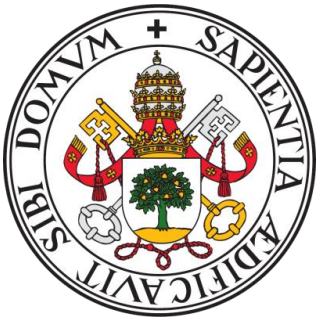
UNIVERSIDAD DE VALLADOLID



E.T.S.I. TELECOMUNICACIÓN

TRABAJO FIN DE GRADO

GRADO EN INGENIERÍA DE TECNOLOGÍAS DE TELECOMUNICACIÓN

Técnicas de aprendizaje por refuerzo profundo (DRL, Deep Reinforcement Learning) para la gestión de recursos en redes de acceso ópticas

Autora:
Clara Ruiz de las Heras
Tutores:
Noemí Merayo Álvarez
Rubén Ruiz González

TÍTULO: **Técnicas de aprendizaje por refuerzo profundo (DRL, Deep Reinforcement Learning) para la gestión de recursos en redes de acceso ópticas**
AUTOR: **Clara Ruiz de las Heras, Rubén Ruiz González**
TUTORES: **Dña. Noemí Merayo Álvarez, D. Rubén Ruiz González**
DEPARTAMENTO: **Teoría de la Señal y Comunicaciones e Ingeniería Telemática**

TRIBUNAL
PRESIDENTE: Ignacio de Miguel Jiménez
SECRETARIO: Noemí Merayo Álvarez
VOCAL: Juan C. Aguado Manzano
SUPLENTE: Ramón J. Durán Barroso Suplente
SUPLENTE: Patricia Fdez. del Reguero

FECHA:
CALIFICACIÓN:

Resumen

En este Trabajo Fin de Grado (TFG), se ha llevado a cabo un estudio de investigación centrado en la optimización de un modelo de *Deep Reinfocement Learning* destinado a la asignación de ancho de banda en redes 10G-EPON (*Ethernet Passive Optical Networks*).

Para ello, se ha llevado a cabo un estudio teórico del funcionamiento de los modelos DRL (*Deep Reinforcement Learning*) y de las redes 10G-EPON. Una vez comprendido el comportamiento de estas tecnologías se procedieron a realizar las modificaciones necesarias al simulador previamente desarrollado por un Trabajo Fin de Grado anterior.

El primer paso fue la integración en el entorno de DRL de una fuente de Pareto realista, en sustitución a la previamente desarrollada, que era demasiado simple, este proceso permitió obtener resultados más cercanos a la realidad en las simulaciones.

Posteriormente se procedió a modificar los distintos escenarios que componían este simulador para que se adaptasen al nuevo modelo de entorno. Estos modelos representan situaciones de tráfico simétrico, asimétrico y dinámico. Además, se diseñó un escenario más realista que combina usuarios en la red PON (ONTs, *Optical Network Units*) de operadores tradicionales, con configuraciones garantizadas, y ONTs de operadores virtuales, con opciones más flexibles.

Finalmente, el aprendizaje del agente DRL se optimizó en todos los escenarios planteados utilizando la herramienta *Optuna*, la cual permite identificar la combinación de hiperparámetros más adecuada para maximizar su rendimiento.

Palabras clave

PON (Red Óptica Pasiva), 10G-EPON, DRL (*Deep Reinforcement Learning*), *Python*, *Optuna*, hiperparámetros.

Abstract

In this Final Degree Project, a research study was done focusing on the optimization of a Deep Reinforcement Learning (DRL) model aimed at bandwidth allocation in 10G-EPON (Ethernet Passive Optical Networks).

To achieve this, a theoretical study was conducted on the operation of Deep Reinforcement Learning models and 10G-EPON networks. Once the behavior of these technologies was understood, the necessary modifications were made to the simulator previously developed in a prior Final Degree Project.

The first step was the integration, into the DRL environment, of a realistic Pareto source to replace the previously developed one, which was too simple. This process allowed for obtaining simulation results closer to real-world conditions.

Later, the different scenarios comprising this simulator were modified so that they would adapt to the new environment model. These models represent symmetric, asymmetric, and dynamic traffic. In addition, a more realistic scenario was designed, combining users (ONTs, Optical Network Units) from traditional operators, with guaranteed configurations, and ONTs from virtual operators, offering more flexible options.

Finally, the learning of the DRL agent was optimized across all proposed scenarios using the Optuna tool, which makes it possible to identify the most suitable combination of hyperparameters to maximize its performance.

Keywords

PON (*Passive Optical Network*), 10G-EPON, DRL (Deep Reinforcement Learning), Python, Optuna, hyperparameters.

Agradecimientos

A mi tutora Noemí Merayo por su gran ayuda a lo largo del trabajo, tanto en lo que respecta al desarrollo de este como a nivel personal, a mi tutor Rubén Ruiz, porque a pesar de estar a 120 km me ha ayudado como ayuda a cualquiera de sus alumnos, a mis padres Araceli y Pablo, por todo lo que me han enseñado y apoyado a lo largo de estos años y a mi pareja Luis por estar ahí siempre.

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 porMICIU/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 byMICIU/AEI/10.13039/501100011033 and by ERFD/EU).

.

Índice

Índice de figuras

Índice de tablas

1

Introducción

1.1 Motivación

Desde los años noventa, como consecuencia del gran desarrollo tecnológico que ha tenido lugar, la demanda de recursos de red ha crecido a una velocidad sorprendente. En este contexto, la fibra óptica y por lo tanto las redes ópticas han demostrado ser una solución eficaz a la hora de satisfacer esta demanda y su uso tiene cada vez más alcance.

Por otro lado, el reciente desarrollo a gran escala de herramientas basadas en el Aprendizaje Automático (*Machine Learning* en inglés) en los últimos años, ha impulsado de manera significativa la capacidad de automatizar procesos y generar soluciones innovadoras en una gran cantidad de campos.

Dentro de este contexto parece lógico intentar combinar las redes ópticas con técnicas de *Machine Learning* de forma que se pueda optimizar la gestión de redes y asignación de recursos.

En este Trabajo Fin de Grado se explorará el uso de técnicas de aprendizaje por refuerzo profundo (DRL, *Deep Reinforcement Learning*) en redes de acceso 10G-EPON (*10G Ethernet Passive Optical Networks*) para optimizar la distribución de ancho de banda entre los usuarios finales, en función de diferentes patrones de Calidad de Servicio (QoS, *Quality of Service*).

Las redes 10G-EPON [1] son un tipo de redes PON, es decir, redes de acceso basadas en fibra óptica que emplean elementos pasivos en toda la planta estándar. Estas redes emplean los principios del estándar *Ethernet* y pueden llegar a grandes velocidades de transmisión de datos, en concreto 10 Gbps, aunque los nuevos estándares podrían llegar a 25/50 Gpbs. Un análisis más profundo de estas infraestructuras de red se integrará en capítulos posteriores.

Por otro lado, el Aprendizaje por Refuerzo Profundo es un subcampo dentro del Aprendizaje Automático que combina los principios del *Deep Learning* y Aprendizaje por Refuerzo [2]. Este tipo de modelos de inteligencia artificial se basan en un agente que, mediante la experiencia que va adquiriendo, interaccionando con su entorno, aprende a tomar decisiones óptimas al recibir una recompensa. El agente no requiere de ningún tipo de supervisión, sino que aprende mediante prueba y error, esto hace que el aprendizaje pueda ser más adaptativo y general, lo que es de especial utilidad en entornos dinámicos y complejos [3].

1.2 Objetivos

1.2.1 Objetivos Generales

Este trabajo tiene tres objetivos principales:

- En primer lugar, plantear, diseñar y desarrollar una serie de escenarios de red que nos permitan observar el comportamiento del agente de DRL bajos diferentes condiciones. En estos escenarios, se simulará una red 10G-EPON con distintas configuraciones en sus patrones de tráfico y de calidad de servicio, desde algunas más simples a otras más complejas y realistas en contextos de red.
- Integrar un agente DRL que funcione interactuando con los distintos escenarios.
- En tercer lugar, optimizar el comportamiento del agente integrado mediante la herramienta *Optuna* [4] de forma que el aprendizaje sea más eficiente y estable.

1.2.2 Objetivos específicos

En cuanto a los objetivos específicos para la consecución de los objetivos generales del punto anterior, pasamos a enumerarlos a continuación:

1. Estudiar el uso de *Python* y sus librerías como herramienta en el desarrollo de simuladores de redes 10G-EPON y de modelos de DRL.

2. Estudiar en profundidad y desarrollar los distintos escenarios planteados de redes 10G-EPON, así como los que estos pueden enseñarnos de cara a la elaboración de una red de acceso con condiciones más realistas.

3. Estudiar la integración de modelos DRL en el contexto de las redes EPON.

4. Observar el efecto de los distintos hiperparámetros y como su variación repercute en el aprendizaje del modelo, todo ello integrando la herramienta *Optuna*.

5. Optimizar el aprendizaje de la herramienta de DRL, ajustando sus hiperparámetros, según el comportamiento de los distintos escenarios.

1.3 Fases y metodología

A continuación, se procede a detallar las fases seguidas y la metodología llevada a cabo en cada una de ellas durante la realización de este trabajo.

1.3.1 Fase de Análisis

El propósito de esta fase inicial es adquirir los suficientes conceptos teóricos para el posterior desarrollo práctico de este Trabajo Fin de Grado. Durante esta fase, se distinguen distintas etapas:

1. El estudio de las características y topología de redes PON, EPON y 10G-EPON.
2. Estudio de los principios del DRL, así como de sus aplicaciones.
3. Familiarización con el lenguaje de programación *Python* [5] y con *Microsoft Visual Studio* [6], que será el editor de código empleado.
4. Documentación y estudio de la biblioteca *Gymnasium* [7] empleada para el desarrollo de entornos DRL, así como de la herramienta *Optuna* que será usada para la optimización de hiperparámetros.

1.3.2 Fase de Implementación

La fase de implementación también comprendió varias etapas que se describen a continuación:

1. Modificación de un simulador inicial DRL en arquitecturas 10G-EPON desarrollado por David Pérez Moreno en un TFG previo, con la integración en el entorno de una fuente de Pareto más realista que la implementada

anteriormente [8] y con modificaciones profundas en su diseño e implementación.

2. Desarrollo e integración de los distintos escenarios con diferentes configuraciones de red 10G-EPON.

3. Modificación del entorno con el que interaccionará el agente DRL, así como del propio agente para satisfacer directrices más complejas.

1.3.3 Fase de Pruebas

Esta última fase y la anterior se realizaron de forma simultánea y con retroalimentación. Esta engloba tanto las simulaciones realizadas para comprobar el funcionamiento del modelo de DRL como los estudios empleando la herramienta *Optuna*, que permitieron ajustar los hiperparámetros que gobernaban el comportamiento del agente para los diferentes escenarios y configuraciones de la red PON. Para evaluar cómo estaba aprendiendo el modelo de DRL, se empleó la curva del *reward*, esta herramienta permite observar como la recompensa que obtiene el agente evoluciona durante el aprendizaje del modelo y es comúnmente usada para analizar el comportamiento de este tipo de agentes ya que ofrece información directa sobre como el agente está aprendiendo en el entorno de análisis.

1.4 Estructura de la Memoria del TFG

El Capítulo 2 describe las herramientas de trabajo que serán empleadas en el TFIG.

En el Capítulo 3 se comienza describiendo las principales características de las redes PON y algoritmos DRL, para proseguir con una explicación del simulador de redes 10G-EPON desarrollado en *Python*.

En el Capítulo 4 se analizarán diferentes escenarios de red y algoritmos simulados en el simulador de *Python* desarrollado en relación con la asignación de ancho de banda en redes 10G-EPON.

El Capítulo 5 recoge las conclusiones generales de este Trabajo Fin de Grado y las líneas futuras que podrían suceder a este estudio.

Por último, en el Capítulo 6 se encuentran las referencias bibliográficas que han servido de apoyo para la realización de este trabajo

2

Entorno de Trabajo

2.1 Introducción

Este capítulo recoge información sobre las herramientas y plataformas *software* empleadas a lo largo de la realización de este trabajo. En este se incluyen tanto las aplicaciones que han facilitado el desarrollo del trabajo, como el lenguaje de programación y las bibliotecas empleadas.

2.2 Microsoft Visual Studio Code

Microsoft Visual Studio Code [6] es el editor de código desarrollado por Microsoft. Esta herramienta presenta varias ventajas, como, por ejemplo: ser de código abierto, la posibilidad de desarrollar *scripts* en múltiples lenguajes de programación, sus herramientas de depuración y las múltiples extensiones con las que cuenta.

En este trabajo, *Visual Studio Code* se emplea como entorno para la edición, organización y ejecución del código, debido a su flexibilidad, facilidad de uso y librerías y herramientas asociadas al *Machine Learning*.

2.3 Python

Python [5] es un lenguaje de programación de alto nivel utilizado en gran variedad de ámbitos. Su sintaxis es sencilla y clara, lo convierte en una herramienta accesible. Además, cuenta con una amplia lista de guías de libre acceso para todos los niveles de programación, lo que facilita su aprendizaje. Su uso es el ámbito de la Inteligencia Artificial es bastante amplio, debido a la gran cantidad de librerías relacionadas con este campo basadas en este lenguaje.

Otra de sus ventajas es que es una herramienta de código abierto con una licencia OSI (*Open-source license*). Esto permite su distribución y uso libre, sin necesidad de pagar una

suscripción. Este lenguaje ya ha sido usado con anterioridad en la elaboración del simulador inicial en el que se basa este proyecto, lo que lo hace la opción más sencilla y segura para trabajar en este caso.

2.4 Optuna

Optuna [4] es una librería destinada a la optimización de hiperparámetros (*hyperparameter optimization framework*) en modelos de IA. En nuestro caso guían el comportamiento del agente DRL desarrollado para agilizar el proceso de búsqueda de configuraciones óptimas en modelos de *Machine Learning*. Además, cuenta con herramientas como *optuna-dashboard*, desde la que se puede controlar el estudio en tiempo real. Se profundizará más en su uso en el Apartado 4.3 de este trabajo.

2.5 Gymnasium

Gymnasium [7] es una librería de código abierto que proporciona una API (*Application Programming Interface*) orientada a la modelización de algoritmos de *Reinforcement Learning* que permite la elaboración de entornos de simulación en los que los agentes pueden entrenar, probar y validar sus comportamientos. Su uso es amplio y diverso, e incluye áreas como las finanzas (estrategias comerciales), la robótica o los juegos, uso bastante conocido de los modelos de DRL.

Mediante su clase principal *Env* se pueden crear entornos con los que interaccionarán los agentes DRL. Esta clase cuenta con una serie de métodos principales, tales como [7]:

- *step():* Nos permite actualizar el estado del agente, determinando las acciones que toma y devolviendo las observaciones de este, así como el que recibe.
- *reset():* Devuelve el entorno a un estado inicial y es necesario que se ejecute antes de llamar al método *step*.
- *render():* Nos permite visualizar lo que ve el agente para así comprenderlo mejor.
- *close():* Cierra el entorno.

