TELECO.

CAMPUS SUR·UPM

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA Y SISTEMAS DE TELECOMUNICACIÓN

POLITÉCNICA

PROYECTO FIN DE GRADO

TÍTULO: Sistema para la detección de defectos en circuitos impresos mediante técnicas de aprendizaje automático

AUTOR/A: Le Liu

TITULACIÓN: Ingeniería de Electrónica de Comunicaciones

TUTOR/A: César Díaz Martín

DEPARTAMENTO: Ingeniería Audiovisual y Comunicaciones

VºBº TUTOR/A

Miembros del Tribunal Calificador:

PRESIDENTE/A: Óscar Ortiz Ortiz

TUTOR/A: César Díaz Martín

SECRETARIO/A: Elena Blanco Martín

Fecha de lectura: 18 de julio de 2024

Calificación:

El Secretario/La Secretaria,

Agradecimientos

A mi tutor, César Díaz Martín, por su orientación y apoyo constante durante este trabajo. Sus conocimientos y consejos fueron fundamentales para la realización de este proyecto.
A mis padres, Yu Liu y Weixia Liao, les estoy enormemente agradecido por su apoyo incondicional a lo largo de mi carrera académica.
A mis compañeros Xiangzhou, Yifan, Aohang, Javier y Yiyuan por su ayuda y amistad durante este proceso.

Resumen

El sector de las placas de circuito impreso, también conocidas como PCB, ha sido significativamente afectado por el veloz desarrollo de sectores tecnológicos como la informática, la electrónica y las telecomunicación en los últimos años. A medida que la demanda de PCBs aumenta y su complejidad crece, la industria se enfrenta al reto de mejorar su calidad, dado que incluso un pequeño defecto en la placa puede provocar graves fallos en el funcionamiento final del producto, lo que genera el desperdicio de materiales, energía y tiempo. Por tal razón, la detección efectiva y precisa de defectos de PCB es fundamental para evitar problemas durante su implementación y uso.

Dada esta situación, surge la necesidad de desarrollar nuevas metodologías para la detección precisa y eficaz de defectos. Por consiguiente, el presente proyecto desarrolla un sistema automatizado mediante técnicas de aprendizaje automático y procesamiento de imágenes para identificar defectos en placas de circuito impreso. Este desarrollo se basa en la adaptación de técnicas de detección de objetos, que han demostrado ser efectivas para identificar y categorizar distintos elementos en imágenes generales, aplicándolas específicamente a la detección de defectos en imágenes de PCBs. En particular, se emplea el modelo YOLOv8 para la detección de defectos, incluyendo su clasificación y localización dentro de la placa. Esta técnica permite mejorar el rendimiento de nuevas tareas utilizando modelos pre-entrenados, lo cual acelera el proceso de aprendizaje del modelo y mejora su precisión.

El proyecto se desarrolla utilizando la plataforma *Google Colab*, que aligera considerablemente el proceso de desarrollo, aumentando la eficiencia y facilitando la iteración de algoritmos. Se elige Python como lenguaje de programación y se utiliza la red neuronal convolucional (CNN) como base del algoritmo de detección de defectos. Además, todos los recursos utilizados para el entrenamiento y la validación son recursos abiertos, por lo que garantiza un resultado reproducible y verificable.

El proceso de desarrollo comienza con la investigación de los últimos avances en tecnologías existentes, la selección de una base de datos adecuada y la determinación de los requisitos finales del sistema. A continuación, se lleva a cabo el diseño y la implementación de algoritmos en Python, que incluyen procesos como el tratamiento de imágenes, el entrenamiento del modelo y la generación de resultados. Tras intensivas iteraciones y ajustes, el sistema alcanza un rendimiento excepcional,

logrando una precisión del 99,4 % en la detección de defectos después del entrenamiento del modelo.

Abstract

The printed circuit board industry, also known as PCB, has been significantly affected by the rapid development of technology sectors such as computers, electronics, and telecommunications in recent years. As the demand for PCBs increases and their complexity grows, the industry faces the challenge of improving their quality, as even a small defect on the board can lead to serious failures in the final product performance, resulting in material wastage. For this reason, effective and accurate detection of PCB defects is essential to avoid problems during implementation and use.

Given this situation, there is a need to develop new methodologies for accurate and efficient defect detection. Consequently, this project creates an automated system using machine learning and image processing techniques to identify defects in printed circuit boards. This development is based on adapting object detection techniques, which have proven effective in identifying and categorizing various elements in general images and applying them to defect detection in PCB images. In particular, the YOLOv8 model is used for defect detection, including its classification and localization within the board. This technique allows for the enhancement of new tasks using pre-trained models, accelerating the model's learning process and improving its accuracy.

The project is developed using the Google Colab platform, which significantly streamlines the development process, increasing efficiency and facilitating algorithm iteration. Python is chosen as the programming language, and a convolutional neural network (CNN) serves as the foundation of the defect detection algorithm. Additionally, all resources used for training and validation are open resources, ensuring a reproducible and verifiable result.

The development process begins with researching the latest advances in existing technologies, selecting an appropriate database, and determining the system's final requirements. Next, algorithms are designed and implemented in Python, including processes such as image processing, model training, and result generation. After intensive iterations and adjustments, the system achieves exceptional performance, reaching an accuracy of 99.4 % in defect detection after model training.

Lista de acrónimos

PCB Printed Circuit Board

AOI Automated optical inspection

IC Integrated Circuit

SMT Surface Mounted Technology

ML Machine Learning

TL Transfer Learning

SSD Single Shot multibox Detector

YOLO You Only Look Once

R-CNN Region-based Convolutional Neural Network

CNN Convolutional Neural Networks

RPN Region Proposal Network

RoI Region of Interest

AI Artificial Intelligence

ReLu Rectified Linear unit

tanh Hyperbolic tangent

SPPF Spatial Pyramid Pooling Fast

CBS Convolution layer, Batch normalization layer, Sigmoid function

SOTA State of The Art

TP True Positive

FP False Positive

TN True Negative

FN False Negative

IoU Intersection Over Union

GPU Graphics Processing Unit

CPU Central Processing Unit

TPU Tensor Processing Unit

CUDA Compute Unified Device Architecture

URL Uniform Resource Locator

VOC Visual Object Classes

XML Extensible Markup Language

ODS Objetivos de Desarrollo Sostenible

mAP mean Average Precision

AP Average Precision

Índice

Agradecimientos III

Resumen V

Abstract VII

Lista de acrónimos X

Índice de tablas XV

Índice de figuras XIX

1. Introducción 1

1.1. Motivación del proyecto 1

1.2. Objetivos 3

1.3. Estructura del documento 4

2. Marco tecnológico 7

2.1. Tecnologías de detección de defectos en PCB 7

2.1.1. Inspección con rayos X 7

2.1.2. Inspección óptica automática (AOI) 8

2.2. Aprendizaje Automático 9

2.3. Transfer Learning 11

2.4. Redes Neuronales Convolucionales 12

2.4.1. Capa de convolución 13

2.4.2. Capa de agrupación 14

2.4.3. Capa totalmente conectada 15

2.4.4. Función de activación 16
2.5. Detección de objetos 17
2.5.1. R-CNN 18
2.5.2. Fast R-CNN y Faster R-CNN 19
2.5.3. YOLO 21

3. Especificaciones y restricciones de diseño 25

4. Descripción de la solución propuesta 27
4.1. Configuración del entorno de trabajo 27
4.2. Procesamiento de datos 31
4.2.1. Ampliación de datos (*Data Augmentation*) 32
4.2.2. Segmentación de datos 33
4.2.3. Conversión del formato de anotaciones 34
4.3. Entrenamiento del modelo 36

5. Pruebas y análisis de resultados 41
5.1. Métricas de evaluación del modelo 41
5.1.1. Matriz de confusión 41
5.1.2. Precisión 43
5.1.3. Exhaustividad (*Recall*) 43
5.1.4. Valor F1 (*F1 score*) 44
5.1.5. Intersección sobre la unión (IoU) 44
5.1.6. mAP 45
5.2. Evaluación de modelos 46

5.3. Análisis conjunto 50
5.4. Resultados finales 51

6. Presupuesto 57

7. Impactos del proyecto 59

8. Conclusiones 61

Bibliografía 63

Anexos 67

A. Segmentación de datos 69

B. Convesión de anotaciones VOC a YOLO 71

C. Parámetros de configuración del entrenamiento 75

Índice de tablas

2.1. Diferencia entre Transfer Learning y Machine Learning tradicional 11

4.1. Dataset de PCB original [1] 31

4.2. Dataset de PCB aumentado [1] 33

5.1. Matriz de confusión 42

5.2. Comparativa de distintos entrenamientos de YOLOv8n . . . 48

5.3. Comparativa de distintos entrenamientos de YOLOv8s . . . 49

5.4. Comparativa de distintos entrenamientos de YOLOv8m . . . 49

6.1. Tabla de presupuesto del proyecto de tesis 57

C.1. Argumentos de configuración del entrenamiento [2] 75

Índice de figuras

1.1. Estimación del mercado de PCB [3] 1

1.2. Distintos tipos de defectos de PCB [4] 2

2.1. Defecto detectado mediante inspección con rayos X [5] . . . 8

2.2. Inspección óptica automática 9

2.3. Arquitectura de CNN 13

2.4. Cálculo de la capa de convolución 14

2.5. Agrupación media 15

2.6. Agrupación máxima 15

2.7. Arquitectura de la capa totalmente conectada 16

2.8. Tareas de reconocimiento de imágenes [6] 17

2.9. Estructuras de distintos tipos de detectores 18

2.10. Arquitectura del modelo R-CNN 19

2.11. Arquitectura del modelo Fast R-CNN 20

2.12. Arquitectura del modelo Faster R-CNN 20

2.13. Comparativa de la velocidad de procesamiento de la familia R-CNN 21

2.14. Cronología de las versiones de YOLO [7] 21

2.15. Concepto de diseño de la serie YOLO 22

2.16. Arquitectura inicial de YOLO 23

2.17. Rendimiento de YOLOv8 comparado con otros modelos YOLO [8] 24

2.18. Comparación de métricas para distintas versiones de YOLOv8 [8] 24

4.1. Código de instalación de *Jupyter Notebook* [9] 28

4.2. Captura de Terminal de *Jupyter Notebook* 28

4.3. Cuadro de diálogo de conexión de *Colab* 29

4.4. Información de la GPU local 29

4.5. Comando de instalación de paquetes *Ultralytics* [8] 30

4.6. Comando de instalación de *Pytorch* [10] 30

4.7. Chequeo del la instalación de Pytorch 31

4.8. Estructura de anotaciones en formato VOC 35

4.9. Ejemplo de etiqueta en formato YOLO 36

4.10. Ejemplo del código de entrenamiento con YOLOv8 [8] 37

4.11. Estructura del fichero de configuración de datos 37

4.12. Código de congelación de capas 39

5.1. Matriz de confusión de clasificación multiclase 42

5.2. Ejemplo de IoU para varios *bounding box* 45

5.3. Ejemplo de detección de objetos 45

5.4. Código de entrenamiento sin congelación de capas 46

5.5. Código de entrenamiento con congelación de capas 47

5.6. Código de configuración del modelo YOLOv8s 48

5.7. Resultado final del entrenamiento con YOLOv8s 51

5.8. Curva Preción-Recall 52

5.9. missing hole 53

5.10. mouse bite 53
5.11. open circuit 53
5.12. short 53
5.13. spur 53
5.14. spurious copper 53
5.15. Ejemplo del error de predición caso 1 54
5.16. Imagen original 54
5.17. Ejemplo del error de predición caso 2 54
5.18. Imagen original 54

A.1. Definición del porcentaje de segmentación 69
A.2. Definición de las rutas de trabajo 69
A.3. Asignación de imágenes a cada subconjunto 70
A.4. Escritura de imágenes a cada subconjunto 70

B.1. Definición de la función de convesión 71
B.2. Gestión del directorio de salida de conversión 72
B.3. Recorrido de archivos XML 72
B.4. Escritura de imágenes a cada subconjunto 73
B.5. Definición de clases y rutas de directorios 73

Capítulo 1

Introducción

1.1. Motivación del proyecto

En los últimos años, la industria de placas de circuitos impresos (PCB, por sus siglas en inglés) ha crecido de forma exponencial debido al veloz desarrollo de la electrónica, informática y otras áreas relacionadas. Según los estudios, el mercado de las PCB se valora en aproximadamente 76,12 mil millones de dólares para 2024 y se estima que alcance los 93,87 mil millones de dólares en 2029. Se espera un crecimiento anual del 4,28 % durante el período comprendido entre 2024 y 2029 [3]. Este crecimiento no sólo refleja el aumento en la demanda de dispositivos electrónicos, sino también el uso generalizado de las placas de circuito impreso en diversas aplicaciones tales como teléfonos inteligentes, computadoras, electrónica automotriz y dispositivos médicos.

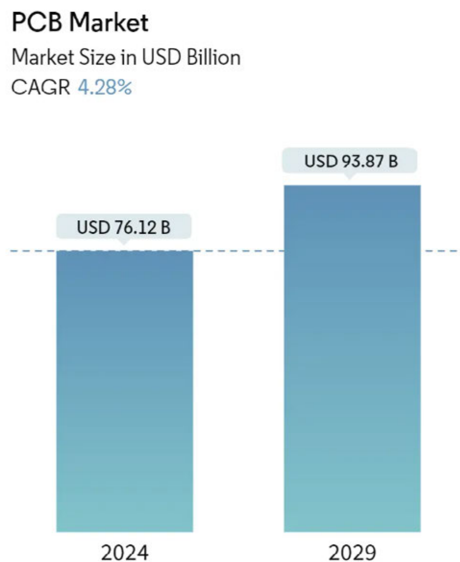

Figura 1.1: Estimación del mercado de PCB [3]

Sin embargo, a medida que aumentan la complejidad y el volumen de producción de las PCB, garantizar su calidad se convierte en un desafío cada vez más significativo. Durante la producción, las PCB pueden presentar diversos defectos, y es fundamental detectar y rectificar estos defectos, dado que una PCB defectuosa puede causar fallos críticos de funcionamiento, pérdida de datos, despredicios de materiales, o incluso riesgos para la seguridad del usuario. Por consiguiente, la eficaz detección e identificación de defectos en las PCB se ha convertido en un tema de gran importancia en la industria.

Hoy en día, los defectos más comunes que pueden encontrarse en las PCB son los siguientes: cortocircuito (*short*), circuito abierto (*open circuit*), falta de agujero de soldadura (*missing hole*), mordeduras de ratón (*mouse bite*), astillas (*spur*) y cobre falso (*spurious copper*). Estos defectos suelen ser pequeños y difíciles de detectar. Tal como se ilustra en la Figura 1.2, se pueden observar los diferentes tipos de defectos.

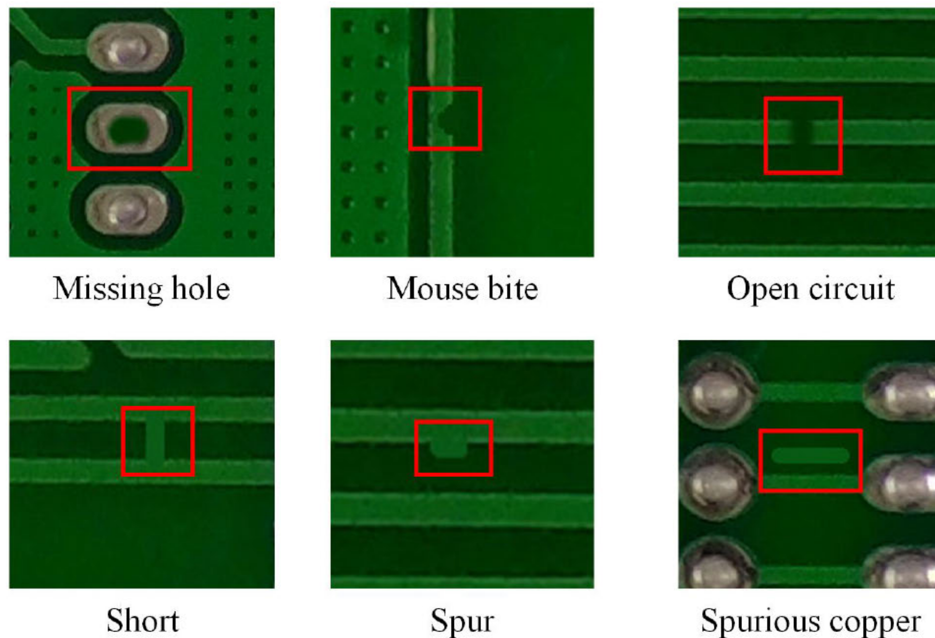

Figura 1.2: Distintos tipos de defectos de PCB [4]

En relación con los métodos de inspección, los métodos tradicionales para la detección de defectos en PCB incluyen principalmente la inspección visual, la inspección con rayos X, y la inspección óptica automática (AOI) [11]. Aunque estos métodos son capaces de identificar defectos en cierta medida, presentan limitaciones notables. En particular, la inspección visual puede ser ineficaz para detectar defectos en áreas de

difícil acceso; la inspección con rayos X, puede ser costosa y requiere equipamentos especializados; y la AOI, aunque es automatizada, puede no ser completamente precisa en la detección de defectos complejos. Estas limitaciones resaltan la necesidad de desarrollar y adoptar métodos de inspección más avanzados y precisos para garantizar la calidad de las PCB.

En los últimos años, la tecnología de aprendizaje automático (*Machine Learning*, ML) ha avanzado significativamente en los campos del procesamiento de imágenes, la detección y el reconocimiento de objetos, proporcionando nuevas soluciones para la detección de defectos en las PCB. En comparación con los métodos tradicionales, el ML puede aprender y extraer características automáticamente para lograr una detección automatizada eficaz y reducir la intervención humana. Mediante el entrenamiento de grandes cantidades de datos, los modelos de ML pueden alcanzar una gran precisión, adaptándose a varios tipos y complejidades de defectos. Además, los modelos de ML pueden optimizarse y actualizarse continuamente para adaptarse a los rápidos cambios de los requisitos de producción y los avances tecnológicos.

Aunque los algoritmos de ML han logrado resultados significativos en el campo de la detección de defectos de PCB, todavía existen algunos desafíos técnicos que deben abordarse. En primer lugar, el equilibrio entre tiempo real y precisión es una dificultad vigente. En segundo lugar, la detección de defectos pequeños y diminutos plantea mayores exigencias en cuanto al rendimiento de los algoritmos. Estos defectos suelen ser difíciles de identificar con precisión mediante algoritmos tradicionales y requieren algoritmos que puedan interpretar mejor los detalles de la imagen. Por último, la mejora de la capacidad de generalización de los modelos también es objetivo de futuras investigaciones.

Por consiguiente, el presente Proyecto Fin de Grado se enfocará en desarrollar un sistema basado en técnicas de aprendizaje automático que aborde estos desafíos anteriormente mencionados, mejorando la detección y clasificación de defectos en PCB de manera más eficiente y precisa, crucial para mantener la calidad en el mercado actual de rápido crecimiento.

1.2. Objetivos

El objetivo principal del presente Proyecto Fin de Grado consiste en desarrollar un sistema de detección de defectos de las PCB mediante técnicas de ML, más concretamente, se aplica la técnica llamada *Transfer*

Learning (TL). Partiendo de esta base, se detallen los objetivos de la siguiente forma:

- Abordar las necesidades específicas de la detección de defectos en PCB para seleccionar el modelo más apropiado para los distintos escenarios de uso y requisitos de comportamiento.
- Desarrollar un modelo de detección de objetos basado en la utilización de una arquitectura convolucional de base preentrenada y sustitución de las capas especializadas para adaptar a la aplicación específica del proyecto.
- Preprocesar el conjunto de datos para que se acomodarse al modelo preentrenado, lo que consiste en formatear las anotaciones del conjunto de imágenes, clasificar dicho conjunto de imágenes en tres conjuntos distintos para tareas de entrenamiento, validación y prueba, respectivamente.

