Estudio de métodos de detección de objetos mediante técnicas de Aprendizaje Profundo

MÁSTER EN INGENIERÍA DE SISTEMAS Y DE CONTROL

TRABAJO FIN DE MÁSTER

Alberto Vaquero Vega

Curso: 2022-2023

Convocatoria: Septiembre 2023

Director: Gonzalo Pajares Martinsanz

Máster en Ingeniería de Sistemas y de Control

Estudio de métodos de detección de objetos mediante técnicas de Aprendizaje Profundo

Proyecto de tipo B: específico propuesto por el alumno

Autor: Alberto Vaquero Vega

Director: Gonzalo Pajares Martinsanz

Convocatoria: Septiembre 2023

Autorización

Autorizamos a la Universidad Complutense y a la UNED a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a sus autores, tanto la memoria de este Trabajo Fin de Máster, como el código, la documentación y/o el prototipo desarrollado.

Firmado: Alberto Vaquero Vega

Resumen

La idea que motiva este proyecto es la resolución del problema de detección de objetos usando técnicas avanzadas de Aprendizaje Profundo mediante el uso de detectores basados en redes neuronales convolucionales. Para ello, se realiza un estudio de qué son este tipo de redes neuronales y, sobre todo, qué parámetros o características las definen y diferencian entre sí. Posteriormente, se realiza el mismo estudio centrado en cinco tipos ampliamente conocidos de detectores basados en redes neuronales, como son: RCNN, Faster-RCNN, SSD, YOLOV3 y YOLOV4. Siempre buscando las características que los definen, las cuales se muestran en la realidad con distintos rendimientos.

De cara a la elaboración práctica se presenta un estudio de las posibilidades base y se desarrolla, a partir de la plataforma de cálculo Matlab nuevo código, junto con un *dataset* de amplio conocimiento general, como es el Microsoft COCO. Además, todo esto se lleva a cabo usando recursos tanto locales como de plataformas en la nube, en este caso Amazon AWS.

De esta forma, estableciendo una serie de premisas y reglas para el entrenamiento de los diferentes detectores se generaron 38 modelos o versiones para la totalidad de los cinco tipos de detectores distintos usados, obteniendo así una media de 7-8 por cada uno. Todo esto, se lleva a cabo, obviamente, realizando el mismo número de entrenamientos con un elevadísimo coste computacional y tiempos. Se logra, de esta forma, obtener tanto una comparativa entre el rendimiento de los distintos tipos de detectores, como en cada uno de estos según la red convolucional base usadas, el tipo de algoritmo de optimización usado o la tasa de entrenamiento entre otras modificaciones.

Dada la naturaleza de los detectores y el coste computacional, el estudio de su comportamiento se ha llevado sobre un determinado tipo de objeto presente en las imágenes del *dataset*, como es la señal de tráfico *stop*.

Los resultados obtenidos, permiten establecer una jerarquía de comportamientos de los detectores según sus distintas variantes y parámetros de configuración.

Palabras clave

Detección de objetos, redes neuronales convolucionales, aprendizaje profundo, YOLO, SSD, RCNN, Faster RCNN.

Índice de contenido

Índice de imágenes

Índice de tablas

1. Introducción

El objeto principal y último de este Proyecto Fin de Máster es la realización de un estudio y análisis sobre distintos detectores de objetos en imágenes, pero profundizando no sólo en las diferencias entre los distintos tipos, sino también entre sí, desde el punto de vista de la implementación y el rendimiento. Para llegar a este fin, se han establecido una serie de metas intermedias a alcanzar que se detallan en esta primera sección de la memoria. Igualmente, la realización y establecimiento de un marco comparativo tanto dentro de los propios detectores, como en el software y plataformas de computación creó otro objetivo en sí mismo.

1.1. Antecedentes

La detección de objetos es para los humanos un proceso trivial, pudiendo identificar y localizar a una persona o un objeto incluso aunque sólo se le vea parcialmente, lo veamos muy lejos o incluso cuando casi no lo veamos ya que en condiciones de iluminación adversas aparezca en oscuridad, por ejemplo. Sin embargo, realizar esto en el campo de la visión por computador es extremadamente complejo. Por este motivo, hasta la aparición y desarrollo de las primeras técnicas de Aprendizaje Profundo (AP) o *Deep Learning* (DP), según la terminología científica internacional, su realización por medio de algoritmos era muy limitada a casos de objetos muy concretos, en condiciones particulares. Pero gracias a la introducción del AP con redes neuronales, se abrió la posibilidad de crear arquitecturas de redes que pudiesen aprender y asemejarse a la detección de objetos humana.

A su vez, este desarrollo reciente, hizo que la detección de objetos despegase como un campo en sí mismo, con suma importancia dentro del gran compendio de lo que llamamos visión por computador. Se convirtió en un campo sobre el cual ha proliferado un mayor número de estudios y avances en los últimos tiempos. Tal es así, que todos los años se establecen concursos donde se ponen a prueba detectores desarrollados con bases de datos conocidas en función de una precisión y unos tiempos. Por tanto, hay una tendencia a un avance progresivo en este campo, que, en última medida, está provocando la aparición de nuevos desarrollos, en gran parte, por su gran cantidad de aplicaciones al análisis de vídeo y de imágenes en tiempo real.

Sin embargo, este rápido avance ha provocado un campo con múltiples tipos de detectores y que con diferentes variaciones interiores evolucionan hacia otros detectores que forman a su vez arquitecturas distintas y con nombre propio. Y que, además, se aplican y entrenan sobre múltiples bases de datos de imágenes de conocimiento público. Por tanto, los ejemplos y bibliografías a este respecto son muy numerosos y fáciles de encontrar. Ejemplo de ello sería [1].

Pero estos ejemplos profundizan más en la comparación entre distintos tipos de detectores por su arquitectura que en distintos tipos de detectores por variaciones internas como

puede ser modificaciones de las redes convolucionales base que usan. Ejemplos de esto son los que se proporcionan en las referencias [2], [3], [4].

En algunos casos, existe bibliografía de comparación entre tipos de redes y variaciones, pero se aplican sólo a la clasificación con redes convolucionales y no a la detección de objetos, tal es el caso del trabajo mostrado en [5].

Por tanto, los casos donde se aplica esta comparativa, pero con variaciones, no sólo entre detectores, sino dentro de ellos, variando la red convolucional de la arquitectura base o partes del entrenamiento, son muy escasas y, cuando las hay, se limitan a una comparativa básica en torno a uno o dos tipos de detectores. Un ejemplo de esto sería el trabajo de la referencia [6].

1.2. Objetivos

Los objetivos marcados para la realización de este trabajo son los siguientes:

- Estudio y comprensión de las redes neuronales convolucionales.
- Estudio y comprensión de las arquitecturas para la detección de objetos.
- Integración de una base de datos de imágenes en una plataforma de computación para la creación de diversos detectores, desarrollados en cinco tipos: RCNN, Faster-RCNN, SSD, YOLOV3 y YOLOV4.
- Desarrollo, implementación y análisis de detectores realizando variaciones sobre los cinco tipos anteriores.
- Toma de resultados y conclusiones.

1.3. Tareas realizadas

Para lograr los objetivos pretendidos en la realización de este Proyecto se tuvo que disgregar y analizar cada uno de ellos en diferentes tareas. Al partir desde un enfoque teórico que diese un marco sólido para la parte práctica, las tareas planteadas se realizaron en este orden:

- Lectura técnica sobre redes neuronales convolucionales y los mecanismos internos de generación de resultados.
- Lectura, análisis y síntesis acerca de algunas de las redes convolucionales más comunes, las cuales se usarán en la parte práctica del proyecto.
- Lectura, análisis y síntesis de la detección de objetos, así como las principales arquitecturas existentes.
- Lectura y análisis del significado de las métricas comúnmente usadas para el análisis de los detectores de objetos.
- Estudio, generación de código y creación de una comunicación entre Matlab y Microsoft COCO, así como la integración de funciones de Microsoft COCO en Matlab.

- Estudio y creación de una comunicación entre Matlab y Amazon AWS para la mejora del entrenamiento de toda la cantidad de detectores de objetos.
- Modificación y generación de código nuevo en Matlab para la adecuación de las imágenes contenidas en COCO *dataset* y que puedan ser completamente aplicadas para la generación de detectores sin problemas de compilación o similar.
- Modificación y generación de código nuevo para la ejecución de entrenamiento de los cinco tipos de detectores usados con Microsoft COCO y apoyado en ordenador local y recursos en la nube por medio de Amazon AWS.
- Creación de un marco común para la realización de variantes de los cinco tipos de detectores creados y permitir una cierta comparación final. Ampliación a variaciones en casos particulares que se quieren comprobar en algunos de esos tipos de detectores.
- Realización de los entrenamientos y toma de resultados.
- Análisis de resultados intentando buscar en la base teórica el porqué de ellos.

Por último, no menos importante, está la tarea de creación de la presente memoria para plasmar el resultado de todos estos puntos anteriores.

1.4. Organización de la memoria

La presente memoria se realizó atendiendo a los objetivos y tareas jerárquicas que se llevaron a cabo en este trabajo. Las secciones que componen esta memoria se organizan como sigue:

- En la primera sección se realiza una breve introducción de la memoria en cuanto al marco técnico alrededor de la creación de este trabajo, así como la constatación de los objetivos y tareas realizadas.
- En la segunda sección se incluye un resumen de la teoría de redes convoluciones, empezando por una introducción a qué son, qué las diferencian de otros modelos clásicos como el perceptrón multicapa y pasando a un glosario técnico de términos comúnmente usados, el cual sirve para mostrar de forma ágil qué son y cómo afectan a las redes. Finalmente, se indican algunos usos de redes neuronales convolucionales.
- En la tercera sección se incluye un resumen de qué es la detección de objetos y de las arquitecturas usadas en la parte práctica de la memoria. Igualmente, se detalla con ejemplos sencillos las métricas disponibles para la comparación entre detectores.
- En la cuarta sección se describen las herramientas usadas tanto de computación, como de base de datos y de ejecución.
- En la quinta sección se detallan los resultados obtenidos con los cinco tipos de detectores usados y, posteriormente, se muestran los de las sucesivas variaciones dentro de ellas. Igualmente, primero se explica el marco base para usar entre detectores creados y permitir la comparación en la medida de lo posible.

- En la sexta sección se incluyen las conclusiones obtenidas a partir de los resultados y se explican las posibilidades de ampliación futura de este Proyecto.
- Finalmente, en la última sección, número siete, se incluye toda la bibliografía consultada.

2. Teoría de las redes CNN

El primer punto en el camino y progreso de esta memoria es el conocimiento de qué son las redes neuronales convolucionales, (*Convolutional Neural Networks*, CNN), así como sus particularidades teóricas y características que definen que unas arquitecturas sean distintas de otras.

2.1. Resumen de las redes CNN

En esta memoria se parte del punto del conocimiento básico de lo que son las redes neuronales y el AP o DL. Dentro de la multitud de redes neuronales existentes en nuestros días, las CNN son un tipo de redes que, debido a su versatilidad y variaciones en su aplicación y formas para el procesamiento de datos (imágenes, texto, sonido o vídeos), representan una de las más importantes y exitosas del momento.

Dicho éxito reside en su excelente aplicación en el campo del mencionado AP ya que permiten la extracción automática de las características de los datos (imágenes, texto, sonido o vídeos) analizados [7].

En sí mismas, yendo a sus características básicas, se trata de redes multicapa feed-forward, con pesos adaptativos y memoria estática (la salida depende de la entrada). La particularidad respecto a las redes básicas del conocido perceptrón y redes basadas en él, como puede ser el perceptrón multicapa, estriba en el uso de convoluciones en lugar de la multiplicación de matrices.

Las redes CNN, son aquellas compuestas de al menos una o más capas convolucionales seguidas de una o más capas *Fully Connected* (FC) [8], estableciéndose una relación entre algunas de las capas por medio de los llamados núcleos de convolución. Visto desde el entrenamiento, se traduciría en el uso de las convoluciones aritméticas transpuestas con el fin de aplicarla en el sentido inverso de la directa [9].

Esta característica es una de las que las diferencian del perceptrón multicapa en una red *feedforward* en el que existe una relación de multiplicación de matrices entre la entrada y la salida o, lo que es lo mismo, la multiplicación con las inversas de las matrices [10].

Al final, esta capa o capas de convolución utilizadas en las redes CNN crean unas propiedades óptimas para su uso en el campo de la detección de características de los datos:

- Se produce una reducción en el número de parámetros requeridos para la definición de la red en comparación con las redes estándar FC como el perceptrón multicapa en la que la unidad mínima de entrada sería el píxel (por lo que, rápidamente los parámetros de la red se hacen inmanejables con imágenes de resoluciones medias). Esto mejora la velocidad y aprendizaje y la memoria necesaria para guardar los valores. Se puede apreciar fácilmente viendo los esquemas básicos de estas dos redes neuronales en las Figuras 1 y 2.

- Los parámetros de la red en lugar de otorgar solución de un óptimo global, como el perceptrón multicapa por ejemplo, extraen la información de forma que se ajusta sucesivamente en las convoluciones de esas capas concretas, que al final extraen, por ejemplo, bordes, regiones, o puntos de interés, es decir, estructuras de bajo nivel [7].
- Equivarianza de translación. Las redes CNN poseen la virtud de poder detectar las características buscadas, aunque se encuentren en lugares diferentes en las imágenes. Por ejemplo, tomando el modelo de la Figura 3, si hay un objeto en la imagen y este se mueve a otra posición, la salida en ambas imágenes será la misma. Obviamente, esto no es tan sencillo y el precio a pagar es el disponer de redes entrenadas con elevados números de imágenes que puedan realizar correctamente esa extracción de datos aun teniendo movimientos, rotaciones u otras configuraciones. Esto es de hecho lo que se busca en los conjuntos de entrenamiento con lo que se denomina *data-augmentation*. Esto mismo, aplicado a una red convencional como el perceptrón no es posible, ya que no son invariantes a la translación (aprendería por ejemplo a detectar ese objeto en esa posición de la imagen).
- Generalización. Si bien los algoritmos de extracción de características de los datos no son exclusivos de las redes neuronales (y se pueden encontrar algoritmos tales como SIFT, SURF, BRIEF u ORB dedicados a tal propósito) son en estas en las que se suelen obtener mejores generalizaciones y resultados. Esto es debido al propio poder de aprendizaje de las redes neuronales en cuanto a la extracción de las características de los datos y al propósito final de los algoritmos anteriormente mencionados que hace crear funciones muy específicas para un fin particular definido [11], [12].
- Estructuración jerárquica. En el caso en el que la red CNN esté compuesta por sucesivas convoluciones, el aprendizaje será secuencial o jerárquico [13]. Las características de más bajo nivel (bordes o texturas) se aprenden en las primeras capas, mientras que en las sucesivas se puede extraer y aprender características o patrones de más alto nivel y, por tanto, más abstractos.

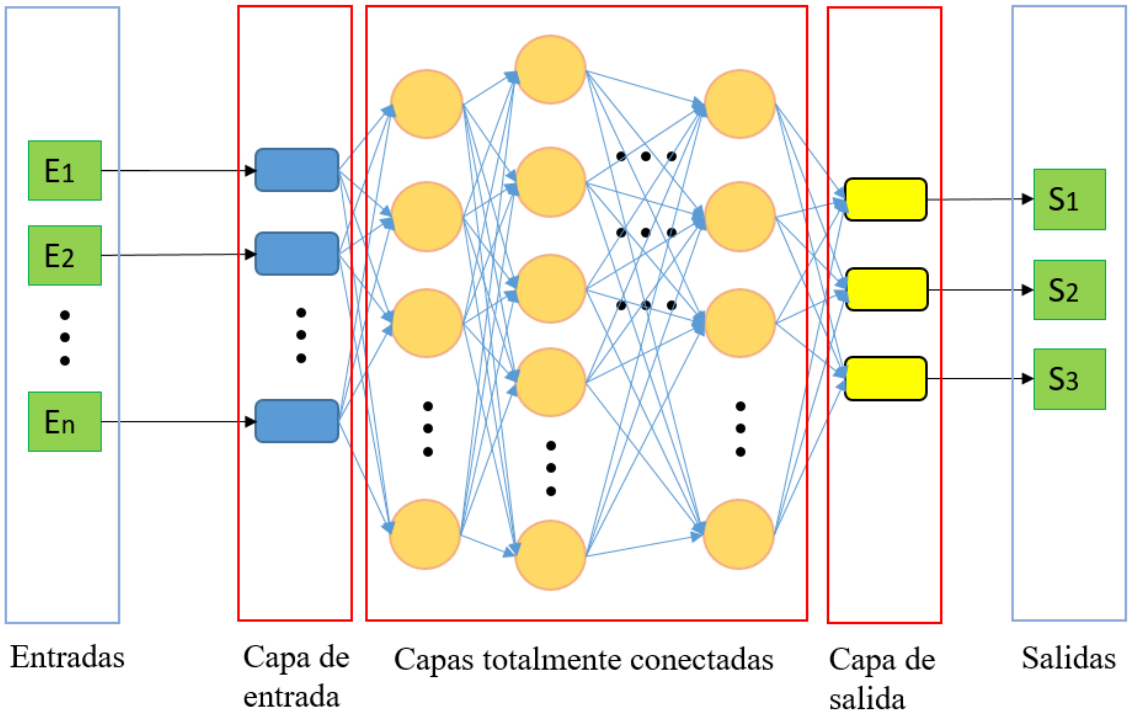

Figura 1. Esquema básico de la red perceptrón multicapa [14]

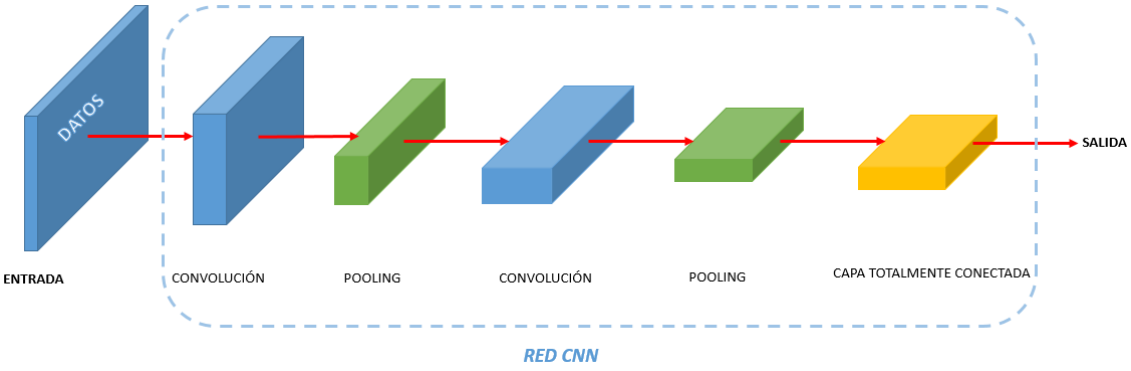

Figura 2. Esquema básico de una red neuronal convolucional (CNN) [15]

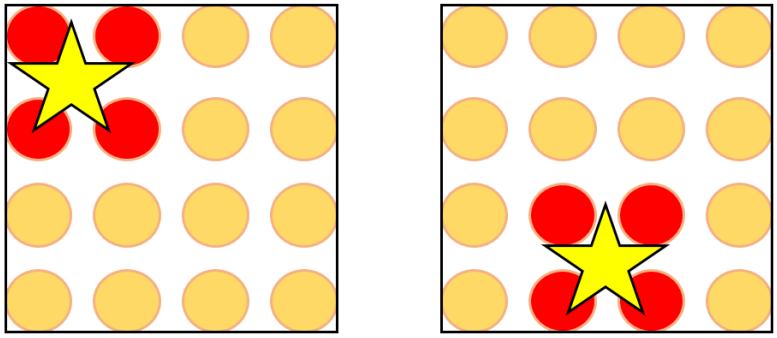

Figura 3. Ejemplo teórico de la forma en que una red perceptrón multicapa activaría distintas neuronas según la posición del objeto

- Capa de entrada: la entrada serán los píxeles de la imagen, en sus dimensiones alto × ancho × profundidad. Esta profundidad será 1 si es imagen de grises o 3 si en color.
- Capa de Convolución: en esta se procesa la salida de la capa de entrada en forma local (en los píxeles cercanos), calculando el producto escalar entre su valor (por ejemplo, valor de intensidad del píxel entre 0 y 255 para grises) y una máscara (también llamada filtro o *kernel*) con unos valores o parámetros que se ajustan en los entrenamientos. Su tamaño de salida depende de varios factores propios como el *stride*, *padding*, entre otros (ver sección 2.2.).
- Capa de activación: en ella se aplica una función de activación de la salida de la convolución (ver sección 2.2.).
- Capa "*Pooling*": en ella se sustituye la salida de la red por una operación estadística que tiene en cuenta los otros valores de salidas cercanas. Es por tanto una agrupación de valores de salida que se usa en casi todas las redes CNN, ya que crea al final una invarianza local ante pequeñas traslaciones espaciales. Realiza, por tanto, una reducción de las dimensiones de la capa de convolución o activación (si la hay) en el ancho y el alto, pero no en la profundidad. Operaciones típicas de este tipo son *Max pooling* (que se refiere a la elección del máximo) o *Average Pooling* (promediado)
- Capa totalmente conectada, del tipo FC: es una disposición de neuronas al modo clásico donde todas las entradas a ellas y las salidas entre sí están interconexionadas. Es decir, los parámetros de las neuronas que la forman se ajustan teniendo en cuenta la totalidad de la entrada, tomando obviamente valores distintos en función de la salida final. Se conecta normalmente con la última capa de *pooling* y finaliza con una capa que contiene el número de salidas correspondientes al número de clases a clasificar [16].

A modo resumido para una mejor comprensión, en la *Tabla 1* se pueden ver las diferencias entre las redes convolucionales aplicadas a imágenes con respecto al modelo común del perceptrón multicapa.

	Perceptrón multicapa	**Red CNN**
Capa de entrada	Los datos son las características que se analizan. Por ejemplo, ancho, alto, etc.	Píxeles de la imagen que serán 1 en blanco y negro y 3 si es color.
Capas intermedias u ocultas	Hay un número de neuronas seleccionadas totalmente conectadas entre sí.	Hay capas de convolución, activación y/o pooling que actúan de forma local sobre valores de la capa de entrada o salidas entre sí.
Capas de salida	El número de neuronas de salida es la cantidad que se quiere analizar.	Antes de la salida se aplican capas totalmente conectadas para terminar de extraer características y clasificar, por ejemplo. El número de neuronas en la última capa de salida es el número de clases a clasificar.
Forma de aprendizaje	Supervisado	Supervisado

Conexionado entre neuronas	Todas las neuronas están completamente conectadas entre una capa y la siguiente. Por tanto, se ajustan todas las variables o pesos de las neuronas	Centrado en la parte de capas intermedias, con convolución, activación y pooling, son conexiones entre aplicaciones de las máscaras o funciones en forma local. Se ajustan sólo los parámetros de las máscaras o *kernels*.
Salida de las capas	No representa nada en sí.	Son mapas de características de distintas formas de las imágenes a bajo nivel (líneas, regiones, curvas o alto nivel en las capas más profundas.
Aprendizaje	Se usa retropropagación para ajustar todos los pesos o parámetros de las neuronas.	Se utiliza para ajustar los parámetros de los *kernels*.

Tabla 1. Resumen de las diferencias principales entre redes de perceptrón multicapa y redes CNN [16]

De forma gráfica, se pueden ver algunas de estas diferencias en la Figura 4:

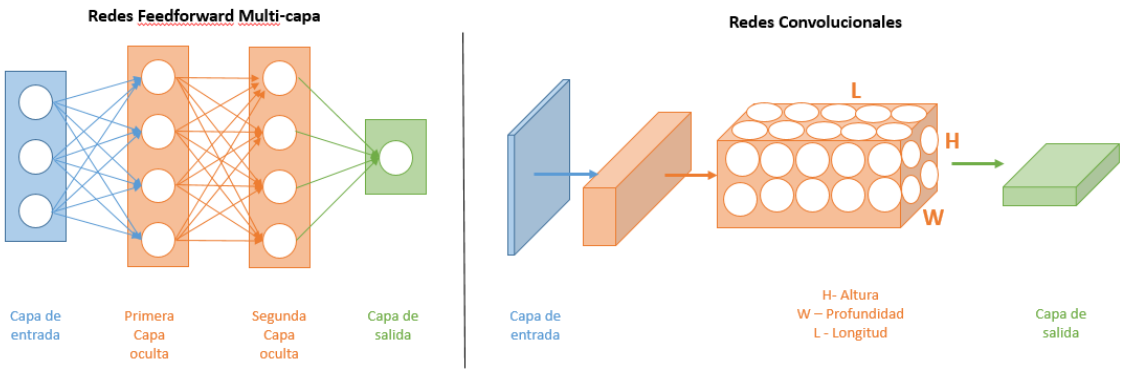

Figura 4. Imagen comparativa del esquema de conexciones en una red de perceptrón multicapa y en una red convolucional [13]

2.2. Glosario básico en las CNN

A continuación, se muestra un breve glosario con el fin de mostrar la base de ciertos conceptos y operaciones comúnmente usadas en la detección de objetos de una forma más o menos instructiva y organizada. En esta explicación se sigue el modelo AlexNet que se describe en la Tabla 2 con referencia a la Figura 5.

2.2.1. Capa de convolución

En estas capas se crea la operación matemática de convolución donde un *kernel* o máscara o filtro de dimensión $n \times n$ se desplaza a lo largo de la imagen original.

No obstante, en lugar de realizarlo con un solo *kernel*, se realiza con un conjunto de ellos, lo que se conoce como filtro. Por lo tanto, el resultado de cada capa de convolución es un tamaño reducido de resolución, pero con tantas matrices como *kernels* han sido usados en la capa precedente. Por ejemplo, como se puede ver en la Figura 5, donde se muestra

de forma esquemática la arquitectura AlexNet y, más en detalle, en la Tabla 2, donde se muestra con detenimiento la implementación práctica de dicha red en Matlab, en la primera capa de convolución se han usado kernels de 11×11×3 (estos son los pesos para la CNN) con *padding* 0 (el *padding* es una técnica para agregar píxeles en la periferia de la imagen y evitar que el tamaño se reduzca excesivamente) y *stride* 4,4 (es el número de posiciones en filas y columnas que se mueve el *kernel* en su desplazamiento a lo largo de la imagen). Igualmente se han usado un conjunto de 96 *kernels*, aplicados una única vez (será el sesgo para la CNN). De esta forma, por cada *kernel* se obtendrá una matriz de 55×55 ((227-11)/4+1=55). Al usarse 96 *kernels*, tendremos un array de 55×55×96, que coincidirá con las neuronas pertenecientes a esta capa. Igualmente, de esta misma forma, se puede decir que tendremos 96 imágenes filtradas con respecto a la original.

Las capas de convolución en las redes CNN suelen ser varias, de manera que la salida de una capa es la entrada en la siguiente (aplicando funciones de activación o *pooling* en medio en muchos casos). Esto es así porque cada capa de convolución viene a extraer distintas propiedades de detalle de las imágenes tras la aplicación de los filtros, tales como líneas, curvas, etc, en las primeras capas. Su base, la aplicación de máscaras o *kernels* sobre imágenes, es, de hecho, uno de los métodos de extracción de bordes y regiones del tratamiento de imágenes por visión por computador.

Cabe destacar también que, al principio, antes de realizar el entrenamiento de una red CNN, los valores o parámetros de los filtros se establecen aleatoriamente y, posteriormente, a lo largo del entrenamiento, se van modificando y ajustando por medio de retropropagación (*backpropagation*). Por este motivo y el de la aplicación secuencial de convoluciones, es muy extraño que se tenga al final dos filtros que sean exactamente iguales (a menos que se haya seleccionado un gran número de filtros).

