UNIVERSIDAD DE VALLADOLID

ESCUELA TÉCNICA SUPERIOR DE INGENIEROS DE TELECOMUNICACIÓN

TRABAJO FIN DE MÁSTER

MÁSTER UNIVERSITARIO EN INGENIERÍA DE TELECOMUNICACIÓN

Aplicación de técnicas de aprendizaje profundo (*deep learning*) para la detección de objetos en Industria 4.0

Autor:

José María Peña Lorenzo

Tutor:

Ignacio de Miguel Jiménez

TÍTULO:	**Aplicación de técnicas de aprendizaje profundo (*deep learning*) para la detección de objetos en Industria 4.0**
AUTOR:	**D. José María Peña Lorenzo**
TUTOR:	**Dro. D. Ignacio de Miguel Jiménez**
DEPARTAMENTO:	**Teoría de la Señal y Comunicación e Ingeniería Telemática**

TRIBUNAL

PRESIDENTE:	Patricia Fernández del Reguero
VOCAL:	Javier Manuel Aguiar Pérez
SECRETARIO:	Ramón J. Durán Barroso
SUPLENTE:	Lourdes Pelaz Montes
SUPLENTE:	Ramón de la Rosa Steinz
SUPLENTE:	Jaime Gómez Gil

FECHA:	29 de septiembre de 2020
CALIFICACIÓN:	

Resumen

La industria 4.0 o cuarta revolución industrial se caracteriza por la digitalización de las empresas mediante la integración de nuevas tecnologías y modelos de negocios con el objetivo de mejorar los procesos empresariales. Uno de los procesos empresariales de mayor relevancia para las empresas es el control de calidad. El control de calidad en la Industria 4.0 se denomina Calidad 4.0 y consiste en la interconexión entre las personas, máquinas y datos mediante diferentes tecnologías con el objetivo automatizar, analizar y obtener resultados positivos que permitan a la empresa abaratar costes y obtener mayores beneficios. La visión artificial industrial es una de las tecnologías de mayor interés en el campo de la Calidad 4.0, esta tecnología consiste en adquirir, procesar y analizar imágenes mediante algoritmos, entre los cuales destacan los de *Deep Learning*. En este Trabajo Fin de Máster (TFM) se ha desarrollado un detector de objetos basado en *Deep Learning* en la Industria 4.0 con el objetivo de detectar la presencia de diferentes defectos en imágenes de placas de circuitos impresos (PCBs). Adicionalmente, se ha diseñado y desarrollado un clasificador de imágenes defectuosas, en este caso para imágenes de impulsores de bombas sumergibles, como paso previo al desarrollo del detector.

El primer lugar, se diseñaron seis configuraciones diferentes de redes neuronales convolucionales. Estas configuraciones logran unos resultados similares a un modelo anterior pero todas ellas presentan un diseño más simple. Para el entrenamiento y evaluación de los clasificadores implementados se empleó un *dataset* de imágenes de bombas impulsoras de agua etiquetadas como defectuosas y no defectuosas. En segundo lugar, se implementaron los detectores de objetos *Single Shot Detector* (SSD) y RetinaNet basados en diferentes CNNs como *backbones* de estas arquitecturas. En el caso de la SSD se implementaron como *backbone* diferentes configuraciones de la red VGG y una CNN personalizada, mientras que para la arquitectura RetinaNet se emplearon dos configuraciones de la red ResNet, la 50 y la 101. Para entrenar y evaluar los detectores se utilizó un *dataset* de imágenes de placas de circuitos impresos (PCBs) las cuales contiene ficheros xml con las etiquetas de los diferentes defectos presentes en las imágenes en formato PASCAL VOC.

Los resultados obtenidos por las diferentes configuraciones del clasificador de imágenes defectuosas son muy buenos, logrando una *accuracy* del 99.24% la mejor de ellas. En el caso del detector, los mejores resultados los logró el RetinaNet con un valor de mAP del 93.78%.

Palabras clave: Industria 4.0, redes neuronales convolucionales, clasificadores de imágenes, detectores de objetos, *Single Shot Detector* y RetinaNet.

Abstract

Industry 4.0 or the fourth industrial revolution is characterized by the digitization of companies through the integration of new technologies and business models with the objetive of improving business processes. One of the most important business processes for companies is quality control. Quality control in Industry 4.0 is called Quality 4.0 and consists of the interconnection between people, machines and data through different technologies with the aim of automating, analyzing and obtaining positive results that allow the company to lower costs and obtain greater benefits. Industrial artificial vision is one of the most interesting technologies in the field of Quality 4.0, this technology consists of acquiring, processing and analyzing images using algorithms, among which those of Deep Learning stand out. In this Master Thesis (TFM), an object detector based on Deep Learning in Industry 4.0 has been developed with the aim of detecting the presence of different defects in images of printed circuit boards (PCBs). Additionally, a defective image classifier has been designed and developed, in this case for images of submersible pump impellers, as a preliminary step to the development of the detector.

First, six different configurations of convolutional neural networks were designed. These configurations achieve similar results to a previous model but all of them feature a simpler design. For the training and evaluation of the implemented classifiers, a dataset of images of water impeller pumps labeled as defective and not defective was used. Second, Single Shot Detector (SSD) and RetinaNet object detectors based on different CNNs were implemented as backbones of these architectures. In the case of the SSD, different configurations of the VGG network and a personalized CNN were implemented as backbone, while for the RetinaNet architecture two configurations of the ResNet network were used, 50 and 101. To train and evaluate the detectors, it was used a dataset of images of printed circuit boards (PCBs) which contains xml files with the labels of the different defects present in the images in PASCAL VOC format.

The results obtained by the different configurations of the defective image classifier are very good, achieving an accuracy of 99.24% the best of them. In the case of the detector, the best results were achieved by the RetinaNet with a mAP value of 93.78%.

Keywords: Industry 4.0, Convolutional Neural Networks, Image Classifiers, Object Detectors, Single Shot Detector and RetinaNet.

Agradecimientos

Quiero mostrar mis agradecimientos a todas las personas que de una manera u otra han contribuido a la realización de este trabajo. En primer lugar, quiero agradecer a mi tutor Ignacio de Miguel por su orientación y ayuda con las dudas que fueron surgiendo durante la realización del trabajo.

También quiero agradecer a mis padres y mi hermana por su amor incondicional, la confianza depositada en mí y su ayuda en todo momento, sin los cuales no podría haber llegado hasta aquí.

Para finalizar, quiero agradecer a Sofía por estar siempre ahí tanto en los buenos como los malos momentos, apoyándome y animándome a seguir adelante.

Contenido

Índice de Figuras

Índice de Tablas

Capítulo 1. Introducción

En este capítulo se van a introducir algunos conceptos asociados al TFM realizado. En primer lugar, se introduce el concepto de Industria 4.0, dado que se trata del campo en el que se desarrolla el TFM. Seguidamente, se definen las hipótesis que motivaron el desarrollo de este trabajo. A continuación, se exponen los objetivos que se ha buscado satisfacer y la metodología empleada para el desarrollo del TFM. Para finalizar este capítulo describe la estructura de este documento.

1.1 Industria 4.0

Durante los últimos tres siglos, la industria ha experimentado grandes cambios asociados principalmente a la evolución de la tecnología. Desde entonces se han producido cuatro grandes cambios en la industria, las cuales se denominan revoluciones industriales (Yang, 2017).

La primera revolución industrial tuvo lugar a finales del siglo XVIII con la incorporación de máquinas impulsadas por energía de agua y vapor como telares o destilerías. Aproximadamente un siglo después, a principios del siglo XX, se produjo la segunda revolución. Esta se desencadena por la introducción del modelo de trabajo conocido como producción en masa, el cual se ayudaba de la energía eléctrica (Lukač, 2015). La siguiente revolución no tuvo lugar hasta principios de la década de 1970, la cual se caracterizó por la automatización y control de los procesos de fabricación utilizando sistemas electrónicos y tecnología de la información. La tercera revolución se denominó revolución digital (F. Shrouf, 2014).

En los últimos años, la industria está experimentando cambios importantes que están dando lugar a la cuarta revolución industrial. A mediados de 2010 en Alemania surgió el término de Industria 4.0 para referirse a esta revolución (Silva T.B., 2020). Desde entonces, muchos expertos han tratado de explicar el concepto de Industria 4.0, algo que no es una tarea sencilla dado que este término depende de diferentes aspectos y se aplica en diversos campos. Según F. Schrouf (2014), la Industria 4.0 busca conseguir la digitalización de los procesos industriales. Para lograrlo se apoya en las tecnologías de la información y comunicación (TIC) y la fabricación de inteligente. De modo que, los productos, los componentes y las máquinas de producción recopilen información y la compartan en tiempo real. Por lo cual, en la Industria 4.0 tiene una gran importancia la capacidad de los sistemas para recibir, analizar y modificar su comportamiento en función de la información, y aprender en función a la experiencia obtenida (F. Shrouf, 2014).

Al igual que en las anteriores revoluciones industriales, la Industria 4.0 está potenciada por la aparición o evolución de ciertas tecnologías. Según Silva, T.B. (2020), algunas de estas tecnologías son:

- **Procedencia industrial**: Esta tecnología dentro de la Industria 4.0 tiene como objetivo proporcionar el origen de las aplicaciones, materias primas o productos durante todo su ciclo de vida a través de las TIC. Con respecto a la empresa, esto representa el control sobre el producto, permitiendo conocer datos como

su origen, el conjunto de materias primas que se utilizó para su fabricación o si ha sufrido alguna restauración, entre otros datos.

- **Realidad aumentada y virtual**: Hay que diferenciar entre realidad aumentada y realidad virtual. Por un lado, la realidad aumentada tiene como objetivo introducir objetos en entornos virtuales y, por otro lado, la realidad virtual busca introducir a los usuarios en entornos virtuales. En los últimos años, el desarrollo de estas tecnologías ha tenido un gran impacto sobre la remodelación del modelo industrial tradicional. Estas tecnologías permiten la integración de personal más especializado, fomenta técnicas de diseño de productos, mantenimiento de equipos, tomas de decisiones y control de seguridad iterativo. El uso de estas tecnologías puede ofrecer reducciones importantes en los costes de las empresas que hagan uso de ellas.
- **Computación en la nube:** es un modelo que permite proporcionar recursos computacionales a demanda, asignándolos y liberándolos con la mínima interacción con el proveedor. Desde el punto de vista industrial, los sistemas integrados con computación en la nube se pueden dividir en núcleo, negocio y fabricación en la nube.
- **Big Data y Data Analytics**: La Industria 4.0 se caracteriza por el crecimiento masivo de los datos en el entorno industrial. Estos datos, deben de ser procesados y usados para mejorar la producción y el rendimiento de las empresas, para ello se emplean estas tecnologías.
- **Simulación**: En los entornos de la Industria 4.0 tienen lugar cambios dinámicos y exponenciales, esto implica que las tecnologías y modelos de trabajo evolucionen de manera continua. Por lo que, la implementación de tecnologías de simulación favorece positivamente los procesos industriales.
- **Internet de las cosas** (*Internet of Things*, IoT): Son un conjunto de dispositivos conectados a Internet, los cuales, interactúan entre sí, entre procesos, personas y tecnologías, generando datos y actuando en entornos internos y externos a través de Internet.
- **Sistemas ciber-físicos** (*Cyber-Physical Systems*, CPS): Son dispositivos que tienen una alta capacidad computacional y que permite una representación confiable de un entorno real.
- **Fabricación aditiva**: El mayor representante de esta tecnología son las impresoras 3D. Este grupo de tecnologías se caracteriza por la combinación de materiales capa a capa para dar lugar a uno o varios productos. Estas tecnologías permiten acortar fases de producción y mejorar la economía de materias primas, entre otras ventajas.

1.2 Hipótesis de trabajo

La Industria 4.0 trajo consigo nuevas tecnologías o mejoras a tecnologías anteriores como el IoT, el Big Data y las técnicas de inteligencia artificial (IA) que han favorecido la digitalización de la industria. Los cambios provocados por la Industria 4.0 afectan a cada rincón de la industria, la calidad y la producción, dos aspectos de gran importancia dentro de la industria (Treelogic, 2019)

El control de calidad es un aspecto de gran importancia dentro de la Industria 4.0, hasta el punto de que se puede denominar Calidad 4.0. El control de calidad puede ser optimizada mediante la incorporación de innovaciones tecnológicas. El control de

calidad tiene un peso muy grande en la cadena de valores de cualquier industria, dado que dota a la producción de una mayor seguridad, fiabilidad y confianza mediante la automatización de procesos productivos y la gestión integral de la cadena de suministro (Treelogic, 2019).

Para garantizar la sostenibilidad y competitividad económica, las industrias deben poseer un protocolo que garantice la calidad de los procesos internos de la empresa (Treelogic, 2019). Uno de los procesos donde más interés tiene la aplicación del control de calidad es en la línea de producción, con el objetivo de determinar si el producto final fabricado presenta algún defecto y, por tanto, su debe ser retirado. Una de las maneras más comunes de analizar la presencia de defectos de un producto es por medio de su visualización directa o analizando una imagen del producto. El control de calidad basado en imágenes del producto, en la Industria 4.0, se hace inviable para las personas dado que las cadenas de fabricación fabrican un gran número de productos lo cual requerirá de un gran número de personal para poder realizar la comprobación de calidad. Este incremento tendrá asociado un coste que no beneficia a la empresa, y a pesar de poder hacerlo, puede que determinados defectos no sean apreciables para las personas y no puedan realizar una correcta clasificación de las imágenes.

En este TFM se ha desarrollado, en primer lugar, un clasificador de imágenes defectuosas y no defectuosas y, en segundo lugar, un detector de defectos en imágenes, ambos basados en técnicas de aprendizaje profundo (*Deep Learning*). En primer lugar, el clasificador de imágenes defectuosas tiene como objetivo presentar un modelo que ayude al control de calidad de una fábrica inteligente, permitiéndole detectar y separar aquellos productos que tienen algún defecto de los que no tienen ninguno, evitando así la comercialización de piezas defectuosas que pudieran causar problemas a la empresa como pérdidas económicas por indemnizaciones o perder poder frente a la competencia. Por último, el detector de defectos es ir un paso más allá del clasificador anterior, detectando y localizando los diferentes tipos de defectos que presenta el producto. La detección de defectos permite todo lo que ya hacía el clasificador, pero además podría permitir la reutilización de determinados componentes del producto si no están afectados por ningún defecto, o en algunos casos reacondicionar el producto si el defecto no es crítico.

1.3 Objetivo del TFM

El objetivo principal de este TFM ha sido desarrollar e implementar un detector de defectos en imágenes industriales basado en *Deep Learning*. Para lograr el objetivo principal, se decidió fijar un objetivo intermedio, el cual consistió en el desarrollo de un clasificador de imágenes defectuosas industriales empleando técnicas de *Deep Learning.* Para cumplir estos objetivos, se plantearon una serie de objetivos específicos que conducen al cumplimiento del objetivo general de una manera guiada y paulatinamente. Los objetivos específicos fueron:

1. Familiarizarse con el lenguaje de programación Python y la biblioteca de código abierto de aprendizaje automático Tensorflow.
2. Realizar una revisión del estado de la técnica acerca de clasificadores de imágenes basados en *Deep Learning*.

3. Buscar una base de datos pública de imágenes industriales apropiada para el problema de clasificación.
4. Desarrollar el modelo del clasificador de imágenes defectuosas.
5. Evaluar el funcionamiento del clasificador diseñado, realizar una comparación con diferentes configuraciones del modelo y con otro modelo, y extraer las conclusiones.
6. Realizar una revisión del estado de la técnica sobre modelos de detección de objetos, poniendo especial interés en la detección de defectos.
7. Buscar bases de datos de imágenes industriales con defectos etiquetados.
8. Diseñar e implementar un modelo del detector de defectos en imágenes industriales.
9. Evaluar los resultados obtenidos para el modelo y extraer las conclusiones de este.

1.4 Fases del TFM

Las fases en las que se ha dividido el desarrollo de este TFM ha estado muy ligado a la consecución de los objetivos específicos marcados en el apartado anterior, y han sido las siguientes:

1. Consulta de bibliografía relacionada con el *Deep Learning*, así como sus principales arquitecturas subyacentes (redes neuronales multicapa y convolucionales) y técnicas de aprendizaje para estos mecanismos.
2. Familiarización con el lenguaje de programación Python 3 y la librería de código abierto de aprendizaje automático Tensorflow.
3. Búsqueda de información acerca de modelos de clasificadores de imágenes, consultando principalmente revistas científicas y libros de actas de congresos.
4. Implementación del clasificador diseñado para identificación de imágenes defectuosas y no defectuosas. Empleando para ello Tensorflow y Python.
5. Obtención de los resultados del entrenamiento y test del modelo de clasificador implementado y análisis de los resultados.
6. Consulta de revistas científicas, actas de congresos y libros sobre modelos de detectores de objetos.
7. Implementación de los detectores de objetos.
8. Recolección y análisis de los resultados de los detectores de objetos.
9. Elaboración de la memoria del trabajo realizado.

En la Figura 1.1 se ilustra el esquema con las fases descritas anteriormente.

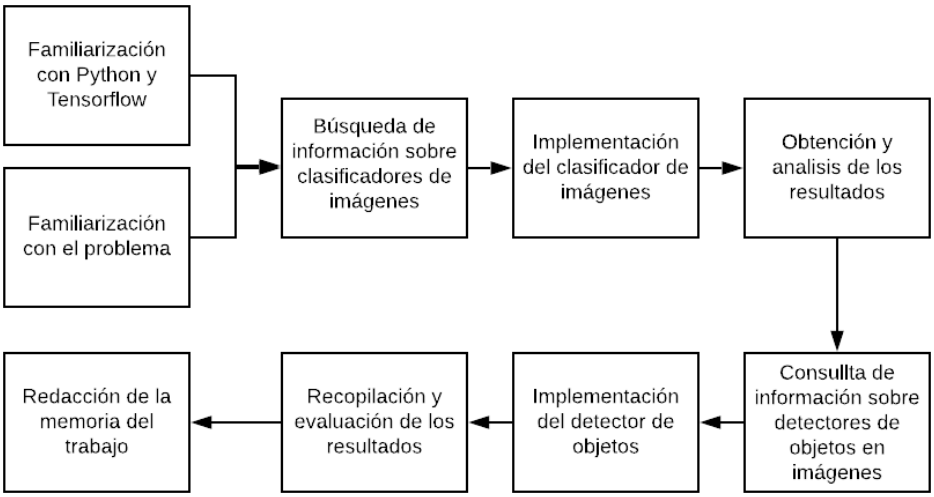


Figura 1.1: Fases seguidas para el desarrollo del TFM

1.5 Estructura del trabajo

Esta memoria se divide en los siguientes capítulos:

- **Capítulo 1: Introducción**
  Como ya se ha visto, se introduce el concepto de la industria 4.0, además se exponen la hipótesis de trabajo, los objetivos a cumplir con la realización de este trabajo y las fases en las que se dividió el trabajo para lograr cumplir los objetivos de manera escalonada.

- **Capítulo 2: Conceptos teóricos**
  En este capítulo se presentan las áreas del aprendizaje automático en los que se desarrolla este trabajo, así como los diferentes algoritmos y técnicas fundamentales que se emplean para su implementación.

- **Capítulo 3: Clasificador**
  En el tercer capítulo se presenta el clasificador de imágenes desarrollado en este trabajo, definiendo el *dataset* de imágenes empleado y la arquitectura del modelo del clasificador. Además, se recopilan los resultados y se compara con diferentes configuraciones de este y con otro modelo.

- **Capítulo 4: Detector de objetos**
  De manera similar a lo que se hizo con el clasificador, en este capítulo se introducen los detectores de objetos utilizados para la detección de defectos, además se introduce el *dataset* de imágenes. Por último, se exponen los resultados obtenidos y se realiza una comparación entre ambos modelos.

- **Capítulo 5: Conclusiones y líneas futuras**
  En este capítulo se exponen las conclusiones obtenidas a partir del trabajo realizado, además se presentan las limitaciones y líneas futuras que se pueden seguir del trabajo realizado.

Además de estos cinco capítulos, se ha incluido un Anexo con las descripciones de *software* y *hardware* empleadas y la descripción del código empleado para la implementación de los detectores SSD y RetinaNet. Por último, se incluye la bibliografía utilizada.

Capítulo 2. Marco teórico

Este capítulo tiene como objetivo presentar los conceptos teóricos en torno a los cuales se desarrolló este trabajo. En primer lugar, se introduce el concepto de *Deep learning*, exponiendo sus principales algoritmos empleados en este trabajo, las redes neuronales artificiales y las convolucionales. Por último, se exponen las técnicas y las métricas más comunes para la evaluación de los modelos.

2.1 *Deep Learning*

El *Deep Learning* o aprendizaje profundo, es una técnica de aprendizaje automático perteneciente al campo de la inteligencia artificial (IA), en específico al subcampo del *Machine learning* o aprendizaje automático. Los algoritmos de *Deep Learning* se caracterizan por el uso de arquitecturas jerárquicas capaces de aprender abstracciones de alto nivel. Estas arquitecturas están formadas por capas de unidades de procesamiento apiladas, y se denominan redes neuronales artificiales dado que su funcionamiento trata de emular el de las células del sistema nervioso de los seres vivos.

Las principales características de los algoritmos de *Deep Learning* que lo diferencian de las otras técnicas de *Machine Learning* son (Grietens, 2018):

- La extracción de características de los datos de entrada en *Deep Learning* se realiza de forma automática mediante el entrenamiento de la red, mientras que, en otras técnicas de *Machine Learning* la extracción de características se realiza manualmente.
- Los algoritmos de *Deep Learning* requieren trabajar con una cantidad de datos muy superior a otras técnicas de *Machine Learning*, debido a que la red debe de ser entrenada para su correcto funcionamiento y contiene un número muy elevado de parámetros que deben ajustarse adecuadamente mediante ese entrenamiento.
- Los algoritmos de *Deep Learning* requieren de sistemas con una capacidad computacional más alta que otros de *Machine Learning*, esto se debe a que estos tienen arquitecturas más complejas y trabajan con mayor cantidad de datos. Esta característica hizo que al comienzo de su desarrollo no tuvieran mucho éxito, el cual no lo alcanzaron hasta la primera década de los 2000 con la aparición de GPUs de mayor potencia.
- Los algoritmos de *Deep Learning* tienen una mayor flexibilidad.

Todas estas características convierten a los algoritmos de *Deep Learning* en unos de los más utilizados en la actualidad en el desarrollo de sistemas de aprendizaje automático. Estos algoritmos tienen especial aplicación en áreas como la medicina, por ejemplo, para el diagnóstico de enfermedades mediante el análisis de datos clínicos como pueden ser imágenes, señales cerebrales, etc., o en la empresarial, donde una de sus aplicaciones es la realización de modelos predictivos (prnoticias, 2016).

La aparición de la Industria 4.0 se ha convertido en uno de los campos donde más posibilidades y mayores avances está teniendo el *Deep Learning*, en especial en la visión artificial. La visión artificial combinada con estos algoritmos permite llevar a cabo tareas como la detección de defectos u objetos, medir distancias o leer textos que han

permitido la automatización de tareas y por tanto una reducción de costes y tiempos a la empresa (atriainnovation, 2019).

El *Deep Learning* se emplea también en muchos otros tipos de aplicaciones, como por ejemplo traductores inteligentes, reconocimiento de voz, reconocimiento facial o interpretación semántica (SmartPanel, 2018).

2.1.1 Redes neuronales artificiales

Como se introdujo anteriormente, la mayor parte de los métodos de aprendizaje de *Deep Learning* emplean arquitecturas de redes neuronales, por lo que comúnmente estos modelos se denominan redes neuronales profundas (Mathworks, 2020). Una red neuronal es un modelo matemático que trata de imitar el funcionamiento y estructura de una red neuronal biológica. De manera que, al igual que una red neuronal biológica está formada por la unión de millones de neuronas, la red neuronal artificial está formada por la unión de neuronas artificiales en diferentes capas (Caparrini, 2019). En la Figura 2.1 se muestra un ejemplo de una arquitectura de una red neuronal, donde los círculos representan las diferentes neuronas que la componen. En esa misma imagen, se puede apreciar tres tipos de capas en función de su disposición en la red, estas son:

- Capa de entrada: Es la encargada de pasar los datos de entrada a la primera capa oculta, el número de neuronas de esta capa dependerá del número de entradas de la red.
- Capas ocultas: Son las encargadas de procesar los datos de entrada, el número de capas y neuronas es muy variable y dependerá de la red.
- Capa de salida: Es la encargada de realizar la clasificación o la regresión final, el número de neuronas depende de cuantas clasificaciones realice la red (o de cuantos valores numéricos reales se quiera predecir en el caso de utilizarse para resolver problemas de regresión).

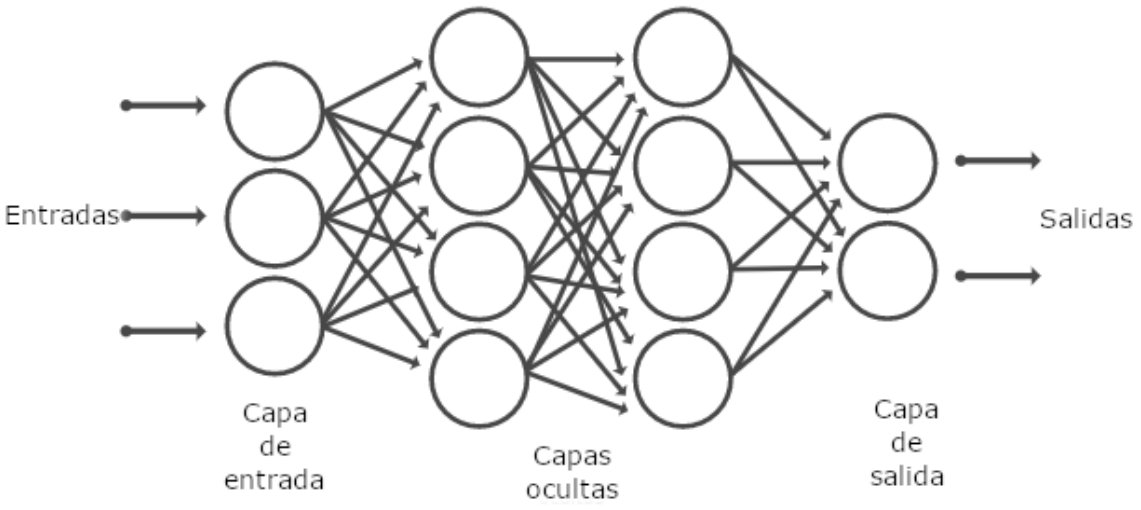
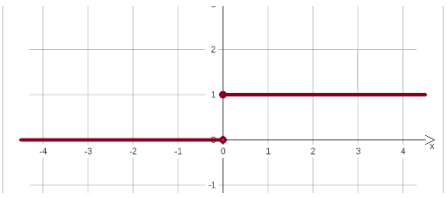
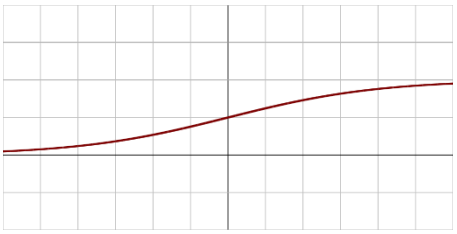


Figura 2.1: Esquema una red neurona artificial.

La neurona artificial es la unidad fundamental de proceso de las redes neuronales, capaz de procesar datos y generar una salida en función a los datos procesados. En la

Figura 2.2 se muestra el esquema de una neurona artificial, en esta figura se muestran las partes de la neurona artificial y su equivalente correspondiente con la parte anatómica de la neurona biológica. Las partes de las que se compone una neurona artificial son (Haykin, 1994):

- Una o varias entradas (X_i): Estas pueden corresponderse con los datos de entrada, si la neurona se encuentra en la capa de entrada, o con la salida de otras neuronas de la capa anterior si se trata de una neurona de la capa oculta o de salida.
- Los pesos (W_i): Son números asociados a las respectivas entradas de la neurona que indican la importancia o peso que tiene cada entrada en la función de red o sumatorio, es decir, la importancia que tiene cada entrada sobre la salida generada por la neurona.
- *Bias*, sesgo o umbral (W_0): Es un término constante que se añade a la función de red. Su finalidad es incrementar o reducir la entrada de la función de activación ya sea positivo o negativo respectivamente.
- Función de red, sumatorio o función de propagación (Σ): Es la encargada de sumar las entradas de la neurona ponderadas por los pesos asociadas a cada entrada.
- La función de activación (f): Es la encargada de generar una salida a partir del valor de salida de la función de red y en función de la ecuación asociada a dicha función. Las funciones de activación más típicas son la sigmoide, la ReLu (*Rectified Linear Unit*), la tangente hiperbólica y la *Softmax* entre otras. En la Tabla 2.1 se recogen la representación gráfica de cada una de estas funciones y su respectiva ecuación matemática (Géron, 2019).

Función de activación	Representación gráfica	Función matemática
Escalón		$f(x) = \begin{cases} 0 \; para \; x < 0 \\ 1 \; para \; x > 0 \end{cases}$
Sigmoide		$f(x) = \frac{1}{1 + e^{-x}}$

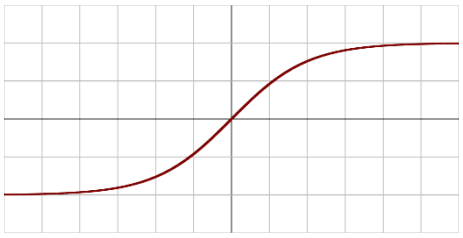
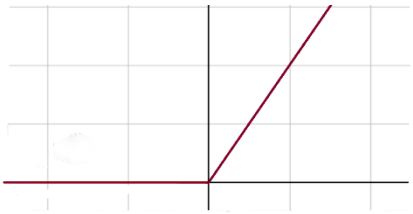
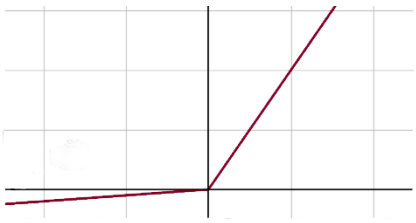
Tangente hiperbólica		$f(x) = \tanh(x) = \frac{(e^x - e^{-x})}{(e^x + e^{-x})}$
ReLu		$f(x) = \begin{cases} 0 \; para \; x < 0 \\ x \; para \; x > 0 \end{cases}$
Leaky ReLu		$f(x) = \begin{cases} 0.01x \; para \; x < 0 \\ x \; para \; x > 0 \end{cases}$

Tabla 2.1: Representación gráfica y fórmula matemática de las principales funciones de activación. (Imágenes obtenidas de (Caparrini, 2019)).

La función *Softmax* es una generalización de la función sigmoide. Esta función de activación se aplica en la capa de salida de clasificadores multiclase, dado que escala las entradas precedentes de un rango entre 0 y 1, y normaliza la capa de salida, de manera que la suma de todas las salidas es 1. Esta función de activación es muy común encontrarla en la capa de salida de redes neuronales convolucionales. La ecuación de esta función es la siguiente (FA Softmax, 2020):

$$f(x)_i = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \; para \; j = 1, \dots, K \qquad \textbf{(2.1)}$$

donde:

- z es el vector de las entradas a la capa de salida.
- j indexa las clases de salida.

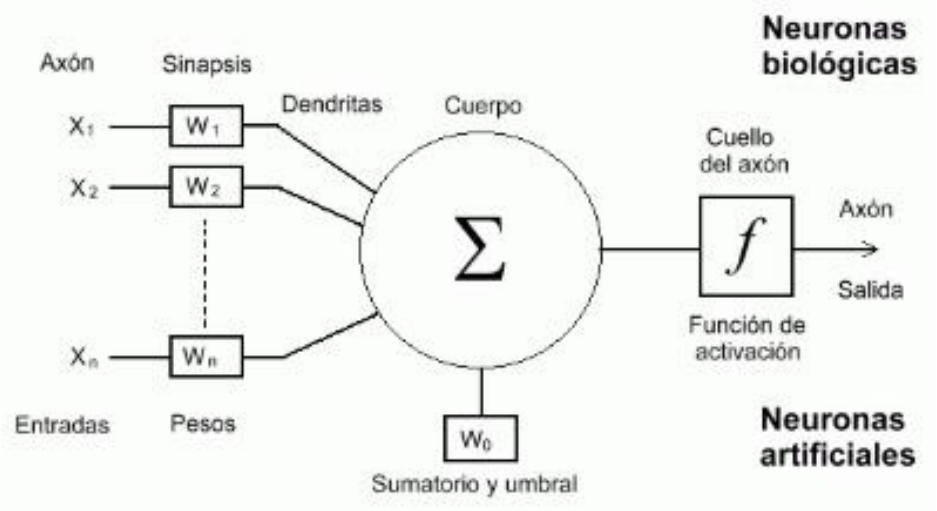


Figura 2.2: Esquema de una neurona artificial (Caparrini, 2019).

En base a estos elementos de la neurona artificial, la ecuación matemática que representa una neurona artificial es la siguiente (Haykin, 1994):

$$Y = f\left(\sum_{i=1}^{n} X_i W_i + W_0\right) \quad \textbf{(2.2)}$$

Las neuronas se organizan en capas dando lugar a una red neuronal artificial, normalmente cada neurona de cada capa está conectada con todas las neuronas de la capa previa. Las capas tienen una gran importancia de cara al proceso de aprendizaje de este tipo de algoritmos. El proceso de aprendizaje consiste en buscar los valores de los pesos que permitan el correcto funcionamiento de la red diseñada (Haykin, 1994).

En la actualidad, hay un número muy amplio de modelos de redes neuronales que permiten resolver diferentes problemas, algunos ejemplos de modelos son las redes neuronales convolucionales, las redes neuronales recurrentes, las redes neuronales probabilísticas y los *autoencoders*. De todos estos modelos, en el TFM nos centramos en las redes neuronales convolucionales ya que son las más idóneas para trabajar con imágenes.

2.1.2 Redes neuronales convolucionales

Las redes neuronales convolucionales (*Convolutional neural network*, CNN) surgieron para suplir la deficiencia que tenían las redes neuronales cuando tenían como entradas datos de gran tamaño. Este modelo de redes neuronales se emplea en campos como el análisis de audios, procesamiento de lenguaje natural o visión por computador (Anwar, 2018).

Las CNN tratan de emular el funcionamiento de la corteza visual del cerebro humano. El objetivo de las CNN es aprender características de orden superior utilizando operaciones convolucionales, estas características no son únicamente de una región de la imagen, sino de todo el conjunto de la imagen. Las tareas dentro del campo de la visión más comunes para las que se emplean son la clasificación de imágenes y detección de objetos (ITELLIGENT, 2018).

Este tipo de redes se caracteriza por una arquitectura especial como la que se muestra en la Figura 2.3, observando esta imagen se pueden apreciar tres tipos de capas características de este modelo: capas convolucionales, capas de *pooling* y capas totalmente conectadas (*Fully conected*).