1.3. Estructura del documento

La documentación del presente Proyecto Fin de Grado se distribuye en varios capítulos distintos, como se describe a continuación:

- **Capítulo 1**. Se describe el estado actual de la detección de defectos de las PCB en el mercado y algunas de las soluciones existentes, se incluye también los objetivos de investigación de este proyecto y una breve descripción de la estructura del presente documento.
- **Capítulo 2**. Se ofrece una descripción detallada de los aspectos técnicos, las herramientas y los métodos que forman parte del proyecto con el fin de ayudar a comprender mejor los fundamentos tecnológicos en los que se basa el proyecto.
- **Capítulo 3**. Se explica los requisitos, criterios y parámetros de diseño, así como las limitaciones y restricciones que deben seguirse durante el proceso de diseño.
- **Capítulo 4**. Se detalle el proceso completo utilizado para abordar los objetivos establecidos anteriormente. Se presenta una exposición exhaustiva de cada fase del desarrollo o implementación del proyecto, discutiendo las decisiones clave tomadas durante el proceso y cómo estas contribuyen a alcanzar los objetivos específicos del proyecto.

- **Capítulo 5**. Se describe sistemáticamente todas las pruebas a las que se sometió el diseño o solución propuesta para verificar su correcto funcionamiento. Se presenta una descripción detallada de los experimentos realizados, asegurando que sea posible reproducirlos. Los resultados obtenidos de estos experimentos se presentan de manera sintetizada, utilizando tablas o gráficos concisos para destacar los hallazgos más relevantes.

- **Capítulo 6**. Se aborda el presupuesto utilizado en el presente proyecto. Se detallan los recursos y materiales asignados para llevar a cabo el proyecto, incluyendo costos asociados con equipos, software, personal y otros gastos relevantes para maximizar la eficiencia y efectividad del proyecto.

- **Capítulo 7**. Se evalúa el impacto tecnológico, ambiental, económico e industrial del proyecto. Se examinan las implicaciones en la industria de detección de defectos de PCB, así como posibles efectos en la sostenibilidad, la eficiencia de producción.

- **Capítulo 8**. Consiste en una síntesis y conclusión del estudio realizado. Se recapitulan los principales hallazgos, conclusiones y contribuciones del proyecto en relación con los objetivos iniciales. Además, se discuten posibles direcciones futuras de investigación o mejoras adicionales que podrían explorarse a partir de los resultados obtenidos.

Capítulo 2

Marco tecnológico

En este apartado se presenta una exploración detallada de los aspectos técnicos fundamentales, herramientas y metodologías que constituyen la base del proyecto con el fin de conseguir una comprensión clara de la infraestructura tecnológica necesaria para llevar a cabo la solución propuesta en el contexto del estudio.

2.1. Tecnologías de detección de defectos en PCB

Dado que el enfoque principal de este proyecto se centra en la detección de defectos en PCB, se comienza presentando algunas de las principales tecnologías utilizadas en el mercado actual.

2.1.1. Inspección con rayos X

Con el avance de la industria de fabricación electrónica, los requisitos para las PCB se han vuelto cada vez más exigentes, impulsando la miniaturización de los productos electrónicos. Sin embargo, los componentes de montaje tradicional a través de agujeros, ya no son suficientes para cumplir con estas exigencias.

En respuesta a esta necesidad, ha surgido la tecnología de montaje superficial (SMT), la cual implica el montaje directo de componentes electrónicos sobre la superficie de las PCB, en lugar de insertarlos a través de agujeros. Actualmente, las PCB que emplean SMT están más densamente pobladas, con componentes más pequeños y conexiones ocultas que no pueden ser observadas a través de ciertos encapsulados. Estas características representan desafíos significativos para la detección de defectos. Para abordar estos problemas, se ha desarrollado la tecnología de inspección con rayos X, con el objetivo de mejorar la exhaustividad y

precisión de las inspecciones, proporcionando una visualización clara de las conexiones y las estructuras internas, y detectando defectos pequeños mediante la característica de absorción de rayos X de los componenetes. A continuación, se muestra un defecto detectado mediante la inspección con rayos X.

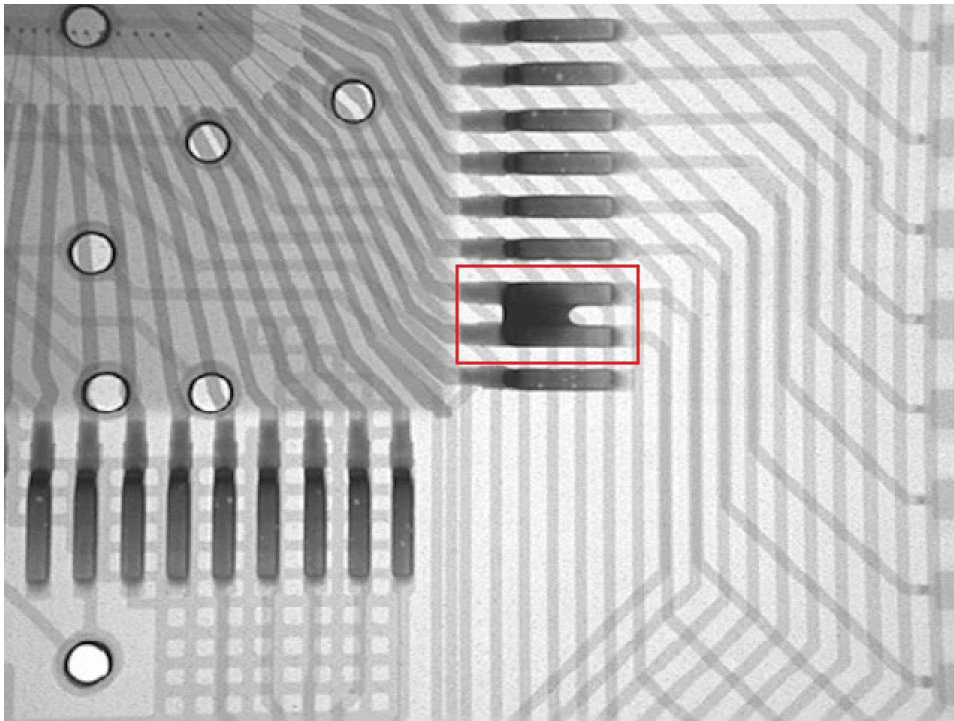

Figura 2.1: Defecto detectado mediante inspección con rayos X [5]

Como inconvenientes de la inspección con rayos X cabe citar su elevado coste, tanto en términos de equipamentos como de formación del personal y consumo de tiempo, la necesidad del personal especialmente formado para manejar la máquina y la complejidad a la que puede enfrentarse al tratar grandes cantidades de datos.

2.1.2. Inspección óptica automática (AOI)

La inspección óptica automatizada (AOI) es una tecnología avanzada que detecta automáticamente defectos en la superficie de las PCB mediante el uso de cámaras de alta resolución y software especializado de procesamiento de imágenes. Este sistema identifica problemas comunes como fallos de soldadura, componentes desalineados, cortocircuitos y

roturas en los circuitos. Funciona capturando imágenes de la superficie de la PCB y comparándolas con modelos ideales utilizando algoritmos predefinidos y patrones de referencia.

Figura 2.2: Inspección óptica automática

La tecnología AOI se destaca por su rapidez y por proporcionar resultados de inspección consistentes, lo cual mejora significativamente la productividad y la calidad del producto final. Sin embargo, en el caso de PCB complejas y con múltiples capas, su capacidad para identificar defectos puede verse limitada. Esto se debe a que el sistema de visión rpuede tener dificultades paa capturar y analizar con precisión todas las capas de la imagen, lo cual representa una consideración importante en su aplicación práctica.

2.2. Aprendizaje Automático

El aprendizaje automático, también conocido como *Machine Learning* (ML), es una rama de la Inteligencia Artificial (AI) en la que se utiliza algoritmos para simular comportamientos de aprendizaje humano con el fin de aprender y mejorar automáticamente a partir de los datos introducidos. Su principal objetivo es permitir a las máquinas descubrir patrones mediante el análisis de grandes cantidades de datos sin instrucciones explícitas de programación, logrando de esta forma las tareas de predicción y clasificación. Los algoritmos de ML mejora continuamente la precisión y el rendimiento de las decisiones al identificar e integrar los errores y

la retroalimentación de los datos analizados, lo que permite a los sistemas informáticos adaptar y optimizar su comportamiento de forma automática.

En términos generales, el aprendizaje automático se divide en las siguientes categorías:

- **Aprendizaje supervisado**. Es una de las categorías más comunes del ML. Se define como el proceso de entrenamiento de algoritmos cuyos datos de entrenamiento son etiquetados, es decir, cada ejemplo de datos está asociado a una etiqueta o una categoría conocida. Estas etiquetas sirven como la respuesta deseada o la salida esperada para el modelo del aprendizaje supervisado durante el proceso de entrenamiento. Cuando se introducen datos en el modelo, éste ajusta sus pesos de entrenamiento evitando el sobreajuste o infraajuste del modelo. Los métodos más comunes del aprendizaje supervisado son las redes neuronales, la regresión lineal, la logística y los bosques aleatorios.

- **Aprendizaje no supervisado**. A diferencia del aprendizaje supervisado, el modelo del aprendizaje no supervisado descubre patrones y estructuras de datos sin necesidad de tener etiquetas predefinidas, dicho de otra forma, sin necesidad de la intervención humana. Las tareas más conocidas de esta metodología son la agrupación (*clustering*), la reducción de la dimensionalidad (*dimensionality reduction*) y el aprendizaje de reglas de asociación (*association rule learning*).

- **Aprendizaje semisupervisado**. Es un término medio que combina elementos del aprendizaje supervisado y no supervisado, en el que los modelos se entrenan con una mezcla de datos etiquetados y no etiquetados. En general, la cantidad de datos etiquetados suele ser menor que la de datos sin etiquetar, siendo los datos etiquetados los que guian a los datos sin etiquetar para obtener un resultado con mayor precisión. Puede ser de especial utilidad para casos en el que resulta difícil obtener suficientes etiquetas para el conjunto completo de datos.

- **Aprendizaje por refuerzo**. Es un método de aprendizaje basado en la interacción de un organismo inteligente (*agent*) con su entorno, es decir, el *agent* a través de la experimentación continua, observa el estado del entorno, selecciona acciones y tras la selección, recibe retroalimentación positivas o negativas según el entorno. Comparado con los métodos anteriores, el aprendizaje por refuerzo no requiere

datos masivos etiquetados para poder entrenar, se adapta mejor a entornos complejos optimizando la toma de desiciones.

2.3. Transfer Learning

Entrenar un nuevo modelo de ML es un proceso largo e intensivo que requiere muchos datos, potencia de cálculo y múltiples iteraciones. Por lo consiguiente, el aprendizaje por transferencia (*Transfer Learning, TL*) resulta ser buena solución a este problema. El TL es una técnica de ML cuyo concepto esencial consiste en aplicar conocimiento de un modelo preentrenado através de técnicas como ajuste fino (*fine tuning*), congelación de capas (*layer freezing*) para adaptar a tareas nuevas con el fin de acelear el proceso de entrenamiento y mejorar el rendimiento y la eficiencia a la hora de entrenar una tarea nueva. Es particularmente útil en tareas específicas donde los datos disponibles son escasos, así como en aplicaciones que demandan altos recursos computacionales. Además, permite aprovechar el conocimiento previamente adquirido en tareas similares y más generales.

A continuación, se explican, en forma de tabla, las principales diferencias entre el TL y el ML tradicional:

	Transfer Learning	**Machine Learning tradicional**
Distribución de datos	No es necesario que los datos de entrenamiento y de prueba se distribuyen de la misma proporción	Los datos de entrenamiento y de prueba se distribuyen en la misma proporción
Datos etiquetados	No exige datos suficientes etiquetados	Exige datos suficientes etiquetados
Modelización	Reutilización de modelos preentrenados	Cada tarea se modeliza por separado

Tabla 2.1: Diferencia entre Transfer Learning y Machine Learning tradicional

2.4. Redes Neuronales Convolucionales

Las redes neuronales convolucionales (CNN, también conocido por su nombre en inglés, Convolutional Neuronal Networks) son un conjunto de algoritmos del aprendizaje automático que se utiliza principalmente para procesar imágenes y vídeos. Además, son adecuadas para el análisis audios, señales biomédicas y otras señales que presentan correlación temporal o espacial, donde los valores de una muestra presenta similitud con las muestras cercanas en el espacio o tiempo. Estas redes se basa fundamentalmente en el sistema visual biológico del ser humano, imitando así el procesamiento visual humano.

La historia de las CNN se remonta a los años ochenta del siglo XX, cuando el informático japonés *Kunihiko Fukushima* introdujo el modelo *Neocognitron* basado en la teoría de las redes neuronales visuales propuesta por *David H. Hubel* y *Torsten N. Wiesel*. El modelo, que puede identificarse como precursor de las CNN, se utiliza para reconocer caracteres manuscritos en japonés y fue el primero en proponer el concepto de capas de extracción de caracteres (*S-cells*) y capas de agrupación (*C-cells*).[12]

Partiendo de esta base, el informático francés *Yann LeCun* y su equipo perfeccionaron las CNN y presentaron LeNet-5 en el año 1998. Este modelo fue el primer modelo de CNN completo y exitoso, en el que utiliza capas de convolución, capas de agrupación y capas de conexión completa en la estructura de las CNN. Se utiliza principalmente para reconocimientos de caracteres manuscritos, con una alta tasa de presición.[13]

Como se menciona anteriormente, las CNN se compone fundamentalmente en tres capas: capa de convolución (*convolutional layer*), capa de agrupación (*pooling layer*) y capa de conexión completa (*fully conected layer*), como se puede apreciar en la siguiente imagen.

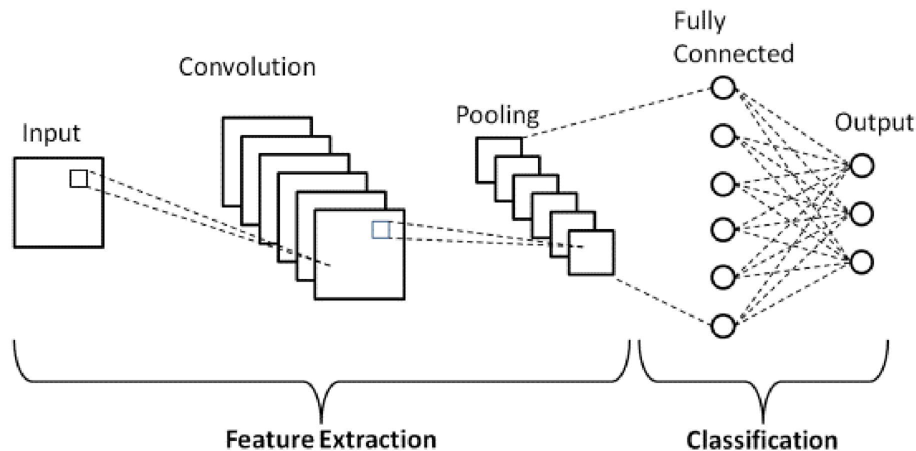

Figura 2.3: Arquitectura de CNN

2.4.1. Capa de convolución

La capa de convolución (*convolutional layer*) es el núcleo de las CNN. En general, se compone de tres partes principales, el dato de entrada, el filtro o *kernel*, y el mapa de activación (*activation map*), que también se conoce como característica convolucionada (*feature convoluted*) o mapa de características (*feature map*).

Si la capa de convolución es el punto vital de las CNN, entonces el *kernel* es la clave de esta capa. El *kernel*, también conocido como matriz de convolución, suele ser una matriz pequeña de dimensión 3x3, aunque también hay casos que utilizan *kernel* de 5x5, cuya función principal es realizar operaciones de convolución con la imagen de entrada para llevar a cabo la extracción de características tales como bordes, texturas, formas y otros patrones relevantes.

El elemento de cada píxel de la imagen de entrada se multiplica por el elemento correspondiente, es decir, el elemento en la misma posición del *kernel* y, a continuación, se suman todos los valores para obtener la mapa de activación. La Figura 2.4 presenta una demostración en la que explica claramente el funcionamiento del método de cálculo.

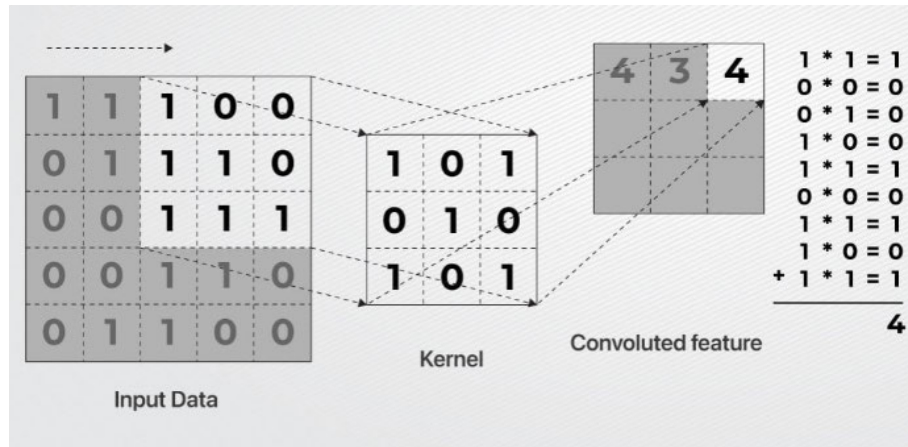

Figura 2.4: Cálculo de la capa de convolución

2.4.2. Capa de agrupación

Por lo general, las capas convolucionales se utilizan junto con las capas de agrupación, también conocido como (*pooling layer*). La capa de agrupación pretende reducir la muestra de las características que quedan después de la convolución, lo que se conoce habitualmente como *downsampling*. Dicho en otras palabras, el *downsampling* disminuye el tamaño de las características extraídas, manteniendo las características originales con el fin de reducir la cantidad de cálculos en el proceso de retropropagación y propagación hacia adelante, y así acelerar el entrenamiento de los datos. Gracias a esta capa de agrupación, reduce la complejidad del modelo y, por lo tanto, consigue evitar la sobreexplotación del modelo (*overfitting*).

La capa de agrupación se divide en dos tipos principales: agrupación media (*Average Pooling*) y agrupación máxima (*Max Pooling*).

- **Agrupación media**. Se calcula el valor medio de todos los píxeles contenidos en la ventana de pooling y se toma dicho valor medio como el resultado del pooling de dicha ventana. Con la agrupación media se consigue un efecto suavizado del ruido tras realizar el cálculo del valor medio. Como podemos apreciar en la Figura 2.5, se utiliza una ventana de pooling de dimensión 2x2. Tomando el subbloque rojo como el ejemplo, hay cuatro números bajo dicha ventana que son 12, 20, 8 y 12, obteniendo un valor medio de estos cuatro números con el valor de 13, y este número se considera como el valor final del pooling de la ventana roja.

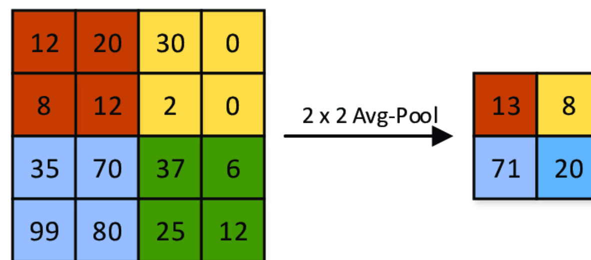

Figura 2.5: Agrupación media

- **Agrupación máxima**. Se elige el carácter más dominante (el valor máximo) de cada ventana de pooling como el resultado de agupación máxima. Al mantener los rasgos más dominantes en cada región, la red puede ignorar el ruido y centrarse en los rasgos más importantes. Además, gracias a la agrupación máxima, la red es menos sensible a pequeñas traslaciones o desplazamientos de la entrada. En el ejemplo de la Figura 2.6, igual que la agupación media, se utiliza una ventana de pooling de dimensión 2x2. En el subbloque rojo, el número máximo en dicho bloque es 20, por lo que 20 se considera como el resultado del pooling de dicha ventana.

