Barreto, Agustina Ayelén

Deep learning aplicado al procesamiento de imágenes para la detecciones de objetos

2022

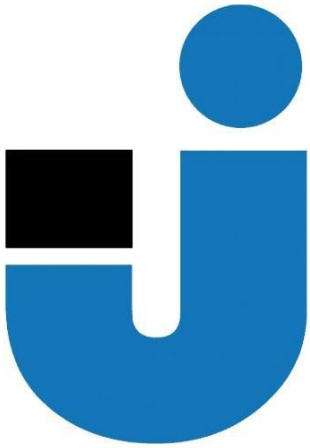
Instituto: Ingeniería y Agronomía

Carrera: Ingeniería en Informática

Universidad Nacional Arturo Jauretche

Instituto de Ingeniería y Agronomía

Ingeniería en Informática



Práctica profesional Supervisada

Informe Final

Deep Learning aplicado al procesamiento de imágenes para la detección de objetos

Estudiante:

Agustina Ayelén Barreto

Fecha: 15 de junio de 2022

ESTUDIANTE

Nombre y Apellido: Barreto, Agustina Ayelén

Correo electrónico: aayelenb@gmail.com

ORGANIZACIÓN DONDE SE REALIZA LA PRÁCTICA PROFESIONAL SUPERVISADA

Nombre de la institución: Universidad Nacional Arturo Jauretche

Dirección: Av. Calchaquí 6200, Florencio Varela, (1888) Buenos Aires, Argentina

Teléfono: +54 11 4275 6100

Sector: Programa Tecnologías de la Información y la Comunicación (TIC) en aplicaciones de interés social, Instituto de Ingeniería y Agronomía

TUTOR ORGANIZACIONAL

Nombre y Apellido: Ing. Salina, Mauro David

Correo electrónico: mdsalina@unaj.edu.ar

DOCENTE SUPERVISOR

Nombre y Apellido: Mg. Ing. OSIO, Jorge

Correo electrónico: josio@unaj.edu.ar

DOCENTE TUTOR DEL TALLER DE APOYO A LA PRODUCCIÓN DE TEXTOS ACADÉMICOS

Nombre y Apellido: Prof. Lavigna, Lía

Correo electrónico: llavigna@unaj.edu.ar

COORDINADOR DE LA CARRERA DE INGENIERÍA INFORMÁTICA

Nombre y Apellido: Dr. Ing. Morales, Martín

Correo electrónico: martin.morales@unaj.edu.ar

1. Resumen

La práctica fue realizada con motivo de entrar en el mundo de la inteligencia artificial, dado que es un tema que presenta un marcado interés social y tecnológico en la actualidad. Además, la idea de detectar objetos defectuosos y no defectuosos surgió de haber cursado una materia donde se estudia el tema de la calidad que se maneja durante los procesos de producción.

El propósito de la práctica fue la adquisición de los conocimientos necesarios para entender en qué consiste la creación y utilización de las redes neuronales para futuros proyectos personales o laborales.

Durante el proyecto, se fue tomando un enfoque cuantitativo siguiendo los métodos y técnicas adquiridos como analista programador adicionando los necesarios mediante investigación para poder concluir de forma óptima el trabajo.

Los resultados obtenidos fueron adecuados debido a que, en los casos de detecciones correctas, los porcentajes de acierto generalmente superaban el 80%. Además, al realizar las evaluaciones de rendimiento de los modelos creados se obtuvo una precisión promedio o mAP por encima del 65%.

Se cumplieron los objetivos inicialmente planteados y se investigaron diferentes técnicas para la detección de objetos, obteniendo buenos rendimientos y realizando pruebas reales en un sistema embebido.

Palabras clave. Inteligencia artificial - Redes neuronales - Sistema embebido - mAP.

1.1 Abstract

The project was done with the purpose of entering into artificial intelligence's world, because it is a very interesting topic. Also, the idea of detecting defective and non-defective objects appear from having studied a subject where quality is studied during production processes. The project's purpose was the acquisition of the necessary knowledge to understand the creation and use of neural networks for future personal or work projects. During the project, a quantitative approach was taken, following the methods and techniques acquired as a programmer analyst, adding the necessary ones through research to conclude the work optimally. The results obtained were good because, in correct detections's cases, the success percentages were generally around 80%. Also, when doing performance evaluations of created models, an average accuracy or mAP used to be above 65%.

The initially set objectives were achieved and different techniques for object detection were investigated, obtaining good performance and carrying out real tests in an embedded system.

Keywords. Artificial intelligence - neural networks - embedded system - mAP.

2. Dedicatorias y agradecimientos

Quiero agradecer a la Universidad Nacional Arturo Jauretche y a todos los docentes con los que tuve la dicha de cursar durante la carrera. Cada docente dejó en mí además de los conocimientos brindados, la motivación y las ganas de superarse y de no rendirse con facilidad. Además, a los compañeros de cursada con los que realice trabajos integradores o con los que solía estudiar para los parciales, siempre se agradece la buena predisposición y el compañerismo.

En especial, también, agradecer a mis tutores Mg. Ing. Jorge Osio e Ing. Mauro Salina que sin la buena predisposición de ellos no hubiera sido posible el desarrollo de la práctica profesional supervisada. Además, agradecer a Coordinador de la carrera Ingeniería en Informática Dr. Ing. Martín Morales que durante la carrera siempre estuvo presente e hizo sentir su calidez humana y profesionalismo.

Dedico el final del camino realizado, que fue largo, a mi familia y amigos los cuales me apoyaron y lo celebro también con ellos.

Índice

Índice de figuras

3. Introducción

El presente trabajo forma parte de la Práctica Profesional Supervisada (PPS) dentro de la carrera de Ingeniería en Informática en la Universidad Nacional Arturo Jauretche (UNAJ). El mismo tiene por objetivo general enfocarse en el desarrollo de una aplicación de software encargada de realizar una búsqueda de objetos en imágenes a partir de una clasificación de las mismas en tiempo real, mediante la cual se buscará detectar la presencia de determinados objetos en una imagen. El desarrollo se enfocará específicamente en el uso de redes neuronales convolucionales, las cuales han demostrado ser las más eficientes en el área del procesamiento de imágenes. Específicamente, se espera que el software permita realizar la tarea de detección de objetos dañados y en buen estado en imágenes o video.

El tema seleccionado presenta un marcado interés tecnológico y social. Por lo tanto, la utilización del software combinando ambos intereses permite, entre otras posibilidades, por ejemplo, ser un recurso importante en el área de la industria para que los ingenieros puedan supervisar los procesos y tomar decisiones acertadas aprovechando la visión artificial para dar sentido al entorno. Otro ejemplo en el que el software puede ser de utilidad es en los comercios de productos ya que se puede utilizar la visión artificial para mejorar la experiencia de compra, aumentar la prevención de posibles pérdidas y detectar las estanterías agotadas, entre otras cosas.

3.1 Estado actual del tema de estudio

Es preciso decir que la aparición del concepto de “Industria 4.0” produjo un cambio de paradigma para las industrias. Mediante la aplicación de este concepto en cualquier empresa del sector industrial, lo primero que se observa es la autosuficiencia en las cadenas de fabricación. Esto da lugar a una mayor eficacia y eficiencia, lo cual genera una respuesta más rápida a las demandas exigidas por el mercado. Otro hecho importante de este concepto, es la posibilidad de disponer de toda la información que sea requerida en tiempo real, independientemente del proceso que sea o de su localización en las instalaciones. En este sentido, la inteligencia artificial es señalada como elemento central de esta transformación digital, íntimamente relacionada con la acumulación creciente de grandes cantidades de datos (“big data”), el uso de algoritmos para procesarlos, y la

interconexión masiva de sistemas y dispositivos digitales. Algunas de las ventajas de su aplicación en la industria son:

✓ Automatización de procesos: permite que las máquinas hagan de forma automática tareas que para los humanos resultan repetitivas y tediosas.

✓ Reduce el error humano: al reducir la intervención de los humanos en ciertos procesos, acaba con las posibilidades de que estos puedan cometer errores. Por ejemplo, una errata al introducir un dato en la contabilidad de un negocio.

✓ Potencia la creatividad: al liberar a los trabajadores de tareas repetitivas y poco motivadoras, la mente de estos es mucho más libre para dedicarse al proceso creativo.

✓ Aporta precisión: al ser capaz de tomar decisiones por sí misma, la inteligencia artificial da lugar a procesos productivos mucho más eficientes y con una menor tasa de error.

✓ Agiliza la toma de decisiones: la inteligencia artificial es capaz de analizar miles de datos en apenas minutos y además tener en cuenta posibles actualizaciones de los mismos. La información bien sintetizada y actualizada ayuda a los profesionales a tomar decisiones estratégicas.

Actualmente existe un campo de la Inteligencia Artificial que viene prosperando: la Visión Artificial. Ésta tiene como objetivo programar un software para que "entienda" una escena o las características de una imagen. Esta información es empleada para tomar decisiones o controlar un proceso.

Los sistemas de Visión Artificial en los procesos tecnológicos y dentro de estos los procesos de producción pueden realizar tareas de manera más efectivas y adecuadas que la visión humana, tal es el caso de los siguientes aspectos:

✓ Dentro del espectro electromagnético la visión humana solamente capta un pequeño rango de frecuencias y amplitudes: “rango de luz visible”, los sistemas de visión artificial pueden captar todo el espectro, es decir, además del rango de luz visible puede captar ondas de radio, de televisión, microondas, infrarrojos, ultravioletas, rayos X, rayos gamma y rayos cósmicos.

✓ La velocidad de respuesta de la visión humana es de 0,06 segundos, mientras que en las cámaras de estado sólido es de 0,00001 segundos y este tiempo se va reduciendo según se mejora la electrónica de estos sistemas.

✓ A diferencia de los sistemas artificiales, la visión humana se cansa, se ve afectada por las emociones y es poco consistente por la fatiga y las distracciones, en cambio la visión artificial mantiene su nivel de rendimiento constante a lo largo de su vida útil. Es ideal en trabajos repetitivos y monótonos.

✓ El ser humano puede discernir entre 10 o 20 niveles de gris, los sistemas de visión artificial tienen una definición muy superior.

✓ La visión humana tiene muy poca precisión y para obtener información cuantitativa necesita apoyarse en instrumentos de medida, los sistemas de visión artificial tienen gran precisión en la medición, dependiendo solamente de la resolución espacial de los componentes del sistema.

✓ Los sistemas de visión artificial pueden trabajar en entornos muy peligrosos, con riesgos radioactivos, químicos, biológicos, ruido, polución, temperaturas muy altas y muy bajas.

El uso de la visión artificial crece rápidamente gracias al descubrimiento de las ventajas que tiene para las industrias, entre ellas:

✓ Procesa de una manera más simple y rápida: permite a los clientes y a las industrias chequear los productos. Además, les da acceso a sus productos.

✓ Fiabilidad: las computadoras y las cámaras no tienen el factor humano del cansancio. La eficiencia suele ser la misma, no depende de factores externos como pueden ser bajas por enfermedad o errores humanos por agotamiento.

✓ Precisión: esta tecnología asegura una mejor precisión en el producto final.

✓ Una amplia gama de usos: se puede ver el mismo sistema informático en varios campos y actividades diferentes (fábricas con seguimiento de almacenes y envío de suministros, y en la industria médica a través de imágenes escaneadas, entre otras múltiples opciones).

✓ La reducción de los costes: el tiempo y la tasa de error se reducen en el proceso.

Lo cierto es que una empresa que demora en evolucionar para encajar en las necesidades de su mercado, tiene muchas probabilidades de desaparecer. Esto no quiere decir que un negocio netamente físico no pueda ser rentable, pero es innegable la gran relevancia que ha tenido la transformación digital en la manera en cómo hacemos las transacciones en la actualidad.

3.2 Motivación: problemas que se plantean

Los productos defectuosos pueden suponer graves problemas en el proceso de producción. El hecho de tener productos con algún defecto tiene como consecuencia inmediata la pérdida de un producto destinado a la comercialización. Es decir, no solo se pierde el tiempo invertido en la elaboración del producto y el valor de las materias primas utilizadas en la misma, sino que también quiere decir

que se ha perdido una venta, ya que el tiempo que hemos perdido con ese producto podríamos haberlo destinado a fabricar otro que quizás se hubiese vendido. El producto con deficiencias puede ser que llegue al cliente final y que este lo comience a usar con normalidad sin conocer sus deficiencias, que un plato sea muy frágil debido a que la elaboración del mismo ha sido errónea, puede que no pueda suponer un peligro inmediato para la persona, pero que, por ejemplo, un automóvil tenga mal montado el sistema del motor y que esto pueda hacerlo incendiar sí supone un riesgo para la persona. Este tipo de defectos, en primera instancia pueden suponer un peligro para la vida de la persona, pero siendo un poco superficiales y fijándonos en las consecuencias puramente empresariales encontraríamos otras causas de pérdida económica. En primer lugar, ese producto puede ser que esté todavía en el periodo de garantía y por lo tanto la empresa se tendrá que hacer cargo de su reparación y todos los trámites que conllevan, trato con el consumidor, envíos, reparación/sustitución. Para cualquier empresa esto son unos gastos bastante elevados ya que tiene que mantener un servicio postventa, de reparación y unos grandes gastos en envíos. Pero es que no solo es eso, y es que una empresa que presente grandes problemas en el funcionamiento de sus productos, corre el riesgo de que su producto pierda valor rápidamente debido a que se considere un mal producto, por lo tanto, baja el valor del producto y en muchas ocasiones la calidad de la marca.

Gracias a la mejora de los algoritmos implementados en estos últimos años, como las redes neuronales convolucionales, se pueden abordar estos inconvenientes mediante técnicas de inteligencia artificial. Las técnicas de inteligencia artificial pueden ser utilizadas para: modelar, identificar, optimizar, predecir, pronosticar y controlar el comportamiento dinámico de diferentes sistemas reales.

4. Objetivos

4.1 Objetivos generales

El objetivo principal de la PPS es desarrollar una aplicación a través de la que se realizará procesamiento de imágenes utilizando Machine Learning, aplicado a la detección de objetos dañados y en buen estado en imágenes. Para concretar este objetivo se investigó el uso de Inteligencia Artificial en ciencias de la computación.

Otro objetivo es la adquisición de conocimiento y competencias en un área en constante crecimiento y con características multidisciplinarias, donde se relacionan los sistemas digitales, la mejora de la calidad en una industria o empresa (mejora continua de los procesos[1]) y la inteligencia artificial.

4.2 Objetivos específicos

La conjetura de trabajo es que a mayor investigación y estudio en el campo de la inteligencia artificial se podrá desarrollar e implementar nuevas propuestas de sistemas basados en aprendizaje automático para la mejora en los procesos de producción, más avanzados en cuanto a diseño y tecnología, que permitirán obtener importantes beneficios respecto a la prevención y detección de productos defectuosos.

Para alcanzar el objetivo principal mencionado anteriormente, con el fin de optimizar la calidad de los productos, primero se debe tener en cuenta los objetos a detectar en una imagen, que en un principio serán latas y botellas y dadas sus características poder detectar si dichos objetos se encuentran o no en buen estado. Con un sistema diseñado a medida, se buscará lograr una aplicación de software encargada de realizar detecciones sobre imágenes en tiempo real, mediante la cual se detecte la presencia de objetos dañados y no dañados contenidos en la imagen. En este punto se puede concluir cuál es la base para el estudio que se propone, en donde se tiene la necesidad de identificar objetos en imágenes mediante un algoritmo de aprendizaje eficiente, que permita conseguir el comportamiento deseado y realizar la correcta detección. A continuación, se describen las tareas a desarrollar:

✓ Analizar los distintos tipos de redes neuronales e investigación de los diferentes usos de las mismas. Para eso se realiza la investigación, el análisis y estudio de los conceptos de Inteligencia Artificial, Machine Learning y Deep Learning.

✓ Analizar metodologías utilizadas para la detección de objetos mediante imágenes utilizando Machine Learning. Por lo que se realiza la recolección de información sobre las nuevas tecnologías de sistemas de procesamiento de imágenes y las técnicas actuales usadas en sistemas de detección de objetos.

[1] Un proceso de mejora continua es la actividad de analizar los procesos que se usan dentro de una organización o administración, revisarlos y realizar adecuaciones para minimizar los errores de forma permanente.

✓ Crear un dataset[2] de imágenes, que serán provistas a la red para su entrenamiento y posterior detección de objetos en dichas imágenes.

✓ Desarrollar un modelo de red neuronal capaz de analizar y detectar objetos en imágenes. La red neuronal a utilizar es la de tipo convolucional, dado que ha demostrado ser la más eficiente en el área del procesamiento de imágenes.

✓ Comparar el modelo con la detección mediante el algoritmo de Haar Cascade.

✓ Testear la aplicación desarrollada y analizar los resultados.

✓ Probar la aplicación a través de un sistema embebido en un ambiente real.

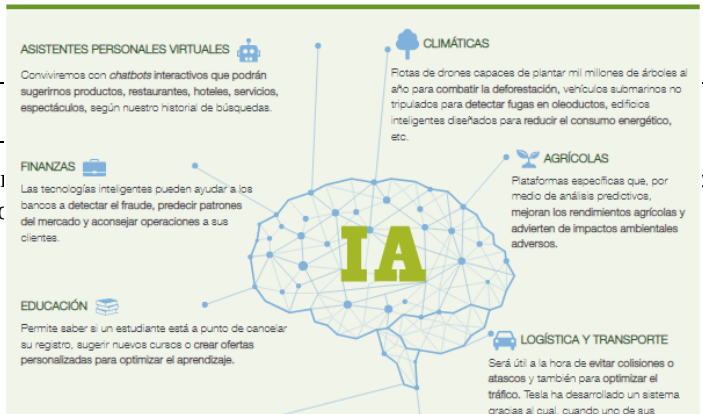
5. Marco Teórico

5.1 Inteligencia Artificial

La Inteligencia Artificial (IA) es la combinación de algoritmos planteados con el propósito de crear máquinas que presenten las mismas capacidades que el ser humano.

Hace tiempo que la inteligencia artificial abandonó el espectro de la ciencia ficción para colarse en la vida cotidiana. Sus aplicaciones en múltiples sectores, como salud, finanzas, transporte o educación, entre otros, han provocado, por ejemplo, que la Unión Europea desarrolle sus propias Leyes de la Robótica.

A continuación, en la figura 1 se muestran las principales aplicaciones de la Inteligencia Artificial.



[2] El dataset es una repr odelo de programación relacional coherente co

Figura 1. Principales aplicaciones de la IA.
Fuente: https://cog.cl/wp-content/uploads/2019/02/asasa-1.jpg

El término Inteligencia Artificial se usa a menudo indistintamente con sus subcampos, que incluyen el Machine Learning y el aprendizaje profundo. Sin embargo, es importante tener en cuenta que, aunque todo Machine Learning es IA, no toda la IA es Machine Learning. En la figura 2 se muestra como se incluyen o no los subcampos de la IA.

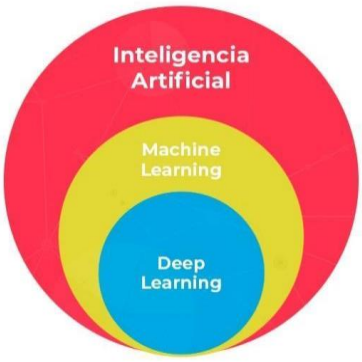


Figura 2. IA y sus subcampos.
Fuente: https://www.masterdatascienceucm.com/wp-content/uploads/2020/12/Conceptos-IA-Machine-Learning-y-Deep-Learning-980x490.png.webp

5.2 Machine Learning

El Aprendizaje Automático o Machine Learning (ML) es el subconjunto de inteligencia artificial (IA) que se centra en desarrollar sistemas que aprenden, o mejoran el rendimiento, en función de los datos que consumen. Hoy en día, el aprendizaje autónomo funciona en todo nuestro alrededor. Esto ocurre cuando se

interactúa con bancos, se compra en línea o se utilizan redes sociales, los algoritmos de aprendizaje autónomo entran en juego para hacer que la experiencia humana sea eficiente, fluida y segura.

El Aprendizaje Automático funciona conceptualmente diferente a la programación regular. En la programación regular normalmente se programan algoritmos para convertir entradas en resultados mediante la escritura de reglas y lógica necesarias para que esos resultados ocurran. Para el Aprendizaje Automático se cuenta con una lista de entradas y resultados, pero no necesariamente se sabe cómo convertir esas entradas en los resultados, es decir, no se conoce el algoritmo que hace la conversión. Lo que se hace es crear un modelo que tome esas entradas, los resultados esperados de cada entrada y que pueda aprender por sí solo el algoritmo necesario para hacer la conversión.

Los algoritmos son los motores que impulsan el aprendizaje autónomo. En general, actualmente se utilizan tres tipos principales de algoritmos de aprendizaje autónomo: aprendizaje supervisado, no supervisado y reforzado. La diferencia entre ellos se define por cómo cada uno aprende acerca de los datos para hacer predicciones. A continuación, se brindará una descripción de cada uno.

5.2.1. Aprendizaje autónomo supervisado

Los algoritmos supervisados de aprendizaje autónomo son los más utilizados. Con este modelo, un científico de datos actúa como guía y enseña al algoritmo las conclusiones que debe hacer. Al igual que un niño que aprende a identificar las frutas al memorizarlas con un libro de imágenes, en el aprendizaje supervisado, el algoritmo se capacita mediante un conjunto de datos que ya está etiquetado y tiene un resultado predefinido.

Los ejemplos de aprendizaje autónomo supervisado incluyen algoritmos tales como: regresión lineal y logística, clasificación multiclase y máquinas de vectores de soporte.

5.2.2 Aprendizaje autónomo no supervisado

El aprendizaje autónomo no supervisado utiliza un enfoque más independiente, en el que una computadora aprende a identificar procesos y patrones complejos sin que un ser humano proporcione una guía cercana y constante. El aprendizaje autónomo no supervisado implica la capacitación basada en datos, que no tiene etiquetas o un resultado específico definido.

Para continuar con la analogía de la enseñanza infantil, el aprendizaje autónomo no supervisado es similar a un niño que aprende a identificar frutas mediante la observación de colores y patrones, en lugar de memorizar los nombres con la ayuda de un maestro. El niño buscaría similitudes entre las imágenes y las separaría en grupos, asignando a cada grupo su propia etiqueta nueva. Los ejemplos de algoritmos de aprendizaje autónomo no supervisados incluyen el agrupamiento de k-means[3], el análisis de componentes principales e independientes y las leyes de asociación.

5.2.3 Aprendizaje autónomo reforzado

Este aprendizaje se aleja más de los dos anteriores, ya que forma parte de lo que se conoce como aprendizaje profundo (Deep Learning). Su objetivo principal es construir modelos que optimicen el rendimiento en base a resultados ya obtenidos anteriormente. Para ello, su sistema de aprendizaje está basado en premios. Si la máquina lo hace bien, recibe un premio (valor positivo) si lo hace mal una "penalización" (valor negativo).

Gracias a este modelo, la máquina es capaz de entrenarse hasta dar con una buena solución y, además, aprende de forma inteligente cuando empieza a tomar las decisiones "acertadas".

El aprendizaje reforzado es una de las técnicas de Machine Learning más famosas del sector tecnológico.

5.3 Deep Learning

Deep Learning o aprendizaje profundo se usa principalmente para crear sistemas modernos y eficientes. Con Deep Learning, es posible desarrollar programas que puedan realizar comportamientos similares a los de los humanos. El aprendizaje

[3] K- means es un método de agrupamiento, que tiene como objetivo la partición de un conjunto de n observaciones en k grupos en el que cada observación pertenece al grupo cuyo valor medio es más cercano. Es un método utilizado en minería de datos.

profundo mejora el aprendizaje automático al dar a las máquinas la capacidad de elegir entre un conjunto de algoritmos, que ofrecen una variedad de respuestas, y actuar sobre las conclusiones determinadas por una variedad de combinaciones. El Deep Learning estructura los algoritmos en capas, para crear una red neuronal artificial, que puede aprender y tomar decisiones por sí misma. Estas redes neuronales identifican patrones y clasifican diferentes tipos de información. Las diferentes capas de las redes neuronales sirven como filtro, yendo desde los elementos más generales a los más sutiles, aumentando la probabilidad de detectar y generar un resultado correcto. Por tanto, cuando un sistema de Deep Learning tiene que reconocer un objeto, lo compara con aquellos que ya conoce.

El Deep Learning es una rama del Machine Learning, y, aunque ambos se encuadran dentro de la Inteligencia Artificial, el Deep Learning va mucho más allá, buscando emular la forma de aprendizaje de los humanos.

Actualmente, el aprendizaje profundo se usa ampliamente en el sector industrial, ya que permite el análisis de una gran cantidad de datos para descubrir patrones importantes y hacer predicciones precisas.

