Facultad de Informática

Trabajo de Fin de Grado

Constructo: Sistema serverless para la generación de topologías mediante Machine Learning

Por
Pablo Ezquerro Moya

UNIVERSIDAD COMPLUTENSE
MADRID

Dirigido por

José Luis Vázquez Poletti
Juan Carlos Fabero Jiménez
Colaborador Externo: David Pacios Izquierdo

MADRID, 2023–2024

Abstract

This work explores the development of a serverless system for generating network topologies using machine learning techniques, specifically through the use of YOLO models. The system leverages object recognition to automatically convert hand-drawn diagrams into digital formats that can be imported directly into GNS3 for network simulation. By automating this process, the project addresses the cumbersome and time-consuming task of manual network configuration, potentially revolutionizing network design practices in educational and professional settings. The implementation uses AWS Lambda to handle processing, providing a cost-effective and scalable solution.

Este trabajo explora el desarrollo de un sistema *serverless* para la generación de topologías de red utilizando técnicas de machine learning, específicamente a través del uso de modelos YOLO. El sistema utiliza el reconocimiento de objetos para convertir automáticamente diagramas dibujados a mano en formatos digitales que pueden importarse directamente en GNS3 para la simulación de redes. Al automatizar este proceso, el proyecto aborda la tarea engorrosa y que consume mucho tiempo de la configuración manual, con el potencial de revolucionar las prácticas de diseño de redes en entornos educativos y profesionales. La implementación utiliza AWS Lambda para manejar el procesamiento, proporcionando una solución rentable y escalable.

Palabras Clave: Arquitectura Serverless, Generación de Topologías de Red, Machine Learning, YOLO (You Only Look Once), Reconocimiento de Objetos, AWS Lambda, Simulación de Redes GNS3.

Índice general

Chapter 1. Introduction

1.1. Context

The project we have been working on emerged as a response to the need to streamline the process of creating network topologies in the GNS3 tool. This tool allows for the design of networks to simulate real-world scenarios using various elements and components.

Generally, the most tedious and demotivating part of computing is configuring the workspaces, or environments, in which to apply your knowledge. This is something that most of us who study or have studied any branch of computing can agree on. Using a tool like GNS3 allows you to experiment with different topologies using each network element independently. However, as mentioned earlier, the process of setting up the environment, the network, and its elements, is boring and monotonous.

In GNS3, topology configuration is done by dragging elements from an element selector on the left side of the application. These elements must be dragged to their desired position and then connections between them must be established. When the network is small, this is not a difficult task. However, as the number of elements and network connections increases, it becomes more complicated.

1.2. Objective

To address this problem, we came up with the idea of creating network topologies by drawing them manually with paper and pen. Then, a program, using object recognition techniques, would be able to interpret these drawings and automatically generate a file that can be imported into GNS3 with the resulting topology.

The drawings should follow the following legend to represent the elements:

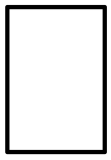


Figura 1.1: SRV

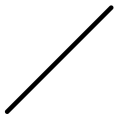


Figura 1.2: Link

Figura 1.3: Router

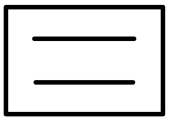


Figura 1.4: Switch

The process to convert the hand-drawn diagrams into importable topologies for GNS3 consists of the following steps:

1. Draw the topology by hand: The user creates a drawing using paper and pen. The symbols used must follow the provided legend to correctly represent the network elements.
2. Scan the drawing: The drawing is digitized using a scanner or a high-quality photo application.
3. Object recognition: The developed program uses image recognition techniques to identify the symbols in the drawing and their connections. The information about the detected objects is stored in a .txt file.
4. Generation of a file for GNS3: From the .txt file, a .json file is generated with the structure of a GNS3 project.

This process allows users to quickly create network topologies using simple tools. Through object recognition techniques, hand-drawn sketches are transformed into real network simulation projects.

Capítulo 1. Introducción

1.1. Contexto

El proyecto en el que hemos estado trabajando surgió como respuesta a la necesidad de agilizar el proceso de creación de topologías de red en la herramienta GNS3. Esta herramienta permite diseñar redes para simular escenarios reales utilizando diversos elementos y componentes.

Generalmente, la parte mas tediosa y desmotivadora de la informática es la configuración de los espacios de trabajo, o entornos, en los que poder aplicar los conocimientos. Eso es algo que la mayoría de personas que estudiamos o hemos estudiado alguna rama de informática estamos de acuerdo. Usar una herramienta como GNS3 permite poder experimentar con distintas topologías utilizando cada elemento de la red de manera independiente. Pero como hemos comentado antes, el proceso de configurar el entorno, la red y sus elementos, es algo aburrido y monótono.

En GNS3 la configuración de la topología se realiza arrastrando elementos desde un selector de elementos ubicado en la parte izquierda de la aplicación. Estos elementos hay que arrastrarlos a su posición deseada y luego establecer las conexiones entre ellos. Cuando la red es pequeña, no es un tarea difícil. Pero cuando se va incrementando el número de elementos y conexiones de la red, se vuelve más complicado.

1.2. Objetivo

Para abordar este problema, se nos ocurrió la idea de crear las topologías de red dibujándolas a mano con papel y bolígrafo. Luego, un programa, mediante técnicas de reconocimiento de objetos, sería capaz de interpretar esos dibujos, y generar automáticamente un archivo que pueda importarse a GNS3 con la topología resultante.

Los dibujos deberán seguir la siguiente leyenda para representar los elementos:

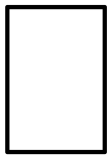


Figura 1.1: SRV

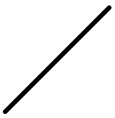


Figura 1.2: Enlace

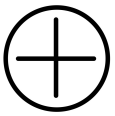


Figura 1.3: Router

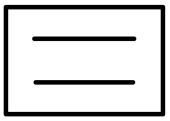


Figura 1.4: Switch

El proceso para convertir los dibujos a mano en topologías importables a GNS3 consta de los siguientes pasos:

1. Dibujar la topología a mano: el usuario crea un dibujo utilizando papel y bolígrafo. Los símbolos utilizados deben seguir la leyenda proporcionada para representar correctamente los elementos de red.
2. Escanear el dibujo: el dibujo se digitaliza mediante un escáner o una aplicación de fotos de alta calidad.
3. Reconocimiento de objetos: el programa desarrollado utiliza técnicas de reconocimiento de imágenes para identificar los símbolos en el dibujo y sus conexiones. La información de los objetos detectados queda almacenada en un archivo *.txt*.
4. Generación de archivo para GNS3: a partir del archivo *.txt* se genera un archivo *.json* con la estructura de un proyecto GNS3.

Este proceso permite a los usuarios crear topologías de red de manera rápida, utilizando herramientas simples. A través de técnicas de reconocimiento de objetos, los dibujos a mano se convierten en proyectos reales de simulaciones de red.

Capítulo 2. Estado del Arte

La manera en la que se entiende en la sociedad el objetivo de la informática ha ido cambiando a lo largo de los años. La principal función es facilitar el acceso a la información y su correcto uso, así como mejorar la eficiencia y productividad en el trabajo. Pero cada vez son mas tecnologías las que están en desarrollo, y tecnologías como la inteligencia artificial pueden contribuir a hacer la tecnología más amigable y accesible para todos.

Para el proyecto se han utilizado herramientas que han producido grandes avances en otros campos. Se han buscado algunos usos en diversos proyectos.

2.1. Melanoma Lesion Detection Using YOLOv4-DarkNet

El uso de técnicas avanzadas de reconocimiento [1] de patrones ha revolucionado muchas áreas, por ejemplo, en el campo de la salud. YOLOv4-Darknet se ha utilizado para detectar melanoma, un tipo de cáncer de piel causado por la radiación ultravioleta del sol. La detección temprana del melanoma es crucial ya que su diagnóstico tardío puede llevar a complicaciones graves. Con métodos como YOLOv4-Darknet se puede acelerar la detección mediante el análisis de imágenes de la piel, lo que puede ayudar a un diagnóstico temprano.

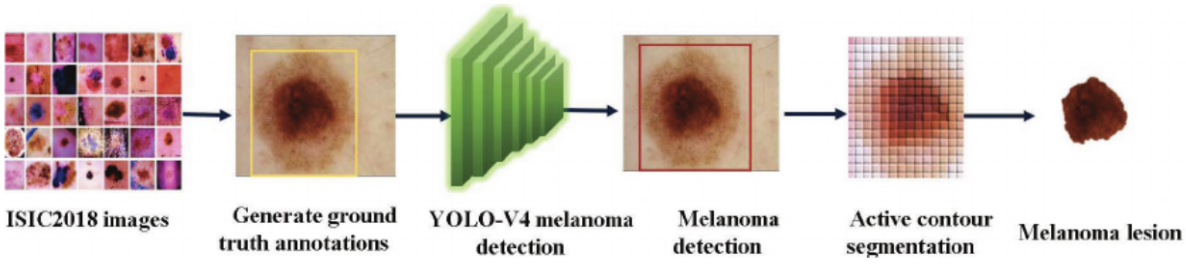


Figura 2.1: Detección Melanoma

La Figura 2.1[1] muestra el resultado de este trabajo. La capacidad que tiene YOLOv4-DarkNet para reconocer patrones nos da la certeza de que

[1]Diagrama del proyecto obtenido del artículo: https://ieeexplore.ieee.org/abstract/document/9247186

es un sistema de detección de objetos que debería funcionar correctamente con el cometido de nuestro proyecto.

2.2. Circuit Recognition Using YOLOv5

Este artículo [2] presenta un enfoque para transcribir circuitos electrónicos dibujados a mano a esquemas compatibles con programas de simulación como LTSpice o PSpice. El proceso de digitalización implica la detección de componentes y el rastreo de conexiones, enfrentando desafíos como variaciones en estilo de dibujo, calidad del papel y ruido en las imágenes.

Para abordar estos problemas, el artículo propone un algoritmo en tiempo real basado en técnicas de aprendizaje profundo. La Figura 2.2[2] ilustra cómo es el reconocimiento de circuitos para este trabajo. YOLOv5 se utiliza para detectar los componentes del circuito, mientras que un enfoque basado en la transformada de Hough [3] se emplea para el reconocimiento de nodos y las conexiones entre componentes.

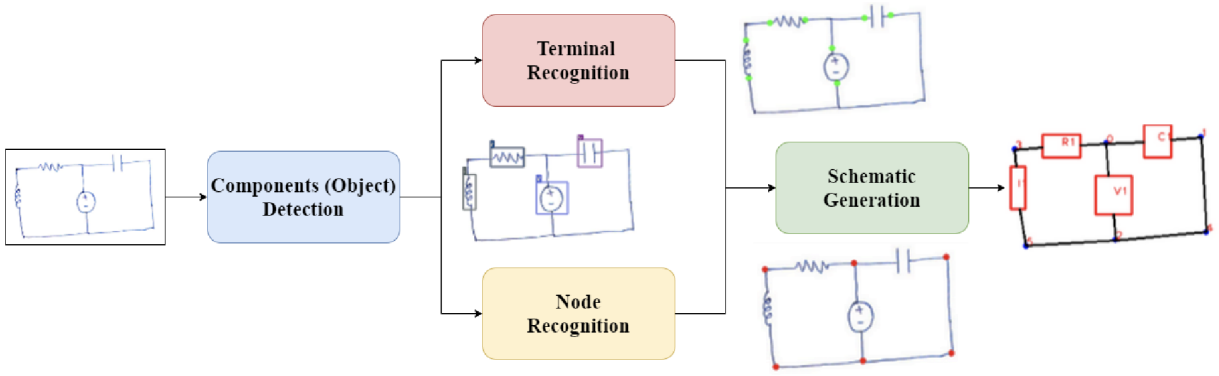


Figura 2.2: Reconocimiento de circuitos

2.3. Real-Time Flying Object Detection with YOLOv8

El siguiente proyecto aborda la problemática de detección de objetos voladores [4], en concreto los drones. Los drones son difíciles de detectar por su tamaño pequeño, maniobrabilidad, y baja firma electromagnética. Estos muchas veces se utilizan con intenciones maliciosas lo que provoca que surga la necesidad de tener un detector rápido y eficaz.

[2]Diagrama obtenido del artículo: https://arxiv.org/abs/2106.11559

El artículo propone un modelo para detectar objetos voladores en tiempo real usando YOLOv8. El modelo generalizado se entrena con 40 categorías de objetos voladores. La principal ventaja que se describe sobre YOLOv8 frente a otros modelos es la velocidad de detección y entrenamiento. La Figura 2.3[3] muestra el resultado de estas detecciones en el caso de los objetos voladores.

Figura 2.3: Reconocimiento de objetos voladores

2.4. Simulación de una red SDN de videovigilancia

En este proyecto [5] se propone desarrollar un sistema de videovigilancia IP utilizando tecnología SDN (Software-Defined Networking). La idea central es adaptar el encaminamiento de los paquetes según el volumen de tráfico y el estado de la red. Dado que no se cuenta con el equipo físico necesario para realizar pruebas reales, se recurre a la simulación mediante GNS3. A través de esta simulación, se exploran diferentes escenarios en los que las rutas de tráfico se ajustan dinámicamente.

El objetivo principal es priorizar el tráfico de videovigilancia IP sobre otros servicios, optimizando tanto el rendimiento de este servicio específico como el de la red en su conjunto.

Esto nos da a entender la cantidad de posibilidades que tiene GNS3 y por qué ha sido elegido como la herramienta para la que digitalizar nuestros dibujos de topologías de red.

[3]Imágenes de las detecciones obtenidas del artículo:https://arxiv.org/abs/2305.09972

2.5. A serverless computing architecture for Martian aurora detection with the Emirates Mars Mission

Este artículo [6] tiene como objetivo el detectar auroras marcianas en la superficie de Marte. Para su análisis utilizaron las imágenes realizadas por la misión HOPE. Fueron utilizadas 200 imágenes, con auroras y sin presencia de las mismas, para la creación del conjunto de datos.

El objetivo principal es agilizar el proceso de detección de auroras y hacerlo más escalable, para ello, se han valido de AWS Lambda y SageMaker para realizar estas detecciones.

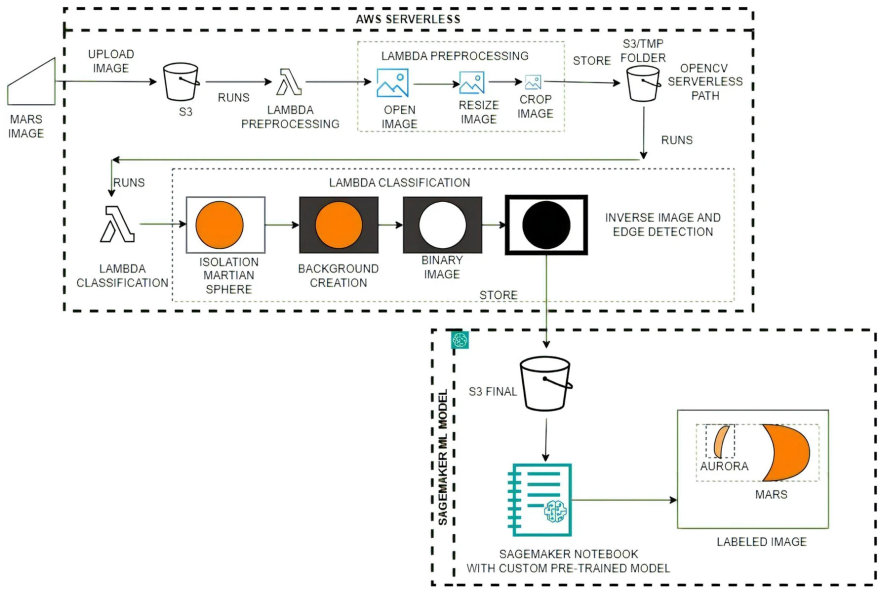


Figura 2.4: Esquema de la arquitectura realizada para la detección de las auroras.

La Figura 2.4[4] muestra el camino que realiza cada una de las imágenes para ser procesada. Para ello, una vez es subida se realiza un preprocesado de una imagen a través de una función de AWS Lambda donde se delimita la zona de detección. Una vez está preprocesada, se depositará en un S3 que disparará la función AWS Lambda dedicada a la clasificación. Para la realización de esta clasificación se aisla la zona concreta de aparición de la aurora, después se le agrega un fondo para poder binarizar la imagen, es decir, delimitar mediante un umbral qué píxeles serán negros y cuáles serán blancos. Después se invierte y se envía a SageMaker para que, mediante el conjunto de datos se determine qué es la aurora y qué es Marte.

[4]Diagrama obtenido del artículo: https://www.nature.com/articles/s41598-024-53492-4

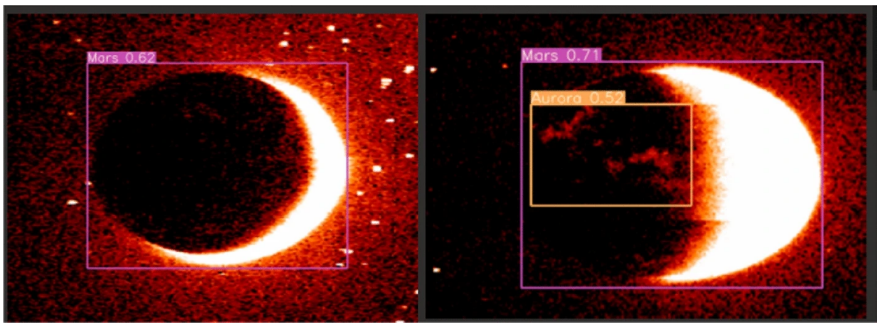


Figura 2.5: Resultado final de la detección de una aurora marciana.

Como se puede observar en la Figura 2.5[5], se ha realizado tanto una detección de Marte, como de Marte y la aurora.

De este trabajo se ha obtenido la idea de identificar cada una de las figuras. Este trabajo combina todo lo visto anteriormente con YOLO para la realización de las detecciones. El punto interesante es la utilización de SageMaker para la realización de las detecciones sin depender de los recursos del ordenador propio.

2.6. Combination of deep cross-stage partial network and spatial pyramid pooling for automatic hand detection

Este artículo [7] trata sobre un estudio centrado en la detección de objetos mediante redes neuronales convolucionales (CNN). Para la realización de estos estudios han realizado una revisión bibliográfica sobre distintas metodologías. Cabe destacar que el trabajo que más está relacionado con su metodología es uno centralizado en la detección de señales de tráfico.

También se abordan temas relacionados con el rendimiento y la optimización de modelos de CNN, como la evaluación del tiempo de entrenamiento en diferentes arquitecturas, como Inception-v3 y ResNet. Se mencionan mejoras incrementales en modelos populares de detección de objetos, como YOLOv3 y YOLOv4, así como el desarrollo de nuevas arquitecturas de red, como CSPNet, para mejorar la capacidad de aprendizaje de las CNN.

Se han realizado distintos casos de uso donde se aplican este tipo de redes. Caben destacar la detección de señales de tráfico, los cascos de moto

[5]Imagen obtenido del artículo: https://www.nature.com/articles/s41598-024-53492-4

y otros casos relacionados con la seguridad vial. El resultado y el diagrama completo es mostrado en la Figura 2.6[6].

