UNIVERSIDAD MIGUEL HERNÁNDEZ DE ELCHE

ESCUELA POLITÉCNICA SUPERIOR DE ELCHE

GRADO EN INGENIERÍA INFORMÁTICA EN TECNOLOGÍAS DE LA INFORMACIÓN

"DESARROLLO DE UNA APLICACIÓN DE DETECCIÓN EN TIEMPO REAL DE PERSONAS EN IMÁGENES UTILIZANDO DEEP LEARNING"

TRABAJO FIN DE GRADO

Julio - 2025

AUTOR: Manuel Delgado García
DIRECTOR: Jesús Javier Rodríguez Sala

RESUMEN

Este trabajo se centrará en desarrollar, mediante Deep Learning, un modelo que permita identificar en imágenes a personas para aplicar esta detección en sistemas en tiempo real.

Para ello, se hará un estudio sobre las herramientas existentes en el mercado actual que permitan integrar de manera rápida modelos de detección de objetos. Por otro lado, se plantearán diferentes alternativas para lograr este cometido. Las principales opciones serán entrenar un modelo completo desde cero o hacer uso de uno pre entrenado cuya configuración sea altamente positiva para el cometido del proyecto.

Sobre la opción escogida, se realizará una investigación más exhaustiva para encaminar el desarrollo de una aplicación para dispositivos que permita la inferencia sobre imágenes tomadas en tiempo real. Se tomarán en cuenta rasgos como el rendimiento en entornos con recursos limitados y métricas de precisión.

Una vez realizado el estudio, se escogerá la herramienta más óptima para el desarrollo final de una aplicación para dispositivos móviles. Esta tendrá una funcionalidad sencilla donde se permita visualizar en la pantalla las detecciones realizadas por el modelo generado.

AGRADECIMIENTOS

A mis familiares y amigos por apoyarme durante el desarrollo, dándome sus puntos de vista y ayudándome en los momentos en los que me quedaba atascado. También a mi tutor del TFG, por los consejos e ideas proporcionadas para mejorar el resultado de este.

ÍNDICE GENERAL

ÍNDICE DE TABLAS

ÍNDICE DE FIGURAS

Capítulo 1
Introducción

1.1.- MACHINE LEARNING

El Machine Learning es un término que, por su definición, está estrechamente ligado con la evolución de la computación. Uno de los hitos más importantes en relación a esto fué conseguido gracias al matemático Alan Turing en el año 1936. Su investigación demostró cómo se podía dotar a una máquina con la capacidad de resolver problemas matemáticos de una manera automatizada [1]. Esta rama de la Inteligencia artificial trata de dotar a un sistema informático con la capacidad de aprender automáticamente a partir de unos datos de entrada obtenidos previamente. El principal objetivo de este es descubrir patrones con estos datos para diferentes objetivos, como realizar predicciones o tomar decisiones.

Hay numerosas aplicaciones para esta modalidad de la IA. Tenemos el reconocimiento en imágenes, que trata de identificar los objetos que se encuentran dentro de una imagen; el Procesamiento de Lenguaje Natural (PLN), usado para que la máquina interprete un texto

introducido; o el modelaje predictivo, una técnica de estadística en el que a partir de datos histórico se intenta prever un resultado esperado [2]. La ventaja de estas actividades es que pueden implementarse en numerosos sectores, como por ejemplo en medicina, industria, finanzas, etc.

Estas son solo algunas de las aplicaciones del Machine Learning, y suelen centrarse en tareas poco complejas. Para actividades que presentan una dificultad más elevada, es posible que necesitemos aumentar la complejidad del modelo que deseamos crear para tener resultados satisfactorios. Estos tipos de problema requieren de mayor capacidad de cómputo y volumen de datos para funcionar correctamente.

1.2.- DEEP LEARNING

Alrededor del 2005 ocurrió el boom del Big Data, la gente empezó a darse cuenta del gran volumen de datos que se generaba con el tráfico de Internet. Plataformas como Facebook, Youtube, Twitter, entre otras, evidenciaban este hecho. Para permitir tratar esta cantidad de información, surgieron diferentes frameworks como Hadoop MapReduce o Apache Shark [3] que facilitan el análisis de esta. Todo esto desencadenó la aparición de una versión mejorada del Machine Learning, el Deep Learning

El Deep Learning es una evolución dentro del Machine Learning que utiliza métodos más avanzados para llevar a cabo tareas similares. La principal mejora está en el uso de redes neuronales profundas, estructuras inspiradas en el funcionamiento de las redes neuronales biológicas. Su diferencia respecto a las redes neuronales usadas para Machine Learning es que son más grandes y usan un mayor número de capas. Esta estructura del Deep Learning permite a los modelos aprender de manera más autónoma y eficiente, extrayendo patrones complejos a partir de grandes volúmenes de datos.

Este sistema facilita completar tareas complejas de una forma más accesible y sencilla. Por ejemplo, en el ámbito de la medicina se puede entrenar para interpretar ecocardiogramas [4], o en el de la agricultura para hacer un control de plagas y de la salud de los cultivos [5]. Ambas funcionalidades emplean modelos entrenados para reconocer diferentes objetos dentro de imágenes.

1.3.- COMPUTER VISION

Se denomina Computer Vision a la disciplina de la Inteligencia Artificial que permite a máquinas simular el funcionamiento del ojo humano para realizar diversas tareas. Esta se centra en analizar imágenes y videos de la misma forma que lo hacemos nosotros y su

finalidad es extraer información relevante que permita realizar una toma de decisiones automatizada.

Este ámbito tiene un amplio recorrido de desarrollo y, con los avances del Deep Learning, los algoritmos que se emplean han mejorado considerablemente la precisión de sus análisis. Así, se ha permitido el desarrollo de aplicaciones para apoyar a profesionales que desempeñan trabajos donde la precisión es crucial, como en el caso de la medicina [6].

1.4.- MODELOS DE RECONOCIMIENTO DE OBJETOS

El reconocimiento de objetos es una de las aplicaciones más relevantes que encontramos dentro del Machine Learning y Deep Learning. Aplicando técnicas de Computer Vision, permite a los sistemas identificar y clasificar objetos en imágenes o videos. Su rango de aplicación es amplio, teniendo un gran impacto en diversos sectores, como la seguridad, la automoción o, como se ha expuesto en puntos anteriores, en la salud y la agricultura.

Los modelos de reconocimientos de objetos funcionan siguiendo los siguientes pasos: primero se obtienen las imágenes que se desean tratar y se procesan, después se extrae la información relevante que se encuentre en ellas y, por último, se localizan los objetos presentes. En los últimos años, las redes neuronales han mejorado significativamente su precisión y velocidad, lo que permite implementar de manera satisfactoria esta herramienta en entornos en tiempo real.

Existen numerosos algoritmos utilizados para desempeñar esta labor, como YOLO (You Only Look Once), SSD (Single Shot Multibox Detector) o SIFT (Scale-Invariant Feature Transform) [7]. Cada uno de estos presentan características diferentes en relación a velocidad y precisión, siendo esto determinante para su aplicación en función de la tarea que se desea realizar. Estos modelos están en continua evolución y han permitido avances como el reconocimiento facial en dispositivos móviles, identificación de anomalías en imágenes médicas o agrícolas y el uso de cámaras inteligentes en seguridad.

A pesar de presentar sus beneficios, estos modelos también tienen limitaciones. Por ejemplo, la necesidad de tener grandes volúmenes de datos para realizar su entrenamiento o el consumo de una alta cantidad de recursos computacionales. También existen variables que pueden no ser tomadas en cuenta a la hora de realizar el entrenamiento, como la iluminación o la posición de los objetos [8]. Aun con esto, la aparición de nuevas arquitecturas y técnicas consiguen mejorar el desempeño de los modelos y ampliar sus aplicaciones en el mundo real.

1.5.- MODELOS PRE-ENTRENADOS

Se puede reducir el tiempo de desarrollo y el costo computacional de crear un modelo empleando modelos pre-entrenados. Estos han sido desarrollados previamente con grandes conjuntos de datos y están preparados para ser reutilizados. Debido a esta característica, se suelen ver resultados más satisfactorios que entrenando modelos de cero con datos limitados.

Para emplear estos modelos, se necesita un periodo de tiempo en el que ajustarlos con el conjunto de datos con el que se desea desarrollar el software. A esto se le llama Transfer Learning y es usado por investigadores y desarrolladores para que adapten la tecnología y funcione en tareas específicas.

Esta solución tiene ventajas frente a entrenar un modelo completamente de cero [9] pero, a su vez, hay que tener en cuenta algunos factores que pueden llevar a que el desarrollo no funcione como se plantea. Sobre todo se debe ajustar correctamente el nuevo dataset para que no ocurra ningún problema a la hora de hacer uso del nuevo modelo.

1.6.- JUSTIFICACIÓN DEL PROYECTO

Desde hace años me ha interesado el campo del Computer Vision. Me sorprendía la capacidad que tenían algunos sistemas para poder detectar y delimitar distintos objetos que se encuentran dentro de una imágen de una manera precisa. Añadido a esto, en los últimos años se ha visto un notable aumento en el interés y la relevancia hacia la Inteligencia Artificial, lo que ha hecho incrementar aún más mi curiosidad en este campo.

Esto no ha sido lo único, también me encuentro trabajando en una empresa cuya especialidad es la de automatización de procesos y uso de distintos modelos de inteligencia artificial. Por ello también tengo interés en mejorar en el ámbito laboral. Esto ha sido razón suficiente para decidirme centrar mi TFG en el desarrollo de un modelo propio. Para aumentar la complejidad de mi proyecto, también me he propuesto llevarlo a una aplicación para dispositivos móviles.

1.7.- OBJETIVOS

En este proyecto se pretende crear una aplicación para dispositivos móviles que implemente un modelo de detección de objetos para identificar personas en tiempo real.

Objetivos generales del proyecto:

- Investigar modelos pre-entrenados
- Obtener datos para entrenamiento
- Entrenar un modelo y validarlo
- Implementar un modelo en una aplicación móvil

Objetivos personales:

- Aumentar conocimientos en IA, Machine Learning y Deep Learning
- Comprender y hacer uso del lenguaje de programación Python
- Aplicar conocimientos de desarrollo de aplicaciones móviles

1.8.- LÍMITES DEL PROYECTO

Este proyecto se completa creando una aplicación para móviles que implemente el modelo desarrollado. En cuanto a las plataformas compatibles con dicha aplicación, únicamente se centrará en el desarrollo para Android, sin tomar en cuenta iOS.

Capítulo 2 Antecedentes y estado de la cuestión

2.1.- HERRAMIENTAS DISPONIBLES EN EL MERCADO

Actualmente, además del uso de librerías y frameworks de código abierto, existen numerosas opciones en el mercado para implementar sistemas de detección de objetos en proyectos. Estas plataformas proporcionan APIs listas para usar que permiten realizar tareas de Computer Vision sin la necesidad de entrenar modelos.

Estas soluciones en la nube ofrecen funciones como detección de objetos, reconocimiento facial, etiquetado automático y detección de textos (OCR), permitiendo escalabilidad tanto de capacidad de procesamiento como en el uso de grandes volúmenes de datos. Sin embargo, presentan algunas desventajas como problemas de privacidad, latencias y costos asociados al uso de sus APIs.

2.1.1.- GOOGLE CLOUD

La empresa de Google proporciona en su sistema Cloud numerosas funciones centradas en el reconocimiento de objetos. Encontramos por ejemplo Cloud Vision [10], una herramienta cuyas características son generales. Con ella podemos realizar numerosas acciones, como etiquetar imágenes, detección de caras y puntos de referencia o funcionalidades OCR para el tratamiento de ficheros.

En cuanto a la implementación de esta solución, esta API funciona mediante un sistema de unidades. Al tener una cuenta creada en la web *Cloud,* se disponen de 1000 unidades gratuitas al principio de cada mes. En el caso de necesitar usar un número mayor, el precio difiere según las acciones que se realicen, aunque en su mayoría son unos 1.50$ por cada 1000 unidades extras empleadas.

Cloud Vision es la opción con las funcionalidades más básicas dentro del sector del Computer Vision, pero Google ofrece otras herramientas más centradas en otras funcionalidades. Está, por ejemplo, Visual Inspector, cuyas características sirven para agilizar el control de calidad para los sectores de industria y fabricación. También ofrece la posibilidad de trabajar con vídeos, permitiendo analizarlos o realizar acciones como la detección y seguimiento de objetos.

Adicionalmente, Google también ofrece la posibilidad de entrenar un modelo propio con características personalizadas con Vertex AI [11]. Este servicio permite entrenar modelos para muchas finalidades, entre las que encontramos la creación de modelos para el reconocimiento de objetos. La principal ventaja que ofrece es que no es necesario adquirir previamente conocimientos sobre Machine Learning para generar un modelo.