En la figura 3 se representa en la parte superior el proceso de aprendizaje de un modelo de Machine Learning y, en la parte inferior, el aprendizaje de un modelo de Deep Learning para resolver un mismo problema.

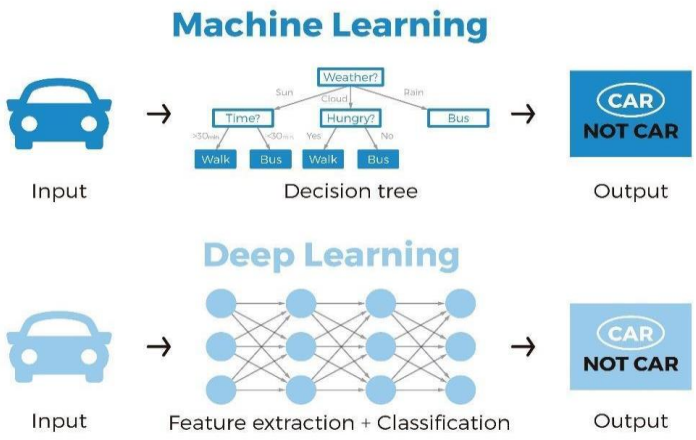


Figura 3. Machine Learning vs Deep Learning.

Fuente: https://blog.bismart.com/hs-fs/hubfs/02-MachinelearningVSDeeplearning_Mesa%20de%20trabajo%201%20copia%202_Mesa%20de%20trabajo%201%20copia%202.jpg?width=1755&name=02-MachinelearningVSDeeplearning_Mesa%20de%20tr

Lo cierto es que tanto el Machine Learning como el Deep Learning imitan la forma de aprender del cerebro humano. Su principal diferencia es el tipo de algoritmos que se usan en cada caso, aunque el Deep Learning, como se dijo anteriormente, se parece más al aprendizaje humano por su funcionamiento como neuronas. El Machine Learning acostumbra a usar árboles de decisión y el Deep Learning redes neuronales, que están más evolucionadas. Además, ambos pueden aprender de forma supervisada o no supervisada.

5.4 Redes neuronales Artificiales

Las redes neuronales artificiales están basadas en el funcionamiento de las redes de neuronas biológicas. Las neuronas que todos tenemos en nuestro cerebro están compuestas de dendritas, el soma y el axón. En el caso de las dendritas, estas se encargan de captar los impulsos nerviosos que emiten otras neuronas. Estos impulsos, se procesan en el soma y se transmiten a través del axón, que emite un impulso nervioso hacia las neuronas contiguas.

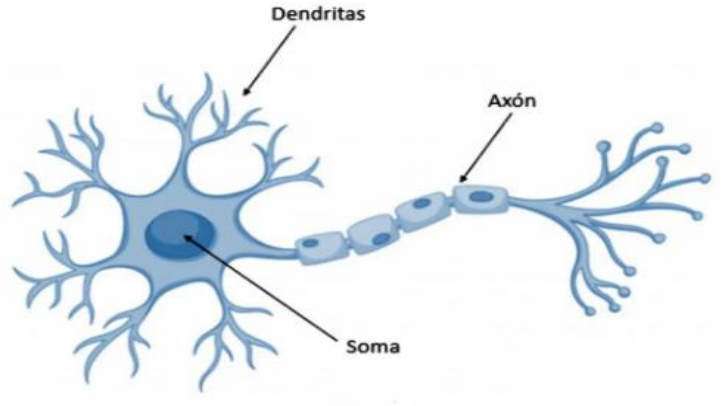


Figura 4. Neurona biológica.

Fuente: https://www.xeridia.com/wp-content/uploads/drupal-files/contenidos/blog/red_neuronal.jpg.webp

En el caso de las neuronas artificiales, la suma de las entradas multiplicadas por sus pesos asociados determina el “impulso nervioso” que recibe la neurona. Este valor se procesa en el interior de la célula mediante una función de activación que devuelve un valor que se envía como salida de la neurona.

Del mismo modo que nuestro cerebro está compuesto por neuronas interconectadas entre sí, una red neuronal artificial está formada por neuronas

artificiales conectadas entre sí y agrupadas en diferentes niveles que denominamos capas.

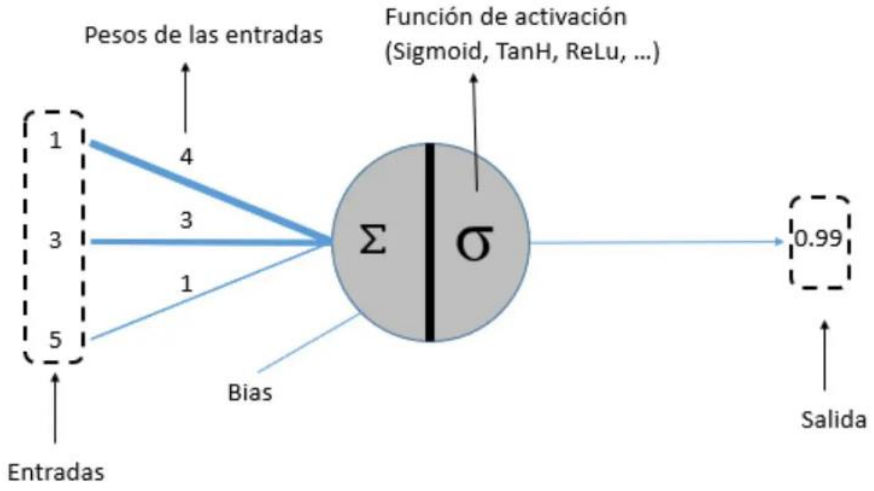


Figura 5. Neurona artificial.

Fuente: https://www.xeridia.com/wp-content/uploads/drupal-files/contenidos/blog/neurona_artificial.jpg.webp

Una capa es un conjunto de neuronas cuyas entradas provienen de una capa anterior (o de los datos de entrada en el caso de la primera capa) y cuyas salidas son la entrada de una capa posterior.

En la figura 6 se puede observar una red con cuatro capas:

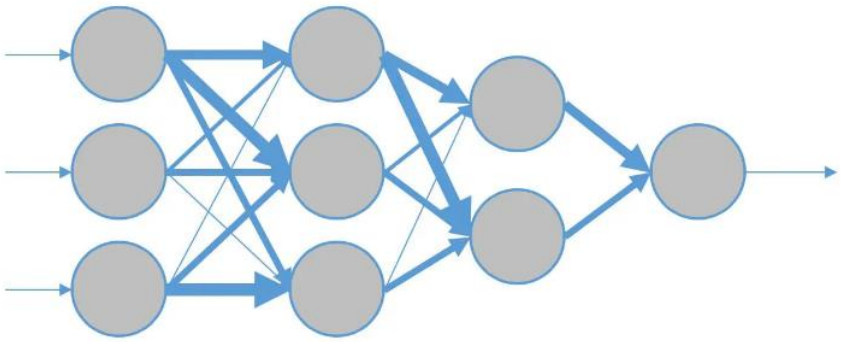


Figura 6. Red neuronal con cuatro capas.

Fuente: https://www.xeridia.com/wp-content/uploads/drupal-files/contenidos/blog/capas_neurona_artificial.jpg.webp

Las neuronas de la primera capa (capa de entrada) reciben como entrada los datos reales que alimentan a la red neuronal. Mientras que la salida de la última capa (capa de salida) es el resultado visible de la red. Las capas que se sitúan entre la capa de entrada y la capa de salida se conocen como capas ocultas ya que se desconoce tanto los valores de entrada como los de salida.

Una red neuronal, por lo tanto, siempre está compuesta por una capa de entrada, una capa de salida (si solo hay una capa en la red neuronal, la capa de entrada coincide con la capa de salida) y puede contener 0 o más capas ocultas. El concepto de Deep Learning nace a raíz de utilizar un gran número de capas ocultas en las redes.

5.4.1 Modelo y arquitectura de una neurona artificial

Como se expresó con anterioridad, una neurona es un procesador elemental tal que a partir de un vector de entrada procedente del exterior o de otras neuronas, proporciona una única respuesta o salida.

Para analizar el funcionamiento de una red neuronal se utiliza la arquitectura más simple, es decir, una única neurona.

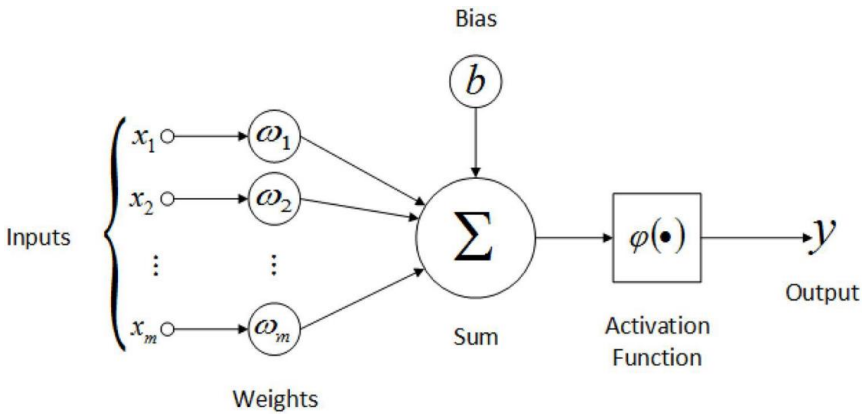


Figura 7. Estructura de una neurona artificial.

*Fuente: https://miro.medium.com/max/3000/0*jtG_gm7tGNoFmbTy*

Como se puede ver en la figura 7 cada entrada de la neurona tiene un valor (x_m) que se asocia a un peso (w_m) y este peso se multiplica por el valor de la entrada. El concepto de peso es muy importante dado que son los valores que se irán ajustando durante el entrenamiento de la red. El paso siguiente es realizar la suma de todos los valores de entrada multiplicados por sus respectivos pesos y luego a esta suma se le aplica una función de activación (φ). Este cálculo es representado por: □ = □(∑□ □=1 □□. □□)

5.4.2 Sesgo/Bias

En la figura 7 se puede ver que antes de aplicar la función de activación a la suma de las entradas por los pesos, se le agrega un bias o sesgo (b), este término puede

verse como una falsa neurona que siempre toma el valor 1 y se agrega a la capa de la red con su correspondiente peso. Este bias da a la función la capacidad de desplazarse hacia la derecha o izquierda permitiendo un mejor ajuste de la función para lograr un aprendizaje exitoso, ya que con el ajuste de los pesos solo se modifica la pendiente o inclinación de la función. Al agregar el sesgo la ecuación 1 se corrige a: $\square = \square(\square + \sum_{\square=1}^{\square} \square_{\square}.\square_{\square})$

5.4.3 Función de activación

La función de activación devuelve una salida que será generada por la neurona dada una entrada o conjunto de entradas. Cada una de las capas que conforman la red neuronal tienen una función de activación que permitirá reconstruir o predecir. Las funciones de activación se dividen en dos tipos como: lineal y no lineal. Para el caso de Deep Learning las funciones de activación son no lineales, ya que si se opta por una función de activación lineal la red se comporta como una sola neurona y solo es capaz de resolver problemas muy simples.

Entre las funciones de activación no lineales existen varios tipos, en las más tradicionales se encuentran la función sigmoide y la función tangente hiperbólica, pero en la actualidad se utilizan funciones de activación como Relu (unidad lineal rectificada) o algún derivado de esta como la función Elu y Leaky Relu. En la figura 8 se puede observar el comportamiento de cada una de estas funciones de activación.

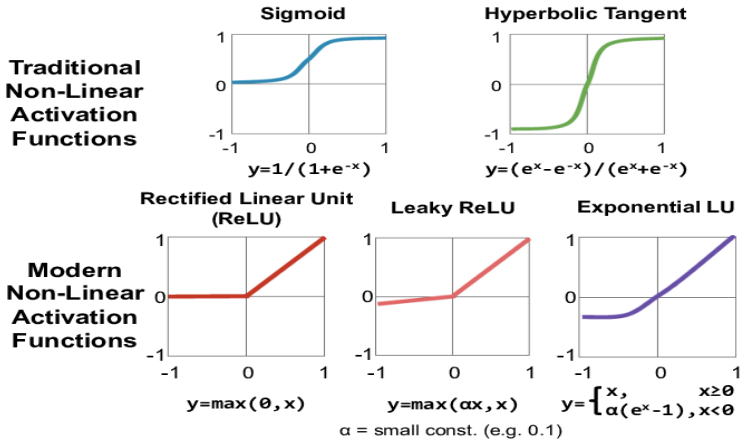


Figura 8. Funciones de activación.

Fuente: https://ignaciogavilan.com/wp-content/uploads/2020/05/Various-forms-of-non-linear-activation-functions.png

5.4.4 Entrenamiento de una red neuronal artificial

Este proceso corresponde al aprendizaje de la red neuronal y consiste en ir ajustando los pesos para que los valores de entrada de la red permitan obtener la salida esperada. Al tratarse de aprendizaje supervisado se parte del indicio de que para cada valor de entrada se conoce el valor de la salida esperado, de esta forma en cada iteración del entrenamiento se ajustarán poco a poco los pesos de cada neurona de modo que, ante todas las entradas, las salidas obtenidas sean las correctas.

El proceso de aprendizaje se puede ver como un proceso iterativo de ida y vuelta por las distintas capas de la red neuronal, siendo sus pasos más destacados la propagación hacia adelante (forward propagation) y la propagación hacia atrás (backpropagation).

5.4.5 El proceso de aprendizaje

La fase de propagación hacia adelante o forward propagation consiste en pasar los datos de entrada a través de toda la red neuronal hasta la capa de salida donde se calcula el valor de la predicción, es decir, cada dato de entrada es pasado por las capas intermedias de la red donde se realiza la transformación. El resultado obtenido es pasado como entrada de la siguiente capa, cuando los datos cruzan toda la capa oculta se pasan los valores de las transformaciones de la última capa oculta a la capa de salida para realizar la predicción.

El paso siguiente consiste en utilizar una *función de pérdida* (loss) o costo para estimar el error y así poder comparar y medir si el resultado obtenido fue bueno/malo comparado con el resultado esperado (al tratarse de aprendizaje supervisado los datos están etiquetados por lo tanto se conoce el valor esperado).

En el caso ideal se espera que el error sea cero, o sea que coincida el valor predicho con el esperado, para ello a medida que se entrena el modelo se irán ajustando los pesos de las interconexiones de las neuronas de manera automática hasta obtener buenas predicciones. Entre las funciones de error más comunes para este tipo de arquitectura de redes neuronales se tiene el error cuadrático medio (MSE) siendo una de las más utilizadas, error absoluto medio (MAE) y el error cuadrático logarítmico medio (MSLE).

Después de realizar el cálculo de error la información se propaga hacia atrás (backpropagation) por la red y durante esta fase la información se propaga desde la capa de salida hacia todas las neuronas de las capas ocultas, que contribuyeron

directamente en la obtención del valor de salida. Se debe tener en cuenta que cada neurona de la capa oculta solo recibe una fracción del valor total del error, basándose aproximadamente en la contribución relativa que haya aportado cada neurona al valor de salida, este proceso se repite a lo largo de todas las capas.

Después de terminada la propagación hacia atrás se ajustan los pesos de las conexiones entre las neuronas. Lo que se busca con esto es que el error sea lo más cercano a cero en la próxima predicción que realice el modelo. Una de las técnicas más utilizadas para lograrlo es la llamada “gradient descent” o método del gradiente descendiente, con esta técnica los pesos se modifican en pequeños pasos basados en la tasa de entrenamiento (learning rate) con la ayuda de la derivada parcial o gradiente de la función de pérdida. Esto permite ver en qué dirección crece la función, por lo tanto, si se utiliza el negativo del gradiente se puede saber la dirección en la que la función decrece hacia un mínimo. Se busca llegar al mínimo global de la función de error, este trabajo se realiza generalmente en lotes de datos (batches) durante las sucesivas iteraciones (epochs) de entrenamiento con todos los datos que le son pasados a la red como entradas.

5.4.6 Método del gradiente descendiente

El optimizador “gradient descent” es la base de muchos otros optimizadores y uno de los algoritmos de optimización más comunes en Machine Learning y Deep Learning. El gradiente es la generalización de la derivada y es el conjunto de todas las derivadas parciales de una función. En el caso de ML se hace foco en el gradiente de la función de pérdida o costo. El proceso consiste en encadenar las derivadas de la función de pérdida de cada capa oculta a partir de las derivadas de la función de su capa superior, incorporando su función de activación en el cálculo, motivo por el cual las funciones de activación utilizadas deben ser derivables. En cada iteración luego de que todas las neuronas obtienen el valor del gradiente de la función de pérdida se actualizan los valores de los parámetros. Se debe tener en cuenta que el gradiente siempre apunta en el sentido en el que se incrementa el valor de la función, por lo tanto, se utiliza el negativo del gradiente para obtener el sentido en el que la función de pérdida tiende a disminuir. Por lo que, el sentido viene dado por el negativo del gradiente y la magnitud de este cambio está dada por el valor del gradiente multiplicado por un hiperparámetro llamado Learning Rate.

5.4.6.1 Optimizadores

Si bien el método de “gradient descent” es el más clásico de los optimizadores, no es el único existente para optimizar la función de error dentro del mundo del Deep Learning, estos optimizadores presentan ciertas ventajas respecto al uso del algoritmo del gradiente descendiente, pero a costa de un mayor costo computacional en su implementación. Algunos de ellos son los siguientes:

✓ SGD: Stochastic Gradient Descent (descenso estocástico del gradiente) este es un caso particular del método del gradiente descendiente, que consiste en aplicar un descenso por gradiente a un solo ejemplo por iteración, o sea, aplicar gradient descent a lotes de datos de tamaño 1. Luego al comenzar la siguiente época de entrenamiento se mezclan aleatoriamente los datos de entrenamiento.

✓ Momentum: es una modificación del descenso del gradiente que da la posibilidad de incluir inercia o momento. Recuerda el incremento aplicado a los pesos en cada iteración y determina la actualización siguiente basándose en una combinación lineal entre el gradiente y esos incrementos anteriores, logrando de esta forma suavizar las oscilaciones en torno al mínimo durante la convergencia del algoritmo gradient descent.

✓ RMSprop: este optimizador al igual que el anterior pretende suavizar las oscilaciones durante la convergencia del método. Esto lo lleva a cabo ajustando el LR de manera automática. En cada iteración cuando se realiza la actualización de los pesos se escoge un valor diferente para la tasa de aprendizaje. Es un método adaptativo, ya que el LR se ajusta durante el entrenamiento.

✓ Adam: es un optimizador que combina los dos métodos anteriores, Momentum y RSMprop.

5.4.7 Hiperparámetros

Para comenzar se debe aclarar la diferencia entre parámetros e hiperparámetros de una red neuronal.

Los parámetros son los valores que se estiman durante el proceso de entrenamiento partiendo del conjunto de datos y no son definidos manualmente. La estimación de los parámetros es una de las partes más importantes y clave en los modelos de aprendizaje automático, ya que es la parte del modelo que aprende de los datos y son necesarios para realizar predicciones correctas, además definen la capacidad del modelo para resolver un cierto problema. La forma habitual de estimar

estos parámetros es la utilización de un optimizador como “gradient descent”, un ejemplo de parámetro son los pesos de las interconexiones.

Mientras que los hiperparámetros del modelo son los valores de las configuraciones utilizados durante el proceso de entrenamiento, estos valores no pueden ser estimados a partir de los datos y son ingresados manualmente por el desarrollador. El valor óptimo de un hiperparámetro no se conoce al principio para un problema dado, por lo tanto, se utilizan valores basados en reglas genéricas o valores que han tenido buenos resultados en problemas similares o, también, se suele optar por buscar la mejor opción mediante prueba y error. Una buena opción para encontrar estos hiperparámetros es la utilización de la validación cruzada.

Existen hiperparámetros tanto a nivel de estructura y topología de la red (como el número de capas o el número de neuronas de cada capa, la función de activación elegida, entre otras) e hiperparámetros a nivel de algoritmos de aprendizaje como la tasa de aprendizaje (learning rate), el número de épocas de entrenamiento (epochs), tamaño de lote (batch size), entre algunos otros.

Al entrenar un modelo de aprendizaje automático se fijan los valores de los hiperparámetros para que con estos se obtengan los parámetros.

✓ Batch size: cuando el set de datos es muy grande es conveniente pasarlos a la red en lotes más reducidos, se suele dividir el set de datos original en subconjuntos que serán pasados por la red para el entrenamiento. Generalmente, se emplean potencias de 2 como tamaño de batches. El tamaño óptimo de este hiperparámetro depende de varios factores, entre ellos la memoria del CPU/GPU, que se utilice para hacer los cálculos.

✓ Epochs: indica cada una de las veces en la que todos los datos de entrenamiento pasarán por la red neuronal en el proceso de aprendizaje, cuando todos los paquetes o batches han pasado una vez por la red se cumple una época de entrenamiento. Como el proceso de minimización de la función de error es iterativo se necesitan varias épocas para entrenar la red. Para definir el número adecuado de épocas de entrenamiento se suele ir incrementando el número de épocas hasta que el porcentaje de acierto con los datos de validación empieza a decrecer.

✓ Learning rate: este es un hiperparámetro que controla la velocidad con la que se avanza en la optimización de la función de error durante el entrenamiento. El vector del gradiente tiene una dirección y una magnitud y el algoritmo de “gradient descent” multiplica la magnitud del gradiente por un escalar. Este escalar es la tasa de aprendizaje o “learning rate”, que sirve para obtener el siguiente punto en la función, determinando el paso que realiza el algoritmo en una iteración del

entrenamiento dada. El valor de este hiperparámetro difiere en cada problema a analizar y su correcta elección marca la diferencia entre un buen aprendizaje y uno no tan bueno, por lo que se busca que el modelo logre una convergencia óptima lo más rápido posible.

✓ Gamma: el parámetro gamma es el inverso de la desviación estándar del kernel RBF[4] , que se utiliza como medida de similitud entre dos puntos. Intuitivamente, un valor gamma pequeño define una función gaussiana con una varianza grande. En este caso, dos puntos se pueden considerar similares, aunque estén lejos el uno del otro. Por otro lado, un valor gamma grande significa definir una función gaussiana con una varianza pequeña y en este caso, dos puntos se consideran similares solo si están cerca uno del otro.

✓ Lr step size: es el número de épocas entre cada decaimiento de *learning rate* por *gamma*.

5.5 Redes neuronales convolucionales

Las redes neuronales convolucionales son un algoritmo de Deep Learning que está diseñado para trabajar con imágenes, tomando estas como input, asignándole importancia (peso) a ciertos elementos en la imagen para así poder diferenciar unos de otros. Estas redes contienen varias hidden layers, donde las primeras pueden detectar líneas, curvas y así se van especializando hasta poder reconocer formas complejas como un rostro, siluetas, etc. Las tareas comunes de este tipo de redes son detección o categorización de objetos, clasificación de escenas y clasificación de imágenes en general. Para realizar dichas tareas, la red toma como entrada los píxeles de una imagen. Por ejemplo, si se tuviera una imagen con apenas 28×28 píxeles de alto y ancho, eso equivale a 784 neuronas. Y eso es si solo se tiene 1 color (escala de grises). Si fuera una imagen a color, se necesitan 3 canales (red, green, blue) y entonces se usan 28x28x3 = 2352 neuronas de entrada las cuales formarán parte de la capa de entrada.

[4] Función gaussiana de kernel popular utilizada en varios algoritmos de aprendizaje kernelizados. En particular, se usa comúnmente en la clasificación de modelos de aprendizaje supervisado con algoritmos de aprendizaje asociados (máquinas de vectores de soporte) que analizan datos para clasificación y análisis de regresión.

5.5.1 Arquitectura de las redes neuronales convolucionales

La arquitectura de una red neuronal convolucional (RNC) se puede separar en dos etapas, por un lado, la etapa de extracción de características que a su vez está compuesta por una o más capas convolucionales en donde se realizan pasos fundamentales y, por otro lado, la etapa de clasificación, también llamada capa completamente conectada, que comienza con un aplanamiento y continua con una red neuronal tradicional totalmente conectada, en donde se realizará la clasificación final.

