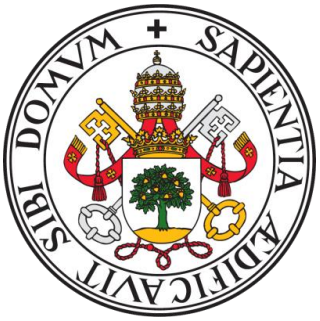
UNIVERSIDAD DE VALLADOLID



E.T.S.I. TELECOMUNICACIÓN

TRABAJO FIN DE GRADO

GRADO EN INGENIERÍA DE TECNOLOGÍAS DE TELECOMUNICACIÓN

Técnicas de Aprendizaje Automático Para la Gestión de Recursos en Redes de Acceso Ópticas

Autor:
David Rodríguez Aragón
Tutor:
Dña. Noemí Merayo Álvarez

TÍTULO: **Técnicas de aprendizaje automático para la gestión de recursos en redes de acceso ópticas**
AUTOR: **David Rodríguez Aragón**
TUTOR: **Dña. Noemí Merayo Álvarez**
DEPARTAMENTO: **Teoría de la Señal y Comunicaciones e Ingeniería Telemática**

TRIBUNAL
PRESIDENTE: Rubén M. Lorenzo Toledo
SECRETARIO: Noemí Merayo Álvarez
VOCAL: J. Carlos Aguado Manzano
SUPLENTE: Ramón J. Durán Barroso
SUPLENTE: Ignacio de Miguel Jiménez

FECHA:
CALIFICACIÓN:

Resumen

Este Trabajo de Fin de Grado aborda la aplicación de técnicas de inteligencia artificial para la optimización de la gestión de recursos en redes de acceso ópticas basadas en la tecnología EPON (*Ethernet Passive Optical Networks*).

En particular, se ha desarrollado e implementado un modelo de red neuronal profunda capaz de predecir de forma dinámica el ancho de banda máximo asignable a cada ONU ciclo tras ciclo en escenarios 1G-EPON y 10G-EPON.

El proceso incluyó la generación de un dataset representativo mediante simulaciones, entrenamiento, integración en el módulo de la OLT (*Optical Line Terminal*) del simulador y validación del propio modelo mediante el análisis del retardo y el ancho de banda otorgado a cada usuario

Adicionalmente, se han incorporado nuevas funcionalidades al simulador con el fin de aumentar su realismo y su capacidad de análisis. Entre estas mejoras destacan la implementación de un nuevo generador de tráfico Pareto, la introducción de perfiles de usuarios con diferentes SLA y la adaptación del simulador EPON para soportar arquitecturas 25G-EPON.

Estas actualizaciones permiten analizar con mayor fidelidad el comportamiento de redes de nueva generación, contribuyendo a la evaluación de estrategias avanzadas de asignación dinámica de recursos.

Palabras clave

Redes PON, EPON, 1G-EPON, 10G-EPON,25G-EPON, Inteligencia Artificial, Aprendizaje Automático, Aprendizaje Profundo, Red Neuronal Profunda (DNN), Asignación Dinámica de Ancho de Banda (DBA), SLA (Service Level Agreement), Simulador de Redes Ópticas.

Abstract

This Final Degree Project explores the use of artificial intelligence techniques to optimize resource management in optical access networks based on EPON technology.

A deep neural network model has been designed and implemented to dynamically predict the maximum bandwidth that can be allocated to each final user on a cycle-by-cycle basis in both 1G-EPON and 10G-EPON environments. The methodology comprised the generation of a representative dataset through simulation, training and validation of the model, and its integration into the OLT module of the simulator. Performance was assessed by analyzing delay and the bandwidth assigned to each user, confirming the model's effectiveness under different load conditions.

Beyond the development of the AI model, several enhancements were introduced into the simulator to increase its realism and analytical scope. These include the implementation of a new Pareto-based traffic generator, the incorporation of user profiles with differentiated Service Level Agreements (SLAs), and the extension of the platform to support emerging 25G-EPON infrastructures.

Together, these improvements provide a more accurate framework for evaluating the performance of next generation access networks and pave the way for advanced strategies in dynamic resource allocation.

Keywords

Passive Optical Networks, 1G-EPON, 10G-EPON, 25G-EPON, Artificial Intelligence, Machine Learning, Deep Learning, Deep Neural Network (DNN), Dynamic Bandwidth Allocation (DBA), Service Level Agreement (SLA), Optical Network Simulator

Agradecimientos

A mis padres por apoyarme, ayudarme y ser un pilar fundamental en el día a día durante toda mi vida, permitiéndome dedicarle al estudio y al desarrollo de este proyecto el tiempo necesario y requerido por los mismos, ya que sin ellos nada de esto habría sido posible.

A mis amigos Jorge, Mario "Nori", Eduardo y Daniel por su apoyo y por aguantar mis conversaciones sobre el desarrollo de este proyecto y sus dificultades.

A mis amigos Daniel y Rubén, por el apoyo, diversión y comprensión mutua durante todas las etapas de la carrera y durante la realización de este proyecto.

A Carla, por apoyarme, animarme y darme la energía y determinación necesaria para seguir adelante, además de aguantar todas mis conversaciones sobre este proyecto.

A mi tutora Noemí, por su gran dedicación, ayuda, consejos y enseñanzas, además de su disposición a atender mis dudas por sencillas que fuesen en cualquier momento.

A Juan Pinto Ríos, por introducirme en el mundo de la IA y de las Redes Neuronales Profundas, además de su disponibilidad para enseñarme todo lo posible.

Este trabajo ha sido financiado por el Ministerio de Ciencia, Innovación y Universidades, la Agencia Estatal de Investigación y por FEDER/UE (proyecto PID2023-148104OB-C41 financiado por MICIU/AEI/10.13039/501100011033 y por FEDER/UE).

This work has been funded by Ministerio de Ciencia, Innovación y Universidades, Agencia Estatal de Investigación and by ERFD/EU (grant PID2023-148104OB-C41 funded by MICIU/AEI/10.13039/501100011033 and by ERFD/EU).

Índice

Índice de figuras

Índice de tablas

1

Introducción

1.1 Motivación

La creciente demanda de recursos de red por parte de los usuarios ha llevado a la expansión de la fibra óptica, especialmente en el tramo de acceso, donde las redes ópticas pasivas (PON, *Passive Optical Networks*) se han consolidado como una solución eficiente y escalable. Dentro de este tipo de redes PON, las redes 10G-PON (10 Gpbs) representan la evolución natural de las GPON (basadas en el estándar Gigabit) y EPON (basadas en el estándar Ethernet), ofreciendo mayores tasas de transmisión para responder a los nuevos requisitos de capacidad y rendimiento.

En este trabajo se emplea el entorno de simulación desarrollado por Víctor Herrezuelo en Python [1], el cual permite modelar redes EPON y 10G-EPON de forma flexible y con un alto grado de personalización.

El objetivo principal es implementar técnicas de Inteligencia Artificial (IA) para optimizar la asignación dinámica del ancho de banda entre los distintos usuarios que comparten la red de acceso. La IA se emplea para identificar patrones de tráfico y adaptar las decisiones de asignación de recursos, especialmente ancho de banda en tiempo real, con el fin de reducir el retardo y aumentar la eficiencia de ancho de banda en los niveles de carga más críticos desde la perspectiva de los operadores de red y proveedores de servicio. Esta estrategia permite comparar los resultados frente a métodos tradicionales de asignación, evaluando mejoras en términos de retardo y calidad del servicio. Así, el trabajo no solo explora nuevas posibilidades para la gestión inteligente de redes PON, sino que también sienta las bases para futuras implementaciones de control autónomo en entornos reales de redes ópticas de telecomunicaciones.

Adicionalmente, se persigue modificar la arquitectura del entorno de simulación inicial para que represente con mayor fidelidad el comportamiento de las redes PON actuales, incorporando la posibilidad de establecer diferentes SLA (*Service Level Agreement*) por usuario o grupo de usuarios, así como introducir fuentes de tráfico más realistas en el lado del usuario. Pero además se ha preparado el simulador para su extensión a redes 25G-PON, en línea con la tecnología que marcará la próxima generación de este tipo de redes de acceso.

1.2 Objetivos

1.2.1 Objetivos generales

Los objetivos generales de este trabajo son tres. En primer lugar, se llevará a cabo el análisis, diseño e implementación de un modelo de IA en el simulador de redes EPON, abarcando tanto entornos 1G-EPON como 10G-EPON. Para este fin, se emplearán técnicas de aprendizaje automático, con especial énfasis en redes neuronales, dada su alta capacidad para optimizar procesos complejos y su eficiencia en diversos escenarios y ámbitos.

En segundo lugar, se introducirá un nuevo generador de tráfico de Pareto (tráfico más realista en el lado del usuario) y se realizará una comparación con la fuentes de tráfico de Pareto original que estaba integrada en el simulador inicial.

En tercer lugar, se realizará la modificación de la arquitectura del simulador para permitir la introducción de diferentes SLAs dentro de las simulaciones y analizar el comportamiento del mismo sobre infraestructuras 25G-PON. Con ello, se dotará al simulador de la capacidad de realizar simulaciones más ajustadas a la realidad y evolución de estas redes PON.

1.2.2 Objetivos específicos

Se describen, a continuación, los objetivos específicos necesarios para la consecución de los objetivos generales ya comentados.

1. Estudiar y analizar el modelo de IA basado en redes neuronales más adecuado para el problema que se aborda en este proyecto.

2. Diseñar, entrenar e implementar el modelo de red neuronal sin alterar significativamente la estructura original del simulador. Dicha

implementación se llevará a cabo en una arquitectura de red EPON (1G-EPON) y una 10G-EPON. Analizar los resultados de las simulaciones tras la implementación de la red neuronal.

3. Implementar y evaluar nuevas funcionalidades sobre redes 10G-PON y 25G-PON, es decir, tanto las nuevas fuentes de tráfico de Pareto como la integración de acuerdos de nivel de servicio SLAs (*Service Level Agreements*).

1.3 Fases y metodología

A continuación, se describen todas las fases desarrolladas y la metodología aplicada en cada una de ellas durante la ejecución del presente trabajo.

1.3.1 Fase de análisis

En esta fase inicial, se busca la adquisición de los conocimientos fundamentales que permitan un desarrollo correcto del presente Trabajo de Fin de Grado:

- Análisis de la topología y de las principales características de las redes EPON, 10G-EPON y 25GPON.
- Revisión y análisis de la documentación correspondiente a las bibliotecas Scikit-Learn [8] y Joblib [14] de Python, con el fin de implementar y desarrollar el modelo de red neuronal.
- Proceso de familiarización con el simulador original, las redes neuronales y el lenguaje de programación Python, además de con las nuevas funcionalidades a implementar en este proyecto dentro de las arquitecturas PON.

1.3.2 Fase de implementación

Durante esta segunda fase se llevará acabo la instalación de las bibliotecas Scikit-Learn y Joblib en Python [2] para poder desarrollar todo lo relacionado con el modelo de red neuronal, realizar el entrenamiento de la red neuronal y para implementar el modelo de red neuronal dentro del simulador original sin realizar cambios significativos en su estructura.

Una vez instaladas las bibliotecas, se procederá con el diseño y programación del entrenamiento del modelo dentro del simulador EPON original [1] sin alterar de manera significativa su estructura, con el objetivo de que el simulador sea lo más flexible y escalable posible. También se realizará el mismo proceso sobre una arquitectura 10G-EPON.

Por último, se procederá a integrar las nuevas funcionalidades, en concreto el nuevo generador de tráfico de Pareto y la posibilidad de emplear diferentes SLAs, además de evolucionar el simulador hacia arquitecturas 25G-EPON.

1.3.3 Fase de pruebas

En esta fase final, se procederá a la realización de una serie de pruebas destinadas a evaluar el funcionamiento del simulador, haciendo uso del modelo de red neuronal previamente desarrollado. Estas pruebas se llevarán a cabo en distintos escenarios de red, abarcando tanto estructuras EPON (1G) como 10G-EPON, con el objetivo de analizar el comportamiento del sistema bajo diversas condiciones.

Asimismo, se efectuarán comprobaciones orientadas a verificar el correcto funcionamiento de las nuevas funciones implementadas para infraestructuras de mayor capacidad, concretamente 10G-EPON y 25G-EPON, garantizando así su operatividad e integración dentro del entorno simulado.

1.4 Estructura de la memoria del TFG

El Capítulo 2 describe las herramientas de trabajo y los recursos software que serán empleados a lo largo de todo el trabajo.

En el Capítulo 3 se analiza la evolución de la IA y las redes PON como tecnologías claves del proyecto. Ambas ofrecen soluciones eficientes y adaptativas para entornos complejos. Su combinación sienta las bases para optimizar redes de comunicaciones inteligentes.

En el Capítulo 4 se define y describe la estructura, proceso de entrenamiento y adaptación al simulador existente del modelo de IA basado en una red neuronal profunda (DNN, *Dense Neuronal Network*), así como su capacidad de aprender patrones complejos.

En el Capítulo 5 se describe el proceso de generación de un dataset representativo y variado, con el objetivo de entrenar eficazmente el modelo de IA, tanto en arquitecturas EPON como 10G-EPON. Se incorporan variaciones en los parámetros fundamentales, apoyadas por un sistema automatizado de captura y gestión de datos en formato .csv, optimizando así el análisis posterior.

En el Capítulo 6 se describe el proceso completo de entrenamiento de una red neuronal profunda en el entorno EPON y 10G-EPON. Se aplicarán diversas técnicas para optimizar el rendimiento predictivo del modelo. El resultado es un sistema entrenado y listo para ser integrado en futuras simulaciones sin necesidad de entrenamiento.

En el Capítulo 7 se describe el proceso técnico de implementación del modelo y su aplicación directa en la política de asignación de recursos del simulador. Además, se analizan los resultados del mismo para escenarios EPON y 10G-EPON.

En el Capítulo 8 se detalla la implementación de nuevas funcionalidades del simulador EPON. En concreto, un nuevo generador de tráfico Pareto en Python, la gestión de SLAs por usuario y la configuración para simulaciones de infraestructuras 25G-EPON. Finalmente, se analizan resultados comparativos de las diferentes fuentes de tráfico de pareto en 10G-EPON y 25G-EPON.

En el Capítulo 9 se recogen las conclusiones generales de este Trabajo de Fin de Grado y las líneas futuras que podrían seguir a este trabajo.

En el Capítulo 10 se recogen las referencias bibliográficas que han servido de apoyo para el entendimiento y realización de este trabajo.

2

Herramientas y recursos software

2.1 Introducción

Para el desarrollo de este TFG ha sido fundamental el uso de herramientas software que permitan simular entornos de red complejos y aplicar técnicas de gestión de recursos. La elección de las tecnologías empleadas responde a criterios de eficiencia, flexibilidad, accesibilidad y adecuación a los objetivos específicos del estudio.

En particular, se ha optado por el lenguaje de programación Python [3], dada su facilidad de uso y su extenso ecosistema de bibliotecas especializadas en simulación, análisis de datos y aprendizaje automático.

Asimismo, el entorno de desarrollo ha sido reforzado mediante el uso de un servidor de altas prestaciones proporcionado por el Grupo de Comunicaciones Ópticas, el cual ha permitido llevar a cabo simulaciones de manera intensiva y en paralelo, mejorando así los tiempos de ejecución.

Este apartado describe detalladamente las herramientas software empleadas, así como los recursos computacionales utilizados para apoyar el diseño, implementación y validación de la implementación del modelo de Inteligencia Artificial y las nuevas funcionalidades al simulador de redes EPON.

2.2 Python

Python es un lenguaje de programación de alto nivel, diseñado para priorizar la facilidad la lectura del código y reducir su coste en términos de mantenimiento [2] y [3].

Incluye programación orientada a objetos, funcional y estructurada que permite construir simuladores con código fácil de comprender.

La gran disponibilidad de bibliotecas, como *NumPy* [7] para cálculos numéricos eficientes, fortalece el análisis experimental...

Python es ideal para prototipado de simuladores científicos, facilitando iteraciones ágiles gracias a entornos como *VisualStudioCode* [4] o *Jupyter*. Además, al ser código abierto, garantiza la fácil integración con otros sistemas.

Por todo ello, *Python* es una opción potente, versátil y eficiente para desarrollar el simulador y evaluar estrategias.

2.3 Bibliotecas de Python

El lenguaje de programación *Python* ofrece un amplio ecosistema de bibliotecas especialmente relevantes para el desarrollo de algoritmos de asignación dinámica de ancho de banda, una tarea central en el simulador de redes PON empleado como base en este TFG.

Entre las herramientas más destacadas se encuentran aquellas orientadas al procesamiento de datos, la computación numérica y la implementación de modelos de aprendizaje automático. Bibliotecas como *pandas* [5] y *numpy* [7] permiten la gestión eficiente de estructuras de datos y operaciones matemáticas de alto rendimiento. Asimismo, *Scikit-learn* [8] proporciona un conjunto robusto de algoritmos de *Machine Learning* (aprendizaje automático) ampliamente validado.

Por otro lado, *torch* [12] facilita la construcción y entrenamiento de redes neuronales profundas gracias a su flexibilidad. Además, *tqdm* [13] permite la visualización en tiempo real del progreso de los entrenamientos, mientras que *joblib* [14] optimiza la serialización de modelos y la ejecución en paralelo de tareas.

El uso conjunto de estas bibliotecas potencia significativamente el desarrollo e implementación eficiente de estrategias inteligentes de asignación de recursos en redes PON.

2.4 Servidor de alto rendimiento destinado a las simulaciones

El acceso proporcionado al servidor Artemis del Grupo de Comunicaciones Ópticas (GCO), a través de una máquina virtual, representa una ventaja significativa para el desarrollo experimental de este TFG.

Dicha máquina virtual, alojada en dicho servidor, dispone de recursos computacionales significativamente superiores. En concreto dispone de 80 CPUs y 128 GB de memoria RAM. Estas características dotan a la máquina virtual de mayor capacidad de procesamiento, memoria RAM y almacenamiento, en comparación con un PC convencional.

Estas prestaciones permiten la ejecución paralela de múltiples simulaciones, lo que contribuye a una mejora sustancial en la eficiencia del flujo de trabajo. Además, la reducción en los tiempos de cómputo acelera notablemente la velocidad de las simulaciones experimentales, facilitando así el ajuste de parámetros y la validación de resultados. La disponibilidad de mayor espacio de almacenamiento también posibilita el manejo de grandes volúmenes de datos generados durante las simulaciones sin comprometer el rendimiento en gran medida.

En conjunto, el uso de esta infraestructura optimiza el desarrollo, la ejecución y el análisis de las simulaciones realizadas en este TFG. El acceso proporcionado al servidor del GCO, permite la realización de varias simulaciones a la vez, tener más almacenamiento y además hace que las simulaciones duren menos tiempo que en un PC normal debido a sus capacidades de potencia y memoria.

2.5 Conclusiones

El uso de *Python* como lenguaje base ha resultado fundamental para el desarrollo ágil y estructurado del modelo de inteligencia artificial y del simulador, gracias a su sintaxis clara y a su ecosistema de bibliotecas científicas.

Herramientas como *NumPy*, *Pandas*, *Scikit-Learn* o *Torch* han facilitado tanto el modelado matemático como la implementación de algoritmos de aprendizaje automático orientados a la asignación dinámica de ancho de banda.

Además, el acceso al servidor del Grupo de Comunicaciones Ópticas ha permitido ejecutar simulaciones intensivas con gran eficiencia computacional, reduciendo significativamente los tiempos de prueba y análisis.

La combinación de estos recursos ha optimizado el proceso de diseño, validación y evaluación de la implementación de un modelo de inteligencia artificial y de las nuevas funcionalidades sobre el simulador de redes EPON, permitiendo alcanzar los objetivos propuestos en este TFG.

3

Marco de trabajo: Inteligencia Artificial y Redes Ópticas Pasivas

3.1 Introducción

En el presente capítulo de la memoria se expone el estado del arte de los principales componentes en los que se sustenta la investigación de este TFG, con el objetivo de contextualizar su desarrollo.

En primer lugar, se analiza la evolución de la IA, desde los enfoques simbólicos basados en reglas hasta el paradigma actual del aprendizaje automático (*Machine Learning*) y su vertiente más avanzada, el aprendizaje profundo (*Deep Learning* [17]), basado en redes neuronales. Se destacarán sus fundamentos, capacidades y factores clave que impulsan su gran expansión en los últimos años.

En segundo lugar, se realiza introduce la arquitectura de las redes ópticas pasivas (*Passive Optical Networks*, PON), detallando su estructura punto a multipunto, el funcionamiento de los canales ascendentes y descendentes, así como los mecanismos de multiplexación y asignación dinámica de ancho de banda.

Esta revisión técnica permite comprender la base sobre la que se articula la solución propuesta en este trabajo.

3.2 Introducción a la IA y Machine Learning

La Inteligencia Artificial es una disciplina dentro de la informática cuyo propósito es automatizar tareas intelectuales que tradicionalmente requieren de intervención humana.

Desde sus inicios, la IA ha evolucionado a través de diferentes enfoques, destacando entre ellos la inteligencia artificial simbólica, basada en reglas explícitas

programadas por humanos (por ejemplo, las IAs que conforman los personajes que no son manejados por el jugador en videojuegos). Sin embargo, este enfoque resultó insuficiente para abordar tareas ambiguas y no estructuradas como la visión por computador o la comprensión del lenguaje natural, lo que condujo al surgimiento del aprendizaje automático (*Machine Learning*, ML) [15].

El aprendizaje automático propone una alternativa radical, en lugar de programar las reglas necesarias para resolver un problema, se entrena a un modelo utilizando grandes volúmenes de datos etiquetados. De esta forma, el sistema infiere por sí mismo patrones y relaciones estadísticas que le permiten generalizar y tomar decisiones ante datos no vistos. Este paradigma ha demostrado ser altamente eficaz en una amplia variedad de aplicaciones, desde el análisis predictivo hasta el diagnóstico médico asistido por ordenador.

Las redes neuronales artificiales constituyen la base de muchos algoritmos de aprendizaje automático. Se estructuran en capas de nodos o "neuronas", cada una de las cuales aplica transformaciones matemáticas a los datos mediante pesos ajustables y funciones de activación. Estas arquitecturas permiten modelar relaciones no lineales complejas, lo que las hace especialmente eficaces en tareas de predicción o reconocimiento de patrones.

En ese contexto emerge el aprendizaje profundo (*Deep Learning*), una subrama del ML que emplea redes neuronales profundas (DNN, *Deep Neural Network*) para aprender representaciones jerárquicas y abstractas de los datos. Estas redes se estructuran en múltiples capas, cada una de las cuales transforma sus entradas en formas más útiles para la tarea final. Este enfoque reduce la necesidad de realizar ingeniería de características manualmente, ya que el sistema es capaz de aprender de forma autónoma las representaciones óptimas a partir de los datos brutos [15].

El funcionamiento de las redes neuronales profundas se basa en el ajuste progresivo de millones de parámetros internos, a través de algoritmos de optimización. La retropropagación del error permite calcular como modificar cada peso en función del error cometido, lo cual guía el aprendizaje de manera eficiente. Inicialmente, los pesos de establecen de forma aleatoria y mediante iteraciones sucesivas sobre los datos se ajustas

para minimizar una función de pérdida que establece el error cometido entre las predicciones del modelo y los resultados esperados.

La expansión del Deep Learning en la última década ha sido posible gracias a la convergencia de tres factores fundamentales: El aumento exponencial de datos digitales disponibles, los avances de hardware para acometer procesamiento paralelo y el perfeccionamiento de los algoritmos de Deep Learning. En definitiva, el Deep Learning representa un cambio de paradigma en la forma en que los sistemas computacionales abordan la resolución de problemas complejos [15]. Al aprender directamente de los datos, sin necesidad de intervención humana en el diseño de reglas, las redes neuronales profundas se han consolidado como una herramienta fundamental en el desarrollo de sistemas inteligentes capaces de operar en entornos dinámicos y no estructurados. Por todos estos beneficios, hemos tomado este paradigma para su integración en nuestro TFG en la asignación de recursos en redes PON.

3.3 Introducción a las Redes PON

Una red de acceso PON es una arquitectura de red de acceso en fibra óptica que permite conectar un único punto de distribución (*OLT*, *Optical Line Termiantion*), con múltiples terminales de cliente conocidos como *Optical Network Units* (*ONU*). Esta conexión se realiza mediante elementos pasivos, es decir, componentes que no requieren alimentación eléctrica, como divisores ópticos (*splitters*), lo que significa su mantenimiento y reduce sus costes operativos [16].