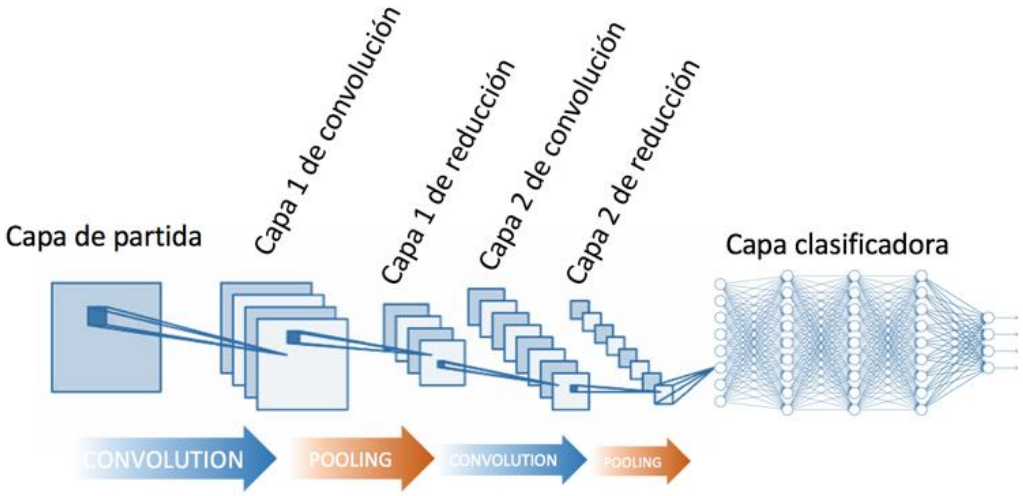


Figura 2.3: Arquitectura de una CNN (Calvo, 2017).

- **Capas convolucionales**: Son las principales capas ocultas de las CNN, de hecho, dan nombre a este modelo de redes neuronales multicapa. Estas capas se llaman así porque tiene la función de llevar a cabo la operación de convolución, esta operación está pensada para trabajar sobre tres ejes, dos de ellos asociados a la altura y anchura de las imágenes, y otro que se corresponde con la profundidad o canal de la imagen, este último puede ser uno si la imagen es en escala de grises y tres si es en color RGB. Aunque la entrada habitual de las capas convolucionales son imágenes, también pueden emplearse con datos de entrada tengan una relación espacial.

  La convolución es una operación matemática que consiste en los productos y sumas de los datos de entrada a la capa y varias matrices que se denominan filtros o *kernels*, dando como resultado un mapa de características o *features maps* (Calvo, 2017). El proceso de convolución comienza con el filtro posicionado sobre la parte izquierda de la matriz de la imagen. Seguidamente, se multiplican los elementos del filtro con los respectivos elementos de la imagen sobre los que se posiciona el filtro y se suman, el resultado de este sumatorio se corresponde con el primer elemento del mapa de características. Este proceso se repite sucesivamente a medida que el filtro recorre la matriz de la imagen paso a paso de izquierda a derecha y de arriba a abajo, tal y como se muestra en la Figura 2.4, donde se representa el proceso de convolución (Cowley, 2018). El resultado de este proceso es una nueva matriz cuyas dimensiones sufren una reducción con respecto a la original, permitiendo reducir el número de conexiones y parámetros a entrenar en la CNN. El tamaño de esta matriz se obtiene con la siguiente ecuación:

$$(h - n + 1) \times (w - n + 1) \times (r \times q) \tag{2.1}$$

donde:

- h y w son la altura y anchura de la imagen de entrada.
- n es la dimensión de altura y anchura del *kernel*.
- r y q es la profundidad de la imagen de entrada y del *kernel* respectivamente.

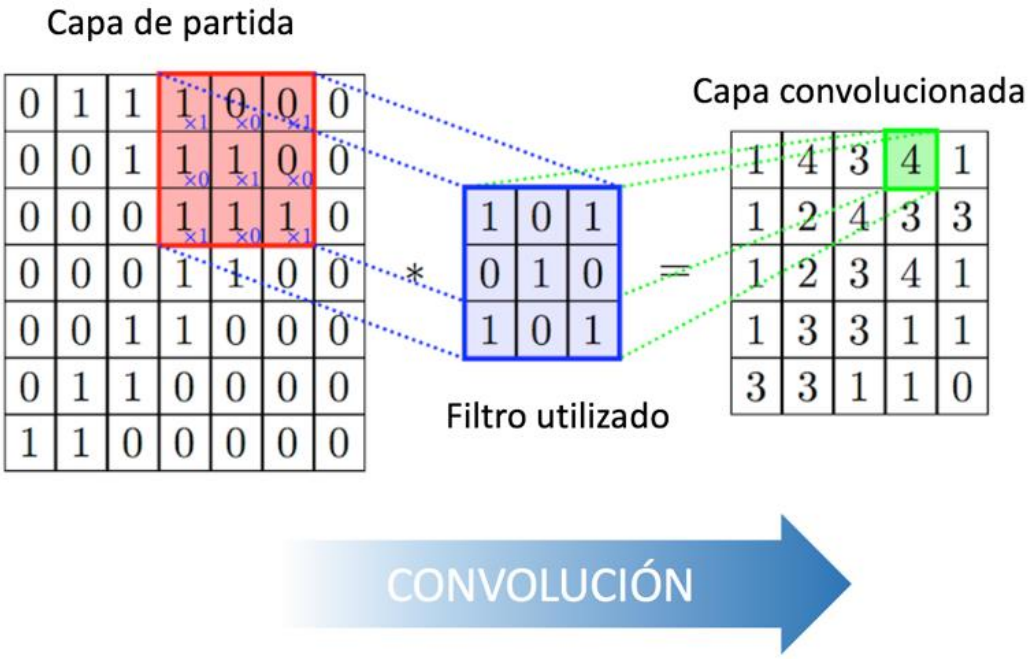


Figura 2.4: Operación de convolución realizada por la capa convolucional (Calvo, 2017).

En algunas ocasiones, esta operación puede producir la pérdida de información, debido a que la información más relevante se dispone en los píxeles de las esquinas. Para conseguir que esta información se sitúe en el centro de la matriz se utiliza la técnica de *Zero Padding* o padeamiento de ceros. Esta técnica añade ceros alrededor de la matriz, en la Figura 2.5 se muestra cómo funciona esta técnica sobre una matriz.

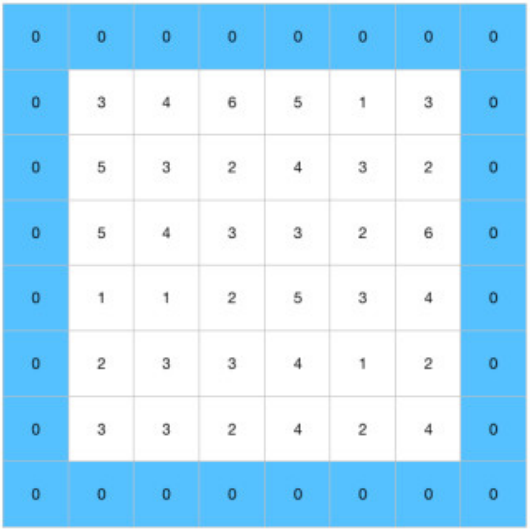


Figura 2.5: Operación de *Zero Padding* (Rodríguez, 2018)

Tras la operación de convolución se aplica una función de activación sobre los mapas de características. Funciones de activación habituales son la ReLu y la Leaky ReLu (Calvo, 2017).

La capa convolucional permite aprender patrones, como bordes o líneas, en la entrada, de manera que pueda reconocer esos patrones en una nueva localización de la entrada o en diferentes entradas. Otra característica importante de estas capas es que la superposición de capas permite la detección de características de mayor complejidad.

- **Capa de *pooling* o de reducción:** Esta capa generalmente se coloca tras una capa convolucional. La función de esta capa es realizar un submuestreo de los mapas de características, reduciendo sus dimensiones conservando la información importante. Esto permite reducir aún más la carga computacional y los parámetros de la CNN de lo que ya hacen las capas convolucionales.

  Hay dos tipos de operaciones de *pooling* (Saha, 2018):
  - *Average-pooling*: Esta operación devuelve el valor medio de los pixeles del mapa de características que abarca la ventana de *pooling*.
  - *Max-pooling*: Devuelve el valor máximo de los pixeles del mapa de características que cubre la ventana de *pooling*.

  El proceso de *pooling* comienza colocando la ventana de *pooling* sobre los respectivos pixeles del mapa de características de la parte superior izquierda. Seguidamente, calcula el máximo o la media de los pixeles que cubre la ventana, en función de la operación seleccionada dando lugar al primer valor del mapa de características de salida. Este proceso se repite sucesivamente recorriendo la imagen de derecha a izquierda y de arriba abajo en función de los pasos indicados. En la Figura 2.6 se representa el proceso de *pooling* con un tamaño de ventana de 2x2 sobre una matriz (que representaría una imagen) de tamaño 4x4.

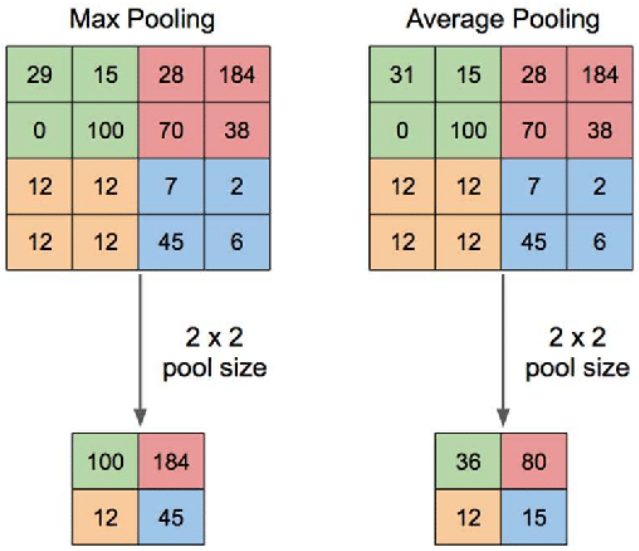


Figura 2.6: Ejemplo de la operación las operaciones de *Max Pooling* (izquierda) *y Averange Pooling* (derecha) de tamaño 2x2 aplicada sobre un matriz 4x4 (Yani, 2019).

- **Capa de clasificación**: Se trata de una o varias capas de neuronas artificiales que forman un perceptrón multicapa, las cuales se disponen al final de las capas convolucionales encargadas de la extracción de características. En primer lugar, esta capa presenta un tipo de capa denominado *flatten* encargada de "aplanar" los mapas de características, convirtiendo la información tridimensional procedente de las convoluciones de modo que dicha información sea compatible para una neurona tradicional. A continuación, presenta una o varias capas *fully-connected* o capas densas, encargadas de la extracción de características de la información de la capa *flatten*. Por último, se tiene una capa de salida encargada de realizar la clasificación, para ello disponen de tantas neuronas como clases se quieran diferenciar, cuando se trata de una clasificación multiclase se suele emplear una capa con función de activación *softmax*, en la Figura 2.7 se muestra un ejemplo de capa de clasificación para un problema multiclase, en ella se distinguen una capa *flatten*, una *fully-conneted* y una de salida con función *sofmax*.

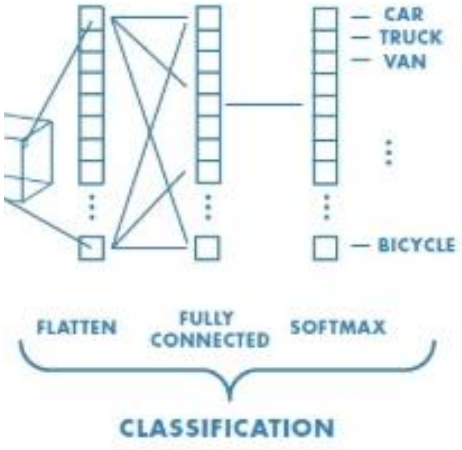


Figura 2.7: Arquitectura de la capa *Fully-connected* (adaptada de (Mathworks, 2020)).

2.1.3 Entrenamiento de redes neuronales

Para lograr el aprendizaje de los pesos de la red neuronal, el algoritmo básico es el denominado algoritmo de *backpropagation* o retropropagación. El algoritmo de *backpropagation* es un mecanismo basado en el descenso del gradiente (*gradient descend*) utilizando una técnica eficiente para calcular los gradientes automáticamente. El proceso consiste en dar los siguientes pasos (Géron, 2019):

- En primer lugar, se introducen los datos en la red, estos atraviesan las diferentes capas de la red hasta llegar a la última, la capa de salida. Durante esta etapa se conservan las salidas de todas las neuronas de cada capa. Esta etapa se conoce como *Forward propagation*.
- Seguidamente, se mide el error de la salida de la red mediante la función de pérdida, comparando la salida obtenida con la que debería ser y devuelve una medida del error.
- Por último, tiene lugar la *back propagation* que consiste en propagar hacia atrás, es decir, de la capa de salida a la capa de entrada, el error. El algoritmo es capaz

de determinar cuál es la contribución de cada neurona de una capa y aplica el algoritmo de descenso de gradiente para ajustar los pesos de la red.

Este proceso se repite con cada iteración, denominada época (*epoch*), y por cada época se realizan tantas iteraciones como lotes o *batches* de datos se configuren. En la Figura 2.8 se representa el esquema del proceso de aprendizaje sobre una red neuronal.

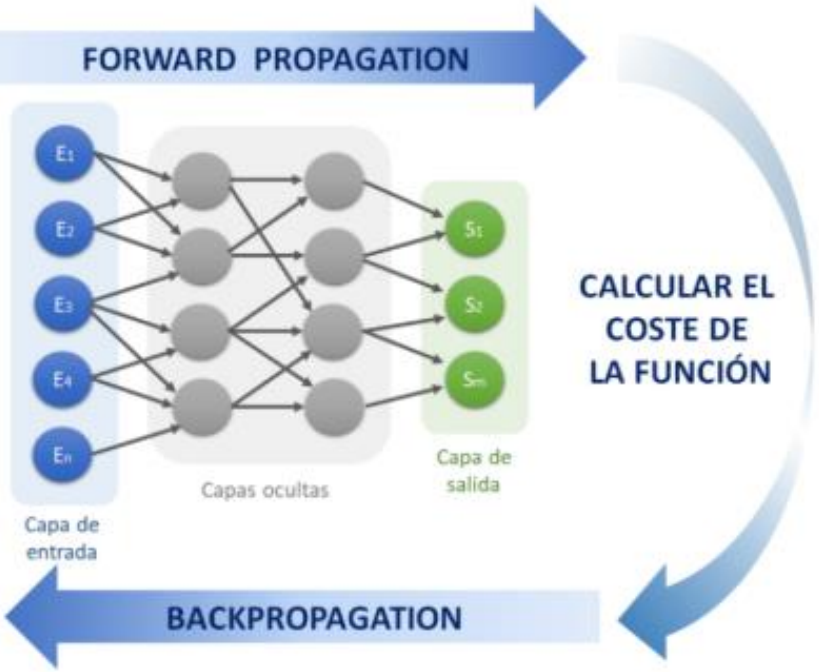


Figura 2.8: Esquema del proceso de aprendizaje de una red neuronal (Calvo, 2017).

2.1.4 Modelos de CNNs preentrenados

Como se expuesto a lo largo de este capítulo, las CNNs son unas herramientas de gran rendimiento en la clasificación y detección de objetos en imágenes. Este hecho ha ocasionado que en los últimos años se hayan propuestos numerosas modelos que intentan mejorar los resultados de los modelos previos, algunos de ellos son muy conocidos como la VGG. ResNet, GoogLeNet o AlexNet. Para poder emplear estos modelos adecuadamente y conseguir el mejor resultado posible se desarrolló la técnica de *Transfer Learning*, la cual se introduce continuación. Además, en este TFM se emplearon dos modelos de CNNs preentrenadas que también se introducen en este apartado, estos modelos son el VGG y ResNet.

2.1.4.1 *Transfer Learning*

El *Transfer Learning* es una de las técnicas de *Deep Learning* de mayor interés. Este interés radica en que permite emplear modelos, que ya han sido previamente entrenados con un amplio conjunto de datos, como punto de partida para entrenar este modelo con nuevos datos con el fin de realizar una nueva tarea. De esta manera, al emplear *Transfer Learning* el modelo no va a ser entrenado desde cero, lo cual supone unas ventajas con respecto a entrenarlo desde cero como:

- El tiempo de entrenamiento del modelo es menor.
- La cantidad datos necesarios para entrena el modelo pueden ser menor.
- Las métricas iniciales del modelo son mejores.

No en todos los casos que se aplique *Transfer Learning* se van a conseguir las ventajas anteriores, de hecho, puede haber algunas que no se den. Las ventajas anteriores serán más notables cuando mayor sea la relación entre los datos con los que se preentrenó el modelo y con los que se quiere entrenar. De modo que, si un modelo es previamente entrenado con un conjunto de datos lo suficientemente amplio y general, el modelo preentrenado obtenido puede resultar útil para muchos otros problemas, a pesar de que estos nuevos problemas no guarden mucha relación con las tareas originales (Torres, 2019).

Hay dos formas de usar *Transfer Learning* (Torres, 2019):

- *Feature Extraction*: Esta técnica de *Transfer Learning* consiste en emplear los parámetros configurados de un modelo preentrenado para la extracción de características de los nuevos datos del problema. En estos casos, únicamente se reutiliza la base convolucional del modelo (capas convolucionales, y de *pooling* del modelo) encargada de la extracción de características, para ello se congelan las capas que conforman la base convolucional. Congelar capas de una red neuronal significa que los parámetros de estas no van a modificarse durante el proceso de entrenamiento. En cuanto a la capa *Fully-connected*, por lo general no se reutiliza, dado que es la encargada de realizar la clasificación de los datos y está preparada diseñada para las clases de los datos originales que pueden no coincidir con las de los nuevos datos. Por ello, en esta técnica la capa *fully-connected* se suele reemplazar por otra diseñada en específico para los nuevos datos, esta capa será entrenable y sus parámetros se ajustarán durante el entrenamiento.

- *Fine Tuning*: Esta técnica es complementaria a la técnica anterior, permitiendo mejorar el rendimiento del modelo obtenido por la técnica anterior. Además, permite un ajuste más fino del modelo preentrenado, para ello, se entrenan algunas de las últimas capas de la base convolucional, que en el caso anterior permanecían congeladas, y la capa *fully-connected* añadida. Se opta por entrenar las últimas capas de la base convolucional porque, como se indica en la sección 2.1.2 medida que se superponen capas convolucionales estas van aprendiendo características más abstractas, las cuales son más interesantes para una mejor clasificación.

2.1.4.2 VGG

El modelo de la red VGG fue propuesto por K. Simonyan and A. Zisserman, investigadores del Grupo de Geometría Visual (*Visual Geometry Group*) de la Universidad de Oxford, y presentado en el desafío ILSVRC de 2014. La red VGG salió subcampeona de dicha competición y mejoró los resultados de la red AlexNet. Además, la VGG obtuvo una exactitud del 92.7% que la convierte en una de las 5 mejores redes en términos de exactitud en ImageNet.

La arquitectura de la red VGG-16 se representa en la Figura 2.9, en ella, se puede apreciar que la entrada a la red tiene que ser una imagen RGB de tamaño 224x224 píxeles (224x224x3). En lo que se refiere a la configuración de sus capas convolucionales, en el documento presentado se indicaban dos configuraciones posibles para el caso particular de 16 capas de pesos, estas configuraciones se muestran en la Tabla 2.2, donde cada columna representa una configuración diferente, aquellas filas que son comunes indican que comparten especificaciones entre las columnas. La base convolucional de la VGG-16 está formada por cinco bloques de capas convolucionales, estos constituidos por dos, dos, tres, tres y tres capas convolucionales. En las columnas C y D de la Tabla 2.2 se muestran la configuración de cada capa, el primer número del nombre de la capa convolucional especifica el tamaño del *kernel*, el cual puede ser 1x1 o 3x3, y el segundo indica el número de características que extrae. Las capas de *maxpooling* tienen un tamaño de 3x3. Por último, la red VGG consta de tres capas *fully connected* y una capa de salida *softmax* para realizar la clasificación (Simonyan, 2014).

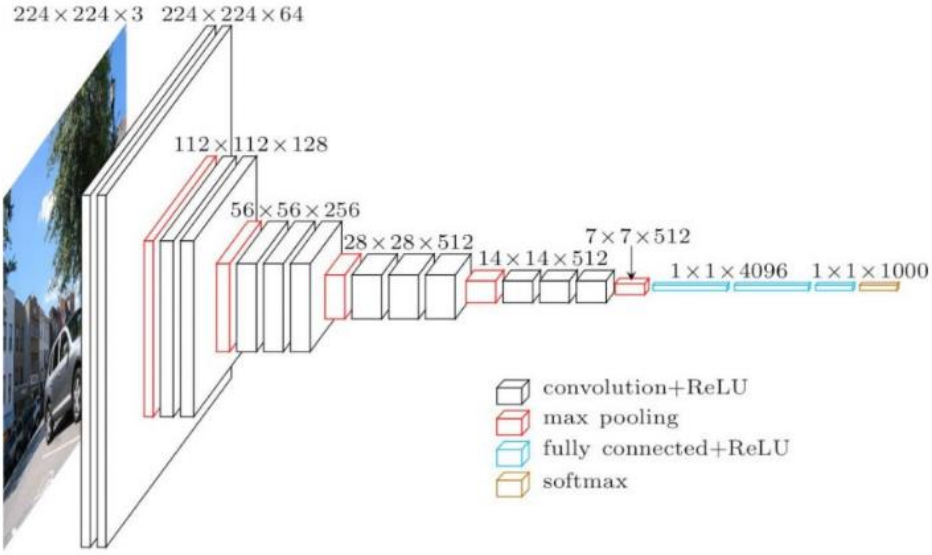


Figura 2.9: Arquitectura de la red VGG-16 (Neurohive, 2018)

La red VGG mejoró los resultados obtenidos por la red AlexNet, la ganadora de la anterior edición del desafio. Ambas redes tienen la misma arquitectura de la capa *fully-connected*. en lo que respecta a la parte convolucional, la VGG tiene un mayor número de capas convolucionales que la red AlexNet, tal y como se muestra en la Figura 2.10 donde se comparan ambas arquitecturas. El hecho de tener un mayor número de capas y que estas tengan un *kernel* de menor tamaño permite a VGG tener un mejor rendimiento y una función de decisión más discriminatoria (Wei, 2019)

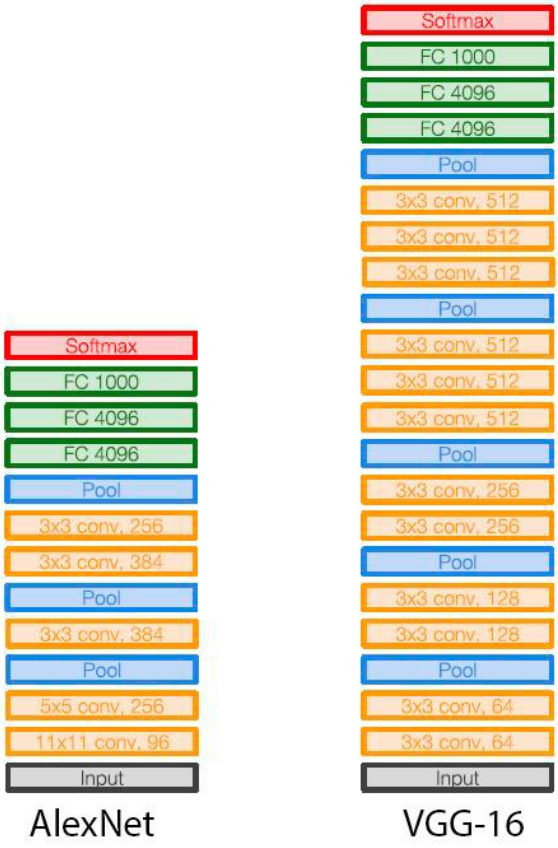


Figura 2.10: Comparación arquitectura AlexNet y VGG-16 (MC.AI, 2018)

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

Tabla 2.2: Configuraciones de la red VGG (Simonyan, 2014)

2.1.4.3 *ResNet*

Al igual que el modelo anterior, *ResNet* (abreviatura en inglés de *Residual Net*) fue presentado por primera vez en el desafío ILSVRC 2015, concurso en el que consiguió el primer puesto. El modelo ResNet supuso una gran innovación con respecto a las anteriores campeonas del desafío, AlexNet y VGG, de hecho, para muchos investigadores *ResNet* ha sido uno de los trabajos más innovadores en el campo de la visión por computación. Esta innovación se debió al incremento de capas adicionales, dando lugar a redes con más de 50 capas, las cuales eran inviables antes por su peor rendimiento. El rendimiento bajo de las redes más profundas se puede atribuir al problema de desvanecimiento del gradiente (*vanishing gradient*), el cual a medida que se propaga hacia atrás puede tener un valor infinitamente pequeño, lo que provoca que a medida que se profundiza en la red, su rendimiento se reduzca. El problema del gradiente se soluciona en el modelo ResNet con la implementación de los bloques residuales.

El bloque residual es la base de la arquitectura ResNet, dicho bloque se caracteriza porque presenta una conexión de su entrada con su salida, denominada conexión de acceso directo de identidad, de manera que la salida del bloque puede ser la salida de las capas convolucionales que lo conforman o la entrada del bloque comportándose como una capa identidad. En la Figura 2.11 Figura 2.11: Comparación entre un bloque con dos capas convolucionales y un bloque residual .se muestra una comparación entre un bloque con dos capas convolucionales seguidas y un bloque residual, el bloque residual es necesario para aplicar la función identidad dado que una capa convolucional no puede por si sola aprender a devolver la entrada que le llega. La salida de estos bloques es la aplicación de la función de activación ReLu a la suma de los dos caminos, a continuación, se muestra la función de salida de un bloque residual:

$$F(x) = h(x) + x \quad \textbf{(2.3)}$$

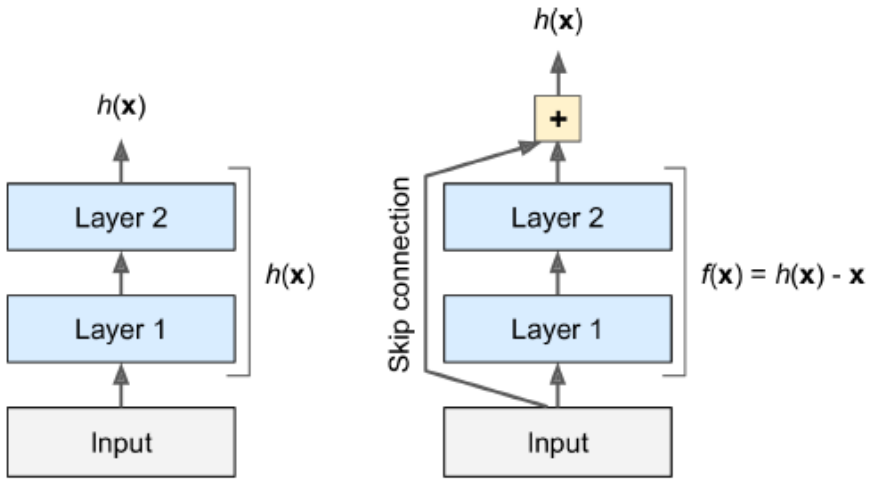


Figura 2.11: Comparación entre un bloque con dos capas convolucionales y un bloque residual (Géron, 2019).

La arquitectura ResNet presenta diferentes configuraciones en cuanto a las capas que la conforman, en este TFM se emplearon únicamente la ResNet-50 y ResNet-101, las

cuales se caracterizan por tener 50 y 101 capas de pesos respectivamente. La estructura genérica de ResNet se representa en la Figura 2.12, donde el conjunto de bloques residuales se representa como un único bloque denominado *Deep*.

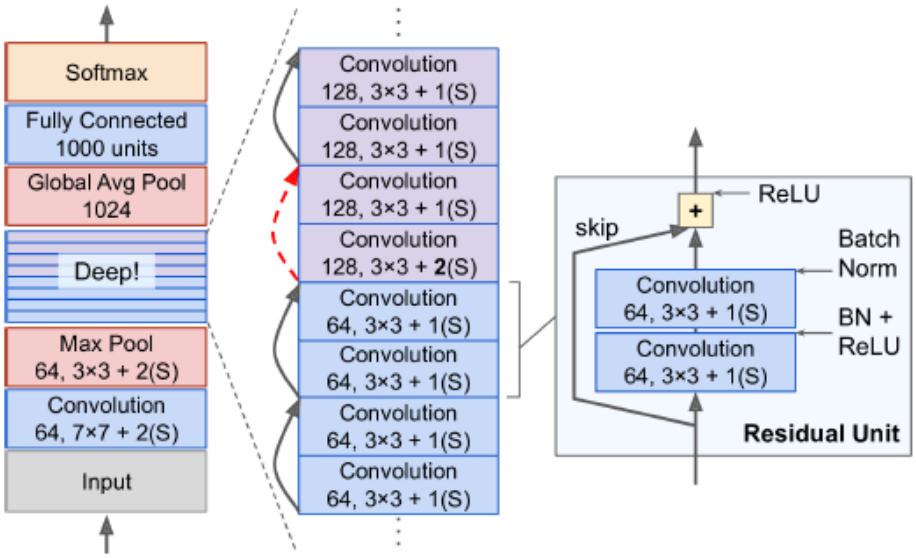


Figura 2.12: Arquitectura general de una red ResNet (Géron, 2019).

La arquitectura de los bloques residuales de ResNet consta de dos capas convolucionales con un *kernel* de tamaño 3x3 y un paso de 1, Batch *Normalization* (BN) y función de activación ReLu. La capa de BN tiene la función de normalizar los valores de salida de la capa predecesora, de manera que el rendimiento de la red mejores.

La arquitectura del bloque residual de la Figura 2.12 supone un problema cuando se quiere sumar la entrada del bloque con las salidas del bloque, debido a que tienen un tamaño diferente. Esta diferencia de tamaño se debe a que cada vez que la entrada atraviesa una capa convolucional el tamaño de esta se reduce, de modo que al atravesar las dos el tamaño de la entrada se reduce el doble. Para conseguir que la suma sea posible, la conexión que une entrada y salida tiene que implementar una capa convolución de *kernel* de tamaño 1x1 y paso 2, tal como se muestra en la Figura 2.13.

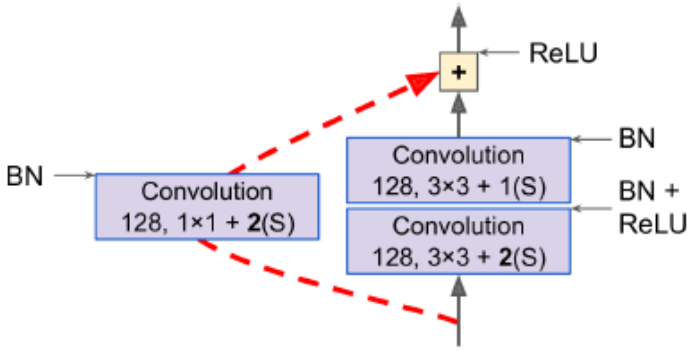


Figura 2.13: Arquitectura del bloque residual (Géron, 2019).

2.2 Evaluación de clasificadores

Tras la implementación de los modelos, se procedió a la evaluación de los modelos implementado. En este apartado se exponen, en primer lugar, las métricas que se emplearon para la evaluación y comparación de modelos. Por último, se describen las técnicas implementadas para poder realizar una correcta evaluación y comparación entre modelos, concretamente, el uso de semillas para la generación de números pseudoaleatorios, y técnicas de regularización (en particular, *early stopping*) de validación (en particular, *k-fold cross validation*).

2.2.1 Métricas

Los clasificadores implementados en este trabajo tienen como objetivo clasificar las imágenes del *Dataset* en dos clases diferentes, dicho *Dataset* se describe más detalladamente en el apartado 3.1:

- Defectuosa (clase Positiva): Cuando el clasificador detecta la presencia de algún tipo de defecto en la imagen de entrada.
- No defectuosa (clase Negativa): Cuando el clasificador no detecta la presencia de ningún defecto en la imagen.

Para evaluar el funcionamiento del modelo, se realizó una comparación entre la clasificación realizada por el modelo y la etiqueta real de cada imagen. La herramienta matemática más adecuada para visualizar esta comparación es la denominada matriz de confusión. La matriz de confusión es una matriz cuadrada de tamaño *n* x *n*, siendo *n* el número de clases del modelo. Las filas de la matriz representan las instancias de cada clase, mientras que, las columnas representan el número de predicciones de cada clase. La matriz de confusión para el clasificador diseñado en este trabajo es de tamaño 2x2 ya que se trata de un problema de clasificación binario. En la Tabla 2.3 se muestra una matriz de confusión 2x2.

		Predicción	
		Defectuosa (Positiva)	**No defectuosa (Negativos)**
Observación	**Defectuosa (Positiva)**	Verdaderos Positivos (VP)	Falsos Negativos (FN)
	No defectuosa (Negativos)	Falsos Positivos (FP)	Verdaderos Negativos (VN)

Tabla 2.3 Matriz de confusión de un clasificador binario

Los elementos de la matriz que se muestra en la Tabla anterior son los siguientes (Barrios, 2019):

- VP: es el número de positivos que fueron detectados como positivos por el clasificador.

- FN: es la cantidad de positivos que fueron detectados como negativos por el clasificador.
- FP: es el número de negativos que fueron detectados como positivos por el clasificador.
- VN: es la cantidad de negativos que fueron detectados como negativos por el clasificador.

Los elementos VP y VN indican el número de veces que el clasificador ha realizado con éxito la predicción, mientras que, los FP y FN se corresponden a las ocasiones en las que el clasificador funcionó incorrectamente.

La matriz anterior y los elementos de este caso particular se pueden extrapolar a una matriz de mayor tamaño en el caso de que el clasificador sea multiclase.

En función de la matriz de confusión obtenida para un modelo, se pueden calcular una serie de métricas de gran utilidad para la evaluación del modelo. Estas métricas son la precisión, la sensibilidad (*recall*), la puntuación F1 (F1 *score*), la *accuracy*, la especificidad y el área bajo la curva. Además de las anteriores métricas, también se puede calcular la pérdida del modelo mediante la función de pérdida. A continuación, se exponen estas métricas más detalladamente.

2.2.1.1 Precisión y Sensibilidad

La precisión es una métrica que representa la probabilidad de realizar una predicción de realizar una predicción correcta de una clase con respecto a todas las clases predichas correctamente. La precisión se calcula mediante la siguiente ecuación (Barrios, 2019):

$$Precisión\ (\%) = \frac{VP}{VP + FP} \times 100 \quad \textbf{(2.4)}$$

La precisión por sí sola no es una métrica muy útil para la evaluación de un modelo, generalmente, se calcula junto a otra métrica llamada sensibilidad (Géron, 2019).

La sensibilidad o *recall*, también se conoce como Tasa de Verdaderos Positivos (TRP). Esta métrica mide la proporción de casos positivos que fueron correctamente clasificados. La ecuación de esta métrica es la siguiente (Géron, 2019):

$$Recall\ (\%) = \frac{VP}{VP + FN} \times 100 \quad \textbf{(2.5)}$$

Cuando se evalúa un modelo, se busca maximizar el valor de la precisión y la sensibilidad. En la práctica, la función de clasificación se encarga de tomar la decisión de clasificar los datos de entrada en una determinada clase en función de un umbral, el umbral supone un compromiso entre la precisión y la sensibilidad, de modo que el objetivo es encontrar el umbral que maximiza ambos valores.