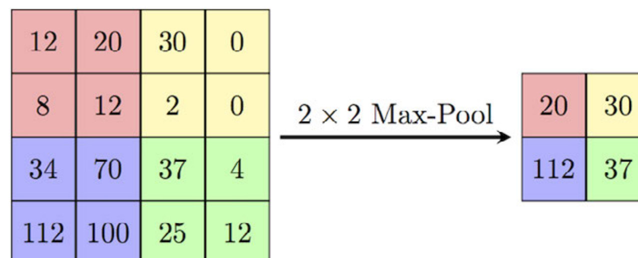

Figura 2.6: Agrupación máxima

2.4.3. Capa totalmente conectada

En las CNN, las capas convolucionales se encargan del proceso de extracción de características mediante la aplicación de filtros y la reducción de dimensionalidad mediante capas de pooling, generando así mapas de características que capturan detalles importantes de la imagen.

Posteriormente, las capas totalmente conectadas (*fully connected*, FC), que suelen situarse al final de la estructura de las CNN, desempeñan un papel vital al recibir estas características extraídas y utilizarlas para tareas

específicas como clasificación o detección de objetos. En el contexto de la detección de objetos, las CNN suelen estructurarse con dos conjuntos de capas FC: uno dedicado a la clasificación de cada objeto identificado y otro para localizar su posición precisa en la imagen.

Cada neurona de esta capa recibe entradas de todas las neuronas de la capa anterior y produce una salida aplicando un conjunto de pesos. Debido a su naturaleza totalmente conectada, dicha capa suele contener el mayor número de parámetros.

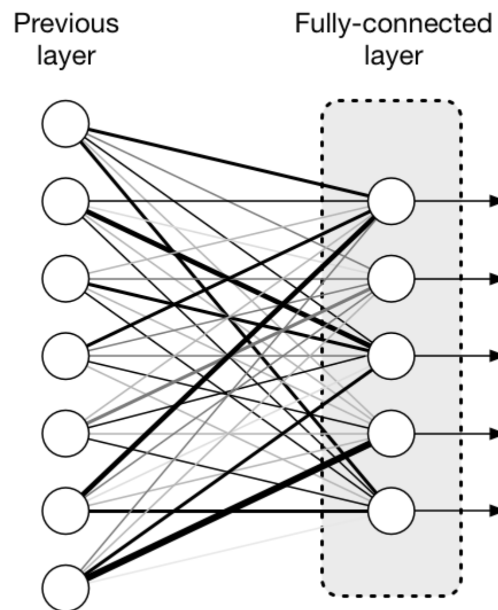

Figura 2.7: Arquitectura de la capa totalmente conectada

2.4.4. Función de activación

La salida de la estructura neuronal es la suma ponderada de todas las entradas, lo que hace que la red neuronal sea un modelo lineal. Si la salida de cada neurona (es decir, el nodo de la red neuronal) se hace pasar por una función no lineal, entonces todo el modelo de la red neuronal deja de ser lineal, y esta función no lineal es la función de activación.

Las funciones de activación suelen aplicarse a la salida de las capas de la CNN, tomando la suma ponderada de las entradas, produce una salida que pasa a la capa siguiente. Las funciones de activación más comunes son: función ReLU (*Rectified Linear unit*), función *Sigmoid* y función tangente hiperbólica (*tanh*). Además, cabe añadir que las funciones de activación en

la última capa dependen de la tarea a desarrollar. Por ejemplo, para una tarea de clasificación binaria, es común utilizar la función *Sigmoid* en la última capa [14].

2.5. Detección de objetos

La detección de objectos, siendo una de las tareas de reconocimiento de imágenes más importantes en el campo del aprendizaje automático, consiste en identificar la presencia y la ubicación de múltiples objetos dentro de una imagen o un video. A diferencia de la clasificación de imágenes, donde se predice una sola etiqueta para toda la imagen, la detección de objetos se centra en identificar y localizar cada objeto individualmente, marcando su posición específica con un cuadro delimitador (*bounding box*) y clasificándolo en una de varias categorías predefinidas.

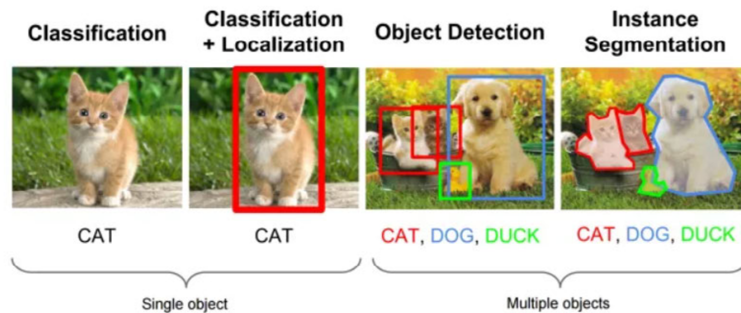

Figura 2.8: Tareas de reconocimiento de imágenes [6]

Sus aplicaciones abarcan desde sistemas de vigilancia, vehículos autónomos hasta herramientas médicas, análisis de imágenes satelitales, control de defectos y anomalías de fabricación. Mejora la seguridad pública, facilita diagnósticos médicos precisos, y apoya la automatización en la producción. En conjunto, la detección de objetos impulsa avances tecnológicos significativos que transforman múltiples sectores con su capacidad para interpretar y procesar información visual de manera efectiva.

Con el avance del aprendizaje automático, los algoritmos para la detección de objetos han ido generalizando gradualmente. En este campo, destacan dos estructuras principales: los detecdores de dos etapas y los detectores de una sola etapa.

Los detecdores de dos etapas, como la familia R-CNN, se caracterizan por dividir claramente la localización de los objetos y la clasificación en dos pasos distintos. Primero, generan regiones propuestas que podrían contener objetos, posteriormente, aplican una red neuronal para clasificar estas regiones.

Por otro lado, los detectores de una sola etapa, como la serie YOLO (*You Only Look Once*), realizan la detección y clasificación de objetos en una sola operación. Estos algoritmos son conocidos por su velocidad y simplicidad, ya que evitan la generación de regiones propuestas y procesan toda la imagen directamente con una red neuronal convolucional, prediciendo simultáneamente los cuadros delimitadores y las probabilidades de clase.

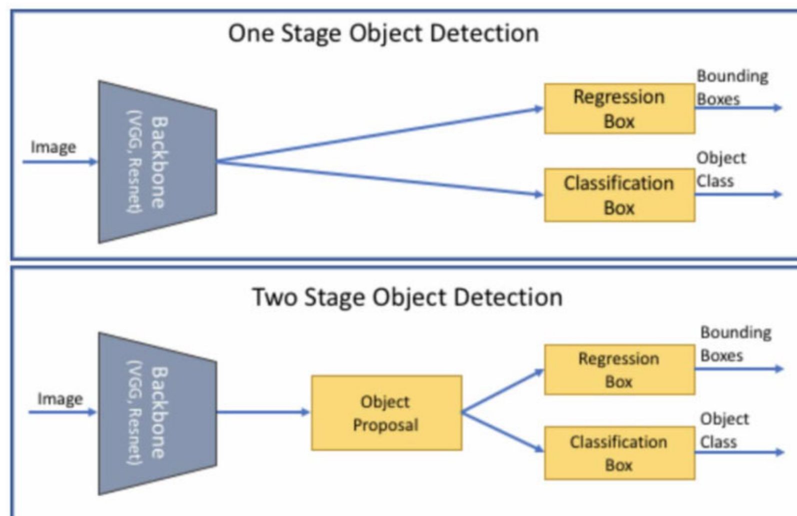

Figura 2.9: Estructuras de distintos tipos de detectores

Ambas estructuras tienen sus ventajas y desventajas en términos de precisión y eficiencia computacional. La elección entre ellos depende de las necesidades específicas de la aplicación de detección de objetos.

2.5.1. R-CNN

La R-CNN (Redes Neuronales Convolucionales Basadas en Regiones) es un modelo de detección de objetos en dos etapas en el que se utiliza un algoritmo tradicional para generar alrededor de 2.000 propuestas de regiones (*region proposals*), luego extrae características de cada región candidata usando una CNN preentrenada. Después, se ajustan las regiones

candidatas mediante una técnica de deformación (*warping*) para normalizar su tamaño, lo que permite introducirlas en una máquina de vectores de soporte (SVM) para la clasificación y determinar si la región contiene un objeto. Además, ajusta los cuadros delimitadores utilizando regresión a través de una red neuronal. R-CNN emplea el aprendizaje por transferencia y ajuste fino para mejorar la precisión de las propuestas de regiones. Sin embargo, presentan problemas como la lentitud en el entrenamiento y la inferencia, y la independencia entre la propuesta de regiones y la clasificación. Estos problemas llevaron al desarrollo de versiones mejoradas como Fast R-CNN y Faster R-CNN.

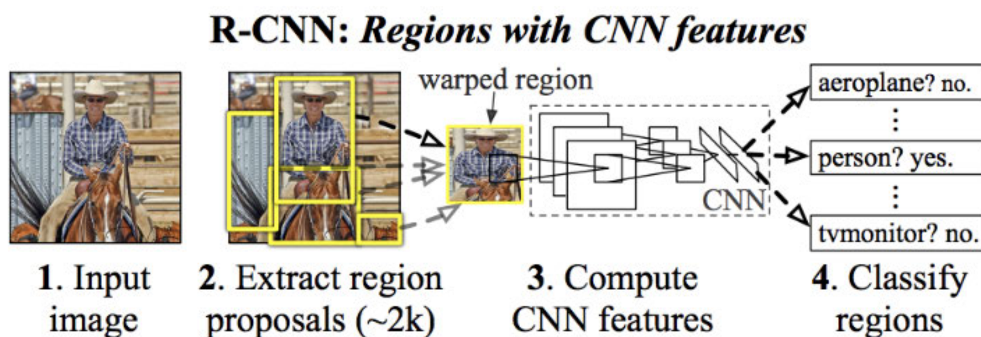

Figura 2.10: Arquitectura del modelo R-CNN

2.5.2. Fast R-CNN y Faster R-CNN

La Fast R-CNN representa una mejora significativa respecto a la R-CNN al integrar la propuesta de regiones, la extracción de características, la clasificación y la regresión de cuadros delimitadores en una única red neuronal. Además, introduce una capa de agrupación de regiones de interés (*RoI Pooling*). A través de esta capa, las regiones candidatas (RoI) de diferentes tamaños se asignan a un mapa de características de tamaño fijo. Esta estrategia no solo acelera el procesamiento de la red, sino que también garantiza una adaptación eficaz a entradas de diversas dimensiones.

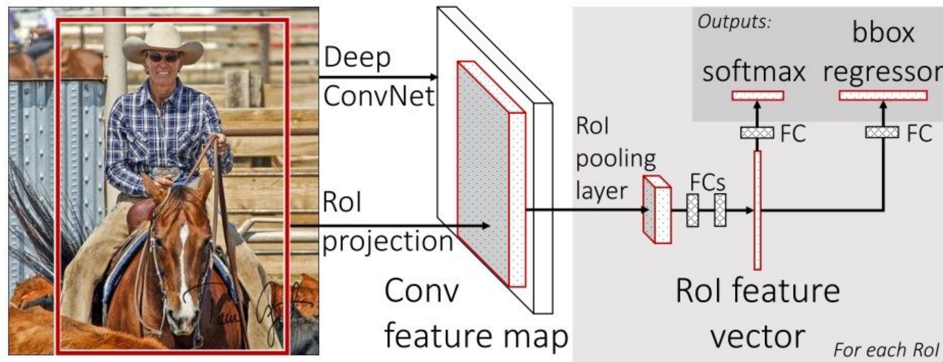

Figura 2.11: Arquitectura del modelo Fast R-CNN

Faster R-CNN mejora aún más a Fast R-CNN mediante dos innovaciones claves. Por un lado, introduce la red de propuesta de región (RPN), que reemplaza el método tradicional de propuesta de regiones. Esto permite al modelo generar propuestas de regiones de manera eficiente y adaptable durante el entrenamiento. Por otro lado, Faster R-CNN logra un entrenamiento de extremo a extremo, es decir, un entrenamiento que integra todas las tareas y pasos relacionados en un marco de aprendizaje unificado para lograr un método de entrenamiento más eficiente, simplificado y consistente. Esta integración mejora significativamente la velocidad y la precisión de la detección, habilitando al modelo para manejar eficazmente tareas de detección complejas.

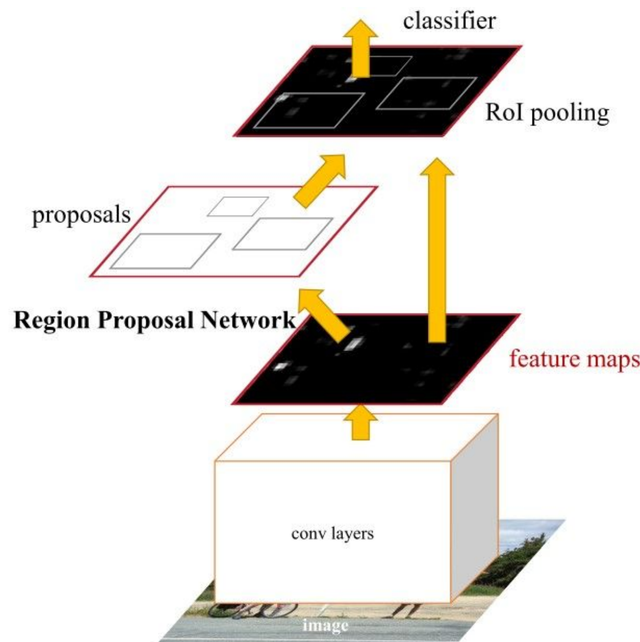

Figura 2.12: Arquitectura del modelo Faster R-CNN

A continuación, se presenta la comparativa de la velocidad de procesamiento de los tres modelos de la familia R-CNN con el fin de ofrecer una perspectiva clara de las diferencias entre ellos. Se puede apreciar una marcada diferencia en el rendimiento al avanzar desde R-CNN hasta Fast R-CNN, y aunque la mejora al pasar de Fast R-CNN a Faster R-CNN no es tan significativa como anteriormente, sigue siendo notable.

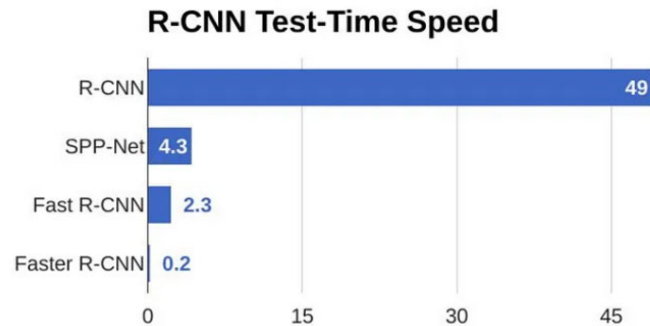

Figura 2.13: Comparativa de la velocidad de procesamiento de la familia R-CNN

2.5.3. YOLO

Entre los diversos algoritmos de detección de objetos, la serie YOLO destaca por su notable equilibrio entre velocidad y precisión a la hora de reconocer objetos en imágenes de forma rápida y fiable. Desde su creación, la familia YOLO ha pasado por varias iteraciones, en cada una de las cuales se han abordado las limitaciones y se ha mejorado el rendimiento de las versiones anteriores. A continuación, se presenta la cronología de las distintas versiones a lo largo de la historia de YOLO.

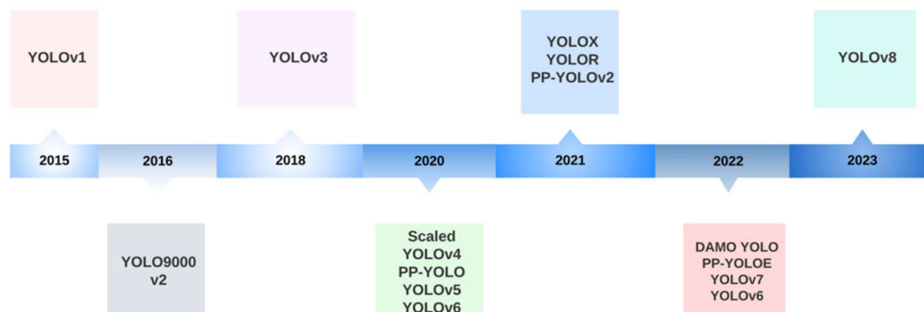

Figura 2.14: Cronología de las versiones de YOLO [7]

La idea principal de los algoritmos de la serie YOLO consiste en dividir la imagen original en cuadrículas de tamaño S x S que no se superponen entre sí, donde cada cuadrícula es responsable de detectar el objeto dentro de esa cuadrícula. En cada cuadrícula, se colocan B *anchors box*, que son un conjunto de cuadros predefinidos cuyas anchuras y alturas se seleccionan para coincidir con las anchuras y alturas de los objetos que desea ser detectados. Cada *anchor box* se utiliza para predecir un cuadro delimitador del objetivo y se incluyen informaciones como puntuación de confianza (*confidence score*) y las coordenadas del cuadro delimitador (*bounding box coordinates*). En las diferentes versiones de YOLO, el valor de B puede variar.

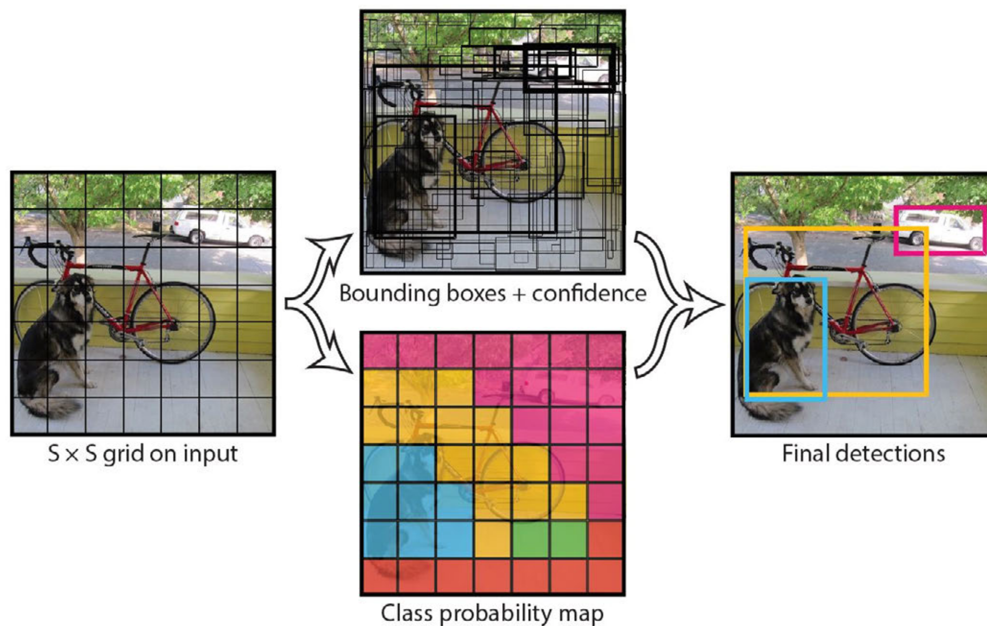

Figura 2.15: Concepto de diseño de la serie YOLO

Como la mayoría de los modelos basados en CNN, YOLO emplea una red convolucional para la extracción de características, seguida por capas completamente conectadas para obtener predicciones. La arquitectura de la red está inspirada en el modelo GooLeNet, que incluye 24 capas convolucionales y 2 capas completamente conectadas, como se ilustra en la Figura 2.16. El resultado final consiste en un vector de tamaño $S * S * (B * 5 + C)$, donde S es el tamaño de la cuadrícula, B es el número de cuadros delimitadores previstos por cuadrícula, 5 son los parámetros de cada cuadro delimitador (x, y, w, h, confianza) y C es el número de categorías.

