ESCUELA TÉCNICA SUPERIOR DE INGENIEROS DE TELECOMUNICACIÓN

GRADO EN INGENIERÍA DE TECNOLOGÍAS DE TELECOMUNICACIÓN

Evaluación de algoritmos de detección de objetos basados en deep learning para detección de incidencias en carreteras

Alumno: JORGE SÁNCHEZ-ALOR EXPÓSITO

Tutores: JUAN PABLO CASASECA DE LA HIGUERA

JAVIER AGUIAR PÉREZ

Valladolid, 30 de Septiembre de 2020

Evaluación de algoritmos de detección de objetos basados en deep learning para detección de incidencias en carreteras

Jorge Sánchez-Alor Expósito

AGRADECIMIENTOS

En primer lugar, quiero expresar mi gratitud a mis tutores Juan Pablo Casaseca de la Higuera y Javier Aguiar Pérez por darme la posibilidad de realizar este Trabajo Fin de Grado y por sus indicaciones y paciencia.

En segundo lugar me gustaría también agradecer al Laboratorio de Procesado de Imagen de la Universidad de Valladolid por haber invertido en medios y ponerlos a disposición de los alumnos para el desarrollo de este y otros Trabajos Fin de Grado.

Por último, quiero dar las gracias a mi familia, compañeros y amigos, por el apoyo y el ánimo que me han brindado, especialmente a mis padres y a mis hermanos, que me han todo su apoyo y cariño en los momentos más oscuros.

RESUMEN

La detección de objetos es una capacidad muy útil para el desarrollo de nuevas aplicaciones de visión artificial, en diferentes ámbitos de la vida cotidiana, en concreto, es especialmente útil en el campo de la conducción asistida. Sin embargo, los requisitos computacionales que requieren suelen ser limitantes a la hora de implementarlos en un sistema embebido. Conocer el rendimiento de los diferentes modelos de detección dentro de un sistema concreto es de gran utilidad a la hora de tomar decisiones de diseño.

El objetivo de este trabajo es diseñar mediante la programación en Python haciendo uso de Tensorflow, un sistema capaz de detectar y rastrear elementos en un entorno de trabajo centrado en la automoción. Para ello se plantea el estudio del rendimiento de diferentes modelos de detección de objetos para determinar cuál es el que se adapta mejor a nuestros sistemas, así como el entrenamiento en una base de datos específica de automoción “Berkeley Deep Dive 100k” (BDD100k). El programa de detección y seguimiento se ha creado de tal forma que sea sencillo cambiar de un detector de objetos a otro si los requisitos cambiaran.

Palabras clave: *Aprendizaje profundo, rastreo, detección de objetos, tensorflow, SSD, F-RCNN, YOLO, filtro de Kalman*.

ABSTRACT

Object detection is a very useful ability for the development of new applications of artificial vision, in different areas of everyday life, specifically, it is especially useful in the field of assisted driving. However, the computational requirements they require are often limiting when implementing them in an embedded system. Knowing the performance of different detection models within a specific system is very useful when making design decisions.

The objective of this work is to design through programming in Python using Tensorflow, a system capable of detecting and tracking elements in an automotive-focused work environment. For this, the study of the performance of different object detection models is proposed to determine which one best suits our systems, as well as training in a specific automotive database "Berkeley Deep Dive 100k" (BDD100k). The detection and monitoring program has been created in such a way that it is easy to change from one object detector to another if the requirements change.

Key words: *Deep learning, tracking, object detection, tensorflow, SSD, F-RCNN, YOLO, Kalman filter*.

INDICE GENERAL

ÍNDICE DE FIGURAS

Capítulo 1
INTRODUCCIÓN

En este capítulo, se busca establecer el contexto del presente trabajo, explicando el problema que se desea resolver, así como los objetivos y la estructura de la memoria.

1.1 MOTIVACIÓN DEL PROBLEMA

De un tiempo a esta parte, el incremento en las capacidades de procesamiento de los ordenadores ha permitido el desarrollo de nuevas tecnologías, que antes debido a limitaciones técnicas eran impensables. Una de ellas es el Aprendizaje Profundo o *Deep Learning*, y más concretamente, su aplicación al campo de detección de objetos en imágenes y vídeos. Dentro de este campo, una de las aplicaciones es la detección de objetos en sistemas de conducción asistida o autónoma.

Con esta idea en mente, se desea conseguir un sistema que sea capaz de detectar, a partir de una señal de vídeo entrante, elementos en el ámbito de la automoción y que a su vez realice un seguimiento de dichos elementos, asignándoles un identificador, incurriendo en el menor retardo posible.

Aunque cada vez las computadoras tienen mejor precisión y tiempo de respuesta, debido a las restricciones de tamaño y consumo de potencia que se presentan los dispositivos de menor tamaño, el rendimiento eficiente es un punto importante a la hora de desarrollar sistemas dentro de este marco. Preguntas cómo, qué dispositivos emplear o qué algoritmos implementar dentro del prototipo son críticos. Por ello hacer un estudio previo del rendimiento de las diferentes partes del sistema es muy importante para el trabajo posterior.

Aunque existen tablas en las que se presentan comparativas de los diferentes algoritmos, como la que se puede ver en el Model Zoo de Tensorflow [1], éstas están sujetas al hardware que se empleó para su medición, y es muy probable que al ejecutarlas en nuestro sistema disten mucho de las originales. Es por ello que ser capaces de realizar las pruebas y evaluaciones en nuestro propio sistema es de vital importancia. Así, el objetivo que se pretende conseguir con este trabajo es realizar en *benchmark* que realice la evaluación de los diferentes modelos de detección de objetos para poder elegir el que mejor se ajuste a un hardware prototipo futuro.

De este modo se evaluarán los detectores que ofrece el equipo de Tensorflow en su Model Zoo [1] y otros basados en *You Only Look Once* (YOLO), para seleccionar un par de ellos para ser entrenados en la base de datos de objetos de automoción Berkeley DeepDrive [2], y medir su desempeño al incluir un proceso adicional de rastreo.

Por lo tanto, en el presente documento primero se expondrán los conceptos clave y las bases del aprendizaje automático y la detección de objetos, primero de una forma más general, y después centrándonos en los modelos y herramientas que se usarán posteriormente para desarrollar las aplicaciones. Por último se presentarán el trabajo realizado y las conclusiones.

1.2 OBJETIVOS

El objetivo principal del trabajo es obtener un programa que sea capaz de realizar una evaluación en cuanto a velocidad y precisión de diferentes algoritmos de detección de objetos que pueda ser ejecutado en máquinas objetivo para poder elegir el que mejor se ajuste a las necesidades y desempeño del sistema en cuestión. Un objetivo secundario es entrenar alguno de los algoritmos en la base de datos Berkeley DeepDrive y añadir un algoritmo de rastreo adicionalmente y medir su efecto en el rendimiento.

Para ello se han fijado los siguientes objetivos:

- Realizar un estudio de la tecnología disponible en el campo de la detección de objetos, con el fin de obtener los conocimientos necesarios para poder comprender las herramientas.
- Desarrollar un *benchmark* para evaluar el rendimiento en términos de velocidad y precisión de un conjunto de detectores. La idea es que este banco de pruebas se pueda ejecutar en el sistema para determinar qué detector es el que se ajusta mejor a nuestras limitaciones de hardware.
- Entrenar el/los detector/es en la base de datos de automoción Berkeley DeepDrive. El entrenamiento se realiza utilizando la técnica del *transfer learning*, en la que se aprovecha los parámetros de las redes neuronales ya entrenadas en otra base de datos como punto de partida para realizar el entrenamiento [3].
- Elegir e implementar algoritmos de seguimiento que no afecten demasiado a los fps y que se ajuste mejor a nuestro problema.
- Realizar una evaluación sobre dichos rastreadores para conocer su impacto y su precisión.
- Presentar las conclusiones finales después de estudiar los resultados obtenidos.

1.3 METODOLOGÍA A SEGUIR

Para llevar a cabo los objetivos marcados en el epígrafe anterior se procederá de la siguiente manera:

- Leer la bibliografía existente sobre aprendizaje profundo, detectores de objetos y rastreadores.
- Familiarizarse con los entornos de programación Python y Tensorflow, así como con el uso de entornos de programación.
- Familiarizarse con los conjuntos de datos (*datasets*) y sus diferentes formatos.
- Realizar un programa en Tensorflow para realizar las tareas de medición del rendimiento.
- Comprender el proceso de entrenamiento de un detector de objetos sobre un *dataset* concreto.
- Elegir e implementar los algoritmos de seguimiento para poder utilizarlos con los detectores de objetos entrenados.

- Modificar los rastreadores para que proporcionen los datos necesarios para poder realizar una evaluación.
- Medir los parámetros y presentar los resultados.
- Presentar las conclusiones

1.4 ESTRUCTURA DE LA MEMORIA

Esta memoria se divide en los siguientes capítulos:

El presente capítulo donde se presenta la motivación del problema, los objetivos que se desean conseguir y la estructura de la memoria.

En el Capítulo 2 se da una visión global de los elementos necesarios para llevar a cabo la detección de objetos y se presentan los elementos que se emplearán posteriormente en el desarrollo del programa, como el reconocimiento de patrones, las redes neuronales, los modelos de detección de objetos, etc.

Posteriormente, en el Capítulo 3, se presenta la metodología que se ha llevado a cabo para ir consiguiendo cumplir los objetivos que se han marcado. Primero se hace el estudio del rendimiento de los diferentes modelos de detección de objetos, para después describir el proceso de entrenamiento de dos de los modelos que se han escogido, para incluirlos posteriormente en el programa que realiza la detección y el seguimiento.

En el Capítulo 4, se presentan los resultados obtenidos de las ejecuciones de nuestro programa.

En el Capítulo 5 se aportan las conclusiones a las que se ha llegado después de la realización de todo el proceso y las posibles líneas futuras para mejorar la implementación propuesta, la bibliografía y los apéndices donde se encuentra información adicional sobre algunos de los aspectos programáticos y los enlaces a los repositorios donde se encuentra el código de los programas.

Capítulo 2

ANTECEDENTES Y ESTADO DEL ARTE

En este capítulo se explicarán los principios en los que se basa la detección de objetos mediante *deep learning* además de presentar los tres modelos de detectores de objetos que son los más empleados en la actualidad. Así mismo, también se expondrán los diferentes métodos de seguimiento (tracking) de objetos para poder elegir uno de ellos para el proyecto.

2.1 MACHINE LEARNING VS DEEP LEARNING

Los pioneros de la Inteligencia Artificial (IA) soñaban con construir complejas máquinas que tuvieran las mismas características que la inteligencia humana allá por la década de los 50. En la actualidad, aunque aún nos sigue pareciendo lejano el hecho de programar algo tan complejo como la mente humana, estamos viviendo un avance tremendo en el uso del *Machine learning* [4], y desde hace unos cuantos años del *Deep learning* [5], con el objetivo de conseguir que las máquinas puedan desempeñar tareas que usualmente requieran de un humano, con la misma o mayor eficiencia.

Aunque los medios sigan tratando de forma indiferente ambos términos (*Machine learning* y *Deep learning*) vamos a aclarar algunos conceptos en más profundidad y, sobre todo el impacto que está teniendo en los avances tecnológicos presentes y lo que está por llegar, tanto en la industria del software como en nuestra vida cotidiana.

El *Machine learning* en su uso más básico es la práctica de usar algoritmos para analizar datos, aprender de ellos y luego ser capaces de hacer una predicción o sugerencia sobre algo. Los programadores deben perfeccionar algoritmos que especifiquen un conjunto de variables para ser lo más precisos posibles en una tarea en concreto. La máquina es entrenada utilizando una gran cantidad de datos dando la oportunidad a dichos programadores de perfeccionar los algoritmos.

Desde los primeros albores de la temprana inteligencia artificial, los algoritmos han evolucionado con el objetivo de analizar y obtener mejores resultados: árboles de decisión, programación lógica inductiva (ILP), clustering para almacenar y leer grandes volúmenes de datos, redes Bayesianas y un numeroso abanico de técnicas que los programadores en ciencia de datos (*data science)* [6] pueden aprovechar.

Siguiendo la evolución del *Machine learning* en la última década se ha popularizado con más fuerza una técnica concreta de *Machine learning* conocida como *Deep learning*. Por definición, *Deep learning* es un subconjunto dentro del campo del *Machine learning*, el cual predica con la idea del aprendizaje desde el ejemplo.

En *Deep learning*, en lugar de enseñarle al ordenador una lista enorme de reglas para solventar un problema, le damos un modelo que pueda evaluar ejemplos y una pequeña

colección de instrucciones para que él mismo modifique el modelo cuando se produzcan errores. Con el tiempo esperamos que esos modelos sean capaces de solucionar el problema de forma extremadamente precisa, gracias a que el sistema es capaz de extraer patrones.

Aunque existen distintas técnicas para implementar *Deep learning*, una de las más comunes es simular un sistema de redes artificiales de neuronas dentro del software de análisis de datos.

Ahora existen recursos y tecnología suficiente para poder tener al alcance de la mano el uso del *Deep learning*. Una de las herramientas, liberada por Google, más comunes para trabajar en este ámbito es TensorFlow [7] que permite aplicar *Deep learning* y otras técnicas de *Machine learning* de una forma bastante potente.

También existen otros servicios auspiciados por otros grandes actores en el tema de software como IBM, Amazon o Microsoft: IBM Watson Developer Cloud, Amazon Machine Learning o Azure Machine Learning.

El *Deep learning* nos está empujando a otra realidad en la que seamos capaces de interpretar de otra forma nuestro mundo a través del reconocimiento de imágenes, el análisis del lenguaje natural y anticiparnos a muchos problemas gracias a la extracción de patrones de comportamiento, que resulta tremendamente costoso con el *Machine learning* convencional.

Entre los retos del aprendizaje automático podemos señalar la necesidad de aumentar la capacidad de procesamiento para poder entrenar las cada vez mayores redes. En este punto una de las mejoras llevadas a cabo estos años ha sido el uso de *GPUs* para realizar estos trabajos de forma más eficiente, lo cual ha ahorrado la necesidad de disponer de gran cantidad de ordenadores para realizar los cálculos. NVIDIA es uno de los principales impulsores de esta tecnología adaptando muchos de sus componentes a esta nueva realidad [8], tanto en la investigación como en el uso de procesadores para la IA de forma autónoma como en los vehículos o drones.

Otro de los retos más importantes es optimizar el uso de grandes volúmenes de datos para extraer patrones de ellos. Es necesario adecuar el almacenamiento de esos datos, indexarlos, y que el acceso sea lo suficientemente rápido para que pueda escalar. Para ello seguimos contando con los *frameworks* disponibles en Big Data [9] como Hadoop y Spark, acompañados de una amplia variedad de bases de datos NoSQL.

Y por último, la precisión. El problema aquí no es ofrecer una precisión muy elevada, si no proporcionar la suficiente precisión para llevar a cabo la tarea encomendada cumpliendo otros requisitos de tiempo o capacidad de procesamiento. Ahí es dónde está el verdadero reto del *deep learning*.

2.2 REDES NEURONALES

Las Redes Neuronales son un campo muy importante dentro de la Inteligencia Artificial. Inspirándose en el comportamiento conocido del cerebro humano (principalmente el referido a las neuronas y sus conexiones), trata de crear modelos artificiales que solucionen problemas difíciles de resolver mediante técnicas algorítmicas convencionales.

2.2.1 Elementos de red neuronal

Una red neuronal está compuesta por un número de neuronas artificiales. Esta última, se puede definir como un dispositivo que a partir de un conjunto de entradas, x_i (i=1...n) o vector $\boldsymbol{x}$, genera una única salida y.

Un nodo o neurona, combina la entrada de los datos con un conjunto de coeficientes, o pesos, que amplifican o amortiguan esa entrada, asignando así importancia a las entradas con respecto a la tarea que el algoritmo está tratando de aprender. Estos productos de peso de entrada se suman y luego la suma se pasa a través de la llamada función de activación de un nodo, para determinar si esa señal debería progresar aún más a través de la red y, en qué medida, afectar el resultado final. Si las señales pasan, la neurona se ha "activado".

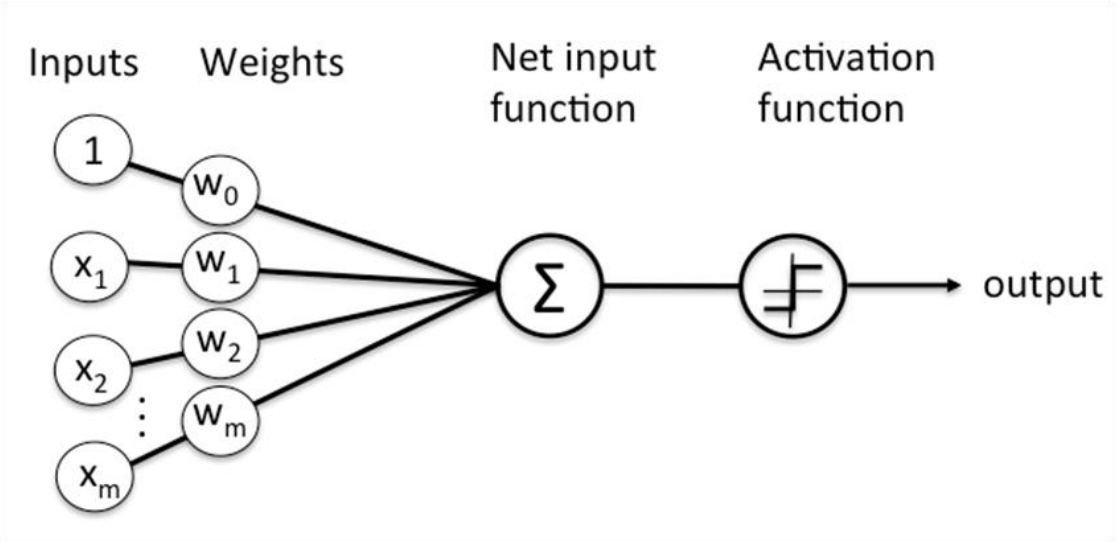

Figura 2-1 Ejemplo de Nodo de una red neuronal

En la Figura 2-1, se muestra un diagrama de cómo se vería un nodo, donde se presentan inicialmente un número determinado de entradas, a las cuales se les aplica un peso para agruparlas posteriormente. El resultado se hace pasar por una función de activación que, para simplificar, se puede considerar como un interruptor, que dependiendo del valor de la señal de entrada permite que ésta se propague a la salida o no.

Una capa (Figura 2-2) es una fila de esos interruptores similares a neuronas que se activan o desactivan cuando la entrada se alimenta a través de la red. La salida de cada capa es simultáneamente la entrada de la capa posterior, comenzando desde una capa de entrada inicial que recibe sus datos.

Emparejando los pesos ajustables del modelo con las características de entrada asignamos importancia a esas características con respecto a cómo la red neuronal clasifica y agrupa la entrada.

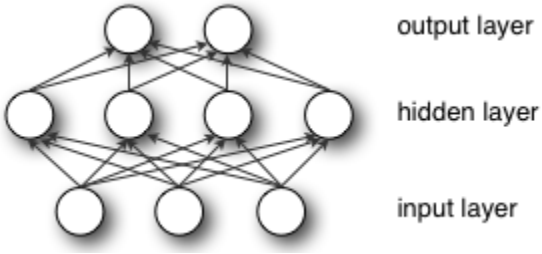

Figura 2-2. Esquema de capas de una red neuronal.

Las redes neuronales que se emplean en el aprendizaje profundo se distinguen de las redes neuronales de capa oculta única más comunes por su profundidad; es decir, el número de capas de nodo a través de las cuales deben pasar los datos en un proceso de varios pasos de reconocimiento de patrones. Las versiones anteriores de redes neuronales, como los primeros perceptrones, eran poco profundas, compuestas de una capa de entrada y una de salida, y como máximo una capa oculta en el medio. No obstante, hay que destacar la idea de que no todas las redes neuronales profundas se utilizan para aprendizaje profundo, pero sí todo aprendizaje profundo emplea redes neuronales profundas.

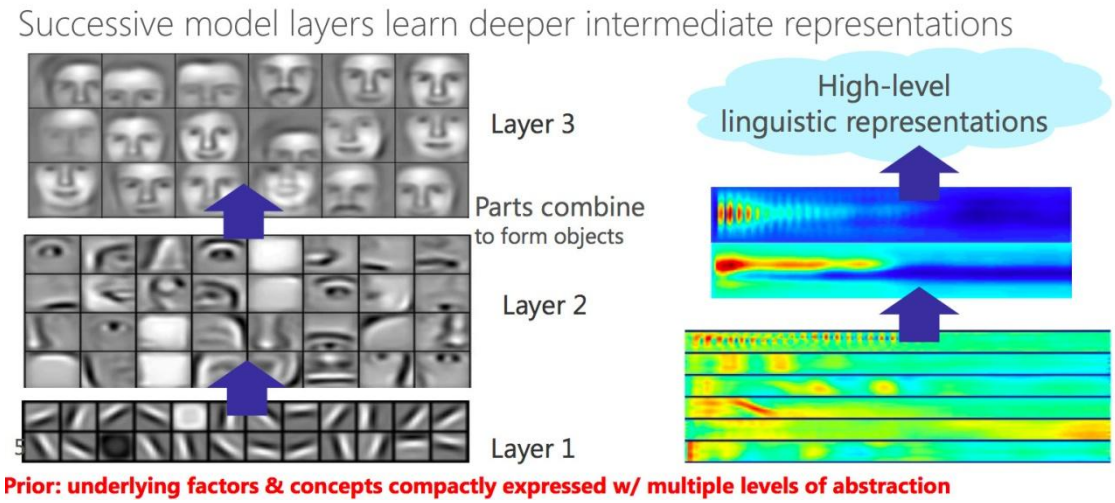

Figura 2-3 Incremento en la capacidad de reconocimiento con el aumento de capas intermedias [10].

Cuanto mayor sea el número de capas de una red neuronal, mayor será la capacidad de dicha red de discernir diferentes características procedentes de los datos. Esto se conoce como jerarquía de características (*feature hierarchy*), y es una jerarquía de creciente

complejidad y abstracción. Permite hacer redes para el aprendizaje profundo capaces de manejar conjuntos de datos muy grandes y de alta dimensión con miles de millones de parámetros que pasan a través de funciones no lineales. Una representación intuitiva de este concepto queda representada en la Figura 2-3 donde se muestra que la capacidad de reconocer patrones más complejos aumenta al incrementar el número de capas. A la izquierda se presenta un caso con imágenes de rostros y a la derecha uno con señales de voz.

2.3 REDES NEURONALES CONVOLUCIONALES (CNN)

Las redes neuronales convolucionales (ConvNets o CNN) son una categoría de redes neuronales que han demostrado ser muy efectivas en áreas como el reconocimiento y la clasificación de imágenes. Las ConvNets han tenido éxito en la identificación de rostros, objetos y señales de tráfico, además de potenciar la visión en robots y automóviles autónomos [11].

Las ConvNets, por lo tanto, son en la actualidad una herramienta importante para la mayoría de las aplicaciones de aprendizaje automático. Sin embargo, comprender las ConvNets y aprender a usarlas por primera vez a veces puede ser una experiencia intimidante. El objetivo principal de este punto es desarrollar una comprensión de cómo funcionan las redes neuronales convolucionales en las imágenes.

2.3.1 La arquitectura LeNet (años 90)

LeNet fue una de las primeras redes neuronales convolucionales que ayudó a impulsar el campo del aprendizaje profundo. Este trabajo pionero de Yann LeCun fue nombrado LeNet5 [12] después de muchas iteraciones exitosas previas desde el año 1988 [13]. En ese momento, la arquitectura LeNet se usaba principalmente para tareas de reconocimiento de caracteres, como leer códigos postales, dígitos, etc.

A continuación, desarrollaremos cómo la arquitectura LeNet aprende a reconocer imágenes. Se han propuesto varias arquitecturas nuevas en los últimos años que son mejoras sobre LeNet, pero todas utilizan los conceptos principales de LeNet y son relativamente más fáciles de entender si tiene una comprensión clara de la primera.

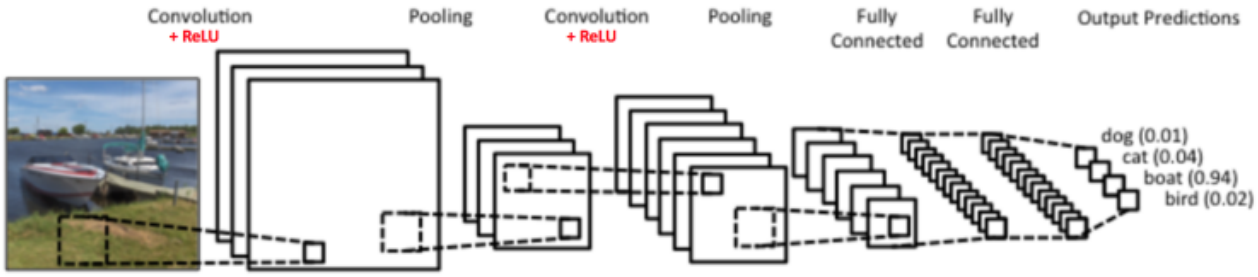

Figura 2-4 Esquema de una CNN simple

La red neuronal convolucional en la Figura 2-4 es similar en arquitectura a la LeNet original y clasifica una imagen de entrada en cuatro categorías: perro, gato, barco o pájaro (la LeNet original se usó principalmente para tareas de reconocimiento de caracteres). Como se desprende de la figura anterior, al recibir una imagen de barco como entrada, la red asigna correctamente la probabilidad más alta de barco (0,94) entre las cuatro categorías. La suma de todas las probabilidades en la capa de salida debe ser uno.

Hay cuatro operaciones principales en la ConvNet que se muestra en la Figura 2-4:

- Convolución
- No linealidad (ReLU)
- *Pooling* o Submuestreo
- Clasificación (capa completamente conectada)

Estas operaciones son los componentes básicos de cada red neuronal convolucional, por lo que comprender cómo funcionan es un paso importante para desarrollar una comprensión sólida de las CNNs. Trataremos de comprender la intuición detrás de cada una de estas operaciones a continuación.

2.3.1.1 Imágenes y matrices

Esencialmente, cada imagen se puede representar como una matriz de valores de píxeles como se muestra en la siguiente Figura 2-5.

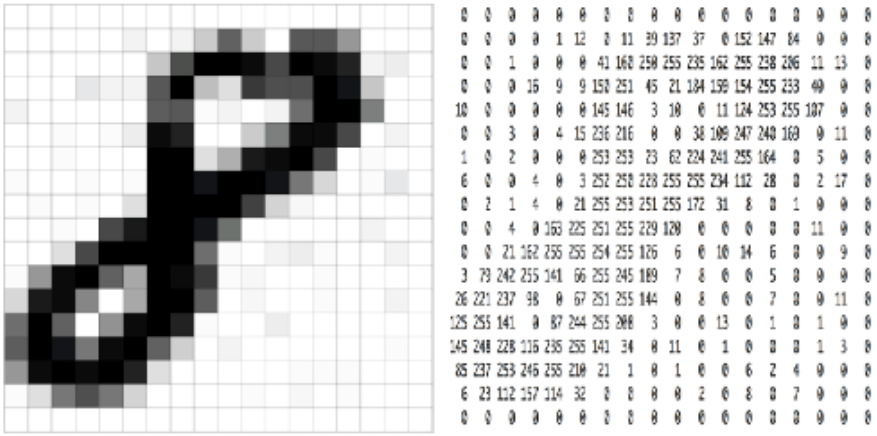

Figura 2-5 Equivalencia entre imagen y matriz de valores de píxeles.

Canal es un término convencional utilizado para referirse a una determinada componente de una imagen. Una imagen de una cámara digital estándar tendrá tres canales: rojo, verde y azul. Podemos imaginarlos como tres matrices 2D apiladas entre sí (una por canal), cada una con valores de píxeles en el rango de 0 a 255. Una imagen en escala de grises, por otro lado, tiene solo un canal. Para el propósito de esta exposición, solo consideraremos imágenes en escala de grises, por lo que tendremos una única matriz 2D que representa una imagen. El valor de cada píxel en la matriz variará entre 0 y 255. Normalmente un valor de 0 representa el color negro y 255 el blanco (en la Figura 2-5 se ha usado el criterio contrario para que visualmente la representación numérica se asemeje a la imagen original).

2.3.1.2 El paso de convolución

Las ConvNets derivan su nombre del operador de "convolución". El propósito principal de la convolución en el caso de una ConvNet es extraer características de la imagen de entrada. La convolución preserva la relación espacial entre píxeles al aprender las características de la imagen utilizando pequeños cuadrados de datos de entrada. Aquí no entraremos en detalles matemáticos de la convolución, pero trataremos de entender cómo funciona sobre las imágenes.

Como discutimos anteriormente, cada imagen se puede considerar como una matriz de valores de píxeles. Consideremos una imagen de 5 x 5 y una matriz 3 x 3 como se muestra en la Figura 2-6:

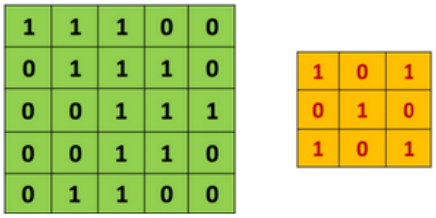

Figura 2-6 Imagen 5x5 y matriz 3x3.

Luego, la Convolución de la imagen 5 x 5 y la matriz 3 x 3 se puede calcular como se muestra en la Figura 2-7 a continuación:

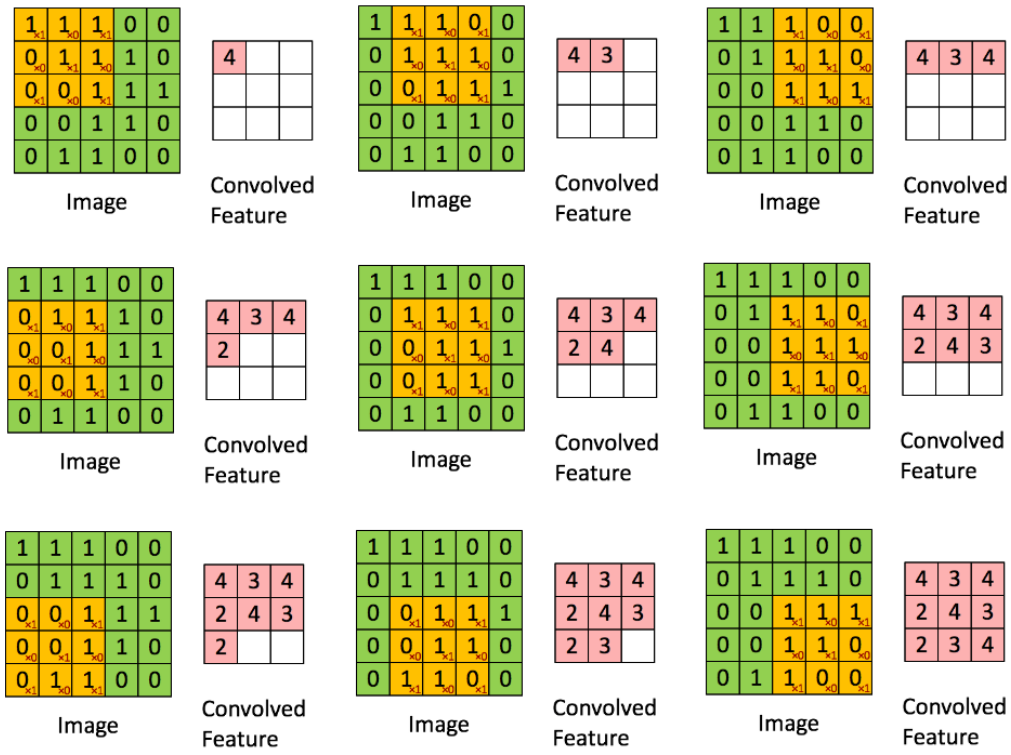

Figura 2-7 Operación de Convolución. La matriz de salida se denomina "Convolved Feature" o "Feature Map".