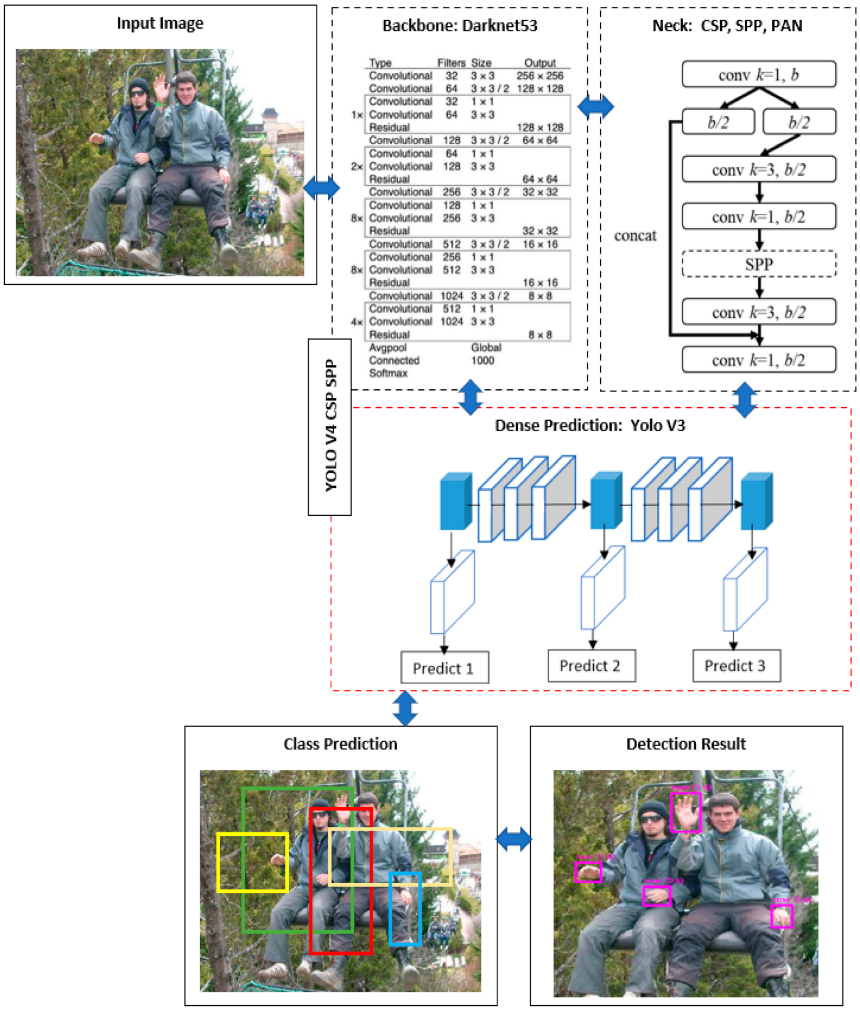


Figura 2.6: Resultado final tras aplicar las CNN para la detección de cascos y manos.

De este trabajo se ha obtenido información acerca de distintas metodologías para la realización la detección de objetos, en este caso se ha realizado mediante CNN.

2.7. Deep learning

Este artículo [8] realiza una revisión del estado del arte sobre las redes neuronales profundas (DNN). De estas aplicaciones se destacan desde el reconocimiento de imágenes hasta el reconocimiento de lenguaje natural.

Se hace hincapié en la importancia de mejorar aún más la comprensión de las DNN y su capacidad para el razonamiento complejo. Se mencionan

[6]Diagrama y resultado obtenidos del artículo: https://doi.org/10.3390/bdcc6030085

avances recientes en el aprendizaje de representaciones y se discute la necesidad de desarrollar nuevas estrategias que combinen el aprendizaje de representaciones con el razonamiento complejo. Se señala que el futuro de la inteligencia artificial dependerá en gran medida de sistemas que integren ambas capacidades de manera efectiva.

También cabe destacar que este artículo menciona que esta tecnología todavía está sentando bases, por lo que las investigaciones realizadas todavía están expandiendo los límites de esta tecnología. Concluye indicando que se tiene que seguir investigando sobre las DNN para la realización de nuevas inteligencias artificiales y para la realización de nuevas DNN.

De este artículo se ha obtenido una nueva metodología para la realización de las detecciones. Para este caso se ha estudiado en profundidad las DNN.

2.8. Static Analysis for AWS Best Practices in Python Code

Este preprint [9] realiza un análisis de distintos tipos de operaciones que se pueden realizar en Python en el contexto de AWS mediante el uso de API. Estas API permiten a los usuarios definir filtros para seleccionar nodos de acción basados en criterios específicos, como el nombre de la función o el número de argumentos. Además, las operaciones de transformación permiten modificar nodos de acción o datos de acuerdo con ciertas reglas, como transformar un nodo de acción en sus argumentos respectivos o sus nodos de captura de errores asociados.

Este tipo de operaciones se realizan para entender el flujo de las operaciones dentro de Python. Por ejemplo, las funciones de filtrado pueden ayudar a identificar vulnerabilidades en seguridad, mientras que las operaciones de transformación de flujo de datos pueden ayudar a identificar si hay algún fallo en la ruta, ya sea en la fuente o en el destino.

También se ilustran distintos casos de uso en relación a las API. Hace referencia a una serie de trabajos de investigación relevantes en el campo del análisis estático de programas, como estudios sobre detección de uso incorrecto de API, inferencia de tipos estáticos y análisis de flujo de datos en Python. Estas investigaciones proporcionan un contexto importante para comprender la importancia y la aplicación práctica de las herramientas y técnicas descritas en el documento.

De este preprint se ha obtenido la base para la realización de la API y cómo coordinarla con AWS. También se hace un enfoque interesante que ha servido para evitar errores en la realización del código en Python en AWS.

2.9. You only look once: unified, real-time object detection

Este estudio [10] presenta un modelo unificado detallado para la detección de imágenes basado en YOLO (You Only Look Once). Para el desarrollo de este modelo se van a comparar los modelos R-CNN y Fast R-CNN.

Este tipo de detección se realiza en tiempo real y está basado en la realización mediante varias capas. Después se llevan a cabo cada uno de los conjuntos de entrenamiento para los modelos R-CNN y Fast R-CNN.

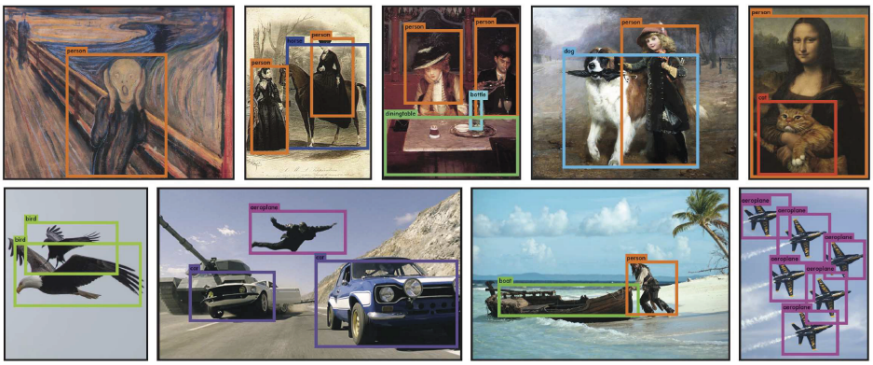


Figura 2.7: Detección de imágenes aplicado a distintos casos de uso aplicando YOLO.

Concluye comparando cada uno de los modelos y estableciendo las limitaciones a la hora de implantar YOLO. Una de las principales que se destaca es la falsa detección, donde se puede observar que hay veces que YOLO puede fallar. La Figura 2.7[7] muestra el resultado de las distintas detecciones.

Este artículo ha sido revisado con el fin de determinar qué versión de YOLO se ajustaba más a la detección que se quería realizar. También se han visto sus limitaciones.

[7]Imagen obtenida del artículo: https://www.cv-foundation.org/openaccess/content_cvpr_2016/html/Redmon_You_Only_Look_CVPR_2016_paper.html

2.10. A Cloud Based Sentiment Analysis through Logistic Regression in AWS Platform

Este estudio [11] presenta un análisis de sentimientos a través de la regresión logística de Amazon Web Service (AWS). Cabe destacar la inclusión de AWS en el ámbito de *cloud computing* y cómo ha simplificado el uso del cliente para la realización de las labores en la nube. Se destaca la importancia de la infraestructura en la nube y se mencionan características como el equilibrio de carga, la autoescalabilidad y los acuerdos de nivel de servicio.

El análisis de sentimientos es una herramienta interesante para comprender las opiniones de los usuarios sobre productos o servicios. Como metodología se ha utilizado un algoritmo de aprendizaje automático, en este caso, es la regresión logística para clasificar las opiniones de los clientes en positivas o negativas.

El estudio incluye una revisión exhaustiva de la literatura relacionada con el análisis de sentimientos, abordando temas como la optimización de la carga de trabajo, la clasificación de datos en la nube, el análisis de opiniones en redes sociales y el uso de algoritmos de aprendizaje automático, como las redes neuronales recurrentes y las máquinas de vectores de soporte. Además, se exploran herramientas y tecnologías relevantes, como Amazon Kinesis Data Firehose, para la ingesta de datos en la nube.

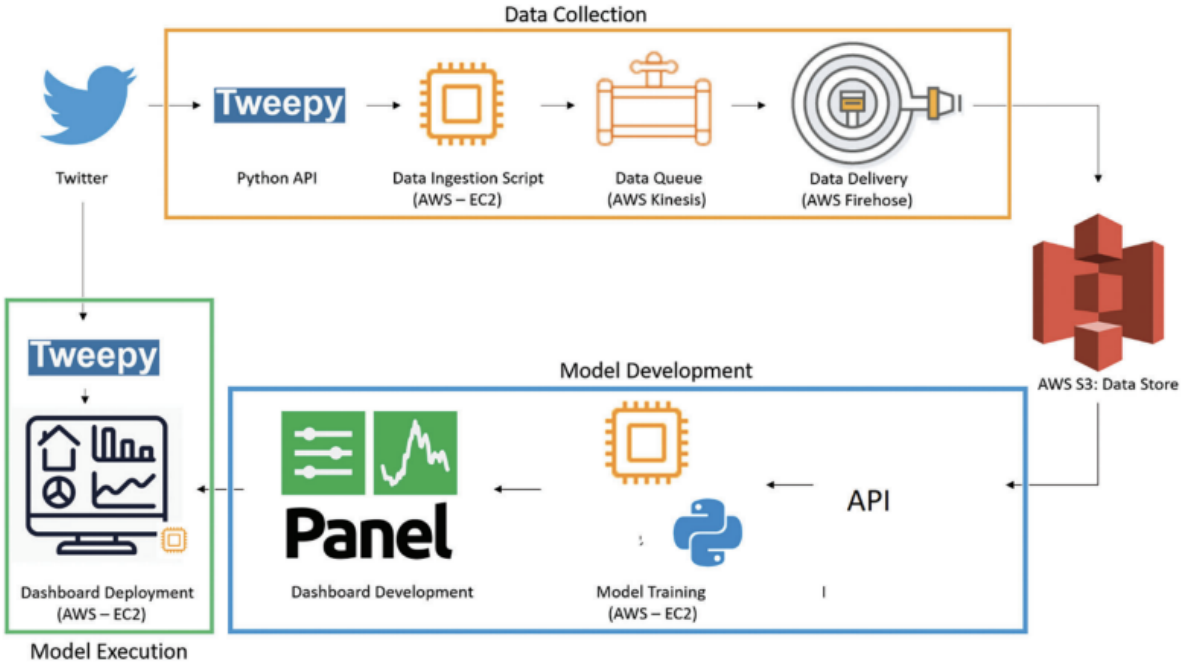


Figura 2.8: Arquitectura realizada para el análisis de sentimientos.

La Figura 2.8[8] muestra cómo han realizado el análisis de sentimientos a través de la red social Twitter. Para ello, primero han adquirido los mensajes mediante la aplicación Tweepy, estos datos son procesados y colectados en Python.

Este estudio ha sido utilizado para entender cómo coordinar la aplicación en Python con AWS y así ver otro caso de uso aplicado a una aplicación donde se combinan Python y AWS.

2.11. Performance Evaluation of Deep Learning Algorithm Using High-End Media Processing Board in Real-Time Environment

El artículo [12] aborda la implementación de sistemas de vigilancia de tráfico utilizando modelos de detección de objetos basados en aprendizaje profundo, específicamente YOLOv3, YOLOv4, YOLOv5s y YOLOv3-tiny. Se discuten varios aspectos de la implementación, incluida la configuración del hardware y el software utilizados, como el entorno de hardware con un procesador Intel Xeon Silver y una GPU NVIDIA Jetson

[8]Diagrama obtenido del artículo: https://cdn.techscience.cn/ueditor/files/csse/TSP_CSSE-45-1/TSP_CSSE_31321/TSP_CSSE_31321.pdf

Xavier, junto con el entorno de software que incluye Python y el marco Pytorch. Se destacan métricas importantes como la temperatura de la GPU, el tiempo de inferencia y la utilización de la memoria RAM en relación con la resolución de las imágenes y el modelo utilizado.

Se observa que el modelo YOLOv3-tiny mantiene tiempos de inferencia más bajos y una menor utilización de la GPU en comparación con otros modelos, lo que lo convierte en una opción atractiva para implementaciones en la placa Jetson Xavier si se busca mantener la utilización de la GPU bajo control. Sin embargo, se señala que YOLOv5s utiliza más recursos de GPU en todas las resoluciones de imagen, lo que puede ser una consideración importante dependiendo de los requisitos del sistema. Además, se menciona que YOLOv4 y YOLOv3-tiny tienen tiempos de inferencia más bajos que otros modelos, lo que los hace adecuados para aplicaciones de transmisión en vivo donde se requiere una rápida detección de objetos en tiempo real.

Cabe destacar que se demuestran casos de uso aplicados para cada tipo de YOLO. Se presenta un caso de uso para una detección de vehículos en lo que se demuestra una alta de éxito. Los investigadores permiten acceso a su *dataset* y así como las distintas detecciones para validar su caso de uso.

Con este artículo se ha obtenido una perspectiva acerca de las distintas versiones de YOLO y se ha seleccionado cuál era la óptima para la presentada en este trabajo.

2.12. Ship Target Detection Algorithm Based on Improved YOLOv3 for Maritime Image

Esta publicación [13] presenta un algoritmo para la detección de embarcaciones basado en AE-YOLOv3. Se ha utilizado este algoritmo para realizar detecciones en imágenes de alta complejidad. Para la metodología se comparó esta tecnología con otras, como por ejemplo, YOLOv3, SSD y Faster R-CNN. AE-YOLOv3 demuestra una mejora sustancial en la detección de objetivos pequeños, la ocultación de objetivos y la información incompleta del objetivo, lo que resulta en una reducción significativa en el número de objetivos no detectados.

El algoritmo AE-YOLOv3 se basa en la integración de un módulo de atención de características y un módulo de mejora de características, que trabajan en conjunto para mejorar la capacidad de extracción de

características del modelo. Estos módulos permiten una mayor capacidad para identificar y seguir los objetivos de las embarcaciones, incluso en situaciones donde los objetivos están parcialmente ocultos o el entorno presenta un alto nivel de distracción. La aplicación de AE-YOLOv3 en sistemas de seguimiento de video se destaca como una herramienta potencial para emitir advertencias tempranas de colisión, reduciendo así la probabilidad de accidentes marítimos.

Al igual que se ha visto en publicaciones anteriores, esta algoritmia tiene limitaciones, como los distintos ángulos por los que se obtiene las imágenes y las condiciones ambientales.

Esta publicación ha servido para el desarrollo del algoritmo de detección y así como para entender las limitaciones de los mismos.

2.13. PG-YOLO: A Novel Lightweight Object Detection Method for Edge Devices in Industrial Internet of Things

Este artículo [14] presenta una detección de objetos basado en PG-YOLO que utiliza el internet de las cosas (IoT). Es capaz de procesar una inmensa cantidad de datos y realizar la detección de una forma rápida y precisa.

Se detalla el proceso de desarrollo y validación, en donde se incluye la selección del conjunto de datos relevante, para este caso es SHWD. Se realizan una serie de comparaciones para valorar el rendimiento de PG-YOLO con otros métodos de detección e objetos.

Al igual que metodologías anteriores, cabe destacar todas las limitaciones que hay al usar este tipo de algoritmos.

De este artículo se ha estudiado con detenimiento toda la algoritmia utilizada para la detección de objetos, sobre todo la parte de rendimiento.

2.14. The Aeroplane and Undercarriage Detection Based on Attention Mechanism and Multi-Scale Features Processing

Este estudio [15] detalla un algoritmo de detección para la detección de aviones, pistas y trenes de aterrizaje. Describen mejoras en la precisión promedio (mAP) y en diversas métricas de rendimiento para diferentes tipos de objetos detectados. Por ejemplo, se observa un aumento del 6,18 % en la mAP para pistas y trenes de aterrizaje, así como mejoras en precisión y tasa de recuperación para aviones y trenes de aterrizaje. Sin embargo, también se señala que la capacidad de detección de objetivos pequeños no ha mejorado lo suficiente y que la velocidad de detección ha disminuido, con un aumento en el tiempo de detección en comparación con versiones anteriores de los algoritmos.

Se han encontrado limitaciones a la hora de desarrollo del algoritmo, como por ejemplo, la necesidad de reducir la carga computacional y aumentar la capacidad de detección.

De este estudio cabe destacar la metodología utilizada para comparar los distintos algoritmos. Se ha utilizado una metodología similar para comparar en términos de computación los algoritmos de detección.

2.15. Conclusiones y Tecnologías

Estos proyectos han demostrado la viabilidad y la eficacia de utilizar tecnologías avanzadas como AWS Lambda, SageMaker y YOLO para el desarrollo de sistemas de visión por ordenador aplicados a la generación de topologías de red. AWS Lambda y SageMaker han sido identificados como herramientas clave para optimizar los recursos, ofreciendo escalabilidad y eficiencia en la gestión de la computación en la nube.

La capacidad de AWS Lambda para manejar tareas de computación específicas de forma eficiente, combinada con la potencia de procesamiento de SageMaker para el entrenamiento y despliegue de modelos de machine learning, proporciona una infraestructura sólida que respalda las necesidades avanzadas de procesamiento de nuestro proyecto. Además, la flexibilidad de Google Colab como herramienta alternativa para el entrenamiento de modelos ha permitido superar las

limitaciones de recursos y acceso a hardware especializado.

Los modelos YOLO, por su parte, han demostrado ser extremadamente efectivos para la detección rápida y precisa de objetos, lo que es crucial para la interpretación de dibujos manuales de topologías de red. La implementación de YOLOv8 ha sido particularmente beneficiosa, ofreciendo mejoras en velocidad y precisión, lo que es esencial para la realización de detecciones en tiempo real necesarias para este proyecto. En conjunto, la integración de estas tecnologías avanzadas en nuestro proyecto no solo ha mejorado la eficiencia del proceso de digitalización de topologías, sino que también ha establecido un marco robusto para futuras investigaciones y desarrollos en el campo de la visión por ordenador aplicada a redes de telecomunicaciones.

Capítulo 3. Tecnologías

A continuación se describen las herramientas y tecnologías utilizadas en el desarrollo del proyecto.

3.1. AWS

Amazon Web Services(AWS)[1], es una plataforma de servicios en la nube ofrecida por Amazon. Proporciona una amplia gama de servicios de infraestructura en la nube, como almacenamiento, cómputo, bases de datos, redes y muchas otras herramientas y servicios, que permiten construir y ejecutar aplicaciones de manera eficiente y rentable.

Una de las ventajas clave de AWS es su flexibilidad y escalabilidad. Gracias a la política de pago por uso, se pueden escalar los recursos de manera rápida y sencilla, permitiendo adaptarse a las fluctuaciones de la demanda sin tener sobrecoste.

Se ofrecen una amplia gama de servicios entre los que se incluye almacenamiento de objetos (Amazon S3) y computación en la nube (Amazon Lambda), que usaremos en nuestro proyecto.

3.1.1. Amazon Lambda

AWS Lambda[2] es un servicio de computación basado en eventos y sin servidor que permite ejecutar código para casi cualquier aplicación o servicio de *backend* sin tener que preocuparse por gestionar o aprovisionar servidores.

3.2. GNS3

GNS3 (Graphic Network Simulation o Simulación Gráfica de Redes)[3] es un software de código abierto de simulación de redes que permite diseñar topologías de red complejas. Esta herramienta permite crear redes virtuales completas y emular dispositivos de red, como routers, switches y firewalls, sin necesidad de tener acceso físico a los equipos reales.

[1] https://aws.amazon.com/es/what-is-aws/
[2] https://aws.amazon.com/es/lambda/
[3] https://www.gns3.com/

La Figura 3.1[4] muestra la interfaz de GNS3.

Además, permite la integración con herramientas externas como Wireshark (un analizador de protocolos). Esto hace que se puedan agregar funcionalidades adicionales a las simulaciones.

