataset*.

Sobre el resto de métricas, vemos que la precisión se estabiliza en torno a 0.9, y el *recall* queda algo más alto, ambos valores positivos. Este último también tiene una oscilación importante al principio, pero se estabiliza lo suficientemente rápido. El maP50 se estabiliza algo por debajo del 0.9, y el maP50-95 algo más bajo, en torno al 0.8. Esto es de esperar, dado que el maP50-95 es una métrica mucho más exigente, un valor de 0.8 es significativamente bueno.

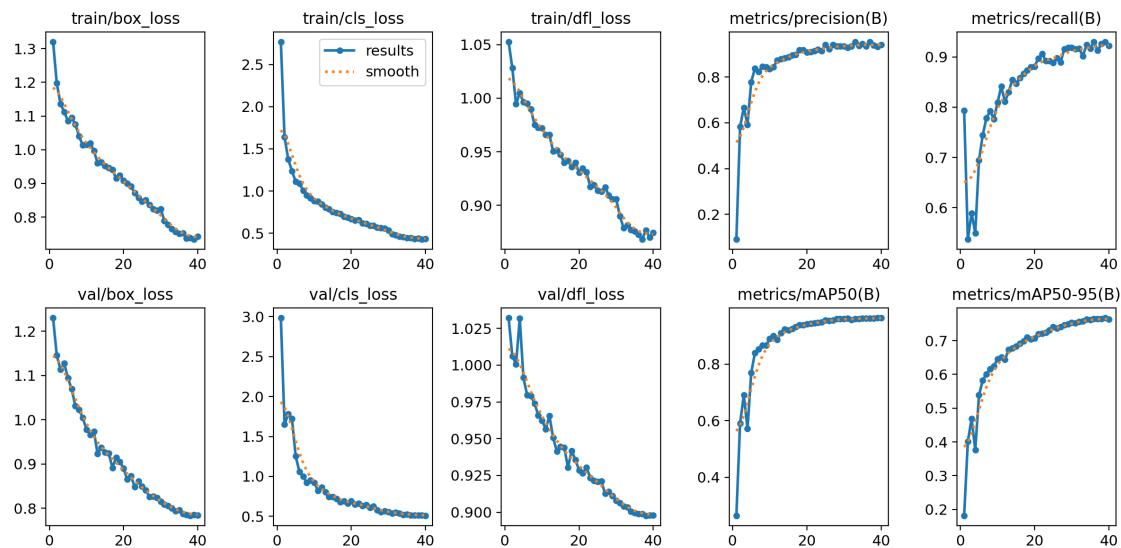


Fig. 5.1. Métricas obtenidas en el entrenamiento final.

En la figura 5.2 tenemos la matriz de confusión normalizada. Las mayores imprecisiones se encuentran entre las clases 1 a 3; es decir, coches, furgonetas ligeras, y furgonetas pesadas, lo cual nos indica que hay lugar para mejorar. Por otro lado, son los tipos de vehículos que más se parecen entre si, de modo que la confusión del modelo es, en cierto modo, comprensible.

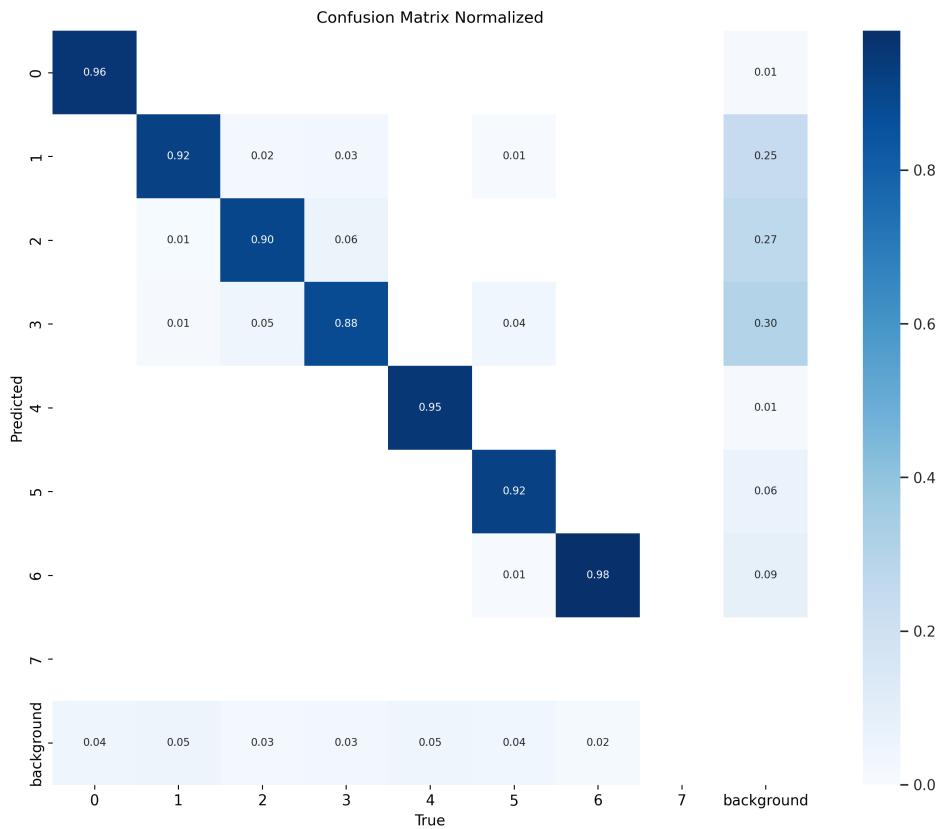


Fig. 5.2. Matriz de confusión normalizada.

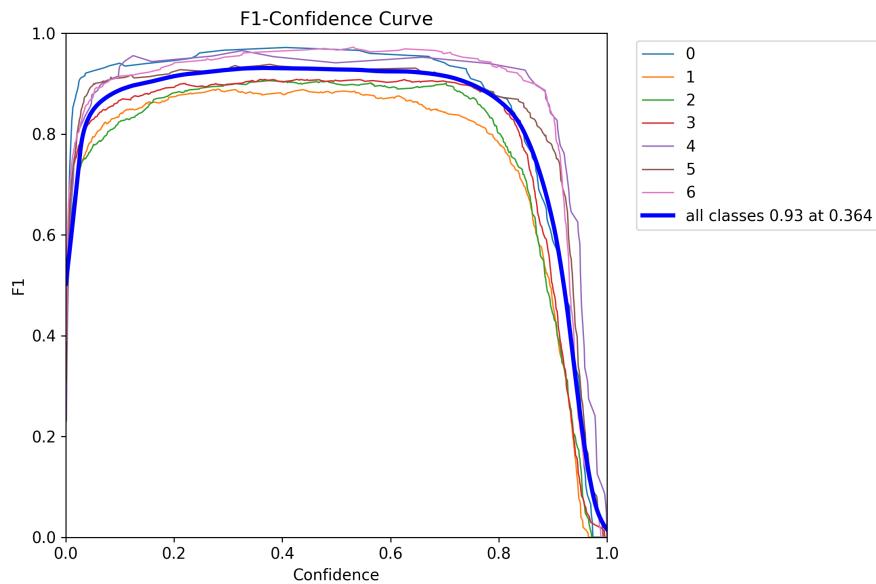


Fig. 5.3. Curva F1.

Sobre la curva F1 (figura 5.3), vemos que llega a un valor estable de 0.93 para un umbral de confianza de 0.364, que es relativamente bajo, lo cual es una buena señal. Se mantiene más estable hasta el 0.7, donde empieza a decaer. Esto nos indica que a partir de dicho umbral, se hacen predicciones más precisas, pero mucha menos cantidad de

predicciones, con lo que el *recall* sufre.

Fijándonos en las curvas de cada clase en concreto, vemos que no hay grandes desvíos, pero de nuevo observamos que las clases 1, 2 y 3 obtienen resultados peores al resto, a pesar de ser las más representadas en el *dataset*. Para analizar más profundamente estos resultados, podemos mirar las curvas de precisión (figura 5.4) y *recall* (figura 5.5).

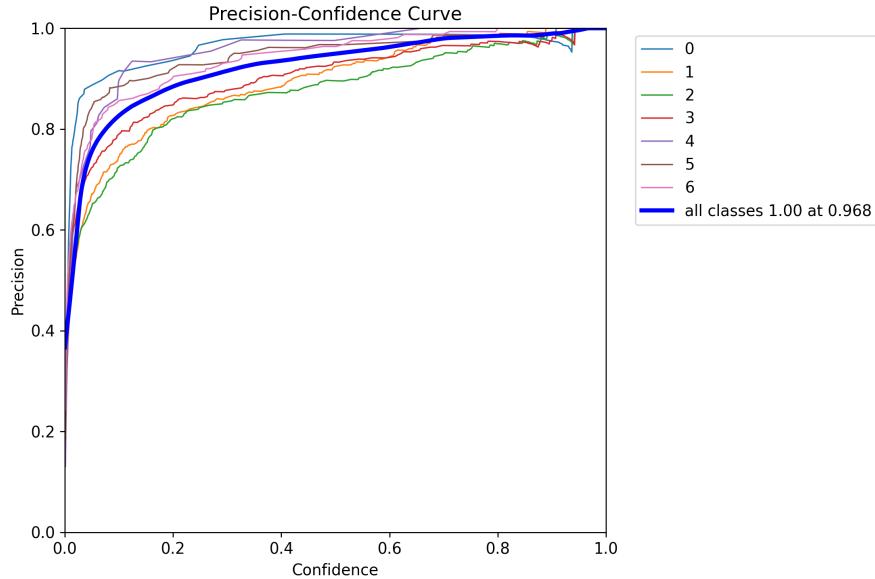


Fig. 5.4. Curva de precisión.

