UNIVERSITAT POLITÈCNICA DE VALÈNCIA

ESCUELA TÉCNICA SUPERIOR INGENIERÍA INDUSTRIAL VALENCIA

TRABAJO FIN DE MASTER EN INGENIERÍA INDUSTRIAL

DISEÑO, IMPLEMENTACIÓN Y EVALUACIÓN DE UNA RED NEURONAL CONVOLUCIONAL PARA LA DETECCIÓN DE PUNTOS PRINCIPALES EN NARANJAS

AUTOR: ESTHER DEL HOYO DORADO

TUTOR: ANTONIO JOSÉ SÁNCHEZ SALMERÓN

Curso Académico: 2019-20

Agradecimientos

En primer lugar, me gustaría agradecer a mis padres por poner todo vuestro esfuerzo para que haya llegado hasta aquí. A mi hermano Alberto, por ser un ejemplo a seguir y aconsejarme en todo lo posible. A toda mi familia y amigos por el apoyo recibido durante estos años.

También quiero agradecer a mi tutor Antonio, por su inestimable ayuda en la elaboración de este trabajo, por estar siempre disponible y facilitar el desarrollo del trabajo pese a la situación excepcional en la que nos encontramos. A todos los profesores que me han formado en esta etapa y a todos los compañeros que he tenido el placer de encontrar en el camino.

Gracias a todos.

Resumen

Este trabajo consistió en diseñar, implementar y evaluar diferentes arquitecturas de redes neuronales convolucionales aplicadas al control de calidad de naranjas durante la fase de clasificación antes de su encajado. Estas redes neuronales debían detectar los pezones, los culos y los defectos sobre las imágenes de naranjas capturadas en una línea de producción.

Inicialmente, el trabajo se centró en la creación y edición de forma supervisada de un conjunto de imágenes bien etiquetadas. Para ello, se utilizó la herramienta de etiquetado 'datasets.ai2.upv.es/oranges' disponible en el Instituto de Automática e Informática Industrial (ai2). Posteriormente se diseñaron distintas arquitecturas de redes neuronales y se optimizaron cada una de ellas sobre el espacio de hiperparámetros utilizando Python y PyTorch. Finalmente, se evaluaron las distintas arquitecturas propuestas, utilizando como criterios de optimización tanto las tasas de aciertos como los costes temporales.

Palabras clave: Redes neuronales convolucionales, detección.

Resum

Aquest treball ha consistit a dissenyar, implementar i avaluar diferents arquitectures de xarxes neuronals convolucionals aplicades al control de qualitat de taronges durant la fase de classificació abans del seu encaixat. Aquestes xarxes neuronals devien detectar els mugrons, els culs i els defectes sobre les imatges de taronges capturades en una línia de producció.

Inicialment, el treball s'ha centrat en la creació i edició de forma supervisada d'un conjunt d'imatges ben etiquetades. Per aquest fi, s'ha utilitzat l'eina d'etiquetatge 'datasets.ai2.upv.es/-oranges' disponible en l'Institut d'Automàtica i Informàtica Industrial (ai2). Posteriorment es dissenyaren diferents arquitectures de xarxes neuronals i s'optimitzaren cadascuna d'elles sobre l'espai de hiperparámetres utilitzant Python i Pytorch. Finalment, s'han avaluat les diferents arquitectures proposades, utilitzant com a criteris d'optimització tant les taxes d'encerts com els costos temporals.

Paraules clau: Xarxes neuronals convolucionals, detecció.

Abstract

This work consisted of designing, implementing and evaluating different convolutional neural network architectures applied to the quality control of oranges during the classification phase before packing. These neural networks had to detect pedicels, bottoms, and defects on images of oranges captured on a production line.

Initially, work focused on supervised creation and editing of a dataset of well-labeled images. For this, the labeling tool 'datasets.ai2.upv.es/oranges' available at the Institute of Industrial Automation and Informatics (ai2) was used. Subsequently, different neural network architectures were designed and each one was optimized on the hyperparameter space using Python and PyTorch. Finally, the different proposed architectures were evaluated, using both the accuracies and the temporary costs as optimization criteria.

Keywords: Convolutional neural networks, detection.

Índice general

Índice de figuras

Índice de tablas

Parte I

Memoria

Capítulo 1

Introducción

Hoy en día, el campo de la inteligencia artificial y más en concreto el del *Deep Learning* y las redes neuronales está en auge. Esta rama tiene como objetivo implementar algoritmos que permitan que las máquinas aprendan de manera autónoma. Este campo combinado con la visión artificial es de suma importancia en muchas aplicaciones tecnológicas, entre ellas cabe destacar los vehículos autónomos, altavoces inteligentes, traducción automática, reconocimiento facial, detección de enfermedades en medicina, clasificación y detección de imágenes y muchos otros ejemplos.

Este trabajo de fin de máster se centra en implementar y evaluar el uso de las redes neuronales artificales para la detección de objetos, teniendo como objetivo la detección de las diferentes partes de una naranja.

1.0.1 Motivación

El presente proyecto complementa los conocimientos adquiridos en el Máster en Ingeniería Industrial, especialidad en Control de Procesos, Automatización y Robótica, además de permitirme profundizar en un campo novedoso a la vez que llamativo para mí, el del *Deep Learning* y las redes neuronales.

Además, este proyecto surge del deseo de automatización de un proceso hasta ahora manual en la mayoría de las empresas del sector agrario. En los almacenes de naranjas, las naranjas llegan en cajas de plástico que se vacían sobre un transportador de rodillos con ayuda de una volcadora. Este transportador de rodillos lleva la mercancía hasta la lavadora donde con agua y un detergente especial se lavan las naranjas. Los rodillos funcionan como drenajes para la evacuación del agua de lavado hacia el fondo de la lavadora. A continuación, las naranjas siguen avanzando por la línea por el mismo transportador de rodillos hasta llegar a un túnel de secado. Cuando el producto llega al final del túnel se encuentra totalmente seco y en las condiciones adecuadas para ser encerado. Este proceso de encerado consiste en el avance de la fruta bajo una atmósfera de cera relativamente caliente que impregna el producto. Progresivamente las naranjas siguen avanzando y entran en otro túnel de secado ya que tras el encerado quedan

húmedas. Una vez que el producto está totalmente seco pasa a la zona de selección, donde el trabajo es manual. Es en este punto de la línea donde tiene cabida la aplicación propuesta en este trabajo.

Con ayuda de la red neuronal convolucional implementada, las naranjas se clasificarían automáticamente dependiendo de unos estándares de calidad establecidos en función de los defectos detectados e incluso por tamaños mediante el cálculo del ratio de aspecto conocida la posición del culo y el pezón de esta.

Los últimos pasos de la línea de almacenaje consisten en envasar las naranjas en sus respectivas cajas, embalarlas y paletizarlas.

1.0.2 Objetivos

El principal objetivo de este proyecto es la detección y correcta clasificación de las diferentes partes de una naranja, se han distinguido 3 zonas, pezón, correspondiente a la parte superior de la naranja por donde esta cuelga del árbol, culo, parte inferior de la naranja y defectos, incluyendo en esta categoría zonas más oscuras en la naranja, partes más verdes, estrías, manchas y otros defectos posibles.

Para alcanzar el objetivo principal del proyecto se han establecido unos objetivos secundarios, entre ellos se destacan los siguientes:

- Investigación en el campo de las redes neuronales artificiales y el uso de ellas para la detección de objetos.
- Correcto etiquetado del conjunto de imágenes que forman el *dataset*.
- Conocimiento del entorno de trabajo *Google Colab*.
- Comprensión de la librería de aprendizaje automático empleada *PyTorch*.
- Creación y configuración del modelo para detección de objetos en imágenes.
- Entrenamiento del modelo.
- Evaluación de los resultados obtenidos.
- Extracción de conclusiones.

1.0.3 Estructura de la memoria

Esta memoria se ha dividido en siete capítulos principales, además de un apartado bibliográfico, que tratan los siguientes aspectos:

1. Introducción donde se relata la motivación para la elaboración de este trabajo, se detallan los objetivos tanto principales como secundarios y se describe la estructura del contenido de la memoria.
2. Estado del Arte. Se describe el estado del arte en el campo de la inteligencia artificial y más concretamente en el ámbito de las redes neuronales.

3. Redes neuronales convolucionales para detección de objetos. En el siguiente capítulo se explican los fundamentos teóricos en los que está basado el proyecto. Se describen en detalle las redes neuronales convolucionales y los algoritmos empleados para la detección de objetos.

4. Tecnologías y Herramientas. En este capítulo se enumeran las herramientas empleadas, desde el conjunto de datos, es decir, las imágenes de naranjas que forman el *dataset* hasta el lenguaje de programación y el entorno de ejecución empleado.

5. Desarrollo. Se comentan los diferentes criterios de etiquetado establecidos para la creación de las anotaciones en las imágenes, se detalla el fenómeno de transferencia de aprendizaje utilizado en el presente proyecto así como las métricas utilizadas. En el último apartado de este capítulo se explica en detalle el desarrollo del código en Python.

6. Resultados. En este capítulo se analizan los resultados obtenidos durante el proyecto.

7. Conclusiones. Se extraen las conclusiones del trabajo desarrollado y se analizan las posibles mejoras y trabajos futuros.

8. Bibliografía. Se hace referencia a las fuentes consultadas a lo largo del proyecto, para el desarrollo de la parte experimental como de la parte teórica de la memoria.

Además, se ha añadido un apartado con el presupuesto previsto para el proyecto y se ha adjuntado el código desarrollado en Python como Anexo.

Capítulo 2

Estado del arte

En este capítulo se detallan algunos fundamentos teóricos necesarios para la elaboración del proyecto. Se distingue entre los conceptos de inteligencia artificial, aprendizaje automático y aprendizaje profundo y se relata brevemente la evolución histórica en el área de la Inteligencia Artificial.

2.1 Inteligencia Artificial

Hoy en día los términos de Inteligencia Artificial, *Machine Learning* y *Deep Learning* son muy utilizados. Aunque suelen emplearse los términos en inglés, en español sería Aprendizaje Automático y Aprendizaje Profundo. A continuación se relata la diferencia entre ellos.

La inteligencia artificial es la inteligencia llevada a cabo por máquinas, en adelante IA. Es el conjunto de todas las técnicas que permiten a las computadoras imitar a los humanos.

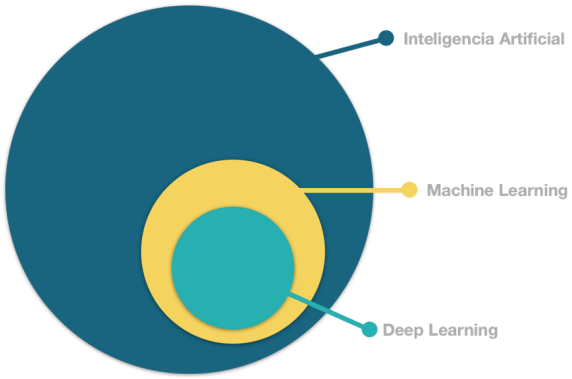


Figura 2.1: IA, ML y DP [1]

2.2 Machine Learning

Un subconjunto de la IA es el aprendizaje automático, *Machine Learning* en la literatura inglesa. Se centra en la capacidad de las máquinas para recibir un conjunto de datos y aprender por sí mismas, cambiando los algoritmos a medida que aprenden más sobre la información que están procesando.

2.3 Deep Learning

El aprendizaje profundo, *Deep Learning* en la literatura inglesa, es un subconjunto del *Machine Learning*.

Dentro del *Deep Learning* se encuentran las redes neuronales artificiales, objeto de este proyecto. Estas redes imitan la conectividad del cerebro humano, clasifican conjuntos de datos y encuentran correlaciones entre ellos. La máquina adquiere nuevo conocimiento sin intervención humana y aplica sus conocimientos a otros conjuntos de datos, cuántos más datos tenga la máquina a su disposición, más precisas serán sus predicciones.

2.4 Historia de la IA hasta las redes neuronales artificiales

En 1936, el inglés Alan Turing, considerado uno de los padres de la computación y precursor de la informática moderna, fue el primero en estudiar el cerebro como una forma de ver el mundo de la computación. En 1950, se publicó un artículo de Turing en la revista Mind titulado "Computing Machinery and Intelligence". En este trabajo se propone un test de inteligencia para máquinas según el cual una máquina presentaría un comportamiento inteligente en la medida en que fuese capaz de mantener una conversación con un humano sin que otra persona pueda distinguir quién es el humano y quién el ordenador.

Aunque el nacimiento de la IA como disciplina de investigación no se remonta hasta 1956, durante una conferencia sobre informática teórica que tuvo lugar en el Dartmouth College de Estados Unidos. Entre los asistentes estaban J. McCarthy, M. Minsky, A. Newell, H. Simon. Estos dos últimos presentaron un trabajo sobre demostración automática de teoremas al que denominaron Logic Theorist. Fue el primer programa de ordenador que emulaba características propias del cerebro humano, por lo que es considerado el primer sistema de inteligencia artificial de la historia.

En 1958 se creó la primera neurona, se llamó Perceptron y fue creada por F. Rosenblatt. Perceptron es un clasificador binario o discriminador lineal, el cual genera una predicción basándose en un algoritmo combinado con el peso de las entradas. Años más tarde, se probó que el Perceptron no era capaz de resolver problemas relativamente fáciles, tales como el aprendizaje de una función no-lineal, lo que provocó un estancamiento en el avance de las redes neuronales durante algunos años.

Tras los primeros trabajos en IA de los años cincuenta, en la década de los sesenta, en parte como respuesta al test de Turing, se produjo el nacimiento de un área conocida como procesado

del lenguaje natural (NLP, Natural Lenguage Processing), una disciplina dedicada a sistemas artificiales capaces de generar frases inteligentes y de mantener conversaciones con humanos.

En los años ochenta empezaron a desarrollarse las primeras aplicaciones comerciales de la IA, fundamentalmente dirigidas a problemas de producción o control de procesos. En 1986 se publicó un artículo sobre el algoritmo de *backpropagation*. En el cual se mostraba experimentalmente cómo calculando el error obtenido en la salida y propagando hacia las capas anteriores, se podía lograr que la red autoajustara sus parámetros para así aprender una representación interna de la información que estaba procesando. En 1989 se crea la primera red neuronal convolucional capaz de extraer características y luego clasificar. Aunque no fue hasta 2012 cuando este tipo de redes se empezaron a usar para la clasificación y localización visual.

El gran desarrollo en el ámbito del *Deep Learning* y las redes neuronales se produce a partir de 2006 gracias al aprovechamiento del poder de las GPUs (*Graphics Proccesing Unit*).

El interés hoy en día es diseñar e implementar sistemas que permitan analizar grandes cantidades de datos de forma rápida y eficiente.

Algunas veces nos interesará caracterizar los datos de forma simplificada para poder realizar un análisis en un espacio de dimensión reducida o para visualizar los datos de forma más eficiente. En otras ocasiones, el objetivo será identificar patrones en los datos para poder clasificar las observaciones en diferentes clases que resulten útiles para tomar decisiones respecto un determinado problema. Por último, en muchos casos es necesario realizar búsquedas entre una cantidad de datos u optimizar una determinada función de coste, por lo que también será necesario conocer métodos de búsqueda y optimización [14] [15] [16] [17].

2.5 Antecedentes de visión artificial

El aumento de la competitividad es un requerimiento primordial en la situación económica actual. Un factor clave para mejorar la competitividad es aumentar la productividad incorporando nuevos sistemas de inspección que permitan la automatización de diferentes procesos. Los sistemas de inspección visual automática, basados en visión por computador, han demostrado ser una herramienta fundamental para mejorar los procesos. Estos sistemas de visión permiten la inspección continua, evitando fatigas y distracciones, y facilitando la cuantificación de las variables de calidad en prácticamente el 100 % de la producción. Esto se traduce, no sólo en una mejora de la calidad final de los productos, sino también en un ahorro en términos económicos y medioambientales.

Durante las últimas décadas se han podido resolver multitud de aplicaciones mediante la implantación de sistemas de inspección 2D en la industria utilizando técnicas de visión por computador tradicionales. El principal problema a resolver en estos sistemas ha sido seleccionar un conjunto mínimo de características discriminante para clasificar los objetos de interés, además de la gran variabilidad de las imágenes que puede dificultar la segmentación de estos objetos de interés. Para resolver este problema se han propuesto algunas técnicas robustas de segmentación que intentan absorber dicha variabilidad [18] [19]. Además, también se han propuesto infinidad de soluciones de seguimiento de objetos, como por ejemplo [20].

Por otro lado, los enormes avances tecnológicos producidos en las cámaras lineales, los escáneres 3D [21] [22] [23] [24] [25] y los sensores hiperespectrales [26] [27] [28] están permitiendo resolver algunas aplicaciones complejas en el sector industrial que hace unos años eran impensables. Estas nuevas tecnologías aplicadas sobre todo al sector agroalimentario y cosmético, se están empezando a introducir en la industria.

Además, los últimos avances producidos en la aplicación de técnicas de detección y seguimiento de objetos deformables [29] [30] [31] [32] [33] [34] basadas en redes neuronales convolucionales; están permitiendo empezar a automatizar algunas aplicaciones de detección complejas.

La clasificación de naranjas es un problema complejo debido a la gran variabilidad de defectos que se deben detectar, y a los altos requerimientos temporales de una línea de calibrado de naranjas dentro de un almacén. En este trabajo se plantean nuevas técnicas de imagen basadas en redes neuronales convolucionales para la detección de los diferentes puntos de interés de las naranjas.

Capítulo 3

Redes neuronales convolucionales para detección de objetos

En este capítulo se presentan los fundamentos teóricos en los que se basa este proyecto. Se detallará en qué consiste la unidad básica de procesamiento de una red neuronal, conocida como neurona, los tipos de estructuras de las redes neuronales, así como las redes neuronales convolucionales, ampliamente utilizadas en el campo de la detección de objetos.

3.1 La neurona

Una red neuronal consta de unidades básicas de procesamiento, las neuronas y conexiones ponderadas entre esas neuronas [35].

Reciben el nombre de neuronas artificiales por su similitud a las neuronas biológicas. De forma similar a una neurona biológica, estas neuronas tienen conexiones de entrada a través de las que reciben estímulos externos, las variables de entrada. Con estos valores la neurona realizará un cálculo interno y generará un valor de salida. Internamente, la neurona utiliza todos los valores de entrada para realizar una suma ponderada de ellos. La ponderación de cada una de las entradas viene dada por el peso que se le asigna a cada una de las conexiones de entrada. Es decir, cada conexión que llega a la neurona tendrá asociado un valor que definirá con qué intensidad cada variable de entrada afecta a la salida de la neurona.

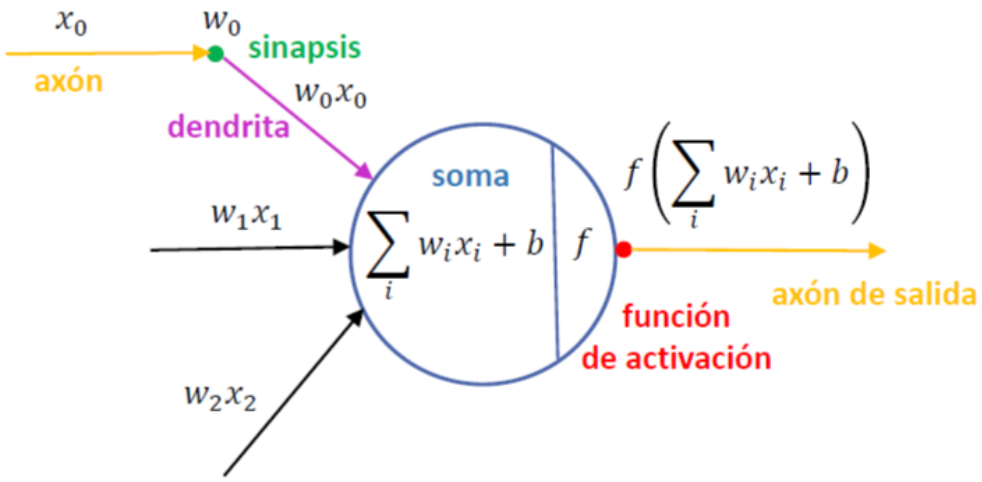


Figura 3.1: Estructura de una neurona artificial [2]

Como se observa en la figura 3.1, las neuronas constan de los siguientes componentes:

- Variables de entrada, x_i.
- Pesos, uno por cada variable de entrada, w_i. Estos pesos determinarán la importancia de cada conexión.
- Sesgo o *bias* en la literatura inglesa. Es el término independiente de la función. El sesgo indicará la facilidad que tiene la neurona de activarse. La neurona permite obtener valores de salida continuos, no binarios. Para que la salida sea binaria se comparará con un determinado umbral, siendo 0 si el valor de la salida es menor o igual que el del umbral y 1 si es mayor que él. El valor del sesgo será el opuesto al umbral.
- Función de propagación, consiste en la suma ponderada de las variables de entrada con sus respectivos pesos y el término de sesgo. En el caso más simple y general esta función de propagación es lineal y tiene la siguiente fórmula matemática:

$$y = \sum_{i=1}^{N} w_i x_i + b \tag{3.1}$$

- Función de activación, recibe la salida de la función de propagación y determinará si se activa la salida de la neurona, $z = f_{act}(y)$. Las más comunes son la sigmoide, la tangente hiperbólica y la función Relu (Unidad Lineal Rectificada) representadas en la figura 3.2. Gracias a ella tiene sentido encadenar varias neuronas.

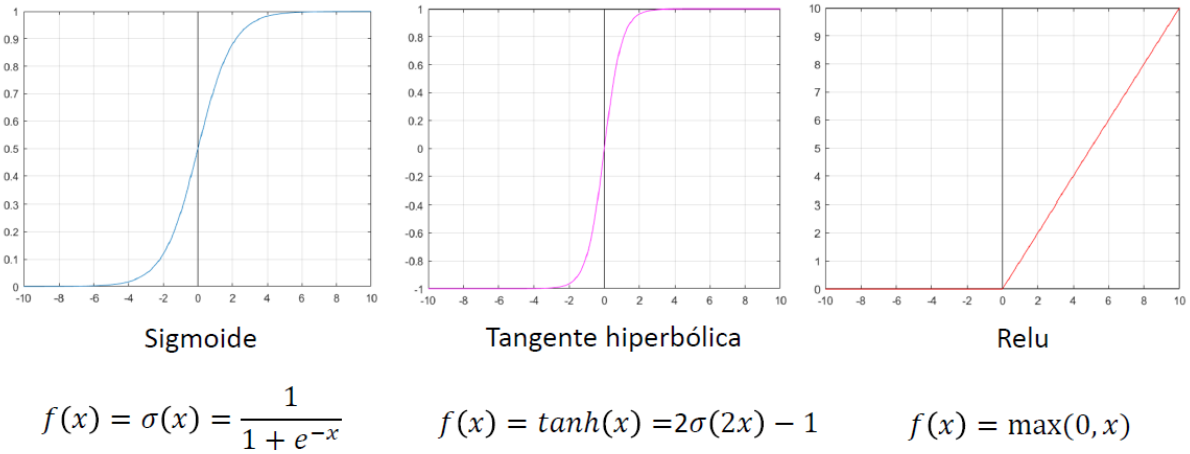


Figura 3.2: Funciones de activación [2]

La función de activación Sigmoide hace que los valores muy grandes se saturen en 1 y los valores muy pequeños en 0. Se emplea en modelos en los que tenemos que predecir la probabilidad como una salida en el rango de 0 a 1.

La función de activación Tanh es como la función Sigmoide pero en el rango de -1 a 1.

La función RELU se comporta como una función lineal cuando es positiva y constante a 0 cuando el valor de entrada es negativo. Esta función tiene algunas ventajas sobre el resto, es menos costosa y converge más rápidamente que Sigmoide o Tanh.

En los problemas de clasificación con múltiples opciones la función de activación empleada es la función Softmax. Esta función se emplea como capa final en los clasificadores basados en redes neuronales, como en el clasificador de Faster R-CNN que se verá en el apartado 3.6.3. Esta función devuelve las probabilidades de cada clase sumando el conjunto de ellas 1 y viene dada por la ecuación 3.2.

$$f(x_i) = \frac{e^{x_i}}{\sum_{j=i}^{k} e^{x_j}} \tag{3.2}$$

Siendo x un vector de entrada de tamaño igual al número de neuronas de la capa y k el número de clases diferentes de objetos.

3.2 Estructuras de las redes neuronales

Las redes neuronales están formadas por neuronas interconectadas, estas neuronas pueden agruparse siguiendo diferentes estructuras. Las más comunes son las redes recurrentes y las redes *feedforward*, estas últimas son las más utilizadas.

Redes *feedforward*

Esta estructura consiste en una capa de entrada, una serie de capas ocultas y una capa de salida donde la información se mueve en una única dirección, de la capa de entrada hacia la de salida.

La capa de entrada en realidad no es una capa compuesta de neuronas, sino un forma de representar las entradas a la primera capa oculta, la cual es la primera capa que consta de neuronas.

Dentro de las redes *feedforward* dependiendo del número de capas, podemos distinguir la red neuronal monocapa o perceptrón simple y la red neuronal multicapa o perceptrón multicapa.

La red neuronal monocapa se corresponde con la red neuronal más simple, las entradas están conectadas directamente a las salidas donde se realizan los diferentes cálculos según lo explicado en el apartado 3.1 [4].

La representación gráfica de una red neuronal monocapa se puede ver en la figura 3.3.

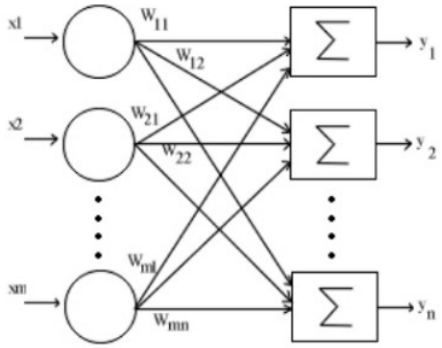


Figura 3.3: Red neuronal monocapa [3]

La red neuronal multicapa es una generalización de la red neuronal monocapa, la diferencia reside en que mientras la red neuronal monocapa está compuesta por una capa de neuronas de entrada y una capa de neuronas de salida, esta dispone de un conjunto de capas intermedias, llamadas capas ocultas, entre la capa de entrada y la de salida [4].