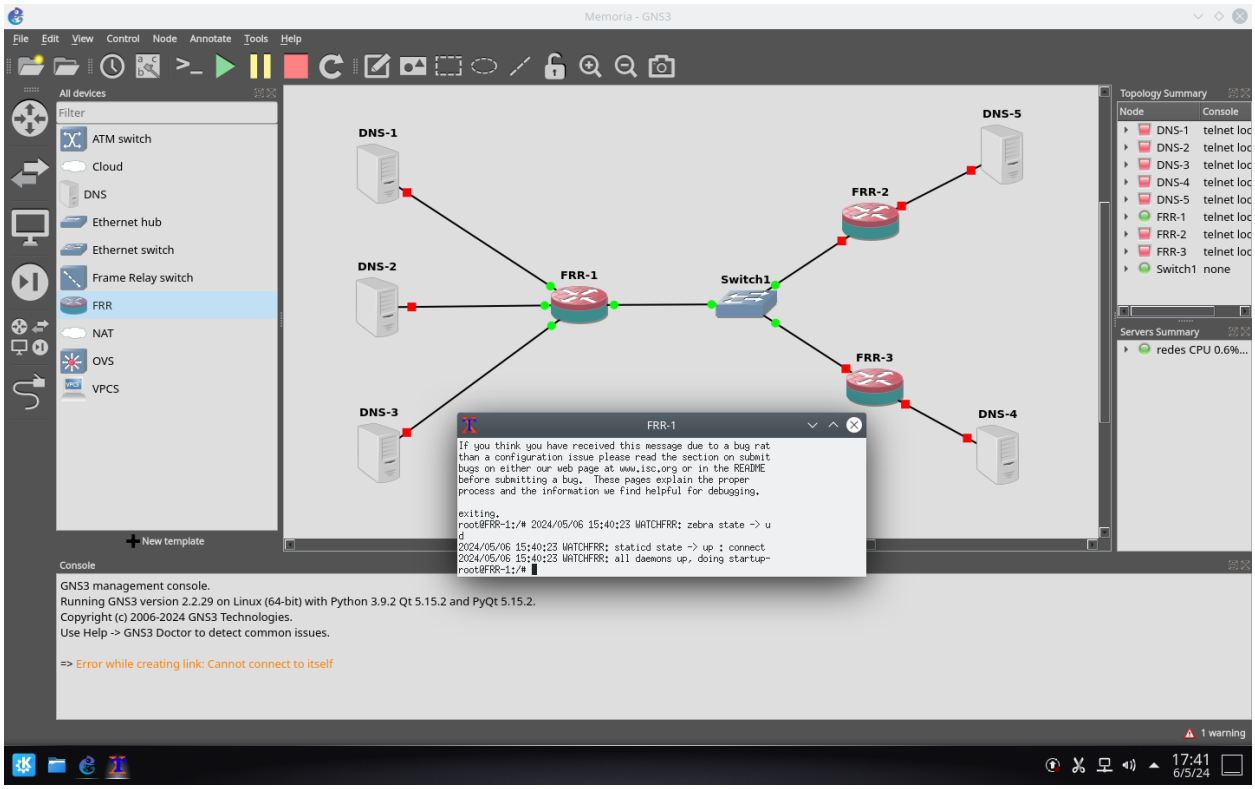


Figura 3.1: Interfaz GNS3

3.3. Python3

Python[5]es un lenguaje de programación interpretado, de alto nivel y de propósito general. Es esencial para el desarrollo de modelos de reconocimiento de objetos. Con bibliotecas como TensorFlow, PyTorch, OpenCV, y Scikit-learn, se pueden crear, entrenar y probar modelos de detección de objetos. Python también se integra bien con otras herramientas, como Google Colab y Roboflow.

[4]Imagen obtenida de la web:https://www.gns3.com/
[5]https://www.python.org/

3.4. Google Colaboratory

Google Colaboratory[6], o Colab, es un servicio gratuito de Google. Funciona como un entorno de desarrollo que utiliza "notebooks" o cuadernos Jupyter en la nube. Este entorno permite escribir y ejecutar código Python en tiempo real. El formato de cuaderno nos da la posibilidad de escribir código, agregar explicaciones en texto e incluir elementos visuales dentro de un solo documento. Además podemos aprovechar las GPU y TPU gratuitas para entrenar y experimentar con los modelos de detección de objetos.

3.5. Roboflow

Roboflow[7] es un software diseñado para simplificar el desarrollo de modelos de reconocimiento de objetos. Además de facilitar la tarea de etiquetar datos y entrenar modelos, Roboflow permite la integración del *dataset* con el código del modelo en Google Colab. Por otro lado, ofrece herramientas visuales, gráficos y elementos interactivos que ayudan a mejorar la calidad y comprensión del modelo. La Figura 3.2[8] muestra la interfaz de Roboflow.

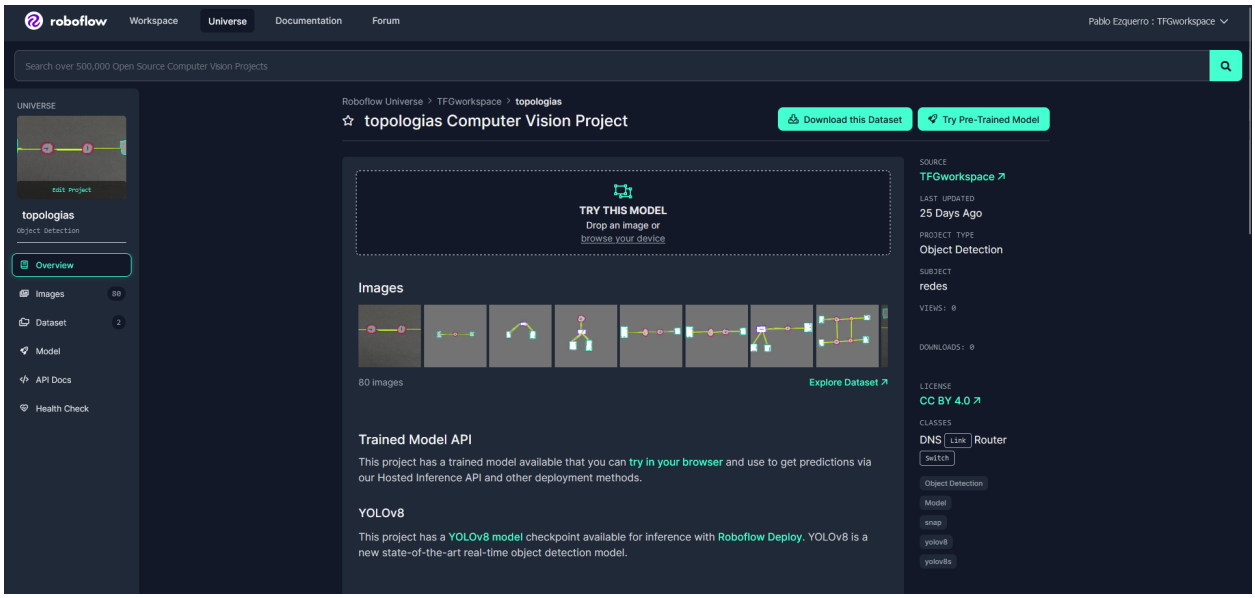


Figura 3.2: Interfaz Roboflow

[6] https://colab.google/
[7] https://roboflow.com/
[8] Imagen obtenida de la web: https://roboflow.com/

3.6. YOLO

YOLO (You Only Look Once)[9] es un algoritmo ampliamente utilizado para la detección y clasificación de objetos mediante el uso de redes neuronales y modelos de aprendizaje profundo. Para su funcionamiento, la red neuronal convolucional divide la imagen en regiones, prediciendo simultáneamente múltiples cuadros delimitadores y las probabilidades de la clase de objeto que delimitan dichos cuadros. Para nuestro proyecto hemos hecho pruebas con YOLOv4 Darknet [16, 17], YOLOv5 [18] y YOLOv8.

También, este modelo nos aporta información importante a la hora de recrear el dibujo en el simulador gráfico. Cada cuadro delimitador tiene cinco valores predichos, coordenadas(x, y), altura, anchura, y precisión.

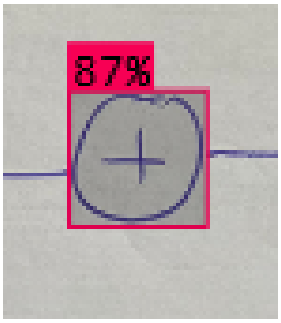


Figura 3.3: Cuadro delimitador predicho


```
{
  "x": 2384,
  "y": 763.5,
  "width": 266,
  "height": 263,
  "confidence": 0.871,
  "class": "Router",
  "class_id": 2,
  "detection_id": "c1436aa9
}
```


Figura 3.4: Información predicha

3.7. Visual Studio Code

Visual Studio Code (VScode)[10] es un editor de código desarrollado por Microsoft. Es una aplicación multiplataforma, gratuita y basada en código abierto que ofrece una gran cantidad de características para facilitar el desarrollo de software. Su interfaz de usuario personalizable y su gran cantidad de extensiones permiten adaptarlo a las necesidades específicas del desarrollador. Además, su integración con sistemas de control de versiones como Git nos ha permitido hacer un seguimiento de los cambios en el proyecto. La interfaz de Visual Code es mostrada en la Figura 3.5[11].

[9] https://pjreddie.com/darknet/yolo/
[10] https://code.visualstudio.com/
[11] Imagen obtenida de la web: https://code.visualstudio.com/

En este proyecto hemos utilizado VScode para convertir la información de las predicciones del modelo, almacenada en un archivo de texto, a un formato JSON que pueda ser importado a GNS3. Hemos escogido esta aplicación debido a la integración con AWS y Github, y a la posibilidad de usar "Jupyter Notebooks" permitiento la portabilidad del código a Google Colab.

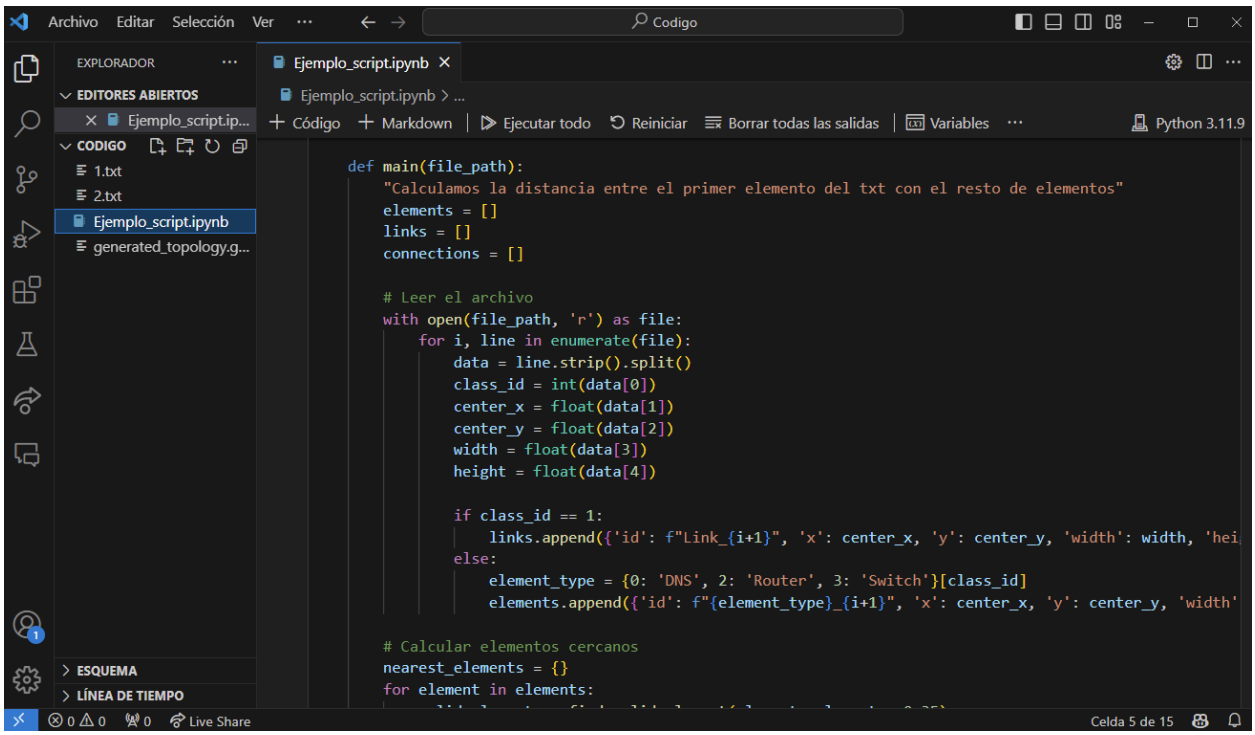


Figura 3.5: Interfaz Visual Studio Code

Capítulo 4. Diseño de Solución

Hasta ahora hemos visto que crear un programa con Inteligencia Artificial es un proceso que lleva tiempo y que necesita muchas herramientas y pruebas. En este capítulo se explicará paso a paso cómo es el trabajo que se ha realizado para crear un programa que pudiera digitalizar dibujos de topologías de red y convertirlos en un formato que GNS3 pudiera importar. Vamos a recorrer todo el proceso, desde el inicio hasta tener el programa funcionando.

4.1. Generacion del Dataset

Uno de los aspectos fundamentales para obtener un modelo de inteligencia artificial preciso es contar con un conjunto de datos de entrenamiento sólido. En nuestro caso, para entrenar un modelo capaz de interpretar diagramas de topologías de red dibujados a mano, necesitamos un dataset que represente fielmente estos dibujos. Los símbolos utilizados en estos diagramas se basan en una leyenda personalizada, como se muestra al inicio (1.2). Esto ha presentado desafíos, ya que no se ha podido aprovechar ningún dataset existente. Por tanto, se ha tenido que construir un conjunto de datos desde cero.

Para facilitar el proceso de anotación y organización de datos, se ha utilizado Roboflow, una plataforma que permite cargar, etiquetar y gestionar imágenes para proyectos de inteligencia artificial. Esta herramienta nos ha proporcionado una interfaz amigable para etiquetar elementos específicos dentro de nuestros dibujos, agilizando la creación de nuestro dataset para el entrenamiento del modelo.

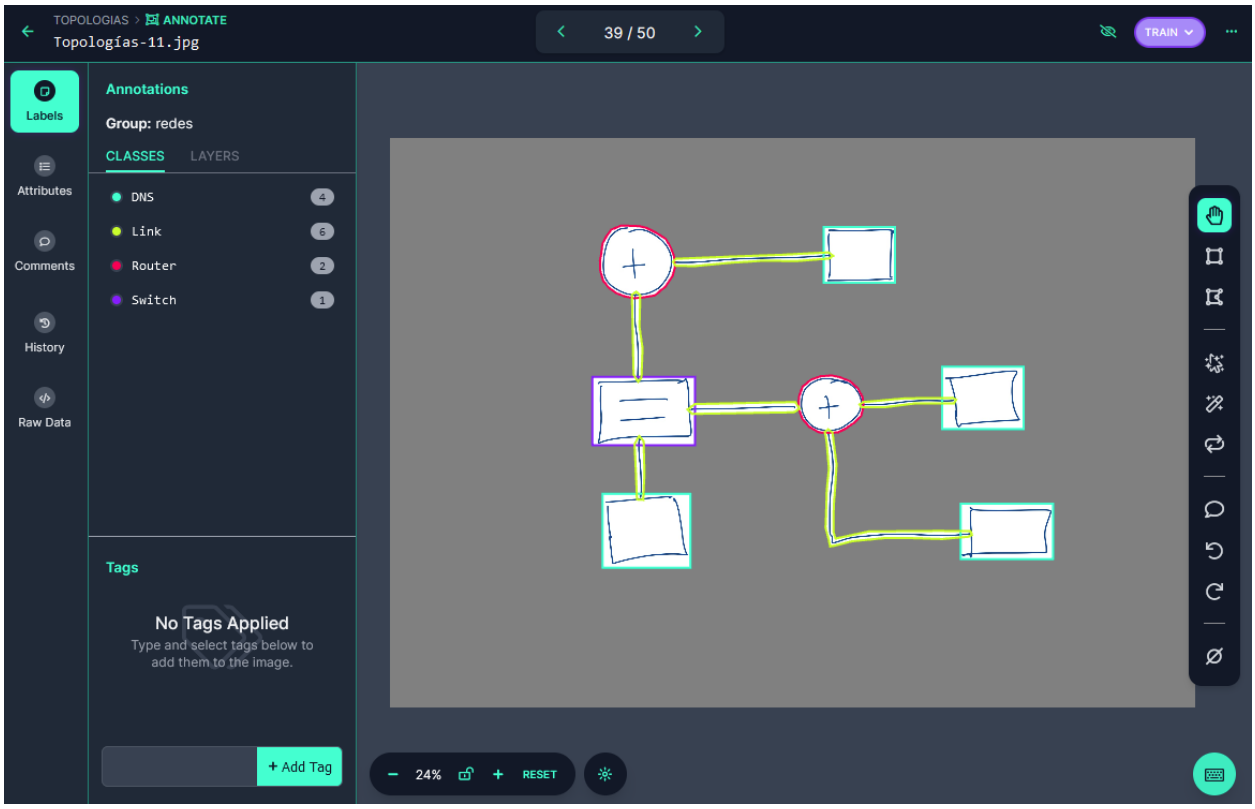


Figura 4.1: Herramienta de etiquetado de imágenes

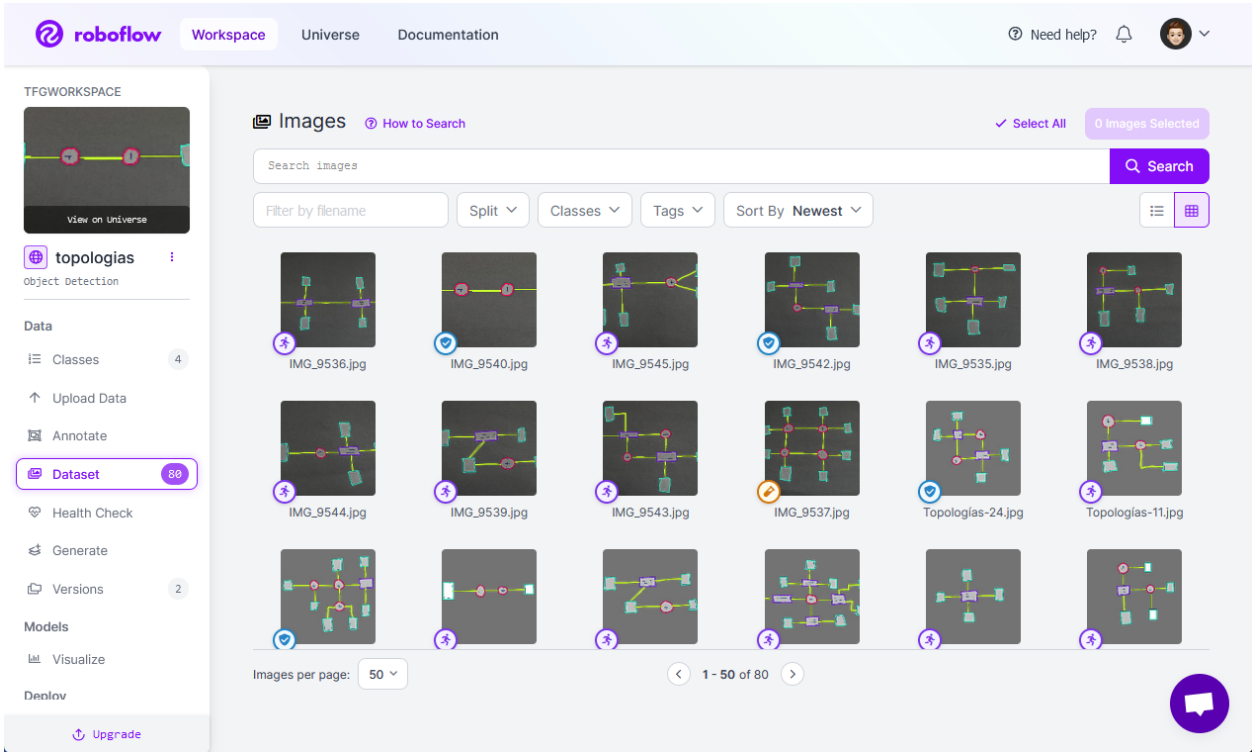


Figura 4.2: Conjunto de datos iniciales etiquetados

Con Roboflow, se ha logrado un conjunto de datos de 136 imágenes, partiendo de 80 dibujos originales que han sido etiquetados manualmente. Sin embargo, dado que este volumen inicial no es suficiente para entrenar adecuadamente el modelo, se han aplicado técnicas de aumento de datos, como rotaciones y otras transformaciones, para generar imágenes adicionales a partir de las 80 originales. También se ha aplicado un preprocesamiento para convertir todas las imágenes a escala de grises, garantizando la uniformidad del dataset.