El proceso es el que se describe a continuación. Deslizamos la matriz naranja sobre nuestra imagen original (verde) de píxel en píxel y para cada posición, calculamos la multiplicación elemento a elemento (entre las dos matrices) y sumamos todos los elementos de la multiplicación para obtener el entero final que determina un solo elemento de la matriz de salida (rosa). Tengamos en cuenta que la matriz 3 × 3 "ve" solo una parte de la imagen de entrada en cada paso.

En la terminología de CNN, la matriz 3 × 3 se llama 'filtro', '*kernel*' o 'detector de características' y la matriz formada al deslizar el filtro sobre la imagen y calcular el producto escalar se llama *Convoluted Feature* (Característica Convolucionada) o *Feature Map* (Mapa de Características). Es importante tener en cuenta que los filtros actúan como detectores de características de la imagen de entrada original.

Es evidente a partir de observar la Figura 2-7 que diferentes valores de la matriz del kernel producirán diferentes mapas de características para la misma imagen de entrada. En la Figura 2-8, podemos ver los efectos de convolución de una imagen con diferentes filtros. Como se muestra, podemos realizar operaciones como detección de bordes, enfocar, desenfocar… simplemente cambiando los valores numéricos de nuestra matriz de filtros antes de la operación de convolución [14]; esto significa que diferentes filtros pueden detectar diferentes características de una imagen, por ejemplo, bordes, curvas, etc.

En el ámbito del *deep learning*, una CNN aprende los valores de estos filtros por sí sola durante el proceso de entrenamiento (aunque aún necesitamos especificar parámetros como el número de filtros, el tamaño del filtro, la arquitectura de la red, etc. antes del proceso de entrenamiento). Cuantos más filtros tengamos, más características de imagen se extraerán y mejor será nuestra red para reconocer patrones en imágenes.

El tamaño del Mapa de características se controla mediante tres parámetros [15] que debemos decidir antes de realizar el paso de convolución:

- *Depth*: la profundidad corresponde al número de filtros que usamos para la operación de convolución. En la red que se muestra en la Figura 2-4, estamos realizando una convolución de la imagen original del barco usando tres filtros distintos, produciendo así tres mapas de características diferentes. Se pueden imaginar estos tres mapas de características como matrices 2D apiladas, por lo tanto, la "profundidad" del mapa de características sería tres.
- *Stride*: *Stride* o zancada, es el número de píxeles por el cual deslizamos nuestra matriz de filtro sobre la matriz de entrada. Cuando el *stride* es 1, movemos los filtros un píxel a la vez. Si se aumenta este parámetro, se incrementa el número de píxeles entre cada aplicación del filtro. Tener un paso más grande producirá mapas de características más pequeños.
- *Zero Padding*: a veces, es conveniente rellenar la matriz de entrada con ceros alrededor del borde, para que podamos aplicar el filtro a los elementos que bordean nuestra matriz de imagen de entrada. Una buena característica del *zero-padding* es que nos permite controlar el tamaño de los mapas de características.

Agregar *zero-padding* también se llama *wide convolution*, y no usar relleno sería una *narrow convolution* [16]. (Estas dos opciones se denominan, *same padding* y *valid padding* a la hora de programar)

Operation	Filter	Convolved Image
Identity	$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$	
Edge detection	$\begin{bmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{bmatrix}$	
	$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$	
	$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$	
Sharpen	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	
Box blur (normalized)	$\frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$	
Gaussian blur (approximation)	$\frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$	

Figura 2-8 Algunos ejemplos de Kernel para filtrado de imágenes [14].

2.3.1.3 Introducción a la no linealidad (ReLU)

Como se puede ver en la Figura 2-4 después de cada operación de convolución se utiliza una operación adicional llamada ReLU cuya expresión se puede ver en (1). ReLU es el acrónimo de *Rectified Linear Unit* y es una operación no lineal. Su salida se puede observar en la Figura 2-9.

$$R(z) = max\{0, z\} \tag{1}$$

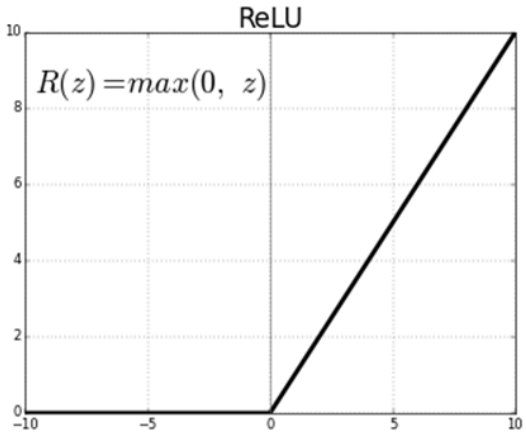

Figura 2-9. La operación no lineal ReLU.

ReLU es una operación basada en elementos (aplicada por píxel) y reemplaza todos los valores de píxeles negativos en el mapa de características por cero. El propósito de ReLU es introducir la no linealidad en nuestra ConvNet, ya que la mayoría de los datos del mundo real que deseamos que aprenda nuestra ConvNet serían no lineales (La convolución es una operación lineal - multiplicación y suma de elementos de matrices -, por lo que incluimos la no linealidad mediante la introducción de una función no lineal como ReLU). El mapa de características de salida aquí también se conoce como el mapa de características "Rectificado" (Figura 2-10).

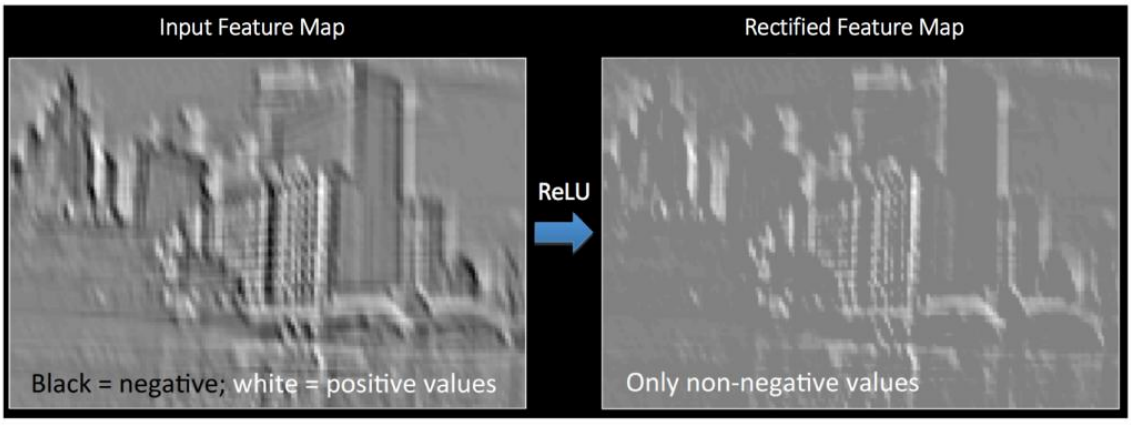

Figura 2-10 Resultado de aplicar ReLU a la convolución.

También se pueden usar otras funciones no lineales como tangentes hiperbólicas o sigmoides, pero ReLU funciona mejor en la mayoría de las situaciones, debido a su simplicidad.

2.3.1.4 El paso de agrupación o Pooling

La agrupación espacial (*spatial pooling*) (también llamada submuestreo o disminución de resolución) reduce la dimensionalidad de cada mapa de características, pero retiene la información más importante. La agrupación espacial puede ser de diferentes tipos: Máx., Promedio, Suma, etc.

En el caso de *Max Pooling*, definimos una vecindad espacial (por ejemplo, una ventana de 2 × 2) y tomamos el elemento más grande del mapa de entidades rectificado dentro de esa ventana. En lugar de tomar el elemento más grande, también podríamos tomar el promedio (Agrupación promedio) o la suma de todos los elementos en esa ventana. En la práctica, se ha demostrado que *Max Pooling* funciona mejor.

La Figura 2-11 muestra un ejemplo de la operación de *Max Pooling* en un mapa de características rectificadas (obtenido después de la operación de convolución + ReLU) mediante el uso de una ventana 2 × 2.

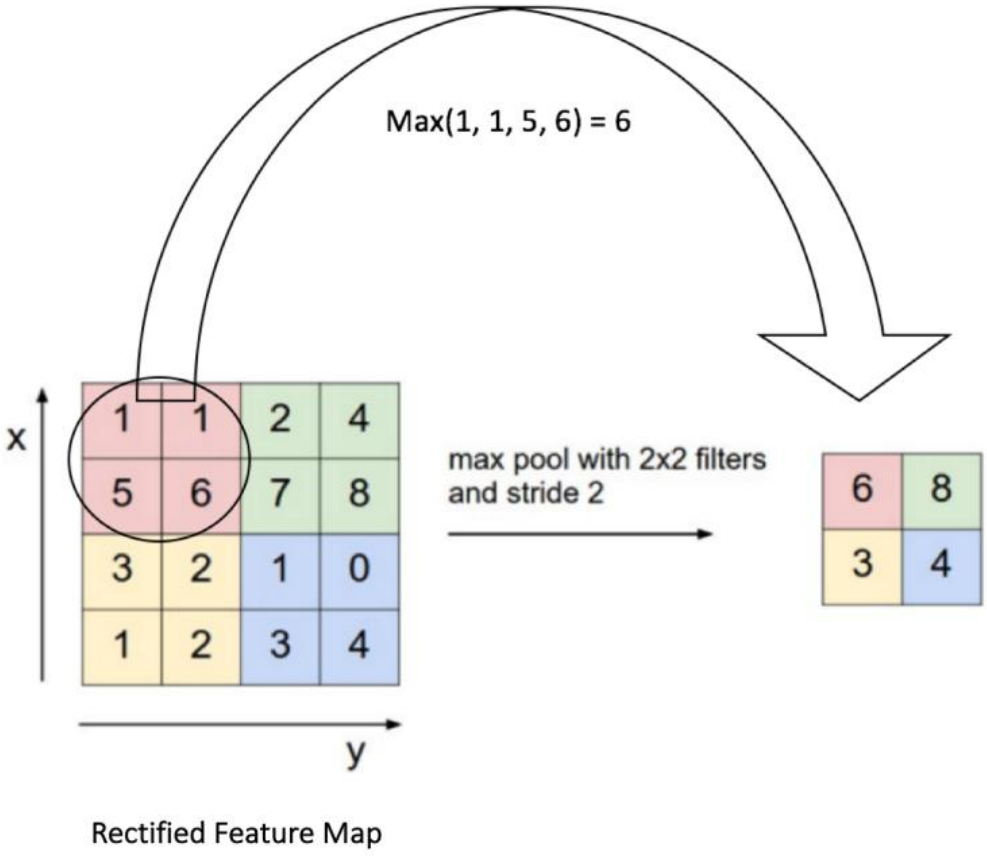

Figura 2-11 Proceso de Max Pooling con ventana de 2x2.

Deslizamos nuestra ventana de 2 x 2 con un '*stride*' de 2 celdas y tomamos el valor máximo en cada región. Como se muestra en la figura anterior, esto reduce la

dimensionalidad de nuestro mapa de características. Esta operación de agrupación se aplica por separado a cada mapa de características.

El objetivo del Pooling es reducir progresivamente el tamaño espacial de la representación de entrada [15]. En particular:

- Hace que las representaciones de entrada a las siguientes capas sean más pequeñas y manejables
- Reduce el número de parámetros y cálculos en la red, por lo tanto, controla el *over-fitting*, que sucede cuando el modelo se especializa demasiado bien al conjunto de entrenamiento y produce que se vuelva difícil para el modelo generalizar a nuevos ejemplos que no estaban en el conjunto de entrenamiento. O dicho de otro modo, el modelo reconoce imágenes específicas en su conjunto de entrenamiento en lugar de patrones generales.
- Hace que la red sea invariable a pequeñas transformaciones, distorsiones y traslaciones en la imagen de entrada (una pequeña distorsión en la entrada no cambiará la salida del *Pooling*, ya que tomamos el valor máximo / promedio en las proximidades).
- Nos ayuda a llegar a una representación casi invariable de nuestra imagen (el término exacto es "equivalente"). Esto es muy poderoso ya que podemos detectar objetos en una imagen sin importar dónde se encuentren.

Hasta ahora se ha expuesto cómo funcionan Convolución, ReLU y *Pooling*. Es importante comprender que estas capas son los componentes básicos de cualquier CNN. Como se mostraba en el ejemplo de la Figura 2-4, tenemos dos conjuntos de capas de Convolución, ReLU y *Pooling:* la segunda capa de Convolución realiza la convolución en la salida de la primera capa de *Pooling* usando seis filtros para producir un total de seis mapas de características. ReLU se aplica individualmente en cada uno de estos seis mapas de características. Luego realizamos la operación de *Max Pooling* por separado en cada uno de los seis mapas de características rectificados.

Juntas, estas capas extraen las características útiles de las imágenes, introducen la no linealidad en nuestra red y reducen la dimensión de las características. La salida de la segunda capa de agrupación actúa como una entrada a la capa totalmente conectada (*fully connected layer*), que discutiremos a continuación.

2.3.1.5 Capa totalmente conectada

La capa totalmente conectada es un Perceptrón multicapa tradicional que usa una función de activación *Softmax* (2) en la capa de salida, la cual transforma un vector K-dimensional de valores reales en otro vector K-dimensional con componentes en el rango [0,1] cuya suma es 1, El término "Totalmente conectado" implica que cada neurona en la capa anterior está conectada a cada neurona en la capa siguiente.

$$\sigma: \mathbb{R}^K \longrightarrow [0,1]^K$$
$$\sigma(\boldsymbol{z})_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}}, para\ j = 1, \dots, K \quad (2)$$

El resultado de las capas convolucionales y de agrupación representan características de alto nivel de la imagen de entrada. El propósito de la capa totalmente conectada es utilizar estas características para clasificar la imagen de entrada en varias clases según el conjunto de datos de entrenamiento. Por ejemplo, la tarea de clasificación de imágenes que nos proponemos realizar tiene cuatro salidas posibles, como se muestra en la parte derecha de la Figura 2-4 (hay que tener en cuenta que no se muestran las conexiones entre los nodos en la capa completamente conectada).

Además de la clasificación, agregar una capa completamente conectada también es una forma (generalmente) barata de aprender combinaciones no lineales de estas características. La mayoría de las características de las capas convolucionales y de agrupación pueden ser buenas para la tarea de clasificación, pero las combinaciones de esas características podrían ser incluso mejores.

La suma de las probabilidades de salida de la capa totalmente conectada es 1. Esto se garantiza mediante el uso de Softmax como la función de activación en la capa de salida de la capa completamente conectada.

2.3.1.6 Entrenamiento usando Backpropagation

Como se discutió anteriormente, las capas convolución y agrupación actúan como extractores de características de la imagen de entrada, mientras que la capa completamente conectada actúa como un clasificador.

Tengamos en cuenta que en la Figura 2-12 a continuación, dado que la imagen de entrada es un barco, la probabilidad objetivo es 1 para la clase Barco y 0 para otras tres clases, es decir:

- Imagen de entrada = Barco
- Vector de destino = [0, 0, 1, 0]

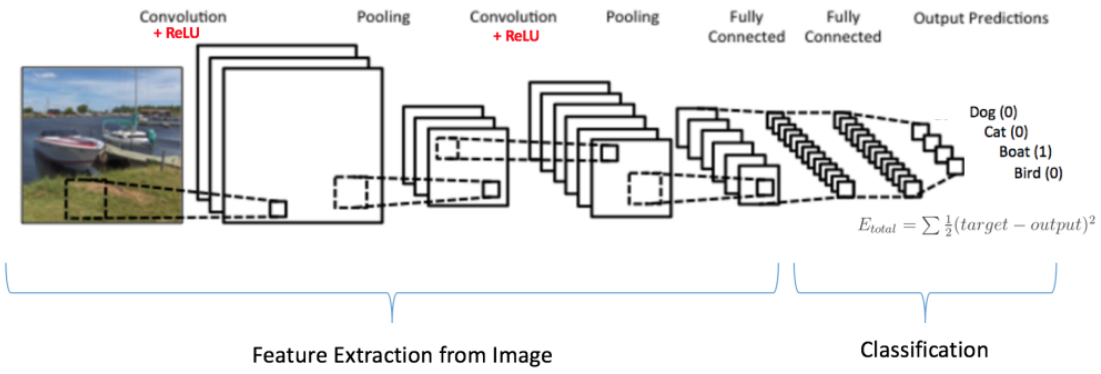

Figura 2-12 Entrenando una red neuronal convolucional.

El proceso general de entrenamiento de la red convolucional puede resumirse de la siguiente manera:

- Paso 1: inicializamos todos los filtros y parámetros / pesos con valores aleatorios
- Paso 2: la red toma una imagen de entrenamiento como entrada, pasa por el paso de propagación hacia adelante (convolución, ReLU y operaciones de agrupación junto con la propagación hacia adelante en la capa totalmente conectada) y encuentra las probabilidades de salida para cada clase. Digamos que las probabilidades de salida para la imagen del barco de arriba son [0.2, 0.4, 0.1, 0.3]. Como los pesos se asignan aleatoriamente para el primer ejemplo de entrenamiento, las probabilidades de salida también son aleatorias.
- Paso 3: Calculamos el error total en la capa de salida (suma de las 4 clases). Error total = Σ ½ (probabilidad objetivo - probabilidad de salida) ^{2}
- Paso 4: Usamos la propagación hacia atrás (*backpropagation*) para calcular los gradientes del error con respecto a todos los pesos en la red y empleamos el *gradient descent*, que es un algoritmo de optimización que se utiliza para minimizar el valor de la función de pérdida mediante el movimiento iterativo en la dirección del descenso más pronunciado según el negativo del gradiente, ajustando los parámetros para que cumplan con dicho criterio. Los parámetros se refieren a coeficientes en regresión lineal y pesos en redes neuronales. para actualizar todos los valores/pesos del filtro y los valores de los parámetros para intentar minimizar el error de salida. Los pesos se ajustan en proporción a su contribución al error total. Cuando la misma imagen se introduce nuevamente, las probabilidades de salida ahora pueden ser [0.1, 0.1, 0.7, 0.1], que está más cerca del vector objetivo [0, 0, 1, 0]. Esto significa que la red ha aprendido a clasificar esta imagen en particular correctamente ajustando sus pesos / filtros de modo que se reduzca el error de salida. Parámetros como el número de filtros, los tamaños de filtro, la arquitectura de la red, etc., se han corregido antes del Paso 1 y no cambian durante el proceso de entrenamiento; solo se actualizan los valores de la matriz del filtro y los pesos de las conexiones.
- Paso 5: repetimos los pasos 2-4 con todas las imágenes en el conjunto de entrenamiento.

Los pasos anteriores entrenan la ConvNet; esto significa esencialmente que todos los pesos y parámetros de ConvNet se han optimizado para clasificar correctamente las imágenes del conjunto de entrenamiento. Hay que destacar no obstante que los pasos anteriores se han simplificado para que la comprensión del proceso de entrenamiento sea más intuitiva. Para profundizar en la formulación matemática y en la complejidad se pueden consultar [15] y [17].

Cuando se introduce una nueva imagen (no vista con anterioridad) en una ConvNet, la red pasaría por el paso de propagación hacia adelante y generaría una probabilidad para cada clase (para una nueva imagen, las probabilidades de salida se calculan utilizando

los pesos que se han optimizado para clasificar correctamente todos los ejemplos de entrenamiento anteriores). Si nuestro conjunto de entrenamiento es lo suficientemente grande, la red se generalizará a las nuevas imágenes y las clasificará en categorías correctas.

En el ejemplo anterior, usamos dos conjuntos de capas alternantes de convolución y agrupación. Sin embargo, hay que tener en cuenta que estas operaciones se pueden repetir cualquier cantidad de veces en un solo ConvNet. De hecho, algunas de las ConvNets de mejor rendimiento de la actualidad tienen decenas de capas de convolución y agrupación. Además, no es necesario tener una capa de agrupación después de cada capa convolucional. Como se puede ver en la Figura 2-13, podemos tener múltiples operaciones de convolución + ReLU en sucesión antes de tener una operación de agrupación. Obsérvese también cómo se visualiza cada capa de ConvNet en la figura mencionada.

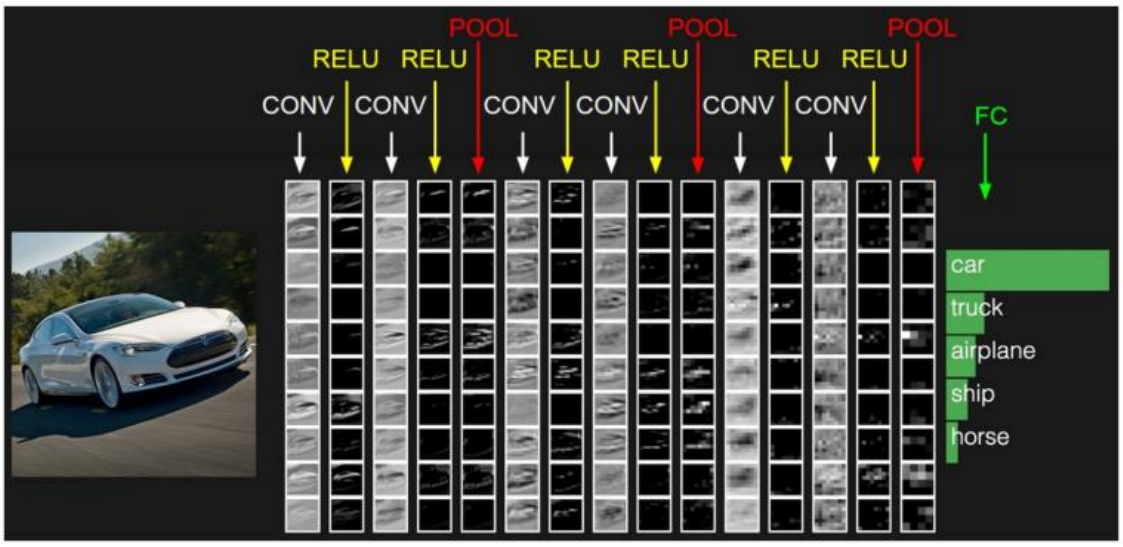

Figura 2-13 ConvNet en la que se han reducido el número de capas de pooling.

2.4 DETECCIÓN DE OBJETOS

La detección de objetos es un problema típico de la visión artificial que trata de identificar y localizar objetos de ciertas clases en la imagen. La interpretación de la localización del objeto se puede hacer de varias maneras, incluida la creación de un cuadro delimitador (bounding box) alrededor del objeto o marcar cada píxel en la imagen que contiene el objeto (denominado segmentación).

La detección de objetos se estudió incluso antes de la popularidad de las Redes Neuronales Convolucionales (CNN) en la visión artificial. Si bien las CNN son capaces de extraer automáticamente características más complejas y mejores, echar un vistazo a los métodos convencionales puede, en el peor de los casos, ser un pequeño desvío y, en el mejor de los casos, una inspiración.

La detección de objetos antes de *Deep learning* fue un proceso de varios pasos, comenzando con la detección de bordes y la extracción de características utilizando técnicas como SIFT (*scale-invariant feature transform*) [18], HOG (*Histogram of Oriented Gradients*) [19], etc. Estas imágenes se compararon con las plantillas de objetos existentes, generalmente a niveles de escala múltiples, para detectar y localizar objetos presentes en la imagen.

A día de hoy, en la detección de objetos se emplean CNN que implementan diferentes modelos y algoritmos para llevar a cabo la tarea de detectar objetos. A continuación hablaremos brevemente sobre los tres que se han utilizado en este trabajo.

Pero antes de ver los diferentes detectores es importante definir un par de conceptos clave necesarios para realizar la evaluación del rendimiento de los detectores que son el IoU y el mAP.

2.4.1 Intersection Over Union (IoU)

La Intersección sobre la Unión (IoU) o índice de Jaccard (3) [20], es una medida de evaluación utilizada para medir la precisión de un detector de objetos en un conjunto de datos en particular. Cualquier algoritmo que proporcione cuadros delimitadores como salida puede evaluarse usando IoU.

$$\mathcal{J}(A,B) = \frac{|A \cap B|}{|A \cup B|} \tag{3}$$

Para poder aplicar IoU en la evaluación de un detector de objetos (arbitrario) necesitamos:

- Los cuadros delimitadores del *ground-truth* (es decir, los cuadros delimitadores etiquetados del conjunto de prueba que especifican en qué parte de la imagen está nuestro objeto).

- Los cuadros delimitadores predichos de nuestro modelo.

En la Figura 2-14, se muestra un ejemplo visual de un cuadro delimitador del *ground-truth* frente a un cuadro delimitador previsto, la representación del área de Intersección y de Unión, y una calificación de una IoU pobre, buena y excelente:

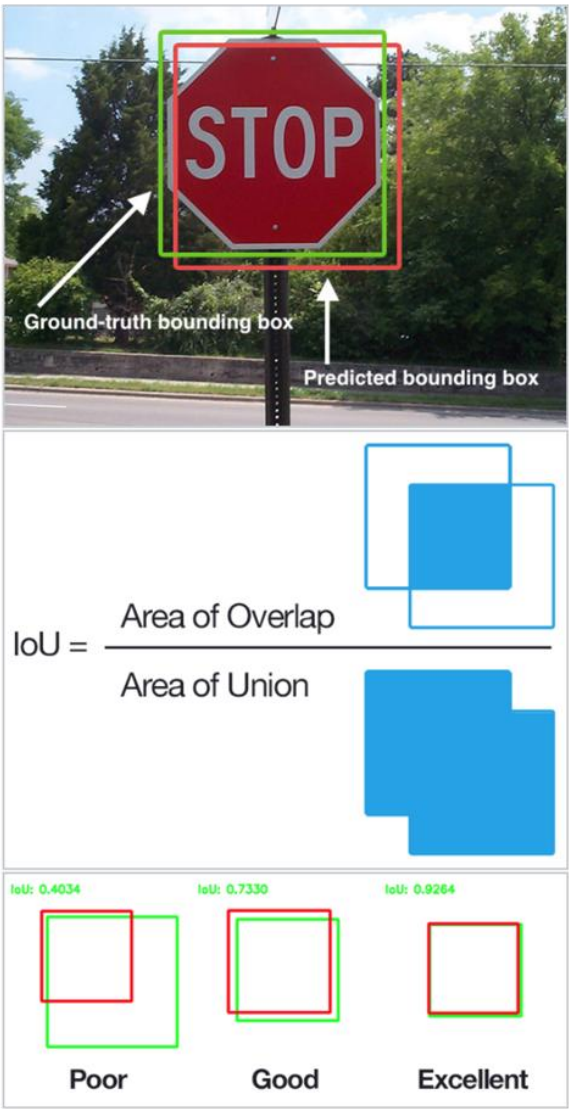

Figura 2-14 Explicación gráfica de Intersection over Union (IoU)

Al examinar la ecuación (3), se puede ver que la intersección sobre la unión es simplemente una relación. En el numerador calculamos el área de superposición entre el cuadro delimitador predicho y el cuadro delimitador del *ground-truth*. El denominador es el área de unión, es decir, el área abarcada tanto por el cuadro delimitador predicho como por el cuadro delimitador del *ground-truth*.

Dividir el área de superposición por el área de unión produce nuestra puntuación final: la intersección sobre la unión.

Cualitativamente hablando, los cuadros delimitadores predichos que se superponen en gran medida con los cuadros delimitadores del *ground-truth* tienen puntuaciones más

altas que aquellos con menos superposición. Esto hace que IoU sea una métrica excelente para evaluar detectores de objetos personalizados.

En realidad, es extremadamente improbable que las coordenadas (x, y) de nuestro cuadro delimitador previsto coincidan exactamente con las coordenadas (x, y) del cuadro delimitador de referencia. No obstante no buscamos la coincidencia exacta de las coordenadas, pero sí queremos asegurarnos de que nuestros cuadros delimitadores predichos coincidan lo más posible: y la IoU ilustra rápidamente esta idea.

2.4.2 Mean Average Precision (mAP)

La precisión y la sensibilidad o *recall*, son dos métricas de uso común para juzgar el desempeño de un modelo de clasificación dado. Para comprender mAP, primero deberíamos revisar la precisión y la sensibilidad.

La precisión (4) de una clase dada en la clasificación, también conocido por valor predicho positivo, se da como la relación de verdadero positivo (TP) y el número total de positivos pronosticados, esto es, incluyendo los falso positivos (FP).

$$Precision = \frac{TP}{TP + FP} \tag{4}$$

De manera similar, la sensibilidad o *recall* (5), también conocida como tasa de verdaderos positivos (TPR), de una clase dada en la clasificación, se define como la relación de TP y el total de positivos del *ground-truth*, es decir, la suma de los TP y los falsos negativos (FN).

$$Recall = \frac{TP}{TP + FN} \tag{5}$$

Con solo mirar las ecuaciones, podríamos constatar que para un modelo de clasificación dado, existe un compromiso entre su precisión y la sensibilidad. Si estamos utilizando una red neuronal, esta compensación se puede ajustar por el umbral softmax de la última capa del modelo.

Para que nuestra precisión sea alta, necesitaríamos disminuir nuestro número de FP, al hacerlo, disminuirá nuestra sensibilidad. Del mismo modo, al disminuir nuestro número de FN aumentaríamos la sensibilidad y disminuiríamos la precisión. Muy a menudo para los casos de recuperación de información y detección de objetos, queremos que nuestra precisión sea alta.

La precisión y la sensibilidad se usan comúnmente junto con otras métricas, como exactitud (*accuracy*), Valor-F (*F1-score*), especificidad o tasa de verdaderos negativos (*True Negative Rate, TNR*), la curva ROC (*Receiver Operating Characteristic*), elevación y ganancia.

Sin embargo, todas estas métricas no proporcionan información útil cuando se trata de determinar si un modelo está funcionando bien en tareas de recuperación de información o detección de objetos. Por ello se define el *Mean Average Precision* (mAP).

Es importante tener en cuenta que los cálculos del mAP para las tareas de detección de objetos y recuperación de información son algo diferentes. Aquí nos centraremos en el primer caso que es el que nos interesa.

Para poder calcular el mAP nos hace falta determinar primero el IoU que se usaría para determinar si un cuadro delimitador (*bounding box*) previsto (BB) es TP, FP o FN. El TN no se evalúa ya que se supone que cada imagen contiene al menos un objeto.

Normalmente, definimos una predicción como TP si el IoU es mayor de 0.5, pero se pueden usar otros valores. Los posibles escenarios se describen a continuación:

- **Verdadero Positivo** (TP), si la IoU es mayor de 0.5
- **Falso Positivo** (FP), que engloba dos situaciones: si el IoU del BB es menor de 0.5 o si obtenemos múltiples BB para un mismo objeto.
- **Falso Negativo** (FN), cuando no hay BB o cuando a pesar de tener un IoU mayor de 0.5 la clase no se corresponde con la del objeto.

Con TP, FP y FN definidos formalmente, ahora podemos calcular la precisión y la sensibilidad de nuestra detección para una clase dada en todo el conjunto de pruebas. Cada BB tendría su nivel de confianza, generalmente dado por su capa *softmax*, y se usará para clasificar la salida. Este valor se emplea para ordenar las detecciones de mayor a menor.

Con las predicciones ordenadas para una misma clase, obtenemos la precisión y la sensibilidad y las representamos en una gráfica que tendrá una forma similar a la que se muestra en la Figura 2-15, la cual representa el ejemplo que se expone en [21].

