Trabajo Fin de Grado
Ingeniería de Telecomunicación

Detección de Objetos con TinyYOLOv3 sobre Raspberry Pi 3

Autor: Ángel Moreno Prieto
Tutor: Antonio Jesús Sierra Collado
Cotutor: Álvaro Martín Rodríguez

Dpto. de Ingeniería Telemática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2020

Trabajo Fin de Grado
Ingeniería de Telecomunicación

Detección de Objetos con TinyYOLOv3 sobre Raspberry Pi 3

Autor:
Ángel Moreno Prieto

Tutor:
Antonio Jesús Sierra Collado

Cotutor:
Álvaro Martín Rodríguez

Dpto. de Ingeniería Telemática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2020

Trabajo Fin de Grado: Detección de Objetos con TinyYOLOv3 sobre Raspberry Pi 3

Autor: Ángel Moreno Prieto

Tutor: Antonio Jesús Sierra Collado

Cotutor Álvaro Martín Rodríguez

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2020

El Secretario del Tribunal

A mi familia

A mis maestros

Agradecimientos

Varias personas son artífices de que este proyecto haya salido adelante.

A mis tutores, Antonio y Álvaro, que siempre han estado ahí pese a las complicaciones de este año; y en general a los profesores de la Escuela Técnica Superior de Ingenieria, que me han enseñado a enfrentarme a cualquier problema con entereza y sin miedo, por muy difícil que fuera.

A los compañeros y grandes amigos que he hecho a lo largo de la carrera, y sin los cuáles es probable que no hubiera llegado tan lejos. En especial, a Pepe y Chema, que me han tenido que soportar estos últimos cuatro años en bastantes proyectos y aventuras.

A mi familia y amigos, por siempre confiar en mí y ponerme las pilas cuando me hacía falta un empujón.

Ángel Moreno Prieto

Sevilla, 2020

Resumen

Desde hace algún tiempo, se viene hablando de nuevas tecnologías que revolucionarán la industria y nuestra forma de vivir en los próximos años, como el Internet de las Cosas, el *Big Data*, el *Cloud Computing* o la Inteligencia Artificial. Conocerlas, trabajar con ellas y estudiar su potencial es clave para entender cómo impactarán en nuestro día a día futuro. En este proyecto haremos uso de una de estas tecnologías: la detección de objetos, una de las múltiples ramas de la inteligencia artificial, cuyo fin último será el de permitir crear máquinas con la capacidad de ver e interpretar el mundo tal y como nosotros lo hacemos a través de nuestros ojos. Además, implementaremos uno de estos sistemas en un dispositivo ligero, no originalmente diseñado para este fin, pero de propósito general, extendido, y económico; con la idea de medir su rendimiento y llegar a una conclusión respecto a la combinación.

En concreto, utilizaremos una versión simplificada de YOLOv3, uno de los algoritmos de detección de objetos más punteros en los últimos años, conocida como TinyYOLOv3; que será implementado en una Raspberry Pi 3. Los objetivos del proyecto serán los de comprobar que la instalación y puesta en marcha del algoritmo es factible, y a continuación probar y medir su rendimiento en aquellas facetas que son más relevantes en la inteligencia artificial: el entrenamiento y la inferencia.

A lo largo de esta memoria, realizaremos un viaje en el que explicaremos los principios que gobiernan el campo de la detección de objetos, desde las bases de la inteligencia artificial hasta el *deep learning*; para luego explicar en detalle el proceso de integración del algoritmo TinyYOLOv3 en Raspberry Pi 3; y finalmente procederemos a llevar a cabo las pruebas pertinentes que nos permitirán cumplir los objetivos del proyecto.

Abstract

For some time now, there has been talk about some new technologies that will revolutionize the industry and our way of life in the coming years, such as the Internet of Things, Big Data, Cloud Computing or Artificial Intelligence. Knowing them, working with them and studying their potential is key to understanding how they will impact our future day-to-day lives. In this project, we will be making use of one of those technologies: object detection, one of the many branches of artificial intelligence, which ultimate goal will be to create machines with the ability to see and interpret the world as we do through our eyes. In addition, we will implement one of these systems in a lightweight device, not originally designed for this purpose, but general-purpose, extended, and economic; with the idea of measuring its performance and reaching a conclusion regarding the combination.

Specifically, we will use a simplified version of YOLOv3, one of the most advanced object detection algorithms in recent years, known as TinyYOLOv3; which will be implemented in a Raspberry Pi 3. The objectives of the project will be to verify that the installation and implementation of the algorithm is feasible, and then test and measure its performance in those facets that are most relevant in artificial intelligence: training and inference.

Throughout this report, we will make a journey in which we will explain the principles that govern the field of object detection, from the basics of artificial intelligence to deep learning, and then explain in detail the process of integrating the algorithm TinyYOLOv3 in Raspberry Pi 3, and finally proceed to carry out the relevant tests that will allow us to fulfill the objectives of the project.

Índice

ÍNDICE DE TABLAS

ÍNDICE DE FIGURAS

1 INTRODUCCIÓN

The future belongs to those who believe in the beauty of their dreams.

- Eleanor Roosevelt -

Este capítulo sirve como breve introducción al proyecto de *Detección de Objetos con TinyYOLOv3 en Raspberry Pi 3*. Se justifica su elección e interés, y se describen resumidamente los objetivos y la finalidad de éste, centrándose primero en los conceptos teóricos del proyecto, y finalizando con las elecciones concretas que le dan nombre.

A continuación, el segundo capítulo se encargará de desarrollar el fundamento teórico detrás de la principal tecnología del proyecto: la visión artificial basada en inteligencia artificial y redes neuronales. Al ser todos estos conceptos relativamente modernos, se tratará de explicar cada uno de la forma más precisa posible, pero a la vez siendo concisos y enfocándonos en la tecnología final utilizada y en el campo en el que se aplica.

Finalmente, los capítulos posteriores tratarán sobre la implementación del modelo elegido, y los resultados obtenidos de los diferentes experimentos, así como una conclusión y breve esquema de los pasos que se podrían seguir en futuras investigaciones.

1.1 Introducción

Desde los orígenes del ser humano, éste se ha caracterizado por querer mejorarse a sí mismo y superar sus propias limitaciones. La invención de herramientas y útiles con las que facilitar tareas y actividades es un claro ejemplo; y llegar a automatizarlas hasta incluso superar nuestras propias capacidades es uno de los aspectos que nos hace únicos como especie. A lo largo del siglo XX, esta faceta se vio revolucionada por el descubrimiento e invención de la computación y de las computadoras, colocándonos en una auténtica carrera de fondo tecnológica en la que cada día parecía que se inventaba algo nuevo. El siglo XXI no es sino la continuación de ese impulso. Ahora, con capacidades de procesamiento computacional impensables hace años, tecnologías que están llamadas a revolucionar el presente y el futuro como lo hiciera la informática el siglo pasado, vienen a marcar las pautas de este joven siglo. El Internet de las Cosas, el *Big Data*, el *Cloud Computing* o la Inteligencia Artificial, son sólo algunas de los nombres que invadirán el mundo en los próximos años [1]. Y en este proyecto venimos a presentar y desarrollar una de ellas: la *visión artificial*, y, en concreto, la *detección de objetos*.

La *visión artificial* se puede definir como la capacidad de una máquina de *ver* e *interpretar* el mundo que le rodea de la misma forma que el ser humano lo hace a través de sus ojos [2]. A lo largo de su particular historia se han llevado a cabo diferentes enfoques para lograr tal objetivo, que van desde el procesamiento digital de imágenes hasta el reconocimiento de patrones; sin embargo, la visión artificial que nosotros trataremos –y que

es el camino a seguir hoy en día-, es aquella que hace uso de la inteligencia artificial y del aprendizaje automático. En pocas palabras, podríamos decir que nosotros no queremos programar una máquina que vea, por ejemplo, comparando imágenes unas con otras y dándonos un porcentaje de parecido; sino que queremos *enseñarle a ver*, que, tras aprender a reconocer objetos en unas pocas imágenes, sea capaz de reconocer los mismos en cualquier otra.

La utilidad de la visión artificial no deja lugar a dudas: desde la vigilancia inteligente [3], hasta la conducción autónoma [4]; pasando por la medicina (diagnósticos inteligentes, detección de enfermedades [5]…) o la investigación científica (por ejemplo, en relación a la pandemia de COVID-19 de 2020 [6]). La visión juega un papel fundamental en la vida y obra del ser humano, y una máquina con la capacidad de ver y entender el mundo como nosotros supondría una auténtica revolución tecnológica.

Conseguir este objetivo no es sencillo, y para lograrlo la tecnología aún tiene que avanzar bastante. En este proyecto, nos centraremos en uno de los múltiples componentes que forman la familia de tecnologías de visión artificial: la *detección de objetos*, que no es sino la capacidad de una máquina de detectar y reconocer objetos de cierta clase (personas, vehículos, señales de tráfico…) en un vídeo o imagen digital, con la mayor precisión y seguridad posible [7]. Dentro de esta rama, utilizaremos la familia *YOLO* (de sus siglas en inglés *"You Only Look Once"*; "*Sólo Miras Una Vez"*), que es un conjunto de sistemas de detección de objetos de última generación. Más adelante entraremos en detalle sobre esto.

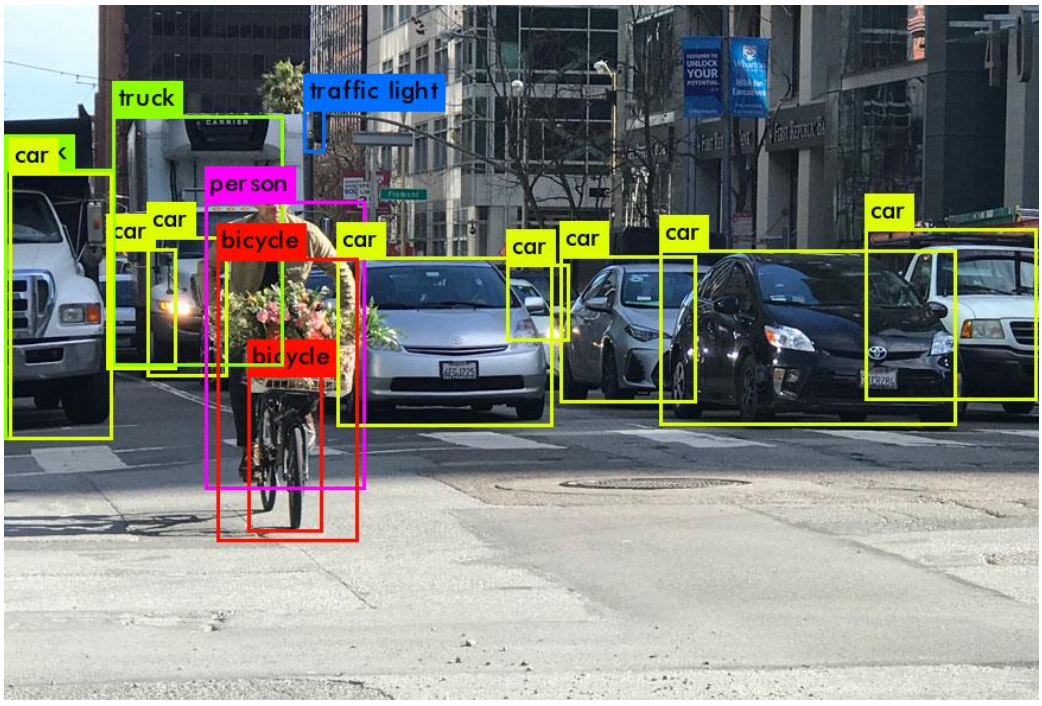


Figura 1-1. Ejemplo de detección de objetos mediante YOLOv3

Por otro lado, este proyecto no se centrará únicamente en la detección de objetos. Uno de los principales problemas al tratar con inteligencia artificial y sus derivados, como es la visión artificial, es el de la capacidad de computación. Estos sistemas requieren la realización de *muchas* operaciones matemáticas, tantas, que para aplicaciones de alto nivel muchas veces no basta con ordenadores convencionales. Incluso, se han desarrollado dispositivos específicos para trabajar con éstos, como pueden ser las TPUs de Google [8], o los *Tensor Core* de NVIDIA [9]; circuitos integrados diseñados específicamente para manejarse con las operaciones básicas que gobiernan la inteligencia artificial. Y pese a que estas nuevas tecnologías mejoran y hacen avanzar la ciencia, también las alejan del usuario medio. Es por ello que en este proyecto no sólo pondremos en práctica un sistema de visión artificial, sino que lo haremos en un dispositivo computador de propósito general, fácil de usar, y asequible; lo que se conoce como una *placa computadora* (en inglés, *single-board computer*, a veces simplificador como *SBC*). En el capítulo 3 entraremos más en detalles respecto a estos dispositivos.

1.2 Objetivos

Las ventajas de trabajar con placas computadoras son múltiples, sobre todo por su sencillez, variedad de usos y disponibilidad. Sin embargo, no son dispositivos especialmente potentes, principalmente porque las aplicaciones para las que suelen usarse no requieren una capacidad cómputo excesiva, además de que suelen tratar de mantener un precio económico. El objetivo de este proyecto será, por tanto, probar cómo un sistema puntero de visión artificial funciona sobre uno de estos dispositivos. En concreto, se utilizará una versión ligera de YOLOv3, *TinyYOLOv3* sobre una *Raspberry Pi 3*. Se entrará en detalle sobre cada uno de ellos más adelante.

Se tratará de comprobar el rendimiento de dicho programa en el dispositivo, en aquellas funciones principales que YOLOv3 nos permite realizar (inferencia y aprendizaje, en detalle en futuros capítulos), y que son la base de la visión artificial. Posteriormente daremos una conclusión basada en los resultados obtenidos, así como un esbozo de las líneas de investigación futuras que podrían llevarse a cabo.

En resumen, se definen los siguientes objetivos:

- Comprobar la correcta puesta en funcionamiento de TinyYOLOv3 en una Raspberry Pi 3.
- Comprobar el funcionamiento en inferencia y medir sus prestaciones.
- Comprobar el funcionamiento en aprendizaje y medir sus prestaciones.
- Concluir y esbozar posibles líneas de investigación futura.

2 FUNDAMENTO TEÓRICO

Can machines think?

- Alan Turing -

En este capítulo se entrará en detalles en el aspecto teórico de las tecnologías que conforman este proyecto. Al ser la detección de objetos una rama de la inteligencia artificial, no se puede entender una sin la otra, por lo que se tratará de explicar cada concepto desde su origen, comenzando por una breve introducción de en qué consisten la inteligencia artificial, el *machine learning* y las redes neuronales; para luego centrarnos en la visión artificial y cómo se usan estas tecnologías para lograr el reconocimiento de imágenes y la detección de objetos. Finalmente, explicaremos detalladamente YOLOv3 en su aspecto más teórico; y reservaremos una breve sección para hablar de las placas computadoras y de la decisión de usar Raspberry Pi. En resumen, al finalizar este capítulo se tendrán conocimientos sobre:

- Conceptos básicos de inteligencia artificial y *machine learning*.
- Redes neuronales, *deep learning* y su aplicación en la visión artificial.
- El modelo de YOLOv3.
- Placas computadoras y Raspberry Pi 3.

2.1 Inteligencia Artificial y *Machine Learning*

Cuando hablamos de inteligencia artificial es fácil que se nos vengan a la cabeza imágenes de robots con forma humana, expresiones humanas y comportamiento humano, capaces de hacer esas cosas que, hasta el momento, creíamos exclusivas de nosotros. Y puede que en parte, ése sea el objetivo. Pero para lo que respecta a este proyecto, entenderemos que la inteligencia artificial es el *estudio y diseño de dispositivos capaces de percibir su entorno y tomar acciones que maximicen la probabilidad de éxito en diferentes objetivos y tareas* [10]. Uno podría pensar que ya existen máquinas capaces de interpretar su alrededor y ejecutar tareas en consecuencia con bastante éxito, sin involucrar ningún tipo de inteligencia artificial, como podría ser una cámara de seguridad con detección de movimiento; pero los dispositivos a los que nos referimos en esta definición no son cualesquiera: se denominan *agentes inteligentes*, y sus características principales es que deben ser lo suficientemente flexibles como para adaptarse a cambios en su entorno, aprender de la experiencia, y tomar la decisión óptima bajo una percepción limitada y unas capacidades de computación finitas [10]; todo ello sin la intervención del ser humano

en el proceso.

Existen muchos ejemplos de *agentes inteligentes*. El propio ser humano es uno de ellos. Un robot que se comportase como un humano sería uno de ellos. En este proyecto, el dispositivo ejecutando el sistema de detección de objetos podría ser uno de ellos: percibirá objetos en imágenes y vídeos, los identificará, y nos los mostrará. Diseñar uno de estos agentes inteligentes no es tarea sencilla; y no es tan trivial como programar una máquina para realizar una tarea concreta. Por ello, la inteligencia artificial trata de comprender cómo otros agentes inteligentes que ya existen (como nosotros mismos, los humanos) funcionan, con el objetivo de replicar sus mecanismos mediante programación y algoritmos y obtener un resultado lo más parecido posible [11]. Y es debido a ello que, en muchas ocasiones, la inteligencia artificial suele definirse como *la capacidad de una máquina de realizar tareas que, generalmente, requieren de inteligencia humana* [12].

En lo que respecta a la realización de actividades y tareas -y, si no nos ponemos filosóficos-, podemos decir que el ser humano no ha sido programado por nadie. En su lugar, poseemos un *cerebro*, nuestro verdadero agente inteligente, un dispositivo que, mediante diversos mecanismos, es capaz de, partiendo de cero, acabar realizando prácticamente cualquier actividad o tarea que se le pida. A este proceso, mediante el cual el ser humano adquiere conocimientos que antes no tenía y que ahora sí, sin que nadie internamente se los haya pre-programado, lo llamamos *aprendizaje* [13]. Buena parte de la inteligencia artificial –y la parte que nos interesa concretamente para este proyecto-, se basa en el aprendizaje, sólo que no en el nuestro, sino en el de las máquinas. A esta rama de la inteligencia artificial se la conoce *Machine Learning* (del inglés, *aprendizaje de máquinas*, comúnmente traducido al español como *aprendizaje automático*).

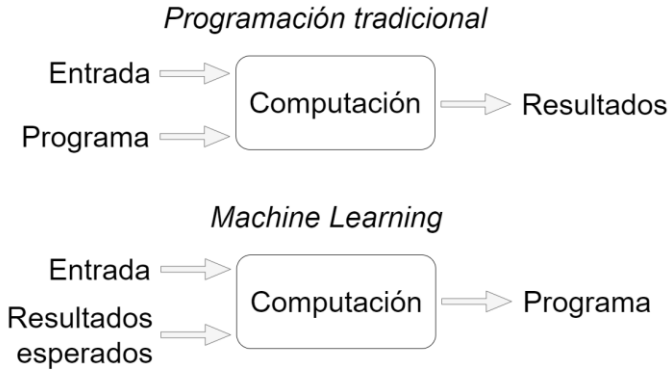


Figura 2-1. Programación tradicional contra *machine learning*

Una manera sencilla de entender lo que es el *machine learning* es comparándolo con la programación tradicional, como podemos observar en la Figura 2-1. Mientras que en ésta diseñamos conscientemente un programa que dadas unas entradas genera unos resultados -y podremos garantizar que dichos resultados son correctos en la medida en que podamos garantizar que el programa está bien diseñado-, mediante el *machine learning* no construimos nosotros el programa, sino que conseguimos que éste se "cree sólo". Para ello, prepararemos un conjunto de datos de entrada similares a los que usaríamos en el programa final, y también prepararemos la salida asociada a cada una de esas entradas (en esta explicación nos centramos exclusivamente en el concepto de *aprendizaje supervisado*, en el cuál disponemos de ejemplos de la entrada y de sus salidas; en contra del *aprendizaje no supervisado*, que carece de estas salidas preparadas y suele usarse principalmente en reconocimiento de patrones en largos conjuntos de datos [15]. A efectos de esta memoria, sólo tendremos en cuenta el aprendizaje supervisado). Alimentando a nuestra *máquina inteligente* (dispositivo que funciona mediante *machine learning*) con este conjunto de ejemplos, conseguiremos que ésta *aprenda* y se auto-configure de tal manera que será capaz de resolver cualquier otro problema del mismo estilo que el que le hemos *enseñado*.

Las palabras antes representadas en cursiva son conceptos fundamentales cuando tratamos con inteligencia artificial. La *enseñanza* y el *aprendizaje* son la base en el *machine learning*. Como ejemplo práctico, supongamos que queremos realizar un programa que sume dos números enteros naturales, independientemente de cuáles sean. Si estuviésemos programando de forma habitual, diseñaríamos un programa probablemente muy similar al siguiente:

función SUMAR (numero_A, numero_B)

devuelve (numero_A + numero_B)

Que ante unas entradas arbitrarias, digamos, 2 y 3, nos daría un resultado preciso.

SUMAR(2, 3) → 5

Mediante *machine learning*, sin embargo, enfocaríamos el problema de una manera muy distinta. En este caso, partiríamos de un conjunto de entradas y sus correspondientes salidas, como se muestra en la siguiente tabla:

Tabla 2-1. Entradas y salidas esperadas del ejemplo de suma mediante *machine learning*

Entradas		**Salida esperadas**
1	3	4
5	2	7
10	4	14
9	1	10
2	7	9
…	…	…

Ahora, si este conjunto de ejemplos es lo suficientemente amplio, podríamos utilizarlo para *enseñar* a nuestra máquina inteligente. Ésta, utilizando, una serie de algoritmos y mecanismos en los que entraremos en detalle más adelante, será capaz de *aprender* a sumar. A este proceso se le llama *entrenamiento*.

Una vez finalizado el entrenamiento de nuestra máquina inteligente, ésta estará capacitada para resolver problemas similares a los que hemos usado como ejemplos; proceso al que llamaremos *inferencia*. No sabremos exactamente cómo, no tendremos delante un código programado en un lenguaje conocido que seamos capaces de interpretar, pero sí tendremos una máquina que será capaz de sumar dos números cualesquiera en las mismas condiciones que lo hacía el programa anterior, que había sido programado precisamente para este fin. Si reutilizamos la Figura 2-1 con este nuevo ejemplo, nos quedaría algo así:

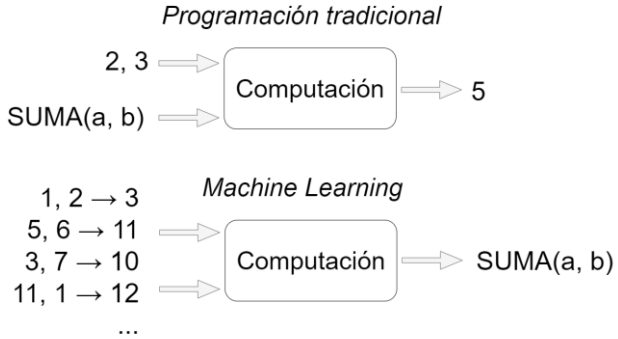


Figura 2-2. Programación tradicional contra *machine learning* y ejemplo de la suma

Es evidente que este ejemplo no es muy realista, pues qué interés habría en enseñar a una máquina a sumar cuando ya puedo programarla para que lo haga, y parece que con menor coste de tiempo y esfuerzo. Lo cierto es que para problemas sencillos, que podemos modelar matemática y lógicamente con facilidad, la programación tradicional es más práctica. Pero cuando los problemas se complican más allá de simples y arbitrarias operaciones matemáticas (véase interpretar y comunicarse mediante el lenguaje humano, jugar a juegos de mesa complejos, o reconocer objetos en imágenes), puede ser la única solución. Cuando la programación tradicional falla, porque aquello que queremos programar es altamente complejo o difícil de modelar, el *machine learning* aparece como una alternativa bastante fiable, siempre que tengamos disponible un grupo suficiente de entradas, y de sus correspondientes salidas, ejemplos con los que poder entrenar a nuestra máquina inteligente.

Así, de la misma forma que el cerebro humano, el *machine learning* no proporciona mecanismos para resolver un problema, pero sí los mecanismos para *aprender a resolver un problema*. El ejemplo anterior es muy básico y simplificado, y no representa todo lo que el *machine learning* engloba, pero sirve para entender a qué nos referimos cuando hablamos del aprendizaje automático. Y una vez comprendidos los conceptos básicos, tan sólo queda preguntarnos cómo se logra que una máquina aprenda.

2.2 Redes Neuronales

Las redes neuronales, propiamente llamadas *redes neuronales artificiales* y muchas veces simplificadas como NNs (de sus siglas en inglés, *neuronal networks*), pueden definirse como un modelo computacional parcialmente inspirado en las redes neuronales biológicas que constituyen el cerebro de los animales [14]. Es una aproximación acertada, si lo que buscábamos, al fin y al cabo, era emular el aprendizaje del cerebro humano (entre otras funciones). Así, una red neuronal artificial consta de numerosos elementos –*neuronas artificiales*- interconectados de tal manera que forman una red capaz de resolver problemas específicos [15], y lo más importante, capaz de *aprender* a resolver dichos problemas. Un esquema de su estructura puede verse en la siguiente figura:

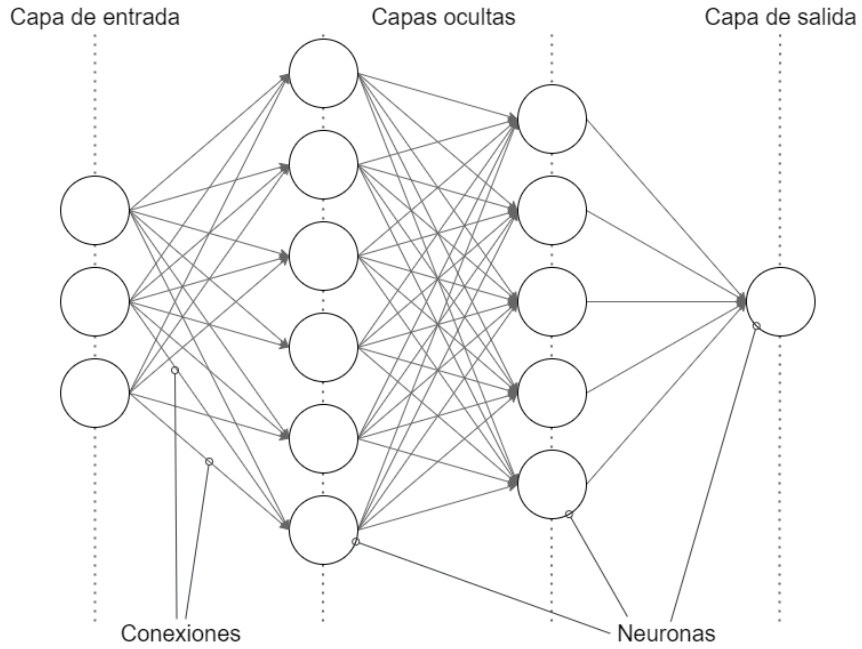


Figura 2-3. Red neuronal artificial estándar

El elemento fundamental de estas redes neuronales es, como ya se ha dicho, la *neurona*. Ésta puede modelarse, de forma muy simplificada, como en la Figura 2-4.

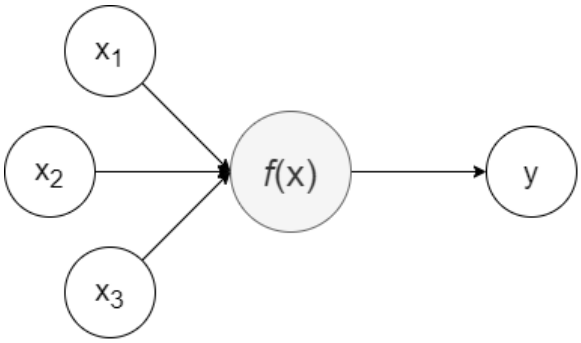


Figura 2-4. Neurona simplificada

Ante una o varias entradas, la neurona genera una salida mediante la aplicación de una o varias funciones y parámetros sobre dichas entradas. Si nos fijamos ahora otra vez en la Figura 2-3, podemos ver que esa salida, a su vez, se convierte en la entrada de varias neuronas diferentes, formándose así una red; aunque, estrictamente, podríamos tener una red neuronal de tan solo una sola neurona. La *magia* que permite a las redes neuronales aprender yace en esa función y en esos parámetros que rigen el comportamiento interno de la neurona, y entraremos en detalle sobre ella en las próximas líneas.

Volviendo a la Figura 2-3, podemos observar que las neuronas se distribuyen en capas (*layers*), y que capas inmediatamente consecutivas están conectadas entre sí todas con todas. Las capas pueden a su vez dividirse en capa de entrada (*input layer*), capa de salida (*output layer*) y capas ocultas (*hidden layers*). Las dos primeras se definen bastante bien por sí solas: son aquellas encargadas de recibir los datos de entrada, y de retornar una salida interpretable, respectivamente. Las capas ocultas, por su parte, son las que transforman esos datos de entrada mediante sus neuronas, generando poco a poco el resultado final. Puede haber varias capas ocultas, o ninguna, y pueden estar configuradas de múltiples maneras; todo dependerá de qué tipo de red neuronal estemos utilizando [16].

Finalmente, estas redes neuronales en su conjunto serán las encargadas de realizar todo el proceso que antes hemos denominado *machine learning*. Durante el entrenamiento, alimentaremos esta red con ejemplos, y las neuronas e interconexiones se irán configurando automáticamente mediante diversos mecanismos que veremos a continuación. Durante la *inferencia*, la red utilizará su configuración actual para generar un resultado dadas unas entradas arbitrarias, y, si el entrenamiento prosperó correctamente, así lo hará el resultado.

Ahora, entraremos en detalle en el funcionamiento concreto de las neuronas artificiales, en las matemáticas que rigen su comportamiento, y en cómo se ha logrado que, mediante un sistema relativamente sencillo, puedan aprender; e iremos escalando poco a poco desde ahí para finalizar comprendiendo en su totalidad los principios de las redes neuronales.

2.2.1 Neuronas artificiales

El modelo representado en la Figura 2-4 es una versión simplificada de lo que es una neurona artificial. Uno más realista podría ser el de Frank Rosenblatt (1928), conocido como *perceptrón* [17], que fue uno de los primeros modelos de neurona artificial jamás diseñados.

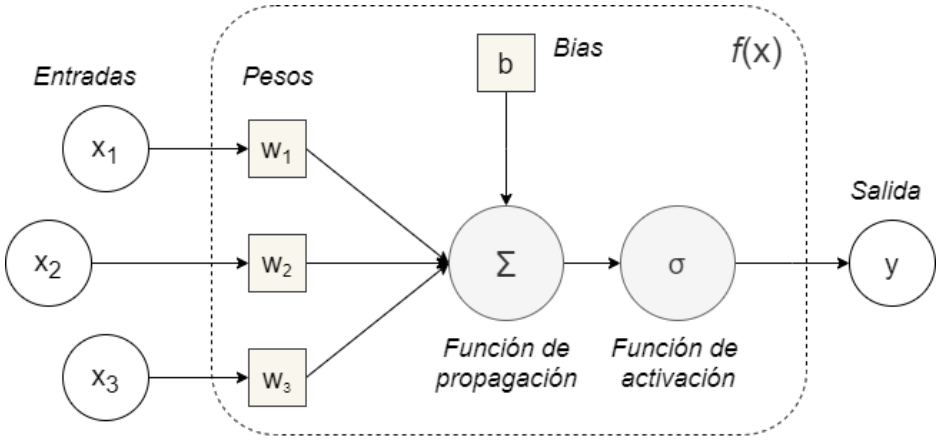


Figura 2-5. Esquema de un perceptrón

Como se puede observar, se mantienen las entradas y la salida del ejemplo de la Figura 2-4, pero además se desarrolla mucho más la función que gobierna la neurona. En la Figura 2-5, podemos distinguir varios nuevos componentes que juegan un papel esencial no sólo en este modelo, sino en general en *todos* los modelos de neuronas artificiales:

- La **función de propagación**, que determinará cómo se transforman las diferentes entradas en una salida potencial, a la que llamaremos *activación*. Normalmente dicha función será la suma ponderada de las entradas, siendo multiplicadas por sus **pesos**, que vemos en la figura representados como w_i [15]. Los pesos serán parámetros muy importantes, pues determinarán cuán influyente es una entrada para esta neurona, y será uno de los principales agentes que se irá modificando en el proceso de aprendizaje de nuestra red.

  También podemos ver que esta función recibe un componente llamado ***sesgo*** o ***bias***. Por ahora, bastará con saber que es un elemento opcional (dependerá del tipo de red que hayamos diseñado) y que se sumará a los anteriores valores. No dependerá de ninguna entrada, y en muchas ocasiones apareceré representado como el peso 0, ligado a una entrada cuyo valor será siempre 1. Su función se verá más clara cuando expliquemos el siguiente componente del perceptrón, la *función de activación.*

  En la ecuación (2-1) vemos la representación matemática de este proceso.

$$z = b + \sum_{i=1} w_i x_i \qquad (2\text{-}1)$$

  Donde z será la salida de la función de propagación (la activación), y b el sesgo. A su vez, el resultado de esta ecuación será la entrada de la función de activación.

- La **función de activación**, por su lado, se comporta como un *filtro*: dada la entrada recibida de la función de propagación, le aplica cierto tipo de función y retorna el valor que será la salida de la neurona. Se suele decir que una neurona se *ha activado* cuando devuelve un valor distinto de 0, y se dice que *no se ha activado* cuando el valor es 0. La elección de dicha función determinará en gran medida el comportamiento que tendrá nuestra red neuronal; y a lo A lo largo de la historia y en función de la aplicación se han utilizado diferentes aproximaciones, de las cuáles hemos destacado las siguientes:

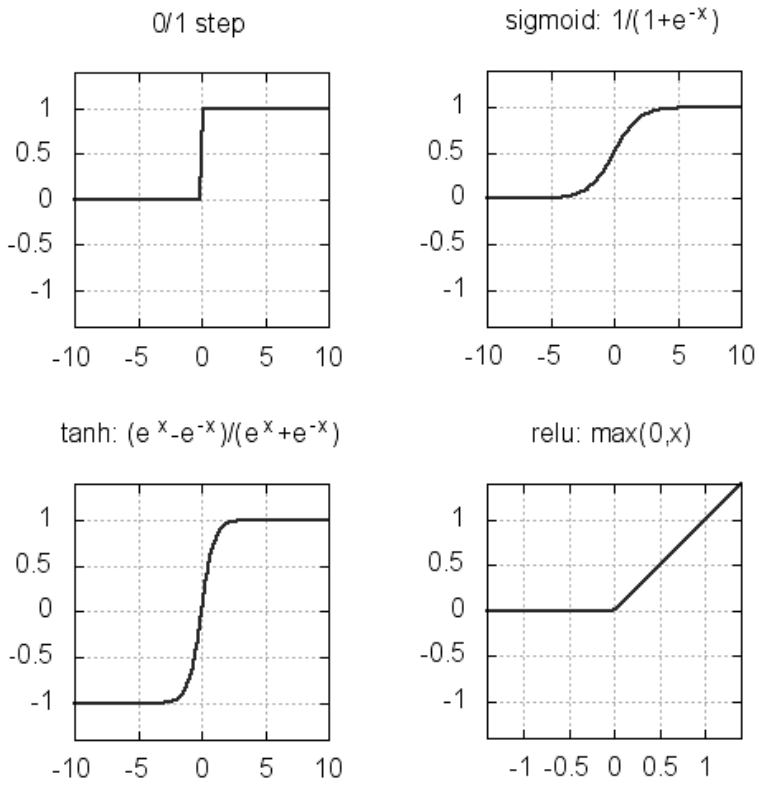


Figura 2-6. Funciones de activación más populares.

De izquierda a derecha y de arriba abajo, podemos identificar la función paso binario, la sigmoide, la tangente hiperbólica, y la *ReLU* (de sus siglas en inglés, *Rectified Linear Unit*). Cada una de ellas presenta unas características y unos inconvenientes, y estará mejor preparada para trabajar con diferentes problemas. Por ejemplo, la función de paso binario es sencilla, pero tan sólo distingue entre dos tipos de salida, 0 o 1 (neurona activada o desactivada), por lo que no se puede usar en aplicaciones que requieran resultados más diversos (no binarios); la función sigmoide permite expresar probabilidades, al tener un rango continuo entre 0 y 1; y la función tangente hiperbólica permite tener en cuenta los valores de entrada negativos, no desactivando la neurona más que en casos muy concretos donde la entrada es igual a 0. En estas funciones es, además, donde el sesgo juega un papel importante, pues permite desplazar el valor original que recibiría esta función si la función de propagación sólo fuera la suma de las entradas ponderadas, como es el caso general, permitiendo así ajustar de una manera más precisa el resultado final.

Para la visión artificial, existen dos funciones de activación muy importantes. La primera es la ReLU, cuya operación es sencilla: si la entrada es negativa, la neurona no se activa; y si es positiva, su salida será igual a la entrada. Ésta es una función de activación popular no sólo en visión artificial, ya que entre sus ventajas destaca que resulta más fácil de entrenar y que presenta un mayor rendimiento en la precisión de los resultados [18]. La otra es la *softmax*, una función que generaliza la función logística (Figura 2-7) para múltiples dimensiones. Se utiliza principalmente para clasificación, ya que dados un conjunto de posibles resultados, puede asignarles a cada uno una probabilidad, y de ahí determinar la elección definitiva.

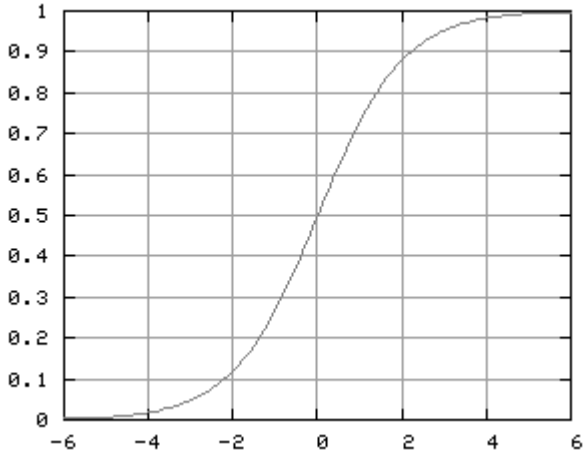


Figura 2-7. Función logística

Con esto tenemos suficientes conocimientos sobre el funcionamiento básico de una neurona artificial y sus operaciones internas. Si volvemos la vista a la Figura 2-3, podemos observar que cada neurona realizará estas acciones con cada entrada, generando así una propagación de información que desembocará en la salida final. A este proceso que se conoce como *forward propagation*, o *propagación hacia delante*. Gracias a éste, las redes neuronales pueden ser configuradas para emular cualquier función matemática real (para lo que es requisito indispensable que la función de activación sea no lineal), independientemente de su forma o complejidad; esto es, dada una función arbitraria *f(x)*, existe una red neuronal artificial cuya salida para cualquier *x* será *f(x)*. Esto se conoce como *teorema universal de aproximación* [19], y es uno de los principios que garantizar que las redes neuronales puedan, en teoría, aprender a realizar cualquier tarea. Por lo tanto, ya tan sólo nos queda ver qué acciones hay que tomar para conseguir que las neuronas, y por ende las redes neuronales, sean capaces de aprender.