2.6 Microsoft Teams

Microsoft Teams [9] es otra aplicación del paquete *Microsoft Office*, en este caso, permite realizar reuniones a distancia entre varios usuarios. La aplicación ofrece una serie de

herramientas útiles como por ejemplo la posibilidad de compartir pantalla con los usuarios que se encuentren en la reunión, lo que en mi caso ha permitido mostrar mis avances de forma más detallada. En general, ha permitido una comunicación más eficaz y cercana.

2.7 Conclusiones

En este primer capítulo de la memoria se ha establecido el marco de trabajo sobre el que se va a desarrollar el resto del estudio y se han explicado las herramientas *software* que se emplearán en el TFG.

3

Asignación de Ancho de Banda en Redes 10G-EPON usando Aprendizaje por Refuerzo Profundo

3.1 Introducción

En este capítulo, se va a describir brevemente las principales características de las redes PON, así como los principios básicos de los modelos de DRL. Sentando así las bases teóricas en las que se fundamente este trabajo.

Por otra parte, se detallarán los aspectos básicos del entorno de DRL desarrollado en *Python* para este entorno de trabajo de investigación, esto es, la aplicación de técnicas de DRL para la gestión de recursos en redes PON, en concreto en redes 10G-EPON.

3.2 Introducción a las Redes de Acceso Ópticas Pasivas (PON, *Passive Optical Networks*) y 10G-EPON

Una Red Óptica Pasiva o PON (*Passive Optical Network)* es un tipo de red telecomunicaciones en la que la conexión entre el proveedor de servicios y el cliente se constituye exclusivamente mediante enlaces de fibra y elementos ópticos pasivos, es decir, elementos que no requieren suministro eléctrico (alimentación) para funcionar, como pueden ser los divisores ópticos o *splitters*.

Debido a la creciente demanda de ancho de banda por parte de los usuarios estas redes se han convertido en una de las tecnologías clave en el sector de las telecomunicaciones, especialmente en el segmento de acceso [10].

3.2.1 Arquitectura de las redes PON

La topología de las redes PON se basa en una estructura de árbol, es decir, punto a multipunto (P2MP) donde una única OLT (*Optical Line Terminal*), que se encuentra ubicada en la central del proveedor de servicios, distribuye señales ópticas a través de uno o varios *splitters* encadenados hacia múltiples dispositivos de usuario. Estas unidades reciben distinto nombre dependiendo de la institución. El IEEE (*Institute of Electrical and Electronics Engineers*) les da el nombre de unidad de la red de fibra óptica (ONU, *Optical Network Unit*), por otro lado, la ITU-T (Unión Internacional de Telecomunicaciones) generalmente se refiere a estas como *Optical Network Terminals* u ONT. A lo largo de este TFG nos referiremos a ellas como ONTs. El esquema general de una red PON es el que se muestra a continuación en la Figura 1 [11].

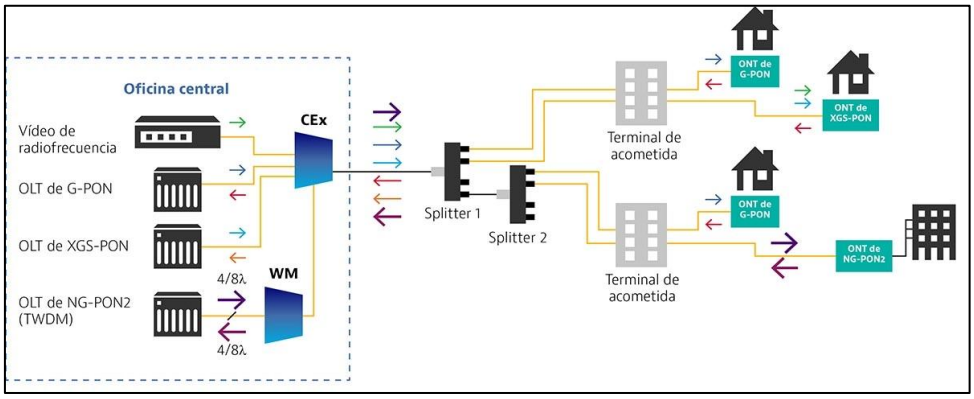


Figura 1. Esquema general de una red PON (Extraído de [11]).

En el canal descendente (desde la OLT hacia las ONTs), la transmisión de datos es de tipo difusión, también llamado *broadcast,* recayendo en la propia ONT la responsabilidad de filtrar el tráfico que le llega, aceptando solo aquel destinado a ella misma.

Por el contrario, el canal ascendente presenta una arquitectura punto a punto (P2P) por lo que es imprescindible evitar colisiones, ya que múltiples ONTs utilizan el mismo canal. En este caso se emplea Acceso Múltiple por División en Tiempo (TDMA, *Time Division Multiple Access*).

3.2.2 Asignación dinámica de ancho de banda

En este tipo de redes se utilizan algoritmos de asignación dinámica de ancho de banda o DBA (*Dynamic Bandwidth Allocation*), basados en TDMA, pero con una asignación dinámica en cada ciclo. Dichos algoritmos son gestionados por la OLT que es la encargada

de proporcionar el ancho de banda a las ONTs tras cada ciclo de tiempo. En el caso del estándar EPON que será explicado más en profundidad en el siguiente apartado, este tiempo de ciclo es adaptativo, pero será como máximo de 2 ms, lo que indica que al menos todas las ONTs deberán transmitir datos en ciclos máximos de 2 ms.

Estos algoritmos ajustan de forma dinámica la cantidad de ancho de banda asignada a cada ONT según diversos factores, estos pueden ser: la demanda en cada momento, el tráfico que se encuentra circulando en la red o los requisitos de calidad de servicio (QoS, *Quality of Service*) que hayan sido contratados por cada abonado con el proveedor de servicios. Este punto, en concreto su aplicación a este trabajo será explicado en profundidad en el Apartado 3.3. Para realizar la asignación de ancho de banda ciclo tras ciclo, la ONT envía mensajes de control, denominados *Report* con el propósito de informar a la OLT de la cantidad de datos (en bits/bytes) que quedan en sus colas tras la transmisión en dicho ciclo. Por otro lado, la OLT, una vez que hace la asignación de ancho de banda para todas las ONTs para el ciclo siguiente, envía un mensaje de control a cada una de ellas, denominado *Gate*, donde se les informa del ancho de banda asignado y del momento en el que debe comenzar a transmitir, ya que se sigue un esquema TDMA dinámico [10]. Dentro de los algoritmos de asignación dinámica de ancho de banda, podemos distinguir dos tipos dos tipos: los algoritmos centralizados y los algoritmos de *polling* cuya descripción se muestra a continuación.

3.2.2.1 Algoritmos centralizados

Los algoritmos centralizados de DBA esperan a recibir el mensaje de tipo *Report* de todas las ONTs antes de decidir la asignación de ancho de banda, lo que puede introducir retardos adicionales y desaprovechamiento de cierto ancho de banda entre ciclos consecutivos. Se encuentran en este grupo aquellos algoritmos donde la OLT asignan el ancho de banda a cada ONT al final de cada ciclo (cuando llega el mensaje *Report* de la última ONT conectada a la red EPON) tras conocer las demandas de ancho de banda de cada una de ellas, por lo que también tiene un conocimiento profundo del estado global de la red antes de aplicar la asignación para el ciclo siguiente [12].

3.2.2.2 Algoritmos de polling

Por otro lado, los algoritmos de *polling* asignan el ancho de banda a cada ONT de forma independiente, tomando decisiones de asignación de forma individual, cuando le llega el mensaje *Report* de cada ONT. De forma concreta, la ONT envía su reporte (*Report*) y la

OLT hace la asignación de ancho de banda y le envía un mensaje *Gate* en el cual se le asigna un ancho de banda, sin esperar el cierre completo del ciclo. Esto permite que no exista ancho de banda desaprovechado entre ciclos consecutivos, aunque no se tiene en cuenta el estado global de todas las ONTs de la red PON. En este tipo de algoritmos, utilizan esquemas de asignación limitados, para evitar que una ONT no monopolice todo el canal o ancho de banda contenido en un ciclo [12].

3.2.3 Descripción de las redes EPON y 10G-EPON

Dentro de las redes PON encontramos las redes EPON, o *Ethernet Passive Optical* Network (estándar IEEE 802.3ah de 2004) [13], que combina los principios de las redes PON con aquellos del estándar Ethernet. De esta forma, todos los datos transmitidos en las redes de fibra óptica se encapsulan en tramas Ethernet.

Su desarrollo se explica por el enorme crecimiento de tráfico de datos y por tanto demanda de ancho de banda en los años 90, que se debe, principalmente, al fuerte desarrollo tecnológico. Debido a esto, parece imprescindible desarrollar una tecnología de telecomunicaciones destinada principalmente a la transmisión de datos dentro de la red [14].Inicialmente se planteó una alternativa basada en ATM (APON) [15]. Desde el principio ATM mostró ciertas limitaciones entre las que se encuentras su alto coste y las dificultades que presentaba para gestionar el tráfico IP. Hoy en día, se considera una tecnología obsoleta. En los años 2000 surgen las redes GPON (Gigabit PON), con velocidades de subida y bajada de hasta 2,4Gbps [16] y que introducen el concepto de GEM (*GPON Encapsulation Method*) que permite la diferenciación de distintos servicios.

Finalmente, tanto GPON como EPON terminan imponiéndose como los estándares más desplegados. En este TFG nos centramos en la tecnología EPON por su facilidad de gestión, escalabilidad y su bajo coste.

Las redes EPON alcanzan velocidades de subida y bajada de 1,25Gbps. Surgen del concepto de *Ethernet in the First Mile* (EFM) o Ethernet de primera milla, descrito en el estándar 802.3ah. En las Redes EPON reestablece una topología punto a multipunto (P2MP). Además, incorpora extensiones en la subcapa de Control MAC (*Media Access Control*). Por otro lado, su mecanismo de Operaciones, Administración y Mantenimiento (OAM) nos permite monitorear y resolver problemas de forma sencilla. En este tipo de

redes la longitud de onda estándar utilizada para ese canal de bajada es de 1490 nm y para el canal ascendente la longitud de onda empleada es de 1310 nm.

Posteriormente las rede EPON fueron mejoradas con el estándar 10G-EPON (estándar IEEE 802.3av de 2009) [1], que permite velocidades de hasta 10 Gb/s de forma simétrica o asimétrica (10 Gb/s bajada y 1 Gb/s subida). Este tipo de redes funcionan a distintas longitudes de que las redes EPON, por lo que se asegura compatibilidad y coexistencia entre ambos estándares en un futuro, y la migración será progresiva. En este caso en el canal descendente emplean una longitud de onda de 1577 nm, mientras que en el canal ascendente esta longitud es de 1270 nm [17].

3.3 Algoritmo de asignación de ancho de banda en nuestro simulador 10G-EPON

Para comprender este apartado es necesario dar una descripción previa de dos conceptos fundamentales en la asignación de ancho de banda aplicada a nuestro caso concreto: el algoritmo DBA empelado en nuestro simulador y los acuerdos de nivel de servicio (SLA, *Service Level Agreement*).

El algoritmo implementado en nuestro simulador EPON es un algoritmo de *polling* con un esquema limitado, basado en el algoritmo *Interleaved Polling with Adaptive Cycle Time* o IPACT [18], que es un algoritmo de asignación dinámica de ancho de banda (DBA) empleado en redes EPON muy extendido por su simplicidad y gran eficiencia. Su objetivo es mejorar la eficiencia del canal ascendente empleando algoritmos de *polling* (definidos en el Apartado 3.2).

Por otro lado, un acuerdo de nivel de servicio o SLA es un contrato entre el usuario y su proveedor de servicios. En él que se definen las garantías mínimas de calidad que debe cumplir la red: ancho de banda mínimo, prioridad del tráfico, latencia máxima, etc. En nuestro caso concreto, el de este TFG, nos vamos a centrar en anchos de banda garantizados ofrecidos por un operador de red o proveedor de servicios a los usuarios.

Por lo tanto, en este proyecto, la asignación de ancho de banda se realiza en un esquema de asignación de ancho de banda limitado basado en el algoritmo IPACT, pero que integra el soporte de varios SLAs, con diferentes restricciones de ancho de banda garantizado. La asignación se llevará a cabo de forma dinámica en la capa MAC (*Medium Access Control*)

del OLT. Los parámetros fundamentales, representados como vectores de tamaño igual al número de ONTs son los siguientes [12]:

- $B_{demand}^{ont_i}$: Ancho de banda demandado total por la ONT_i, es decir el tamaño de sus colas.
- B_{max} : Ancho de banda máximo que puede ser asignado a las ONTs cada una asociada al mismo SLA o a distintos.

Estos parámetros son determinantes a la hora de asignar el ancho de banda. En concreto, se presentan dos casos en la asignación de ancho de banda:

Sí $B_{demand}^{onu_i} \leq B_{max}$, el ancho de banda demandado para la ONT_i es menor que el máximo para esa ONT asociada al SLA. Por lo tanto, el ancho de banda asignado en este caso, $B_{allocated}^{onu_i}$, corresponde con el demandado por dicha ONT, tal y como se observa en la Ecuación 1:

$$B_{allocated}^{onu_i} = B_{demand}^{onu_i} \qquad \text{Ecuación 1}$$

Sí $B_{demand}^{onu_i} > B_{max}$, el ancho de banda demandado para la ONT_i es mayor que el máximo para esa ONT asociada al SLA. En este caso, el ancho de banda asignado corresponde con el máximo asignado para dicha ONT según al SLA que corresponda, tal y como se observa en la Ecuación 2:

$$B_{allocated}^{onu_i} = B_{max} \qquad \text{Ecuación 2}$$

De esta manera, se asigna el ancho de banda a cada ONT dependiendo de la demanda de sus colas en ese ciclo en concreto. Este esquema limitado de ancho de banda se utiliza porque después de hacer un análisis de todos los esquemas en se demostró que se trata del más eficiente. Este enfoque garantiza no solo una distribución equitativa y controlada de los recursos de red, sino también una gestión eficiente en escenarios con múltiples niveles de servicio. Con el fin de potenciar esta propuesta y algoritmo, se incorporarán técnicas avanzadas de inteligencia artificial, en particular métodos basados en aprendizaje por refuerzo, cuya aplicación y beneficios se detallarán en apartados posteriores.

3.4 Definición de escenarios de prueba en el simulador EPON

Para evaluar el desempeño del modelo de asignación de ancho de banda propuesto al que se le integrarán técnicas de aprendizaje por refuerzo, se elaboraron cuatro escenarios distintos que simulan situaciones desde ideales a más complejas en el ámbito de las redes EPON y 10G-EPON. Estos escenarios nos permiten estudiar la capacidad del modelo DRL para aprender a la hora de enfrentarse a casos simples o casos más complejos y cercanos a la realidad.