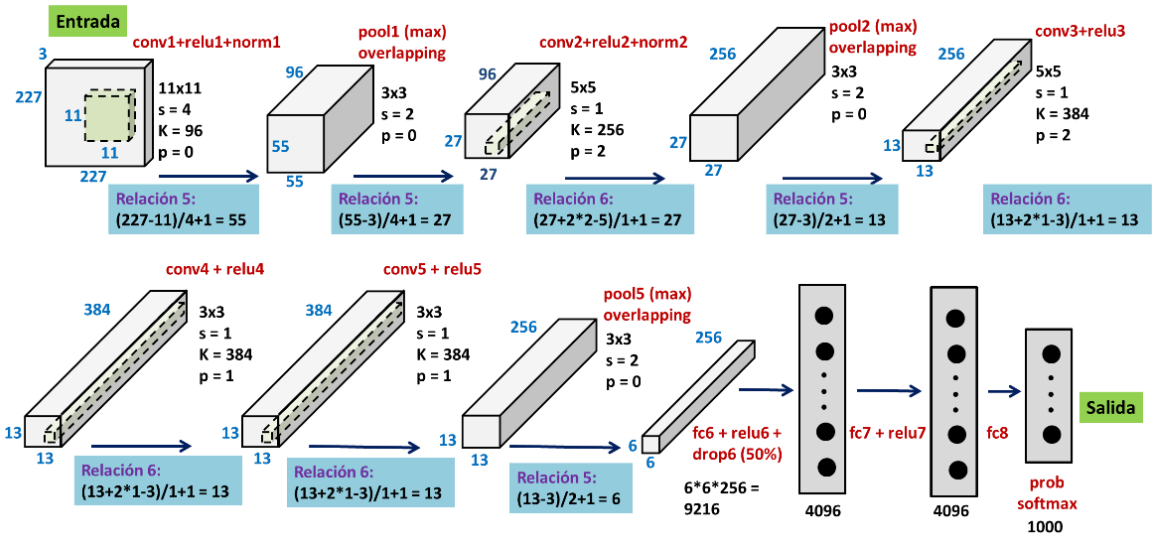

Figura 5. Estructura de capas de AlexNet [9]

	Nombre de capa en Matlab	**Tipo de capa**	**Tamaño salida de capa**	**Parámetros de la capa**
1	data 227x227x3 imágenes con normalización de centro cero	Entrada de imagen	227x227x3	
2	conv1 96 11x11x3 convoluciones con stride [4 4] and padding [0 0 0 0]	Convolución	55x55x96	11x11x3x96 1x1x96
3	relu1 ReLU	ReLU	55x55x96	
4	norm1 normalización de canal cruzado con 5 canales por elemento	Normalización de canal cruzado	55x55x96	
5	pool1 3x3 max pooling con stride [2 2] and padding [0 0 0 0]	Max Pooling	27x27x96	
6	conv2 2 grupos de 128 5x5x48 convoluciones con stride [1 1] and padding [2 2 2 2]	Convolución grupal	27x27x256	5x5x48x128x 128 1x1x128x2
7	relu2 ReLU	ReLU	27x27x256	
8	norm2 normalización de canal cruzado con 5 canales por element	Normalización de canal cruzado	27x27x256	
9	pool2 3x3 max pooling con stride [2 2] y padding [0 0 0 0]	Max Pooling	13x13x256	
10	conv3 384 3x3x256 convoluciones con stride [1 1] and padding [1 1 1 1]	Convolución	13x13x384	3x3x256x384 1x1x384
11	relu3 ReLU	ReLlJ	13x13x384	
12	conv4 2 grupos de 192 3x3x192 convoluciones con stride [1 1] and padding [2 2 2 2]	Convolución grupal	13x13x384	113x13x192x 192 1x1x192x2
13	relu4 ReLU	ReLlJ	13x13x384	
14	conv5 2 grupos de 192 3x3x192 convoluciones con stride [1 1] and padding [2 2 2 2]	Convolución grupal	13x13x384	113x13x192x 192 1x1x192x2
15	relu5 ReLU	ReLlJ	13x13x384	
16	pool5 3x3 max pooling con stride [2 2] y padding [0 0 0 0]	Max Pooling	6x6x256	
17	fc6 4096 capa totalmente conectada	Totalmente conectada	1x1x4096	4096x9216 4096x1

18	relu6 ReLU	ReLU	1x1x4096	
19	drop6 50% *dropout*	*Dropout*	1x1x4096	
20	fc7 4096 capa totalmente conectada	Totalmente conectada	1x1x4096	4096x096 4096x1
21	relu7 ReLU	ReLlJ	1x1x4096	
22	drop7 50% *dropout*	*Dropout*	1x1x4096	
23	fc8 1000 capa totalmente conectada	Totalmente conectada	1x1x1000	1000x4096 1000x1
24	prob softmax	Softmax	1x1x1000	
25	output crossentropyex con 'trench' y 999 otras clases	Salida de clasificación		

Tabla 2. Capas de la red pre-entrenada AlexNet implementada en Matlab

2.2.2. Mapa de características

Es la salida de una capa de convolución, la cual por ejemplo en una CNN de clasificación de imágenes, resalta la/las características del objeto u objetos en cuestión una vez es entrenada correctamente.

2.2.3. Tensor

Es el nombre genérico que se da a la entrada de una capa de convolución que se define en alto × ancho × canales. Se da este nombre también a la salida de una capa de convolución ya que esta suele ser entrada de capas subsecuentes, bien sean de convolución, *pooling* u otras.

Por ejemplo, la imagen inicial, es en verdad un tensor que representa completamente la imagen según un alto × ancho × canales (al final son los píxeles y número de canales dependiendo de si es color o blanco y negro) ya que es la entrada de una primera capa de convolución. En capas convolucionales intermedias, si no hubiese activación y *pooling*, el tensor de entrada de la siguiente capa y el mapa de características de salida de la anterior, serían lo mismo.

Si bien, el término tensor, en la bibliografía, suele tener una referencia de elemento de input/output entre capas de la CNN (por ejemplo, un vector), el mapa de características hace hincapié a lo que se representa dentro de ese tensor, es decir, a qué característica fue extraída en una capa de convolución.

2.2.4. Función de activación (ReLU, *Rectified Lineal Unit*)

Es una función utilizada para conseguir una mejora de resultados, introduciendo un carácter no-lineal a la red CNN. Las activaciones de los nodos o neuronas importantes

para la red y desactivaciones / eliminaciones de los nodos o neuronas no importantes se realizan al finalizar una capa de convolución.

La función de activación más extendida es *ReLu* que corresponde a la siguiente expresión matemática.

$$(x) = \max(0, x) \qquad 0\ si\ x < 0\ ; x\ si\ x \geq 0 \qquad \textbf{(1)}$$

2.2.5. Capa de agrupación (*Pooling*)

La capa de *pooling* realiza una función estadística en su salida en función de los valores periféricos a un valor dentro del tensor, lo que gráficamente, podría verse como en la Figura 6. Por tanto, ayuda a reducir la "dimensionalidad" tras una capa de convolución sin perder efectividad en la extracción de características de los datos. Dos ejemplos de estas son: *Average pooling* y *Max pooling*, siendo este último el más común.

La función de *Max pooling* consiste en recorrer cada resultado de la capa *ReLu* o de la capa normalización con una máscara de $k \times k$ píxeles, moverla según el *stride* (desplazamiento) definido y seleccionar el mayor de los valores en cada uno de los movimientos. Usando el ejemplo anterior de la Tabla 2, utilizado para la explicación de la capa de convolución, para una capa denominada "*pool 1*" en la que se pone en uso la función max pooling definido por:

- Max pooling: 3×3
- Padding 0
- Stride 2,2

Se recorrerá cada una de las 96 matrices de 55 x 55 pixeles y se seleccionará el mayor valor de cada 3 x 3.

La máscara aplicada se moverá 2 posiciones en horizontal y 2 en vertical de manera que cada matriz se vea reducida a 27 pixeles ((55-3)/2+1). Por tanto, el número de neuronas y tamaño del array resultante de la aplicación de la función Max. Pooling resulta ser de 27×27×96.

La capa *pooling*, tal como se recoge en [9], ayuda a que la red sea aproximadamente invariante ante pequeñas traslaciones en la entrada.

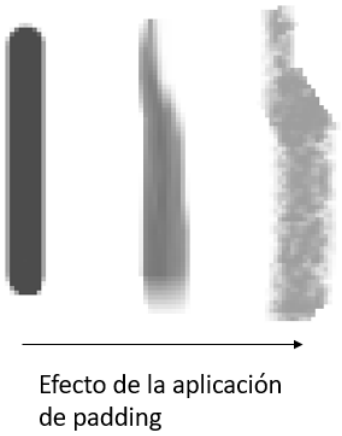

Figura 6. Ejemplo del efecto que provoca la aplicación del padding sobre la imagen de una línea vertical [17]

2.2.6. Capa densa (*dense layer*) o capas totalmente conectadas

Se trata de las capas que en la literatura sobre este tema suelen llamarse como capas *feed-forward*, como las encontradas en el perceptrón multicapa, tal y como se menciona en el apartado 2.1 y en las que los nodos o neuronas están totalmente conectadas entre una capa y la siguiente.

En las redes CNN, se usan generalmente al final de la red, en las proximidades de la salida, tras las sucesivas convoluciones. Pero antes de su uso, se realiza un aplanamiento del tensor de la capa final de convolución, *pooling*, etc (de la parte típica de la convolución) y se convierte en una dimensión (se pasa de varios canales a un vector), con el fin de poder aplicarlas. Así, se obtiene un vector, que recordando lo explicado en el apartado 2.1 para las redes de perceptrón multicapa, sería una representación de las características que se quieren analizar con las capas totalmente conectadas. De esta forma, en las sucesivas capas se realiza un aprendizaje no lineal de las características extraídas en la red CNN para realizar la clasificación de los datos, en este caso imágenes analizadas.

2.2.7. Normalización

La función de normalización es la unificación, en este caso del rango, de los pixeles resultantes de la capa de activación (o similar).

La función tiene en cuenta todos los mapas de características de la entrada y los lleva a un rango determinado, de manera que la convergencia durante el entrenamiento se acelere. Se trata de una capa especial que no se usa en la red base CNN y que cuya aplicación será entre las capas principales de la red (convoluciones o *pooling* por ejemplo).

2.2.8. Dropout

La finalidad de la función es mejorar la convergencia en cada clase o categoría analizada. Para ello, la capa *dropout* anula un determinado número de neuronas de forma aleatoria

en función de un parámetro que da constancia de la probabilidad de supervivencia de cada una.

Como la normalización, se trata de una capa especial que no se usa en la red base CNN y cuyo punto de aplicación será entre las diferentes capas principales de la red convolucional.

2.2.9. Weight decay o regularización

El *weight decay*, también conocido como regularización. Es un proceso por el que se penaliza en una función objetivo, los nodos con pesos más elevados. Esto se puede ver bien desde la teoría de los óptimos locales y globales, dado que, ante un óptimo local, los pesos elevados harían imposible moverse a otros óptimos (que puede ser el global) en las sucesivas iteraciones de entrenamiento. Desde el punto de vista de las convoluciones, viene a penalizar pesos de los filtros que extraen mejores características según el entrenamiento de retropropagación.

2.2.10. Softmax

La capa *softmax* se encarga de pasar los valores entrantes a la capa a probabilidades en el rango [0,1] a las neuronas de salida. Por ejemplo, una salida [0,2 0,8] con sólo dos clases flor y coche, indica un 20% de probabilidad de que sea flor y un 80% de que sea coche. Suele ser la última capa en una red CNN convencional de clasificación y su tamaño es siempre igual al número de objetos a clasificar.

2.2.11. Sobreajuste

El sobreajuste (*overfitting*) se puede producir en cualquier red neuronal. Durante el entrenamiento, los parámetros aprendidos se ajustan en exceso a los resultados del conjunto de entrenamiento incurriendo en un problema de generalización de la red. Es decir, la red creada es demasiado compleja para el problema a resolver.

Algunos métodos para evitar el *overfitting* son la regularización global, el *dropout* (aunque esto es dentro de la propia red entre algunas capas, por ejemplo), la división del conjunto de datos en entrenamiento, validación y test (un 60/10/30 o 70/10/20), la inicialización de pesos con valores procedentes de entrenamientos previos (hace falta tener una pequeña idea de qué valores pueden tener) o la restricción de los pesos si se supera un determinado umbral en algunos de ellos.

2.2.12. Módulos de *inception*

El aumento de la profundidad de una red neuronal se ha pensado para la obtención de mejores resultados. Sin embargo, su uso implica una serie de problemas entre los que se encuentran los mencionados a continuación:

- Conllevan un increíble aumento de los parámetros a ajustar en el aprendizaje lo que hace que se conduzca a redes con unos costes computacionales altísimos
- Cuanto mayor sea la profundidad de una red CNN convencional, mayor será su tendencia al *overfitting* si el conjunto de entrenamiento no es lo suficientemente elevado.

Los módulos *inception* proponen analizar o aumentar la dimensión del análisis de la red a lo ancho de la misma en lugar de ahondar en su profundidad.

Para ello se realiza sobre una misma zona o tensor una serie de convoluciones en paralelo con distintos tamaños de *kernels* (Figura 7).

La intención es detectar características en los distintos tamaños al mismo tiempo en lugar de añadir convoluciones sucesivas. Los *kernels* de mayores dimensiones son mejores para obtener características más grandes y globales, mientras que los de dimensiones menores son más apropiados para características más pequeñas y locales.

En la práctica se hace uso de máscaras 1×x1, 3×3 o 5×5 en paralelo (incluso con la función *Max-pooling*). La salida de ello resulta en una concatenación de dichos filtros apilados/agregados.

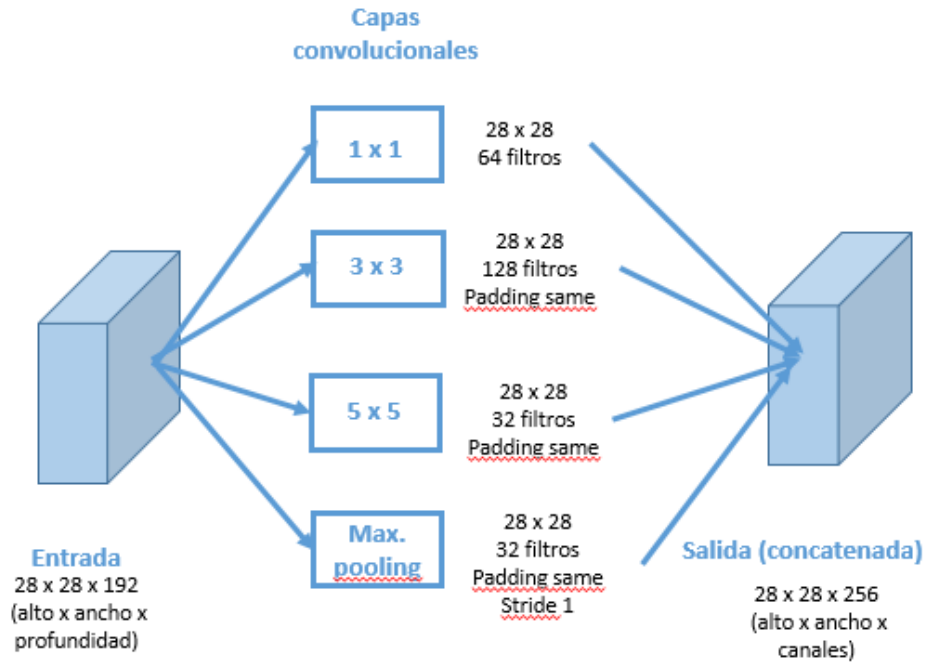

Figura 7. Esquema de un módulo inception [18]

En paralelo, con el propósito de reducir el número de parámetros a ajustar en el aprendizaje, se ha desarrollado una variante del esquema anterior que reduce la dimensionalidad mencionada. No obstante, existe una variante que incorpora además filtros 1×1 para reducir principalmente la dimensionalidad total de los filtros 3×3 y 5×5 con el fin de reducir aún más los parámetros, Figura 8.

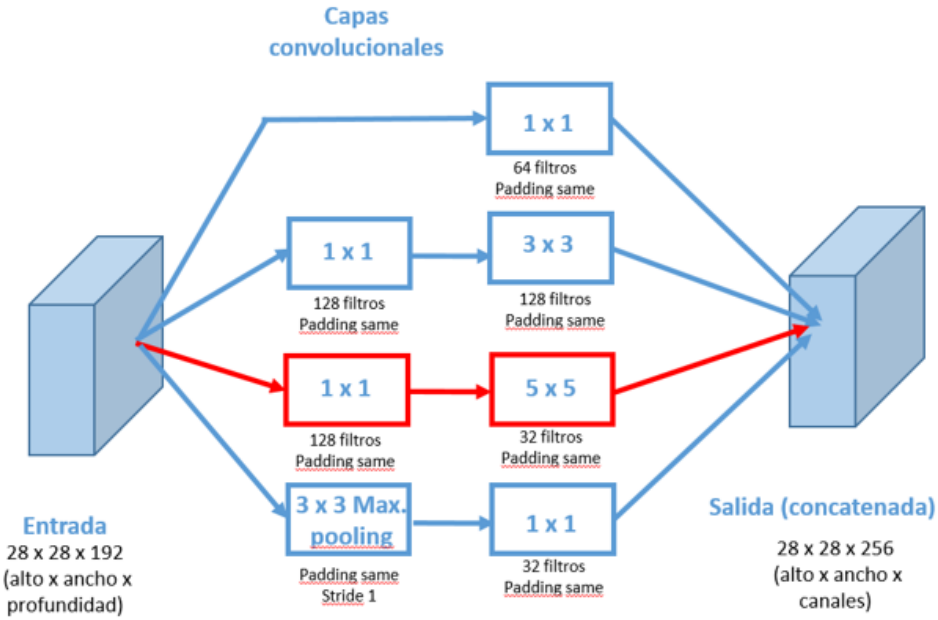

Figura 8. Esquema de módulo inception mejorado para reducir número de parámetros en la red [18]

Por estas características, se podría decir que los módulos *inception* suponen un uso avanzado en las redes CNN.

2.2.13. Cortocircuitos o atajos (*identity shortcut connection*)

Los cortocircuitos (o atajos) o en inglés *identity shorcut connection* en las redes CNN son otro caso de estrategias avanzadas que se pueden aplicar en las redes CNN.

Se basa en el simple planteamiento de hacer que de las capas de las redes CNN no sean secuenciales, sino que se puedan producir saltos entre las mismas.

Al igual que en el caso de los módulos de inception, estos cortocircuitos o atajos se implementan en algunas redes CNN avanzadas con el fin de eliminar los problemas que generan los aumentos de profundidad de las redes en busca de mejores resultados. Mejora sobre todo el problema del gradiente que se da al aumentar la profundidad. El gradiente se calcula sobre las capas durante el entrenamiento en retropropagación. Al aumentar la profundidad, el cambio entre dos capas consecutivas podría ser pequeño haciendo también mínimo el valor del gradiente e incluso convirtiéndolo en un valor cercano a 0. En caso de que esto suceda, se dirá que la red se degeneró. Obviamente, cuanto más profunda sea una red, es decir, más número de convoluciones realice, más peligro tiene de degenerarse.

Cómo realizar los saltos de los cortocircuitos y decidir cuáles utilizar depende de unas arquitecturas CNN a otras. Así, las redes ResNet, HighwayNet o FractalNet, usan respectivamente, bloques residuales, puerta de transmisión y bloques fractales para ejecutar saltos entre capas dentro de las redes CNN [19].

En cualquier caso, estos saltos, se pueden ver de forma global y esquemática según la Figura 9. Aquí, el camino normal de datos en la red sería el horizontal de las flechas rojas que se realiza entre capas de convolución, bloques azules. En la red, en bloques de capas de convolución, se permitirán saltos para no pasar por estos (líneas verdes). Pero al final, el salto o el camino normal de paso por los bloques de capas confluyen en el mismo punto de unión entre bloques (se realiza una agregación en ese punto). Esto se representa por los cilindros naranjas en la misma figura.

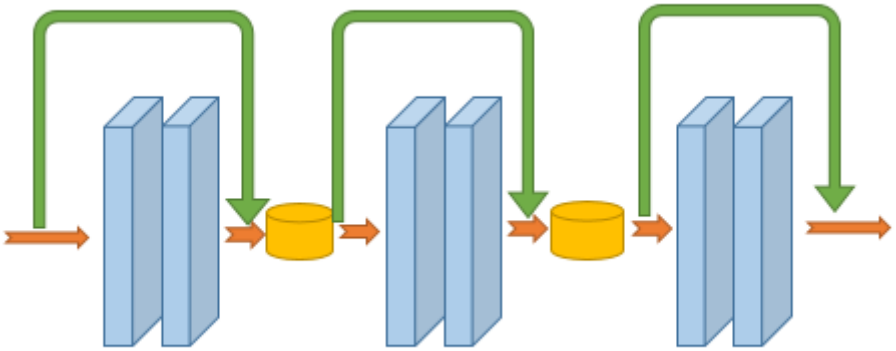

Figura 9. Esquema conceptual de los atajos realizados en las arquitecturas con cortocircuitos

Al comienzo de cada bloque la información X, se bifurca. Y atraviesa simultáneamente la conexión atajo (que aplica la simple función identidad) y la capa interna. Por ejemplo, en la arquitectura HighwayNet representada en la Figura 10, las puertas C y T determinan la contribución de cada camino en la configuración del resultado final, *y*.

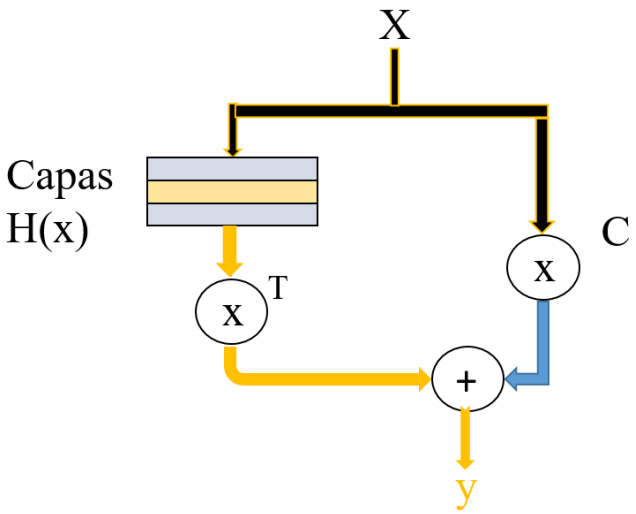

Figura 10. Esquema de la función que de puerta de transmisión que permite salto en la red HighwayNet [19]

2.2.14. *Transfer learning* o transferencia de aprendizaje

La transferencia de aprendizaje es una característica propia de las redes CNN, en la cual se permite la reutilización de redes ya entrenadas con conjuntos de datos distintos a los

que se pretenden utilizar en un momento determinado (a veces referidas como preentrenadas en la literatura).

Tal y como se explicó previamente, para las capas densas o totalmente conectadas, como en las capas de convolución, en la estructura básica de una red CNN hay una primera parte donde la red aprende a extraer características de los datos de entrenamiento (la parte donde se producen las convoluciones, *pooling*, normalización, etc.) y otra parte donde la red aprende a unir ese conocimiento de extracción de características de acuerdo a la resolución de un determinado problema (la parte de las capas totalmente conectadas). Entre ambas estructuras se produce un aplanamiento de los tensores. De esta forma, es intuitivo pensar que dadas dos redes CNN, de forma que una por ejemplo clasifica coches y furgonetas y otra coches y camiones con otro conjunto de datos en su primera parte van a ser iguales, dado que la extracción de características de imágenes donde hay coches, furgonetas y camiones será la misma e incluso los valores de sus parámetros en la convolución serán similares al ser objetos todos ellos con características similares. De esta forma, la mayor variación estaría en la parte final de la red. Así, si sobre, por ejemplo, en la primera red, se elimina las capas finales totalmente conectadas y se cambian por otras "nuevas" a entrenar con otro conjunto de datos, se puede hacer el uso total de la red CNN anterior sin necesidad de realizar un gran entrenamiento.

Esto conlleva una enorme ventaja ya que se pueden ajustar o diseñar redes CNN para distintas funciones a partir de modelos ya entrenados, lo que reduce drásticamente el tiempo de entrenamiento, que es elevado al tener que incluir muchas imágenes como se comentó anteriormente. De la misma forma, se reducirá, por tanto, el coste computacional asociado.

En la práctica, se usan redes ya entrenadas (pre-entrenadas) sobre conjuntos muy grandes de conjuntos de datos con muchos objetos. Así se puede hacer uso muy versátil de ellas para crear las redes CNN. Por ejemplo, en el caso de la red anterior descrita en la Tabla 2, es una red AlexNet con posibilidad de clasificar 1000 objetos.

2.3. Arquitecturas de modelos CNN típicas

Como se ha visto previamente, los modelos CNN, al igual que el resto de métodos basados en redes neuronales, son muy abiertos y ofrecen multitud de posibilidades, al permitir multitud de combinaciones en cuanto al número de convoluciones, uso de capas totalmente conectadas o no, tipo de activación, normalizaciones *pooling* o incluso la posibilidad de la inclusión de técnicas avanzadas como son los módulos *inception* o los cortocircuitos.

Desde sus comienzos, han ido apareciendo distintos modelos CNN que se diferencian, además de su arquitectura, principalmente por su precisión, tiempos y requerimientos de computación. Algunas de ellas, han sido divulgadas en mayor manera que otras, en base a estas características u otras como puede ser la facilidad de aplicación de la red en algoritmos mayores, tales como la detección de objetos. En cualquier caso, al final, la

selección de una u otra red, vendrá definida, entre otros criterios, por los requisitos finales del análisis de datos y del entorno y posibilidades de computación.

Para conocer más en profundidad los modelos CNN, a continuación, se muestran y explican algunas de las más típicas, que son las que se usan en el desarrollo práctico de este trabajo:

- AlexNet, para mostrar las particularidades base de una red CNN.
- Squeezenet, para comprobar la aplicación de bloques similares a los módulos Inception.
- ResNet, para comprobar el uso de los cortocircuitos.

2.3.1. AlexNet

Dentro de las redes neuronales convolucionales, una de las más generalizadas es la red AlexNet (Figura 5), creada por Alex Krizhevsky en colaboración con Ilya Sutskever y Geoffrey Hinton. Fue la ganadora de la competición ImageNet LSVRC (2012) en el año 2012 y, por tanto, una de las primeras redes CNN con mayor repercusión. Según la propia clasificación de Matlab, está englobada dentro de los modelos básicos de redes [20].

AlexNet es una red con una profundidad de 7 capas convolucionales creada para un conjunto de 1000 imágenes para cada una de las 1000 categorías en las que es capaz de clasificar y donde cada imagen tiene una resolución 227 x 227 x 3. Como se puede ver en la tabla 2, consta de unas 25 capas en total entre entrada (adquisición de datos de la imagen-píxeles-), convolución, ReLu, normalización, *max pool*, totalmente conectadas (*fully connected*), *dropout*, *softmax* y salida.

Dentro de sus particularidades cabe destacar:

- Es una de las primeras CNN que usan capas apiladas de convolución (antes de ella se solía realizar siempre convolución seguida de una capa de pooling). Además, la salida se corresponde con las 1000 categorías de imágenes, pensadas para la clasificación de ese número de objetos, en correspondencia con el conjunto de datos utilizado en la competición mencionada previamente (ImageNet).
- Fue de las primeras donde la función usada para la activación tras la convolución es ejecutada por la función ReLU (antes de esta las más usuales eran sigmoidea y tangente hiperbólica), lo que reduce enormemente el coste computacional, permitiendo, además, su uso con GPS.
- Fue también una de las primeras en usar capas de *dropout* o apagado por probabilidad, para mejorar problemas de sobreajuste y ganar generalización en la red.
- Por último, destacar que se realiza una capa de agrupación máxima superpuesta ("*overlapped Max-pool*") después de la primera, segunda y quinta capas de convolución. Las capas superpuestas de *maxpool* son simplemente capas de agrupaciones por máximos con pasos menores que el tamaño de la ventana. La capa *maxpool* 3x3 se usa con un paso de 2, por lo que se crean campos receptivos superpuestos [21].

En la Tabla 3 se sintetiza la arquitectura del modelo AlexNet, especificando las dimensiones de las entradas y salidas en cada capa, así como la naturaleza de éstas, se indica también los pasos que se aplican en las convoluciones (*stride*), el tamaño de los añadidos si existen (*padding*), los tamaños de los núcleos (*kernels*) de convolución y el número de éstos núcleos, y finalmente el número de parámetros involucrados en cada capa, que determinan el número de pesos a ajustar durante el proceso de aprendizaje.

Entrada			**Salida**			**Capa**	**Stride**	**Pad.**	**Tamaño kernel**		**In**	**Out**	**Nº de Parámetros**
227	227	3	55	55	96	**conv1**	4	0	11	11	3	96	34.944
55	55	96	27	27	96	**maxpool1**	2	0	3	3	96	96	0
27	27	96	27	27	256	**conv2**	1	2	5	5	96	256	614.656
27	27	256	13	13	256	**maxpool2**	2	0	3	3	256	256	0
13	13	256	13	13	384	**conv3**	1	1	3	3	256	384	885.120
13	13	384	13	13	384	**conv4**	1	1	3	3	384	384	1.327.488
13	13	384	13	13	256	**conv5**	1	1	3	3	384	256	884.992
13	13	256	6	6	256	**maxpool5**	2	0	3	3	256	256	0
						fc6			1	1	9216	4096	37.752.832
						fc7			1	1	4096	4096	16.781.312
						fc8			1	1	4096	1000	4.097.000
TOTAL													**62.378.344**

Tabla 3. Resumen de parámetros de la red AlexNet

2.3.2. Squeezenet

Se trata de una CNN creada para ser compacta y usada en sistemas donde el espacio de memoria usada es importante, pero siempre sin perder precisión. Fue creada por miembros de la Universidad de California, Berkeley y Stanford en el año 2016.

Al igual que otras CNN, fue creada partiendo como base de la pionera red AlexNet explicada en el punto anterior 2.3.1 para realizar modificaciones positivas sobre ésta.

Estas mejoras se basan en tres principios o estrategias que en fundamento son similares a los de las capas de *inception* (sección 2.2.12.):

1. Sustituir los filtros/kernels 3 x 3 por 1 x 1.
2. Disminuir el número de canales de los tensores de entrada a las convoluciones con filtros 3×3.
3. Realizar un submuestreo posterior en la red para incrementar los tamaños de los mapas de características en las salidas de convolución, basándose en el hecho de que tener mapas mayores conlleva a mejores resultados [22]. La idea base es realizar casi todas las convoluciones con *stride* 1 y sólo en las finales usar *strides* mayores que 1 (a mayor *stride* menores mapas de características en la salida porque mayor es el paso de salto del *kernel*). Con esto se consigue mantener un tamaño más o menos grande en los mapas de salida.

Como resultado global de estas premisas, los puntos 1 y 2 conllevan una reducción directa de los parámetros de la red y, para compensarlo, se apoya en la estrategia del punto 3, la cual permite aumentar la precisión aun con un bajo número de parámetros. En la implementación realizada en la realidad, esto se traduce en una serie de usos de

convoluciones, *maxpool* y sobre todo de los llamados módulos *fire*, que son la implementación directa de los principios 1 y 2.

Respecto de este nuevo concepto de módulo *fire,* hay que decir que se trata de un bloque de capas de convolución, que puede ser dividido en dos, tal y como se puede ver en la Figura 11:

- Una primera parte con convoluciones 1x1 que realiza una capa comprimida o *squeeze* (de ahí su nombre), la cual reduce el número de canales tras la convolución.
- Una segunda parte con convoluciones utilizando *kernels* de dimensiones 1×1 y 3×3, que crean la llamada capa de expansión, cuyos resultados son concatenados y aplicados sobre una activación ReLu. Entre *squeeze* y expansión también se usa la activación por ReLu.