La representación gráfica de una red neuronal multicapa se puede ver en la figura 3.4.

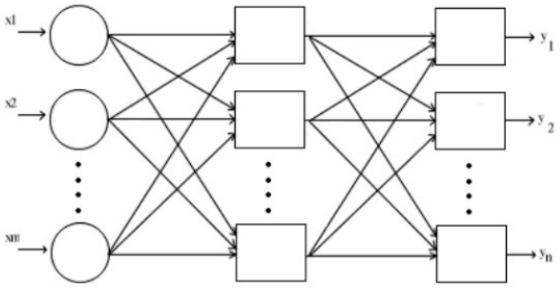


Figura 3.4: Red neuronal multicapa [3]

Red Neuronal Convolucional (CNN)

La principal diferencia de la red neuronal convolucional con el perceptrón multicapa viene en que cada neurona no se una con todas y cada una de las capas siguientes, sino que solo con un subgrupo de ellas, con esto se consigue reducir el número de neuronas necesarias y la complejidad computacional necesaria para su ejecución.

La representación gráfica puede verse en la figura 3.5.

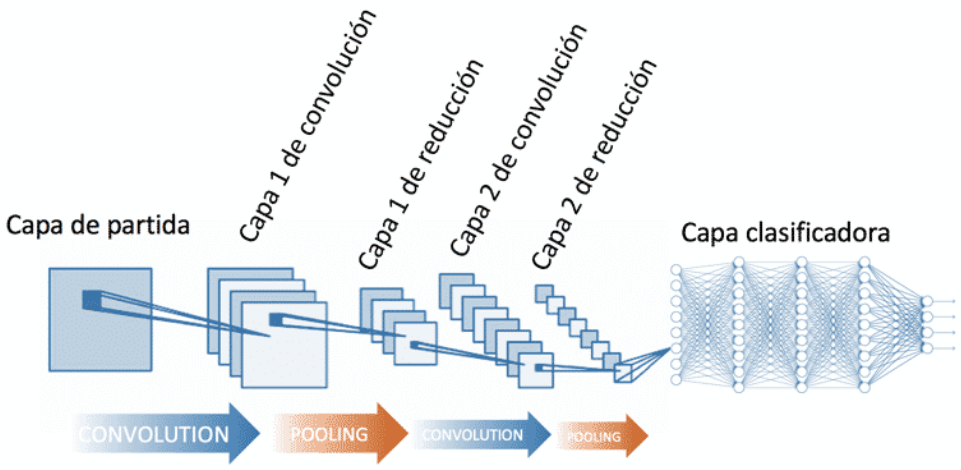


Figura 3.5: Red neuronal convolucional [4]

Red Neuronal Recurrente

Las redes neuronales recurrentes permiten conexiones arbitrarias entre las neuronas, incluso pudiendo crear ciclos, con esto se consigue crear la temporalidad permitiendo que la red tenga memoria [4].

La representación gráfica puede verse en la figura 3.6.

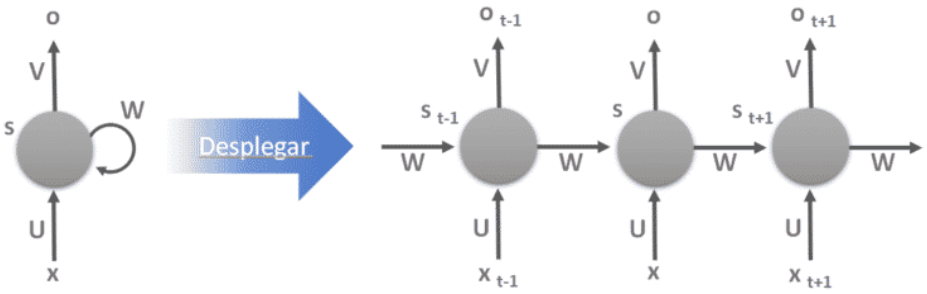


Figura 3.6: Red neuronal recurrente [4]

3.3 Algoritmos de entrenamiento

El entrenamiento de una red neuronal puede entenderse como un problema de optimización. El objetivo de dicho problema es encontrar los pesos y biases que minimizan la función de coste, estos parámetros serán ajustados mediante entrenamiento.

3.3.1 Función de coste

La función de coste indica lo cerca que está la predicción obtenida de la solución real. Normalmente, las funciones de coste se calculan haciendo una media de los costes de los N ejemplos de entrenamiento según la ecuación 3.3.

$$C(W, b) = \frac{1}{N} \sum_{i=1}^{N} C_i(W, b) \tag{3.3}$$

Donde W y b representan los conjuntos totales de pesos y biases de la red respectivamente. C_i representa la función de coste para la instancia i del conjunto de entrenamiento. Si la función de coste tiene un valor próximo a 0 la estimación será buena, mientras que si el valor de la función de coste es alto la estimación será mala.

La función de coste empleada en el proyecto es la llamada *Cross-Entropy* definida por la ecuación 3.4.

$$C = -\sum_{i}^{M} y_i log(Y_i) \tag{3.4}$$

donde Y_i es el valor previsto, y_i es el valor obtenido, M es el número de clases totales e i es una clase concreta [36].

3.3.2 Descenso de gradiente

El descenso de gradiente es un método iterativo para encontrar los mínimos locales de una función. En el campo de las redes neuronales, el descenso de gradiente es usado para minimizar la función de coste actualizando los pesos de la red y disminuyendo la desviación de la salida. Para ello es de vital importancia el ratio de aprendizaje o *learning rate*, este define cuánto afecta el gradiente a la actualización de nuestros parámetros en cada iteración, o lo que es lo mismo, cuánto se avanza en cada paso. Un *learning rate* muy pequeño resultará en una convergencia extremadamente lenta y, por el contrario, si el valor es muy alto el algoritmo no llegará a converger.

En este proyecto se ha utilizado una técnica llamada *learning rate decay*, consiste en ir disminuyendo la tasa de aprendizaje a medida que avanza el entrenamiento. De esta forma se empieza con *learning rates* grandes que permiten que el aprendizaje sea más rápido y a medida que se van ajustando los parámetros se disminuye el *learning rate* para que sea más fácil encontrar el mínimo de la función.

En primer lugar, se inicializan los parámetros de la función que se desea minimizar de forma aleatoria, $C(v) = C(v_1, v_2, ..., v_N)$, siendo N el número de variables. El gradiente en el punto v_0 se calcula según la ecuación 3.5.

$$\nabla C(v_0) = (\frac{\partial C}{\partial v_1}, \frac{\partial C}{\partial v_2}, ..., \frac{\partial C}{\partial v_N}) \tag{3.5}$$

Para avanzar en busca del mínimo, el siguiente punto se calcula según la ecuación 3.6.

$$v_{t+1} = v_t - \eta \nabla C(v_t) \tag{3.6}$$

Siendo v el vector de parámetros (pesos y biases), $C(v)$ la función de coste y η el *learning rate*. Con una elección adecuada de este valor se comprueba que el nuevo valor del coste será menor al anterior, acercándonos al óptimo.

La actualización de los pesos y biases de la red se hacen según las ecuaciones 3.7 y 3.8.

$$w_{k_{t+1}} = w_{k_t} - \eta \frac{\partial C}{\partial w_{k_t}} \tag{3.7}$$

$$b_{k_{t+1}} = b_{k_t} - \eta \frac{\partial C}{\partial b_{k_t}} \tag{3.8}$$

A pesar de que este algoritmo es capaz de converger en un mínimo presenta algunos problemas:

1. Solo permite encontrar mínimos locales. Lo que nos interesa es obtener el mínimo global y mediante este algoritmo, dependiendo de la inicialización de las variables puede que nunca lleguemos a él.

2. Calcular el gradiente de la función de coste puede llegar a ser complicado debido a la gran cantidad de parámetros el tiempo de actualización puede ser muy alto.

Debido a estos problemas, se suele emplear una variante de este algoritmo, el descenso de gradiente estocástico o *SGD* [37].

3.3.3 Descenso de gradiente estocástico

Esta variante del algoritmo de descenso de gradiente divide el conjunto de datos de entrenamiento en M conjuntos iguales elegidos aleatoriamente, estos subconjuntos se denominan *mini-batches*.

Para cada *mini-batch* se aproxima el valor de la función de coste según la ecuación 3.9.

$$C(W, b) = \frac{1}{N} \sum_{i=1}^{N} C_i(W, b) \approx \frac{1}{M} \sum_{j=1}^{M} C_j(W, b) \tag{3.9}$$

Donde C_j es el coste asociado a cada ejemplo dentro de un *mini-batch*. Se calcula el gradiente para la aproximación de la ecuación 3.9 y se actualizan los pesos y biases para cada *mini-batch* según las ecuaciones 3.10 y 3.11.

$$w_{k_{t+1}} = w_{k_t} - \eta \frac{1}{M} \sum_{j=1}^{M} \frac{\partial C_j}{\partial w_{k_t}} \approx w_{k_t} - \eta \frac{\partial C}{\partial w_{k_t}} \tag{3.10}$$

$$b_{k_{t+1}} = b_{k_t} - \eta \frac{1}{M} \sum_{j=1}^{M} \frac{\partial C_j}{\partial b_{k_t}} \approx b_{k_t} - \eta \frac{\partial C}{\partial b_{k_t}} \tag{3.11}$$

La principal ventaja de este método respecto al descenso por gradiente es su mayor rapidez. Mientras que en el algoritmo de descenso de gradiente los pesos se actualizaban una vez para cada dato de entrenamiento, en el descenso de gradiente estocástico los pesos se actualizan para cada *mini-batch* del conjunto de datos de entrenamiento. Además, calcular una aproximación del gradiente y ajustar los parámetros escogiendo subconjuntos aleatoriamente hace que el algoritmo filtre parte del ruido y se avance en la dirección correcta [37].

Uno de los problemas que pueden darse es que el optimizador llegue a un mínimo local y se quede atascado en él. Para evitarlo se utiliza un hiperparámetro llamado *momentum*, se trata de una constante entre 0 y 1 que pondera los gradientes anteriores indicando cual es la contribución del gradiente de la iteración anterior a la actual. El gradiente actual se multiplicará por la tasa de aprendizaje η y el valor de la actualización anterior por el momentum γ. Conseguimos así que el avance sea más rápido cuando nos movemos en la dirección correcta y las posibles oscilaciones se atenúen cuando se rebote en la función de coste. Ambas características hacen que la convergencia se produzca más rápido [38].

En el presente proyecto se ha utilizado este optimizador acompañado del parámetro *momentum*.

3.3.4 Propagación hacia atrás

Las redes neuronales actualizan sus pesos a medida que la red va aprendiendo en el proceso de entrenamiento. Este proceso se denomina propagación hacia atrás o *backpropagation*. El objetivo del algoritmo es calcular las derivadas parciales $\frac{\partial C}{\partial w}$ y $\frac{\partial C}{\partial b}$ de las ecuaciones 3.10 y 3.11.

Para realizar este cálculo debemos introducir la función de error δ de cada neurona de la red que representa cuánto varía la función de coste correspondiente a un ejemplo de entrenamiento al cambiar la entrada a su función de activación.

$$\delta_j^l = \frac{\partial C_i}{\partial y_j^l} \tag{3.12}$$

Donde δ_j^l es el error correspondiente a la neurona j de la capa l, suele conocerse como gradiente local, C_i es la función de coste asociada a un ejemplo de entrenamiento i e y_j^l es la salida de la función de propagación de la neurona j de la capa l que es por tanto la entrada a la función de activación de esa misma neurona. Siendo $z_j^l = f_{act}(y_j^l)$ la ecuación 3.12 quedaría:

$$\delta_j^l = \frac{\partial C_i}{\partial y_j^l} = \frac{\partial C_i}{\partial z_j^l}\frac{\partial z_j^l}{\partial y_j^l} = \frac{\partial C_i}{\partial z_j^l} f_{act}^{'}(y_j^l) \tag{3.13}$$

Para continuar con el desarrollo necesitamos expresar la ecuación 3.13 en forma matricial. Así, el error de todas las neuronas de una capa queda como:

$$\delta^l = \nabla_z C_i \odot f_{act}^{'}(y^l) \tag{3.14}$$

$\nabla_z C_i$ representa el conjunto de derivadas parciales de la función de coste respecto a la salida de todas las neuronas de una capa, $f_{act}^{'}(y^l)$ es un vector que contiene el valor de $f_{act}^{'}(y_j^l)$ para cada neurona j de la capa l y $\odot$ representa el producto de Hadamard o producto elemento a elemento.

El algoritmo *backpropagation* se basa en propagar el error de las capas más cercanas a la salida de la red hacia las neuronas de las capas iniciales. Por ello, el primer paso es calcular el error en la capa de salida. Suponiendo que $l \in [1, L]$ y tomando como función de coste la del error cuadrático (ecuación 3.15), para obtener el error de las neuronas de la capa de salida primero debe calcularse el valor de $\nabla_z C_i$, agrupando las derivadas parciales de cada neurona de la capa, las cuales se calculan según la ecuación 3.16.

$$C(W, b) = \frac{1}{N}\sum_{i=1}^{N}\frac{1}{2}\left\|t_i - o_i\right\|^2 \tag{3.15}$$

$$C_i = \frac{\left\|t_i - o_i\right\|^2}{2} \Rightarrow \frac{\partial C_i}{\partial z_j^L} = z_j^L - t_{ij} \tag{3.16}$$

donde t_{ij} es la salida deseada de la red.

Una vez que conocemos el error en la capa de salida es posible calcular el error en el resto de neuronas del resto de capas utilizando la información de los errores de la siguiente capa. La ecuación 3.17 muestra cómo calcular el vector de errores de la capa l en función de los errores de la capa $l + 1$.

$$\delta^l = \nabla_z C_i \odot f_{act}^{'}(y^l) = (w^{l+1}\delta^{l+1}) \odot f_{act}^{'}(y^l) \tag{3.17}$$

Se define una matriz de pesos w^l para cada capa, de esta forma, w^{l+1} es la matriz de pesos de la capa $l + 1$. Para demostrar esta igualdad hay que representar el error de una neurona de la capa l en función de los errores de las neuronas de la capa $l + 1$. Teniendo en cuenta que existe dependencia entre todas las salidas y_k^{l+1}, $k \in [1, K]$ e y_j^l, aplicando la regla de la cadena la ecuación 3.13 puede escribirse como:

$$\delta_j^l = \frac{\partial C_i}{\partial y_j^l} = \sum \frac{\partial C_i}{\partial y_k^{l+1}}\frac{\partial y_k^{l+1}}{\partial y_j^l} = \sum \frac{\partial y_k^{l+1}}{\partial y_j^l}\delta_k^{l+1} \tag{3.18}$$

Para calcular $\frac{\partial y_k^{l+1}}{\partial y_j^l}$ es necesario representar y_k^{l+1} en función de y_j^l. Según la ecuación 3.1 se puede escribir como:

$$y_k^{l+1} = \sum_j (w_{kj}^{l+1} z_j^l + b_k^{l+1}) = \sum_j w_{kj}^{l+1} f_{act}(y_j^l) + b_k^{l+1} \tag{3.19}$$

Donde w_{kj}^{l+1} es el peso de la conexión desde la neurona j de la capa l hasta la neurona k de la capa $l+1$ y ocupará la posición correspondiente a la fila k y columna j dentro de la matriz. De esta forma:

$$\frac{\partial y_k^{l+1}}{\partial y_j^l} = w_{kj}^{l+1} f_{act}^{'}(y_j^l) \tag{3.20}$$

La ecuación de δ_j^l puede escribirse como:

$$\delta_j^l = \sum_k w_{kj}^{l+1} \delta_k^{l+1} f_{act}^{'}(y_j^l) \tag{3.21}$$

Por último, al poder calcular los errores para cada neurona es posible calcular los valores de las derivadas parciales necesarias para las actualizaciones de los pesos y los biases de las ecuaciones 3.10 y 3.11.

En el caso de los biases, teniendo en cuenta que $y_j^l = \sum_k w_{jk}^l z_k^{l-1} + b_j^l$ se obtiene:

$$\frac{\partial C_i}{\partial b_j^l} = \frac{\partial C_i}{\partial y_j^l} \frac{\partial y_j^l}{\partial b_j^l} = \delta_j^l \tag{3.22}$$

El vector de biases b^l agrupa los biases de cada neurona de la capa l.

Y en el caso de los pesos:

$$\frac{\partial C_i}{\partial w_{jk}^l} = \frac{\partial C_i}{\partial y_j^l} \frac{\partial y_j^l}{\partial w_{jk}^l} = z_k^{l-1} \delta_j^l \tag{3.23}$$

Para la propagación del error hacia atrás hay que calcular la derivada de la función de activación, por ello, estas deben ser derivables [2] [37] [39].

3.3.5 Regularización o dropout

Uno de los problemas más comunes en las redes neuronales profundas es el sobreajuste u *overfitting*. Ocurre cuando el modelo se ajusta tanto a los datos de entrenamiento que no es capaz de generalizar para los datos de test. El objetivo de los modelos de aprendizaje automático es el de obtener patrones de los datos de entrenamiento que luego sean extrapolables a nuevos datos.

El *overfitting* ocurre cuando un modelo se entrena demasiado o con datos anómalos que hace que el algoritmo aprenda patrones que no son generales. Mientras que el *underfitting* es lo contrario, se produce cuando el modelo se entrena con pocos datos o con un número demasiado pequeño de los parámetros del modelo.

Para detectar tanto el *underfitting* como el *overfitting* se divide el conjunto de datos de entrada para entrenamiento en dos subconjuntos, uno para entrenamiento y otro para test [40].

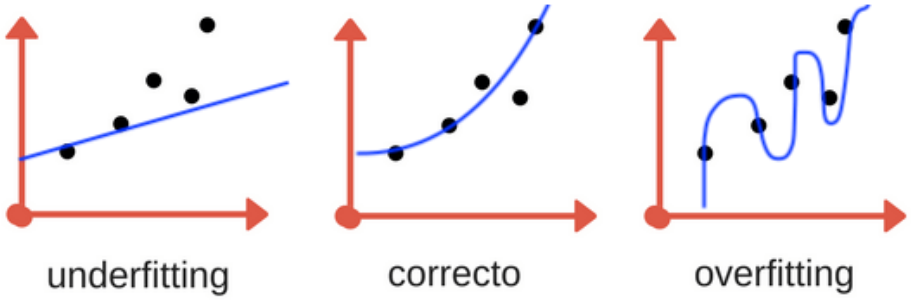


Figura 3.7: Underfitting y Overfitting [5]

La regularización o *dropout* es un método para la prevención del sobreajuste en redes profundas que consiste en suprimir aleatoriamente un porcentaje de neuronas y sus conexiones de la red neuronal durante el entrenamiento. Es decir, la red activa aleatoriamente unas neuronas mientras que otras permanecen desactivadas para un *batch* durante las etapas de *forward* y *backward propagation*, luego se reactivan y se vuelven a desactivar aleatoriamente para el siguiente *batch* [2].

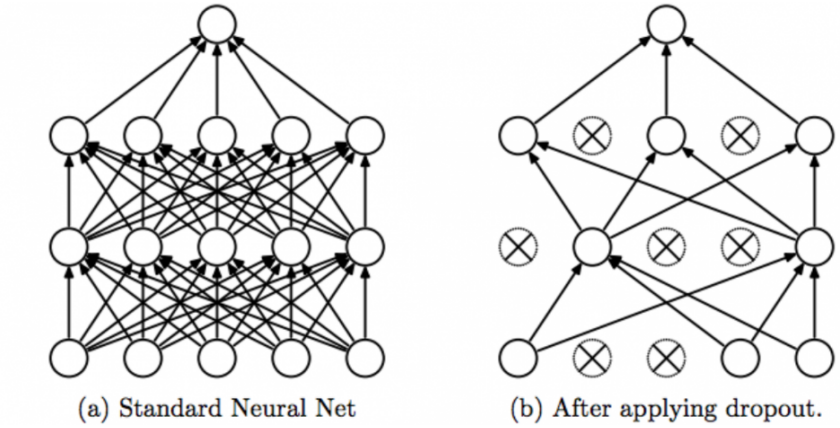


Figura 3.8: Red antes y después de aplicar *dropout* [6]

El *dropout* aleatorio impide que las neuronas de la red se adapten entre ellas al evitar que una unidad particular sea determinante.

3.3.6 Normalización del batch

La normalización en lotes consiste en añadir un paso extra entre las neuronas y la función de activación para así normalizar las activaciones de salida. La normalización se hace usando la media y la varianza de todo el conjunto de entrenamiento excepto en el caso de que se emplee el descenso de gradiente estocástico que se usará la media y la varianza de cada mini-lote de entrada.

La normalización por lotes es una técnica de ayuda al entrenamiento que ayuda a que la convergencia hacia el mínimo global se produzca más rápidamente.

Aplicando el parámetro *momentum* cuando se introduce un nuevo *mini-batch* de entrada se usan una media y una desviación similares a las de la iteración anterior. Gracias a esto se consigue reducir el sobreajuste [41].

3.3.7 Data augmentation

El aumento de datos consiste en aplicar transformaciones sobre las entradas originales, obteniendo muestras ligeramente diferentes. Esta técnica es ampliamente utilizada en el campo de la visión donde una imagen de entrada será procesada por la red neuronal tantas veces como épocas se ejecute el entrenamiento. Esto puede provocar que la red acabe memorizando la imagen si se entrena demasiado. Una forma de solventar este problema es aplicar transformaciones de forma aleatoria cada vez que se vuelva a introducir la imagen a la red.

Así, se dispondrá de más información para el entrenamiento sin incrementar el tiempo, consiguiendo que la red generalice mejor y no sobreajuste [41].

Una de las técnicas más empleadas para *Data augmentation* es la llamada *Horizontal flip* que consiste en voltear una imagen horizontalmente según se ve en la figura 3.9 [42].

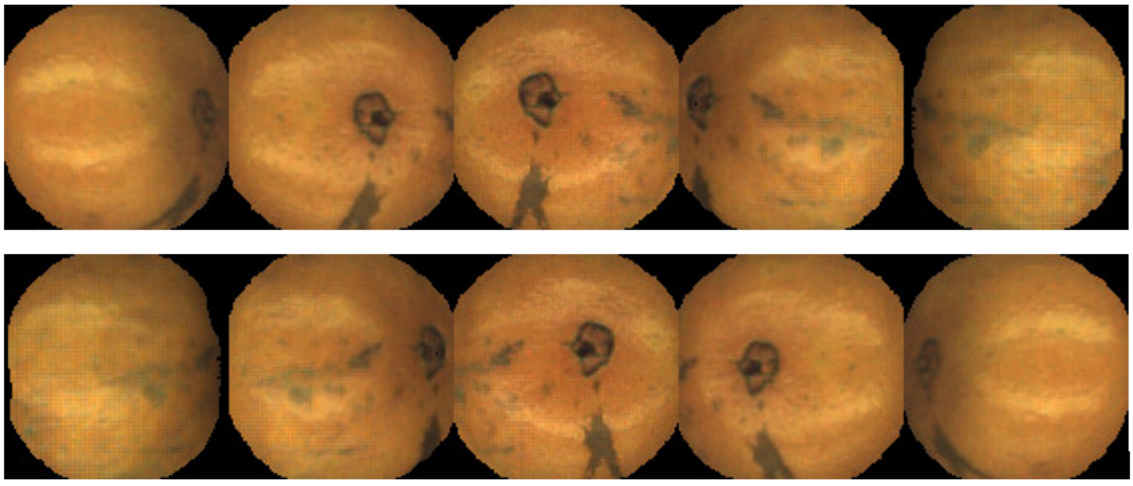


Figura 3.9: Horizontal flip

De esta manera, la red percibe la imagen como una totalmente nueva y disminuye la posibilidad de que la red aprenda orientaciones obligando a que aprenda las características deseadas. En el presente proyecto no se ha realizado aumento de datos ya que los resultados obtenidos sin su aplicación son buenos.

3.4 Redes neuronales convolucionales

Las redes neuronales convolucionales (*Convolutional Neural Networks o CNN*) son un algoritmo de *Deep Learning* que está diseñado para trabajar con imágenes, tomando estas como entrada, asignándole pesos a ciertos elementos en la imagen para así poder diferenciar unos de otros. Las redes convolucionales contienen varias capas ocultas (capas que se encuentran entre la capa de entrada y la capa de salida), en las primeras capas se extraen características con más detalle como pueden ser los bordes o los colores, mientras que en las capas más profundas se extraen características cada vez más abstractas, es decir, lo que se quiere detectar en sí, en el caso de aplicación concreto estaríamos hablando de los defectos, culos o pezones. Las tareas comunes de este tipo de redes son la detección o categorización de objetos, clasificación de escenas y clasificación de imágenes en general [43].

En este tipo de redes las neuronas de las capas que la componen no se unen con todas las neuronas de la capa siguiente, sino sólo con alguna de ellas. Esto simplifica el aprendizaje de la red y reduce los costes computacionales [4].

Los algoritmos de detección de objetos distinguen dos fases: la extracción de características de la imagen y la búsqueda de objetos basada en esas características para su clasificación. Las CNN realizan estas dos fases de forma paralela, facilitando el aprendizaje de la red.

Estas redes pueden ser entrenadas desde cero con un amplio conjunto de imágenes etiquetadas mediante el cual la red aprenderá las características de los datos etiquetados. Esto puede tener un alto coste computacional y el tiempo de entrenamiento puede llegar a ser muy elevado.

Sin embargo, en este proyecto se utilizará la transferencia de aprendizaje. Este proceso consiste en el ajuste detallado de un modelo previamente entrenado. Se comienza con una red previamente entrenada y se le proporcionan datos nuevos con clases desconocidas para la red. Gracias a este método el tiempo de entrenamiento se ve reducido notablemente.

La estructura de las CNN, como se puede ver en la figura 3.5, se compone de tres tipos de capas, capa de convolución, capa de reducción o *pooling* y capa totalmente conectada o *fully connected*. Las neuronas que componen una CNN están especializadas en dos operaciones, la operación de convolución y la operación de reducción o *pooling*.