3.4.1 Escenario 1: ONTs con SLAs garantizados y tráfico simétrico

En este primer caso de estudio, todas las ONTs comparten el mismo nivel de SLA y generan cargas de tráfico simétricas, de modo que la carga promedio es idéntica para cada una de ellas. Esta configuración permite analizar el comportamiento del sistema en condiciones muy equilibradas.

3.4.2 Escenario 2: ONTs con SLAs garantizados y tráfico asimétrico

Las ONTs conservan el mismo SLA, pero presentan cargas de tráfico diferentes entre sí, es decir, el tráfico de cada una de las ONT es asimétrico y diferente entre sí. Para ello se define un vector de tamaño igual al número de ONTs que toma valores aleatorios entre un mínimo y máximo en su tasa de transmisión.

3.4.3 Escenario 3: ONTs con SLAs garantizados y dinámicos

El escenario comienza con SLAs fijos para las ONTs, pero en un punto específico del tiempo, a los usuarios (ONTs) se les modifica su SLA. Diferenciamos dos casos en este escenario. En el primer escenario, el SLA se modifica pasando en un instante aleatorio a un valor diferente, pero igual para todos. En el caso 2, el SLA de cada una de las ONTs se modifica a un valor aleatorio diferente al inicial, pero además en cada una de ellas en un instante aleatorio. Este segundo caso es muy extremo y dinámico, por lo que sirve para demostrar el funcionamiento del modelo en casos extremos.

3.4.4 Escenario 4: ONTs con SLAs garantizados y no garantizados

En este caso diferenciamos en dos casos. Las ONTs garantizadas corresponderían con clientes de operadores tradicionales (OMRs, Operadores Móviles de Red) mientras que las ONTs flexibles pertenecen a clientes con operadores virtuales (OMVs, Operadores Móviles Virtuales). La diferencia principal es que los operadores tradicionales poseen su propia infraestructura, mientras que los operadores virtuales alquilan esta infraestructura. Esto hace que su servicio sea generalmente más barato, pero con menos garantías [19][20]. Algunos ejemplos son el operador virtual *Lowi* que pertenece al operador tradicional *Vodaphone* o *Jazztel* de *Orange.*

Así pues, este último escenario incluye una mezcla de ONTs con SLA garantizados (de operadores tradicionales) y otras con políticas flexibles (de operadores virtuales), asignando primero el ancho de banda a las ONTs con SLA asegurado y distribuyendo el ancho de banda restante entre las flexibles, pero garantizando siempre un ancho de banda asignado mínimo. Cabe destacar que, aunque se fije un ancho de banda máximo garantizado también en las ONTs flexibles, no es necesario que este se pueda garantizar siempre, en caso de que la red tenga mucha demanda, se intentará abastecer siempre primero a las ONTs garantizadas. Este caso es muy realista en el contexto real de operadores y proveedores de servicio hoy en día y resulta de gran interés a la hora de plantearlo y analizarlo.

3.5 Introducción al DRL (Deep Reinforcement Learning)

El aprendizaje por refuerzo profundo, también referido como *Deep Reinforcement Learning* (DRL) en inglés, combina los principios del aprendizaje por refuerzo (*Reinforcement Learning*, RL) con las redes neuronales profundas (*Deep Neural Networks*, DNN). Este modelo de aprendizaje tiene como objetivo el desarrollo de agentes, es decir, entidades que toman decisiones, capaces de aprender políticas de decisión óptimas a partir de la información de entrada [2]. Actualmente, el aprendizaje por refuerzo profundo se emplea en diversas áreas. Algunos ejemplos son la robótica, videojuegos y telecomunicaciones.

3.5.1 Principales elementos de los modelos de aprendizaje por refuerzo

En un modelo de aprendizaje por refuerzo, el agente interactúa con su entorno de la siguiente forma: capta un estado, realiza una acción y recibe una recompensa (*reward*) según lo buena que haya sido la decisión tomada. El agente buscará obtener la máxima recompensa posible mediante sus acciones, mejorando gradualmente. A continuación, se describe en más detalle los distintos elementos de este tipo de modelos [2].

3.5.1.1 El agente y su entorno

El agente es una entidad autónoma que percibe su entorno a través de una serie de sensores y actúa sobre este a través de actuadores. El agente irá aprendiendo de las interacciones con su entorno, este aprendizaje se conseguirá mediante una recompensa, concepto que se explicará más adelante. Un agente puede ser virtual, como aquellos diseñados para aprender a jugar videojuegos como los famosos juegos de *Atari* o el ajedrez o existir de forma física, como por ejemplo un robot camarero. En la Figura 2 se muestra un robot que funciona como agente de un sistema de DRL.

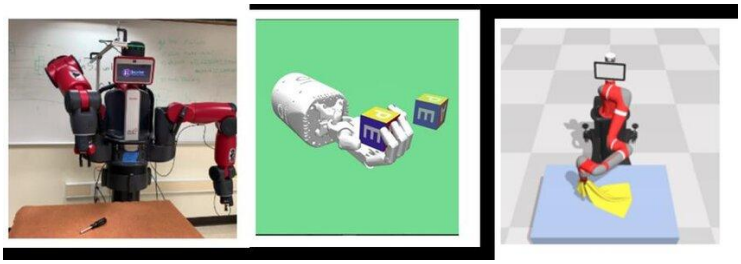


Figura 2. Robot DRL interaccionando con su entorno (Extraído de [21]).

Por otro lado, el entorno es el *espacio* con el que el agente interacciona y que cambia según las decisiones que va tomando el agente. El entorno envía su estado al agente, es decir, como se encuentra en ese momento una vez realizadas las acciones del agente. Todo aquello que el agente no pueda cambiar de manera arbitraria se considera fuera de él y, por tanto, parte de su entorno. En la Figura 3 se muestra un conjunto de escenarios de DRL, que consisten en distintas pantallas de los juegos de *Atari*.

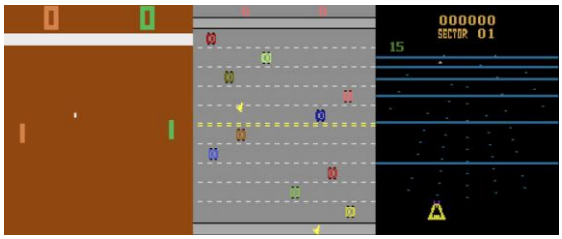


Figura 3. Escenarios DRL en videojuegos de Atari (Extraído de [22]).

Las interacciones entre el agente y el entorno siguen la estructura un Proceso de Decisión de Markov (MDP, *Markov Decision Process*) [23], una estructura matemática que permite describir decisiones secuenciales y que incluye los elementos siguientes: un conjunto de estados, un conjunto de acciones, una función de transición estocástica, que determine la probabilidad de pasar de un estado al siguiente, una función de recompensa inmediata y un factor de descuento, que permita priorizar las recompensas inmediatas sobre las futuras.

El comportamiento del agente se describe mediante una política (concepto descrito con mayor profundidad en el Apartado 3.5.2.1) El objetivo, es encontrar una política óptima que maximice el valor esperado del estado [2]. Este proceso general se muestra de manera visual en la Figura 4.

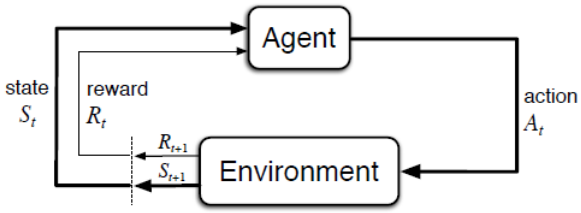


Figura 4. Esquema de la interacción agente-estado en un agente de RL (Extraído de [2]).

3.5.1.2 La acción y el estado.

Las acciones son las decisiones que puede tomar el agente. Todas ellas se incluyen en el llamado espacio de acción. Estas acciones guiarán al agente a alcanzar su objetivo y deben ser lo más eficientes posibles. Un ejemplo simple es el caso de un robot limpiador, sus acciones pueden ser girara la izquierda, derecha o seguir de frente y pararse cuando detecte un obstáculo [2].

El estado del entorno es la situación en la que el entorno se encuentra. Pueden estar determinados por una serie de sensores o pueden ser descripciones simbólicas de objetos, es decir, entornos parcialmente observables. Los estados deben contener suficiente

información para que el agente tome decisiones informadas, sin embargo, deben ser lo suficientemente compactos para ser gestionables desde el punto de vista computacional [2].

3.5.1.3 La recompensa

La recompensa o *reward* se da después de cada acción al agente y mide lo buenas que son las decisiones de este. El objetivo del agente va a ser maximizar la suma total de recompensas.

3.5.2 Conceptos importantes en el aprendizaje por refuerzo

3.5.2.1 Políticas del agente DRL

La política, son las reglas que guían la toma de decisiones del agente, es decir, como el agente se comporta frente a distintas situaciones. Puede ser simple como una tabla de consulta o en otros casos pueden requerir cálculos complejos. Además, suelen ser funciones estocásticas [2]. En relación con el uso de los datos y de la política actual del modelo encontramos algoritmos *on-policy* y *off-policy*.

En los algoritmos *on-policy* el agente aprende usando los datos de las acciones que él mismo ha tomado según su política actual. Es un método estable y fácil de usar, pero no aprovecha bien los datos, porque cada acción se usa solo una vez.

Los algoritmos *off-policy* están basados en *Q-learning* y el agente aprende usando datos que habían recogido antes y se guardan en un *replay buffer*, aunque provengan de otra política. Esto permite aprovechar mejor los datos, pero puede ser menos estable y más complicado de usar [2][24].

3.5.2.2 La función de recompensa

La función de recompensa debe ser diseñada de forma muy cuidadosa ya que es esencial a la hora de optimizar el proceso de decisión. Esta función proporciona retroalimentación al agente sobre como de efectivas son las decisiones que toma. Estas funciones deben cumplir una serie de propiedades para ser adecuadas: la invarianza, es decir, que cualquier ajuste del *reward* no cambie la política óptima del agente; la interpretabilidad, lo que significa que la retroalimentación sea comprensible; y finalmente la *informatividad* [25].

3.5.2.3 La función de valor

La función de valor es esencial ya que nos permite evaluar el comportamiento del agente a largo plazo. La función de valor ayuda al agente a evaluar los estados o acciones basándose no solo en las recompensas inmediatas (mediante la función de recompensa), sino también en las recompensas futuras que se prevén [2].

3.5.3 Principales algoritmos de aprendizaje por refuerzo

Dentro de todos los algoritmos de aprendizaje por refuerzo que se pueden implementar en un sistema DRL, el empleado en este trabajo es el *Proximal Policy Optimization* o PPO. Para explicar y justificar la elección de este algoritmo, es conveniente definir los tipos de algoritmos más representativos que nos podemos encontrar en estos sistemas. Estos son:

3.5.3.1 Q-Learning

El método *Q-Learning* surge en 1989 como un algoritmo *model-free,* es decir, el agente no necesita conocer el modelo del entorno, simplemente se guía por las interacciones que tiene con él [26][27]. En él, el comportamiento del agente se mide en función del valor de la función Q o *Q-function*, que representa el valor de la recompensa acumulada según las acciones del agente, o expresado de otra forma, la utilidad acumulada de las acciones que el agente puede tomar [28].

3.5.3.2 SARSA (State-Action-Reward-State-Action)

Este algoritmo nace como una modificación del algoritmo *Q-Learning* años después y se le da el nombre de *Modified Connectionist Q-Learning* (MCQ-L). Es Richard Sutton quien en 1998 le da el nombre *State-Action-Reward-State-Action*. La principal diferencia entre SARSA y Q-Learning e que este primero es un método *on-policy*, mientras que el segundo es *off-policy*. Debido a esto SARSA correrá menos riesgos a la hora de realizar acciones [2].

3.5.3.3 Deep Q-Network (DQN)

Este algoritmo es una modificación del *Q-Learning* que emplea una red convolucional profunda que aproxima la función Q a partir de secuencias de imágenes. Además de esto, el uso de repetición de experiencias (*experience replay*) y su aprendizaje es estable. Estas cualidades y el resto de sus características hacen que sea de gran utilidad en entornos de píxeles como videojuegos[29].

3.5.3.4 Policy Gradient Methods

Este enfoque es una modificación de un problema de aprendizaje supervisado ampliamente utilizado en el campo de la robótica. Su uso en la programación de robots se explica por una de sus características más importantes y es que, permite cambios pequeños y graduales en los agentes evitando cambios drásticos, es decir, si un robot necesita levantar un objeto frágil debe hacerlo poco a poco, si hace movimientos bruscos el objeto se cae. Por otro lado, si bien puede funcionar como *model-free* podemos dar al agente nociones básicas sobre su propio comportamiento, limitando las acciones de este (decirle al robot que solo puede mover el brazo en un cierto ángulo) [30][31].

3.5.3.5 Proximal Policy Optimization (PPO)

El fácil manejo y su eficacia han sido elementos clave en el éxito del PPO. Como emplea técnicas que acotan los límites en los que se desarrollan los cambios en su política, se limitan drásticamente las bajadas de rendimiento a lo largo del entrenamiento. Así, se muestra eficaz en diferentes entornos de simulación y es un algoritmo de referencia para la resolución de problemas RL [32].

El algoritmo empleado en este trabajo será un *Proximal Policy Optimization* (PPO), que introduce restricciones suaves en las actualizaciones de política, garantizando un aprendizaje más estable. PPO ha ganado gran popularidad debido a su balance entre rendimiento y simplicidad de implementación [3]. Sin embargo, en trabajos de investigación futuros, se puede pensar en la integración y comparativa con alguno de los otros algoritmos descritos previamente.

3.5.4 Ventajas en el uso de Deep Reinforcement Learning

Los métodos DRL presentan grandes ventajas frente a los métodos RL (permitiendo trabajar en entornos más amplios) y otros métodos tradicionales empleados en la gestión de recursos. Algunas de estas ventajas se mencionan a continuación [33]:

- En la actualidad muchos de estos problemas se resuelven usando heurísticas (reglas empíricas). Este proceso puede ser tedioso ya que si algún aspecto del problema, como la carga de trabajo o la métrica de interés se modifica, se deberían emplear nuevas reglas. En contraposición, los métodos de DRL se adaptan mejor a los cambios.

- En los sistemas de gestión de recursos las decisiones tomadas son a menudo repetitivos, lo que es muy útil a la hora de obtener datos de entrenamiento [34].
- Aprender estrategias de gestión de recursos usando directamente la experiencia, puede ser útil a la hora de presentar resultados más realistas que los métodos basados en reglas empíricas.

3.6 Integración del agente DRL para gestionar ancho de banda en redes 10G-EPON

En este apartado de la memoria se explica cómo se implementado el agente DRL para gestionar ancho de banda en redes 10G-EPON. Nuestro sistema consta de un agente, el algoritmo que será el encargado de ir tomando decisiones, y de un *environment* o entorno con el que el agente interaccionará que cuenta con un número de ONTs.

El objetivo concreto en este caso es simular entornos que representen redes 10G-EPON con diferentes condiciones de tráfico y calidad de servicio. En cada una de estas redes las diferencias fundamentales serán la diferencia en los patrones de tráfico que presenta cada ONT y sus SLAs asociados.