2.1.2.- AMAZON REKOGNITION

Amazon también dispone de un servicio relacionado con el reconocimiento de objetos. La empresa ofrece una API que permite ser integrada para realizar actividades como el reconocimiento de imágenes o el análisis de videos empleando Machine Learning. Esta herramienta se llama Amazon Rekognition [12] y el coste de su uso varía tanto en función de los tipos de datos que se van a utilizar (imágenes o videos), como de qué funciones se quieren utilizar para el desarrollo.

Para trabajar con imágenes, es posible usar aquellas que estén almacenadas en el dispositivo local pero, en el caso de los videos, se requieren otros métodos [13]. En este caso, existen dos posibilidades: la primera, utilizar vídeos en streaming aplicando una herramienta llamada Amazon Kinesis Video Streams para procesarlos correctamente; y la

segunda, almacenarlos en Amazon Simple Storage Service (Amazon S3), propia de Amazon, y usarlos desde ahí.

2.1.3.- AZURE AI VISION

Microsoft proporciona entre sus servicios en la nube una API que permite la detección de objetos y análisis de imágenes, Azure AI Vision [14]. Permite a los desarrolladores una integración fácil y sus capacidades son avanzadas. Tiene la capacidad de procesar imágenes y videos a través de su API REST o usando SDKs en lenguajes como Python, C# o JavaScript, entre otros. Al igual que las herramientas de Google y Amazon, se tiene la posibilidad de utilizar modelos pre entrenados desarrollados por Microsoft, eliminando la necesidad de tener conocimientos avanzados en Machine Learning.

Azure AI Vision también cuenta con integración directa con otros servicios de Azure, como Azure Cognitive Search, Azure Functions o Power Platform, facilitando la creación de flujos de trabajo automatizados. Adicionalmente, existe Vision Studio, un portal gráfico donde se puede experimentar con las capacidades de la plataforma sin escribir líneas de código.

En cuanto al precio para poder utilizarse, este varía en función de la región desde donde se solicite el uso de la API. También existe la posibilidad de no disponer del servicio que se desee. Esto es así porque debido a diferentes causas, como políticas de cumplimiento o requisitos regulatorios, puede ser necesario realizar una solicitud previa para comprobar la disponibilidad en la región.

2.1.4.- CLARIFAI

Clarifai es una plataforma especializada en inteligencia artificial y su principal enfoque es el Computer Vision [15]. Ofrece una amplia gama de funcionalidades orientadas al análisis de imágenes y vídeos haciendo uso de aprendizaje automático. Los modelos que se pueden usar con Clarifai son desde preeentranados hasta otros personalizados. Proporciona una interfaz visual y un entorno en la nube. También está diseñado para integrar los modelos en aplicaciones haciendo uso de APIs y SDKs en varios lenguajes de programación. Por otro lado, no existe la opción de personalizar los recursos hardware que se le destinan al entrenamiento.

Entre sus utilidades, se encuentran la detección y clasificación de objetos, reconocimiento facial, etiquetado automático de imágenes e incluso análisis de contenido explícito. A su vez, dispone de soporte para análisis de video, incluyendo funciones como detección de eventos y seguimiento de objetos en movimiento. La herramienta dispone de un

marketplace donde la comunidad añade modelos creados por ellos mismos para hacer uso de ellos en entornos diferentes.

Esta herramienta dispone de un plan gratuito con recursos limitados, útil para desarrollos académicos y para la realización de pruebas. Permite 1000 llamadas API mensuales y modelos pre entrenados, además de soporte de la comunidad. Para desarrollos más elaborados existen otras ofertas fichas de 30$ y 300$ mensuales en ambos casos. No obstante, existe la posibilidad de crear un plan personalizado.

2.2.- PUNTOS RELEVANTES PARA ENTRENAR UN MODELO

Como se ha dicho anteriormente, un modelo de detección de objetos puede obtenerse de varias maneras (usando un modelo con entrenamiento previo o creando uno nuevo desde cero). No obstante, sin importar de qué forma se decida plantear el desarrollo, existen aspectos clave comunes a considerar al crear un modelo confiable y de alto rendimiento.

La calidad y eficiencia del resultado depende de múltiples factores que deben ser cuidadosamente tratados durante el desarrollo. Desde la selección del conjunto de datos hasta la elección del algoritmo y la configuración de los parámetros de entrenamiento, cada decisión influye directamente en la precisión y el rendimiento del modelo. También es importante disponer de un buen entorno de desarrollo. Un ejemplo sería hacer uso de gráficas Nvidia, ya que disponen de la tecnología CUDA, plataforma de computación paralela desarrollada por la misma empresa y que tiene la capacidad de manejar una gran cantidad de operaciones en paralelo.

Hacer uso de datos adecuados y representativos es esencial para garantizar que el modelo funcione de manera satisfactoria al ser usado en diferentes entornos y escenarios, Asimismo, la elección del algoritmo más apropiado según los requisitos del proyecto permite optimizar un equilibrio entre precisión y velocidad de cómputo. Acciones como el preprocesamiento de datos, el ajuste de parámetros y la correcta evaluación del modelo son claves para la obtención de un sistema eficiente y fiable para la detección de objetos.

2.3.- ELECCIÓN DEL TIPO DE MODELO

En el desarrollo de sistemas de detección de objetos, la elección entre utilizar modelos pre entrenados o desarrollar modelos personalizados es una decisión crítica que afecta directamente al rendimiento, tiempo de desarrollo.

Los modelos personalizados requieren un proceso de entrenamiento desde cero, lo que implica la recopilación de un conjunto de datos adecuados, la selección de una arquitectura neuronal y la configuración de hiperparámetros. Aunque este enfoque permite una mayor adaptación a tareas específicas, también demanda un tiempo elevado, más recursos computacionales y experiencia en el diseño de modelos de aprendizaje automático.

Por otra parte, los modelos pre entrenados han sido entrenados con grandes volúmenes de datos y están listos para ser utilizados en tareas concretas. Estos presentan ventajas significativas frente a los modelos personalizados, pues el tiempo necesario para su entrenamiento es más reducido y necesita un menor volúmen de datos etiquetados. Estas características permiten que sean seleccionados para realizar proyectos con recursos limitados o plazos ajustados [16].

2.3.1.- REDES NEURONALES

Un modelo emplea para su funcionamiento Redes Neuronales Artificiales (RNA), estructuras inspiradas en el cerebro humano para procesar la información que se recibe. Al igual que este, está formada por neuronas las cuales procesan la información recibida y toman diferentes valores en función de las decisiones tomadas por el modelo. Cada neurona artificial tiene un valor específico y pueden ser binarias (proporcionan valores fijos de {-1, 1} o {0, 1}) o reales (proporcionan valores entre intervalos de [-1, 1] o [0, 1]). Estas están distribuidas en diferentes capas [17] y se pueden agrupar en 3 grupos:

- Capa de Entrada: Se encarga de recibir los datos iniciales del problema. Cada neurona perteneciente a esta capa representa una característica del conjunto de datos de entrada.

- Capas ocultas: Procesan la información de manera interna mediante neuronas interconectadas. El número de capas y neuronas por capas de esta parte de la red neuronal influye de manera directa en la capacidad de aprendizaje y generalización del modelo.

- Capa de Salida: Proporciona la respuesta final del modelo (una predicción, una clasificación, etc., dependiendo de la tarea a realizar).

En referencia al Deep Learning, existen diferentes arquitecturas de redes neuronales dependiendo del propósito que tengan. Para modelos que realizan procesamiento de lenguaje se emplean RNN (redes neuronales recurrentes) o para generación de imágenes, audio o texto, GAN (redes generativas adversariales). En el caso del tratamiento de imágenes o videos, se emplea la arquitectura CNN (redes neuronales convolucionales)

[18]. La particularidad de esta es que funciona procesando la información en forma de cuadrículas y extrayendo la información necesaria para realizar los trabajos de detección y clasificación

2.3.2.- TRANSFER LEARNING

El transfer learning o aprendizaje por transferencia es una técnica en la que, aprovechando el conocimiento ya adquirido por un modelo en una tarea previa, se aplica a otra que tenga relación con la tarea original del modelo pre entrenado [19]. Este enfoque es útil al reutilizar características ya aprendidas y comprobadas en el modelo, mejorando la precisión y eficiencia del resultado que se obtenga con el nuevo desarrollo.

2.3.3.- FINE-TUNING

Para orientar el modelo hacia la resolución de una tarea concreta, se realiza un entrenamiento adicional usando un conjunto de datos más pequeño y específico. Este proceso es conocido como Fine-Tuning [20] (ajuste fino), y es una técnica dentro del Transfer Learning. Este método permite que el modelo refine sus parámetros para adaptarse mejor a particularidades de la nueva tarea, mejorando su rendimiento sin necesidad de un entrenamiento desde cero. Es común que durante este proceso, se congelen capas para que no sean modificadas.

2.4.- DATOS DE ENTRENAMIENTO

Para obtener un buen modelo final, es necesario recurrir a datos de entrenamiento de buena calidad y variados. En el caso de modelos de detección de objetos, es importante disponer de imágenes que presenten diferencias de iluminación, orientación, etc. Obtener los datos de entrenamiento de manera manual puede resultar altamente tedioso, por lo que lo recomendable es recurrir a conjuntos de datos en la web, como ImageNET o COCO Dataset [21]. Hay varias técnicas de preprocesamiento que se deben aplicar antes de comenzar el entrenamiento con el objetivo de asegurar la homogeneidad y representatividad de los datos que se van a utilizar.

2.4.1.- NORMALIZACIÓN

La normalización se basa en ajustar los valores de los píxeles de las imágenes a un rango uniforme, como [-1, 1] o [0, 1]. Su principal objetivo es estabilizar el proceso de

aprendizaje, permitiendo que los pesos de la red neuronal se ajusten de manera más eficiente, evitando que los valores extremos dominen el cálculo de gradientes. Normalmente los píxeles comprenden valores en un rango [0, 255] y al normalizarse sus valores se previene que el modelo esté sesgado por rangos o intensidades de colores [22].

2.4.2.- REDIMENSIONADO

Un requisito común es que las imágenes deben tener las mismas dimensiones para poder procesarlas correctamente [23]. Redimensionarlas asegura que estas son compatibles con la arquitectura del modelo y reduce la complejidad computacional del entrenamiento. También es necesario mantener la proporción de aspecto de las imágenes para evitar que sean deformadas al redimensionarlas.

2.4.3.- AUMENTO DE DATOS

Puede darse la situación de que no se disponga de suficiente variedad de datos para el entrenamiento. Este problema suele ser común dada la complejidad que conlleva a veces obtener imágenes etiquetadas. Para solventar dicho problema se puede proceder a realizar un aumento de datos o data augmentation. Esta práctica consiste en generar nuevos datos a partir de los que ya se disponen haciendo uso de distintas transformaciones [24].

Las ventajas del aumento de datos son múltiples. Se incrementa el tamaño efectivo del conjunto de entrenamiento sin necesidad de recolectar datos de nuevo. A su vez, mejora la capacidad de generalización del modelo al disponer de una mayor variedad de ejemplos. Esto ayuda a reducir el sobreajuste, una situación en la que el modelo se ajusta demasiado a un tipo de dato concreto, afectando negativamente a su capacidad de predicción). Además, permite simular condiciones más realistas que pueden no estar representadas correctamente.

Las transformaciones más comunes incluyen operaciones geométricas, como rotaciones, volteo horizontal o escalado de imágenes; y modificaciones fotométricas, como cambios en la saturación, brillo o contraste de los colores. Por otro lado, existen técnicas más complejas como la mezcla de imágenes o el corte aleatorio de regiones.

2.5.- FRAMEWORKS Y LIBRERÍAS

Existe una amplia variedad de frameworks y librerías disponibles en línea para realizar el desarrollo de un modelo de Deep Learning. Estas herramientas proporcionan

implementaciones optimizadas de algoritmos complejos, lo que facilita la gestión de grandes volúmenes de datos.

Estos entornos de desarrollo permiten entrenar modelos personalizados haciendo uso de arquitecturas predefinidas. También suelen posibilitar tareas comunes, como el preprocesamiento de datos, la visualización de resultados o la exportación del modelo generado para hacer uso de él en distintas plataformas. Respecto a esto, existen versiones orientadas al despliegue en dispositivos móviles, centrándose en el rendimiento en entornos con recursos limitados.

2.5.1.- TENSORFLOW

TensorFlow [25] es uno de los frameworks de Deep Learning más utilizados. Desarrollado por Google, tiene una amplia comunidad que facilita la búsqueda de información y modelos preentrenados con esta tecnología. A su vez, es soportado por los principales lenguajes de programación mediante llamadas API, siendo la más completa la de Python.