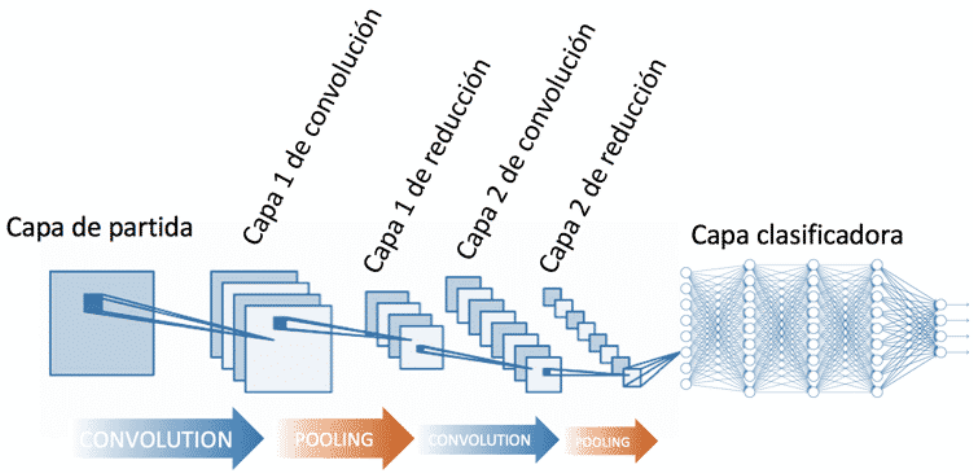


Figura 9. Arquitectura RNC.

Fuente: https://www.diegocalvo.es/wp-content/uploads/2017/07/red-neuronal-convolucional-arquitectura.png

5.5.2 Convolución

La convolución es el pilar de la arquitectura de las redes convolucionales y para llevarla a cabo, se realiza una operación matemática que describe cómo fusionar dos conjuntos de información. La diferencia fundamental entre una capa densamente conectada y una capa especializada en la operación de convolución, llamada capa de convolución, es que la capa densa aprende patrones globales tomados de todo el dato de entrada, mientras que las capas convolucionales aprenden patrones locales en pequeñas ventanas de dos dimensiones.

La capa de entrada de la RNC tiene tantas neuronas como píxeles tiene la imagen de entrada (cada píxel se considera una neurona). En el proceso de convolución se transforman los datos de entrada usando un producto escalar entre una región de las neuronas (sub-matriz de la imagen) de la capa de entrada y la matriz de pesos

asignados (kernel). Por lo general, como salida se obtiene una nueva imagen convolucionada con dimensiones espaciales menores o iguales a las de la imagen de entrada, la profundidad de la capa convolucional está dada por la cantidad de filtros que se apliquen a la imagen de entrada. En otras palabras, el resultado del proceso de convolución devuelve un mapa de las características de la imagen original.

Existen también hiperparámetros específicos que se deben determinar en una RNC para establecer la disposición espacial y el tamaño del volumen de salida de una capa convolucional.

✓ Tamaño de kernel: hace referencia a las dimensiones de la matriz de los filtros a utilizar, generalmente este tamaño es mucho más chico que el tamaño de la imagen de entrada.

✓ Stride o paso: este hiperparámetro refiere a la separación con la que se desplazan los filtros a través de la imagen. Mientras más grande sea el valor del stride más chica será la salida resultante.

✓ Zero-Padding: cantidad de “anillos” de ceros agregados alrededor de la imagen de entrada para evitar reducir el tamaño de la salida, es decir, que la matriz de salida tenga las mismas dimensiones que la matriz de entrada.

En la figura 10 se muestra cómo el filtro se desplaza a través de los datos de entrada para producir la capa convolucionada. Esta ventana va desplazándose (de izquierda a derecha y de arriba a abajo) a lo largo de toda la capa de neuronas realizando el producto escalar entre las matrices, cada valor que se obtiene como resultado corresponderá a un elemento de la nueva matriz generada por la convolución (o pixel de la imagen filtrada).

El tamaño de la matriz de salida (M_{sal}) viene dado por: $M_{sal} = M_{ent} – K + 1$

Siendo K el tamaño del filtro y M_{ent} el tamaño de la matriz de entrada.

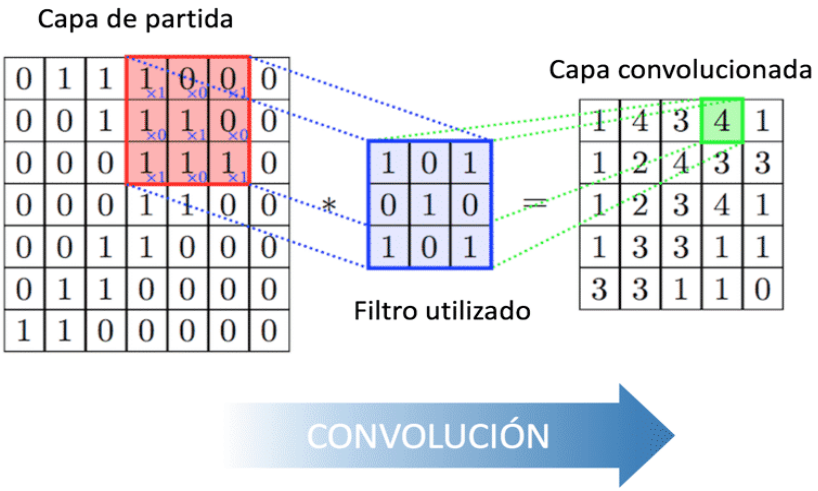


Figura 10. Ejemplo de convolución.

Fuente: https://www.diegocalvo.es/wp-content/uploads/2017/07/convoluci%C3%B3n.png

5.5.3 Capa de reducción o pooling

La capa de reducción o pooling se coloca generalmente después de la capa convolucional. Su utilidad principal radica en la reducción de las dimensiones espaciales (ancho x alto) del volumen de entrada para la siguiente capa convolucional. No afecta a la dimensión de profundidad del volumen. La operación realizada por esta capa también se llama reducción de muestreo, ya que la reducción de tamaño conduce también a la pérdida de información. Sin embargo, una pérdida de este tipo puede ser beneficioso para la red por dos razones:

✓ La disminución en el tamaño conduce a una menor sobrecarga de cálculo para las próximas capas de la red.

✓ También trabaja para reducir el sobreajuste.

La operación que se suele utilizar en esta capa es max-pooling, que divide a la imagen de entrada en un conjunto de rectángulos y, respecto de cada subregión, se va quedando con el máximo valor.

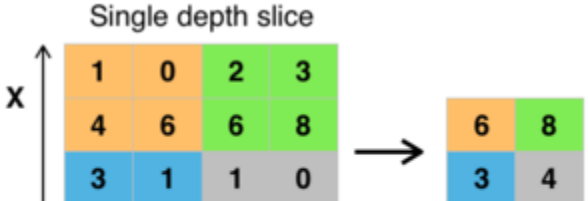


Figura 11. Operación max-pooling.
Fuente: https://relopezbriega.github.io/images/Max_pooling.png

5.5.4 Clasificación

La etapa de clasificación es la etapa final de una CNN. Esta fase consiste, en primer lugar, en realizar un aplanamiento o “flatten” de los mapas de características y luego pasarlos por una red totalmente conectada, que se encargará de realizar la clasificación para determinar la clase a la que pertenece la imagen de entrada. El aplanamiento consiste en tomar todos los mapas de características obtenidos en la última capa convolucional y pasarlos por una función que transformará estos datos tridimensionales en un vector de una dimensión. Este vector será utilizado como entrada de una red totalmente conectada. Además, en este tipo de red, cada píxel se considera como una neurona separada al igual que en una red neuronal regular. Esta tendrá tantas neuronas como el número de clases que se debe predecir.

5.6 Redes para detección de objetos: Faster R-CNN

En el apartado 5.5 se han explicado los elementos básicos necesarios para la clasificación de imágenes. Por otra parte, también existen sistemas capaces no solo reconocer y clasificar cada objeto en una imagen, sino que además de localizarlas, por ejemplo, dibujando un cuadro alrededor del objeto que se desea detectar. Una forma eficiente de llevar a cabo una detección de objetos utilizando redes neuronales es utilizar un tipo de arquitectura basada en propuestas de región de interés.

En este trabajo se ha utilizado una determinada arquitectura, la cual se denomina Faster R-CNN y se compone de los siguientes módulos:

1. Un extractor de características. Una red convolucional profunda compuesta por los bloques básicos introducidos en el apartado 5.5.

2. Una red de propuesta de región de interés. Es una red convolucional profunda que toma la salida del extractor de características y devuelve regiones que contienen candidatos a ser un objeto que se desea detectar.

3. El clasificador Fast R-CNN. Utiliza las regiones propuestas para asignar las clases referentes a cada objeto que aparece en la imagen.

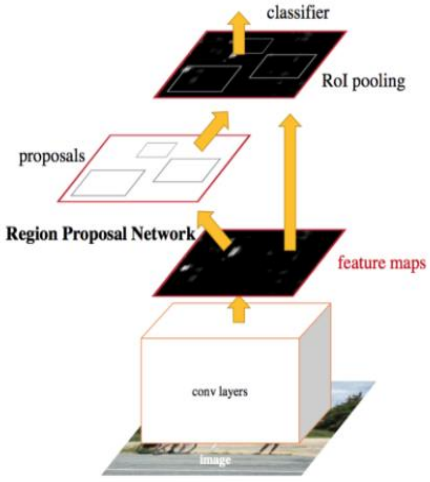


Figura 12. Arquitectura del módulo Faster-RCNN.

Fuente: https://images4.programmerclick.com/956/40/408f277398f71389b4f3e57bfb7688dc.JPEG

5.6.1 Extractor de características

Tal como se mencionó previamente, el extractor de características, es una red convolucional profunda básica. Por otra parte, en su reemplazo, se puede utilizar un extractor de características particular llamado *Feature Pyramid Network* (FPN), el cual genera múltiples capas de mapas de características (de múltiples escalas) con información de mejor calidad que la pirámide de características normal para la detección de objetos. Estos mapas de características son de múltiples escalas generando una “pirámide” de mapa de características.

Figura 13. Representación de FPN.

*Fuente: https://miro.medium.com/max/1000/1*aMRoAN7CtD1gdzTaZIT5gA.png*

FPN se compone de un camino de abajo hacia arriba y de arriba hacia abajo. La ruta de abajo hacia arriba es la red convolucional habitual para la extracción de características. A medida que subimos, la resolución espacial disminuye. Con más estructuras de alto nivel detectadas, el valor semántico de cada capa aumenta.

Si bien las capas reconstruidas son fuertes desde el punto de vista semántico, las ubicaciones de los objetos no son precisas después de todo el submuestreo y el sobremuestreo. Es por eso que se agregan conexiones laterales entre las capas reconstruidas y los mapas de características correspondientes para ayudar al detector a predecir mejor la ubicación. También actúa como conexiones de salto para facilitar el entrenamiento similar a lo que hace la arquitectura *ResNet* (se explica en el apartado 5.8.1*)*.

En resumen, FPN genera una pirámide de mapas de características que luego de aplicar RPN se generarán los ROIs. (ambos términos se explicarán en los próximos apartados).

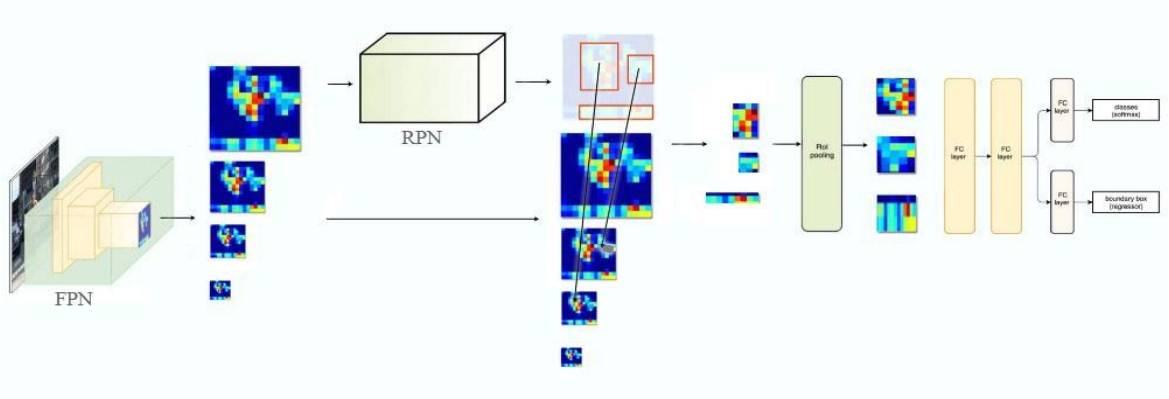


Figura 14. FPN con Faster R-CNN.

*Fuente: https://miro.medium.com/max/1400/1*Wvn0WG4XZ0w9Ed2fFYPrXw.jpeg*

5.6.2 Redes de propuesta de región de interés (RPN)

Una Red de propuesta de región o *región proposal network* (RPN) toma una imagen como entrada y devuelve un conjunto de propuestas de objetos rectangulares, cada una con una puntuación que mide si esa región contiene un objeto o es parte del fondo. Para generar propuestas desliza una pequeña red sobre la salida del módulo que extrae el mapa de características. Esta pequeña red toma como entrada una ventana espacial n x n del mapa de características convolucionales. Para localizar un objeto en el mapa de características, se realiza un barrido de diferentes cuadros delimitadores, los cuales se conocen como anclajes, cajas de anclaje o *anchor boxes*.

5.6.2.1 Anclajes

Se colocan sobre la imagen simétricamente distribuidos los puntos de anclaje. Sobre cada punto de anclaje se colocarán diferentes anclajes, y cada uno de estos será candidato de contener un objeto de interés. Sobre cada punto de anclaje se coloca un número k de anclajes, a diferentes escalas y diferentes relaciones de aspecto. En la configuración predeterminada de Faster R-CNN, hay 9 anclajes en cada punto de anclaje de una imagen. En la siguiente figura se muestran 9 anclajes en la posición (320, 320) de una imagen con tamaño (600, 800), 3 escalas y 3 relaciones de aspecto.

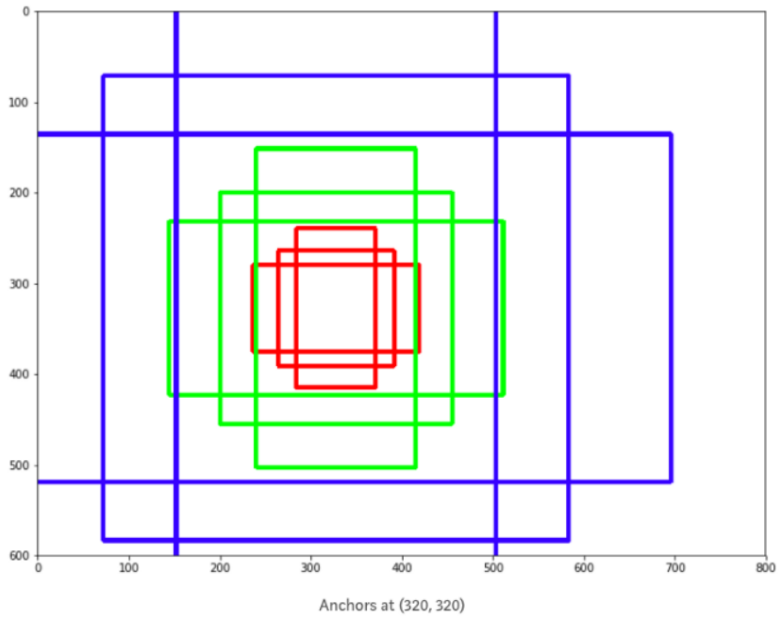


Figura 15. Ejemplo de propuesta de regiones en una imagen mediante anclajes.
*Fuente: https://miro.medium.com/max/930/1*S-0--pfK-uDU0X_kHuJEoA.png*

5.6.2.2 Entrenamiento y función de coste

Para poder entrenar la RPN del Faster-RCNN primero se asigna una clase binaria a cada anclaje, la cual indica si lo que contiene es un objeto o es el fondo de la imagen. Para activar los anclajes que contienen un objeto se utiliza un umbral en el valor de la intersección entre la unión del anclaje y el *ground-truth* (es decir, el cuadro delimitador etiquetado del conjunto de prueba que especifica en qué parte de la imagen está el objeto). Para eso se realiza la Intersección sobre la Unión (IoU) o *índice de Jaccard:* □ (□, □) = |□ ∩ □| / |□ ∪ □|. Este índice es una métrica que indica cuánto se parecen dos conjuntos, y en detección de objetos, cuanto se acerca el área de una detección, o el anclaje en este caso, a la etiqueta en los datos. A los anclajes que tengan un IoU>0.7 se les asigna una etiqueta positiva, en el caso especial en el que ningún anclaje supere el valor de 0.7 se etiquetan como objetos los anclajes con mayor valor IoU. Los anclajes con un valor de IoU<0.3 se etiquetan como fondo y el resto, los anclajes que no son ni fondo ni objeto, no se utilizan para el entrenamiento.

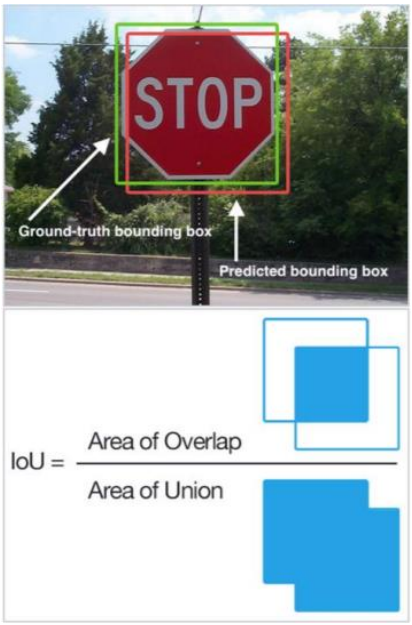


Figura 16. Ejemplo de las áreas de intersección y unión en el par ground-truth bounding box y predicted bounding box (anclaje predicho).

Fuente: https://images4.programmerclick.com/769/ce/ce6157056cf28a4985ef08186af7a0f1.JPEG y https://images4.programmerclick.com

Una vez se ha especificado cuáles serán los datos con los que trabajará la RPN, para definir la función de coste es imprescindible tener en cuenta dos aspectos: el error de clasificación que supone el clasificar como fondo un objeto y viceversa, y el error de regresión que representa cuanto se acerca un anclaje a la etiqueta real.

La función de coste para una imagen se define mediante la siguiente expresión:

$C(\{p_i\}, \{t_i\}) = 1/N_{cls} \sum_{\square} \quad C_{cls}(p_i, p_i^*) + \lambda\ 1/N_{reg} \sum_{\square} \quad p_i^*\ C_{reg}(t_i, t_i^*)$

donde:

i: hace referencia al anclaje i.

p_i: es la probabilidad predicha de que las anclas contengan un objeto o no.

p_i^*: es 1 si el anclaje contiene un objeto y 0 en caso de que no.

t_i: es un vector de cuatro componentes que representan las coordenadas del cuadro delimitador (o anclaje) predicho.

t_i^*: es el cuadro delimitador del ground-truth.

C_{cls}: es la pérdida de clasificador (log loss, $C_{cls} = -p_i^* \log(p_i)$).

C_{reg}: el coste de localización C_{reg} es la función $smooth_{L1}\ (t_i - t_i^*)$.

$$\text{smooth}_{L_1}(x) = \begin{cases} \frac{x^2}{2} & si \quad |x| < 1 \\ |x| - \frac{1}{2} & si \quad |x| \geq 1 \end{cases}$$

Figura 17. Función smoothL1(x).

Fuente: R. Girshick, "Fast R-CNN," 2015 IEEE International Conference on Computer Vision (ICCV), 2015, pp. 1440-1448, doi: 10.1109/ICCV.2015.169.

El hecho de que el coste de localización C_{reg} este multiplicándose con el valor p_i^* implica que únicamente contribuyen al ajuste de la localización los anclajes que contienen un objeto, ya que no tiene sentido llevar a cabo un ajuste en el cuadro delimitador en un área en la que no debería existir detección alguna.

N_{cls}: es el parámetro de normalización del tamaño de mini lote (~ 256).

N_{reg}: es el parámetro de normalización de regresión (igual al número de ubicaciones de anclaje (~ 2400).

λ: es 10 el valor por defecto.

Dicha RPN no se encarga de asignar clases a los objetos, sino de devolver candidatos a ser un objeto, es decir, indicar que en una región concreta puede haber un objeto, sin especificar que es ese objeto. De la asignación de clases a los objetos se encargará el módulo RCNN en una fase posterior. También es importante destacar que una vez entrenada la RPN, los tamaños y formas de los anclajes son fijos. Durante el entrenamiento se ajusta cuáles serán esos tamaños y formas, pero una vez terminado el entrenamiento siempre se utilizarán las mismas relaciones de aspecto y tamaños de anclaje para proponer regiones que contengan un objeto. Al igual que en la clasificación, la tarea de ajustar el cuadro delimitador de manera personalizada para cada imagen la llevará a cabo el módulo RCNN. Dado que que muchos de los anclajes generalmente se superponen, como cabe esperar, muchas de las propuestas también se superponen sobre el mismo objeto. Para resolver el problema de las propuestas duplicadas, se utiliza el método llamado *Non-Maximum Suppression* (NMS). El NMS toma la lista de propuestas ordenadas por puntuación y descarta las propuestas que tienen un valor de IoU mayor que un umbral predefinido con la propuesta que tiene la puntuación más alta. Por último, después de eliminar las propuestas duplicadas, se seleccionan las N propuestas que mayor puntuación tienen, descartando el resto.

5.6.2.3 RoI Pooling

Después de haber localizado las regiones de interés con la RPN, estas serán introducidas al clasificador y regresor que se encargará de clasificar las regiones de interés en sus clases correspondientes, y de afinar la posición del cuadro delimitador que contiene los objetos en la imagen. El problema es que las regiones propuestas por la RPN serán de tamaños diferentes, y crear una estructura eficiente que trabaje con tamaños diferentes es complicado, para ello se lleva a cabo el pooling de la región de interés, o *region of interest pooling* (RoI pooling).

El RoI pooling consiste en fijar un número concreto q, partir las regiones de interés en q partes iguales, o lo más parecidas posibles, y realizar un max pooling (explicado en el apartado 5.5.3) en esas q partes, de esta manera, a la salida del pooling, todas las regiones de interés tendrán el mismo tamaño.

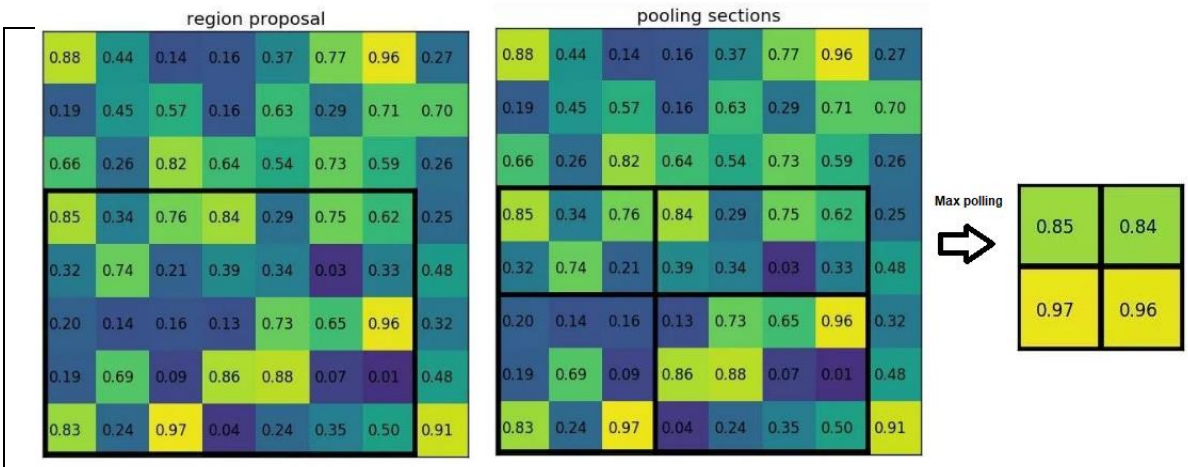


Figura 18. Ejemplo de RoI pooling para un tamaño de q = 4.
Fuente: https://deepsense.ai/wp-content/uploads/2017/02/x2.jpg.pagespeed.ic.3Lf_DPNXND.webp,https://deepsense.ai/wp-content/uploads/2017/02/x3.jpg.pagespeed.ic.PtYdkn0RSB.webp y https://deepsense.ai/wp-co

5.6.2 Clasificador Fast-RCNN

La última fase del Faster-RCNN es el clasificador de imágenes. El objetivo del clasificador es el de un clasificador de imágenes común (explicado en el apartado 5.5.4), recibe el mapa de características de las regiones propuestas como entrada y les asigna una clase. Se añade una clase extra, la cual hace referencia al fondo de la imagen, y se utiliza para descartar regiones mal propuestas. Además, en esta última fase del detector se modifica el cuadro delimitador que contendrá el objeto para ajustarlo más a los límites del objeto. Este módulo está formado por dos subredes:

1. Una capa totalmente conectada con N + 1 unidades donde N es el número total de clases y ese extra es para la clase de fondo.