Por ejemplo, en el llamado bloque "Fire3" de la red Squeezenet se usan 16 filtros de 1x1 en la primera capa comprimida y 64 filtros de 1x1 y 64 de 3x3 en la capa de expansión.

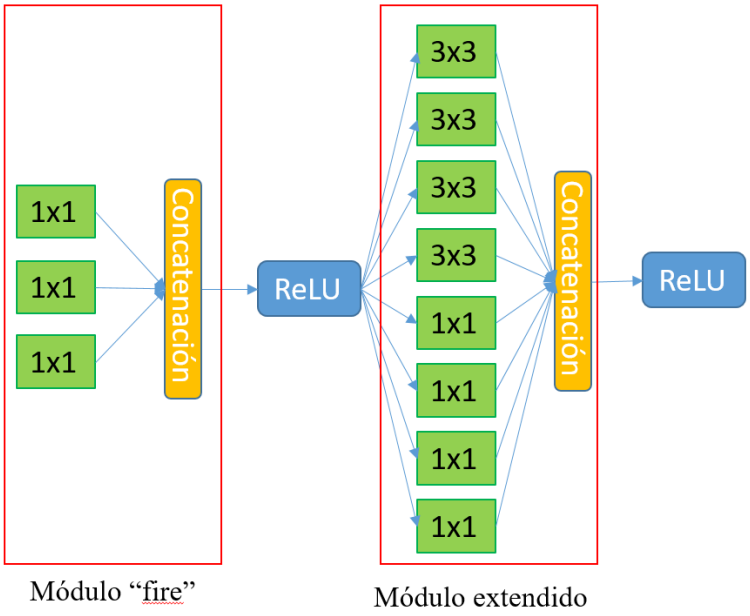

Figura 11. Esquema del bloque "fire" y etendido/expandido [9]

Cabe destacar que al final de la red, hay un *average pool* y una capa *softmax* con salida 1000 usada para la clasificación (como en la red AlexNet) y que tras el bloque de *Fire9* hay una capa de *dropout* de probabilidad 50% de eliminación para ganar generalización (como AlexNet).

Otra característica destacable es que no hay capas totalmente conectadas, lo que hace que las necesidades de computación se reduzcan enormemente al reducir igualmente el número de parámetros. Esto se consigue gracias a la convolución 10 ("conv10") donde la salida tiene 1000 canales (14×14×1000) y, la cual, viene a representar ya un mapa de características para cada clase. Tras esto, la capa de *average pool* comentada previamente,

lo aplana totalmente, promediando la respuesta de cada mapa de características de cada clase.

Todo esto hace que se obtenga una red con 50 veces menos parámetros que AlexNet, precisión similar y tamaño 0.5Mb [22] (ver la tabla 4). Por esta razón, Matlab la clasifica dentro de las CNN a usar para el desarrollo en sistemas embebidos [20].

En la Tabla 4 se resume el modelo Squeezenet, mostrando el tamaño de salida y los núcleos (*kernels*) utilizados, así como la profundidad de cada capa. Igualmente, se muestra la repetición de las fases de *squeezenet* explicadas en este apartado y el número de parámetros finales involucrados en el proceso de aprendizaje.

Nombre capa	Tamaño salida	Tamaño kernel/ stride	Produndidad	1 x 1 (squeeze)	1 x 1 (expansion)	3x 3 (expansion)	Nº parámetros
input image	224 x 224 x 3						
conv1	111 x 111 x 96	7 x 7/2 (x 96)	1		100 % (7 x 7)		14.208
maxpool1	55 x 55 x 96	3 x 3/2	0				
fire2	55 x 55 x 128		2	16	64	64	5.746
fire3	55 x 55 x 128		2	16	64	64	6.258
fire4	55 x 55 x 256		2	32	128	128	20.646
maxpool4	27 x 27 x 256	3 x 3/2	0				
fire5	27 x 27 x 256		2	32	128	128	24.742
fire6	27 x 27 x 384		2	48	192	192	44.700
fire7	27 x 27 x 384		2	48	192	192	46.236
fire8	27 x 27 x 512		2	64	256	256	77.581
maxpool8	13 x 12 x 512	3 x 3/2	0				
fire9	13 x 13 x 512		2	64	256	256	77.581
conv10	13 x 13 x 1000	1 x 1/1 (x 1000)	1				103.400
avgpool10	1 x 1 x 1000	13 x 13/1	0				
TOTAL							421.198

Tabla 4. Resumen de parámetros de la red Squeezenet

2.3.3. ResNet

ResNet o Residual Network fue desarrollada por miembros de Microsoft y ganó el concurso ILSVRC 2015 y COCO 2015. Su motivación estriba en el problema comentado anteriormente basado en que en redes de gran profundidad aparecen problemas de sobreajuste y gradientes que tienden a 0, creando problemas en el entrenamiento (ver sección 2.2.11.) y traduciéndose en pérdida de precisión a pesar del aumento de profundidad (problema de la degeneración).

ResNet, Figura 11, viene a solucionar la segunda parte, creando redes bastante profundas, pero sin problemas de pérdida de precisión por ello.

Para ello, las ResNet usan la arquitectura de cortocircuito anteriormente comentada en el apartado 2.2.13. . Según este principio, se permite la supresión o salto de ciertas capas de la red, en virtud de, en este caso, los bloques residuales. En lugar de usar convoluciones apiladas o convoluciones seguidas de *pooling*, en las redes ResNet se apilan sucesivamente bloques residuales. De esta forma, en lugar de ajustar parámetros en la convolución para que se obtenga un mapa con unas características que en su conjunto arrojen el resultado exacto deseado, se ajustan parámetros en la convolución para que el

residuo sea 0 (el residuo es el error cometido entre el mapa de características y el resultado que debe ser).

De esta forma, como se puede ver en la Figura 12, se ajustan los parámetros para que la función residuo, *F*, implementada en la red y compuesta de funciones de excitación con activación intermedia (son convoluciones entre ellas), sea 0, en lugar de *X* (el resultado que inicialmente sería el esperado). Esta metodología facilita el cálculo con respecto al modelo inicial en el que se ha de ajustar la función para obtener el valor de *X*.

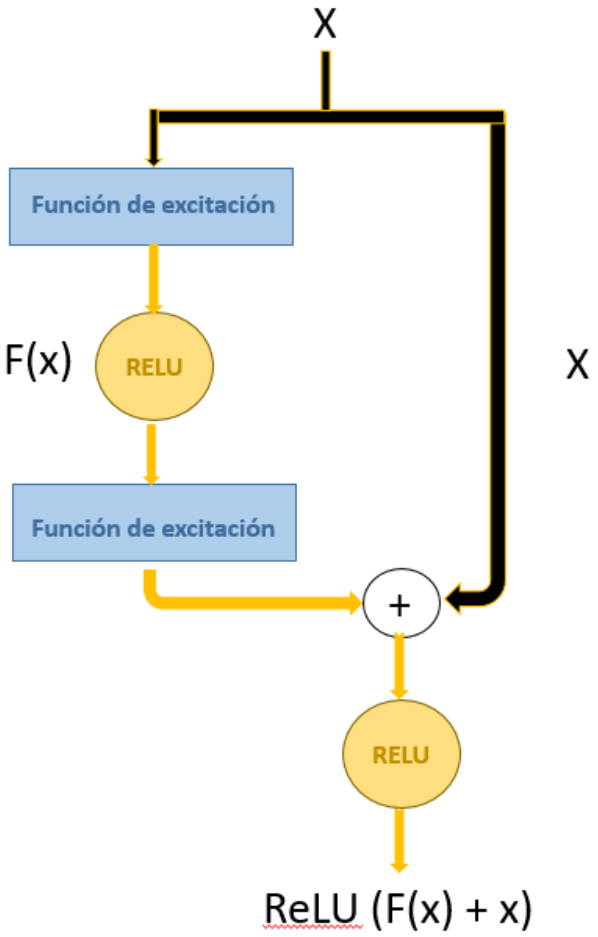

Figura 12. Esquema de la función residuo y el posible salto del cortocircuito en ResNet

Esto es fácil de entender si se analiza desde la perspectiva del gradiente en la retropropagación a capas no lineales. De esta forma, cuando se ajusta $F(X)$ para obtener el resultado deseado X, $F(X)=0$, con lo que su salida será X, que viene a ser a su vez la identidad. De esta forma sencilla, pero a su vez efectiva, se elimina el problema de que los gradientes se acerquen a 0, dado que cuando lo hacen, se "saltan" esas convoluciones de las funciones de activación del bloque residual evitando el problema.

Debido en parte a la supresión del problema del gradiente, la red ResNet se puede desarrollar hasta bastante profundidad. Así se encuentran distintas redes llamadas ResNetXXX, donde XXX es el número de capas. Son habituales ResNet18, ResNet34,

ResNet50, ResNet101 y ResNet152 [7]. El modelo ResNetXXX se basa en una convolución inicial con kernel 7×7 y a partir de ahí bloques de convoluciones sobre la comentada función objetivo. En cada bloque se usan siempre filtros 3×3 que se repiten una serie de veces dependiendo del modelo concreto de ResNet que se utilice.

En ellas se conserva el tamaño del mapa de características a lo largo del bloque y, en los cambios de bloque, la primera de esas convoluciones se hace con un *stride* de 2 para reducir el tamaño del mapa de características a la mitad. Cabe destacar que, con el fin de reducir el coste computacional al aumentar la profundidad, a partir de ResNet50 (incluida esta), se incorporan también filtros 1×1, seguidos de filtros 3×3 y se finaliza con otros filtros 1×1. El salto siempre se permite dentro de esos bloques en sub-bloques, es decir, en ResNet18 y ResNet34 cada dos convoluciones con filtros 3×3 y en ResNet50, ResNet101 y ResNet152, cada 1 convolución se aplicará un filtro 1×1, luego con 3×3 y finalmente 1x1. Esto puede verse de manera esquemática en la Figura 13, donde se muestra el bloque de convoluciones con estos filtros 1×1 y 3×3, en las cuales se calcula el residuo F(x), mostrado en la Figura 12.

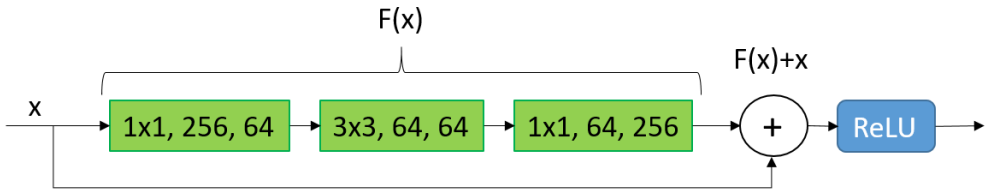

Figura 13. Ejemplo de salto típico en los bloques de ResNet50, ResNet101 y ResNet152 con la muestra de los 3 sub-bloques

En la Tabla 5 se resume el modelo ResNetXXX, mostrando los cinco niveles de convoluciones máximos y el tamaño de salida dado en cada uno. Igualmente, se describe la combinación de filtros 1×1 y 3×3 usados en cada tipo de detector Resnet, así como la repetición de estas combinaciones en cada tipo.

Nombre capa	Tamaño salida	ResNet18	ResNet34	ResNet50	ResNet101	ResNet152
conv1	112 x 112	7 x 7, 64, stride 2				
conv2 . x	56 x 56	3 x 3, max pool, stride 2				
		$\begin{bmatrix} 3\,x\,3, 64 \\ 3\,x\,3, 64 \end{bmatrix}$ x 2	$\begin{bmatrix} 3\,x\,3, 64 \\ 3\,x\,3, 64 \end{bmatrix}$ x 3	$\begin{bmatrix} 1x1, 64 \\ 3x3, 64 \\ 1x1, 256 \end{bmatrix} x\ 3$	$\begin{bmatrix} 1x1, 64 \\ 3x3, 64 \\ 1x1, 256 \end{bmatrix} x\ 3$	$\begin{bmatrix} 1x1, 64 \\ 3x3, 64 \\ 1x1, 256 \end{bmatrix} x\ 3$
conv3 . x	28 x 28	$\begin{bmatrix} 3\,x\,3, 128 \\ 3\,x\,3, 128 \end{bmatrix}$ x 2	$\begin{bmatrix} 3\,x\,3, 128 \\ 3\,x\,3, 128 \end{bmatrix}$ x 4	$\begin{bmatrix} 1x1, 128 \\ 3x3, 128 \\ 1x1, 512 \end{bmatrix} x\ 4$	$\begin{bmatrix} 1x1, 128 \\ 3x3, 128 \\ 1x1, 512 \end{bmatrix} x\ 4$	$\begin{bmatrix} 1x1, 128 \\ 3x3, 128 \\ 1x1, 512 \end{bmatrix} x\ 8$
conv4 . x	14 x 14	$\begin{bmatrix} 3\,x\,3, 256 \\ 3\,x\,3, 256 \end{bmatrix}$ x 2	$\begin{bmatrix} 3\,x\,3, 256 \\ 3\,x\,3, 256 \end{bmatrix}$ x 6	$\begin{bmatrix} 1x1, 256 \\ 3x3, 256 \\ 1x1, 1024 \end{bmatrix} x\ 6$	$\begin{bmatrix} 1x1, 256 \\ 3x3, 256 \\ 1x1, 1024 \end{bmatrix} x\ 23$	$\begin{bmatrix} 1x1, 256 \\ 3x3, 256 \\ 1x1, 1024 \end{bmatrix} x\ 36$
conv5 . X	7 x 7	$\begin{bmatrix} 3\,x\,3, 512 \\ 3\,x\,3, 512 \end{bmatrix}$ x 2	$\begin{bmatrix} 3\,x\,3, 512 \\ 3\,x\,3, 512 \end{bmatrix}$ x 3	$\begin{bmatrix} 1x1, 512 \\ 3x3, 512 \\ 1x1, 2048 \end{bmatrix} x\ 3$	$\begin{bmatrix} 1x1, 512 \\ 3x3, 512 \\ 1x1, 2048 \end{bmatrix} x\ 3$	$\begin{bmatrix} 1x1, 512 \\ 3x3, 512 \\ 1x1, 2048 \end{bmatrix} x\ 3$
	1 x 1	average pool, 1000 totalmente conectada, softmax				

Tabla 5. Resumen de parámetros de las distintas redes ResNetXXX

Por último, al final de la red se usa un *average pool* y se acaba con una capa *softmax* de 1000 clases para clasificación al igual que AlexNet y Squeezenet.

2.4. Usos comunes de las CNN

En esta sección se detallan brevemente algunas aplicaciones donde se usan comúnmente las redes CNN. En concreto, existen multitud de usos de las CNN y, es por eso, por lo que su conocimiento y desarrollo es tan profundo. No obstante, desde un punto de vista general, se pueden encontrar algunos objetivos importantes y concretos en el uso de estas redes:

1. *Clasificación de objetos*. Se trata básicamente en la identificación de objetos en una imagen dada. Es la aplicación origen de las CNN. Dada una imagen cualquiera, se puede identificar los objetos que en ella aparecen, generando un determinado umbral de clasificación dado por el modelo CNN.
2. *Detección de objetos*. Se trata de un paso más allá de la clasificación, ya que además se localiza el objeto en la imagen por medio de un *bounding box* (ver apartado 3).
3. *Segmentación de objetos*. Yendo aún más allá, de la detección, está la segmentación, a través de la que se conoce además a la región a la que pertenece dicho objeto. Esto se consigue sabiendo si un píxel pertenece a una categoría u a otra en el modelo CNN.
4. *Reconocimiento de audio*. Se trata de reconocimiento de voz que bien pueden ser vocales, sílabas, palabras o frases.
5. *Procesamiento de vídeos*. Al igual que el procesado de imágenes, existen modelos CNN que pueden procesar vídeos, si bien en este tipo de aplicaciones aparece una componente temporal, cuyo tratamiento es más apropiado mediante la utilización de modelos recurrentes de redes.

3. Detección de objetos

3.1. Introducción a la detección de objetos

La detección de objetos conlleva un proceso conjunto de clasificación y localización de objetos en una imagen. De esta forma, por un lado, los detectores de objetos son capaces de encontrar y clasificar dichos objetos dentro de las distintas clases para las que aquellos fueron entrenados, Figura 14, y se generará una salida de clase con una precisión asociada. Por otro lado, generará automáticamente un área donde se encuentra ese objeto clasificado en la imagen. Esta localización se lleva a cabo por medio de un rectángulo delimitador conocido como *bounding box*, el cual encuadra espacialmente dicho objeto dentro de la imagen, otorgándole una posición X e Y, con un ancho y alto a partir de este. De forma resumida, la Figura 14, vendría a mostrar dicha detección de un objeto de clase corazón y otro de clase estrella.

De esta forma, los resultados en lugar de ser únicamente un vector con valores entre 0 y 1 de igual longitud a las clases entrenadas, indicando la probabilidad de que ese objeto corresponda a la clase correspondiente, se da también una posición X, Y, un acho y un alto para cada objeto detectado en esa clase. Así, al incluir el factor de la localización espacial, se complica la relación de resultados de clasificación con los objetos.

De la misma forma, dos objetos que incluso pueden pertenecer a dos clases distintas pueden tener *bounding boxes* solapados, por ejemplo. También puede ocurrir que se detecten varios objetos en la misma imagen, cosa que con las redes de detección de clasificación puede crear problemas en la detección si hay varios objetos de clases distintas (recordar que la salida es un vector con valores entre 0 y 1 de probabilidades para cada pertenencia a la clase y los resultados, por tanto, pueden no ser claros). De igual forma, dos objetos de la misma clase no se detectarán (detectaría uno sólo en el mejor caso).

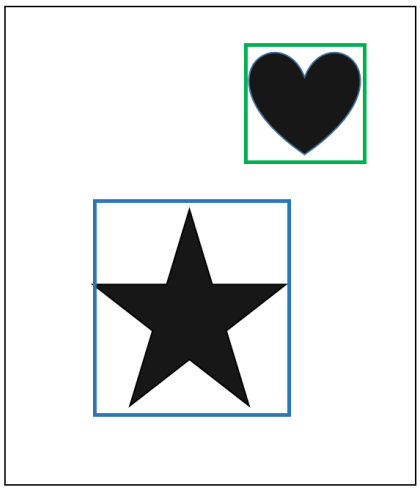

Figura 14. Esquema de dos objetos (corazón y estrella) detectados por dos bounding boxes

En la detección de objetos se puede encontrar un primer grupo, que al igual que ocurría con los modelos CNN, empleaban algoritmos de clasificación y localización fuera del ámbito del aprendizaje profundo. Algunos ejemplos son el algoritmo de Viola-Jones, uso de wavelets de Haar o el uso de histogramas de gradientes orientados los cuales se suelen usar con técnicas de aprendizaje avanzadas como AdaBoost o similares.

De la misma forma a como ocurría en las CNN, la creación artificial de estos algoritmos para adaptar el proceso natural de detección y aprendizaje crea, en algunos casos, problemas de generalización en ellos para distintos tipos de objetos en distintas posiciones, formas y tamaños. Su única ventaja es que no es necesario un gran número de imágenes para poder llevar a cabo el proceso de detección. Por el contrario, los detectores basados en CNN sí que necesitan un elevado número de imágenes y una gran capacidad de computación (tanto para el entrenamiento como para el proceso de detección). No obstante, a cambio, adquieren una mayor generalización. Aun así, tienen ciertos problemas como son la necesidad de entrenamiento con un gran coste computacional por retropropagación, el uso del tamaño y forma de los *anchor boxes* requeridos para enmarcar objetos como paso previo al entrenamiento (si el detector en cuestión usa *anchor boxes*) y el fallo en la detección de objetos muy pequeños en las imágenes (por el límite de imagen de entrada a los modelos CNN, por el coste computacional de imágenes mayores o por el uso de *anchor boxes* mayores en el entrenamiento).

En el segundo grupo de algoritmos de detección de objetos, se encuentran lo que serían los detectores neuronales basados en CNN de clasificación vistas en la sección 2. Estos detectores parten siempre de una determinada CNN a la que se le implementan ciertas capas, ya sea de convolución, totalmente conectadas o *pooling*, por ejemplo, o los resultados en esas capas se usan como entradas a otro tipo de clasificadores. Por ejemplo, en máquinas de vector soporte ("*Support Vector Machine*"–SVM–) con el fin de conseguir las tareas de localización del objeto en la imagen a la vez que se hace la clasificación.

Al final, el funcionamiento teórico se basa en que las redes CNN hacen tareas de clasificación en lugar de a la imagen completa, a una pequeña zona de ella. De esta forma, se pueden detectar múltiples objetos distintos o iguales. Posteriormente, se hace en una zona contigua con igual tamaño. Y así sucesivamente. Esto hace que, en última instancia, el coste computacional de los detectores de objetos sea mucho más alto que en una clasificación normal.

Este tipo de estrategias se puede a su vez diferenciar en dos subgrupos, por un lado, los que "miran una vez", detectores de un estado, y los que "miran dos veces", detectores de doble estado. La diferencia principal entre ellos es que en los de doble estado se extrae de la imagen primeramente una serie de regiones candidatas a contener los objetos a detectar y, después, estas se usan para su clasificación. Mientras, las de un estado realizan los dos pasos al mismo tiempo. Entre los de dos estados se encuentras, RCNN, Fast-RCNN, Faster-RCNN, Mask-RCNN, mientras que en los de un estado estarían YOLO (con sus distintas versiones), SSD o CornetNet, que se describen en detalle más adelante.

Por último, conviene destacar que los detectores de objetos son un campo de crecimiento ya que tiene aplicaciones muy útiles, tal y como pueden ser, las funciones de seguridad de personas en entornos de trabajo o peligrosos, el contaje de personas o distintos vehículos, detección de patologías en imágenes médicas, mejora del proceso de recarga de mercancía en supermercados, detección de patologías en agricultura o conducción autónoma.

3.2. Glosario de términos y conceptos en los detectores de objetos

A continuación, se muestra un pequeño glosario de términos y conceptos con el fin de determinar la base de ciertas palabras comúnmente usadas en la detección de objetos de una forma más o menos instructiva y organizada.

3.2.1. *Bounding box* o caja delimitadora

Se trata de un rectángulo que viene a delimitar la posición o ubicación de un objeto en la imagen. Lo delimita, por tanto, espacialmente en coordenadas X, Y, y con un ancho y un alto definidos.

Para el entrenamiento de los detectores por retropropagación se usa una gran cantidad de imágenes acompañadas de anotaciones donde se establecen la categoría a la que pertenece un determinado rectángulo o *bounding box*. Estas anotaciones tienen que ser previamente realizadas por expertos en el área de aplicación sobre las imágenes, con el fin de realizar el aprendizaje supervisado propiamente dicho. En este caso del rectángulo que delimita exactamente el objeto, el *bounding box* se viene a llamar técnicamente *ground truth bounding box* para diferenciarlo del que resulta de una detección con un detector cualquiera ya entrenado.

Por otro lado, los *anchor boxes*, son un tipo de *bounding boxes* con tamaños predefinidos. *Bounding box* es por tanto una forma genérica de llamar a las diferentes cajas delimitadoras de objetos que se usan en las herramientas de detección de objetos.

Un punto a tener en cuenta es que, dependiendo del detector y su técnica implícita, en una detección puede haber varios *bounding boxes* para un mismo objeto, pero con diferentes coeficientes o *scores* (ver sección 3.2.4.). Por este motivo, se hace uso de métodos de regresión para estimar los tamaños de los objetos a detectar. Es importante, no confundir esto con que en una imagen haya solapamiento entre distintos *bounding boxes* de distintos objetos diferentes, cosa que es intuitivamente lógica.

Por último, cabe destacar que aun en detectores con gran precisión entrenados sobre ciertos objetos, los *bounding box* resultantes de la detección pueden probablemente no coincidir con los *ground truth bounding boxes* por las propias características de las redes CNN.

3.2.2. *Labels* o nombres de las clases

Se trata de los nombres de las clases de los objetos para los que se entrena un detector. Es decir, qué objeto es. Como se comentó en los *bounding boxes*, es un experto humano el que añade la información del nombre de la clase asociada a los *bounding boxes* de una imagen

3.2.3. IoU

Intersection over Union o intersección sobre la unión se trata de un índice, también llamado índice de Jaccard [23], el cual mide la similitud entre dos conjuntos.

Este viene a determinar el área de solapamiento entre un *bounding box* (o *anchor box* si lo usa el detector) y el *ground truth bounding box*, cuya expresión es:

$$IoU = \frac{Área\ solapamiento\ bounding\ box\ o\ anchor\ con\ ground\ thruth\ bounding\ box}{Área\ suma\ de\ bounding\ box\ y\ ground\ thruth\ bounding\ box} \quad \textbf{(2)}$$

De esta expresión se deduce que tomará siempre valores entre 0 y 1 y su significado gráfico vendría a ser el representado en la Figura 15.

Viene a indicar, por tanto, cómo de bien un detector ajusta sus *bounding boxes* de detección a los óptimos usados del aprendizaje, *ground truth bounding box*. Las diferencias pueden estar en X, Y o en el ancho y el alto de los *bounding boxes* generados, Figura 16.

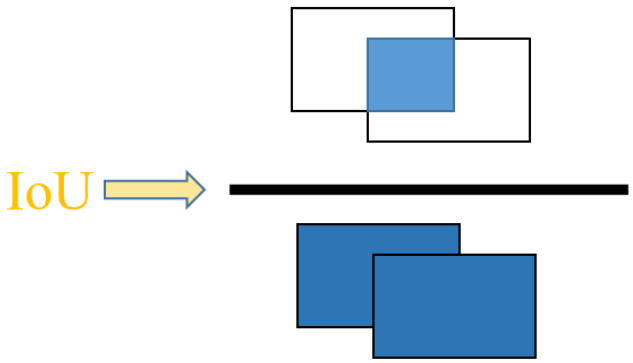

Figura 15. Ejemplo significado gráfico del IoU

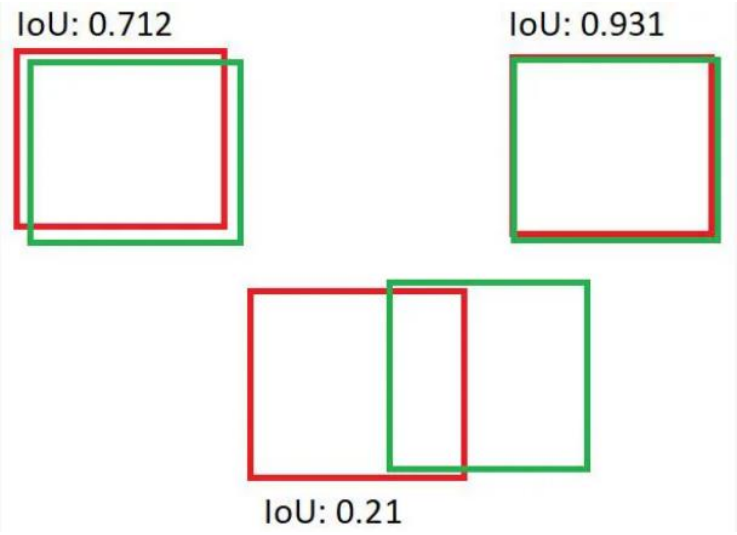

Figura 16. Ejemplo de diferentes IoU que se pueden dar entre bounding boxes y ground truth bounding boxes

3.2.4. *Confidence score*s o coeficientes de confianza

Se trata de la probabilidad de que un *bounding box* (o un *anchor box*) contenga un objeto. Se trata por tanto de un índice relacionado con la capacidad de clasificación de los detectores de objetos (extensible a las CNN). De esta forma, no se puede considerar el *confidence score* por sí mismo para una detección correcta. Para ello, además de un *confidence score* alto, tiene que haberse identificado correctamente el objeto en cuestión con su clase adecuada y que el IoU, visto anteriormente, sea también alto. Esto, de forma práctica, se realiza estableciendo umbrales para delimitar si el *score* es alto o no y si el IoU es apto o no.

Los *scores* vendrán siempre expresados con valores comprendidos entre 0 y 1, ya que vienen a indicar una probabilidad, tal y como se vio en la sección 2 con las redes CNN. Así, 1, sería una detección de 100% de un determinado objeto con respecto a la clase identificada y, 0, ninguna detección o de una clase incorrecta.

En relación con el último punto, cabe destacar que todos los *bounding boxes* usados en una red tendrán una detección mínima de 0. Así, si en una detección se incluyese un umbral de valor mayor o igual que 0, como en la Figura 17, aparecerían todos los *anchor boxes* que usa el detector.