Así pues, la recompensa o *reward* ofrecida al agente, buscará maximizar el uso del ancho de banda en la red en función de esos patrones de tráfico y niveles de calidad de servicio. Para ello el agente irá realizando acciones sobre el entorno, por lo que según lo útiles que sean estas acciones a la hora de optimizar el uso de ancho de banda, recibirá un *reward* más alto o menos. En la Figura 5 se muestra el comportamiento de nuestro agente y entorno. Observamos que las acciones del agente consisten en asignar ancho de banda a las ONTs, es decir, su acción consiste en asignar un valor *B_alloc* a cada una de las ONTs. Por otro lado, dentro del entorno se establece el vector *B_demand* que tiene un tamaño igual al número de ONT y que se refiere al tamaño de las colas de cada una de las ONTs del sistema, esto es, la demanda de ancho de banda en tiempo real de cada ONT.

También podemos observar que dentro del entorno hay tres factores principales que influyen en que el entorno considere una asignación buena o mala y que por tanto de más o menos *reward*. Estos factores son: la reducción del tamaño de la cola tras la asignación, el uso efectivo del ancho de banda según los requisitos de calidad de servicio (SLAs en nuestro caso) y que no se supere la capacidad de máxima de la red. Una descripción más

detallada de la función de recompensa y todos sus elementos se explicará con detalle en los siguientes apartados, para cada uno de los escenarios de red concretos.

De este modo, el agente recibe el estado en el que se encuentra el entorno, así como el *reward* y dependiendo de eso va optimizando sus decisiones. Esta interacción se describe de forma gráfica en la Figura 5.

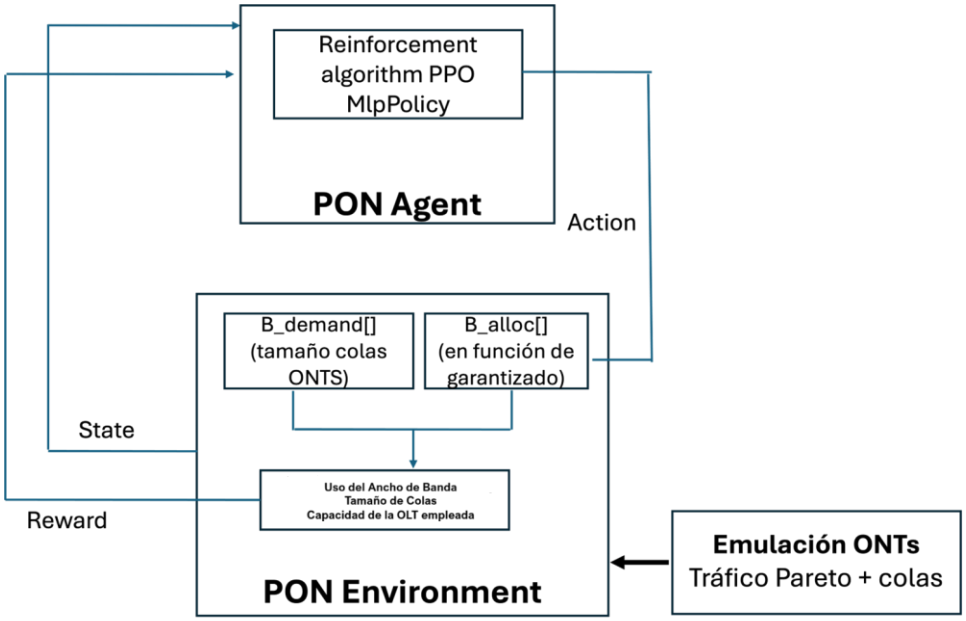


Figura 5. Esquema general de elementos que componen este modelo de DRL.

Para facilitar la comprensión del simulador he considerado que sería útil realizar una tabla que resumiese a que parte de un modelo DRL se corresponde cada uno de los elementos del simulador. Esta relación queda descrita en la Tabla 1.

Elemento DRL	Elemento del Simulador
Entorno	Espacio conformado por 16 ONTs (cuyas características varían dependiendo del escenario) con capacidad máxima de 1Gbps y una OLT de capacidad 10 Gbps. El tráfico de las ONTs sigue una distribución de Pareto.
Agente	Algoritmo que aprende de sus decisiones al interactuar con el entorno, este se dedica a asignar ancho de banda a las ONTs

Acción	Asignación de ancho de banda a cada una de las ONTs por parte del agente.
Estado	Como se encuentran las ONTs, su cola, cuanto tráfico demandan. En general como se encuentra el conjunto de la red, es decir, si la red PON está muy saturada o al contrario si hay poca carga
Recompensa	Premio de valor numérico que le damos al agente cuando asigna el ancho de banda de forma correcta y eficiente en función de los requisitos de QoS aplicados. El valor de la recompensa aumenta cuanto mejor asigne el ancho de banda el agente.

Tabla 1. Resumen de cada elemento de nuestro simulador según su rol en el modelo DRL

3.6.1 Organización de los ficheros del modelo DRL en redes 10G-EPON

En esta sección se describe la organización y ficheros que conforman el simulador de la red 10G-EPON integrado en el modelo DRL (Figura 6) disponible en un repositorio de Github [35]. Todos ellos están incluidos dentro de una carpeta, denominada *Redes_opticas_escenarios*. En primer lugar, el primer fichero que encontramos es el fichero *Escenarios.ipynb*. Este fichero tiene como función inicializar el entorno y el agente de DRL, iniciar el entrenamiento y test del modelo y finalmente representar los resultados obtenidos.

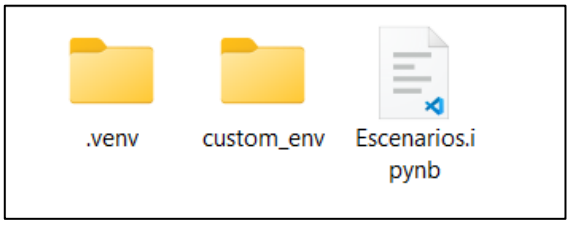


Figura 6. Fichero y carpetas que encontramos en *Redes_opticas_escenarios.*

Si nos metemos en la carpeta *custom_env*, encontraremos aquellos ficheros que han servido para desarrollar los entornos con los que interactúa el agente (Figura 7). Por un lado, en el fichero *traficoparetopython.py* se encuentra la fuente de tráfico de Pareto, que ha sido modificada para adaptarse a los distintos escenarios. Por otro lado, se encuentran los ficheros *redes_opticas_env.py*, donde se definen cada uno de los entornos, que corresponden con los cuatro escenarios diferentes configurados.

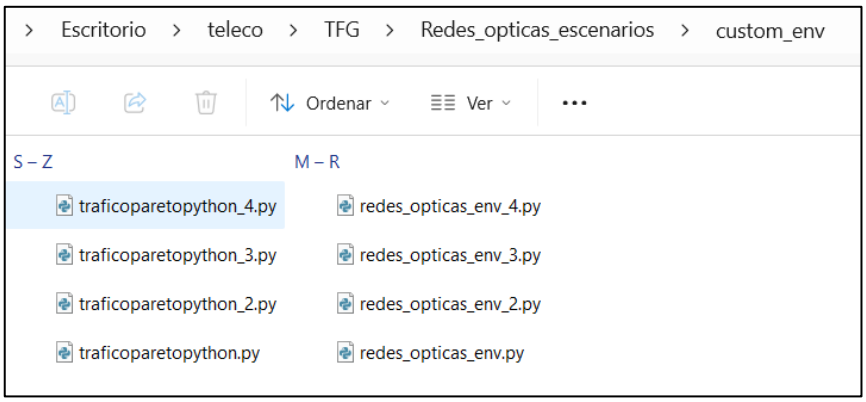


Figura 7. Ficheros que describen los distintos entornos en *custom_env*.

Las funciones que constituyen estos ficheros serán descritas en mayor profundidad en los próximos apartados.

3.6.2 Definición del Entorno

Nuestro entorno es un simulador de redes 10G-EPON. Esta red cuenta con una serie de ONTs, por defecto 16, aunque el sistema es escalable y se han hecho pruebas con diferente número de usuarios. Para simular el comportamiento de este tipo de redes ha sido necesario definir una serie de parámetros en relación con la gestión del ancho de banda disponible. A continuación, en la Tabla 2, se presentan los parámetros que han permitido definir y configurar el simulador 10G-EPON. Algunos de estos parámetros son propios del estándar EPON y han sido explicados en apartados previos, en la asignación de ancho de anda.

Parámetros	Valores por Defecto	Descripción
num_ont	16	Número de ONTs en la red.
temp_ciclo	2 ms	Duración de un ciclo máximo en EPON
n_ciclos	400	Número total de ciclos de simulación
TxRate	10Gbps	Tasa de transmisión máxima en 10G-EPON
B_alloc	_____	Ancho de banda asignado a cada ONT en un ciclo de forma dinámica
B_available	TxRate * temp_ciclo	Ancho de banda total disponible en cada ciclo
B_max	_____	Ancho de banda máximo asignable a cada ONT

B_demand	_____	Ancho de banda demandado por cada ONT (colas)
trafico_entrada	_____	Tráfico generado por cada ONT
prev_demand	_____	Demanda anterior de cada ONT, útil para seguimiento
max_queue	1.6 MB por ONT	Capacidad máxima de las colas de cada ONT
b_excess	_____	Cantidad de ancho de banda que el agente intentó asignar por encima de la capacidad disponible del OLT

Tabla 2. Parámetros que definen el entorno DRL

3.6.2.1 Declaración del entorno

La llamada al entorno se lleva a cabo al inicio del script *Escenarios.ipynb* (Figura 8), mediante la función de la librería *Gymnasium make_env()*. Esta función recibe los siguientes parámetros:

- *num_ont:* número de ONTs del entorno.
- *TxRate:* tasa de transmisión máxima del enlace, 10 Gpbs en este caso por ser una red 10G-EPON.
- *B_guaranteed:* lista de anchos de banda garantizados por cada ONT (en Mbps).
- *n_ciclos:* número de ciclos de simulación.
- *rank:* identificar entornos paralelos, es decir, cuando ejecutamos varios entornos iguales al mismo tiempo (en este trabajo no lo vamos a emplear).
- *seed:* semilla para que los resultados sean reproducibles, si se vuelve a ejecutar el código con la misma semilla se obtiene el mismo comportamiento aleatorio.

Esta función *make_env()* devuelve otra función (*_init*). Dentro de la función *_init* se crea una instancia del entorno *RedesOpticasEnv,* a la que se le pasan los siguientes parámetros:

- *render_mode=None:* no se mostrará visualización del entorno.
- *seed=seed:* para que la simulación sea reproducible.
- *num_ont, TxRate, B_guaranteed, n_ciclos:* los parámetros que se habían pasado ya a la función *make_env.*

Además, tal y como observa en la Figura 8, la clase *Monitor*, que también empleamos en la función *make_env*, sirve para registrar las recompensas y estadísticas mientras el agente interactúa con el entorno. Los logs se guardan en *./reward_logs/reward.* Finalmente, la función *_init* devuelve el entorno listo para usarse.

```
def make_env(num_ont, TxRate=10e9,B_guaranteed=[], n_ciclos=200, rank=0, seed=0):
    def _init():
        env = RedesOpticasEnv1(render_mode=None,seed=seed, num_ont=num_ont, TxRate=TxRate, B_guaranteed=B_guaranteed,n_ciclos=n_ciclos)
        env = Monitor(env, filename="./reward_logs/reward")
        return env
    return _init
```

Figura 8. Llamada al entorno en el *script Escenarios.ipynb.*

3.6.2.2 Funciones que definen el entorno

A continuación, se presentan las funciones fundamentales que definen el entorno de *Gymnasium RedesOpticasEnv*, definido en el script *redes_opticas_env.py*, tal y como se mostraba en la Figura 8. El entorno simula la asignación de ancho de banda en una red 10G-EPON ciclo tras ciclo y será modificado para satisfacer las directrices de cada uno de los cuatro escenarios definidos en este trabajo.

3.6.2.2.1 Función _get_obs

La función *_get_obs* permite obtener el estado actual en el que se encuentra el sistema. Para ello, devuelve el nivel de congestión de cada ONT (cola actual/cola máxima). Esta información se usa como entrada para el agente, de forma que la toma de decisiones se realice correctamente, tal y como se muestra en la Figura 9.

```
def _get_obs(self):
      return self.B_demand / self.max_queue
```

Figura 9. Función *_get_obs* del entorno.

3.6.2.2.2 Función _get_info

La función *_get_info* devuelve un diccionario con los datos clave, tal y como se muestra en el código de la Figura 10. Tal y como se observa en la Figura 10, las variables más importantes son: *B_demand* (bits), que corresponde al ancho de banda demandado por una ONT en un ciclo concreto, *B_alloc* (bits), que es el ancho de banda asignado a cada ONT, *trafico_entrada generado* por cada ONT, *prev_demand* (bits) demanda anterior de cada

ONT, *B_max* (bits) máximo asignable a cada ONT, *B_available* (bits) ancho de banda total disponible en cada ciclo y *max_cola* (bits) tamaño máximo de la cola de cada ONT.

```
def _get_info(self):
    return {
        "B_demand": self.B_demand.copy(),
        "B_alloc": self.B_alloc.copy(),
        "trafico_entrada": self.trafico_entrada.copy(),
        "prev_demand": self.prev_demand.copy(),
        "B_max": self.B_max.copy(),
        "B_available": self.B_available,
        "max_cola": self.max_queue.copy()}
```

Figura 10. Función *_get_info* del entorno.

3.6.2.2.3 Función calculate_reward

A continuación, se muestra la descripción de la función *_calculate_reward* correspondiente a los cuatro escenarios. En concreto, los tres primeros escenarios comparten la misma función de recompensa, dado que únicamente varían en el patrón de tráfico y las condiciones de SLA que pueden abordarse sin modificar dicha función. En cambio, el escenario 4 resulta más complejo, pues incorpora políticas de calidad de servicio que obligan a redefinir la función de recompensa para integrar parámetros adicionales relacionados con dicha QoS. A continuación, se describen con profundidad las funciones de recompensa para dichos escenarios.

Escenario 1, 2 y 3: Descripción de la función de recompensa (*calculate_reward*).

En esta función de recompensa, se evalúa lo buena que ha sido la decisión tomada por el agente, calculado la recompensa que merece. Para ello, como se ha mencionado previamente, se evalúa el estado mediante tres criterios: la reducción de cola (si disminuyen las colas, se premia), la eficiencia en el uso del ancho de banda en función de los requisitos de QoS estipulados (SLAs en este caso) y el exceso de ancho de banda asignado (si se pide más de lo disponible, esto es, 10 Gpbs se penaliza). A continuación, se procede a detallar más a fondo como se programan en el código estas condiciones.