2.2.1.2 Puntuación F1 (F1 *Score*)

Esta métrica está directamente relacionada con las dos anteriores, es la media armónica de la precisión y la sensibilidad. Su fórmula matemática es:

$$F1 = \frac{2}{\frac{1}{Precisión} + \frac{1}{Recall}} = 2 \times \frac{precisión \times recall}{precisión + recall} = \frac{TP}{TP + \frac{FN + FP}{2}} \quad \textbf{(2.6)}$$

2.2.1.3 Accuracy

La *accuracy* mide el porcentaje de predicciones realizadas correctamente por el clasificador, independientemente de la clase. Matemáticamente se expresa como:

$$Accuracy\ (\%) = \frac{VP + VN}{VP + FP + VN + FN} \times 100 \quad \textbf{(2.7)}$$

2.2.1.4 Especificidad

La especificidad o tasa de verdaderos negativos mide la proporción de casos negativos correctamente clasificados. La especificidad se calcula mediante la siguiente ecuación:

$$Especificidad\ (\%) = \frac{VN}{VN + FP} \times 100 \quad \textbf{(2.8)}$$

2.2.1.5 Área debajo de la curva ROC

La curva ROC (*Receiver Operating Characteristic*) es otra herramienta muy común y de gran utilidad para la evaluación de clasificadores binarios. La curva ROC es una gráfica que representa el ratio de verdaderos positivos (*recall*) versus el ratio de falsos positivos (especificidad), este último es el ratio de instancias negativas correctamente clasificadas por el clasificador, en la Figura 2.14 se muestra un ejemplo de una curva ROC, la cual se representa como una línea continua de color azul (Géron, 2019).

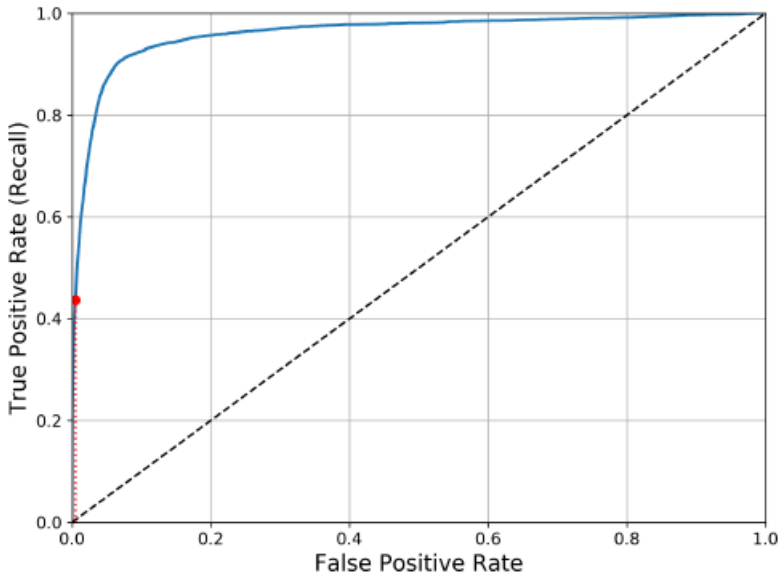


Figura 2.14: Representación de la curva ROC (Géron, 2019).

La curva ROC es una herramienta de gran utilidad para determinar comparar modelos y determinar cuál de ellos es el óptimo para el problema analizado. La curva ROC es siempre creciente, este hecho representa el compromiso que tiene la sensibilidad con la especificidad, de modo que, si se aumenta el valor de corte de la sensibilidad se reducirá el valor de la especificidad. Si el clasificador no pudiera distinguir entre clases, la curva de la ROC sería la diagonal que une el vértice inferior izquierdo con el derecho superior, este se representa como una diagonal puntada en la Figura 2.14 (de Ullibarri Galparsoro, 1998).

La exactitud del clasificador aumenta a medida que la curva se desplaza desde la diagonal hacia el vértice superior izquierdo, es decir, cuando la curva aumenta por encima de la diagonal. De modo que, cuando mayor sea el área bajo la curva ROC (*Area Under the Curve*, AUC) mayor será la probabilidad de clasificar una instancia correctamente, esta es una métrica muy común para evaluar el modelo a partir de la curva ROC de este (Burgueño, 1995).

2.2.1.6 Función de pérdida

La función de pérdida o función de coste mide el error entre el valor estimado por la red neuronal y el valor real, es decir, mide como de bueno es el funcionamiento de la red neuronal. Si el error obtenido es elevado, el funcionamiento de la red será malo, mientras que si el error es bajo el funcionamiento de la red será mejor. Durante el entrenamiento de la red se busca minimizar este error. Hay un amplio número de funciones de pérdida, entre las cuales hay que escoger la más adecuada al problema que intenta resolver la red neuronal. En este TFM se han empleado diferentes funciones de pérdidas, para el clasificador se empleó la entropía cruzada binaria (*Binary Cross-Entropy*). Mientras que para el detector se emplearon la pérdida definida en el artículo de Liu Wei para el detector basado en la SSD y la pérdida focal para la RetinaNet.

2.2.2 Semillas para la generación de números pseudoaleatorios

Tensorflow cuenta con determinadas operaciones que tienen un componente de aleatoriedad. Esta aleatoriedad supone un problema a la hora de evaluar un modelo dado que en cada ejecución los resultados obtenidos sufrirán determinada variación, esto resulta un obstáculo para poder realizar comparaciones entre diferentes configuraciones del modelo u otros modelos de clasificador.

La aleatoriedad de las operaciones depende de la semilla que se utilice (pues es la que marca la secuencia de números pseudoaleatorios que se generará). Esta semilla deriva a su vez de dos semillas, la semilla global y las semillas de nivel operativo. Las interacciones entre la semilla global y las de operaciones genera la secuencia aleatoria. Por lo tanto, para conseguir que los resultados de un modelo sean repetibles y la componente aleatoria no repercuta sobre ellos, se fijaron la semilla global y las de nivel operativo de manera que la secuencia aleatoria obtenida con ellas es siempre la misma.

2.2.3 *Early stopping*

Uno de los principales problemas que pueden darse a la hora de entrenar una red neuronal es el sobreajuste de esta. Una de las técnicas más comunes para evitar el sobreajuste es el *Early Stopping* por su facilidad a la hora de implementarse y por sus buenos resultados en comparación a otras técnicas (Prechelt, 1998).

Esta técnica de regularización va a realizar un seguimiento de determinada métrica del modelo durante el entrenamiento, exactamente se va a centrar en una métrica del conjunto de validación. Cada determinado número de épocas, se va a evaluar la métrica comparándola con la obtenida en la anterior evaluación, este proceso se repite

sucesivamente hasta que el valor de esta métrica no mejora durante un número determinado de evaluaciones denominado *patience*. En la realidad, la curva del error de validación no es regular como se muestra en la Figura 2.15, sino que, es irregular y presenta máximos y mínimos locales, como se representa en la Figura 2.16, que pueden dar lugar a equivocación si no se fija un valor de *patience* lo suficientemente grande. Cuando se supera el número de *patience* sin mejoras, se detiene el entrenamiento de la red neuronal y se usan los pesos que tenía la red en la última evaluación antes de que la métrica dejará de mejorar (Prechelt, 1998).

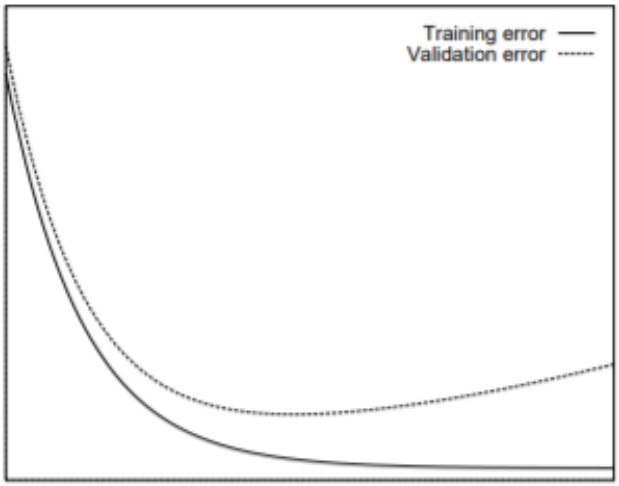


Figura 2.15: Curvas del error de entrenamiento (continua) y error de validación (discontinua) ideales (Prechelt, 1998).

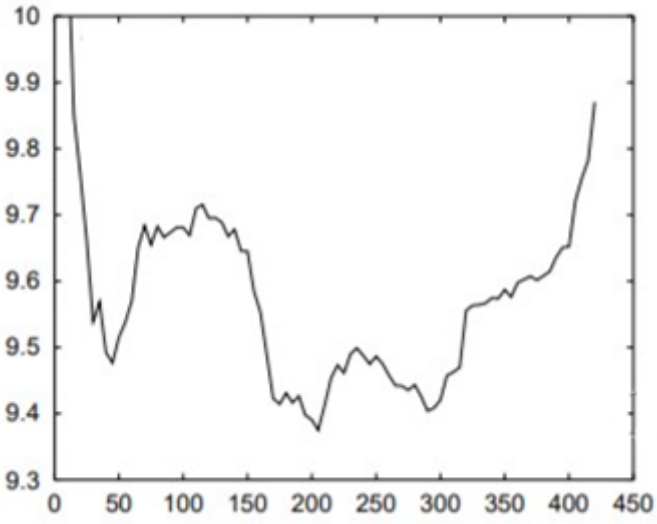


Figura 2.16: Ejemplo de una curva del error de validación real (Prechelt, 1998).

En este trabajo se configuró la técnica de *Early Stopping* con la pérdida de validación como métrica de seguimiento, la cual se evalúa cada época y una *patience* de 10 épocas.

2.2.4 *K-Fold cross validation*

Para realizar el entrenamiento y evaluación de un modelo se debe dividir el *dataset* de imágenes en tres subconjuntos: entrenamiento, validación y test. En este TFM, los subconjuntos son creados aleatoriamente, esto supone cierto problema a la hora de analizar los resultados obtenidos por el modelo, ya que es posible que los subconjuntos creados estén desbalanceados o las clases no sean suficientemente variadas.

Para solucionar dicho problema y conseguir mejores resultados en cuanto medición de la predicción de modelo, y además hacer un uso más eficiente de los datos, se empleó la validación cruzada de *K* iteraciones (*K-fold Cross-Validation*) siguiendo el procedimiento que se describe a continuación. La validación cruzada de *K* iteraciones es un proceso iterativo, el cual consiste en dividir los datos de forma aleatoria en *K* grupos aproximadamente del mismo tamaño. De estos grupos de datos, *K*-1 se emplean para entrenamiento y validación del modelo y el otro grupo restante se emplea para test, este proceso se repite iterativamente *K* veces empleando en cada iteración un grupo diferente para el test del modelo (Rodrigo, 2016).

Tras la creación de los grupos, en cada iteración el grupo de entrenamiento/validación, se divide de forma aleatoria en dos subconjuntos, uno para entrenamiento y otro para validación (para aplicar *early stopping*), con una relación de 80% y 20% respectivamente. Una vez finalizado el entrenamiento, se evalúa el modelo entrenado con el grupo de test, los datos obtenidos de esta evaluación se almacenan y se procede a realizar la siguiente iteración donde se repite el mismo proceso anterior. Una vez concluidas las *K* iteraciones, las métricas obtenidas en las diferentes iteraciones se representan en forma de *box plot* o se promedian para poder evaluar finalmente el modelo.

Este proceso supone un incremento de la carga computacional del modelo, aunque el rendimiento de este proceso sigue siendo inferior a otras técnicas como *Leave-One-Out-Cross-Validation* (LOCOM). Además, no desperdicia demasiados datos, lo cual es muy interesante cuando no se dispone de un *dataset* muy amplio (scikit-learn.org, 2020).

En este trabajo se implementó un proceso de validación cruzada de 10 iteraciones (*K*=10), porque, como se indica en Amat, R. J. (2016) y Flórez, L. R., & Fernández, F. J. M. (2008), se consigue una buena fiabilidad sin elevar excesivamente la carga computacional.

Estrictamente hablando, la validación cruzada resulta útil para la evaluación de prestaciones, pero si se utiliza para comparar varios modelos y seleccionar uno de ellos se introduce sesgo de selección. Una forma de evitarlo es implementar una técnica denominada validación cruzada anidada (Raschka, 2020), a costa de aumentar significativamente el coste computacional. De todas formas, hay autores que consideran que, generalmente, no es necesario realizar la validación cruzada anidada (Y.S. Abu-Mostafa, 2012) (Johnson, 2016). En el TFM no se utilizará la validación anidada debido a que el número de modelos que se compararán no será muy elevado (y al coste computacional que supondría la alternativa).

Capítulo 3. Clasificador de imágenes

En este capítulo se aborda el diseño e implementación de un modelo basado en el aprendizaje profundo para la clasificación de imágenes industriales como defectuosas y no defectuosas. En primer lugar, se introduce la base de datos de imágenes empleadas para el desarrollo del clasificador. Seguidamente, se explica el pre procesado que se aplica a las imágenes de entrada. A continuación, se expone el modelo diseñado para el clasificador y el modelo diseñado por el usuario de *Kaggle* GuillaumeSimler (GuillaumeSimler, 2020), con el que se quiere comparar. Por último, se exponen las técnicas y métodos empleados para la evaluación del modelo y los resultados obtenidos.

3.1 *Dataset* de imágenes

La selección de un *dataset* de imágenes es una tarea de gran importancia a la hora de diseñar un clasificador de imágenes que funcione adecuadamente. Por ello, tras realizar una búsqueda exhaustiva de diferentes *Datasets* de imágenes industriales, se optó por usar un *Dataset* de la página *Kaggle* titulado *"Casting product image data for quality inspection"* y diseñado por Ravitajsinh Dabhi (Dabhi, 2020).

Este *Dataset* consta de 7204 imágenes de impulsores de bomba sumergible, las cuales fueron tomadas desde un punto de vista superior a la pieza. Las imágenes fueron tomadas con una cámara Canon EOS 1300D y con una iluminación estable. Además, las imágenes se reconvirtieron a escala de grises y resolución 300x300 píxeles, de manera que cada imagen tiene un solo canal y cada píxel puede tomar un valor en el rango de 0 a 255.

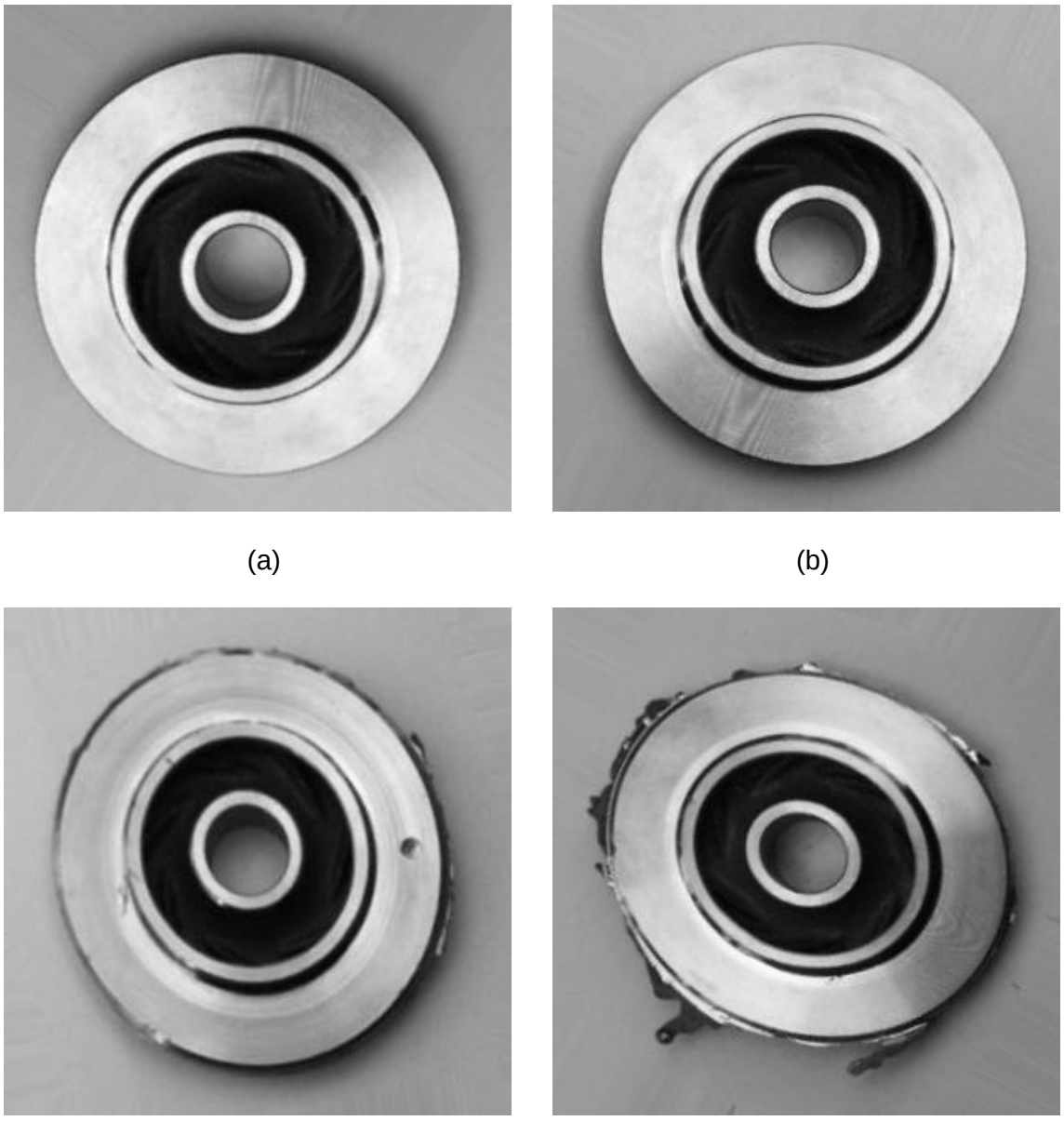
Las imágenes del *Dataset* se dividen en dos categorías diferentes, defectuosas y no defectuosas. Hay un total de 3037 imágenes no defectuosas y 4167 imágenes defectuosas. Estos datos se recogen en la Tabla 3.1.

	Número de imágenes
Defectuosas	3037
No defectuosas	4167
Total	7204

Tabla 3.1 Clases de las imágenes del *Dataset*

En la Figura 3.1 se muestran cuatro ejemplos de imágenes empleadas, dos etiquetadas como no defectuosas (a y b) y otras dos imágenes como defectuosas (c y d). Como se puede observar en las imágenes defectuosas, estas no tienen un único tipo de defecto, sino que tienen varios, y el tamaño y visibilidad de estos varía, siendo algunos muy visibles como los defectos en los bordes de la pieza y otros menos como pequeñas muescas o rayaduras.

Este *Dataset* se ha dividido en diez conjuntos de imágenes en función del mecanismo de validación cruzada descrito en el apartado 2.2.4. Cada conjunto se divide en tres subconjuntos, entrenamiento, validación (para aplicar *early stopping*) y test. En la Tabla 3.2 se muestra el número de imágenes de cada subgrupo en cada iteración.



(a) (b)

(c) (d)

Figura 3.1: Ejemplos de imágenes del *Dataset* empleado: (a) y (b) no defectuosas y (c) y (d) defectuosas (Dabhi, 2020).

Grupo	Número de imágenes
Entrenamiento	5186
Validación (*early stopping*)	1297
Test	721

Tabla 3.2: Descripción del número de imágenes de cada grupo.

3.2 Preprocesado de las imágenes

En el campo de la visión por computación es muy común el uso de técnicas de preprocesado. Esta técnica se emplea para limpiar datos, integrarlos, transformarlos y reducirlos para facilitar y optimizar su posterior procesado. El conjunto de técnicas de preprocesado pueden agruparse en dos grandes grupos, las técnicas de preparación de datos y las de reducción de datos. La preparación de datos está formada por un grupo de técnicas cuya finalidad es iniciar correctamente los datos de entrada al modelo. Por otro lado, están las técnicas de reducción, las cuales tienen como objetivo reducir la representación de los datos originales, manteniendo el mayor grado posible de integridad e información existente en los datos originales (Herrera, 2016).

El preprocesado implementado en este TFM consta de los siguientes pasos:

1. Reducción del tamaño de la imagen: Como se explicó en el apartado anterior (3.1) el tamaño original de la imagen del *dataset* es de 300 x 300 píxeles, con la finalidad de reducir la carga computacional y el tiempo de entrenamiento y test del clasificador, se optó por redimensionar las imágenes de entrada a un tamaño de 100 x 100 píxeles.
2. Además, se realizó un re escalado de los pixeles de las imágenes dividiendo el valor de cada píxel (de 0 a 255) por 255, de manera que los valores de los píxeles pasan a tener un valor entre 0 y 1.
3. Por último, se normalizaron los pixeles de las imágenes con el objetivo de facilitar el procesado y extracción de características. La normalización consistió en restar la matriz de pixeles de las imágenes, tanto de entrenamiento, validación y test, entre la matriz con el valor medio de los pixeles de las imágenes únicamente del conjunto de entrenamiento. Al resultado de esa resta se dividió entre la matriz de la desviación típica de los pixeles de las imágenes de entrenamiento, la ecuación asociada a la normalización es la siguiente:

$$I_{Normalizada} = \frac{I - X_{\mu\_train}}{X_{\sigma\_train\_}} \quad \textbf{(3.1)}$$

Donde:

- I es la matriz que representa la imagen.
- X_{μ_train} es la matriz con la media de los pixeles del conjunto de imágenes de entrenamiento.
- $X_{\sigma\_train\_}$ se trata de la matriz con la desviación típica de los pixeles del conjunto de imágenes de entrenamiento.

3.3 Implementación del clasificador

Para el desarrollo del clasificador se optó por implementar una CNN, ya que como se comentó en el apartado 2.1.2, este tipo de redes neuronales están diseñado para trabajar con imágenes, consiguiendo generalmente un mejor rendimiento y resultados que otros tipos de redes. En este apartado se exponen las diferentes configuraciones

diseñadas para el modelo del clasificador, además se presenta la arquitectura del clasificador diseñado por el usuario GuillaumeSimler de Kaggle para realizar la tarea de clasificación sobre el mismo *dataset* y comparar con el mismo.

3.3.1 Clasificadores propios realizados en el TFM

La arquitectura base del clasificador diseñado en este trabajo es bastante sencilla en comparación con los modelos más populares, como VGG o ResNet. Se optó por una CNN de mayor sencillez debiado a que el número de imágenes disponibles no es muy elevado, unido a que las especificaciones del ordenador con el que se trabajó (el portátil personal) no dispone de unas especificaciones técnicas elevadas.

Se han realizado 6 configuraciones diferentes para el modelo del clasificador, las seis constan de cuatro capas convolucionales, además las capas de *pooling* utilizadas son de tipo *max-pooling* dado que este tipo reduce el efecto del ruido en las imágenes. Además, se emplearon capas de *dropout*. Dichas capas se emplean para implementar el método de *dropout*. El *dropout* es una de las principales técnicas de regulación de redes neuronales. Esta técnica consiste en congelar un porcentaje de las neuronas de una capa permitiendo reducir el riesgo de *overfitting* de la red neuronal. Las configuraciones implementadas se describen de forma más detallada a continuación.

3.3.1.1 Configuración 1

La arquitectura de esta configuración se muestra en la Figura 3.2.

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 100, 100, 16)      416
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 50, 50, 16)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 50, 50, 16)        2320
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 25, 25, 16)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 25, 25, 32)        4640
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 12, 12, 32)        0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 12, 12, 16)        12816
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 6, 6, 16)          0
_________________________________________________________________
flatten (Flatten)            (None, 576)               0
_________________________________________________________________
dense (Dense)                (None, 256)               147712
_________________________________________________________________
dropout (Dropout)            (None, 256)               0
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 257
=================================================================
Total params: 168,161
Trainable params: 168,161
Non-trainable params: 0
_________________________________________________________________
```

Figura 3.2: Arquitectura del clasificador de la configuración 1.

Las características de las capas convolucionales son:

- Primera capa convolucional: 16 filtros de tamaño 5x5 y función de activación ReLu.
- Primera capa de *Maxpooling*: Tamaño 2x2.
- Segunda capa convolucional: 16 filtros de tamaño 3x3 y función de activación ReLu.
- Segunda capa *Maxpooling*: Tamaño 2x2.
- Tercera capa convolucional: 32 filtros de tamaño 3x3 y función de activación ReLu.
- Tercera capa de *Maxpooling*: Tamaño 2x2.
- Cuarta capa convolucional: 16 filtros de tamaño 3x3 y función de activación ReLu.
- Cuarta capa *Maxpooling*: Tamaño 2x2.

Por otro lado, el perceptrón multicapa encargado presenta la siguiente arquitectura:

- Capa *flatten*.
- Primera capa dense: 256 neuronas con función de activación ReLu.
- Capa *dropout*: 0.5 de ratio.
- Capa dense de salida: 1 neurona y función de activación sigmoide.

3.3.1.2 Configuración 2

La arquitectura de la configuración 2 se muestra en la Figura 3.3:

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 100, 100, 16)      160
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 50, 50, 16)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 50, 50, 16)        2320
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 25, 25, 16)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 25, 25, 32)        4640
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 12, 12, 32)        0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 12, 12, 16)        4624
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 6, 6, 16)          0
_________________________________________________________________
flatten (Flatten)            (None, 576)               0
_________________________________________________________________
dense (Dense)                (None, 256)               147712
_________________________________________________________________
dropout (Dropout)            (None, 256)               0
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 257
=================================================================
Total params: 159,713
Trainable params: 159,713
Non-trainable params: 0
_________________________________________________________________
```

Figura 3.3: Arquitectura del clasificador de la configuración 2.

La configuración 2 se inspiró en el modelo VGG, la cual, en su momento, logró mejorar los resultados de los clasificadores previos reduciendo el tamaño del *kernel* de sus capas convolucionales a 3x3. De modo que, en esta configuración, el *kernel* de la capa convolucional se redujo de 5x5 a 3x3. Las características de las demás capas se mantienen igual que en la configuración 1. Además, si comparamos las dos arquitecturas, se puede observar que el número de parámetros entrenables en la configuración 2 se reducen respecto a la 1.

3.3.1.3 Configuración 3

La arquitectura de la configuración 3 puede observarse en la Figura 3.4.

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 100, 100, 32)      320
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 50, 50, 32)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 50, 50, 64)        18496
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 25, 25, 64)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 25, 25, 128)       73856
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 12, 12, 128)       0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 12, 12, 64)        73792
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 6, 6, 64)          0
_________________________________________________________________
flatten (Flatten)            (None, 2304)              0
_________________________________________________________________
dense (Dense)                (None, 256)               590080
_________________________________________________________________
dropout (Dropout)            (None, 256)               0
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 257
=================================================================
Total params: 756,801
Trainable params: 756,801
Non-trainable params: 0
_________________________________________________________________
```

Figura 3.4: Arquitectura del clasificador de la configuración 3.

3.3.1.4 Configuración 4

La arquitectura del clasificador para la configuración 4 se representa en la Figura 3.5:

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 100, 100, 32)      320
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 50, 50, 32)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 50, 50, 64)        18496
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 25, 25, 64)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 25, 25, 128)       73856
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 12, 12, 128)       0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 12, 12, 64)        73792
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 6, 6, 64)          0
_________________________________________________________________
flatten (Flatten)            (None, 2304)              0
_________________________________________________________________
dense (Dense)                (None, 256)               590080
_________________________________________________________________
dropout (Dropout)            (None, 256)               0
_________________________________________________________________
dense_1 (Dense)              (None, 128)               32896
_________________________________________________________________
dropout_1 (Dropout)          (None, 128)               0
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 129
=================================================================
Total params: 789,569
Trainable params: 789,569
Non-trainable params: 0
_________________________________________________________________
```

Figura 3.5: Arquitectura del clasificador de la configuración 4.

Hasta ahora, en todas las configuraciones que se expusieron únicamente se realizaron cambios en las capas convolucionales, por ello, en la configuración 4 se decidió modificar la capa de clasificación (el perceptrón multicapa), añadiendo una capa densa adicional seguida de la primera de las anteriores configuraciones. Además, esta nueva capa convolucional se acompañó de una capa de *dropout* de ratio de descarte de 0.2. Por último, el ratio de la primera capa de *dropout* se redujo de 0.5 a 0.2.

Esta configuración presenta un incremento considerable del número de parámetros entrenables con respecto a las anteriores, aunque no llega a pasar del millón.

3.3.1.5 Configuración 5

La configuración 5 se implementó con el objetivo de evaluar que efecto sobre el clasificador tiene reducir la complejidad del perceptrón multicapa en este problema. Para ello, se eliminó la segunda capa densa y la segunda capa de *dropout* de la configuración 4. Además, se modificó el número de neuronas de la primera capa densa, reduciéndolo de 256 a 64 neuronas. Con estos cambios se consigue una reducción importante de parámetros con respecto a la configuración 4, pero aun así, el número de parámetros sigue siendo mayor que en las primeras configuraciones. En la Figura 3.6 se muestra la arquitectura para esta configuración.

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 100, 100, 32)      320
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 50, 50, 32)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 50, 50, 64)        18496
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 25, 25, 64)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 25, 25, 128)       73856
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 12, 12, 128)       0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 12, 12, 64)        73792
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 6, 6, 64)          0
_________________________________________________________________
flatten (Flatten)            (None, 2304)              0
_________________________________________________________________
dense (Dense)                (None, 64)                147520
_________________________________________________________________
dropout (Dropout)            (None, 64)                0
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 65
=================================================================
Total params: 314,049
Trainable params: 314,049
Non-trainable params: 0
_________________________________________________________________
```

Figura 3.6: Arquitectura del clasificador de la configuración 5.

3.3.1.6 Configuración 6

La configuración 6, al igual que la 2, está inspirada en algunas de las arquitecturas más populares de clasificadores de objetos como VGG o Resnet, las cuales presentan varias capas convolucionales seguidas. De modo que en esta configuración se decidió eliminar la capa de *max-pooling* que hay entre la segunda capa convolucional y la tercera, de manera que dichas capas se concatenen. En la Figura 3.7 se muestran las capas de la configuración 6.

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 100, 100, 32)      320
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 50, 50, 32)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 50, 50, 64)        18496
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 50, 50, 128)       73856
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 25, 25, 128)       0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 25, 25, 64)        73792
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 12, 12, 64)        0
_________________________________________________________________
flatten (Flatten)            (None, 9216)              0
_________________________________________________________________
dense (Dense)                (None, 64)                589888
_________________________________________________________________
dropout (Dropout)            (None, 64)                0
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 65
=================================================================
Total params: 756,417
Trainable params: 756,417
Non-trainable params: 0
_________________________________________________________________
```

Figura 3.7: Arquitectura del clasificador de la configuración 6.

3.3.2 Clasificador de GuillaumeSimler

En este trabajo también se implementó el modelo de clasificador diseñado por el usuario de *Kaggle* GuillaumeSimler (GuillaumeSimler, 2020) para compararlo con la mejor de las configuraciones del modelo propuestas en este TFM. La arquitectura del clasificador se muestra en la Figura 3.8. La arquitectura diseñada por este usuario también sencilla, pero tiene más parámetros de entrenamiento que cualquiera de las configuraciones propuestas en este trabajo. La base convolucional de esta arquitectura consta de dos capas convolucionales, cada una de ellas seguida por una capa de *max-pooling*, Las características de la base convolucional son:

- Primera capa convolucional: 16 filtro de tamaño 3x3 y función de activación ReLu.
- Primera capa de *Maxpooling*: Tamaño 2x2.
- Segunda capa convolucional: 16 filtros de tamaño 3x3 y función de activación ReLu.
- Segunda capa *Maxpooling*: Tamaño 2x2.

Por último, el perceptrón multicapa consta de una capa *flatten* seguida de una capa de *dropout*, dos capas *fully-connected* cada una de ellas seguida por una capa de *dropout* y una capa densa de salida. Las características de estas capas son las siguientes:

- Primera capa densa: Consta de 128 neuronas con una función de activación ReLu.

- Primera capa de *Dropout*: Configurada para un ratio de 0.2 (20%).
- Srgunda capa densa: Consta de 128 neuronas con ReLu como función de activación.
- Segunda capa de *Dropout*: Configurada para un ratio de 0.2 (20%).
- Tercera capa densa: Consta de 64 neuronas con ReLu como función de activación.
- Tercera capa de *Dropout*: Configurada para un ratio de 0.2 (20%).
- Capa de clasificación: Tiene 1 capa de activación y una función sigmoide como función de activación.