Efectivamente, los coches, y ambos tipos de furgonetas se detectan con menor precisión. Por otro lado, vemos que a partir de valores de confianza de 0.3, la precisión ya sube por encima de 0.9, y las predicciones hechas con una confianza mayor que 0.968 aciertan siempre.

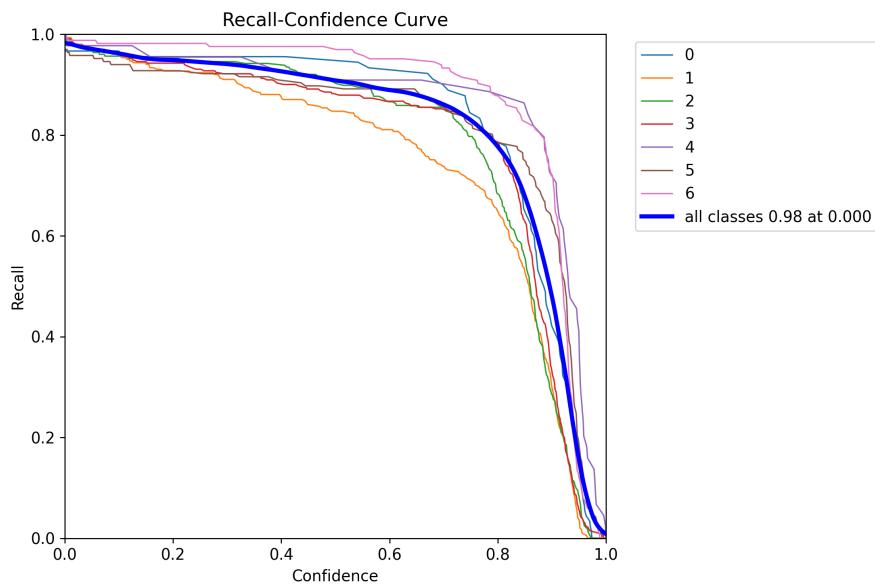


Fig. 5.5. Curva de recall.

En la curva de *recall*, lo más significativo que vemos es que la curva correspondiente a la clase “coche”, queda bastante por debajo que el resto de clases, lo que nos dice que el modelo deja bastantes coches sin detectar.

6. CONCLUSIONES

Finalmente hemos obtenido unos resultados bastante buenos, si bien habría que ver cómo se comporta el modelo con imágenes totalmente ajenas al *dataset* de partida. También hemos comprobado la importancia de un buen tratamiento de los datos, y especialmente de las variedades de los mismos.

En general, se podría mejorar el proceso de ajuste de los hiperparámetros, sin olvidar que la inclusión de nuevas imágenes en el *dataset* también sería muy beneficiosa.

Podemos dar los objetivos planteados por conseguidos, aunque queda espacio para la mejora, y este modelo todavía no sería apto para utilizarlo en peajes inteligentes, puesto que le falta precisión en sus predicciones y, en este contexto, se necesita una precisión muy alta, no sólo alta, dado que de ello depende que se cobre el importe correcto a los usuarios de las carreteras.

6.1. Trabajo futuro

Para mejorar el ajuste de hiperparámetros, se podría ejecutar el algoritmo de optimización PSO con el *dataset* aumentado y utilizando validación cruzada en la función de coste también. A pesar de que el tiempo de entrenamiento subiría considerablemente, merece la pena puesto que pueden mejorar mucho los resultados. También se podría ajustar los parámetros de entrada de PSO.

BIBLIOGRAFÍA

- [1] M. Bie, Y. Liu, G. Li, J. Hong y J. Li, «Real-time vehicle detection algorithm based on a lightweight You-Only-Look-Once (YOLOv5n-L) approach,» *Expert Systems with Applications*, vol. 213, p. 119108, 2023. doi: <https://doi.org/10.1016/j.eswa.2022.119108>. [En línea]. Disponible en : <https://www.sciencedirect.com/science/article/pii/S0957417422021261>.
- [2] , *Peajes que reducen el tráfico y la contaminación en España*, <https://www.bankinter.com/blog/finanzas-personales/los-nuevos-peajes-inteligentes-espana>, [Último acceso: 08-09-2024].
- [3] , *SCOOT® - TRL Software — trlsoftware.com*, <https://trlsoftware.com/products/traffic-control/scoot/>, [Último acceso: 07-09-2024].
- [4] C. Luo, J. Chen, X. Feng, J. Zhang y J. Li, «BSL: Sustainable Collaborative Inference in Intelligent Transportation Systems,» *IEEE Transactions on Intelligent Transportation Systems*, vol. 24, n.º 12, pp. 15995-16005, 2023. doi: <10.1109/TITS.2023.3308370>.
- [5] T. Ramirez-Guerrero, M. Toro, G. A. Villegas López y L. F. Castañeda, «Low-cost computational systems applied to physical architectures in public transportation systems of intermediate cities,» *Journal of Physics: Conference Series*, vol. 1702, n.º 1, p. 012018, nov. de 2020. doi: <10.1088/1742-6596/1702/1/012018>. [En línea]. Disponible en : <http://dx.doi.org/10.1088/1742-6596/1702/1/012018>.
- [6] F. Kelly, «Road Pricing: Addressing Congestion, Pollution and the Financing of Britain's Road,» *Ingenia*, vol. 39, pp. 36-42, 2006.
- [7] Akash, *Top Electronic Toll Collection Companies: Innovating Tolling Technology*, <https://www.strategymrc.com/blog/top-electronic-toll-collection-companies/>, [Accessed 19-09-2024].
- [8] J. De las Heras Molina, J. Gómez Sánchez y J. M. Vassallo Magro, «Electronic Toll Collection Systems and their Interoperability: The State of Art,» en *Libro de Actas CIT2016. XII Congreso de Ingeniería del Transporte*, ép. CIT2016, Universitat Politècnica València, jun. de 2016. doi: <10.4995/cit2016.2016.3186>. [En línea]. Disponible en : <http://dx.doi.org/10.4995/CIT2016.2016.3186>.
- [9] N. Schindler, «GNSS-Based Tolling: State-of-the-Art and Beyond,» en *IRF Annual Conference 2023*, [Último acceso: 14-09-2024], International Road Federation, 2023. [En línea]. Disponible en : https://irfnet.ch/wp-content/uploads/2023/11/Norbert-Schindler-GNSS-Based-Tolling-State-of-the-Art-and-Beyond_IRF-Annual-Conference-2023.pdf.

- [10] A. Shukla, P. Bhattacharya, S. Tanwar, N. Kumar y M. Guizani, «DwaRa: A Deep Learning-Based Dynamic Toll Pricing Scheme for Intelligent Transportation Systems,» *IEEE Transactions on Vehicular Technology*, vol. 69, n.º 11, pp. 12 510-12 520, 2020. doi: [10.1109/TVT.2020.3022168](https://doi.org/10.1109/TVT.2020.3022168).
- [11] A. Gholamhosseiniyan y J. Seitz, «Vehicle Classification in Intelligent Transport Systems: An Overview, Methods and Software Perspective,» *IEEE Open Journal of Intelligent Transportation Systems*, vol. 2, pp. 173-194, 2021. doi: [10.1109/OJITS.2021.3096756](https://doi.org/10.1109/OJITS.2021.3096756).
- [12] M. Bugeja, A. Dingli, M. Attard y D. Seychell, «Comparison of Vehicle Detection Techniques applied to IP Camera Video Feeds for use in Intelligent Transport Systems,» *Transportation Research Procedia*, vol. 45, pp. 971-978, 2020, Transport Infrastructure and systems in a changing world. Towards a more sustainable, reliable and smarter mobility.TIS Roma 2019 Conference Proceedings. doi: <https://doi.org/10.1016/j.trpro.2020.02.069>. [En línea]. Disponible en : <https://www.sciencedirect.com/science/article/pii/S2352146520301198>.
- [13] Ultralytics, *Ultralytics YOLOv8 Tasks — docs.ultralytics.com*, <https://docs.ultralytics.com/tasks>, [Último acceso: 02-09-2024].
- [14] A. Carrio, C. Sampedro Pérez, A. Rodríguez Ramos y P. Campoy, «A Review of Deep Learning Methods and Applications for Unmanned Aerial Vehicles,» *Journal of Sensors*, vol. 2017, pp. 1-13, ago. de 2017. doi: [10.1155/2017/3296874](https://doi.org/10.1155/2017/3296874).
- [15] ¿Qué son las redes neuronales convolucionales? | IBM — [ibm.com](https://www.ibm.com/es-es/topics/convolutional-neural-networks), <https://www.ibm.com/es-es/topics/convolutional-neural-networks>, [Último acceso: 27-08-2024].
- [16] ¿Qué es el preprocesamiento de datos? Definición, importancia y pasos — astera.com, <https://www.astera.com/es/type/blog/data-preprocessing>, [Último acceso: 28-08-2024].
- [17] Ultralytics, *YOLO Métricas de rendimiento — docs.ultralytics.com*, <https://docs.ultralytics.com/es/guides/yolo-performance-metrics/>, [Último acceso: 04-09-2024].
- [18] Guía para elegir furgoneta: ¿L1 H1, L1 H2, L3 H3, L4 H3?... el tamaño sí que importa — espaciofurgo.com, <https://www.espaciofurgo.com/guia-para-elegir-furgoneta-l1-h1-l1-h2-l3-h3-l4-h3-el-tamano-si-que-importa>, [Último acceso: 02-09-2024].
- [19] ¿Buscando una furgoneta pequeña? Aquí las pocas alternativas — autonucion.com, <https://www.autonucion.com/furgonetas-pequenas-comerciales>, [Último acceso: 02-09-2024].
- [20] Nuevo Ford Ranger - pick up 4x4 | Ford ES — [ford.es](https://www.ford.es/furgonetas-pick-up/nuevo-ranger), <https://www.ford.es/furgonetas-pick-up/nuevo-ranger>, [Último acceso: 02-09-2024].

