UNIVERSIDAD DE VALLADOLID

E.T.S.I. TELECOMUNICACIÓN

TRABAJO FIN DE MÁSTER

MÁSTER EN INGENIERÍA DE TELECOMUNICACIÓN

Estudio de la arquitectura YOLO para la detección de objetos mediante *deep learning*

Autor:
D. Saúl Rozada Raneros

Tutor:
D. Ignacio de Miguel Jiménez

Valladolid, Enero de 2021

TÍTULO: **Estudio de la arquitectura YOLO para la detección de objetos mediante *deep learning***

AUTOR: **D. Saúl Rozada Raneros**

TUTOR: **D. Ignacio de Miguel Jiménez**

DEPARTAMENTO: **Teoría de la Señal y Comunicaciones e Ingeniería Telemática**

TRIBUNAL

PRESIDENTE: **D.ª Patricia Fernández del Reguero**

VOCAL: **D. Ramón J. Durán Barroso**

SECRETARIO: **D. Javier M. Aguiar Pérez**

SUPLENTE: **D.ª Lourdes Pelaz Montes**

SUPLENTE: **D. Ramón de la Rosa Steinz**

SUPLENTE: **D. Jaime Gómez Gil**

FECHA:

CALIFICACIÓN:

Resumen del TFM

El aprendizaje automático y, en particular, el aprendizaje profundo o *deep learning* cada día tienen más presencia en la industria y en la sociedad. Permiten el desarrollo de automatización de tareas sin presencia humana con una gran precisión y también el desarrollo de nuevas aplicaciones y software que facilita la vida de las personas, permitiendo nuevas posibilidades a la hora de explotar la tecnología existente. En este trabajo estudiamos en qué consisten estos conceptos y vemos la forma de aplicarlos a la detección de objetos en imágenes, mediante la arquitectura YOLO (*You Only Look Once*), y se realiza un estudio práctico con evaluación del rendimiento.

Palabras clave

Machine Learning, Deep Learning, YOLO, detección de objetos.

Abstract

Machine learning and, in particular, deep learning, are becoming more and more prevalent in industry and society. They allow the development of automation of tasks without human presence with great precision, and also the development of new applications and software that facilitates the life of people, allowing new possibilities when exploiting existing technology. In this work we study what these concepts are and see how to apply them to the detection of objects in images, through the YOLO architecture (You Only Look Once). A practical study with performance evaluation is also carried out.

Keywords

Machine Learning, Deep Learning, YOLO, object detection

Agradecimientos

En primer lugar, me gustaría agredecer a mi tutor D. Ignacio de Miguel Jiménez, por darme la oportunidad de empezar a aprender los conceptos relacionados con Deep Learning y Machine learning, que permiten el estudio del interesante mundo de la inteligencia artificial.

En segundo lugar, me gustaría agradecer a D. Emiliano Rubio por el apoyo recibido a lo largo del Grado y posteriormente Máster en Ingeniería de Telecomunicación, y que gracias a su ayuda he conseguido llegar hasta aquí.

No me quiero olvidar de mis amigos del Máster, que han permitido que el ambiente y el desarrollo de este sea excepcional en cuanto a ayuda y colaboración entre todos los que lo cursamos. En especial agradecimiento a mi buen amigo Miguel Alonso Felipe, por los consejos y apoyo durante todo este tiempo.

También me gustaría dedicárselo a mi familia, por tantos años de sacrificio en aras de conseguir un futuro mejor para mí. En especial a mi abuelo materno por este momento que no pudo llegar a ver.

ÍNDICE

1 Introducción 9

1.1 Motivación 9

1.2 Objetivos 9

1.2.1 Objetivo General 9

1.2.2 Objetivos Específicos 10

1.3 Estructura de la Memoria del TFM 10

2 Estado del Arte 12

2.1 Qué es Machine Learning 12

2.2 Deep Learning 13

2.2.1 Concepto 13

2.2.2 Aplicaciones 14

2.3 Detección de objetos 14

2.4 Descripción de un detector de objetos 15

2.4.1 Red neuronal 16

2.4.2 Filtros 18

2.4.3 Proceso de entrenamiento 20

2.4.4 Métricas de rendimiento 25

2.5 Arquitecturas de detección de objetos 30

2.5.1 Sistemas de ventana deslizante 30

2.5.2 R-CNN 31

2.5.3 Single Shot Detector (SSD) 32

2.5.4 RetinaNet 33

2.6 YOLO (You Only Look Once) 35

2.6.1 Beneficios 35

2.6.2 Funcionamiento 36

2.6.3 Arquitectura de red 38

2.6.4 Virtudes de YOLO 42

2.7 Evolución del sistema YOLO 42

2.7.1 YOLOv2 [20] 42

2.7.2 YOLO9000 [19] [20] 46

2.7.3 YOLOv3 [21] 47

2.7.4 YOLOv4 50

2.8 Sistemas de detección de objetos YOLO. 51

2.8.1 TrainYourOwnYOLO 52

2.8.2 Ultralytics YOLOv3 56

2.8.3 YOLOv3 Implemented with Tensorflow 2.0, Zihao Zhang 58

3 Estudio práctico del sistema YOLO 59

3.1 Estructura del sistema de detección de objetos 59

3.1.1 Métricas usadas en nuestro proyecto 64

3.2 Detección de un conjunto de datos particular: Clase única compuesta por cuadros de arte 65

3.3 Resultados obtenidos del conjunto de datos de los cuadros de arte 71

3.3.1 Umbral de confianza a 0.2 72

3.3.2 Umbral de confianza a 0.2 y Learning Rate de 1e-4 75

3.3.3 Umbral de confianza a 0.2 y Optimizador SGD 76

3.4 Detección de un conjunto de datos particular. Tres clases diferentes de un objeto que ya reconoce. Detección de razas de perros. 77

3.5 Resultados obtenidos para el conjunto de datos de razas de perro 79

3.5.1 Caso de Umbral de confianza a 0.3 80

3.5.2 Caso de Umbral de confianza a 0.2 82

3.5.3 Caso de Umbral de confianza a 0.2 y métrica de entrenamiento “precision” 84

3.5.4 Caso de Umbral de confianza a 0.2 y función de optimización SGD (Stochastic Gradient Descendent) 85

3.6 Análisis de los resultados obtenidos con los conjuntos de datos 88

4 Conclusiones y Líneas Futuras 90

5 Bibliografía 92

ÍNDICE DE FIGURAS

Figura 1: Estructura de un TLU o unidad lógica de umbral..16

Figura 2: Composición de un perceptrón 16

Figura 3: Estructura de un MLP 17

Figura 4: Esquema de cómo opera una red neuronal convolucional................................18

Figura 5: Ejemplo de aplicación de los filtros a imágenes en cada capa 19

Figura 6: Ejemplo visual de la explicación del proceso de transferencia de aprendizaje [6] 23

Figura 7: Explicación gráfica de la intersección sobre la unión 25

Figura 8: Ejemplo de Matriz de confusión...27

Figura 9 Ejemplo de curva ROC 27

Figura 10: Ilustración del funcionamiento del umbral de confianza en las curvas de Precision-Recall 29

Figura 11: Ejemplo de una curva de un sistema de detección de objetos (azul) y la curva ideal que se debe buscar (verde) 30

Figura 12: Método de detección de objetos por ventana deslizante. [11] 31

Figura 13: Arquitectura de la red SSD [13] 33

Figura 14: Diferencia entre pirámide de imagenes (izquierda) y pirámide de mapas de características (derecha) 34

Figura 15 Arquitectura de RetinaNet 35

Figura 16: Ejemplo de cómo se calculan las coordenadas del cuadro de una imagen. El tamaño es 448x448 pixeles y S=3. [16] 37

Figura 17: Descripción gráfica de la intersección sobre la unión. 38

Figura 18: Esquema de la arquitectura de la red neuronal [12] 39

Figura 19: Ilustracion del funcionamiento del algoritmo K-Means.................................44

Figura 20: Predicciones de los cuadros delimitadores 45

Figura 21: Explicación intuitiva de la posicion de un cuadro delimitador.......................45

Figura 22: Explicación de la detección con YOLO9000 47

Figura 23 Esquema gráfico de cada uno de los elementos implicados en la obtención de las coordenadas [21] ... 49

Figura 24: Arquitectura de la red YOLOv4 ... 50

Figura 25: Ejecución de un entrenamiento del sistema TrainYourOwnYOLO ... 54

Figura 26: Detecciones correctas de caras de gatos ... 55

Figura 27 Detección errónea de caras de gato ... 55

Figura 28: Detecciónes realizadas por el sistema de detección de objetos de ultralytics. 57

Figura 29 Detección realizada por el sistema sin entrenar previamente ... 66

Figura 30 Detección de un segundo cuadro en el que el sistema confunde con otro objeto ... 66

Figura 31 Proceso de etiquetado de imagenes en LabelImg ... 68

Figura 32: Progreso de un entrenamiento ... 70

Figura 33 Detección de un cuadro con umbral a 0.5 y a 0.2 ... 72

Figura 34: Selección de detecciones de cuadros realizadas por el sistema ... 73

Figura 35: Detección de un falso positivo de una imagen de monitor de televisión ... 74

Figura 36: Doble detección de un cuadro en el caso de umbral de confianza de 0,2 y bajando lr a 1e-4 ... 75

Figura 37: Detección de un cuadro con los parámetros de umbral a 0,2 y optimizador SGD ... 76

Figura 38: Curva de Precision-Recall con umbral de 0,2 y optimizador SGD ... 77

Figura 39: Razas de perro a detectar. Afgano, Beagle y Golden Retriever respectivamente ... 78

Figura 40 Golden Retriever detectado como Afgano de manera errónea con umbral de 0.3 ... 81

Figura 41 Precisión con umbral de confianza de 0.3 ... 81

Figura 42: Curva Precision-Recall y Precision para el caso del Afgano con umbral a 0.2 ... 82

Figura 43: Curva Precision-Recall y Precision para el caso del Beagle con umbral a 0.283

Figura 44: Caso de falso positivo originado por el sistema, en el que etiqueta un Golden Retriever como Afgano ... 84

Figura 45: Precisión media mAP del caso con umbral de confianza en 0.2 ... 84

Figura 46: Curva Precision-Recall para un umbral de 0.2 y optimizador SGD (Afgano) 86

Figura 47 Curva Precision-Recall para un umbral de 0.2 y optimizador SGD (Beagle) ..87

Figura 48 Curva Precision-Recall para un umbral de 0.2 y optimizador SGD (Golden Retriever)..87

Figura 49: Detecciones de Golden Retriver, antes no detectadas 88

Figura 50: mAP del sistema global con SGD, y umbral de confianza a 0.2 88

ÍNDICE DE ECUACIONES

Ecuación 1: Definición matemática de Precisión 28

Ecuación 2: Definición matemática de Recall 28

Ecuación 3 Obtención de la puntuación de confianza del objeto [12] 38

Ecuación 4: Pérdida de clasificación del sistema YOLO 40

Ecuación 5: Pérdida de localización del sistema YOLO 40

Ecuación 6: Pérdida de confianza si hay un objeto detectado 41

Ecuación 7: Pérdida de confianza si no hay un objeto detectado 41

Ecuación 8: Función de pérdidas final.. 41

Ecuación 9: Transformaciones para conseguir el centro del cuadro delimitador, así como sus dimensiones [21]... 48

1

Introducción

1.1 Motivación

Cada vez más los procesos industriales y el mundo están más digitalizados y automatizados. Se consigue ganar con ello precisión y velocidad de los procesos de fabricación u obtención de resultados sin la necesidad de renunciar a calidad en los mismos. Esto es crucial en un mundo en el que los cambios se suceden con una velocidad cada vez mayor y en el que la eficiencia es clave para lograr beneficios respecto a la competencia en los distintos sectores de la industria.

Aquí entra en juego lo que se conoce como aprendizaje automático, que es la capacidad de las máquinas de aprender de manera autónoma. Con este concepto se consigue lo anteriormente descrito. Es fundamental en un mundo donde la complejidad de las tareas cada vez es mayor y para mejorar la calidad de productos o servicios exige una precisión nunca vista, tanta que un humano no es capaz en ocasiones de poder gestionar. Tareas como la seguridad, la automatización de robots, el coche autónomo o el hogar conectado exigen de una capacidad de aprendizaje que permita mejorar la calidad de vida de las personas o de la industria en general y esto lo conseguimos gracias a que las maquinas puedan aprender de manera autónoma sin supervisión humana.

1.2 Objetivos

1.2.1 Objetivo General

El objetivo de este trabajo es el de comenzar a conocer el mundo del aprendizaje automático o *machine learning* desde un punto de vista teórico y práctico. Para ello, es necesario obtener una capacidad de análisis que permita por un lado entender en

profundidad en qué consiste el aprendizaje automático y sus dependencias, así como comprender si los resultados obtenidos mediante una forma práctica a partir de los conceptos madurados teóricamente son los adecuados o no.

1.2.2 Objetivos Específicos

Nos centraremos en estudiar el *machine learning* y, en particular, el aprendizaje profundo o *deep learning* para comprender cómo funcionan exactamente y encontrar una forma de aplicarlos a problemas de la vida real. También veremos en qué consiste la detección de objetos y su relación con el aprendizaje automático y distintas técnicas que hay para conseguirlo. Nos centraremos en una de ellas para analizar con mayor profundidad su funcionamiento y así ser capaces de entender cómo puede funcionar adecuadamente un sistema de este tipo.

También se tratará de entender como lidiar con problemas propios de la utilizacion de un sistema de este tipo y las ventajas e inconvenientes de usar distintas estrategias de procesamiento como la programación a nivel local o en la nube. Además, es importante centrarnos en los resultados que vamos obteniendo, así como los errores que puede arrojar y como enfrentarnos ante tales eventualidades, para entender y aprender la forma de solucionarlos de una manera crítica y consistente.

1.3 Estructura de la Memoria del TFM

En primer lugar, haremos un inciso en lo que es el aprendizaje automático y el *deep learning*, así como su utilidad a día de hoy. Posteriormente, conoceremos en que consiste la deteccion de objetos y realizaremos un repaso a las distintas técnicas utilizadas en la detección de objetos. Después nos centraremos en la Arquitectura YOLO y la estudiaremos en detalle, y haremos un repaso a cada una de las versiones que han ido apareciendo a lo largo del tiempo, además de conocer métricas de evaluacion del rendimiento de este tipo de sistemas.

También conoceremos distintos repositorios donde hay proyectos de detección de objetos mediante el uso de la arquitectura YOLO. Finalmente, realizaremos un estudio a través de dos repositorios donde podremos aplicar los conocimientos teóricos estudiados

en la primera parte de esta memoria y donde de verdad podremos evaluar el rendimiento del sistema de detección que hemos seleccionado para realizar el estudio.

2

Estado del Arte

2.1 Qué es Machine Learning

En la detección de objetos se suelen emplear técnicas de Aprendizaje Automático o *Machine Learning*. Se trata de una disciplina informática que consiste en crear sistemas que pueden aprender por sí mismos. Todo ello, está relacionado con el desarrollo de inteligencia artificial. Todo esto tiene como beneficio, que muchas tareas complejas se puedan desarrollar de manera automática, reduciendo la intervención de personas en el proceso para el que esté destinado el sistema.

El sistema aprende, es decir, es capaz de identificar una serie de patrones complejos a partir de los datos. Todo esto lo hace mediante un algoritmo de aprendizaje, ya que se introducen una serie de datos, y a partir de ellos, mediante el algoritmo de aprendizaje, es capaz de generar una hipótesis que le permita realizar predicciones cuando se enfrenta a nuevas entradas [1].

Existen varios tipos de aprendizaje automático:

- Aprendizaje supervisado: Se trata de un tipo de aprendizaje donde al sistema se le alimenta con datos de entrenamiento etiquetados. Una vez se ha completado el entrenamiento, se pueden introducir nuevos datos al sistema de manera que ya no están etiquetados, puesto que al estar entrenado el sistema es capaz de reconocer patrones para identificar los datos correspondientes. En este tipo de aprendizaje están los denominados problemas de clasificación y los de regresión [1].
- Aprendizaje no supervisado: En este tipo de aprendizaje se alimenta el sistema sin datos etiquetados. Uno de los problemas más comunes de este tipo de aprendizaje automático es el de agrupamiento o *clustering*, en el que el sistema aprende a

organizar los datos en distintos grupos, de modo que cada grupo contenga datos similares entre sí [1].

- Aprendizaje por refuerzo: Se trata de que el sistema aprenda a partir de la experiencia. Esto es que, el sistema al aprender, si se equivoca se le penaliza de forma que el sistema va aprendiendo con el objetivo de maximizar la recompensa. Es algo muy parecido al aprendizaje humano, cuando se enseña a los niños conceptos o conductas y si lo hacen correctamente se les recompensa y si no se les penaliza. Es una idea muy similar [2].

En nuestro caso, vamos a utilizar la primera de las variantes, es decir, el aprendizaje automático. Concretamente, se tratará de tomar imágenes donde están etiquetados los objetos que queremos detectar, con el objetivo de que el sistema vaya aprendiendo las características de dichos objetos, y una vez que acabe el entrenamiento (el proceso de aprendizaje), comenzaría el proceso de detección, en el que se debe comprobar si el sistema ha aprendido correctamente los objetos para el que ha sido entrenado. Esto es una gran ventaja pues es un "cerebro" artificial, con la capacidad de reconocer el objeto sin necesidad de que la tarea la realice un humano.

2.2 Deep Learning

2.2.1 Concepto

El aprendizaje profundo o *deep learning* es un campo dentro del área del *Machine Learning*. Se trata de un paradigma de aprendizaje automático que intenta asemejarse a la forma del aprendizaje humano para poder aprender determinados datos, tareas… [3]. Para el procesamiento de la información, usa redes neuronales artificiales, pero concatenando un gran número de capas formadas por unidades de procesamiento sencillas (neuronas), que van transformando progresivamente la información. Hay varios tipos de capas, la capa de entrada a la red, las capas ocultas, y la capa de salida. Más adelante las veremos con detalle la función que tienen cada una de las capas, pero quedémonos con que la capa de entrada sirve como entrada de la información a la red neuronal, las capas ocultas permiten el procesamiento de la información, descomponiendo la información de entrada, y la capa de salida es la que realiza la toma de decisiones en base al procesamiento realizado en las capas anteriores [3].

2.2.2 Aplicaciones

Cada vez más el *deep learning* está llegando a más áreas. Por ejemplo, tiene muchas aplicaciones en el mundo de la medicina, para ofrecer diagnosticos cada vez mas precisos y claros, y en el mundo financiero, donde los modelos pueden ser de gran utilidad para la predicción del comportamiento de los mercados. Otras áreas donde está cada vez mas presente son el coche autónomo, la detección de defectos, el procesamiento de datos… [3]

Incluso si miramos a nuestro alrededor, ya podemos encontrar algunas aplicaciones que usan dicho concepto. Por ejemplo, los traductores inteligentes como el de Google, que va aprendiendo de las oraciones y palabras que escriben las personas para futuras traducciones. Otro es el reconocimiento de voz, que es muy usado en teléfonos, ordenadores o en coches. Un ejemplo es el de Siri en los dispositivos de Apple. También existen aplicaciones que, mediante el uso de la cámara del dispositivo móvil, permiten el reconocimiento de palabras o oraciones escritas en un idioma, y realizan una traducción automática de las mismas. Otro ejemplo que podemos mencionar es el reconocimiento facial que permite obtener un sistema de seguridad biométrico o colocar filtros en las imágenes que se toman mediante la cámara de un teléfono, y que permite añadir adornos en la cara de una persona, como mascaras o dibujos [3].

Todas estas aplicaciones son algunas de las muchas que se están implementando hoy en día, y cada día crece más el número de aplicaciones que pueden hacer uso de esta tecnología.

2.3 Detección de objetos

Normalmente los problemas más “habituales” dentro de *machine learning*, son los llamados problemas de clasificación. Se trata de que a partir de unos datos concretos y en función de los parámetros con los que se configure el sistema, éste sea capaz de clasificarlos de una manera adecuada, en función siempre de una serie de variables.

Por ejemplo, tenemos el caso más visual que es la clasificación de imágenes. Supongamos que estamos utilizando el aprendizaje supervisado y que lo que hacemos es alimentar el sistema de *machine learning* mediante imágenes de distintos objetos. Cada uno de estos objetos llevará asociada al menos una etiqueta, el tipo de objeto que es. Con ello, entrenamos el sistema durante un tiempo, después del cual, podremos mostrar a dicho

sistema un objeto similar al de las imágenes con las que se ha entrenado, y debería ser capaz de clasificar el objeto en cuestión de manera correcta.

Sin embargo, hay un problema de *machine learning* que va un paso más allá, que es la detección de objetos. Se trata de un problema más complejo que el de la clasificación, ya que consiste en no solo decir si el objeto está o no presente en la imagen en cuestión, sino también en qué región de la imagen se encuentra dicho objeto. Esto es que, en una imagen utilizada para clasificación, lo único que se necesita es que el objeto que queremos clasificar esté en la imagen. Pero en este caso, queremos saber si está o no presente y además dónde está dicho objeto concretamente en la imagen.

En nuestro caso, la detección de objetos consistirá en el uso de un sistema ya diseñado y aprovecharemos el conocimiento que tiene el mismo, para poder detectar otro tipo de objetos o variaciones de objetos que ya es capaz de detectar el propio sistema. Para ello, como veremos más adelante deberemos de realizar algunos ajustes para que pueda realizar las detecciónes de una manera más idónea.

Para que sea más intuitivo, podemos ver el problema de detección de objetos visto por una persona. Desde pequeños podemos ver imágenes y mediante el aprendizaje repetitivo de los objetos que se nos van mostrando, podemos ir aprendiendo todos los objetos que aparecen en nuestro entorno. Nuestro cerebro va reconociendo los distintos objetos que son "grabados" gracias al aprendizaje que nos transfieren nuestros progenitores a base del nombre de dicho objeto (que son las etiquetas) o distintas personas que nos rodean en nuestro entorno.

Para ello, necesitamos en primer lugar el ojo, que es el punto de entrada de la información y después el cerebro, que compuesto por neuronas es el que se encarga de procesar la información. Todo ello, realizando estas tareas de manera repetitiva, se va consiguiendo asimilar o aprender la clase de objeto que tenemos delante de nosotros y así la próxima vez que nos encontramos con objetos similares, somos capaces de reconocerlo.

2.4 Descripción de un detector de objetos

En este apartado vamos a explicar las partes de las que se compone un detector de objetos, así como el proceso de la detección, y entrenamiento para que el detector funcione de la forma esperada.

El eje central de un detector de objetos basado en *deep learning* es la red neuronal, que será la encargada de realizar las operaciones necesarias para extraer las características de las imágenes y así poder obtener las características que se corresponden con un objeto, todo con el objetivo de que pueda aprender a reconocer el objeto en una imagen dada.

2.4.1 Red neuronal

En primer lugar, hablaremos de la base del sistema de detección de objetos, el núcleo del mismo, que no es otro que la red neuronal. Para entender la red neuronal es necesario explicar lo que es una TLU (*threshold logic unit*) o unidad lógica de umbral. Se trata de una neurona artificial, que se compone de una serie de entradas y de una salida. Las entradas están asociadas a unos pesos de forma que la TLU calcula la suma ponderada de sus entradas para obtener una salida, como podemos ver en la Figura 1 [4].

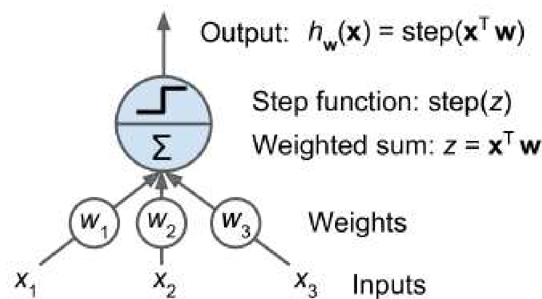

Figura 1: Estructura de un TLU o unidad lógica de umbral

Un perceptrón, es una capa de TLUs con cada TLU conectada a todas las entradas. A continuación, en la Figura 2 apreciamos una serie de TLU que en este caso forman parte de la capa de salida, y además están completamente conectadas a sus entradas [4].