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 98, 98, 16)        160
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 49, 49, 16)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 47, 47, 16)        2320
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 23, 23, 16)        0
_________________________________________________________________
flatten (Flatten)            (None, 8464)              0
_________________________________________________________________
dense (Dense)                (None, 128)               1083520
_________________________________________________________________
dropout (Dropout)            (None, 128)               0
_________________________________________________________________
dense_1 (Dense)              (None, 128)               16512
_________________________________________________________________
dropout_1 (Dropout)          (None, 128)               0
_________________________________________________________________
dense_2 (Dense)              (None, 64)                8256
_________________________________________________________________
dropout_2 (Dropout)          (None, 64)                0
_________________________________________________________________
dense_3 (Dense)              (None, 1)                 65
=================================================================
Total params: 1,110,833
Trainable params: 1,110,833
Non-trainable params: 0
```

Figura 3.8: Arquitectura del clasificador de GuillaumeSimler.

3.4 Resultados

En este apartado se exponen los resultados obtenidos para los modelos descritos en el apartado anterior. Para evaluar y comparar los resultados de los diferentes modelos se optó por realizar una validación cruzada de 10 iteraciones. En primer lugar, se exponen y comparan los resultados de las seis configuraciones del clasificador diseñado para este trabajo. Por último, se exponen los resultados del clasificador de GuillaumeSimler y se realiza una comparación entre los resultados de este modelo y la mejor configuración de las anteriores.

3.4.1 Evaluación de las configuraciones del clasificador desarrollado en el TFM

En primer lugar, se presentarán tablas con los resultados obtenidos en cada una de las 10 iteraciones del proceso de validación cruzada con cada una de las configuraciones y posteriormente, en la sección 3.4.1.7, se representarán gráficas generadas a partir de dichos valores para comparar las distintas configuraciones. Las métricas analizadas son las descritas en la sección 2.2.1, *accuracy* (Acc), AUC, precisión (Prec), *recall* (Rec), F1 *score* y pérdida de validación (Val_loss).

3.4.1.1 Configuración 1

	K=1	K=2	K=3	K=4	K=5	K=6	K=7	K=8	K=9	K=10
Acc	0.9847	0.9944	0.9875	0.9889	0.9847	0.9916	0.9917	0.9931	0.9861	0.9889
AUC	0.9988	0.9998	0.9968	0.9975	0.9972	0.9966	0.9992	0.9999	0.9979	0.9997
Prec	0.9927	0.9976	0.9928	0.9881	0.9975	0.9951	0.9975	0.9904	0.9833	0.9903
Rec	0.9808	0.9928	0.9856	0.9928	0.9760	0.9904	0.9880	0.9975	0.9928	0.9903
F1	0.9867	0.9952	0.9892	0.9904	0.9867	0.9927	0.9927	0.9940	0.9880	0.9903
Val_loss	0.0479	0.0177	0.0742	0.0456	0.0713	0.0532	0.0369	0.0117	0.0512	0.0225

Tabla 3.3: Valores de las métricas obtenidas en cada una de las iteraciones (K=10) de la configuración 1.

3.4.1.2 Configuración 2

	K=1	K=2	K=3	K=4	K=5	K=6	K=7	K=8	K=9	K=10
Acc	0.9875	0.9944	0.9889	0.9958	0.9888	0.9902	0.9958	0.9930	0.9875	0.9972
AUC	0.9995	0.9999	0.9971	0.9974	0.9975	0.9975	0.9993	0.9996	0.9993	0.9998
Prec	0.9857	0.9975	0.9951	1.0	1.0	0.9975	0.9833	0.9904	0.9833	1.0
Rec	0.9928	0.9928	0.9856	0.9928	0.9808	0.9856	0.9951	0.9975	0.9951	0.9952
F1	0.9892	0.9952	0.9903	0.9963	0.9903	0.9915	0.9892	0.9940	0.9892	0.9976
Val_loss	0.0316	0.0138	0.0498	0.0378	0.0576	0.0381	0.0418	0.0242	0.0418	0.0132

Tabla 3.4: Valores de las métricas obtenidas en cada una de las iteraciones (K=10) de la configuración 2.

3.4.1.3 Configuración 3

	K=1	K=2	K=3	K=4	K=5	K=6	K=7	K=8	K=9	K=10
Acc	0.9902	0.9972	0.9889	0.9958	0.9930	0.9958	0.9972	0.9958	0.9958	0.9972
AUC	0.9996	0.9999	0.9989	0.9990	0.9993	0.9992	0.9995	0.9999	0.9996	0.9998
Prec	0.9904	0.9976	0.9952	1.0	1.0	1.0	1.0	0.9952	0.9952	0.9975
Rec	0.9928	0.9976	0.9856	0.9928	0.9880	0.9928	0.9952	0.9975	0.9976	0.9975
F1	0.9916	0.9976	0.9904	0.9963	0.9939	0.9963	0.9975	0.9964	0.9964	0.9975
Val_loss	0.0231	0.0086	0.0409	0.0259	0.0249	0.0204	0.0169	0.0167	0.0202	0.0126

Tabla 3.5: Valores de las métricas obtenidas en cada una de las iteraciones (K=10) de la configuración 3.

3.4.1.4 Configuración 4

	K=1	K=2	K=3	K=4	K=5	K=6	K=7	K=8	K=9	K=10
Acc	0.9916	0.9944	0.9930	0.9944	0.9902	0.9930	0.9931	0.9944	0.9833	0.9958
AUC	0.9998	0.9999	0.9985	0.9984	0.9988	0.9987	0.9996	0.9999	0.9993	0.9999
Prec	0.9928	1.0	0.9951	0.9975	1.0	0.9975	0.9928	0.9951	0.9809	0.9952
Rec	0.9928	0.9904	0.9928	0.9928	0.9832	0.9904	0.9952	0.9951	0.9904	0.9975
F1	0.9929	0.9951	0.9939	0.9952	0.9915	0.9939	0.9940	0.9951	0.9856	0.9964
Val_loss	0.0152	0.0153	0.0276	0.0263	0.0408	0.0334	0.0204	0.0152	0.0375	0.0125

Tabla 3.6: Valores de las métricas obtenidas en cada una de las iteraciones (K=10) de la configuración 4.

Configuración 5

3.4.1.5

	K=1	K=2	K=3	K=4	K=5	K=6	K=7	K=8	K=9	K=10
Acc	0.9944	0.9958	0.9902	0.9958	0.9889	0.9916	0.9944	0.9972	0.9888	0.9972
AUC	0.9998	0.9997	0.9962	0.9969	0.9976	0.9974	0.9997	0.9999	0.9993	0.9997
Prec	0.9952	0.9976	0.9975	1.0	0.9975	0.9975	0.9975	0.9952	0.9951	0.9975
Rec	0.9952	0.9952	0.9856	0.9928	0.9832	0.9880	0.9928	1.0	0.9855	0.9975
F1	0.9952	0.9963	0.9915	0.9964	0.9903	0.9928	0.9951	0.9976	0.9903	0.9976
Val_loss	0.0202	0.0119	0.0508	0.0304	0.0524	0.0386	0.0210	0.0140	0.0335	0.0122

Tabla 3.7: Valores de las métricas obtenidas en cada una de las iteraciones (K=10) de la configuración 5.

3.4.1.6 Configuración 6

	K=1	K=2	K=3	K=4	K=5	K=6	K=7	K=8	K=9	K=10
Acc	0.9944	0.9958	0.9889	0.9930	0.9930	0.9958	0.9944	0.9902	0.9944	0.9986
AUC	0.9999	0.9999	0.9987	0.9969	0.9977	0.9984	0.9993	0.9997	0.9999	0.9999
Prec	0.9904	0.9952	0.9951	0.9975	1.0	1.0	0.9975	0.9834	0.9975	1.0
Rec	1.0	0.99760	0.9856	0.9904	0.9880	0.9928	0.9928	1.0	0.9927	0.9975
F1	0.9952	0.9964	0.9903	0.9939	0.9939	0.9963	0.9952	0.9916	0.9952	0.9987
Val_loss	0.0133	0.0078	0.0411	0.0333	0.0476	0.0270	0.0262	0.0239	0.0114	0.0070

Tabla 3.8: Valores de las métricas obtenidas en cada una de las iteraciones (K=10) de la configuración 6.

3.4.1.7 Comparación entre las 6 configuraciones

Una vez recopilados los resultados de cada configuración, se realiza una comparación entre las configuraciones con el objetivo de escoger el modelo que mejor funciona para la clasificación de imágenes defectuosas. Para que la comparación sea más sencilla y visual, se representan los *boxes plot* o diagramas de cajas de cada métrica de las diferentes configuraciones.

Pérdida en la validación (Val_loss):

En la Figura 3.9 se representan los diagramas de cajas de la pérdida de validación de las diferentes configuraciones expuestas anteriormente. Además, en la Tabla 3.9 se recogen los valores medios y la desviación estándar de esta métrica para cada configuración de modo que nos facilite la comparación.

Al analizar el diagrama de cajas y la tabla de valores medios y desviación, se puede observar que los resultados proporcionados por todos los clasificadores son muy buenos, consiguiendo todos ellos unas pérdidas medias inferiores a 0.05. El mejor de los valores medios obtenidos durante el proceso de validación cruzada fue la configuración 3, aunque la diferencia con las demás es muy pequeña. El diagrama de cajas de la configuración 3 presenta una distribución simétrica, aunque cabe destacar que presenta un valor atípico, es decir, uno de los valores obtenidos de la pérdida de validación es se localiza fuera de los valores del diagrama de cajas. De todas formas, teniendo en cuenta que el diagrama de cajas se representa a partir de solo 10 medidas (correspondientes a cada una de las 10 iteraciones del proceso de validación cruzada) es habitual que se produzcan este tipo de situaciones.

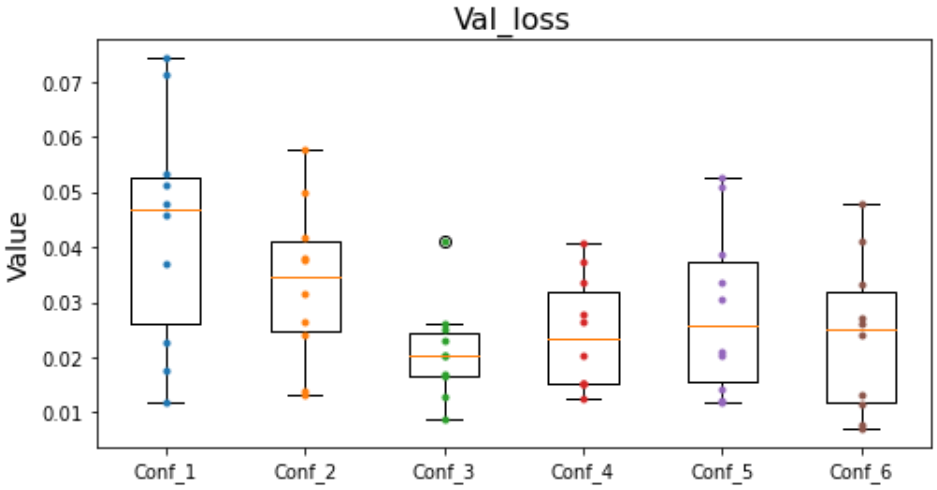


Figura 3.9: Comparativa entre los boxes plots de la pérdida de validación de las diferentes configuraciones.

	Val_loss	
	Media	Desviación estándar
Conf_1	0.0432	0.0201
Conf_2	0.0334	0.0137
Conf_3	**0.0210**	**0.0083**
Conf_4	0.0244	0.0097
Conf_5	0.0285	0.0144
Conf_6	0.0239	0.0133

Tabla 3.9: Media y desviación típica de la pérdida de validación de las seis configuraciones durante la validación cruzada.

Accuracy:

fijándonos en la Figura 3.10, en la cual se representan los *boxes plots* de la *accuracy* de las seis configuraciones durante la validación cruzada, se puede apreciar que en lo que respecta a la *accuracy* el mejor valor lo consiguió la configuración número tres, pero al igual que ocurrió con la pérdida de validación presenta un valor atípico. Esta configuración presenta una distribución asimétrica para esta métrica.

Los resultados anteriores obtenidos de la observación de los *boxes plots* se reafirman observando la Tabla 3.10, en la cual se recogen los valores de la media y la desviación típica de la *accuracy* de los diferentes modelos.

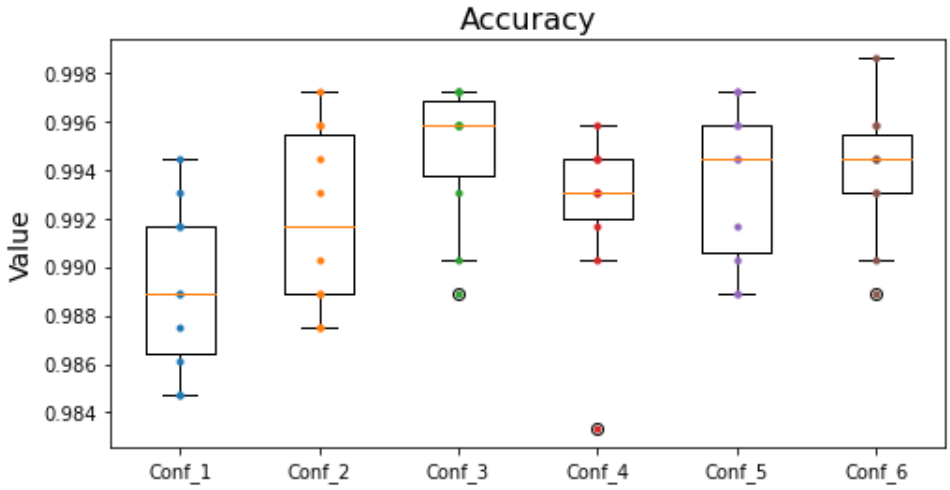


Figura 3.10: Comparativa entre los boxes plots de la *accuracy* de las diferentes configuraciones.

	Accuracy	
	Media	Desviación estándar
Conf_1	0.9891	0.0033
Conf_2	0.9919	0.0036
Conf_3	**0.9947**	**0.0028**
Conf_4	0.9923	0.0035
Conf_5	0.9934	0.0031
Conf_6	0.9938	0.0026

Tabla 3.10: Media y desviación típica de la *accuracy* de las seis configuraciones durante la validación cruzada.

Área bajo la curva (AUC):

Analizando los diagramas de cajas del área bajo la curva mostrado en la Figura 3.11 se puede apreciar que de nuevo, para esta métrica los mejores resultados son alcanzados por la configuración 3, puesto que en términos de valor medio es mayor que el resto de configuraciones y, además, tiene una desviación menor, los valores exactos pueden observarse en la Tabla 3.11.

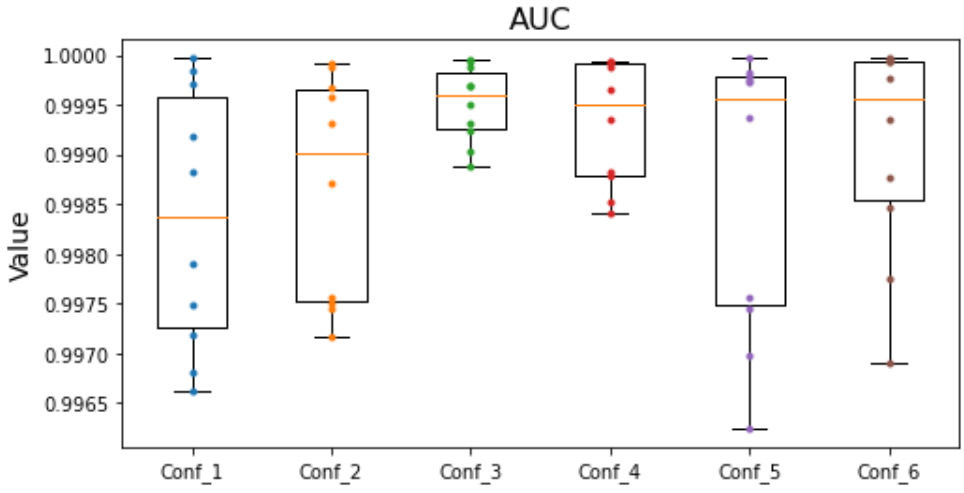


Figura 3.11: Comparativa entre los boxes plots de la AUC de las diferentes configuraciones.

	AUC	
	Media	Desviación estándar
Conf_1	0.9983	0.0012
Conf_2	0.9986	0.0010
Conf_3	**0.9995**	**0.0003**
Conf_4	0.9993	0.0005
Conf_5	0.9986	0.0013
Conf_6	0.9991	0.0010

Tabla 3.11: Media y desviación típica de la AUC de las seis configuraciones durante la validación cruzada.

Precisión:

En términos de precisión, la configuración 3 vuelve a conseguir los mejores resultados, pero por muy poca diferencia respecto a las configuraciones 2, 5 y 6 tal y como puede apreciarse al comparar los valores medios de la Tabla 3.12. La razón por la cual se escogió la configuración 3 por delante de las demás cuyos valores son similares fue porque la simetría de la configuración 3 es mejor que las de las otras, esto puede apreciarse observando los diagramas de cajas de la precisión que se representan en la Figura 3.12. La única que podría dar lugar a duda es la configuración 5, pero como esta presenta un valor atípico se antepuso la configuración 3, además, de que esta última es la que está consiguiendo los mejores resultados en todas las métricas.

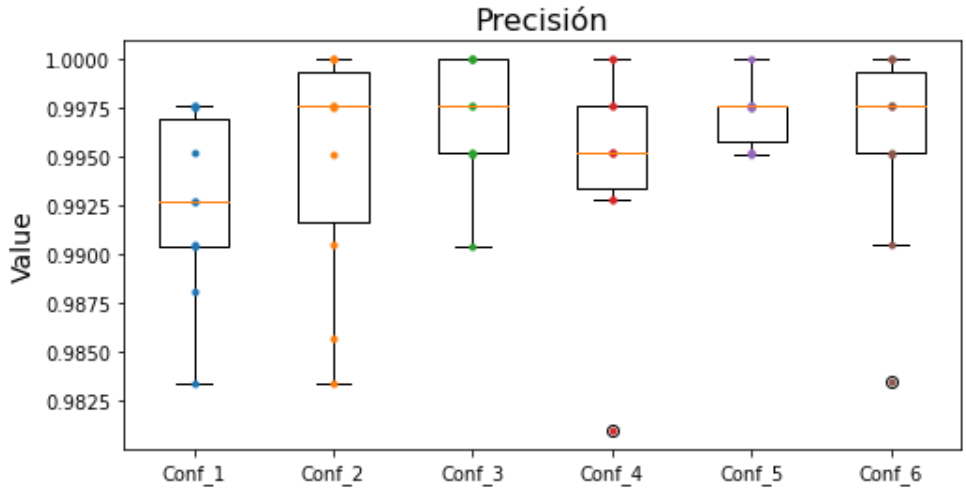


Figura 3.12: Comparativa entre los boxes plots de la precisión de validación de las diferentes configuraciones

	Precisión	
	Media	Desviación estándar
Conf_1	0.9925	0.0044
Conf_2	0.9947	0.0058
Conf_3	**0.9971**	**0.0029**
Conf_4	0.9947	0.0051
Conf_5	**0.9971**	**0.0014**
Conf_6	0.9957	0.0022

Tabla 3.12: Media y desviación típica de la precisión de las seis configuraciones durante la validación cruzada.

Recall:

Analizando los boxes plots de la *recall* que se representan en la Figura 3.13 se puede observar que los mejores resultados medios los logra, otra vez, la configuración 3, pero presenta una fuerte asimetría, así como un valor atípico. El resto de las configuraciones logran resultados un poco inferiores a la 3, pero al igual que esta todas tienen cierta asimetría. En la Tabla 3.13 se recogen los valores medios y la desviación de cada una de las configuraciones.

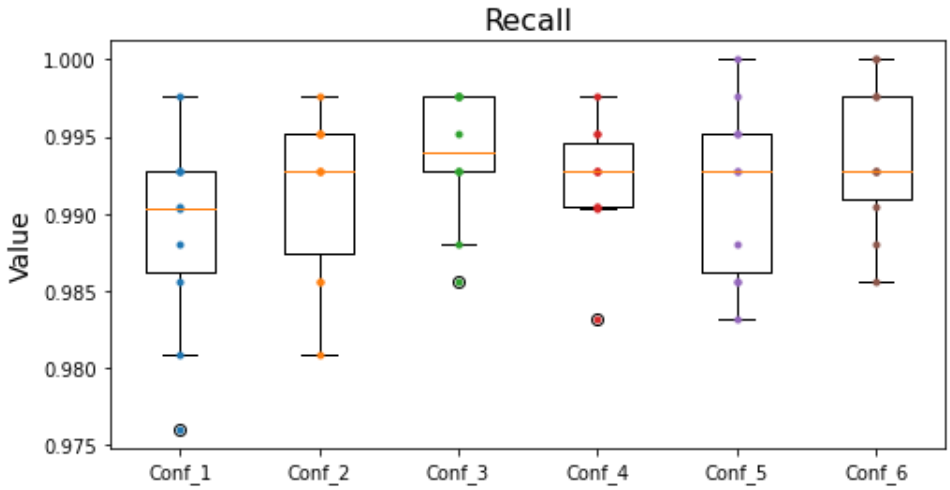


Figura 3.13: Comparativa entre los boxes plots de la *recall* de las diferentes configuraciones

	Recall	
	Media	Desviación estándar
Conf_1	0.9887	0.0060
Conf_2	0.9913	0.0051
Conf_3	**0.9937**	**0.0040**
Conf_4	0.9920	0.0037
Conf_5	0.9916	0.0053
Conf_6	**0.9937**	**0.0046**

Tabla 3.13: Media y desviación típica de la *recall* de las seis configuraciones durante la validación cruzada.

F1:

Observando los *boxes plots* de la F1 representados en la Figura 3.14 se puede apreciar que el valor medio más alto lo logra la configuración 3, aunque al igual que ocurrió con la *recall*, esta también presenta una asimetría y un valor atípico.

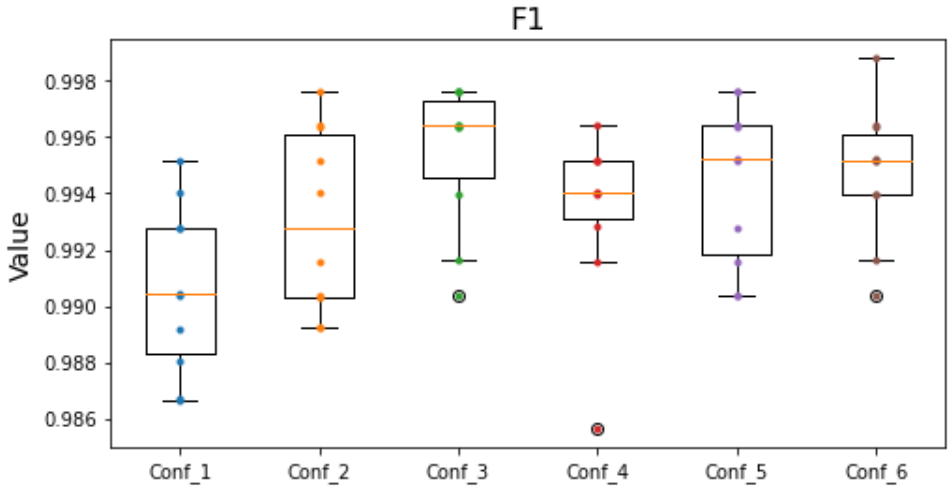


Figura 3.14: Comparativa entre los boxes plots del F1 de las diferentes configuraciones

	F1	
	Media	Desviación estándar
Conf_1	0.9906	0.0028
Conf_2	0.9930	0.0030
Conf_3	**0.9954**	**0.0024**
Conf_4	0.9934	0.0028
Conf_5	0.9943	0.0027
Conf_6	0.9947	0.0022

Tabla 3.14: Media y desviación típica del F1 de las seis configuraciones durante la validación cruzada.

En resumen, basándonos en los resultados obtenidos se consideró que la mejor configuración es la 3, aunque todas ellas consiguen unos resultados muy buenos para un clasificador. A continuación, se compara el clasificador 3 con el clasificador descrito en el apartado 3.3.2.

3.4.2 Comparación con el clasificador de GuillaumeSimler

Tras analizar en el subapartado anterior los resultados de las diferentes configuraciones del clasificador diseñado en este TFM, se compararon los resultados obtenidos por la mejor de las configuraciones con el clasificador diseñado por el usuario de *Kaggle* GuillaumeSimler. La arquitectura de este modelo se describió en el subapartado 3.3.2.

Este modelo se evaluó de la misma forma que se hizo con el clasificador propio con la finalidad de que los resultados obtenidos fueran comparables. En la tabla se recogen los valores de las métricas obtenidas por este clasificador en cada una de las iteraciones de la validación cruzada.

	K=1	K=2	K=3	K=4	K=5	K=6	K=7	K=8	K=9	K=10
Acc	0.9931	0.9986	0.9930	0.9930	0.9875	0.9902	0.9930	0.9944	0.9944	0.99861
AUC	0.9998	0.9999	0.9991	0.9993	0.9986	0.9993	0.9994	0.9999	0.9998	1.0
Prec	0.9928	0.9976	0.9951	0.9952	1.0	0.9951	0.9975	0.9904	0.9928	0.9976
Rec	0.9952	1.0	0.9928	0.9928	0.9784	0.9880	0.9904	1.0	0.9975	1.0
F1	0.9940	0.9988	0.9939	0.9939	0.9890	0.9916	0.9939	0.9952	0.9952	0.9987
Val_loss	0.0178	0.0098	0.0294	0.0255	0.0786	0.0295	0.0209	0.0111	0.0174	0.0063

Tabla 3.15: Valores de las métricas obtenidas en cada una de las iteraciones (K=10) del modelo de GuillaumeSimler.

Para comparar los resultados obtenidos por el mejor de los clasificadores implementados en este TFM con el modelo de GuillaumeSimler se representaron los *boxes plots* de las diferentes métricas para cada modelo, de modo que se obtienen los *boxes plots* que se representan en la Figura 3.15, además se computaron y recogieron las medias y desviaciones de cada métrica para los dos modelos en la Tabla 3.16.

Analizando los *boxes plots* se puede observar que los valores medios y desviaciones son muy parecidos entre ambos modelos, pero ligeramente superiores para la configuración 3. Por otro lado, ambos modelos presentan puntos atípicos, aunque estos son más pronunciados en el caso del modelo GuillaumeSimler.

		Configuración 3	Modelo GuillaumeSimler
Val_loss	**Media**	**0.021063821**	0.024673475
	Desviación estándar	0.008375092	0.019519083
Acc	**Media**	**0.99472606**	0.9936142
	Desviación estándar	0.002828074	0.0031791115
AUC	**Media**	0.9995109	**0.9995432**
	Desviación estándar	0.0003606369	0.00043117392
Prec	**Media**	**0.9971193**	0.995447
	Desviación estándar	0.0029929902	0.0027073636
Recall	**Media**	**0.99376315**	0.99352455
	Desviación estándar	0.00403968	0.00643883
F1	**Media**	**0.9954318**	0.99446785
	Desviación estándar	0.0024533	0.002771441

Tabla 3.16: Media y desviación de las métricas obtenidas por la configuración 3 y el modelo de GuillaumeSimler.

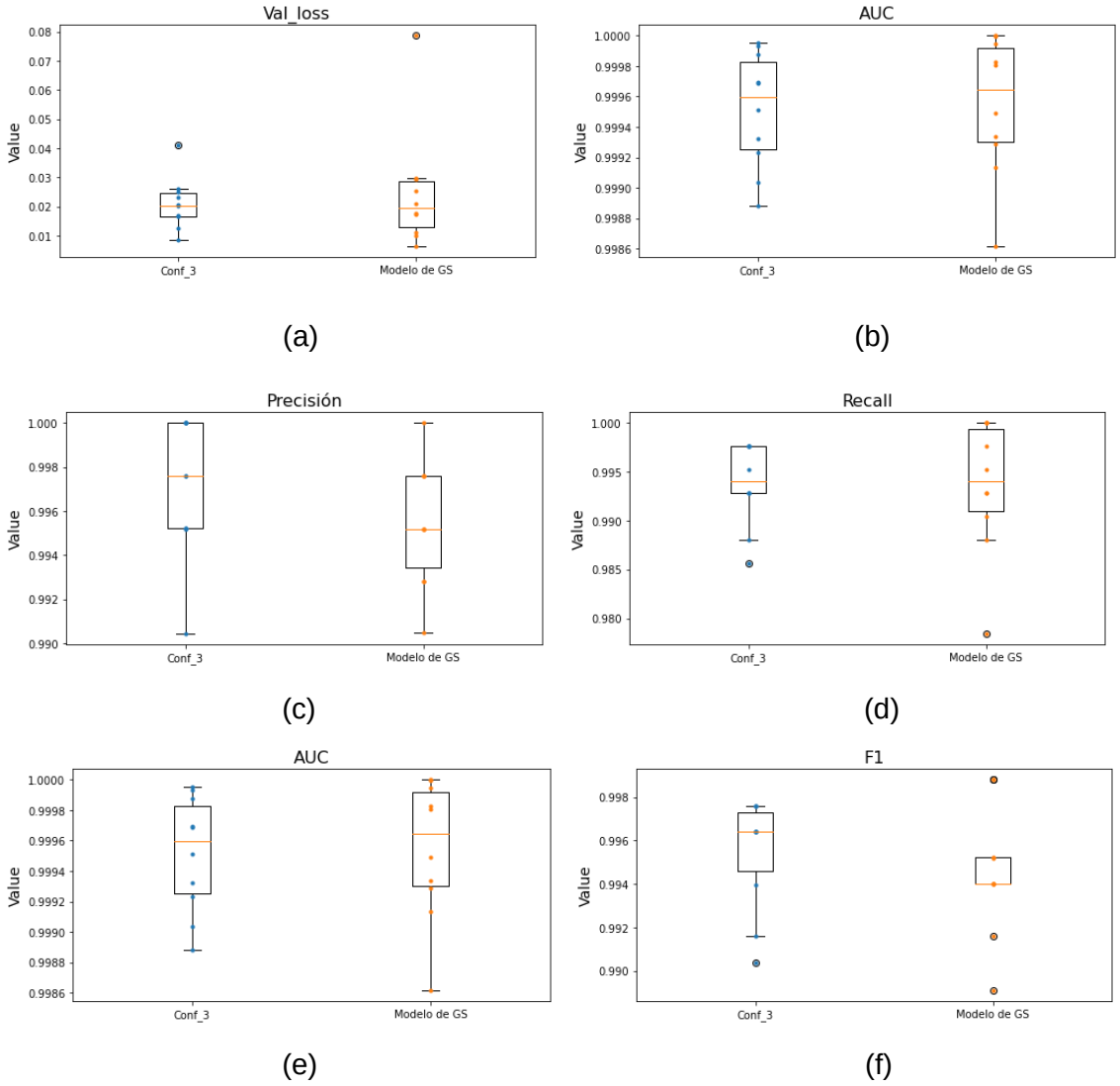


Figura 3.15: Comparativa de los *boxes plots* de las métricas (a) pérdida de validación, (b) accuracy, (c) precisión, (d) *recall*, (e) área bajo la curva y (f) F1.

Aunque según se observa en la Tabla 3.16, la configuración 3 parece obtener resultados ligeramente mejores en prácticamente todas las métricas que el modelo de GuillaumeSimler, la diferencia no es estadísticamente significativa, por lo que no puede afirmarse que lo supere en ese sentido. Ahora bien, sí se puede afirmar que se trata de un modelo más simple que el de GuillaumeSiler (pues tiene 756801 parámetros frente a 1110833).

3.4.3 Generación del modelo final

Por último, una vez comparado las diferentes configuraciones y modelos, y haber obtenido cuál de las configuraciones alcanzó los mejores resultados, se procedió a generar el modelo final. El objetivo de este subapartado es obtener los resultados del modelo que se mostrarían a la empresa interesada en el desarrollo del clasificador. Estos resultados no son fiables en comparación con los resultados del apartado anterior.

Estos resultados se han obtenido dividiendo la base de datos en dos subconjuntos, uno de entrenamiento y otro de validación, con una relación 80% y 20% respectivamente. La CNN se entrenó con las imágenes de entrenamiento (5763 imágenes) y se evaluó el entrenamiento con el conjunto de validación (1441 imágenes). Tras entrenar la red, se predicen los datos del conjunto de validación y con esas predicciones se obtienen los resultados finales del modelo.

La evolución de los valores de la pérdida y exactitud durante el entrenamiento de la red se observan en la Figura 3.16, donde se puede observar que se logran valores muy buenos, con una exactitud en torno a 0.99 y una pérdida en torno a 0.015.

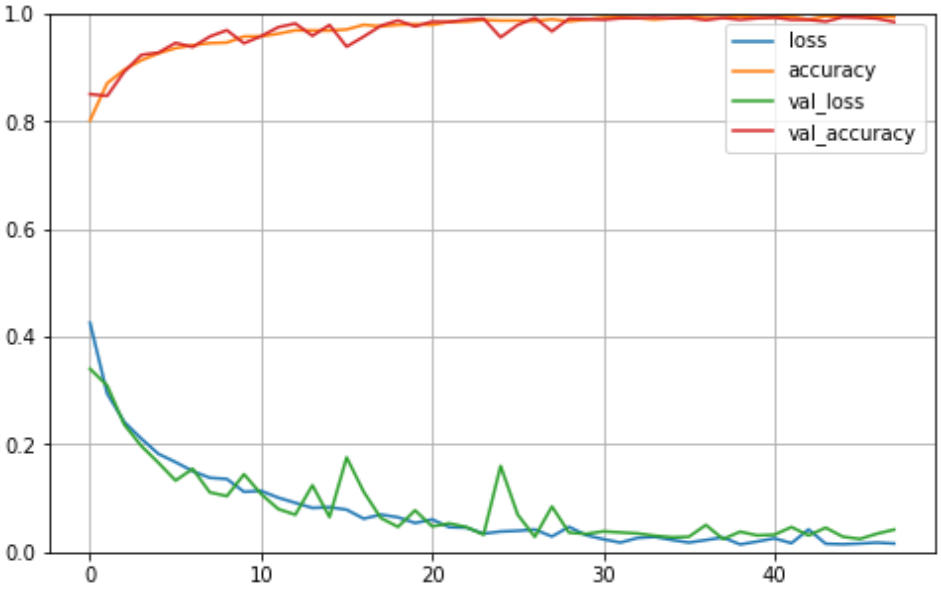


Figura 3.16: Evolución de la perdida y la exactitud (*accuracy*) durante el entrenamiento y validación del clasificador.

Una vez el clasificador es entrenado, se procedió a probar el rendimiento de este, para ello se empleó el conjunto de datos de validación. Obteniendo la matriz de confusión que se muestra en la Tabla 3.17, donde la clase defectuosa se considera la positiva, mientras que la no defectuosa es considerada como la negativa.

		Predicciones	
		Defectuosa	No defectuosa
Observaciones	Defectuosa	826	8
	No defectuosa	3	604

Tabla 3.17: Matriz de confusión del clasificador

Con la ayuda de la matriz de confusión se obtuvieron las métricas de clasificador para este conjunto de datos, los valores de las métricas se recogen en la Tabla 3.18.

Loss	Acc	AUC	Prec	Recall	F1
0.0241	0.9924	0.9995	0.9964	0.9904	0.9934

Tabla 3.18: Valores de las métricas para el testeo del clasificador

Los resultados obtenidos por el clasificador son muy buenos, alcanzando valores del 99% de exactitud. Además, analizando la matriz de confusión (Tabla 3.17) se puede observar que el clasificador ha logrado clasificar correctamente un total de 1430 imágenes de 1441, equivocándose únicamente en 11 imágenes. De las imágenes que clasifica incorrectamente, 3 se corresponden a los falsos negativos y las otras 8 restantes son los falsos positivos. En la Figura 3.17 se representan las 3 imágenes correspondientes a los falsos positivos y en la Figura 3.18 se los 8 falsos negativos.

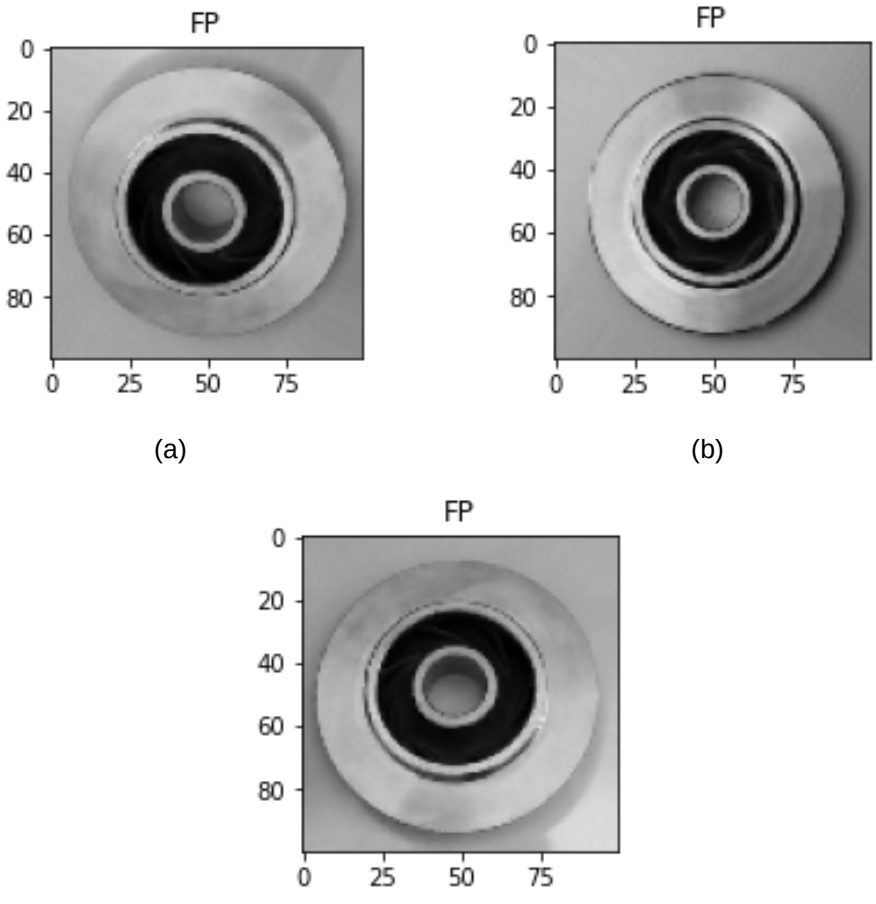
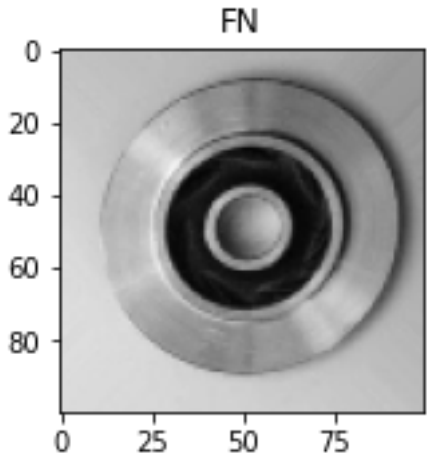
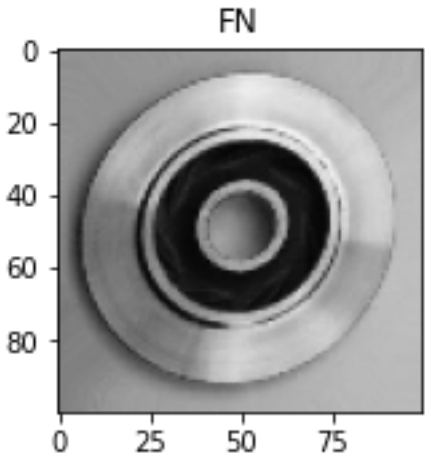


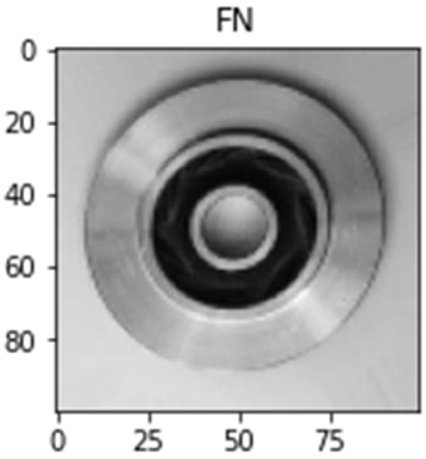
Figura 3.17: Imágenes correspondientes a los falsos positivos de la matriz de confusión.


FN
0
20
40
60
80
0
25
50
75

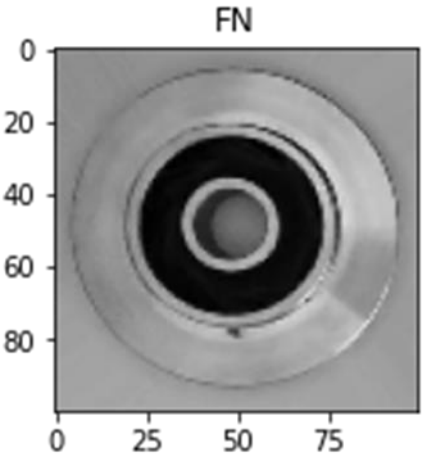
(a)


FN
0
20
40
60
80
0
25
50
75

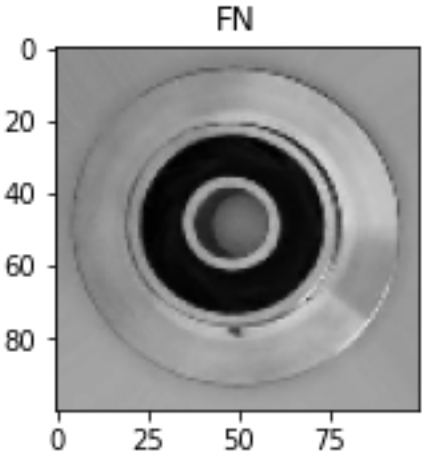
(b)


FN
0
20
40
60
80
0
25
50
75

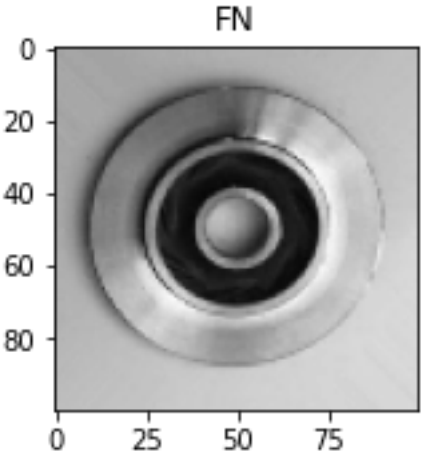
(c)


FN
0
20
40
60
80
0
25
50
75

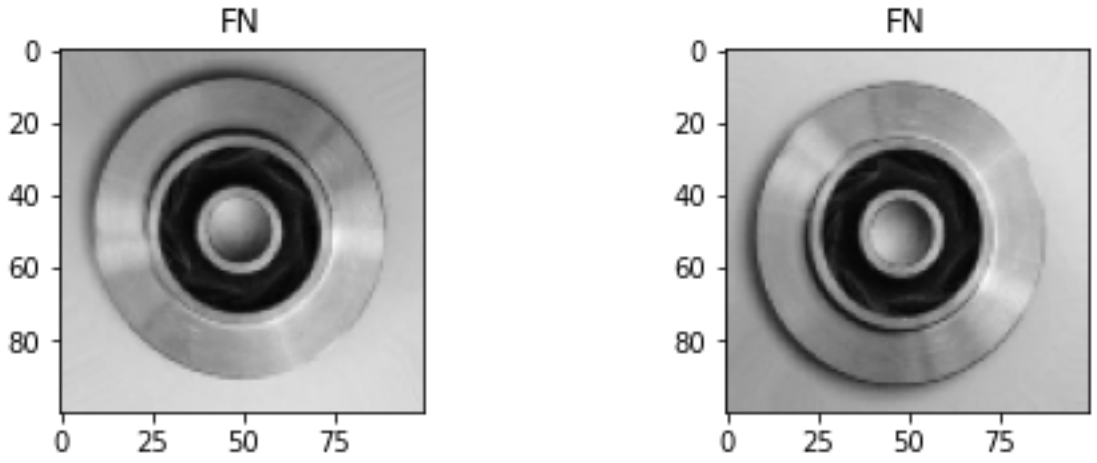
(d)


FN
0
20
40
60
80
0
25
50
75

(e)


FN
0
20
40
60
80
0
25
50
75

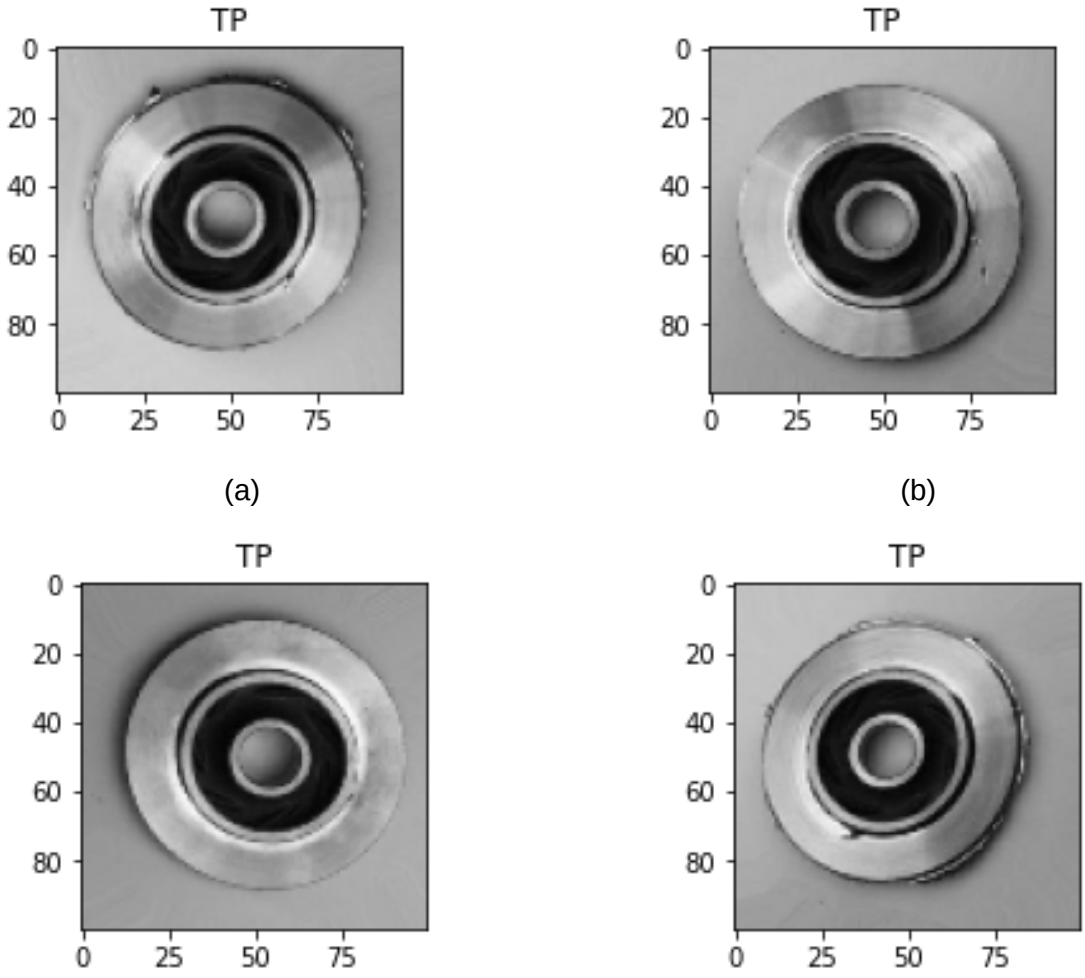
(f)



(g) (h)

Figura 3.18: Imágenes correspondientes a algunos falsos negativos de la matriz de confusión.

Además, en la Figura 3.19 se muestran cuatro ejemplos de verdaderos positivos y en la Figura 3.20 cuatro ejemplos de verdaderos negativos.



(a) (b)

(c) (d)

Figura 3.19: Imágenes correspondientes a algunos verdaderos positivos de la matriz de confusión.

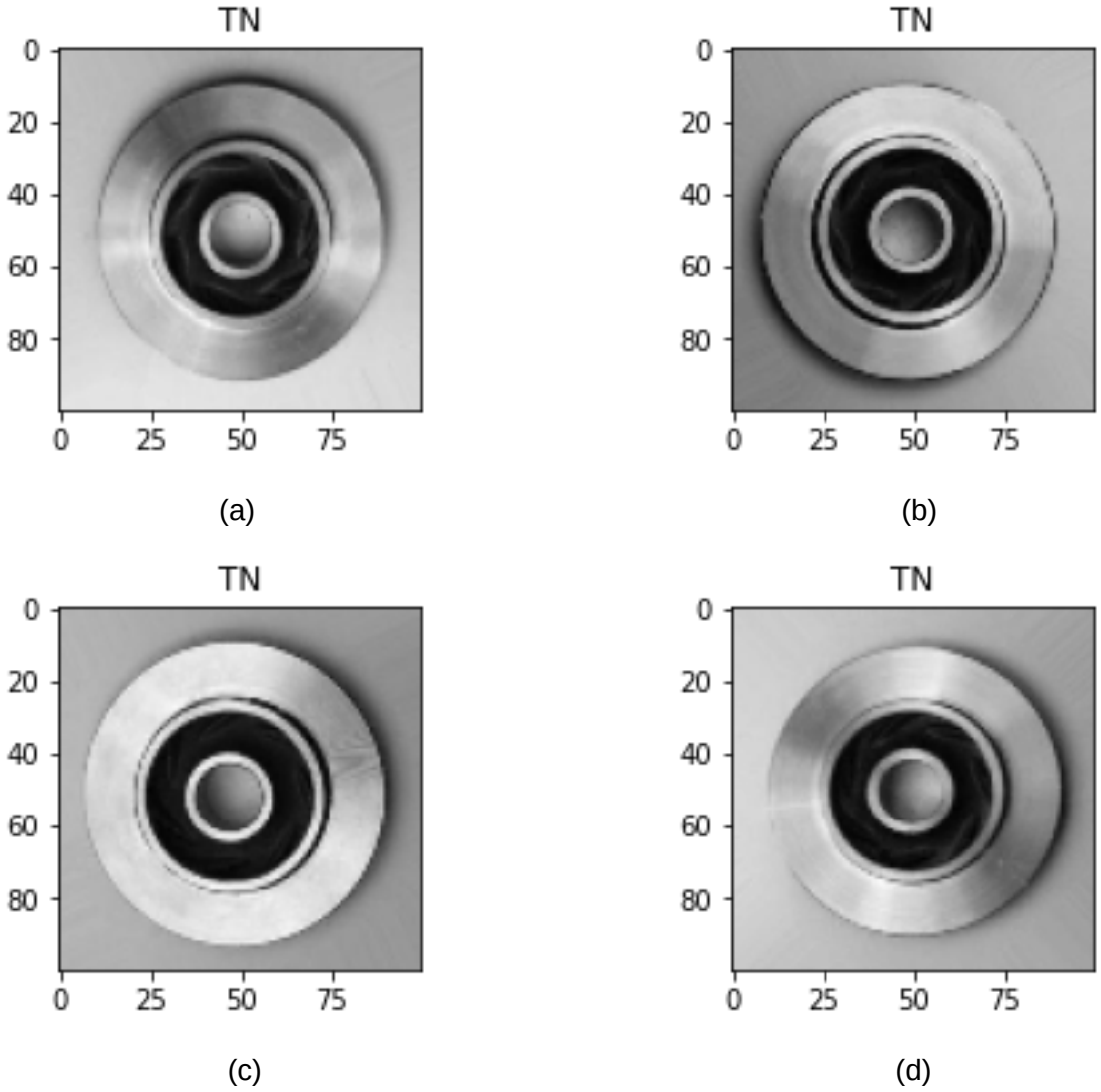


Figura 3.20: Imágenes correspondientes a algunos verdaderos negativos de la matriz de confusión.

Analizando las imágenes de los falsos negativos (Figura 3.18), se aprecia que la mayoría de ellas a simple vista parecen imágenes de piezas no defectuosas. Dichos defectos son inapreciables a simple vista.

Capítulo 4. Detector de objetos

La aparición de los clasificadores de imágenes supuso un gran avance en el campo de la visión por computación, aunque estos algoritmos presentan algunas limitaciones como problemas a la hora de clasificar aquellas imágenes que contienen más de un objeto y que, aunque el clasificador realice la clasificación de una imagen, no indica en que región de la imagen se encuentra el objeto clasificado. Por ello, en los últimos años el campo de la visión por computación se ha popularizado los algoritmos de detección de objetos, los cuales van un paso más allá de los clasificadores, permitiendo detectar varios objetos en una misma imagen, determinando qué clase de objeto es y localizarlos dentro de la imagen. La popularización de estos algoritmos radica en el amplio número de aplicaciones y potencial que tienen.

Este capítulo se centra en el diseño de los detectores de objetos orientados al ámbito industrial. En primer lugar, al igual que se hizo en el Capítulo 3, se describe la base de datos empleada para implementación de los detectores de objetos. A continuación, se definen las métricas que se emplean para evaluar los modelos y se introducen los dos detectores que se han empleado en este trabajo, el SSD y el RetinaNet. Por último, se exponen los resultados obtenidos para cada detector.

4.1 Dataset

Al tratarse de un algoritmo basado en *Deep Learning*, la utilización de un *Dataset* amplio y variado es clave para lograr un detector que funcione correctamente. Además, al tratarse de un método de aprendizaje supervisado se requieren etiquetas, en el caso de los detectores se precisa una etiqueta que indique la clase de cada objeto que aparece en la imagen y las coordenadas de la región donde se ubica cada uno.

En primer lugar, se pensó en emplear la misma base de datos que se utilizó en el desarrollo del clasificador, pero dicha base de datos no estaba preparada para la tarea de detección y el etiquetado de esta no era una tarea simple debido a ciertos tipos de defectos. Por ello se optó por buscar en diferentes repositorios de *Datasets* alguno que fuera más apropiado para esta tarea.

Tras una búsqueda exhaustiva, se encontró un repositorio de Github creado por la usuaria lxiaohuihuihui, el cual está asociado al *paper* de Ding, R. *et al.* (2019) donde colaboró. El *Dataset* consta de 10668 imágenes de 600 x 600 píxeles a color de placas de circuito impresos (*Printed Circuit Board*, PCB). Las imágenes pueden presentar 6 tipos diferentes de defectos generados sintéticamente: agujeros, mordedura de ratón, circuito abierto, cortocircuito, espurios y espurio de cobre. Además, cada imagen tiene asociado un fichero xml que contiene información acerca de la imagen, qué tipo de defectos aparecen en la imagen y en qué región de la imagen se localizan. El fichero xml sigue el formato PASCAL VOC. En la Figura 4.1 se muestra un ejemplo de un fichero de anotaciones xml con formato PASCAL VOC. Observando esta figura se puede apreciar que este fichero contiene diferente información acerca de la imagen pero la más importante para el detector son los objetos que aparecen en la imagen, los cuales se identifican con la etiqueta <object>, dentro de esta etiqueta se describe cuál es la clase a la que pertenece el objeto con la etiqueta <name> y la localización de este con las etiquetas <xmin> e <ymin> para indicar la esquina superior izquierda y <xmax> e <ymax> para la esquina inferior derecha del rectángulo que delimita al objeto. Otra información útil que contienen estos ficheros son el tamaño y profundidad de la imagen.