3.4.1 Convolución

La red toma como entrada una imagen y la trata como una matriz de dimensiones *número de píxeles x número de píxeles x número de canales de color*. Si la imagen es en color el número de canales será 3, *RGB*.

Para realizar la convolución se necesita un filtro que se encargará de extraer las características de la imagen, este filtro se denomina *Kernel*. En el caso de que la imagen sea *RGB* se tendrán 3 *kernels* del mismo tamaño que se sumarán para obtener una imagen de salida.

La convolución consiste en desplazar el *Kernel* por cada uno de los elementos de la imagen original y realizar el producto escalar entre la matriz y el filtro obteniendo así el mapa de características.

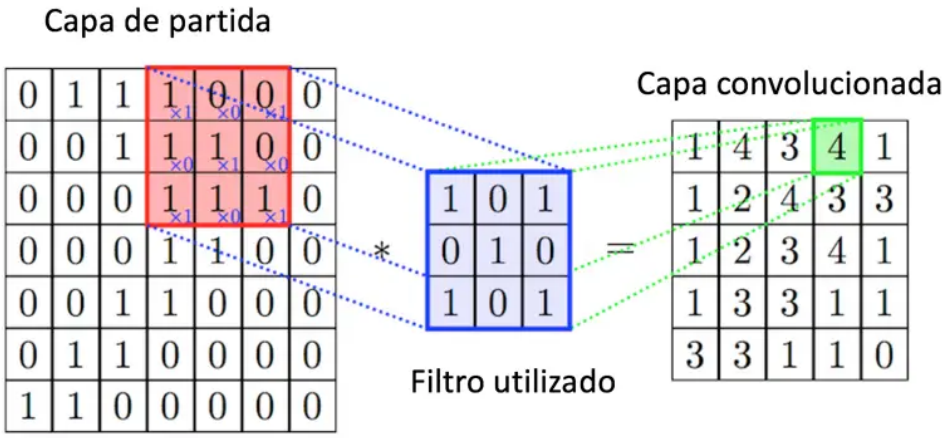


Figura 3.10: Convolución de un *kernel* sobre una imagen, tamaño de salto 1 píxel y *kernel* de 3x3 [7]

Como se observa en la figura 3.10 la salida no tiene el mismo tamaño que la entrada.

Existen varios parámetros que nos permiten controlar las dimensiones del mapa de características de salida:

- El tamaño del *kernel* así como el número de ellos, este parámetro indicará la profundidad de la salida. Por ejemplo, si tenemos 6 filtros, obtendremos 6 mapas de activación separados.
- El parámetro *stride*: Cuando la convolución del *kernel* con la imagen no se realiza en todos los píxeles sino que se hace cada cierto número de ellos. Este paso se denomina *stride* y cuanto mayor sea más pequeña será la salida.
- Zero padding: Consite en agregar píxeles de valor cero alrededor de la imagen original, de esta forma, el mapa de características podrá tener el mismo tamaño que la imagen de entrada [2].

A medida que se entrena la red, los valores de los *kernels* o lo que es lo mismo, los pesos de las neuronas se van actualizando para disminuir el error en la predicción.

Después de aplicar la convolución se le aplica a los mapas de características una función de activación [7].

3.4.2 Pooling

En las capas de reducción o *pooling* se disminuyen las dimensiones espaciales del mapa de características, esto no afecta a la dimensión de profundidad. El objetivo es hacer una simplificación de la información y crear una versión condensada de las características más importantes. La forma más común se conoce como *max pooling*, conserva el valor máximo de una región fija del mapa de características [7] [8].

Se define un tamaño de ventana, en el caso de la figura 3.11 2x2, está ventana va recorriendo el mapa de características y se queda con el valor mayor en cada paso. El tamaño de los datos se reduce por un factor igual al tamaño de la ventana de muestra, como se observa en la figura 3.11 pasamos de 16 a 4 datos.

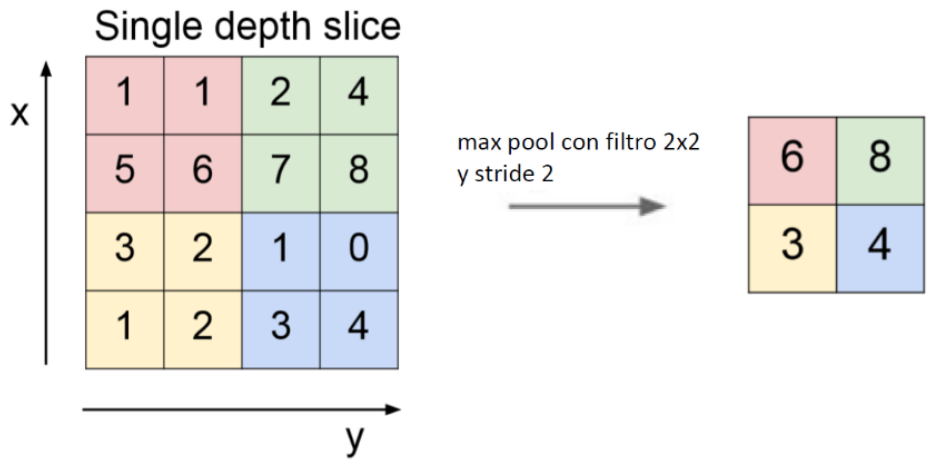


Figura 3.11: *Max Pooling* [8]

El *pooling* se utiliza para reducir la dimensionalidad espacial de los datos anteriores además de conseguir que el sistema sea invariante a pequeñas traslaciones de la entrada, es decir, ante pequeños cambios en la posición de la imagen [44].

3.4.3 Capa totalmente conectada o fully connected

Después de varias capas sucesivas de convolución y *pooling* se añaden una o más capas totalmente conectadas. Se trata de una capa en la que todos los resultados de la capa anterior están conectados a todos los nodos de la capa siguiente [45].

3.4.4 Capa de clasificación

La última capa de esta red es una capa clasificadora que tendrá tantas neuronas como número de clases se quieran predecir. Para el caso de estudio concreto de partes de la naranja, existirán 3 clases, pezón, culo, defecto y una clase adicional para el fondo de la imagen. La salida de esta capa será un vector que indique la probabilidad de una subimagen de la imagen de entrada de pertener a una clase concreta.

3.4.5 Arquitecturas de las CNN

Existen algunas arquitecturas de redes neuronales convolucionales, las más utilizadas son las siguientes:

- LeNet: Fue la primera aplicación exitosa descrita por Yann LeCun. El sistema se desarrolló para su uso en un problema de reconocimiento de caracteres a mano.
- AlexNet: Fue un modelo creado por Alex Krizhevsky para el concurso ILSVRC que tiene por objetivo clasificar fotografías de objetos. Algunos de los métodos novedosos empleados en el modelo AlexNet fueron la utilización de funciones de activación ReLU después de cada capa convolucional, el uso de la función de activación *softmax* en la capa de salida que actualmente es un elemento básico para la clasificación de múltiples clases con redes neuronales y la aplicación de *maxpooling* en lugar de la agrupación media. Otra novedad fue la apilación de capas convolucionales, anteriormente cada capa convolucional estaba seguida de una capa de *pooling*.
- VGGNet: Fue implementada por Karen Simonyan y Andrew Zisserman. La primera diferencia con las arquitecturas previamente definidas es el uso de un gran número de filtros pequeños.
- Redes residuales ResNet: Esta innovación fue propuesta por Kaiming He en 2016. Este modelo utiliza conexiones residuales que permiten conexiones directas entre capas no consecutivas, se saltan las capas intermedias [46]. En este proyecto se ha utilizado la arquitectura ResNet50.

3.5 Parámetros e hiperparámetros

Las redes neuronales constan de parámetros e hiperparámetros, los primeros son las variables que se estiman durante el proceso de entrenamiento, es decir, los pesos de las neuronas. Los hiperparámetros son variables que pueden ser configuradas por el usuario para ajustar los algoritmos de aprendizaje, algunos de ellos son los siguientes:

- Número de capas de la red: Cuánto más capas contenga la red mayor será el tiempo de cálculo y su capacidad de aprendizaje.
- Número de neuronas por capa.
- Número de épocas: Cada época representa el paso de todos los datos de entrenamiento por la red neuronal.
- Tamaño del *batch*: El conjunto de datos se divide en lotes más pequeños, *batches*, el tamaño del *batch* indica el número de ejemplos de entrenamiento que contiene cada lote.
- Tamaño de los filtros empleados.
- Tasa de aprendizaje o *learning rate*: Este hiperparámetro regula la velocidad de aprendizaje, indica cuánto afecta el gradiente a la actualización de nuestros parámetros en cada iteración.

- Tipo de optimización: Como se ha comentado en el apartado 3.3, los algoritmos de optimización empleados pueden ser el descenso de gradiente, descenso de gradiente estocástico o descenso de gradiente con momentum entre otros.
- Función de activación: Las funciones de activación más empleadas se han comentado en el aparatado 3.1.
- Función de coste o pérdidas.

3.6 Redes para detección de objetos

En la tarea de detección de objetos en imágenes las redes empleadas son las redes neuronales convolucionales basadas en regiones. Los avances en este campo empezaron con los detectores basados en regiones, en primer lugar, R-CNN, posteriormente Fast R-CNN y finalmente, Faster R-CNN.

3.6.1 Región CNN (R-CNN)

Uno de los primeros avances en el uso de redes neuronales convolucionales en un sistema de detección de objetos fue el *Region based Convolutional Neural Networks* conocido como Region CNN o R-CNN, que tenía un rendimiento de detección de objetos mucho mayor que otros métodos populares en ese momento.

La primera etapa de R-CNN es la generación de propuestas de regiones de una imagen que pueden pertenecer a un objeto en particular. Se usa un algoritmo selectivo de búsqueda que genera subsegmentaciones en la imagen que pueden pertenecer a un objeto, basado en el color, textura, tamaño y forma. Se obtienen alrededor de 2000 propuestas de región.

Estas 2000 regiones se convierten a entradas fijadas de tamaño 227x227 mediante una deformación simple para permitir su uso en el ajuste de la CNN ya que para alimentar a la CNN se necesitan entradas de tamaño fijo.

La red neuronal convolucional utilizada se llama *AlexNet* y se alimentará con las 2000 regiones propuestas para clasificarlas y obtener la clase a la que pertenecen. Esto se consigue a través de máquinas de vectores de soporte (SVM, *Support Vector Machine*) lineales que proporcionan el nivel de confianza con el que las regiones propuestas pertenecen a una clase.

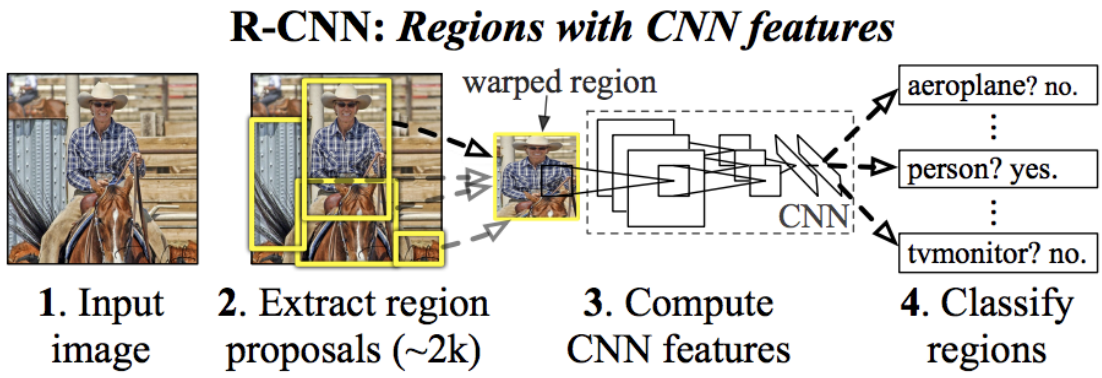


Figura 3.12: Esquema R-CNN [9]

En la figura 3.12 está representado el esquema de la estructura Region CNN, se observa como en primer lugar el sistema recibe una imagen de entrada, a continuación, extrae aproximadamente 2000 propuestas de regiones, posteriormente, calcula las características para cada propuesta de región utilizando una red neuronal convolucional y por último, clasifica cada región utilizando SVM lineales.

3.6.2 Fast R-CNN

Fast R-CNN es el sucesor de R-CNN, como su propio nombre indica es más rápido que este último. R-CNN necesitaba clasificar 2000 propuestas de región por cada imagen, lo que resultaba en un gran coste computacional de entrenamiento de la red.

Como mejora a este problema Fast R-CNN toma como entrada una imagen completa. Realiza la extracción de las características de la imagen antes de proponer las regiones.

De cada imagen se produce un mapa de características. Para cada propuesta de objeto la RoI (región de interés o *Region of Interest*) pooling layer extrae un vector de características de longitud fija del mapa de características. Cada vector de características se introduce en las capas totalmente conectadas (FC, *Fully Connected*) que finalmente se ramifican en dos capas de salida. Una capa softmax que indica la probabilidad sobre K clases de objetos más una clase de fondo general y otra capa que genera cuatro valores reales para cada una de las K clases de objetos. Estos cuatro valores corresponden con las coordenadas de las esquinas del cuadro delimitador.

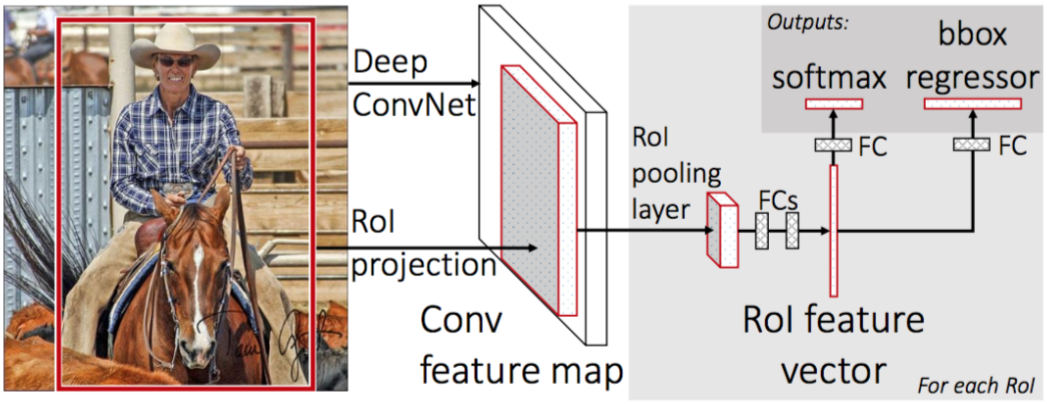


Figura 3.13: Fast R-CNN [9]

La principal diferencia entre R-CNN y Fast R-CNN es que mientras en R-CNN se extraían las características de la imagen a través de la red neuronal convolucional para cada propuesta de región en Fast R-CNN se obtiene el mapa de características de toda la imagen y las propuestas de región obtenidas comparten mapa de características. De esta forma no es necesario ejecutar la red neuronal convolucional para cada una de las propuestas de región como se hacía en R-CNN. El mapa de características generado se emplea para el aprendizaje del clasificador lineal de los objetos y el regresor de los cuadros delimitadores o *bounding boxes* en la literatura inglesa [47].

3.6.3 Faster R-CNN

La evolución entre las diferentes versiones dentro de la familia R-CNN se diferencia en términos de eficiencia computacional, reducción en el tiempo de test y mejora de rendimiento.

Estas arquitecturas generalmente se componen de:

- Una etapa de extracción de características para obtener las características de estos objetos, normalmente usando una red neuronal convolucional.
- Un algoritmo de propuesta de región de interés para generar *bounding boxes* o ubicaciones de posibles objetos en la imagen.
- Una capa de clasificación para predecir a qué clase pertenece cada objeto.
- Una capa de regresión para mejorar la precisión de las coordenadas de los cuadros delimitadores de objetos.

Tanto R-CNN como Fast R-CNN utilizan algoritmos de propuesta de región basados en CPU (*Central Processing Unit*). Faster R-CNN usa otra red convolucional conocida como RPN, *region proposal network* en la literatura inglesa, para generar las propuestas de región. Esto no solo reduce el tiempo sino que también permite que esta etapa comparta capas con las siguientes etapas de detección, lo que provoca una mejora general en la representación de características.

Faster R-CNN usa una RPN como algoritmo de propuesta de región y Fast R-CNN como red de detección, es decir, el módulo RPN le dice al módulo Fast R-CNN donde mirar, la estructura de la red Faster R-CNN se observa en la figura 3.14.

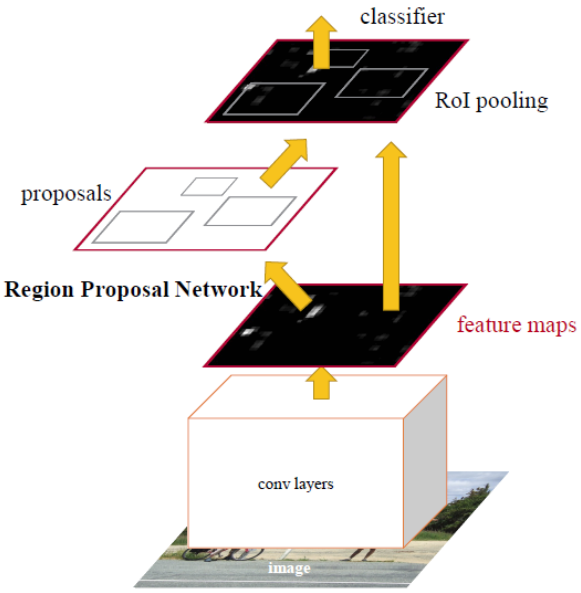


Figura 3.14: Arquitectura Faster R-CNN [10]

Los módulos de los que consta la arquitectura Faster R-CNN se detallan a continuación.

Red de propuesta de región de interés, RPN

La red de propuesta de región toma una imagen como entrada y devuelve un conjunto de propuestas de región, indicando si se trata de un objeto o es parte del fondo dependiendo de la puntuación.

Para generar propuestas de región se desliza una pequeña red sobre el mapa de características. Esta pequeña red toma como entrada una ventana espacial de dimensiones n x n del mapa de características. Para localizar un objeto en el mapa de características se realiza un barrido de diferentes cuadros delimitadores conocidos como anclajes.

Sobre la imagen de entrada se colocan los puntos de anclaje simétricamente distribuidos y sobre estos puntos se coloca un número k de anclajes con diferentes escalas y relaciones de aspecto. Los anclajes indican posibles objetos en varios tamaños y relaciones de aspecto en esta ubicación.

La figura 3.15 muestra 9 posibles anclajes en 3 relaciones de aspecto diferentes y 3 tamaños diferentes colocados en la imagen de entrada para un punto A en el mapa de características de salida.

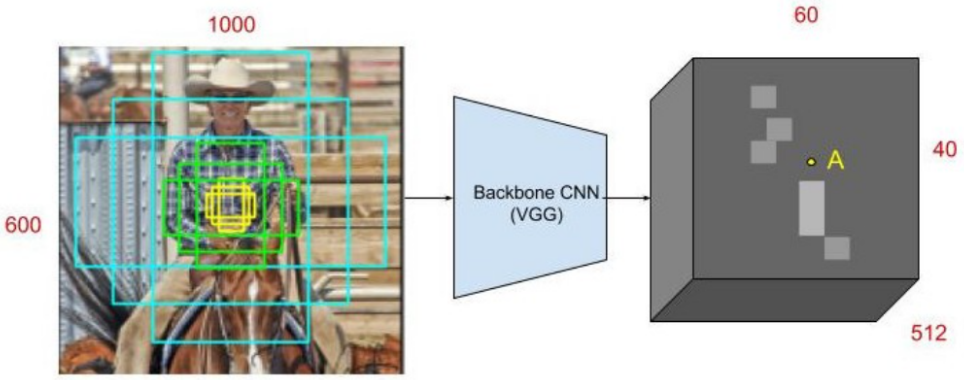


Figura 3.15: Posibles anclajes en la imagen de entrada en la ubicación correspondiente al punto A en el mapa de características [11]

A medida que la red se mueve por cada pixel en el mapa de características de salida, tiene que verificar si estos k anclajes correspondientes que abarcan la imagen de entrada realmente contienen objetos y refinar las coordenadas de estos para dar cuadros delimitadores como propuestas de objetos.

La imagen de entrada se pasa primero por la red neuronal convolucional troncal para obtener el mapa de características de tamaño (60, 40, 512). Además de la mejor eficiencia en tiempo de test, otra razón clave para usar una RPN como generador de propuestas de región es la ventaja de compartir el peso entre la red troncal RPN y la red troncal del detector Fast R-CNN.

En primer lugar, se aplica una convolución 3x3 con 512 unidades al mapa de características de la red troncal como se muestra en la figura 3.16 para proporcionar un mapa de características de dimensión 512 para cada ubicación. A esto le siguen dos capas hermanas, una capa de convolución 1x1 con 18 unidades para la clasificación de objetos y una capa de convolución 1x1 con 36 unidades para la regresión de los cuadros delimitadores.

Las 18 unidades en la rama de clasificación dan una salida de tamaño (H, W, 18). Esta salida se usa para dar probabilidades de si cada punto en el mapa de características de la red troncal contiene un objeto dentro de los 9 anclajes en ese punto.

Las 36 unidades en la rama de regresión dan una salida de tamaño (H, W, 36). Esta salida se utiliza para dar los 4 coeficientes de regresión de cada uno de los 9 anclajes para cada punto en el mapa de características de la red troncal. Estos coeficientes de regresión se utilizan para mejorar las coordenadas de anclajes que contienen objetos.

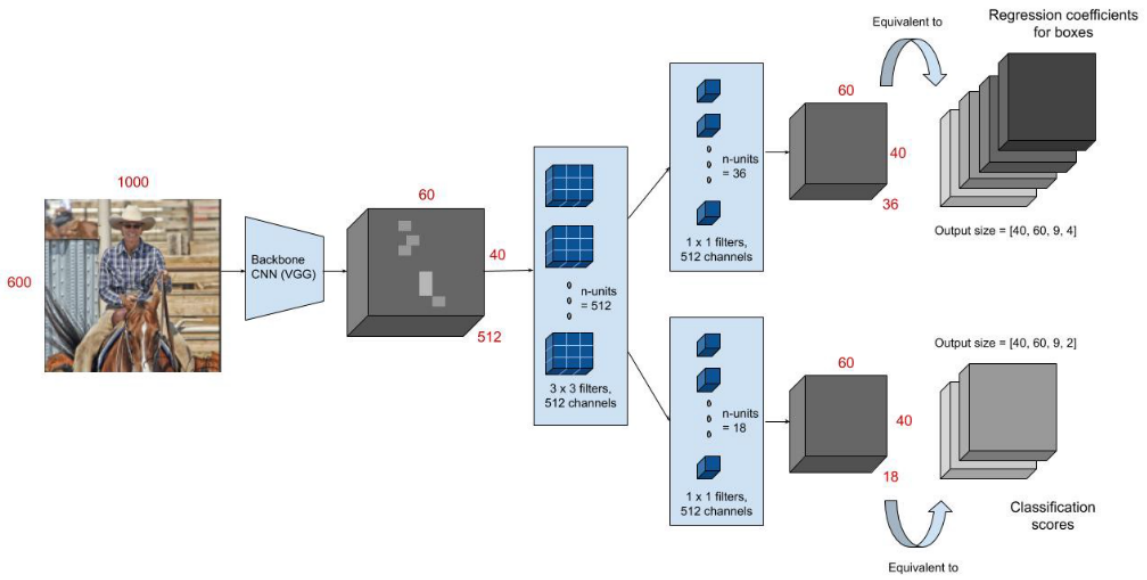


Figura 3.16: Arquitectura de la red de propuesta de región [11]

Para el entrenamiento de la RPN se asigna una clase binaria a cada anclaje, la cual indica si se trata de un objeto o del fondo de la imagen. Para ello se utiliza un umbral en el valor de la intersección entre la unión, IoU (*Intersection over Union*). IoU es una métrica que indica el grado de similitud entre un anclaje y la etiqueta en los datos (figura 3.17).

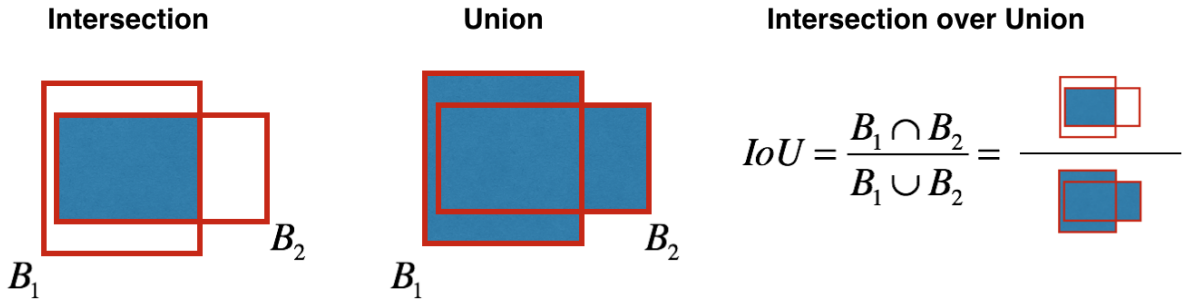


Figura 3.17: *Intersection over Union* [12]

El mapa de características de salida consta aproximadamente de unas 40x60 ubicaciones, correspondientes a 40x60x9, unos 20000 anclajes en total.

Se considera que un anclaje es una muestra "*positiva*", es decir, se trata de un objeto si:

- El anclaje tiene un IoU mayor que 0.7.
- Si ningún anclaje supera el valor de 0.7, se etiquetarán como objetos los anclajes con mayor valor IoU.

Un anclaje es etiquetado como "*negativo*", perteneciente al fondo, si su IoU con todos los cuadros de verdad es inferior a 0.3. Los anclajes restantes (ni positivos, ni negativos) no se tienen en cuenta para el entrenamiento RPN. Esto deja alrededor de 6000 anclajes por imagen.

Cada mini-lote para entrenar la RPN proviene de una sola imagen. El muestreo de todos los anclajes de esta imagen sesgaría el proceso de aprendizaje hacia muestras negativas, por lo que se seleccionan al azar 128 muestras positivas y 128 negativas para formar el lote, rellenando con muestras negativas adicionales si hay un número insuficiente de positivos.