2. Una capa completamente conectada con 4*N unidades. El objetivo es realizar una regresión del cuadro delimitador, por lo tanto, es necesario ajustar $\Delta centro_x$, $\Delta centro_y$, $\Delta ancho$, $\Delta altura$ para cada una de las N clases posibles. El procedimiento para entrenar el clasificador Fast-RCNN es similar al que se aplica en la RPN. En este caso, a las propuestas con un valor de IoU>0.5 con alguna clase se les asigna dicha clase, a las propuestas con un valor de IoU entre 0.1 y 0.5 se les asigna la clase fondo, y las propuestas con un valor menor a 0.1 se ignoran. Esto se debe a que al ser la fase más avanzada de la red se cuenta con tener propuestas suficientemente buenas, y el objetivo real y la tarea más complicada que debe llevar a cabo esta fase es clasificar los objetos. Los valores $\Delta centro_x$, $\Delta centro_y$, $\Delta ancho$ y

∆altura de la red de regresión se obtienen comparando los valores de salida con los valores del anclaje. Las funciones de coste en estas capas son las mismas que en la RPN, la función $smooth_{L1}$ para los valores de la regresión y el log loss para la clasificación.

5.7 Red neuronal: Haar Cascade

Haar cascade es un algoritmo que puede detectar objetos en imágenes, independientemente de su escala en la imagen y ubicación. La detección de objetos mediante este algoritmo está basada en el aprendizaje automático en el que se entrena una función en cascada a partir de muchas imágenes positivas y negativas que luego se usan para detectar objetos en otras imágenes. El algoritmo necesita muchas imágenes positivas (donde se encuentre el objeto a detectar) e imágenes negativas (donde no se encuentre el objeto a detectar) para entrenar al clasificador. Luego se necesita extraer características de él. Para esto, se utilizan las *características de Haar*.

Las características de Haar son funciones rectangulares simples de 2 dimensiones en las que se varía el tamaño y la posición de recuadros blancos y negros. Estas características se extraen buscando la diferencia entre la suma de los píxeles dentro del recuadro negro y la suma de los píxeles bajo el rectángulo blanco. Para ello, se aplican todas y cada una de las funciones en las imágenes de entrenamiento. Para cada característica, se encuentra el mejor umbral que clasificará el objeto a detectar en positivos y negativos. Dado que, habrá errores o clasificaciones erróneas se seleccionan las características con una tasa de error mínima, lo que significa que son las características que clasifican con mayor precisión las imágenes del objeto a detectar y las imágenes que no lo contengan.

Algunas características de este algoritmo son los siguientes:

✓ Son rápidas y pueden funcionar bien en tiempo real.

✓ No es tan precisa como lo son las técnicas modernas de detección de objetos como Faster R-CNN, SSD, etc.

✓ Son simples de implementar y requiere menos poder de cómputo.

Algunas limitaciones son:

✓ Alta detección de falsos positivos

✓ Menos preciso que las técnicas basadas en el aprendizaje profundo

✓ Ajuste manual de parámetros.

✓ No es fácil entrenar Haar Cascade en un objeto personalizado.

5.7.1 Harr Cascade y CNN

La relación entre el algoritmo Harr cascade y las redes neuronales convolucionales son las características de Haar nombradas en el inciso 5.7 ya que estas representan a la matriz de pesos o kernel de las CNN solo que los valores del kernel se determinan mediante el entrenamiento, mientras que una característica de Haar se determinan manualmente.

5.8 Transferencia de aprendizaje (Transfer Learning)

En muchas aplicaciones, las redes neuronales suelen entrenarse desde cero, y se basan únicamente en los datos de entrenamiento para ajustar sus parámetros. Sin embargo, a medida que se entrenan cada vez más redes para diversas tareas, es razonable buscar métodos que eviten tener que empezar desde cero y, en su lugar, pueda basarse en los resultados de las redes previamente entrenadas. En el caso de la detección de objetos, dado que las redes tienden a ser muy profundas, y por lo tanto el número de parámetros que se deben ajustar es muy elevado, para un entrenamiento eficiente se requiere una base de datos de entrenamiento muy extensa, al igual que un tiempo de entrenamiento muy alto. Por eso, es habitual partir de una red previamente entrenada para un propósito similar en lugar de inicializar los pesos y sesgos aleatoriamente. Las redes de detección de objetos que han sido entrenadas con conjuntos de datos grandes y para detectar una amplia variedad de objetos cuentan con la capacidad de distinguir e identificar múltiples formas, lo cual hace que partir de los parámetros de una red pre entrenada sea mucho más conveniente que entrenarla desde cero en muchos casos, sobre todo cuando no se cuenta con un gran número de ejemplos de entrenamiento. Para esta situación es bastante útil utilizar ajuste fino o *fine tuning*.

El fine tuning es una técnica de reutilización de modelos además de la extracción de características. El *fine tuning* consiste en descongelar algunas de las capas superiores de la base del modelo congelado en la red neuronal utilizada para la extracción de características y entrenar conjuntamente tanto la parte recién agregada del modelo como las capas superiores. Esta técnica se denomina fine

tuning, ya que ajusta ligeramente las representaciones más abstractas del modelo de ajuste fino que se reutiliza para que pueda volverse más relevante para el problema en cuestión.

En el próximo apartado, se realizará una breve descripción de las arquitecturas utilizadas para desarrollar modelos basados en *transfer learning*, que se investigaron o involucraron durante este proyecto.

5.8.1 ResNet50

ResNet significa Red Residual. Es una red neuronal innovadora que fue presentada por primera vez por Kaiming He, Xiangyu Zhang, Shaoqing Ren y Jian Sun en su artículo de investigación de visión artificial de 2015 titulado "Aprendizaje residual profundo para el reconocimiento de imágenes".

Cuando se trabaja con redes neuronales convolucionales profundas para resolver un problema relacionado con la visión por computadora, se suele adicionar muchas capas. Estas capas adicionales ayudan a resolver problemas complejos de manera más eficiente, ya que las diferentes capas pueden entrenarse para diferentes tareas para obtener resultados altamente precisos.

Si bien la cantidad de capas apiladas puede enriquecer las características del modelo, una red más profunda puede mostrar el problema de la degradación. En otras palabras, a medida que aumenta la cantidad de capas de la red neuronal, los niveles de precisión pueden saturarse y degradarse lentamente después de un punto. Como resultado, el rendimiento del modelo se deteriora tanto en los datos de entrenamiento como de prueba. Esta degradación no es el resultado de un sobreajuste. En cambio, puede ser el resultado de la inicialización de la red, la función de optimización o, lo que es más importante, el problema de la desaparición o explosión de gradientes. ResNet se creó con el objetivo de abordar este problema exacto. Las redes residuales profundas utilizan bloques residuales para mejorar la precisión de los modelos. El concepto de "saltar conexiones", que se encuentra en el núcleo de los bloques residuales, es la fuerza de este tipo de red neuronal. Estas conexiones de salto funcionan de dos maneras. En primer lugar, alivian el problema de la desaparición del gradiente al establecer un atajo alternativo para que pase el gradiente. Además, permiten que el modelo aprenda una función de identidad. Esto asegura que las capas superiores del modelo no funcionen peor que las capas inferiores. ResNet mejora la eficiencia de las redes neuronales profundas con más capas neuronales y minimiza el porcentaje de errores.

ResNet tiene muchas variantes que se ejecutan en el mismo concepto, pero tienen diferentes números de capas. Resnet50 se utiliza para indicar la variante que puede funcionar con 50 capas de redes neuronales. La red tiene un tamaño de entrada de imagen de 224 x 224.

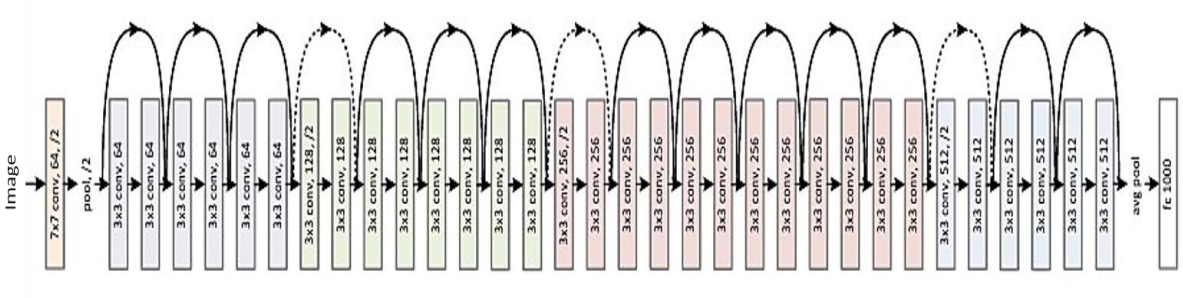


Figura 19. Arquitectura Resnet50.

Fuente: https://jananisbabu.github.io/ResNet50_From_Scratch_Tensorflow/images/resnet50.png

5.8.1.1 FPN + ResNet

Continuando la idea del apartado 5.6.1 respecto al extractor de características FPN, se comprendió que el mismo proporciona una ruta de arriba hacia abajo para construir capas de mayor resolución a partir de una capa rica en semántica.

Esta idea combinada con la arquitectura ResNet se puede visualizar en el siguiente gráfico:

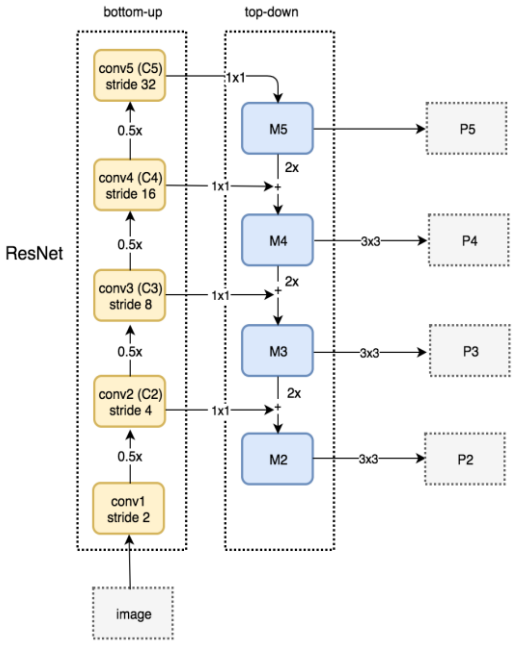


Figura 20. Arquitectura FPN con ResNet.

Fuente: https://images4.programmerclick.com/882/5f/5f2e57968d7921e47004cad98ad6a1b2.png

Observando la figura anterior se puede decir que la vía de abajo hacia arriba utiliza ResNet para construir dicha vía. La arquitectura se compone de muchos módulos

de convolución (conv $_i$ para i es igual a 1 a 5) cada uno tiene muchas capas de convolución. A medida que se avanza por la vía, la dimensión espacial se reduce a la mitad. La salida de cada módulo de convolución se etiqueta como C_i y luego se usa en la ruta de arriba hacia abajo.

Al iniciar la ruta que va de arriba hacia abajo, se aplica un filtro de convolución de 1 × 1 para reducir la profundidad del canal C5 a 256-d para crear M5. Esta se convierte en la primera capa de mapa de características utilizada para la predicción de objetos.

A medida que se avanza por la ruta de arriba hacia abajo, se aumenta la muestra de la capa anterior en 2 usando el muestreo ascendente de los vecinos más cercanos. Nuevamente se aplica una convolución de 1 × 1 a los mapas de características correspondientes en la ruta de abajo hacia arriba. Luego se agregan por elementos y se aplica una convolución de 3 × 3 a todas las capas fusionadas. Este filtro reduce el efecto de *aliasing* [5]cuando se fusiona con la capa muestreada.

Se repite el mismo proceso para “P3” y “P2”; sin embargo, se detiene en “P2” porque la dimensión espacial de C1 es demasiado grande, de lo contrario, ralentiza demasiado el proceso. Debido a que se comparte el mismo clasificador y regresor de caja de todos los mapas de características de salida, todos los mapas de características piramidales (P5, P4, P3 y P2) tienen canales de salida de 256-d.

5.8.2 MobilenetV3

MobileNet es una arquitectura propuesta por Google pensada principalmente para aplicaciones de visión móviles. Por tanto, se centra principalmente en reducir lo más posible la potencia computacional necesaria para el algoritmo, en consecuencia, busca una arquitectura muy sencilla a pesar de que esto le haga sacrificar algo de precisión y rendimiento. Esto lo consigue en gran medida gracias a la utilización de convoluciones separables por profundidad.

La última versión de MobileNet fue la V3 lanzada en 2019. La principal contribución de MobileNetV3 es el uso de *AutoML* para encontrar la mejor arquitectura de red neuronal posible para un problema determinado. Esto contrasta con el diseño artesanal de versiones anteriores de la arquitectura. Específicamente, MobileNetV3 aprovecha dos técnicas de AutoML: *MnasNet* y *NetAdapt*. MobileNetV3 primero busca una arquitectura gruesa usando MnasNet, que usa el aprendizaje por

[5] Aliasing, o solapamiento, es el efecto que causa que señales continuas distintas se tornen indistinguibles cuando se muestran digitalmente. Cuando esto sucede, la señal original no puede ser reconstruida de forma unívoca a partir de la señal digital.

refuerzo para seleccionar la configuración óptima de un conjunto discreto de opciones. Después de eso, el modelo ajusta la arquitectura utilizando NetAdapt, una técnica complementaria que recorta los canales de activación infrautilizados en pequeños decrementos.

Otra idea novedosa de MobileNetV3 es la incorporación de un bloque de compresión y excitación en la arquitectura central. La idea central de los bloques de compresión y excitación es mejorar la calidad de las representaciones producidas por una red mediante el modelado explícito de las interdependencias entre los canales de sus características convolucionales. Con este fin, se propone un mecanismo que le permite a la red realizar una recalibración de funciones, a través del cual puede aprender a usar información global para enfatizar selectivamente las funciones informativas y suprimir las menos útiles.

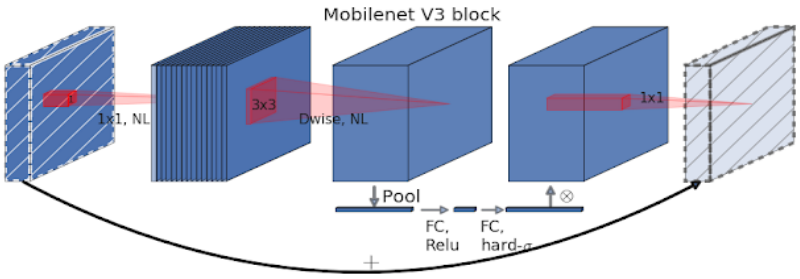


Figura 21. Arquitectura MobilenetV3.

Fuente: https://www.researchgate.net/profile/Giulia-Pazzaglia/publication/349512022/figure/fig4/AS:1004873269121026@1616591865215/Differences-between-MobileNetV2-and-MobileNetV3.ppm

5.9 Pasos para la detección de objetos

Para dar inicio al modelado y desarrollo de una red neuronal capaz de detectar objetos es necesario primero recolectar la mayor cantidad de imágenes posibles y que estas sean lo suficientemente descriptivas para obtener resultados deseados. Cuantos más tipos de elementos (clases) se desee detectar en una imagen, mayor tendrá que ser la cantidad de imágenes. Estas imágenes no deben ser muy grandes, ya que esto podría provocar lentitud en la detección. Luego de la recolección de imágenes es necesario etiquetarlas estableciendo las coordenadas donde están los objetos que se quiere detectar.

5.10 Datos de entrenamiento o training data

Los datos de entrenamiento son los datos que se usan para entrenar un modelo, en este caso, los datos serán imágenes. La calidad del modelo de aprendizaje automático va a ser directamente proporcional a la calidad de los datos. Por eso las labores de limpieza, depuración o *data wrangling*[6] consumen un porcentaje importante del tiempo de los científicos de datos.

5.11 Datos de prueba, validación o testing data

Los datos de prueba, validación o *testing data* son los datos que se reservan para comprobar si el modelo que se ha generado a partir de los datos de entrenamiento "funciona". Es decir, si las respuestas predichas por el modelo para un caso totalmente nuevo son acertadas o no.

Es importante que el conjunto de datos de prueba tenga un volumen suficiente como para generar resultados estadísticamente significativos, y a la vez, que sea representativo del conjunto de datos global.

Normalmente el conjunto de datos se suele repartir en un 70% de datos de entrenamiento y un 30% de datos de test, pero se puede variar la proporción según el caso. Lo importante es ser siempre conscientes de que hay que evitar el sobreajuste u *overfitting*.

5.12 Sobreajuste u overfitting

El sobreajuste ocurre cuando un modelo está sobreentrenado. Son modelos complejos que se ajustan tan milimétricamente al conjunto de datos a partir del cual se han creado, que pierden gran parte de su poder predictivo y ya no son útiles para otros conjuntos de datos. Esto se debe a que los datos siempre tienen cierto grado de error o imprecisión, e intentar ajustarse demasiado a ellos, complica el modelo inútilmente al mismo tiempo que le resta utilidad.

[6] Data wrangling a veces denominada "manipulación de datos", es el proceso de transformar y mapear datos de un formulario de datos "sin procesar" a otro formato con la intención de hacerlo más apropiado y valioso para una variedad de propósitos posteriores, como el análisis.

5.13 Subajuste o underfitting

El subajuste ocurre cuando el conjunto de datos de entrenamiento es insuficiente o poco representativo. Como consecuencia, nos lleva a un modelo excesivamente simple, con poco valor predictor.

Por ello, para generar un buen modelo de aprendizaje automático, es importante encontrar el punto medio entre sobreajuste y subajuste.

Otra forma de expresar este equilibrio es mediante los conceptos de Bias vs Varianza.

5.14 Aumento de datos

El aumento de datos es una de las técnicas utilizadas para aumentar la cantidad de datos agregando copias ligeramente modificadas de datos ya existentes o datos sintéticos recién creados a partir de datos existentes. Actúa como regularizador y ayuda a reducir el sobreajuste al entrenar un modelo de aprendizaje automático.

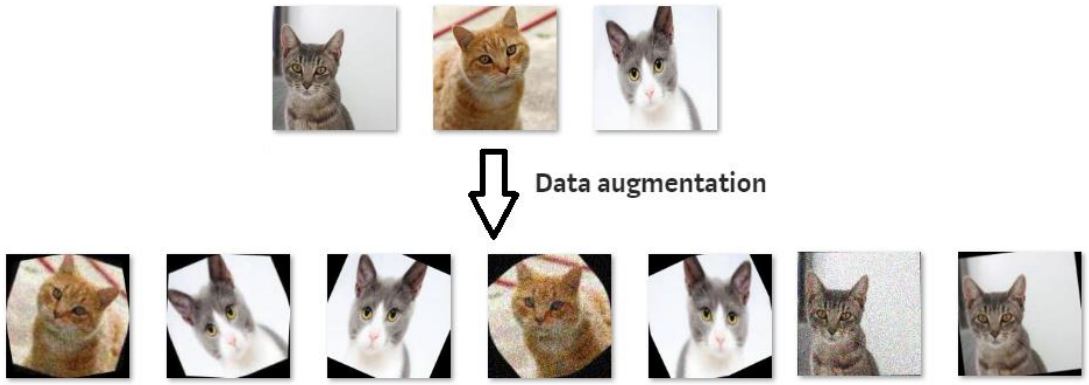


Figura 22. Ejemplo de aumento de datos.
*Fuente: https://miro.medium.com/max/1400/1*G7AOG4ivciLI6y1wYFx-sQ.png*

5.15 Dropout

Dropout es una técnica de regularización para reducir el sobreajuste en redes neuronales artificiales. Es una forma eficiente de realizar promedios de modelos con redes neuronales. El término dropout significa "abandonar" u omitir aleatoriamente neuronas (tanto ocultas como visibles) durante el proceso de entrenamiento de una

red neuronal. Tanto la reducción de los pesos como omitir unidades obtienen el mismo tipo de regularización. Al aplicar esta técnica de que las neuronas se “apaguen” durante el entrenamiento, dado que las mismas están siendo poco usadas, genera que la red sea resiliente.

5.16 Weight Decay

Weight Decay o *decaimiento de peso* es una técnica de regularización para reducir el sobreajuste en redes neuronales artificiales. Dado que existen datos que en el mundo real no serán simples, sino que serán bastante complejos, esta técnica permite agregar una pequeña penalización a la complejidad, generalmente la norma L2[7] de los pesos (todos los pesos del modelo), a la función de pérdida.

6. Herramientas utilizadas

En este apartado se explicarán las herramientas que fueron utilizadas a lo largo del desarrollo del trabajo.

6.1 LabelImg

LabelImg es una herramienta utilizada para el etiquetado del set de imágenes. Lo que se hace con este programa es abrir una imagen, seleccionar un recuadro para marcar el objeto que se quiere que el programa aprenda a detectar y guardar un archivo XML con la información de las coordenadas.

Figura 23. Logo labelImg.

Fuente: https://images4.programmerclick.com/902/f4/f4f587a488b19683c1c42537d30f5d46.png

[7] L1 y L2 son dos funciones de pérdida en el aprendizaje automático que se utilizan para minimizar el error. La función de pérdida L2 significa errores de mínimos cuadrados.

6.2 Python

Python es un lenguaje de programación interpretado cuya filosofía hace hincapié en la legibilidad de su código. Se trata de un lenguaje de programación multiparadigma, ya que soporta parcialmente la orientación a objetos, programación imperativa y, en menor medida, programación funcional. Es un lenguaje interpretado, dinámico y multiplataforma.

Python está basado en dos premisas: la filosofía *DRY* (Don't Repeat Yourself) y la *RAD* (Rapid Application Development). Gracias a esto, Python posee una gran variedad de bibliotecas[8], principal razón por la cual se ha convertido en el lenguaje de programación más popular para la Inteligencia Artificial (IA).

Las bibliotecas de Python proporcionan elementos de nivel básico para que los desarrolladores no tengan que codificarlos desde el principio cada vez. De esta manera, reducen su tiempo de aprendizaje de las complejidades del stack y así pueden comenzar con el desarrollo de Inteligencia Artificial y pasar a la creación de Algoritmos y programas de IA.

Figura 24. Logo Python.

Fuente: https://upload.wikimedia.org/wikipedia/commons/thumb/c/c3/Python-logo-notext.svg/1024px-Python-logo-notext.svg.png

6.3 Google colab

Colab es un servicio cloud, basado en los Notebooks de Jupyter, que permite el uso gratuito de las GPUs y TPUs de Google, con librerías como: Scikit-learn, PyTorch, TensorFlow, Keras y OpenCV. Todo ello con bajo Python 2.7 y 3.6, que aún no está disponible para R y Scala.

Es una herramienta ideal, no solo para practicar y mejorar los conocimientos en técnicas y herramientas de Data Science, sino también para el desarrollo de

[8] Una biblioteca es un módulo o un grupo de módulos publicados por diferentes fuentes que incluyen un fragmento de código preescrito que permite a los usuarios alcanzar alguna funcionalidad o realizar diferentes acciones.

aplicaciones (pilotos) de machine learning y deep learning, sin tener que invertir en recursos hardware o del Cloud.

Con Colab se pueden crear *notebooks*[9] o importar los que ya se tengan creados, además de compartirlos y exportarlos cuando se requiera. Esta fluidez a la hora de manejar la información también es aplicable a las fuentes de datos que se usen en los proyectos (notebooks), de modo que se pueda trabajar con información contenida en Google Drive, unidad de almacenamiento local, github e incluso en otros sistemas de almacenamiento cloud.

Figura 25. Logo Google colab.
Fuente: https://colab.research.google.com/img/colab_favicon_256px.png

6.4 Paquete y librerías

Algunos paquetes y librerías que se utilizan para Machine Learning son:

- Detecto: es un paquete de Python que le permite crear modelos de visión por computadora y detección de objetos completamente funcionales con solo 5 líneas de código. La inferencia en imágenes fijas y videos, la transferencia de aprendizaje en conjuntos de datos personalizados y la serialización de modelos en archivos son solo algunas de las funciones de Detecto. Detecto también se basa en *PyTorch*, lo que permite una fácil transferencia de modelos entre las dos bibliotecas.

- Pytorch: es una biblioteca de aprendizaje automático de código abierto basada en la biblioteca de Torch, utilizado para aplicaciones que implementan cosas como visión artificial y procesamiento de lenguajes naturales, principalmente desarrollado por el Laboratorio de Investigación de Inteligencia Artificial de Facebook (FAIR).

- Torch: es una biblioteca de código abierto para aprendizaje automático, un marco de computación científica y un lenguaje de script basado en el lenguaje de

[9] Notebooks son los entornos de trabajo donde se puede combinar código ejecutable y texto enriquecido junto con imágenes, HTML, etc.

programación Lua. Proporciona una amplia gama de algoritmos de aprendizaje profundo y usa el lenguaje de script LuaJIT sobre una implementación en C.