- [21] *Furgonetas pequeñas: 3 modelos para 9 marcas* — *coches.net*, <https://www.coches.net/noticias/furgonetas-pequenas>, [Último acceso: 13-09-2024].
- [22] T. M. Shami et al., «Particle Swarm Optimization: A Comprehensive Survey,» *IEEE Access*, vol. 10, pp. 10 031-10 061, 2022. doi: [10.1109/ACCESS.2022.3142859](https://doi.org/10.1109/ACCESS.2022.3142859).

ANEXO 1 RESULTADOS DEL ENTRENAMIENTO EN FUNCIÓN DEL OPTIMIZADOR UTILIZADO

Todos los resultados se obtuvieron utilizando los valores por defecto en el entrenamiento.

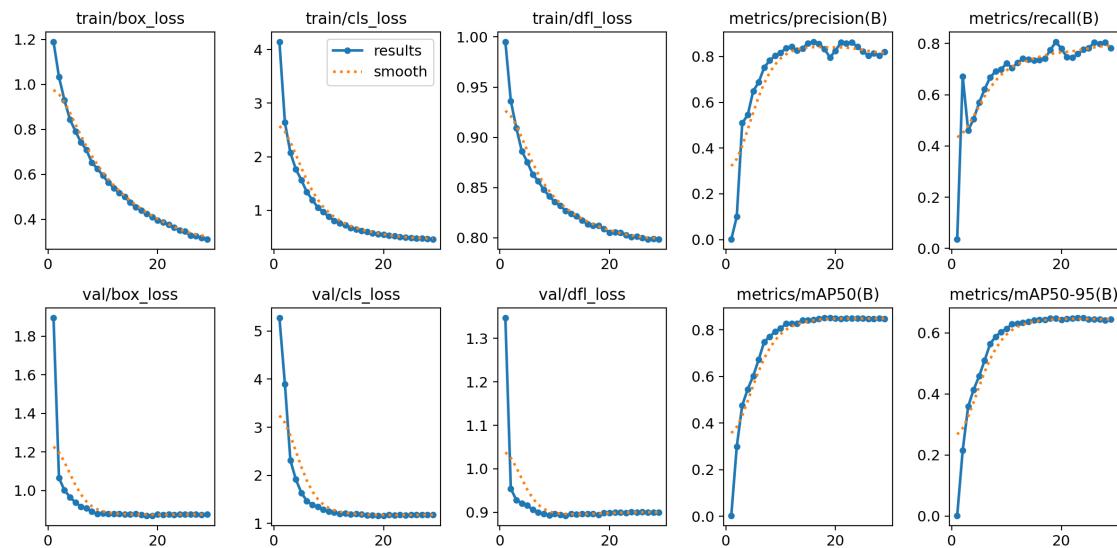


Fig. 6.1. Resultados del entrenamiento con el algoritmo de optimización Adam. Tiempo empleado 244.6s.

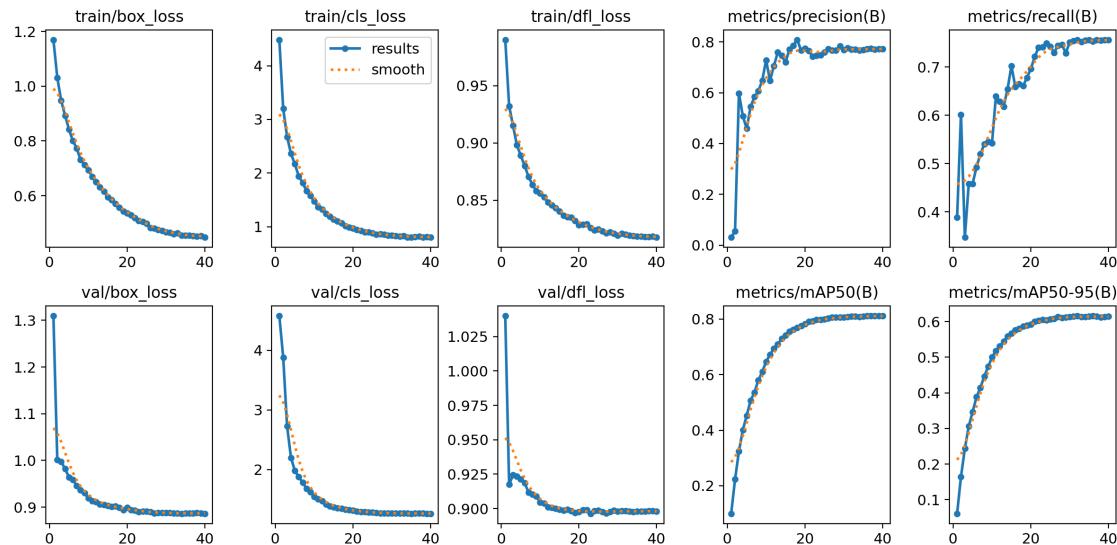


Fig. 6.2. Resultados del entrenamiento con el algoritmo de optimización Adamax. Tiempo empleado 346.1s.

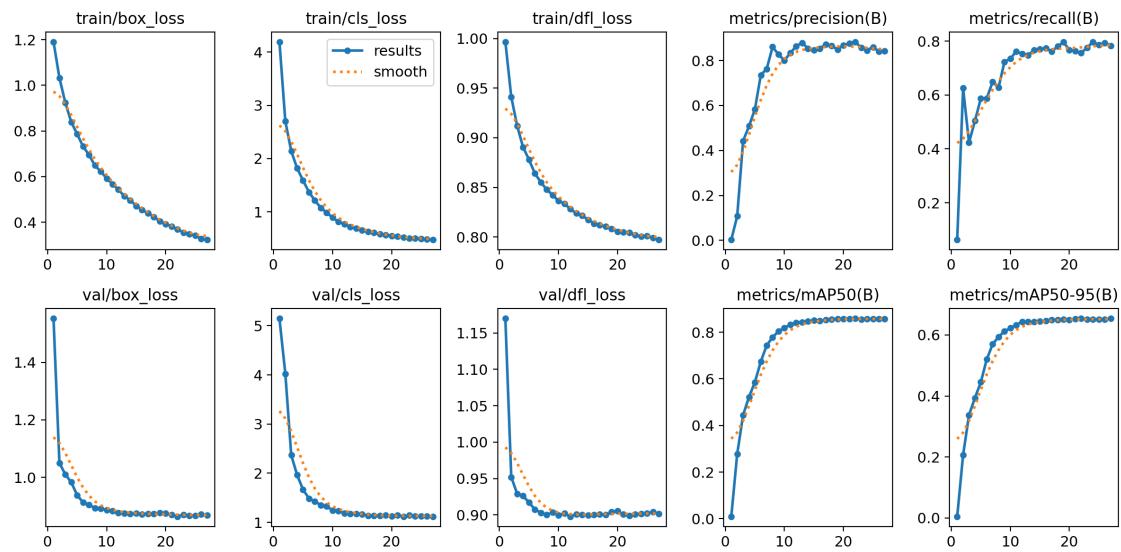


Fig. 6.3. Resultados del entrenamiento con el algoritmo de optimización AdamW. Tiempo empleado 274.2s.