Las redes PON presentan una topología punto a multipunto, también descrita como topología en árbol. En esta configuración, la OLT, situada habitualmente en las oficinas centrales del proveedor de servicio correspondiente, se comunica con varias ONUs mediante un *splitter* óptico pasivo (ver Figura 1). Este *splitter* cumple la función de dividir la señal óptica procedente de la OLT y distribuirla equitativamente entre las diferentes fibras ópticas que conectan con cada ONU que forma parte de la red. Gracias a este diseño, se consigue una arquitectura escalable y eficiente que permite dar servicio a múltiples usuarios desde una única infraestructura de transmisión.

A lo largo de esta memoria se utilizará el término ONUs (*Optical Network Units*) para referirse indistintamente a estas unidades, aclarando que tanto ONU como ONT (*Optical Network Terminal*) hacen referencia al mismo concepto.

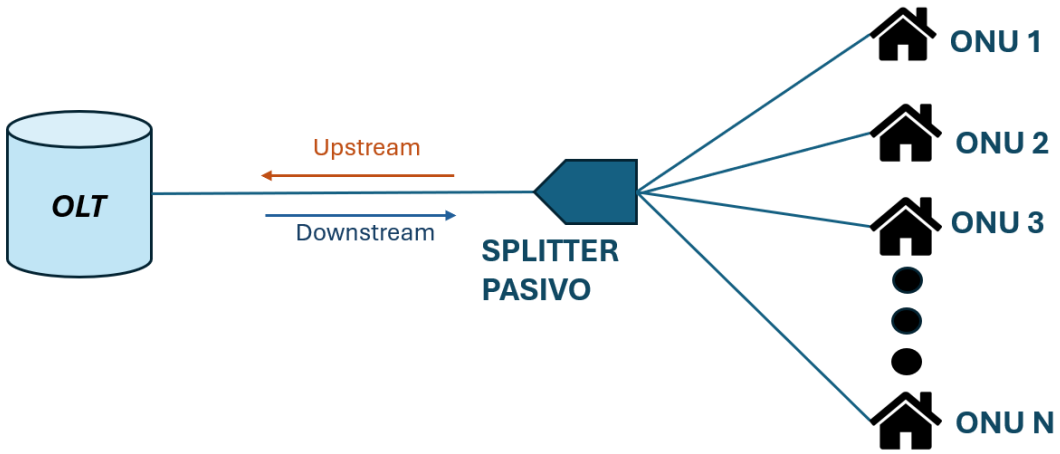


Figura 1: Esquema de una Red PON (*Passive Optical Network*) [Elaboración propia]

EEn el canal ascendente, es decir, en el sentido de transmisión desde la ONU hacia la OLT, la comunicación se establece bajo la lógica punto a punto. Dado que todas las ONUs comparten el mismo medio físico para enviar datos a la OLT, se emplea un mecanismo de acceso múltiple por división en el tiempo (*Time Division Multiplexing, TDM*). Este acceso es gestionado directamente por la OLT, que asigna a cada ONU una franja temporal específica para transmitir, evitando colisiones en el canal compartido. En redes EPON, la longitud de onda utilizada para el canal ascendente es de 1310 nm (segunda ventana), una de las bandas típicamente empleadas en redes ópticas por su bajo nivel de atenuación en la fibra.

En cuanto al canal descendente, que abarca la transmisión desde las OLT hacia las ONUs, la red se comporta como una topología punto a multipunto. En este caso, la OLT transmite de forma continua hacia todas las ONUs, y el *splitter* pasivo distribuye la señal óptica entre ellas, en un mecanismo de broadcast. Cada ONU, a su vez, es responsable de filtrar la información que le corresponde descartando todos aquellos datos destinados a otros terminales. La longitud de onda seleccionada para el canal descendente en redes EPON es de 1490 nm (tercera ventana).

Para una gestión eficiente del tráfico, las redes PON implementan algoritmos de asignación dinámica de ancho de banda, denominados DBA (*Dynamic Bandwidth Allocation*), basado en TDMA. Estos algoritmos permiten a la OLT distribuir de forma adaptativa el ancho de banda disponible entre las distintas ONUs, basándose en parámetros como la demanda de tráfico, la congestión de la red o los acuerdos de calidad de servicio

(*Quality of Service*, QoS) establecidos en la negociación entre proveedor de servicio y abonados. En el contexto de redes EPON, el estándar establece que el tiempo máximo de ciclo en el que la OLT asigna recursos a cada ONT/ONU es de 2 milisegundos. No obstante, cuando el nivel de tráfico es bajo, este ciclo se adapta dinámicamente. Dicho enfoque adaptativo optimiza el uso de los recursos de red, ajustándose a las condiciones de tráfico y contribuyendo a mejorar la experiencia del usuario final.

3.4 Conclusiones

A modo de resumen, se ha revisado el estado actual de los dos pilares fundamentales para el desarrollo del proyecto, la Inteligencia Artificial (IA) y las Redes Ópticas Pasivas (PON).

En el ámbito de la IA, se ha evidenciado una evolución significativa desde enfoques simbólicos hasta sistemas de aprendizaje profundo capaces de abordar problemas complejos mediante el procesamiento de grandes volúmenes de datos. Por otro lado, las redes PON se consolidan como una solución eficiente y escalable para el acceso a redes de alta velocidad, gracias a su arquitectura pasiva y a su gestión dinámica del ancho de banda.

La convergencia de ambas tecnologías permite avanzar hacia entornos más inteligentes, adaptativos y eficientes, en los que la optimización del rendimiento de la red puede beneficiarse de las capacidades predictivas y adaptativas de los modelos IA. Esta base tecnológica sienta el contexto necesario para el desarrollo y la implementación de soluciones innovadoras orientadas a mejorar la calidad del servicio y la experiencia de usuario en el ámbito de las redes de comunicaciones ópticas, en concreto en el ámbito de las redes de acceso PON.

4

Modelo de aprendizaje automático implementado en el simulador EPON

4.1 Introducción

En este capítulo de la memoria se describe el modelo de aprendizaje automático que se ha desarrollado e implementado en el marco de este proyecto, para abordar el problema de predicción de asignación de recursos en redes ópticas pasivas (PON). El propósito será optimizar la gestión del ancho de banda y adaptarse a las condiciones de tráfico y calidad de servicio de los usuarios de la red EPON.

Para dicha implementación, se seleccionó una red neuronal. Este modelo es especialmente adecuado para capturar relaciones complejas y no lineales entre múltiples variables de entrada y presenta una alta compatibilidad con la estructura del simulador original realizado en un TFG anterior [1], permitiendo su integración sin requerir modificaciones en la arquitectura inicial de carácter sustancial.

En este capítulo, se describe tanto la estructura interna del modelo como el proceso de entrenamiento y optimización de sus parámetros, así como los beneficios que aporta esta aproximación en términos de precisión y adaptabilidad. Este enfoque permite generar predicciones robustas del ancho de banda que debe asignarse a cada ONU/ONT, optimizando así el rendimiento del sistema y ajustándose a diversos escenarios realistas generados a través de la simulación.

4.2 Descripción de las redes neuronales profundas (*Dense Neural Network*, DNN)

En este TFG se ha desarrollado e integrado un modelo de Inteligencia Artificial basado en una DNN, que es un modelo de aprendizaje automático inspirado en el funcionamiento del cerebro humano [17], [19], [20].

Estas redes están formadas por múltiples capas de neuronas artificiales interconectadas entre sí que son capaces de transformar de manera progresiva los datos de entrada (*inputs*) lo que le permite aprender patrones complejos [17], [18]. Estos modelos son especialmente útiles en problemas donde las relaciones entre las variables que se tienen no son lineales, lo cual representa un problema complejo [18], [19].

Su funcionamiento se basa en ajustar progresivamente parámetros internos a partir de los datos de entrenamiento, con el objetivo de realizar predicciones cada vez más precisas. A continuación, se detalla con más profundidad la estructura y proceso de funcionamiento de una DNN.

4.2.1 Estructura de una DNN

Una DNN se compone fundamentalmente por tres tipos de capas que tendrán un número diferentes de neuronas según el problema que se quiere abordar [17], [20]:

- **Capa de Entrada (*Input Layer*):** Esta primera capa se encarga de recibir los datos originales (inputs) en formato vectorial y contendrá toda la información necesaria para iniciar el proceso de predicción. Cada neurona de esta capa representa una característica del conjunto de datos. Por ejemplo, en este proyecto, las entradas incluirán valores como la demanda de ancho de banda, la carga de la red asociada a la simulación y el identificador de la ONU, que se detallarán en secciones posteriores [18].
- **Capas Ocultas (*Hidden Layers*):** Estas capas son las encargadas del aprendizaje del modelo. Su función consiste en procesar y refinar la información procedente desde la capa de entrada mediante una serie de transformaciones. En las primeras capas ocultas, la red neuronal aprende a identificar patrones o características básicas que están presentes en los datos, a medida que la información avanza hacia las capas más profundas, estas combinan y reinterpretan los patrones que ya se

detectaron previamente, permitiendo a la red neuronal formar representaciones internas más complejas y de mayor nivel. Cada neurona de esta capa aplica una transformación matemática que incluye la multiplicación de los valores de entrada por un conjunto de pesos, la suma de un término de sesgo (constante que permite desplazar la función de activación para mejorar el ajuste del modelo) y la aplicación de una función de activación no lineal (operación que introduce no linealidad, permitiendo así a la red aprender relaciones complejas entre los datos) [17], [20]. Este proceso permite al modelo representar funciones complejas y capturar relaciones de alto nivel que no son evidentes a partir de una observación directa. En el presente proyecto, y tal y como se detallará más adelante, se ha configurado una capa oculta con 64 neuronas y dos capas ocultas con 64 neuronas en cada una, lo que permite a la red neuronal alcanzar un equilibrio entre capacidad de representación y eficiencia computacional.

- **Capa de Salida (*Output Layer*):** Constituye la etapa final del modelo y es la responsable de generar la predicción del sistema en función de la información procesada por las capas anteriores. Su número de neuronas depende directamente del tipo de problema que se quiera resolver. En el caso de este proyecto, la capa de salida está compuesta por una única neurona (tal y como se explicará más adelante), cuya activación producirá un valor numérico continuo que será el ancho de banda máximo que debe asignarse a cada ONU teniendo en cuenta las condiciones de la red en cada momento [19], [20]. Esta predicción será el resultado del aprendizaje acumulado durante el entrenamiento, durante el cual la red ajusta sus parámetros para minimizar el error entre las salidas generadas y los valores reales observados. Por lo tanto, la salida produce una síntesis optimizada de todas las entradas y patrones, con el objetivo de ofrecer una respuesta precisa y coherente.

4.2.2 Proceso de entrenamiento en redes DNN

El entrenamiento en este tipo de modelos se basa en un ciclo iterativo que tiene las siguientes fases [17]:

- **Propagación hacia delante (*Forward Propagation*):** Los datos de entrada son transmitidos a través de las distintas capas de la red neuronal. En cada neurona, se realiza la siguiente operación $Z = M * x + b$ donde M se encarga de

representar los pesos, *x* las entradas y b el sesgo. Posteriormente, el valor *Z* se transforma mediante una función $a = f(Z)$. Algunas de dichas funciones son *ReLU* (Unidad Lineal Rectificada), que devuelve el valor de entrada si este es positivo y un 0 en caso contrario, lo que es computacionalmente eficiente; o por ejemplo, la sigmoide (*Sigmoid*), que transforma cualquier valor real en un rango comprendido entre 0 y 1 y que puede causar saturación y ralentizar el proceso de aprendizaje. También nos encontramos con la tangente hiperbólica (*Tanh*), similar a la sigmoide, pero en este caso, transforma los valores de entrada en un rango entre -1 y 1. Este proceso se repite hasta alcanzar la capa de salida, donde se da la predicción del modelo [18], [20].

- **Cálculo de la Función de Pérdida (*Loss Function Calculation*):** La salida generada se compara con el valor real esperado mediante una función que mide el grado de error cometido. Las funciones más comunes son el error cuadrático medio (*MSE, Mean Square Error*) y el error absoluto medio (*MAE, Mean Absolute Error*). Esta métrica orienta el ajuste del modelo para mejorar su precisión [20].
- **Retropropagación y optimización (*BackPropagation and Optimization*):** Su objetivo es minimizar la función de pérdida global del modelo. Se calcula el gradiente de la función de pérdida con respecto a los pesos de la red utilizando el algoritmo de retropropagación. A continuación, se actualizar los pesos mediante un algoritmo de optimización como el descenso del gradiente $M = M - \eta * \frac{dP}{dM}$, donde η es la tasa de aprendizaje y el término $\frac{dP}{dM}$ el gradiente de la pérdida respecto a los pesos. Este proceso es repetido durante múltiples iteraciones permitiendo a la red mejorar su desempeño de forma progresiva [17], [20].

4.3 Configuración de una DNN en un escenario de red EPON

La red neuronal diseñada en nuestro simulador ha sido configurada para abordar escenarios en el contexto de redes EPON con el objetivo de gestionar el ancho de banda disponible [16]. Para ello, la red neuronal se configura en la capa de entrada con entradas correspondientes a tres parámetros, representados por $x_1 = B_{demand}$, $x_2 = Carga$ y $x_3 = ONU_{ID}$. Estos datos de entrada permiten caracterizar el estado actual de la

red y constituyen la base del proceso de aprendizaje del modelo. El objetivo de la red es predecir un único valor de salida, $y = B_{Max}$, el cual representa el ancho de banda máximo asignado en un instante determinado a cada usuario u ONU, en función de las condiciones de red observadas.

Se han empleado dos configuraciones distintas de capas ocultas, una arquitectura con una única capa oculta compuesta por 64 neuronas y otra arquitectura más profunda con dos capas ocultas cada una con 64 neuronas. Cada neurona en las capas ocultas aplica una operación matemática que incluye la multiplicación de los valores de entrada por sus respectivos pesos, la adición de un sesgo y la posterior aplicación de una función de activación no lineal (en el caso de este proyecto *ReLU y Tanh*) [18], [20]. Así pues, la configuración final de nuestra DNN para la configuración de recursos en una red PON se muestra gráficamente en la Figura 2.

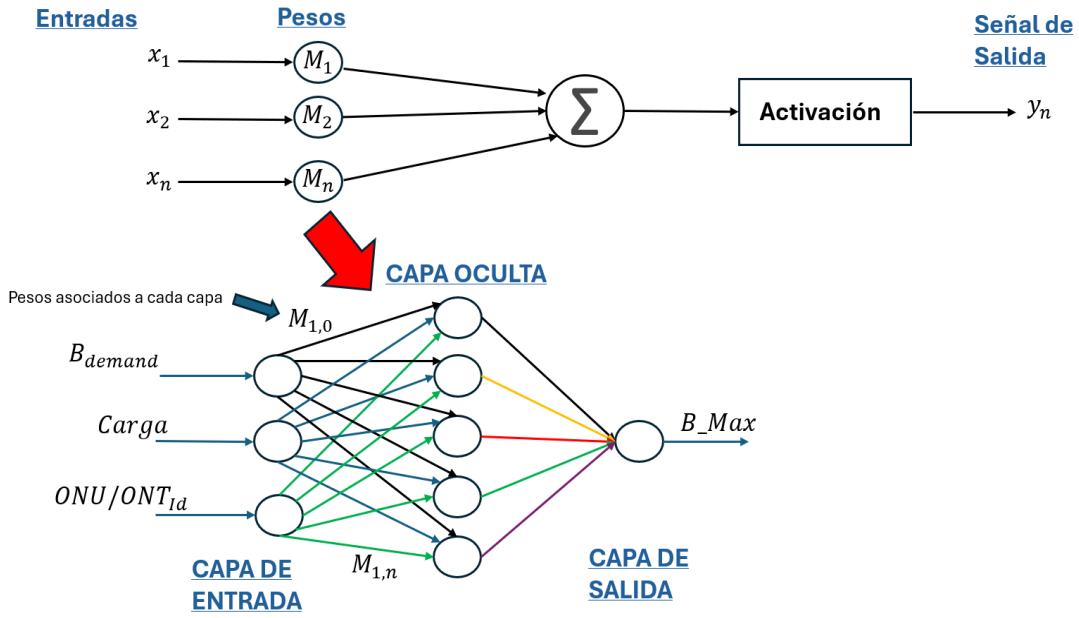


Figura 2: Configuración de una DNN para la asignación de recursos en una red EPON

El uso de redes neuronales profundas en la asignación de recursos de redes ópticas, como es en nuestro caso el ancho de banda asignado a cada ONU, permite una anticipación a las necesidades reales de los usuarios de la red, mejorando la eficiencia del sistema [16], [17], [19]. Gracias a su capacidad para identificar patrones en los datos, este modelo permite realizar asignaciones más precisas y adaptativas, incluso en condiciones de

simulación cambiantes. Su capacidad de generalización y flexibilidad lo convierte en una herramienta fundamental para abordad desafíos en entornos de redes PON.

4.4 Conclusiones

La aplicación de una red neuronal profunda (DNN) en este proyecto podrá permitir abordar con éxito el problema de asignación de ancho de banda en redes PON. Este modelo podría tener la capacidad de adaptarse a diferentes escenarios y condiciones de simulación, ofreciendo resultados precisos y coherentes. Además, su estructura facilita una integración sencilla en el sistema original, sin necesidad de abordar grandes modificaciones. El proceso de entrenamiento permite que el modelo aprenda de los datos y mejore su rendimiento de manera progresiva.

En este capítulo se ha descrito el modelado de la red neuronal para predecir el ancho de banda máximo que se le puede dar a cada usuario (ONU) de forma dinámica en función de las condiciones de la red EPON. En definitiva, este modelo de red neuronal puede representar un avance significativo para la gestión eficiente de recursos en redes ópticas, haciendo posible una optimización más justa y optimizada del ancho de banda disponible.

5 Generación y preparación del dataset de entrenamiento en entornos de redes EPON

5.1 Introducción

El diseño de un modelo de red neuronal DNN para entornos de redes EPON requiere no solo la arquitectura adecuada, sino también un conjunto de datos de entrenamiento que represente con fidelidad tanto la complejidad como la variabilidad de lo que se quiere modelar. En este proyecto, centrado en redes EPON y 10G-EPON, se ha considerado fundamental la creación de un *dataset* que incorpore condiciones realistas, incluyendo distintas cargas de tráfico, configuraciones de red y perfiles de demanda representativos.

Con este fin se ha realizado una copia del archivo *OLT.py* perteneciente al simulador original desarrollado en el TFG realizado por Víctor Herrezuelo [1], y sobre dicha copia, denominada como *OLT_IA.py*, se han llevado a cabo las modificaciones necesarias. En este archivo se gestionan las tareas de procesamiento de los mensajes de control *Report* enviados por cada ONU a la OLT, así como la asignación de recursos y ancho de banda a las distintas ONUs en cada ciclo de la simulación. La información correspondiente es transmitida desde la OLT mediante mensajes de control denominados *Gate*.

Este capítulo detalla el conjunto de acciones realizadas para adaptar el simulador base EPON, incluyendo la introducción de variabilidad en determinados parámetros, nuevas métricas de evaluación y configuraciones específicas para diferentes topologías de red. Asimismo, se crea un sistema automatizado de recopilación y estructuración de datos, orientado a facilitar su posterior uso en procesos de entrenamiento. El propósito final es garantizar que el modelo de red neuronal adquiera la capacidad de generalizar su comportamiento ante situaciones diversas, mejorando su aplicabilidad en escenarios reales.

Este capítulo establece, por tanto, las bases sobre las que se construye la base de datos empleada en las fases posteriores de desarrollo de este proyecto.

5.2 Algoritmo DBA para la asignación de ancho de banda implementado en el simulador EPON

Nuestro simulador de redes EPON implementa un algoritmo de asignación de ancho de banda DBA que sigue un esquema limitado. En este esquema, el OLT asigna ancho de banda a una ONU/ONT para el ciclo siguiente una vez recibe su mensaje de control (mensaje *Report*) con la demanda para el ciclo siguiente (esto es, con el estado de sus colas). La OLT transmite la información de asignación de ancho de banda a cada ONU mediante el mensaje de control *Gate*, especificando tanto el intervalo de tiempo asignado (slot) como el instante exacto en el que debe iniciar su transmisión. Este proceso se enmarca en un esquema TDMA, dado que todas las ONUs comparten un único canal de subida y requieren una coordinación estricta para evitar colisiones. Para que una o varias ONTs no monopolicen el ancho de banda disponible en un ciclo, se aplica un esquema limitado a dicha asignación a través de la variable *B_alloc* en nuestro simulador (ver Figura 3). En concreto, la variable *B_alloc* representa el ancho de banda que la OLT asigna a una ONU concreta en un ciclo determinado (en bits). Esta asignación se realiza en función de tres variables principales:

- ***B_demand[ont_id]:*** Demanda real de la ONU.
- ***B_max[ont_id]:*** Ancho de banda máximo permitido en ese ciclo para esa ONU, acorde a su SLA correspondiente. En este caso, solo se trabaja con un único SLA con un peso asignado *w_sla* = 1.
- ***tamano_report:*** Tamaño del mensaje de control (*REPORT*), que corresponde a 64 bytes según el estándar EPON.

El cálculo de *B_alloc* se define como lo que se muestra en la Figura 3.

```
self.B_alloc[ont_id] = min(self.B_demand[ont_id], self.B_max[ont_id]) + tamano_report
```

Figura 3: Asignación de Ancho de Banda para cada ONU por parte de la OLT

Para la asignación dinámica del ancho de banda, lo primero que realiza el algoritmo DBA es conocer el ancho de banda que demanda la ONU, el cual se recibe en el mensaje de *Report* (*MensajeReport* en el código del simulador). Este ancho de banda demandado

se calcula mediante la suma del tamaño de todas las colas de la ONU en cuestión de la siguiente manera $B_{Demand} = \sum Q_i$ donde Q_i se refiere al tamaño de las colas perteneciente a la ONU i.

Por otro lado, es preciso conocer el ancho de banda máximo asignable a cada ONU en un ciclo dependiendo del SLA que tenga asociado, el cual es fijo en cada ciclo. Como en el caso abordado en la primera parte del proyecto solo se tiene un SLA, todas las ONUs tendrán el mismo ancho de banda máximo. Este ancho de banda máximo se calcula mediante la siguiente ecuación:

$$B_{Max}\,[onu_{id}] = \frac{B_{Available} * W_{Sla}}{\sum W_{sla} * N_{ONUs}}$$

donde N_{ONUs} es el número de ONUs que hay asociadas al SLA, W_{Sla} es el peso asociado al SLA, $B_{Available}$ es el ancho de banda máximo disponible en cada ciclo.

Una vez se tienen calculados estos dos parámetros, se procede a la explicación de la asignación de ancho de banda en cada ciclo. Para ello, cuando a la OLT le llega un mensaje de control (llamado *Report*) de una ONU en cada ciclo, se compara el ancho de banda demandado por la ONU con el máximo que se puede asignar a dicha ONU según el SLA definido para la misma. En concreto se aplica el siguiente esquema limitado:

- Si $B_{demand} \leq B_{Max}$ indica el ancho de banda demandado por la ONU es menor que el máximo para esa ONU. Por tanto, el ancho de banda asignado será el demandado por dicha ONU más el tamaño del mensaje de control (*tamano_report*), ya que en el ancho de banda asignado se tiene que guardar un cierto ancho de banda para que la ONU transmita dicho mensaje de control en el ciclo siguiente, $B_{alloc}[ont\_id] = B_{demand}\,[\mathrm{ont_{id}}] + tamano\_report$.
- Si $B_{demand} > B_{Max}$ indica el ancho de banda demandado por la ONU es mayor que el máximo para esa ONU. Por tanto, el ancho de banda será el máximo asignado para dicha ONU según el SLA correspondiente más el tamaño del mensaje de control (*tamano_report*), $B_{alloc}[\mathrm{ont_{id}}] = B_{Max}[\mathrm{ont_{id}}] + tamano\_report$. Esta asignación máxima, hará que se limite el ancho de banda asignado a una ONU en un ciclo, de manera que no monopolice todo el canal y por lo tanto todo el ancho de banda disponible en el ciclo máximo disponible (2 ms en el estándar EPON).

De esta manera el ancho de banda será asignado a cada ONU dependiendo de la demanda que tenga en sus colas. La expresión de la Figura 3 garantiza que *B_alloc* nunca supere ni la demanda de la ONU (*B_demand[ont_id]*) ni el ancho de banda máximo establecido por su SLA en ese ciclo (*B_max[ont_id]*), es un esquema limitado.