Para definir la función de coste se tienen en cuenta el error de clasificación (clasificar un objeto como fondo y viceversa) y el error de regresión (cuánto se acerca un anclaje a la etiqueta real). La función de coste para una imagen se define según la ecuación 3.24.

$$C(\{p_i\},\{t_i\}) = \frac{1}{N_{cls}}\sum_i C_{cls}(p_i, p_i^*) + \lambda\frac{1}{N_{reg}}\sum_i p_i^* C_{reg}(t_i, t_i^*) \tag{3.24}$$

donde i es el índice del anclaje en el mini-lote, p_i es la probabilidad predicha de que un anclaje contenga o no un objeto, p_i^* es 1 si el anclaje contiene un objeto y 0 si no lo contiene, t_i es un vector de 4 variables $[t_x, t_y, t_w, t_h]$ que representan las coordenadas del cuadro delimitador predicho, y t_i^* es el cuadro delimitador objetivo de la regresión.

El coste de clasificación C_{cls} se define como:

$$C_{cls} = -p_i^* log(p_i) \tag{3.25}$$

El coste de regresión C_{reg} se calcula como la función $smooth_{L_1}(t_i - t_i^*)$:

$$smooth_{L_1}(x) = \begin{cases} \frac{x^2}{2} & \text{si } |x| < 1 \\ |x| - \frac{1}{2} & \text{si } |x| \geq 1 \end{cases} \tag{3.26}$$

C_{reg} se activa solo si el anclaje realmente contiene un objeto, es decir, si p_i^* es 1.

Los dos términos están normalizados por N_{cls} y N_{reg} y ponderados por un parámetro de equilibrio λ. En la ecuación 3.24, C_{cls} se normaliza por el tamaño del *mini-batch*, es decir, $N_{cls} = 256$, y C_{reg} se normaliza por el número de ubicaciones de anclaje, es decir, $N_{reg} \approx 2400$. En [10] el valor de λ es de $\lambda = 10$.

El objetivo de regresión t_i* se calcula como:

$$t_x^* = (x^* - x_a)/w_a,\ t_y^* = (y^* - y_a)/h_a,\ t_w^* = log(w^*/w_a),\ t_h^* = log(h^*/h_a) \tag{3.27}$$

donde x, y, w y h corresponden a las (x, y) de la esquina superior izquierda y la altura h y la anchura w del cuadro. $x*$, x_a representan las coordenadas del cuadro de anclaje y su correspondiente cuadro delimitador de verdad.

La red RPN entrena k diferentes subredes que no comparten pesos, de esta manera se ajustan las relaciones de aspecto y tamaños de cada uno de los k anclajes que se deslizan por la imagen. Entonces, la pérdida de regresión para un anclaje i se aplica a su regresor correspondiente (si es una muestra positiva).

En el momento del test, la salida de regresión aprendida t_i se puede aplicar a su cuadro de anclaje correspondiente y los parámetros x, y, w y h para el cuadro delimitador de propuesta de objeto pronosticado se pueden volver a calcular a partir de:

$$t_x = (x - x_a)/w_a,\ t_y = (y - y_a)/h_a,\ t_w = log(w/w_a),\ t_h = log(h/h_a) \tag{3.28}$$

Una vez entrenada la red, los tamaños y relaciones de aspecto de los anclajes son fijos. Dichos tamaños se ajustan durante el entrenamiento pero posteriormente se utilizarán siempre las mismas relaciones de aspecto y tamaño de los anclajes para proponer regiones que contengan un objeto.

Debido a la superposición de algunas propuestas de región se aplica un método llamado supresión no máxima o NMS (*Non-Maximum Suppression*). El NMS ordena las propuestas por puntuación y descarta todos los cuadros delimitadores que tienen un IoU mayor que 0.7 con otro cuadro delimitador.

Una vez eliminadas las propuestas duplicadas se seleccionan las N propuestas que tienen mayor puntuación, en [10] el valor de N es de $N = 2000$.

ROI Pooling

Las regiones de interés propuestas por la red RPN se introducen al clasificador que clasificará las regiones en sus clases correspondientes y al regresor que dará una posición más precisa del cuadro delimitador que contiene objetos en la imagen.

Las regiones propuestas por la RPN tienen diferentes tamaños por ello se debe realizar un *pooling* de la región de interés conocido como *region of interest pooling* o *ROI pooling*.

Consiste en fijar un tamaño q y dividir las regiones de interés en q partes iguales, realizando *max pooling* en esas q partes. Así, todas las regiones de interés tendrán el mismo tamaño a la salida de la capa de *ROI pooling*.

Después de pasar por las dos capas completamente conectadas, las características se introducen en las ramas de clasificación y regresión.

Clasificador Fast R-CNN

El clasificador de imágenes recibe el mapa de características y les asigna una clase. Esta última fase consiste en una capa para la clasificación y un regresor de los cuadros delimitadores para mejorar la precisión de estos.

Por tanto, este módulo se compone de dos subredes:

1. Una capa totalmente conectada con *N+1* unidades donde *N* es el número de clases y se añade una clase que hace referencia al fondo de la imagen. Las características se pasan por una capa *softmax* para obtener la puntuación de clasificación, es decir, la probabiliadad de una propuesta de pertenecer a una clase concreta.

2. Una capa totalmente conectada con *4N* unidades encargada de realizar la regresión de los cuadros delimitadores. El regresor es independiente del tamaño pero es específico para cada clase, es decir, todas las clases tienen regresores individuales con 4 parámetros cada uno correspondiente a una de las *N*4* salidas de la capa de regresión.

En el entrenamiento del clasificador *Fast R-CNN* las propuestas con un valor de IoU mayor de 0.5 con alguna clase se clasifican como pertenecientes a dicha clase. Las propuestas con un valor de IoU entre 0.1 y 0.5 se clasifican como fondo y las propuestas con un valor de IoU menor de 0.1 se ignoran.

Las funciones de coste utilizadas son las mismas que para la red RPN, el *log loss* para la clasificación y la función $smooth_{L_1}$ para la regresión.

Para forzar a la red a compartir los pesos de la red neuronal convolucional troncal entre la RPN y el detector se utiliza un método de entrenamiento en 4 pasos:

- La RPN se entrena de manera independiente. La red neuronal convolucional troncal para esta tarea se inicializa con pesos de una red capacitada para una tarea de clasificacion ImageNet y se ajusta para la tarea de propuesta de región.

- La red detectora Fast R-CNN también se entrena independientemente. La CNN troncal para esta tarea se inicializa con pesos de una red capacitada para una tarea de clasificación ImageNet y se ajusta para la tarea de detección de objetos. Los pesos RPN son fijos y las propuestas de RPN se utilizan para entrenar la Faster R-CNN.

- La RPN se inicializa ahora con pesos de la Faster R-CNN y se ajusta para la tarea de propuesta de región. Esta vez, los pesos en las capas comunes entre la RPN y el detector permanecen fijos y solo se ajustan los de las capas exclusivas de la RPN. Esta es la RPN final.

- Una vez más usando la nueva RPN, el detector Fast R-CNN se ajusta. Otra vez, solo las capas exclusivas de la red detectora se ajustan y los pesos de las capas comunes quedan fijos.

Esto proporciona un marco de detección Faster R-CNN que ha compartido capas convolucionales [11].

Para el desarrollo de este proyecto, se empleará la arquitectura Faster R-CNN, ya que muestra un gran rendimiento en detección de objetos, cuenta con implementaciones en PyTorch y con varias redes con parámetros preentrenados y de código abierto en el repositorio [48].

Capítulo 4

Tecnologías y Herramientas

En este capítulo se presentan las tecnologías y herramientas que se han empleado durante el desarrollo de este proyecto.

4.1 Dataset

El *dataset* es el conjunto de datos que se usan para el entrenamiento de la red.

Para el desarrollo de este proyecto se ha utilizado la base de imágenes COCO (*Common Objects in Context)*. COCO es un conjunto de datos de detección, segmentación y subtitulación de objetos a gran escala [49]. El conjunto de datos COCO ha sido empleado por los modelos utilizados en el presente proyecto para su previo entrenamiento.

Además, en este proyecto se ha creado un *dataset* a partir de una colección de 2928 imágenes de naranjas. Estas imágenes han sido capturadas mientras la naranja gira en un único sentido sobre unos rodillos, capturando 5 fotogramas que se concatenan formando la imagen final que puede verse en la figura 4.2.

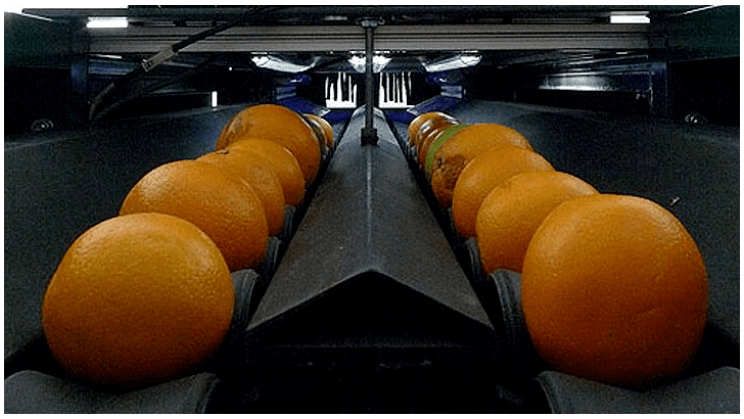


Figura 4.1: Captura de las imágenes

4.1.1 Creación de las anotaciones

La creación de las anotaciones es una parte fundamental en la preparación del *dataset*. Para ello, se crean cuadros delimitadores o *bounding boxes* sobre la imagen.

Existen 3 clases a detectar, pezón o pedicel, culo o bottom y defecto. El etiquetado consiste en colocar los cuadros delimitadores en las diferentes partes de la naranja que se corresponden con las 3 clases existentes. Se han empleado *boxes* rojos para el pezón o pedicel, *boxes* azules para el culo o bottom y *boxes* verdes para los defectos.

Estos cuadros se definen mediante las anotaciones, que corresponden con las cuatro coordenadas en pixeles de las cuatro esquinas del cuadro. Tienen la forma $(\mathrm{x}_1, y_1, x_2 - x_1, y_2 - y_1)$.

Las imágenes se han etiquetado mediante la herramienta de etiquetado *datasets.ai2.upv.es/oranges* desarrollada por el Instituto Universitario de Automática e Informática Industrial (ai2).

Mediante dicha herramienta de etiquetado que puede verse en la figura 4.2 se procede manualmente a colocar los boxes delimitando las distintas partes de la naranja en las imágenes de entrenamiento y es la propia herramienta la que genera las anotaciones de cada uno de los *boxes*.

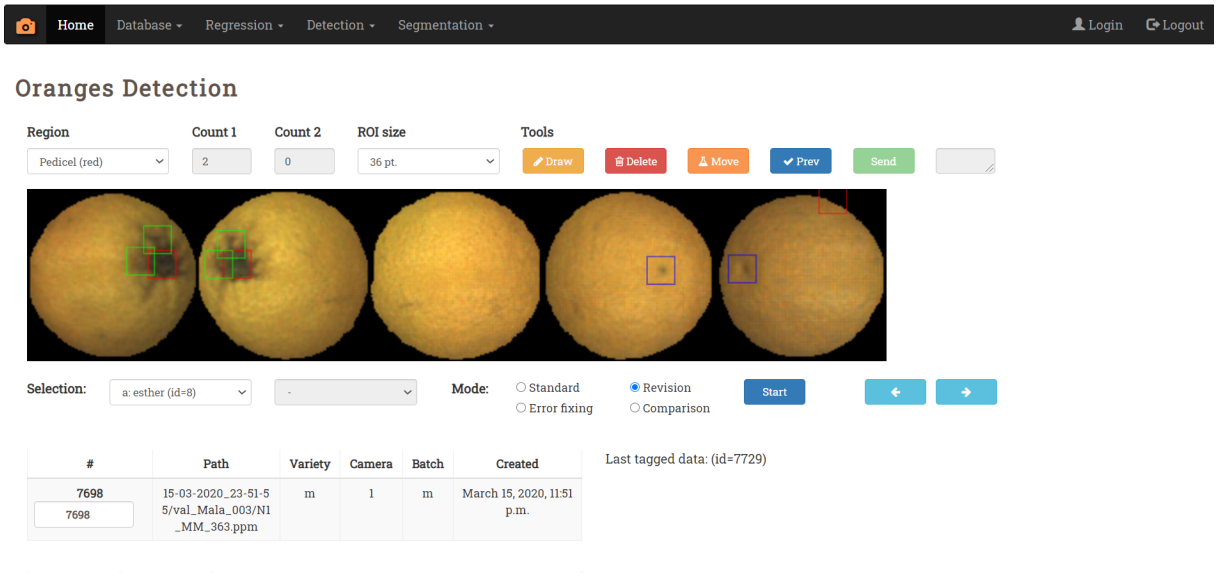


Figura 4.2: Herramienta de etiquetado

Las anotaciones realizadas por la herramienta web se guardan en archivos con extensión .xml que leeremos para entrenar el modelo.

4.2 Lenguaje de Programación

El lenguaje de programación empleado ha sido Python, por ser el más utilizado en el campo de la inteligencia artificial y más concretamente en el del *Deep Learning*.

Una de las ventajas que ofrece este lenguaje de programación es la amplia variedad de librerías desarrolladas para él. Algunas de las empleadas en este proyecto son:

- PyTorch, detallada en la sección 4.3.
- Numpy, librería de código abierto que facilita la computación numérica con Python [50].
- *Python Imaging Library* (PIL), librería de código abierto para Python que proporciona potentes capacidades de procesamiento de imágenes [51].

4.3 PyTorch

PyTorch es una librería de aprendizaje automático de código abierto para realizar cálculos numéricos haciendo uso de la programación de tensores. Además permite su ejecución en GPU para acelerar los cálculos.

PyTorch fue principalmente desarrollado por Facebook AI e introducido en 2016. Se usa para la investigación y desarrollo en el campo del *Machine Learning*, centrado principalmente en el desarrollo de redes neuronales [52] [53].

Algunas alternativas a PyTorch son TensorFlow, Caffe, Theano, Keras, entre otras.

4.4 Google Colaboratory

Para el desarrollo de este proyecto se ha utilizado el entorno gratuito Google Colaboratory. Es un entorno interactivo que permite desarrollar código Python de manera dinámica. Además, Google Colaboratory permite hacer uso de una GPU en la nube.

Entrenar un modelo es un proceso iterativo con un gran número de cálculos que puede llegar a ser muy costoso. Este proceso puede verse acelerado gracias al uso de un hardware de altas prestaciones.

Por ello, es necesario el uso de una GPU que permite reducir considerablemente los tiempos en comparación al uso de una CPU.

Capítulo 5

Desarrollo

En el presente capítulo se van a explicar cada uno de los pasos que se han llevado a cabo para la elaboración de este proyecto. Desde el inicial etiquetado de las imágenes hasta la implementación de la red neuronal convolucional y su posterior evaluación.

5.1 Criterios de etiquetado

Una de las etapas más importantes para el correcto funcionamiento del modelo es el etiquetado de las imágenes. Como se ha comentado en la sección 4.1 nuestro *dataset* consta de 2928 imágenes de naranjas, cada una de ellas consiste en cinco fotogramas de la naranja girando en una misma dirección.

Para el correcto etiquetado de estas imágenes se han establecido los siguientes criterios de etiquetado:

1. Etiquetar un solo *box* de la clase *Bottom* y *Pedicel* en cada foto de la naranja.

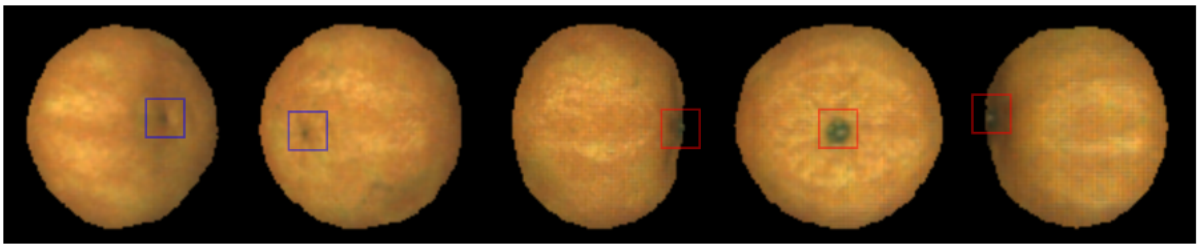


Figura 5.1: Criterio 1

2. En el caso de que el pezón o el culo de la naranja sean defectuosos etiquetarlos también como defectos. En este caso tendremos doble etiquetado.

Figura 5.2: Criterio 2

3. Cuando existan defectos grandes se utilizarán varios *boxes* de tamaño pequeño fijo (18 px) para marcar los defectos. Si se trata de una naranja verde, este defecto se puede etiquetar con un *box* de tamaño variable que cubra todo el defecto.

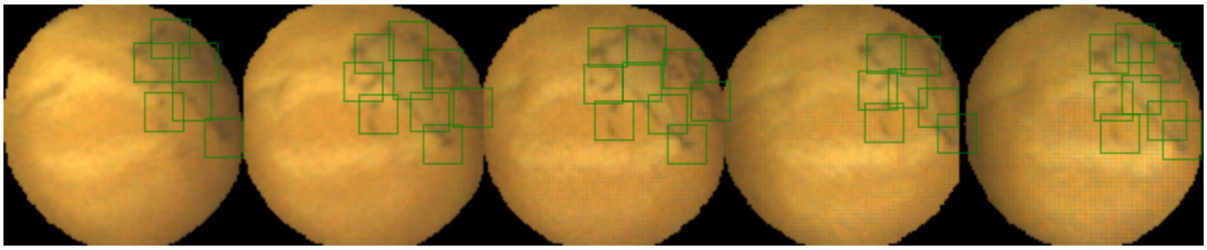


Figura 5.3: Criterio 3

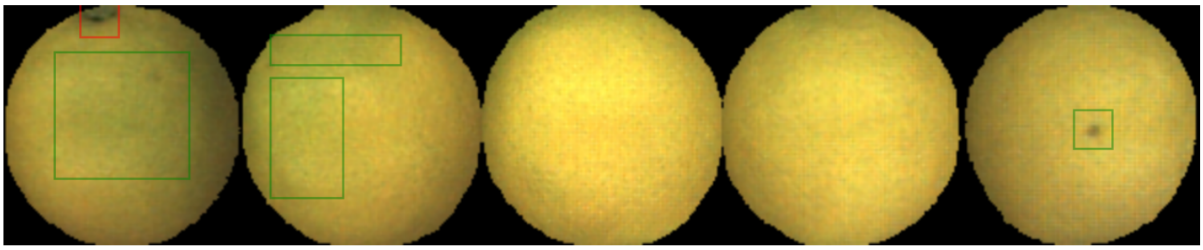


Figura 5.4: Criterio 3

4. Se etiquetará el culo o pezón de la naranja siguiendo la secuencia de giro lógica aunque no se tenga una información clara para la detección. En este caso deberá haber algo de información que nos permita localizar el *box* correctamente. Por ejemplo, en el caso de *pedicel* sería un pequeño resalto.

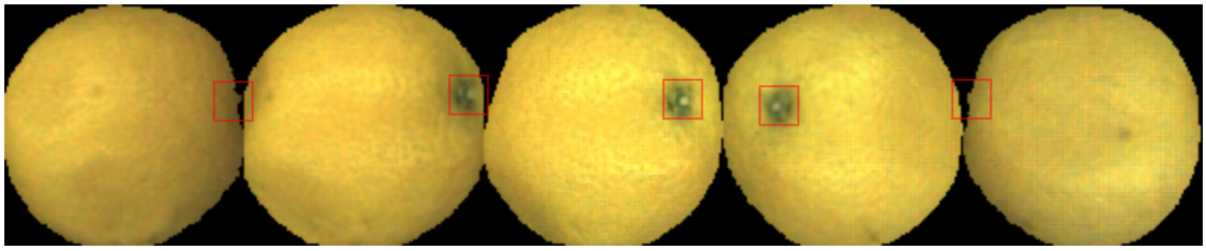


Figura 5.5: Criterio 4

5.2 Transferencia de aprendizaje

Entrenar una red desde cero puede llegar a ser una tarea muy costosa y lenta computacionalmente. Dado que las redes neuronales empleadas para detección de objetos son redes profundas con muchas capas, el tiempo de entrenamiento puede ser muy lento. Por esta razón, es habitual partir de una red previamente entrenada y ajustar los valores de los parámetros para el caso específico de detección que se desea abordar. Este fenómeno es conocido como transferencia de aprendizaje.

La transferencia de aprendizaje tiene dos ventajas principales, un menor tiempo de entrenamiento y un menor número de datos requeridos para el entrenamiento [54].

5.3 Métricas de evaluación

Para evaluar un modelo en detección de objetos se utilizan diferentes métricas que miden la calidad de sus predicciones. Algunas de ellas se comentan a continuación.

5.3.1 Matriz de confusión

La matriz de confusión evalúa los resultados obtenidos, dividiéndose en verdaderos positivos, falsos positivos, falsos negativos y verdaderos negativos. Con estos valores se puede calcular las métricas de *precision* y *recall*.

- Verdaderos positivos (TP): Predicciones correctas respecto a los datos reales.
- Falsos positivos (FP): Se detecta una clase que no se encuentra en los datos reales.
- Falsos negativos (FN): Cuando una determinada clase no se detecta o se detecta y se clasifica erróneamente como perteneciente a otra clase.
- Verdaderos negativos (TN): Valores de detección que no son detectados y realmente no existen en los datos reales.

Tabla 5.1: Métricas

		Actual	
		Positivo	Negativo
Predicción	Positivo	Verdadero Positivo (TP)	Falso Positivo (FP)
	Negativo	Falso Negativo (FN)	Verdadero Negativo (TN)

La métrica *Precision* mide el porcentaje de predicciones positivas correctas entre los casos positivos en realidad. Se calcula según la ecuación 5.1.

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

La métrica *Recall* o sensibilidad mide el porcentaje de predicciones correctas entre las predicciones hechas. Esta métrica se calcula según la ecuación 5.2 [55].

$$Recall = \frac{TP}{TP + FN} \tag{5.2}$$

La métrica *Accuracy* nos da el porcentaje total de los aciertos de nuestro modelo y se calcula según la ecuación 5.3 [13].

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{5.3}$$

5.3.2 Intersection over Union (IoU)

Esta métrica es comunmente usada en detección de objetos y tal y como se ha mencionado en el apartado 3.6.3 se utiliza para medir la precisión de un detector de objetos en un conjunto de datos en particular.

Para la aplicación de esta métrica se necesitan los *bounding boxes* reales, es decir, los que se han etiquetado mediante la herramienta *datasets.ai2.upv.es/oranges* y los *bounding boxes* obtenidos mediante la predicción del modelo.

Para el cálculo de esta métrica se realiza la división entre el área de superposición de ambos *bounding boxes* y el área de unión de ellos.

Si el valor de IoU obtenido es mayor que un valor fijado la predicción será considerada como buena.

5.3.3 AP (Average Precision)

Esta métrica se emplea para resumir la curva que forma la relación entre las métricas de *Precision* y *Recall*.

Para cada umbral de IoU podemos medir la precisión y sensibilidad del detector creando una curva. En COCO los valores de umbral de IoU se cambian desde 50 % a 95 % con un paso de 5 %, lo que se conoce como AP@[0.5:0.95]. Por lo que tenemos 10 parejas *precision-recall* y obtendríamos una curva similar a la figura 5.6 [55].

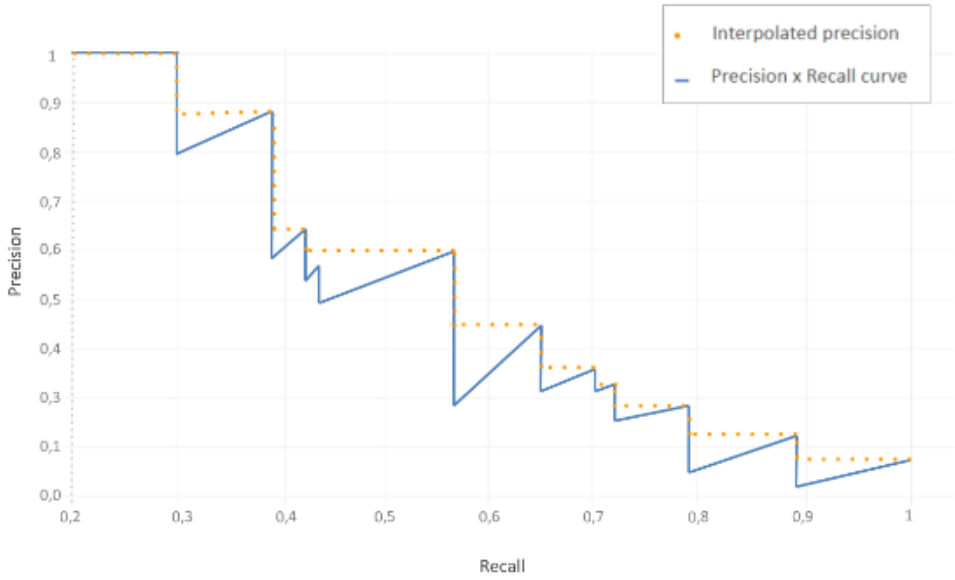


Figura 5.6: Curva *pecision-recall* [13]

El área por debajo de dicha curva es la precisión media o AP.

En el caso de la figura 5.6 el área bajo la curva está dividida en 10 áreas, un área se define por una caída de precisión en un cierto *recall* como se puede observar en la figura 5.7.

Se observa que la curva es decreciente, lo que tiene sentido ya que existe un compromiso entre *precision* y *recall*.

