APLICACIÓN DE TÉCNICAS DE APRENDIZAJE PROFUNDO EN IMÁGENES PARA EL RECONOCIMIENTO DE OBJETOS

APPLICATION OF DEEP LEARNING TECHNIQUES FOR OBJECT RECOGNITION IN IMAGES

TRABAJO DE FIN DE MÁSTER
CURSO 2021-2022

AUTOR
MANUEL GUERRERO MOÑÚS

DIRECTOR
GONZALO PAJARES MARTÍNSANZ

MÁSTER EN INGENIERÍA INFORMÁTICA
FACULTAD DE INFORMÁTICA
UNIVERSIDAD COMPLUTENSE DE MADRID

APLICACIÓN DE TÉCNICAS DE APRENDIZAJE PROFUNDO EN IMÁGENES PARA RECONOCIMIENTO DE OBJETOS

APPLICATION OF DEEP LEARNING TECHNIQUES FOR OBJECT RECOGNITION IN IMAGES

TRABAJO DE FIN DE MÁSTER EN INGENIERÍA INFORMÁTICA
DEPARTAMENTO DE INGENIERÍA DEL SOFTWARE E INTELIGENCIA ARTIFICIAL

AUTOR
MANUEL GUERRERO MOÑÚS

DIRECTOR
GONZALO PAJARES MARTÍNSANZ

CONVOCATORIA: JUNIO 2022
CALIFICACIÓN: 10

MÁSTER EN INGENIERÍA INFORMÁTICA
FACULTAD DE INFORMÁTICA
UNIVERSIDAD COMPLUTENSE DE MADRID

22 DE JUNIO DE 2020

DEDICATORIA

A toda mi familia, especialmente a mis padres y a mi hermana, que nunca tiraron la toalla conmigo, y a quienes estoy enormemente agradecido de haberme inculcado el valor del esfuerzo para superar todos los retos que me propusiera. A mis amigas del instituto, que tanto me quieren. A mis amigos, compañeros y profesores de: el CFGS DAM del instituto Joyfe, la Facultad de Informática de la UCM, y la academia Maths Informática. Quienes me animaron y ayudaron a crecer en lo profesional y personal. A Lu, mi amiga más antigua.

AGRADECIMIENTOS

Quiero agradecerle a mi profesor de Ingeniería del Conocimiento, y mi director de TFG y TFM, Gonzalo Pajares Martínsanz, que me haya permitido participar con él en estos trabajos tan sumamente interesantes en el campo del Aprendizaje Profundo. Pues gracias a él he ampliado mi formación académica, y mi visión de la Ingeniería Informática y el papel que desempeña en nuestra sociedad, una sociedad a la que todos, con voluntad y esfuerzo, podemos contribuir a mejorar.

RESUMEN

El presente trabajo se engloba dentro del ámbito del Aprendizaje Profundo, donde los modelos de Redes Neuronales Artificiales se han convertido a lo largo del tiempo en una pieza clave para resolver problemas tan complejos como el reconocimiento de imágenes, voz, movimientos corporales, o el procesamiento del lenguaje natural.

En este trabajo en particular, la atención se ha puesto en aquellos modelos de Red Neuronal Artificial, dentro de los denominados convolucionales, que se han diseñado para reconocer, ubicar, e incluso en algunos casos, segmentar objetos en imágenes, pues son de gran interés de cara a su aplicación en diferentes ámbitos.

Sin embargo, la configuración y puesta a punto de los modelos para la detección de objetos mediante técnicas de aprendizaje profundo no resultan tarea fácil, pues se necesitan una serie de requerimientos hardware para poder trabajar con ellas, en particular, durante la fase de entrenamiento. Destaca en este sentido la necesidad de disponer de una amplia memoria RAM, discos duros rápidos con un espacio considerable, e incluso potentes GPU's o TPU's, económicamente costosas.

Por esta razón, se han creado una serie de programas que, apoyándose en la tecnología de *Google*, *FiftyOne*, y *TensorFlow*, solventan los anteriores requisitos con un mínimo coste, y proporcionan al usuario las siguientes funcionalidades:

- Generación, análisis y exportación automática de sets de imágenes para los procesos de entrenamiento y validación de los modelos.
- Configuración, entrenamiento, validación y exportación de modelos.
- Prueba de modelos para evaluar su rendimiento y desempeño real en cuanto a clasificación, ubicación y segmentación de objetos.

Finalmente, en lo que al marco teórico de este trabajo respecta, se ha realizado un estudio de la arquitectura, capas y operaciones de los siguientes detectores de objetos: SSD, R-CNN, FAST R-CNN, FASTER R-CNN, MASK R-CNN y YOLO.

PALABRAS CLAVE

Inteligencia Artificial, Aprendizaje Automático, Aprendizaje Profundo, Detección de Objetos, Segmentación de Objetos, Redes Neuronales Convolucionales.

CRÉDITOS DE IMAGEN

Las imágenes tomadas de la referencia [3] están autorizadas por Gonzalo Pajares Martínsanz, primer autor del libro Aprendizaje Profundo, y director de este trabajo.

ABSTRACT

The current work is under the Deep Learning field, where Artificial Neural Network models have become a key element to solve complex problems such as image recognition, voice, body movements, or natural language processing.

In this case, attention has been paid on those Artificial Neural Network models, within the convolutional models, that have been designed to recognize, locate, and even in some cases, segment objects in images. They are of great interest for its applications in scientific-engineering projects.

However, their configuration and set-up aren't easy tasks, it is necessary a series of hardware requirements to work with them, particularly during the training phase. In this sense, is important the need of a large RAM memory, fast hard disks with considerable space, and even powerful GPU's or TPU's, which are expensive.

For this reason, a series of programs have been created, that relying on *Google* technology, Voxel51, and TensorFlow, they solve the above requirements with a minimum cost, and provide its users the following functionalities:

- Automatic generation, analysis and export of image sets for processes of training and validation of models.
- Configuration, training, validation and export of models.
- Testing of models to evaluate their actual performance in terms of classification, location and segmentation of objects.

Finally, with respect to the theoretical framework of this work, a study of the architecture, layers and operations of certain models designed to detect objects has been carried out: SSD, R-CNN, FAST R-CNN, FASTER R-CNN, MASK R-CNN and YOLO.

KEYWORDS

Artificial Intelligence, Machine Learning, Deep Learning, Object Detection, Object Segmentation, Convolutional Neural Networks.

IMAGE CREDITS

Images taken from reference [3] are authorized by Gonzalo Pajares Martínsanz, first author of the book Aprendizaje Profundo, and the director of this work.

ÍNDICE DE CONTENIDOS

ÍNDICE DE FIGURAS

Capítulo 1 - Introducción

1.1 Antecedentes

Transcurridos los dos grandes Inviernos de la Inteligencia Artificial [1], ubicados entre (1974–1980) y (1986–1993), que se caracterizaron por la pérdida del interés en la investigación y la reducción de fondos para proyectos, este sector científico vive una nueva era gracias a los cambios tecnológicos de las últimas décadas, entre los que cabe destacar: un cierto abaratamiento del hardware, la optimización de los algoritmos, la evolución de los discos duros, la construcción de hardware específico para ciertos problemas computacionales, etc. Todo ello contribuyó, sin duda, a la aparición de un renovado interés investigador que ha posibilitado el impulso actual.

Esto ha traído de vuelta la inversión para la realización de toda clase de proyectos en el campo de la inteligencia artificial y sus sub-áreas, donde cobran una especial importancia la Visión por Computador (*Computer Vision*) [2] y el Aprendizaje Profundo (*Deep Learning*) [3], los campos dedicados al estudio del procesamiento inteligente de imágenes, donde el primero tiene un enfoque más tradicional, basado en el uso de algoritmos, mientras que el segundo se enfoca en el desarrollo de modelos computacionales de carácter bio-inspirado llamados, Redes Neuronales Artificiales [4, 19].

Estos modelos, aunque resuelven las mismas problemáticas que algunos algoritmos de visión, han demostrado ser superiores, sobre todo en lo que al reconocimiento de imágenes y objetos respecta, pues las Redes Neuronales Convolucionales (CNN's; *Convolutional Neural Networks*) [22, 23] generalizan mejor los conceptos de los elementos presentes en las imágenes al no tener los problemas que presentan los algoritmos de visión, los cuales requieren que dichos elementos sean rígidos, tengan una posición y orientación fijas, o posean unas características de estudio específicas [5].

Es por estas razones por las que los modelos neuronales se han impuesto frente a los algoritmos de visión, y por lo que su uso está cada vez más extendido. Podemos encontrarlas en proyectos muy diversos que, por lo general, ponen esta tecnología al servicio de las personas con fines de automatización de tareas, reducción de tiempos, mejora en la toma de decisiones, optimización de recursos, etc.

Un ejemplo de empresa que utiliza este tipo de modelos es Tesla, que se dedica a la fabricación de coches eléctricos de gama alta. Pero sus modelos no son exclusivos únicamente por sus lujosos diseños, sino también por los avanzados sistemas [6] que incorporan, como el de *Autopilot*, cuyo núcleo es HydraNet [7], una red formada a su vez por varias CNN's, o ramas derivadas. Las cámaras del vehículo capturan el entorno para alimentar con estos datos a HydraNet, que clasifica y ubica los elementos en los alrededores para utilizar esa información en la elaboración del trazado de la ruta hacia el destino.

Otra empresa que utiliza estas tecnologías es Airbus, un fabricante de aeronaves. El desarrollo de algunos de sus proyectos internos [8], como *Wayfinder* [9] y *ATTOL* [10], o la colaboración [11] entre estos, han dado como fruto la creación de aviones que son capaces de realizar las maniobras de rodaje, despegue, aproximación y aterrizaje, sin la intervención de pilotos. Esto es gracias a la introducción de modelos de Aprendizaje Profundo (*Deep Learning*) [3], en este caso concreto, una versión propia del modelo *Single Shot MultiBox Detector* [12], cuya información permite obtener la distancia a la pista, la desviación lateral con la línea central de la pista, y la desviación respecto de la pendiente de planeo.

Por otra parte, existen empresas que, en lugar de incorporar esta tecnología en sus productos, crean con ella soluciones a problemas específicos para posteriormente comercializarlas. Por ejemplo, Landing AI [13] creó ante la situación de pandemia de COVID una herramienta para ayudar a sus clientes a controlar el distanciamiento social en el trabajo. El proyecto consiste en tres sencillos pasos para su aplicación: calibración de una cámara de vídeo, uso de un modelo *Faster R-CNN* [14] para detectar peatones y dibujar sus cuadros delimitadores, y estimación de la distancia real entre todo par de personas.

Finalmente, existen empresas que solo proporcionan servicios o plataformas a sus clientes para que estos construyan sus propias soluciones de detección de objetos, este es el caso de Viso, Cogniac y Chooch.

Teniendo en cuenta las posibilidades de esta tecnología, su aplicación en grandes proyectos y el progreso tecnológico actual, se concluye que es de vital importancia, pues su desarrollo está ligado al progreso humano, motivo por el cual, su estudio y comprensión teórico-práctica se vuelve necesario.

1.2 Motivación

El principal interés del presente trabajo reside en la comprensión teórico-práctica de aquellos modelos del campo del Aprendizaje Profundo (*Deep Learning*) [3], que han sido diseñados para la detección de objetos en imágenes, las Redes Neuronales Convolucionales [22, 23], cuyo uso está en pleno auge debido a las posibilidades de sus aplicaciones en diferentes ámbitos tecnológicos, pues con el paso del tiempo se presentan cada vez más, nuevos enfoques y planteamientos para resolver de manera eficaz, toda clase de problemas relacionados con el sentido de la visión, que es sin duda, la principal fuente de información para la apreciar los sucesos que acontecen en un entorno.

Las problemáticas de esta faceta humana se encuentran presentes prácticamente en todas las actividades que realizamos, tanto cotidianas como profesionales, lo cual las convierte en poderosas herramientas para solucionar nuestros problemas mediante un enfoque computacional. Por ello, no es de extrañar que estos modelos se estén integrando continuamente en la sociedad actual, encontrándolos en: a) fábricas, donde supervisan la existencia de defectos en piezas, montajes y materiales; b) hospitales, donde su uso para poder reconocer patologías en las imágenes médicas es un apoyo vital a los profesionales sanitarios; c) aeropuertos, donde se utilizan como una parte de los modernos sistemas de seguridad de los controles para lograr la identificación de personas fichadas por la policía, armas o drogas; d) la industria de los transportes aéreos y terrestres, donde tratan de automatizar todas las maniobras del manejo de estos vehículos, o por el contrario, proporcionar asistencia a los conductores que deseen emplear los controles manuales; e) ciudades, donde su uso para la monitorización, ética y responsable, de ciertas actividades o acontecimientos posibilita la obtención de datos con los que mejorar la toma de decisiones y el reparto de recursos, lo cual las hace evolucionar, con la ayuda de otros conceptos y prácticas técnicas, hacia el modelo de Ciudad Inteligente, un enfoque más sostenible y vanguardista.

Llegados a este punto, es necesario comprender la importancia que tienen estos modelos, así como sus ventajas y limitaciones para con el progreso humano actual y venidero, lo que constituye en sí el verdadero motivo por el que explorar y crear soluciones inteligentes de vanguardia basadas en ellos.

1.3 Objetivos

El objetivo principal de este trabajo es ofrecer una solución conceptual sobre los requerimientos y la aplicabilidad tanto a nivel de hardware como de software, para cubrir las necesidades propias de las actividades para la puesta a punto de modelos CNN diseñados para el reconocimiento de objetos en imágenes (SSD, R-CNN, FAST R-CNN, FASTER R-CNN, MASK R-CNN, y YOLO). Dichas actividades se concretan en:

- Generación, análisis y exportación automática de *datasets* de imágenes para los procesos de entrenamiento y validación de los modelos.
- Configuración, entrenamiento, validación y exportación de modelos.
- Prueba de modelos para evaluar su rendimiento y desempeño real en lo que respecta a la clasificación, ubicación y segmentación de objetos.

La solución que aquí se plantea cubre los tres puntos anteriores proporcionando un diseño programado por cada uno de ellos, lo que simplifica cada labor y proporciona a su vez un marco de trabajo amigable. Para la construcción de estos programas se plantea el uso en las siguientes tecnologías, que son las finalmente estudiadas en profundidad: el lenguaje de programación Python3 [57], la *TensorFlow Object Detection API* [67], el software *FiftyOne* [77], y los servicios de *Google Colab Plus* [73], *Google Compute Engine* [77], *Google Cloud Storage* [78], y *Google Drive* [70].

Todas estas tecnologías conforman el núcleo de la plataforma modular desarrollada, por lo que se han estudiado convenientemente, adaptado, y modificado para lograr su integración en los desarrollos realizados.

Los objetivos concretos para lograr el desarrollo de esta solución inteligente que se plantea, se dividen en dos grupos, a saber: investigación, y desarrollo.

- Investigación:
    - Revisión bibliográfica, análisis preliminar, y recopilación de métodos basados en Aprendizaje Profundo para detectar objetos en imágenes.
    - Estudio de los lenguajes, librerías y *frameworks* disponibles para trabajar con modelos orientados al reconocimiento de objetos.
    - Búsqueda de servicios web que cubran las necesidades de trabajo:
        - Software: compatibilidad con el lenguaje y las librerías elegidas.

        - Hardware: amplia memoria RAM y espacio en disco, soporte para una GPU [64] de alta capacidad y TPU (*Tensor Processing Unit*) [65], amplio espacio para el almacenamiento de archivos en la nube.
    - Exploración de *datasets* de imágenes abiertos y anotados para realizar los procesos de entrenamiento y validación de los detectores.
- Desarrollo:
    - Implementación de un programa que automatice la generación, el análisis y la exportación de *datasets* con los que entrenar los modelos.
    - Construcción de un programa que entrene y valide los modelos ofreciendo métricas gráficas y numéricas. Los modelos podrán ser exportados.
    - Creación de un programa portable que permita probar el desempeño real de los modelos en pc's, portátiles, y microcomputadores.
    - Modificación de *scripts* y librerías para adaptar su funcionamiento a las necesidades operativas del proyecto.

1.4 Plan de trabajo

Para cumplir con los objetivos descritos en el apartado anterior se han realizado las actividades que se enumeran a continuación, sin embargo, cabe destacar que la realización de las mismas no siguió siempre este orden estricto, al menos durante las actividades técnicas (3-5). Esto se debió a las dificultades relacionadas con las limitaciones tecnológicas, y a la falta de experiencia con las tecnologías empleadas, que dieron pie a varias situaciones de revisión, cambios e intercalado de tareas.

Actividades planteadas y realizadas:

1. Establecimiento de los objetivos y los plazos de trabajo con el tutor.
2. Estudio de las tecnologías, y de las fuentes de imágenes abiertas y anotadas, disponibles para el desarrollo del proyecto y los procesos de los modelos.
3. Desarrollo de un generador automático de *datasets* para el entrenamiento y la validación de modelos de red orientados al reconocimiento de objetos.
4. Desarrollo de un programa para el entrenamiento, la validación y exportación de modelos de red orientados al reconocimiento de objetos.

5. Desarrollo de una aplicación portable para evaluar el desempeño real de los modelos en la clasificación, ubicación y segmentación de los objetos.
6. Redacción y organización de la memoria.

1.5 Organización de la memoria

La secuencia de capítulos que componen esta memoria es la siguiente:

1. Introducción: expuesta previamente, donde se comentan ejemplos reales de la aplicación de modelos para el reconocimiento de objetos en la sociedad actual, lo cual proporciona una visión general de su importancia, y ofrece, a través de la reflexión sobre sus aplicaciones, motivos suficientes como para centrarse en ellos. También se exponen cuáles son los objetivos de este trabajo y las actividades que han sido necesarias para alcanzarlos.
2. Marco conceptual: capítulos en los que se profundiza en los aspectos teóricos clave de diferentes modelos de red neuronal que han sido diseñados para detectar objetos en imágenes. Los modelos estudiados son: SSD, R-CNN, FAST R-CNN, FASTER R-CNN, MASK R-CNN, y YOLO.
3. Marco técnico: se detalla y justifica de forma individualizada cada una de las tecnologías utilizadas en los desarrollos del trabajo, también se expone el diseño conceptual que unifica todas estas tecnologías elegidas, lo que da una visión de conjunto del marco de trabajo que se pretende crear para facilitar a sus usuarios, el ciclo completo de actividades necesarias para poner a punto estos modelos.
4. Resultados: una vez se han comprendido los aspectos teóricos y tecnológicos del trabajo, se comentan los aspectos positivos y negativos tanto de los desarrollos, como de los resultados obtenidos de entrenar, validar y probar los modelos de red que están disponibles con la tecnología empleada.
5. Conclusiones y trabajo futuro: en este capítulo se exponen las conclusiones propias de la realización del trabajo, así como una serie de propuestas para mejorar los resultados obtenidos, y pulir aquellos aspectos técnicos que han sido más deficitarios.

Capítulo 2 - Introduction

2.1 Preliminary

After the two great winters of Artificial Intelligence [1] (1974-1980) and (1986-1993), which were characterized by the loss of interest in research and the reduction of funds for projects, this scientific sector is experiencing a new era thanks to the technological changes of recent decades, which include: a certain reduction of the hardware price, algorithms optimization, hard disks evolution, the construction of specific hardware for certain computational problems, etc. All this has contributed to the rise of a renewed research interest that has made the current advances possible.

This has brought back investment for the realization of all kinds of projects in the field of artificial intelligence and its sub-areas, where Computer Vision [2] and Deep Learning [3] have a special importance, the fields dedicated to the study of intelligent image processing, where the first has a more traditional approach, based on the use of algorithms, while the second is focused on the development of bio-inspired computational models called Artificial Neural Networks [4, 19].

These models, although they solve the same problems as some vision algorithms, have proven to be superior, especially in terms of image and object recognition, since Convolutional Neural Networks (CNN's; Convolutional Neural Networks) [22, 23] generalize better the concepts of the elements present in the images since they don’t have the problems presented by vision algorithms, which require that these elements be rigid, have a fixed position and orientation, or have specific study characteristics [5].

It’s for these reasons that neural models have prevailed over vision algorithms, and that is why their use is becoming increasingly extended. We can find them in very types of projects that, in general, put this technology at the service of people for the purpose of automating tasks, reducing time, improving decision making, optimizing resources, etc.

One example of company that uses these models is Tesla, which manufactures high end electric cars. But its models are not only exclusive due to their luxurious designs, but also because of the advanced systems [6] they incorporate, such as Autopilot, whose core is HydraNet [7], a network formed by several CNN's. The vehicle cameras capture the environment to provide its data to HydraNet, which classifies and locates the elements in the surroundings to use this information to create the route to the destination.

Another company that uses these technologies is Airbus, an aircraft manufacturer. The development of some of its internal projects [8], such as Wayfinder [9] and ATTOL [10], or the collaboration[11] between them, have resulted in the creation of aircrafts that are able of performing taxiing, takeoff, approach and landing maneuvers without the intervention of pilots. This is thanks to the introduction of Deep Learning models [3], in this particular case, a proprietary version of the Single Shot MultiBox Detector model [12], whose information allows obtaining the distance to the landing strip, the lateral deviation with the landing strip centerline, and the deviation with respect to the glide slope.

On the other hand, there are companies that, instead of incorporating this technology into their products, create solutions to specific problems and then commercialize them. For example, Landing AI [13] created a tool to help its customers manage social distancing at work due to the COVID pandemic situation. The project consists of three simple steps for implementation: calibration of a video camera, use of a Faster R-CNN model [14] to detect walkers and draw their bounding boxes, and estimation of the actual distance between any pair of people.

Finally, there are companies that only provide services or platforms to their customers so that they can build their own object detection solutions, this is the case of Viso, Cogniac and Chooch.

Keeping in mind the possibilities of this technology, its application in big projects and the current technological progress, it's concluded that it's of vital importance, since its development is linked to human progress, which is why its study and theoretical-practical understanding becomes necessary.

2.2 Motivation

The main interest of the present work resides in the theoretical and practical understanding of those models of the Deep Learning field [3], which have been designed for the detection of objects in images, the Convolutional Neural Networks [22, 23] whose use is growing due to the possibilities of their applications in different technological fields. Since as time goes by, new approaches are being presented to efficiently solve all kinds of problems related to the sense of vision, which is undoubtedly the main source of information to appreciate what is happening in an environment.

The problems of this human aspect are present in almost all the activities we perform, daily and professional, which makes them powerful tools to solve our problems through a computational approach. Therefore, it isn’t surprising that these models are continuously being integrated into today's society, being found in: a) factories, where they monitor the presence of defects in pieces, assemblies and materials; b) hospitals, where their use to recognize pathologies in medical images is a vital support to health professionals; c) airports, where they are used as part of modern security systems for the identification of people on police records, weapons or drugs; d) the air and ground transportation industry, where they try to automate all maneuvers in the operation of these vehicles, or on the other hand, to provide assistance to drivers who wish to use manual controls; e) cities, where their use for ethical and responsible monitoring of certain activities or events makes it possible to obtain data with which to improve decision making and resource distribution, which makes them move, with the help of other technical concepts and practices, towards the Smart City model, a more progressive and vanguard approach.

At this point, it’s necessary to understand the importance of these models, as well as their advantages and limitations for current and future human progress, which is the real reason to explore and create intelligent solutions based on them.

2.3 Objetives

The main objective of this work is to provide a conceptual solution at hardware and software level, to cover the needs of the set-up activities of CNN models designed for object recognition in images (SSD, R-CNN, FAST R-CNN, FASTER R-CNN, MASK R-CNN, and YOLO). These activities include the following:

- Automatic generation, analysis and export of image datasets for model training and validation processes.
- Configuration, training, validation and export of models.
- Testing of models to evaluate their true performance in terms of object classification, location and segmentation.

  The solution proposed here covers the three previous points providing a programmed design for each of them, which simplifies each task and provides a friendly framework. For the construction of these programs, the use of the following technologies is proposed, which are the ones finally studied in depth: Python3 programming language [57], *TensorFlow Object Detection API* [67], *FiftyOne* [77], *Google Colab Plus* [73], *Google Compute Engine* [77], *Google Cloud Storage* [78], and *Google Drive* [70].

  The specific objectives to achieve the development of the proposed intelligent solution are divided into two groups: research and development.

- Research:
  - Literature review, preliminary analysis, and compilation of methods based on Deep Learning to detect objects in images.
  - Study of the languages, libraries and frameworks available to work with object recognition models.
  - Search for web services that cover the work needs:
    - Software: support for the chosen language and libraries.
    - Hardware: large RAM and disk space, support for a high-capacity GPU [64] and TPU (*Tensor Processing Unit*) [65], large space for file storage in the cloud.
  - Exploration of open and annotated image datasets to perform detectors training and validation processes.
- Development:
  - Implementation of a program that automates the generation, analysis and export of datasets with which to train the models.

- Construction of a program to train and validate the models by providing graphical and numerical metrics. The models may be exported.
- Creation of a portable program to test the true performance of the models on pc’s, laptops, and microcomputers.
- Modification of *scripts* and libraries to adapt their functionality to the needs of the project.

2.4 Work plan

In order to achieve the objectives described in the previous section, the activities listed below were carried out, however, it should be emphasized that they did not always follow this strict order, at least during the technical activities (3-5). This was due to difficulties related to technological limitations, and to the inexperience with the technologies used, which resulted in several situations of revision, changes and intercalation of tasks.

Activities planned and carried out:

1. Establishment of objectives and work deadlines with the tutor.
2. Study of the technologies, and sources of open and annotated images available for the development of the project and the processes of the models.
3. Development of an automatic dataset generator for training and validation of object recognition models.
4. Development of a program for training, validation and export of object recognition models.
5. Development of a portable application to evaluate the true performance of models in object classification, location and segmentation.
6. Writing and organization of the memory.

2.5 Memory organization

The sequence of chapters that compose this memory is as follows:

1. Introduction: previously exposed, where real examples of the application of models for object recognition in today’s society are discussed, which provides an overview of their importance, and offers, through reflection on their

applications, sufficient reasons to focus on them. The objectives of this work and the activities that have been necessary to achieve them are also presented.

2. Conceptual framework: chapters which are the key of theoretical aspects of the different neural network models which have been designed to detect objects in images. Exactly, the models studied are: SSD, R-CNN, FAST R-CNN, FASTER R-CNN, MASK R-CNN, and YOLO.
3. Technical framework: each of the technologies used in the development of the work is detailed and justified individually, as well as the conceptual design that unifies all these chosen technologies, which gives an overview of the framework created to provide users with the complete cycle of activities necessary to develop these models.
4. Results: once the theoretical and technological aspects of the work have been understood, the positive and negative aspects of the developments and the results obtained from training, validating, and testing the models that are available with the technology used, are discussed.
5. Conclusions and future work: this chapter presents the conclusions of the work carried out, as well as a series of proposals for improving the results obtained and refine those technical aspects that have been more deficient.

Capítulo 3 - Redes Neuronales

3.1 Introducción a las redes neuronales

Este capítulo comienza con la introducción del concepto de Red Neuronal Artificial [4, 19] a partir de una breve introducción al concepto de Red Neuronal Biológica, y continúa con la descripción de un modelo basado en la retropropagación del error (BPE; *Backpropagation Error*) [21], cuya unidad más básica es el Perceptrón [18]. Se estudian los aspectos más relevantes, junto con los parámetros y mecanismos relacionados con el proceso de aprendizaje indicado.

3.2 Redes neuronales biológicas

Una neurona es un tipo de célula que constituye la unidad más básica del sistema nervioso, estas se disponen de forma que componen una red por donde circulan los impulsos nerviosos, que se pueden transmitir química o eléctricamente. Con ellos, las neuronas comunican a otras células estímulos internos o externos al cuerpo. El impulso nervioso llega a través de las neuronas al sistema nervioso central, donde se elabora una respuesta y se envía una señal hacia el órgano diana.

Las partes de una neurona biológica se enuncian a continuación, y pueden verse representadas gráficamente en la figura 3.1:

- Soma o cuerpo celular: contiene el núcleo de la célula, proporciona la energía necesaria para mantener y realizar sus funciones vitales.
- Dendritas: prolongaciones cortas y ramificadas del soma, son receptores de impulsos nerviosos de otras células.
- Axón: prolongación alargada del soma, se encarga de llevar la información desde este hacia otra neurona o célula blanco, las cuales pueden estar en un órgano, glándula o músculo.

Los impulsos nerviosos viajan siempre en el mismo sentido, llegan a una neurona a través de las dendritas, que captan las transmisiones de una neurona anterior. Las señales recibidas son excitatorias, inhibidoras o moduladoras, dependiendo de si incrementan, disminuyen o regulan la producción de un potencial de acción.

Si el cúmulo de señales excitatorias e inhibidoras de la entrada en la neurona es suficiente como para generar un potencial de acción, una señal se transmitirá a través del axón para llegar a otras neuronas, y así, hasta alcanzar finalmente al sistema nervioso central, donde se elabora la respuesta correspondiente.

Cabe recordar que Santiago Ramón y Cajal recibió el Premio Nobel de Medicina en 1906, junto con Camillo Golgi, por su trabajo sobre la estructura y morfología del sistema nervioso [15]. Pues descubrió que el tejido cerebral está compuesto por células individuales, llamadas neuronas, que forman una red.

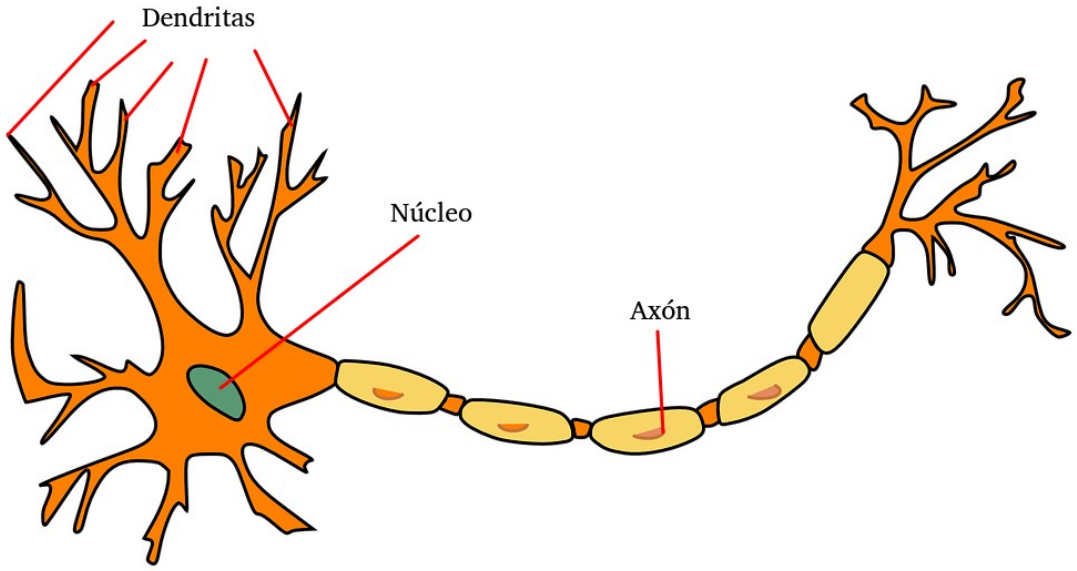

Figura 3.1: Partes de una neurona biológica. Adaptada de: Pixabay [81].

3.3 Redes neuronales artificiales

Las Redes Neuronales Artificiales [4, 19] imitan la estructura del sistema nervioso para tratar de construir sistemas de procesamiento paralelos, distribuidos, y adaptativos, que pueden presentar cierto comportamiento inteligente [16].

Estas redes, a su vez, se descomponen internamente en multitud de elementos más simples denominados neuronas, interconectadas de forma más o menos densa, y cuyo funcionamiento, en conjunto, da lugar a un procesamiento no lineal complejo [17].

Los elementos clave de una neurona artificial son los siguientes, y pueden verse gráficamente representados en la figura 3.2.

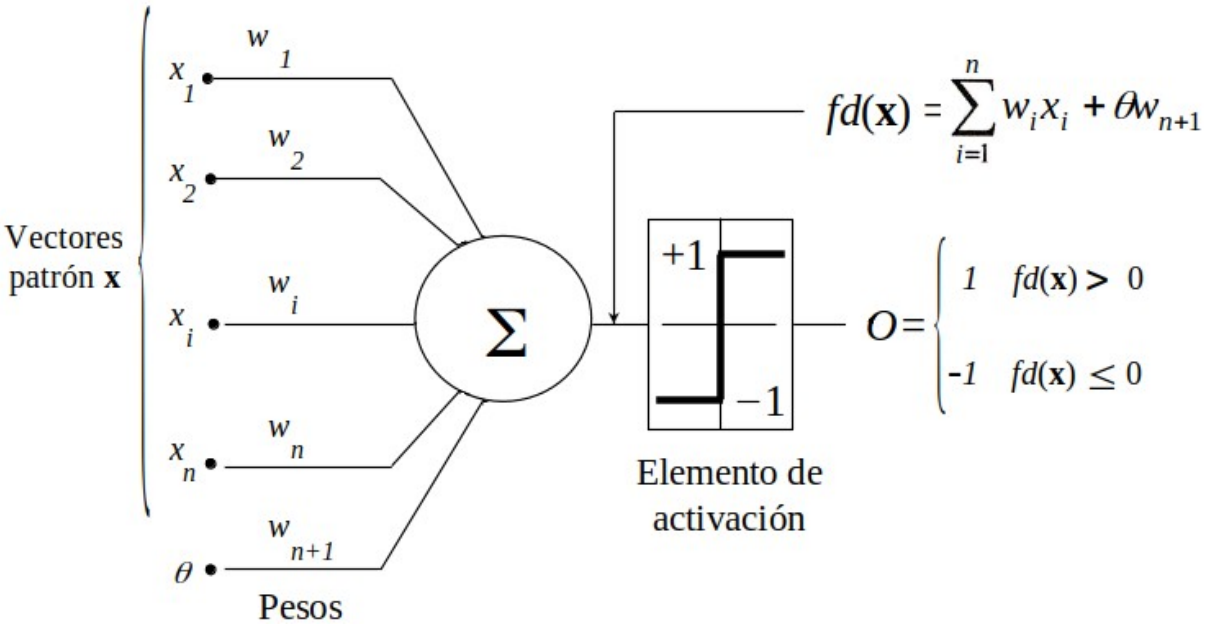

Figura 3.2: Elementos de una neurona artificial. Fuente: Aprendizaje Profundo [3].

- Conexiones sinápticas: cada neurona ***j*** de una capa de red está conectada a un número ***i*** de neuronas de una capa anterior, que actúan como sus entradas. Sus valores emitidos forman un vector de x_i características, que tienen asociadas un factor w_{ij}, que indica la importancia, o peso, que se le concede al valor x_i emitido por la neurona ***i*** a la neurona ***j***.
- Función potencial de entrada: una neurona se activa cuando la suma de sus señales de entrada x_i, por sus correspondientes pesos w_{ij}, superan un cierto valor umbral. Son proclives a activarse si sus pesos w_{ij} son positivos, o de carácter excitatorio, y el valor del parámetro adicional θ_j, llamado *bias*, o sesgo, es bajo. Este potencial de entrada se calcula según la ecuación 3.1, donde la suma de los potenciales está complementada por el factor de sesgo.

$$y = \sum_{i=1}^{n} \left(x_i * w_{ij} \right) - \theta_j \qquad (3.1)$$

- Función de activación: una neurona solo puede encontrarse en uno de dos estados posibles, excitación o reposo, lo cual siempre depende de la suma de los valores que reciba como entrada, y la función de activación que tenga asociada, la cual cumple dos objetivos:

- Escoger el tipo de salida más adecuada para la neurona dependiendo de los intereses, haciendo que tenga un carácter binario, tomando los valores discretos {0,1} o {-1,1}, o bien, que tenga un carácter más analógico-probabilista, emitiendo valores continuos en el rango [0,1] o [-1,1]. Las ecuaciones 3.2 – 3.6 muestran algunas de las funciones más empleadas:

$$Step: f(x) = sign(x);\ f(x) \in [0,1] \text{ o } f(x) \in [-1,1] \tag{3.2}$$

$$ReLU: f(x) = max(0,x);\ f(x) \in [0,x] \tag{3.3}$$

$$Leaky\ ReLU: f(x) = max(0.001 * x, x);\ f(x) \in (-1,x] \tag{3.4}$$

$$Sigmoid: f(a,c,x) = \frac{1}{1+e^{-a*(x-c)}};\ f(a,c,x) \in [0,1] \tag{3.5}$$

$$tanh: \tanh(x) = 2 * sigmoid(2 * x) - 1;\ \tanh(x) \in [-1,1] \tag{3.6}$$

- Evitan el colapso de una red neuronal compleja formada por varias capas de neuronas. Una modificación no lineal de las salidas evita que la red no se vea reducida a una única neurona que surge como el resultado de combinar las funciones lineales del conjunto de ellas, algo que sería desastroso, ya que con una única neurona no pueden realizarse las operaciones necesarias como para abordar problemas de clasificación multiclase no lineales.

Una neurona que presenta estas características recibe el nombre de Perceptrón [18], y la función que realiza a través de sus operaciones consiste en aplicar sobre las características de entrada, alguna función que las separe linealmente, como por ejemplo las de las puertas lógicas AND, OR, NOT o NAND, pero no XOR, ya que esta última requeriría más de una recta para poder realizarse.

3.3.1 Arquitectura de una red neuronal artificial

Cuando se tiene un problema de clasificación multiclase y no lineal, se necesita una arquitectura compleja compuesta por varias capas neuronales conectadas, estas conforman un Perceptrón Multicapa (MLP; *Multi-Layer Perceptron*) [17], un tipo de modelo de red con conexiones sólo hacia adelante, o *feed-forward network*. En la figura 3.3 puede verse su estructura.

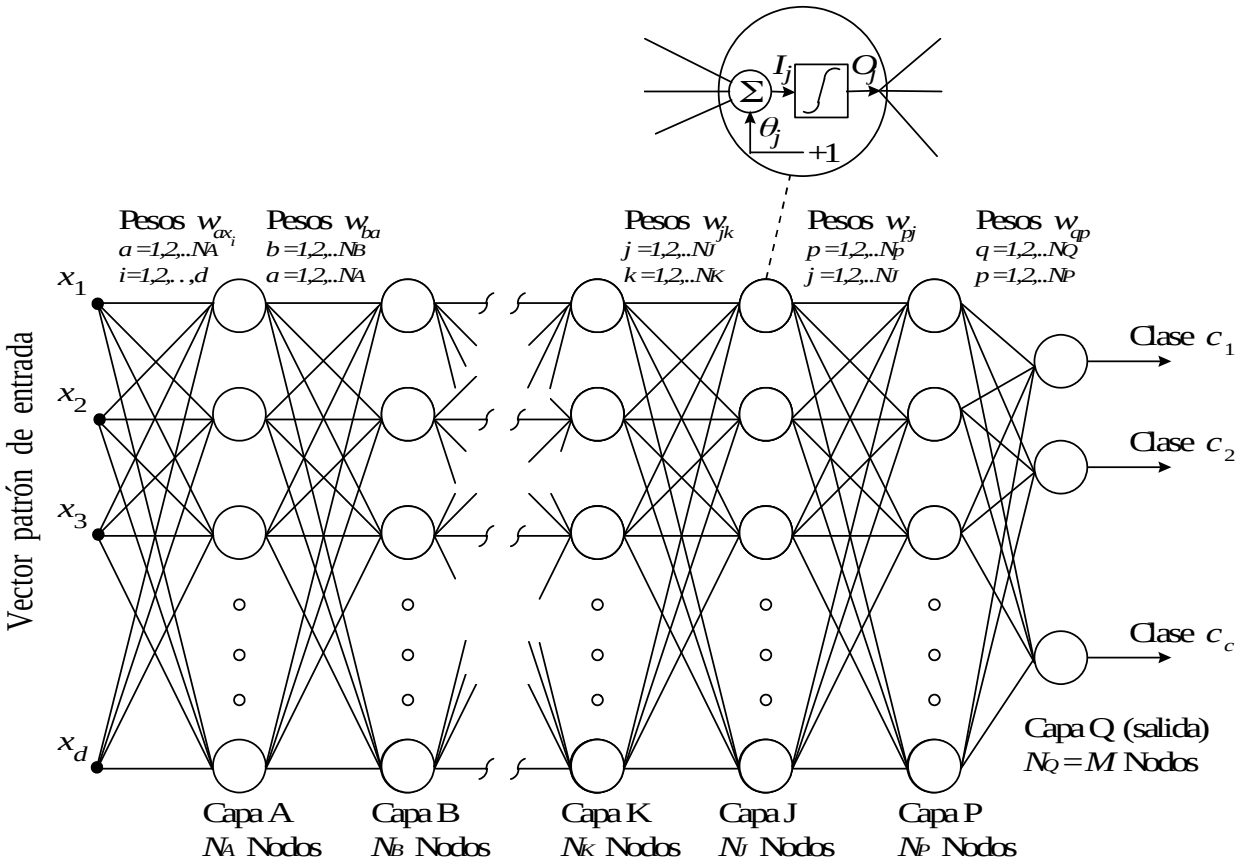

Figura 3.3: Perceptrón multicapa. Fuente: Aprendizaje Profundo [3].

Una descripción más formal de este tipo de arquitecturas de red recurre a la teoría de grafos [20], donde las neuronas son nodos conectados que forman un grafo dirigido, cuyas propiedades son:

1. Cada nodo ***j*** tiene asociada una variable de estado x_j.
2. A cada conexión (***i***,***j***) de los nodos se le asocia un peso $w_{ij} \in R$.
3. A cada nodo o neurona ***j*** se le asocia un sesgo o valor umbral θ_j.
4. Para cada nodo ***j*** se define una función $f_j\left(x_i, w_{ij}, \theta_j\right)$ que depende de los estados de los nodos ***i*** conectados a él, los pesos de sus conexiones, y el umbral, o sesgo, que tenga.

Estas redes están formadas por tres tipos de capas:

- Capa de entrada: compuesta por neuronas que representan la entrada de datos de la red, hay tantas como valores entrantes tenga el modelo.

- Capas ocultas: capas intermedias entre la entrada y la salida. Sus neuronas analizan la entrada de la red, las primeras extraen de esta las características de más bajo nivel, mientras que las últimas las de más alto nivel.
- Capa de salida: contiene tantas neuronas como categorías tiene un problema de clasificación multiclase, devuelve una predicción que indica la categoría a la que cree que pertenece la entrada que le es pasada al modelo.

3.3.2 Tipos de aprendizaje

En lo que respecta al ámbito de este trabajo, el aprendizaje de los modelos se lleva a cabo mediante lo que se conoce como aprendizaje supervisado, sin embargo, a continuación se introducen los cuatro tipos de aprendizaje más comunes entre los modelos de Red Neuronal Artificial [16], cuya finalidad es dar a conocer brevemente otros planteamientos y estrategias que intervienen en los modelos de Aprendizaje Profundo (*Deep Learning*) [3] más allá de su modelo de red neuronal más emblemático, el Perceptrón Multicapa (MLP; *Multi-Layer Perceptron*) [17]. Estas formas de aprendizaje son:

- Aprendizaje supervisado: consiste en estimar una cierta función multivariable f: $\mathbb{R}^n \rightarrow \mathbb{R}^m$, a partir de muestras ($\boldsymbol{x} \in \mathbb{R}^n$, $\boldsymbol{y} \in \mathbb{R}^m$), con $\boldsymbol{x}$ entrada, e $\boldsymbol{y}$ salida, que le permitan relacionar unas con otras. Para ello se realiza un proceso de carácter iterativo, durante el que se minimiza el valor de una función E[$\boldsymbol{W}$], que representa el error ($\boldsymbol{E}$) cometido por la red, en función de sus pesos ($\boldsymbol{W}$). Esto se logra mediante la aplicación de una aproximación estocástica. Ejemplos de red que realizan este tipo de aprendizaje, son el Perceptrón Multicapa, y las Redes Neuronales Convolucionales (CNN's; *Convolutional Neural Networks*) [22, 23].
- Aprendizaje no supervisado: en este tipo de aprendizaje la red recibe una serie de muestras de entrada, pero sin sus respectivas salidas asociadas. En este caso, lo que hace es estimar una función de densidad $f(\mathbf{x})$, que describa la distribución de patrones pertenecientes a $\mathbb{R}^n$, el espacio de la entrada, a partir de las muestras. Una vez estimada dicha función, el modelo ya puede extraer rasgos, o agrupar patrones en función de su similitud (*clustering*). Ejemplos de red que se basan en esta forma de aprendizaje son, el Mapa de Red Autoorganizado (SOM; *Self Organizing Map*), y el Neocognitrón.

- Aprendizaje híbrido: se da cuando coexisten en un mismo modelo de red, en distintas capas neuronales, el aprendizaje supervisado y el no supervisado. Ejemplos de redes que emplean este tipo de aprendizaje son la Red de Base Radial (RBF; *Radial Basis Function Network*), y la Red de Contrapropagación.
- Aprendizaje por refuerzo: es una forma de aprendizaje a medio camino entre el aprendizaje supervisado y el no supervisado. Al igual que en el primero de estos, se emplea la información del error cometido al clasificar, sin embargo, solo existe una única señal de error para la red, la cual representa de forma global, lo bien o lo mal que ésta lo está haciendo cuando clasifica. Por otra parte, al igual que en el aprendizaje no supervisado, la red no tiene ninguna información sobre qué salidas se corresponden con las entradas que le son pasadas. Un ejemplo de modelo que emplea este tipo de aprendizaje es la Red Profunda Q-Network (DQN; *Deep Q-Network*).

3.3.3 Método de aprendizaje

Los modelos de red neuronal pueden modificar sus parámetros durante lo que se conoce como proceso de entrenamiento, para ajustar su funcionamiento ante un problema de clasificación. Normalmente se modifican los pesos de las conexiones neuronales, inicialmente fijados de forma aleatoria, y se considera que ha habido aprendizaje si los valores de estos parámetros se estabilizan a lo largo del entrenamiento.

El método de aprendizaje que atañe a los modelos de red que son de interés tanto para esta introducción, como para el trabajo en sí, es conocido como Corrección del Error, y se basa en el uso de dos algoritmos fundamentales.

Estos son, el Descenso del Gradiente (GD; *Gradient Descent*), y la Retropropagación del Error (BPE; *Backpropagation Error*) [21], teniendo en cuenta que el primero de estos se apoya en el segundo. Estos algoritmos son ejecutados cada vez que la red termina de procesar una entrada de datos, pues es en ese preciso momento cuando se dispone de la información necesaria para realizar los ajustes del modelo.

3.3.3.1 Corrección del Error

El error cometido por una red es la diferencia entre predicción generada por esta, y el resultado que debería haber emitido. Para una neurona ***j*** de la red, su error ***e***, en el instante ***k***, se define como la diferencia entre su salida deseada ***d***, en ***k***, y su salida generada ***g***, en ***k***, tal y como en la ecuación (3.7) [17].

$$e_j(k) = d_j(k) - g_j(k) \tag{3.7}$$

Para poder calcular el error cometido por las neuronas de la capa de salida, se define una función denominada, de error (*error*), coste (*cost*), o pérdida (*loss*), que mide lo próximas que están las predicciones del modelo a los resultados que hubieran sido deseables que emitiera. En este punto, cabe destacar que hay dos tipos de funciones para realizar esta tarea, aquellas orientadas a problemas de clasificación, cuya salida es categórica, o discreta, y las orientadas a problemas de regresión, de salida contínua [3].

La función que más se utiliza en problemas de clasificación es la entropía cruzada, o *cross entropy*, que sirve para averiguar lo bien que se desempeña un modelo. Esta función se define como podemos ver en la ecuación (3.8), donde $\boldsymbol{p_{i,c}}$ es una función de distribución que dada una muestra de entrada ***i***, devuelve la probabilidad real de que esta pertenezca a una clase ***c***, del problema de clasificación, mientras que $\boldsymbol{q_{i,c}}$ es una función de distribución de probabilidad que es predicha por el modelo, esta función, dada una muestra de entrada ***i***, devuelve lo que el modelo estima que la muestra de entrada pertenece a una clase ***c***. La salida de este cálculo devuelve un valor entre 0 y 1 que indica cómo de cerca o lejos está la función de distribución de probabilidad estimada por el modelo, de la función de distribución de probabilidad.

$$H(p,q) = -\sum_{c \in Classes} p_{i,c} * \log q_{i,c} \tag{3.8}$$

Cuando el problema de clasificación es reducido en lo que se refiere al número de clases, por ejemplo, sólo dos clases, entonces se suele usar la fórmula de la entropía cruzada binaria, o *binary cross entropy*, que se describe a través de la ecuación (3.9), donde en este caso, $\boldsymbol{p_i}$ es un indicador binario (0 “o” 1), que denota la clase a la que pertenece la muestra ***i***, mientras que $\boldsymbol{q_i}$ devuelve la probabilidad predicha por el modelo para la muestra de entrada ***i***.

$$H(p,q) = (-1) * \left[p_i * \log(q_i) + (1-p_i) * \log(1-q_i) \right] \quad (3.9)$$

La función más común ante problemas de regresión que se aplica en los detectores de objetos, es la función Error Cuadrático Medio, o *Mean Square Error*, definida como podemos ver en la ecuación (3.10), donde ***n*** es la cantidad total de muestras de entrada, y $\boldsymbol{e_j}$ el error cometido por cierta neurona ***j*** en una capa del modelo, calculado como se indicó en la ecuación (3.7).

$$MSE = \frac{1}{n} * \sum_{j=1}^{n} e_j^2 \quad (3.10)$$

Una variante del Error Cuadrático Medio, es la descrita en la ecuación (3.11), que es de naturaleza similar a la ecuación original, donde se cambia el cálculo de la media por el factor ½, lo que se hace a conveniencia para facilitar el cálculo de laderivada. Esta función de error recibe el nombre de Error Cuadrático Instantáneo o *Instantaneous Square Error*.

$$ISE = \frac{1}{2} * \sum_{j=1}^{n} e_j^2 \quad (3.11)$$

Otra función que también es empleada en el ámbito de los problemas de regresión es *Log-Cosh Loss*, que es más suave que MSE, y se define según la ecuación (3.12).

$$LCL = \sum_{j=1}^{n} \log\left(\cos h\left(e_j\right)\right) \quad (3.12)$$

3.3.3.2 Optimización: Gradiente Descendente

Una vez se ha definido la función de error más conveniente para el modelo de red, esta puede utilizarse para plantear un problema de optimización que sirva para determinar cómo variar los valores de los pesos de las conexiones neuronales, con el objetivo de reducir el error que comete la red al clasificar o estimar los valores de los parámetros de una regresión.

Si recordamos la optimización matemática de funciones, ya sea para maximizar o minimizar el valor de una función, se recurre al concepto de derivada, la cual se define en la ecuación (3.13). Este concepto indica cuál es el valor de la pendiente, o inclinación de una recta tangente a dicha función, en un punto concreto $\boldsymbol{x_0}$.

$$f'(x_o) = \lim_{h \to 0} \frac{f(x_o+h) - f(x_o)}{h} \text{ o } f'(x_o) = \lim_{x \to x_o} \frac{f(x) - f(x_o)}{x - x_o} \qquad (3.13)$$

Este concepto lleva asociado consigo un teorema normalmente conocido el “criterio de la primera derivada”, que indica lo que simboliza el signo de este cálculo:

- Si $f'(x_o) > 0$ entonces la función es creciente en x_o.
- Si $f'(x_o) < 0$ entonces la función es decreciente en x_o.
- Si $f'(x_o) = 0$ entonces la función no crece ni decrece en x_o, pudiendo ser un mínimo, un máximo, o un punto de inflexión.

Por lo que este criterio permitiría, dado un punto cualquiera en el que una función esté definida, averiguar la dirección en la que desplazase para acabar en un mínimo o máximo. La figura 3.4 representa un ejemplo ilustrativo de esto, donde puede verse dibujada en azul una representación de la función $f(\mathbf{x}) = ½ * \mathbf{x}^2$, y en rojo la representación de la ecuación de la recta tangente a los puntos (0, 0) y (2, 2), que representa el crecimiento entre estos dos puntos de la función.

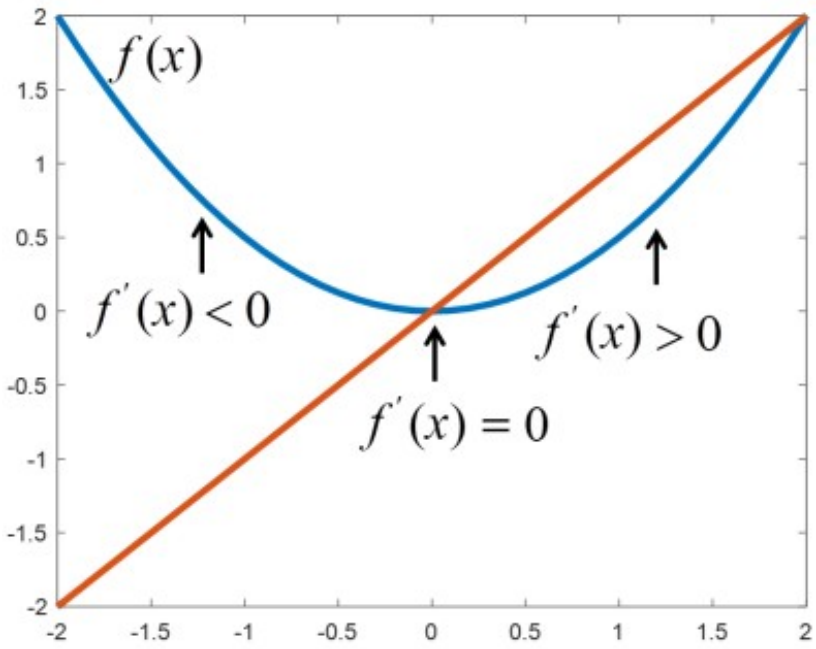

Figura 3.4: Ejemplo de uso del criterio la 1ª derivada. Fuente: Aprendizaje Profundo [3].

Pese a la utilidad de esta herramienta matemática, recordemos que en una función existen dos conceptos de mínimo y máximo, los locales y los globales. Un mínimo o máximo local de una función es un punto donde el valor de la función es menor, o mayor, que el de sus puntos vecinos, mientras que un mínimo o máximo global es el punto con el valor más bajo, o alto, de una función. Por otra parte, existen los puntos de inflexión, que son aquellos puntos con derivada nula rodeados por puntos vecinales que crecen y otros que decrecen, pero que no son ni máximos ni mínimos.

Esto provoca la aparición de una complicación de cara al problema de optimización de la función de error, que es la dificultad para hacerla converger al mínimo global, sobre todo en modelos de *Deep Learning*, que no son tan simples como el concepto que representa la función del ejemplo anterior, sino que trabajan con funciones con una cierta complejidad a la hora de optimizar (funciones multivariable) que podrían tener muchos mínimos locales no óptimos, así como muchos puntos de inflexión rodeados de regiones muy planas.

En figura 3.5 puede verse un breve resumen que representa esta problemática de la optimización de funciones.

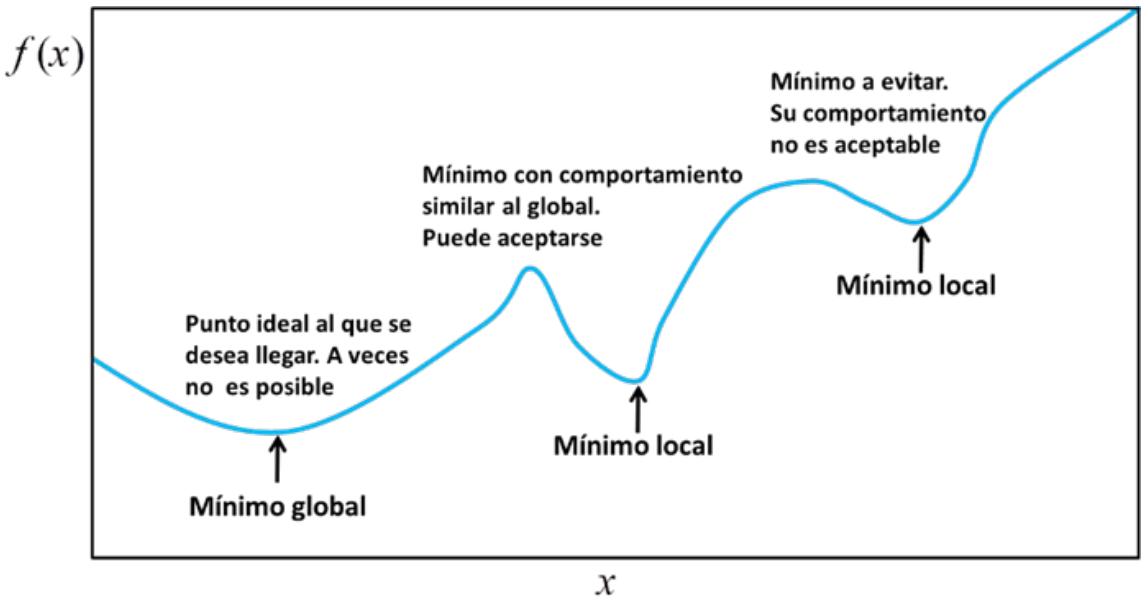

Figura 3.5: Optimización con múltiples puntos críticos. Fuente: Aprendizaje Profundo [3].

Por las razones anteriores, es por lo que, por norma general, es común conformarse con encontrar algún valor que sea muy bajo, pero no necesariamente el mínimo en ningún sentido formal [3].

Otra dificultad relativa al uso de la derivada para la corrección de los pesos por medio de la optimización de la función de error, es que esta debe ser diferenciable. Para ello, es necesario que todas las funciones de las capas de la red neuronal, sean diferenciables (derivables), ya que entonces la función de error también lo será, por lo que podrá utilizarse para corregir el error cometido en cada capa. Por desgracia, algunas de las funciones de activación, como por ejemplo la *ReLU*, que se define posteriormente, provocarán que los cálculos no sean diferenciables, lo que implica tener que usar una aproximación derivativa para poder utilizar ciertas funciones con el algoritmo de *retropropagación* del error [3].

Pese a las complicaciones anteriores, el método del gradiente descendente sigue siendo uno de los métodos más utilizados en Aprendizaje Automático para estimar coeficientes, ya que computacionalmente es poco costoso al regular de forma iterativa en cada paso del entrenamiento (*step*) los valores de los pesos, logrando así disminuir el valor de la función de error hasta hacerla converger en un valor mínimo, o lo más pequeño posible. En general el algoritmo da buenos resultados.

Llegados a este punto, cabe a hacerse un inciso, y es que hasta ahora sólo se ha hablado de funciones de una sola variable, y el uso de la derivada para poder optimizarlas, sin embargo, esto no es correcto, ya que la complejidad inherente a los modelos de *Deep Learning*, como por ejemplo en el caso del Perceptrón Multicapa (MLP; *Multi-Layer Perceptron*) [17], requiere trabajar con funciones de tipo multivariable $f : \mathbb{R}^n \rightarrow \mathbb{R}$.

En el caso de estas funciones, en lugar de calcularse la derivada, la cual devuelve un valor escalar que representa el valor de la pendiente en un punto, se recurre al concepto de las derivadas parciales, las cuales permiten averiguar cómo cambia una cierta función f, según aumenta el valor de una variable $\boldsymbol{x_i}$, en un punto $\boldsymbol{x}$. El cálculo de estas derivadas parciales permite construir un vector que contiene en cada posición el resultado de cada una de ellas, este se denomina vector gradiente, y generaliza el cálculo de la derivada para las funciones de tipo multivariable. En la ecuación (3.14), se puede observar la notación relativa al cálculo de este vector.

$$\nabla_x f(x) = \left(\frac{\delta f(x)}{\delta x_1}, \frac{\delta f(x)}{\delta x_2}, \ldots, \frac{\delta f(x)}{\delta x_{n-1}}, \frac{\delta f(x)}{\delta x_n} \right) \tag{3.14}$$

Algunas propiedades interesantes del vector $\nabla_x f(\mathbf{x})$, dado un punto $\boldsymbol{x}$, son:

1. Señala la dirección en que moverse para que f aumente.
2. Indica la pendiente de la función f en dicho punto.
3. Es perpendicular a las curvas de nivel de f.

La primera propiedad es la que más interesa de cara al estudio que aquí se plantea, ya que, como gracias a ella se sabe hacia donde moverse para que el valor de $f(\mathbf{x})$ se incremente, también podemos saber que realizar el movimiento opuesto en el sentido del descenso del gradiente $-\nabla_x f(\mathbf{x})$, nos moveríamos hacia donde $f(\mathbf{x})$ se decrementa, lo cual permite utilizar estos conceptos para tratar de buscar algún punto crítico aceptable que cumpla $\nabla_x \mathbf{E}(\mathbf{x}) = \mathbf{0}$, minimizando así la función de error.