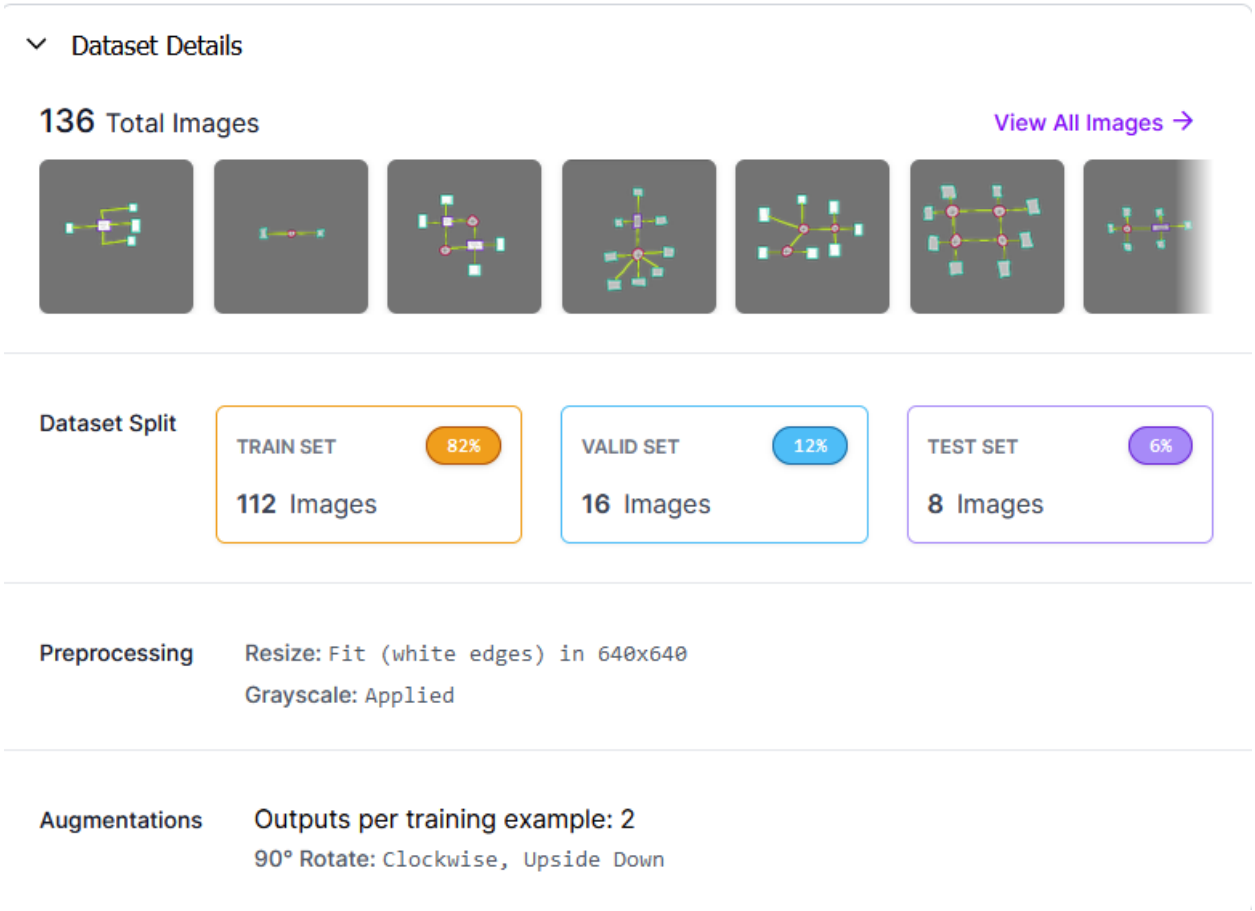


Figura 4.3: Preprocesado y asignacion de datos

Este proceso nos permitió obtener un conjunto de datos consistente, necesario para el entrenamiento del modelo destinado a interpretar dibujos de topologías de red manuscritos.

4.2. Modelos

Una vez que se dispone de un conjunto de datos de entrenamiento adecuado, el siguiente paso es experimentar con diferentes modelos

de aprendizaje automático para encontrar el que ofrezca el mejor rendimiento. Como se mencionó en el capítulo sobre las tecnologías utilizadas (3), Roboflow permite la integración directa del dataset con plataformas de experimentación como Google Colab. Gracias a esta integración, hemos podido realizar pruebas y ajustes en los modelos YOLOv4 Darknet, YOLOv5 y YOLOv8.

Para poder explicar las conclusiones y el modelo que mejor se ha adaptado a nuestro caso de uso, vamos a desarrollar como está compuesta la arquitectura YOLO, y las diferencias entre las versiones que han sido utilizadas.

4.2.1. YOLOv4 Darknet

YOLOv4[1] se divide en tres partes principales: Backbone, Neck, y Head.

- Backbone: Es la base del modelo, encargada de extraer características de la imagen. Usa CSPDarknet53 como componente principal y está preentrenada en ImageNet y es responsable de extraer características clave de las imágenes de entrada. Este Backbone tiene la tarea de identificar patrones, bordes y formas básicas en la imagen, que luego serán utilizadas para detectar los componentes.
- Neck: utiliza PANet, que actúa como un conector entre el Backbone y el Head. Su función es procesar las características extraídas por el Backbone y prepararlas para la detección final.
- Head: Esta es la parte final del modelo, donde se realiza la detección real de objetos. Toma la información procesada por el Backbone y el Neck y determina la ubicación y la clase de los objetos en la imagen. El Head utiliza la estructura de YOLOv3 para hacer las detecciones finales y las clasificaciones. Este segmento es responsable de predecir las clases y los cuadros delimitadores de los objetos.

[1] https://docs.ultralytics.com/es/models/yolov4/

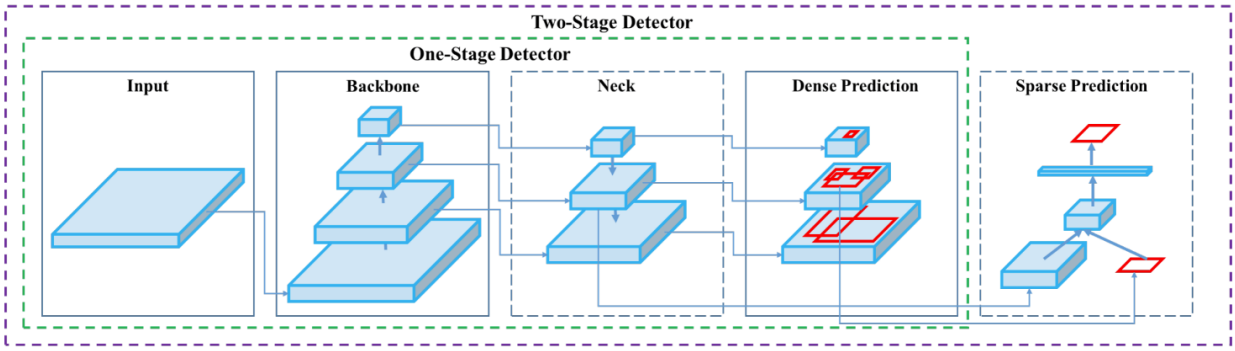


Figura 4.4: Esquema de la arquitectura de YOLOv4

4.2.2. YOLOv5

Tanto YOLOv4 como YOLOv5 [2] comparten el concepto de Backbone, Neck y Head. Sin embargo, en YOLOv5 se introducen algunas optimizaciones y simplificaciones.

- Backbone: YOLOv5 también usa una versión modificada de CSPDarknet53, pero con optimizaciones que simplifican la implementación y mejoran la eficiencia computacional. Esto se traduce en una estructura más ligera y rápida.
- Neck: Es similar pero incluye mejoras como SPPF (Spatial Pyramid Pooling Fast), que ayuda a extraer características a diferentes escalas de manera más eficiente. Se simplifica la estructura CSPDarknet53-PANet, contribuyendo a una implementación más ligera.
- Head: YOLOv5 mantiene un enfoque similar pero con optimizaciones para hacer la estructura más ligera y rápida, reduciendo la latencia.

4.2.3. YOLOv8

YOLOv8 incorpora mejoras significativas tanto en precisión como en velocidad [19], además de la posibilidad de utilizar detección sin anclajes (anchor-free). Los anclajes son cajas delimitadoras que se utilizan para la predicción de los objetos. En otras versiones de YOLO, estas cajas se ubican en lugares predefinidos sobre la imagen y se van ajustando sus medidas según va avanzando el entrenamiento. YOLOv8 predice directamente las coordenadas de las cajas delimitadoras sin depender de anclajes específicos.

[2] https://docs.ultralytics.com/yolov5/tutorials/architecture_description/

- Backbone: En esta versión, además de CSPDarknet53 se utiliza EfficientDet. Ambos aportan una gran capacidad de aprendizaje y eficiencia.
- Neck: YOLOv8, al igual que YOLOv5 utiliza PANet, pero la estructura de YOLOv8 permite una mejor combinación de características, lo que puede mejorar la precisión en detección de objetos de diferentes tamaños.
- Head: Se mantiene el uso de cajas ancladas para predecir objetos de diferentes formas y tamaños, permitiendo tanto detección con anclajes como detección sin anclajes (anchor-free). YOLOv5, por el contrario, utiliza un enfoque tradicional basado en anclajes.

4.3. Estudio comparativo de YOLO e interpretación

Una vez realizado el estudio sobre las diferentes versiones de YOLO, se ha procedido a la experimentación. El primer modelo que se ha entrenado ha sido YOLOv4 con el framework de Darknet.

El proceso de configuración y entrenamiento de YOLOv4 ha sido muy tedioso por varias razones. Debido a que YOLOv4 dejó de recibir actualizaciones oficiales después de 2020, muchas de las herramientas y bibliotecas están obsoletas. Por ejemplo, algunas dependencias clave, como OpenCV y CUDA, han evolucionado a versiones que no son compatibles con el código base de Darknet, lo que ha generado conflictos al intentar compilar el proyecto.

Hemos tenido que modificar muchas variables en el Makefile para ajustar la configuración a nuestro entorno de trabajo, como rutas de archivos, opciones de compilación, configuración de la GPU y alguna variable que no estaba definida. Otro problema importante ha sido el coste computacional. El modelo YOLOv4 es bastante pesado y requiere un alto nivel de recursos para entrenamiento. En el notebook que nos proporcionaba Roboflow, la configuración predeterminada tenía parámetros configurados para ser ejecutados en un entorno con recursos significativos. Sin embargo, al intentar entrenar este modelo en la versión gratuita de Google Colab, nos encontramos con limitaciones severas en la GPU y restricciones de tiempo de ejecución. Debido a estas restricciones, el entrenamiento no solo era lento, sino que a menudo era interrumpido

por restricciones de tiempo o limitaciones de recursos como se puede ver en la figura 4.5.

Pese a haber intentado entrenarlo de la mejor manera posible nos hemos dado cuenta que era mejor idea explorar versiones más recientes de YOLO, como YOLOv5, que se basan en frameworks más recientes y tienen mejor soporte. El cambio a estas versiones más modernas nos permitió beneficiarnos de una comunidad más activa, documentación actualizada y un mejor rendimiento en entornos de nube con limitaciones de recursos.

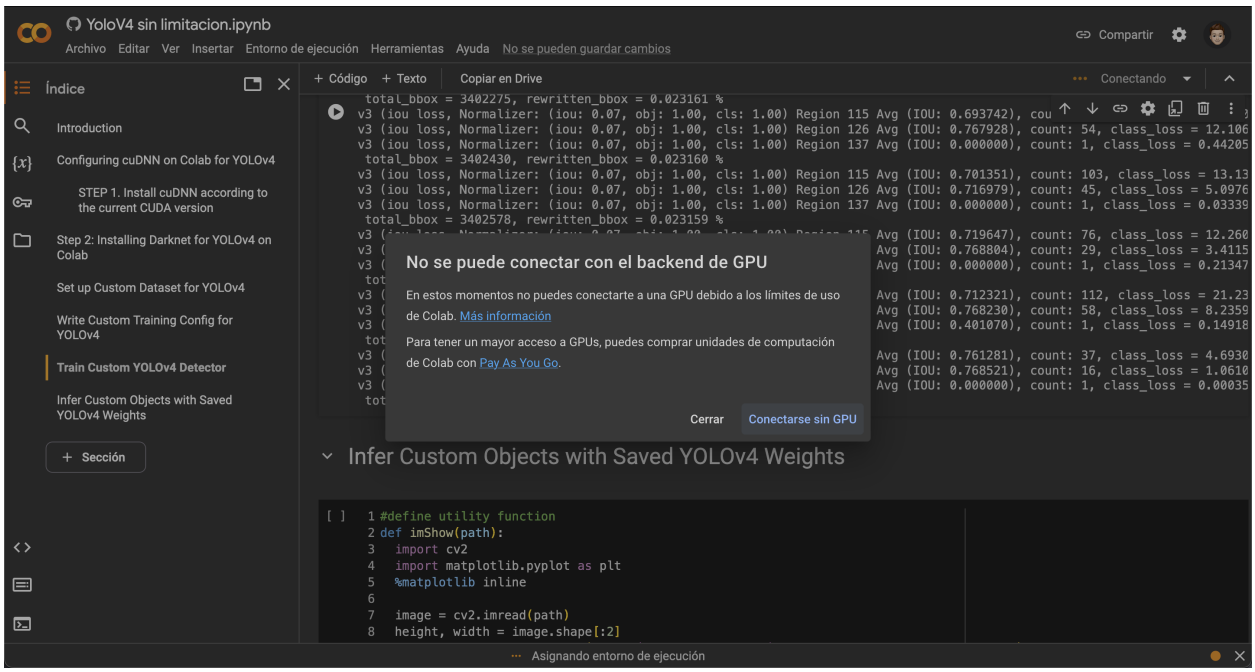


Figura 4.5: Limitación Google Colab en entrenamiento con YOLOv4

El entrenamiento con YOLOv5 utilizando un *dataset* de 136 imágenes ha presentado varios desafíos. Aunque YOLOv5 es más moderno y flexible que YOLOv4, tuvimos que ajustar el modelo para adaptarlo a nuestro caso particular.

El notebook proporcionado por Roboflow viene con una configuración predeterminada que incluye un tamaño de imagen de 416 píxeles, un batch de 16 y 100 épocas. Estas configuraciones suelen ser adecuadas para *datasets* medianos o grandes, pero para nuestro pequeño *dataset* de 136 imágenes necesitamos ajustes adicionales para optimizar el entrenamiento y evitar el sobreajuste. Nuestra solución ha sido reducir el tamaño del *batch* a 8 y aumentar el número de épocas a 250, permitiendo más iteraciones sobre el mismo conjunto de datos.

Usar un tamaño de *batch* más pequeño, como 8, reduce la cantidad de memoria necesaria para procesar cada paso de entrenamiento, lo que ha sido beneficioso en nuestro caso, dado que tenemos recursos limitados en Google Colab. Además, esta configuración permite al modelo tener más iteraciones sobre los datos, lo cual puede ayudar a mejorar el aprendizaje.

Reducir el *batch* a 8 ha sido un acierto, ya que permite al modelo entrenarse de manera más eficiente, aunque con un tiempo de entrenamiento más prolongado. Sin embargo, aumentar en exceso el número de épocas incrementa el riesgo de sobreajuste. Para mitigar este riesgo, hemos implementado varias estrategias, como el aumento de datos, que aumenta la diversidad del *dataset*. Esta técnica fue aplicada y se describe en la sección de generación del *dataset* 4.1. Podemos ver las gráficas de evolución del entrenamiento en la figura 4.6

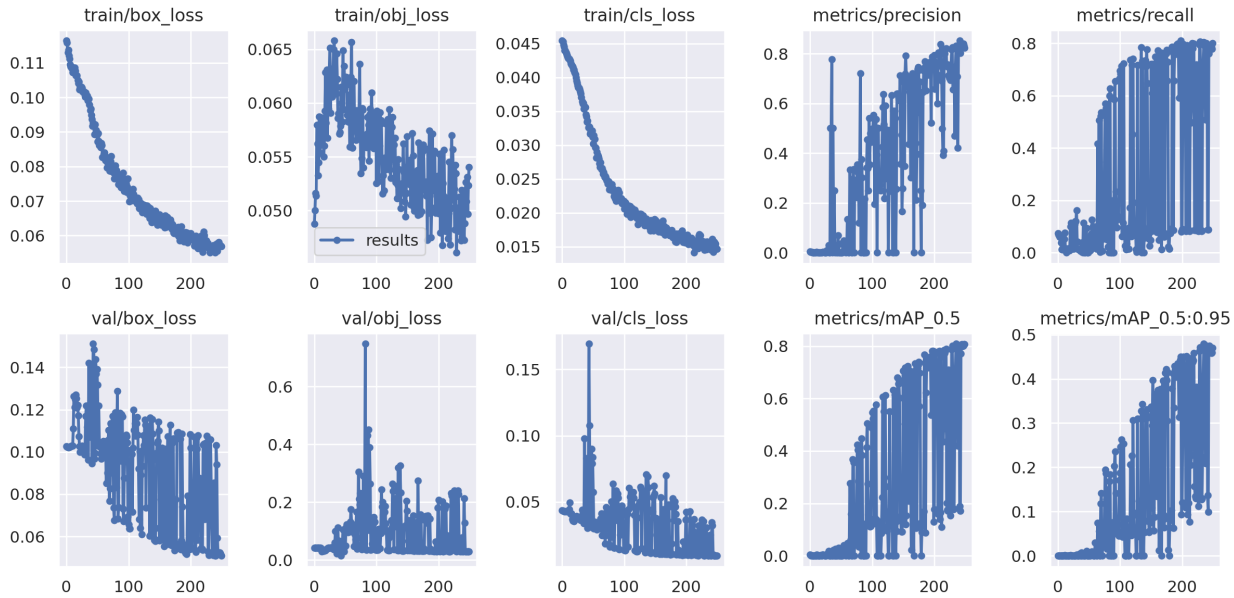


Figura 4.6: Gráficos de evolución del entrenamiento con YOLOv5

Después de probar diferentes configuraciones de épocas, concluimos que 250 era una buena opción basándonos en los resultados obtenidos como se ven en la figura 4.7. Estos resultados confirman que los ajustes realizados han permitido optimizar el entrenamiento sin caer en el sobreajuste, proporcionando una base sólida para futuras mejoras. Sin embargo, los datos no cumplen nuestras expectativas, por lo que hemos decidido probar con YOLOv8 para ver si podemos obtener resultados más satisfactorios.

```
      Epoch    GPU_mem   box_loss   obj_loss   cls_loss  Instances       Size
    249/249      1.02G    0.05687    0.05403    0.01467        190        416: 100% 14/14 [00:02<00:00,  5.12it/s]
                 Class     Images  Instances          P          R      mAP50   mAP50-95: 100% 1/1 [00:00<00:00,  4.43it/s]
                   all         16        254      0.824        0.8      0.807       0.47

250 epochs completed in 0.283 hours.
Optimizer stripped from runs/train/yolov5s_results/weights/last.pt, 14.8MB
Optimizer stripped from runs/train/yolov5s_results/weights/best.pt, 14.8MB

Validating runs/train/yolov5s_results/weights/best.pt...
Fusing layers...
custom_YOLOv5s summary: 182 layers, 7254609 parameters, 0 gradients
                 Class     Images  Instances          P          R      mAP50   mAP50-95: 100% 1/1 [00:00<00:00,  4.38it/s]
                   all         16        254      0.835      0.805      0.809       0.48
                   DNS         16         87      0.687       0.69       0.62      0.426
                  Link         16        120      0.936        0.9      0.934        0.3
                Router         16         28      0.933          1      0.969      0.736
                Switch         16         19      0.782      0.632      0.712      0.458
Results saved to runs/train/yolov5s_results
CPU times: user 12.1 s, sys: 1.31 s, total: 13.4 s
Wall time: 17min 40s
```

Figura 4.7: Resultados del entrenamiento con YOLOv5

Complementando las métricas de los resultados obtenidos, en la figura 4.8 podemos ver ejemplos de las cajas delimitadoras predichas con YOLOv5 sobre algunas topologías.

Figura 4.8: Cajas delimitadoras predichas con YOLOv5

Para terminar hemos querido comprobar si existe mucha diferencia entre YOLOv5 y YOLOv8, y si era suficiente para justificar el cambio.