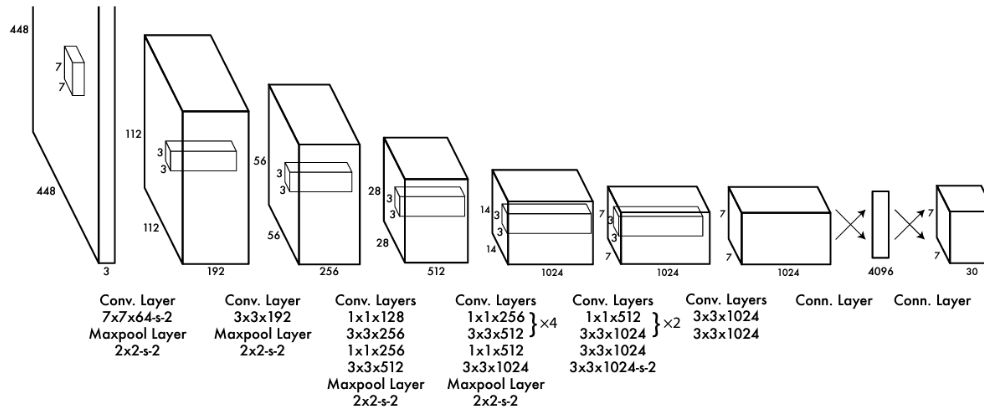

Figura 2.16: Arquitectura inicial de YOLO

Gracias a la estructura de YOLO, se logran velocidades rápidas de detección a la hora de procesar imágenes, dado que toda la imagen es analizada por una sola red. Esto elimina los costos adicionales de tiempo asociados al proceso de generación de regiones candidatas. Sin embargo, presenta algunas limitaciones tales como la precisión en la detección de objetos pequeños, por lo que resalta la necesidad de nuevas optimizaciones del modelo que aborden estos desafíos.

Tras numerosas iteraciones de optimización en diferentes versiones, surge el modelo YOLOv8, que es la última versión de la serie YOLO, desarrollada por *Ultralytics*, hereda la alta eficacia de la serie YOLO y, al mismo tiempo, introduce una serie de mejoras en la estructura del modelo y en la metodología de entrenamiento para mejorar aún más el rendimiento. Las innovaciones más destacadas son: una nueva estructura de extracción de características, también conocido como *backbone*, una nueva cabecera de detección *Ancher-Free* y una nueva función de pérdida que puede ejecutarse en una amplia gama de plataformas de hardware, desde CPU hasta GPU.

Como podemos apreciar en la Figura 2.17, gracias a estas innovaciones, el rendimiento y la velocidad de procesamiento de YOLOv8 es claramente superior al de las versiones anteriores de YOLO.

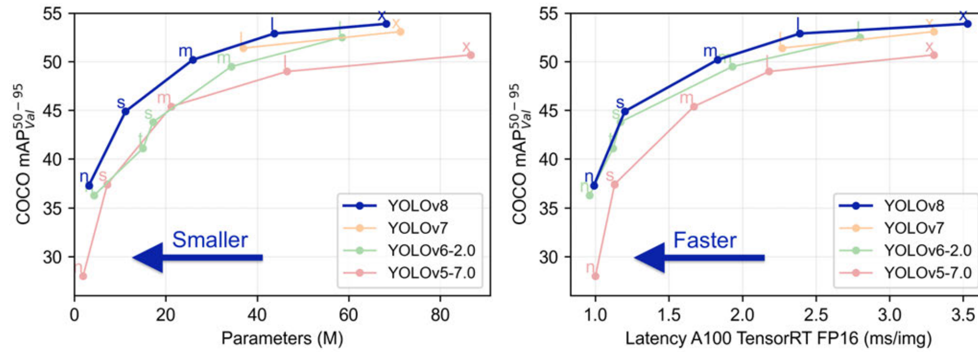

Figura 2.17: Rendimiento de YOLOv8 comparado con otros modelos YOLO [8]

Dependiendo de las distintas situaciones y del entorno operativo, YOLOv8 se ha lanzado varios modelos según su tamaño. La principal diferencia radica en el número de parámetros utilizados en cada capa, cuanto mayor número de parámetros posee el modelo, mayor será la precisión, pero, de manera análoga, más largo será el tiempo de ejecución. En la Figura 2.18, podemos observar las distintas métricas de evaluación según las versiones de YOLOv8.

Model	size (pixels)	mAP^{val} 50-95	Speed CPU ONNX (ms)	Speed A100 TensorRT (ms)	params (M)	FLOPs (B)
YOLOv8n	640	37.3	80.4	0.99	3.2	8.7
YOLOv8s	640	44.9	128.4	1.20	11.2	28.6
YOLOv8m	640	50.2	234.7	1.83	25.9	78.9
YOLOv8l	640	52.9	375.2	2.39	43.7	165.2
YOLOv8x	640	53.9	479.1	3.53	68.2	257.8

Figura 2.18: Comparación de métricas para distintas versiones de YOLOv8 [8]

Capítulo 3

Especificaciones y restricciones de diseño

El presente PFG se desarrolla de acuerdo con los siguientes requisitos funcionales:

- Se conoctará al GPU local a través de la herramienta *Jupyter Notebbok* para acelear el procesamiento de datos y el entrenamiento del modelo.
- Se empleará Python como lenguaje de programación para el desarrollo e implementación del sistema de detección.
- Se utilizará base de datos públicos para el entrenamiento, la validación y la prueba del modelo.
- La base de datos utilizada para el entrenamiento y la validación del modelo debe contener un número suficiente de detectos para un resultado mejor de detección.
- La base de datos debe de cubrir la todalidad de los defectos que se desean ser detectados, tales como, cortociruito (*short*), circuito abierto (*open circuit*), falta de agujero de soldadura (*missing hole*), mordiscos de ratón (*mouse bite*), astilla (*spur*) y cobre falso (*spurious copper*).
- El sistema utilizará redes neuronales convolucionales (CNN) como base para el desarrollo de los algoritmos.
- El sistema informará sobre el tipo de defectos detectados en cada imagen relativa a una placa de circuitos impresos dada como entrada. Además, se devolve a su salida esa misma imagen, sobre la que marca las posiciones de los defectos detectados en dicha imagen.

Por otro lado, cabe destacar también los requisitos técnicos:

- Windows 11
- Pytorch versión 2.3.1
- CUDA versión 12.1

Capítulo 4

Descripción de la solución propuesta

De acuerdo con los objetivos expresados anteriormente, se procede a la fase de desarrollo de la propuesta con el fin de desarrollar un sistema de detección de defectos de las PCBs empleando la técnica de aprendizaje por transferencia. A continuación se detalla las fases que han sido identificadas para llevar a cabo este proyecto, empezando por la configuración del entorno de trabajo, seguido por el preprocesamiento de datos, el desarrollo del modelo, el entrenamiento y validación, la evaluación y optimización, y finalmente la implementación y pruebas en el mundo real. Estas etapas abarcan desde la instalación de herramientas necesarias y la recolección y preparación de datos, hasta la selección y ajuste de modelos preentrenados, el entrenamiento y validación del modelo, su evaluación y optimización, y finalmente su integración y prueba en un entorno de producción real.

4.1. Configuración del entorno de trabajo

Cuando se trata de la configuración del entorno de trabajo, lo primero que debe mencionarse es la plataforma *Google Colaboratory*, también conocido como *Google Colab*. Se trata de un entorno interactivo basado en la nube desarrollado por *Google*. Proporciona recursos informáticos gratuitos tales como CPU, GPU y TPU, que permiten a los usuarios elaborar y ejecutar códigos en un navegador sin necesidad de configuración ni instalación, a través de la cual los desarrolladores pueden ejecutar fácilmente en ella *frameworks* de ML como *Tensorflow*, *Pytorch*, etc.

Aunque *Google Colab* ofrece ciertos recursos gratuitos, la cantidad de recursos es limitada y todos los tiempos de ejecución de *Colab* se restablecen después de un período de tiempo. Sin embargo, los suscriptores de *Colab Pro* tendrá un tiempo limitado mayor que los usuarios no suscriptores.

Dada la dificultad de algoritmos y el tamaño de los datos, los recursos

proporcionados por la version gratuita de *Colab* no es suficiente. Según las pruebas realizadas, el tiempo de ejecución del código utilizando CPU o GPU remota proporcionada por *colab* es aproximadamente el triple comparado con el tiempo de ejecución empleando GPU local. Así, con el fin de reducir el coste de desarrollo, finalmente se decide utilizar *Jupyter Notebook* para conectar a la GPU local con el fin de acelerar la ejecución de algoritmos.

De acuerdo con la información proporcionada por *Colab*, en primer lugar es necesario descargar *Jupyter Notebook* de manera local [15], utilizando el siguiente código de instalación:

```
pip install notebook
```

Figura 4.1: Código de instalación de *Jupyter Notebook* [9]

Una vez instalado el *Jupyter Notebook*, para iniciar el programa es necesario ejecutar el comando *jupyter notebook* desde el terminal. Tras haber iniciado correctamente el *Jupyter*, se mostrará en la terminal un enlace de conexión que se debe de introducir posteriormente en *Colab*.

```
PS E:\> Jupyter Notebook

Read the migration plan to Notebook 7 to learn about the new features and the actions to take if you are using extensions.

https://jupyter-notebook.readthedocs.io/en/latest/migrate_to_notebook7.html

Please note that updating to Notebook 7 might break some of your extensions.

jupyter_http_over_ws extension initialized. Listening on /http_over_websocket
[W 19:37:31.060 NotebookApp] Error loading server extension jupyter_lsp
    Traceback (most recent call last):
      File "C:\Users\irene\AppData\Local\Programs\Python\Python311\Lib\site-packages\notebook\notebookapp.py", line 2050, in init_server_extensions
        func(self)
      File "C:\Users\irene\AppData\Local\Programs\Python\Python311\Lib\site-packages\jupyter_lsp\serverextension.py", line 76, in load_jupyter_server_extension
        nbapp.io_loop.call_later(0, initialize, nbapp, virtual_documents_uri)
        ^^^^^^^^^^^^^
    AttributeError: 'NotebookApp' object has no attribute 'io_loop'
[W 2024-06-15 19:37:32.834 LabApp] 'notebook_dir' has moved from NotebookApp to ServerApp. This config will be passed to ServerApp. Be sure to update your confi
g before our next release.
[W 2024-06-15 19:37:32.834 LabApp] 'notebook_dir' has moved from NotebookApp to ServerApp. This config will be passed to ServerApp. Be sure to update your confi
g before our next release.
[I 2024-06-15 19:37:32.841 LabApp] JupyterLab extension loaded from C:\Users\irene\AppData\Local\Programs\Python\Python311\Lib\site-packages\jupyterlab
[I 2024-06-15 19:37:32.842 LabApp] JupyterLab application directory is C:\Users\irene\AppData\Local\Programs\Python\Python311\share\jupyter\lab
[I 2024-06-15 19:37:32.842 LabApp] Extension Manager is 'pypi'.
[I 19:37:33.088 NotebookApp] Serving notebooks from local directory: E:\Jupyter notebook
[I 19:37:33.088 NotebookApp] Jupyter Notebook 6.5.4 is running at:
[I 19:37:33.088 NotebookApp] http://localhost:8888/?token=6d736c55d23a65212081abe94e5c4d2c1af629fa2b133bc0
[I 19:37:33.089 NotebookApp]  or http://127.0.0.1:8888/?token=6d736c55d23a65212081abe94e5c4d2c1af629fa2b133bc0
[I 19:37:33.089 NotebookApp] Use Control-C to stop this server and shut down all kernels (twice to skip confirmation).
[C 19:37:33.139 NotebookApp]

    To access the notebook, open this file in a browser:
        file:///C:/Users/irene/AppData/Roaming/jupyter/runtime/nbserver-30164-open.html
    Or copy and paste one of these URLs:
        http://localhost:8888/?token=6d736c55d23a65212081abe94e5c4d2c1af629fa2b133bc0
     or http://127.0.0.1:8888/?token=6d736c55d23a65212081abe94e5c4d2c1af629fa2b133bc0
0.00s - Debugger warning: It seems that frozen modules are being used, which may
0.00s - make the debugger miss breakpoints. Please pass -Xfrozen_modules=off
0.00s - to python to disable frozen modules.
0.00s - Note: Debugging will proceed. Set PYDEVD_DISABLE_FILE_VALIDATION=1 to disable this validation.
```

Figura 4.2: Captura de Terminal de *Jupyter Notebook*

Después de que se arranque correctamente y se obtiene el enlace de conexión, se traslada al *Colab*, haciendo click en el botón *Conectar*, situado

en la esquina superior derecha del navegador y seleccionando la opción *Conectar a un entorno de ejecución local*. Finalmente, se introduce la URL del paso anterior en el cuadro de diálogo que aparece tal como se indica en la Figura 4.4.

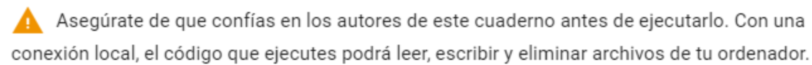
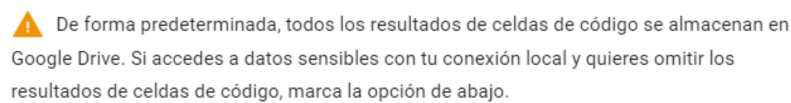

Figura 4.3: Cuadro de diálogo de conexión de *Colab*

Una vez completada la conexión local, se puede verificar la disponibilidad de GPU local introduciendo el siguiente código en *Colab*, donde podemos ver la infomación del GPU disponible, en este caso, se utiliza un GPU local del modelo *NVIDIA GeForce RTX 3060Ti*.

```
!nvidia-smi

Mon Jun 10 14:18:01 2024
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 551.23                 Driver Version: 551.23       CUDA Version: 12.4     |
|-----------------------------------------+----------------------+----------------------+
| GPU  Name                     TCC/WDDM  | Bus-Id          Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |           Memory-Usage | GPU-Util  Compute M. |
|                                         |                        |               MIG M. |
|=========================================+========================+======================|
|   0  NVIDIA GeForce RTX 3060 Ti    WDDM  |   00000000:01:00.0  On |                  N/A |
|  0%   48C    P8             17W /  240W |    1957MiB /   8192MiB |     39%      Default |
|                                         |                        |                  N/A |
+-----------------------------------------+------------------------+----------------------+
```

Figura 4.4: Información de la GPU local

Además de la conexión local, también es imprescindible descargar un conjunto de herramientas como *Ultralytics*, *Pytorch* para poder empezar

a desarrolar los algoritmos. A continuación, se explica detalladamente el proceso de instalación de cada una de ellas.

Según recomendaciones publicadas por Ultralytics, la instalación del conjunto de herramientas de *Ultralytics* es un proceso sencillo. Para llevarlo a cabo, simplemente se introduce el siguiente código en la línea de comandos de *Colab*.

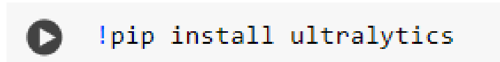

Figura 4.5: Comando de instalación de paquetes *Ultralytics* [8]

Por otra parte, hablando de *PyTorch*, existen dos versiones diferentes: la versión CPU y la versión GPU. Las diferencias entre estas dos versiones provienen principalmente de los entornos en los que se ejecutan y de los recursos de hardware que utilizan. Dado que la versión para GPU de *PyTorch* está diseñada para entornos de procesadores gráficos, y las GPU se utilizan ampliamente para entrenar modelos de ML ya que son capaces de procesar un gran número de tareas computacionales en paralelo, haciendo que el entrenamiento del ML sea mucho más rápido, por lo que se decide instalar la versión GPU del *Pytorch* para el desarrollo de algoritmos del presente proyecto.

En función del sistema operativo y los requisitos comentados en el capítulo 3, en el presente proyecto se han seleccionado instalar las opciones *Pytorch* versión 2.3.1 y CUDA versión 12.1 como se indica en la Figura 4.6, ya que CUDA se trata de una plataforma de cálculo paralelo desarrollado por NVIDIA que permite aumentar drásticamente el rendimiento computacional aprovechando la capacidad de procesamiento de las GPUs.

PyTorch Build	Stable (2.3.1)			Preview (Nightly)	
Your OS	Linux		Mac		Windows
Package	Conda	Pip		LibTorch	Source
Language	Python			C++ / Java	
Compute Platform	CUDA 11.8	CUDA 12.1	~~CUDA 12.4~~	~~ROCm 6.0~~	CPU
Run this Command:	pip3 install torch torchvision torchaudio --index-url https://download.pytorch.org/whl/cu121				

Figura 4.6: Comando de instalación de *Pytorch* [10]

Una vez finalizada la instalación de *Pytorch*, se puede comprobar usando el siguiente código en el que se ordena a detectar un dispositivo GPU, en caso afirmativo, se denominará *cuda:0*, mientras que en caso contrario, se denominará *cpu*. Como se puede apreciar en el resultado, el dispositivo utilizado es *cuda:0*, por lo que se puede confirmar que la versión GPU de Pytorch se ha instalado correctamente.

```
# Verificamos si estamos utilizando gpu
import torch

device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')
print(f"Using device: {device}")
```

Figura 4.7: Chequeo del la instalación de Pytorch

4.2. Procesamiento de datos

El conjunto de datos original utilizado en este proyecto es un conjunto de datos de PCB de acceso público, publicado por la Universidad de Pekín, que contiene 1.386 imágenes de resolución de 2.777 x 2.138 píxeles con distintos tipos de defectos [16] y sus correspondientes anotaciones. Entre los imágenes originales en formato PASCAL VOC (*Visual Object Classes*), se han seleccionado 693 imágenes aplicables a la tarea de detección, y cada imagen incluirá un número diferente de defectos. El detalle del número de defectos se desglosan en la Tabla 4.1 [1]:

Tipo de defectos	**Número de imágenes**	**Número de defectos**
missing hole	115	497
mouse bite	115	492
open circuit	116	482
short	116	491
spur	115	488
spurious copper	116	503
Total	693	2.953

Tabla 4.1: Dataset de PCB original [1]

Entre los distintos tipos de defectos se categorizan en los siguientes seis defectos princiales, tales como cortociruito (*short*), circuito abierto (*open circuit*), falta de agujero de soldadura (*missing hole*), mordiscos de ratón (*mouse bite*), astilla (*spur*) y cobre falso (*spurious copper*).

4.2.1. Ampliación de datos (*Data Augmentation*)

En el aprendizaje automático, el aumento de datos (*data augmentation*) es una técnica habitual para ampliar el conjunto de datos de entrenamiento con el fin de mejorar la generalización y el rendimiento del modelo. El aumento de datos genera nuevas muestras aplicando una serie de transformaciones o perturbaciones aleatorias a los datos originales, que son estadísticamente similares a las muestras originales pero con algunas diferencias.

El objetivo del aumento de datos es hacer que el modelo sea más robusto a los cambios en los datos de entrada introduciendo diversidad y variabilidad. Al aumentar la diversidad del conjunto de datos, el modelo puede aprender mejor las características invariantes y genéricas de los datos, mejorar la generalización y reducir el sobreajuste (*overftting*).