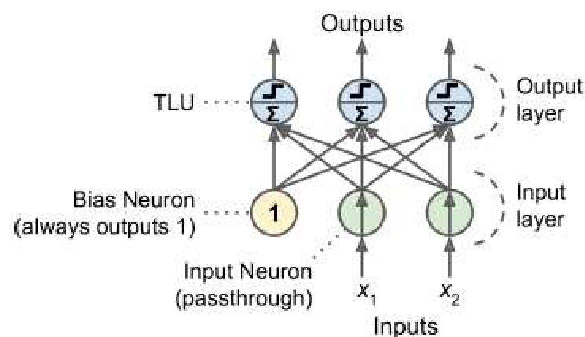

Figura 2: Composición de un perceptrón

Cada TLU se activará en función de si la suma ponderada de las entradas tiene un valor mayor o menor que un umbral. De esta forma si la suma ponderada es mayor que el umbral, tendremos una salida de valor 1, mientras que si es menor que el umbral, la salida será nula o bien -1, todo dependiendo del tipo de función de activación [4]. Las funciones de activación dan un valor de salida según sea el valor que tienen a la entrada. Estos valores de salida pueden ser entre 0 y 1 o bien entre -1 y 1, aunque también existen otras posibilidades.

Finalmente tenemos los MLP o *multilayer perceptron*, que no es mas que una capa de entrada, una o mas capas ocultas y una capa de salida, donde las capas cercanas a la de entrada se llaman capas inferiores y las que están cerca de la salida se llaman superiores. Esto lo vemos con más detalle en la Figura 3 [4].

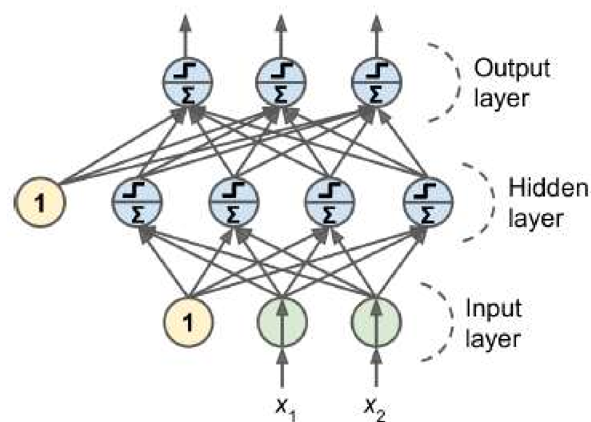

Figura 3: Estructura de un MLP

En una red neuronal para detección de objetos, usaremos las redes neuronales convolucionales, ya que el problema que nos ocupa es el procesamiento de imágenes. En este caso lo que hace la red es trabajar como entradas, con los píxeles de cada imagen. En este caso, en una red convolucional, en la primera capa, no está conectada a todos los píxeles de la imagen de entrada, como hemos comentado anteriormente, sino que está conectada a una porción de ellos, a saber, los que podemos apreciar en la Figura 4 [4].

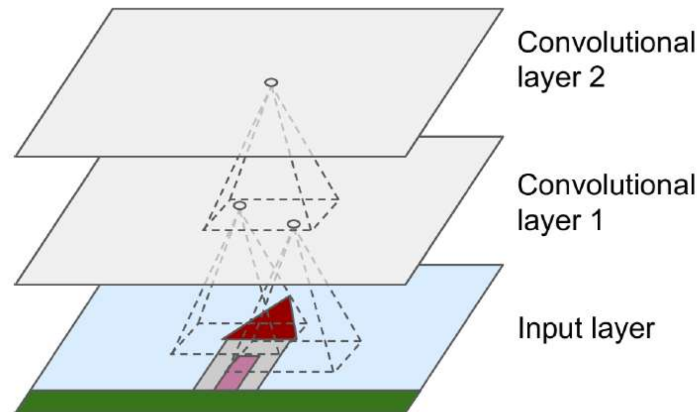

Figura 4: Esquema de cómo opera una red neuronal convolucional

En la figura podemos ver como de la primera capa convolucional está conectada únicamente al área de pixeles delimitada por los recuadros que podemos ver en la imagen de entrada, que son los pixeles de los campos receptivos. A su vez, cada neurona en la segunda capa convolucional está conectada solo a las neuronas ubicadas dentro de un pequeño rectángulo en la primera capa. De esta forma se van extrayendo las características pequeñas en la primera capa oculta, y se luego ensamblarlas en características más grandes de nivel superior en la siguiente capa oculta, y así sucesivamente. Esta estructura jerárquica es común en las imágenes del mundo real, que es una de las razones por las cuales las CNN funcionan tan bien para el reconocimiento de imágenes [4].

2.4.2 Filtros

Los pesos de una neurona se pueden representar como una imagen pequeña del tamaño del campo receptivo. Si nos fijamos en los filtros tomando como ejemplo la Figura 5, vemos cómo se aplican dos tipos de filtros, el izquierdo al usarlo, las neuronas que usan dichos pesos ignorarán todo en su campo receptivo excepto para la línea vertical central, ya que todos se multiplican por cero excepto los de la línea central que se multiplica por uno. Y lo mismo para para el otro caso en el que tenemos una línea horizontal central [4].

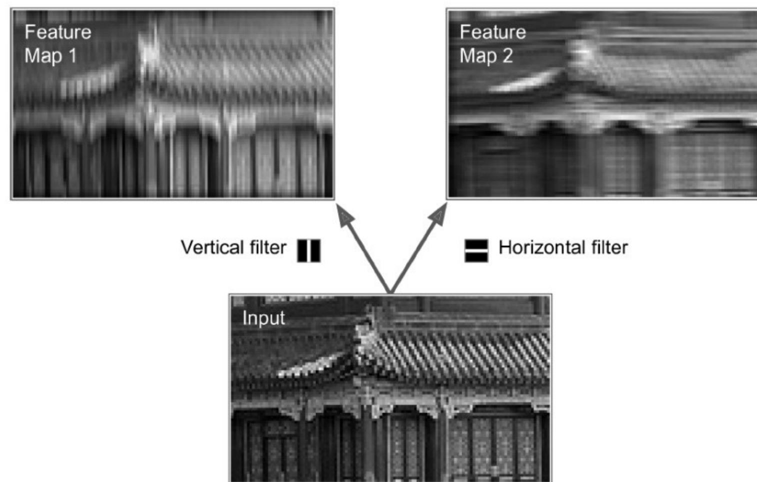

Figura 5: Ejemplo de aplicación de los filtros a imágenes en cada capa

En el primer caso vemos qué ocurre cuando todas las neuronas de una capa usan el mismo filtro, vemos que la capa genera la imagen superior izquierda como podemos apreciar en la Figura 5, donde se aprecia que las líneas verticales están más detalladas, mientras que el resto está borroso. Y lo mismo pasa en el caso contrario, con el filtro de la línea horizontal. Si todas las neuronas de una capa utilizan el filtro de la derecha, vemos como la salida de la capa será la imagen superior derecha donde las líneas horizontales están más marcadas, mientras que el resto está más borroso. Al resultado de la aplicación de uno de estos filtros se llama, mapa de características, que es la salida de la aplicación del mismo a una imagen de la capa anterior [4].

El proceso que acabamos de contar es lo que se aplica a cada capa de todas las existentes en la red. Normalmente cada capa convolucional tiene varios filtros, con los que para cada uno de ellos se genera un mapa de características. Con estos mapas, se consigue extraer la información de los objetos que hay contenidos en las imágenes y con ello se puede reconocer la composición de los objetos en una imagen, es decir, sus características [4].

Hay que destacar que en este tipo de redes de procesamiento de imágenes se suelen usar capas "*Pooling*", que son capas que permiten reducir la memoria y carga computacional a la hora de procesar los datos, y lo que hace es un submuestreo para poder lograr dicha tarea [4].

2.4.3 Proceso de entrenamiento

Vamos a explicar en qué consiste el proceso de entrenamiento, puesto que nos va a servir, por un lado, para explicar elementos necesarios para entender el funcionamiento de un detector, y por otro lado nos servirá para explicar el proceso a llevar a cabo de como un objeto contenido en una imagen es detectado por el sistema. El entrenamiento es el proceso por el que el sistema va consumiendo los datos de entrada, para que pueda ir obteniendo de cada entrada los objetos que se desean detectar mediante la descomposición que hemos comentado antes de las imágenes. Con ello, permitirá que el sistema aprenda el objeto que deseamos que detecte y que así se pueda usar para reconocimiento del mismo de manera automática sin la supervisión humana.

2.4.3.1 Entradas

El primer paso para comentar es el de la preparación de los datos. En este punto hay que destacar que algunas redes utilizadas en la detección de objetos, como es el caso del sistema YOLO (del que se hablará más adelante), funcionan con un tamaño de imagen determinado. Esto es que de manera óptima las imágenes de entrada se deben de adaptar al tamaño que requiere el sistema por ahorro de coste computacional.

El siguiente paso es etiquetar la imagen puesto que en la detección de objetos es necesario indicar al sistema dónde está localizado el objeto en la imagen, para que este sea capaz de aprender dicho objeto. Esto es un proceso tedioso, puesto que, aunque existen programas que facilitan realizar el etiquetado, sigue siendo un proceso largo. Pensemos que hay que indicar mediante cuadros delimitadores donde está el objeto en la imagen y eso para una colección de imágenes que puede ser de 500 en el mejor de los casos. Sin embargo, una vez etiquetado, obtenemos un fichero de anotación para cada imagen, en el que estará presente cada objeto que hay en la imagen, con su respectiva clase para que el sistema pueda utilizarlo a la hora de entrenar.

Tomando ya las entradas como hemos explicado anteriormente, bien por el etiquetado de imágenes de manera masiva o bien por la descarga de un conjunto de datos ya elaborado, lo que se hace es en primer lugar dividir el conjunto de datos en entrenamiento, validación y test. El conjunto de entrenamiento nos servirá para poder alimentar el sistema en el proceso de entrenamiento como su propio nombre indica. A

partir de él, el sistema irá aprendiendo las características de los objetos que deseamos detectar. En segundo lugar, tenemos el conjunto de datos de validación, que nos permitirá ir realizando una supervisión del entrenamiento. Concretamente nos permite realizar una evaluación del ajuste del modelo mientras se ajustan los hiperparámetros del mismo [5]. Por último, el conjunto de datos de prueba o test, nos servirá para poder evaluar el rendimiento del sistema, ya que con el podremos observar cómo de bueno ha sido el entrenamiento o como de bueno es el detector de objetos. Además, nos servirá para obtener métricas de rendimiento con los que poder evaluar el mismo sistema.

2.4.3.2 Pre-procesamiento

Antes de poder alimentar el sistema es necesario adecuar las imágenes con las que se va a alimentar el sistema. Para ello, lo primero es realizar una normalización del tamaño de las imágenes y de los valores de los píxeles de la imagen. En el caso de normalizar el tamaño, se hace debido a fines de mejora de la eficiencia de computación y además porque la red se adaptará mejor a un tamaño de imagen concreto en lugar de tamaños de imagen dispares. Por otra parte, tenemos la normalización de los pixeles, para que tomen valores entre 0 y 1. Esto se realiza para que sea también más eficiente el entrenamiento.

En el caso de YOLO, que es la arquitectura que nos ocupa en este estudio, se cargan también junto con las imágenes y las transformaciones los cuadros de anclaje. Los cuadros de anclaje son un conjunto de anchos y alturas predeterminados que se escogen con el objetivo de que coincidan con los anchos y altos de los objetos que tenemos en el conjunto de datos.

2.4.3.3 Carga del modelo, parámetros implicados y generación de un entrenamiento

Para la correcta ejecución del entrenamiento es necesario especificar una serie de parámetros para su adecuado progreso. En primer lugar, encontramos las épocas de las que consta el entrenamiento, que es el número de veces que vamos a iterar sobre todo el conjunto de datos durante el entrenamiento. Esto es que, si usamos 50 épocas, el entrenamiento durará el tiempo en que se tarde en recorrer el conjunto de datos de entrenamiento 50 veces. Por otro lado, tenemos el *BatchSize*, que es el número de imágenes

que vamos a enviar a la red para dar lugar a una actualización de los pesos. Por ejemplo, si el *BatchSize* tuviera un valor igual a la décima parte del número de datos disponibles, los pesos se actualizarían 10 veces durante una época. Otro parámetro importante es el número de clases, que es el número de clases de objetos distintos que vamos a entrenar.

Además, debemos prestar atención a la tasa de aprendizaje (*learning rate*). Se trata del tamaño de los pasos que se realizan para poder llegar a una solución óptima. Esto tiene su explicación junto con la función de coste. La función de coste es una medida de error entre el valor real y el valor que predice el modelo. Por tanto, durante el proceso de entrenamiento buscamos el valor que minimiza esta función de coste. Y aquí es donde entra en juego la tasa de aprendizaje pues si el valor de esta es alto, hace que no se llegue a encontrar el mínimo de la función de coste y que hasta pueda diverger, no llegando a una solución óptima. Sin embargo, si el valor de la tasa de aprendizaje es demasiado bajo, llevara bastante tiempo el llegar a una solución óptima, pero como los pasos son más pequeños, es más probable que se llegue a la solución que minimiza la función de coste [4].

Posteriormente, es necesario especificar la red neuronal que se vamos a utilizar para nuestra detección y por ende para nuestro entrenamiento. Esto se puede programar de diversas maneras, como por ejemplo declarando cada una de las capas una a una, aunque hay casos en los que se pueden cargar un modelo predeterminado de red. Algunos ejemplos son Resnet, GoogleNet, VGG… Estas redes se pueden entrenar desde cero, es decir alimentando la red directamente con los datos de entrada, pero también se puede aprovechar el aprendizaje que ha realizado la red para una serie de objetos y aplicarlo al aprendizaje de un nuevo objeto que deseamos que detecte, y esto es lo que se conoce como *Transfer learning* [6].

Supongamos que tenemos una red neuronal que ha aprendido a identificar una clase dada de objeto, pero creamos una nueva red neuronal con la que deseamos detectar otra clase de objeto distinta. En este punto podemos o bien entrenar inicializando aleatoriamente pesos y sesgos de las primeras capas de la nueva red neuronal, o bien podemos inicializarlos al valor de los pesos y sesgos de las capas inferiores de la primera red. De esta manera, la red no tendrá que aprender desde cero todas las estructuras de bajo

nivel, evitando un tiempo de entrenamiento mayor. Con esto, la red solo tendrá que aprender las estructuras de nivel superior. La ventaja de esta técnica es que el entrenamiento es mucho más rápido, y además se requieren menos datos a la hora de entrenar [6].

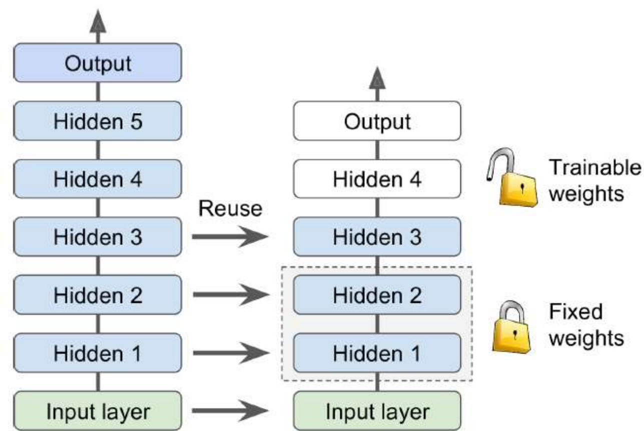

Figura 6: Ejemplo visual de la explicación del proceso de transferencia de aprendizaje [6]

Si nos fijamos en la Figura 6, vemos cómo tomamos una nueva red para una tarea nueva de forma que aprovechamos las capas ya entrenadas de otra red, que ha sido entrenada para una tarea distinta. ¿Cómo hacemos esto? Simplemente creando capas para esta segunda tarea a imagen y semejanza de la red que ya esta entrenada, y cargando en la nueva red los pesos y sesgos de la ya entrenada. Con ello, en el entrenamiento estas capas con los pesos preentrenados, se congelan de forma que los pesos no se alteran, y lo que se hace es entrenar las capas superiores con el fin de realizar pequeños ajustes en la red para que sea capaz de detectar otros objetos.

Por tanto, en el entrenamiento a la hora de realizar un entrenamiento con aprendizaje de transferencia se deben de cargar los pesos preentrenados en la red en la cual vamos a realizar el nuevo entrenamiento.

El siguiente paso a la hora de configurar el entrenamiento es el de realizar la compilación del modelo, para poder definir la función de pérdidas y el optimizador que se seguirá durante el proceso. El optimizador permite ajustar los parámetros del modelo para minimizar las pérdidas de la función de coste sobre el conjunto de datos de entrenamiento. Por otra parte, la función de pérdida nos permite evaluar cómo de bien funciona el algoritmo al consumir los datos.

Posteriormente a la compilación del modelo podemos definir una serie de parámetros adicionales que nos permitirán gestionar de una mejor manera el entrenamiento. Uno de ellos es la gestión de *checkpoints* o puntos de control de entrenamiento, que guardan el valor exacto de todos los parámetros usados por un modelo. Esto significa que al finalizar una época en el entrenamiento se genera un punto de control con los valores de los parámetros que se han utilizado durante esa época. Esto es útil sobre todo en el caso de las detecciones como veremos más adelante.

Otro parámetro importante es *EarlyStopping*, que se trata de un parámetro que permite detener el entrenamiento de manera automática si las pérdidas que se obtienen no disminuyen después de una época o épocas determinadas. Nosotros podemos configurar este parámetro de forma que se monitoricen las pérdidas de validación que se van obteniendo, ya que el objetivo es minimizarlas para que se llegue al mínimo de la función de coste. Por tanto, si dichas pérdidas no disminuyen en un número de épocas consecutivas, se asume que se ha alcanzado el mínimo de la función de coste (aunque no sea así necesariamente) y por tanto podemos detener el entrenamiento.

Finalmente, una vez definidos los parámetros anteriores y ejecutadas las correspondientes tareas descritas, ejecutamos el entrenamiento para que la red neuronal comience a aprender las características del objeto que deseamos detectar.

2.4.3.4 Detección de un objeto

Finalmente tenemos el proceso de detección de los objetos sobre la red que se ha entrenado para la detección de estos. En este caso, lo que hacemos es cargar la red neuronal con los pesos que se acaban de ajustar durante el entrenamiento, y posteriormente se alimenta la red con datos (imágenes en nuestro caso) que contienen (o no) el objeto que estamos intentando detectar. En este punto, lo que hará el sistema es realizar la detección de los objetos que hay contenidos en las imágenes, mediante el proceso que hemos comentado anteriormente y se dibujarán los cuadros delimitadores, que marcarán la presencia del objeto para el que hemos entrenado el sistema. Además de ello, en el caso de la arquitectura que vamos a exponer más adelante, nos dará las coordenadas exactas de la ubicación del objeto en la imagen, así como el tipo de objeto que se ha detectado, es decir, la clase del objeto.

2.4.4 Métricas de rendimiento

En realidad, no solo vale con que el sistema de detección detecte los objetos que nosotros deseamos, sino que debemos de saber cuál es el rendimiento de dicho sistema de detección de objetos. Para ello debemos evaluarlo mediante una serie de métricas que nos permitan obtener un análisis claro de si el sistema funciona bien o funciona mal.

Una técnica muy interesante para evaluar las prestaciones es la validación cruzada. Consiste en dividir el conjunto de entrenamiento en conjuntos mas pequeños de entrenamiento y validación, para luego entrenar el modelo con cada conjunto de entrenamiento que se acaba de conseguir y a continuación aplicarlo al conjunto de validación para evaluar el error cometido en el mismo [7]. Este método nos permite obtener una estimación del rendimiento del modelo (promediando los errores de validación obtenidos), pero también nos da cuánto de precisa es la estimación [7].

Para entender cómo funcionan las métricas y su funcionamiento, debemos de realizar algunas definiciones importantes.

- <u>Intersección sobre la unión (IOU, *Intersection over Union*):</u> En este caso es la medida que evalúa el porcentaje de superposición de dos cuadros delimitadores (*Bounding box*). Concretamente tenemos 2 tipos de cuadros delimitadores, los *Groundbox*, que son las anotaciones que se han realizado sobre las imágenes y que permiten poder realizar el entrenamiento y validación en el sistema de detección de objetos, y por otro tenemos los cuadros que se obtienen de la detección que ha realizado el sistema. Esto lo podemos ver en la Figura 7. IOU será 1, si coinciden tanto la detección como los cuadros delimitadores de las anotaciones, mientras que cuanto menor sea el valor, más alejado estará de una detección fiable.

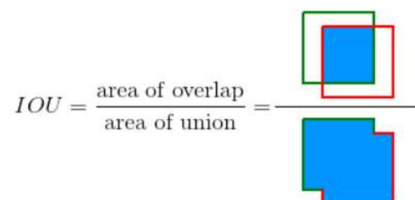

Figura 7: Explicación gráfica de la intersección sobre la unión

Concretamente, para conocer si una detección es o no buena tenemos los siguientes conceptos.

- Verdadero Positivo (True Positive, TP): Se produce cuando se ha detectado el objeto y IOU>=Umbral. Es una detección correcta.

- Falso Positivo (False Positive, FP): Se produce cuando el sistema informático detecta el objeto, pero al calcular el IOU se obtiene que IOU<Umbral. Se trata por tanto de una detección incorrecta.

- Falso negativo (False Negative, FN): Es cuando existe un *groundbox* en la imagen pero no se detecta.

- Verdadero Negativo (True Negative, TN): No se utiliza puesto que serían todos los cuadros delimitadores posibles que no contienen y en los que no se detectan objetos. En detección de objetos hay una infinidad de cuadros delimitadores posibles que no se deben detectar en una imagen.

- Umbral: El que se establece para que se detecte un objeto. Generalmente 50%, 75%, o 95%.

2.4.4.1 Matriz de confusión

Las métricas que acabamos de definir suelen representarse en la denominada matriz de confusión. Se trata de una matriz cuadrada que se usa para evaluar el rendimiento de un modelo, normalmente de clasificación. En ella se compara los valores predichos por el sistema con los reales. Esto nos da una forma más de ver cómo de bien esta funcionando nuestro modelo. Para poder interpretarla nos podemos fijar en la Figura 8

	Real Values		
Predicted Values		Positive	Negative
	Positive	TP = True Positive	FP = False Negative
	Negative	FN = False negative	TN = True Negative

Figura 8: Ejemplo de Matriz de confusión

Si nos fijamos, encontramos que la variable de destino tiene dos posibles valores, bien positivo o negativo. Por otra parte, los valores reales se representan en la columna, mientras que en las filas se representan los valores predichos. Si por ejemplo tenemos un sistema en el que tenemos 1000 datos y en el que se obtiene una matriz de confusión con los siguientes valores: TP=600, TN=300, FN=50, FP=50. Tenemos que es un sistema bastante con una tasa de acierto del 90% (pues se identifican correctamente 900 de los 1000 datos) pero también nos permite ver qué tipos de errores comete, pues según la aplicación puede tener más relevancia un tipo de errores que otros [7] [8]. En los siguientes apartados describiremos algunas métricas relacionadas con los valores obtenidos en la matriz de confusión.

2.4.4.2 Curva ROC

Se trata de una curva que representa la tasa de verdaderos positivos (TPR), que es la ratio de instancias positivas que se clasifican correctamente frente a la tasa de falsos positivos (FPR), que es la ratio de instancias negativas que se clasifican incorrectamente como positivos. Veamos esto con más detalle en la Figura 9.