En YOLOv8, hemos hecho ajustes muy similares a YOLOv5, modificando el número de épocas y el tamaño del *batch* para optimizar el rendimiento y evitar el sobreajuste. El valor predeterminado en el *notebook* que proporciona Roboflow para YOLOv8 es 16 para el tamaño del *batch* y 25 para el número de épocas. Dado que tenemos un *dataset* reducido y hemos trabajado con recursos limitados en Google Colab, ajustamos el tamaño del batch a 4 y aumentamos el número de épocas a 50.

El batch más pequeño reduce el uso de memoria y permite más iteraciones, aunque puede aumentar el riesgo de sobreajuste porque el modelo tiene más tiempo para memorizar el dataset. Para mitigar este riesgo, YOLOv8 tiene una característica que los anteriores no tenían. *Early stopping*, que detiene el entrenamiento cuando ya no hay mejoras significativas, evitando el sobreajuste. Podemos ver las gráficas de evolución del entrenamiento en la figura **??**

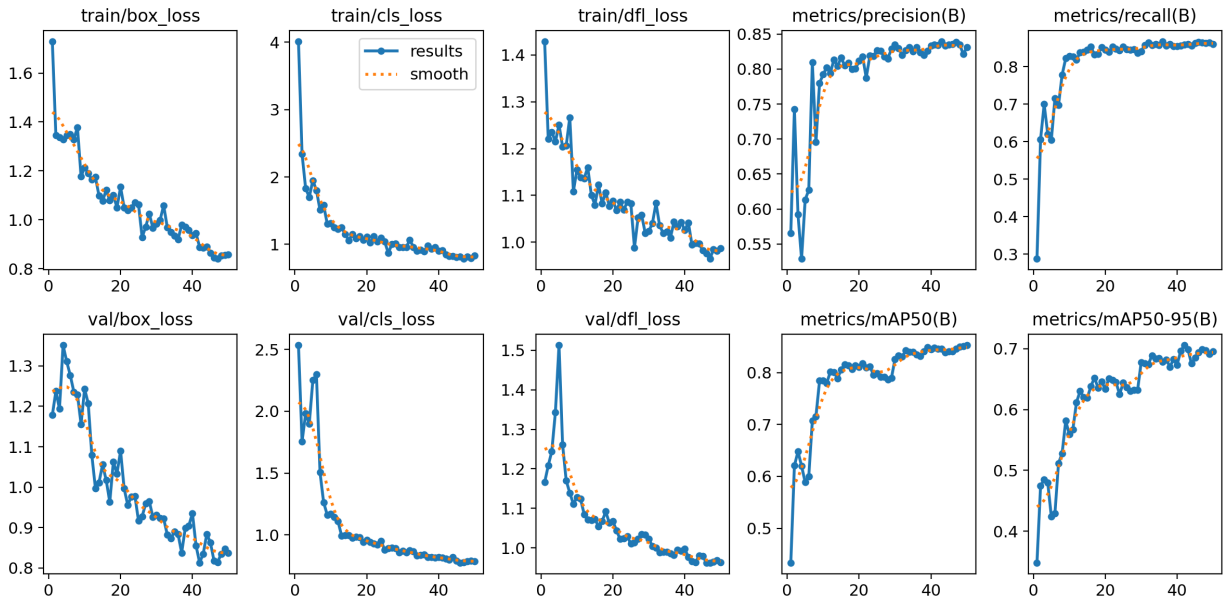


Figura 4.9: Gráficos de evolución del entrenamiento con YOLOv8

Los resultados obtenidos con YOLOv8, como se ven en la figura 4.10, son muy positivos considerando el tamaño limitado de nuestro *dataset*. Estos resultados sugieren que el modelo tiene un buen rendimiento general y que estamos en el camino correcto. Con un *dataset* más grande podríamos acercarnos a un modelo casi perfecto para nuestro caso de uso, con detecciones más precisas y consistentes.

```
      Epoch    GPU_mem   box_loss   cls_loss   dfl_loss  Instances       Size
      50/50      1.99G     0.8572     0.8331     0.9872         38        800: 100% 28/28 [00:04<00:00,  6.46it/s]
                 Class     Images  Instances      Box(P          R      mAP50  mAP50-95): 100% 2/2 [00:00<00:00,  7.63it/s]
                   all         16        254      0.831      0.859      0.853      0.695

50 epochs completed in 0.135 hours.
Optimizer stripped from runs/detect/train/weights/last.pt, 22.6MB
Optimizer stripped from runs/detect/train/weights/best.pt, 22.6MB

Validating runs/detect/train/weights/best.pt...
Ultralytics YOLOv8.0.196 🚀 Python-3.10.12 torch-2.2.1+cu121 CUDA:0 (Tesla T4, 15102MiB)
Model summary (fused): 168 layers, 11127132 parameters, 0 gradients, 28.4 GFLOPs
                 Class     Images  Instances      Box(P          R      mAP50  mAP50-95): 100% 2/2 [00:00<00:00,  2.93it/s]
                   all         16        254      0.837      0.858      0.847      0.706
                   DNS         16         87      0.686       0.69      0.649       0.55
                  Link         16        120      0.886      0.933      0.975      0.653
                Router         16         28      0.981          1      0.995      0.936
                Switch         16         19      0.794       0.81      0.767      0.686
Speed: 2.8ms preprocess, 11.0ms inference, 0.0ms loss, 3.3ms postprocess per image
Results saved to runs/detect/train
💡 Learn more at https://docs.ultralytics.com/modes/train
```

Figura 4.10: Resultados del entrenamiento con YOLOv8

Complementando las métricas de los resultados obtenidos, en la figura 4.11 podemos ver ejemplos de las cajas delimitadoras predichas con YOLOv8 sobre algunas topologías.

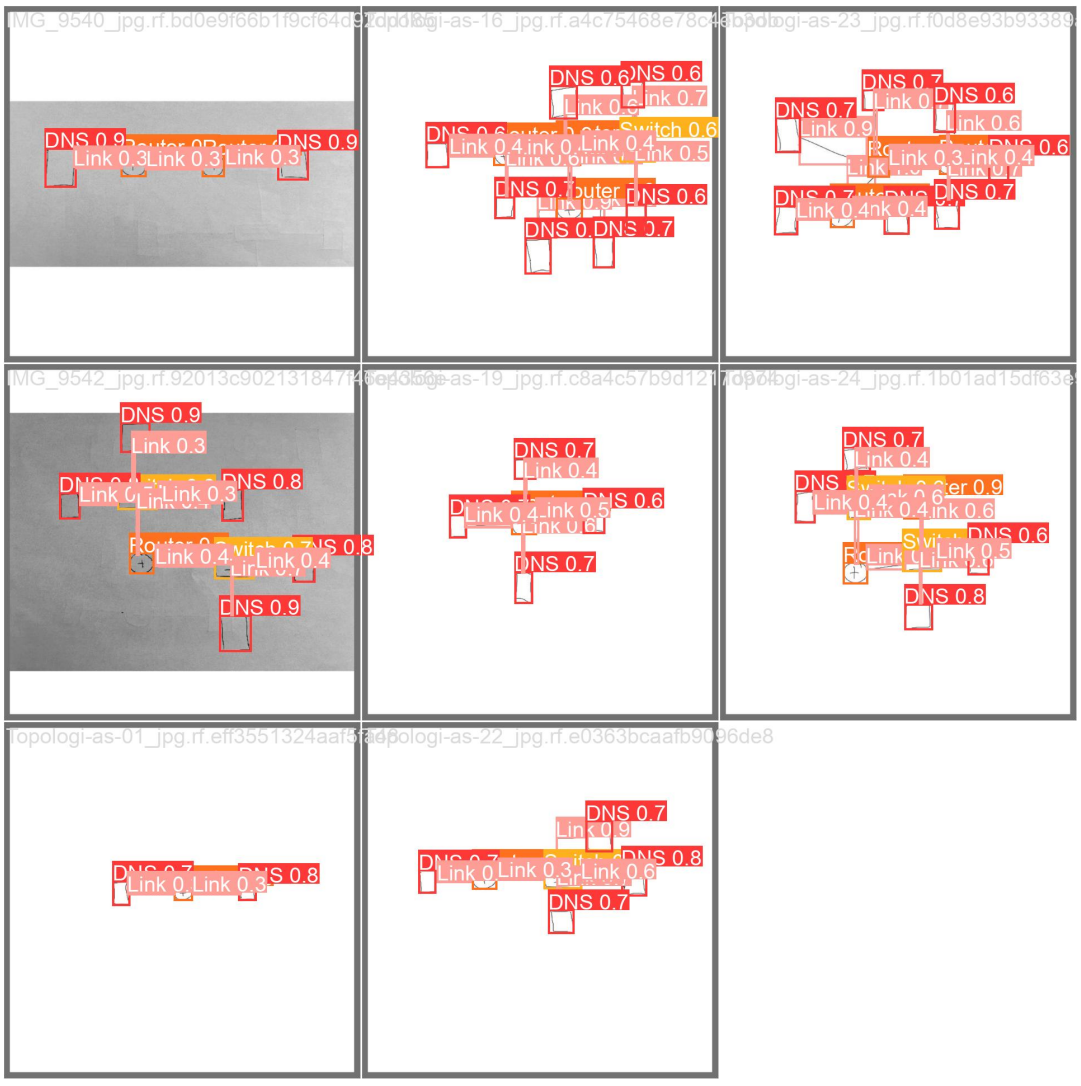


Figura 4.11: Cajas delimitadoras predichas con YOLOv8

Comparando los resultados de YOLOv8 4.10 y YOLOv5 4.7, podemos sacar conclusiones que nos demuestran que merece la pena utilizar YOLOv8 frente a YOLOv5. En términos de rendimiento global, muestra una ligera ventaja en la mayoría de las métricas:

- Precisión (P): métrica que evalúa la exactitud de las detecciones correctas con respecto al total de detecciones realizadas. Ambos modelos tienen casi la misma precisión (0.836 para YOLOv8 y 0.835 para YOLOv5), lo que indica una tasa similar de detecciones correctas respecto a las detecciones totales.
- Recall (R): métrica que evalúa la capacidad del modelo para detectar todas las instancias verdaderas de los objetos, calculándose como la proporción de verdaderos positivos sobre el total de objetos. YOLOv8 tiene un recall más alto (0.867), lo que sugiere que detecta más instancias verdaderas en comparación con YOLOv5 (0.805).
- mAP50: métrica que evalúa la precisión del modelo en la detección de objetos donde la intersección entre la predicción y el objeto real es al menos el 50 %. YOLOv8 tiene un mayor mAP50 (0.852) que YOLOv5 (0.809), lo que indica que es más preciso en la detección de objetos con un umbral de IoU del 50 %.
- mAP50-95: métrica que evalúa la precisión del modelo en la detección de objetos con múltiples umbrales de intersección, desde 50 % hasta 95 %. La mayor diferencia se observa en mAP50-95, con YOLOv8 obteniendo 0.689 frente a 0.48 para YOLOv5. Esto significa que YOLOv8 es significativamente mejor a umbrales de precisión más altos.

4.4. Interpretación de la salida

El proceso más técnico del proyecto se centró en la codificación de un script capaz de interpretar la salida generada por el modelo y transformarla en un archivo con formato `.gns3`.

4.4.1. Formato del archivo `.gns3`

El formato del código fuente de un proyecto en GNS3 es bastante simple. Se trata de un archivo JSON que contiene la configuración general del proyecto, así como información específica de cada nodo y las conexiones entre ellos.

Configuración general

```
# Definir datos generales del proyecto
project_data = {
    "auto_close": True,
    "auto_open": False,
    "auto_start": False,
    "drawing_grid_size": 25,
    "grid_size": 75,
    "name": "1",
    "project_id": str(uuid.uuid4()),
    "revision": 9,
    "scene_height": 1000,
    "scene_width": 2000,
    "show_grid": False,
    "show_interface_labels": False,
    "show_layers": False,
    "snap_to_grid": False,
    "supplier": None,
    "topology": {
        "computes": [],
        "drawings": [],
        "links": [],
        "nodes": []
    },
    "type": "topology",
    "variables": None,
    "version": "2.2.29",
    "zoom": 100
}
```

Figura 4.12: Configuración general de `.gns3`

- **Configuración Global:**
  - auto_close: indica si el proyecto se cerrará automáticamente al salir.
  - auto_open: determina si el proyecto se abrirá automáticamente al iniciar GNS3.
  - auto_start: define si los dispositivos en el proyecto se iniciarán automáticamente.
- **Configuración de la Cuadrícula:**
  - drawing_grid_size: tamaño de la cuadrícula para dibujar elementos en la interfaz de GNS3.
  - grid_size: tamaño de la cuadrícula para alinear elementos.
- **Información General del Proyecto:**
  - name: nombre del proyecto.
  - project_id: identificador único del proyecto.
  - revision: la revisión o versión del proyecto.

  - scene_height y scene_width: dimensiones de la vista de la topología en GNS3.
  - show_grid: indica si la cuadrícula se mostrará en la vista de la topología.
  - show_interface_labels: determina si se mostrarán las etiquetas de interfaz.
  - show_layers: define si se mostrarán las capas.
  - snap_to_grid: establece si los elementos se alinearán automáticamente con la cuadrícula.

- **Proveedor:**
  - supplier: contiene información sobre el proveedor o es null si no se especifica.

- **Topología de Red:** esta sección tiene la informacion de los componentes de la red y es específica para cada proyecto

- **Información del Proyecto:**
  - type: indica que este es un proyecto de topología.
  - variables: puede contener variables definidas para el proyecto, siendo null en este caso.
  - version: la versión de GNS3 utilizada en el proyecto.
  - zoom: el nivel de zoom predeterminado para la vista de la topología.

Configuración Específica

Este fragmento de código proporciona la configuración de un nodo DNS.


```
{
    "compute_id": "local",
    "console": 5000,
    "console_auto_start": True,
    "console_type": "telnet",
    "custom_adapters": [],
    "first_port_name": None,
    "height": 70,
    "label": {
        "rotation": 0,
        "style": "font-family: TypeWriter;font-size: 10.0;font-weight: bold;fill: #000000;fill-opacity: 1.0;",
        "text": "DNS-1",
        "x": -2,
        "y": -25
    },
    "locked": False,
    "name": "DNS-1",
    "node_id": "2610c969-3da7-49d4-92c7-a6286d80faef",
    "node_type": "docker",
    "port_name_format": "Ethernet{0}",
    "port_segment_size": 0,
    "properties": {
        "adapters": 1,
        "aux": 5001,
        "console_http_path": "/",
        "console_http_port": 80,
        "console_resolution": "800x600",
        "container_id": "cd6ac2e1f58382a0e307ac82799340254b833e710d04c6a578e30d151385759a",
        "environment": "MAC=0\nTYPE=1",
        "extra_hosts": None,
        "extra_volumes": [
            "/etc",
            "/var/cache/bind"
        ],
        "image": "jcfabero/server:latest",
        "start_command": "/bin/bash",
        "usage": "Configure bind9 in /etc/bind\nNetwork can be configured with DHCP via %dhclient -v eth0%"
    },
    "symbol": ":/symbols/classic/server.svg",
    "template_id": "fd0fc58f-b0f9-4516-b431-b297339d6b4f",
    "width": 49,
    "x": -249,
    "y": -38,
    "z": 1
}
```


Figura 4.13: Configuración nodo DNS de `.gns3`

- compute_id: identifica la computadora donde se ejecuta el nodo. En este caso, está configurado como "local", lo que indica que se está ejecutando en la misma máquina donde se realiza la simulación.
- console: especifica el puerto utilizado para la consola del nodo. En este caso, se establece en el puerto 5000.

- console_auto_start: determina si la consola del nodo se iniciará automáticamente. Está configurado como Verdadero (True), lo que indica que la consola se iniciará automáticamente.
- console_type: indica el tipo de consola utilizado para el nodo. Aquí, se emplea telnet.
- custom_adapters: representa una lista de adaptadores personalizados, que en este caso está vacía, indicando que no se han definido adaptadores personalizados.
- first_port_name: se refiere a la personalización del nombre de los puertos en algunos nodos.
- height: especifica la altura del nodo en la interfaz de simulación, medida en unidades de la interfaz.
- label: contiene la configuración de la etiqueta del nodo, que incluye el texto de la etiqueta, su posición, estilo y rotación.
- locked: indica si el nodo está bloqueado. En este caso, está configurado como Falso (False), lo que significa que el nodo no está bloqueado.
- name: es el nombre del nodo, que aquí se denomina ”DNS-1”.
- node_id: Proporciona un identificador único para el nodo.
- node_type: Indica el tipo de nodo, que en este caso es de tipo ”docker”.
- port_name_format: Define el formato del nombre de los puertos del nodo. Aquí, los puertos se nombran como “Ethernet0”.
- port_segment_size: define la cantidad de interfaces que hay en el grupo principal.
- properties: Contiene propiedades específicas del nodo, como el número de adaptadores, puerto auxiliar, comando de inicio, imagen de Docker utilizada, entre otros.
- symbol: Es la ruta del símbolo utilizado para representar el nodo en la interfaz de simulación.
- template_id: Proporciona el identificador del template utilizado para el nodo.

- width: Define el ancho del nodo en la interfaz de simulación, medido en unidades de la interfaz.
- x, y, z: Son las coordenadas del nodo en la interfaz de simulación, que determinan su posición en la interfaz.

4.4.2. *Script* de generación del archivo GNS3

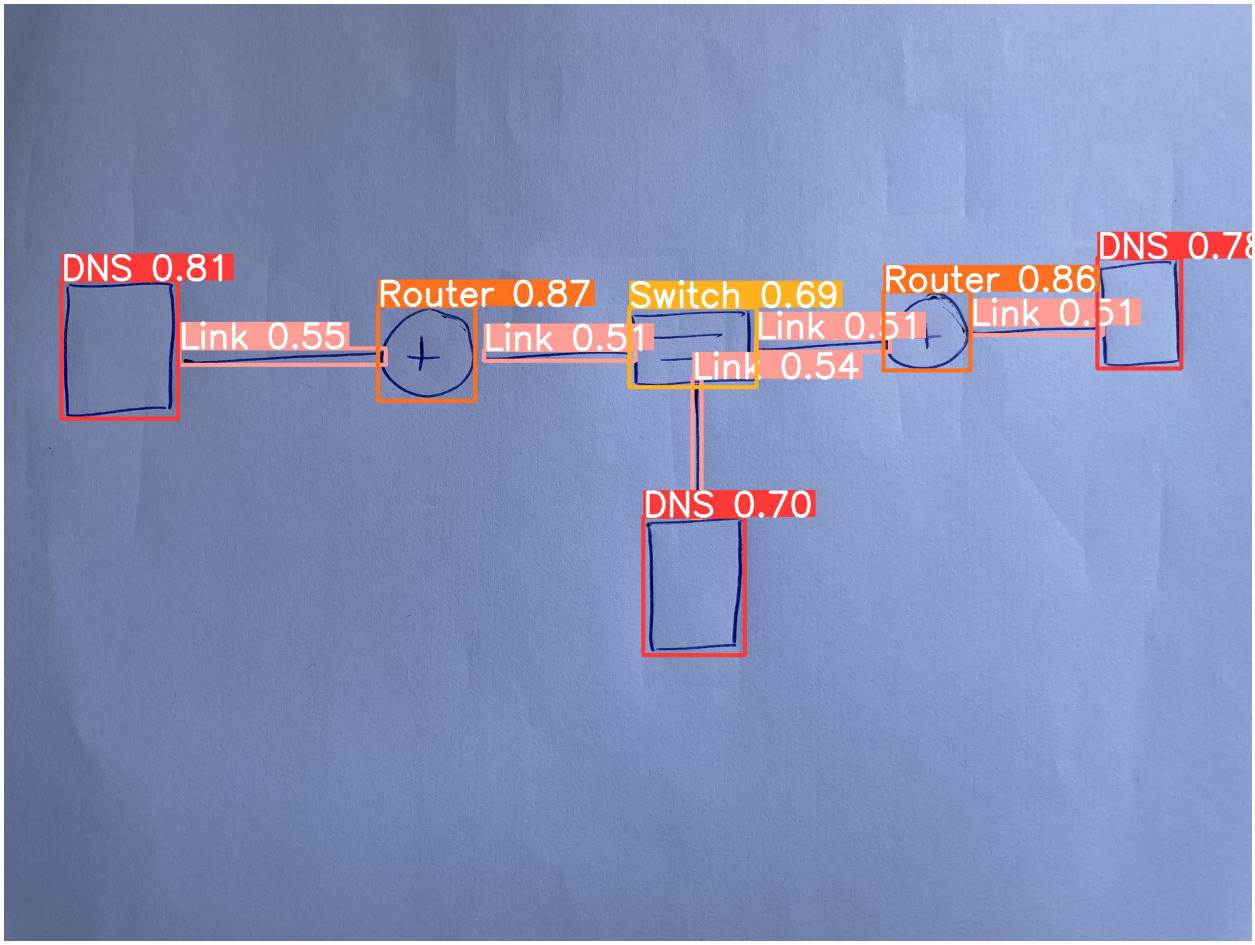


Figura 4.14: Dibujo sobre el que vamos a ver los resultados del Script

Una vez que hemos comprendido el formato requerido para el archivo final, hemos examinado el archivo generado por el modelo. Tanto en YOLOv5 como en YOLOv8, durante el proceso de inferencia, existe una opción que permite guardar las predicciones en un archivo de texto, activada mediante el flag *save_txt=True*. Este archivo, como se ve en la figura 4.15 contiene información crucial que simplifica los cálculos necesarios para generar el archivo en formato `.gns3`.