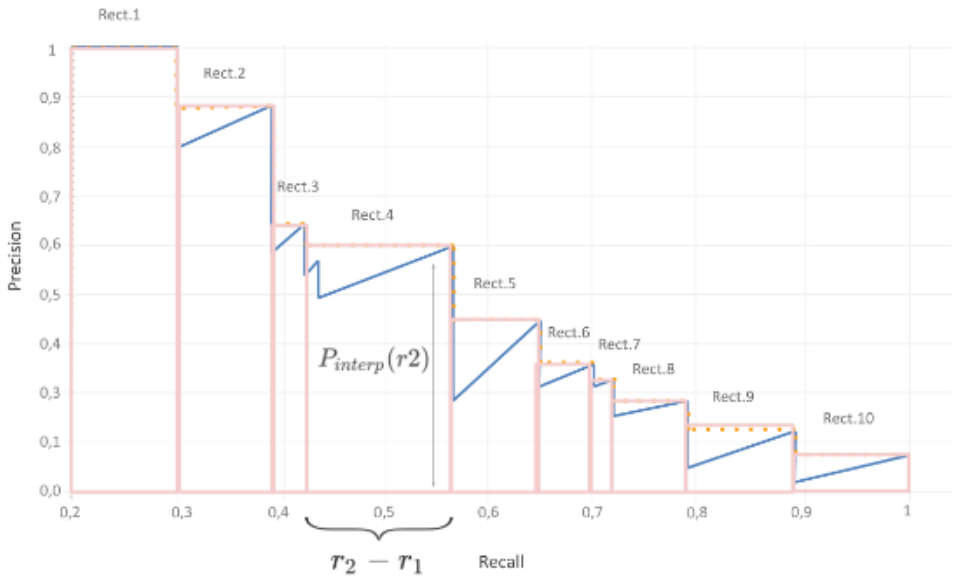


Figura 5.7: División áreas bajo la curva [13]

Matemáticamente se calcula como:

$$AP = \sum (r_n - r_{n-1}) p_{interp}(r_n) \tag{5.4}$$

donde $p_{interp}(r_n) = \underset{\tilde{r}>r_n}{max}\, p(\tilde{r})$ [13].

También se evalua individualmente el AP para umbrales de IoU de 0.5 y 0.75, esto se conoce como AP@0.5 y AP@0.75 respectivamente.

El rendimiento se evalúa en distintas dimensiones, donde AP^{small} se usa para objetos de área menor que 32^2, AP^{medium} para objetos de área $32^2 < area < 96^2$ y AP^{large} para objetos grandes de área mayor que 96^2 [13].

5.3.4 mAP (Mean Average Precision)

Si el conjunto de datos contiene N clases, mAP promedia AP sobre las N clases.

$$mAP = \frac{1}{N} \sum_{class=1}^{N} AP_{class} \tag{5.5}$$

COCO promedia mAP sobre diferentes umbrales de IoU, desde 0.5 hasta 0.95 con un paso de 0.05, se denota como mAP@[0.5:0.95]. Por lo tanto, COCO no solo promedia el AP en todas las clases, sino también en los umbrales de IoU definidos [13].

5.3.5 AR (Average Recall) y mAR (Mean Average Recall)

En lugar de evaluar la métrica *recall* para un determinado valor umbral de IoU, se calcula la métrica *average recall* o AR para valores umbrales de IoU entre 0.5 y 0.95 y así se resume la distribución de la sensibilidad en un rango de valores umbral de IoU.

AR describe el área duplicada bajo la curva *Recall x IoU* que muestra los resultados de *recall* para cada valor umbral de IoU entre 0.5 y 0.95, con los valores de IoU en el eje X y los valores de *recall* en el eje Y.

De forma similar a mAP, mAR es la media de AR para las clases de un dataset [13].

5.4 Desarrollo del código

En este apartado se explican los pasos a seguir para el entrenamiento del modelo, detallando el código implementado para esta tarea.

5.4.1 Definición del Dataset

En primer lugar, se crea la clase Dataset. Para ello, se leen las anotaciones creadas con la herramienta de etiquetado *datasets.ai2.upv.es/oranges* en formato .xml y se crea un diccionario que contiene las siguientes carpetas:

- *Boxes*: En ella se especifican las coordenadas de los *bounding boxes*.
- Área de los *boxes*, los cuadros utilizados pueden ser de área 144 px, 324 px o área variable.
- Etiquetas o *labels* para cada *box*: Indican a que clase pertenece cada cuadro. Si se trata del pezón de la naranja la etiqueta será un 1, si se trata del culo será un 2 y si es un defecto será un 3. El número 0 queda reservado para la clase de fondo de la imagen.
- Id de la imagen: Se trata de un identificador único para cada imagen.

5.4.2 Definición del modelo

Para el desarrollo del presente proyecto se ha utilizado el modelo *Faster R-CNN*. Para entrenar el modelo se ha partido de un modelo pre-entrenado y se ha ajustado para las 4 clases del proyecto en particular. El modelo pre-entrenado utilizado es *"Faster R-CNN ResNet-50"*, se trata de una red de 50 capas [56].

5.4.3 Evaluación de modelos

El conjunto de datos se divide en subconjuntos:

- Entrenamiento: En este subconjunto se encuentran aproximadamente el 80 % de los datos. Es el conjunto empleado para enseñar a la red.
- Validación: 20 % restante. Este subconjunto se emplea para validar el modelo, es decir, elegir los parámetros adecuados. A lo largo del código a este subconjunto se le ha denominado test.

Tanto para el dataset de entrenamiento como para el de test se definen algunas transformaciones como la normalización y la transformación a tensor.

5.4.4 Parámetros e hiperparámetros de la red

Para el desarrollo del código se han utilizado funciones definidas en el código del repositorio de *Github* [48], por lo que existen parámetros fijos que no pueden ser variados. Los hiperparámetros que pueden ser modificados para ajustar el modelo y que realice mejores predicciones son los siguientes:

- *Batch size*
- *Batch size test*
- *Learning rate*
- *Momentum*
- *Weight decay*
- Número de épocas de entrenamiento

Las distintas configuraciones probadas y los resultados obtenidos para cada una de ellas se comentan en el capítulo 6.

Se ha empezado con tamaños de dataset pequeños que no incluían todas las imágenes etiquetadas, tamaños de lote pequeños y pocas épocas de entrenamiento. Progresivamente se han ido incrementando estos parámetros mientras se ajustan el resto de hiperparámetros.

El tamaño del *batch* define el número de datos que se le pasan a la red a la vez, por ejemplo, si el tamaño del *batch* es 2, los pesos se actualizarán cada 2 muestras. Para generar las muestras en grupos del tamaño definido se crea un *Dataloader*. El *Dataloader* genera las muestras además de reordenar aleatoriamente el dataset en cada iteración para el caso del *Dataloader* de entrenamiento.

5.4.5 Creación del modelo y entrenamiento

Se genera el modelo usando una de las funciones del repositorio de *Github* [48].

A continuación, se construye el optimizador, se ha empleado el método de descenso de gradiente estocástico con momento y se ha planificado la tasa de aprendizaje para que vaya disminuyendo a medida que avanza el aprendizaje.

Finalmente, para comenzar el entrenamiento se utiliza la función "*train_one_epoch*" definida en el repositorio de *Github* [48].

Capítulo 6

Resultados

En el presente capítulo se muestra el método empleado para la selección de los hiperparámetros así como los resultados obtenidos para las diferentes combinaciones de hiperparámetros probadas. Una vez seleccionados los hiperparámetros se entrena el modelo y se comentan los resultados obtenidos, tanto las métricas de evaluación explicadas en el apartado 5.3, como el coste temporal del modelo de entrenamiento y de test.

6.1 Selección de los hiperparámetros

Antes de comenzar el entrenamiento se deben seleccionar los hiperparámetros. Tal y cómo se ha comentado en el apartado 5.4.4 los hiperparámetros a ajustar son el *learning rate*, tamaño del *mini-batch* y número de épocas de entrenamiento. Los parámetros *momentum*, *weight decay*, *gamma* y *step size* se establecen fijos e iguales a 0.9, 0.005, 0.1 y 3 respectivamente.

El primer paso es fijar el tamaño del *mini-batch*, se han realizado pruebas para tamaños de *mini-batch* igual a 2 y 4. Al utilizar *Google Colab* como entorno de ejecución existe un límite en los recursos y no es posible definir un tamaño de *mini-batch* superior.

En primer lugar, se ha realizado una comparativa de diferentes tasas de aprendizaje para un tamaño de *mini-batch* igual a 2, posteriormente se ha aumentado el tamaño del *mini-batch* a 4 consiguiendo una mayor robustez del entrenamiento y obteniendo las métricas de la tabla 6.1.

En la figura 6.1 se han representado gráficamente estas métricas mAP obtenidas en el entrenamiento para las diferentes tasas de aprendizaje cuando el tamaño de *mini-batch* es 4.

Se observa que el *learning rate* que proporciona mejores métricas es 0.005, siendo el mAP de 0.3283. Por tanto, será esta tasa de aprendizaje la que se usará para entrenar el modelo.

El número de épocas seleccionado ha sido 10 por la misma razón comentada anteriormente sobre los límites de *Google Colab*.

Tabla 6.1: mAP para distintos *Learning Rate* con tamaño de *mini-batch* = 4

Learning Rate	**0.008**	**0.006**	**0.005**	**0.002**	**0.0005**
Época / mAP					
0	0,2935	0,2894	0,2626	0,2388	0,1174
1	0,3026	0,3013	0,3195	0,3032	0,2199
2	0,2951	0,2922	0,3084	0,3134	0,2602
3	0,3107	0,3239	0,3275	0,3203	0,2743
4	0,3153	0,3223	0,3278	0,3190	0,2759
5	0,3148	0,3252	0,3212	0,3217	0,2816
6	0,3193	0,3267	0,3294	0,3235	0,2815
7	0,3200	0,3267	0,3269	0,3236	0,2811
8	0,3191	0,3272	0,3274	0,3246	0,2810
9	0,3194	0,3265	0,3283	0,3248	0,2811

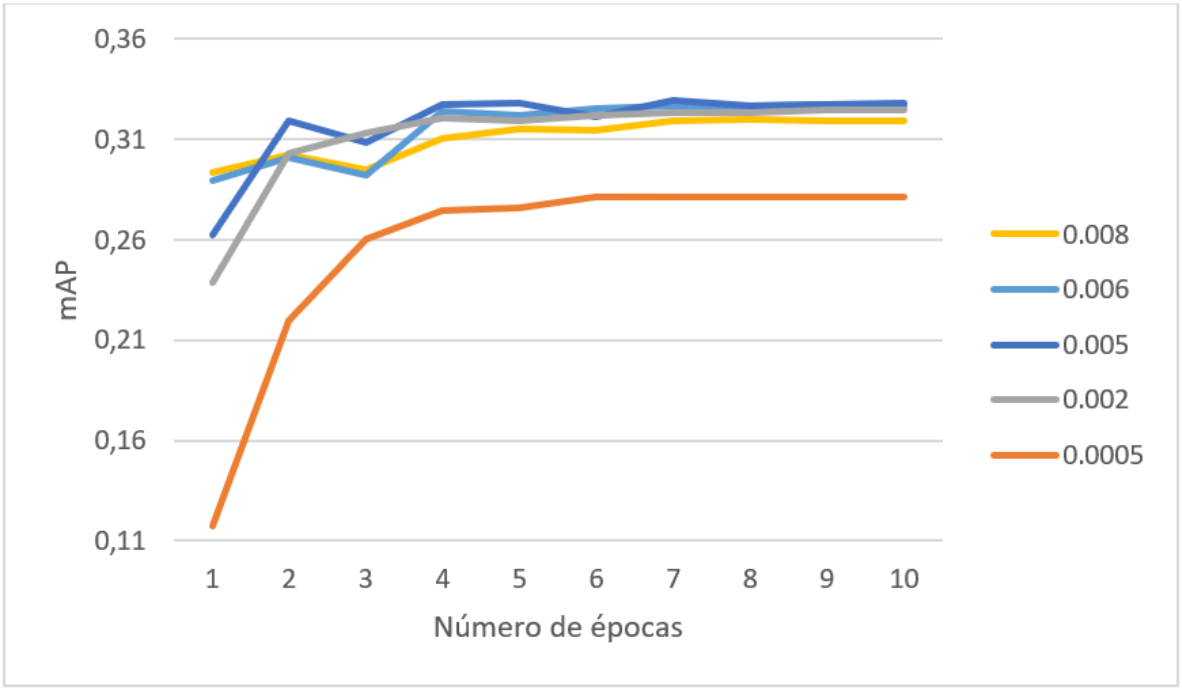


Figura 6.1: Gráfica mAP vs número de épocas para diferentes *learning rate* y tamaño de *mini-batch* = 4

6.2 Modelo final entrenado

Una vez seleccionados los hiperparámetros, se inicia el entrenamiento con los hiperparámetros de la tabla 6.2. Al inicio, el *learning rate* es 0.005 y mediante la técnica de *learning rate decay* explicada en el apartado 3.3.2 se va disminuyendo la tasa de aprendizaje a medida que avanza el entrenamiento. En este caso, se multiplica la tasa de aprendizaje por 0.1 (*gamma*) cada 3 épocas (*step size*).

Tabla 6.2: Hiperparámetros

Parámetro	Valor
Learning rate inicial	0.005
Step size	3
Gamma	0.1
Tamaño de lote entrenamiento	4
Tamaño de lote test	4
Número de épocas	10
Momentum	0.9

6.2.1 Métricas mAP

Se ha escogido la métrica mAP como métrica para evaluación de la precisión del modelo. Los resultados obtenidos se muestran en la tabla 6.3 y se han representado gráficamente en la figura 6.2.

Tabla 6.3: mAP

Epoch	Mean Average Precision (mAP)
0	0.2882
1	0.2931
2	0.3310
3	0.3567
4	0.3641
5	0.3661
6	0.3705
7	0.3693
8	0.3721
9	0.3708

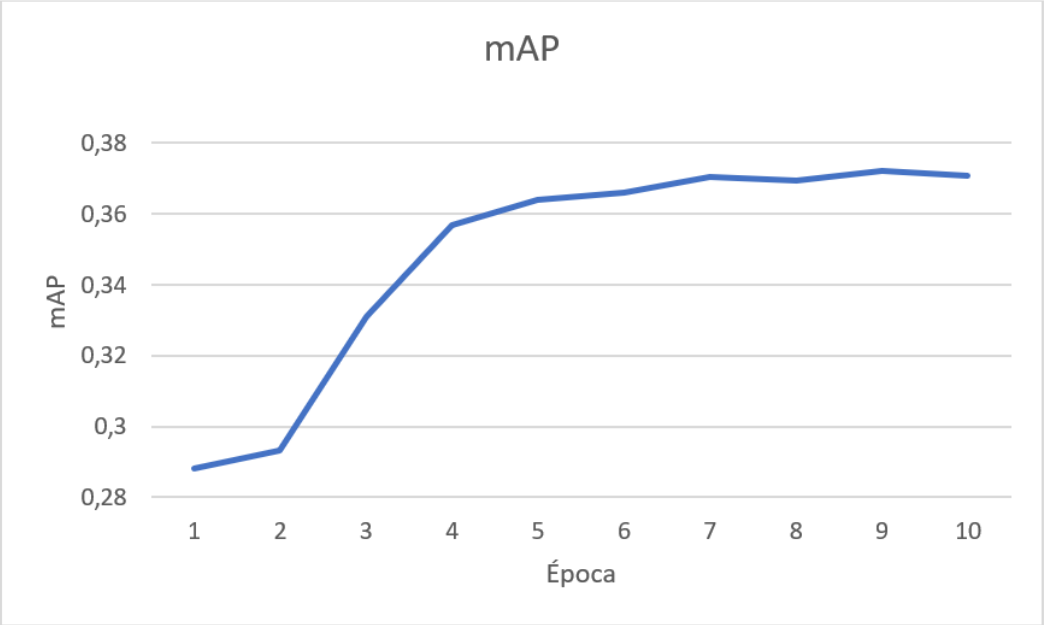


Figura 6.2: Gráfica mAP vs número de épocas

6.2.2 Coste temporal de entrenamiento y test

El coste temporal de entrenamiento y test se mide en segundos por iteración, es decir el tiempo que tarda la red en procesar las muestras contenidas en un *mini-batch*. En este caso como el tamaño del *mini-batch* es igual a 4 sería el tiempo en procesar 4 muestras. Este coste temporal puede variar dependiendo de la tarjeta gráfica que asigna *Google Colab*.

Ambos costes temporales se han representado gráficamente en la figura 6.3.

Tabla 6.4: Coste temporal de entrenamiento y test

Epoch	Tiempo entrenamiento (s/iteración)	Tiempo de test (s/iteración)
0	0.3287	0.1923
1	0.3284	0.2138
2	0.3281	0.1971
3	0.3275	0.1952
4	0.3275	0.1949
5	0.3280	0.1974
6	0.3283	0.1969
7	0.3285	0.1958
8	0.3287	0.1955
9	0.3295	0.1966

No todas las imágenes tardan lo mismo en ser procesadas, la media del tiempo de entrenamiento para un *mini-batch* es de 0.3336 segundos/*mini-batch* con una varianza de 2.6113e-4. La media del tiempo de test es de 0.1976 segundos/*mini-batch* con una varianza de 3.1298e-5.

Figura 6.3: Costes temporales de entrenamiento y test vs número de épocas

6.2.3 Pérdidas durante el entrenamiento

Las pérdidas se van reduciendo a medida que avanza el entrenamiento, hasta llegar a ser prácticamente constantes.

Figura 6.4: Pérdidas vs número de épocas

6.3 Predicción del modelo

En las siguientes figuras se muestran las predicciones realizadas por el modelo entrenado. Se puede observar que son acertadas.

Previamente se han filtrado los resultados obtenidos por la puntuación o *score* obtenido en las predicciones. Se ha fijado un umbral igual a 0.7.

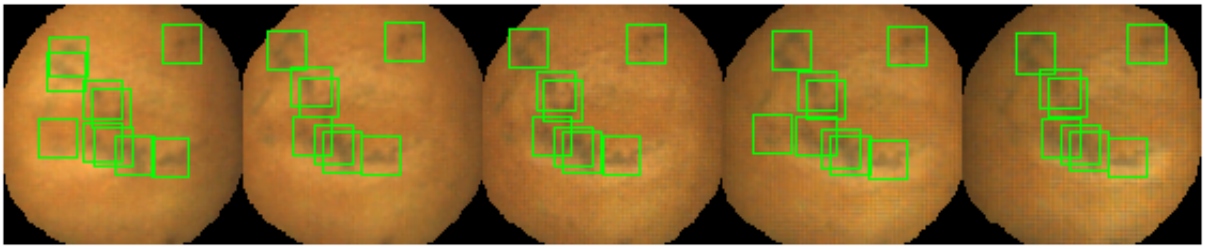


Figura 6.5: Predicción 1

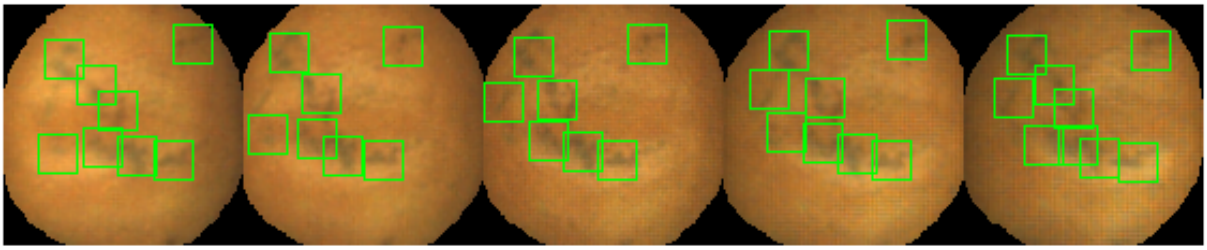


Figura 6.6: Imagen real 1

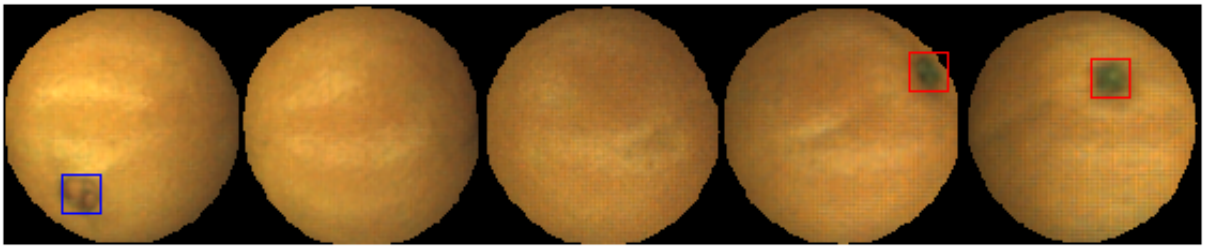


Figura 6.7: Predicción 2

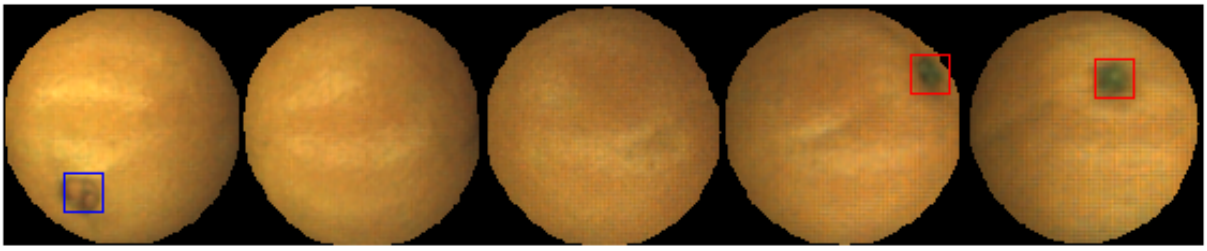


Figura 6.8: Imagen real 2

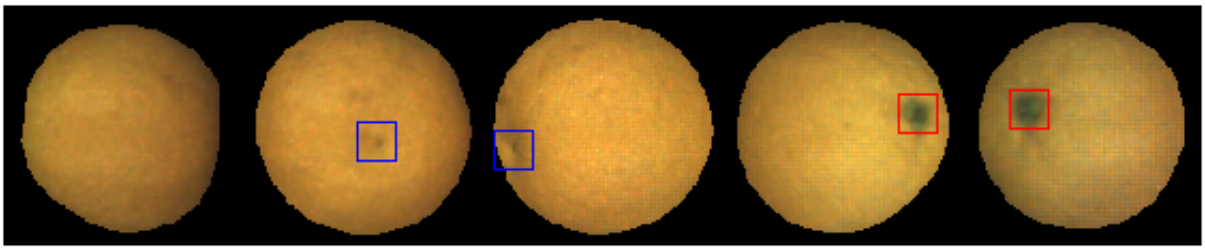


Figura 6.9: Predicción 3

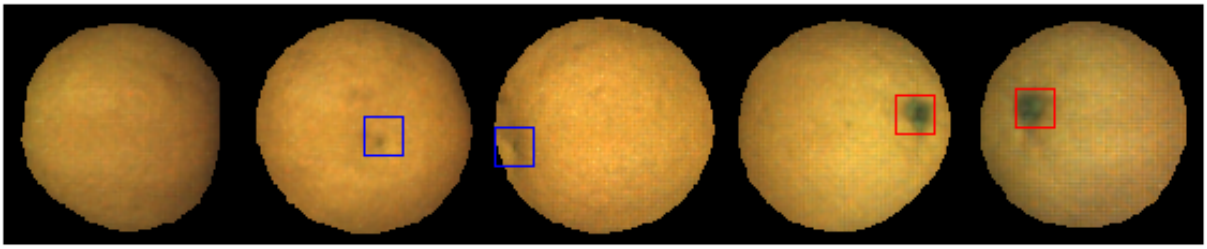


Figura 6.10: Imagen real 3

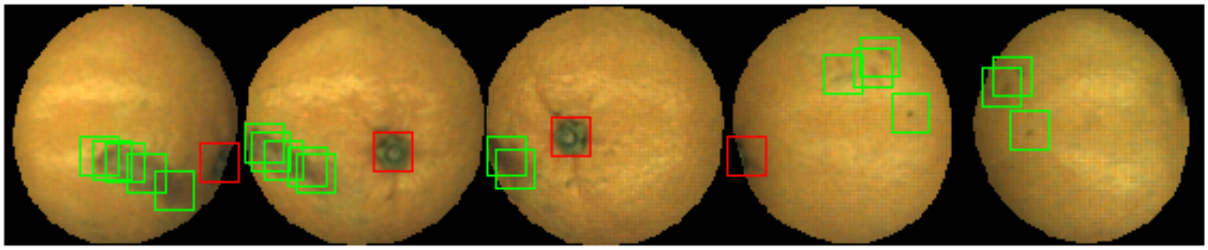


Figura 6.11: Predicción 4

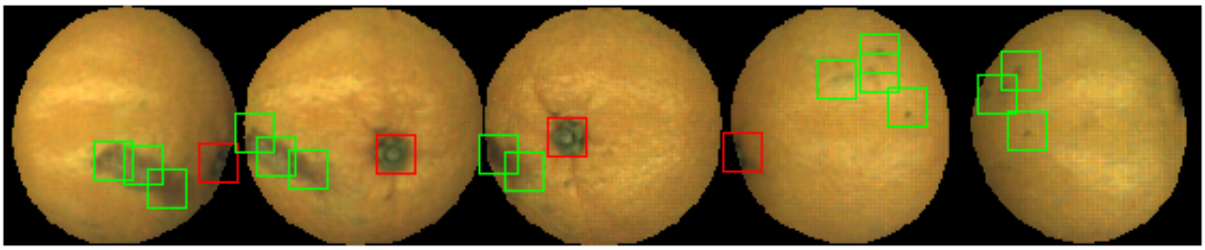


Figura 6.12: Imagen real 4

Capítulo 7

Conclusiones

En este trabajo se ha presentado un método de resolución del problema de detección y clasificación de los puntos principales de una naranja mediante la aplicación de redes neuronales convolucionales.

Para ello, en primer lugar, se ha hecho un repaso de los conceptos teóricos en los que se fundamenta la solución escogida, desde los aspectos más básicos hasta los detectores de objetos en imágenes.

Para el desarrollo del código se ha empleado el lenguaje de programación Python, usando un entorno de ejecución web como es *Google Colab*. Además, se han empleado diferentes librerías, destacando *PyTorch* para el desarrollo de la red neuronal.