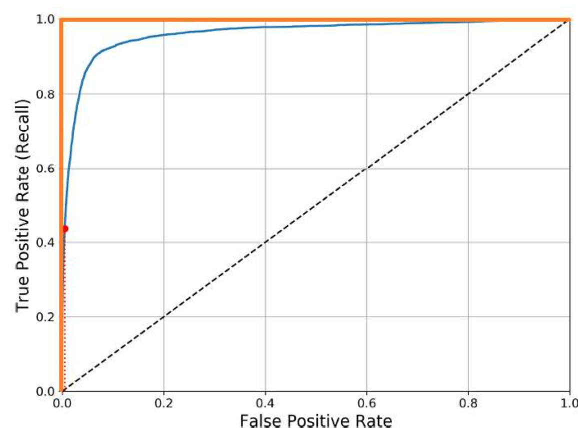

Figura 9 Ejemplo de curva ROC

La línea naranja se corresponde con un sistema ideal mientras que la línea azul se corresponde con un sistema real. Un buen sistema se acercará lo más posible a la línea

naranja que marca el ideal. Si nos fijamos, vemos como encontramos un compromiso. Cuanto mayor sea la tasa de verdaderos positivos (TPR), más falsos positivos (FPR) se producirán en el sistema [7].

Para poder comparar distintos sistemas, la forma más efectiva es calcular el área bajo la curva (AUC). Un sistema ideal tendrá un AUC de 1 mientras que un sistema real estará por debajo de este valor, pero que cuanto más se aproxime a 1, da cuenta de que mejor sistema será puesto que se aproxima al ideal [7].

2.4.4.3 Precision y Recall

La precisión nos da una medida de la calidad del modelo propuesto. Indica, de todas las cajas que devuelve el sistema informático (es decir, de todos los objetos que encuentra), qué fracción de ellas realmente contienen el objeto a detectar. Su valor ideal sería la unidad. En la Ecuación 1 vemos cómo lo podemos calcular [9].

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} = \frac{\text{TP}}{\text{all detections}}$$

Ecuación 1: Definición matemática de Precisión

El *Recall* o sensibilidad, indica qué fracción de objetos, de los que verdaderamente existen, encuentra el detector. Si el sistema es capaz de encontrar todos los casos relevantes, es decir, que encuentre todos los *ground truth*, que son todas las anotaciones que hay en una imagen, diremos que es un sistema muy sensible. En la Ecuación 2 vemos cómo se calcula [9].

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} = \frac{\text{TP}}{\text{all ground truths}}$$

Ecuación 2: Definición matemática de Recall

2.4.4.4 Curva de Precision-Recall

Los dos conceptos descritos anteriormente están estrechamente relacionados entre sí. Si nosotros configuramos el sistema para obtener una mayor precisión, tendremos en general como resultado una disminución del Recall, es decir el sistema será menos sensible y por tanto se detectarán menos objetos. Por otra parte, si configuramos el sistema para tener más sensibilidad, vemos que el sistema será menos preciso puesto que a la hora de realizar detecciónes, el sistema es más laxo a la hora de decidir. Todo esto se gestiona a

partir del umbral de puntuación (*Threshold score*). En la Figura 10 vemos gráficamente lo qué sucede, si aumentamos el umbral, vemos como decae la sensibilidad, pero aumenta la precisión. Por otro lado, si lo bajamos, tendremos en general el caso contrario, mayor sensibilidad y peor precisión [7] [10].

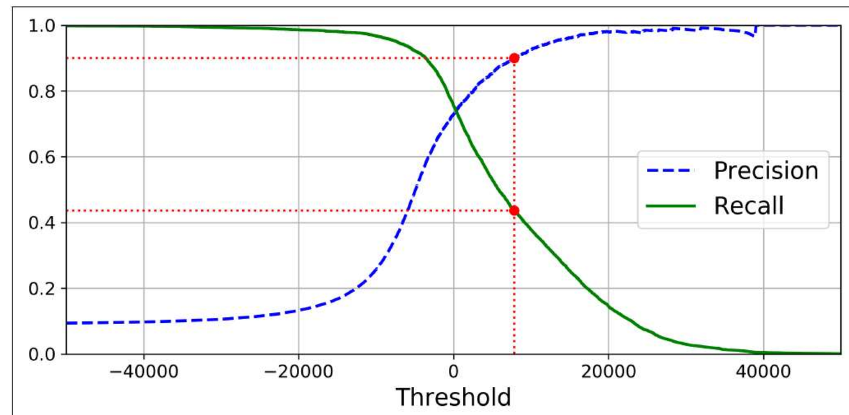

Figura 10: Ilustración del funcionamiento del umbral de confianza en las curvas de Precision-Recall

Un detector lo consideramos bueno, si para cuando va aumentando el Recall la precisión se va manteniendo alta. Esto lo podemos ver en la Figura 11, en la cual encontramos un caso de un detector ideal y otro de un detector real. Vemos que la línea verde es la línea ideal de cómo se debería de comportar un sistema perfecto, mientras que la línea azul representa un ejemplo de un sistema real [7] [10].

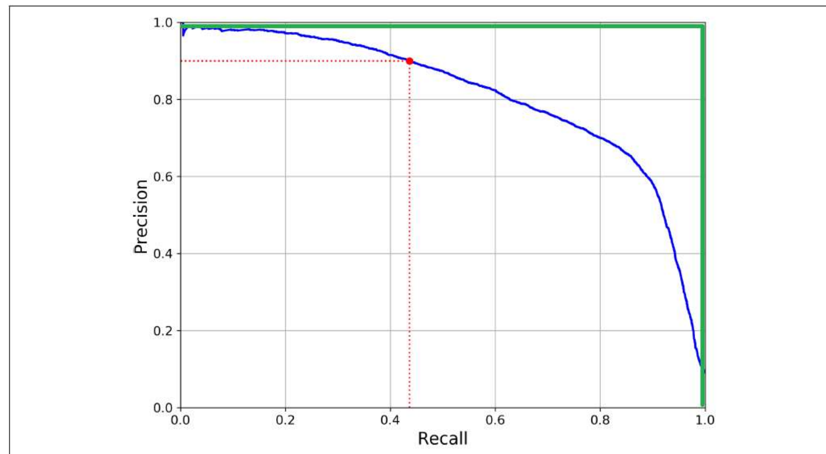

Figura 11: Ejemplo de una curva de un sistema de detección de objetos (azul) y la curva ideal que se debe buscar (verde)

2.4.4.5 Precisión media (mAP)

Es otra de las formas de encontrar un buen o mal detector de objetos es utilizar la precisión media o mAP. Se trata de encontrar el área bajo la curva de Precision-Recall, de esta forma encontramos una medida numérica que facilita la tarea de saber si es bueno un detector o no [7].

2.5 Arquitecturas de detección de objetos

En este apartado comentaremos distintas arquitecturas de detección de objetos existentes en la actualidad. Es importante conocer su funcionamiento, ya que puede darnos una visión general de las distintas formas que hay para realizar un detector.

2.5.1 Sistemas de ventana deslizante

Hasta hace unos años, el método que se usaba para lograr un detector de objetos, era tomar una CNN que había sido entrenada para clasificación y a continuación, dividir la imagen en una malla, en la cual se iba desplazando un rectángulo de una dimensión determinada para poder ir detectando la posición del objeto en la imagen.

El resultado es que cada vez que se iba desplazando una ventana a través de la imagen, se iba detectando partes del objeto en dicha ventana. Como resultado, tenemos que se va detectando la posición del objeto simplemente desplazando la ventana y viendo si en esa región que delimita dicha ventana, está presente una parte del objeto o no. Esto lo vemos en la Figura 12 en detalle.

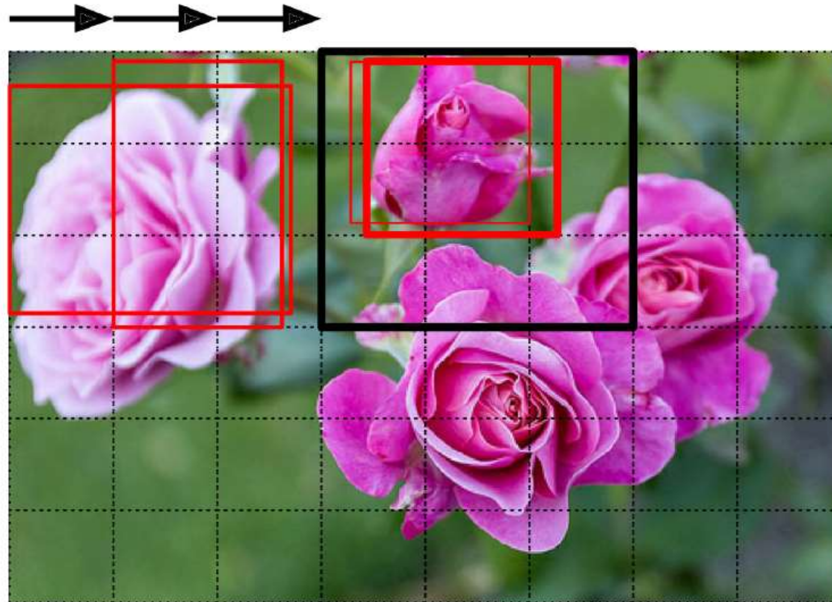

Figura 12: Método de detección de objetos por ventana deslizante. [11]

En suma, veremos cuál es la región donde está el objeto en la imagen, sin más que añadir la suma de las detecciónes que se han producido durante el desplazamiento de la ventana a través de la imagen. Esta una técnica muy robusta pero que implica la detección del mismo objeto múltiples veces en distintas posiciones y por ello, se hace necesario un procesamiento posterior de las cajas de contorno que no son necesarias. Esto es lo que se conoce como modelos de partes deformables (DPM) [11] [12].

2.5.2 R-CNN

La siguiente evolución en la detección de objetos es R-CNN. Utiliza el concepto de la *Region Proposal Network* (RPN). En este modelo, lo primero que se realiza es determinar las regiones de interés dentro de la imagen, es decir, de una manera selectiva determinar dónde está el objeto. Una vez que se determina esta región de interés sobre la imagen, se realiza una clasificación sobre las regiones de interés de dichas imágenes mediante una red ya preentrenada [13].

Esto implica varios algoritmos. En una primera etapa lo que se hace es generar los potenciales cuadros delimitadores (*Bounding Boxes*) en la imagen mientras que, en una segunda etapa, se usa un clasificador para que funcione en dichos cuadros delimitadores. Después de la clasificación se usa un procesamiento posterior para afinar las cajas de detección (*Bounding Box*), para ganar en precisión, puntuar de nuevo el objeto en base a otros que aparecen en la imagen y eliminar cajas de detección duplicadas.

El principal problema de este modelo es que es muy lento. Vemos cómo ejecuta un primer algoritmo para las regiones, después el clasificador y después afina la detección del objeto, con lo que cada etapa consume bastante tiempo.

Para tratar de mejorar este punto débil, se trató de realizar mejoras, creando dos nuevos algoritmos. El primero es Fast R-CNN, que mejora el algoritmo inicial debido a aprovechar algunos recursos como las características que extrae la red neuronal convolucional, mejorando algo el tiempo de entrenamiento y detección, pero no de manera sustancial. El otro algoritmo es Faster R-CNN, que consigue mejorar la velocidad del proceso al integrar en la red neuronal convolucional el algoritmo de propuesta de región ya descrito. Además, usa cuadros de anclaje, que son alturas y anchuras predeterminadas, para poder tener una primera aproximación del tamaño del objeto que se desea detectar (lo veremos más adelante).

2.5.3 Single Shot Detector (SSD)

En este caso, la arquitectura que posee este detector en su red neuronal convolucional es de tipo piramidal, lo que le permite obtener la detección de objetos grandes y pequeños. Concretamente utiliza multiples capas de detección, con lo que consigue múltiples mapas de características, cada mapa para una escala diferente.

La red neuronal convolucional va reduciendo la dimensión espacial gradualmente, y con ello, la resolución de los mapas también disminuye. Con ello, los mapas de menor resolución se utilizan para la detección de objetos de mayor escala, mientras que los de mayor resolución se usan para los objetos pequeños.

La arquitectura de la red de SSD, es la que podemos observar en la Figura 13 de abajo, donde usa la red VGG16 para extraer características y mediante capas de convolución va realizando detecciones.

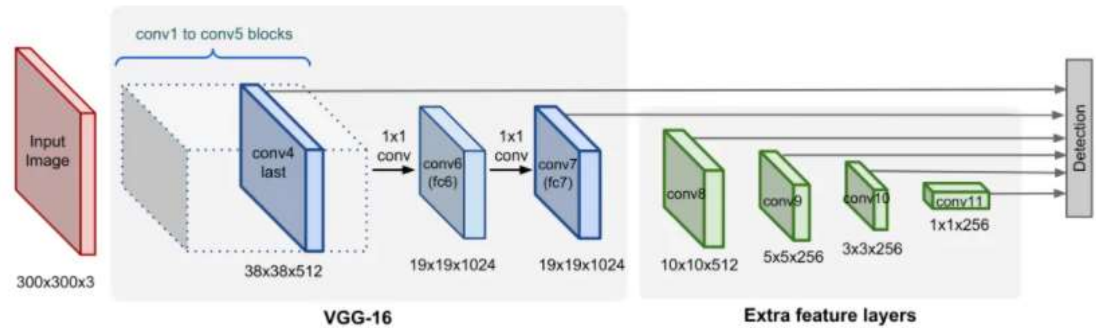

Figura 13: Arquitectura de la red SSD [14]

Las predicciones se componen de un cuadro delimitador y de *N*+1 puntuaciones de clase, siendo *N* el número de clases que puede detectar, más una que no se corresponde a ningún objeto. Además, SSD no utiliza la propuesta de región como pasaba en el caso de R-CNN, sino que va obteniendo las puntuaciones y ubicaciones mediante pequeños filtros de convolución, después de extraer los mapas de características. Después de VGG16, se agregan 6 capas más, en las que 5 de ellas serán para la detección de objetos. Con estos mapas de características multiples se logra una gran mejora de la precisión.

Este modelo consigue una mejora sustancial de la velocidad, con lo que se puede emplear en la detección de objetos en tiempo real [14].

2.5.4 RetinaNet

Se trata de un modelo de detección de objetos que funciona bien tanto para objetos pequeños como para objetos de gran tamaño. Este modelo nace gracias a dos mejoras principales sobre detectores de una única etapa y son la red piramidal de características (FPN, *Feature Pyramid Network*) y la pérdida focal.

La pérdida focal, es una mejora sobre la pérdida de entropía cruzada y se introduce con el fin de terminar con el problema del desequilibrio de las clases con modelos de detección de objetos de una única etapa. Este tipo de pérdida reduce el impacto que tienen los valores que predijo correctamente la red, centrándose así en los casos en los que predijo una clase incorrecta.

Feature Pyramid Network (FPN) surge debido al problema que es detectar objetos en diferentes escalas. Una primera aproximación sería tomar una imagen a diferentes escalas y de cada una de ellas, obtener su mapa de características, lo que se conoce como una pirámide de imágenes. Esto lo podemos ver en la Figura 14 en la izquierda. Sin embargo, computacionalmente es muy costoso y lleva mucho tiempo.

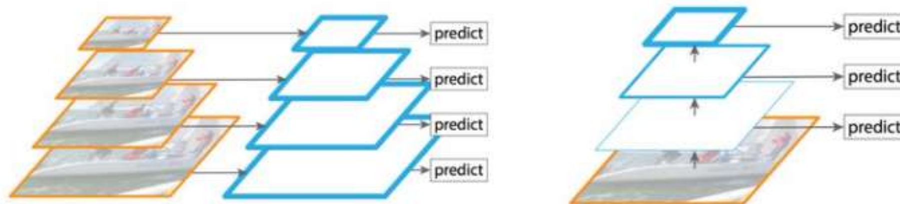

Figura 14: Diferencia entre pirámide de imagenes (izquierda) y pirámide de mapas de características (derecha)

Por ello, lo que se suele hacer es cambiar de enfoque y hacer lo que se ilustra en la Figura 14 derecha, que es crear una pirámide de características y usarla para detección de objetos. Los mapas de características cercanos a la capa de entrada (la de la imagen), suelen componerse de estructuras de bajo nivel que son poco efectivas en la detección de objetos. De aquí sale FPN, que es un eXtractor de características diseñado en forma piramidal de forma que tiene buena precisión y velocidad.

Tiene dos flujos de trabajo. Si vamos de abajo a arriba, es lo que ya conocemos de las redes convolucionales habituales, es decir aplicación de filtros para extraer las características. El flujo de trabajo de arriba a abajo es para construir capas con mayor resolución espacial. Esto ayuda a facilitar el entrenamiento y predecir mejor la ubicación [15].

Con esto podemos obtener una mejor visión de la arquitectura de RetinaNet, que se compone de 4 elementos, como podemos ver en la Figura 15.

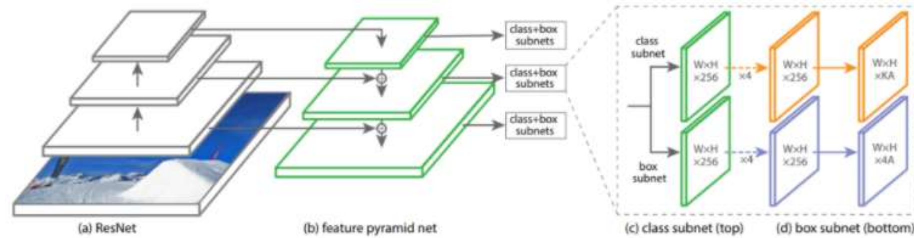

Figura 15 Arquitectura de RetinaNet

Si vamos en orden de izquierda a derecha lo primero que nos encontramos es el camino ascendente, que es la red troncal que calcula los mapas de características en distintas escalas. A la derecha tenemos el camino descendente con sus conexiones laterales, donde se van tomando muestras de los mapas de características que son mas gruesos espacialmente de los niveles superiores de la pirámide, mientras que las conexiones laterales fusionan las capas de arriba abajo y viceversa con el mismo tamaño espacial. Después más a la derecha, encontramos la red de clasificación, que predice la probabilidad de que el objeto esté o no en una posición espacial dada. Por último, encontramos la red de regresión, que devuelve el desplazamiento de los cuadros delimitadores en la predicción, respecto a los *Ground Truth*.

2.6 YOLO (You Only Look Once)

Se trata de una arquitectura de detección de objetos muy rápida y precisa, que fue creada por Joseph Redmond como principal autor [12]. El hecho de que sea una arquitectura muy rápida, la hace idónea para que se use en la detección en video en tiempo real. YOLO consiste en una red neuronal convolucional que predice simultáneamente múltiples cuadros delimitadores y las probabilidades de la clase de objeto que delimitan dichos cuadros delimitadores.

2.6.1 Beneficios

- Es un sistema muy rápido, debido a que se reduce la detección a un problema de regresión lineal y esto conlleva a que no se requiera un *pipeline* complejo.

Simplemente se ejecuta la red neuronal en una imagen en la prueba para hacer predicciones.

- Para hacer las predicciones, a diferencia de los métodos anteriores, este sistema consume la imagen completa, en lugar de regiones de la misma. Esto hace que se limiten los errores a la hora de reconocer las clases de objetos que hay en la imagen.
- Aprende representaciones generalizables de objetos, lo que hace que, si se introducen datos de entrada nuevos, tenga menos probabilidades de fallo que las técnicas descritas anteriormente.

2.6.2 Funcionamiento

El funcionamiento se basa en una red neuronal convolucional, como se comentó previamente, en la que usa características de la imagen al completo para trazar los cuadros delimitadores (*Bounding Box*). Con ello se consigue que la red neuronal reaccione a todos los objetos que hay en una imagen. La imagen se divide en una rejilla de tamaño S x S de forma que, si un objeto cae en una celda de la cuadrícula, esa celda es responsable de detectar ese objeto, como podemos ver en la Figura 16, que en este caso es la casilla central.

Por otra parte, cada celda de la cuadrícula predice B cuadros delimitadores con sus puntuaciones confianza (*score*). Estas puntuaciones de confianza reflejan cómo de fiable es el modelo si esa celda contiene un objeto y también cómo de preciso es al predecirlo. Formalmente definimos la confianza como *Pr (Objeto)*IOU* (como vemos en Ecuación 3). Si no hay un objeto en esa celda, las puntuaciones de confianza deben ser cero. De lo contrario, si hay un objeto en dicha celda, queremos que la puntuación de confianza sea igual a la intersección sobre unión (IOU). Un poco más adelante, lo explicaremos con mayor detalle.

Cada cuadro delimitador consta de 5 predicciones: x, y, w, h, y la confianza. Las coordenadas (x; y) representan el centro del cuadro en relación con los límites de la celda de la cuadrícula. Como podemos ver en la Figura 16, esto se ve en el punto central marcado de color rojo con coordenadas (220,190). Como tenemos que calcularlo en referencia a la celda vemos que, si las celdas tienen un tamaño de 149×149, lo que hacemos es calcular la normalización respecto a la celda, con lo que el valor queda entre 0 y 1 como podemos observar en los cálculos. El caso de la coordenada y, es exactamente el mismo principio.

La anchura y la altura se toman en relación con la imagen completa, es decir, que en este caso la normalización no se toma en relación a la celda sino al tamaño de la imagen. Por tanto, lo que hacemos es tomar el ancho de la detección del objeto (cuadro de color rojo en la Figura 16), y la altura, y la normalizamos respecto al tamaño de la imagen que en este caso es 448×448.

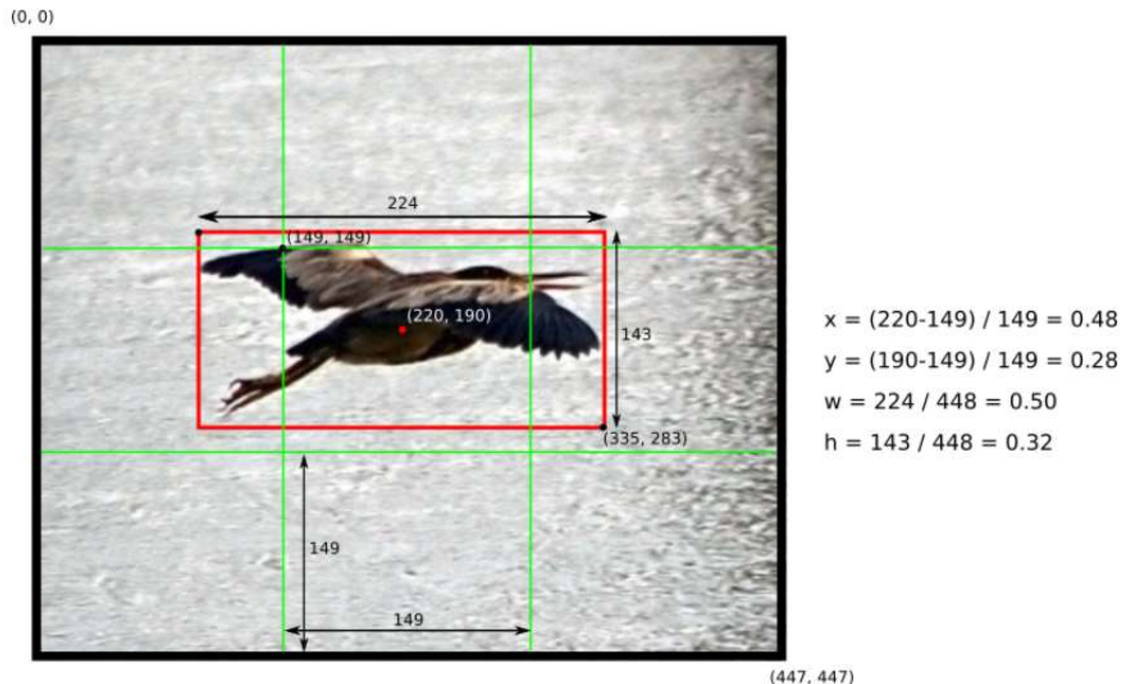

Figura 16: Ejemplo de cómo se calculan las coordenadas del cuadro de una imagen. El tamaño es 448x448 pixeles y S=3. [16]

Finalmente, para obtener la puntuación de confianza de la detección, necesitamos varios parámetros. En primer lugar, se necesita la intersección sobre la unión IOU del cuadro delimitador y la anotación realizada sobre la imagen (*Groundtruth*), que podemos ver en la Figura 17. Como podemos ver, calculamos la IOU (intersección sobre la unión) como el cociente de la intersección entre la unión. Esto nos dará como resultado, la intersección.