Con esto, la definición general para la precisión promedio (AP) (6) es calcular el área bajo la curva PR (*precision-recall*):

$$AP = \int_0^1 p(r)dr \tag{6}$$

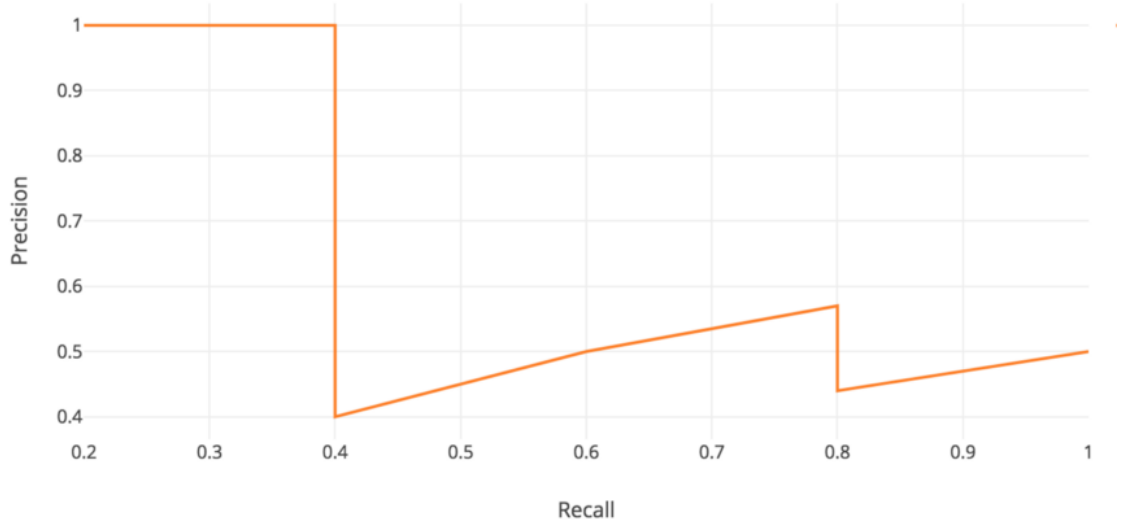

Figura 2-15. Ejemplo de Representación gráfica de la Precisión frente a la Sensibilidad para las detecciones de una misma clase.

La precisión y la sensibilidad siempre están entre 0 y 1. Por lo tanto, el AP también está comprendido entre de 0 y 1. Antes de calcular el AP para la detección de objetos, a menudo se suaviza primero el patrón de zigzag, sin más que asignando el valor máximo de precisión al intervalo para hacerlo constante (Figura 2-16).

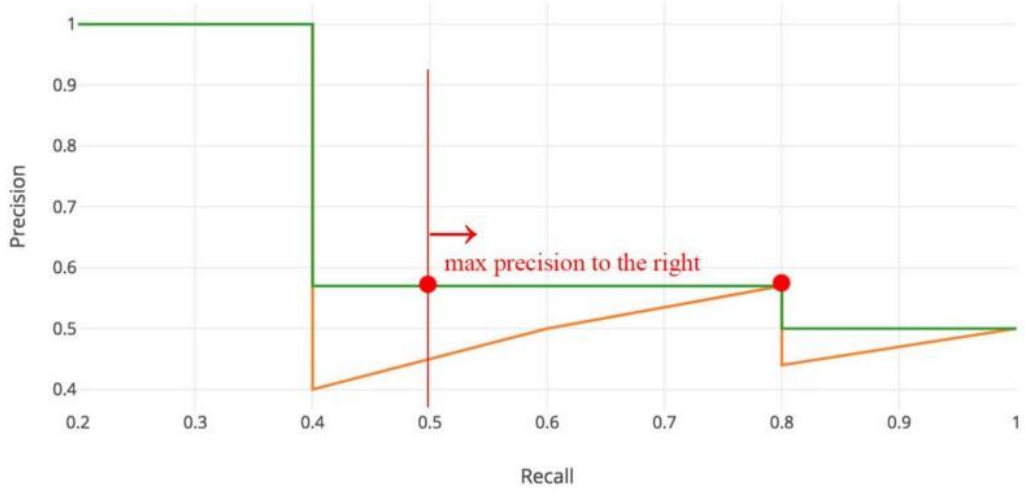

Figura 2-16. Ejemplo de Interpolación de la curva PR para eliminar las rampas.

Con todo lo anterior, ya podemos calcular el mAP para la detección de objetos sin más que promediar el AP calculado para todas las clases.

2.4.3 Faster RCNN

Faster R-CNN se publicó originalmente en el congreso "NIPS 2015" [22]. Todo comenzó con "*Rich feature hierarchies for accurate object detection and semantic segmentation*" (R-CNN) en 2014 [23], que utilizó un algoritmo llamado búsqueda selectiva para proponer posibles regiones de interés y una CNN estándar para clasificarlas y ajustarlas. Rápidamente evolucionó a Fast R-CNN [24], publicado a principios de 2015, donde una técnica llamada "*Region of Interest Pooling*" permitió compartir cálculos costosos e hizo que el modelo fuera mucho más rápido. Finalmente llegó Faster R-CNN [22], donde se propuso el primer modelo de esta arquitectura.

2.4.4 SSD

El documento sobre SSD: *Single Shot MultiBox Detector* [25] (por C. Szegedy et al.) se publicó a finales de noviembre de 2016 y alcanzó nuevos récords en términos de rendimiento y precisión para las tareas de detección de objetos, con una puntuación de más del 74% de mAP (*mean Average Precision*) [21] a 59 cuadros por segundo en conjuntos de datos estándar como PascalVOC [26] y COCO [27]. Para comprender mejor SSD, expliquemos de dónde proviene el nombre de esta arquitectura:

- *Single Shot*: esto significa que las tareas de localización y clasificación de objetos se realizan en un solo paso directo de la red.
- *MultiBox*: este es el nombre de una técnica para la regresión del cuadro delimitador desarrollada por C. Szegedy et al en [25].
- *Detector*: la red es un detector de objetos que también clasifica los objetos detectados.

2.4.5 YOLO

YOLO (*You Only Look Once*), es una red para la detección de objetos. Los métodos anteriores para esto, como R-CNN y sus variaciones, utilizaron un *pipeline* para realizar esta tarea en varios pasos. Esto puede ser lento de ejecutar y también difícil de optimizar, porque cada componente individual debe ser entrenada por separado. YOLO, lo hace todo con una sola red neuronal. De la publicación [28]: "Replanteamos la detección de objetos como un problema de regresión único, directamente desde los píxeles de la imagen hasta las coordenadas del cuadro delimitador y las probabilidades de clase".

Entonces, en esencia, toma una imagen como entrada, la pasa a través de una red neuronal que se parece a una CNN normal, y obtiene un vector de cuadros delimitadores y predicciones de clase en la salida.

Se abordarán los detalles del funcionamiento y arquitecturas de estos detectores en el Capítulo 3.

2.5 TRACKING

El seguimiento de objetos o *tracking* se basa en estimar el estado del objeto de interés presente en la escena a partir de información previa y es una de las principales tareas de la visión artificial. El seguimiento de objetos se utiliza en la gran mayoría de las aplicaciones, tales como: videovigilancia, seguimiento de automóviles (estimación de distancia), detección y seguimiento de personas, etc. Los rastreadores de objetos generalmente necesitan algunos pasos de inicialización, como la ubicación inicial del objeto, que puede proporcionarse manualmente o automáticamente mediante el uso de un detector de objetos. Existen varios problemas importantes relacionados con el seguimiento:

- oclusión
- seguimiento de múltiples objetos
- escala, iluminación y cambio de apariencia
- movimientos difíciles y rápidos

Muchos de los primeros acercamientos a la resolución del problema del seguimiento de objetos tenían que resolver la detección de objetos y el rastreo al mismo tiempo.

Debido al progreso reciente en la detección de objetos, el *tracking-by-detection* se ha convertido en el paradigma líder en el seguimiento de múltiples objetos. Dentro de este paradigma, las trayectorias de los objetos generalmente se encuentran en un problema de optimización global que procesa lotes completos de video a la vez. Por ejemplo, las formulaciones de flujo de red [29]–[31] y los modelos gráficos probabilísticos [32]–[34] se han convertido en marcos populares de este tipo. Sin embargo, debido al procesamiento por lotes, estos métodos no son aplicables en escenarios en línea donde una identidad objetivo debe estar disponible en cada paso. Algunos métodos más tradicionales son el Filtro de Kalman [35], el "*Multiple Hypothesis Tracking*" (MHT) [36] y el "*Joint Probabilistic Data Association Filter*" (JPDAF) [37]. Estos métodos realizan una asociación de datos cuadro a cuadro. En el Filtrado de Kalman, se procede en dos pasos, el de predicción, donde se estiman los valores de las variables actuales, y el de actualización, donde las estimaciones se actualizan en base a las observaciones que se van recibiendo. En el JPDAF, se genera una hipótesis de un solo estado ponderando las mediciones individuales por sus probabilidades de asociación. En MHT, se rastrean todas las hipótesis posibles, pero los esquemas de descarte deben aplicarse para la trazabilidad computacional. Estos dos últimos métodos han sido revisados recientemente en un escenario de *tracking-by-detection* [38], [39] y han mostrado resultados prometedores. Sin embargo, el rendimiento de estos métodos viene con una mayor complejidad computacional y de implementación.

Veamos a continuación una breve explicación de algunos algoritmos que se emplean para el rastreo:

2.5.1 SORT

El "*Simple online and realtime tracking*" (SORT) [40] es un marco mucho más simple que realiza el filtrado de Kalman en el espacio de la imagen y la asociación de datos cuadro a cuadro utilizando el método húngaro con una métrica de asociación que mide la superposición del cuadro delimitador. Este enfoque simple logra un rendimiento favorable a altas velocidades de cuadro. En el conjunto de datos del *Multi Object Tracking Challenge* [41], SORT con un detector de personas basado en Faster R-CNN consiguió, en promedio, una puntuación más alta que MHT en detecciones estándar. Esto no solo subraya la influencia del rendimiento del detector de objetos en los resultados generales de seguimiento, sino que también es un aspecto importante a tener en cuenta a la hora de tomar decisiones de diseño.

2.5.2 ROLO

Recurrent YOLO (*ROLO*) [42] es un método de seguimiento de un solo objeto que combina detección de objetos y redes neuronales recurrentes. ROLO es una combinación de *YOLO* y *LSTM* (*Long-Short Term Memory*) [43] . El módulo de detección de objetos utiliza *YOLO* para recopilar características visuales, junto con anteriores inferencias de ubicación. En cada *frame*, el *LSTM* recibe un vector de características de entrada de longitud 4096 y devuelve la ubicación del objeto rastreado.

En la siguiente Figura 2-17 se presenta la arquitectura de ROLO.

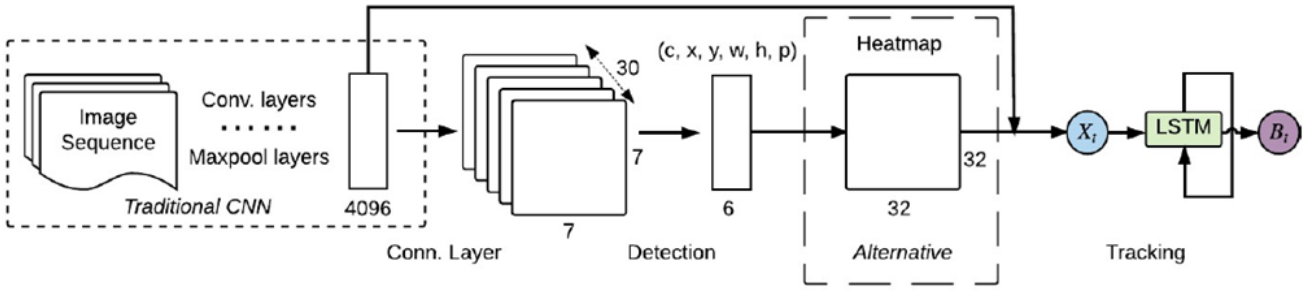

Figura 2-17. Arquitectura ROLO

2.5.3 SiamMask

Cuando se trata del seguimiento de un solo objeto, SiamMask [44] es una excelente opción. Se basa en la red neuronal siamesa [45], que son dos redes neuronales iguales que funcionan en tándem, a las que se alimenta con diferentes entradas para calcular diferencias, que aumentó en popularidad con Facenet de Google. Además de producir cuadros delimitadores rotados a 55 fotogramas por segundo, también proporciona máscaras de segmentación de objetos independientes de la clase. Para lograr esto, SiamMask debe inicializarse con un único cuadro delimitador para que pueda rastrear el objeto deseado. Sin embargo, esto también significa que el seguimiento de objetos

múltiples (*Multiple Object Tracking*, MOT) no es viable con *SiamMask*, y modificar el modelo para admitirlo nos dejaría con un detector de objetos significativamente más lento.

2.5.4 Deep SORT

Anteriormente hemos mencionado SORT como un enfoque algorítmico para el seguimiento de objetos. Deep SORT [46] es una mejora sobre SORT al reemplazar la métrica de asociación con un aprendizaje de métrica de coseno, un método para aprender un espacio de características donde la semejanza de coseno se optimiza efectivamente a través de la reparametrización del régimen *softmax*.

El marco de manejo de identificadores y filtrado de Kalman es casi idéntico al SORT original, excepto que los cuadros delimitadores se procesan utilizando una red neuronal convolucional previamente entrenada en un *dataset* para la reidentificación de un determinado objeto (en el documento original eran personas). Este método es un excelente punto de partida para la detección de múltiples objetos, ya que es fácil de implementar, ofrece una precisión sólida, y proporciona robustez frente oclusiones.

2.5.5 TrackR-CNN

TrackR-CNN [47] se introdujo como la base para el desafío *Multi Object Tracking and Segmentation* (MOTS), pero resultó ser más efectivo de lo que se esperaba. En primer lugar, el módulo de detección de objetos utiliza Mask R-CNN [48] en la parte superior de una red troncal ResNet-101. El rastreador se crea integrando convoluciones 3D que se aplican a las características de la red troncal, incorporando el contexto temporal del video. Como alternativa, también se considera el LSTM convolucional, pero este último método no produce ganancias de rendimiento en comparación con la implementación base.

TrackR-CNN también extiende Mask R-CNN [48], que en vez de cuadros delimitadores proporciona máscaras que delimitan la forma de los objetos, por un encabezado de asociación, para poder asociar detecciones a lo largo del tiempo. Esta es una capa totalmente conectada que recibe propuestas de región y genera un vector de asociación para cada propuesta. El encabezado de asociación se inspira en las redes siamesas y los vectores de inclusión utilizados en la reidentificación de personas. Se entrena usando una adaptación de secuencia de video de pérdida de triplete estricto por lotes (*batch hard triplet loss*), que es un método más eficiente que la pérdida de triplete original. Para producir el resultado final, el sistema debe decidir qué detecciones se deben mantener. La coincidencia entre las detecciones de cuadros anteriores y las propuestas actuales se realiza utilizando el algoritmo húngaro, al tiempo que solo permite pares de detecciones con vectores de asociación más pequeños que algún umbral.

2.5.6 Tracktor++

El *Multiple Object Tracking Benchmark* [41] hace que sea más fácil encontrar los avances más recientes en MOT, gracias a su tabla de clasificación pública. El CVPR (*Conference on Computer Vision and Pattern Recognition*) 2019 Tracking Challenge motivó el progreso tanto en la precisión como en la velocidad de los rastreadores. Tracktor++ [49] dominó la tabla de clasificación con un enfoque muy simple pero efectivo. Este modelo predice la posición de un objeto en el siguiente cuadro calculando la regresión del cuadro delimitador, sin necesidad de entrenar u optimizar el seguimiento de los datos. El detector de objetos para Tracktor++ es el R-CNN más rápido habitual con ResNet y FPN de 101 capas, entrenado en el conjunto de datos de detección de peatones MOT17Det.

La idea principal de Tracktor ++ es usar la rama de regresión de Faster R-CNN para el seguimiento cuadro a cuadro extrayendo características del cuadro actual y luego utilizando ubicaciones de objetos del cuadro anterior como entrada para el proceso de agrupación de ROI para revertir su ubicación en el marco actual. También utiliza algunos modelos de movimiento, como la compensación de movimiento de la cámara basada en el registro de imágenes y la reidentificación a corto plazo. El método de reidentificación almacena en caché los rastreadores desactivadas para un número fijo de cuadros, y luego compara los rastreadores recién detectados con ellos para una posible reidentificación. La distancia entre rastreadores se mide mediante una red neuronal siamesa.

2.5.7 JDE

Joint Detection and Embedding (JDE) [50] es una propuesta muy reciente similar a RetinaNet que se desvía del paradigma de dos etapas. Este detector de disparo único está diseñado para resolver un problema de aprendizaje de tareas múltiples, es decir, clasificación de anclaje, regresión de cuadro delimitador y aprendizaje integrado. JDE usa Darknet-53 como la columna vertebral para obtener mapas de características de la entrada a tres escalas. Posteriormente, los mapas de características se fusionan mediante muestreo ascendente y conexiones residuales. Finalmente, los encabezados de predicciones se adjuntan en la parte superior de los mapas de características fusionadas, que generan un mapa de predicción denso para las tres tareas que se mencionaron anteriormente.

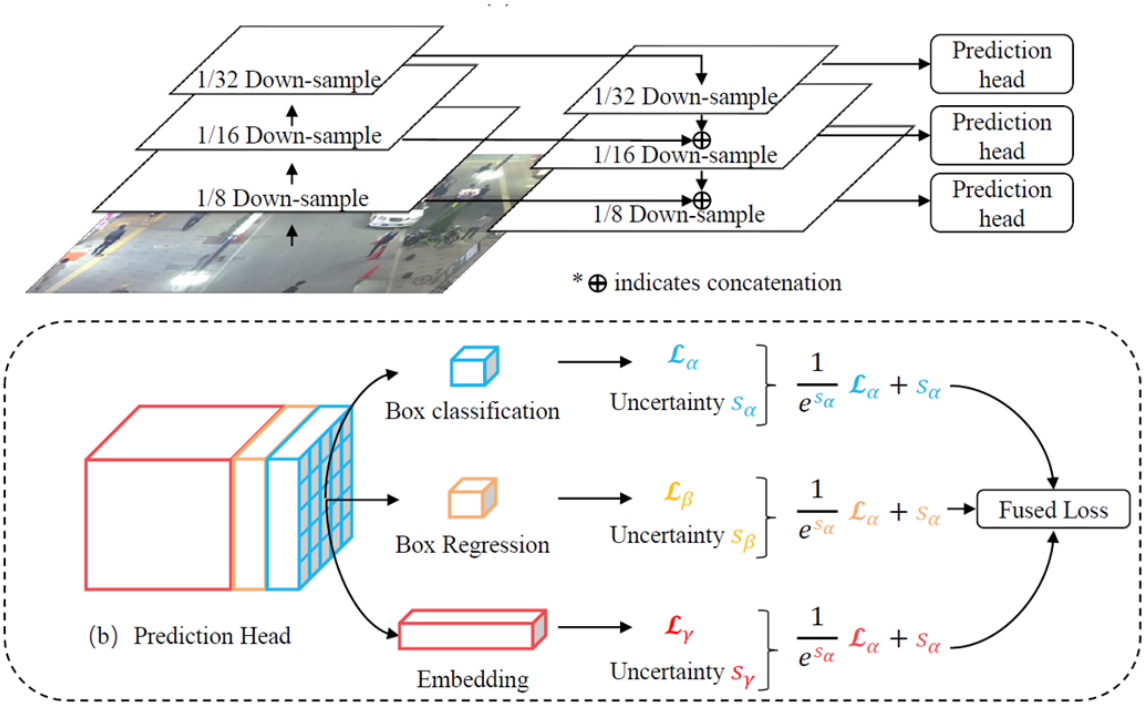

Figura 2-18. Arquitectura JDE

Para lograr el seguimiento de objetos, además de las clases y los cuadros delimitadores, el modelo JDE también genera vectores de inclusión de apariencia al procesar los *frames*. Estas inclusiones de apariencia se comparan con las inclusiones de objetos detectados previamente utilizando una matriz de afinidad. Finalmente, el algoritmo húngaro y el filtro Kalman se utilizan para suavizar las trayectorias y predecir las ubicaciones de los objetos detectados previamente en el *frame* actual. En la Figura 2-18 se puede ver la Arquitectura de JDE

Capítulo 3

METODOLOGíA

En este capítulo comentaremos todos los pasos que se han llevado a cabo para obtener el programa final. Empezando por el análisis de los *datasets* que se van a utilizar, que fue necesario para poder realizar operaciones sobre los ficheros de anotaciones, para continuar con el estudio del rendimiento de los diferentes modelos de detección de objetos, el proceso de entrenamiento de los modelos en un *dataset* diferente mediante *transfer learning* [3], y finalmente la realización del script encargado de la detección y el seguimiento de objetos en carretera.

Este proceso queda representado en la Figura 3-1.

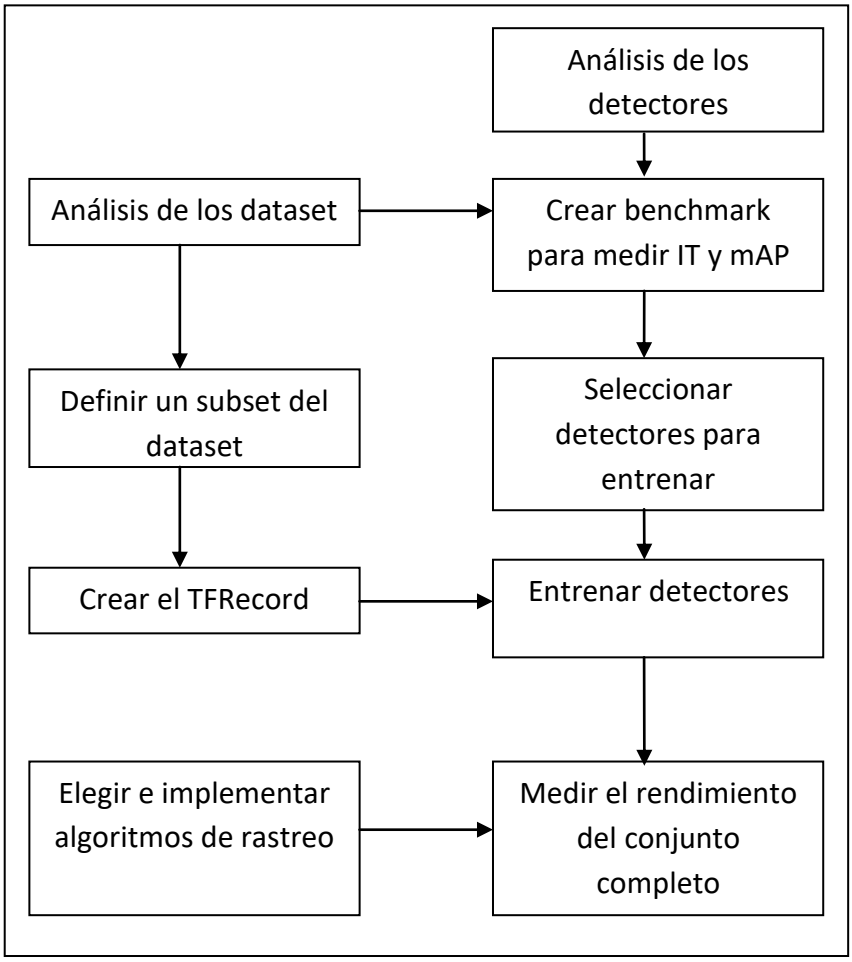

Figura 3-1. Diagrama de bloques de la metodología empleada.

3.1 DATASETS

Lo primero que se necesita es la base de datos con la que se va a trabajar, en nuestro caso se han empleado el *dataset* de COCO [27] y el *dataset* BDD100k [51]. Una vez descargados y descomprimidos comprobamos la información que contienen. Como la mayoría de *datasets* están compuestos por tres carpetas donde se guardan cada una de los set que se usarán durante el entrenamiento y evaluación de los detectores, que son el "*train set*", el "*validation set*" y el "*test set*". De manera general:

- *Train set*: El conjunto de datos que usamos para entrenar el modelo. El modelo ve y aprende de estos datos.
- *Validation set*: El conjunto de validación se usa para evaluar un modelo dado, pero esto es para una evaluación frecuente. Se usan estos datos para ajustar los hiperparámetros del modelo. Por lo tanto, el modelo ocasionalmente ve estos datos, pero nunca "Aprende" de ellos. Usamos los resultados del conjunto de validación y actualizamos los hiperparámetros de nivel superior. Por lo tanto el conjunto de validación afecta a un modelo, pero solo indirectamente. El conjunto de validación también se conoce como el conjunto de desarrollo. Esto tiene sentido ya que este conjunto de datos ayuda durante la etapa de "desarrollo" del modelo.
- *Test set*: El conjunto de datos de prueba se emplea para evaluar el modelo. Solo se usa una vez que un modelo está completamente entrenado (usando los conjuntos de validación y entrenamiento). El conjunto de prueba es generalmente lo que se usa para evaluar los modelos en las competiciones (por ejemplo, en muchas competiciones de Kaggle, el conjunto de validación se lanza inicialmente junto con el conjunto de entrenamiento y el conjunto de prueba solo se proporciona cuando la competición está a punto de cerrarse, y es el resultado del modelo en el conjunto de prueba que decide el ganador). El conjunto de prueba generalmente está bien organizado y contiene datos cuidadosamente muestreados que abarcan las diversas clases a las que se enfrentaría el modelo cuando se usa en el mundo real.

Además contienen sendas anotaciones con la información referente a cada una de las imágenes. Dependiendo del *dataset*, los ficheros de anotaciones pueden darse de forma global o individual, y en diferentes formatos de fichero. En nuestro caso ambos incluyen ficheros del tipo "JSON" [52] en el que están contenidas todas las etiquetas. Sin embargo los formatos de las anotaciones difieren de un *dataset* a otro por lo que hay que poner especial cuidado a la hora de utilizarlos.

Los detalles y la estructura de los *datasets* empleados se detallan en el Apéndice A.

3.2 ANÁLISIS DEL FUNCIONAMIENTO DE LOS DETECTORES

Habiendo expuesto las generalidades de los detectores de objetos en la sección 2.4, profundizaremos a continuación en los detalles de cada uno de ellos.

3.2.1 Faster R-CNN

3.2.1.1 Arquitectura

La arquitectura de Faster R-CNN (Figura 3-2) es compleja porque tiene varias partes móviles. El punto de partida es una imagen de entrada, de la que queremos obtener:

- una lista de cuadros delimitadores (bounding boxes).
- una etiqueta asignada a cada cuadro delimitador.
- una probabilidad para cada etiqueta y cuadro delimitador.

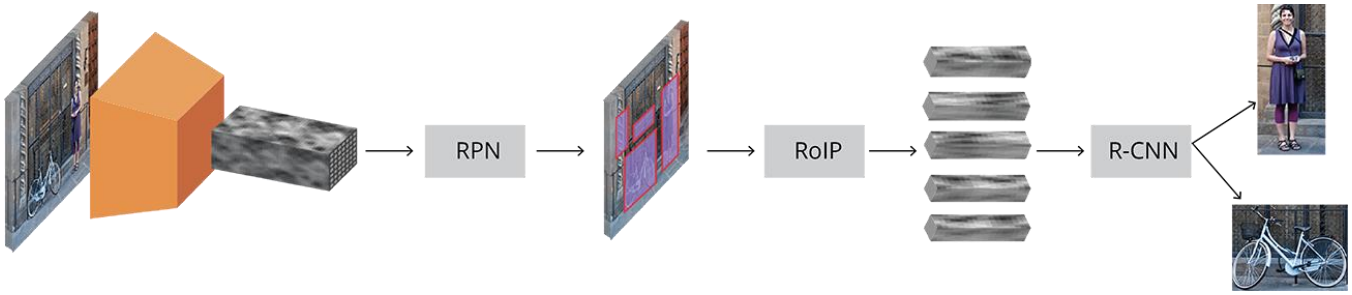

Figura 3-2 Arquitectura completa de Faster-RCNN.

Las imágenes de entrada se representan como tensores Altura x Anchura x Profundidad (matrices multidimensionales), que se pasan a través de una CNN previamente entrenada hasta una capa intermedia, terminando con un mapa de características convolucional. Usamos esto como un extractor de características para la siguiente parte.

A continuación, tenemos lo que se llama una "Region Proposal Network" (RPN, para abreviar). Usando las características que calculó la CNN, se usa para encontrar hasta un número predefinido de regiones (cuadros delimitadores), que pueden contener objetos.

Probablemente el problema más difícil con el uso de *Deep learning* (DL) para la detección de objetos es generar una lista de cuadros delimitadores de longitud variable. Al modelar redes neuronales profundas, el último bloque suele ser una salida de tensor de tamaño fijo (excepto cuando se utilizan redes neuronales recurrentes). Por ejemplo, en la clasificación de imágenes, la salida es un tensor con N componentes, siendo N el número de clases, donde cada elemento en la posición i contiene la probabilidad de que esa imagen sea de la clase i.

El problema de longitud variable se resuelve en el RPN mediante el uso de cuadros de anclaje o *anchors*: cuadros delimitadores de referencia de tamaño fijo que se colocan de manera uniforme en toda la imagen original. En lugar de tener que detectar dónde están los objetos, modelamos el problema en dos partes. Por cada ancla, preguntamos:

- ¿Este *anchor* contiene un objeto relevante?
- ¿Cómo ajustaríamos este *anchor* para que se ajuste mejor al objeto en cuestión?

Después de tener una lista de posibles objetos relevantes y sus ubicaciones en la imagen original, se convierte en un problema más sencillo de resolver. Usando las características extraídas por la CNN y los cuadros delimitadores con objetos relevantes, aplicamos la Agrupación de las Regiones de Interés (*RoI Pooling*) y extraemos esas características que corresponderían a los objetos relevantes en un nuevo tensor.

Finalmente, viene el módulo R-CNN, que usa esa información para:

- Clasificar el contenido en el cuadro delimitador (o descartarlo, utilizando "fondo" como etiqueta).
- Ajustar las coordenadas del cuadro delimitador (para que se ajuste mejor al objeto).

Obviamente, faltan algunos datos importantes, pero esa es básicamente la idea general de cómo funciona Faster R-CNN.

A continuación, repasaremos los detalles sobre la arquitectura para cada uno de los componentes.

3.2.1.2 Red base

Como mencionamos anteriormente, el primer paso es usar un CNN pre-entrenado con un banco de imágenes, para la tarea de clasificación (por ejemplo, ImageNet [53]) y usar la salida de una capa intermedia. Esto puede sonar muy simple para las personas con un conocimiento de *deep learning*, pero es importante comprender cómo y por qué funciona, así como visualizar qué forma tiene la salida de la capa intermedia.

No existe un consenso real sobre qué arquitectura de red es la mejor. El Faster R-CNN original utilizaba ZFNet [54] y VGG [55] previamente entrenados en ImageNet, pero desde entonces ha habido muchas redes diferentes con un número variable de pesos. Por ejemplo, MobileNet [56], una arquitectura de red más pequeña y eficiente optimizada para la velocidad, tiene aproximadamente 3.3M de parámetros, mientras que ResNet-152 (de 152 capas)[57], tiene alrededor de 60M. Más recientemente, nuevas arquitecturas como DenseNet [58] están mejorando los resultados al tiempo que reducen el número de parámetros.

3.2.1.2.1 VGG

Antes de hablar sobre cuál es mejor o peor, tratemos de entender cómo funciona todo utilizando la arquitectura VGG-16 básica como ejemplo la cual se representa en la Figura 3-3.