2.2.2 Aprendizaje

Como ya se introdujo en párrafos anteriores, el aprendizaje de una red neuronal se lleva a cabo durante la fase de entrenamiento, y su principio básico es utilizar un conjunto lo suficientemente amplio de ejemplos del problema que queremos resolver, con el objetivo de enseñar a la red para que sea capaz de resolver cualquiera similar. A este conjunto de ejemplos se le conoce como *muestras*, o *samples*. También hemos adelantado que esto se conseguirá ajustando poco a poco y automáticamente los parámetros internos de la red, de cada neurona. El proceso que se sigue para conseguir dicho objetivo se conoce como *propagación hacia atrás*, o *backpropagation* [20], y puede describirse en los siguientes pasos:

1. Se parte de una red arbitrariamente configurada, habiendo elegido las que serán las funciones de propagación y activación; y dándoles un valor inicial (generalmente cercano a 0) a los pesos y los *sesgos*.
2. Se alimenta la red con la primera muestra (*sample*) del conjunto. Las entradas de ésta se propagarán por todas las neuronas (*forward propagation*), que llevarán a cabo sus operaciones, hasta dar con un resultado en la última capa. Ahora, la red comparará dicho resultado con el resultado esperado de la muestra mediante una *función de coste*, con la que obtendremos un *error de entrenamiento*, o *pérdidas de entrenamiento* (*training loss*) [15], que en cierto modo medirá cuánto se ha equivocado, como idealmente se representa en la ecuación (2-2), donde *y* es la salida obtenida, e *y'* la esperada. El objetivo será minimizar dicho error.

$$E = C(y, y') \tag{2-2}$$

3. El error de entrenamiento será analizado por cada neurona de cada capa de la red, en orden, y se utilizará para ajustar los pesos y el sesgo de cada una (*backpropagation*), siguiendo un algoritmo que puede variar en función de la red que hayamos diseñado. Las siguientes ecuaciones representan conceptualmente el proceso, siendo Δw_i: la variación con la que ajustaremos el peso de cada neurona.

$$\Delta w_i = G(E) = G(C(y, y')) \tag{2-3}$$

$$w_i = w_i + \Delta w_i \tag{2-4}$$

4. Se repiten los pasos a partir del 2, tratando de minimizar al máximo el error, haciéndolo 0 o muy cercano a 0. En función del tipo de datos que queramos que la red aprenda, aceptaremos un valor de error más o menos alto para determinar que la red ha finalizado su aprendizaje.

A la ejecución completa de estos pasos para una única muestra la llamaremos *bucle*, o *loop*. Cuando todas las muestras de las que se compone el set de entrenamiento han pasado por el bucle y han tenido la oportunidad de actualizar los parámetros, se dice que ha pasado una *época*, o *epoch*. El número de épocas es uno de los parámetros más utilizados para determinar cuánto debe durar el entrenamiento [21].

Estos pasos representan una explicación muy teórica de los algoritmos reales que se llevan a cabo para entrenar una red neuronal. En función de la red y el método concreto, existirán variantes adaptadas a la aplicación o a lo que se quiere enseñar, y se determinará qué función de coste utilizar y qué algoritmo usar para ajustar los pesos en cada iteración. En esta memoria explicaremos el más extendido, el *algoritmo de gradiente descendente*, que es, además, el que utiliza YOLOv3.

El algoritmo parte eligiendo como **función de coste** el cuadrado de la distancia euclidiana, cuya ecuación se representa a continuación (mantendremos la notación de los pasos anteriores):

$$E = C(y, y') = \frac{1}{2}\|y - y'\|^2 \tag{2-5}$$

Gráficamente, esta ecuación tendrá una forma parabólica que se puede aproximar de la siguiente manera:

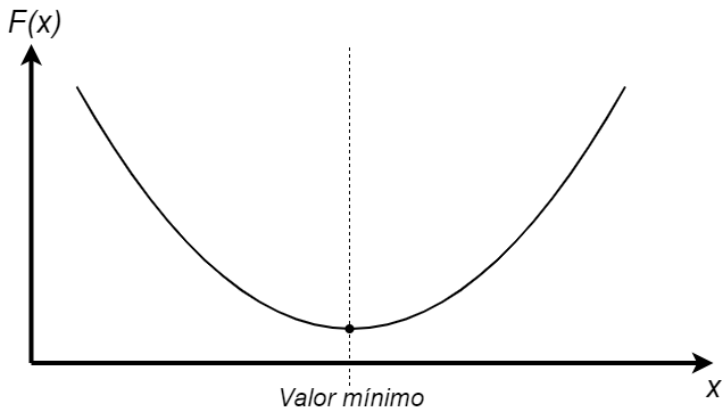


Figura 2-8. Gráfica del gradiente descendente

Donde *F(x)* será en nuestro caso el error, por lo que, como podemos observar, para lograr el objetivo (recordemos, minimizar al máximo dicho error), habría que *descender* por la curva hasta su punto mínimo. Aquí es donde entra en juego el gradiente descendente.

El gradiente descendente es un algoritmo que sirve para minimizar una función objetivo (en nuestro caso, *C(y, y')*) partiendo de un punto arbitrario de ésta mediante la actualización de sus parámetros en la dirección opuesta al *gradiente* de la función [22]. Como la salida *y* de dicha función es producida por los pesos de las neuronas, podemos utilizar el gradiente de la función de coste para actualizar el valor de los pesos. Si reutilizamos las ecuaciones (2-3) y (2-4):

$$\Delta w_i = -\nabla C(y, y') \tag{2-6}$$

$$w_i = w_i + \Delta w_i = w_i - \eta \nabla C(y, y') \tag{2-7}$$

Donde el operador nabla ∇ representa el gradiente, y η es un nuevo parámetro llamado *ratio de aprendizaje* (*learning rate*). No entraremos en detalle respecto a cómo se resuelve la operación de gradiente, porque es un desarrollo matemático que sobrepasa las intenciones de esta memoria, y bastará con saber que entra en juego la derivada parcial de la función de coste respecto al peso o a los pesos que vayamos a ajustar. Por otro lado, el *ratio de aprendizaje* se utiliza para determinar el tamaño del *paso* que se da para alcanzar el mínimo. Gráficamente, se representaría como la distancia que avanzamos en la curva de la gráfica con cada ejecución. Su valor varía entre 0 y 1, y es configurado a mano antes de comenzar el entrenamiento. Idealmente debería ser el más grande posible, porque a mayor sean los pasos que demos, más rápido llegaremos al óptimo; sin embargo, escoger un valor demasiado grande puede provocar que la función objetivo *diverja*, es decir, que "nos pasemos" del mínimo, como si saltásemos al otro lado de la curva, no obteniendo buenos resultados; y escoger un valor demasiado pequeño puede resultar en un aprendizaje demasiado lento [23]. Seleccionar un buen valor de ratio de aprendizaje es importante, y para ello se requiere un análisis previo del tipo de información que se quiere aprender y de la arquitectura de red neuronal que vamos a utilizar. En algunos modelos, además, el valor del ratio de aprendizaje irá variando a medida que se sucedan las épocas. En YOLOv3, por ejemplo, se utiliza un rango que va desde 0'01 hasta 0'0001 [24].

Dentro de los algoritmos de gradiente descendente, existen tres variantes, diferenciadas por el número de muestras necesarias para llevar a cabo la actualización de los pesos, conjunto que se conoce como *lote* o *batch*. Éstas son:

- **Gradiente descendente por lotes:** conocida en inglés como *batch gradient descent*, es el modelo estándar, y en muchas ocasiones aparece mencionado como *gradiente descendente* a secas. En éste, el tamaño del lote equivale al número de muestras que componen el set de entrenamiento, o lo que es lo mismo: se actualizan los pesos al finalizar cada época [22]. Para ello, el error de cada bucle se acumula y luego se divide entre el número de muestras, y se usa éste para realizar los cálculos que determinarán los nuevos pesos. Al proceso por el que un lote pasa por red antes de que esta se actualice se lo conoce como *iteración*.

  Su principal ventaja es la garantía de llegar a una solución óptima, en este caso, a un mínimo absoluto de la función de coste [22]; por otro lado, sus desventajas son que, al tener que esperar a que todas las imágenes del dataset alimenten la red, el proceso se puede alargar bastante, más cuanto mayor sea dicho dataset Además, al tener que almacenar los errores en cada bucle para posteriormente ajustar los parámetros, se requiere de una mayor capacidad en memoria.

- **Gradiente descendente estocástico**: conocido en inglés como *stochastic gradient descent* y comúnmente simplificado como *SGD*, es un modelo que realiza la actualización de parámetros *en cada* bucle, que sería lo mismo que decir que tiene un tamaño de lote igual a 1. Podríamos decir que, en cierto modo, es la versión opuesta del sistema por lotes. En este caso, la ventaja principal es que el proceso es considerablemente más rápido, pues los parámetros se actualizan con cada imagen, y requiere de menos memoria, porque ya no hay que almacenar un histórico de errores, lo que lo hace especialmente conveniente para dispositivos poco potentes. A su vez, su mayor desventaja es que pierde la garantía de convergencia en un mínimo absoluto de la función. Esto quiere decir que puede "pararse" y no avanzar al llegar a un mínimo local de la función, que aunque podría ser válido para cierto entrenamiento, no es óptimo. Por otro lado, el número de operaciones a realizar también se ve incrementado, debido a la rápida actualización de parámetros, lo que podría suponer un mayor coste computacional.

  Si comparásemos gráficamente ambos métodos, obtendríamos unas curvas similares a las de la siguiente figura, donde podemos ver cómo varía el error de entrenamiento a medida que más datos pasan por la red:

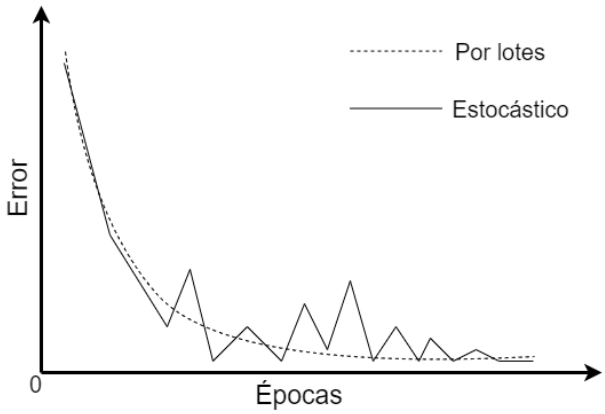


Figura 2-9. Comparativa métodos de gradiente descendente principales.

Las diferencias en el error de los sistemas estocásticos son más aleatorias porque son relativamente independientes unas de otras (cada imagen genera su propio error), mientras que en el caso del sistema por lotes llevamos a cabo una ponderación de errores entre el conjunto completo de imágenes, suavizando así la curva.

- **Gradiente descendente por mini-lotes**: es un punto intermedio entre el método por lotes y el SGD. Como se puede adivinar por su nombre, mediante este sistema usaremos *mini-lotes*: subconjuntos del grupo total de muestras, cuyo tamaño puede ser variable (aunque todos los mini-lotes deberían tener el mismo), y que se generan aleatoriamente en cada época. Es uno de los mecanismos más populares, porque mezcla lo mejor de ambas opciones: por una parte, reduce el número de actualizaciones de parámetros que se realizan, aligerando el coste computacional; pero a la vez el número de mini-lotes puede ser ajustado lo suficiente para no sobrecargar la memoria, volviéndolo aceptable en dispositivos menos potentes. Además, garantiza la convergencia en un mínimo absoluto de la función objetivo, tal y como conseguíamos con el sistema por lotas, pero en un número de épocas similar al alcanzable mediante SGD [22].

Con esta explicación, podemos dar por concluida la visión general del proceso de aprendizaje de las redes neuronales. Existen, no obstante, más mecanismos y variantes que se han ido diseñando para mejorar las prestaciones de redes neuronales genéricas y específicas, con el fin de aumentar la velocidad de éstas o su precisión media (*mAP*, *Mean Average Precision*, un parámetro muy utilizado para medir la calidad de acierto de las redes neuronales). El documento referenciado en [22], en el que se ha basado una gran parte de esta explicación, profundiza sobre estas alternativas, haciendo una descripción detallada y comparando ventajas y defectos de cada una de ellas.

Antes de finalizar con este apartado, se hará mención de errores comunes que pueden darse durante el entrenamiento; en concreto, el *overfitting* y el *underfitting*. Podemos empezar echando un vistazo a la siguiente gráfica:

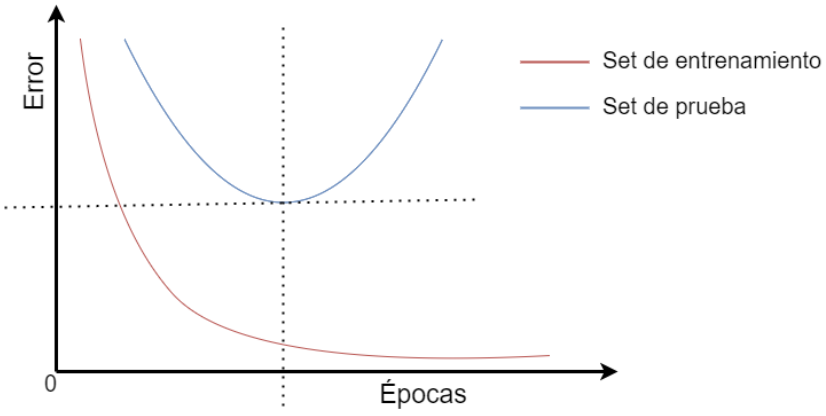


Figura 2-10. Comparativa error de entrenamiento contra error en inferencia.

En ésta podemos ver cómo varían el error de entrenamiento (que usa el set de entrenamiento) y el error de inferencia (que usa datos diferentes a los que se han usado en el entrenamiento). El error de entrenamiento (en rojo) mantiene su rumbo natural, minimizándose a medida que avanzan las épocas. Sin embargo, vemos que el error de inferencia (en azul) comienza paralelo al de entrenamiento pero acaba incrementándose radicalmente. Éste es un error que se suele dar con mucha frecuencia a la hora de entrenar redes neuronales. Si marcamos el punto donde se cruzan las líneas discontinuas como el óptimo, la situación previa se conoce como *underfitting*, y, la posterior, *overfitting* o *sobreajuste*.

La primera se da cuando ni los datos de entrenamiento ni los de inferencia arrojan un buen resultado de error. Es el caso más sencillo, pues simplemente quiere decir que la red no está lo suficientemente entrenada, y hay que dejarle más tiempo. El segundo, más complejo, es en el que el error de entrenamiento comienza a ser aceptable pero, por alguna razón, el de inferencia comienza a incrementarse otra vez. Éste, el *overfitting*, se da cuando la red se ha *acostumbrado* demasiado a los patrones de los datos del set de entrenamiento; tanto, que es incapaz de predecir resultados válidos en cualquier otro.

No existe una forma concreta de resolver estos problemas, más allá del análisis de los resultados a lo largo del entrenamiento y la modificación del set de entrenamiento o de los parámetros de la red en caso de que se detecten los fallos. Uno de los mecanismos más populares para facilitar la observación es el uso de un *set de validación*, que no es más que un conjunto de datos, distintos a los del set de entrenamiento y que no se usarán para entrenar la red, con los que se prueba ésta cada cierto tiempo mientras se está entrenando, calculando así un *error de validación* que puede compararse con el error de entrenamiento, y así determinar en qué situación nos encontramos. Se presenta la siguiente tabla como ejemplo, basándose en la que se puede encontrar en la referencia [25]:

Tabla 2-2. Ejemplo de posibles situaciones en función de los errores de entrenamiento y validación

Error de entrenamiento	***Error de validación***	***Observación***
1%	11%	Podríamos estar ante un caso de *overfitting*, porque el entrenamiento está yendo correctamente, pero no así la validación. Habría que comprobar la evolución del error de validación para determinar si éste está volviendo a crecer.
14%	32%	Ambos errores son altos, bien podríamos estar ante un caso de *underfitting*, donde necesitaríamos más entrenamiento; o bien podría ser que los datasets de entrenamiento y validación no son los más adecuados.
0,3%	1,1%	El entrenamiento y la validación van bien, por lo que este modelo parece que funcionará correctamente.

2.2.3 Parámetros e hiperparámetros

Hasta ahora, hemos tratado a todas las variables, constantes y, en general, elementos que forman parte de las redes neuronales y que son configurables (ya sea manual, o automáticamente) como parámetros. Sin embargo, en el vocabulario del *machine learning*, se realiza una división: se llaman exclusivamente *parámetros* a aquellas variables que se configuran automáticamente mediante los datos que viajan por la red; y se llaman *hiperparámetros* a aquéllos que se pueden manipular manualmente y no se obtienen mediante los datos. Una descripción más profunda se detalla en la siguiente tabla [26]:

Tabla 2-3. Comparativa entre parámetros e hiperparámetros

Parámetros	Hiperparámetros
Se necesitan para realizar las predicciones	Se utilizan para configurar la red y generar los parámetros del modelo
Sus valores determinan la habilidad del modelo	Generalmente son especificados manualmente por el diseñador de la red
Se estiman y aprenden mediante los datos de entrada durante el entrenamiento	Se pueden configurar mediante heurística [51]
Rara vez se modifican a mano	Se ajustan a un problema determinado
Se almacenan como parte del modelo, una vez éste está definitivamente entrenado	

La importancia de saber diferenciar parámetros e hiperparámetros se verá más clara cuando analicemos las redes neuronales dedicadas a la visión artificial en la siguiente sección y entremos ligeramente en el *Deep Learning*. Por ahora, vamos a listar algunos de los parámetros e hiperparámetros más relevantes, muchos de los cuales ya se han explicado previamente:

- **Parámetros:** Los pesos y *s*esgos de las neuronas; los coeficientes de las funciones de coste de algunos modelos de red neuronal; el error de entrenamiento y validación.
- **Hiperparámetros:** El número máximo de iteraciones, épocas o *batches* antes de dar por finalizado el entrenamiento; el tamaño de los *batches* o de los *mini-batches* y de los sets de entrenamiento y validación; el ratio de aprendizaje y su variación a lo largo del entrenamiento.

Con esta breve aclaración, se da por terminado la sección dedicada a redes neuronales. Cabe únicamente puntualizar que en estos apartados y párrafos, aunque se han explicado los principios básicos de las redes neuronales, comunes en su inmensa mayoría a todas éstas, se ha hecho foco en un tipo concreto de redes, las llamadas *redes neuronales hacia adelante* (*feedforward neural networks*, generalmente se usa la denominación en inglés). Éstas son aquéllas en las que las que las neuronas sólo se conectan en una dirección, sin que haya bucles (en contraposición, por ejemplo, de las *redes neuronales recurrentes*), y el flujo de datos en inferencia es exclusivamente de capa de entrada a capa de salida. La razón por la que nos hemos centrado en este modelo es que es el más utilizado en el campo que nos concierne, la visión artificial, concretamente mediante un subtipo conocido como *redes neuronales convolucionales*, generalmente simplificado como *CNN* o *ConvNets* (de sus siglas en inglés, *Convolutional Neural Networks*). Estas redes tienen la característica principal de que asumen que sus datos de entrada serán imágenes (generalmente, aunque técnicamente su función es explotar la relación espacial de cualquier conjunto de datos), por lo que están diseñadas para trabajar con éstas de diferentes maneras, buscando optimizar las operaciones que se quieran hacer sobre ellas. Esto es algo que guarda mucha relación con el *deep learning*, otro de los múltiples campos en los que se divide la inteligencia artificial, y que detallaremos en la próxima sección.

2.3 *Deep Learning* y redes neuronales aplicadas a visión artificial

Hasta ahora nos hemos centrado en el funcionamiento interno de las redes neuronales y en cómo se lleva a cabo el proceso de aprendizaje de éstas. Hemos hablado de neuronas, de *backpropagation* y de hiperparámetros; pero hemos dejado de lado uno de los elementos más importantes a la hora de trabajar con inteligencia artificial: los datos.

Hasta ahora, hemos tratado lo datos de forma generalizada, usándolos como entradas que fluyen por la red y

desembocan en una salida; o como sets o conjuntos que se usan de forma diferenciada para entrenar, validar o probar las redes. Para aplicaciones sencillas, como podría ser el ejemplo que poníamos al comienzo en el que buscábamos que una máquina aprendiese a sumar, con esto podría ser suficiente, pues, al fin y al cabo, el objetivo que buscábamos (sumar) y los datos que utilizábamos (números) eran a su vez sencillos. Sin embargo, si recordamos que uno de los fines de la inteligencia artificial es el de ser capaz de realizar trabajos que actualmente parecen exclusivos de los seres humanos, tenemos que aceptar que en la gran mayoría de ocasiones, los datos que querremos que nuestra inteligencia artificial maneje no serán simplemente números, sino toda aquella información que el propio ser humano es capaz de percibir.

La visión artificial es un claro ejemplo de esta situación. Si el objetivo es diseñar máquinas que sean capaces de *ver* como lo hace el ser humano, tendrá que ser capaz, de alguna manera, de *percibir* de igual manera o similar a como lo hace el ser humano mediante la visión. Actualmente, podemos digitalizar aquello que vemos mediante cámaras que producen vídeos e imágenes; pero una red neuronal, como hemos visto, trabaja con algoritmos y operaciones matemáticas numéricas, por lo que de alguna forma debemos convertir esas entradas complejas –que bien pueden ser imágenes, como en la visión artificial, o audios para reconocimiento de voz, o texto para un intérprete de lenguaje- en datos que la red pueda "entender" y en los que pueda trabajar.

El rendimiento de los métodos de *machine learning* depende fuertemente de la forma de representar estos datos complejos, en consecuencia [27]. En redes neuronales convencionales, esto se traduce en el empleo de un gran esfuerzo en preprocesar y transformar dichos datos, con el fin de adecuarlos al sistema concreto; contradiciendo el principio de querer que la inteligencia artificial aprenda por sí sola como lo haría un ser humano. Por lo tanto, lo ideal sería diseñar un sistema que fuese capaz de *abstraerse* lo suficiente de los datos principales, mediante mecanismos generales que permitan transformar dichos datos y reconocer características y patrones en ellos sin necesidad alguna, en principio, de supervisión, y garantizando un buen resultado dentro de los métodos de *machine learning*. A todo este proceso se le conoce como *representation learning*, o *aprendizaje de características*, y uno de los mecanismos más populares para afrontarlo es el *aprendizaje profundo* o *deep learning*.

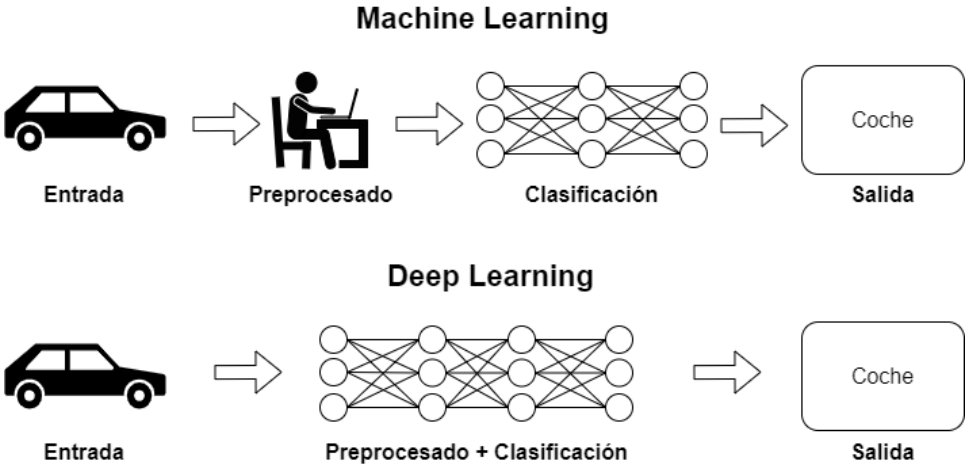


Figura 2-11. *Machine Learning* contra *Deep Learning*

Existen diversas definiciones de lo que es el *deep learning*. Generalmente, se conoce así a la tecnología que usa redes neuronales con múltiples capas, y cuyo objetivo es ser capaz de identificar características de alto nivel de un conjunto de datos sin procesar [28]. Su aplicación es muy variada: desde visión artificial, como ya hemos visto, hasta reconocimiento del habla, traducción, bioinformática, diseño de medicamentos o análisis de imágenes médicas; entre otras. Mediante el *deep learning*, los datos se van transformando capa a capa, derivando en una representación abstracta que resulta en una mejora de la eficiencia y la capacidad de la red. En la Figura 2-12 vemos un sencillo ejemplo de cómo una red neuronal que usa aprendizaje profundo actúa sobre una imagen.

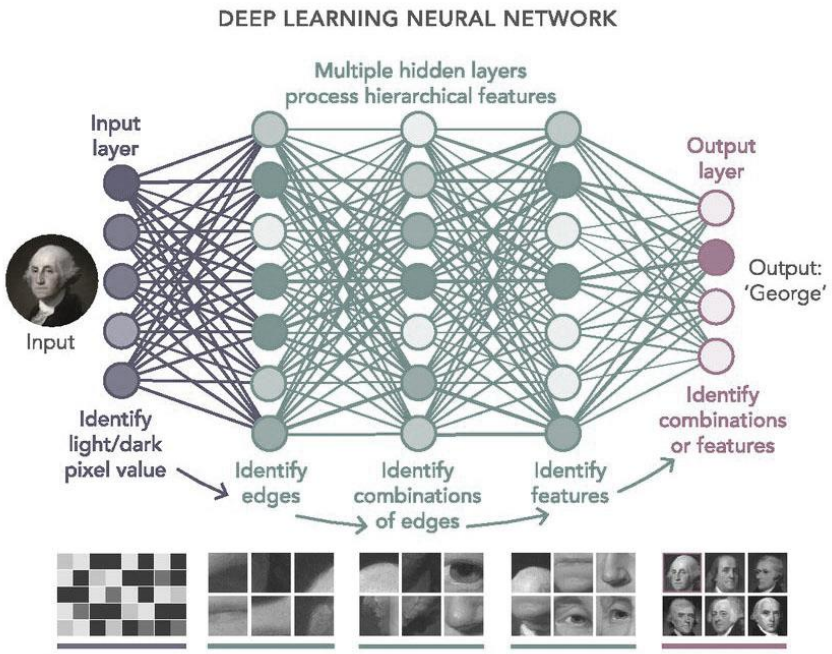


Figura 2-12. Ejemplo de red neuronal con *machine learning*

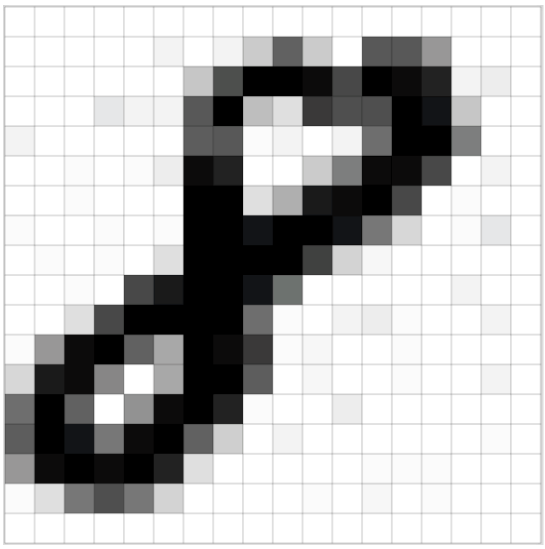
Cuando la red esté entrenada, cada capa se habrá especializado en un patrón o característica común de las imágenes que hayamos usado como entrenamiento, desde diferencias de contraste hasta figuras concretas, como facciones o siluetas. Y lo más importante es que no habrá requerido de intervención humana para tratar los datos en un primer momento, sino que podrá trabajar directamente sobre dichos datos en cuestión, independientemente de cuáles sean estos.

Las capas de una red neuronal con *deep learning* realizan diferentes operaciones en función de los datos que vayan a recibir. Así, aunque el principio es el mismo, una red diseñada para reconocimiento de imágenes no trabajará de la misma forma que una diseñada para, por ejemplo, interpretación del lenguaje o traducción. Por ello, a continuación explicaremos aquellas redes diseñadas específicamente para trabajar con imágenes, que, como ya adelantábamos anteriormente, se engloban dentro de las *redes neuronales convolucionales*, o CNNs.

2.3.1 Redes Neuronales Convolucionales

A primera vista, una CNN no difiere excesivamente de una red neuronal común básica como la que hemos explicado previamente. Sus neuronas trabajan con pesos y sesgos de la misma forma, se conectan entre ellos en un flujo unidireccional organizadas en capas, y calculan su activación y posterior error siguiendo los mismos mecanismos de suma ponderada y filtro mediante función objetivo. Lo que las distingue es que las redes convolucionales asumen que sus entradas serán imágenes (estrictamente, *datos bidimensionales* que serán interpretados como una matriz; más adelante entenderemos el motivo), por lo que podemos diseñar su arquitectura específica con la que incrementar la precisión y reducir los tiempos de entrenamiento e inferencia y el coste computacional de la red.

Para poder comenzar a explicar cómo funcionan estas redes, primero tendremos que hacer una reflexión sobre cómo deberíamos tratar las imágenes que vamos a utilizar como datos. Para nosotros, una imagen es el resultado de la percepción de la luz a través de nuestros ojos por parte de nuestro cerebro; pero en una máquina, una imagen no es más que una *matriz de píxeles*, siendo a su vez un píxel la unidad mínima de información que compone una imagen digital. Visualmente, podemos representar un píxel como un punto con color; pero como mencionábamos al comienzo de esta sección, para poder utilizar una red neuronal debemos *traducir* nuestros datos a lenguaje matemático que la red pueda interpretar. En este caso, podríamos asignarle a cada color un número, y convertir nuestra matriz de píxeles en una matriz numérica, como se representa en la siguiente figura:



0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	12	0	11	39	137	37	0	152	147	84	0	0	0
0	0	1	0	0	0	41	160	250	255	235	162	255	238	206	11	13	0
0	0	0	16	9	9	150	251	45	21	184	159	154	255	233	40	0	0
10	0	0	0	0	0	145	146	3	10	0	11	124	253	255	107	0	0
0	0	3	0	4	15	236	216	0	0	38	109	247	240	169	0	11	0
1	0	2	0	0	0	253	253	23	62	224	241	255	164	0	5	0	0
6	0	0	4	0	3	252	250	228	255	255	234	112	28	0	2	17	0
0	2	1	4	0	21	255	253	251	255	172	31	8	0	1	0	0	0
0	0	4	0	163	225	251	255	229	120	0	0	0	0	0	11	0	0
0	0	21	162	255	255	254	255	126	6	0	10	14	6	0	0	9	0
3	79	242	255	141	66	255	245	189	7	8	0	0	5	0	0	0	0
26	221	237	98	0	67	251	255	144	0	8	0	0	7	0	0	11	0
125	255	141	0	87	244	255	208	3	0	0	13	0	1	0	1	0	0
145	248	228	116	235	255	141	34	0	11	0	1	0	0	0	1	3	0
85	237	253	246	255	210	21	1	0	1	0	0	6	2	4	0	0	0
6	23	112	157	114	32	0	0	0	0	2	0	8	0	7	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figura 2-13. Matriz de píxeles interpretada numéricamente

Ahora, tenemos una matriz numérica de 18x18 valores, algo con lo que sí podríamos alimentar nuestra red neuronal. En este caso concreto, podríamos diseñar una capa de entrada con 18*18=324 neuronas, cada una "dedicada" a uno de los píxeles de nuestra imagen. Si además asumimos que tenemos una red totalmente conectada, cada neurona de una hipotética primera capa oculta tendría 324 pesos, uno por cada neurona de entrada. Esto sería algo aceptable en cuanto a cantidad de pesos (N*324, siendo N el número de neuronas en la primera capa oculta), pero estamos partiendo de un ejemplo muy simple, de una imagen muy pequeña, y, además, en blanco y negro; se puede prever que este modelo estándar con neuronas totalmente conectadas no escalará bien si complicamos la imagen.

Supongamos que tuviéramos algo más real, por ejemplo, una foto de carnet de 378x508 píxeles, y a color, pues queremos diseñar una red neuronal para reconocimiento facial. Podemos reutilizar la idea de crear una matriz de valores, pero para el color, ya no nos valdría con un valor numérico representativo como el de antes, porque ahora no sólo diferenciamos entre tonos de blanco y negro, sino en todo el espectro visual. Para solventar este problema, generalmente se utiliza un sistema de separación de colores en *canales*, de tal manera que, para cada canal de color, los valores que asignemos a cada píxel representan la intensidad de éste. La suma de todos los canales de color, daría la imagen original. Un modelo de canales de color muy extendido es el RGB, que crea tres canales, uno para el rojo, otro para el verde, y el último para el azul (que son los colores primarios de la luz, a su vez). Así, tras nuestra conversión de la imagen a valores numéricos, tendríamos tres matrices de 378x508 números; o lo que es lo mismo, un *volumen* de entrada de 378x508x3. Si mantenemos el mismo modelo de red neuronal en el que cada valor de entrada tiene asignado una neurona, el total de éstas en nuestra capa de entrada se incrementaría a 576072, que se traduce en que cada neurona de la primera capa oculta tendrá 576072 pesos a utilizar, y por lo tanto, 576072 operaciones a ejecutar; y todo esto sólo en *una* neurona de la primera capa. A la falta de escalabilidad, se le suma también el coste computacional, que se verá incrementado a medida que aumentemos el tamaño de la imagen. Además, generalmente no conviene tener demasiados parámetros cuando tratamos redes neuronales, ya que suele derivar en sobreajustes.

Para eludir este problema, los modelos de redes neuronales convolucionales siguen los siguientes pasos:

- Las neuronas de las capas están organizadas **tridimensionalmente**: poseen *anchura*, *altura* y *profundidad*. Esto quiere decir que recibirán un **volumen** tridimensional como entrada, también conocido como **tensor**, y generarán otro como salida. Por ejemplo, la primera capa de la red que diseñábamos antes tenía una entrada tridimensional de 378x508x3, y podría producir una salida de 189x254x5. Hay que destacar que, en este caso, las dimensiones de la entrada tienen un valor físico identificable (anchura, altura, canales de color), pero cuando ésta se va transformando a lo largo de la red, esto no tiene por qué ser así (la red se va *abstrayendo*).
- Las neuronas de capas posteriores sólo se conectarán a una pequeña región de las de la capa anterior, en vez de estar conectadas todas con todas, como hasta ahora hemos estado viendo. Esto no sólo reduce la cantidad de parámetros y operaciones a realizar, sino que también permite a subgrupos concretos de

neuronas especializarse en características concretas de las imágenes.

Como podemos observar, ambos principios cumplen los objetivos del *deep learning* que desarrollábamos antes: abstracción y capacidad de aprender características específicas. Para lograrlo, las redes convolucionales presentan una serie de capas que realizan operaciones específicas, y éstas a su vez se ordenan siguiendo una arquitectura definida, como la que se representa en la siguiente figura:

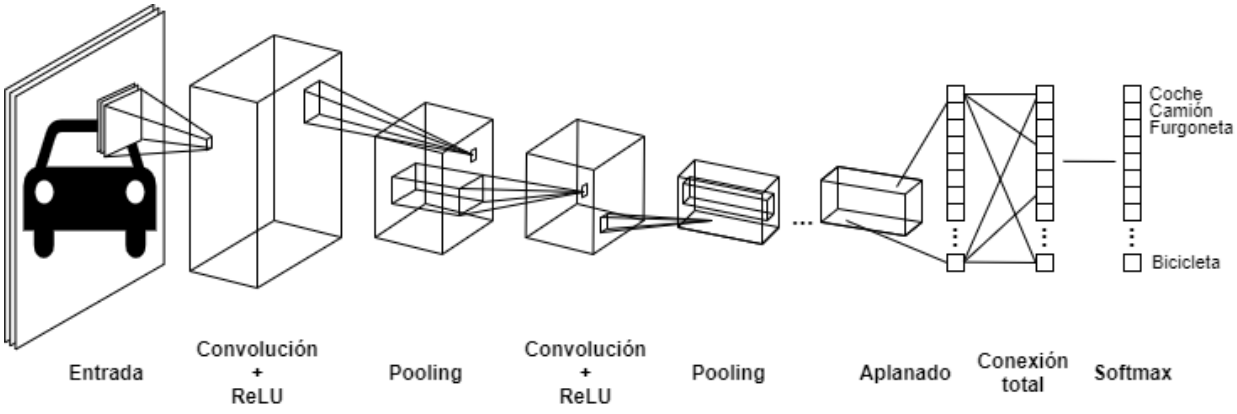


Figura 2-14. Ejemplo de red convolucional general

Algunas de estas capas ya las hemos comentado en apartados y secciones anteriores. La entrada, por ejemplo, ya sabemos que será un volumen de WxHx3, donde *W* será la anchura, y *H* la altura, en píxeles; y 3 se refiere al número de canales de color que conforman la imagen. Las funciones *ReLU* y *Softmax* también las hemos visto previamente: la primera se usará para filtrar por valores (recordemos que la función ReLU realiza una operación *max(umbral, entrada)*); y la segunda, para generar una probabilidad para todas las opciones posibles de salida, y determinar cuál es la más adecuada.

El resto de capas será lo que explicaremos a continuación, detallando su labor, características principales e hiperparámetros relacionados.

2.3.1.1 Capa convolucional

La capa convolucional (*convolutional layer*) es uno de los puntos clave de este tipo de redes, y lleva a cabo las operaciones más pesadas computacionalmente. Está encargada de detectar elementos característicos en las imágenes, como bordes, zonas de cierto color o figuras. Para ello, se vale de una serie de filtros que operan sobre los datos de entrada, mediante un proceso que se explica a continuación:

1. Se define el *tamaño* de filtro, determinado por su altura y anchura, y extendido en toda la profundidad del volumen de entrada. Todos los filtros tendrán el mismo tamaño.
2. El filtro se *desliza* a lo largo y ancho del volumen de entrada, calculando los productos escalares de los valores de entrada del área definida para el filtro respecto a los parámetros de éste, y generando en cada aplicación una salida. A este proceso se le llama *convolución*. Cada filtro resultará en un mapa bidimensional, cuya altura y anchura será, en general, menor que la del volumen de entrada.
3. Cuando todos los filtros sean aplicados, tendremos un conjunto *K* de mapas bidimensionales, uno por cada filtro; o lo que es lo mismo, un volumen de salida de W'xH'xK, donde *W'* y *H'* representan las nuevas anchura y altura, y *K* el número de filtros aplicados.

Con este sistema, las neuronas ya no necesitarán estar conectadas todas con toda en capas sucesivas. Cada una de ellas se conectará exclusivamente a las salidas de la capa anterior que forman parte del tamaño de su filtro. Veámoslo más claro con un ejemplo: si partimos de nuestra foto de carnet anterior, que se traducía en un volumen de entrada de 378x508x3; y fijamos el tamaño del filtro en 5x5 (recordemos, la profundidad siempre equivale a la del volumen de entrada, por lo que no se especifica), cada neurona cubrirá una región de 5x5x3 de la entrada, lo que equivale a 5*5*3=75 pesos; bastante menos de los cerca de medio millón que se necesitaban en el modelo totalmente conectado. Al conjunto de interconexiones de una neurona con el volumen de entrada

(a su vez salida de las neuronas de la capa anterior) se le conoce como *campo receptivo*, y es uno de los hiperparámetros que se definen para las capas convolucionales. Más adelante en la Figura 2-17 se deja un ejemplo gráfico de la capa de convolución, para más claridad.