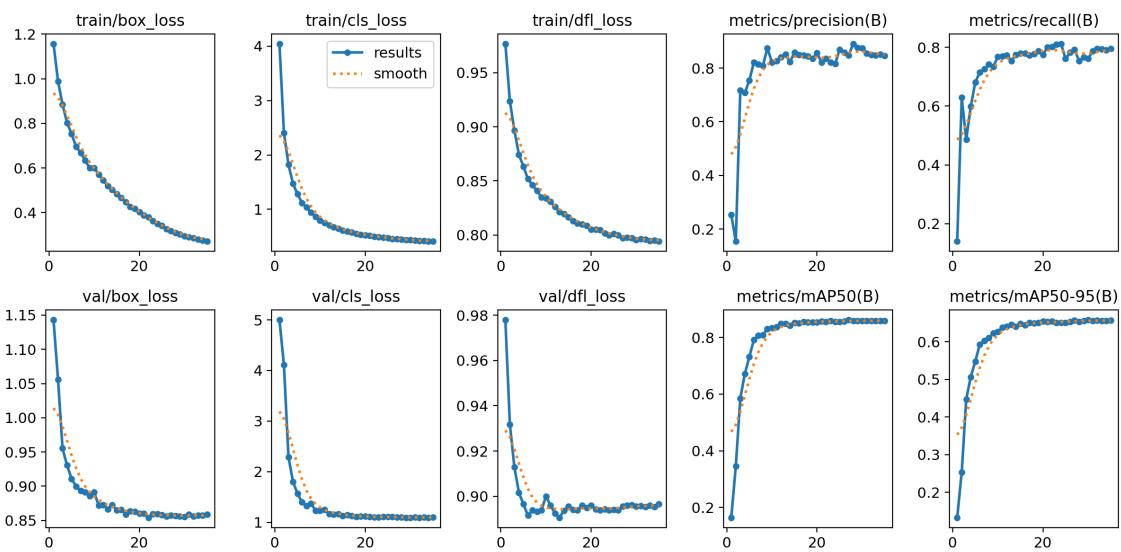


Fig. 6.4. Resultados del entrenamiento con el algoritmo de optimización NAdam. Tiempo empleado 318.6s.

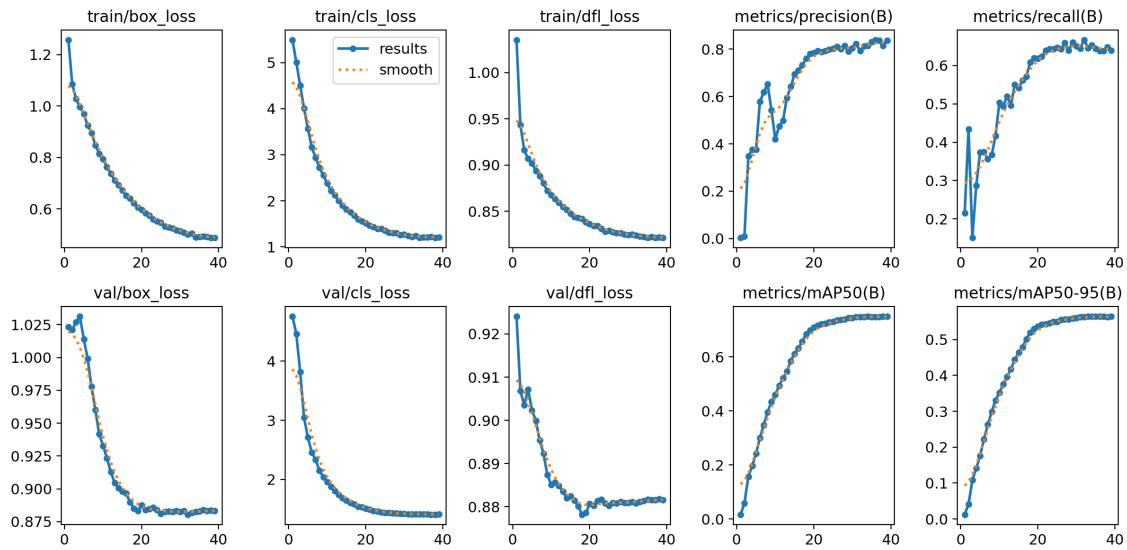


Fig. 6.5. Resultados del entrenamiento con el algoritmo de optimización RAdam. Tiempo empleado 343.9s.

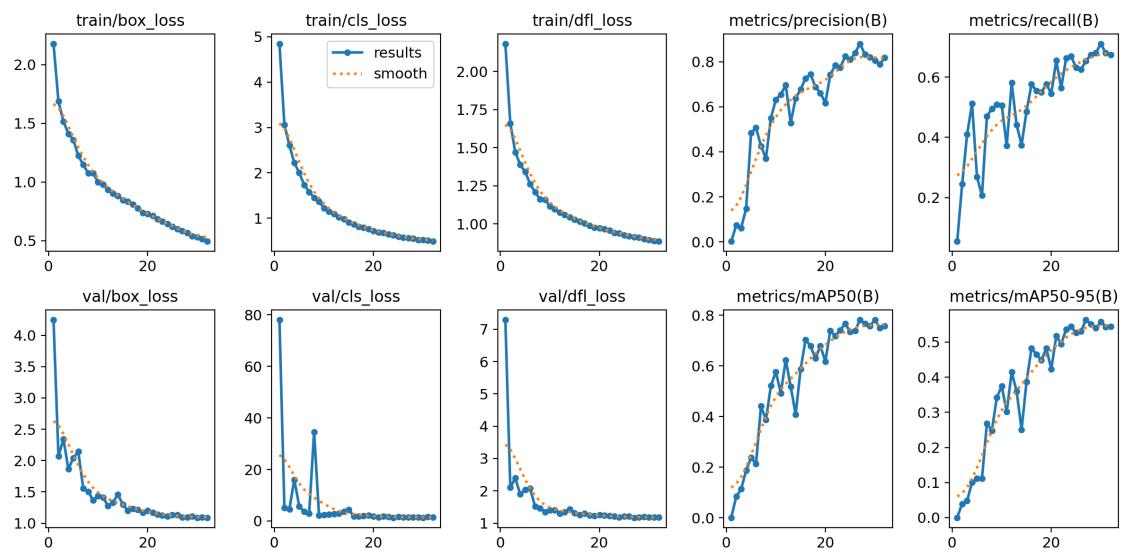


Fig. 6.6. Resultados del entrenamiento con el algoritmo de optimización RMSProp. Tiempo empleado 316.3s.

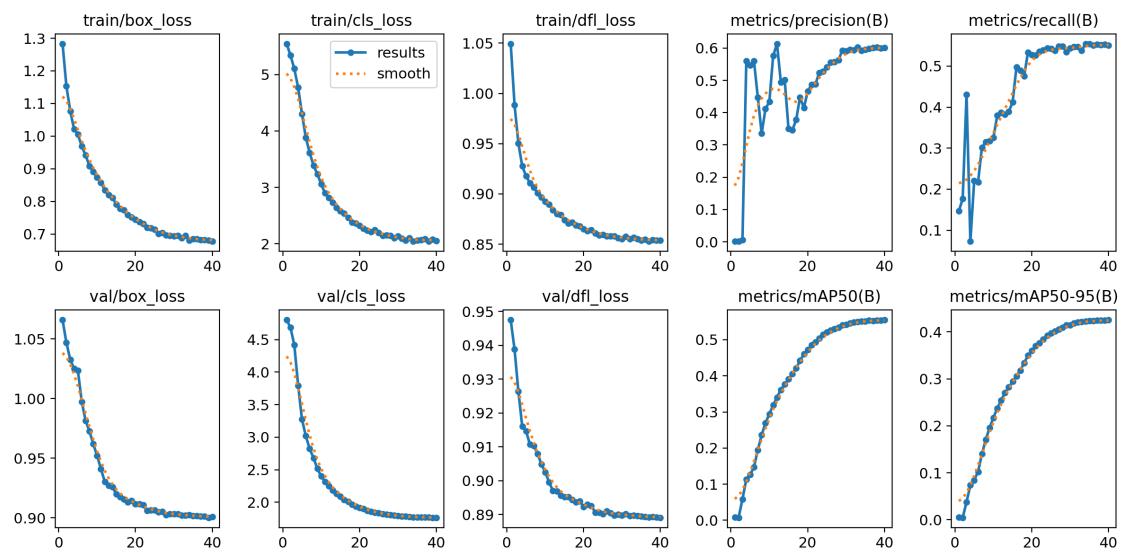


Fig. 6.7. Resultados del entrenamiento con el algoritmo de optimización SGD. Tiempo empleado 230.1s.