```xml
<?xml version="1.0" encoding="utf-8"?><annotation>
  <folder>JPEGImages</folder>
  <filename>light_01_spurious_copper_20_3_256</filename>
  <path>F:\PCB\PCB_DATASET_New\VOCPCB_800\_crop_600\JPEGImages\light_01_spurious_copper_20_3_256.jpg</path>
  <size>
    <width>600</width>
    <height>600</height>
    <depth>3</depth>
  </size>
  <segmented>0</segmented>
  <object>
    <name>spurious_copper</name>
    <pose>Unspecified</pose>
    <truncated>0</truncated>
    <difficult>0</difficult>
    <bndbox>
      <xmin>370</xmin>
      <ymin>349</ymin>
      <xmax>402</xmax>
      <ymax>419</ymax>
    </bndbox>
  </object>
  <object>
    <name>spurious_copper</name>
    <pose>Unspecified</pose>
    <truncated>0</truncated>
    <difficult>0</difficult>
    <bndbox>
      <xmin>360</xmin>
      <ymin>510</ymin>
      <xmax>444</xmax>
      <ymax>543</ymax>
    </bndbox>
  </object>
</annotation>
```

Figura 4.1: Fichero xml con las anotaciones de una imagen

Las 10668 imágenes del *Dataset* contienen distintos números y tipos de defectos. En la Tabla 4.1 se representan la distribución de los defectos en las diferentes clases.

	Número de instancias
Missing hole	3612
Mouse bite	3684
Open circuit	3548
Short	3508
Spur	3636
Spurious copper	3676
TOTAL	21664

Tabla 4.1: Número de instancias de cada tipo de defecto.

4.2 Métricas para evaluar el detector

Como se expuso en el apartado 2.2, para realizar la evaluación de un clasificador únicamente es necesario disponer de una etiqueta que indica si la clasificación ha sido correcta o no. Sin embargo, evaluar un detector de objetos es más complejo, ya que además de una etiqueta que indica la clase a la que pertenece, también es necesario indicar las coordenadas donde se localiza dicho objeto. La forma más común de indicar la localización de un objeto es mediante las coordenadas cartesianas de las esquinas opuestas del *bounding box* que contiene al objeto, es decir, las coordenadas de la esquina superior izquierda y la esquina inferior derecha del *bounding box*, de modo que comparando las coordenadas donde el detector localizó el objeto con las coordenadas reales donde se ubica el objeto (*Ground truth*) se puede evaluar el funcionamiento del

detector, para comparar dichas coordenadas se emplea **la intersección sobre la unión (*Intersection over Union,* IoU)**, la IoU es una métrica que permite evaluar como de buena es una predicción. La IoU se calcula mediante la siguiente formula:

$$IoU = \frac{Área\ de\ solapamiento}{Área\ de\ la\ unión} \quad (4.1)$$

Donde el área de solapamiento es la región que tiene en común el *bounding box* predicho y el *Ground truth* y el área de la unión es la región resultante de combinar el *bounding box* predicho y el *Ground truth*, en la Figura 4.2 se muestra la ecuación de la IoU, representando las regiones asociadas a cada una de las componentes de la ecuación.

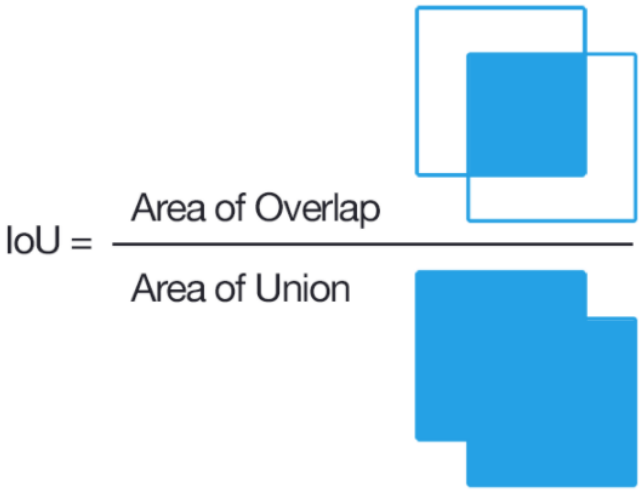


Figura 4.2: Fórmula matemática de la IoU y la representación gráfica de sus componentes (Forson, 2017).

En la práctica, es muy complicado que la detección realizada por el detector sea igual al *Ground truth*, es decir que la IoU sea 1. Como se muestra en la Figura 4.3, conseguir una IoU entorno al 0.9 significaría que el detector diseñado es excelente, mientras que si la IoU está en torno a 0.7 se trata de un detector bueno. Por lo contrario, si la IoU es inferior 0.4 el rendimiento del detector es pobre. Por lo general, se considera aceptables valores de IoU superiores a 0.5, como se hace en los desafíos de PASCAL VOC.

Figura 4.3: Ejemplo de diferentes casos de IoU (Forson, 2017)

En resumen, la IoU permite medir como de precisa ha sido la localización del objeto, pero no sirve para medir la precisión de la clasificación del objeto detectado, para evaluar la clasificación y la detección, en la mayoría de los desafíos de detección y *papers*, se emplea la **precisión media promedio (*Mean Average Precision*, mAP)**. La mAP es la media sobre las clases, del *Average Precision* (AP) interpolado para cada clase. La AP está relacionado con la precisión y la sensibilidad, introducidos en el subapartado 2.2.1. Exactamente es el área bajo la curva de precisión / sensibilidad para las detecciones. En la Figura 4.4 se muestra un ejemplo de esta curva (Henderson, 2016).

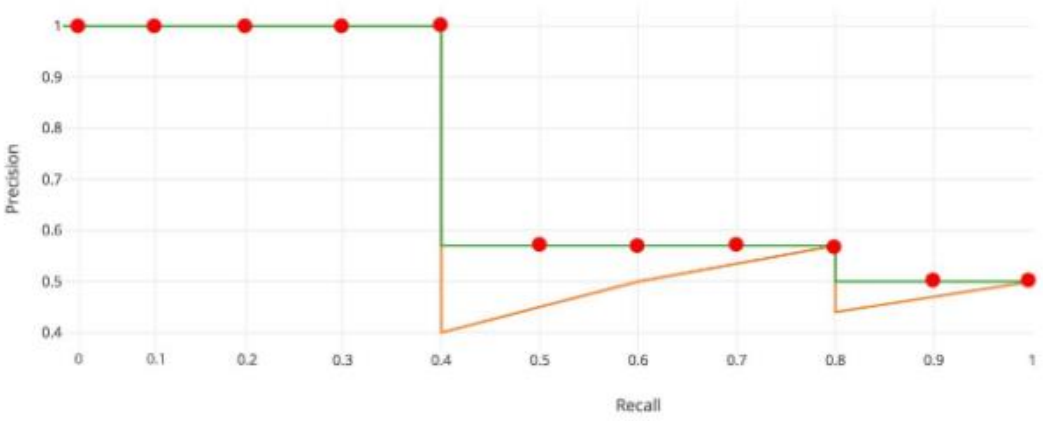


Figura 4.4: Curva precisión / sensibilidad (Hui, mAP (mean Average Precision) for Object Detection, 2018)

A lo largo de los años han aparecido diferentes formas de calcular la AP, algunas de las más conocidas son:

- En el desafío PASCAL VOC2008 se propuso calcular la AP como el promediado de la precisión máxima en 11 puntos equidistantes, de 0 a 1, en saltos de 0.1 tal y como se muestra en la Figura 4.4, representando dichos puntos en color rojo. La ecuación para calcular la AP es la siguiente (Hui, mAP (mean Average Precision) for Object Detection, 2018) :

$$AP = \frac{1}{11} \sum_{r \epsilon \{0, 0.1, \ldots, 1\}} p_{interp}(r) \qquad \textbf{(4.2)}$$

  Donde $p_{interp}(r)$ es la precisión máxima en cada uno de los once puntos de la curva.
- La propuesta de PASCAL VOC2012 es calcular la AP como el área bajo la curva de forma que no se requiera realizar una aproximación ni interpolación (Henderson, 2016).
- COCO (Evaluate: COCO, 2020) proporcionó nuevos métodos para calcular la AP, la más conocida se basa en el promediado de 101 puntos que hacen referencia a diferentes valores de IoU como criterio de decisión de los *bounding box*, de modo que la IoU promediada va de 0.5 a 0.95 en pasos de 0.05 (Hui, mAP (mean Average Precision) for Object Detection, 2018).

4.3 Detectores de objetos

En este TFM se ha optado por emplear modelos de detectores de objetos preentrenados y aplicar *transfer learning* para adaptar dichos modelos y conseguir con el detectar los defectos de los PCBs. *Tensorflow* dispone de una API de *Object detection* que dispone de un amplio número de modelos ya preentrenados y con sus respectivos archivos de configuración que simplifican el trabajar con estos modelos. El problema fue que esta API no es compatible con las nuevas versiones de *Tensoroflow* (a partir de la versión 2), por ello se decidió buscar en GitHub repositorios que proporcionen implementaciones de modelos ya preentrenados compatibles con las versiones más nuevas de *Tensorflow.* Los modelos escogidos para este trabajo fueron el SSD y el RetinaNet, los cuales se exponen a continuación:

4.3.1 Single Shot Multibox Detector (SSD)

La red *Single Shot Multibox Detector* (SSD) fue desarrollada por Wei Liu *et al.* y presentada en el artículo "*SSD: Single Shoot Multibox detector*" (2016). La SDD supuso una innovación en el campo de la detección de objetos en aquel momento, logrando récords en términos de rendimiento y precisión. Los resultados se obtuvieron con el *Dataset* de imágenes de PASCAL VOC2007, consiguiendo un mAP de 74.3% con una tasa de 59 fotogramas por segundo (FPS) para la SSD300 y un mAP de 76.9% con 22 FPS, que le confieren un alto rendimiento en tiempo real (Liu, 2016).

El nombre de *Single Shot* lo tomó porque se trata de un detector de un solo paso, es decir, esta red solo necesita recorrer una vez la red hacia delante para realizar múltiples predicciones. Esta característica le confiere un mayor rendimiento en comparación con redes de dos pasos, uno para detectar los candidatos y otro para seleccionar los candidatos. Un ejemplo de redes de dos pasos son las redes de propuestas regionales (*Region Proposal Network*, RPN) como la R-CNN. En comparación con YOLO, otra red de un solo paso, la SSD logró mejores resultados en términos de mAP, donde la YOLO logró un 63.4% para el *Dataset* PASCAL VOC2007.

4.3.1.1 Arquitectura / Modelo

La arquitectura de la SSD consta de dos partes. El comienzo de la red es una red convolucional estándar de clasificación de objetos truncada, eliminando las capas *fully-connected* denominada red base, cuyo objetivo es la extracción de características de las imágenes de entrada. La red propuesta por Wei Liu *et al.* (2016) utilizó como red neuronal base una red VGG-16 preentrenada (Liu, 2016). La otra parte de la SSD se implementa a continuación de la red base. Esta parte se trata de una red neuronal convolucional auxiliar, caracterizada por que el tamaño de sus convoluciones decrece a medida que se avanza en la red y tiene como objetivo realizar la detección de objetos en múltiples escalas. En la Figura 4.5 se representa la arquitectura de la SSD con la VGG-16 como red base (Liu, 2016).

La red convolucional auxiliar se caracteriza por (Liu, 2016):

- **Mapas de características multiescala**: A medida que se avanza en la red el tamaño de las capas convolucionales disminuye permitiendo la detección multiescala. A medida que el tamaño de las convoluciones también disminuye la resolución de los mapas de características, de modo que las capas más profundas, las de menor resolución, se emplean para detectar objetos de gran tamaño, mientras que las capas iniciales ayudan a detectar objetos de menor tamaño (Hui, 2018).
- **Predictores convolucionales**: las capas de extracción de características añadidas o de alguna de las capas de la red base pueden extraer un conjunto fijo de predicciones, las predicciones consisten en las puntuaciones para las clases y las coordenadas de la caja que lo contiene. En la arquitectura de la SDD mostrada en la Figura 4.5, las capas convolucionales de 3x3x512 se aplican 4 filtros que asignan 512 canales de entrada a 25 de salida, donde 21 son las puntuaciones de las clases y 4 las coordenadas de la caja que encierra al objeto.
- **Cajas por defecto y relación de aspectos**: Cada mapa de características tiene asociado un conjunto de cuadros delimitadores o *bounding boxes*. La posición de cada caja con respecto a su mapa de característica es fija. Para cada caja se computa la puntuación de las clases y las cuatro coordenadas asociadas al desplazamiento de la caja predeterminada. Estas cajas por defecto son similares al *anchor boxes* de las R-CNN, pero que se aplican a mapas de características de diferentes escalas. Además, la relación de aspectos permite utilizar cajas de diferentes dimensiones lo que permite discretizar más eficientemente el espacio.

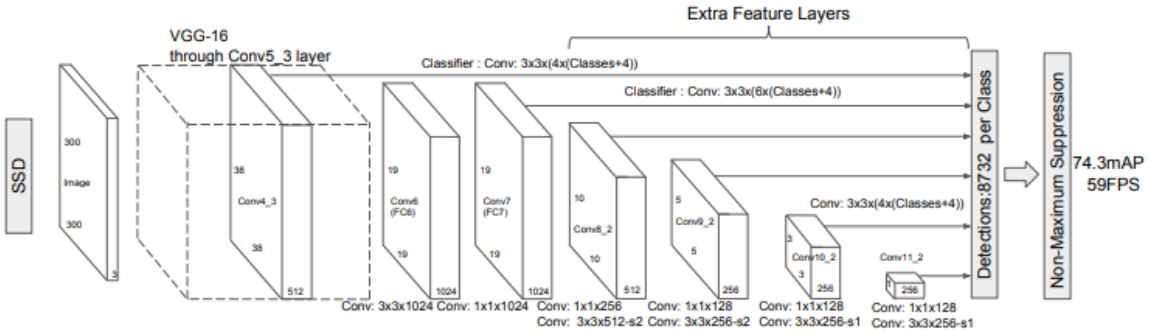


Figura 4.5: Arquitectura del modelo SDD basado en una VGG-16 (Liu, 2016).

4.3.1.2 Entrenamiento

La clave del entrenamiento de la SSD es que la información de los *ground truth* tiene que ser asignada a una de las salidas del detector. Tras realizar esta asignación, se aplica la función de pérdida y *back propagation*. Otra de las partes clave del entrenamiento de la SSD es la selección de los cuadros por defecto y las escalas de detección (Liu, 2016).

- **Estrategia de emparejamiento**: Durante el entrenamiento se busca determinar que cuadros se corresponden con los *ground truth* y entrenar la red en función a estos. La SSD clasifica las predicciones como coincidencias positivas o negativas, de las cuales únicamente emplea las positivas para calcular el coste de localización. Si el *bounding box* por defecto tiene una IoU superior a un umbral

de 0.5 con respecto al *ground truth*, ese cuadro se etiqueta como positivo, si por el contrario es inferior al umbral este es negativo.
Como se comentó anteriormente, la SSD combina mapas de características de diferentes escalas para detectar objetos de diferentes tamaños. De modo que aquellos mapas de características de mayor resolución se emplean para detectar los objetos pequeños, para evitar problemas de detección con objetos pequeños es recomendable emplear imágenes de mayor resolución.

- **Función de pérdida**: Esta puede definirse como la suma de dos pérdidas específicas, la pérdida de localización y la pérdida de confianza.
  La pérdida de localización mide el desajuste entre el *bounding box* predicho y el *ground truth*. Únicamente se consideran las condiciones positivas. La ecuación de la pérdida de localización es la siguiente:

$$L_{loc}(x,l,g) = \sum_{i\in Pos}^{N}\sum_{m\in\{cx,cy,w,h\}} x_{ij}^{k}\,\mathrm{smooth}_{\mathrm{L1}}\,(l_i^m - \hat{g}_j^m) \qquad \textbf{(4.3)}$$

$$\hat{g}_j^{cx} = (g_j^{cx} - d_i^{cx})/d_i^w \qquad \hat{g}_j^{cy} = (g_j^{cy} - d_i^{cy})/d_i^h$$

$$\hat{g}_j^{w} = \log\left(\frac{g_j^w}{d_i^w}\right) \qquad \hat{g}_j^{h} = \log\left(\frac{g_j^h}{d_i^h}\right)$$

  Donde l es el cuadro predicho y g es el *ground truth*. La $\mathrm{smooth}_{\mathrm{L1}}$es la pérdida suave L1, la cual es una función de perdida basada en la combinación de la pérdida L1 y L2, con centros en (c_x, c_y) del *bounding box* (d) y con un ancho (w) y altura (h) (Liu, 2016).
  La pérdida de confianza es la pérdida asociada al hacer una predicción de clase. Para cada coincidencia positiva, la perdida se obtiene basándose en su puntuación. Mientras que para las coincidencias negativas se calcula basándose en la puntuación de la clase 0 o fondo, la cual se asocia a la no presencia de ningún objeto. La ecuación matemática de la pérdida de confianza se define como:

$$L_{\mathrm{conf}}(x,c) = -\sum_{i\in\mathrm{Pos}}^{N} x_{ij}^{p}\log\,(\hat{c}_i^p) - \sum_{i\in Neg}\log\,(\hat{c}_i^0) \quad Donde\ \hat{c}_i^p = \frac{\exp(c_i^p)}{\sum_p \exp\,(c_i^p)} \qquad \textbf{(4.4)}$$

  De modo que la ecuación de la función de perdida, en función de estas dos pérdidas se define como:

$$L(x,c,l,g) = \frac{1}{N}(L_{conf}(x,c) + \alpha L_{loc}(x,l,g)) \qquad \textbf{(4.5)}$$

- **Selección de escalas y relación de aspectos de las cajas por defecto:** Se trata de uno de los puntos claves de la SSD, para poder detectar objetos de diferentes escalas se emplean mapas de características inferiores y superiores. Cuanto más número de mapas de características se implementen mayor será la carga computacional de la red. Los mapas de características tienen diferentes tamaños en función del nivel de la red. Las escalas se diseñan de manera que se adapten en función del número de mapas de características, de modo que, si

se empelan *m* mapas de características, las escalas predeterminadas por cada capa se calculan como:

$$s_k = s_{min} + \frac{s_{max} - s_{min}}{m - 1}(k - 1), k \in [1, m] \quad \textbf{(4.6)}$$

Donde s_{min} es la escala mínima y es 0.2, mientras que s_{max} es la escala máxima, que toma el valor de 0.9.
Una vez se calculan las escalas, se combinan con la relación de aspectos para calcular la anchura y la longitud de los cuadrados predeterminados. Las ecuaciones para calcular la anchura y la longitud son las siguientes:

$$w = scale \cdot \sqrt{aspect\ ratio} \quad \textbf{(4.7)}$$

$$h = \frac{scale}{\sqrt{aspect\ ratio}} \quad \textbf{(4.8)}$$

Para la relación de aspecto 1, la escala se define como:

$$scale = \sqrt{scale \cdot scale_{siguiente\ nivel}} \quad \textbf{(4.9)}$$

De esta manera se consiguen un amplio número de predicciones, que cubren diversos tamaños y formas de objetos de entrada.

- ***Hard negative mining***: Uno de los problemas a los que se enfrenta la SSD es el desequilibrio entre coincidencias positivas y negativas. Para atenuar dicho desequilibrio se clasifican empleando una mayor pérdida de confianza para cada cuadro por defecto y se seleccionan los mejores de manera que la relación entre los negativos y los positivos sea a lo sumo 3:1. Esto da lugar a un entrenamiento más rápido y estable.

Para mejorar el entrenamiento y generar un modelo más robusto, al igual que se hace en otros algoritmos de *Deep Learning* se puede emplear la técnica de *Data Augmentation* con el fin de aumentar el número y variedad de imágenes de entrenamiento.

Para la implementación de la SSD se empleó el repositorio de Github de Pierluigi Ferrari llamado ssd_keras, basado en el artículo de Wei Liu (2016) titulado “SSD: *Single Shot Multibox Detector*”. En este repositorio se presentan tres tipos de SSD para implementar, las cuales son:

- **SSD 300**: La arquitectura de esta SSD es igual a la definida en el artículo mencionado antes, es decir consta de una VGG 16 como red base y las capas de predictores consta de 7 capas, lo que permite un total de 7 escalas posibles. Además, las imágenes de entrada deben tener un tamaño de 300x300 píxeles.

- **SSD 512**: Esta red es similar a la anterior, pero trabaja con imágenes de mayor resolución, 512x512 píxeles. Al procesar imágenes de mayor resolución, la capa de predicción consta de una capa más que la anterior, lo que implica que tiene una escala más además de un paso adicional.
- **SSD 7**: Esta es una versión modificada de la SSD con una red base personalizada y 5 capas para ocuparse de las predicciones, en total consta de 7 capas. El hecho de que conste de un menor número de capas y parámetros de que las anteriores permite que su ejecución sea más rápida y consuma menos recursos.

4.3.2 RetinaNet

RetinaNet es un detector de objetos creado por el equipo de *Facebook AI Research* (FAIR) en 2018, el cual se diseñó para suplir los problemas de baja exactitud que presentaban hasta el momento los detectores de una etapa con respecto a los de dos. Dicho problema tiene su origen en el desequilibrio de la clase de primer plano y fondo que tienen los detectores de una etapa. En el caso de los detectores de dos etapas, existe un equilibrio de clase entre fondo debido a sus dos etapas de funcionamiento, dado que en la primera etapa se reduce el número de candidatos, filtrando la mayoría de las muestras de fondo, y en la segunda etapa se realiza la clasificación de los candidatos. En el caso de los detectores de una etapa, existe un desbalanceo extremo entre las clases y el fondo. Esto se debe a que el detector evalúa un número muy elevado de candidatos a objeto, de modo que, aunque el detector sea muy bueno clasificando los candidatos de fondo, supone un problema de identificación. Para solucionar el desequilibrio se desarrolló una nueva función de pérdida denominada pérdida focal (Lin, 2017).

4.3.2.1 Arquitectura

RetinaNet es una red única y unificada compuesta por una red troncal y dos subredes para tareas específicas. La primera de las dos subredes específicas es una red convolucional encargada de la clasificación de las salidas de la red troncal. La segunda subred calcula la regresión convolucional de los *bounding box*. Las dos subredes tienen un diseño simple diseñado específicamente para la detección en un paso La arquitectura de RetinaNet se muestra en la Figura 4.6. En ella se pueden apreciar los cuatro componentes que las forman, los cuales son (Lin, 2017):

- Red troncal o *backbone*: Es una red convolucional estándar cuya función es la extracción de los mapas de características de las imágenes de entrada. En RetinaNet se emplea como red troncal una ResNet (50 o 101), introducida en el apartado 2.1.4.3.

- *Feature Pyramid Network* (FPN): Esta técnica permite a RetinaNet realizar predicciones multiescala de las imágenes de entrada. La FPN consiste en el aumento de una red convolucional estándar de arriba hacia abajo y con conexiones laterales, de modo que se construya una pirámide de extracción de características multi escalas para una imagen de entrada con escala fija. Cada nivel de la pirámide detecta objetos de una escala diferente.

Cada nivel tiene asociado un total de 9 escalas, en función de la relación de aspectos fijadas para cada capa se pueden cubrir diferentes rangos de escalas. De esta manera se generan las *anchor boxes* candidatas, cada una de ellas tiene asociado un vector de longitud igual al número de clases que se quieren detectar y otro de longitud cuatro con las coordenadas de las aristas del *anchor box*.

- Subred de clasificación: Tiene la función de predecir la probabilidad de presencia de objetos en cada una de las posiciones espaciales de las *anchor boxes* para cada clase de objetos. Esta red es una capa *fully-connected* que toma como entrada cada uno de los mapas de características de entrada con *C* canales procedentes de la FPN. La subred consta de cuatro capas convolucionales con un *kernel* de 3x3 y función de activación ReLu. Por último, consta de una capa de activación sigmoidea.

- Subred de regresión de cajas: Implementada en paralelo a la subred de clasificación. Se trata de otra capa *fully-connected* conectada a cada uno de los niveles de la FPN, al igual que la de clasificación. El objetivo de esta capa es calcular el desplazamiento relativo entre el *anchor box* y las cajas verdaderas. La subred de clasificación y la de regresión de cajas, a pesar de compartir estructura, no utilizan parámetros en común.

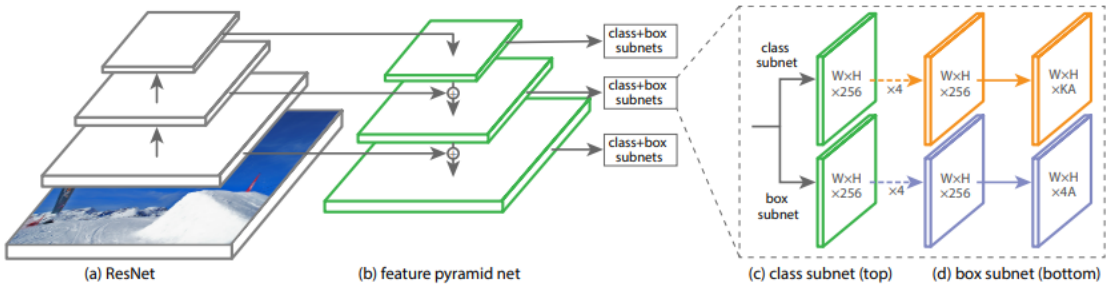


Figura 4.6: Arquitectura de RetinaNet (Lin, 2017)

4.3.2.2 Pérdida Focal

La pérdida focal (*Focal loss*) es una de las características más diferenciantes de RetinaNet. Al igual que ocurría con la SSD y otros detectores de objetos de un solo paso, las imágenes presentan un desbalanceo de clases debido a que la mayor parte de elementos pertenecen a la clase *background*, la cual serían el resto de los elementos que no pertenecen a ninguna de las clases de objetos que se quiere detectar. Este desbalanceo dificultad la tarea de detección de los otros elementos que no sean *background*, para solucionar este modelo es para lo que se emplea la pérdida focal (Lin, 2017).

La pérdida focal es una reconfiguración de la pérdida de entropía cruzada. Esta pérdida permite realizar un ajuste de los parámetros en el cual las clases más fáciles de clasificar tengan menor impacto que las más difíciles (Lin, 2017).

La función de pérdida focal es la siguiente (Lin, 2017):

$$FL(p_t) = -(1-p_t)^{\gamma} * \alpha * \log(p_t) \quad \textbf{(4.10)}$$

donde:

- p_t es la probabilidad estimada de la clase etiquetada y se define como:

$$p_t = \begin{cases} p & si\ y = 1 \\ 1-p & otros\ casos \end{cases} \quad \textbf{(4.11)}$$

  siendo la $y \in \{\pm 1\}$ la clase real del objeto localizado y $p \in [0,1]$ es la probabilidad estimada del modelo para la clase y.
- γ y α son variables reguladoras.

4.3.2.3 Inferencia / entrenamiento

Durante la fase de inferencia de RetinaNet solo se decodifican los cuadros predichos de como mucho 1000 predicciones de alta puntuación por nivel de la FPN, después de limitar el umbral de confianza del detector a 0.05. Las mejores prediciones de todos los niveles se combinan y se aplica una supresión no máxima con un umbral de 0.5 para obtener las predicciones finales, de esta forma se optimiza la velocidad del detector (Lin, 2017).

Durante el entrenamiento de la red RetinaNet se aplica la pérdida focal a todos los *anchors* (en torno a 100000) de cada imagen muestreada, de modo que la perdida focal de la red se calcula como la suma de las perdidas focales asociadas a cada una de las pérdidas de los *anchors* normalizada por el número de *anchors* asociados a un *ground truth*. La selección de la α y γ están relacionadas, de modo que si se disminuye la α habrá que aumentar la γ, asegurando que el peso de la clase más difícil de clasificar tenga un rango estable, los valores que generalmente mejor funcionan son $\gamma = 2$ y $\alpha = 0.25$ (Lin, 2017).

4.4 Aplicación de SSD y RetinaNet para la detección de defectos en PCB

Como se mencionó en el apartado 3.1, la base de datos que se ha utilizado en este trabajo es la que se emplea en el artículo de Ding, R. *et al.* (2019). En dicho artículo se expone un detector de defectos denominado red TDD (*Tiny defect detection*). El autor proporciona una descripción de la arquitectura de la red, pero no proporciona su código.

Para la implementación de la SSD se partió del código proporcionado en el repositorio ssd_keras del github de pierluigiferrari (github/pierluigiferrari/ssd_keras). Mientras que para la RetinaNet se empleó el repositorio keras-RetinaNet del github de fyzr (github/fizyr/keras-retinanet).

En este TFM se han llevado a cabo dos métodos de obtención de resultados debido a que la carga computacional y los tiempos de ejecución de los dos detectores es alta.

En un primer análisis, se dividió el *dataset* en tres conjuntos diferentes para entrenamiento, validación (para usar la técnica de *early stopping*) y test. En este primer escenario los resultados obtenidos aportan una información sesgada del comportamiento de los detectores para estos conjuntos de imágenes. Lógicamente, el hecho de probar distintas configuraciones sobre un mismo conjunto de test introduce sesgo de selección, por lo que disminuye la fiabilidad de los resultados de las prestaciones.

Por último, se realizó un segundo análisis empleando validación cruzada de 5 iteraciones con el objetivo de obtener unos resultados más precisos acerca del funcionamiento del detector. En este caso se realizó únicamente para el detector basado en RetinaNet 50 dado que los tiempos de ejecución eran muy altos (más de 24 horas por iteración)

4.4.1 SSD

Las diferentes configuraciones de la red SSD evaluadas en este TFM son las siguientes.

4.4.1.1 Configuración 1

La primera configuración propuesta consistió en emplear el modelo de la SSD 300 con la misma configuración que tiene por defecto para el desafío Pascal VOC, lo que permite aplicar *transfer learning* de este modelo entrenado previamente con el set de imágenes de Pascal VOC. La configuración de las escalas y relación de aspectos son las siguientes:

- Escalas = [0.1, 0.2, 0.37, 0.54, 0.71, 0.88, 1.05]
- Relación de aspectos (por capas) = [[1.0, 2.0, 0.5],
  [1.0, 2.0, 0.5, 3.0, 1.0/3.0],
  [1.0, 2.0, 0.5, 3.0, 1.0/3.0],
  [1.0, 2.0, 0.5, 3.0, 1.0/3.0],
  [1.0, 2.0, 0.5],
  [1.0, 2.0, 0.5]]

Los resultados de la pérdida de validación obtenidos durante el entrenamiento del modelo se muestran en la Figura 4.7. Observando la figura mencionada anteriormente, se puede apreciar que la pérdida de validación no llega a converger, lo cual es un indicativo de un mal funcionamiento de este modelo.

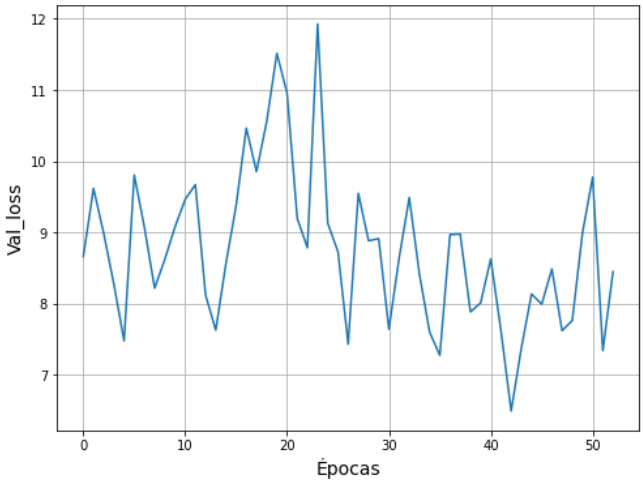


Figura 4.7: Representación de la pérdida de validación durante el entrenamiento de la configuración 1.

Una vez entrenado esta configuración de la SSD, se realizó el testeo con el conjunto de test. Los resultados obtenidos fueron negativos, siendo el modelo incapaz de detectar ninguno de los defectos etiquetados en las imágenes.

4.4.1.2 Configuración 2

A raíz de los resultados obtenidos en la configuración anterior, se modificaron los valores de la escala y la relación de aspecto con la finalidad de detectar mejor los objetos de pequeño tamaño. Para seleccionar unos valores apropiados para los dos parámetros mencionados, se inspeccionó los tamaños típicos de los *ground truth* de los objetos etiquetados y mediante las ecuaciones del apartado 4.3.1.2 se calcularon. Los valores obtenidos de las escalas y relaciones de aspectos fueron los siguiente:

- Escalas = [0.01667, 0.02357, 0.03333, 0.05, 0.083333, 0.1, 0.25]
- Relación de aspectos (para todas las capas) = [0.5, 1, 2, 3, 1/3, 4, 1/4]

En la Figura 4.8 se representa la evolución de la pérdida de validación. Al igual que ocurrió en la anterior configuración, el valor de la pérdida cuando el modelo converge es elevado.

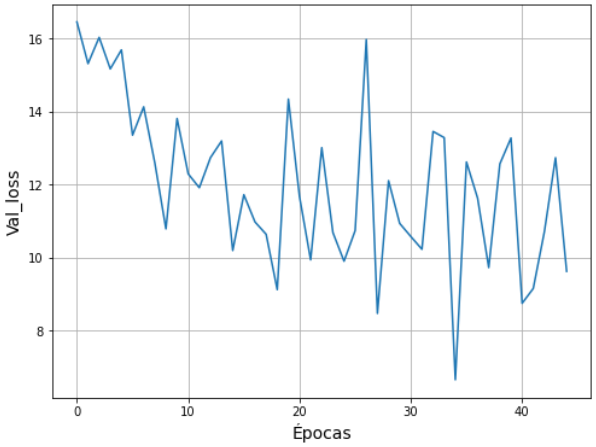


Figura 4.8: Representación de la pérdida de validación durante el entrenamiento de la configuración 2.

Tras el entrenamiento de esta configuración, se evaluó el modelo con el conjunto de test. Esta configuración fue incapaz de detectar ningún defecto en las imágenes.

4.4.1.3 Configuración 3

Como las dos configuraciones anteriores obtuvieron muy malos resultados, se decidió cambiar el modelo de SSD implementada, pasando de la SSD 300 a la SSD 512, la cual tiene como entrada imágenes de mayor resolución que la anterior lo cual mejora en cierta medida la detección de objetos pequeños a costa de una mayor carga computacional. Además, se los valores de las escalas y relaciones de aspectos se modificaron, los valores empleados para estos parámetros fueron:

- Escalas = [0.01667, 0.02357, 0.03333, 0.05, 0.083333, 0.1, 0.25, 0.5]
- Relación de aspectos (para todas las capas) = [0.5, 1, 2, 4, 1/4]

El valor de la pérdida de validación durante el entrenamiento del modelo es peor que en las dos configuraciones, lo cual indica que su comportamiento va a ser peor que las dos configuraciones previas, los valores y la evolución de la pérdida de validación durante el entrenamiento se muestran en la Figura 4.9.

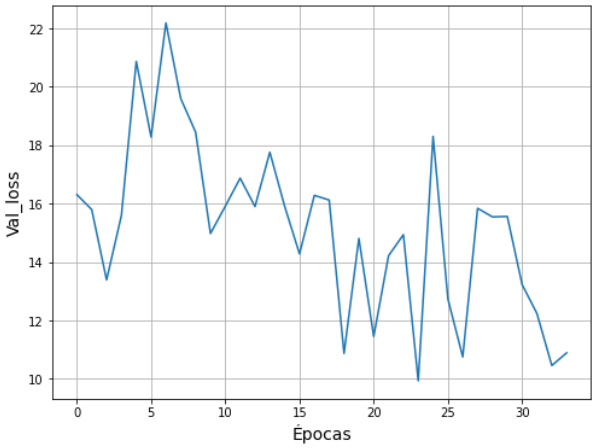