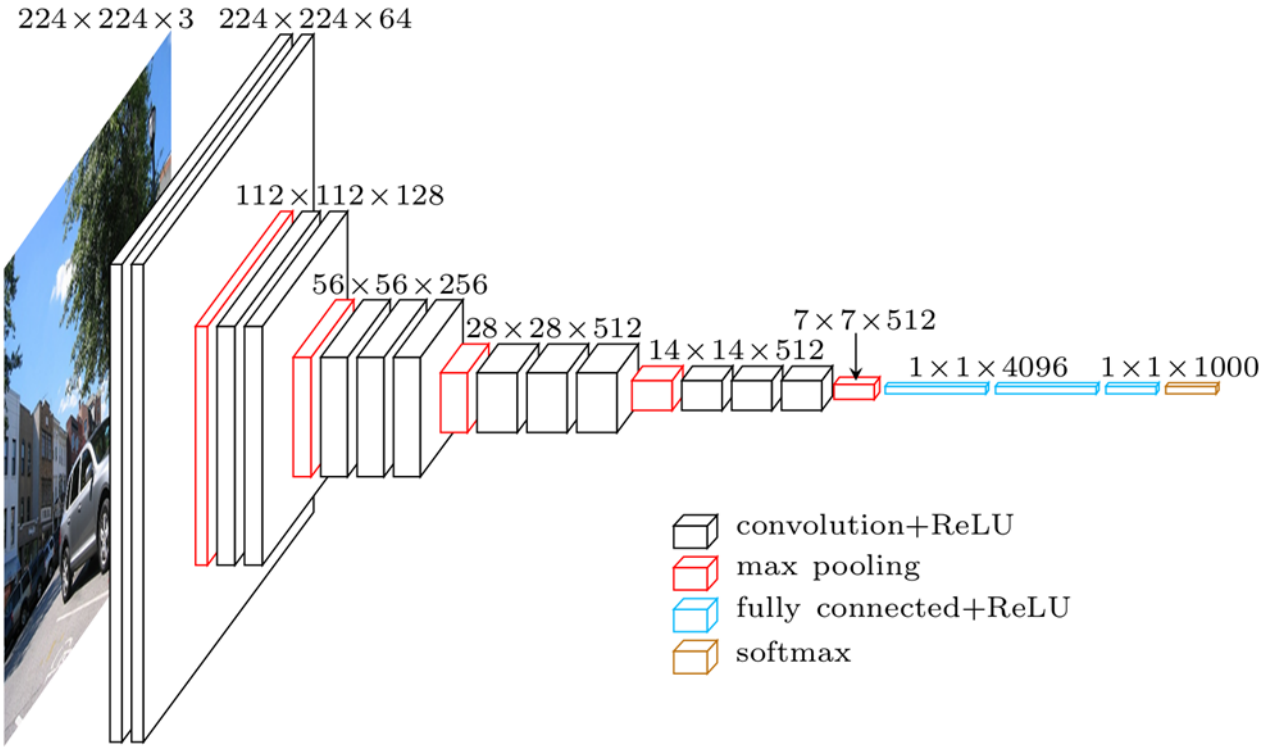

Figura 3-3 Arquitectura VGG

VGG, cuyo nombre proviene del equipo que lo utilizó en la competeción ImageNet ILSVRC 2014, fue publicado en el documento "*Very Deep Convolutional Networks for Large-Scale Image Recognition*" por Karen Simonyan y Andrew Zisserman [55]. Según los estándares actuales, no se consideraría muy profundo, pero en ese momento duplicó con creces el número de capas que se usaban comúnmente e inició la ola "más profundidad → más capacidad → mejor" (donde el factor limitante es la capacidad de entrenamiento).

Cuando se usa VGG para la clasificación, la entrada es un tensor 224 × 224 × 3 (eso significa una imagen RGB de 224x224 píxeles). Esto debe permanecer fijo para la clasificación porque el bloque final de la red utiliza capas completamente conectadas (*Fully-Connected*, FC) (en lugar de convolucionales), que requieren una entrada de longitud fija. Esto generalmente se hace al aplanar la salida de la última capa convolucional, obteniendo un tensor de rango 1, antes de usar las capas FC.

Como vamos a utilizar la salida de una capa convolucional intermedia, el tamaño de la entrada no es nuestro problema. Al menos, no es el problema de este módulo ya que solo se usan capas convolucionales. Vamos a profundizar un poco más en los detalles de bajo nivel y definir qué capa convolucional vamos a utilizar. El documento no especifica qué capa usar; pero en la implementación oficial se puede observar que usan la salida de la capa conv5/conv5_1.

Cada capa convolucional crea abstracciones basadas en la información previa. Las primeras capas generalmente aprenden bordes, la segunda encuentra patrones en los

bordes para activar formas más complejas y así sucesivamente. Finalmente, terminamos con un mapa de características convolucionales que tiene dimensiones espaciales mucho más pequeñas que la imagen original, pero con mayor profundidad. El ancho y la altura del mapa de entidades disminuyen debido a la agrupación aplicada entre capas convolucionales y la profundidad aumenta en función del número de filtros que aprende la capa convolucional.

En su profundidad, el mapa de características convolucionales ha codificado toda la información de la imagen mientras mantiene la ubicación de las "cosas" que ha codificado en relación con la imagen original. Por ejemplo, si había un cuadrado rojo en la parte superior izquierda de la imagen y las capas convolucionales se activan para ello, entonces la información para ese cuadrado rojo todavía estaría en la parte superior izquierda del mapa de características convolucionales.

3.2.1.2.2 VGG vs ResNet

Hoy en día, las arquitecturas de ResNet han reemplazado principalmente a VGG como la red base para extraer funciones. Tres de los coautores de Faster R-CNN (Kaiming He, Shaoqing Ren y Jian Sun) también fueron coautores de "*Deep Residual Learning for Image Recognition*", el documento original que describe ResNet [57].

La ventaja obvia de ResNet sobre VGG es que es más grande, por lo tanto, tiene más capacidad para aprender realmente lo que se necesita. Esto es cierto para la tarea de clasificación y debería ser igualmente cierto en el caso de la detección de objetos.

Además, ResNet facilita el entrenamiento de modelos profundos con el uso de conexiones residuales (*residual connections)* y la normalización de lotes (*batch normalization*), que no se inventó cuando se lanzó VGG por primera vez.

3.2.1.3 Cuadros de Anclaje (Anchors)

Ahora que estamos trabajando con una imagen procesada, necesitamos encontrar propuestas, es decir, regiones de interés para la clasificación. Anteriormente mencionamos que los anclajes son una forma de resolver el problema de longitud variable, pero omitimos la mayor parte de la explicación.

Nuestro objetivo es encontrar cuadros delimitadores en la imagen. Estos tienen forma rectangular y pueden venir en diferentes tamaños y relaciones de aspecto. Supongamos que estamos tratando de resolver el problema sabiendo de antemano que hay dos objetos en la imagen. La primera idea que viene a la mente es entrenar una red que devuelve 8 valores: dos tuplas "*xmin*, *ymin*, *xmax*, *ymax*" que definen un cuadro delimitador para cada objeto. Este enfoque tiene algunos problemas fundamentales. Por ejemplo, las imágenes pueden tener diferentes tamaños y relaciones de aspecto, tener un buen modelo entrenado para predecir coordenadas en bruto puede resultar muy complicado (si no imposible). Otro problema son las predicciones no válidas: al predecir *xmin* y *xmax* tenemos que hacer cumplir de alguna manera que $xmin < xmax$.

Resulta que hay un enfoque más simple para predecir cuadros delimitadores aprendiendo a predecir las compensaciones de los cuadros de referencia. Tomamos un cuadro de referencia *xcenter*, *ycenter*, *width*, *height* y aprendemos a predecir *Δxcenter*, *Δycenter*, *Δwidth*, *Δheight*, que generalmente son valores pequeños que modifican el cuadro de referencia para ajustarse mejor a lo que queremos.

Los anclajes son cuadros delimitadores fijos que se colocan en toda la imagen con diferentes tamaños y proporciones que se utilizarán como referencia al predecir por primera vez las ubicaciones de los objetos.

Como estamos trabajando con un mapa de características convolucionales de tamaño *width* × *height* × *depth*, se crean un conjunto de anclajes para cada uno de los puntos en *width* × *height*. Es importante comprender que, aunque los anclajes se definen en función del mapa de características convolucional, los anclajes finales hacen referencia a la imagen original.

Como solo tenemos capas convolucionales y de agrupación, las dimensiones del mapa de entidades serán proporcionales a las de la imagen original. Matemáticamente, si la imagen tenía un tamaño $w \times h$, el mapa de características terminará siendo $w/r \times h/r$ donde r se denomina relación de submuestreo (*subsampling ratio*). Si definimos un anclaje por posición espacial del mapa de características, la imagen final terminará con un montón de anclajes separados por r píxeles. En el caso de VGG, $r = 16$. Esto se puede observar en la Figura 3-4.

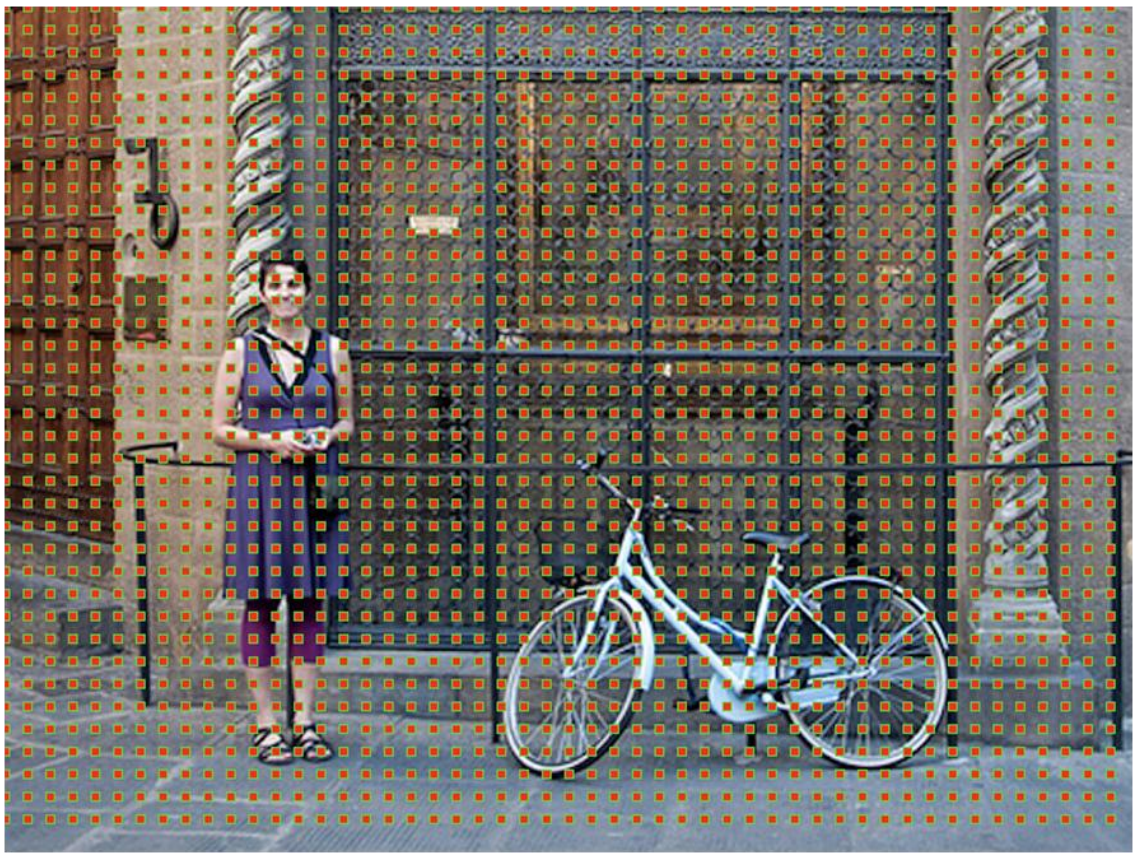

Figura 3-4 Los centros de los anclajes en la imagen original

Para elegir el conjunto de anclajes, generalmente definimos un conjunto de tamaños (por ejemplo, 64px, 128px, 256px) y un conjunto de relaciones entre el ancho y el alto de las cajas (por ejemplo, 0.5, 1, 1.5) y utilizamos todas las combinaciones posibles de tamaños y relaciones.

3.2.1.4 Region Proposal Network

Como mencionamos anteriormente, el RPN toma todos los cuadros de referencia (anclajes) y genera un conjunto de buenas propuestas para objetos. Lo hace al tener dos salidas diferentes para cada uno de los anclajes.

El primero es la probabilidad de que un ancla sea un objeto. Cabe destacar que al RPN no le importa qué clase de objeto es, solo que, de hecho, parece un objeto (y no un fondo). Vamos a utilizar esta puntuación para filtrar las malas predicciones para la segunda etapa. El segundo resultado es la regresión del cuadro delimitador para ajustar los anclajes para que se correspondan mejor al objeto que está prediciendo.

El RPN se implementa de manera eficiente de una manera completamente convolucional, utilizando el mapa de características convolucionales devuelto por la red base como entrada. Primero, usamos una capa convolucional con 512 canales y un tamaño de kernel 3x3 y luego tenemos dos capas convolucionales paralelas que usan un kernel 1x1, cuyo número de canales depende del número de anclajes por punto (Figura 3-5).

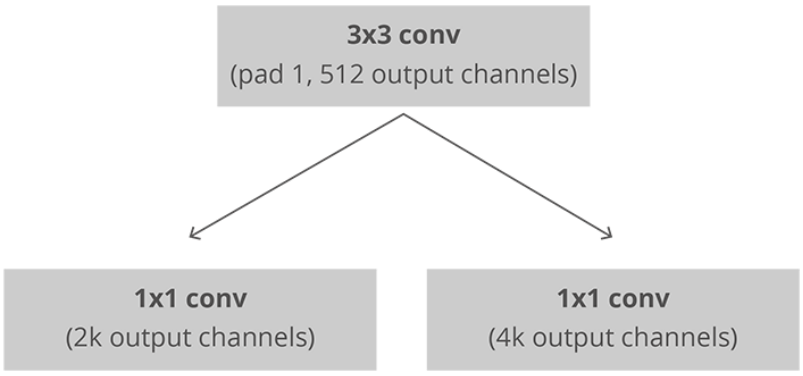

Figura 3-5 Implementación convolucional de una arquitectura RPN, donde k representa el número de anclajes

Para la capa de clasificación, mostramos dos predicciones por anclaje: la puntuación de que sea fondo (no un objeto) y la puntuación de que sea primer plano (un objeto real).

Para la capa de regresión o ajuste de cuadro delimitador, emitimos 4 predicciones: los valores *Δxcenter*, *Δycenter*, *Δwidth*, *Δheight* que aplicaremos a los anclajes para obtener las propuestas finales.

3.2.1.4.1 Entrenamiento, objetivos y funciones de pérdida

El RPN realiza dos tipos diferentes de predicciones: la clasificación binaria y el ajuste de regresión del cuadro delimitador.

Para el entrenamiento, tomamos todos los anclajes y los colocamos en dos categorías diferentes. Los que se superponen a un objeto del *ground truth* con una Intersección sobre Unión (IoU) [59] mayor que 0.5 se consideran "primer plano" y los que no se superponen a ningún objeto de *ground-truth* o tienen menos de 0.1 IoU con objetos del *ground truth* se consideran "fondo".

Luego, muestreamos al azar esos anclajes para formar un mini lote de tamaño 256, tratando de mantener una relación equilibrada entre los anclajes de primer plano y de fondo.

El RPN utiliza todos los anclajes seleccionados para el mini lote para calcular la pérdida de clasificación utilizando la entropía cruzada binaria [60]. Luego, usa solo los anclajes marcados como primer plano para calcular la pérdida de regresión. Para calcular los objetivos para la regresión, usamos el anclaje del primer plano y el objeto del *ground truth* más cercano y calculamos el Δ correcto necesario para transformar el anclaje en el objeto.

En lugar de usar una simple pérdida de L1 (7) (*Mean Absolute Error*) o L2 (8) (*Mean Squared Error*) para el error de regresión, el artículo sugiere usar la pérdida Smooth L1 (9). Smooth L1 es básicamente L1, pero cuando el error de L1 es menor que 1, se emplea L2 y la pérdida disminuye a un ritmo más rápido.

$$L1(x, y) = |x - y| \tag{7}$$

$$L2(x, y) = (x - y)^2 \tag{8}$$

$$SmoothL1(x, y) = \begin{cases} 0.5(x - y)^2, & si\ |x - y| < 1 \\ |x - y| - 0.5, & para\ el\ resto \end{cases} \tag{9}$$

El uso de lotes dinámicos puede ser un desafío por varias razones. Aunque intentamos mantener una relación equilibrada entre los anclajes que se consideran de fondo y los que se consideran en primer plano, eso no siempre es posible. Dependiendo de los objetos del *ground truth* en la imagen y el tamaño y las proporciones de los anclajes, es posible terminar con cero anclajes en primer plano. En esos casos, recurrimos al uso de los anclajes con la IoU mayor en relación con los cuadros del *ground truth*. Esto está

lejos de ser ideal, pero es práctico en el sentido de que siempre tenemos muestras y objetivos en primer plano para aprender.

3.2.1.4.2 Postprocesamiento

En esta fase se aplican principalmente dos procesos, la Supresión de no máximo (*Non-maximun suppression*) y la Selección de Propuestas.

Supresión de no máximo: dado que los anclajes generalmente se superponen, las propuestas también se superponen sobre el mismo objeto. Para resolver el problema de las propuestas duplicadas, utilizamos un enfoque algorítmico simple llamado *Non-Maximum Suppression* (NMS). NMS toma la lista de propuestas ordenadas por puntuación e iteraciones sobre la lista ordenada, descartando aquellas propuestas que tienen un IoU mayor que un umbral predefinido con una propuesta que tiene la puntuación más alta.

Si bien esto parece simple, es muy importante tener cuidado con el umbral de IoU. Si se fija a un valor demasiado bajo, podemos terminar perdiendo propuestas de objetos; Si por el contrario, el valor es demasiado alto, podría terminar con demasiadas propuestas para el mismo objeto. Un valor comúnmente usado es 0.6.

Selección de propuestas: Después de aplicar NMS, mantenemos las N propuestas principales ordenadas por puntuación. En el artículo se propuso inicialmente N = 2000, pero comprobaron que era posible reducir ese número a 300 y aún así obtener resultados bastante buenos [22].

3.2.1.4.3 Aplicación independiente

El RPN se puede usar solo sin necesidad del modelo de segunda etapa. En problemas donde solo hay una clase de objetos, la probabilidad de objetividad se puede usar como la probabilidad de clase final. Esto se debe a que, en este caso, "primer plano" = "clase única" y "fondo" = "clase no única".

Algunos ejemplos de problemas populares (pero aún desafiantes) de aprendizaje automático que pueden beneficiarse del uso independiente del RPN son la detección de rostros y la detección de textos.

Una de las ventajas de usar solo el RPN es la ganancia de velocidad tanto en el entrenamiento como en la predicción. Dado que el RPN es una red muy simple que solo usa capas convolucionales, el tiempo de predicción puede ser más rápido que el uso de la red base de clasificación.

3.2.1.5 Agrupación de la región de interés (Region of Interest Pooling)

Después del paso RPN, tenemos un montón de propuestas de objetos sin clase asignada. Nuestro siguiente problema a resolver es cómo tomar estos cuadros delimitadores y clasificarlos en nuestras categorías deseadas.

El enfoque más simple sería tomar cada propuesta, recortarla y luego pasarla a través de la red de base previamente entrenada. Luego, podemos usar las características extraídas como entrada para un clasificador de imágenes. El principal problema es que ejecutar los cálculos para todas las propuestas (inicialmente 2000, como ya se ha visto anteriormente) es realmente ineficiente y lento.

Faster R-CNN intenta resolver, o al menos mitigar, este problema reutilizando el mapa de características convolucional existente. Esto se realiza mediante la extracción de mapas de características de tamaño fijo para cada propuesta utilizando la agrupación de regiones de interés (Figura 3-6). Se necesitan mapas de características de tamaño fijo para el R-CNN para clasificarlos en un número fijo de clases.

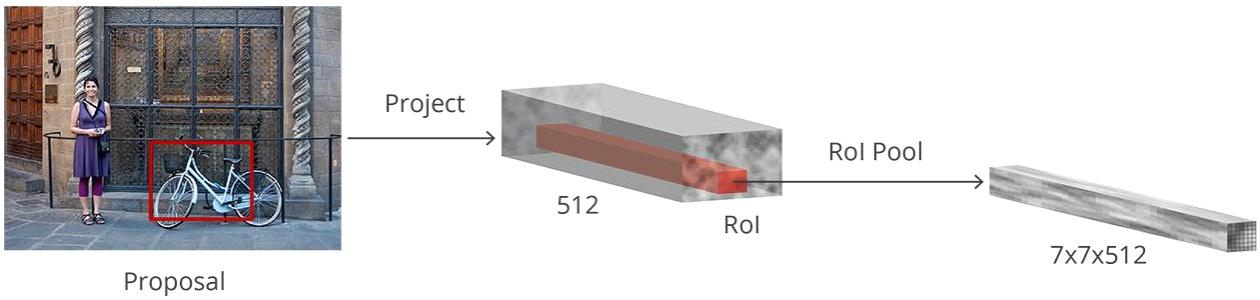

Figura 3-6 Region of Interest Pooling

Un método más simple, que es ampliamente utilizado por las implementaciones de detección de objetos, incluido el Faster R-CNN de Luminoth [61], es recortar el mapa de características convolucionales usando cada propuesta y luego cambiar el tamaño de cada recorte a un tamaño fijo de 14 × 14 × *depth* usando interpolación (generalmente bilineal) . Después del recorte, la agrupación máxima con un *kernel* de 2x2 se usa para obtener un mapa de características de 7 × 7 × *depth* final para cada propuesta.

La razón para elegir esas formas exactas se relaciona con la forma en que el siguiente bloque (R-CNN) lo utiliza. Es importante comprender que son personalizables según el uso de la segunda etapa.

3.2.1.6 Red neuronal convolucional basada en región (Region-based Convolutional Neural Network)

La red neuronal convolucional basada en región (R-CNN) es el paso final en el *pipeline* de Faster R-CNN. Después de obtener un mapa de características convolucional de la imagen, usarlo para obtener propuestas de objetos con el RPN y finalmente extraer características para cada una de esas propuestas (a través de *RoI Pooling*), finalmente necesitamos usar estas características para la clasificación. R-CNN intenta imitar las etapas finales de clasificación de CNN donde se utiliza una capa completamente conectada para generar una puntuación para cada posible clase de objeto (Figura 3-7).

R-CNN tiene dos objetivos diferentes:

- Clasificar las propuestas en una de las clases, más una clase de fondo (para eliminar propuestas incorrectas).
- Ajustar mejor el cuadro delimitador para la propuesta de acuerdo con la clase prevista.

En el documento original Faster R-CNN, el R-CNN toma el mapa de características para cada propuesta, lo aplana y usa dos capas completamente conectadas de tamaño 4096 con activación ReLU.

Luego, usa dos capas completamente conectadas diferentes para cada uno de los diferentes objetos:

- Una capa totalmente conectada con N + 1 unidades donde N es el número total de clases y la adicional es para la clase de fondo.
- Una capa totalmente conectada con unidades 4xN. Queremos tener una predicción de regresión, por lo tanto, necesitamos *Δxcenter*, *Δycenter*, *Δwidth*, *Δheight* para cada una de las N clases posibles.

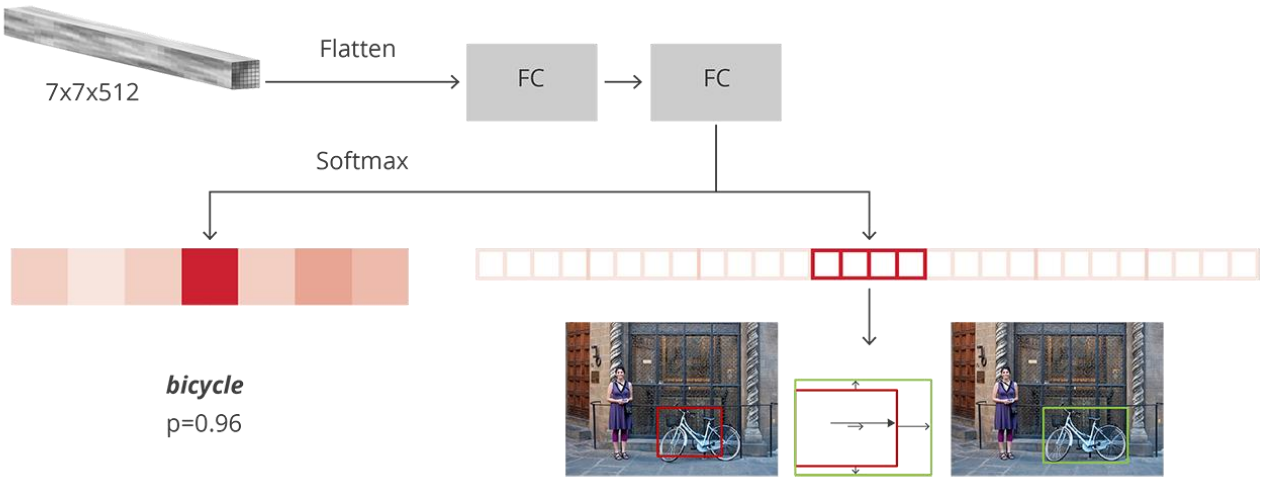

Figura 3-7 Aquitectura R-CNN

3.2.1.6.1 Entrenamiento y objetivos

Los objetivos para R-CNN se calculan casi de la misma manera que los objetivos RPN, pero teniendo en cuenta las diferentes clases posibles. Tomamos las propuestas y los cuadros de *ground-truth*, y calculamos el IoU entre ellos.

Las propuestas que tienen un IoU mayor que 0.5 con cualquier cuadro del *ground-truth* se asignan a dicho *ground-truth*. Los que tienen entre 0.1 y 0.5 se etiquetan como fondo. Al contrario de lo que hicimos al armar objetivos para el RPN, ignoramos las propuestas sin ninguna intersección. Esto se debe a que en esta etapa estamos asumiendo que tenemos buenas propuestas y estamos más interesados en resolver los casos más

difíciles. Por supuesto, todos estos valores son hiperparámetros que se pueden ajustar para adaptarse mejor al tipo de objetos que está tratando de encontrar.

Los objetivos para la regresión del cuadro delimitador se calculan como el desplazamiento entre la propuesta y su cuadro de *ground-truth* correspondiente, solo para aquellas propuestas a las que se les ha asignado una clase según el umbral de IoU.

Supongamos que muestreamos al azar un mini lote equilibrado de tamaño 64 en el que tenemos hasta un 25% de propuestas en primer plano (con clase) y un 75% de propuestas de fondo.

Siguiendo el mismo camino que hicimos para las pérdidas de RPN, la pérdida de clasificación ahora es una pérdida de entropía cruzada multiclase, utilizando todas las propuestas seleccionadas y la pérdida Smooth L1 para las propuestas del 25% que coinciden con un cuadro del *ground-truth*. Debemos tener cuidado al obtener esa pérdida, ya que la salida de la red totalmente conectada R-CNN para las regresiones de cuadro delimitador tiene una predicción para cada una de las clases. Al calcular la pérdida, solo tenemos que tener en cuenta el de la clase correcta.

3.2.1.6.2 Postprocesamiento

Similar al RPN, terminamos con un montón de objetos con clases asignadas que necesitan un procesamiento adicional antes de mostrar los resultados finales.

Para aplicar los ajustes del cuadro delimitador, debemos tener en cuenta cuál es la clase con mayor probabilidad para esa propuesta. También tenemos que ignorar aquellas propuestas que tienen la clase de fondo como la que tiene la mayor probabilidad.

Después de obtener los objetos finales e ignorar los previstos como fondo, aplicamos NMS basados en clases. Esto se hace agrupando los objetos por clase, ordenándolos por probabilidad y luego aplicando NMS a cada grupo independiente antes de unirlos nuevamente.

Para nuestra lista final de objetos, también podemos establecer un umbral de probabilidad y un límite en el número de objetos para cada clase.

3.2.2 SSD

3.2.2.1 Arquitectura

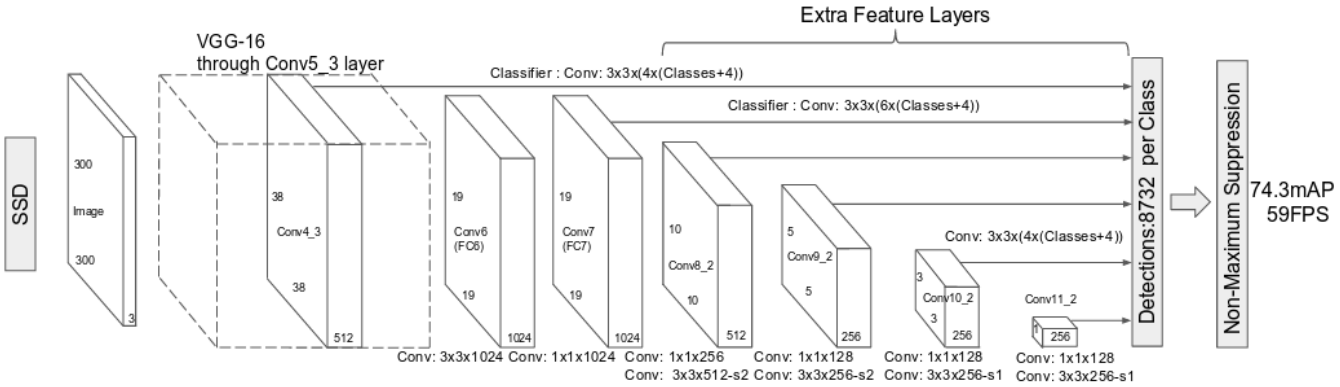

Figura 3-8 Arquitectura de Single Shot MultiBox Detector (la entrada tiene dimensiones 300x300x3)

Como puede ver en la Figura 3-8, la arquitectura de SSD se basa en la aclamada arquitectura VGG-16, pero descarta las capas completamente conectadas (FC). La razón por la que se usó VGG-16 como red base es por su fuerte desempeño en tareas de clasificación de imágenes de alta calidad y su popularidad para resolver los problemas en los que el aprendizaje por transferencia ayuda a mejorar los resultados. En lugar de las capas completamente conectadas VGG originales, se agregó un conjunto de capas convolucionales auxiliares (desde conv6 en adelante), lo que permite extraer características en múltiples escalas y disminuir progresivamente el tamaño de la entrada a cada capa posterior.

3.2.2.2 MultiBox

La técnica de regresión de cuadro delimitador (*bounding box regression*) de SSD está inspirada en el trabajo de Szegedy en MultiBox [62], un método para propuestas rápidas de coordenadas de cuadro delimitador independientes de la clase. Curiosamente, en el trabajo realizado en MultiBox se utiliza una red convolucional de estilo “Inception” [63]. Las convoluciones 1x1 que se ven en la Figura 3-9 que muestra la arquitectura Multibox, ayudan en la reducción de la dimensionalidad ya que el número de dimensiones disminuirá (pero "anchura" y "altura" seguirán siendo las mismas).

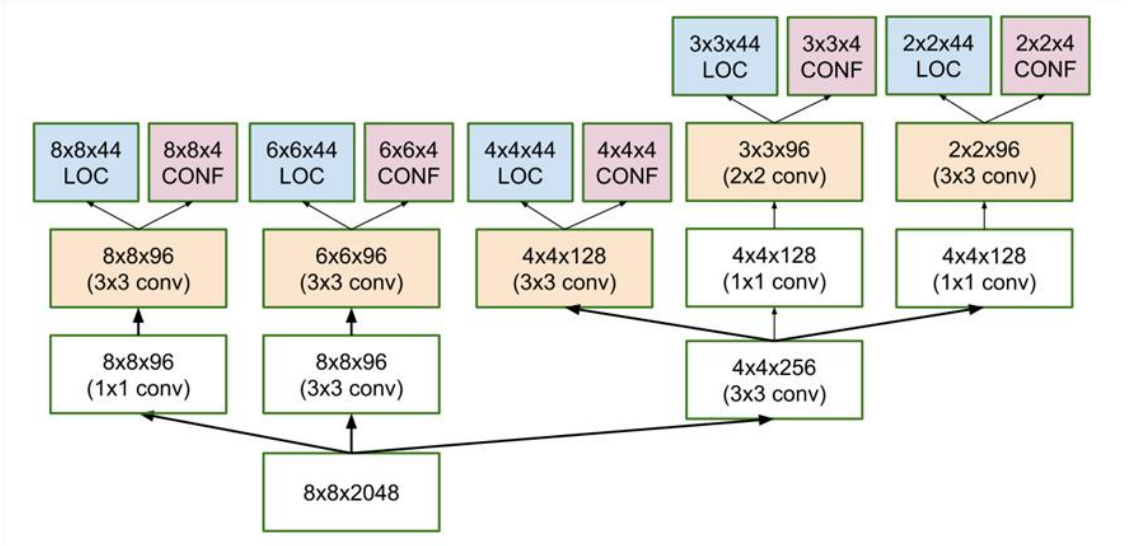

Figura 3-9 Arquitectura de un predictor multiescala de las ubicaciones y confianzas en Multibox.