Queda por discutir el número de neuronas que forman la capa, y que determinan el tamaño del volumen de salida. Este se controla, de la misma forma que el número de interconexiones, mediante hiperparámetros, en este caso, tres:

- **Profundidad (*Depth*):** Equivale al número de filtros que usaremos, cada uno especializado en ciertas características de la entrada, como veíamos antes. Por cada filtro, tendremos un grupo de neuronas que estarán conectadas a la misma área del volumen de entrada; o lo que es lo mismo, compartirán el campo receptivo, y se diferenciarán en la configuración de los parámetros de sus filtros. A este grupo de neuronas se le suele llamar *columna de profundidad* (*depth column*) o *fibra*.
- **Paso (*Step*):** Como hemos mencionado previamente, cada filtro se *desliza* a lo largo y ancho del volumen de entrada. El paso regula cuántos "saltos" se dan al deslizar dicho filtro. Por ejemplo, si estuviéramos en la primera capa y nuestro volumen de entrada fuese la imagen, el paso determinaría cuántos píxeles dejamos entre filtro y filtro. Si el valor fuese 1, se deslizaría un píxel cada vez; si fuera 2, se deslizaría dos píxeles. En la siguiente figura se representa de forma más clara:

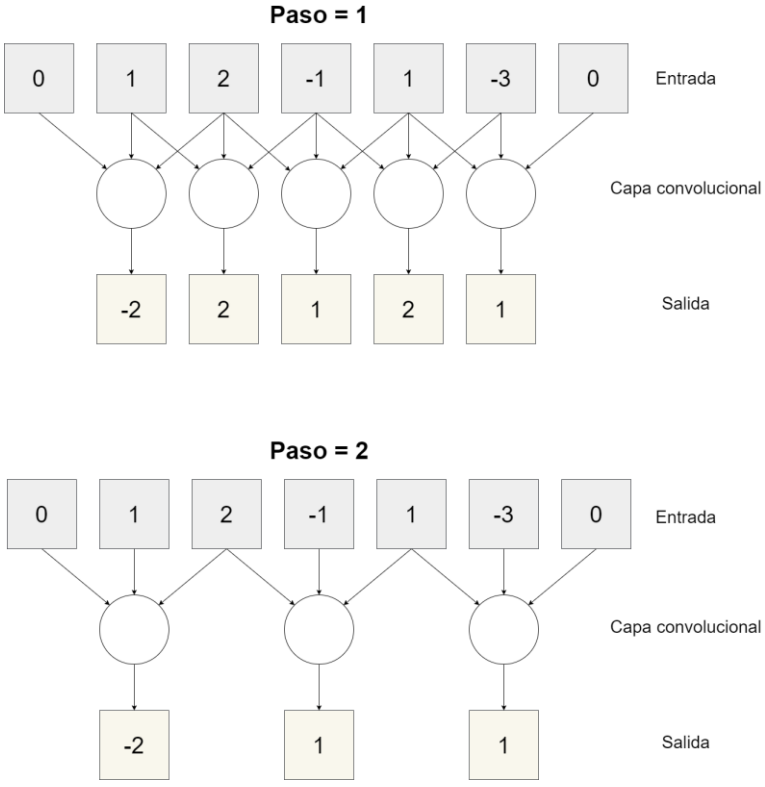


Figura 2-15. Ejemplo de capa convolucional con distintos pasos

Este hiperparámetro modifica el tamaño bidimensional del volumen de salida (altura y anchura), como puede apreciarse en la figura. A mayor número de pasos, se realizaran menos filtrados, y se reducirá a su vez el número de neuronas; pero es notable que con un paso igual a 1, que es el mínimo, también estemos variando el tamaño. Para evitar este efecto, se define el hiperparámetro a continuación.

- **Relleno con ceros (*Zero-padding*):** Se utiliza principalmente para mantener la relación de tamaño de los volúmenes de entrada y salida, y para evitar perder la información en los bordes del volumen. Consisten en "rodear" las matrices de entrada con ceros, cuyo *grosor* es controlado por este hiperparámetro. Si nos fijamos en los vectores de entrada de las redes de la Figura 2-15, veremos que tienen un *zero-padding* igual a 1, es decir, están rodeados por una única capa de ceros. Gracias a esto, se puede observar que el tamaño de la entrada original (que, ignorando los ceros, es 5), se mantiene en

el caso del paso igual a 1. En el caso del paso igual a 2 no es así, pero ajustar el relleno permite aumentar el número de neuronas de la capa, que sin éste, sería de dos.

La siguiente fórmula puede utilizarse para determinar cuál será la salida de cierta capa de convolución (y, por lo tanto, el número de neuronas de la capa) a partir de sus hiperparámetros, siendo W y H el ancho y alto del volumen de entrada, respectivamente; F el tamaño del campo de recepción (o el tamaño del filtro), S el paso aplicado, P el relleno usado, y K el número de filtros:

$$\left(\frac{W - F + 2P}{S} + 1\ ;\ \frac{H - F + 2P}{S} + 1\ ;\ K\right) \tag{2-8}$$

Como ejemplo podemos usar la arquitectura de Krizhevsky [29], que acepta imágenes de 227x227x3, y que en su primera capa de convolución utiliza neuronas con un campo de recepción de 11x11, un paso de 4, sin relleno de ceros, y una profundidad de 96. Si aplicamos la fórmula, tendremos un volumen de salida 55x55x96, con 55*55*96=290400 neuronas conectadas a áreas de 11x11x3 del volumen de entrada.

Cabe destacar también que estos hiperparámetros presentan ciertas restricciones mutuas. Así, con un espacio de entrada de 10x10, filtros de 3x3 y sin relleno de ceros, el valor del paso no podría ser 2, por ejemplo, ya que aplicando la fórmula para la altura y anchura de la salida vemos que no nos sale un número entero (en este caso saldría 4'5). Técnicamente, esto se traduce en que las neuronas no "encajan" ordenada y simétricamente en la entrada. La configuración de los hiperparámetros de las capas convolucionales puede llegar a ser compleja debido a esto, y generalmente se usará el relleno con ceros para aliviar la situación.

Para finalizar, volvamos a echar un vistazo a la arquitectura de Krizhevsky. Como decíamos, en su primera capa convolucional ésta presenta un total de 290400 neuronas (N), y como el campo de recepción es de 11x11, cada neurona tendrá un total de 11*11*3=363 pesos (W), más un sesgo. Esto hace un total de 105705600 parámetros (N*W), es decir, alrededor de cien millones. Comparado con lo que obtendríamos en el modelo de interconexión total (donde cada neurona tendría por si sola 227*227*3=154587 parámetros, a multiplicar por la misma cantidad de neuronas), es una inmensa mejoría, pero sigue siendo un número muy alto.

Resulta que esto se puede reducir si hacemos la siguiente suposición: que si una característica de la entrada, obtenida mediante la aplicación de cierto filtro por parte de cierta neurona, es útil en una posición arbitraria (x_1, y_1), también puede serlo en otra aleatoria (x_2, y_2). En otras palabras: si un filtro detecta en una zona de la imagen un borde puntiagudo, y por lo tanto la neurona pertinente aprende a reconocerlo, podemos asumir que cualquier otra neurona de la capa también se beneficiaría de poder detectar dicha característica. Así, podríamos utilizar un único grupo de pesos y sesgos para todas las neuronas en cada nivel de profundidad de la capa (en cada filtro), reduciendo considerablemente la cantidad de parámetros totales. A este conjunto de neuronas que comparten parámetros se las conoce como *rebanadas de profundidad* (*depth slices*), y generalmente equivalen a la profundidad de la propia capa. Al proceso completo, se le conoce como *compartición de parámetros* (*parameter sharing*).

Veamos cómo aplica en el caso anterior. La profundidad de la primera capa de Krizhevsky vale 96, o lo que es lo mismo, disponemos de 96 filtros diferentes, y 96 grupos de 55x55 neuronas. Antes, cada neurona disponía de sus propios parámetros, lo que nos daba la cifra de cien millones en total. Pero si ahora asumimos que las neuronas de cada rebanada de profundidad compartirán parámetros, tendremos 11*11*3=363 parámetros, a multiplicar por cada nivel de profundidad, es decir 363*96=34848 (sin contar los sesgos). Esto es apenas un 0'33% comparado con el caso anterior, que se traduce en una disminución considerablemente alta del número de parámetros que supone un gran alivio en la computación.

Teniendo esto en cuenta, la capa convolucional puede modelarse matemáticamente como la convolución de los pesos de las neuronas respecto al volumen de entrada, y de ahí obtiene su nombre; y también es la razón por la que se suelen llamar a dichos pesos *filtros* o *kernels*. En la Figura 2-16 se puede ver la representación gráfica de los filtros de la primera capa convolucional de Krizhevsky en cierta ejecución (recordemos, 11x11, y como tiene una profundidad de 3 derivada de los canales de color de las imágenes de entrada, podemos representarlos gráficamente). Se puede ver como cada filtro se especializa en un detalle distinto, como bordes o cambios bruscos de color.

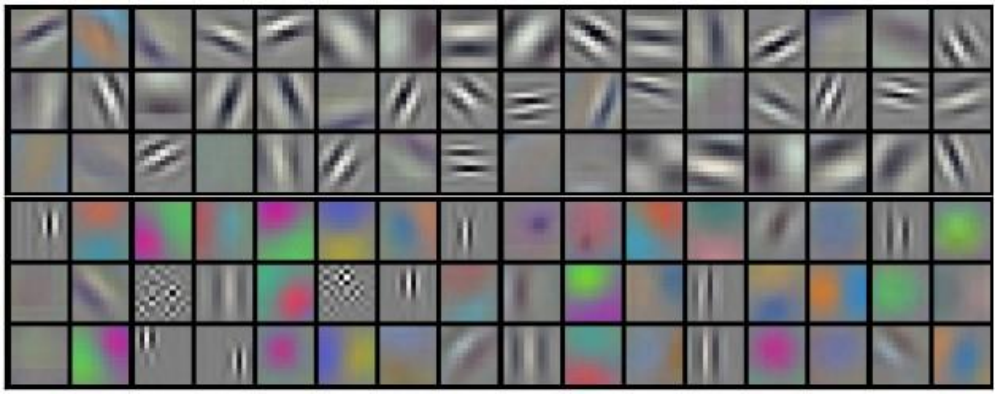


Figura 2-16. Filtros de primera capa convolucional en una red de Krizhevsky

Por otro lado, cabe destacar que no siempre es inteligente compartir parámetros. Podría darse la situación de tener un conjunto de imágenes de entrada con una forma específica, quizá con una estructura concreta que no hiciera útil el reconocer ciertas características en *todas* las zonas de la imagen. El caso que poníamos al comienzo de la sección, donde usábamos fotos de carnet, podría servirnos como ejemplo: si las caras de las personas están centradas en la imagen, algunas características –como la forma de los ojos o los bordes de las caras- no serían útiles más que en su posición concreta. En estos casos, esta compartición de parámetros se limita, quizá permitiendo compartir tan sólo en algunas áreas específicas, o directamente sin ningún tipo de compartición. A este tipo de capas se las suele conocer como *capas localmente conectadas* (*locally-connected layer*).

Para finalizar con las capas convolucionales, dejamos el siguiente resumen gráfico de todo el proceso en la siguiente figura. En éste podemos observar un volumen de entrada de 5x5x3, con un *zero-padding* igual 1, filtros de tamaño 3x3, dos capas de profundidad y un paso igual a 2, que podríamos haber despejado de la fórmula antes comentada. Hay compartición de parámetros, por ello, tan sólo se representan dos volúmenes de filtro, de tamaño 3x3x3. El resultado final será la suma del producto de cada posición del volumen de entrada por su filtro correspondiente:

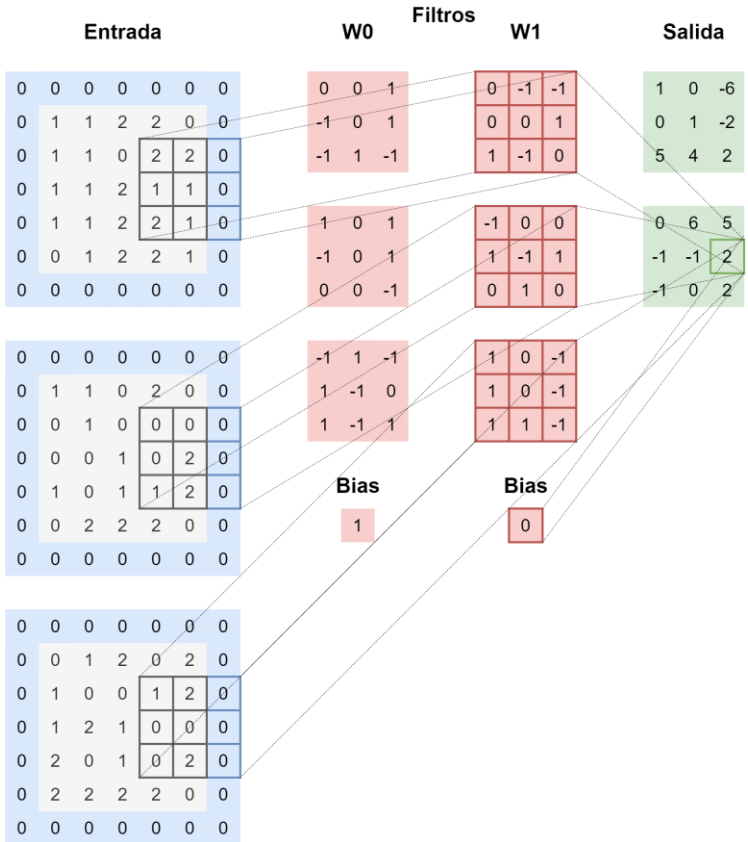


Figura 2-17. Ejemplo gráfico de funcionamiento de una capa convolucional

2.3.1.2 Capa de agrupación

La capa de agrupación (*pooling layer*) suele colocarse entre capas convolucionales sucesivas (como puede observarse en la Figura 2-14). Su labor es reducir el tamaño *espacial* (anchura y altura) del volumen de entrada, sin afectar a su pronfundidad, con el fin de reducir el número de parámetros, y con ello el costo computacional y el sobreajuste. Para ello, se seleccionan pequeñas áreas bidimensionales del volumen, y se operan de tal forma que generen una única salida. Paralelamente a la capa convolucional, este pequeño espacio se deslizará a lo largo y ancho de cada nivel de profundidad del volumen, construyendo así poco a poco la salida. Por lo tanto, podemos reutilizar algunos de los hiperparámetros de dicha capa, concretamente, el tamaño del filtro y el paso. El volumen de salida tendrá la forma:

$$\left(\frac{W-F}{S}+1\,;\,\frac{H-F}{S}+1\,;K\right) \tag{2-9}$$

En líneas generales, los valores típicos de F y S serán de 2 y 2, respectivamente; aunque una alternativa popular es $F=3$ y $S=2$, caso que se conoce como *agrupación superpuesta* (*overlapping pooling*).

Por último, queda por determinar cuál será la operación que se lleve a cabo sobre los valores en el área definida por el filtro. El objetivo es que no se pierda información relevante al estar minimizando el tamaño, por lo que existen varias alternativas: históricamente, se usó la media aritmética de los valores (*average pooling*), o la norma euclídea (*L2-norm pooling*), pero actualmente se usa con más frecuencia la operación *max*, es decir, de todos los valores del filtro, se escoge el más alto (*max pooling*). En la Figura 2-18 se muestra un ejemplo.

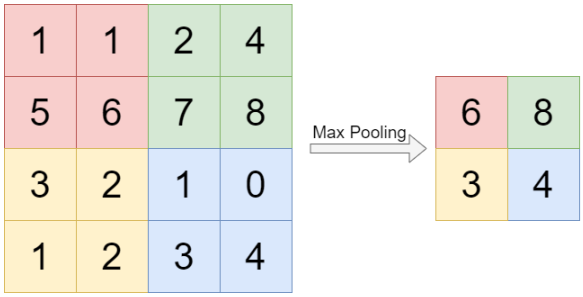


Figura 2-18. Ejemplo de *max pooling*

2.3.1.3 Capa totalmente conectada

La capa totalmente conectada (*fully-connected layer*) es, generalmente, la última sección de las redes convolucionales. Suelen consistir en más de una, como puede observarse al final de la Figura 2-14. Estas capas siguen un sistema más similar al que comentamos en el apartado 2.2, dedicado a la introducción a las redes neuronales. Ahora podemos olvidarnos de entradas y neuronas multidimensionales, y también de interconexiones parciales y filtros. Estas capas se usaran para decidir, finalmente, el resultado de la red, una vez la imagen de entrada ha reducido su tamaño lo suficiente y ha pasado por los filtros pertinentes. Por ejemplo, en un caso de reconocimiento de objetos, en el que lo que buscamos es, dada una imagen, determinar qué se observa en ella dentro de un grupo de posibles candidatos (a este conjunto se le suele llamar *clases*), estas últimas capas dispondrán de neuronas especializadas en cada una de las clases. Si usamos además como función de activación la función *softmax*, como adelantábamos al comienzo de la sección, tendremos un vector en el que cada clase aparecerá representada junto con la probabilidad de que dicha imagen pertenezca a ella. Más adelante veremos el modelo de YOLO, que no solo es capaz de reconocer objetos, sino también de ubicarlos en la imagen y marcar su posición.

Convertir de un volumen de entrada tridimensional a un vector es un proceso que se conoce como *aplanado* (*flatten*). La idea principal podría decirse que es no entender que estamos reduciendo el número de dimensiones, sino más bien que estamos pasando de WxHxK a 1x1xK'. Para ello, una posible solución es colocar una capa

convolucional cuyo tamaño de filtro sea igual al tamaño espacial del volumen de entrada, de tal manera que la salida tendrá que ser 1x1 necesariamente. Por ejemplo, el modelo de Krizhevsky llega a una última capa tridimensional con un volumen de 7x7x512, y, para *aplanarlo*, utiliza una capa convolucional con tamaño de filtro *F=7*, y 4096 niveles de profundidad; lo que produce una salida de 1x1x4096; o lo que es lo mismo, un vector de salida de 4096 valores. Éste puede alimentar fácilmente a una red neuronal clásica, totalmente conectada, que en su primera capa tuviera 4096 neuronas. El número de parámetros en una primera capa oculta sería de 4096*4096=16777216, un valor grande, pero sin duda, mejor del que sería utilizar en todo el proceso capas totalmente conectadas, recordando que el modelo de Krizhevsky aceptaba imágenes de 224x224x3.

La organización general de estas capas está muy bien representada en la Figura 2-14. La entrada suele conectarse directamente a una primera capa convolucional, que a su vez suelen ir seguidas de un filtro ReLU; y esta combinación suele ir seguida de una capa de agrupación. Este patrón se repite varias veces, con diferentes hiperparámetros en cada capa en función del diseño concreto; hasta desembocar en una red completamente conectada, de la que se extraerá la información que generará el resultado final. En muchas ocasiones, se conoce al proceso que ocurre antes del *aplanamiento* como "aprendizaje de características"; y al posterior como "clasificación".

Por último, respecto a los hiperparámetros, queda decir que sus valores suelen ser bastante comunes, aunque difieran ligeramente entre un diseño y otro. En general, las imágenes deberán tener un tamaño espacial que permita ser divido entre dos varias veces (224, 64, 32…), derivado de que las capas de agrupación, a su vez, suelen tener un tamaño de filtro y paso igual a dos. Las capas convolucionales, por su lado, acostumbran a usar filtros de entre 3x3 y 5x5, con pasos de 1, y dejando el relleno de ceros para ajustar el volumen de entrada sin perder extensión espacial a la salida. Tan sólo en las primeras capas, cuando se trabaja directamente con la imagen como entrada, se suele aumentar el tamaño de filtro y paso.

2.3.2 Redes convolucionales para detección de objetos

Las redes convolucionales estándar, como la que hemos venido explicando en los párrafos anteriores, suelen dedicarse a un campo de la visión artificial conocido como *reconocimiento de imágenes*, que ya definíamos por encima en el último aparatado dedicado a las capas totalmente conectadas. Recapitulando, éste consiste en, dadas una serie de clases (posibles resultados), determinar a cuál pertenece cierta imagen. Sin embargo, YOLO, que es el modelo utilizado en esta memoria, se aplica en *detección de objetos*, un campo que extiende el simple reconocimiento al *reconocimiento* y *señalización* de objetos en imágenes. Para ello, la salida de nuestra red deberá indicarnos, de la forma más aproximada posible, dónde se encuentra cada uno de los objetos detectados en la imagen de entrada (vemos que, ahora, no sólo podemos reconocer un solo elemento, sino varios). En la Figura 2-19 se puede ver una comparativa entre la salida de un sistema y otro:

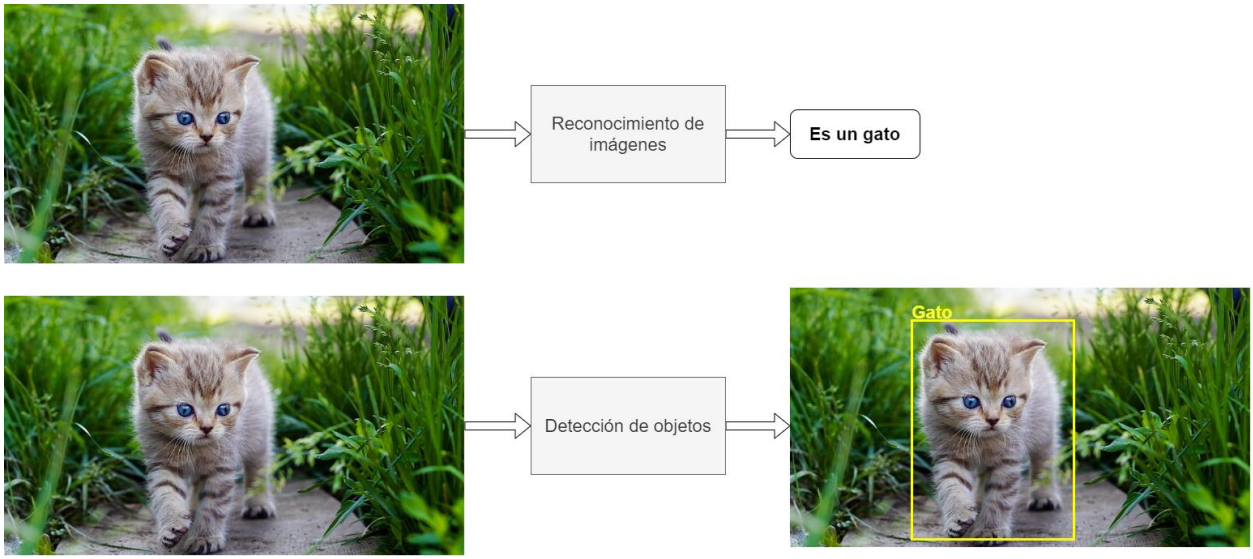


Figura 2-19. Comparativa reconocimiento de imágenes y detección de objetos

A continuación, listaremos y explicaremos brevemente algunos de los métodos principales dedicados a la detección de objetos; para luego dedicar la sección 2.4 a profundizar sobre el modelo concreto de YOLO. En concreto, ahora hablaremos sobre tres redes concretas, pioneras en la detección de objetos: *R-CNN, Fast R-CNN* y *Faster R-CNN*.

2.3.2.1 R-CNN

Una primera aproximación a la detección de objetos podría ser subdividir la imagen en concreto en múltiples regiones, y aplicar a cada una de ellas un sistema de reconocimiento individual, identificando así cada uno de los posibles objetos presentes. Así, la complejidad del mecanismo no sería en sí mismo la *detección*, sino el método que se utiliza para llevar a cabo dicha división de la imagen, pues cada región debería, idealmente, delimitar cada una de las figuras relevantes.

El método R-CNN, detallado por sus autores en la referencia [30], basa su mecanismo en proponer exactamente 2000 regiones, elegidas mediante el *algoritmo de búsqueda selectiva*. Este algoritmo se desarrolla en profundidad en la referencia [31], pero bastará con saber que es un método iterativo que aprovecha las diferencias de color, contraste y sombra identificadas en la imagen, junto con otros mecanismos matemáticos y de segmentación, para tratar de detectar formas y figuras que *puedan* pertenecer a un mismo concepto. Luego, las selecciones más probables serán las que formen las 2000 *regiones propuestas* (*region proposals*), como se puede observar en la figura.

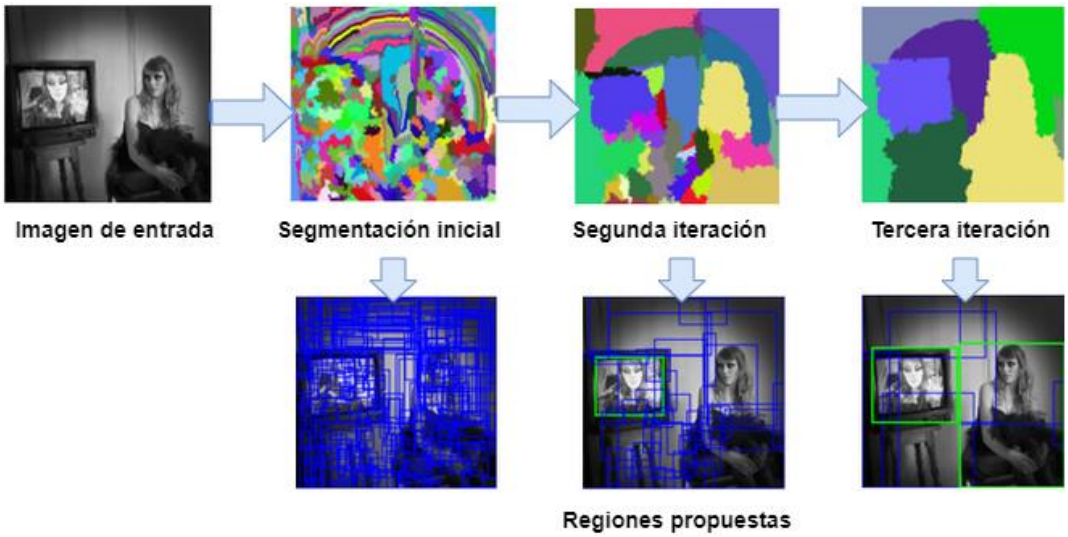


Figura 2-20. Algoritmo de búsqueda selectiva

Estos recuadros que surgen como resultados de la selección selectiva, alimentan, cada uno, a una red convolucional estándar (que, de hecho, es la de Krizhevsky) que genera un vector de salida de 4096 valores, donde todas las características relevantes de cada región habrán sido extraídas. A continuación, se utiliza una *máquina de vectores de soporte* (*Support Vector Machine, SVM*), un algoritmo de aprendizaje automático que se usa para clasificar [32] (es una alternativa más compleja a la función *softmax* que tantas veces hemos mencionado). Finalmente, tendremos, un vector de resultados compuesto por el objeto reconocido en cada región y su posición, generalmente representada por cuatro valores que ubican espacialmente el recuadro indicativo, llamado *cuadro* o *caja delimitadora* (*bounding box*, a veces abreviado como *bbox*). En la Figura 2-21 podemos ver una representación esquemática:

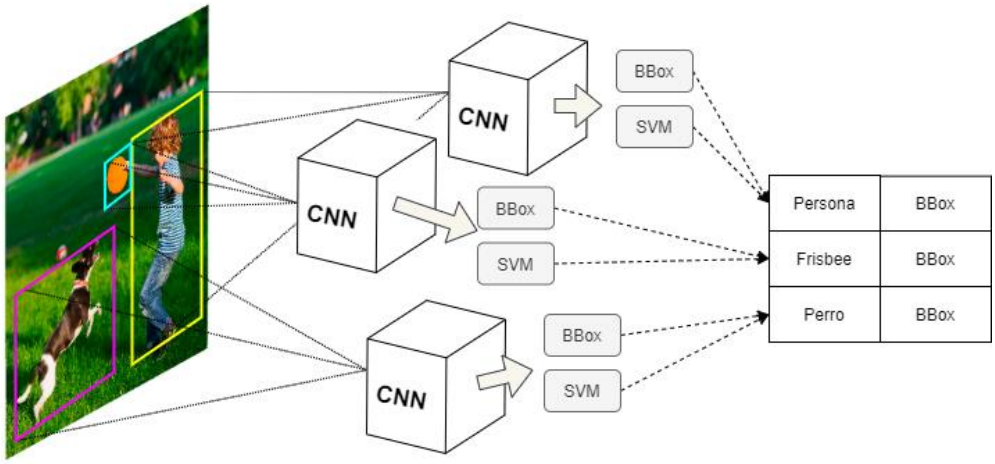


Figura 2-21. Esquema R-CNN

La principal desventaja de las R-CNN es que, pese a ser funcionales, son extremadamente lentas tanto en entrenamiento como en inferencia, ya que el proceso de búsqueda selectiva debe designar las 2000 regiones en cada imagen que se procese. Esto provoca que en inferencia, por ejemplo, se tomen alrededor de 47 segundos por imagen, que es bastante si tenemos en cuenta que muchos de estos mecanismos pretenden funcionar en detección sobre vídeo. A esto se le suma que el algoritmo de selección selectiva realmente no participa en el proceso de aprendizaje, ya que no tiene mecanismos para "aprender", lo que lo convierte en parte del pre-procesado, fase que, como ya comentamos al comienzo de la sección, conviene limitar o eliminar del todo en la medida de lo posible.

2.3.2.2 Fast R-CNN

Las desventajas de las R-CNN fueron solventadas por uno de sus propios autores en una nueva aproximación a la que se llamó *Fast R-CNN* [33] (en español, *R-CNN Rápida*). A diferencia de su predecesor, *Fast R-CNN* primero pasa la imagen completa por una capa convolucional, de donde se extraerá un mapa de características en el que se identificarán las diferentes regiones propuestas. Dicha región propuesta tendrá un tamaño arbitrario, por lo que usaremos una capa de agrupación para redimensionarla a un volumen más regular. A esta capa se la conoce como *capa de agrupación por región de interés* (*Region of Interest pooling layer*, muchas veces abreviado como *RoI pooling layer*), y su sistema de agrupación es idéntico al de las *max pooling layers*, es decir, utiliza la función *max* para discriminar. La diferencia principal, es que la *RoI* puede tener un tamaño de filtro variable, en función del tamaño de la región propuesta. En la siguiente figura puede verse un ejemplo de funcionamiento:

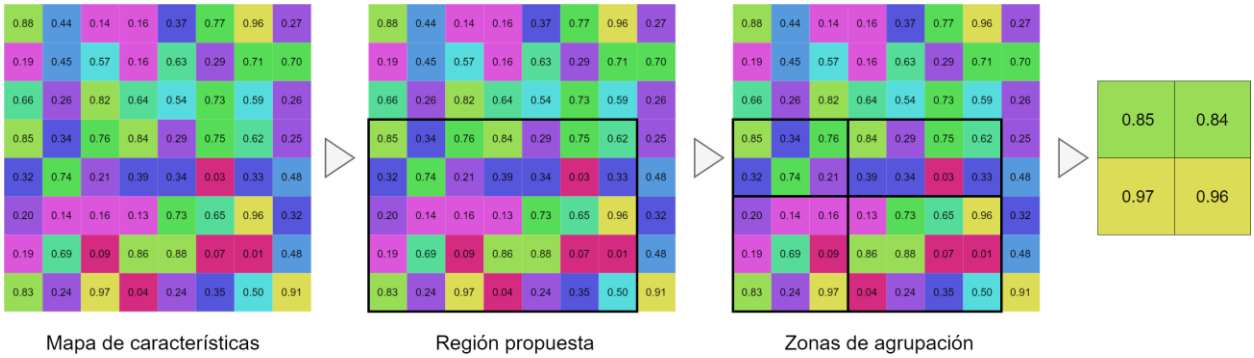


Figura 2-22. Capa de agrupación RoI

Tras este paso, aplanaremos el volumen de salida de cada región propuesta, y alimentaremos una red neuronal totalmente conectada, paralelo a lo que hacíamos en R-CNN. De ésta última fase obtendremos dos resultados:

el primero lo pasaremos por un filtro *softmax*, para clasificar el objeto identificado (a diferencia de R-CNN, donde usábamos SVM); y el segundo, pasará por un *regresor de caja delimitadora* (*Bounding Box Regressor*), un algoritmo utilizado para definir con más precisión el recuadro localizador del objeto [34].

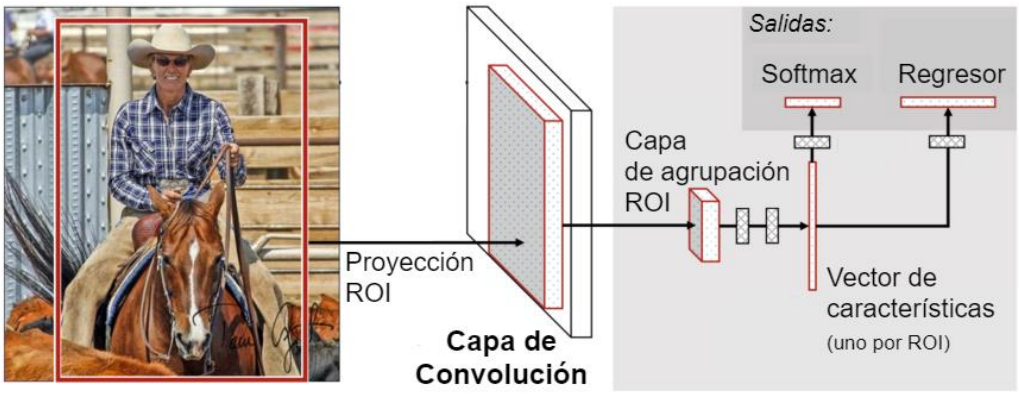


Figura 2-23. Arquitectura de Fast R-CNN

Fast R-CNN es más rápido que el R-CNN común porque no requiere generar las 2000 regiones propuestas sobre la imagen de entrada, ahorrando tiempo de computación en las primeras capa de la red. Sin embargo, no es perfecta, pues, para encontrar las regiones propuestas (tras la primera capa convolucional) vuelve a utilizar el algoritmo de búsqueda selectiva, que, si recordamos, no es dinámico y no puede aprender. En consecuencia, pese a la mejora en velocidad, seguimos ante una práctica no aconsejable.

2.3.2.3 Faster R-CNN

Con el fin de solventar el problema del uso de algoritmos para la identificación de regiones propuestas, se diseñó *Faster R-CNN* [35] (en español, *R-CNN Más Rápido*). El comienzo del algoritmo es igual que se predecesor: la imagen se pasa por una red convolucional que devuelve un mapa de características sobre el que identificar regiones propuestas. Sin embargo, ahora, en vez de usar un algoritmo estático, usaremos una red neuronal separada para predecir dichas regiones. El resto, se mantiene igual: una vez obtenidas las regiones, antes con el algoritmo, ahora con la red neuronal alternativa, utilizamos una capa de agrupación RoI y luego una capa totalmente conectada con la que obtener los objetos reconocidos y sus cuadros delimitadores.

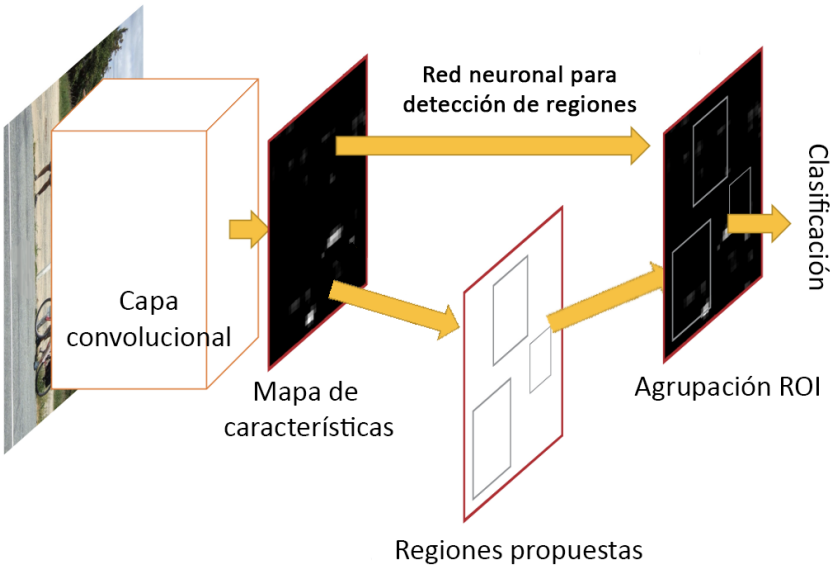


Figura 2-24. Esquema de arquitectura de Faster R-CNN

La comparativa entre los tres modelos deja claro cuál la opción más viable, si priorizamos tiempos de ejecución:

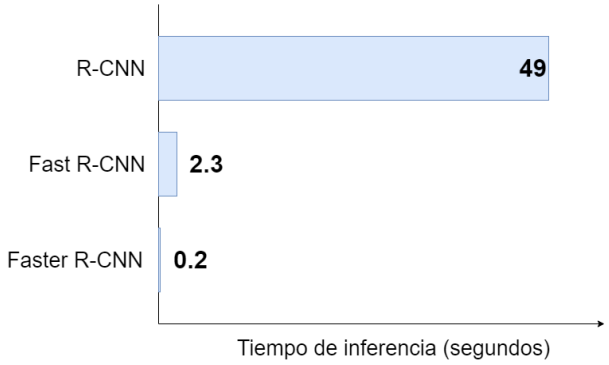


Figura 2-25. Comparativa distintos métodos de R-CNN

Con *Faster R-CNN*, terminamos el grupo de algoritmos introductorios a la detección de objetos. Tan sólo hemos explicado brevemente los más básicos, aquellos que se basan principalmente en la identificación de regiones y su posterior clasificación. Como comentábamos al comienzo de la sección, ésta es una aproximación muy sencilla, y se ha demostrado que puede llegar a ser eficaz. Sin embargo, los algoritmos que actualmente se utilizan y forman parte del estado del arte presentan otros mecanismos más avanzados para llevar a cabo el objetivo, la mayoría centrados en el uso de una única red neuronal que interprete toda la imagen en su conjunto, y no tan sólo sus subdivisiones, y diseñados para permitir su utilización en la detección de objetos *en tiempo real* (como se utilizaría, por ejemplo, en una cámara de seguridad inteligente o un coche autónomo). Algunos ejemplos son las SSD (*Single Shot MultiBox Detector*), RefineDet (*Single Shot Refinement Neural Network*), RetinaNet; o YOLO, en la que nos centraremos a continuación.

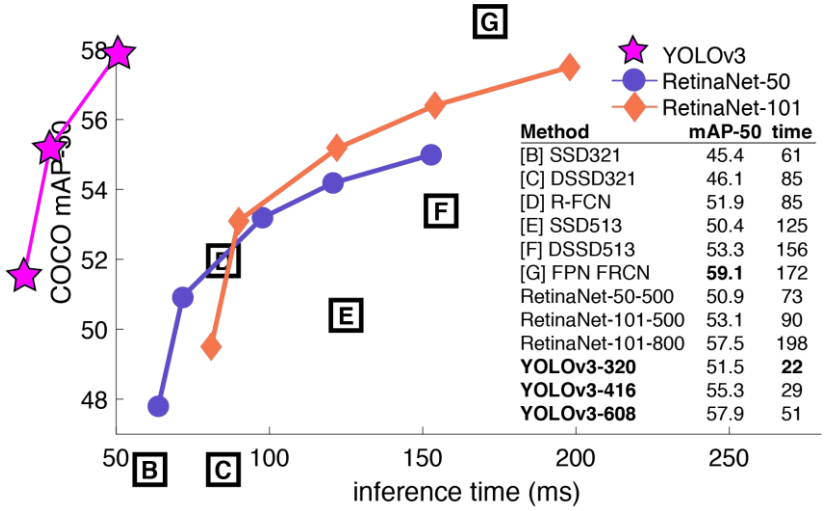


Figura 2-26. Comparativa de métodos de detección de objetos en tiempo real del estado del arte

2.4 YOLO

YOLO son las siglas de *You Only Look Once*, que se traduce como *"sólo miras una vez"*, y se podría decir que ésta es su principal premisa. Como se puede observar en la Figura 2-26, es un sistema muy eficiente, comparado con otros mecanismos también del estado del arte. Compensa una probabilidad de acierto media (mAP) ligeramente inferior con una potente velocidad de inferencia; más adelante veremos el porqué de ambas características.

Existen actualmente cuatro versiones de YOLO. La primera y original se llamó simplemente YOLO, aunque aquí usaremos YOLOv1 para referirnos a la primera versión del sistema, y YOLO para cualquier versión de forma general [36]. Definió los principios del detector, que entraremos a detallar en los siguientes párrafos. La

segunda versión, conocida como YOLOv2 o YOLO9000 [37], presentó algunas mejoras, como el uso de capas de normalización o la introducción de cuadros delimitadores predefinidos para aumentar la velocidad de aprendizaje de la red. YOLOv3 [38], tercera versión y la que usaremos en esta memoria, trató de mejorar uno de los principales problemas que YOLO arrastraba desde su comienzo: la detección de objetos pequeños (cuando veamos el funcionamiento interno, entenderemos el por qué). Finalmente, en abril de 2020 se presentó YOLOv4 [39], que busca convertirse en el sistema más rápido para detección en tiempo real.