Figura 17: Descripción gráfica de la intersección sobre la unión.

Posteriormente, se obtiene la puntuación de confianza, teniendo en cuenta una serie de probabilidades. Cada celda de la cuadrícula también predice *C* probabilidades de clases condicionales, *Pr(Class|Object).* Estas probabilidades están condicionadas en la celda de la cuadrícula que contiene un objeto. Además, predice un solo objeto independientemente del número de cuadros delimitadores *B*. En el momento de la detección, se multiplica la probabilidad condicional de la clase por la probabilidad de un cuadro individual, lo que da las puntuaciones de confianza específicas de la clase para cada cuadro delimitador. Estas puntuaciones de confianza nos dan la probabilidad de que el objeto este en dicho cuadro y cómo de bien se ajusta el cuadro predicho al objeto.

$$\Pr(\text{Class}_i|\text{Object}) * \Pr(\text{Object}) * \text{IOU}_{\text{pred}}^{\text{truth}} = \Pr(\text{Class}_i) * \text{IOU}_{\text{pred}}^{\text{truth}}$$

Ecuación 3 Obtención de la puntuación de confianza del objeto [12]

2.6.3 Arquitectura de red

El modelo se implementa en una red neuronal convolucional, ya que solo contiene capas convolucionales, a menudo se le suele denominar FCN o *fully convolutional network.* En las capas iniciales se extraen las características de la imagen y las capas completamente conectadas predicen las probabilidades de salida y las coordenadas. Esta red está inspirada en el modelo GoogleNet usado en la clasificación de imágenes.

Concretamente esta red se compone de 24 capas convolucionales y 2 capas completamente conectadas. Para reducir el número de capas los creadores usan una convolución 1x1, lo que permite reducir la profundidad de los mapas de características. A esto le sigue una capa de convolución 3x3. Estas capas 1x1 y 3x3 se van alternando como podemos apreciar en la Figura 18. Además, la ultima capa de convolución, genera un tensor con forma (7,7,1024). Finalmente, el tensor se reduce, aplicando 2 capas completamente conectadas, y da como salida un tensor de tamaño 7x7x30, como podemos apreciar en la Figura 18 [12] [17].

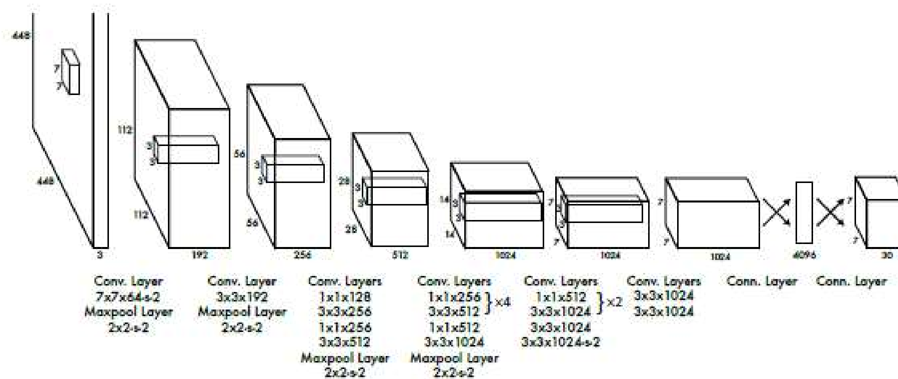

Figura 18: Esquema de la arquitectura de la red neuronal [12]

La función de activación usada es LeakyReLu, salvo en la capa lineal, que usa una capa de activación lineal. En este caso, la función LeakyReLu, transforma los valores introducidos multiplicando los negativos por un coeficiente para rectificarlos y los coeficientes que son positivos no los rectifica, sino que los deja como están [18].

En cuanto a la función de pérdida, en YOLO se predicen muchos cuadros delimitadores (*Bounding Box*) por celda. Para calcular la pérdida de un verdadero positivo, solo se tiene en cuenta uno de los cuadros delimitadores, de todos los posibles y es aquel que al calcular la intersección sobre la unión IOU ofrece el resultado más alto con el cuadro de la anotación (*Groundtruth*) [17].

Para calcular la perdida YOLO usa el error de suma cuadrada (SSE, *Sum Squared Error*) entre los cuadros delimitadores de las detecciones (*Bounding Box*) y los cuadros delimitadores de las anotaciones (*Groundbox*). Con ello, la función de perdida se compone de tres elementos [17].

- La pérdida de clasificación, que es el error al cuadrado de las probabilidades condicionales de clase para cada clase. Esta computa si hay un objeto en la celda, y si no lo hay, es cero. La pérdida viene dada por la expresión de la Ecuación 4 donde $p_i(c)$ es la probabilidad condicional de clase para la clase c en la celda i [17].

$$+\sum_{i=0}^{S^2} \mathbb{1}_i^{\text{obj}} \sum_{c\in\text{classes}} \left(p_i(c) - \hat{p}_i(c)\right)^2$$

Ecuación 4: Pérdida de clasificación del sistema YOLO

- La pérdida de localización, que mide los errores que hay entre las ubicaciones y los tamaños de los cuadros delimitadores de las detecciones y de las anotaciones. Computa cuando el cuadro delimitador de la detección j es responsable de la detección del objeto en la celda i. Es cero en caso contrario. La expresión de esta pérdida la podemos ver en la Ecuación 5 [17].

$$\lambda_{\text{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} (x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2$$
$$+ \lambda_{\text{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left(\sqrt{w_i} - \sqrt{\hat{w}_i}\right)^2 + \left(\sqrt{h_i} - \sqrt{\hat{h}_i}\right)^2$$

Ecuación 5: Pérdida de localización del sistema YOLO

Aquí hay algo importante, que es que no se desea ponderar por igual los errores que se cometen en cuadros delimitadores grandes que en los pequeños, puesto que un error de un número determinado de pixeles no es igual en un cuadro grande que en uno pequeño, la diferencia de error en función del tamaño es considerable [17].

Para evitar esto, lo que se hace es que se calcula la raíz cuadrada del ancho y del alto del cuadro delimitador, en lugar de aplicarlas directamente (sin raíz cuadrada). Además, para poner más importancia en la precisión del cuadro delimitador se multiplica por la pérdida λ_{coord} con un valor predeterminado de 5 [17].

- La pérdida de confianza, siempre que se detecta un objeto en el cuadro delimitador es la que observamos en la Ecuación 6, donde C_i es la puntuación de confianza del cuadro delimitador *j* en la celda *i*

$$+\sum_{i=0}^{S^2}\sum_{j=0}^{B}\mathbb{1}_{ij}^{\text{obj}}\left(C_i-\hat{C}_i\right)^2$$

Ecuación 6: Pérdida de confianza si hay un objeto detectado

Y si no hay un objeto detectado es la que tenemos en la Ecuación **7**.

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

Ecuación 7: Pérdida de confianza si no hay un objeto detectado

λ_{noobj} se refiere a un factor que se coloca porque la mayoría de los cuadros delimitadores no contienen ningún objeto.

Finalmente, la pérdida final, es la suma de todas las perdidas comentadas anteriormente, como podemos ver en la Ecuación 8. Esta es la función que se debe de minimizar a la hora de realizar el entrenamiento, y se compone de las pérdidas de confianza, señaladas en el recuadro rojo; de las pérdidas de clasificación, señaladas en color azul; y de las pérdidas de localización, señaladas en el recuadro verde.

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

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

$$+\sum_{i=0}^{S^2}\sum_{j=0}^{B}\mathbb{1}_{ij}^{\text{obj}}\left(C_i-\hat{C}_i\right)^2$$

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

$$+\sum_{i=0}^{S^2}\mathbb{1}_{i}^{\text{obj}}\sum_{c\in\text{classes}}\left(p_i(c)-\hat{p}_i(c)\right)^2$$

Ecuación 8: Función de pérdidas final.

2.6.4 Virtudes de YOLO

Entre las virtudes de este modelo podemos destacar, en primer lugar, que es un sistema muy rápido, y esto hace que sea la principal alternativa para detección de objetos en tiempo real. Además, las detecciones se realizan a partir de una única red neuronal, pudiéndose entrenar de principio a fin para mejorar la precisión.

YOLO al contrario que los modelos de propuesta de región, entrena y detecta con la imagen al completo, y detecta menos falsos positivos.

2.7 Evolución del sistema YOLO

Vamos a describir, en rasgos generales, las versiones mejoradas del modelo que se han ido sucediendo a lo largo de los años desde su aparición en 2015. Con ello, daremos los aspectos mas interesantes del sistema, o las mejoras mas reseñables que se han ido acometiendo con cada revisión del modelo [19].

2.7.1 YOLOv2

Esta versión, tiene como objetivo mejorar la precisión de manera notable y hacer una detección aún más rápida. En cuanto a las mejoras de precisión, se añade la normalización por lotes en las capas de convolución [19] [20].

La normalización por lotes o *Batch Normalization* se trata de una técnica que consiste en añadir operaciones en el modelo, antes o después de la función de activación de cada capa oculta. A continuación, se normaliza cada entrada, y se realiza un centrado en cero. Finalmente se escala y se cambia el resultado usando dos nuevos vectores de parámetros por capa. Esta operación permite que el modelo aprenda la escala y la media óptimas de cada una de las entradas de la capa [6].

Otro aspecto para destacar es que mejora el clasificador de forma que en la primera versión se entrenaba con imágenes 224x224 y luego se realizaba la detección con imágenes de tamaño 448x448. En esta segunda versión el clasificador también se entrena con imágenes de este último tamaño, mejorando la precisión media (mAP).

Otro de los aspectos que ha mejorado es el hecho de que ahora se usan cajas de anclaje, lo que permite que el sistema pueda detectar más de un objeto por celda, cuando hasta ahora era solamente un objeto por celda. Esto antes era un problema puesto que, con la idea anterior, si en una celda había dos o más objetos, no los podía detectar todos.

Con la idea de los cuadros de anclaje, YOLO puede detectar un gran número de cuadros delimitadores. El cuadro de anclaje no es más que ancho y un alto determinados, con el que se puede predecir el cuadro delimitador de la detección. Esto consigue la detección en relación con el mencionado cuadro de anclaje, en lugar de predecir el cuadro delimitador respecto a la imagen completa. Las formas de las cajas de anclaje no son seleccionadas a mano, sino que las selecciona YOLO de forma que facilite a la red el que aprenda a detectar objetos. El tamaño lo selecciona YOLO mediante el algoritmo de aprendizaje no supervisado *K-Means* [19]. Se trata de un algoritmo que ayuda a organizar los datos según las características que tiene. Lo que hacemos es alimentar el sistema con todos los datos que tenemos y vamos a decir en cuántos grupos o "clusters" queremos que nos los organice. Para poder realizar una explicación más intuitiva, vamos a suponer que queremos realizar una organización de los datos en 2 grupos o "clusters". Para ello en la Figura 19, en el caso a), podemos observar en primer lugar la cantidad de datos que tenemos.

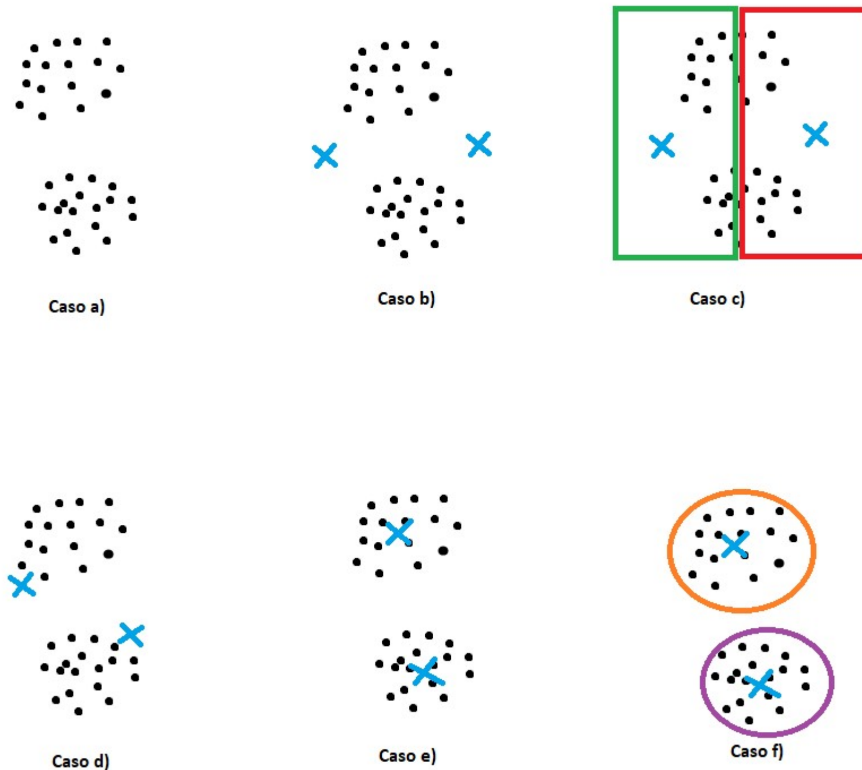

Figura 19: Ilustracion del funcionamiento del algoritmo K-Means

Para poder realizar la organización en dos grupos, el algoritmo crea dos puntos aleatorios llamados centroides, como podemos observar en la Figura 19 en el caso b), donde los encontramos marcados en azul. A continuación, lo que hará el algoritmo es sacar el promedio de los puntos mas cercanos a cada centroide. Supongamos que en el caso c) se marcan que los puntos mas cercanos a cada uno de los centroides son los que aparecen marcados por los cuadros verde y rojo, tanto para el centroide 1 y 2 respectivamente. Una vez el algoritmo saca el promedio de los puntos mas cercanos a cada centroide, el centroide se moverá a dicha posición como podemos ver en el caso d). Esto se repetirá durante varias iteraciones hasta que se llegue a la situación del caso e), donde el algoritmo decidirá que los puntos mas cercanos al primer centroide es el que pertenece al grupo número 1 (marcado en naranja), mientras que los puntos más cercanos al segundo centroide serán los que pertenezcan al grupo número 2 (marcado en morado), como podemos ver en el caso f).

Esto es lo que se realiza con las cajas de anclaje, solo que en este caso es ejecutar el algoritmo que acabamos de comentar ahora varios valores de grupos (k), para generar el IOU promedio con el centroide mas cercano.

Las coordenadas de la ubicación en esta versión de YOLO, se predice teniendo en cuenta las celdas de la cuadricula. Esto implica que en los *GroundTruth* se tengan valores de entre 0 y 1. La red genera hasta 5 cuadros delimitadores para cada celda y predice 5 coordenadas para cada cuadro delimitador: t_x, t_y, t_w, t_h y t_o. Si la celda está desplazada desde la esquina superior izquierda de la imagen por c_x y c_y y el cuadro de anclaje tiene un ancho y un alto dado por p_w y p_h respectivamente, entonces tenemos que las predicciones serán las que podemos ver en la Figura 20. Intuitivamente lo podemos ver con más claridad en la Figura 21

$$
\begin{aligned}
b_x &= \sigma(t_x) + c_x \\
b_y &= \sigma(t_y) + c_y \\
b_w &= p_w e^{t_w} \\
b_h &= p_h e^{t_h} \\
Pr(\text{object}) * IOU(b, \text{object}) &= \sigma(t_o)
\end{aligned}
$$

Figura 20: Predicciones de los cuadros delimitadores

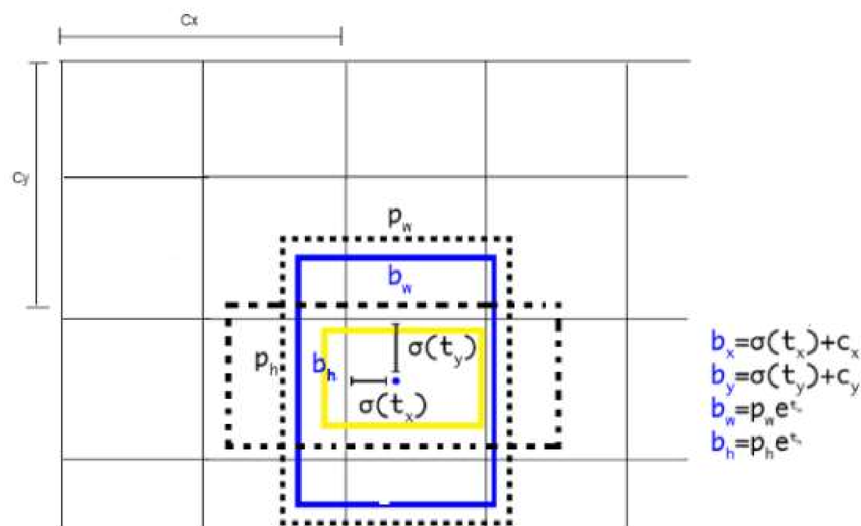

Figura 21: Explicación intuitiva de la posicion de un cuadro delimitador

En cuanto a la arquitectura de red, YOLOv2 es un sistema más preciso y mas complejo debido a que se propone un modelo de clasificación nuevo, denominado Darknet-19. Esta red consta de 19 capas convolucionales y 5 capas de máxima agrupación. El resultado es que es que alcanza una precisión muy notable, mejorando a la de YOLOv1.

2.7.2 YOLO9000

Se trata de una versión concreta que, en lugar de detectar únicamente 20 clases, es capaz de detectar más de 20 clases, concretamente permite la detección de mas de 9000 clases distintas de objetos mediante una buena adaptación entre detección y clasificación. En el caso anterior, teníamos que primero se entrenaba para clasificación y luego para detección porque los conjuntos de datos son distintos para ambos casos. En este caso en YOLO9000 [20] se propone un mecanismo para entrenar con ambos conjuntos a la vez.

En el entrenamiento se mezclan los conjuntos de clasificación y detección, entonces cuando el sistema ve una imagen de detección, se retropropaga la función de pérdidas completa de YOLOv2, pero si el sistema ve una imagen de clasificación, se retropropaga únicamente la parte de la función de pérdidas que tiene que ver con la clasificación.

La mezcla de los conjuntos de datos trae algunos problemas. Primero que los conjuntos de datos para clasificación suelen ser mucho mayores que los de detección. Segundo que los conjuntos de datos de detección suelen ser mas generales, mientras que los de clasificación son mas específicos. Por ejemplo, si para detección tenemos un "perro" como etiqueta, para clasificación podemos tener razas de esos perros como etiquetas, es decir, más específico.

Con esto, se genera un árbol de palabras para fusionar los dos conjuntos. El árbol de palabras consiste en un árbol de estructura jerárquica donde de las clases generales caen las clases específicas. Supongamos por ejemplo que del nodo raíz cuelga un objeto que es perro, que es mas general y se usa para detección, pues de este nodo perro colgarán todas las razas especificas de perros, que serán utilizadas para clasificación.

Una vez realizado esto, lo que se hace es calcular la función Softmax con aquellas clases que están relacionadas o que derivan de una clase determinada. Por ejemplo, si tenemos en cuenta el árbol de palabras que acabamos de explicar, y tomamos una clase general llamada "perro" de la cual derivan elementos relacionados con el mismo como pueden ser "Patas", "orejas", "Bulldog"… Se calculará la función Softmax para la clase general y para las que están relacionadas, que serán las de este ejemplo que acabamos de comentar. Esto hace que se usen varias funciones Softmax, ya que hay varias clases distintas. Esto es importante puesto que a la hora de realizar detecciones, se predice el cuadro delimitador y el árbol de probabilidades de cada clase, pero al usar varias funciones

Softmax, necesitamos atravesar el árbol hasta encontrar la clase en cuestión. Concretamente se parte del nodo raíz, y se va atravesando el árbol por sus distintas ramificaciones siempre tomando la ruta con mayor confianza. Entonces, cuando alcanzamos un nodo con una probabilidad que es menor a la del umbral, encontramos la clase del objeto en cuestión. Esto lo podemos ver en la Figura 22, donde siguiendo esta explicación, se recorre la rama de la izquierda para poder llegar a la detección de un Beagle.

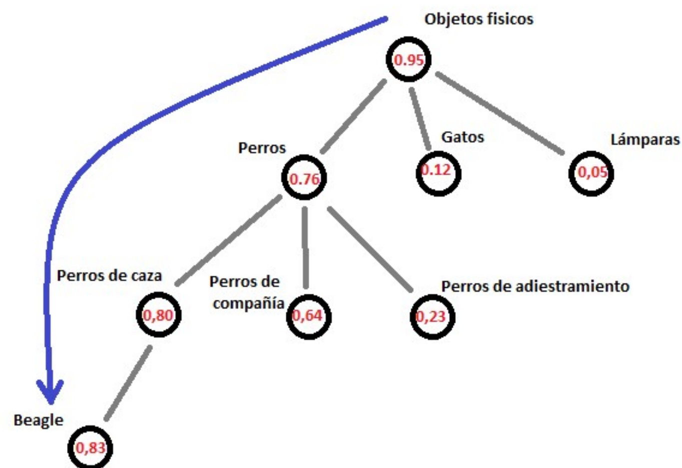

Figura 22: Explicación de la detección con YOLO9000

El detector pasa por los nodos de “Objetos físicos”, “Perros”, “Perros de caza”, y se detendría en “Beagle” donde la probabilidad de la clase sería menor que la del umbral. Si fuese mayor, quiere decir que la clase del objeto que se esta intentando detectar no es “Beagle”, y por tanto, seguiría bajando de nivel en el árbol viendo cual es la clase que tiene mayor probabilidad, y siguiendo el proceso como se ha explicado antes. Esto tiene una ventaja y es que, si por ejemplo el modelo no conoce una clase de objeto dada, pongamos que esta intentando detectar la raza “Foxterrier” cuando solamente conoce o tiene una gran confianza “Beagle” y “Pointer” (ambas clases, pertenecientes a perros de caza), el detector se detendrá en la clase “perros de caza” en lugar la clase “foxterrier” pues no tiene mucha confianza en que clase de perro es. Así, la salida que dará el detector es “perros de caza”.

2.7.3 YOLOv3

Llegamos a la versión de YOLO que hemos utilizado en el proyecto. En ella enumeraremos las evoluciones que ha manifestado respecto a anteriores versiones para

poder apreciar cómo es de diferente el sistema en sí con el paso de las diferentes mejoras que se van aplicando. YOLOv3 [21] nace con el propósito de aumentar en precisión de detección de objetos. Esto hace que se gane en robustez, y esto se debe principalmente a la arquitectura de red usada, que es Darknet-53.

Esta red utilizada para la extracción de características es una red híbrida entre lo que ya se tenía de YOLOv2 y la red residual. El bloque residual, permite que tenga conexiones atajo ya que, en las redes neuronales tradicionales, cada capa se conecta a la siguiente dentro de la arquitectura de capas. Con los bloques residuales, se consigue que cada capa se conecte a la capa inmediatamente siguiente y a capas que están a unos saltos de distancia, lo que hace que este tipo de conexiones, se denominen conexiones atajo.

La implicación de estas conexiones atajo la veremos un poco más adelante. Esta nueva red está compuesta por 53 capas completamente convolucionales, de ahí el nombre de Darknet-53. Como resultado se obtiene una red mucho mas potente que Darknet-19 de YOLOv2, y el rendimiento es similar a otros clasificadores, pero con menos operaciones de coma flotante y mayor velocidad, ya que hace también un uso eficiente de la GPU, por lo que influye en el rendimiento de manera importante [21].

También vemos que hay cambios en la predicción de los cuadros delimitadores. la red predice 4 coordenadas para cada cuadro delimitador, t_x, t_y, t_w, t_h. Si la celda está desplazada desde la esquina superior izquierda de la imagen por (c_x, c_y) y el cuadro delimitador anterior tiene ancho y alto p_w, p_h, entonces las predicciones corresponden a las ecuaciones que aparecen en la Ecuación 9.

$$
\begin{aligned}
b_x &= \sigma(t_x) + c_x \\
b_y &= \sigma(t_y) + c_y \\
b_w &= p_w e^{t_w} \\
b_h &= p_h e^{t_h}
\end{aligned}
$$

Ecuación 9: Transformaciones para conseguir el centro del cuadro delimitador, así como sus dimensiones [21]

Donde vemos que b_x y b_y se corresponden al centro del cuadro delimitador, que es calculado con un offset respecto a la celda donde está el centro del objeto, mientras que b_h y b_w se corresponden con las dimensiones del mismo.