La clave de esta definición de *B_alloc[ont_id]* está en la naturaleza no determinista de *B_max[ont_id]*, la cual incorpora una variación aleatoria controlada y que se explicará en el apartado siguiente. Esto se hace con el fin de simular la dinámica real de una red PON donde las asignaciones de *B_alloc[ont_id]* no son siempre idénticas, aun bajo condiciones similares de tráfico.

Esta modificación tiene un impacto directo en la calidad del *dataset* que se va a generar. Al variar el valor de *B_max[ont_id]* en cada ciclo, se evita que el modelo de inteligencia artificial memorice patrones fijos, y en su lugar, se le entrena para adaptarse a diferentes variaciones y restricciones del entorno. Esto mejora su capacidad de generalización, ya que aprende a establecer *B_alloc[ont_id]* no solo en función de la demanda, sino también bajo condiciones variables de calidad de servicio de cada ONU.

Además, esto mejora la representatividad del simulador, generando datos más próximos a los que se esperarían en una red PON real. Con este enfoque, *B_alloc[ont_id]* se convierte en una variable más rica y representativa de escenarios realistas, ya que refleja las políticas reales de la OLT y ofrece al modelo de inteligencia artificial información rica y diversa durante el entrenamiento.

Finalmente, los valores de *B_alloc[ont_id]* se almacenan en la base de datos, y formarán parte del conjunto de datos utilizado para entrenar el modelo. Su inclusión permite construir un modelo más robusto, capaz de aprender bajo escenarios variados más adaptados a redes PON reales, reflejando con mayor fidelidad el comportamiento real de la red.

5.3 Implementación de variabilidad en el ancho de banda máximo en el algoritmo DBA

Uno de los principios clave del aprendizaje automático es que el modelo necesita ser expuesto a una amplia gama de situaciones y comportamientos posibles del entorno que está tratando de modelo, con el fin de generalizar correctamente ante nuevos datos.

En el contexto de este TFG, y tal y como se muestra en la Figura 2, la variable que representa el ancho de banda máximo asignado a una ONU en cada ciclo (*B_max*, en el Código), constituye la variable objetivo que el modelo debe aprender a predecir. Sin embargo, si los valores utilizados durante el entrenamiento se generan siempre de la misma manera, la red neuronal podría memorizar patrones fijos, en lugar de conseguir una adaptación a variaciones reales del entorno.

Por lo tanto, se necesita introducir una cierta variabilidad en esta variable *B_max*, contribuyendo a que el modelo de IA desarrolle una mayor capacidad de generalización. Además, permitirá emular de manera más precisa la naturaleza dinámica de la red PON, donde diversos factores pueden provocar que la asignación óptima del ancho de banda no sea siempre la misma, especialmente porque la naturaleza del tráfico es rafagosa.

Una vez conocida la importancia de introducir dicha variabilidad, se procede a la introducción de esta en el simulador original desarrollado por Víctor Herrezuelo [1], dentro del archivo *OLT_IA.py*. Dichas pruebas se han hecho para condiciones de redes PON de 1Gbps y para redes PON de 10G. Para ello, se realiza una modificación en el momento de la asignación del ancho de banda máximo, esto es la variable *B_max*:

- **Para EPON**: La variabilidad viene introducida por *+(random.randrange(-8,8) * 10000)* tal y como se ve en la Figura 4. Esta instrucción de *Python* añade una variación aleatoria al *B_max* siendo esta variación una entre -80000 bits y +70000 bits (ya que se excluye al 8). Este valor, posteriormente será almacenado en la base de datos (*DataSet*) que será empleada para entrenar el modelo de IA de este proyecto.

```
self.B_max.append((B_AVAILABLE*self.w_sla[i])/sum(self.w_sla) +(random.randrange(-8, 8) * 10000))
```

Figura 4: Implementación de variabilidad en *B_max* en casos de 1GPON

- **Para 10G-EPON**: Se introduce la variabilidad del *B_max* de la misma manera que en el caso de GPON (ver Figura **5**), con el único cambio de que ahora la variabilidad se realiza entre -800000 y + 700000 bits, ya que se tienen un orden más en la tasa de transmision de la red (10G).

```
self.B_max.append((B_AVAILABLE*self.w_sla[i])/sum(self.w_sla) +(random.randrange(-8, 8) * 100000))
```

Figura 5: Implementación de variabilidad en *B_max* en casos 10GPON

5.4 Creación de la base de datos de entrenamiento

Una vez comprendido el proceso de asignación dinámica del ancho de banda y aplicadas las modificaciones previamente descritas, cuyo propósito ha sido introducir variabilidad y enriquecer la representatividad de los datos simulados frente al comportamiento real de una rede PON, se procede a la generación del dataset de entrenamiento. Este conjunto de datos constituye un elemento fundamental para el posterior entrenamiento del modelo, ya que contiene la información estructurada necesaria para que el sistema aprenda a inferir patrones de comportamiento.

Para tal fin, se han desarrollado e integrado en el archivo *OLT_IA.py* distintas funcionalidades específicas orientadas a la creación automatizada de archivos en formato .csv (*Comma-Separated Values*). Estos archivos almacenan los datos de manera ordenada, permitiendo una lectura eficiente y estandarizada por parte de bibliotecas comunes en entornos de aprendizaje automático. Además, los archivos generados se almacenan de forma centralizada en una única carpeta del sistema, lo que facilita su organización, trazabilidad y uso posterior durante las fases de entrenamiento, validación y prueba del modelo. Esta estructura automatizada asegura la coherencia del proceso y permite escalar el sistema con nuevos datos de forma ágil.

5.4.1 Creación de los archivos de la base de datos

Para la creación de estos archivos, se añaden las siguientes líneas al código (*OLT_IA.py*) para poder generar una carpeta donde almacenar de forma estructurada todos los ficheros .csv que sean generados durante la simulación y también darle un nombre para todos los ficheros .csv, tal y como se observa en la Figura 6.

```
# Creamos la carpeta donde se guardarán los archivos CSV si no existe
carpeta_resultados = 'RND_resultados_csv_L05_Bmax_V2_0.8'
if not os.path.exists(carpeta_resultados):
    os.makedirs(carpeta_resultados)

# Generamos el nombre del archivo CSV con un marcador de tiempo
nombre_archivo = f'valores_{time.strftime("%Y%m%d_%H%M%S")}.csv'

# Combinamos la ruta de la carpeta con el nombre del archivo
ruta_completa = os.path.join(carpeta_resultados, nombre_archivo)
```

Figura 6: Fragmento de código de *OLT_IA.py* donde se realiza la creación de archivos .csv

La primera línea *carpeta_resultados = 'RND_resultados_csv_L05_Bmax_V2_0.8'* es utilizada para darle nombre a la carpeta, con la variable *carpeta_resultados* que contiene una cadena de texto, donde se almacenarán todos los archivos, según la carga de tráfico de cada simulación de la que se quieran obtener datos, el término final (el sufijo 0.8) deberá ser ajustado manualmente estableciendo la carga a la que se corresponda según la simulación. En concreto, la carga es el parámetro define la carga de cada ONU de la red utilizada para generar paquetes. A este parámetro se le puede asignar cualquier valor comprendido entre 0 y 1, en saltos de 0.1 (desde 0.1 hasta 0.9).

Posteriormente se evalúa mediante el uso de la función de Python *os.path.exists()*. Si ya existe, se corresponderá con una carpeta con el nombre del directorio que contiene *carpeta_resultados* y si no existe la función *os.makedirs()* la crea, evitando así que el programa falle al intentar guardar un archivo en una carpeta inexistente.

Por último, se crea un nombre para cada archivo .csv que se genere durante una simulación. Este nombre se crea dinámicamente usando el formato *valores_YYYYMMDD_HHMMSS.csv* donde *%Y* es el año, *%m* el mes, *%d* el día, *%H* la hora, *%M* los minutos y *%S* los segundos del momento en el que se genera el archivo, lo que asegura que cada archivo tendrá un nombre único basado en la hora de creación de este, evitando sobrescribir archivos anteriores. Por ejemplo, si la simulación se realiza el 23 de junio de 2025 a las 17:18:39 horas, el archivo que se genera en ese momento tendrá como nombre *valores_20250623_171839.csv*.

Finalmente, la función *os.path.join()* concatena las partes de la ruta (*carpeta_resultados* + *nombre_archivo*). El resultado es la ruta completa del archivo, por ejemplo: *RND_resultados_csv_L05_Bmax_V2.08/valores_20250623_171839.csv.*

5.4.2 Recopilación de datos en el dataset

Todas las modificaciones de código relacionadas con la generación de archivos .csv se implementan en el fichero OLT_IA.py que se ha explicado previamente. Con el fin de cumplir con el objetivo de recopilar datos en los archivos .csv que se generan se añadieron las siguientes líneas de código que pueden observarse en la Figura 7.

```
# Guardamos los valores de B_alloc y n_alloc en el archivo CSV
with open(ruta_completa, mode='a', newline='') as file:
    writer = csv.writer(file)
    if file.tell() == 0:
        writer.writerow(['Carga','Onu_id', 'B_demand_bits', 'B_demand_MBS', 'B_max_bits', 'B_max_MBS','B_alloc_bits', 'B_alloc_MBS',
                         'B_guaranteed', 'error_max_aloc', 'error_demand_alloc', 'error_max_demand'])

    writer.writerow([CONFIG_CARGA,ont_id, self.B_demand[ont_id], ((self.B_demand[ont_id]*(10**-6))/(T_AVAILABLE))/8,
                     self.B_max[ont_id], ((self.B_max[ont_id]*(10**-6))/(T_AVAILABLE))/8, self.B_alloc[ont_id],
                     ((self.B_alloc[ont_id]*(10**-6))/(T_AVAILABLE))/8,((self.B_alloc[ont_id]*(10**-6))/(T_AVAILABLE))/8,
                       self.B_guaranteed[ont_id], (self.B_max[ont_id]-self.B_alloc[ont_id]),(self.B_max[ont_id]-self.B_demand[ont_id])])
```

Figura 7: Fragmento de código de *OLT_IA.py* donde se realiza la recopilación de datos.

La primera instrucción *with open(ruta_completa, mode='a', newline=' ') as file* crea un archivo cuya ruta está especificada en *ruta_completa*, dicho archivo se abre en modo *append (mode='a', adición)* asegurándose así que los nuevos datos sean añadidos al final para no sobrescribir el contenido ya existente y *newline=' '* evita la inserción de líneas en blanco adicionales al escribir el archivo .csv. Por último, *as file* asigna el objeto archivo que devuelve *open()* a una variable llamada *file* dentro del *bloque with*; por tanto, *file* es el nombre local que se usa dentro del *bloque with* para interactuar con el archivo abierto.

La función de *Python csv.writer(file)* crea un objeto escritor csv que se encarga de escribir listas de Python como filas de un archivo .csv, separando de manera automática los diferentes campos con comas. *file* tal y como ha sido explicado previamente, es el archivo abierto por *open()*.

La tercera línea, *if file.tell() == 0* hace uso de la función de Python *tell()*, la cual se encarga de devolver la posición del cursor dentro del archivo en bytes. Si devuelve un 0, significa que el archivo está vacío o recién creado. Esta línea es la encargada de decidir si se escribe la cabecera con los nombres de cada uno de los valores que se quieran recopilar, los cuales serán escritos mediante la cuarta línea *writer.writerow([...])*.

Por último, la línea *writer.writerow([...])* se ejecuta siempre que la función *tell()* no devuelva un 0 y se encarga de escribir una única fila en el archivo .csv utilizando la lista de valores que estén incluidos entre los corchetes. Cada valor es escrito separado por comas de los otros y estos podrán ser numéricos o cadenas de texto.

El código explicado previamente se ejecuta dentro del método *procesa_report* de la clase OLT, el cual es invocado cada vez que una ONU envía un mensaje de control *Report*. Estos mensajes son generados de forma periódica por las ONUs durante la simulación, en ciclos consecutivos. Para cada segundo de simulación, se genera o se

reutiliza, si ya existe, un archivo .csv cuyo nombre incluye una marca temporal con precisión de segundo. Cada vez que se recibe un *MensajeReport*, se abre el archivo correspondiente al segundo actual se añade una nueva fila con los datos extraídos de dicho *MensajeReport* (Véase 5.4.3). De este modo, se construye un archivo .csv por segundo de simulación, que se va completando dinámicamente a medida que las ONUs generan mensajes durante ese intervalo. Esta estrategia permite una segmentación temporal de los datos, facilitando así su trazabilidad y organización.

Una vez finalizada la simulación para una determinada carga de red, todos los archivos generados durante ese experimento se almacenan en una carpeta específica asociada a dicha configuración. Posteriormente, todos los archivos .csv de esa carpeta se combinan en una única base de datos unificada para esa carga. Además, una vez generadas las bases de datos individuales de cada carga, todas ellas se integran en un *DataSet* final que reúne el conjunto completo de escenarios simulados. Ambos procesos de consolidación se llevan a cabo mediante el script *creador_bbdd_v4.py*, cuyo funcionamiento será detallado en apartados posteriores (Véase 5.6).

5.4.3 Descripción de los campos del Dataset

En este apartado se aborda cuáles y en qué consisten los datos que se recopilan en cada fila de cada archivo .csv que se genera y que es determinado por la lista de valores que se le da a la función *writer.writerow([…]).*

La lista de valores se observa en la Figura 8 entre corchetes dentro de ambas funciones *writer.writerow([…])*. Por tanto, los datos más importantes y que posteriormente se emplearán en el entrenamiento, que se recopilan dentro de cada línea de un archivo .csv y que se generan durante las simulaciones son:

1. ***Carga:*** Representa el nivel de carga de tráfico del cada ONU utilizado en la simulación. Este valor viene definido por la variable *CONFIG_CARGA* (ver archivo *configuration.py*) y es un número decimal entre 0 y 1 que representa la carga de cada ONU. Por ejemplo, un valor de 0.8 representa una carga del 80%. Este valor de carga es el mismo en media para todas las ONUs ya que se considera tráfico simétrico en todas las ONUs.

2. ***Onu_id:*** Identificador numérico único asignado a cada ONU que está enviando el mensaje *Report* mediante la variable *ont_id* dentro del simulador. Esto relaciona cada línea de datos generada con la ONU que lo ha generado mediante el *MensajeReport*.
3. ***B_demand_bits:*** Cantidad total de bits demandados por la ONU en el instante de tiempo correspondiente. Este valor se calcula como la suma del tamañao de todas las colas de tráfico de esa ONU en el momento de emitir el mensaje *Report*. Este valor viene dado por la variable *self.B_demand[ont_id]* en el simulador.
4. ***B_max_bits:*** Representa la cantidad máxima de bits que la OLT asigna a la ONU en el ciclo correspondiente. Este valor viene dado por la política de planificación de la OLT (en el contexto de este proyecto se emplea un esquema limitado). Establece el máximo número de bits que se le puede proporcionar a una ONU en un ciclo, incluso si esta ONU demanda más. Se establece en la variable *self.B_max[ont_id]* en el simulador.

El resto de los valores como *B_demand_MBS*, *B_max_MBS*, *B_alloc_bits*, *B_alloc_MBS*, *B_guaranteed*, *Error_max_alloc*, *Error_demand_alloc* y *Error_max_demand* únicamente se integraron para tener la posibilidad de realizar análisis más profundos en el comportamiento del simulador en análisis futuros. En siguientes revisiones del simulador EPON el número de valores se reducirá y estos valores serán eliminados.

```
    writer.writerow(['Carga','Onu_id', 'B_demand_bits', 'B_demand_MBS', 'B_max_bits', 'B_max_MBS','B_alloc_bits', 'B_alloc_MBS',
                     'B_guaranteed', 'error_max_aloc', 'error_demand_alloc', 'error_max_demand'])

writer.writerow([CONFIG_CARGA,ont_id, self.B_demand[ont_id], ((self.B_demand[ont_id]*(10**-6))/(T_AVAILABLE))/8,
                 self.B_max[ont_id], ((self.B_max[ont_id]*(10**-6))/(T_AVAILABLE))/8, self.B_alloc[ont_id],
                 ((self.B_alloc[ont_id]*(10**-6))/(T_AVAILABLE))/8,((self.B_alloc[ont_id]*(10**-6))/(T_AVAILABLE))/8,
                   self.B_guaranteed[ont_id], (self.B_max[ont_id]-self.B_alloc[ont_id]),(self.B_max[ont_id]-self.B_demand[ont_id])])
```

Figura 8: Fragmento de Código de *OLT_IA.py* donde se muestran los campos que tendrá el dataset.

5.5 Configuración del simulador EPON para la obtención del dataset

En este apartado se tratarán las diferentes configuraciones del simulador EPON para extraer los datos en los casos de estudio referentes a este trabajo [23]. En concreto, se va a realizar una base de datos para el caso de redes EPON (1G) y otra base de datos para

redes 10G-EPON. A continuación, se procede a describir los parámetros que se han escogido para ello.

Para cada una de las configuraciones EPON (EPON y 10G-EPON), se han realizado 10 simulaciones, en las que se ha evaluado el rendimiento del sistema bajo diferentes niveles de carga de las ONUs, variando desde 0.1 hasta 0.9 en incrementos de 0.1. No obstante, entre los valores de carga 0.6 y 0.7, se utilizó un incremento más fino de 0.05, incluyendo específicamente la carga intermedia de 0.65. A modo de aclaración, para un entorno de red 10G-EPON, una carga de 0.1 indica que cada ONU está transmitiendo a una velocidad de 100 Mbps, considerando que la velocidad máxima de transmisión por ONU es de 1 Gbps. Para todas las cargas se ha elegido un tiempo de simulación de 300 s, para no tener una base de datos excesivamente grande. A continuación, se procede a la descripción del resto de parámetros que se quiera extraer datos de una red ethernet PON (EPON) o una red 10G-EPON.

5.5.1 Recopilación de datos de entrenamiento para una red EPON

En este caso se emplean los mismos parámetros empleados en la simulación para una red Ethernet PON (EPON) de 1G del trabajo realizado por Víctor Herrezuelo [1] que se muestran en la Tabla 1.

Parámetros (Archivo parameters.py)	**Valor del Parámetro**
Número de ONUs	16 ONUs
Tasa de Transmisión de la red (*R_tx*)	1 Gbit/s
Tasa de Transmisión máxima de cada ONU	100 Mbits/s
Longitud de la Red (OLT – ONT)	20 km
Período del Ciclo	2 milisegundos ($2 * 10^{-3}s$)
Tiempo de Guarda	5 microsegundos ($5 * 10^{-6}s$)

Tamaño del Buffer	10 MBytes
Tamaño de los Paquetes	Paquetes de 64, 594 y 1500 Bytes
Número de Colas (Servicios)	1 Cola (1 único servicio)

Tabla 1: Configuración para la recopilación de datos de entrenamiento en casos EPON

Todos los archivos .csv que se generan durante cada uno de los barridos realizados serán recopilados de manera estructurada en carpetas con el nombre *RND_resultados_csv_L05_Bmax_V2_carga*, siendo *carga* la carga empleada para cada una de las simulaciones (establecido en *configuration.py*). Por lo tanto, tendríamos finalmente una única carpeta que recopilaría todos los archivos .csv generados durante el barrido para cada una de las cargas de la ONU. Todas estas carpetas (10 en total) serán finalmente almacenadas en una carpeta final denominada *BBDD_CSV_1G300s*.

5.5.2 Recopilación de datos de entrenamiento para una red 10G-EPON

En este caso se emplean los mismos parámetros de simulación para una red 10G-EPON que fueron empleados por Víctor Herrezuelo en su TFG [1], y que se muestran en la Tabla 2.

Parámetros (Archivo Parameters.py)	**Valor del parámetro**
Número de ONUs	16 ONUs
Tasa de Transmisión de la red (*R_tx*)	10 Gbit/s
Tasa de Transmisión máxima de una ONU	1 Gbit/s
Longitud de la Red (OLT – ONT)	20km
Período del Ciclo	2 milisegundos
Tiempo de Guarda	1 microsegundo

Tamaño del Buffer	100 MBytes
Tamaño de los paquetes	Paquetes de 64, 594 y 1500 Bytes
Número de Colas (Servicios)	1 Cola

Tabla 2: Configuración para la recopilación de datos de entrenamiento en casos 10G-EPON

Todos los archivos .csv que se generan durante cada uno de los barridos serán recopilados de manera estructurada en carpetas con el nombre *RND_resultados_csv_L05_Bmax_V2_carga*, siendo *carga* la carga empleada para cada una de las simulaciones (establecido en *configuration.py*). Por lo tanto, tendríamos una carpeta que recopilaría todos los archivos .csv generados durante el barrido para cada una de las cargas. Todas estas carpetas (10 en total) serán finalmente almacenadas en una carpeta final denominada *BBDD_CSV_10G300s*.

5.6 Creación del Dataset completo

Una vez extraídos todos los archivos .csv necesarios, se procede a la creación de la base de datos que se utilizará para entrenar el modelo de aprendizaje automático basado en una red neuronal y desarrollado en este proyecto.

Aunque se crearán dos bases de datos distintas, una para 1G-EPON y otra para 10G-EPON, el código utilizado para su generación es común en ambos casos, y su funcionamiento general es equivalente, con diferencias únicamente en aspecto puntuales.

Para este fin, se ha desarrollado un script en *Python* denominado *creador_bbdd_v4.py*, ubicado en la carpeta *Modelo_Funcional* del simulador. Su función principal es concatenar todos los archivos generados durante los barridos descritos en el Apartado 5.5. Estos archivos se encuentran organizados en las carpetas que contienen en su interior tanto *BBDD_CSV_1G300s como BBDD_CSV_10G300s,* correspondientes a los datos recogidos para EPON y 10G-EPON, respectivamente. La ejecución de este script da lugar a una base de datos consolidad, que será utilizada en la fase posterior de entrenamiento del modelo.

Dicho código se muestra en la Figura 9. El código define en primer lugar una función destinada a validar y limpiar los datos numéricos del *DataFrame*. Esta función

emplea las siguientes funciones de *Python*: *df.dropna()* que elimina todas las filas que contengan valore NaN, *df = df.apply(pd.to_numeric, errors='coerce')*, y es empleada para intentar convertir todos los elementos del *DataFrame* a valores numéricos. Si no se puede convertir ningún valor lo reemplazará por NaN. Por último, se vuelve a emplear *df.dropna()* para eliminar cualquier fila que tenga valores NaN tras la conversión realizada. Como resultado del uso de estas tres instrucciones el *DataFrame* final solo contendrá valores numéricos válidos.

Posteriormente se define en *carpeta* el directorio donde se tienen almacenadas todas las carpetas con los archivos .csv para cada una de las cargas (*BBDD_CSV_1G300s* para 1G-EPON y *BBDD_CSV_1G300s* para 10GPON), y además se crea una lista denominada *dataframes* que almacenará los *DataFrames* resultantes de leer cada uno de los archivos. La siguiente sección de código (Figura 9) realiza una iteración sobre los archivos presentes en el directorio siguiendo el siguiente proceso.

```
import os
import pandas as pd
# Función para verificar y limpiar los datos del DataFrame
def limpiar_datos(df):
    # Eliminar filas con valores NaN o no numéricos
    df = df.dropna()
    df = df.apply(pd.to_numeric, errors='coerce')
    df = df.dropna()
    return df
# Carpeta donde se encuentran los archivos CSV
carpeta = 'BBDD_CSV_10G100s'
# Lista para almacenar DataFrames de cada archivo CSV
dataframes = []
# Buscar todos los archivos CSV en la carpeta
for archivo in os.listdir(carpeta):
    if archivo.endswith('.csv'):
        ruta_archivo = os.path.join(carpeta, archivo)
        # Leer el archivo CSV y mostrar su nombre al usuario
        print(f"Leyendo archivo: {archivo}")
        #valor_carga = float(input("Ingrese el valor de carga: "))
        # Cargar el archivo CSV en un DataFrame y agregar la carga
        df = pd.read_csv(ruta_archivo)
        # Agregar el DataFrame a la lista
        dataframes.append(df)
# Combinar todos los DataFrames en uno solo
df_final = pd.concat(dataframes, ignore_index=True)
# Escribir el DataFrame final en un archivo CSV
archivo_final = 'BBDD_5.csv'
df_final.to_csv(archivo_final, index=False)
print(f"Se ha creado el archivo CSV final: {archivo_final}")
```

Figura 9: Archivo *creador_bbdd_v4.py* empleado para crear Bases de Datos.