En primer lugar, la reducción de la cola se obtiene mediante dos pasos, tal y como se muestra en la Figura 11. El primero es calcular la diferencia entre la demanda previa y la demanda actual, donde se definen las variables:

- *self.prev_demand:* representa el ancho de banda demandado en el paso anterior.
- *self.B_demand:* representa el ancho de banda demandado en el *step* actual.

El resultado debe ser interpretado de la siguiente forma:

- Si es positivo la cola se ha reducido.
- Si es negativo la cola ha aumentado.
- Si es cero no hubo cambio en la cola.

El segundo paso es normalizar la diferencia para que no dependa del tamaño total de la cola. Para ello se divide *delta_cola* entre *self.max_queue*, es decir, el tamaño máximo posible de la cola. Luego se aplica *np.mean()*, que calcula la media. De esta forma el resultado es un valor medio normalizado.

```
def _calculate_reward (self):
      delta_cola = self.prev_demand - self.B_demand
      delta_cola_norm = np.mean(delta_cola / self.max_queue)
```

Figura 11. Factor *delta_cola_norm* del *reward.*

En segundo lugar, el uso efectivo del ancho de banda (Figura 12) se calcula dividiendo la suma del ancho de banda total asignado a todas las ONTs entre la suma del ancho de banda máximo de cada ONT. Este ancho de banda máximos se corresponde con el ancho de banda garantizado del SLA al que está asociado esta ONT, y, por lo tanto, es el parámetro de calidad de servicio más importante en nuestro caso de uso.

```
        uso_bw = np.sum(self.B_alloc) / np.sum(self.B_max)
```

Figura 12. Factor del *reward uso_bw*.

En tercer lugar, es importante que no se supere la capacidad de la OLT (Figura 13). En primer lugar, se calcula *max_excess* como la diferencia entre el ancho de banda máximo que se puede asignar a las ONTs (*np.sum(self.B_max)*) y la capacidad máxima disponible en la red (*self.B_available*). Luego, se obtiene *lim_olt* dividiendo el exceso de tráfico actual (*self.b_excess*) entre dicho margen, lo que normaliza el valor. Después de calcular *lim_olt*, se reinicia *self.b_excess* a 0 para el siguiente paso de simulación. Esto asegura que solo se considere el exceso actual.

```
max_excess = np.sum(self.B_max) - self.B_available
lim_olt = self.b_excess / max_excess
self.b_excess = 0
```

Figura 13. Explicación del cálculo del factor *lim_olt* del *reward.*

Debido a que el *reward* se compone por distintos factores y no por un solo término, es preciso dar a cada uno de estos un peso relacionado directamente con la importancia que tiene cada uno en el correcto funcionamiento del sistema. Como podemos ver a continuación en la Figura 14, el factor más importante que hemos considerado en el cálculo del *reward* es el *uso_bw* es decir, garantizar el ancho de banda asociado al SLA de la ONT. Esto se debe a que en los escenarios 1, 2 y 3 es esencial que se cubra la demanda establecida según el SLA, de hecho, podríamos decir que es el factor fundamental que guía el comportamiento del agente. Mientras, con un valor menor igual a 0.5, se encuentran *delta_cola_norm* y *lim_olt* ya que su efecto se considera menos crítico a la hora de guiar el comportamiento, aunque sí que contribuyen a que este sea más eficiente y realista.

Estos pesos se eligieron como una aproximación inicial basada en la teoría y en simulaciones realizadas, es decir, no representan necesariamente los valores óptimos que el sistema debería emplear. En trabajos futuros se prevé realizar un estudio más avanzado para optimizarlos, una vez se haya comprobado que el agente funciona correctamente en los distintos escenarios.

```
reward = (
  0.5 * delta_cola_norm +
  1.3 * uso_bw +
  0.5 * lim_olt)
 return float(reward)
```

Figura 14. Configuración final del *reward.*

A continuación, se muestra el análisis del conjunto de simulaciones realizadas con diferentes combinaciones de pesos. Cabe destacar que para realizar este análisis fue necesario tener en cuenta la curva de *reward* para estudiar como los pesos influían en el entrenamiento. Esta curva es una medida empleada para el análisis de los modelos DRL y representa la evolución del *reward* que nuestro agente recibe a medida que aprende. Lo

ideal por tanto es que crezca rápido hasta llegar a un valor óptimo que funcione correctamente.

En cuanto al estado de la red 10G-EPON, las distintas configuraciones se evaluaron en una situación de saturación de la red, en la que *B_guaranteed* tenía un valor de 800 Mbps para todas las ONTs (SLA) y la carga media de las ONTs era de 0.9 (ONTs transmitían a 900 Mbps, siendo el máximo 1 Gpbs). De esta forma nos asegurábamos de que todos los factores del *reward* tuviesen un impacto en el aprendizaje. En esta prueba, al requerirse una garantía de 800 Mbps por ONT, si todas demandaran simultáneamente este valor —lo cual ocurrirá siempre, dado que la carga media es de 900 Mbps— se excedería la capacidad de la OLT, ya que la demanda total sería de 12,8 Gbps, superando su límite máximo de 10 Gbps. Esto hace que el factor *lim_olt* sea importante, ya que tiene que asegurarse de que este límite no se sobrepase. En la Tabla 3 observamos el análisis de las curvas de *reward* de las simulaciones para las combinaciones de los distintos pesos analizados. Cabe destacar que estos datos se presentan a modo de resumen, pero que la elección se realizó tras un análisis visual de las gráficas. En verde se señalan los pesos finales seleccionados.

Pesos	**Reward Inicial**	**Reward Final**	**Ganancia Total**	**Steps Convergencia**	**Pendiente**
delta_cola = 0.5 *uso_bw* = 0.5 *lim_olt* = 0.5	90	500	410	35.000	11.714
delta_cola = 0.1 *uso_bw* = 1.8 *lim_olt* = 0.1	230	600	370	35.000	10.571
delta_cola = 0.2 *uso_bw* = 1.5 *lim_olt* = 0.2	200	550	350	38.000	9.211
delta_cola = 0.5 *uso_bw* = 1.3 *lim_olt* = 0.5	170	600	430	42.000	10.238
delta_cola = 0.5 *uso_bw* = 0.5 *lim_olt* = 1.3	64	158	94	25.000	3.760
delta_cola = 1.3 *uso_bw* = 0.5 *lim_olt* = 0.5	65	350	285	30.000	9.500

Tabla 3. Resumen de análisis curva de *reward* bajo distintos pesos.

Como se observa en la Tabla 3 la ganancia de la cuarta simulación (la que corresponde a los pesos escogidos) es la más alta, aunque tarde un poco más en converger en el valor óptimo (42000 *timesteps*). La ganancia es el resultado de la operación *reward_final-reward_inicial*. Es decir, como aumenta el *reward* que el agente va obteniendo desde el inicio del entrenamiento. Además, la pendiente de la gráfica antes de converger es más acentuada que en los otros casos (10,24), lo que indica que la gráfica se aproxima mejor que la mayoría de los experimentos a la forma ideal. A continuación, se muestra la curva de *reward* en las Figuras 15 y 16. La Figura 15 muestra la curva con pesos seleccionados y la Figura 16 con el peor resultado, de forma que se puede ver gráficamente la mejora.

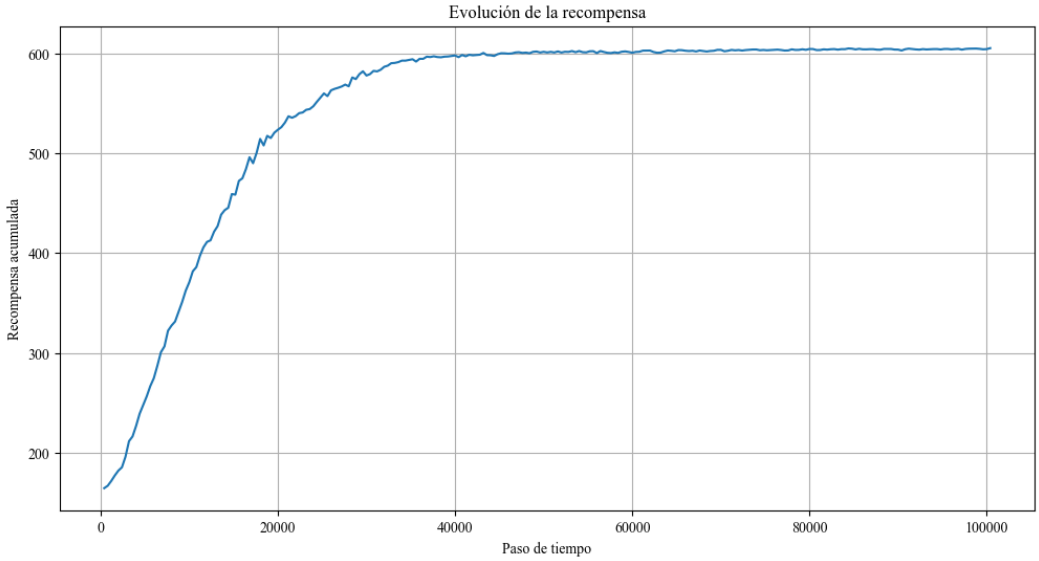


Figura 15. Curva de *reward* con los pesos seleccionados.

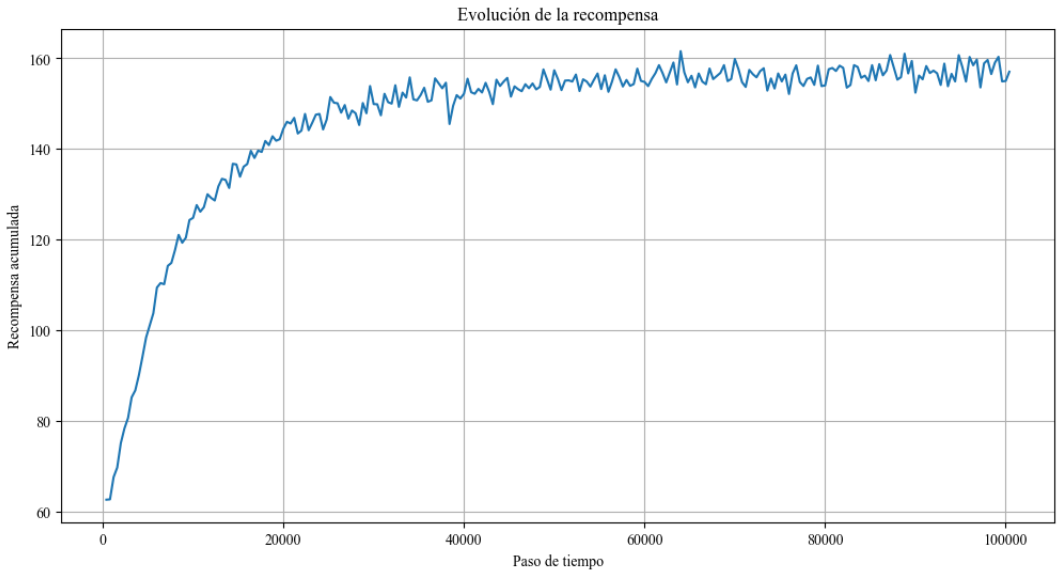


Figura 16. Curva de *reward* en el peor caso.

Escenario 4: Descripción de la función de recompensa (*calculate_reward*).

En este apartado se procede a detallar las diferencias esenciales en el método *calculate_reward* del escenario 4. En este caso el entorno se modifica de forma que definimos dos tipos de ONTs con comportamientos distintos. Por un lado, las ONT garantizadas, que emulan clientes con operadores tradicionales y por otro las ONT flexibles que corresponderían con clientes de operadores virtuales. Este nuevo planteamiento más realista dificulta la toma de decisiones del agente, que debe comportante distinto dependiendo del tipo de ONT al que quiera abastecer. En este caso concreto se establecen un cierto número de ONTs garantizadas y otro conjunto de ONTs flexibles, en concreto hemos tomado 6 ONTs garantizadas y 10 flexibles, aunque este valor es configurable y puede ser modificado fácilmente.

Así pues, el cálculo del *reward* se vuelve también más complicado. Los factores que determinarán lo buena que ha sido la acción de la agente son: que no sobrepase la capacidad de la OLT (*olt_excess_reward*), que se satisfaga la demanda de las ONT garantizadas (*guaranteed_ont_reward*), que las ONT flexibles no reciban menos ancho de banda de su mínimo garantizado (*flex_ont_min_penalty*), que la demanda de ancho de banda de estas se satisface teniendo en cuenta que hay un ancho de banda garantizado máximo (*flex_ont_reward*) y que el reparto de ancho de banda entre las ONT flexibles sea equitativo (*fairness_index*). El primer factor de esta nueva recompensa, como se acaba de comentar es el *olt_excess_reward* (Figura 17) y se aplica a las ONT garantizadas y flexibles. Si la asignación de ancho de banda (*np.sum(self.B_alloc)*) supera la capacidad disponible (*B_available*), se registra ese exceso en *self.b_excess*. A partir de esto, se establece la recompensa del OLT. Es una recompensa booleana que, si hay exceso, vale 0 y si no lo hay vale 1.

```
def _calculate_reward(self):
        epsilon = 1e-9
        total = np.sum(self.B_alloc)
        if total > self.B_available + self.epsilon:
            self.b_excess = total - self.B_available
        else:
            self.b_excess = epsilon
        if self.b_excess > epsilon:
            self.olt_excess_reward = 0.0
        else:
            self.olt_excess_reward = 1.0
```

Figura 17. Factor *olt_excess_reward* del *reward* del escenario 4.

El siguiente es el factor *guaranteed_ont_reward* que solo se aplica a las ONTs garantizadas (Figura 18). Si la demanda es positiva, se recorta con el máximo permitido (*clipped_demand*). Si la asignación es al menos el 99% de esa demanda, se suma a la recompensa un valor proporcional al grado en el que se satisface la demanda.

```
        guaranteed_ont_reward = 0
        for i in range(self.num_guaranteed_ont):
            if self.B_demand[i] > self.epsilon:
                clipped_demand = min(self.B_demand[i], self.B_max[i])
                if self.B_alloc[i] > clipped_demand * (0.99 - self.epsilon):
                    guaranteed_ont_reward += min(self.B_alloc[i], clipped_demand) /
(clipped_demand + self.epsilon)
```

Figura 18. Factor *guaranteed_ont_reward* del *reward* del escenario 4.

En el caso de los ONTs flexibles la función tiene en cuenta dos factores principales (Figura 19). Por un lado, el *flex_ont_reward*, y por otro *flex_ont_min_penalty*. El factor *flex_ont_reward* mide qué proporción de su demanda reciben las ONTs flexibles premiando así las asignaciones que se acerquen a cubrir lo solicitado sin sobrepasar su límite máximo garantizado (*B_max*), dado por su SLA flexible asociado. Por otro lado, el factor de penalización *flex_ont_min_penalty* se aplica en dos casos:

- Cuando la demanda supera el umbral mínimo garantizado pero la asignación de ancho de banda (*B_alloc*) es menor que este mínimo (*B_min_ONT2*).