2.4.1 YOLOv1

Como decíamos, la premisa de YOLO es que *sólo se mira una vez*. En YOLO, una sola red neuronal se encarga de interpretar toda la imagen, con lo que disminuye el tiempo de aprendizaje. Además, en YOLO se lleva a cabo una separación entre los conceptos de *clasificación* y *detección de cuadros delimitadores*, tareas que se realizan paralelamente para aumentar la velocidad general del sistema (tanto en inferencia como en aprendizaje). De esta manera, YOLO predecirá de forma separada *qué* objetos aparecen en la imagen y *dónde* se encuentran; para luego combinar ambas predicciones en un resultado definitivo.

Entrando en detalles, YOLO parte de una división de la imagen de entrada en una cuadrícula regular fija de ***SxS*** segmentos, donde cada uno se dedicará a la predicción, por separado, de ***un*** único objeto y de ***B*** cuadros delimitadores (tal y como mencionábamos antes, por un lado se detecta qué es, y por otro dónde está), como se puede observar en la Figura 2-27. Cabe destacar que, pese a la división en segmentos, esto no quiere decir que dicho objeto o cuadro delimitador deba estar completamente limitado en dicho recuadro, sino que tan sólo debe aparecer el *centro* de éste. Varios segmentos contiguos podrían detectar el mismo objeto y la misma caja delimitadora si el mismo objeto ocupase varias celdas, sin perjuicio para el resultado final. Por otro lado, la restricción de detectar un solo objeto por segmento sí limita la capacidad de YOLO, estando acotado el número máximo de posibilidades a S*S objetos. Es por ello que una de las debilidades más conocidas del sistema es su incapacidad para reconocer objetos pequeños, ya que acaban encuadrados en una misma celda, y tan sólo uno de los dos podrá ser detectado.

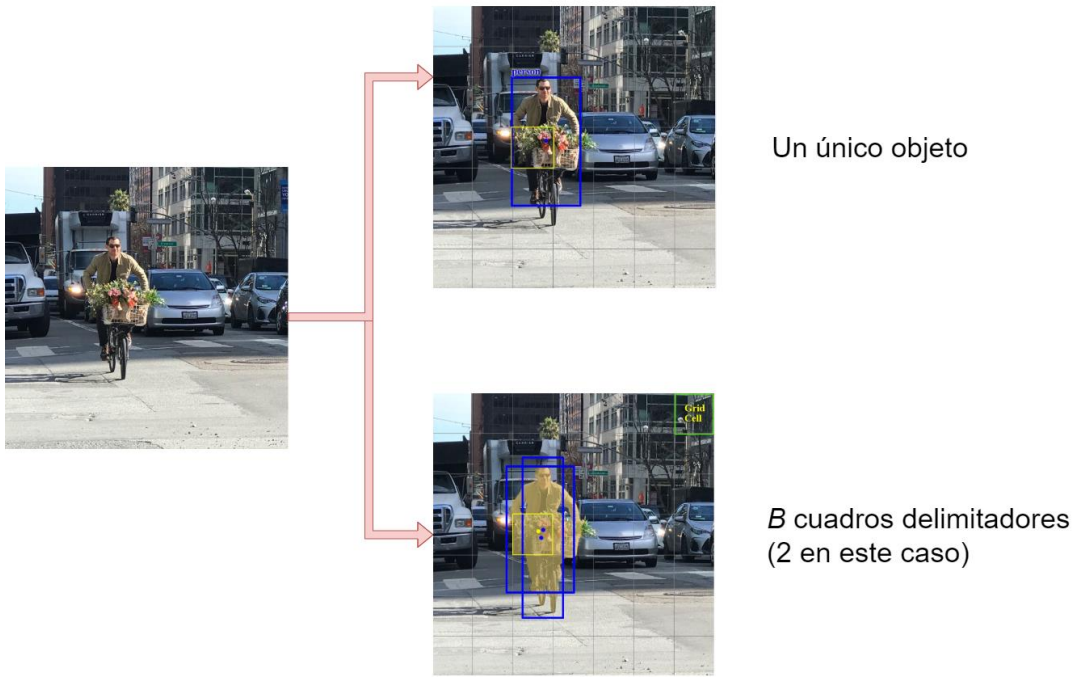


Figura 2-27. Detección y localización por separado en YOLO

Con el fin de detectar un objeto en cada segmento, YOLO genera una *probabilidad de clase condicional* (*conditional class probability*) por cada clase que podamos detectar; es decir, genera ***C*** probabilidades. Sumado a esto, cada cuadro delimitador está compuesto de cinco componentes: comienzo del recuadro en el eje X, comienzo en el eje Y, anchura, altura y *puntuación de confianza de recuadro* (*box confidence score*). Los cuatro primeros, designados generalmente como (***x***, ***y***, ***w***, ***h***) dibujan el recuadro sobre la imagen, usando valores

relativos al tamaño de ésta (por lo que siempre estarán entre 0 y 1); mientras que la puntuación de confianza determina con cuánta probabilidad hay un objeto dentro de éste (*objectness*) y cuán preciso es el recuadro. Por lo tanto, cada segmento de la imagen original tendrá asociados un total de (B*5 + C) valores; o lo que es lo mismo, el volumen de salida de la red de YOLO tendrá una forma de SxSx(B*5+C). Finalmente, por cada clase y cuadro delimitador, se computará la *puntuación de confianza de clase*, mediante que le se mide la probabilidad de que cierto objeto aparezca en cierta localización.

Para el caso concreto de YOLO con el dataset Pascal VOC, se usó una red configurada tal que:

- S = 7; es decir, una cuadrícula de 7x7 segmentos.
- C = 20; representando las 20 clases (tipos diferentes de objetos reconocibles) que dicho dataset recoge.
- B = 2; dos cuadros delimitadores por segmento.

Lo que resulta un volumen de salida final de 7x7x30, que se representa en la Figura 2-28 junto con el resto de la red de YOLO.

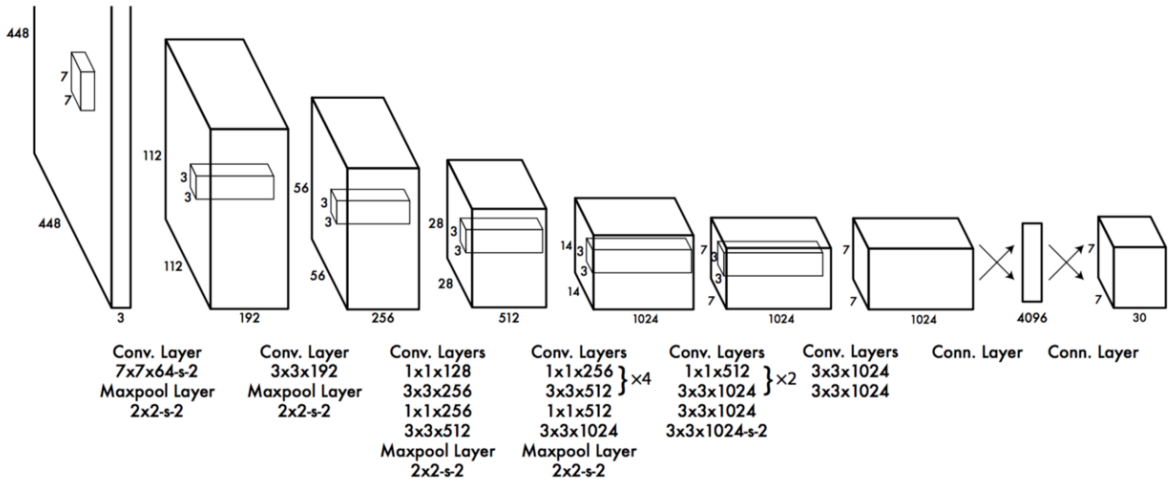


Figura 2-28. Red de YOLO

YOLO utiliza 24 capas convolucionales, con capas de agrupación tipo máximo, que finalizan en dos redes completamente conectadas de las que se obtiene el volumen de salida. Otros modelos de YOLO ajustan el número de capas convolucionales, en función de las características del dispositivo en el que se ejecutará o de las necesidades de la red dado el tipo de imágenes en el que se quiere entrenar. Por ejemplo, TinyYOLO es una versión más ligera, adecuada a dispositivos menos potentes, y por eso ha sido la elección para esta memoria.

Durante el entrenamiento, YOLO calcula sus pérdidas dividiendo la función de pérdidas en tres grupos, cada uno aplicado a cada celda:

- ***Pérdidas en clasificación:*** si un objeto ha sido detectado, calcula el error en las probabilidades de clase determinadas para cada clase.
- ***Pérdidas en localización:*** calcula los errores en los recuadros predichos respecto a los reales (recordemos que, en el set de entrenamiento, las imágenes que utilicemos tendrán que indicar la localización de los objetos y sus recuadros). Para ello, tan sólo tomaremos *un único* cuadro delimitador por cada objeto detectado. Para determinar cuál, se utilizará la *itersección sobre unión*, o *IoU* (del inglés, *Intersection over Union*), una operación que permite medir la superposición de dos selecciones, indicando de cierta manera cuán similares son. La Figura 2-29 lo representa más claramente. En este caso, aquel recuadro con mayor IoU, será el *representante* de dicho objeto, y el que se utilizará para entrenar la red.
- ***Pérdidas de confianza:*** miden el error del *objectness*, es decir, la probabilidad de que haya un objeto en el recuadro delimitador encontrado. Debido a que la gran mayoría de las veces los recuadros no encontrarán objetos dentro, se contabiliza de forma controlada, para evitar que la red se *sobreentrene*

identificando fondos de imágenes sin contenido útil.

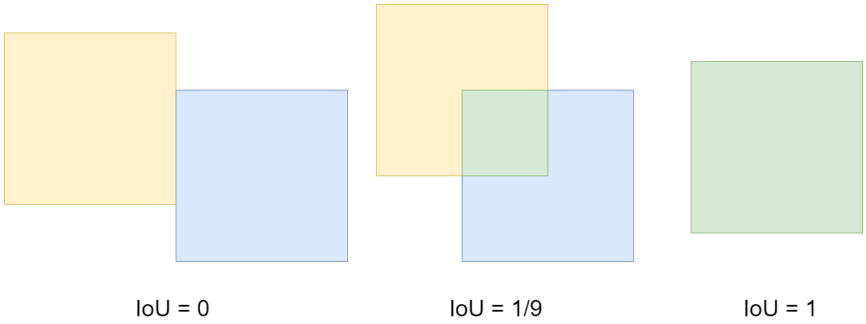


Figura 2-29. Representación gráfica del IoU

Por otro lado, durante la inferencia, debemos tener en cuenta que, como adelantábamos al comienzo, dos o varios segmentos contiguos pueden llegar a detectar el mismo objeto, incluso dentro de recuadros delimitadores similares (con un alto IoU entre ellos, por ejemplo). Para corregir este comportamiento, YOLO aplica una *supresión no-máxima* (*non-maximal supression*) o *NMS*, un algoritmo que discrimina entre los diferentes candidatos, y mantiene tan sólo aquellos con mayor índice de confianza pero que a la vez tengan un bajo IoU entre ellos, para evitar que objetos distintos sean identificados como uno mismo. Esto puede llegar a aumentar el mAP en un 2 o 3%.

Figura 2-30. Efecto de la supresión no-máxima

Un ejemplo de la salida final de una inferencia en YOLO puede verse al comienzo de esta memoria, en la Figura 1-1.

Las ventajas principales de YOLO, algunas de las cuales ya se han mencionado, son las siguientes:

- La propuesta de una única red neuronal y la división de tareas lo hace esencialmente rápido, lo que a su vez lo vuelve válido para uso en detección en tiempo real.
- Fácil de entrenar, derivado del uso de una sola red neuronal también.
- Clasificación y detección en el contexto total de la imagen, sin división en partes como otros métodos de reconocimiento, lo que proporciona menos fallos a la hora de identificar objetos. Hay que recordar que, pese a que YOLO sí utiliza una segmentación de la imagen, cada segmento puede observar el conjunto completo de ésta, no solo su recuadro predefinido.
- Al forzarse a encontrar un objeto en cada segmento de la imagen, admite una diversidad espacial que mejora la calidad de las predicciones.

2.4.2 YOLOv2

YOLOv2 nace para competir con los sistemas *Single-Shot* (disparo único), o SSD, que a cierto nivel presentan una precisión mayor en el procesamiento en tiempo real de imágenes. Así, el objetivo de esta versión, principalmente, fue mejorar en este aspecto, para lo que se aplicaron:

- **Capas de normalización por lotes** (*batch normalization*) en las capas convolucionales. Su función es normalizar los valores de activación de las capas anteriores, manteniéndolos así en un rango concreto que permite agilizar el aprendizaje [40].
- **Clasificación de alta resolución**. Durante el entrenamiento, YOLOv1 entrena dos veces los sets de imágenes, una para la clasificación de objetos, y otra para la detección y localización de éstos. En YOLOv2, el entrenamiento de clasificación se ejecuta dos veces (más otro entrenamiento para detección), una con las imágenes escaladas a 224x224, y la otra a 448x448; lo que incrementa la precisión, aunque puede ralentizar el aprendizaje.
- **Cuadros delimitadores previos** (*prior bounding boxes*) o *cajas ancla*. El concepto es sencillo: son cuadros delimitadores *predefinidos*, con una forma que suele ser común entre varios objetos muy diferentes entre ellos. Por ejemplo, las personas, cuando están de pie, suelen tener un cuadro delimitador de forma rectangular alargada con una relación de aspecto de 0'41. Si analizamos varios de estos cuadros y vemos patrones que se repiten, podemos usarlos para que la red parta de ellos, en vez de tener que crearlos desde cero sin ninguna referencia. Luego, la propia red calculara las diferencias necesarias *respecto* a los cuadros previos, aligerando y estabilizando el entrenamiento, sobre todo en los primeros momentos.

El diseño de la red también varía ligeramente, ya que se libera de la capa completamente conectada y la sustituye por varias capas convolucionales extra que convierten un volumen de 7x7x1024 en el volumen final deseado, que sigue manteniendo la forma que justificábamos antes: *S*x*S*x*(B*2+C)*.

La diferencia final con respecto a YOLOv1, medido en el dataset de Pascal VOC 2007 (recordemos, 20 clases, 9963 imágenes diferentes) es bastante notable, siendo la mAP (recordemos, ratio de acierto medio) de éste de 63'4%, por los 78'6% de YOLOv2.

En la siguiente figura se compara también con otros mecanismos:

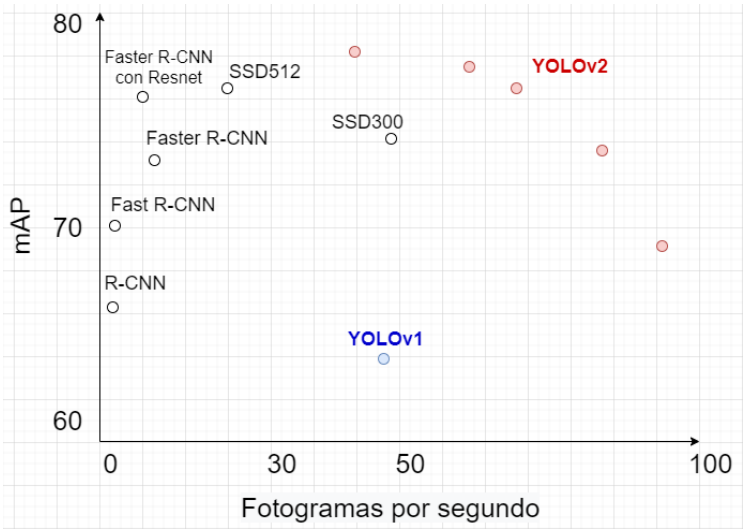


Figura 2-31. Comparativa YOLOv2 para Pascal VOC 2007

Los fotogramas por segundo, también llamados *FPS* (de sus siglas en inglés, *Frames Per Second*) equivalen al número de imágenes que se infieren cada segundo, por lo que cuanto mayor sea dicho valor, mejor será el sistema.

2.4.3 YOLOv3

YOLOv2 consiguió convertirse en uno de los algoritmos más rápidos y precisos cuando apareció por primera vez, pero con los años acabó perdiendo la batalla de la precisión contra otros algoritmos como RetinaNet o SSD, que también hemos mencionado en varias ocasiones. La causa la hemos adelantado ya en párrafos anteriores: YOLO, por su funcionamiento interno, tiene dificultades para detectar objetos pequeños.

Con el fin de mejorar la precisión, YOLOv3 sacrifica parte de su hasta ahora envidiable velocidad de inferencia, pero que supone un aumento de cerca del 13'5% de precisión (mAP). Para ello, las siguientes mejoras fueron introducidas:

- **Nueva arquitectura de red neuronal:** Donde YOLOv2 utilizaba una red de 30 capas (19 para clasificación, 11 para detección), YOLOv3 presenta un nuevo sistema de 106 capas convolucionales (53 para clasificación, 53 para detección). Ésta es la razón de que ahora sea más lento; y también la de que sea más preciso. En la Figura 2-32 se puede ver un esquema con la nueva arquitectura.
- **Detección en tres escalas:** Como se puede ver representado en la figura que mencionábamos antes, ahora YOLO lleva a cabo tres detecciones sobre la misma imagen, en vez de solo una, en tres escalas diferentes. De esta manera, mejora su capacidad de detectar objetos pequeños, toda vez que aquellos más grandes habrán sido detectados en las primeras ocasiones, y pueden ser ignorados posteriormente.
- **Mayor volumen de salida:** Donde las versiones anteriores tenían una salida de tamaño SxSx(B*5+C), YOLOv3 presenta un tensor de SxSx(B*(5+C)); es decir, ahora, cada cuadro delimitador presenta sus propias probabilidades por clase, en vez de ser independientes. Se aumentan también el número de cuadros, de 2 a 5, y no podemos olvidar que, además, ahora se realizan tres predicciones a diferentes escalas sobre la misma imagen; lo que finalmente resulta en más detecciones y, por lo tanto, más probabilidad de acertar.
- **Uso de regresión logarítmica para etiquetado de objetos:** Hasta ahora, YOLO había estado utilizando el método *softmax* para determinar a qué clase pertenecía un objeto cuando varias opciones se presentaban. Esto se modifica ya que la suposición de que una clase tan sólo puede pertenecer a un objeto falla a medida que los *dataset* se vuelven más complejos y poseen más clases. YOLOv3 admite múltiples etiquetas para un mismo objeto, y utiliza una función de regresión logarítmica para determinarlos.

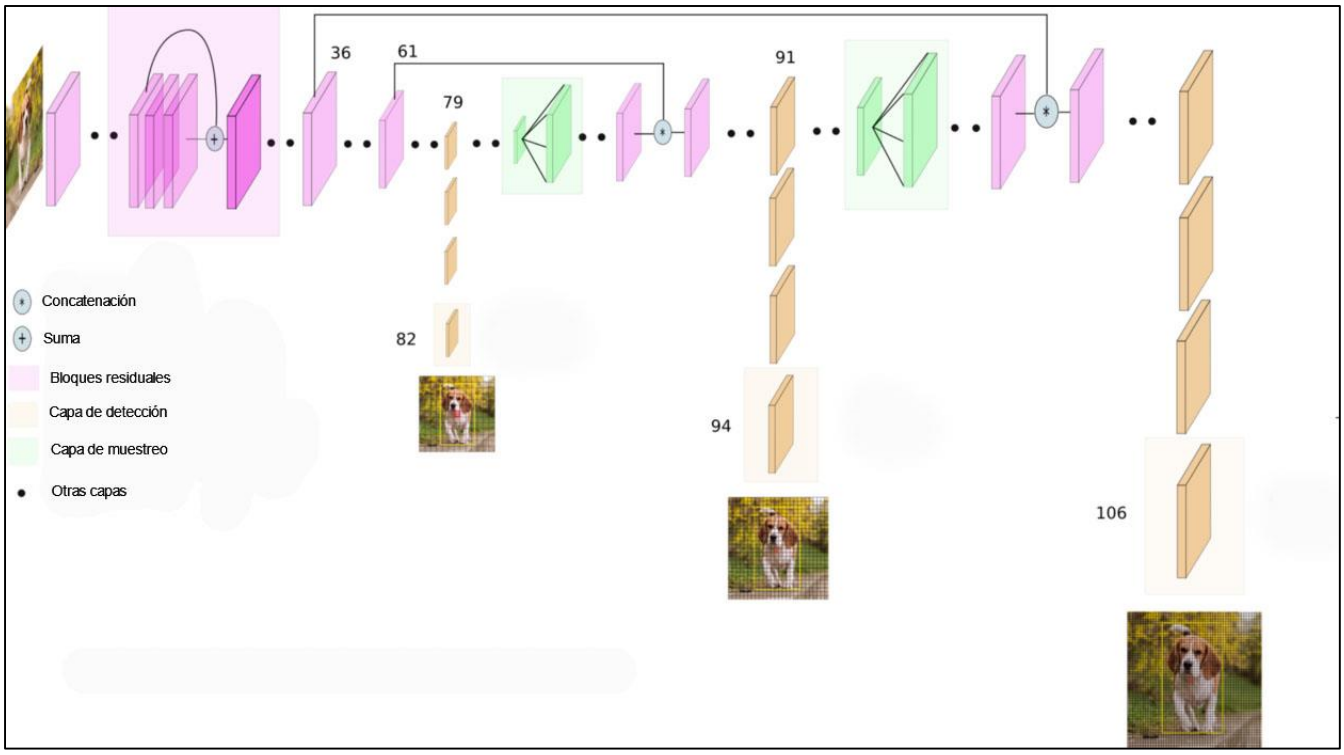


Figura 2-32. Arquitectura de YOLOv3

Estos cambios se traducen en una mejora notable de las prestaciones de YOLOv3, como podíamos ver en la Figura 2-26, donde llega a superar en velocidad e igualar en precisión a otros algoritmos del estado del arte, como RetinaNet. Es por ello que YOLOv3 se presenta como un mecanismo excelente para llevar a cabo proyectos complejos que requieran visión artificial. En los próximos capítulos y párrafos lo probaremos, entrando en detalle sobre la parte técnica del algoritmo, y midiendo sus prestaciones en una aplicación real.

2.5 Hardware

Una vez explicado y desarrollado todo el fundamento teórico relacionado con la detección de objetos y YOLO en concreto, queda por hacer mención al hardware en el que probaremos el algoritmo. En esta sección nos centraremos en el aspecto teórico y en la justificación de la elección de Raspberry Pi como placa computadora; mientras que en el próximo capítulo se entrará en el aspecto técnico tanto del dispositivo como de YOLO, y se desarrollará en detalle el la instalación y puesta en marcha.

Se comenzará con un breve apartado dedicado a explicar la diferencia entre el uso de CPU y GPU para la ejecución de sistemas de inteligencia artificial, que será necesaria para entender la decisión final y parte de la instalación de YOLOv3. A continuación, se procederá a una comparativa entre diferentes dispositivos populares con características similares, y a la elección definitiva de Raspberry.

2.5.1 CPU vs GPU

Como mencionábamos en la introducción de esta memoria, los sistemas de inteligencia artificial pueden llegar a ser costosos computacionalmente y requieren de una gran cantidad de recursos para funcionar, sobre todo en campos como la visión artificial, que utilizan datos y redes neuronales extensas y complejas; aparte de necesitar largas sesiones de entrenamiento para obtener un resultado de calidad, proceso que puede llegar a ser muy lento si la máquina en la que se ejecutan no está preparada.

En una computadora estándar, la CPU es, generalmente, la encargada de llevar a cabo las operaciones pertinentes para cualquier programa que se ejecute en ésta. Está preparada para realizar operaciones grandes y complejas en muy poco tiempo, pero está limitada en la cantidad de operaciones diferentes que puede realizar a la vez. Esto puede ser un problema en inteligencia artificial, ya que la mayoría de modelos requieren realizar operaciones sencillas, pero en mucha cantidad, por lo que la *paralelización* de éstas (la capacidad de realizarlas a la vez) es fundamental si queremos agilizar el proceso. Es por ello que, en los últimos años, se ha popularizado el uso de las GPUs [41].

La GPU (*Graphics Processor Unit*, unidad de procesamiento de gráficos) es un procesador separado de la CPU que está especializado en la *renderización* de imágenes y vídeos, que no es sino la representación gráfico de éstos y su manejo y manipulación. Tiene especial importancia en programas fundamentalmente gráficos, como los visores y editores de imágenes y vídeo, las herramientas de diseño 3D, o los videojuegos [42]. Sin embargo, debido a su modo de funcionamiento interno, también puede ser aplicado a inteligencia artificial, y con muy buenos resultados [41]; ya que donde las CPUs se especializaban en operaciones largas y complejas, las GPUs se especializan en operaciones sencillas, pero en gran cantidad, y con una alta paralelización. Es por ello que, actualmente, la mayoría de sistemas de inteligencia artificial están preparados para trabajar sobre GPU, hasta el punto incluso de existir plataformas dedicadas a la explotación de los recursos de éstas (como Nvidia CUDA [43]).

Pese a su importancia, en este proyecto no utilizaremos la GPU para hacer funcionar nuestro sistema de visión artificial, ya que Raspberry Pi 3 no tiene una especialmente potente, mientras que su CPU sí lo es, como veremos a continuación. La razón por la que aun así se ha escrito esta breve sección es que, como se verá más adelante, se utilizarán ciertas herramientas para optimizar el funcionamiento de YOLO en CPU; además de que la GPU es una característica importante para la comparativa entre placas computadores que se realizará en la siguiente sección.

2.5.2 Placas computadoras

Como comentábamos en la introducción de esta memoria, el objetivo de este proyecto no es solo probar el funcionamiento de un sistema de detección de objetos como YOLOv3, sino además hacerlo en un dispositivo no originalmente diseñado para la ejecución de inteligencia artificial, con el fin de comprobar hasta qué punto es viable.

La decisión de utilizar una placa computadora para el proyecto viene de la propia definición de éstos: son sistemas computacionales sencillos, económicos, y multifunción. Además, en los últimos años se están convirtiendo en alternativas fiables para una gran cantidad de tareas cotidianas [44], y se estima que el 98% de los microprocesadores fabricados se destinan a sistemas embebidos [45]. Analizar y probar el funcionamiento de sistemas de inteligencia artificial sobre este tipo de dispositivos podría servir para acercar estas nuevas tecnologías al usuario de a pie.

En concreto, para esta memoria introduciremos tres placas diferentes, dos de ellas siendo de las más populares en el mercado actual (Arduino y Raspberry Pi), y la otra siendo una placa dedicada específicamente a la inteligencia artificial (NVIDIA Jetson). Explicaremos brevemente cada una de ellas y finalmente expondremos una tabla comparativa a través de la cual justificaremos la decisión tomada.

- **Arduino:** Arduino se presenta como una plataforma de electrónica de código abierto, cuyo lema principal es el diseño y creación de hardware y software fácil de utilizar. Las placas Arduino utilizan microcontroladores y varios periféricos que le permiten servir para muchos y variados propósitos, además de ser fácilmente programables. En concreto, la placa Arduino Nano 33 BLE Sense es recomendado por la propia página oficial para su dedicación a inteligencia artificial.
- **Raspberry Pi:** Las Raspberry Pi son una serie de placas desarrolladas originalmente para su dedicación en la enseñanza de las ciencias de computación. Son placas gobernadas por un microprocesador ligero pero suficientemente potente y que carece de periféricos por defecto. Debido a su económico precio, en los últimos años han sido utilizadas como base para muchos proyectos dedicados a la robótica y la automatización. Actualmente existen varias versiones, siendo la última la 4, lanzada a comienzos del año 2020. Para este proyecto, sin embargo, consideraremos la Raspberry Pi 3, ya que está más extendida.
- **NVIDIA Jetson:** Las placas NVIDA Jetson fueron comercializadas por primera vez en 2014, y su premisa es ser ordenadores dedicados a inteligencia artificial en formato de placa computadora, existiendo varios modelos con diferentes precios y prestaciones. Posee características técnicas idóneas para este fin; pero no están tan extendidas como las anteriormente mencionadas.

En la Tabla 2-4 se realiza una comparación de las diferentes características técnicas que nos deberían ser relevantes a la hora de decidir nuestra elección. Se han escogido, para cada placa computadora, su modelo más adecuado en relación rendimiento-precio para el proyecto que nos incumbe.

Tabla 2-4. Comparativa placas computadoras para IA

	CPU	**GPU**	**Dedicado a IA**	**Software IA**	**Precio**
Arduino Nano 33 BLE Sense	ARM Cortex-M4 32bit (64MHz)	No tiene	Sí	TinyML	27€
Raspberry Pi 3 B	4xARM Cortex-A53 64bit (1.2GHz)	Broadcom VideoCore IV	No	Cualquiera	40€
NVIDIA Jetson Nano	4xARM Cortex-A57 64bit (1.43GHz)	128-core Nvidia Maxwell	Sí	JetPack SDK	100€

Como podemos observar, tanto los modelos de Arduino como Jetson son placas diseñadas con el fin de trabajar

con inteligencia artificial, y, en consecuencia, estarán más preparadas que la Raspberry Pi. Por otro lado, en términos de CPU, la diferencia entre Raspberry Pi y Nvidia no es tan amplia, y es que ambos microprocesadores son especialmente potentes, siendo el A57 de la Jetson tan solo ligeramente superior, como se puede ver en la comparativa de la referencia [46]. En cuanto a la GPU, Jetson es la única placa que posee una que pueda ser utilizada en inteligencia artificial, ya que el de Raspberry Pi es muy ligero y sólo se dedica para operaciones gráficas muy sencillas. Arduino, por su lado, carece de GPU y tiene un microcontrolador potente pero que no compite con los otros dos; compensando estas carencias con un software adaptado y dedicado como es TinyML [47].

En conclusión, cualquiera de las anteriores placas sería una buena opción para llevar a cabo el proyecto, y la decisión final de usar Raspberry Pi 3 no necesariamente mejora a ninguna de las otras dos. Una de las ventajas que sí ofrece Raspberry Pi sobre Arduino y Jetson es que, al no tener un software específico para trabajar con inteligencia artificial, nos permite utilizar la implementación original de YOLO, sin necesidad de adaptarla; pero, por lo demás, cualquier podría ser una buena candidata.

En concreto para este proyecto, se utilizará una Raspberry Pi 3 B con sistema operativo Raspberry Pi OS, configurado de forma estándar y cuya instalación puede seguirse en el Anexo A.

3 INSTALACIÓN

En los siguientes párrafos se explicará y describirá el proceso de instalación y puesta en marcha de YOLOv3 en una Raspberry Pi 3. Se detallarán las partes que intervienen, el funcionamiento de éstas y su configuración.

Se comenzará explicando YOLOv3 desde su punto de vista técnico, continuación de la explicación teórica del capítulo anterior, haciendo hincapié tanto en su visión general como en aquélla adaptada para sistemas de recursos limitados como el que vamos a utilizar. Se detallarán sus componentes y se justificarán sus necesidades y se explicará cómo llevar a cabo la instalación completa, partiendo de una Raspberry Pi configurada como se detalla en el Anexo A.

3.1 Presentación técnica y componentes

Hasta ahora, hemos hablado de YOLOv3 desde un punto de vista puramente teórico, sin entrar en detalles de cómo funciona internamente o cómo es su arquitectura a bajo nivel. Esta sección la dedicaremos para ese fin, describiendo y detallando los componentes que necesitamos para hacer funcionar YOLO en Raspberry Pi, y justificando todas las consideraciones y decisiones que se han tomado para que sea posible.

3.1.1 Darknet

En primer lugar, hay que puntualizar que la familia de algoritmos YOLO define *modelos* de redes neuronales, pero no los *implementan* directamente. Las arquitecturas propuestas por las diferentes versiones de YOLO deben implementarse en una red neuronal que las interprete y ejecute, llevando a cabo las operaciones necesarias y los procesos de entrenamiento e inferencia. En líneas generales, cualquier entorno de trabajo con redes neuronales valdría (por ejemplo, algunos muy populares como Google Tensorflow o PyTorch), pero la versión original de YOLO está codificada sobre un entorno específico: Darknet.

Darknet es un *framework* (entorno de trabajo dedicado) de redes neuronales de código abierto, programado en C y CUDA, y que se centra en la velocidad y simplicidad de la instalación. Además, permite trabajar sobre CPU y GPU indistintamente, que es un requisito fundamental para funcionar en Raspberry Pi, como veíamos en las secciones anteriores. Aparte de YOLO, Darknet también permite realizar clasificación de imágenes (del dataset de ImageNet), generación de texto mediante redes neuronales recurrentes, e incluso jugar al Go, juego de mesa de origen chino que por su complejidad suele usarse para demostrar la capacidad potencial de la inteligencia artificial. Entender cómo funciona Darknet, por lo tanto, es fundamental para poder ejecutar YOLO. Más adelante veremos cómo se realiza la instalación y puesta en marcha, pero ahora nos centraremos en modo de funcionamiento.

Para comenzar, Darknet requiere dos elementos básicos: un archivo de configuración (*cfg-file*) y un archivo de pesos (*weights-file*). Estos dos archivos son los encargados de definir y describir la arquitectura de la red neuronal que se va a utilizar, y proporcionar los pesos que alimentan los parámetros de ésta, respectivamente. El archivo de configuración debe ser modificado y proporcionado por el usuario; y es donde se localizarán los hiperparámetros que explicábamos en capítulos anteriores, y también será donde indiquen, de forma ordenada, las capas que forman la red, su tipo, sus volúmenes de entrada y salida, y demás hiperparámetros específicos de cada una de ellas, como podría ser el tamaño del filtro en una capa convolucional. El archivo de pesos, por su lado, no es editable, y se genera automáticamente durante la fase de entrenamiento de la red, y posteriormente se usa para la inferencia. Darknet además proporciona una amplia variedad de archivos de pesos pre-entrenados en diversos datasets y para diversas configuraciones de red, en caso de que tan sólo se quiera probar la inferencia o no se requiera un entrenamiento específico.

Además de ser un framework, Darknet también define varias configuraciones de red. En el caso de YOLOv3,

por ejemplo, la arquitectura que describíamos teóricamente en la sección 2.4.3 se basa en la de Darknet-53, a la que YOLOv3 añade su propio método de detección y clasificación. Por otro lado, para este proyecto utilizaremos *TinyYOLOv3*, una versión ligera del algoritmo que reduce el número de capas y simplifica las operaciones, características imprescindibles para poder funcionar en un dispositivo como Raspberry Pi 3. Esta versión en concreto se basa en Darknet-19, que se representa en la siguiente figura [48], donde las columnas indican, en orden, el número de la capa, su tipo, el número de filtros en el caso de las capas convolucionales, el tamaño de los filtros y su paso, y finalmente el tamaño de entrada y salida de la capa.

Layer	Type	Filters	Size/Stride	Input	Output
0	Convolutional	16	3 × 3/1	416 × 416 × 3	416 × 416 × 16
1	Maxpool		2 × 2/2	416 × 416 × 16	208 × 208 × 16
2	Convolutional	32	3 × 3/1	208 × 208 × 16	208 × 208 × 32
3	Maxpool		2 × 2/2	208 × 208 × 32	104 × 104 × 32
4	Convolutional	64	3 × 3/1	104 × 104 × 32	104 × 104 × 64
5	Maxpool		2 × 2/2	104 × 104 × 64	52 × 52 × 64
6	Convolutional	128	3 × 3/1	52 × 52 × 64	52 × 52 × 128
7	Maxpool		2 × 2/2	52 × 52 × 128	26 × 26 × 128
8	Convolutional	256	3 × 3/1	26 × 26 × 128	26 × 26 × 256
9	Maxpool		2 × 2/2	26 × 26 × 256	13 × 13 × 256
10	Convolutional	512	3 × 3/1	13 × 13 × 256	13 × 13 × 512
11	Maxpool		2 × 2/1	13 × 13 × 512	13 × 13 × 512
12	Convolutional	1024	3 × 3/1	13 × 13 × 512	13 × 13 × 1024
13	Convolutional	256	1 × 1/1	13 × 13 × 1024	13 × 13 × 256
14	Convolutional	512	3 × 3/1	13 × 13 × 256	13 × 13 × 512
15	Convolutional	255	1 × 1/1	13 × 13 × 512	13 × 13 × 255
16	YOLO				
17	**Route 13**				
18	Convolutional	128	1 × 1/1	13 × 13 × 256	13 × 13 × 128
19	Up-sampling		2 × 2/1	13 × 13 × 128	26 × 26 × 128
20	**Route 19 8**				
21	Convolutional	256	3 × 3/1	13 × 13 × 384	13 × 13 × 256
22	Convolutional	255	1 × 1/1	13 × 13 × 256	13 × 13 × 256
23	YOLO				

Figura 3-1. Arquitectura de TinyYOLOv3

En el Anexo B, además, se incluye una copia de esta arquitectura en el formato de archivo de configuración utilizado por Darknet. Algunas de las secciones más relevantes de estos archivos y sus hiperparámetros se presentan a continuación en formato tabla, indicando cuáles son los valores por defecto o más comunes en TinyYOLOv3 y explicando su función e importancia:

Tabla 3-1. Sección de red de los archivos de configuración de Darknet

[net]	Configuración base de la arquitectura. Define los hiperparámetros generales, como el tamaño de las imágenes, los lotes y mini-lotes, etcétera.	
Hiperparámetros	***Definición***	***TinyYOLOv3***
batch	Número de muestras que se procesarán en un lote. Este hiperparámetro será útil en entrenamiento, mientras que en inferencia no tendrá uso y se quedará a 1.	1 para inferencia 64 recomendado para entrenamiento
subdivisions	Número de subdivisiones a hacer en cada lote, con lo que logramos mini-lotes, tal que *imágenes por mini-lotes = tamaño de lotes / subdivisiones*. Durante la inferencia, al igual que el número de lotes, no tiene significado y se fija a 1.	1 para inferencia 16 recomendado para entrenamiento
max_batches	Número de iteraciones tras los cuales se dará por finalizado el entrenamiento.	500200
width	Volumen de entrada de las imágenes. Si éstas no corresponden, serán ajustadas automáticamente; aunque no conviene utilizar imágenes que difieran mucho del tamaño estipulado, para evitar que pierdan detalle al aumentarse o minimizarse.	416
height		416
channels		3
momentum	Optimizador, se usa para determinar cuánto afecta el historial de cambios de los pesos a las próximas modificaciones de éstos.	0.9
decay	Optimizador que lleva a cabo una actualización más débil de los pesos de las características típicas de las imágenes.	0.0005
saturation	Hiperparámetros que modifican aleatoriamente características de las imágenes de entrenamiento, como la saturación del color o el brillo. Introducir esta aleatoriedad permite mejorar el entrenamiento, al ser las imágenes parcialmente diferentes cada vez que alimentan la red.	1.5
exposure		1.5
hue		0.1
learning_rate	Ratio de aprendizaje inicial.	0.001
burn_in	Modificador del ratio de aprendizaje para las primeras imágenes, lo que aumenta la velocidad del entrenamiento en sus primeros compases.	1000
policy	Política para cambiar el ratio de aprendizaje a medida que avanza el entrenamiento. Puede ser por pasos, exponencial, aleatorio…	steps
steps	En caso de que `policy` sea `steps`, sirve para indicar cada cuántas iteraciones se modifica el ratio de aprendizaje.	400000,450000
scales	En caso de que `policy` sea `steps`, sirve para indicar por cuánto se multiplica el ratio de aprendizaje cuando se cumple cada paso marcado por `steps`.	.1,.1

Una gran parte de estos parámetros de red están orientados a definir cómo se llevará a cabo el entrenamiento, y

se ignorarán durante el proceso de inferencia (que en YOLO es comúnmente denominado *detección*). En líneas generales, sus valores por defecto no requieren modificación, exceptuando el tamaño de imagen de entrada, que debe adecuarse al de aquellas con las que se piense entrenar; el número de lotes y sus subdivisiones, que suele ajustarse para no sobrecargar la memoria del dispositivo que se esté utilizando; y el total de lotes antes de dar por finalizado el entrenamiento, que dependerá en gran medida del número de lotes y mini-lotes escogidos y de la cantidad de imágenes del dataset de entrenamiento.