ANEXO A SCRIPT CLASIFY_VANS.PY

```
import os
import tkinter as tk
from tkinter import ttk
from PIL import ImageTk, Image
import cv2

NEW_IMG_H = 550

class Display(tk.Tk):
    def __init__(self,
                 image_folder,
                 label_folder,
                 new_label_folder,
                 bb_image_folder,
                 dataset_info_folder,
                 ):
        super().__init__()
        self.image_folder = image_folder
        self.label_folder = label_folder
        self.new_label_folder = new_label_folder
        self.bb_image_folder = bb_image_folder
        self.dataset_info_folder = dataset_info_folder

        self.geometry("1200x800")

        self.id = 0

        self.file_names = [
            f[:-4] for f in os.listdir(label_folder) if f.endswith(".txt")]
        ]
        self.n_pics = len(self.file_names)
        self.imgs_info = [None] * self.n_pics

        self.frame = ttk.Frame(master=self)
        self.inputs_frame = ttk.Frame(self.frame)
        self.idpp_entry_var = tk.StringVar(value=str(self.id + 1))
        self.idpp_entry = ttk.Entry(
            master=self.frame, width=5, textvariable=self.idpp_entry_var
        )

        self.update_disp()

        self.frame.pack(expand=True)
        self.bind("<Left>", func=self.left)
        self.bind("<Right>", func=self.right)
        self.bind("<Button-3>", func=self.right)
        self.bind("<Button-2>", func=self.left)
        # self.bind('<space>', func=self.right)
        self.bind(".", func=self.toggle_pesada_ligera)
        self.bind("l", func=self.focus_next_van)
        self.bind("k", func=self.focus_previous_van)

    def toggle_pesada_ligera(self, *ignore):
        if self.focus_get() not in self.info["chk"]:
            self.focus_next_van(None)
```

```

index = self.info["chk"].index(self.focus_get())

if str(self.info["chk"][index]["state"]) in ["normal", "selected"]:
    self.info["chk"][index].invoke()

def focus_next_van(self, *ignore):
    if self.focus_get() not in self.info["chk"]:
        index = 0
        while str(self.info["chk"][index]["state"]) == "disabled":
            index += 1
            if index == len(self.info["chk"]):
                return
        self.info["chk"][index].focus_set()
    else:
        index = self.info["chk"].index(self.focus_get()) + 1
    if index == len(self.info["chk"]):
        index = 0
    first_index = index
    while str(self.info["chk"][index]["state"]) == "disabled":
        index += 1
        if index == len(self.info["chk"]):
            index = 0
        if index == first_index:
            return
    if index <= len(self.info["chk"]) - 1:
        self.info["chk"][index].focus_set()
    else:
        index = 0
        while str(self.info["chk"][index]["state"]) == "disabled":
            index += 1
            if index == len(self.info["chk"]):
                return
        self.info["chk"][index].focus_set()

def focus_previous_van(self, *ignore):
    if self.focus_get() not in self.info["chk"]:
        index = len(self.info["chk"]) - 1
        while str(self.info["chk"][index]["state"]) == "disabled":
            index -= 1
            if index < 0:
                return
        self.info["chk"][index].focus_set()
    else:
        index = self.info["chk"].index(self.focus_get()) - 1
    first_index = index
    while str(self.info["chk"][index]["state"]) == "disabled":
        index -= 1
        if index < 0:
            index = len(self.info["chk"]) - 1
        if index == first_index:
            return
    if index >= 0:
        self.info["chk"][index].focus_set()
    else:
        print("index1", index)
        index = len(self.info["chk"]) + index
        print("index2", index)
        while str(self.info["chk"][index]["state"]) == "disabled":
            index -= 1
            if index < 0:
                return
        self.info["chk"][index].focus_set()

def read_yolo_labels(self, label_path):

```

```

with open(label_path, "r") as file:
    lines = file.readlines()
classes = []
for line in lines:
    values = line.strip().split()
    classes.append(int(values[0]))
return classes

def read_db_stats(self, path):
    with open(path, "r") as file:
        lines = file.readlines()
    colors = lines[0].strip().split()
    fronts = lines[1].strip().split()
    radios = lines[2].strip().split()
    return (colors, fronts, radios)

def get_image_info(self, image_path: str):
    height, width, _ = cv2.imread(image_path).shape
    return height, width

def save_classes(self):
    if self.info is None:
        return
    for index, var in enumerate(self.info["chk_vars"]):
        if str(self.info["chk"][index]["state"]) != "disabled":
            self.info["classes"][index] = var.get() + 2

    original_txt = os.path.join(self.label_folder, self.info["label_file"])
    new_txt = os.path.join(self.new_label_folder, self.info["label_file"])
    stats_txt = os.path.join(self.dataset_info_folder, self.info["label_file"])

    with open(original_txt, "r") as file:
        lines = file.readlines()
    with open(new_txt, "w+") as file:
        for n_line, line in enumerate(lines):
            if len(self.info["classes"]) > n_line:
                clase = str(self.info["classes"][n_line])
                line = clase + line[1:]
                file.write(line)

    # Save stats
    with open(stats_txt, "w+") as f:
        lines = []
        f.write(" ".join(color.get() for color in self.info["clr_vars"])) # type: ignore
        f.write(
            "\n" + " ".join(str(front.get()) for front in self.info["front_vars"])
        ) # type: ignore
        f.write(
            "\n" +
            " ".join(str(radio.get()) for radio in self.info["radio_vars"])
            + "\n"
        ) # type: ignore

    self.imgs_info[self.id] = self.info

def right(self, *ignore):
    self.save_classes()
    self.id += 1
    if self.id >= self.n_pics:
        self.id = 0
    self.update_disp()

def left(self, *ignore):
    self.save_classes()

```

```

        self.id -= 1
        if self.id < 0:
            self.id = self.n_pics - 1
        self.update_disp()

    def new_n_image(self, *ignore):
        self.save_classes()
        self.id = int(self.idpp_entry.get()) - 1
        if self.id >= self.n_pics:
            self.id = self.n_pics - 1
        elif self.id < 0:
            self.id = 0
        self.update_disp()

    def update_disp(self):
        for w in self.frame.winfo_children():
            w.grid_forget()
            w.pack_forget()
        file = self.file_names[self.id]
        image_file = file + ".jpg"
        image_path = os.path.join(self.image_folder, image_file)
        label_file = file + ".txt"
        img_h, img_w = self.get_image_info(image_path)
        new_img_w = int(img_w * NEW_IMG_H / img_h)
        img = ImageTk.PhotoImage(Image.open(image_path).resize((new_img_w, NEW_IMG_H)))
        if os.path.isfile(os.path.join(self.new_label_folder, label_file)):
            classes = self.read_yolo_labels(
                os.path.join(self.new_label_folder, label_file)
            )
        else:
            classes = self.read_yolo_labels(os.path.join(self.label_folder, label_file))

        if os.path.isfile(os.path.join(self.dataset_info_folder, label_file)):
            colors, fronts, distances = self.read_db_stats(
                os.path.join(self.dataset_info_folder, label_file)
            )
        else:
            colors = ["wh"] * len(classes)
            fronts = [-1] * len(classes)
            distances = [-1] * len(classes)

        chkbuttons_vars = []
        chkbuttons = []
        clr_entries_vars = []
        clr_entries = []
        front_chks_vars = []
        front_chks = []
        radios = []
        radios_vars = []
        for index, (clase, color, front_saved, distance) in enumerate(
            zip(classes, colors, fronts, distances)
        ):
            if clase in [2, 3]:
                value = clase - 2
                state = tk.NORMAL
            else:
                value = 0
                state = tk.DISABLED
            clr_var = tk.StringVar(value=color)
            front_var = tk.IntVar(value=int(front_saved))
            radio_var = tk.IntVar(value=int(distance))
            var = tk.IntVar(value=value)

            chkbutton = ttk.Checkbutton(

```

```

        master=self.inputs_frame, text=index, variable=var, state=state
    )
    chkbuttons_vars.append(var)
    chkbuttons.append(chkbutton)

    clr_entry = ttk.Entry(
        master=self.inputs_frame, textvariable=clr_var, width=10, state=state
    )
    clr_entries_vars.append(clr_var)
    clr_entries.append(clr_entry)

    front_chk = ttk.Checkbutton(
        master=self.inputs_frame, variable=front_var, state=state
    )
    front_chks_vars.append(front_var)
    front_chks.append(front_chk)

    radio_c = ttk.Radiobutton(
        master=self.inputs_frame, variable=radio_var, state=state, value=0
    )

    radio_m = ttk.Radiobutton(
        master=self.inputs_frame, variable=radio_var, state=state, value=1
    )

    radio_l = ttk.Radiobutton(
        master=self.inputs_frame, variable=radio_var, state=state, value=2
    )
    radio_ml = ttk.Radiobutton(
        master=self.inputs_frame, variable=radio_var, state=state, value=3
    )
    radios_vars.append(radio_var)
    radios.append((radio_c, radio_m, radio_l, radio_ml))

self.imgs_info[self.id] = { # type: ignore
    "img_file": image_file,
    "img_path": image_path,
    "label_file": label_file,
    "img": img,
    "classes": classes,
    "chk_vars": chkbuttons_vars,
    "chk": chkbuttons,
    "clr_vars": clr_entries_vars,
    "clr": clr_entries,
    "front_vars": front_chks_vars,
    "front": front_chks,
    "radio_vars": radios_vars,
    "radios": radios,
}
}

self.info = self.imgs_info[self.id]

ttk.Label(master=self.frame, text="Imagen número: ").grid(
    row=0, column=0, sticky="e", pady=10
)

self.idpp_entry_var.set(str(self.id + 1))
self.idpp_entry.grid(row=0, column=1, pady=10, sticky="w")
self.idpp_entry.bind("<Return>", self.new_n_image)

img_name = tk.Text(
    master=self.frame,
    height=1,
    width=50,