El procedimiento del uso del algoritmo del gradiente descendente es como sigue:

1. Inicialización de los valores de los pesos $\boldsymbol{w_{ij}}$, pudiendo ser de forma aleatoria, y establecimiento del valor del factor de aprendizaje $\boldsymbol{\varepsilon}$, o *learning rate*.
2. Calcular el gradiente de la función de error como en la ecuación (3.15).

$$\nabla_x \; E(w) = \left(\frac{\delta E(w)}{\delta w_1}, \frac{\delta E(w)}{\delta w_2}, \; \ldots \; , \frac{\delta E(x)}{\delta w_{n-1}}, \frac{\delta E(x)}{\delta w_n} \right) \qquad (3.15)$$

3. Actualizar el valor de los pesos aplicando la ecuación algebraica (3.16), donde $\boldsymbol{t}$ representa la iteración actual, y $\boldsymbol{\varepsilon}$ la tasa de aprendizaje.

$$w(t+1) = w(t) \quad \varepsilon \; * \; \nabla_x E(x) \qquad (3.16)$$

4. Repetir el primer paso hasta que finalicen las iteraciones del entrenamiento o el algoritmo converja a una solución.

Una interpretación gráfica de este procedimiento se puede visualizar a través de la siguiente metáfora: inicialmente partimos de un punto aleatorio (inicialización) en un territorio montañoso (hiper-valle formado por los pesos), pero podemos tantear el terreno (vector gradiente) para bajar poco a poco (tasa de aprendizaje) por una pendiente (derivada) que nos lleve a una cuenca del valle (mínimo local o global).

3.3.3.3 Optimización: Descenso Estocástico del Gradiente

El Descenso Estocástico del Gradiente o *Stochastic Gradient Descent* (SGD), es una estrategia de minimización de funciones que tienen una forma similar a la descrita en la ecuación (3.17), donde J es la función de error, w es un valor que debe estimarse para minimizar la función de error, y J_i el valor de la función de error para el ejemplo i de un conjunto de datos de entrenamiento.

$$J(w) = \frac{1}{n}\sum_{i=1}^{n} J_i(w) \tag{3.17}$$

La fórmula que emplea este algoritmo de optimización para minimizar la función de error a través de los pesos es la mostrada en la ecuación (3.18).

$$w(t+1) = w(t) - \varepsilon * \frac{1}{n}\sum_{i=1}^{n} J_i(w(t)) \tag{3.18}$$

En cuanto al concepto estocástico, esto implica la realización de un proceso de forma aleatoria que, en este caso, atañe a la forma en que se realiza el ajuste.

Esta diferencia con el método del descenso del gradiente estándar reside en que mientras que en este se utilizan todos los datos del conjunto de entrenamiento para realizar el ajuste de la red, en el descenso del gradiente estocástico no, siendo su estrategia más común realizar una división aleatoria de los ejemplos del conjunto de entrenamiento en varios subconjuntos de igual tamaño, denominados *mini-batches*, posteriormente en cada iteración del entrenamiento, realizar el ajuste del modelo para cada uno de los *mini-batch*.

El procedimiento para el uso de este algoritmo de optimización es el siguiente:

1. Inicialización de los valores de los pesos w_{ij}, pudiendo ser de forma aleatoria, y establecimiento del valor del factor de aprendizaje ε, o *learning rate*.
2. Aplicar una estrategia para la selección aleatoria de ejemplos del conjunto de entrenamiento.
3. Calcular el gradiente de la función de error como en la ecuación (3.14).
4. Actualizar el valor de los pesos aplicando la ecuación algebraica (3.15), donde t representa la iteración actual, y ε la tasa de aprendizaje.

5. Repetir el primer paso hasta que finalicen las iteraciones del entrenamiento o el algoritmo converja a una solución.

Las ventajas del algoritmo SGD frente al algoritmo GD son:

- Requiere menos memoria y es más rápido, pues procesa un número inferior de ejemplos respecto del total que componen el conjunto de entrenamiento.
- Con un conjunto de entrenamiento muy grande puede converger más rápido, ya que se realizan un mayor número de actualizaciones del modelo de red.
- Durante los pasos hacia los mínimos hay más oscilaciones, lo que dificulta la convergencia en mínimos locales.

3.3.3.4 Optimización: SGD con Momento

Existen diversas variantes del algoritmo SGD, de entre ellas destaca una conocida como Descenso del Gradiente Estocástico con Momento (SGDM; *Stochastic Gradient Descent with Momentum*) [21].

Lo que hace esta variante de SGD es recordar la actualización (gradiente) $\boldsymbol{\Delta w}$ de cada iteración $\boldsymbol{t}$, de forma que se crea la combinación lineal del gradiente actual y el anterior [27]. La actualización de los pesos se puede realizar sustituyendo la ecuación (3.20) en la (3.19), de forma que se obtiene la (3.21). En estas ecuaciones, es la constante $\boldsymbol{\alpha} \in [0,1]$ la que representa el momento, quien controla la velocidad de actualización de $\boldsymbol{\Delta w(t)}$.

$$w(t+1) = w(t) + \Delta w(t) \tag{3.19}$$

$$\Delta w(t) = \alpha * \Delta w(t) - \varepsilon * \frac{1}{n}\sum_{i=1}^{n} \nabla_x J_i(w(t)) \tag{3.20}$$

$$w(t+1) = w(t) + \alpha * \Delta w(t) - \varepsilon * \frac{1}{n}\sum_{i=1}^{n} \nabla_x J_i(w(t)) \tag{3.21}$$

Este concepto de momento proviene de la física, donde el vector de pesos $\boldsymbol{w}$ sería visto como una partícula que recorre el espacio de parámetros, la cual adquiere una aceleración a partir de una fuerza, lo cual la haría tender a mantenerse en la dirección de esta, y reduciría sus posibles oscilaciones [3].

Existen otros métodos conocidos para la optimización de la función de error, como Adam y RMSprop, entre otros, que no se han utilizado en el presente trabajo.

3.3.3.5 Cálculo de derivadas: Retropropagación del Error

La diferencia entre el algoritmo GD, o sus variantes, y el algoritmo BPE, reside en la función que cada uno aplica de cara a la minimización de la función de error. Como se ha descrito anteriormente, GD propone una solución eficaz para el ajuste de los pesos de la red neuronal mediante el cálculo del vector gradiente, sin embargo, no dice nada sobre la forma de calcular todas las derivadas parciales del vector.

Esa es la misión que desempeña BPE. Más precisamente, este algoritmo iterativo implementa de forma eficiente el cálculo de gradientes en estructuras de tipo grafo, como son los modelos de Red Neuronal Artificial [4, 19], mediante la automatización del cálculo de derivadas parciales y de la regla de la cadena, ya que estos modelos están conformados por funciones compuestas y multivariables que han de ajustarse. Es importante matizar esta diferencia entre ambos algoritmos, ya que muchas veces se generaliza la idea de que es BPE el que corrige todo el comportamiento errático de un modelo de red neuronal, lo cual no es correcto, pues es el uso de ambos algoritmos lo que permite conseguir dicho objetivo.

Antes de explicar cómo BPE realiza el cálculo del gradiente, hay que saber que el proceso que realiza se hace en el sentido inverso del funcionamiento de la red, es decir, desde la capa de salida hacia la capa de entrada, pasando por cada una de las capas ocultas, de ahí el nombre de retropropagación. Este funcionamiento tiene sentido por dos razones, siendo la primera, que el error que comete el modelo se calcula a partir de los resultados de su capa de salida, lo que la convierte en la primera capa cuyas neuronas pueden ajustarse en base a su error cometido, y la segunda razón, es que cada neurona de una capa oculta solo es responsable en una cierta parte del error cometido en la salida, por lo que para saber cuánto contribuye a dicho error, se ha de corregir su comportamiento teniendo en cuenta los reajustes de las capas posteriores a donde ella se encuentra. Por eso, esta depuración de las responsabilidades de cada neurona con respecto al error que se comete al final, ha de hacerse hacia atrás.

Una vez comprendido el porqué del recorrido a la inversa de la red neuronal para la corrección de su funcionamiento, el proceso que sigue este algoritmo para calcular los gradientes de cada neurona de la red es el siguiente [17].

Primero se calcula el descenso del gradiente tal y como se ve en la ecuación (3.22), donde α es el factor de aprendizaje, y la derivada representa lo que varía el error e cometido por una neurona con respecto al cambio del valor de sus pesos w_{ij}, y el superíndice L es el identificador de la capa a la que pertenece la neurona, donde a no ser que se indique lo contrario, se considera que es la capa de salida del modelo.

$$\Delta w_{ij} = -\alpha * \left(\frac{\partial e}{\partial w_{ij}^{L}} \right) \tag{3.22}$$

Aplicando la regla de la cadena en la ecuación (3.22), se obtiene la (3.23), donde f_j es la función de activación de la neurona de salida j, y n_j la función que representa la suma promedio de sus entradas complementada por el *bias*.

$$\Delta w_{ij} = -\alpha * \left(\frac{\frac{\partial e}{\partial f_j^{L}} * \partial f_j^{L}}{\partial n_j^{L}} * \partial n_j^{L} \right) \tag{3.23}$$

La primera derivada de la ecuación (3.23) se puede calcular de forma inmediata, ya que se conoce cuál es el valor de salida deseado para ella, por tanto, puede calcularse lo que varía su error, según cambia la salida de su función de activación. Para ello se obtiene la ecuación (3.24), donde la función de error es *Instantaneous Square Error*, d_k la salida deseada por la neurona k, g_k la salida generada por k, y N el total de neuronas de la capa de salida. Se observa que el resultado de esta derivada es el valor negativo del error cometido por la neurona, abreviado como -e_j.

$$\frac{\partial e}{\partial f_j^{L}} = \frac{\frac{1}{2} * \sum_{k=1}^{N} \left(d_k - g_k \right)^2}{f_j^{L}} = -\left(d_j - f_j^{L} \right) = -e_j \tag{3.24}$$

La segunda derivada en la ecuación (3.23) representa la variación de la función de activación de una neurona ***j***, con respecto al cambio de la suma ponderada y su factor *bias* asociado. Se resuelve directamente como se indica en la ecuación (3.25), donde se ve que su resultado es la derivada de la función de activación de ***j***.

$$\frac{\partial f_j^L}{\partial n_j^L} = f'^L_j \tag{3.25}$$

Conviene destacar que por cuestión de simplificación de los cálculos y de notación, las cadenas de derivadas que surgen de la derivada de la función de error respecto de la suma parcial de las entradas y el factor de *bias*, como la calculada con estas dos primeras derivadas, se abrevian como se indica en la ecuación (3.26).

$$\frac{\partial e}{\partial n} = \frac{\frac{\partial e}{\partial f} * \ldots * \partial f}{\partial n} = \delta \tag{3.26}$$

La tercera derivada de la ecuación (3.23) representa la variación de la suma ponderada de los pesos de una neurona ***j***, donde se considera que el *bias* es un peso más, con respecto al cambio de valor de los pesos $\boldsymbol{w_{ij}}$. Se resuelve como se indica en la ecuación (3.27), donde la variable ***N*** representa el número total de conexiones entrantes a la neurona ***j***, y $\boldsymbol{x_i}$ simboliza el valor de cada conexión entrante a ***j***. El resultado de esta ecuación es el vector de entradas ***x***, cuyos elementos son los $\boldsymbol{x_i}$ valores entrantes.

$$\frac{\partial n_j^L}{\partial w_{ij}^L} = \frac{\sum_{i=1}^{N} \left(w_{ij}^L * x_i^L \right)}{\delta w_{ij}^L} = x_i^L \tag{3.27}$$

Una vez especificadas todas las derivadas necesarias para calcular el gradiente de una neurona de la capa de salida (***L***), se juntan todos los resultados obtenidos en la ecuación (3.23), y el cálculo del gradiente queda como en la ecuación (3.28), donde $\boldsymbol{\delta_j^L}$ representa el producto de los resultados de las ecuaciones (3.24) y (3.25).

$$\Delta w_{ij} = \alpha * \delta_j^L * x_i^L \tag{3.28}$$

Por otra parte, cuando la neurona no pertenece a la capa de salida, este cálculo no puede realizarse directamente, ya que como se explicó anteriormente, no se posee el valor de salida deseado para ella. Cuando esto ocurre, lo que se hace es calcular su parte del error a través de las neuronas posteriores a las que está conectada.

Para calcular el gradiente para una neurona ***j*** que pertenece a la capa anterior (***L***-1) a la de salida (***L***), se parte de la ecuación (3.29), que representa la función de error *Instantaneous Square Error*, definido en la ecuación (3.24).

$$ISE = \frac{1}{2} * \sum_{k=1}^{N} \left(d_k - g_k\right)^2 \tag{3.29}$$

Como la salida generada por una neurona cualquiera de la capa de salida $\boldsymbol{g_k}$, no es más que su función de activación $\boldsymbol{f_k}$, que a su vez está compuesta por la función que representa la suma ponderada $\boldsymbol{n_k}$ de su entrada, la ecuación (3.29) se reescribe en base a estas funciones, obteniéndose como resultado la ecuación (3.30).

$$ISE = \frac{1}{2} * \sum_{k=1}^{N} \left(d_k - f_k^L\left(w_{ik}^L * x_i^L\right)\right)^2 \tag{3.30}$$

Como las entradas $\boldsymbol{x_i}$ de cualquier ***k*** son las salidas de las funciones de activación $\boldsymbol{f_j}$ de la capa anterior, y estas a su vez se componen por sus respectivas funciones de suma ponderada $\boldsymbol{n_j}$ complementada por el *bias*, considerado otro peso más, la ecuación (3.30) se reescribe en base a estas funciones, obteniéndose la (3.31).

$$ISE = \frac{1}{2} * \sum_{k=1}^{N} \left(d_k - f_k^L\left(w_{jk}^L * f_j^{L-1}\left(w_{ij}^{L-1} * x_i^{L-1}\right)\right)\right)^2 \tag{3.31}$$

Una vez se ha realizado el recorrido hacia atrás para expresar la función de error en base a las ecuaciones de las neuronas de la capa de red ***L***-1, puede obtenerse la derivada del error cometido por la neurona de la capa oculta respecto de sus pesos, la cual se calcula aplicando la regla de la cadena como en la ecuación (3.32).

$$\Delta w_{ij} = -\alpha * \left(\frac{\frac{\frac{\frac{\delta e}{\delta f_k^L} * \delta f_k^L}{\delta n_k^L} * \delta n_k^L}{\delta f_j^{L-1}} * \delta f_j^{L-1}}{\delta n_j^{L-1}} * \delta n_j^{L-1}}{\delta w_{ij}^{L-1}} \right) \tag{3.32}$$

Observando las derivadas de la ecuación anterior, se pueden apreciar unos hechos importantes, y es que las dos primeras derivadas ya están calculadas bajo $\boldsymbol{\delta_j^L}$, y por otra parte, las derivadas cuarta y quinta se calculan tal como en las ecuaciones (3.25) y (3.27), por lo que lo único que falta por conocer es la tercera derivada.

Afortunadamente, su cálculo es sencillo, ya que la derivada de las sumas ponderas con respecto de las salidas de las funciones de activación de la capa anterior, da como resultado el vector de pesos $\boldsymbol{w_{jk}}$, que representa las ponderaciones existentes entre las neuronas ***j*** de la capa actual, y cualquier neurona ***k*** de la capa siguiente.

En la ecuación (3.33) se sintetiza la forma de llegar a este resultado, en ella, ***N*** representa todas las conexiones de salida de una neurona ***j*** hacia las neuronas ***k***.

$$\frac{\partial n_k^L}{\partial f_j^{L-1}} = \frac{\sum_{k=1}^{N} w_{jk}^L * x_i^L}{\partial f_j^{L-1}} = \frac{\sum_{i=1}^{N} w_{jk}^L * f_j^{L-1}}{\partial f_j^{L-1}} = w_{jk}^L \tag{3.33}$$

Para expresar el resultado simplificado del cálculo del gradiente, se ha de calcular $\boldsymbol{\delta_j^{L-1}}$, que si recordamos lo dicho en (3.26), lo forman las cuatro primeras derivadas de las cinco consecutivas que aparecen en (3.32). De estas derivadas, como se dijo anteriormente, las dos primeras ya están calculadas, son $\boldsymbol{\delta_j^L}$, la tercera es el vector $\mathbf{w^L_{jk}}$ de la ecuación (3.33), y la cuarta se calcula según la (3.24), por lo que si estos términos se juntan, conforman la ecuación (3.34).

$$\delta_j^{L-1} = \sum_{k=1}^{N} \left(\delta_k^L * w_{jk}^L \right) * f'^{L-1}_j \tag{3.34}$$

Si se reflexiona un poco sobre la ecuación anterior, puede verse que esta no solo proporciona el resultado de $\boldsymbol{\delta_j^{L-1}}$, sino que permite extraer una lógica concreta para calcular cualquier $\boldsymbol{\delta_j^{L-\beta}}$, para $\boldsymbol{\beta} > 0$. Esta lógica queda reflejada en la ecuación (3.35), y pese a que pueda sugerir un cálculo recursivo, es iterativo, ya que según se aumenta el valor de $\boldsymbol{\beta}$ desde 1 hasta $\boldsymbol{L}$-1, se puede apreciar al desglosar cada $\boldsymbol{\delta_j^{L-\beta}}$, que ya se disponían de varias derivadas calculadas, lo que disminuye el número de operaciones a realizar.

$$\delta_j^{L-\beta} = \sum_{k=1}^{N} \left(\delta_k^{L-\beta-1} * w_{jk}^{L-\beta-1} \right) * f'^{L-\beta}_j \tag{3.35}$$

Finalmente, se concluye que el gradiente puede calcularse como se indica en la ecuación (3.36), donde $\boldsymbol{\delta_j^{L-\beta}}$ se calcula como el producto de las ecuaciones (3.24) y (3.25), si $\boldsymbol{\beta} = 0$, y como indica la regla dada en la ecuación (3.34), si $\boldsymbol{\beta} > 0$.

$$\Delta w_{ij} = \alpha * \delta^{L-\beta} * x_i^{L-\beta} \tag{3.36}$$

Cabe destacar que al cálculo de $\boldsymbol{\delta}$ se le suele denominar como cálculo de la señal de error, ya que este es proporcional al error de salida [16].

3.3.4 Entrenamiento, validación, e hiperparmámetros

Ahora que se conoce el método empleado con carácter general por las redes neuronales para aprender, se ha de entender el entrenamiento como un proceso que recurre a este método para tratar de conseguir que un modelo generalice una serie de conceptos, este proceso, a su vez, va acompañado de otro denominado, de validación, que se realiza para verificar si realmente ha habido aprendizaje por parte del modelo.

Antes de comenzar con estos procesos, es necesario realizar un par de actividades:

1. Establecer un conjunto de ejemplos para el proceso de aprendizaje:
    - Las redes neuronales requieren grandes cantidades de ejemplos para que puedan aprender a generalizar los conceptos presentes en los problemas de clasificación, aunque algunas veces podrán ser suficientes cientos, o miles, la gran mayoría de las veces serán necesarios muchos más.

- El conjunto de ejemplos para el aprendizaje se ha de subdividir a su vez en dos conjuntos, denominados, conjunto de entrenamiento, y conjunto de validación, donde el reparto de ejemplos por cada clase del problema suele seguir una regla 80% - 20%, 85% - 15%, o similar, pero siempre predominando la cantidad de ejemplos que van destinados al conjunto de entrenamiento. La utilidad de cada uno de estos conjuntos es:
  - Entrenamiento: los ejemplos de este conjunto son utilizados para que un modelo de red aprenda a generalizar una serie de conceptos.
  - Validación: los ejemplos de este conjunto sirven para verificar si lo que el modelo ha aprendido con los ejemplos del entrenamiento le ha servido para identificar ejemplos similares.

2. Configurar los hiperparámetros, aquellas variables que han ser ajustadas de forma manual por un usuario, ya que, a diferencia de los parámetros, el modelo no los inferirá automáticamente a partir de las entradas, como pasa por ejemplo con los pesos y los sesgos. Algunos de los hiperparámetros más comunes son los siguientes: factor de aprendizaje (*learning rate*), tamaño del lote (*batch size*), cantidad de ejecuciones del entrenamiento y validación (*steps, epochs*, *iterations*), el factor de fuerza (*momentum*) en SGDM [21], etc.

3.3.4.1 Proceso de entrenamiento

Una vez realizadas las dos actividades anteriores, se pueden realizar los procesos de entrenamiento y validación, donde el primero de estos procesos posee hasta 3 formas diferentes de realizarse:

- Aprendizaje en serie u *on-line* [16]:
  1. Inicialización aleatoria de los pesos y sesgos de la red.
     a. Generalmente se usan valores pequeños, positivos y negativos.
     b. No inicializar todo a cero, ya que el aprendizaje no podrá progresar al ser tanto las salidas, como las actualizaciones de los pesos, todo ceros.
  2. Dado un elemento del conjunto de ejemplos, a partir del cual se quiere que el modelo aprenda, introducirlo en la red para obtener su respuesta.

3. Calcular los gradientes mediante BPE [21].
4. Actualizar los pesos y sesgos con *Gradient Descent*.
5. Calcular el error cometido por el modelo.
6. Repetir el proceso desde el paso número 2 si el error no se ha reducido lo suficiente, o si aún quedan ciclos de entrenamiento.

- Aprendizaje por lotes o *batch* [16]:
  1. Este paso se realiza del mismo modo que en el aprendizaje en serie.
  2. Para cada ejemplo del conjunto de aprendizaje:
     a. Introducirlo en la red para obtener su respuesta.
     b. Calcular los gradientes parciales con BPE [21].
  3. Calcular los gradientes finales.
  4. Actualizar los pesos y sesgos con *Gradient Descent*, o una variante.
  5. Calcular el error cometido por el modelo.
  6. Repetir el proceso desde el punto número 2 si el error no se ha reducido lo suficiente, o si aún quedan ciclos de entrenamiento.
- Aprendizaje por mini lotes o *mini-batches* :
  1. Este paso se realiza del mismo modo que en el aprendizaje en serie.
  2. Para cada mini lote:
     a. Para cada ejemplo del mini lote:
        - Introducirlo en la red para obtener su respuesta.
        - Calcular los gradientes parciales mediante BPE.
     b. Calcular los gradientes finales.
     c. Actualizar los pesos y sesgos con *Gradient Descent*, o una variante.
  3. Calcular el error cometido por el modelo.
  4. Repetir el proceso desde 2 si el error no se ha reducido lo suficiente, o si aún quedan ciclos de entrenamiento.

3.3.4.2 Proceso de validación

Este proceso es mucho más sencillo en comparación con el de entrenamiento, ya que consiste en pasar uno o varios ejemplos al modelo, para ver la diferencia que existe entre los resultados generados por las clasificaciones de cada conjunto y lo esperado, o lo que se denomina, validación cruzada. Este proceso cuenta con dos versiones distintas de realizarse:

1. Validar el modelo cuando acaba el quinto paso del entrenamiento, por lo que ambos procesos se ejecutan uno tras otro continuamente, un enfoque que permite evaluar progresivamente el modelo y comparar sus resultados a la vez. Suele ser la estrategia de validación habitual.
2. Validar el modelo cuando finalice el entrenamiento. Requiere guardar todo el estado del modelo en determinados instantes, para así posteriormente poder cargarlo y evaluarlo. Esta última versión consume más tiempo y recursos, ya que ambos procesos se ejecutan por separado, y en el caso de la validación se ha de realizar la carga y descarga en memoria de cada estado por el que el modelo pasó. Otra de las problemáticas de esta versión es que no pueden compararse directamente los resultados del entrenamiento y la validación.

3.3.4.3 Hiperparámetros: Learning Rate

Es el hiperparámetro que controla la velocidad de aprendizaje, definido en las ecuaciones (3.16) y (3.18), como $\boldsymbol{\varepsilon}$, y en la (3.22) y (3.23), como $\boldsymbol{\alpha}$. Si su valor es muy grande, el aprendizaje será más rápido, sin embargo, es posible que también haya más oscilaciones durante el proceso, o incluso que se vuelva inestable. Además, un factor de aprendizaje excesivamente pequeño tampoco garantiza la convergencia de la red, ya que se podría caer en un mínimo local de la función a minimizar, y ante esa situación, la red ya no aprendería más [17].

Algunas estrategias para tratar de solucionar estos problemas relacionados con el factor de aprendizaje son:

- Hacer que disminuya a lo largo del entrenamiento. Factores de aprendizaje altos hacen que se aprenda más rápido al principio del entrenamiento, pero llegado un punto, se necesitará un valor más bajo para converger al mínimo de la función de error. Por otro lado, factores de aprendizaje bajos hacen que

la convergencia se realice muy lentamente, por lo que el modelo puede no converger. Una solución sería hacer que este factor dependiera de los ciclos del entrenamiento, para así comenzar el proceso con un factor que le permita aprender más rápido al principio, y poco a poco, disminuirlo progresivamente hasta el final para aproximar el error mínimo [3].

- Regular el factor de aprendizaje dinámicamente. Para ello se incrementa su valor linealmente mientras el error cometido disminuya, y cuando crezca, se reducirá multiplicando por un factor entre 0 y 1 [17].

3.3.4.4 Hiperparámetros: Batch Size

Define la cantidad de ejemplos que le son pasados a un modelo de red neuronal para que ajuste sus pesos en base al error que comete al clasificarlos. Su valor es una potencia de base 2, y de él depende la versión del proceso de entrenamiento que se aplicará a la red, que, si recordamos, puede ser *on-line*, *batch*, o *mini-batches*, dependiendo de si los pesos se actualizan tras procesar cada ejemplo del conjunto de entrenamiento, tras todos los ejemplos del conjunto de entrenamiento, o tras cierto número de ejemplos del conjunto de entrenamiento.

3.3.4.5 Hiperparámetros: Steps, Epochs, Iterations.

Estos conceptos están vinculados a la cantidad de veces que los procesos de entrenamiento y validación se ejecutan. Conceptualmente se definen como sigue:

- *Step*: un paso del entrenamiento implica que ha concluido el procesamiento completo de un único *mini-batch* por parte de un modelo de red neuronal.
- *Iteration*: una iteración del entrenamiento hace referencia a la cantidad de pasos que son necesarios para procesar todos los *mini-batch*, o dicho de otra forma, la cantidad de pasos necesarios para procesar todos los datos.
- *Epochs*: las épocas del entrenamiento se refieren a la cantidad de veces que todos los datos son procesadas de principio a fin por la red neuronal.

La relación entre estos conceptos puede ser expresada a través de ecuaciones, de forma que se obtienen la (3.37), (3.38), y (3.39), donde en esta última, $\boldsymbol{N}$ es la cantidad de iteraciones que se realizan.

$$steps = \frac{images}{mini-batch-\text{size}} \tag{3.37}$$

$$iteration = steps \tag{3.38}$$

$$epochs = \frac{images}{mini-batch-\text{size}} * N = iteration * N \tag{3.39}$$

Dependiendo de la tecnología con la que se trabaje, el hiperparámetro que permite establecer las veces que se ejecuta el entrenamiento y la validación puede variar, por ejemplo, en MATLAB se emplea el concepto de *epoch*, lo que es cómodo ya que permite definir cuantas veces se procesan todos los *mini-batch*, mientras que en *TensorFlow* [59] se emplea el concepto de *step*, que implica un ajuste más fino.

3.3.5 El problema de la generalización

La capacidad de una red neuronal para generalizar conceptos se refiere a su habilidad para clasificar correctamente un ejemplo con el que no ha sido entrenada, en su categoría correspondiente, asumiendo un margen de error no significativo.

Como las redes neuronales son modelos capaces de aproximarse muy bien a funciones que relacionan una entrada con una salida, reciben por ello el nombre de aproximadores universales de funciones, por lo que se partirá de esta forma de ver una red neuronal, como una función que aproxima a otra, de forma análoga a una regresión de tipo polinómico, para introducir el concepto de generalización.

Si tuviéramos un conjunto de muestras de ejemplo, en base a las cuales se quisiera estimar un valor de salida, se podría recurrir al concepto de la regresión polinómica para aproximar los valores de otras muestras de las que no se dispone. Cuando se emplea un polinomio de un grado muy bajo, como por ejemplo 1, se obtendrá una recta cuyos valores aproximarán en cierta medida a los valores ya conocidos, y también a otros no conocidos, lo cual es una primera aproximación correcta, pero mejorable, por lo que se recurre a aumentar el grado del polinomio.

Con un mayor grado, lo que tiene lugar es que la aproximación a los valores de los datos ya conocidos es más fiel, y por lo tanto, también lo será para aquellos nuevos

datos no disponibles, sin embargo, esta aproximación a los valores ya conocidos, sigue sin ser perfecta, por lo que se continuaría aumentando el grado del polinomio.

Llegado un punto, el grado del polinomio sería tal, que, dada una muestra de ejemplo conocida, se obtendría su valor de salida exacto, por lo que se habría obtenido una función que calcula perfectamente las salidas de los datos de que se dispone, lamentablemente, para aquellas muestras no conocidas, el valor de la aproximación a ellas sería mucho peor, por lo que se puede decir, que se han sobreajustado el grado y los valores de los coeficientes del polinomio.

La figura 3.6, representa gráficamente lo descrito en este ejemplo, donde una aproximación muy débil se denomina infrajuste, o *underfitting*, y una aproximación excesiva a un conjunto de datos, sobreajuste u *overfitting*.

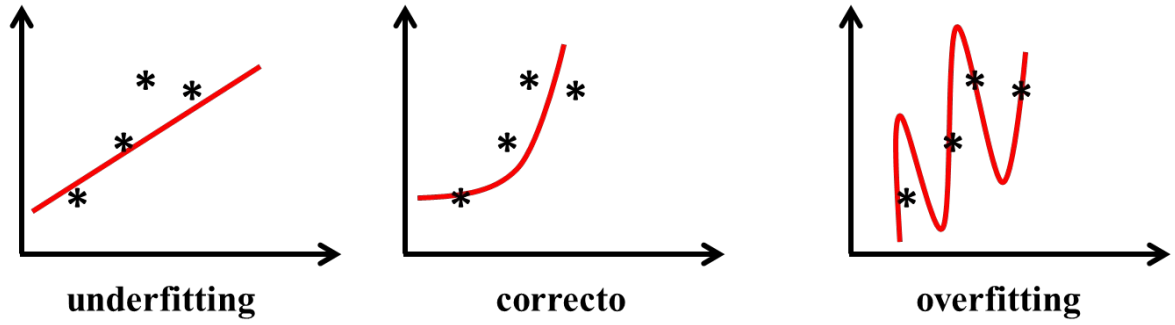

Figura 3.6: Símil entre regresión polinómica y CNN. Fuente: Aprendizaje Profundo [3].

Para evitar caer, en la medida de lo posible, en estos problemas, se han propuesto diversas soluciones a este respecto, algunas de las más empleadas son las que se detallan a continuación [16, 3]:

- Separar todas las muestras de ejemplo en tres conjuntos: siendo rigurosos, los ejemplos deberían separarse en 3 subconjuntos, el de entrenamiento, que contiene las muestras de ejemplo para que el modelo pueda aprender, el de validación, que se emplea para probar el ajuste de los hiper-parámetros, y el conjunto de prueba, o *test*, cuyas muestras se emplean para comprobar la generalización del modelo con datos que no se han empleado en los procesos anteriores. En este caso, se pueden seguir por ejemplo las distribuciones 60% *train* – 20% *val* – 20% *test*, 70% *train* – 15% *val* – 15% *test*, u otras.

- Técnica de la parada temprana: al comenzar un proceso de entrenamiento la red se adaptará progresivamente al conjunto de datos de aprendizaje, y poco a poco mejorará la generalización de sus clasificaciones, sin embargo, en algún momento, el modelo se ajustará a las características que poseen estos datos, por lo que disminuirá su capacidad para clasificar otros ejemplos de carácter similar con los que no se ha entrenado, es decir, pierde su capacidad para generalizar. Por eso, una estrategia habitual suele ser parar el proceso de entrenamiento cuando se aprecia que el error de las clasificaciones con el conjunto de entrenamiento, converge, y el error de las clasificaciones con el conjunto de validación, diverge, lo que puede parecer sencillo, pero no lo es, pues lo más normal es que se produzcan varias oscilaciones en los resultados de las clasificaciones del conjunto de validación, por lo que habrá que tratar de escoger el mínimo adecuado señalado por los resultados de la validación.
- Encontrar el número de ejemplos adecuados para el entrenamiento de la red: a menudo, el número de ejemplos para el entrenamiento está limitado, o es reducido, y sin embargo, el número de parámetros de la red es muy elevado. Por suerte, se demostró [25], y corroboró [26], que dada una red de n entradas, h capas ocultas, y un total de w pesos, se requiere un número de ejemplos para el aprendizaje p, con $p = w / \varepsilon$, para alcanzar un error de generalización de ε. Esta relación también sirve para que, dado un número fijo de ejemplos para el entrenamiento p, y un error ε, que se desea alcanzar, permita averiguar el tamaño de red que se necesita.
- Limitar el número de entradas de la red: es otra forma diferente de reducir de forma considerable el número de pesos del modelo, lo que permitirá obtener una relación $p = w / \varepsilon$ más favorable.
- Desconexión de neuronas (*dropout*): la idea que subyace bajo el concepto de *dropout* es que, en redes muy profundas, algunas conexiones alcanzarán una capacidad predictiva superior a otras, por lo que, para evitar esa situación, en cada iteración se inhabilitan aleatoriamente algunas neuronas del modelo, de forma que este queda definido por las que prevalecen activas. Este efecto favorece que las supervivientes ajusten mejor el valor de sus pesos.

- Añadir, o quitar, parámetros, y/o capas, al modelo de red: del mismo modo que en el caso de la regresión polinómica la complejidad del ajuste depende del grado del polinomio y los valores de sus coeficientes, con los modelos de red neuronal ocurre algo similar, su complejidad viene dada por sus pesos, sus sesgos, y los valores que toman estos parámetros. Si el problema para el que se utiliza la red es sencillo, se necesitará un modelo con pocos pesos y sesgos que ajustar para resolverlo, es decir, se necesitará una red más simple, mientras que si el problema al que se enfrenta es complejo, entonces se necesitarán más pesos y sesgos que ajustar para poder resolverlo, o lo que es lo mismo, un modelo de mayor tamaño. Por lo tanto, debe ajustarse el tamaño de la red a la complejidad del problema que se está tratando. En este punto cabe destacar que se ha demostrado [24], que el número efectivo de parámetros del modelo es menor que su número total de pesos. Y esta es la razón por lo que funcionan las técnicas de parada temprana para evitar el sobreajuste, ya que detener el entrenamiento, es el equivalente a reducir la cantidad de parámetros efectivos del modelo, en caso contrario, esta iría aumentando a medida que el entrenamiento del modelo progresa.
- Regularización (*regularization*): consiste en modificar la función de error que se desea minimizar añadiéndola términos adicionales que penalicen los pesos con valores elevados. El tipo más común de regularización es $\boldsymbol{L_2}$, con la cual la función de error se reescribe como en la ecuación (3.40), y el cálculo de los pesos se realiza como se indica en la ecuación (3.41), en ambas, $\boldsymbol{\lambda}$ es el coeficiente de regularización, cuyo valor permite controlar la protección contra el sobreajuste. La interpretación de $\boldsymbol{L_2}$ es muy intuitiva, pretende hacer que la red trate de emplear todas sus entradas un poco, en lugar de emplear solo algunas de sus entradas mucho, para ello, penaliza los pesos con valores altos frente a aquellos con valores más bajos, por lo que existe una clara preferencia por pesos con valores más discretos. Una peculiaridad de $\boldsymbol{L_2}$ en la ecuación (3.41), es que su uso hará que el valor de los pesos tienda a cero (*weight decay*).

$$J(w) = J(w) + \frac{\lambda}{2} * w^2 \tag{3.40}$$

$$w_i(t+1) = w_i(t) - \varepsilon * \nabla J_i(w) - \varepsilon * \lambda * w_i(t) \quad (3.41)$$

- Selección de datos: cuando se posee un número limitado de ejemplos para el entrenamiento, y cada uno tiene un número de características elevado, es conveniente seleccionar aquellas características más importantes.
- Aumento de datos (*data augmentation*): conjuntos de datos grandes reducen la posibilidad de que la red se sobreajuste, sin embargo, a veces no se podrán recopilar tantos datos como se desea. En este caso, el conjunto de ejemplos se puede incrementar aplicando transformaciones a sus ejemplos, o creando nuevas muestras mediante interpolación lineal.
- Validación cruzada de ***K*** iteraciones (***K****-fold cross-validation*): se dividen todas las muestras de ejemplo de que se disponen en ***K*** grupos, donde ***K***-1 grupos se utilizan para el entrenamiento del modelo, y el grupo restante se emplea para evaluar el modelo. Con esta estrategia se puede averiguar el error que el modelo comete al calcular el error promedio para las ***K*** pruebas.

Capítulo 4 - Redes Neuronales Convolucionales

4.1 Introducción

En este apartado se introducirá el concepto de Red Neuronal Convolucional (CNN; *Convolutional Neural Network*), aplicadas a imágenes, que es el objetivo del trabajo. Se trata de una de las evoluciones más populares de las redes neuronales formadas solo por neuronas, como por ejemplo el Perceptrón Multicapa (MLP; *Multi-Layer Perceptron*) [17], visto anteriormente. Las diferencias de estos modelos respecto de las redes del tipo MLP son:

- Entrada, salida, y objetivo del modelo: en las redes MLP los datos de entrada son muestras del tipo ($\boldsymbol{x} \in \mathbb{R}^n$, $\boldsymbol{y} \in \mathbb{R}^m$), y el objetivo del modelo consiste en autoajustarse para aproximar una función que le permita relacionar las entradas con las salidas. En el caso de las CNN con uso en imágenes, las entradas son lotes de imágenes digitales con las cuales un modelo tratará de autoajustarse para generalizar la extracción de las características de estas, lo que posteriormente le permitirá clasificarlas en categorías. En este tipo de modelos la salida es el nombre de la categoría con mayor probabilidad de pertenencia a una clase del problema de clasificación, para una imagen de entrada.
- Tensores: a diferencia de las redes de tipo MLP donde las entradas se definen como vectores de datos, en las CNN, para representar matemáticamente una imagen se recurre al concepto de tensor como una matriz multidimensional, lo que es útil para generalizar los objetos que pueden entrar a procesarse en una red neuronal cualquiera. Por ejemplo, un vector de tamaño 1xN que sirve para representar las características de una muestra de ejemplo a pasar a un MLP, es un tensor de una dimensión, una imagen en escala de grises con un tamaño NxM, es un tensor de dos dimensiones, una imagen a color con un tamaño NxMx3, es un tensor de tres dimensiones, y un lote (*mini-batch*) de 64 imágenes a color, es un tensor de cuatro dimensiones (64xNxMx3). El concepto de tensor también aparece internamente en los modelos de CNN, ya que todas sus capas internas manipulan matrices multidimensionales para poder procesar la imagen con el objetivo determinar su categoría.

- Arquitectura y operaciones: otra diferencia destacable entre las redes de tipo MLP y las CNN, es que las primeras están formadas únicamente por capas de neuronas (perceptrones), mientras que, en las segundas existen varias capas que implementan distintas operaciones para lograr extraer las características de una imagen. De entre todas las operaciones que puede haber en una red CNN, la más emblemática es la de convolución, que se explicará más tarde.

En la figura 4.1 se muestra el modelo de red AlexNet [27], un ejemplo de CNN que servirá para explicar estas redes, sus capas, y operaciones. En rojo pueden verse las operaciones de cada capa, en azul las relaciones, que indican cual es el tamaño de los tensores de salida de las operaciones, que se ven en gris.

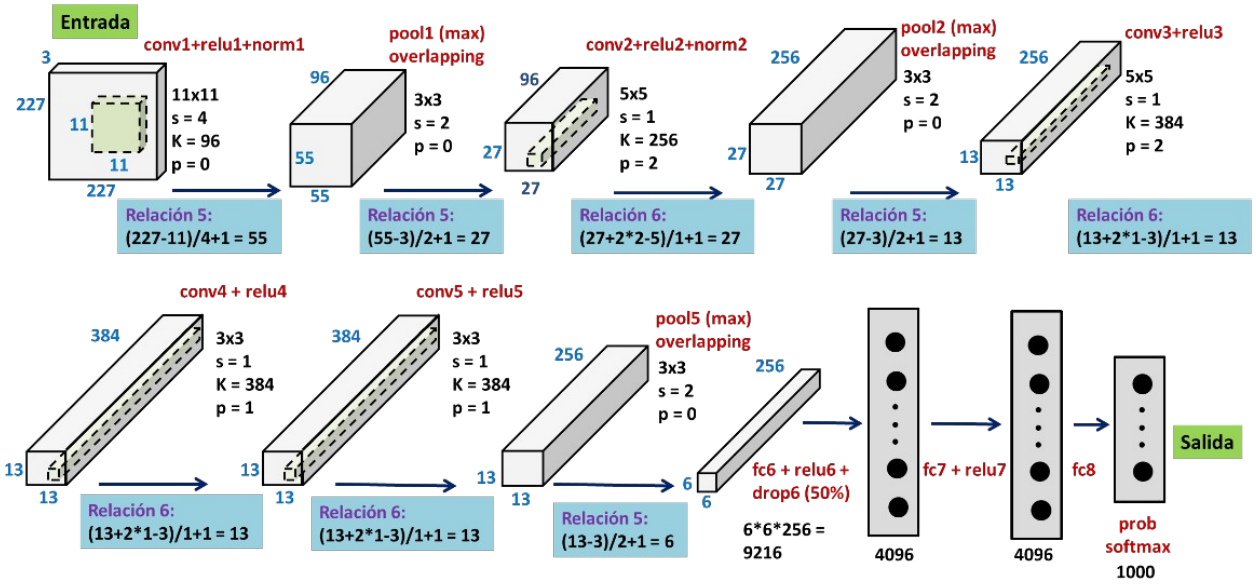

Figura 4.1: Capas, operaciones y tensores de AlexNet. Fuente: Aprendizaje Profundo [3].

Partiendo de la figura 4.1 podemos observar cómo una CNN tiene asociado un tensor de entrada, el cual no es más que una imagen digital, que en el caso de AlexNet, debe tener un tamaño de (227x227x3). Una vez este tensor tridimensional entra en la red, se realiza un recorrido secuencial hacia adelante (*feed-forward*), a lo largo de una serie de capas que implementan distintas operaciones para extraer sus características, lo que permitirá al modelo clasificarla en alguna categoría de un problema de clasificación. Estas capas del modelo son: convolución (*convolution*), ReLU (*Rectified Linear Unit*), agrupación (*pooling*), normalización (*normalization*), desconexión (*dropout*), capa totalmente conectada (*fully connected*), y exponencial normalizada (*softmax*).

También se puede apreciar en la figura 4.1 las dimensiones de los núcleos (*kernels*; ***k***), el desplazamiento (*stride*; ***s***), y el rellenado con ceros (*zero padding*; ***p***) que se aplica en cada capa de convolución. Estos parámetros son indispensables para determinar las dimensiones de un tensor (mapa de características; *feature map*) de salida de una capa de este tipo. Estos parámetros y las ecuaciones que describen estas transformaciones de los tensores serán explicados con detalle más adelante.

A continuación, se realizará una introducción a las capas de AlexNet [27], mencionadas anteriormente, y a las operaciones que estas realizan para dar a entender de forma general el funcionamiento de las CNN's. Para su descripción se han seguido las directrices teóricas de Pajares et al. [3].

4.2 Capa de convolución

Para entender el concepto de convolución se partirá del siguiente ejemplo didáctico. Supongamos que se dispone de una fuente de luz variable cuya intensidad es medida por un sensor, el cual proporciona una salida en un determinado instante de tiempo ***x***(**t**). Considerando que es muy probable que la lectura de la señal por parte del sensor esté ligeramente contaminada por un cierto ruido, lo correcto sería realizar un promediado de la salida con varias medias. Si además se tiene en cuenta que las medidas más recientes tienen más relevancia que las que se alejan en el tiempo, el promediado puede ponderarse concediendo más relevancia a las recientes, lo cual puede hacerse a través de la función de promediado ***w***(**a**), donde ***a*** representa el alejamiento en el tiempo.

Si se realiza esta operación de promediado a lo largo del tiempo, entonces se tiene una nueva función ***s***(**t**) que devuelve el valor de la señal en un instante de tiempo, esta puede verse en las ecuaciones (4.1) y (4.2), donde en la primera se considera que el tiempo toma valores continuos, y en la segunda solamente valores discretos.

$$s(t) = \int_{-\infty}^{\infty} x(a) * w(t-a)\,da \tag{4.1}$$

$$s(t) = \sum_{a=-\infty}^{\infty} x(a) * w(t-a) \tag{4.2}$$

Esta operación se denomina convolución, y se representa según las ecuaciones (4.3) y (4.4), nuevamente dependiendo de si el valor de ***t*** es continuo o discreto.

$$s(t) = \int_{-\infty}^{\infty} (x * w)(t)dt \tag{4.3}$$

$$s(t) = \sum_{a=-\infty}^{\infty} (x * w)(t)dt \tag{4.4}$$

Cabe destacar que ***w*** es una función de densidad que asegura que la salida está promediada, y que devuelve cero cuando su valor de entrada es negativo, lo que imposibilita la toma de valores en instantes de tiempo futuros.

En el ámbito de las CNN, el producto de las funciones ***x*** y ***w*** es sustituido por uno entre dos tensores, donde el primero sería una imagen digital (*Input*), o un mapa de características (*feature map*) si se refiere a una capa intermedia, y el segundo un tensor de parámetros (*Kernel*), normalmente de tamaño impar, que el modelo ajustará durante el aprendizaje. Con estos elementos, la salida de la operación pasa de ser un escalar, a ser un nuevo tensor, es decir, una matriz multidimensional.

Otra diferencia significativa es que los tensores están conformados por un número de elementos discretos, y que pueden poseer más dimensiones que una función de una sola variable, por lo que la convolución a realizar ha de ser para valores de tipo discreto, y además debe realizarse sobre más de un eje a la vez.

Teniendo en cuenta estas diferencias, se introduce la convolución discreta en dos dimensiones definida en la ecuación (4.5), que puede interpretarse como una forma de cuantificar el impacto del *kernel* al desplazarse este sobre las distintas regiones de la imagen de entrada.

Las variables de la ecuación (4.5) son, la imagen de entrada ***I***, el *kernel* ***K*** con el que convolucionar la imagen de entrada, la posición espacial (***i*** y ***j***) de una característica en el tensor de salida ***S***, y ***m*** y ***n*** los valores que marcan el recorrido por el kernel ***K***.

$$S(i,j) = (K * I)(i,j) = \sum_{n=-F}^{F} \sum_{m=-F}^{F} I(i-m, j-n) * K(m,n) \tag{4.5}$$

Cabe destacar que los elementos de esta última ecuación se han de visualizar como en la imagen de la figura 4.2, donde ***T*** define las dimensiones de la imagen a convolucionar (*Tensor*), ***F*** las dimensiones del núcleo convolucional (*Filter*), y ***R*** las dimensiones del mapa de características de salida (*Result*). Con esta disposición de los elementos puede observarse directamente que estos se dividen como si estuvieran centrados en un sistema de coordenadas cartesianas, y con un ejemplo puede comprobarse fácilmente que el recorrido del *kernel* sobre la imagen tiene lugar en zigzag desde abajo a la derecha, hasta llegar arriba a la izquierda. Una última observación al respecto de esta operación es que el *kernel* se multiplica rotado 180º por cada región de la imagen por la que va pasando.

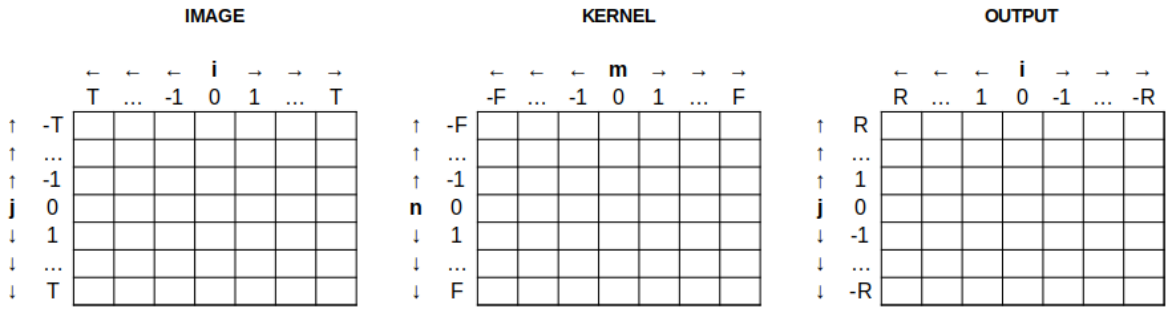

Figura 4.2: Elementos de entrada y salida de una operación de convolución

Existe otra versión de la fórmula de convolución, ecuación (4.6). Esta fórmula surge de la propiedad conmutativa de la convolución, y se interpreta gráficamente como una forma de cuantificar el impacto de la imagen al desplazarse esta sobre las distintas regiones del *kernel*. Esta versión es menos interesante que la vista en la ecuación (4.5), ya que su implementación en un programa es menos eficiente.

$$S(i,j)=(K * I)(i,j)=\sum_{n=-F}^{F}\sum_{m=-F}^{F} I(i-m,j-n) * K(m,n) \qquad (4.6)$$

Al margen de la convolución en dos dimensiones, hay otra operación que es similar a esta, la correlación cruzada, que surge cuando se invierten los signos de ***m*** y ***n*** en ***I***, en la ecuación (4.5), resultando entonces la ecuación (4.7).

$$S(i,j) = (K * I)(i,j) = \sum_{n=-F}^{F}\sum_{m=-F}^{F} I(i+m,j+n) * K(m,n) \qquad (4.7)$$

Esta operación realiza un recorrido en zigzag del núcleo, sin invertirlo 180º, sobre la imagen, pero en sentido contrario a la convolución, es decir, el *kernel* se mueve desde arriba a la izquierda hasta a abajo a la derecha. La interpretación que tiene esta operación es comprobar el parecido existente entre una región de la imagen, y un patrón (característica) descrito por el núcleo convolucional (*kernel*). Debido a su simplicidad conceptual y operacional frente a la convolución, algunos *frameworks* de desarrollo de modelos neuronales artificiales, como *TensorFlow* [59], o *PyTorch*, han comenzado a implementar esta operación en lugar de convolución, aunque se ha mantenido el nombre de convolución. Es por eso por lo que, a partir de ahora, el término convolución se empleará únicamente para referirse a la correlación cruzada. Puede verse un ejemplo de esta operación en 2D en la figura 4.3.

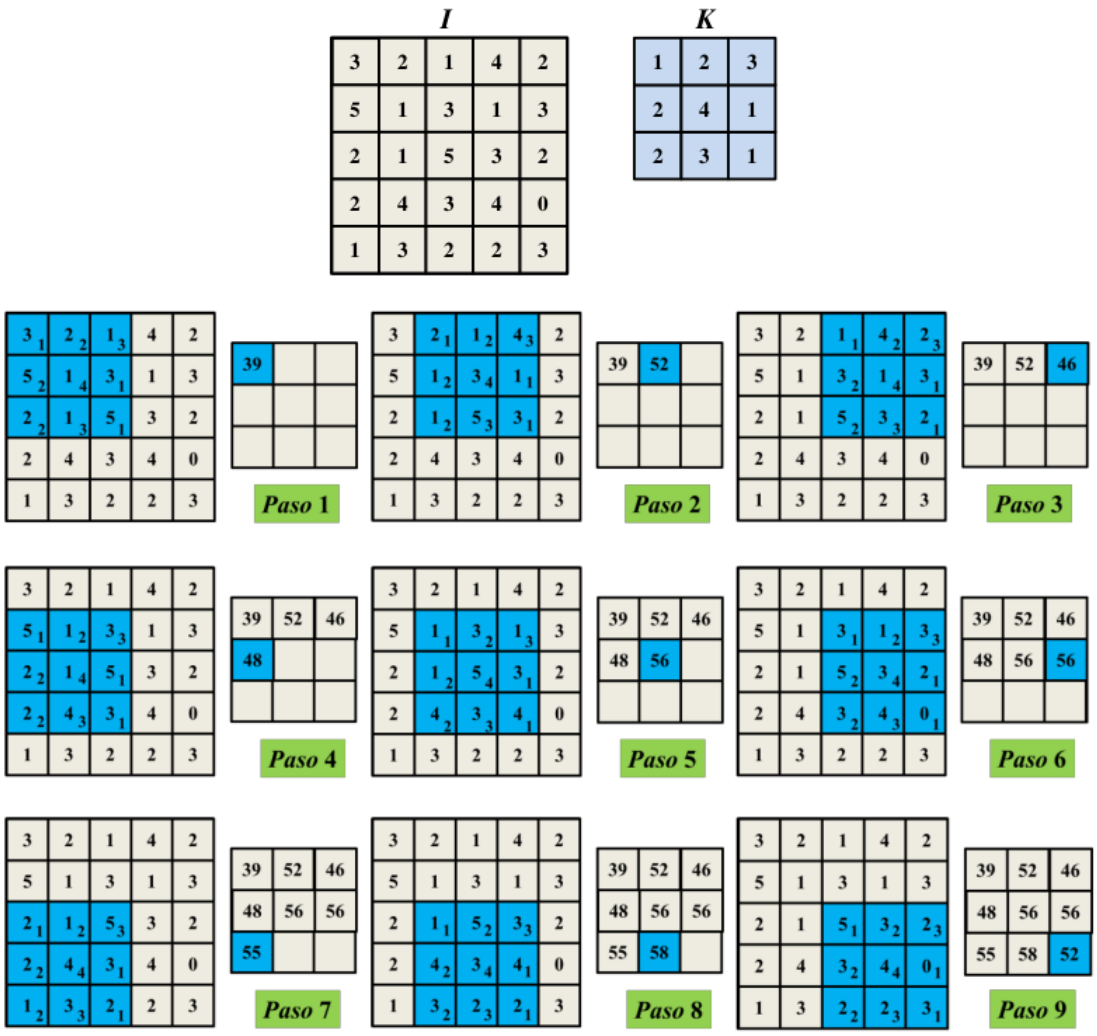

Figura 4.3: Convolución en 2D. Fuente: Aprendizaje Profundo [3].

En el ejemplo de la figura 4.3 se puede observar que la operación no se realiza sobre los píxeles en los bordes de la imagen, por lo que el tensor de salida tuene un tamaño inferior al de entrada (*valid*). Pero si se deseara un tensor de salida con un tamaño igual al de la entrada (*same*), se recurriría a lo que se conoce como rellenado con ceros, o *zero padding*, en torno a la imagen. Otra posibilidad que ofrece la convolución es la posibilidad de ajustarla para muestrear menos regiones de la imagen de entrada, lo que se puede conseguir realizando un desplazamiento (*stride*) superior a la unidad, del núcleo convolucional (*kernel*) sobre la imagen.

La operación de convolución es extensible al ámbito 3D, para el cual se aplica la fórmula de ecuación (4.8).

$$S(i,j,k) = \sum_{n=-F}^{F} \sum_{m=-F}^{F} \sum_{p=-Z}^{Z} I(i+m, j+n, k+p) * K(m,n,p) \qquad (4.8)$$

Y siguiendo de la misma forma que en la ecuación anterior, se podrían extender las operaciones a un número de dimensiones mayor aplicando tensores y núcleos con una dimensión superior a tres.

Es importante saber que, independientemente de que se escoja la convolución o la correlación cruzada para llevar a cabo la extracción de características, la red podrá converger a una solución, ya que se ha comprobado que si los pesos de un *kernel* convergen a ciertos valores de tal manera que estos conforman una matriz ***W*** al emplearse la convolución, cuando se emplea la correlación cruzada los valores del *kernel* convergen hacia la matriz de pesos ***-W***, es decir, la matriz ***W*** rotada 180º.

Ahora que se han visto cada uno de los elementos y conceptos que intervienen en esta operación, es posible determinar las dimensiones del mapa de características de salida $\boldsymbol{o}_{\mathbf{j}}$ teniendo en cuenta:

- $\boldsymbol{i}_{\boldsymbol{j}}$: la dimensión del tensor de entrada a lo largo de su eje ***j***.
- $\boldsymbol{k}_{\boldsymbol{j}}$: la dimensión del *kernel* a lo largo de su eje **j**.
- $\boldsymbol{s}_{\boldsymbol{j}}$: la distancia entre dos posiciones consecutivas del núcleo a lo largo del eje ***j*** (*stride*).
- $\boldsymbol{p}_{\boldsymbol{j}}$: el número de ceros concatenados al comienzo y al final del eje ***j*** (*zero padding*).

Lo más habitual es que los valores de los conceptos citados tomen el mismo valor para cada uno de sus ejes, por lo que se asumirá esto para el cálculo de la salida, lo que implica que los conceptos anteriores se citarán con bajo las mismas variables, pero sin subíndice: ***o***, ***i***, ***k***, ***s***, ***p***.

Las combinaciones de estos factores dan lugar a un total 6 posibilidades diferentes para calcular el tamaño del tensor de salida, estas son las siguientes:

1. Sin relleno (*no zero padding*) y una unidad de desplazamiento (*unit stride*):
    - Dados ***i***, ***k***, ***p*** = 0, y ***s*** = 1, entonces la dimensión de la salida viene dada por la ecuación (4.9):

$$o = (i - k) + 1 \tag{4.9}$$

    - La figura 4.4 muestra un ejemplo gráfico tomando los valores ***i***=4, ***k***=3, ***s***=1, ***p***=0:

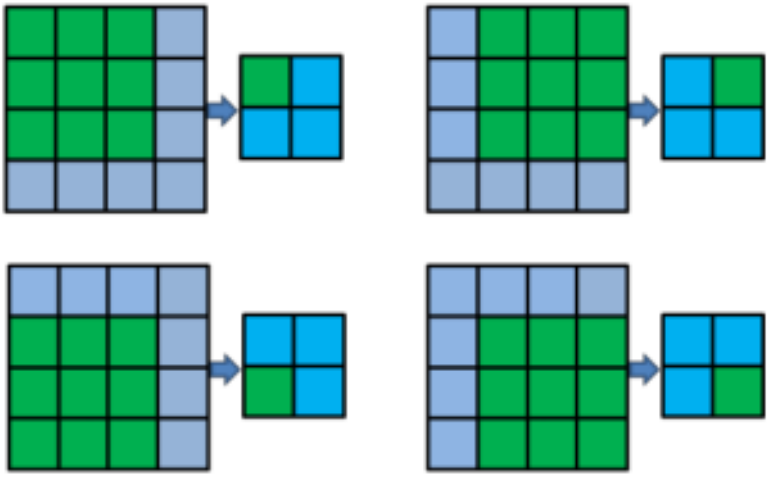

Figura 4.4: Convolución *no zero padding* y *unit stride*. Fuente: Aprendizaje Profundo [3].

2. Con relleno (*zero padding*) y una unidad de desplazamiento (*unit stride*):
    - Dados ***i***, ***k***, ***p***, y ***s*** = 1, entonces la dimensión de la salida viene dada por la ecuación (4.10):

$$o = (i - k) + 2 * p + 1 \tag{4.10}$$

    - La figura 4.5 muestra un ejemplo gráfico tomando los valores ***i***=5, ***k***= 4, ***s***=1, ***p***=2:

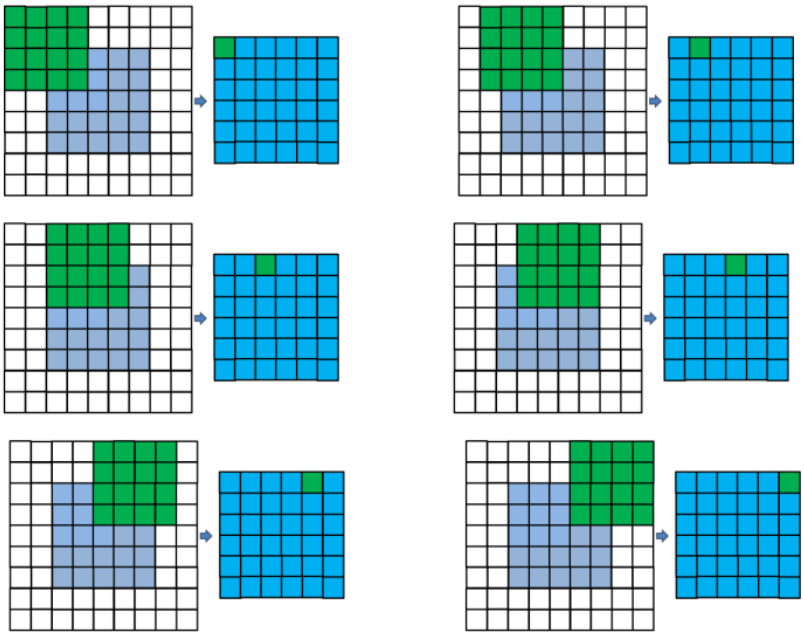

Figura 4.5: Convolución *zero padding* y *unit stride*. Fuente: Aprendizaje Profundo [3].