- Torchvision: es la librería principal de Facebook para aplicaciones de aprendizaje profundo. Sus elementos fundamentales son los tensores, que se pueden equiparar con vectores de una o varias dimensiones.

- Matplotlib: es una biblioteca para la generación de gráficos a partir de datos contenidos en listas o arrays en el lenguaje de programación Python y su extensión matemática NumPy.

- OS: Permite acceder a funcionalidades dependientes del sistema operativo. Sobre todo, aquellas que refieren información sobre el entorno del mismo y facilitando la manipulación de la estructura de directorios.

6.5 Hardware

El uso de técnicas de Machine Learning y en especial de Deep Learning como es el caso de las redes neuronales convolucionales implica un costo computacional muy elevado, debido a que están compuestas por un gran número de capas y neuronas y, además, trabajan con gran cantidad de imágenes que en muchos casos tienen una alta resolución.

En los ordenadores normales de hoy en día se pueden utilizar dos unidades de procesamiento para realizar cálculos: por un lado, se encuentran las unidades de procesamiento central (CPUs) y, por otro, las unidades de procesamiento gráfico (GPUs).

- CPUs: son los microprocesadores que se han utilizado siempre como unidad de procesamiento. Estos microprocesadores son óptimos haciendo tareas secuenciales, que no se pueden paralelizar. Hay distintas marcas, las más conocidas y que compiten por prácticamente todo el mercado son Intel y AMD.

- GPUs: son los procesadores gráficos que tienen las tarjetas gráficas. Se usan muchísimo para gaming, donde los videojuegos se benefician de la manera de procesar matrices que tienen las GPUs. Al fin y al cabo, las imágenes en HD no dejan de ser la unión de matrices de datos y entre las marcas más conocidas se encuentran Nvidia y AMD.

La diferencia más importante entre los dos componentes a nivel de arquitectura es el número de núcleos que suelen tener. Las CPUs tienen muchos menos núcleos que las GPUs, lo que hace que para procesamiento paralelo sean peores. Sin

embargo, los núcleos de las CPUs son más potentes, por lo que para tareas secuenciales son mejores.

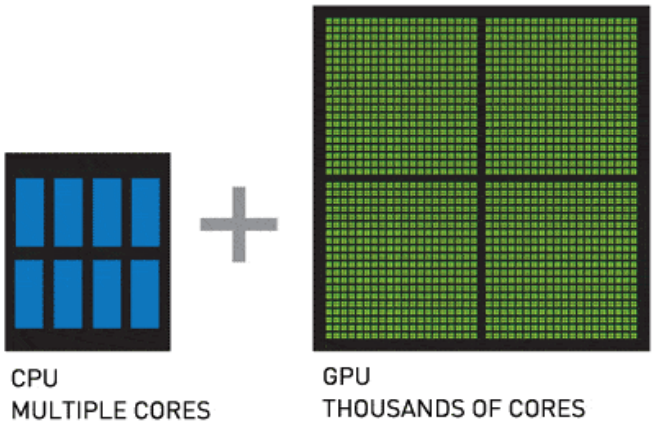


Figura 26. Comparativa de las arquitecturas de una CPU y de una GPU.
Fuente: https://images.nvidia.com/content/nvem/docs/IO/144271/cpu-and-gpu.jpg

Las tarjetas gráficas inicialmente tuvieron como propósito el de servir para procesar imágenes, pero se ha ido aprovechando su potencia para problemas de cálculo matricial. Inicialmente los desarrolladores que usaban la potencia de las GPUs para sus problemas matemáticos tenían que traducir sus problemas a triángulos y polígonos. Afortunadamente, este hecho cambió con la llegada de las arquitecturas *CUDA*[10], que permitían usar las GPUs para cálculo paralelo de propósito general. Actualmente, usando Python, es posible utilizar CUDA para calcular problemas matemáticos con GPUs.

7. Primera aproximación

Después de la investigación llevada a cabo, se realizó un primer algoritmo de detección de objetos utilizando un set de datos de 224 imágenes.

Como se comentó en el apartado 4.2 el objetivo es la detección de latas y botellas y dadas sus características poder detectar si dichos objetos se encuentran o no en buen estado. Por lo que, por cada objeto a identificar se asignó un nombre que representará su clase y son los siguientes: Lata, Botella, Lata_defectuosa,

[10] CUDA son las siglas de Compute Unified Device Architecture (Arquitectura Unificada de Dispositivos de Cómputo) que hace referencia a una plataforma de computación en paralelo incluyendo un compilador y un conjunto de herramientas de desarrollo creadas por Nvidia que permiten a los programadores usar una variación del lenguaje de programación C (CUDA C) para codificar algoritmos en GPU de Nvidia.

Botella_defectuosa con el fin del etiquetado de las imágenes. El etiquetado se realizó con la herramienta labelImg y por cada imagen se guardará un archivo XML. Este archivo tendrá las coordenadas de los objetos ubicados en la imagen y el nombre (name) que representa la clase de cada objeto ubicado en dicha imagen.

```
<annotation>
        <folder>imagenes</folder>
        <filename>0e106ca451dea2b98a07710a4dac5925.png</filename>
        <path>C:\Users\Ayelen\Desktop\imagenes\0e106ca451dea2b98a07710a4dac5925.png</path>
        <source>
                <database>Unknown</database>
        </source>
        <size>
                <width>470</width>
                <height>370</height>
                <depth>3</depth>
        </size>
        <segmented>0</segmented>
        <object>
                <name>Lata_defectuosa</name>
                <pose>Unspecified</pose>
                <truncated>1</truncated>
                <difficult>0</difficult>
                <bndbox>
                        <xmin>78</xmin>
                        <ymin>2</ymin>
                        <xmax>372</xmax>
                        <ymax>370</ymax>
                </bndbox>
        </object>
</annotation>
```

Figura 27. Archivo XML generado por labelImg.

Fuente: producción propia.

7.1 Transfer Learning en conjuntos de datos personalizados

Teniendo en cuenta la poca cantidad de imágenes recolectadas en una primera prueba piloto, se decidió utilizar el método de transfer learning. Partiendo de esto se seleccionó un modelo ya entrenado originalmente con el conjunto de datos *COCO*[11].

Se evaluó utilizar los siguientes modelos:

- Faster R-CNN ResNet50 FPN
- Faster R-CNN Mobilenetv3 Large FPN

Finalmente se eligió el modelo Faster R-CNN ResNet50 FPN. La arquitectura del modelo está dada por la siguiente figura:

[11] Este es un conjunto de datos de subtítulos, segmentación y detección de objetos a gran escala publicado por Microsoft.

```
FasterRCNN(
  (transform): GeneralizedRCNNTransform(
      Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
      Resize(min_size=(800,), max_size=1333, mode='bilinear')
  )
  (backbone): BackboneWithFPN(
    (body): IntermediateLayerGetter(
      (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
      (bn1): FrozenBatchNorm2d(64, eps=0.0)
      (relu): ReLU(inplace=True)
      (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
      (layer1): Sequential(
        (0): Bottleneck(
          (conv1): Conv2d(64, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
          (bn1): FrozenBatchNorm2d(64, eps=0.0)
          (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
          (bn2): FrozenBatchNorm2d(64, eps=0.0)
          (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
          (bn3): FrozenBatchNorm2d(256, eps=0.0)
          (relu): ReLU(inplace=True)
          (downsample): Sequential(
            (0): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
            (1): FrozenBatchNorm2d(256, eps=0.0)
          )
        )
    ,
    (1): Bottleneck(
      (conv1): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): FrozenBatchNorm2d(64, eps=0.0)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): FrozenBatchNorm2d(64, eps=0.0)
      (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): FrozenBatchNorm2d(256, eps=0.0)
      (relu): ReLU(inplace=True)
    )
    (2): Bottleneck(
      (conv1): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): FrozenBatchNorm2d(64, eps=0.0)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): FrozenBatchNorm2d(64, eps=0.0)
      (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): FrozenBatchNorm2d(256, eps=0.0)
      (relu): ReLU(inplace=True)
    )
  )

(layer2): Sequential(
  (0): Bottleneck(
    (conv1): Conv2d(256, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): FrozenBatchNorm2d(128, eps=0.0)
    (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
    (bn2): FrozenBatchNorm2d(128, eps=0.0)
    (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): FrozenBatchNorm2d(512, eps=0.0)
    (relu): ReLU(inplace=True)
    (downsample): Sequential(
      (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False)
      (1): FrozenBatchNorm2d(512, eps=0.0)
    )
  )
  (1): Bottleneck(
    (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): FrozenBatchNorm2d(128, eps=0.0)
    (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): FrozenBatchNorm2d(128, eps=0.0)
    (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): FrozenBatchNorm2d(512, eps=0.0)
    (relu): ReLU(inplace=True)
  )
```

```
      (3): Bottleneck(
        (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn1): FrozenBatchNorm2d(128, eps=0.0)
        (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn2): FrozenBatchNorm2d(128, eps=0.0)
        (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn3): FrozenBatchNorm2d(512, eps=0.0)
        (relu): ReLU(inplace=True)
      )
    )
    (layer3): Sequential(
      (0): Bottleneck(
        (conv1): Conv2d(512, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn1): FrozenBatchNorm2d(256, eps=0.0)
        (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
        (bn2): FrozenBatchNorm2d(256, eps=0.0)
        (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn3): FrozenBatchNorm2d(1024, eps=0.0)
        (relu): ReLU(inplace=True)
        (downsample): Sequential(
          (0): Conv2d(512, 1024, kernel_size=(1, 1), stride=(2, 2), bias=False)
          (1): FrozenBatchNorm2d(1024, eps=0.0)
        )
      )

      (1): Bottleneck(
        (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn1): FrozenBatchNorm2d(256, eps=0.0)
        (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn2): FrozenBatchNorm2d(256, eps=0.0)
        (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn3): FrozenBatchNorm2d(1024, eps=0.0)
        (relu): ReLU(inplace=True)
      )
      (2): Bottleneck(
        (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn1): FrozenBatchNorm2d(256, eps=0.0)
        (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn2): FrozenBatchNorm2d(256, eps=0.0)
        (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn3): FrozenBatchNorm2d(1024, eps=0.0)
        (relu): ReLU(inplace=True)
      )
      (3): Bottleneck(
        (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn1): FrozenBatchNorm2d(256, eps=0.0)
        (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn2): FrozenBatchNorm2d(256, eps=0.0)
        (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn3): FrozenBatchNorm2d(1024, eps=0.0)
        (relu): ReLU(inplace=True)
      )

      (4): Bottleneck(
        (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn1): FrozenBatchNorm2d(256, eps=0.0)
        (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn2): FrozenBatchNorm2d(256, eps=0.0)
        (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn3): FrozenBatchNorm2d(1024, eps=0.0)
        (relu): ReLU(inplace=True)
      )
      (5): Bottleneck(
        (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn1): FrozenBatchNorm2d(256, eps=0.0)
        (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn2): FrozenBatchNorm2d(256, eps=0.0)
        (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn3): FrozenBatchNorm2d(1024, eps=0.0)
        (relu): ReLU(inplace=True)
      )
    )
```

```
    (layer4): Sequential(
      (0): Bottleneck(
        (conv1): Conv2d(1024, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn1): FrozenBatchNorm2d(512, eps=0.0)
        (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
        (bn2): FrozenBatchNorm2d(512, eps=0.0)
        (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn3): FrozenBatchNorm2d(2048, eps=0.0)
        (relu): ReLU(inplace=True)
        (downsample): Sequential(
          (0): Conv2d(1024, 2048, kernel_size=(1, 1), stride=(2, 2), bias=False)
          (1): FrozenBatchNorm2d(2048, eps=0.0)
        )
      )
      (1): Bottleneck(
        (conv1): Conv2d(2048, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn1): FrozenBatchNorm2d(512, eps=0.0)
        (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn2): FrozenBatchNorm2d(512, eps=0.0)
        (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn3): FrozenBatchNorm2d(2048, eps=0.0)
        (relu): ReLU(inplace=True)
      )
      (2): Bottleneck(
        (conv1): Conv2d(2048, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn1): FrozenBatchNorm2d(512, eps=0.0)
        (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn2): FrozenBatchNorm2d(512, eps=0.0)
        (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn3): FrozenBatchNorm2d(2048, eps=0.0)
        (relu): ReLU(inplace=True)
      )
```

```
  (fpn): FeaturePyramidNetwork(
    (inner_blocks): ModuleList(
      (0): Conv2d(256, 256, kernel_size=(1, 1), stride=(1, 1))
      (1): Conv2d(512, 256, kernel_size=(1, 1), stride=(1, 1))
      (2): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1))
      (3): Conv2d(2048, 256, kernel_size=(1, 1), stride=(1, 1))
    )
    (layer_blocks): ModuleList(
      (0): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (3): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    )
    (extra_blocks): LastLevelMaxPool()
  )
)
(rpn): RegionProposalNetwork(
  (anchor_generator): AnchorGenerator()
  (head): RPNHead(
    (conv): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (cls_logits): Conv2d(256, 3, kernel_size=(1, 1), stride=(1, 1))
    (bbox_pred): Conv2d(256, 12, kernel_size=(1, 1), stride=(1, 1))
  )
)
(roi_heads): RoIHeads(
  (box_roi_pool): MultiScaleRoIAlign(featmap_names=['0', '1', '2', '3'], output_size=(7, 7), sampling_ratio=2)
  (box_head): TwoMLPHead(
    (fc6): Linear(in_features=12544, out_features=1024, bias=True)
    (fc7): Linear(in_features=1024, out_features=1024, bias=True)
  )
```

```
      (box_predictor): FastRCNNPredictor(
        (cls_score): Linear(in_features=1024, out_features=5, bias=True)
        (bbox_pred): Linear(in_features=1024, out_features=20, bias=True)
      )
    )
  )
```

Figura 28. Arquitectura de Faster R-CNN ResNet50 FPN.

Fuente: producción propia.

El procedimiento para la implementación del modelo de prueba consistió en aprovechar todo el entrenamiento de la etapa de extracción de características ya brindado por el modelo entrenado. Tomando la arquitectura del modelo entrenado se incorporó a un nuevo modelo y se adicionó el dataset de las 224 imágenes. Este se programó para realizar 30 épocas de entrenamiento y se utilizó el optimizador SGD con una tasa de aprendizaje de 0.005, Momentum 0,9, weight decay de 0.0005, gamma de 0,1 y lr step size de 3.

Se utilizó google colab para el desarrollo del modelo donde se usó google drive como directorio de almacén de las imágenes y sus XML correspondientes.

Se crearon tres directorios:

- images: contiene el total del set de imágenes etiquetadas, 224 imágenes y 224 archivos xml.

- training_data: contiene los datos de entrenamiento, almacena el 70 % del contenido de images.

- validation_data: contiene los datos de validación, almacena el 30% restante del contenido de images.

Como la idea es realizar muchas ejecuciones sobre los datos de entrenamiento y validación, es necesario generar un archivo CSV a partir de los datos XML para cada uno. Para eso se utilizó el paquete *detecto* que tiene definido tres módulos, entre ellos el módulo detecto.utils el cual posee la función que recibe un directorio que tiene los archivos XML y retorna un archivo CSV. Esta función se llama xml_to_csv y se define de la siguiente manera:

```
def xml_to_csv(xml_folder, output_file=None):
    xml_list = []
    image_id = 0
    # Recorre cada archivo xml
    for xml_file in glob(xml_folder + '/*.xml'):
        tree = ET.parse(xml_file)
        root = tree.getroot()

        filename = root.find('filename').text
        size = root.find('size')
        width = int(size.find('width').text)
        height = int(size.find('height').text)

        # Cada objeto representa cada etiqueta de imagen actual
        for member in root.findall('object'):
            box = member.find('bndbox')
            label = member.find('name').text

            # Agrega el nombre del archivo de imagen, el tamaño de la imagen, la etiqueta y las coordenadas del cuadro al archivo CSV
            row = (filename, width, height, label, int(float(box.find('xmin').text)),
                   int(float(box.find('ymin').text)), int(float(box.find('xmax').text)), int(float(box.find('ymax').text)), image_id)
            xml_list.append(row)

        image_id += 1

    # Guarda como un archivo CSV
    column_names = ['filename', 'width', 'height', 'class', 'xmin', 'ymin', 'xmax', 'ymax', 'image_id']
    xml_df = pd.DataFrame(xml_list, columns=column_names)

    if output_file is not None:
        xml_df.to_csv(output_file, index=None)

    return xml_df
```

Figura 29. Definición del método xml_to_csv.

Fuente: producción propia.

La siguiente figura muestra la conversión de los datos XML de entrenamiento y validación a CSV:

```
utils.xml_to_csv('training_data/', 'training.csv')
utils.xml_to_csv('validation_data/', 'validation.csv')
```

	filename	width	height	class	xmin	ymin	xmax	ymax	image_id
0	IMG_20211226_182351836.jpg	4160	3120	Lata	270	1031	3141	2226	0
1	IMG_20211225_115143833.jpg	4160	3120	Botella	731	369	1455	2573	1
2	IMG_20211225_115143833.jpg	4160	3120	Lata	1565	1160	2170	2560	1
3	IMG_20211225_115143833.jpg	4160	3120	Lata_defectuosa	2293	1140	2965	2602	1
4	IMG_20211225_115143833.jpg	4160	3120	Lata	3255	1126	3955	2607	1
5	IMG_20211225_115151765.jpg	4160	3120	Botella	308	135	1331	2488	2
6	IMG_20211225_115151765.jpg	4160	3120	Lata	1389	1097	2070	2535	2
7	IMG_20211225_115151765.jpg	4160	3120	Lata_defectuosa	2241	1154	2974	2631	2
8	IMG_20211225_115151765.jpg	4160	3120	Lata	3112	1183	4122	2669	2
9	IMG_20211225_115200128.jpg	4160	3120	Lata	112	612	1441	2331	3
10	IMG_20211225_115200128.jpg	4160	3120	Lata_defectuosa	1384	693	2431	2388	3
11	IMG_20211225_115200128.jpg	4160	3120	Lata	2727	669	4103	2407	3
12	IMG_20211225_115204417.jpg	4160	3120	Lata	151	1002	1493	2435	4
13	IMG_20211225_115204417.jpg	4160	3120	Lata_defectuosa	1484	1135	2474	2521	4
14	IMG_20211225_115204417.jpg	4160	3120	Lata	2698	1154	4093	2559	4
15	IMG_20211225_115215243.jpg	4160	3120	Lata	603	1631	3446	2959	5
16	IMG_20211225_115215243.jpg	4160	3120	Lata_defectuosa	831	64	3636	1421	5
17	IMG_20211225_120437569.jpg	4160	3120	Botella_defectuosa	222	664	4098	2145	6
18	IMG_20211225_120445550.jpg	4160	3120	Botella_defectuosa	174	678	4151	2735	7
19	IMG_20211226_182356165.jpg	4160	3120	Lata_defectuosa	298	983	3317	2597	8
20	IMG_20211226_182405948.jpg	4160	3120	Lata_defectuosa	465	1012	3755	2593	9

Figura 30. Conversión de XML a CSV.

Fuente: producción propia.

Adicionalmente, se utiliza la técnica de aumento de datos a los datos de entrenamiento. Para esto se emplea la librería torchvision y nuevamente el módulo detecto.utils. En la siguiente figura se convierte las imágenes en imágenes PIL[12], se aplican aumentos de cambio de tamaño, giro y saturación y finalmente se vuelve a convertir a tensores normalizados.

[12] PIL es la biblioteca de imágenes de Python que proporciona al intérprete de Python capacidades de edición de imágenes.

```python
from torchvision import transforms
from detecto.utils import normalize_transform

# lista de transformaciones para aplicar en las imágenes
transform_img = transforms.Compose([
    transforms.ToPILImage(),
    transforms.RandomHorizontalFlip(0.4),
    transforms.ColorJitter(saturation=0.3),
    transforms.ToTensor(),
    utils.normalize_transform(),
])
```

Figura 31. Transformaciones para aumento de datos.

Fuente: producción propia.

Luego, a partir de otro módulo de detecto llamado detecto.core se crea un objeto de conjunto de datos de entrenamiento y de validación a partir de las imágenes y etiqueta de las mismas. En dicho módulo ese objeto se define como la clase *DataSet*. Esta clase recibe una carpeta con las imágenes y etiquetas (archivos XML) o las etiquetas convertidas a CSV, la carpeta con las imágenes y las transformaciones a aplicar (opcional).

```python
class Dataset(torch.utils.data.Dataset):

    def __init__(self, label_data, image_folder=None, transform=None):
        # El archivo CSV contiene: nombre de archivo, ancho, alto, clase, xmin, ymin, xmax, ymax
        if os.path.isfile(label_data):
            self._csv = pd.read_csv(label_data)
        else:
            self._csv = xml_to_csv(label_data)

        # Si no se proporciona la carpeta de imágenes, se configura en la carpeta de etiquetas
        if image_folder is None:
            self._root_dir = label_data
        else:
            self._root_dir = image_folder

        if transform is None:
            self.transform = default_transforms()
        else:
            self.transform = transform

    # Devuelve la longitud de este conjunto de datos.
    def __len__(self):
        # número de entradas == número de image_id únicos en csv.
        return len(self._csv['image_id'].unique().tolist())


    # Es lo que le permite indexar el conjunto de datos, p. dataset[0]
    # dataset[index] devuelve una tupla que contiene la imagen y los targets
    def __getitem__(self, idx):
        if torch.is_tensor(idx):
            idx = idx.tolist()

        # Lee en la imagen del nombre del archivo en la columna 0
        object_entries = self._csv.loc[self._csv['image_id'] == idx]

        img_name = os.path.join(self._root_dir, object_entries.iloc[0, 0])
        image = read_image(img_name)

        boxes = []
        labels = []
        for object_idx, row in object_entries.iterrows():
            # Lee en xmin, ymin, xmax, and ymax
            box = self._csv.iloc[object_idx, 4:8]
            boxes.append(box)
            # Lee en la etiqueta
            label = self._csv.iloc[object_idx, 3]
            labels.append(label)

        boxes = torch.tensor(boxes).view(-1, 4)

        targets = {'boxes': boxes, 'labels': labels}
```

```
        # Realiza transformaciones
        if self.transform:
            width = object_entries.iloc[0, 1]
            height = object_entries.iloc[0, 2]

            # Aplica las transformaciones manualmente para poder tratar con ellas
            # transformaciones como Resize o RandomHorizontalFlip
            updated_transforms = []
            scale_factor = 1.0
            random_flip = 0.0
            for t in self.transform.transforms:
                # Agrega cada transformación a la lista
                updated_transforms.append(t)

                # Si existe una transformación de cambio de tamaño, reduce las coordenadas
                # de la caja por la misma cantidad que el cambio de tamaño
                if isinstance(t, transforms.Resize):
                    original_size = min(height, width)
                    scale_factor = original_size / t.size

                # Si existe una transformación de giro horizontal, obtengo su probabilidad
                # para que podamos aplicarlo manualmente tanto a la imagen como a las cajas.
                elif isinstance(t, transforms.RandomHorizontalFlip):
                    random_flip = t.p


            # Aplica cada transformación manualmente
             for t in updated_transforms:
                 # Maneja el caso de giro horizontal, donde necesitamos aplicar
                 # la transformación tanto de la imagen como de las etiquetas de la caja
                 if isinstance(t, transforms.RandomHorizontalFlip):
                     if random.random() < random_flip:
                         image = transforms.RandomHorizontalFlip(1)(image)
                         for idx, box in enumerate(targets['boxes']):
                             # Flip box's x-coordinates
                             box[0] = width - box[0]
                             box[2] = width - box[2]
                             box[[0,2]] = box[[2,0]]
                             targets['boxes'][idx] = box
                 else:
                     image = t(image)

            # Reduce el cuadro si es necesario
             if scale_factor != 1.0:
                 for idx, box in enumerate(targets['boxes']):
                     box = (box / scale_factor).long()
                     targets['boxes'][idx] = box

         return image, targets
```

Figura 32. Clase DataSet.

Fuente: producción propia

La siguiente figura muestra la creación de DataSet para los datos de entrenamiento pasándole las transformaciones establecidas y el DataSet de los datos de validación:

```
#Conjunto de datos de entrenamiento
dataset = core.Dataset('training.csv', 'training_data/', transform=transform_img)

#Conjunto de datos de validación
val_dataset = core.Dataset('validation.csv', 'validation_data/')
```

Figura 33. Creación de DataSet para entrenamiento y validación.

Fuente: producción propia.

Luego se tiene que crear y entrenar un modelo del conjunto de datos. Primero, se especifica qué clases se desea predecir al inicializar el modelo. Después de eso, se crea un *DataLoader* sobre el conjunto de datos para ayudar a definir cómo se agrupan y se alimentan las imágenes en el modelo para el entrenamiento.