Cabe destacar, además, que entre estos hiperparámetros no se mencionan las épocas, que, si recordamos, enumera cuántas veces todas las imágenes del dataset de entrenamiento han tenido la oportunidad de actualizar los parámetros de la red, y que es utilizado en muchas ocasiones para determinar cuándo finalizar dicho entrenamiento. Darknet no permite la configuración de este hiperparámetro, aunque se puede estimar siguiendo la siguiente fórmula:

$$max\_epochs = \frac{max\_batches \times batches}{total\_muestras} \tag{3-1}$$

Continuando con las secciones del archivo de configuración, a continuación se muestran aquellas designadas para configurar las diferentes capas que conforman la red neuronal:

Tabla 3-2. Configuración para las capas de convolución

`[convolutional]`	Descripción de los hiperparámetros de las capas de convolución	
Hiperparámetros	***Definición***	***TinyYOLOv3***
`batch_normalize`	Aplica normalización por lotes, que, si recordamos, normalizaba todos los valores obtenidos tras las operaciones en la capa para mantenerlos en un rango conocido. Vale 1 si se aplica, 0 si no.	1
`filters`	Número de filtros diferentes a aplicar.	*Varía*
`size`	Tamaño de los filtros. Suele ser entre 1, 2 o 3.	*Varía*
`stride`	Tamaño del paso. Generalmente vale 1, aunque en las primeras capas se aplican algunos de 2 para agilizar.	*Varía*
`pad`	Relleno con ceros. En este caso, este parámetro valdrá 1 si se quiere usar como relleno `size/2` de forma automática; si se quisiera definir manualmente, habría que usar `padding`.	1
`activation`	Especifica la función de activación a utilizar. Para nuestro caso, TinyYOLOv3 utilizará generalmente `leaky`, que es una versión modificada de ReLU que no desactiva las neuronas cuando su valor es negativo, sino que les otobrga un valor negativo muy cercano a 0. En algunas capas finales, también se una función lineal (`linear`), que realmente es como si no hubiera función de activación alguna, y sólo se usa antes de las capas de detección YOLO, que veremos más adelante.	`leaky` `linear`

Tabla 3-3. Configuración de las capas de *max-pooling*

[maxpool]	Llevan a cabo un filtro de agrupación por máximo (*max-pooling*)	
Hiperparámetros	***Definición***	***TinyYOLOv3***
`size`	Tamaño del filtro. Suele estar fijado a 2x2.	2
`stride`	Paso del filtro. Suele valer 1 o 2.	*Varía*

Tabla 3-4. Configuración de las capas de detección de YOLO

[yolo]	Capa de detección de YOLO	
Hiperparámetros	***Definición***	***TinyYOLOv3***
`mask`	Índices de las cajas anclas que se van a utilizar. Recordemos que estas cajas eran cuadros delimitadores predefinidos, utilizados para partir de alguna base a la hora de localizar objetos.	*Varía*
`anchors`	Tamaños iniciales de los cuadros delimitadores.	*Varía*
`num`	Número total de cuadros delimitadores a proponer.	6
`classes`	Total de clases que de objetos que pueden ser detectadas. Este valor dependerá del dataset de entrenamiento que hayamos utilizado (inferencia) o vayamos a utilizar (entrenamiento).	*Depende*
`jitter`	Recorta y reajusta el tamaño de la imagen sin cambiar su ratio de aspecto. Mejora la calidad de los cuadros delimitadores.	0.3
`ignore_thresh`	Filtra las detecciones duplicadas que pueden hallarse, y que serán posteriormente fusionadas mediante NMS.	0.7
`truth_thresh`	Filtra las detecciones duplicadas que pueden hallarse y que serán posteriormente fusionadas mediante NMS. Acompaña a `ignore_thresh` pero desde otra perspectiva.	1
`random`	Aleatoriamente modifica el tamaño de las imágenes a la entrada de la red cada diez iteraciones. Permite mejorar la variabilidad de las detecciones realizadas y así mejorar la confianza en éstas, pero supone un gran coste computacional.	1

Si nos fijamos, todos estos tipos de capas aparecen representados en el esquema de arquitectura de TinyYOLOv3 que mostrábamos en la Figura 3-1. Tan sólo quedarían por mencionar las capas `[route]` y `[upsample]`, que no tienen una configuración tan exhaustiva como para desglosarlas en una tabla, ni las hemos tratado en los fundamentos teóricos. La primera es la *capa de ruta*, y sirve para concatenar las salidas de diferentes capas en una sola; y la segunda es la *capa de sobremuestreo*, que sirve para aumentar el tamaño de la entrada mediante la duplicación de sus elementos de forma controlada. Ambas capas son específicas de la arquitectura interna de YOLO, y no es necesario entrar en muchos detalles sobre su utilidad o beneficios.

Para finalizar con Darknet –a falta de detallar el proceso de instalación y puesta en marcha-, quedan por discutir un archivo más que también es fundamentales para la ejecución del programa: el archivo de datos (*data file*). Se utiliza para dar información del dataset que se ha usado (inferencia) o se va a usar (entrenamiento), especificando, por ejemplo, el número de clases disponibles y sus nombres. Un ejemplo sencillo podría ser el siguiente:

```
classes = 3
train = train.txt
valid = test.txt
names = obj.names
backup = backup/
```

En este caso, estaríamos indicando que nuestro dataset posee 3 clases; y, además, también señalamos que las muestras para el entrenamiento están listadas en el archivo *train.txt* y las de validación en *test.txt*; que los nombres de las clases están en *obj.names* (que también es un archivo de texto plano); y que los *backups* deben guardarse en la carpeta *backup/*. De los archivos de entrenamiento y validación y de los *backups* hablaremos más adelante, pero el archivo de nombres (aquí llamado *obj.names*) es relevante, ya que es en éste donde listaremos los nombres de todas las clases disponibles en nuestro dataset, y que la red tendrá o bien que aprender, o bien que detectar. Por ejemplo, para este caso, que disponemos de tres clases según el archivo de datos, podríamos tener un archivo de nombres tal que:

```
dog
cat
bird
```

Lo que querría decir que en este dataset buscamos detectar perros, gatos o pájaros. Evidentemente, esto está íntegramente relacionado con las muestras de entrenamiento, ya que para que nuestra red sea capaz de distinguir qué es cada cosa, tendremos que tener un conjunto de ejemplos en los que cada uno esté bien identificado. Más adelante detallaremos cómo se lleva a cabo el proceso de entrenamiento en Darknet y YOLO.

3.1.2 NNPACK

NNPACK es un paquete de aceleración para cálculos en redes neuronales convolucionales, que se centra en proporcionar implementaciones de alto rendimiento para redes que trabajen sobre CPU. Provee mecanismos y primitivas a bajo nivel, escritas en C, que pueden ser utilizadas por frameworks dedicados a *deep learning*, como es en este caso Darknet; además de que permite manejar todos los núcleos de la CPU, mejorando la paralelización de operaciones.

Su utilidad en este proyecto es innegable, ya que vamos a trabajar sobre un dispositivo que depende totalmente de su CPU para poder llevar a cabo las pruebas con YOLO. Esto no quiere decir que NNPACK sea *obligatorio* para poder ejecutar Darknet, ya que el framework de por sí puede configurarse para trabajar exclusivamente con la CPU (o exclusivamente con la GPU); pero mediante NNPACK podremos mejorar sustancialmente su rendimiento. Por otro lado, como se ha comentado, NNPACK tan sólo es un paquete que define una serie de funciones y algoritmos, pero no integra directamente el framework de red neuronal. Es por ello que, para poder usarlo, tenemos que buscar una implementación de Darknet con NNPACK.

Existen varias distribuciones de Darknet-NNPACK de código abierto, con ligeras diferencias entre unas y otras en las que varían la interfaz y el rendimiento. A lo largo de este proyecto se ha trabajado con dos de ellas, a las que se referirá con los nombres de sus creadores: digitalbrain79 (https://github.com/digitalbrain79/darknet-nnpack), y shizukachan (https://github.com/shizukachan/darknet-nnpack); siendo la primera la distribución original, que se basa directamente en el Darknet estándar e integra NNPACK desde cero; mientras que la segunda es una bifurcación de esta primera con ciertas mejoras de rendimiento y organización. Sin embargo, para llevar a cabo las diferentes pruebas de este proyecto, se han tenido que realizar ciertas modificaciones del código fuente de Darknet, además de añadir scripts y códigos auxiliares para automatizar los experimentos, por lo que se ha preparado un repositorio personalizado (https://github.com/angmorpri/darknet-tfg) en GitHub, que toma de base la distribución propuesta por shizukachan (ya que ésta ha demostrado tener mejor rendimiento que la de digitalbrain79, como se verá más adelante). En líneas generales, cada vez que se lleven a cabo tareas sobre Darknet, se especificará en qué distribuciones se han realizado y se comentarán los diferentes resultados (si los hay), pero si no se mencionase, siempre nos referiríamos a la versión personalizada.

3.2 Instalación y puesta en marcha

La instalación de la distribución Darknet-NNPACK utilizada se ha divido en cuatro secciones, dedicadas cada una a los diferentes componentes que forman el proyecto:

1) Requisitos previos
2) NNPACK
3) Darknet
4) TinyYOLOv3 y primera prueba de funcionamiento

Veremos cada uno de estos puntos en detalle a continuación. Cabe destacar que, excepto para el tercer paso, para el resto no es relevante qué distribución de Darknet-NNPACK vayamos a utilizar, bien sea la de digitalbrain79, shizukachan, o la personalizada. En cualquier caso, sí que se recomienda seguir los pasos de instalación aquí explicados o los que vienen en el archivo *README* del repositorio personalizado (en inglés); incluso aunque se usase otra de las distribuciones, ya que éstos están más actualizados y son menos propensos a fallos.

3.2.1 Requisitos y paquetes previos

Una vez puesta en marcha la Rasbperry Pi como se propone en el Anexo A, tendremos que instalar algunos paquetes y librerías extra para poder instalar tanto NNPACK como Darknet. En concreto, estos serán CMake y Ninja, ambos diseñados para construir y compilar programas a partir de ciertas instrucciones. También hará falta instalar Git, para poder clonar los repositorios necesarios desde GitHub.

Se propone utilizar los siguientes comandos para realizar la instalación de ambos:

```
$> sudo apt-get install git cmake
$> git clone git://github.com/ninja-build/ninja.git
$> cd ninja
$>./configure.py --bootstrap
$> sudo cp ninja /usr/sbin/
$> export PATH="${PATH}:~/ninja"
```

Para comprobar que la instalación de ambos paquetes ha sido correcta, podemos llevar a cabo las siguientes acciones:

```
$> cmake --version
cmake version 3.13.4

CMake suite maintained and supported by Kitware (kitware.com/cmake).
$> ninja --version
1.10.0.git
```

3.2.2 Instalación de NNPACK

Bastará con ejecutar los comandos a continuación listados. Cabe destacar que no se utilizará el repositorio original de NNPACK (creado por Maratyszcza (https://github.com/Maratyszcza/NNPACK), sino la versión de shizukachan (https://github.com/shizukachan/NNPACK), que ha dado menos fallos durante las pruebas y parece funcionar mejor con el Darknet del mismo creador.

```
$> cd ~
$> git clone https://github.com/shizukachan/NNPACK
$> cd NNPACK
$> mkdir build && cd build
$> cmake -G Ninja -D BUILD_SHARED_LIBS=ON -DCMAKE_C_FLAGS=-march=armv6k ..
$> ninja
$> sudo ninja install
```

A continuación, hay que crear el archivo `nnpack.conf` en `/etc/ld.so.conf.d/` (crear directorio si no existe) y escribir en éste `/usr/local/lib`. Esto será necesario para que Darknet sepa dónde localizar las

librerías compiladas por NNPACK.

3.2.3 Instalación de Darknet

Para descargar Darknet, bastará con clonar el repositorio de su URL correspondiente. En este caso, utilizaremos el repositorio personalizado, pero podría sustituirse tanto por el de digitalbrain79 como shizukachan.

```
$> cd ~
$> git clone https://github.com/angmorpri/darknet-tfg
```

Luego, entraremos en el directorio que se habrá creado, y que tendrá el mismo nombre del repositorio (`darknet-tfg` si es el personalizado, `darknet-nnpack` si es alguno de los otros). La organización de los archivos dependerá de la distribución descargada, pero por ahora, tan sólo será relevante el archivo `Makefile`, que tiene las instrucciones para llevar a cabo la compilación e instalación de Darknet. En él tendremos que modificar algunas banderas, que son variables que controlan cómo y qué partes se instalan y cuáles no, con el objetivo de prepararlo para trabajar con NNPACK y en CPU.

Las banderas a modificar y sus valores para la distribución personalizada y shizukachan son las siguientes:

```
GPU=0
OPENCV=0
NNPACK=1
NNPACK_FAST=1
ARM_NEON=1
```

En caso de utilizar digitalbrain79, algunas de estas banderas no aparecerán, por lo que bastará con ignorarlas, pero también habrá que rellenar la siguiente:

```
LIBSO=1
```

A continuación, valdrá con ejecutar el siguiente comando, con el que, si no hay fallos (aparecerán muchas líneas de advertencia que podemos ignorar), habremos construido e instalado Darknet correctamente:

```
$> make
```

Además, en caso de haber utilizado el repositorio personalizado, será necesario ejecutar los comandos siguientes, que permiten que algunos de los scripts para pruebas desarrollados puedan funcionar correctamente:

```
$> sudo cp libdarknet.so /usr/local/lib
$> export LD_LIBRARY_PATH="/usr/local/lib:/usr/lib"
```

Podremos comprobar que Darknet ha sido instalado correctamente comprobando si el archivo ejecutable `darknet` ha sido generado.

3.2.4 TinyYOLOv3 y primera prueba de funcionamiento

Finalmente, una vez Darknet-NNPACK está correctamente instalado, tan sólo queda probar que funciona. Para ello, trataremos de llevar a cabo una prueba de inferencia sencilla con TinyYOLOv3. Recordemos que para ejecutar cualquier red neuronal en Darknet será indispensable tener un archivo de configuración que nos diga cómo es la red, un archivo de pesos entrenados, y un archivo de datos con la información del dataset utilizado. En nuestro caso, el archivo de configuración será el que defina la arquitectura de TinyYOLOv3 que mostrábamos en secciones anteriores. Viene incluido en la distribución de Darknet, localizándose en `cfg/yolov3-tiny.cfg`. Por su parte, el archivo de datos estará también descargado por defecto, en `cfg/coco.data`. Este archivo está preparado para trabajar sobre el dataset de COCO (https://cocodataset.org/#home), que es bastante popular y que distingue entre 80 clases de objetos distintos. Finalmente, utilizaremos un archivo de pesos pre-entrenados en dicho dataset, pero que habrá que descargarlo aparte, ya que estos archivos suelen ocupar bastante espacio en el disco y tenerlos todos en el repositorio desde un comienzo ralentizaría mucho su descarga. Podemos para ello ejecutar el siguiente comando:

```
$> wget https://pjreddie.com/media/files/yolov3-tiny.weights
```

Finalmente, sólo nos haría falta un candidato para probar la inferencia. Todos los repositorios traen un pequeño grupo de imágenes para probar. En digitalbrain79 y shizukachan se encuentran en la carpeta `data/`, mientras que en el repositorio personalizado están en `testing/`. Un ejemplo de ejecución podría ser el siguiente:

```
$> ./darknet detect cfg/yolov3-tiny.cfg yolov3-tiny.weights testing/dog.jpg
```

El parámetro `detect` indica que queremos realizar una detección directamente con el archivo de datos de COCO, por eso éste no se incluye en los posteriores parámetros, que indican el resto de archivos necesarios y la imagen. En caso de que quisiéramos utilizar un archivo de datos distinto, utilizaríamos el parámetro `detector`, acompañado directamente del archivo de datos requerido, y manteniendo el resto de parámetros igual.

La ejecución de Darknet debería generar una salida similar a la siguiente:

```
layer     filters    size              input                output
    0 conv     16  3 x 3 / 1   416 x 416 x   3   ->   416 x 416 x  16  0.150 BFLOPs
    1 max          2 x 2 / 2   416 x 416 x  16   ->   208 x 208 x  16
    2 conv     32  3 x 3 / 1   208 x 208 x  16   ->   208 x 208 x  32  0.399 BFLOPs
    3 max          2 x 2 / 2   208 x 208 x  32   ->   104 x 104 x  32
    4 conv     64  3 x 3 / 1   104 x 104 x  32   ->   104 x 104 x  64  0.399 BFLOPs
    5 max          2 x 2 / 2   104 x 104 x  64   ->    52 x  52 x  64
    6 conv    128  3 x 3 / 1    52 x  52 x  64   ->    52 x  52 x 128  0.399 BFLOPs
    7 max          2 x 2 / 2    52 x  52 x 128   ->    26 x  26 x 128
    8 conv    256  3 x 3 / 1    26 x  26 x 128   ->    26 x  26 x 256  0.399 BFLOPs
    9 max          2 x 2 / 2    26 x  26 x 256   ->    13 x  13 x 256
   10 conv    512  3 x 3 / 1    13 x  13 x 256   ->    13 x  13 x 512  0.399 BFLOPs
   11 max          2 x 2 / 1    13 x  13 x 512   ->    13 x  13 x 512
   12 conv   1024  3 x 3 / 1    13 x  13 x 512   ->    13 x  13 x1024  1.595 BFLOPs
   13 conv    256  1 x 1 / 1    13 x  13 x1024   ->    13 x  13 x 256  0.089 BFLOPs
   14 conv    512  3 x 3 / 1    13 x  13 x 256   ->    13 x  13 x 512  0.399 BFLOPs
   15 conv    255  1 x 1 / 1    13 x  13 x 512   ->    13 x  13 x 255  0.044 BFLOPs
   16 detection
   17 route  13
   18 conv    128  1 x 1 / 1    13 x  13 x 256   ->    13 x  13 x 128  0.011 BFLOPs
   19 upsample            2x    13 x  13 x 128   ->    26 x  26 x 128
   20 route  19 8
   21 conv    256  3 x 3 / 1    26 x  26 x 384   ->    26 x  26 x 256  1.196 BFLOPs
   22 conv    255  1 x 1 / 1    26 x  26 x 256   ->    26 x  26 x 255  0.088 BFLOPs
   23 detection
Loading weights from yolov3-tiny.weights... (Version 2) Done!
data/dog.jpg: Predicted in 4.257082 seconds.
5
Box 0 at (x,y)=(0.745051,0.209170) with (w,h)=(0.279108,0.171397)
Box 1 at (x,y)=(0.509042,0.521773) with (w,h)=(0.480992,0.519662)
Box 2 at (x,y)=(0.291488,0.621354) with (w,h)=(0.300040,0.581163)
Box 3 at (x,y)=(0.324342,0.611325) with (w,h)=(0.312303,0.574243)
Box 4 at (x,y)=(0.751867,0.218836) with (w,h)=(0.115131,0.108963)
dog: 57%
car: 52%
truck: 56%
car: 62%
bicycle: 59%
```

En esta salida podemos observar algunas cosas interesantes. Las primeras líneas (las que están numeradas) representan cada una de las capas que conforman la red (si la comparamos con la Figura 3-1 veremos que coinciden), junto con algunos de sus parámetros característicos, como el tamaño del filtro o los volúmenes de entrada y salida. La última columna, además, muestra los *FLOPS* de cada capa, que es el número de operaciones en coma flotante realizadas por segundo (en este caso se muestran los *BFLOPS*, que equivale a cada billón[1] de operaciones por segundo) que se llevarán a cabo durante la ejecución. A continuación, se muestra cómo se cargan los pesos y entonces se lleva a cabo la detección, que podemos observar que en la prueba tarda poco más de cuatro segundos. Los resultados de esta detección se pueden ver justo después, donde se indica la cantidad

[1] Billón americano, equivale a mil millones europeos

de detecciones realizadas y se desglosa en cajas delimitadoras (primero) y objetos con su porcentaje de seguridad (después). Además, una representación gráfica del resultado se almacena en el archivo `predictions.png` al finalizar la ejecución. En la siguiente figura se representan tanto la imagen original como el resultado de la detección:

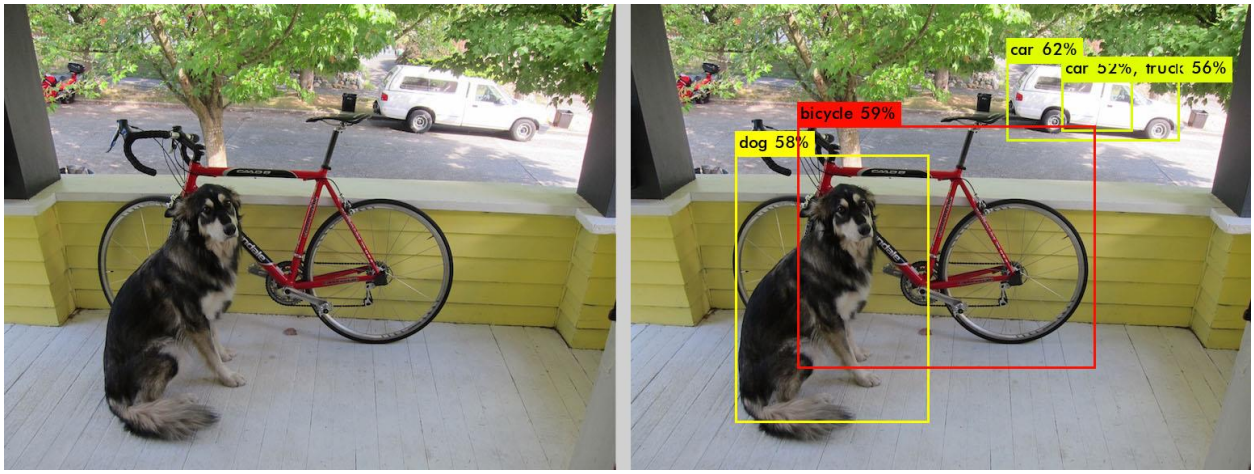


Figura 3-2. Resultado de la prueba de detección mediante TinyYOLOv3

Como se puede observar, los cuadros delimitadores y los objetos detectados corresponden con lo indicado en la salida textual por pantalla, pero mediante la imagen son más fáciles de interpretar. Respecto a la calidad de la detección, se advierten algunos errores: los cuadros delimitadores tanto del perro como de la bicicleta no son del todo precisos, y la furgoneta del fondo ha sido interpretada de tres maneras diferentes (dos de ellas redundantes), aunque las tres son relativamente correctas (el dataset de COCO no distingue específicamente a una furgoneta, tan sólo un coche o un camión, como es el caso aquí). Aun así, dadas las condiciones y el hecho de estar utilizando una red neuronal bastante sencilla como es TinyYOLOv3, es un resultado que podríamos tomar como positivo.

4 PRUEBAS Y VALIDACIÓN

En este capítulo se desarrollarán y explicarán las diferentes pruebas que se han realizado sobre la instalación comentada en el capítulo anterior, junto con el código y las modificaciones llevadas a cabo para realizar dichas pruebas. Éstas se utilizarán para validar, a continuación, los objetivos planteados para este proyecto, que, recordamos resumidamente, es comprobar el funcionamiento y rendimiento de TinyYOLOv3 sobre Raspberry Pi 3.

Como hemos ido adelantando en anteriores capítulos, las pruebas se dividirán en las dos funciones principales de los sistemas de detección de objetos: inferencia y entrenamiento. Para cada una de estas secciones se pretende completar los siguientes puntos:

- Experimentos a realizar
- Desarrollo de los experimentos y pruebas
- Validación de los experimentos y análisis de los resultados

En este último punto, además, extraeremos las conclusiones de dichos resultados, que serán desarrolladas más en profundidad en el siguiente capítulo.

4.1 Códigos desarrollados y modificaciones en Darknet

Como se ha mencionado previamente, para llevar a cabo las pruebas del proyecto se han desarrollado scripts y pequeños programas auxiliares, y se ha requerido realizar ciertas modificaciones del código fuente de Darknet. Se recuerda que parte de la distribución de shizukachan, y que todo puede hallarse directamente en el repositorio personalizado que se ha creado para este proyecto.

Esta sección estará dedicada a explicar estas modificaciones y la función de dichos scripts, y en el próximo capítulo, dedicado a pruebas y resultados, se pondrán en práctica y se indicará cuándo se han usado unos y otros. También en esta sección se detallas las librerías auxiliares requeridas para ejecutar algunos scripts, y el proceso de instalación de cada una de ellas. Los códigos fuente serán incluidos a su vez en los anexos.

4.1.1 Modificaciones en código fuente para el entrenamiento

Se llevaron a cabo en el código examples/detector.c, encargado tanto de la inferencia como del entrenamiento. En este caso, se trabajó sobre la función `train_detector`, que lleva a cabo todo el proceso de entrenamiento. Las modificaciones consisten en añadir las siguientes líneas al comienzo:

```
1.  #ifdef NNPACK
2.      nnp_initialize();
3.  #endif
```

Y las siguientes al final:

```
1.  #ifdef NNPACK
2.      nnp_deinitialize();
3.  #endif
```

La razón es que, durante las pruebas de entrenamiento, se detectó un fallo persistente que parecía estar relacionado con NNPACK. Comparando las funciones de inferencia (`test_detector`) y la mencionada de entrenamiento, se dedujo que las líneas antes presentadas servían para inicializar NNPACK, y al no estar siendo ejecutadas durante el entrenamiento, la librería no podía funcionar correctamente y eso conducía al error.

El motivo por el que algo tan crítico como esto no está incluido en las distribuciones originales puede ser que, en general, no se recomienda llevar a cabo un entrenamiento en un dispositivo tan poco potente y preparado como Raspberry Pi, por lo que la mayoría de estas distribuciones se centran más en la parte de inferencia y descuidan la de entrenamiento.

4.1.2 Script para detección rápida: `fast_detect.sh`

Este script no es más que un atajo para la línea:

```
$> ./darknet detect cfg/yolov3-tiny.cfg yolov3-tiny.weights <image>
```

Donde `<image>` puede ser cualquier de las imágenes que se incluyen en la carpeta `testing/`. Un ejemplo de ejecución podría ser el siguiente:

```
$> ./fast_detect.sh dog.jpg
```

Cuyos resultados serían exactamente los mismos que veíamos previamente en la sección de validación de instalación.

El código de este script puede hallarse en el Anexo C.

4.1.3 Código para pruebas de inferencia

Con el fin de poder obtener datos relevantes de la ejecución del proceso de detección, más allá de comprobar que funciona con una sola imagen como se ha hecho en la sección anterior, se ha desarrollado este pequeño programa escrito en Python que adapta el mecanismo de detección anterior, y que permite inferir sobre múltiples imágenes y obtener resultados útiles como los FPS y representarlos gráficamente.

Principalmente, el código se compone de un *wrapper*, que, de forma general, es un tipo de librería diseñada para permitir ejecutar funciones de un código escrito en cierto lenguaje X desde otro escrito en un lenguaje Y. En este caso, utilizaremos Python para ejecutar las funciones de C que conforman Darknet y que permiten llevar a cabo el proceso de inferencia. De esta manera, podemos aprovechar las ventajas que nos ofrece Python a la hora de captación, manejo y representación de datos; sin alterar ni tener que reescribir el núcleo de Darknet.

Debido a que algunas de las partes del código en C no podían ser replicadas directamente en Python, se han creado funciones adaptadoras en C, localizadas en los módulo `src/py_utils.h` y `src/py_utils.c`, que realizan estas tareas y pueden ser llamadas luego a través del *wrapper* desde Python. Estos dos archivos se han incluido en el Anexo G.

Se puede probar el funcionamiento del programa ejecutando simplemente:

```
$> python3 detect.py
```

Cuya salida, ignorando la parte de descripción de la red (que debería ser la misma que con la detección estándar mediante Darknet) debería ser:

```
Inferencia de 1 imágenes
Duración total = 4.9484
Duración media por imagen = 4.9484
FPS medios = 0.2021

Imagen: testing/dog.jpg
 - Tiempo: 4.9484
 - FPS acumulado: 0.2021
 -  'dog'       con prob 0.5707 at (x, y) = (0.3243, 0.6113) with (width, height) =
(0.3123, 0.5742) and objectness = 0.5721
 -  'car'       con prob 0.5173 at (x, y) = (0.7519, 0.2188) with (width, height) =
(0.1151, 0.1090) and objectness = 0.8537
 -  'truck'     con prob 0.5578 at (x, y) = (0.7519, 0.2188) with (width, height) =
(0.1151, 0.1090) and objectness = 0.8537
 -  'car'       con prob 0.6153 at (x, y) = (0.7451, 0.2092) with (width, height) =
(0.2791, 0.1714) and objectness = 0.8626
 -  'bicycle'   con prob 0.5850 at (x, y) = (0.5090, 0.5218) with (width, height) =
(0.4810, 0.5197) and objectness = 0.5878
```

Si no se indica lo contrario, se usará la misma imagen que utilizábamos en secciones previas para probar el

detector; y podemos comprobar que, en efecto, los resultados son los mismos. En el próximo capítulo utilizaremos este programa para realizar las pruebas de inferencia.

El código fuente se puede consultar en el Anexo D. Se pueden ver las distintas opciones de funcionamiento del programa ejecutando:

```
$> python3 detect.py -h
```

4.1.4 Código para análisis de las pruebas de entrenamiento

Como se verá más adelante en detalle, el comando de Darknet para llevar a cabo el entrenamiento genera una salida por pantalla en la que se van mostrando, poco a poco, los resultados de éste. El programa propuesto, plotter.py, es capaz de analizar dichos resultados y mostrarlos gráficamente, generando diferentes gráficas con las que estudiar la evolución del entrenamiento.

Para poder ejecutar dicho programa, es necesario instalar ciertas librerías auxiliares de Linux y Python, orientadas al manejo de gráficos. Se pueden seguir los siguientes pasos:

```
$> sudo apt-get install python3-pip libatlas-base-dev libopenjp2-7 libtiff5
$> pip3 install numpy matplotlib
```

Para comprobar que el código funciona, se puede ejecutar de la siguiente manera y comprobar que el resultado coincide con el mostrado (saltará un error y se mostrará la ayuda):

```
$> python3 plotter.py
usage: plotter.py [-h] [-x X] [-y Y] [--csv CSV] [--plot_file PLOT_FILE]
                  logfile
plotter.py: error: the following arguments are required: logfile
```

El código fuente se puede consultar en el Anexo E. Se pueden ver las distintas configuraciones de funcionamiento ejecutando:

```
$> python3 plotter.py -h
```

4.2 Pruebas de inferencia

Con estas pruebas principalmente se busca comprobar si TinyYOLOv3 puede ser utilizado para generar detecciones en largos conjuntos de imágenes, experimento que puede servir de precedente para probar su utilidad en vídeos, por ejemplo, captados por una cámara web conectada a la Raspberry. Los puntos a tratar para determinar su viabilidad son:

- **Duración de la inferencia** en un conjunto amplio y controlado de imágenes. Mediremos tanto tiempos como FPS (imágenes por segundo), y el resultado será más adecuado cuanto más se reduzca el tiempo y más se incrementen los FPS. Como referencia, podemos tomar que, según el repositorio de la distribución de digitalbrain79, el tiempo medio de predicción ronda 1 segundo por imagen.
- **Calidad de las predicciones:** Habrá que comprobar que en las imágenes en las que se ejecute la inferencia se detectan los objetos correctos, en su posición y con sus recuadros pertinentes. Existen varias métricas para evaluar la precisión, como la mAP, que hemos mencionado en varias ocasiones.
- **Utilidad de la inferencia:** Determinar si es recomendable o no utilizar TinyYOLOv3 en una Raspberry Pi para llevar a cabo inferencias en imágenes, acorde a los resultados de las pruebas, y también decidir si se podría llegar a usar para detectar objetos sobre vídeos.

Como recordatorio, el proceso de inferencia consiste en la aplicación de una red neuronal ya suficientemente entrenada en la realización de cierta tarea, por lo que alimentaremos la red con datos específicos y obtendremos resultados acordes a la tarea en cuestión. En este caso, utilizaremos imágenes como entrada, y nuestra red, TinyYOLOv3, identificará y ubicará en éstas los objetos en los que ha sido entrenada para detectar.

4.2.1 Experimento a realizar

En primero lugar, puesto que sólo vamos a probar la inferencia, requerimos de una red ya entrenada, y también debemos saber en qué tipo de imágenes y objetos se entrenó, para escoger un conjunto de muestras cuyo contenido la red sea capaz de detectar. Para este experimento trabajaremos concretamente con la red pre-entrenada que usábamos para probar la implementación en el capítulo anterior. Ésta está entrenada en el dataset de COCO 2017 (https://cocodataset.org/#download), que distingue entre 80 clases de conceptos comunes y cotidianos, entre personas, animales, vehículos, comidas, herramientas y un largo etcétera. Su set de muestras de entrenamiento se compone de 200000 imágenes, y también ofrece un conjunto de imágenes de prueba, por lo que podemos usarlo directamente para la inferencia, garantizando que serán imágenes válidas para la red. Como veíamos además en la prueba de la implementación, en la distribución de Darknet ya disponemos de todos los archivos necesarios para poder realizar la inferencia (configuración y datos), y el script *detect.py* está programado para usarlos por defecto.

Como se explicaba en la introducción, el objetivo será analizar el comportamiento de la red a la hora de inferir varias imágenes seguidas, una tras otra, midiendo tanto la media de tiempo por imagen y de FPS, como la evolución de dichos valores a lo largo de la inferencia. Para ello, se utilizará un subconjunto de 1000 imágenes obtenidas del conjunto de prueba ofrecido por el dataset de COCO, y se utilizará *detect.py* tanto para calcular los valores de duración y FPS, como para generar las gráficas pertinentes.

Por otro lado, también querremos medir la precisión de las detecciones, con el fin de analizar el rendimiento de TinyYOLOv3. Para ello, prepararemos otro subconjunto de imágenes, en este caso de 10, que sean variadas en tamaño y calidad; contengan diferentes elementos fácilmente identificables; y de las que dispongamos de los objetos que sobre el papel deberían ser detectados por la red. Tras la inferencia, sobre los resultados obtenidos (recordemos que Darknet representará sobre las imágenes los recuadros indicando los objetos detectados), utilizaremos la métrica conocida como *puntuación* (*F1-score*) [49], con la que determinaremos la calidad de las predicciones. La razón de usar esta métrica sobre otras, como la mAP que tanto hemos mencionado, es que es más sencilla de calcular, teniendo en cuenta que realizaremos el proceso a mano, en vez de automatizarlo como hacemos con, por ejemplo, los FPS. La puntuación F1 es un test sencillo generalmente utilizado en análisis estadístico que relaciona los conceptos de *precisión* y *exhaustividad*. Las fórmulas que lo gobiernan se representan a continuación:

$$F1 = 2 \cdot \frac{precision \cdot recall}{precision + recall} \tag{4-1}$$

$$precision = \frac{TP}{TP + FP} \tag{4-2}$$

$$recall = \frac{TP}{TP + FN} \tag{4-3}$$

Donde:

- ***TP***, o ***Verdadero Positivo*** (*True Positive*) son todos aquellos objetos correctamente detectados en un conjunto de imágenes inferidas; esto es, se han clasificado bien, y su cuadro delimitador es lo suficientemente preciso respecto al original (ver Figura 4-1).
- ***FP***, o ***Falso Positivo*** (*False Positive*) son aquellos casos en los que la clasificación del objeto ha sido correcta, pero o bien el cuadro delimitador no es del todo preciso, o bien hay varios cuadros delimitadores para un mismo objeto. También se considerarían falsos positivos los casos en los que se detectan objetos que no están en la imagen.
- ***FN***, o ***Falso Negativo*** (*False Negative*) es cuando el objeto no se ha detectado, o se ha detectado pero no se ha identificado correctamente.
- ***Precision***, o **precisión**, indica cuántos objetos se han clasificado correctamente, de todos aquéllos que se han identificado.
- ***Recall***, o **exhaustividad**, mide cuántos objetos han sido correctamente identificado, de todos los objetos potencialmente detectables en la imagen.

- ***F1*** es el valor final del factor, que relaciona, para el set total, la precisión y la exhaustividad en ésta.

Generalmente, para determinar cuándo estamos ante un verdadero o falso positivo, utilizamos la IoU con un valor de umbral cercano 0'5. Esto quiere decir, si recordemos que la IoU mide la diferencia entre el cuadro delimitador generado por la red y el original mostrado en el dataset, que tan sólo aceptaremos como verdadero positivo aquellas detecciones cuyo recuadro marque, al menos, la mitad del objeto en cuestión. Para estas pruebas, en las que carecemos de un mecanismo para realizar esta operación de una forma más precisa, consideraremos que la detección es un verdadero positivo si el cuadro delimitador envuelve la mayor parte del objeto, a simple vista. También hay que destacar que, en líneas generales, este tipo de factores, tanto F1 como mAP, suelen aplicarse *por cada clase* del dataset independientemente. En este caso, como tan sólo queremos algo orientativo y el conjunto de imágenes sobre el que trabajaremos es pequeño, llevaremos a cabo la operación sobre todas las clases detectadas a la vez.

Los pasos a seguir para calcular este factor serán:

1. Llevar a cabo la inferencia del set de 10 imágenes, almacenando las predicciones de Darknet.
2. Por cada imagen, contabilizar los verdaderos y falsos positivos y los falsos negativos, comparando directamente con las detecciones teóricas que la página de COCO propone y que se representan en la Figura 4-1.
3. Calcular la precisión y exhaustividad, y posteriormente la F1.

Por último, queda por mencionar que se llevarán a cabo varias pruebas para cada uno de los puntos a validar. Todas consistirán en variar el tamaño de las imágenes de entrada de la red (usando siempre múltiplos de 32). Si recordamos de la explicación teórica, YOLOv3 ajusta el tamaño de las imágenes en su entrada (manteniendo siempre las proporciones), por lo que este parámetro se puede alterar en el archivo de configuración para distintos fines. En este experimento, por ejemplo, reducir el tamaño de las imágenes de entrada debería suponer una mejora en la velocidad de la inferencia, al tener un volumen de entrada menor, que supone menos neuronas en las diferentes capas, que suponen menos operaciones a llevar a cabo; pero a su vez, la calidad y precisión de las detecciones puede verse disminuida. Esto último es especialmente sensible, ya que depende tanto del tamaño con el que la red se entrenó en un principio, como del tamaño de las imágenes que se vayan a usar durante la inferencia, ya que si los tamaños originales distan mucho del usado por la red, se puede perder información y calidad en la imagen durante el cambio de tamaño, sobre todo cuando ésta se comprime a dimensiones inferiores. En el caso del dataset de COCO, los tamaños máximos de imagen son de 640x640, y los mínimos de 280x280; y la red se entrenó con un tamaño de entrada de 416x416 (que si nos fijamos, es un punto más o menos intermedio). Con el fin de tratar de conseguir la mejor relación entre calidad de predicción y duración de la inferencia, se probarán los siguientes tamaños para cada uno de los objetivos a tratar:

- Para las pruebas de precisión, sobre el subconjunto de 10 imágenes, se realizarán trece pruebas, desde 608x608 hasta 224x224, disminuyendo en cada prueba 32 puntos (ya que recordemos que el tamaño de entrada siempre debe ser múltiplo de 32); y se compararán entre ellas los valores de precisión, exhaustividad y F1.
- Para las pruebas de duración, sobre el subconjunto de 1000 imágenes, se realizarán cuatro pruebas, para entradas de 448x448, 416x416, 352x352, y 288x288; y se compararán entre ellas las mejoras en tiempos de ejecución y FPS.