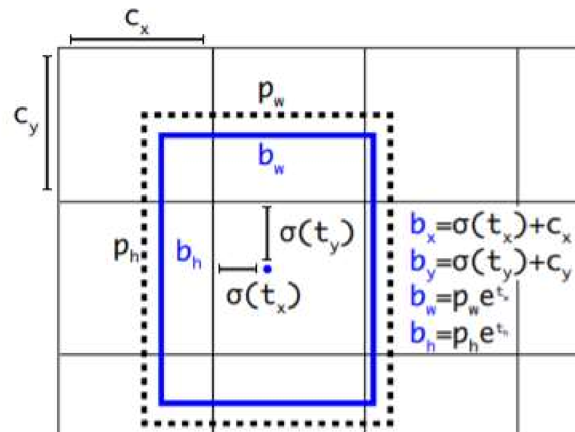

Figura 23 Esquema gráfico de cada uno de los elementos implicados en la obtención de las coordenadas [21]

En esta versión, también se predice una puntuación de confianza para cada cuadro delimitador, que es uno si el cuadro delimitador se superpone a un objeto, es decir, que coincide con el *groundtruth*, más que cualquier otro cuadro delimitador. Si el cuadro delimitador no es el mejor, esto es que no tiene el IOU más alto, pero se superpone con otro *groundtruth* por encima de un cierto umbral, se ignora la predicción. Este sistema asigna solamente un cuadro delimitador por *groundtruth*. Si un cuadro delimitador no se asigna a ningún *groundtruth*, no incurre en ninguna pérdida de coordenadas o predicciones de clase [21].

Otro aspecto que mejora a sus antecesores es el hecho de la predicción de etiquetas múltiples. En algunos conjuntos de datos existen objetos que se pueden etiquetar de distinta forma, por ejemplo, un coche se puede etiquetar como tal pero también como “Berlina”. Esto hace que al usar Softmax para predecir, hace suponer que cada cuadro tiene una clase únicamente, y como hemos visto no es el caso, pues un mismo objeto puede tener varias etiquetas. Para poder utilizar el sistema con múltiples etiquetas, YOLOv3 no usa Softmax, sino que usa clasificadores logísticos independientes para cualquier clase. Con este tipo de clasificadores un coche se puede detectar como “Coche” y como “Berlina” al mismo tiempo. Para las predicciones de clase usa la pérdida de entropía cruzada binaria [20].

YOLOv3, también predice cuadros delimitadores en varias escalas distintas. En las versiones 1 y 2 de YOLO, la salida se predice en la ultima capa, pero YOLOv3, predice cuadros delimitadores en 3 escalas diferentes. Para cada escala se usan 3 cuadros de anclaje y predice 3 cuadros para la celda de la cuadrícula en cuestión [21].

Otra característica importante es el uso de conexiones atajo. Estas permiten obtener un mejor rendimiento en cuanto a la detección de objetos pequeños, puesto que consigue una información mas detallada del mapa de características. Un mapa de características es el resultado de aplicar un filtro a la capa anterior. Con ello se consigue que al pasar el filtro por la capa píxel a píxel, se consigue que se vayan activando las neuronas y el resultado de pasar el filtro por toda la imagen se recoge en el llamado mapa de características [21].

2.7.4 YOLOv4

Mientras se desarrollaba este proyecto, se publicó la actualización de YOLOv3, que es YOLOv4. Se trata de una versión en la que el principal artífice del concepto YOLO, Joseph Redmon ya no es autor, ya que al finalizar la versión 3, cesó su investigación por el posible mal uso que se podía dar a su tecnología, con fines militares o de terceros con nefastos propósitos.

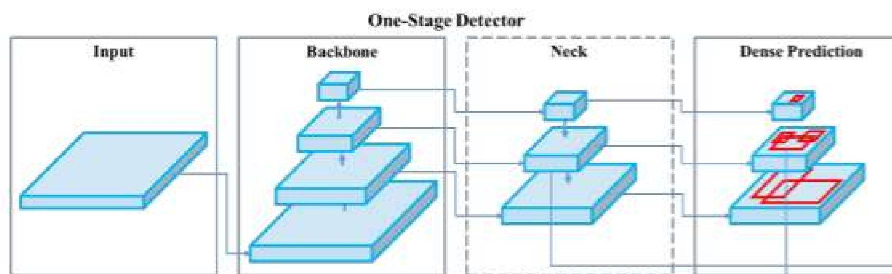

Figura 24: Arquitectura de la red YOLOv4

En esta versión, su motivo principal es optimizar el detector para realizar cálculos en paralelo. Se compone de 3 etapas que podemos observar en la Figura 24:

- *Backbone*: CSPDarknet53, una red que aumenta la capacidad de aprendizaje y que gracias a la unión del módulo de agrupación de pirámide espacial permite mejorar el campo receptivo y distinguir características importantes.
- *Neck*: Módulo de agrupación de piramides espaciales y agregación de ruta PANet, principalmente. PANet se implementa como sustitución de las redes piramidales de características empleadas para la detección en YOLOv3.
- *Head*: YOLOv3

También encontramos dos conceptos que son: *Bag of Specials* y *Bag of Freebees*. *Bag of specials* se refiere a las mejoras que se realizan en el proceso de capacitación como la inferencia de datos, función de coste… pero que no presentan un impacto en la velocidad de inferencia. Mientras que *Bag of specials* se refiere al impacto marginal que tienen en el tiempo de inferencia y el buen rendimiento de las mejoras anteriores. Además, esto mejora el aprendizaje de las características [22] [23].

2.8 Sistemas de detección de objetos YOLO.

En este apartado comprobaremos el funcionamiento de varios sistemas de detección de objetos ya implementados con el objetivo de seleccionar uno que nos pueda servir de utilidad para realizar nuestro studio sobre la arquitectura YOLO. Para ello, comprobaremos los repositorios existentes en la plataforma GitHub, y que puedan funcionar convenientemente.

Es importante destacar que hay una gran cantidad de proyectos creados para realizar una detección de objetos completa, entendiendo como tal, un sistema que sea capaz de poder realizar entrenamientos, detecciones, así como las mismas operaciones, pero sobre un conjunto de datos personalizado o propio. Sin embargo, es importante tener en cuenta que no todos los proyectos que hay en dicha plataforma funcionan a la perfección, pues muchos distan de estar acabados y/o completos, por lo que en muchas ocasiones se antoja necesario realizar correcciones y/o adaptaciones para poder realizar las tareas que pretendemos en este estudio convenientemente.

También es importante destacar el porqué tomar como punto de partida un detector de objetos ya implementado en lugar de realizar una programación partiendo completamente de cero. Y no es otro que el tiempo que llevaría implementar un detector de este tipo. La cantidad de tiempo a emplear sería mucho mayor si deseamos crear un sistema desde cero, que conlleva una tarea exhaustiva de programación y comprobación de funcionamiento y depuración de errores, además de una gran cantidad de tiempo necesaria para obtener un correcto funcionamiento.

Debemos tener en cuenta que hemos realizado el analisis sobre la última versión de YOLO posible, que es la versión 3. Es cierto que en el momento que se comenzó a

desarrollar este proyecto, ya había salido la documentación con la versión 4 de la arquitectura e incluso pudimos encontrar un sistema que se estaba comenzando a desarrollar. Sin embargo, optamos por la versión 3 ya que era la versión más estable hasta la fecha.

Además, es importante que tengamos en cuenta que nos hemos fijado principalmente en proyectos que estuviesen contruidos sobre la biblioteca de aprendizaje automático TensorFlow, aunque como veremos también hemos explorado alguna otra opción. Dentro de los proyectos en los que se usa TensorFlow, nos hemos fijado en aquellos que estuviesen desarrollados con la version 2.X de TensorFlow.

Esto es muy importante ya que hay infinidad de proyectos con la version 1.X, pero en la actualidad se está realizando migraciones de proyectos con esae versión a la 2.X. Es cuestión de utilizar la última versión disponible de la API y trabajar con vistas a futuros desarrollos. Esto trae consigo inconvenientes, ya que la version 2.X de TensorFlow se comenzó a desplegar en 2019, y hacia finales del mismo año se comenzaron a desarrollar proyectos sobre esta nueva versión. En la API oficial de detección de objetos de TensorFlow [24], aún no estaba realizada la migración hacia esta nueva versión al comienzo de este proyecto, lo que implicaba explorar otras opciones que no fueran la oficial, y de ahí la búsqueda de proyectos existentes de terceros. Únicamente nos sirve para entender cómo se realiza la detección de objetos y dar los primeros pasos del aprendizaje de este tipo de tecnología.

A continuación, pasamos a repasar algunos de los repositorios que nos han resultado prometedores a la hora de realizar este estudio, y cuál ha sido la elección en base a las distintas propuestas.

2.8.1 TrainYourOwnYOLO

Se trata de un repositorio creado por Anton Muehlemann, alumno de la universidad de Berkeley y Oxford [25]. En él, se permite realizar una detección de objetos personalizada mediante YOLOv3 y con Tensorflow 2.3 además del uso de la última versión de Keras. Se trata de un repositorio bien estructurado y explicado ya que tiene un fichero de texto en el que se explica las partes de las que se compone el mismo y cada uno de los pasos a seguir para realizar un entrenamiento, una detección, detección después de hacer un entrenamiento con un conjunto de datos personalizado...

Este es uno de los puntos fuertes de este repositorio. Muchos de los repositorios existentes en Internet carecen de una documentación adecuada, lo que los hace inabarcables a nivel de utilización. Además, en ocasiones no especifican si están o no en correcto funcionamiento o si están completos. En este caso se especifica que está terminado, así como las versiones utilizadas en cuanto a programación de los distintos ficheros.

Tambien ofrece distintas posibilidades a la hora de poder ejecutar los distintos ficheros del detector. Esto es que ofrece la posibilidad de que se pueda usar tanto en sistemas operativos Linux, Mac o Windows, con las correspondientes instrucciones para poder instalar debidamente el repositorio, así como la opción de ejecutarlo en línea mediante Colab. Además, cabe destacar que el código se puede editar y redistribuir siempre y cuando se incluya la licencia.

En nuestro caso, hemos realizado una prueba en línea utilizando Google Colab, para lo que ha sido necesario registrarnos para obtener los tokens, que nos permiten usar los codigos que hay en dicha aplicación. Para ello, una vez que accedemos al código por defecto del repositorio en Colab, podemos ir ejecutando cada una de las celdas, en la que primero se descargan todos los ficheros necesarios, así como los distintos conjuntos de datos. En este punto tenemos que introducir también el Token de nuestra cuenta de Colab, que nos permite realizar la ejecución del resto del código.

En este punto, debemos destacar que hemos realizado las pruebas con el conjunto de datos que posee el autor y con el que ha realizado el entrenamiento por defecto en Google Colab, que es un conjunto de datos compuesto por imágenes de caras de gatos, ya que lo que pretende el autor es realizar ese tipo de detección. Sin embargo, además de poder entrenar con conjuntos de datos ya preparados, se proporciona la opción de poder generar los nuestros propios, y mediante las instrucciones oportunas que están dadas en el repositorio, podemos obtener una detección del objeto que deseemos.

Siguiendo con la secuencia de ejecución necesaria para la detección del objeto en cuestión, a continuación se ejecuta un fichero de detecciones, para poder comprobar que todas las dependencias y paquetes funcionan correctamente, para después proceder a

la etapa de entrenamiento. Para ello, se decargan los pesos de YOLO y posteriormente se realiza la conversión de estos, necesarios para la red Darknet, y guardarlos en un fichero de extensión *.h5*.

Seguidamente se ejecuta el entrenamiento que podemos observar en la Figura 25, donde se parte de unas 51 épocas, donde primero se congelan las capas profundas para poder realizar el ajuste y posteriormente se descongelan para poder completar el entrenamiento con los pesos ajustados.

```
Train on 90 samples, val on 10 samples, with batch size 32.
Epoch 1/51
2020-11-10 16:14:36.121361: E tensorflow/core/grappler/optimizers/meta_optimizer.cc:581] layout failed: Invalid argument: Subshape must have c
2020-11-10 16:14:36.144225: E tensorflow/core/grappler/optimizers/meta_optimizer.cc:581] remapper failed: Invalid argument: Subshape must have
2020-11-10 16:14:36.345179: E tensorflow/core/grappler/optimizers/meta_optimizer.cc:581] remapper failed: Invalid argument: Subshape must have
2020-11-10 16:14:36.789665: I tensorflow/stream_executor/platform/default/dso_loader.cc:48] Successfully opened dynamic library libcudnn.so.7
2020-11-10 16:14:38.338591: I tensorflow/stream_executor/platform/default/dso_loader.cc:48] Successfully opened dynamic library libcublas.so.1
1/2 [=============>..............] - ETA: 0s - loss: 10042.37702020-11-10 16:14:42.879011: I tensorflow/core/profiler/lib/profiler_session.c
2020-11-10 16:14:44.463876: I tensorflow/core/profiler/internal/gpu/cupti_tracer.cc:1513] CUPTI activity buffer flushed
2020-11-10 16:14:44.497523: I tensorflow/core/profiler/internal/gpu/device_tracer.cc:223]  GpuTracer has collected 6877 callback api events an
2/2 [==============================] - ETA: 0s - loss: 9442.9590 2020-11-10 16:14:49.636176: E tensorflow/core/grappler/optimizers/meta_optimi
2020-11-10 16:14:49.653657: E tensorflow/core/grappler/optimizers/meta_optimizer.cc:581] remapper failed: Invalid argument: Subshape must have
2020-11-10 16:14:49.814911: E tensorflow/core/grappler/optimizers/meta_optimizer.cc:581] remapper failed: Invalid argument: Subshape must have
2/2 [==============================] - 10s 5s/step - loss: 9442.9590 - val_loss: 7948.7949
Epoch 2/51
2/2 [==============================] - 8s 4s/step - loss: 7318.4443 - val_loss: 6437.7656
Epoch 3/51
2/2 [==============================] - 8s 4s/step - loss: 5944.0283 - val_loss: 5082.7119
Epoch 4/51
2/2 [==============================] - 8s 4s/step - loss: 4852.0991 - val_loss: 4192.7764
Epoch 5/51
2/2 [==============================] - 8s 4s/step - loss: 3811.7573 - val_loss: 3253.4871
Epoch 6/51
2/2 [==============================] - 8s 4s/step - loss: 3093.3745 - val_loss: 2664.0032
Epoch 7/51
2/2 [==============================] - 8s 4s/step - loss: 2520.6936 - val_loss: 2271.7568
Epoch 8/51
2/2 [==============================] - 8s 4s/step - loss: 2014.7615 - val_loss: 1832.2311
Epoch 9/51
2/2 [==============================] - 8s 4s/step - loss: 1649.2892 - val_loss: 1479.5115
```

Figura 25: Ejecución de un entrenamiento del sistema TrainYourOwnYOLO

Vemos como podemos ir viendo el progreso del entrenamiento a lo largo de sus distintas épocas, y cómo las pérdidas van disminuyendo conforme avanza el entrenamiento. Además, permite la visualización del progreso que va realizando en cuanto a minimización de las pérdidas durante el entrenamiento a través de la herramienta Tensorboard. Esto es un menú donde se puede ver de manera más grafica e intuitiva, el progreso realizado durante el entrenamiento.

Una vez terminado el entrenamiento se puede ejecutar el fichero de detección, donde una vez se cargan los pesos ya entrenados, la red comienza a realizar las deteccíónes de las imágenes de entrada. Si tomamos alguna muestra de imágenes, tendremos una idea de cómo funciona este sistema. Si nos fijamos en la Figura 26, tenemos algunas muestras de un conjunto de prueba compuesto por imágenes de caras de gatos, ya que el sistema ha sido entrenado para la detección de tal objeto. Vemos como realiza buenas detecciones en general con un umbral de confianza muy elevado (superior al 90%).

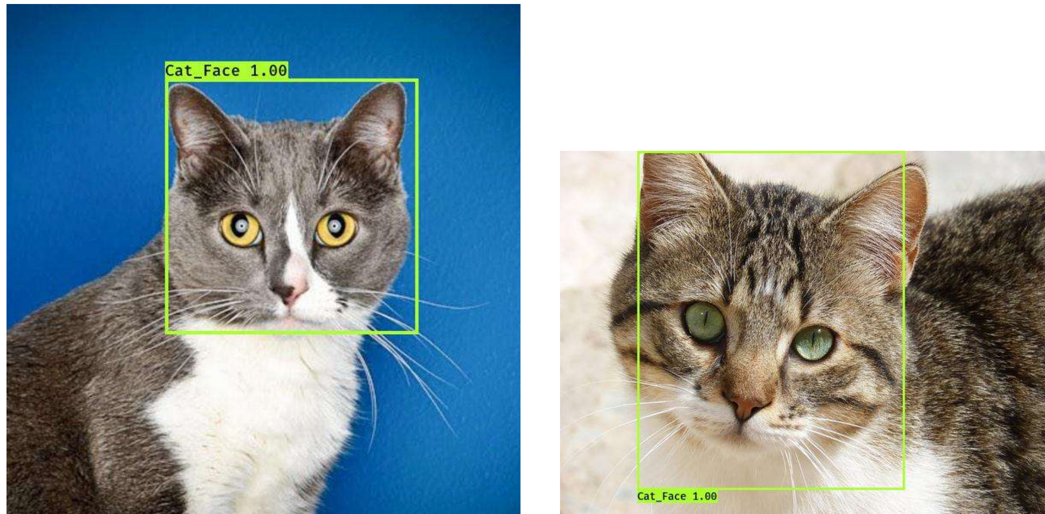

Figura 26: Detecciones correctas de caras de gatos

Sin embargo, como podemos apreciar en la Figura 27, esto queda lastrado por los falsos positivos que también tienen un umbral de confianza abstante alto. Esto es porque el sistema confunde las caras de gatos con las caras de perros, y en algun caso, si hay algun gato en el que no se le ve mirando a cámara, vemos como el sistema no lo detecta. Por tanto, el rendimiento del sistema, para la detección de caras de gatos está bien en algunos casos, pero falla para muchos otros.

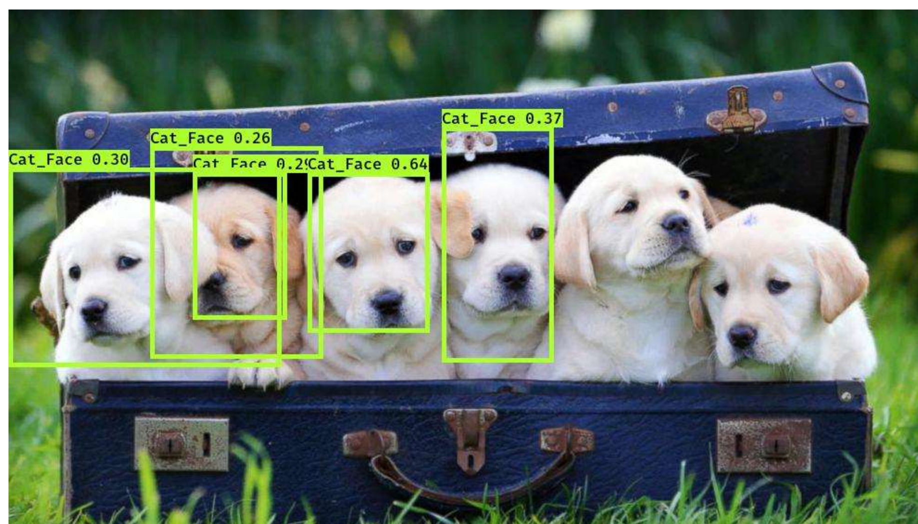

Figura 27 Detección errónea de caras de gato

Un aspecto a comentar es que la ejecución depende del uso o no de GPU, y en caso de realizar su uso, depende del modelo. Por ejemplo, si usamos AWS (ya que este repositorio da la opción de usarlo como soporte adicional) tenemos una GPU Tesla K80 de 12 GB de memoria. Pero si por ejemplo deseamos realizar todo el trabajo en una

maquina local, dependerá del hardware del que dispongamos. En nuestro caso tenemos una NVIDIA 1050Ti con 4GB, lo que hace que tarden mucho más las ejecuciones.

2.8.2 Ultralytics YOLOv3

Se trata de un repositorio creado por Glenn Jocher de Ultralytics, empresa estadounidense, dedicada a soluciones y proyectos de inteligencia artificial [26]. Es un proyecto desarrollado sobre la base de OpenCV que es una biblioteca de visión artificial creada por Intel, y PyTorch que es una biblioteca de aprendizaje automático de código abierto.

En cuanto a los requisitos, vemos que funciona correctamente sobre cualquier sistema operativo, por lo que es un punto a favor teniendo en cuenta de que en muchas ocasiones el desarrollo se realiza sobre un sistema operativo concreto y complica el uso del mismo en otro tipo de sistema operativo. Además, permite el uso de Google Colab para poder realizar detecciónes y entrenamientos. En principio no hemos encontrado problemas de licencia, ya que no indica nada, aunque es conveniente realizar una adecuada citación del repositorio, o permiso al autor si el fin es comercial.

En cuanto al funcionamiento, hemos optado como en el caso anterior, por una prueba en Google Colab, que permita confirmar el buen funcionamiento del repositorio. La ejecución local de todos los ficheros dependerá de los programas instalados, así como los paquetes con la versión precisa para la correcta ejecución, en caso de no ser así puede dar lugar a problemas.

En este caso lo primero que se hace es, como siempre, preparar el entorno de ejecución, y descarga de paquetes. Una vez realizado esto, podemos entrenar el sistema bien con un conjunto de datos propio o bien con un conjunto de datos ya preparado como

es VOC2012[1]. En este caso lo realizamos con este último para poder realizar una prueba del sistema.

Al terminar el entrenamiento podemos observar como realiza las detecciones. Para ello podemos tomar algunas imágenes y podemos observar los resultados. En la Figura 28 vemos como tenemos algunas detecciones en las cuales podemos observar que el sistema trabaja aceptablemente bien.

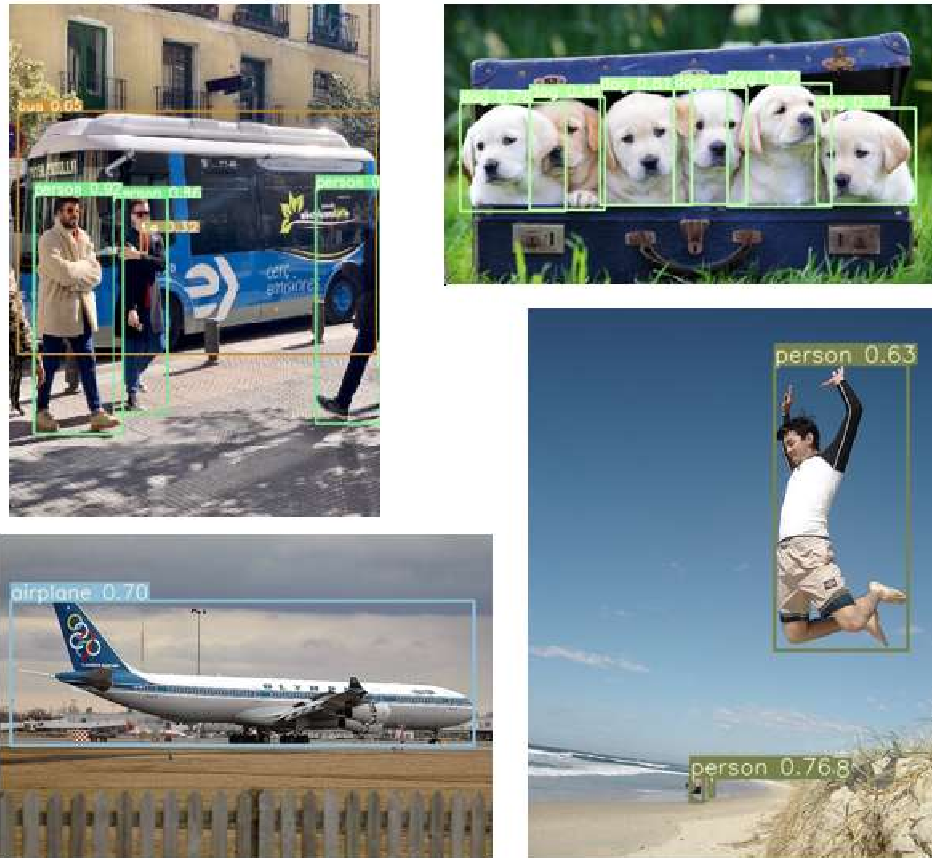

Figura 28: Deteccióones realizadas por el sistema de detección de objetos de ultralytics

En este caso vemos que los objetos que detecta tienen una confianza grande, sin embargo, no es perfecto, ya que en este caso vemos cómo en la imagen de los perros, hay confianzas de menos de 0.5 en algunos objetos, y en el caso de otros objetos como por

[1] Este dataset, se puede encontrar en el siguiente enlace: http://host.robots.ox.ac.uk/pascal/VOC/index.html

ejemplo la persona o el avión, tienen una confianza de entre el 60 y 70% lo que lo que dista de ser un detector perfecto.