En primer lugar, se utiliza la función de *Python os.listdir(carpeta)* para listar todos los nombres de los archivos contenidos en la carpeta. A continuación, se aplica el método *archivo.endswith('.csv')*, que se emplea para filtrar únicamente aquellos archivos que

tienen extensión .csv. A continuación, *os.path.join(carpeta, archivo)* se usa para construir la ruta completa de cada archivo .csv. Posteriormente, se carga cada archivo como un *DataFrame* utilizando *pd.read_csv(...)*. Finalmente, con *dataframes.append(df)* se añade cada *DataFrame* a la lista general que se creó previamente.

Una vez leídos todos los archivos se procede a consolidar los datos en un único *DataFrame* empleando la función de Python *df_final = pd.concat(dataframes, ignore_index=True)*. Esta función genera un nuevo objeto denominado *df_final*, que contiene todos los registros provenientes de los distintos archivos. El parámetro *ignore_index=True* asegura que los índices del nuevo *DataFrame* resultante se reasignen de forma secuencial y sin ser duplicados. A continuación, se asigna un nombre a la base de datos que se obtiene como salida en *archivo_final*, y posteriormente se emplea la función *df_final.to_csv(archivo_final, index=False)* con el objetivo de exportar el *DataFrame* consolidado a un nuevo archivo .csv con el nombre que contenga *archivo_final*. El parámetro *index=False* evita exportar la columna de índices como una columna adicional en el archivo .csv que se genera. Una vez generadas las bases de datos, es posible comprobar la cantidad de muestras (líneas) que contiene cada una mediante comandos de líneas de comandos. En concreto:

- En el Terminal de *Linux*, se puede utilizar el comando *wc -l NombreBBDD.csv* donde *NombreBBDD.csv* representa el nombre del archivo correspondiente. Este comando devuelve el número total de líneas que tiene el archivo .csv incluida la cabecera. Por tanto, se debería restar una unidad si se quiere obtener el número de muestras.
- En entornos *Windows*, específicamente en *PowerShell*, puede usarse el comando *(Get-Content .\NombreBBDD.csv).Count*. De igual forma, se debe considerar si el archivo incluye una línea de cabecera para restarle una unidad.

Así pues, finalmente la base de datos para 10G-EPON tiene 39.794.842 muestras y la base de datos para EPON (1G) tiene 33.964.268 muestras.

5.7 Conclusiones

La generación del *dataset* constituye un componente esencial en el desarrollo del modelo de aprendizaje automático basado en una red neuronal, al proporcionar la base sobre la cual se entrena su capacidad de predicción y generalización. A lo largo de este

apartado se han detallado las modificaciones implementadas sobre el simulador original [1], permitiendo la inclusión de comportamientos más realistas mediante la introducción de variabilidad en parámetros críticos como *B_max* y, en consecuencia, *B_alloc* (ancho de banda asignado en cada ciclo). Esto permite representar de forma más fiel las dinámicas propias de una red EPON, en la que factores como la congestión, la variabilidad de tráfico entre otros influyen directamente en el rendimiento del sistema.

Asimismo, se ha desarrollado una infraestructura automatizada y estructurada para la recolección, almacenamiento y organización de datos en formato .csv, facilitando su posterior tratamiento durante el proceso de entrenamiento. La segmentación de los datos por niveles de carga y la inclusión de múltiples variables asociadas a cada ciclo de simulación permite enriquecer la base de datos. Esto permite exponer al modelo a una amplia gama de condiciones operativas, fundamentales para mejorar su capacidad de adaptación a escenarios reales.

En conclusión, las acciones descritas en este apartado establecen una base metodológica para la construcción de un *dataset* robusto y representativo, indispensable para el éxito de las fases posteriores de construcción del modelo de IA para este proyecto.

6 Proceso de entrenamiento de la red neuronal en el simulador EPON

6.1 Introducción

En el ámbito del aprendizaje automático, el entrenamiento de los modelos basados en redes neuronales profundas (*DNN*), constituye una herramienta clave para abordar problemas donde la relación entre las variables no es lineal.

El entrenamiento del modelo que se explica en este capítulo, implementado en un programa de *Python*, tiene como objetivo desarrollar un modelo predictivo capaz de estimar el valor de una variable *B_max_bits* (valor máximo de bit asignado a una ONU en cada ciclo) a partir de un conjunto de datos de entrada, en concreto, *Carga*, *Onu_i*d y *B_demand_bits*. Para ello, se emplea un modelo de red neuronal multicapa proporcionado por la bibilioteca *scikit-learn*, [8] en combinación con técnicas de preprocesamiento y selección de hiperparámetros que optimizan su rendimiento predictivo. El flujo del script se desarrolla en varias fases, que abarcan desde la preparación y normalización de los datos de entrada, la división del conjunto de datos, la definición del espacio de búsqueda de hiperparámetros, el entrenamiento y la validación cruzada mediante búsqueda en rejilla (*Grid Search*), hasta la persistencia del modelo entrenado y del normalizado. A continuación, en los siguientes apartados se procede a detallar de manera rigurosa cada uno de estos componentes.

6.2 Implementación y descripción funcional del script para el entrenamiento de la red neuronal

El script completo se muestra en la Figura 10 y en la Figura 11. A continuación, se procede a su explicación detallada, paso a paso y por bloques, en las siguientes secciones.

```python
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.preprocessing import MinMaxScaler
from sklearn.neural_network import MLPRegressor
from tqdm import tqdm
import joblib
import warnings
# Suprimir todos los warnings de numpy
warnings.filterwarnings('ignore', category=np.VisibleDeprecationWarning)
# Cargar datos desde el archivo CSV
data = pd.read_csv("BBDD_5.csv",delimiter=',')
# Seleccionar las columnas de entrada y salida
X = data[["Carga","Onu_id","B_demand_bits"]]
y = data["B_max_bits"]
# Normalizar los datos
scaler = MinMaxScaler()
X_scaled = scaler.fit_transform(X)
# Dividir los datos en conjuntos de entrenamiento y prueba
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size=0.2, random_state=42)
# Definir el espacio de búsqueda de hiperparámetros
param_grid = {
    'hidden_layer_sizes': [(64,), (64, 64)],
    'activation': ['relu', 'tanh'],
    'solver': ['adam', 'sgd'],
    'max_iter': [10, 50]
}
# Definir el modelo DNN
model = MLPRegressor()
# Realizar la búsqueda en cuadrícula
with tqdm(total=len(param_grid['hidden_layer_sizes']) * len(param_grid['activation']) *
          len(param_grid['solver']) * len(param_grid['max_iter'])) as pbar:
    grid_search = GridSearchCV(model, param_grid, cv=3, verbose=3, n_jobs=-1)
    grid_search.fit(X_train, y_train)
    pbar.update(1)
# Obtener el mejor modelo
best_model = grid_search.best_estimator_
```

Figura 10: Script de Python para el entrenamiento de la DNN Parte 1

```python
# Guardar el mejor modelo entrenado
joblib.dump(best_model, "mejor_modelo_dnn_v5.pkl")
# Si se aplica, guardar el scaler
if scaler is not None:
    joblib.dump(scaler, "scaler_v5.pkl")
print("Mejor modelo guardado como 'mejor_modelo_dnnv3.pkl'.")
```

Figura 11: Script de Python para el entrenamiento de la DNN Parte 2

6.2.1 Importación de Librerías

Este bloque (ver Figura 12) importa las dependencias necesarias para la correcta ejecución del script de Python. En particular, se importan las librerías:

- *Pandas:* Biblioteca para la manipulación y análisis de *DataFrames*, como los contenidos en los archivos .csv [5].

- *Numpy:* Biblioteca para la realización de operaciones numéricas eficientes sobre matrices y vectores [7].
- Sklearn.model_selection: Aporta herramientas para la división del dataset y prueba (*train_test_split)*, así como para explorar distintas combinaciones de parámetros de manera automáticas (*GridSearchCV*) [9].
- *Sklearn.preprocessing:* Permite la transformación de características numéricas en un rango normalizado [10].
- *Sklearn.neural_network import MLPRegressor:* Provee de un modelo de red neuronal para regresión [11].
- *Tqdm:* Aporta una opción para el seguimiento en tiempo real del proceso [13].
- *Joblib:* Permite guardar modelos entrenados en disco para poder usarlos más adelante sin necesidad de repetir el entrenamiento [14].
- *Warnings:* Maneja las advertencias, evitando así mensajes innecesarios en la consola durante la ejecución [21].

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.preprocessing import MinMaxScaler
from sklearn.neural_network import MLPRegressor
from tqdm import tqdm
import joblib
import warnings
```

Figura 12: Librerías empleadas en el script de entrenamiento de la DNN

6.2.2 Supresión de warnings y carga de datos

La primera línea de esta parte del *script* (ver Figura 13) *warnings.filterwarnings('ignore', category=np.VisibleDeprecationWarning)*, filtra los *warnings* del tipo *VisibleDepreciationWarning*, que son emitidas por *NumPy* cuando se utilizan listas de diferente longitud. Así se evitan ruidos "visuales" que aparezcan por consola, que no afectan al funcionamiento del modelo.

```
# Suprimir todos los warnings de numpy
warnings.filterwarnings('ignore', category=np.VisibleDeprecationWarning)

# Cargar datos desde el archivo CSV
data = pd.read_csv("BBDD_5.csv",delimiter=',')
```

Figura 13: Extracto del script de entrenamiento donde se tratan los warnings y se carga la base de datos.

La instrucción *data = pd.read_csv("BBDD_5.csv",delimiter=',')* realiza la lectura del archivo CSV, en este caso denominado *BBDD_5.csv*. Esta base de datos es interpretada como una estructura *DataFrame* que contiene las variables necesarias para entrenar el modelo. El delimitador especificado es la coma, que es el formato convencional en archivos CSV.

6.2.3 Selección de variables de entrada y de salida

De todas las variables que contiene la base de datos generada en este proyecto, se eligen *Carga*, *Onu_id* y *B_demand_bits* (ver Figura 14) como variables predictoras y se forma con ellas una matriz de características (*x*), compuesta por tres columnas numéricas.

Por otro lado, en la variable *y* se crea el vector objetivo, que representa la variable que se quiere predecir, es decir, la salida del modelo. En este caso tal y como ha sido explicado en apartados anteriores se desea estimar un *B_max_bits* que asignar a cada ONU en cada ciclo y de forma dinámica. Este paso es crucial, ya que delimita el dominio de entrada y de salida del modelo de aprendizaje que estamos creando en este proyecto.

```
# Seleccionar las columnas de entrada y salida
X = data[["Carga","Onu_id","B_demand_bits"]]
y = data["B_max_bits"]
```

Figura 14: Extracto del script donde se realiza la selección de las variables.

6.2.4 Normalización de las características de entrada

En este paso se aplica una transformación de escalado de los datos, mediante la función *MinMaxScaler()*; en concreto sobre las variables predictoras (o características) tal y como se observa en la Figura 15. Esta transformación lineal lleva cada variable predictora (aquellas de entrada) al rango [0,1], de acuerdo con la fórmula $X_{scaler} = \frac{X - X_{Min}}{X_{Max} - X_{Min}}$ (para cada una de las variables). El objetivo de estas instrucciones es evitar que variables con escalas mayores al rango [0,1] dominen el aprendizaje. Además, la normalización ayuda a mejorar la convergencia del algoritmo de entrenamiento basado en una red neuronal.

```
# Normalizar los datos
scaler = MinMaxScaler()
X_scaled = scaler.fit_transform(X)
```

Figura 15: Extracto del script donde se emplea el *scaler*

6.2.5 División del conjunto de datos para el entrenamiento

Una vez normalizados los datos, es fundamental dividirlos en dos grupos diferentes para garantizar que el modelo aprenda correctamente. Esta línea que se observa en la Figura 16 realiza precisamente esa separación. Así pues, los datos se dividen como:

- *X_scaled:* Son los datos de entrada (características), ya escalados entre 0 y 1, tal y como se explicó en el apartado anterior (6.2.4).
- *Y:* Es el valor que el modelo debe predecir, ancho de banda máximo asignado.

A continuación, la función de *Python train_test_split* toma los datos y los divide en dos subconjuntos, en concreto:

- *Conjunto de Entrenamiento (X_train, y_train):* Representa el 80% de los datos y es el conjunto que se emplea para enseñar al modelo como relacionar las entradas con la salida.
- *Conjunto de Prueba (X_test, y_test):* Representa el 20% de los datos, no se usa durante el entrenamiento. Se guarda para evaluar si el modelo realmente aprendió o si simplemente memorizó los datos.

Esto es importante, ya que, si se entrenara y evaluara el modelo con los mismos datos no se podría saber si es capaz de generalizar bien ante situaciones nuevas. Separar los datos permite simular un caso real, en el que el modelo se enfrenta a datos que no había visto previamente.

```
# Dividir los datos en conjuntos de entrenamiento y prueba
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size=0.2, random_state=42)
```

Figura 16: Extracto del script donde se dividen los datos

Por último, es importante elegir aleatoriamente que datos van al conjunto de entrenamiento y cuáles van al conjunto de prueba. Por tanto, si se ejecuta este mismo código dos veces, se obtendría una división distinta cada vez. Para que la división de los datos sea la misma siempre que se ejecute el código se emplea un argumento *random_state=42*, que sirve para la división aleatoria de datos sea siempre la misma cada vez que se ejecute el script.

6.2.6 Definición del espacio de hiperparámetros

En este paso se define una lista de combinaciones de valores para diferentes parámetros de la red neuronal que el modelo va a probar para ver cuál funciona de una manera óptima (ver Figura 17). A este proceso se le denomina espacio de búsqueda de hiperparámetros (*param_grid*), en otras palabras, no se entrena solo un modelo, sino muchos, cambiando los valores de los hiperparámetros para detectar la mejor combinación que obtenga mejores resultados [17]. En concreto, en nuestro modelo de red neuronal se ajustarán los siguiente hiperparámetros:

- *Hidden_layer_sizes:* Define el tamaño de la red neuronal, aquí se indica cuántas capas ocultas (intermedias entre entrada y salida) tiene la red neuronal y cuantas neuronas hay en cada capa. En nuestro caso tomamos una sola capa de 64 neuronas *(64,)* y dos capas con 64 neuronas en cada una *(64,64)*. Estas capas son las responsables de aprender los patrones. En general, más capas y neuronas permiten aprender patrones más complejos, pero también pueden hacer que se tarde más en entrenar o se vuelva más difícil de ajustar correctamente.
- *Activation:* Define la función de activación, funciones que se usan dentro de cada neurona y permiten que la red no se limite a aprender relaciones lineales y pueda aprender relaciones no lineales. En nuestro caso hemos probado con *ReLU y Tanh* La función *ReLU*, deja pasar los valores positivos y establece los valores negativos a cero. La función *Tanh* transforma los valores en un rango entre -1 y 1.
- *Solver:* Define el método para ajustar el modelo (optimizador), este valor indica la estrategia que va a usar el modelo para aprender, es decir, para ajustar sus pesos. En nuestra búsqueda de hiperparámetros integramos *Adam y Sgd* (*Descenso de Gradiente Estocástico*). *Adam* ajusta los valores del modelo de forma adaptativa y eficiente. Funciona bien con muchos tipos de datos. Por su parte, *Sgd* es una técnica directa para aprender, va ajustando poco a poco los valores con cada ejemplo de entrenamiento.
- *Max_iter:* Establece el número máximo de iteraciones (número de épocas) que se permite al modelo pasar por los datos para seguir aprendiendo. Para la realización de este proyecto se ha establecido [10,50], 10 son muy pocas veces y sirve para probar rápidamente, mientras que 50 permite más entrenamiento, pero también tarda más y puede sobreajustarse demasiado a los datos y al sistema.

```
# Definir el espacio de búsqueda de hiperparámetros
param_grid = {
    'hidden_layer_sizes': [(64,), (64, 64)],
    'activation': ['relu', 'tanh'],
    'solver': ['adam', 'sgd'],
    'max_iter': [10, 50]
}

# Definir el modelo DNN
model = MLPRegressor()
```

Figura 17: Extracto del script donde se define *param_grid*

La última línea de la Figura 17, *model=MLPRegressor()*, es la encargada de crear el modelo base que se va a emplear en el caso de este proyecto, es decir, una red neuronal tipo *MLPRegressor*, diseñada para predecir valores numéricos continuos. *MLP* significa *Multi-Layer Perceptron*, o sea, una red neuronal con varias capas de neuronas interconectadas entre sí. El término *Regressor* indica que se va a usar para resolver un problema de predicción de números reales, como en nuestro caso (*B_max_bits*).

6.2.7 Optimización del modelo mediante búsqueda de hiperparámetros con Grid Search

La técnica conocida como *Grid Search* constituye un método de búsqueda exhaustiva dentro del espacio definido por el parámetro *param_grid*. Este procedimiento evalúa de forma sistemática todas las combinaciones posibles de hiperparámetros especificadas, con el objetivo de identificar aquella configuración que maximice el rendimiento del modelo. Esta estrategia resulta especialmente útil en entornos donde la selección óptima de hiperparámetros puede tener un impacto significativo sobre la capacidad predictiva del modelo entrenado. En nuestro contexto, el bloque de código correspondiente a esta técnica representa uno de los componentes clave dentro del proceso de entrenamiento del modelo de aprendizaje automático. Es precisamente en este fragmento del *script* donde se realiza también la validación cruzada, se ajustan los modelos y se elige el mejor estimador para ser utilizado posteriormente.

Este bloque de código puede observarse en la Figura 18. En concreto, la función *GridSearchCV* es una función de *Python* que toma el modelo base, en el caso de este proyecto *MLPRegressor*, prueba todas las combinaciones de parámetros que han sido definidas en *param_grid* y para cada combinación entrena el modelo y evalúa su rendimiento usando una técnica de validación cruzada (*CV, Cross-Validation*).

```
# Realizar la búsqueda en cuadrícula
with tqdm(total=len(param_grid['hidden_layer_sizes']) * len(param_grid['activation']) *
          len(param_grid['solver']) * len(param_grid['max_iter'])) as pbar:
    grid_search = GridSearchCV(model, param_grid, cv=3, verbose=3, n_jobs=-1)
    grid_search.fit(X_train, y_train)
    pbar.update(1)
```

Figura 18: Extracto del código donde se realiza el entrenamiento de la DNN

La validación cruzada es una técnica utilizada para garantizar que el modelo no solo funcione adecuadamente con un subconjunto de datos, sino que también mantenga un rendimiento consistente en diferentes particiones del conjunto de entrenamiento. En este estudio se emplea la validación cruzada con *cv=3*, lo que implica dividir el conjunto de entrenamiento en tres particiones del mismo tamaño. En cada iteración, el modelo se entrena con 2 de las particiones y se evalúa con la tercera, rotando esta última en cada iteración. Este proceso se repite tres veces, y los resultados obtenidos en cada iteración se promedian para estimar la eficacia general de la combinación de hiperparámetros evaluada.

Esto se repite para cada combinación posible de los hiperparámetros que se han definido. A pesar de que en muchos casos se recomienda utilizar validaciones cruzadas de 5 o incluso 10 particiones para obtener estimaciones más robustas del rendimiento del modelo, en este proyecto se ha optado por *cv=3* por motivos prácticos relacionados con la memoria del entorno de simulación (ya que nuestro *dataset* de entrenamiento es muy grande). Además, no es tan necesario este proceso en conjuntos de datos que tienen un número elevado de muestras, como es nuestro caso de estudio. En concreto y debido al tamaño de la base de datos empleada, se identificó que valores más altos de *cv* provocaban problemas de sobrecarga de memoria en el servidor donde se llevaban a cabo las simulaciones y la misma terminaba cancelándose. Por tanto, *cv=3* representa un compromiso razonable entre una evaluación cruzada válida y una ejecución estable sin interrupciones por falta de memoria.

Por otro lado, la instrucción *tqdm(...)* añade una barra de progreso visual, la parte *total=...* calcula el número total de combinaciones, en nuestro caso mostrará el avance sobre un total de 16 combinaciones. El parámetro *verbose=3* indica que *GridSearch* informe paso a paso sobre lo que está realizando, y cuánto mayor sea el número, más detallados son los mensajes. En este caso, se imprime información útil sobre el progreso y

cada combinación que se está probando. Finalmente, el último parámetro *n_jobs= -1* indica que se deben usar todos los núcleos del procesador disponibles para realizar el trabajo en paralelo. Esto permite que el ordenador realice varias pruebas al mismo tiempo acelerando así el proceso total.

6.2.8 Identificación y elección del modelo de red neuronal de mejor desempeño

Una vez completada la búsqueda de combinaciones de hiperparámetros con validación cruzada, el siguiente paso es seleccionar el mejor modelo entrenado y guardar tanto el modelo como el escalador de datos, para poder reutilizarlos en el futuro sin necesidad de volver a entrenar. Tal y como se observa en la Figura 19, existen dos partes diferenciadas que se corresponden con las siguientes acciones:

- Selección del Mejor Modelo: Se realiza mediante la instrucción de *Python best_model = grid_search.best_estimator*, que extrae el modelo que obtuvo el mejor rendimiento durante la búsqueda en cuadrícula de hiperparámetros. En este caso, como se trata de un problema de regresión (es decir, predicción de valores numéricos continuos), la métrica utilizada para seleccionar el mejor modelo es el coeficiente de determinación R^2. Este es un indicador que mide qué tan bien consigue el modelo reproducir los valores reales a partir de las entradas. Para interpretarlo, siempre compara con un modelo muy simple, aquel que ignora las variables de entrada y predice siempre el mismo valor fijo, ese valor fijo corresponde a la media de todos los valores reales de la variable que queremos predecir (en este caso *B_max*) en el conjunto de datos. Por ejemplo, si los valores reales de salida tuvieran una media de 100, ese modelo de referencia sería siempre 100 para cualquier caso, sin importar los datos de entrada que le pasen.

  De esta forma, el R^2 nos dice cuánto mejor es nuestro modelo respecto a esa estrategia básica de "predecir siempre el promedio", un $R^2 = 1$ significa que el modelo predice perfectamente sin error, un R^2 indica que el modelo no mejor nada respecto a predecir siempre la media y un $R^2 < 0$ refleja que el modelo lo hace incluso peor que predecir siempre la media. Lo interesante de R^2 es que no solo refleja el tamaño de los errores, sino también cuánta variabilidad total de los datos logra explicar el modelo. Si los valores reales varían mucho alrededor de su media, un buen modelo debería ser capaz de captar esa variación. Cuanto más se

acerque el R^2 a 1, mayor será la proporción de variabilidad explicada y mejor será la capacidad predictiva del modelo.

En este trabajo, *GridSearchCV* prueba todas las combinaciones de hiperparámetros definidas en la cuadrícula y calcula el valor medio de R^2 para cada una. Finalmente, selecciona la configuración que logra el mayor R^2, garantizando que el modelo elegido no solo se ajusta bien a los datos de entrenamiento, sino que también generaliza de manera adecuada a nuevos datos. Ese modelo es el que se guarda como definitivo y queda listo para su uso en predicciones.

- Guardado del Mejor Modelo: Se realiza mediante la instrucción de *Python joblib.dump(best_model, "mejor_modelo_dnn_v5.pkl")*, que guarda el mejor modelo entrenado en un archivo denominado *mejor_modelo_dnn_v5.pkl* usando la biblioteca *joblib*. Esto permite guardar el entrenamiento realizado y cargarlo en cualquier momento sin tener que volver a realizar un entrenamiento. Los archivos con extensión .pkl (*pickle file*) son archivos binarios que guardan objetos de *Python* ya procesados, de forma que se pueda cerrar el programa y volver a usar el objeto más adelante, sin necesidad de crearlo desde cero.
- Guardado del Escalador: Se guarda el escalador de datos (*MinMaxScaler*) que se utilizó para normalizar las variables de entrada, tal y como se explicó en el Apartado 6.2.4. Esto es muy importante porque cuando se quiera hacer una predicción con nuevos datos, esos datos han de ser escalados de la misma forma que durante el entrenamiento realizado. Guardar el escalado garantiza que eso pueda hacerse de manera correcta.