La función de pérdida de MultiBox también combinó dos componentes críticos que llegaron a SSD:

- *Confidence Loss*: mide cuan segura está la red de la objetividad del cuadro delimitador calculado. La entropía cruzada (cross-entropy) (10) [60] se utiliza para calcular esta pérdida.

$$cross - entropy = -\sum_{c=1}^{M} y_{o,c}\log(p_{o,c}) \quad (10)$$

  - M – número de clases
  - y – indicador binario (0 o 1) si la clase c es correcta para la observación o
  - p – probabilidad de que la observación o pertenezca a la clase c

- *Location Loss*: esto mide cuán lejos están los cuadros delimitadores pronosticados de la red de los verdaderos del conjunto de entrenamiento. La norma L2 (8) (L2-Norm) se usa aquí.

Con estas dos componentes, la expresión de la pérdida, que mide el error de la predicción, viene dada por la ecuación (11).

$$L_{multibox} = L_{conf} + \alpha * L_{loc} \quad (11)$$

El término α ayuda a equilibrar la contribución de la pérdida de ubicación. Como es habitual en el aprendizaje autónomo, el objetivo es encontrar los valores de los

parámetros que reducen de manera óptima la función de pérdida, acercando así nuestras predicciones al *ground-truth*.

3.2.2.3 MultiBox Priors y IoU

En MultiBox, los investigadores crearon lo que llamamos "*priors*" (o "*anchors*" en la terminología Faster-R-CNN), que son cuadros límite de tamaño fijo precalculados que intentan aproximar la distribución de los cuadros del *ground-truth* originales. De hecho, esos "priors" se seleccionan de tal manera que su relación IoU es mayor que 0.5. Un IoU de 0.5 todavía no es lo suficientemente bueno, pero proporciona un punto de partida sólido para el algoritmo de regresión del cuadro delimitador: es una estrategia mucho mejor que comenzar las predicciones con coordenadas aleatorias. Por lo tanto, MultiBox comienza con los *priors* como predicciones e intenta aproximarse lo más posible a los cuadros delimitadores del *ground truth*.

La arquitectura resultante contiene 11 *priors* por celda de mapa de características (8x8, 6x6, 4x4, 3x3, 2x2) y solo uno en el mapa de características 1x1, resultando en un total de 1420 *priors* por imagen, permitiendo así una cobertura robusta de imágenes de entrada a múltiples escalas, para detectar objetos de varios tamaños.

Al final, MultiBox solo retiene un número K de predicciones que han minimizado las pérdidas tanto de ubicación (*location*) como de confianza (*confidence*).

3.2.2.4 Mejoras en SSD

De vuelta en SSD, se agregaron varios ajustes para hacer que esta red sea aún más capaz de localizar y clasificar objetos.

- **Priors fijos:** a diferencia de MultiBox, cada celda del mapa de características está asociada con un conjunto de cuadros delimitadores predeterminados de diferentes dimensiones y relaciones de aspecto. Estos *priors* se eligen manualmente (pero con cuidado), mientras que en MultiBox, se eligieron porque su IoU con respecto al *ground truth* era superior a 0,5. En teoría, esto debería permitir que SSD se generalice para cualquier tipo de entrada, sin requerir una fase de pre-entrenamiento para la generación anterior. Por ejemplo, suponiendo que hayamos configurado 2 puntos diagonalmente opuestos *(x1, y1) y (x2, y2)* para cada b cuadros delimitadores predeterminados por celda del mapa de entidades y c clases para clasificar, en un mapa de entidades dado de tamaño *f = m * n*, SSD calcularía los valores de *f * b * (4 + c)* para este mapa de características.
- **Pérdida de ubicación (*Location Loss*):** SSD utiliza la norma SmoothL1 para calcular la pérdida de ubicación. Si bien no es tan precisa como la norma L2, sigue siendo altamente efectiva y le da a SSD más espacio para maniobrar, ya que no trata de ser "perfecto en píxeles" en su predicción de cuadro delimitador

(es decir, una diferencia de unos pocos píxeles difícilmente sería notable para muchos de nosotros).

- **Clasificación:** MultiBox no realiza la clasificación de objetos, mientras que SSD sí. Por lo tanto, para cada cuadro delimitador predicho, se calcula un conjunto de predicciones, para cada clase posible en el conjunto de datos.

3.2.2.5 Entrenamiento y Ejecución de SSD

A continuación se muestran algunos detalles que hay que tener en cuenta a la hora de entrenar y ejecutar un modelo SSD.

3.2.2.5.1 Cuadros delimitadores predeterminados

Se recomienda configurar un conjunto variado de cuadros delimitadores predeterminados, de diferentes escalas y relaciones de aspecto para garantizar que se puedan capturar la mayoría de los objetos. En el artículo sobre SSD se emplean 6 cuadros delimitadores por celda de mapa de características [25].

3.2.2.5.2 Mapas de características

Los mapas de características (es decir, los resultados de los bloques convolucionales) son una representación de las características dominantes de la imagen a diferentes escalas, por lo tanto, ejecutar MultiBox en mapas de características múltiples aumenta la probabilidad de que algún objeto (grande y pequeño) sea finalmente detectado, localizado y debidamente clasificado La Figura 3-10 muestra cómo la red "ve" una imagen determinada en sus mapas de características:

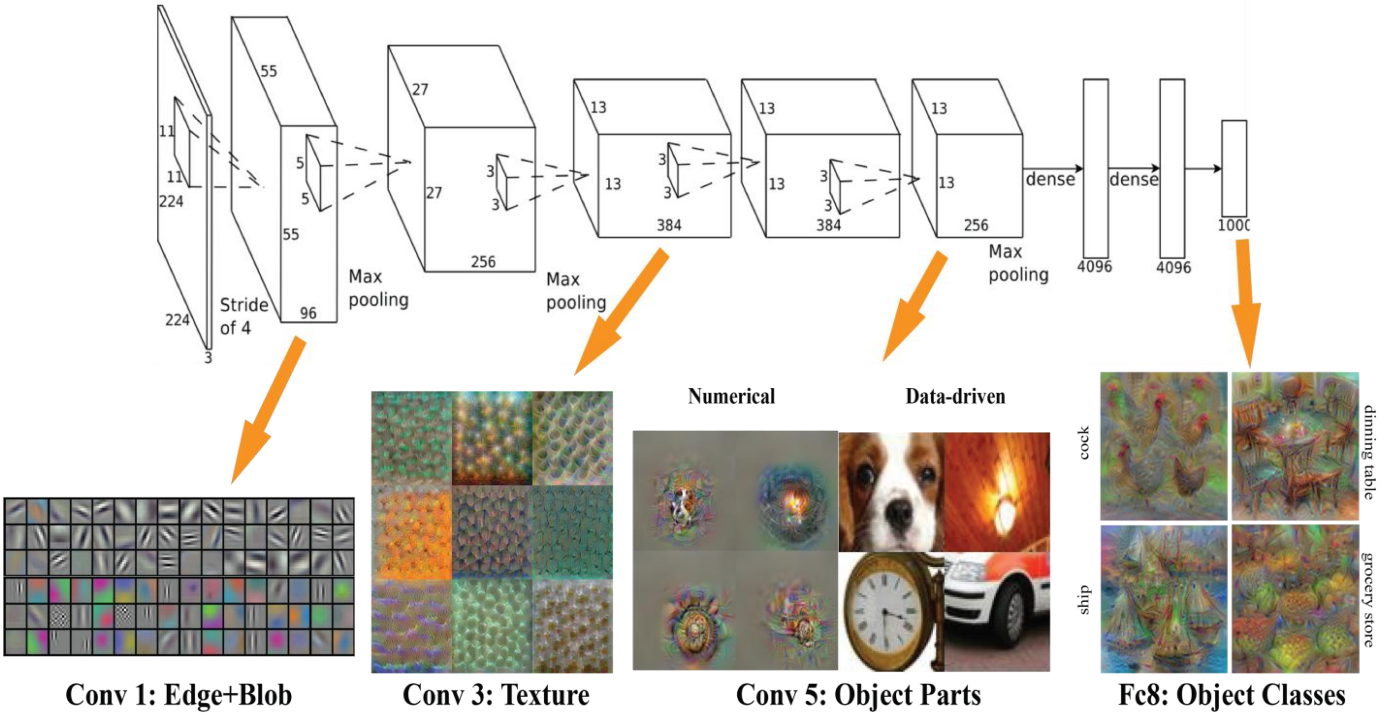

Figura 3-10 Visualización de los Mapas de Características para VGG

3.2.2.5.3 Data Augmentation

Los autores de SSD declararon que el aumento de datos, como en muchas otras aplicaciones de aprendizaje profundo, ha sido crucial para enseñar a la red a ser más robusta a los diversos tamaños de objetos en la entrada. Con este fin, generaron ejemplos de entrenamiento adicionales con fragmentos de la imagen original en diferentes proporciones de IoU (por ejemplo, 0.1, 0.3, 0.5, etc.) y fragmentos aleatorios también. Además, cada imagen también se voltea horizontalmente al azar con una probabilidad de 0.5 (véase la Figura 3-11), asegurando así que los objetos potenciales aparezcan a la izquierda y a la derecha con una probabilidad similar.

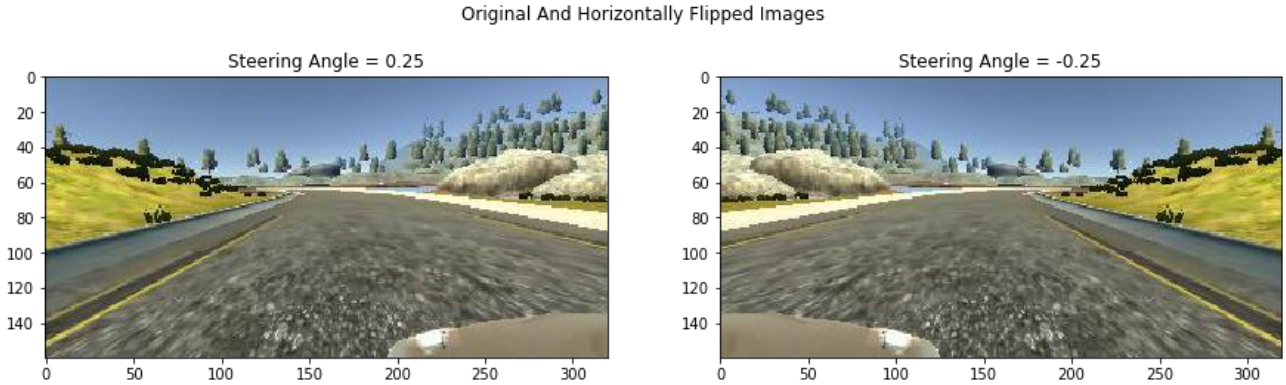

Figura 3-11 Ejemplo de aplicación de simetría horizontal para implementar Data Augmentation.

3.2.2.5.4 Non-Maximum Suppression (NMS)

Dada la gran cantidad de cuadros delimitadores generados durante la ejecución de SSD en el tiempo de inferencia, es esencial eliminar la mayor parte de los cuadros delimitadores mediante la aplicación de una técnica conocida como NMS: casillas con un umbral de pérdida de confianza inferior a un umbral (ej: 0.01) e IoU inferiores a otro umbral (ej: 0,45) se descartan y solo se mantienen las N predicciones principales. Esto garantiza que solo las predicciones más probables sean retenidas por la red, mientras que las más ruidosas se eliminan (Figura 3-12).

Figura 3-12 Ejemplo de NMS.

3.2.2.6 Notas adicionales sobre SSD

El artículo sobre SSD hace las siguientes observaciones adicionales:

- Más cuadros predeterminados dan como resultado una detección más precisa, aunque hay un impacto en la velocidad.
- Tener MultiBox en múltiples capas también resulta en una mejor detección, debido a que el detector se ejecuta en funciones con múltiples resoluciones.
- El 80% del tiempo se gasta en la red VGG-16 base: esto significa que con una red más rápida e igualmente precisa, el rendimiento de SSD podría ser aún mejor.
- SSD confunde objetos con categorías similares (por ejemplo, animales). Esto probablemente se deba a que las ubicaciones se comparten para varias clases.
- SSD-500 (la variante de resolución más alta que usa imágenes de entrada de 512x512) logra el mejor mAP en Pascal VOC2007 a 76.8%, pero a expensas de la velocidad, donde su velocidad de cuadros baja a 22 fps. SSD-300 es, por lo tanto, una compensación mucho mejor con 74.3 mAP a 59 fps.
- SSD produce un peor rendimiento en objetos más pequeños, ya que pueden no aparecer en todos los mapas de características. El aumento de la resolución de la imagen de entrada alivia este problema pero no lo resuelve por completo.

3.2.3 YOLO

3.2.3.1 El vector de predicciones

El primer paso para comprender YOLO es cómo codifica su salida. La imagen de entrada se divide en una cuadrícula de celdas SxS. Para cada objeto que está presente en la imagen, se dice que una celda de la cuadrícula es "responsable" de predecirla. Esa es la celda donde cae el centro del objeto.

Cada celda de la cuadrícula predice los cuadros delimitadores B y las probabilidades de la clase C. La predicción del cuadro delimitador tiene 5 componentes: (*x*, *y*, *w*, *h*, *confianza*). Las coordenadas (*x*, *y*) representan el centro del cuadro, en relación con la ubicación de la celda de la cuadrícula (si el centro del cuadro no cae dentro de la celda de la cuadrícula, entonces esta celda no es responsable de éste). Estas coordenadas están normalizadas para caer entre 0 y 1. Las dimensiones del cuadro (*w*, *h*) también están normalizadas a [0, 1], en relación con el tamaño de la imagen. Se muestra un ejemplo en la Figura 3-13.

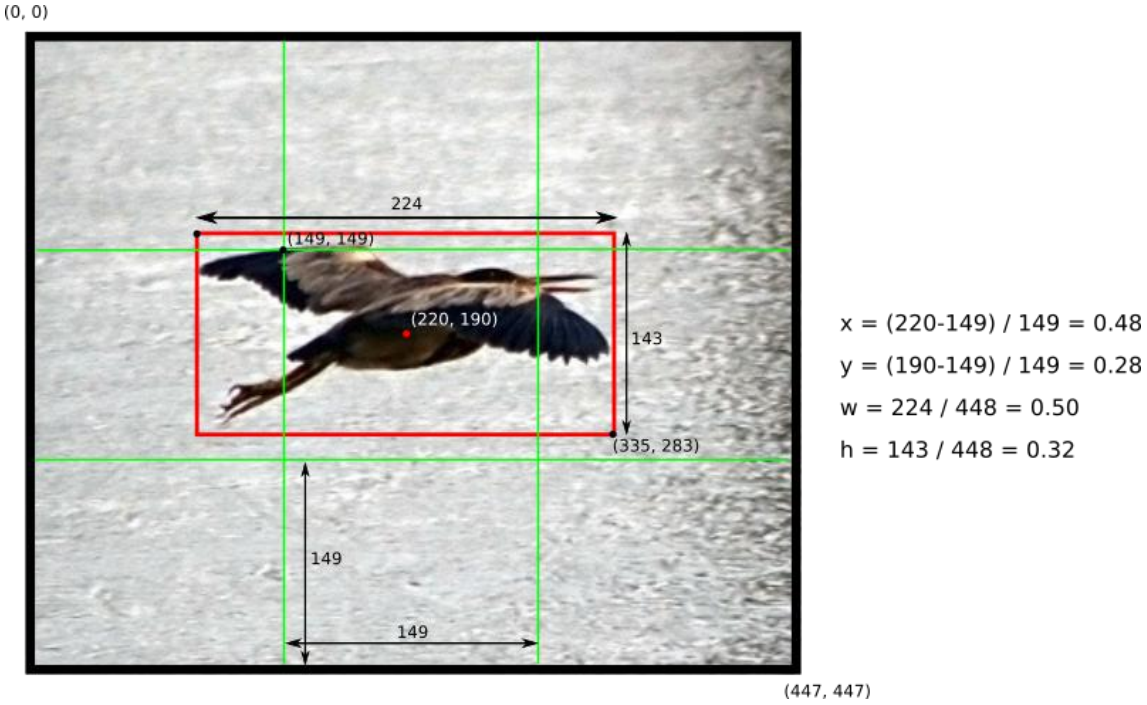

Figura 3-13 Ejemplo de cómo calcular coordenadas de caja en una imagen 448x448 con S = 3. Obsérvese cómo se calculan las coordenadas (x, y) en relación con la celda de la cuadrícula central.

Todavía hay un componente más en la predicción del cuadro delimitador, que es la puntuación de *confianza*. Formalmente definen la confianza como:

$$\text{Confidence} = \mathcal{P}(obj) * IoU(pred, gt) \quad (12)$$

Si no existe ningún objeto en esa celda, la puntuación de confianza debe ser cero. De lo contrario, se busca que la puntuación de confianza sea igual a la intersección sobre la unión (IoU) entre el cuadro predicho y el *ground truth*. Cabe destacar que la confianza refleja la presencia o ausencia de un objeto de cualquier clase.

Ahora que entendemos las 5 componentes de la predicción de recuadro, hay que recordar que cada celda de la cuadrícula hace B de esas predicciones, por lo que en total hay $S^2 \cdot B \cdot 5$ salidas relacionadas con predicciones de cuadro delimitador.

También es necesario predecir las probabilidades de clase, $\mathcal{P}(\text{clase (i)} \mid \text{objeto})$. Esta probabilidad está condicionada por la celda de la cuadrícula que contiene un objeto. En la práctica, significa que si no hay ningún objeto presente en la celda de la cuadrícula, la función de pérdida no lo penalizará por una predicción de clase incorrecta, como veremos más adelante. La red solo predice un conjunto de probabilidades de clase por celda, independientemente del número de cuadros B. Eso hace que se tengan $S^2 \cdot C$ probabilidades de clase en total (Figura 3-14). Agregando las predicciones de clase al vector de salida, obtenemos un tensor SxSx(B · 5 + C) como salida.

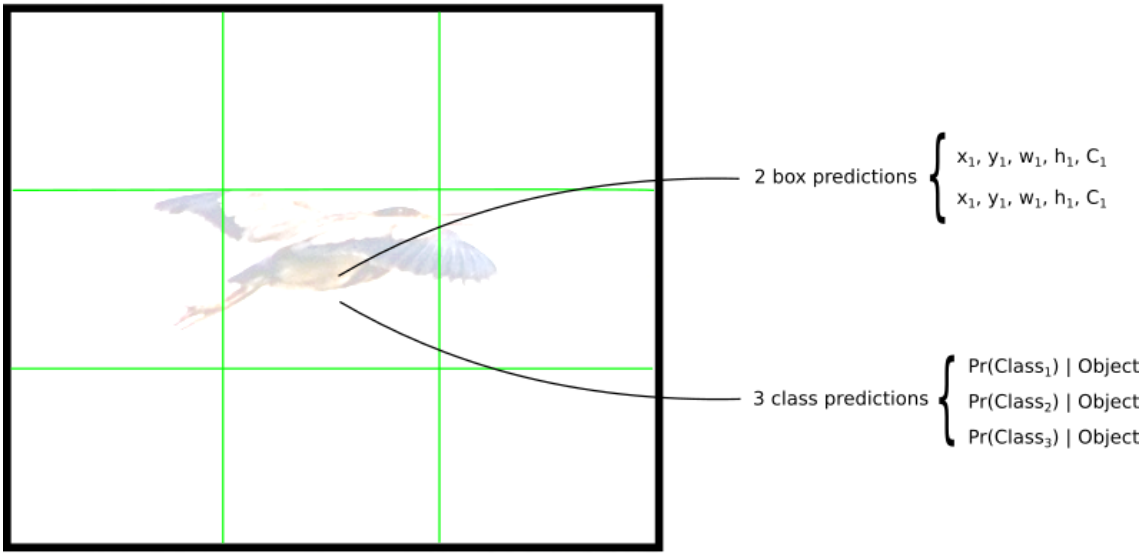

Figura 3-14. Cada celda de la cuadrícula hace predicciones de cuadro delimitador B y predicciones de clase C (S = 3, B = 2 y C = 3 en este ejemplo).

3.2.3.2 La red

Una vez comprendido cómo se codifican las predicciones, el resto es más sencillo. La estructura de la red (ver Tabla 3.2-1) se parece a una CNN normal, con capas convolucionales y de agrupación máxima, seguidas de 2 capas completamente conectadas al final:

Nombre	Filtros	Dimensión de salida
Conv 1	7 x 7 x 64, stride=2	224 x 224 x 64
Max Pool 1	2 x 2, stride=2	112 x 112 x 64
Conv 2	3 x 3 x 192	112 x 112 x 192
Max Pool 2	2 x 2, stride=2	56 x 56 x 192
Conv 3	1 x 1 x 128	56 x 56 x 128
Conv 4	3 x 3 x 256	56 x 56 x 256
Conv 5	1 x 1 x 256	56 x 56 x 256
Conv 6	1 x 1 x 512	56 x 56 x 512
Max Pool 3	2 x 2, stride=2	28 x 28 x 512
Conv 7	1 x 1 x 256	28 x 28 x 256
Conv 8	3 x 3 x 512	28 x 28 x 512
Conv 9	1 x 1 x 256	28 x 28 x 256
Conv 10	3 x 3 x 512	28 x 28 x 512
Conv 11	1 x 1 x 256	28 x 28 x 256
Conv 12	3 x 3 x 512	28 x 28 x 512
Conv 13	1 x 1 x 256	28 x 28 x 256
Conv 14	3 x 3 x 512	28 x 28 x 512
Conv 15	1 x 1 x 512	28 x 28 x 512
Conv 16	3 x 3 x 1024	28 x 28 x 1024
Max Pool 4	2 x 2, stride=2	14 x 14 x 1024
Conv 17	1 x 1 x 512	14 x 14 x 512
Conv 18	3 x 3 x 1024	14 x 14 x 1024
Conv 19	1 x 1 x 512	14 x 14 x 512
Conv 20	3 x 3 x 1024	14 x 14 x 1024
Conv 21	3 x 3 x 1024	14 x 14 x 1024
Conv 22	3 x 3 x 1024, stride=2	7 x 7 x 1024
Conv 23	3 x 3 x 1024	7 x 7 x 1024
Conv 24	3 x 3 x 1024	7 x 7 x 1024
FC 1	-	4096
FC 2	-	7 x 7 x 30 (1470)

Tabla 3.2-1, Arquitectura de la red de YOLO.

Algunos comentarios sobre la arquitectura:

- La arquitectura fue diseñada para su uso en el conjunto de datos Pascal VOC, donde los autores utilizaron S = 7, B = 2 y C = 20. Esto explica por qué los mapas de características finales son 7x7, y también explica el tamaño de la salida (7x7x $(2 \cdot 5 + 20)$). El uso de esta red con un tamaño de cuadrícula diferente o un número diferente de clases puede requerir el ajuste de las dimensiones de la capa.
- Hay una versión más rápida de YOLO, llamada Tiny-YOLO, con menos capas convolucionales. La tabla anterior, sin embargo, muestra la versión completa.
- Las secuencias de las capas de reducción 1x1 y las capas convolucionales 3x3 se inspiraron en el modelo GoogLeNet (Inception) [63].
- La capa final utiliza una función de activación lineal. Todas las otras capas usan un *Leaky ReLU* (13) con $\alpha = 0.01$.

$$LeakyReLU(z) = \begin{cases} z, & z > 0 \\ \alpha z, & z \leq 0 \end{cases} \quad (13)$$

.

3.2.3.3 La función de pérdida

Hay mucho que decir sobre la función de pérdida, así que hagámoslo por partes.
Comienza así:

$$\lambda_{coord} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{obj} (x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2 \quad (14)$$

La ecuación (14) calcula la pérdida relacionada con la posición prevista del cuadro delimitador *(x, y)*. La función calcula una suma sobre cada predictor de cuadro delimitador *(j = 0 ... B)* de cada celda de la cuadrícula *(i = 0 ... S ^ 2)*.

λ_{coord} es una constante dada y $\mathbb{1}^{obj}$ se define de la siguiente manera:

- 1, si un objeto está presente en la celda de la cuadrícula *i* y el predictor *j-ésimo* del cuadro delimitador es "responsable" de esa predicción
- 0, de lo contrario

Nos encontramos con el problema de cómo determinar qué predictor es responsable del objeto. Citando el artículo original [28]:

"YOLO predice varios cuadros delimitadores por celda de cuadrícula. En el momento del entrenamiento, solo queremos que un predictor de cuadro delimitador sea responsable de cada objeto. Asignamos a un predictor para que sea "responsable" de predecir un objeto en función del cual la predicción tiene el IOU real más alto con el *ground truth*."

Los otros términos en la ecuación deberían ser fáciles de entender: *(x, y)* son la posición prevista del cuadro delimitador y *(x̂, ŷ)* son la posición real de los datos de entrenamiento.

Pasemos a la segunda parte:

$$\lambda_{coord} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{obj} (\sqrt{w_i} - \sqrt{\hat{w}_i})^2 + (\sqrt{h_i} - \sqrt{\hat{h}_i})^2 \quad (15)$$

Esta es la pérdida relacionada con el ancho/alto del cuadro predicho. La ecuación (15) se parece a la primera, excepto por la raíz cuadrada. Citando nuevamente el artículo [28]:

"Nuestra métrica de error debe reflejar que las pequeñas desviaciones en las cajas grandes importan menos que en las cajas pequeñas. Para abordar esto parcialmente, predecimos la raíz cuadrada del ancho y alto del cuadro delimitador en lugar del ancho y alto directamente."

Pasando a la tercera parte:

$$\sum_{i=0}^{S^2}\sum_{j=0}^{B} \mathbb{1}_{ij}^{obj} (C_i - \hat{C}_i)^2 + \lambda_{noobj} \sum_{i=0}^{S^2}\sum_{j=0}^{B} \mathbb{1}_{ij}^{noobj} (C_i - \hat{C}_i)^2 \quad (16)$$

En (16) calculamos la pérdida asociada con la puntuación de confianza para cada predictor de cuadro delimitador. *C* es el puntaje de confianza y $\hat{C}$ es la intersección sobre la unión del cuadro delimitador predicho con el *ground truth*. $\mathbb{1}^{obj}$ es igual a uno cuando hay un objeto en la celda, y 0 en caso contrario. $\mathbb{1}^{noobj}$ es lo contrario.

Los parámetros λ que aparecen aquí y también en la primera parte se usan para ponderar de manera diferente las partes de las funciones de pérdida. Esto es necesario para aumentar la estabilidad del modelo. La penalización más alta es para predicciones de coordenadas ($\lambda_{coord} = 5$) y la más baja para predicciones de confianza cuando no hay ningún objeto presente ($\lambda_{noobj} = 0.5$).

La última parte de la función de pérdida es la pérdida de clasificación:

$$\sum_{i=0}^{S^2} \mathbb{1}_{ij}^{obj} \sum_{c \in classes} (p_i(c) - \hat{p}_i(c))^2 \quad (17)$$

La ecuación (17) se asemeja a una suma de errores al cuadrado para la clasificación, excepto por el término $\mathbb{1}^{obj}$. Este término se usa porque no penalizamos el error de clasificación cuando no hay ningún objeto presente en la celda (de ahí la probabilidad de clase condicional discutida anteriormente).

3.2.3.4 Limitaciones de YOLO

YOLO impone fuertes restricciones espaciales en las predicciones de cuadro delimitador ya que cada celda de la cuadrícula solo predice dos cuadros y solo puede tener una clase. Esta restricción espacial limita el número de objetos cercanos que el modelo puede predecir. El modelo presenta dificultades a la hora de detectar pequeños objetos que aparecen en grupos, como bandadas de pájaros. Dado que aprende a predecir cuadros delimitadores a partir de los datos, se esfuerza por generalizar a objetos en relaciones o configuraciones de aspecto nuevas o inusuales. YOLO también utiliza características relativamente burdas para predecir cuadros delimitadores, ya que nuestra arquitectura tiene múltiples capas de disminución de resolución de la imagen de entrada. Finalmente, mientras entrenamos en una función de pérdida que se aproxima al rendimiento de detección, la función de pérdida trata los errores de la misma manera en los cuadros delimitadores pequeños que en los cuadros delimitadores grandes. Un error pequeño en un cuadro grande es generalmente benigno, pero un pequeño error en un cuadro pequeño tiene un efecto mucho mayor en la IoU. La principal fuente de error en YOLO son las localizaciones incorrectas.

3.3 MEDIDA DEL RENDIMIENTO DE LOS DETECTORES DE OBJETOS

Para medir el rendimiento de los diferentes algoritmos se ha optado por preparar un programa que realice unas pruebas sobre un limitado número de imágenes del *dataset* y mida los dos parámetros principales para la detección, "*inference time*" y "*mAP*", para cada uno de los algoritmos que se desean probar.

Para definir el conjunto de imágenes se han escogido las primeras *n* imágenes del *dataset*, donde *n* es un parámetro regulable. Después, para cada detector se ha realizado la detección y almacenado los resultados, para compararlos con los del *ground-truth*.

Para cada imagen del conjunto se realiza la detección y se repite múltiples veces (en nuestro experimento se optó por un número no demasiado grande 10, para no aumentar en exceso el tiempo de ejecución total), para así calcular el tiempo de inferencia medio para cada imagen. Una vez se llega a la última imagen se hace de nuevo la media para calcular el tiempo de inferencia medio global. Este es el valor que después se usa en los resultados.

Para calcular el mAP, realizamos el proceso expuesto en la Sección 2.4.2 de este documento, pero tomando como positivas las detecciones que tengan IoU mayor de 0.75. Y así obtenemos el valor de mAP para el detector y el *subset* de imágenes.

Los resultados obtenidos se muestran en el Capítulo 4.

3.4 ENTRENAMIENTO EN EL DATASET DE BERKELEY

Ahora que se tienen unas medidas del rendimiento general de los detectores, vamos a proceder a entrenarlos sobre nuestro *dataset*. Ajustar los parámetros de la red neuronal para que se ajusten a los objetos que queremos detectar. Para ello se alimenta a la red con datos, y gracias al aprendizaje profundo la propia red va modificando los valores de los parámetros internos hasta que la salida se ajuste a lo que se espera.

En nuestro caso, se ha realizado el entrenamiento de dos detectores de objetos, uno basado en SSD con MOBILENETv2 [64] y otro basado en YOLOv3 [65], teniendo principalmente en cuenta los requisitos de tiempo real, es decir, primando el tiempo de inferencia.

Para realizar el entrenamiento del detector SSD se ha empleado el script que proporciona *Tensorflow Object Detection API* (TODAPI) para ello, llamado *model_main.py*. La ventaja de disponer de un programa ya implementado es que nos ahorramos el tiempo de desarrollo, pero la desventaja es que debemos ajustarnos a las especificaciones del programa para poder ejecutarlo. Por ello, antes de poder emplearlo se deben cumplir ciertos requisitos. Primero es necesario transformar nuestro *dataset* al formato específico de Tensorflow, TFRecord [1]. Para ello, basándose en el script de ejemplo que presentan en la documentación, se ha programado el script

"*bdd_to_tfrecord.py*". Lo que hace es recorrer cada una de las imágenes del *dataset* (de entrenamiento o validación) junto con sus anotaciones, genera el TFRecord correspondiente y los agrupa en "shards" de 100 elementos.