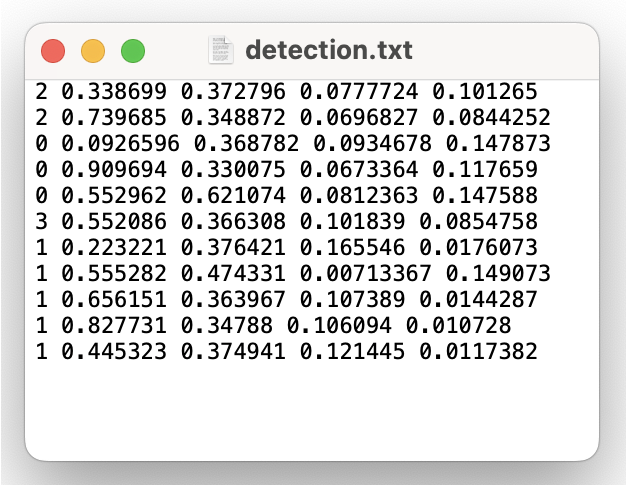


Figura 4.15: Inferencia del modelo en formato `.txt`

Cada línea de este archivo representa un componente de la red, y el orden de los valores en cada línea es el siguiente: tipo de componente, coordenada x, coordenada y, ancho y altura. Los tipos de componentes pueden ser: 0 para DNS, 1 para Enlace, 2 para Router y 3 para Switch.

En primer lugar, leemos las líneas del archivo y almacenamos los componentes en dos listas de diccionarios: una para guardar los Enlaces y su información, y otra para guardar los nodos y su información.

El principal desafío al que nos enfrentamos es determinar qué elementos están conectados entre sí según sus coordenadas. Inicialmente, hemos desarrollado una estrategia simple como punto de partida, sobre la cual realizar mejoras incrementales. En esta primera estrategia, consideramos que cada nodo solo puede establecer conexión con otro nodo si son los más cercanos entre sí y comparten el Enlace más cercano.

Primero, almacenamos el nodo más cercano para cada nodo de la red en la estructura *connections_to_elements*. Esta lista de diccionarios almacena para cada identificador de nodo el nodo más cercano y la distancia a él. Luego, para cada nodo, buscamos el enlace más cercano y almacenamos en la variable *connections_to_Links* el identificador del enlace y la distancia a él.

Finalmente, aplicamos la estrategia mencionada anteriormente: dos nodos están conectados solo si comparten el enlace más cercano y son los elementos más cercanos entre sí. Verificamos si la conexión guardada en *connections_to_elements* para cada nodo es igual a la del otro nodo y luego comprobamos si ambos nodos tienen el mismo ID de enlace en *connections_to_Links*. Si ambas condiciones se cumplen, la conexión es

válida.

Aunque esta estrategia proporciona una base, la estrategia es errónea, ya que puede perder muchas conexiones válidas. En el caso del ejemplo que estamos analizando, perdemos todas las conexiones, ya que ninguna pareja de nodos cumple con las condiciones de la estrategia.

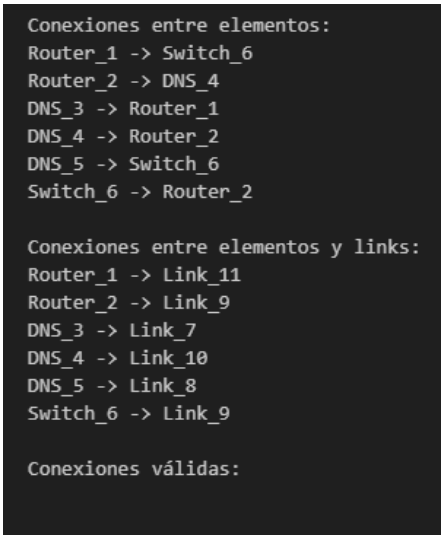


Figura 4.16: Resultado primera estrategia

Tras revisar los resultados obtenidos en la figura 4.16, hemos llegado a la conclusión de que depender únicamente del elemento más cercano no proporciona una representación completa de las conexiones entre elementos. Como respuesta a esta limitación, hemos desarrollado una nueva estrategia que emplea dos umbrales distintos: uno para los Enlaces y otro para los nodos. Este enfoque nos permite almacenar todos los nodos y Enlaces que se encuentren a una distancia menor que el umbral especificado. Además, hemos realizado modificaciones en la estructura del código para mejorar su mantenibilidad y escalabilidad.

El nuevo código incluye una función que, dado un nodo, una lista de Enlaces o de nodos, y un umbral, devuelve los elementos de la lista que cumplen con el umbral de distancia en relación al nodo dado, como se muestra en el diagrama de la figura 4.17. Además, hemos creado una función que, al recibir las listas de elementos cercanos y Enlaces cercanos, aplica una estrategia para determinar las conexiones válidas. Esta estrategia sigue un enfoque similar al anterior, pero sin la necesidad de verificar si un nodo está presente en las conexiones del otro nodo. Esto se debe a que ambas conexiones se han encontrado bajo el mismo umbral. En su lugar, se verifica que ambos nodos tengan un Enlace en común,

como se ve en la figura 4.18. Este Enlace se encuentra utilizando un umbral inferior aplicado sobre los nodos, ya que debe estar en medio de ellos.

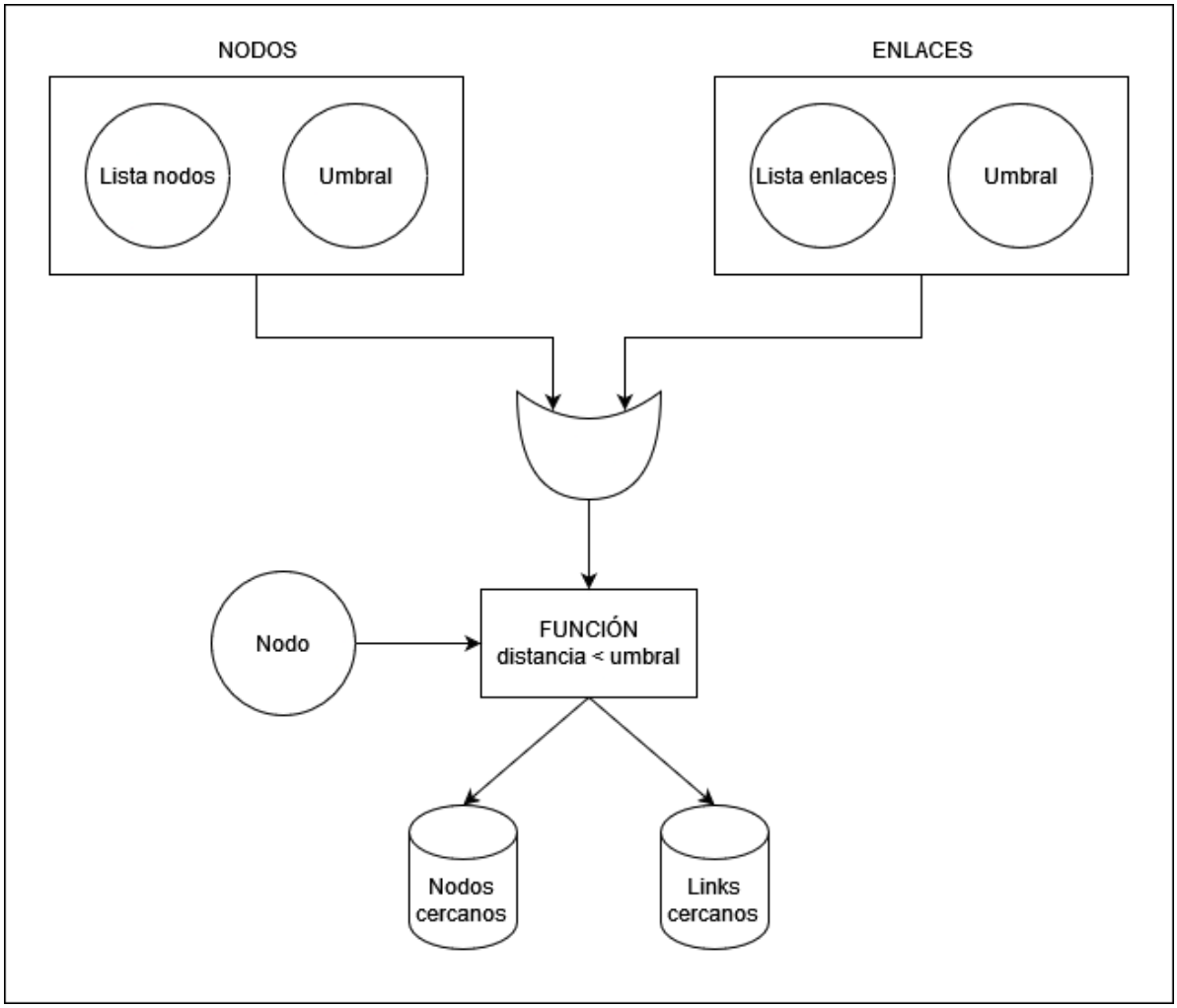


Figura 4.17: Diagrama del flujo de datos para encontrar elementos cercanos a un nodo.

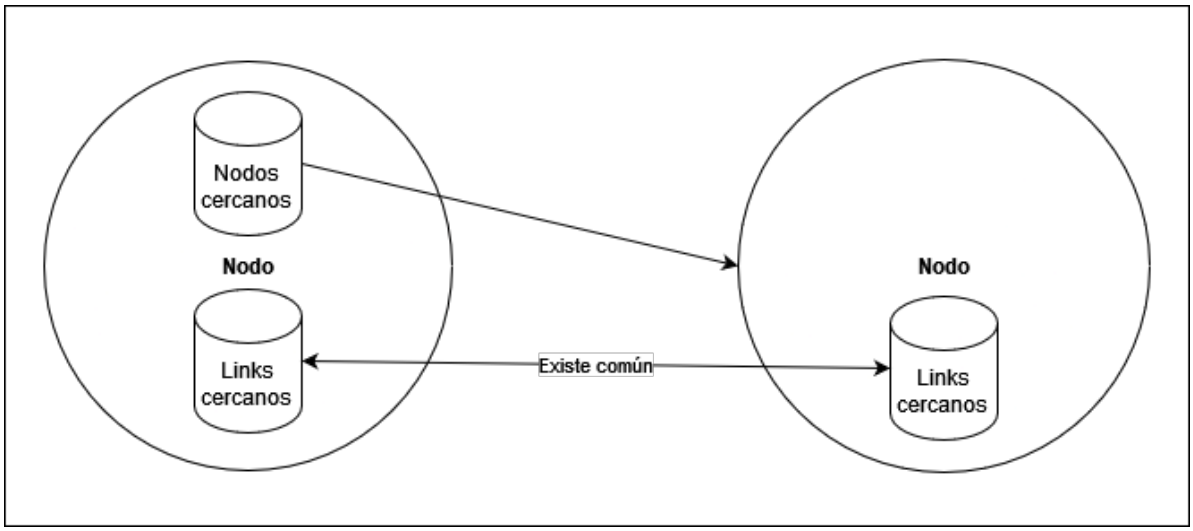


Figura 4.18: Diagrama de validación de conexiones según la estrategia propuesta.

Los resultados obtenidos con esta estrategia son significativamente mejores, como se observa en al figura 4.19. En el caso que estamos mostrando como ejemplo, son perfectos. Además, esta estrategia permite ajustar los umbrales para ser más precisos.

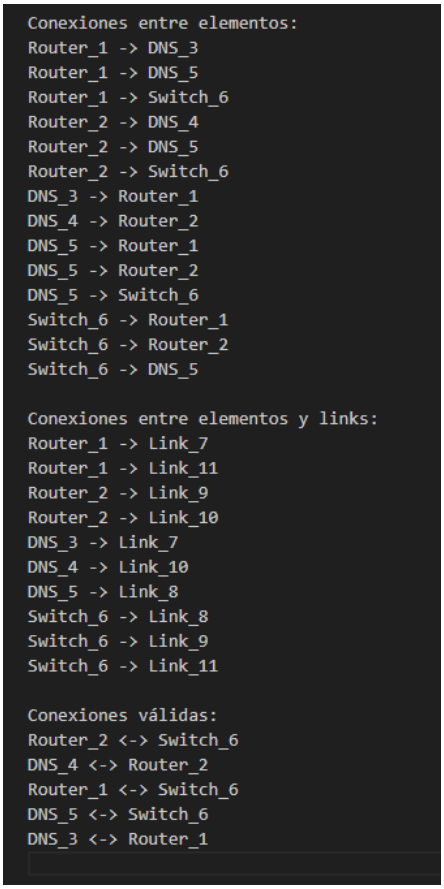


Figura 4.19: Resultado segunda estrategia

A partir de esta estrategia, podemos crear el archivo final necesario para importar el proyecto en la herramienta GNS3. Este proceso consta de dos partes distintas:

La primera parte se centra en establecer la configuración general, que es común a todos los proyectos y abarca información detallada en el apartado 4.4.1.

La segunda parte consiste en una función que toma el identificador del nodo y su información, y devuelve un valor con formato *.json* que contiene toda la configuración específica del nodo, tal como se describe en el apartado 4.4.1.

Finalmente los valores devueltos por la funcion se añaden a la parte correspondiente de la variable de las configuraciones generales y se crea el *.json*.

4.5. Despliegue en AWS

La implementación de soluciones tecnológicas en la nube ha revolucionado la forma en que las organizaciones escalan y gestionan sus infraestructuras de TI. AWS, siendo líder en soluciones de *cloud computing*, ofrece un conjunto robusto de servicios que permiten a los desarrolladores y a las empresas implementar aplicaciones de manera eficiente y segura. El despliegue de una función Lambda en AWS no es una excepción a esta revolución y constituye un ejemplo clave de cómo se puede aprovechar la computación en la nube para ejecutar código en respuesta a eventos con administración automática de los recursos computacionales.

Esta sección detalla el proceso de configuración y despliegue de una función Lambda en AWS, comenzando con la creación y configuración de usuarios IAM, seguido por la especificación y configuración del servicio Lambda. Se enfatizará la importancia de una configuración adecuada de IAM para garantizar un manejo seguro y restringido de los permisos, lo cual es crucial para la protección de los recursos en la nube y la minimización de riesgos de seguridad. Posteriormente, se discutirá cómo configurar adecuadamente el entorno de ejecución de Lambda, incluyendo la asignación de memoria y los permisos necesarios, para asegurar que la función se ejecute de manera óptima y coste-efectiva.

4.5.1. Usuarios IAM

La gestión de identidades y accesos (IAM) es fundamental para la seguridad y la administración en AWS, ofreciendo control granular sobre quién puede hacer qué en cada recurso de AWS. La creación de usuarios IAM específicos para diferentes tareas asegura que los servicios operen bajo el principio de mínimo privilegio, reduciendo el riesgo de accesos no autorizados o malintencionados. Al iniciar el despliegue de una función Lambda, es primordial configurar un usuario IAM con permisos precisos que permitan gestionar Lambda y otros servicios necesarios sin exceder las capacidades requeridas.

El primer paso implica crear un grupo IAM con políticas que confieren acceso necesario para operar funciones Lambda, tales como *AWSLambdaFullAccess*, que permite a los usuarios gestionar funciones y recursos relacionados en Lambda. Adicionalmente, se añade la política *IAMFullAccess* temporalmente para permitir la configuración de roles y políticas adicionales necesarias durante la fase inicial de configuración y despliegue. La creación de un usuario dentro de este grupo y la asignación

de credenciales de acceso programático son acciones que habilitan la interacción con AWS a través de la CLI o SDK, herramientas esenciales para automatizar y gestionar aplicaciones en la nube de manera eficiente.

4.5.2. Configuración de Lambda

Una vez establecida la gestión segura de accesos mediante IAM, el siguiente paso es configurar la función Lambda propiamente. La configuración adecuada de una función Lambda incluye especificar el entorno de ejecución, los permisos y la gestión de recursos. Se selecciona un entorno de ejecución que corresponda al lenguaje de programación del código fuente, en este caso, Python 3.8. La asignación de memoria y el tiempo de ejecución máximos son configurados en según las necesidades estimadas de la función, que en este caso se establecen en 250 MB de memoria y 15 segundos de tiempo de ejecución máximo, respectivamente.

Para permitir que la función Lambda interactúe eficientemente con otros servicios de AWS sin comprometer la seguridad, se crea un rol específico de IAM, *LambdaExecutionRole*, con políticas que otorgan los permisos necesarios para ejecutar la función. Este rol incluye políticas como *AWSLambdaBasicExecutionRole*, que permite a la función escribir registros en Amazon CloudWatch, esencial para monitorizar y depurar la función. La definición de estos permisos asegura que la función tenga acceso solo a los recursos que necesita, cumpliendo con las mejores prácticas de seguridad.

Finalmente, se procede a desplegar el código fuente mediante AWS CLI, especificando todos los parámetros configurados y el rol de ejecución. Este paso es crucial porque compila y activa la función en el entorno de nube, listo para ser invocado según sea necesario. La capacidad de desplegar y gestionar funciones de manera programática mediante AWS CLI no solo aumenta la eficiencia, sino también asegura que el despliegue sea repetible y consistente, eliminando errores manuales potenciales y facilitando la integración continua y la entrega continua (CI/CD) en entornos de desarrollo profesional.

Capítulo 5. Desarrollo de la arquitectura

Este capítulo detalla el desarrollo y la implementación de una serie de modelos de detección de objetos basados en las versiones mejoradas del algoritmo YOLO (You Only Look Once). Comenzaremos explorando el modelo YOLOv4, implementado en AWS Lambda para optimizar tanto costos como recursos, seguido de una transición hacia YOLOv5 y finalmente YOLOv8, cada uno con mejoras significativas en eficiencia y precisión. Los modelos son diseñados para operar en entornos *serverless*, aprovechando plataformas como Google Colab y Amazon SageMaker, facilitando la integración y manejo eficiente de los recursos computacionales. Esta documentación cubre desde la configuración inicial de los *datasets* hasta la generación final de archivos GNS3, pasando por las etapas de entrenamiento, validación y detección, con el objetivo de proporcionar una comprensión exhaustiva del proceso y las tecnologías utilizadas.