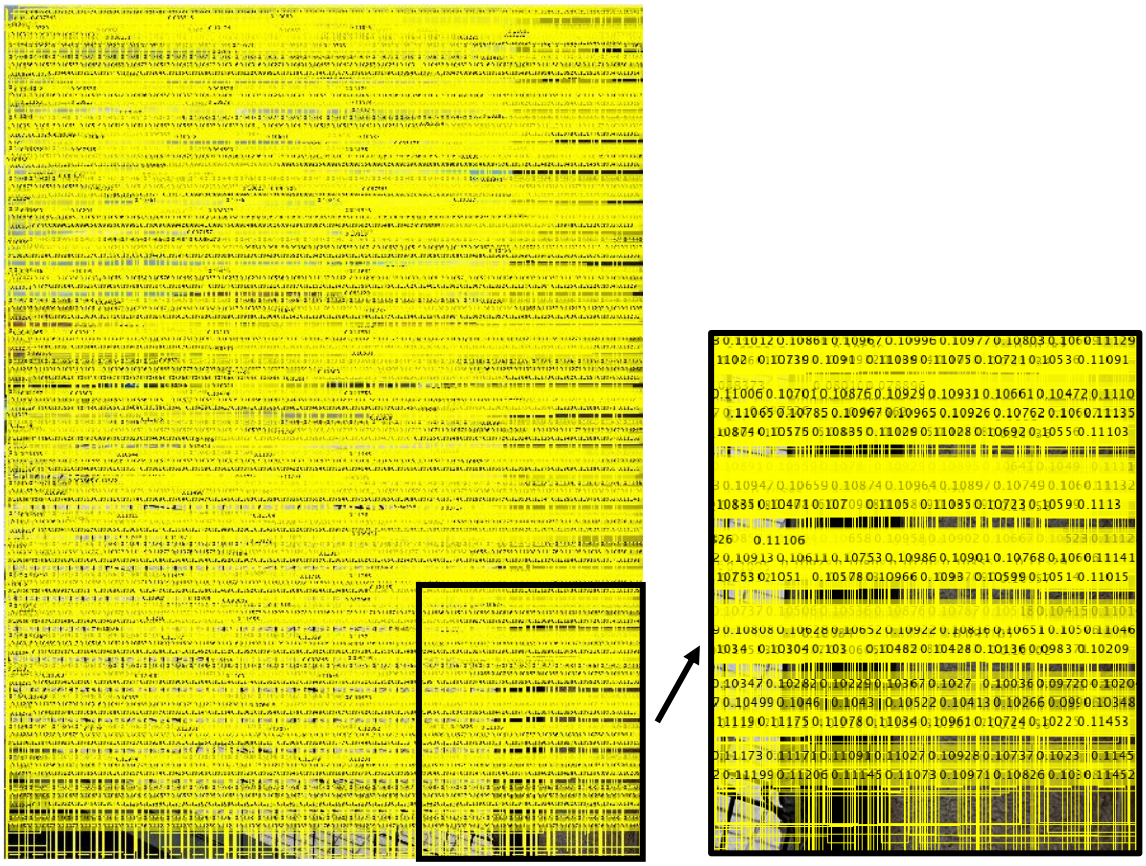

Figura 17. Ejemplo de anchor boxes de una imagen en Matlab con los confidence scores muy bajos (umbral de 0.1 en los coeficientes)

3.2.5. Anchor boxes

Los *anchor boxes* se pueden definir como regiones probables o propuestas que están delimitadas por *bounding boxes*. Visto de otra forma, un *anchor box* es una región o área de la imagen definida por el rectángulo de *bounding box* en variables X, Y, alto y ancho que se establece como posible contenido de un objeto en cuestión. Los *anchor boxes* no dejan de ser un artificio usado por algunos de los tipos de detectores de objetos con el fin de seleccionar determinadas áreas de una imagen o bien usarlos para el entrenamiento con el fin de ajustar los parámetros del detector de manera que los *bounding boxes* generados sean los más similares a los ground truth bounding boxes

Diferenciándolo tendríamos por un lado los *anchor boxes* usados en el entrenamiento de algunos detectores. En estos se usan un conjunto de *anchor boxes* en la imagen. A cada *anchor box* se le asigna la categoría de fondo en el entrenamiento y un *offset* o desplazamiento sobre los *ground truth bounding boxes*, es decir, la distancia en x e y de la imagen al respectivo *ground truth bounding box*. Obviamente, la mayoría de los *bounding boxes* no solaparán con los ground truth *bounding boxes* y, por ello, tiene que asignarseles categoría Fondo (*Ground*), Figura 18.

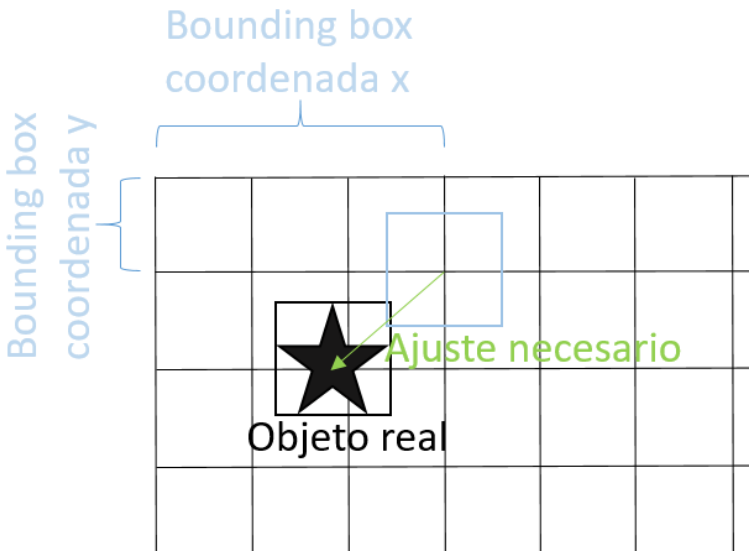

Figura 18. Ejemplo esquemático del ajuste del anchor box de detección a la posición real del objeto

Por otro lado, desde el punto de vista de la detección, se establecen diferentes *anchor boxes* en posiciones fijas de la imagen, sobre las cuales se determinará la precisión de clases (para cada *anchor box* se realizará una predicción de clase) y el cálculo de los *offsets* a cada *bounding box* de los objetos con el fin de crear otros *bounding boxes* que se asemejen lo máximo posible a los *ground truth bounding boxes*. Estos *bouinding boxes* predichos pueden ser similares espacialmente entre sí al partir de varios *anchor boxes*. Lo que se hace es eliminar los de categoría Fondo (Ground), listarlos y ordenarlos según el confidence *scores* y, mediante técnicas de supresión no-máxima a cada uno de la lista, quedarse sólo con aquellos que no intersectan por encima de un umbral consigo mismos, es decir con un IoU por debajo de un determinado umbral. Una vez realizado esto y al estar ordenados por *confidence scores*, los resultantes serán la predicción del detector para esos objetos de la imagen.

Como es obvio, el tamaño y forma de los *anchor boxes* (alargados verticalmente u horizontalmente, cuadrado, etc.) tienen un gran impacto en la detección de objetos. Para establecer estos *anchor boxes*, bien se pueden indicar directamente con sus coordenadas X, Y, ancho y alto o bien se pueden calcular con algún algoritmo para tal efecto. El primer caso conlleva un gran riesgo y parte de conocer al menos la forma de los objetos a calcular, para en definitiva saber la forma de su *ground truth bounding box*. Por ejemplo, la detección de un objeto de clase peatón será siempre un rectángulo alargado verticalmente. No obstante, su tamaño puede variar por la relación de tamaño del peatón con la imagen o su altura de rectángulo por altura del peatón que variará de unas imágenes a otras. Una de estas técnicas de generación es el agrupamiento por K-medias (*K-means*) para buscar la similitud entre *anchor boxes* propuestos y los *ground truth bounding boxes* por medio de la distancia entre ambos, dada por el IoU. Por ejemplo, Matlab tiene desarrollada una función que estima y genera los anchores boxes de unas imágenes usando el IoU como distancia para medir la mencionada similitud (función "estimateAnchorBoxes"). La virtud es que ésta es invariante al tamaño de los *anchor*

boxes, lo que beneficia a producir errores mayores con tamaños mayores de *anchor boxes* y permite la agrupación de *anchor boxes* de aspecto similar [24].

3.2.6. Ground truth

Se identifica de esta forma al fondo de la imagen de entrenamiento (*training*) donde se encuentran los objetos. Ya se ha visto antes que *Ground* sería la clase Fondo creada para trabajar con los *anchor boxes* correctamente, pero esta viene a relacionarse con los objetos. El rectángulo que inscribe a esos objetos es el llamado *Ground truth bounding box*.

3.3. Tipos de detectores de objetos

Los detectores de objetos es un campo de la inteligencia artificial en constante estudio y desarrollo, con múltiples campos de evolución, tal y como vimos en la primera parte de esta sección. Algunos ejemplos de estas vías de desarrollo serían la medicina, la agricultura o la vigilancia, por citar sólo algunos.

No obstante, el principal camino de desarrollo es la búsqueda de unos detectores que solucionen su principal problema, que no es otro que la velocidad de aplicación, ya que esta está restringida por los recursos computacionales. Igualmente, se centran también en el aumento de la precisión en general y, en objetos concretos, como puede ser objetos de pequeño tamaño en relación con el total de la imagen.

Como se explicó en la sección 3.1, existen detectores que sólo miran una vez (de un paso) y otros que miran dos veces (de dos pasos). En cualquier caso, en este trabajo se explican para los de dos pasos, RCNN, Fast-RCNN y Faster-RCNN, y de un paso, SSD, YOLO V3 y YOLO V4. Todos ellos, salvo Fast-RCNNk, que se explica para poder entender Faster-RCNN, se usan en el desarrollo práctico de este proyecto.

3.3.1. RCCN

Los detectores RCNN o *Region-based* CNN [25], son uno de los primeros detectores de objetos y uno de los más básicos, Figura 18.

Se trata de un detector que “mira dos veces” (detectores de dos estados o dos pasos) y, al igual que estos, se basa en primero obtener una serie de regiones en la imagen, que se pueden obtener mediante técnicas de SURF [26] o SIFT [27], agrupaciones de bordes u otras aplicaciones de visión por computador como un algoritmo de edge boxes. Matlab, por ejemplo, usa 2000 regiones de interés (ROI, *Region of Interest*) por defecto en las funciones de entrenamiento de redes RCNN extraídas por un método de *edge boxes*.

En una segunda fase, se redimensionan las regiones para que entren a un tamaño adecuado para la CNN básica preentrenada (normalmente se reducen de tamaño).

Así pues, se aplicará una red CNN por cada región propuesta, las cuales tendrán un vector de salida que será un vector con valores de las características de esa región. De esta forma, con este vector y, bien haciendo uso de la propia red CNN, o, bien de una arquitectura externa como una SVM, se clasificará cada ROI (en los objetos a detectar o en la clase

"Fondo"). Por ejemplo, en Matlab los detectores se entrenan usando normalmente su propia red preentrenada añadiendo nuevas capas, pero siendo la salida de la última capa final solamente el número de objetos que se quiere detectar más una, que es la del Fondo [28] (como se ha visto en la sección 3.2.5.).

Cabe destacar que toda red base CNN ha de ser modificada para lograr en la salida un vector que posea las características de los objetos que posteriormente se pueda usar en la clasificación y localización. Para ello, hay que eliminar las capas finales puras de clasificación como son *softmax* y la propia salida de clasificación. Además, se intenta siempre que este vector de características de los objetos sea de las capas más avanzadas con el fin de que incluya el mayor aprendizaje jerárquico posible.

A continuación, se realiza una regresión lineal para adaptar el *bounding box* al *truth bounding box* mediante un algoritmo de regresión lineal de rectángulos (4 parámetros y un vector regresor) y se realiza para cada clase un muestreo de IoU, de la misma forma al explicado en la sección 3.2.5. :

- Se eliminan los de la clase artificial Fondo.
- Para cada clase se ordenan por resultado de probabilidad los *bounding boxes*.
- Empezando por el primero como fijo o base, se calcula el IoU del resto sobre este. Si supera un determinado valor de umbral, se considera que es el mismo objeto y se elimina.
- Se realiza este proceso sucesivamente hasta pasar por todos *bounding boxes* o ROI's ajustados por regresión.

Toda esta explicación se muestra resumida en la Figura 19.

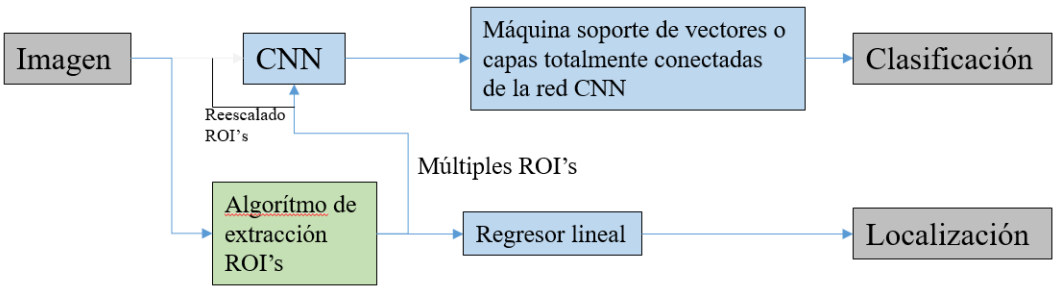

Figura 19. Esquema de la arquitectura de detectores RCNN

El mayor inconveniente que poseen los detectores RCNN es el alto coste computacional, ya que hay que pasar todas esas ROI por la red CNN preentrenada. Sólo hay que pensar en los comentados 2000 ROI's por defecto de Matlab para hacerse una idea de esto. Así, por este motivo, no se pueden utilizar detectores RCNN en aplicaciones que se necesiten hacer en tiempo real. Otro inconveniente, es el gran espacio de memoria que genera por la creación de muchos mapas de características de forma continua, que han de ser almacenados temporalmente en su ejecución.

Por último, destacar que el entrenamiento de los detectores también es un problema por su lentitud ya que tiene que:

- Ser capaz de ajustar parámetros de la red CNN preentrenada con las características particulares de los objetos a detectar.
- Entrenar los parámetros de la SVM o las capas nuevas añadidas a la red CNN (según la solución aplicada).
- Entrenar el vector de regresión del regresor de ROI's.

3.3.2. Fast-RCNN

Para explicar el siguiente tipo de detector que se quiere tratar en profundidad, Faster-RCNN [29], se ha de explicar Fast-RCNN [30], al ser éste su base.

Este tipo de detector consiste en una evolución de RCNN, que trata de reducir su cuello de botella principal: la aplicación de la CNN preentrenada a cada una de las ROI's. Para ello, lo que se hace es dividir el camino de computación en dos partes y separar la extracción de ROI's de la parte de extracción de características de las imágenes. Así, la parte de extracción de ROI's la sigue haciendo con la red CNN base, pero la extracción de características para la clasificación de objetos en la imagen se hace aplicando la red preentrenada CNN directamente a la imagen.

De esta forma, no se necesita escalado alguno sobre la entrada al modelo CNN y se aplica sobre la imagen entera, lo cual supone una gran ventaja. No obstante, la limitación de memoria y aplicación de las redes CNN hacen que, en la práctica, se reescale previamente la imagen. Por este motivo, herramientas como Matlab, realiza este reescalado de imágenes dentro del *training* de RCNN con la función "trainRCNNObjectDetector" y, para el training de Fast-RCNN, con la función "trainFastRCNNObjectDetector". No obstante, Matlab aconseja en los ejemplos de uso realizar un escalado previo de las imágenes antes de la ejecución de la función de training. Esto es importante, dado que, mientras la entrada al "trainRCNNObjectDetector", es sólo una tabla de imágenes y *bounding boxes*, a partir de "trainFastRCNNObjectDetector" permite el uso de estructuras específicas denominadas *datastores*, que tienen su ventaja en el tratamiento de imágenes en grandes conjuntos. Así pues, aunque en teoría el no uso del pre-escalado es una ventaja en el costo computacional, en la práctica se hace necesaria esta implementación para tratar de compensar el tener que hacerla fuera de la función o proceso del detector.

Volviendo al inicio, una vez aplicada la CNN sobre la imagen, se extraen los mapas de características, como si fuese la salida de una red convolucional estándar de las vistas en el apartado 2.3 previo, y se incluyen en una nueva capa llamada "ROI pool" junto con los ROI's extraídos, como se comentó anteriormente (SURF, SIFT, algoritmo de Edge Boxes, etc.). En esta capa, se aplica el mapa de características a cada región o área del ROI y después se realiza un *max-pool*. De esta forma, se produce una división o submuestreo como se realiza en [7]. Así, por ejemplo, si de la CNN sale un mapa de características de dimensiones H×W×K, con cada ROI de tamaño N×M, se establece una relación en la aplicación en la que se obtiene otro mapa de características de H'×W'×K con la relación de H'=H/N y W'=W/M.

Su principal ventaja es que los nuevos mapas de características pueden ser tratados con capas neuronales totalmente conectadas al tener el mismo tamaño todas [9].

Precisamente, los detectores Fast-RCNN sustituyen las SVM y se restringe a capas *softmax* de clasificación con otras capas previas de convolución o totalmente conectadas. Lo mismo ocurre con el regresor lineal, ya que ahora se aplica la regresión con unas capas especiales para tal uso. Cabe destacar, que todo esto se realiza gracias a la extracción de los mapas de características de las ROI's.

En la práctica, siguiendo el esquema de Matlab, el modelo base CNN se "abre" tras alguna capa de las convoluciones o grupos de convoluciones y esta se convierte en el "feature extraction layer". De esta forma, esta elección hecha en base a ensayo-error, afecta al rendimiento del detector.

Tal y como se ha comentado previamente, la eliminación de escalado y aplicación de CNN a cada ROI reduce enormemente la carga computacional. El uso de capas neuronales para clasificar y hacer regresión mejora la precisión media, ya que aumenta la generalización frente a las SVM y los regresores lineales. Por tanto, la velocidad y precisión final de la red supone una mejora frente a RCNN.

Por último, cabe mencionar de cara al training que es necesario ajustar en este caso los parámetros de la CNN y de las últimas capas creadas para la clasificación y regresión, pero ya no requiere de SVM o de los parámetros que realizan la regresión lineal. La principal ventaja es la eliminación de la extracción y aplicación de CNN sobre la imagen y que las capas de clasificación y regresión se pueden incorporar a la misma red, con lo cual son procesos similares. Por tanto, si bien esto último, en términos de computación, no es tan relevante, si lo es en términos de memoria.

Igual, que en el apartado anterior, toda esta explicación, se sintetiza en la Figura 20.

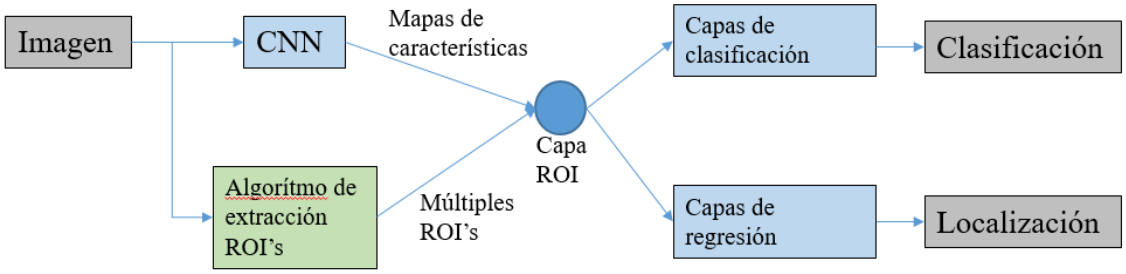

Figura 20. Esquema de la arquitectura de detectores Fast-RCNN

3.3.3. Faster-RCNN

Una vez explicado el modelo Fast-RCNN, Faster-RCNN se puede entender de forma sencilla, ya que la red es la misma, pero soluciona el problema de la generación con algoritmos SIFT, edge boxes, etc de las ROI's. Los detectores Faster-RCNN eliminan estos métodos e integran un bloque de red convolucional para la propuesta de regiones, cuya entrada es a su vez un mapa de características de la propia CNN que se aplica directamente sobre la imagen. Esta parte de la red se llama "Red para propuesta de regiones" (RPN), la cual estima la presencia de objetos en una imagen y los *bounding boxes* de cada uno.

Este proceso mejora al final el detector, ya que, por un lado, no se tiene que implementar un detector externo, realizándose todo dentro de la red (en una de sus ramas), lo que mejora la velocidad, y, además, la detección con la red RPN es más eficiente en cuanto a la generalización.

A su vez, esta red RPN lo que hace es un muestreo con deslizamiento de una ventana con el fin de encontrar los *bounding boxes*. Esto, en la práctica se realiza usando *anchor boxes* que sean los *bounding boxes* con parámetros a entrenar y a los que se aplican sobre los mapas de características de la red CNN base usada en el detector. Por cada *anchor box* se estima un número igual de ROI's. De esta forma, si bien la precisión no mejora sustancialmente con respecto a *Fast*, el tiempo y coste computacional sí que lo hacen.

Por último, en cuanto al entrenamiento, es necesario ajustar en este caso:

- Los parámetros de la CNN
- Los parámetros de capas nuevas de clasificación y regresión
- Los parámetros de esta nueva capa de extracción RPN

Por tanto, los tiempos serán algo mejores por el rendimiento computacional de detección de *Faster* referente a Fast-RCNN y RCNN como comentamos más arriba, pero no para el entrenamiento, ya que, por ejemplo, el detector externo que se sustituye en Faster-RCNN por la red RPN no precisa de entrenamiento.

De forma resumida, toda esta explicación puede verse en la Figura 21

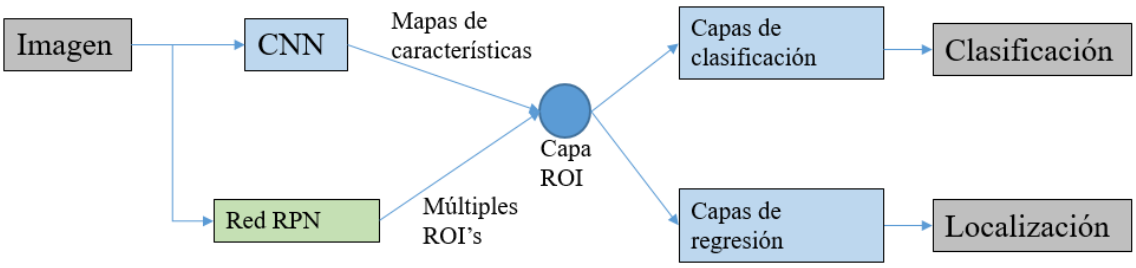

Figura 21. Esquema de la arquitectura de detectores Faster-RCNN

3.3.4. SSD

El detector *Single Shot Multibox Detector* (SSD) [31], como se ha indicado previamente, es un detector que mira sólo una vez (de un estado), al contrario que RCNN, Fast-RCNN y Faster-RCNN. La principal diferencia es, por tanto, que ya no se realiza una primera extracción de regiones propuestas (ROI's) y luego se realizan las "tareas" de clasificación y regresión de *bounding boxes*. Ahora, se realiza todo en una misma etapa. La Figura 22, muestra un esquema de este detector.

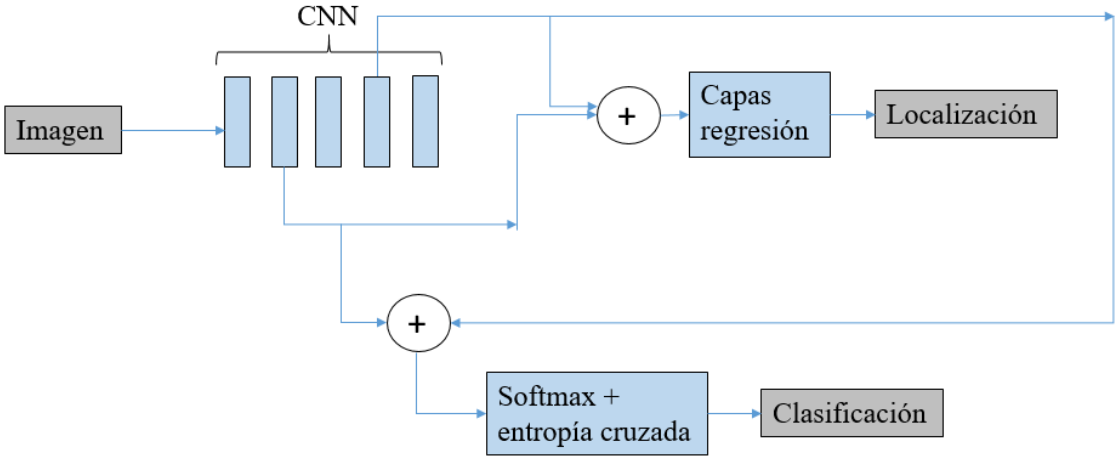

Figura 22. Esquema de la arquitectura de detectores SSD

Obviamente, esto representa una gran aceleración en la velocidad de detección y entrenamiento del detector, pero también una importante penalización en la precisión, que SSD y, por añadido el resto de detectores de un paso, intenta solventar con mejores o peores resultados.

En el caso de SSD, solventa este punto realizando múltiples detecciones en distintas escalas de la red. Explicado mejor, los detectores SSD parten de una red CNN preentrenada a la cual se le eliminan sus últimas capas totalmente conectadas como mínimo. Con esto, se pueden utilizar diferentes mapas de características generados en las distintas convoluciones de la red a diferentes tamaños o resoluciones. Con esto, lo que se pretende es que el detector se aproveche de las características a distintas resoluciones. Por ejemplo, los objetos pequeños van a ser mejor detectables en las primeras convoluciones, ya que la resolución del mapa de características permitirá adquirir correctamente sus características, pero en las convoluciones avanzadas será difícilmente posible esto, ya que habrán quedado comprimidos excesivamente en el mapa de características. Algo similar, pero desde el lado contrario, pasa en el caso de los objetos grandes, ya que, en convoluciones más profundas, es mayor el número de mapas de características por los que pasa y pueden aparecer más fácilmente sus propiedades para que sean correctamente extraídas. Por ejemplo, en la imagen a modo de ejemplo de la Figura 23 en resolución de imagen en 4×4 (ejemplo de resolución de un mapa de características de primeras convoluciones), estaría en 2×2, mientras que en 2×2 (ejemplo de resolución de un mapa de características de siguientes convoluciones), el objeto estaría en 1×1.

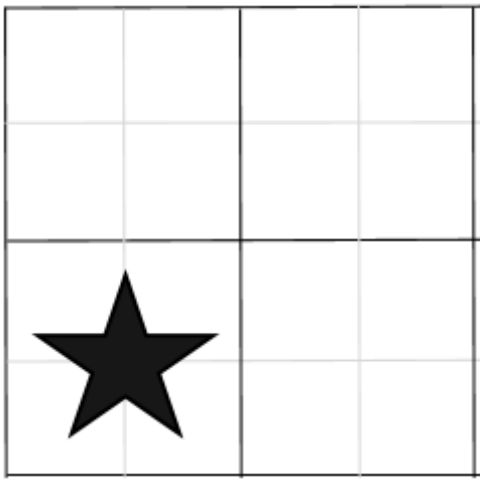

Figura 23. Ejemplo teórico de un objeto en una imagen con resolución 2x2 (líneas negras) y 4x4 (líneas grises)

Con esta premisa, es obvio que, en cada una de esas extracciones de mapas de características a distintas resoluciones, se realiza siempre una tarea de clasificación y una de regresión con el fin de conseguir los mejores resultados posibles, incluso en objetos pequeños.

Todas estas clasificaciones se introducen en una capa *softmax*, teniendo en cuenta la entropía binaria cruzada de la función de pérdida o error en ellas, con la que se obtendrá el error total de la imagen en la detección al final. Lo mismo pasa con las regresiones, donde todas se unen y se procesan mediante la norma L1.

Cabe destacar que esta estructura e idea permite el añadido de más capas de convolución, siempre hacia tamaños menores y extractores de mapas de características de distintos niveles. Esto reforzará el proceso de extracción y el desempeño del detector. Si bien, si los objetos son de pequeño tamaño, puede no proporcionar los mejores resultados.

En la práctica, por ejemplo, en Matlab, se añaden capas de clasificación y regresión sobre los extractores de la red base truncada CNN. Posteriormente, estos se unen en otra capa "Merge" y se aplica "softmax" y una "focal loss layer" (aplica la entropía binaria cruzada para separar entre clasificaciones y fondo) [32]. En cuanto a la regresión, se procede de la misma manera. Como es un proceso largo, Matlab incorpora una función "ssdObjectDetector" que simplifica este proceso.

Ahora bien, los detectores SSD incorporan otra opción, tanto en la detección como en el *training*. Esta es el uso de *anchor boxes* predefinidos. De esta forma en cada mapa de características se establece un conjunto de *anchor boxes* para la extracción. Estos *anchor boxes* generan la forma del objeto que, en verdad, sería la del *bounding box* final (*ground truth bounding box*). Así, para cada mapa de características se establecen clasificaciones y regresiones según sus *anchor boxes* asociados. Esto permite que se predefina un tamaño inicial de *bounding box* y sobre este se pueda inicialmente clasificar y realizar la regresión. Por este motivo, es obvio que a mayor número de *anchor boxes*, mejores precisiones se consiguen, pero mayores tiempos y coste computacional requieren.

Finalmente, cabe mencionar que de cara al entrenamiento es bastante más liviano en carga computacional, que las RCNN, Fast o Faster-RCNN, ya que sólo se ajustan parámetros de la red base CNN y las nuevas capas añadidas de clasificación y regresión por mapa de características y globales de la unión. Por tanto, el grueso del detector se sitúa en la red base CNN truncada y no tanto en añadidos de regresores lineales, SVM u otros como los detectores descritos previamente. Por este motivo, el uso de modelos CNN más compactas, pero con buenos resultados de precesión, como por ejemplo la ya vista *squeezenet*, proporciona también buenos resultados de rendimiento del detector

3.3.5. YOLO

YOLO, acrónimo de You Only Look Once [33], consiste en un algoritmo de detección de un estado, al igual que en el caso de SSD. En YOLO, se puede decir que han ido existiendo varias versiones del mismo algoritmo sobre el que se han ido introduciendo sucesivas mejoras para ganar velocidad y precisión. Es cierto, que esto mismo, también se da en los otros tipos de detectores de objetos tratados, donde se han creado variaciones específicas según el modelo CNN usada, realizando añadidos sobre la base, etc.

Pero es con los detectores YOLO donde se hace una mayor diferenciación y reconocimiento general de diferentes versiones de YOLO, de forma bien marcada y diferenciada, en parte porque la creación de cada versión es una mejora de la versión anterior y, parte, por tanto, de esta y de sus creadores originales. Así, se habla de YOLO V2, V3, etc.