Figura 4.9: Representación de la pérdida de validación durante el entrenamiento de la configuración 3

Al igual que ocurrió en los dos casos anteriores, cuando se realizó el testeo de la configuración esta fue incapaz de detectar ninguna de los defectos.

4.4.1.4 Configuración 4

Otro de los modelos de SSD implementados en el repositorio empleado es la denominada SSD 7, la cual consta de 7 capas lo cual permite un entrenamiento más rápido y con menor carga computacional para el ordenador. El entrenamiento de este modelo se realiza desde cero, debido a que no hay ningún modelo preentrenado de este con alguna base de datos.

- Escalas = [0.0333, 0.0986, 0.1323, 0.25, 0.50]
- Relación de aspectos (para todas las capas) = [1, 1.333,0.666666, 1/4, 4, 1.15, 0.85]

La evolución de la pérdida de validación de esta configuración durante la fase de entrenamiento se representa en la Figura 4.10, en la cual se puede apreciar que la evolución de esta métrica es irregular, pero toma mejores valores que las anteriores configuraciones.

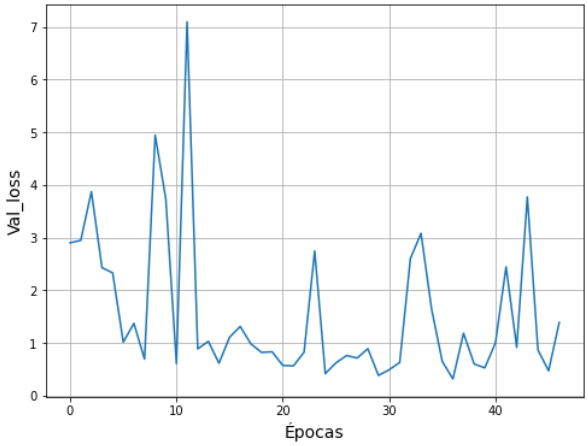


Figura 4.10: Representación de la pérdida de validación durante el entrenamiento de la configuración 4

Los resultados del testeo se recogen en la Tabla 4.2, analizándolos se puede apreciar que sigue siendo incapaz de detectar correctamente los defectos.

	Valor
AP Missing hole	0.0
AP Mouse bite	0.002
AP Open circuit	0.003
AP Short	0.003
AP Spur	0.005
AP Spurious copper	0.045
mAP	0.01

Tabla 4.2: Valores de la AP de cada clase y la mAP de la configuración 4

4.4.1.5 Configuración 5

La configuración 5 consistió en implementar una SSD similar a la de la configuración anterior pero aumentado la resolución de las imágenes de entrada de 300x300 a 600x600 píxeles, con el objetivo de facilitar la detección de los defectos. La configuración empleada para las escalas y relaciones de aspectos es la misma que la de la configuración anterior. La evolución de la pérdida de validación durante el entrenamiento se representa en la Figura 4.11, analizando la gráfica se puede apreciar que la evolución de esta métrica es muy irregular presentando números picos de una amplitud considerable.

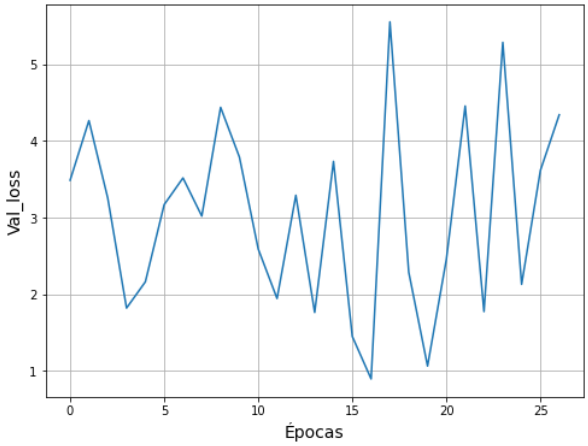


Figura 4.11: Representación de la pérdida de validación durante el entrenamiento de la configuración 5

Seguidamente, se realizó la evaluación del modelo obteniendo los resultados que se recogen en la Tabla 4.3. Los resultados de las APs de cada defecto y la mAP del modelo siguen siendo negativos, lo cual indica la incapacidad de esta configuración para detectar los defectos.

	Valor
AP Missing hole	0.0
AP Mouse bite	0.009
AP Open circuit	0.098
AP Short	0.021
AP Spur	0.018
AP Spurious copper	0.005
mAP	0.025

Tabla 4.3: Valores de la AP de cada clase y la mAP de la configuración 5

4.4.1.6 Configuración 6

Por último, se propuso una configuración que implementaba como base el modelo de la SSD 7, pero se añadieron capas a su red base, con el objetivo de aumentar las características extraídas y probar si era posible modificar la red base sin afectar a su funcionalidad. Exactamente, se añadió una capa convolucional a continuación de la séptima capa de convolución y una capa de *pooling* entre ambas. Esta octava convolución consta de 64 filtros de tamaño 3x3.

En la Figura 4.12 se representa la evolución de la pérdida de validación durante el entrenamiento del modelo.

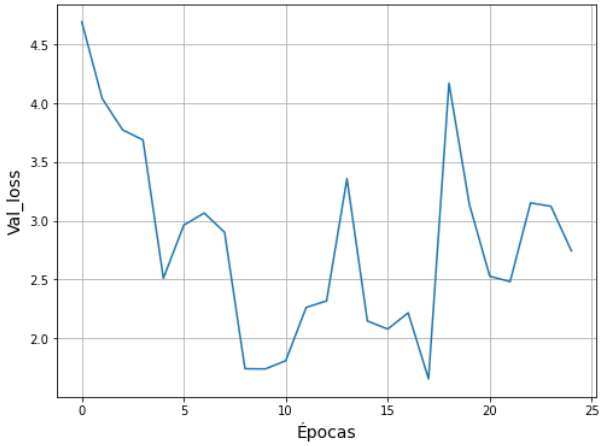


Figura 4.12: Representación de la pérdida de validación durante el entrenamiento de la configuración 6

Las APs de cada clase y la mAP que consiguió este modelo cuando se evaluó el modelo se muestran en la Tabla 4.4. Los resultados logrados por esta configuración son superiores a las anteriores, pero, aun así, esta configuración no consigue alcanzar unos valores lo suficientemente buenos. Esta configuración es incapaz de detectar los *missing hole* y todas las métricas toman valores bajos.

	Valor
AP Missing hole	0.0
AP Mouse bite	0.303
AP Open circuit	0.22
AP Short	0.122
AP Spur	0.359
AP Spurious copper	0.402
mAP	0.285

Tabla 4.4: Valores de la AP de cada clase y la mAP de la configuración 6

4.4.2 RetinaNet

En el caso de la red RetinaNet se han comparado dos modelos diferentes, uno con la red ResNet 50 como *backbone* y otro con la red ResNet 101.

4.4.2.1 RetinaNet basada en ResNet 50

En primer lugar, se realizó el entrenamiento de la red con las capas del *backbone* congeladas. La evolución del la pérdida de validación y la mAP de la red se representa en la Figura 4.13.

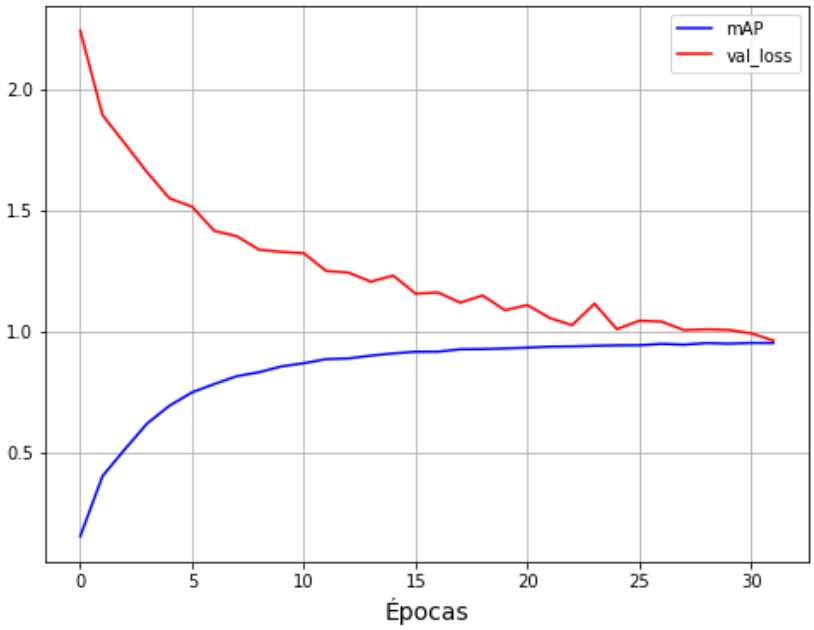


Figura 4.13: Gráficas de la mAP (azul) y pérdida de validación (rojo) durante el entrenamiento de Retinanet basada en ResNet 50 con las capas del *backbone* congeladas.

Además, se decidió evaluar la red congelada con el conjunto de imágenes de test, obteniendo los resultados que se muestran en la Tabla 4.5. Esta red logra mucho mejores resultados que cualquiera de las configuraciones de la red SSD a pesar de tener las capas del *backbone* congeladas.

	Valor
AP Missing hole	0.9861
AP Mouse bite	0.9113
AP Open circuit	0.7689
AP Short	0.9504
AP Spur	0.8998
AP Spurious copper	0.9309
mAP	0.9079

Tabla 4.5: Valores de las APs de cada defecto y la mAP del modelo obtenidos en el test de Retinanet basada en ResNet 50 con las capas del *backbone* congeladas.

Como se puede observar analizando la tabla anterior, el detector es capaz de detectar correctamente la mayoría de los defectos de las imágenes, presentando peores resultados en la detección de los circuitos abiertos, aun así, la mAP del modelo es del 90%, lo cual indica que este detector tiene un gran rendimiento para la detección de defectos en PCBs.

Por último, se descongelaron las capas del *backbone* y se continuó con el entrenamiento, en la Figura 4.10 se representa la evolución de la mAP y la pérdida de validación del modelo. Los datos mejoran con respecto a los representados en la Figura 4.9.

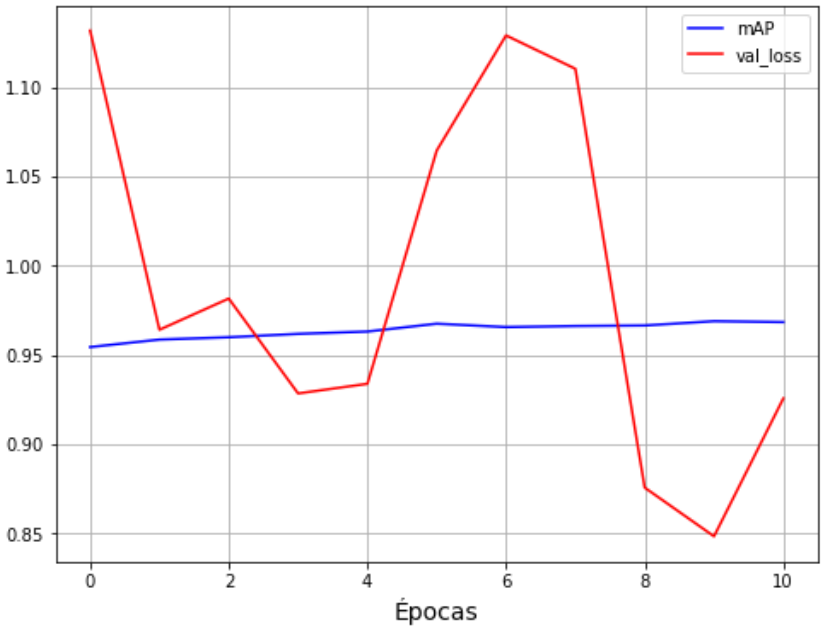


Figura 4.14: Gráficas de la mAP (azul) y pérdida de validación (rojo) durante el entrenamiento de Retinanet basada en ResNet 50 con todas las capas descongeladas.

Una vez se entrenó el modelo, se realizó su evaluación con las imágenes del conjunto de test, los resultados obtenidos se recogieron en la Tabla 4.6. Observando dichos resultados se aprecia cómo se mejoraron todas las APs de cada defecto provocando a su vez que también lo haga la mAP del modelo.

	Valor
AP Missing hole	0.9874
AP Mouse bite	0.9304
AP Open circuit	0.8176
AP Short	0.9629
AP Spur	0.9296
AP Spurious copper	0.9402
mAP	0.9280

Tabla 4.6: Valores de las APs de cada defecto y la mAP del modelo obtenidos en el test de ResNet 50 con todas las capas descongeladas.

4.4.2.2 RetinaNet basada en Resnet 101

De igual modo que se procedió en el caso anterior, primeramente, se congelaron las capas del *backbone* de la RetinaNet, que en este modelo se trata de una ResNet 101. En la Figura 4.15 se muestra la evolución de la perdida de validación y la mAP del modelo durante la fase de entrenamiento hasta que el modelo converge. Comparando la gráfica de este modelo con el anterior se puede apreciar que ambos son muy similares, aunque el basado en la ResNet 101 presenta una gráfica para la validación cruzada con un mayor número de variaciones.

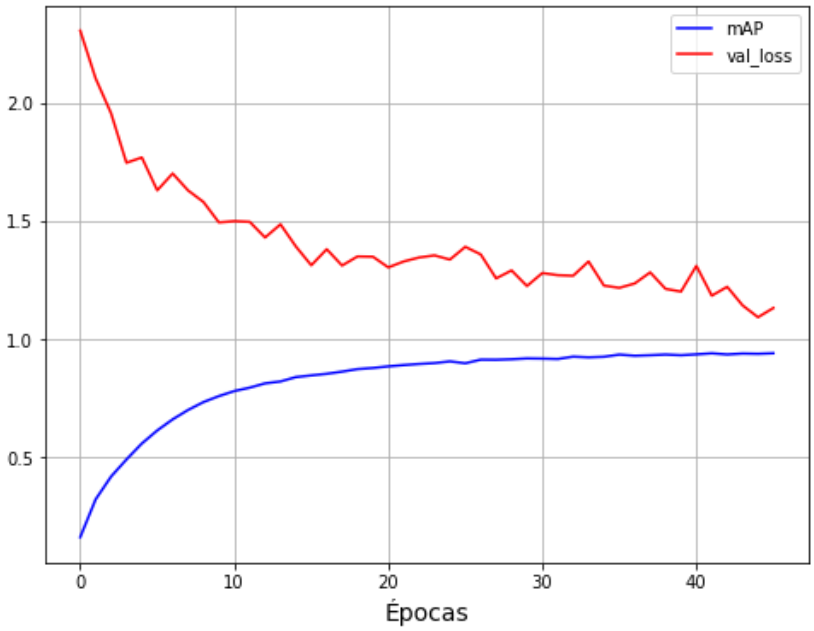


Figura 4.15: Gráficas de la mAP (azul) y pérdida de validación (rojo) durante el entrenamiento de Retinanet basada en ResNet 101 con las capas del *backbone* congeladas.

Una vez concluido el entrenamiento, se realizó el test de la red con las capas congeladas obteniéndose los resultados que se muestran en la Tabla 4.7. Los resultados obtenidos en este caso son inferiores que los que se obtenían con la ResNet 50 con las capas congeladas (Tabla 4.5).

	Valor
AP Missing hole	0.9856
AP Mouse bite	0.8761
AP Open circuit	0.6471
AP Short	0.9319
AP Spur	0.8569
AP Spurious copper	0.9041
mAP	0.8670

Tabla 4.7: Valores de las APs de cada defecto y la mAP del modelo obtenidos en el test de RetinaNet basada en la red ResNet 101 con las capas del *backbone* congeladas.

Una vez el modelo con las capas congeladas logró su convergencia, se procedió a la descongelación de las capas del *backbone* y se volvió a entrenar. En la Figura 4.16 se muestra la evolución de las métricas de la pérdida de validación y mAP del modelo durante el entrenamiento.

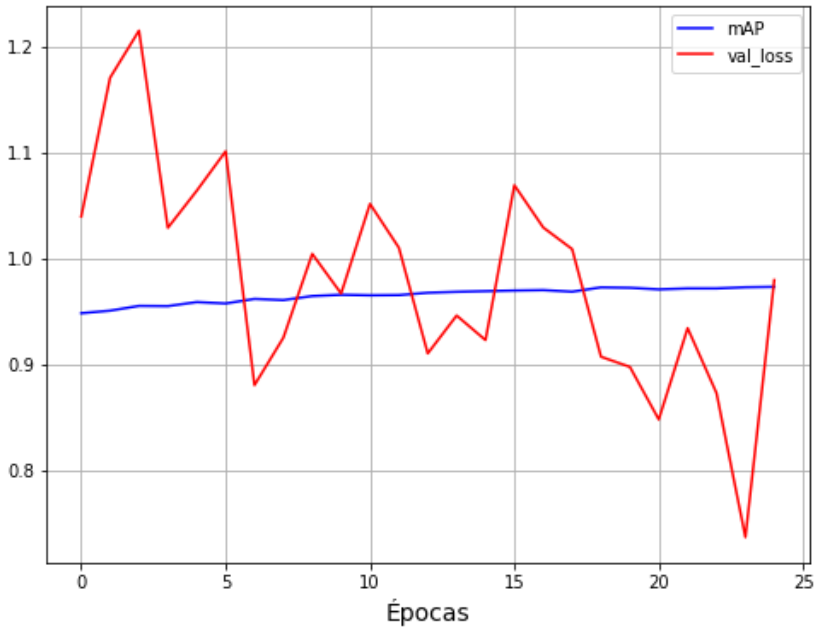


Figura 4.16: Gráficas de la mAP (azul) y pérdida de validación (rojo) durante el entrenamiento de Retinanet basada en ResNet 101 con todas las capas descongeladas.

Por último, se realizó el testeo del modelo y se obtuvieron las medidas de las APs asociadas a la detección de cada defecto y la mAP final del modelo, estas métricas se recogen en la Tabla 4.8.

	Valor
AP Missing hole	0.9914
AP Mouse bite	0.9427
AP Open circuit	0.8412
AP Short	0.9778
AP Spur	0.9527
AP Spurious copper	0.9363
mAP	0.9404

Tabla 4.8: Valores de las APs de cada defecto y la mAP del modelo obtenidos en el test de RetinaNet basada en la red ResNet 101 descongelando todas las capas.

Comparando los resultados finales de los dos modelos RetinaNet se puede observar que los obtenidos por la RetinaNet basada en la red ResNet 101 son ligeramente superiores, aunque si nos fijamos en los tiempos de entrenamiento la red basada en ResNet 50 sería mejor que la basada en la 101.

4.4.3 Comparativa entre los detectores

A la vista de los resultados anteriores, se deduce que el detector que mejores resultados logra en la detección de defectos en PCBs es el RetinaNet, tanto el basado en la CNN ResNet 50 como en la ResNet 101, aunque este último presenta unos tiempos de

ejecución que duplican al primero logrando una ligera mejora en los resultados finales. En lo que respecta a la SSD, a pesar de intentar solventar la limitación que presenta para la detección de objetos de pequeño tamaño mediante la modificación de los parámetros de escala y relación de aspectos de las cajas de predicción, no logra buenos resultados, siendo estos muy inferiores a los obtenidos por la RetinaNet.

4.4.4 Validación cruzada de RetinaNet basada en ResNet 50

Teniendo en cuenta los resultados de la comparativa, nos centramos en la arquitectura ResNet 50 y haremos una evaluación más detallada de sus prestaciones, empleando para ello validación cruzada.

Las APs de cada defecto etiquetado y la mAP del detector obtenidas en cada una de las iteraciones de la validación cruzada se recogen en la Tabla 4.9, además se representan los *boxes plot*s de las APs de cada defecto y la mAP del detector en la Figura 4.17 y Figura 4.18 respectivamente.

	K=1	K=2	K=3	K=4	K=5
AP Missing hole	0.9829	0.9887	0.9777	0.9884	0.9858
AP Mouse bite	0.9250	0.9627	0.9459	0.9564	0.9471
AP Open circuit	0.8699	0.8380	0.8642	0.8546	0.8166
AP Short	0.9578	0.9527	0.9614	0.9756	0.9506
AP Spur	0.9387	0.9675	0.9338	0.9489	0.9379
AP Spurious copper	0.9314	0.9346	0.9477	0.9556	0.9347
mAP	0.9343	0.9407	0.9385	0.9466	0.9288

Tabla 4.9: Valores de las APs de cada defecto y la mAP del detector RetinaNet basado en la CNN ResNet 50 en cada una de las 5 iteraciones de la validación cruzada

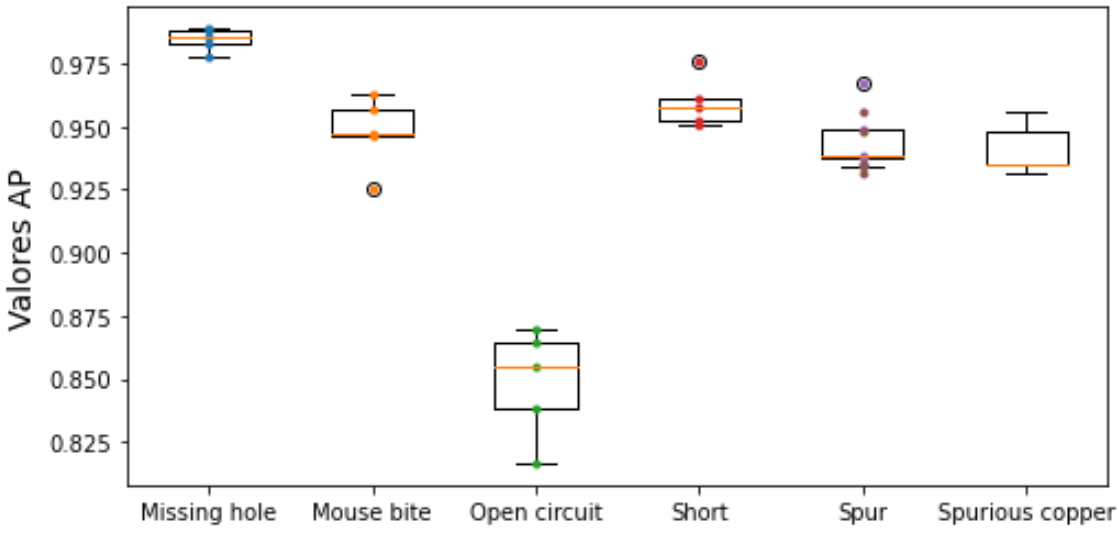


Figura 4.17: *Boxes plots* de las AP de cada defecto de los PCBs obtenidos para el detector RetinaNet basado en la CNN ResNet 50

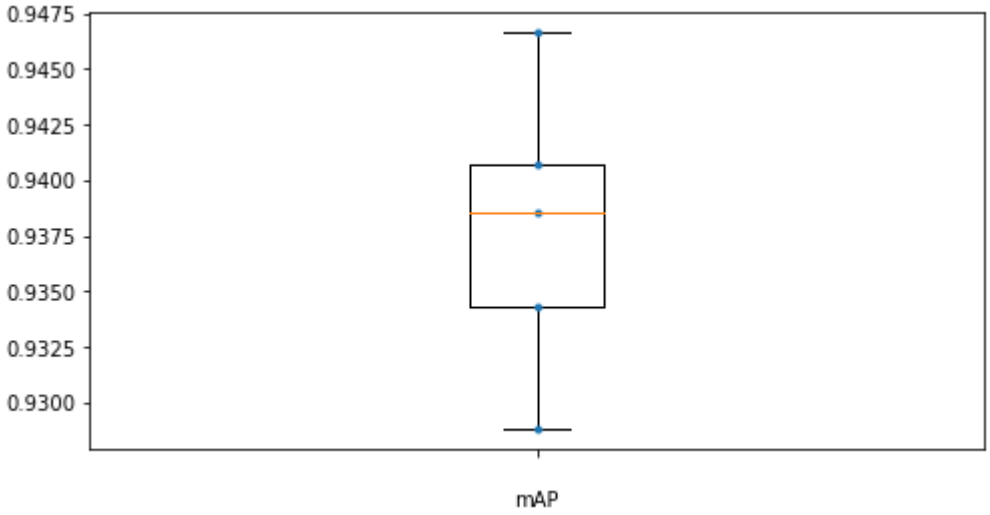


Figura 4.18: *Box plot* de la mAP del detector RetinaNet basado en la CNN ResNet 50

Las mAPs obtenidas en cada iteración son muy buenas, alcanzando como peor valor un 92.88% y como mejor valor un 94.66%, lo que supone un valor medio de 93.78%, dicho valor medio es muy similar al obtenido cuando se realizó el estudio segado del mismo detector. En lo que respecta a las APs de cada defecto, los resultados son similares a los obtenidos en el análisis sesgado del modelo, consigue unos buenos valores para todos los defectos, en la todos se alcanzan APs superiores al 93% exceptuando los circuitos abiertos, los cuales presenta cierta limitación a la hora de realizar su detección alcanzo un valor medio de 84.87%. En la Tabla 4.10 se recogen los valores de la media y de la desviación estándar de

	Media	Desviación estándar
AP Missing hole	0.9847	0.0040
AP Mouse bite	0.9474	0.0128
AP Open circuit	0.8487	0.0193
AP Short	0.9596	0.0088
AP Spur	0.9453	0.0121
AP Spurious copper	0.9408	0.0093
mAP	0.9378	0.0059

Tabla 4.10: Media y desviación estándar de las APs de cada defecto y la mAP de la RetinaNet basada en la red ResNet50 durante la validación cruzada.

Los resultados obtenidos por la red RetinaNet basada en la ResNet 50 consiguió unos resultados inferiores a la red TDD expuesta en al artículo de Ding, R. (2019), el cual obtuvo una mAP de 98.90%. Cabe destacar que ambos estudios no son comparables, puesto que en el planteado en este subapartado emplea la técnica de la validación cruzada de 5 iteraciones, mientras que el de la red TDD se realiza una evaluación dividiendo el *dataset* en un conjunto de entrenamiento y otro de test.

4.4.5 Ejemplos de predicciones de la RetinaNet basada en ResNet 50

A continuación, se muestran algunos ejemplos de detección de defectos correcta realizada por este detector.

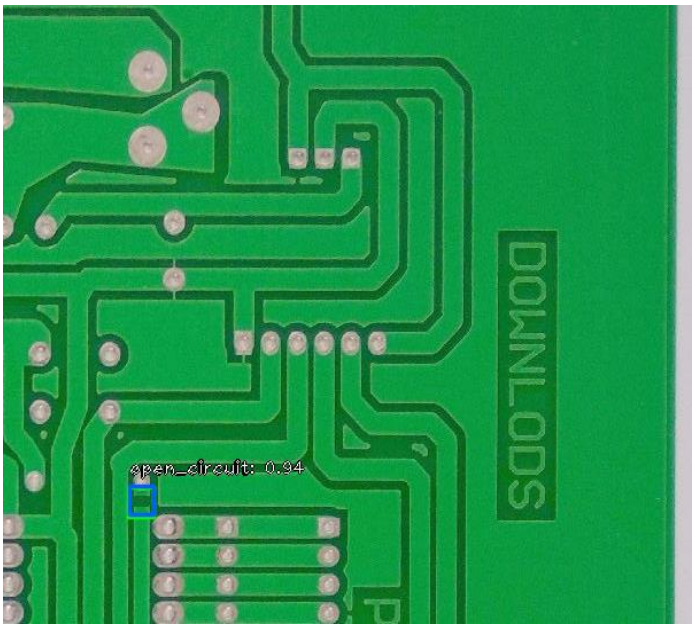


Figura 4.19: Detección correcta de un circuito abierto (*ground truth* color verde y *prediction box* azul)

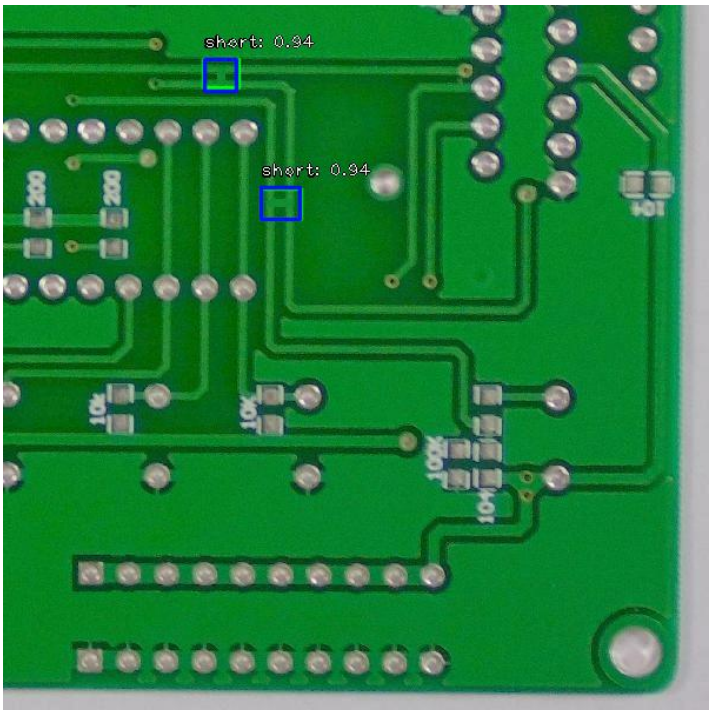


Figura 4.20: Detección correcta de dos cortos (*ground truth* color verde y *prediction box* azul)

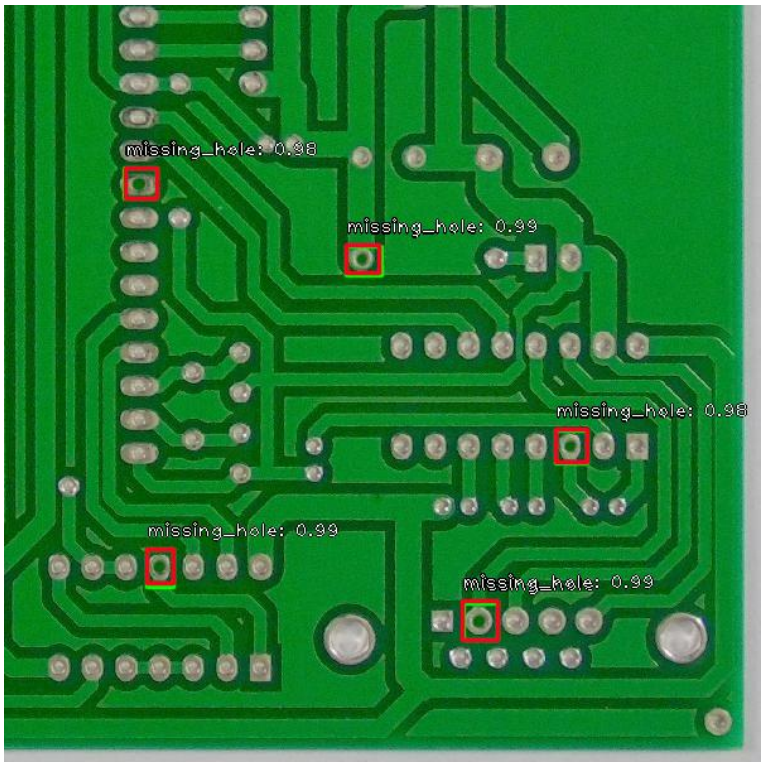


Figura 4.21: Detección correcta de 5 *missing hole* (*ground truth* color verde y *prediction box* rojo)

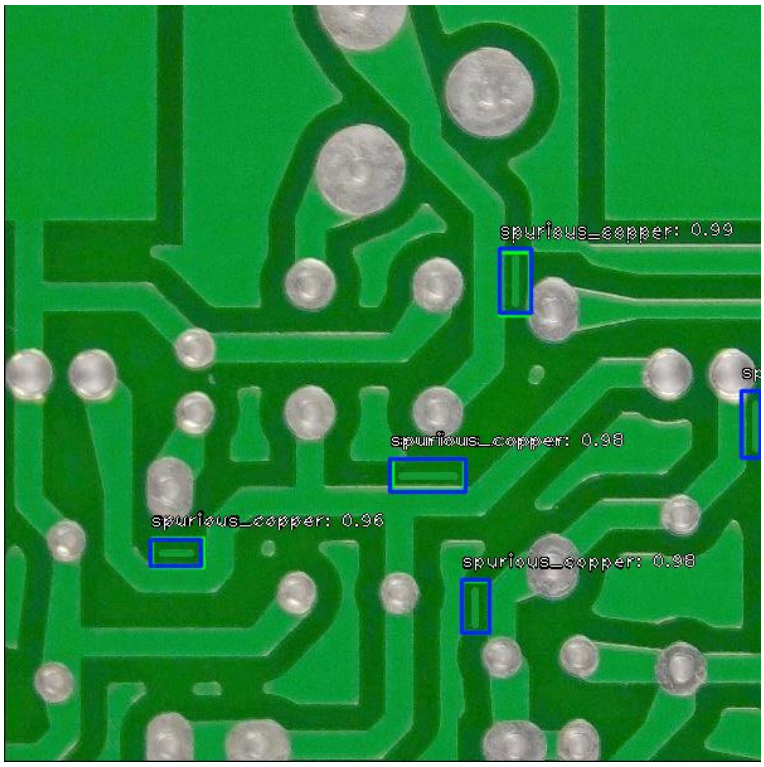


Figura 4.22: Detección correcta de 5 espurios de cobre (*ground truth* color verde y *prediction box* azul)

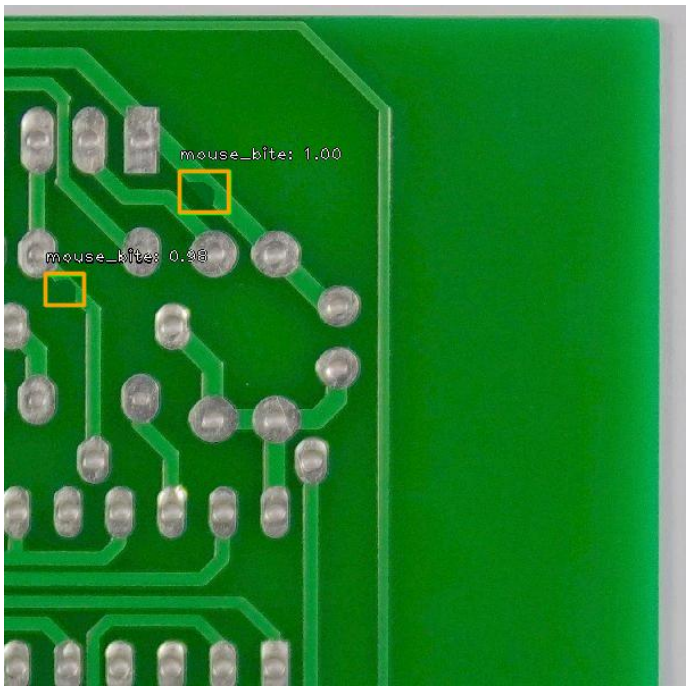


Figura 4.23: Detección correcta de 2 mordidas de ratón (*ground truth* color verde y *prediction box* naranja)

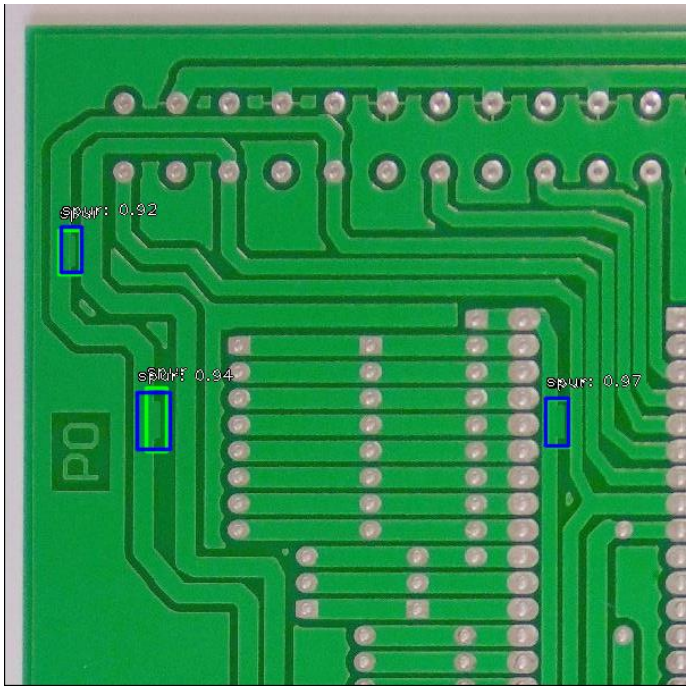


Figura 4.24: Detección correcta de 3 espurios (*ground truth* color verde y *prediction box* azul)

Por otro lado, los principales fallos que se pueden producir a la hora de realizar la predicción de los defectos son:

En primer lugar, el detector puede ser incapaz de detectar alguno de los defectos tal y como se muestra en la Figura 4.25. En dicha figura hay un *missing hole* que no ha sido detectado por el clasificador dado que como puede observarse, hay un *missing hole* que carece de *prediction box* (caja de color rojo).

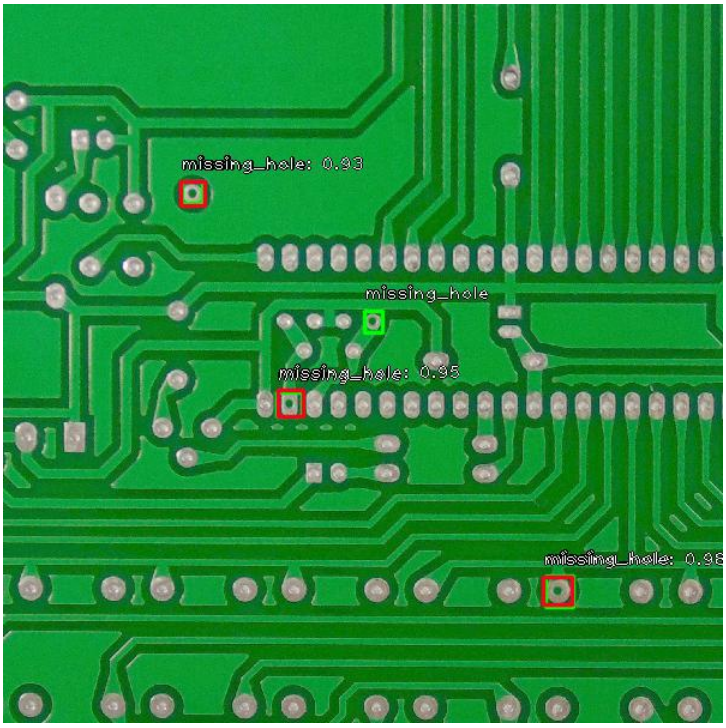


Figura 4.25: No detección de un *missing hole*.

Otro de los posibles defectos es la detección de un defecto varias veces, un ejemplo de este tipo de fallo se muestra en la Figura 4.26. En dicha figura uno de los *mouse bite* ha sido detectado dos veces y por lo tanto se marcan dos *prediction box* (caja naranja) para un mismo defecto.

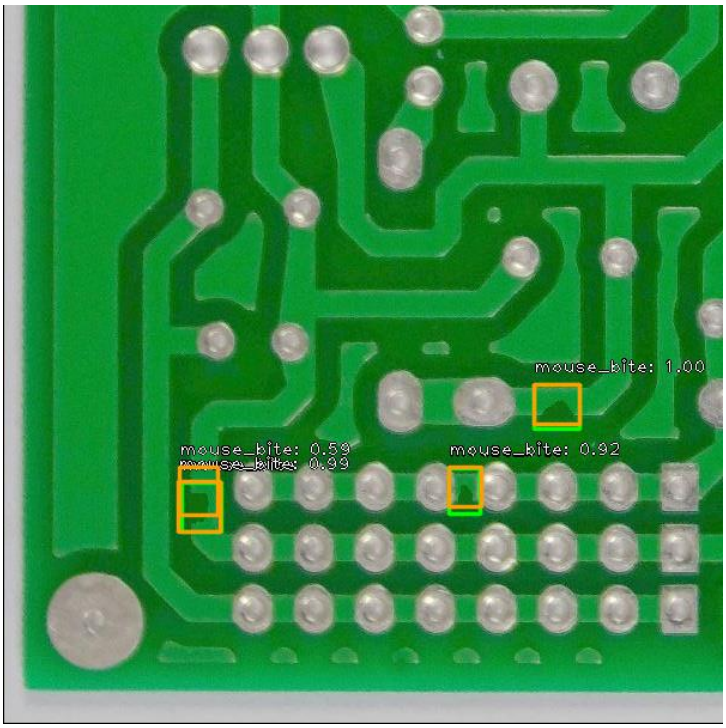


Figura 4.26: Detección doble de un *mouse bite*.

Por último, es posible que el detector detecte un defecto como otro diferente. En la Figura 4.27 se muestra un ejemplo de este tipo de fallos, en el cual se ha detectado un *spur* (caja de color azul) donde no había.

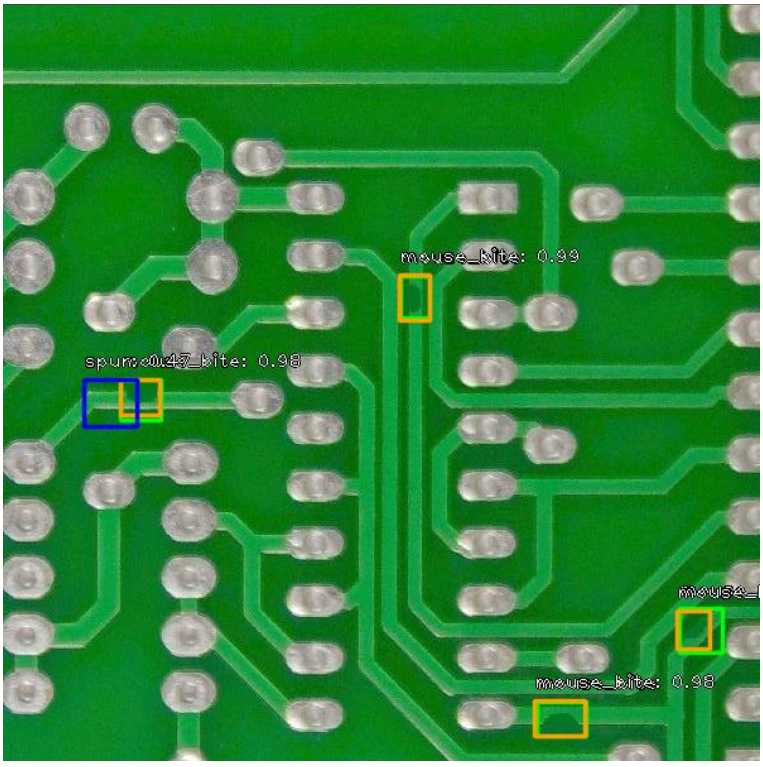


Figura 4.27: Detección incorrecta de un espurio (caja de color azul).

Capítulo 5. Conclusiones y líneas futuras

Para concluir con la memoria de este TFM, en este capítulo se exponen las principales conclusiones extraídas tras analizar los resultados obtenidos tanto para el clasificador de imágenes como para el detector de objetos. Además, se definen las limitaciones encontradas durante el desarrollo del trabajo y las futuras líneas de investigación que se pueden seguir.

5.1 Conclusiones

En este TFM se ha abordado el diseño de un detector de objetos basado en *Deep Learning* orientado a la Industria 4.0, capaz de detectar de forma automática los diferentes defectos presentes en algún producto industrial, este trabajo se centró en imágenes de PCBs. Para lograr el desarrollo del detector, se propuso como paso previo al desarrollo de un clasificador de imágenes defectuosas y no defectuosas, debido a que, aunque ambas tareas parezcan diferentes, están muy relacionadas entre sí.

Antes de exponer las conclusiones, hay que mencionar que se han cumplido todos los objetivos expuestos en el capítulo 1:

1. Se ha empleado el lenguaje de programación *Python* 3 junto a la biblioteca de aprendizaje automático *Tensorflow* mediante el software Anaconda haciendo uso de *Jupyter Notebook*.

2. Se ha realizado una revisión del estado de la técnica de los clasificadores de imágenes basados en *Deep Learning*, analizando diferentes arquitecturas de clasificadores y técnicas de preentrenamiento. Esta revisión ha sido la base empleada para desarrollar el modelo del clasificador propuesto en este trabajo.

3. Se ha efectuado una búsqueda de una base de datos de imágenes adecuada para el problema de clasificación planteado en este TFM. Para ello, se consultaron diferentes repositorios de bases de datos públicas como los ofrecidos en la página web *Kaggle*, los repositorios de *machine learning* UCI de la Universidad de California o el repositorio SNAP proporcionado por la Universidad de Standford.

4. Se han desarrollado diferentes algoritmos basados en *Deep Learning* de clasificación de imágenes defectuosas de piezas. Dichos algoritmos se entrenaron y evaluaron con la base de datos buscada anteriormente, obteniendo unos resultados muy satisfactorios, al nivel de modelos previos, pero con menor complejidad al tener un menor número de parámetros.

5. Se ha realizado una revisión del estado del arte de los algoritmos de detección de objetos basados en *Deep Learning*, consultando artículos científicos donde se exponen modelos populares y otros en los autores exponen sus propios modelos. Además, se han consultado diferentes repositorios de GitHub donde se proporcionan funciones o códigos para implementar modelos de detectores populares, y se han utilizado para emplear las arquitecturas SSD y RetinaNet.

6. Se ha buscado una base de datos de imágenes etiquetadas para la detección de objetos. En particular, se buscaron bases de datos de imágenes industriales que contengan defectos etiquetados.

7. Se han implementado detectores de objetos basados en la arquitectura de la SSD y RetinaNet. probando diferentes configuraciones para adaptarlos a la detección de defectos en imágenes de PCBs y evaluando sus prestaciones. En el Capítulo 6 se incluye un anexo donde se exponen fragmentos del código empleado para la implementación de las redes SSD y RetinaNet.

Una vez finalizado el trabajo, las conclusiones extraídas de este son:

1. La normalización de las imágenes antes de ser procesadas por el clasificador permite lograr mejores resultados que cuando no se aplica.

2. Todas las configuraciones del clasificador implementadas y evaluadas en este TFM han obtenido unos resultados muy buenos para el *dataset* empleado. No obstante, de las seis configuraciones propuestas la que mejores resultados logró fue la configuración número 3. Dicha configuración consta de cuatro capas convolucionales seguidas de sus respectivas capas de *maxpooling* y dos capas densas.

3. La técnica de *transfer learning* es más eficaz cuanto más parecidos son los nuevos datos a los datos con los que se entrenó el modelo anteriormente. Pero, aunque los datos no se parezcan a los del preentrenamiento, el uso de esta técnica mejora los tiempos de convergencia con respecto al entrenamiento desde cero de la red en la mayoría de los casos.

4. Ajustando las escalas y relaciones de aspectos de las capas de la SSD se puede mejorar los resultados a la hora de detectar objetos de pequeño tamaño.

5. El detector basado en la SSD para la detección de defectos en PCBs consigue mejores resultados y rendimiento utilizando como red base una red más simple que cuando se usa la VGG-16.

6. La arquitectura RetinaNet logra mejores resultados que la SSD, aunque hay que tener cuidado a la hora de configurar los umbrales de la IoU para evitar la detección de defectos en regiones donde no hay.

5.2 Limitaciones y líneas futuras

A pesar de que los resultados obtenidos tanto para el clasificador de imágenes defectuosas como para el detector de defectos en PCBs han sido favorables, los modelos empleados en este TFM presentan algunas limitaciones.

En el caso del clasificador, la principal limitación que presenta está en la clasificación correcta de imágenes que presentan defectos no muy visibles como rayaduras o muescas cercanas al aro de la bomba de color oscuro. Además, el método empleado

para la carga de imágenes está poco optimizado y tarda bastante tiempo en cargar las imágenes.

Por otro lado, el detector de objetos basado en la SSD presenta limitaciones a la hora de detectar objetos de pequeño tamaño, aunque se reduzcan las escalas, las relaciones de aspectos o aumentando la resolución de las imágenes de entrada.

En lo que respecta al detector basado en RetinaNet consigue muy buenos resultados, mucho mejores que el modelo basado en la SSD, pero aun así presenta algunas limitaciones. La principal limitación que presenta este modelo es la dificultad para detectar ciertos defectos como cortocircuitos.

Por último, a raíz de los resultados y las limitaciones expuestas, surgen una serie de líneas futuras de investigación a partir del trabajo realizado. En primer lugar, sería interesante la evaluación del clasificador diseñado con otros *datasets* de imágenes de otras áreas como imágenes médicas o geográficas para estudiar su rendimiento en otras áreas ajenas a la Industria 4.0. En segundo lugar, sería interesante poder realizar una validación cruzada para el detector RetinaNet basado en la red ResNet 101, el cual no ha podido realizarse en este trabajo por falta de tiempo y de potencia de computación del ordenador de trabajo.

Por otro parte, sería conveniente evaluar los detectores con otros *datasets* de imágenes industriales diferentes a PCBs. Tanto la SSD como la RetinaNet sería muy interesante realizar modificaciones para permitir el uso de cualquier CNN como red base del detector y no tener que emplear los predefinidos por el creador del código.

Además, en un futuro sería interesante emplear la nueva API de detección de objetos de *tensorflow* en lugar de recurrir a repositorios de *Github* donde se implementan diferentes modelos de detectores de objetos. La API de detección de objetos añadió soporte para *tensorflow* 2 el 10 de julio de 2020 (Blog, 2020).

Capítulo 6. Anexo

6.1 Hardware y software

Para el desarrollo de este trabajo se empleó un ordenador portátil personal, con Windows 10, un procesador i7-6700u de cuatro núcleos a 3.6 GHz y una tarjeta gráfica Nvidia gtx950M compatible con CUDA. Además, se empleó el siguiente software:

- Python 3.7.7, 32bits.
- Anaconda, se instaló la última versión de 64 bits compatible con Python 3.7.
- Jupyter Notebook 6.0.3.
- Tensorflow-gpu 2.1.1.
- CUDA 10.1.

Además del uso del ordenador personal, también se empleó la herramienta de procesado en la nube Google Colab para agilizar la evaluación de los modelos pudiendo ejecutar dos instancias a la vez, una en el ordenador y otra en la nube. Para poder emplear Google Colab solo se necesita una cuenta de Google.

6.2 Instalación de los repositorios

6.2.1 SSD

Como se comentó en el apartado 4.3.1, para implementar una red SSD se recurrió al repositorio de Github de Pierluigi Ferrari llamado ssd_keras, donde se implementa la SSD descrita en el artículo de Wei Liu (2016) titulado “SSD: Single Shot Multibox Detector”. A dicho repositorio se puede acceder mediante este enlace: https://github.com/pierluigiferrari/ssd_keras.

El autor del repositorio proporciona una serie de ficheros ipynb en los cuales presenta ejemplos de cómo se implementa el entrenamiento, la evaluación o la inferencia de la SSD, estos ficheros se emplearon como referencia para desarrollar el código para implementar la SSD, entrenarla y evaluarla. Además, se incluyen ficheros de Python que contienen funciones para trabajar con la SSD, estos ficheros se dividen en las siguientes carpetas:

- bounding_box_utils: En esta carpeta se ubica un fichero .py que permite la configuración y creación de los bounding box encargados de detectar los objetos.
- data_generator: contiene diferentes ficheros .py que contienen funciones diseñadas para trabajar con las imágenes, como métodos de carga de imágenes o redimensionamiento de imágenes entre otras.
- eval_utils: en este directorio se encuentran ficheros .py que contienen funciones para la evaluación de los modelos.

• keras_layers: contiene ficheros para la creación de los *anchor boxes* y capas que apliquen la normalización.

• models: En esta carpeta se encuentran localizados los ficheros .py de los tres tipos de SSD implementados en este repositorio, la SSD_300, SSD_512 y SSD_7.

• ssd_encoder_decoder: en esta carpeta se localizan unos ficheros .py que contienen información para codificar y decodificar los *anchor boxes* y los ground thrut en el formato que se necesite.

• Images: Esta carpeta se crea para almacenar las imágenes del *dataset* empleado.

6.2.2 RetinaNet

Para implementar una red RetinaNet de forma sencilla se empleó el repositorio de GitHub creado por Fizyr llamado keras-retinanet, donde se implementa la RetinaNet descrita por Tsung-Yi Lin et al. en el paper (2017) titulado “Focal Loss for Dense Object Detection”, se puede acceder al repositorio mediante el siguiente enlace: https://github.com/fizyr/keras-retinanet.

Este repositorio se caracteriza por presentar ficheros .py simialares a los de la API de Object Detection, que permiten la ejecución del entrenamientos, test o evaluación del modelo directamente desde la línea de comandos o desde una celda de Jupyter Notebook pasándole los flags adecuados en su llamada.

El repositorio de RetinaNet organiza sus ficheros en diferentes carpetas, las carpetas que se van a emplear en este trabajo son las siguientes:

• images: en esta carpeta se ubicarán las imágenes del *dataset* propio junto con los ficheros xml con las anotaciones asociadas a las imágenes.

• snapshots: en este directorio se almacenan los modelos guardados durante el entrenamiento del modelo.

• Log: ubicación donde se almacenan los logs generados durante el entreanimento.

• keras_retinanet: en esta carpeta se ubican todos los ficheros de Python fundamentales para trabajar con la red RetinaNet, estos se organizan en carpetas en relación con la función de estos ficheros, los más importantes son:

- bin: En esta carpeta se almacenan los ficheros para realizar el entrenamiento y evaluación del modelo.
- callbacks: En este directorio se ubican archivos que muestra por pantalla los resultados obtenidos en cada paso y época durante el entrenamiento.
- layers: los ficheros ubicados en esta carpeta son necesarios para configurar los *anchor boxes* del modelo.

- models: Esta carpeta contiene ficheros que implementan la red RetinaNet y diferentes redes populares que pueden emplearse como red troncal de la RetinaNet.
- preprocessing: Los ficheros localizados en esta carpeta se emplean para leer las imágenes o generar los ficheros de las anotaciones entre otras cosas. En este trabajo no se emplean puesto que se implementó otra forma de generar los ficheros de anotaciones.
- utils: En esta carpeta se localizan diferentes ficheros empleados para diferentes utilidades variadas, como ajustes de colores de las cajas, transformaciones que se pueden aplicar a las imágenes, etc.

Adicionalmente se creó una carpeta denominada modelos donde se ubican los modelos preentrenados que se emplean para aplicar *transfer learning*.

6.3 Instalación de los repositorios

6.3.1 Local

Para poder emplear los ficheros y funciones implementadas en los repositorios citados anteriormente para trabajar con las redes SSD y RetinaNet de forma más sencilla, únicamente hay que clonar los repositorios de GitHub en un directorio del ordenador donde se trabaja.

6.3.2 Google Colab

Para poder usar los repositorios desde Google Colab es necesario disponer de una cuenta de Google. En primer lugar, se debe copiar el repositorio del modelo en una carpeta de Google Drive, a continuación, en la primera celda del cuaderno debe de introducir el fragmento de código que se muestra en la Figura 6.1 para montar el Google Drive en la máquina virtual.