Algunos ejemplos de modelos que se encuentran ya desarrollados son COCO-SSD [26], una versión para TensorFlow que utiliza el tipo de modelo de detección de objetos Single Shot Detector, entrenado con el conjunto de datos COCO; y un modelo de detección de rostros simples llamado Blazeface [27], que puede detectar múltiples rostros a partir de 6 puntos clave, siendo estos los ojos, nariz, centro de la boca y las orejas.

En cuanto a las versiones disponibles para entornos con recursos limitados, existe una llamada TensorFlow Lite. Esta versión está optimizada en cuanto a latencia, privacidad, conectividad, tamaño y consumo de energía, tiene una alta compatibilidad y, además, proporciona un alto rendimiento [28]. Un estudio sobre el uso de TinyML (aplicaciones de Machine Learning en dispositivos de recursos limitados) para la plataforma web StreamNet demuestra cómo esta herramienta acelera la inferencia local para microcontroladores sin aumentar el uso de memoria, facilitando el uso de IA en dispositivos de baja potencia [29].

2.5.2.- PYTORCH

La empresa Meta, antes conocida como Facebook, ofrece a su vez un framework desarrollado por su departamento Meta AI, llamado PyTorch [30]. Esta herramienta se usa principalmente con el lenguaje de Python junto a la librería Torch, una biblioteca de Machine Learning de código abierto que se emplea para la creación de redes neuronales profundas. Aun siendo Python el lenguaje principal, existe una librería llamada LibTorch

con la misma capacidad de aprendizaje que PyTorch, pero para desarrollar en un entorno de C++.

Para tareas de detección y segmentación de objetos, existe Detectron2 [31], biblioteca de código abierto también desarrollada por Meta y basada en PyTorch. Esta biblioteca es la evolución de Detectron, herramienta escrita en Python y creada mediante el framework Caffe2. Detecton2 está escrita también en lenguaje Python y soporta múltiples arquitecturas, como Faster R-CNN o Mask R-CNN [32]. Es muy utilizada para la investigación y la creación de proyectos industriales de vanguardia.

Para el desarrollo en dispositivos de recursos limitados, PyTorch dispone de una solución llamada ExecuTorch [33]. Está diseñada para ser extremadamente ligera y eficiente. También es compatible con una amplia variedad de plataformas de cómputo, como móviles o microcontroladores, con la ventaja de permitir a los desarrolladores usar el entorno de desarrollo de PyTorch para el desarrollo del modelo.

2.5.3.- ULTRALYTICS YOLO

YOLO es una familia de modelos centrados en la detección de objetos con numerosas versiones destinadas a diferentes propósitos [34]. El procesamiento que realiza esta tecnología permite que se detecten directamente las posiciones de los objetos en las imágenes únicamente al incluirlas dentro de la red neuronal. Por ello la velocidad de las soluciones que se generan usando este software es elevada, lo que es una gran ventaja. En sus primeras versiones, uno de sus principales problemas era la precisión en la detección de los objetos, dado que se procesaba la imagen al completo para ello. Esto se ha ido solventando conforme han sido creadas nuevas versiones.

La última versión disponible hasta el momento es YOLO12 [35], la cual presenta capacidades mejoradas para extracción de características, alta flexibilidad en cuanto a entorno de implementación y una optimización en cuanto a eficiencia y velocidad, entre otras.

En caso de querer desarrollar un modelo con la tecnología de YOLO, disponemos de UltralyticsYOLO [36]. Esta librería está especialmente preparada para la detección de objetos y la segmentación de imágenes en tiempo real. Una desventaja es que su uso está restringido únicamente para el lenguaje de programación Python y sistemas que tengan compatibilidad con CUDA.

Para dispositivos móviles, existe una aplicación llamada Ultralytics HUB, la cual permite ejecutar los modelos YOLO directamente en dispositivos Android. No obstante, también

existe la posibilidad de entrenar modelos adaptados concretamente para casos específicos, como entornos con una velocidad de computo limitada.

2.6.- HIPERPARÁMETROS

Durante el entrenamiento, es necesario ir eligiendo y ajustando unos valores llamados hiperparámetros. Estos son determinantes en el rendimiento final del modelo y son variables externas que no se aprenden directamente a partir de los datos de entrenamiento. Aun así condicionan el proceso de aprendizaje y la capacidad del modelo resultante para generalizar correctamente los datos. Existen valores recomendados que varían en función de múltiples factores, como el algoritmo seleccionado o las características de los datos de entrenamiento, pero lo ideal para un desarrollo satisfactorio es actualizar estos valores conforme se va realizando el entrenamiento [37].

2.6.1.- TASA DE APRENDIZAJE

La tasa de aprendizaje (learning rate) es uno de los parámetros más importante a la hora de realizar el entrenamiento. Mide cuánto se actualizan los pesos que contiene el modelo con cada iteración que se realice de entrenamiento. Al disponer de un valor bajo, la velocidad de aprendizaje será lenta, pero creará una solución más precisa. En cambio, tener uno elevado agiliza el proceso pero también genera un funcionamiento más errático.

Modificar este parámetro durante el entrenamiento es altamente beneficioso. Existen métodos, como el Learning Rate Decay, que proporciona un desarrollo beneficioso para el entendimiento de patrones complejos sin demorarse demasiado en el tiempo [38].

2.6.2.- NÚMERO DE ÉPOCAS

El número de épocas (epoch), determina el número de veces que se tratará el conjunto de datos de manera completa en la red. Según el valor que se le de, es posible que ocurran problemas en el resultado final. Con un valor bajo, puede darse el caso de un subajuste del modelo, ocasionando que la precisión de las predicciones sea baja. Por contra, un valor elevado del número de épocas ocasionará un sobreajuste, produciendo un mal rendimiento para la predicción de nuevos ejemplos (no pertenecientes al dataset de entrenamiento).

Existen técnicas que se pueden aplicar para seleccionar un valor óptimo según el punto en el que se encuentre el entrenamiento. Por ejemplo, la validación temprana (early stopping) tiene una sencilla aplicación [39]. Se trata de comprobar durante el proceso cuando deja de

mejorar el rendimiento del modelo. Al detectarlo, se vuelve al punto donde el modelo tenga una mejor puntuación.

2.6.3.- TAMAÑO DE LOTE

Para definir cuál es el tamaño de la muestra que se va a tratar antes de cambiar los pesos de las neuronas en una red neuronal, se utiliza el tamaño de lote o batch size. El valor de este parámetro influye en múltiples aspectos del entrenamiento. Al disponer de un tamaño de lote pequeño, se generan actualizaciones más frecuentes. Esto hace que el modelo mejore su capacidad para salir de mínimos locales, pero también genera mayor ruido en el gradiente (fluctuaciones aleatorias generadas en las estimaciones de gradientes). Por el contrario, con lotes más grandes se generan estimaciones más precisas, pero perjudicando la capacidad de generalización del modelo [40].

Además, este parámetro está relacionado con el tipo de GPU que se utilice para crear el modelo [41]. Al estar relacionado con la memoria, lo óptimo es utilizar valores en potencia de base 2 (2, 4, 8, 16, 32…), ya que se alinean con las arquitecturas de procesamiento paralelas.

2.6.4.- DROPOUT

El dropout es una técnica cuyo funcionamiento trata de prevenir el sobreajuste desactivando de manera aleatoria un porcentaje de las neuronas que componen la red neuronal durante el entrenamiento [42]. Esto permite un entrenamiento más robusto evitando que algunas neuronas tengan demasiado peso a la hora de hacer las predicciones. Tras completar el proceso de creación del modelo, este valor ya no realiza la misma acción, pero sí influye en el proceso de inferencia durante las predicciones, escalando los valores de las neuronas. Para su valor, generalmente se escoge un valor entre 0.2 y 0.5.

2.6.5.- FUNCIONES DE PÉRDIDA Y OPTIMIZADORES

Durante el entrenamiento, hay dos parámetros que ayudan a encaminar correctamente el desarrollo. Estas son las funciones de pérdida y los optimizadores. Ambos elementos están relacionados ya que trabajan conjuntamente durante el proceso.

Las funciones de pérdida (loss functions) cuantifican el error cometido por el modelo durante cada iteración que realiza. Esto lo hace comparando la predicción obtenida con la etiqueta real. Concretamente, dada la naturaleza de la detección de objetos, se suelen

utilizar pérdidas compuestas, que combinan la clasificación del objeto con la regresión de sus coordenadas espaciales en la imágen. Por ejemplo, se puede hacer la combinación de una función de pérdida de clasificación como categorical cross-entropy (usada para clasificación multiclase) y una de regresión como Smooth L1 Loss (compara números continuos, como las coordenadas del bounding box) o IU Loss (mide el solapamiento de bounding boxes entre el modelo y el real) [43].

Por otro lado, los optimizadores utilizan los valores calculados por el parámetro anterior para ajustar los pesos de la red neuronal. Uno de los más conocidos es Stochastic Gradient Descent (SGD), Este actualiza los parámetros utilizando pequeños subconjuntos de datos y puede complementarse con otras técnicas para mejorar su funcionamiento. También destaca Adam, que adapta automáticamente la tasa de aprendizaje de los parámetros, y RMSProp, que ajusta sus actualizaciones en función de la magnitud reciente de los gradientes [44].

La relación de estos dos parámetros puede ayudar en gran medida a crear un modelo altamente preciso. Por ello, es natural probar distintas combinaciones y probarlas al inicio del entrenamiento.

2.7.- EVALUACIÓN DEL MODELO

Una vez finalizado el proceso de entrenamiento del modelo, es necesario evaluarlo en función de distintas métricas. Esto permite visualizar con mayor facilidad su eficacia y adecuación al problema planteado. Esta evaluación se puede dividir en dos categorías: la precisión del modelo y la eficiencia computacional.

2.7.1.- MÉTRICAS DE PRECISIÓN

Las métricas de precisión cuantifican la calidad de las predicciones realizadas por el modelo. Para ello se mide su capacidad para detectar correctamente los objetos y localizarlos correctamente en el espacio. Entre las métricas más utilizadas para evaluar modelos de detección de objetos [45], se encuentran:

- IoU (Intersection over Union): Métrica que mide el grado de solapamiento entre las bounding boxes identificadas durante la predicción y las reales. Es muy importante para el cálculo de otros valores de precisión ya que, a partir de un umbral predefinido, se decide si una predicción realizada es correcta o no.

- AR (Average Recall): Evalúa la capacidad del modelo para encontrar de manera correcta todos los objetos que se encuentran presentes en la imagen. Para ello se realiza una media entre la tasa de acierto y el número máximo de detecciones.

- AP (Average Precision): Este parámetro combina la precisión (verdaderos positivos entre todas las predicciones positivas) y exhaustividad (verdaderos positivos entre predicciones reales) de las detecciones.

- mAP (mean Average Precision): Es la media de AP para todas las clases del conjunto de datos.

2.7.2.- MÉTRICAS DE RENDIMIENTO

Las métricas de rendimiento son importantes para determinar cómo de eficaz será el modelo generado en un entorno real. Para ello se hacen operaciones destinadas a obtener valores relacionados con el coste computacional o el tiempo necesario para realizar una predicción. En el contexto de la detección de objetos se pueden destacar:

- Tiempo de inferencia: Indica el tiempo que le lleva al modelo procesar una imagen, realizar los cálculos necesarios y devolver su predicción.

- FPS (Frames Per Second): Mide el número de imágenes por segundo que puede procesar el modelo.

- Uso de memoria GPU/CPU: Útil para estimar la carga que tendrá el modelo sobre el entorno en el que se use durante su ejecución.

2.8.- VALORACIÓN

Tras la investigación realizada para completar este capítulo, queda evidenciado que, para disponer de un modelo de detección de objetos, es necesario tener en cuenta múltiples factores. En función del objetivo que se tenga, es muy posible encontrar una solución ya desarrollada para ello haciendo una simple búsqueda de los requerimientos necesarios. A su vez, las principales empresas disponen de numerosas variaciones en cuanto a las capacidades de los modelos ofertados. También es remarcable la facilidad con la que se pueden implementar estas herramientas para su uso inmediato.

En el caso de no querer depender de nadie y desarrollar un modelo propio, es innegable la cantidad de recursos disponibles tanto en forma de frameworks, librerías, APIs, etc. Las

principales herramientas y las más usadas disponen tanto de una buena documentación oficial, como de grandes comunidades respaldando nuevos proyectos haciendo uso de estas.

Aun con todas las facilidades, es remarcable la importancia de un buen planteamiento para realizar el entrenamiento de un modelo. Hay que tener en cuenta múltiples factores que, si no se desarrollan correctamente, pueden llevar a la creación de un desarrollo poco óptimo. El conjunto de datos empleado tiene un gran peso en el resultado final. En el caso de la detección de objetos, emplear un etiquetado correcto y datos variados, con diferente iluminación y ángulos, entre otros parámetros.