3. Con relleno (*zero padding*) suficiente como para obtener la misma dimensión de entrada que de salida, y una unidad de desplazamiento (*unit stride*):
   - Dados ***i***, ***k***, ***p*** = [***k*** / 2], y ***s*** = 1, entonces la dimensión de la salida viene dada por la ecuación (4.11), donde los corchetes representan la función suelo (*floor*), que implica el desprecio de los decimales del resultado.

$$o = i + 2 * \left[\frac{k}{2}\right] - (k - 1) \qquad (4.11)$$

   - La figura 4.6 muestra un ejemplo gráfico tomando los valores ***i***=5, ***k***=3, ***s***=1, ***p***=1:

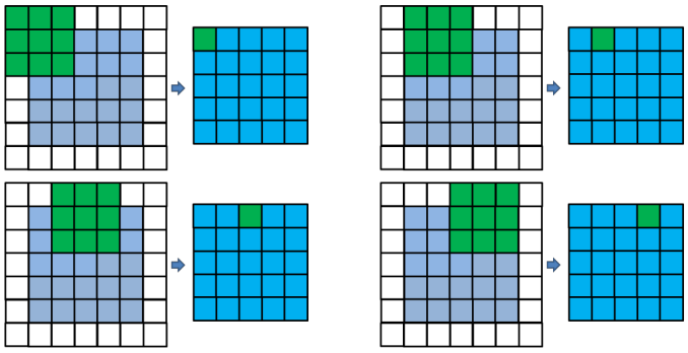

Figura 4.6: Convolución *half (same) padding*. Fuente: Aprendizaje Profundo [3].

4. Con relleno (*zero padding*) completo para obtener una mayor dimensión de salida, y una unidad de desplazamiento (*unit stride*):
   - Dados ***i***, ***k***, ***p*** = ***k*** - 1, y ***s*** = 1, entonces la dimensión de la salida viene dada por la ecuación (4.12):

$$o = i + 2 * \left\lceil \frac{k}{2} \right\rceil - (k - 1) \tag{4.12}$$

   - La figura 4.7 muestra un ejemplo gráfico tomando los valores ***i***=5, ***k***=3, ***s***=1, ***p***=2:

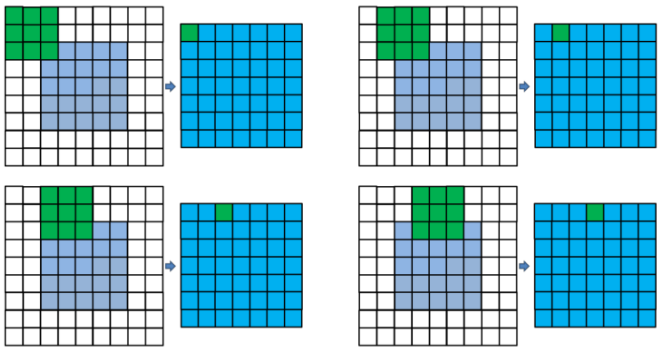

Figura 4.7: Convolución *full padding*. Fuente: Aprendizaje Profundo [3].

5. Sin relleno (*no zero padding*) y con un desplazamiento superior a la unidad (*non unit stride*):
   - Dados ***i***, ***k***, ***p*** = 0, y ***s*** > 1, entonces la dimensión de la salida viene dada por la ecuación (4.13):

$$o = \left\lceil \frac{i - k}{s} \right\rceil + 1 \tag{4.13}$$

   - La figura 4.8 muestra un ejemplo para los valores ***i***=5, ***k***=3, ***s***=2, ***p***=0:

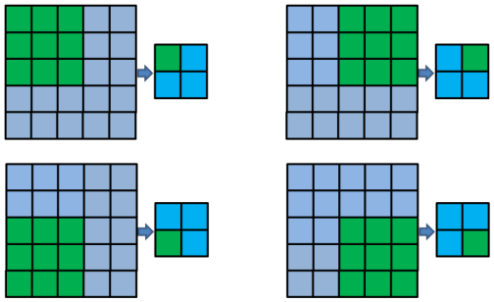

Figura 4.8: Convolución *no zero padding* y *no unit stride*. Fuente: Aprendizaje Profundo [3].

6. Con relleno (*zero padding*) y con un desplazamiento superior a la unidad (*non unit stride*):
   - Dados ***i***, ***k***, ***p*** > 0, y ***s*** > 1, entonces la dimensión de la salida viene dada por la ecuación (4.14):

$$o = \left[\frac{i+2 * p-k}{s} \right] +1 \qquad (4.14)$$

   - La figura 4.9 muestra un ejemplo para los valores ***i***=5, ***k***=3, ***s***=2, ***p***=1:

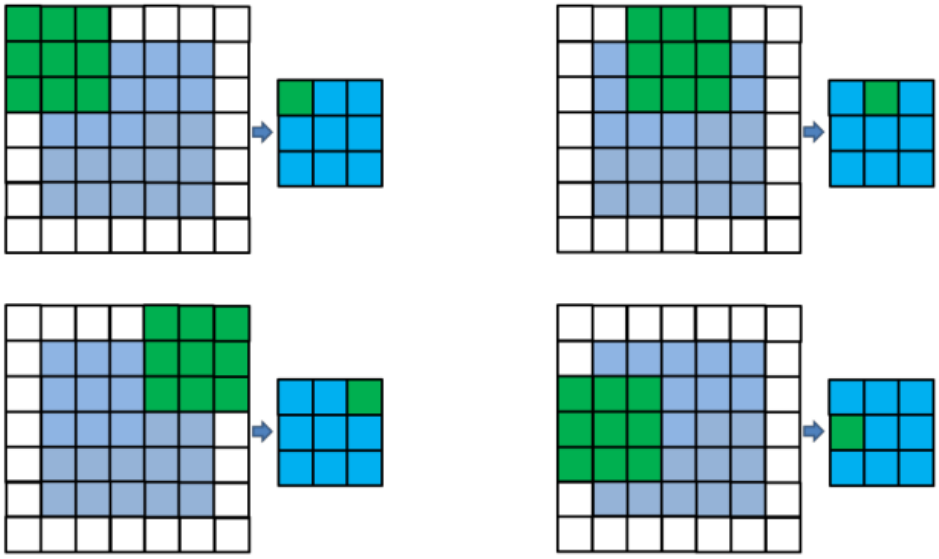

Figura 4.9: Convolución *zero padding* y *no unit stride*. Fuente: Aprendizaje Profundo [3].

4.3 Capa completamente conectada

Una capa completamente conectada es aquella que se compone de una o varias de las neuronas que implementan la operación lineal que se describe en la ecuación (3.1), del apartado 3.3. El conjunto de estas capas sirve para realizar la extracción de características a múltiples niveles, comenzando por características de nivel más bajo, y terminando por aquellas que son de nivel más alto. Por ejemplo, una CNN como AlexNet [27] posee hasta tres capas de este tipo. La primera es para la extracción características de bajo nivel, como, por ejemplo, los bordes. La segunda es para la extracción de características de nivel intermedio, por ejemplo, los contornos. Y la tercera, para describir un objeto en base a las partes que lo componen.

4.4 Capa de activación no lineal

Una capa de activación no lineal se suele situar a continuación de otras capas cuyas operaciones están basadas en funciones lineales, como es el caso de las capas de convolución y la *fully connected*. Esta capa, como se comentó en el apartado 3.3, tiene una responsabilidad doble. La función matemática que emplea, por un lado, permite expresar el grado en que una característica está presente, y por otro lado, debe evitar el colapso de capas contiguas que apliquen operaciones lineales en una sola capa, algo desastroso, ya que la CNN perdería la utilidad de sus operaciones para abordar problemas de clasificación multiclase no lineales.

Las fórmulas de las funciones de activación más comunes ya están descritas en el apartado 3.3 a través de las ecuaciones (3.2), (3.3), (3.4), (3.5), y (3.6), sin embargo, es necesario exponer las características principales de las más populares.

Una de las funciones de activación más clásicas es la función sigmoide, ecuación (3.5), cuyo parámetro ***c***, especifica el punto del eje horizontal en el que la función se centra, y el parámetro ***a***, la orientación de la función y lo rápido que esta converge hacia los extremos cero o uno. La función es de mucha utilidad para expresar que algo es muy grande o muy pequeño, sin embargo, presenta un par de problemas:

1. El gradiente de la función tiende a cero (se satura) cuando esta se aproxima a sus extremos, lo que repercute negativamente al ajustar los pesos.
2. Su valor medio de salida no es cero; los pesos tienden a ser positivos.

La imagen de la figura 4.10 muestra dos sigmoides con distinta orientación.

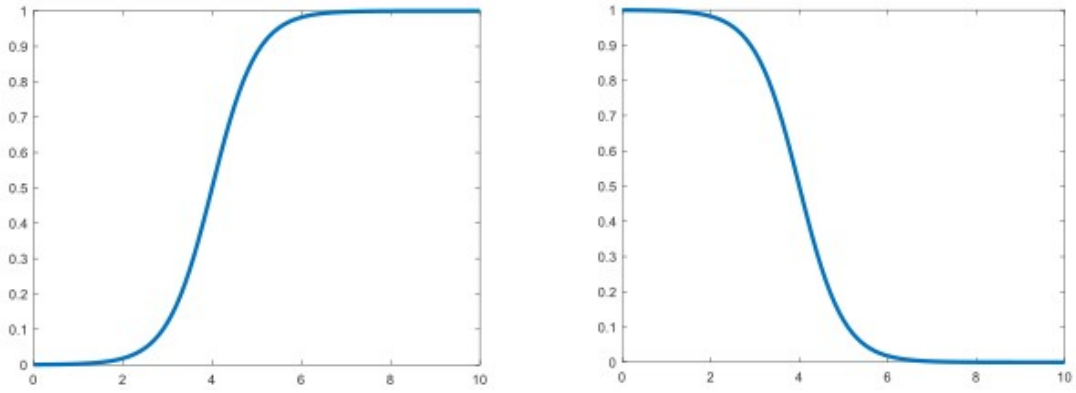

Figura 4.10: Funciones de tipo sigmoide. Fuente: Aprendizaje Profundo [3].

Otra función de activación clásica es la tangente hiperbólica, ecuación (3.6), que es muy parecida gráficamente a la sigmoide, sin embargo, a diferencia de ella, toma valores de salida comprendidos en el rango [-1, 1]. La principal problemática que presenta esta función es que también provoca la saturación del vector gradiente. A continuación, puede verse en la figura 4.11 su representación gráfica.

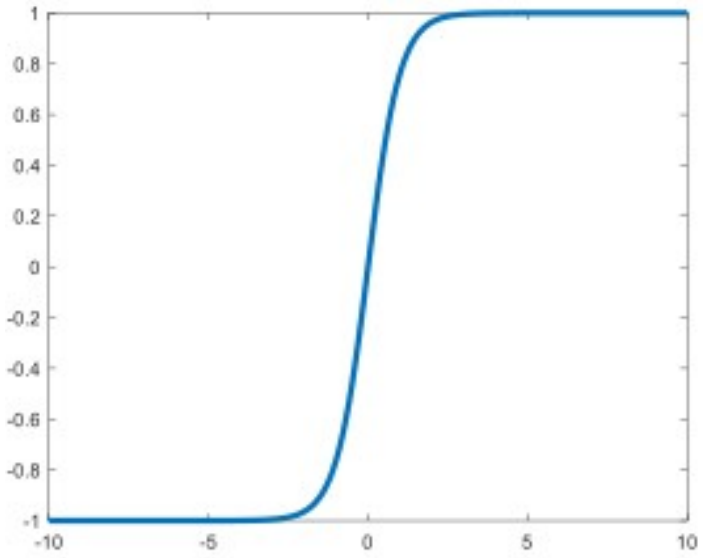

Figura 4.11: Función tangente hiperbólica. Fuente: Aprendizaje Profundo [3].

Aunque las dos funciones anteriores son muy populares, algunas de las que más se emplean en la actualidad para conseguir activaciones no lineales en las CNN son la función *ReLU* (*Rectified Linear Unit*), ecuación (3.3), la función *Leaky ReLU* (*Leaky Rectified Linear Unit*), ecuación (3.4), y otras funciones gráficamente muy similares a ellas, por lo que se describirán estas al considerarse altamente representativas.

Las características principales de estas funciones son las siguientes:

- Ambas funciones solucionan el problema de la saturación del gradiente para valores de entrada mayores que cero. Sin embargo, la función *ReLU* se satura para valores menores o iguales a cero, por lo que, al no poder derivarse en ese intervalo, suele recurrirse a generar un valor aleatorio pequeño que esté en el rango [0, 1] como reemplazo. La función de activación *Leaky ReLU* no posee este problema, ya que, para valores de entrada inferiores a la unidad, multiplica su valor por un coeficiente muy pequeño, lo que la hace derivable.
- Sendas funciones implican una complejidad computacional muy baja debido a sus sencillos cálculos.

La figura 4.12 muestra a la izquierda la función *ReLU* y a la derecha la función *Leaky ReLU*.

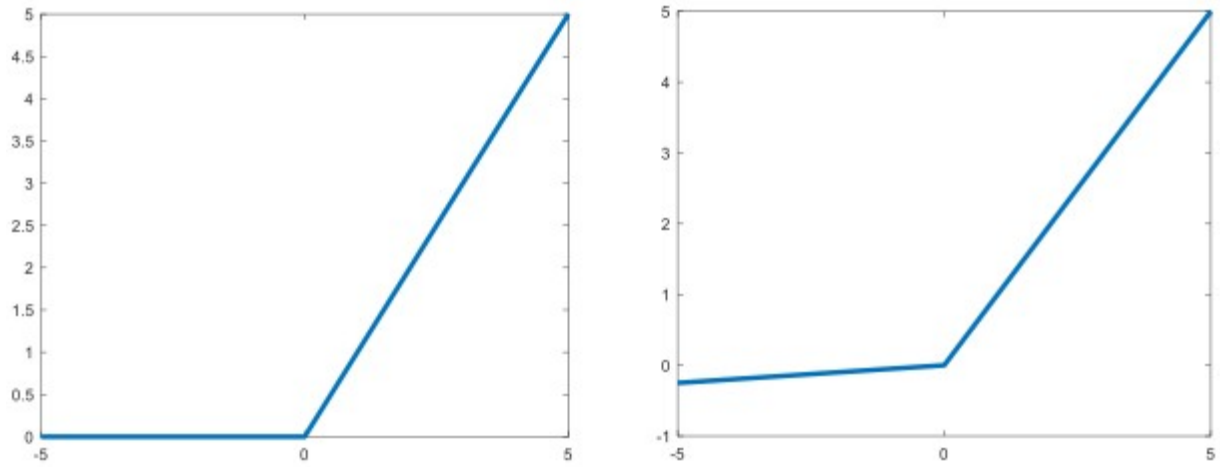

Figura 4.12: Función *ReLU* y *Leaky ReLU*. Fuente: Aprendizaje Profundo [3].

4.5 Capa de agrupamiento

Una capa de agrupamiento, o *pooling*, ayuda a reducir la dimensionalidad de una imagen o tensor, a obtener una representación aproximadamente invariante a pequeñas traslaciones de esta, y a identificar si alguna característica está presente en ella, pero sin preservar la ubicación espacial exacta de esta.

La operación consiste en dividir la imagen (tensor) de entrada en ventanas de (NxN) dimensiones y aplicar sobre cada región enmarcada una operación de máximo (*max pooling*), o bien una media de valores (*average pooling*).

Para realizar esta tarea lo que se hace es crear una única ventana que se desplaza cierta cantidad de unidades (*stride*) sobre la imagen (tensor), siguiendo el mismo sentido al desplazarse que la operación de convolución (*cross correlation*). Su salida depende del tamaño de la entrada, del tamaño de la ventana, y de la longitud los desplazamientos (*strides*) de la ventana sobre la imagen. La operación se puede realizar con o sin *zero padding*.

La figura 4.13 muestra a su izquierda un ejemplo de imagen binaria a modo de entrada, que es dividida en varias ventanas de (3x3), mientras tanto, a su derecha, muestra los resultados numéricos y visuales de aplicar tanto *max pooling* como *average pooling*.

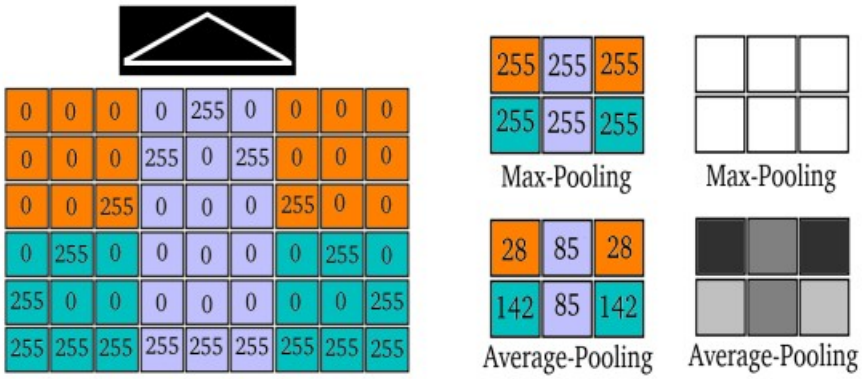

Figura 4.13: Ejemplo ilustrativo de *max pooling* y *average pooling*

En este punto, cabe destacar que una capa típica de convolución consta de tres estados. En el primero se realizan varias convoluciones para generar un conjunto de activaciones lineales. En el segundo cada activación lineal es pasada a través de una función de activación no lineal, como por ejemplo *ReLU*. A este segundo estado se le conoce como *detector stage*. Y en el tercero, se utiliza la función de *pooling* para reemplazar la salida de la red en distintas localizaciones, con una operación de carácter estadístico que involucra a las salidas más cercanas.

La figura 4.14 muestra de forma abstracta cómo una CNN recibe, en dos ocasiones diferentes, el carácter 3 a modo de entrada, y luego lo procesa a través de la pila de operaciones descrita en el párrafo anterior: *convolution*, *ReLU*, y *max pooling*.

En dicha figura 4.14, los treses de la fila central simbolizan diferentes núcleos de convolución para poder reconocer un 3.

Estas características aprendidas por los tres núcleos de convolución reflejan que la red aprendió las diferentes transformaciones de la entrada que resultan invariantes de cara a la generación de la salida. Después de las convoluciones, cada salida generada es pasada por una unidad de activación (*detector*), el cual emite una señal a la capa de *max pooling* que refleja el grado en que se ha detectado cómo de presente está el 3 en la imagen. En el primer caso, cuando el 3 aparece en la capa de entrada, se produce una señal de activación fuerte en el primer detector, sin embargo, en el segundo caso, cuando el 3 aparece en la entrada, es el tercer detector el que emite una fuerte señal. En cualquiera de los dos casos, al realizarse la operación *max pooling* se obtendrá una fuerte activación independientemente del detector que se active, por lo que ambas situaciones producirán un resultado aproximadamente similar.

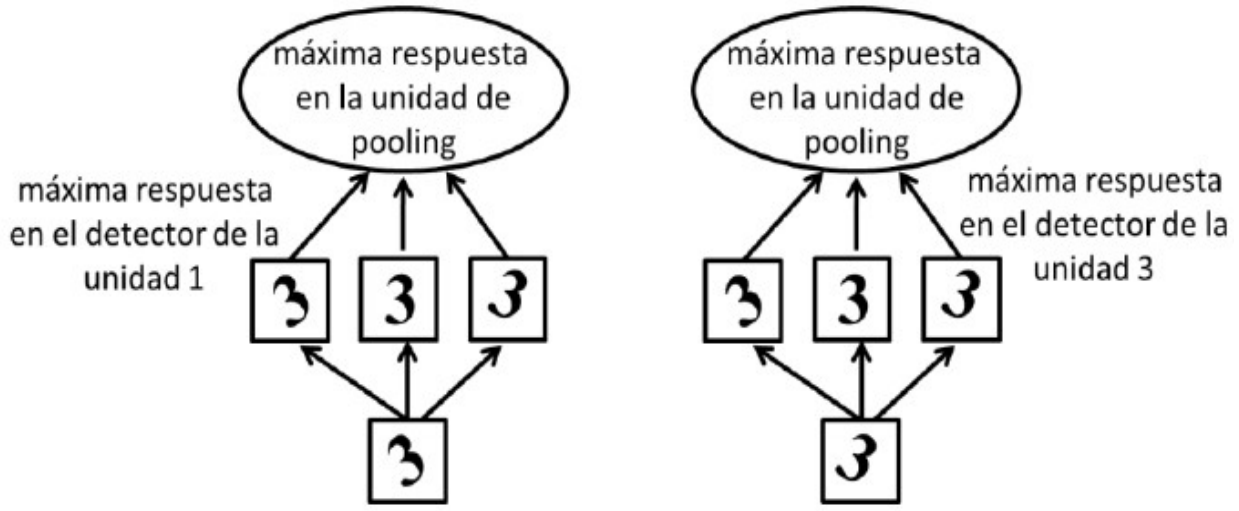

Figura 4.14: Pila *convolution*, *ReLU*, y *max pooling*. Fuente: Aprendizaje Profundo [3].

4.6 Capa de desconexión

La capa de desconexión, o *dropout*, se describió anteriormente en el apartado 3.3.5 como una forma de solventar el problema de generalización de los modelos. Lo que hace es provocar desconexiones aleatorias de las neuronas de una capa, con ello lo que consigue es que las neuronas supervivientes actualicen mejor sus pesos.

4.7 Capa de normalización

Como se ha mencionado previamente, durante el proceso de entrenamiento, las redes neuronales ajustan los pesos de sus capas mediante *Gradient Descent* y *Backpropagation Error* [21], pero suponiendo que no se producirán cambios en las capas anteriores, un error, ya que la distribución de entradas (las activaciones de entrada) de cada capa cambiarán cuando sus capas anteriores actualicen sus pesos. Este problema es conocido como Desplazamiento Covariante Interno (*Internal Covariate Shift*), y hace referencia a que las capas de un modelo de red se están adaptando continuamente a los cambios de las entradas, lo que prolonga la duración del proceso de entrenamiento.

Normalizar las entradas provenientes de una capa anterior de la red evitará que los cambios de los valores de entrada de una capa cambien bruscamente, lo que se traduce en una aceleración de la convergencia del error [28].

En Ioffe y Szegedy [29] se propone la estrategia de realizar la normalización por lotes (*mini-batches*), y aplicarla en cada mapa de características escalares para que sus valores sigan una distribución de valores de media cero y varianza uno. Esta estrategia permite emplear razones de aprendizaje más altas, y ser un poco menos cuidadosos con la inicialización de los parámetros.

Para una capa de red con una entrada de tamaño *d*-dimensional, $\boldsymbol{x}=\{x^{(1)}, \ldots, x^{(d)}\}$, se normaliza cada dimensión (activación) siguiendo la fórmula dada en la ecuación (4.15), donde ***E*** es la media de las activaciones, ***Var*** la varianza de las activaciones, y la raíz cuadrada de ***Var***, la desviación típica de las activaciones.

$$\hat{x}^{(k)} = \frac{x^{(k)} - E\left[x^{(k)}\right]}{\sqrt{Var\left[x^{(k)}\right]}} \tag{4.15}$$

Un problema importante en este punto es que, tal vez, la normalización provoque que los valores de entrada de una función de activación puedan quedar restringidos a un rango en el que la función se comporta como una función lineal. Para abordar este problema se introduce para cada nivel de activación $\boldsymbol{x}^{(k)}$, un par de parámetros $\boldsymbol{\beta}^{(k)}$ y $\boldsymbol{\gamma}^{(k)}$, que escalan y aplican una traslación sobre el valor normalizado, tal como se puede ver en la ecuación (4.16). Estos parámetros son aprendidos por la CNN.

$$x^{(k)} = \gamma^{(k)} * \hat{x}^{(k)} + \beta^{(k)} \tag{4.16}$$

Para aplicar este procedimiento de normalización en una CNN, se puede recurrir al Algoritmo-1 ***MN***, basado en las ecuaciones (4.15) y (4.16), vistas anteriormente. Este funciona de la siguiente manera. Supongamos que se posee un mini-lote ***M*** de dimensión ***m***, el cual produce las salidas $A = \{x_1, \ldots, x_m\}$ de una activación $\boldsymbol{x}^{(k)}$ concreta. Llamaremos a $\hat{x}_1, \ldots, \hat{x}_m$, las salidas normalizadas de la activación, y a $y_1, \ldots, y_m$, a las transformaciones lineales de las salidas normalizadas. De este modo, puede definirse la función $MN_{\beta,\gamma}(x_i) : A \rightarrow \{y_1, \ldots, y_m\}$. Una vez se ha definido esta función, para cada activación $\boldsymbol{x}^{(k)}$ de una capa, el algoritmo realiza la operación de la ecuación (4.17), que se compone a su vez de la operación de la ecuación (4.18), donde $\boldsymbol{\varepsilon}$ es una constante que se le añade a la varianza del mini-lote con fines de estabilidad numérica.

$$MN_{\gamma,\beta}\left(x_i\right) = y_i = \gamma * \hat{x}_i + \beta \tag{4.17}$$

$$\hat{x}_i = \frac{x_i - \mu_A}{\sqrt{\sigma_A^2 + \varepsilon}} \tag{4.18}$$

Cuando se aplica la normalización mediante ***MN***, se hace necesario retropropagar el gradiente de la función de pérdida a través de esta transformación.

4.8 Capa función exponencial normalizada

La capa función exponencial normalizada, o *softmax*, suele ser la última capa que aparece en una CNN. Sirve para transformar un vector ***x***, en otro vector *softmax*(**x**), de igual dimensionalidad al primero y cuyas componentes son números reales en el rango [0, 1]. La suma de todas las componentes de *softmax*(**x**) da como resultado 1, y cada una de las componentes de este vector indica el grado de pertenencia de una muestra de ejemplo introducida en la red neuronal, a una categoría concreta de un problema de clasificación. Cada una de las componentes *softmax*(**x**)$_i$ se obtiene teniendo en cuenta todas las componentes del vector ***x***, tal y como se indica en la ecuación (3.19).

$$softmax\left(x\right)_i = \frac{\exp\left(x_i\right)}{\sum_{j=1}^{n} \exp\left(x_j\right)}; \forall i = 1,\ldots,n; x = \left(x_1, x_2, \ldots, x_n\right) \tag{3.19}$$

4.9 Transferencia del aprendizaje

El concepto de transferencia del aprendizaje (TL; *Transfer Learning*) es una de las estrategias más populares de cara al entrenamiento de las CNN's en imágenes, lo cual se debe a su sencillez conceptual y aplicativa. La idea básica de esta técnica es la siguiente. Una vez se ha entrenado un modelo de CNN, este ha aprendido una serie de pesos (*weigths*), sesgos (*biases*), normalmente a través de los núcleos de convolución (*kernels*), y otros parámetros que le permiten identificar una serie de imágenes particulares, por ejemplo, furgonetas y motos.

La primera vez que se entrena una CNN para identificar unas categorías de interés, los pesos (*weigths*), sesgos (*biases*), núcleos de convolución (*kernels*), y los otros parámetros, tienen que ser aprendidos desde cero, ya que como se mencionó en el apartado 3.3.3, al principio la red inicializa sus valores de forma aleatoria, y no con valores nulos. Este primer entrenamiento de la CNN para obtener los valores de los parámetros suele ser muy costoso, sobre todo en cuestiones de memoria, ya que hay que alimentar al modelo con muchos ejemplos, lo que a su vez implica que también se necesite mucho tiempo para que el modelo aprenda a generalizar.

Sin embargo, cuando las categorías del problema de clasificación son cambiadas por otras diferentes, se podría pensar que es necesario volver a entrenar el modelo de nuevo, pero no necesariamente.

Si las nuevas categorías son un subconjunto de aquellas con las que el modelo se entrenó inicialmente, o si estas comparten ciertos rasgos similares con un conjunto de categorías nuevo, entonces los parámetros del modelo vinculados a la extracción de características, como por ejemplo los correspondientes a la parte convolucional de la red, podrían preservarse en lugar volver a entrenarlos.

En este sentido, tampoco haría falta reentrenar todas las capas *fully connected* que constituyen su Perceptrón Multicapa (MLP; *Multi-Layer Perceptron*) [17] interno que realiza la clasificación de las imágenes de entrada una vez han sido extraídas sus características, ya que si se recuerda lo expuesto en el apartado 3.3.1, las primeras capas de este tipo extraen características de bajo y medio nivel, y al ser estas similares a las del problema de clasificación original, entonces estas capas también pueden conservarse.

En principio, lo único que se habría que cambiar en la CNN es la última de las capas *fully connected*, ya que como esa se encarga de describir un objeto en una imagen en base a las partes que lo componen, y estos no son necesariamente los mismos que había en el problema original, esta ha de sustituirse por una nueva capa del mismo tipo que tenga tantas neuronas como clases tenga el nuevo problema de clasificación, por lo que habrá que entrenarla para que el modelo aprenda a describir los objetos. Sin embargo, este cambio de la última capa *fully connected* implica la modificación de las capas posteriores a esta, ya que su número de entradas y salidas debe coincidir con el número de clases del nuevo problema de clasificación.

En el caso de AlexNet [27], aplicar esta técnica supondría cambiar las tres últimas capas del modelo, estas son, la última capa de tipo *fully connected* (*fc8*), que como se ha explicado, es la que describe los objetos a alto nivel, la capa *softmax*, ya que hay que adaptar el número de exponenciales normalizadas a realizar, pues es necesario uno por clase, y finalmente, la capa de salida (*output*), la cual ha debe recibir el mismo número de salidas que emite la capa *softmax*, es decir, una por categoría. La imagen de la figura 4.15 muestra una representación gráfica de estas capas dentro de la arquitectura del modelo AlexNet.

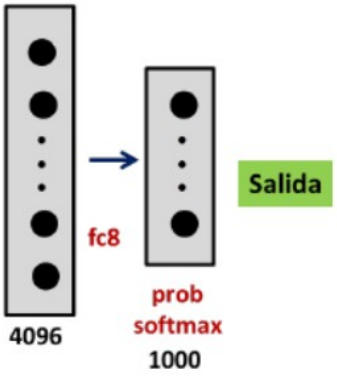

Figura 4.15: *Transfer learning* aplicado a AlexNet. Fuente: Aprendizaje Profundo [3].

Los beneficios de la aplicación de la transferencia del aprendizaje (*transfer learning*) es que acorta el tiempo del entrenamiento y ayuda al modelo a generalizar, pues la red no tiene que aprender a extraer características desde cero, salvo en una capa, y los parámetros del resto de capas de la parte convolucional y del clasificador (MLP), no deberían cambiar mucho para ajustarse al nuevo problema de clasificación.

Para finalizar, se propondrá el siguiente ejemplo didáctico y conceptual de *transfer learning*. Si se tiene una red que es capaz de identificar coches y biclicletas, como se dijo en párrafos anteriores, y el interés de nuestro proyecto cambia de forma que lo que interesa identificar son furgonetas y motos, entonces se pueden conservar lo que la CNN ya aprendió en su parte convolucional y en las primeras capas de su clasificador MLP, ya que sendos problemas de clasificación son similares, lo que se debe a que las categorías de imágenes a reconocer entre ambos problemas, tienen ciertas características similares. Por lo tanto, lo único que habría de hacerse para entrenar el modelo es aplicar *transfer learning*, para lo cual se han de cambiar sus últimas capas a partir de la última de tipo *fully connected*, para adaptarlo al nuevo problema de clasificación.

Capítulo 5 - Detectores de objetos

5.1 Introducción a los detectores de objetos

En este capítulo se realizará una revisión de los diferentes métodos y técnicas para la detección de objetos en imágenes mediante el uso de CNN's, que constituye el objetivo principal del trabajo Para realizar esta tarea se han seguido nuevamente aquellos conceptos pertinentes a esta temática detallados en el trabajo de Pajares et al. [3].

La idea básica de un detector de objetos consiste en el desarrollo de un modelo que pueda analizar varias regiones de interés en una imagen (RoI; *Region of Interest*), para posteriormente recortarlas y pasarlas a una CNN que clasificará su contenido, de esta manera, se averigua la probabilidad de que dichas regiones contengan un objeto de cierta clase dentro de un problema de clasificación.

Este tipo de modelos, llamados detectores, constan de dos partes fundamentales, una columna vertebral o red troncal (CNN), y una cabeza, que se utiliza para poder predecir las clases y las cajas delimitadoras (*bounding boxes*). Algunos detectores añaden una capa más entre la columna vertebral y la cabeza, a la que denominan cuello, que sirve para extraer y mezclar características que provienen de distintas etapas de la red troncal (CNN), con el objetivo de fortalecer las características de los distintos *feature maps*, lo que favorece la detección de los objetos.

La imagen de la figura 5.1 muestra de forma esquemática un resumen de la estructura interna general de los detectores de objetos.

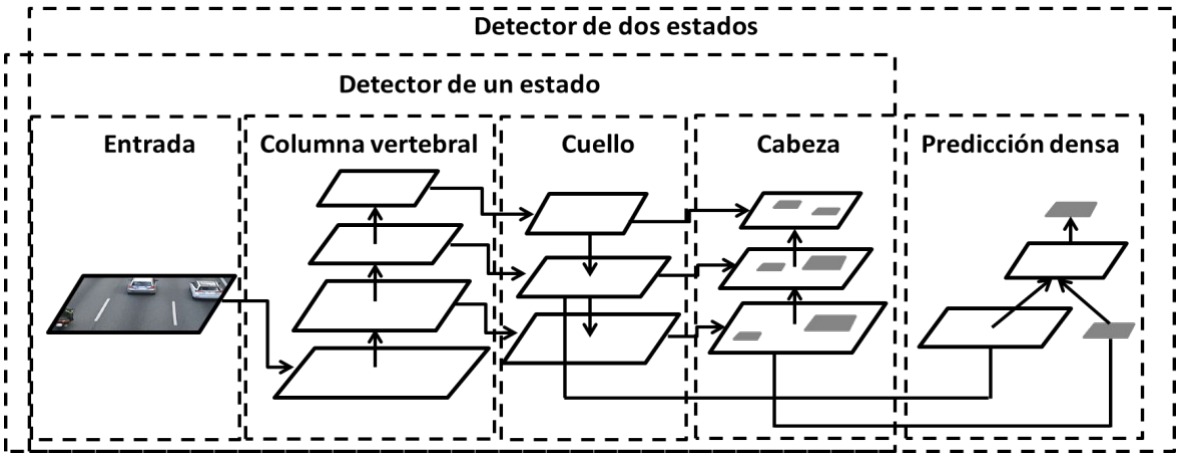

Figura 5.1: Estructura de los detectores de objetos. Fuente: Aprendizaje Profundo [3].

En lo que respecta a la parte de la cabeza se distinguen dos tipos de detectores, los de un estado, que realizan la propuesta de regiones con objetos y la clasificación en una sola etapa (*sparse prediction*), y los de dos estados, que realizan la propuesta de regiones y la clasificación de los objetos en etapas separadas (*dense prediction*). De entre estos detectores, son los de un estado los que proponen un mayor número de regiones candidatas para llevar a cabo la detección de objetos en imágenes.

Cabe destacar que habría que hacer una diferenciación entre los detectores de uno y dos estados que hacen uso de los *anchors boxes*, y los que no hacen uso de ellos.

Para establecer una clasificación más precisa se muestra a continuación el esquema propuesto por Bochkovsky et al. [30], para exponer los diferentes modelos y arquitecturas de detectores, aunque esta categorización no es la única existente.

- Columna vertebral: VGG16, ResNet50, ResNeXt, Darknet.
- Cuello: FPN, BiFPN, PANet.
- Cabeza y predicción dispersa (un estado):
  - Basadas en Anchor Boxes: SSD, YOLO, ReinaNet.
  - No basadas en Anchors Boxes: CornerNet, CenterNet, FCOS.
- Cabeza y predicción densa (dos estados):
  - Basadas en Anchor Boxes: Faster R-CNN, R-FCN, Mask R-CNN.
  - No basadas en Anchor Boxes: RepPoints.

5.2 Conceptos previos

Partiendo de las nociones generales de los detectores de objetos introducidas hasta el momento, se realizará una introducción de aquellos conceptos fundamentales para poder comprender su funcionamiento interno, estos son, los *anchor boxes*, el solapamiento de regiones, y las métricas de desempeño de los detectores.

5.2.1 Anchor Boxes

Proponer una región de interés (RoI) para la búsqueda de objetos en una imagen, o poder ubicar los objetos en ellas, requiere en ambos casos que se enmarque una zona con un rectángulo en base a unas coordenadas, por ejemplo, la esquina superior izquierda y la esquina inferior derecha, o bien punto central y los valoresde ancho y alto.

En cualquiera de los dos casos, estos rectángulos que enmarcan una región reciben el nombre de cajas delimitadoras, o *bounding boxes*. Estos tres términos: rectángulos, cajas delimitadoras, y *bounding boxes* significan lo mismo, sin embargo, el término más popular internacionalmente es *bounding box*.

Por otra parte, los *anchor boxes* son aquellos *bounding box* que se disponen sobre una imagen, y que permiten determinar si la región que enmarcan contienen algún objeto de interés. Desde el punto de vista del proceso de entrenamiento, un *anchor box* se considera como una muestra de ejemplo del conjunto de entrenamiento.

Una vez aclarada la diferencia existente entre estos conceptos, se procede a explicar el uso completo de los *anchor boxes*, de cara al funcionamiento de un detector de objetos. Inicialmente, un detector aplica una estrategia de generación de *anchor boxes*, la cual determinará sus formas y la cantidad de ellos que se emplearán para poder encontrar los objetos de interés en las imágenes. Posteriormente tiene lugar una serie de asignaciones entre los *anchor boxes* y los *ground truth bounding box*, que son aquellos rectángulos que delimitan de forma exacta (según el criterio de un experto) los objetos de interés presentes en una imagen. Acto seguido, para cada *anchor box* se lleva a cabo tanto el etiquetado de la categoría de su *ground truth bounding box* asociado, como el etiquetado de su desplazamiento (*offset*) respecto a su *ground truth bounding box* asociado. Finalmente, un detector empleará un mecanismo de regresión, regulable por mínimos cuadrados, para calcular los *bounding boxes* de los objetos detectados, los cuales son el resultado de una transformación que proyecta los anchor boxes sobre los *ground truth bounding box* objetivos. Cuando hay muchos *bounding boxes* predichos sobre uno o varios *ground truth bounding box* objetivo, estos se pueden filtrar mediante el método denominado supresión no máxima (NMS; *Non-Maximun Supression*), que permite obtener los *bounding boxes* más prometedores.

Como estrategia para la generación de los *anchor boxes*, se propone lo siguiente según el trabajo Zhang et al [31]. Suponiendo que una imagen tenga un tamaño de (***W*** x ***H***), se toma cada píxel (***x***, ***y***) como centro para generar diferentes *anchor boxes*, y una vez definido cada centro, se ha de establecer el ancho (***w***) y alto (***h***) de cada uno, jugando con un conjunto $\boldsymbol{r_1}$, $\boldsymbol{r_2}$, …, $\boldsymbol{r_n}$, de relaciones de aspecto (*aspect ratio*), y con un conjunto $\boldsymbol{s_1}$, $\boldsymbol{s_2}$, …, $\boldsymbol{s_m}$, de factores de escala (*scale factor*) que varíen sus tamaños.

Combinando todas las relaciones de aspecto y los factores de escala para cada píxel como centro, se obtiene una propuesta total de (w x h x n x m) *anchor boxes* para cubrir todos los *ground truth bounding boxes*. Esta estrategia es demasiado costosa computacionalmente, por lo que generalmente, solo se tienen en cuenta un determinado número de píxeles como centros para generar los *anchor boxes*. Para realizar esta tarea se utiliza un factor de desplazamiento d, que establece cuál es la separación vertical y horizontal entre los centros de los *anchor boxes*, lo que hace que descienda el número de los que se utilizan. Como la estrategia aún podría seguir resultando muy costosa, se reduce la cantidad total de combinaciones de r_i y s_j a probar; de esta manera, la estrategia se vuelve aún más liviana. Cabe matizar en este punto, que es muy probable que algunos de los anchor boxes generados sobresalgan del tamaño de la imagen con la que se trabaja, es normal, y cuando ocurre, solo se considera la parte del *anchor box* que interseca con la imagen.

A continuación se muestran en la figura 5.2, diferentes ejemplos de generación de *anchor boxes* con distintas relaciones de aspecto y tamaños, para los objetos de una misma imagen. La distribución inicial de estos es como si formasen un mosaico.

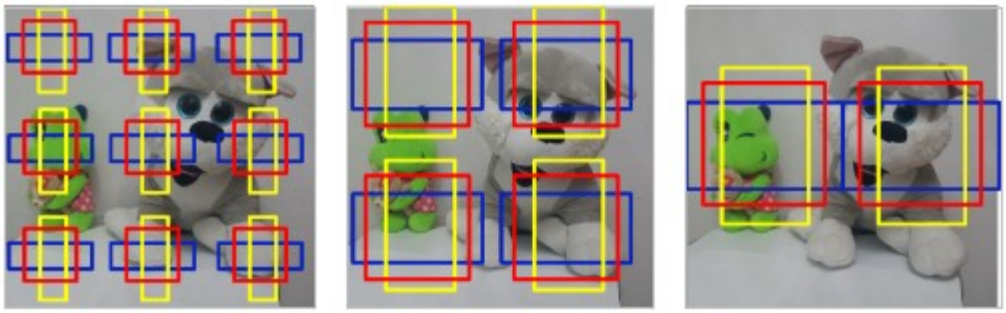

Figura 5.2: *Anchor boxes* para la detección de objetos. Fuente: Aprendizaje Profundo [3].

Para saber cómo se asignan los *anchor boxes* a los *ground truth bounding box*, se procede nuevamente según lo indicado en el trabajo de Zhang et al. [31]:

1. Definir una serie de anchor boxes A_1, A_2, …, A_n, y una serie de *ground truth bounding boxes* B_1, B_2, …, B_m, para dar lugar a la creación de la matriz $X_{n,m}$, para n número de fila, m número de columna, y con $m \leq n$.
2. Rellenar cada intersección $x_{i,j}$ de X con el valor resultante del índice IoU, que se explica más adelante, del *anchor box* A_i y el *ground truth bounding box* B_j.
3. Buscar el elemento $x_{i,j}$ de mayor valor en X, de forma que se obtiene aquel *anchor box* A_i con mayor similitud al *ground truth bounding box* B_j.

4. Descartar todos los elementos de la fila ***i***-ésima y la columna ***j***-ésima de ***X***.
5. Repetir desde el punto 3 mientras no se descarten las ***m*** columnas de ***X***.

Una vez hecho esto, se exploran el resto de los *anchor boxes* no seleccionados de la siguiente manera. Para cada *anchor box* $\boldsymbol{A}_i$ restante, se trata de encontrar el *ground truth bounding box* $\boldsymbol{B}_j$ no asignado con el que guarde el mayor valor del índice IoU, siempre que este supere o iguale un cierto valor umbral prefijado, por ejemplo 0.5.

A continuación, se muestra en la imagen de la figura 5.3, los *ground truth bounding boxes* en blanco, que indican la ubicación de los objetos en la imagen y su categoría dentro del problema de clasificación, y en otros colores, los *anchor boxes* que han sido asignados a los *ground truth bounding boxes*.

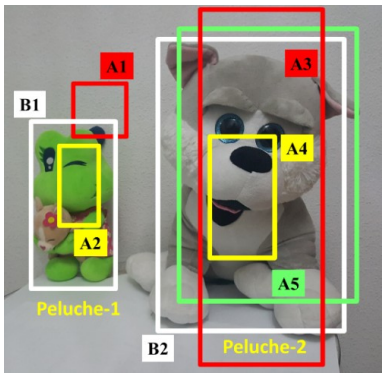

Figura 5.3: *anchors* y *ground truth bounding boxes*. Fuente: Aprendizaje Profundo [3].

Tras esta asignación, a cada *anchor box* se le asigna la categoría correspondiente del *ground truth bounding box* al que está asociado. En este punto cabe destacar que aquellos *anchor boxes* que no se asignan a un *ground truth bounding box* se les asigna la categoría de fondo o *background.* Los *anchor boxes* que reciben esta categoría se denominan negativos, mientras que los que han sido asignados a algunos de los *ground truth bounding box*, se les denomina positivos.

Al mismo tiempo que tiene lugar el etiquetado del *anchor box* con la categoría del *ground truth bounding box* correspondiente, tiene lugar el etiquetado del *offset* del *anchor box* respecto de su *ground truth bounding box* asociado. Este etiquetado se realiza siguiendo las ecuaciones (5.5) a (5.8).

Una vez que acaba el etiquetado de los *anchor boxes*, el modelo predice la categoría de la región de interés (RoI), y emplea un mecanismo de regresión para predecir los *bounding boxes* finales.

Para realizar esta última tarea, se ha de partir de los rectángulos ***P*** (*proposed anchor box*) y G (*ground truth bounding box*), definidos por los vectores (***P***$_x$, ***P***$_y$, ***P***$_w$, ***P***$_h$) y (***G***$_x$, ***G***$_y$, ***G***$_w$, ***G***$_h$), con ***P*** definido por algún método de propuesta de regiones. Un ejemplo de esto es el que se muestra en la imagen de la figura 5.4.

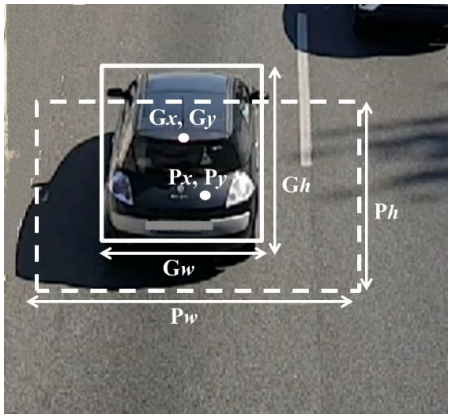

Figura 5.4: ejemplo de *anchor box* p*ropuesto*. Fuente: Aprendizaje Profundo [3].

Según Girshick et al. [32], la transformación de ***P*** a ***G*** debe ser parametrizable en términos de las funciones que regulan la cantidad de desplazamiento (*offset*), es decir las funciones ***d***$_x$(***P***), ***d***$_y$(***P***), ***d***$_w$(***P***), y ***d***$_h$(***P***). Pues de esta forma, pueden revertirse las ecuaciones para despejar dichas funciones, y averiguar cómo calcular el valor de los coeficientes de desplazamiento, que no solo permiten pasar de ***P*** a ***G***, sino que pueden emplearse como los coeficientes objetivo de una regresión, para mejorar la aproximación de los *bounding boxes* predichos mediante la reducción del error por el método de mínimos cuadrados.

Para entender los parámetros que definen la transformación de ***P*** a ***G***, se partirá de un conjunto de entrenamiento de ***N*** pares de ejemplos de la forma ($\mathbf{P}^i$, $\mathbf{G}^i$)$_{i=1..N}$ donde $\mathbf{P}^i$=(***P***$_x^i$, ***P***$_y^i$, ***P***$_w^i$, ***P***$_h^i$) está definido por un punto central y los valores de ancho y alto asociados, de este mismo modo, se define $\mathbf{G}^i$=(***G***$_x^i$, ***G***$_y^i$, ***G***$_w^i$, ***G***$_h^i$).

Las ecuaciones que definen la transformación para pasar de ***P*** a ***G*** son las (5.1), (5.2), (5.3) y (5.4) donde las dos primeras indican traslaciones invariantes a la escala del centro de ***P***, mientras que las dos últimas expresan traslaciones logarítmicas en base al ancho y al alto de ***P***.

$$\widehat{G}_x = P_w * d_x(P) + P_x \qquad (5.1)$$

$$\widehat{G_y} = P_h * d_y(P) + P_y \tag{5.2}$$

$$\widehat{G_w} = P_w * \exp\left(d_w(P)\right) \tag{5.3}$$

$$\widehat{G_h} = P_h * \exp\left(d_h(P)\right) \tag{5.4}$$

Si se despejan las ecuaciones anteriores, para expresarlas en función de los coeficientes $\boldsymbol{d_x(P)}$, $\boldsymbol{d_y(P)}$, $\boldsymbol{d_w(P)}$, y $\boldsymbol{d_h(P)}$, entonces se obtienen las ecuaciones (5.5), (5.6), (5.7), y (5.8), que proporcionarn los *offsets* para ajustar un *anchor box* al *bounding box ground truth* que tiene asignado.

$$t_x = \frac{\left(G_x - P_x\right)}{P_w} \tag{5.5}$$

$$t_y = \frac{\left(G_y - P_y\right)}{P_h} \tag{5.6}$$

$$t_w = \log\left(\frac{G_w}{P_w}\right) \tag{5.7}$$

$$t_h = \log\left(\frac{G_h}{P_h}\right) \tag{5.8}$$

Continuando el trabajo de Girshick, et al. [32], cada función $\boldsymbol{d_*(P)}$, con * como una referencia a cada subíndice ***x***, ***y***, ***h***, ***w***, se describe como una función lineal de las características de la región ***P*** obtenidas de la capa *pool5* de AlexNet [27], denotadas $\boldsymbol{\emptyset_5(P)}$, con un vector de pesos $\boldsymbol{w_*}$ que el modelo aprenderá mediante el método de mínimos cuadrados. La ecuación (5.9) refleja este cálculo, que devuelve los *offsets* estimados de cada componente ***x***, ***y***, ***h***, ***w***, con los que el detector predice *bounding box* para localizar un objeto en la imagen.

$$d_*\left(P^i\right) = \hat{w}_*^T * \emptyset_5\left(P^i\right) \quad (5.9)$$

Por otra parte, el vector de pesos $\boldsymbol{w}_*$ se optimiza como se indica en la expresión de la ecuación (5.10), que muestra cómo se plantea el problema de regresión, en ella, $\boldsymbol{t}_*^i$ son los coeficientes objetivo (*offsets*), que son invariantes a las transformaciones afines, $\boldsymbol{\lambda}$ es el coeficiente de regularización, que se fija a un valor de 1000, y $\boldsymbol{d}_*(\boldsymbol{P})$ se define como en la ecuación (5.9). Gracias a este sistema, la predicción de los *bounding boxes* mejorará a lo largo del entrenamiento según se reduzca el error.

$$w_* = arg_{w_*} min \sum_i^N \left(t_*^i - d_*\left(P^i\right)\right)^2 + \lambda * ||\hat{w}_*||^2 \quad (5.10)$$

Finalmente, es posible que cuando existan muchos *anchor boxes*, el modelo prediga *bounding boxes* muy similares para el mismo objeto (*target*). Cuando esto ocurre, el objetivo es eliminar dichas predicciones similares, para lo cual se puede emplear el algoritmo de supresión no máxima (NMS; *Non-Maximun Supression*), como sigue:

1. Obtener una lista $\boldsymbol{L}$ formada por las predicciones realizadas por un detector para una imagen, donde cada una de ellas está representada por un par de elementos ($\boldsymbol{B}$, $\boldsymbol{p}$), que indican el *bounding box* y la puntuación de confianza (*confidence score*) predichos.
2. Crear una lista vacía de resultados $\boldsymbol{R}$ (*results*).
3. Establecer un cierto umbral $\boldsymbol{T}$ (*threshold*) de IoU (*Intersection over Union*).
4. Averiar el par de $\boldsymbol{L}$ cuyo $\boldsymbol{p}$ es mayor, y pasarlo a $\boldsymbol{R}$.
5. Para el último par añadido a $\boldsymbol{R}$, calcular su índice IoU con los pares que aún residen en la lista $\boldsymbol{L}$, de modo que, si el IoU resultante es superior o igual al umbral $\boldsymbol{T}$, entonces el par de $\boldsymbol{L}$ se elimina.
6. Volver al paso 4 si $\boldsymbol{L}$ no está vacía.

5.2.2 Solapamiento de regiones

Para determinar la similitud existente entre rectángulos independientemente de su unidad de medida, se define el índice IoU (*Intersection over Union*), o de Jaccard [37], como se ve en la ecuación 5.11, donde $\boldsymbol{A}$ y $\boldsymbol{B}$ son dos rectángulos.

$$IoU = \frac{A \cap B}{A \cup B} \tag{5.11}$$

Este índice propone una medida intuitiva de la similitud ente rectángulos a partir de la idea básica de que, si ambos son coincidentes, entonces el solapamiento debe ser de la unidad, pero si no coinciden en absoluto, entonces el solapamiento debe de ser nulo. Esta medida de similitud es ideal en el campo de reconocimiento de objetos, ya que permite calcular cómo de bien está prediciendo el detector aquellos *bounding boxes* con los que espera remarcar los objetos de interés presentes en una imagen. Un ejemplo de esta aplicación puede verse en la siguiente figura 5.5, donde el IoU cuantifica la similitud entre un *bounding box* predicho, y un *bounding box ground truth*.

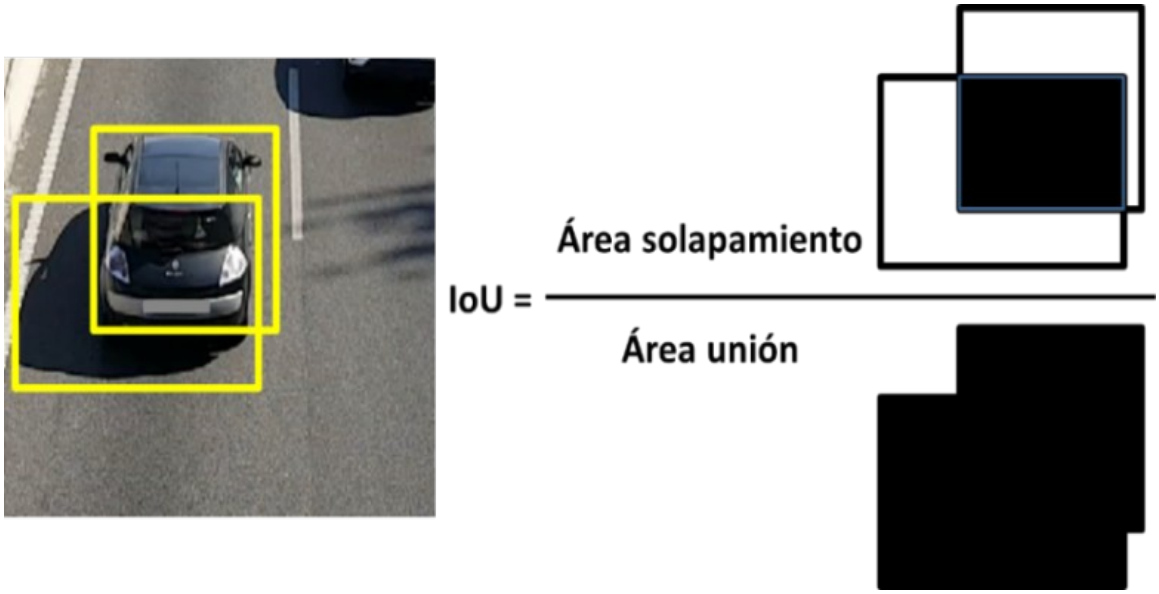

Figura 5.5: Intersección sobre la Unión. Fuente: Aprendizaje Profundo [3].

5.2.3 Métricas de desempeño de detectores

En términos de evaluación, se define la puntuación de confianza (*confidence score*) como la probabilidad p_0, de que un *anchor box* contenga un objeto. Esta puntuación es calculada por el detector para cada *anchor box* al que se le asigna un objeto en una imagen.

En lo que respecta a las detecciones realizadas por un detector, estas se clasifican de la siguiente manera:

- Verdadero Positivo (***VP***) o *True Positive* (***TP***):
  - p_0 es mayor que un determinado umbral.
  - La clase predicha coincide con la especificada por el *ground truth*.
  - El IoU del *bounding box* predicho es mayor que un umbral dado.
- Falso Positivo (***FP***) o *False Positive* (***FP***):
  - p_0 es mayor que un determinado umbral, pero alguna de las siguientes condiciones no se cumple:
    - La clase predicha coincide con la especificada por el *ground truth*.
    - El IoU del *bounding box* predicho es mayor que un umbral dado.
- Verdadero Negativo (***VN***) o *True Negative* (***TN***):
  - Se da cuando se espera, y cumple, que el valor $\mathbf{p_0}$ de una detección sea menor que un umbral, de forma que no se detecta nada.
- Falso Negativo (***FN***) y *False Negative* (***FN***):
  - Se da cuando se espera, pero no se cumple, que el valor p_0 de una detección sea mayor que un umbral para reconocer un objeto.

Teniendo en cuenta las definiciones de los distintos tipos de predicciones que puede realizar un detector, se introducen algunas de las principales métricas empleadas para determinar el desempeño de un detector. Todas ellas devuelven resultados en el intervalo [0, 1], y cuanto más próximos a 1, mejor desempeño del detector.

- *Accuracy*: ¿Del total de predicciones, cuantas han sido correctas?
  - $A = (TP + TN) / (TP + TN + FP + FN)$
- *Precission*: ¿Cuantas predicciones positivas se han clasificado correctamente?
  - $P = (TP) / (TP + FP)$
- *Recall*: ¿Está el detector realizando todas las predicciones que debería?
  - $R = (TP) / (TP + FN)$
- *Specifity*: ¿Funciona bien el detector cuando no debería predecir nada?
  - $R = (TN) / (TN + FP)$

Otra métrica es la denominada, precisión promedio (**AP**; *Average Precision*), que se calcula como el área bajo la curva ***PR***, de *precission* frente a *recall*, que se usa para evaluar la precisión de los detectores en términos generales. Para obtener la curva, ***R*** se sitúa sobre el eje de abscisas y ***P*** se sitúa sobre el eje de ordenadas, de forma que para un valor ***r***, entre 0 y 1, en ***R***, se obtiene el correspondiente valor ***p***(***r***) en ***P***.

Para calcular el valor del área bajo la curva ***PR*** y obtener ***AP***, se aplica la ecuación (5.12), la cual indica que como el coste computacional de su cálculo es elevado, lo que se recurre a hacer en su lugar es un cálculo discreto interpolado. Dicha interpolación se materializa en la ecuación (5.13), que indica que para cada ***r*** de entrada a la función, se obtiene como valor de salida el máximo en el intervalo ***r***, ***r'***, donde ***r*** ya viene dado, y ***r'*** es un valor de la abscisa que obtiene un ***p***(***r***) máximo en relación a todos los valores ***r*** de la abscisa posteriores a él.

$$AP = \int_0^1 precission(r)\,dr \rightarrow AP = \sum_{i=1}^{n-1} (r_{i+1} - r_i) * p_{\text{int}}(r_{i+1}) \qquad (5.12)$$

$$p_{\text{int}}(r) = max_{r' \geq r}\, p(r') \qquad (5.13)$$

El cálculo de ***AP*** solo involucra una clase del problema de detección, pero como en estos normalmente hay un total de ***K*** clases, se opta por el cálculo de la precisión promedio (***mAP***; *mean Average Precission*), definida en la ecuación (5.14).

$$mAP = \frac{1}{K} * \sum_{i=1}^{K} AP_i \qquad (5.14)$$

También existe otra métrica muy utilizada, el recall promedio (**AR**; *Average Recall*), que es el *recall* ***R*** promediado sobre todos los IoU $\in$ [0.5, 1.0], el cual se define según la ecuación (5.15), es decir, dos veces el área bajo la curva recall-IoU, donde ***o*** se refiere a IoU, y recall(***o***) es el correspondiente *recall*.

$$AR = 2 * \int_{0.5}^{1} recall(o)\,do \qquad (5.15)$$

Como pasaba anteriormente con el cálculo de ***AP***, el cálculo de ***AR*** solo involucra a una única clase del problema de detección, pero como en estos casos suele haber un total de ***K*** clases, se opta por calcular la media del *recall* promedio (***mAR***; mean *Average Recall*), esta se calcula tal y como se indica en la ecuación (5.16), donde ***K*** es nuevamente el total de clases existentes en el problema de detección.

$$mAR = \frac{1}{K} * \sum_{i=1}^{K} AR_i \tag{5.16}$$

5.3 Modelos para la detección de objetos

En este apartado se realiza una introducción a los detectores de objetos que han sido de interés en el ámbito técnico de este trabajo, que en este caso, han sido algunos de los detectores más populares de uno y dos estados que se basan en el uso de *anchor boxes* como mecanismo para la detección, concretamente son:

- Detectores de dos estados: R-CNN, Fast R-CNN, Faster R-CNN y MASK R-CNN.
- Detectores de un estado: SSD y YOLO.

5.3.1 Region Based CNN (R-CNN)

Este modelo se basa en el trabajo de Girshick et al. [32], y su funcionamiento interno es como sigue:

- Se realiza una selección de regiones para proponer aquellas que tengan una cierta probabilidad de contener un objeto. Algunos de los métodos que más se emplean para llevar a cabo esta tarea son:
  - Agrupaciones de regiones por color, textura, geometría o concentración de bordes [33].
  - Distinción de zonas por diferenciación con el contorno circundante, donde destacan las conocidas como *objectness* [34].
  - Agrupaciones de bordes (*edge-boxes*) [35].
  - Agrupación de pequeñas subregiones (*regionlets*) [36].