- Cuando la demanda (*B_demand*) está por debajo del umbral mínimo garantizado y la asignación inferior a la demanda, es decir, la ONT demanda menos ancho de banda que su mínimo garantizado (*B_min_ONT2*) pero no se le da dicha demanda. Para ello se calcula *self.B_demand[i] * 0.002* es decir el 0.2% de la demanda. De esta forma si la asignación (*B_alloc*) es menor que el 0.2% de esa demanda (es decir, prácticamente cero frente a lo que se necesita) se penaliza.

```
        flex_ont_reward = 0
        flex_ont_min_penalty = 0

        for i in range(self.num_guaranteed_ont, self.num_ont):
            clipped_demand = min(self.B_demand[i], self.B_max[i])
            if self.B_demand[i] > epsilon:
                flex_ont_reward += min(self.B_alloc[i], clipped_demand) / (clipped_demand +
self.epsilon)
            if self.B_demand[i] > self.B_min_ONT2 + epsilon and self.B_alloc[i] < self.B_min_ONT2
- self.epsilon:
                flex_ont_min_penalty += 1.0
            if self.B_demand[i] < self.B_min_ONT2 - epsilon and self.B_alloc[i] < self.B_demand[i]
* 0.002 - self.epsilon:
                flex_ont_min_penalty += 1.0
```

Figura 19. Factores de *reward* para onts flexibles del escenario 4.

Además, con el objetivo de garantizar ciertos niveles de equidad en la asignación de ancho de banda en las ONTs flexibles se añade al *reward* el factor *fairness_index* a través del JFI (*Jain Factor Index*) y el algoritmo *water-filling*. A continuación, se pasan a explicar cada uno de los métodos empleados.

El índice JFI es ampliamente utilizado para calcular la equidad en la asignación de recursos entre usuarios en un sistema de comunicaciones. Podemos interpretar su valor de la siguiente forma: si es cercano a 1 la asignación de recursos es muy justa, si por el contrario el índice es cercano a 0 la asignación es muy injusta [36].

Por otro lado, el algoritmo *water-filling* se emplea dentro de esquema DBA y es un método de asignación de recursos que busca distribuir capacidad de manera justa entre múltiples

ONTs, en este caso en las redes PON. Es un modelo cíclico cuya idea central consiste en llenar cada receptor hasta su demanda o hasta que se agote el recurso disponible, de manera similar a cómo se llenan recipientes de agua con niveles diferentes [37][38]. Este algoritmo nos permite ajustar la asignación, incrementando recursos a las ONT con menor carga hasta alcanzar su demanda, sin afectar negativamente a las ONTs que ya cumplen con su mínimo garantizado. Este algoritmo se aplica solo para el caso de las ONTs flexibles.

Para comprender mejor el funcionamiento del algoritmo planteo el siguiente ejemplo: una OLT tiene una capacidad de 6,4 Gbps, esta capacidad debe dividirse entre 10 ONTs sin sobrepasarse. En el caso de cumplir la condición de equidad estrictamente, lo ideal sería que cada ONT recibiese 640 Mbps, pero ¿y si algunas ONTs demandan menos de 640Mbps de media? Por ejemplo, si dos de las ONTs demandasen 400 Mbps, lo lógico sería darles lo que piden y distribuir el ancho de banda sobrante entre las ONTs que demandan más (hasta su máximo permitido). En este caso si 2 de las 10 ONTs demandan 400 Mbps, sobran 480Mbps ([640Mbps-400Mbps]*2) que podrían distribuirse entre ONTs que demanden 800Mbps, por ejemplo. El algoritmo de *water filling* resulta especialmente útil en escenarios donde la carga de las ONTs es asimétrica. Si bien el caso analizado no permite mostrar todo el potencial del algoritmo, es importante definirlo para este contexto específico, así como para situaciones en las que, en promedio y en distintos instantes, las cargas presentan ligeras variaciones debido a la naturaleza auto-semejante del tráfico.

Este algoritmo solo afectará a las ONTs flexibles y su implementación ha sido realizada de la siguiente forma. En primer lugar, se calcula la demanda de cada ONT, usando la siguiente función descrita en la Figura 20. De esta forma, nos aseguramos de que ninguna demanda exceda el ancho de banda máximo que se puede asignar a una ONT.

```
def water_filling_onts():
            demands = np.minimum(self.B_demand, self.B_max)
```

Figura 20. Cálculo de *B_demand* en función *water_filling_onts.*

Después, se identifican las ONTs flexibles (Figura 21) cuya demanda excede el umbral mínimo garantizado. Estas ONTs son candidatas para recibir la parte que sobra del ancho de banda disponible en el sistema. El set *remaining* se refiere a aquellas ONTs cuya demanda aún no ha sido atendida.

```
            flex_indices = [
                    i for i in range(self.num_guaranteed_ont, self.num_ont) if self.B_demand[i]
> self.B_min_ONT2 + self.epsilon
                    ]
                remaining = set(flex_indices)
```

Figura 21. Identificación de las ONT flexibles con demanda superior a 480Mbps.

A continuación, se calcula la cantidad de ancho de banda disponible (Figura 22), sumando la cantidad de ancho de banda actualmente asignada a los ONTs flexibles.

```
            available = np.sum(self.B_alloc[self.num_guaranteed_ont:])
```

Figura 22. Cálculo de la cantidad de ancho de banda disponible.

Dentro de un *while* (Figura 23), se calcula una cuota para las ONTs. Para esto se divide el ancho de banda total disponible entre el número de ONTs restantes (en el ejemplo propuesto anteriormente la cuota serían los 640 Mbps). El objetivo es buscar las ONTs cuya demanda es menor que la cuota (en el ejemplo anterior las dos ONTs que demandan 400Mbps) . Estas ONTs se retiran del conjunto *remaining*, es decir son ONTs que se deben atender y se restara su demanda del total. Cuando no hay ningún elemento cuya demanda se encuentre por debajo de la cuota (lo que serían las 8 ONTs restantes del ejemplo), el bucle se rompe. Las ONTs que queden en el *set remaining* serán las ONTs que más demanda tienen, y por tanto las que no han sido atendidas. Estas ONTs tendrán que compartir equitativamente el ancho de banda restante según el algoritmo *water-filling* (los 480Mbps restantes del ejemplo).

```
            while remaining:
                    quota = available / (len(remaining) + self.epsilon)
                    low_demand = [i for i in remaining if demands[i] < quota - self.epsilon]
                    if not low_demand:
                        break
                    for i in low_demand:
                        available -= demands[i]
                        remaining.remove(i)
                return list(remaining)
```

Figura 23. Cálculo de cuota de *water-filling* y reparto de ancho de banda entre ONTs

Por lo tanto, en la fórmula de la recompensa total (Figura 24) el término más relevante es el *lim_olt*, que tiene un peso de 2.1. Le sigue el factor *guaranteed_ont_reward* con un peso de 2.0, debido a la prioridad de atender a los usuarios que hayan adquirido servicios garantizados de un operador tradicional. Para las ONTs flexibles la penalización *flex_ont_min_penalty* tiene un peso negativo de -1.4 y una recompensa *flex_ont_reward* con un peso reducido de 0.25, ya que son usuarios menos prioritarios. Finalmente, el índice de equidad *fairness_index* recibe un peso significativo de 2.0,

```
reward = (
      2.1  * self.olt_excess_reward
    + 2.0  * guaranteed_ont_reward
    - 1.4  * flex_ont_min_penalty
    + 0.25 * flex_ont_reward
    + 2.0  * fairness_index
)
return float(reward)
```

Figura 24. Configuración del *reward* del escenario 4.

Además, para asegurarnos de que los pesos empleados en el *reward* iban de acuerdo con los resultados deseados, se empleó un *script* que analizó 59049 casos distintos de asignación de ancho de banda en los que se podía encontrar la red, asignando la recompensa correspondiente a cada caso y haciendo un *ranking* de las mejores obtenidas. De esta forma, si el *reward* tiene pesos adecuados, obtendrá una recompensa óptima cuando todas las ONTs garantizadas reciban lo que demandan sin superar su ancho de banda garantizado y todas las flexibles obtendrán un ancho de banda por encima de su mínimo garantizado y por debajo de su máximo en un nivel acorde con su demanda, teniendo en cuenta que si su demanda es menor que el mínimo se le asignará lo que pide. El *script* empleado para comprobar el correcto funcionamiento de la recompensa es análogo a la función *_calculate_reward()* del entorno, pero empleando una serie de valores predefinidos concretos respecto a las condiciones de los SLAs y las ONTs asociadas, de forma que no se tenga que explorar el *reward* que ofrecen todos los tipos de asignación posibles. A continuación, veremos cómo se han definido esta serie de parámetros, que definen como funciona el entorno en la Figura 25. En esta parte del código de prueba del *reward* se definen: *B_demand, B_available, B_max* y *B_min_ONT2*, así como el número de ONTs garantizadas y totales.

```
temp_ciclo = 0.002
B_demand = np.array([900e6*temp_ciclo] * 16)
num_guaranteed_ont = 6
num_ont = 16
TxRate = 10e9

B_available = TxRate * temp_ciclo     # = 20,000,000
B_max = np.array([600e6*temp_ciclo]*6 + [800e6*temp_ciclo]*10)
B_min_ONT2 = 800e6 * temp_ciclo * 0.6
temp_ciclo = 0.002
```

Figura 25. Definición de parámetros para el correcto funcionamiento de la prueba del *reward.*

Posteriormente como se observa en la Figura 26, se define la parte fija (las ONTs garantizados) y la parte variable (las ONTs flexibles), generando todas las combinaciones posibles de la parte variable usando *itertools.product.*

```
fixed_part = [600e6 * temp_ciclo]*6
num_fixed_elements = 6

variable_values = [0, 640e6* temp_ciclo, 800e6 * temp_ciclo]
num_variable_elements = 10

all_variable_combinations = list(itertools.product(variable_values, repeat=num_variable_elements))
```

Figura 26. Definición de la parte fija y variable de las ONTs para la prueba del *reward.*

A continuación, se itera sobre cada combinación variable y la concatena con la parte fija (Figura 27). Cada resultado se guarda en la lista *results.*

```
    best_reward = -float('inf')
    best_b_alloc = None
    results = []

    print(f"Total number of combinations to check: {len(all_variable_combinations)}")
    for combo in all_variable_combinations:
        current_b_alloc = np.array(fixed_part + list(combo))
        reward = calculate_reward(current_b_alloc, B_available, B_demand, B_max,
num_guaranteed_ont, num_ont, B_min_ONT2)
        results.append({'B_alloc': current_b_alloc.tolist(), 'reward': reward})
```

Figura 27. Iteración sobre cada combinación variable en la prueba del *reward.*

Finalmente, imprime la recompensa más alta encontrada. Además, ordena todos los resultados de mayor a menor recompensa y muestra las 5 mejores combinaciones de asignación de ancho de banda, es decir, con que asignaciones se obtienen los mejores *rewards*, como vemos en la Figura 28.

```
if reward > best_reward:
            best_reward = reward
            best_b_alloc = current_b_alloc

    print("\nExploration complete.")
    print(f"Highest reward found: {best_reward}")
    print(f"Corresponding B_alloc: {best_b_alloc}")
    results.sort(key=lambda x: x['reward'], reverse=True)
    print("\nTop 5 combinations:")
    for i in range(min(5, len(results))):
        print(f" Rank {i+1}: B_alloc = {results[i]['B_alloc']}, Reward = {results[i]
['reward']:.2f}")
```

Figura 28. Representación de los resultados de la prueba de *reward.*

La salida obtenida se muestra a continuación en la Figura 29. En ella podemos observar que la configuración con el mejor *reward* en este caso concreto de carga simétrica y con un SLA igual para todas las ONTs de cada tipo, es aquella en la que a todas las ONTs flexibles se les asigna el mismo ancho de banda. Es decir, el *reward* tiene el comportamiento deseado.

```
Total number of combinations to check: 59049
Exploration complete.
Highest reward found: 6.3
Corresponding B_alloc: [600. 600. 600. 600. 600. 600. 640. 640. 640. 640. 640. 640. 640. 640. 640. 640.]
Top 5 combinations:
Rank 1: B_alloc = [600. 600. 600. 600. 600. 600. 640. 640. 640. 640. 640. 640. 640. 640. 640. 640.], Reward = 6.30
Rank 2: B_alloc = [600. 600. 600. 600. 600. 600.   0. 640. 640. 640. 640. 640. 640. 640. 640. 640.], Reward = 5.94
Rank 3: B_alloc = [600. 600. 600. 600. 600. 600. 640.   0. 640. 640. 640. 640. 640. 640. 640. 640.], Reward = 5.94
Rank 4: B_alloc = [600. 600. 600. 600. 600. 600. 640. 640.   0. 640. 640. 640. 640. 640. 640. 640.], Reward = 5.94
Rank 5: B_alloc = [600. 600. 600. 600. 600. 600. 640. 640. 640.   0. 640. 640. 640. 640. 640. 640.], Reward = 5.94
```

Figura 29. Salida de la ejecución de la prueba del *reward.*

3.6.2.2.4 Función step

Un *step* (o paso) corresponde a la unidad básica de tiempo en la que nuestro agente percibe el estado en el que se encuentra, selecciona una acción, obtiene una recompensa y avanza hacia un nuevo estado. Se trata de la interacción puntual entre el agente y su entorno: en cada paso se toma una decisión y se actualiza la situación del sistema. Gracias a esta dinámica, el agente puede aprender progresivamente a elegir acciones que le permitan maximizar sus recompensas a lo largo del tiempo.

Al inicio de la función *step*, se inicializa el cronómetro mediante *start_time = time.time(),* lo que permite registrar el instante en que comienza la ejecución del *step* y calcular la duración real de dicho ciclo. A continuación, se guarda una copia del estado anterior de las colas de las ONTs (*self.B_demand*) en la variable *self.prev_demand.* Esta copia es fundamental porque servirá para comparar el estado anterior con el nuevo (Figura 30).

```
def step (self, action):
    start_time = time.time()
    self.prev_demand = np.copy(self.B_demand)
```

Figura 30. Declaración en la función *step* de *start_time* y de *self.prev_demand.*

En concreto, dentro del *step,* se llama en primer lugar a la función *simular_trafico()* del *script traficoparetopython.py* para simular el nuevo tráfico a una cierta carga (Figura 31). Este *script* es explicado en mayor profundidad en el Apartado 3.6. donde se explica en profundidad la integración de una fuente de tráfico de Pareto real, que será la que genere el tráfico en nuestro sistema.

```
self.trafico_entrada_por_ciclo,self.onts = simular_trafico(self.onts)
self.trafico_entrada = self.trafico_entrada_por_ciclo
```

Figura 31. Llamada a la función *simular_trafico* del script *traficoparetopython.py* en la función *step*.