```

```

        borderwidth=0,
        background=self.cget("background"),
    )
    img_name.tag_configure("center_text", justify="center")
    img_name.insert(tk.END, self.info["label_file"] if self.info else "")
    img_name.config(state=tk.DISABLED)
    img_name.tag_add("center_text", "1.0", tk.END)
    img_name.grid(row=1, column=0, columnspan=2, sticky="n")

label = ttk.Label(self.frame, image=self.info["img"] if self.info else "")
label.image = self.info["img"] if self.info else "" # type: ignore
label.grid(sticky="n", column=0, row=2)

unlabelled_img = ImageTk.PhotoImage(
    Image.open(os.path.join(self.bb_image_folder, image_file)).resize(
        (new_img_w, NEW_IMG_H)
    )
)
label = ttk.Label(self.frame, image=unlabelled_img) # type: ignore
label.image = unlabelled_img # type: ignore
label.grid(sticky="n", column=1, row=2)

for w in self.inputs_frame.winfo_children():
    w.grid_forget()
    w.pack_forget()

ttk.Label(master=self.inputs_frame, text="Es pesada:").grid(
    row=0, column=0, sticky="n", padx=0
)
col = 1
for chk in self.info["chk"]:
    # type: ignore
    chk.grid(row=0, column=col, sticky="n", padx=10)
    col += 1

ttk.Label(master=self.inputs_frame, text="Color:").grid(
    row=1, column=0, sticky="n", padx=0
)
col = 1
for clr in self.info["clr"]:
    # type: ignore
    clr.grid(row=1, column=col, sticky="n", padx=10)
    col += 1

ttk.Label(master=self.inputs_frame, text="de frente?").grid(
    row=2, column=0, sticky="n", padx=0
)
col = 1
for front in self.info["front"]:
    # type: ignore
    front.grid(row=2, column=col, sticky="n", padx=5)
    col += 1
self.inputs_frame.grid(sticky="n", pady=5, columnspan=2)

ttk.Label(master=self.inputs_frame, text="Cerca").grid(
    row=3, column=0, sticky="n", padx=0
)
ttk.Label(master=self.inputs_frame, text="Media").grid(
    row=4, column=0, sticky="n", padx=0
)
ttk.Label(master=self.inputs_frame, text="Lejos").grid(
    row=5, column=0, sticky="n", padx=0
)
ttk.Label(master=self.inputs_frame, text="Muy lejos").grid(
    row=6, column=0, sticky="n", padx=0
)
col = 1

```

```

for radios_info in self.info["radios"]: # type: ignore
    r_c, r_m, r_l, r_ml = radios_info
    r_c.grid(row=3, column=col, sticky="n", padx=5)
    r_m.grid(row=4, column=col, sticky="n", padx=5)
    r_l.grid(row=5, column=col, sticky="n", padx=5)
    r_ml.grid(row=6, column=col, sticky="n", padx=5)
    col += 1
self.inputs_frame.grid(sticky="n", pady=5, columnspan=2)

buttons_frame = ttk.Frame(master=self.frame)
ttk.Button(master=buttons_frame, text="<", command=self.left).grid(
    row=0, column=0, sticky="s"
)
ttk.Button(master=buttons_frame, text=">", command=self.right).grid(
    row=0, column=1, sticky="s"
)
buttons_frame.grid(sticky="s", columnspan=2)

if __name__ == "__main__":
    general_dir = r"TFM\archive\for_relabelling"
    bb_image_folder = general_dir + r"\unlabeled_images"
    image_folder = general_dir + r"\images"
    label_folder = general_dir + r"\labels"
    new_label_folder = general_dir + r"\new_labels"
    dataset_info_folder = general_dir + r"\dataset_info"
    root = Display(
        image_folder,
        label_folder,
        new_label_folder,
        bb_image_folder,
        dataset_info_folder,
    )
    root.mainloop()

```

ANEXO B: SCRIPT LS2YOLO.PY.

```
import json

# Specify the class list
classes = ['M', 'C', 'FL', 'FP', 'A', 'CL', 'CP', 'CPA']

# Load the json file
with open('jsonmin.json', 'r') as f:
    data = json.load(f)

# Loop through each image in the data
for image in data:
    # Create a txt file with the same name as the image
    path = 'FURGONETAS_LIGERAS_Y_PESADAS_LABELS\\'
    filename = image['image'].split('/')[-1].split('.')[0].split('-')[1] + '.txt'

    with open(path + filename, 'w') as file:
        # Loop through each label in the image
        print(image['id'])
        if 'label' in image:
            for label in image['label']:
                for label in image['label']:
                    WIDTH = int(label['original_width'])
                    HEIGHT = int(label['original_height'])
                    # Get the class index
                    class_index = classes.index(label['rectanglelabels'][0])
                    x=float(label['x'])/100 * WIDTH
                    y=float(label['y'])/100 * HEIGHT
                    w=float(label['width'])/100 * WIDTH
                    h=float(label['height'])/100 * HEIGHT

                    x = (x +w/2)/WIDTH
                    w = (w)/WIDTH
                    y = (y +h/2)/HEIGHT
                    h = (h)/HEIGHT

                    print(class_index,x,y,w,h)
                    # Write the coordinates in the yolov5 format, scaling the percentages to be between 0 and 1
                    file.write(f'{class_index} {x} {y} {w} {h}\n')
```

ANEXO C: SCRIPT PARA AUMENTO DE IMÁGENES.

```
import albumentations as A
import cv2
import os
import json
import random

# Helper function to read YOLO labels
def read_yolo_labels(label_path):
    with open(label_path, "r") as file:
        labels = []
        for line in file:
            cls, x_center, y_center, width, height = map(float, line.strip().split())
            labels.append([x_center, y_center, width, height, cls])
    return labels

# Helper function to write YOLO labels
def write_yolo_labels(label_path, labels):
    with open(label_path, "w") as file:
        for label in labels:
            cls = int(label[-1])
            x_center, y_center, width, height = label[:-1]
            file.write(f"{cls} {x_center} {y_center} {width} {height}\n")

# Helper function to read dataset info
def read_dataset_info(info_path):
    with open(info_path, "r") as file:
        info = []
        lines = file.readlines()
        colors = lines[0].strip().split()
        fronts = [elem for elem in lines[1].strip().split()]
        dists = [elem for elem in lines[2].strip().split()]
        for index, color in enumerate(colors):
            info.append([color, fronts[index], dists[index]])
    return info

# Helper function to write dataset info
def write_dataset_info(info_path, info):
    colors = " ".join([info_vehicle[0] for info_vehicle in info])
    fronts = " ".join([info_vehicle[1] for info_vehicle in info])
    dists = " ".join([info_vehicle[2] for info_vehicle in info])

    with open(info_path, "w") as file:
        file.write(colors + "\n")
        file.write(fronts + "\n")
        file.write(dists + "\n")

transform_scale_rot = A.Compose(
    [
        A.HorizontalFlip(),
        A.ShiftScaleRotate(
            scale_limit=0, rotate_limit=10, shift_limit_y=0.005, shift_limit_x=0.3, p=1
        ),
        A.RandomResizedCrop(
```

```

        height=608, width=608, scale=(0.6, 0.9), ratio=(0.99, 1.01), p=1.0
    ),
    A.RandomBrightnessContrast(p=0.2),
    A.HueSaturationValue(hue_shift_limit=10, sat_shift_limit=20, p=0.1),
    A.RandomShadow(shadow_roi=(0.5, 0.5, 1.0, 1.0), p=0.1),
],
bbox_params=A.BboxParams(
    format="yolo", label_fields=["category_ids", "dataset_info"]
),
)
)

transform_color = A.Compose(
[
    A.HorizontalFlip(),
    A.RandomBrightnessContrast(brightness_limit=0.4, p=0.8),
    A.HueSaturationValue(
        hue_shift_limit=30, sat_shift_limit=30, val_shift_limit=30, p=0.8
    ),
    A.RandomRain(rain_type="drizzle", p=0.3),
    A.RandomShadow(shadow_roi=(0.5, 0.5, 1.0, 1.0), p=0.1),
    A.CoarseDropout(max_holes=4, max_height=32, max_width=32, p=0.1),
],
bbox_params=A.BboxParams(
    format="yolo", label_fields=["category_ids", "dataset_info"]
),
)
transform_rain = A.Compose(
[
    A.HorizontalFlip(),
    A.RandomBrightnessContrast(p=0.3),
    A.RandomRain(
        slant_lower=-10,
        slant_upper=10,
        drop_length=15,
        drop_width=2,
        drop_color=(170, 170, 170),
        blur_value=3,
        brightness_coefficient=0.8,
        p=0.7,
    ),
],
bbox_params=A.BboxParams(
    format="yolo", label_fields=["category_ids", "dataset_info"]
),
)
)

transform_weather = A.Compose(
[
    A.HorizontalFlip(),
    A.RandomBrightnessContrast(p=0.3),
    A.HueSaturationValue(hue_shift_limit=10, sat_shift_limit=30, p=0.3),
    A.RandomFog(fog_coef_lower=0.1, fog_coef_upper=0.3, alpha_coef=0.1, p=0.2),
    A.RandomSunFlare(p=0.2),
    A.RandomShadow(
        shadow_roi=(0.5, 0.5, 1.0, 1.0),
        num_shadows_lower=1,
        num_shadows_upper=2,
        shadow_dimension=3,
        p=0.4,
    ),
    A.CoarseDropout(max_holes=3, max_height=42, max_width=32, p=0.1),
],
bbox_params=A.BboxParams(