También es cierto que en el caso de objetos pequeños también hay una confianza amplia como es el caso de la imagen inferior derecha, donde se pueden apreciar personas caminando por la playa y que el sistema ha logrado detectar con una confianza bastante grande para el tipo de objeto que es, lo que lo hace preciso.

2.8.3 YOLOv3 Implemented with Tensorflow 2.0, Zihao Zhang

Se trata de un repositorio cuyo autor, Zihao Zhang, realizó a finales del año 2019, siendo uno de los primeros en conseguir una implementación que funcionase con las versiones de Tensorflow2.X instaladas en un ordenador particular, como es nuestro caso. Tiene licencia MIT, con lo que podemos realizar el uso de este repositorio siempre y cuando referenciemos y mostremos la licencia de derechos de autor en partes significativas del código utilizado [27].

Finalmente, decidimos optar por este repositorio ya que permitía de una manera muy simple trabajar en local sin crear conflictos excesivos entre requisitos del sistema y los drivers que se debian tener instalados. En los casos anteriores, la utilización de los repositorios en local era simple debido a que existían las instrucciones oportunas para poder realizar cada una de las operaciones, pero los requisitos del sistema junto con que en algún caso no se trabajaba con Tensorflow 2.X, hacia que la mejor opción al realizar pruebas en local fuese ésta.

3

Estudio práctico del sistema YOLO

Para realizar el estudio de la arquitectura YOLO, hemos optado por el sistema de Zihao Zhang [27], que otorga una documentación exhaustiva de cómo poder realizar un uso adecuado del sistema, además de que se trata de un sistema muy intuitivo. Se trata de un sistema que puede detectar objetos que han sido entrenados mediante el conjunto de datos VOC2012, lo que hace que pueda distinguir una gran cantidad de objetos como coches, monitores de televisión, perros, gatos, aviones…

El objetivo en este capitulo, se trata no de ver como funciona el sistema mediante un entrenamiento de un objeto para el cual el sistema ya ha sido entrenado, sino que se trata de ver como reacciona el sistema a un objeto para el cual no ha sido entrenado y ver cómo se comporta ante tal eventualidad. Para ello, nos apoyaremos en el concepto de *Transfer Learning* de forma que usaremos el conocimiento de los objetos que reconoce la red neuronal para poder realizar entrenamientos sobre otros nuevos, y que la red pueda aprender las características de los nuevos objetos para poder evaluar su rendimiento.

3.1 Estructura del sistema de detección de objetos

Una vez esclarecido el sistema de detección que vamos a utilizar, vamos a describir cómo es la estructura de ficheros y los elementos de los que se compone el sistema de detección de objetos. No se trata de una explicación exhaustiva con todo detalle del funcionamiento de éste, sino una explicación genérica que ayude a entender qué contiene el sistema.

Inicialmente disponíamos de los siguientes elementos:

- Carpeta Docs: Contiene un fichero de instrucciones del autor del sistema de detección de objetos para poder realizar los entrenamientos de manera correcta.

- Carpeta Data: Se trata de una carpeta donde vienen los ficheros de clases *COCO*[2] y *VOC*[3]. Estos ficheros se corresponden a los conjuntos de datos COCO y el conjunto de datos VOC2012 respectivamente, y en dichos ficheros podemos encontrar las clases de objeto correspondientes a cada conjunto de datos y que el sistema puede detectar. En esta carpeta es donde además se generan los ficheros *TFRecord* necesarios para el entrenamiento. También trae imágenes de muestra para poder probar el sistema en cuanto a detección. Dentro de la misma carpeta encontramos también una que se refiere al conjunto de datos *VOC2012*, para poder realizar entrenamientos con dicho conjunto de datos.

- Carpeta YOLOv3_tf2: Esta carpeta es donde se almacenan distintos recursos. Aquí están en primer lugar, los ficheros de configuración del sistema que están contenidos en la carpeta *__pycache__,* y además tenemos el fichero *__init__.py* necesario para la correcta ejecución del código, el fichero *dataset.py* que es para la gestión del conjunto de datos, como la carga y la transformación de los datos para que sean consumidos por el sistema de manera correcta, el fichero *models.py* donde se define la red Darknet [28] y las funciones YOLO asociadas para poder crear la red neuronal que es la base del detector de objetos. Las funciones YOLO asociadas tienen que ver con las cajas de detección y la función de pérdidas. Además, encontramos el fichero *utils.py*, que contiene las funciones referentes a la carga de pesos de la red Darknet [28], además de dibujar las salidas en las imágenes detectadas, dibujar la intersección de la unión IOU, hasta la congelación de capas necesaria para el fichero de entrenamiento.

---

[2] Conjunto de datos COCO: https://cocodataset.org/

[3] Conjunto de datos VOC2012: http://host.robots.ox.ac.uk/pascal/VOC/index.html

- Carpeta Tools: se trata de una carpeta donde encontramos ficheros de utilidad *Python*. En nuestro caso concreto los útiles son los de *voc2012.py* que sirve para la generación de los ficheros *TFRecord*, que son necesarios para entrenar. Como veremos después, se necesitan modificaciones para poder realizar un entrenamiento con un conjunto de datos propio, y a dicho fichero le nombramos como *voc2012Custom.py*. También hay ficheros como *visualize_dataset.py*, que permite visualizar de manera secuencial el conjunto de datos, en este caso solo usado para *voc2012*, y no para un conjunto de datos propio.

- Convert.py: Fichero necesario para la conversión de los pesos YOLOv3.

- Train.py: Se trata del fichero de entrenamiento. En él se define cómo se cargan los datos, así como la definición de los optimizadores de cómo se crea el modelo YOLO y cómo se genera la congelación de las capas para el caso de entrenar con transferencia de aprendizaje, así como también la obtención de las métricas, la compilación, entrenamiento y cuestiones adicionales relacionadas con los *callbacks* y la función de entrenamiento (*fit*).

- Carpeta Checkpoints: Se trata de una carpeta donde se van almacenando cada uno de los puntos de entrenamiento al final de cada época. Además, esta carpeta se crea al convertir los pesos *YOLOv3.weights* mediante el fichero *convert.py*.

- Carpeta Logs: Se trata de una carpeta que contiene los registros que se pueden usar para realizar un seguimiento del entrenamiento realizado mediante *TensorBoard*. Dentro están los ficheros que se cargarán en *TensorBoard* tanto para entrenamiento como validación.

- Detect.py: Función que permite realizar la evaluación de las imágenes una a una mediante la introducción de su nombre por teclado.

- Detect_video.py: que es el mismo que el anterior, pero para la detección de vídeo ya que YOLO permite la detección de objetos en tiempo real siendo muy rápido en las deteccióncs. Por ello, se usa en este tipo de circunstancias.

- Setup.py: Que son especificaciones del sistema, junto con referencias del autor.

Este es el sistema inicial, con el que se empezaron a realizar las primeras pruebas. Sin embargo, se debieron realizar nuevas modificaciones o la generación de nuevos ficheros para el propósito que nosotros deseábamos. Por ello, las principales modificaciones realizadas al repositorio inicial fueron las siguientes:

- Splitter.py: se trata de un fichero que permite partir los conjuntos de datos en entrenamiento, validación y test o bien entrenamiento y validación únicamente. Usa una librería que permite realizar operaciones con imágenes anotadas [29] y permite realizar una división tanto de las imágenes como de los ficheros de anotación en estos conjuntos anteriormente mencionados.

- SplitterDogs.py: Es igual que el que acabamos de describir, solo que se trata de una adaptación del fichero para realizar un estudio sobre un conjunto de datos formado por distintas razas de perros, con el fin de que el sistema pueda realizar la detección en función de una raza de perro determinada. Lo veremos más adelante con mayor detalle.

- Detect_all.py: Se trata de una variación del fichero de detección que traía ya el sistema solo que con algunas modificaciones. En el original, detect.py, debíamos de introducir por teclado el nombre de las imágenes que se deseaban detectar. Esto es muy tedioso en el caso de realizar un gran número de evaluaciones de imágenes. Por ello lo que realizamos fue un bucle que permite evaluar una gran cantidad de imágenes de manera automática.

Por otra parte, la puntuación de confianza, la clase de objeto y las coordenadas de éste, salían por ventana de comandos cuando para realizar las evaluaciones de las métricas, que veremos más adelante, se requería que las evaluaciones se fuesen guardando en un fichero por imagen. Esto quiere decir que cada imagen tiene un fichero de detección asociado, en el que se almacenan las mismas detecciones que va realizando el sistema. Todo ello, se va guardando en ficheros con nombre incremental numérico ya que, para realizar las métricas, va a ser de gran utilidad.

- InputDetections: Se trata de una carpeta donde colocaremos las imágenes del conjunto de test para que puedan ser evaluadas por el sistema mediante el fichero detect_all.py.

- OutputDetections: Se trata de una carpeta que recoge las detecciones realizadas por el sistema. Cada una de las imágenes colocadas en *InputDetections*, se evalúan por el sistema, y después son colocadas en esta carpeta. Concretamente dentro de esta carpeta encontramos dos, que son *Files* y *ObjectsDetected*. En *Files*, tenemos los ficheros con las detecciones realizadas, es decir, el objeto detectado, su puntuación de confianza y coordenadas. En *ObjectsDetected*, tenemos las imágenes con las detecciones que se han realizado, es decir con las *Bounding Box*, que enmarcan al objeto detectado.

- CompletCustomPreparation.py: Fichero que permite la preparación de las listas que va a usar el sistema para entrenamiento o validación. Esto es debido a que el sistema tiene las imágenes en la misma carpeta, pero las anotaciones no. Entonces lo que realizamos es una lectura de los nombres que hay en los ficheros de anotación referentes a cada imagen, y con ello preparamos un fichero de lista de nombres de imagen, que permite al sistema decidir si una imagen dada es para entrenamiento o bien para validación.

- YOLO_a_pascal_voc.py: Se trata de un fichero que permite realizar la conversión del formato de *Bounding box* (con coordenadas *x1, y1, x2, y2*) al formato YOLO (*xywh*), en el que pasamos de las coordenadas inferior izquierda y superior derecha, a unas coordenadas donde tenemos el centro del objeto en la imagen en la coordenada *x* y en la coordenada *y*, además de la anchura y altura de la *Bounding Box* relativo al tamaño de la imagen. Para ello, el fichero lo que hace es coger cada uno de los ficheros XML de una carpeta, y parsea los valores referentes a estas coordenadas. Después, aplica unas transformaciones necesarias para alcanzar el formato YOLO y los escribe en un fichero de texto según el formato necesario que necesitamos para obtener las métricas de medición que nos permitirán obtener la precisión del sistema de detección de objetos.

3.1.1 Métricas usadas en nuestro proyecto

Para generar unas métricas de rendimiento que nos permitan evaluar el rendimiento del proyecto de una manera óptima, hemos usado un repositorio de gran interés donde permite realizar una evaluación de los objetos detectados en base a los objetos de test que usamos para las detecciones [10].

Las métricas en las que se basan los autores son el mAP, que es la precisión media y la curva de Precisión-Recall. Esto se debe a que con distintos tipos de competiciones usan distintas métricas.

- PASCAL VOC Challenge usa la curva de Precision-Recall y mAP.
- COCO DetectionChallenge usa hasta 12 métricas distintas, entre las que se encuentra mAP.
- Google Open ImagesDataset[4] V4 Competition, usa mAP.

[4] https://opensource.google/projects/open-images-dataset

- ImageNet Object Localization Challenge[5] define un error por cada imagen considerando la clase y la región de superposición del IOU. El error total, se calcula como el promedio de todos los errores

Ante tanta disparidad de métricas a la hora de realizar evaluaciones de cada conjunto de datos, los autores del repositorio empleado se han centrado en la curva de Precision-Recall y en la precisión media (mAP), de modo que esas serán las que consideremos.

3.2 Detección de un conjunto de datos particular: Clase única compuesta por cuadros de arte

Para el estudio de nuestro sistema lo que vamos a realizar es un entrenamiento sobre un conjunto de datos específico, concretamente de un objeto que no sea capaz de reconocer. Para ello, lo que debemos hacer es tomar imágenes de cualquier objeto y probar detecciones con el sistema. Si encontramos una imagen que contiene un objeto que el sistema no es capaz de detectar, entonces habremos encontrado el objeto sobre el cual debemos realizar el entrenamiento.

En nuestro caso, en la Figura 29 vemos que, al realizar pruebas de detección, hemos obtenido que el sistema no es capaz de detectar los cuadros de arte.

[5] https://www.kaggle.com/c/imagenet-object-localization-challenge

Figura 29 Detección realizada por el sistema sin entrenar previamente

Además, lejos de no detectarlos, en algunos casos, como el que señalamos en la Figura 30, los confunde con otros objetos como por ejemplo monitores de televisión, con lo que el estudio puede ser más concluyente en cuanto a nivel de precisión del sistema y capacidad de detección en caso de lograr un éxito en la detección de dicho objeto.

Figura 30 Detección de un segundo cuadro en el que el sistema confunde con otro objeto

Para poder realizar un entrenamiento con éxito, lo que hacemos por tanto es tomar imágenes de cuadros de arte. Para ello en primer lugar lo que hacemos es buscar un conjunto de datos ya creado de imágenes de cuadros con sus respectivas anotaciones, pero al no poder encontrar ningún conjunto de datos ya creado sobre este tipo de objeto en particular, debemos realizarlo de manera manual.

Para ello lo que hicimos en un primer momento es buscar imágenes de cuadros de manera masiva a través de la red, con el fin de crear un gran número de imágenes con los que poder realizar el entrenamiento, validación y test. Para ello, buscamos en portales como Pixabay [30], donde las imágenes a utilizar son libres de derechos de autor, sin embargo, ante la escasez de estas, debimos recopilar más imágenes a través del buscador de imágenes de Google. Este proceso fue algo laborioso ya que se trataba de descargar un conjunto de imágenes de manera que además de no repetirse ninguna de ellas, debía ser un conjunto amplio. Si bien es cierto, si tuviera que realizar un nuevo proceso similar usaría algún tipo de herramienta de *Web Scrapping* es decir, técnicas para el recopilado de información de distintos sitios web con el fin de agilizar la tarea de recopilación de imágenes, ya que se haría una recopilación de forma automática.

Una vez que realizamos la descarga manual de todas las imágenes, utilizamos un programa que permitía detectar si dos imágenes estaban duplicadas. Este software es *Find.Same.Images.Ok* y es de descarga gratuita[6]. Para poder realizar un entrenamiento aceptablemente bueno, el número de imágenes que debe consumir el sistema debe ser alto. Ahora bien, dado que el etiquetado de los *bounding boxes* se realiza manualmente, decidimos limitarlo a 800 imágenes.

Una vez realizamos la recopilación de imágenes, lo que hacemos es realizar un re-escalado de las mismas. Para ello, lo que hacemos es tomar un código que nos permite el reescalado [31]. Esto lo realizamos para poder tener un tamaño de imagen común ya que el sistema debería consumir las imágenes para entrenar y si son de gran tamaño, el tiempo de entrenamiento será mayor.

Cabe destacar que cuanto más pequeñas sean las imágenes, mejor las consumirá el sistema pues el tiempo de procesamiento de estas, es menor. Sin embargo, tampoco podemos reducir la resolución de estas de manera aleatoria, ya que la imagen será ilegible y el sistema no será capaz de entrenar de una manera correcta, llegando a producirse errores en el caso de que la imagen este muy pixelada. Por tanto, tenemos un compromiso. En un primer momento, se escogió como tamaño de entrada de las imágenes al sistema de 150x150 pixeles, pero la resolución fue muy baja, así que se probó con 300x300 pixeles,

[6] https://www.softwareok.com/?seite=Freeware/Find.Same.Images.OK

con lo que la resolución mejoro notablemente pero no lo suficiente. Como el tamaño por defecto de las imágenes a la red Yolo es de 416x416, finalmente se probó con ese tamaño y se pudo comprobar que era idóneo en el compromiso de eficiencia y calidad de resolución de la imagen. Como comentamos anteriormente, las imágenes no tenían una proporción 1:1, por lo que hubo que reescalarlas.

Una vez realizada esta tarea, procedimos al etiquetado, con lo que debíamos encontrar una herramienta para etiquetar de una forma eficiente las imágenes. En este caso tomamos LabelImg [32] que es una herramienta muy sencilla y gratuita para proceder al etiquetado de las imágenes. En la Figura 31, podemos observar el menú que dispone esta aplicación. Consta de un menú principal donde se selecciona el tipo de fichero de anotación que deseamos, que en nuestro caso será Pascal VOC. Además, permite seleccionar el directorio donde están las imágenes y la opción de etiquetar imagen que al seleccionarla nos permitirá crear una caja que delimitará el objeto. Una vez dibujada la caja, colocamos el nombre de la etiqueta y salvamos los cambios.

Figura 31 Proceso de etiquetado de imagenes en LabelImg

El proceso de etiquetado lleva tiempo, ya que debemos pensar que es un repositorio de 800 imágenes de cuadros de arte, lo que es tedioso. Una vez etiquetadas las imágenes, lo que debemos hacer es realizar una separación de todas las imágenes con sus respectivas anotaciones en conjuntos de entrenamiento, validación y test. Para ello, el porcentaje

aplicado a cada uno de los conjuntos fue de 64% para imágenes de entrenamiento, 16% a imágenes de validación y un 20% a imágenes de test.

Una vez que tenemos esto realizado, lo único que queda es colocar las imágenes en los directorios correspondientes para que al realizar la ejecución de los scripts se realicen sin ningún tipo de error. Para ello creamos un directorio muy similar al que usa el sistema de detección de objetos por defecto, que es la carpeta *VOC2012*. Para ello al lado de dicho directorio, creamos el directorio *Custom*, y la estructura de este directorio será exactamente igual a la del directorio *VOC2012*.

Con ello nos evitamos complicaciones a la hora de editar los scripts del sistema de detección de objetos, para poder editar así lo más esencial para que el sistema funcione correctamente. Los directorios de los que consta son *Annotations* donde están los ficheros de anotación de las imágenes que va a consumir el sistema, *JPEGImages* donde están las imágenes que va a consumir el sistema, e *ImageSets* que es una carpeta donde se almacenan los ficheros de identificación que utiliza el sistema para entrenar.

Esta última carpeta es importante porque almacena ficheros de texto que identifican los nombres de los ficheros de imagen y anotación que se usan tanto para entrenar como para validar. La idea aquí es simple. En las carpetas de anotación e imágenes están todos los ficheros al completo, pero a la hora de entrenar, el sistema toma una imagen para entrenar o validar en función del nombre de la imagen que aparezca en los ficheros de texto de la carpeta *ImageSets*.

Por tanto, encontraremos dos ficheros, que es uno el que contiene los nombres de las imágenes como referencia a aquellas que se van a usar para entrenar, y lo mismo pasa con el fichero de validación. Es decir, la separación del cómputo global de imágenes que consume el sistema se realiza aquí. Todo esto lo conseguimos con la ejecución del fichero *CompletCustomPreparation.py.*

Una vez colocadas las imágenes con la misma estructura de la carpeta *VOC2012*, lo que debemos hacer es crear los ficheros de entrenamiento con extensión *TFRecord* a partir de estos ficheros de texto. Para ello usamos el fichero *voc2012Custom.py* que permite la generación a través de comandos tanto del conjunto de entrenamiento como del de validación. Solo queda una opción más y es el fichero de clases que colocaremos en la carpeta data, llamado *voc2012Custom.txt* en el que colocaremos únicamente el nombre de

las clases con las que se va a entrenar el sistema. En este caso colocaremos una única clase llamada *PictureFramed*, haciendo referencia al nombre que le hemos dado a los cuadros en el proceso de etiquetado.

Para finalizar, accedemos a la web del repositorio para entender el comando que nos va a permitir ejecutar el entrenamiento, a través del script *train.py*. En él especificamos varias opciones como la referencia a los *TFRecords*, la referencia al fichero de clases, y otros parámetros necesarios para que se ejecute correctamente el entrenamiento como el número de clases que se van a entrenar, el modo de entrenamiento que en este caso como vamos a usar pesos *Yolo.v3* debemos seleccionar el modo *fit* para poder realizar un ajuste fino, así como especificar la opción de transferencia de aprendizaje , mediante la opción *transfer Darknet* para poder realizar el uso de los pesos pre-entrenados. Por otro lado tenemos los parámetros necesarios para el entrenamiento como la selección del fichero de pesos *yolo.v3*, el número de épocas y el *batchsize*. Una vez hechas todas estas tareas, ejecutamos el entrenamiento.