A continuación, se recibe la acción del agente, es decir, el ancho de banda asignado (*self.B_alloc*). Esta acción se normaliza de forma que nunca puede exceder el ancho de banda máximo (*B_max*) de cada ONT. Posteriormente, se calcula el ancho de banda total asignado a las ONTs mediante la suma de las acciones normalizadas del agente (*total = np.sum(self.B_alloc)*) y se verifica si este supera la capacidad disponible de la OLT (*self.B_available*). En caso de que sí se supere, se guarda el exceso (*self.b_excess*) para su uso posterior en el *reward* y se ajusta proporcionalmente la asignación de ancho de banda (*self.B_alloc*) para no sobrepasar el límite físico del sistema, multiplicando la acción *B_alloc* por un factor igual a *self.B_available/total* (Figura 32).

```
self.B_alloc = np.clip(action, 0, 1) * self.B_max
total = np.sum(self.B_alloc)
if total > self.B_available:
    self.b_excess = total - self.B_available
    self.B_alloc *= (self.B_available / total)
```

Figura 32. Definición del ancho de banda asignado mediante la acción en la función *step*.

A continuación, se declara un bucle que recorre cada ONT (Figura 33), esto es, se añade el nuevo tráfico entrante (bits) a la cola (*self.B_demand[i] += self.trafico_entrada[i]*), se corrige la asignación de ancho de banda si esta excede la demanda real de la ONT, y posteriormente se descuenta el ancho de banda ofrecido de la cola.

```
for i in range(self.num_ont):
            self.B_demand[i] += self.trafico_entrada[i]
            if self.B_alloc[i] > self.B_demand[i]:
                self.B_alloc[i] = self.B_demand[i]
            self.B_demand[i] -= self.B_alloc[i]
```

Figura 33. Actualización del tráfico de las ONTs en la función *step*.

Finalmente, como vemos en la Figura 34, se aplica un *np.clip* para asegurar que el tamaño de cada cola nunca sea negativo ni supere el máximo permitido (*self.max_queue[i]*).

```
self.B_demand[i] = np.clip(self.B_demand[i], 0, self.max_queue[i])
```

Figura 34. Aplicación de la función *clip* para delimitar el tamaño de la cola.

Para calcular la recompensa tras realizar la acción (Figura 35), llamamos a la función *_calculate_reward*, explicada anteriormente.

```
reward = self._calculate_reward ()
```

Figura 35. Llamada a la función *_calculate_reward* dentro del *step*.

El siguiente paso es actualizar el tiempo del episodio, tal y como se muestra en la Figura 36. Dentro de este código se observa lo siguiente. Primero se compara el número de ciclos transcurridos (*self.instantes*) con el número máximo de ciclos permitidos (*self.n_ciclos*). Si ambos coinciden, significa que se ha alcanzado el final del episodio y se marca la variable done como *True*; y en caso contrario, el episodio continúa (*done = False*). Después, se incrementa en uno el contador de instantes. Finalmente, se mide el tiempo real que ha tardado en ejecutarse el paso mediante la resta entre el instante final (*end_time*) y el inicial (*start_time*), lo que permite calcular la duración de cada ciclo (*step_duration*).

```
if self.instantes == self.n_ciclos:
        done = True
    else:
        done = False
    self.instantes += 1
    end_time = time.time()
    step_duration = end_time - start_time
```

Figura 36. Actualización del tiempo del episodio en la función *step*.

Finalmente, se devuelve la observación mediante la llamada a la función *_get_obs* explicada previamente en el Apartado 3.6.3.2.1, la recompensa y otros datos, tal y como se muestra en la Figura 37.

```
info = self._get_info()
return self._get_obs(), reward, done, False, info
```

Figura 37. Datos que devuelve la función *step*.

3.6.3 Ejecución el entorno de simulación

La selección y ejecución del entorno que corresponde de cada escenario se realiza desde el *script* *Escenarios.ipynb*. Con la ejecución de este *script* al usuario se le dará la posibilidad de seleccionar el entorno que desee ejecutar, como se muestra en la Figura 38.

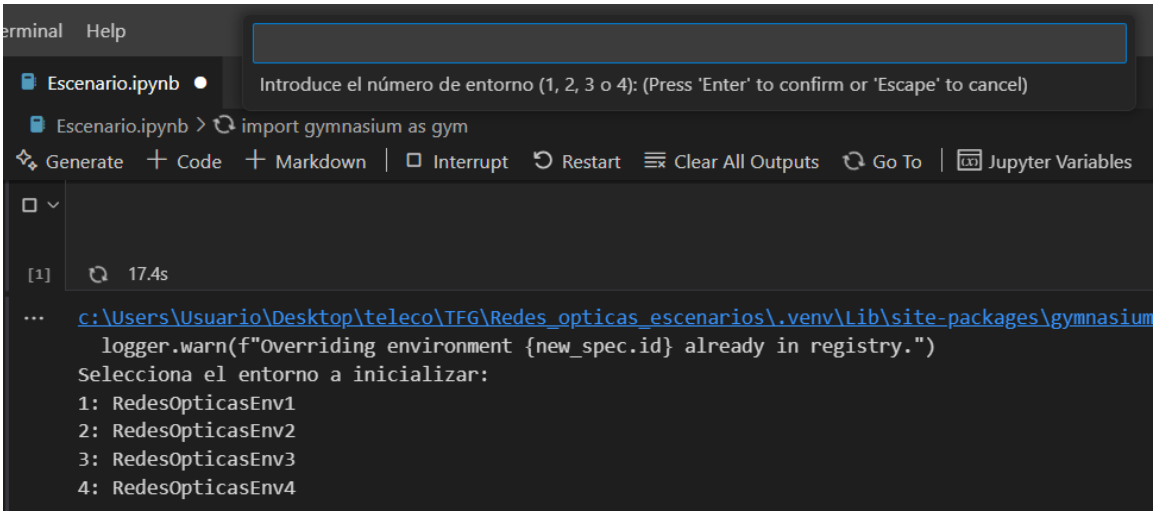


Figura 38. Opciones para el usuario para ejecutar el entorno.

Además, en el caso del escenario 3, se le darán dos opciones, dependiendo de si se quiere simular el caso 1 o 2, como se puede observar en la Figura 39. Una vez seleccionado el escenario, el entrenamiento del modelo se pondrá en marcha.

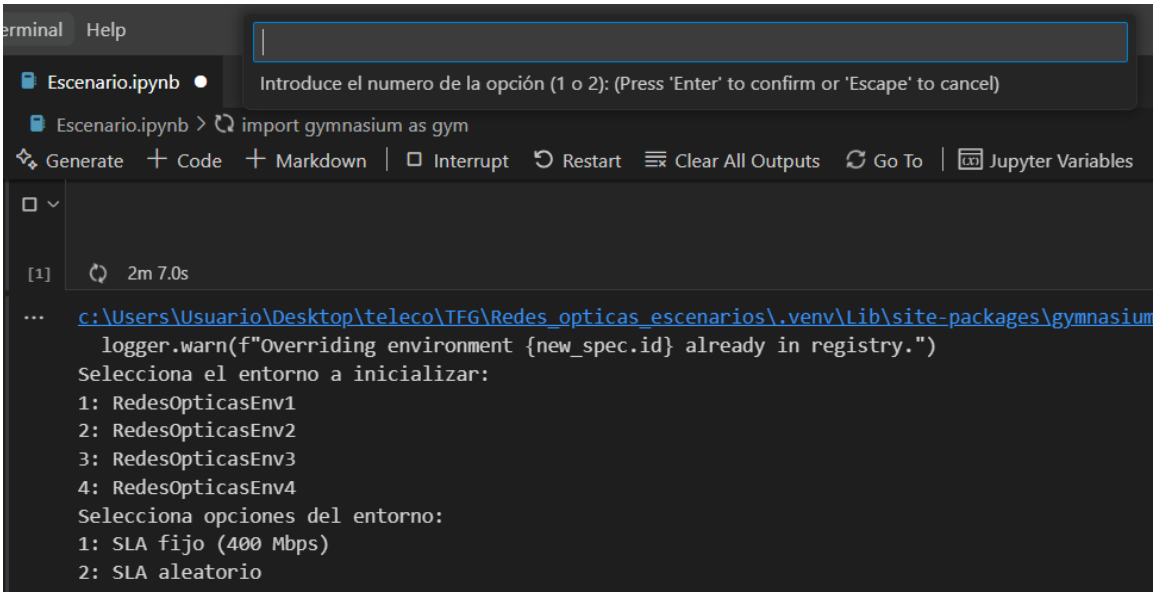


Figura 39. Opciones que el usuario puede elegir dentro del escenario 3

3.6.4 Definición del agente DRL

El agente se define en el script *Escenarios.ipynb* y su comportamiento está determinado por un conjunto de hiperparámetros específicos descritos en la Tabla 4. Estos controlan cómo el agente percibe el entorno, decide las acciones y aprende de la experiencia a lo largo de los episodios de simulación. En este trabajo, el agente se implementa utilizando el algoritmo PPO (*Proximal Policy Optimization*) de la librería *Stable Baselines3* [39], luego estos hiperparámetros se corresponden a los utilizados por dicho algoritmo. En la Tabla 4 se definen los hiperparámetros y se da una descripción y definición de cada uno de ellos.

Hiperparámetros	Descripción
n_steps	Número de pasos recolectados antes de cada actualización del modelo.
batch_size	Tamaño de los mini-lotes usados en cada actualización.
learning_rate	Tasa de aprendizaje del optimizador.
n_epochs	Número de veces que se reutiliza cada lote de datos para actualizar la red.
gamma	Factor de descuento para las recompensas futuras.
gae_lambda	Parámetro de lambda para la estimación de ventaja generalizada (GAE).
clip_range	Rango de recorte para la probabilidad en la política (PPO clipping).
clip_range_vf	Rango de recorte para la función de valor (estabilidad del valor estimado).
max_grad_norm	Límite para la normalización de gradientes.

Tabla 4. Hiperparámetros que definen el agente

Además de los hiperparámetros mencionados en la tabla existen una serie de parámetros importantes definidos en la declaración del modelo:

- *"MlpPolicy":* indica que la política del agente una red neuronal multicapa (MLP), adecuada para entradas vectoriales (no imágenes).
- *vec_env:* es el entorno o vector de entornos donde entrenará el agente.

- *tensorboard_log="./ppo_tensorboard_logs/":* carpeta donde se guardan los logs para *TensorBoard,* para visualizar recompensas, pérdidas y métricas de entrenamiento.
- *device="cpu":* indica dónde se ejecutará el entrenamiento: *cpu* o *cuda* para GPU.

A continuación, se presenta la definición del agente de DRL en el código (Figura 40), mediante la configuración del algoritmo PPO. En esta Figura 40 se observa para cada parámetro del algoritmo el valor inicial escogido que luego pasará a optimizarse.

```
model = PPO(
        "MlpPolicy",
        vec_env,
        n_steps=512,
        batch_size=32,
        learning_rate =1e-3,
        n_epochs=30,
        gamma= 0.95,
        gae_lambda= 0.9,
        tensorboard_log="./ppo_tensorboard_logs/",
        device="cpu"
    )
```

Figura 40. Definición del modelo DRL en el código.

3.7 Modelado e integración de tráfico basado en Pareto

La distribución de Pareto es un modelo estadístico que se suele emplear a la hora de describir fenómenos en los que una pequeña parte de un conjunto genera una gran parte de los efectos. Este fenómeno fue estudiado por primera vez por el economista italiano Vilfredo Pareto que estudiando las distribuciones de riqueza en Italia identificó que el 80 % de la riqueza estaba en manos del 20 % de la población, posteriormente esta distribución sería usada ampliamente en diversos campos como, por ejemplo, la economía y el análisis de redes [40]. Esta distribución tiene una forma particular, como puede observarse en la Figura 41. La mayoría de los valores son pequeños, pero hay unos pocos valores muy grandes que influyen mucho en la media. Esta propiedad es fundamental en aquellos casos

en los que los eventos extremos tienen alta probabilidad relativa, como por el ejemplo el tráfico de datos en la red.

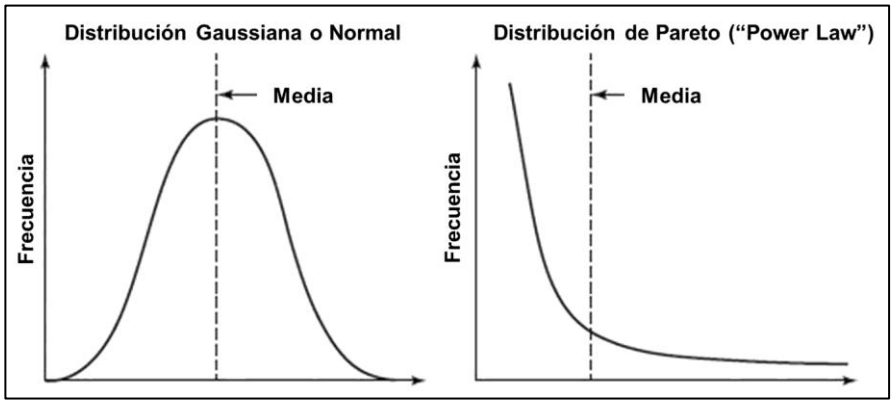


Figura 41. Comparación de la Distribución de Pareto con una Distribución Gaussiana (Extraído de [41]).

La distribución de Pareto es adecuada para representar fuentes muy variables y que transmiten en ráfagas (*bursts*), las cuales son difíciles de modelar correctamente con otro tipo de distribuciones como pueden ser la exponencial o de Poisson.

En el generador de tráfico que vamos a integrar en nuestro simulador, se genera tráfico auto-similar (*seft-similar)* con fuentes de Pareto. Este tipo de tráfico consiste en generar de paquetes a ráfagas siguiendo un patrón [42]. Para ello se agregan una serie de *sub-streams*, que alternan tiempos de *ON* (tiempo en el que se transmite) y *OFF* (tiempo en el que no se transmite) con distribuciones de Pareto independientes[12]. En la Figura 42 se muestra un esquema de este proceso.

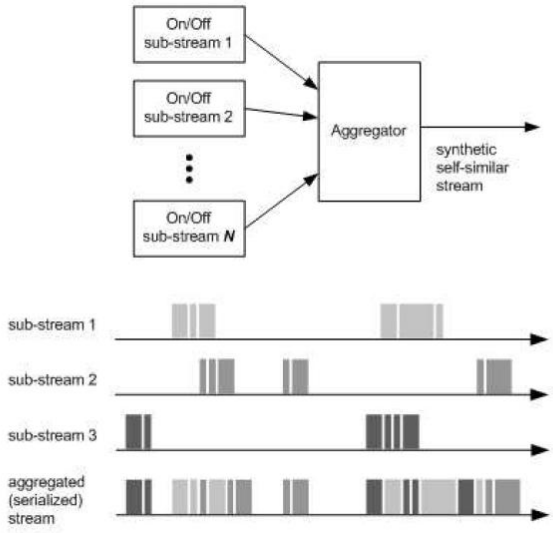


Figura 42. Generación de tráfico *self-similar* (Extraído de [12]).

Este ha sido uno de los cambios esenciales realizados en el simulador desarrollado por David Pérez ya que, mientras que en este primero solo existía una única fuente de Pareto, en este caso tenemos varias fuentes de Pareto multiplexadas, en concreto 32, para generar el tráfico auto-similar de un modo más realista.