5.1. Modelo de entrenamiento de YOLOv4

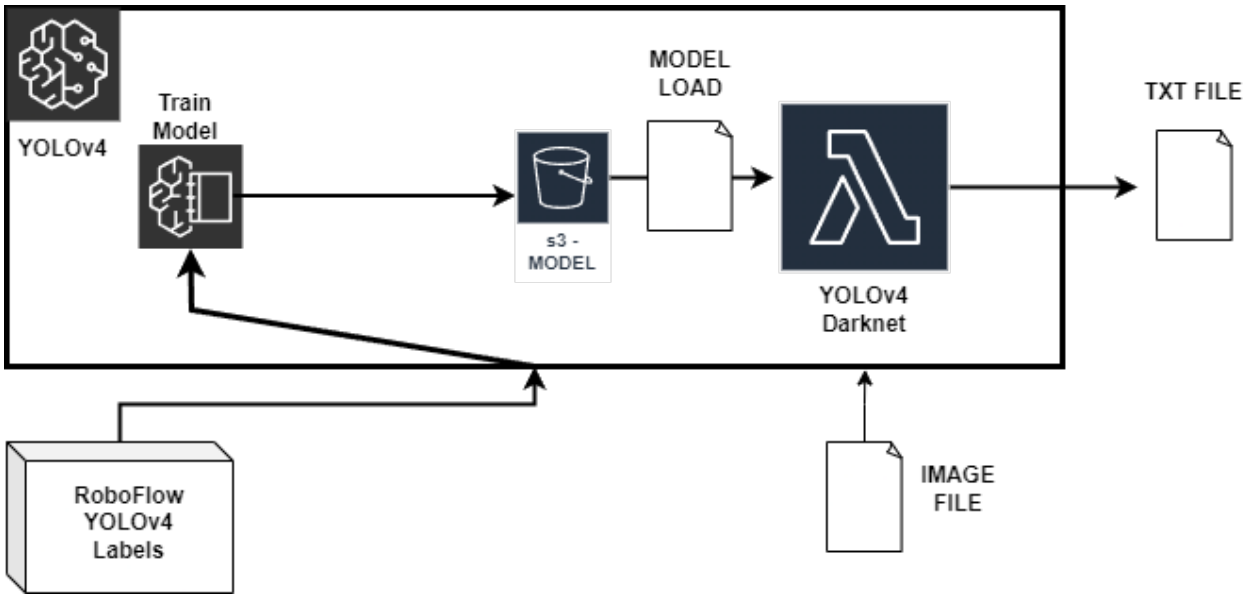


Figura 5.1: Arquitectura del modelo de YOLO V4 Darknet implementado en AWS Lambda para optimización de costos y recursos, mostrando la interacción entre los componentes principales como AWS Lambda, S3 y Google Colab.

El modelo de YOLO V4 Darknet es la primera arquitectura que

discutiremos. Se eligió debido a su eficacia comprobada en el estado del arte y su aplicación exitosa en proyectos previos usando Darknet para detección en Amazon Web Service Lambda. Este enfoque es clave para reducir costos operativos, evitando el uso de Amazon SageMaker o *endpoints* que incrementan los costos por tener que gestionar su creación y eliminación continuas.

En nuestro enfoque, buscamos optimizar recursos al ejecutar el detector de manera nativa en YOLOv4 desde Lambda, permitiendo procesamientos paralelos para manejar múltiples detecciones y la generación simultánea de archivos GNS3. Para implementar esto, utilizamos un *notebook* generado por RoboFlow, que se puede adaptar tanto en SageMaker como en Google Colab. Este *notebook* se conecta a nuestra API para descargar modelos etiquetados, preparados especialmente para YOLOv4 a partir de elementos previamente dibujados a mano.

Antes de iniciar el entrenamiento, estos datos se cargan en el *notebook*, donde se realiza el proceso de entrenamiento. Una limitación observada es que, mientras SageMaker completa el entrenamiento aunque sea lento, Google Colab frecuentemente no concluye este proceso, resultando en modelos de baja calidad debido al principio de ”garbage in, garbage out”. Esto subraya la importancia de la calidad de los datos de entrada.

No obstante, el entrenamiento exitoso genera un modelo que se almacena en un *bucket* S3 en AWS, esencial para operar el modelo de YOLOv4 en Lambda, que requiere un archivo de pesos específico. El sistema está configurado para que, al recibir una imagen a través de una API o un *bucket* S3, se active un disparador que carga el modelo de S3 para realizar la detección. Los resultados se depositan en un archivo `.txt` que se puede procesar posteriormente para refinar los resultados y generar un archivo GNS3 preciso.

5.2. Modelo de entrenamiento de YOLOv5

El modelo YOLOv5 representa una evolución en nuestros esfuerzos de optimización tras experimentar con YOLOv4. Este modelo se ha seleccionado dentro del estado del arte por su capacidad para realizar un gran número de detecciones eficientes con un conjunto limitado de datos, ideal dado que nuestro *dataset* fue creado manualmente y es de tamaño reducido. La optimización de recursos es crucial, ya que carecemos de acceso a *hardware* avanzado como GPU o TPU para un entrenamiento

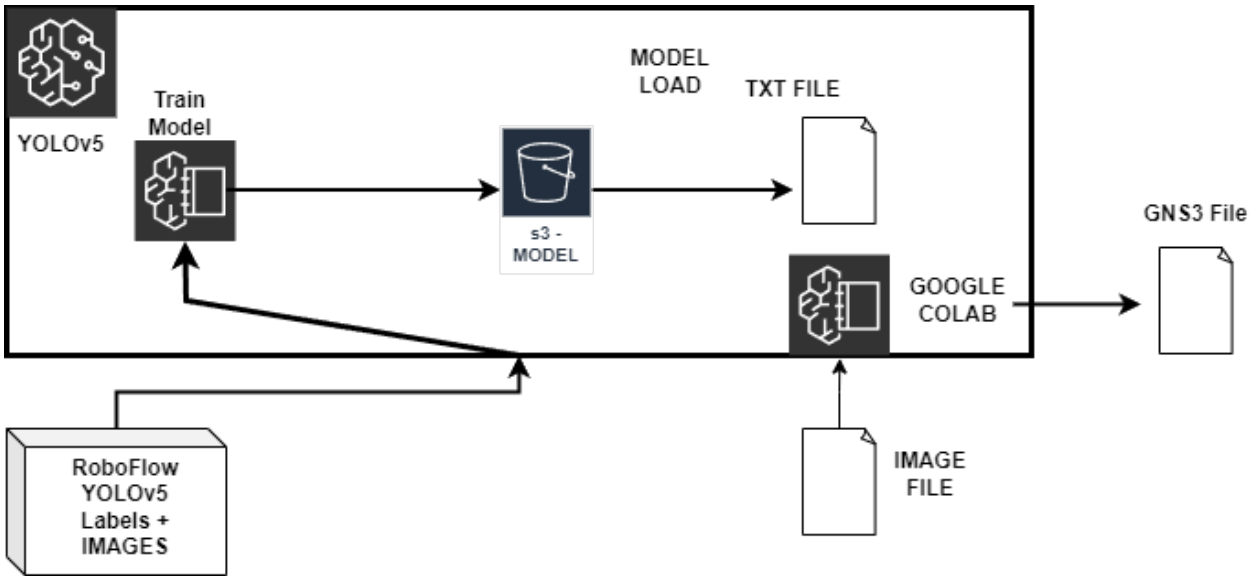


Figura 5.2: Esquema del modelo YOLOv5 implementado en plataformas de computación en la nube, ilustrando la estructura de datos y el flujo de entrenamiento y detección.

eficiente localmente, lo que nos lleva a utilizar plataformas como Google Colab y Amazon SageMaker para minimizar costos y aprovechar sus capacidades computacionales.

Iniciamos el proceso de entrenamiento con archivos proporcionados por RoboFlow, que esta vez incluyen una estructura organizada en dos carpetas principales: una para etiquetas y configuraciones, y otra para los datos de entrenamiento, validación y test, distribuidos en tres subcarpetas separadas. Este formato facilita la gestión y uso de las imágenes en las etapas sucesivas. Los datos son cargados a través de una API a un cuadernillo que es compatible tanto con SageMaker como con Google Colab, produciendo resultados similares en términos de tiempo y eficacia.

Para esta arquitectura, hemos optado por utilizar Google Colab debido a su accesibilidad sin costo. El modelo resultante no es un archivo de pesos convencional, sino un archivo `.pt` compatible con PyTorch, lo cual implica un cambio significativo en la gestión del modelo. Dado que YOLOv5 no se integra directamente en las funciones Lambda de AWS debido a las exigencias de hardware, como una GPU, para su funcionamiento óptimo, nos enfrentamos a restricciones en la implementación directa en AWS Lambda.

La solución provisional ha sido utilizar Amazon SageMaker o Google Colab para recibir este modelo y realizar las detecciones, generando un archivo `.txt` que puede ser procesado posteriormente en una función

Lambda para la creación del archivo GNS3. Esta aproximación nos permite continuar con nuestro objetivo de minimizar el uso de recursos mientras enfrentamos las limitaciones técnicas de la plataforma.

5.3. Modelo de entrenamiento de YOLOv8

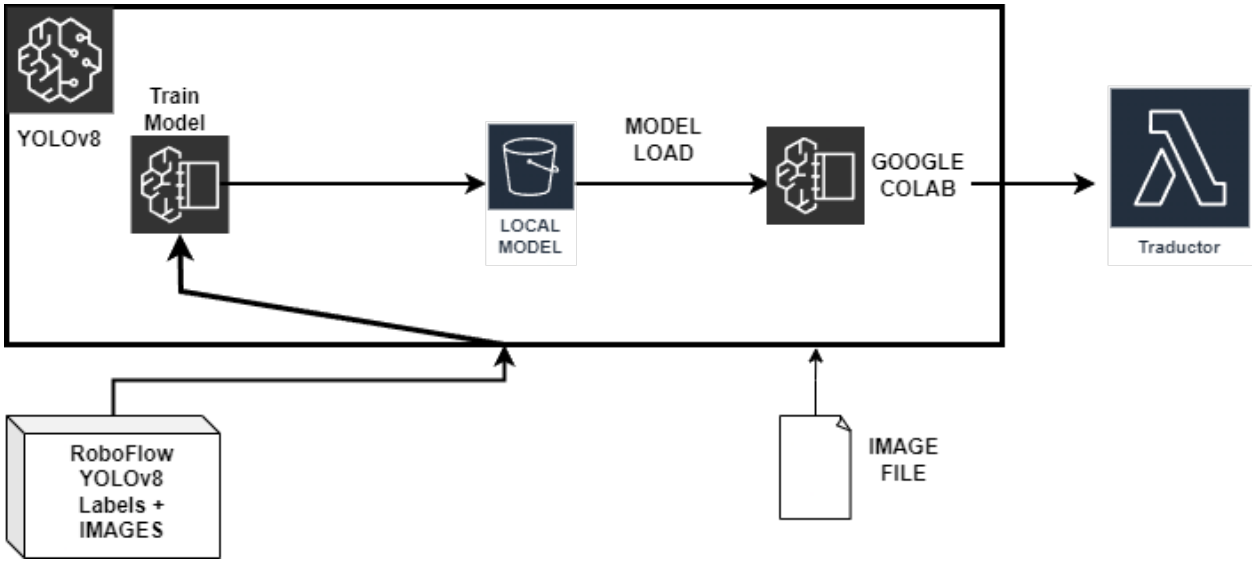


Figura 5.3: Diagrama del flujo de entrenamiento y detección del modelo YOLOv8 en Google Colab, destacando la eficiencia y optimización en el uso de recursos.

El modelo YOLOv8 representa la cima de nuestra serie de optimizaciones en los modelos YOLO. A partir de una evaluación detallada de la literatura y estudios de caso, hemos observado que la transición de YOLOv5 a YOLOv8 ofrece mejoras significativas en términos de eficiencia en el consumo de recursos, velocidad de procesamiento y precisión en la detección. En la práctica, el modelo YOLOv8 se ha configurado y entrenado a través de un proceso integral que utiliza Google Colab, partiendo de una estructura de datos organizada y cargada mediante una API desde RoboFlow.

El cuadernillo de Google Colab, especialmente configurado para el procesamiento de YOLOv8, maneja la carga inicial de datos, ejecuta el entrenamiento y genera el modelo óptimo que vamos a implementar en nuestras arquitecturas futuras. Este modelo se convierte en la base sobre la que se construirán las soluciones de detección posteriores, prescindiendo del uso de Amazon SageMaker en favor de una solución totalmente integrada en Google Colab.

Esta estrategia no solo simplifica la infraestructura necesaria sino que también concentra todas las operaciones de entrenamiento y detección dentro de Google Colab. El resultado de este proceso es un archivo *.txt*, el cual es procesado posteriormente para refinar los resultados y generar un archivo *.gns3* optimizado, asegurando una integración eficiente y efectiva en nuestras aplicaciones finales.

5.4. Arquitectura de detección en Cloud Serverless y generación de GNS3

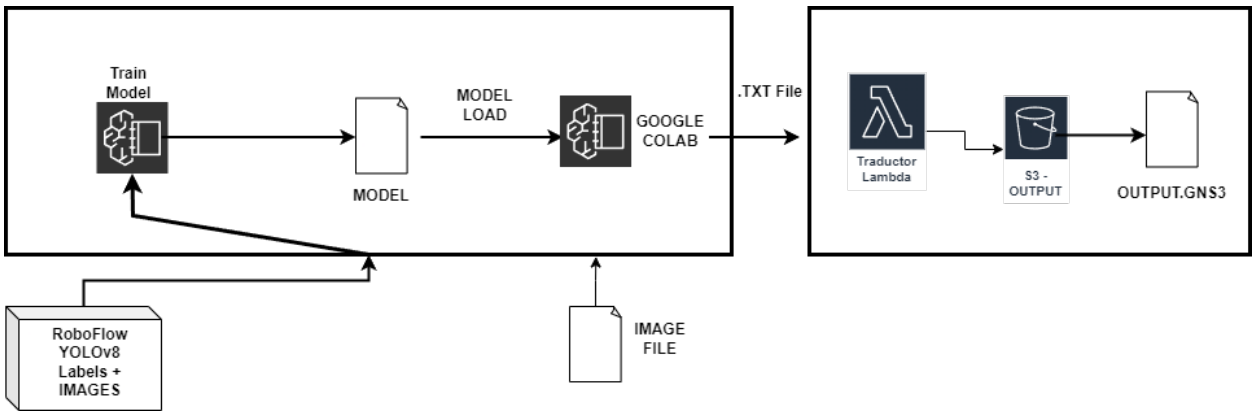


Figura 5.4: Diagrama de la arquitectura serverless para detección utilizando YOLOv8 con integración de Google Colab y AWS Lambda, mostrando el flujo de datos desde la generación del modelo hasta la salida final en archivos GNS3.

Concluyendo, la arquitectura completa incorpora el modelo YOLOv8, integrando las opciones de detección a través de SageMaker Lambda o Google Colab. Si optamos por Google Colab, se implementa un panel de control directamente en el cuadernillo que permite una interacción eficiente con la API de Amazon Web Service Lambda para el despliegue del archivo `.txt` generado. El flujo operativo se mantiene consistente: iniciamos con la creación del modelo en YOLOv8 dentro del cuadernillo y, una vez completado, el modelo es exportable sin complicaciones.

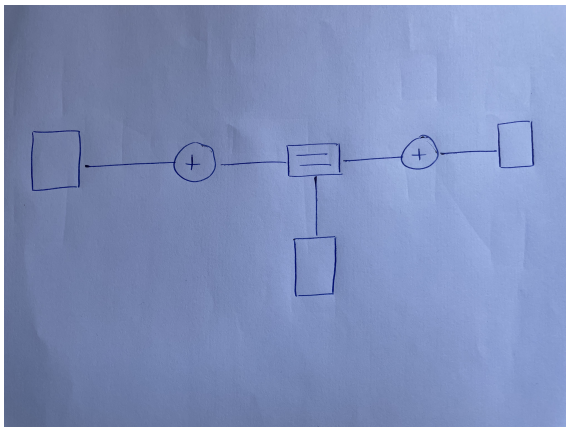
Desde el panel de Google Colab, se activa un disparador que deposita el archivo `.txt` en un *bucket* S3 de AWS Lambda. Posteriormente, las imágenes para detección se añaden manualmente o mediante API en una carpeta de entrada. Dependiendo de la infraestructura elegida, ya sea un cuadernillo de SageMaker o de Google Colab, estas imágenes son procesadas para generar un archivo `.txt`. En nuestro caso, para reducir costos, hemos seleccionado Google Colab, que automáticamente

produce un archivo `.txt` detallando las detecciones y coordenadas de cada elemento.

Este archivo se almacena luego en un bucket S3 específico que actúa como un disparador para procesar hasta 1000 archivos simultáneamente, optimizando recursos y acelerando la generación de archivos GNS3 de manera eficiente. Estos archivos se depositan finalmente en una carpeta de salida, donde una función Lambda procesa cada uno, traduciendo el contenido del archivo `.txt` en archivos `.gns3` según los procedimientos establecidos en capítulos anteriores.

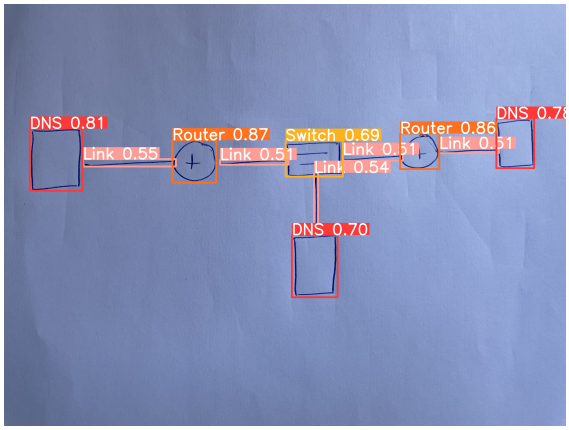
5.5. Transición de Dibujo a GNS3

El siguiente diagrama muestra la transición desde el dibujo a mano hasta el modelo final en GNS3.



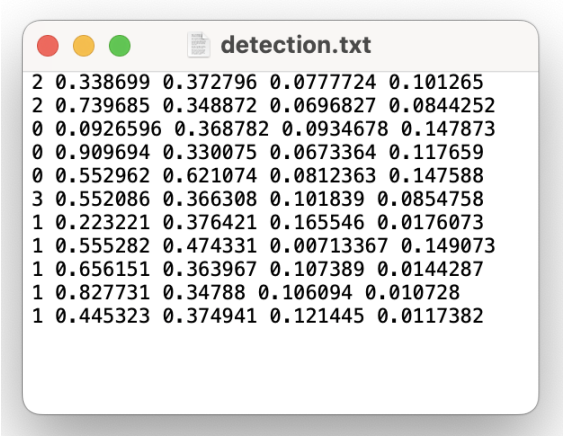
1 - Dibujo a mano

↓



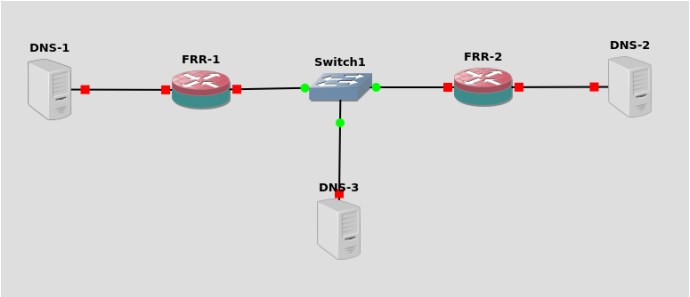
2 - Reconocimiento de objetos en el dibujo

↓



3 - Archivo generado a partir de la predicción

↓



4 - Topología generada en GNS3.

1. Dibujamos la topología que queremos digitalizar siguiendo la leyenda descrita anteriormente en el apartado 1.2.

2. Después de realizar una foto al dibujo. Subimos esa foto al modelo para que haga sus predicciones, y nos genere las cajas en los elementos que ha predicho.

3. Las predicciones son generadas, por parte del modelo, en un archivo *.txt*. Cada linea de este archivo contiene la información del elemento así como sus coordenadas y su anchura y altura.

4. A partir del archivo *.txt* se genera un archivo *.gns3* con la estructura necesaria para importar el proyecto a la herramienta GNS3.

Capítulo 6. Resultados, mediciones y conclusiones

6.1. Metodología de Pruebas de Medición y Costos

Durante este proyecto, hemos explorado el proceso desde la generación y preparación del *dataset*, pasando por la experimentación con distintos modelos, hasta el desarrollo de estrategias para interpretar los datos finales y la subida a la nube de AWS. Este trabajo nos ha permitido tener una visión general de todo el proceso de creación de proyectos orientados a la visión por ordenador.

La primera parte ha sido la más meticulosa, ya que hemos tenido que generar un *dataset* desde cero. Hemos utilizado la plataforma Roboflow para etiquetar manualmente 80 dibujos, que consideramos que es una buena base sobre la que luego poder generar más dibujos con técnicas de aumento de datos. Así hemos conseguido un total de 136 imágenes. Además, preprocesamos las imágenes convirtiéndolas a escala de grises para garantizar uniformidad en el *dataset* y prepararlo para el entrenamiento de los modelos. Este proceso nos ha proporcionado un conjunto de datos coherente y diverso para nuestro proyecto.