Otro factor importante es hacer uso de diferentes combinaciones de hiperparámetros. Existen valores recomendados, pero no hay ninguna solución universal que permita un desarrollo que sea siempre eficaz. Por ello es muy recomendable la validación y comparación de los rendimientos y valores obtenidos tras el entrenamiento en cada caso.

Capítulo 3
Hipótesis de trabajo

3.1.- Lenguajes de programación

Para el desarrollo completo de los modelos que se van a ajustar y evaluar, junto a la aplicación para dispositivos Android, es necesario hacer uso de distintos lenguajes de programación. Las elecciones dependen de las diferentes posibilidades o requisitos que se planteen durante el desarrollo.

3.1.1.- Python

Para todo el trabajo relacionado con la preparación de los modelos de detección de objetos, se empleará el lenguaje de programación Python. Este dispone de numerosas librerías y es la principal herramienta utilizada para tareas relacionadas con Deep Learning [46]. Entre sus características destacables, ofrece numerosas posibilidades relacionadas con la

visualización de los resultados obtenidos durante el periodo de entrenamiento y validación de los modelos.

Cuenta con una amplia comunidad, lo que facilita en gran medida la búsqueda de recursos y documentación. Además, Python emplea una sintaxis clara y sencilla, lo que facilita su aprendizaje en el caso de no haber tenido un contacto previo con el lenguaje y disponer de conocimientos previos en la programación.

3.1.2.- Kotlin

Todas las funcionalidades necesarias para el correcto funcionamiento de la aplicación desarrollada serán programadas en Kotlin. Este lenguaje dispone de soporte nativo para la programación asíncrona. Esto lo logra mediante las coroutines [47], las cuales permiten realizar diferentes operaciones en hilos distintos. Así, evita que ocurra problemas inesperados, como la sobrecarga del dispositivo.

Otro punto a favor es que la sintaxis de este lenguaje es sencilla y altamente legible. Esto permite agilizar la revisión del código en caso de detectar un problema en el programa generado. También dispone de una comunidad altamente activa al ser el principal lenguaje para el desarrollo en dispositivos móviles. Con esto, la búsqueda de soluciones o ejemplos es una tarea más sencilla.

A su vez, Kotlin cuenta con numerosas librerías que permiten una fácil integración de herramientas de Machine Learning e interoperabilidad con el lenguaje Java, lo que permite el uso de librerías procedentes de este último de una manera directa en el código.

3.1.3.- XML

Para la interfaz gráfica se usará el lenguaje XML, estándar usado para el diseño de aplicaciones en Android. Su uso abarcará desde la interfaz básica, hasta el diseño de la visualización de las predicciones hechas durante la inferencia del modelo en tiempo real. Con esto se podrá comprobar en la aplicación la ubicación de cada persona presente en la imagen detectada por el modelo.

3.2.- Roboflow

La web de Roboflow [48] consiste en un portal online que permite múltiples funcionalidades relacionadas con el Deep Learning. Tiene una amplia comunidad de

usuarios activos que aportan diferentes desarrollos, como conjuntos de datos o modelos para determinados entornos.

Su principal ventaja es que facilita en gran medida algunos de los puntos críticos a la hora de crear modelos, tanto de detección de objetos como segmentación o clasificación de imágenes. Dentro de sus funcionalidades encontramos la creación de dataset y su anotación gráfica, permitiendo delimitar correctamente las cajas de colisión o las figuras de los objetos sobre los que se desee trabajar. También permite organizar de una manera sencilla estos conjuntos en los principales grupos necesarios para un correcto entendimiento (train, validation y test). Por otro lado, la página automatiza el proceso de data augmentation, preprocesamiento de imágenes y exportación en múltiples formatos, como Tensorflow, YOLO, Keras, etc.

En cuanto a este último punto, Roboflow dispone de una API [49] que permite la descarga de un conjunto de datos creado o guardado en una cuenta personal. A esta se le puede determinar el formato deseado para su uso en el entrenamiento de los modelos, agilizando el uso de los datos para el entrenamiento sin tener que realizar una descarga manual desde la web.

3.3.- Google Colab

Google Colab es una de las principales elecciones a la hora de realizar tareas relacionadas con el Deep Learning. Esta ofrece la posibilidad de usar de manera gratuita, con algunas limitaciones, de CPU, GPU y TPU en la nube.

El lenguaje usado en este entorno es Python y, como se ha comentado anteriormente, es uno de los lenguajes más potentes para trabajar con el aprendizaje automático. Además, Colab permite la ejecución de notebooks creados por otras personas, permitiendo realizar ejecuciones de prueba rápidamente. También tiene una distinción entre bloques de texto y código del notebook para tener una mejor organización del proyecto. Cada bloque de código dispone de una salida individual de registros de ejecución, facilitando la depuración.

3.4.- Modelos seleccionados

Para el desarrollo del modelo que se usará en la aplicación final, se usarán tres modelos pre-entrenados diferentes. Concretamente tres versiones diferentes de la familia Ultralytics YOLO. Esta decisión viene dada tras evaluar las dificultades técnicas identificadas al usar diferentes métodos de entrenamiento para los modelos. Esto permitirá hacer una

comparación más homogénea entre las versiones, sin dificultar el proceso de entrenamiento y validación, siendo este una fase intermedia dentro del desarrollo del proyecto.

En cuanto a la viabilidad de desarrollar sobre 3 modelos de la misma familia, existen artículos que demuestran las diferencias existentes entre versiones de YOLO [50]. Cada modelo presenta algunas diferencias que les aportan ventajas sobre otros para determinadas tareas. Adicionalmente, estos presentan versiones más ligeras que permiten un mejor rendimiento en dispositivos con recursos limitados.

Se han seleccionado concretamente los siguientes modelos:

- YOLO5nu: Una de las versiones más ligeras y con inferencia más rápida. Aun siendo menos reciente, estas propiedades lo hacen una opción válida actualmente para su uso. Adicionalmente, la versión 5 disponible cuenta con las mejoras introducidas en la octava versión del modelo.

- YOLO8n: Es un punto intermedio entre el rendimiento de los últimos modelos y los más antiguos. Esta versión introdujo un cambio sustancial en cuanto a su arquitectura, permitiendo una mejora en su rendimiento y velocidad.

- YOLO12n: Se trata de la última versión disponible dentro de la familia YOLO. Se ha elegido para poder estudiar las diferencias que existen entre esta versión y las anteriores, para analizar y conocer mejor el estado actual de la herramienta y sus capacidades.

3.5.- Dataset para el ajuste de los modelos

Se usará un único conjunto de datos para el ajuste de los modelos. Todas las imágenes y anotaciones usadas para el entrenamiento, validación y testeo de los modelos generados proceden del dataset COCO Dataset. Más concretamente se trata de una versión limitada del mismo, en la que solo se encuentran las imágenes que incluyen la etiqueta “person” [51].

Los modelos seleccionados han sido entrenados previamente con la totalidad de las imágenes presentes en COCO Dataset. Al utilizar un conjunto de datos más focalizado en la clase correspondiente a personas, permite ahorrar tiempo y recursos durante el ajuste de los mismos. Por otro lado, aplicando técnicas de “*data augmentation*” (o aumento de datos), aseguramos que el modelo cuente con nuevos casos para el entrenamiento, mejorando su capacidad de generalización. Este proceso trata de aplicar, como se ha comentado en el segundo capítulo, diferentes transformaciones a las imágenes, como

rotaciones, recortes, etc; para aumentar el volumen de datos y la variedad del conjunto. Es posible hacer este procedimiento directamente en la plataforma Roboflow, mencionada anteriormente, al crear una versión final y usable del dataset.

3.6.- Conversión del modelo

Tras seleccionar el modelo que se va a utilizar, es necesario hacer una conversión del modelo para permitir un correcto funcionamiento en dispositivos móviles. Mediante esta conversión se adapta el modelo para afrontar correctamente limitaciones como la limitada capacidad de procesamiento o memoria.

Para ello se usará el conjunto de herramientas ofrecidas por TensorFlow Lite (TFLite) [52]. Su objetivo es optimizar el modelo para su uso en dispositivos determinados y es una de las principales opciones para este fin. Es proporcionada por Google y altamente utilizada para desarrollo Android debido a su compatibilidad. La conversión realizada genera una nueva versión del modelo, con un formato más ligero y eficiente.

Capítulo 4
Metodología y resultados

4.1.- Planificación del proyecto

Para el proceso de creación de la aplicación de detección de personas en tiempo real, se ha estimado la planificación que se ilustra en la Fig. 4.1. Esta se ha dividido en distintas semanas, estableciendo el desarrollo en un total de cinco. Cada semana se ha enfocado en una sección diferente del desarrollo, las cuales explicaremos brevemente a continuación.

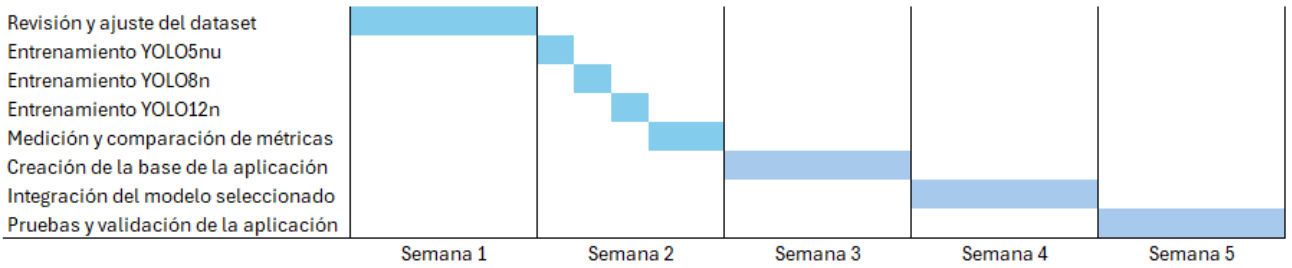

Fig. 4.1. Planificación del proyecto

Para la primera semana, se realizará la descarga del dataset y revisión de las imágenes para verificar las anotaciones incluidas en ellas. Para ello se verificarán las dimensiones de las cajas de colisión que delimitan a las personas en ellas.

La siguiente semana se entrenarán los tres modelos seleccionados. Con ello se obtendrán de cada uno el mejor modelo ajustado a la clase necesaria para realizar las detecciones. De ellos, se tomarán métricas de rendimiento y precisión para seleccionar el idóneo comparando los resultados obtenidos.

Tras esto, comenzará la codificación de la aplicación para dispositivos móviles. Para empezar con esta tarea, se creará la estructura principal sin tener en cuenta la implementación del modelo. Tras esto, se añadirá incluyendo las funcionalidades necesarias para el correcto funcionamiento del resultado final.

Para finalizar el proyecto, se realizarán unas pruebas del desarrollo. Con esto se hará una validación final para comprobar que se han abordado correctamente todos los puntos a incluir en la aplicación y que esta funcione correctamente.

4.2.- Obtención y validación del dataset

Como se ha comentado en el capítulo anterior, para el entrenamiento de todos los modelos se usará el mismo conjunto de imágenes, homogeneizando el conjunto de entrenamiento utilizado. Este contiene únicamente anotaciones referentes a personas, lo que es idóneo para ser usado en el proyecto. La volumetría de este conjunto es de algo menos de 5500 imágenes, lo que es un buen volúmen para casos donde solo se use una única clase durantes el entrenamiento y se esté trabajando con modelos pre entrenados con una mayor cantidad de imágenes [53].

Habiendo accedido al dataset mencionado (apartado 3.5), hemos comprobado que el propósito principal que tiene este dataset es entrenar modelos de segmentación de objetos. Es así por que las anotaciones que se pueden visualizar no son bounding boxes, si no delimitaciones más precisas de las siluetas de las personas que se pueden ver en las imágenes, como se muestra en la Fig. 4.2. Esto lo hace incompatible con el propósito del proyecto, por lo que primero se ha realizado la exportación de las imágenes y sus anotaciones en un formato cuyo formato de anotaciones sea de bounding boxes, concretamente en formato Tensorflow Object Detection.

Con el fichero obtenido, ya es posible volver a subir las imágenes con sus perspectivas anotaciones a un proyecto creado con una cuenta de Roboflow. Este proyecto, a su vez, se

encuentra dentro de un escritorio de trabajo donde es posible almacenar múltiples proyectos. Tras subir el fichero,se obtienen una anotaciones que ya corresponden a las usadas para tareas de detección de objetos. Aun con esto, existe un problema al hacer esta conversión forzada de las anotaciones.