- Una vez seleccionadas las regiones, cada una es recortada y redimensionada al tamaño de la entrada de la CNN troncal del detector, que se le pasan como entrada. Suponiendo que la red sea AlexNet [27], figura 4.1, el proceso es como sigue:
  - Emplea los rectángulos de las regiones propuestas para poder estimar los parámetros (offsets) de los bounding box por regresión lineal, utilizando para ello las coordenadas que comprenden las características de la capa pool5. La parte de la red que realiza esta tarea se conoce como Regresor.
  - Obtiene como salida de su capa *fc6,* un vector de características (4096 x 1) para cada región de interés (RoI; *Region of Interest*), que le es pasado a una serie de capas *fully connected* que forman el Clasificador, el cual trata de averiguar la categoría de un objeto. Una alternativa a estas capas para realizar la clasificación sería una Máquina de Vectores de Soporte (SVM; *Support Vector Machine*) [38, 39].

La figura 5.6 representa conceptualmente este funcionamiento del detector R-CNN.

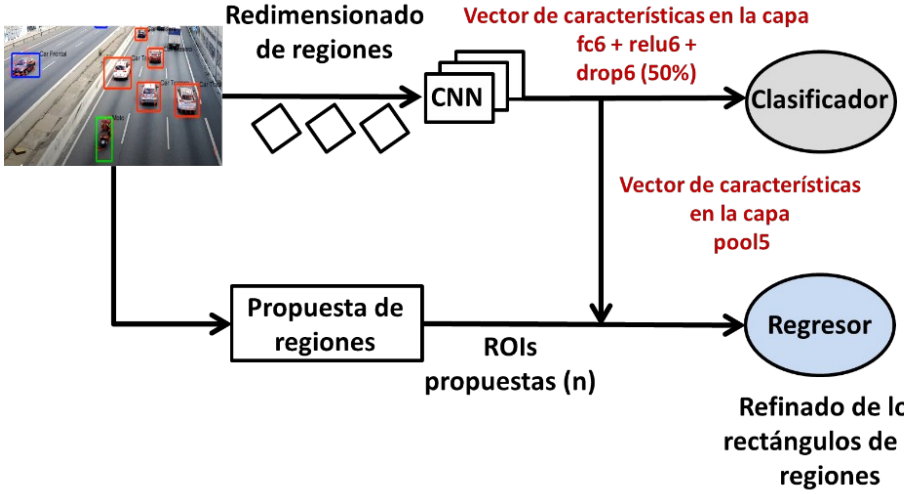

Figura 5.6: Arquitectura de un detector R-CNN. Fuente: Aprendizaje Profundo [3].

Según Pang et al. [40], en relación a los detectores de tipo R-CNN, el éxito de su entrenamiento depende de tres aspectos:

- Que las muestras de las regiones seleccionadas sean representativas.
- Que las características visuales extraídas se utilicen por completo.
- Que la función objetivo sea óptima.

Según los autores existe un desequilibrio en el entrenamiento debido a estos tres aspectos, por lo que proponen la incorporación de tres elementos novedosos:

- Muestreo equilibrado. Extracción de muestras de acuerdo a su índice IoU con el *ground truth* asignado.
- Pirámide de características balanceadas. Esta fortalece las características de múltiples niveles mediante el uso de características semánticas balanceadas.
- Función de pérdida L_1 balanceada. Promueve gradientes cruciales con los que reequilibra la clasificación, y la localización tanto general como precisa.

Una diferencia significativa de este detector con respecto a los que se introducirán a continuación es que este modelo realiza un procesamiento por cada una de las regiones de interés (RoI) propuestas, mientras los siguientes solo procesan la imagen completa una única vez.

5.3.2 Fast R-CNN

El anterior modelo de detector R-CNN funciona adecuadamente, sin embargo, su ajuste requiere de diversos recursos computacionales para entrenar sus partes:

- La CNN, para aprender a extraer las características de los objetos.
- El Regresor, para estimar los rectángulos que delimitan los objetos.
- El Clasificador, para predecir la clase a la que pertenece cada objeto.

Otro problema de este detector es la lentitud que conlleva la estrategia que aplica, ya que por cada región propuesta se realiza un procesamiento en su CNN troncal. Por esta razón, y las anteriores, el detector es un modelo costoso en términos de tiempo de cómputo y de espacio de memoria. Todos estos aspectos son destacados en el que fue el siguiente trabajo de Girshick [41], quien para resolver estas problemáticas, planteó un nuevo enfoque.

En el nuevo modelo concebido solo se ejecuta la CNN troncal una vez por imagen, para posteriormente compartir el resultado del proceso de convolución de la red, en lugar de realizarlo para todas y cada una de las regiones propuestas. Esta es la idea esencial de un detector de tipo Fast R-CNN.

El detector Fast R-CNN toma como entrada una imagen, y un conjunto de RoIs para la propuesta de objetos, que son sugeridas con los métodos vistos en R-CNN, o similares. Suponiendo que la CNN del detector sea AlexNet [27], esta procesa posteriormente la imagen con sus respectivas capas de convolución, y obtiene el mapa de características (*feature map*) de entrada a la capa de *pool5*. Dicho tensor de salida está definido como un volumen de dimensiones (***W*** x ***H*** x ***K***), que está definido por su ancho, alto, y la cantidad de sub-mapas que lo conforman. En este punto, para cada una de las RoI del tensor anterior, se extraen sus características. A la parte del detector que realiza esta tarea se la conoce por ello como extractor de características, que conoce cuál es la relación entre los píxeles del tensor (imagen) de entrada, y los del tensor salida.

Por ejemplo, si partimos de una imagen de entrada de AlexNet (227 x 227 x 3), y el tamaño del *feature map* obtenido en *pool5* es de (13 x 13 x 256), se dividen las dimensiones de ancho, y de alto, obteniéndose aproximadamente 17 en ambos casos, por lo que para pasar de una RoI en la capa de entrada, definida por la tupla ($\boldsymbol{X_{ROI}}$, $\boldsymbol{Y_{ROI}}$, $\boldsymbol{W_{ROI}}$, $\boldsymbol{H_{ROI}}$), a su equivalente proyectada en la salida de la última capa de convolución, denotada por la tupla ($\boldsymbol{X_{PROJ}}$, $\boldsymbol{Y_{PROJ}}$, $\boldsymbol{W_{PROJ}}$, $\boldsymbol{H_{PROJ}}$), lo que hay que hacer es dividir cada una de las componentes de la tupla de entrada entre 17.

Una vez extraídas las características de una RoI proyectada en la capa de red *pool5*, estas son pasadas a una capa de tipo SPP (*Spatial Pyramid Pooling*) [42], identificada como RoI *Pool*. Esta capa lo que hace es aplicar a cada RoI ***N*** ventanas de *max pool* un total de ***K*** veces, es decir, tantas veces como sub-mapas de características componen la profundidad del tensor obtenido a la salida de *pool5*.

Esta operación es muy interesante, pues permite obtener un vector de datos con un tamaño fijo, que no depende de las dimensiones (***W*** x ***H***) del tensor de *pool5*, si no solo de ***K***, lo que significa que la red se vuelve independiente del tamaño de la imagen de entrada.

La imagen de la figura 5.7 muestra cómo opera la capa *Spatial Pyramid Pooling* según lo especificado en el trabajo de He et al. [42], la cual describe cómo al tensor de entrada (***W*** x ***H*** x ***K***) de esta capa se le aplican un total de tres ventanas de *max pooling* de dimensiones 1x1, 2x2, y 4x4.

En el caso de AlexNet [27], el tensor de entrada tendría las dimensiones (13 x 13 x 256), por lo que el vector de salida que recibirían las capas de clasificación, que vendrían a continuación, tendría el tamaño de la suma de los vectores: (1 x 256), (4 x 256), y (16 x 256). Lo que resultaría en un vector de características de 1 x (1 x 256 + 4 x 256 + 16 x 256) = 1 x (256 x (1 + 4 + 16)) = 1 x (256 x 21) = (1 x 5376).

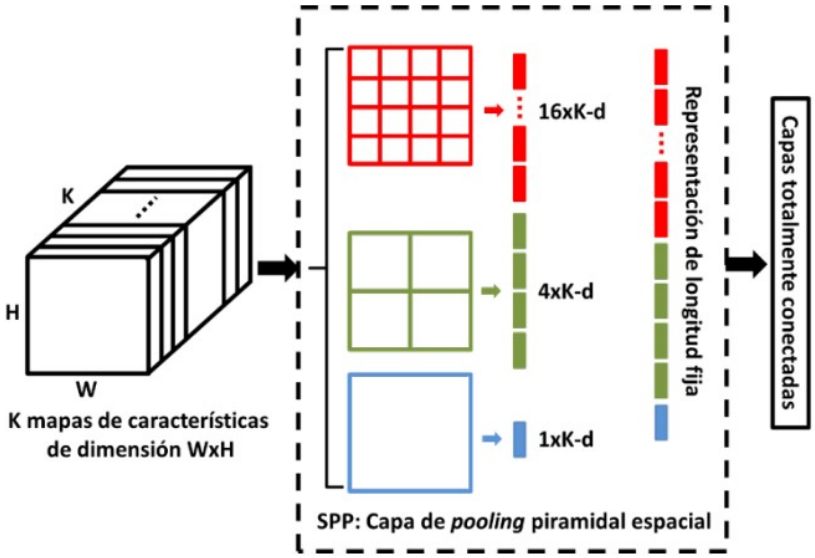

Figura 5.7: Capa *Spatial Pyramid Pooling*. Fuente: Aprendizaje Profundo [3].

La figura 5.8 muestra cómo se aplica una ventana de *max pooling* de 4x4 sobre una RoI (*Region of Interest*) proyectada en la capa *pool5* de la CNN AlexNet.

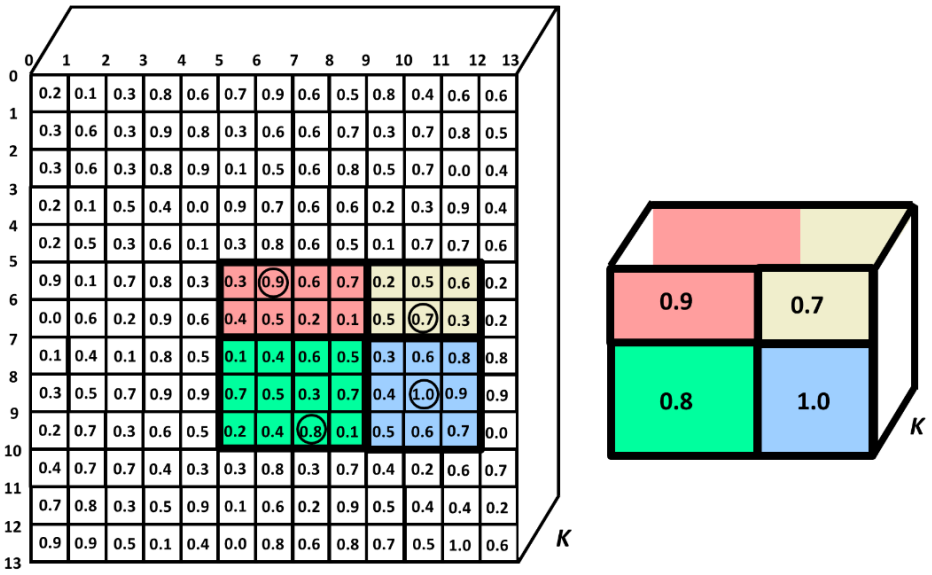

Figura 5.8: Funcionamiento de *Spatial Pyramid Pooling*. Adaptada de: Aprendizaje Profundo [3].

La figura 5.9 contiene un resumen conceptual de todo el procesamiento interno que tiene lugar en un detector Fast R-CNN hasta la capa SPP (*Spatial Pyramid Pooling*).

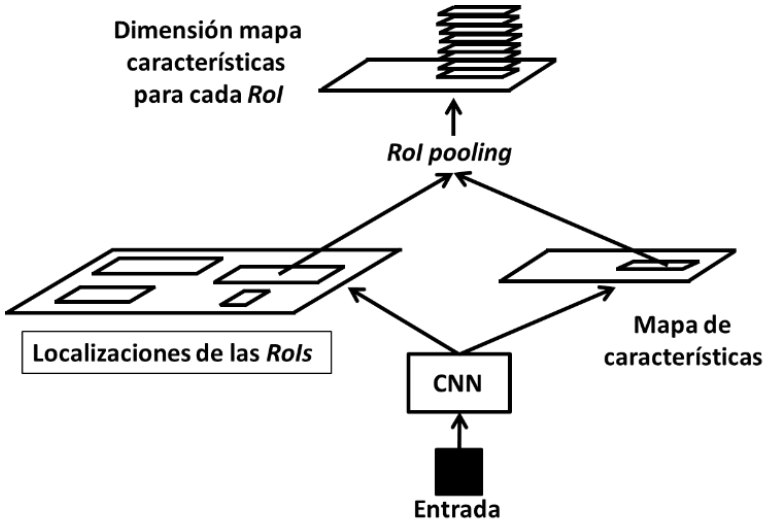

Figura 5.9: Resumen de Fast R-CNN hasta SPP. Fuente: Aprendizaje Profundo [3].

Finalmente, las capas de clasificación que se mencionaron anteriormente son una secuencia de capas de tipo *fully connected* que terminan bifurcándose en dos ramas. Por un lado, una constituye la capa de clasificación (Clasificador), la cual termina en una capa *softmax*, y por otro lado, la capa de regresión (Regresor), que estima los parámetros de los *bounding boxes* que enmarcan los objetos.

La a figura 5.10 muestra conceptualmente el funcionamiento de Fast R-CNN.

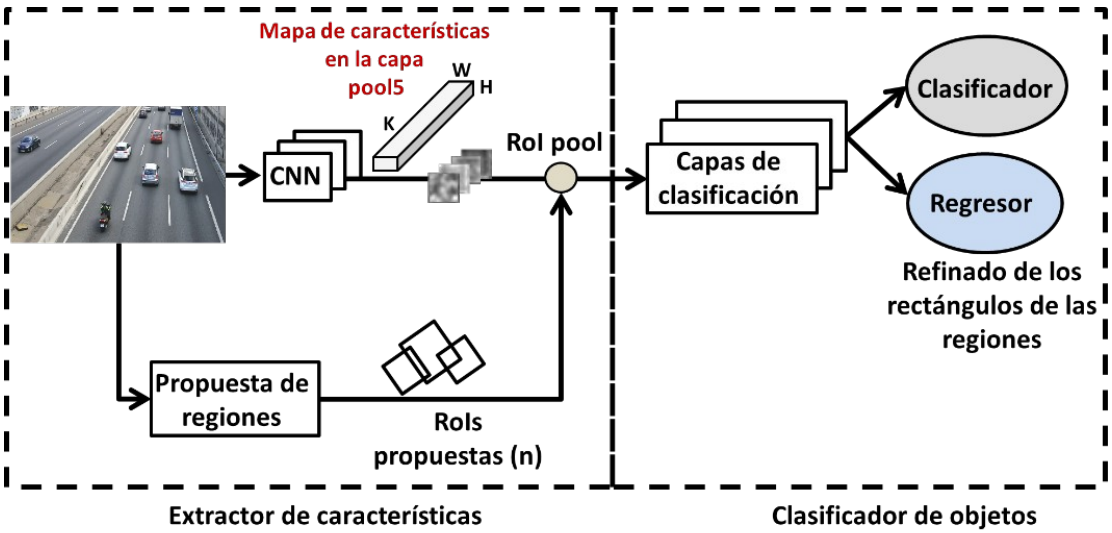

Figura 5.10: Arquitectura de un detector Fast R-CNN. Fuente: Aprendizaje Profundo [3].

Según el trabajo de Girshick [41], en sus experimentos los pesos del detector se obtuvieron mediante un entrenamiento en el que se aplicaba el método de minimización del error, y dentro de este, mediante los algoritmos SGD (*Stochastic Gradient Descent*) y BPE (*Backpropagation Error*) [21]. El conjunto de datos para el entrenamiento proviene del *dataset* PASCAL VOC [43, 44], para el que se emplearon un total de 60.000 *mini-batches*, que son muestreados jerárquicamente, primero muestreando ***N*** imágenes, y luego (***R*** / ***N***) RoIs (***R***) por cada imagen. Cuando el valor de ***N*** es pequeño, el proceso de computación de los *mini-batches* disminuye. Por ejemplo, si se tiene ***N*** = 128, y ***R*** = 128, entonces el procesamiento es más costoso, ya que hay que cargar en memoria y procesar una RoI por imagen. Pero con ***N*** = 2, y ***R*** = 128, la mejora (*speedup*) es significativa, ya que en una sola imagen se cargan en memoria 64 RoIs que ahora pueden procesarse.

5.3.2.1 Función de pérdida multi-task

Como se explicó anteriormente, un detector de tipo Fast R-CNN tiene dos capas de salida hermanas. La primera proporciona una distribución de probabilidad por cada RoI sobre ***K***+1 clases de objetos, donde la clase adicional es la clase de fondo (*background*). La probabilidad de pertenencia a cada clase se calcula en la última capa del Clasificador, que es una capa de tipo *softmax* de ***K***+1 salidas, que recibe sus ***K***+1 entradas de una capa de tipo *fully connected*. La segunda capa hermana es la que genera los desplazamientos (*offsets*) de los *anchor boxes* por medio de regresión. Los desplazamientos se calculan para cada clase en la capa del Regresor, y estos se representan de la forma $\boldsymbol{t^k} = (\boldsymbol{t^k_x}, \boldsymbol{t^k_y}, \boldsymbol{t^k_w}, \boldsymbol{t^k_h})$, para ***K*** clases.

La función de pérdida *multi-task* ***L***, se define para entrenar de forma conjunta tanto el Clasificador como el Regresor del detector, esta recibe el nombre de pérdida $\boldsymbol{L_1}$ balanceada (*balanced* $\boldsymbol{L_1}$ *loss*), y se define como se indica en el trabajo de Pang et al. [40], según la ecuación (5.17). En ella, la expresión $[\boldsymbol{u} \geq 1]$ es la función del corchete de Iverson, empleada para omitir las RoIs de fondo, pues esta toma el valor 1 si la clase no es el *background*, y 0 en caso contrario, por otra parte, $\boldsymbol{\lambda}$ controla el equilibrio entre las dos funciones de pérdida, su valor habitual es 1.

$$L\left(p, u, t^u, v\right) = L_{cls}\left(p, u\right) + \lambda * \left[u \geq 1\right] * L_{reg}\left(t^u, v\right) \qquad (5.17)$$

La función de pérdida relativa a la clasificación de objetos que realiza el detector es de carácter logarítmico, y se define según la ecuación (5.18). En ella, la variable ***u*** es el identificador de la clase verdadera, y p_u es la probabilidad de pertenencia a dicha clase. En este punto, dada la clase ***u***, la probabilidad de pertenencia predicha para ella p_u debería ser la unidad, por ello, a medida que p_u se acerque a uno, el valor que se obtendrá en la ecuación (5.18) disminuirá, y viceversa, cuanto más se aleje p_u de la unidad, mayor será el valor devuelto por la ecuación. La figura 5.11 muestra gráficamente la forma en que varían los valores de la ecuación (5.18).

$$L_{cls}(p,u) = -\log p_u \tag{5.18}$$

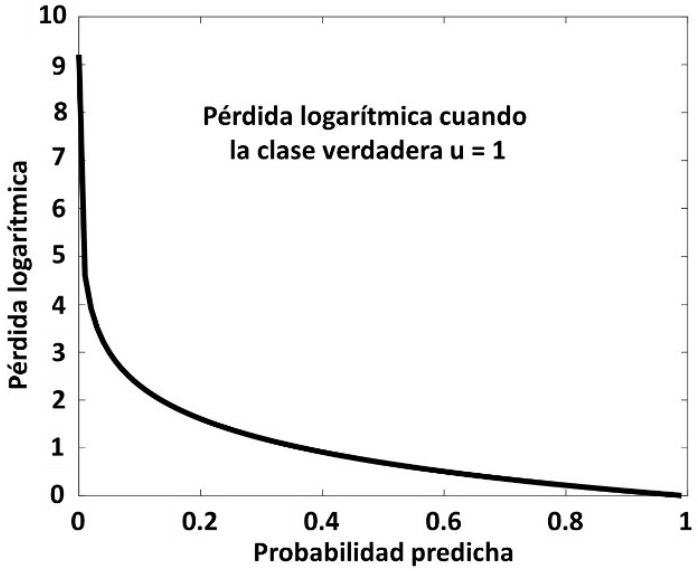

Figura 5.11: Función de pérdida logarítmica. Fuente: Aprendizaje Profundo [3].

La función de pérdida relacionada con la generación de los *bounding boxes* con los que enmarcar los objetos se define según la ecuación (5.19), donde los *offfsets* predichos y los objetivos de regresión han sido definidos como $\boldsymbol{t^u} = (\boldsymbol{t^u_x}, \boldsymbol{t^u_y}, \boldsymbol{t^u_w}, \boldsymbol{t^u_h})$ y $\boldsymbol{v} = (\boldsymbol{v_x}, \boldsymbol{v_y}, \boldsymbol{v_w}, \boldsymbol{v_h})$, estando los valores de este último vector **v** normalizados. En cuanto a $\boldsymbol{S_{L1}}$, esta es una función conocida como $\boldsymbol{L_1}$ suavizada, o *smooth* $\boldsymbol{L_1}$, la cual se ve menos afectada por valores atípicos (*outliers*), y a problemas afines a estos.

$$L_{reg}(t^u, v) = \sum_{i \in \{x,y,w,h\}} S_{L1}(t_i^u - v_i) \; ; \; S_{L1} = \begin{cases} 0.5 * x^2 si |x| < 1 \\ |x| - 0.5 \; si \; |x| \geq 1 \end{cases} \tag{5.19}$$

5.3.2.2 Muestreo del mini-batch

Cada *mini-batch* se construye con $N = 2$ imágenes de dimensión $R = 128$, por lo que se muestrean 64 RoIs por cada imagen. Siguiendo la propuesta de Girshick [41], se realiza una selección del 25% de las RoIs propuestas para la detección de objetos cuyo índice de solapamiento IoU con un *ground truth bounding box* sea superior o igual a 0.5. Estas RoIs son etiquetados como una clase de objeto, y no como fondo, por lo que la función corchete de Iverson $[u \geq 1]$ se evalúa a 1 para ellas. Las RoIs restantes se muestrean a partir de aquellas regiones cuyo índice de solapamiento IoU con un *ground truth bounding box* se encuentra en el intervalo [0.1, 0.5), como se propone en He et al. [42]. Estos constituyen los ejemplos de fondo, por lo que la función corchete de Iverson $[u \geq 1]$ devuelve 0 para ellos. Durante este entrenamiento, se recurre a la técnica del aumento de datos, o *data augmentation*, para poder conseguir un mayor número de ejemplos, las imágenes fueron rotadas horizontalmente con una probabilidad de 0.5, no se realizaron otras transformaciones adicionales.

5.3.2.3 Retropropagación a través de la capa RoI pool

La retropropagación del error encamina las derivadas a través de la capa RoI pool, o SPP (*Spatial Pyramid Pooling*). Para el cálculo de su derivada se ha de considerar:

- r: una RoI concreta.
- j: un índice que concreta una sub-ventana en una operación de *max pooling*.
- $x_i \in \mathbb{R}$: la i-ésima entrada (activación) de cierta r.
- $R(r, j)$: los índices de las entradas x_i de j al hacer *max pooling* sobre r.
- $h(r, j)$: índice de $R(r, j)$ asociado a la entrada x_i cuyo valor es el máximo de j.
- y_{rj}: la salida de j al realizar *max pooling* en r; $y_{rj} = y_{h(r, j)}$.

Habiendo considerados los factores anteriores, la derivada de la capa SPP se calcula como se indica en la ecuación (5.20).

$$\frac{\partial L}{\partial x_i} = \sum_r \sum_j \frac{\partial L}{\partial y_{rj}} \; ; \; i = h(r, j) \tag{5.20}$$

5.3.2.4 Hiperparámetros SGDM

Las capas *fully connected* utilizadas para la clasificación mediante *softmax*, y la regresión de los *bounding boxes*, se inicializan mediante distribuciones gaussianas de media 0 y desviaciones estándar 10^{-2} y 10^{-3}. Los parámetros de sesgo (*bias*) se inicializan a 0. Todas las capas utilizan una razón de aprendizaje global de 10^{-3}. Los entrenamientos con los *datasets* de imágenes VOC07 y VOC012 de PASCAL VOC (*Visual Object Classes*) [43, 44], se ejecutaron para 30.000 *mini-batches*. Posteriormente, se disminuyó la razón de aprendizaje global a 10^{-4} y se volvió a entrenar, esta vez para un total de 10.000 *mini-batches*. El factor de fuerza (*momentum*) aplicado en SGDM [21], fué de 0,9.

5.3.3 Faster R-CNN

Este detector funciona de forma parecida a Fast R-CNN, con la diferencia de que, en lugar de realizar la propuesta de regiones mediante un selector externo, ahora se introduce una Red para Propuesta de Regiones (RPN; *Region Proposal Network*) que comparte las características convolucionales generadas a partir de la imagen, lo que mejora el rendimiento. Esta red es de tipo convolucional, se entrena para generar propuestas con la máxima calidad posible, prediciendo tanto la posibilidad de existencia de los objetos, como los rectángulos que los envuelven.

La imagen de la figura 5.12 muestra la arquitectura del modelo de detector Faster R-CNN [14], muy similar a la del modelo de detector Fast R-CNN.

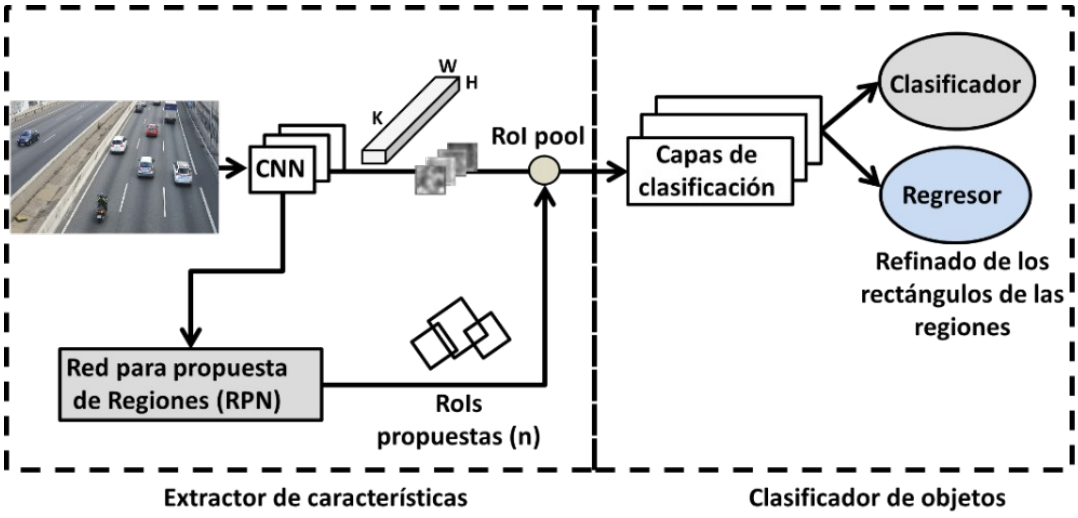

Figura 5.12: Arquitectura del detector Faster R-CNN. Fuente: Aprendizaje Profundo [3].

5.3.3.1 Funcionamiento de la red RPN

Sobre el mapa de características de entrada (***M*** x ***N*** x ***P***) obtenido en la última capa convolucional compartida, se desliza una pequeña ventana de tamaño (***n*** x ***n***) para realizar por cada posición de la ventana sobre el mapa, una propuesta de ***k*** *anchor boxes*. Según el trabajo de Ren et al. [14], ***n*** = 3 y ***k*** = 9, por lo que el número total *de anchor boxes* que se propondrán será de (***M*** x ***N*** x ***k***).

Cada ventana deslizante se transforma por cada localización en un vector de menor dimensión D-d, por ejemplo 512-d para VGG, seguido de una operación ReLU. Para obtener los vectores se realizan ***P*** convoluciones con ***P*** *kernels* de tamaño (***n*** x ***n***), por ventana.

Cada vector obtenido es operado en dos capas convolucionales hermanas. La capa de regresión (REG) aplica cada vector (4 x ***k***) *kernels* de (1 x 1 x ***P***), para obtener un tensor de (***M*** x ***N*** x 4***k***) que contiene las coordenadas de los **k** *anchor boxes*, denotados (***x***, ***y***, ***w***, ***h***), que enmarcan las regiones de interés. Y la capa de clasificación (CLS), que aplica al vector (2 x ***k***) *kernels* con tamaño (1 x 1 x ***P***), para obtener un tensor de tamaño (***M*** x ***N*** x 2***k***) que contiene 2***k*** valores de probabilidad que determinen la presencia (*foreground*) o ausencia (*background*) de un objeto en cada región de interés propuesta (*anchor box*).

La figura 5.13 muestra un resumen de la arquitectura y el funcionamiento de la red RPN, donde CNN-c es lo que se denomina red de cabecera, que es la red de la que se obtiene el mapa de características de entrada.

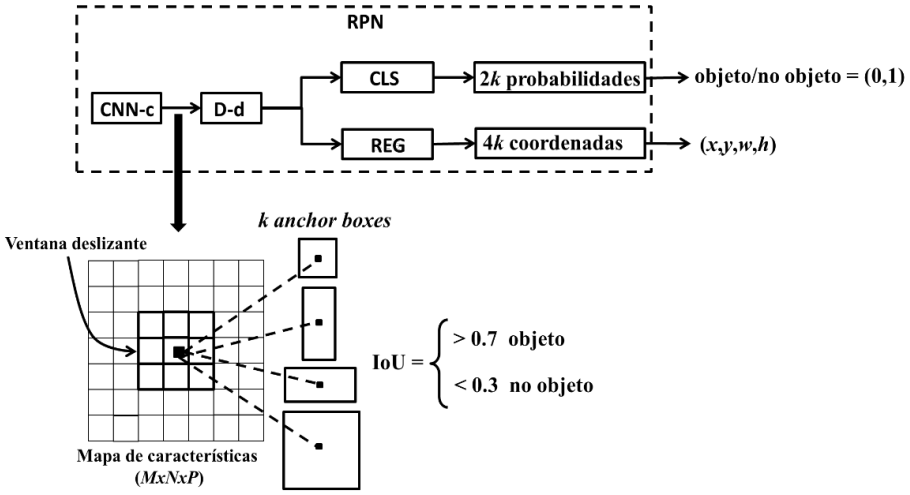

Figura 5.13: *Region Proposal Network*. Fuente: Aprendizaje Profundo [3].

5.3.3.2 Generación de regiones de la red RPN

En cada localización de la ventana deslizante se realizan múltiples propuestas RoIs, que son denominadas *anchor boxes*, siendo el máximo ***k*** por cada ventana. Cada uno de estos ***k*** *anchor box* es generado centrado respecto de la ventana, y posee una escala y una relación de aspecto, por ello, dado un mapa de características convolucional con un tamaño de (***M*** x ***N***), se obtendrán un total de (***M*** x ***N*** x ***k***) regiones de interés (*anchor boxes*).

Según lo indicado en el trabajo de Ren et al. [14], se utilizan 3 *anchor boxes* a los que se les dan 3 tamaños y relaciones de aspecto diferentes, lo que hace un total de ***k*** = 9 *anchor boxes* a emplear por cada ventana.

Después de generarse los (***M*** x ***N*** x ***k***) *anchor boxes* correspondientes, a cada uno de ellos se le asigna una etiqueta de carácter binario que indica si este contiene o no un objeto. Para la asignación de etiquetas se tienen en cuenta estos los criterios que se detallan a continuación:

- Se concede una etiqueta positiva a los *anchor boxes* con mayor índice IoU, al solaparse estos con un *ground truth bounding box*. También le es concedida a los anchor boxes cuyo índice IoU al solaparse con los *ground truth bounding boxes* es superior a 0,7.
- Se asigna una etiqueta negativa a los *anchor boxes* cuyo IoU, al solaparse con los *ground truth bounding boxes*, es inferior a 0,3.
- Los anchor boxes ni positivos ni negativos, no contribuyen de ninguna forma de cara al entrenamiento de la RPN.

5.3.3.3 Función de pérdida de la red RPN

Teniendo en cuenta lo comentado en el apartado 5.3.3.1, la minimización del error cometido se lleva a cabo a través de la función de la ecuación (5.21), que posee unas características similares a la función de error del detector Fast R-CNN, la cual puede verse en la ecuación (5.17). Esta función de error, al igual que la del modelo Fast R-CNN, se divide a su vez en las correspondientes funciones relativas al error cometido en la clasificación ($\mathbf{L_{cls}}$), y el cometido al generar los *bounding box* mediante regresión ($\mathbf{L_{reg}}$).

$$L\left(p_i, t_i\right) = \frac{1}{N_{cls}} \sum_i L_{cls}\left(p_i, p_i^*\right) + \frac{1}{N_{reg}} \sum_i p_i^* * L_{reg}\left(t_i, t_i^*\right) \quad (5.21)$$

De esta ecuación (5.21), $\boldsymbol{p_i}$ es la probabilidad predicha de que el $\boldsymbol{i}$-ésimo *anchor box* sea un objeto. La etiqueta $\boldsymbol{p^*_i}$ es 1 si el *anchor box* es positivo, y 0 de otro modo.

Por otra parte, $\boldsymbol{t_i}$ representa las 4 coordenadas del *bounding box* predicho, mientras que $\boldsymbol{t^*_i}$ son las coordenadas de un *ground truth bounding box*.

$\boldsymbol{L_{cls}}$ es la función de pérdida logarítmica binaria sobre dos clases, objeto frente a no objeto, definida según la ecuación (5.22), con ella lo que se hace es penalizar las desviaciones de las predicciones respecto de los *ground truth*.

$$L_{cls} = -p_i^* * \log\left(p_i\right) - \left(1 - p_i^*\right) * \log\left(1 - p_i\right) \quad (5.22)$$

En cuanto a la pérdida de la regresión $\boldsymbol{L_{reg}}$, esta se calcula según ecuación (5.19), que es la función $\boldsymbol{L_1}$ suavizada (*smooth* $\boldsymbol{L_1}$).

Según lo indicado en el trabajo de Ren et al. [14], el valor establecido para $\boldsymbol{N_{cls}}$ fue 256, la dimensión de un *mini-batch*, mientras que a $\boldsymbol{N_{reg}}$ se le asignó 2.400, que es el número de localizaciones de los *anchor boxes*. En lo referente al valor $\boldsymbol{\lambda}$, este fue fijado a 10 de forma que los términos ***REG*** y ***CLS*** se compensasen, aunque los resultados demostraron ser poco sensibles a los valores de este parámetro.

5.3.3.4 Entrenamiento de la red RPN

Siguiendo con el trabajo de Ren et al. (2017), el modelo puede entrenarse mediante BPE (*Backpropagation Error*) y SGDM (*Stochastic Gradient Descent with Momentum*) [21], este último con un factor de momento de 0,9. Para esto se sigue la estrategia conocida como *image concentric* [41].

Los pesos de las capas nuevas se inicializan utilizando una distribución gaussiana de media 0, y desviación estándar 0,01. Los pesos de las capas compartidas se inicializan a partir del modelo ImageNet de Russakovsky et al. [44]. Se utilizó una razón de aprendizaje de 10^{-3} para (6×10^4) *mini-batches*, y 10^{-4} para los siguientes (2×10^4) *mini-batches*. El set de imágenes utilizado fué PASCAL VOC (*Visual Object Classes*) [43, 44].

5.3.4 Mask R-CNN

Este detector propuesto por He et al. [46], es una variación del Faster R-CNN, que incorpora una rama en paralelo con fines de segmentación de objetos. Esta extensión añade a las capacidades del detector de clasificar y localizar objetos, la segmentación semántica, es decir, la capacidad de clasificar cada píxel de una RoI en un conjunto de categorías, por lo que ya no sería completamente necesario localizar objetos mediante *bounding boxes*.

La operación de segmentación se realiza en una rama paralela a la de detección y localización de objetos. Esta rama recibe el nombre de máscara (*mask*), y la forma una pequeña Red Totalmente Conectada (FCN; Fully Connected Network) que se aplica a cada RoI para producir una máscara de segmentación, píxel a píxel.

La salida de la rama *mask* es distinta de las salidas de la clasificación y regresión, requiere un ajuste espacial más fino de los objetos a nivel de píxel, razón por la que se incluye un módulo de alineamiento píxel a píxel, en lugar de la capa RoI pooling de los modelos Fast R-CNN y Faster R-CNN.

El entrenamiento de este modelo solo añade un pequeño sobrecoste en relación al entrenamiento del detector Faster R-CNN.

La imagen de la figura 5.14 muestra la arquitectura del modelo Mask R-CNN, donde la primera etapa del modelo es igual a la del detector Faster R-CNN, mientras que la segunda añade de forma paralela a la predicción de clases y *offsets* de los *bounding boxes* la rama *mask*, que genera una máscara binaria para cada RoI.

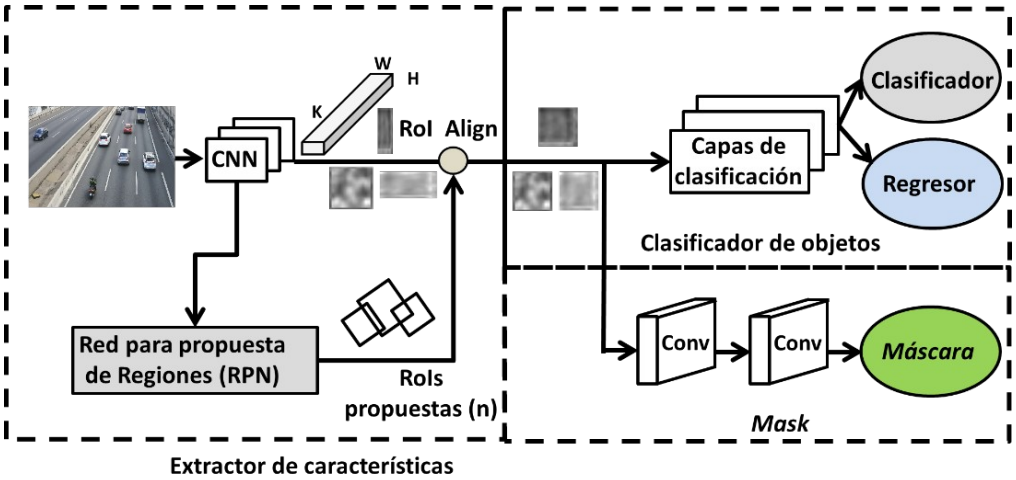

Figura 5.14: Arquitectura de Mask R-CNN. Fuente: Aprendizaje Profundo [3].

5.3.4.1 Función de pérdida

La función de pérdida de este detector es: $L = L_{cls} + L_{reg} + L_{mask}$. Donde las dos primeras funciones L_{cls} y L_{reg} ya fueron descritas a través de las ecuaciones (5.18) y (5.19), del apartado 5.3.2.1, relativo a la función de pérdida de Fast R-CNN.

La rama correspondiente a la máscara (*mask*) tiene una salida de dimensión $K * m^2$ para cada RoI, ya que codifica K máscaras binarias de dimensión (m x m), una para cada una de las K clases. Para ello se aplica una función sigmoide a cada píxel, y se define la función L_{mask} como la entropía cruzada binaria, ecuación 5.23, donde N es el número de píxeles de la RoI, p_i la probabilidad predicha de que el píxel sea del objeto, y p^*_i la probabilidad verdadera de que el píxel sea del objeto.

$$L_{mask} = \frac{1}{N} * \sum_{i}^{N} \left(- p_i^* * \log\left(p_i\right) - \left(1 - p_i^*\right) * \log\left(1 - p_i\right) \right) \qquad (5.23)$$

5.3.4.2 Representación de la máscara

Una máscara codifica la estructura espacial de un objeto de entrada. Esta se extrae realizando su procesamiento, píxel a píxel, a través de las necesarias operaciones, haciendo corresponder esta, con las correspondientes localizaciones de un objeto propuesto. La máscara es calculada por la red FCN [47].

La figura 5.15 muestra un ejemplo del uso de máscaras binarias. En la imagen de la izquierda se tiene una imagen de entrada de la que se extraen las máscaras de los objetos que están presentes en ella. Posteriormente, estas se representan a color en la imagen de la derecha para los vehículos. La clasificación y los *bounding boxes* son obtenidos de las respectivas unidades de clasificación y regresión.

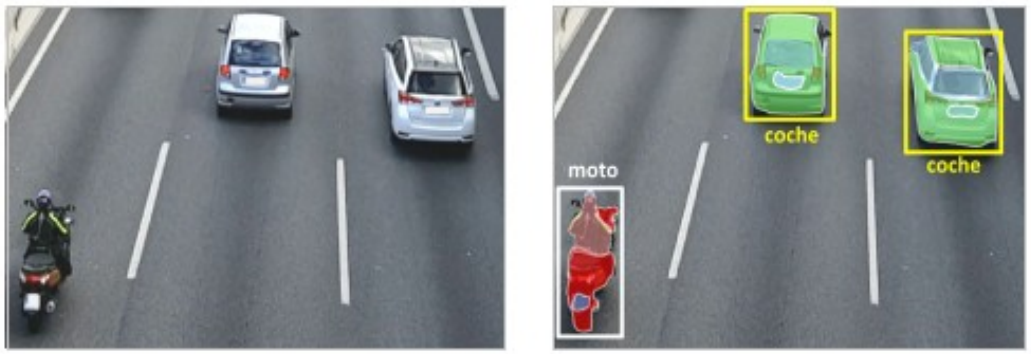

Figura 5.15: Ejemplo de máscaras binarias. Fuente: Aprendizaje Profundo [3].

5.3.4.3 Alineamiento de la RoI

La operación RoI *pooling*, como se explicó en el apartado 5.3.2 (Fast R-CNN), divide una RoI (*Region of Interest*) que es proyectada en un mapa de características, en cuadrículas espaciales (*bins*), con el objetivo de realizar una agrupación de valores, *max pooling* en este caso. Para hallar la correspondencia entre la RoI de la imagen de entrada, con la proyectada en el mapa de características generado en la última capa de convolución de la red, se cuantifican (redondean) los valores reales para que la RoI se ajuste por completo a los límites de las cuadrículas espaciales (*bins*). A su vez, la división de las ventanas de *pooling* en sub-ventanas de (***n*** x ***n***), también requiere cuantizar (redondear) los valores para que los límites de las sub-ventanas se ajusten a los límites de las cuadrículas espaciales.

Estas cuantizaciones (redondeos) de los valores que aplica la capa RoI *pooling* para operar, introduce desalineaciones espaciales entre la RoI y el correspondiente mapa de características. Estas desalineaciones se traducen en pequeños desplazamientos respecto a la ubicación del objeto, los cuales no son significativos en lo que respecta a su clasificación, o a la localización de este mediante el uso de *bounding boxes*. Sin embargo, estos desplazamientos sí son críticos cuando lo que se desea es generar una máscara de bits que extraiga con una precisión a nivel de píxel, la estructura espacial de un objeto.

Para solventar este problema se recurre al alineamiento de la RoI, lo que implica no cuantificar los valores, de esta manera, la ventana de *pooling* se coloca exactamente en el lugar correspondiente en el espacio de reales que constituye el mapa de características, donde se encontraría la RoI proyectada. Para averiguar los valores correspondientes de las características de las sub-ventanas de *pooling* bajo esta estrategia, lo que se hace es recurrir a la interpolación bilineal.

Este método permite calcular el valor de una característica en cierta posición (***x***, ***y***) de un mapa de características en función de los valores de los píxeles que tiene a su alrededor, como en la figura 5.16. La interpolación proporcionará un valor medio ponderado con los cuatro valores que lo rodean. Esta se define formalmente como en la ecuación (5.24), donde ***p***(***x***, ***y***) es resultado de la interpolación, ***p***(***i***, ***j***) es el valor del mapa de características en la posición espacial (***i***, ***j***), y ***h***(***x***, ***y***) es lo que se llama núcleo de interpolación.

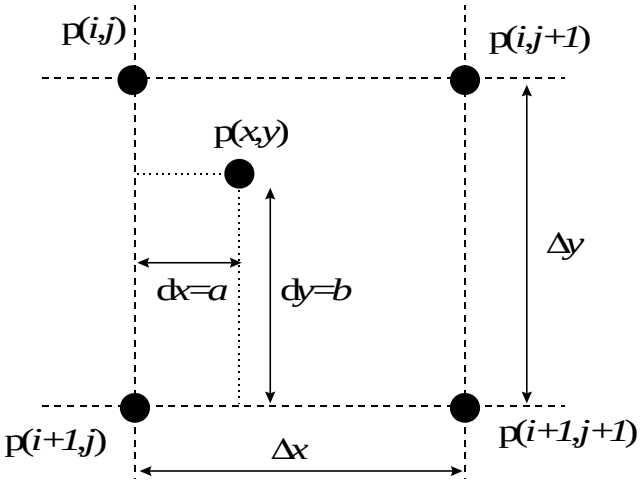

Figura 5.16: Esquema gráfico de la interpolación. Fuente: Aprendizaje Profundo [3].

$$p(x,y) = \sum_{i=-n}^{n} \sum_{i=-m}^{m} p(i,j) * h(x-i, y-j) \tag{5.24}$$

La ecuación (5.24) puede expresarse más sencillamente, como en la ecuación (5.25), donde los coeficientes $\boldsymbol{a_i}$ de los valores que rodean al punto (***x***, ***y***) se definen mediante las ecuaciones (5.26) y (5.27), donde $0 \leq \boldsymbol{dx} \leq 1$, $0 \leq \boldsymbol{dy} \leq 1$, $\boldsymbol{\Delta x}=1$, y $\boldsymbol{\Delta y}=1$.

$$p(x,y) = a_1 p(i,j) + a_2 p(i+1,j) + a_3 p(i,j+1) + a_4 p(i+1,j+1) \tag{5.25}$$

$$a_1 = \frac{\left(1 - \frac{dx}{\Delta x}\right) * dy}{\Delta y} \; ; \; a_2 = \frac{\frac{dx}{\Delta x} * dy}{\Delta y} \tag{5.26}$$

$$a_3 = \left(1 - \frac{dx}{\Delta x}\right) * \left(1 - \frac{dy}{\Delta y}\right) \; ; \; a_4 = \frac{dx}{\Delta x} * \left(1 - \frac{dy}{\Delta y}\right) \tag{5.27}$$

Para obtener los coeficientes definidos en (5.25) y (5.26), el núcleo de interpolación se definió como en la ecuación (5.28).

$$h(x,y) = h(x) * h(y) \; ; \; h(y) = h(x) = \begin{cases} 1 - |x| & si \;\; 0 < |x| < 1 \\ 0 & de\, otro\, modo \end{cases} \tag{5.28}$$

A continuación, se describe un ejemplo práctico de RoI *align* partiendo de la imagen de la figura 5.17. Supongamos que se extrae un mapa de características de tamaño (5 x 5) obtenido de la última capa de convolución de la red troncal de un detector, y que tenemos una RoI proyectada sobre dicho mapa de características, destacada en negro, la cual es de (2 x 2) *bins*. En este punto, el proceso de alineación permite obtener el valor de cada *bin* de la RoI mediante interpolación lineal, a partir de los puntos más cercanos a cada uno.

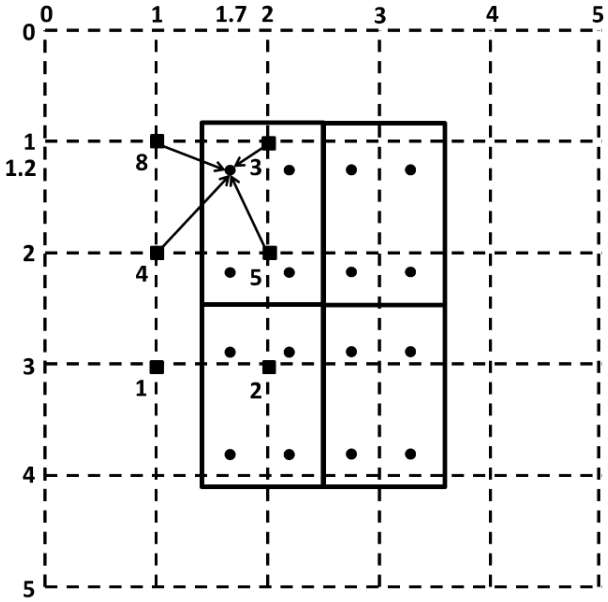

Figura 5.17: Ejemplo práctico de RoI *aling*. Fuente: Aprendizaje Profundo [3].

Para el cálculo de la esquina superior izquierda se emplean los siguientes valores: (***x***, ***y***) = (1.7, 1.2), (1, 1) = 8, (2, 1) = 3, (1, 2) = 4, (2, 2) = 5, ***dx*** = 0.7, y ***dy*** = 0.8. Que al sustituirse en la ecuación 4.25, se obtiene: ***p***(***x***, ***y***) = 8 * (1–0,7) * 0,8 + 3 * 0,7 * 0.8 + 4 * (1–0,7) * (1–0,8) + 5 * 0,7 * (1–0,8) = 4,54. Puede comprobarse que al sustituirse estos valores en la ecuación 5.24, con el núcleo de interpolación tal y como se definió en la ecuación (5.28), se obtiene el mismo resultado.

Una vez obtenidos todos los valores de la RoI alineada, se tiene un nuevo mapa de características de (2 x 2), el cual se utilizará en adelante para poder obtener tanto la clasificación y el *bounding box* del objeto de entrada, como su máscara binaria, o lo que es lo mismo, su estructura espacial.

Cabe a destacar que esta estrategia de RoI *Align* frente a RoI *pooling* ha conseguido mejores resultados en lo que respecta al rendimiento de los detectores.

5.3.4.4 Arquitectura de la red

En el trabajo de He et al. [46], se propone un detector de arquitectura de red múltiple, distinguiendo entre lo que se denomina como, la red de columna vertebral, que es la CNN que extrae al principio del detector las características de una imagen de entrada, y la red de cabecera, que es la que se ocupa de realizar la clasificación del objeto, la regresión de los *bounding boxes*, y de calcular la máscara binaria del objeto.

Para la red columna vertebral se utilizan los modelos ResNet [48], y RexNeXt [50], que tienen 50 y 70 capas. En el caso de la primera red, en la implementación original de Faster R-CNN [14] la extracción de las características tenía lugar a partir de la última capa de convolución del cuarto bloque C4. Esta arquitectura de ResNet con 50 capas, que ejerce a modo de columna vertebral, se denota ResNet-50-C4.

Para la red de cabecera se realiza una extensión de la red de cabecera del detector Faster R-CNN. Su esquema se puede ver la figura 5.18.

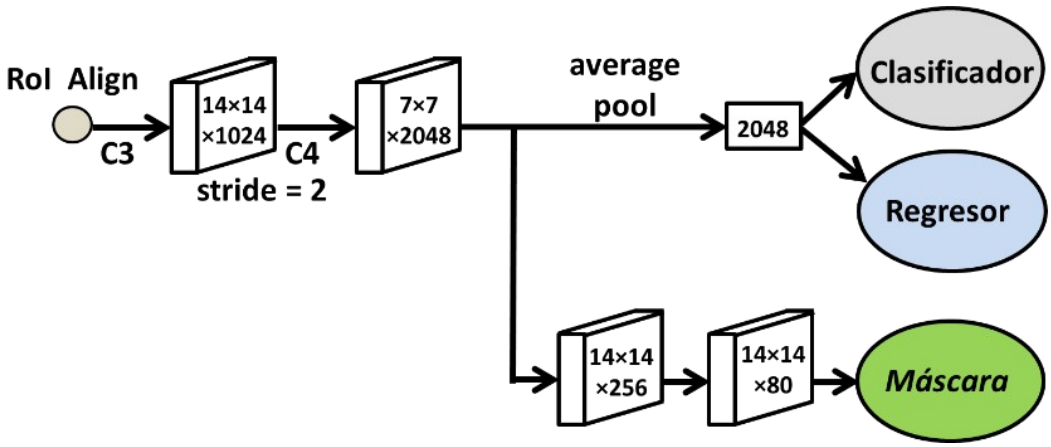

Figura 5.18: Arquitectura de la red de cabecera. Fuente: Aprendizaje Profundo [3].

Como puede observarse en la anterior imagen, una vez realizado el alineamiento de las regiones (RoI *align*), se toma el mapa de características generado en el bloque C3 con dimensiones (14 x 14 x 1024), que sirve como entrada al bloque C4, de tal forma que, aplicando un desplazamiento (*stride*) de dimensión 2, se obtiene el mapa de características (7 x 7 x 2048). También se ha añadido una rama adicional con dos capas convolucionales de tamaño (14 x 14 x 256) y (14 x 14 x 80), esta es la rama que realiza el cálculo de la máscara binaria, que se suma a la de clasificación y regresión.

5.3.5 SSD: Single Shot Multibox Detector

Es un modelo de detección de objetos de carácter multi-escala, propuesto por Liu et al. [12], cuyas principales componentes son diversos bloques conectados en serie. El primer bloque es una red base de tipo convolucional, en el trabajo original se propone una VGG-16 truncada antes de las capas de clasificación, mientras que otras propuestas sugieren modelos como ResNet o SqueezeNet [51].

Independientemente del modelo elegido, la idea es añadir capas al final de la red base truncada, los cuales constituirán los bloques siguientes. Las dimensiones de los mapas de características de estas capas irán disminuyendo de forma progresiva, lo que permite realizar predicciones de detección de objetos a múltiples escalas. Por este motivo se proyectan salidas a partir de mapas de características de distintas resoluciones, de forma que los objetos grandes todavía conservan propiedades en mapas de características de menor resolución, mientras que los objetos pequeños, a partir de un cierto mapa de características con una resolución dada, ya no es posible su descripción por el nivel de resolución del mapa. De aquí la idea de proyección de los mapas a distintos niveles de resolución.

Por tanto, la finalidad de la CNN base es tratar de producir los mapas de características de salida con las mayores dimensiones posibles de alto y ancho. Esto permite que se puedan generar un mayor número de *anchor boxes*, lo que hace que la probabilidad de detectar objetos pequeños sea mayor, en comparación a la probabilidad de las otras capas. A partir de este punto, cada bloque de características multi-escala reduce el alto y ancho del mapa de características con respecto al bloque previo, es decir, con respecto al mapa de características anterior.

Que los mapas de características generados sean más reducidos según se avanza hacia adelante en la red tiene dos consecuencias. La primera es que la cantidad de anchor boxes que puede generar la red según se avanza hacia adelante decrece. La segunda es que, cuanto más se progresa avanza hacia adelante en las capas de la red, mayor es el campo receptivo (*receptive field*) de cada elemento en el mapa de características, y por tanto, mayor es la posibilidad de detectar objetos de mayores dimensiones, tal y como se ha indicado previamente.

A medida que se generan *anchor boxes* de diferentes tamaños en el bloque de red base, y en cada bloque de características multi-escala, se predicen las categorías de los objetos, y los *offsets* de los *anchor boxes*, es decir, los *bounding boxes* que son predichos por el detector.

La figura 5.19 muestra una arquitectura de red de tipo SSD [12] cuya base es una red VGG-16 truncada hasta la capa Conv7.

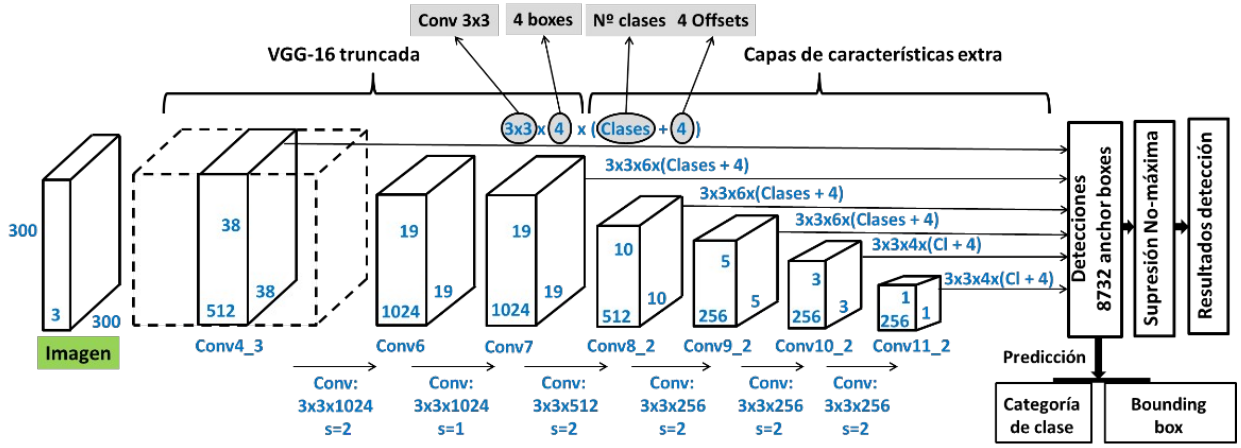

Figura 5.19: Arquitectura de un detector SSD. Fuente: Aprendizaje Profundo [3].

A continuación de la capa Conv7, aparecen las capas de características extra, que son exactamente cuatro: Conv8_2, Conv9_2, Conv10_2, y Conv11_2.

Como se ha mencionado anteriormente, según la imagen va siendo procesada hacia adelante, se va reduciendo su dimensión, lo que conduce a la posibilidad de que se detecten objetos de mayores tamaños en las capas finales, mientras que los de dimensiones reducidas pierden su identificación a niveles de resolución reducida. Puede observarse en la imagen de la figura 5.19, cómo el tamaño del mapa de características pasa de ser de (19 x 19) en Conv7, a (3 x 3) en Conv10_2, lo que significa que la información se ha concentrado en un espacio de menor dimensión, por lo que en el hipotético caso de que un objeto ocupase toda la imagen original, este ya estaría concentrado bajo estas dimensiones. Para ilustrar este hecho, puede observarse en la figura 5.20 cómo se requiere un menor número de localizaciones espaciales (cuadros) blancas que negras, para delimitar el mismo objeto.

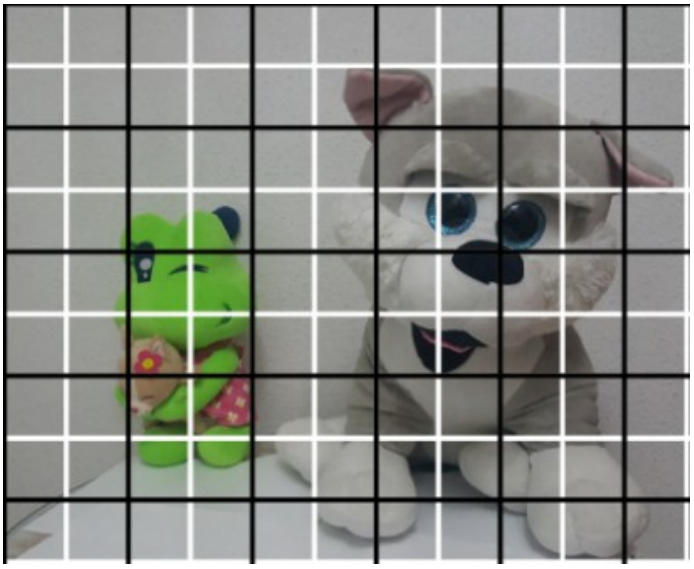

Figura 5.20: Ejemplo de dimensionalidad. Fuente: Aprendizaje Profundo [3].

Procesando la imagen hacia adelante a través de las distintas capas de convolución, con sus diferentes dimensiones, número de núcleos convolucionales (***k***; *kernels*), y desplazamientos (***s***; *strides*), se obtienen los mapas de características (***m*** x ***n*** x ***k***), tal como puede verse en la imagen de la figura 5.19. Un ejemplo de esto sería pasar del mapa de características de Conv7 al de Conv8_2, para lo cual se aplica un núcleo de dimensión (3 x 3), con ***k*** = 12, y ***s*** = 2.

Para cada una de las (***n*** x ***m***) localizaciones espaciales de la imagen, y de los mapas de características que son generados por el detector, se definen ***B*** *anchor boxes* de diferentes tamaños y relaciones de aspecto. Una vez generados, para cada ***B*** se predice la pertenencia a cada una de las ***c*** categorías existentes, junto con los 4 desplazamientos (*offsets*) relativos a los *ground truth bounding boxes*. En total se obtienen un total de (***n*** x ***m***) x ***B*** x (***c*** *Classes* + 4 *Offsets*) resultados por capa.

Para realizar la predicción de clases y la regresión de los *bounding boxes*, en lugar de emplearse dos capas de tipo *fully connected* paralelas como en el caso de otros detectores, se recurre a una capa de convolución que toma su entrada del mapa de características (***m*** x ***n***) correspondiente. Esta capa aplica en cada localización del mapa de características (***m*** x ***n***), un total de (***B*** x (***c*** + 4)) *kernels* de (3 x 3 x ***P***), para generar como salida un tensor de tamaño (***m*** x ***n*** (***B*** x (***c*** + 4))) que contiene para cada localización del mapa de características, el grado de pertenencia a cada clase, y las coordenadas de un *bounding box*, para cada uno de los *anchor boxes*.