4.2.2 Desarrollo

Se han preparado, para la realización del experimento, dos carpetas, *detect_test* y *qual_test*, cada una, respectivamente, con las imágenes para las pruebas de duración, y las de precisión. Estas últimas se muestran en la Figura 4-1, junto con los elementos que habría que detectar en cada una de ellas, obtenidos directamente de la página del dataset de COCO.

Para llevar a cabo los experimentos utilizaremos el script *detect.py* con diferentes parámetros en función de la prueba que vayamos a realizar. El script automáticamente cargará los archivos de configuración, datos y pesos

necesarios, todos relativos a TinyYOLOv3 y el dataset de COCO.

En el caso de las pruebas de duración, seleccionaremos la carpeta correspondiente, *detect_test*, y activaremos la generación de gráficas de progreso de los tiempos y FPS:

```
$> python3 detect.py -i detect_test/ -g -v
```

Una vez la ejecución finalice, se imprimirán por pantalla los resultados de tiempos y FPS medios, y se habrán generado las gráficas, que almacenaremos para futuras comparaciones.

En el caso de las pruebas de precisión, seleccionaremos la carpeta con las diez imágenes y activaremos la opción para que se generen las predicciones sobre dichas imágenes, que serán almacenadas en una carpeta *predictions*. Utilizaremos las predicciones para recopilar los valores que nos permitirán calcular la F1:

```
$> python3 detect.py -i qual_test/ -p -v
```

Finalmente, las modificaciones de tamaños de entrada de las imágenes en la red deberán hacerse en el archivo de configuración de TinyYOLOv3, *cfg/yolov3-tiny.cfg*, en los hiperparámetros de la sección `[net]`; `width` y `height`.

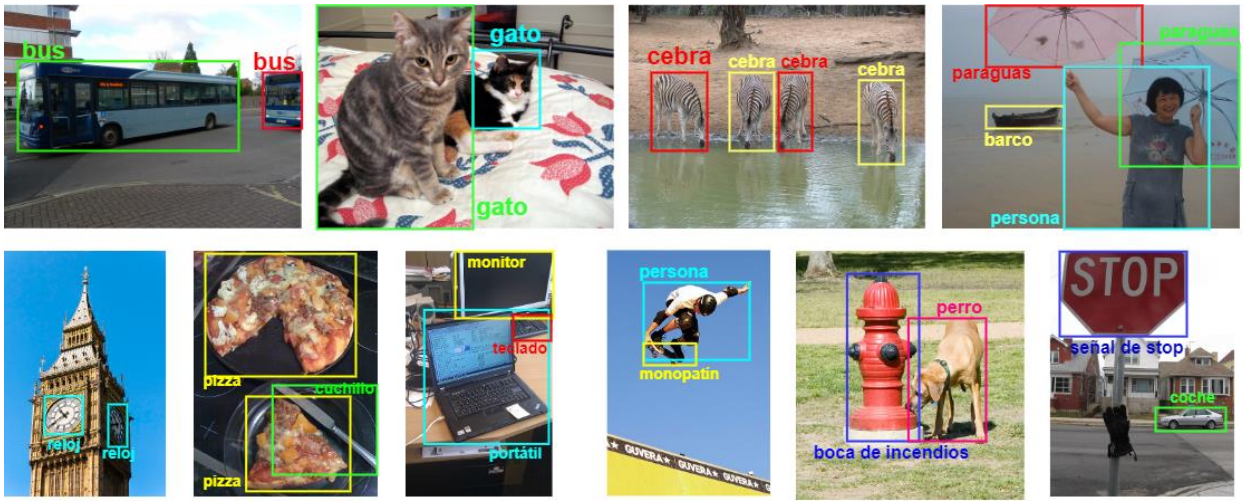


Figura 4-1. Imágenes usadas para analizar la precisión

4.2.3 Resultados y validación

Analizando en primer lugar las pruebas de precisión, se completó la Tabla 4-1, en la que se muestran todos los datos recopilados para cada uno de los tamaños de imagen de entrada, y en la que podemos apreciar que:

- Los valores de precisión son generalmente altos, siendo la media de alrededor de 0’8, lo que quiere decir que, sobre los objetos detectados, la clasificación y el cuadro delimitador de éstos son mayoritariamente acertados. En efecto, si recordamos de la explicación teórica de YOLO, este algoritmo se destaca por ser especialmente preciso.
- Los valores de exhaustividad, por el contrario, no son tan alentadores como los de precisión. La media ronda los 0’6 puntos, lo que quiere decir que alrededor de cuatro de cada diez objetos no se llegan a detectar. Incluso en el caso más favorable, 416x416, estamos en menos de un 0’75, es decir, que se pierde uno de cada cuatro objetos detectables. Esto también puede relacionarse con uno de los problemas principales de YOLO, que es la detección de objetos pequeños o que comparten una ubicación muy cercana a otro objeto.

Tabla 4-1. Resultados pruebas de precisión en inferencia

	TP	**FP**	**FN**	***precision***	***recall***	***F1***
608x608	10	4	12	0,714	0,455	0,556
576x576	12	6	8	0,667	0,600	0,632
544x544	14	3	9	0,824	0,609	0,700
512x512	16	2	8	0,889	0,667	0,762
480x480	13	3	10	0,813	0,565	0,667
448x448	17	1	8	0,944	0,680	0,791
416x416	17	3	6	0,850	0,739	0,791
384x384	12	4	10	0,750	0,545	0,632
352x352	15	4	7	0,789	0,682	0,732
320x320	12	4	10	0,750	0,545	0,632
288x288	13	4	9	0,765	0,591	0,667
256x256	13	3	10	0,813	0,565	0,667
224x224	10	6	11	0,625	0,476	0,541

Otro factor que no se puede apreciar en la tabla pero que se ha notado al analizar las predicciones realizadas es que, en bastantes ocasiones, se generaban varios cuadros delimitadores para un mismo objeto, clasificándose correctamente, pero suponiendo un falso positivo. Esto es algo que podría corregirse modificando el valor de NMS, que si recordamos, es el mecanismo que YOLO utiliza para combinar cuadros delimitadores sobre mismos objetos. Las pruebas se realizaron con un valor por defecto de 0’3 (que es menor que el por defecto que usa YOLO, de 0’45). Aunque no es recomendable reducir mucho este valor, pues puede provocar que objetos distintos se acaben englobando dentro de los mismos recuadros, de hacerlo se podría prever una disminución de los falsos positivos en pos de los verdaderos positivos, lo que incrementaría la precisión. Al ser ésta ya bastante favorable, no se ha visto necesidad en repetir las pruebas.

Por otro lado, hay que destacar que todos estos valores son orientativos. Al ser el dataset muy reducido y con imágenes manualmente seleccionadas, y al estar calculados los positivos y negativos a mano; los resultados pueden no ser estrictamente representativos de todo el dataset. Aun así, pueden servir como primera impresión.

Si representamos gráficamente la precisión y la exhaustividad, podremos ver que, como es de esperar, a medida que nos alejamos del tamaño medio de las imágenes, se empeoran los resultados, al estar éstas deformadas:

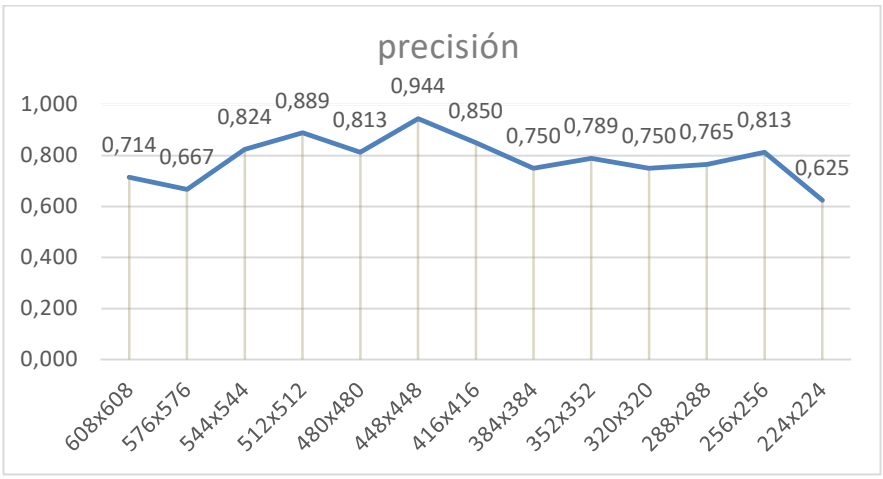


Figura 4-2. Gráfica de precisión para varios tamaños

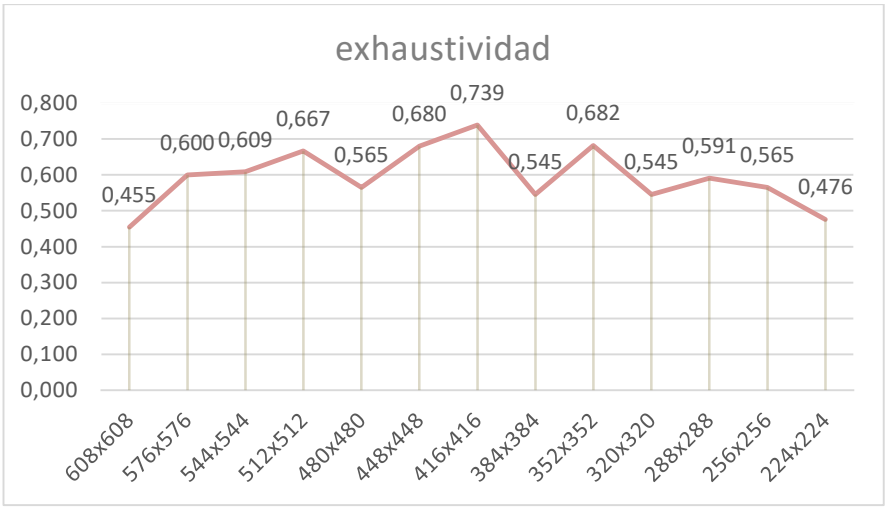


Figura 4-3. Gráfica de exhaustividad para varios tamaños

El resultado final de F1 lo podemos ver a continuación, comparado con las gráficas de precisión y exhaustividad también:

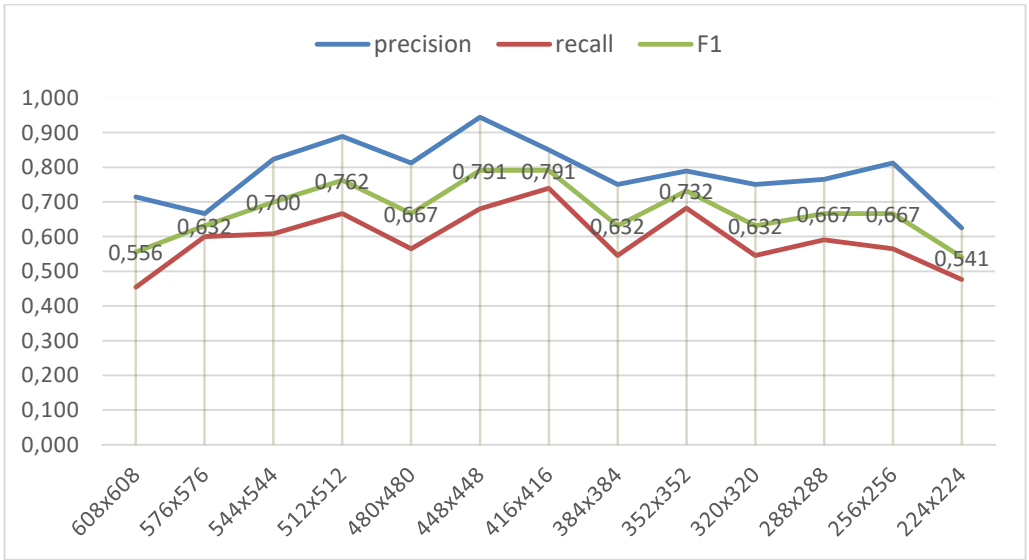


Figura 4-4. Gráfica de F1 para varios tamaños

Los valores más altos se alcanzan en las medidas 448x448 y 416x416. Ambas cobran bastante sentido, al estar entorno a la media de los tamaños de las imágenes del dataset (que sería 432x432) y, además, ser la segunda la medida en la que está entrenada la red. En conjunto, un valor de F1 de prácticamente 0'8 es bastante aceptable, si lo comparamos entre otros algoritmos de detección de objetos [50]:

Tabla 4-2. Comparativa F1 entre diferentes algoritmos

	Precisión	**Exhaustividad**	**F1**
SSD	0,99	0,68	0,84
Faster R-CNN	0,82	0,94	0,87
YOLOv3	0,98	0,91	0,94
Experimento (416x416)	*0,85*	*0,74*	*0,79*

Para las pruebas de duración, el resultado final se puede observar en la siguiente tabla:

Tabla 4-3. Resultados de pruebas de duración en inferencia

	448x448	**416x416**	**352x352**	**288x288**
Duración total de la prueba (en minutos)	126,21	118,28	81,16	59,51
Tiempo medio por imagen (en segundos)	7,573	7,097	4,869	3,571
FPS medio de la prueba	0,132	0,141	0,205	0,280

Como era de esperar, los tiempos van disminuyendo a medida que variamos el tamaño de las imágenes (y, en consecuencia, aumentan los FPS). Aun así, es apreciable que no se alcanza el tiempo de 1 segundo que se indica en el repositorio de digitalbrain79, y los valores de FPS no son especialmente favorables si el objetivo es aplicarlo en vídeo, ya que actualmente, en el mejor de los casos (que sería 288x288, perdiendo precisión en las detecciones), tenemos que no se infieren siquiera una imagen por segundo (cuando YOLOv3 puede llegar a alcanzar 23 FPS sobre CPU, trabajando en un dispositivo adecuado). Además, si observamos las gráficas generadas con la evolución de unos parámetros y otros, podemos ver que el rendimiento de la red empeora a medida que se van infiriendo más imágenes:

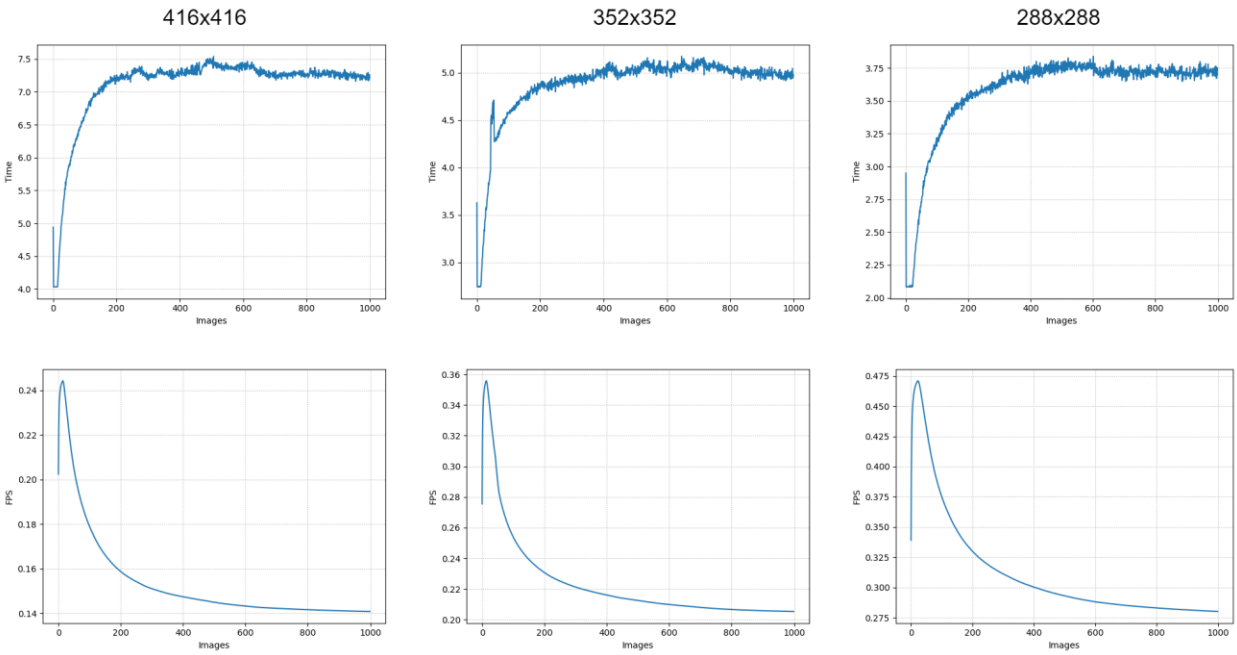


Figura 4-5. Gráficas de progresión de tiempo y FPS

Éste es un comportamiento extraño, ya que nuestra implementación de la red es *sin estado* (*stateless*), es decir, el resultado en cada inferencia es completamente independiente a los anteriores o posteriores, por lo que deberían tardar siempre lo mismo dada una misma configuración, y no se justifica lo que parece ser un régimen transitorio hasta llegar a la estabilidad. Como el propósito de este experimento era determinar el resultado de tiempos y FPS al finalizar la ejecución completa de la inferencia (y éste se ha logrado), no es objeto de esta memoria resolver esta cuestión, aunque, atendiendo a la forma de las gráficas, una de las razones más plausibles es que haya problemas con la gestión de la memoria en cada inferencia. Cada vez que se ejecuta una detección, Darknet debe reservar memoria para cargar la imagen que vaya a inferir, y, posteriormente, deberá liberar esta memoria. Si este proceso de carga-liberación no se realiza lo suficientemente rápido, la siguiente imagen que haya que cargar puede encontrarse con que no hay espacio y deberá esperar a que la memoria se libere para poder proseguir. Lo que estaríamos viendo en este caso en la zona de estabilidad es el punto en el que, cada vez que se requiere reservar espacio de memoria, ésta está completamente llena y hay que esperar a que se libere, introduciendo de esta manera un retardo que acaba siendo computado como de tiempo de inferencia.

Finalmente, se puede concluir que, en general, TinyYOLOv3 tiene un buen rendimiento en cuanto a la calidad de las predicciones, pero se ve mermado por el tiempo que tarda en realizarlas. La siguiente tabla refleja los resultados combinados para los tamaños de imagen de 448x448 y 416x416, que son los que mejores han desempeñado:

Tabla 4-4. Resultados finales pruebas de inferencia

	448x448	**416x416**
Precisión	0,944	0,850
Exhaustividad	0,680	0,739
F1	0,791	0,791
Tiempo medio por imagen	7,573	7,097
FPS medio	0,132	0,141

A modo de resumen, se muestra la siguiente tabla, indicando los puntos que se proponían al comienzo del ejercicio y su resolución:

Tabla 4-5. Resultados de las pruebas de inferencia

Objetivo	Resultado	Observaciones
Duración y FPS	Insatisfactorio	Los tiempos por imagen son demasiado altos, incluso cuando se reduce el tamaño de las imágenes; provocando que los FPS no lleguen ni a la imagen por segundo.
Calidad de las predicciones	Satisfactorio	Las pruebas de calidad demuestran que se pueden realizar buenas predicciones utilizando TinyYOLOv3, alcanzando una puntuación de F1 cercana a 0'8 sobre 1.
Utilidad	Inconcluyente	Los buenos resultados de la calidad de las predicciones contrastan con los obtenidos en las pruebas de duración. Para imágenes, el experimento es exitoso, pero para vídeo, sobre todo si es vídeo en directo como podría ser una webcam, los resultados no arrojan buenas impresiones (habría que analizar directamente el comportamiento de la red en estos entornos).

4.3 Pruebas de entrenamiento

El objetivo de estas pruebas será comprobar si podemos entrenar una red TinyYOLOv3 directamente funcionando sobre una Raspberry Pi 3 con un conjunto de datos escogidos por nosotros. El concepto del experimento a realizar siempre será el mismo, pero se modificarán los distintos parámetros y configuraciones del entrenamiento en la medida de lo necesario con el fin de solucionar posibles fallos u obtener unos resultados más precisos. En concreto, se persigue comprobar, medir y valorar los siguientes puntos:

- **Viabilidad del entrenamiento:** El proceso de entrenamiento de una red neuronal es el más costoso computacionalmente, por lo que cabe la posibilidad de que, al realizarlo en un dispositivo ligero y no especializado, el programa falle, no pudiendo realizarse el experimento.
- **Duración y calidad del entrenamiento**: En caso de que el entrenamiento prospere, tendremos que medir tanto el tiempo que tarda la red en entrenarse lo suficiente como para empezar a hacer predicciones mínimamente aceptables (lograr un error de entrenamiento cercano a cero), como el porcentaje de acierto que presenta y la calidad de éste.
- **Utilidad del entrenamiento:** Determinar finalmente si merece o no la pena entrenar la red en Raspberry, acorde al resto de resultados obtenidos y la dificultad percibida en el proceso de entrenamiento, así como las condiciones en las que un entrenamiento óptimo podría llevarse a cabo.

Se recuerda que el proceso de entrenamiento de una red neuronal es aquél mediante el cual conseguimos que la red *aprenda* a realizar cierto tipo de tarea, partiendo para ello de un conjunto de datos y sus resultados asociados. En este caso, este conjunto de datos será un dataset, o conjunto de muestras, compuesto por *imágenes anotadas*, es decir, imágenes en las que los objetos que queremos que la red detecte y la posición de éstos aparecen indicados de alguna forma que la red pueda interpretar y utilizar para entrenarse. Estas indicaciones,

generalmente llamadas *anotaciones*, suelen venir en archivos aparte, íntimamente ligados a cada una de las imágenes cuyos objetos identifican. Idealmente, al finalizar dicho entrenamiento tendríamos una red que sería capaz de, para una imagen arbitraria, detectar todos aquellos objetos en los que ha sido entrenada, como hacíamos en las pruebas de inferencia.

4.3.1 Experimento a realizar

Dado que el objetivo es únicamente comprobar la viabilidad del entrenamiento y medir su rendimiento, el experimento será sencillo, consistiendo en un entrenamiento sobre un dataset que sólo distingue entre dos objetos (es decir, tiene dos clases), perros y gatos, y que se compondrá de alrededor de 3700 imágenes. Comparado con el dataset en el que estaban pre-entrenados los pesos usados para las pruebas de inferencia –200000 imágenes de entrenamiento, 80 clases-, la diferencia es bastante apreciable. El dataset seleccionado ha sido obtenido de Kaggle (https://www.kaggle.com/), una plataforma web que, entre otros, ofrece múltiples conjuntos de imágenes gratuitos para distintos fines, entre ellos el entrenamiento de redes neuronales; y nos hemos decantado por el mencionado de perros y gatos en concreto porque tenía un número total de clases e imágenes reducido, y sus estas últimas además ya estaban anotadas (de lo contrario, habría que haber usado alguna herramiento que nos permitiese anotar, uno por uno, los objetos de cada imagen).

El entrenamiento se llevará a cabo directamente con Darknet, y utilizaremos los logs que se generan durante éste en conjunto con el script *plotter.py* para crear gráficas y analizar y visualizar su evolución. Generalmente representaremos las pérdidas de entrenamiento (*average loss*) en función del número de épocas o iteraciones, cuyas diferencias son:

- En función de las iteraciones tendremos las pérdidas cada vez que un grupo de imágenes (lote o mini-lote) actualizan los parámetros y pesos de la red.
- En función de las épocas, tendremos las pérdidas cada vez que *todas* las imágenes han alimentado la red, y sus parámetros se han actualizado varias veces en consecuencia.

Además, para identificar cuándo dar el entrenamiento por finalizado, buscaremos un valor de pérdidas estable, que no parezca reducirse más; y, gracias a que Darknet genera automáticamente copias de los pesos en distintos momentos del entrenamiento (cada 100 o 1000 iteraciones en función de la distribución, como veremos más adelante), podremos usar el modelo y obtener los resultados que pasaremos a evaluar.

A continuación se explicará cómo se lleva a cabo el proceso de entrenamiento en YOLOv3 y Darknet, y se detallarán las configuraciones y valores de hiperparámetros elegidos para este experimento y su motivación.

4.3.2 Desarrollo

En primer lugar, tenemos que echar un vistazo al archivo de datos de Darknet, que introducíamos en el capítulo dedicado a la implementación, ya que en éste se organizan los componentes necesarios para el entrenamiento:

```
classes = 2
train = training/oxford-pet/cat-dog-train.txt
valid = training/oxford-pet/cat-dog-test.txt
names = training/oxford-pet/cat-dog.names
backup = backup
```

Esta es una copia del archivo utilizado para este experimento. En él podemos ver tanto el número de clases detectables como el nombre y localización del archivo de nombres, que explicábamos anteriormente; pero ahora podemos centrarnos también en el resto de parámetros.

Tanto `train` como `valid` apuntan a sendos archivos de texto, en los que debe aparecer un listado con la ruta de cada una de las imágenes que conforman el dataset de entrenamiento y validación, respectivamente. Por ejemplo, las primeras líneas del archivo de entrenamiento utilizado para este dataset son los siguientes:

```
training/oxford-pet/images/training/cat_0.jpg
training/oxford-pet/images/training/cat_1.jpg
training/oxford-pet/images/training/cat_10.jpg
(…)
```

Todas las imágenes deben estar incluidas en la misma carpeta (las de entrenamiento por un lado y las de validación por otro) y cada una de ellas deberá ir acompañada de un *archivo de anotaciones*, que ya introducíamos previamente, y su localización. Dicho archivo será un texto plano (necesariamente `.txt`) con el mismo nombre que la imagen a la que "representa", y las anotaciones tendrán el siguiente formato (cada una, en caso de haber varios objetos en la imagen, en una línea):

```
<object-class> <x> <y> <width> <height>
```

Donde `<object-class>` debe ser el índice de la línea en la que dicha clase aparece representada en el archivo de nombres; `<x>` e `<y>` son las posiciones relativas del centro del cuadro delimitador; y `<width>` y `<height>` su anchura y altura relativa, respectivamente.

Un ejemplo podría ser el archivo `dog_2408.txt`, que necesariamente representa a la imagen `dog_2408.jpg`, y cuyo contenido es:

```
1 0.337000 0.377333 0.646000 0.749333
```

Lo que significa que hay un único objeto en dicha imagen, identificado por el índice 1 (que si mirásemos el archivo de nombres, *cat-dog.names*, veríamos que referencia a un perro), y con la localización relativa indicada por los cuatro valores correspondientes.

Cerrando la explicación de estos dos archivos, cabe destacar que, mientras que el archivo de entrenamiento es obligatorio, el de validación es opcional, y, además, hay que indicarle a Darknet específicamente que lo use, pues la ejecución estándar no lo tiene en cuenta. Esto es así porque el proceso de validación ralentiza todo el entrenamiento e incrementa el coste computacional de éste. Debido a ello, para este proyecto **no** se ha tenido en cuenta en un primer lugar, y se ha incluido en el archivo de datos sólo con carácter explicativo. Si los resultados de las pruebas en las condiciones antes especificadas y sin validación fuesen favorables, se podría plantear una prueba con validación, que permitiría acotar con más precisión los fallos que pudieran ocurrir durante el entrenamiento, como se explicaba en capítulos anteriores.

El último parámetro del archivo de datos es `backup`, que apunta a la carpeta que almacenará lo que Darknet denomina *backups*. Éstos son archivos de pesos, iguales a los que explicábamos en el capítulo anterior, que el programa genera de forma automática durante el entrenamiento, cada cierto número de iteraciones (recordemos que una iteración es el proceso por el cual un lote completo de imágenes alimenta y entrena la red neuronal una vez). Gracias a ellos podemos tener un histórico de pesos en diferentes momentos del entrenamiento, con los que podremos ir probando poco a poco el funcionamiento y/o rendimiento de la red, y así determinar si éste está progresando correctamente o no, o si puede darse por finalizado o debe continuar. En función de la distribución de Darknet utilizada, estos archivos se generan cada 100 iteraciones (shizukachan, personalizado) o 1000 (digitalbrain79).

Continuando con la preparación del entrenamiento, tenemos también que modificar algunos parámetros del archivo de configuración de la red. En el capítulo anterior ya detallábamos los hiperparámetros que componían cada una de las secciones de dicho archivo, por lo que ahora nos centraremos simplemente en aquéllos que nos son relevantes para el entrenamiento:

- **Tamaño de las imágenes**, mediante `width` y `height`, determinará el tamaño de las imágenes con las que vamos a entrenar la red. Hay que recordar que Darknet modificará el tamaño de las imágenes que introduzcamos en la red para ajustarlo a aquél que indiquemos aquí. El estándar de TinyYOLOv3 es de 416x416, pero para no tener problemas con la memoria, reduciremos este valor en las pruebas, sabiendo que es necesario que dicho tamaño sea múltiplo de 32.
- **Lotes y mini-lotes**, mediante `batches` y `subdivisions`, con los que controlaremos con cuántas imágenes se alimenta a la red en cada iteración y con cuántas actualizaremos los parámetros de ésta. TinyYOLOv3 recomienda 64 lotes y 16 subdivisiones (lo que supone mini-lotes de 4 imágenes). Cuantas menos subdivisiones más rápido será el entrenamiento, pero a su vez más se cargará la memoria porque habrá más imágenes procesándose, por lo que hay que tenerlo controlado.
- **Máximo número de lotes y pasos de modificación del ratio de aprendizaje**, a controlar mediante `max_batches` y `steps`, que determinarán respectivamente tras cuantas iteraciones dar por finalizado

el entrenamiento, y cada cuántas iteraciones modificar el ratio de aprendizaje. Estos dos parámetros están íntegramente relacionados, ya que se recomienda desde el repositorio principal de YOLO que los pasos sean dos: uno al 80% del número máximo de lotes, y otro al 90%. A su vez, no se recomienda que este número máximo de lotes sea inferior a 6000. Como queremos optimizar la velocidad del entrenamiento, ya que éste va a ser lento de cualquier manera debido a las limitaciones de la Raspberry, mantendremos dicho valor máximo de lotes, con dos pasos a las 4800 y 5400 iteraciones.

- **Modificaciones en los hiperparámetros de las capas YOLO y previas a YOLO**. En concreto, en la capa `[yolo]` hay que modificar el valor de `classes` al número de clases disponibles, en nuestro caso 2; y en la capa previa a ésta, que será convolucional, hay que cambiar el valor de `filters` por *(clases + 5)*3*, que será 21 en este caso.

Para finalizar con los archivos necesarios para llevar a cabo el entrenamiento, Darknet requiere un archivo de pesos *de partida*. Como explicábamos en los fundamentos teóricos, cuando una red neuronal se entrena en una tarea nueva, parte de unos pesos y parámetros arbitrarios, a partir de los cuáles modifica y genera los óptimos. En Darknet podemos especificar el valor base de dichos parámetros, lo que, entre otras cosas, nos permitiría mejorar una red ya entrenada, o reducir el tiempo de entrenamiento de redes diferentes pero con datos similares. Más aun, Darknet permite crear, a partir de un archivo de pesos previo y el archivo de configuración que vayamos a usar, un archivo de pesos de partida ajustado, en el que "congela" los pesos de algunas capas iniciales, forzando a que la red no se centre en entrenar y modificar dichos parámetros, que se especializan en rasgos y características más simples y comunes y pueden mantenerse constantes entre diferentes datasets; sino que lo haga en las capas finales, que resultarán más determinantes y diferenciarán rasgos concretos y específicos de nuestro dataset.

Para este experimento se ha partido del archivo de pesos pre-entrenados que utilizábamos en las pruebas de inferencia, `yolov3-tiny.weights`, como se recomienda en el repositorio original de YOLO, y se ha generado un archivo de pesos de partida mediante el siguiente comando:

```
  ./darknet partial training/yolov3-tiny-obj.cfg training/yolov3-tiny.weights
yolov3-tiny.conv.15 15
```

Donde los dos últimos parámetros son, respectivamente, el nombre que vayamos a darle al archivo de pesos base, y el número de capas a "congelar" en éste. Si nos fijamos en la arquitectura de TinyYOLOv3 (Figura 3-1) podremos ver que, de hecho, las 15 primeras capas son las previas a ejecutar la capa de detección de YOLO, y son las que aquí vamos a "congelar".

Con este último paso, ya tenemos los tres archivos necesarios para poder ejecutar el entrenamiento en Darknet. Recapitulando, hemos comentado:

- El archivo de datos, con la lista de imágenes y la carpeta de *backups*.
- El archivo de configuración, modificado para el entrenamiento y, en concreto, para uno ligero en memoria y que minimice el tiempo de ejecución.
- El archivo de pesos de partida, basado en el pre-entrenado de TinyYOLOv3, obtenido mediante `partial`.

Para ejecutar el entrenamiento, tendríamos que llamar a Darknet con los siguientes parámetros:

```
  $> darknet detector train <archivo de datos> <archivo de configuración>
<archivo de pesos base>
```

Completando cada uno de los campos entre <…> con el archivo correspondiente. Esto iniciaría directamente el entrenamiento, y por pantalla comenzaría a verse el log generado, que informa poco a poco de cómo va progresando éste. Sin embargo, como nosotros necesitamos utilizar dicho log para otras tareas, en nuestra ejecución particular redirigiremos la salida del programa a un archivo de texto, que será el que usemos para generar gráficas mediante el script. Además, como prevemos un entrenamiento largo, no es conveniente ligar la ejecución de Darkent a la sesión de Linux en uso, ya que si hubiera cualquier problema con ésta (que se apagara la terminal o se parase la sesión SSH con la que nos conectamos a la Raspberry, por ejemplo) se detendría el

programa. Por ello, utilizaremos el comando Nohup de Linux, que permite que un proceso se ejecute sin sesión alguna y se mantenga en funcionamiento siempre y cuando la máquina esté encendida.

La ejecución final se llevaría a cabo de la siguiente manera, con los parámetros concretos para este experimento:

```
$> nohup darknet detector train training/cat-dog.data training/yolov3-tiny-obj.cfg training/yolov3-tiny.conv.15 > results.txt &
```

4.3.3 Resultados y validación

En total, se han llevado a cabo 30 pruebas de entrenamiento sobre el mismo dataset, modificando entre ellas los hiperparámetros utilizados, y llegándose a probar en los dos repositorios que en apartados anteriores mencionábamos, digitalbrain79 y el personalizado, basado en shizukachan. En el Anexo F se adjunta una tabla-histórico de pruebas realizadas, con toda la información respecto al repositorio en el que se ejecutaron, los hiperparámetros utilizados, y las anotaciones tomadas como conclusión o detalles de la prueba.

De estas 30 pruebas, 26 no progresaron debido a fallos de Darknet, como se ha recogido en la tabla. En concreto, dos eran los fallos más recurrentes que obteníamos:

- *Segmentation Fault*: Es un error general de C que indica principalmente que un programa intenta acceder a localizaciones de memoria en las que carece de permiso. En el caso de Darknet, este error puede estar relacionado con el concepto conocido como *fuga de memoria* (*memory leak*), por el cual un programa no es capaz de gestionar su uso de memoria correctamente, generalmente no liberando la memoria que ya no necesita.
- *Realloc Error*: `realloc()` es una función de C que permite redimensionar un espacio de memoria reservado. Este error se da cuando el redimensionamiento no puede llevarse a cabo por diversas razones, por ejemplo, la falta de memoria.

Con el fin de localizar dónde estaban surgiendo estos errores, se modificó ligeramente el código fuente para que imprimiera por pantalla las diferentes funciones que se iban ejecutando, de tal manera que pudiéramos identificar exactamente en qué punto el programa fallaba. Un ejemplo de la salida de Darknet durante este proceso podría ser el siguiente, donde además, podemos observar cómo se presenta la información de una iteración (índice, error obtenido, media total del error, ratio de aprendizaje, tiempo tardado, y número de imágenes totales desde el comienzo de la ejecución):

```
540: 0.603852, 1.090611 avg loss, 0.000085 rate, 1.665769 seconds, 540 images
[DEBUG] i iteration 540
[DEBUG] Freeing data
[DEBUG] Joining thread
[DEBUG] Loading arguments
[DEBUG] Training network
[DEBUG] Starting train_network_waitkey
[DEBUG] Running xcalloc for X and Y
[DEBUG] Batches iteration 0
[DEBUG] Starting get_next_batch
[DEBUG] Exiting get_next_batch
[DEBUG] Batch image with code: 0
[DEBUG] Starting train_network_datum
[DEBUG] Starting forward_network
[DEBUG] Exiting forward_network
[DEBUG] Starting backward_network
[DEBUG] Exiting backward_network
[DEBUG] Starting update_network
[DEBUG] Exiting update_network
[DEBUG] Exiting train_network_datum
[DEBUG] Exiting train_network_waitkey
[DEBUG] Getting current batch
```

Sin embargo, los resultados finales no fueron claros, ya que en algunas ocasiones los errores saltaban a la hora de reservar espacio para cargar nuevas imágenes mientras que en otras sucedían a la hora de crear las estructuras de datos para gestionar las capas de la red. Finalmente, como no es objetivo de esta memoria trabajar directamente con el código de Darknet, y atendiendo a que los errores estaban directamente relacionados con la gestión de la memoria, un problema que ya sabíamos que podía ocurrir dada las condiciones de la Raspberry, se decidió proceder simplificando poco a poco los hiperparámetros de la red, tratando de que fueran lo menos perjudiciales en memoria (por ejemplo, reduciendo el número de lotes y el tamaño de los volúmenes de entrada a la red), y analizando cómo iban evolucionando los resultados.

La primera prueba que se consiguió llevar a cabo sin un fallo del programa se realizó con los mínimos para cada uno de los hiperparámetros directamente involucrados; en concreto, lotes de una sola imagen, sin mini-lotes (`subdivisions = batches = 1`), y con el tamaño de imagen mínimo disponible, que es de 32x32 (para permitir que sea divisible entre 32, como exige YOLO). Ésta es una aproximación bastante pobre, ya que las imágenes del dataset usado, aunque variadas en tamaño, suelen tener uno de los lados igual a 500 píxeles, por lo que al tener que convertirlas se pierde calidad en éstas (ver Figura 4-6), como mencionábamos en las pruebas de inferencia.