Por consiguiente, para crear el modelo y el DataLoader se utiliza el módulo detecto.core donde se define la clase Modelo y DataLoader.

```
class Model:

    DEFAULT = 'fasterrcnn_resnet50_fpn'
    MOBILENET = 'fasterrcnn_mobilenet_v3_large_fpn'
    MOBILENET_320 = 'fasterrcnn_mobilenet_v3_large_320_fpn'

    def __init__(self, classes=None, device=None, pretrained=True, model_name=DEFAULT):
        self._device = device if device else config['default_device']

        # Carga un modelo previamente entrenado en COCO
        if model_name == self.DEFAULT:
            self._model = torchvision.models.detection.fasterrcnn_resnet50_fpn(pretrained=pretrained)
        elif model_name == self.MOBILENET:
            self._model = torchvision.models.detection.fasterrcnn_mobilenet_v3_large_fpn(pretrained=pretrained)
        elif model_name == self.MOBILENET_320:
            self._model = torchvision.models.detection.fasterrcnn_mobilenet_v3_large_320_fpn(pretrained=pretrained)
        else:
            raise ValueError(f'Invalid value {model_name} for model_name. ' +
                             f'Please choose between {self.DEFAULT}, {self.MOBILENET}, and {self.MOBILENET_320}.')

        if classes:
            # Obtiene el número de características de entrada para el clasificador
            in_features = self._model.roi_heads.box_predictor.cls_score.in_features
            # Reemplaza la pre-trained head  por una nueva (nota: +1 debido a la __background__ class)
            self._model.roi_heads.box_predictor = FastRCNNPredictor(in_features, len(classes) + 1)
            self._disable_normalize = False
        else:
            classes = config['default_classes']
            self._disable_normalize = True

        self._model.to(self._device)

        # Asignaciones para convertir etiquetas de cadenas a enteros y viceversa
        self._classes = ['__background__'] + classes
        self._int_mapping = {label: index for index, label in enumerate(self._classes)}

# Devuelve las predicciones sin procesar al introducir una imagen o una lista de imágenes en el modelo
def _get_raw_predictions(self, images):
    self._model.eval()

    with torch.no_grad():
        # Convierte la imagen en una lista de longitud 1 si aún no es una lista
        if not _is_iterable(images):
            images = [images]
        # Convierte a tensor y normaliza si aún no lo ha hecho
        if not isinstance(images[0], torch.Tensor):

            # Esta es una solución temporal a la mala precisión
            # al normalizar en pesos predeterminados.
            if self._disable_normalize:
                defaults = transforms.Compose([transforms.ToTensor()])
            else:
                defaults = default_transforms()
            images = [defaults(img) for img in images]

        # Envia imágenes al dispositivo especificado
        images = [img.to(self._device) for img in images]

        preds = self._model(images)
        # Envia predicciones a la CPU si aún no lo ha hecho
        preds = [{k: v.to(torch.device('cpu')) for k, v in p.items()} for p in preds]
        return preds
```

Figura 34. Clase Model.

Fuente: producción propia.

```python
class DataLoader(torch.utils.data.DataLoader):

    def __init__(self, dataset, **kwargs):
        super().__init__(dataset, collate_fn=DataLoader.collate_data, **kwargs)


    # Convierte una lista de tuplas en una tupla de listas para que
    # se puede alimentar correctamente al modelo para el entrenamiento
    @staticmethod
    def collate_data(batch):
        images, targets = zip(*batch)
        return list(images), list(targets)
```

Figura 35. Clase DataLoader.

Fuente: producción propia.

La clase Model va a recibir el nombre de las clases a predecir, el dispositivo en el que se va a ejecutar el modelo (opcional), como CPU o GPU, si el modelo está o no pre entrenado y el nombre del modelo (opcional). Mientras que la clase DataLoader recibe el objeto DataSet creado a partir de los datos de entrenamiento y parámetros como batch size y *shuffle* para mezclar los datos después de cada época de entrenamiento.

La implementación de dichas clases queda representada por la siguiente figura:

```python
# Modelo con todas las clases únicas que se van a predicir
model = core.Model(['Lata_defectuosa', 'Botella_defectuosa','Lata', 'Botella'])

# El loader para el conjunto de datos de entrenamiento
loader = core.DataLoader(dataset, batch_size=5, shuffle=True)
```

Figura 36. Implementación de Model y DataLoader.

Fuente: producción propia.

Finalmente, para entrenar el modelo se utiliza el método *fit* definido en la clase Model.

```python
def fit(self, dataset, val_dataset=None, epochs=10, learning_rate=0.005, momentum=0.9,
        weight_decay=0.0005, gamma=0.1, lr_step_size=3, verbose=True):

    if verbose and self._device == torch.device('cpu'):
        print('It looks like you\'re training your model on a CPU. '
              'Consider switching to a GPU; otherwise, this method '
              'can take hours upon hours or even days to finish. '
              'For more information, see https://detecto.readthedocs.io/'
              'en/latest/usage/quickstart.html#technical-requirements')
    # Si realiza un entrenamiento personalizado, lo más probable es que las imágenes dadas sean
    # normalizadas. Esto debería solucionar el problema del bajo rendimiento en
    # clases predeterminadas al normalizar, así que reanude la normalización.
    if epochs > 0:
        self._disable_normalize = False


    # Convierte el conjunto de datos en un cargador de datos si aún no lo ha hecho
    if not isinstance(dataset, DataLoader):
        dataset = DataLoader(dataset, shuffle=True)

    if val_dataset is not None and not isinstance(val_dataset, DataLoader):
        val_dataset = DataLoader(val_dataset)

    losses = []
    # Obtiene parámetros que tienen graduación activada (es decir, parámetros que deben ser entrenados)
    parameters = [p for p in self._model.parameters() if p.requires_grad]
    # Crea un optimizador que usa SGD (descenso de gradiente estocástico) para entrenar los parámetros
    optimizer = torch.optim.SGD(parameters, lr=learning_rate, momentum=momentum, weight_decay=weight_decay)

    # Crea un programador de learning rate que disminuye el learning rate debido a gamma cada lr_step_size epocas
    lr_scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=lr_step_size, gamma=gamma)

# Entrena en todo el conjunto de datos durante la cantidad especificada de veces (epochs)
for epoch in range(epochs):
    if verbose:
        print('Epoch {} of {}'.format(epoch + 1, epochs))

    # Training step
    self._model.train()

    if verbose:
        print('Begin iterating over training dataset')

    iterable = tqdm(dataset, position=0, leave=True) if verbose else dataset
    for images, targets in iterable:
        self._convert_to_int_labels(targets)
        images, targets = self._to_device(images, targets)

        # Calcula la pérdida del modelo (es decir, qué tan bien lo hace en el actual
        # imagen y objetivo, siendo mejor una menor pérdida)
        loss_dict = self._model(images, targets)
        total_loss = sum(loss for loss in loss_dict.values())
        # Cero cualquier gradiente antiguo/existente en los parámetros del modelo
        optimizer.zero_grad()

        # Calcula gradientes para cada parámetro en función del cálculo de pérdida actual
        total_loss.backward()

        # Actualiza los parámetros del modelo desde los gradientes: param -= learning_rate * param.grad
        optimizer.step()

    # Paso de validación
    if val_dataset is not None:
        avg_loss = 0
        with torch.no_grad():
            if verbose:
                print('Begin iterating over validation dataset')

            iterable = tqdm(val_dataset, position=0, leave=True) if verbose else val_dataset
            for images, targets in iterable:
                self._convert_to_int_labels(targets)
                images, targets = self._to_device(images, targets)
                loss_dict = self._model(images, targets)
                total_loss = sum(loss for loss in loss_dict.values())
                avg_loss += total_loss.item()
        # Actualiza la tasa de aprendizaje cada pocas épocas
        lr_scheduler.step()

    if len(losses) > 0:
        return losses
```

Figura 37. Definición del método fit.

Fuente: producción propia

El método fit recibe el DataLoader del entrenamiento, el DataSet de validación, epochs, learning_rate, momentum, weight_decay, gamma, lr_step_size y *verbose*. Este último sirve para imprimir la época actual, el progreso y pérdida (si se brinda

un DataSet de validación) en cada paso, junto con algunas advertencias adicionales si usa una CPU.

La implementación de dicho método se muestra en la figura siguiente:

```
# Entrenando modelo
losses = model.fit(
    loader,
    val_dataset,
    epochs=30,
    verbose=True)
```

```
It looks like you're training your model on a CPU. Consider switching to a GPU; otherwise, this method can take hours upon hours or even d
Epoch 1 of 30
Begin iterating over training dataset
  0%|          | 0/7 [00:00<?, ?it/s]/usr/local/lib/python3.7/dist-packages/torch/functional.py:445: UserWarning: torch.meshgrid: in an up
  return _VF.meshgrid(tensors, **kwargs)  # type: ignore[attr-defined]
100%|██████████| 7/7 [15:58<00:00, 136.94s/it]
Begin iterating over validation dataset
100%|██████████| 10/10 [01:16<00:00,  7.69s/it]
Loss: 0.3653318338096142
Epoch 2 of 30
Begin iterating over training dataset
100%|██████████| 7/7 [16:19<00:00, 139.98s/it]
Begin iterating over validation dataset
100%|██████████| 10/10 [01:06<00:00,  6.68s/it]
Loss: 0.374638681858778
Epoch 3 of 30
Begin iterating over training dataset
100%|██████████| 7/7 [19:04<00:00, 163.45s/it]
Begin iterating over validation dataset
100%|██████████| 10/10 [01:06<00:00,  6.65s/it]
Loss: 0.34658622816205026
Epoch 4 of 30
Begin iterating over training dataset
100%|██████████| 7/7 [16:43<00:00, 143.38s/it]
Begin iterating over validation dataset
100%|██████████| 10/10 [01:08<00:00,  6.87s/it]
Loss: 0.3576229393482208
```

Figura 38. Implementación del método fit.

Fuente: producción propia.

Una vez finalizado el entrenamiento, por medio de la biblioteca matplotlib, es posible imprimir un gráfico para observar el comportamiento de la precisión a lo largo del tiempo.

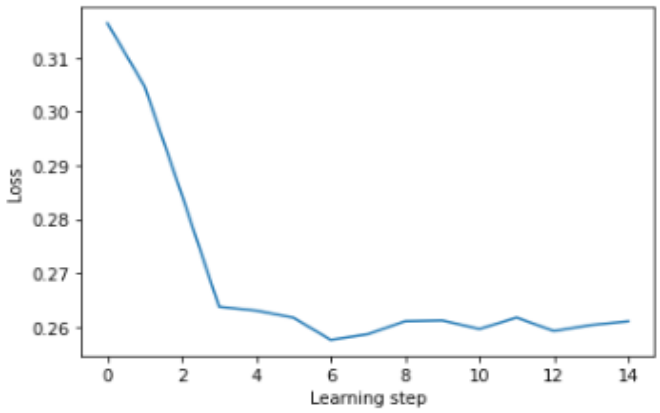


Figura 39. Precisión durante el entrenamiento.

Fuente: producción propia.

7.2 Resultados obtenidos

Para evaluar el modelo entrenado se utilizó otro módulo de detecto llamado detecto.visualize para poder detectar objetos en vivo a través de una cámara web.

En dicho módulo está definida la función *detect*_live, cuya implementación se muestra en la siguiente figura:

```python
def detect_live(model, score_filter=0.6):
    cv2.namedWindow('Detecto')
    try:
        video = cv2.VideoCapture(0)
    except:
        print('No webcam available.')
        return

    while True:
        ret, frame = video.read()
        if not ret:
            break

        labels, boxes, scores = model.predict(frame)

        # Grafica cada caja con su etiqueta y puntuacion
        for i in range(boxes.shape[0]):
            if scores[i] < score_filter:
                continue
            box = boxes[i]
            cv2.rectangle(frame, (int(box[0]), int(box[1])), (int(box[2]), int(box[3])), (255, 0, 0), 3)
            if labels:
                cv2.putText(frame, '{}: {}'.format(labels[i], round(scores[i].item(), 2)), (int(box[0]), int(box[1] - 10)),
                            cv2.FONT_HERSHEY_SIMPLEX, 1, (255, 0, 0), 3)

        cv2.imshow('Detecto', frame)


        # Si se presiona la tecla 'q' o ESC, sale del bucle
        key = cv2.waitKey(1) & 0xFF
        if key == ord('q') or key == 27:
            break

    cv2.destroyWindow('Detecto')
    video.release()
```

Figura 40. Definición del método detect_live

Fuente: producción propia.

Este método muestra en una ventana las predicciones del modelo en la actual transmisión de cámara web en vivo de una computadora.

Los resultados obtenidos fueron los siguientes:

Figura 41. Detección de lata defectuosa.

Fuente: producción propia.

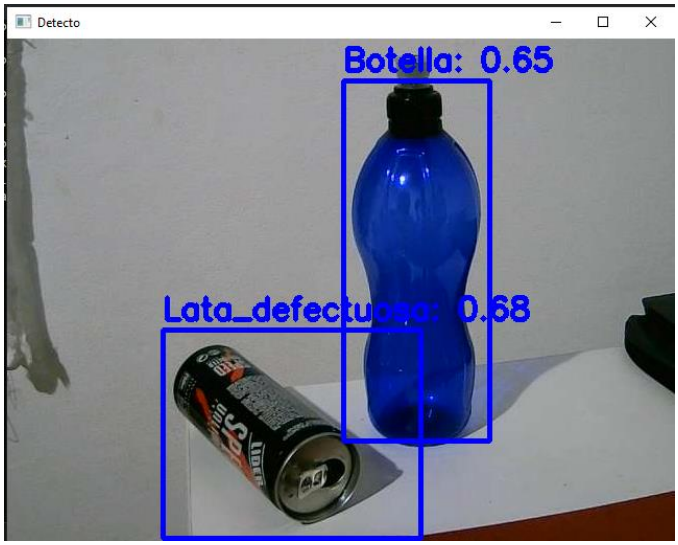


Figura 42. Detección de lata defectuosa y botella.

Fuente: producción propia.

Además de lo anterior se realizó un algoritmo para identificar objetos a partir de imágenes, los resultados fueron los siguientes:

Figura 43. Detección de objetos.

Fuente: producción propia.

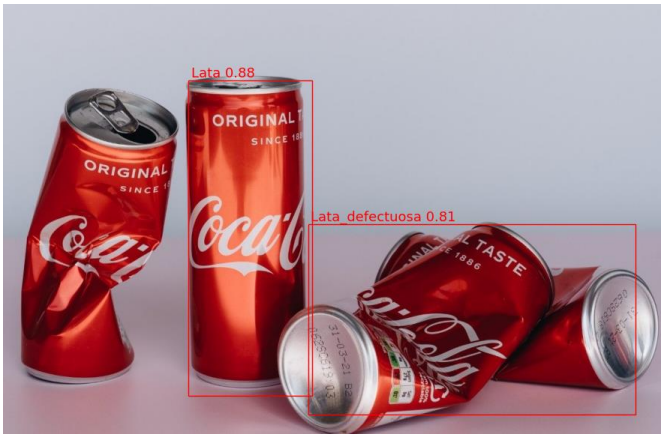


Figura 44. Detección de objetos.
Fuente: producción propia.

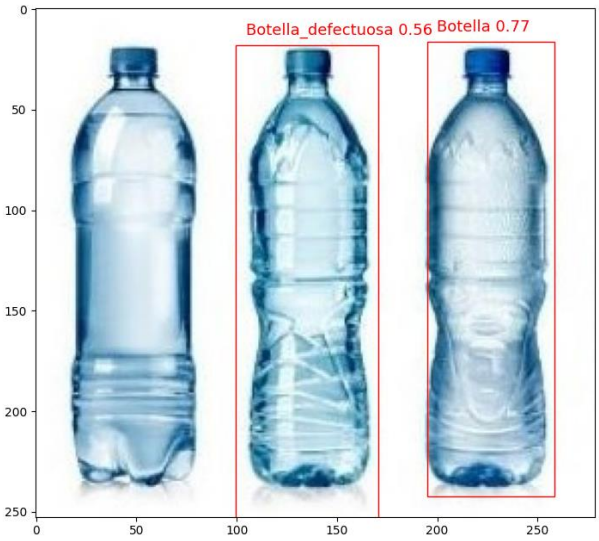


Figura 45. Detección de objetos.
Fuente: producción propia.

De lo obtenido en la figura 39, a medida que se va entrenando va disminuyendo la pérdida del mismo, observándose una mediana estabilidad, lo cual podría mejorarse estableciendo más tiempo de entrenamiento.

Con respecto a las detecciones realizadas en las figuras 41 y 42, estas son correctas. Por otra parte, en las figuras 43 y 45 ocurren detecciones erróneas.

Los resultados de las detecciones pueden deberse a la insuficiente cantidad de imágenes utilizadas para el entrenamiento y validación del modelo y al bajo número de épocas de entrenamiento.

7.3 Problemas detectados

La utilización del paquete *detecto* si bien resultó ser muy útil, contó con varias limitantes:

✓ Insuficientes métricas: si bien cuando culmina el entrenamiento se retorna la pérdida durante el mismo para poder graficar, este solamente representa el promedio de pérdida acumulada durante la iteración de la validación.

✓ Error al redimensionar: al intentar cambiar las dimensiones de las imágenes ocurrió un error que resultaba en cuadros delimitadores de los objetos con coordenadas negativas, por lo que se tuvo que entrenar el modelo con los tamaños originales debido a que el etiquetado de imágenes fue lo primero que había realizado.

El último problema ocasionó, que debido a la falta de memoria disponible y dado el gran tamaño de la mayoría de las imágenes, no se pudiera aumentar el número de épocas de entrenamiento. Además, provocó que se vuelva lenta la transmisión en vivo de la cámara web.

7.4 Mejoras realizadas

Para poder solventar los problemas anteriormente mencionados y adicionar mejoras con el objetivo de obtener métricas se realizó lo siguiente:

✓ Se utilizó el código fuente de las clases utilizadas en el apartado 7 y se le realizaron los ajustes necesarios para poder generar un gráfico cada 2 épocas acerca de la pérdida de entrenamiento y validación por cada iteración.

✓ Se resolvió el error que no permitía hacer una redimensión de las imágenes y los correspondientes cuadros delimitadores etiquetados.

✓ Otro ajuste realizado fue customizar una clase dataset para poder guardar el nombre de la imagen, con el objetivo de una vez finalizado el entrenamiento y la validación, levantar el modelo generado y poder evaluar el rendimiento del mismo.

8. Entrenamiento de modelo ResNet50

Una vez agregadas las mejoras del código fuente, se realizó un nuevo entrenamiento, esta vez con 100 épocas, batch de 32 y se utilizó el optimizador SGD con una tasa de aprendizaje de 0,001, Momentum 0,9, weight decay de 0.0005. Estas configuraciones fueron elegidas debido a que, en la primera aproximación realizada, faltaron más iteraciones en el entrenamiento para lograr un mayor porcentaje en la precisión de las detecciones. También, se utilizó la técnica de aumento de datos a los datos, aplicando giro, rotación y desenfoque. Además, se mantuvo la cantidad de imágenes recolectadas (224) y se aplicó una redimensión de 448 x 448 del tamaño a las mismas y de cada bounding box etiquetado.

Otro punto a mencionar, es que no se utilizó google colab, se usó una notebook que, a pesar de no contar con GPU, cuenta con una memoria RAM de 16.0 GB.

Por último, el entrenamiento se limitó el ajuste fino durante el entrenamiento solo a las últimas capas del modelo pre entrenado Faster R-CNN ResNet50 FPN.

Durante el entrenamiento se fue generando un gráfico de la pérdida que se producía en el entrenamiento y en la validación cada dos épocas.

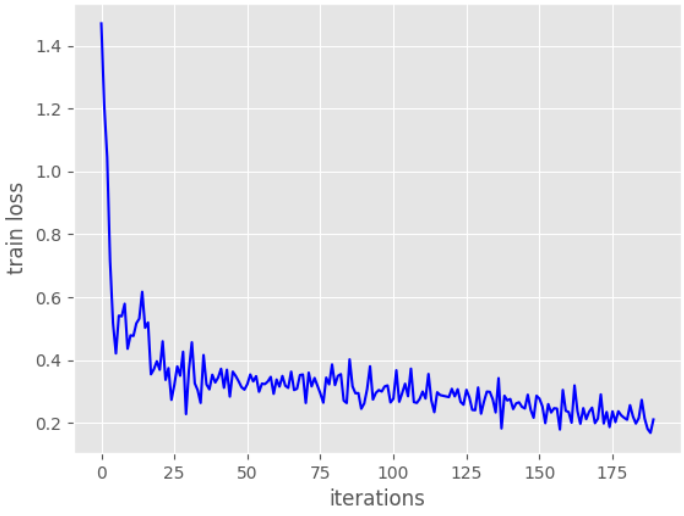


Figura 46. Pérdida durante el entrenamiento en la época 38.

Fuente: producción propia

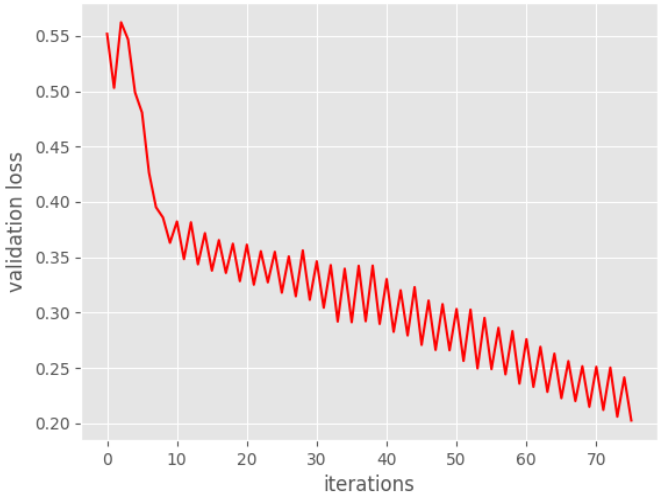


Figura 47. Pérdida durante la validación en la época 38.

Fuente: producción propia.

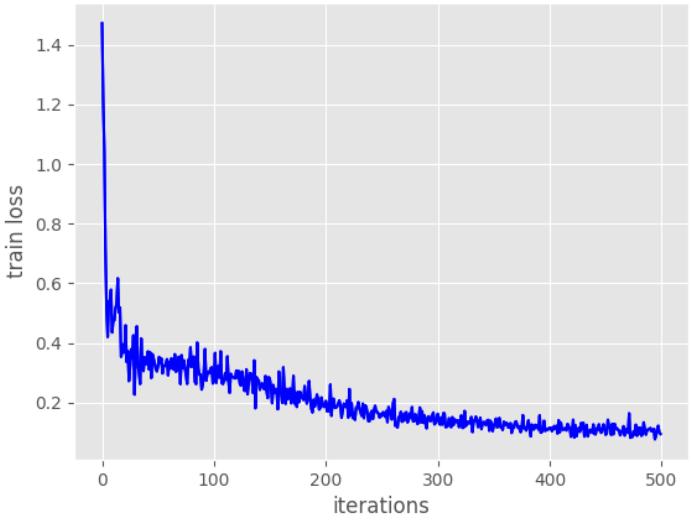


Figura 48. Pérdida durante el entrenamiento en la época 100.

Fuente: producción propia.

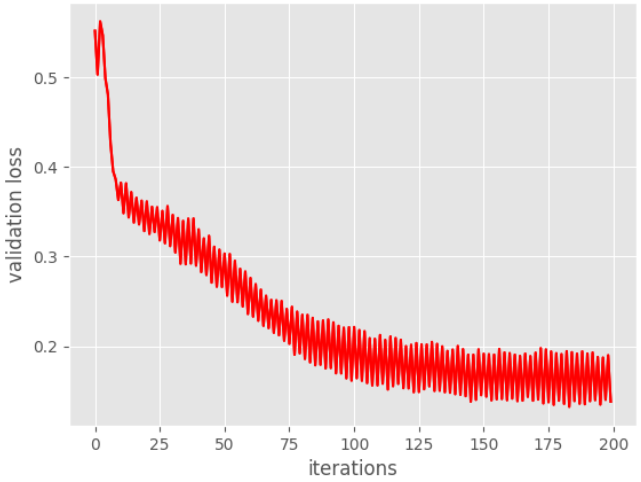


Figura 49. Pérdida durante la validación en la época 100.
Fuente: producción propia.

A partir de los gráficos anteriores, se puede decir que la pérdida de entrenamiento se estabilizó después de la iteración 400. Por otra parte, la pérdida de validación disminuyó hasta el final del entrenamiento. Esto significa que el modelo todavía estaba aprendiendo de manera que, entrenar durante más tiempo, hubiera ayudado.