5.3.5.1 Entrenamiento

Para el entrenamiento se requiere establecer una relación de asociación entre los anchor boxes y los ground truth bounding boxes. Para cada ground truth bounding box se seleccionan distintos rectángulos que varían en localización, razón de aspecto, y escala. Esta asociación se establece en función de los valores del índice IoU, siguiendo el criterio de que este debe resultar mayor a un cierto valor umbral, por ejemplo 0.5, en cuyo caso el anchor box es etiquetado como positivo. De esta forma la red puede predecir altos valores para múltiples boxes que están solapados, evitando la selección del que tiene el máximo valor de solapamiento. Tras esta asociación se aplica tanto la función de pérdida (loss), como la retropropagación (backpropagation) [21].

Cabe destacar que, antes de llevar a cabo el entrenamiento, se ha de seleccionar un conjunto de *anchor boxes*, con sus respectivos tamaños y relaciones de aspecto, para utilizarlos en la detección de objetos durante el proceso de entrenamiento.

5.3.5.2 Función de pérdida

La función de pérdida viene definida en la ecuación (5.29), es una suma promediada de las funciones de pérdida correspondientes a la clasificación (conf) y a la regresión (loc), donde $\boldsymbol{\alpha}$ es un parámetro regularizador para controlar el equilibrio entre las dos funciones de pérdida.

$$L(x,c,l,g) = \frac{1}{N} * \left(L_{conf}(x,c) + \alpha * L_{loc}(x,l,g)\right) \tag{5.29}$$

La función de pérdida relativa a la clasificación se define como en la ecuación (5.30), donde el índice $\boldsymbol{i}$ sirve para poder recorrer tanto los *anchor boxes* etiquetados como positivos, como los etiquetados como negativos. La variable $\hat{\boldsymbol{c}}_{\boldsymbol{i}}^{\boldsymbol{p}}$ indica el grado de pertenencia del contenido del *anchor box* $\boldsymbol{i}$ a la clase $\boldsymbol{p}$. Este grado de confianza se calcula mediante la función exponencial normalizada (*softmax*). La variable $\hat{\boldsymbol{c}}_{\boldsymbol{i}}^{\boldsymbol{0}}$ indica el grado del contenido del *anchor box* $\boldsymbol{i}$ al fondo.

$$L_{conf} = -\sum_{i \in Positivo}^{N} x_{ij}^{p} * \log\left(\hat{c}_i^p\right) - \sum_{i \in Negativo}^{N} \log\left(\hat{c}_i^0\right) ;\ \hat{c}_i^p = \frac{\exp\left(c_i^p\right)}{\sum_p \exp\left(c_i^p\right)} \tag{5.30}$$

La función de pérdida relacionada con la regresión de los *bounding boxes* se define en la ecuación (5.31), donde l^m_i es la componente ***m*** del *bounding box* ***i*** que se ha predicho, $\hat{g}^m_i$ es el valor de la componente ***m*** del *ground truth bounding box* ***i***.

$$L_{loc}(x, l, g) = \sum_{i \in Positivo} \sum_{m \in \{cx, cy, w, h\}} x^k_{ij} * S_{L1}\left(l^m_i - \hat{g}^m_j\right) \quad (5.31)$$

En ambas ecuaciones anteriores (5.30) y (5.31), las variables x_{ij}^k y x_{ij}^p, representan lo mismo, son un indicador de correspondencia entre el *anchor box* ***i*** y el *ground truth bounding box* ***j***, de categoría ***k***, o ***p***. Esta correspondencia toma 1 por valor si el valor del índice IoU es superior al valor umbral fijado, y 0 en caso contrario.

5.3.6 YOLOv1

Dentro de las diferentes versiones de YOLO existentes en la actualidad, la v1, es la que posee un esquema de funcionamiento más simple. Consiste en una red de tipo CNN que predice múltiples *bounding boxes* junto con sus probabilidades de pertenencia a una determinada clase. El entrenamiento se realiza sobre imágenes completas y no sobre partes de una imagen, lo que hace que el modelo sea un detector de un estado. Posee una alta velocidad de cara al procesamiento, y obtiene una buena precisión en las clasificaciones al generalizar las representaciones de los objetos.

YOLO funciona dividiendo una imagen en una rejilla de (***S*** x ***S***) celdas (*anchors*). Si el centro de un objeto cae dentro de una celda, entonces esa misma celda se responsabiliza de la detección del objeto. Cada celda predice ***B*** *bounding boxes* y valores de confianza para dichos *bounding boxes*. El valor de la confianza sirve para expresar la seguridad que tiene el modelo de que un *bounding box* contenga un objeto, y lo precisa que es la caja que predice. Esta confianza se define según la ecuación (5.32), que toma el valor 0 si no hay un objeto en la celda. Los elementos de esta fórmula son, ***P***(*object*), que indica la probabilidad de que un *bounding box* contenga un objeto, y el IoU entre el bounding box predicho y el *ground truth bounding box*.

$$boxconfidence = P(object) * IoU^{groundtruth}_{predicted} \quad (5.32)$$

Cada *bounding box* predicho ***B***, consta de cinco predicciones (***x***, ***y***, ***w***, ***h***, ***c***), donde ***x*** e ***y*** son el centro del *bounding box*, el cual se encuentra dentro de los límites de cierta rejilla en que se ha subdividido la imagen. Los valores de ancho y alto, ***w*** y ***h***, que se dan en relación al tamaño de la imagen completa, y el grado de confianza ***c***, definido como en la ecuación 5.32.

Cada celda de la rejilla en que se dividió la imagen, aparte de predecir ***B*** *bounding boxes*, como se explicó anteriormente, también predice ***C*** probabilidades de clase condicionadas $P(C_i|object)$, que indican la probabilidad de que un objeto presente pertenezca a una cierta clase ***i***. Estas probabilidades se calculan con independencia del número de *bounding boxes* predichos ***B***.

Calculadas todas las predicciones de cada ***B***, y todas las ***C*** probabilidades de pertenencia de un objeto a una clase, se calcula la confianza específica de la clase.

Esta confianza específica de la clase se calcula según la ecuación (5.33), que se simplifica como puede según la ecuación (5.34), donde en esta última, $P(C_i)$ es la probabilidad de que el objeto pertenezca a la clase C_i. El valor de la confianza de clase indica la probabilidad de que cierta clase aparezca en el *bounding box* que es predicho, y cómo de bien este *bounding box* predicho se ajusta al objeto.

$$classconfidence = P(C_i \vee object) * P(object) * IoU_{predicted}^{groundtruth} \quad (5.33)$$

$$classconfidence = P(C_i) * IoU_{predicted}^{groundtruth} \quad (5.34)$$

En resumen, YOLO divide una imagen en una rejilla de de (***S*** x ***S***) celdas (*anchors*), y para cada una se realizan *5**B*** predicciones de *bounding boxes*, y ***C*** probabilidades condicionadas de pertenencia a una clase determinada para un objeto dado. Todas estas predicciones conforman un tensor de salida de tamaño ***S*** x ***S*** x (*5**B***+***C***), cuyos datos permiten calcular los valores de la confianza específica de la clase, ecuación (5.34). Con estos valores calculados, se aplica la estrategia de supresión no máxima para descartar las predicciones repetidas peores, prevaleciendo así solo las mejores.

Se ha de tener siempre presente que, aunque el detector predice varios *bounding boxes* por cada celda de la rejilla, solo se predice un único objeto por cada celda.

5.3.6.1 Arquitectura del modelo

El diseño del detector YOLO en su primera versión, según Redmon et al. [52], es el mostrado en la figura 5.21. Este está inspirado en la arquitectura de GoogleLeNet [53].

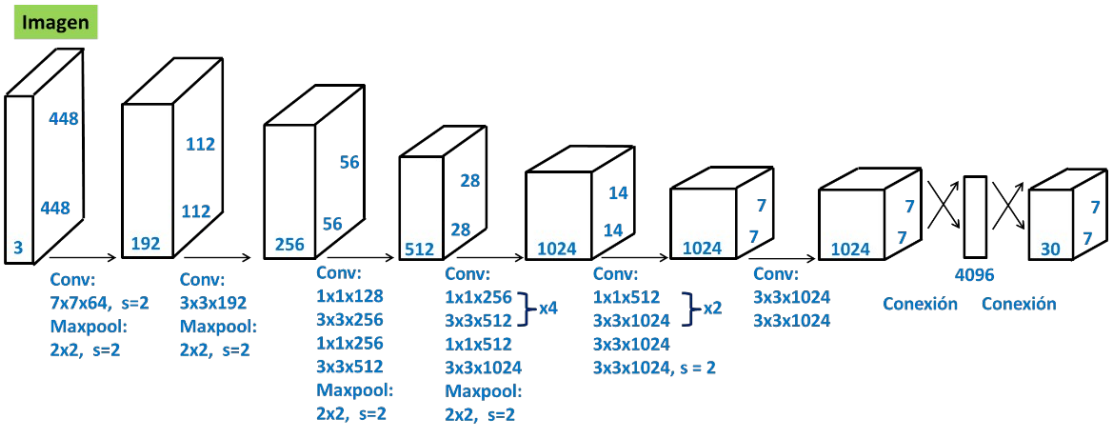

Figura 5.21: Arquitectura de un detector YOLOv1. Fuente: Aprendizaje Profundo [3].

El citado modelo de la figura 5.21, está formado por 24 capas de tipo convolucional seguidas por 2 de tipo totalmente conectadas. La diferencia con GoogleLeNet reside en que en lugar de utilizarse sus módulos *inception*, se utilizan capas de convolución de (1 x 1) seguidas por capas de convolución de (3 x 3). De esta forma, alternando capas de convolución (1 x 1) se reducen las dimensiones del espacio de características a partir de las capas precedentes. El tensor de salida de la red posee un tamaño de (7 x 7 x 30).

5.3.6.2 Función de pérdida

a) Pérdida de clasificación:

La función de pérdida relativa a la clasificación se define en la ecuación (5.35), la cual se calcula como el error cuadrático de las probabilidades condicionadas por cada clase. En esta ecuación, $\hat{p}_i(c)$ es la probabilidad de que un objeto pertenezca a una clase c condicionada a la celda i, mientras que la variable O_i^{obj}, se evalúa como 1 si un objeto aparece en la celda i, y como 0 en caso contrario. Finalmente, S^2 es la cantidad de celdas en que se divide la imagen original.

$$L_{cls} = \sum_{i=0}^{S^2} O_i^{obj} \sum_{c \in classes} \left(p_i(c) - \hat{p}_i(c) \right)^2 \qquad (5.35)$$

b) Pérdida de confianza:

Si un objeto se detecta en un *bounding box*, la pérdida de la confianza se determina con la expresión a la izquierda de la suma en la ecuación (5.36), en caso contrario, se determina con la expresión a la derecha de dicha suma. La variable $\hat{C}_i$ representa el valor de la confianza del *bounding box* ***j*** en la celda ***i***, mientras que O_{ij}^{obj} vale 1 si el *bounding box* ***j*** de la celda ***i***, es el responsable de detectar del objeto, y 0 en caso contrario. La variable O_{ij}^{noobj} se comporta de forma complementaria a O_{ij}^{obj}, de tal forma que toma el valor 1 O_{ij}^{obj} si es 0, y viceversa.

$$L_{conf} = \sum_{i=0}^{S^2} \sum_{j=0}^{B} O_{ij}^{obj} * \left(C_i - \hat{C}_i\right)^2 + \lambda_{noobj} \sum_{i=0}^{S^2} \sum_{j=0}^{B} O_{ij}^{noobj} * \left(C_i - \hat{C}_i\right)^2 \qquad (5.36)$$

c) Pérdida de localización:

La función de pérdida que está relacionada con la localización se describe mediante la ecuación (5.37), esta mide los errores de la localización y el dimensionamiento de los *bounding boxes* predichos para la detección de objetos. Para ello solo se tiene en cuenta el rectángulo responsable de la detección del objeto.

$$L_{loc} = \lambda_{coord} \sum_{i=0}^{S^2} \sum_{j=0}^{B} O_{ij}^{obj} \left(\left(x_i - \hat{x}_i\right)^2 + \left(y_i - \hat{y}_i\right)^2 + \left(\sqrt{w_i} - \sqrt{\hat{w}_i}\right)^2 + \left(\sqrt{h_i} - \sqrt{\hat{h}_i}\right)^2 \right) \qquad (5.37)$$

d) Función de pérdida final:

La función de pérdida final se calcula como en la ecuación (5.38), es decir, sumando las funciones de pérdida de la clasificación, confianza y localización.

$$L = L_{cls} + L_{conf} + L_{loc} \qquad (5.38)$$

De esta función de pérdida, y sus sub-funciones, han de aclararse ciertos aspectos:

- La suma del error cuadrático se utiliza porque es fácil de optimizar, pero no consigue realmente el objetivo de maximizar la precisión promedio.
- Es necesario introducir los factores de regulación λ_{coord} y λ_{noobj}, para corregir que los errores de localización tengan el mismo peso que los de clasificación.

- En cada imagen, muchas celdas de la rejilla no contienen objetos, esto hace que su confianza tienda a cero, imponiéndose sobre las celdas que contienen objetos. Esto puede producir un sesgo, llevar al modelo hacia la inestabilidad, y originar que el entrenamiento diverja de forma temprana. Para evitar esto, se incrementa la pérdida relativa a la predicción de las coordenadas de los *bounding boxes*, y a la vez se decrementa la pérdida de las predicciones de confianza de los *bounding boxes* que no contienen objetos. Los factores para la regulación se pueden fijar como sigue para evitar los problemas que han sido descritos: $\boldsymbol{\lambda_{coord}} = 5$, $\boldsymbol{\lambda_{noobj}} = 0.5$.
- Para evitar que los errores de la localización de los *bounding boxes* computen por igual para los grandes que para los pequeños, se utilizan las raíces sobre los altos y anchos para evitar este problema.

Capítulo 6 - Marco técnico

6.1 Introducción

En el presente capítulo se describen las tecnologías, diseños, y desarrollos, relativos a la construcción del proyecto que se propone en este trabajo de cara al reconocimiento de objetos en imágenes mediante detectores de objetos basados en Aprendizaje Profundo (*Deep Learning*) [3]. Con tal propósito, se crea un conjunto de programas que constituyen una plataforma para la automatización del ciclo de actividades relacionadas con la puesta a punto y evaluación de modelos de detectores de objetos, con la finalidad de poder implantarlos de forma posterior en otros proyectos científico-ingenieriles. A continuación se identifican los lenguajes de programación y los *frameworks* que dan soporte a los desarrollos, y las plataformas para su alojamiento y ejecución.

6.2 Tecnologías y recursos

El conjunto de tecnologías y recursos empleados para el diseño y el desarrollo de los programas que forman este proyecto se describen a continuación.

6.2.1 Python3

Python[57] ha sido uno de los lenguajes de programación más populares de los últimos años, algo que puede corroborarse a través del índice TIOBE [54]. Su aceptación entre los desarrolladores se ha visto incrementada con el tiempo debido a la multitud de ventajas que este ofrece:

- Código abierto y gratuito: Python se distribuye bajo una licencia de tipo PSFL (*Python Software Foundation License*) [55] que es similar a la BSD (*Berkeley Software Distribution*) [56], siendo pues, una licencia libre permisiva, ya que cualquier persona puede descargar su código, modificarlo y distribuirlo como desee.
- Baja curva de aprendizaje: este lenguaje es sencillo de aprender y utilizarse, algo que gusta tano a programadores primerizos como expertos. Algunos de los factores que contribuyen a este aspecto es su sencillez sintáctica, y su legibilidad a la hora de leer o escribir programas hechos con él.

- Multitud de bibliotecas y *frameworks*: la gran cantidad de desarrolladores que hacen uso de Python [57] han propiciado la aparición de múltiples librerías y *frameworks* que han ampliado la capacidad original del lenguaje, haciendo que este se volviera cada vez más muti-propósito y multi-paradigma.
- Multi-propósito: aunque en un principio fue concebido para poder desarrollar aplicaciones de sistemas, con el tiempo se traspasó esta frontera. Ahora es utilizado, tanto para aplicaciones de sistema, como de escritorio, móviles, páginas y servicios web, videojuegos, acceso y gestión de datos, analítica de datos, cómputo distribuido y paralelo, aprendizaje automático y aprendizaje profundo, microcontroladores y microcomputadores, etc. Es en el aprendizaje profundo donde se focaliza este trabajo.
- Multi-paradigma: aparte de soportar los paradigmas de programación más fundamentales, como son la programación estructurada y la orientada a objetos, también soporta otros paradigmas diferentes, como la programación funcional, la programación lógica (reglas), la programación basada en el uso de restricciones, etc.
- Multi-plataforma: Python es similar a Java en cuanto a la forma en la que este consigue la portabilidad. Su código fuente de alto nivel se traduce a lo que se denomina *bytecode*, que son instrucciones de bajo nivel, ajenas a las que produciría el procesador de un computador, que pueden ser analizadas y ejecutadas rápidamente por un intérprete, normalmente CPython [58].
- Entornos de Desarrollo Integrados (IDE's): pueden crearse aplicaciones con Python desde el intérprete de comandos del sistema, e incluso desde editores de texto. Sin embargo, cuando el tamaño de los programas pasa de pequeño o mediano, a grande o muy grande, se vuelve necesario el uso de programas que proporcionan facilidades para el desarrollo, por ejemplo, proporcionando analizadores sintácticos y de errores, estrategias que permitan construcción y organización del código, sistemas de control de versiones, comparadores y depuradores de código, mecanismos para la realización de pruebas, etc. En este sentido, la comunidad de Python está bien provista de entornos para el desarrollo, contando por ejemplo con Visual Studio Code, Atom, Vim, Emacs, Spyder, PyCharm, etc.

- Ámplia comunidad: una de las grandes ventajas de este leguaje es que al haber tantos desarrolladores tras su uso o ampliación, es que estos proveen un gran soporte ante la aparición de dudas, o errores de carácter técnico o teórico, proporcionando ante ellos asistencia mediante ejemplos, soluciones, consejos basados en la experiencia o en buenas prácticas, etc.

6.2.2 TensorFlow

TensorFlow [59] es una biblioteca dedicada a la construcción de modelos de aprendizaje automático, concretamente, de redes neuronales. Esta fue liberada en 2015 por *Google* como un proyecto de código abierto que fue el resultado de la optimización de un proyecto anterior, *DistBelief*, de su equipo de investigación en inteligencia artificial, *Google Brain* [60]. Las características principales de esta API son las siguientes:

- Su licencia es la Apache 2.0 [61], una licencia de software libre permisiva creada por la propia fundación Apache [62].
- Está implementada por completo para soportar los lenguajes Python [57] y C++, no siendo así para otros lenguajes.
- Puede ejecutarse en múltiples sistemas operativos, así como CPU's, y GPU's[64] que tengan soporte para CUDA (*Compute Unified Device Architecture*) [63].
- Genera modelos de aprendizaje profundo basados en el cálculo con tensores, que como se indicó previamente son matrices multidimensionales, de ahí su nombre.
- Soporta el entrenamiento de modelos de aprendizaje profundo con TPU's (*Tensor Processing Units*) [65], un hardware específico creado para optimizar el cálculo matricial.
- Actualmente se encuentra en su versión 2.0, aunque aún se mantiene la compatibilidad necesaria para poder ejecutar los desarrollos de la anterior versión 1.0, pero se recomienda la transacción de esta última a la primera.
- Para facilitar su manejo se está promoviendo el uso de una API de alto nivel con un carácter más amigable, esta recibe el nombre de Keras [66], la cual cobró soporte por parte del proyecto de *TensorFlow* a partir de 2017.

6.2.3 TensorFlow Object Detection API

La *TensorFlow Object Detection API* [67] es un *framework* de código abierto que ha sido construido sobre la propia API de *TensorFlow* [59] por investigadores de *Google*. Esta tiene objetivo de facilitar la construcción, el entrenamiento, y el despliegue de los detectores de objetos. De entre sus aspectos más destacables pueden citarse los siguientes:

- Se encuentra bajo la licencia Apache 2.0 [61], que es una licencia de software libre permisiva creada por la propia fundación Apache [62].
- Está liberada tanto para la versión 1 de *TensorFlow*, como para su versión 2, y aún se mantiene la compatibilidad para poder ejecutar los desarrollos de la versión 1, aunque se recomienda una migración de la versión 1 a la 2.
- Los modelos para detección de objetos soportados son:
  - Detectores de un estado:
    - Basados en *anchors*: EfficientDet (D0 – D7), SSD (MobileNet; ResNet).
    - No basados en anchors: CenterNet (MobileNet; ResNet; HourGlass).
  - Detectores de dos estados:
    - Basados en *anchors*: Faster R-CNN (ResNet; Inception-ResNet), Mask R-CNN (Inception-ResNet).
    - No basados en *anchors*: no hay soporte.
- Respecto a sus detectores soportados:
  - Fueron entrenados con COCO (*Common Objects in Context*) [68].
  - Pueden ser entrenados mediante una GPU [64] que cuante con soporte para CUDA (*Compute Unified Device Architecture*) [63], sin embargo, no todos poseen soporte para ser entrenados con una TPU [65].
  - Su configuración no se ajusta mediante el uso de comandos, si no a través de sus correspondientes ficheros (*.config), que contienen secuencias de *protobuffers*, un formato para el intercambio de datos de código abierto que fué inventado por *Google*. Dentro de un fichero de este tipo, se define el *pipeline* del entrenamiento, que se divide en 5 partes:

        - model: define el tipo de detector que se va a entrenar (SSD, Faster R-CNN, etc), y dentro de este, el número de categorías en que clasificar, el tamaño al que redimensionar las imágenes de entrada, la red que hace de extractor de características, la función de pérdida a emplear y su regulación, la realizaciónn de *Non-Maximun Supression*, etc.
        - train_config: define los parámetros relacionados con el entrenamiento del modelo, es decir, el número de *steps*, el tamaño del *mini-batch*, las opciones de *dataugmentation*, el optimizador de la función de error (SGD, SGDM [21], ADAM, RMSProp, etc), etc.
        - eval_config: define las métricas que se utilizarán para evaluar el detector. Por defecto, las métricas con las que se evalúa son con las de COCO (*Common Objects in Context*) [68], aunque también pueden elegirse las de PASCAL VOC PASCAL VOC (*Visual Object Classes*) [43, 44], o las de Open Images [69].
        - train_input_config: define el conjunto de datos para el entrenamiento.
        - eval_input_config: define el conjunto de datos para la validación.
- Proporciona *scripts* para facilitar ciertas tareas, por ejemplo:
    - La generación de registros de tipo *TF-Records*, que codifican las imágenes del *dataset*, y sus metadatos, para realizar un entrenamiento con ellos en un formato que puede manejarse de forma eficiente desde el punto de vista del almacenamiento.
    - La exportación de un detector entrenado en un archivo (*.pb), este tiene tanto su arquitectura, como sus variables (*weigths*, *biases*, etc). Lo que es de utilidad para salvaguardar un modelo y cargarlo cuando sea necesario.
    - Realizar el proceso de entrenamiento de un detector. Este genera como resultado uno o varios ficheros que contienen el estado (variables) de un detector, y uno o varios ficheros que contienen metadatos asociados al estado (variables). El estado se almacena en un fichero con la extensión (*.data), mientras que los metadatos asociados al estado se almacenan en un fichero con la extensión (*.index).

- Realizar el proceso de validación de un detector de forma paralela a su proceso de entrenamiento. Este proceso permanece a la espera hasta que el proceso de entrenamiento genera un archivo (*.data), que se procesa para cargar el estado en el modelo correspondiente, y entonces evaluarlo.

6.2.4 Google Drive

Es un servicio de almacenamiento de archivos creado por la empresa *Google* en el año 2012. Su aparición supuso la unificación del espacio de alojamiento en la nube de que disponen los usuarios, pues para cada uno, este es compartido tanto por el servicio de almacenamiento de archivos, como por el servicio de correo electrónico, y otros servicios.

Algunos aspectos notables de este servicio en la nube son:

- Cada usuario posee 15 GB gratuitos de espacio para el almacenamiento.
- Permite la sincronización de un ordenador con archivos remotos en la nube.
- La creación de documentos de texto, hojas de cálculo, u otros, con los que los usuarios pueden trabajar en remoto de forma cooperativa y simultánea, en tiempo real. Ciertas aplicaciones de documentos compartidos incluso ven reforzados este aspecto al disponer de un chat mediante el cual se pueden comunicar los usuarios.
- Independientemente de si se trabaja directamente con archivos en la nube, o mediante el mecanismo de sincronización de nuestro local con la nube, los archivos cuentan con un sistema de control de cambios, y también de control de versiones.

6.2.5 Google Cloud Platform

Es una plataforma creada por *Google* en 2012 para poder proporcionar servicios de computación en la nube (*cloud computing*), los cuales ofrecen a sus usuarios la posibilidad de emplear hardware y/o software remotos con los que poder construir sus propias soluciones. Estos recursos se pueden gestionar, configurar, y ejecutar mediante sencillos interfaces web. En función de sus necesidades, un cliente puede contratar el tipo de servicio de *Google Cloud Platform* [71] que más conveniente le sea, teniendo en cuenta las siguientes definiciones de servicios:

- Infraestructura como servicio (IaaS; *Insfraestructure as Service*): este tipo de servicios de computación en la nube son aquellos proporcionan recursos para el procesamiento, el almacenamiento, o la comunicación, que deben ser gestionados por el usuario que los contrata. Algunos ejemplos de este tipo de servicios serían, por ejemplo, la contratación de máquinas virtuales, redes virtuales, o almacenes de archivos.

- Plataforma como servicio (PaaS; *Platform as Service*): estos servicios son los que proporcionan a un cliente, una plataforma de hardware y software para desarrollar, desplegar, ejecutar, y gestionar aplicaciones. En el caso de este tipo de servicios, los desarrolladores pueden valerse de las ventajas de poder utilizar la plataforma para realizar sus desarrollos y gestionar los datos, sin tener que preocuparse por tener que configurarla o mantenerla. Ejemplos de este tipo de servicios basados en plataformas de desarrollo son Google App Engine, Amazon Web Services Elastic Beanstalk, Heroku, Dokku, Flynn, etc.

- Software como servicio (SaaS; *Software as Service*): este tipo de servicios son aquellos en los que su proveedor gestiona y mantiene la infraestructura y las aplicaciones, por lo que los usuarios finales solo se preocupan por el uso de un programa informático concreto, y nada más. Algunos de los ejemplos más fundamentales de este tipo de servicios serían, por ejemplo, nuestro servidor de correo electrónico, cualquier programa de un paquete ofimático utilizado a través de internet, Slack, Jira, Netflix, Spotify, etc.

6.2.6 Google Compute Engine

Es un servicio para la adquisición de infraestructura (*IaaS*) ofertado por la empresa *Google* desde el año 2012. Este mismo servicio compone a su vez parte de la propia infraestructura de los otros servicios ofertados por *Google*. Los usuarios que lo contratan pueden ejecutar máquinas virtuales basadas en Windows o en Linux, ya predefinidas, o personalizarlas en base a sus necesidades. Las predefinidas son combinaciones que surgen de los diversos usos de la CPU y la RAM, de forma que se juega con el número de *cores* y/o con el espacio de la memoria de trabajo.

En el caso de que un usuario quisiera una máquina virtual todavía más específica, este podría seleccionar los recursos vinculados a esta, por ejemplo, el tipo de CPU

virtual (procesador) que desea utilizar, ya que cada uno cuenta con una frecuencia de trabajo y distintos niveles de caché con distintas capacidades, la cantidad de CPU's virtuales, la cantidad de memoria RAM, su tipo de disco de almacenamiento, que puede ser mecánico o de tipo SSD (*Solid State Drive*), la cantidad de espacio de almacenamiento en el disco de almacenamiento, el modelo de GPU [64] a utilizar, ya que cada uno tiene una velocidad y una cantidad de memoria de trabajo propia y, finalmente, el número de GPU's que desea utilizar en total del tipo seleccionado.

El costo que le supone al usuario disponer de sus máquinas virtuales es totalmente dependiente de lo caro que el hardware sea, además de la cantidad de tiempo que el usuario utilice la máquina virtual. Si no hay uso de la máquina virtual, no se cobrará por esta.

6.2.7 Google Cloud Storage

Es un servicio de infraestructura como servicio (*IaaS*) creado por *Google* en 2010, que está dedicado al almacenamiento de cualquier tipo de archivos, y a su posterior acceso. Las diferencias de este servicio respecto al ofrecido por *Google Drive* [70] son:

- No está orientado al almacenamiento y acceso de archivos personales, sino más bien al uso corporativo de la información.
- No hay opción de almacenamiento gratuito, se paga por el uso que se da a los almacenes de datos (*buckets*) que se creen.
- Los almacenes de datos (*buckets*) no tienen predefinido un límite en lo que respecta a su capacidad de almacenamiento.
- Permite establecer mecanismos de seguridad, por ejemplo, roles (permisos) de acceso a los datos, y el cifrado de estos.
- Los datos almacenados son comprimidos para poder optimizar el espacio de almacenamiento y para realizar accesos más eficientes.
- Alta disponibilidad de los datos, y baja latencia a la hora de acceder a ellos.
- Existe una gran cantidad de países que cuentan con centros de datos en los que pueden crearse almacenes de datos (*buckets*).

- Permite señalar la clase de almacenamiento, es decir, el uso que se le dará al almacén de datos (*bucket*):
    - *Standar Storage*: cuando lo que se desea es acceder a datos con una alta frecuencia, o cuando estos van a permanecer poco tiempo almacenados.
    - *Nearline Storage*: cuando los datos van a almacenarse mucho tiempo, y la frecuencia de acceso a ellos va a ser baja, en torno a un mes.
    - *Coldine Storage*: cuando los datos van a almacenarse mucho tiempo, y la frecuencia de acceso a ellos es va a ser todavía más baja, en torno a los tres meses.
    - *Archive Storage*: cuando los datos van a almacenarse mucho tiempo, y la frecuencia de acceso a ellos va a ser muy muy baja, en torno a un año.
- Independientemente del tipo de almacenamiento que se haya seleccionado, es posible escoger la redundancia geográfica de los datos:
    - *Region*: indica que los datos se almacenarán en un único lugar geográfico, es decir, en un centro de datos de una ciudad.
    - *Dual-Region*: la información se encuentra en un par de regiones, es decir, en dos centros de datos, cada uno perteneciente a una ciudad concreta.
    - *Multi-Region*: la información se encuentra en un área geográfica grande, por ejemplo, centros de datos de diferentes ciudades de Europa.
- Uno o varios almacenes de datos (*buckets*), de un usuario, existen bajo un identificador unívoco (*proyect*), que es establecido por el usuario, y dentro de este, cada almacén (*bucket*) es identificado por su propio nombre.

6.2.8 Google Colaboratory

Google Colaboratory [73], o *Google Colab*, es un entorno proporcionado por la empresa *Google* a partir del año 2017 para realizar desarrollos con la tecnología del proyecto *Jupyter* [74], los *Jupyter Notebooks*, llevados a la computación en la nube a modo de *Paas* (*Platform as Service*), por lo que ambos proyectos comparten las características propias de la tecnología del proyecto *Jupyter*.

Un *Jupyter Notebook* es un documento interactivo creado con tecnología web, que contiene una lista ordenada de celdas que pueden contener código ejecutable de Python [57], u otros lenguajes, o secuencias de algún lenguaje de marcado, como por ejemplo, *markdown*, *HTML*, o *LaTeX*, con los que pueden declararse títulos, enunciados, apartados, enlaces, tablas, imágenes, fórmulas, etc. Es decir, con estos lenguajes se establecerá cómo ha de ser la presentación del documento ante los usuarios. Los *Notebooks* se guardan con la extensión (*.ipynb), y albergan un formato JSON [75] que contiene la lista de celdas ejecutables, y de presentación.

Esta tecnología suele ser usada por científicos de datos, analistas matemáticos y/o estadísticos, personas dedicadas a la visualización de la información y la generación de informes de datos, personas que trabajan en aprendizaje automático o profundo, estudiantes y otros profesionales en general.

La principal diferencia de *Google Colab* con respecto al proyecto *Jupyter* es, como se ha dicho anteriormente, la disponibilidad de los *Noteboks* como una plataforma de desarrollo, despliegue, y ejecución de código en la nube (*Paas*). Esta se ejecuta sobre una máquina virtual de *Google Engine* [72] que proporciona la infraestructura para la ejecución del servicio (*IaaS*). La máquina virtual ejecuta un sistema operativo de tipo GNU\Linux ya configurado para que el servicio del *Notebook* pueda ejecutarse sin problemas, de hecho, esta no es perceptible a simple vista por el usuario, ya que este solo ve el *Notebook* que está desarrollando, sin embargo, es posible interactuar con ella a través de una sintaxis especial que permite la ejecución de comandos del intérprete *bash* del sistema GNU\Linux subyacente. Esta estrategia del servicio que ofrece *Google Colab* para la ejecución de *Jupyter Notebooks*, trae consigo las siguientes ventajas:

- El usuario no tiene que instalar el proyecto *Jupyter* [74] ni ninguna dependencia necesaria para su funcionamiento, como librerías del sistema, u otros, ya que todo esto está configurado en la máquina virtual GNU\Linux de *Google Engine* [72] que se ejecuta por debajo del *Notebook*. En este aspecto, otra ventaja es que el servicio viene instalado con más librerías de las que vendrían por defecto en una instalación normal de Python [57], lo que evita que el usuario tenga que interactuar con la máquina virtual subyacente para descargarse librerías que periten implementar la funcionalidad que necesita.

- Una ventaja de desarrollar en un *Notebook* mediante un servicio en la nube, es que al contar este con el respaldo de *Google*, se proporcionan algunos recursos hardware adicionales, por ejemplo, una GPU [64] (*Graphics Processing Unit*), e incluso una TPU (*Tensor Processing Unit*) [65], lo que es beneficioso para entrenar gratuitamente modelos de aprendizaje automático o profundo, construir aplicaciones gráficas, o acelerar programas mediante librerías que permiten realizar una programación eficiente, como *PyOpenCL*, o *PyCuda*.

- El servicio ya viene preparado para integrarse con la plataforma de GitHub, de forma que si lo que se desea es llevar un control sobre el desarrollo de un *Notebook* concreto, y/o permitir que este pueda ser accedido públicamente por otros usuarios, esto puede hacerse fácilmente. El servicio también está integrado con *Google Drive* [70], lo que permite que los usuarios puedan guardar sus *Notebooks* de forma privada, o utilizar este servicio para la generación de copias de seguridad de sus propios proyectos.

- Estos *Notebooks* cuentan con una sintaxis especial que amplía su capacidad, posibilitando el uso de elementos visuales (*widgets*) propios de las interfaces de usuario interactivas, lo que hace que los documentos sean menos estáticos y se comporten de una forma más similar a una aplicación basada en el uso de formularios. Como alternativa, existe la posibilidad de emplear librerías que dan soporte al uso de *widgets*, como por ejemplo *ipywidgets*, que puede utilizarse en los *Notebooks* de *Colab* [73], y en los de *Jupyter* [74].

6.2.9 FiftyOne

FiftyOne [77] es un software, y una librería escrita para el lenguaje Python [57], creada por la comañía Voxel51 en 2020, cuya labor consiste en crear herramientas que faciliten el trabajo a los ingenieros dedicados a la Visión por Computador (*Computer Vision*) [2] y al Aprendizaje Automático (*Machine Learning*) [17], como por ejemplo: balancear y exportar datasets, anotar imágenes, y analizar las predicciones de los modelos. Es decir, herramientas que les permitan experimentar rápidamente con modelos que emplean datos para el aprendizaje.

En el caso de *FiftyOne*, esta es una herramienta de código abierto liberada bajo la licencia Apache 2.0 [61] que permite:

- Cargar y visualizar conjuntos de datos codificados en diversos formatos con muy poco esfuerzo. Solo con un par de líneas de código puede cargarse y visualizarse un *dataset* propio, o de internet, por ejemplo COCO (*Common Objects in Context*) [68], u Open Images [69], por lo que deja de ser necesario programar código fuente para realizar estas tarreas manualmente.
- Pasar de un formato de codificación de datos a otro. Es posible trabajar con los formatos de datos (*metadatos*) de los *datasets* COCO, KITTI, PASCAL VOC [43, 44], etc. E incluso realizar una conversión entre los formatos de estos.
- Analizar conjuntos de datos. Para ello proporciona gráficas y un lenguaje de consulta de datos propio, capaz de realizar cálculos y búsquedas avanzadas dentro de un dataset, y sobre los campos (*metadatos*) de las muestras que lo componen.
- Analizar modelos de Machine Learning [17]. Proporciona mecanismos para realizar una evaluación cuantitativa y fiel del desempeño de los modelos mediante matrices de confusión, e informes que detallan el AP (*Average Precision*) del modelo respecto de cada categoría de los datos, y su mAP (*mean Average Precision*). Estos informes se pueden visualizar gráficamente.
- Análisis automático del etiquetado de *datasets*. La función *mistakeness* sirve para indicar para cada una de las muestras, si la etiqueta de una muestra es correcta, como de inexacto es su *ground truth bounding box*, y un booleano que indica si el supuesto objeto etiquetado existe. Esta función también es capaz de detectar en algunos casos la presencia en imágenes de elementos reconocibles que no han sido etiquetados.

6.2.10 COCO Dataset

COCO (*Common Objects in Context*) [68], u Objetos Comunes en Contexto, es un *dataset* de imágenes publicado en 2014 por la empresa Microsoft con la finalidad de proporcionar a los ingenieros que trabajan en Visión por Computador (*Computer Vision*) [2] y Aprendizaje Automático (*Machine Learning*) [17], un set de imágenes con el que poder entrenar modelos para el reconocimiento y la segmentación de objetos, con el fin de promover la investigación y los avances en reconocimiento de imágenes. Actualmente el proyecto que supone este *dataset* de imágenes continúa siendo perfeccionado, siendo su última actualización importante en el año 2017.

Normalmente COCO (*Common Objects in Context*) [68] se utiliza para comparar tanto algoritmos de Visión por Computador (*Computer Vision*) [2], como modelos de Deep Learning [3], de los cuales se espera que puedan resolver el problema de la detección de objetos en imágenes en tiempo real.

Algunas de las características principales de este *dataset* de imágenes son:

- Está sujeto a una licencia *Creative Commons* (CC BY 4.0) [80].
- Cuenta con un total de 330.000 imágenes, de las cuales más de 200.000 ya han sido etiquetadas.
- Incluye:
  - Al menos 5 etiquetas por imagen.
  - 80 categorías de objetos disponibles.
  - 3 conjuntos de imágenes: entrenamiento, validación, y test.
  - 1,5 millones de ejemplos de objetos, de entre los cuales también los hay de elementos superpixelados que han sido etiquetados y segmentados.
- Emplea un formato de datos (*metadatos*) propio para el almacenamiento de la información relativa a las imágenes que lo componen, lo que requiere usar una librería o software que pueda interpretar dicha información para trabajar con él, por ejemplo *FiftyOne* [77].

En el Apéndice 2 de este trabajo se incluye información adicional que detalla todas las categorías de objetos disponibles en el *dataset*, así como la cantidad de objetos e imágenes que están vinculadas a cada una de ellas.

6.3 La plataforma

A continuación se introducen los diseños conceptuales de los programas que se han desarrollado para crear la plataforma que cubre el ciclo de trabajo con un detector de objetos. Los diseños han sido elaborados aplicando el estándar de modelado UML (*Unified Modeling Language*) [76], y cada uno está acompañado de una explicación que aumenta el nivel de detalle del programa correspondiente.

6.3.1 COCO Data-Set Generator (CDSGenerator)

El programa *CDSGenerator* sirve para poder construir de forma automática *datasets* de imágenes con los que entrenar detectores de objetos, lo que supone un ahorro de tiempo significativo para el responsable de su configuración y entrenamiento.

Esta automatización es posible gracias a *FiftyOne* [77], un software de código abierto, y librería de Python, que se ha utilizado en el programa para generar *datasets* de imágenes basados en COCO (2017) (*Common Objects in Context*) [68], por lo que puede verse como un generador de subconjuntos de dicho *dataset*.

El funcionamiento de este programa se resume de forma conceptual en las figuras 6.1, 6.2, y 6.3, que representan su diagrama de casos de uso, una captura del aspecto de su interfaz gráfica de usuario, creada con la librería *ipywidgets*, y su diagrama de despliegue. Las figuras 6.1 y 6.2 son las más relevantes para entender el funcionamiento del programa, en la primera puede verse una descripción de las acciones que el usuario puede realizar al interactuar con él, y en la segunda, su interfaz gráfica de usuario, que sirve para esclarecer mejor cómo es la interacción del usuario con el programa.

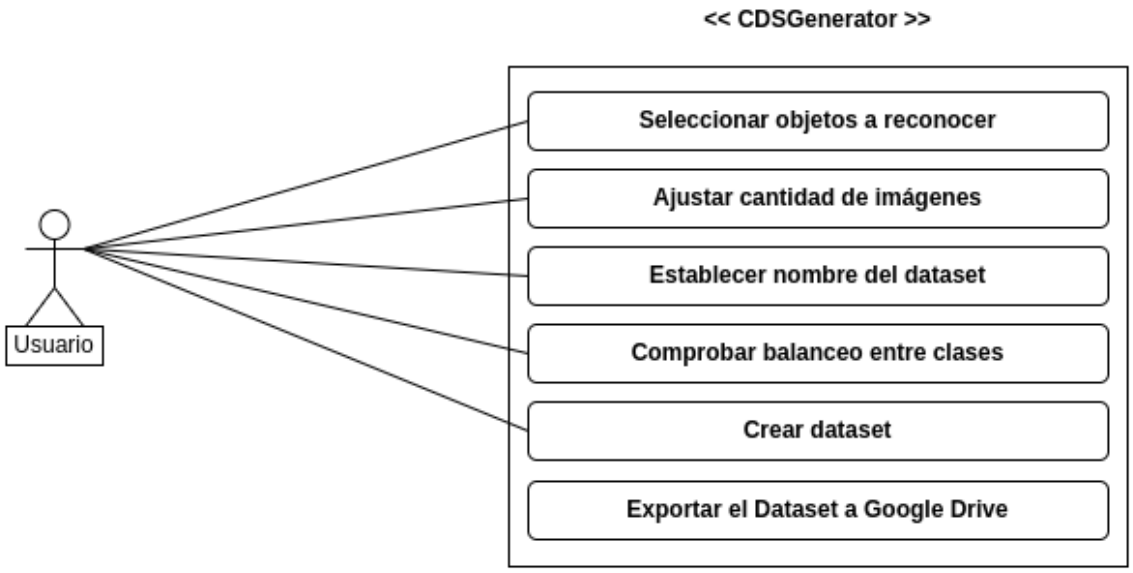

Figura 6.1: Diagrama de casos de usos del programa *CDSGenerator*

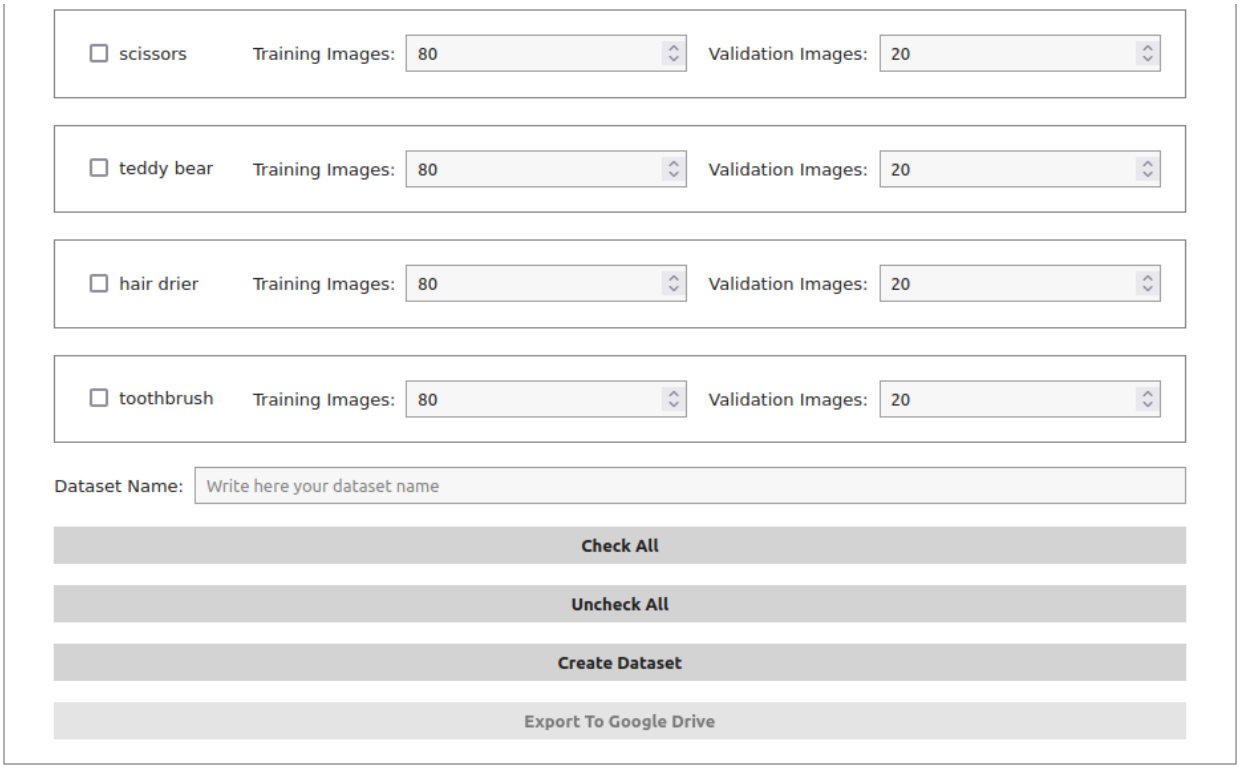

Figura 6.2: Interfaz de usuario del programa *CDSGenerator*

A continuación, en la figura 6.3 se puede ver el diagrama de despliegue del programa *CDSGenerator*, este diagrama ilustra cuál es la relación existente entre el usuario y los servicios, entre los servicios, y la forma en que estos se comunican.

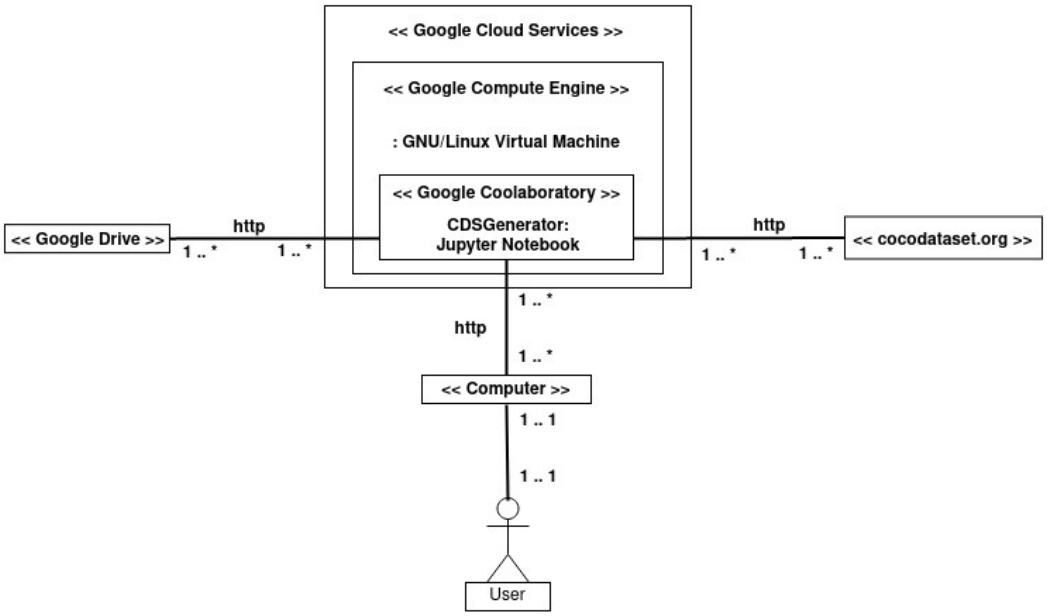

Figura 6.3: Diagrama de despliegue del programa *CDSGenerator*

A modo de resumen, el manejo de este programa por parte de un usuario se realiza de la siguiente manera. El usuario abre el *Notebook* desde su *Google Drive* [70], lo que le lleva a abrir el servicio de *Google Colaboratory* [73], que se sustenta en una máquina virtual que proporciona la infraestructura del servicio. Lo primero que ha de hacerse es ejecutar todas las celdas del *Notebook*, ya que este se ejecutará más bien cómo un programa basado en formularios, que como un *Notebook* al uso. Una vez se ha cargado el contenido de todas las celdas, el programa estará listo para usarse. Este es de un uso muy sencillo, solo han de marcarse los nombres de las categorías de objetos que se desea que estén presentes en el *dataset*, y luego para cada una de dichas categorías, lo que se ha de hacer es indicar el número de imágenes que se desea añadir tanto al conjunto de entrenamiento, como al de validación. Finalmente, queda comprobar con el visor de *FiftyOne* [77] qué número objetos entre clases está balanceado, ponerle un nombre al *dataset*, y seleccionar el botón de exportar a *Google Drive*. Para una descripción más completa de cómo utilizar este programa, consultar el Apéndice 1.

6.3.2 Detector Tool (Dettool)

El programa *Dettool* sirve para poder entrenar y validar modelos de detectores de objetos soportados por la *TensorFlow Object Detection API* [67], con *datasets* que han sido generados con el programa *CDSGenerator*.

El despliegue de esta aplicación es similar al de *CDSGenerator*, pero en este caso, desaparece el acceso al *dataset* de COCO (*Common Objects in Context*) [68] vía internet, y se añade un almacén de datos (*bucket*) proporcionado por el servicio de *Google Cloud Storage* [78], tal y como puede verse en la figura 6.4. Este almacén de datos (*bucket*) se hace totalmente necesario para proporcionar acceso a los ficheros de un detector, tanto a la GPU [64], como a la TPU [65], ya que esta última no tiene acceso al sistema de ficheros de la máquina virtual que ejecuta el *Notebook* de *Dettool*, por lo que no se podría trabajar con ella. Dentro de *Google Compute Engine* [72], las TPU se ofrecen como otro servicio.

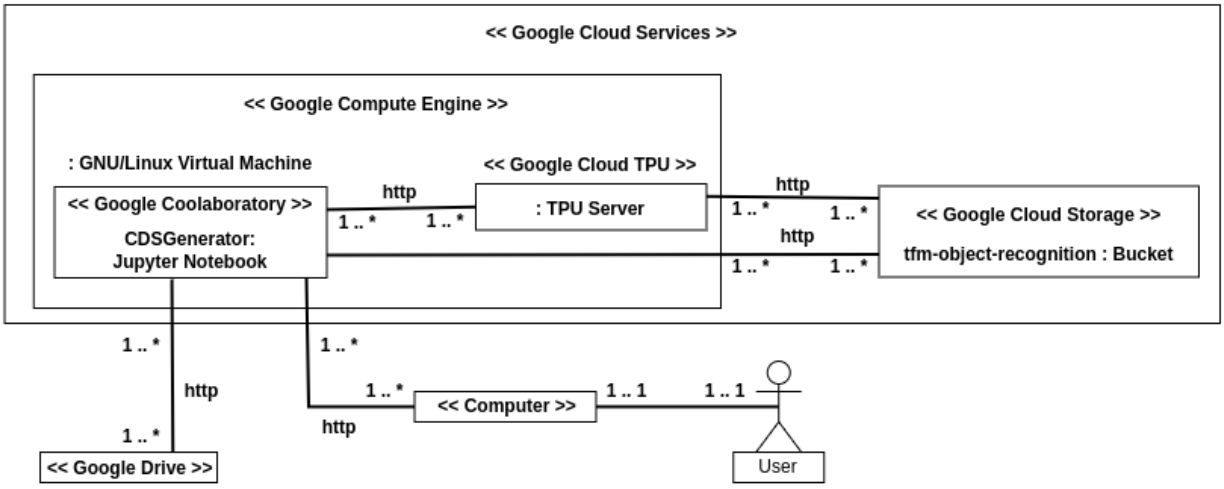

Figura 6.4: Diagrama de despliegue de la aplicación *Dettool*

La interacción de un usuario con este programa se resume a través de su diagrama de casos de uso, figura 6.5, el cual se complementa a su vez con la figura 6.6, que muestra su interfaz de usuario para ayudar a entender mejor su funcionamiento.

Esta interacción del usuario con el programa se puede describirse de forma breve como sigue. Primero se escogería un detector de objetos, y luego se procedería a la configuración de los múltiples hiper-parámetros. Los pasos (*steps*) de calentamiento (*warmup*) y post-calentamiento (*base*), así como el factor de aprendizaje (*learning rate*) con el que se empieza en ambas fases.

El factor de fuerza del aprendizaje (*momentum*), el tamaño del *batch* (*mini-batch*), las opciones de *data augmentation* para conseguir más imágenes para el proceso de entrenamiento. La cantidad de pasos a dar para validar el detector de objetos (*validate each*), y el umbral de validación (*validation thresold*).

Finalmente, se elige un *dataset* que haya sido generado con *CDSGenerator* para entrenar el detector, y se presiona el botón *Train & Validate*. Después de esto se obtienen en *TensorBoard* [79] resultados gráficos del entrenamiento y la validación.

Un detalle más exhaustivo del uso de *Dettool* se muestra en el apéndice 3.

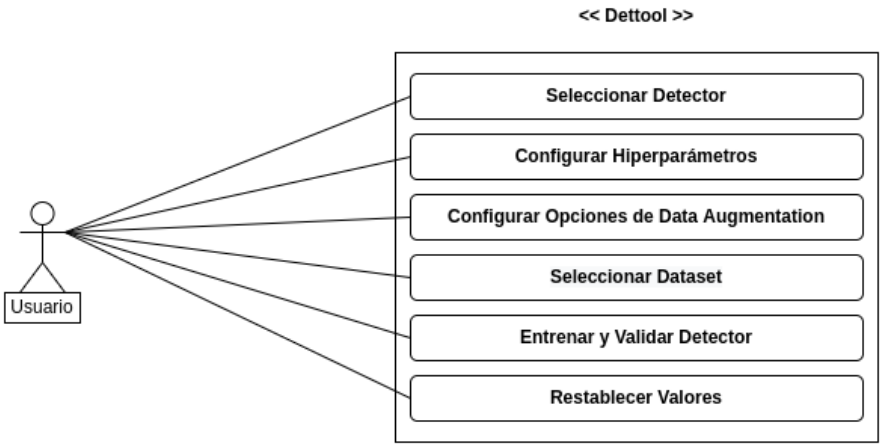

Figura 6.5: Diagrama de casos de uso del programa *Dettool*

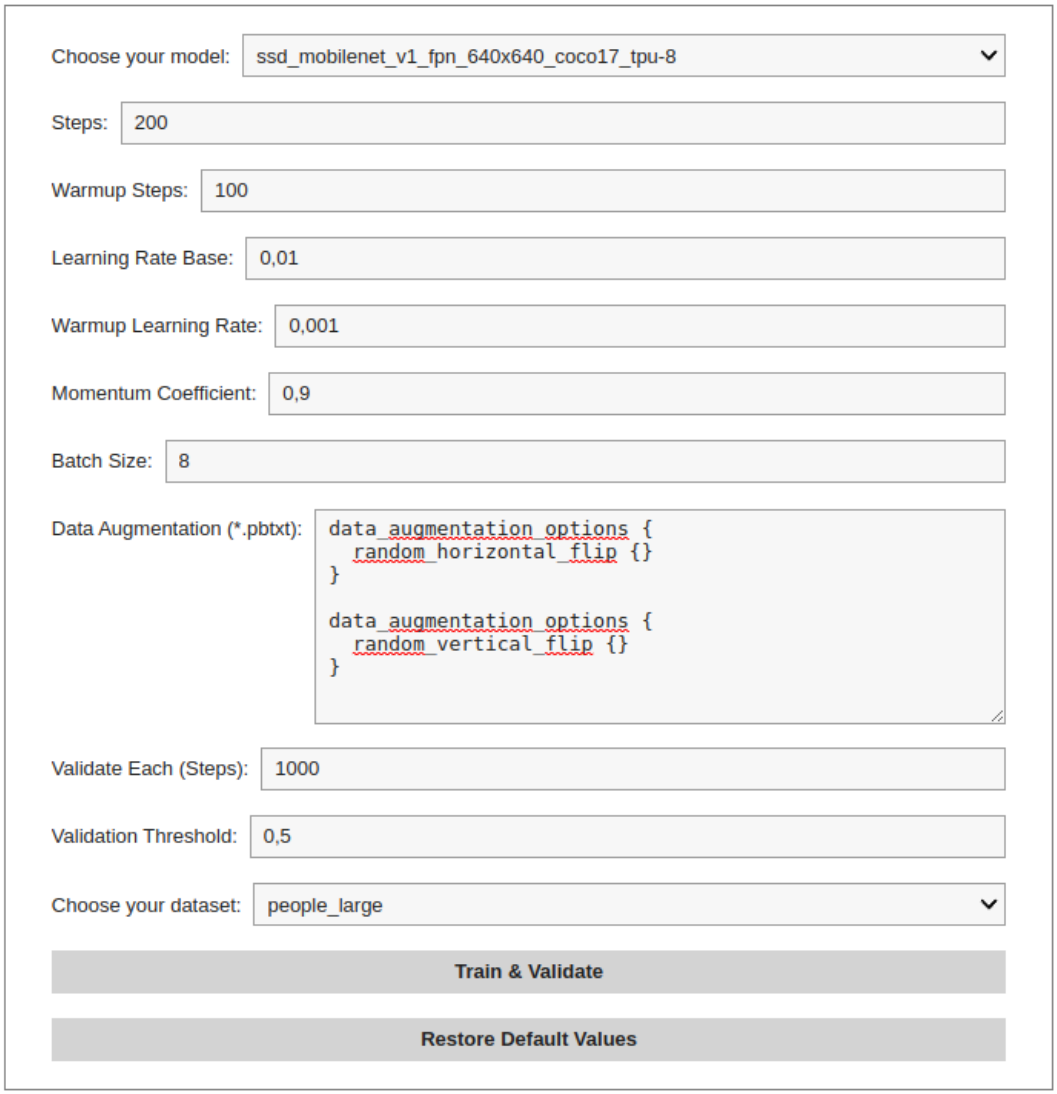

Figura 6.6: Interfaz de usuario del programa *Dettool*

6.3.3 Detector Tester (*Dettest*)

Dettest es un programa para comprobar el funcionamiento real de un detector de objetos en algún tipo de sistema, por ejemplo, computador, un microcomputador, o un microcontrolador. Como este programa no se ha creado sobre los servicios de *Google Cloud Platform* [71], se ha sustituido su diagrama de despliegue por un diagrama de clases, ya que el despliegue de una aplicación en un computador es de menor interés, al ser una actividad típica entre desarrolladores. Sin embargo, su estructura de clases sí resulta interesante, ya que permite visualizar de una forma resumida las partes que lo componen, la interacción entre ellas, y las operaciones que cada una realiza. La imagen de la figura 6.7 muestra la estructura interna del programa a partir de su correspondiente diagrama de clases.