La siguiente fase del proyecto ha sido la experimentación con los modelos YOLOv4 Darknet, YOLOv5 y YOLOv8. El proceso de configuración de YOLOv4 ha resultado tedioso debido a la obsolescencia de muchas herramientas y bibliotecas, lo que ha generado conflictos al intentar compilar el proyecto en Darknet. Además, el coste computacional fue elevado, especialmente al intentar entrenar el modelo en Google Colab con recursos limitados.

Ante estas dificultades, se han explorado versiones más recientes como YOLOv5, que ofrece mejor soporte y rendimiento en entornos con recursos limitados. Se han ajustado los parámetros de entrenamiento para adaptar el modelo a nuestro caso particular, incluyendo la reducción del tamaño del *batch* y el aumento del número de épocas. Este proceso también nos ha presentado desafíos, ya que con los ajustes realizados no hemos querido caer en el sobreajuste. Los resultados obtenidos con este modelo han sido esperanzadores, pero hemos experimentado con la última versión de

YOLO para ver su evolución.

Finalmente, hemos experimentado con YOLOv8, realizando ajustes similares a los de YOLOv5 en los parámetros de entrenamiento. Se ha configurado un tamaño de *batch* más pequeño y se ha aprovechado la función de *early stopping* para detener el entrenamiento cuando no se observaran mejoras significativas. Los resultados obtenidos han sido positivos, demostrando un buen rendimiento del modelo para el tamaño limitado del *dataset*.

La penúltima fase del proyecto ha sido la interpretación de las salidas de los modelos que supuso la implementación de estrategias para validar conexiones entre los elementos de la red a partir de sus coordenadas. También, la generación del archivo final para GNS3 implicó el estudio y diseño del archivo que queríamos dar como salida en formato `.gns3`.

Para finalizar, hemos probado los modelos YOLO con todos los archivos del *dataset* y para generar los archivos en base a estos, también se han realizado de forma paralela 6 generaciones de topologías para comprobar si salían de forma correcta corriendo la función lambda varias veces con los mismos datos de detección. Estas pruebas han permitido validar la eficacia de nuestras estrategias y ajustes, garantizando la capacidad del sistema para manejar múltiples tareas de forma simultánea y efectiva.

6.2. Evaluación del Rendimiento de los Modelos de Detección

La selección y optimización de modelos de detección de objetos son cruciales para el éxito de proyectos que involucran visión por computadora, especialmente en aplicaciones que requieren la transformación de datos visuales complejos en representaciones digitales procesables. En este proyecto, se evaluaron tres modelos avanzados: YOLOv4 Darknet, YOLOv5 y YOLOv8. Cada uno de estos modelos ofrece características únicas en términos de precisión, velocidad y consumo de recursos computacionales.

6.2.1. Evaluación de YOLOv4 Darknet

Aunque YOLOv4 ha sido reconocido por su alta precisión en la detección de objetos, nos enfrentamos a desafíos significativos debido a la obsolescencia de algunas bibliotecas y a la intensidad de recursos requerida para su entrenamiento. Este modelo mostró una alta precisión en la detección, pero su configuración y el ambiente de entrenamiento en

Google Colab limitaron su aplicabilidad debido a restricciones de tiempo y memoria.

6.2.2. Transición y Optimización con YOLOv5

Al experimentar con YOLOv5, observamos mejoras en la velocidad de entrenamiento y una gestión más eficiente de los recursos computacionales. Este modelo permitió una mayor flexibilidad en la adaptación de los parámetros, como el tamaño del batch y el número de épocas, para evitar el sobreajuste y mejorar la generalización del modelo. Los resultados fueron prometedores, ofreciendo un balance adecuado entre precisión y velocidad, adecuado para nuestro conjunto de datos de tamaño moderado.

6.2.3. Implementación de YOLOv8

La introducción de YOLOv8 marcó un avance significativo, especialmente en términos de eficiencia operativa y manejo de recursos. La capacidad de YOLOv8 para detener automáticamente el entrenamiento cuando no se observan mejoras significativas (early stopping) optimizó el uso de la nube, minimizando los costos computacionales sin sacrificar la calidad de la detección.

6.2.4. Comparación de Modelos y Selección Final

La comparativa entre los modelos demostró que YOLOv8 es el más adecuado para aplicaciones que requieren una alta eficiencia en tiempo real y un manejo óptimo de recursos. Sin embargo, la elección del modelo adecuado debe considerar tanto el contexto de aplicación específico como las limitaciones de recursos. En nuestro caso, YOLOv8 proporcionó el mejor equilibrio, preparando el proyecto para una implementación más amplia y eficiente.

Esta evaluación del rendimiento no solo fundamenta nuestra selección de modelo, sino que también contextualiza la discusión sobre los costos operativos que se explorarán en la siguiente sección, destacando cómo la eficiencia del modelo influye directamente en la viabilidad económica del proyecto.

6.3. Aproximación de Costes para Función Lambda

Dado que los precios de AWS son variables, es vital realizar una aproximación de costes para la planificación presupuestaria. Para una función Lambda con 250 MB de memoria y una duración máxima de 15

segundos, procedemos a calcular los costos asociados.

6.3.1. Cálculo de Costos

Los costos de AWS Lambda se descomponen en costo por petición y costo por cómputo. El cálculo se puede representar de la siguiente manera:

$$\text{Costo Función Lambda} = \text{Costo por Petición} + \text{Costo por Cómputo}$$

$$\text{Función (250 MB, 15 s)} = \frac{n}{10^6} \times 0{,}2 + \left(\frac{250}{1024} \times 15\right) \times n \times 0{,}00001667$$

donde n representa el número de ejecuciones de la función.

6.3.2. Análisis de Costos

Para una mejor visualización de estos costos, proporcionamos una tabla resumen con los costos estimados para un lote de procesamiento de 1000 archivos:

Cuadro 6.1: Resumen de Costos y Tiempos para un Lote de Procesamiento (1000 archivos en total)

Función	**Tiempo (s)**	**Costo (USD)**
Función Lambda (Procesamiento)	15	\$0.06125
Total	15	\$0.06125

6.3.3. Discusión

Los costos calculados arrojan luz sobre la eficiencia y viabilidad económica de utilizar AWS Lambda para el procesamiento serverless en nuestro proyecto. A pesar de la baja tarifa unitaria, el costo total puede incrementarse significativamente con el aumento del número de ejecuciones, lo cual es crítico para proyectos a gran escala. Estos resultados nos permiten considerar ajustes en la memoria asignada o la optimización del tiempo de ejecución para controlar mejor los gastos operativos.

6.3.4. Conclusiones

Esta aproximación nos permite anticipar el presupuesto necesario para el despliegue de funciones serverless en un entorno real, y subraya la importancia de optimizar tanto el código como la asignación de recursos para minimizar los costos operacionales.

6.4. Objetivos Realizados

Podemos destacar que, si bien el proyecto ha cumplido gran parte de los objetivos planteados en la introducción, existen áreas donde aún se necesita trabajo adicional. El programa desarrollado ha demostrado su capacidad para interpretar dibujos de topologías de red y generar automáticamente un archivo compatible con GNS3 que representa los nodos, pero ha quedado como trabajo futuro el formateado de los Enlaces para seguir el estandar de `.gns3`. Aunque los objetivos no se han logrado en su totalidad, el proyecto ha sentado las bases para futuras mejoras y desarrollos.

6.5. Análisis Crítico de los Resultados

En este proyecto, se ha llevado a cabo una exploración exhaustiva de la generación automática de topologías de red a través del reconocimiento de dibujos manuscritos, utilizando para ello modelos avanzados de detección de objetos como YOLOv4, YOLOv5 y YOLOv8. A pesar de los avances significativos y los resultados alentadores obtenidos, es importante reconocer las limitaciones y desafíos enfrentados durante el desarrollo.

Uno de los principales retos ha sido la variabilidad en la calidad del reconocimiento de los dibujos, lo cual ha puesto de manifiesto la importancia de un preprocesamiento eficaz y una anotación precisa del dataset. Aunque se han hecho esfuerzos para mejorar la calidad del dataset mediante técnicas de aumento de datos y conversión a escala de grises, el principio de “garbage in, garbage out” sigue siendo una barrera significativa para la optimización del rendimiento del modelo.

Además, el uso de plataformas como Google Colab y AWS Lambda ha presentado desafíos relacionados con la limitación de recursos y la gestión de costos computacionales, especialmente cuando se trabaja con modelos computacionalmente demandantes como YOLOv4. Aunque las versiones más recientes, como YOLOv8, han mostrado mejoras en eficiencia y rendimiento, la transición de YOLOv4 a YOLOv8 ha requerido ajustes significativos en los parámetros de entrenamiento y configuración.

6.6. Trabajo Futuro

Mirando hacia el futuro, hay varias áreas de mejora y expansión que podrían enriquecer aún más este proyecto. Primero, se propone la expansión del *dataset* con más dibujos y variaciones para mejorar la robustez del modelo frente a variaciones en los datos de entrada. Esto incluiría la exploración de técnicas adicionales de aumento de datos y la posible integración de aprendizaje semi-supervisado o no supervisado para aprovechar los datos no etiquetados.

También sería beneficioso explorar algoritmos alternativos de reconocimiento de objetos que puedan ofrecer un mejor equilibrio entre precisión, velocidad y costos computacionales. Por ejemplo, la implementación de redes neuronales más ligeras o la personalización de modelos existentes para adaptarlos específicamente a las necesidades y limitaciones del proyecto.

Finalmente, para abordar las cuestiones de escalabilidad y gestión de recursos en la nube, se recomienda una investigación más profunda sobre las arquitecturas *serverless* y los modelos de precios en plataformas como AWS. Esto ayudaría a optimizar los costos y mejorar la eficiencia operativa del sistema propuesto.

Chapter 6. Results, Measurements, and Conclusions

6.1. Methodology of Measurement and Cost Testing

During this project, we explored the process from generating and preparing the dataset, through experimenting with different models, to developing strategies for interpreting the final data and uploading to AWS cloud. This work has allowed us to have an overview of the entire process involved in creating computer vision-oriented projects.

The first part has been the most meticulous, as we had to generate a dataset from scratch. We used the Roboflow platform to manually label 80 drawings, which we consider a solid base on which to later generate more drawings using data-augmentation techniques. This resulted in a total of 136 images. Additionally, we pre-processed the images by converting them from grayscale to ensure uniformity in the data set and to prepare it for training the models. This process provided us with a coherent and diverse dataset for our project.

The next phase of the project involved experimenting with the YOLOv4 Darknet, YOLOv5, and YOLOv8 models. The setup process for YOLOv4 was tedious due to the obsolescence of many tools and libraries, which created conflicts when trying to compile the project in Darknet. Moreover, the computational cost was high, especially when attempting to train the model on Google Colab with limited resources.

In light of these difficulties, we explored newer versions like YOLOv5, which offers better support and performance in resource-limited environments. We adjusted the training parameters to tailor the model to our specific case, including reducing the batch size and increasing the number of epochs. This process also presented challenges, as we did not want to fall into overfitting with the adjustments made. The results obtained with this model were promising, but we experimented with the latest version of YOLO to see its evolution.

Finally, we experimented with YOLOv8, making similar adjustments to the training parameters as with YOLOv5. We set up a smaller batch size and utilized the early stopping feature to halt training when no significant improvements were observed. The results were positive, demonstrating

good model performance given the limited size of the dataset.

The penultimate phase of the project was the interpretation of the model outputs, which involved implementing strategies to validate connections between network elements based on their coordinates. Also, generating the final file for GNS3 involved studying and designing the file we wanted to output in *.gns3* format.

In conclusion, we tested the YOLO models with all the files in the dataset and to generate the files based on these, we also conducted six parallel generations of topologies to check if they were produced correctly by running the lambda function multiple times with the same detection data. These tests have validated the effectiveness of our strategies and adjustments, ensuring the system's ability to handle multiple tasks simultaneously and effectively.

6.2. Performance Evaluation of Detection Models

The selection and optimization of object detection models are crucial for the success of projects involving computer vision, especially in applications that require the transformation of complex visual data into processable digital representations. In this project, we evaluated three advanced models: YOLOv4 Darknet, YOLOv5, and YOLOv8. Each of these models offers unique features in terms of accuracy, processing speed, and computational resource consumption.

6.2.1. Evaluation of YOLOv4 Darknet

Although YOLOv4 has been recognized for its high object detection accuracy, we faced significant challenges due to the obsolescence of some libraries and the resource intensity required for training. This model demonstrated high accuracy in detection, but its configuration and the training environment on Google Colab limited its applicability due to time and memory constraints.

6.2.2. Transition and Optimization with YOLOv5

When experimenting with YOLOv5, we observed improvements in training speed and more efficient management of computational resources. This model allowed greater flexibility in adapting parameters, such as batch size and number of epochs, to prevent overfitting and enhance model generalization. The results were promising, offering a suitable balance between accuracy and speed, appropriate for our

moderately sized dataset.

6.2.3. Implementation of YOLOv8

The introduction of YOLOv8 marked a significant advancement, especially in terms of operational efficiency and resource management. YOLOv8's ability to automatically stop training when no significant improvements are observed (early stopping) optimized cloud usage, minimizing computational costs without sacrificing detection quality.

6.2.4. Model Comparison and Final Selection

The comparison between the models demonstrated that YOLOv8 is most suitable for applications that require high real-time efficiency and optimal resource management. However, the choice of the right model should take into account both the specific application context and resource limitations. In our case, YOLOv8 provided the best balance, preparing the project for a wider and more efficient implementation.

This performance evaluation not only substantiates our model selection, but also contextualizes the discussion on operational costs to be explored in the following section, highlighting how the efficiency of the model directly influences the economic viability of the project.

6.2.5. Cost Analysis

For better visualization of these costs, we provide a summary table with the estimated costs for processing a batch of 1000 files:

Cuadro 6.2: Summary of Costs and Times for a Processing Batch (1000 files total)

Function	**Time (s)**	**Cost (USD)**
Lambda Function (Processing)	15	$0.06125
Total	15	$0.06125

6.2.6. Discussion

The calculated costs shed light on the efficiency and economic viability of using AWS Lambda for serverless processing in our project. Despite the low unit rate, the total cost can increase significantly with the number of executions, which is critical for large-scale projects. These results allow us to consider adjustments in memory allocation or optimization of execution time to better control operational expenses.

6.2.7. Conclusions

This approach allows us to anticipate the budget needed for deploying serverless functions in a real environment and highlights the importance of optimizing both the code and resource allocation to minimize operational costs.

6.3. Achieved Objectives

It should be noted that, while the project has met many of the objectives set out in the Introduction, there are areas where additional work is needed. The developed program has demonstrated its ability to interpret network topology drawings and automatically generate a GNS3 compatible file that represents the nodes, but future work remains to be done on the formatting of the Links to follow the `.gns3` standard. Although the objectives have not been fully achieved, the project has laid the groundwork for future improvements and developments.

6.4. Critical Analysis of Results

In this project, an exhaustive exploration of the automatic generation of network topologies through the recognition of handwritten drawings was carried out, utilizing advanced object detection models such as YOLOv4, YOLOv5 and YOLOv8. Despite significant advances and encouraging results obtained, it is important to recognize the limitations and challenges encountered during development.

One of the main challenges has been the variability in the quality of the drawings' recognition, which has highlighted the importance of effective preprocessing and accurate dataset annotation. Although efforts have been made to improve the quality of the data set using data augmentation techniques and conversion to grayscale, the principle of "garbage in, garbage out" remains a significant barrier to optimizing model performance.

Moreover, the use of platforms like Google Colab and AWS Lambda has presented challenges related to resource limitations and the management of computational costs, especially when working with computationally demanding models such as YOLOv4. Although newer versions, such as YOLOv8, have shown improvements in efficiency and performance, the transition from YOLOv4 to YOLOv8 required significant adjustments in training parameters and configurations.

6.5. Future Work

Looking ahead, there are several areas of improvement and expansion that could further enrich this project. First, an expansion of the dataset with more drawings and variations is proposed to enhance the model's robustness against variations in input data. This would include exploring additional data augmentation techniques and the potential integration of semisupervised or unsupervised learning to leverage unlabeled data.

It would also be beneficial to explore alternative object recognition algorithms that might offer a better balance between accuracy, speed, and computational costs. For example, implementing lighter neural networks or customizing existing models to specifically suit the needs and limitations of the project.

Finally, to address issues of scalability and resource management in the cloud, further research on serverless architectures and pricing models on platforms like AWS is recommended. This would help optimize costs and improve the operational efficiency of the proposed system.

Bibliografía

[1] S. Albahli, N. Nida, A. Irtaza, M. H. Yousaf, M. Tariq, and M. Mahmood, "Melanoma lesion detection and segmentation using yolov4-darknet and active contour," 11 2020.

[2] R. R. Reddy and M. R. Panicker, "Hand-drawn electrical circuit recognition using object detection and node recognition," *CoRR*, vol. abs/2106.11559, 2021. [Online]. Available: https://arxiv.org/abs/2106.11559

[3] A. S. Hassanein, S. Mohammad, M. Sameer, and M. E. Ragab, "A survey on hough transform, theory, techniques and applications," *CoRR*, vol. abs/1502.02160, 2015. [Online]. Available: http://arxiv.org/abs/1502.02160

[4] D. Reis, J. Kupec, J. Hong, and A. Daoudi, "Real-time flying object detection with yolov8," 2023.

[5] D. Casado Jimenez, "Simulación de una red SDN de videovigilancia IP basada en GNS3," Ph.D. dissertation, Universitat Politècnica de València, 2020.

[6] D. Pacios, J. L. Vázquez-Poletti, D. B. Dhuri, D. Atri, R. Moreno-Vozmediano, R. J. Lillis, N. Schetakis, J. Gómez-Sanz, A. D. Iorio, and L. Vázquez, "A serverless computing architecture for martian aurora detection with the emirates mars mission," *Scientific Reports*, vol. 14, no. 1, p. 3029, 2024.

[7] C. Dewi and H. Juli Christanto, "Combination of deep cross-stage partial network and spatial pyramid pooling for automatic hand detection," *Big Data and Cognitive Computing*, vol. 6, no. 3, 2022. [Online]. Available: https://www.mdpi.com/2504-2289/6/3/85

[8] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[9] R. Mukherjee, O. Tripp, B. Liblit, and M. Wilson, "Static analysis for AWS best practices in Python code," *arXiv preprint arXiv:2205.04432*, 2022.

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

[11] M. Sha, "A cloud based sentiment analysis through logistic regression in AWS platform." *Computer Systems Science & Engineering*, vol. 45, no. 1, 2023.

[12] M. Asif, T. Rajab, S. Hussain, M. Rashid, S. Wasi, A. Ahmed, K. Kanwal *et al.*, "Performance evaluation of deep learning algorithm using high-end media processing board in real-time environment," *Journal of Sensors*, vol. 2022, 2022.

[13] D. Chen, S. Sun, Z. Lei, H. Shao, and Y. Wang, "Ship target detection algorithm based on improved YOLOv3 for maritime image," *Journal of Advanced Transportation*, vol. 2021, pp. 1–11, 2021.

[14] C. Dong, C. Pang, Z. Li, X. Zeng, and X. Hu, "Pg-yolo: A novel lightweight object detection method for edge devices in industrial internet of things," *IEEE Access*, vol. 10, pp. 123 736–123 745, 2022.

[15] R. Gao, S. Zhang, H. Wang, J. Zhang, H. Li, Z. Zhang *et al.*, "The aeroplane and undercarriage detection based on attention mechanism and multi-scale features processing," *Mobile Information Systems*, vol. 2022, 2022.

[16] A. Bochkovskiy, C. Wang, and H. M. Liao, "YOLOv4: Optimal speed and accuracy of object detection," *CoRR*, vol. abs/2004.10934, 2020. [Online]. Available: https://arxiv.org/abs/2004.10934

[17] C.-Y. Wang, A. Bochkovskiy, and H.-Y. M. Liao, "Scaled-YOLOv4: Scaling cross stage partial network," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2021, pp. 13 029–13 038.

[18] G. Jocher, "YOLOv5 by ultralytics," 2020. [Online]. Available: https://github.com/ultralytics/yolov5

[19] J. Torres, "YOLOv8 architecture explained: Exploring the YOLOv8 architecture," 2024. [Online]. Available: https://yolov8.org/yolov8-architecture-explained/