9. Evaluación de rendimiento

Para saber cómo evaluar el rendimiento de un modelo de detección de objetos, primero se debe comprender cuáles son sus objetivos:

✓ Clasificar: se identifica si un objeto está presente en la imagen y la clase del objeto.

✓ Localizar: se predicen las coordenadas del cuadro delimitador alrededor del objeto cuando un objeto está presente en la imagen. En esta parte se comparan las *ground-truth* y los bounding box predichos.

Por lo que, para evaluar un modelo de detección de objetos, es necesario evaluar el rendimiento tanto de la clasificación como de la localización del uso de cuadros delimitadores en la imagen. Para hacerlo se utiliza el concepto de Intersección sobre Unión (IoU) mencionado en el apartado 5.6.2.2. Tal como allí se mencionó, se puede establecer un valor de umbral para que IoU determine si la detección de objetos es válida o no. Si se establece, por ejemplo, IoU en 0.5:

✓ Si IoU ≥0.5, se clasifica la detección de objetos como verdadero positivo (TP).

✓ Si Iou <0.5, es una detección incorrecta y se clasifica como falso positivo (FP).

✓ Cuando un *ground-truth* está presente en la imagen y el modelo no puede detectar el objeto, se clasifica como falso negativo (FN).

✓ La clasificación como verdadero negativo (TN) es cada parte de la imagen donde no se predice un objeto. Esta métrica no es útil para la detección de objetos, por lo que se ignora este TN.

Se utiliza *Precision* y *Recall* como métricas para evaluar el rendimiento. La precisión y la recuperación se calculan utilizando verdaderos positivos (TP), falsos positivos (FP) y falsos negativos (FN).

$$\text{Precision} = \frac{\text{True Positive}}{\text{True Positive + False Positive}}$$

$$\text{Recall} = \frac{\text{True Positive}}{\text{True Positive + False Negative}}$$

Figura 50. Precisión y Recall.

*Fuente: https://miro.medium.com/max/403/1*pt50d5q2KyrjA6BQhdNPHg.png*

También se tiene en cuenta la puntuación o *score* de cada objeto detectado por el modelo en la imagen. Se consideran todos los cuadros delimitadores predichos con un score por encima de cierto umbral. Los cuadros delimitadores por encima del valor de umbral se consideran cuadros positivos y todos los cuadros delimitadores predichos por debajo del valor de umbral se consideran negativos.

9.1 Precisión promedio o mAP (mean Average Precision)

AP (Precisión promedio) es una métrica muy utilizada para medir la precisión de los detectores de objetos como Faster R-CNN, SSD, etc. Para calcular el valor se puede utilizar una técnica llamada precisión promedio interpolada de 11 puntos.

El primer paso es graficar la precisión y recall y para eso, se muestran los valores de precisión y recall en un gráfico de recuperación de precisión. Cabe resaltar que dicho gráfico puede disminuir monótonamente por lo que siempre hay una compensación entre precisión y recuperación (recall), dado que al aumentar uno

disminuirá el otro. Algunas veces, el gráfico no siempre disminuye monótonamente debido a ciertas excepciones y/o falta de datos.

El segundo paso es calcular la precisión promedio (mAP), usando la técnica de interpolación de 11 puntos. La precisión interpolada es la precisión promedio medida en 11 niveles de recuperación (recall) igualmente espaciados de 0.0, 0.1, 0.2, 0.3…0.9, 1.0. De esta manera, para resolver el problema de la ausencia de disminución monótona de la recuperación de precisión, se establece un máximo de precisión para un único nivel de recuperación. Por lo que, a cada nivel de recuperación, se asigna el máximo valor de precisión con el objetivo de que, tanto la precisión como la recuperación, mejoren. Por ejemplo, si inicialmente para el valor de recall 0.20 se tiene las precisiones 0.50 y 1.0, para la evaluación de interpolación para el nivel de recall 0.20 se corresponderá la precisión 1.0.

Luego de realizar la asignación de precisión máxima para cada nivel de recall, se puede graficar la recuperación de precisión y la precisión interpolada. Finalmente se aplica la fórmula de precisión promedio de los valores obtenidos.

$$AP = \frac{1}{11} \sum_{r \in \{0, 0.1, \ldots, 1\}} \rho_{\text{interp}}(r)$$

Figura 51. Fórmula mAP.

*Fuente: https://miro.medium.com/max/348/1*Fq-d46tDerMSRJ8wFoKoBQ.png*

10. Rendimiento del modelo

Para aplicar la técnica de interpolación de 11 puntos y obtener la precisión promedio del modelo se desarrolló un algoritmo en python. A continuación, se realiza una muestra y explicación de los métodos realizados.

```
def calc_iou( gt_bbox, pred_bbox):
    '''
    This function takes the predicted bounding box and ground truth bounding box and
    return the IoU ratio
    '''
    x_topleft_gt, y_topleft_gt, x_bottomright_gt, y_bottomright_gt= gt_bbox
    x_topleft_p, y_topleft_p, x_bottomright_p, y_bottomright_p= pred_bbox

    if (x_topleft_gt > x_bottomright_gt) or (y_topleft_gt> y_bottomright_gt):
        raise AssertionError("Ground Truth Bounding Box is not correct")
    if (x_topleft_p > x_bottomright_p) or (y_topleft_p> y_bottomright_p):
        raise AssertionError("Predicted Bounding Box is not correct",x_topleft_p, x_bottomright_p,y_topleft_p,y_bottomright_gt)
```

```
    x_top_left =np.max([x_topleft_gt, x_topleft_p])
    y_top_left = np.max([y_topleft_gt, y_topleft_p])
    x_bottom_right = np.min([x_bottomright_gt, x_bottomright_p])
    y_bottom_right = np.min([y_bottomright_gt, y_bottomright_p])

    intersection_area = (x_bottom_right- x_top_left + 1) * (y_bottom_right-y_top_left  + 1)

    union_area = (GT_bbox_area + Pred_bbox_area - intersection_area)

    return intersection_area/union_area
```

Figura 52. Método calc_iou.

Fuente: Producción propia.

El método mencionado de la figura anterior toma el cuadro delimitador predicho y el cuadro delimitador real y devuelve la relación IoU.

```
def calc_precision_recall(image_results):
    """Calculates precision and recall from the set of images
    Args:
        img_results (dict): dictionary formatted like:
            {
                'img_id1': {'true_pos': int, 'false_pos': int, 'false_neg': int},
                'img_id2': ...
                ...
            }
    Returns:
        tuple: of floats of (precision, recall)
    """
    true_positive=0
    false_positive=0
    false_negative=0
    for img_id, res in image_results.items():
        true_positive +=res['true_positive']
        false_positive += res['false_positive']
        false_negative += res['false_negative']
        try:
            precision = true_positive/(true_positive+ false_positive)
        except ZeroDivisionError:
            precision=0.0
        try:
            recall = true_positive/(true_positive + false_negative)
        except ZeroDivisionError:
            recall=0.0
    return (precision, recall)
```

Figura 53. Método calc_precision_recall.

Fuente: Producción propia.

El método mencionado en la figura anterior calcula la precisión y recall de una lista de imágenes.

```
def get_single_image_results(gt_boxes, pred_boxes, iou_thr):

    all_pred_indices= range(len(pred_boxes))
    all_gt_indices=range(len(gt_boxes))
    if len(all_pred_indices)==0:
        tp=0
        fp=0
        fn=len(gt_boxes)
        return {'true_positive':tp, 'false_positive':fp, 'false_negative':fn}
    if len(all_gt_indices)==0:
        tp=0
        fp=len(pred_boxes)
        fn=0
        return {'true_positive':tp, 'false_positive':fp, 'false_negative':fn}

    gt_idx_thr=[]
    pred_idx_thr=[]
    ious=[]
    for ipb, pred_box in enumerate(pred_boxes):
        for igb, gt_box in enumerate(gt_boxes):
            iou= calc_iou(gt_box, pred_box)

            if iou >iou_thr:
                gt_idx_thr.append(igb)
                pred_idx_thr.append(ipb)
                ious.append(iou)
    iou_sort = np.argsort(ious)[::1]
    if len(iou_sort)==0:
        tp=0
        fp=len(pred_boxes)
        fn=len(gt_boxes)
        return {'true_positive':tp, 'false_positive':fp, 'false_negative':fn}
    else:
        gt_match_idx=[]
        pred_match_idx=[]
        for idx in iou_sort:
            gt_idx=gt_idx_thr[idx]
            pr_idx= pred_idx_thr[idx]
            # If the boxes are unmatched, add them to matches
            if(gt_idx not in gt_match_idx) and (pr_idx not in pred_match_idx):
                gt_match_idx.append(gt_idx)
                pred_match_idx.append(pr_idx)
        tp= len(gt_match_idx)
        fp= len(pred_boxes) - len(pred_match_idx)
        fn = len(gt_boxes) - len(gt_match_idx)
    return {'true_positive': tp, 'false_positive': fp, 'false_negative': fn}
```

Figura 54. Método get_single_image_results.

Fuente: Producción propia.

El método *get_single_image_results* devuelve verdadero positivo, falso positivo y falso negativo para el lote de cuadros delimitadores para una sola imagen.

```
def get_avg_precision_at_iou(gt_boxes, pred_bb, iou_thr=0.5):

    model_scores = get_model_scores(pred_bb)
    sorted_model_scores= sorted(model_scores.keys())
# Sort the predicted boxes in descending order (lowest scoring boxes first):
    for img_id in pred_bb.keys():

        arg_sort = np.argsort(pred_bb[img_id]['scores'])
        pred_bb[img_id]['scores'] = np.array(pred_bb[img_id]['scores'])[arg_sort].tolist()
        pred_bb[img_id]['boxes'] = np.array(pred_bb[img_id]['boxes'])[arg_sort].tolist()
        pred_boxes_pruned = deepcopy(pred_bb)
```

```
    rec_levels.append(recall_level)


avg_prec = np.mean(prec_at_rec)

#create precision recall curve
fig, ax = plt.subplots()
ax.plot(rec_levels, prec_at_rec, color='green')
#add axis labels to plot
ax.set_title('Precision-Recall Curve interpolated')
ax.set_ylabel('Precision')
ax.set_xlabel('Recall')
fig.savefig(f"{OUT_DIR}/Precision-Recall Curve interpolated.png")


return {
    'avg_prec': avg_prec,
    'precisions': precisions,
    'recalls': recalls,
    'model_thrs': model_thrs}
```

Figura 55. Método get_avg_precision_at_iou.

Fuente: Producción propia.

Finalmente, el método *get_avg_precision_at_iou* hace uso de los métodos anteriores para calcular el mAP utilizando la técnica de interpolación de 11 puntos. Para ello recibe los delimitadores de verdad y los delimitadores predichos de todas las imágenes del set de datos. Además, se puede determinar el umbral de IoU, que de no ser especificado, se usa un valor predeterminado de 0.5.

Para aplicarlo al modelo entrenado se utilizó el set de datos utilizado completo, es decir, el de entrenamiento y de validación. A continuación, se muestran los resultados obtenidos:

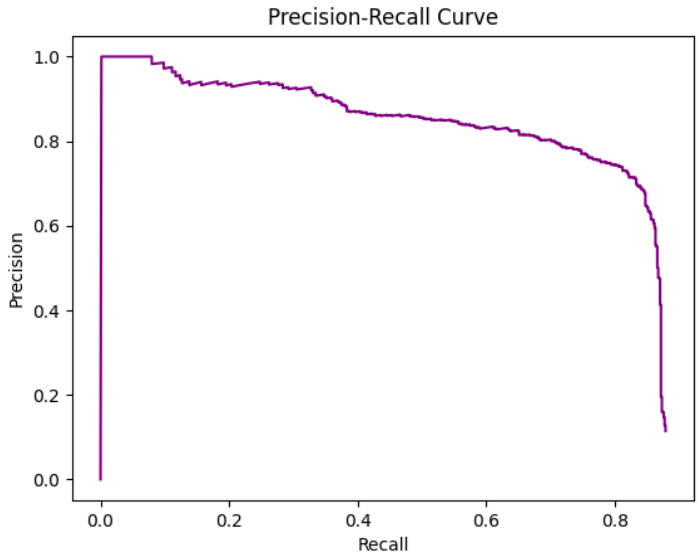


Figura 56. Gráfico de precisión y recall.

Fuente: Producción propia.

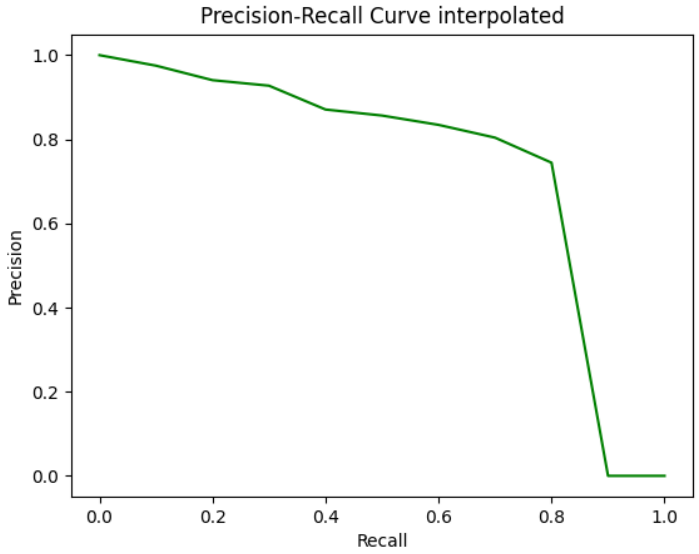


Figura 57. Gráfico de recuperación de precisión y precisión interpolada.

Fuente: Producción propia.

Las figuras 56 y 57 fueron obtenidas a partir de lo resuelto por el método get_avg_precision_at_iou. Adicionalmente y lo más importante, la performance del modelo entrenado fue de mAP 0.76, es decir, que el modelo tiene una precisión del 76% y con esta evaluación obtenida, se procedió a testear el nuevo modelo entrenado a través de una cámara web y arrojando los siguientes resultados:

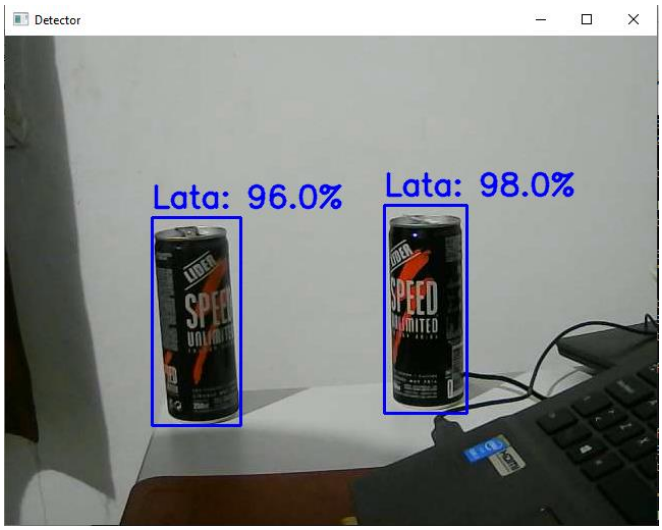


Figura 58. Detección de latas.

Fuente: Producción propia.

Figura 59. Detección de lata y botella defectuosas.

Fuente: Producción propia.

Figura 60. Detección de lata y botella.

Fuente: Producción propia.

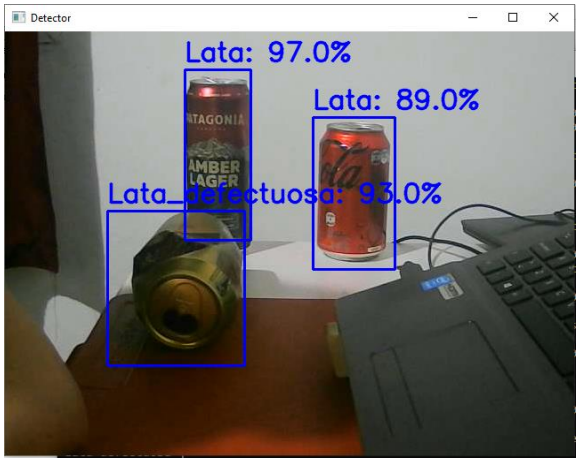


Figura 61. Detección de latas y lata defectuosa.

Fuente: Producción propia.

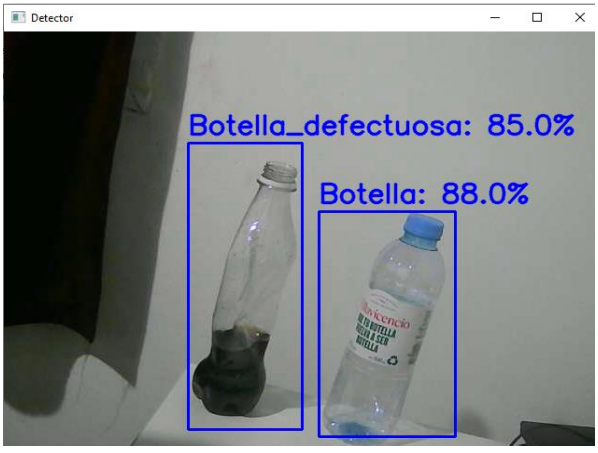


Figura 62. Detección de botella y botella defectuosa.

Fuente: Producción propia.

En base a los últimos resultados, se puede deducir que los resultados obtenidos son mucho mejores. Por otra parte, la detección en vivo mejoró muy poco a pesar de haber redimensionado las imágenes, esto puede deberse a la arquitectura Faster R-CNN ResNet50 FPN utilizada, ya que posee una gran arquitectura con 50 capas y las detecciones no se realizan en una GPU, por lo que, el procesamiento va a ser lento. Debido a esto, se realizó un segundo entrenamiento con el modelo Faster R-CNN Mobilenetv3 Large FPN teniendo en cuenta que en el inciso 5.8.2 se mencionó que este tiene como objetivo reducir la potencia computacional necesaria para el algoritmo a pesar de sacrificar precisión.

11. Entrenamiento modelo mobileNet

El objetivo de este segundo entrenamiento es mejorar la velocidad de transmisión por cámara web para obtener una detección en vivo lo más fluida posible. Para el mismo se emplearon los mismos parámetros e imágenes del inciso 8. Finalmente, el resultado obtenido cumplió con el objetivo de mejorar la velocidad de detección vía cámara web y como se esperaba la precisión en la detección si bien no empeoró demasiado, si disminuyó.

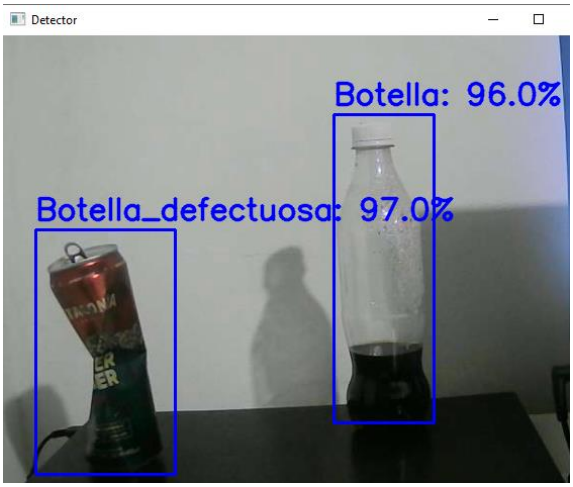


Figura 63. Detección con mobilenet.

Fuente: Producción propia.

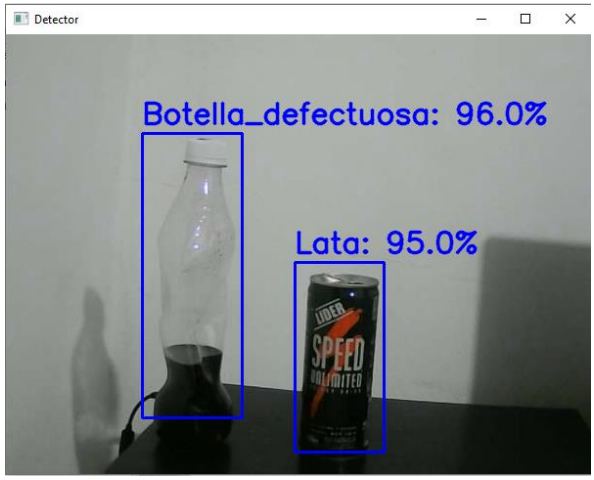


Figura 64. Detección con mobilenet.

Fuente: Producción propia.

Figura 65. Detección con mobilenet.

Fuente: Producción propia.

En la figura 63 se refleja la detección de una botella defectuosa lo que no es correcto, ya que se trata de una lata defectuosa y, además, se identificó una botella en lugar de detectar una botella defectuosa dado que este recipiente no estaba lleno. Por otro lado, en la figura 64 la detección de los dos objetos fue correcta. Finalmente, en la figura 65 se identifica incorrectamente una botella defectuosa debido a que es una lata defectuosa y se detecta correctamente una lata.

12. Rendimiento del modelo mobileNet

Para evaluar el rendimiento de este modelo se utilizó la técnica de interpolación de 11 puntos descrita en el inciso 9.1.

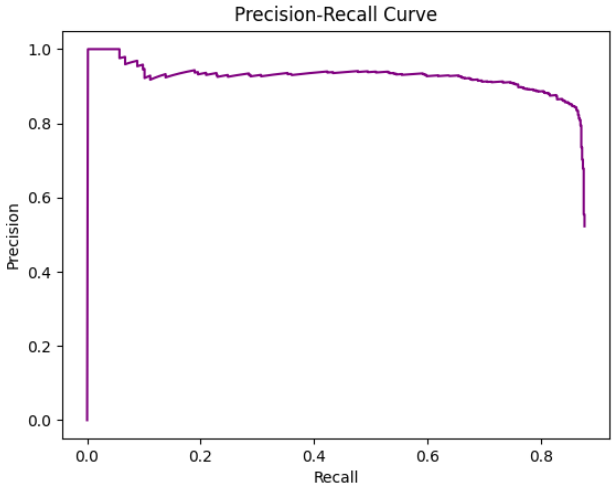


Figura 66. Gráfico de precisión y recall mobilenet.

Fuente: Producción propia.

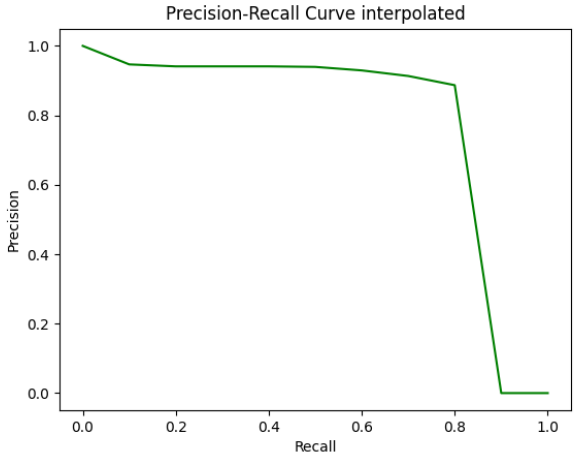


Figura 67. Gráfico de recuperación de precisión y precisión interpolada mobilenet.

Fuente: Producción propia.

La performance del modelo entrenado fue de mAP 0.72 sobre el set de datos completo (imágenes de entrenamiento e imágenes de validación). En comparación al modelo resnet50 entrenado anteriormente la precisión disminuyó un 0,04.

13. Pruebas en sistema embebido en tiempo real

Una vez obtenidos los modelos entrenados, se procedió a probarlos en una Raspberry Pi 3 utilizando un módulo Raspberry Pi Camera para poder realizar detecciones en tiempo real. Para ello se desarrolla el siguiente algoritmo de detección:

```
def detect_live_pi_camera(model, score_filter=0.6):

    print("[INFO] starting pi camera...")
    camera = picamera.PiCamera()
    camera.resolution = (1280, 960)
    camera.vflip = True
    camera.framerate = 30
    rawCapture = PiRGBArray(camera, size = (1280, 960))

    while True:
        for frame in camera.capture_continuous(rawCapture, format="bgr"):

            image_np = np.array(frame.array)

            print('Ejecutandose detectión..')
            labels, boxes, scores = predict(image_np, model)

            for i in range(boxes.shape[0]):
             if scores[i] < score_filter:
              continue

             print("Se está detectando: " )
             # text
             scoreValue=str((round(scores[i].item(), 2))*100)+"%"

             print(labels[i]+" "+scoreValue+"\n")
             box = boxes[i]

             text = '{}: {}'.format(labels[i], scoreValue)

             # font
             font = cv2.FONT_HERSHEY_SIMPLEX
             # org
             org = (00, 185)
             # fontScale
             fontScale = 1
             # Red color in BGR
             color = (255, 0, 0)
             # Line thickness of 2 px
             thickness = 2

             cv2.rectangle(image_np, (int(box[0]), int(box[1])), (int(box[2]), int(box[3])), color, thickness)
             if labels:
              cv2.putText(image_np, text, (int(box[0]), int(box[1] - 10)),font, fontScale, color, thickness, cv2.LINE_AA, I


             print('Hecho. Visualizando..')


            cv2.imshow('object detection', cv2.resize(image_np, (1280, 960)))
            rawCapture.truncate(0)
            if cv2.waitKey(25) & 0xFF == ord('q'):
                cv2.destroyAllWindows()
                break

    print('Saliendo')
    cap.release()
    cv2.destroyAllWindows()
```

Figura 68. Función detect_live_pi_camera.