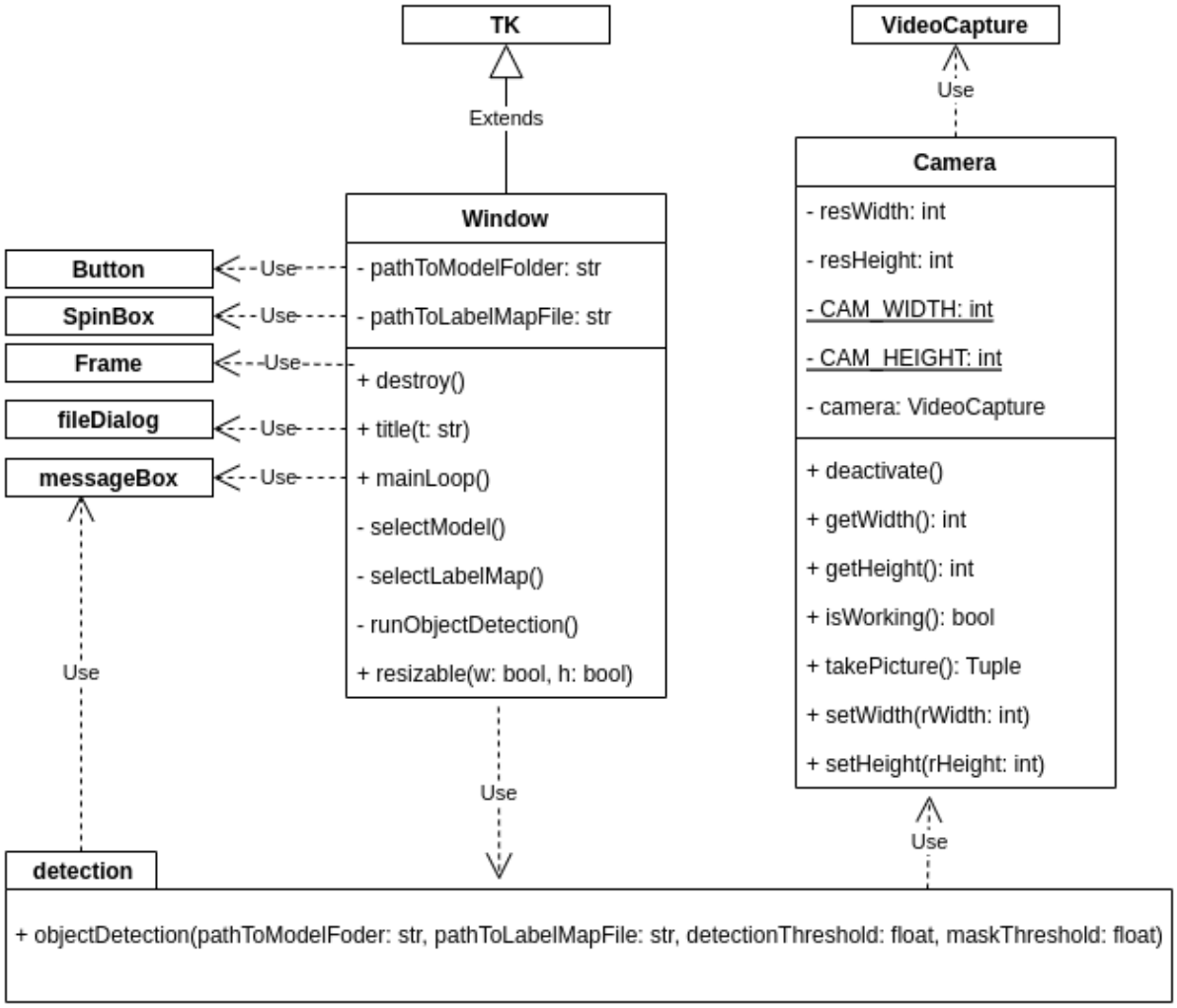

Figura 6.7: Diagrama de clases del programa *Dettool*

A continuación, se muestra el diagrama de casos de uso de la aplicación, figura 6.8, y su interfaz gráfica de usuario, figura 6.9, ambas con la intención de permitir que se pueda vislumbrar mejor cómo será la interacción del usuario con el programa.

En lo que a esta aplicación respecta, no se incluirá una sección de apéndice al final para explicar su instalación y su uso, ya que ambas son muy simples. La instalación simplemente requiere ejecutar como súper usuario en una *shell* de *bash* el *script* install.sh, situado en la ruta *ObjectRecognition\programs\dettest*. Este *script* ya está preparado para instalar tanto las librerías del sistema que sean necesarias, como las librerías de Python [57].

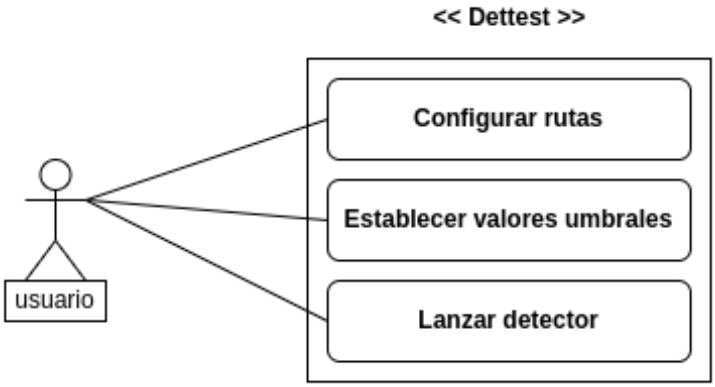

Figura 6.8: Diagrama de casos de uso de *Dettest*

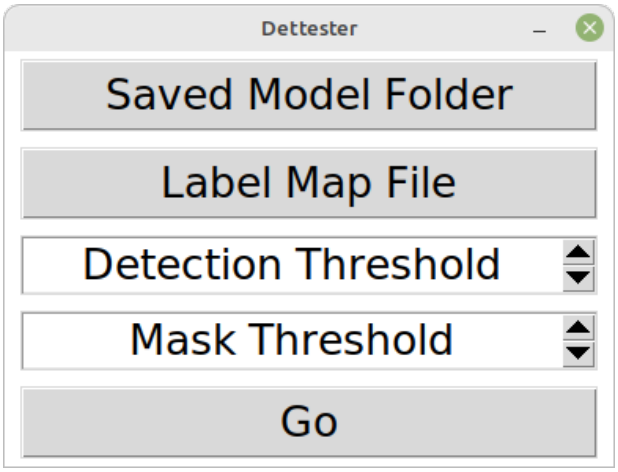

Figura 6.9: Interfaz de usuario del programa *Dettest*

Una vez instaladas las dependencias de la aplicación, para utilizarla, se puede, por un lado, lanzar desde la *shell* de *bash* del sistema, o bien a través de su interfaz gráfica de usuario.

Para lanzarla desde el intérprete de *bash*, bastará con ejecutar el comando: python3 <pathToModelFolder> <pathToLabelMapFile> <detectionThreshold> <maskThreshold>. Los parámetros de este comando son, la ruta hasta la carpeta donde se encuentra el modelo, es decir, hasta su archivo *saved_model.pb*, y la ruta hasta su fichero *label_map.pbtxt*, el cual contiene la correspondencia entre el nombre de cada clase del problema de clasificación, y un número entero que actúa a modo de identificador unívoco de cada clase. Después se introducen los valores umbrales, que son dos números reales en el rango [0 – 1]. El primero indica a partir de qué grado de confianza de una detección, esta debe mostrarse. El segundo indica cuál debe ser como mínimo el grado de confianza de un píxel concreto de una imagen, como para que este sea considerado como perteneciente a un objeto.

En caso de querer lanzar la aplicación a través de su interfaz gráfico de usuario, bastará con lanzar el comando: python3 view.py, que lanzará la interfaz vista en la figura 6.9, la cual hace mucho más fácil la ejecución del programa al ofrecer cajas de texto, y los correspondientes cuadros de diálogo. El fichero *readme.md*, que se encuentra bajo la ruta *ObjectRecognition\programs\dettest*, tiene un recordatorio de cómo realizar tanto instalación como la ejecución, esta última cualquiera de sus dos formas. En esa misma carpeta, se incluye un modelo de detector ligero, *SSD Mobilenet v2*, que servirá para experimentar este programa cuando no se disponga de un detector de objetos propio.

Cabe hacer un último inciso respecto al funcionamiento de esta aplicación, y es que la cámara con la que se grabará será la cámara que venga integrada en su computador, a menos que se posea una cámara conectada por USB, en cuyo caso, esta cobrará más prioridad, y será la que se ejecute.

Capítulo 7 - Resultados

7.1 Introducción

En el presente capítulo se realiza una descripción de los aspectos más importantes, tanto positivos como negativos, vinculados al desarrollo y al funcionamiento de los programas que forman la plataforma para la automatización del ciclo de trabajo con detectores de objetos, además de algunos resultados de detectores que han sido entrenados, validados, y posteriormente probados con ella.

7.2 La plataforma

Aunque durante el primer mes de desarrollo los programas estaban pensados para desarrollarse en modo local, en un ordenador portátil con Python3 [57], la librería *OpenCV*, y el *framework* ofrecido por la *TensorFlow Object Detection API* [67], pronto la idea quedó descartada ante un par de problemas que impidieron continuar con este enfoque:

- Prestaciones insuficientes del ordenador portátil: aunque el hardware que se enuncia a continuación sería más que suficiente para la realización de varios tipos de proyectos y programas sin ningún problema, su principal carencia de cara trabajar con *Deep Learning* residió en la poca capacidad de memoria de la GPU [64] (4 GB), lo que provocó que los modelos sí se pudieran testear, pero a duras penas entrenar, ya que cuando estos se cargaban en memoria, apenas quedaba espacio restante para tener suficiente capacidad para manejar el *mini-batch-size*, oscilando este entre 1 y 10, valores demasiado bajos que no servirían para ajustar los parámetros de un detector lo suficiente como para que este convergiese hacia una solución aceptable. Concretamente, las características del sistema en cuestión fueron:
    - Portátil y sistema operativo: DELL XPS 15; Linux Mint 20.3 Cinnamon.
    - Procesador, núcleos y frecuencia: Intel Core i9 de 8 núcleos a 2,40GHz.
    - Memoria RAM: 2 bancos RAM DDR4 de 8 GB a 2666 Mhz.
    - Tarjeta gráfica: NVIDIA GeForce GTX 1650, con 4 GB de capacidad.
    - Disco duro: 512 GB (SSD; *Solid State Drive*).

- Incompatibilidad de *OpenCV* con detectores de algunos *frameworks*: aunque la documentación de la librería afirmaba ser compatible con los detectores de *TensorFlow* [59], *Caffe*, *OXNN*, *DarkNet*, etc. Ya fuera directamente, o aplicando una transformación, se experimentaron dificultades, pues los modelos de *TensorFlow* soportados, eran los de la versión 1 de esta API. Por otra parte, ningún modelo se pudo exportar o importar de *OXNN*, aunque si había un buen funcionamiento con los modelos de *DarkNet* o *Caffe*.

Este par de inconveniencias, sumadas a la incomodidad de tener que generar varios *scripts* de instalación para dar soporte a la GPU [64] con CUDA (*Compute Unified Device Architecture*) [63], las librerías de Python3, y del sistema operativo, propiciaron el replanteamiento del proyecto, de forma que este pasó a desarrollarse y desplegarse casi en su totalidad sobre los servicios de *Google Cloud Platform* [71], como ya se describió en los apartados del capítulo anterior.

La razón por que la plataforma desarrollada se creó principalmente sobre el servicio de *Google Colaboratory* [73], se debió a que este proporciona unos recursos hardware y software muy buenos. En primer lugar, el servicio permitía continuar con Python3 como lenguaje de desarrollo, por lo que no hubo necesidad de realizar una migración en lo que respecta a la tecnología de la programación. Y, en segundo lugar, los recursos de hardware que el servicio proporciona son bastante adecuados para trabajar con reconocedores de objetos. Puede verse un resumen de estos recursos en la imagen de la figura 7.1.

Hardware / Subscription	Google Colab Free	Google Colab Plus
GPU	K80, T4	K80, T4, V100, P100
TPU	Yes (Unknow model)	Yes (Unknow model)
RAM	12 GB	35 GB
Hard Disk	78 GB	225 GB
Cost	Free	9,25 € / month

Figura 7.1: Prestaciones de *Colab* en función de la subscripción al servicio

Aunque en un principio se empleó la versión gratuita de *Colab*, finalmente se utilizó la versión *Plus* del servicio para la implementación del proyecto, lo que se debió a las limitaciones de la versión gratuita, y al bajo precio a pagar por la mejoría.

En este punto, antes de procederse a la descripción de los resultados y problemas obtenidos en cada programa, se hará hincapié en el principal problema que padecen los programas que han sido desarrollados con *Google Colab* [73], que recordemos, son: *CDSGenerator*, y *Dettool*. Esta problemática que afecta a ambos programas tiene que ver con la forma en que ambos han sido construidos, e implica que los dos pueden experimentar desconexiones del servicio de *Google Colab*, más allá de una caída fortuita de internet que podría ocurrirle a un usuario. Estas desconexiones de los programas podrían darse por las dos siguientes razones:

- Cuando se hace uso del servicio de *Google Colaboratory* para el desarrollo de aplicaciones, lo habitual es programar y ejecutar el código de las celdas, y cuando se detecta que ya no hay interacción con ninguna celda de código transcurrido un cierto tiempo, el *Notebook* recibe automáticamente una señal de desconexión para propiciar el ahorro o la redistribución de recursos de la nube de *Google*. Normalmente eso no debería ser problemático, sin embargo, la dificultad de simularse un programa de escritorio es que este solo ejecuta cada celda una única vez para realizar tanto la carga de recursos, como de la interfaz de usuario. Luego la aplicación se usa desde la región de *display* (salida) de su propia celda. Esta forma de trabajar permite que el *Notebook* esté un buen rato conectado al servicio de *Colab* y funcionando, sin embargo, como no se vuelve a ejecutar ninguna celda más, con el paso del tiempo esto se interpreta como que el servicio ha permanecido inactivo, por lo que se lanza una señal de desconexión que aborta su funcionamiento. El tiempo transcurrido para que un *Notebook* sea desconectado por esta razón es variable, pues se han experimentado interrupciones entorno a una hora.
- Lo más natural en *Google Colab* es desarrollar códigos medianamente ligeros en las celdas para que su ejecución acabe rápidamente. Sin embargo, cuando una celda pasa demasiado tiempo ejecutándose en *Colab*, el propio servicio emite una señal de desconexión, ya que esta cantidad de tiempo que una celda puede pasarse ejecutándose de continuo varía en función del tipo de suscripción al servicio, e independientemente de eso, nunca se garantiza una ejecución indefinida. En lo que a esto respecta, se ha experimentado que las desconexiones tienen lugar en torno a las dos horas de ejecutarse una celda, por lo que según pase ese tiempo, más probable será sufrir una desconexión.

A continuación, se introduce para cada programa que integra de la plataforma de trabajo cada uno de sus aspectos más fundamentales, las problemáticas que fueron encontradas, cuales se han podido resolver, y cuales aún no se han resuelto.

7.2.1 COCO Data-Set Generator (CDSGenerator)

En el caso del programa *CDSGenerator*, los aspectos más relevantes de este hacen referencia a su capacidad para poder generar grandes *datasets* de objetos ya etiquetados y balanceados, que puedan exportarse al formato de *TensorFlow* [59] (**.records*) para conseguir entrenamientos más rápidos de los modelos, ya que este formato almacena en forma de registros binarios, una representación de las imágenes del *dataset* que pueden ser manipuladas eficientemente. En lo que a esto respecta, los resultados obtenidos han sido los siguientes:

- La capacidad para generar *datasets* de objetos depende por completo de la velocidad de la conexión a internet, lo que es debido a los 2 problemas de desconexión comentados antes de este apartado. En este caso, se ha podido emplear conexión inalámbrica 5G con una velocidad aproximada de bajada de 300 Mbs, lo que ha permitido que puedan crearse 4 *datasets* de diferentes tamaños para entrenar los modelos. Los *datasets* creados se representan a a continuación por sus archivos de *summary.json*, figuras 7.2, 7.3, 7.4, y 7.5.


```
{
    name: "people_large",
    creation: "08/04/2022 15:35:12",
    num_classes: 1,
    class_mames: [
        "person"
    ],
    total: {
        images: 18600,
        training_images: 16000,
        validation_images: 2600,
        detections: 75880,
        training_detections: 65277,
        validation_detections: 10603
    },
    training: {
        person: {
            images: 16000,
            detections: 65277
        }
    },
    validation: {
        person: {
            images: 2600,
            detections: 10603
        }
    }
}
```


Figura 7.2: Archivo *summary.json* del *dataset people_large*

```
{
    name: "transports",
    creation: "11/04/2022 01:44:19",
    num_classes: 8,
  ▼ class_mames: [
            "bicycle",
            "car",
            "motorcycle",
            "airplane",
            "bus",
            "train",
            "truck",
            "boat"
    ],
  ▼ total: {
            images: 11704,
            training_images: 10690,
            validation_images: 1014,
            detections: 30354,
            training_detections: 27248,
            validation_detections: 3106
    },
  ▶ training: {  8 items  },
  ▶ validation: {  8 items  }
}
```


Figura 7.3: Archivo *summary.json* del *dataset transports*


```
{
    name: "electronics",
    creation: "11/04/2022 15:36:56",
    num_classes: 6,
  ▼ class_mames: [
            "tv",
            "laptop",
            "mouse",
            "remote",
            "keyboard",
            "cell phone"
    ],
  ▼ total: {
            images: 3626,
            training_images: 3029,
            validation_images: 597,
            detections: 8579,
            training_detections: 7256,
            validation_detections: 1323
    },
  ▶ training: {  6 items  },
  ▶ validation: {  6 items  }
}
```


Figura 7.4: Archivo *summary.json* del *dataset electronics*

```
{
    name: "people_short",
    creation: "08/04/2022 15:44:11",
    num_classes: 1,
    class_names: [
        "person"
    ],
    total: {
        images: 100,
        training_images: 80,
        validation_images: 20,
        detections: 455,
        training_detections: 363,
        validation_detections: 92
    },
    training: {
        person: {
            images: 80,
            detections: 363
        }
    },
    validation: {
        person: {
            images: 20,
            detections: 92
        }
    }
}
```


Figura 7.5: Archivo *summary.json* del *dataset people_short*

- En lo que respecta al uso de la librería de *FiftyOne* [77] para conseguir *datasets* automáticamente etiquetados, se ha comprobado que es posible la creación de estos muy sencillamente, ya que apenas con unas pocas instrucciones de esta librería es posible descargar y generar automáticamente *datasets* a partir de otros disponibles a través de internet, como COCO (2017) (*Common Objects in Context*) [68], para su posterior visualización, análisis y exportación. Lo que evita tener que hacer manualmente la descarga de los *datasets*, o incluso tener que programar software adicional de tipo *web scrapping* para automatizar el proceso de búsqueda y descarga de imágenes, así como el código pertinente para para interpretar el formato de datos del *dataset* descargado para poder trabajar con él. Tampoco requiere software adicional para el etiquetado de las imágenes, ya que las que extrae de los *datasets* disponibles a través de internet ya están completamente anotadas. Todas estas razones han hecho que *FiftyOne* sea el software más adecuado para impulsar el programa CDSGenerator, ya que es muy completo y simplifica mucho la cantidad de actividades que habrían de realizarse.

- En cuanto a la generación de *datasets* balanceados por parte del programa, esto es relativamente posible de conseguir, aunque a veces es complicado de llevar a cabo, pues como se dijo anteriormente, las imágenes se descargan aleatoriamente a través de *FiftyOne*, y algunas categorías de objeto aparecen con más frecuencia y en mayor cantidad que otras. Si a esto se le añade que el dataset de COCO (2017) [68] no está bien balanceado, como se puede ver en las tablas del apéndice 2, entonces se han de intentar varias descargas para lograr un balanceo aceptable entre las categorías de los objetos del conjunto de entrenamiento (*training*). Los dos *dataset* de *people* mostrados antes ya están balanceados al estar únicamente conformados por una sola clase, por otra parte, en el de *transports*, la diferencia de detecciones entre la clase que más tiene, y la que menos, es de 181 (*training*), y en el de electronics, 122.
- Otro problema vinculado a la generación de datsets mediante *FiftyOne* [77], es que la librería está diseñada para que, dada una lista de clases, y un número de imágenes que se desea descargar para formar un *dataset*, se prioriza la descarga de aquellas que tienen el mayor número de clases seleccionadas presentes. Una estrategia inadecuada, pues favorece que se creen *datasets* desbalanceados. Para corregir este comportamiento, se ha programado una estrategia sencilla que consiste en crear un *dataset* vacío y luego realizar una petición de descarga de imágenes por cada categoría en la que se indica el número de imágenes a descargar. De esta manera puede hacerse que cada descarga asociada a una categoría se almacene en un *dataset* auxiliar cuyo contenido luego es añadido al *dataset* inicialmente vacío, lo que permite crear un *dataset* más balanceado.
- Respecto al balanceo de los *datasets*, de cara al conjunto de validación, esto es prácticamente imposible, ya que la cantidad de detecciones (objetos) que hay presentes en este conjunto entre categorías es demasiado dispar, y aún más, en caso de que algunas categorías guardaran una diferencia reducida de detecciones, no habría suficientes imágenes como para poder mantener la regla 80% (*training*) – 20% (validation), u otras reglas que indiquen alguna distribución aceptable de imágenes entre ambos conjuntos de datos.

- Otro aspecto a tener el cuenta es el espacio de almacenamiento del que se dispone en el servicio de almacenamiento *Google Drive* [70], 15 GB, suficiente como para formar los *datasets* mediante *FiftyOne* [77] debido a que la cantidad de tiempo para formarlos es limitada, ya que de lo contrario, con un tiempo de conexión indefinido para poder formar los *datasets*, sería posible generar algunos cuyo tamaño provocarían que rápidamente se agotase el espacio en Drive, e incluso podría generarse uno que abarcara todo el espacio del que se dispone en el servicio. Estos aspectos anteriores sugieren que es mejor trabajar con un *dataset* que esté conformado por un número de imágenes suficientemente grande como para poder entrenar modelos de detectores de objetos, pero no tan extenso como para que el usuario tienda a querer generar *datasets* que agoten muy rápido el espacio disponible en Drive, o cuyos tiempos de descarga no sean factibles. Por esa razón se ha preferido emplear el *dataset* COCO (*Common Objects in Context*) [68] frente a otros como por ejemplo Open Images [69] como base para la generación automática de *datasets*, ya que ambos pueden manipularse a través de *FiftyOne*, pero COCO es más ad-hoc a este propósito al ser más reducido. Finalmente, cabe a destacar que, en caso de ser necesario, los 15 GB gratuitos de *Drive* pueden ser ampliables, aunque esta solución haría necesaria realización de un estudio de costes adicional.
- En cuanto a la generación de los ficheros **.records* para poder obtener unas representaciones de las imágenes de los *datasets* que puedan ser eficientes de manipular. Al principio se empleó el software de *FiftyOne* [77] para realizar la conversión de las anotaciones del formato JSON [75] de COCO [68], al formato de registros binarios de *TensorFlow* [59], sin embargo, pronto se pudo comprobar en la documentación de *FiftyOne* que este no exportaba las máscaras de bits, por lo que se creó el *script create_coco_tf_records.py*, una modificación del *script create_coco_tf_record.py* de la *TensorFlow Object Detection API* [67]. La diferencia del *script create_coco_tf_records.py* con respecto al que ya está por defecto en la *TensorFlow Object Detection API*, es que todos los registros binarios van en un mismo archivo **.records*, en lugar de en varios archivos.

7.2.2 Detector Tool (*Dettool*)

En cuanto al programa para entrenar, validar, y exportar modelos de detectores de objetos, *Detool*. Sus aspectos más relevantes giran, por una parte, en torno a las prestaciones hardware que este pueda proporcionar para entrenar de la manera más eficiente posible a los detectores disponibles en la *TensorFlow Objet Detection API* [67]. Y por otra parte, en torno a la capacidad de este mismo programa para poder realizar entrenamientos precisos, y generar resultados que permitan entender cómo de bien se ha ajustado el modelo tras pasar por los procesos de entrenamiento y validación, para así saber si el ajuste realizado puede mejorarse, o si está bien realizado. En lo que a estos aspectos respecta, los resultados obtenidos en el caso de *Dettool* han sido los siguientes:

- El programa cuenta con un buen respaldo de hardware, como se ha expuesto en la tabla de la figura 7.1, donde puede observarse que se disponen de 35 GB de RAM, 225 GB de disco duro, uno entre varios modelos de GPU [64] con 16 GB de capacidad, y una TPU (*Tensor Processing Unit*) [65], cuyo modelo no es conocido. Aunque todo este hardware, a priori, debería ser suficiente como para entrenar un detector, lo cierto es que para algunos modelos el recurso de la cantidad de memoria de la GPU, o la TPU, siguen siendo insuficientes. Tal es el caso de los modelos de detector *Faster R-CNN ResNet 50 v1*, que solo puede entrenarse con un total de 28 imágenes (*mini-batch-size*) utilizando la TPU, y el detector *Mask R-CNN Inception ResNet v2*, cuyo *pipeline* en principio solo le permite ser entrenado por una o varias *GPUs* en paralelo, pero como no se dispone de más de una, y el tamaño de las imágenes con las que trabaja es muy grande (1024 x 1024), solo se dispone de una capacidad de 2 imágenes (*mini-batch-size*) al utilizar la GPU disponible. Cabe plantearse, en los casos de ambos modelos, si la insuficiencia de memoria es debida completamente a la magnitud de estos, que la ocupan en demasía, o a una posible implementación ineficiente en lo que respecta a la reserva de memoria. Por otra parte, para detectores más ligeros, como por ejemplo *SSD Mobilenet v1*, y *SSD Mobilenet v2*, se puede emplear un tamaño de *mini-batch-size* de hasta 512 imágenes, un resultado bastante positivo.

- Un aspecto ineficiente respecto al uso del hardware, ha sido plantear que los procesos de entrenamiento y validación se ejecuten de forma secuencial, en lugar de ejecutarse de forma simultánea para favorecer el ahorro de tiempo, sin embargo, se ha de considerar que los *Notebooks* han sido creados para la ejecución de código en serie, y no en paralelo, por lo que en principio ha sido la única manera de dar soporte a la ejecución de ambos, lo que se ha conseguido mediante la modificación del *script model_lib_v2.py*, de tal forma que este evalúe todos los *checkpoint* que se han generado durante el proceso de entrenamiento, en lugar de evaluar solamente el último.
- En lo que respecta a la cantidad de detectores que pueden ser ofrecidos por *Dettool* para ser entrenados, esta es directamente dependiente de los que *TensorFlow 2 Detection Model Zoo* tenga en su lista, lo cual es una de las principales razones por las que se ha elegido la *TensorFlow Object Detection API* [67], ya que esta proporciona soporte para trabajar con diversos modelos de detección de objetos basados en el uso de *anchor boxes*, que son aquellos en los que este trabajo se centra principalmente. Sin embargo, de entre estos modelos, no se encontraba disponible ninguna versión de *YOLO*, por lo que este detector solo se encuentra definido en el marco teórico de este trabajo, y no se ofrece ningún resultado práctico en lo que a él respecta, dentro del marco técnico. Por otra parte, aquellos modelos basados en el uso de *anchor boxes* que sí se encuentran en el ámbito técnico y teórico de este trabajo han sido: *SSD MobileNet v1*, *SSD MobileNet v2*, *Mask R-CNN Inception ResNet v2*, y *Faster R-CNN ResNet 50 v1*.
- En lo que respecta a la obtención de métricas con las que analizar el ajuste de los detectores de objetos tras los procesos de entrenamiento y validación cabe señalar:
    - Por un lado, se dispone de los informes automáticos que genera el *script model_main_tf2.py* durante el proceso de validación, lo que es de utilidad tanto para saber cuánto resta para la finalización del entrenamiento, ya que este *script* solo genera un informe cada vez que evalúa un estado (*checkpoint*) de un detector, como para ver cómo varían las métricas de *mean Average Recall* y *meam Average Precision*, a medida que transcurre el proceso de entrenamiento.

- Por otro lado, se dispone de las métricas que *TensorBoard* [79] genera cuando finalizan los procesos de entrenamiento y validación. Dichas métricas son, la cantidad de *steps* procesados por unidad de tiempo, la variación del factor de aprendizaje a lo largo del proceso de entrenamiento, los valores de la función de pérdida de la clasificación, localización, regularización, y la pérdida total, que es la suma de las tres anteriores. Adicionalmente, se añaden varias gráficas de *mean Average Precicsion* en las que se mide el valor de esta métrica con distintos valores umbrales de IoU (0.5, 0.75, y 0.5:.5:0.95), y también con distintos tamaños de objetos (*small*: *area* < 32^2; *medium*: 32^2 < *area* < 96^2; *large*: *area* > 96^2). Por último, se añaden varias gráficas de *mean Average Recall* en las que se mide el valor de esta métrica con distintos números de detecciones por imagen (1, 10, y 100), y con distintos tamaños de objetos (*small*: *area* < 32^2; *medium*: 32^2 < *area* < 96^2; *large*: *area* > 96^2).

- Por último, se ha experimentado con el uso de este programa realizando los entrenamientos y las validaciones de los siguientes detectores de la *TensorFlow Object Detection API* [67], SSD MobileNet v1, SSD MobileNet v2, *Faster R-CNN ResNet 50 v1*, y *Mask R-CNN Inception ResNet v2*. Los resultados obtenidos han sido:

  1) Faster R-CNN ResNet 50 v1:

  - La figura 7.6 muestra la configuración del entrenamiento y la validación de este modelo.
  - Las figuras 7.7 – 7.13 muestran las distintas métricas de *TensorBoard* para indicar los resultados del entrenamiento y la validación de este detector. Como se puede comprobar en la figura 7.6, debería haber un punto de validación del estado del detector (*checkpoint*) cada 1000 *steps*, sin embargo, todas las gráficas de 7.7 a 7.13 muestran que se produce algún tipo de error, ya que los resultados de validación no se muestran más allá de los 1000 primeros *steps*. Este error podría darse al escribir o acceder al fichero de *logs* de la carpeta *eval* en el *bucket*, ya que este debería contener los datos que posteriormente se muestran gráficamente en los paneles de *TensorBoard* [79].

- Los resultados generados por *TensorBoard* [79] no pueden juzgarse por el error que ha tenido lugar, sin embargo, esto no evita que a través del informe final de las métricas *mean Average Recall* y *mean Average Precision*, que genera el *script model_main_tf2.py* automáticamente mediante la salida de la celda del programa en el *Notebook* cada vez que se alcanza un punto de validación cada cierto número de *steps*, pueda realizarse un análisis del entrenamiento y la validación. Este informe de *mAP* y *mAR* se puede ver en la figura 7.14, donde:

  - La eficacia de las detecciones del modelo, en relación con los valores umbrales del índice IoU, ha sido mayor cuando se han empleado valores umbrales menores. Puede observarse que con un valor umbral de 0.5 en el índice IoU, casi se logra un 0.5 de *mAP*, mientras que con umbrales de IoU superiores a 0.5, la eficacia de las detecciones ha sido aproximadamente de 0.3 *mAP*.

  - La eficacia de las detecciones del modelo, en relación con el tamaño de los objetos, es mayor cuando más grande es el objeto, pues se obtienen valores de 0.04, 0.26, y 0.51, de *mAP*, para objetos con un tamaño pequeño, mediano, y grande.

  - La capacidad del modelo para detectar, en relación al número de detecciones por imagen, es considerable cuando se trata de objetos de tamaño grande y mediano, ya que, como puede verse, estos están muy cerca de ofrecer un 0.5 de *mAR*. Por otra parte, los objetos de tamaño pequeño son algo más complicados de poder ser detectados, pues su valor de *mAR* está en 0.32.

  - La capacidad del modelo para detectar, en relación con el tamaño de los objetos, es buena para objetos de gran tamaño (0.72 *mAP*), es aceptable para objetos de tamaño mediano (0.5 *mAP*), y es muy mala a la hora de tratar de detectar objetos de tamaño pequeño (0.14 *mAP*).

- Un aspecto a destacar ha sido la necesidad de bajar el *mini-batch-size* de la TPU [65] de 32 a 24, ya que este debe ser múltiplo de 8, y con 32 tenía lugar un error debido a que se intentaba reservar más memoria de la disponible para cargar las imágenes.

- En lo que respecta a las opciones de *data augmentation*, se ha optado por generar imágenes adicionales mediante la aplicación de 4 tipos de transformaciones: volteo vertical de la imagen, volteo horizontal de la imagen, rotación máxima de 90 grados, y recorte de la imagen. Pues se ha considerado que los aparatos electrónicos pueden aparecer de muchas formas diferentes en las imágenes.

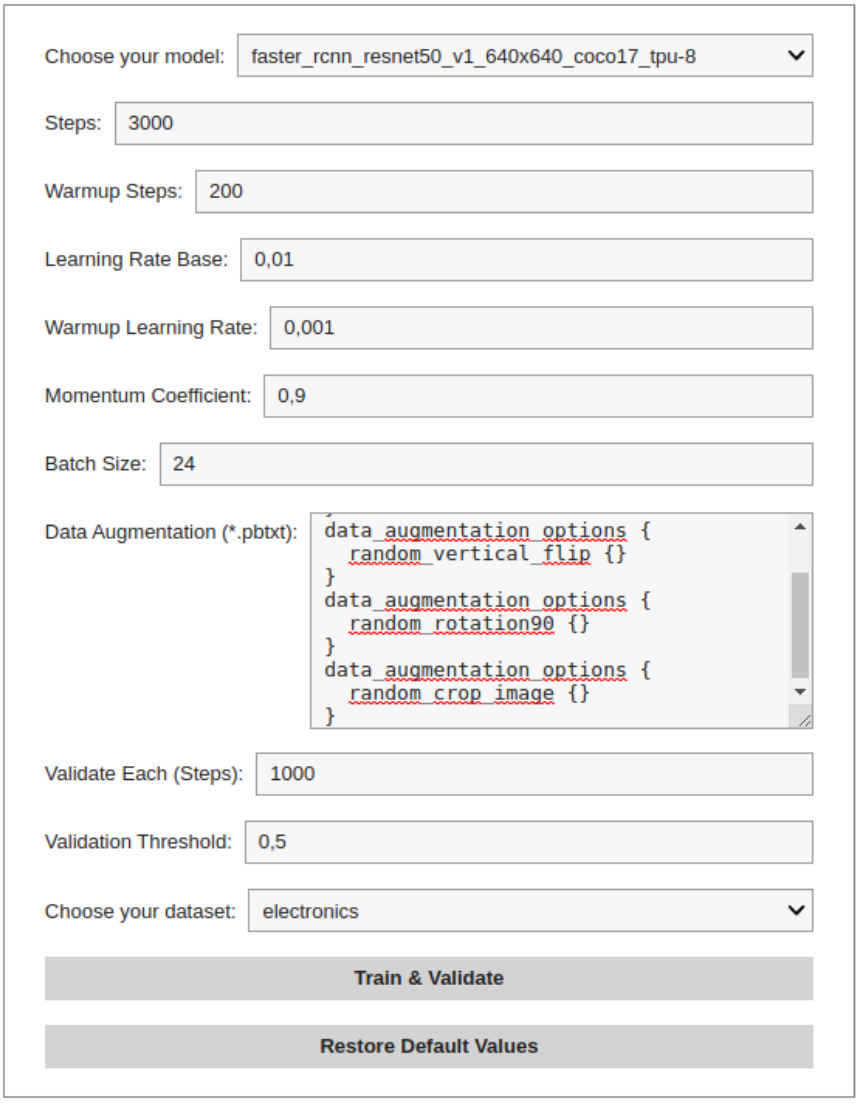

Figura 7.6: Opciones del entrenamiento y la validación de Faster R-CNN

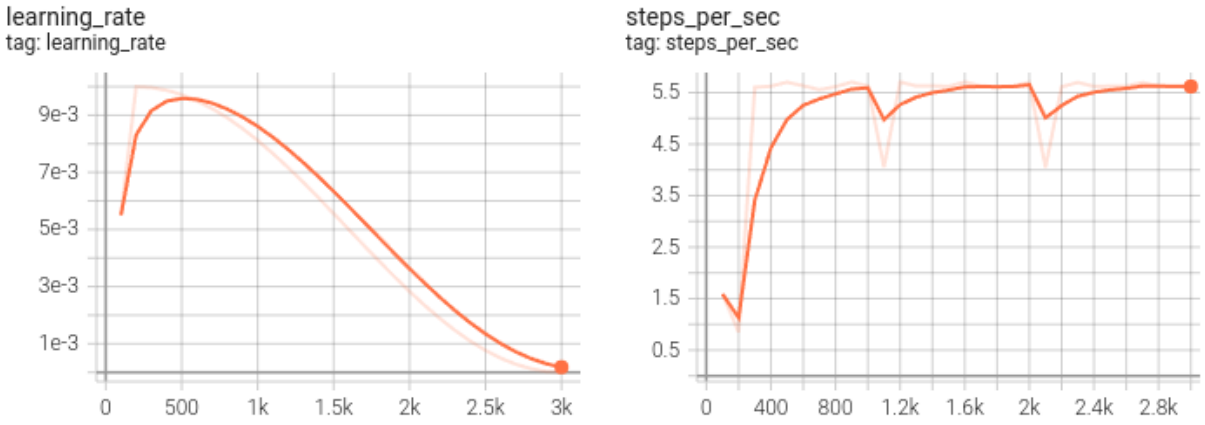

Figura 7.7: Evolución del *learning rate* y los *steps per sec*

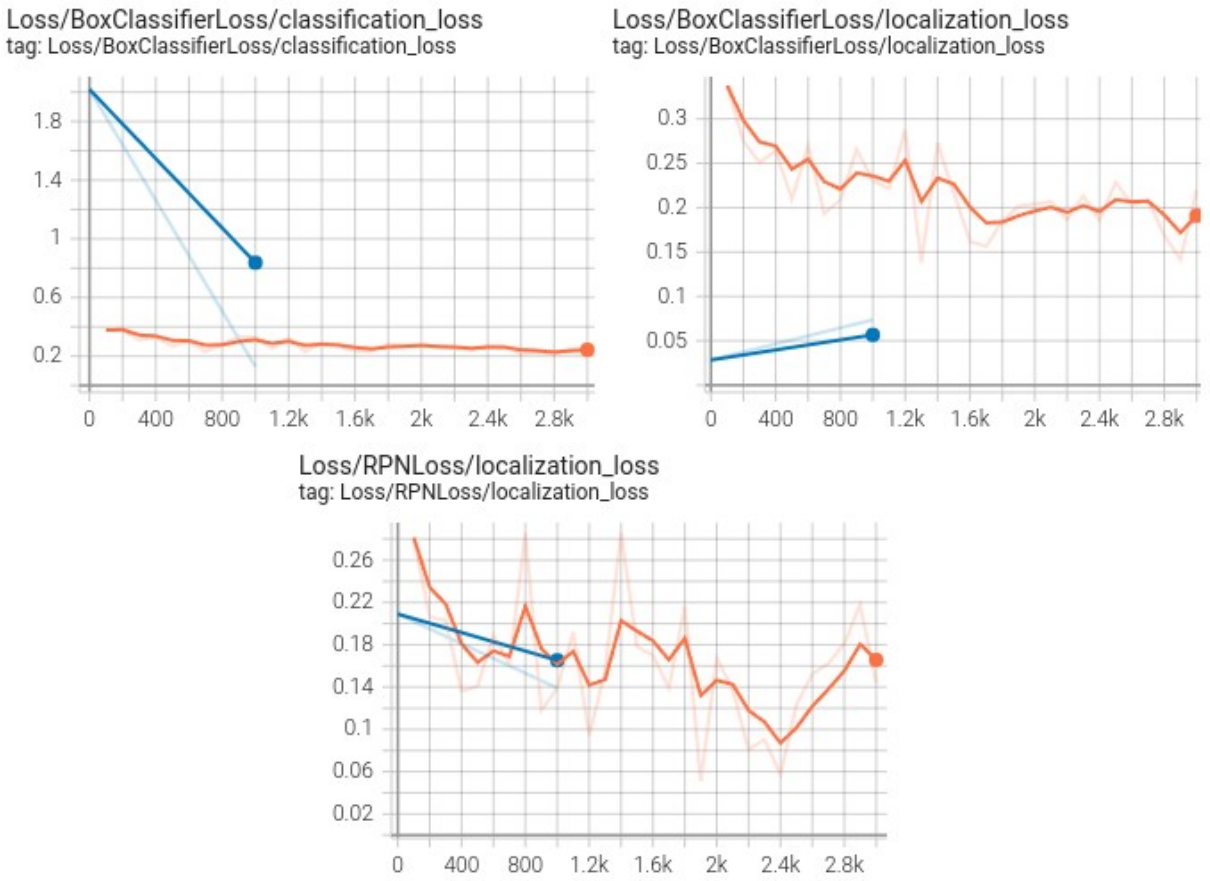

Figura 7.8: Funciones de pérdida de la clasificación, localización, y la RPN

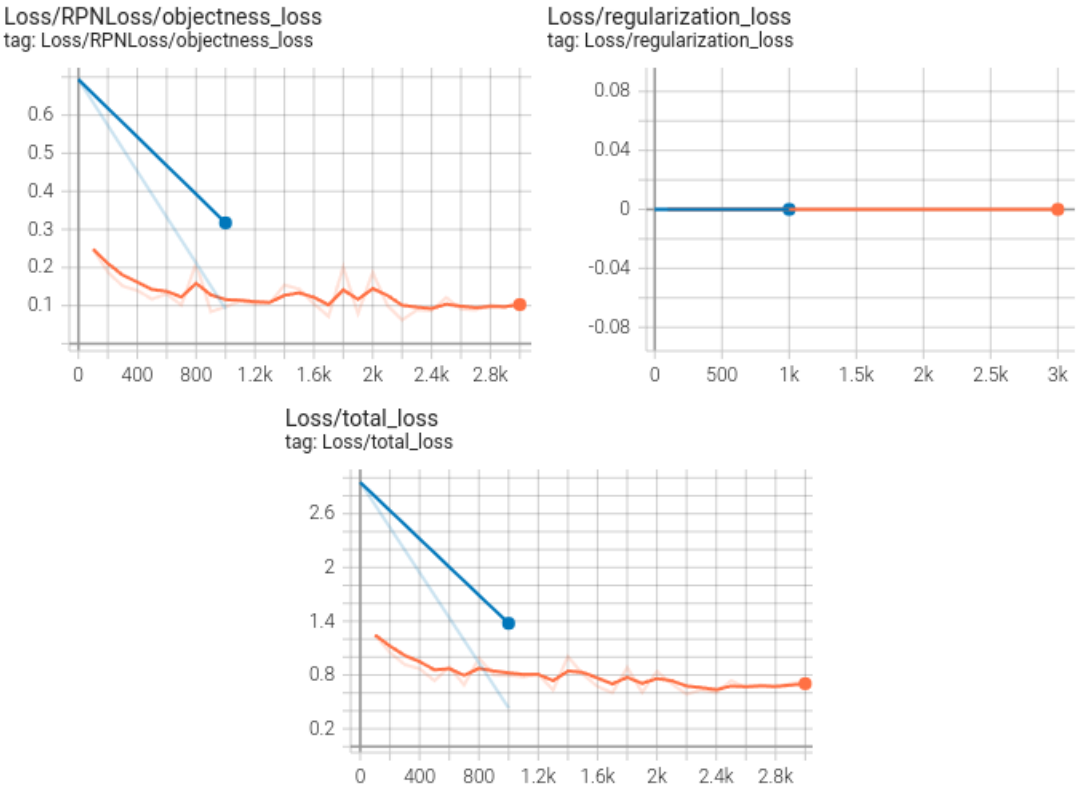

Figura 7.9: Pérdida de la proposición de regiones, la regularización, y total

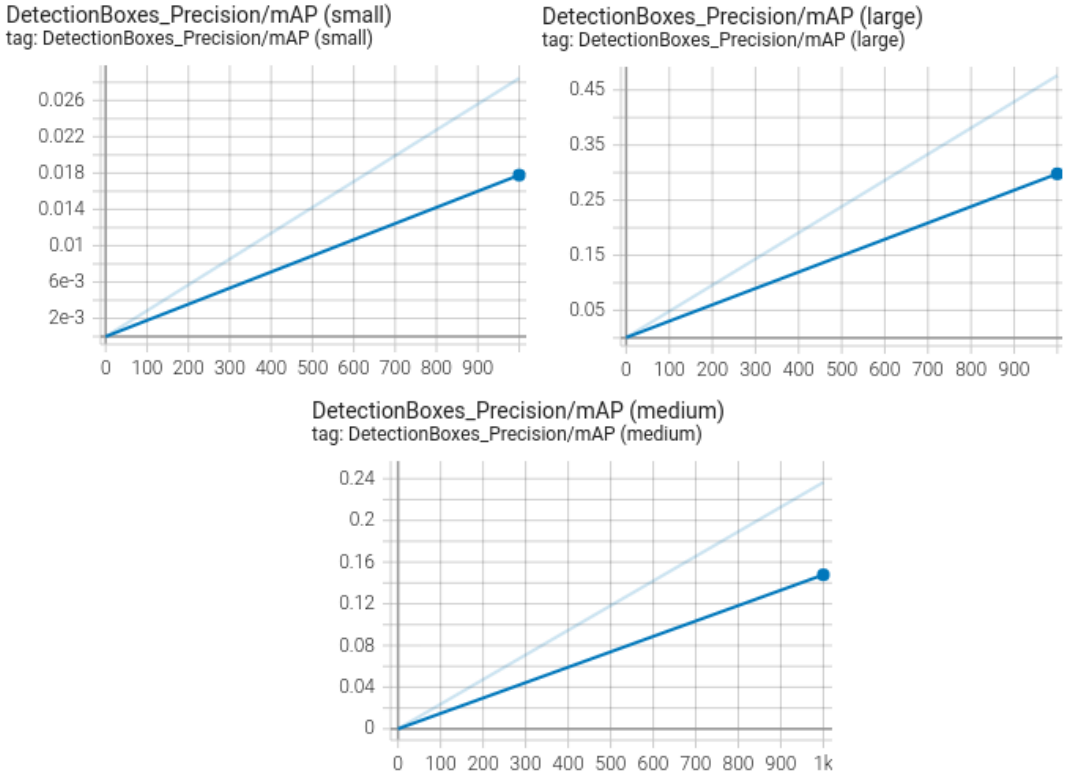

Figura 7.10: Evolución de *mAP* en base al tamaño de los objetos

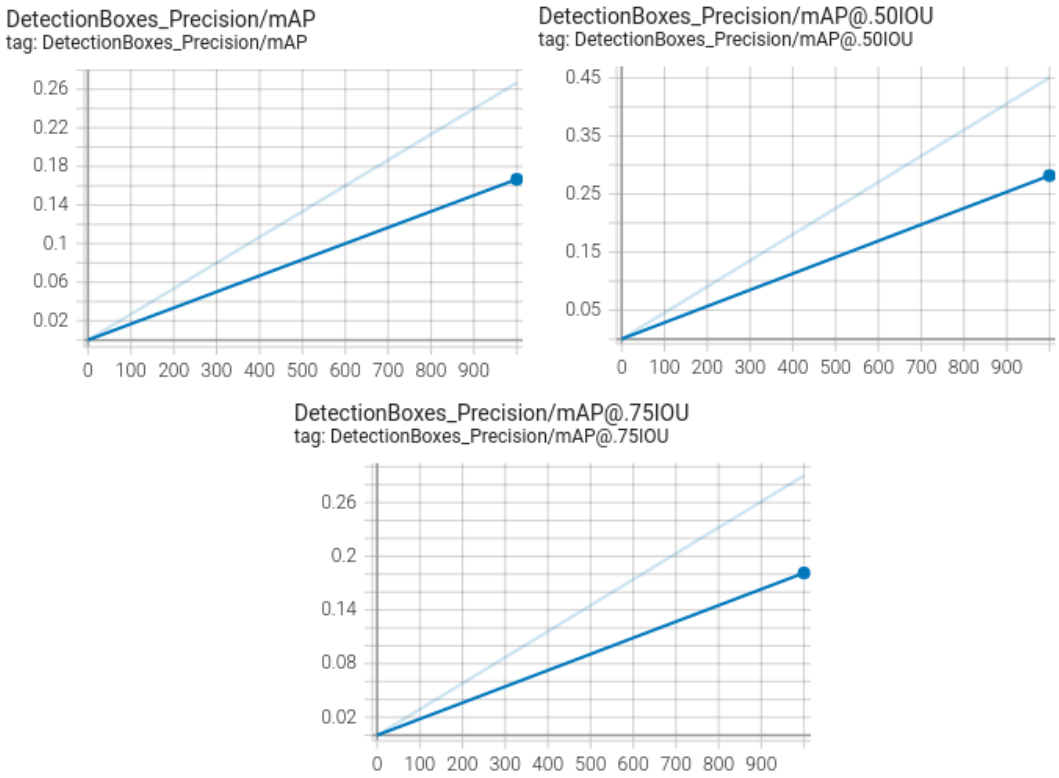

Figura 7.11: Evolución de *mAP* en base a los valores umbrales IoU

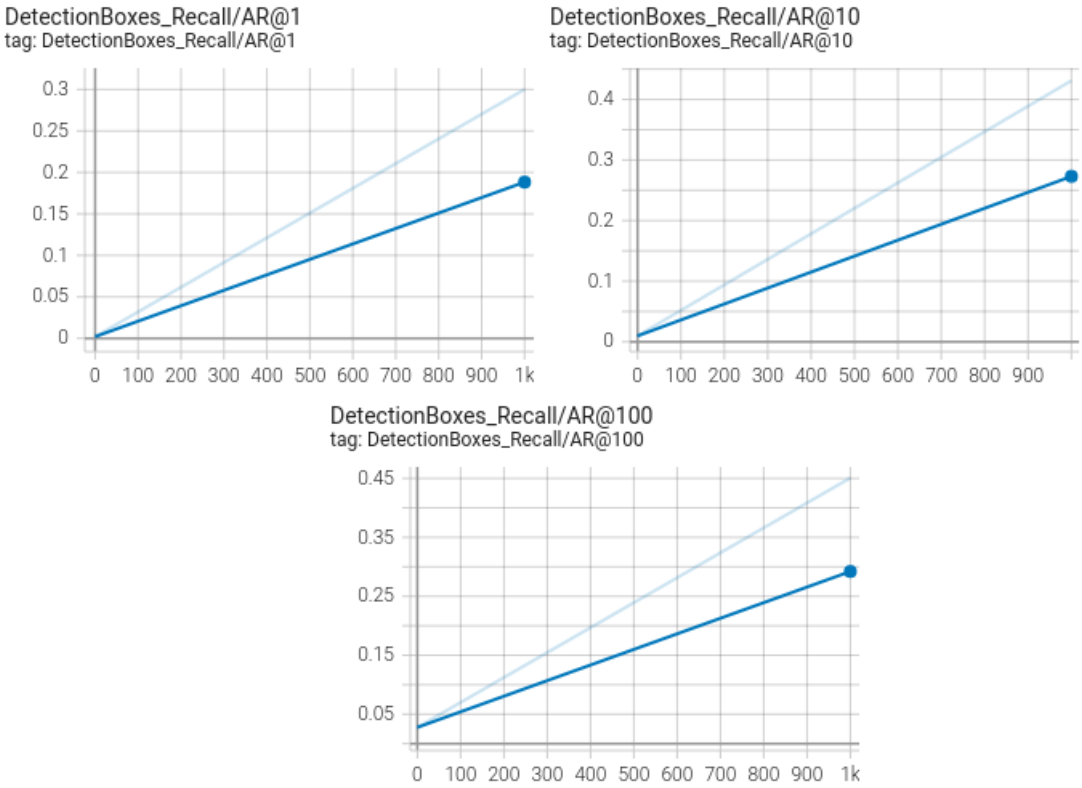

Figura 7.12: Evolución de *mAR* en base a la cantidad de detecciones

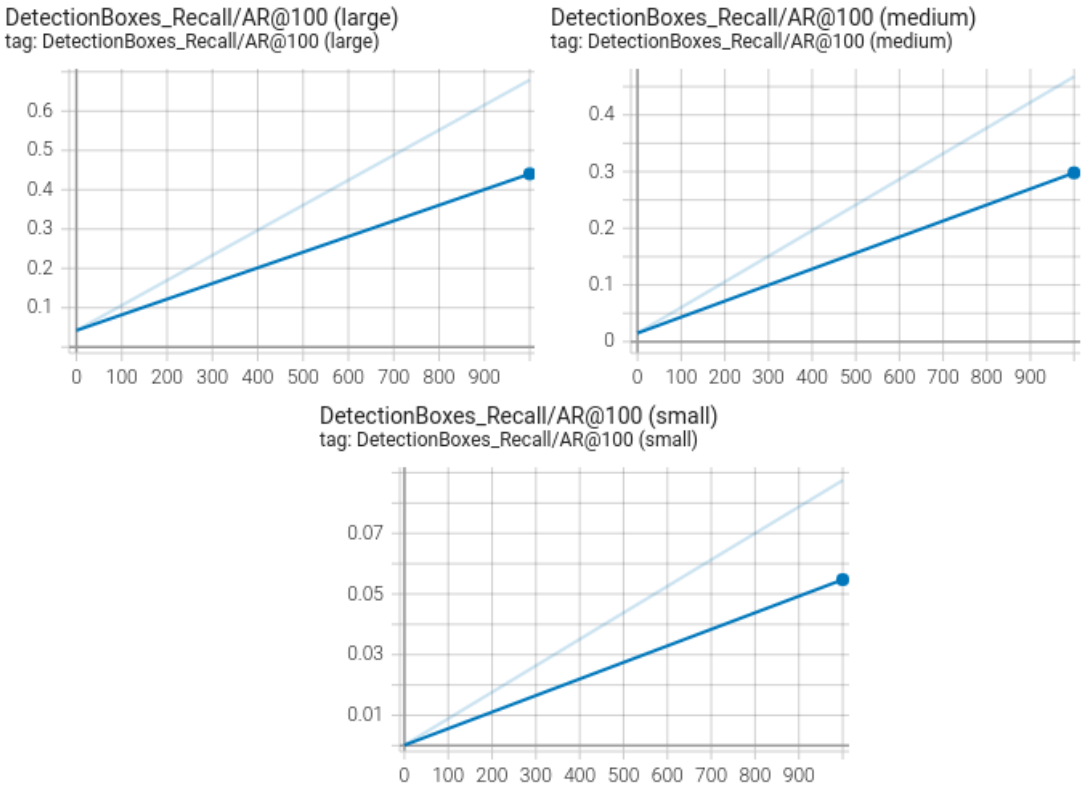

Figura 7.13: Evolución de *mAR* en base al tamaño de los objetos

```
Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.291
Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.472
Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.317
Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.044
Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.265
Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.513
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.320
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.469
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.495
Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.141
Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.504
Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.722
```

Figura 7.14: Informe final de *mean Average Precision* y *mean Average Recall*

2) SSD MobilneNet v1:

- La figura 7.15 muestra cuál fue la configuración del entrenamiento y la validación de este detector. Las figuras 7.16 – 7.21 muestran las métricas de *TensorBoard* [79], que reflejan los resultados de los procesos de entrenamiento y validación. La figura 7.22 muestra un informe con el resultado final de la evolución de las métricas *mean Average Recall* y *mean Average Precision*.

- Se ha optado por generar imágenes adicionales para el entrenamiento mediante el método de *data augmentation*, con opciones que varían el color, brillo, contraste, la saturación, y los matices del color. Se han escogido estas opciones ya que, al tratarse de imágenes de vehículos en espacios abiertos, puede jugarse con las propiedades visuales para enriquecer los aspectos con los que estos pueden aparecer dentro de las imágenes en entornos reales.

- En lo que respecta a la figura 7.17, donde pueden verse las funciones de pérdida del modelo, se observa que el error cometido tanto por la clasificación como por la localización están cercanas a cero, 0.49 y 0.375 aproximadamente. Esto sugeriría que ambas capacidades del detector deberían funcionar, al menos decentemente. Posteriormente, está la pérdida de la regularización, que es de tipo $\boldsymbol{L_1}$, y que alcanza un bajo valor (0.75), lo que sería una señal de que los pesos del modelo están consiguiendo una adecuada generalización de este, evitando el sobreajuste. Se ha de tener en cuenta que el coeficiente de regularización penaliza el efecto de los pesos durante el proceso de minimización, lo que significa que, si dicho factor disminuye, la penalización es menor y, por tanto, la generalización es mejor.

- En la figura 7.18 se puede ver cuál ha sido la evolución de la métrica *mean Average Precision* en base a los valores umbrales del índice IoU. Se observa que la precisión de las detecciones es mayor cuando se aplica un valor umbral de IoU más bajo (0.5), alcanzando en ese caso un 0.55 de *mAP*, valor que se reduce hasta 0.30 a medida que este se eleva hasta un umbral de 0.75, o más.

- En la figura 7.19 se ve cómo ha sido la evolución de la métrica *mean Average Precision* en base al tamaño de los objetos. En ella se puede ver que ni siquiera los objetos grandes tienen una detección precisa adecuada, ya que esta ronda el 0.4 de *mAP*. Por otra parte, los objetos medianos son todavía más difíciles de detectar, ya que cuentan con un *mAP* del 0.20. Por último, parece que los objetos pequeños son casi imposibles de detectar, pues su *mAP* es prácticamente nula.

- La figura 7.20 refleja la evolución de *mean Average Recall* en base a la cantidad de detecciones por imagen a lo largo del proceso de entrenamiento. Puede verse fácilmente que con aproximadamente 10 detecciones por imagen ya se puede lograr una mediana capacidad de detección de objetos (0.45 *mAR*), lo cual tiende a ser positivo, sin embargo, cuando el número de detecciones por imagen aumenta a 100, la capacidad de detección apenas sube (0.55 *mAR*), y aunque ya es algo positivo, no resulta un progreso muy significativo. En último lugar, puede verse que capacidad de detección es mediocre para objetos de tamaño pequeño (0.3 de *mAR*).

- La figura 7.21 muestra la evolución de la métrica *mean Average Recall* a lo largo del entrenamiento, en base al tamaño de los objetos; por otro lado, la figura 7.22 muestra el resultado final de esta métrica, en base al anterior criterio, una vez finalizado el proceso de validación del detector. En esta última figura, puede comprobarse que la capacidad del modelo para detectar los objetos aumenta a medida que sean más grandes, pues el valor de *mAR* es de 0.06, 0.34, y 0.69, para los objetos de tamaño pequeño, mediano, y grande.