```

```

        format="yolo", label_fields=["category_ids", "dataset_info"]
    ),
)

transform_dropout = A.Compose(
[
    A.ShiftScaleRotate(shift_limit=0.1, scale_limit=0, rotate_limit=5, p=0.3),
    A.RandomBrightnessContrast(p=0.3),
    A.HueSaturationValue(hue_shift_limit=10, sat_shift_limit=30, p=0.3),
    A.RandomRain(
        slant_lower=-15,
        slant_upper=15,
        drop_length=20,
        drop_width=3,
        drop_color=(150, 150, 150),
        blur_value=5,
        brightness_coefficient=0.7,
        p=0.2,
    ),
    A.RandomShadow(shadow_roi=(0.5, 0.5, 1.0, 1.0), p=0.2),
    A.CoarseDropout(max_holes=8, max_height=42, max_width=32, p=1),
],
bbox_params=A.BboxParams(
    format="yolo", label_fields=["category_ids", "dataset_info"]
),
)
)

FILTERED_IMAGES_DIR = r"/home/maria/TFM/data/datasets/filtered_DATASET_v2/images"
FILTERED_LABELS_DIR = r"/home/maria/TFM/data/datasets/filtered_DATASET_v2/labels"
FILTERED_INFO_DIR = r"/home/maria/TFM/data/datasets/filtered_DATASET_v2/dataset_info"

AUG_IMAGES_DIR = r"/home/maria/TFM/data/datasets/filtered_DATASET_v2/aug_images"
AUG_LABELS_DIR = r"/home/maria/TFM/data/datasets/filtered_DATASET_v2/aug_labels"
AUG_INFO_DIR = r"/home/maria/TFM/data/datasets/filtered_DATASET_v2/aug_dataset_info"
SCRIPTS_DIR = r"/home/maria/TFM/scripts"
# set seed for random generated numbers
random.seed(444)

# load json data

with open(
    os.path.join(SCRIPTS_DIR, "augment_prob.json"),
    "r",
    encoding="utf-8",
) as f:
    augment_probs = json.load(f)

with open(
    os.path.join(SCRIPTS_DIR, "similar_images.json"),
    "r",
    encoding="utf-8",
) as f:
    similar_images = json.load(f)

errored: list[str] = []
for file_name in os.listdir(FILTERED_IMAGES_DIR):
    if file_name.split("_")[1] in [
        "augScaleRot",
        "augColor",
        "augWeather",
        "augDropout",
        "augRain",
    ]:
        continue

```

```

if file_name.endswith(".jpg"):
    try:
        image = cv2.imread(os.path.join(FILTERED_IMAGES_DIR, file_name))
        label_path = os.path.join(
            FILTERED_LABELS_DIR, file_name.replace(".jpg", ".txt"))
    )
    labels = read_yolo_labels(label_path)
    info_path = os.path.join(
        FILTERED_INFO_DIR, file_name.replace(".jpg", ".txt"))
    )
    info = read_dataset_info(info_path)
    # Separate bounding boxes and category ids
    bboxes = [label[:4] for label in labels]
    category_ids = [label[4] for label in labels]

    if random.random() <= augment_probs[file_name[:-4]]:
        if file_name in similar_images:
            to_augment = similar_images[file_name].pop(
                random.randrange(len(similar_images[file_name])))
            )
        else:
            to_augment = file_name
        image_to_augment = cv2.imread(
            os.path.join(FILTERED_IMAGES_DIR, to_augment))
        )
        print(to_augment)
        augmented = transform_scale_rot(
            image=image_to_augment,
            bboxes=bboxes,
            category_ids=category_ids,
            dataset_info=info,
            )
        new_name = file_name[:5] + "_augScaleRot_" + file_name[5:]
        cv2.imwrite(os.path.join(AUG_IMAGES_DIR, new_name), augmented["image"])
        new_label_name = new_name.replace(".jpg", ".txt")
        new_label_path = os.path.join(AUG_LABELS_DIR, new_label_name)
        write_yolo_labels(
            new_label_path,
            [
                list(bbox) + [category_id]
                for bbox, category_id in zip(
                    augmented["bboxes"], augmented["category_ids"])
                )
            ],
            )
        info_path = os.path.join(AUG_INFO_DIR, new_label_name)
        write_dataset_info(info_path, augmented["dataset_info"])

    if random.random() <= augment_probs[file_name[:-4]]:
        if file_name in similar_images:
            to_augment = similar_images[file_name].pop(
                random.randrange(len(similar_images[file_name])))
            )
        else:
            to_augment = file_name
        image_to_augment = cv2.imread(
            os.path.join(FILTERED_IMAGES_DIR, to_augment))
        )
        augmented = transform_color(
            image=image_to_augment,
            bboxes=bboxes,
            category_ids=category_ids,
            dataset_info=info,
            )

```

```

new_name = file_name[:5] + "_augColor_" + file_name[5:]
cv2.imwrite(os.path.join(AUG_IMAGES_DIR, new_name), augmented["image"])
new_label_name = new_name.replace(".jpg", ".txt")
new_label_path = os.path.join(AUG_LABELS_DIR, new_label_name)
write_yolo_labels(
    new_label_path,
    [
        list(bbox) + [category_id]
        for bbox, category_id in zip(
            augmented["bboxes"], augmented["category_ids"]
        )
    ],
)
info_path = os.path.join(AUG_INFO_DIR, new_label_name)
write_dataset_info(info_path, augmented["dataset_info"])

if random.random() <= augment_probs[file_name[:-4]]:
    if file_name in similar_images:
        to_augment = similar_images[file_name].pop(
            random.randrange(len(similar_images[file_name]))
        )
    else:
        to_augment = file_name
    image_to_augment = cv2.imread(
        os.path.join(FILTERED_IMAGES_DIR, to_augment)
    )
    augmented = transform_rain(
        image=image_to_augment,
        bboxes=bboxes,
        category_ids=category_ids,
        dataset_info=info,
    )
    new_name = file_name[:5] + "_augRain_" + file_name[5:]
    cv2.imwrite(os.path.join(AUG_IMAGES_DIR, new_name), augmented["image"])
    new_label_name = new_name.replace(".jpg", ".txt")
    new_label_path = os.path.join(AUG_LABELS_DIR, new_label_name)
    write_yolo_labels(
        new_label_path,
        [
            list(bbox) + [category_id]
            for bbox, category_id in zip(
                augmented["bboxes"], augmented["category_ids"]
            )
        ],
    )
    info_path = os.path.join(AUG_INFO_DIR, new_label_name)
    write_dataset_info(info_path, augmented["dataset_info"])

if random.random() <= augment_probs[file_name[:-4]]:
    if file_name in similar_images:
        to_augment = similar_images[file_name].pop(
            random.randrange(len(similar_images[file_name]))
        )
    else:
        to_augment = file_name
    image_to_augment = cv2.imread(
        os.path.join(FILTERED_IMAGES_DIR, to_augment)
    )
    augmented = transform_weather(
        image=image_to_augment,
        bboxes=bboxes,
        category_ids=category_ids,
        dataset_info=info,
    )