Fuente: Producción propia.

La función de la figura anterior realiza la captura de *frames* o fotogramas obtenidos de la transmisión de la cámara y éstos se pasan a la etapa de predicción utilizando el modelo recibido en dicha función. Luego, una vez obtenidas las predicciones, por cada una se dibuja en el frame el recuadro junto con el nombre del objeto detectado y el score. Una vez realizado esto se devuelve el frame a la transmisión.

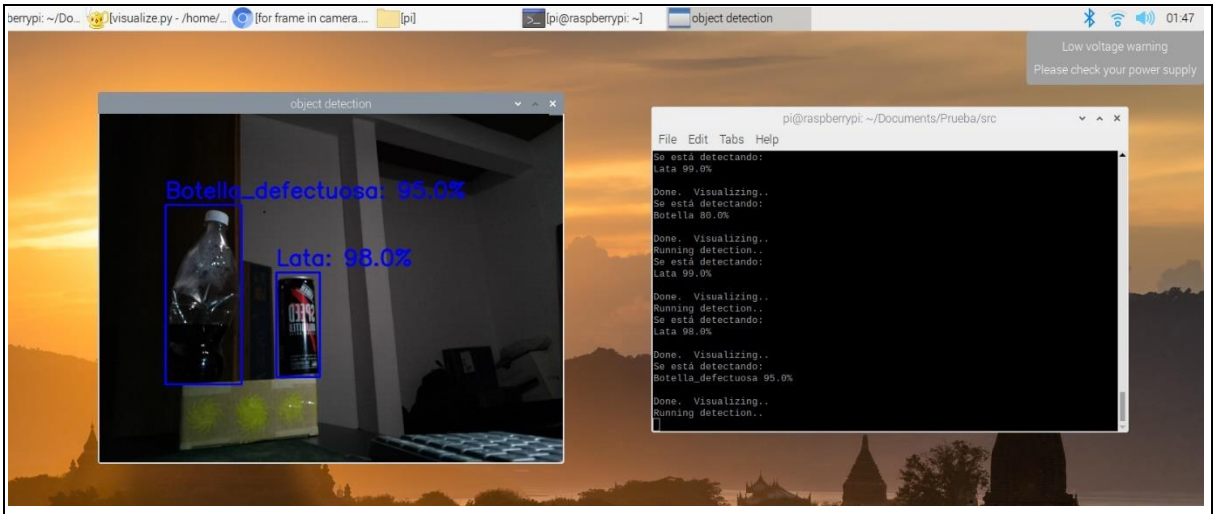


Figura 69. Detecciones en Raspberry Pi.

Fuente: Producción propia.

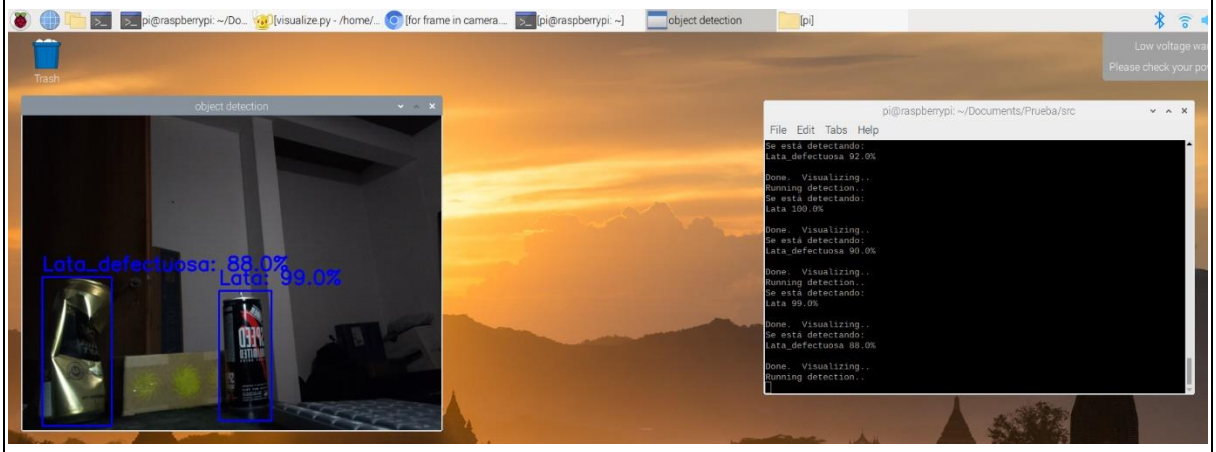


Figura 70. Detecciones en Raspberry Pi.

Fuente: Producción propia.

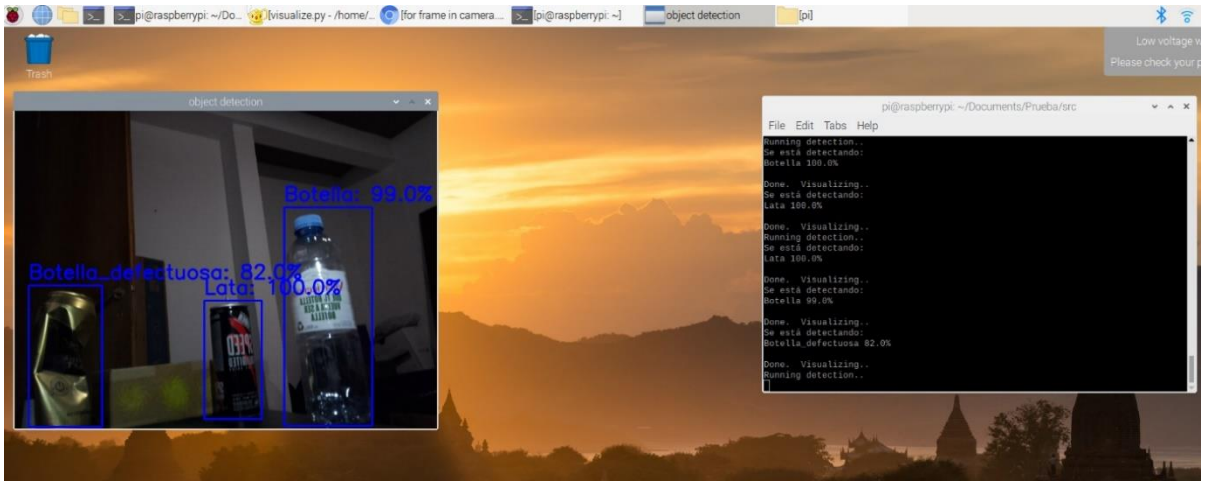


Figura 71. Detecciones en Raspberry Pi.

Fuente: Producción propia.

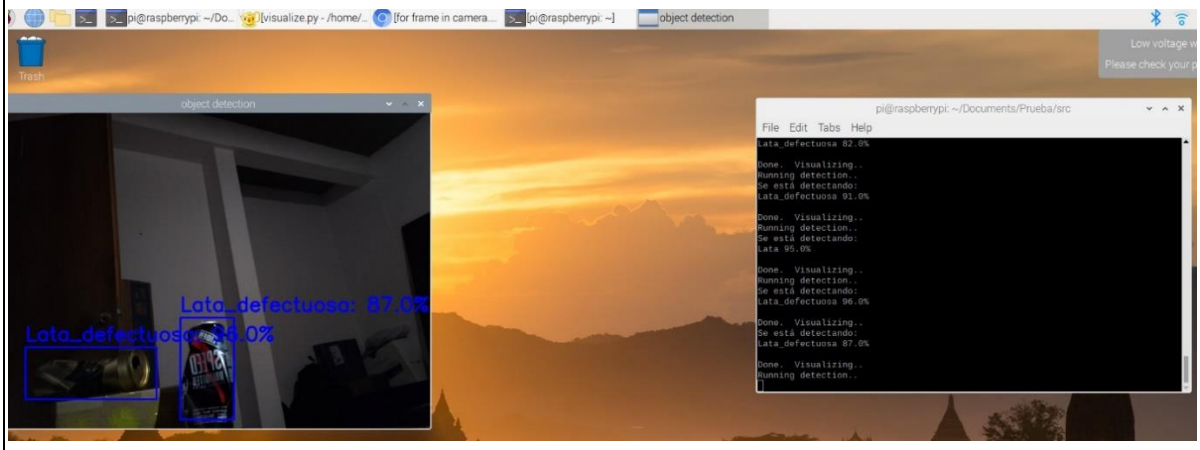


Figura 72. Detecciones en Raspberry Pi.

Fuente: Producción propia.

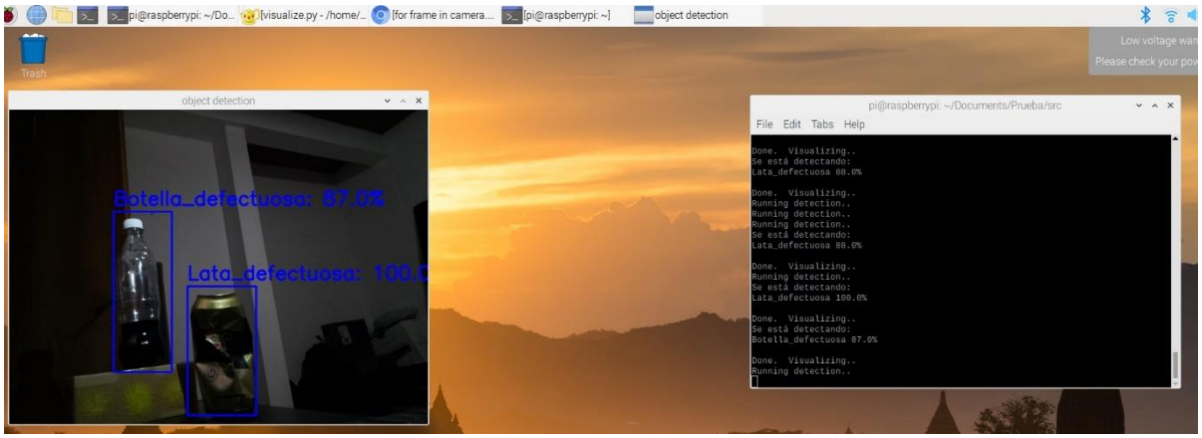


Figura 73. Detecciones en Raspberry Pi.

Fuente: Producción propia.

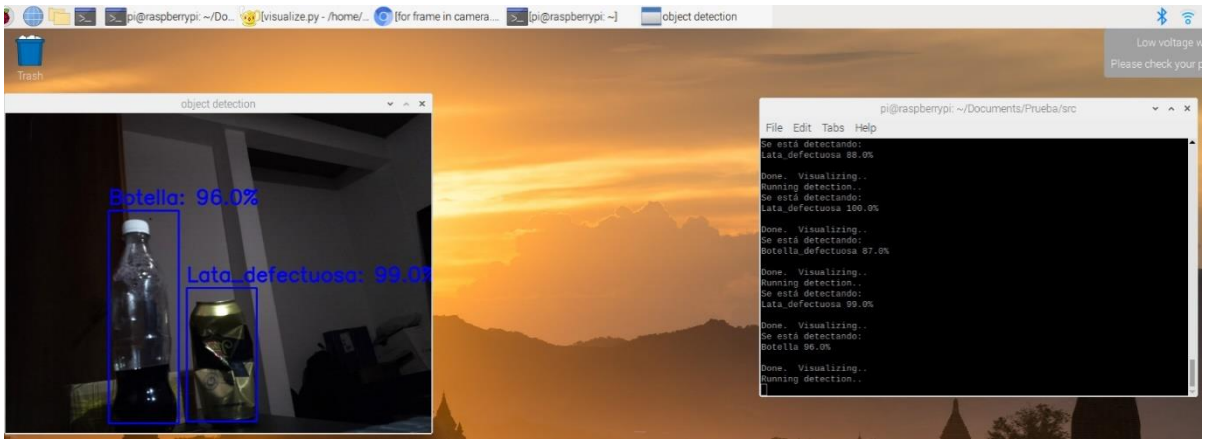


Figura 74. Detecciones en Raspberry Pi.

Fuente: Producción propia.

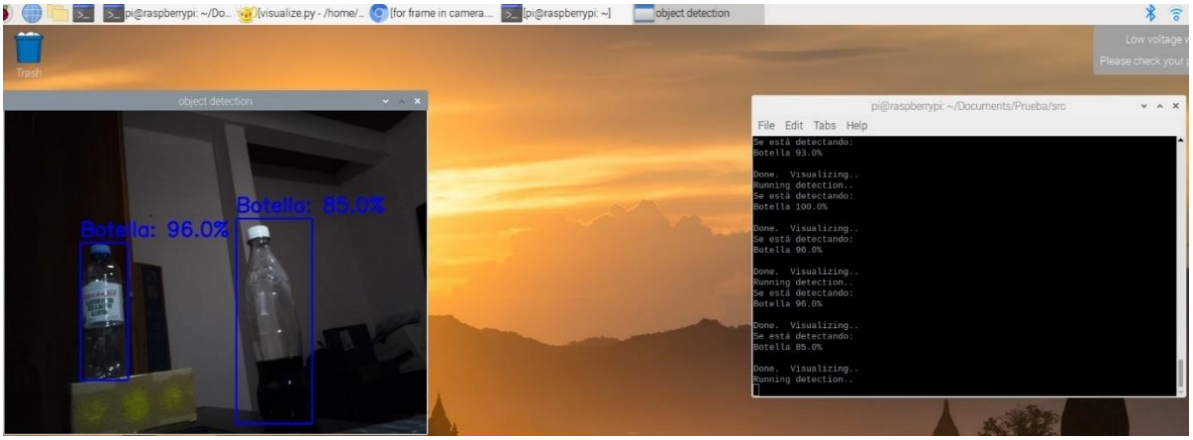


Figura 75. Detecciones en Raspberry Pi.

Fuente: Producción propia.

Las figuras anteriores son el resultado de las detecciones en la Raspberry y las mismas reflejan, en su mayoría, una buena precisión de detección. A continuación, se exponen los resultados de las detecciones:

Detecciones correctas:

✓ En la figura 69 se visualizan dos objetos, una botella defectuosa (95%) y una lata (98%).

✓ En la figura 70 se visualizan dos objetos, una lata defectuosa (88%) y una lata (99%).

✓ En la figura 72 se visualizan dos objetos, dos latas defectuosas con precisión del 96% y 87%.

✓ En la figura 73 se visualizan dos objetos, una botella defectuosa (87%) y una lata defectuosa (100%).

Detecciones incorrectas:

✓ En la figura 71 se visualizan dos objetos correctos y uno incorrecto, los objetos detectados correctamente son la lata (100%) y la botella (99%), mientras que la detección de botella defectuosa (82%) no es correcta ya que es una lata defectuosa.

✓ En la figura 74 se visualizan un objeto correcto y uno incorrecto, el objeto correcto es la lata defectuosa (99%), mientras que la detección de una botella (96%) no es correcta ya que es una botella defectuosa.

✓ En la figura 75 se visualizan un objeto correcto y uno incorrecto, el objeto correcto es la botella con precisión del 96%, mientras que la detección de la botella con precisión del 85% no es correcta ya que es una botella defectuosa.

14. Conclusiones

Para poder avanzar con el desarrollo del trabajo primero se tuvo que realizar una investigación sobre los principales conceptos acerca de machine learning y redes convolucionales y posteriormente sobre los conceptos avanzados puntualmente de redes neuronales para la detección de objetos. Una vez que la investigación fue realizada, se eligió Python como lenguaje de programación dado que éste tiene una gran variedad de bibliotecas abocadas a la inteligencia artificial. Por consiguiente, la disyuntiva estuvo en la elección entre las dos bibliotecas de aprendizaje automático de código abierto más potentes, Keras o PyTorch. Según lo investigado, Keras podría haber sido una buena elección debido a su simplicidad, sin embargo, para la detección de objetos implicaba la mayor parte del desarrollo en la configuración de archivos y prácticamente no había existencia de código debido a la API de alto nivel de Keras. Además, existía más información y soporte acerca de modelos de clasificación de objetos con dicha biblioteca que para modelos de detección de objetos. Por el contrario, para la creación de un modelo de detección de objetos con Pytorch se debía codificar y utilizar librerías debido a que ésta tiene una API de bajo nivel y provee más soporte gracias a que la comunidad que la brinda es mucho mayor que la de Keras. Por lo que la biblioteca elegida fue Pytorch.

Otro tema también fue la elección entre la creación de un modelo desde cero o la creación de un modelo a partir de uno pre entrenado. Por lo que, observando la limitante en cantidad de datos (imágenes) que se requieren para el entrenamiento de un modelo de detección de objetos y adicionalmente sabiendo que se deben detectar 4 tipo de objetos por imagen, se optó por un modelo pre entrenado aprovechando lo brindado por éste en la etapa de extracción de características.

También, otra situación fue la elección de los hiperparámetros para poder controlar el proceso de entrenamiento del modelo, ya que es bastante difícil encontrar los valores óptimos que se ajusten.

Por último, la prueba del modelo realizado en un sistema embebido en tiempo real pudo realizarse a pesar de inicialmente tener problemas con diversos módulos de la librería torchvision. El inconveniente se daba debido a que la versión de la librería instalada en la computadora donde se realizó el desarrollo contaba con algunos módulos clave para la utilización del modelo y para la raspberry en la versión de torchvision existente esos módulos no estaban disponibles. Para solucionar esto se debió modificar el código fuerte de la librería instalada en la raspberry agregando

los módulos faltantes. Una vez resuelto, se pudieron realizar las pruebas de manera exitosa. En dichas pruebas se efectuaron 7 capturas de detecciones, en las cuales se identificaron un total de 15 objetos de los cuales un 80% fueron clasificados correctamente, un 20% se clasificaron como falsos positivos mientras que hubo 0% de falsos negativos.

En conclusión, se pudo cumplir con los objetivos inicialmente propuestos y afrontar los problemas que se fueron presentando para finalmente obtener buenos resultados. Mientras esto ocurría, la experiencia permitió adquirir la habilidad de utilizar de nuevas librerías y desarrollando una capacidad analítica para mejorar el entrenamiento de la red convolucional.

14.1 Líneas futuras

En primer lugar, se podría aumentar el set de datos y volver a realizar el entrenamiento para poder obtener aún mejores resultados.

Otro tema a mejorar es la velocidad de transmisión por cámara web en la raspberry. A pesar de que utilizar la arquitectura mobileNet hizo que mejorara la velocidad de transmisión, todavía la velocidad no es óptima. Una solución podría crear un modelo de detección de objetos desde cero y así contar con menos capas de entrenamiento y, de este modo sería menos pesado el modelo a cargar en memoria, pero para ello es necesario aumentar el set de datos.

Finalmente, se podría implementar alguna aplicación móvil para realizar la detección de objetos defectuosos y no defectuosos para llevarlo a un plano más real, por ejemplo, a un depósito de mercadería o inclusive incluirlo en los departamentos de calidad de las empresas.

14.2 Reflexión sobre la práctica Profesional Supervisada como espacio de formación

Realizar este trabajo fue muy desafiante ya que no tenía conocimiento acerca de Machine Learning y todo lo que implicaba. Adicionalmente, como no tuve la oportunidad de ver y aplicar el tema en alguna materia de la carrera, me pareció una buena opción a elegir entre los temas propuestos para realizar la PPS.

Por otra parte, la inteligencia artificial es un tema latente en la sociedad actual en la que vivimos y como profesional es necesario ser capaz de comprender esa área.

Como resultado de la práctica retomé y perfeccioné mis conocimientos en el lenguaje de programación Python ya que no lo estaba utilizando desde hacía tiempo. También, durante el desarrollo de la práctica, aprendí a utilizar nuevas herramientas tales como Google Colab. Además, y lo más importante, es que incorporé nuevas aptitudes en materia de Deep Learning lo cual me podría llegar a servir a futuro a nivel laboral o académico.

Bibliografía

✓ Bagnato, Juan Ignacio, 2020, “Aprende Machine Learning en español”, https://www.aprendemachinelearning.com.

✓ Benavides, German, 2017, “Visión Artificial: la innovación disruptiva en la educación”, https://recursos.educoas.org/sites/default/files/5186.pdf, Universidad Jorge Tadeo Lozano, Bogotá, Colombia.

✓ Bismart, 2022, “¿Cuál es la diferencia entre el machine learning y el deep learning?”, https://blog.bismart.com/diferencia-machine-learning-deep-learning, España.

✓ Bootcamp AI, 2019, “Intro a las redes neuronales convolucionales”, https://bootcampai.medium.com/redes-neuronales-convolucionales-5e0ce960caf8#:~:text=Las%20redes%20neuronales%20convolucionales%20es,poder%20diferenciar%20unos%20de%20otros.

✓ Calvo, Diego, 2017, “Red Neuronal Convolucional CNN”, https://www.diegocalvo.es/red-neuronal-convolucional/.

✓ Dimecres, 2007, “Producto defectuoso: consecuencias para la empresa”, http://fibempresa.blogspot.com/2007/12/producto-defectuoso-consecuencias-para.html.

✓ García Villanueva, José David, 2020, “Redes neuronales desde cero (I) – Introducción”

✓ Gavilan, Ignacio, 2020, “Catálogo de componentes de redes neuronales (II): funciones de activación”, https://ignaciogavilan.com/catalogo-de-componentes-de-redes-neuronales-ii-funciones-de-activacion/.

✓ Gavilan, Ignacio, 2020,” Catálogo de componentes de redes neuronales (y IV): optimizadores”, https://ignaciogavilan.com/catalogo-de-componentes-de-redes-neuronales-y-iv-optimizadores/.

✓ Girshick, Ross, "Fast R-CNN," 2015, IEEE International Conference on Computer Vision (ICCV), pp. 1440-1448.

✓ Howard, Andrew - Sandler, Mark - Chu, Grace - Chen, Liang-Chieh - Chen, Bo-Tan, Mingxing - Wang Weijun - Zhu Yukun - Pang Ruoming - Vasudevan Vijay - V. Le, Quoc-Adam, Hartwig, 2019,” Searching for MobileNetV3”,

https://openaccess.thecvf.com/content_ICCV_2019/papers/Howard_Searching_for_MobileNetV3_ICCV_2019_paper.pdf

✓ Hui, Jonathan, 2018, “Understanding Feature Pyramid Networks for object detection (FPN)”, https://jonathan-hui.medium.com/understanding-feature-pyramid-networks-for-object-detection-fpn-45b227b9106c.

✓ Iberdrola S.A., “¿Qué es la visión artificial y cuáles son sus aplicaciones?”, https://www.iberdrola.com/innovacion/vision-artificial.

✓ Khandelwal, Renu, 2020, “Evaluating performance of an object detection model”, https://towardsdatascience.com/evaluating-performance-of-an-object-detection-model-137a349c517b

✓ Martínez, Jesús, 2020, “¿Qué es un Optimizador y Para Qué Se Usa en Deep Learning?”, https://datasmarts.net/es/que-es-un-optimizador-y-para-que-se-usa-en-deep-learning/.

Nabi, Javaid, 2019, “Hyper-parameter Tuning Techniques in Deep Learning”, https://towardsdatascience.com/hyper-parameter-tuning-techniques-in-deep-learning-4dad592c63c8

✓ Olivera, Oscar García-Olalla, 2019, “Redes Neuronales artificiales: Qué son y cómo se entrenan”, https://www.xeridia.com/blog/redes-neuronales-artificiales-que-son-y-como-se-entrenan-parte-i.

Pytorch, 2019, “FROM RESEARCH TO PRODUCTION”, https://pytorch.org/

Pytorch, “PYTORCH DOCUMENTATION”, https://pytorch.org/docs/stable/index.html

✓ Quattrociocchi, Victoria, 2021, “Redes Neuronales Convolucionales para Procesamiento del Lenguaje Natural”, https://victoriaquattrociocchi.hashnode.dev/redes-neuronales-convolucionales-para-procesamiento-del-lenguaje-natural.

✓ Sardana, Nikhil, 2017, “Object Detection”, https://tjmachinelearning.com/lectures/1718/obj/.

✓ Tzutalin, “Labelimg is a graphical image annotation tool”, 2015, https://github.com/tzutalin/labelImg.

✓ Universidad Complutense, 2021, “¿Qué es el Machine Learning?”, https://www.masterdatascienceucm.com/que-es-machine-learning/, Madrid, España.