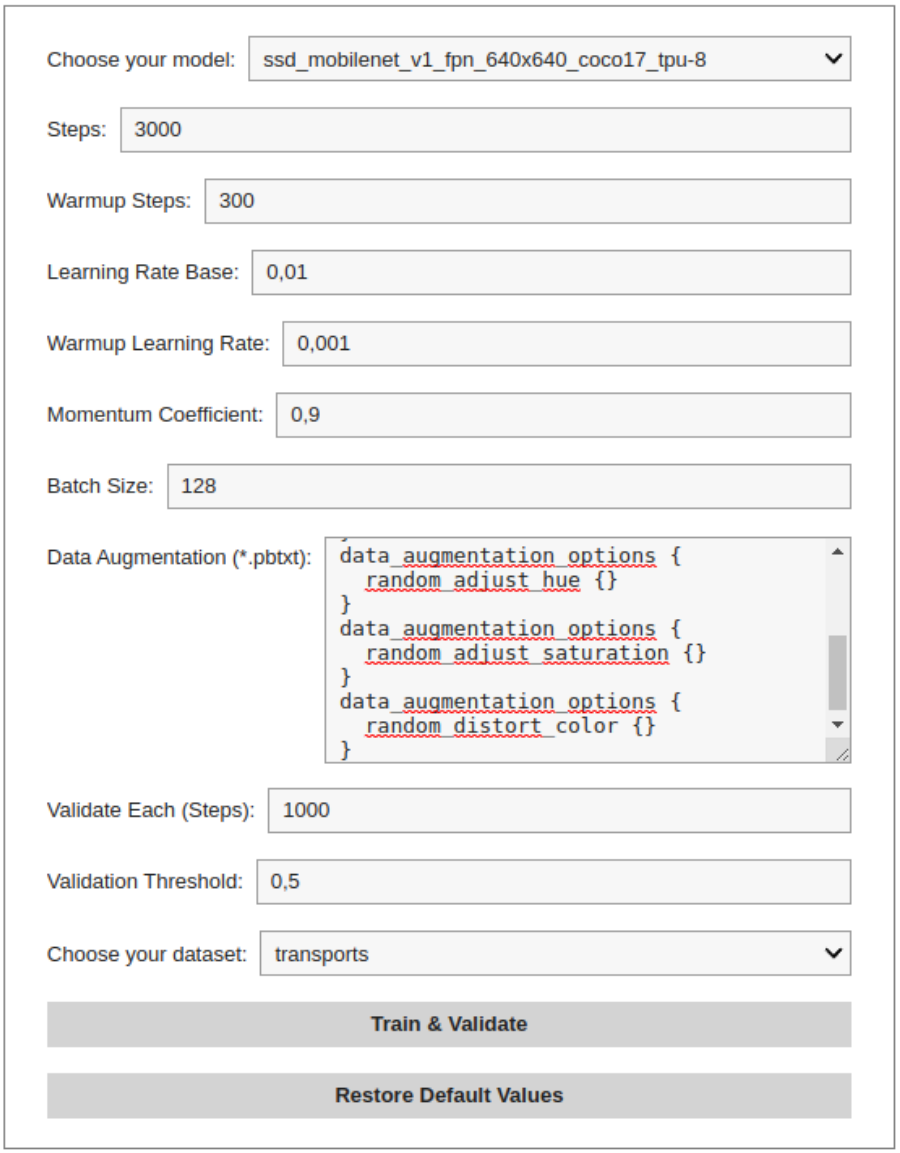

Figura 7.15: Opciones del entrenamiento y la validación de SSD MobileNet v1

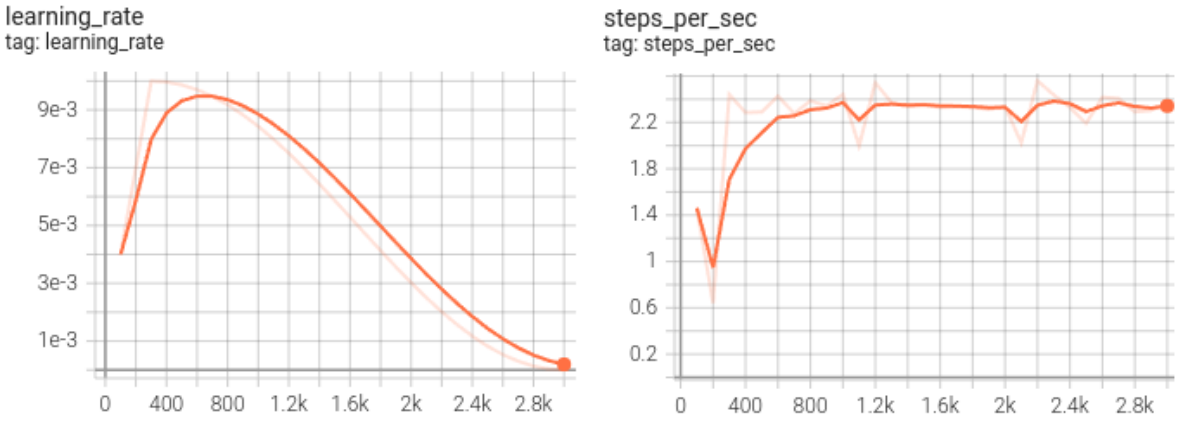

Figura 7.16: Evolución del *learning rate* y los *steps per sec*

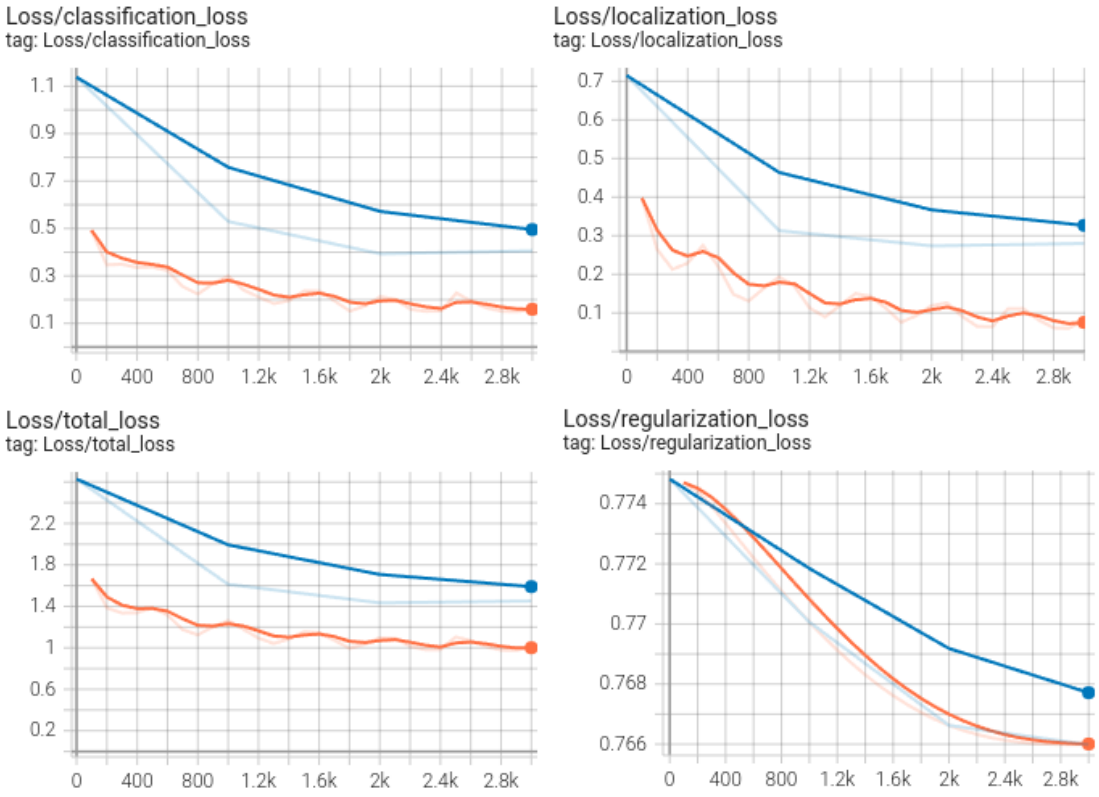

Figura 7.17: Pérdida de la clasificación, localización, regularización, y total

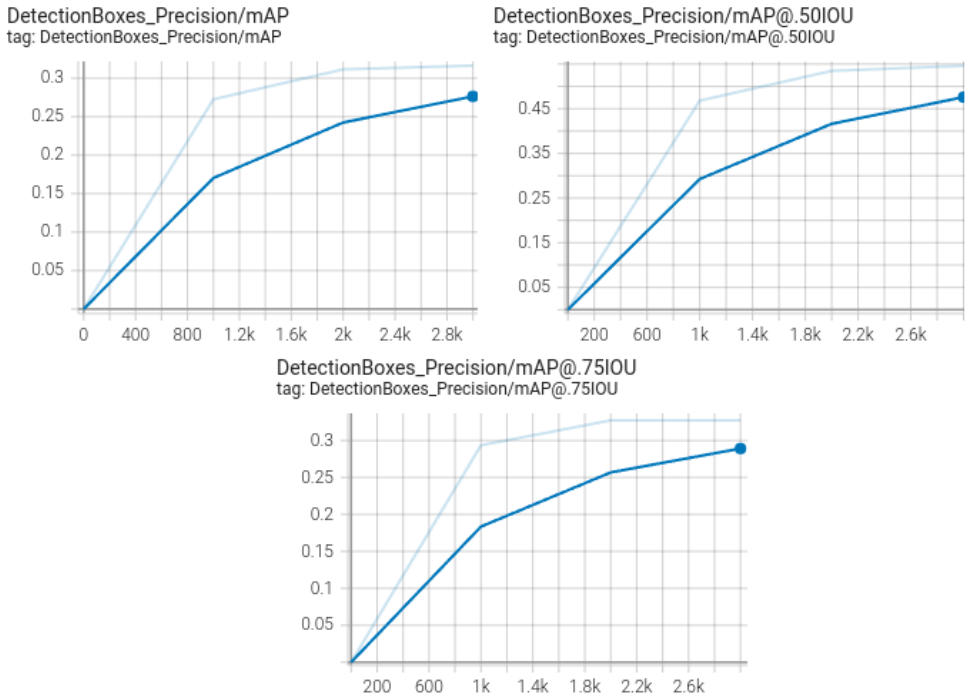

Figura 7.18: Evolución de *mAP* en base a los valores umbrales IoU

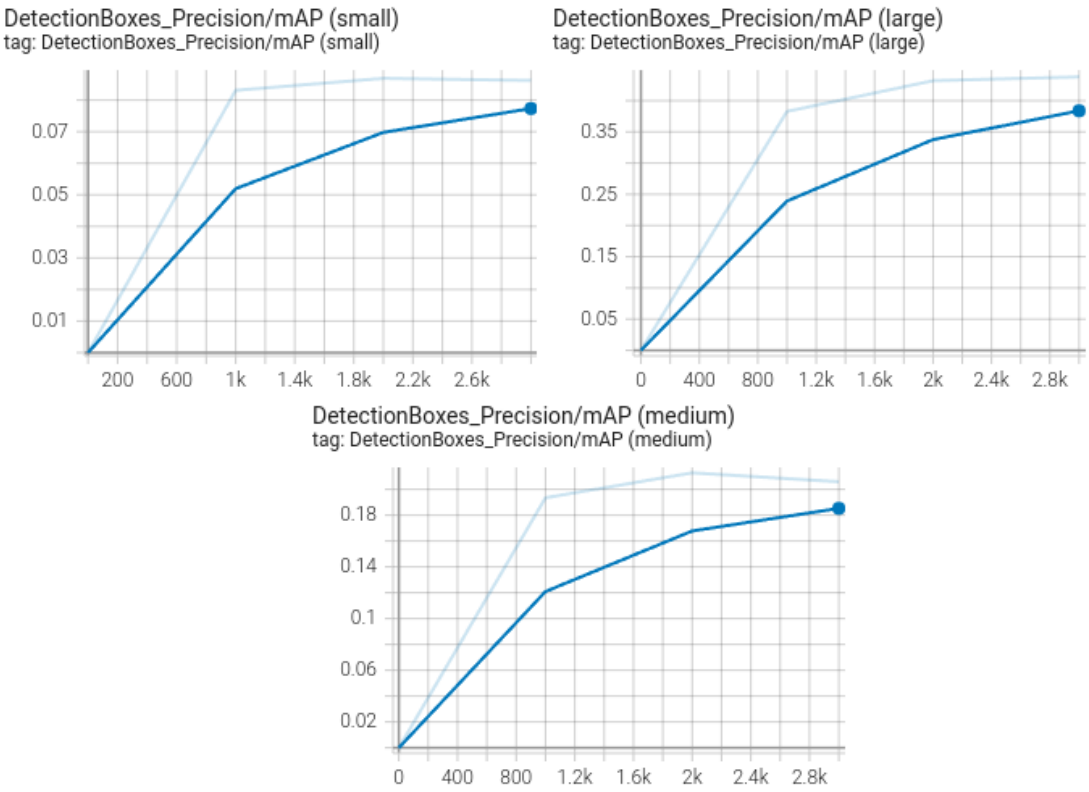

Figura 7.19: Evolución de *mAP* en base al tamaño de los objetos

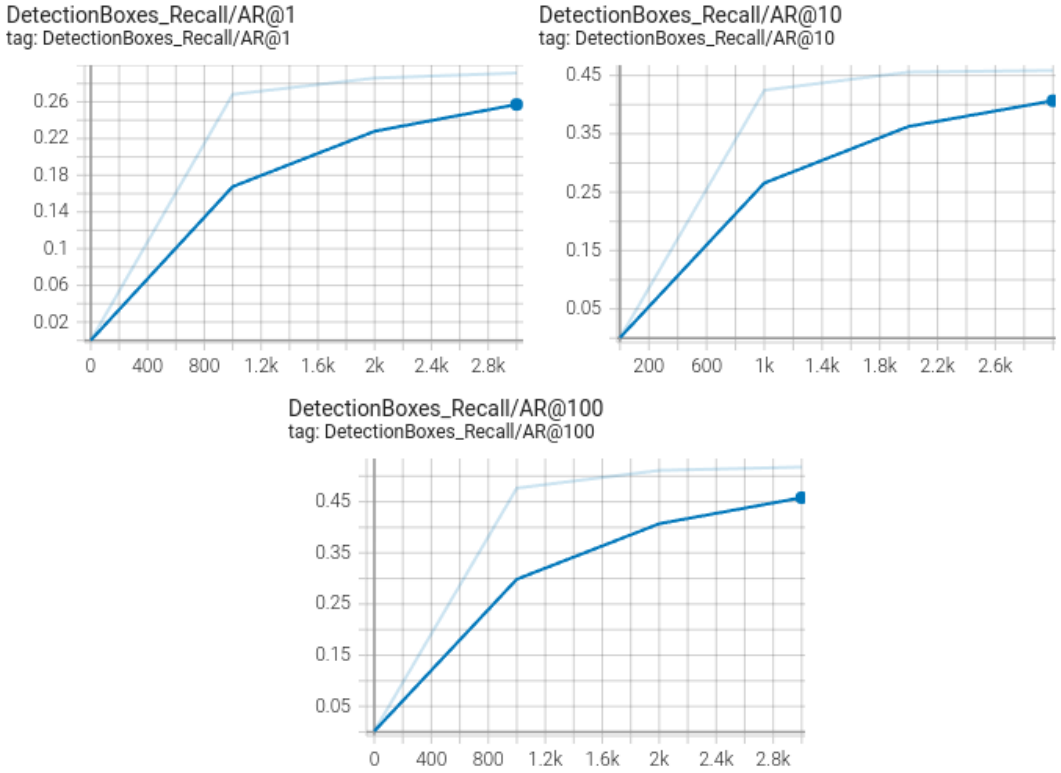

Figura 7.20: Evolución de *mAR* en base a la cantidad de detecciones

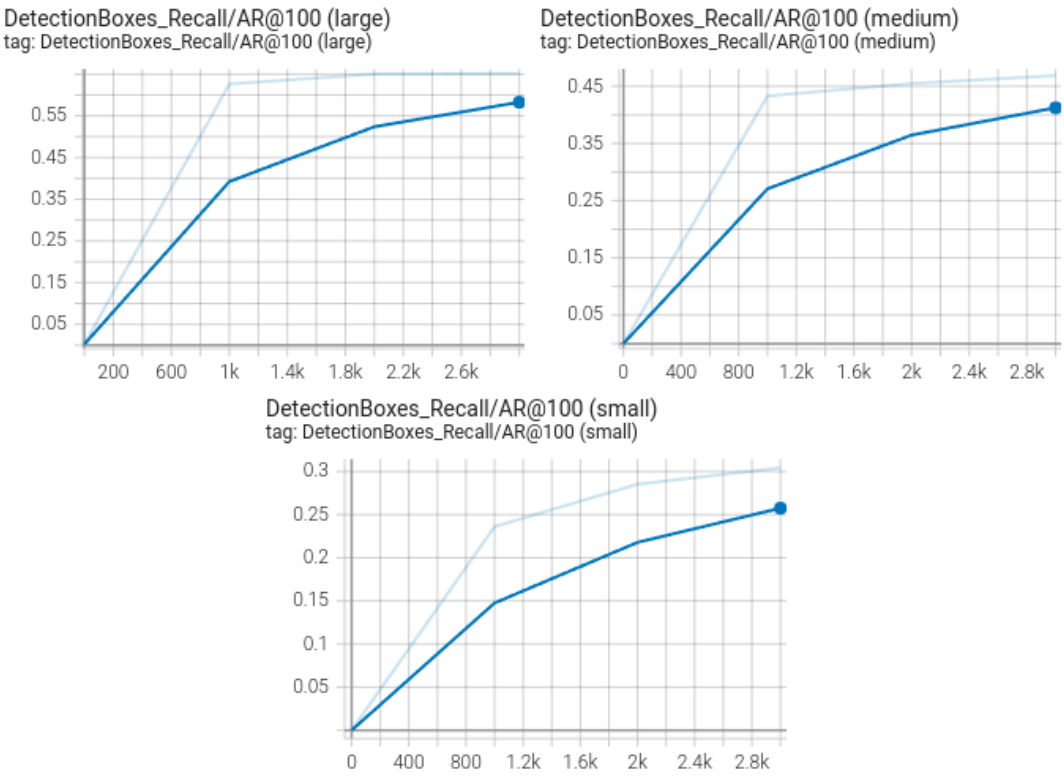

Figura 7.21: Evolución de *mAR* en base al tamaño de los objetos

```
Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.316
Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.546
Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.327
Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.086
Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.206
Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.438
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.291
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.458
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.518
Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.304
Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.469
Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.652
```

Figura 7.22: Informe final de *mean Average Precision* y *mean Average Recall*

3) SSD MobilneNet v2

- La figura 7.23 revela la configuración del entrenamiento y la validación de este modelo. Las figuras 7.24 – 7.29 muestran aquellas métricas de *TensorBoard* [79] que indican cuáles han sido los resultados de su proceso de entrenamiento y validación. La figura 7.30 muestra un informe con el resultado final de la evolución de las métricas *mean Average Recall* y *mean Average Precision*.

- Se han generado imágenes adicionales para el entrenamiento del detector mediante el método de *data augmentation*. Las opciones que se han elegido permiten variar la rotación de una imagen, o bien realizar un recorte de esta. Se han escogido estas opciones debido a que las personas que pueden estar presentes dentro de las imágenes, pueden aparecer de muchas maneras, por lo que será conveniente generar todavía más imágenes que puedan ayudar a generalizar cuando aparece la figura de una persona, ya sea total, o parcialmente.

- En la figura 7.25 pueden verse las funciones de pérdida del modelo. Al igual que ocurrió con el modelo *SSD MobileNet v1*, el error cometido en la clasificación (0.8), y en la localización (0.49), son inferiores a 1, por lo que cabría esperarse un funcionamiento decente del detector. Sin embargo, en el caso de la regularización mediante $\boldsymbol{L_1}$, el valor final de esta es muy bajo, prácticamente nulo (0.08), por lo que podría decirse que el modelo generaliza suficientemente bien.

- En la figura 7.26 puede observarse la evolución de la métrica *mean Average Precision* atendiendo a los valores umbrales del índice IoU. Se observa que la precisión de las detecciones es mayor cuando se aplica un valor umbral de IoU más bajo (0.5), alcanzándose en ese caso 0.52 de *mAP*. Por otra parte, cuando se incrementa el valor umbral del índice IoU, la precisión de las detecciones disminuye, quedando en los casos restantes, cerca de 0.3 *mAP*.

- En la figura 7.27 puede apreciarse cómo ha evolucionado la métrica *mean Average Precision* en base al tamaño de los objetos. Aquellos que poseen un tamaño más grande son los que parecen que se detectan con mayor precisión, pues rondan el 0.57 de *mAP*. Por otro lado, se observa una disminución de la precisión de las detecciones en lo que respecta a los objetos medianos o pequeños, ya que, en el caso de los primeros, su *mAP* es cercano a 0.20, y en de los segundos, esta métrica es prácticamente nula.

- A través de la figura 7.28 puede verse la evolución de la métrica *mean Average Recall* en base a la cantidad de detecciones por imagen. Según lo indicado por la métrica, parece que aplicar hasta 10 y 100 detecciones por imagen, solo proporciona aproximadamente un 0.35 de *mAR*, lo que implica una capacidad poco aceptable para poder detectar objetos en general. En cuanto a la aplicación de una detección por imagen, esta da como resultado un 0.15 de *mAR*, lo que indica que es muy difícil realizar detecciones de objetos en este caso.

- Finalmente, en la figura 7.29 se puede ver la evolución de la métrica *mean Average Recall* en base al tamaño de los objetos. La capacidad del modelo para detectar los objetos se ve decrementada a medida que el tamaño de estos se hace más pequeño, concretamente, se obtienen los valores de *mAR*, 0.69, 0.34, y 0.06, para los objetos de tamaño grande, mediano, y pequeño.

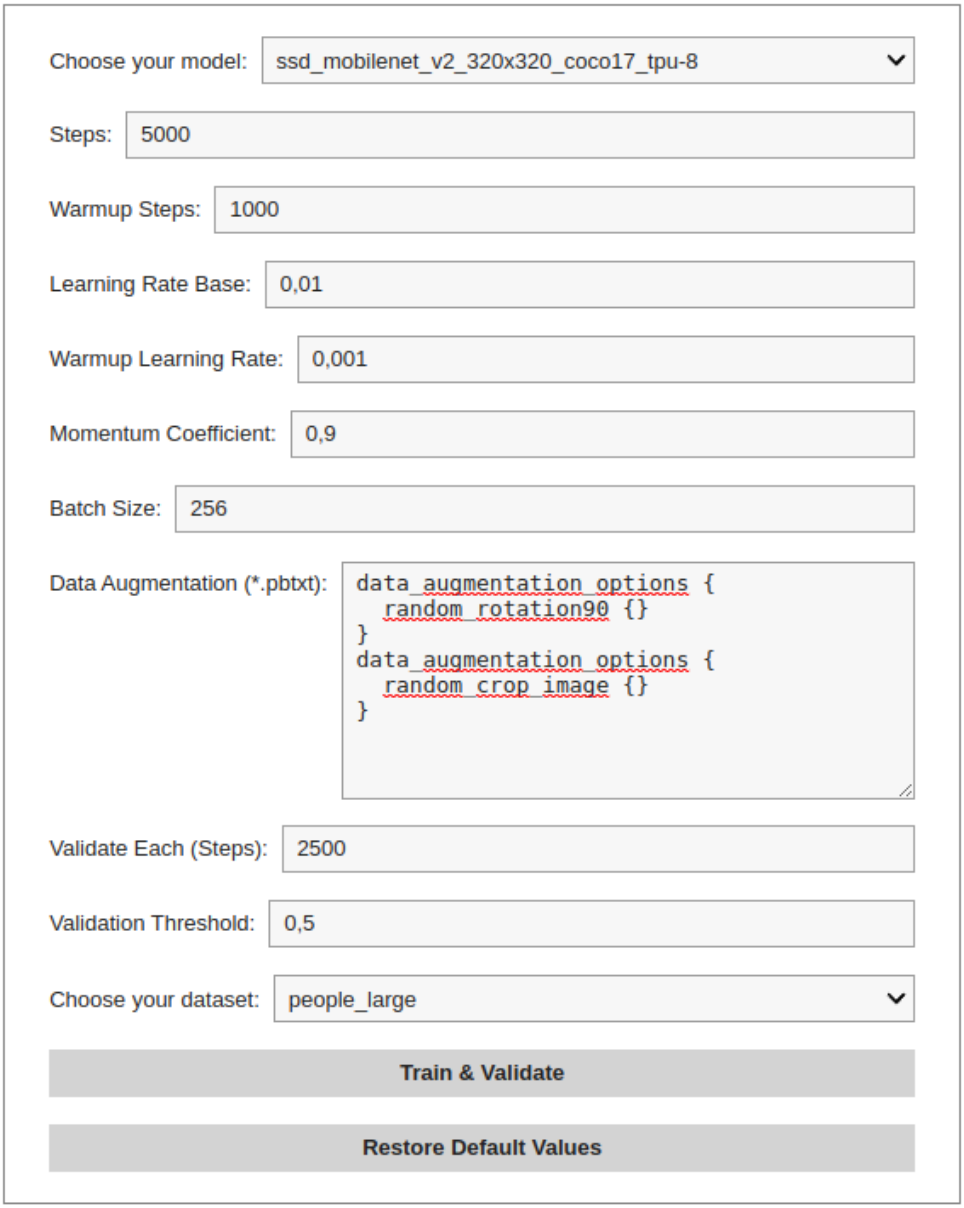

Figura 7.23: Opciones del entrenamiento y la validación de SSD MobileNet v2

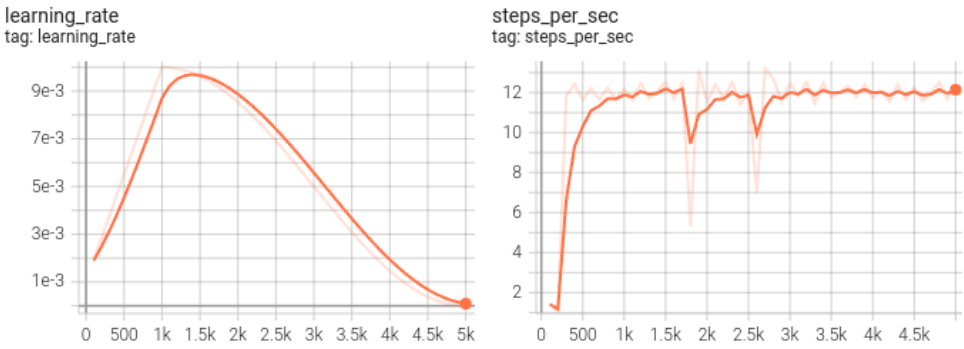

Figura 7.24: Evolución del *learning rate* y los *steps per sec*

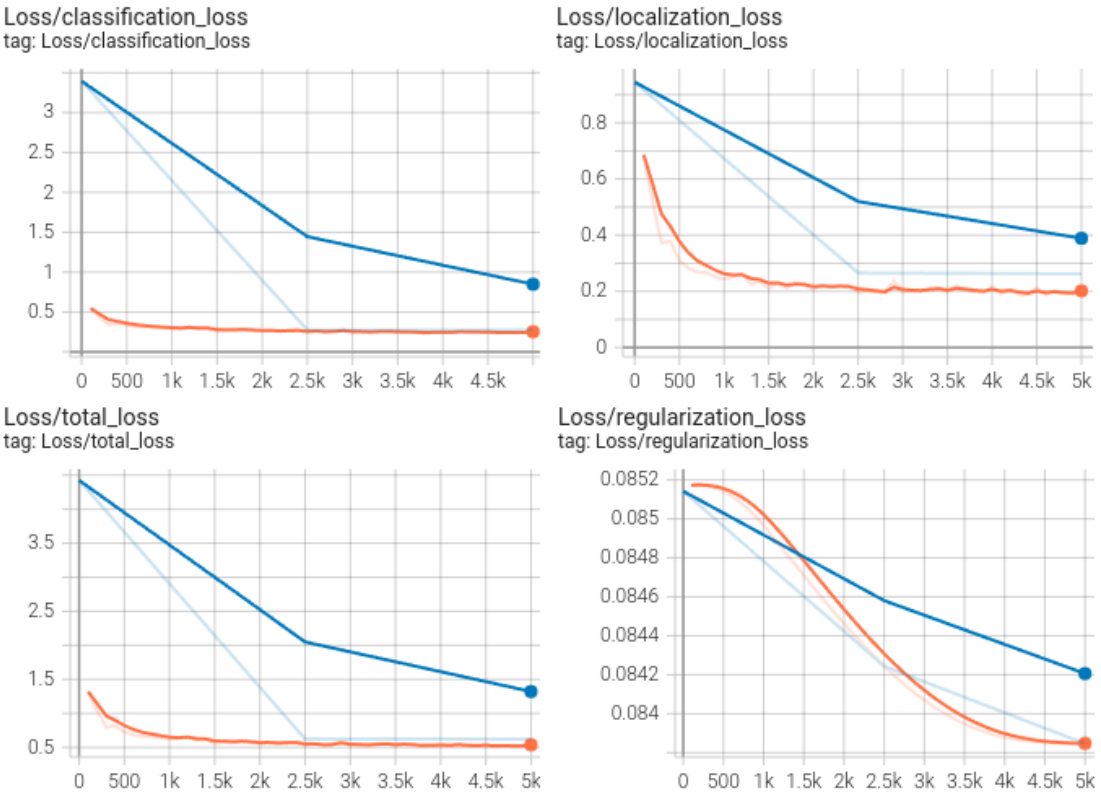

Figura 7.25: Pérdida de la clasificación, localización, regularización, y total

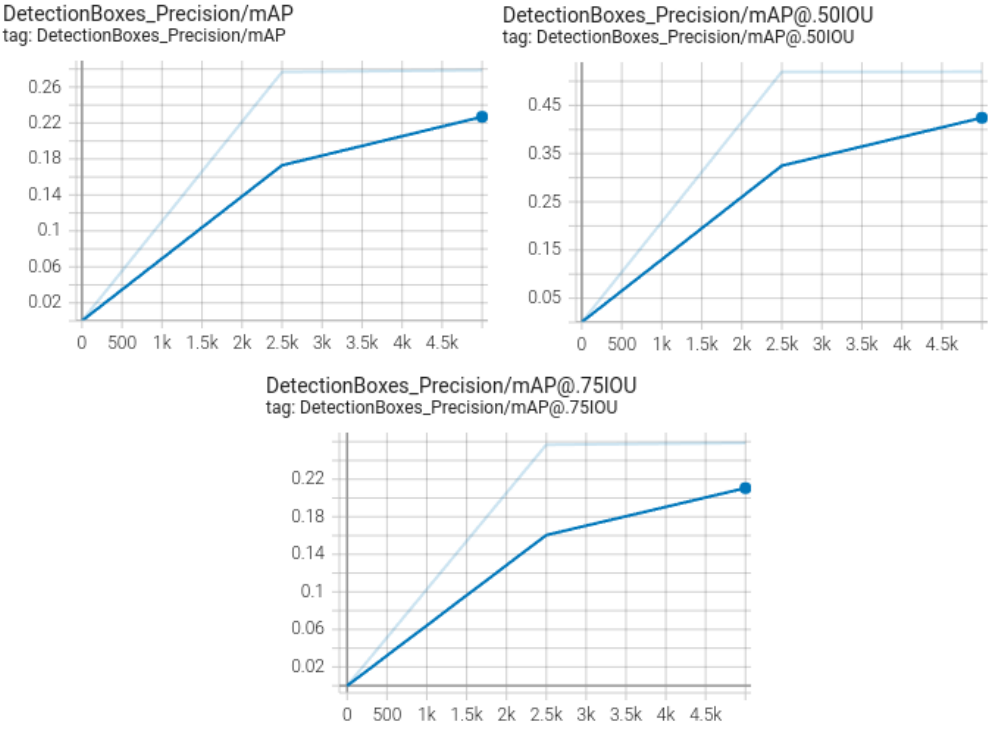

Figura 7.26: Evolución de *mAP* en base a los valores umbrales IoU

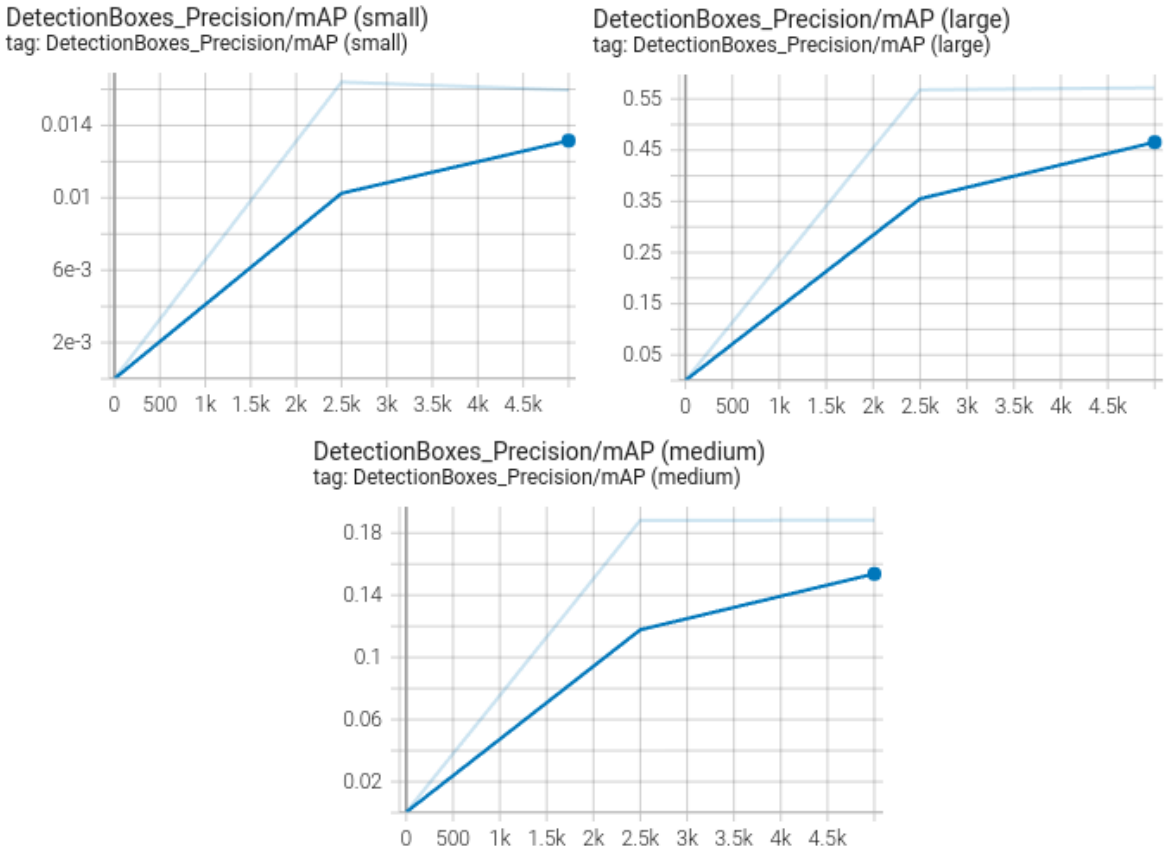

Figura 7.27: Evolución de *mAP* en base al tamaño de los objetos

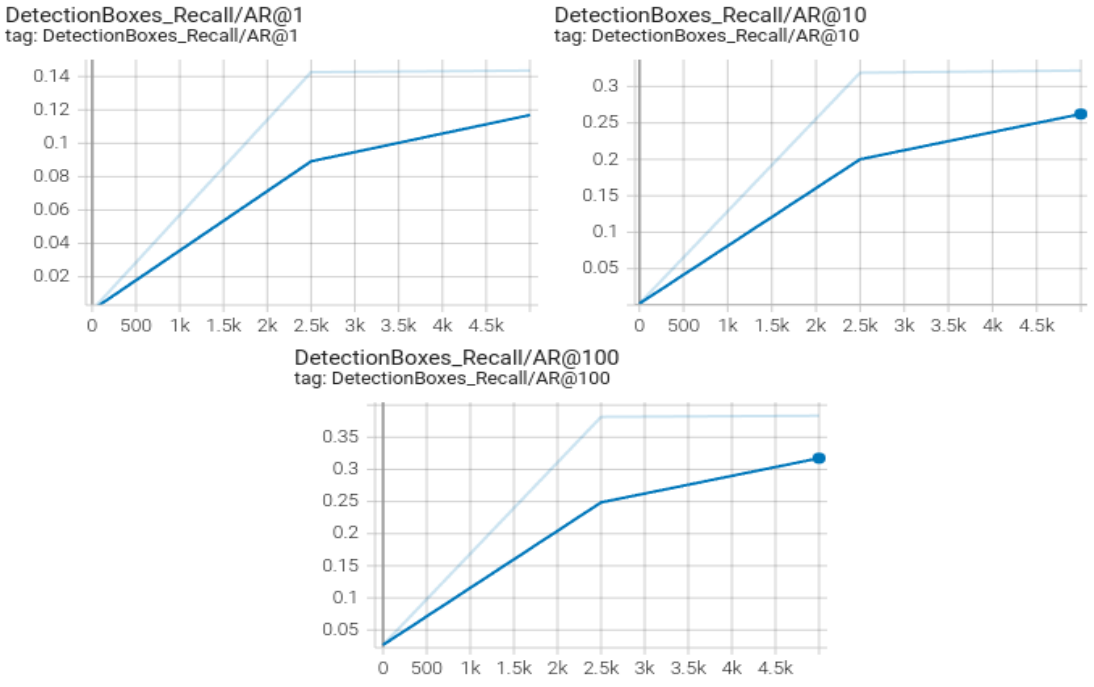

Figura 7.28: Evolución de *mAR* en base a la cantidad de detecciones

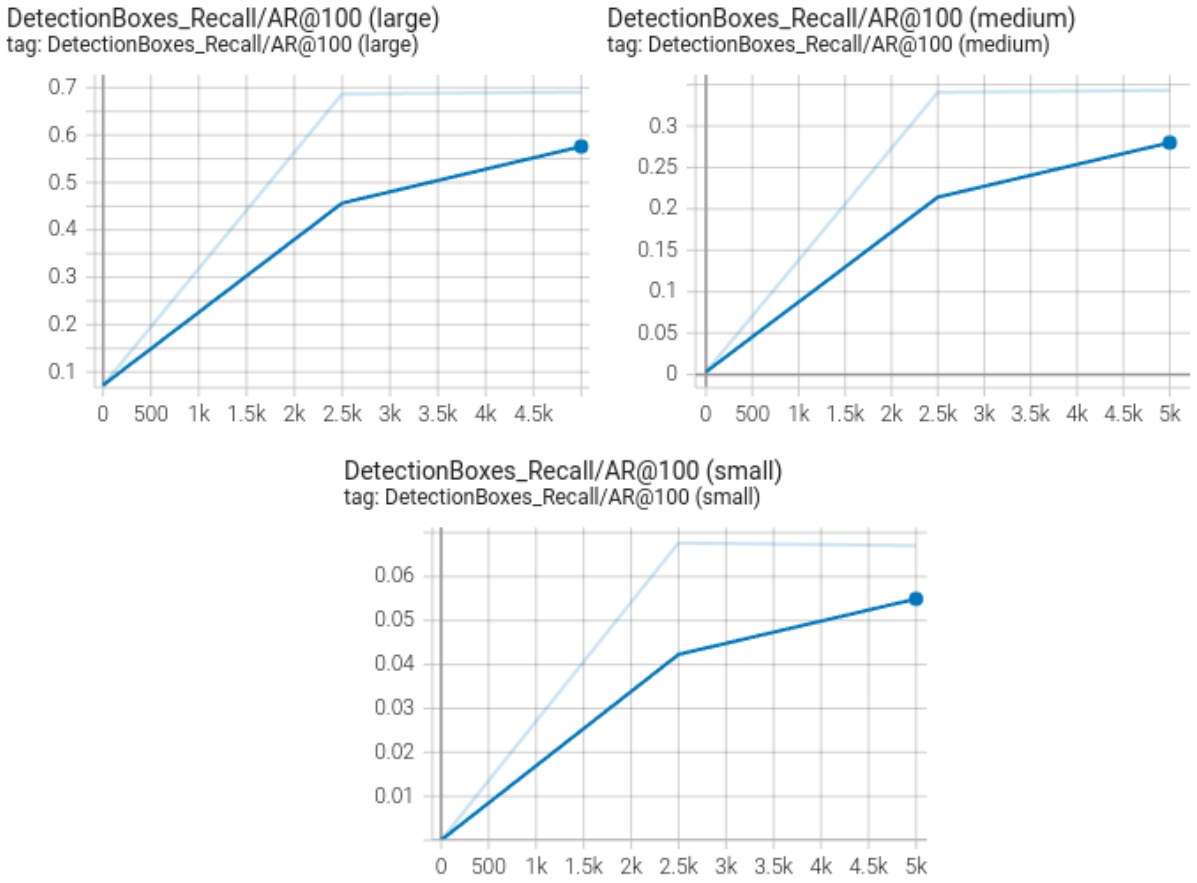

Figura 7.29: Evolución de *mAR* en base al tamaño de los objetos

```
Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.279
Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.520
Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.259
Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.016
Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.188
Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.572
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.144
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.322
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.383
Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.067
Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.343
Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.691
```

Figura 7.30: Informe final de *mean Average Precision* y *mean Average Recall*

4) Mask R-CNN Inception ResNet v2:

- Desafortunadamente, debido a un error imprevisto que surge al tratar de entrenar este modelo, no es posible entrenarlo ni validarlo por el momento.

7.2.3 Detector Test (*Dettest*)

Debido a la simplicidad de este programa y a su pequeño tamaño, no es de interés realizar un análisis exhaustivo del mismo; sin embargo, se aprovecha esta sección para exponer un par de experimentos realizados para probar el funcionamiento de los detectores *Faster R-CNN ResNet 50 v1*, y *SSD MobileNet v2*, que fueron entrenados y validados mediante *Dettool*, como se ha indicado en el apartado 7.2.2.

Los experimentos consistieron el probar el funcionamiento de estos dos detectores con el portátil DELL descrito al principio del apartado 7.2, y una cámara web conectada a este. Ambos se probaron de la siguiente forma, disponiendo uno o varios objetos a cierta distancia, y regulando esta para comprobar cómo variaban las detecciones del modelo. En el caso de *Faster R-CNN ResNet 50 v1*, fueron empleados un portátil, un par de teléfonos móviles, y un par de mandos a distancia, todos ellos dispuestos sobre una mesa de forma que la cámara se iba acercando a ellos. En el caso de *SSD MobileNet v2* se requirió la ayuda de una persona, esta se acercaba poco a poco para comprobar cómo variaban las detecciones que el detector emitía respecto de ella. Por cuestiones de privacidad, solo se incluyen imágenes de la experimentación con *Faster R-CNN ResNet 50 v1*. A continuación se exponen los resultados obtenidos al realizar estos experimentos:

1) Detector *Faster R-CNN ResNet 50 v1 (1s latency approx)*

- Valor umbral de la detección: 0.5.
  - 2,5m: a esta distancia el detector tenía dificultades para identificar los objetos más pequeños, los móviles, e incluso se confundían estos con los mandos. El teclado del portátil no pudo detectarse a esta distancia. Esta situación se representa a través de la imagen de la figura 7.31.

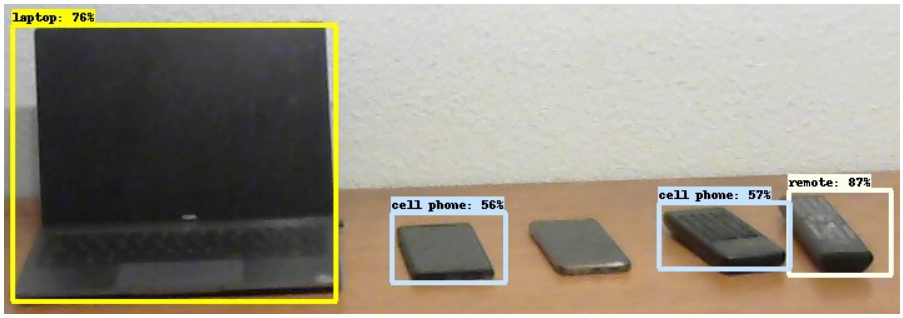

Figura 7.31: Detección de dispositivos electrónicos a una distancia de 2,5m

- 1,5m: a esta distancia la detección de objetos mejoró, aunque aún se daban ciertas confusiones en las clasificaciones de los móviles y los mandos. El teclado del ordenador portátil comenzaba a detectarse, sin embargo, su detección era costosa y poco precisa, pues se puede ver en la figura 7.32 cómo su *bounding box* generado no es muy correcto.

Figura 7.32: Detección de dispositivos electrónicos a una distancia de 1,5m

- 0,5m: con esta distancia las detecciones ya ocurrían frecuentemente y con más precisión, sin embargo, se experimentaban eventualmente algunas dificultades en la detección. Por otro lado, puede verse en la imagen de la figura 7.33 que, aunque la localización de los *bounding boxes* es bastante aproximada, aún es susceptible de mejorarse.

Figura 7.33: Detección de dispositivos electrónicos a una distancia de 0,5m

- Valor umbral de la detección: 0.8.
  - 2,5m: con este nivel de umbral y a esta distancia, prácticamente solo se detectaba el ordenador portátil, y alguna vez algún mando o móvil. Este resultado se puede ver en la imagen de la figura 7.34.

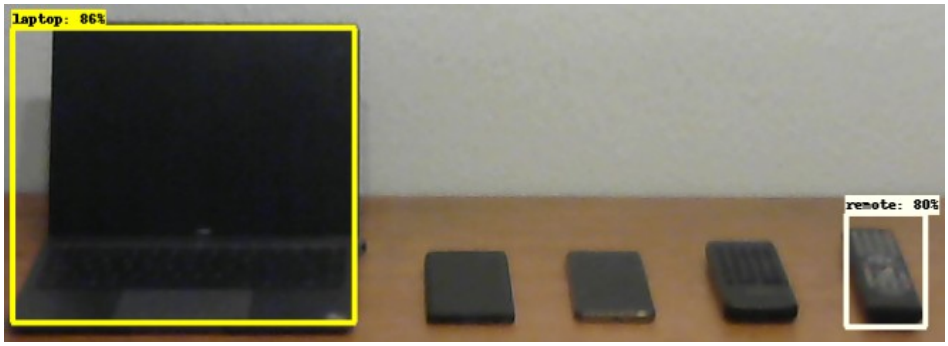

Figura 7.34: Detección de dispositivos electrónicos a una distancia de 2,5m

- 1,5m: con esta distancia las detecciones de los objetos pequeños se daban con mayor frecuencia, sin embargo, estas apenas pasaban el valor umbral, por lo que oscilaban mucho. Por otra parte, continuaba existiendo cierta confusión a la hora de distinguir entre los mandos y los móviles. Esto puede verse reflejado en la imagen de la figura 7.35.

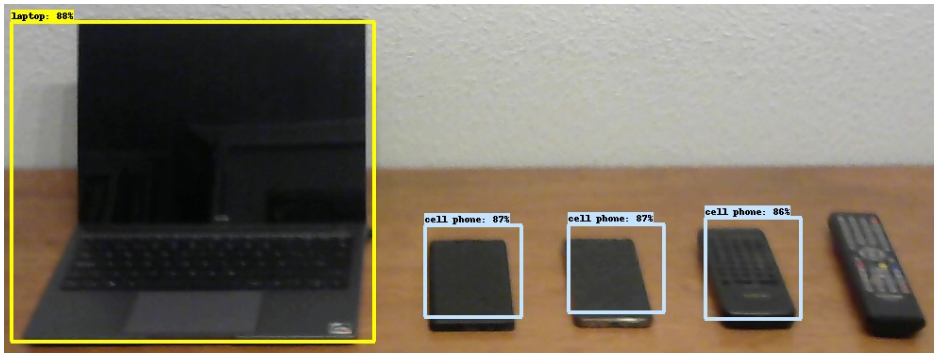

Figura 7.35: Detección de dispositivos electrónicos a una distancia de 1,5m

- 0,5m: incluso estando tan cerca de los móviles y los mandos, seguía habiendo ciertas dificultades para distinguirlos. Con este valor umbral, no se puede identificar el teclado a ninguna distancia. Todo esto se ve reflejado a través de la imagen de la figura 7.36.

Figura 7.36: Detección de dispositivos electrónicos a una distancia de 0,5m

2) Detector: *SSD MobileNet v2* (*real time detector*)

- Valor umbral de la detección: 0.5.
  - 4m: la persona colocada frente a la cámara fue detectada al instante con aproximadamente un valor del 72% de confianza en la detección.
  - 2m: la capacidad para detectar la persona frente a la cámara subió hasta un valor de 86% de confianza en la detección.
  - 1m: a esta distancia no se produjo un incremento de las detecciones, sino que el valor de confianza de estas quedó oscilando en el rango [80 – 90].
- Valor umbral de la detección: 0.8.
  - 4m: la persona frente a la cámara no podía detectarse con este valor umbral.
  - 2m: la persona sí podía ser detectada, y su rango de confianza en la detección se movía entre 84 y 90.
  - 1m: en este caso prácticamente no hubo mejoría, la confianza en la clasificación continuó siendo similar a la del punto anterior.

Capítulo 8 - Conclusiones y trabajo futuro

8.1 Introducción

En este capítulo se exponen las conclusiones sobre la realización de este trabajo, y se comentan aquellos aspectos que quedan pendientes de ser mejorados y de incorporarse al mismo para el perfeccionamiento de la plataforma de trabajo con los modelos de detección de objetos.

8.2 Conclusiones

En este trabajo se propone la creación de una plataforma que simplifica y facilita el ciclo de actividades a realizar para lograr la configuración y puesta en marcha de modelos de *Deep Learning* que han sido creados para la detección de objetos, a saber: recolección y etiquetado de imágenes, creación de *datasets*, entrenamiento, validación y exportación de modelos, y la realización de pruebas de desempeño en un entorno controlado.

Dicha plataforma de trabajo con detectores ha sido creada principalmente con el servicio de *Google Colab (Plus)* [73], cuyo soporte reside en el servicio de *Google Engine* [72]. *Colab* ha hecho posible su desarrollo con la tecnología de Python3 [57] y el *framework* que ofrece la *TensorFlow Object Detection API* [67], así como su despliegue y ejecución en la nube de Google. También ha proporcionado unos estupendos recursos de hardware que han sido la clave para poder trabajar con modelos de detección de objetos, ya que las actividades que están relacionadas con estos requieren una considerable capacidad de espacio de almacenamiento (HDD), una buena memoria de trabajo (RAM), y gran capacidad de cómputo (GPU [64]; TPU [65]). Adicionalmente, se han utilizado otros servicios de *Google*, como *Google Drive* [70], o *Google Cloud Storage* [78]. El primero es importante para el almacenamiento de los *datasets* con los que entrenar y validar detectores, así como para el almacenamiento de los mismos, mientras que el segundo, ha sido de vital importancia para lograr el entrenamiento de los modelos con una TPU.

El resultado obtenido ha sido una plataforma económicamente asequible, pues el costo mensual de *Google Colab Plus* [73] sumado al costo del uso de un *bucket* en *Google Cloud Storage* [78], no debería superar los 35-40€ mensuales.

En cuanto a las capacidades de la plataforma desarrollada, esta es completamente capaz de generar automáticamente *datasets* de imágenes a partir de COCO 2017 (*Common Objects in Context*) [68], lo que ahorra mucho tiempo, al no tener que buscar imágenes, etiquetarlas, o tener que realizar el reparto de imágenes entre los conjuntos para el entrenamiento y la validación de forma manual. Por otra parte, es posible realizar el entrenamiento de modelos de detectores de objetos recurriendo a la aceleración por hardware de este proceso, ya sea con una GPU [64], o una TPU (*Tensor Processing Unit*) [65]. Finalmente, los modelos entrenados y validados pueden testearse con un programa portable que permite regular el umbral de confianza de sus detecciones y las máscaras de los objetos.

En lo que respecta a los detectores de objetos, se han analizado desde un punto de vista teórico los modelos: R-CNN, FAST R-CNN, FASTER R-CNN, MASK R-CNN, SSD, y YOLO. Mientras que los únicos modelos que han podido ser analizados dentro del marco técnico, con las prestaciones actuales de la plataforma de trabajo creada, han sido: *SSD MobileNet v1*, *SSD MobileNet v2*, y *Faster R-CNN ResNet 50 v1*. Por lo que, en este sentido, queda pendiente la mejora de la plataforma de trabajo para proporcionar a los usuarios los dos modelos de detector de interés restantes, que serían: Mask R-CNN y YOLO.

8.3 Trabajo futuro

Queda pendiente la realización de las siguientes correcciones al estado actual de la plataforma de trabajo con detectores de objetos:

- Solucionar el *bug* que impide reflejar los resultados del segundo punto de validación en adelante en *TensorBoard* [79] para el modelo *Faster R-CNN*.
- Corregir el error que impide entrenar y validar el detector *Mask R-CNN*.

A continuación, se enuncian algunas mejoras que incrementarían las capacidades de la plataforma de trabajo con detectores de objetos:

- Emplear técnicas de cómputo en paralelo para la descarga y composición de los *datasets* que realiza la herramienta *CDSGenerator*. Esta estrategia haría un mejor uso de los recursos de cómputo, reduciría los tiempos de creación de los *datasets*, y haría que el programa pudiera sufrir menos desconexiones ante una supuesta inactividad, o una ejecución muy larga.

- Hacer que el entrenamiento y validación de los detectores se lleve a cabo en paralelo en *Dettool*. Esta mejora permitiría aprovechar mejor la capacidad de cómputo de *Colab*, así como la cantidad de tiempo de que se dispone para entrenar y validar los modelos, haciendo que el programa sea menos susceptible de sufrir desconexiones por una supuesta inactividad, o ante una ejecución demasiado larga.
- Investigar cómo modificar el *pipeline* del detector *Mask R-CNN*, u otros, para hacer que estos sean compatibles con el modo de entrenamiento con la TPU.
- Crear una versión de *Dettest* en *Google Colab* [73]. Aunque este programa se hizo portable para probar los detectores en otros sistemas independientes a los que realizan cómputo en la nube, podría crearse una versión en la nube para que toda la funcionalidad de la plataforma esté disponible en ella.
- Añadir opciones a *Dettest* para incrementar el número de formas de testear los detectores. Actualmente los detectores se testean observando cuál es su comportamiento mientras graban una secuencia indefinida de imágenes, es decir, un *streaming*. Sería positivo, por otra parte, el poder realizar un testeo con una secuencia finita de imágenes seleccionadas por un usuario.
- Añadir gráficas con métricas adicionales, o más visuales, para la evaluación de los detectores. Para aplicar esta estrategia, siempre se podrían programar las funcionalidades a mano, sin embargo, tal vez otra estrategia mucho más inteligente sería pasar los resultados de las detecciones realizadas por los modelos al formato JSON [75] de COCO (*Common Objects in Context*) [68], para que a través del software de *FiftyOne* [77] se pudieran leer, con la finalidad de generar matrices de confusión, las curvas de *precision* frente a *recall* para cada clase, y otros informes.
- Investigar si es posible ejecutar un mismo *Notebook* en varias pestañas del navegador, ya que, si cada pestaña fuera una ejecución independiente del mismo programa sobre *Google Colab*, entonces eso podría ser una puerta para realizar los entrenamientos de varios modelos a la vez, aunque para realizar esa mejora, habría de implementarse un mecanismo de trabajo con múltiples *buckets* en *Google Cloud Storage* [78], de forma que en cada uno se gestionen los archivos del entrenamiento de un modelo.

Capítulo 9 - Conclusions and future work

9.1 Introduction

In this chapter conclussions of the realization of the current work are exposed, and those aspects that have to be improved and incorporated to it for the perfection of the working platform with object detection models, are comented.

9.2 Conclusions

In this work it is proposed the creation of a platform to simplify and facilitate the cycle of activities that have to be made to achieve the setting up and the configuration of Deep Learning models that have been created for object detection, they are: image collection and labeling, dataset creation, training, validation and export of models, and the making of performance tests in some controlled environment.

That working platform with detectors has been created with the *Google Colab (Plus)* [73] service mainly, whose support resides in *Google Engine* [72] service. Colab has made possible its development with the technology of Python3 and the framework that is offered by *TensorFlow Object Detection API* [67], as well as its deployment and execution in Google Cloud. Also, it has provided some great hardware resources that have been the key to work with object detection models, so that the activities that are related with these need a considerable disk space (HDD), a good working memory (RAM), and a high computing capacity (GPU [64]; TPU [65]). In addition, other *Google Services*, as *Google Drive* [70], or *Google Cloud Storage* [78] have been used. The first of them is important for dataset storage to train and validate detectors, and also to store them, while the second has been very important to achieve models training with a TPU.

The obtained result has been an economically reasonable platform, so the monthly cost of *Google Colab Plus* added to a bucket cost in *Google Cloud Storage*, shouldn't be greater than 35-40€.

Regarding the capabilities of the developed platform, this is completely able to generate automatically image datasets from COCO [68] (2017) (Common Objects in

Context), which saves a lot of time because it isn't necessary to search for images, label them, or have to distribute the images between the training and validation sets manually. On the other hand, it's possible to make the model training appealing to hardware acceleration of this process, either with a GPU [64], or a TPU (*Tensor Processing Unit*) [65]. Finally, trained and validated models can be tested with a portable program that allow to regulate the threshold of confidence of their detections and object masks.

Regarding the object detectors, the following models have been analyzed from a theoretical point of view: R-CNN, FAST R-CNN, FASTER R-CNN, MASK R-CNN, SSD, and YOLO. While the models that we have been analyzed within the technical framework, with the current features of the working platform created, have been: SSD MobileNet v1, SSD MobileNet v2, and Faster R-CNN ResNet 50 v1. So, in this sense, it remains pending the improvement of the working platform to provide users the two remaining detector models of interest, these would be: Mask R-CNN and YOLO.

9.3 Future work

The next corrections are pending to be applied to the current state of the working platform with object detectors.

- To fix the bug that prevents to reflect the results from the second validation point on onwards in *TensorBoard* [79], for the Faster R-CNN model.
- To correct the mistake that prevents to train and validate the Mask R-CNN detector.

Below, some improves are announced that would increase the capabilities of the working platform with object detectors:

- To investigate how to modify the pipeline of the Mask R-CNN detector, or others, to make that those to be compatible with the TPU [65] training mode.
- To use parallel computing techniques for the download and composition of datasets that CDSGenerator performs. This strategy would make a better use of the computing resources, would reduce the creation times of the datasets, and would make that the program could suffer less disconnections because of a supposed inactivity, or a very long execution.

- To make that the training and validation of the detectors must be carried out in parallel in *Dettool*. This improvement would allow to make a better use of *Google Colab* [73] compunting capacity, as well as the avaliable time to train and validate the models, making the program to be less susceptible of suffering disconnections for a an idle period of inactivity, or for a too long execution.
- To create a Dettest version on *Google Colab*. Although this program was made portable to test detectors in independent computing systems of the cloud computing of a company, a version in the cloud could be created so all the functionality can be available in it.
- To add options to *Dettest* to increase the number of ways to test the detectors. Currently, detectors are tested observing its behaviour meanwhile they are recording a video. It would be positive, on the other hand, to realize a test with a finite sequence of images selected by an user.
- To add graphics with additional measures, or more visual, for the evaluation of the detectors. To apply this strategy, it could be programmed by hand always, however, maybe other strategy much more clever would be to pass the results of the detections realized by the models to the JSON [75] format of COCO (*Common Objects in Context*) [68] so that the *FiftyOne* [77] software it could be read, with the finality of generate confusion matrices, the precision recall curves for each class, and other reports.
- To investigate if it is possible to execute the same Notebook in some tabs of the browser, so that, if each tab would be an independent execution of the same program over *Google Colab* [73], then that could be an open door to realize the training of various models at the same time, although to realize this improve, it should be implemented a mechanism of working with several buckets in *Google Cloud Storage* [78], so that in each of them the files related with a model training are managed.

BIBLIOGRAFÍA

[1] Invierno IA (2022) Wikipedia. [Consultado el 12 de abril de 2022]. Disponible en: https://es.wikipedia.org/wiki/Invierno_IA

[2] Pajares, G., Cruz, J.M. (2007). Visión por Computador: Imágenes digitales y aplicaciones. RA-MA, Madrid.

[3] Pajares, G., Herrera, P.J, Besada, E. (2021). Aprendizaje Profundo. RC Libros, Madrid.

[4] Rumelhart, D. E., McClelland, J.L. (1986a). Parallel Distributed Proccesing. Vol 1: Foundations. MIT Press.

[5] MVTec Software - Experts for Machine Vision Software (n.d.). Machine Vision vs. Deep Learning [Consultado el 12 de abril de 2022]. Disponible en: https://www.mvtec.com/technologies/deep-learning/classic-machine-vision-vs-deep-learning

[6] Tesla (n.d.). Artificial Intelligence & Autopilot [Consultado el 12 de abril de 2022]. Disponible en: https://www.tesla.com/AI

[7] Shazeer, N., Fatahalian, K., Mark, W.R., Mullapudi, R.T. (2018). HydraNets: Specialized Dynamic Architectures for Efficient Inference [Consultado el 12 de abril de 2022]. Disponible en: https://ieeexplore.ieee.org/document/8578941

[8] Airbus (n.d.). Autonomous flight [Consultado el 12 de abril de 2022]. Disponible en: https://www.airbus.com/en/innovation/autonomous-connected/autonomous-flight

[9] Airbus (n.d.). Wayfinder [Consultado el 12 de abril de 2022]. Disponible en: https://acubed.airbus.com/projects/wayfinder/

[10] Airbus (2020). Airbus concludes ATTOL with fully autonomous flight tests [Consultado el 12 de abril de 2022]. Disponible en: https://www.airbus.com/en/newsroom/press-releases/2020-06-airbus-concludes-attol-with-fully-autonomous-flight-tests

[11] Airbus (2019). How Wayfinder is Using Neural Networks for Vision-Based Autonomous Landing [Consultado el 12 de abril de 2022]. Disponible en: https://acubed.airbus.com/blog/wayfinder/how-wayfinder-is-using-neural-networks-for-vision-based-autonomous-landing/

[12] Liu, W., Anguelov, D., Erhan, D., Szegedy, C., Reed, C., Fu, C.Y., Berg, A.C. (2016). SSD: Single Shot MultiBox Detector [Consultado el 12 de abril de 2022]. Disponible en: https://arxiv.org/abs/1512.02325

[13] Landing AI (2020). Landing AI Creates an AI Tool to Help Customers Monitor Social Distancing in the Workplace [Consultado el 18 de abril de 2022]. Disponible en: https://landing.ai/landing-ai-creates-an-ai-tool-to-help-customers-monitor-social-distancing-in-the-workplace/

[14] Ren, S., He, K., Girshick, R., Sun, J., (2016). Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks. [Consultado el 12 de abril de 2022]. Disponible en: https://arxiv.org/abs/1506.01497

[15] Ramón y Cajal, S. (1899). Textura del Sistema Nervioso del Hombre y de los vertebrados. N. Moya, Madrid.

[16] Brío, B.M., Sanz-Molina, A. (2006). Redes Neuronales y Sistemas Borrosos, RA-MA, Madrid.

[17] Pajares, G., Cruz-García, J.M. (Eds.) (2010). Aprendizaje Automático. Un enfoque práctico. RA-MA, Madrid.

[18] Rosenblatt, F. (1958). The perceptron: A probabilsitic model for information torage and organization in the brain. Psychological Review, 65, 386 – 408.

[19] Rumelhart, D. E., McClelland, J.L. (1986b). Parallel Distributed Proccesing. Vol 2: Psychological and biological models. MIT Press.

[20] Müller, B., Reinhardt, J. (1990) Neural Networks. An introduction. Springer-Verlag.

[21] Rumelhart, D.E, Hinton, G.E., Williams, R.J. (1986). Learning representations by backpropagating errors. Nature, 323, 533 – 536.

[22] LeCun, Y. (1989). Backpropagation applied to handwritted zip code

recognition. Neural Computation, 1(4), 541-551.

[23] LeCun, Y. (1989). Generalization and network design strategies. Technical ReportCRG-TR-89-4, Unniversity of Toronto. 326, 345747. [Consultado el 4 de junio de 2022]. Disponible en: http://yann.lecun.com/exdb/publis/pdf/lecun-89.pdf

[24] Bishop, C.M. (1994). Neural Networks and their applications. Rev. Sci. Instrum., 65, 6, 1803-1832.

[25] Baum, E.B., Haussler, D. (1989). What size net gives valid generalization? Neural Computation, 1, 151-160.

[26] Haykin, S. (1999). Neural Networks. A comprehensive Foundation. 2ª edición. Prentice-Hall, 1994, 1999.

[27] Krizhevsky, A., Sutskever, I., Hinton, G. (2012). ImageNet Classification with Deep Convolutional Neural Networks." In Proc. 25th Int. Conf. on Neural Information Processing Systems (NIPS'12), vol. 1, 1097-1105.