```
# Obtener el mejor modelo
best_model = grid_search.best_estimator_

# Guardar el mejor modelo entrenado
joblib.dump(best_model, "mejor_modelo_dnn_v5.pkl")

# Si se aplica, guardar el scaler
if scaler is not None:
    joblib.dump(scaler, "scaler_v5.pkl")

print("Mejor modelo guardado como 'mejor_modelo_dnnv3.pkl'.")
```

Figura 19: Extracto del script donde se elige el mejor modelo entrenado y se guarda.

En resumen, este bloque finaliza el proceso, dejando guardados el mejor modelo y el escalador empleado para escalar los datos. Gracias a este proceso, el modelo puede ser

aplicado fácilmente a nuevos datos más adelante, sin necesidad de repetir el trabajo ya realizado. En cuanto a los hiperparámetros definitivos, la arquitectura del mejor modelo guardado corresponde a una red neuronal con:

- 1 única capa oculta formada por 64 neuronas.
- Función de activación que mejor rendimiento ofreció fue ReLU, ampliamente utilizada por su capacidad de introducir no linealidad y por su eficiencia computacional
- El optimizador Adam, lo cual resulta especialmente adecuado en problemas de regresión con datos normalizados.
- Un total de 10 épocas de entrenamiento. Esto se debe a que en la búsqueda de hiperparámetros se probaron dos valores posibles para el número máximo de iteraciones (10 y 50), y el modelo con 10 iteraciones fue el que tuvo un mejor rendimiento.

Esta configuración de hiperparámetros permitió alcanzar el mejor desempeño en la validación cruzada, asegurando un equilibrio entre la simplicidad de la arquitectura, velocidad de entrenamiento y capacidad predictiva del modelo.

6.3 Conclusiones

El proceso de entrenamiento descrito en este capítulo ha permitido desarrollar un modelo de red neuronal profunda con capacidad predictiva, aplicando una metodología estructurada. A partir de una adecuada selección de variables, una normalización de los datos y una división en conjuntos de entrenamiento y prueba, se sentaron las bases para un aprendizaje robusto. La definición de espacio de hiperparámetros y el uso de la validación cruzada mediante la técnica de búsqueda *Grid Search*, garantizó una exploración de las distintas opciones, ajustadas a las restricciones de memoria del entorno de simulación. Finalmente, guardar el mejor modelo entrenado junto con el escalador empleado, permite su reutilización sin pérdida de fidelidad en entornos futuros.

Este enfoque no solo optimiza el correcto entrenamiento y optimiza el rendimiento del modelo, sino que también asegura la escalabilidad del sistema predictivo basado en una red neuronal desarrollado en este proyecto.

7 Implementación y resultados del modelo de red neuronal en el simulador EPON

7.1 Introducción

En este capítulo de la memoria se aborda la integración del modelo de red neuronal desarrollado en este proyecto dentro del módulo de la OLT del simulador (*OLT_IA.py*), con el propósito de evaluar su desempeño frente al esquema de asignación de recursos clásico implementado en el TFG realizado por Víctor Herrezuelo [1].

El objetivo fundamental es analizar cómo un enfoque basado en inteligencia artificial puede contribuir a mejorar la eficiencia en la distribución del ancho de banda en redes de acceso PON. En este sentido, el modelo desarrollado se ha diseñado para estimar dinámicamente el valor de ancho de banda máximo a asignar a cada ONU en cada ciclo de simulación, permitiendo una gestión más adaptativa de los recursos de red.

La necesidad de esta implementación surge del comportamiento característico de los sistemas PON, en los que la saturación de red conduce a incrementos bruscos en el retardo medio y a un uso ineficiente del canal disponible. Frente a esta problemática, el modelo predictivo se plantea como una herramienta capaz de anticipar condiciones de carga y ajusta la asignación de forma proactiva, evitando que la congestión se manifieste de manera temprana. La estrategia de integración se realiza de manera progresiva adaptando la estructura del fichero original de la OLT (*OLT.py*) en otro fichero denominado *OLT_IA.py*, para incorporar las salidas del modelo sin alterar en exceso la lógica interna del simulador, garantizando así la estabilidad durante las ejecuciones.

La implementación del modelo requiere la incorporación de librerías adicionales, así como adaptar los procedimientos de normalización de datos de entrada y la gestión de las predicciones realizadas por el modelo. Este proceso permitirá que las estimaciones que

genere la red neuronal se integren de manera directa en el cálculo del ancho de banda asignado a cada ONU.

Finalmente, se configurará el simulador para ejecutar comparativas entre ambos enfoques, abarcando diferentes escenarios de carga y considerando tanto entornos 1G-EPON como 10G-EPON. Los resultados que se obtengan permitirán la realización de un análisis de las métricas clave como el ancho de banda medio asignado a cada ONU y el retardo medio, proporcionando una visión completa del impacto del modelo DNN en el rendimiento del sistema.

7.2 Implementación del modelo de red neuronal en el módulo de la OLT

Tal y como ha sido expuesto en secciones anteriores, el modelo desarrollado en este proyecto tiene como finalidad estimar el ancho de banda máximo *B_max* asignado a cada ONU en cada ciclo. Esta predicción se representa mediante la variable *self.B_predicted_B_max*, la cual forma parte del fichero *OLT_IA.py* descrito en apartados anteriores. La integración del modelo se ha realizado directamente sobre dicho fichero *Python*, adaptando su estructura para incorporar las salidas del sistema de predicción sin alterar demasiado su estructura. Esta fase de implementación permite que el modelo de red neuronal intervenga de forma dinámica en el proceso de asignación de recursos, aportando una estimación personalizada del ancho de banda máximo para cada ONT en cada ciclo de la simulación. A continuación, se detallan las modificaciones introducidas para garantizar una integración coherente y funcional del modelo predictivo.

7.2.1 Carga de librerías

A diferencia del fichero *OLT.py* del TFG de partida [1], en el nuevo script *OLT_IA.py* se importan unas librerías extra que se emplean en la predicción y normalización (ver Figura 20) necesarias para poder emplear el modelo. En concreto:

- *Joblib:* Sirve para cargar el mejor modelo entrenado y el escalador que se ha guardado durante el proceso de entrenamiento en un archivo .pkl (*pickle file*) [14].
- *Warnings y DataConversionWarning:* Sirven para realizar la gestión de las advertencias de tipo datos al hacer las predicciones [22].

- *Torch:* Sirve para el desarrollo y entrenamiento de modelos de deep learning, y permite construir redes neuronales mediante una interfaz flexible y dinámica [12].
- *NumPy:* Sirve para poner en el formato esperado los datos de entrada del modelo [7].

```
import torch
import joblib
import numpy as np
import warnings
from sklearn.exceptions import DataConversionWarning
```

Figura 20: Librerías empleadas para implementar el modelo en *OLT_IA.py*

7.2.2 Carga del modelo entrenado

En la variable *model* de la Figura 21, se carga el modelo para realizar la predicción, que en este proyecto corresponde a una red neuronal previamente entrenada y optimizada en el proceso de búsqueda de hiperparámetros. Esta operación se realiza mediante la función *model = joblib.load("mejor_modelo_dnn_v5.pkl").* Posteriormente en la variable *scaler* se carga el escalador, que es el *MinMaxScaler* empleado para normalizar las entradas del modelo, tal y como se explicó en el Apartado 6.2.4. Esta operación se realiza mediante la función *scaler = joblib.load("scaler_v5.pkl").* Ambas funciones vienen proporcionadas por la librería *joblib* de *Python*. La separación entre escalador y modelo garantiza que las condiciones de entrenamiento se respeten al realizar la predicción, evitando así distorsiones por diferencias entre las magnitudes de entrada.

```
# Cargar el modelo entrenado
model = joblib.load("mejor_modelo_dnn_v5.pkl")

# Definir el MinMaxScaler y cargarlo desde el archivo
scaler = joblib.load("scaler_v5.pkl")
```

Figura 21: Parte de *OLT_IA.py* donde se carga el mejor modelo entrenado

7.2.3 Uso de la función de predicción del modelo

Esta función encapsula el procedimiento para realizar la predicción a través de las variables de entrada *B_demand*, *Onu_id* y *Carga.* Dicha función se muestra en el código de la Figura 22. En primer lugar, se incluye la instrucción *warnings.filterwarnings(action='ignore', category=DataConversionWarning)*, que se

encarga de realizar un manejo explícito de advertencias para evitar errores del tipo *DataConversionWarning*. Esta medida contribuye a garantizar la estabilidad del sistema durante simulaciones de larga duración.

A continuación, se normalizan las variables de entrada mediante el uso del escalador previamente cargado. Esto se realiza mediante la instrucción *entrada_normalizada = scaler.transform([[demand_bits, onu_id, carga]])* que transforma las variables originales *demand_bits*, *onu_id* y *carga* (nombres que se les asigna localmente en esta función), a un rango adecuado para el modelo predictivo. Finalmente, la estimación se lleva a cabo a través de la instrucción *estimacion = model.predict(entrada_normalizada)*. En este paso, la red neuronal procesa los datos normalizados y devuelve el valor estimado correspondiente.

```python
def estimar_valor(demand_bits, onu_id, carga):
    import warnings

    from sklearn.exceptions import DataConversionWarning

    # Desactivar todas las advertencias relacionadas con la conversión de datos
    warnings.filterwarnings(action='ignore', category=DataConversionWarning)

    try:
        # Normalizar los datos de entrada
        entrada_normalizada = scaler.transform([[demand_bits, onu_id, carga]])
    except DataConversionWarning:
        pass  # Ignorar las advertencias relacionadas con la conversión de datos

    # Realizar la estimación utilizando el modelo
    estimacion = model.predict(entrada_normalizada)

    # Restaurar el comportamiento normal de las advertencias
    warnings.filterwarnings(action='default', category=DataConversionWarning)

    return estimacion[0]
```

Figura 22: Función *estimar_valor()*

A continuación, se procede a la realización de modificaciones en la función *procesa_report()* que gestiona el procesamiento de los mensajes tipo *Report* enviados por las ONUs en cada ciclo.

Por otro lado, y previo a realizar la estimación, es necesario preparar los datos de entrada para que cumplan el formato esperado por el modelo. Este conjunto de variables constituye el vector de estado, el cual describe las condiciones actuales de la red y de la ONU en cuestión (ver Figura 23). En nuestro modelo, el vector de entrada está compuesto por tres características principales:

- *B_demand:* Demanda de ancho de banda de la ONU en un ciclo, expresada en bits. Corresponde a la suma del tamaño total de las colas.
- *Onu_id:* Identificador de la ONU que ha realizado el envío del *MensajeReport*.
- *Carga:* Valor escalar que representa el nivel de carga de cada ONU durante la simulación. Ha de establecerse manualmente.

Estas tres variables se agrupan en una estructura de tipo *NumPy Array*, utilizando la instrucción *input_data = np.array([[B_demand_bits, Onu_id,Carga]], dtype=np.float32)*, que posteriormente será normalizada mediante un objeto *MinMaxScaler* que previamente ha sido cargado en *scaler*. La normalización es esencial para que las magnitudes de entrada sean compatibles con los valores para los que el modelo fue entrenado. Este bloque de código (ver Figura 23) asegura que los datos se encuentren en el formato adecuado para la etapa de predicción y representa la entrada estándar sobre la que se ejecutará la predicción del modelo.

```
# Extraer el valor de B_demand[ont_id] como un número
B_demand_bits = self.B_demand[ont_id]

# Definir Onu_id como el propio ont_id
Onu_id = ont_id

# Definir Carga como un valor fijo
Carga = 0.8

# Empaquetar las variables en un arreglo de NumPy
input_data = np.array([[B_demand_bits, Onu_id,Carga]], dtype=np.float32)
```

Figura 23: Creación en *OLT_IA.py* del vector de características

7.2.4 Predicción y uso del valor estimado en la asignación dinámica de ancho de banda

Durante este proceso, tal y como se ve en el código de la Figura 24, se realiza una llamada a la función explicada previamente *estimar_valor()* a la que se le pasan como argumentos las variables del vector de entrada. Con ello, la red neuronal realiza la predicción del *B_max* para esa ONU en el ciclo en cuestión. El resultado, almacenado en la variable *self.predicted_B_max*, representa el ancho de banda máximo estimado que se permitirá a la ONU en el siguiente ciclo de transmisión.

```
self.predicted_B_max = estimar_valor(B_demand_bits, Onu_id,Carga)
```

Figura 24: Realización en *OLT_IA.py* de la predicción de B_max

A continuación, el valor *self.predicted_B_max* se utiliza como límite superior en el cálculo del ancho de banda a asignar a la ONU *self.B_alloc*, tal y como se observa en la Figura 25. La asignación se realiza mediante la instrucción *self.B_alloc[ont_id] = min(self.B_demand[ont_id], self.predicted_B_max) + tamano_report*, ya que se aplica un esquema de asignación limitado. De esta manera, nuestro modelo influye directamente en la política de asignación de recursos de la OLT, permitiendo que la estimación de *B_max* se adapte dinámicamente a la demanda y al contexto de la red, en lugar de depender únicamente de reglas estáticas. Este enfoque proporciona al sistema una capacidad adaptativa, permitiendo una asignación de recursos más eficiente frente a condiciones de tráfico dinámico.

```
if self.predicted_B_max> 119488:
    self.predicted_B_max=119488

self.B_alloc[ont_id] = min(self.B_demand[ont_id], self.predicted_B_max) + tamano_report
```

Figura 25: Asignación de Ancho de Banda con la implementación de la predicción

7.3 Configuración del simulador EPON para el uso del modelo de red neuronal

Una vez implementadas todas las modificaciones en el fichero *OLT_IA.py* para integrar el modelo de red neuronal desarrollado en este proyecto, es necesario definir cómo debe configurarse el simulador para ejecutar simulaciones tanto con el uso como sin el uso del modelo. Tal y como se observa en la Figura 26, en la carpeta *classes/packages* del simulador se encuentran, entre otros, los siguientes archivos clave:

- *OLT.py:* Implementación tradicional de la OLT, utilizada por defecto por el simulador. El algoritmo DBA que se emplea en este caso será determinista y se basará en el esquema limitado explicado en apartados anteriores.
- *OLT_IA.py:* Versión alternativa de la OLT que incorpora toda la lógica relacionada con el modelo de inteligencia artificial basado en una red neuronal.

Figura 26: Archivos del simulador EPON

Este cambio es necesario porque el simulador carga y emplea automáticamente el archivo que se denomina *OLT.py*, ignorando el resto de las variables disponibles. Por lo que, para alternar entre los modos de simulación, es necesario renombrar los ficheros del siguiente modo [23]:

- Simulación sin emplear IA: Mantener los nombres originales (*OLT.py*, *OLT_IA.py*). El simulador ejecutará la versión estándar de la OLT, sin intervención del modelo predictivo.
- Simulación empleando IA: Se ha de renombrar *OLT_IA.py* como *OLT.py* y cambiar el nombre original de *OLT.py* a otro (por ejemplo, *OLT_ORIGINAL.py*). El simulador empleará entonces la versión actualizada que incluye el modelo de inteligencia artificial desarrollado en este proyecto.

Además del ajuste en los archivos utilizados, las simulaciones que incorporan el modelo basado en redes neuronales requieren modificar manualmente la variable *Carga* dentro del archivo *OLT.py* (previamente denominado *OLT_IA.py*). Tal como se ilustra en la Figura 27 esta variable debe tomar el mismo valor que *CONFIG_CARGA*, definido en el módulo *configuration.py*, y representa el nivel de carga del sistema durante la simulación, usualmente dentro del rango [0.1, 0.9].

Dado que la generación de archivos .csv se realizan al ejecutar este archivo, cada ejecución con IA conlleva un uso intensivo de memoria por lo que se recomienda realizar las simulaciones de forma secuencial (una a una), evitando su ejecución conjunta. Además,

debido a que la variable *Carga* no está parametrizada ni gestionada dinámicamente en el código, no es posible automatizar los barridos de carga cuando se emplea el modelo predictivo. Por tanto, cada simulación debe ser ejecutada de forma independiente, modificando el valor de *Carga* manualmente para cada escenario.

```
# Extraer el valor de B_demand[ont_id] como un número
B_demand_bits = self.B_demand[ont_id]

# Definir Onu_id como el propio ont_id
Onu_id = ont_id

# Definir Carga como un valor fijo
Carga = 0.8

# Empaquetar las variables en un arreglo de NumPy
input_data = np.array([[B_demand_bits, Onu_id,Carga]], dtype=np.float32)

self.predicted_B_max = estimar_valor(B_demand_bits, Onu_id,Carga)
```

Figura 27: Lugar donde modificar *Carga* manualmente

Como alternativa para evitar la configuración manual de la variable *Carga* en cada ejecución, y a su vez prevenir posibles problemas de saturación de memoria derivados de la generación continua de archivos .csv, se propone una solución más eficiente y escalable. Esta consiste en realizar una llamada directa al módulo *configuration.py* mediante la instrucción *from packages.configuration.configuration import CONFIG_CARGA* (Figura 28), e incorporar la asignación *Carga = CONFIG_CARGA* dentro del archivo *OLT.py* (previamente *OLT_IA.py*) tal y como se ve en la Figura 29. Con esta modificación, el valor de carga utilizado se controla de forma centralizada a través del fichero de configuración, lo cual permite automatizar fácilmente barridos de carga.

```
from packages.configuration.configuration import CONFIG_CARGA
```

Figura 28: Modificación para poder automatizar las simulaciones para diferentes cargas

```
# Extraer el valor de B_demand[ont_id] como un número
B_demand_bits = self.B_demand[ont_id]

# Definir Onu_id como el propio ont_id
Onu_id = ont_id

# Definir Carga como un valor fijo
Carga = CONFIG_CARGA

# Empaquetar las variables en un arreglo de NumPy
input_data = np.array([[B_demand_bits, Onu_id,Carga]], dtype=np.float32)

self.predicted_B_max = estimar_valor(B_demand_bits, Onu_id,Carga)
```

Figura 29: Implementación de la modificación para automatizar los barridos de carga

Además, con el fin de reducir el consumo de memoria durante estas ejecuciones en serie, se recomienda comentar o deshabilitar temporalmente el bloque de código responsable de la exportación de resultados a archivos .csv (Capítulo 5) tal y como se ve en la Figura 30. De este modo, se consigue una ejecución más liviana, evitando la escritura constante en disco y posibilitando una secuencia automatizada de pruebas con diferentes niveles de carga, todo ello manteniendo la coherencia entre los parámetros definidos y utilizados en cada simulación.

```
## Aqui es donde se hace extracción de datos
        # Creamos la carpeta donde se guardarán los archivos CSV si no existe
        #carpeta_resultados = 'RND_resultados_csv_L05_Bmax_V2_0.8'
        #if not os.path.exists(carpeta_resultados):
        #    os.makedirs(carpeta_resultados)
        # Generamos el nombre del archivo CSV con un marcador de tiempo
        #nombre_archivo = f'valores_{time.strftime("%Y%m%d_%H%M%S")}.csv'
        # Combinamos la ruta de la carpeta con el nombre del archivo
        #ruta_completa = os.path.join(carpeta_resultados, nombre_archivo)