El problema es que nuestro *dataset* es muy grande, y por ello es mejor trabajar con un subgrupo de imágenes (para reducir el tiempo de entrenamiento), por ello se ha ajustado el script para que sólo emplee un número limitado de imágenes con sus respectivas anotaciones, al que se ha denominado "*bdd_to_tfrecord_subset.py*". En nuestro caso se ha elegido una de cada diez imágenes del *dataset* para pasar de 100000 a 10000 imágenes.

Una vez tenemos los ficheros TFRecord, y antes de ejecutar el script de entrenamiento, necesitamos ajustar los parámetros del modelo. El API de Tensorflow hace uso de unos fichero de configuración denominados "*protobuf files*" que contienen los parámetros que se le pasarán al estimador para que funcione de forma correcta. Estos ficheros se dividen en varias partes:

La configuración *model*, define qué tipo de modelo se entrenará (es decir, meta-arquitectura, extractor de características).

El *train_config*, que decide qué valores se deben usar para entrenar los parámetros del modelo (es decir, los parámetros SGD, el preprocesamiento de entrada y los valores de inicialización del extractor de características).

El *eval_config*, que determina qué conjunto de métricas se informará para la evaluación.

El *train_input_config*, que define en qué conjunto de datos debe entrenarse el modelo.

El *eval_input_config*, que define en qué conjunto de datos se evaluará el modelo. Normalmente, esto debería ser diferente al conjunto de datos de entrada de entrenamiento.

De esta forma ajustamos los valores correspondientes para que se ajusten a nuestras necesidades. A continuación se muestra el fichero de configuración que se ha usado para entrenar el modelo SSD Mobilenetv2, que se ha generado a partir del que proporciona el API. La estructura de dicho fichero se puede consultar en el Apéndice C.

Una vez tenemos el fichero de configuración listo, procedemos a ejecutar el script de entrenamiento indicando la ruta de la salida, los ficheros de configuración, y otras opciones adicionales.

Una vez se inicia el proceso de entrenamiento, podemos ver su evolución mediente TensorBoard indicando la ruta de salida del entrenador.

En el caso de YOLOv3, no podemos emplear este sistema ya que el API no tiene implementado este modelo. En principio, YOLOv3 se basa en la arquitectura Darknet53 que está escrita en lenguaje C, lo cual representa un problema, ya que nosotros estamos trabajando en *python* y *tensorflow*. Sin embargo, como ya se ha comentado anteriormente existen implementaciones de YOLOv3 en *tensorflow* realizadas por

diferentes usuarios de *github*. Volvemos a hacer uso de la implementación de YOLO empleada en el programa anterior, que ya trae implementado todo el flujo de entrenamiento y validación. Sin embargo para hacer uso de dicha implementación, necesitamos cumplir con los requisitos de formato que se emplea para los *dataset*, que es un fichero de texto que contiene el nombre del fichero de la imagen y cada una de los cuadros de detección separados por espacios.

Por lo tanto, primeramente hay que transformar nuestro *groundtruth* a un fichero de texto con el formato específico que se basa en una línea para cada imagen donde se encuentran la ruta al fichero, la clase y las coordenadas de la caja:

- ruta_imagen índice_etiqueta1 xmin ymin xmax ymax …

Para ello generamos un script, "*convert_annot.py*" para transformar nuestras anotaciones al formato deseado.

Necesitamos también, los *anchors* que se usarán para proponer los *boxes* así que ejecutamos el script "*get_kmeans.py*" y guardamos el resultado (solo la ristra de números con las coordenadas) en un fichero de texto.

Así mismo necesitamos transformar los pesos que se proporcionan en la página de YOLO al formato que emplea Tensorflow. Para ello ejecutamos "*convert_weights.py*".

Con todos estos elementos ya podemos ejecutar el script para empezar el entrenamiento en nuestro *dataset*.

3.5 SELECCIÓN DE ALGORITMOS DE TRACKING

Habiendo seleccionado y entrenado nuestros modelos de detectores de objetos, la misión ahora es seleccionar el rastreador que en base a las detecciones proporcionadas realice un seguimiento (tracking) de cada objeto detectado asignándole in identificador.

Para ello, implementamos el siguiente pipeline:

- Captación de la imagen, de un vídeo o una secuencia de imágenes.
- Aplicamos la detección en un *frame*.
- Ejecutamos el algoritmo de tracking para establecer los rastreadores.
- Lo vamos ejecutando de forma recurrente.

Para llevar a cabo la tarea se ha optado por implementar dos rastreadores, un algoritmo SORT y un algoritmo DeepSORT, principalmente por tres motivos:

- Tienen un bajo impacto en el tiempo de ejecución.
- Proporcionan rastreo de múltiples objetos.
- Se basan en estimaciones de cuadros delimitadores que es lo que proporcionan nuestros detectores.

Ambos rastreadores se basan en el filtrado de Kalman, como ya se comento en 2.5. Expliquemos a continuación con más detalle su funcionamiento.

3.5.1 SORT

Como ya se ha mencionado anteriormente, SORT hace uso del filtro de Kalman para predecir las posiciones de los cuadros delimitadores.

3.5.1.1 Filtro de Kalman para la determinación del cuadro delimitador

El filtro Kalman [66] tiene las siguientes características importantes de las que se puede beneficiar el seguimiento:

- Predicción de la ubicación futura del objeto.
- Corrección de la predicción basada en nuevas mediciones.
- Reducción del ruido introducido por detecciones inexactas
- Facilitar el proceso de asociación de múltiples objetos a sus pistas.

El filtro de Kalman consta de dos pasos: predicción y actualización. El primer paso usa estados anteriores para predecir el estado actual. Las ecuaciones para la fase de predicción son:

$$\bar{\mathbf{x}} = \mathbf{F}\mathbf{x} + \mathbf{B}\mathbf{u} \tag{18}$$

$$\bar{\mathbf{P}} = \mathbf{F}\mathbf{P}\mathbf{F}^{\mathbf{T}} + \mathbf{Q} \tag{19}$$

donde,

- **x**: state mean
- **P**: state covariance
- **F**: state transition matrix
- **Q**: process covariance
- **B**: control function (matrix)
- **u**: control input

El segundo paso utiliza la medición actual, en nuestro caso la ubicación del cuadro delimitador de detección, para corregir el estado, cuyas ecuaciones son:

$$\mathbf{y} = \mathbf{z} - \mathbf{H}\bar{\mathbf{x}} \tag{20}$$

$$\mathbf{K} = \bar{\mathbf{P}}\mathbf{H}^{\mathbf{T}}(\mathbf{H}\bar{\mathbf{P}}\mathbf{H}^{\mathbf{T}} + \mathbf{R})^{-\mathbf{1}} \tag{21}$$

$$\mathbf{x} = \bar{\mathbf{x}} + \mathbf{K}\mathbf{y} \tag{22}$$

$$\mathbf{P} = (\mathbf{I} - \mathbf{K}\mathbf{H})\bar{\mathbf{P}} \tag{23}$$

donde,

- **H**: measurement function (matrix)

- **z**: measurement
- **R**: measurement noise covariance
- **y**: residual
- **K**: Kalman gain

3.5.1.2 Asignación de los datos. Algoritmo Húngaro

Para asignar las detecciones a objetos existentes, la geometría de cada cuadro delimitador del objeto se estima prediciendo su nueva ubicación en el *frame* actual. La matriz de costes de asignación se calcula como la IOU entre cada detección y todos los cuadros delimitadores previstos de los objetos existentes. La tarea se resuelve de manera óptima utilizando el algoritmo húngaro. Además, se impone un IoU mínimo para rechazar asignaciones donde la superposición de la detección sobre el objeto es menor que la IOU mínima [40].

3.5.1.3 Creación y eliminación de identidades

Cuando los objetos entran y salen de la imagen, es necesario crear o eliminar las identidades de dichos objetos. Para la creación de rastreadores, consideramos que cualquier detección con una superposición menor que una IoU mínima significa la existencia de un objeto sin seguimiento. El rastreador se inicializa utilizando la geometría del cuadro delimitador con la velocidad establecida en cero. Como la velocidad no se observa en este punto, la covarianza de la componente de velocidad se inicializa con valores grandes, lo que refleja esta incertidumbre. Además, el nuevo rastreador pasa por un período de prueba en el que el objetivo debe asociarse con detecciones para acumular suficiente evidencia a fin de evitar el seguimiento de falsos positivos [40].

Los rastreadores se eliminan si no se detectan para un número determinado de *frames*. Esto evita un crecimiento ilimitado en la cantidad de rastreadores y errores de localización causados por predicciones de duración excesiva sin correcciones del detector.

Los detalles de la implementación en *python* se exponen en el Apéndice D.

3.5.1.4 Limitaciones:

El problema principal es la oclusión. Por ejemplo, cuando un automóvil está adelantando a otro automóvil, los dos automóviles pueden estar muy cerca el uno del otro. Esto puede engañar al detector para que emita un cuadro único (y posiblemente más grande), en lugar de dos cuadros delimitadores separados. Además, el algoritmo de seguimiento puede tratar esta detección como una nueva detección y configura una nueva pista. El algoritmo de seguimiento puede fallar nuevamente cuando uno de los automóviles que pasan se aleja de otro automóvil. Para aliviar este problema se creó DeepSORT, que utiliza una pequeña red para extraer las características de los objetos en los cuadros delimitadores para poder reconocerlos posteriormente.

3.5.2 DeepSORT

DeepSORT es una extensión de SORT. Se basa en los mismo principios que el anterior, tenemos un detector de objetos que nos da detecciones, el filtro de Kalman lo rastrea y completa y ayuda a rellenar los fallos en la detección, y el algoritmo húngaro resuelve el problema de asociación. Pero como ya se ha comentado anteriormente, el filtro de Kalman falla en determinados escenarios, así que para solucionar el problema los autores de DeepSORT añadieron una nueva métrica de distancia basada en la apariencia del objeto.

3.5.2.1 El vector de características de apariencia

La idea de obtener un vector que pueda describir todas las características de una imagen dada es bastante simple. Primero construimos un clasificador sobre nuestro conjunto de datos, lo entrenamos hasta que logre una precisión razonablemente buena y luego eliminamos la capa de clasificación final. Suponiendo una arquitectura clásica, nos quedaremos con una capa densa que produce un único vector de características, a la espera de ser clasificado. Ese vector de características se convierte en nuestro "descriptor de apariencia" del objeto.

Ahora la nueva métrica de distancia responde a:

$$D = \lambda \cdot D_k + (1 - \lambda) \cdot D_a \tag{24}$$

$$D_k(\vec{x}, \vec{y}) = \sqrt{(\vec{x} - \vec{y})^T \boldsymbol{C}^{-1} (\vec{x} - \vec{y})} \tag{25}$$

$$D_a(\vec{x}, \vec{y}) = cos\theta = \frac{\vec{x} \cdot \vec{y}}{\|\vec{x}\| \|\vec{y}\|} \tag{26}$$

donde D_k es la distancia de Mahalanobis, D_a es la distancia del coseno entre los vectores de apariencia y λ es un peso.

En consecuencia, el proceso es el mismo que en SORT, pero se añade este paso adicional para intentar solventar los problemas que sufría este último.

3.6 MEDIDA DE LA EFICIENCIA DE LOS RASTREADORES

Teniendo sendas implementaciones de los rastreadores y habiendo comprobado que funcionan, es hora de realizar las pruebas para determinar las cualidades de cada uno.

Para ello se han realizado medidas para el rendimiento mediante la observación de los fps en ejecución del conjunto detector-rastreador, y también se han realizado pruebas para medir la precisión de dicho conjunto sobre secuencias de imágenes con anotaciones proporcionadas recientemente por BDD, haciendo uso de la biblioteca *py-motmetrics* [67].

Los resultados de las pruebas se muestran en el Capítulo 4.

Capítulo 4

RESULTADOS

4.1 MÁQUINA DE BAJAS PRESTACIONES

Las pruebas se han realizado en la máquina "amaltea" con procesador i7-7700 de 8 núcleos a 3.6GHz, con 16Gb de memoria RAM y una GPU nVidia GTX 1050 de 2G de memoria.

Tras la ejecución del programa se han obtenido los siguientes resultados, para el tiempo de inferencia (IT), la precisión media (mAP) y la relación entre mAP e IT.

La Figura 4-1 representa el tiempo medio de inferencia para cada uno de los modelos probados. No aparecen todos los modelos debido a que muchos de ellos no se podían probar en esta máquina debido a falta de memoria (*Out Of Memory error*).

La Figura 4-2 representa la precisión media de cada modelo siguiendo el criterio expuesto en la Sección 3.3. Como se puede observar los modelos basados en FRCNN tienen una precisión mayor a costa de necesitar más tiempo para llevar a cabo la detección.

Para tener una visión genérica del coste rendimiento, se ha optado por realizar una gráfica con la relación entre mAP e IT que se puede ver en la Figura 4-3

Aunque también podemos mostrarla mediante una relación directa de los datos y es lo que se muestra en la Figura 4-4. Como se puede observar, YOLOv3 proporciona una precisión equivalente a la que dan los modelos basados en FRCNN pero con tiempos de inferencia de aproximadamente el 50%. Hay que tener muy en cuenta que los tiempos de detección están fuertemente ligados a la implementación hardware de la máquina en la que se ejecuten, por lo que no hay que darle demasiada importancia a los valores absolutos, si no a los valores relativos.

También hay que tener en cuenta que YOLOv3 necesita más memoria para trabajar que los modelos basados en SSD (de hecho al realizar las pruebas en nuestra máquina aparece un aviso sobre que el rendimiento podría mejorar si dispusiéramos de una cantidad de memoria mayor).

4.2 MÁQUINA DE ALTAS PRESTACIONES

Se realizaron las pruebas en la máquina "neptuno" que en el momento de la ejecución, poseía una GPU nVidia GTX 1080Ti, para medir, si existieran, las mejoras de rendimiento. Los resultados, son análogos en formato a los expuestos en el apartado anterior, solo que esta vez se han podido ejecutar más pruebas con más modelos y se pueden observar en las Figura 4-5, la Figura 4-6, la Figura 4-7 y la Figura 4-8.

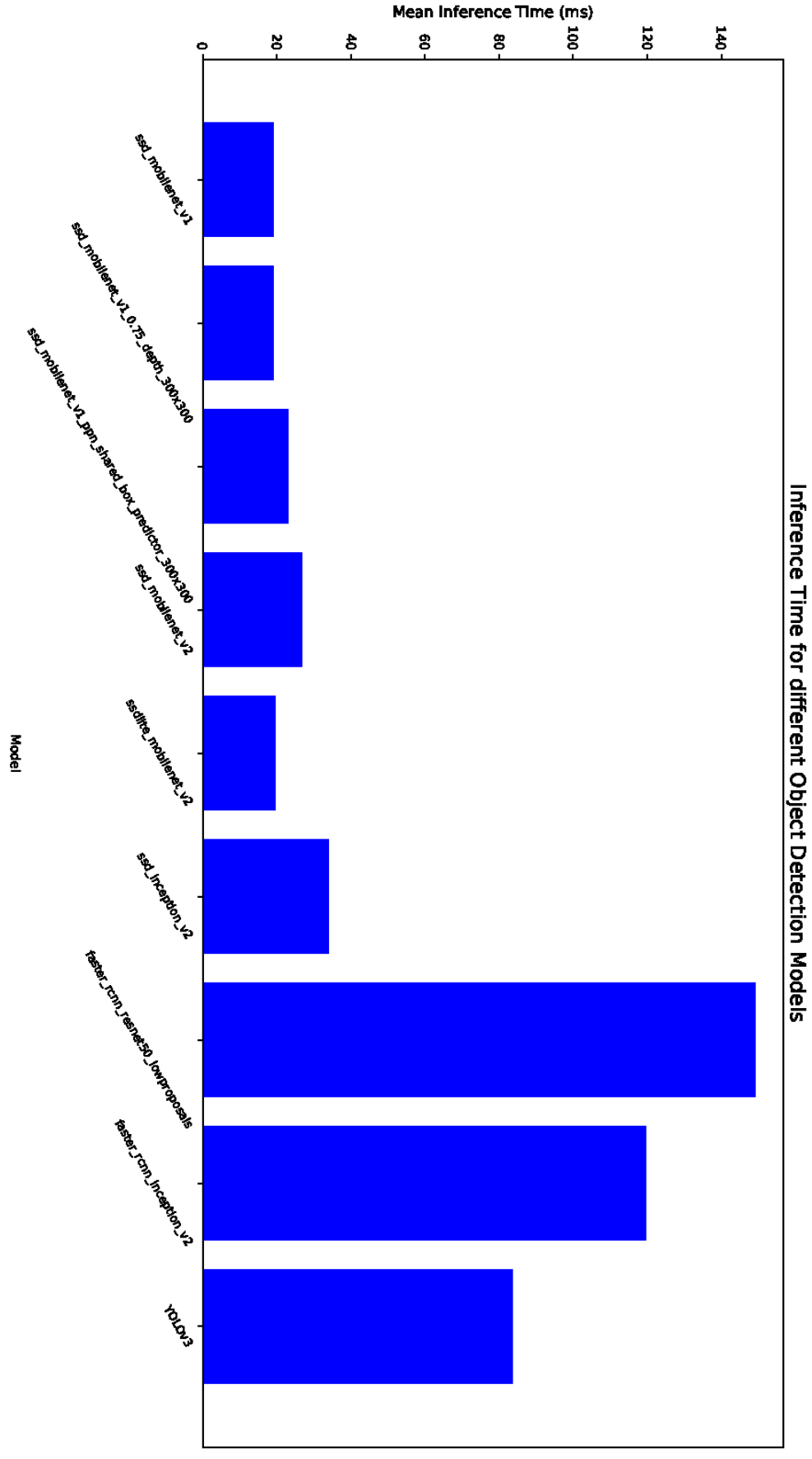

Figura 4-1. Tiempo de Inferencia para "amaltea".

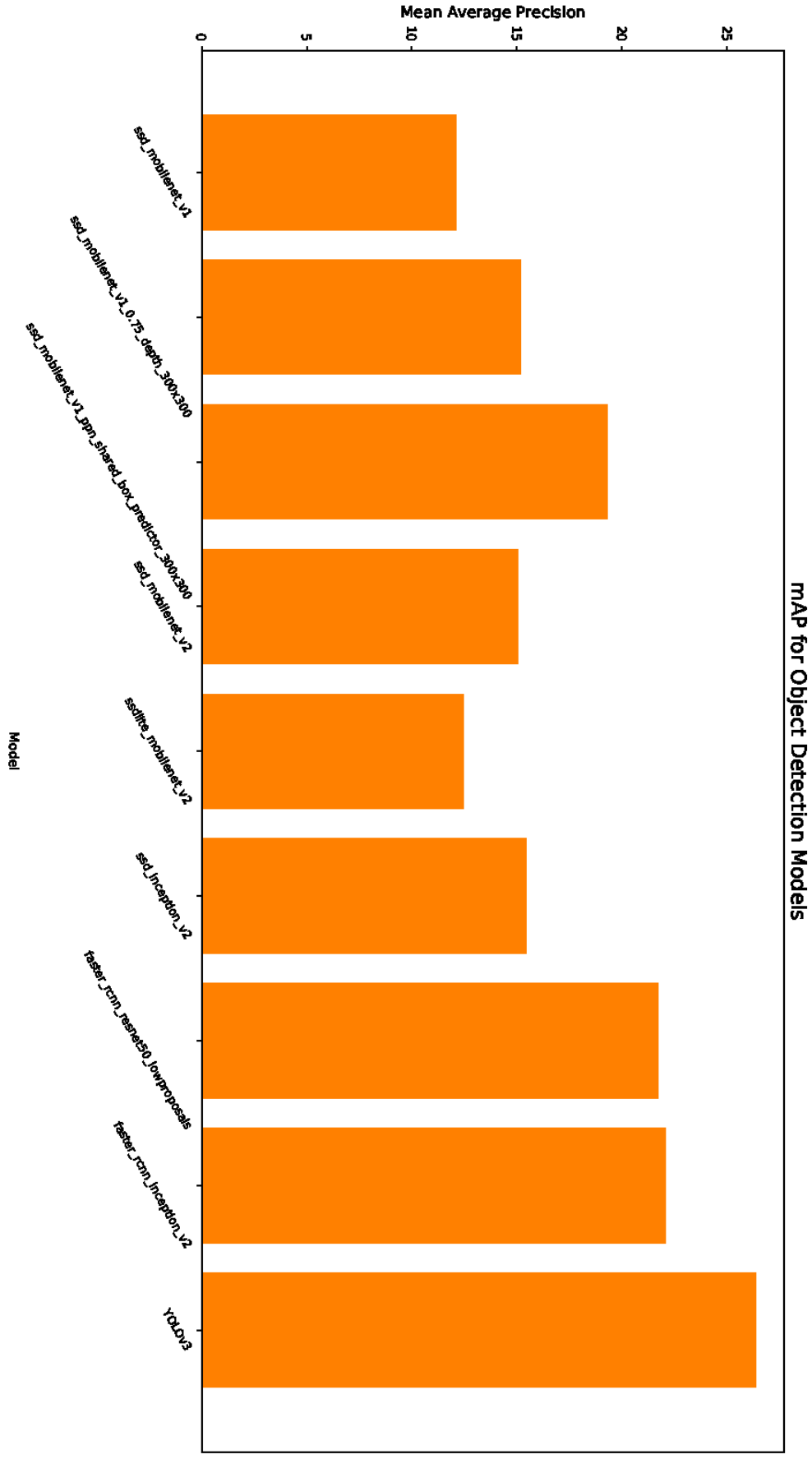

Figura 4-2. mAP para "amaltea".

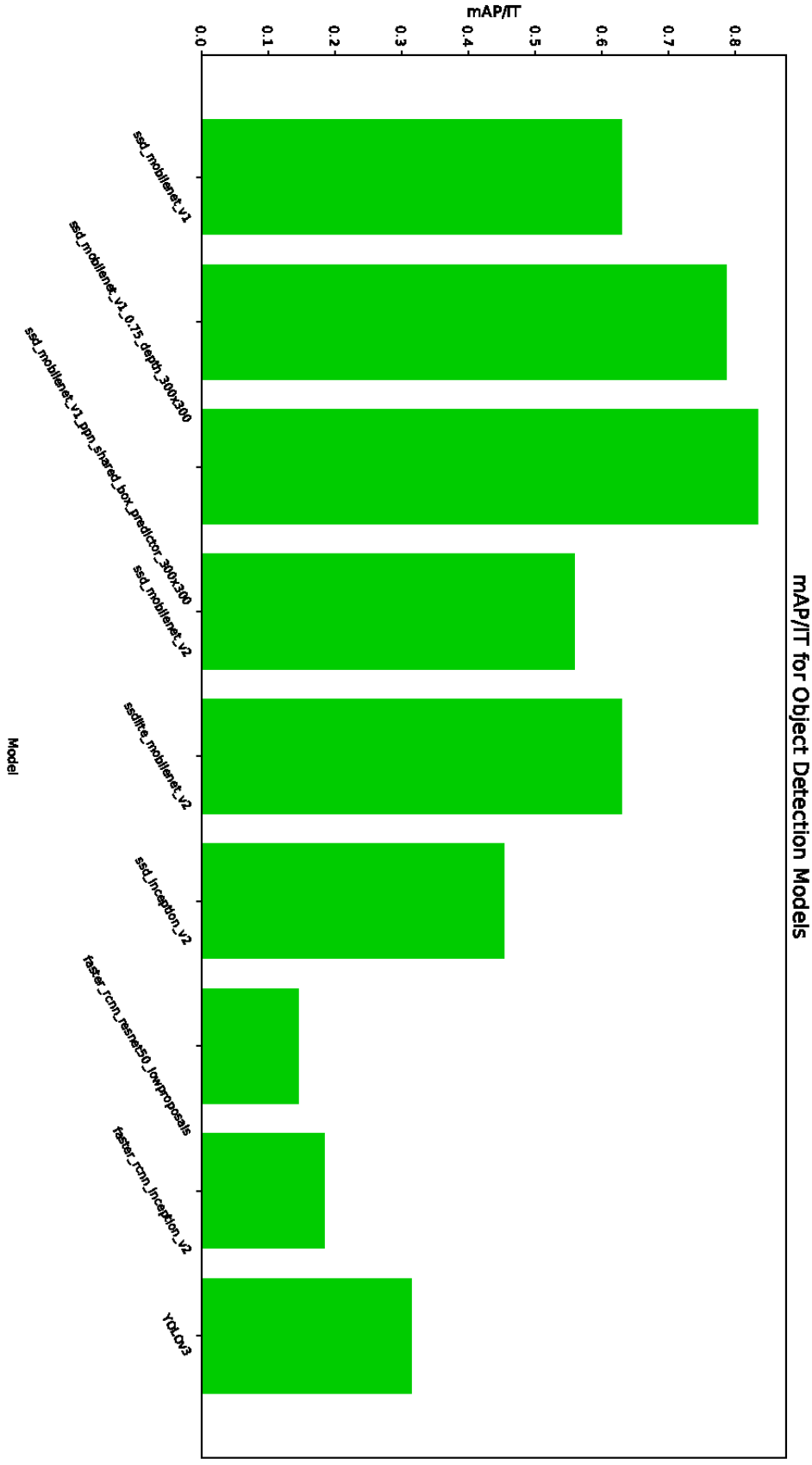

Figura 4-3. Relación mAP/IT para "amaltea".

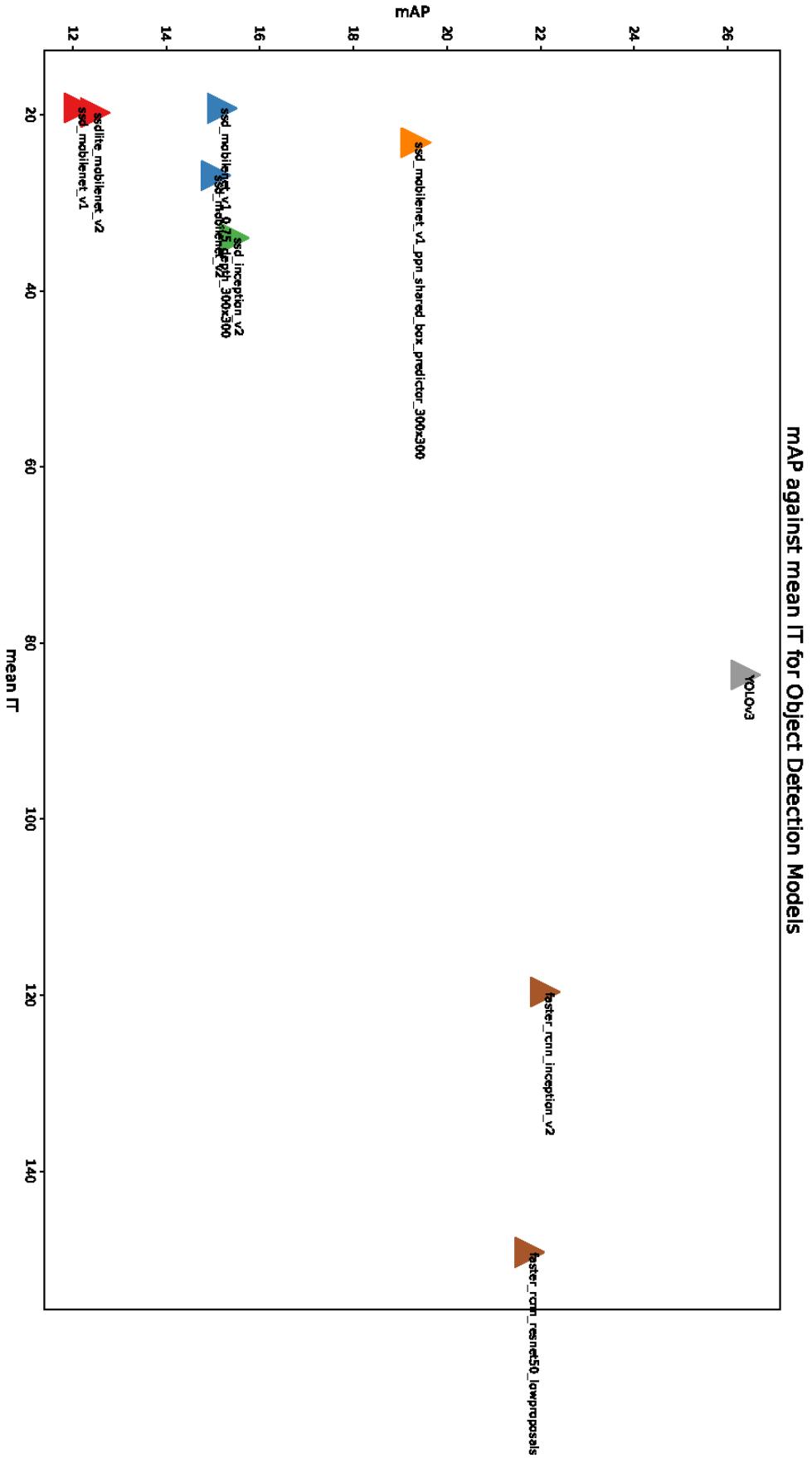

Figura 4-4. mAP vs IT para "amaltea".

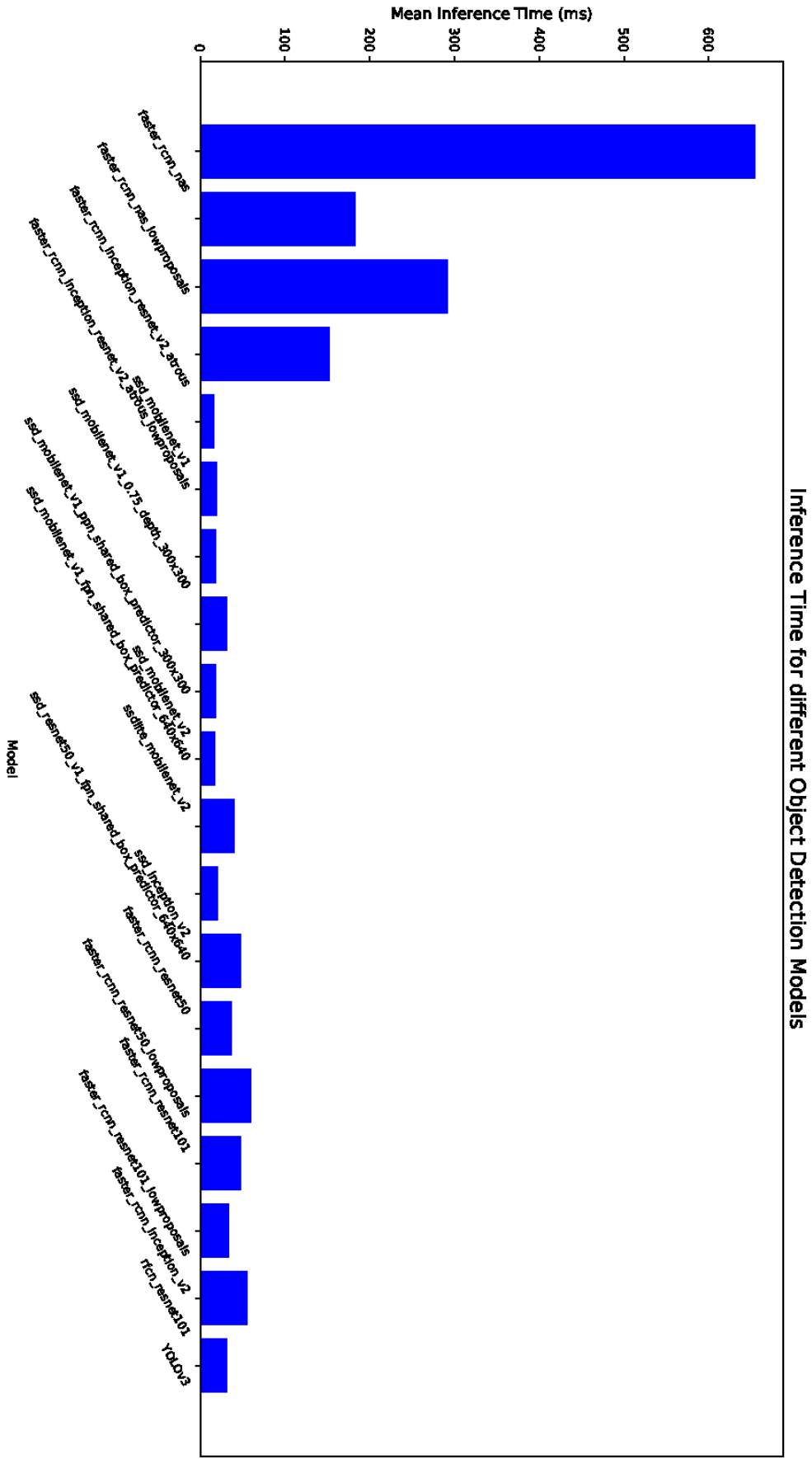

Figura 4-5. Inference Time para "neptuno".