Hay que tener en cuenta que tenemos parámetros en el script de entrenamiento como *EarlyStopping* configurados de forma que, en este caso, el entrenamiento se detiene en cuanto las pérdidas de las siguientes 3 épocas no mejoran a la última más baja. También se realizaron pruebas adicionales con valores más altos (10, 20 e incluso 40), pero sin obtenerse mejores resultados.

```
C:\Windows\system32\cmd.exe - python  train.py --dataset ./data/Cuadros_train.tfrecord --val_dataset ./data/Cuadros_val.tfrecord --...
 2/86 [..............................] - ETA: 2:04 - loss: 16.0057 - yolo_output_0_loss: 0.5704 - yolo_output_1_loss: 0.
 3/86 [>.............................] - ETA: 1:38 - loss: 15.8917 - yolo_output_0_loss: 0.4801 - yolo_output_1_loss: 0.
 4/86 [>.............................] - ETA: 1:24 - loss: 15.8201 - yolo_output_0_loss: 0.4083 - yolo_output_1_loss: 0.
 5/86 [>.............................] - ETA: 1:16 - loss: 15.9950 - yolo_output_0_loss: 0.4144 - yolo_output_1_loss: 0.
 6/86 [=>............................] - ETA: 1:10 - loss: 15.9709 - yolo_output_0_loss: 0.4215 - yolo_output_1_loss: 0.
 7/86 [=>............................] - ETA: 1:06 - loss: 15.9342 - yolo_output_0_loss: 0.4139 - yolo_output_1_loss: 0.
 8/86 [=>............................] - ETA: 1:03 - loss: 16.0395 - yolo_output_0_loss: 0.4164 - yolo_output_1_loss: 0.
 9/86 [==>...........................] - ETA: 1:00 - loss: 16.0032 - yolo_output_0_loss: 0.4109 - yolo_output_1_loss: 0.
10/86 [==>...........................] - ETA: 58s - loss: 15.9901 - yolo_output_0_loss: 0.4310 - yolo_output_1_loss: 0.7
11/86 [==>...........................] - ETA: 56s - loss: 15.9894 - yolo_output_0_loss: 0.4548 - yolo_output_1_loss: 0.7
12/86 [===>..........................] - ETA: 54s - loss: 16.1093 - yolo_output_0_loss: 0.4629 - yolo_output_1_loss: 0.8
13/86 [===>..........................] - ETA: 52s - loss: 16.0854 - yolo_output_0_loss: 0.4637 - yolo_output_1_loss: 0.8
14/86 [===>..........................] - ETA: 51s - loss: 16.1580 - yolo_output_0_loss: 0.4672 - yolo_output_1_loss: 0.9
15/86 [====>.........................] - ETA: 50s - loss: 16.1776 - yolo_output_0_loss: 0.4544 - yolo_output_1_loss: 0.9
16/86 [====>.........................] - ETA: 48s - loss: 16.1442 - yolo_output_0_loss: 0.4494 - yolo_output_1_loss: 0.9
17/86 [====>.........................] - ETA: 47s - loss: 16.1937 - yolo_output_0_loss: 0.4502 - yolo_output_1_loss: 0.9
18/86 [=====>........................] - ETA: 46s - loss: 16.1496 - yolo_output_0_loss: 0.4351 - yolo_output_1_loss: 0.9
19/86 [=====>........................] - ETA: 45s - loss: 16.1083 - yolo_output_0_loss: 0.4226 - yolo_output_1_loss: 0.9
20/86 [=====>........................] - ETA: 44s - loss: 16.0800 - yolo_output_0_loss: 0.4154 - yolo_output_1_loss: 0.9
21/86 [======>.......................] - ETA: 43s - loss: 16.0510 - yolo_output_0_loss: 0.4071 - yolo_output_1_loss: 0.9
22/86 [======>.......................] - ETA: 42s - loss: 16.0242 - yolo_output_0_loss: 0.4000 - yolo_output_1_loss: 0.9
23/86 [=======>......................] - ETA: 41s - loss: 15.9980 - yolo_output_0_loss: 0.3930 - yolo_output_1_loss: 0.8
24/86 [=======>......................] - ETA: 41s - loss: 15.9761 - yolo_output_0_loss: 0.3865 - yolo_output_1_loss: 0.8
25/86 [=======>......................] - ETA: 40s - loss: 15.9480 - yolo_output_0_loss: 0.3774 - yolo_output_1_loss: 0.8
26/86 [========>.....................] - ETA: 39s - loss: 15.9540 - yolo_output_0_loss: 0.3973 - yolo_output_1_loss: 0.8
27/86 [========>.....................] - ETA: 38s - loss: 15.9393 - yolo_output_0_loss: 0.3988 - yolo_output_1_loss: 0.8
28/86 [========>.....................] - ETA: 37s - loss: 15.9207 - yolo_output_0_loss: 0.3917 - yolo_output_1_loss: 0.8
29/86 [=========>....................] - ETA: 36s - loss: 15.9141 - yolo_output_0_loss: 0.3950 - yolo_output_1_loss: 0.8
30/86 [=========>....................] - ETA: 36s - loss: 15.9805 - yolo_output_0_loss: 0.3934 - yolo_output_1_loss: 0.8
982 - yolo_output_2_loss: 3.8261
```

Figura 32: Progreso de un entrenamiento

Una vez que hemos completado el entrenamiento cuyo progreso podemos apreciar en la Figura 32, procedemos a realizar las detecciones con los ficheros que hemos

preparado para testear el sistema. Para ello, lo que hacemos es tomar el fichero *detect_all.py*. Se trata de un fichero modificado a partir del original del sistema de detección de objetos. En el fichero original, se escribía por teclado a través de la ventana de comandos el nombre de la imagen sobre la cual deseábamos realizar una detección y el sistema sacaba como salida la imagen son la detección en caso de que la hubiese, y por la ventana de comandos la clase, la puntuación de confianza del objeto, y las coordenadas sobre los que el objeto está en la imagen.

En nuestro caso, para poder evaluar posteriormente el rendimiento, debíamos realizar modificaciones. En primer lugar, debíamos detectar todas las imágenes que hubiera en una carpeta, por lo que realizamos esa modificación. Además, las coordenadas del objeto junto con la clase y la puntuación de confianza, las debíamos de recoger en ficheros que correspondiesen a cada imagen. Si tenemos 100 imágenes necesitaríamos 100 ficheros con todos los objetos que se hubiesen detectado. Esto será de gran utilidad en el momento que lleguemos a evaluar el rendimiento del sistema.

3.3 Resultados obtenidos del conjunto de datos de los cuadros de arte

Una vez realizadas las modificaciones oportunas vamos a ver los resultados más significativos que hemos obtenido en base a modificaciones en distintos hiperparámetros con el fin de mejorar el rendimiento del sistema lo máximo posible. Para ello realizamos un entrenamiento con el conjunto de datos que hemos realizado. Debemos remarcar que el entrenamiento lo hacemos con aprendizaje por transferencia (*Transfer Learning*) y que permite aprovechar los pesos de la red neuronal, para aprender nuevos objetos.

Esto permite reducir el tiempo de entrenamiento considerablemente puesto que no es necesario que la red aprenda un objeto completamente desde cero. Para ello, en el entrenamiento colocamos en el comando de ejecución el modo *fit* (ajuste) y en la opción *transfer*, que es para la transferencia, colocamos *Darknet*, teniendo en cuenta la red *Darknet* sobre la que está funcionando YoloV3.

Una vez que termina el entrenamiento, gracias a la configuración del parámetro *EarlyStopping* antes mencionado, ejecutamos las detecciones de las imágenes de prueba.

Vimos que no había ninguna detección, por lo que se hizo necesario realizar algunas modificaciones en el sistema para mejorar el rendimiento del mismo. Para ello pensamos en los hiperparámetros que se podrían modificar para poder obtener una mejor respuesta del sistema ante las detecciones de cuadros.

3.3.1 Umbral de confianza a 0.2

El primer hiperparámetro en el que pensamos fue el *Threshold score*. Se trata del umbral de confianza que permite determinar si una detección es legítima o no, en el sentido de que si puede dar lugar a un verdadero positivo o falso positivo. Si lo incrementamos, permite que se detecten solo los objetos que están por encima del umbral de confianza lo que hace que el sistema solo detecte aquellos objetos que son muy claros en las imágenes, es decir, de los que el sistema está seguro del objeto que está detectando. Sin embargo, si lo bajamos, el sistema no detectará los objetos que tienen una gran puntuación de confianza, sino que aparecerán también objetos con una puntuación de confianza menor. En la Figura 33 podemos ver exactamente eso, puesto que en el primer caso vemos que el umbral de confianza (*Threshold Score*) es de 0.5 y en el segundo caso es de 0.2 con lo que vemos que en el primer caso no lo ha detectado y en el segundo sí.

Figura 33 Detección de un cuadro con umbral a 0.5 y a 0.2

¿Por qué no hemos tomado un umbral de confianza mayor a 0.2? La razón es muy sencilla. Hemos ido bajando el umbral de confianza de 0.5 a 0.4, 0.3… de forma que

hallamos una relación entre el número de detecciones que hace el sistema y el umbral de confianza que colocamos. Si recordamos el análisis que realizamos de las características *Precision-Recall*, veíamos que, si bajábamos el umbral, conseguíamos un sistema mucho más sensible en las detecciones, restando precisión en las mismas.

Esto hace que, a menor umbral de confianza, la sensibilidad es mayor y, por tanto, hace que se detecten muchos más cuadros. Además, debemos tener en cuenta que la confianza con la que se detectan los cuadros es menor de 0.5 en todos los casos de este experimento, lo que hace que, si colocamos el umbral a ese valor o a uno mayor, no haya ninguna detección.

Bajando el umbral a 0.2 se detecta una gran cantidad de cuadros, muchos más que en el caso del umbral a 0.4 y del umbral a 0.3, por ello hemos escogido mostrar este caso.

Figura 34: Selección de detecciones de cuadros realizadas por el sistema

Como podemos observar en la Figura 34, las detecciones realizadas, son bastante aceptables, si bien distan de ser perfectas. En el primer caso sí que vemos que es una buena detección al igual que en la segunda imagen. Sin embargo, en el caso de la tercera y cuarta imagen se cometen ligeros errores. Este es uno de los riesgos de hacer que el sistema sea

muy sensible, en lugar de preciso. Trata de detectar los objetos con suma facilidad, pero sacrifica la precisión, y por tanto puede dar lugar a errores. Ahora bien, estos ejemplos ponen de manifiesto una dificultad existente en el caso de la detección de cuadros y es que los marcos asociados a los cuadros pueden ser muy diferentes. Además, hay fotos de cuadros que incluyen la pared sobre la que está colocado el cuadro y otras no. Pensamos que ésta es una causa que dificulta obtener valores altos de IOU y por ello es necesario emplear umbrales bajos.

El sistema es capaz de realizar múltiples detecciones, pero que no es demasiado preciso, puesto que falla bastante en la detección de algunos cuadros y otros no los detecta. Además, en cuanto a los falsos positivos, vemos que hay una mala detección de otros objetos que no son cuadros pero que los detecta como tal. Para poder observar este hecho, hemos tomado monitores de televisión como podemos apreciar en la Figura 35, algunos de los cuales están reproduciendo imágenes, y vemos como el sistema en algunos de ellos, los detecta como cuadros de arte y no como monitores de televisión.

Figura 35: Detección de un falso positivo de una imagen de monitor de televisión

El mAP que ha arrojado para esta clase es de casi un 20%. Debemos tener en cuenta que hay algunas detecciones que no se han realizado, porque no las ha reconocido el sistema, y por otra parte, hay detecciones que las ha realizado mal como acabamos de comentar anteriormente. Eso hace que junto a los valores de confianza que son como mucho de 0.5, hace que se la precisión media baje bastante.

Si por ejemplo hubiésemos querido colocar un umbral de confianza de 0.3, vemos que el número de detecciones hubiera sido menor que en este caso, y que también lo hubiésemos visto en las gráficas de Precision-Recall que no salen como desearíamos. Además, la precisión media (mAP), baja porque los cuadros que el sistema intenta detectar y cuya confianza es menor que el umbral, no se detectan, y con ello, el número de detecciones es menor.

3.3.2 Umbral de confianza a 0.2 y Learning Rate de 1e-4

Tratando de buscar mejoras en el sistema, intentamos bajar la tasa de aprendizaje para intentar encontrar el mínimo de la función de coste de una forma más precisa, lo que hace un entrenamiento mucho más largo. En este caso, respecto al anterior, notamos que las detecciones son ligeramente peores, puesto que donde antes había cuadros que se detectaban bien, ahora no se detectan y además hay detecciones redundantes es decir se detecta el mismo cuadro varias veces como podemos ver en la Figura 36.

Figura 36: Doble detección de un cuadro en el caso de umbral de confianza de 0,2 y bajando lr a 1e-4

Vemos como en este caso, un cuadro que antes estaba bien detectado (en el caso de la imagen de la izquierda), ahora ya no lo está, pues lo detecta, pero de manera redundante. Esto añadido a que se producen menos detecciones hace pensar que el bajar la tasa de aprendizaje no es lo mejor.

En el análisis de las métricas, vemos como los resultados no son buenos. Por ejemplo, la precisión media ha bajado casi un 9% con lo que los resultados son peores.

3.3.3 Umbral de confianza a 0.2 y Optimizador SGD

Tomamos otro camino en el intento de mejorar los resultados y en este caso tenemos que podemos cambiar la función de optimización. Seleccionamos el SGD (*Stochastic Gradient Descent*) que es un optimizador que es una variante del descenso del gradiente, que en lugar de trabajar con todo el conjunto de datos a la vez (o un conjunto de los mismos) realizando los cálculos para dar lugar a una actualización de los pesos de la red, lo que hace es realizar los cálculos sobre un solo dato para actualizar los pesos (y repetir esta acción iterativamente sobre todos los datos). Cambiando el optimizador, podemos encontrar una mejoría en los resultados.

Con ello, probamos a entrenar y realizar las detecciones, y lo que nos encontramos es también una peor respuesta. Aún peor que el caso anterior, porque aquí sí que detecta cuadros, pero lo hace en un porcentaje notablemente menor. Eso sí las pocas detecciones que realiza las no realiza correctamente, ya que es bastante impreciso, lo que hace pensar que tampoco es el camino adecuado para realizar el ajuste. Por ejemplo, vemos en la Figura 37 que la detección es bastante imprecisa además de tener una confianza muy baja.

Figura 37: Detección de un cuadro con los parámetros de umbral a 0,2 y optimizador SGD

Si nos fijamos en el caso de la gráfica de Precision-Recall en la Figura 38, vemos como los resultados distan mucho de acercarse al ideal como hemos explicado anteriormente, además el mAP es muy bajo, aunque en línea con el caso anterior.

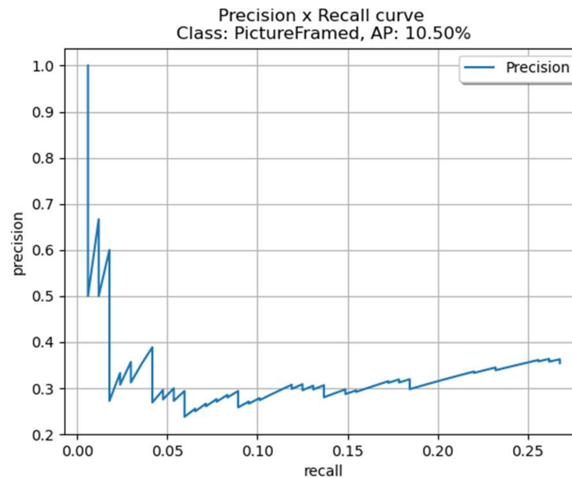

Figura 38: Curva de Precision-Recall con umbral de 0,2 y optimizador SGD

3.4 Detección de un conjunto de datos particular. Tres clases diferentes de un objeto que ya reconoce. Detección de razas de perros.

En este apartado vamos a realizar un estudio sobre un conjunto de datos extraído de la base de datos de Kaggle [33] y se trata del conjunto de datos de perros de la universidad de Stanford [34]. Se trata de un conjunto de datos de hasta 120 razas distintas de perros, que fue ideado inicialmente para problemas de clasificación.

La gran ventaja de este conjunto de datos respecto a muchos otros que también fueron examinados es que en este caso las imágenes estaban ya etiquetadas de la misma forma que habíamos etiquetado el caso de los cuadros, ya que presentaban ficheros *.xml* de anotación junto con cada imagen. Esto simplificaba mucho las cosas puesto que solamente debíamos de colocar los ficheros de anotación y las imágenes en nuestro sistema, para que pudiésemos ejecutar los scripts necesarios para generar los ficheros que se emplearían en el entrenamiento.

Sin embargo, en muchos otros conjuntos de datos, las anotaciones, no estaban en este formato, por ejemplo, aparecían en formato JSON, o incluso, solamente aparecían las imágenes, teniendo que realizar nosotros mismos las anotaciones. Esto implica o bien realizar las anotaciones de un gran número de imágenes, lo que conlleva tiempo, o bien en

el caso de tomar ficheros JSON, habría que cambiar totalmente la programación de muchos de los scripts ya utilizados, con lo que el tiempo se habría ampliado considerablemente. Por ello, tomamos la opción de usar este conjunto de datos.

Dado que es un número muy grande de razas de perros, y que puede dar lugar a una elaboración del estudio demasiado compleja, hemos decidido tomar como muestra 3 razas distintas de perro para poder hacer un estudio más eficiente tanto en tiempo como en resultados. Por ello, tomamos las razas de Afgano, Beagle y Golden Retriever, en las cuales en la Figura 39 podemos observar de qué tipos de perros estamos hablando.

Figura 39: Razas de perro a detectar. Afgano, Beagle y Golden Retriever respectivamente

Con estas 3 razas, lo que hacemos en un primer lugar es realizar una copia tanto de los ficheros de anotación y las imágenes en una carpeta aparte, para poder realizar las divisiones de los conjuntos en entrenamiento, validación y test.

Para ello, una vez que los tenemos ya copiados, mediante el script *spliterDogs.py* realizamos las divisiones. Este script es igual que el ya comentado anteriormente *spliter.py*, con la salvedad de que en este caso tenemos que realizar varias divisiones a la vez. Esto es porque las divisiones en cada uno de estos tres conjuntos lo realizamos en cada una de las razas, y una vez realizadas las divisiones, lo que hacemos es juntar los datos de entrenamiento, validación y test de cada una de las razas. Así conseguimos que las divisiones sean lo más equitativas posible, y es lo más equitativo dado que cada una de las razas no tiene el mismo número de imágenes para que el sistema las pueda consumir.

Una vez, realizado esto, lo que hacemos es colocarlo en la carpeta *Custom* de nuestro sistema dentro de *data*. Y allí colocamos las imágenes en la carpeta *ImageSets*, mientras que las anotaciones van en *Annotations* pero dividiendo en entrenamiento y

validación. Una vez hecho esto, lo que queda es ejecutar el resto de scripts. El primero, *CustomCompletPreparationDogs.py* que es el mismo que en el caso de *CompletCustomPreparation.py*, solo que se hacían necesarias algunas modificaciones dado que, en los ficheros de anotación de los perros, encontrábamos que el nombre del fichero no tenía la extensión ".jpg", entonces al parsear los datos, daba error. La solución es arreglar con unas líneas de código, este problema y una vez solventado, ya se puede ejecutar los ficheros relativos a la generación de los ficheros *TFRecord*, y después ya podemos realizar el entrenamiento. En este caso el fichero de clases debe tener las 3 clases de los perros con los que vamos a alimentar el sistema.

Una vez hecho esto, programamos el entrenamiento, de forma que colocamos que el número de clases a entrenar es 3 y colocamos un número arbitrario pero elevado de épocas, para que pueda ejecutarse el mecanismo de *EarlyStopping* en el momento que no mejore las pérdidas de validación por 3 épocas consecutivas al menos (también se realizaron pruebas fijando dicho parámetro a 10, 20 y 40, pero sin obtener mejores prestaciones). Todo ello con la opción *fit* para realizar el ajuste mediante transferencia de aprendizaje.

3.5 Resultados obtenidos para el conjunto de datos de razas de perro

En un primer momento realizamos un entrenamiento sin ningún tipo de modificación, en el cual dejamos los parámetros de la red y del fichero de entrenamiento sin modificación alguna a la espera de ver como eran los resultados, y a partir de ahí, realizar modificaciones si las hubiera. En cuanto recibimos las imágenes ya detectadas podemos ver que el sistema identifica las imágenes correctamente sin fallos, pero a costa de identificar aquellas cuyo umbral de confianza es mayor a 0.5. Por ello vemos pocas imágenes etiquetadas. Si vemos la precisión media mAP, vemos que en este caso es de alrededor del 16% para el afgano, lo que quiere decir que la precisión es muy baja.

Para el caso del Beagle, la precisión media es menor, cercana al del 14,93%. Aquí también tiene que ver el hecho de que haya más imágenes para entrenamiento de una raza de perro que de otra, pues en el caso del afgano que hay 152 para entrenar (239 imágenes

en total, para los conjuntos de entrenamiento, test y validación), por las 124 que hay para el entrenamiento de la raza Beagle (195 imágenes en total para todos los conjuntos). Sin embargo, la diferencia es mínima.

Por último, encontramos el caso del Golden Retriever, que es el que peor resultados arroja. En este caso tenemos un número de imágenes para entrenar de 104 (150 imágenes en total para todos los conjuntos). Con ello, el sistema sólo ha detectado una imagen de Golden Retriever, lo que hace que la confianza del resto de detecciones esté por debajo de 0.5. Esto quiere decirnos que para el caso del Afgano, el sistema es más preciso que en el caso del Beagle y por supuesto que en el caso del Golden Retriever. La precisión media de esta clase es del 3,12% que viene dada por las pocas detecciones que ha realizado el sistema.

Finalmente, la precisión media conjunta, teniendo en cuenta las 3 clases es del 11,57%, es decir, un valor muy bajo.

3.5.1 Caso de Umbral de confianza a 0.3

Con la configuración anterior vemos que el sistema no falla en detecciones, pero no detecta demasiados objetos puesto que la confianza de detección de los mismos es muy baja e inferior al umbral de 0.5. Así que en la primera aproximación lo que debemos hacer es bajar el umbral para ver como realiza las detecciones. Por ello pasamos el umbral de confianza del sistema de 0.5 a 0.3. Esto lo encontramos en el fichero *models.py*, y el parámetro se denomina *Score Threshold*. Una vez realizamos eso, convertimos los pesos mediante *convert.py* y volvemos a ejecutar el entrenamiento.

Una vez acaba de entrenar, realizamos las detecciones mediante *detect_all.py*, y obtenemos tanto las imágenes como los ficheros que resultan de las detecciones. En el caso del afgano, la precisión crece hasta el 27% lo cual mejora los resultados aparentemente. En el Beagle, vemos cómo los resultados parecen mantenerse y no hay mejora alguna, siendo una precisión muy parecida al caso anterior, en este caso de entorno al 15%. En el caso del Golden Retriever, la situación empeora y se consigue solamente un 0,78% de precisión.

Además, cabe destacar que hay falsos positivos porque etiqueta algunos Golden Retriever como afganos, como podemos ver en la Figura 40.

La razón por la que hay falsos positivos puede atender a que en el caso del conjunto de entrenamiento de los Golden Retriever es menor que en el caso de las otras dos razas, y además en la figura, el sistema confunde la imagen de un Golden Retriever en concreto con un Afgano porque el umbral está más bajo, lo que implica que la precisión es menor y la sensibilidad es mayor (que es lo que implica el trade-off entre Precision-Recall).

Figura 40 Golden Retriever detectado como afgano de manera errónea con umbral de 0.3

En este caso la precisión se sitúa en el 14%, algo que tiene que ver con el número de mayores detecciones que se han realizado puesto que el umbral es menor. Esto lo podemos ver en la Figura 41.

```
AP: 26.95% (Afghan_hound)
AP: 14.43% (beagle)
AP: 0.78% (golden_retriever)
mAP: 14.05%
```

Figura 41 Precisión con umbral de confianza de 0.3

3.5.2 Caso de Umbral de confianza a 0.2

Buscamos seguir investigando en el funcionamiento del sistema por lo que tomamos ahora un umbral de confianza de 0.2. Esto implica que el sistema es aún más sensible con lo que podemos obtener un mayor número de detecciones. Sin embargo, puede que haya detecciones erróneas, lo cual empeoraría los resultados aún más que en el caso anterior.

Vemos que en el caso del afgano mejora los resultados pues hay más detecciones aunque con una puntuación de confianza más baja. La mAP, vemos que es del 62.85% lo que es una cifra muy seria en comparación con las anteriores. Además, si nos fijamos en la Figura 42, que muestra la curva Precision-Recall, vemos el compromiso claro entre las dos características. Sin embargo, no se muestra demasiado bueno ya que no se acerca al ideal, recordemos que es la esquina superior derecha. Esto también viene determinado porque antes no había tantas detecciones, mientras que en este caso sí que hay más detecciones porque el umbral es más bajo, la sensibilidad por tanto es mayor.

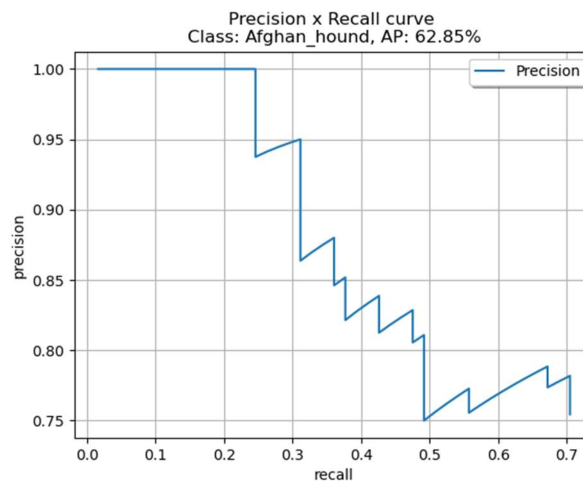

Figura 42: Curva Precision-Recall y Precision para el caso del Afgano con umbral a 0.2

En el caso del Beagle, como muestra la Figura 43, la curva se aleja del ideal, y además baja aún más la precisión para este tipo de clase.