        # Guardamos los valores de B_alloc y n_alloc en el archivo CSV
        #with open(ruta_completa, mode='a', newline='') as file:
        #    writer = csv.writer(file)
        #    if file.tell() == 0:
        #        writer.writerow(['Carga','Onu_id', 'B_demand_bits', 'B_demand_MBS', 'B_max_bits', 'B_max_MBS','B_alloc_bits',
        # 'B_alloc_MBS', 'B_guaranteed', 'error_max_aloc', 'error_demand_alloc', 'error_max_demand'])
        #    writer.writerow([CONFIG_CARGA,ont_id, self.B_demand[ont_id], ((self.B_demand[ont_id]*(10**-6))/(T_AVAILABLE))/8,
        # self.B_max[ont_id], ((self.B_max[ont_id]*(10**-6))/(T_AVAILABLE))/8, self.B_alloc[ont_id],
        # ((self.B_alloc[ont_id]*(10**-6))/(T_AVAILABLE))/8,((self.B_alloc[ont_id]*(10**-6))/(T_AVAILABLE))/8,
        # self.B_guaranteed[ont_id], (self.B_max[ont_id]-self.B_alloc[ont_id]),(self.B_max[ont_id]-self.B_demand[ont_id])])
## Aqui es donde termina
```

Figura 30: Desactivación de la recopilación de archivos para hacer barridos de prueba

7.4 Análisis de resultados del modelo de red neuronal para redes 1G-EPON y 10G-EPON

En este apartado se describe la configuración seleccionada para realizar las simulaciones y el análisis de resultados, con el objetivo de establecer una comparativa entre la simulación realizada con el simulador original [1] y la simulación que emplea el modelo de inteligencia artificial implementado en este proyecto.

En dichas simulaciones, se va a barrer la carga de las ONUs de 0.1 a 0.9 en intervalos de 0.1 en 0.1, excepto entre 0.6 y 0.7 que habrá un salto intermedio de 0.05 abordando así también la carga 0.65. El tiempo de simulación escogido ha sido de 1000s para todas las cargas para las que se realizarán las simulaciones. También se considera que todas las ONUs tienen la misma carga en media, es decir son simétricas en la carga. A continuación, se procede a detallar los parámetros específicos para los dos casos que se van a abordar, 1G-EPON y 10G-EPON.

7.4.1 Configuración del entorno de simulación para 1G-EPON

Para realizar la simulación 1G-EPON los parámetros han sido los siguientes que se pueden ver en la Tabla 3 [1].

Parámetros (Archivo parameters.py)	**Valor del Parámetro**
Número de ONUs	16 ONUs
Tasa de Transmisión de la red (*R_tx*)	1 Gbit/s
Tasa de Transmisión máxima de las ONUs	100 Mbit/s
Longitud de la Red (OLT – ONT)	20 km
Período del Ciclo	2 milisegundos
Tiempo de Guarda	5 microsegundos
Tamaño del Buffer	10 MBytes
Tamaño de los Paquetes	Paquetes de 64, 594 y 1500 Bytes
Número de Colas (Servicios)	1 Cola
Algoritmo Implementado	Algoritmo limitado

Tabla 3: Configuración para realizar simulaciones con entornos 1G-EPON

7.4.2 Configuración del entorno de simulación para 10G-EPON

Para realizar la simulación 10G-EPON los parámetros han sido los siguientes que se pueden ver en la Tabla 4 [1]:

Parámetros (Archivo parameters.py)	**Valor del Parámetro**

Número de ONUs	16 ONUs
Tasa de Transmisión de la red (*R_tx*)	10 Gbit/s
Tasa de Transmisión máximo de las ONUs	1 Gbit/s
Longitud de la Red (OLT – ONT)	20 km
Período del Ciclo	2 milisegundos
Tiempo de Guarda	5 microsegundos
Tamaño del Buffer	100 MBytes
Tamaño de los Paquetes	Paquetes de 64, 594 y 1500 Bytes
Número de Colas (Servicios)	1 Cola
Algoritmo Implementado	Algoritmo limitado

Tabla 4: Configuración para realizar simulaciones con entornos 10G-EPON

7.5 Análisis de los resultados

A continuación, se expone el análisis detallado de los resultados obtenidos a partir de las simulaciones ejecutadas para los escenarios correspondientes a las arquitecturas 1G-EPON o EPON y 10G-EPON. Estas simulaciones han sido diseñadas para evaluar el comportamiento del sistema en condiciones variables de carga y configuración, permitiendo así una caracterización más precisa de su rendimiento.

El objetivo principal de este apartado es interpretar el funcionamiento del modelo desarrollado cuando se enfrenta a diferentes escenarios de red, valorando su funcionamiento y eficiencia bajo distintos contextos operativos. Para ello, se analizarán y compararán diversas métricas clave, como el retardo medio o el ancho de banda medio asignado a cada ONU por parte de la OLT, con el fin de extraer conclusiones significativas.

Este análisis comparativo entre entornos 1G-EPON y 10G-EPON proporciona información valiosa sobre las fortalezas y limitaciones del sistema y del modelo, contribuyendo a validar su aplicabilidad en redes ópticas pasivas de próxima generación.

7.5.1 Evaluación de resultados en redes 1G-EPON

En primer lugar, en la Figura 31, se analiza la evolución del retardo medio que sufren los paquetes en el sentido de subida, esto es, desde que se generan en cada ONT hasta que se reciben en el OLT, haciendo un barrido a todas las cargas de la ONTs (0.1-0.9). La Figura 31 muestra la evolución del retardo medio (en segundos) en función de la carga de la ONT, comparando dos aproximaciones distintas para la gestión de los recursos: por un lado, el algoritmo original con esquema limitado e implementado en el TFG de partida [1], representado en color azul, y por otro lado, el algoritmo que implementa un modelo basado en redes neuronales (DNN) desarrollado en este proyecto, representado en color rojo.

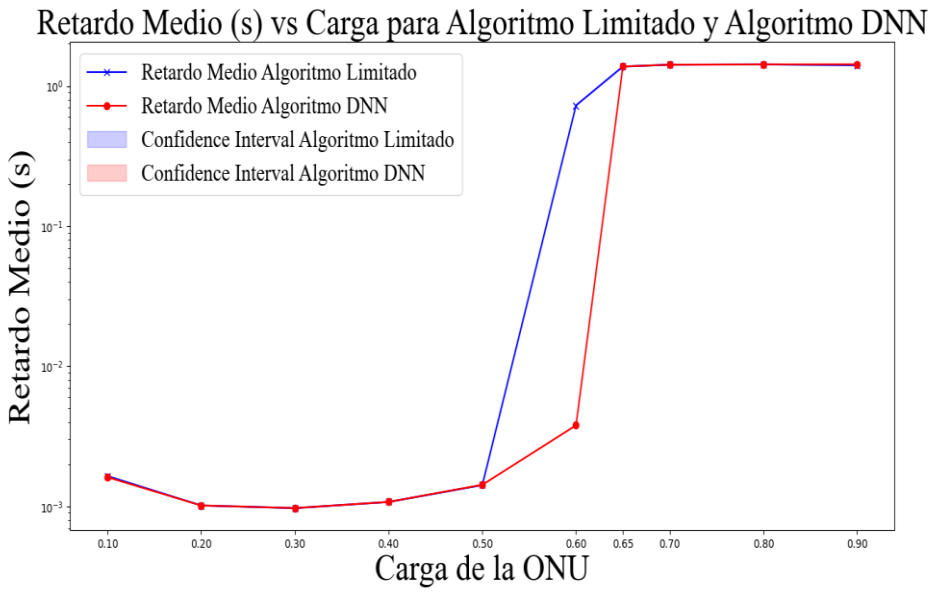


Figura 31: Retardo medio comparando el algoritmo limitado clásico con el algoritmo basado en redes neuronales (NN) para redes 1G-EPON considerando todas las cargas de las ONUs

En primer lugar, se observa que en el intervalo de baja carga (0.1-0.5) ambos algoritmos presentan un comportamiento muy similar. El retardo medio en estas condiciones permanece en el rango de los milisegundos, lo que indica que la red dispone de suficiente capacidad de transmisión y los mecanismos de asignación, independientemente de la estrategia empleada, son capaces de garantizar un servicio eficiente y con un impacto mínimo en la latencia o retardo medio percibido. En este

régimen, tanto el método clásico como el modelo con DNN cumplen el objetivo de mantener un retardo bajo y estable, sin diferencias significativas.

Sin embargo, al incrementar la carga de la ONU a valores superiores, en torno a 0.6, se comienzan a manifestar diferencias notables entre ambos esquemas. En el caso del algoritmo original, el retardo medio experimenta un incremento abrupto, pasando de valores muy bajos a magnitudes del orden de los segundos en un intervalo reducido de carga. Este crecimiento pronunciado refleja la saturación del sistema, donde la capacidad de asignación de recursos del esquema limitado deja de ser suficiente para dar respuesta a la demanda creciente de tráfico.

Por el contrario, el algoritmo DNN logra posponer este punto crítico de crecimiento del retardo, manteniendo un desempeño más estable en la franja de carga de 0.6 a 0.65 y retrasando el inicio de la degradación de la calidad del servicio y el aumento del retardo. Aunque finalmente acaba convergiendo a retardos elevados al aproximarse al régimen de saturación (0.7-0.9), el hecho de extender la zona de funcionamiento eficiente indica una mejor capacidad de gestión de recursos bajo condiciones de alta exigencia. Dicho de otro modo, el modelo predictivo basado en redes neuronales profundas (DNN) consigue aprovechar de forma más eficiente el ancho de banda disponible, reduciendo la probabilidad de congestión temprana y permitiendo un mayor margen operativo antes de alcanzar la saturación. En síntesis, la gráfica pone de relieve dos conclusiones principales:

1. En condiciones de carga baja y alta, ambos enfoques son equivalentes y garantizan retardos reducidos.
2. En condiciones de carga media, el algoritmo clásico sufre una degradación de la calidad del servicio mucho más temprana, mientras que el algoritmo DNN consigue extender la región de operación estable, ofreciendo un mejor control sobre la latencia y, en consecuencia, una mejor calidad de servicio percibida por el usuario de la red.

Por otro lado, la Figura 32 muestra la evolución del ancho de banda medio asignado a cada ONU (en Mbps) en función de la carga de la ONU, comparando el comportamiento del algoritmo original [1], representado mediante la curva de color azul, con el modelo basado en redes neuronales profundas (DNN) desarrollado en este proyecto, representado mediante la curva de color rojo. En primer lugar, se aprecia que en condiciones de carga baja (0.1-0.4) ambos algoritmos siguen trayectorias prácticamente iguales, con un

incremento proporcional de los recursos demandados y disponibles a medida que crece la demanda del tráfico en función de la carga. Esto refleja que, en un escenario de red bajo, tanto el enfoque tradicional como el basado en DNN logran una distribución eficiente del ancho de banda, sin que se evidencie ventaja significativa alguna de uno sobre otro.

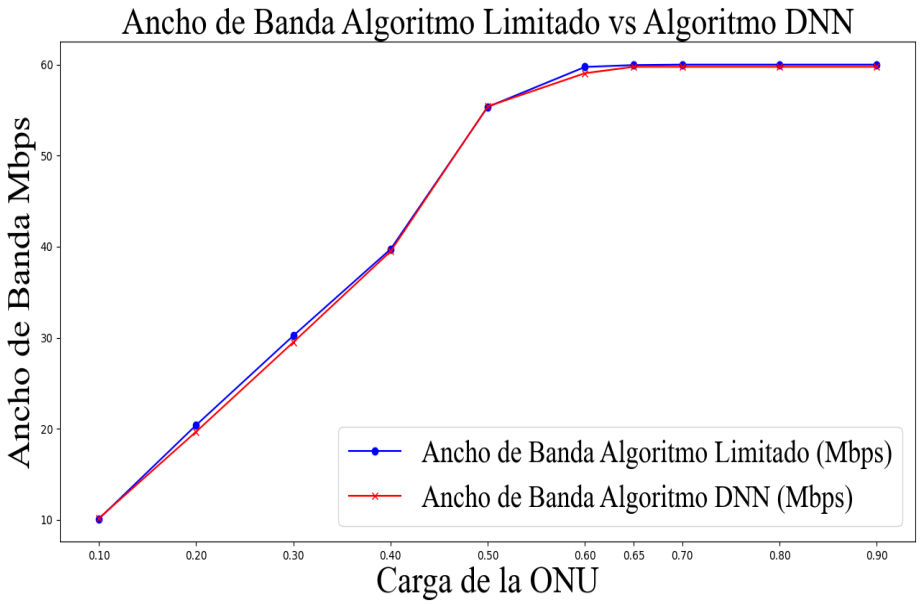


Figura 32: Evolución del ancho de banda comparando el algoritmo clásico con el algoritmo basado en redes neuronales (NN) para redes 1G-EPON considerando todas las cargas de las ONUs

A medida que la carga se incrementa hacia valores intermedios, en torno a 0.5-0.6, los dos algoritmos alcanzan un valor cercano a los 55-60Mbps, que corresponde al límite de asignación debido a la capacidad máxima del canal de subida (1 Gbps) entre las 16 ONUs. Es importante señalar que ambos métodos muestran una rápida convergencia hacia este umbral máximo, lo que prueba que el sistema de control de ancho de banda está correctamente dimensionado y que la asignación de recursos tiende a saturar de forma natural al aproximarse al límite físico disponible.

A partir de la carga 0.7 en adelante, tanto el algoritmo original como el DNN se estabilizan en el valor máximo de 60 Mbps por ONU, sin superar en ningún momento este umbral. Este comportamiento en la zona de alta carga es un resultado esperado, dado que la restricción fundamental está impuesta por la capacidad del medio físico y no por la estrategia de asignación. De este modo, bajo condiciones de saturación, ambos métodos operan al límite superior permitido, garantizando la equidad en la distribución del ancho de banda entre usuarios.

Si bien las diferencias entre los algoritmos en esta métrica son prácticamente inexistentes, el resultado es muy relevante desde el punto de vista de la validación del sistema. Por un lado, confirma que el modelo basado en DNN respeta las restricciones impuestas por la infraestructura, alcanzando el mismo límite de capacidad que el algoritmo clásico. Por otro lado, avala que la inteligencia artificial no introduce sesgos ni desequilibrios en la asignación de ancho de banda, incluso bajo condiciones extremas de operación. En resumen, la gráfica evidencia que la estrategia de asignación de ancho de banda es consistente y robusta en ambos enfoques, y que, aunque el algoritmo DNN aporta mejoras significativas en otros indicadores como el retardo medio (tal y como se pudo observar en la Figura 31), en este caso no altera los valores máximos alcanzables por usuario, garantizando así la equidad y estabilidad del sistema en condiciones de saturación.

7.5.2 Evaluación de resultados en redes 10G-EPON

En este caso, se van a generar las mismas gráficas a partir de los ficheros de resultados que se generan una vez finalizadas las simulaciones llevadas a cabo para 10G-EPON. En primer lugar, en la Figura 33, se analiza la evolución del retardo medio versus la carga de la ONU, comparando ambos algoritmos.

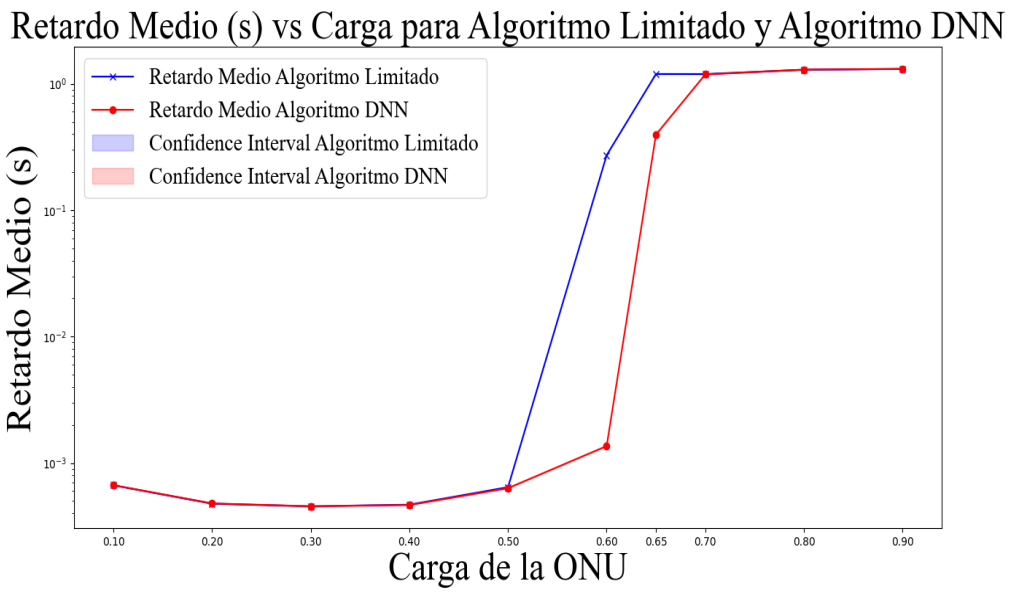


Figura 33: Evolución del retardo medio comparando el algoritmo clásico con el algoritmo basado en redes neuronales (NN) para redes 1G-EPON considerando todas las cargas de las ONUs

La gráfica muestra que en situaciones de cargas bajas (0.1 a 0.4), ambos algoritmos muestran un retardo medio muy bajo y similar. Sin embargo, a partir de una carga de 0.5, comienzan a observarse diferencias significativas, ya que el algoritmo clásico presenta un

aumento abrupto en el retardo medio a partir de 0.6, alcanzando valores superiores a 1 segundo. En contraste, el algoritmo con redes neuronales logra mantener el retardo bajo control hasta cargas más altas (cerca de 0.7), con una pendiente de crecimiento más suave. Esto refleja una mayor robustez y capacidad de adaptación del enfoque con IA en situaciones de mayor exigencia, especialmente en cargas medias y altas. El comportamiento por lo tanto es similar al caso de 1G-EPON.

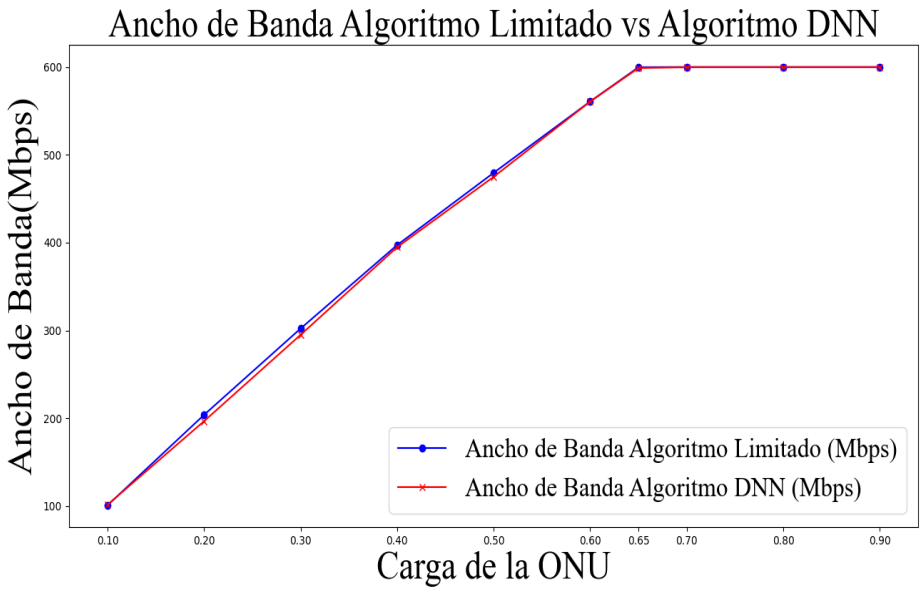


Figura 34: Evolución del ancho de banda comparando el algoritmo clásico con el algoritmo basado en redes neuronales (NN) para redes 10G-EPON considerando todas las cargas de las ONUs

En segundo lugar, se presenta la Figura 34 resultante del análisis del ancho de banda asignado de media a cada una de las ONUs frente a la carga de la ONU. La gráfica representa la evolución del ancho de banda medio asignado por ONU (en Mbps) en función de la carga de la ONU, comparando el comportamiento del algoritmo con redes neuronales, (en rojo) y el algoritmo con esquema limitado clásico (en azul). En general, ambos algoritmos asignan un ancho de banda similar a todas las ONUs a lo largo de todo el rango de cargas, mostrando un comportamiento casi idéntico en condiciones bajas y medias (0.1 a 0.6). A partir de una carga de 0.65, ambos alcanzan el límite máximo de asignación (600 Mbps), sin que se observe ventaja clara de uno sobre otro. Esto sugiere que, en términos de asignación de recursos, el uso de IA no introduce una mejora significativa, aunque sí podría influir indirectamente en métricas como el retardo.

7.6 Conclusiones

Del estudio realizado en este capítulo se extraen varias conclusiones relevantes. En primer lugar, la integración del modelo de red neuronal en el módulo de la OLT se ha llevado a cabo con éxito, permitiendo que el sistema sea capaz de realizar predicciones dinámicas del ancho de banda máximo a asignar a cada ONU en función de las condiciones actuales de la red. Esta incorporación ha sido realizada sin comprometer la estabilidad del simulador, respetando la estructura original y asegurando la coherencia de la lógica de asignación de recursos.

Los resultados obtenidos en las simulaciones para entornos 1G-EPON y 10G-EPON ponen de manifiesto que, bajo cargas bajas y altas, ambos enfoques, tanto el algoritmo clásico como el modelo basado en DNN, presentan un rendimiento prácticamente equivalente, tanto la métrica de retardo medio como en la asignación de ancho de banda. Esto valida la fiabilidad del modelo y garantiza que la introducción de técnicas de inteligencia artificial no afecta de manera negativa al comportamiento esperado del sistema en condiciones habituales de operación.

Sin embargo, al aproximarse a escenarios de cargas medias, el modelo predictivo demuestra ventajas claras frente al algoritmo limitado. En particular, el DNN consigue retrasar el punto crítico en el que el retardo medio se dispara, extendiendo la zona de operación eficiente y proporcionando un mejor control de la latencia. En cuanto al ancho de banda en este rango de cargas medias, ambos algoritmos convergen en el mismo límite máximo impuesto por la capacidad de la red, lo que evidencia que el modelo propuesto mantiene la equidad en la distribución de recursos, incluso bajo cargas extremas.

En conjunto, estos resultados confirman que la aplicación de técnicas de aprendizaje automático en sistemas EPON es viable y también beneficiosa para mejorar la eficiencia y la calidad de servicio en condiciones de alta exigencia. La capacidad de adaptación que aporta el modelo basado en redes neuronales refuerza su aplicabilidad en redes ópticas pasivas de próxima generación, constituyendo un paso hacia arquitecturas más inteligentes y eficientes.

8 Implementación de nuevas funcionalidades en el simulador EPON

8.1 Introducción

La evolución de las redes de acceso ópticas, en particular de las arquitecturas EPON hacia redes de acceso más avanzadas y con mayor tasa de transmisión, como 25G-EPON (25 Gpbs), requiere el desarrollo de herramientas de simulación que permitan reproducir de forma realista las condiciones de operación y evaluar el impacto de distintas configuraciones de tráfico y políticas de asignación de recursos. En este trabajo se han introducido mejoras sustanciales en el simulador EPON, orientadas a incrementar su capacidad de modelado y análisis.

En primer lugar, se implementa un nuevo generador de tráfico basado en distribuciones Pareto, capaz de reproducir comportamientos característicos del tráfico real en redes Ethernet modernas. En segundo lugar, se incorporarán perfiles de abonados con SLAs (*Service Level Agreements*) diferenciados, lo cual dotará al simulador de la capacidad de gestionar de forma precisa la asignación de ancho de banda garantizado a cada ONU, en función de parámetros contractuales definidos. Finalmente, se extenderá la herramienta para soportar escenarios de 25G-EPON, con parámetros adaptados a esta tecnología. Estas mejoras convierten al simulador en un entorno robusto y flexible para el estudio del rendimiento de redes ópticas de nueva generación.

8.2 Implementación de un nuevo generador de tráfico con distribución Pareto

El nuevo generador de tráfico de pareto, archivo *GeneraTráfico.py* en el Código, implementa una fuente *self-similar* basada en el modelo de Kramer, originalmente integrado en C++ en el Proyecto Fin de Grado de Jose María Robledo Sáez [24] para

simular tráfico en redes EPON. Por lo tanto, lo que se ha adaptado es dicha versión de la fuente y tráfico de Pareto a Python.

El modelo de generación de tráfico incorporado en esta versión del simulador se basa en múltiples fuentes que siguen distribuciones de Pareto en sus periodos de actividad (ON) e inactividad (OFF), lo cual permite simular tráfico autosimilar (tipo de tráfico que presenta una estructura similar a diferentes escalas temporales) de forma realista en redes EPON. La autosimilitud es una propiedad clave del tráfico en redes modernas como Ethernet, caracterizada por ráfagas de paquetes con correlación temporal de largo alcance, incluso con distintas escalas de tiempo (autosimilitud). Esta característica se puede cuantificar mediante el parámetro de Hurst (H), que toma valores entre 0.5 y 1. En concreto el tráfico autosimilar presenta H > 0.5, indicando autocorrelación o comportamiento autosimilar a largo plazo.

Existe una relación directa entre el valor de Hurst y el parámetro de forma α de la distribución de Pareto que rige los tiempos de los periodos ON y OFF, expresada mediante la siguiente fórmula: $H = \frac{3-\alpha}{2}$. Según esta relación, para un valor $\alpha = 1.4$, se obtiene H ≈ *0.8,* lo cual confirma un comportamiento claramente autosimilar. Este valor ha sido ampliamente validado en la literatura en *Leland* [25] y en estudios empíricos sobre tráfico de redes de datos [24].

En esta versión del simulador, se han adoptado los valores $\alpha_{ON} = 1.4\ y\ \alpha_{OFF} = 1.2$ para los periodos de actividad e inactividad, respectivamente, con el objetivo de reproducir de la forma más fidedigna posible el tráfico bursty (rafagoso) y autocorrelado que se observa en redes Ethernet reales. La combinación de estos parámetros genera flujos de tráfico que alternan entre ráfagas y periodos de inactividad, reflejando así la naturaleza dinámica del tráfico de red.

El proceso de generación de dicho tráfico con fuentes de Pareto (ver Figura 35) se inicia dentro de la clase *GeneraTrafico*, donde se lanza el proceso de inicio de emisión de paquetes por fuentes independientes a través de la instrucción: self.action = *env.process(self.generador_pareto_paquetes(12000, carga_real)).*

En esta llamada (ver Figura 35), el valor 12000 representa el tamaño del *payload* (carga útil) de cada paquete, expresado en bits. Este valor equivale a 1500 Bytes, que

corresponde al tamaño típico de una trama Ethernet sin cabecera. Junto con los bytes de encabezado definidos por la variable *tamano_cabecera*, se determina el tamaño total de la trama generada. La variable *carga_real*, por su parte, define el nivel de carga deseado en la red, y se utiliza dinámicamente para ajustar los parámetros de la distribución Pareto OFF y así mantener el equilibrio entre el tráfico generado y capacidad del sistema.Cada fuente encapsulada en el sistema genera paquetes con estas características y los introduce en las colas de la ONU correspondiente, en función de la configuración de encolado definida en el simulador.

```
class GeneraTrafico:
    # Simula por una parte la capa de aplicación que va generando daos a una velocidad R_datos,
    # y por otra simula la capa de transporte que va segmentando los datos en datagramas de 1500 Bytes
    def __init__(self, env, id, carga, seed_1=None, seed_2=None):

        self.env = env
        self.id = id
        self.colas = []
        self.colas_longitudes = []
        self.Bytes_generados=0
        self.Bytes_descartados=0
        self.paquetes_generados=0
        self.paquetes_descartados=0
        self.id_paq = 0
        self.carga_onu = 0 # carga de la onu en bps
        self.retardo_estadisticas = []


        #esto solamente se puede cambiar si hay más de una fuente, sino no
        if(multiples_colas):
            carga_real = carga - .0448
        else:
            carga_real = carga

        self.action = env.process(self.generador_pareto_paquetes(12000, carga_real))
```

Figura 35: Inicio del nuevo generador Pareto

A continuación, se define el generador de tráfico Pareto (ver Figura 36) que se usa una agregación de múltiples fuentes que son encapsuladas mediante la clase *SourcePareto*. Siendo *packet_size = 1526*, *streams = 32* (definidos en *parameters.py*) y *carga* el nivel de carga de la ONU definida en *configuration.py*.

```
self.pAG = Generator()

for src in range(streams + 1):
   self.pAG.AddSource(SourcePareto(src, 0, packet_size, 0, carga/streams, 1.4, 1.2))
```

Figura 36: Agregación de fuentes

Cada fuente representa un flujo con una carga parcial, *Generator* se encarga de combinarlas y gestionar la temporización tal y como se ve en la Figura 37.

```
while True:

    AG = self.pAG

    # Obtener el siguiente paquete
    AG.GetNextPacket()

    # Obtener el próximo paquete
    trc = AG.PeekNextPacket()

    # Obtener el tiempo actual
    tiempo = AG.GetTime()

    # Obtener el ByteStamp actual
    byte_stamp = AG.GetByteStamp()

    # Calcular el intervalo
    intervalo = byte_stamp - tiempo

    # Calcular la variable interarrivaltime a partir del intervalo determinado en bits
    interarrivaltime = intervalo * 8 / ONU_bit_rate
```

Figura 37: Cálculo del intervalo de tiempo en el que llega el paquete.

Con estos datos, se calcula el tiempo entre llegadas, necesario para poder mantener la carga deseada. Cada paquete generado, tal y como se observa en el código de la Figura 38, se encapsula como una trama Ethernet y se introduce en las colas de la ONU asociada, en caso de que haya sitio suficiente.