```

```

        new_name = file_name[:5] + "_augWeather_" + file_name[5:]
        cv2.imwrite(os.path.join(AUG_IMAGES_DIR, new_name), augmented["image"])
        new_label_name = new_name.replace(".jpg", ".txt")
        new_label_path = os.path.join(AUG_LABELS_DIR, new_label_name)
        write_yolo_labels(
            new_label_path,
            [
                list(bbox) + [category_id]
                for bbox, category_id in zip(
                    augmented["bboxes"], augmented["category_ids"]
                )
            ],
        )
        info_path = os.path.join(AUG_INFO_DIR, new_label_name)
        write_dataset_info(info_path, augmented["dataset_info"])

    if random.random() <= augment_probs[file_name[:-4]]:
        if file_name in similar_images:
            to_augment = similar_images[file_name].pop(
                random.randrange(len(similar_images[file_name]))
            )
        else:
            to_augment = file_name
        image_to_augment = cv2.imread(
            os.path.join(FILTERED_IMAGES_DIR, to_augment)
        )
        augmented = transform_dropout(
            image=image_to_augment,
            bboxes=bboxes,
            category_ids=category_ids,
            dataset_info=info,
        )
        new_name = file_name[:5] + "_augDropout_" + file_name[5:]
        cv2.imwrite(os.path.join(AUG_IMAGES_DIR, new_name), augmented["image"])
        new_label_name = new_name.replace(".jpg", ".txt")
        new_label_path = os.path.join(AUG_LABELS_DIR, new_label_name)
        write_yolo_labels(
            new_label_path,
            [
                list(bbox) + [category_id]
                for bbox, category_id in zip(
                    augmented["bboxes"], augmented["category_ids"]
                )
            ],
        )
        info_path = os.path.join(AUG_INFO_DIR, new_label_name)
        write_dataset_info(info_path, augmented["dataset_info"])
    except Exception as e:
        print(e.__class__, ": ", e)
        errored.append(file_name)

# save the updated similar images json
with open(
    os.path.join(SCRIPTS_DIR, "similar_images_after_augmenting.json"),
    "w",
    encoding="utf-8",
) as f:
    json.dump(similar_images, f, indent=4)
with open(
    os.path.join(SCRIPTS_DIR, "errored.txt"),
    "w",
    encoding="utf-8",
) as f:
    for er in errored:
        f.write(er + "\n")

```

```
f.write(er + "\n")
```

ANEXO D PARÁMETROS POR DEFECTO DEL ENTRENAMIENTO CON ULTRALYTICS YOLO

```
# Ultralytics YOLO, AGPL-3.0 license
# Default training settings and hyperparameters for medium-augmentation COCO training

task: detect # (str) YOLO task, i.e. detect, segment, classify, pose
mode: train # (str) YOLO mode, i.e. train, val, predict, export, track, benchmark

# Train settings -----
model: # (str, optional) path to model file, i.e. yolov8n.pt, yolov8n.yaml
data: # (str, optional) path to data file, i.e. coco8.yaml
epochs: 100 # (int) number of epochs to train for
time: # (float, optional) number of hours to train for, overrides epochs if supplied
patience: 100 # (int) epochs to wait for no observable improvement for early stopping of training
batch: 16 # (int) number of images per batch (-1 for AutoBatch)
imgsz: 640 # (int | list) input images size as int for train and val modes, or list[h,w] for predict and export modes
save: True # (bool) save train checkpoints and predict results
save_period: -1 # (int) Save checkpoint every x epochs (disabled if < 1)
cache: False # (bool) True/ram, disk or False. Use cache for data loading
device: # (int | str | list, optional) device to run on, i.e. cuda:0 or device=0,1,2,3 or device=cpu
workers: 8 # (int) number of worker threads for data loading (per RANK if DDP)
project: # (str, optional) project name
name: # (str, optional) experiment name, results saved to 'project/name' directory
exist_ok: False # (bool) whether to overwrite existing experiment
pretrained: True # (bool | str) whether to use a pretrained model (bool) or a model to load weights from (str)
optimizer: auto # (str) optimizer to use, choices=[SGD, Adam, Adamax, AdamW, NAdam, RAdam, RMSProp, auto]
verbose: True # (bool) whether to print verbose output
seed: 0 # (int) random seed for reproducibility
deterministic: True # (bool) whether to enable deterministic mode
single_cls: False # (bool) train multi-class data as single-class
rect: False # (bool) rectangular training if mode='train' or rectangular validation if mode='val'
cos_lr: False # (bool) use cosine learning rate scheduler
close_mosaic: 10 # (int) disable mosaic augmentation for final epochs (0 to disable)
resume: False # (bool) resume training from last checkpoint
amp: True # (bool) Automatic Mixed Precision (AMP) training, choices=[True, False], True runs AMP check
fraction: 1.0 # (float) dataset fraction to train on (default is 1.0, all images in train set)
profile: False # (bool) profile ONNX and TensorRT speeds during training for loggers
freeze: None # (int | list, optional) freeze first n layers, or freeze list of layer indices during training
multi_scale: False # (bool) Whether to use multiscale during training
# Segmentation
overlap_mask: True # (bool) masks should overlap during training (segment train only)
mask_ratio: 4 # (int) mask downsample ratio (segment train only)
# Classification
dropout: 0.0 # (float) use dropout regularization (classify train only)

# Val/Test settings -----
val: True # (bool) validate/test during training
split: val # (str) dataset split to use for validation, i.e. 'val', 'test' or 'train'
save_json: False # (bool) save results to JSON file
save_hybrid: False # (bool) save hybrid version of labels (labels + additional predictions)
conf: # (float, optional) object confidence threshold for detection (default 0.25 predict, 0.001 val)
iou: 0.7 # (float) intersection over union (IoU) threshold for NMS
max_det: 300 # (int) maximum number of detections per image
half: False # (bool) use half precision (FP16)
dnn: False # (bool) use OpenCV DNN for ONNX inference
plots: True # (bool) save plots and images during train/val

# Predict settings -----
```

```

source: # (str, optional) source directory for images or videos
vid_stride: 1 # (int) video frame-rate stride
stream_buffer: False # (bool) buffer all streaming frames (True) or return the most recent frame (False)
visualize: False # (bool) visualize model features
augment: False # (bool) apply image augmentation to prediction sources
agnostic_nms: False # (bool) class-agnostic NMS
classes: # (int | list[int], optional) filter results by class, i.e. classes=0, or classes=[0,2,3]
retina_masks: False # (bool) use high-resolution segmentation masks
embed: # (list[int], optional) return feature vectors/embeddings from given layers

# Visualize settings -----
show: False # (bool) show predicted images and videos if environment allows
save_frames: False # (bool) save predicted individual video frames
save_txt: False # (bool) save results as .txt file
save_conf: False # (bool) save results with confidence scores
save_crop: False # (bool) save cropped images with results
show_labels: True # (bool) show prediction labels, i.e. 'person'
show_conf: True # (bool) show prediction confidence, i.e. '0.99'
show_boxes: True # (bool) show prediction boxes
line_width: # (int, optional) line width of the bounding boxes. Scaled to image size if None.

# Export settings -----
format: torchscript # (str) format to export to, choices at https://docs.ultralytics.com/modes/export/#export-formats
keras: False # (bool) use Keras
optimize: False # (bool) TorchScript: optimize for mobile
int8: False # (bool) CoreML/TF INT8 quantization
dynamic: False # (bool) ONNX/TF/TensorRT: dynamic axes
simplify: False # (bool) ONNX: simplify model using 'onnxslim'
opset: # (int, optional) ONNX: opset version
workspace: 4 # (int) TensorRT: workspace size (GB)
nms: False # (bool) CoreML: add NMS

# Hyperparameters -----
lr0: 0.01 # (float) initial learning rate (i.e. SGD=1E-2, Adam=1E-3)
lrf: 0.01 # (float) final learning rate (lr0 * lrf)
momentum: 0.937 # (float) SGD momentum/Adam beta1
weight_decay: 0.0005 # (float) optimizer weight decay 5e-4
warmup_epochs: 3.0 # (float) warmup epochs (fractions ok)
warmup_momentum: 0.8 # (float) warmup initial momentum
warmup_bias_lr: 0.1 # (float) warmup initial bias lr
box: 7.5 # (float) box loss gain
cls: 0.5 # (float) cls loss gain (scale with pixels)
dfl: 1.5 # (float) dfl loss gain
pose: 12.0 # (float) pose loss gain
kobj: 1.0 # (float) keypoint obj loss gain
label_smoothing: 0.0 # (float) label smoothing (fraction)
nbs: 64 # (int) nominal batch size
hsv_h: 0.015 # (float) image HSV-Hue augmentation (fraction)
hsv_s: 0.7 # (float) image HSV-Saturation augmentation (fraction)
hsv_v: 0.4 # (float) image HSV-Value augmentation (fraction)
degrees: 0.0 # (float) image rotation (+/- deg)
translate: 0.1 # (float) image translation (+/- fraction)
scale: 0.5 # (float) image scale (+/- gain)
shear: 0.0 # (float) image shear (+/- deg)
perspective: 0.0 # (float) image perspective (+/- fraction), range 0-0.001
flipud: 0.0 # (float) image flip up-down (probability)
fliplr: 0.5 # (float) image flip left-right (probability)
bgr: 0.0 # (float) image channel BGR (probability)
mosaic: 1.0 # (float) image mosaic (probability)
mixup: 0.0 # (float) image mixup (probability)
copy_paste: 0.0 # (float) segment copy-paste (probability)
auto_augment: randaugment # (str) auto augmentation policy for classification (randaugment, autoaugment, augmix)
erasing: 0.4 # (float) probability of random erasing during classification training (0-0.9), 0 means no erasing, must be 1
crop_fraction: 1.0 # (float) image crop fraction for classification (0.1-1), 1.0 means no crop, must be greater than 0.

```

```
# Custom config.yaml -----
cfg: # (str, optional) for overriding defaults.yaml

# Tracker settings -----
tracker: botsort.yaml # (str) tracker type, choices=[botsort.yaml, bytetrack.yaml]
```