```
from google.colab import drive
drive.mount('/content/drive')
```

Figura 6.1: Código para montar el Drive en Google Colab

El código anterior genera un enlace de salida y nos pide un código de autorización para montar el Drive, pinchando en el enlace y permitiendo el acceso a la cuenta de Google, se generará un código de autorización el cual se debe copiar y pegar en la petición realizada en el cuaderno.

Una vez montado el Drive, la ruta para acceder a los ficheros o carpetas de Drive es /content/drive/My Drive/.

Para emplear la GPU de la máquina virtual como entorno de ejecución para el cuaderno, se debe clickear en la opción “Entorno de ejecución->Cambiar tipo de entorno de ejecución” y en la ventana que se abre seleccionar GPU como acelerador por hardware y se guarda. De este modo, cuando se ejecute el cuaderno se empleará la GPU en lugar de la CPU si está disponible.

Durante este trabajo se empleó una cuenta de Google Colab gratuita, la cual tiene las siguientes limitaciones:

- No se dispone de prioridad para acceder a la GPU de la máquina virtual.
- La memoria RAM disponible de la máquina virtual está limitada a 12.72 Gb y un espacio de almacenamiento de 68.4 Gb.
- El tiempo de utilización de la máquina virtual está limitado a 30 horas.

6.4 Código

6.4.1 Creación de los conjuntos de imágenes y sus anotaciones

El dataset está formado por imágenes y ficheros xml que contienen las anotaciones de los objetos que contiene cada imagen. Para poder entrenar y evaluar correctamente los modelos es necesario generar un fichero que contenga las anotaciones de cada uno de los objetos, como ambos modelos soportan las anotaciones en formato csv se optó por emplear este tipo de ficheros.

Para generar los ficheros csv con las anotaciones de las imágenes se diseñaron dos funciones, xml_to_csv_SSD y xml_to_csv_retinanet. Se crearon dos funciones muy similares porque la codificación del identificador es diferente para cada red, la SSD emplea el identificador numérico mientras que la RetinaNet usa el nombre de la clase. A dichas funciones se les pasa una lista con el nombre de imágenes, para cada imagen se abre su fichero xml asociado, se leen los valores de interés y se almacenan en un *dataframe*. Ambas funciones devuelven un *dataframe* con la anotación de cada defecto por fila, el formato de las anotaciones es el siguiente:

<nombre de la imagen>, <Xmin>, <Ymin>, <Xmax>, <Ymax>, <identificador de la clase>

El código de estas dos funciones se muestra en la Figura 6.2 y Figura 6.3.