```
#Aquí creamos la trama Ethernet
paquete = TramaEthernet(self.id_paq, 'L', self.id, tam_paq + tamano_cabecera, self.env.now, prioridad)
self.id_paq += 1

 # Contabilizamos paquetes
self.Bytes_generados +=(tam_paq + tamano_cabecera)/8
self.paquetes_generados += 1

# Metemos el paquete en la cola correspondiente, en función de la configuración
if insertionmethod_separatequeue0_priorityqueue1==False:
        # Método de inserción de colas separadas
        self.encolador_colas_separadas(paquete, prioridad)
else:
        # Método de inserción de prioridad de colas
        self.encolador_prioridad_colas(paquete, prioridad)
```

Figura 38: Creación del paquete ethernet y inserción en el sistema de colas de la ONU

El método de inserción de paquetes en las colas de las ONUs depende del esquema de encolado configurado en el simulador. Existen dos opciones implementadas definidas como: *encolador_colas_separadas* y *encolador_prioridad_colas* (Figura 38), definidas dentro de la clase *GeneraTrafico*. La selección entre ambas se controla mediante la variable *insertionmethod_separatequeue0_priorityqueue1* en el archivo *configuration.py*, donde el valor *False* activa el método de colas separadas y *True* el método basado en prioridades.

Ambos métodos evalúan dinámicamente la disponibilidad del buffer, si hay espacio, el paquete se inserta en la cola correspondiente (ver Figura 39), en caso contrario se descarta. Esta lógica fue descrita originalmente y de forma más detallada en el Trabajo de Fin de Grado de Víctor Herrezuelo [1]

En el presente proyecto, se ha utilizado *encolador_prioridad_colas*, por su capacidad adaptativa y eficiente para gestionar múltiples clases de servicio y simular entornos con tráfico diferenciado en casos en los que sea necesario.

```
if lon_cola_total + paquete.len <= L_BUFFER_ONTS:

    self.colas[prioridad].append(paquete)
    self.colas_longitudes[prioridad] += paquete.len
elif(prioridad==N_COLAS-1):
    # Si el paquete es de la menor prioridad de todas y no queda sitio, lo descartamos.
    # Si tenemos cola única y el paquete no cabe, se descarta.
    self.Bytes_descartados += paquete.len/8
    self.paquetes_descartados += 1
```

Figura 39: Código para la inserción de los paquetes en las colas de las ONUs

En cuanto al generador aleatorio, aunque no se emplea explícitamente *MersenneTwister*, como en el Trabajo de José María Robledo, se usa en nuestro caso *numpy.random*. La nueva implementación es modular, reproducible y fiel al comportamiento estadístico del tráfico de las redes reales.

8.3 Implementación de perfiles de abonados con SLA (Service Level Agreements) en el simulador EPON

Con el propósito de emular de manera más precisa el comportamiento de una red PON (*Passive Optical Network*) en entornos reales, se han realizado modificaciones significativas en el archivo *OLT.py* del simulador original [1]. El objetivo principal de estas modificaciones es incorporar soporte para múltiples *Service Level Agreements* (SLAs), lo que permite representar distintos perfiles de usuario, cada uno con requisitos específicos de calidad de servicio. Esta ampliación funcional dota al simulador de la capacidad de asignar a cada ONU un ancho de banda garantizado personalizado, alineado con los SLA acordados con el proveedor de servicio. De esta manera, se reflejan de forma realista las políticas de asignación de recursos típicos en entornos GPON e EPON comerciales. La implementación de esta nueva funcionalidad se articula en torno a la creación e inicialización de varios vectores clave:

- *Self.B_guaranteed:* Define el ancho de banda mínimo garantizado (en bits por segundo) para cada ONU según su SLA. Véase Figura 40.

- *Self.t_inicio_tx, self.t_inicio_tx_ant y self.t_ciclo:* Almacenan y calculan los tiempos de transmisión y ciclos de transmisión en cada ciclo de cada ONU. Véase Figura 41.

Estos elementos permiten aplicar un control de los recursos asignados por ciclo, en función de las políticas de calidad de servicio configuradas en cada simulación. El vector *self.B_guaranteed* (ver Figura 40) se declara en el constructor de la clase OLT y tiene como finalidad almacenar el ancho de banda garantizado asignado a cada ONU, en función de su nivel de SLA. Se trata de un vector de tipo lista, con longitud igual al número total de ONUs (*N_ONTS*), definido en el módulo de configuración *parameters.py*. La inicialización del vector se realiza con valores nulos (*self.B_guaranteed=[0.0]*N_ONTS*). Posteriormente, este vector se rellena con valores personalizados que representan la capacidad garantizada para cada ONU (en bits por segundo).

```
self.B_guaranteed=[0.0]*N_ONTS # Vector que guarda el SLA que tendrá cada ONU en Mbps
```

Figura 40: Creación del vector *self.B_guaranteed*

Esta estructura es esencial para controlar el límite de recursos asignados a cada ONU, en línea con su acuerdo de nivel de servicio (SLA). La OLT consultará este vector en cada ciclo para determinar cuál es el umbral máximo de ancho de banda que puede ofrecer a una ONU específica. Además del vector *self.B_guaranteed*, se definen otros tres vectores fundamentales en la gestión temporal de cada ONU (Ver Figura 41):

- *Self.t_inicio_tx:* Almacena el tiempo de inicio de la transmisión actual para cada ONU (en segundos).
- *Self.t_inicio_tx_ant:* Registra el tiempo de la transmisión anterior, es decir, cuándo se transmitió por última vez (en segundos).
- *Self.t_ciclo:* Contiene el tiempo transcurrido entre dos ciclos, calculado dinámicamente como la diferencia entre los dos tiempos anteriores.

La inicialización se realiza tal y como se ve en la Figura 41. Esta lógica permite modelar un comportamiento dinámico de la red, adaptando las decisiones de asignación a la actividad reciente de cada ONU.

```
self.t_inicio_tx = [0.0]*N_ONTS # vector que representa el tiempo de inicio de transmisión de cada ONT
self.t_inicio_tx_ant = [0.0]*N_ONTS
self.t_ciclo=[0.0]*N_ONTS
```

Figura 41: Creación de los vectores *self.t_inicio_tx*, *self.t_inicio_tx_ant* y *self.t_ciclo*.

A continuación, el vector *self.B_guaranteed* se inicializa con valores diferenciados para grupos específicos de ONUs (ver Figura 42). Esta asignación se realiza mediante estructuras condicionales en el código, que permiten asignar distintos niveles de capacidad garantizada según el servicio deseado. Por ejemplo, en la configuración mostrada en la figura, las ONUs con SLA alto (ONUs 0 y 1) pueden recibir hasta 900 Mbps; las ONUs con SLA medio (ONUs 2 a 5) se configuran con 750 Mbps; y las restantes, con el SLA más bajo, reciben 500 Mbps. Esta asignación se define directamente mediante el índice de cada ONU, lo que facilita una configuración personalizada de los SLAs. Esto nos permite simular entornos con distintos perfiles de servicio, lo cual es representativo de situaciones reales en redes de acceso PON.

```
# Asignar ancho de banda garantizado según el grupo de ONTs
if i<=1:
    self.B_guaranteed[i]=900000000 # 900 Mbps
if 2<=i<=5:
    self.B_guaranteed[i]=750000000 # 750 Mbps
if 6<=i<=15:
    self.B_guaranteed[i]=500000000 # 500 Mbps
```

Figura 42: Implementación de los diferentes SLAs con diferentes anchos de banda garantizados

Cada vez que se recibe un *MensajeReport* de una ONU concreta, la OLT calcula cuánto tiempo (en segundos) ha pasado desde la última transmisión de dicha ONU (Véase Figura 43) de la siguiente manera: $t_{Ciclo} = tInicio_{Tx}\,[ont_{id}] - tInicio_{TxAnt}\,[ont_{id}]$. Este valor permite establecer la ventana temporal sobre la cual se calculará el límite de recursos que se le puede asignar a la ONU en el siguiente ciclo, de acuerdo con su acuerdo de nivel de servicio (SLA).

```
# Calculo de t_ciclo, excepto la primera vez que es 0 y usamos T_AVAILABLE
# Resto usamos el t_ciclo dinámico calculado
t_ciclo=self.t_inicio_tx[ont_id]-self.t_inicio_tx_ant[ont_id]
```

Figura 43: Cálculo dinámico del tiempo de ciclo *t_ciclo*

Una vez calculado el valor de *t_ciclo* para una determinada ONU, el siguiente paso clave es determinar cuántos bits puede transmitir como máximo durante el nuevo ciclo. Este valor se conoce como *B_max*, y representa el umbral superior de asignación de ancho de banda por ciclo, condicionado por el SLA de cada ONU. La lógica para calcula *B_max* se basa en la expresión condicional que se puede observar en la Figura 44:

- Caso 1, *t_ciclo* > 0: Cuando el valor de *t_ciclo* es positivo, significa que la ONUs ha tenido ancho de banda asignado en el ciclo anterior y que el sistema ha podido medir el tiempo transcurrido desde su última transmisión. En este caso el ancho de banda máximo asignable se calcula de la siguiente manera $B_{Max}\,[ont_{id}]\,(bits) = B_{Guaranteed}\,[ont_{id}](bps) * t_{Ciclo}(s)$, donde $B_{Guaranteed}\,[ont_{id}]$ representa la capacidad máxima en bits por segundo que se ha acordado contractualmente con la ONU y t_{Ciclo} el tiempo en segundos entre dos ciclos consecutivos. El resultado es un valor en bits que representa el límite superior que puede recibir la ONU en el ciclo siguiente, teniendo en cuenta su SLA.
- Caso 2, *t_ciclo* ≤ 0: Este caso contempla situaciones excepcionales como El primer ciclo de simulación, donde no se tienen datos previos y por tanto *t_inicio_tx_ant* = 0 o casos en los que no se ha realizado un cálculo válido de *t_ciclo*.

Para evitar comportamientos incorrectos, se emplea el valor *T_AVAILABLE*, que representa el tiempo máximo disponible en un ciclo estándar del sistema (configurado en *parameters.py*). Esto asegura, que incluso en el primer ciclo, la ONU reciba al menos el volumen de datos que le correspondería por SLA durante un ciclo.

```
# Calculamos el B_max dinámicamente según el tiempo de ciclo de cada ONU
if t_ciclo > 0:
    self.B_max[ont_id]=self.B_guaranteed[ont_id]*t_ciclo #bps * s = bits
else:
    self.B_max[ont_id]=self.B_guaranteed[ont_id]*T_AVAILABLE #bps * s = bits
```

Figura 44: Cálculo dinámico de *B_max* en cada ciclo

Una vez calculado, *B_max* actúa como umbral en el proceso de asignación final de recursos (*B_alloc*), tal y como puede observarse en la Figura 45. El simulador compara la

demanda actual de la ONU (*B_demand*) con el valor de *B_max* calculado y aplica la siguiente política:

- Si la demanda (*B_demand*) está por debajo del máximo permitido, se asigna exactamente lo que ha solicitado.
- Si la demanda supera el máximo permitido, se limita la asignación a *B_max*, respetando de manera estricta el SLA.

Este control dinámico de *B_alloc* garantiza que ninguna ONU pueda sobrepasar los recursos a los que tiene derecho por su acuerdo de nivel de servicio (SLA), incluso si existe más capacidad disponible en la red. De este modo, se preserva la fidelidad del modelo respecto al comportamiento de redes PON comerciales.