Fig. 4.2. Imágen procedente del dataset original

Al tratar de comprobar las nuevas anotaciones, se puede encontrar un problema con estas. Ahora las cajas no corresponden correctamente con donde se visualizan a las personas en cada imágen. Por ello, es necesario hacer una revisión completa de las imágenes obtenidas. Con este proceso, se logra ajustar correctamente las anotaciones de las imágenes, así como eliminar aquellas que proporcionen información innecesaria para el entrenamiento. Un caso de ejemplo de imágen innecesaria para el proyecto sería la mostrada en la Fig. 4.3, donde se puede observar que se etiqueta a una persona la cual únicamente se le ven las manos.

Para estos casos, se eliminará el archivo. Esto se hace porque para que el modelo resultante generalice correctamente y realice predicciones precisas, hay que seguir unas pautas a la hora de seleccionar los datos de entrenamiento. Concretamente se han seguido las siguientes:

- Personas visibles completamente.
- Personas visibles pero que se encuentren tras objetos como vallas, cristales, etc.
- Personas a las que se les distinga de la mitad del torso hacia arriba.
- Personas a las que se les distinga del pecho hacia abajo.
- Personas que tienen visible la mitad derecha o izquierda de su cuerpo.

Siguiendo estas directrices, se logra obtener un buen conjunto, eliminando imágenes que contengan partes individuales del cuerpo, como una pierna o una mano.

Fig. 4.3. Imágen perjudicial para el entrenamiento

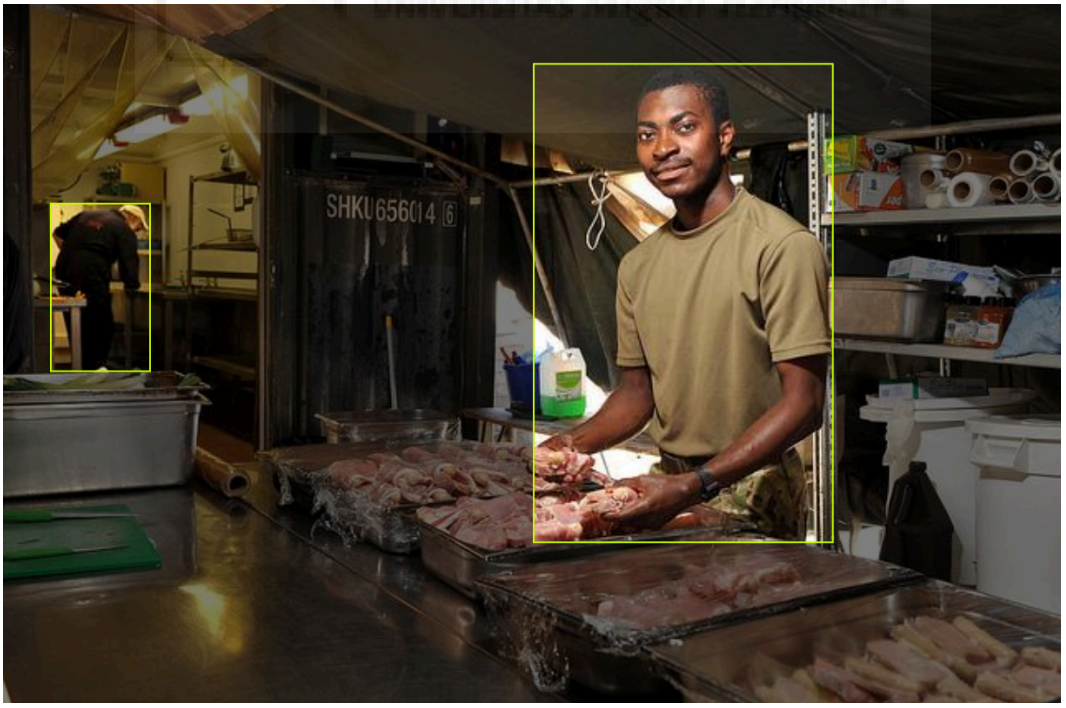

Fig. 4.4. Imágen ajustada apta para el entrenamiento

Tras realizar todo el proceso de revisión y ajuste de imágenes, se ha obtenido alrededor de 4900 imágenes usables para el proyecto. Para poder descargarlo y utilizarlo, es necesario

guardar una versión final del conjunto. Haciendo esto se nos permite personalizar cómo se va a generar el resultado final de nuestro conjunto de datos modificando diversos parámetros.

Primero, se selecciona qué cantidad de imágenes se quiere destinar a cada parte del dataset (entrenamiento, validación y test). Cada una de estas se utiliza para diferentes funcionalidades. El conjunto de entrenamiento será usado para realizar el entrenamiento durante; el de validación, para obtener métricas que nos permitan analizar el rendimiento del modelo; y el de test para almacenar algunas imágenes con el objetivo de hacer pruebas más individuales, como realizar el proceso de inferencia sobre una imagen concreta para comprobar cómo es la salida del modelo. Para este proyecto se ha realizado una división en proporción de 85/10/5 aproximadamente. De esta forma se da prioridad al conjunto de entrenamiento, el cual es el que tiene mayor importancia, mientras que el de validación contendría alrededor de 750 imágenes para obtener una buena valoración del rendimiento del modelo. Por último, el grupo de test permite disponer de datos que no han sido tratados por este para pruebas concretas más realistas.

Dejando claro este punto, se continúa realizando el preprocesamiento de las imágenes, lo que hace que todas presenten las mismas características. Aquí se añade padding o se redimensiona imágenes para lograr este objetivo, junto con orientación automática de las imágenes en caso de que exista alguna con rotación. Para el entrenamiento se ha decidido usar imágenes de 640x640, dado que es el estándar para los modelos de la familia YOLO.

Para finalizar la creación de la versión del conjunto de imágenes, Roboflow permite aplicar técnicas de data augmentation. Estas modificaciones permiten generar nuevos casos para que el modelo disponga de una mayor variedad de datos. La plataforma permite diversas técnicas para ello, muchas de ellas ya mencionadas previamente, y se puede indicar cuántas imágenes se crearán a partir de cada una de las ya existentes. Se ha decidido generar 2 imágenes por cada una, triplicando de esta manera la volumetría del conjunto de entrenamiento. Para ello, se han seleccionado los siguientes parámetros a modificar para generar las nuevas imágenes:

- Rotación: entre -15º y +15º
- Saturación: entre -25% y +25%
- Luminosidad: entre -15% y +15%

Estos cambios serán decididos aleatoriamente por la web mientras se crea la versión final del dataset. Haciendo estas modificaciones con valores menores al 50%, se evita que aparezcan nuevos casos poco realistas. Esto haría que empeorara la calidad del modelo generado. Con todo esto definido se logra obtener los datos que serán usados durante todo el proceso de entrenamiento y validación del modelo de detección de objetos.

4.3.- Entrenamiento de los modelos

El entrenamiento de los modelos se ha realizado en el entorno de Google Colab, que permite la ejecución en la nube de código Python mediante su sistema de notebooks. Los modelos seleccionados y comentados en previamente (apartado 3.4) pertenecen al mismo entorno, YOLO Ultralytics.

Este dispone de una librería para Python [54] que facilita enormemente el proceso de entrenamiento sobre modelos pre entrenados de este mismo ecosistema. Permite tanto realizar el entrenamiento, como la evaluación e inferencias individuales sobre imágenes concretas a través de funciones diferenciadas.

4.3.1.- Preparación del entorno

Antes de comenzar, es necesario hacer varias configuraciones iniciales para poder conseguir un rendimiento satisfactorio del entorno. Colab permite seleccionar los componentes que se utilizarán para ejecutar el código. Dado lo demandante que es el entrenamiento de modelos de detección de objetos, es necesario seleccionar una gráfica acorde a la tarea. Dentro del plan gratuito, está disponible el uso de gráficas T4 de manera limitada, pudiendo usarse durante alrededor de dos o tres horas cada día, aunque esto puede variar en caso de que la web presente un alto volúmen de usuarios activos en el momento

Adicionalmente, es necesario instalar mediante comandos PIP los paquetes de Ultralytics para los modelos y Roboflow para poder descargar el dataset generado previamente. También es altamente recomendable hacer una conexión con Google Drive para evitar perder los archivos que se generen, cosa que se hará para este proyecto.

```python
from google.colab import drive
drive.mount('/content/drive')

!pip install ultralytics
!pip install roboflow
```

Fig. 4.5. Código de preparación del entorno

Instaladas las dependencias, el siguiente paso es descargar el dataset para su uso. Es necesario usar una clave API para poder trabajar con la cuenta creada previamente y obtener los datos necesarios. Esta clave se encuentra en la propia web de Roboflow, en el

apartado de opciones. También es necesario indicar tanto el nombre del espacio de trabajo como el del proyecto, datos obtenidos en la URL de la web al acceder al proyecto. Para generalizar, el código mostrado en la Fig. 4.6. en esta sección, tiene modificaciones relacionadas con estos valores. También es necesario indicar cuál es la versión del dataset que se desea descargar, siendo el caso de la versión 1 para este proyecto.

Un punto importante referente a esta descarga es que cada modelo usa un formato diferente en cuanto a cómo se almacena y usa los datos. Por esta misma razón, en el código dónde se descarga una copia del conjunto de datos a usar (Fig. 4.6.), es necesario incluir qué formato tendrán estos ficheros. Para ello únicamente hay que definir la variable "*formato*" con el nombre de la versión del modelo que se va a entrenar en cada caso.

```python
from roboflow import Roboflow

formato = "yolov8"
varRoboFlow= Roboflow(api_key="clave-api")

proyecto = rf.workspace("nombre-workspace").project("nombre-proyecto")
dataset = project.version(1).download(formato)
```

Fig. 4.6. Código para descargar el dataset

4.3.2.- Proceso de entrenamiento

Una vez descargado el modelo, ya se puede comenzar con su entrenamiento. Al trabajar con la librería oficial de YOLO, no es necesario hacer una descarga previa de los modelos. Para usarlo, es posible invocarlos directamente en una variable indicando la versión deseada.

Antes de comenzar el entrenamiento, es necesario definir correctamente los hiperparámetros que se van a utilizar. Previamente se han comentado algunos de ellos en el capítulo 2. Antes de comenzar el entrenamiento final, se han realizado entrenamientos parciales con número de época bajo para verificar y seleccionar unos valores correctos. Tras este proceso se han seleccionado los valores que mejores resultados proporcionaban para evitar problemas mayores y se han mantenido constantes para todos los modelos.

Finalmente, se ha decidido realizar un entrenamiento de 50 épocas. El tamaño de las imágenes indicado es el mismo con el que se generó el dataset y el tamaño de lote se ha mantenido en 16 para obtener un equilibrio entre precisión y rendimiento durante el entrenamiento. Se ha utilizado una tasa de aprendizaje de 0.001 para evitar que los ajustes que se vayan realizando provoquen cualquier tipo de inestabilidad, aunque esto ralentice el

entrenamiento. También se han indicado 3 épocas de calentamiento para mejorar la estabilidad del entrenamiento. Por último, se utiliza una regularización mediante decaimiento de pesos de 0.0005 para evitar un sobreajuste del modelo al tratar con valores altos en los pesos junto con el optimizador SGD, cuyo uso es común para modelos de detección de objetos.

Junto con los valores mencionados, también se han indicado dos parámetros más a la función de entrenamiento. El valor de "*data*" corresponde al fichero contenido en el dataset que indica donde se encuentra cada conjunto de imágenes. Este se genera al descargar el dataset en cada uno de los formatos necesarios. La función localiza dentro de este fichero la ruta concreta donde se encuentra el conjunto de entrenamiento. Por otra parte, el valor de "*pretrained*" indica que el modelo va a usar los pesos con los que se ha entrenado previamente. Esto es así dado que el objetivo de este trabajo es ajustar un modelo pre entrenado, por lo que es necesario que dicho modelo cuente con estos pesos.

```python
from ultralytics import YOLO

modelo = YOLO("yolov8n.pt")
entrenamiento = modelo.train(
                    data="/content/TFG-1/data.yaml",
                    epochs=50,
                    patience=15,
                    imgsz=640,
                    batch=16,
                    lr0=0.001,
                    warmup_epochs=3,
                    weight_decay=0.0005,
                    optimizer="SGD",
                    pretrained=True)
```

Fig 4.7. Código de entrenamiento de los modelo

Respecto a la evolución que se ha observado durante los entrenamientos, se puede destacar que todos han visto una mejora significativa a través de cada época, mejorando métricas como su precisión o su recall. Una de las gráficas que permite ver de una manera sencilla es la matriz de confusión, que es generada al finalizar el entrenamiento ejecutando una época sobre el conjunto de validación. En ella se detalla: verdaderos positivos, falsos positivos, falsos negativos y verdaderos negativos. En el caso particular de YOLO8n (Fig. 4.8) se han realizado correctamente predicciones sobre un 80% del dataset de entrenamiento. Hay que remarcar que esta matriz se calcula sobre el número total de objetos etiquetados (en nuestro caso personas).