En todas las versiones de YOLO, la base reside en lo implementado con YOLO V1. Así, de forma resumida se pueden ver las diferentes versiones como a continuación:

- YOLO V1. La base de YOLO consiste en la división ficticia de una imagen dentro en una rejilla de dimensión S×S. A cada una de esas divisiones se les llama celdas y en cada una se generarán distintos *bounding boxes* y valores de confianza de acuerdo con la presencia de un determinado objeto en ellos. De esta forma, se puede decir que cada *bounding box* generado guarda la información de las coordenadas x e y (posición de su centro con respecto a la posición local o relativa en la celda), de su ancho y alto respecto de la imagen completa en función de ancho y alto de celda y de la confianza. Por tanto, un término fundamental en YOLO es el de la confianza, la cual sería la probabilidad de que un *bounding box* contenga un determinado objeto o no, la cual será al final el IoU del *bounding box* contra el *ground truth*. Además, cada una de esas celdas contendrá la pertenencia a las distintas clases para las que se entrena el detector al haberse detectado este. Esto se trata de una probabilidad condicionada por la propia confianza del *bounding box* explicado previamente. Así, se procede seleccionando la probabilidad máxima que será la de esa celda. Cabe aclarar que la detección de un objeto en una celda se considera siempre que el centro de ese objeto este dentro de la propia celda. Así de forma resumida, YOLO V1 tiene algunas particularidades:
    - Si por ejemplo una imagen se divide en 9×9 celdas, habría 9×9×(5×Nºbounding_boxes+Nºclases_detectables).

    - En YOLO, al final, la estructura creada se traduce en una red basada en la regresión conjunta, al estar relacionando las coordenadas espaciales con la clasificación. Es decir, se elimina la diferencia de clasificador y regresor, pasando a ser en YOLO solo un regresor que usa tres pérdidas: de clasificación, de localización y de confianza.
    - Por último, destacar que usa la supresión no-máxima (*Non-max suppression*), explicado en 3.2.5., para eliminar el problema de detección de un mismo objeto por varias celdas adyacentes.

- YOLO V2 [34]. Introduce unas siete modificaciones con el fin de mejorar la precisión y la velocidad. Las principales consisten en el uso de la normalización de valores de la imagen (de los valores de píxeles de la imagen) lo cual mejora la convergencia y reduce la complejidad de la arquitectura necesaria en la red, el uso de *anchor boxes* predefinidos que permiten trabajar de manera similar a cómo funcionan los anchor boxes en Faster-RCNN para tener unos *bounding boxes* predefinidos y no partir de cero (por explicarlo de una forma simplificada), el trabajo a diferentes escalas o el uso de una mayor resolución de imágenes para la mejora de detección de objetos de pequeño tamaño.

- YOLO V3 [35]. Se trata de una evolución de YOLO V2 con el fin principal de mejorar la velocidad de decisión. Los tres puntos principales que se añaden son el uso de extracción de características de diferentes capas a distintas resoluciones (como se hacía con SSD) con lo que mejora la capacidad de detección, la eliminación de capa *softmax* por clasificadores logísticos que mejoran la velocidad y permiten la identificación o *training* de objetos pertenecientes a dos clases y el uso de regresores logísticos para el cálculo de la probabilidad de detección de un objeto en el *bounding box*. En este tipo de detectores vuelve a ocurrir como en SSD, ya que las capas elegidas para conseguir el multiescalado es de forma manual, con lo que afecta al desempeño de la red y provoca que se tenga que realizar pruebas ensayo-error.

- YOLO V4 [36]. Es la siguiente evolución respecto a YOLO V3. Aquí los principales cambios residen en que se incorpora una capa adicional entre la red CNN preentrenada y la consiguiente parte del detector ya explicada para YOLO V3. De esta forma, se incorpora un bloque o cuello realizado con Spatial Pyramid Pooling, SPP [37], o PaNeT o ambos (como tiene integrado Matlab). Bien sea uno u otro, actúan extrayendo características de distintas capas de la red base y concatenándolas según el modelo, proceso al que no se entra a explicar esta memoria. Concretamente, se aplica tras las capas de *Max-pooling* en diferentes niveles. Al final, estas concatenaciones aumentan la precisión, sobre todo de los objetos pequeños.

3.4. Métricas para evaluación de detectores

Tanto para determinar el comportamiento mejor o peor de un detector frente a unas imágenes de entrada o para comparar entre sí distintos detectores, es necesario definir una serie de métricas. En el caso de los detectores se basan siempre en el IoU y en el hecho sencillo de si una detección es correcta o no en virtud del objeto. El IoU se ha explicado en la sección 3.2.3. y la detección correcta se basa en las siguientes situaciones:

- Verdaderos positivos (VP): Son las detecciones que resultaron totalmente correctas. Es decir, que se detectó un objeto y efectivamente su clase es correcta y el IoU supera el umbral mínimo (ver apartado 3.2.4)
- Verdaderos negativos (VN): Son las veces donde no se produjo una detección y efectivamente no había objeto. Es decir, se generó un *score* menor que el umbral siendo correcto al no superar el umbral de IoU o no ser de la clase. En la práctica no tiene tanta importancia ya que no se tienen en cuenta en las métricas. Obviamente Verdaderos negativos hay cientos en cada imagen donde se aplicase un cierto detector.
- Falsos positivos (FP): Son las detecciones donde se detecta un objeto, pero este no está presente. Se tiene un *score* por encima del umbral, pero no existe objeto. Es decir, el IoU no supera el umbral o la clase predicha no es correcta. Por tanto, se detecta un objeto que no está presente parcial o totalmente o lo detecta como perteneciente a otra clase al que es en realidad
- Falsos negativos (FN): Son las veces donde no se produjo detección, pero sí que tenía que haberla habido. Es decir, se detecta un *score* por debajo del umbral donde se tenía que haber predicho un objeto

Para entender mejor estas definiciones supóngase el ejemplo ilustrativo de la Figura 24, sólo desde el punto de vista teórico con la detección de un objeto de una única clase. Hay un conjunto de datos de test de tres imágenes, de forma que en la primera imagen hay 3 objetos donde cada uno viene identificado con su *ground truth bounding box*. En la segunda imagen hay dos y en la última una. De esta forma, habrá en total de seis objetos posibles a detectar. De la misma forma, tal y como se puede ver en el ejemplo de la Figura 24, un cierto detector realiza las detecciones del rectángulo azul con los confidence *scores* correspondientes para cada una.

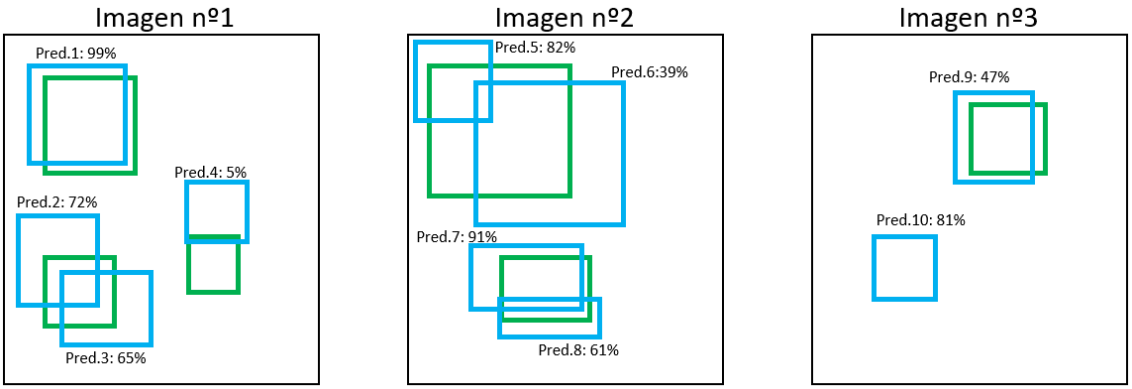

Figura 24. Ejemplo teórico de detecciones en 3 imágenes con los objetos reales indicados por los rectángulos verdes y las detecciones por los rectángulos azules (siendo "Pred." el confidence score de cada uno)

Así, identificando los resultados según lo visto previamente al principio de esta sección y teniendo en cuenta un umbral de 0,6 para el *score* y uno de 0,8 para el IoU, se obtienen los resultados mostrados en la Tabla 6.

Detección	***Score* (%)**	**IoU (%)**	**Análisis del resultado**
1	99	91	VP
2	72	75	FP
3	65	82	VP
4	30	5	FN
5	82	19	FP
6	39	68	FN
7	91	89	VP
8	61	32	FP
9	47	86	FN
10	81	0	FP

Tabla 6. Tabla resumen de detecciones del ejemplo teórico de la Figura 24

A partir de estas definiciones, tendremos diferentes métricas que individualmente o combinadas (para formar otras), proporcionan las métricas necesarias para la comparación de detectores.

3.4.1. Precisión y recall

La precisión es el número de detecciones correctas entre el número total de detecciones. Por ejemplo, si se realizan 50 detecciones correctas en la comprobación de un detector por medio de un conjunto de imágenes de test y en total se hacen 60, la precisión será 50/80=0,625. Su expresión es:

$$Precisión = \frac{VP}{VP + FP} \quad \textbf{(3)}$$

Se puede ver cómo las veces en que había funcionado el detector, detectando un objeto, este lo hizo correctamente y no detectó falsamente objetos con una clase que no era la

suya u objetos con un *bounding box* que casi no intersecta con el objeto en cuestión o directamente que no hay objeto y es fondo.

Mientras tanto, el *recall*, es llamado en la traducción al español con diferentes nombres como sensibilidad o exhaustividad [9], recuperación [38] o recuerdo [39]. Por ese motivo, se usa en esta memoria el nombre original en inglés, *recall*. En definitiva, el *recall* se define como el número de detecciones correctas entre el número total de detecciones de ese objeto que tendría que haber habido. Por ejemplo, si se realizan 50 detecciones correctas en la comprobación de un detector por medio de un conjunto de imágenes de test y en total se tendrían que haber hecho 100, porque el objeto apareció 100 veces y por tanto había 100 *ground truth bounding boxes*, la precisión será 50/100=0.5. La expresión es como sigue:

$$Recall = \frac{VP}{VP + FN} \quad \textbf{(4)}$$

Se puede ver cómo las veces en que había funcionado el detector, detectando un objeto, este lo hizo correctamente frente en las que tenía que haber funcionado realmente.

Al final, la precisión es una métrica que indica el porcentaje de los objetos detectados correctamente frente a detecciones fallidas, mientras que el *recall* indica el porcentaje de detecciones correctas totales, es decir, cómo de bien el detector consigue detectar ese objeto.

3.4.2. Curva precisión-recall

La curva de precisión-*recall*, Figura 25, es un diagrama de líneas donde se coloca en ordenadas la precisión y en abscisas el *recall*. Es una gráfica muy útil ya que a simple vista muestra el rendimiento de un detector, al permitir observar cosas como que si la precisión se mantiene alta cuando el *recall* sube, se trata de un detector con un buen comportamiento y eficaz. La clave es que la precisión y el *recall* suelen tener un relación inversa en la mejora de una frente a la otra [40].

Esto se puede ver de una forma sencilla. Si el umbral para decidir si se detecta un objeto o no en base al *confidence score* aumenta, se tenderá a detectar más veces objetos de forma correcta, es decir, mejorará su precisión. Sin embargo, habrá muchas veces donde no se realizarán detecciones porque el *score* no es lo suficientemente alto y sí que había objeto, es decir, su *recall* bajará. De forma inversa, bajar el umbral dará más detecciones, pero no todas tienen porqué ser correctas.

Así, el *recall* puede aumentar o mantenerse, dependiendo de cómo sean de correctas esas detecciones nuevas que se han hecho, pero no disminuir ya que al final el número total de objetos a detectar es siempre el mismo.

De esta forma, por ejemplo, puede haber detectores que tengan una alta precisión, pero un bajo *recall*, que viene a significar que el detector funciona pocas veces, pero cuando lo hace es muy seguro en la detección del objeto. En caso contrario, puede haber detectores con alto *recall*, pero baja precisión, que realiza muchas detecciones pero que tiene una mayor probabilidad de detectar el objeto de forma inadecuada con su *bounding*

box, en sitios donde no hay objeto incluso o errando la clase de ese objeto y confundiéndola con otra Tabla 7.

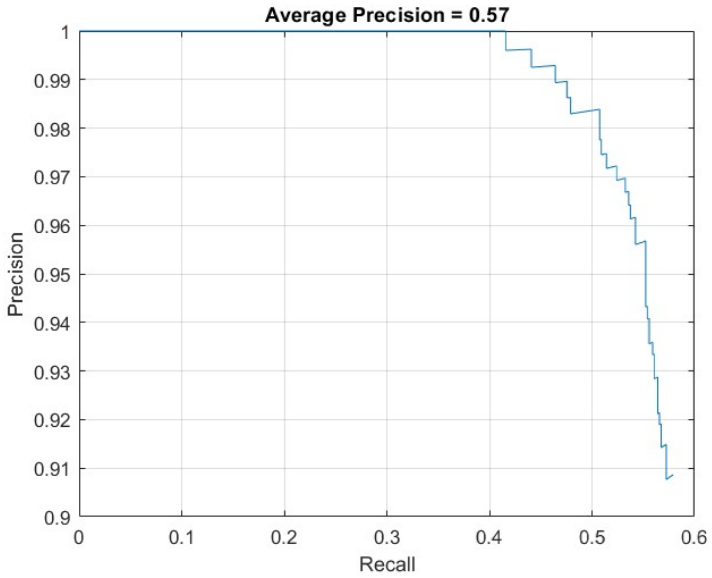

Figura 25. Curva de precisión-recall de un detector SSD

Para entender mejor cómo se forma esta curva sígase el ejemplo mostrado al principio de esta sección al hablar de la definición de verdadero positivo y negativo y falso positivo y negativo, donde había 6 objetos en las imágenes y 10 detecciones. Tomando la Tabla 6 como referencia y limpiando los valores de verdadero negativo y falso negativo (para el cálculo del *recall*), se crea el acumulado de precisión y el *recall*, ya que es lo que se dibuja en la curva precisión-*recall*: la precisión según el valor de *recall* correspondiente (Tabla 7). En esta representación siempre se parte de precisión valor 1 y *recall* valor 0.

Detección	***Score* (%)**	**IoU (%)**	**Análisis del resultado**	**VP acumulado**	**FP acumulado**	**Precisión**	***Recall***
1	99	91	VP	1	0	1	0,17
2	72	75	FP	1	1	0,5	0,17
3	65	82	VP	2	1	0,67	0,33
4	30	5	FN	2	1	0,67	0,33
5	82	19	FP	2	2	0,5	0,33
6	39	68	FN	2	2	0,5	0,33
7	91	89	VP	3	2	0,6	0,5
8	61	32	FP	3	3	0,5	0,5
9	47	86	FN	3	3	0,5	0,5
10	81	0	FP	3	4	0,43	0,5

Tabla 7. Tabla resumen de detecciones del ejemplo teórico de la Figura 24 ampliada con la precisión y recall

De la misma forma, trasladando estos valores a un gráfico con curva de precisión-*recall*, se tienen los resultados de Figura 26.

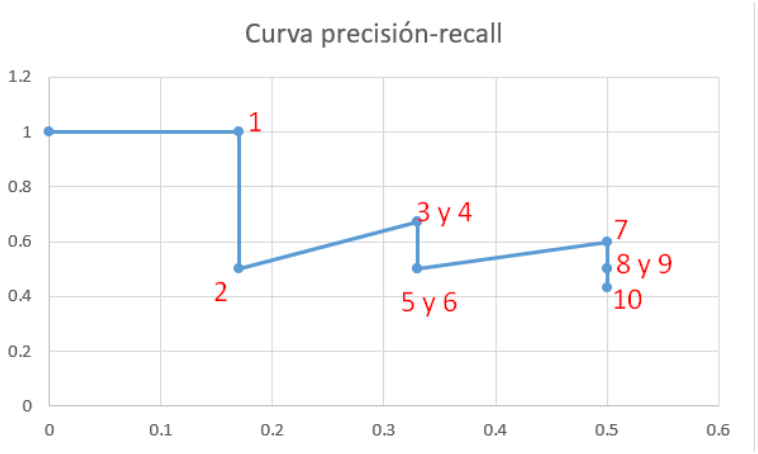

Figura 26. *Curva precisión-recall del ejemplo teórico de la Figura 24*

3.4.3. Precisión promedio y valor medio de la precisión promedio

La precisión promedio (AP, *Average Precision*) es el área bajo la curva precisión-*recall*. Sería por tanto la integral de los valores de precisión en el rango de valores de *recall* desde 0 a 1 (su mínimo y su máximo).

En la práctica, para no calcular integrales se desarrolla la habitual suma de pequeñas áreas, dividiendo el rango del *recall* en *n* mini-rangos y tomando el valor de precisión máximo dentro de ese mini-rango.

Siguiendo el ejemplo de la curva de precisión-recall creada anteriormente, se puede calcular la discreta con el rango mayor como se muestra en la Figura 27. Al final el área que encierra y, por tanto, el AP es igual a 0,38.

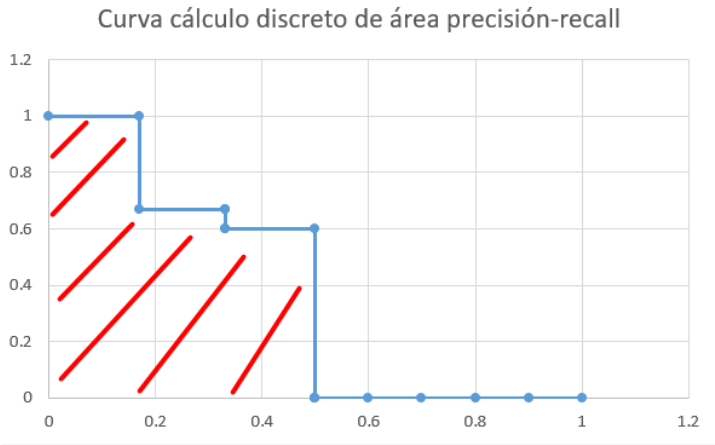

Figura 27. Curva precisión-recall con cálculo discreto del ejemplo teórico de la Figura 24

La precisión promedio (AP) resulta ser la métrica principal para simplificar la calidad de un detector, ya que se trata de un único valor que da muestra de todo el detector. Ahora bien, esta precisión promedio es para cada clase. Si hay más de una clase y se quiere tener un único valor que muestre el desempeño del detector, se ha de calcular el valor medio de la precisión promedio (mAP), que es al final la media de las precisiones promedio de

cada clase de objeto. Este mAP es la principal métrica que se muestra en tablas y gráficas donde se comparan distintos detectores entre sí, sobre todo detectores entrenados con conjuntos de datos conocidos.

En el ejemplo previo al haber sólo una clase de objeto para el detector, mAP es igual a AP y, por tanto, mAP es 0,38.

3.4.4. Matriz de confusión

Se trata de una sencilla disposición, en forma de tabla, del número de VP, VN, FP y FN, Tabla 8. Su verdadera utilidad reside en la aplicación a detectores de varias clases, donde se aplican colores escalados todos igual en cada clase.

De esta forma se crearán tantos mapas de confusión como clases y se podrán usar como mapas de calor para ver con un vistazo rápido para que clases hay más detecciones correctas, para cuales se "confunde" más las detecciones de esa clase (porque se detecte otra cosa, por ejemplo). En resumen, una matriz de confusión tendría un aspecto similar al de la Tabla 8.

		Detección	
		Positiva (hay detección)	Negativa (no hay detección)
Presencia real del objeto	Positiva (hay objeto realmente en esa localización)	Verdadero positivo (VP)	Falso negativo (FN)
	Negativa (no hay objeto realmente en esa localización)	Falso positivo (FP)	Verdadero negativo (VN)

Tabla 8. Esquema de la distribución de datos de la matriz de confusión

3.4.5. F-1 *score*

Esta métrica se calcula también a partir de la precisión y el *recall*, pero en lugar de tratarse por separado, incluye a ambas. Se trata de la media del valor ponderado de precisión y *recall* con la siguiente expresión,

$$F1\ score = 2 .* \frac{Precisión * Recall}{Precisión + Recall} \quad \textbf{(5)}$$

Un valor alto de F1-*score* implican valores altos de precisión y *recall*, mientras que bajos indican desequilibrio entre ambos [41].

4. Recursos utilizados

A continuación, se describen las principales herramientas utilizadas en la realización del trabajo, tanto desde el punto de vista software como de hardware. Se explicará no sólo de forma breve de qué se está hablando, sino que también se profundizará en los detalles y particularidades de cada una. Igualmente, se hablará de la integración entre ellas.

4.1. Matlab

Matlab [42] es una plataforma para el cálculo y computación integrada en distintos sistemas operativos como Windows y Linux y con herramientas disponibles en la nube a través de Matlab Cloud u Online. Una de las principales virtudes de Matlab es su extensión y aplicación a multitud de ramas del conocimiento donde sus capacidades de cómputo pueden resultar de utilidad. Así, se pueden encontrar utilidades desde economía hasta visión por computador como parte de la inteligencia artificial, que es el caso que más nos interesa. Si bien, dispone de una gran base con funciones ya creadas que simplifican la generación de funcionalidades desde cero o la copia de repositorios de código para ganar tiempo, son funciones genéricas. Para aplicaciones más particulares Matlab ha buscado la solución a través de *Add-Ons* instalables según necesidad. Con esto, se consigue la escalabilidad según lo necesario y se evita un consumo de ocupación de memoria en el disco excesivo (siempre hablando de la versión escritorio). La Tabla 9 recoge los *Add-Ons* instalados en la versión usada de Matlab R2022b:

Funcionalidades (toolboxes) de Matlab utilizadas
Automated Driving Toolbox version 3.6
Computer Vision Toolbox version 10,3
Computer Vision Toolbox Model for YOLO v3 Object Detection version 22.2.0
Computer Vision Toolbox Model for YOLO v4 Obiect Detection version 22 2 0
Aprendizaje profundo Toolbox version 14.5
Aprendizaje profundo Toolbox Model for AlexNet Network version 22.2.0
Aprendizaje profundo Toolbox Model for ResNet-18 Network version 22.2.0
Aprendizaje profundo Toolbox Model for ResNet-50 Network version 22.2.0
Image Acquisition Toolbox version 6.7
Image Processing Toolbox version 11.6
Neural Network Toolbox version 1.2
Parallel Computing Toolbox version 7.7
Statistics and Machine Learning Toolbox version 12.4

Tabla 9. Lista de Add-Ons de Matlab utilizadas

Otra de las particularidades que posee Matlab es la opción de uso de Matlab Online en la nube con una cuenta de Mathworks (nombre de la compañía propietaria de Matlab). En esta opción ya se encuentran todos los *Add-Ons* disponibles y no es necesaria selecciones individuales. No obstante, aunque se exploró esta vía para el desarrollo práctico del presente proyecto, para tener mejores velocidades que en el ordenador local usado, no se pudo llevar a cabo, ya que, si bien, Matlab Drive, la herramienta de la nube para el guardado de archivos para Matlab Online permitía almacenar el conjunto de usado para el training, en la práctica, el manejo de un conjunto tan grande de imágenes producía bloqueos continuos en la ejecución.

Uno de los puntos desfavorables en Matlab es que posee un lenguaje propio para la programación en los archivos ejecutables, *scripts*. Otro de los puntos, quizás el peor, se entiende a partir de la siguiente explicación. Matlab incorpora tanto en el programa general, como en los diferentes *Add-Ons*, multitud de funciones creadas para diferentes resoluciones y ejecuciones y posee una sólida serie de ejemplos bien documentados y apoyados en una sencilla explicación teórica. Esto, obviamente es una ventaja ya que las funciones, ejemplos y manual de funcionamiento se hayan centralizadas en la página de Mathworks. Sin embargo, se convierte en un gran problema si esas funciones no se realizan con los tipos de datos elegidos, se quieren realizar diferentes implementaciones, etc. Este es quizás el mayor inconveniente de Matlab, el cual en la creación de herramientas muy genéricas modificadas en ejemplos muy concretos crea muchos problemas en el desarrollo y compilación correcta de los programas. En parte, viene motivado por el lenguaje de programación y la falta de una comunidad donde se puedan resolver dudas de forma ágil. Por ejemplo, apenas hay desarrollo de programas en la conocida plataforma GitHub. Si bien, posee un repositorio propio de usuarios de Matlab, denominado Matlab Central, donde la comunidad científica ofrece soluciones resueltas a diversos problemas. De la misma forma, las funciones tienden a ser genéricas, aun en las integradas dentro de las *Add-Ons,* y al estar desarrolladas para distintas funcionalidades, que no distintas formas de resolución o posibilidades de uso de datos. Todo esto conlleva, al final, a una excesiva generalización de funciones que crea también, a su vez, un excesivo uso de funciones concatenadas. Esto dificulta mucho el seguimiento del script, sobre todo para la comprobación de errores de compilación o ejecución y complica el encontrar los errores. Aunque siempre existe la herramienta *Debbuger* para resolver esta dificultad, en un entorno muy amigable con visualizaciones directas de los valores de las variables.

De esta manera, en lo relativo a la parte que ocupa la memoria de la detección de objetos, los ejemplos guía de las funciones son sólidos y con conceptos básicos explicados, pero se ajustan a casos sencillos. Por ejemplo, mismamente, una simple función correspondiente a "*data augmentation"* con unos datos distintos a los del ejemplo, hace que no funcione la función de training de los detectores YOLO, SSD y Fast-RCNN al estar hecha con imágenes genéricas, centradas, con buena visión del objeto. Por tanto, se pueden considerar "fáciles" para la detección. Así, se tuvieron que crear funciones de *data augmentation*, de preprocesado de las imágenes, de transformación a tres canales, etc, además de modificar las existentes. Otro ejemplo concreto de la generalización de Matlab, es el fallo que ocurre en la compilación en Matlab por tener una categoría "*Stop*

signs" haciendo uso de la función de entrenamiento de detectores RCNN "trainRCNNObjectDetector". El motivo es simplemente que la salida de la red en las categorías/*labels* es una variable de tipo *struct* y, esta variable, no permite espacios. Sin embargo, en otros detectores no se usa *struct*.

De esta misma forma, Matlab no incorpora funciones especiales o está perfectamente preparada para el uso con *Datasets* de imágenes, tal y como veremos en la sección siguiente.

Teniendo en cuenta el planteamiento explicado, una parte importante del trabajo consistió en la selección, modificación y pruebas en bloques pequeños de los scripts creados para evitar el problema de búsqueda del error en múltiples funciones encadenadas, tal y como se comentó anteriormente.

Al margen de este asunto, la implementación de los detectores en Matlab posee unas mismas fases para todos los tipos posibles que permite:

1. Una primera fase donde se realiza la carga o captura de datos del *Dataset* o conjunto de datos.
2. Una segunda donde se procesan o convierten las imágenes y *bounding boxes* extraídos del *dataset* para adecuar su información al tipo de variable que usa el detector y al propio funcionamiento del detector. Del mismo modo, se forman los grupos de *training*, *test* y *validación* y se realizan conversiones de datos.
3. Una tercera fase donde se crean los detectores a partir de funciones específicas de Matlab y modificaciones propias incorporadas.
4. Una cuarta de indicación de opciones para el entrenamiento.
5. Una quinta donde se lanza el entrenamiento a partir de funciones propias y específicas de Matlab.
6. Una última fase donde se extraen las métricas.

Por último, cabe destacar que para el uso de los detectores en Matlab se intentó utilizar en todos los casos un tipo de variable propia de Matlab que son los *datastores* (si bien los detectores RCNN en Matlab no permiten el uso de este tipo de estructuras). Estos son en verdad almacenes con recopilaciones de datos y/o imágenes, usado para casos en los que el volumen de esos datos/imágenes sea tal, que ocupe un gran peso en la memoria. Este es el caso de las imágenes del dataset COCO usado en este proyecto, donde existe un elevado número de imágenes a procesar, aplicar funciones, dividir en grupos, etc. Así se crearon "imagedatastores" y "boxLabelDatastore" para imágenes y *bounding boxes* respectivamente, así como combinaciones de ellas. Su uso pasa por llamadas a funciones de transformación para guardar la trazabilidad de cambios y código aplicado sobre ellas. La reducción en velocidad es drástica y, de hecho, en el presente proyecto, con los detectores RCNN, los cuales no admiten *datastores* para el entrenamiento, se decidió estructurar el tratamiento preliminar del *dataset* en tablas, que se convirtiese a *datastore* como en los otros detectores y luego volver a pasarlo a tabla para usarlo en el detector. Todo esto, realizado con funciones de conversión de tipos de variables de Matlab.

El segundo tipo de variables principales usadas ha sido *cell* (arreglos de celdas), la cual contiene bloques de texto, de letras y números o de arreglos numéricos [43]. Por último,

el tercer tipo principal de variables son las tablas, las cuales permiten almacenar en formato tabla convencional datos numéricos en las columnas.

Estas son las principales variables usadas ya que para distintas partes de los algoritmos creados se necesita en ese tipo concreto debido a las variables creadas. Por este motivo, se usan variables de conversión entre tipos como cell2table, etc, y en otros casos se tuvieron que modificar funciones con problemas, haciendo que funcionasen correctamente. Un ejemplo ha sido la función "validateInputData_tr_tr": modificación de la función ya modificada de Matlab para detectar *bounding boxes* incorrectos, pero adaptada a tabla.

No obstante, en el ámbito de las redes neuronales convolucionales, Matlab permite importe de redes y gráficas de capas de TensorFlow™ 2, TensorFlow-Keras, PyTorch®, el formato de modelos ONNX™ (Open Neural Network Exchange) y Caffe. También puede exportar redes de Deep Learning Toolbox™ y gráficas de capas al formato de modelos TensorFlow™ 2 y ONNX™.

También posee dos herramientas interactivas como Image Labeler o Video Labeler para el etiquetado propio de imágenes a todos los niveles, incluyendo las necesarias anotaciones en el caso de los detectores de objetos.