Entre los métodos más comunes de ampliación de datos se encuentran:

- Volteo horizontal/vertical: voltea la imagen horizontal o verticalmente para generar muestras invertidas.
- Recorte aleatorio: recorta aleatoriamente subimágenes de la imagen en diferentes posiciones y tamaños para simular cambios en el punto de vista o en la escala del objetivo.
- Rotación y escalado: realizar operaciones aleatorias de rotación y escalado en la imagen para añadir cambios de perspectiva y cambios de escala.
- Inyección de ruido: añadir ruido aleatorio, como ruido gaussiano, ruido pretzel, etc., a la imagen o los datos para aumentar la robustez del modelo frente al ruido.
- Ajuste de brillo y contraste: ajustar el brillo y el contraste de la imagen para aumentar la resistencia a los cambios de iluminación.

Con todo lo anteriormente explicado y ante un conjunto de datos pequeño, se adoptan técnicas de ampliación de datos antes de proceder

a entrenar los datos, resultando un nuevo conjunto de datos aumentado con imágenes de 600 x 600 pixeles, ya que dicho tamaño es el aceptado por la red, por lo que es necesario redimensionar las imágenes de entrada cuando sea necesario. En la Tabla 4.2 se muestran más detalles del conjunto de datos de defectos de PCB aumentados.

Tipo de defectos	**Número de imágenes**	**Número de defectos**
missing hole	1.832	3.612
mouse bite	1.852	3.684
open circuit	1.740	3.548
short	1.732	3.508
spur	1.752	3.636
spurious copper	1.760	3.676
Total	10.668	21.664

Tabla 4.2: Dataset de PCB aumentado [1]

4.2.2. Segmentación de datos

Para entrenar y evaluar eficazmente el modelo, se divide el conjunto de datos en subconjuntos de entrenamiento, de validación y de prueba. La segmentación de datos es un paso fundamental en la tarea de aprendizaje automático, que ayuda a garantizar que el modelo no se ajuste en exceso durante el proceso de entrenamiento, y al mismo tiempo proporcionar un conjunto de datos independiente para la evaluación final del rendimiento.

La proporción habitual para la segmentación de datos en el aprendizaje automático suele ser la siguiente: aproximadamente el 70 % - 80 % de los datos se reservan para el conjunto de entrenamiento, mientras que el 10 % - 20 % se destina al conjunto de validación y otro 10 % - 20 % se utiliza para el conjunto de prueba. Esta distribución ayuda a garantizar que el modelo se entrene adecuadamente, se ajuste correctamente y se evalúe de manera objetiva en datos no vistos durante el entrenamiento.

En concreto, se ha dividivo las 10.668 imágenes en subconjuntos de entrenamiento, validación y prueba en una proporción del 70 %, 15 %, 15 %, respectivamente. Esta proporción se utiliza ampliamente en la práctica

y puede proporcionar suficientes datos de entrenamiento, garantizando al mismo tiempo que haya suficientes datos para la validación y prueba del modelo. Los pasos específicos de segmentación son los siguientes, cuya implementación se especifica en el Anexo A:

- Conjunto de entrenamiento (70 %): se utiliza para el entrenamiento del modelo y contiene un total de 7.466 imágenes. A través del conjunto de entrenamiento, el modelo puede aprender las características y los patrones de los datos para un entrenamiento eficaz.
- Conjunto de validación (15 %): se utiliza para ajustar los parámetros del modelo y seleccionar el mejor modelo, y contiene un total de 1.600 imágenes. El conjunto de validación evalúa periódicamente el rendimiento del modelo durante el proceso de entrenamiento para ayudar a identificar y evitar problemas de sobreajuste.
- Conjunto de prueba (15 %): se utiliza para la evaluación final del modelo y contiene un total de 1.601 imágenes. Una vez completados el entrenamiento y la validación del modelo, éste se evalúa de forma independiente utilizando el conjunto de prueba para garantizar el rendimiento del modelo con datos desconocidos.

4.2.3. Conversión del formato de anotaciones

En el contexto del procesamiento de datos para entrenamiento de modelos, es habitual encontrarse distintos estándares de formato para las anotaciones de objetos en imágenes. Uno de los estándares más comunes es el formato PASCAL VOC, ampliamente utilizado en el ámbito de reconocimiento de objetos. En este apartado, se explora en qué consiste el formato PASCAL VOC, y por qué es necesario convertirlo al formato YOLO, y cómo se lleva a cabo esta conversión.

4.2.3.1. Formato PASCAL VOC

El formato PASCAL VOC es un estándar ampliamente reconocido para la anotación de imágenes en conjuntos de datos utilizados en tareas de detección de objetos. Este formato utiliza archivos XML estructurados que contienen información detallada sobre la ubicación y la categoría de objetos en imágenes. Cada archivo XML describe objetos, en este caso, defectos,

especificando la categoría mediante la etiqueta *name* y las coordenadas del cuadro delimitador (*xmin, ymin, xmax, ymax*) dentro de la etiqueta *bndbox*.


```xml
<?xml version="1.0" ?>
<annotation>
    <folder>PCB</folder>
    <filename>l_light_01_missing_hole_01_1_600</filename>
    <source>
        <database>My Database</database>
        <annotation>PCB</annotation>
        <image>flickr</image>
        <flickrid>NULL</flickrid>
    </source>
    <owner>
        <flickrid>NULL</flickrid>
        <name>idaneel</name>
    </owner>
    <size>
        <width>600</width>
        <height>600</height>
        <depth>3</depth>
    </size>
    <segmented>0</segmented>
    <object>
        <name>missing_hole</name>
        <pose>Unspecified</pose>
        <truncated>0</truncated>
        <difficult>0</difficult>
        <bndbox>
            <xmin>240</xmin>
            <ymin>404</ymin>
            <xmax>282</xmax>
            <ymax>437</ymax>
        </bndbox>
    </object>
</annotation>
```


Figura 4.8: Estructura de anotaciones en formato VOC

Por otro lado, el modelo YOLOv8 requiere un formato de anotación diferente para funcionar eficazmente durante el entrenamiento y la inferencia. YOLOv8 utiliza archivos de texto simples que enumeran objetos detectados, junto con las coordenadas normalizadas del centro del cuadro delimitador, así como su ancho y alto relativos a las dimensiones de la imagen. En la Figura 4.9 se presenta un ejemplo de anotaciones en formato YOLO.

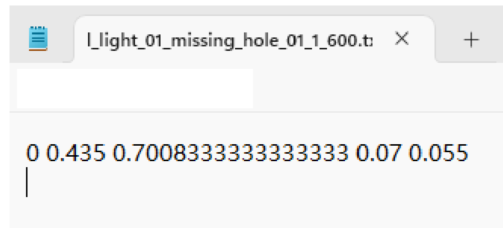

Figura 4.9: Ejemplo de etiqueta en formato YOLO

La diferencia fundamental entre los dos formatos radica en la representación de las coordenadas de los cuadros delimitadores y la forma en que se almacena la información de categoría. Mientras que PASCAL VOC emplea un enfoque más estructurado y detallado en archivos XML, YOLOv8 requiere un formato de texto plano más ligero y eficiente para manejar grandes volúmenes de datos de entrenamiento. Por lo que surge la necesidad de conversión del formato de anotaciones.

El proceso de conversión de anotaciones de PASCAL VOC a YOLO implica varios pasos clave. Se empieza extrayendo la categoría y las coordenadas del cuadro delimitador desde los archivos XML de PASCAL VOC. Posteriormente, las coordenadas absolutas del cuadro delimitadora se transforman a coordenadas normalizadas, calculando el centro, el ancho y el alto en relación con las dimensiones de la imagen. Finalmente, se genera un archivo de texto para cada imagen en formato YOLOv8, que incluye la categoría y las coordenadas normalizadas. Este proceso permite adaptar los datos para que sean compatibles con el modelo YOLO, optimizando así el entrenamiento y la inferencia. El despliegue del proceso de conversión completo se presentará detalladamente en el Anexo B.

4.3. Entrenamiento del modelo

Una vez preparado el conjunto de datos, se procede al entrenamiento del modelo. Entre las numerosas ventajas de YOLOv8, sobresale su facilidad de aplicación, lo que permite completar tareas de entrenamiento, validación y prueba con un código relativamente sencillo. En cuanto a los algoritmos de entrenamiento específicos, en el presente proyecto se ha tomado como referencia el código de ejemplo oficial proporcionado por *Ultralytics* [8] y se ha realizado las siguientes modificaciones:

```
from ultralytics import YOLO

# Load a model
model = YOLO("yolov8n.yaml")  # build a new model from YAML
model = YOLO("yolov8n.pt")  # load a pretrained model (recommended for training)
model = YOLO("yolov8n.yaml").load("yolov8n.pt")  # build from YAML and transfer weights

# Train the model
results = model.train(data="coco8.yaml", epochs=100, imgsz=640)
```

Figura 4.10: Ejemplo del código de entrenamiento con YOLOv8 [8]

- Se ha creado un nuevo archivo de configuración del conjunto de datos, denominado *pcb_dataset.yaml*, cuyo contenido consiste en parámetros específicos del conjunto de datos, tales como rutas de los datos de entrenamiento y validación, nombres de clases (array de *names*) y número de clases de defectos (6).

```
train: E:\Jupyter notebook\train_images
val: E:\Jupyter notebook\validation_images
test: E:\Jupyter notebook\test_images
nc: 6
names: ["missing_hole", "mouse_bite", "open_circuit", "short", "spur", "spurious_copper"]
splits:
  train: E:\Jupyter notebook\ImageSets\Main\train.txt
  val: E:\Jupyter notebook\ImageSets\Main\val.txt
  test: E:\Jupyter notebook\ImageSets\Main\test.txt
```

Figura 4.11: Estructura del fichero de configuración de datos

- Como se mencionó anteriormente, YOLOv8 presenta diferentes versiones adaptadas a diversas tareas, cada una de ellas con un número distinto de parámetros y tiempo de ejecución del código. En este estudio, se han seleccionado tres versiones para realizar pruebas y comparaciones: YOLOv8n (*nano*), YOLOv8s (*small*) y YOLOv8m (*medium*). La versión *medium* cuenta con más parámetros que las versiones *nano* y *small* debido a su mayor tamaño, lo cual resulta en un desempeño relativamente superior.

  Específicamente, la versión *nano* tiene 3,2 millones de parámetros y 225 capas, la versión *small* tiene 11,2 millones de parámetros y 225 capas, y por último, la versión *medium* posee 25,9 millones de parámetros y 295 capas. Las tres versiones comparten la misma estructura de red básica; la diferencia radica en que los modelos más grandes utilizan una estructura de red más profunda, con más capas convolucionales, lo cual conduce a un mayor número de parámetros y, por ende, a un tiempo de procesamiento más prolongado.

La versión *n* está diseñada para ser más rápida y ligera, adecuada para aplicaciones en dispositivos con recursos limitados, mientras que la versión *s* ofrece una mayor precisión a costa de un mayor consumo de recursos. La versión *m*, siendo la más grande, proporciona la mayor precisión y capacidad de detección, adecuada para aplicaciones que pueden soportar el mayor consumo de recursos requerido.

La comparación detallada de los resultados se expondrá en el siguiente capítulo.

- En cuanto a otros parámetros de entrenamiento, se ha modificado el tamaño de las imágenes a un valor de 600 x 600 píxeles, ya que el conjunto de datos está compuesto por imágenes de dicho tamaño. Además, se ha modificado el número de *epochs* (iteraciones), los cuales se ajustan en función del rendimiento de cada modelo. Tras varias pruebas, se ha precisado un número de épocas de 100 *epochs* iniciales. Sin embargo, este número resultó insuficiente en los resultados de la validación, requiriendo la adición de más iteraciones para maximizar el rendimiento del modelo. Por esta razón, también se ha probado a entrenar con un mayor número de *epochs*, cuyos resultados se detallarán en el capítulo 5.

  También cabe destacar que dentro de los parámetros de entrenamiento del modelo YOLOv8, se encuentra el parámetro denominado *patience*, cuya función principal consiste en monitorear el rendimiento del modelo durante el entrenamiento y decidir detener el proceso si no se detecta una mejora significativa después de un número específico de iteraciones consecutivas. Este parámetro establece el umbral de *patience*, determinando el número máximo de iteraciones permitidas sin observar mejoras antes de finalizar el entrenamiento. Este enfoque, también conocido como detención temprana (*early stop*), se utiliza principalmente para prevenir el sobreajuste del modelo.

  Finalmente, se recuerda que aparte de esta modificación, todos los demás parámetros se mantienen en sus valores por defecto. El listado completo de parámetros se detallará en el Anexo C.

- Otro aspecto que merece ser mencionado es que, durante el proceso de desarrollo, se ha investigado un método de optimización del tiempo de entrenamiento. Este método consiste en la congelación de un total de 10 capas del bloque de extracción de características, también conocido como *backbone*. Estas capas corresponden a las 10 primeras capas dentro de la estructura de red del YOLOv8. Al congelar estas capas, se detiene la actualización de sus parámetros durante el proceso

de entrenamiento, permitiendo que las capas restantes se actualicen conforme vayan aprendiendo nuevos conocimientos a lo largo del entrenamiento. Esta medida no solo acelera significativamente el entrenamiento del modelo al reducir la cantidad de cálculos necesarios, sino que también actúa como una estrategia para evitar el sobreajuste derivado del entrenamiento excesivo.

Este enfoque de congelar específicamente las primeras capas del bloque de extracción de características se detalla en la Figura 4.12, donde se ilustra cómo se implementa esta técnica para mejorar la eficiencia y efectividad del entrenamiento del YOLOv8.

```
#freeze all layers of backbone
freeze = [f'model.{x}.' for x in range(10)]  # layers to freeze
for k, v in model.named_parameters():
    v.requires_grad = True  # train all layers
    if any(x in k for x in freeze):
        print(f'freezing {k}')
        v.requires_grad = False
```

Figura 4.12: Código de congelación de capas

El principal efecto de este código es inactivar los parámetros de todas las capas del bloque de extracción de características, impidiendo su actualización. En este proyecto, esta operación se realiza de manera eficiente utilizando el parámetro de entrenamiento *freeze*. Este parámetro se configura en un valor de 10, lo que significa que se congelan las primeras 10 capas del modelo, es decir, todo el *backbone*. Esta configuración simplifica el proceso de congelación de capas, asegurando que solo las capas específicas seleccionadas no se actualicen durante el entrenamiento, mientras que las demás capas pueden ajustarse dinámicamente para mejorar el rendimiento del modelo.

Durante el proceso de pruebas, se exploran diferentes configuraciones de congelación de capas. Es importante destacar que no todas las pruebas realizadas en este estudio implican la congelación de capas del *backbone*. Esta fue una opción explorada para determinar su impacto en el rendimiento del modelo en diferentes escenarios y conjuntos de datos específicos.

- Para mejorar aún más la precisión del modelo YOLOv8, es fundamental ajustar algunos hiperparámetros claves durante el entrenamiento. Estos hiperparámetros incluyen:

- **Tamaño del lote (*batch size*)**: Los tamaños de lote más grandes pueden estabilizar el entrenamiento, pero requieren más memoria. En este estudio, se ha utilizado un tamaño de lote de 16, ajustado según la capacidad de la GPU disponible.
- **Tasa de aprendizaje (*learning rate, lr0*)**: Controla el tamaño del paso para las actualizaciones de los pesos; tasas más pequeñas permiten ajustes finos pero ralentizan la convergencia. Se ha configurado una tasa de aprendizaje inicial de 0,01, que se ajustará dinámicamente durante el entrenamiento.
- **Momento (*momentum*)**: Ayuda a acelerar los vectores de gradiente en las direcciones correctas, amortiguando las oscilaciones. El valor utilizado en este caso es 0,937.
- **Tamaño de la imagen (*image size, imgsz*)**: Tamaños de imagen más grandes pueden mejorar la precisión pero incrementan la carga computacional. Para este proyecto, se ha fijado el tamaño de imagen en 600 x 600 píxeles para coincidir con el conjunto de datos.

Capítulo 5

Pruebas y análisis de resultados

En esta sección se detallan específicamente ciertas pruebas clave evaluadas durante el desarrollo del estudio. Primero, se presentarán las métricas utilizadas para la evaluación de los modelos. Posteriormente, se procederá a evaluar tres modelos diferentes de manera independiente. Finalmente, se comparan los resultados de los tres modelos para obtener un resultado final que ajuste mejor a los objetivos establecidos.

5.1. Métricas de evaluación del modelo

Las métricas de evaluación del modelo son normas y criterios utilizados para medir el rendimiento de los modelos de ML, del cual ayuda a comprender el rendimiento del modelo en distintos aspectos, tales como la precisión, la recuperación, etc. Es importante destacar que en un primer momento se ha centrado en las métricas que evalúan la clasificación de objetos, y no su posicionamiento. A continuación, se presentan las métricas de evaluación del modelo más comunes:

5.1.1. Matriz de confusión

La matriz de confusión es un resumen de los resultados obtenidos por la red. El número de predicciones correctas e incorrectas se resume mediante valores numéricos y se desglosa por cada clase, mostrando cuántas veces el modelo acierta o se confunde. Las matrices de confusión proporcionan una forma bastante intuitiva y estructurada a la hora de mostrar la relación entre las predicciones de un modelo de clasificación y la situación real.

Por lo general, las definiciones de la matriz de confusión suele ser del caso de clasificación binaria, en la que la matriz de confusión está formada por las siguientes cuatro celdas claves, como se muestra en el Tabla 5.1:

- **Verdadero Positivo (TP, *True Positive*)**. Cuando el resultado real es positivo y el modelo predice un resultado positivo.
- **Falso Positivo (FP, *False Positive*)**. Cuando el resultado real es negativo pero el modelo predice un resultado positivo.
- **Falso Negativo (FN, *False Negative*)**. Cuando el resultado real es positivo pero el modelo predice un resultado negativo.
- **Verdadero Negativo (TN, *True Negative*)**. Cuando el resultado real es negativo y el modelo predice un resultado negativo.

<table>
<tr><th colspan="2" rowspan="2"></th><th colspan="2">Valores reales</th></tr>
<tr><th>Positivo</th><th>Negativo</th></tr>
<tr><th rowspan="2">Valores predichos</th><th>Positivo</th><td>Verdadero positivo</td><td>Falso positivo</td></tr>
<tr><th>Negativo</th><td>Falso negativo</td><td>Verdadero negativo</td></tr>
</table>

Tabla 5.1: Matriz de confusión

Sin embargo, en problemas de clasificación multiclase, como puede ser en este caso, la matriz de confusión presenta ciertas diferencias con respecto a la matriz de clasificación binaria. Esta matriz tiene dimensiones N x N, donde N es el número de clases. Cada fila representa los FP, y cada columna representa los FN, como se puede apreciar en la Figura 5.1. Cuando el índice de fila coincide con el de la columna, corresponde al TP, es decir, donde las predicciones del modelo coinciden con la realidad observada.

Figura 5.1: Matriz de confusión de clasificación multiclase

5.1.2. Precisión

La precisión se define como el número de predicciones correctas (TP) en aquellos datos que se predicen como positivos (TP + FP), y se calcula de la siguiente manera:

$$Precisión = \frac{TP}{TP + FP} \tag{5.1}$$

Para tareas de clasificación multiclase, la precisión se calcula como:

$$\text{Precisión}_k = \frac{MC(k,k)}{\sum_j MC(j,k)} \tag{5.2}$$

Donde:

- $MC(k,k)$ es el número de verdaderos positivos (TP) para la clase k.
- $\sum_j MC(j,k)$ es la suma total de todos los elementos que el modelo identifica como pertenecientes a la clase k, que incluye tanto los verdaderos positivos como los falsos positivos (FP).