Para obtener una solución óptima se ha realizado un estudio de los parámetros que afectan al entrenamiento de las redes modificándolos hasta llegar a dicha solución.

Un sistema de detección de puntos principales en naranjas requiere de un modelo de detección robusto con una gran cantidad de imágenes etiquetadas correctamente. Para simplificar los costes computacionales se redujo el dataset original que constaba de 7730 imágenes empleando 2928 imágenes. Las razones fueron que el resto de imágenes habían sido etiquetadas por otros etiquetadores además de que incluir el dataset completo ralentizaría en gran medida el proceso de aprendizaje llegando incluso a impedirlo debido a los límites de *Google Colab*.

Como conclusión global, este trabajo plantea una solución válida al problema de detección de puntos principales en naranjas, tal y como se puede observar en el apartado 6.3, las imágenes son etiquetadas satisfactoriamente por la red.

Algunos de los trabajos futuros planteados se presentan a continuación.

7.1 Trabajos futuros

Uno de los trabajos futuros sería la homogenización de criterios de etiquetado entre etiquetadores y selección de las etiquetas correctas para poder entrenar la red con el dataset completo de 7730 imágenes.

Otra opción para el aumento de datos sería la aplicación de la técnica de *Data augmentation* comentada en la sección 3.3.7, usando por ejemplo *Horizontal Flip*.

Uno de los problemas encontrados durante el entrenamiento fueron los límites de la web *Google Colab*. Por ello, una mejora sería la ejecución del entrenamiento en un entorno de ejecución local. Con esta mejora, podríamos aumentar el tamaño del *mini-batch* y reducir el ruido que se obtiene actualizando los parámetros con un *mini-batch* pequeño.

El último trabajo futuro a realizar sería la implantación de este sistema de clasificación en una línea de un almacén de naranjas para automatizar el proceso de clasificación hasta ahora manual.

Parte II

Presupuesto

Capítulo 8

Presupuesto

El presupuesto consiste en una valoración económica del proyecto "Diseño, implementación y evaluación de una red neuronal convolucional para la detección de puntos principales en naranjas".

Para la elaboración de este capítulo se ha consultado el documento recomendado por la UPV: Recomendaciones en la elaboración de presupuestos en actividades de I+D+I"[57].

El presupuesto se compone de dos conceptos: costes de personal y costes de material inventariable.

8.1 Coste de personal

El coste de cada participante en el proyecto se calcula según la ecuación 8.1.

$$Coste\ (€) = Coste\ horario\ (€/h) \cdot Dedicación\ (h) \tag{8.1}$$

8.2 Material inventariable

La amortización de los equipos que se han empleado en el proyecto se calcula según la ecuación 8.2.

$$Coste\ (€) = \frac{Tiempo\ de\ uso\ (meses) \cdot Coste\ del\ equipo\ (€)}{Periodo\ de\ amortización\ (años) \cdot 12} \tag{8.2}$$

Tabla 8.1: Amortizaciones

Clasificación económica del gasto	Amortización (años)
Adquisición de equipo para procesos de información	6
Adquisición de aplicaciones informáticas	6

8.3 Costes totales

Tabla 8.2: Coste de personal

Coste de personal

Denominación de personal	Precio (€)	Cantidad (horas)	Total (€)
Tutor	51.4	40	2056
Graduado en Ingeniería de Tecnologías Industriales	20	300	6000
		Total personal:	8056

Tabla 8.3: Coste de material inventariable

Coste de material inventariable

Concepto	Precio (€)	Amortización (meses)	Tiempo de uso (meses)	Cantidad	Total (€)
Ordenador portátil	610	6	8	1	67.78
Fibra óptica	30	6	8	1	3.33
Windows 10	116	6	8	1	12.89
Costes indirectos				10 %	8.4
				Total material inventariable:	92.4

Tabla 8.4: Presupuesto de ejecución material

Concepto	Coste Total (€)
Coste de personal	8056
Coste de material inventariable	92.4
Presupuesto de Ejecución Material	8148.4

Tabla 8.5: Presupuesto de ejecución material

		Importe
Presupuesto de Ejecución Material		8148.4 €
Gastos generales (13 %)		1059.29 €
Beneficio industrial (6 %)		488.90 €
Presupuesto de Ejecución por Contrata		9696.59 €
IVA (21 %)		2036.28 €
	Presupuesto de Base de Licitación	11726.00 €

El coste total del proyecto asciende a ONCE MIL SETECIENTOS VEINTISEIS EUROS.

Bibliografía

[1] Ligdi González. Diferencia entre inteligencia artificial, machine learning y deep learning. https://ligdigonzalez.com/diferencia-entre-inteligencia-artificial-machine-learning-deep-learning/, 2018. Accedido el 18-6-2020.

[2] *Apuntes curso "Deep Learning aplicado al análisis de señales e imágenes", CFP Universidad Politécnica de Valencia*, 2020.

[3] Red neuronal monocapa. https://sistemasinteligentesuss.blogspot.com/2016/07/redes-neuronales-las-redesneuronales.html, 2016. Accedido el 22-6-2020.

[4] Diego Calvo. Clasificación de redes neuronales artificiales. https://www.diegocalvo.es/clasificacion-de-redes-neuronales-artificiales/, 2017. Accedido el 23-6-2020.

[5] Overfitting. https://www.aprendemachinelearning.com/que-es-overfitting-y-underfitting-y-como-solucionarlo/, 2017. Accedido el 6-8-2020.

[6] Dropout. https://sitiobigdata.com/2019/12/24/maquina-de-aprendizaje-de-python-keras/#, 2019. Accedido el 10-8-2020.

[7] Diego Calvo. Red neuronal convolucional. https://www.diegocalvo.es/red-neuronal-convolucional/, 2017. Accedido el 3-7-2020.

[8] Antonio José Sánchez Salmerón. *Apuntes Antonio José Sánchez Salmerón*, 2020.

[9] Fast r-cnn. https://www.geeksforgeeks.org/r-cnn-vs-fast-r-cnn-vs-faster-r-cnn-ml/, 2020. Accedido el 12-5-2020.

[10] Shaoqing Ren; Kaiming He; Ross Girshick; Jian Sun. Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks. Jan 2016.

[11] Shilpa Ananth. Faster r-cnn for object detection. https://towardsdatascience.com/faster-r-cnn-for-object-detection-a-technical-summary-474c5b857b46, 2019. Accedido el 13-5-2020.

[12] Richard Burton Nimesh Patel Leonardo Araujo Iffat Zafar, Giounona Tzanidou. Hands-on convolutional neural networks with tensorflow. https://www.oreilly.com/library/view/

hands-on-convolutional-neural/9781789130331/a0267a8a-bd4a-452a-9e5a-8b276d7787a0.xhtml, 2018. Accedido el 21-8-2020.

[13] Evaluating object detection models: Guide to performance metrics. https://manalelaidouni.github.io/manalelaidouni.github.io/Evaluating-Object-Detection-Models-Guide-to-Performance-Metrics.html#:~:text=Average%20recall%20describes%20the%20area,recall%20on%20the%20y%2Daxis.&text=Similarly%20to%20mAP%2C%20mAR%20is,of%20classes%20within%20the%20dataset., 2019. Accedido el 21-8-2020.

[14] Wilmer Rivas Asanza; Bertha Mazón Olivo. *Redes Neuronales Artificiales aplicadas al reconocimiento de patrones*, 2018.

[15] Raúl Benítez; Gerard Escudero; Samir Kanaan; David Masip Rodó. *Inteligencia Artificial Avanzada*, 2013.

[16] Fran Ramírez. Historia de la ia. https://empresas.blogthinkbig.com/historia-de-la-ia-frank-rosenblatt-y-e/, 2018. Accedido el 20-4-2020.

[17] Juan Ignacio Bagnato. Breve historia de las redes neuronales artificiales. https://www.aprendemachinelearning.com/breve-historia-de-las-redes-neuronales-artificiales/, 2018. Accedido el 20-4-2020.

[18] Agusti M. Sánchez A. y Rodas A. Benlloch, J. V. Colour segmentation techniques for detecting weed patches in cereal crops. *Proc. of Fourth Workshop on Robotics in Agriculture and the Food-industry (pp. 30-31)*, Octubre 1995.

[19] Albarracin W. Grau R. Ricolfe C. y Barat J. M. Sánchez, A. J. Control of ham salting by using image segmentation. *Food Control, 19(2), 135-142*, 2008.

[20] C. Sánchez, A. y Ramos. Seguimiento visual de objetos utilizando técnicas de predicción. *XXI Jornadas de Automática*, 2000.

[21] J. A. Sánchez, A. y Marchant. Fusing 3D information for crop/weeds classification. *In Proceedings 15th International Conference on Pattern Recognition ICPR-2000 (Vol. 4, pp. 295-298) IEEE*, 2000.

[22] A. J. Ricolfe-Viala, C. y Sánchez-Salmerón. Optimal conditions for camera calibration using a planar template. *In 2011 18th IEEE International Conference on Image Processing (pp. 853-856) IEEE*, Septiembre 2011.

[23] Amat S. V.-Sánchez A. J. Barat J. M. y Grau R. Ivorra, E. Continuous monitoring of bread dough fermentation using a 3D vision Structured Light technique. *Journal of Food Engineering, 130, 8-13*, 2014.

[24] Sánchez A. J.-Camarasa J. G. Diago M. P. y Tardaguila J. Ivorra, E. Assessment of grape cluster yield components based on 3D descriptors using stereo vision. *Food Control, 50, 273-282*, 2015.

[25] Ivorra E. Sánchez-A. J. Barat J. M. y Grau R. Verdú, S. Relationship between fermentation behaviour measured with a 3D vision Structured Light technique and the internal structure of bread. *Journal of Food Engineering, 146, 227-233*, 2015.

[26] Sánchez A. J.-Girón J. Iborra E. Fuentes A. y Barat J. M. Grau, R. Nondestructive assessment of freshness in packaged sliced chicken breasts using SW-NIR spectroscopy. *Food Research International, 44(1), 331-337*, 2011.

[27] Sánchez A. J.-Verdú S. Barat. J. M. y Grau R. Ivorra, E. Shelf life prediction of expired vacuum-packed chilled smoked salmon based on a KNN tissue segmentation method using hyperspectral images. *Journal of Food Engineering, 178, 110-116*, 2016.

[28] Ivorra E. Sánchez-A. J. Barat. J. M. y Grau R. Verdú, S. Study of high strength wheat flours considering their physicochemical and rheological characterisation as well as fermentation capacity using SW-NIR imaging. *Journal of Cereal Science, 62, 31-37*, 2015.

[29] Salmerón A. J. S. Benimeli F. Berti, E. M. Kalman filter for tracking robotic arms using low cost 3D vision systems. *The Fifth International Conference on Advances in Computer-Human Interactions, Valencia, Spain, Jan. 30 - Feb. 4, pp. 236-240*, 2012.

[30] Salmerón A. J. S. Benimeli F. Berti, E. M. Human-Robot Interaction and Tracking Using low cost 3D Vision Systems. *Romanian J. Tech. Sci. Appl. Mech. 2012, 7, 1-15*, 2012.

[31] Sánchez A. Ricolfe C. Nina O. Martínez, E. Human Pose Estimation for RGBD Imagery with Multi-Channel Mixture of Parts and Kinematic Constraints. *WSEAS Trans. Comput. 2016, 15, 279-286*, 2016.

[32] Sánchez-Salmerón A. Ricolfe-Viala C. Martínez, E. 4D-DPM model for pose estimation using Kalman filter constraints. *Int. J. Adv. Robot. Syst., 14, 1-13*, 2017.

[33] Nina-O. Sánchez A.-Ricolfe C. Martínez, E. Optimized 4D-DPM for Pose Estimation on RGBD Channels using polisphere models. *In Proceedings of the 12th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications, Porto, Portugal, 27 February - 1 March 2017; Volume 5, pp. 281-288*, 2017.

[34] Sánchez-Salmerón A. J. Ricolfe-Viala C. Martínez-Berti, E. Dual quaternions as constraints in 4D-DPM models for pose estimation. *Sensors (Switzerland)*, 2017.

[35] David Kriesel. *A Brief Introduction to Neural Networks*, 2007.

[36] Loss functions. https://ml-cheatsheet.readthedocs.io/en/latest/loss_functions.html#cross-entropy, 2017. Accedido el 1-9-2020.

[37] Michael Nielsen. *Neural Networks and Deep Learning*, 2019.

[38] Jaime Durán. Descenso del gradiente aplicado a redes neuronales. https://medium.com/metadatos/. Accedido el 6-8-2020.

[39] Simeon Kostadinov. Backpropagation. https://towardsdatascience.com/understanding-backpropagation-algorithm-7bb3aa2f95fd, 2019. Accedido el 3-8-2020.

[40] Álvaro Gonzalo. Overfitting. https://machinelearningparatodos.com/que-es-el-sobreajuste-u-overfitting-y-por-que-debemos-evitarlo/, 2020. Accedido el 6-8-2020.

[41] Jaime Durán. Técnicas de regularización básicas para redes neuronales. https://medium.com/metadatos, 2019. Accedido el 4-9-2020.

[42] Ayoosh Kathuria. Data augmentation for bounding boxes: Rethinking image transforms for object detection. https://blog.paperspace.com/data-augmentation-for-bounding-boxes/, 2018. Accedido el 27-8-2020.

[43] Sarahí Silva y Estefanía Freire. Introducción a las redes neuronales convolucionales. https://medium.com/@bootcampai/redes-neuronales-convolucionales-5e0ce960caf8, 2019. Accedido el 6-7-2020.

[44] Yoshua Bengio y Aaron Courville Ian Goodfellow. Deep learning. http://www.deeplearningbook.org/, 2016. Accedido el 7-7-2020.

[45] James Cowley. Técnicas de regularización básicas para redes neuronales. https://developer.ibm.com/es/technologies/artificial-intelligence/articles/cc-convolutional-neural-network-vision-recognition/, 2018. Accedido el 3-9-2020.

[46] Innovaciones arquitectónicas en redes neuronales. https://sitiobigdata.com/2019/05/01/innovaciones-arquitectonicas-redes-neuronales-clasificacion-imagenes/, 2019. Accedido el 20-8-2020.

[47] Shilpa Ananth. Fast r-cnn for object detection. https://towardsdatascience.com/fast-r-cnn-for-object-detection-a-technical-summary-a0ff94faa022, 2019. Accedido el 12-5-2020.

[48] Francisco Massa. Github pytorch detection. https://github.com/pytorch/vision/tree/v0.3.0/references/detection, 2019. Accedido el 15-4-2020.

[49] Coco. https://cocodataset.org/#home, 2020. Accedido el 27-4-2020.

[50] Numpy. https://numpy.org/about/, 2019-2020. Accedido el 15-4-2020.

[51] Pil. https://www.pythonware.com/products/pil/, 2019-2020. Accedido el 15-4-2020.

[52] Pytorch. https://pytorch.org/, 2019-2020. Accedido el 10-4-2020.

[53] Cleverpy. Pytorch. https://cleverpy.com/que-es-pytorch-y-como-se-instala/, 2019-2020. Accedido el 10-4-2020.

[54] Deep learning. https://es.mathworks.com/discovery/deep-learning.html. Accedido el 24-8-2020.

[55] The confusing metrics of ap and map for object detection. https://mc.ai/the-confusing-metrics-of-ap-and-map-for-object-detection/, 2018. Accedido el 21-8-2020.

[56] Shaoqing Ren Jian Sun Kaiming He, Xiangyu Zhang. Deep Residual Learning for Image Recognition. Diciembre 2005.

[57] UPV. Recomendaciones en la elaboración de presupuestos en actividades de I+D+I. 2018.

Anexos

1. Librerías

```
import torch.utils.data
import torchvision
from torchvision import transforms
from torchvision.models.detection.faster_rcnn import FastRCNNPredictor
import PIL
from PIL import Image
import os
from lxml import etree
import re
import numpy as np
```

```
%%shell
pip install cython
pip install -U 'git+https://github.com/Melendez95/PyCocoTools.git#subdirectory=PythonAPI'
```

Ficheros .py locales

```
# We'll start by creating a directory in which we'll define our new
# module to be imported.
!mkdir -p local_modules/demo_module
```

```
%%writefile local_modules/demo_module/__init__.py
# Save a module init file that contains a custom function that we'll use
# to verify that import works.

def SomeFunction():
  return 'Function from a local module'
```

```
# Add the local_modules directory to the set of paths
# Python uses to look for imports.
import sys
sys.path.append('local_modules')
```

```
# Now we can import our new module and call our function.
import demo_module
demo_module.SomeFunction()
```

```
from google.colab import files
import zipfile, io, os
```

```
def upload_dir_file(case_f):
    # author: yasser mustafa, 21 March 2018
    # case_f = 0 for uploading one File or Package(.py) and case_f = 1 for uploading one Zipped Direc
tory
    uploaded = files.upload()    # to upload a Full Directory, please Zip it first (use WinZip)
    for fn in uploaded.keys():
        name = fn  #.encode('utf-8')
        #print('\nfile after encode', name)
        #name = io.BytesIO(uploaded[name])
    if case_f == 0:    # case of uploading 'One File only'
        print('\n file name: ', name)
        return name
    else:   # case of uploading a directory and its subdirectories and files
        zfile = zipfile.ZipFile(name, 'r')   # unzip the directory
        zfile.extractall()
        for d in zfile.namelist():   # d = directory
            print('\n main directory name: ', d)
            return d
print('Done!')
```

```
file_name = upload_dir_file(0)
```

```
Saving coco_eval.py to coco_eval.py
Saving coco_utils.py to coco_utils.py
Saving engine.py to engine.py
Saving group_by_aspect_ratio.py to group_by_aspect_ratio.py
Saving train.py to train.py
Saving transforms.py to transforms.py
Saving utils.py to utils.py
```

```
 file name:  utils.py
```

```
from engine import train_one_epoch, evaluate
import utils
import engine
import coco_eval
import coco_utils
import group_by_aspect_ratio
import train
import transforms as T
```

2. Abrir archivos drive

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

3. Leer de .xml

```
#Listamos todos los ficheros .xml que existen en la carpeta Annotations

annotations_dir=f"/content/gdrive/My Drive/Naranjas/dataset_new/Annotations"

pedicel = [] #Almacenar las coordenadas de las cajas
bottom = []
defects = []
annots = []
x=0

for i in os.listdir(annotations_dir):
  carpetas_in_annotations=annotations_dir + '/' + i
  #print(carpetas_in_annotations)
  for i in os.listdir(carpetas_in_annotations):
    carpeta_in_carpetafecha_ann=carpetas_in_annotations+'/'+i
    #print(carpeta_in_carpetafecha_ann)
    for i in os.listdir(carpeta_in_carpetafecha_ann):
      annotations=carpeta_in_carpetafecha_ann+'/'+i
      annots.append(annotations)
      print(annotations)
      doc=etree.parse(annotations)
      raiz=doc.getroot()
      x=x+1
      print(x)
      for i in range(len(raiz)):
        etiqueta=raiz[i]
        print (etiqueta.tag + ": " + raiz[i].text)  #etiqueta.tag = pedicel, bottom, defect, raiz[i].
text = números siendo i=1, pedicel, i=2 bottom e i=3 defecto
        if i==1:
          pedicel.append(raiz[i].text)
        if i==2:
          bottom.append(raiz[i].text)
        if i==3:
          defects.append(raiz[i].text)

#4 números por cada caja, (x1,y1,x2-x1,y2-y1) Eje X de izq a dcha, eje Y de arriba a abajo
```

4. Listar imágenes

```
#Listamos todos los ficheros .ppm que existen en la carpeta Images

images_dir=f"/content/gdrive/My Drive/Naranjas/dataset_new/Images"
imgs=[] #Tupla para guardar los directorios de las imágenes
imgs_prueba=[]

for i in os.listdir(images_dir):
  carpetas_in_images=images_dir + '/' + i
  #print(carpetas_in_images)
  for i in os.listdir(carpetas_in_images):
    carpeta_in_carpetafecha=carpetas_in_images+'/'+i
    imgs_prueba.append(carpeta_in_carpetafecha)
    #print(carpeta_in_carpetafecha)
    for i in os.listdir(carpeta_in_carpetafecha):
      imagenes=carpeta_in_carpetafecha+'/'+i
      imgs.append(imagenes)
      print(imagenes)
```

5. Defining the Dataset

Diccionario:

* Cajas: FloatTensor[N,4], coordenadas de las N cajas [x0, y0, x1, y1] de 0 a anchura y de 0 a altura.
* Área caja: Tensor[N]
* Etiquetas para cada caja: Int64Tensor[N] (Culo, pedicel y defecto)
* Id imagen
* is crowd

```
#Class for a dataset object


class Naranjas(torch.utils.data.Dataset):
  def __init__(self, root, transforms=None):
```

```
    self.root = root
    self.transforms = transforms
    self.imgs = imgs

  def __getitem__(self,idx):
    img_path = self.imgs[idx]
    img = Image.open(img_path).convert("RGB")

    if self.transforms is not None:
      img = self.transforms(img)

    #Extraer las cajas
    boxes1 = []
    annot=img_path.replace('Images','Annotations')
    annot_path=annot.replace('.ppm','.xml')
    doc=etree.parse(annot_path)
    raiz=doc.getroot()
    for i in range(len(raiz)):
      etiqueta=raiz[i] #etiqueta = pedicel, bottom, defect
      boxes1.append(raiz[i].text)

    boxes=[]
    labels = [] #1 pedicel #2 bottom #3 defect

    for ind in range(3):
      if (len(boxes1[ind])>2):
        p=re.findall(r'\d+', boxes1[ind]) #Extraer solo los números
        nboxes=int(len(p)/4) #El número de boxes será el número de elemntos entre 4
        for i in range(nboxes):
          xmin=p[4*i]
          ymin=p[1+4*i]
          xmax=int(xmin)+int(p[2+4*i])
          ymax=int(ymin)+int(p[3+4*i])
          boxes.append([int(xmin), int(ymin), int(xmax), int(ymax)])
          labels.append(ind+1)

    boxes = torch.as_tensor(boxes, dtype=torch.float32)
    boxes = boxes.reshape(-1, 4)
    labels = torch.as_tensor(labels, dtype=torch.int64)
    image_id = torch.tensor([idx])

    area = []
    for i in range(len(boxes)):
      area.append((int(boxes[i][3]) - int(boxes[i][1])) * (int(boxes[i][2]) - int(boxes[i][0])))

    area = torch.as_tensor(area, dtype=torch.float32)

    num_objs = len(area)
    iscrowd = torch.zeros((num_objs,), dtype=torch.int64)


    target = {}
    target["boxes"] = boxes
    target["labels"] = labels
    target["image_id"] = image_id
    target["area"] = area
    target["iscrowd"] = iscrowd

    return img, target

  def __len__(self):
    return len(self.imgs)
```

##6. Defining the model

```
def get_instance_segmentation_model(num_classes):
    # load an instance segmentation model pre-trained on COCO
    model = torchvision.models.detection.fasterrcnn_resnet50_fpn(pretrained=True)

    # get the number of input features for the classifier
    in_features = model.roi_heads.box_predictor.cls_score.in_features
    # replace the pre-trained head with a new one
    model.roi_heads.box_predictor = FastRCNNPredictor(in_features, num_classes)

    return model
```

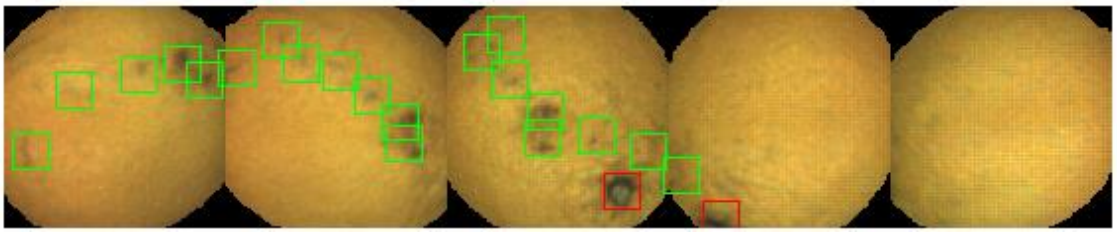
##7. Training and evaluation functions

```
mean = np.array([0.5, 0.5, 0.5])
std = np.array([0.5, 0.5, 0.5])

# Se transforman las imágenes en tensor y se aumenta el conjunto de datos:
data_transforms = {
    'train': transforms.Compose([
```

```
        transforms.ToTensor(),  # Transformar en tensor
        transforms.Normalize(mean, std)  # Normalizar (recomendable)
    ]),

    'test': transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize(mean, std)
    ]),

}
```

##8. Putting everything together

```
dataset = Naranjas(f"/content/gdrive/My Drive/Naranjas/dataset_new/",data_transforms['train'])
dataset_test = Naranjas(f"/content/gdrive/My Drive/Naranjas/dataset_new/",data_transforms['test'])

img, target = dataset[1735]
mean1=0.5;
std1=0.5;
img=img*std1+mean1;
img=img.mul(255).permute(1, 2, 0).byte().numpy()
Image.fromarray(img)
img = img[:, :, ::-1].copy()

import cv2
from google.colab.patches import cv2_imshow

for i in range(len(target['boxes'])):

  x1=int(target['boxes'][i][0])
  y1=int(target['boxes'][i][1])
  x2=int(target['boxes'][i][2])
  y2=int(target['boxes'][i][3])

  if int(target['labels'][i])==1:
    color=(0,0,255)
  if int(target['labels'][i])==2:
    color=(255,0,0)
  if int(target['labels'][i])==3:
    color=(0,255,0)
  img = cv2.rectangle(img,(x1,y1),(x2,y2),color,1)
cv2_imshow(img)
```



Dividir el dataset en train y test

```
torch.manual_seed(1)
indices = torch.randperm(len(dataset)).tolist()
dataset = torch.utils.data.Subset(dataset, indices[:-586]) #Todos hasta los últimos 586
dataset_test = torch.utils.data.Subset(dataset_test, indices[-586:]) #Últimos 586 (20%)
```