```
import xml.etree.ElementTree as ET

# Función lee los ficheros XML y almacena la informacion de los
objetos etiquetados en
# un fichero CSV
def xml_to_csv_SSD(files):
    path='images/chips/'
    xml_list = []
    clases = ['missing_hole', 'mouse_bite', 'open_circuit',
'short', 'spur', 'spurious_copper']
```

```
    # Recorre toda la lista de imágenes
    for file in files:
        x = file.split('.') # Corta la extension del fichero
        tree = ET.parse(x[0]+'.xml') #Añade al nombre la extensión
xml
        root = tree.getroot()
        for member in root.findall('object'): # recorre cada objeto
de la imagen etiquetado
            # Se almacenan los datos de cada objeto
            value = (x[0]+'.jpg',
                     int(member[4][0].text),
                     int(member[4][1].text),
                     int(member[4][2].text),
                     int(member[4][3].text),
                     int(clases.index(member[0].text)+1)) # La
clase se identifica con un id
            xml_list.append(value)
    column_name = ['image_name', 'xmin', 'ymin', 'xmax', 'ymax',
'class_id']
    xml_df = pd.DataFrame(xml_list, columns=column_name)
    return xml_df # Se devuelve el dataframe con todos los objetos
de la lista de imágenes
```

Figura 6.2: Código de la función xml_to_csv_SDD

```
import xml.etree.ElementTree as ET

def xml_to_csv_retinanet(files):
    path='images/chips/'
    xml_list = []
    for file in files:
        x = file.split('.')
        tree = ET.parse(x[0]+'.xml')
        root = tree.getroot()
        for member in root.findall('object'):
            value = (x[0]+'.jpg',
                     int(member[4][0].text),
                     int(member[4][1].text),
                     int(member[4][2].text),
                     int(member[4][3].text),
                     member[0].text
                     )
            xml_list.append(value)
    column_name = ['image_name', 'xmin', 'ymin', 'xmax', 'ymax',
'class']
    xml_df = pd.DataFrame(xml_list, columns=column_name)

    return xml_df
```

Figura 6.3: Código de la función xml_to_csv_retinanet

Para realizar el entrenamiento y evaluación de los detectores implementados, se decidió dividir el *dataset* en tres conjuntos diferentes: entrenamiento, validación y test, para ello se implementó el fragmento de código mostrado en la Figura 6.4. En dicho código, primero se listan todas imágenes del directorio del dataset, a continuación, se dividen las imágenes en los conjuntos de entrenamiento, validación y test. Por último, para cada conjunto de imágenes se llama a la función xml_to_csv, y se generan los ficheros csv con las anotaciones de todos los objetos etiquetados en las imágenes de cada conjunto. Además, se crea un csv que contiene cada clase de objetos con su identificador asociado.

```
list_img=[]
list_images=[]

# Se leen todas las imágenes del directorio
for file in glob.glob("images/chips/*.jpg"):
    list_img.append(file)
# Se generan los subconjuntos de entrenamiento, validación y test
training_set, test_set = train_test_split(list_img, test_size=0.2,
shuffle=True, random_state=42)
train_set, val_set = train_test_split(training_set, test_size=0.2,
shuffle=True, random_state=42)

# Se genera un fichero CSV para cada subconjunto con todos los
objetos etiquetados
# en las imágenes de cada subconjunto
df_train_SSD = xml_to_csv_SSD(train_set)
df_train_retinanet = xml_to_csv_retinanet(train_set)
df_train_SSD.to_csv('train.csv', index=False) # El csv en el formato
para la SSD
df_train_retinanet.to_csv('train_r.csv', header=False, index=False)
# El csv en el formato para la RetinaNet


df_val_SSD = xml_to_csv(val_set)
df_val_retinanet = xml_to_csv_retinanet(val_set)
df_val.to_csv('val.csv', index=False)
df_val_retinanet.to_csv('val_r.csv', header=False, index=False)

df_test_SSD = xml_to_csv(test_set)
df_test_retinanet = xml_to_csv_retinanet(test_set)
df_test.to_csv('test.csv', index=False)
df_test_retinanet.to_csv('test_r.csv', header=False, index=False)
```

Figura 6.4: Fragmento de código que crea los subconjuntos de imágenes y sus anotaciones.

Todas estas funciones y fracciones de códigos se implementan en el fichero create_sets_annotations.ipynb.

6.4.2 SSD

6.4.2.1 Configuración y creación del modelo

El primer paso para implementar una red SSD es definir los diferentes parámetros que la definen, en la Figura 6.5 se muestra el fragmento de código donde se definen esos parámetros.

```
img_height = 300 # Altura de la imagen de entrada al modelo
img_width = 300 # Anchura de la imagen de entrada al modelo
img_channels = 3 # Canales de la imagen de entrada al modelo
mean_color = [123, 117, 104] # La media por canal de la imágen. ESTE
VALOR NO DEBE MODIFICARSE SI SE USAN MODELOS PREENTRENADOS
swap_channels = [2, 1, 0] # El orden con el que se organizan los
canales de las imágenes en la SSD
n_classes = 20 # Número de clases para detectar
scales_pascal = [0.1, 0.2, 0.54, 0.71, 0.88, 1.05] # El factor de
escala de los anchor box de la SSD300 original para el dataset
Pascal VOC
scales_coco = [0.00164, 0.0033, 0.51, 0.69, 0.87, 1.05] # El factor
de escala de los anchor box de la SSD300 original para el dataset MS
COCO
scales = scales_pascal # Se escoge cuales de las dos escalas
anteriores se emplea
aspect_ratios = [[1.0, 2.0, 0.5],
                 [1.0, 2.0, 0.5, 3.0, 1.0/3.0],
                 [1.0, 2.0, 0.5, 3.0, 1.0/3.0],
                 [1.0, 2.0, 0.5, 3.0, 1.0/3.0],
                 [1.0, 2.0, 0.5],
                 [1.0, 2.0, 0.5]] # Relación de aspecto para cada
capa del modelo
aspect_ratios_global = [1.0, 2.0, 0.5] # Relación de escalas para
todas las capas
two_boxes_for_ar1 = True
steps = [8, 16, 32, 64, 100, 300] # Es espacio entre dos centros de
dos anchor box adyacentes
offsets = [0.5, 0.5, 0.5, 0.5, 0.5, 0.5] # Los desplazamientos de
los puntos centrales del primer anchor box desde los bordes superior
e izquierdo de la imagen como una fracción del tamaño del paso para
cada capa de predicción.
clip_boxes = False # Si se deben o no recortar los anchor box para
que esten todos dentro de la imagen
variances = [0.1, 0.1, 0.2, 0.2] # Las variaciones por las cuales
las coordenadas codificadas se dividen como en la implementación
original
normalize_coords = True # Si las coordenadas de los anchor box tienen
que ser normalizadas
```

Figura 6.5: Definición y configuración de los parámetros de la SSD

Una vez definidas las características de la SSD, se procede a la creación del modelo, para crear el modelo se hace uso de las funciones implementadas en los ficheros ssd300.py, ssd512.py o ssd7.py tal y como se muestra en la Figura 6.6. Una vez creado el modelo, se define el optimizador y la función de pérdidas, esta última se implementa mediante una función proporcionada por el autor del repositorio y se corresponde con la función de perdida expuesta en el apartado 4.3.1.1.

```
# Eliminamos los modelos de keras que pueda haber creados
previamente
K.clear_session()
# Llamada a la función encargada de crear la SSD 300 junto con los
parámetros de configuración
model = ssd_300(image_size=(img_height, img_width, img_channels),
                n_classes=n_classes,
                mode='training',
                l2_regularization=0.0005,
                scales=scales,
                aspect_ratios_per_layer=aspect_ratios,
                two_boxes_for_ar1=two_boxes_for_ar1,
                steps=steps,
                offsets=offsets,
                clip_boxes=clip_boxes,
                variances=variances,
                normalize_coords=normalize_coords,
                subtract_mean=mean_color,
                swap_channels=swap_channels)

weights_path = 'C:/Users/josem/TFM/TFM_SSD/ssd_keras-
master/pesos/preentrenado/VOC/VGG_VOC0712_SSD_300x300_iter_120000.h5
'
model.load_weights(weights_path, by_name=True) # Se cargan los pesos
ya preentrenados con el dataset PASCAL VOC
#Se configuran los diferentes optimizadores
adam = Adam(lr=0.001, beta_1=0.9, beta_2=0.999, epsilon=1e-08,
decay=0.0)
sgd = SGD(lr=0.001, momentum=0.9, decay=0.0, nesterov=False)
# Llamada a la función SSDLoss que crea la función de pérdida de la
SSD
ssd_loss = SSDLoss(neg_pos_ratio=3, alpha=1.0)
model.compile(optimizer=adam, loss=ssd_loss.compute_loss)
```

Figura 6.6: Código de creación del modelo, carga de los pesos preentrenados y compilación del modelo

6.4.2.2 Creación de los *generators*

Para leer las imágenes y las anotaciones y prepararlas para que el modelo pueda procesarlas se emplean funciones diseñadas por el autor del repositorio para esta finalidad. En la Figura 6.7 se muestra el código para crear los generadores de datos para el entrenamiento y validación del modelo. Para ello, primero se inicializan los generadores para los dos subconjuntos, a continuación, en el código se definen las funciones de transformaciones que se aplican a las imágenes de entra, estas transformaciones son la data *augmentation*, la redimensión y el cambio de número de canales de las imágenes. Por último, se configura la codificación de las *ground truth* de los objetos y se generan los *generators* de las imágenes y anotaciones.

```
predictor_sizes =
model.get_layer('conv4_3_norm_mbox_conf').output_shape[1:3],
                    model.get_layer('fc7_mbox_conf').output_shape[1:3],
                    model.get_layer('conv6_2_mbox_conf').output_shape[1:3],
                    model.get_layer('conv7_2_mbox_conf').output_shape[1:3],
                    model.get_layer('conv8_2_mbox_conf').output_shape[1:3],
                    model.get_layer('conv9_2_mbox_conf').output_shape[1:3]]

train_dataset = DataGenerator(load_images_into_memory=False,
hdf5_dataset_path=None)
val_dataset = DataGenerator(load_images_into_memory=False,
hdf5_dataset_path=None)

# Se definen el conjunto de transformaciones sobre las imágenes
# Data Augmentation
data_augmentation_chain = DataAugmentationSatellite(resize_height=img_height,
                                                    resize_width=img_width,
                                                    random_hue=(18, 0.5),
                                                    random_flip=0.5)
# Redimensionaminto de las imágenes
resize = Resize(height=img_height, width=img_width)

# Ajuste del número de canales
convert_to_3_channels = ConvertTo3Channels()

# Se realiza una codificación de los ground thrut de los objetos
ssd_input_encoder = SSDInputEncoder(img_height=img_height,
                                    img_width=img_width,
                                    n_classes=20,
                                    predictor_sizes=predictor_sizes,
                                    scales=scales,
                                    aspect_ratios_per_layer=aspect_ratios,
                                    two_boxes_for_ar1=two_boxes_for_ar1,
                                    steps=steps,
                                    offsets=offsets,
                                    clip_boxes=clip_boxes,
                                    variances=variances,
                                    matching_type='multi',
                                    pos_iou_threshold=0.5,
                                    neg_iou_limit=0.3,
                                    normalize_coords=normalize_coords)
# Se crean los generators de entrenamiento y validación que se emplearan
# para el entrenamiento
train_generator = train_dataset.generate(batch_size=batch_size,
                                         shuffle=True,
                                         transformations=[resize],
                                         label_encoder=ssd_input_encoder,
                                         returns={'processed_images',
                                                  'encoded_labels'},
                                         keep_images_without_gt=False)
val_generator = val_dataset.generate(batch_size=batch_size,
                                     shuffle=True,
                                     transformations=[resize],
                                     label_encoder=ssd_input_encoder,
                                     returns={'processed_images',
                                              'encoded_labels'},
                                     keep_images_without_gt=False)
```

Figura 6.7: Código de la adquisición de imágenes y *ground truth*

6.4.2.3 Entrenamiento de la SSD

El entrenamiento del modelo se realiza mediante el método fit, para ello hay que pasar los *generators* de entrenamiento y validación creados anteriormente y configurar los parámetros del entrenamiento. En la Figura 6.8 se muestra un ejemplo del código que se emplea para entrenar el modelo.

```
initial_epoch   = 0
final_epoch     = 1000
steps_per_epoch = 1000
# Se realiza el entrenamiento y validación del modelo
history = model.fit(train_generator,
                    steps_per_epoch=ceil(train_dataset_size/batch_size),
                    callbacks=callbacks,
                    epochs=final_epoch,
                    validation_data=val_generator,
                    validation_steps=ceil(val_dataset_size/batch_size),
                    initial_epoch=initial_epoch)
```

Figura 6.8: Código del entrenamiento del modelo de la SSD

6.4.2.4 Carga del modelo en modo inferencia

El primer paso para testear y evaluar un modelo es cargarlo en modo inferencia, para ello se vuelve a crear un modelo de la SSD igual que se hacía para el entrenamiento, pero en esta ocasión se cambia el modo de *training* a *inference*. Una vez creado, se cargan los pesos del modelo entrenado previamente, y se compila el modelo. Todo este proceso implementa mediante el código de la Figura 6.9.

```
K.clear_session() # Se eliminan los modelos previos en memoria
# Se crea el modelo SSD en base a los aprametros de configuración
model = ssd_300(image_size=(img_height, img_width, img_channels),
                n_classes=n_classes,
                mode='inference',
                l2_regularization=0.0005,
                scales=scales,
                aspect_ratios_per_layer=aspect_ratios,
                two_boxes_for_ar1=two_boxes_for_ar1,
                steps=steps,
                offsets=offsets,
                clip_boxes=clip_boxes,
                variances=variances,
                normalize_coords=normalize_coords,
                subtract_mean=mean_color,
                swap_channels=swap_channels)
# Se cargan los pesos del modelo entrenado
weights_path = 'evaluacion/ssd300_chips_conf1.h5'
model.load_weights(weights_path, by_name=True)

adam = Adam(lr=0.001, beta_1=0.9, beta_2=0.999, epsilon=1e-08, decay=0.0)
ssd_loss = SSDLoss(neg_pos_ratio=3, alpha=1.0)
# Se compila el modelo
model.compile(optimizer=adam, loss=ssd_loss.compute_loss)
```

Figura 6.9: Código de creación y carga del modelo en modo inferencia

6.4.2.5 Creación de los *generators* de test

Este proceso se realiza del mismo modo que se crean los *generators* para el entrenamiento del modelo.

```
test_anotation = 'C:/Users/josem/TFM/TFM_SSD/ssd_keras-master/test.csv'
path_images = 'C:/Users/josem/TFM/TFM_SSD/ssd_keras-master/'
test_dataset = DataGenerator(load_images_into_memory=False,
hdf5_dataset_path=None)
#Transformaciones de las imágenes
resize = Resize(height=img_height, width=img_width)
convert_to_3_channels = ConvertTo3Channels()
# Se genera el dataset
test_dataset.parse_csv(images_dir=path_images,
                  labels_filename=test_anotation,
                  input_format=['image_name', 'xmin', 'ymin', 'xmax',
'ymax', 'class_id'],
                  include_classes='all')
# Se crea el generator
predict_generator = test_dataset.generate(batch_size=20,
                                          shuffle=True,
                                          transformations=[resize,
                                          convert_to_3_channels],
                                          label_encoder=ssd_input_encoder,
                                          returns={'processed_images',
                                                   'processed_labels',
                                                   'filenames'},
                                          keep_images_without_gt=False)
```

Figura 6.10: Código de carga de las imágenes y *ground truth* de test

6.4.2.6 Evaluación del modelo

Para evaluar el modelo, se emplean dos funciones diseñadas por el autor, de modo que se realiza el testeo del modelo y con los resultados se computan las métricas, proporcionando la mAP, la precisión y la *recall* de cada clase. En la Figura 6.11 se muestra el código que efectúa la evaluación.

```
# Se realiza la evaluación del modelo
evaluator = Evaluator(model=model,
                      n_classes=20,
                      data_generator=test_dataset,
                      model_mode=model_mode)
# Se computan los resultados de la evaluación
results = evaluator(img_height=img_height,
                    img_width=img_width,
                    batch_size=8,
                    data_generator_mode='resize',
                    round_confidences=False,
                    matching_iou_threshold=0.5,
                    border_pixels='include',
                    sorting_algorithm='quicksort',
                    average_precision_mode='sample',
```

```
                    num_recall_points=11,
                    ignore_neutral_boxes=True,
                    return_precisions=True,
                    return_recalls=True,
                    return_average_precisions=True,
                    verbose=True)

# Se separan los resultados
mean_average_precision, average_precisions, precisions, recalls = results
```

Figura 6.11: Código de la evaluación del modelo

6.4.3 RetinaNet

6.4.3.1 Entrenamiento de RetinaNet

Para entrenar el modelo se emplea el fichero train.py que contiene las funciones para crear el modelo en función de los *flags* que se le pasen en la llamada. En el fichero train.py se encuentran comentadas todas las *flags* que se pueden pasar para configurar el entrenamiento de la RetinaNet. Algunas de las *flags* más importantes son:

- `--freeze-backbone`: Se congelan las capas del *backbone* del modelo.
- `--weights`: Se le indica la dirección del fichero que contiene los pesos de un modelo prenetrenado.
- `--batch-size`: Sirve para configurar el tamaño del *batch*.
- `--compute-val-loss`: Esta opción permite el cálculo de la pérdida del conjunto de validación en cada época.
- `--steps`: Fija el número de pasos del entrenamiento.
- `--epochs`: Indica el número de épocas del entrenamiento.
- `csv`: indica a la función de entrenamiento que las anotaciones de las imágenes de entrenamiento y el fichero que contiene las clases y sus identificadores están en formato csv.
- `--val-annotations`: pasa el fichero que contiene las anotaciones de las imágenes de validación.

En la Figura 6.12 se muestra un ejemplo del código de llamada a la función train.py

```
#Entrenado con la base de dados COCO
modelo_preentrenado='./modelos/preentrenado/resnet50_coco_best_v2.1.0.h5'

# Se ejecuta el .py de entrenamiento pasándole los flags con la
configuración y parámetros del entrenamiento

%run keras_retinanet/bin/train.py --freeze-backbone -backbone resnet50 --
weights {modelo_preentrenado} --compute-val-loss --tensorboard-dir "" --
batch-size 4 --steps 854 --epochs 10000 csv train.csv
anotaciones/carts/clases.csv --val-annotations val.csv
```

Figura 6.12: Código del entrenamiento de la red RetinaNet

6.4.3.2 Test y evaluación del modelo

Para realizar la evaluación de la RetinaNet se emplea el fichero Python evaluate.py, el cual dispone de un conjunto de funciones para ello. La evaluación puede configurarse mediante un conjunto de flags que se le pasan cuando se realiza la llamada, en el propio fichero el creador del repositorio incluye en forma de comentario todas las posibles flags que se pueden emplear y que datos hay que pasarles. Algunos de los flags empleados para la evaluación son:

- `--score-threshold`: Fija el valor del umbral de la puntuación mínima a partir de la cual el detector considera que dicho anchor box contiene un objeto.
- `--iou-threshold`: Fija el valor del umbral de la IoU del detector.
- `--save-path`: Esta opción permite que, tras realizada la detección de objetos, se guarde la imagen con los *anchor boxes* detectados por el detector y los ground truth. Esta opción tiene que ir seguida del directorio donde se guardaran las imágenes.
- `--convert-model`: Esta opción realiza la conversión del modelo de modo entrenamiento a inferencia de forma interna.
- `csv`: Se le pasa en formato csv.

En la Figura 6.13 se muestra un ejemplo de llamada a evaluate.py

```
%run keras_retinanet/bin/evaluate.py --score-threshold 0.45 --iou-threshold
0.5   --save-path   images\predicciones\chips_iou65   --convert-model   csv
test.csv anotaciones/carts/clases.csv snapshots/retinanet_tl_chips.h5
```

Figura 6.13: Código para la evaluación de la RetinaNet

Glosario de acrónimos

AP: Average Precision

AUC: Area Under the Curve

CNN: Redes neuronales convolucionales

FC: Capa totalmente conectada

FPN: Features Pyramid Network

IA: Inteligencia Artificial

ILSVRC: ImageNet Large Scale Visual Recognition Challenge

IoT: Internet of Things

IoU: Intersection Over Union

mAP: Mean Average Precision

RNA: Red Neuronal Artificial

SSD: Single Shot Multibox Detector

TDD: Tiny Defect Detector

TFM: Trabajo Fin de Máster

TIC: Tecnologías de la información y la comunicación.

VGG: Visual Geometry Group

Referencias

Anwar, S. M. (2018). Medical Image Analysis using Convolutional Neural Networks: A Review. *Journal of Medical Systems*.

atriainnovation. (2019, Octubre 3). *Deep Learning y sus muchas aplicaciones.* Retrieved from atriainnovation: https://www.atriainnovation.com/deep-learning-aplicaciones/#:~:text=En%20la%20actualidad%20se%20hace,de%20voz%3B%20clasificaci%C3%B3n%20de%20correos

Barrios, A. (2019, Julio 26). *La matriz de confusión y sus métricas.* Retrieved from juanbarrios.com: https://www.juanbarrios.com/matriz-de-confusion-y-sus-metricas/

Blog, T. (2020, Septiembre 18). *TensorFlow 2 meets the Object Detection API*. Retrieved from TensorFlow Blog Web Site: https://blog.tensorflow.org/2020/07/tensorflow-2-meets-object-detection-api.html

Burgueño, M. J.-B.-B. (1995). Las curvas ROC en la evaluación. *Med Clin (Barc)*, 661-670.

Calvo, D. (2017, Julio 20). *Red Neuronal Convolucional CNN.* Retrieved from diegocalvo web site: https://www.diegocalvo.es/red-neuronal-convolucional/

Caparrini, F. S. (2019, Diciembre 14). *Redes Neuronales: una visión superficial.* Retrieved from Dpto. de Ciencias de la Computación e Inteligencia Artificial Universidad de Sevilla: http://www.cs.us.es/~fsancho/?e=72

Cowley, J. (2018, Diciembre 7). *Redes neuronales convolucionales.* Retrieved from IBM Developer: https://developer.ibm.com/es/technologies/artificial-intelligence/articles/cc-convolutional-neural-network-vision-recognition/#

Dabhi, R. (2020, Enero 23). *kaggle*. Retrieved from casting product image data for quality inspection: https://www.kaggle.com/ravirajsinh45/real-life-industrial-dataset-of-casting-product

de Ullibarri Galparsoro, L. &. (1998). Curvas ROC. *Atención Primaria en la Red*, 229-235.

Ding, R. D. (2019). TDD-net: a tiny defect detection network for printed circuit boards. *CAAI Transactions on Intelligence Technology, 4(2)*, 110-116.

Evaluate: COCO. (2020, 9 17). Retrieved from COCO web site: https://cocodataset.org/#detection-eval

F. Shrouf, J. O. (2014). Smart factories in Industry 4.0: A review of the concept and of energy management approached in production based on the Internet of Things paradigm. *IEEE International Conference on Industrial Engineering and Engineering Management, Bandar Sunway*, 697-70.

FA Softmax. (2020). Retrieved from Numerentur.org web site: http://numerentur.org/funcion-de-activacion-softmax/

Flórez, L. R., & Fernández, F. J. (2008). *Las redes neuronales artificiales: : fundamentos teóricos y aplicaciones prácticas.* Oleiros, La Coruña.: Netbiblo.

Forson, E. (2017, Noviembre 18). *Understanding SSD MultiBox — Real-Time Object Detection In Deep Learning*. Retrieved from Toward data science web site:

https://towardsdatascience.com/understanding-ssd-multibox-real-time-object-detection-in-deep-learning-495ef744fab

fyzr. (2020, Septiembre 18). *github/fizyr/keras-retinanet*. Retrieved from Github.com: https://github.com/fizyr/keras-retinanet

Géron, A. (2019). *Hands-on Machine Learning with Scikit-Learn, Keras, and TensorFlow. Concepts, Tools, and Techniques to Build Intelligent Systems.* O'Reilly Media, Inc.

Grietens, J. (2018, abril 12). *What's the difference between Machine Learning & Deep (Machine) Learning?* Retrieved from Verhaert site web: https://verhaert.com/difference-machine-learning-deep-learning/

GuillaumeSimler. (2020, 9 18). *Casting Quality -> 98% training acc | 99,58% test.* Retrieved from kaggle: https://www.kaggle.com/guillaumesimler/casting-quality-98-training-acc-99-58-test

Haykin, S. (1994). *Neural Networks: A Comprehensive Foundation.* USA: Prentice Hall PTR.

Henderson, P. &. (2016). End-to-end training of object class detectors for mean average precision. *Asian Conference on Computer Vision*, 198-213.

Herrera, F. (2016). Big Data: Preprocesamiento y calidad de datos. *novática*, 7.

Hui, J. (2018, Marzo 7). *mAP (mean Average Precision) for Object Detection*. Retrieved from medium.com: https://medium.com/@jonathan_hui/map-mean-average-precision-for-object-detection-45c121a31173#:~:text=Open%20in%20app-,mAP%20(mean%20Average%20Precision)%20for%20Object%20Detection,value%20over%200%20to%201.

Hui, J. (2018, Marzo 14). *SSD object detection: Single Shot MultiBox Detector for real-time processing*. Retrieved from Medium Web site: https://medium.com/@jonathan_hui/ssd-object-detection-single-shot-multibox-detector-for-real-time-processing-9bd8deac0e06

ITELLIGENT. (2018, Enero 3). *Deep learning & Convolutional Neuronal Network: qué es y en qué consiste.* Retrieved from itelligent web site: https://itelligent.es/es/deep-learning-convolutional-neuronal-network-cnn-consiste/

Johnson, M. K. (2016). Applied Predictive Modeling. *Springer*.

Lin, T. Y. (2017). Focal loss for dense object detection. *Proceedings of the IEEE international conference on computer vision.*, 2980-2988.

Liu, W. A. (2016). Ssd: Single shot multibox detector. *European conference on computer vision* (pp. 21-37). Springer, Cham.

Lukač, D. (2015). The fourth ICT-based industrial revolution "Industry 4.0". *Telecommunications Forum Telfor*, 835-838.

Mathworks. (2020, Junio 12). *Deep Learning. Tres cosas que es necesario saber.* Retrieved from mathworks web site: https://es.mathworks.com/discovery/deep-learning.html#:~:text=El%20Deep%20Learning%20es%20una%20tecnolog%C3%ADa%20clave%20presente%20en%20los,televisores%20y%20a

MC.AI. (2018, Junio 8). *CNN Architectures — VGGNet*. Retrieved from MC.AI Web Site: https://mc.ai/cnn-architectures-vggnet/

Neurohive. (2018, Noviembre 20). *VGG16 – Convolutional Network for Classification and Detection*. Retrieved from Neurohive Web Site: https://neurohive.io/en/popular-networks/vgg16/

pierluigiferrari. (2020, septiembre 18). *github/pierluigiferrari/ssd_keras*. Retrieved from Github.com: https://github.com/pierluigiferrari/ssd_keras

Prechelt, L. (1998). Early Stopping - But When?. *Neural Networks: Tricks of the Trade. Lecture Notes in Computer Science, vol 1524*, 55-69.

prnoticias, r. (2016, Julio 21). *6 aplicaciones reales del deep learning.* Retrieved from prnoticias web site: https://historico.prnoticias.com/comunicacion/nombramientos/20154951-6-aplicaciones-reales-del-deep-learning

Raschka, S. (2020, 9 15). *Machine Learning FAQ: How do I evaluate a model?* Retrieved from https://sebastianraschka.com/faq/docs/evaluate-a-model.html

Rodrigo, J. A. (2016, Noviembre). *Validación de modelos de regresión: Cross-validation, OneLeaveOut, Bootstrap.* Retrieved from cienciadedatos.net: https://www.cienciadedatos.net/documentos/30_cross-validation_oneleaveout_bootstrap

Rodríguez, V. (2018, Noviembre 17). *Redes neuronales convolucionales*. Retrieved from Vicente Rodríguez Blog: https://vincentblog.xyz/posts/redes-neuronales-convolucionales

Saha, S. (2018, Diciembre 15). *A Comprehensive Guide to Convolutional Neural Networks — the ELI5 way.* Retrieved from Towards data science: https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53

scikit-learn.org. (2020, Junio 16). *3.1. Cross-validation: evaluating estimator performance*. Retrieved from scikit-learn API: https://scikit-learn.org/stable/modules/cross_validation.html

Silva T.B., M. E. (2020). Blockchain and Industry 4.0: Overview, Convergence, and Analysis. In A. A. Rosa Righi R., *Blockchain Technology for Industry 4.0. Blockchain Technologies* (pp. 27-58). Singapore: Springer, Singapore.

Simonyan, K. &. (2014). Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv*, 1409.1556.

SmartPanel. (2018, Abril 10). *¿Qué es el Deep Learning?* Retrieved from smartpanel web site: https://www.smartpanel.com/que-es-deep-learning/

Szegedy, C. R. (2014). Scalable, high-quality object detection. *arXiv preprint arXiv*, 1412-1441.

Torres, J. (2018). *Deep Learning Introdución práctica con keras (primera parte).* Barcelona: WHAT THIS SPACE. Retrieved from torres.ai web site.

Torres, J. (2019). *Deep Learning: Introducción práctica con Keras (segunda parte).* Barcelona: WHAT THIS SPACE.

Treelogic, E. (2019, Junio 17). *Control de calidad en la industria 4.0*. Retrieved from treelogic Web Site: https://www.treelogic.com/es/Control_Calidad_Industria40.html

Wei, J. (2019, Julio 3). *VGG Neural Networks: The Next Step After AlexNet*. Retrieved from Towards data science: https://towardsdatascience.com/vgg-neural-networks-the-next-step-after-alexnet-3f91fa9ffe2c

Y.S. Abu-Mostafa, M. M.-I. (2012). *Learning from Data: A Short Course.* amlbook.com.

Yang, L. (2017). Industry 4.0: A survey on technologies, applications and open research issues. *Journal of Industrial Information Integration, 6*, 1-10.

Yani, M. (2019). Application of transfer learning using convolutional neural network method for early detection of terry's nail. IOP Publishing.

Yanming Guo, Y. L. (2016). Deep learning for visual understanding: A review. *Neurocomputing*, 27-48.