4.2. COCO *Dataset*

Como se ha mencionado, una de las particularidades de los detectores de objetos es que necesitan una cantidad bastante grande de imágenes para el entrenamiento, bien generadas propiamente o bien provenientes de un *dataset* externo. Existen otras muchas bases de conjuntos de imágenes, bien de varias clases distintas entre sí, como Imagenet, Google's Open Images, PASCAL VOC u otras más específicas, de una única clase o de varias clases con relación entre sí, como PCB Defect (para defectos en placas base donde detecta 6 tipos distintos) o MNIST (con detección de dígitos escritos a mano).

En el caso del presente trabajo, se seleccionó Microsoft COCO *dataset* (MS COCO), abreviatura de Microsoft Common Objects in Context [31], bajo licencia Creative Common 4.0, Figura 28. Se trata de un *dataset* que contiene en la totalidad de las diferentes versiones y según sus creadores, 91 clases diferentes de objetos y con más de 200000 imágenes etiquetadas. El etiquetado de los objetos en las imágenes no se hizo marcando sólo el *bounding box* correspondiente, sino que se marcan los objetos en una segmentación de los objetos sobre el fondo, lo que requiere mucho más trabajo manual de etiquetado de imágenes y segmentación que simples *bounding boxes*.

MS COCO se puede usar en la detección de objetos, la segmentación clásica de píxel según la clase, *panoptic segmentation*, *keypoint* y *densepose task*.

MS COCO *dataset* fue originariamente creado en 2014 y desde entonces se fueron añadiendo más y más imágenes en las sucesivas publicaciones anuales. La última fue la del 2017, que es la usada en este proyecto. Para la detección consta de 118287 imágenes que tienen que ser descargadas, ocupando una memoria de casi 18GB. Aparte, han de

descargarse las anotaciones de 241MB que contienen 886284 anotaciones en total en esas imágenes. Ambos ficheros pueden verse en la siguiente Figura 28.

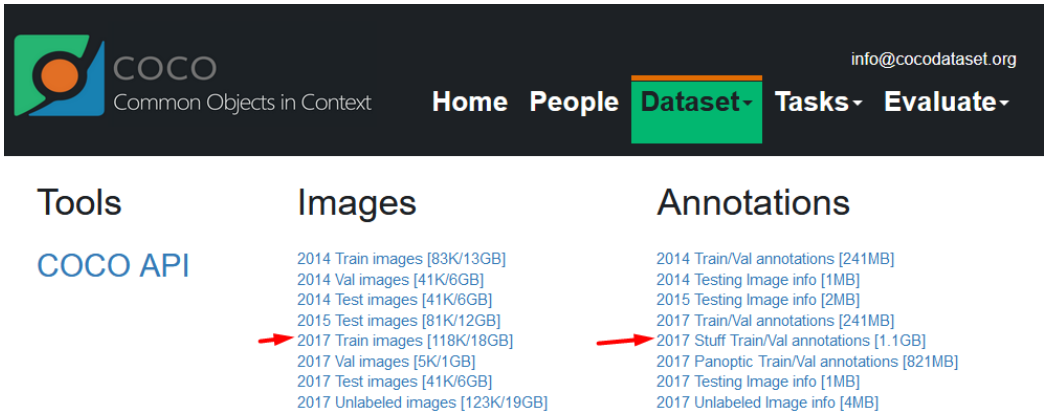

Figura 28. Localización de los dataset de Microsoft COCO usados para este proyecto (captura de la página https://cocodataset.org bajo licencia Creative Common 4.0)

MS COCO está asociado con FiftyOne que es un programa de distribución libre para visualizar todos los datos de COCO con el fin de poder llevar a cabo una evaluación de imágenes y filtrado de las mismas, Figura 29. El problema es que solo está desarrollado en Python y el Sistema operativo usado en el ordenador base, como se verá más adelante en la sección 4.4. , es Microsoft Windows®. Por este motivo, se usó el visor propio de la página web donde se puede buscar por la clase del objeto deseado y se puede ver fácilmente el número total de imágenes para esa clase de objeto y diferentes ejemplos de imágenes. Un ejemplo de esto se puede observar en la siguiente Figura 29.

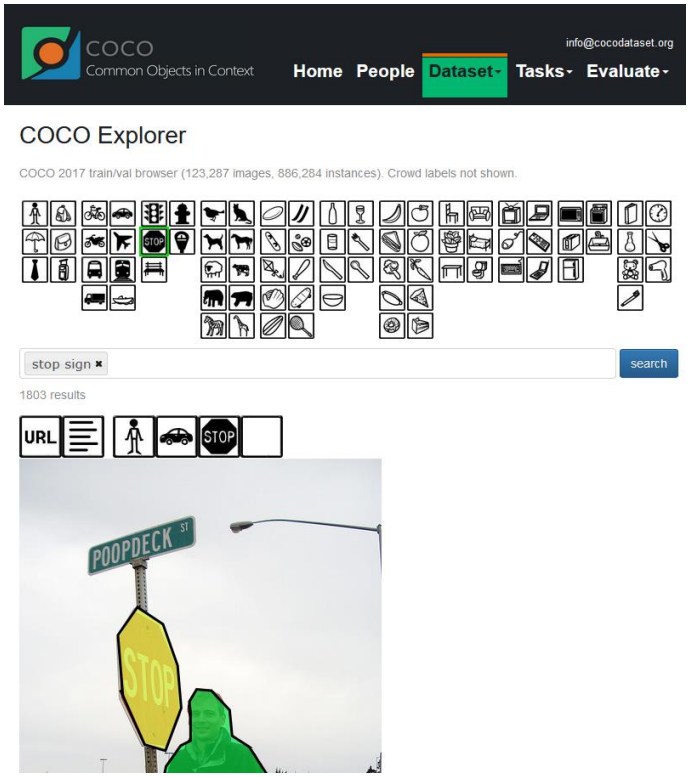

Figura 29. Ejemplo del visor de la web de Microsoft COCO para la clase "stop sign" (captura de la página https://cocodataset.org bajo licencia Creative Common 4.0)

Cabe destacar que, a partir de este *dataset*, se organizó, al igual que ocurre con otras bases de datos, un concurso de detectores, el MS COCO *challenge*, pasado a ser en los últimos años al LVIS 2021 Challenge and Workshop. Al final, se usan estos *dataset* para la comparación de detectores en base a las métricas de evaluación introducidas en el apartado 2.4. (principalmente mAP) así como la velocidad.

En cuanto a su estructura, las imágenes son imágenes en formato .jpeg que pueden ser leídas directamente en Matlab, pero no así sus anotaciones, ya que están escritas en JSON (*JavaScript Object Notation*) y Matlab es incapaz de leerlas directamente, aunque posee funciones específicas para ello. Dentro de esas anotaciones, hay información de la imagen con la que está relacionado por medio de un id y el nombre de la imagen (también incluye su resolución de píxeles), la clase de los objetos etiquetados en esa imagen por medio de un id (para tomar un número de referencia común de búsqueda para todos los de esa clase), información del nombre de la clase (perros, coches, camas, ….),el agrupamiento de esa clase en una categoría superior (animales, vehículos, muebles,…) [44] e información del *bounding box* del objeto relacionado con la anterior clase, así como posiciones de los objetos en las clases con sus coordenadas correspondientes (x_1, y_1, x_2, y_2) y con el parámetro "*iscrowd"* se indica si hay más de un objeto que no se pueden segmentar cada uno por separado.

En cuanto a la integración con las plataformas de cálculo, COCO dispone de una API (Application Programming Interface) para realizar dicha comunicación. Concretamente dispone de un COCO API para Matlab, Python y Lua, las cuales se han de descargar de la página web de MS COCO *dataset* con un enlace a la plataforma GitHub [45]. En el

caso del presente trabajo, al ser realizado en la plataforma Matlab, se ha de descargar dicho archivo que contiene las funciones de la API, guardarla en la memoria del ordenador y desde Matlab, crear una variable que apunte a la carpeta donde se ha guardado esta con un puntero a las rutas de trabajo. De esta forma, por medio de la función "CocoApi.m" junto con la dirección exacta del archivo de anotaciones de MS COCO, en este caso "instances_train2017.json", se consigue el acceso desde Matlab al *dataset* de COCO. Así, una vez realizada esta operación, ya se puede tener acceso a búsquedas en las anotaciones del número de identificación de imágenes relacionadas con las categorías de interés (con "getCatIds"), de las imágenes (con "getImgIds"), etc.

Una vez realizado esto, Matlab necesita convertir esa información de ficheros .json a ficheros .mat, lo cual se realiza mediante funciones de llamadas a las imágenes por categorías, para extraer el nombre de las propias imágenes, los *bounding boxes*, las etiquetas y la máscara, las cuales se guardan en un archivo .mat mediante varias variables conteniendo esta información.

4.3. AWS

AWS (Amazon Web Services) [46] es una plataforma que ofrece diferentes servicios de computación en la nube (cálculo, almacenamiento, procesos de inteligencia artificial, gestión de bases de datos, etc.). Su acceso se realiza online o a través de una consola propia y mediante una cuenta asociada. Se pueden solicitar diferentes servicios individualmente, según determinadas condiciones deseadas y bajo un precio establecido por hora, ocupación, etc. Su propietario es Amazon.com, inc. y fue lanzada inicialmente en 2006. Su competencia podría decirse que son Microsoft Azure, Google Cloud Platform, etc.

Uno de los servicios de los que dispone es el de las llamadas Amazon EC2 (Amazon Elastic Compute Cloud). Amazon EC2 permite crear máquinas virtuales de distintos sistemas operativos y de múltiples capacidades. Al final, EC2 permite lanzar aplicaciones a través de acceso web en las máquinas virtuales creadas. Así, cualquier usuario de EC2 está creando una arquitectura o imagen a partir de la máquina virtual creada y seleccionada. En EC2, dicha arquitectura recibe el nombre de instancia y puede contener cualquier aplicación o programa que se desee. La clave de esta rama dentro de AWS, es que cualquier usuario puede crear, traspasar, finalizar o lanzar las instancias deseadas (en verdad hay límites para el control del uso sobre los que se pueden pedir aumentos) y, a cambio, pagar una cantidad por hora en función de las capacidades de computación, sistemas operativo y localización del servidor donde se desarrollan. Se apoya en las tecnologías de virtualización, permitiendo utilizar gran variedad de sistemas operativos a través de sus interfaces de servicios web, personalizarlos, gestionar permisos de acceso a la red y ejecutar tantos sistemas como admita.

En las instancias EC2 hay multitud de máquinas virtuales con posibilidad de ser creadas, las cuales se agrupan en familias según sus características generales, sus usos finales o su tecnología de tarjeta gráfica. De esta forma, existen instancias EC2 desde la básica familia T2 (la instancia gratuita y base de AWS), hasta familias especiales para trabajos de

aprendizaje profundo con GPUs (Graphics Processing Units) de última tecnología, las cuales aceleran enormemente el tratamiento de las imágenes en el entrenamiento e inferencia, como G4, G5, P3, P4 (aunque ya no son gratuitas). Para este trabajo se crearon dos instancias, una primera con *g4dn.xlarge* con arquitectura Windows y una segunda con *g5.2xlarge* para agilizar la computación del proceso de entrenamiento en los modelos de mayor carga computacional. La *g4dn.xlarge* se creó con AMI (Imágenes de aplicaciones y sistemas operativos -*Amazon Machine Image*-) mediante Microsoft Windows® Server 2022 y la segunda, *g5.2xlarge*, con AMI de Amazon Linux. Una AMI es una plantilla que contiene la configuración de software (sistema operativo, servidor de aplicaciones y aplicaciones) necesaria para lanzar la instancia. La Tabla 10 muestra un resumen de las características de ambas instancias de Amazon.

Características de las instancias de Amazon		
	g5.2xlarge	**g4dn.xlarge**
Número de CPU's virtuales	8	4
Memoria (GiB)	32.0	16
Memoria por cada CPU(GiB)	4.0	4
Procesador	AMD EPYC 7R32	Intel Xeon Family
Velocidad reloj (GHz)	2.8	2.5
Arquitectura de CPU	x86_64	x86_64
Número de GPU's	1	1
Arquitectura de GPU	nvidia a10g	Nvidia t4 tensor core
Memoria de vídeo de GPU (GiB)	24	16
Capacidad de computación de GPU según lo establecido [47]	7.5	7.5

Tabla 10. Resumen de las características principales de computación de las dos instancias de Amazon AWS usadas en el presente proyecto

Respecto a la integración con Matlab, existen dos opciones para usar la potencia de cálculo de las máquinas virtuales. Primeramente, estaría el cálculo en paralelo con un clúster en AWS. El problema es que se necesita para ello la licencia de Matlab "Parallel Computing Toolbox" y otra de "Matlab parallel server". Al no disponer en la licencia de estudiante usada en Matlab para este proyecto, se descartó dicho camino. La segunda vía, es el uso de otra herramienta de AWS, el llamado CloudFormation, que permite la creación de máquinas virtuales donde ya se incluye la instalación completa de Matlab con sus correspondientes *toolboxes*. En realidad, se crean varios recursos a partir de un fichero JSON que define las particularidades de esa máquina virtual. Este es el camino tomado en el planteamiento de este proyecto, donde se usó el propio desarrollo realizado por Matlab e incluido en GitHub [48]. La Figura 30 muestra una conexión esquemática de este procedimiento.

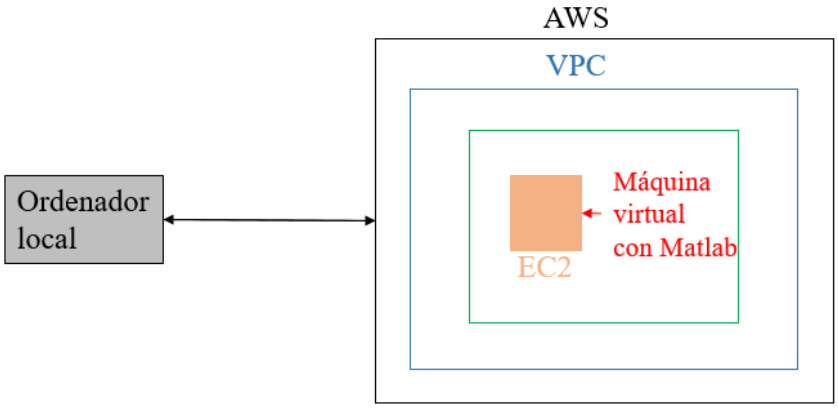

Figura 30. Esquema arquitectura de conexión de Matlab con AWS a través de la creación de máquinas virtuales a través de Amazon CloudFormation

Al hacer uso de esta creación a través AWS CloudFormation, se tiene una máquina virtual con Matlab, Simulink, *toolboxes* y programas de soporte de GPU's en el caso que fuesen instancias EC2 con esta opción. Además, incluye algunos *Add-Ons* pequeños de modelos CNN. No obstante, en cualquier caso, el usuario puede descargar nuevos Add-Ons si no los hubiese, guardar la imagen de esa máquina virtual e incluso trasladarla a otra instancia EC2 con otras capacidades de computación.

Cabe mencionar, que en esta creación con AWS Cloudformation se pueden crean varios recursos (no solo la instancia EC2):

- Grupo de seguridad para conectarse con la instancia, por medio de SSH y RDP.
- La comentada instancia EC2.
- Una IP elástica.
- Una vigilancia de las instancias EC2 (cloudWatch) para cuando quedan inactivas por mucho tiempo o tienen una capacidad computacional muy alta.
- Un AWS Systems Manager document (*SSM document*) preconfigurado para la anotación de la gestión del sistema de la instancia.
- Roles con permisos distintos para acceder a la instancia.

Cabe destacar que se puede seleccionar la AMI que se quiera usar (Windows o Linux) y la versión de Matlab. En el caso del presente trabajo, tal y como se indicó previamente, una instancia se realiza con Windows y la otra con Linux, pero siempre con la versión 2022b de Matlab, para que coincidiera con la instalada en el ordenador local donde estaba instalada esta versión.

Por último, mencionar la visualización del almacenamiento de datos. Tal y como se vio en el apartado anterior 4.2. el conjunto de imágenes de COCO *dataset* es extremadamente pesado. Por este motivo, se intentó utilizar otro servicio de Amazon, S3, el cual funciona como una nube de almacenamiento con paquetes o *buckets*. De esta forma, se puede tener llamadas a los datos tanto desde un ordenador local como desde las instancias de estos *buckets*. No obstante, finalmente se desechó su uso tras varias pruebas de funcionamiento. El principal motivo fue que las instancias al final se pueden lanzar en diferentes localizaciones donde están los servidores. En el caso de este proyecto se lanzaron en EE. UU Oeste (Oregón), identificada la zona por us-west-2 porque ofrecía

mayor número de opciones de instancias a precios menores (los precios varían según las localizaciones). Esto, junto con datos en la nube ralentizaba en exceso el acceso a las imágenes y la computación en partes del código. Por ese motivo, se decidió introducir directamente los archivos de MS COCO en las máquinas virtuales, siendo descargados de carpetas compartidas en la nube.

4.4. Ordenadores utilizados

Tal y como se comentó previamente para el presente proyecto se usaron dos instancias creadas en AWS, (*g4dn.xlarge* y *g5.2xlarge*). Pero también se utilizó un ordenador personal que apoyase el entrenamiento de los detectores a probar. Se usó sobre todo en aquellos que mayor tiempo requerían por un tema de control de gastos. Los datos principales sobre las características del computador son los que se muestran en la Tabla 11.

Características del Computador	
Nombre del dispositivo	DESKTOP-QCO753J
Procesador	Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz 2.81 GHz
RAM instalada	8,00 GB (7,88 GB usable)
Tipo de sistema	Sistema operativo de 64 bits, procesador basado en x64

Tabla 11. Resumen de características principales del ordenador local usado en el presente proyecto

La entrada táctil o manuscrita no está disponible para la pantalla disponible en el equipo. Como se puede deducir, incorpora también una tarjeta gráfica Nvidia, GeForce GTX 1050 Ti, base con disponibilidad de GPU. La única salvedad es que ésta dispone sólo de 4GB de memoria frente y el propio Matlab con la función “gpuDeviceTable” indica que tiene una capacidad de computación de 6.1 con 4GB de memoria en la GPU. Esto supone relativamente poca capacidad frente a *g4dn.xlarge* con 16GB y *g5.2xlarge* con 24GB.

5. Resultados

Como se detalló en el inicio de esta memoria, el objetivo de este trabajo es el estudio de diferentes técnicas de detección de objetos mediante técnicas de Aprendizaje Profundo. Con la base de las herramientas comentadas en el apartado 3.3. , usando la combinación MS COCO *dataset* y Matlab y, bien utilizando la plataforma Amazon AWS o bien en local, se han analizado los distintos detectores para una única clase por la limitación de memoria y largo tiempo empleado para el entrenamiento de cada modelo. Como se quiso priorizar el estudio de un número determinado de detectores y sus posibilidades de variación, frente a su eficiencia en un número mayor de clases, se concretó en una de las clases incluidas en MS COCO asegurando que hubiese suficientes imágenes. Para ello, se determinó como mínimo unas 1500 imágenes para que fuese lo suficientemente representativo a la hora de ajustar la red y para poder comparar resultados sin caer fácilmente en *overfitting*. Usando el explorador de la página web de COCO, comentado en el apartado 4.2. , se buscaron diferentes tipos de objetos que encajasen en esta premisa y finalmente se seleccionó la clase "*stop sign*", señales de *stop,* que en el explorador marca 1803 imágenes, aunque luego en la descarga y extracción sólo aparecieron 1734 imágenes. Conviene señalar que, inicialmente, no se buscó una clase con un número de objetos determinado y se eligió la clase "*car*", coches. Pero COCO posee 12786 imágenes donde hay objetos de tipo coche, por lo que entrenamientos, hasta con los modelos de detección de objetos más veloces y que precisan menos recursos, como SSD o YOLOV3, precisaban un elevado costo computacional y, como resultado final, una duración excesiva para el tiempo total y alcance del presente proyecto.

5.1. Premisas seguidas para los resultados

Para que los resultados obtenidos entre entrenamientos a lo largo del proceso tuviesen sentido desde el punto de vista comparativo, se establecieron durante los entrenamientos una serie de criterios que se indican a continuación:

a) Para realizar el entrenamiento, se dividió el conjunto de imágenes "*stop sign*" en tres grupos, *training*/entrenamiento, validación y test, con un 60%, 10% y 30% del total. Se hizo lo mismo para todos los detectores, aunque hubiese algún tipo de detector en Matlab, que no aceptaba un conjunto de imágenes *y bounding boxes* para realizar validación, como es el caso de RCNN y YOLO V3.

b) Se realizó el test en 5 tipos diferentes de detectores: RCNN, Faster-RCNN, SSD, YOLO V3 y YOLO V4. Para el training se usaron las funciones propias disponibles de Matlab, con la salvedad de YOLO V3, que no tiene función de training como tal y se usó la parte de código de entrenamiento de YOLO V3 aconsejado por Matlab.

c) En Matlab, como paso previo para ejecutar el *training*, es necesario definir las opciones de entrenamiento y crear el detector, el cual se guarda como una variable más mediante un tipo especial de variable de Matlab. Por ejemplo, "yolov3ObjectDetection" para este detector en concreto. Indistintamente del tipo de detector, Matlab permite utilizar cualquier tipo de modelo CNN base para crear el detector, por eso, con el fin de establecer una base para todos, se seleccionó y usó como base de los detectores el modelo CNN denominado ResNet-50, descrita previamente en la sección 2.3.3.

d) Todos los modelos son entrenados usando la GPU bien con las AMI de AWS o del ordenador local.
e) Para el entrenamiento se establecieron como base común los parámetros y características que aparecen en la Tabla 12.

Opciones de entrenamiento	
Tamaño del paquete (Minibatch size)	32
Tasa de aprendizaje inicial (Initial learning rat)	0,001
Reducción de tasa de aprendizaje (Learn rate schedule)	Si (identificado con "Piecewise")
Tasa de reducción de aprendizaje (Learn rate drop factor)	0.1
Límite de épocas para introducir el factor de reducción de aprendizaje (Lear rate drop period)	60
Número máximo de épocas (Max epochs)	100
Validación de datos (Validation Data)	Si (salvo en YOLO V3 y RCNN)
Entorno de entrenamiento (Execution Enviroment)	GPU

Tabla 12. Características base comunes para todos los detectores realizados en el presente proyecto

f) Faster-RCNN y RCNN, se ejecutan con los mismos parámetros, pero con muchas menos iteraciones debido al enorme coste computacional. Por ejemplo, en Faster-RCNN hubo tiempos de cerca de 12h de entrenamiento para sólo 4 épocas (cada época se corresponde con una pasada completa por el conjunto de datos de entrenamiento). En cualquier caso, son detectores que ocupan mucha memoria, por lo que es necesario disminuir el número de imágenes por *minibatch* (lote de imágenes para actualizar los pesos), con lo que se incrementaron bastante las iteraciones bajando las épocas. Por ejemplo, en el caso de RCNN se pudo llegar a tener un *minibatch* de 4 imágenes con 7 épocas totales mientras que Faster-RCNN sólo permite un *minibatch* y las mismas 7 épocas. Aclarar, que estos datos son fruto de distintas pruebas realizadas a partir de las características base de entrenamiento explicadas en el anterior punto e) previo y con estas particularidades, con el fin de determinar cuándo el entrenamiento produce o no mejora en el ajuste.

g) En YOLOV3 no existe una función propia en Matlab y los detectores son creados mediante código de programa sugerido por Matlab, tal y como se ha comentado previamente. Se usa la función "dlfeval" que permite entrenar detectores personalizados según funciones de gradiente, sigmoide, y otras configuraciones. En este trabajo se usa la técnica del gradiente, si bien aumentando ligeramente el *learning rate*, o tasa de aprendizaje, desde un valor bajo hasta un determinado límite, en lugar de comenzar con el indicado en la tabla 12. Con tal propósito se consideran dos parámetros para modificar la mencionada tasa: *warmupPeriod* y *PenaltyThreshold*.
El primero, *warmupPeriod*, incrementa la tasa de forma exponencial según la siguiente expresión

$$tasa\ de\ aprendizaje = \left(\frac{número\ de\ iteración}{warmupPeriod}\right)^4 \quad \textbf{(6)}$$

El segundo, *PenaltyThreshold*, es un umbral que establece que un valor IoU por debajo de él según el *ground truth bounding box*, se penaliza.

h) Todos los detectares analizados, excepto RCNN, utilizan de una forma u otra *anchor boxes* predefinidos, tal y como se vio en la sección 3.3. Para las pruebas de los detectores se usó la función de Matlab "estimateAnchorBoxes", la cual, en base a un conjunto de imágenes crea el número de *anchor boxes* que se deseen mediante el algoritmo de clasificación *K-means*. No obstante, esta función también proporciona un valor de IoU (sección 3.2.3.) de esos *anchor boxes* con los objetos reales (es decir *ground truth bounding boxes* de los objetos). Así, iterando para un número determinado de anchor boxes se vio cómo para un número reducido de *anchor boxes*, los valores IoU eran bajos y al ir aumentando el número, el IoU también lo hacía, pero a partir de un cierto número apenas aumentaba o incluso disminuía ligeramente. Llegaba por tanto a una especie de asíntota como se puede ver en la Figura 31.

i) El número de *anchor boxes* para cada capa de la CNN base usada en la detección tiene una relación entre sí. Es decir, se asignará a cada capa de extracción un número de *anchor boxes* para ser usados en esa capa. En los distintos detectores usados se seleccionaron 2 y 3 capas, salvo algún caso de estudio concreto como el refuerzo del modelo CNN base para la prueba en SSD. Por eso, se establecieron grupos o divisiones de los *anchor boxes* totales en 2 o 3 según el caso. Como se pretendió realizar la extracción con el mismo número de anchor boxes para cada capa, dentro de esas 2 o 3 divisiones se eligió el mismo número de *anchor boxes*. Finalmente, con todo ello, se seleccionó 18, como el número de *anchor boxes* finalmente usados, ya que permitía más de un 0,65 de IoU con la clase "car", aunque finalmente no se desarrollaron en esta clase por lo expuesto previamente, y más de un 0,7 con la clase "*stop sign*" (Figura 31 y Figura 32).

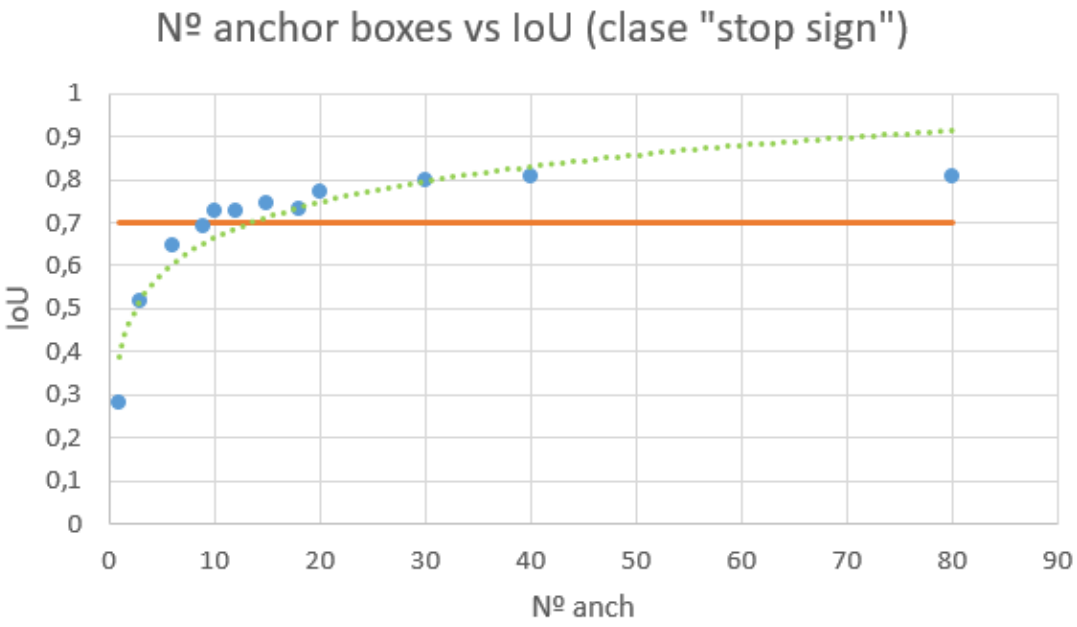

Figura 31. Gráfica comparativa del número de anchor boxes usados contra el IoU para la clase "stop sign"

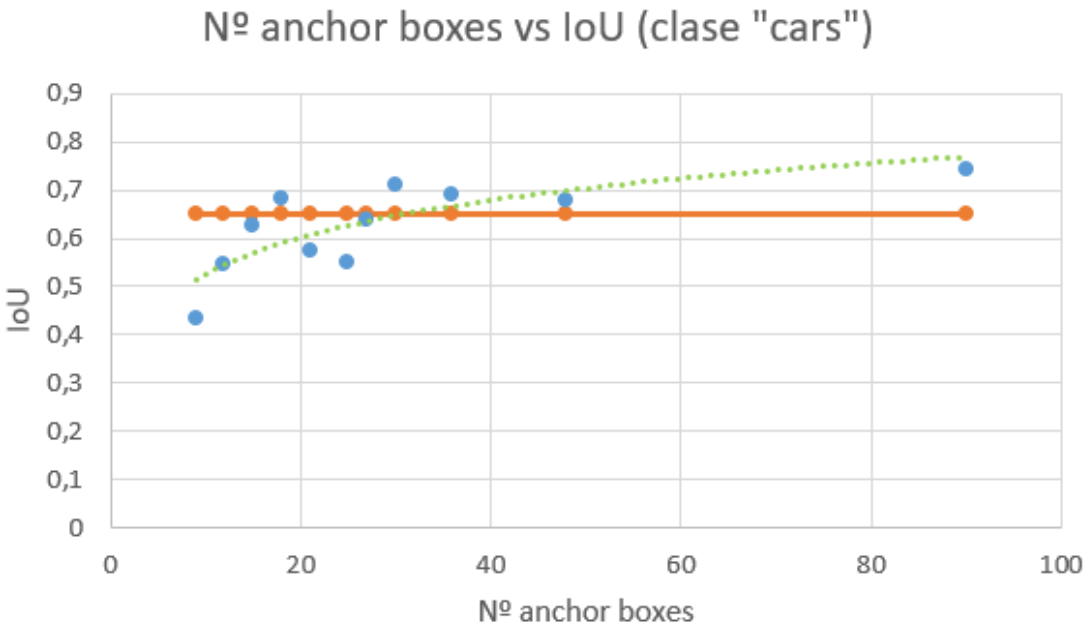

Figura 32. Gráfica comparativa del número de anchor boxes usados contra el IoU para la clase "car"

Hay que recordar que, por lo general, más número de *anchor boxes* que mejoren el IoU, conlleva una mejora de la precisión promedio del detector, aunque con un coste computacional mayor que se traduce en un aumento del tiempo y una necesidad de memoria también mayor para procesar el entrenamiento del detector.