De los 3, únicamente el modelo YOLO12n ha parado su entrenamiento antes de llegar al final. Como se muestra en la Fig 4.9., a partir de la época número 35 del entrenamiento, el valor de la métrica mAP50-95 no ha logrado mejorarse en las siguientes épocas.

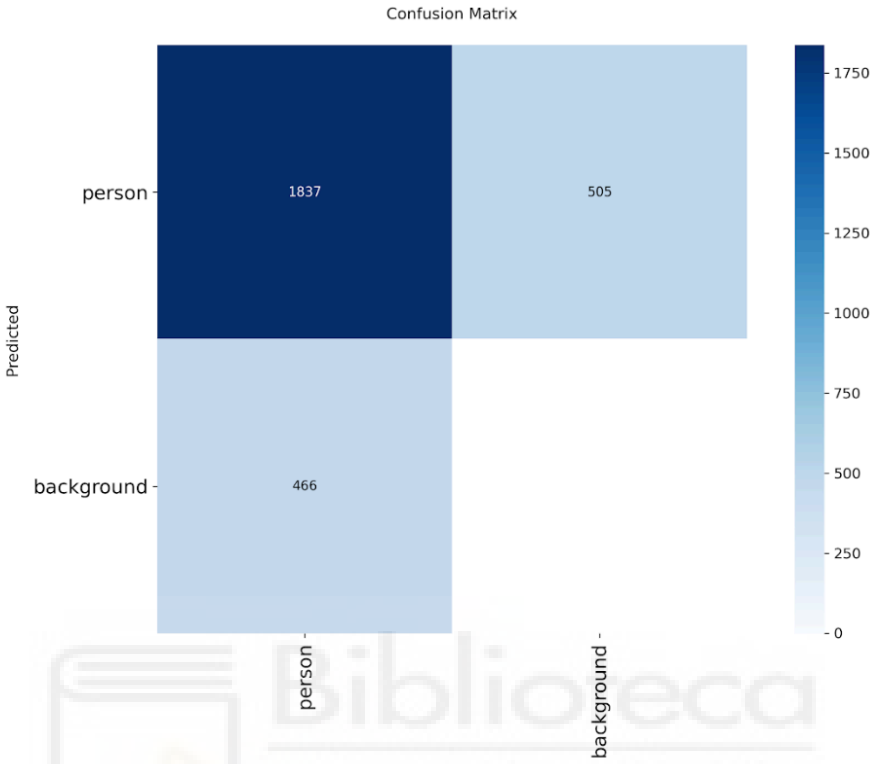

Fig 4.8. Matriz de confusión de YOLO8n

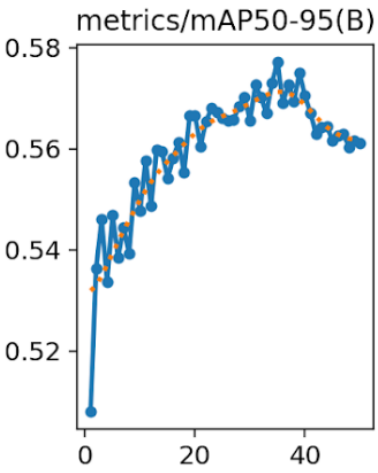

Fig. 4.9. Gráfica mAP50-95 del entrenamiento de YOLO12n

La razón por la que esta métrica ha ocasionado que el proceso finalizara antes de completar todas las épocas está relacionada con el hiperparámetro "*patience*". Lo que hace este es monitorear el valor de mAP50-95 para que, en el caso de que no mejore su valor en el número de épocas que se le indique, termine el proceso.

Durante cada época, si ha habido mejoras respectivas a este último hiperparámetro, se guarda una versión del modelo entrenado (best.pt). Aun así, en caso de que no se supere este valor, tras cada época se guarda el modelo generado en esta también (last.pt). Para la selección final, se ha escogido el fichero best.pt respectivo de cada modelo. Estos se utilizarán para realizar la toma de métricas de rendimiento y precisión de los modelos y su posterior comparación.

4.4.- Comparación de resultados obtenidos

Teniendo los modelos ya ajustados para realizar la tarea que deseamos, se puede comenzar a tomar distintos valores que demuestren cual es el rendimiento de cada uno. Para esta tarea, se ha seleccionado el conjunto de validación del dataset y usado en los 3 modelos.

```python
metricas = modelo.val(data="/content/TFG-1/data.yaml")

print("=== Precisión y rendimiento ===")
print(f"mAP50: {metricas.box.map50:.4f}")
print(f"mAP50-95: {metricas.box.map:.4f}")
print(f"Precisión (P): {metricas.box.mp:.4f}")
print(f"Recall (R): {metricas.box.mr:.4f}")
```

Fig. 4.10. Código para obtener las métricas de precisión

```python
inicio = time.time()
for _ in range(50):
    results = model(img, verbose=False)
final = time.time()
mediaTiempo = (final - inicio) / numInferencias
fps = 1 / mediaTiempo

print(f"Tiempo medio por inferencia: {avg_inference_time:.4f} segundos")
print(f"FPS estimados: {fps:.2f}")

torch.cuda.reset_peak_memory_stats()
memoriaInicial= torch.cuda.memory_allocated() / 1024**2  # en MB
infrencia= model(img, verbose=False)
memoriaFinal= torch.cuda.max_memory_allocated() / 1024**2  # en MB
print(f"Memoria GPU usada: {memoriaFinal - memoriaInicial:.2f} MB")
```

Fig. 4.11. Código para obtener las métricas de rendimiento

En la tabla 4.1, se hace un desglose de cada métrica calculada con su valor por modelo entrenado. Para realizar el análisis comparativo, se ha observado las principales diferencias entre cada uno de los valores apuntados.

Tabla 4.1: Comparativa de métricas

	YOLO5nu	YOLO8n	YOLO12n
mAP50	0.8238	0.8334	0.8399
mAP50-95	0.5389	0.5570	0.5767
Precisión (P)	0.8594	0.8263	0.8592
Recall (R)	0.7134	0.7538	0.7392
Tiempo promedio de inferencia	0.0118 s	0.0118 s	0.0212 s
FPS estimados	84.45	85.03	47.26
Tamaño del modelo	5 MB	6 MB	5.3 MB
Consumo de GPU	18.62 MB	20.74 MB	23.70 MB

En cuanto a la precisión individual de cada uno, el parámetro que más va a afectar en la decisión final es en Recall. Este indica el porcentaje de detecciones realizadas correctamente, lo que es primordial para su ejecución en entornos de tiempo real. Por esta misma razón su valor es determinante en la decisión final contando las métricas de rendimiento. Se observa que la diferencia entre los modelos no es muy notable en cuanto a precisión se trata. Se puede concluir que en términos de precisión, el modelo YOLO12n es ligeramente superior que los otros dos, lo que indica que ha habido una evolución positiva tras cada actualización de la arquitectura.

Pasando al rendimiento, si que se observa que existen diferencias notables, sobre todo observando tiempo de inferencia y FPS estimados. YOLO12n presenta una diferencia notable, tardando aproximadamente el doble de tiempo en ejecutar sus inferencias. Esto también influye en el rendimiento calculado que tendría el modelo si se desplegara en un sistema, siendo un 50% más lento que los otros dos modelos.

```python
from ultralytics import YOLO

modelo = YOLO('ruta/al/fichero/best.pt')

modelo.export(format='tflite')
```

Fig. 4.12. Código para exportar el modelo

Teniendo todo esto en cuenta, se ha decidido seguir con el desarrollo haciendo uso del modelo YOLO8n. Sus valores presentan una buen equilibrio entre rendimiento y precisión, además de presentar el mayor valor en cuanto a Recall. Con la decisión ya realizada, se

selecciona el fichero correspondiente para exportar el modelo en formato Tensorflow Lite (Fig. 4.11.). Este formato es el más común y recomendable para ser utilizado en modelos destinados a entornos con recursos limitados, como se ha comentado en puntos anteriores (apartado 2.4.2).

Un dato importante que se debería recuperar durante el proceso de exportación es el tamaño de los tensores. Tras finalizar el proceso, los logs de la ejecución indican en una línea cuáles son las dimensiones del tensor de entrada y el de salida. Esto será necesario para, más adelante, configurar el modelo en la aplicación de detección de objetos

4.5.- Diseño de la aplicación móvil

Antes de comenzar el desarrollo de la aplicación para dispositivos móviles, es muy importante tener claro qué se quiere crear. Esto agiliza el proceso permitiendo tener claro los hitos que se deben conseguir. Indicaremos este punto a continuación. Por ello, se van a definir previamente tanto las funcionalidades, como el diseño y la estructura del proyecto.

4.5.1.- Diseño inicial

Primero, se ha realizado un diseño a mano alzada de la interfaz que se desea implementar (Fig. 4.12.). Se seguirá un diseño de una única pantalla, sobre la cual se incluirá un botón y un contador de las personas que se detecten en cada inferencia. Tras cada predicción realizada por el modelo, dicho contador se actualizará y la pantalla del dispositivo móvil dibujará una caja alrededor de la persona detectada, usando los datos devueltos por el modelo para ello

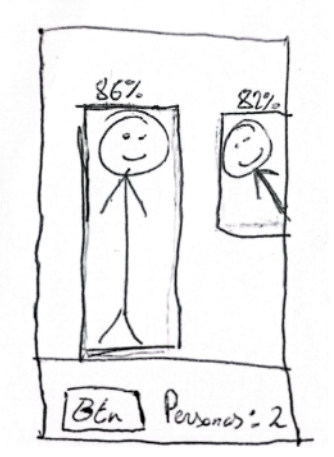

Fig 4.13. Dibujo a mano alzada de la interfaz (mockup)

Dado que la aplicación tendrá un funcionamiento sencillo y no presenta funcionalidades muy complejas, no es necesario crear los diagramas propios de la creación de aplicaciones software. Aun así, para tener registrado como se debe comportar la aplicación, se ha creado un diagrama de flujo sencillo.

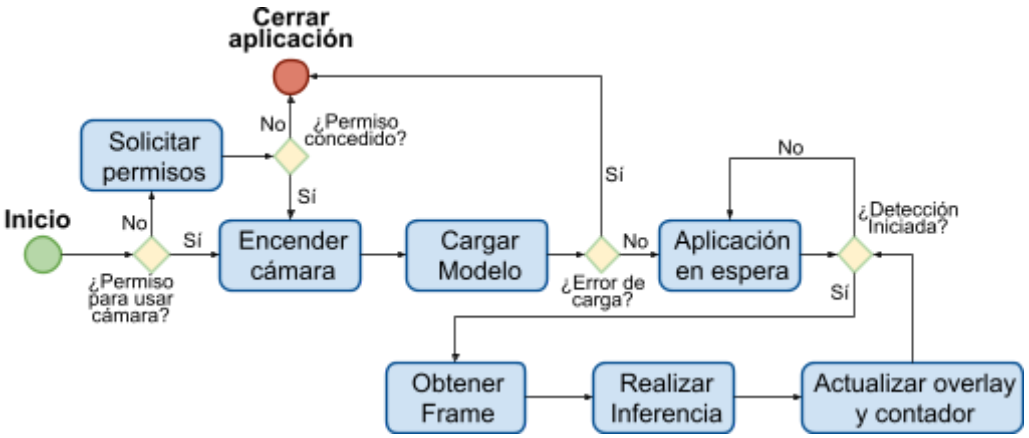

Fig. 4.14. Diagrama de flujo

La aplicación requiere de permisos para usar la cámara del móvil, por lo que lo principal es solicitarlos antes de poder iniciarse completamente. Tras permitirlo, se cargará el modelo y la aplicación mostrará la visión de la cámara. Para comenzar las inferencias, únicamente habrá que pulsar el botón que se muestra en pantalla. Haciendo esto, la aplicación recuperará en bucle frames de la cámara sobre las cuales realizará inferencias. Tras cada una, se actualizará la vista de la aplicación, mostrando el conteo y las correspondientes bounding boxes. Para finalizar este proceso, únicamente habrá que volver a pulsar el botón en pantalla.

4.5.2.- Estructura del proyecto

Para almacenar los ficheros del proyecto, se usará la estructura habitual que genera el entorno de desarrollo Android Studio. El modelo seleccionado se almacenará en una carpeta creada manualmente llamada "*assets*" para seguir buenas prácticas de desarrollo. Para las variables de texto, se empleará el fichero predeterminado "*strings.xml*" y para la interfaz, se generará un archivo XML llamado "*activity_main.xml*" dentro de la carpeta "*layout*".