[28] LeCun, Y., Bottou, L., Orr, G.B., Müller, K.R. (1998). Efficient BackProp. In: Orr, G.B., Müller, KR. (eds) Neural Networks: Tricks of the Trade. Lecture Notes in Computer Science, vol 1524. Springer, Berlin, Heidelberg.

[29] Ioffe, S., Szegedy, C. (2015). Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covarianze Shift. [Consultado el 5 de junio de 2022]. Disponible en: https://arxiv.org/abs/1502.03167

[30] Bochkovsky, A., Wang, C.Y., Mark-Liao, H.Y. (2020). YOLOv4: Optimal Speed and Accuracy of Object Detection. [Consultado el 5 de junio de 2022]. Disponible en: https://arxiv.org/abs/2004.10934

[31] Zhang, A., Lipton, Z. C., Li, M., Smola, A. J. (2021). Dive into Deep Learning. [Consultado el 5 de junio de 2022]. Disponible en: https://arxiv.org/pdf/2106.11342.pdf

[32] Girshick, R., Donahue, J., Darrel, T., Malik, J. (2014). Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation. Proc. 2014 IEEE Conf. on Computer Vision and Pattern Recognition (CVPR'14), 580-587.

[33] Endres, I., Hoiem, D. (2010). Category Independent Object Proposals. In: Daniilidis K., Magaros P., Paragios N. (eds) Computer Vision – ECCV 2010. Lecture Notes in Computer Science, vol 6315. Springer, Berlin, Heidelberg.

[34] Alexe, B., Desealers, T., Ferrari, V. (2012). Measuring the objectness of image windows. IEEE Trans. Pattern Anal. Machine Intell., 34(11), 2189-2102.

[35] Zitnick, C.L., Dollar, P. (2014). Edge boxes: Locating object proposals from edges. Computer Vision-ECCV. Springer Internacional Publishing, 391 – 405.

[36] Wang, X., Ming, Y., Zhu, S., Lin, Y. (2015). Regionlets for Generic Object Detection. IEEE Trans. Pattern Anal. Machine Intell., 37(10), 2071 – 2084.

[37] Jaccard, P. (1908). Nouvelles reserches sur la distribution florale. Bull. Soc. Vaudoise Sci. Nat. 44, 233 – 270.

[38] Boser, B.E., Guyon, I.M. and Vapnik, V.N. (1992) A Training Algorithm for Optimal Margin Classifiers. Proceedings of the 5th Annual Workshop on Computational Learning Theory (COLT'92), Pittsburgh, 144-152. [Consultado el 5 de junio de 2022]. Disponible en: https://dl.acm.org/doi/pdf/10.1145/130385.130401

[39] Cortes, C., Vapnik, V. (1995). Support-Vector Networks. Machine Learning, 20(3), 273 - 297. [Consultado el 5 de junio de 2022]. Disponible en: https://link.springer.com/content/pdf/10.1007/BF00994018.pdf

[40] Pang, J., Chen, C., Shi, J., Feng, H., Ouyang, W., Lin, D. (2019). Libra R-CNN: Towards balanced learning for object detection. In Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 821 – 830.

[41] Girshick, R. (2015). Fast R-CNN. Proc. IEEE International Conference on Computer Vision, 1440 – 1448, 7 – 13, Santiago, Chile.

[42] He, K., Zhang, X., Ren, S., Sun, J., (2015). Spatial Pyramyd Pooling in Deep Convolutional Networks for Visual Recognition. [Consultado el 5 de junio de 2022]. Disponible en: https://arxiv.org/abs/1406.4729

[43] Everingham, M., Van Gool, L., Williams, C.K.I., Winn, J., Zisserman, A. (2010). The PASCAL visual object classes (VOC) challenge. Int J. Computer Vision,

88(2), 303 – 338.

[44] Everingham, M., Van Gool, L., Williams, C.K.I., Winn, J., Zisserman, A., Ali-Eslami, S.M. (2015). The PASCAL Visual Object Classes Challenge: A Retrospective. Int. J. Computer Vision, 111(1), 98 – 136.

[45] Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., Berg, A.C., Fei-Fei, L. (2015). ImageNet Large Scale Visual Recognition Challenge. International Journal of Computer Vision (IJCV). 115(3), 211 – 252.

[46] He, K., Gkioxari, G., Dollár, P., Girshick, R. (2018). Mask R-CNN. [Consultado el 5 de junio de 2022]. Disponible en: https://arxiv.org/abs/1703.06870

[47] Long, J., Shelhammer, E., Darrel, T. (2015). Fully convolutional networks for semantic segmentation. In Proc. IEEE Conf. Computer Vision and Pattern Recognition (CVPR'15), 3431 – 3440, Boston, MA, USA.

[48] He, K., Gkioxari, G., Dollár, P., Girshick, R. (2016). Deep residual learning for image recognition. In Proc. of the IEEE conference on computer vision and pattern recognition, 770 – 778.

[50] Xie, S., Girshick, R., Dollár, P., Tu, Z., He, K. (2017). Aggregated Residual Transformations for Deep Neural Networks. [Consultado el 5 de junio de 2022]. Disponible en: https://arxiv.org/abs/1611.05431

[51] Rao, J., Qiao, Y., Ren, F., Wang, J., Du, Q. (2017). A mobile Outdoor Augmented Reality Method Combining Deep Learning Object Detection and Spatial Relationships for Geovisualization. Sensors, 17, 1951.

[52] Redmon, J., Divvala, S., Girshick, R., Farhadi, A. (2016). You Only Look Once: Unified, real-time object detection. Proc. IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 779 – 788. Las Vegas, USA.

[53] Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., Rabinovich, A. (2014). Going Deeper with Convolutions. [Consultado el 5 de junio de 2022]. Disponible en: https://arxiv.org/abs/1409.4842

[54] Índice TIOBE (2021) Wikipedia. [Consultado el 20 de junio de 2022]. Disponible en: https://es.wikipedia.org/wiki/%C3%8Dndice_TIOBE

[55] Python (2022). History and License. [Consultado el 20 de junio de 2022]. Disponible en: https://docs.python.org/3/license.html

[56] FreeBSD (2022). Qué es BSD. [Consultado el 20 de junio de 2022]. Disponible en: https://docs.freebsd.org/doc/8.3-RELEASE/usr/share/doc/freebsd/es/articles/explaining-bsd/article.html

[57] Python (2022). What is Python? Executive Summary. [Consultado el 20 de junio de 2022]. Disponible en: https://www.python.org/doc/essays/blurb/

[58] CPython (2022) Wikipedia. [Consultado el 20 de junio de 2022]. Disponible en: https://es.wikipedia.org/wiki/CPython

[59] TensorFlow (2022). [Consultado el 20 de junio de 2022]. Disponible en: https://www.tensorflow.org

[60] Google Research (2022). Brain Team. [Consultado el 20 de junio de 2022]. Disponible en: https://research.google/teams/brain/

[61] Apache (2022). Apache License, Version 2.0. [Consultado el 20 de junio de 2022]. Disponible en: https://www.apache.org/licenses/LICENSE-2.0

[62] Apache (2022). The Apache Software Foundation. [Consultado el 20 de junio de 2022]. Disponible en: https://www.apache.org/

[63] NVIDIA (2022). CUDA Zone. [Consultado el 20 de junio de 2022]. Disponible en: https://developer.nvidia.com/cuda-zone

[64] NVIDIA (2022). ¿Qué es la computación acelerada por GPU? [Consultado el 20 de junio de 2022]. Disponible en: https://www.nvidia.com/es-la/drivers/what-is-gpu-computing/

[65] Google Cloud (2022). Cloud Tensor Processing Unit (TPU). [Consultado el 20 de junio de 2022]. Disponible en: https://cloud.google.com/tpu/docs/tpus?hl=es_419

[66] Keras (2022). Keras: the Python deep learning API. [Consultado el 20 de junio de 2022]. Disponible en: https://keras.io/

[67] TensorFlow Object Detection API (2022). [Consultado el 20 de junio de 2022]. Disponible en: https://github.com/tensorflow/models/tree/master/research/object_detection

[68] Lin, TY., Maire, M., Belongie, S., Bourdev, L., Girshick, R., Hays, J., Perona, P., Ramanan, D., Zitnick, C. L., Dollár, P. (2015). Microsoft COCO: Common Objects in Context. [Consultado el 20 de junio de 2022]. Disponible en: https://arxiv.org/abs/1405.0312

[69] Kuznetsova, A., Rom, H., Alldrin, N., Uijlings, J., Krasin, I., Pont-Tuset, J., Kamali, S., Popov, S., Malloci, M., Kolesnikov, A., Duerig, T., Ferrari, V. (2020). The Open Images Dataset V4: Unified image classification, object detection, and visual relationship detection at scale. [Consultado el 20 de junio de 2022]. Disponible en: https://arxiv.org/pdf/1811.00982.pdf

[70] Google Drive (2022). [Consultado el 20 de junio de 2022]. Disponible en: https://www.google.com/intl/es_es/drive/

[71] Google Cloud Platform (2022). [Consultado el 20 de junio de 2022]. Disponible en: https://cloud.google.com/gcp

[72] Google Compute Engine (2022). [Consultado el 20 de junio de 2022]. Disponible en: https://cloud.google.com/compute

[73] Google Colab (2022). [Consultado el 20 de junio de 2022]. Disponible en: https://colab.research.google.com/

[74] Jupyter (2022). [Consultado el 20 de junio de 2022]. Disponible en: https://jupyter.org/

[75] JSON (2022). Introducing JSON. [Consultado el 20 de junio de 2022]. Disponible en: https://www.json.org/

[76] UML (2022). [Consultado el 20 de junio de 2022]. Disponible en: https://www.uml.org/

[77] FiftyOne (2022). [Consultado el 20 de junio de 2022]. Disponible en: https://voxel51.com/docs/fiftyone/

[78] Google Cloud Storage (2022). [Consultado el 20 de junio de 2022]. Disponible

en: https://cloud.google.com/storage

[79] TensorBoard (2022). [Consultado el 20 de junio de 2022]. Disponible en: https://www.tensorflow.org/tensorboard

[80] Creative Commons (2022). Atribución 4.0 Internacional (CC BY 4.0). [Consultado el 8 de julio de 2022]. Disponible en: https://creativecommons.org/licenses/by/4.0/deed.es

[81] Pixabay (2022). Neurona [Consultado el 9 de junio de 2022]. Disponible en: https://pixabay.com/es/vectors/neurona-c%c3%a9lula-del-nervio-axon-296581

APÉNDICES

Apéndice 1 - Manual de usuario de CDSGenerator

Teniendo presentes las figuras 6.1 y 6.2, el funcionamiento de este programa es como sigue. Primero, debe tener ubicado el proyecto *ObjectRecognition* en la raíz de *Google Drive*, acto seguido, ejecutar de forma secuencial todas las celdas del *Notebook* para que el programa cargue por completo, ya que este se parecerá más a una aplicación basada en formularios que a un *Notebook* al uso. Durante la carga del programa aparecerá un cuadro de diálogo que solicitará permiso para acceder a *Google Drive*, que debe ser concedido para proceder a su ejecución. Una vez cargadas todas las celdas, el programa puede comenzar a utilizarse marcando las categorías (casillas) de los objetos que se desea incluir en el *dataset*, como este estará basado en COCO 2017, se puede seleccionar hasta 80 categorías de objetos.

Para realizar esta labor pueden seleccionarse manualmente las categorías, o utilizar los botones de *Check All*, y *Uncheck All*, para facilitar la tarea. Una vez han sido seleccionadas las categorías, se ha de indicar para cada una cuántas imágenes se desean incluir en el *dataset* de cara a los conjuntos de entrenamiento y validación, por defecto se sigue una regla 80 – 20 (imágenes) para cualquier clase escogida. A continuación, el usuario debe pulsar el botón *Create Dataset*; cuando ello ocurre, el programa lanza las peticiones para descargar las imágenes que compondrán el *dataset*, donde este se ha de entender como una carpeta que tiene los conjuntos para el entrenamiento y la validación, que son realmente los *datasets* generados.

Formado el *dataset*, este se puede visualizar a través del visor gráfico de *FiftyOne*, que es cargado en una celda del *Notebook* con la ayuda de un *magic*. Siendo los *magics* una serie de comandos especiales que proporcionan funciones de diversa complejidad de programación en un *Notebook*. Dentro de este tipo de comandos pueden encontrarse, por ejemplo, algunas llamadas a los programas del sistema GNU/Linux, y la llamada al instalador de paquetes de Python (*pip*).

Para visualizar el *dataset* con *FiftyOne*, pulsar sobre el icono de la esquina superior izquierda dentro de la celda donde *FiftyOne* se ejecuta, lo que refrescará el *dataset*.

Después de esto, indicar en la celda de texto a la derecha del icono el conjunto de datos que se desea ver, que recordemos, son los verdaderos *dataset*. Para esto último, puede escribirse la palabra *training*, o la palabra *validation*, o bien pulsar en la celda de texto con el ratón, y seleccionar uno de estos dos conjuntos en la lista desplegable que se mostrará. Estos pasos pueden verse reflejados en la figura A.1.1, en la que se muestra el icono a pulsar para refrescar, y la lista desplegable de la que seleccionar un conjunto para visualizar, ambos remarcados.

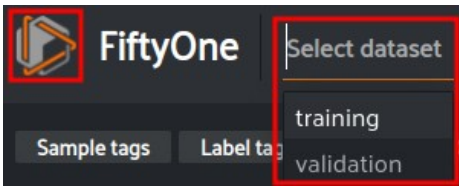

Figura A.1.1: Elementos básicos del interfaz gráfico de *FiftyOne*

Una vez realizados los pasos anteriores el programa visor se actualizará y mostrará un interfaz similar al de la figura A.1.2.

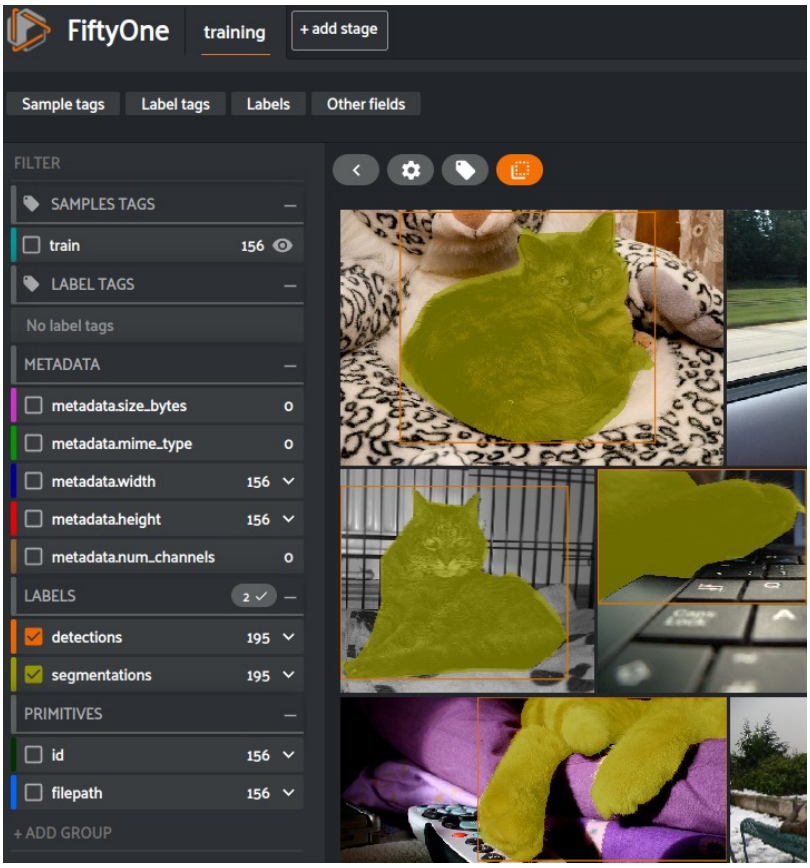

Figura A.1.2: Opciones de *FiftyOne* para la exploración de un *dataset*

En el menú del panel lateral izquierdo puede escogerse la información que se desea ver en el *collage* de imágenes de la derecha que conforma el conjunto elegido.

- Número de imágenes (ejemplos) etiquetadas (*samples tags*): 156 (*training*).

- Metadatos de cada imagen (*metadata*): en este caso solo se ha incluido la información relativa al alto y ancho de las 156 imágenes.

- Etiquetas (labels): existen 195 objetos ubicados (*detections*) y segmentados (*segmentations*) entre las 156 imágenes que conforman el conjunto que se ha seleccionado, en este caso corresponde a un ejemplo de 80 gatos y 80 perros. En cualquier caso conviene tener en cuenta el aviso correspondiente al efecto:

    - En este caso no se alcanzan las 160 imágenes en total debido a que el conjunto de imágenes de gatos, y el de perros, tienen 4 imágenes en común, por ello marca que el total de imágenes es 156.

    - En otras circunstancias podría ocurrir que el conjunto de imágenes por categoría solicitado para formar parte de un conjunto no llegase a esa cifra, por ejemplo, en COCO hay un total de 177 imágenes de perros, pero se solicita un número de imágenes de perros que sea superior a este, *FiftyOne* solo podrá proporcionar el máximo disponible de 177.

- Primitivas (*primitives*): identificadores (*id's*) y rutas (*paths*) a las imágenes.

El menú del panel superior formado por los botones de *Sample tags*, *Label tags*, *Labels*, y *Other fields*, se puede utilizar para ver esta información a través de un gráfico de barras. Por ejemplo, si se desea ver la cantidad de objetos identificados (*detections*) y segmentados (*segmentations*), de cada categoría, se seleccionará la opción *Labels*, que cargará un gráfico que sirve para comprobar si la cantidad de objetos de totas las categorías están aproximadamente balanceadas. En general se considera que un conjunto está balanceado si la diferencia de detecciones entre las clases no es excesiva. En el ejemplo que se muestra en la figura A.1.3, la diferencia es de 1 detección, por lo que se considera que sí está balanceado.

Figura A.1.3: Balanceo de etiquetas

Como las imágenes se descargan de forma aleatoria del *dataset* de COCO, es muy poco probable que los conjuntos de entrenamiento y validación estén balanceados, por lo que la estrategia más recomendable será consultar las tablas de COCO en el próximo apéndice, las cuales contienen la cantidad de imágenes y detecciones por cada una de las categorías del *dataset*, para así poder ajustar el número de imágenes por clase, y realizar varias descargas de los conjuntos de entrenamiento y validación, hasta que una configuración dé como resultado una diferencia de detecciones aceptable entre clases, o entre la clase que más detecciones tiene y la que menos, para ambos conjuntos de datos.

Balanceados los conjuntos de entrenamiento y validación, solo resta indicar cuál será el nombre del *dataset* al exportarse, para lo cual ha de rellenarse la casilla de *Dataset Name*, y pulsar el botón *Export To Google Drive*. Esto último hace que se ejecute una modificación del *script create_coco_tf_records* de la *TensorFlow Object Detection API,* que toma las imágenes y sus anotaciones en JSON*,* y genera con ellas los ficheros **.records* para el entrenamiento y la validación. Generados estos archivos, estos se exportan automáticamente junto a otros a la carpeta *datasets*, dentro del proyecto *ObjectRecognition*, en nuestra unidad de *Google Drive*.

Bajo la carpeta de un *dataset* exportado podrá encontrarse lo siguiente:

- Carpetas *training* y *validation*: estas contienen las imágenes de los conjuntos de entrenamiento y validación.
- Archivo *summary.json*: contiene un resumen del contenido del *dataset*.
    - Nombre, fecha de creación, número de clases, y nombres de las clases.

    - Total de imágenes en el *dataset*, en el conjunto de entrenamiento, y en el conjunto de validación.
    - Total de detecciones en el *dataset*, en el conjunto de entrenamiento, y en el conjunto de validación.
    - Para cada clase del conjunto de entrenamiento, y de validación, se indica el número de imágenes que contiene dicha clase, y el total de detecciones de objetos de dicha clase contenidos en el total de imágenes del conjunto.

- Archivo *label_map.pbtxt*: es el fichero donde se define la correspondencia del nombre de cada categoría, con un identificador numérico.

- Archivos *tranining.json* y *validation.json*: son los ficheros que contienen los metadatos del dataset, es decir, contiene información relativa tanto a las imágenes que lo conforman, como a los objetos que están presentes en ellas.

    - De una imagen se almacena su: identificador (*id*), nombre (*name*), altura (*height*), anchura (*width*), licencia (*license*), y url.

    - De cada detección se almacena su: identificador (*id*), identificador de la imagen (*image_id*), rectángulo delimitador (*bounding box*), máscara (*mask*), área (*area*), un identificador del tipo de segmentación (*iscrowd*), y el nombre de la categoría padre (*supercategory*).

- Archivos *training.records* y *validation.records*: son los ficheros que contienen las imágenes y sus metadatos codificadas en registros binarios (*tfrecords*) para poder lograr una manipulación más eficiente de los datos, de forma que puedan reducirse los tiempos de los procesos de entrenamiento y validación.

De estos archivos generados por el proceso de exportación *Google Drive*, solo los archivos *summary.json*, *label_map.pbtxt, training.records*, y *validation.records*, son utilizados por el programa *Dettool* para realizar el entrenamiento y la validación de un detector, por lo que se recomienda encarecidamente no modificarlos.

Apéndice 2 - Tablas de COCO *dataset* 2017

A continuación se muestran en las figuras A.2.1 y A.2.2, la cantidad de imágenes y detecciones de cada objeto que puede encontrarse en el *dataset* COCO (2017) para el conjunto de entrenamiento.

CATEGORY	IMAGES	DETECTIONS
80	117266	860001
##########	##########	##########
person	64.115	262.465
chair	12.774	38.491
car	12.251	43.867
dining table	11.837	15.714
cup	9.189	20.650
bottle	8.501	24.342
bowl	7.111	14.358
handbag	6.841	12.354
truck	6.127	9.973
bench	5.570	9.838
book	5.332	24.715
backpack	5.528	8.720
cell phone	4.803	6.434
sink	4.678	5.610
clock	4.659	6.334
tv	4.561	5.805
potted plant	4.452	8.652
couch	4.423	5.779
dog	4.385	5.508
knife	4.326	7.770
sports ball	4.262	6.347
traffic light	4.139	12.884
cat	4.114	4.768
umbrella	3.968	11.431
bus	3.952	6.069
tie	3.810	6.496
bed	3.682	4.192
vase	3.593	6.613
train	3.588	4.571
fork	3.555	5.479
spoon	3.529	6.165
laptop	3.524	4.970
motorcycle	3.502	8.725
surfboard	3.486	6.126
skateboard	3.476	5.543
tennis racket	3.394	4.812
toilet	3.353	4.157
bicycle	3.252	7.113
bird	3.237	10.806
pizza	3.166	5.821

Figura A.2.1: Conjunto de entrenamiento del *dataset* COCO 2017, parte 1

CATEGORY	IMAGES	DETECTIONS
80	117266	860001
############	############	############
boat	3.025	10.759
skis	3.082	6.646
remote	3.076	5.703
airplane	2.986	5.135
horse	2.941	6.587
cake	2.925	6.353
oven	2.877	3.334
baseball glove	2.629	3.747
kite	2.261	9.076
giraffe	2546	5131
wine glass	2.533	7.913
baseball bat	2.506	3.276
suitcase	2.402	6.192
sandwich	2.365	4.373
refrigerator	2.360	2.637
banana	2.243	9.458
frisbee	2184	2682
elephant	2.143	5.513
teddy bear	2.140	4.793
keyboard	2.115	2.855
cow	1.968	8.147
broccoli	1.939	7.308
zebra	1.916	5.303
mouse	1.876	2.262
stop sign	1.734	1.983
fire hydrant	1.711	1.865
orange	1.699	6.399
carrot	1.683	7.852
snowboard	1.654	2.685
apple	1.586	5.851
microwave	1.547	1.673
sheep	1.529	9.509
donut	1.523	7.179
hot dog	1.222	2.918
toothbrush	1.007	1.954
scissors	947	1.481
bear	960	1.294
parking meter	705	1.285
toaster	217	225
hair drier	189	198

Figura A.2.2: Conjunto de entrenamiento del *dataset* COCO 2017, parte 2

A continuación se muestran en las figuras A.2.3 y A.2.4, la cantidad de imágenes y detecciones de cada objeto que puede encontrarse en el *dataset* COCO (2017) para el conjunto de validación.

CATEGORY	IMAGES	DETECTIONS
80	117266	860001
##########	##########	##########
person	2.693	11.004
chair	580	1.791
car	535	1932
dining table	501	697
cup	390	899
bottle	379	1.025
bowl	314	626
handbag	292	540
clock	267	204
truck	250	415
bench	235	413
book	230	1.161
backpack	228	371
cell phone	214	262
tv	207	288
couch	195	261
traffic light	191	637
bus	189	285
sink	187	225
cat	184	202
laptop	183	231
knife	181	326
dog	177	218
sandwich	177	98
umbrella	174	413
potted plant	172	343
sports ball	169	263
tennis racket	167	225
bed	163	149
motorcycle	159	371
train	157	190
fork	155	215
pizza	153	285
spoon	153	253
keyboard	153	106
bicycle	149	316
surfboard	149	269
toilet	149	179
baseball bat	146	97
remote	145	283

Figura A.2.3: Conjunto de validación del *dataset* COCO 2017, parte 1

CATEGORY	IMAGES	DETECTIONS
80	117266	860001
##########	##########	##########
tie	145	254
oven	143	115
airplane	143	97
vase	137	277
horse	128	273
skateboard	127	179
hot dog	127	51
bird	125	440
cake	124	316
boat	121	430
refrigerator	121	126
skis	120	241
frisbee	115	84
wine glass	110	343
suitcase	105	303
banana	103	379
giraffe	101	232
baseball glove	100	148
teddy bear	94	191
kite	91	336
elephant	89	255
mouse	88	106
cow	87	380
fire hydrant	86	101
orange	85	287
zebra	85	268
carrot	81	371
apple	76	239
broccoli	71	316
stop sign	69	75
sheep	65	361
donut	62	338
microwave	54	55
bear	49	71
snowboard	49	69
parking meter	37	60
toothbrush	34	57
scissors	28	36
hair drier	9	11
toaster	8	9

Figura A.2.4: Conjunto de validación del *dataset* COCO 2017, parte 2

Apéndice 3 - Manual de usuario de Dettool

Para facilitar la comprensión de cómo utilizar esta aplicación, se recomienda tener en mente las figuras 6.5 y 6.6, donde se exponen sus casos de uso y su aspecto. En primer lugar, el proyecto *ObjectRecognition* debe encontrarse en el directorio raíz de su *Google Drive*, una vez ubicado este allí, debe editarse el fichero *models.json* en *ObjectRecognition/exfiles*, que contiene bloques JSON con la estructura:

- *"modelName"*: {
    - *"gpu_batch_size": <<DefaultGpuBatchSize>>*,
    - *"tpu_batch_size": <<DefaultTpuBatchSize>>*,
    - *"url": "<<UrlToDetectorInTf2DetectionModelZoo>>"*
- }

Cada una de las estructuras de este tipo en *models.json* define un nombre que sirve para identificar un detector, el tamaño de *batch-size* (*mini-batch size*) que por defecto empleará este detector cuando se entrene con una GPU. El tamaño de *batch-size* (*mini-batch*) que por defecto empleará este detector cuando se entrene con una GPU, y la URL (*Uniform Resource Locator*) del archivo comprimido que contiene el modelo, el cual puede encontrarse en la web *TensorFlow 2 Detection Model Zoo*. Un detector que aparezca en models.json de esta manera, podrá ser usado en *Dettool* posteriormente para su configuración, entrenamiento y validación.

Establecidos los modelos con los que se va a trabajar, abrir el *Notebook* de *Dettool* y acceder a *Entorno_de_ejecución/Cambiar_tio_de_entorno_de_ejecución*, donde se indicará en *Acelerador_por_hardware* y *Características_del_entorno_de_ejecución*, el mejor hardware de que se disponga con su subscripción a Colab, para acelerar el entrenamiento y aumentar la capacidad, siendo preferible siempre una TPU a una GPU, y la mayor cantidad de memoria RAM posible para poder ajustar sin problemas el tamaño del *batch-size* (*mini-batch size*).

En este punto, deben ejecutarse secuencialmente todas las celdas del *Notebook* de *Dettool* para completar la carga del programa, que recordemos, será más similar a un programa basado en formularios, que a un *Notebook* al uso. La ejecución hará

que un cuadro de diálogo aparezca para conceder el acceso del programa a *Google Drive*, este debe de aceptarse para proseguir la ejecución. Una vez cargado todo el programa, seleccionar del primer desplegable el detector que se quiere entrenar y validar, acto seguido, poceder a la configuración de los hiperparámetros: *warmup steps*, *base steps*, *warmup learning rate*, *learning rate base*, *momentum*, *batch-size*.

Los cuatro primeros hiperparámetros, *warmup steps*, *base steps*, *warmup learning rate*, *learning rate base*, *y momentum*, se deben a que se está aplicado el algoritmo SGDM para la optimización de la función de error, y el método *cosine decay learning rate*, para favorecer la convergencia del modelo a una solución. El algoritmo SGDM es tal y como se definió en el apartado 2.3.3.4 de esta memoria, y el método de *cosine decay learning rate* para favorecer la convergencia a una solución, implica dividir los pasos (*steps*) del entrenamiento en dos fases, en la primera (*warmup*), se dan una serie de (*steps*), durante los cuales se incrementa la tasa de aprendizaje (*learning rate*), y una segunda fase (*base*), durante la cual se va decrementando poco a poco la tasa de aprendizaje (*learning rate*) (*cosine decay*). El quinto y último hiperparámetro, es el tamaño del *batch* (*mini-batch* size), que dependerá de la capacidad de memoria del acelerador *hardware* de que se disponga con su subscripción a Google Colab.

Configurados todos los hiperparámetros, se pueden incluir adicionalmente opciones de *data augmentation* para conseguir más imágenes para el entrenamiento. Para esto se dispone de una caja de texto en las que indicar estas opciones, pudiendo consultarse las opciones disponibles en el fichero *preprocessor.proto* del repositorio GitHub de la *TensorFlow Object Detection API*. Cada opción de *data augmentation* se define dentro de un bloque *data_augmentation_options*, y cada uno de estos bloques, no requieren de un separador para distinguirse de otro al escribirse en la caja de texto. Un ejemplo de *data augmentation* con dos opciones se definiría con la siguiente lista, donde cada punto representaría una línea de la caja de texto:

- data_augmentation_options {random_horizontal_flip {}}
- data_augmentation_options {random_vertical_flip {}}

A continuación de las opciones de *data augmentation*, se indicará la cantidad de pasos (*steps*) cada cual validar el detector (*validate each*), y el umbral de validación

(*validation thresold*), que es la puntuación mínima que debe obtener el detector para que el *bounding box* de cierto objeto se visualice, por defecto es 0,5.

Después de esto, se elige un *dataset* que haya sido generado con *CDSGenerator* para entrenar al detector, y se presiona el botón *Train & Validate*, lo que dispara en primer lugar el proceso de entrenamiento, y luego el proceso de validación.

Posteriormente, se obtiene un informe de resultados de ambos procesos en la celda del programa *TensorBoard*, en el que se muestran varias métricas de desempeño vistas anteriormente en el apartado 4.2.3, que sirven para poder evaluar distintos aspectos de un detector. A continuación, se muestran distintas figuras de ejemplo que muestran *TensorBoard*, y las mediciones que componen su informe.

La figura A.3.1 muestra el menú principal de *TensorBoard* (*Scalars*), donde se muestran cinco paneles blancos que al desplegarse muestran distintas mediciones relacionadas con el título de cada uno (*Loss*, *learning_rate*, etc).

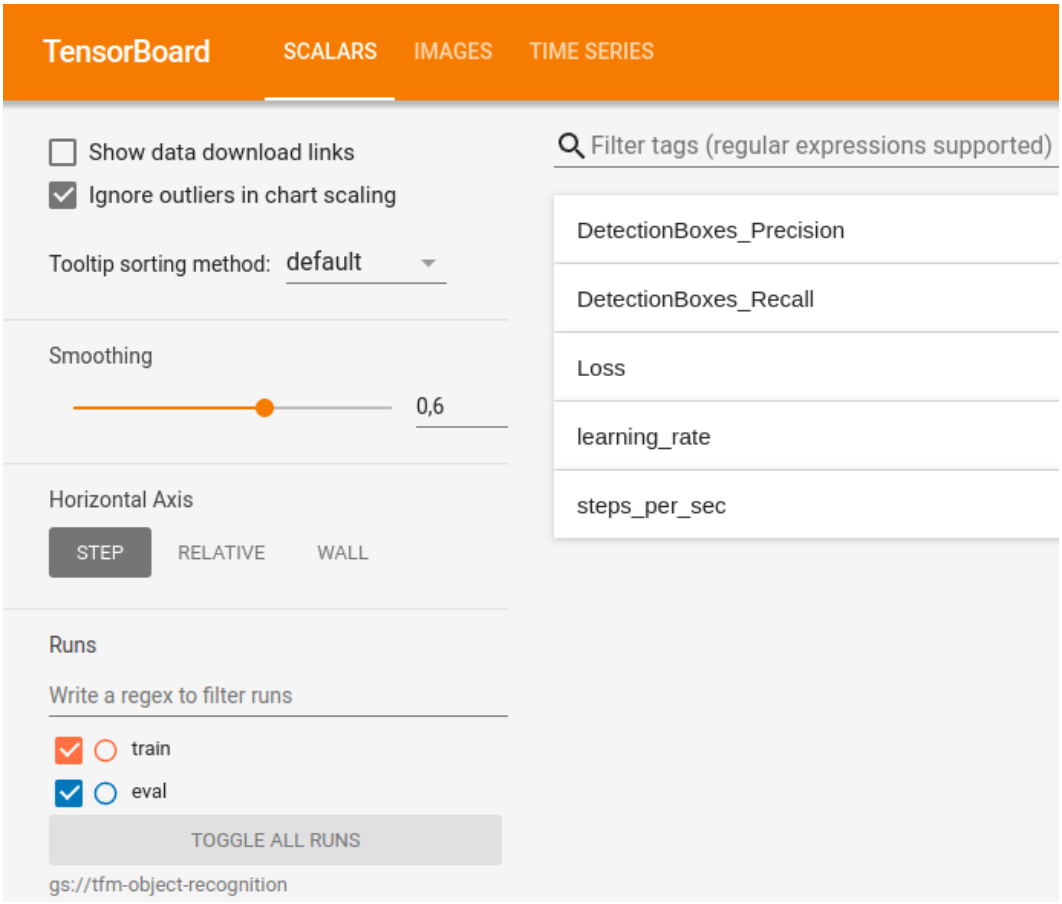

Figura A.3.1: Panel principal de *TensorBoard*

La figura A.3.2 muestra dos gráficas que reflejan la evolución del *learning rate* a lo largo de los pasos (*steps*) del entrenamiento, y la cantidad de pasos (*steps*) por segundo que se dan entre dos instantes de tiempo diferentes a lo largo del proceso. Ambas gráficas muestran en naranja sus resultados porque estos están vinculados al proceso de entrenamiento.

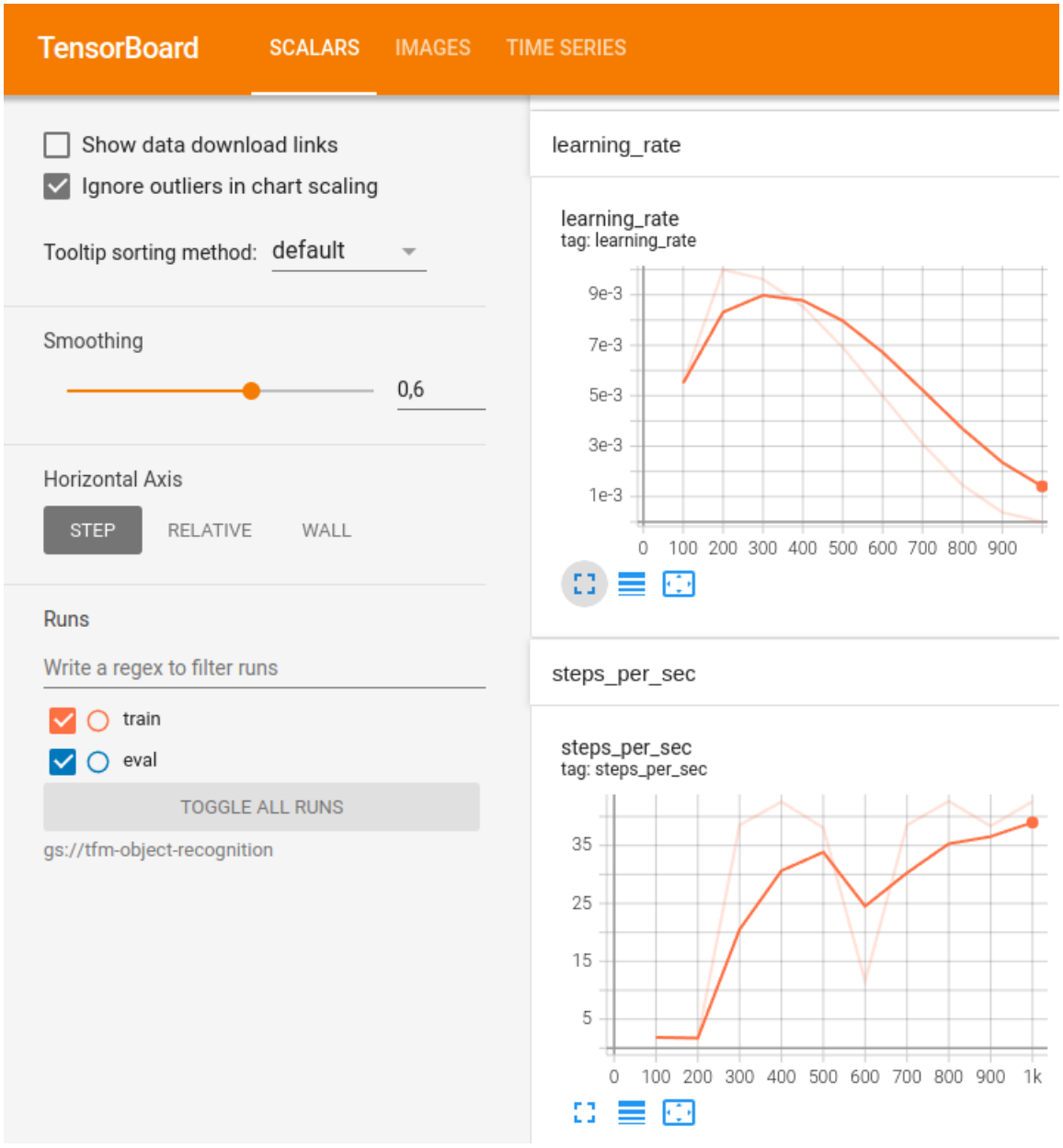

Figura A.3.2: Evolución del *learning rate* y los *steps*

La figura A.3.3, muestra los resultados de la función de pérdida de un detector, que de forma general, se resume brevemente como: $\boldsymbol{Loss} = \boldsymbol{L}_{lassification} + \boldsymbol{L}_{regression}$.

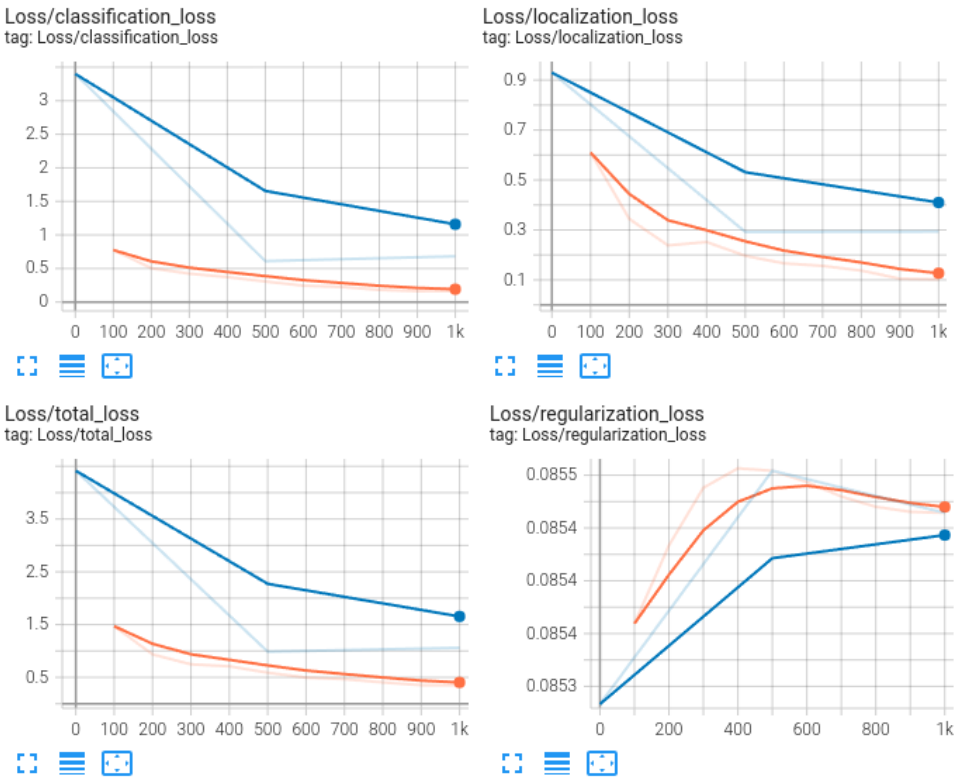

Figura A.3.3: Progreso de la función de pérdida

La figura A.3.4 muestra cómo cambia la métrica *medium Average Recall*, ecuación (4.16), durante el entrenamiento. Esta se ha calculado ateniéndose por un lado a distintas cantidades de detecciones por imágen (1, 10, y 100), y por otro, a distintos tamaños de objetos (*small*: *area*<32²; *medium*: 32²<*area*<96²; *large*: *area*>96²).

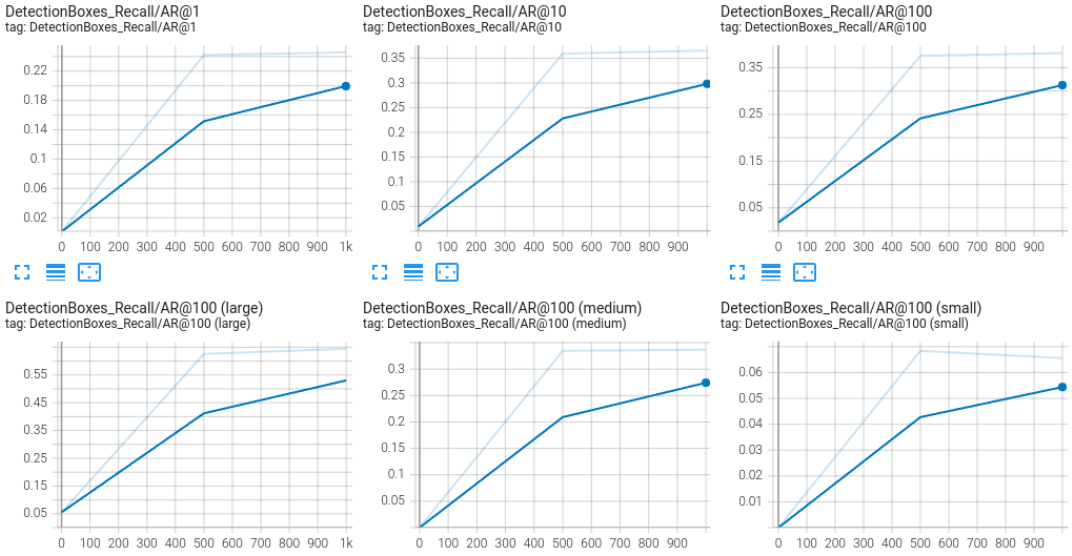

Figura A.3.4: Evolución de *medium Average Recall*

La última métrica que muestra el informe es la evolución de *medium Average Precision*, ecuación (4.14), figura A.3.5. Esta se ha calculado ateniéndose por un lado a distintos valores umbrales de IoU (0.5, 0.75, y 0.5:.5:0.95), y por otro, a distintos tamaños de objetos según se indica en cada uno de los casos (*small*: *area*<32^2; *medium*: 32^2<*area*<96^2; *large*: *area*>96^2).

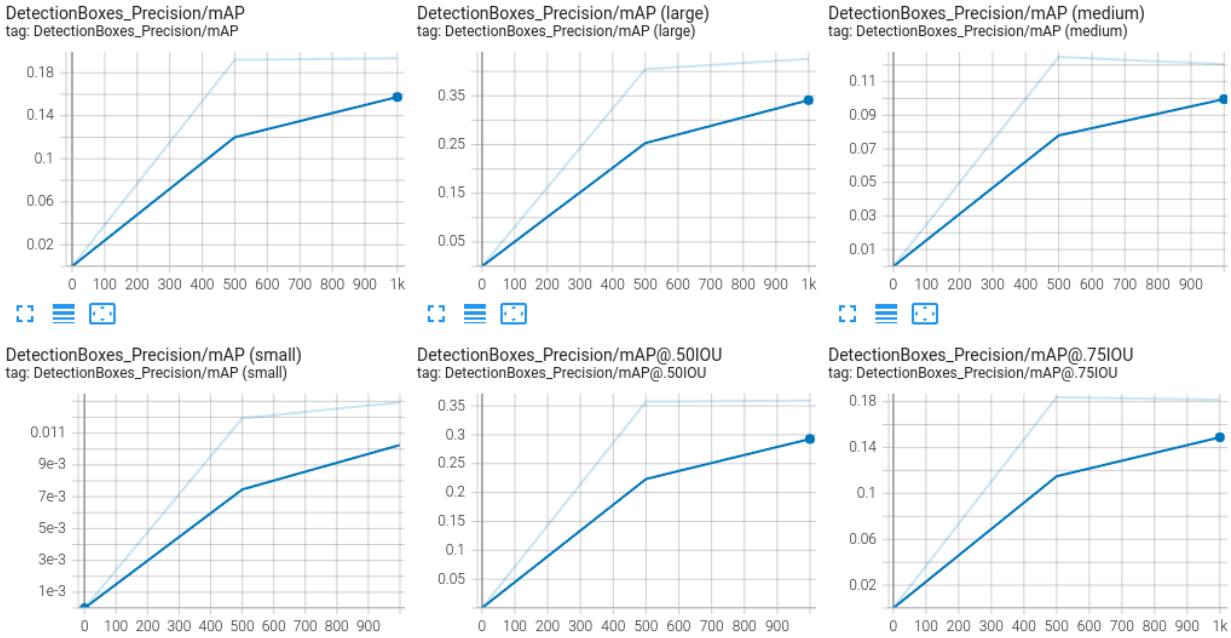

Figura A.3.5: Evolución de *medium Average Precision*

En el menú de *TensorBoard* (*Images*), figura A.3.6, se muestran algunas imágenes que han sido empleadas en el proceso de validación para comprobar de una forma más intuitiva la evolución de la localización y la clasificación de los objetos en imágenes. Cada imagen mostrada se divide en dos, a la izquierda se muestran las predicciones que el detector emite, y a la derecha los *ground truth bounding boxes*. El último menú de *TensorBoard* (*Time Series*), contiene exactamente todo lo que se ha descrito hasta este punto de *TensorBoard*.

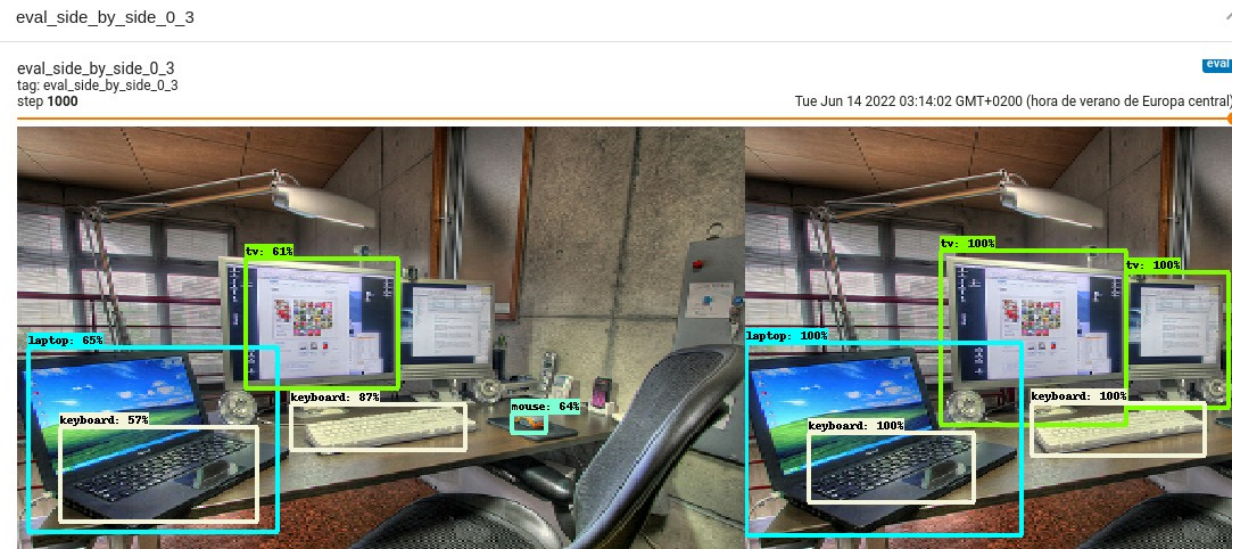

Figura A.3.6: Localización de objetos

En el caso de que los resultados del informe de *TensorBoard* sean de agrado como para querer exportar el detector a nuestro propio *Google Drive*, debe accederse al panel del lateral izquierdo del *Notebook*, y desplegar la carpeta archivos (*files*), que es una representación gráfica del contenido de la carpeta */content* del sistema operativo GNU/Linux de la máquina virtual bajo el *Notebook*. Esta carpeta contiene la estructura de trabajo del programa *Dettool*, que puede verse en la figura A.3.7, en ella, se ha remarcado en rojo el icono de la carpeta archivos (files).

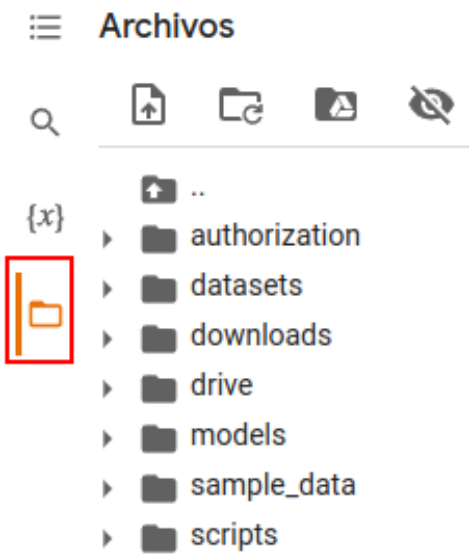

Figura A.3.7: Estructura de trabajo de *Dettool*

Como puede verse en la anterior figura A.3.7, el árbol de trabajo está compuesto por las carpetas:

- sample_data: es una carpeta que siempre es creada por cualquier Notebook al iniciarse, no es una carpeta relevante y, por tanto, podría eliminarse.
- authorization: esta carpeta contiene el fichero *auth.json*, el cual incluye un permiso para poder acceder al almacén de datos (*bucket*) desde el Notebook.
- datasets: esta carpeta contiene los *datasets* que han sido generados con el programa *CDSGenerator*, que deberían estar en *ObjectRecognition/datasets*. De esta ubicación se cargan los archivos necesarios de cada *dataset* para poder entrenar un modelo, estos archivos son: *summary.json*, *pipeline.config, label_map.pbtxt*,, *training.records*, y *validation.records*.
- dowloads: es una carpeta temporal que se emplea para descargar recursos de internet, por ejemplo, la carpeta de la *TensorFlow Object Detection API*, las carpetas comprimidas de los detectores, etc.
- drive: carpeta a través de la cual se accede a su unidad personal de *Google Drive*. Es necesaria para poder exportar los detectores entrenados.
- models: es la carpeta que contiene los detectores de objetos soportados por la *TensorFlow Object Detection API* que un usuario puede optar por utilizar, estos son los descritos anteriormente en el fichero de *models.json*, el cual también se encuentra dentro de esta carpeta. Ahora, dentro de cada carpeta de un detector, podrán encontrarse los siguientes elementos:
  - Carpeta *checkpoints*:
    - Fichero *checkpoint*: un registro que contiene los ficheros de *ckpt-*.
    - Fichero *ckpt-0.data*: contiene las variables de un modelo, es decir, los pesos (*weigths*), sesgos (biases), etc.
    - Fichero *ckpt-0.index*: es una tabla clave-valor, cada clave identifica un tensor, y el valor asociado son metadatos vinculados a este.
  - Carpeta *saved_model*:
    - Carpeta *variables*:
      - Fichero variables.data: es como un fichero *ckpt-0.data*.

        - Fichero variables.index: es como un fichero *ckpt-0.index*.
    - Fichero *saved_model.pb*: es un fichero que almacena tanto un detector de objetos, como las variables (pesos, sesgos, etc.) de este.
    - Fichero *pipeline.config*: contiene tanto la configuración de un modelo, como la de sus procesos de entrenamiento y validación.
- *scripts*: carpeta que contiene varios *scripts* de la *API* de *TensorFlow* para la detección de objetos, estos hacen posible que *Dettool* funcione correctamente.
    - exporter_main_v2.py: es el *script* responsable de poder exportar un modelo en un fichero *saved_model.pb*.
    - model_lib_v2.py: este *script* sustituye al que viene en la *TensorFlow Object Detection API* con su mismo nombre, durante su instalación. Es una modificación del original para realizar el entrenamiento y la validación en serie, que es como se trabaja en los *Notebooks*.
    - model_main_tf2.py: es el *script* que automatiza tanto el proceso de entrenamiento, como el de validación.

Estas carpetas y sus contenidos no deberían alterarse, manteniendo así el correcto funcionamiento de *Dettool*. Continuando con la exportación del modelo, podemos ver en la figura A.3.8 que este guardó dentro de la carpeta *models*, en la de su modelo, y dentro de ella, en otra con su nombre que acaba con los dígitos del día, mes, año, hora, minuto y segundo, en que finalizaron los procesos de entrenamiento, validación, y la exportación del modelo a una carpeta.

El contenido de la carpeta exportada es como puede verse en la figura A.3.8:

- *checkpoint_base*: carpeta con los ficheros de *checkpoint* (*ckpt-*) del entrenamiento original del detector con el *dataset* de COCO.
- *checkpoint_base*: carpeta con los ficheros de *checkpoint* (*ckpt-*) que se han generado durante el reentrenamiento del detector con COCO.
- *train* y *eval*: son carpetas con un fichero de *log* que contiene lo que ocurre durante el proceso de entrenamiento, o validación, gracias a los cuales se puede visualizar el informe gráfico que se ve *TensorBoard*.

- *pipeline_file* y *label_map_file*: carpeta con el *pipeline*, y el *label_map*.
- *record_files*: es la carpeta que contiene los ficheros *training.records*, y *validation.records*, que contienen una representación binarizada de las muestras para los procesos de entrenamiento y validación del modelo.
- *saved_model*: es una carpeta que contiene el *checkpoint* (las variables) del modelo, un archivo *saved_model.pb* que contiene tanto el modelo como sus variables, una copia del fichero *pipeline.config*, y una carpeta que contiene los ficheros *variable*, que son como los de *checkpoint*.

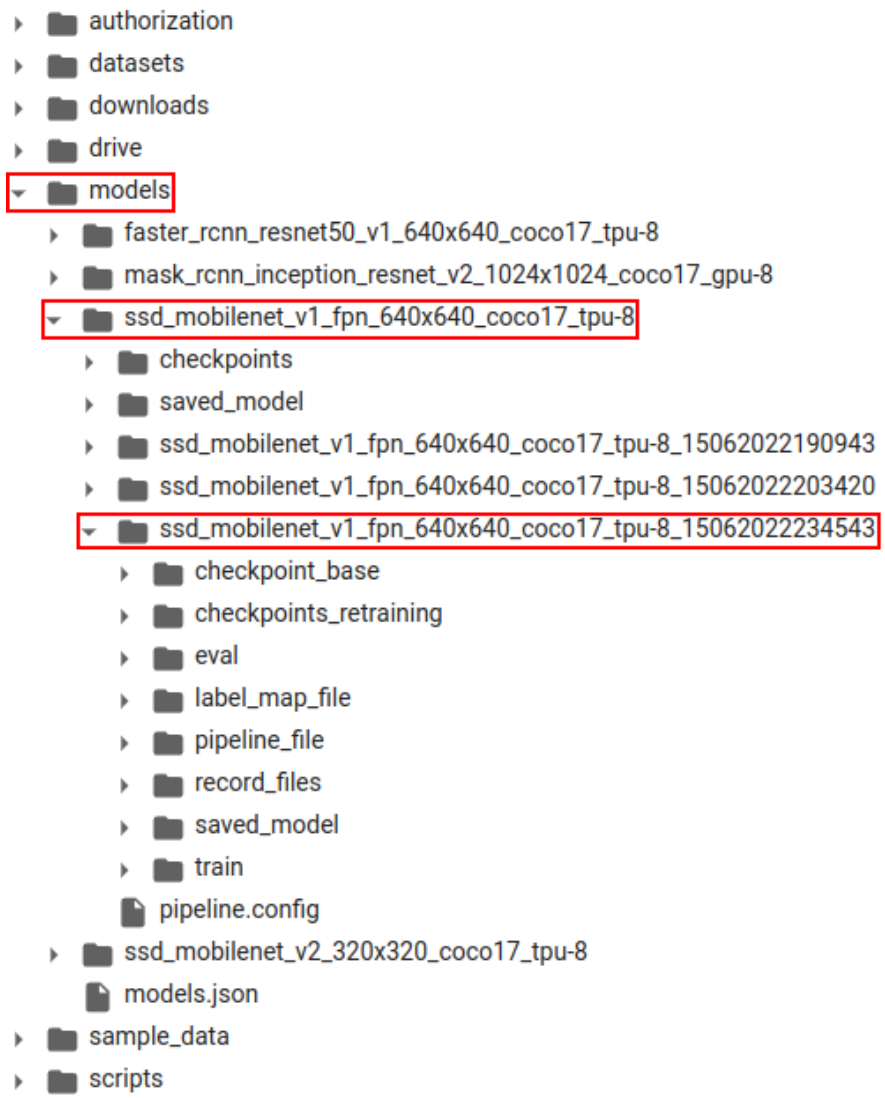

Figura A.3.8: Ubicación de un modelo entrenado y validado con *Dettool*

Finalmente, para exportar, arrastrar el modelo a la carpeta de *exports* como se puede ver en la imagen de la carpeta A.3.9.

Figura A.3.9: Exportación de un detector a *Google Drive*

Apéndice 4 - Descarga e instalación del proyecto

El proyecto aquí planteado puede ser descargado a través del siguiente enlace a su repositorio de GitHub: https://github.com/ManuelGMS/ObjectRecognition

Considerandos:

- No es necesario realizar una instalación para poder utilizar la plataforma aquí desarrollada, solo posicionar la carpeta del proyecto en la raíz de su *Google Drive* y ejecutar los *Notebooks* con *Google Chrome*. Sin embargo, sí hay que realizar ciertas configuraciones para utilizar el programa *Dettool*, ya que este recurre a un *bucket* personal de *Google Cloud Storage*, por lo que es preciso cambiar dicho *bucket* por el suyo. Para ello se deben seguir estos pasos:
  1. Crear un *bucket* propio en *Google Cloud Storage* agregando al servidor de TPU vinculado al *Notebook* los permisos: lector de *buckets* heredados de almacenamiento, propietario de objetos heredados de almacenamiento, administrador de almacenamiento, propietario de *buckets* heredados de almacenamiento, y lector de objetos heredados del almacenamiento.
  2. Obtener para el *bucket* recién creado en *Google Cloud Storage* el fichero JSON para el acceso autentificado a este *bucket*. Para esto, debe acceder a *Cuentas de Servicio* en *IAM y Administración*, una vez allí, seleccione el nombre de su *bucket* y seleccione la pestaña *Claves*, y dentro de ella, en el desplegable *Agregar Clave* seleccionar *Crear Clave Nueva*, esto abrirá un cuadro de diálogo de donde se puede descargar el fichero JSON con el permiso de acceso. El fichero obtenido debe reemplazar a auth.json en la ruta ObjectRecognition\exfiles dentro del proyecto de ObjectRecognition.
  3. Configurar el *Notebook Dettool* de la siguiente manera: en la celda con el nombre *Autorización para el acceso a un bucket de Google Cloud Storage*, cambiar el valor de la variable *bucketName* al nombre de su *bucket*.
- La plataforma se ha desarrollado con la versión de *Google Colab Plus* debido a las limitaciones del hardware de la versión gratuita de *Colab*, y aunque puede utilizarse con esta, es posible que disminuya muy notablemente la capacidad para realizar el entrenamiento y la validación de los detectores.