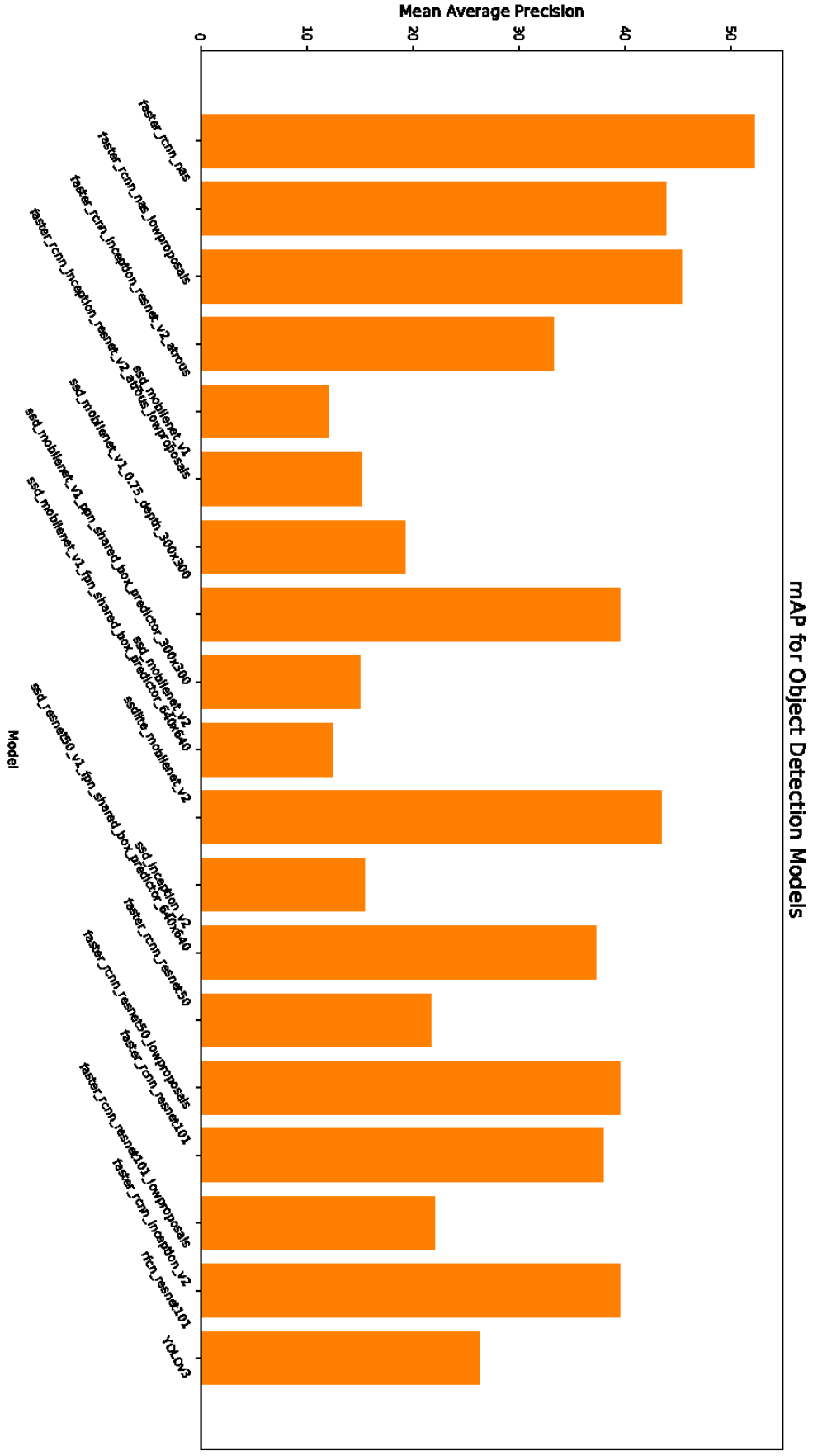

Figura 4-6. mAP para "neptuno".

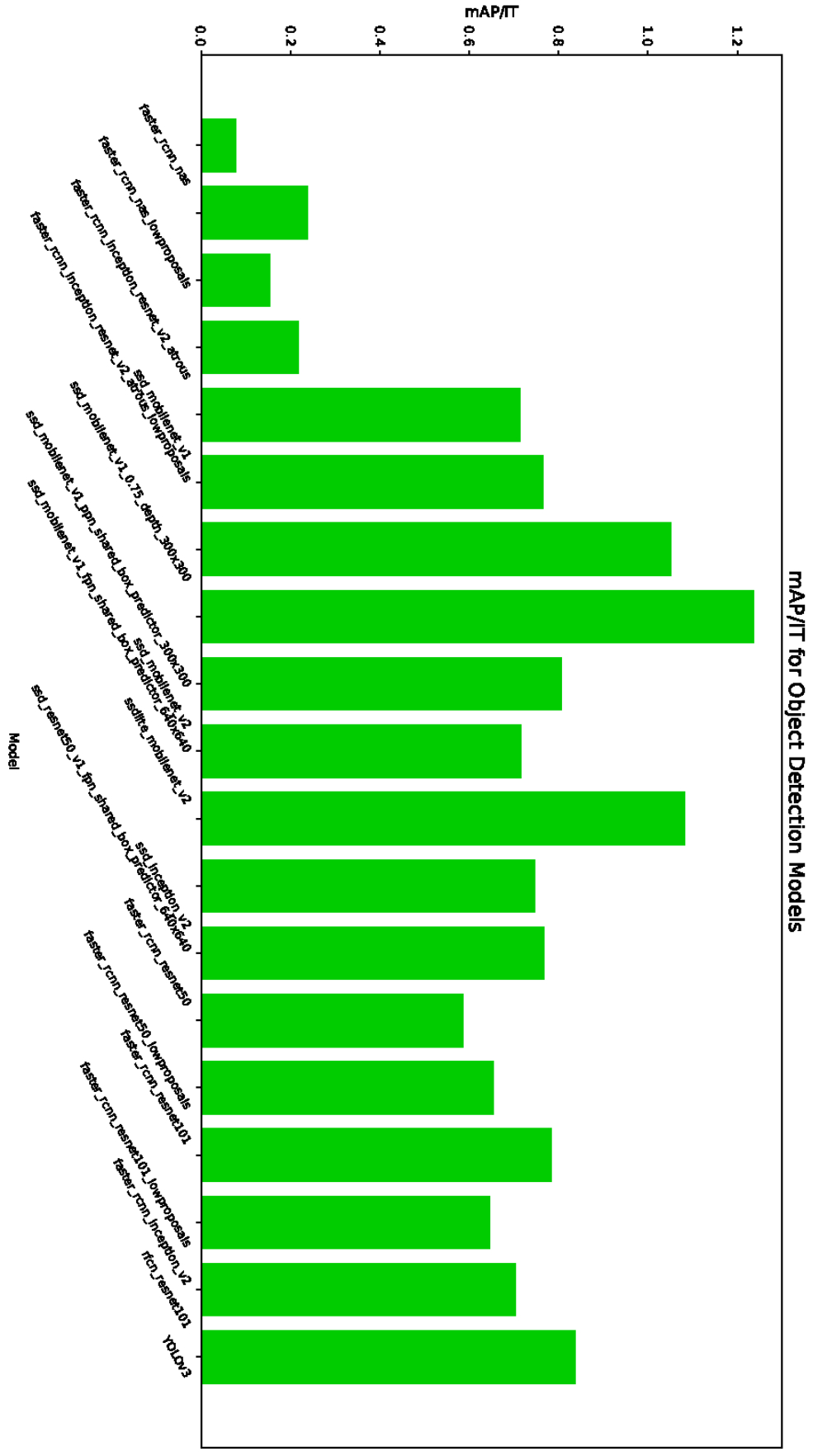

Figura 4-7. Relación mAP/IT para "neptuno".

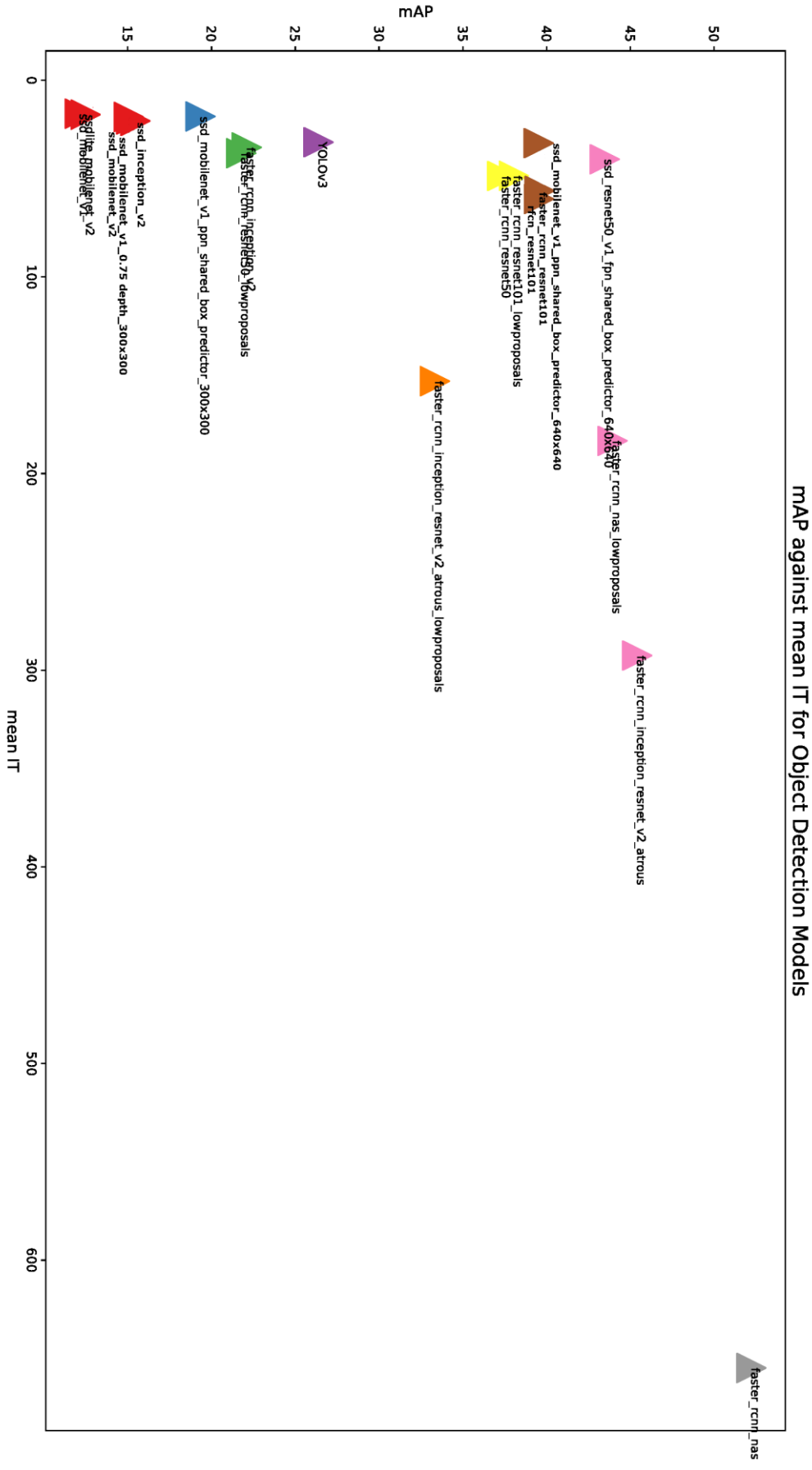

Figura 4-8. mAP vs IT para "neptuno".

Como se puede observar a la vista de los resultados, el hardware empleado afecta sensiblemente al tiempo de inferencia (no así al mAP que es agnóstica del hardware), además de permitir la ejecución de más modelos al disponer de más memoria gráfica. Esto es muy importante a la hora de determinar el sistema que se quiera implementar para realizar la detección.

4.3 RESULTADOS PARA LOS RASTREADORES

En cuanto al rastreo, se ha realizado la prueba, en la máquina de bajas prestaciones que es la que presumiblemente se ajustará más al sistema final, empleando los dos detectores entrenados, y los dos rastreadores, mediante la ejecución del programa sobre vídeos a 30 fps del *dataset*. El experimento se ha repetido 50 veces y los resultados obtenidos se han promediado (se ha obtenido una $\sigma^2 = 1.1$). Se recogen en la Tabla 4.3-1.

	No Tracker	SORT	DeepSORT
SSD	25 fps	22 fps	21 fps
YOLO	8 fps	6 fps	5 fps

Tabla 4.3-1 Velocidades de ejecución para la duplas detector-rastreador

De esta tabla se puede observar que implementar un algoritmo de rastreo, tiene un impacto directo en la velocidad, pero por otro lado, el impacto de DeepSORT sobre el rendimiento solo es ligeramente mayor al de SORT.

Para medir la precisión se hizo uso de la biblioteca para *python* "py-motmetrics", como ya se ha comentado en el apartado anterior. Se ha ejecutado cada pareja de detector-rastreador sobre 50 secuencias de imágenes del set de validación que proporciona BDD, con sus respectivas anotaciones y se han guardado los resultados siguiendo el formato que se expone en [41] que es el que emplea *motmetrics* para realizar las evaluaciones. Tras la ejecución del script de evaluación se obtiene la precisión sobre cada secuencia y la total promedio, que es la que se ha recogido en la Tabla 4.3-2.

	SORT	DeepSORT
SSD	25.4%	44.6%
YOLO	47.6%	62.1%

Tabla 4.3-2. Precisión promedio, para las duplas detector-rastreador

A la vista de los resultados podemos concluir que DeepSORT mejora mucho los resultados frente a predecesor, y como ya se ha mostrado anteriormente, con un coste pequeño sobre el rendimiento.

También se puede destacar que un detector más preciso puede obtener mejores resultados a pesar de usar un rastreador más simple, por lo que el éxito del rastreo depende en gran medida de la precisión de nuestro detector.

Capítulo 5
CONCLUSIONES Y LÍNEAS FUTURAS

5.1 CONCLUSIONES

A lo largo de este documento se han ido exponiendo las diferentes tecnologías que existen a día de hoy en el campo de la detección de objetos y el tracking con el objetivo de implementar un sistema que haciendo uso de ellas nos permitiera conseguir una detección capaz de identificar las incidencias en carretera y realizara su seguimiento en tiempo real.

Para poder elegir el sistema que mejor se ajustara a nuestras necesidades, se realizó un proceso de evaluación de los diferentes detectores siguiendo principalmente dos parámetros, la precisión y el tiempo de inferencia.

Tras realizar las pruebas se pudo constatar que si se desea una mayor precisión hay que recurrir a modelos que requieren mayor tiempo de procesamiento y que la capacidad de procesamiento de la GPU tiene un alto impacto en el tiempo de inferencia.

A la vista de estos resultados, podemos concluir que los modelos SSD son una buena opción cuando tenemos limitaciones en el hardware o restricciones elevadas de tiempo de ejecución, sacrificando precisión. Si no tenemos restricciones de tiempo o memoria, los algoritmos FRCNN nos proporcionarán la mejor precisión disponible. Y por último si tenemos suficiente memoria y debemos realizar las detecciones en tiempo real YOLO proporciona la mejor precisión dentro del abanico de detectores.

En cuanto a los rastreadores, podemos concluir que aunque DeepSORT introduce algo más de retardo en el proceso, lo suple con una mejor precisión a la hora de realizar el seguimiento a la par que proporciona mayor robustez frente a oclusiones y pérdidas de identificador. Así que parece razonable usarlo en vez de SORT.

Como siempre en el mundo de la ingeniería, hay que llegar a una solución de compromiso, y por eso disponer de información adicional siempre es muy necesario a la hora de tomar la decisión que mejor se ajuste a nuestros requisitos.

5.2 LÍNEAS FUTURAS

En el desarrollo de este Trabajo Fin de Grado se han tenido que tomar algunas decisiones que no han sido óptimas debido a limitaciones y dificultades a la hora de implementar los sistemas deseados.

Al llevar a cabo las pruebas para medir la precisión de los rastreadores, que no se pudieron hacer en un primer momento debido a que no se disponía de etiquetas para las secuencias de vídeo (más tarde Berkeley introdujo un set de secuencias con

anotaciones), se puso de manifiesto que nuestros detectores no estaban rindiendo correctamente. Lo más probable es que se deba a deficiencias en el proceso de entrenamiento, así que se hace necesaria una revisión, si se deseara implementarlos en un sistema funcional.

También sería recomendable adaptar el código a la nueva versión de Tensorflow 2.0, que salió en Septiembre del 2019 [68] y tiene ventajas como el *eager execution* que permite realizar operaciones sin necesidad de crear sesiones.

Además se podría explorar con la posibilidad de emplear el nuevo YOLOv4 [69] que fue publicado en Abril del 2020, y que es una mejora directa sobre su versión anterior.

Lo que está claro es que el campo de los detectores de objetos y seguimiento sigue mejorando día a día, y que aparecen nuevas propuestas constantemente, por lo que hay que estar siempre preparado para el futuro.

BIBLIOGRAFÍA

[1] «Tensorflow Object Detection API», *GitHub*. https://github.com/tensorflow/models/tree/master/research/object_detection (accedido may 04, 2020).

[2] «Berkeley DeepDrive». https://bdd-data.berkeley.edu/ (accedido jul. 08, 2020).

[3] A. Géron, *Hands-On Machine Learning with Scikit-Learn and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. O'Reilly Media, Inc., 2017.

[4] E. Alpaydin, *Introduction to Machine Learning*. MIT Press, 2020.

[5] I. Goodfellow, Y. Bengio, y A. Courville, *Deep Learning*. MIT Press, 2016.

[6] «Ciencia de datos», *Wikipedia, la enciclopedia libre*. jun. 27, 2020, Accedido: sep. 13, 2020. [En línea]. Disponible en: https://es.wikipedia.org/w/index.php?title=Ciencia_de_datos&oldid=127292864.

[7] «TensorFlow», *TensorFlow*. https://www.tensorflow.org/?hl=es (accedido abr. 26, 2020).

[8] R. Álvarez, «Tesla P40 y Tesla P4, las nuevas y potentes GPUs de Nvidia para inteligencia artificial», *Xataka*, sep. 13, 2016. https://www.xataka.com/componentes/tesla-p40-y-tesla-p4-las-nuevas-y-potentes-gpus-de-nvidia-para-inteligencia-artificial (accedido abr. 26, 2020).

[9] «Top Big Data Processing Frameworks», *KDnuggets*. https://www.kdnuggets.com/top-big-data-processing-frameworks.html/ (accedido abr. 26, 2020).

[10] «The Network Effect of Voice». https://www.linkedin.com/pulse/network-effect-voice-jerry-lu (accedido sep. 13, 2020).

[11] A. A. M. Al-Saffar, H. Tao, y M. A. Talab, «Review of deep convolution neural network in image classification», en *2017 International Conference on Radar, Antenna, Microwave, Electronics, and Telecommunications (ICRAMET)*, oct. 2017, pp. 26-31, doi: 10.1109/ICRAMET.2017.8253139.

[12] Y. Lecun, L. Bottou, Y. Bengio, y P. Haffner, «Gradient-Based Learning Applied to Document Recognition», *Proc. IEEE*, vol. 86, pp. 2278-2324, dic. 1998, doi: 10.1109/5.726791.

[13] E. Culurciello, «Neural Network Architectures», *Medium*, dic. 24, 2018. https://towardsdatascience.com/neural-network-architectures-156e5bad51ba (accedido abr. 27, 2020).

[14] «Kernel (image processing)», *Wikipedia*. abr. 16, 2020, Accedido: abr. 27, 2020. [En línea]. Disponible en: https://en.wikipedia.org/w/index.php?title=Kernel_(image_processing)&oldid=951378908.

[15] «CS231n Convolutional Neural Networks for Visual Recognition». https://cs231n.github.io/convolutional-networks/ (accedido abr. 27, 2020).

[16] D. Britz, «Understanding Convolutional Neural Networks for NLP», *WildML*, nov. 07, 2015. http://www.wildml.com/2015/11/understanding-convolutional-neural-networks-for-nlp/ (accedido abr. 27, 2020).

[17] S. Mallat, «Understanding Deep Convolutional Networks», *Philos. Trans. R. Soc. Math. Phys. Eng. Sci.*, vol. 374, n.º 2065, p. 20150203, abr. 2016, doi: 10.1098/rsta.2015.0203.

[18] P. Piccinini, A. Prati, y R. Cucchiara, «Real-time object detection and localization with SIFT-based clustering», *Image Vis. Comput.*, vol. 30, n.º 8, pp. 573-587, ago. 2012, doi: 10.1016/j.imavis.2012.06.004.

[19] K. Mizuno, Y. Terachi, K. Takagi, S. Izumi, H. Kawaguchi, y M. Yoshimoto, «Architectural Study of HOG Feature Extraction Processor for Real-Time Object Detection», en *2012 IEEE Workshop on Signal Processing Systems*, Quebec City, QC, oct. 2012, pp. 197-202, doi: 10.1109/SiPS.2012.57.

[20] «Jaccard Index», *DeepAI*, may 17, 2019. https://deepai.org/machine-learning-glossary-and-terms/jaccard-index (accedido abr. 29, 2020).

[21] J. Hui, «mAP (mean Average Precision) for Object Detection», *Medium*, abr. 03, 2019. https://medium.com/@jonathan_hui/map-mean-average-precision-for-object-detection-45c121a31173 (accedido may 06, 2020).

[22] S. Ren, K. He, R. Girshick, y J. Sun, «Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks», *ArXiv150601497 Cs*, ene. 2016, Accedido: abr. 28, 2020. [En línea]. Disponible en: http://arxiv.org/abs/1506.01497.

[23] R. Girshick, J. Donahue, T. Darrell, y J. Malik, «Rich feature hierarchies for accurate object detection and semantic segmentation», *ArXiv13112524 Cs*, oct. 2014, Accedido: abr. 28, 2020. [En línea]. Disponible en: http://arxiv.org/abs/1311.2524.

[24] R. Girshick, «Fast R-CNN», *ArXiv150408083 Cs*, sep. 2015, Accedido: abr. 28, 2020. [En línea]. Disponible en: http://arxiv.org/abs/1504.08083.

[25] W. Liu *et al.*, «SSD: Single Shot MultiBox Detector», *ArXiv151202325 Cs*, vol. 9905, pp. 21-37, 2016, doi: 10.1007/978-3-319-46448-0_2.

[26] «The PASCAL Visual Object Classes Homepage». http://host.robots.ox.ac.uk/pascal/VOC/ (accedido abr. 29, 2020).

[27] «COCO - Common Objects in Context». http://cocodataset.org/#home (accedido abr. 29, 2020).

[28] J. Redmon, S. Divvala, R. Girshick, y A. Farhadi, «You Only Look Once: Unified, Real-Time Object Detection», *ArXiv150602640 Cs*, may 2016, Accedido: abr. 29, 2020. [En línea]. Disponible en: http://arxiv.org/abs/1506.02640.

[29] Li Zhang, Yuan Li, y R. Nevatia, «Global data association for multi-object tracking using network flows», en *2008 IEEE Conference on Computer Vision and Pattern Recognition*, jun. 2008, pp. 1-8, doi: 10.1109/CVPR.2008.4587584.

[30] H. Pirsiavash, D. Ramanan, y C. C. Fowlkes, «Globally-optimal greedy algorithms for tracking a variable number of objects», en *CVPR 2011*, jun. 2011, pp. 1201-1208, doi: 10.1109/CVPR.2011.5995604.

[31] J. Berclaz, F. Fleuret, E. Turetken, y P. Fua, «Multiple Object Tracking Using K-Shortest Paths Optimization», *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 33, n.º 9, pp. 1806-1819, sep. 2011, doi: 10.1109/TPAMI.2011.21.

[32] B. Yang y R. Nevatia, «An online learned CRF model for multi-target tracking», en *2012 IEEE Conference on Computer Vision and Pattern Recognition*, jun. 2012, pp. 2034-2041, doi: 10.1109/CVPR.2012.6247907.

[33] B. Yang y R. Nevatia, «Multi-target tracking by online learning of non-linear motion patterns and robust appearance models», en *2012 IEEE Conference on Computer Vision and Pattern Recognition*, jun. 2012, pp. 1918-1925, doi: 10.1109/CVPR.2012.6247892.

[34] A. Milan, K. Schindler, y S. Roth, «Detection- and Trajectory-Level Exclusion in Multiple Object Tracking», en *2013 IEEE Conference on Computer Vision and Pattern Recognition*, jun. 2013, pp. 3682-3689, doi: 10.1109/CVPR.2013.472.

[35] G. Welch, «An Introduction to the Kalman Filter», p. 16, 1997.

[36] D. Reid, «An algorithm for tracking multiple targets», *IEEE Trans. Autom. Control*, vol. 24, n.º 6, pp. 843-854, dic. 1979, doi: 10.1109/TAC.1979.1102177.

[37] T. Fortmann, Y. Bar-Shalom, y M. Scheffe, «Sonar tracking of multiple targets using joint probabilistic data association», *IEEE J. Ocean. Eng.*, vol. 8, n.º 3, pp. 173-184, jul. 1983, doi: 10.1109/JOE.1983.1145560.

[38] C. Kim, F. Li, A. Ciptadi, y J. M. Rehg, «Multiple Hypothesis Tracking Revisited», en *2015 IEEE International Conference on Computer Vision (ICCV)*, dic. 2015, pp. 4696-4704, doi: 10.1109/ICCV.2015.533.

[39] S. H. Rezatofighi, A. Milan, Z. Zhang, Q. Shi, A. Dick, y I. Reid, «Joint Probabilistic Data Association Revisited», en *2015 IEEE International Conference on Computer Vision (ICCV)*, dic. 2015, pp. 3047-3055, doi: 10.1109/ICCV.2015.349.

[40] A. Bewley, Z. Ge, L. Ott, F. Ramos, y B. Upcroft, «Simple Online and Realtime Tracking», *2016 IEEE Int. Conf. Image Process. ICIP*, pp. 3464-3468, sep. 2016, doi: 10.1109/ICIP.2016.7533003.

[41] «MOT Challenge». https://motchallenge.net/ (accedido sep. 16, 2020).

[42] G. Ning, Z. Zhang, C. Huang, Z. He, X. Ren, y H. Wang, «Spatially Supervised Recurrent Convolutional Neural Networks for Visual Object Tracking», *ArXiv160705781 Cs*, jul. 2016, Accedido: jun. 23, 2020. [En línea]. Disponible en: http://arxiv.org/abs/1607.05781.

[43] S. Hochreiter y J. Schmidhuber, «Long Short-term Memory», *Neural Comput.*, vol. 9, pp. 1735-80, dic. 1997, doi: 10.1162/neco.1997.9.8.1735.

[44] Q. Wang, L. Zhang, L. Bertinetto, W. Hu, y P. H. S. Torr, «Fast Online Object Tracking and Segmentation: A Unifying Approach», *ArXiv181205050 Cs*, may 2019, Accedido: jun. 23, 2020. [En línea]. Disponible en: http://arxiv.org/abs/1812.05050.

[45] D. Chicco, «Siamese Neural Networks: An Overview», en *Artificial Neural Networks*, H. Cartwright, Ed. New York, NY: Springer US, 2020, pp. 73-94.

[46] N. Wojke, A. Bewley, y D. Paulus, «Simple Online and Realtime Tracking with a Deep Association Metric», *ArXiv170307402 Cs*, mar. 2017, Accedido: jun. 23, 2020. [En línea]. Disponible en: http://arxiv.org/abs/1703.07402.

[47] P. Voigtlaender *et al.*, «MOTS: Multi-Object Tracking and Segmentation», *ArXiv190203604 Cs*, abr. 2019, Accedido: jun. 23, 2020. [En línea]. Disponible en: http://arxiv.org/abs/1902.03604.

[48] K. He, G. Gkioxari, P. Dollár, y R. Girshick, «Mask R-CNN», *ArXiv170306870 Cs*, ene. 2018, Accedido: sep. 16, 2020. [En línea]. Disponible en: http://arxiv.org/abs/1703.06870.

[49] P. Bergmann, T. Meinhardt, y L. Leal-Taixe, «Tracking without bells and whistles», *ArXiv190305625 Cs*, ago. 2019, Accedido: jun. 23, 2020. [En línea]. Disponible en: http://arxiv.org/abs/1903.05625.

[50] Z. Wang, L. Zheng, Y. Liu, y S. Wang, «Towards Real-Time Multi-Object Tracking», *ArXiv190912605 Cs*, sep. 2019, Accedido: jun. 23, 2020. [En línea]. Disponible en: http://arxiv.org/abs/1909.12605.

[51] F. Yu *et al.*, «BDD100K: A Diverse Driving Dataset for Heterogeneous Multitask Learning», *ArXiv180504687 Cs*, abr. 2020, Accedido: may 06, 2020. [En línea]. Disponible en: http://arxiv.org/abs/1805.04687.

[52] «JSON». https://www.json.org/json-en.html (accedido may 06, 2020).

[53] «ImageNet». http://www.image-net.org/ (accedido abr. 28, 2020).

[54] M. D. Zeiler y R. Fergus, «Visualizing and Understanding Convolutional Networks», *ArXiv13112901 Cs*, nov. 2013, Accedido: abr. 29, 2020. [En línea]. Disponible en: http://arxiv.org/abs/1311.2901.

[55] K. Simonyan y A. Zisserman, «Very Deep Convolutional Networks for Large-Scale Image Recognition», *ArXiv14091556 Cs*, abr. 2015, Accedido: abr. 29, 2020. [En línea]. Disponible en: http://arxiv.org/abs/1409.1556.

[56] A. G. Howard *et al.*, «MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications», *ArXiv170404861 Cs*, abr. 2017, Accedido: abr. 29, 2020. [En línea]. Disponible en: http://arxiv.org/abs/1704.04861.

[57] K. He, X. Zhang, S. Ren, y J. Sun, «Deep Residual Learning for Image Recognition», *ArXiv151203385 Cs*, dic. 2015, Accedido: abr. 29, 2020. [En línea]. Disponible en: http://arxiv.org/abs/1512.03385.

[58] G. Huang, Z. Liu, L. van der Maaten, y K. Q. Weinberger, «Densely Connected Convolutional Networks», *ArXiv160806993 Cs*, ene. 2018, Accedido: abr. 29, 2020. [En línea]. Disponible en: http://arxiv.org/abs/1608.06993.