5.1.3. Exhaustividad (*Recall*)

La exhaustividad o *recall* es la proporción de verdaderos positivos (TP) respecto a todos los verdaderos positivos y falsos negativos (FN). La fórmula general y la específica para la clase k son:

General:

$$\text{Recall} = \frac{TP}{TP + FN} \tag{5.3}$$

Para una clase k:

$$\text{Recall}_k = \frac{MC(k,k)}{\sum_i MC(k,i)} \tag{5.4}$$

Donde:

- $MC(k,k)$ es el número de verdaderos positivos (TP) para la clase k.
- $\sum_i MC(k,i)$ es la suma total de todos los elementos que pertenecen realmente a la clase k, que incluye tanto los TP como los FN.

5.1.4. Valor F1 (*F1 score*)

La precisión y la *recall* están inversamente relacionadas, es decir, cuando la tasa de precisión es alta, la tasa de *recall* disminuye, y en algunos escenarios es necesario equilibrar la precisión y la *recall*, por lo que suerge el valor F1.

El valor F1 evalúa de forma exhaustiva la tasa de precisión y la tasa de *recall*. Si la tasa de precisión y la tasa de *recall* son altas, el F1 también lo será. La fórmula general y la específica para la clase k son:

General:

$$F1 = 2 \times \frac{Precisión + Recall}{Precisión \times Recall} = \frac{2TP}{2TP + FN + FP} \tag{5.5}$$

Para una clase k:

$$F1_k = 2 \cdot \frac{\text{Precisión}_k \cdot \text{Recall}_k}{\text{Precisión}_k + \text{Recall}_k} \tag{5.6}$$

Donde:

- Precisión_k y Recall_k son la precisión y el recall específicos para la clase k, respectivamente.

5.1.5. Intersección sobre la unión (IoU)

La intersección sobre la unión (IoU) es una herramienta para medir la precisión del sistema en la tarea de localización de los objetos. Evalúa la exactitud de la localización o la precisión de la segmentación de un modelo calculando la relación entre el área de intersección del cuandro predicho (*bounding box*) y real, y el resultado oscila entre 0 y 1, donde un valor más alto indica una predicción más exacta del modelo. El IoU se utiliza a menudo para establecer un umbral para juzgar el rendimiento de un modelo.

La expresión matemática se describe a continuación:

$$IoU = \frac{\text{Área de intersección}}{\text{Área de unión}} \tag{5.7}$$

Figura 5.2: Ejemplo de IoU para varios *bounding box*

5.1.6. mAP

El término mAP (mean Average Precision) es una métrica comúnmente utilizada en la evaluación de modelos de detección de objetos. Se utiliza para medir la precisión del modelo en varias categorías de objetos.

La mAP se calcula promediando la precisión de cada categoría de objetos. En este caso, la precisión de las distintas categorías se determina valorando si la categoría es correcta o incorrecta con un nivel de confianza superior a un determinado umbral (*P_threshold*), y el solapamiento del *bounding box* predicho IoU de predicción y el *box* real es superior a un determinado umbral (*IoU_threshold*). Como se muestra en la Figura 5.3, si el $P_threshold = 0{,}6$, $IoU_threshold = 0{,}5$, el cuadro predicho se considera una predicción correcta y se registra como Verdadero positivo.

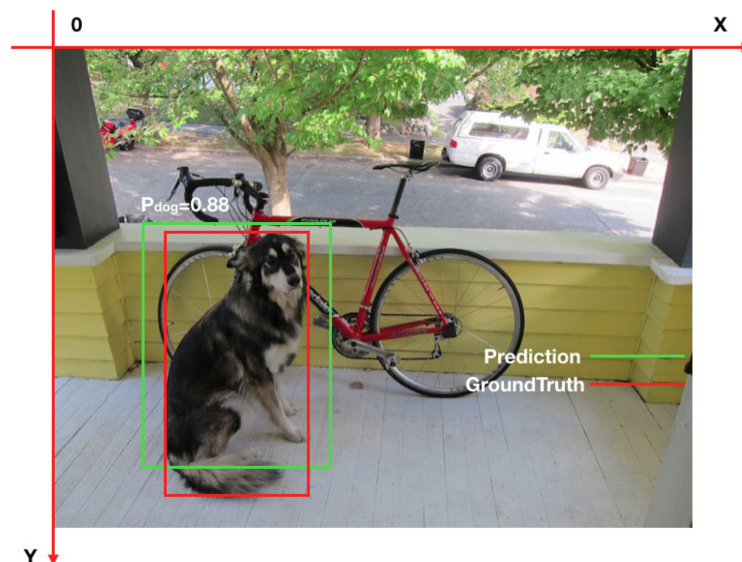

Figura 5.3: Ejemplo de detección de objetos

Existen distintos tipos de mAP según los valores de IoU utilizados para calcular las detecciones correctas. Algunos ejemplos son:

- **mAP@50**. Calculado con un umbral de IoU del 50 %.
- **mAP@50:95**. Calculado promediando los valores de mAP en 10 umbrales de IoU, desde 50 % hasta 95 % con incrementos del 5 %.

El mAP@50:95 proporciona una evaluación más detallada del rendimiento del modelo en comparación con un solo umbral de IoU.

5.2. Evaluación de modelos

Basándose en la técnica de aprendizaje por transferencia, se ha llevado a cabo un estudio exhaustivo para evaluar objetivamente el rendimiento de los modelos YOLOv8n, YOLOv8s, y YOLOv8m. Para tal fin, se ha realizado dos pruebas distintas: la primera consiste en entrenar el modelo manteniendo las capas del *backbone* congeladas, mientras que en la segunda prueba se entrena el modelo sin congelar dichas capas. El propósito de este análisis es proporcionar una comprensión clara y detallada de cómo las diferentes configuraciones de entrenamiento afectan el rendimiento del modelo. A continuación, se detallan los parámetros y el código utilizados en cada uno de los entrenamientos.

```
# modificamos el codigo para que utilice gpu en vez de cpu
import yaml
import torch

# Check if a GPU is available and set the device accordingly
device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')

# Load a model
model = YOLO('yolov8n.yaml').to(device)  # build a new model from YAML and move it to GPU
model = YOLO('yolov8n.pt').to(device)    # load a pretrained model and move it to GPU
model = YOLO('yolov8n.yaml').load('yolov8n.pt').to(device)  # build from YAML, transfer weights, and move to GPU

# Train the model on GPU
# path local
path_data = 'E:/Jupyter notebook/PCB_dataset.yaml'

# Move the model to the GPU if it's not there already
model = model.to(device)

# Specify that the model and data should be moved to GPU
results = model.train(data=path_data, epochs=100, imgsz=600, device=device)
```

Figura 5.4: Código de entrenamiento sin congelación de capas

```
import yaml
import torch

# Check if a GPU is available and set the device accordingly
device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')

# Load a model
model = YOLO('yolov8n.yaml').to(device)  # build a new model from YAML and move it to GPU
model = YOLO('yolov8n.pt').to(device)    # load a pretrained model and move it to GPU
model = YOLO('yolov8n.yaml').load('yolov8n.pt').to(device)  # build from YAML, transfer weights, and move to GPU

# Train the model on GPU
# path local
path_data = 'E:/Jupyter notebook/PCB_dataset.yaml'

# Move the model to the GPU if it's not there already
model = model.to(device)

# Specify that the model and data should be moved to GPU
results = model.train(data=path_data, epochs=50, imgsz=600, device=device,freeze=10)
```

Figura 5.5: Código de entrenamiento con congelación de capas

Primero se procede a la construcción del modelo YOLOv8n, seguido por la carga de los pesos preentrenados correspondientes. Una vez completada la carga del modelo, se configura el entrenamiento para realizar 50 iteraciones (*epochs*), utilizando una resolución inicial de imagen de 600 x 600 píxeles. Durante el entrenamiento efectivo, se ajusta esta resolución a 608 píxeles, dado que el tamaño de las imágenes debe ser un múltiplo de 32. Cabe señalar que todo el proceso se realiza utilizando recursos de procesamiento gráfico (GPU).

Además, se implementa, en una de las dos pruebas planteadas para la evalución de modelos, la técnica de congelación de capas en la que se congelan todas las capas del *backbone*. Para ello, se añade el parámetro *freeze* con un valor de 10 durante la configuración del entrenamiento. Esto implica que las primeras diez capas del modelo, que corresponden al *backbone*, permanecen congeladas, es decir, sus pesos no se actualizan durante el proceso de entrenamiento.

A continuación, se presentarán los resultados obtenidos a partir de estos dos entrenamientos diferentes. El primer resultado se obtiene de un modelo en el que no se ha aplicado la técnica de congelación de capas. En este caso, el parámetro *freeze* no ha sido utilizado, permitiendo que todas las capas del modelo participen en el entrenamiento y actualicen sus pesos.

El segundo resultado proviene de un modelo entrenado con la técnica de congelación de capas del *backbone*. Esta técnica tiene la ventaja de acelerar significativamente el proceso de entrenamiento, ya que reduce la cantidad de parámetros que necesitan ser ajustados. Sin embargo, es posible que esta estrategia resulte en una ligera disminución de la precisión final del

modelo, debido a que algunas de las capas no se optimizan completamente.

	YOLOv8n sin congelación	**YOLOv8n con congelación**
Tiempo de ejecución	1h 1m 8s	0h 54m 32s
mAP_0.5	0.988	0.966
mAP_0.5:0.95	0.614	0.533
Presición	0.984	0.961
Recall	0.985	0.935

Tabla 5.2: Comparativa de distintos entrenamientos de YOLOv8n

Los resultados muestran que la aplicación de la congelación durante el entrenamiento acelera significativamente el proceso de aprendizaje, siendo aproximadamente un 11 % más rápido que el entrenamiento sin congelación. No obstante, esta técnica también conlleva una ligera reducción en la precisión final alcanzada por el modelo. A pesar de la reducción en el tiempo de entrenamiento, es crucial considerar el equilibrio entre la velocidad y la precisión del resultado final al decidir la implementación de esta metodología. Esta comparación proporciona una base sólida para tomar decisiones informadas sobre la estrategia de entrenamiento más adecuada, teniendo en cuenta las necesidades específicas y los recursos disponibles en cada contexto.

Respecto al modelo YOLOv8s, se realizaron las mismas dos pruebas mencionadas anteriormente. La única diferencia radica en que se sustituye el modelo YOLOv8n por YOLOv8s en el proceso de entrenamiento. Se realizaron ajustes mínimos en el código específicamente para cargar y utilizar el modelo YOLOv8s. El resto del código permantece sin cambios. A continuación se muestra la modificación realizada en el código:

```
# Load a model
model = YOLO('yolov8s.yaml').to(device)
model = YOLO('yolov8s.pt').to(device)
model = YOLO('yolov8s.yaml').load('yolov8s.pt').to(device)
```

Figura 5.6: Código de configuración del modelo YOLOv8s

El procedimiento de entrenamiento para YOLOv8s se ejecuta de manera similar al utilizado para YOLOv8n, manteniendo consistencia en las

pruebas realizadas y la metodología aplicada, siendo los resultados como se presentan en la Tabla 5.3:

	YOLOv8s sin congelación	**YOLOv8s con congelación**
Tiempo de ejecución	1h 33m 28s	1h 11m 45s
mAP_0.5	0.990	0.985
mAP_0.5:0.95	0.664	0.585
Presición	0.984	0.981
Recall	0.986	0.972

Tabla 5.3: Comparativa de distintos entrenamientos de YOLOv8s

Al aplicar el mismo método de evaluación, se examina también el rendimiento del modelo YOLOv8m. Los resultados obtenidos son similares a los de los dos modelos anteriores.

	YOLOv8m sin congelación	**YOLOv8m con congelación**
Tiempo de ejecución	3h 6m 10s	1h 55m 33s
mAP_0.5	0.990	0.990
mAP_0.5:0.95	0.676	0.628
Presición	0.984	0.984
Recall	0.989	0.978

Tabla 5.4: Comparativa de distintos entrenamientos de YOLOv8m

En el análisis comparativo de los modelos YOLOv8n, YOLOv8s y YOLOv8m bajo diferentes configuraciones de entrenamiento, se observa que YOLOv8m ofrece el mejor rendimiento en términos de precisión, seguido por YOLOv8s y luego YOLOv8n. Sin embargo, conforme aumenta el tamaño del modelo, también se incrementa la diferencia en el tiempo de entrenamiento al aplicar congelación de capas, destacándose en YOLOv8m con una reducción significativa de más de una hora en comparación con el entrenamiento completo. La elección entre congelar o no congelar las capas del *backbone* debe considerar el equilibrio entre la velocidad de entrenamiento y la precisión requerida para la aplicación específica del modelo.

5.3. Análisis conjunto

Después de la evaluación métrica de tres modelos diferentes, los resultados indican que el modelo YOLOv8m presenta el mejor rendimiento en términos de indicadores de precisión. Sin embargo, su tiempo de ejecución es considerablemente más largo. Por otro lado, el modelo YOLOv8n tiene un tiempo de ejecución más corto, pero la precisión y el mAP son relativamente más bajos. Considerando todos estos factores, se decidió finalmente utilizar el modelo YOLOv8s para cumplir con los objetivos de este trabajo.

En cuanto a la técnica de congelación de capas, se determina que al no utilizar esta técnica, la precisión y el mAP son relativamente más altos, y el tiempo de ejecución no es excesivo. Por lo tanto, la decisión final es entrenar el modelo sin utilizar la técnica de congelación. Esta elección se basa en un análisis exhaustivo de los resultados obtenidos, buscando un equilibrio óptimo entre precisión y eficiencia temporal.

Una vez decidico utilizar el modelo YOLOv8s, se procede a analizar los resultados obtenidos durante el entrenamiento. Tras investigaciones sobre dichos resultados, se observa que la tendencia de la función de pérdida y la del mAP de 50 % a 95 % aún no se han estabilizado, lo que indica la necesidad de más entrenamiento y un mayor número de iteraciones (*epochs*). Por consiguiente, se decide realizar un entrenamiento de 400 *epochs* en el modelo.

Para asegurar la efectividad del entrenamiento, se establece un valor de *patience* de 20. Esto significa que, si no se observa una mejora en el rendimiento dentro de 20 *epochs* consecutivas, el entrenamiento se detendrá automáticamente. Este hace que, durante el proceso de entrenamiento de 400 *epochs*, se pueda garantizar que los resultados obtenidos no se vean afectados negativamente por un posible sobreajuste (*overfitting*), lo que optimiza el rendimiento del modelo YOLOv8s, asegurando así la validez y la calidad de los resultados de entrenamiento.

Además, a través de un entrenamiento más extenso, se espera que la precisión de la detección de cajas mejore, dado que el objetivo del presente proyecto es detectar objetos pequeños, lo cual es bastante desafiante. Si se pueden obtener mejores resultados con más entrenamiento, sería un logro significativo, ya que los resultados hasta este momento en términos de precisión y *recall* ya son bastante buenos, alcanzando, aproximádamente, hasta un 98 % de precisión y de *recall.*

5.4. Resultados finales

Después de 400 *epochs* de entrenamiento, se obtiene los siguientes resultados, cuya representación gráfica se muestra en la Figura 5.7:

- Las tres pérdidas disminuyen gradualmente con el número de iteraciones de entrenamiento, lo que indica que el modelo está optimizando progresivamente sus capacidades de localización y clasificación.
- La precisión y el *recall* se estabilizan en un valor de 99 %, lo cual refleja un rendimiento excepcional.
- El rendimiento en el conjunto de validación es igualmente notable, ya que las tres pérdidas diferentes disminuyen gradualmente y el mAP50 se estabiliza en un 99,4 %. En cuanto al mAP50-95, su rendimiento continúa mejorando, lo cual es normal debido a que las curvas de pérdida y mAP50-95 suelen tardar más en estabilizarse.

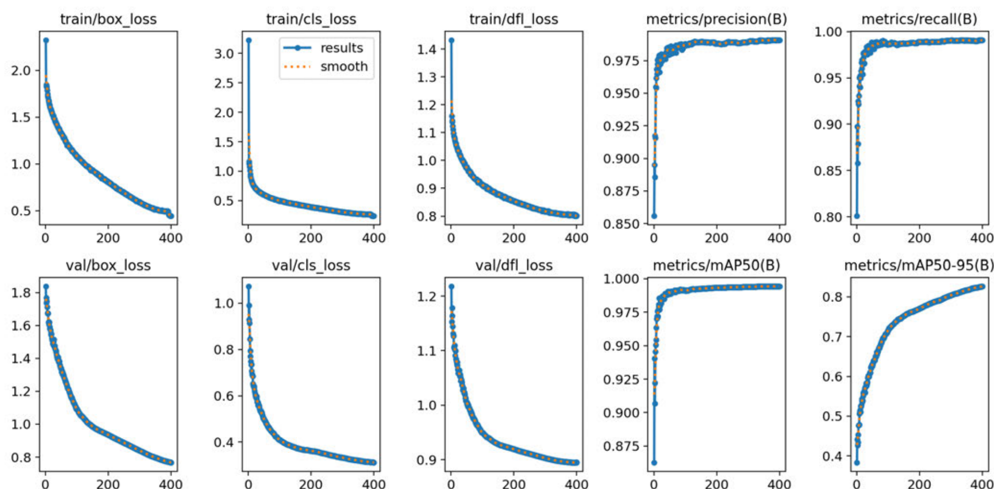

Figura 5.7: Resultado final del entrenamiento con YOLOv8s

En cuanto a otras métricas de evaluación, cabe mencionar la curva de precisión-recall, en la que se genera una curva con cada combinación de valores de precisión y de recall para cada clase de defectos con un umbral de decisión dado. Esta curva proporciona una representación clara de la capacidad del modelo para mantener altos niveles de precisión sin sacrificar el recall, lo que es crucial para tareas de clasificación y detección.

Como se puede apreciar en la Figura 5.8, el modelo mantiene una precisión y una exhaustividad muy altas (cercanas a 1) en todas las categorías, lo que indica un rendimiento excelente en tareas de detección y clasificación. Las curvas de precisión-recall de cada categoría se superponen casi por completo, lo que demuestra la estabilidad y coherencia del modelo en todas las categorías. Este solapamiento indica que el modelo es igualmente efectivo en identificar y clasificar distintos tipos de objetos, subrayando su capacidad para generalizar de manera uniforme en diferentes escenarios.

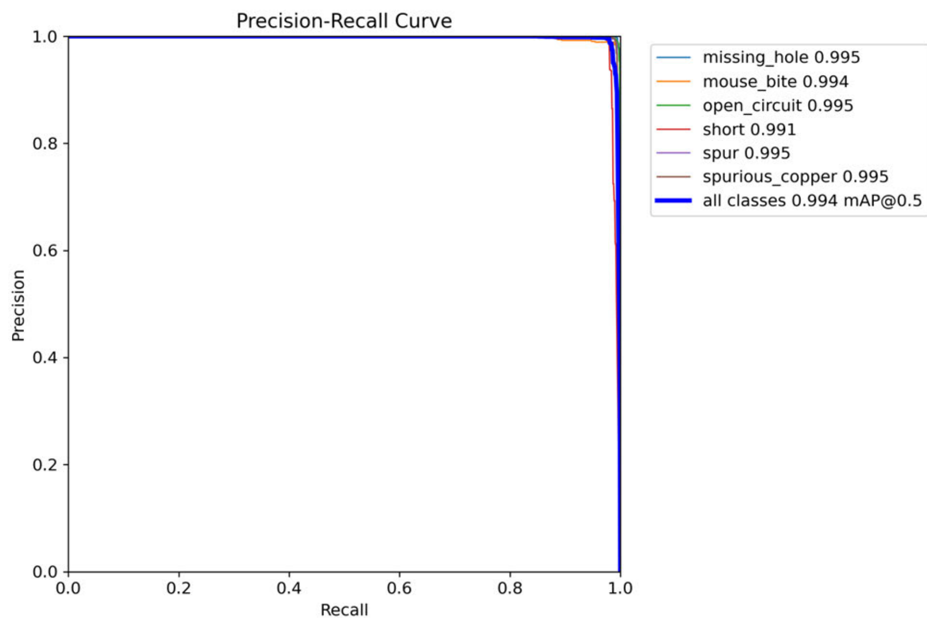

Figura 5.8: Curva Preción-Recall

Por último, se procede a probar el modelo entrenado con varias imagenes seleccionadas al azar del conjunto de imágenes de prueba, cuyos resultados se presenta a continuación. Esta prueba demuestra la efectividad del modelo en un contexto real, evidenciando su capacidad para realizar detecciones y clasificaciones precisas en nuevas imágenes no vistas durante el entrenamiento.