```
# Se da el ancho de banda en B_alloc, siempre limitado por el máximo establecido por su SLA
if self.B_demand[ont_id] < self.B_max[ont_id]: # Comparación en bits
    self.B_alloc[ont_id]=self.B_demand[ont_id] + tamano_report # En bits ya
else:
    self.B_alloc[ont_id]=self.B_max[ont_id] + tamano_report # En bits ya
```

Figura 45: Establecimiento del ancho de banda asignado, *B_alloc*, ciclo tras ciclo

Con estas modificaciones, el simulador permite modelar escenarios con múltiples acuerdos de nivel de servicio (SLA) asociados a cada usuario u ONU, integrando de forma precisa las restricciones de capacidad y temporización propias de una red PON real.

A través del cálculo dinámico de parámetros como *t_ciclo*, *B_max* y *B_alloc*, se logra una asignación eficiente y ajustada a los valores contractuales de cada abonado. En particular, el sistema garantiza que cada ONU transmite únicamente la cantidad de datos que le corresponde, durante el tiempo estrictamente necesario. Esto introduce un nivel de disciplina temporal y control de acceso que refleja fielmente las exigencias operativas de las redes PON actuales.

Gracias a esta lógica, el simulador permitirá estudiar analizar el impacto de diferentes estrategias de asignación sobre el rendimiento global del sistema. Esta capacidad puede resultar especialmente valiosa en contextos de investigación o simulación de escenarios comerciales, donde el cumplimiento estricto de los acuerdos de nivel de servicio (SLA) y la estabilidad de la red son factores críticos para el éxito de la implementación.

8.4 Actualización del simulador EPON para soportar infraestructuras 25G (25G-EPON)

En este apartado se aborda la actualización del simulador para posibilitar la realización de simulaciones con infraestructuras 25G-EPON, conforme a lo definido en el estándar IEEE 802.3ca 25G/50G-EPON [26]. Dicho estándar establece las especificaciones para redes de acceso óptico de próxima generación, incorporando velocidades de transmisión simétricas de 25 Gb/s y configuraciones escalables hasta 50 Gb/s, lo que permite responder a la creciente demanda de ancho de banda. Entre sus características principales se incluyen una mayor eficiencia espectral, compatibilidad con arquitecturas de división de hasta 1:64, soporte de longitudes de onda múltiples y mecanismos de coexistencia con generaciones previas de EPON. Estas mejoras garantizan un incremento significativo en la capacidad de la red, manteniendo al mismo tiempo la interoperabilidad con equipos heredados y facilitando una migración progresiva hacia servicios de ultra alta velocidad [27].

Para poder realizar dichas simulaciones basta con realizar los siguientes cambios en los parámetros de simulación, recogidos en el archivo *parameters.py* del simulador tal y como se ve en la Tabla 5.

Parámetros (Archivo parameters.py)	**Valor del Parámetro**
Número de ONUs	>= 64 ONUs
Tasa de Transmisión de la red (*R_tx*)	25 Gbit/s simétricos
Tasa de Transmisión máxima de las ONUs	1 Gbit/s
Longitud de la Red (OLT – ONT)	>= 20 km
Período del Ciclo	2 milisegundos

Tiempo de Guarda	1 microsegundos
Tamaño del Buffer	100 MBytes
Tamaño de los Paquetes	Paquetes de 64, 594 y 1500 Bytes
Número de Colas (Servicios)	1 Cola
Algoritmo Implementado	Algoritmo limitado

Tabla 5: Configuración de parámetros para simulaciones de infraestructuras 25G-EPON

8.5 Validación de resultados del generador de tráfico Pareto en el simulador EPON

La primera parte de resultados se centra en analizar una comparativa entre los diferentes generadores de tráfico de Pareto. La Figura 47 muestra el comportamiento del ancho de banda asignado a grupos de ONUs bajo diferentes acuerdos de nivel de servicio (SLAs) en una red 10G-EPON. Se analizaron tres configuraciones: SLA de 900 Mbps para 2 ONUs, SLA de 750 Mbps para 4 ONUs y SLA de 500 Mbps para 10 ONUs, comparando el desempeño de los dos tipos de fuentes de tráfico de Pareto, por un lado, Pareto antiguo (implementado en el simulador original [1]) y Pareto nuevo (implementado en este trabajo, descrito en el Apartado 8.2). En el eje horizontal se representa la carga de las ONUs, mientras que en el eje vertical se observa el ancho de bando asignado a cada SLA. En la Figura 46 se muestra la evolución del ancho de banda medio demandado por las ONUs (en Mbps) por cada uno de los tres grupos de ONUs en función de la carga de las ONUs, diferenciando entre las configuraciones de SLA para Pareto antiguo y Pareto nuevo.

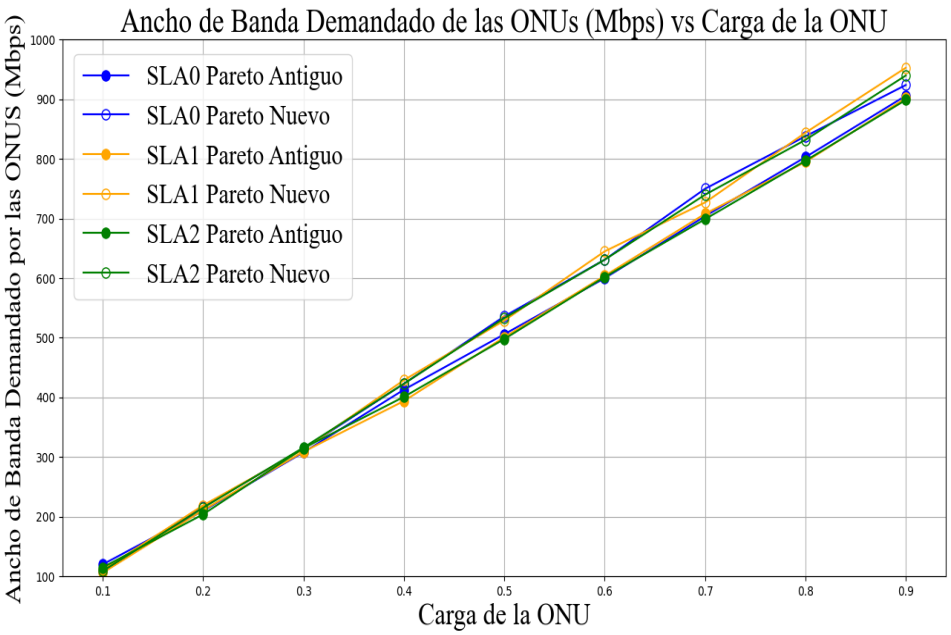


Figura 46: Carga Media solicitada por Grupo de ONUs vs Carga de Simulación para Pareto Antiguo y Pareto Nuevo

Como se puede apreciar, ambas configuraciones mantienen una tendencia lineal respecto a la carga de simulación, lo que confirma la coherencia del modelo. Sin embargo, se observa una diferencia significativa: en el caso del Pareto nuevo, la demanda de ancho de banda tiende a situarse de manera sistemática por encima de los valores teóricos esperados para la carga configurada. Este comportamiento se acentúa a partir de una carga de simulación de 0.4, donde en lugar de ajustarse a los 400 Mbps previstos, el grupo solicita algunos Mbps adicionales. La misma tendencia se mantiene en cargas superiores, de modo que, siempre que no existan limitaciones externas, el Pareto nuevo conduce a una ligera sobreasignación de ancho de banda en comparación con lo que correspondería estrictamente a la carga de simulación.

La importancia de este hecho radica en que condiciona las fases posteriores de asignación de recursos. En particular, al solicitar más carga (o ancho de banda) de la prevista, los grupos de ONUs bajo el Pareto nuevo pueden llegar a recibir un mayor ancho de banda efectivo en los escenarios que no existan limitaciones de los SLAs, generando una diferencia respecto al comportamiento del Pareto antiguo. Este efecto será analizado con más detalle en la Figura 47 donde se discute directamente el impacto sobre la asignación final del ancho de banda. En escenarios de baja carga (valores 0.1 a 0.3), ambos

modelos presentan un comportamiento casi idéntico. El ancho de banda crece linealmente con la carga, sin que se perciban diferencias significativas entre políticas, lo que indica que, en situaciones con bajo nivel de congestión, la asignación de recursos es prácticamente transparente al modelo de Pareto utilizado.

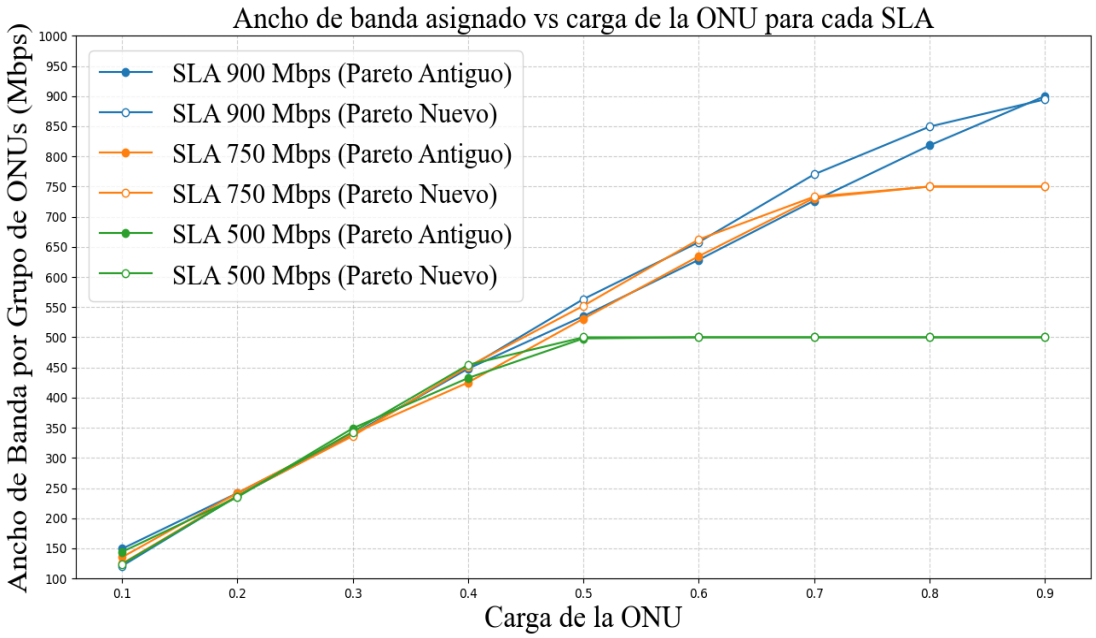


Figura 47: Comparación del ancho de Banda por grupos de ONUs entre ambos generadores de tráfico Pareto para diferentes acuerdos de nivel de servicio.

A partir de una carga de 0.5, comienzan a evidenciarse ciertas diferencias. Para el SLA de 900 Mbps, el Pareto nuevo asigna un ancho de banda ligeramente superior al Pareto antiguo, alcanzando hasta los 900 Mbps cuando es la carga máxima. El Pareto antiguo, en cambio, tiende a limitar la asignación y se mantiene más cercano al SLA, reduciendo la posibilidad de sobreasignación. En el caso del SLA de 750 Mbps, el comportamiento es más equilibrado. Ambos modelos presentan una evolución similar hasta cargas medias, pero hacia cargas más altas, el Pareto nuevo permite un leve incremento por encima del umbral de 700 Mbps en la carga 0.7, mientras que el Pareto antiguo tiende a estabilizarse justo en el límite de 700 Mbps, una vez superados los 750 Mbps ambas fuentes de Pareto se limitan a dicho valor de 750 Mbps tal y como dice su SLA. Por su parte, el SLA de 500 Mbps, asociado al grupo de mayor cantidad de ONUs (10) muestra un patrón completamente estable a partir de una carga de 0.5. Tanto en el Pareto antiguo como en el Pareto nuevo, la asignación queda limitada de manera estricta a los 500 Mbps, sin presentar incremento adicional. Este comportamiento evidencia que, en condiciones donde la densidad de usuarios es alta y los recursos son más sensibles, ambos

modelos priorizan la contención estricta de ancho de banda, garantizando que el SLA se cumpla de manera estricta. En futuros trabajos se abordará un análisis más exhaustivo de este comportamiento, orientado a explicar las razones por las que la nueva fuente de Pareto produce una carga ligeramente mayor. Este efecto resulta consistente con su propia definición, dado que integra 32 fuentes de Pareto multiplexadas, frente a la configuración inicial que consideraba una única fuente.

Por otro lado, la Figura 48 muestra la comparación del retardo medio diferentes acuerdos de nivel de servicio SLAs en una red 10G-EPON, considerando tanto el modelo Pareto antiguo como el Pareto nuevo. Los SLAs evaluados son: 900 Mbps (2 ONUs), 750 Mbps (4 ONUs) y 500 Mbps (10 ONUs). El eje horizontal representa la carga de las ONUs, mientras que el eje vertical, en escala logarítmica, refleja el retardo medio experimentado por los grupos de ONUs. En condiciones de baja carga (0.1-0.3), ambos modelos presentan retardos muy reducidos, cercanos al orden de milisegundos o incluso inferiores. Esto es consistente con la baja congestión del canal en estas condiciones, donde la asignación de recursos resulta suficiente para atender la demanda sin generar colas significativas en los buffers. Cabe destacar que en este rango los retardos del Pareto antiguo son particularmente bajos para los SLAs de 750 y 500 Mbps respectivamente, indicando una gestión más eficiente de la latencia en escenarios de poca saturación.

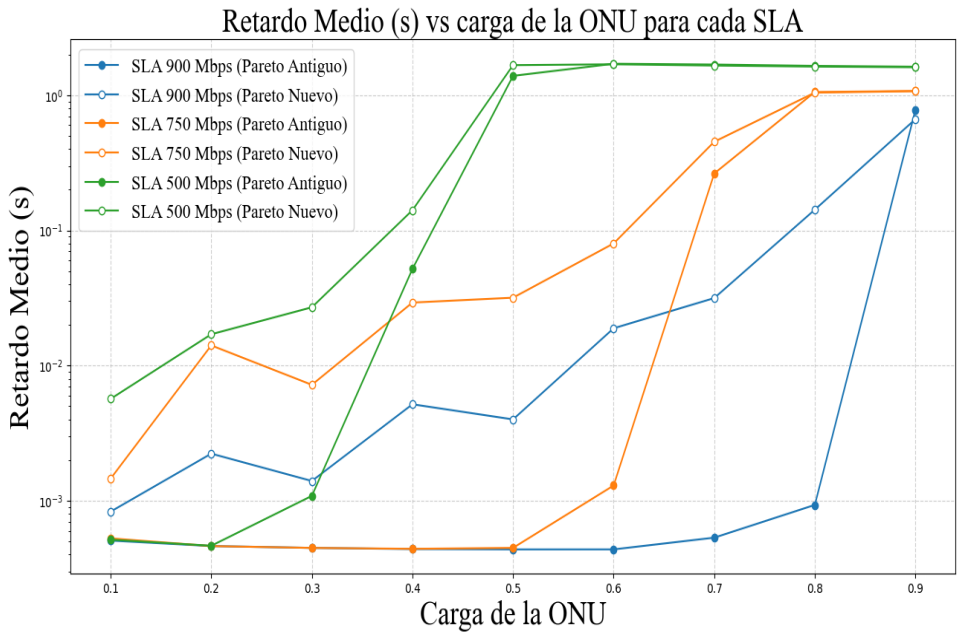


Figura 48: Comparación de Retardos para cada grupo de ONUs entre los diferentes generadores de tráfico empleados para diferentes SLAs.

A partir de cargas medias (0.4-0.6), se observa el incremento diferenciado en los retardos según el SLA y el modelo de Pareto. En el SLA de 900 Mbps, los retardos se mantienen relativamente bajos en ambos modelos, aunque el Pareto antiguo muestra un aumento más gradual y controlado. En cambio, para el SLA de 750 Mbps, el Pareto antiguo mantiene valores inferiores a 10^{-2} segundos hasta una carga de 0.6, mientras que el Pareto nuevo presenta un incremento más temprano, aunque logra estabilizarse entorno a valores constantes.

El caso más crítico corresponde al SLA de 500 Mbps. En este escenario, el Pareto antiguo consigue mantener retardos bajos hasta cargas moderadas, pero hacia cargas superiores a 0.5 se observa una gran elevación, alcanzando valores superiores al segundo. El Pareto nuevo, en contraste, empieza a crecer de manera más anticipada, pero se estabiliza rápidamente en el orden de 1 segundo, mostrando un límite más rígido en la degradación del servicio. Esto sugiere que la política nueva introduce más latencia en escenarios intermedios y evita el crecimiento abrupto en situaciones de congestión extrema.

En términos generales, los resultados permiten identificar que el modelo Pareto antiguo ofrece un mejor desempeño en retardos para cargas bajas y medias, siendo más eficiente en el uso de los recursos disponibles. No obstante, en condiciones de alta carga, su comportamiento puede deteriorarse rápidamente, generando picos elevados de latencia, lo cual es problemático en aplicaciones sensibles al retardo. El modelo Pareto nuevo, por su parte, se caracteriza por mantener una tendencia más estable y predecible en condiciones de congestión, aunque sacrifica eficiencia en escenarios de cargas bajas donde tiene mayor retardo que en el caso de Pareto antiguo. Este comportamiento puede ser debido una vez más a que emplea 32 fuentes mientras que el Pareto antiguo emplea 1 única fuente, lo que hace que puedan existir retardos mayores en algunos SLAs y bajo ciertas cargas.

8.6 Validación del simulador con múltiples configuraciones de SLA en redes 25G-EPON

En este apartado se muestra el comportamiento del simulador para la configuración descrita anteriormente de 25G-EPON, para ella se empleará tanto el nuevo generador de tráfico Pareto introducido en este proyecto, como 3 diferentes acuerdos de nivel de servicio (SLA) para cada grupo de ONUs que seleccionemos. En concreto, la Figura 49 representa

la evolución del ancho de banda promedio asignado por grupo de ONUs en una infraestructura 25G-EPON, bajo tres configuraciones de SLA diferentes: Un grupo de 4 ONUs con un SLA de 600 Mbps, un grupo de 20 ONUs con un SLA de 500 Mbps y un grupo de 40 ONUs con un SLA de 300 Mbps, haciendo un total de 64 ONUs. El eje horizontal muestra la carga de las ONUs, mientras que el eje vertical muestra el ancho de banda promedio efectivo alcanzado por cada SLA en cada carga. En condiciones de baja carga (0.1-0.3), los tres SLAs presentan un crecimiento prácticamente lineal del ancho de banda. Esto refleja que, en escenarios de poca congestión, los recursos del canal son asignados de manera proporcional a la demanda, sin que existan restricciones impuestas por los límites del SLA, esto es, el ancho de banda asignado es igual al demandado porque hay suficiente ancho de banda en el canal. En este rango, las curvas de los tres grupos se superponen parcialmente, lo que indica un reparto equilibrado de la capacidad disponible.

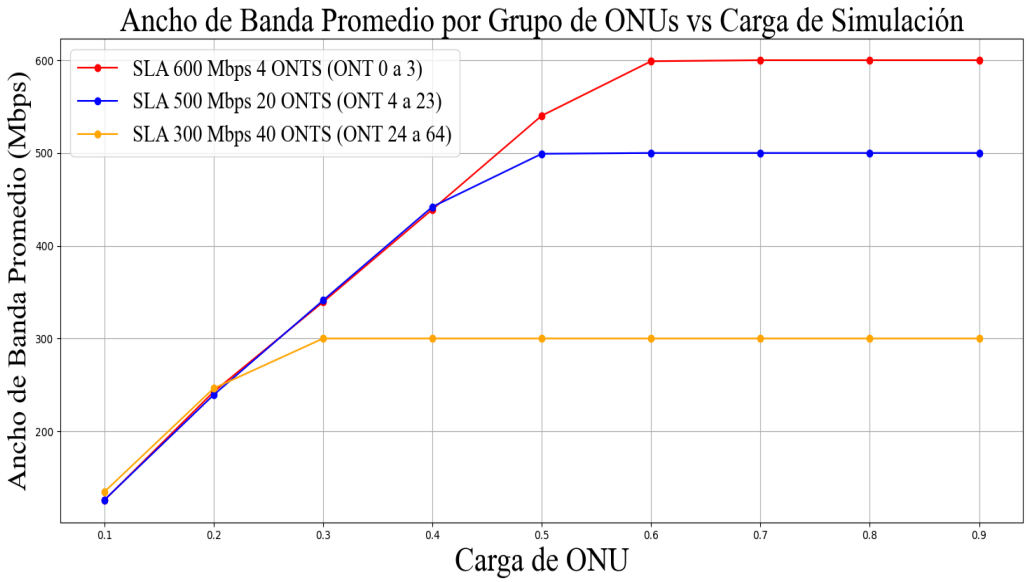


Figura 49: Evolución del ancho de banda promedio asignado a cada SLA en función de la carga de la ONU en una infraestructura 25G-EPON

A partir de cargas medias (0.4-0.6), las diferencias entre los SLAs comienzan a hacerse evidentes. El grupo de 40 ONUs y SLA de 300 Mbps alcanza rápidamente su límite, estabilizándose alrededor de dicho valor a partir de carga de 0.3. Esto refleja que, en escenarios con un número elevado de ONUs y un SLA relativamente bajo, el sistema prioriza garantizar el cumplimiento del límite contractual sin otorgar ancho de banda adicional, incluso si aún existen recursos disponibles. En concreto, a partir de cargas de 0.4 (400 Mbps), la demanda total de las 64 ONUs es mayor o igual a 25 Gbps, en concreto

para 0.4 dicha demanda total es de 25.6 Gpbs, lo que hace imposible al algoritmo de dar a todas las ONUs su demanda. Esto no impide que el algoritmo garantice el ancho de banda acordado en cada SLA, que será igual a 12 Gbps para el SLA de 300 Mbps (40x300 Mbps), 10 Gbps para el SLA de 500Mbps (20x500 Mbps) y 2.4 Gbps para el SLA de 600 Mbps (4x600 Mbps), lo que hace un total de 24.4 Gbps (inferior a los 25 Gbps de la red).

Esto hace que el grupo de 20 ONUs y SLA de 500 Mbps mantiene un crecimiento progresivo hasta alcanzar su límite nominal en torno a una carga de 0.5, momento en el que la curva se estabiliza de manera estricta en 500 Mbps, ya que no puede darle más de su ancho de banda garantizado. Por su parte, el grupo con 4 ONUs y SLA de 600 Mbps presenta la mayor flexibilidad y aprovechamiento de recursos. La curva crece de manera sostenida hasta alcanzar el máximo permitido por su SLA alrededor de carga de 0.6, estabilizándose a partir de ahí. Este comportamiento es consistente, lo cual facilita la asignación de ancho de banda adicional en condiciones de carga creciente, antes de llegar al nivel máximo de su ancho de banda garantizado, dado por su SLA.

Por otro lado, la Figura 50 muestra el retardo medio experimentado por los grupos de ONUs en la infraestructura 25G-EPON bajo las mismas tres configuraciones de SLAs: 600 Mbps para 4 ONUs, 500 Mbps para 20 ONUs y 300 Mbps para 40 ONUs. El eje horizontal muestra la carga de simulación, mientras que el eje vertical, en escala logarítmica, representa el retardo medio en segundos.

En condiciones de baja carga (0.1-0.2), los retardos se mantienen en valores muy reducidos, del orden de los milisegundos o incluso inferiores, particularmente para los SLAs con menor número de ONUs como son los de 600 y 500 Mbps. Esto es coherente con el hecho de que, bajo baja utilización, los recursos son suficientes para atender a la demanda sin formación de colas significativas. El SLA de 40 ONUs, sin embargo, ya muestra un retardo mayor en este rango, lo que refleja la presión que ejerce el elevado número de usuarios sobre el acceso al medio, incluso en escenarios de poca carga. A medida que la carga aumenta (0.3-0.5), se evidencian las diferencias entre los SLAs. El grupo de 40 ONUs con 300 Mbps alcanza rápidamente retardos elevados, llegando a valores cercanos a 10 segundos a partir de una carga de 0.3 y estabilizándose en dicho nivel. Esto sugiere que el SLA más bajo combinado con el mayor número de ONUs genera un escenario de congestión temprana, con tiempos de espera prolongados en los buffers. Es decir, se le da su ancho de banda garantizado, 300 Mbps, pero su demanda es mayor a

medida que aumenta la carga de las ONUs, lo que hacen que aumente el retardo a partir de cargas de 0.3 (300 Mbps).

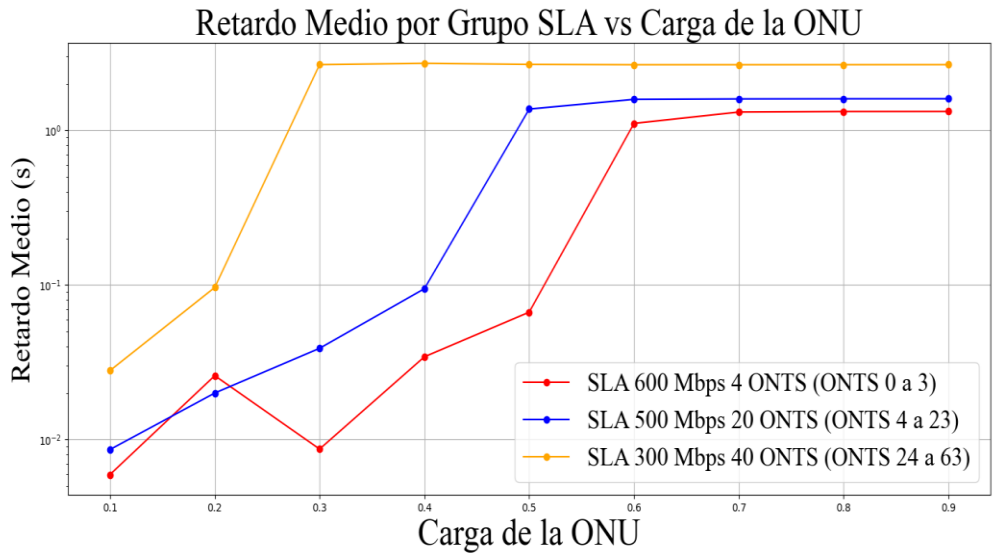


Figura 50: Evolución del retardo medio versus la carga de las ONUs en una infraestructura 25G-EPON con diferentes acuerdos de nivel de servicio (SLAs)

El grupo de 20 ONUs con SLA de 500 Mbps presenta un crecimiento más gradual, manteniéndose por debajo de 0.1 segundos hasta una carga de 0.4, pero experimentando un salto notable en el rango 0.5-0.6, donde se estabiliza en valores próximos a los 2 segundos. Esto indica un retardo aceptable en condiciones moderadas, pero con un deterioro importante en altas cargas. Este comportamiento es coherente con la asignación de su ancho de banda máximo garantizado a partir de cargas de 0.5 (500 Mbps).

Por su parte, el grupo de 4 ONUs con SLA de 600 Mbps ofrece el mejor desempeño en términos de retardo, este se mantiene muy bajo hasta cargas medias y aunque experimenta un aumento abrupto a partir de la carga 0.6, se estabiliza entorno a 1 segundo, lo que sigue siendo mejor que los otros grupos en escenarios de saturación. Esto evidencia la ventaja de tener un grupo reducido de ONUs, ya que se minimiza la competencia por el acceso al canal.

En términos generales, los resultados muestran que el tamaño del grupo de ONUs y el SLA acordado tienen un impacto directo sobre el retardo medio que experimenta cada grupo de ONUs. A mayor número de ONUs y menor SLA, los retardos crecerán más rápidamente y alcanzarán niveles críticos incluso en cargas bajas. En conclusión, mientras los grupos pequeños con SLAs altos presentan retardos contenidos incluso bajo carga

elevada, los grupos grandes con SLAs bajos muestran una rápida degradación del retardo medio. Esto confirma que, en entornos de 25G-EPON, la planificación de SLAs no solo debe considerar el ancho de banda, sino también el impacto del tamaño del grupo sobre los retardos medios que se producen.

8.7 Conclusiones

Las ampliaciones introducidas en el simulador EPON han demostrado mejorar de forma significativa su capacidad para modelar entornos de red más cercanos a la realidad.

La inclusión de un nuevo generador de tráfico Pareto autosimilar ha permitido reproducir patrones de tráfico altamente realistas, ajustados a las propiedades estadísticas observados en redes de datos modernas. Asimismo, la incorporación de perfiles de SLA por ONU ha facilitado la representación precisa de políticas de calidad de servicio, garantizando un control más estricto sobre la asignación de recursos.

En el caso de la actualización hacia infraestructuras 25G-EPON, los resultados han evidenciado la eficacia del simulador para analizar escenarios de mayor capacidad, manteniendo la equidad en la asignación de ancho de banda y evaluando con detalle los retardos medios bajo distintos niveles de carga.

En definitiva, el simulador actualizado se presenta como una plataforma válida para estudiar estrategias de gestión de tráfico, validar nuevas propuestas de asignación de recursos y anticipar el comportamiento de redes PON de próxima generación.

9

Conclusiones y líneas futuras

9.1 Conclusiones

La construcción del modelo de inteligencia artificial basado en una red neuronal profunda ha demostrado ser un proceso viable y técnicamente sólido para su integración en entornos EPON y 10G-EPON. La metodología aplicada, desde la generación y estructuración de un dataset robusto hasta el entrenamiento y validación mediante técnicas de búsqueda de hiperparámetros, ha permitido obtener un modelo predictivo estable y reutilizable. Su implementación en el módulo de la OLT mostró que es capaz de realizar predicciones dinámicas del ancho de banda máximo a asignar a cada ONU sin comprometer la estabilidad del simulador. Aunque en cargas bajas y altas el rendimiento es similar al del algoritmo clásico, en escenarios de cargas medias el modelo con IA mostró una mayor capacidad de adaptación, retrasando el punto crítico de congestión y mejorando la gestión del retardo. Estos resultados confirman que la aplicación de aprendizaje automático en redes ópticas pasivas constituye una herramienta válida y con proyección hacia arquitecturas inteligentes de próxima generación.

La ampliación de funcionalidades del simulador EPON ha permitido dotarlo de mayor flexibilidad y realismo en la evaluación de estrategias de asignación de recursos. La inclusión del nuevo generador de tráfico basado en distribución Pareto supuso una mejora respecto al modelo previo, al reproducir con mayor fidelidad las dinámicas de tráfico típicas de escenarios reales. La incorporación de perfiles de SLA por ONU añadió un nivel extra de control y personalización, esencial para representar diferentes políticas de calidad de servicio en el lado del usuario, realistas y acordes con las ofrecidas por los operadores y proveedores de servicio. Finalmente, la adaptación del simulador a arquitecturas 25G-EPON validó su utilidad como plataforma para estudiar redes de nueva generación,

manteniendo la equidad en la asignación del ancho de banda y evaluando retardos en condiciones de alta carga. En conjunto, estas actualizaciones posicionan al simulador como una herramienta versátil y preparada para el análisis de soluciones futuras en la gestión de recursos en redes PON.

9.2 Líneas futuras

De cara a trabajos futuros, uno de los principales retos consiste en extender la implementación del modelo de inteligencia artificial a entornos de 25G-EPON. La mayor capacidad de estas redes plantea un escenario ideal para validar la escalabilidad del modelo y su capacidad de adaptación a demandas más exigentes. En este contexto, resultará de especial interés analizar su comportamiento bajo condiciones de carga extremas y evaluar métricas adicionales relacionadas con la eficiencia o la robustez frente a fallos.

Además, la introducción de múltiples perfiles de SLA con diferentes niveles de prioridad y garantías de ancho de banda abre una línea para explorar algoritmos de asignación inteligente en este aspecto. La integración de la IA en este marco permitiría gestionar dinámicamente los recursos no solo en función de la carga de la red, sino también considerando acuerdos contractuales entre operador y cliente, aproximándose a escenarios más realistas.

Por otro lado, resultaría de gran relevancia avanzar hacia arquitecturas de redes neuronales capaces de autoaprender en entornos no supervisados. Este enfoque permitiría que el sistema evolucione de forma autónoma ante variaciones en las condiciones de tráfico, sin depender de manera exclusiva de datasets previamente generados y entrenados. De esta manera, el simulador se convertiría en una plataforma adaptable y con mayor capacidad de generalización, acercándose a la visión de redes de acceso autónomas e inteligentes.

Por último, resulta de gran relevancia realizar un análisis en profundidad del simulador con el nuevo generador Pareto implementado, con el fin de determinar las causas por las cuales, en determinadas cargas, asigna una carga superior a la configurada inicialmente de manera teórica.

10

Bibliografía

[1] Víctor Herrezuelo Paredes. “Implementación de un simulador de redes de acceso ópticas pasivas en Python”. Trabajo de Fin de Grado en Ingeniería Tecnoogías de Telecomunicación, E.T.S.I de Telecomunicación. Universidad de Valladolid [En Línea]. Disponible en: https://uvadoc.uva.es/handle/10324/71253?show=full [Último acceso: julio 2025]

[2] Python Tutorial, [En línea]. Available: https://docs.python.org/3/tutorial/ [Último acceso: septiembre 2025]

[3] Python Software Foundation, “About Python”. [En Línea]. Disponible en : https://www.python.org/about/ [Último acceso: septiembre 2025]

[4] Microsoft, “Documentation for Visual Studio Code”. [En Línea]. Disponible en: https://code.visualstudio.com/docs [Último acceso: julio 2025]

[5] Pandas Development Team, “pandas”. [En Línea]. Disponible en: https://pandas.pydata.org/ [Último acceso: julio 2025]

[6] W. McKinney, “pandas: Powerful Python Data Analysis Toolkit”. Zenodo, 2017. [En Línea]. Disponible en: https://pandas.pydata.org/pandas-docs/version/1.4/pandas.pdf [Último acceso: julio 2025]

[7] NumPy Developers, “NumPy”. [En Línea]. Disponible en: https://numpy.org/ [Último acceso: julio 2025]

[8] Scikit-Learn Developers, “Scikit-learn”. [En Línea]. Disponible en: https://scikit-learn.org/stable/

[9] Scikit-Learn Developers, “Sklearn.model_selection”, scikit-learn API Reference. [En Línea]. Disponible en: https://scikit-learn.org/stable/api/sklearn.model_selection.html [Último acceso: julio 2025]

[10] Scikit-Learn Developers, “sklearn.preprocessing”, scikit-learn API Reference. [En Línea]. Disponible en: https://scikit-learn.org/stable/api/sklearn.preprocessing.html [Último acceso: julio 2025]

[11] Scikit-Learn Developers, “sklearn.neural_network.MLPRegressor”, scikit-learn API Reference. [En Línea]. Disponible en: https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPRegressor.html [Último acceso: julio 2025]

[12] PyTorch Developers, “PyTorch”. [En Línea]. Disponible en: https://pytorch.org/ [Último Acceso: Julio 2025]

[13] Tqdm Developers, “tqdm”. [En Línea]. Disponible en: https://tqdm.github.io/ [Último Acceso: Julio 2025]

[14] Joblib Development Team, “joblib: Running Python functions as pipeline jobs.” [En Línea]. Disponible en: https://joblib.readthedocs.io/ [Último Acceso: julio 2025]

[15] F.Chollet, Deep Learning with Python. Shelter Island, NY: Manning Publications, 2018

[16] C. F. Lam, Ed., Passive Optical Networks: Principles and Practice. Burlington, MA: Elsevier Science & Technology, 2007. [En Línea]. Disponible en: https://www.sciencedirect.com/book/9780123738530/passive-optical-networks [Último Acceso: septiembre 2025]

[17] J. Schmidhuber, “Deep learning in neural networks: An overview”, Neural Networks, vol. 61, pp. 85-117, 2015. [En Línea]. Disponible en:

https://doi.org/10.1016/j.neunet.2014.09.003 [Último Acceso: septiembre 2025]

[18] L. Alzubaidi et al., "Review of deep learning: concepts, CNN architectures, challenges, applications, future directions," Journal of Big Data, vol.8, no. 1, p. 53, 2021. [En Línea]. Disponible en: https://journalofbigdata.springeropen.com/articles/10.1186/s40537-021-00444-8 [Último Acceso: septiembre 2025]

[19] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning", Nature, vol. 521, no. 7553, pp. 436-444, 2015.

[20] I. Goodfellow, Y. Bengio, and A. Courville, Deep Learning. Cambridge, MA: MIT Press, 2016.

[21] Python Software Foundation, "warnings-Warning control", Python 3.9 Documentation. [En Línea]. Disponible en: https://docs.python.org/3/library/warnings.html [Último Acceso: septiembre 2025]

[22] Scikit-learn Developers, "sklearn.exceptions-DataConversionWarning", scikit-learn API Reference. [En Línea]. Disponible en: https://scikit-learn.org/stable/modules/generated/sklearn.exceptions.DataConversionWarning.html [Último Acceso: Septiembre 2025]

[23] David Rodríguez Aragón, "ConfiguracionSimulador.docx". [En Línea]. Disponible en: https://docs.google.com/document/d/1OY7yr6zcMaRfxgPYziKYfNZK4lvoGDA3/edit?usp=drive_link&ouid=111326273054394620832&rtpof=true&sd=true [Último Acceso: septiembre 2025]

[24] José María Robledo Sáez (2012), "Implementación de un simulador de redes de acceso pasivas en OMNeT++", Proyecto Fin de Carrera en Ingeniería

Técnica de Telecomunicación en Sistemas de Telecomunicación, E.T.S.I de Telecomunicación, Universidad de Valladolid

[25] W. Leland, M. Taqqu, W. Willinger, and D. Wilson, “On the self-similar nature of Ethernet traffic (extended version)”, IEEE/ACM Transactions on Networking, vol. 2, no. 1, pp. 1-15, Feb. 1994

[26] IEEE 802.3 Working Group, “50G-EPON Task Force, Baseline Proposals & Technical Motions”. [En Línea]. Disponible en: https://www.ieee802.org/3/ca/public/living_documents/motions.shtml [Último Acceso: septiembre 2025]

[27] IEEE 1904.4 Task Force, “Standard for Service Interoperability in 25 Gb/s and 50 Gb/s Ethernet Passive Optical Networks”. [En Línea]. Disponible en: https://grouper.ieee.org/groups/1904/4/tf4_home.shtml [Último Acceso: septiembre 2025]