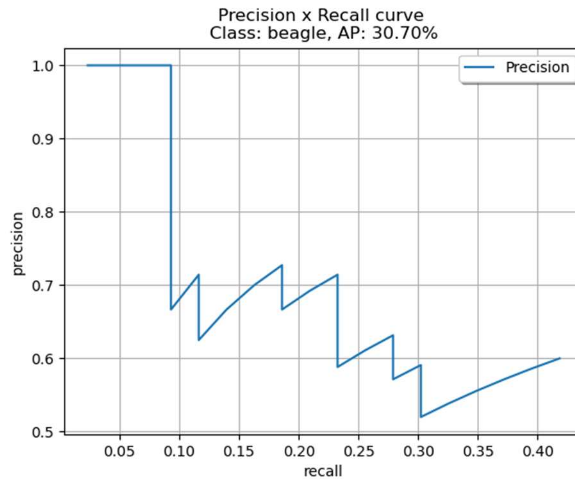

Figura 43: Curva Precision-Recall y Precision para el caso del Beagle con umbral a 0.2

Finalmente, para el caso del Golden Retriever, sigue siendo un resultado muy malo, puesto que la precisión media se coloca en un 1,57%. Esto se debe nuevamente a que como vemos en la Figura 44 hace al sistema confundir en muchas ocasiones al Golden Retriever con un afgano debido a que el umbral de confianza es menor. También, notamos como hay muy pocas detecciones de Golden Retriever, lo que *incide* en el hecho de que harían falta más imágenes para entrenar dicha raza y que el sistema fuese capaz de realizar una buena detección. Esto influye negativamente en la curva de Precision-Recall, pues influye en que, al haber pocas detecciones y falsos positivos, no haya muestras suficientes como para obtener la curva con garantías.

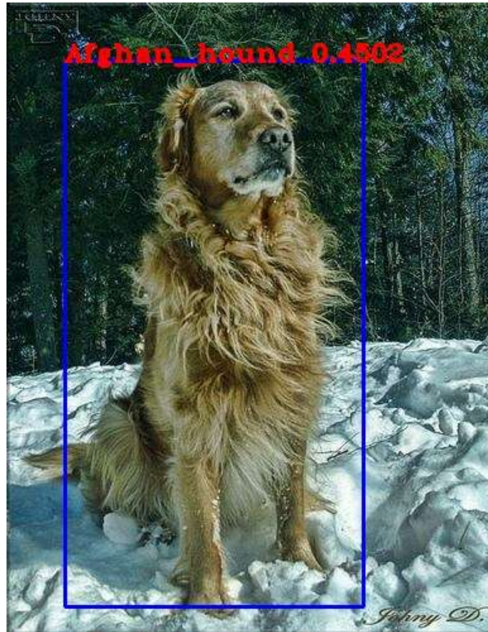

Figura 44: Caso de falso positivo originado por el sistema, en el que etiqueta un Golden Retriever como afgano

En el caso de la precisión media, vemos como al hacer la media de las tres clases, el detector ha arrojado una precisión del 31,71% (Figura 45), la más alta hasta el momento, aunque alejada de valores importantes como deberían ser a partir del 70%.

```
AP: 62.85% (Afghan_hound)
AP: 30.70% (beagle)
AP: 1.56% (golden_retriever)
mAP: 31.71%
```

Figura 45: Precisión media mAP del caso con umbral de confianza en 0.2

3.5.3 Caso de Umbral de confianza a 0.2 y métrica de entrenamiento “precision”

En este caso vamos a intentar mejorar el funcionamiento del sistema, toda vez que hemos visto que en el umbral de 0.2, mejora el número de detecciones. Debemos de intentar mejorar que las detecciones sean de mejor calidad, que intente mejorar en el caso de los Golden Retriever y además que no obtenga falsos positivos.

Para ello intentamos que, en el entrenamiento, la validación tome como métrica para saber cuál es el punto óptimo del mismo (cuando funciona el mecanismo de *EarlyStopping*), la precisión en lugar de las perdidas. Es decir, hasta ahora en el entrenamiento hacíamos que el mecanismo de *EarlyStopping* no entrase en funcionamiento hasta que, durante 3 épocas consecutivas, las pérdidas de validación no mejorasen a la más baja conseguida. En ese punto, el entrenamiento se detenía, pues la red había acabado de entrenarse convenientemente y no había posibilidad de mejorar esas pérdidas. Ahora lo que buscamos es que en la validación, se centre en la precisión, es decir, intentar que mejore la tasa de precisión lo máximo posible en las predicciones [35] y si no es capaz de mejorarla que se detenga el entrenamiento.

Realizando esta modificación, notamos unos resultados peores que en el caso anterior tanto en términos de precisión media como de forma de las curvas de Precision-Recall. Por ello, no podemos seguir mejorando los resultados por este camino. Sobre todo, en la clase Afgano es donde la caída de rendimiento es más acusada, mientras que en el caso del Beagle pierde hasta casi un 10% de precisión, lo cual no parece buen indicador de mejora de las prestaciones.

Otras modificaciones que se realizaron fueron reducir el *learning rate* a 1e-4 (en lugar del valor por defecto de 1e-3), y aumentar el *BatchSize*, pero sin obtener mejoras en las prestaciones.

3.5.4 Caso de Umbral de confianza a 0.2 y función de optimización SGD (Stochastic Gradient Descendent)

El sistema utiliza, por defecto, el optimizador Adam, pero podemos probar a realizar un cambio en la función de optimización para encontrar mejoras, en este caso, acudiendo a las funciones de optimización que tenemos disponibles en la página de Tensorflow [34]. Con ello, realizamos el entrenamiento, y recopilamos los resultados. Los resultados mejoran ligeramente respecto a los casos anteriores, puesto que las detecciones en el caso de los afganos, bajan ligeramente en cuanto al número, en el caso de los Beagle detecta un pequeño número, y en este caso sí que mejora el caso de los Golden Retriever.

Si bien no es una mejora sustancial, vemos que ya el sistema es capaz de detectar bien algunos casos de Golden Retriever, salvando los casos en los que se confunde por un Afgano como la imagen que hemos comentado anteriormente y que ese falso positivo se debe más a la influencia del umbral de confianza.

En el caso de las gráficas de las curvas de Precision-Recall, vemos que en el caso Afgano, como podemos observar en la Figura 46, vemos como en este caso, la curva tiene mejor aspecto, pues se nota el compromiso entre los dos parámetros, y en este caso sí que tiende ligeramente al ideal, que es la esquina superior derecha, recordemos. El mAP es del 37%, más bajo que en algún caso anterior.

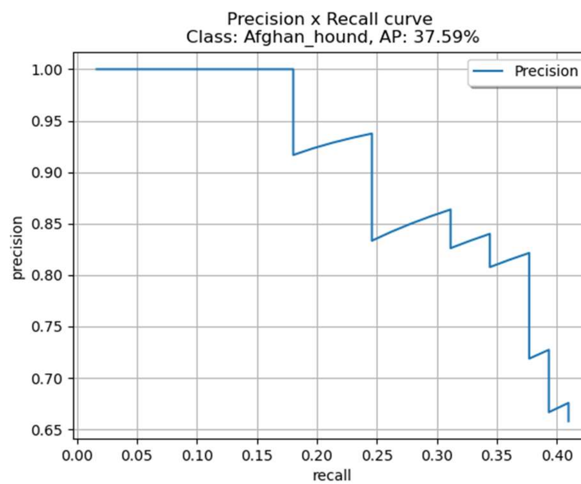

Figura 46: Curva Precision-Recall para un umbral de 0.2 y optimizador SGD (Afgano)

En el caso de la curva para el Beagle nos podemos fijar en la Figura 47, vemos como también se nota el compromiso entre las dos características, sin embargo, se aleja del ideal notablemente. Aun así, tenemos mejores resultados que en los casos anteriores. Hay un intervalo en el que según avanza el Recall, la Precisión tiende a mantenerse, a partir de 0.13. Recordemos que hay más detecciones de Beagle que en los casos anteriores y esto se manifiesta en que la mAP es ligeramente superior también (17%).

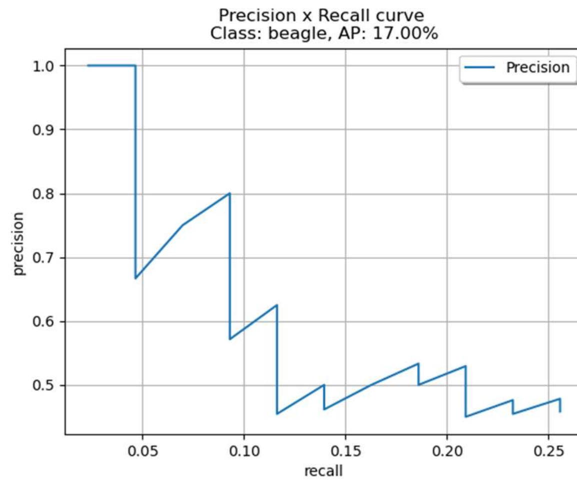

Figura 47 Curva Precision-Recall para un umbral de 0.2 y optimizador SGD (Beagle)

Donde realmente se nota la mejora, es en el caso del Golden Retriever, que es donde obtenemos el mejor resultado hasta el momento. En la Figura 48 vemos de entrada que el mAP es de un 12% cuando el valor *más* alto conseguido hasta ahora era como mucho cercano al 2%.

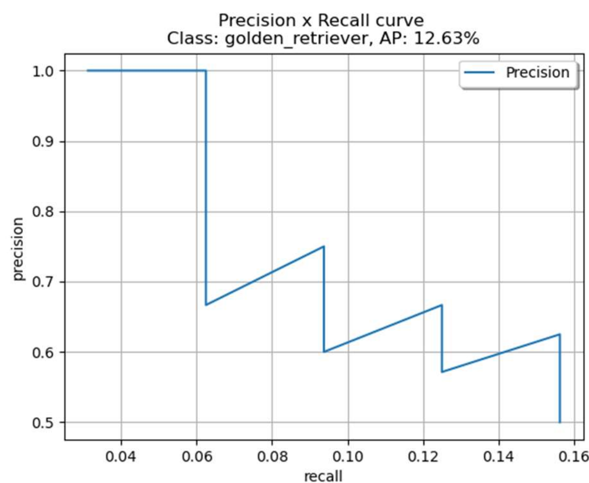

Figura 48 Curva Precision-Recall para un umbral de 0.2 y optimizador SGD (Golden Retriever)

Ahora se consiguen mejores detecciones del Golden Retriever, pues visualizando las detecciones como vemos en la Figura 49 se nota que hay un número de mayor de las mismas. Sin embargo, sigue habiendo algunos errores en las detecciones puesto que en algunas reconoce un perro de una clase distinta a la que debería ser.

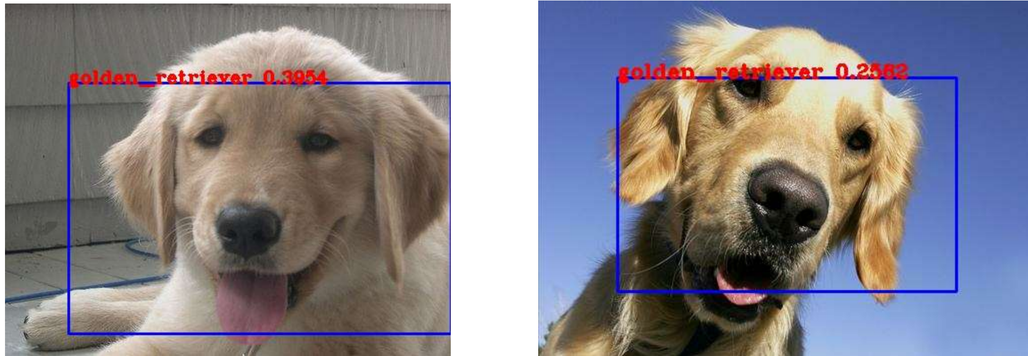

Figura 49: Detecciones de Golden Retriver, antes no detectadas

En cuanto al mAP del sistema, como vemos en la Figura 50 para este conjunto de datos y esta combinación de hiperparámetros tenemos que es un 22,4%, una cifra muy baja atendiendo a que debería ser mucho mejor, pero que en el compendio global de las modificaciones de hyperparametros que hemos realizado, ha sido la mejor.

```
AP: 37.59% (Afghan_hound)
AP: 17.00% (beagle)
AP: 12.63% (golden_retriever)
mAP: 22.41%
```

Figura 50: mAP del sistema global con SGD, y umbral de confianza a 0.2

3.6 Análisis de los resultados obtenidos con los conjuntos de datos

Lo cierto es que no existe una fórmula específica por la cual, con una combinación de parámetros determinada, se consiga obtener un buen rendimiento del sistema. A priori, se puede obtener una referencia de cómo funciona un determinado parámetro si lo modificamos. Sabemos que es lo que implica la modificación de un parámetro sobre el papel, pero una vez lo trasladamos a la práctica y a un conjunto de datos específico los resultados pueden cambiar notablemente.

En el caso del conjunto de datos de los cuadros, la mejor combinación conseguida es únicamente bajando el umbral de confianza de 0.5 a 0.2. Como sabemos los resultados no son nada satisfactorios puesto que implica que las detecciones tienen una puntuación de confianza muy baja cuando lo normal es esperar que haya una puntuación de confianza de al menos un 70% u 80% para considerar que sea una detección satisfactoria. En el mejor

de los casos hemos conseguido un 50% de puntuación de confianza tanto en la detección de los cuadros como en la detección de razas de perros, lo cual es bastante bajo teniendo en cuenta lo que se considera una detección satisfactoria.

Como hemos comentado, los resultados distan mucho de ser los esperados. En primer lugar, porque un umbral de confianza del detector es demasiado bajo. Esto hace que el sistema sea muy sensible y detecte muchos objetos pero que la calidad de la detección baje puesto que la precisión es muy baja. También se han intentado realizar, sin éxito, cambios en la red *Darknet*. Otra de las formas de intentar mejorarlo, podría haber sido un cambio radical de la red. Esto implicaba que debíamos empezar de cero a crear un nuevo sistema, además de realizar nuevas formas de encontrar las cajas de predicción (*Bounding Boxes*), preprocesamiento de los datos… Algo que hubiera implicado muchísimo más tiempo de trabajo, y que podría no haber dado resultado, puesto que como hemos comentado a lo largo de la discusión de los resultados, no existe un parámetro que pueda mejorar las cosas de una manera efectiva, ya que sobre el papel puede tener unas buenas propiedades pero en la práctica, puede no ser así. Un ejemplo claro es el caso del optimizador en el caso del conjunto de datos de los perros, en el que Adam es una opción sobre el papel mejor que SGD, pero que una vez probados ambos en el sistema se comporta mejor SGD, cuando a priori, debería dar peores resultados. Todo depende del conjunto que tengamos, y como adaptar el sistema para sacar el mayor rendimiento a ese conjunto.

4

Conclusiones y Líneas Futuras

Hemos podido estudiar en qué consiste el aprendizaje automático aplicado a la detección de objetos, repasando los elementos de los que se compone un sistema de este tipo, así como las arquitecturas existentes y nos hemos centrado en la arquitectura YOLO, que hemos podido estudar con mayor profundidad, repasando la evolución de la misma y cómo se puede aplicar en la práctica.

También hemos comprendido las diferencias existentes en realizar computación a nivel local, que exige una instalación de librerias y paquetes específica y adecuada al hardware del que disponemos, mientras que una programación en la nube proporciona un nivel de abstracción a posibles incompatibilidades, puesto que es tranparente al usuario la instalación de los paquetes. Si bien es cierto, que la programación a nivel local permite la comprensión y el enfrentamiento a la resolución de problemas de programación, con lo que ayuda en el desarrollo de distintas habilidades de programación.

Además, hemos visto que a nivel teórico los hiperparámetros no tienen por qué comportarse siempre de la misma forma a nivel práctico, sino que depende de muchos factores, como el conjunto de datos, la arquitectura de red neuronal… Hemos conseguido una visión completa de cómo funciona un sistema de detección de objetos y las partes de las que se componen, así como la secuencia de funcionamiento del mismo y cómo se aplica de manera práctica. Con ello hemos aprendido a encontrar una forma de analizar los resultados obtenidos, teniendo en cuenta la teoría estudiada.

En cuanto a las líneas futuras, sería interesante comprender y profundizar en las causas que hacen en que el sistema no se comporte como cabría esperar. Posiblemente sería interesante realizar un estudio práctico más exhaustivo y examinar el funcionamiento con

más conjuntos de datos los resultados obtenidos para obtener una visión más completa de cómo se comporta el sistema con ellos.

5

Bibliografía

[1] *Y.S. Abu-Mostafa, M. Magdon-Ismail y H.T. Lin. "Learning from Data: A Short Course",* Amlbook.com, 2012.

[2] *R.S. Sutton y A.G. Barto.* "*Reinforcement Learning: An Introduction*". MIT Press, 2018.

[3] *Aurélien Gerón.* Hands-On Machine Learning with Scikit-Learn, Keras, and Tensorflow. O´Reilly

[4] *Aurélien Gerón. "Chapter 10: Introduction to Artificial Neural Networks with keras."* Hands-On Machine Learning with Scikit-Learn, Keras, and Tensorflow. O´Reilly

[5] *Tarang Shah. "About Train, Validation and Test Sets in Machine Learning."* Towards data science. https://towardsdatascience.com/train-validation-and-test-sets-72cb40cba9e7. Fecha de última consulta: Octubre 2020

[6] *Aurélien Gerón. "Chapter 11: Training Deep Neural Networks."* Hands-On Machine Learning with Scikit-Learn, Keras, and Tensorflow. O´Reilly

[7] *Aurélien Gerón. "Chapter 3: Performance measures."* Hands-On Machine Learning with Scikit-Learn, Keras, and Tensorflow. O´Reilly

[8] *Aniruddha Bhandari. "Everything you Should know about confusión matrix for machine learning."* Analytics Vidhya. https://www.analyticsvidhya.com/blog/2020/04/confusion-matrix-machine-learning/ Fecha de última consulta: Octubre 2020]

[9] *R.Padilla, S.L.Netto y E.A.B.da Silva "A Surveyon Performance MetricsforObject-Detection"* International ConferenceonSystems, Signals and Image Processing (IWSSIP) Algorithms, https://github.com/rafaelpadilla/Object-Detection-Metrics. Fecha de ultima consulta: Octubre de 2020]

[10] *Jaime Ramírez "Curvas PR y ROC". Medium*. https://medium.com/bluekiri/curvas-pr-y-roc-1489fbd9a527. Fecha de última consulta: Septiembre 2020

[11] *Aurélien Gerón. "Chapter 14: Deep computer visión using convolutional networks."* Hands-On Machine Learning with Scikit-Learn, Keras, and Tensorflow. O´Reilly

[12] *Joseph Redmon, Santosh Divvala, Ross Girshick, Ali Harhali. "You Only Look Once: Unified, Real-Time Object Detection."*, 2016. https://arxiv.org/abs/1506.02640

[13] *Aprende Machine Learning. "Modelos de detección de objetos. Aprende Machine learning".* https://www.aprendemachinelearning.com/modelos-de-detección-de-objetos/. Fecha de última consulta: Octubre 2020

[14] *Jonathan Hui. "SSD object detection: Single Shot Multibox Detector for real-time processing."* https://jonathan-hui.medium.com/ssd-object-detection-single-shot-multibox-detector-for-real-time-processing-9bd8deac0e06. Fecha de última consulta: Octubre 2020]

[15] *Jonathan Hui. "Understanding Feature Pyremid Networks for object detection (FPN)."* https://jonathan-hui.medium.com/understanding-feature-pyramid-networks-for-object-detection-fpn-45b227b9106c. Fecha de última consulta: Octubre 2020

[16] *Mauricio Menegaz. "Understanding YOLO."* Hackernoon. https://hackernoon.com/understanding-YOLO-f5a74bbc7967 . Fecha de última consulta: Octubre 2020

[17] *Jonathan Hui. "Real-time Object Detection with YOLO, YOLOv2, and now YOLOv3."* https://jonathan-hui.medium.com/real-time-object-detection-with-YOLO-YOLOv2-28b1b93e2088#:~:text=Network%20design&text=Source-,YOLO%20has%2024%20convolutional%20layers%20followed%20by%202%20fully%20connected,7%2C%207%2C%201024. Fecha de última consulta: Octubre 2020

[18] *Diego Calvo. "Función de activación – Redes neuronales."* https://www.diegocalvo.es/funcion-de-activacion-redes-neuronales/. Fecha de última consulta: Septiembre 2020.

[19] *Amko Kamal. "YOLO, YOLOv2 and YOLOv3: All You Want to Know."* https://medium.com/@amrokamal_47691/YOLO-YOLOv2-and-YOLOv3-all-you-want-to-know-7e3e92dc4899#:~:text=Fast%20YOLO%20is%20a%20fast,but%20slower%20than%20real%2Dtime. Fecha de última consulta: Octubre 2020.

[20] *J. Redmon y A. Farhadi, "YOLO9000: Better, Faster, Stronger",* 2016, https://arxiv.org/abs/1612.08242v1

[21] *Joseph Redmon, Ali Rarhadi. "YOLOv3: An incremental improvement."* 2018, https://arxiv.org/abs/1804.02767

[22] *Neelam Tyagi. "Introduction to YOLOv4."* AnalyticsSteps. https://www.analyticssteps.com/blogs/introduction-YOLOv4. Fecha de última consulta: Octubre 2020

[23] *Alexey Bochkovskiy, Chien-Yao Wang, Hong-Yuan Mark Liao "YOLOv4: Optimal Speed and Accuracy of Object Detection"* 2020, https://arxiv.org/abs/2004.10934

[24] TensorflowObjectdetection API. https://github.com/Tensorflow/models/tree/master/research/object_detection. Fecha de última consulta: Septiembre 2020

[25] *Anton Muehlemann. "TrainYourOwnYOLO: Building a Custom Object Detector from Scratch."* 2019. https://github.com/AntonMu/TrainYourOwnYOLO. Fecha de última consulta: Noviembre 2020

[26] *Glenn Jocher. "YOLOv3 Object detection."* Ultralitycs https://github.com/ultralytics/YOLOv3. Fecha de última consulta: Julio 2020

[27] *Zihao Zhang. "YOLOV3 Implemented in Tensorflow 2.0*". https://github.com/zzh8829/YOLOv3-tf2#readme. Fecha de última consulta: Julio de 2020

[28] *Joseph Chet Redmon, "The Darknet neural net"*. https://pjreddie.com/ Fecha de última consulta: Septiembre de 2020

[29] Annotated-Images 0.1.4. MIT License https://pypi.org/project/annotated-images/. Fecha de última consulta: Septiembre de 2020

[30] *Pixabay. "Imágenes gratis para descargar sin derechos de autor."* https://pixabay.com/es/. Fecha de última consulta, Julio 2020.

[31] *Gilbert Tanner. "Tensorflow Object Detection with Tensorflow 2: Creating a custom model."* https://www.GilbertTanner.com. Fecha de última consulta: Abril 2020.

[32] *Darrenl Tzutalin. "LabelImg."* https://github.com/tzutalin/labelImg. Fecha de última consulta: Julio 2020

[33] *"Kaggle: Your Machine Learning and Data Science Community."* https://www.kaggle.com/. Fecha de última consulta: Septiembre 2020.

[34] *Aditya Khosla, Nityananda Jayadevaprakash, Bangpeng Yao and Li Fei-Fei. Novel dataset for Fine-Grained Image Categorization. First Workshop on Fine-Grained Visual Categorization (FGVC), IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2011.

[35] *Derrick Mwiti. "Métricas de keras, todo lo que necesita saber",* Neptune https://neptune.ai/blog/keras-metrics, Fecha de última consulta: Septiembre de 2020

[36] *Módule tf.keras.optimizers* www.Tensorflow.org. https://www.Tensorflow.org/api_docs/python/tf/keras/optimizers