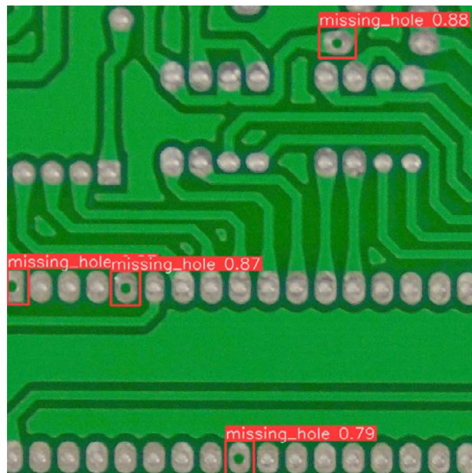

Figura 5.9: missing hole

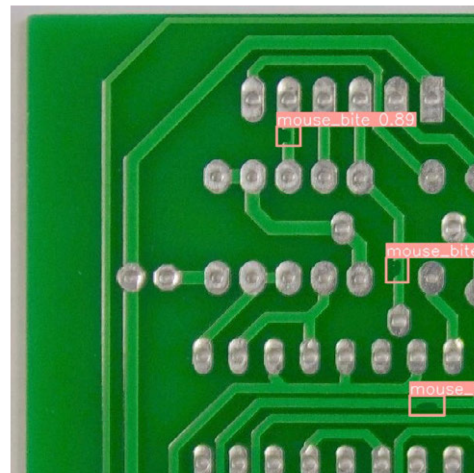

Figura 5.10: mouse bite

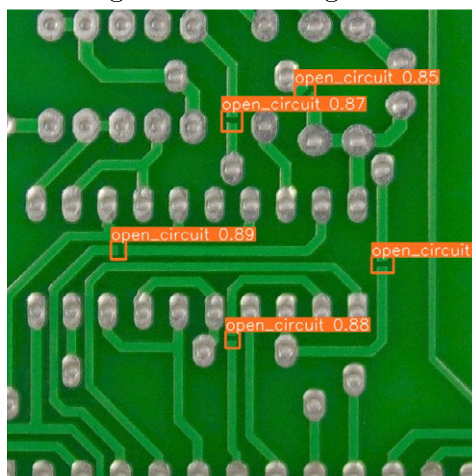

Figura 5.11: open circuit

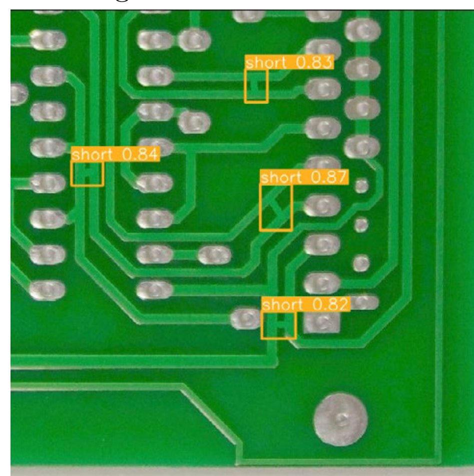

Figura 5.12: short

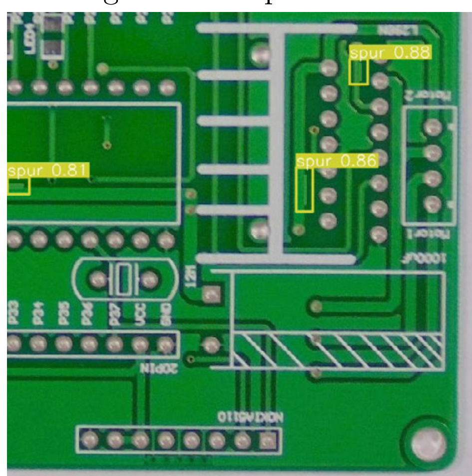

Figura 5.13: spur

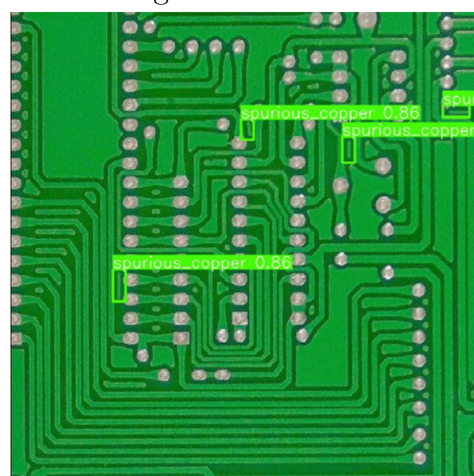

Figura 5.14: spurious copper

Aunque la precisión promedio de los resultados finales alcanza el 99,4 %, todavía se presentan casos de predicciones incorrectas. A continuación, se muestran dos ejemplos diferentes de tales errores: uno en el que no se detectan los defectos reales que deben de haberse identificado, y otro en el que se señalan defectos adicionales inexistentes. En el primer conjunto de gráficos, se detectan únicamente 3 defectos cuando en realidad deberían haber sido 4. En el segundo conjunto de gráficos, el defecto real era 1, pero el sistema identifica 2. Una posible explicación de estos errores podría radicar en la proximidad del defecto al borde de la imagen, lo cual podría disminuir la capacidad de la red para detectarlo correctamente, ya que no se logra un reconocimiento adecuado.

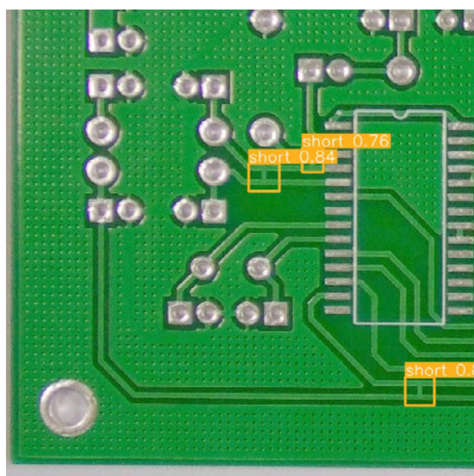

Figura 5.15: Ejemplo del error de predición caso 1

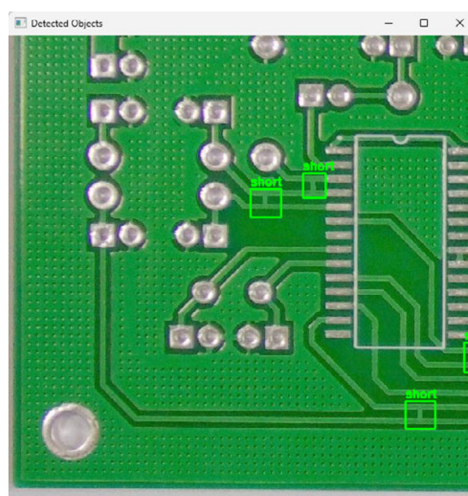

Figura 5.16: Imagen original

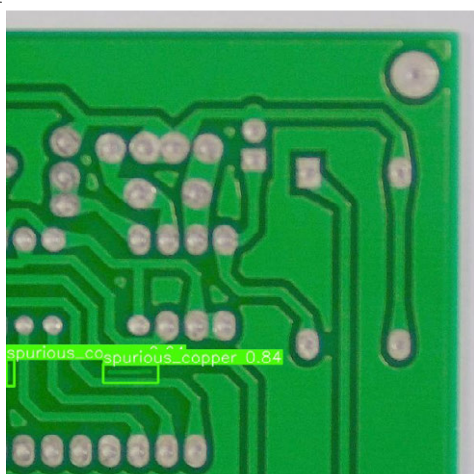

Figura 5.17: Ejemplo del error de predición caso 2

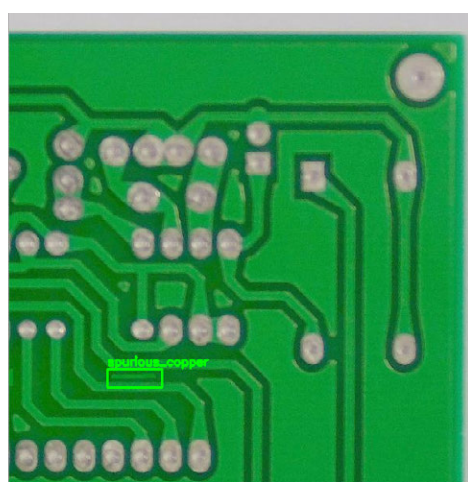

Figura 5.18: Imagen original

Estas dos situaciones distintas también invitan a la reflexión sobre si es preferible enfocarse en la precisión a expensas del recall, o mantener un alto recall sacrificando la precisión. En la industria de la fabricación electrónica, un pequeño defecto puede llevar al descarte del producto final. En este contexto, mantener un alto recall es más crucial que una alta precisión, debido a que pasar por alto un verdadero defecto puede tener consecuencias muy graves. En tales casos, es mucho mejor detectar varios defectos erróneos que no detectar un defecto real.

Capítulo 6

Presupuesto

En este capítulo se ofrece una descripción detallada de los costes asociados al desarrollo de este Proyecto Fin de Grado en un contexto comercial. Es fundamental señalar que para la realización de este proyecto se han utilizado recursos y servicios de computación gratuitos.

A continuación, se presenta los detalles de los costes estimados que implicaría llevar a cabo la solución propuesta:

Tabla 6.1: Tabla de presupuesto del proyecto de tesis

Concepto	**Detalles**	**Coste/unidad**	**Unidades**	**Total (€)**
Hardware	CPU	305 (€)	1	305 (€)
	Memoria RAM	99,90 (€)	1	99,90 (€)
	Tarjeta gráfica	523,61 (€)	1	523,61 (€)
	Monitor	220 (€)	1	220 (€)
	Otros periféricos	850 (€)	1	850 (€)
Electricidad		30 (€ / mes)	4 meses	120 (€)
Salario del programador [17]	Se suma 30 % de seguridad social	18 * 1,3 (€ / hora)	500 horas	11.700 (€)
Total				**13.818,51 (€)**

Capítulo 7

Impactos del proyecto

El desarrollo de un sistema de detección de defectos en placas de circuito impreso (PCB) basado en técnicas de aprendizaje automático es un avance significativo en el campo de la manufactura y la tecnología de la información. Las PCB son componentes esenciales en prácticamente todos los dispositivos electrónicos modernos, desde teléfonos móviles hasta equipos médicos y sistemas de control industrial. La precisión y la fiabilidad de estos componentes son cruciales para el rendimiento y la seguridad de los dispositivos finales. Sin embargo, la detección de defectos en las PCB es un proceso complejo y crítico que tradicionalmente ha requerido una inspección manual intensiva y propensa a errores humanos.

Al implementar un sistema automatizado de detección de defectos, se puede mejorar significativamente la eficiencia y la precisión del proceso de inspección, reduciendo tanto los costos de producción como el desperdicio de materiales, y aumentando la calidad y la fiabilidad de los productos electrónicos. Este proyecto no solo tiene el potencial de transformar la industria electrónica, sino que también puede tener implicaciones positivas en varias otras áreas.

- **Impacto en la Salud y Seguridad**. El uso de sistemas automatizados para la detección de defectos en PCB puede mejorar las condiciones de trabajo en fábricas y plantas de ensamblaje. Al reducir la exposición de los trabajadores a tareas que implican inspecciones visuales continuas, se puede disminuir el riesgo de fatiga visual y otros problemas de salud relacionados. Además, la mejora en la calidad de los PCB contribuye a la producción de dispositivos electrónicos más seguros y fiables.

- **Implicaciones Ambientales**. La detección temprana de defectos en PCB puede reducir el desperdicio de materiales y recursos, ya que se evita la producción de lotes defectuosos y la necesidad de reprocesar o desechar productos defectuosos. Esto, a su vez, disminuye la cantidad de residuos electrónicos y contribuye a un uso más eficiente de los recursos naturales, apoyando así la sostenibilidad ambiental.

- **Implicaciones Tecnológicas**. El uso de técnicas de ML para la detección de defectos en PCB representa un avance significativo en la automatización de procesos industriales. Este sistema puede servir como un modelo para la implementación de tecnologías similares en otras áreas de manufactura y producción, promoviendo la adopción de tecnologías de inteligencia artificial y aprendizaje automático en la industria.
- **Implicaciones Industriales**. La industria de la electrónica puede beneficiarse enormemente de la integración de este sistema, ya que mejora la calidad y la fiabilidad de los productos finales.

En cuanto a la aportación a los Objetivos de Desarrollo Sostenible (ODS) destaca lo siguiente [18]:

- ODS 3: Salud y Bienestar: Mejora las condiciones de trabajo y reduce los riesgos de salud asociados con tareas repetitivas y manuales, contribuyendo al bienestar de los trabajadores.
- ODS 8: Trabajo Decente y Crecimiento Económico: Mejora la eficiencia de producción y crea empleos en áreas tecnológicas avanzadas, apoyando el crecimiento económico sostenido y sostenible.
- ODS 9: Industria, Innovación e Infraestructura: Promueve la innovación y contribuye al desarrollo de infraestructuras resilientes mediante la implementación de tecnologías avanzadas en la producción industrial.
- ODS 12: Producción y Consumo Responsables: Reduce el desperdicio de materiales y mejora la eficiencia de recursos, apoyando patrones sostenibles de producción y consumo.

Capítulo 8

Conclusiones

En este proyecto, se ha desarrollado con éxito un sistema de detección de defectos en placas de circuito impreso (PCB) utilizando técnicas de aprendizaje automático. Para alcanzar este objetivo, se ha empleado el modelo YOLOv8s en las fases de entrenamiento, validación y pruebas. Adicionalmente, se han implementado técnicas de aprendizaje por transferencia para optimizar tanto el rendimiento como la velocidad del modelo.

Para garantizar la capacidad del modelo de identificar con precisión una amplia gama de defectos en PCBs, utilizamos un conjunto de datos original de 693 imágenes proporcionado por la Universidad de Pekín, enriquecido mediante técnicas de aumento de datos para garantizar la diversidad del conjunto de datos de los defectos, resultando un conjunto final de 10.668 imágenes. Estos datos son preprocesados adecuadamente para cumplir con los requisitos de entrada del modelo YOLOv8s. Posteriormente, se emplean técnicas de aprendizaje por transferencia, ajustando el modelo YOLOv8s con pesos previamente entrenados. Este enfoque no solo acelera el proceso de entrenamiento, sino que también mejora significativamente la precisión de detección del modelo. Además, se realizan múltiples experimentos para determinar los parámetros de entrenamiento y las configuraciones de hiperparámetros óptimas.

Durante las fases de validación y prueba, se evalúa exhaustivamente el rendimiento del modelo. Los resultados demuestran que el modelo posee una capacidad de detección sobresaliente, con una precisión media del 99,4 % en todas las categorías. Este resultado no solo cumple con los objetivos de desarrollo establecidos, sino que también proporciona una base sólida para futuras aplicaciones prácticas.

Los resultados de este proyecto demuestran que es posible lograr una detección de alta precisión de defectos en las PCB utilizando el modelo YOLOv8s y técnicas de aprendizaje por transferencia. El sistema demuestra un desempeño excelente en las pruebas prácticas, satisfaciendo los requisitos de las aplicaciones industriales.

En conclusión, este proyecto valida el potencial del aprendizaje automático en la detección de defectos en PCB y ofrece una referencia valiosa para investigaciones futuras en este campo, contribuyendo al avance de la tecnología de detección de PCB.

En cuanto a las líneas futuras, se puede notar en el transcurso de este proyecto que, aunque la precisión y el recall han mostrado resultados notablemente buenos, las métricas de pérdida y mAP obtenidas continúan mejorando. Por lo que se puede continuar explorando varias áreas de optimización y ampliación del sistema. Algunas posibles direcciones pueden ser:

- Optimización de hiperparámetros: Continuar ajustando los hiperparámetros del modelo para mejorar aún más el rendimiento y la precisión de detección.
- Desarrollo de Interfaces de Usuario: Diseñar y desarrollar interfaces de usuario intuitivas y fáciles de usar que faciliten la interpretación de los resultados y la toma de decisiones por parte de los operadores en las líneas de producción.
- Ampliación de Conjuntos de Datos: Incluir nuevos conjuntos de datos que presenten características y tipos de anotación diferentes, lo que puede enriquecer el entrenamiento del modelo y mejorar su capacidad para detectar una mayor variedad de defectos.

Bibliografía

[1] R. Ding, L. Dai, G. Li, and H. Liu, "Tdd-net: a tiny defect detection network for printed circuit boards," *CAAI Transactions on Intelligence Technology*, vol. 4, pp. 110–116, Junio 2019.

[2] G. Jocher, Fatih, Laughing, and Burhan. (2023) Model training with ultralytics yolo. (Acceso: 15.05.2024). [Online]. Available: https://docs.ultralytics.com/modes/train/

[3] MordorIntelligence. (2024) Tamaño del mercado de pcb y análisis de participación tendencias de crecimiento y pronósticos (2024-2029). (Acceso: 03.06.2024). [Online]. Available: https://www.mordorintelligence.com/es/industry-reports/printed-circuit-board-market

[4] H. Xu, L. Wang, and F. Chen, "Advancements in electric vehicle pcb inspection: Application of multi-scale cbam, partial convolution, and nwd loss in yolov5," *World Electric Vehicle Journal*, vol. 15, Enero 2024.

[5] Proto-Electronics. Pruebas y métodos de inspección de placas de circuito impreso (pcb). (Acceso: 3.07.2024). [Online]. Available: https://www.proto-electronics.com/es/blog/resumen-pruebas-metodos-inspeccion-placas-circuito-impreso-pcb

[6] S. Halbe. Object detection and instance segmentation: A detailed overview. (Acceso: 3.07.2024). [Online]. Available: https://medium.com/swlh/object-detection-and-instance-segmentation-a-detailed-overview-94ca109274f2

[7] J. Terven and D. Cordova-Esparza, "A comprehensive review of yolo architectures in computer vision: From yolov1 to yolov8 and yolo-nas," *Mach. Learn. Knowl. Extr.*, vol. 5, pp. 1680–1716, Noviembre 2023.

[8] Ultralytics. Ultralytics. (Acceso: 07.06.2024). [Online]. Available: https://github.com/ultralytics/ultralytics

[9] Jupyter. Installing jupyter. (Acceso: 15.06.2024). [Online]. Available: https://jupyter.org/install

[10] Pytorch. Start locally. (Acceso: 15.06.2024). [Online]. Available: https://pytorch.org/get-started/locally/

[11] EMSG. Common pcb inspection equipment and techniques. (Acceso: 3.07.2024). [Online]. Available: https://emsginc.com/resources/pcb-inspection-techniques-and-technologies/

[12] K. Fukushima, "Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position," *Biological Cybernetics*, vol. 36, pp. 193–202, Abril 1980.

[13] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE.*, pp. 2278–2324, Noviembre 1988.