Conviene destacar que, tal y como se comentó en el punto h) anterior, la obtención de estos detectores se llevó a cabo de forma automática usando la función de Matlab "estimateAnchorBoxes", la *K*-medias para su selección respecto de los *ground truth bounding boxes*.
Por último, es importante precisar que lo explicado aquí no aplica a los detectores del tipo RCNN y Faster-RCNN donde no existe ese tipo de extracción tal y como se vio en el apartado 3.3.1 y 3.3.3. Aunque, tal y como se indica en el apartado 3.3.3, los detectores Faster-RCNN sí que usan *anchor boxes*, pero no en la detección en sí, sino para mejora del propio algoritmo. No obstante, se usaron igualmente 18 *anchor boxes* en ellos.

j) Se realizó en todos los casos un proceso de *data augmentation*, que consiste en generar nuevas imágenes a partir del subconjunto de entrenamiento para mejorar la generalización final del detector. Esto se realizó usando una inversión simétrica de la imagen en un eje vertical centrado en ella. Se realizó además una limitación de intersección de la unión de los *ground truth bounding boxes* originales sobre la imagen invertida dividida por el área de los *bounding boxes* generados tras la inversión con la imagen invertida, ya que suele haber pequeñas diferencias. Esta limitación se estableció en 0,25 y se vio que, con ella, en algunos casos se reducía ligeramente el *ground truth bounding box* invertido respecto del original, pero permitía disminuir el número de *bounding boxes* que sobrepasaban los límites de la imagen debido a un ancho y alto excesivos. La razón de esto es que uno de los principales problemas de MS COCO en la integración con Matlab, es que se tiene una gran variedad de imágenes y en muchas de ellas los objetos se ubican en los límites de la imagen y, además, son de tamaño muy pequeño. En caso de que haya *bounding boxes* con valores negativos, cero, de ancho o alto 0 o con 1 o 2 canales en lugar de 3, la compilación da error por las funciones propias de Matlab, cosa que es obvia e intuitiva. De esta manera, en el proceso de inversión del *data augmentation* se generaban imágenes con *ground truth bounding boxes* con coordenadas *x, y* fuera de los límites de la imagen o con su ancho y alto. La comentada limitación por umbral en la intersección soluciona el problema del ancho y alto extendido para ajustarlo a la imagen. El primer problema de coordenadas *x, y* fuera de la imagen (con valores de coordenadas de valor cero o negativos) se trata también por código en la función que realiza el proceso de *data augmentation*, tanto antes como después de la inversión de la imagen. Realizarlo después es indispensable, obviamente, por la restricción de las funciones de entrenamiento y porque lógicamente puede presentar un problema en la extracción de regiones en los detectores.
El realizarlo antes de la inversión también se debe a que se vio que además de redondear los valores, la salida generaba ya unos mejores resultados de los *ground truth bounding boxes*. Por el mismo motivo, no podía hacerse sólo antes, ya que había algunos casos, por el desplazamiento ligero de los *ground truth bounding boxes invertidos*, que generaban valores negativos o cero a

pesar de haber confinado los *bounding boxes* dentro de las dimensiones de las imágenes.
También se realiza un ajuste respecto al ancho y alto mínimo del *ground truth bounding box* a 1 ya que a veces se generaban rectángulos de ancho o alto cero al aplicar el umbral para que no se saliesen de la imagen (casos de *ground truth bounding boxes* muy pequeños y pegados a los bordes).
Únicamente, con RCNN no aparece el problema de la no compilación (Faster-RCCN tiene problemas con los *bounding boxes*, pero no con las imágenes de menos de 3 canales). No obstante, se aplicó también la misma función de *data augmentation*,con la salvedad que se hizo con *Datastores* convertidos luego a tabla como se verá más tarde en el punto 5.7.

k) Se realiza en todos los detectores un escalado de las imágenes del conjunto de entrenamiento para reducir la dimensión y el coste computacional. En cada caso, se ajusta al tamaño de la red base CNN usada en los detectores que es 224×224 en ResNet (también en el caso especial estudiado de Darknet para YOLO V4) y 227×227 en Squeezenet. Además, se vuelve a aplicar, como en el caso del *data augmentation* que los valores de los *ground truth bounding boxes* sean enteros, nunca 0 o negativo en *x* e *y* (dentro de la imagen) y con alto y anchos de valor mínimo 1 (también nunca 0 o negativos en los ancho y altos en caso de error en los valores de los *bounding boxes* en el conjunto de partida). Esto se aplica tanto antes del escalado, por si no se usaba el *data augmentation* (aunque siempre se usó en los ejemplos usados en la memoria) como después del escalado, ya que generaba siempre decimales en los bounding boxes, además de valores de ancho y alto menores que 1 en algunos objetos muy pequeños que tenían etiquetados algunas imágenes.
Este preprocesamiento se aplicó a todos los tipos de detectores probados y analizados, salvo a YOLOV3 que usa la función propia de Matlab "*preprocess*", específica de YOLOV3 y que realiza una normalización (otros detectores la normalización es parte de la función de entrenamiento en sí) y un reescalado. Tras realizar esto, se necesita aplicar una función que arregle los comentados problemas de los *bounding boxes* ya que la función *preprocess* no lo realiza. Esta diferencia respecto al resto de detectores se debe a que, como se ha comentado previamente, el entrenamiento de YOLOV3 no es con una función propia de Matlab y el entorno de *training* en Aprendizaje Profundo que este posee, sino que es mediante código generado para ese propósito.

l) En SSD, YOLOV3 y YOLOV4 se aplicó una función que cambiaba las imágenes en blanco y negro a 3 canales, dado que otro del inconveniente de MS COCO es que tiene algunas imágenes en grises y estos 3 detectores presentan problemas con ellas (su compilación da error) debido al uso del *anchor boxes* de extracción de regiones. Este proceso se realizó triplicando el canal existente. Por tanto, se puede considerar que no afecta a los resultados de detección y, por consiguiente, no se aplicó en los detectores de RCNN y Faster-RCNN.

m) A las imágenes del set de *test* se les aplicó el preprocesamiento de imágenes para tenerlas en igual tamaño que las usadas en el *training*.

n) A las imágenes de validación en los casos que se podían usar (RCCN y YOLOV3 no admiten conjunto de validación), se aplicó *data augmentation* y preprocesamiento de imágenes para ser lo más parecidas a las de training.

o) No se realizó parada por empeoramiento consecutivo del error respecto a las imágenes del conjunto de validación, aunque Matlab permita esta función. Si bien se probó inicialmente, y se comprobó que realizando comprobación de imagen de validación cada 50 iteraciones como se pretendía, a lo largo del entrenamiento la convergencia de la función de pérdida y precisión fluctuaba bastante y producía siempre paradas tempranas que hacían que el aprendizaje no fuese lo suficientemente eficiente. Por tanto, todos los entrenamientos se pararon al alcanzar el número de épocas establecidas en las opciones de *training*.

p) Se usaron las variables de *datastores* para guardar direcciones de imágenes y *bounding boxes* e ir realizando la secuencia de aplicación de funciones sobre estas. Únicamente, RCNN no permite en Matlab el uso de *datastores*. No obstante, como se explicó en la sección 4.2, se realizó una conversión de tabla a *datastores* para aplicar las funciones de la misma manera en todos los casos y, al final, se realizó una reconversión a tabla por medio de una función específica, pero que no afecta a resultados al ser simplemente conversión entre tipos de variables u objetos usados por Matlab.

q) El entrenamiento de Faster-RCNN y RCNN se realiza con propuesta de 1200 imágenes sobre el conjunto, en lugar de las 2000 por defecto de Matlab. Esto mejora resultados de velocidad, pero perjudica la precisión. No obstante, tras diversas pruebas se convirtió en el valor de referencia al ser el más equilibrado, ver punto f) anterior.

5.2. Resultados con ResNet 50 como base

La comparación de resultados se lleva a cabo entre los cinco tipos de detectores, RCNN, Faster-RCNN, SSD, YOLOV3 y YOLOV4, con un tipo de red convolucional base. Esta es una de las ventajas que posee Matlab, dado que permite el uso de cualquiera de sus redes CNN preentrenadas disponibles para la generación de estos detectores. Así pues, se seleccionó ResNet 50 (sección 2.3.3) como base en los cinco detectores. Para la extracción de mapas de características, en SSD, YOLOV3 y YOLOV4, se usaron las capas “activation_22_relu" y "activation_40_relu”.

La selección de capas de extracción debe hacerse mediante prueba de ensayo y error al depender mucho del tamaño de las imágenes y objetos. Para Faster-RCNN que usa una

detección para la CNN generadora de ROI's, Matlab sólo permite una capa y tal caso se usó "activation_40_relu".

Como se puede ver en la Figura 33, se obtuvieron los resultados mostrados de *recall*-precisión con los entrenamientos realizados teniendo en cuenta las pautas comunes establecidas en el punto anterior. Cabe destacar que en YOLOV4 la tasa de aprendizaje tuvo que reducirse a 0,0001 ya que durante el entrenamiento presentaba continuamente de problemas al producir desbordamiento de los parámetros del detector.

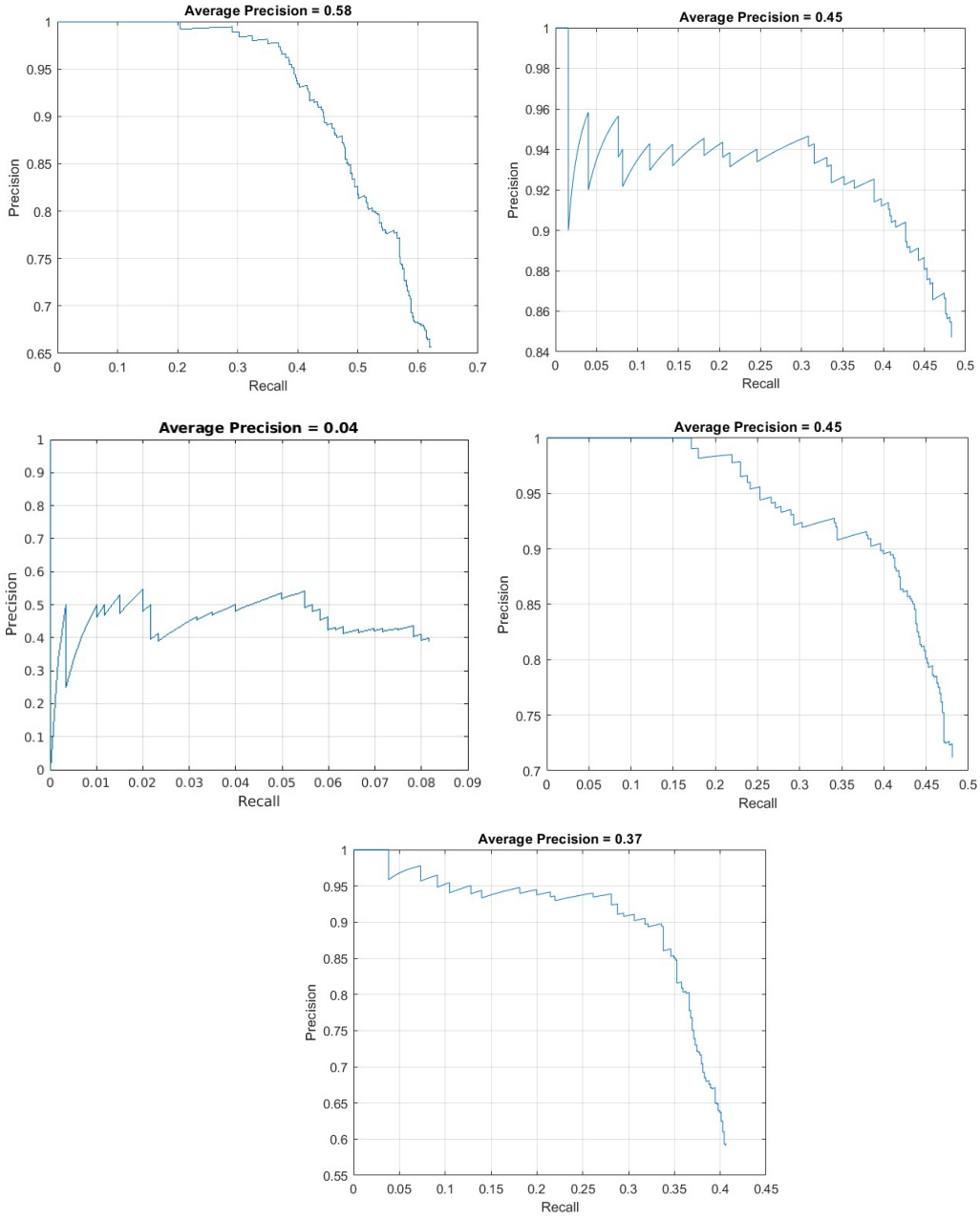

Figura 33. Curvas de precisión-recall para los cinco tipos de detectores principales y las características base comunes indicadas en la Tabla 12 (arriba izquierda, SSD, arriba derecha, YOLOV3, medio izquierda, YOLOV4, medio derecha, Faster-RCNN, abajo, RCNN)

A partir de los resultados mostrados en la Figura 33 se puede inferir, primeramente, cómo YOLOV4 presenta problemas con esta integración. Tiene una precisión alrededor del 50% pero un *recall* extremadamente bajo. En cuanto a YOLOV3 y Faster-RCNN se ve que generan una precisión promedio igual, pero con una precisión mayor. Esto, como se explicó en la sección 3.3. , significa que el detector Faster-RCNN es capaz de obtener más resultados verdaderos que YOLOV3. Se puede ver también que el que mejor precisión promedio tiene es SSD y alcanza también un valor de *recall* mayor, es decir, realiza más detecciones, aunque es verdad que la precisión baja algo en esas detecciones extras. La Tabla 13 muestra un resumen de resultados *mAP* comparativos entre los detectores con ResNet-50.

	SSD	**YOLOV3**	**YOLOV4**	**Faster-RCNN**	**RCNN**
mAP con ResNet 50, condiciones base	0,58	0,45	0,17	0,45	0,37

Tabla 13. Comparativa de la precisión promedio para los cinco tipos de detectores principales y las características base comunes indicadas en la Tabla 12

Con respecto a los tiempos de entrenamiento, en la Tabla 14 se muestran los resultados obtenidos, que, si bien se refieren a distintas configuraciones, proporciona una referencia de la rapidez de unos detectores frente a otros.

	SSD	**YOLOV3**	**YOLOV4**	**Faster-RCNN**	**RCNN**
Tiempos de entrenamiento con ResNet 50, condiciones base (horas)	2.58	9.81	2.45	31.2	9.8
Sitio computación	AWS g4	Ordenador	AWS g5	Ordenador	Ordenador

Tabla 14. Tiempos de entrenamiento para los cinco tipos de detectores principales y las características base comunes indicadas en la Tabla 12

Así pues, en vista de los mejores resultados de mAP y tiempos de entrenamiento obtenidos, SSD se convirtió en el detector preferente y sobre el que más variaciones básicas se hicieron, una vez seleccionado, como se verá en el siguiente apartado.

Por último, la tabla Tabla 15 muestra los tiempos de detección sobre las imágenes del conjunto de prueba, siendo de nuevo el más rápido SSD seguido de YOLOV4. Cabe destacar la mayor velocidad de Faster-RCNN frente a RCNN pese a la lentitud del entrenamiento.

	SSD	**YOLOV3**	**YOLOV4**	**Faster-RCNN**	**RCNN**
Tiempos de detección set de test con ResNet 50, condiciones base (horas)	26.67	101.97	44.37	1152.66	2522.66
Sitio computación	AWS g4	Ordenador	AWS g5	Ordenador	Ordenador

Tabla 15. Tiempos de detección sobre el set de test para los cinto tipos de detectores principales y las características base comunes indicadas en la Tabla 12

5.3. Detectores SSD

Una vez seleccionado SDD, sobre él se establecieron distintas variaciones sobre las premisas básicas explicadas en el apartado 5.1. y ResNet 50 como se vio en el apartado anterior 5.2. ,ya centrándonos en este detector. Estas variaciones fueron:

- Prueba con una red base CNN de mayor profundidad: ResNet101. Se usan las capas "res5c_relu", "res4b22_relu", "res3b3_relu" para la extracción de características.
- Prueba con una red base CNN de menor profundidad: ResNet18. Se usan las capas “res2b_relu", "res4b_relu” para la extracción de características.
- Prueba con una red base diferente y compacta: Squeezenet. Se usan las capas “fire9-concat”, “fire5-concat” para la extracción.
- Prueba con tasa de aprendizaje menor: 0.0001
- Prueba con entrenamiento usando el optimizador Adam en lugar de SGDM. SGDM se refiere al de gradiente descendente estocástico con momento y Adam es similar, pero con tasas de decaimiento de la media móvil de gradiente y de gradiente cuadrado. Se estableció en todos los modelos el gradiente de decaimiento (GradientDecayFactor) como 0.9 y el gradiente cuadrado (SquaredGradientDecayFactor) de 0.99

Con estas variaciones, se obtuvieron los resultados de precisión promedio que se muestran en la Tabla 16.

SSD	mAP
Modelo base (con ResNet50, learningrate 0.001 y sgdm)	0.58
ResNet18	0.46
ResNet101	0.57
Squeezenet	0.00
Learning rate menor a 0.0001	0.51
Adam	0.57

Tabla 16. Comparativa de la precisión promedio para las variantes comunes de detectores creadas en el tipo SSD

Como se puede ver una red de mayor profundidad, como es ResNet101, no aporta mejores resultados y una de menos los reduce, como ocurre con ResNet18. Mientras, con Squeezenet no se consigue un detector correcto. Además, un *learning rate* menor disminuye ligeramente los resultados y con el optimizador Adam apenas hay diferencia. La figura 33 muestra la gráfica precisión-*recall* para SSD con ResNet18 a modo de ejemplo.

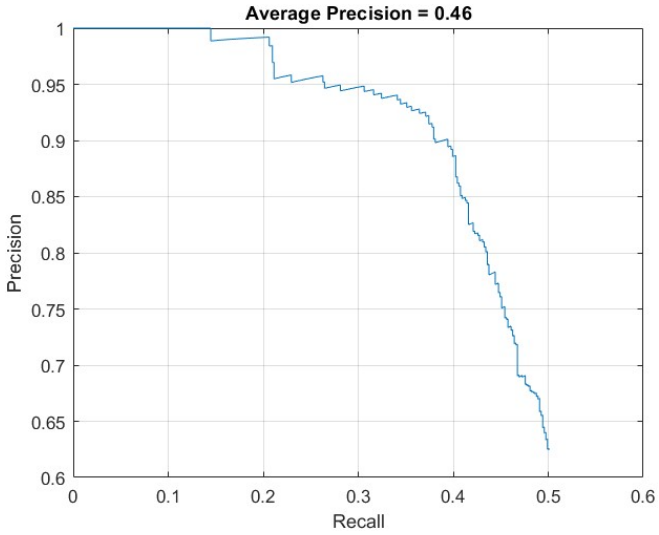

Figura 34. Curva precisión-recall del detector SSD realizado con base ResNet18

Además de las variaciones mencionadas respecto a los parámetros y configuraciones de SSD, se aplicaron otras variaciones particulares en cada tipo de detector. Si bien, al haber sido SSD el más prometedor, las pruebas se focalizaron en él, realizando un mayor número que en otros tipos, tal y como se resume a continuación:

- Entrenamiento con ResNet50, pero añadiendo más capas convolucionales a esta para utilizar mayor número de detectores como propone Matlab en el ejemplo [49]. Así, se incorporan siete capas convolucionales tras la "activation_40_relu", que es una de las dos que se usaba en el modelo base de ResNet50 para la extracción. De esta forma, se usan para la extracción "activation_22_relu", "activation_40_relu", "relu6_2", "relu7_2", "relu8_2".
- Se creó un detector con 180 épocas en lugar de sólo 100 como en el detector base.
- Se creó un detector donde se realizaba un preprocesamiento para ajustar su tamaño a la resolución de imagen de 500 ×500×3. Esta es mayor que el input requerido para la red CNN ResNet 50 (224x224x3), por lo que internamente es necesario aplicar escalados para el ajuste.
- Se creó un detector donde se invertían las capas de extracción. Es decir, tal y como se explicó en la sección 3.2.5. *los anchor boxes* se dividieron en 2 o 3 grupos (2 en este caso) y estos tenían una relación con las capas de extracción. Se supone, por tanto, que los *anchor boxes* de menor área encerrada han de usarse en las capas primeras y los que tengan mayor área encerrada en las últimas capas. Esto es para aprovechar la escalabilidad de los detectores (ver sección 3.3.4.) y, el hecho de que al avanzar en la convolución la resolución de los mapas de características son cada vez más pequeños por lo que *anchor boxes* muy pequeños pueden tener complicado la intersección IoU correcta con el *ground truth bounding box* del objeto. Se probó a invertir activation_40_relu por activation_22_relu (de normal en el resto activation_40_relu se usó para *los anchor boxes* de mayor área encerrada y activation_22_relu con los de mayor área entregada)
- Se crearon varios detectores modificando el parámetro L2, factor para la regularización en la función de pérdida. Este parámetro, permite al final introducir

rechazos en la red para que el detector no caiga en mínimos locales fácilmente y pueda ir al mínimo global, por así decirlo. Con esto se previene el *overfitting*. Esto se aplicó como prueba al ver que las validaciones se quedaban lejos del ajuste deseado con el *dataset* del *training*. El valor por defecto en Matlab es 0.0001, si bien se probaron los siguientes valores 0.001, 0.1 y 1 respectivamente.

En base a estas variaciones se obtienen los resultados que se muestran en la Tabla 17.

SSD	mAP
Entrenamiento con ResNet 50 e insertando capas intermedias , "relu6_2", "relu7_2", "relu8_2"	0.53
Con 180 epochs en lugar de 100 epochs	0.58
Con aumento del tamaño de imagen de la red a 500×500×3	0.53
Entrenamiento con ResNet 50, variando el orden o asignación de los *anchor boxes* predefinidos	0.52
Entrenamiento con ResNet 50 y L2 de valor 0.001	0.58
Entrenamiento con ResNet 50 y L2 de valor 0.01	0.57
Entrenamiento con ResNet 50 y L2 de valor 1	0.41

Tabla 17. Comparativa de la precisión promedio para las variantes específicas realizadas para el tipo SSD

Se puede observar claramente cómo los valores de regulación, L2 apenas tiene impacto en el modelo, aunque sí que se consigue acercar la curva de validación a la de training. Por otro lado, se ve cómo el número de épocas ya no tiene impacto. Es decir, se sobreentrenaría el detector sin motivo. Igualmente, aumentar el tamaño de la imagen empeora los resultados, aunque el tiempo de entrenamiento apenas se resiente.

5.4. Detectores YOLOV3

Para los detectores YOLOV3 se vuelven a realizar variaciones similares al caso anterior, con el fin de tener una base de comparación referencial entre detectores. La única salvedad es que en este no se hizo el estudio con el optimizador Adam, ya que, como se ha mencionado previamente, con YOLOV3 se hace el entrenamiento por bloques de código generado y no en el entorno de entrenamiento de Aprendizaje profundo de Matlab. En cualquier caso, la Tabla 18 recoge los resultados para este caso.

YOLOV3	mAP
Modelo base (con ResNet50, learningrate 0.001 y sgdm)	0.45
ResNet18	0.42
ResNet101	0.1
Squeezenet	0.49
Learning rate menor a 0.0001	0.09
~~Adam~~	-

Tabla 18. Comparativa de la precisión promedio para las variantes comunes de detectores creadas en el tipo YOLOV3

Se puede ver cómo existe un problema con la red de mayor densidad de ResNet101 para generar el detector (Figura 35). La precisión promedio se reduce drásticamente al menos con estas condiciones de entrenamiento.

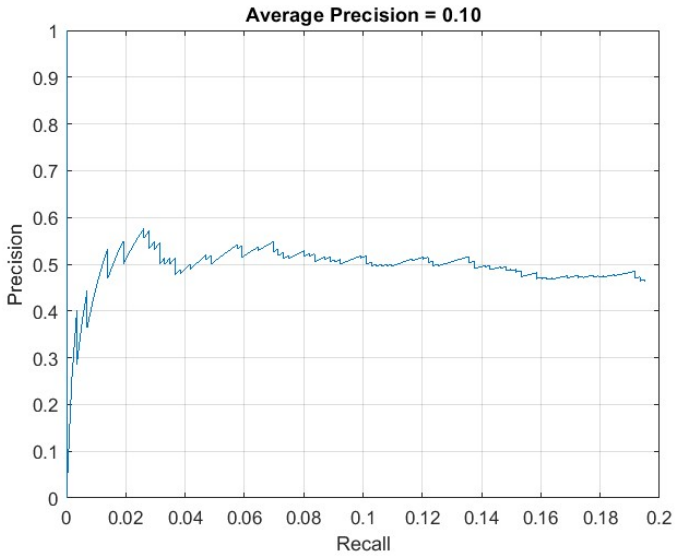

Figura 35. Curva precisión-recall del detector YOLOV3 realizado con base ResNet101

Por otro lado, se crearon también otros detectores para comprobar particularidades de YOLOV3, cambiando:

- El warmupPeriod (ver apartado 5.1.) a 1000
- El warmupPeriod a 1000 y se redujo la tasa de aprendizaje a 0.0001

Con estas dos variaciones se obtuvo error en el primer caso y, en el segundo, una disminución elevada de la precisión promedio, como se aprecia en la Tabla 19.

YOLOV3	mAP
Entrenamiento con ResNet 50 y warmupPeriod igual a 1000	Error
Entrenamiento con ResNet 50 y warmupPeriod igual a 1000 y tasa de aprendizaje de 0.0001	0.27

Tabla 19. Comparativa de la precisión promedio para las variantes específicas realizadas para el tipo YOLOV3

5.5. Detectores YOLOV4

Como se vio en el apartado 5.2. , el detector base realizado con ResNet50 y YOLOV4 tiene problemas de precisión y capacidad para detectar objetos. Cabe destacar que la tasa de aprendizaje en el modelo básico de YOLOV4 tuvo que realizarse con 0.0001 al contrario que en los otros. Un valor de la tasa de aprendizaje de 0.001 generaba fallos durante el entrenamiento. La Tabla 20 muestra los resultados de YOLOV4 con respecto a mAP. Aquí, se puede apreciar cómo los casos tienen malos resultados de precisión, siendo bastante similares entre sí, a excepción del entrenamiento con Adam en lugar del optimizador sgdm.

YOLOV4	mAP
Modelo base (con ResNet50, learningrate 0.0001 y sgdm)	0.17
ResNet18	0.12
ResNet101	0.30
Squeezenet	0.16
Learning rate menor a 0.0001	NO APLICA
Adam	0.54

Tabla 20. Comparativa de la precisión promedio para las variantes comunes de detectores creadas en el tipo YOLOV4

Al igual que en otros tipos de detectores, se realizaron las siguientes variaciones propias, si bien sólo para YOLOV4.

- Uso de la red base preentrenada "csp-darknet53-coco" que no deja de ser un detector ya entrenado en todas las clases de Microsoft COCO. No obstante, en este trabajo se realizó un entrenamiento sólo para la clase "*stop sign*".
- Inversión de las capas de extracción usadas para generar el layout "activation_22_relu" y "activation_40_relu".

La Tabla 21 muestra los resultados obtenidos para mAP con las variaciones mencionadas. Se ve cómo la inversión de capas perjudica seriamente al detector y cómo la precisión media con "csp-darknet53-coco" mejora los resultados a pesar de estar entrenada con el solucionador SGDM. No obstante, no llega a superar el valor de precisión del detector entrenado con el optimizador Adam.

YOLOV4	mAP
Se usó una red base preentrenada "csp-darknet53-coco"	0.41
Se hizo un entrenamiento con ResNet 50 e invirtiendo las capas de extracción	0.003

Tabla 21. Comparativa de la precisión promedio para las variantes específicas realizadas para el tipo YOLOV4

5.6. Detectores Faster-RCNN

Siguiendo el mismo esquema de variaciones que en los casos anteriores, se cambiaron los modelos CNN base, realizando las mismas modificaciones en todos los casos. Cabe destacar, que el entrenamiento con Faster-RCNN y RCNN no tiene nada que ver con las versiones de YOLO y SSD. El entrenamiento de estos últimos, por ejemplo, en el ordenador portátil, es de algunas horas y en el caso de los primeros hablamos casi de días

En cualquier caso, la Tabla 22, refleja los resultados según el mAP.

Faster-RCNN	mAP
Modelo base (con ResNet50, learning rate 0.001 y sgdm)	0.45
ResNet18	0.48
ResNet101	Sin resultado: excesivo tiempo de *training*
Squeezenet	0.43
Learning rate menor a 0.0001	0.19
Adam	0.12

Tabla 22. Comparativa de la precisión promedio para las variantes comunes de detectores creadas en el tipo Faster-RCNN

Conviene destacar en este caso, que una red menos densa como ResNet18 obtiene resultados ligeramente mejores que con la ResNet50 básica, mientras que con una más compacta como *squeezenet* empeora ligeramente. Por otro lado, al cambiar la tasa de aprendizaje o usar el optimizador Adam empeoran los resultados al bajar la mAP a más de la mitad. Si bien es cierto, que se podría haber extendido el entrenamiento incrementando el número de épocas, en cuyo caso se podrían haber obtenido mejores resultados. La Figura 36 muestra la curva precisión-*recall* para este clasificador, observándose resultados no suficientemente satisfactorios.