Para la parte del Backend de la aplicación, se dividirá el código en función de su uso en diferentes ficheros para cada clase que se vaya a emplear. Las clases previstas a usar son las siguientes:

- MainActivity: contiene la lógica principal de la aplicación para iniciarse e interactuar entre las distintas clases adicionales que se creen.
- PersonDetector: Tiene funcionalidades necesarias para inicializar el modelo, realizar las inferencias y guardar los datos calculados por YOLO8n.
- DetectionOverlay: Lógica para actualizar la pantalla tras cada inferencia.
- ImageUtils: Funciones creadas para realizar la conversión del input que recibe la aplicación desde la cámara del dispositivo móvil. Esta clase es muy importante pues es necesario hacer dicha conversión para poder trabajar correctamente con el modelo.
- AppConfig: Clase auxiliar donde se centralizan los parámetros utilizados por la aplicación, como dimensiones de las bounding boxes o el umbral de confianza para filtrar las detecciones.

4.6.- Creación de la aplicación con el modelo implementado

Teniendo el conocimiento completo sobre cómo va a trabajar la aplicación y su organización de ficheros, se puede comenzar el desarrollo final. El objetivo de este punto es hacer hincapié en todas las funcionalidades implementadas para el correcto funcionamiento del modelo y el tratamiento de los datos de entrada y salida necesarios para un correcto funcionamiento.

Se comenzará indicando como se carga el modelo en memoria para su uso. Luego, se explicarán puntos a tener en cuenta para realizar las inferencias, dado que es muy importante tener unos conocimientos previos sobre los tipos de entrada y salida que necesita el modelo. Por último, se indicará cómo se realiza la actualización continua de la interfaz de la aplicación tras cada inferencia.

4.6.1.- Carga del modelo

El primer paso es cargar el modelo en memoria para poder efectuar las inferencias. Para ello se usa el fichero del detector de objetos en formato Tensorflow Lite generado previamente. Este modelo se invoca en el código haciendo uso de una clase propia de la API de Tensorflow Lite, "*Interpreter*" [55]. Su configuración se hace principalmente creando una variable donde se cargue el fichero .tflite del modelo.

Adicionalmente, se puede crear otro tipo de variable para definir la configuración que seguirá el modelo cuando se cargue. Esto ayuda a controlar de mejor forma cómo se comportará el modelo durante la ejecución de inferencias. Por ejemplo, es posible definir

el número de hilos que emplea el modelo o activar opciones de optimización para mejorar el rendimiento.

```kotlin
var interpreter: Interpreter? = null

// Cargar el modelo
val modelo: MappedByteBuffer = FileUtil.loadMappedFile(context, modelPath)

// Configuración del intérprete
val options = Interpreter.Options().apply {
        setNumThreads(threadCount.coerceIn(1, 4))
        setUseNNAPI(false) // Desactivado para mayor compatibilidad
        setUseXNNPACK(true) // Optimización CPU
}

interpreter = Interpreter(modelo, options)
```

Fig. 4.15. Código para cargar el modelo

4.6.2.- Ejecución de inferencias

Como se ha comentado al comienzo de este punto, hay que tener múltiples consideraciones a la hora de llamar al modelo para ejecutar inferencias. Lo primero es configurar en dos variables el tamaño de los tensores de entrada y salida del modelo. Este valor podemos recuperarlo durante la conversión del modelo a .tflite (explicado en el último párrafo del apartado 4.4).

En el caso del modelo YOLO8n ajustado a una única clase "*person*", los tamaños son los siguientes, para el tensor de entrada, (1, 3, 640, 640); y para el de salida, (1, 5, 8400). A continuación se explicará que indica cada valor en el orden en el que aparecen indicados:

- Tensor de entrada:
    - Tamaño de lote, indicando que se va a procesar una única imágen.
    - Número de canales de la imágen, en este caso el estándar RGB.
    - Altura de la imágen.
    - Anchura de la imágen.

- Tensor de salida:
    - Tamaño de lote, indicando que se va a procesar una única imágen.
    - Número de valores devueltos por la detección (x, y, w, h, score).
    - Número de candidatos a detección.

Este último valor puede resultar confuso de explicar, pero proviene de cómo se realizan las predicciones en los modelos YOLO [56]. Visto esto, ya se puede definir correctamente la salida y entrada para trabajar correctamente con las inferencias.

Para evitar la sobrecarga del dispositivo al ejecutar diversos procesos al mismo tiempo, se emplea tanto la librería "*coroutines*" de Kotlin como la extensión de Android Jetpack "*lifecycles*" [57]. Su uso conjunto se realiza llamando a diferentes hilos mediante los contextos indicados en el objeto "*Dispatchers*". Hay varios hilos predeterminados a los que se puede invocar para hacer ejecuciones, se han usado los siguientes:

- Default: Usado para ejecutar tareas intensivas en CPU.
- Main: Corresponde al hilo principal.
- IO: Destinado a operaciones de entrada y salida.

```kotlin
detectionJob = lifecycleScope.launch(Dispatchers.Default) {

  try {
    val detections = detector.detectarPersonas(bitmap)
    // Actualizar UI en el hilo principal
    withContext(Dispatchers.Main) {
      if (isActive && deteccionActiva) {
        actualizarDetecciones(detections, bitmap.width, bitmap.height)
        actualizarUI(detections)
      }
    }
  }

  catch (e: Exception) {
    Log.e(TAG, "Error en detección", e)
  }

  finally {
    // Liberar bitmap
    if (!bitmap.isRecycled) {
      bitmap.recycle()
    }
    isProcessing.set(false)
  }

}
```

Fig. 4.16. Código para ejecutar múltiples hilos

Sabiendo todo esto se ha configurado una clase interna llamada "*analizadorImagen*" en el archivo principal del proyecto. Esta hereda de la clase "*Analyzer*" presente en la librería "*ImageAnalisis*" [58]. Esta clase obtiene directamente de la cámara los frames que captura

usando una variable tipo "*ImageProxy*". Dicha variable no puede usarse directamente para realizar las inferencias, por lo que hay que hacer una conversión de este tipo de variable a una "*BitMap*" para ello.

Una vez configurado todo correctamente, es posible realizar las inferencias de una manera segura sin que la aplicación detecte errores inesperados. La Fig. 4.16. muestra la función creada para realizar las detecciones, incluyendo algunas verificaciones previas y llamando finalmente al método "*run*" para realizar la predicción.

Kotlin

```kotlin
fun detectarPersonas(bitmap: Bitmap): List<Detection> {

  val interpreter = this.interpreter ?: return emptyList()
  val inputArray = this.inputArray ?: return emptyList()
  val outputArray = this.outputArray ?: return emptyList()

  return try {
    if (bitmap.isRecycled || bitmap.width <= 0 || bitmap.height <= 0) {
      Log.w(TAG, "Bitmap inválido")
      return emptyList()
    }

    val resizeInfo = if (bitmap.width != 640 || bitmap.height != 640) {
       ImageUtils.resizeWithPadding(bitmap, 640)
    } else {
       ImageUtils.ResizeInfo(bitmap, 1f, 1f, 0f, 0f)
    }


    llenarInput(inputArray, resizeInfo.bitmap)
    limpiarOutput(outputArray)

    // Ejecutar inferencia
    Log.d(TAG, "Ejecutando inferencia...")
    interpreter.run(inputArray, outputArray)

    if (resizeInfo.bitmap != bitmap && !resizeInfo.bitmap.isRecycled) {
       resizeInfo.bitmap.recycle()
    }

    return procesarSalida(bitmap, outputArray)

  } catch (e: Exception) {
    Log.e(TAG, "Error durante detección", e)
    return emptyList()
  }
}
```

Fig. 4.17. Código para ejecutar las inferencias

4.6.3.- Visualización del resultado

Para visualizar los datos obtenidos durante la inferencia, es necesario procesarlos correctamente. Hay que tener en cuenta el formato que tiene el tensor de salida del modelo. Lo que hay que hacer es recuperar, por cada una de las detecciones que ha hecho el modelo, los valores (x, y, w, h, score).

Esta tarea no supone mucho problema, pero hay que añadir una comprobación extra para obtener las detecciones. Se aplica una técnica llamada Non-Maximum Suppression [57] que se usa para filtrar detecciones válidas e inválidas. Su funcionamiento trata de comprobar el valor de IoU entre detecciones ya validadas y nuevas detecciones. Definiendo un umbral para este valor, se logra realizar este filtro. Es importante seleccionar correctamente este valor, dado que podría ocurrir que se validaran detecciones duplicadas tras una inferencia.

Hechas estas comprobaciones, se obtiene una lista de detecciones en una variable tipo "*RectF*", que almacena distintos valores que indican las dimensiones de una figura rectangular.

Para visualizar las bounding boxes en tiempo real, se define un overlay superpuesto a la pantalla principal, sobre la cual se dibujan las cajas usando los valores de cada detección. Añadido a esto, como tras validar las detecciones se obtiene el número final de detecciones válidas, también se actualiza el valor del contador en pantalla con dicho número.

4.7.- Validación de la aplicación

Finalizada la creación completa de la aplicación, el último paso es la realización de distintas pruebas en entornos reales para poder validar que el funcionamiento del modelo es el esperado. El objetivo principal de esta aplicación es detectar personas en entornos no muy concurridos tanto de interiores como de exteriores.

Para las pruebas se ha usado un dispositivo móvil personal, concretamente un Samsung Galaxy A35, cuya memoria es de 8 GB. Se ha comprobado realizando diversas ejecuciones de la aplicación los siguientes puntos:

- La aplicación inicia correctamente, inicia la cámara y carga el modelo.
- La cámara no presenta tirones durante el uso de la aplicación.
- Las detecciones se visualizan correctamente sobre la vista de la cámara.
- El tiempo de respuesta es razonable para la configuración escogida.

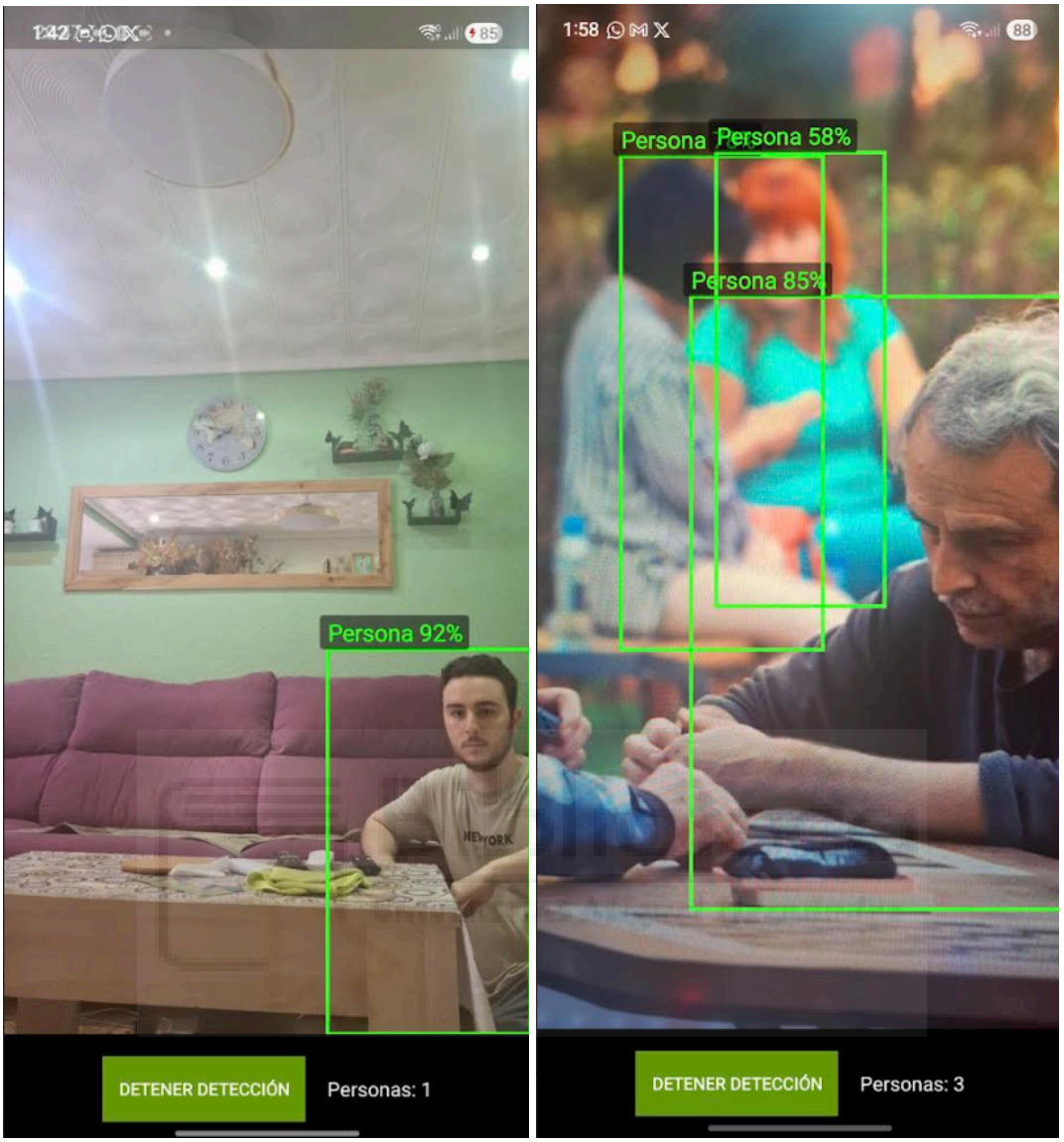

Fig. 4.18. Prueba de detección en interior y exterior

En la Fig 4.17 se muestran dos ejemplos de detección realizada por la aplicación durante las ejecuciones de prueba. En la imágen de la izquierda se detecta con un 92% de confianza a la persona de la imágen. Para este caso la detección se ha realizado en un espacio interior.