Figura 4-6. Imagen original contra imagen percibida por la red al realizar la redimensión

El resultado del entrenamiento tras alrededor de 24 horas puede observarse en la gráfica a continuación, donde enfrentamos las iteraciones con el error de entrenamiento:

```
$> python3 plotter.py results/results-1.txt
```

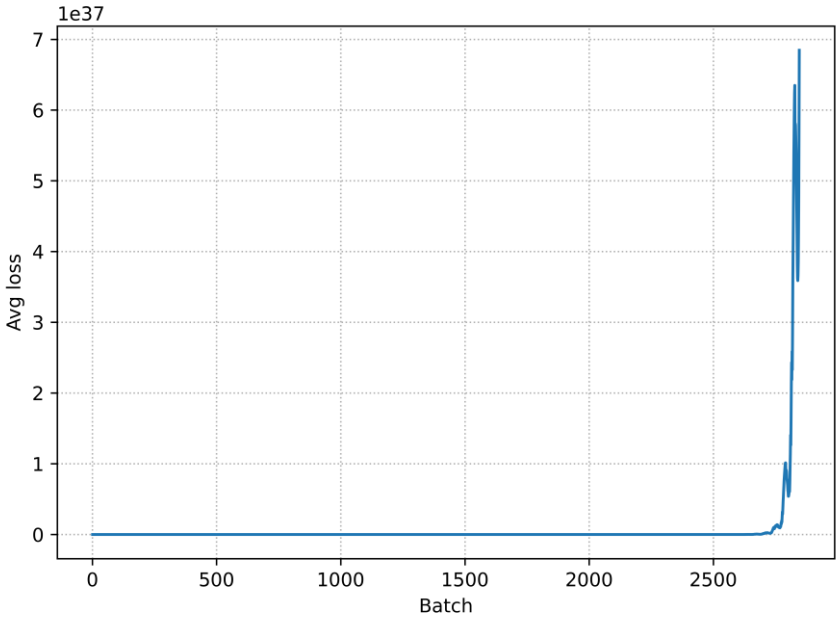


Figura 4-7. Prueba 1, evolución de las pérdidas en función de las iteraciones

El número de iteraciones totales del experimento fue de 3998, lo que, dados los hiperparámetros, supone apenas una época (recordemos que el número de imágenes del dataset es de 3700, y una época sucede cuando todas las imágenes del dataset tienen la oportunidad de actualizar los parámetros de la red). Sin embargo, como podemos observar en la gráfica, a partir de las 2700 iteraciones, el error de entrenamiento comienza a incrementarse hasta llegar a 10^{37}, momento en el cuál dejan de representarse más valores, dando la sensación de que el entrenamiento se para entonces. En la salida estándar de Darknet podríamos ver hasta donde avanzó realmente el entrenamiento:

```
3998: inf, inf avg loss, 0.000010 rate, 1.664737 seconds, 3998 images
```

Este es un comportamiento extraño, ya que no hay una razón específica que pueda causar este incremento radical del error. Si por otro lado observamos el resultado de esta prueba eliminando la parte final, obtenemos algo mucho más acorde al funcionamiento general de un entrenamiento:

```
$> python3 plotter.py results/results-1.txt -xmax 900
```

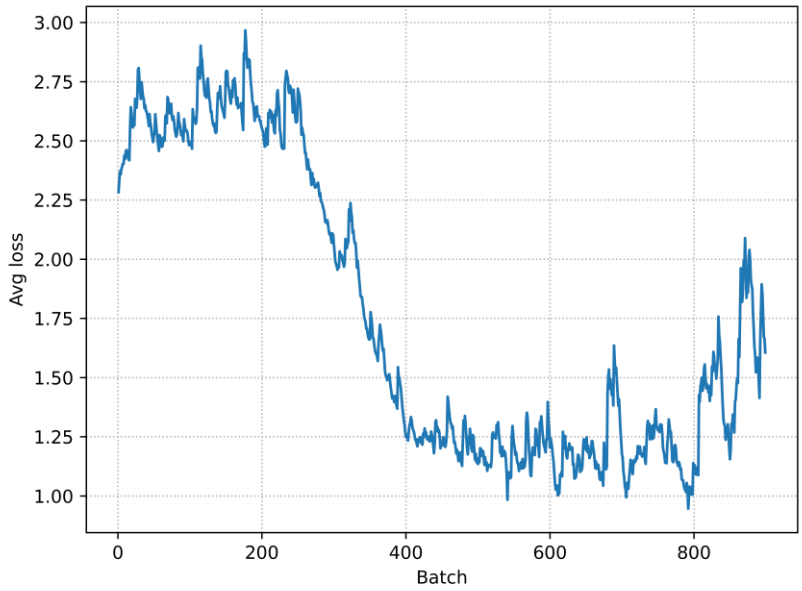


Figura 4-8. Prueba 1, evolución de las pérdidas frente a las primeras 900 iteraciones

Donde el error va reduciéndose a medida que las imágenes alimentan la red, acorde a lo que explicábamos que sucedería en capítulos anteriores. Es de notable observación, además, que, al haber configurado la red con lotes de una sola imagen, el modelo se ha vuelto un gradiente descendiente estocástico, y podemos advertir los picos y fluctuaciones tan radicales del error, cosa que señalábamos que sucedería en la Figura 2-9.

Por otro lado, las pruebas de inferencia realizadas con los pesos a las 1000 iteraciones, que es el primer *backup* que el repositorio de digitalbrain79 genera y, acorde a la gráfica, el que menos error de entrenamiento debería presentar, no da ningún resultado; es decir, no se detecta nada en ninguna de las imágenes en las que se ha probado. Pese a que no es una norma general, el hecho de que a esa altura tan solo 1000 imágenes del dataset hayan entrenado la red sumado a la configuración de los hiperparámetros (en concreto, el haber reducido dichas imágenes a 32x32), pueden ser la causa principal.

Volviendo al excepcional comportamiento de la gráfica de la Figura 4-7, se proponen las siguientes razones que podrían justificarlo:

- De forma general, podría estar causado por lo conocido como *gradientes explosivos* (*exploding gradients*). Este es un problema que sucede en redes neuronales basadas en gradientes y propagación

hacia atrás, y se caracteriza porque los pesos son actualizados de forma radical, con grandes incrementos o decrementos, provocando que el error de entrenamiento se altere hasta resultar en valores indefinidos, que serían lo que en la gráfica estamos interpretando como infinito. Aunque la definición estricta es más extensa, el efecto provocado es similar al que estamos obteniendo. Entre las posibles soluciones, destaca rediseñar el modelo de la red; utilizar regularización de pesos, penalizando e impidiendo que éstos se actualicen de forma drástica; o aplicar *gradient clipping* (*recorte de gradiente*), que es similar a la regularización de pesos pero limita las variaciones grandes del gradiente.

- De forma específica, el problema puede estar directamente relacionado con la implementación de Darknet o con la red de TinyYOLOv3. A su vez, el dataset, la configuración de hiperparámetros o los pesos de partida que utilizamos podrían estar causándolo.
- Finalmente, podría estar relacionado, de alguna forma, con los problemas de memoria que también causaban los errores comentados al comienzo de esta sección.

En las sucesivas pruebas que fueron realizadas sin errores de memoria y de las que se pudo obtener gráficas se observa también este comportamiento, como veremos a continuación. De igual manera, ninguno de los pesos de *backup* generados en las mejores situaciones de pérdidas de entrenamiento dieron resultados positivos en la inferencia. Aun así, estas pruebas han servido para analizar poco a poco el comportamiento de la red a medida que se iban mejorando los hiperparámetros, y con ello estudiar la evolución del error de entrenamiento durante las primeras iteraciones y épocas (antes de que las pérdidas se hicieran infinitas) y medir la duración de estos entrenamientos.

Con estos objetivos en mente, la segunda prueba realizada sobre la distribución de digitalbrain79 se llevó a cabo con un número de imágenes por lote igual a 64, que es el valor recomendado para el entrenamiento de TinyYOLOv3. Manteniendo el número de subdivisiones a 1, convertíamos el modelo en un gradiente descendente por lotes, y esto se ve reflejado en la gráfica, que es más "suave" que la que representábamos anteriormente:

```
$> python3 plotter.py results/results-2.txt
$> python3 plotter.py results/results-2.txt -xmax
```

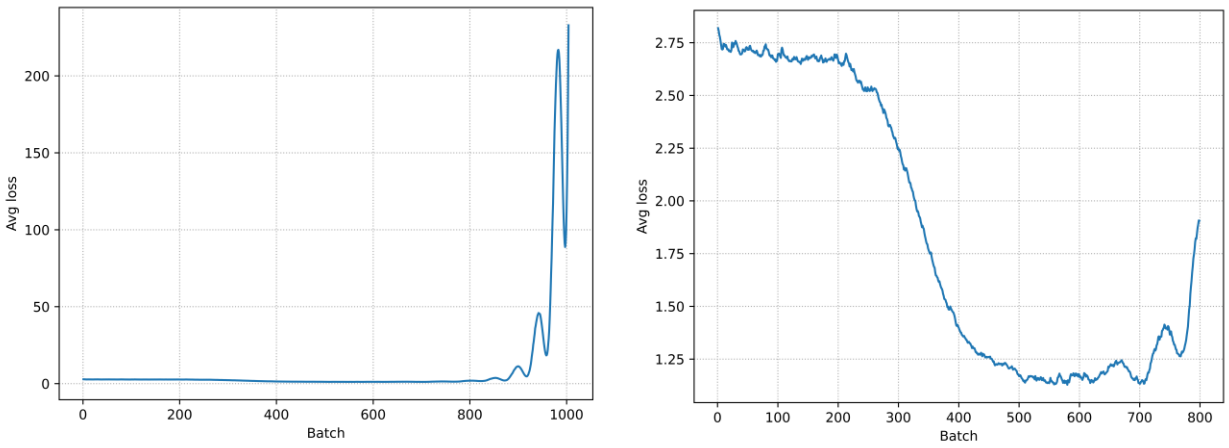


Figura 4-9. Prueba 2, evolución de las pérdidas frente a las iteraciones (total y primeras 800)

Esta prueba se llevó a cabo durante 1004 iteraciones, lo que suponen 17 épocas (representadas a continuación) y tomó alrededor de 26 horas. Podemos observar que obtenemos unos valores mínimos de pérdidas inferiores a 1'25, similares a los que obteníamos en la prueba anterior, en una cantidad de tiempo también similar, pero en este caso habiendo utilizado muchas más imágenes para entrenar la red (unas 51200).

```
$> python3 plotter.py results/results-2.txt -x epoch
$> python3 plotter.py results/results-2.txt -x epoch -xmax
```

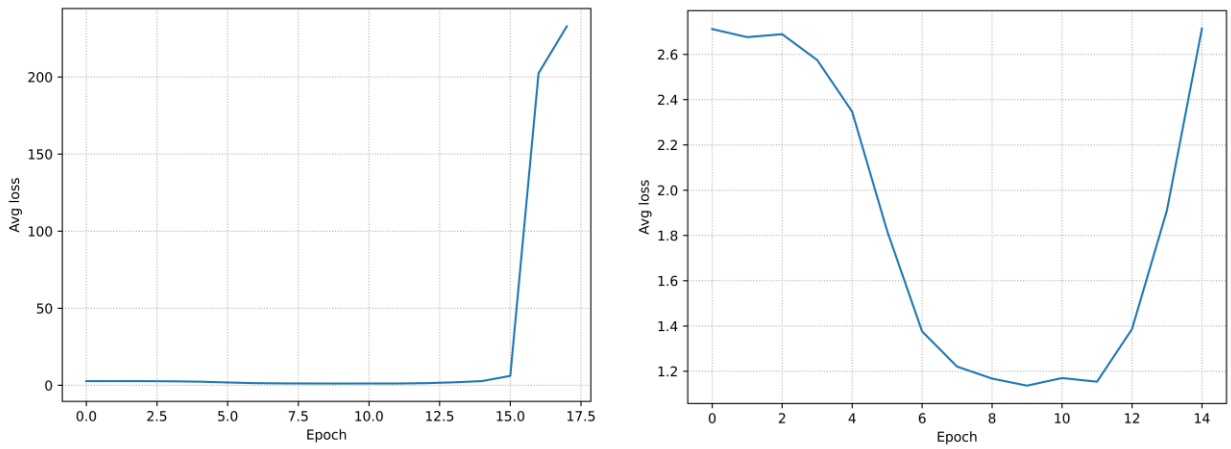


Figura 4-10. Prueba 2, evolución de las pérdidas frente a las épocas (total y primeras 14)

Para la tercera prueba se cambió de distribución, de digitalbrain79 a shizukachan (versión personalizada), principalmente para comprobar si el entrenamiento funcionaba de diferente manera y si daba mejores resultados. Una de las principales ventajas es que la versión de shizukachan crea los pesos de *backup* cada 100 iteraciones, lo que daba más oportunidades para probar si en algún momento la red estaba lo suficientemente entrenada como para probarla en inferencia.

En vista de que los modelos estocástico y por lotes se habían podido llevar a cabo, y siendo este último especialmente importante porque suponía una buena gestión de la memoria, para las siguientes pruebas se comenzó a utilizar el sistema por mini-lotes, que como ya se explicó, es más recomendable. En concreto, se crearon mini-lotes de 4 imágenes (tamaño de lote igual a 64, 16 subdivisiones), que es lo recomendado para TinyYOLOv3. De igual manera que con las pruebas anteriores, podemos observar que la gráfica corresponde a lo previsto para este tipo de gradiente: un punto media entre la de SGD y la de lotes.

```
$> python3 plotter.py results/results-3.txt
$> python3 plotter.py results/results-3.txt -xmax
```

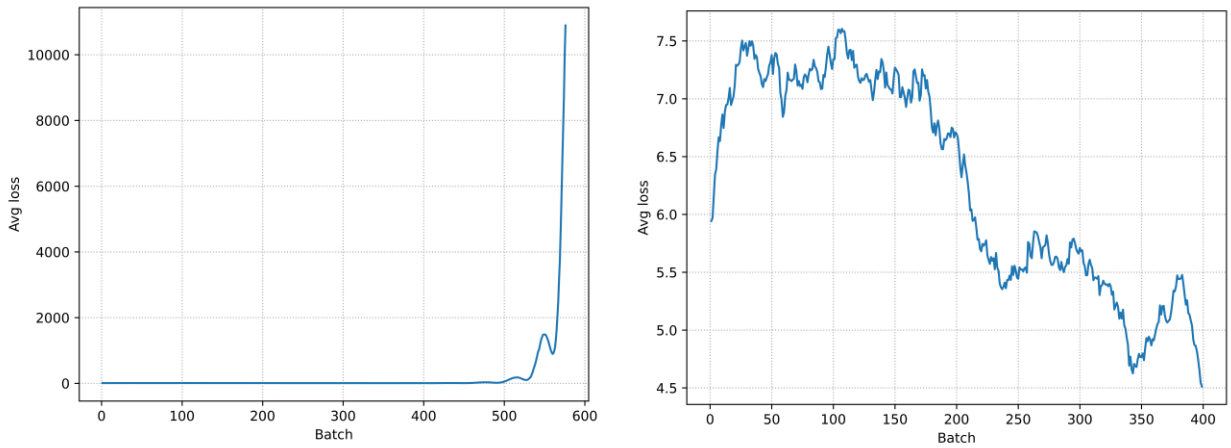


Figura 4-11. Prueba 3, evolución de las pérdidas frente a las iteraciones (total y primeras 400)

```
$> python3 plotter.py results/results-3.txt -x epoch
$> python3 plotter.py results/results-3.txt -x epoch -xmax
```

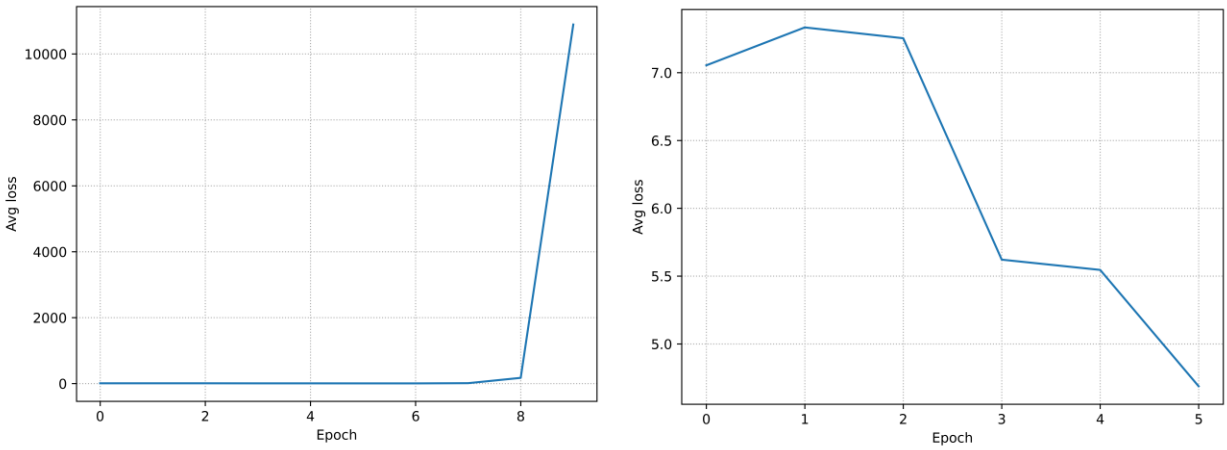


Figura 4-12. Prueba 3, evolución de las pérdidas frente a las épocas (total y primeras 5)

En este caso, la prueba se llevo a cabo durante 970 iteraciones (cerca de 17 épocas), con un error de entrenamiento mínimo de 4'5 y una duración de 35 horas.

La última prueba llevada a cabo, también sobre la distribución de shizukachan para poder probar los pesos de *backup* con más frecuencia, se realizó con la misma configuración de lotes y mini-lotes, pero aumentando el tamaño de las imágenes. Hasta ahora, éste se había mantenido en 32x32, lo cuál hacía que los resultados obtenibles no fueran especialmente alentadores debido a la redimensión de las imágenes de entrada, pero a su vez sobrecargaban lo mínimo la memoria. Para esta prueba, se aumentó hasta 128x128, el cuádruple, aunque seguía suponiendo un 70% del tamaño original de las imágenes. Aun así, el objetivo seguía siendo no causar fallos en memoria. Los resultados fueron:

```
$> python3 plotter.py results/results-4.txt
$> python3 plotter.py results/results-4.txt -xmax
```

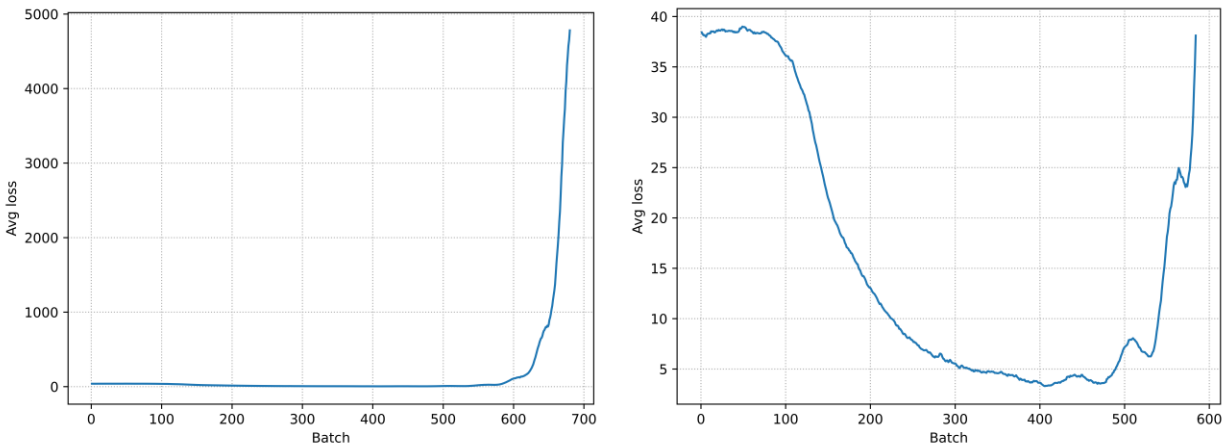


Figura 4-13. Prueba 4, evolución de las pérdidas frente a las iteraciones (total y primeras 600)

```
$> python3 plotter.py results/results-4.txt -x epoch
$> python3 plotter.py results/results-4.txt -x epoch -xmax
```

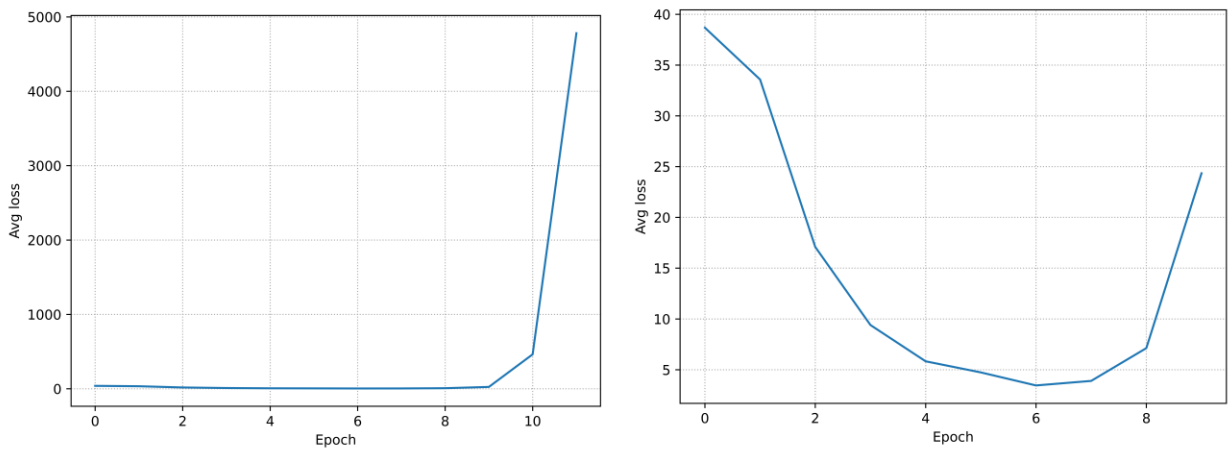


Figura 4-14. Prueba 4, evolución de las pérdidas frente a las épocas (total y primeras 9)

Podemos observar que la curva sigue un curso muy similar al de la gráfica anterior, decreciendo poco a poco hasta un mínimo cercano a 4'5. Las fluctuaciones del error debido al modelo por mini-lotes son menos apreciables, probablemente debido a que en este caso comenzamos con un error mucho mayor. En general, antes de las 6/7 épocas, podríamos decir que estamos ante una evolución favorable del error. Sin embargo, esta prueba se extendió durante **75 horas**, comenzando a producirse el incremento del error hasta infinito a las 50. Es de suponer que, si seguimos aumentando el tamaño de las imágenes de entrada, este tiempo se seguiría viendo incrementado, quizá obteniendo mejores resultados, pero no haciendo viable el entrenamiento. Si bien los entrenamientos de redes neuronales pueden llegar a extenderse semanas incluso en condiciones favorables (sobre dispositivos espcializados, por ejemplo), las condiciones de estos entrenamientos suelen ser más realistas de las que lo han sido para este experimento (dataset de pocas imágenes y clases, volúmenes de entrada ajustados…); por lo que en este punto se decidió cesar con las pruebas. Futuros acercamientos podrían haber tratado de:

- Seguir aumentando el tamaño de las imágenes, lo que habría incrementado a su vez la duración del entrenamiento.
- Cambiar el archivo de pesos base, o "congelar" menos capas, por ejemplo, 11 en vez de 15, permitiéndose así la especialización de algunas capas previas a la de detección de YOLO.
- Cambiar de dataset, por ejemplo, por uno donde los objetos a detectar estuvieran más claros, o dichos objetos fueran más simples, quizá incluso tratando de detectar una sola clase.
- Probar la ejecución con validación, aunque habría que comprobar previamente si es factible sin fallos de memoria.

A modo de resumen de todo el experimento, se adjunta la siguiente tabla, donde se han indicado cada uno de los objetivos que se marcaban al comienzo de la sección, con sus resultados y observaciones:

Tabla 4-6. Resumen de objetivos y resultados del entrenamiento

Objetivo	Resultado	Observaciones
Viabilidad – Sin fallos en la ejecución del programa	Satisfactorio	Se ha conseguido que no haya fallos de memoria durante, al menos, 4 pruebas
Duración y calidad	Insatisfactorio	Ninguna de las 4 pruebas realizadas han conseguido entrenar la red; además, los tiempos medios de entrenamiento, pese a ello, superan en general las 24 horas.
Utilidad	Insatisfactorio	La duración de los entrenamientos en condiciones favorables (tamaño de imagen reducido, dataset de dos clases y 4000 imágenes) llega hasta las 75 horas sin resultados favorables, por lo que se desaconseja usar Raspberry para entrenamiento

5 CONCLUSIONES Y LÍNEAS FUTURAS

En este capítulo se tratarán las conclusiones globales del proyecto, relacionadas con los objetivos que se proponían al comienzo de esta memoria; y se comentarán brevemente las posibles mejoras y alternativas que podrían haberse realizado o podrían realizarse en un futuro manteniendo las pautas de este proyecto.

TinyYOLOv3 ha demostrado ser bastante eficiente en Raspberry en la detección de objetos en imágenes, aunque la velocidad de inferencia no acompañe para largos conjuntos o vídeos. Por otro lado, las dificultades a la hora de entrenar suponen que hay que usar una red ya entrenada, o entrenar la red en un dispositivo más preparado. En conclusión, se recomienda usar TinyYOLOv3 en Raspberry para:

- Inferencia en imágenes y conjuntos de imágenes (si el tiempo total / FPS no es relevante) en los que puedan utilizarse redes ya entrenadas.

Y no se recomienda usarlo para:

- Entrenamiento, en general.

Quedaría por resolver, realizando pruebas específicas, los siguientes puntos:

- Inferencia en vídeo o en *streaming* (vídeo en directo). En principio, los resultados en conjuntos de imágenes no animan a pensar que pueda llevarse a cabo con resultados favorables (menos de 1 FPS), pero podría realizarse la prueba. Para ello, habría que instalar un componente más, OpenCV, necesario para ver el resultado de las predicciones a medida que se van aplicando sobre el vídeo.

A modo de resumen, en la siguiente tabla se muestran los resultados y comentarios respecto a los objetivos que planteábamos al comienzo de la memoria:

Tabla 5-1. Observaciones de los objetivos del proyecto

Objetivo	Resultado	Observaciones
Viabilidad de la instalación de TinyYOLOv3 en Raspberry Pi	Satisfactorio	Las distribuciones de Darknet con NNPACK que se han probado se instalan correctamente y funcionan sin problemas con TinyYOLOv3.
Comprobar funcionamiento de la inferencia y medir sus prestaciones	Parcialmente satisfactorio	La inferencia se puede realizar sin fallos y el resultado de las predicciones es bastante preciso, aunque los tiempos de inferencia por imagen son demasiado altos y podrían suponer un problema a la hora de detectar objetos en vídeos.
Comprobar funcionamiento del entrenamiento y medir sus prestaciones	Insatisfactorio	Las limitaciones de memoria y capacidad de la Raspberry impiden realizar en muchas ocasiones los entrenamientos completos. De los que se han conseguido llevar a cabo, para el dataset utilizado, se ha tenido que adaptar la configuración tanto que no se han obtenido resultados fructíferos, a lo que se le suma que la duración de los entrenamientos llegaba a sobrepasar las 24 horas en varias ocasiones sin llegar a superar apenas las 15 épocas en un dataset de 4000 imágenes.

5.1 Líneas futuras

Respecto a experimentos que podrían realizarse bajo las mismas condiciones o con modificaciones mínimas, se recogen:

- Diseñar un programa en Python que permitiera calcular las métricas para valorar la predicción de forma automática, basándose en los resultados obtenidos durante la inferencia y los cuadros delimitadores de las imágenes originales.
- Llevar a cabo más pruebas de entrenamiento, tal y como se comentan al final de 4.3.3. El objetivo sería, principalmente, cambiar el dataset, buscando uno con otro tipo de contenido (quizá de tan sólo una clase) y cuyas imágenes fuesen de un tamaño original lo más bajo posible; y seguir probando con distintas combinaciones de tamaños de entrada y división de lotes y mini-lotes.
- Llevar a cabo las pruebas de entrenamiento aplicando validación. Las imágenes de validación bien pueden ser las mismas que se usen para las pruebas de inferencia posteriores, siempre y cuando estén anotadas. En Darknet, se pueden activar usando la opción `-map` en la ejecución estándar del comando de entrenamiento.
- Realizar las pruebas sobre vídeos. Esto requiere la instalación de OpenCV, una librería de funciones desarrollada específicamente para visión artificial. Con ésta instalada, Darknet puede llevar a cabo una inferencia sobre vídeo y mostrar por pantalla cómo se van llevando a cabo las predicciones en tiempo real.

En cuanto a mejoras sustanciales del proyecto, se propone:

- Utilizar el modelo Raspberry Pi 4 en lugar del 3. Este modelo se lanzó a mediados de 2019 y comenzó a venderse al público a finales del año. Entre sus mejoras principales, se encuentra una CPU más potente (4xARM Cortex A72 64-bit 1.5GHz, lo que supera incluso a la CPU de la Nvidia Jetson) y una memoria SDRAM de hasta 8GB, por los 1GB LPDDR del modelo 3. Ambas características deberían garantizar un mejor rendimiento y menos problemas en memoria, lo que mejoraría radicalmente tanto los resultados de las pruebas de inferencia como los de entrenamiento. Su precio actual oscila los 100 euros para la versión de 8GB de memoria, y 60€ para la de 4GB.

REFERENCIAS

[1] V. Roblek, M. Meško y A. Krapež, «A complex view of industry 4.0,» *Sage Open,* vol. 6, nº 2, p. 2158244016653987, 2016.

[2] M. Sonka, V. Hlavac y R. Boyle, Image Processing, Analysis, and Machine Vision, Cengage Learning, 2014.

[3] S. Ibrahim, «A comprehensive review on intelligent surveillance systems,» *Communications in Science and Technology,* 2016.

[4] M. Daily, S. Medasani, R. Behringer y M. Trivedi, «Self-driving cars,» *Computer,* vol. 50, nº 12, pp. 18-22, 2017.

[5] B. Chance y Z. Zhao, «Simple ac circuit for breast cancer detection and object detection,» *Review of scientific instruments,* vol. 77, nº 6, p. 064301, 2006.

[6] I. D. Apostolopoulos y T. A. Mpesiana, «Covid-19: automatic detection from x-ray images utilizing transfer learning with convolutional neural networks,» Springer, 2020.

[7] S. Dasiopoulou, V. Mezaris, I. Kompatsiaris, V. Papastathis y M. G. Strintzis, «Knowledge-assisted semantic video object detection,» *IEEE Transactions on Circuits and Systems for Video Technology,* vol. 15, nº 10, pp. 1210-1224, 2005.

[8] J. Osborne, «TechRadar,» 22 Agosto 2016. [En línea]. Available: https://www.techradar.com/news/computing-components/processors/google-s-tensor-processing-unit-explained-this-is-what-the-future-of-computing-looks-like-1326915. [Último acceso: 26 Agosto 2020].

[9] S. Markidis, S. Wie Der Chien, E. Laure, I. Bo Peng y J. S. Vetter, «Nvidia tensor core programmability, performance & precision,» de *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, IEEE, 2018, pp. 522-531.

[10] D. Poole, A. Mackworth y R. Goebel, Computational Intelligence: A Logical Approach, Nueva York: Oxford University Press, 1998.

[11] B. G. Buchanan, «A (Very) Brief History of Artificial Intelligence,» *AI Magazine,* vol. 26, nº 4, pp. 53-53, 2005.

[12] Oxford University Press, OED Online, Oxford University Press, 2020.

[13] F. Rojas Velásquez, «Enfoque sobre el aprendizaje humano,» *Departamento de Ciencia y Tecnología del Comportamiento. Universidad Simón Boívar,* 2001.

[14] D. Shlegel, «Deep Machine Learning on GPU,» *University of Heidelber-Ziti,* vol. 12, 2015.

[15] M. van Gerven y S. Bohte, «Artificial neural networks as models of neural information processing,»

Frontiers in Computational Neuroscience, vol. 11, p. 114, 2017.

[16] D. C. Ciresan, U. Meier, J. Masci, L. M. Gambardella y J. Schmidhuber, «Flexible, high performance convolutional neural networks for image classification,» de *Twenty-second international joint conference on artificial intelligence*, 2011.

[17] C. Van Der Malsburg, «Frank Rosenblatt: principles of neurodynamics: perceptrons and the theory of brain mechanisms,» de *Brain theory*, Springer, 1986, pp. 245-248.

[18] K. Jarrett, K. Kavukcuoglu, M. Ranzato y Y. LeCun, «What is the best multi-stage architecture for object recognition?,» de *2009 IEEE 12th International Conference on Computer Vision*, 2009, pp. 2146-2153.

[19] D.-X. Zhou, «Universality of deep convolutional neural networks,» *Applied and computational harmonic analysis,* vol. 48, nº 2, pp. 787-794, 2020.

[20] H. Barrow, «Connectionism and Neural Networks,» de *Artificial intelligence*, M. A. Boden, Ed., San Diego, Academic Press, 1996, pp. 135-155.

[21] J. Brownlee, «What is the Difference Between a Batch and an Epoch in a Neural Network?,» *Machine Learning Mastery,* 2018.

[22] S. Ruder, «An overview of gradient descent optimization algorithms,» *arXiv preprint arXiv:1609.04747,* 2016.

[23] M. D. Zeiler, «Adadelta: an adaptive learning rate method,» *arXiv preprint arXiv:1212.5701,* 2012.

[24] J. Redmon, S. Divvala, R. Girshick y A. Farhadi, «You Only Look Once: Unified, real-time object detection,» de *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 779-788.

[25] U. Michelucci, «Feedforward Neural Networks,» de *Applied Deep Learning: A Case-Based Approach to Understanding Deep Neural Networks*, Apress, 2018, pp. 83-136.

[26] J. Brownlee, «What is the Difference Between a Parameter and a Hyperparameter,» 2018.

[27] J. Pearl, Heuristics: intelligent search strategies for computer problem solving, Addison-Wesley, 1984.

[28] Y. Bengio, A. Courville y V. Pascal, «Representation learning: A review and new perspectives,» *IEEE transactions on pattern analysis and machine intelligence,* vol. 35, nº 8, pp. 1798-1828, 2013.

[29] L. Deng y D. Yu, «Deep Learning: Methods and Applications,» *Microsoft,* 2014.

[30] A. Krizhevsky, I. Sutskever y G. E. Hinton, «ImageNet Classification with Deep Convolutional Neural Networks,» de *Advances in Neural Information Processing Systems 25*, Curran Associates, Inc., 2012, pp. 1097-1105.

[31] R. Girshick, J. Donahue, T. Darrell y J. Malik, «Rich feature hierarchies for accurate object detection and semantic segmentation,» de *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2014, pp. 580-587.

[32] J. R. Uijlings, K. E. Van De Sande, T. Gevers y A. W. Smeulders, «Selective search for object recognition,» *International Journal of Computer Vision,* vol. 104, nº 2, pp. 154-171, 2013.

[33] C. Cortes y V. Vapnik, «Support-vector networks,» *Machine learning,* vol. 20, nº 3, pp. 273-297, 1995.

[34] R. Girshick, «Fast R-CNN,» de *Proceedings of the IEEE international conference on computer vision*, 2015, pp. 1440-1448.

[35] H. Rezatofighi, N. Tsoi, J. Gwak, A. Sadeghian, I. Reid y S. Savarese, «Generalized intersection over union: A metric and a loss for bounding box regression,» de *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2019, pp. 658-666.

[36] S. Ren, K. He, R. Girshick y J. Sun, «Faster r-cnn: Towards real-time object detection with region proposal networks,» de *Advances in neural information processing systems*, 91-99, p. 2015.

[37] J. Redmon, S. Divvala, R. Girshick y A. Farhadi, «You only look once: Unified, real-time object detection,» de *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 779-788.

[38] J. Redmon y A. Farhadi, «YOLO9000: better, faster, stronger,» de *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 7263-7271.

[39] A. Farhadi y J. Redmon, «YOLOv3: An incremental improvement,» *Retrieved September,* vol. 17, p. 2018, 2018.

[40] A. Bochkovskiy, C.-Y. Wang y H.-Y. M. Liao, «YOLOv4: Optimal Speed and Accuracy of Object Detection,» *arXiv preprint arXiv:2004.10934,* 2020.

[41] S. Ioffe y C. Szegedy, «Batch normalization: Accelerating deep network training by reducing internal covariate shift,» *arXiv preprint arXiv:1502.03167,* 2015.

[42] T. Baji, «Evolution of the GPU Device widely used in AI and Massive Parallel Processing,» de *2018 IEEE 2nd Electron Devices Technology and Manufacturing Conference (EDTM)*, IEEE, 2018, pp. 7-9.

[43] J. D. Owens, H. Houston, D. Luebke, S. Green, J. E. Stone y J. C. Phillips, «GPU computing,» *Proceedings of the IEEE,* vol. 96, nº 5, pp. 879-899, 2008.

[44] F. Abi-Chahla, «Nvidia's CUDA: The End of the CPU?,» *Tom's Hardware,* pp. 1954-7, 2008.

[45] M. Barr y A. Massa, «Introduction,» de *Programming embedded systems: with C and GNU development tools*, O'Reilly Media, Inc., 2006, pp. 1-2.

[46] M. Barr, «Real men program in C,» *Embedded systems design,* vol. 22, nº 7, p. 3, 2009.

[47] A. L. Shimpi, «ARM's Cortex A57 and Cortex A53: The First 64-bit ARMv8 CPU Cores,» *AnandTech. Np,* vol. 30, 2012.

[48] P. Warden y D. Situnayake, Tinyml: Machine learning with tensorflow lite on arduino and ultra-low-power microcontrollers, O'Reilly Media, Incorporated, 2020.

[49] H. He, C.-W. Huang, L. Wei, L. Li y G. Anfu, «TF-YOLO: An Improved Incremental Network for Real-Time Object Detection,» *Applied Sciences,* vol. 9, nº 1, p. 3225, 2019.

[50] D. R. Musicant, V. Kumar y A. Ozgur, «Optimizing F-Measure with Support Vector Machines,» de

FLAIRS conference, 2003, pp. 356-360.

[51] M. J. Park y B. Chul Ko, «Two-Step Real-Time Night-Time Fire Detection in an Urban Environment Using Static ELASTIC-YOLOv3 and Temporal Fire-Tube,» *Sensors,* vol. 20, p. 2202, 4 2020.

ANEXO A: PREPARACIÓN RASPBERRY PI

Se listan y detallan a continuación los pasos a seguir para instalar y preparar una Raspberry Pi 3 B tal y como la que se ha utilizado en este proyecto.

En primer lugar, se requerirá instalar el sistema operativo necesario, que será un Raspberry Pi OS Lite, en una tarjeta SD (se recomienda que sea de 32GB, para evitar problemas de memoria). Para ello, se propone descargar el instalador oficial de Raspberry, *Raspberry Pi Imager* desde la página web oficial (https://www.raspberrypi.org/downloads/), y a continuación seguir las instrucciones facilitadas en https://www.raspberrypi.org/blog/raspberry-pi-imager-imaging-utility/. El SO necesario estará en el menú de selección de sistema operativo, dentro del grupo *Raspberry Pi OS (other)*.

Una vez descargado, la tarjeta SD se introducirá en la Raspberry y está deberá ser conectada a una fuente de alimentación y encendida. Para la configuración inicial del sistema operativo será recomienda disponer de un monitor y un teclado, que deberán ser conectados a la Raspberry. En líneas generales, manejaremos la Raspberry mediante una conexión SSH, pero para la puesta en marcha no tendremos acceso por esa vía, por lo que será necesario acceder a ella directamente.

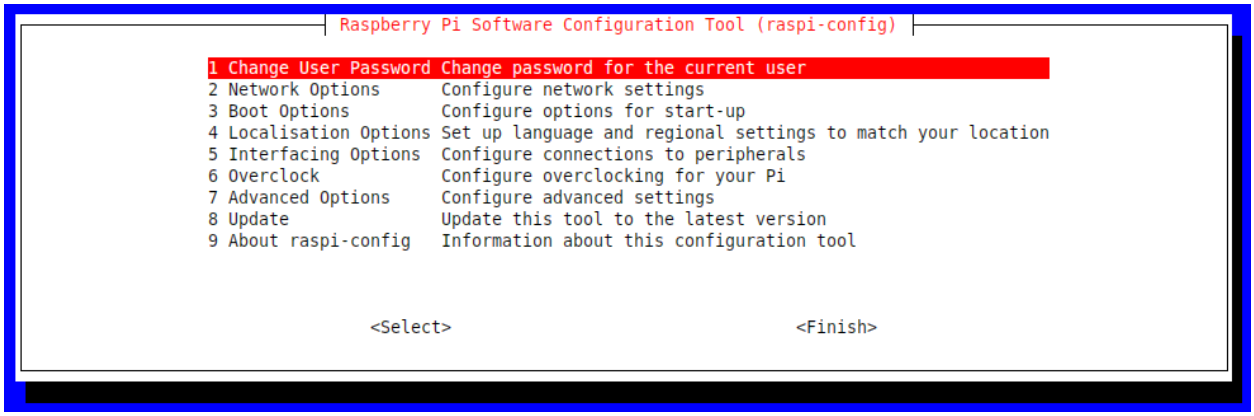
Una vez encendida, tendremos que esperar a que nos pida el inicio de sesión, con la siguiente línea:

```
raspberrypi login:
```

Por defecto, el usuario es *pi* y la contraseña (que la pedirá a continuación) *raspberry*. Una vez introducidos ambos, podríamos comenzar a prepararla. Como estamos usando una versión ligera del sistema operativo (para optimizar al máximo el consumo de memoria), tendremos que configurarla manualmente. Para ello, comenzaremos usando el comando `raspi-config`, con el que ajustaremos el país y el teclado y nos conectaremos a internet. Es posible que debido a la configuración del teclado por defecto en inglés, el guion simple (-) no esté en la tecla correspondiente, sino en la de cierre de interrogación (?).

```
$> sudo raspi-config
```

Se nos abrirá un menú por el que podremos navegar con las flechas de teclado y seleccionar pulsando la tecla Enter.