[59] «Intersection over Union (IoU) for object detection», *PyImageSearch*, nov. 07, 2016. https://www.pyimagesearch.com/2016/11/07/intersection-over-union-iou-for-object-detection/ (accedido abr. 29, 2020).

[60] «A Friendly Introduction to Cross-Entropy Loss». https://rdipietro.github.io/friendly-intro-to-cross-entropy-loss/ (accedido abr. 29, 2020).

[61] «Luminoth: Open source toolkit for Computer Vision». https://luminoth.ai/ (accedido sep. 14, 2020).

[62] C. Szegedy, S. Reed, D. Erhan, D. Anguelov, y S. Ioffe, «Scalable, High-Quality Object Detection», *ArXiv14121441 Cs*, dic. 2015, Accedido: abr. 29, 2020. [En línea]. Disponible en: http://arxiv.org/abs/1412.1441.

[63] C. Szegedy *et al.*, «Going Deeper with Convolutions», *ArXiv14094842 Cs*, sep. 2014, Accedido: abr. 29, 2020. [En línea]. Disponible en: http://arxiv.org/abs/1409.4842.

[64] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, y L.-C. Chen, «MobileNetV2: Inverted Residuals and Linear Bottlenecks», *ArXiv180104381 Cs*, mar. 2019, Accedido: sep. 21, 2020. [En línea]. Disponible en: http://arxiv.org/abs/1801.04381.

[65] J. Redmon y A. Farhadi, «YOLOv3: An Incremental Improvement», *ArXiv180402767 Cs*, abr. 2018, Accedido: sep. 21, 2020. [En línea]. Disponible en: http://arxiv.org/abs/1804.02767.

[66] S. Haykin, *Kalman Filtering and Neural Networks*. John Wiley & Sons, 2004.

[67] C. Heindl, *cheind/py-motmetrics*. 2020.

[68] «TensorFlow 2.0 is now available!» https://blog.tensorflow.org/2019/09/tensorflow-20-is-now-available.html (accedido sep. 16, 2020).

[69] A. Bochkovskiy, C.-Y. Wang, y H.-Y. M. Liao, «YOLOv4: Optimal Speed and Accuracy of Object Detection», *ArXiv200410934 Cs Eess*, abr. 2020, Accedido: sep. 16, 2020. [En línea]. Disponible en: http://arxiv.org/abs/2004.10934.

[70] «COCO - Common Objects in Context». http://cocodataset.org/#format-data (accedido may 06, 2020).

[71] «ucbdrive/bdd100k», *GitHub*. https://github.com/ucbdrive/bdd100k (accedido may 06, 2020).

[72] «Scalabel». https://www.scalabel.ai/ (accedido may 06, 2020).

[73] Wizyoung, *wizyoung/YOLOv3_TensorFlow*. 2020.

Apéndice A

FORMATO DE LOS DATASETS EMPLEADOS

A.1 COCO

Las anotaciones de COCO para detección de objetos usan el siguiente formato [70]:

El *dataset* de COCO emplea ficheros JSON y es una colección de "*info*", "*licenses*", "*images*", "*annotations*", "*categories*", y "*segment_info*":

```
{
    "info": {...},
    "licenses": [...],
    "images": [...],
    "annotations": [...],
    "categories": [...], <-- Not in Captions annotations
    "segment_info": [...] <-- Only in Panoptic annotations
}
```

Info:

La sección "*info*" contiene información de alto nivel sobre el conjunto de datos.

```
"info": {
    "description": "COCO 2017 Dataset",
    "url": "http://cocodataset.org",
    "version": "1.0",
    "year": 2017,
    "contributor": "COCO Consortium",
    "date_created": "2017/09/01"
}
```

Licenses:

La sección "*licenses*" contiene una lista de licencias que se aplican a las imágenes en el conjunto de datos.

```
"licenses": [
    {
        "url": "http://creativecommons.org/licenses/by-nc-sa/2.0/",
        "id": 1,
        "name": "Attribution-NonCommercial-ShareAlike License"
    },
    {
        "url": "http://creativecommons.org/licenses/by-nc/2.0/",
        "id": 2,
        "name": "Attribution-NonCommercial License"
    },
    ...
]
```

Images:

La sección "*images*" contiene la lista completa de imágenes en el conjunto de datos. No hay etiquetas, cuadros delimitadores ni segmentaciones especificadas en esta parte, es simplemente una lista de imágenes e información sobre cada una. Los campos "*coco_url*", "*flickr_url*" y "*date_captured*" son solo para referencia.

Los identificadores de imagen son únicos (dentro del conjunto de imágenes), pero no necesariamente tienen que coincidir con el nombre del fichero.

```
"images": [
    {
        "license": 4,
        "file_name": "000000397133.jpg",
        "coco_url":
"http://images.cocodataset.org/val2017/000000397133.jpg",
        "height": 427,
        "width": 640,
        "date_captured": "2013-11-14 17:02:52",
        "flickr_url":
"http://farm7.staticflickr.com/6116/6255196340_da26cf2c9e_z.jpg",
        "id": 397133
    },
    {
        "license": 1,
        "file_name": "000000037777.jpg",
        "coco_url":
"http://images.cocodataset.org/val2017/000000037777.jpg",
        "height": 230,
        "width": 352,
        "date_captured": "2013-11-14 20:55:31",
        "flickr_url":
"http://farm9.staticflickr.com/8429/7839199426_f6d48aa585_z.jpg",
        "id": 37777
    },
    ...
]
```

Categories:

La sección "*categories*" contiene una lista de categorías (por ejemplo, perro, barco) y cada una de ellas pertenece a una supercategoría (por ejemplo, animal, vehículo). El conjunto de datos original de COCO contiene 90 categorías. Cada id de categoría debe ser único (entre el resto de las categorías).

```
"categories": [
    {"supercategory": "person","id": 1,"name": "person"},
    {"supercategory": "vehicle","id": 2,"name": "bicycle"},
    {"supercategory": "vehicle","id": 3,"name": "car"},
    {"supercategory": "vehicle","id": 4,"name": "motorcycle"},
    {"supercategory": "vehicle","id": 5,"name": "airplane"},
    ...
    {"supercategory": "indoor","id": 89,"name": "hair drier"},
    {"supercategory": "indoor","id": 90,"name": "toothbrush"}
]
```

Annotations:

La sección "*annotations*" contiene una lista de cada anotación de objeto individual de cada una de las imágenes dentro del conjunto de datos. Por ejemplo, si hay 64 bicicletas repartidas en 100 imágenes, habrá 64 anotaciones de bicicleta. A menudo habrá múltiples instancias de un mismo objeto en una imagen.

"*Is Crowd*" especifica si la segmentación es para un solo objeto o para un conglomerado de objetos.

La identificación de la imagen corresponde a una imagen específica en el conjunto de datos.

El formato del "*bounding box*" COCO es [posición superior izquierda x, posición superior izquierda y, ancho, altura].

La identificación de la categoría corresponde a una única categoría especificada en la sección de categorías.

Cada anotación también tiene un ID (único para todas las demás anotaciones en el conjunto de datos).

```
"annotations": [
    {
        "segmentation":
[[510.66,423.01,511.72,420.03,...,510.45,423.01]],
        "area": 702.1057499999998,
        "iscrowd": 0,
        "image_id": 289343,
        "bbox": [473.07,395.93,38.65,28.67],
        "category_id": 18,
        "id": 1768
    },
    ...
]
```

Al trabajar con los ficheros JSON en Python, la información que contienen se transforma en listas y diccionarios, lo cual es muy importante a la hora de manipular los datos.

A.2 BERKELEY

Las anotaciones de BDD100k usan el siguiente formato [71]:

```
name: string
url: string
videoName: string (optional)
attributes:
    weather:"rainy|snowy|clear|overcast|undefined|partly cloudy|foggy"
    scene:    "tunnel|residential|parking    lot|undefined|city    street|gas
stations|highway|"
```

```
    timeofday: "daytime|night|dawn/dusk|undefined"
intrinsics
    focal: [x, y]
    center: [x, y]
    nearClip:
extrinsics
    location
    rotation
timestamp: int64 (epoch time ms)
frameIndex: int (optional, frame index in this video)
labels [ ]:
    id: int32
    category: string (classification)
    manualShape: boolean (whether the shape of the label is created or
modified manually)
    manualAttributes: boolean (whether the attribute of the label is created
or modified manually)
    attributes:
        occluded: boolean
        truncated: boolean
        trafficLightColor: "red|green|yellow|none"
        areaType: "direct | alternative" (for driving area)
        laneDirection: "parallel|vertical" (for lanes)
        laneStyle: "solid | dashed" (for lanes)
        laneTypes: (for lanes)
    box2d:
       x1: float
       y1: float
       x2: float
       y2: float
    box3d:
       alpha: (observation angle if there is a 2D view)
       orientation: (3D orientation of the bounding box, used for 3D point
cloud annotation)
       location: (3D point, x, y, z, center of the box)
       dimension: (3D point, height, width, length)
   poly2d: an array of objects, with the structure
       vertices: [][]float (list of 2-tuples [x, y])
       types: string (each character corresponds to the type of the vertex
with the same index in vertices. 'L' for vertex and 'C' for control point of a
bezier curve.
       closed: boolean (closed for polygon and otherwise for path)
```

Cada entrada corresponde a una única imagen que contienen una lista con cada una de las etiquetas (labels) correspondientes a cada objetos con sus respectivos campos dependiendo del tipo de etiqueta. Cabe destacar que este es el formato genérico que emplea su aplicación de etiquetado (Scalabel [72]) y que no todos los campos estarán presentes en cada anotación. En nuestro caso nos interesan las etiquetas que contienen anotaciones del tipo "*box2d*" que contienen las coordenadas de la "*bounding box*" con el formato [coordenada x mínima, coordenada y mínima, coordenada x máxima, coordenada y máxima].

Apéndice B

BENCHMARK DE RENDIMIENTO

En este apéndice explicaremos la estructura y el funcionamiento de los scripts que realizan el *benchmark*.

B.1 ESTRUCTURA Y FUNCIONAMIENTO

El programa está compuesto por los siguientes scripts.

convert_detections.py:

Es un script simple que se encarga de transformar las detecciones a un formato común sobre el que realizar la evaluación.

`convert_detections.py` transforma las detecciones en un formato apropiado para realizar la evaluación. Realiza un cambio en el formato para facilitar el proceso de evaluación.

generate_test_annt.py:

Este script genera ficheros JSON con un número limitado de imágenes de un *dataset* que se emplearán en la detección y evaluación de los modelos.

Tiene sendas variables que se pueden modificar para facilitar la ejecución. Puesto que se ha trabajado a caballo entre dos SO (Windows y Ubuntu) posee una variable para seleccionar el entorno de trabajo, y otra para elegir el *dataset* (BDD o COCO). Por último define una variable para determinar el número de imágenes que se incluirán en los ficheros de anotaciones.

`generate_test_annt.py` transforma las anotaciones de los ficheros fuente de anotaciones de los *datasets* y genera ficheros JSON con un número reducido de imágenes (el cual se puede configurar) sobre las que se realizaran las pruebas. Hay que ejecutar este script primero señalando dónde se encuentran los ficheros fuente de las anotaciones junto con las imágenes a las que hacen referencia.

model_list.py:

Este fichero (en todas sus versiones) alberga una lista con los nombres de los ficheros que contienen los modelos así como los nombres de dichos modelos.

`model_list.py y model_list_selected.py` son ficheros que se usan para cargar las listas que contienen los nombres de todos los modelos proporcionados por el equipo de TODAPI, y los que se usan en la prueba respectivamente.

model.py

Este fichero contiene la definición del modelo de YOLO para tensorflow, donde se especifican las capas y los hiperparámetros que lo componen.

`model.py` contiene la implementación base de YOLOv3 en Tensorflow.

La carpeta data contiene los parámetros de la red de YOLOv3, los pesos, los nombres de las clases y los anchors.

convert_weights.py

Este script transforma los parámetros del modelo entrenado sobre el set de COCO de YOLO disponible en la página web principal al formato adecuado para pasárselos al modelo implementado en `model.py`. Basta descargar el fichero yolov3.weights, copiarlo a la carpeta /data/darknet_weights y ejecutar el script.

`convert_weight.py` se encarga de traducir los pesos del modelo desde Darknet a Tensorflow.

utils

Esta carpeta contiene scripts auxiliares, como por ejemplo dibujar las cajas sobre las imágenes.

La carpeta utils contiene varias funciones auxiliares de la implementación de YOLO.

bench_full_test.py:

bench_full_test.py que es el script principal y es el encargado de realizar la evaluación de rendimiento.

La estructura se compone de diferentes bloques. El primero representa la carga de los módulos necesarios para la ejecución del programa.

Después se encuentran las funciones auxiliares para realizar diferentes tareas, como la inferencia o la carga de las imágenes en matrices.

A continuación aparecen las variables, en concreto el directorio donde se encuentran las imágenes, las anotaciones y los modelos. En principio, el usuario sólo necesitará modificar el directorio donde se encuentran las imágenes y los modelos.

El programa descarga los ficheros con los modelos de la página del repositorio, y los guarda en la carpeta "*models*" en el directorio especificado por la variable "*models_path*".

Tras cargar el modelo y preparar las variables necesarias para la detección, se ejecuta la función de inferencia y se mide el tiempo necesario para llevarla a cabo. Se realizan

varias repeticiones (el número de iteraciones se puede cambiar) para obtener una media con el tiempo de inferencia. Toda esta información se guarda en *arrays* para su posterior representación.

Después, calcula la precisión comparando los resultados con el *ground_truth* generado previamente mediante el script `generate_test_annt.py`.

Tras la ejecución genera ficheros de texto con los tiempos que ha tomado llevar a cabo la detección y el "*mAP*" junto con el nombre del modelo, así como las imágenes con los objetos detectados.

B.2 PARA LA EVALUACIÓN DE YOLOV3

Para evaluar el rendimiento de YOLOv3 no podemos hacer uso del pipeline que emplea el API ya que no tiene implementado dicho algoritmo en sus componentes. Una posibilidad sería adaptar e implementar, siguiendo sus estructuras y formatos, los ficheros necesarios para su implementación dentro del API. Una segunda posibilidad es emplear una de las muchas implementaciones de YOLOv3 en Tensorflow existentes. Se ha optado por esta segunda opción para reducir el tiempo de desarrollo, aunque se está estudiando la posibilidad de implementarlo empleando la primera opción.

El programa se basa en la implementación de YOLOv3 sobre Tensorflow del usuario de *github* **wizyoung** [73].

Se ha modificado el script `test_single_image.py` para crear un script que realice una detección lo más parecida a la que se propone para el TODAPI. Dicho script se bautizó con el nombre `Bench_yolo_coco.py`. Puesto que se va a realizar la prueba sobre el *dataset* de COCO no hace falta más que descargarse el fichero de pesos de la página oficial de YOLO y ejecutar el script `convert_weights.py` para obtener los parámetros del modelo entrenado.

Posteriormente se ha incorporado el código al programa principal, y se han adaptado las salidas para que sean coherentes y así poder realizar comparaciones.

Apéndice C

FICHERO DE CONFIGURACIÓN

En este Apéndice se incluye el contenido del fichero que se empleó para la configuración del entrenamiento del detector SSD Mobilenetv2 en el *dataset* de Berkeley.

C.1 MODELO SSD MOBILENET V2

```
# SSD with Mobilenet v2 configuration for BDD Dataset.
# Users should configure the fine_tune_checkpoint field in the train config
as
# well as the label_map_path and input_path fields in the train_input_reader
and
# eval_input_reader. Search for "PATH_TO_BE_CONFIGURED" to find the fields
that
# should be configured.

model {
  ssd {
#    num_classes: 90
    num_classes: 10
    box_coder {
      faster_rcnn_box_coder {
        y_scale: 10.0
        x_scale: 10.0
        height_scale: 5.0
        width_scale: 5.0
      }
    }
    matcher {
      argmax_matcher {
        matched_threshold: 0.5
        unmatched_threshold: 0.5
        ignore_thresholds: false
        negatives_lower_than_unmatched: true
        force_match_for_each_row: true
      }
    }
    similarity_calculator {
      iou_similarity {
      }
    }
    anchor_generator {
      ssd_anchor_generator {
        num_layers: 6
        min_scale: 0.2
        max_scale: 0.95
        aspect_ratios: 1.0
        aspect_ratios: 2.0
        aspect_ratios: 0.5
        aspect_ratios: 3.0
        aspect_ratios: 0.3333
      }
    }
    image_resizer {
      fixed_shape_resizer {
        height: 300
```

```
    width: 300
  }
}
box_predictor {
  convolutional_box_predictor {
    min_depth: 0
    max_depth: 0
    num_layers_before_predictor: 0
    use_dropout: false
    dropout_keep_probability: 0.8
    kernel_size: 1
    box_code_size: 4
    apply_sigmoid_to_scores: false
    conv_hyperparams {
      activation: RELU_6,
      regularizer {
        l2_regularizer {
          weight: 0.00004
        }
      }
      initializer {
        truncated_normal_initializer {
          stddev: 0.03
          mean: 0.0
        }
      }
      batch_norm {
        train: true,
        scale: true,
        center: true,
        decay: 0.9997,
        epsilon: 0.001,
      }
    }
  }
}
feature_extractor {
  type: 'ssd_mobilenet_v2'
  min_depth: 16
  depth_multiplier: 1.0
  conv_hyperparams {
    activation: RELU_6,
    regularizer {
      l2_regularizer {
        weight: 0.00004
      }
    }
    initializer {
      truncated_normal_initializer {
        stddev: 0.03
        mean: 0.0
      }
    }
    batch_norm {
      train: true,
      scale: true,
      center: true,
      decay: 0.9997,
      epsilon: 0.001,
    }
  }
```

```
    }
    loss {
      classification_loss {
        weighted_sigmoid {
        }
      }
      localization_loss {
        weighted_smooth_l1 {
        }
      }
      hard_example_miner {
        num_hard_examples: 3000
        iou_threshold: 0.99
        loss_type: CLASSIFICATION
        max_negatives_per_positive: 3
        min_negatives_per_image: 3
      }
      classification_weight: 1.0
      localization_weight: 1.0
    }
    normalize_loss_by_num_matches: true
    post_processing {
      batch_non_max_suppression {
        score_threshold: 1e-8
        iou_threshold: 0.6
        max_detections_per_class: 100
        max_total_detections: 100
      }
      score_converter: SIGMOID
    }
  }
}

train_config: {
#  batch size: 32
  batch_size: 16
  optimizer {
    rms_prop_optimizer: {
      learning_rate: {
        exponential_decay_learning_rate {
          initial_learning_rate: 0.0002
          decay_steps: 200000
          decay_factor: 0.95
        }
      }
      momentum_optimizer_value: 0.9
      decay: 0.9
      epsilon: 1.0
    }
  }
  fine_tune_checkpoint:
"D:/Anaconda/envs/tf12/Projects/models/ssd_mobilenet_v2_bdd100k/model.ckpt-
108000"
  fine_tune_checkpoint_type: "detection"
  # Note: The below line limits the training process to 200K steps, which we
  # empirically found to be sufficient enough to train the pets dataset. This
  # effectively bypasses the learning rate schedule (the learning rate will
  # never decay). Remove the below line to train indefinitely.
  #num_steps: 200000
  data_augmentation_options {
    random_horizontal_flip {
```

```
    }
  }
  data_augmentation_options {
    ssd_random_crop {
    }
  }
}

train_input_reader: {
  tf_record_input_reader {
    input_path:     "D:/Anaconda/envs/tf12/Projects/data/bdd100k_train.record-
?????-of-00100"
  }
  label_map_path: "D:/Anaconda/envs/tf12/Projects/data/bdd_label_map.pbtxt"
}

eval_config: {
  num_examples: 1000
  # Note: The below line limits the evaluation process to 10 evaluations.
  # Remove the below line to evaluate indefinitely.
  max_evals: 10
}

eval_input_reader: {
  tf_record_input_reader {
    input_path:       "D:/Anaconda/envs/tf12/Projects/data/bdd100k_val.record-
?????-of-00010"
  }
  label_map_path: "D:/Anaconda/envs/tf12/Projects/data/bdd_label_map.pbtxt"
  shuffle: false
  num_readers: 1
}
```

Apéndice D

IMPLEMENTACIÓN DEL FILTRO DE KALMAN

En este apéndice, se describe la implementación del filtro de Kalman en detalle.

D.1 CONSIDERACIONES INICIALES

El vector de estado tiene ocho elementos de la siguiente manera:

```
[up, up_dot, left, left_dot, down, down_dot, right, right_dot]
```

Es decir, utilizamos las coordenadas y sus derivadas de primer orden de la esquina superior izquierda y la esquina inferior derecha del cuadro delimitador.

La matriz del proceso, suponiendo la velocidad constante (por lo tanto, sin aceleración), es:

```
self.F = np.array([[1, self.dt, 0,  0,  0,  0,  0, 0],
                   [0, 1,  0,  0,  0,  0,  0, 0],
                   [0, 0,  1,  self.dt, 0,  0,  0, 0],
                   [0, 0,  0,  1,  0,  0,  0, 0],
                   [0, 0,  0,  0,  1,  self.dt, 0, 0],
                   [0, 0,  0,  0,  0,  1,  0, 0],
                   [0, 0,  0,  0,  0,  0,  1, self.dt],
                   [0, 0,  0,  0,  0,  0,  0,  1]])
```

La matriz de medición, dado que el detector sólo genera la coordenada (no la velocidad), es:

```
self.H = np.array([[1, 0, 0, 0, 0, 0, 0, 0],
                   [0, 0, 1, 0, 0, 0, 0, 0],
                   [0, 0, 0, 0, 1, 0, 0, 0],
                   [0, 0, 0, 0, 0, 0, 1, 0]])
```

El estado, el proceso y los ruidos de medición son:

```
# Initialize the state covariance
 self.L = 100.0
 self.P = np.diag(self.L*np.ones(8))


 # Initialize the process covariance
 self.Q_comp_mat = np.array([[self.dt**4/2., self.dt**3/2.],
                                    [self.dt**3/2., self.dt**2]])
 self.Q = block_diag(self.Q_comp_mat, self.Q_comp_mat,
                             self.Q_comp_mat, self.Q_comp_mat)

# Initialize the measurement covariance
self.R_scaler = 1.0/16.0
self.R_diag_array = self.R_ratio * np.array([self.L, self.L, self.L,
self.L])
self.R = np.diag(self.R_diag_array)
```

Aquí `self.R_scaler` representa la "magnitud" del ruido de medición en relación con el ruido de estado. Un `self.R_scaler` bajo indica una medición más fiable. Las siguientes figuras visualizan el impacto del ruido de medición en el proceso del filtro de Kalman. El cuadro delimitador verde representa el estado de predicción (inicial). El cuadro delimitador rojo representa la medida. Si el ruido de medición es bajo, el estado actualizado (cuadro delimitador de color cian) está muy cerca de la medición (el cuadro delimitador cian se superpone completamente sobre el cuadro delimitador rojo).

Figura 5-1. Cuadros delimitadores con ruido de medición bajo: estado inicial de la predicción (verde), medida (rojo) y estado actualizado (cian).

Por el contrario, si el ruido de medición es alto, el estado actualizado está muy cerca de la predicción inicial (el cuadro delimitador cian se superpone completamente al verde).

Figura 5-2. Cuadros delimitadores con ruido de medición alto: estado inicial de la predicción (verde), medida (rojo) y estado actualizado (cian).

D.2 ASIGNACIÓN DE DETECCIÓN A RASTREADOR

El módulo `assign_detections_to_trackers(trackers, detections, iou_thrd = 0.3)` toma de la lista actual de rastreadores y nuevas detecciones, detecciones coincidentes de salida, rastreadores no coincidentes, detecciones no coincidentes.

Figura 5-3. Proceso de asignación de detecciones a rastreadores.

D.3 ASIGNACIÓN LINEAL Y ALGORITMO HÚNGARO (MUNKRES)

Si hay múltiples detecciones, debemos hacer coincidir (asignar) cada una de ellas con un rastreador. Utilizamos la intersección sobre la unión (IOU) de un cuadro delimitador del rastreador y el cuadro delimitador de detección como una métrica. Resolvemos la maximización de la suma del problema de asignación de IOU utilizando el algoritmo húngaro (también conocido como algoritmo Munkres). El paquete de aprendizaje automático scikit-learn tiene una función de utilidad incorporada que implementa el algoritmo húngaro.

```
matched_idx = linear_assignment(-IOU_mat)
```

Hay que tener en cuenta que `linear_assignment` por defecto minimiza una función objetivo. Por lo tanto, debemos invertir el signo de `IOU_mat` para la maximización.

D.4 DETECCIONES Y RASTREADORES SIN ASOCIAR

Según los resultados de la asignación lineal, mantenemos dos listas para detecciones y rastreadores no coincidentes, respectivamente. Cuando un automóvil entra en un cuadro y se detecta por primera vez, no coincide con ninguna pista existente, por lo tanto, esta detección particular se conoce como una detección sin igual, como se muestra en la siguiente figura. Además, cualquier coincidencia con una superposición menor que `iou_thrd` significa la existencia de un objeto no rastreado. Cuando un objeto abandona el cuadro, la pista establecida previamente no tiene más detección con la que asociarse. En este escenario, la pista se denomina pista no coincidente. Por lo tanto, el rastreador y la detección asociada en la coincidencia se agregan a las listas de rastreadores y detección no coincidentes, respectivamente.

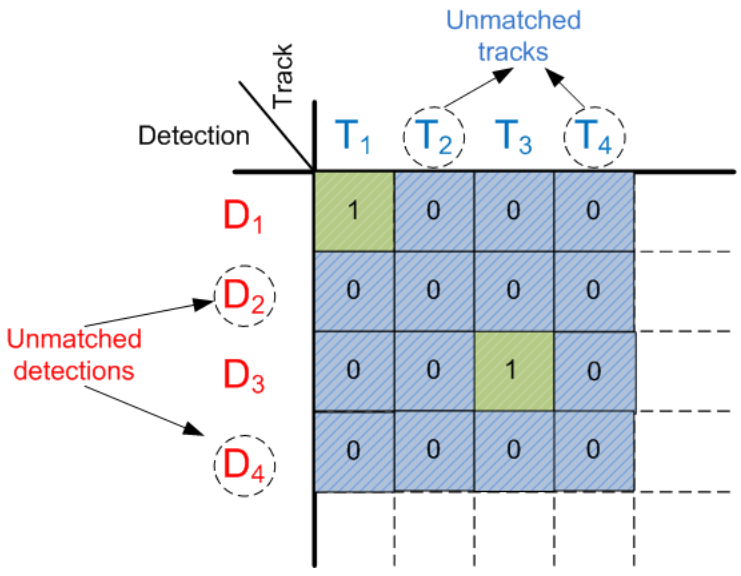

Figura 5-4. Detectores y rastreadores sin asociación.

D.5 PIPELINE

Incluimos dos parámetros de diseño importantes, `min_hits` y `max_age`, en el pipeline. El parámetro `min_hits` es el número de coincidencias consecutivas necesarias para establecer una pista. El parámetro `max_age` es el número de detecciones consecutivas no coincidentes antes de que se elimine una pista. Ambos parámetros deben ajustarse para mejorar el rendimiento de seguimiento y detección.

El pipeline se ocupa de la detección coincidente, la detección incomparable y los rastreadores no coincidentes secuencialmente. Anotamos las pistas que cumplen las condiciones `min_hits` y `max_age`. También se necesita una contabilidad adecuada para eliminar las pistas obsoletas.

Los siguientes ejemplos muestran el proceso del pipeline. Cuando el automóvil se detecta por primera vez en el primer *frame* de video, la ejecución de la siguiente línea de código devuelve una lista vacía, una lista de un elemento y una lista vacía para `matched`, `unmatched_dets`, y `unmatched_trks`, respectivamente.

```
matched, unmatched_dets, unmatched_trks \
    = assign_detections_to_trackers(x_box, z_box, iou_thrd = 0.3)
```

Por lo tanto, tenemos una situación de detecciones sin igual. El siguiente bloque de código procesa las detecciones incomparables:

```
if len(unmatched_dets)>0:
        for idx in unmatched_dets:
            z = z_box[idx]
            z = np.expand_dims(z, axis=0).T
            tmp_trk = Tracker() # Create a new tracker
            x = np.array([[z[0], 0, z[1], 0, z[2], 0, z[3], 0]]).T
            tmp_trk.x_state = x
            tmp_trk.predict_only()
            xx = tmp_trk.x_state
            xx = xx.T[0].tolist()
            xx =[xx[0], xx[2], xx[4], xx[6]]
            tmp_trk.box = xx
            tmp_trk.id = track_id_list.popleft() # assign an ID for
the tracker
            tracker_list.append(tmp_trk)
            x_box.append(xx)
```

Este bloque de código lleva a cabo dos tareas importantes:

1) crear un nuevo rastreador `tmp_trk` para la detección;

2) llevar a cabo la etapa de predicción del filtro de Kalman `tmp_trk.predict_only ()`

Este rastreador recién creado todavía está en período de prueba, es decir, `trk.hits = 0`, por lo que se establece al final del pipeline. La imagen de salida es la misma que la imagen de entrada: el cuadro delimitador de detección no está anotado (Figura 5-5).

```
Frame: 1
[419 886 466 982] , confidence:  0.88656265 ratio: 0.48953234038121024
```


```
Detection:  [array([419, 886, 466, 982])]
x_box:  []
matched: []
unmatched_det: [0]
unmatched_trks: []
Ending tracker_list:  1
Ending good tracker_list:  0
```

Figura 5-5. Proceso de asignación de rastreadores I.

Cuando el automóvil se detecta nuevamente en el segundo cuadro de video, la ejecución de la siguiente `assign_detections_to_trackers` devuelve una lista de un elemento, una lista vacía y una lista vacía para `matched`, `unmatched_dets`, y `unmatched_trks`, respectivamente. Se tiene una detección coincidente, que será procesada por el siguiente bloque de código:

```
if matched.size >0:
        for trk_idx, det_idx in matched:
            z = z_box[det_idx]
            z = np.expand_dims(z, axis=0).T
            tmp_trk= tracker_list[trk_idx]
            tmp_trk.kalman_filter(z)
            xx = tmp_trk.x_state.T[0].tolist()
            xx =[xx[0], xx[2], xx[4], xx[6]]
            x_box[trk_idx] = xx
            tmp_trk.box =xx
            tmp_trk.hits += 1
```

```
Frame: 2
[417 867 461 954] , confidence:  0.7204721 ratio: 0.5056890012642224
```


```
Detection:  [array([417, 867, 461, 954])]
x_box:  [[419, 886, 466, 982]]
matched: [[0 0]]
unmatched_det: []
unmatched_trks: []
updated box:  [417, 868, 461, 956]

Ending tracker_list:  1
Ending good tracker_list:  1
```

Figura 5-6. Proceso de asignación de rastreadores II.

Este bloque de código lleva a cabo dos tareas importantes:

1) llevar a cabo las etapas de predicción y actualización del filtro de Kalman `tmp_trk.kalman_filter()`

2) aumentar los golpes de la pista en uno `tmp_trk.hits +=1`. Con esta actualización, la condición `if ((trk.hits >= min_hits) and (trk.no_losses <=max_age))` está fijada, por lo que la pista está completamente establecida. Como resultado, el cuadro delimitador se anota en la imagen de salida, como se muestra en la Figura 5-6.

Apéndice E

ENLACES EXTERNOS

A continuación se muestran los enlaces a los repositorios donde se puede consultar y descargar el código de los programas que se han realizado para este Trabajo Fin de Grado.

- Benchmark de rendimiento: https://github.com/jotasalor/Benchmark
- Scripts para crear los TFRecords: https://github.com/jotasalor/TFrecord_reload
- SORT - Detector y Tracker: https://github.com/jotasalor/SORT-Tracker
- DeepSORT – Detector y Tracker: https://github.com/jotasalor/DeepSORT-Tracker