En la parte derecha de la misma figura se muestra una ejecución diferente. Esta se ha hecho detectando las personas procedentes de un video tomado en una entorno exterior, concretamente en un parque. En esta se ha detectado correctamente a las 3 personas en la imágen, incluyendo las 2 del fondo que se encuentran menos nítidas a la cámara.

Capítulo 5
Conclusiones y trabajo futuro

5.1.- CONCLUSIONES

Al principio de mi proyecto, dejé constancia de mi interés sobre el funcionamiento de modelos de Deep Learning, concretamente en el ámbito del Computer Vision. Tras completar este, he descubierto múltiples factores que tener en cuenta para crear y crear modelos de detección de objetos.

Si se tiene poca experiencia y no es necesario disponer de un modelo altamente complejo, existen numerosas opciones en el mercado para ello. Si por otro lado, se quiere usar un modelo personalizado para tareas concretas, conlleva un desarrollo alargado. Lo más importante es tener en cuenta los datos de entrenamiento, ya que ellos moldean cómo se va a comportar el resultado final.

En lo personal, la realización de este trabajo ha conseguido que mi interés por estas herramientas haya aumentado. Conociendo ahora todo el proceso que hay detrás, me motiva a seguir buscando información relativa al tema. Si que es cierto que el proceso es exhaustivo (anotación y revisión de imágenes, procesos largos de entrenamiento, etc) pero, viendo el resultado final, queda claro que este es un campo de estudio interesante y muy útil.

5.2.- POSIBLES DESARROLLOS FUTUROS

En lo que respecta a entrenamiento de modelos y creación de aplicaciones, el proyecto podría evolucionar de distintas formas. Tratando de aumentar el alcance de un desarrollo de este estilo, podría implementarse numerosas variaciones.

En cuanto al entrenamiento, se podría implementar técnicas avanzadas de fine-tuning, como el entrenamiento incremental. También está la posibilidad de utilizar datasets personalizados más variados o específicos para escenarios concretos (detección de exteriores, entornos de baja luminosidad, etc).

En relación a la aplicación móvil, se proponen varias m,ejoras funcionales que incrementarían su versatilidad. Entre ellas destacan:

- Selección dinámica de modelo: Relacionado con la mejora propuesta para espacializar los modelos a situaciones concretas, podría implementar un selector de modelos para distintos entornos. También se podría enfocar este punto en usar distintos tipos de modelos que varíen en precisión y rendimiento.
- Compatibilidad con cámara frontal: Añadir soporte a la aplicación para usar tanto la cámara trasera (desarrollo actual) como la frontal en caso de quererlo así.
- Rotación automática de la cámara: Permitir que la aplicación adapte la vista y proceso de inferencia en función de la orientación de la cámara. Este desarrollo conlleva tratar los datos de inferencia de maneras distintas para orientación vertical y horizontal.
- Selector de imágenes en galería: Implementar una opción para cargar imágenes presentes en el almacenamiento del dispositivos para hacer pruebas puntuales sin necesidad de utilizar inferencia en tiempo real.

Estas extensiones permitirían conseguir una herramienta con un grado más alto de profesionalidad. También ampliará el alcance de uso de la aplicación a otros entornos.

Bibliografía

[1] On Computable Numbers, With An Application To The Entscheidungsproblem
A. M. Turing (1936)

[2] Machine Learning and Deep Learning: A Review of Methods and Applications
Koosha Sharifani, Mahyar Amini (2023)
https://papers.ssrn.com/sol3/papers.cfm?abstract_id=4458723

[3] Big data analytics on Apache Spark
Salman Salloum, et al. (2016)
https://link.springer.com/article/10.1007/s41060-016-0027-9

[4] Deep learning interpretation of echocardiograms
Amirata Ghorbani, et al. (2020)
https://tvst.arvojournals.org/article.aspx?articleid=2762344

[5] Deep Learning in Agriculture: A Review
Pallab Bharman, et al. (2022)
https://acortar.link/y2Z8vb

[6] Applied Computer Vision on 2-Dimensional Lung X-Ray Images for Assisted Medical Diagnosis of Pneumonia
Ralph Joseph S.D. et al. (2022)
https://arxiv.org/pdf/2207.13295

[7] Comparación de algoritmos detectores de puntos singulares para reconocimiento de objetos en vídeo quirúrgico
I. García Barquero, et al. (2012)
https://oa.upm.es/20480/1/INVE_MEM_2012_135438.pdf

[8] Object Recognition in Different Lighting Conditions at Various Angles by Deep Learning Method
Imran Khan Mirani, et al. (2022)
https://arxiv.org/pdf/2210.09618

[9] Deep Residual Learning for Image Recognition
Kaiming He, et al. (2016)
https://www.cv-foundation.org/openaccess/content_cvpr_2016/papers/He_Deep_Residual_Learning_CVPR_2016_paper.pdf

[10] Documentación de Google Cloud Vision API
https://cloud.google.com/vision/docs?hl=es-419

[11] Documentación de Vertex AI
https://cloud.google.com/vertex-ai/docs?hl=es-419

[12] Amazon Rekoginition
https://aws.amazon.com/es/rekognition/

[13] Amazon Rekognition - Working with images and videos
https://docs.aws.amazon.com/rekognition/latest/dg/programming.html

[14] Azure AI Vision
https://portal.vision.cognitive.azure.com/gallery/featured

[15] Clarifai - Computer Vision
https://www.clarifai.com/computer-vision

[16] Why Pre-Trained Models Matter for Machine Learning.
https://www.ahead.com/resources/why-pre-trained-models-matter-for-machine-learning/

[17] Desarrollos de la Ingeniería ambiental en la evaluación de la calidad de los recursos naturales y la salud ambiental.
Edgar Serna M. (2017)
https://acortar.link/snIYLO

[18] What is Transfer Learning? Exploring the Popular Deep Learning Approach.
Niklas Donges
https://builtin.com/data-science/transfer-learning

[19] Convolutional Neural Networks (CNNs), Deep Learning, and Computer Vision
https://www.intel.com/content/www/us/en/internet-of-things/computer-vision/convolutional-neural-networks.html

[20] What is fine-tuning?
Dave Bergmann
https://www.ibm.com/think/topics/fine-tuning

[21] COCO Dataset
https://cocodataset.org/#home

[22] ¿Por qué es necesario normalizar los valores de los píxeles antes de entrenar el modelo?
https://acortar.link/G4TEBL

[23] Deep Learning with Python (Capítulo 5.2.4)
François Chollet
https://www.manning.com/books/deep-learning-with-python

[24] A survey on Image Data Augmentation for Deep Learning
Connor Shorten, et al. (2019)
https://journalofbigdata.springeropen.com/articles/10.1186/s40537-019-0197-0

[25] Documentación de TensorFlow
https://www.tensorflow.org/api_docs

[26] COCO-SSD con TensorFlow
dbcp1
https://github.com/tensorflow/tfjs-models/tree/master/coco-ssd

[27] MediaPipe FaceDetection
dbcp1
https://github.com/tensorflow/tfjs-models/tree/master/face-detection

[28] Guía de TensorFlow Lite
https://www.tensorflow.org/lite/guide?hl=es-419

[29] StreamNet: Memory-Efficient Streaming Tiny Deep Learning Inference on the Microcontroller
Hong-Sheng Zheng et al. (2023)
https://proceedings.neurips.cc/paper_files/paper/2023/file/7526508f11bbe0a123af62b9dab1fbe1-Paper-Conference.pdf

[30] Documentación de PyTorch
https://pytorch.org/get-started/locally/

[31] Detectron2
Yanghang Wang
https://github.com/facebookresearch/detectron2

[32] Roboflow - Detectron2
https://roboflow.com/model/detectron2#:~:text=Detectron2%20is%20a%20computer%20vision,datasets%20in%20COCO%20JSON%20format

[33] ExecuTorch Documentation
https://pytorch.org/executorch/stable/index.html

[34] A Review of Yolo Algorithm Developments
Peiyuan Jiang et al. (2022)
https://www.sciencedirect.com/science/article/pii/S1877050922001363

[35] YOLO12
https://docs.ultralytics.com/es/models/yolo12/

[36] Ultralytics YOLO Docs
https://docs.ultralytics.com/es

[37] Optimization of Hyperparameters in Object Detection Models Based on Fractal Loss Function
Ming Zhou et al. (2022)
https://www.mdpi.com/2504-3110/6/12/706

[38] How does learning rate decay help modern neural networks?
Kaichao You et al. (2019)
https://arxiv.org/pdf/1908.01878

[39] Early Stopping in Deep Learning: A Simple Guide to Prevent Overfitting Piyush Kashyap
https://medium.com/@piyushkashyap045/early-stopping-in-deep-learning-a-simple-guide-to-prevent-overfitting-1073f56b493e

[40] On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima
Nitish Shirish Keskar et al. (2017)
https://arxiv.org/pdf/1609.04836

[41] What Does Batch Size Mean in Deep Learning? An In-Depth Guide
Coursera Staff
https://www.coursera.org/articles/what-does-batch-size-mean-in-deep-learning

[42] What is Dropout Regularization in Deep Learning?
Aashi Verma
https://www.pickl.ai/blog/what-is-dropout-regularization-in-deep-learning/

[43] Object Detection with Deep Learning: A Review
Zhong-Qiu Zhao et al. (2019)
https://arxiv.org/pdf/1807.05511

[44] Hands-On Machine Learning with Scikit-Learn and TensorFlow (Capítulo 11)
Aurélien Géron
https://www.clc.hcmus.edu.vn/wp-content/uploads/2017/11/Hands_On_Machine_Learning_with_Scikit_Learn_and_TensorFlow.pdf

[45] A Comparative Analysis of Object Detection Metrics with a Companion Open-Source Toolkit
Rafael Padilla et al. (2021)
https://paperswithcode.com/paper/a-comparative-analysis-of-object-detection

[46] Python Developers Survey 2023 Results
https://lp.jetbrains.com/python-developers-survey-2023/

[47] Coroutines Kotlin Documentation
https://kotlinlang.org/docs/coroutines-overview.html

[48] Roboflow
https://roboflow.com

[49] Roboflow Developer Documentation
https://docs.roboflow.com/developer

[50] Innovative Deep Learning Techniques for Obstacle Recognition: A Comparative Study of Modern Detection Algorithms
Santiago Pérez et al. (2024)
https://arxiv.org/pdf/2410.10096

[51] COCO Dataset Limited (Person Only) Computer Vision Project
shreks swamp (2022)
https://universe.roboflow.com/shreks-swamp/coco-dataset-limited--person-only

[52] Guía de TensorFlow Lite
https://www.tensorflow.org/lite/guide?hl=es-419

[53] A Comprehensive Study of Modern Architectures and Regularization Approaches on CheXpert5000
Sontje Ihler et al. (2023)
https://arxiv.org/abs/2302.06684

[54] Documentación de Ultralytics en Python
https://docs.ultralytics.com/es/usage/python/

[55] Clase Interpreter de Tensorflow Lite
https://ai.google.dev/edge/api/tflite/java/org/tensorflow/lite/Interpreter

[56] YOLOv8 final detection output
https://acortar.link/xiX8GA

[57] Documentación de lifecycle
https://developer.android.com/jetpack/androidx/releases/lifecycle

[58] Documentación de ImageAnalysis
https://developer.android.com/reference/androidx/camera/core/ImageAnalysis

[59] A Deep Dive Into Non-Maximum Suppression (NMS)
https://builtin.com/machine-learning/non-maximum-suppression
Chinmay Bhalerao (2023)