Lo primero que convendría hacer es cambiar las opciones de localización (`Localisation Options`, menú 4). En este menú podemos tanto cambiar el idioma como la configuración del teclado; nos centraremos en esta última. Presionamos Enter para acceder al menú de localización, y luego navegamos hasta la opción `Change Keyboard Layout`. Nos trasladará a la configuración del teclado. En la primera ventana mantenemos la opción que aparezca por defecto (debería detectar el teclado que estemos utilizando), y pulsamos `<OK>`, que debería aparecer en la ventana abajo a la derecha. En la siguiente ventana nos aparecerá el idioma del teclado, que por

defecto debería ser '`English (UK)`'. Navegamos hasta `Other`, que nos abrirá otro menú en el que tendremos que seleccionar `Spanish`. Volveremos al menú anterior, ahora para seleccionar la región concreta dentro del español. Escogeremos la primera de la lista, que debería ser '`Spanish`'. El resto de menús que nos aparecerán los podemos dejar con sus valores por defecto ('`The default for the keyboard layout`' primero; '`No compose key`' después). Una vez hecho esto, el teclado estará correctamente configurado.

El siguiente paso es conectarnos a una red. Aquí explicaremos cómo conectarnos a una inalámbrica, como puede ser Wi-Fi. Buscaremos el menú `Network Options` → `Wireless LAN`, y seleccionaremos el país correspondiente, que debería ser '`ES Spain`'. Nos pedirá a continuación un SSID, que no es sino el nombre de la red Wi-Fi a la que nos queremos conectar. Lo introducimos y nos pedirá la clave. No nos dará ningún mensaje informativo de si la conexión ha ido bien, así que tendremos que confiar en que sí. Si no fuera así (lo descubriríamos en pasos posteriores), bastaría con cargar `raspi-config` otra vez y repetir estos pasos.

Una vez conectado a una red, configuraremos la Raspberry para que siempre que se enciende active una interfaz SSH por la que poder conectarnos a través de cualquier terminal. Navegamos hasta el menú `Interfacing Options` → `P2 SSH` y pulsamos `<Yes>` en el mensaje de advertencia.

Finalmente, salimos de `raspi-config` navegando hasta la opción `<Finish>`. Nos pedirá reiniciar, a lo que le diremos que sí. Tras ello, ejecutamos el siguiente comando, destinado a actualizar paquetes y librerías.

```
sudo apt-get upgrade && sudo apt-get update
```

Si todo funciona sin fallos, tendremos la Raspberry lista para proceder a la instalación de Darknet y YOLO. Podemos desconectar ahora el monitor y el teclado, y conectarnos mediante SSH desde algún programa (tendremos que conocer la IP de la máquina, para lo que podemos ejecutar el comando `ip a` y buscar el valor de `inet` en la entrada `wlan0`).

En este proyecto, todas las conexiones han sido realizadas desde los programas WinSCP y PuTTY en Windows, con los que podemos navegar entre archivos (WinSCP), y ejecutar comandos y programas desde una terminal (PuTTY). Ambos utilizan SSH para llevar a cabo sus conexiones.

ANEXO B: ARCHIVO DE CONFIGURACIÓN DE TINYYOLOV3

Utilizado por Darknet para conocer la arquitectura de TinyYOLOv3. Consta de una sección de configuración de la red, y 23 secciones de capas.

```
[net]
# Testing
batch=1
subdivisions=1
# Training
# batch=64
# subdivisions=16
width=128
height=128
channels=3
momentum=0.9
decay=0.0005
angle=0
saturation = 1.5
exposure = 1.5
hue=.1
learning_rate=0.001
burn_in=1000
max_batches = 500200
policy=steps
steps=400000,450000
scales=.1,.1

[convolutional]
batch_normalize=1
filters=16
size=3
stride=1
pad=1
activation=leaky

[maxpool]
size=2
stride=2

[convolutional]
batch_normalize=1
filters=32
size=3
stride=1
pad=1
activation=leaky

[maxpool]
size=2
stride=2

[convolutional]
batch_normalize=1
filters=64
size=3
```

```
stride=1
pad=1
activation=leaky

[maxpool]
size=2
stride=2

[convolutional]
batch_normalize=1
filters=128
size=3
stride=1
pad=1
activation=leaky

[maxpool]
size=2
stride=2

[convolutional]
batch_normalize=1
filters=256
size=3
stride=1
pad=1
activation=leaky

[maxpool]
size=2
stride=2

[convolutional]
batch_normalize=1
filters=512
size=3
stride=1
pad=1
activation=leaky

[maxpool]
size=2
stride=1

[convolutional]
batch_normalize=1
filters=1024
size=3
stride=1
pad=1
activation=leaky

###########

[convolutional]
batch_normalize=1
filters=256
size=1
stride=1
pad=1
activation=leaky

[convolutional]
```

```
batch_normalize=1
filters=512
size=3
stride=1
pad=1
activation=leaky

[convolutional]
size=1
stride=1
pad=1
filters=255
activation=linear

[yolo]
mask = 3,4,5
anchors = 10,14,  23,27,  37,58,  81,82,  135,169,  344,319
classes=80
num=6
jitter=.3
ignore_thresh = .7
truth_thresh = 1
random=1

[route]
layers = -4

[convolutional]
batch_normalize=1
filters=128
size=1
stride=1
pad=1
activation=leaky

[upsample]
stride=2

[route]
layers = -1, 8

[convolutional]
batch_normalize=1
filters=256
size=3
stride=1
pad=1
activation=leaky

[convolutional]
size=1
stride=1
pad=1
filters=255
activation=linear

[yolo]
mask = 0,1,2
anchors = 10,14,  23,27,  37,58,  81,82,  135,169,  344,319
classes=80
num=6
jitter=.3
```

```
ignore_thresh = .7
truth_thresh = 1
random=1
```

ANEXO C: SCRIPT DE DETECCIÓN RÁPIDA

Se detalla aquí el código de `fast_detect.sh`, escrito en Bash, utilizado para ejecutar una inferencia sobre una imagen con la configuración estándar de Darknet (configuración y datos de COCO Dataset) y TinyYOLOv3; como se menciona en la subsección 4.1.2.

https://github.com/angmorpri/darknet-tfg/blob/master/fast_detect.sh

```
#!/bin/bash
# Fast darknet detect using Tiny YOLOv3 configuration and weights
# Receives one parameter that must be an image from ./testing_images

# Checking correct number of arguments
[[ $# -ne 1 ]] && { echo "Use: fast_detect.sh <image>"; exit 2; }

# Checking given image exists and running darknet
IMAGE="$1"
if [ -f "$IMAGE" ]; then
     ./darknet detect cfg/yolov3-tiny.cfg yolov3-tiny.weights "$IMAGE"
else
     IMAGE="testing/$IMAGE"
     if [ -f "$IMAGE" ]; then
          ./darknet detect cfg/yolov3-tiny.cfg yolov3-tiny.weights "$IMAGE"
     else
          echo "$IMAGE does not exist."
     fi
fi
```

ANEXO D: SCRIPT PARA PRUEBAS DE INFERENCIA

Se presenta el código *detect.py*, escrito en Python, para llevar a cabo las pruebas de inferencia sobre varias imágenes, como se explica en la subsección 4.1.3 y posteriormente se utiliza en la sección 4.2.

https://github.com/angmorpri/darknet-tfg/blob/master/detect.py

```
#!python3
#-*- coding: utf-8 -*-
"""
    Código para inferencia de varias imágenes con Darknet.
    Utiliza el mismo mecanismo que el código fuente de Darknet mediante un
    wrapper escrito en Python, basado en ./python/darknet.py.
    Se han tenido que añadir algunas funciones extra en C, para permitir algunas
    acciones que no podían ser envueltas. Todas estas se hallan en
    'src/py_utils.c'.

    Creado:                 09 May 2020
    Última modificación:    08 Nov 2020

    @author: Ángel Moreno Prieto

"""
import argparse
import statistics as stats
import random
import time
from ctypes import *
from os import listdir, mkdir
from os.path import isdir, isfile, join, basename
from pprint import pprint


# Definiciones necesarias para el wrapper con ctypes
class BOX (Structure):
    _fields_ = [("x", c_float),
                ("y", c_float),
                ("w", c_float),
                ("h", c_float)]

class DETECTION (Structure):
    _fields_ = [("bbox", BOX),
                ("classes", c_int),
                ("prob", POINTER(c_float)),
                ("mask", POINTER(c_float)),
                ("objectness", c_float),
                ("sort_class", c_int)]

class IMAGE (Structure):
    _fields_ = [("w", c_int),
                ("h", c_int),
                ("c", c_int),
                ("data", POINTER(c_float))]

class METADATA (Structure):
    _fields_ = [("classes", c_int),
                ("names", POINTER(c_char_p))]

lib = CDLL("libdarknet.so", RTLD_GLOBAL)
```

```
lib.network_width.argtypes = [c_void_p]
lib.network_width.restype = c_int
lib.network_height.argtypes = [c_void_p]
lib.network_height.restype = c_int

do_nms_sort = lib.do_nms_sort
do_nms_sort.argtypes = [POINTER(DETECTION), c_int, c_int, c_float]

free_detections = lib.free_detections
free_detections.argtypes = [POINTER(DETECTION), c_int]

free_image = lib.free_image
free_image.argtypes = [IMAGE]

free_net_threadpool = lib.py_free_net_threadpool
free_net_threadpool.argtypes = [c_void_p]

free_network = lib.free_network
free_network.argtypes = [c_void_p]

get_network_boxes = lib.get_network_boxes
get_network_boxes.argtypes = [c_void_p, c_int, c_int, c_float, c_float, \
                              POINTER(c_int), c_int, POINTER(c_int)]
get_network_boxes.restype = POINTER(DETECTION)

letterbox_image_thread = lib.py_letterbox_image_thread
letterbox_image_thread.argtypes = [IMAGE, c_void_p]
letterbox_image_thread.restype = IMAGE

load_image_thread = lib.py_load_image_thread
load_image_thread.argtypes = [c_char_p, c_void_p]
load_image_thread.restype = IMAGE

load_meta = lib.get_metadata
load_meta.argtypes = [c_char_p]
load_meta.restype = METADATA

load_network = lib.load_network
load_network.argtypes = [c_char_p, c_char_p, c_int]
load_network.restype = c_void_p

network_predict = lib.network_predict
network_predict.argtypes = [c_void_p, POINTER(c_float)]

nnp_deinitialize = lib.nnp_deinitialize

nnp_initialize = lib.nnp_initialize

    set_net_threadpool = lib.py_set_net_threadpool
    set_net_threadpool.argtypes = [c_void_p]

    set_batch_network = lib.set_batch_network
    set_batch_network.argtypes = [c_void_p, c_int]

    draw_predictions = lib.py_draw_predictions
    draw_predictions.argtypes = [IMAGE, POINTER(DETECTION), c_int, c_float, \
                                 POINTER(c_char_p), c_int, c_char_p]

    srand = lib.srand
    srand.argtypes = [c_int]


    # Código principal
    class Accum (object):
        """Acumulador de estadísticas"""
        def __init__ (self):
            # Estadísticas a presentar
            self.total = 0.0    # Suma total
```

```
121.             self.mean = 0.0     # Media
122.             self.stdev = 0.0    # Desviación estándar
123.             self.max = 0.0      # Valor máximo
124.             self.min = 0.0      # Valor mínimo
125.
126.             # Interno
127.             self._accum = list()
128.
129.         def update (self, value):
130.             """Añade un nuevo valor"""
131.             self._accum.append(value)
132.             self.total = sum(self._accum)
133.             self.mean = self.total / len(self._accum)
134.             try:
135.                 self.stdev = stats.stdev(self._accum)
136.             except:
137.                 self.stdev = 0.0
138.             self.max = max(self._accum)
139.             self.min = min(self._accum)
140.
141.         def __str__ (self):
142.             ret = f"Media: {self.mean}; Desviación: {self.stdev};” \
143.                   f“ MaxVal: {self.max}; " \
144.                   f"MinVal: {self.min}"
145.             return ret
146.
147.
148.     class Detection (object):
149.         """Clase que adapta la estructura original "detección"
150.
151.         Almacena información de una detección exacta en una inferencia. Un conjunto
152.         de objetos Detection conforman el resultado de una inferencia.
153.
154.         En concreto, almacena:
155.             * classname (str): Nombre de la clase detectada
156.             * probability (float) ¡: Probabilidad de que sea la clase indicada
157.             * box (float * 4): Cuatro flotantes indicando la localización y tamaño
158.                 exactos del recuadro que enmarca el objeto detectado
159.             * objectness (float): Confianza en que lo detectado sea un objeto.
160.
161.         """
162.         NCLASSES = None
163.         CLASS_NAMES = None
164.
165.         def __init__ (self, det, iclass):
166.             self.classname = Detection.CLASS_NAMES[iclass]
167.             self.prob = det.prob[iclass]
168.             self.box_x = det.bbox.x
169.             self.box_y = det.bbox.y
170.             self.box_w = det.bbox.w
171.             self.box_h = det.bbox.h
172.             self.objectness = det.objectness
173.
174.         def __str__ (self):
175.             ret = f"{self.classname.decode('utf-
    8')!r: <10} con prob {self.prob:.4f} at "
176.             ret += f"(x, y) = ({self.box_x:.4f}, {self.box_y:.4f}) with " \
177.                    f"(width, height) = ({self.box_w:.4f}, {self.box_h:.4f}) and " \
178.                    f"objectness = {self.objectness:.4f}"
179.             return ret
180.
181.
182.     class YOLOResults (object):
183.         """Clase que almacena los resultados de una inferencia completa.
184.
185.         Presenta resultados útiles sobre la ejecución.
186.
```

```
187.         Atributos:
188.             * time (Accum): Almacena los tiempos de ejecución.
189.             * fps (float): Mantiene una relación de los FPS tras la última imagen
190.                 ejecutada.
191.             * objs (Accum): Acumula el número de objetos identificados
192.             * empty (int): Cuenta el número de imágenes que no tuvieron detecciones
193.             * results (list): Lista de los resultados de cada una de las imágenes
194.                 ejecutadas. Cada celda de la lista se corresponde a un diccionario
195.                 con los siguientes elementos:
196.                     - image_path (str): Path de la imagen.
197.                     - time (float): Tiempo tardado en ejecutar.
198.                     - current_fps (float): FPS en el momento de terminar esta imagen
199.                     - detection (list): Lista de objetos Detection, con los resul-
200.                         tados de la inferencia en esta imagen.
201.
202.         """
203.         def __init__ (self):
204.             self.time = Accum()
205.             self.fps = 0.0
206.             self.objs = Accum()
207.             self.empty = 0
208.             self.results = list()
209.             self._total = 0
210.
211.         def append (self, img, time, dets, nboxes):
212.             """Añade nuevas imágenes a la lista de resultados."""
213.             self._total += 1
214.             self.time.update(time)
215.             self.results.append({'time': time,
216.                                  'current_fps': 0.0,
217.                                  'image_path': img,
218.                                  'detection': list()})
219.             self.results[-1]['current_fps'] = self.get_fps()
220.             for box in range(nboxes):
221.                 for clase in range(Detection.NCLASSES):
222.                     if dets[box].prob[clase] > 0:
223.                         self.results[-1]['detection'].append(Detection(dets[box],
224.                                                                        clase))
225.             total_dets = len(self.results[-1]['detection'])
226.             if total_dets == 0:
227.                 self.empty += 1
228.             self.objs.update(total_dets)
229.
230.         def get_fps (self):
231.             """Calcula los FPS actuales"""
232.             self.fps = self._total / self.time.total
233.             return self.fps
234.
235.         def print (self):
236.             """Imprime por pantalla la ejecución total."""
237.             self.short_print()
238.             for result in self.results:
239.                 print(f"Imagen: {result['image_path'].decode('utf-8')}")
240.                 print(f" - Tiempo: {result['time']:.4f}")
241.                 print(f" - FPS acumulado: {result['current_fps']:.4f}")
242.                 for det in result['detection']:
243.                     print(" - ", det)
244.                 print("\n######################################################\n")
245.
246.         def short_print (self):
247.             print()
248.             print(f"Inferencia de {len(self.results)} imágenes")
249.             print(f"Duración total = {self.time.total:.4f}")
250.             print(f"Duración media por imagen = {self.time.mean:.4f}")
251.             print(f"FPS medios = {self.fps:.4f}")
252.             print(f"Objetos detectados por imagen en media = {self.objs.mean:.2f}")
253.             print(f"Total de imágenes sin objetos detectados = " \
254.                   f"{self.empty} ({self.empty/self._total:.2%})")
```

```
255.             print()
256.
257.
258.     def detect (fdata, fcfg, fweight, fimages, thresh=.5, hier_thresh=.5, nms=.3,
259.                 verbose=False, predictions=False):
260.         """Ejecuta un proceso de detección completo, de una o varias imágenes,
261.         en función de diferentes parámetros.
262.
263.         Parámetros:
264.             * fdata (str): Archivo '.data' a utilizar.
265.             * fcfg (str): Archivo '.cfg' a utilizar.
266.             * fweight (str): Archivo '.weight' a utilizar.
267.             * fimages (str): Lista de paths a imágenes a utilizar.
268.             * thresh, hier_thresh (float): Mínimo límite a partir del cual se
269.                 admiten las predicciones de la red. Por defecto, 0.50 ambos.
270.             * nms (float): Non-Maximum Supression, parámetro para eliminar detec-
271.                 ciones redundantes. Por defecto, 0.45.
272.             * verbose (bool): Activar el modo verbose (por defecto desactivado).
273.
274.         """
275.         meta = load_meta(fdata)
276.
277.         # Cargar network
278.         net = load_network(fcfg, fweight, 0)
279.         set_batch_network(net, 1)
280.         srand(2222222)
281.
282.         # Cargar NNPACK
283.         nnp_initialize()
284.         set_net_threadpool(net)
285.
286.         # Ejecutando la inferencia para todas las imágenes pasadas
287.         if verbose: print(f"> Infiriendo {len(fimages)} imágenes...")
288.         results = YOLOResults()
289.         Detection.NCLASSES = meta.classes
290.         Detection.CLASS_NAMES = meta.names
291.         for i, image_path in enumerate(fimages):
292.             # Cargando imagen
293.             if verbose: print(f"> [{i+1}/{len(fimages)}] Cargando {image_path.decode('
    utf-8')!r}...")
294.             img = load_image_thread(image_path, net)
295.             sized = letterbox_image_thread(img, net)
296.
297.             # Prediciendo y detectando
298.             tstart = time.perf_counter()
299.             network_predict(net, sized.data)
300.             tstop = time.perf_counter()
301.             nboxes = c_int(0)
302.             nboxes_pointer = pointer(nboxes)
303.             dets = get_network_boxes(net, img.w, img.h, thresh, hier_thresh, None,
304.                                      1, nboxes_pointer)
305.             nboxes = nboxes_pointer[0]
306.
307.             # Aplicando NMS
308.             if (nms):
309.                 do_nms_sort(dets, nboxes, meta.classes, nms)
310.
311.             # Dibujando las predicciones
312.             if predictions:
313.                 predname = "prediction-" + basename(image_path).decode('utf-8')[:-4]
314.                 draw_predictions(img, dets, nboxes, thresh, meta.names, meta.classes,
    \
315.                                  f"predictions/{predname}".encode("utf-8"))
316.                 if verbose: print(f"  Predicción guardada en {predname!r}")
317.
318.             # Obteniendo resultados
319.             results.append(image_path, tstop-tstart, dets, nboxes)
```

```
320.
321.             # Liberando memoria reservada para la imagen
322.             free_detections(dets, nboxes)
323.             free_image(img)
324.             free_image(sized)
325.
326.             if verbose: print(f"  Hecho. {len(results.results[-
    1]['detection'])} objetos identificados")
327.
328.         # Liberando memoria general del programa
329.         free_net_threadpool(net)
330.         nnp_deinitialize()
331.         free_network(net)
332.
333.         if verbose: print(f"> Finalizado en {results.time.total} segundos.")
334.         return results
335.
336.
337.     # Main
338.     if __name__ == "__main__":
339.         # Argumentos en línea de comandos:
340.         parser = argparse.ArgumentParser(description="Ejecuta el detector de Darknet")
341.
342.         #   Relacionados con archivos
343.         parser.add_argument('-d', default="cfg/coco.data", dest="data", type=str,
344.                             help="selecciona el archivo de datos (por defecto, coco.da
    ta)")
345.         parser.add_argument('-c', default="cfg/yolov3-
    tiny.cfg", dest="cfg", type=str,
346.                             help="selecciona el archivo de configuración (por defecto,
    yolov3-tiny.cfg)")
347.         parser.add_argument('-w', default="yolov3-
    tiny.weights", dest="weights", type=str,
348.                             help="selecciona el archivo de pesos (por defecto, yolov3-
    tiny.weights)")
349.         parser.add_argument('-
    i', default="testing/dog.jpg", dest="images", type=str,
350.                             help="selecciona la imagen o el directorio de imágenes")
351.
352.         #   Relacionados con hiperparámetros
353.         parser.add_argument('-t', '--thresh', default=.5, dest="thresh", type=float,
354.                             help="cambia el thresh de la red (por defecto, 0.5)")
355.         parser.add_argument('-ht', '--hier-
    thresh', default=.5, dest="hthresh", type=float,
356.                             help="cambia el hier_tresh de la red (por defecto, 0.5)")
357.         parser.add_argument('--nms', default=.3, dest="nms", type=float,
358.                             help="cambia el valor de NMS (por defecto, 0.3)")
359.
360.         #   Otros
361.         parser.add_argument('-v', '--verbose', action="store_true", dest="verbose",
362.                             help="modo verbose")
363.         parser.add_argument('-n', default=-1, dest="limit", type=int,
364.                             help="escoge la cantidad indicada de imágenes del director
    io indicado")
365.         parser.add_argument("-p", action="store_true", dest="predict",
366.                             help="genera una imagen con las predicciones de cada infer
    encia")
367.         parser.add_argument('--long-output', action="store_true", dest="lout",
368.                             help="Imprime por pantalla la detección completa."\
369.                                  "Si solo se infiere una imagen, se activa por defecto
    .")
370.         parser.add_argument('-g', action="store_true", dest="graphics",
371.                             help="genera dos gráficas con la evolución de FPS y duraci
    ón")
372.
373.         # Ejecución
```

```
374.        args = parser.parse_args()
375.
376.        # Convirtiendo imagen/carpeta en una lista de imágenes aleatoria.
377.        fimage = args.images
378.        images_list = list()
379.        if isdir(fimage):
380.            images_list = [join(fimage, file) \
381.                           for file in listdir(fimage) \
382.                           if isfile(join(fimage, file))]
383.            if args.limit > 0:
384.                random.shuffle(images_list)
385.                images_list = images_list[:args.limit]
386.        elif isfile(fimage):
387.            images_list = [fimage]
388.        images_list = [bytes(i, encoding="utf-8") for i in images_list]
389.
390.        # Preparando otros elementos
391.        if args.predict:
392.            try:
393.                mkdir("./predictions")
394.            except FileExistsError:
395.                pass
396.            print("Las predicciones serán guardadas en ./predictions/")
397.
398.        # Llamando al detector
399.        res = detect(bytes(args.data, encoding="utf-8"),
400.                     bytes(args.cfg, encoding="utf-8"),
401.                     bytes(args.weights, encoding="utf-8"),
402.                     images_list,
403.                     args.thresh, args.hthresh, args.nms, verbose=args.verbose,
404.                     predictions=args.predict)
405.
406.        # Mostrando el resultado.
407.        if len(images_list) == 1:
408.            res.print()
409.        else:
410.            if args.lout:
411.                res.print()
412.            else:
413.                res.short_print()
414.
415.        if args.graphics:
416.            print("Generando gráficas de FPS y tiempo")
417.            fps_table = list()
418.            time_table = list()
419.            for r in res.results:
420.                fps_table.append(r['current_fps'])
421.                time_table.append(r['time'])
422.
423.            import matplotlib.pyplot as plt
424.            fps_fig = plt.figure()
425.            plt.plot(list(range(len(fps_table))), fps_table)
426.            plt.grid(True, linestyle=':')
427.            plt.xlabel("Images")
428.            plt.ylabel("FPS")
429.            plt.tight_layout()
430.            fps_fig.savefig(f"{len(res.results)}-images-fps-progress.png")
431.            print("Gráfica de FPS generada")
432.
433.            time_fig = plt.figure()
434.            plt.plot(list(range(len(time_table))), time_table)
435.            plt.grid(True, linestyle=':')
436.            plt.xlabel("Images")
437.            plt.ylabel("Time")
438.            plt.tight_layout()
439.            time_fig.savefig(f"{len(res.results)}-images-time-progress.png")
440.            print("Gráfica de tiempo generada")
```

ANEXO E: SCRIPT PARA ANÁLISIS DE ENTRENAMIENTO

Se incluye el código *plotter.py*, escrito en Python, que sirve para interpretar los logs generados durante el entrenamiento de Darknet y representar gráficamente los resultados, como se detalla en la subsección 4.1.4 y se utiliza en la sección 4.3.

https://github.com/angmorpri/darknet-tfg/blob/master/plotter.py

```
#!python3
# -*- coding: utf-8 -*-
"""
    Código para generar gráficas informativas sobre los resultados de un
    entrenamiento con Darknet.
    Parte de los logs que Darknet genera.
    Creado:                 08 Jun 2020
    Última modificación:    03 Nov 2020
    @author: Ángel Moreno Prieto
"""
import argparse
import matplotlib.pyplot as plt


_TRAINING_IMAGES_FILE = "./training/oxford-pet/cat-dog-train.txt"
TOTAL_IMAGES = 3688
DESCRIPTION = "Análisis de logs de entrenamiento. \nLos posibles parámetros "\
              "para los ejes son: batch, loss, avg_loss, rate, time, images y epoch"


class Batch (object):
    """Clase Batch.
    Almacena toda la información disponible por cada lote, es decir:
        * loss (float): Pérdida en entrenamiento, de la última iteración.
        * avg_loss (float): Media de pérdidas en entrenamiento.
        * learning_rate (float): Ratio de aprendizaje en la última iteración.
        * time_taken (float): Tiempo tardado en ejecutar la iteración.
        * images (float): Número de imágenes procesadas.
        * epoch (int): Número de épocas desde el inicio.
    """
    def __init__ (self, line):
        """Obtiene los parámetros a partir de la línea de log del batch."""
        params = [p.strip() for p in line.split(',')]
        self.batch = int(params[0].split(':')[0])
        self.loss = float(params[0].split(':')[1])
        self.avg_loss = float(params[1].split(' ')[0])
        self.learning_rate = float(params[2].split(' ')[0])
        self.time_taken = float(params[3].split(' ')[0])
        self.images = float(params[4].split(' ')[0])
        self.epoch = int(self.images // TOTAL_IMAGES)   # Calculada a mano.

    def getXY (self, x="batch", y="avg_loss"):
        """Devuelve dos valores indicados como X e Y por su nombre."""
        if x == "time": x = "time_taken"
        if y == "time": y = "time_taken"
        if x == "rate": x = "learning_rate"
        if y == "rate": y = "learning_rate"

        retx, rety = None, None
        try:
            retx = eval(f"self.{x}")
```

```
            rety = eval(f"self.{y}")
        except AttributeError:
            if retx is None:
                print(f"Error: El parámetro '{x}' para el eje X no existe")
            elif rety is None:
                print(f"Error: El parámetro '{y}' para el eje Y no existe")

        return (retx, rety)

    def __str__ (self):
        """Presenta los datos obtenidos de forma clara."""
        ret = f"Batch {self.batch}: {self.avg_loss} avg loss, {self.loss} loss,"
        ret += f" {self.learning_rate} rate, {self.time_taken} segundos,"
        ret += f" {self.images} imágenes acumuladas, {self.epoch} épocas."
        return ret


# Main
if __name__ == "__main__":
    # Argumentos en línea de comandos:
    parser = argparse.ArgumentParser(description=DESCRIPTION)
    parser.add_argument("logfile", help="archivo de log del entrenamiento")
    parser.add_argument("-x", type=str, default="batch",
                        help="parámetro del eje X")
    parser.add_argument("-y", type=str, default="avg_loss",
                        help="parámetro del eje Y")
    parser.add_argument("-xmax", type=int, default=None,
                        help="máximo valor en el eje X")
    parser.add_argument("-xmin", type=int, default=None,
                        help="mínimo valor en el eje X")
    parser.add_argument("-ymax", type=int, default=None,
                        help="máximo valor en el eje Y")
    parser.add_argument("-ymin", type=int, default=None,
                        help="mínimo valor en el eje X")
    parser.add_argument("--csv", type=str, default="results.csv",
                        help="nombre del archivo de salida CSV")
    parser.add_argument("--plot_file", type=str, default=None,
                        help="nombre del archivo de la gráfica")
    args = parser.parse_args()

    # Se recogen las líneas útiles del archivo de log:
    print("Filtrando log...")
    batches = list()
    flog = open(args.logfile, 'r')
    for line in flog:
        line = line.strip()
        if "avg" in line:
            # Línea de batch, creamos un nuevo objeto Batch
                batches.append(Batch(line))

        # Una vez se tiene la lista de batches, se preparan los valores de las
        # gráficas a imprimir.
        xvalues = list()
        yvalues = list()
        for batch in batches:
            xy = batch.getXY(args.x, args.y)
            xvalues.append(xy[0])
            yvalues.append(xy[1])

        print("Preparando CSV")
        print(f" Eje X: {args.x}, Eje Y: {args.y}")
        fcsv = open(args.csv, 'w')
        for x, y in zip(xvalues, yvalues):
            fcsv.write(f"{x}, {y}\n")
        fcsv.close()
        print(f" CSV guardado en {args.csv}")

        print("Preparando gráfica")
```

```
        xmax = args.xmax or float("inf")
        xmin = args.xmin or float("-inf")
        ymax = args.ymax or float("inf")
        ymin = args.ymin or float("-inf")
        subxvalues = list()
        subyvalues = list()
        for x, y in zip(xvalues, yvalues):
            if (xmin < x < xmax) and (ymin < y < ymax):
                subxvalues.append(x)
                subyvalues.append(y)
        fig = plt.figure()
        plt.plot(subxvalues, subyvalues)
        plt.grid(True, linestyle=':')
        plt.xlabel(f"{args.x.replace('_', ' ').capitalize()}")
        plt.ylabel(f"{args.y.replace('_', ' ').capitalize()}")
        plt.tight_layout()
        if not args.plot_file:
            args.plot_file = f"plot-{args.x}-{args.y}.png"
        fig.savefig(args.plot_file, dpi=1000)
        print(f" Gráfica guardada en {args.plot_file}")
        print("Finalizado!")
```

ANEXO F: HISTÓRICO DE PRUEBAS DE ENTRENAMIENTO

Las configuraciones de hiperparámetros se muestran sólo cuando modifican las que hubiera anteriormente. También se han simplificado los hiperparámetros `width` y `height` por `size`.

Los fallos causados por *Realloc Error* están relacionados directamente con problemas con la memoria, generalmente por falta de recursos. Los fallos *Segmentation Fault* son causados por problemas de memoria, del mismo modo que *Realloc Erro*, pero también por errores al tratar de cargar imágenes corruptas o al intentar ejecutar las funciones de la red neuronal si los tamaños de entrada de las imágenes no son múltiplos de 32.

Distribución digitalbrain79			
Fecha	**Hiperparámetros modificados**	**Resultado**	**Observaciones**

22-may-20	`batch=32`
	`subdivision=2`
	`size=320x320`
	`max_batches=500200`
	`steps=400000,450000`
		Realloc Error	Primera prueba, configuración estándar.

22-may-20	-	Realloc Error	Fallo inmediato. Posible solución, forzar el puntero que hace realloc() a NULL antes.
22-may-20	-	Segmentation Fault	Fallo inmediato.
22-may-20	`size=192x192`	Segmentation Fault	384 imágenes.
22-may-20	`batch=64` `subdivisions=16`	Realloc invalid old size Aborted	640 imágenes. Problema relacionado con la memoria. Posible solución, forzar el puntero que hace realloc() a NULL antes.
22-may-20	`size=128x128` `random=0`	Segmentation Fault	Puesto el parámetro de YOLO random a 0, para evitar consumo de memoria. Fallo inmediato, menos de 300 imágenes
22-may-20	`batch=1` `subidivisions=1`	Segmentation Fault	Fallo inmediato, menos de 300 imágenes

23-may-20	`batch=64`
	`subdivisions=16`
	`size=96x96`
		Segmentation Fault	Fallo inmediato, menos de 300 imágenes

23-may-20	`max_batches=1000`
	`steps=800,900`
	`size=256x256`
		Segmentation Fault	Fallo tras 384 imágenes.

23-may-20	`max_batches=6000`
	`steps=4800,5400`
	`size=32x32`
		Segmentation Fault	Fallo tras 4288 imágenes.

23-may-20	`max_batches=1000` `steps=800,900`	Segmentation Fault	Fallo inmediato, ni siquiera 128 imágenes

23-may-20	`subdivisions=32`
	`max_batches=6000`
	`steps=4800,5400`
		Segmentation Fault	Fallo tras 704 imágenes.

23-may-20	`batches=32` `subdivisions=8`	Segmentation Fault	Fallo tras 928 imágenes.
23-may-20	-	Segmentation Fault	Fallo tras 928 imágenes. Se repite el error previo por lo que no es circustancial.

23-may-20	`batches=64`
	`subdivisions=16`
	`max_batches=4000`
	`steps=3200,3600`
		Segmentation Fault	Ejecutado *partial* con la nueva configuración y código 15.
			Añadido por primera vez el parámetro *-clear* para borrar configuraciones previas.
			Fallo tras 896 imágenes; 14 iteraciones.

23-may-20	`max_batches=500200` `steps=400000,450000`	Segmentation Fault	Valores de max_batches y steps originales. Ejecutando *partial* una vez más. Fallo tras 1152 imágenes; 18 iteraciones.

23-may-20	`subdivisions=8`
	`max_batches=4000`
	`steps=3200,3600`
		Segmentation Fault	Añadido parámetro extra '1'.
			Fallo tras 256 imágenes, 4 iteraciones.

23-may-20	`batches=24` `subdivisions=1`	Segmentation Fault	Fallo tras 552 imágenes, 23 iteraciones.
23-may-20	`batches=128` `subdivisiones=1`	Segmentation Fault	Fallo tras 1536 imágenes, 12 iteraciones.
23-may-20	`batches=1`	Segmentation Fault	Fallo tras 290 imágenes, 290 iteraciones.
01-jun-20	-	Segmentation Fault	Fallo tras 2703 imágenes, 2703 iteraciones. En "Training network".
01-jun-20	-	Segmentation Fault	Fallo tras 1105 imágenes, 1105 iteraciones. En "Starting get_next_batch" iteración 0.
01-jun-20	`batches=64` `subdivisions=64`	Segmentation Fault	Tras 1024 imágenes, 16 iteraciones. En "Starting get_next_batch".
01-jun-20	-	Segmentation Fault	Ejecutando dos veces seguidas para ver si el fallo está relacionado con una imagen en concreto. Primer fallo tras 2880 imágenes, 45 iteraciones. Mismo lugar, hipotética imagen dog_1388.jpg. Segunda prueba inconclusa: analizando el código, se ve que el error no está ahí.
01-jun-20	`batches=1` `subdivisions=1`	Segmentation Fault	Fallo tras 1779 iteraciones, 1779 imágenes. Error al cargar la última imagen con la que iba a entrenar, parece que yerra cuando se encuentra con una imagen corrupta.

01-jun-20	-	**Parado, pero funcionaba**	Eliminadas imágenes que causaban fallo (se guardaban en un fichero *bad.list*). Finaliza, pero los pesos generados no funcionan en inferencia.

02-jun-20	`batches=64`
	`max_batches=6000`
	`steps=4800,5400`
		Parado, pero funcionaba	Ejecutado *partial*.
			Cortado a las 1000 iteraciones, tras unas 26 horas funcionando correctamente. Los resultados para los pesos a las 1000 iteraciones no generan ningún resultado al usarlos para inferencia.

Distribución shizukachan / personalizado			
Fecha	**Hiperparámetros modificados**	**Resultado**	**Observaciones**
04-jun-20	`subdivisions=16`	Parado, pero funcionaba	Probando si shizukachan admite el entrenamiento. Parado tras 270 iteraciones, funciona similar a digitalbrain79.
05-jun-20	-	**Parado por no aprendizaje**	Ejecutando durante 1000 iteraciones. Parado a las 970, tras 35 horas. A partir de las 670 iteraciones deja de aprender (el error empieza a ser infinito)
07-jun-20	`size=128x128`	**Parado por no aprendizaje**	Ejecutado *partial*. Una vez más a partir de las 670 iteraciones (10 épocas), deja de aprender y aumenta el *loss* hasta infinito.

ANEXO G: CÓDIGO AUXILIAR EN C PARA SCRIPT DE INFERENCIA

Se incluyen a continuación los códigos auxiliares escritos en C para la distribución Darknet utilizada, que permiten que el script *detect.py* (Anexo D) funcione correctamente, como se explica en la subsección 4.1.3.

Código de la cabecera *py_utils.h* (https://github.com/angmorpri/darknet-tfg/blob/master/src/py_utils.h):

```
/*
    Funciones auxiliares en C para wrapper (cabecera)

    Created:        09 May 20
    Last modified:  08 Nov 20
*/
#ifndef PYUTILS_H
#define PYUTILS_H

#include "darknet.h"
#include "image.h"

/* Declarations */
image py_load_image_thread (char *filename, network *net);
image py_letterbox_image_thread (image img, network *net);

void py_set_net_threadpool (network *net);
void py_free_net_threadpool (network *net);

void py_draw_predictions (image im, detection *dets, int nboxes, float thresh,
                          char **names, int classes, const char *outfile);

#endif
```

Código del cuerpo *py_utils.c* (https://github.com/angmorpri/darknet-tfg/blob/master/src/py_utils.c):

```
/*
    Funciones auxiliares en C para wrapper (cuerpo)

    Created:        09 May 20
    Last modified:  08 Nov 20
*/
#include "py_utils.h"

/* Definitions */
image py_load_image_thread (char *filename, network *net) {
    // Loads an image in a different thread.
    return load_image_thread(filename, 0, 0, net->c, net->threadpool);
}

image py_letterbox_image_thread (image img, network *net) {
    // Resizes an image in a different thread and returns.
    return letterbox_image_thread (img, net->w, net->h, net->threadpool);
}

void py_set_net_threadpool (network *net) {
    #ifdef QPU_GEMM
        net->threadpool = pthreadpool_create(1);
    #else
        net->threadpool = pthreadpool_create(4);
```

```
    #endif
}

void py_free_net_threadpool (network *net) {
    pthreadpool_destroy(net->threadpool);
}

void py_draw_predictions (image im, detection *dets, int nboxes, float thresh,
                          char **names, int classes, const char *outfile) {
    image **alphabet = load_alphabet();
    draw_detections(im, dets, nboxes, thresh, names, alphabet, classes);
    save_image(im, outfile);
}
```