[14] M. Sotaquirá. La función de activación. (Acceso: 7.07.2024). [Online]. Available: https://www.codificandobits.com/blog/funcion-de-activacion/#la-funci%C3%B3n-sigmoidal-%CF%83

[15] Colaboratory. Entornos de ejecución locales. (Acceso: 15.06.2024). [Online]. Available: https://research.google.com/colaboratory/intl/es/local-runtimes.html

[16] L. Dai. Pku-market-pcb. (Acceso: 17.06.2024). [Online]. Available: https://robotics.pkusz.edu.cn/resources/dataset/

[17] talent. Salario medio para ingeniero de software en españa, 2024. (Acceso: 20.06.2024). [Online]. Available: https://es.talent.com/salary?job=ingeniero+de+software

[18] ONU. Objetivos de desarrollo sostenible. (Acceso: 24.06.2024). [Online]. Available: https://www.un.org/sustainabledevelopment/es/objetivos-de-desarrollo-sostenible/

[19] G. Jocher. (2023) Ultralytics yolov8 modes. (Acceso: 15.05.2024). [Online]. Available: https://docs.ultralytics.com/modes/

[20] IBM. What are convolutional neural networks? (Acceso: 30.05.2024). [Online]. Available: https://www.ibm.com/topics/convolutional-neural-networks

[21] J. Wang, R. Turko, O. Shaikh, H. Park, N. Das, F. Hohman, M. Kahng, and P. Chau. Cnn explainer. (Acceso: 30.05.2024). [Online]. Available: https://poloclub.github.io/cnn-explainer/

[22] B. Kumar. (2021) Convolutional neural networks: A brief history of their evolution. (Acceso: 31.05.2024). [Online]. Available: https://medium.com/appyhigh-technology-blog/convolutional-neural-networks-a-brief-history-of-their-evolution-ee3405568597

[23] AWS. What is transfer learning? - transfer learning in machine learning explained. (Acceso: 02.06.2024). [Online]. Available: https://aws.amazon.com/what-is/transfer-learning/

[24] F. Giménez-Palomares, J. A. Monsoriu, and E. Alemany-Martínez, “Aplicación de la convolución de matrices al filtrado de imágenes,” *Modelling in Science Education and Learning*, vol. 9, 2016.

[25] MMYOLO. Algorithm principles and implementation with yolov8. (Acceso: 07.06.2024). [Online]. Available: https://mmyolo.readthedocs.io/en/latest/recommended_topics/algorithm_descriptions/yolov8_description.html

[26] A. Medewar. Mean average precision (map) in object detection. (Acceso: 25.06.2024). [Online]. Available: https://abhishri-medewar.medium.com/mean-average-precision-map-in-object-detection-78900922b3f0

[27] Proto-Electronics. ¿qué es inspección aoi y para qué sirve en la fabricación de circuitos impresos? (Acceso: 3.07.2024). [Online]. Available: https://microensamble.com/inspeccion-aoi-sirve-la-fabricacion-circuitos-impresos/

[28] J. Sensio. Detección de objetos. (Acceso: 3.07.2024). [Online]. Available: https://juansensio.com/blog/048_cv_detection

[29] R. Gandhi. R-cnn, fast r-cnn, faster r-cnn, yolo — object detection algorithms. (Acceso: 4.07.2024). [Online]. Available: https://towardsdatascience.com/r-cnn-fast-r-cnn-faster-r-cnn-yolo-object-detection-algorithms-36d53571365e

Anexos

Apéndice A

Segmentación de datos

El objetivo de la segmentación de datos consiste en dividir el conjunto de imágenes en tres subconjuntos diferentes para el entrenamiento, la validación y la prueba, respectivamente. En este apartado se explica detalladamente la implementación de la segmentación de datos y los pasos que han seguido para llevarla a cabo:

1. Importar las librerías necesarias para la operación de segmentación y definir los porcentajes de segmentación de cada subconjunto de datos, 70 % para el subconjunto de entrenamiento, 15 % para el subconjunto de validación, y 15 % para el subconjunto de prueba.

```
import os
import random

# Definir los porcentajes para cada conjunto
train_percent = 0.7
val_percent = 0.15
test_percent = 0.15
```

Figura A.1: Definición del porcentaje de segmentación

2. Se define las rutas necesarias para la segmentación, así como la ruta de todas las anotaciones de las imágenes *dir_annotations*, la ruta de los archivos que contienen los nombres de cada subconjunto de imágenes *txtsavepath*.

```
# path local
path_dataset = 'E:/Jupyter notebook/'

dir_annotations = os.path.join(path_dataset, 'Annotations')
txtsavepath = os.path.join(path_dataset, 'ImageSets', 'Main')
total_xml = os.listdir(dir_annotations)
```

Figura A.2: Definición de las rutas de trabajo

3. Se calcula el número total de imágenes que debe de contener cada subconjunto de imágenes de acuerdo con el porcentaje anteriormente definido. Según el número calculado, se asignan las imágenes de forma aleatoria a cada subconjunto.

```
num = len(total_xml)
print(f"num_total_xml: {num}")
indices = list(range(num))
print(f"indices: {indices}")

# Calcular el tamaño de cada conjunto
num_train = int(num * train_percent)
num_val = int(num * val_percent)
num_test = num - num_train - num_val

# Mezclar los índices y dividir en conjuntos
random.shuffle(indices)
train_indices = indices[:num_train]
val_indices = indices[num_train:num_train + num_val]
test_indices = indices[num_train + num_val:]
```

Figura A.3: Asignación de imágenes a cada subconjunto

4. Por último, se escribe el nombre de las imágenes asignadas en el paso anterior en cada fichero de txt correspondientes a cada subconjunto dividido.

```
# Crear los archivos necesarios
ftrain = open(os.path.join(txtsavepath, 'train.txt'), 'w')
fval = open(os.path.join(txtsavepath, 'val.txt'), 'w')
ftest = open(os.path.join(txtsavepath, 'test.txt'), 'w')

# Escribir los nombres de los archivos en los conjuntos correspondientes
for i in train_indices:
    name = total_xml[i][:-4] + '\n'
    ftrain.write(name)

for i in val_indices:
    name = total_xml[i][:-4] + '\n'
    fval.write(name)

for i in test_indices:
    name = total_xml[i][:-4] + '\n'
    ftest.write(name)

# Cerrar los archivos
ftrain.close()
fval.close()
ftest.close()
```

Figura A.4: Escritura de imágenes a cada subconjunto

Apéndice B

Convesión de anotaciones VOC a YOLO

En este capítulo se detalla los algoritmos utilizados para convertir los datos de anotación de detección de objetos en formato VOC (archivo de anotación XML) a un archivo de texto en formato YOLO para su posterior entrenamiento con YOLOv8.

1. Importar las librerías necesarias para este proceso y definir la función de conversión, en el cual se incluyen tres parámetros de entrada tales como la ruta del directorio de archivos XML en formato VOC (*voc_dir*), la ruta del directorio de salida de archivos de texto en formato YOLO (*output_dir*) y el listado de nombres de categorías para el etiquetado (*class_names*).

```
import os
import xml.etree.ElementTree as ET

def convert_voc_to_yolo(voc_dir, output_dir, class_names):
```

Figura B.1: Definición de la función de convesión

2. Gestión del directorio de salida: En caso de que el directorio de salida no exista, se procederá a crear uno nuevo. En el caso de que ya exista, se llevará a cabo la eliminación de todos los archivos del directorio existente para permitir la regeneración de un nuevo archivo de texto con el formato YOLO requerido.

```
# si el directorio no exista, se crea
if not os.path.exists(output_dir):
    os.makedirs(output_dir)
else:
    # Si el directorio ya existe, eliminamos todos los archivos en él
    for file in os.listdir(output_dir):
        file_path = os.path.join(output_dir, file)
        try:
            if os.path.isfile(file_path):
                os.remove(file_path)
                print(f"Deleted: {file_path}")
        except Exception as e:
            print(f"Error while deleting {file_path}: {e}")
```

Figura B.2: Gestión del directorio de salida de conversión

3. Recorrer todos los archivos XML de la carpeta de anotaciones en formato VOC con el propósito de leer su contenido para su posterior conversión.

```
for xml_file in [f for f in os.listdir(voc_dir) if f.endswith(".xml")]:
    with open(os.path.join(output_dir, xml_file.replace(".xml", ".txt")), "w") as out_file:
        tree = ET.parse(os.path.join(voc_dir, xml_file))
        root = tree.getroot()
```

Figura B.3: Recorrido de archivos XML

4. Extraer informaciones de los archivos XML, obteniendo las dimensiones de ancho y alto de cada imagen. Luego, itera sobre cada anotación de objeto presente, identificando su nombre de clase y asignándole un ID basado en una lista predefinida de clases proporcionada. Se capturan las coordenadas del cuadro delimitador (*bounding box*) y se transforman en coordenadas normalizadas para YOLO: x_center y y_center representan el centro del objeto en relación con las dimensiones de la imagen, mientras que *width* y *height* indican el ancho y la altura del cuadro delimitador normalizados. Finalmente, esta información transformada se escribe en archivos de texto en formato YOLO correspondientes.

```
img_size = root.find("size")
img_width = int(img_size.find("width").text)
img_height = int(img_size.find("height").text)

for obj in root.findall("object"):
    class_name = obj.find("name").text
    if class_name not in class_names:
        continue
    class_id = class_names.index(class_name)

    bbox = obj.find("bndbox")
    x_min = float(bbox.find("xmin").text)
    y_min = float(bbox.find("ymin").text)
    x_max = float(bbox.find("xmax").text)
    y_max = float(bbox.find("ymax").text)

    # Convert VOC bbox format to YOLO format
    x_center = (x_min + x_max) / 2 / img_width
    y_center = (y_min + y_max) / 2 / img_height
    width = (x_max - x_min) / img_width
    height = (y_max - y_min) / img_height

    out_file.write(f"{class_id} {x_center} {y_center} {width} {height}\n")
```

Figura B.4: Escritura de imágenes a cada subconjunto

5. Definir las clases de defectos que se desea detectar, la ruta de directorio de archivos XML en formato VOC, y la ruta del directorio de salida de archivos de texto en formato YOLO.

```
classes = ["missing_hole", "mouse_bite", "open_circuit", "short", "spur", "spurious_copper"]
dir_annotations_yolo = os.path.join(path_dataset, 'YOLOv8Annotations')
dir_annotations = os.path.join(path_dataset, 'Annotations')
convert_voc_to_yolo(dir_annotations, dir_annotations_yolo, classes)
```

Figura B.5: Definición de clases y rutas de directorios

Apéndice C

Parámetros de configuración del entrenamiento

Tabla C.1: Argumentos de configuración del entrenamiento [2]

Argumento	Valor por defecto	Descripción
model	None	Especifica el archivo modelo para el entrenamiento. Acepta una ruta a *.pt* modelo preentrenado o un *.yaml* archivo de configuración. Esencial para definir la estructura del modelo o inicializar los pesos.
data	None	Ruta al archivo de configuración del conjunto de datos (por ejemplo, *coco8.yaml*). Este archivo contiene parámetros específicos del conjunto de datos, incluidas las rutas a los datos de entrenamiento y validación, los nombres de las clases y el número de clases.
epochs	100	Número total de épocas de entrenamiento. Cada época representa una pasada completa por todo el conjunto de datos. Ajustar este valor puede afectar a la duración del entrenamiento y al rendimiento del modelo.
time	None	Tiempo máximo de entrenamiento en horas. Si se establece, anula la opción epochs para que el entrenamiento se detenga automáticamente tras la duración especificada. Útil para situaciones de entrenamiento con limitaciones de tiempo.

Tabla C.1: (Continuación)

Argumento	Valor por defecto	Descripción
patience	100	Número de épocas que hay que esperar sin que mejoren las métricas de validación antes de detener el entrenamiento anticipadamente. Ayuda a evitar el sobreajuste deteniendo el entrenamiento cuando el rendimiento se estabiliza.
batch	16	Tamaño del lote, con tres modos: establecido como un número entero (por ejemplo, batch=16), modo automático para una utilización del 60 % de la memoria de la GPU (batch=-1) o modo automático con una fracción de utilización especificada (batch=0,70).
imgsz	640	Tamaño de imagen para el entrenamiento. Todas las imágenes se redimensionan a esta dimensión antes de introducirlas en el modelo. Afecta a la precisión del modelo y a la complejidad computacional.
save	True	Permite guardar los puntos de control del entrenamiento y los pesos finales del modelo. Útil para reanudar el entrenamiento o el despliegue del modelo.
save_period	-1	Frecuencia con la que se guardan los puntos de control del modelo, especificada en épocas. Un valor de -1 desactiva esta función. Útil para guardar modelos provisionales durante sesiones de entrenamiento largas.
cache	False	Activa el almacenamiento en caché de las imágenes del conjunto de datos en la memoria (True/ram), en disco (disk), o desactívalo (False). Mejora la velocidad de entrenamiento reduciendo la E/S de disco a costa de un mayor uso de memoria.

Tabla C.1: (Continuación)

Argumento	Valor por defecto	Descripción
device	None	Especifica el dispositivo o dispositivos de cálculo para el entrenamiento: una única GPU (device=0), varias GPU (device=0,1), CPU (device=cpu), o MPS para el silicio de Apple (device=mps).
workers	8	Número de subprocesos de trabajo para la carga de datos (por RANK si el entrenamiento es Multi-GPU). Influye en la velocidad de preprocesamiento de datos y alimentación del modelo, especialmente útil en configuraciones multi-GPU.
project	None	Nombre del directorio del proyecto donde se guardan los resultados del entrenamiento. Permite almacenar de forma organizada los distintos experimentos.
name	None	Nombre de la ejecución de entrenamiento. Se utiliza para crear un subdirectorio dentro de la carpeta del proyecto, donde se almacenan los registros de entrenamiento y los resultados.
exist_ok	False	Si es Verdadero, permite sobrescribir un directorio de proyecto/nombre existente. Útil para la experimentación iterativa sin necesidad de borrar manualmente las salidas anteriores.
pretrained	True	Determina si se inicia el entrenamiento a partir de un modelo preentrenado. Puede ser un valor booleano o una cadena de ruta a un modelo específico desde el que cargar los pesos. Mejora la eficacia del entrenamiento y el rendimiento del modelo.

Tabla C.1: (Continuación)

Argumento	Valor por defecto	Descripción
optimizer	'auto'	Elección del optimizador para el entrenamiento. Las opciones incluyen SGD, Adam, AdamW, NAdam, RAdam, RMSProp etc., o auto para una selección automática basada en la configuración del modelo. Afecta a la velocidad de convergencia y a la estabilidad.
verbose	False	Activa la salida detallada durante el entrenamiento, proporcionando registros detallados y actualizaciones del progreso. Útil para depurar y supervisar de cerca el proceso de entrenamiento.
seed	0	Establece la semilla aleatoria para el entrenamiento, garantizando la reproducibilidad de los resultados entre ejecuciones con las mismas configuraciones.
deterministic	True	Fuerza el uso de algoritmos deterministas, lo que garantiza la reproducibilidad, pero puede afectar al rendimiento y la velocidad debido a la restricción de algoritmos no deterministas.
single_cls	False	Trata todas las clases de los conjuntos de datos multiclase como una sola clase durante el entrenamiento. Útil para tareas de clasificación binaria o cuando te centras en la presencia del objeto más que en la clasificación.
rect	False	Permite el entrenamiento rectangular, optimizando la composición del lote para un relleno mínimo. Puede mejorar la eficacia y la velocidad, pero puede afectar a la precisión del modelo.

Tabla C.1: (Continuación)

Argumento	Valor por defecto	Descripción
cos_lr	False	Utiliza un programador de la tasa de aprendizaje coseno, que ajusta la tasa de aprendizaje siguiendo una curva coseno a lo largo de las épocas. Ayuda a gestionar la tasa de aprendizaje para una mejor convergencia.
close_mosaic	10	Desactiva el aumento de datos en mosaico en las últimas N épocas para estabilizar el entrenamiento antes de finalizarlo. El valor 0 desactiva esta función.
resume	False	Reanuda el entrenamiento desde el último punto de control guardado. Carga automáticamente los pesos del modelo, el estado del optimizador y el recuento de épocas, continuando el entrenamiento sin problemas.
amp	True	Permite el entrenamiento Automático de Precisión Mixta (AMP), reduciendo el uso de memoria y posiblemente acelerando el entrenamiento con un impacto mínimo en la precisión.
fraction	1.0	Especifica la fracción del conjunto de datos que se utilizará para el entrenamiento. Permite entrenar con un subconjunto del conjunto de datos completo, útil para experimentos o cuando los recursos son limitados.
profile	False	Permite perfilar las velocidades de ONNX y TensorRT durante el entrenamiento, útil para optimizar el despliegue del modelo.
freeze	None	Congela las N primeras capas del modelo o capas especificadas por índice, reduciendo el número de parámetros entrenables. Útil para el ajuste fino o el aprendizaje por transferencia.

Tabla C.1: (Continuación)

Argumento	Valor por defecto	Descripción
lr0	0.01	Tasa de aprendizaje inicial (es decir SGD=1E-2, Adam=1E-3) . El ajuste de este valor es crucial para el proceso de optimización, ya que influye en la rapidez con que se actualizan las ponderaciones del modelo.
lrf	0.01	Tasa de aprendizaje final como fracción de la tasa inicial = (lr0 * lrf), que se utiliza junto con los programadores para ajustar el ritmo de aprendizaje a lo largo del tiempo.
momentum	0.937	Factor de impulso para SGD o beta1 para optimizadores Adam, que influye en la incorporación de gradientes pasados en la actualización actual.
weight_decay	0.0005	Término de regularización L2, que penaliza los pesos grandes para evitar el sobreajuste.
warmup_epochs	3.0	Número de épocas para el calentamiento de la tasa de aprendizaje, aumentando gradualmente la tasa de aprendizaje desde un valor bajo hasta la tasa de aprendizaje inicial para estabilizar el entrenamiento desde el principio.
warmup_momentum	0.8	Impulso inicial para la fase de calentamiento, ajustándose gradualmente al impulso establecido durante el periodo de calentamiento.
warmup_bias_lr	0.1	Tasa de aprendizaje de los parámetros de sesgo durante la fase de calentamiento, que ayuda a estabilizar el entrenamiento del modelo en las épocas iniciales.
box	7.5	Peso del componente de pérdida de caja en la función de pérdida, que influye en el énfasis que se pone en predecir con precisión las coordenadas de la caja delimitadora.

Tabla C.1: (Continuación)

Argumento	Valor por defecto	Descripción
cls	0.5	Peso de la pérdida de clasificación en la función de pérdida total, que afecta a la importancia de la predicción correcta de la clase en relación con otros componentes.
dfl	1.5	Peso de la pérdida focal de distribución, utilizado en algunas versiones de YOLO para una clasificación de grano fino.
pose	12.0	Peso de la pérdida de pose en los modelos entrenados para la estimación de la pose, lo que influye en la importancia de predecir con precisión los puntos clave de la pose.
kobj	2.0	Ponderación de la pérdida de objetividad del punto clave en los modelos de estimación de la pose, equilibrando la confianza en la detección con la precisión de la pose.
label_smoothing	0.0	Aplica el suavizado de etiquetas, suavizando las etiquetas duras a una mezcla de la etiqueta objetivo y una distribución uniforme sobre las etiquetas, puede mejorar la generalización.
nbs	64	Tamaño nominal del lote para la normalización de la pérdida.
overlap_mask	True	Determina si las máscaras de segmentación deben solaparse durante el entrenamiento, aplicable en tareas de segmentación de instancias.
mask_ratio	4	Relación de reducción de la muestra para las máscaras de segmentación, que afecta a la resolución de las máscaras utilizadas durante el entrenamiento.

Tabla C.1: (Continuación)

Argumento	Valor por defecto	Descripción
dropout	0.0	Tasa de abandono para la regularización en tareas de clasificación, que evita el sobreajuste omitiendo aleatoriamente unidades durante el entrenamiento.
val	True	Activa la validación durante el entrenamiento, lo que permite evaluar periódicamente el rendimiento del modelo en un conjunto de datos distinto.
plots	False	Genera y guarda gráficos de las métricas de entrenamiento y validación, así como ejemplos de predicción, proporcionando una visión visual del rendimiento del modelo y de la progresión del aprendizaje.