DataLoaders

```
data_loader = torch.utils.data.DataLoader(
    dataset, batch_size=4, shuffle=True, num_workers=4,
    collate_fn=utils.collate_fn)

data_loader_test = torch.utils.data.DataLoader(
    dataset_test, batch_size=4, shuffle=False, num_workers=4,
    collate_fn=utils.collate_fn)

device = torch.device('cuda') if torch.cuda.is_available() else torch.device('cpu')

# our dataset has three classes only - pedicel, defect and bottom
num_classes = 4

# get the model using our helper function
model = get_instance_segmentation_model(num_classes)
#model.load_state_dict(torch.load(f"/content/gdrive/My Drive/Naranjas/Red-entrenada.pt"))
# move model to the right device
model.to(device)

# construct an optimizer
```

```
params = [p for p in model.parameters() if p.requires_grad]
optimizer = torch.optim.SGD(params, lr=0.005,
                            momentum=0.9, weight_decay=0.005)

# and a learning rate scheduler which decreases the learning rate by
# 10x every 3 epochs
lr_scheduler = torch.optim.lr_scheduler.StepLR(optimizer,
                                               step_size=3,
                                               gamma=0.1)

# let's train it for 10 epochs
num_epochs = 10
min = np.Inf
losses_tuple = []

for epoch in range(num_epochs):
    # train for one epoch, printing every 100 iterations
    training, loss = train_one_epoch(model, optimizer, data_loader, device, epoch, print_freq=100)

    losses_tuple.append(loss)

    # update the learning rate
    lr_scheduler.step()
    # evaluate on the test dataset
    evaluation = evaluate(model, data_loader_test, device=device)

    if np.mean(evaluation.coco_eval['bbox'].stats[0]) <= min:
      epoch_best_model = epoch
      torch.save(model.state_dict(), f"/content/gdrive/My Drive/Naranjas/Red-entrenada.pt")
      min = np.mean(evaluation.coco_eval['bbox'].stats[0])
```

```
Epoch: [0]  [  0/586]  eta: 0:06:00  lr: 0.000014  loss: 3.3686 (3.3686)  loss_classifier: 1.7902
(1.7902)  loss_box_reg: 0.0363 (0.0363)  loss_objectness: 1.4828 (1.4828)  loss_rpn_box_reg: 0.0593
(0.0593)  time: 0.6155  data: 0.2634  max mem: 2652
Epoch: [0]  [100/586]  eta: 0:02:41  lr: 0.000867  loss: 0.8658 (1.2613)  loss_classifier: 0.3677
(0.5066)  loss_box_reg: 0.4082 (0.3234)  loss_objectness: 0.0737 (0.3876)  loss_rpn_box_reg: 0.0219
(0.0437)  time: 0.3283  data: 0.0101  max mem: 2653
Epoch: [0]  [200/586]  eta: 0:02:07  lr: 0.001721  loss: 0.6866 (1.0119)  loss_classifier: 0.2761
(0.4032)  loss_box_reg: 0.3254 (0.3431)  loss_objectness: 0.0540 (0.2343)  loss_rpn_box_reg: 0.0174
(0.0313)  time: 0.3284  data: 0.0098  max mem: 2653
Epoch: [0]  [300/586]  eta: 0:01:34  lr: 0.002575  loss: 0.6911 (0.8898)  loss_classifier: 0.2489
(0.3519)  loss_box_reg: 0.3404 (0.3359)  loss_objectness: 0.0385 (0.1752)  loss_rpn_box_reg: 0.0223
(0.0268)  time: 0.3289  data: 0.0103  max mem: 2653
Epoch: [0]  [400/586]  eta: 0:01:01  lr: 0.003429  loss: 0.4928 (0.8125)  loss_classifier: 0.2063
(0.3187)  loss_box_reg: 0.2596 (0.3236)  loss_objectness: 0.0391 (0.1459)  loss_rpn_box_reg: 0.0162
(0.0243)  time: 0.3276  data: 0.0097  max mem: 2653
Epoch: [0]  [500/586]  eta: 0:00:28  lr: 0.004283  loss: 0.4314 (0.7599)  loss_classifier: 0.1794
(0.2960)  loss_box_reg: 0.2001 (0.3130)  loss_objectness: 0.0342 (0.1279)  loss_rpn_box_reg: 0.0136
(0.0231)  time: 0.3285  data: 0.0096  max mem: 2653
Epoch: [0]  [585/586]  eta: 0:00:00  lr: 0.005000  loss: 0.4524 (0.7300)  loss_classifier: 0.1906
(0.2832)  loss_box_reg: 0.2395 (0.3069)  loss_objectness: 0.0285 (0.1176)  loss_rpn_box_reg: 0.0127
(0.0223)  time: 0.3204  data: 0.0100  max mem: 2653
Epoch: [0] Total time: 0:03:12 (0.3287 s / it)
creating index...
index created!
Test:  [  0/147]  eta: 0:00:55  model_time: 0.1663 (0.1663)  evaluator_time: 0.0148 (0.0148)  time:
0.3784  data: 0.1950  max mem: 2653
Test:  [100/147]  eta: 0:00:09  model_time: 0.1422 (0.1429)  evaluator_time: 0.0316 (0.0379)  time:
0.2066  data: 0.0098  max mem: 2653
Test:  [146/147]  eta: 0:00:00  model_time: 0.1418 (0.1423)  evaluator_time: 0.0263 (0.0367)  time:
0.1784  data: 0.0091  max mem: 2653
Test: Total time: 0:00:28 (0.1923 s / it)
Averaged stats: model_time: 0.1418 (0.1423)  evaluator_time: 0.0263 (0.0367)
Accumulating evaluation results...
DONE (t=0.22s).
IoU metric: bbox
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.293
 Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.649
 Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.207
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.293
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.000
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = -1.000
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.151
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.414
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.471
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.471
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.000
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = -1.000

 Mean Average Precission (Validation index) --> 0.2882

Epoch: [1]  [  0/586]  eta: 0:05:43  lr: 0.005000  loss: 0.2447 (0.2447)  loss_classifier: 0.0757
(0.0757)  loss_box_reg: 0.1587 (0.1587)  loss_objectness: 0.0056 (0.0056)  loss_rpn_box_reg: 0.0047
(0.0047)  time: 0.5869  data: 0.2567  max mem: 2653
Epoch: [1]  [100/586]  eta: 0:02:41  lr: 0.005000  loss: 0.4767 (0.5260)  loss_classifier: 0.1686
(0.1985)  loss_box_reg: 0.2471 (0.2622)  loss_objectness: 0.0431 (0.0487)  loss_rpn_box_reg: 0.0140
(0.0165)  time: 0.3269  data: 0.0097  max mem: 2653
```

```
Epoch: [1]  [200/586]  eta: 0:02:07  lr: 0.005000  loss: 0.5298 (0.5200)  loss_classifier: 0.1980
(0.1972)  loss_box_reg: 0.2595 (0.2602)  loss_objectness: 0.0280 (0.0465)  loss_rpn_box_reg: 0.0144
(0.0161)  time: 0.3281  data: 0.0097  max mem: 2653
Epoch: [1]  [300/586]  eta: 0:01:34  lr: 0.005000  loss: 0.4381 (0.5128)  loss_classifier: 0.1611
(0.1938)  loss_box_reg: 0.2300 (0.2578)  loss_objectness: 0.0360 (0.0453)  loss_rpn_box_reg: 0.0129
(0.0159)  time: 0.3282  data: 0.0098  max mem: 2653
Epoch: [1]  [400/586]  eta: 0:01:01  lr: 0.005000  loss: 0.5249 (0.5165)  loss_classifier: 0.2126
(0.1935)  loss_box_reg: 0.2951 (0.2614)  loss_objectness: 0.0317 (0.0454)  loss_rpn_box_reg: 0.0185
(0.0163)  time: 0.3271  data: 0.0098  max mem: 2653
Epoch: [1]  [500/586]  eta: 0:00:28  lr: 0.005000  loss: 0.5321 (0.5140)  loss_classifier: 0.1985
(0.1926)  loss_box_reg: 0.2497 (0.2604)  loss_objectness: 0.0460 (0.0450)  loss_rpn_box_reg: 0.0142
(0.0160)  time: 0.3277  data: 0.0098  max mem: 2653
Epoch: [1]  [585/586]  eta: 0:00:00  lr: 0.005000  loss: 0.5123 (0.5162)  loss_classifier: 0.1859
(0.1936)  loss_box_reg: 0.2522 (0.2607)  loss_objectness: 0.0523 (0.0459)  loss_rpn_box_reg: 0.0158
(0.0159)  time: 0.3195  data: 0.0099  max mem: 2653
Epoch: [1] Total time: 0:03:12 (0.3284 s / it)
creating index...
index created!
Test:  [  0/147]  eta: 0:00:54  model_time: 0.1634 (0.1634)  evaluator_time: 0.0201 (0.0201)  time:
0.3705  data: 0.1845  max mem: 2653
Test:  [100/147]  eta: 0:00:10  model_time: 0.1424 (0.1435)  evaluator_time: 0.0525 (0.0596)  time:
0.2271  data: 0.0095  max mem: 2653
Test:  [146/147]  eta: 0:00:00  model_time: 0.1419 (0.1428)  evaluator_time: 0.0417 (0.0576)  time:
0.1965  data: 0.0093  max mem: 2653
Test: Total time: 0:00:31 (0.2138 s / it)
Averaged stats: model_time: 0.1419 (0.1428)  evaluator_time: 0.0417 (0.0576)
Accumulating evaluation results...
DONE (t=0.34s).
IoU metric: bbox
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.299
 Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.649
 Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.215
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.299
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.004
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = -1.000
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.151
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.411
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.475
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.475
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.150
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = -1.000

 Mean Average Precission (Validation index) --> 0.2931

Epoch: [2]  [  0/586]  eta: 0:05:32  lr: 0.005000  loss: 0.4983 (0.4983)  loss classifier: 0.1726
(0.1726)  loss_box_reg: 0.2232 (0.2232)  loss_objectness: 0.0869 (0.0869)  loss_rpn_box_reg: 0.0155
(0.0155)  time: 0.5672  data: 0.2173  max mem: 2653
Epoch: [2]  [100/586]  eta: 0:02:40  lr: 0.005000  loss: 0.5319 (0.5121)  loss_classifier: 0.2047
(0.1894)  loss_box_reg: 0.2771 (0.2680)  loss_objectness: 0.0292 (0.0385)  loss_rpn_box_reg: 0.0160
(0.0162)  time: 0.3277  data: 0.0098  max mem: 2653
Epoch: [2]  [200/586]  eta: 0:02:07  lr: 0.005000  loss: 0.4312 (0.5049)  loss_classifier: 0.1659
(0.1881)  loss_box_reg: 0.2271 (0.2592)  loss_objectness: 0.0254 (0.0421)  loss_rpn_box_reg: 0.0108
(0.0156)  time: 0.3272  data: 0.0097  max mem: 2653
Epoch: [2]  [300/586]  eta: 0:01:33  lr: 0.005000  loss: 0.4838 (0.5050)  loss_classifier: 0.1763
(0.1903)  loss_box_reg: 0.2399 (0.2573)  loss_objectness: 0.0375 (0.0424)  loss_rpn_box_reg: 0.0153
(0.0150)  time: 0.3271  data: 0.0098  max mem: 2653
Epoch: [2]  [400/586]  eta: 0:01:01  lr: 0.005000  loss: 0.4634 (0.5043)  loss classifier: 0.1768
(0.1892)  loss box reg: 0.2429 (0.2584)  loss objectness: 0.0319 (0.0416)  loss_rpn_box_reg: 0.0150
(0.0151)  time: 0.3275  data: 0.0099  max mem: 2653
Epoch: [2]  [500/586]  eta: 0:00:28  lr: 0.005000  loss: 0.5582 (0.5092)  loss_classifier: 0.2009
(0.1910)  loss_box_reg: 0.2845 (0.2603)  loss_objectness: 0.0270 (0.0426)  loss_rpn_box_reg: 0.0147
(0.0153)  time: 0.3287  data: 0.0103  max mem: 2653
Epoch: [2]  [585/586]  eta: 0:00:00  lr: 0.005000  loss: 0.5589 (0.5085)  loss_classifier: 0.2048
(0.1907)  loss_box_reg: 0.2814 (0.2599)  loss_objectness: 0.0393 (0.0424)  loss_rpn_box_reg: 0.0177
(0.0155)  time: 0.3193  data: 0.0094  max mem: 2653
Epoch: [2] Total time: 0:03:12 (0.3281 s / it)
creating index...
index created!
Test:  [  0/147]  eta: 0:00:54  model_time: 0.1621 (0.1621)  evaluator_time: 0.0169 (0.0169)  time:
0.3707  data: 0.1877  max mem: 2653
Test:  [100/147]  eta: 0:00:09  model_time: 0.1421 (0.1428)  evaluator_time: 0.0357 (0.0436)  time:
0.2105  data: 0.0098  max mem: 2653
Test:  [146/147]  eta: 0:00:00  model_time: 0.1419 (0.1422)  evaluator_time: 0.0334 (0.0417)  time:
0.1813  data: 0.0092  max mem: 2653
Test: Total time: 0:00:28 (0.1971 s / it)
Averaged stats: model_time: 0.1419 (0.1422)  evaluator_time: 0.0334 (0.0417)
Accumulating evaluation results...
DONE (t=0.23s).
IoU metric: bbox
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.339
 Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.733
 Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.243
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.339
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.001
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = -1.000
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.167
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.439
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.494
```

```
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.494
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.008
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = -1.000

 Mean Average Precission (Validation index) --> 0.3310

Epoch: [3]  [  0/586]  eta: 0:06:06  lr: 0.000500  loss: 0.5392 (0.5392)  loss_classifier: 0.1944
(0.1944)  loss_box_reg: 0.2772 (0.2772)  loss_objectness: 0.0512 (0.0512)  loss_rpn_box_reg: 0.0164
(0.0164)  time: 0.6254  data: 0.2964  max mem: 2653
Epoch: [3]  [100/586]  eta: 0:02:40  lr: 0.000500  loss: 0.4196 (0.4614)  loss_classifier: 0.1597
(0.1743)  loss_box_reg: 0.2226 (0.2437)  loss_objectness: 0.0195 (0.0298)  loss_rpn_box_reg: 0.0108
(0.0135)  time: 0.3267  data: 0.0098  max mem: 2653
Epoch: [3]  [200/586]  eta: 0:02:06  lr: 0.000500  loss: 0.3836 (0.4590)  loss_classifier: 0.1426
(0.1716)  loss_box_reg: 0.2065 (0.2432)  loss_objectness: 0.0205 (0.0308)  loss_rpn_box_reg: 0.0103
(0.0134)  time: 0.3265  data: 0.0098  max mem: 2653
Epoch: [3]  [300/586]  eta: 0:01:33  lr: 0.000500  loss: 0.5352 (0.4678)  loss_classifier: 0.1971
(0.1728)  loss_box_reg: 0.2760 (0.2488)  loss_objectness: 0.0280 (0.0318)  loss_rpn_box_reg: 0.0171
(0.0144)  time: 0.3262  data: 0.0096  max mem: 2653
Epoch: [3]  [400/586]  eta: 0:01:00  lr: 0.000500  loss: 0.4670 (0.4677)  loss_classifier: 0.1759
(0.1714)  loss_box_reg: 0.2409 (0.2484)  loss_objectness: 0.0268 (0.0335)  loss_rpn_box_reg: 0.0117
(0.0145)  time: 0.3268  data: 0.0098  max mem: 2653
Epoch: [3]  [500/586]  eta: 0:00:28  lr: 0.000500  loss: 0.3840 (0.4617)  loss_classifier: 0.1444
(0.1686)  loss_box_reg: 0.2100 (0.2458)  loss_objectness: 0.0189 (0.0330)  loss_rpn_box_reg: 0.0113
(0.0144)  time: 0.3267  data: 0.0098  max mem: 2653
Epoch: [3]  [585/586]  eta: 0:00:00  lr: 0.000500  loss: 0.4614 (0.4602)  loss_classifier: 0.1592
(0.1680)  loss_box_reg: 0.2523 (0.2455)  loss_objectness: 0.0288 (0.0324)  loss_rpn_box_reg: 0.0141
(0.0143)  time: 0.3197  data: 0.0098  max mem: 2653
Epoch: [3] Total time: 0:03:11 (0.3275 s / it)
creating index...
index created!
Test:  [  0/147]  eta: 0:00:52  model_time: 0.1651 (0.1651)  evaluator_time: 0.0138 (0.0138)  time:
0.3596  data: 0.1781  max mem: 2653
Test:  [100/147]  eta: 0:00:09  model_time: 0.1420 (0.1428)  evaluator_time: 0.0327 (0.0406)  time:
0.2079  data: 0.0098  max mem: 2653
Test:  [146/147]  eta: 0:00:00  model_time: 0.1420 (0.1422)  evaluator_time: 0.0311 (0.0399)  time:
0.1868  data: 0.0094  max mem: 2653
Test: Total time: 0:00:28 (0.1952 s / it)
Averaged stats: model_time: 0.1420 (0.1422)  evaluator_time: 0.0311 (0.0399)
Accumulating evaluation results...
DONE (t=0.22s).
IoU metric: bbox
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.365
 Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.746
 Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.289
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.365
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.017
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = -1.000
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.177
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.458
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.513
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.513
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.108
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = -1.000

 Mean Average Precission (Validation index) --> 0.3567

Epoch: [4]  [  0/586]  eta: 0:05:49  lr: 0.000500  loss: 0.3395 (0.3395)  loss classifier: 0.1046
(0.1046)  loss box reg: 0.1969 (0.1969)  loss objectness: 0.0290 (0.0290)  loss_rpn_box_reg: 0.0091
(0.0091)  time: 0.5960  data: 0.2608  max mem: 2653
Epoch: [4]  [100/586]  eta: 0:02:40  lr: 0.000500  loss: 0.4758 (0.4555)  loss_classifier: 0.1576
(0.1630)  loss_box_reg: 0.2516 (0.2480)  loss_objectness: 0.0316 (0.0302)  loss_rpn_box_reg: 0.0161
(0.0144)  time: 0.3268  data: 0.0097  max mem: 2653
Epoch: [4]  [200/586]  eta: 0:02:06  lr: 0.000500  loss: 0.4612 (0.4534)  loss_classifier: 0.1599
(0.1636)  loss_box_reg: 0.2388 (0.2443)  loss_objectness: 0.0218 (0.0312)  loss_rpn_box_reg: 0.0136
(0.0143)  time: 0.3276  data: 0.0098  max mem: 2653
Epoch: [4]  [300/586]  eta: 0:01:33  lr: 0.000500  loss: 0.3935 (0.4496)  loss_classifier: 0.1283
(0.1618)  loss_box_reg: 0.2312 (0.2417)  loss_objectness: 0.0243 (0.0316)  loss_rpn_box_reg: 0.0120
(0.0144)  time: 0.3272  data: 0.0100  max mem: 2653
Epoch: [4]  [400/586]  eta: 0:01:00  lr: 0.000500  loss: 0.4353 (0.4532)  loss_classifier: 0.1546
(0.1626)  loss box reg: 0.2365 (0.2443)  loss objectness: 0.0305 (0.0318)  loss_rpn_box_reg: 0.0136
(0.0145)  time: 0.3271  data: 0.0099  max mem: 2653
Epoch: [4]  [500/586]  eta: 0:00:28  lr: 0.000500  loss: 0.4078 (0.4475)  loss_classifier: 0.1432
(0.1602)  loss_box_reg: 0.2200 (0.2419)  loss_objectness: 0.0205 (0.0310)  loss_rpn_box_reg: 0.0123
(0.0143)  time: 0.3268  data: 0.0099  max mem: 2653
Epoch: [4]  [585/586]  eta: 0:00:00  lr: 0.000500  loss: 0.4587 (0.4457)  loss_classifier: 0.1552
(0.1596)  loss_box_reg: 0.2455 (0.2413)  loss_objectness: 0.0213 (0.0306)  loss_rpn_box_reg: 0.0148
(0.0142)  time: 0.3195  data: 0.0097  max mem: 2653
Epoch: [4] Total time: 0:03:11 (0.3275 s / it)
creating index...
index created!
Test:  [  0/147]  eta: 0:01:02  model_time: 0.1585 (0.1585)  evaluator_time: 0.0150 (0.0150)  time:
0.4219  data: 0.2462  max mem: 2653
Test:  [100/147]  eta: 0:00:09  model_time: 0.1421 (0.1428)  evaluator_time: 0.0342 (0.0402)  time:
0.2097  data: 0.0101  max mem: 2653
Test:  [146/147]  eta: 0:00:00  model_time: 0.1419 (0.1423)  evaluator_time: 0.0285 (0.0389)  time:
0.1802  data: 0.0092  max mem: 2653
Test: Total time: 0:00:28 (0.1949 s / it)
Averaged stats: model_time: 0.1419 (0.1423)  evaluator_time: 0.0285 (0.0389)
```

```
Accumulating evaluation results...
DONE (t=0.21s).
IoU metric: bbox
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.366
 Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.759
 Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.296
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.366
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.034
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = -1.000
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.177
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.455
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.510
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.511
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.142
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = -1.000

 Mean Average Precission (Validation index) --> 0.3641

Epoch: [5]  [  0/586]  eta: 0:05:34  lr: 0.000500  loss: 0.3914 (0.3914)  loss_classifier: 0.1421
(0.1421)  loss_box_reg: 0.2149 (0.2149)  loss_objectness: 0.0228 (0.0228)  loss_rpn_box_reg: 0.0115
(0.0115)  time: 0.5712  data: 0.2315  max mem: 2653
Epoch: [5]  [100/586]  eta: 0:02:40  lr: 0.000500  loss: 0.4621 (0.4448)  loss_classifier: 0.1518
(0.1555)  loss_box_reg: 0.2413 (0.2437)  loss_objectness: 0.0241 (0.0314)  loss_rpn_box_reg: 0.0143
(0.0142)  time: 0.3272  data: 0.0098  max mem: 2653
Epoch: [5]  [200/586]  eta: 0:02:06  lr: 0.000500  loss: 0.4244 (0.4405)  loss_classifier: 0.1535
(0.1554)  loss_box_reg: 0.2359 (0.2402)  loss_objectness: 0.0195 (0.0307)  loss_rpn_box_reg: 0.0133
(0.0142)  time: 0.3267  data: 0.0097  max mem: 2653
Epoch: [5]  [300/586]  eta: 0:01:33  lr: 0.000500  loss: 0.4464 (0.4418)  loss_classifier: 0.1421
(0.1564)  loss_box_reg: 0.2606 (0.2405)  loss_objectness: 0.0229 (0.0307)  loss_rpn_box_reg: 0.0149
(0.0143)  time: 0.3275  data: 0.0098  max mem: 2653
Epoch: [5]  [400/586]  eta: 0:01:01  lr: 0.000500  loss: 0.3923 (0.4401)  loss classifier: 0.1467
(0.1556)  loss_box_reg: 0.2140 (0.2396)  loss_objectness: 0.0224 (0.0307)  loss_rpn_box_reg: 0.0114
(0.0143)  time: 0.3276  data: 0.0100  max mem: 2653
Epoch: [5]  [500/586]  eta: 0:00:28  lr: 0.000500  loss: 0.4052 (0.4398)  loss_classifier: 0.1553
(0.1558)  loss_box_reg: 0.2328 (0.2390)  loss_objectness: 0.0253 (0.0308)  loss_rpn_box_reg: 0.0128
(0.0142)  time: 0.3273  data: 0.0101  max mem: 2653
Epoch: [5]  [585/586]  eta: 0:00:00  lr: 0.000500  loss: 0.4582 (0.4398)  loss_classifier: 0.1721
(0.1555)  loss_box_reg: 0.2634 (0.2392)  loss_objectness: 0.0231 (0.0309)  loss_rpn_box_reg: 0.0152
(0.0142)  time: 0.3204  data: 0.0098  max mem: 2653
Epoch: [5] Total time: 0:03:12 (0.3280 s / it)
creating index...
index created!
Test:  [  0/147]  eta: 0:00:55  model_time: 0.1559 (0.1559)  evaluator_time: 0.0149 (0.0149)  time:
0.3794  data: 0.2058  max mem: 2653
Test:  [100/147]  eta: 0:00:09  model_time: 0.1421 (0.1433)  evaluator_time: 0.0352 (0.0431)  time:
0.2111  data: 0.0094  max mem: 2653
Test:  [146/147]  eta: 0:00:00  model_time: 0.1417 (0.1426)  evaluator_time: 0.0356 (0.0413)  time:
0.1815  data: 0.0095  max mem: 2653
Test: Total time: 0:00:29 (0.1974 s / it)
Averaged stats: model_time: 0.1417 (0.1426)  evaluator_time: 0.0356 (0.0413)
Accumulating evaluation results...
DONE (t=0.21s).
IoU metric: bbox
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.372
 Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.758
 Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.293
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.373
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.034
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = -1.000
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.178
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.457
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.514
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.515
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.167
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = -1.000

 Mean Average Precission (Validation index) --> 0.3661

Epoch: [6]  [  0/586]  eta: 0:05:42  lr: 0.000050  loss: 0.5138 (0.5138)  loss_classifier: 0.1729
(0.1729)  loss box reg: 0.2781 (0.2781)  loss objectness: 0.0404 (0.0404)  loss_rpn_box_reg: 0.0223
(0.0223)  time: 0.5852  data: 0.2405  max mem: 2653
Epoch: [6]  [100/586]  eta: 0:02:40  lr: 0.000050  loss: 0.3951 (0.4184)  loss_classifier: 0.1610
(0.1476)  loss_box_reg: 0.2184 (0.2293)  loss_objectness: 0.0290 (0.0281)  loss_rpn_box_reg: 0.0123
(0.0134)  time: 0.3272  data: 0.0098  max mem: 2653
Epoch: [6]  [200/586]  eta: 0:02:06  lr: 0.000050  loss: 0.4205 (0.4304)  loss_classifier: 0.1404
(0.1506)  loss_box_reg: 0.2370 (0.2369)  loss_objectness: 0.0203 (0.0290)  loss_rpn_box_reg: 0.0129
(0.0139)  time: 0.3279  data: 0.0097  max mem: 2653
Epoch: [6]  [300/586]  eta: 0:01:33  lr: 0.000050  loss: 0.4532 (0.4240)  loss_classifier: 0.1450
(0.1487)  loss_box_reg: 0.2590 (0.2335)  loss_objectness: 0.0219 (0.0282)  loss_rpn_box_reg: 0.0158
(0.0136)  time: 0.3285  data: 0.0103  max mem: 2653
Epoch: [6]  [400/586]  eta: 0:01:01  lr: 0.000050  loss: 0.3888 (0.4268)  loss_classifier: 0.1500
(0.1491)  loss_box_reg: 0.2151 (0.2349)  loss_objectness: 0.0241 (0.0289)  loss_rpn_box_reg: 0.0108
(0.0139)  time: 0.3275  data: 0.0096  max mem: 2653
Epoch: [6]  [500/586]  eta: 0:00:28  lr: 0.000050  loss: 0.4140 (0.4289)  loss classifier: 0.1448
(0.1500)  loss_box_reg: 0.2261 (0.2358)  loss_objectness: 0.0276 (0.0291)  loss_rpn_box_reg: 0.0147
(0.0140)  time: 0.3301  data: 0.0108  max mem: 2653
```

```
Epoch: [6]  [585/586]  eta: 0:00:00  lr: 0.000050  loss: 0.4381 (0.4287)  loss_classifier: 0.1435
(0.1499)  loss_box_reg: 0.2371 (0.2356)  loss_objectness: 0.0233 (0.0291)  loss_rpn_box_reg: 0.0152
(0.0141)  time: 0.3210  data: 0.0098  max mem: 2653
Epoch: [6] Total time: 0:03:12 (0.3283 s / it)
creating index...
index created!
Test:  [  0/147]  eta: 0:00:57  model_time: 0.1644 (0.1644)  evaluator_time: 0.0140 (0.0140)  time:
0.3889  data: 0.2076  max mem: 2653
Test:  [100/147]  eta: 0:00:09  model_time: 0.1427 (0.1433)  evaluator_time: 0.0349 (0.0426)  time:
0.2109  data: 0.0099  max mem: 2653
Test:  [146/147]  eta: 0:00:00  model_time: 0.1421 (0.1427)  evaluator_time: 0.0293 (0.0406)  time:
0.1806  data: 0.0095  max mem: 2653
Test: Total time: 0:00:28 (0.1969 s / it)
Averaged stats: model_time: 0.1421 (0.1427)  evaluator_time: 0.0293 (0.0406)
Accumulating evaluation results...
DONE (t=0.21s).
IoU metric: bbox
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.375
 Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.761
 Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.305
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.375
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.037
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = -1.000
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.178
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.459
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.514
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.514
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.158
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = -1.000

 Mean Average Precission (Validation index) --> 0.3705

Epoch: [7]  [  0/586]  eta: 0:05:18  lr: 0.000050  loss: 0.5096 (0.5096)  loss_classifier: 0.1451
(0.1451)  loss_box_reg: 0.2840 (0.2840)  loss_objectness: 0.0617 (0.0617)  loss_rpn_box_reg: 0.0188
(0.0188)  time: 0.5439  data: 0.2055  max mem: 2653
Epoch: [7]  [100/586]  eta: 0:02:40  lr: 0.000050  loss: 0.4112 (0.4260)  loss_classifier: 0.1373
(0.1478)  loss_box_reg: 0.2365 (0.2366)  loss_objectness: 0.0201 (0.0278)  loss_rpn_box_reg: 0.0111
(0.0138)  time: 0.3277  data: 0.0099  max mem: 2653
Epoch: [7]  [200/586]  eta: 0:02:07  lr: 0.000050  loss: 0.4011 (0.4289)  loss_classifier: 0.1468
(0.1501)  loss_box_reg: 0.2204 (0.2367)  loss_objectness: 0.0185 (0.0282)  loss_rpn_box_reg: 0.0108
(0.0140)  time: 0.3291  data: 0.0104  max mem: 2653
Epoch: [7]  [300/586]  eta: 0:01:34  lr: 0.000050  loss: 0.4121 (0.4283)  loss_classifier: 0.1651
(0.1504)  loss_box_reg: 0.2325 (0.2357)  loss_objectness: 0.0199 (0.0283)  loss_rpn_box_reg: 0.0117
(0.0139)  time: 0.3277  data: 0.0100  max mem: 2653
Epoch: [7]  [400/586]  eta: 0:01:01  lr: 0.000050  loss: 0.4659 (0.4337)  loss classifier: 0.1600
(0.1522)  loss_box_reg: 0.2466 (0.2388)  loss_objectness: 0.0220 (0.0286)  loss_rpn_box_reg: 0.0170
(0.0142)  time: 0.3287  data: 0.0104  max mem: 2653
Epoch: [7]  [500/586]  eta: 0:00:28  lr: 0.000050  loss: 0.4099 (0.4293)  loss_classifier: 0.1389
(0.1505)  loss_box_reg: 0.2444 (0.2362)  loss_objectness: 0.0217 (0.0286)  loss_rpn_box_reg: 0.0140
(0.0140)  time: 0.3281  data: 0.0102  max mem: 2653
Epoch: [7]  [585/586]  eta: 0:00:00  lr: 0.000050  loss: 0.4124 (0.4299)  loss_classifier: 0.1449
(0.1505)  loss_box_reg: 0.2288 (0.2365)  loss_objectness: 0.0276 (0.0289)  loss_rpn_box_reg: 0.0129
(0.0141)  time: 0.3192  data: 0.0095  max mem: 2653
Epoch: [7] Total time: 0:03:12 (0.3285 s / it)
creating index...
index created!
Test:  [  0/147]  eta: 0:00:58  model_time: 0.1642 (0.1642)  evaluator_time: 0.0140 (0.0140)  time:
0.3983  data: 0.2133  max mem: 2653
Test:  [100/147]  eta: 0:00:09  model_time: 0.1427 (0.1434)  evaluator_time: 0.0335 (0.0407)  time:
0.2105  data: 0.0099  max mem: 2653
Test:  [146/147]  eta: 0:00:00  model_time: 0.1422 (0.1427)  evaluator_time: 0.0298 (0.0393)  time:
0.1819  data: 0.0100  max mem: 2653
Test: Total time: 0:00:28 (0.1958 s / it)
Averaged stats: model_time: 0.1422 (0.1427)  evaluator_time: 0.0298 (0.0393)
Accumulating evaluation results...
DONE (t=0.20s).
IoU metric: bbox
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.374
 Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.760
 Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.304
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.374
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.035
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = -1.000
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.179
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.459
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.514
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.514
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.158
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = -1.000

 Mean Average Precission (Validation index) --> 0.3693

Epoch: [8]  [  0/586]  eta: 0:05:20  lr: 0.000050  loss: 0.4671 (0.4671)  loss classifier: 0.1304
(0.1304)  loss box reg: 0.2701 (0.2701)  loss objectness: 0.0509 (0.0509)  loss_rpn_box_reg: 0.0157
(0.0157)  time: 0.5463  data: 0.2110  max mem: 2653
Epoch: [8]  [100/586]  eta: 0:02:41  lr: 0.000050  loss: 0.4645 (0.4313)  loss_classifier: 0.1617
(0.1506)  loss_box_reg: 0.2441 (0.2367)  loss_objectness: 0.0297 (0.0301)  loss_rpn_box_reg: 0.0159
(0.0139)  time: 0.3284  data: 0.0101  max mem: 2653
```

```
Epoch: [8]  [200/586]  eta: 0:02:07  lr: 0.000050  loss: 0.4576 (0.4312)  loss_classifier: 0.1565
(0.1494)  loss_box_reg: 0.2507 (0.2376)  loss_objectness: 0.0270 (0.0299)  loss_rpn_box_reg: 0.0109
(0.0144)  time: 0.3285  data: 0.0100  max mem: 2653
Epoch: [8]  [300/586]  eta: 0:01:34  lr: 0.000050  loss: 0.4189 (0.4282)  loss_classifier: 0.1374
(0.1493)  loss_box_reg: 0.2394 (0.2361)  loss_objectness: 0.0186 (0.0287)  loss_rpn_box_reg: 0.0122
(0.0141)  time: 0.3276  data: 0.0100  max mem: 2653
Epoch: [8]  [400/586]  eta: 0:01:01  lr: 0.000050  loss: 0.3877 (0.4284)  loss_classifier: 0.1406
(0.1489)  loss_box_reg: 0.2192 (0.2368)  loss_objectness: 0.0186 (0.0285)  loss_rpn_box_reg: 0.0116
(0.0142)  time: 0.3269  data: 0.0098  max mem: 2653
Epoch: [8]  [500/586]  eta: 0:00:28  lr: 0.000050  loss: 0.4515 (0.4289)  loss_classifier: 0.1509
(0.1492)  loss_box_reg: 0.2497 (0.2363)  loss_objectness: 0.0310 (0.0292)  loss_rpn_box_reg: 0.0145
(0.0142)  time: 0.3288  data: 0.0102  max mem: 2653
Epoch: [8]  [585/586]  eta: 0:00:00  lr: 0.000050  loss: 0.4440 (0.4269)  loss_classifier: 0.1538
(0.1486)  loss_box_reg: 0.2165 (0.2352)  loss_objectness: 0.0282 (0.0290)  loss_rpn_box_reg: 0.0120
(0.0141)  time: 0.3192  data: 0.0097  max mem: 2653
Epoch: [8] Total time: 0:03:12 (0.3287 s / it)
creating index...
index created!
Test:  [  0/147]  eta: 0:00:55  model_time: 0.1541 (0.1541)  evaluator_time: 0.0143 (0.0143)  time:
0.3759  data: 0.2053  max mem: 2653
Test:  [100/147]  eta: 0:00:09  model_time: 0.1423 (0.1432)  evaluator_time: 0.0334 (0.0407)  time:
0.2095  data: 0.0098  max mem: 2653
Test:  [146/147]  eta: 0:00:00  model_time: 0.1422 (0.1426)  evaluator_time: 0.0285 (0.0392)  time:
0.1805  data: 0.0094  max mem: 2653
Test: Total time: 0:00:28 (0.1955 s / it)
Averaged stats: model_time: 0.1422 (0.1426)  evaluator_time: 0.0285 (0.0392)
Accumulating evaluation results...
DONE (t=0.20s).
IoU metric: bbox
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.376
 Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.762
 Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.310
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.376
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.035
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = -1.000
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.180
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.460
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.516
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.516
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.158
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = -1.000

 Mean Average Precission (Validation index) --> 0.3721

Epoch: [9]  [  0/586]  eta: 0:05:51  lr: 0.000005  loss: 0.4140 (0.4140)  loss classifier: 0.1347
(0.1347)  loss_box_reg: 0.2308 (0.2308)  loss_objectness: 0.0332 (0.0332)  loss_rpn_box_reg: 0.0153
(0.0153)  time: 0.5991  data: 0.2697  max mem: 2653
Epoch: [9]  [100/586]  eta: 0:02:41  lr: 0.000005  loss: 0.4172 (0.4185)  loss_classifier: 0.1330
(0.1458)  loss_box_reg: 0.2489 (0.2332)  loss_objectness: 0.0227 (0.0257)  loss_rpn_box_reg: 0.0136
(0.0137)  time: 0.3286  data: 0.0101  max mem: 2653
Epoch: [9]  [200/586]  eta: 0:02:07  lr: 0.000005  loss: 0.4136 (0.4266)  loss_classifier: 0.1465
(0.1480)  loss_box_reg: 0.2258 (0.2376)  loss_objectness: 0.0178 (0.0271)  loss_rpn_box_reg: 0.0120
(0.0140)  time: 0.3274  data: 0.0098  max mem: 2653
Epoch: [9]  [300/586]  eta: 0:01:34  lr: 0.000005  loss: 0.4623 (0.4310)  loss_classifier: 0.1603
(0.1497)  loss_box_reg: 0.2511 (0.2383)  loss_objectness: 0.0280 (0.0289)  loss_rpn_box_reg: 0.0156
(0.0140)  time: 0.3292  data: 0.0100  max mem: 2653
Epoch: [9]  [400/586]  eta: 0:01:01  lr: 0.000005  loss: 0.4305 (0.4274)  loss classifier: 0.1478
(0.1485)  loss box reg: 0.2307 (0.2363)  loss objectness: 0.0215 (0.0288)  loss_rpn_box_reg: 0.0126
(0.0138)  time: 0.3293  data: 0.0103  max mem: 2653
Epoch: [9]  [500/586]  eta: 0:00:28  lr: 0.000005  loss: 0.3964 (0.4236)  loss_classifier: 0.1348
(0.1470)  loss_box_reg: 0.2141 (0.2346)  loss_objectness: 0.0202 (0.0282)  loss_rpn_box_reg: 0.0118
(0.0138)  time: 0.3278  data: 0.0098  max mem: 2653
Epoch: [9]  [585/586]  eta: 0:00:00  lr: 0.000005  loss: 0.4530 (0.4281)  loss_classifier: 0.1482
(0.1488)  loss_box_reg: 0.2305 (0.2361)  loss_objectness: 0.0292 (0.0291)  loss_rpn_box_reg: 0.0128
(0.0140)  time: 0.3212  data: 0.0100  max mem: 2653
Epoch: [9] Total time: 0:03:13 (0.3295 s / it)
creating index...
index created!
Test:  [  0/147]  eta: 0:01:01  model_time: 0.1763 (0.1763)  evaluator_time: 0.0172 (0.0172)  time:
0.4177  data: 0.2219  max mem: 2653
Test:  [100/147]  eta: 0:00:09  model_time: 0.1427 (0.1438)  evaluator_time: 0.0350 (0.0413)  time:
0.2102  data: 0.0100  max mem: 2653
Test:  [146/147]  eta: 0:00:00  model_time: 0.1426 (0.1431)  evaluator_time: 0.0293 (0.0398)  time:
0.1810  data: 0.0093  max mem: 2653
Test: Total time: 0:00:28 (0.1966 s / it)
Averaged stats: model_time: 0.1426 (0.1431)  evaluator_time: 0.0293 (0.0398)
Accumulating evaluation results...
DONE (t=0.21s).
IoU metric: bbox
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.376
 Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.762
 Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.307
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.376
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.034
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = -1.000
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.180
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.462
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.516
```

```
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.517
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.158
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = -1.000

 Mean Average Precission (Validation index) --> 0.3708
```