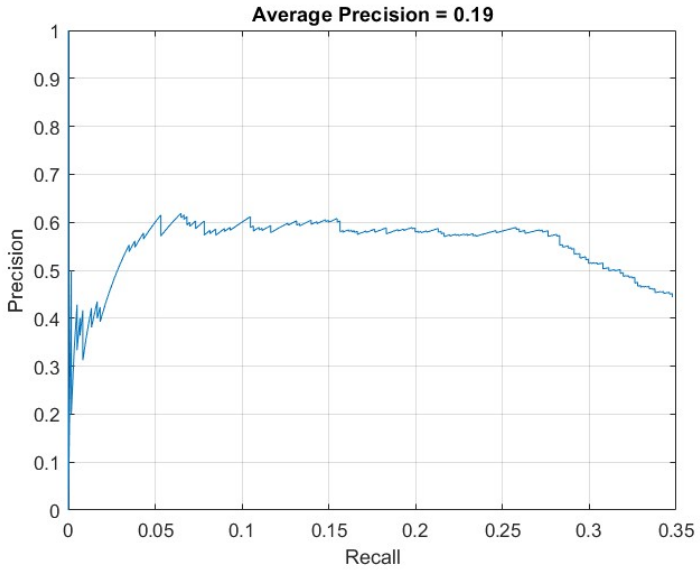

Figura 36. Curva precisión-recall del detector Faster-RCNN realizado con tasa de aprendizaje (learning rate de 0.0001)

Por otro lado, en cuanto a las variaciones para crear detectores propios, se estableció sólo la de la comparativa con respecto a la base de aumentar el número de ROI's generados a 2000 (el valor por defecto por Matlab para el entrenamiento de Faster-RCNN y RCNN). Cabe recordar que para el modelo base y las anteriores modificaciones dentro de este apartado fueron realizadas con un valor de 1200. La Tabla 23 refleja el resultado mAP, relativamente insatisfactorio del modelo.

Faster-RCNN	mAP
Modelo base con 2000 ROI's en lugar de 1200	0.44

Tabla 23. Comparativa de la precisión promedio para las variantes específicas realizadas para el tipo Faster-RCNN

Como se puede ver, no hay apenas variación e incluso es un punto porcentual menor que sin la mencionada modificación.

5.7. Detectores RCNN

Por último, para este tipo de detectores se obtuvieron los resultados que se muestran en la Tabla 24 con las variaciones base indicadas. En esta se puede ver cómo el uso de una red más densa como ResNet101 obtiene mejores resultados que otras menos densas como ResNet50 y ResNet18. Igualmente se observa cómo una red compacta como Squeezenet tiene una precisión similar a estos detectores basados en ResNet menos densos. Por último, destacar la pérdida de precisión a menores valores de *learning rate y usando el optimizador Adam.*

RCNN	mAP
Modelo base (con ResNet50, learningrate 0.001 y sgdm)	0.37
ResNet18	0.36
ResNet101	0.43
Squeezenet	0.37
Learning rate menor a 0.0001	0.26
Adam	0.26

Tabla 24. Comparativa de la precisión promedio para las variantes comunes de detectores creadas en el tipo RCNN

A partir de los resultados de la tabla anterior, se puede ver cómo al disminuir la tasa de aprendizaje se reduce la precisión promedio, ocurriendo algo similar a lo que pasaba con Faster-RCNN.

Por último, en cuanto a las variaciones particulares se realizaron las siguientes recogidas en la Tabla 25:

- Aumento de las épocas a 15 en lugar de 7 sobre el modelo de ResNet18 (se eligió este en lugar del base de ResNet50 para reducir el tiempo de entrenamiento).
- Se aumentó el número de ROI's propuestas a 2000.

RCNN	mAP
Se aumentaron las épocas sobre el modelo de ResNet18 a 15	0.32
Se usó un detector a partir del base pero que genera 2000 ROI's en lugar de 1200	0.39

Tabla 25. Comparativa de la precisión promedio para las variantes específicas realizadas para el tipo Faster-RCNN

Como se puede ver en la tabla, se reduce la precisión al subir las épocas. Claramente se cae en un sobreajuste y pérdida de generalización que, al considerar el conjunto de test, crea esta disminución de precisión.

5.8. Resumen de resultados

Por último, y a modo de conclusión general, la Tabla 26 engloba todos los detectores hasta aquí planteados en relación con las modificaciones base, que son las más comparables entre sí. En esta tabla, se marcan en color gris los detectores con mejor precisión media según su tipo.

Resultados mAP					
	SSD	YOLOV3	YOLOV4	Faster-RCNN	RCNN
ResNet 50	0.58	0.45	0.17	0.45	0.37
ResNet18	0.46	0.42	0.12	0.48	0.36
ResNet101	0.57	0.1	0.30	Imposible por tiempo de training	0.43
Squeezenet	0.00	0.49	0.16	0.43	0.37
Learning rate menor a 0.0001	0.51	0.09	NO APLICA	0.19	0.26
Adam	0.57	NO APLICA	0.54	0.12	0.26

Tabla 26. Tabla resumen de todos los detectores creados con las variaciones comunes a todos (destacado en gris los de mayor precisión promedio)

A grandes rasgos, se deduce el mejor desempeño de los detectores SSD, el apropiado de YOLOV3 y Faster-RCNN y la pérdida de precisión promedio de RCNN. Si bien es cierto que inicialmente estos detectores RCNN se diseñaron sin reducción de imagen obteniendo resultados algo mejores. Uno de los principales problemas de MS COCO es que posee muchos objetos pequeños, por lo que las características propias de extracción de RCNN por métodos no de aprendizaje profundo (ver sección 3.3.1.) tiene mayores problemas para generarlos.

Cabe destacar también cómo entre distintos detectores creados con diferentes redes básicas CNN, los resultados son siempre parecidos entre sí. Lo que ya tiene mayor impacto son las tasas de aprendizaje y el optimizador usado. Es, por tanto, más importante determinar qué modelo CNN base usar.

Por último, en base a los resultados recopilados y resumidos en la Tabla 26, se seleccionaron los detectores que ofrecían mayor valor de precisión promedio y se aplicaron a 6 imágenes distintas tomadas sobre el mismo objeto y creadas de tal forma que supongan distintos retos para los detectores a la hora de detectar la señal de *stop*, para los que se han entrenado. Así se identifican los siguientes, mostrando en las Figuras 36 a 41 cada uno de ellos, respectivamente.

- Imagen de una señal de *stop* centrada y más o menos el objeto mediano, referida como “Grande cent.” (Figura 36)
- Mismo objeto pero con la señal mayor y no en el centro, referido como “Grande lat.” (Figura 37)
- Mismo objeto pero con la imagen borrosa, referido como “Borroso” (Figura 38).
- Mismo objeto pero parcialmente oculto, referido como “Oculto” (Figura 39).

- Mismo objeto pero muy pequeño en relación a la imagen, referido como "Pequeño" (Figura 40).
- Imagen donde la señal de *stop* se sustituye por una de circulación prohibida. Referida como "No es *stop*" (Figura 41).

En la Tabla 27 se resume el comportamiento de los detectores mencionados a la hora de realizar inferencias sobre las imágenes utilizadas como prueba.

		¿Detecta el objeto? (Si/No)	***Confidence score***	**¿Hay falsos positivos?**	**Número falso positivos**
SSD	Grande cent.	Si	0.650	No	-
	Grande lat.	No	-	Si	-
	Borroso	No	-	-	-
	Oculto	No	-	-	-
	Pequeño	No	-	-	-
	No es *stop*	No	-	-	-
YOLOV3	Grande cent.	Si	0.600	No	-
	Grande lat.	No	-	-	-
	Borroso	No	-	-	-
	Oculto	No	-	-	-
	Pequeño	No	-	-	-
	No es *stop*	No	-	-	-
YOLOV4	Grande cent.	No	1	No	-
	Grande lat.	No	1	No	-
	Borroso	No	0.703	No	-
	Oculto	No	-	-	-
	Pequeño	No	-	-	-
	No es *stop*	No	0.999	Si	1
Faster-RCNN	Grande cent.	Si	0.999	No	-
	Grande lat.	Si	0.999	No	-
	Borroso	Si	0.580	No	-
	Oculto	Si	0.920	No	-
	Pequeño	Si	0.600	No	-
	No es *stop*	Si	0.930	Si	2
RCNN	Grande cent.	Si	1	No	-
	Grande lat.	Si	1	Si	2
	Borroso	Si	0.790	Si	2
	Oculto	No	-	-	-
	Pequeño	No	-	-	-
	No es *stop*	No	-	-	-

Tabla 27. Resumen del comportamiento de la detección de cada uno de los detectores de la Tabla 26 con mejor precisión promedio

En las figuras 37 a 42 se muestran imágenes ilustrativas que representan los objetos comentados en la Tabla 27.

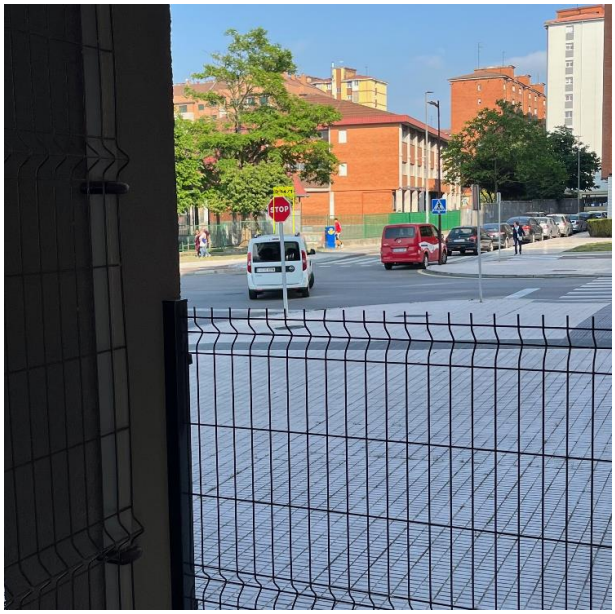

Figura 37. (Grande cent.) *Imagen real para objeto centrado con tamaño mediano/grande con umbral de 0.5, con detector SSD y ResNet50*

Figura 38. (Pequeño.) *Imagen real para objeto pequeño centrado con umbral de 0.5, con detector Faster-RCNN y ResNet18*

Figura 39. (No es *stop*) *Imagen real para objeto que no es señal de stop con umbral de 0.5 y YOLO V4 con ResNet50 y optimizador Adam*

Figura 40. (Oculto) *Imagen real para objeto parcialmente oculto con umbral de 0.5 y Faster-RCNN con ResNet18*

Figura 41. (Grande lat.) *Imagen real para objeto lateral con tamaño mediano/grande con umbral de 0.5 y RCNN con ResNet101*

Figura 42. (Borroso) *Imagen real para objeto borroso con umbral de 0.5 y RCNN con ResNet101*

6. Conclusiones y trabajo futuro

6.1. Conclusiones

Como se ha visto reflejado a lo largo del presente trabajo se ha demostrado que se puede hallar una solución al problema de la detección de objetos aplicando técnicas de Aprendizaje Profundo. De forma más concreta, se ha demostrado cómo la aplicación de la teoría de las redes neuronales convolucionales con las arquitecturas superiores de detección de objetos, tanto las de un paso (SSD, YOLOV3 y YOLOV4) como las de dos (RCNN y Faster-RCNN), permiten llevar a cabo estas detecciones.

Otro objetivo del trabajo, el uso de imágenes reales procedentes de un *dataset* genérico con miles de imágenes, ha sido gratamente cumplido utilizando Matlab, aunque en el camino se haya tenido que hacer uso de plataformas de cálculo y computación en la nube debido al alto coste computacional del entrenamiento de algunos de los detectores.

Desde un punto de visto genérico de los resultados, se puede concluir cómo los detectores de un solo paso presentan, por lo general, resultados aceptables sin incurrir en grandes tiempos de aprendizaje y detección (ver por ejemplo Tabla 15 para los tiempos y Tabla 26 para el resumen de detectores). Existe un caso particular como es el de YOLOV4, el cual tiene problemas de detección en su uso con Matlab. Este parece más bien estar motivado por la generación del propio detector en Matlab a partir de una CNN preentrenada genérica, ya que, en el uso de una red particular ya preentrenada en COCO y disponible en Matlab para los detectores YOLO (csp-darknet53-coco), presenta buenos resultados tras un entrenamiento ajustado a la clase de objeto "*stop sign*" usado en esta memoria.

Por otro lado, entre estos detectores de un paso, destacan los del tipo SSD, con mejores resultados en cuanto a precisión promedio y *recall* en casi todas las variaciones comunes realizadas entre los cinco tipos de detectores.

Entre los detectores de dos pasos, se puede apreciar cómo los resultados de Faster-RCNN son mejores en casi todas las variaciones comunes de detectores realizadas frente a las de RCNN. Esto viene a ratificar la mejora teórica de Faster-RCNN frente a RCCN en función de sus arquitecturas, tal y como se vio en la sección Faster-RCNN, 3.3.3.

Siguiendo con las conclusiones de carácter general, se aprecia la importancia de la tasa de aprendizaje en la búsqueda de un óptimo global en el proceso de retropropagación realizada en el entrenamiento, así como del optimizador utilizado. En los detectores estudiados, prácticamente en todos, la reducción de la tasa de aprendizaje o *learning rate* provocó pérdidas de precisión promedio aun utilizando un número alto de épocas durante el entrenamiento. Al contrario, pasa con el aumento del factor L2 (probado con SSD), que penaliza resultados para evitar problemas de sobreajustes, ya que apenas tiene influencia hasta alcanzar valores muy altos. Algo similar ocurre con el aumento de épocas si ya están correctamente entrenados los detectores y el tamaño de imagen es apropiado (los

escalados internos en las redes para los tamaños de capa de entrada de las CNN base de los detectores limitan su efecto).

En cuanto a particularidades, se puede ver cómo los detectores SSD presentan resultados aceptables en casi todas sus variaciones y apenas están influenciados por cambios de orden de las capas de extracción (en teoría capas más profundas deberían ser usadas para *anchor boxes* mayores como se explicó en la sección 3.3.4.).

De igual forma, se observa cómo YOLOV3 se comporta mucho mejor con redes menos densas como *squeezenet* y mal en las muy densas, a pesar de tener similitudes en la extracción de mapas de características en varios niveles de la red CNN base como se determinó en la sección 3.3.5. . Por último, en cuanto a Fast-RCNN se puede ver cómo tiene importancia el tipo de red CNN en la precisión, pero no tanto en RCNN. Al eliminar el uso de capas de extracción se minimiza el criterio subjetivo de la selección en la red, que hace mejorar unos detectores frente a otros al margen de densidad o arquitectura de los detectores.

También, cabe destacar que tal y como se deduce de la Tabla 27, la aplicación en inferencia sobre imágenes reales no obtiene buen comportamiento en casi todos los detectores y variantes. Sobre imágenes reales las mejores inferencias se consiguen con Faster-RCNN dado su mejor *confidence score*.

Por último, cabe destacar que, a lo largo de los entrenamientos y pruebas, se vio que el mayor problema de precisión que presentaban los detectores residía en el hecho de que MS COCO, al menos en la clase usada "*stop sign*", incorpora muchas imágenes reales de dicho objeto de dimensiones reducidas en píxeles (algunas casi inapreciables) e incluso generando etiquetados incorrectos. Esto viene a explicar, por qué el modelo SSD es el que mejor resultados obtiene, ya que tal y como se vio en la sección 3.3.4. su arquitectura se basa principalmente en la extracción en varios niveles de la red para la mitigación de problemas de detección de objetos pequeños frente a objetos grandes.

6.2. Trabajo futuro

Este trabajo se puede considerar como una base sólida para avanzar en trabajos futuros, una vez comprobada la correcta integración entre elementos y plataformas. Se pueden realizar, por tanto, multitud de variaciones y ampliaciones atendiendo tanto a la modificación de reglas fijadas en el entrenamiento entre detectores como a su propia creación. Al margen, quedaría también pendiente la mejora de los detectores creados con YOLOV4.

No obstante, una primera ampliación podría estar basada en obtener estos detectores entrenados para otras clases de las incluidas dentro de MS COCO o incluso en todas las clases a la vez, aunque su coste computacional sea elevado y titánico para las más de 30 variaciones de detectores creadas y usadas en este proyecto. En la misma línea estaría la selección de otras capas de extracción en los mapas de características de los detectores, con el fin de analizar correctamente su efecto, el cual se intuye muy importante en vista de los resultados obtenidos.

En otro camino, estaría la mejora del sobreajuste que parece intuirse en muchos de los detectores creados, ya que el error del conjunto de validación, en muchos de los entrenamientos, solía ser mayor que el error del conjunto de entrenamiento.

También queda pendiente la mejora de detección de objetos pequeños en las imágenes, ya que la existencia de ellos es abundante en el conjunto de datos de MS COCO. Para ello, pueden realizarse diversas modificaciones dentro de las distintas posibilidades que se dan para mejorarlas: aumentación de imágenes, incremento de la resolución del detector en aquellos que lo permitan o nuevos detectores como versiones más nuevas de YOLO, tales como YOLOV5 y superiores o "High-resolution Detection Network".

Por último, un estudio muy interesante, a partir de los resultados de este trabajo, sería el análisis de por qué unos detectores tienen mejores comportamientos que otros, aun con los mismos parámetros de entrenamiento (aunque con distinta red base CNN).

Sería, por tanto, objeto de introducción de técnicas de inteligencia artificial explicativas (xAI, Explainable Artificial Intelligence), que permitan conocer qué características son las que realmente se están extrayendo del conjunto de datos de entrenamiento. De esta forma, se puede comparar con los datos de la presente memoria y concluir las relaciones existentes.

7. Bibliografía

[1] S. K. Pal, A. Pramanik, J. Maiti y P. Mitra , «Deep learning in multi-object detection and tracking: state of the art,» *Applied Intelligence; https://doi.org/10.1007/s10489-021-02293-7*, vol. 51, p. 6400–6429, 2021.

[2] J. Jeong, H. Park y N. Kwak, «Enhancement of SSD by concatenating feature maps for object detection,» *arXiv preprint arXiv:1705.09587,* 2017.

[3] X. Lu, X. Kang, S. Nishide y F. Ren, «Object detection based on SSD-ResNet,» de *2019 IEEE 6th International Conference on Cloud Computing and Intelligence Systems (CCIS); DOI: 10.1109/CCIS48116.2019.9073753*, Singapore , 2019.

[4] Y. Xiao, Z. Tian, J. Yu, Y. Zhang, S. Liu, S. Du y X. Lan , «A review of object detection based on deep learning,» *Multimedia Tools and Applications; https://doi.org/10.1007/s11042-020-08976-6*, vol. 79, p. 23729–23791, 2020.

[5] P. Z. Y. J. P. e. a. Ding, «A Comparison: Different DCNN Models for Intelligent Object Detection in Remote Sensing Images,» *Neural Process Letters; https://doi.org/10.1007/s11063-018-9878-5*, vol. 49, p. 1369–1379, 2019.

[6] J. Pedoeem y R. Huang, «YOLO-LITE: A Real-Time Object Detection Algorithm Optimized for Non-GPU Computers,» *IEEE international conference on big data (big data),* pp. 2503-2510, 2018.

[7] B. Sikka, Elements of Deep Learning for Computer Vision, BPB Publications, 2021.

[8] N. Andrew, N. Jiquan, F. Y. Chuan , M. Yifan, S. Caroline, C. Adam, M. Andrew, H. Awni, H. Brody, W. Tao y T. Sameep, «UFLDL Tutorial,» *http://ufldl.stanford.edu/tutorial/*, 2023.

[9] G. Pajares Martinsanz, P. J. Herrera Caro y E. Besada Portas, Aprendizaje profundo, RC Libros, 2021.

[10] S. S. Abbas Zaidi, M. S. Ansari, A. Aslam, N. Kanwal, M. Asghar y B. Lee, «A survey of modern deep learning based object detection models,» *Digital Signal Processing,* vol. 126, nº 103514, 2022.

[11] E. Khoramshahi, R. Oliveira, N. Koivumäki y E. Honkavaara, «An Image-Based Real-Time Georeferencing Scheme for a UAV Based on a New Angular Parametrization,» *Remote Sensing3185https://doi.org/10.3390/rs12193185,* vol. 12(19), nº 3185, 2020.

[12] U. Özaydın, T. Georgiou y M. Lew, «A Comparison of CNN and Classic Features for Image Retrieval; https://doi.org/10.1109/CBMI.2019.8877470,» 2019.

[13] M. Stewart, «Simple Introduction to Convolutional Neural Networks,» *Towards Data Science; https://towardsdatascience.com/simple-introduction-to-convolutional-neural-networks-cdf8d3077bac,* 2019.

[14] Deingenierias.com, «De Ingenierías,» *https://deingenierias.com/inteligencia-artificial/redes-neuronales-en-inteligencia-artificial/,* 2019.

[15] H. Carrera, S. Sinche y P. Hidalgo, «Modelo para detectar el uso correcto de mascarillas en tiempo real utilizando redes neuronales convolucionales,» *Revista de Investigación en Tecnologías de la Información; https://dialnet.unirioja.es/servlet/articulo?codigo=7741839,* vol. 9, nº Extra 17, pp. 111-120, 2021.

[16] Na8, «Aprende Machine Learning,» *https://www.aprendemachinelearning.com/como-funcionan-las-convolutional-neural-networks-vision-por-ordenador/,* 2018.

[17] U. Michelucci, Advanced Applied Deep Learning: Convolutional Neural Networks and, APRESS, 2019.

[18] R. Alake, «Towards Data Science: Deep Learning, Understanding The Inception Module,» *https://towardsdatascience.com/deep-learning-understand-the-inception-module-56146866e652,* 2020.

[19] R. Rodríguez Abril, «La Máquina del Oráculo: Redes Residuales: ResNet,» *https://lamaquinaoraculo.com/computacion/redes-residuales/,* 2023.

[20] J. Pingel, «MathWorks: Deep Learning Q&A: All about Pretrained Models,» *https://es.mathworks.com/campaigns/offers/next/all-about-pretrained-models.html,* 2023.

[21] A. Anwar, «Towards Data Science: Difference between AlexNet, VGGNet, ResNet, and Inception,» *https://towardsdatascience.com/the-w3h-of-alexnet-vggnet-resnet-and-inception-7baaaecccc96,* 2019.

[22] M. Shatnawi, F. Albreiki, A. Alkhoori y M. Alhebshi, «Deep Learning and Vision-Based Early Drowning Detection,» *Information; https://doi.org/10.3390/info14010052,* vol. 14, nº 52, 2023.

[23] P. Jaccard, «Distribution de la flore alpine dans le bassin des Dranses et dans quelques régions voisines,» *Bulletin de la Société vaudoise des sciences naturelles,* nº 37, pp. 241-272, 1901.

[24] MathWorks, «MathWorks: estimateAnchorBoxes,» *https://es.mathworks.com/help/vision/ref/estimateanchorboxes.html,* 2023.

[25] R. D. J. D. T. M. J. Girshick, «Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation,» *2014 IEEE Conference on Computer Vision and Pattern Recognition*, nº CVPR'14, pp. 580-587, 2014.

[26] H. T. T. V. G. L. Bay, « SURF: Speeded Up Robust Features,» *Computer Vision and Image Understanding*, nº 2008, pp. 346-359, 2008.

[27] D. Lowe, «Distinctive Image Features from Scale-Invariant Keypoints,» *International Journal of Computer Vision*, nº 60(2), pp. 91-110, 2004.

[28] MathWorks, «Mathworks: Getting Started with R-CNN, Fast R-CNN, and Faster R-CNN,» *https://es.mathworks.com/help/vision/ug/getting-started-with-r-cnn-fast-r-cnn-and-faster-r-cnn.html*, 2023.

[29] S. Ren, K. He, R. Girshick y J. Sun, «Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks,» *IEEE Transactions on Pattern Analysis and Machine Intelligence*, nº 39(6), pp. 1137-1149, 2017.

[30] R. Girshick, «Fast R-CNN,» de *IEEE International Conference on Computer Vision (ICCV) pp.1440-1448, 7-13*, Santiago, Chile, 2015.

[31] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C. Fu y A. Berg, «SSD: Single shot multibox detector,» de *In Proc. of the European Conference on Computer Vision, pp.21-37 arXiv:1512.02325 [cs.CV]*, Amsterdam, The Netherlands, 2016.

[32] MathWorks, «MathWorks: Create SSD Object Detection Network,» *https://se.mathworks.com/help/vision/ug/create-ssd-object-detection-network.html#CreateSSDObjectDetectionNetworkExample-5*, 2023.

[33] J. Redmon, S. Divvala, R. Girshick y A. Farhadi, «You Only Look Once: Unified, Real-Time Object Detection,» de *Proc. IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 779–788*, Las Vegas, USA, 2016.

[34] J. Redmon y A. Farhadi, «YOLO9000: Better, Faster, Stronger,» de *Proc. IEEE Conf. on Computer Vision and Pattern Recognition (CVPR'17), 6517–6525*, Honolulu, USA, 2017.

[35] J. Redmon y A. Farhadi, «Yolov3: An incremental improvement,» de *ArXiv abs/1804.02767*, 2018.

[36] A. Bochkovskiy, C. Y. Wang y H. Y. Mark-Liao, «Yolov4: Optimal speed and accuracy of object detection,» de *arXiv:2004.10934v1 [cs.CV]*, 2020.

[37] K. He, X. Zhang, S. Ren y J. Sun, «Spatial Pyramid Pooling in Deep Convolutional Networks for Visual Recognition,» *IEEE transactions on pattern analysis and machine intelligence*, nº 37(9), pp. 1904-1916, 2015.

[38] ArcGIS Pro, «ArcGIS Pro: Cómo funciona Calcular precisión para la detección de objetos,» *https://pro.arcgis.com/es/pro-app/latest/tool-reference/image-analyst/how-compute-accuracy-for-object-detection-works.htm*, 2023.

[39] R. Abril Rodríguez, «La Máquina del Oráculo: Detección de objetos I: YOLO,» *https://lamaquinaoraculo.com/computacion/deteccion-de-objetos/*, 2023.

[40] IBM, «IBM: IBM Maximo Visual Inspection, Métricas,» *https://www.ibm.com/docs/es/maximo-vi/continuous-delivery?topic=models-understanding-metrics*, 2023.

[41] A. Sharma, «Pyimagesearch: Mean Average Precision (mAP) Using the COCO Evaluator,» *https://pyimagesearch.com/2022/05/02/mean-average-precision-map-using-the-coco-evaluator/*, 2022.

[42] «The Mathworks,» *https://es.mathworks.com/*, 2023.

[43] «MathWorks: cell,» *https://es.mathworks.com/help/matlab/ref/cell.html*, 2023.

[44] Som, Medium, «Medium: Coco dataset, What is it? and How can we use it?,» *https://medium.com/mlearning-ai/coco-dataset-what-is-it-and-how-can-we-use-it-e34a5b0c6ecd*, 2021.

[45] «GitHUb: cocoapi, MatlabAPI,» *https://github.com/cocodataset/cocoapi/tree/master/MatlabAPI*, 2018.

[46] «Amazon AWS,» *https://aws.amazon.com/es/*, 2023.

[47] Instances, «Vantage: Instance Details, g5.2xlarge,» *https://instances.vantage.sh/aws/ec2/g5.2xlarge?region=us-west-2&os=linux&cost_duration=hourly&reserved_term=null*, 2023.

[48] GitHub, «GitHub: MATLAB on Amazon Web Services,» *https://github.com/mathworks-ref-arch/matlab-on-aws*, 2023.

[49] MathWorks, «MathWorks: Object Detection Using SSD Deep Learning,» *https://es.mathworks.com/help/deeplearning/ug/object-detection-using-ssd-deep-learning.html*, 2023.

[50] COCODataset, «COCODataset,» *https://cocodataset.org/#download*, 2023.

Listado de siglas, abreviaturas y acrónimos

AMI	*Amazon Machine Images*
AP	*Aprendizaje profundo*
API	*Application Programming Interface*
AWS	*Amazon Web Services*
CNN	*Convolutional Neural Network*
CPU	*Central Processing units*
DP	*Deep learning*
EC2	*Elastic Compute Cloud*
Faster-RCNN	*Faster Region-based Convolution Neural Network*
Fast-RCNN	*Fast Region-based Convolution Neural Network*
FC	*Fully connected*
FN	*Falso negativo*
FP	*Falso positivo*
GHz	*Gigahertz*
GiB	*Gibibyte*
GPU	*Graphics Processing units*
IoU	*Intersection of Union*
IP	*Internet Protocol*
JSON	*JavaScript Object Notation*
MS COCO	*Microsoft Common Objects in Context*
RCNN	*Region-based Convolution Neural Network*
RDP	*Remote Desktop Protocol*
ReLU	*Rectified Lineal Unit*
ROI	*Region of Interest*

RPN	*Region Proposal Network*
SPP	*Spatial Pyramid Pooling*
SSD	*Single-Shot Detector*
SSH	*Secure SHell*
SVM	*Support Vector Machine*
VN	*Verdadero negativo*
VP	*Verdadero positivo*
VPC	*Virtual Private Cloud*
XAI	*Explainable artificial intelligence*
YOLO	*ou Only Look Once*
YOLO V1	*You Only Look Once version 1*
YOLO V2	*You Only Look Once version 2*
YOLO V3	*You Only Look Once version 3*
YOLO V4	*You Only Look Once version 4*