****Predicción****

```
num = 4
# pick one image from the test set
img, target = dataset_test[num]
# put the model in evaluation mode
model.eval()
with torch.no_grad():
    prediction = model([img.to(device)])

dataset_test[4]
```

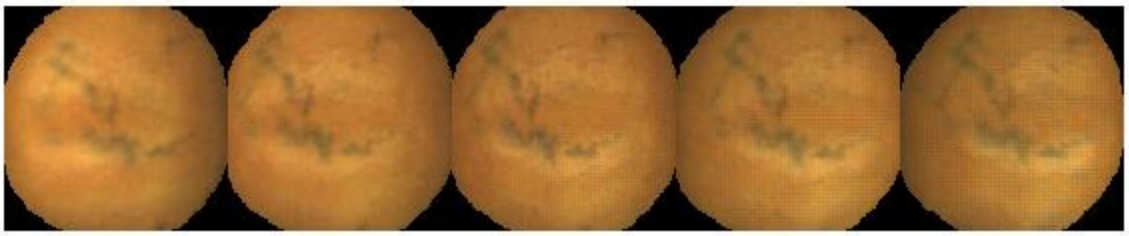
```
(tensor([[[-1., -1., -1.,  ..., -1., -1., -1.],
          [-1., -1., -1.,  ..., -1., -1., -1.],
          [-1., -1., -1.,  ..., -1., -1., -1.],
          ...,
          [-1., -1., -1.,  ..., -1., -1., -1.],
          [-1., -1., -1.,  ..., -1., -1., -1.],
          [-1., -1., -1.,  ..., -1., -1., -1.]],

         [[-1., -1., -1.,  ..., -1., -1., -1.],
          [-1., -1., -1.,  ..., -1., -1., -1.],
          [-1., -1., -1.,  ..., -1., -1., -1.],
          ...,
          [-1., -1., -1.,  ..., -1., -1., -1.],
          [-1., -1., -1.,  ..., -1., -1., -1.],
          [-1., -1., -1.,  ..., -1., -1., -1.]],

         [[-1., -1., -1.,  ..., -1., -1., -1.],
          [-1., -1., -1.,  ..., -1., -1., -1.],
          [-1., -1., -1.,  ..., -1., -1., -1.],
          ...,
          [-1., -1., -1.,  ..., -1., -1., -1.],
          [-1., -1., -1.,  ..., -1., -1., -1.],
          [-1., -1., -1.,  ..., -1., -1., -1.]]]),
 {'area': tensor([324., 324., 324., 324., 324., 324., 324., 324., 324., 324., 324., 324.,
          324., 324., 324., 324., 324., 324., 324., 324., 324., 324., 324., 324.,
          324., 324., 324., 324., 324., 324., 324., 324., 324., 324., 324., 324.,
          324., 324., 324.]),
  'boxes': tensor([[ 19.,  16.,  37.,  34.],
          [ 34.,  28.,  52.,  46.],
          [ 44.,  40.,  62.,  58.],
          [ 16.,  60.,  34.,  78.],
          [ 37.,  57.,  55.,  75.],
          [ 53.,  61.,  71.,  79.],
          [ 70.,  63.,  88.,  81.],
          [ 79.,   9.,  97.,  27.],
          [124.,  13., 142.,  31.],
          [139.,  32., 157.,  50.],
          [114.,  51., 132.,  69.],
          [137.,  53., 155.,  71.],
          [149.,  61., 167.,  79.],
          [168.,  63., 186.,  81.],
          [177.,  10., 195.,  28.],
          [238.,  15., 256.,  33.],
          [249.,  35., 267.,  53.],
          [291.,   9., 309.,  27.],
          [245.,  54., 263.,  72.],
          [261.,  59., 279.,  77.],
          [277.,  63., 295.,  81.],
          [224.,  36., 242.,  54.],
          [357.,  12., 375.,  30.],
          [348.,  30., 366.,  48.],
          [374.,  34., 392.,  52.],
          [356.,  50., 374.,  68.],
          [373.,  55., 391.,  73.],
          [389.,  60., 407.,  78.],
          [405.,  63., 423.,  81.],
          [412.,   7., 430.,  25.],
          [468.,  14., 486.,  32.],
          [481.,  28., 499.,  46.],
          [490.,  39., 508.,  57.],
          [476.,  56., 494.,  74.],
          [492.,  56., 510.,  74.],
          [502.,  62., 520.,  80.],
          [520.,  64., 538.,  82.],
          [526.,  12., 544.,  30.],
          [462.,  34., 480.,  52.]]),
  'image_id': tensor([1762]),
  'iscrowd': tensor([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

```
          0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]),
  'labels': tensor([3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,
          3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3])}]
```

Prediction

```
[{'boxes': tensor([[359.0295,  12.9659, 377.0854,  30.9638],
          [473.3358,  13.9850, 491.3409,  32.1528],
          [404.0995,  63.3312, 422.0889,  81.3033],
          [167.8448,  61.2096, 185.7786,  79.2070],
          [149.6853,  59.8830, 167.8336,  78.1542],
          [123.8888,  12.4795, 141.9063,  30.7930],
          [ 21.7929,  15.5715,  39.7823,  33.6043],
          [236.8291,  11.6322, 254.9471,  29.7655],
          [ 52.7585,  61.2780,  70.9030,  79.3984],
          [280.0240,  61.9854, 297.7437,  79.9741],
          [261.6604,  59.9140, 279.8497,  78.0256],
          [413.2296,  10.9316, 431.2872,  28.7428],
          [291.9601,   9.3027, 309.9761,  27.2356],
          [386.7506,  61.6236, 405.0047,  79.7320],
          [178.6861,   8.4939, 196.6437,  26.3798],
          [249.1223,  31.2316, 267.5792,  49.4117],
          [ 69.0008,  62.1692,  86.9788,  80.1553],
          [498.5014,  59.5032, 516.6342,  77.5333],
          [371.3634,  31.0820, 389.5589,  49.2890],
          [516.9154,  62.3985, 534.6765,  80.2311],
          [134.9426,  29.8759, 153.1650,  47.9836],
          [ 37.3259,  35.0328,  55.4434,  53.1946],
          [485.0424,  53.2073, 503.1219,  71.3004],
          [247.7785,  52.6278, 265.8596,  70.6869],
          [488.1779,  34.7274, 506.2968,  52.9055],
          [ 37.7840,  55.3800,  55.7930,  73.3613],
          [135.6484,  52.7431, 153.7080,  70.7781],
          [138.2182,  34.5590, 156.4386,  52.8733],
          [375.1924,  35.4556, 393.2844,  53.6781],
          [525.6511,   9.8494, 543.6890,  27.6580],
          [484.9922,  30.1883, 503.2774,  48.3771],
          [350.5710,  51.7001, 368.6715,  69.6609],
          [ 41.0026,  39.2633,  59.1682,  57.3970],
          [252.1258,  35.8860, 270.4236,  54.0200],
          [370.8911,  52.8487, 389.0245,  70.9113],
          [382.1434,  58.5589, 400.1505,  76.5717],
          [494.3407,  56.9087, 512.4250,  74.9654],
          [ 16.4033,  53.9075,  34.3673,  71.8006],
          [257.9472,  57.1512, 275.9093,  75.1427],
          [ 20.4539,  22.3774,  38.4023,  40.4920],
          [ 74.6545,   9.8062,  92.7403,  27.6142],
          [145.6290,  56.9068, 163.5419,  74.9082],
          [ 42.5949,  57.9206,  60.5267,  75.9347],
          [140.1204,  39.9507, 158.1332,  58.0002],
          [492.4326,  60.8167, 531.8372,  78.6688],
          [ 46.6126,  61.5298,  86.5888,  79.5660],
          [136.0964,  28.3501, 155.6970,  61.9151],
          [257.9168,  61.1535, 294.8973,  79.0756],
          [ 14.3588,  42.1492,  34.3307,  75.3537]], device='cuda:0'),
  'labels': tensor([3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,
          3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,
          3], device='cuda:0'),
  'scores': tensor([0.9573, 0.9556, 0.9550, 0.9526, 0.9517, 0.9517, 0.9451, 0.9441, 0.9439,
          0.9398, 0.9362, 0.9331, 0.9303, 0.9302, 0.9242, 0.9237, 0.9219, 0.9181,
          0.9177, 0.9175, 0.9142, 0.9094, 0.9062, 0.8914, 0.8903, 0.8795, 0.8777,
          0.8748, 0.8727, 0.8681, 0.8655, 0.8607, 0.8560, 0.8505, 0.8459, 0.8439,
          0.8318, 0.8287, 0.8265, 0.8259, 0.8037, 0.7796, 0.7493, 0.6856, 0.2212,
          0.1577, 0.0958, 0.0892, 0.0559], device='cuda:0')}]
```

```
mean1=0.5;
std1=0.5;
img=img*std1+mean1;
img=img.mul(255).permute(1, 2, 0).byte().numpy()
Image.fromarray(img)
```



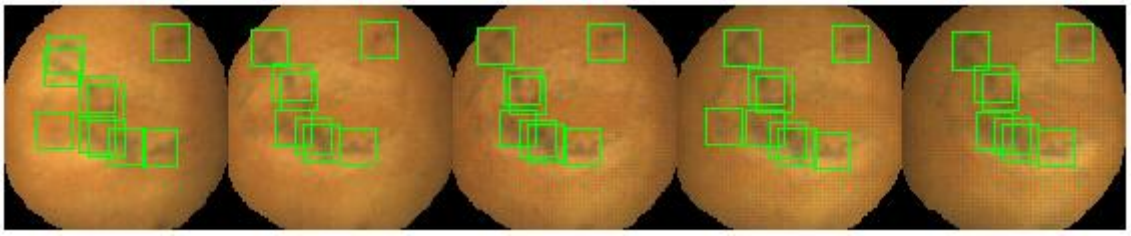
```
img_p = img[:, :, ::-1].copy()
img_r = img[:, :, ::-1].copy()
```

Filtrar y seleccionar los scores más altos

```
img_pf1 = img[:, :, ::-1].copy()
prediction_f  = []
boxes_f = []
labels_f = []
scores_f = []

dict = {}
dict["boxes"] = boxes_f
dict["labels"] = labels_f
dict["scores"] = scores_f

prediction_f.append(dict)

for i in range(len(prediction[0]['boxes'])):
  if prediction[0]['scores'][i]>0.7:
    boxes_f.append(prediction[0]['boxes'][i])
    labels_f.append(prediction[0]['labels'][i])
    scores_f.append(prediction[0]['scores'][i])
```

Boxes de la predicción filtrados

```
import cv2
from google.colab.patches import cv2_imshow

for i in range(len(prediction_f[0]['boxes'])):

  x1=int(prediction_f[0]['boxes'][i][0])
  y1=int(prediction_f[0]['boxes'][i][1])
  x2=int(prediction_f[0]['boxes'][i][2])
  y2=int(prediction_f[0]['boxes'][i][3])

  if int(prediction_f[0]['labels'][i])==1:
    colors=(0,0,255)
  if int(prediction_f[0]['labels'][i])==2:
    colors=(255,0,0)
  if int(prediction_f[0]['labels'][i])==3:
    colors=(0,255,0)

  img_pf1 = cv2.rectangle(img_pf1,(x1,y1),(x2,y2),colors,1)

cv2_imshow(img_pf1)
```



Boxes reales

```
import cv2
from google.colab.patches import cv2_imshow

for i in range(len(dataset_test[num][1]['boxes'])):

  x1=int(dataset_test[num][1]['boxes'][i][0])
  y1=int(dataset_test[num][1]['boxes'][i][1])
  x2=int(dataset_test[num][1]['boxes'][i][2])
  y2=int(dataset_test[num][1]['boxes'][i][3])

  if int(dataset_test[num][1]['labels'][i])==1:
    color=(0,0,255)
  if int(dataset_test[num][1]['labels'][i])==2:
    color=(255,0,0)
  if int(dataset_test[num][1]['labels'][i])==3:
    color=(0,255,0)
  img_r = cv2.rectangle(img_r,(x1,y1),(x2,y2),color,1)

cv2_imshow(img_r)
```