A continuación, se explica de forma detallada cómo se ha integrado dicho simulador de fuentes de tráfico de Pareto al simulador de nuestro TFG, explicando y describiendo las clases y métodos implementados. Entre las modificaciones que fueron necesarias para adaptar el *script traficoparetopython.py* al entorno de trabajo, una de las más importantes fue la creación de la clase ONT, que permitió que el sistema diseñado fuese escalable, ya que en un principio el script solo simulaba el tráfico de una única fuente. A continuación, se describen las distintas funciones que constituyen esta clase.

El método *__init__* es el constructor de la clase e inicializa todos los atributos de la ONT, tal y como se muestra en la Figura 43. Entre estos se incluyen el identificador de cada ONT, la tasa de transmisión (*self.BitRate*), el tamaño de los paquetes (*self.PacketSize*), la carga de tráfico (*self.Load*) y el número de fuentes de tráfico (*self.Streams*).

```
class ONT:
    def __init__(self, id, bit_rate, packet_size, load, streams):
        self.ID = id
        self.BitRate = bit_rate
        self.PacketSize = packet_size
        self.Load = load
        self.Streams = streams
        self.Generator = Generator()
        self.Cola = []
        self.MaxColaSize = 800e8
        self.CurrentColaSize = 0
        self.DiscardedPackets = 0
        self.Timestamps = []
        self.LastProcessedTime = 0.0

        for src in range(self.Streams):
            self.Generator.AddSource(SourcePareto(src, 0, self.PacketSize, 0, self.Load /
self.Streams, 1.4, 1.2))
```

Figura 43. Método *_init* dentro de la clase ONT.

El método *GetNextPacket()* (Figura 44) obtiene el siguiente paquete generado por la ONT usando la clase *Generator* otra de las clases de *traficoparetopyhon*. Una vez generado, el paquete se añade a la cola mediante la función *AddToCola*. Además, se registra el tiempo de llegada del paquete en la lista *Timestamps*.

```
def GetNextPacket(self):
        packet = self.Generator.GetNextPacket()
        self.AddToCola(packet)
        self.Timestamps.append(self.GetCurrentTime())
        return packet
```

Figura 44. Método *GetNextPacket* dentro de la clase ONT.

El método *AddToCola()* (Figura 45) se encarga de gestionar la cola de la ONT. Si la suma del tamaño actual de la cola y del paquete que llega no supera el tamaño máximo de la cola (*MaxColaSize*), el paquete se añade a la cola y se actualiza *CurrentColaSize*. En caso contrario, el paquete no se añade y se incrementa el contador *DiscardedPackets*, contabilizando los paquetes que han sido descartados al saturarse la cola.

```
def AddToCola(self, packet):
        if self.CurrentColaSize + packet.PacketSize ⩽ self.MaxColaSize:
            self.Cola.append(packet)
            self.CurrentColaSize += packet.PacketSize
        else:
            self.DiscardedPackets += 1
```

Figura 45. Método *AddToCola* dentro de la clase ONT.

El método *GetColaSize()* (Figura 46) devuelve simplemente el número de paquetes que hay actualmente en la cola de la ONT. Por otro lado, El método *GetDiscardedPackets* devuelve el número total de paquetes que han sido descartados por la ONT debido a que exceden el tamaño de su cola.

```
def GetColaSize(self):
    return len(self.Cola)

def GetDiscardedPackets(self):
    return self.DiscardedPackets
```

Figura 46. Métodos *GetColaSize* y *GetDiscardedPackets* dentro de la clase ONT.

La función *GetCurrentTime()* (Figura 47) obtiene el tiempo actual del generador de paquetes (*Generator*), mientras que *GetByteStamp* devuelve la marca de bytes actual, información que se utiliza para calcular intervalos entre paquetes y analizar la carga de tráfico de manera precisa.

```
def GetCurrentTime(self):
      return self.Generator.GetTime()

def GetByteStamp(self):
      return self.Generator.GetByteStamp()
```

Figura 47. Método *GetCurrentTime* y *GetByteStamp* dentro de la clase ONT.

El método *GetTotalLoad()* (Figura 48) calcula la carga total de la ONT como el número de bits generados dividido por el producto del tiempo transcurrido y la tasa de transmisión (*self.BitRate*). Además, el método *Reset()* (Figura 49) reinicia el estado de la ONT.

```
def GetTotalLoad(self):
elapsed_time = self.GetCurrentTime()
if elapsed_time > 0:
    return (self.Generator.GetBytes() * 8) / (elapsed_time * self.BitRate)
return 0.0
```

Figura 48. Método *GetTotalLoad* dentro de la clase ONT.

```
def Reset(self):
    self.Generator.Reset()
    self.Cola.clear()
    self.Timestamps.clear()
```

Figura 49. Método *Reset* dentro de la clase ONT.

Finalmente, el método *__str__* (Figura 50) devuelve una en cadena de texto con información importante como el número total de paquetes generados, los paquetes descartados, los bytes totales transmitidos y el tamaño de la cola.

```
        def __str__(self):
            return f"ONT {self.ID}, Paquetes totales={self.Generator.GetPackets()},Paquetes
descartados={self.GetDiscardedPackets()}, Bytes totales={self.Generator.GetBytes()}, Cola=
{len(self.Cola)} paquetes"
```

Figura 50. Método *_str_* dentro de la clase ONT.

Como se observará en el siguiente apartado (Apartado 3.7.2) las únicas funciones que realmente vamos a emplear en este simulador referentes a la clase ONT son: *GetNextPacket()*, *GetCurrentTime()* y *GetByteStamp()*, mientras que el resto de las funciones, como las referentes a el tamaño máximo de la cola, se han externalizado. Por otro lado, se definió la función *simular_trafico()*, en la que se calcula el tráfico y la carga de cada una de las ONTs. Esta parte del código, es decir la Figura 51, hace referencia a la clase ONT, que también tuvo que ser creada para el desarrollo de este proyecto.

```
def simular_trafico(onts=None):
        if onts is None:
        onts = [ONT(id=i, bit_rate=ONT_bit_rate, packet_size=packet_size, load=load_dmb,
streams=streams) for i in range(num_ont)]
    suma_tam_packets = [0] * 16
        total_loads = [0] * 16
    trafico_entrada_por_ciclo = [0] * 16
```

Figura 51. Función *simular_tráfico* en el *script traficoparetopython.py.*

En el bucle siguiente, descrito en la Figura 52 se calcula la carga generada por cada ONT (*total_loads*) y el tráfico de entrada que recibe cada una durante un ciclo de simulación (*trafico_entrada_por_ciclo*). Dentro de cada iteración, se simula la llegada de paquetes hasta alcanzar el límite de tiempo (*time_limit*). En cada paso se obtiene el siguiente paquete disponible (*GetNextPacket()*), se registra el tiempo actual (*GetCurrentTime()*) y se calcula el intervalo entre paquetes en función del tamaño del paquete y la tasa de transmisión de la ONT (*ONT_bit_rate*). Estos intervalos se acumulan para determinar la duración total de

observación de cada ONT, mientras que se suman los tamaños de los paquetes recibidos (*suma_tam_packets*). Finalmente, se calcula la carga total de la ONT como la relación entre los bits transmitidos y el producto del tiempo total por la tasa de transmisión y se almacena el tráfico de entrada por ciclo en bits. Tanto *trafico_entrada_por_ciclo()* como la lista de ONTs actualizada se devuelven al entorno para ser utilizadas posteriormente en la asignación de ancho de banda y el cálculo de recompensas.

```
          for ont in onts:
        current_time = 0.0
        while current_time ⩽ time_limit:
                trc = ont.GetNextPacket()
                tiempo = ont.GetCurrentTime()
                byte_stamp = ont.GetByteStamp()
                intervalo_actual = byte_stamp - tiempo
                interarrivaltime = intervalo_actual * 8 / ONT_bit_rate
                suma_tam_packets[ont.ID] += trc.PacketSize
                current_time += interarrivaltime            total_loads[ont.ID] =
(suma_tam_packets[ont.ID] * 8) /(current_time * ONT_bit_rate)
            trafico_entrada_por_ciclo[ont.ID] = suma_tam_packets[ont.ID]*8
    return trafico_entrada_por_ciclo, onts
```

Figura 52. Cálculo de *total_load* y *trafico_entrada_por_ciclo* en la función *simular_tráfico* en el *script traficoparetopython.py.*

3.8 Conclusiones

Este capítulo incluye el contexto teórico del trabajo y la implementación realizada para desarrollar un simulador de asignación de ancho de banda en redes 10G-EPON mediante DRL. En primer lugar, se describieron los fundamentos de las redes PON y su evolución hasta el estándar 10G-EPON (Apartados de 3.2.1 a 3.2.3). Posteriormente se presentó el algoritmo de asignación del simulador (basado en IPACT) (Apartado 3.3), así como la definición de los cuatro escenarios (Apartado 3.4), que van desde configuraciones con tráfico simétrico hasta un caso con ONTs garantizadas y flexibles. Además, se profundizó en los conceptos de DRL (Apartado 3.5), evaluando diferentes algoritmos y justificando la elección del modelo PPO. Este contexto permitió explicar la configuración del simulador (Apartado 3.6), destacando el papel de la función de recompensa. Además, se explicaron otras mejoras como el nuevo generador de tráfico de Pareto (Apartado 3.7).

4

Simulación y Validación de del agente DRL en redes 10G-PON

4.1 Introducción

Este capítulo está dedicado al estudio de los distintos escenarios implementados en el simulador de redes 10G-EPON mediante un agente DRL.

Primero se dará un breve resumen de los cambios realizados en el entorno de cada uno de los escenarios para que las simulaciones se pudiesen llevar a cabo de forma adecuada y se procederá a explicar cómo se ha realizado la optimización mediante *Optuna*, detallando las distintas funciones que conforman el *script*.

Después, se profundizará en los resultados obtenidos en las distintas simulaciones, analizando las gráficas que muestran el tráfico de las distintas ONTs y sus colas. También se analizará la curva de *reward* antes y después de la optimización de hiperparámetros con *Optuna*. Además, se proporcionarán gráficas referentes al índice de equidad y a la asignación de ancho de banda total asignado en el caso del escenario 4.

4.2 Configuración de los escenarios de simulación

4.2.1 Escenario 1: ONTs con SLAs garantizados y tráfico simétrico

En este primer escenario, todas las ONTs cuentan con un mismo ancho de banda garantizado (SLAs), y el tráfico que generan es simétrico, es decir, la carga promedio es en media equivalente para todas ellas. Este primer escenario representa un caso ideal, en el que las condiciones de tráfico se mantienen estables y homogéneas.

Debido a que este escenario funciona como base para el resto, el código que conforma su entorno ya ha sido explicado anteriormente.

4.2.2 Escenario 2: ONTs con SLAs garantizados y tráfico asimétrico

Este segundo escenario es más complejo. Los SLAs siguen siendo los mismos para todas las ONTs, pero el tráfico es asimétrico, es decir, el tráfico se distribuye de forma asimétrica. En otras palabras, la carga promedio varía entre cada ONT, lo que refleja condiciones más cercanas a un escenario real. Para desarrollar este escenario tan solo fue necesario modificar el valor de la carga (*load*) definido en el *script traficoparetopython.py*. Este parámetro pasó de ser una constante a un número aleatorio que variaba con cada ONT, tal y como se observa en la Figura 53.

```
random_loads = [random.choice([i / 10 for i in range (1, 11)]) for_ in range(num_ont)]
```

Figura 53. Declaración de vector de cargas aleatorias para las ONT del escenario 2.

Para que el cambió fuese efectivo también fue necesario crear la llamada a la clase ONT dentro de la función *simular_trafico*, según la Figura 54.

```
    if onts is None:
        onts = [ONT (id=i, bit_rate=ONU_bit_rate, packet_size=packet_size, load=random_loads[i],
streams=streams) for i in range(num_ont)]
```

Figura 54. Llamada a la clase ONT dentro de la función *simular_trafico*.

4.2.3 Escenario 3: ONTs con SLAs garantizados y dinámicos

En este escenario, todas las ONTs cuentan con SLAs garantizados, pero el ancho de banda garantizado para el SLA cambiaba de forma dinámica a lo largo del tiempo. Es decir, el ancho de banda máximo para cada ONT puede variar en un momento dado, porque el proveedor de servicios modifique las condiciones del SLA contratado en tiempo real en un momento dado. Este entorno representa una situación más próxima a la realidad, en la que los patrones de tráfico pueden variar y las condiciones de los parámetros de calidad de servicio ofrecidos por los proveedores (SLAs) pueden modificarse en tiempo real. En consecuencia, la red EPON y los algoritmos DBA implementados debe adaptarse dinámicamente para mantener los niveles de calidad de servicio establecidos. Dentro de este escenario se consideran dos variantes. En la primera, el SLA cambia en un instante

aleatorio hacia un valor fijo, común para todas las ONTs; mientras que, en la segunda, el SLA asignado a cada ONT se determina de forma aleatoria en un instante concreto. Este último comportamiento es un caso extremo que nos ayudará a observar cómo opera el modelo en una situación extrema.

Para el desarrollo de este escenario fue necesario realizar una serie de modificaciones en el entorno que se muestran en las Figuras 55 y 56. Primero, se solicita al usuario elegir cómo se comportará el de los ONT. Al usuario se le muestran dos opciones: *1: SLA fijo (400 Mbps)* y *2: SLA aleatorio*. Este debe introducir el número 1 o 2 dependiendo del escenario que quiera ejecutar. Para ambas opciones, se definen los instantes en los que cada ONT cambia su SLA. Esto se hace generando el array *instantes_cambio*. Estos valores se generan de forma aleatoria dentro de un rango. Finalmente, para la opción 2, se define *self.B_guaranteed_change*, que también se genera aleatoriamente para cada ONT. Representa el nuevo valor de ancho de banda garantizado que tendrá cada ONT en el momento en que cambia.

```
  def __init__ (self, render_mode=None, seed=0, num_ont=16, TxRate=10e9, B_guaranteed=[],
n_ciclos=200):


        print ("Selecciona opciones del entorno:")
        print ("1: SLA fijo (400 Mbps)")
        print ("2: SLA aleatorio")
        self.SLA_seleccionado= input ("Introduce el numero de la opción (1 o 2): ")
        self.B_guaranteed_change = np.array([random.choice([i / 10 for i in range(2000, 8000)])
for _ in range(self.num_ont)])
        self.instantes_cambio = np.array([random.choice([i / 10 for i in range(500, 4000)]) for _
in range(self.num_ont)])
```

Figura 55. Modificaciones en la función *_init* del entorno 4.

Además, en el *step* también fue necesario diferenciar estos dos casos, así como modificar el valor de *B_guaranteed* para cada uno de ellos.

El bucle recorre cada ONT y en cada iteración se compara el instante actual de la simulación con el instante de cambio específico de cada ONT. Si se ha elegido la opción de SLA aleatorio y el instante actual ha alcanzado o superado el instante de cambio, el ancho de banda garantizado de ese ONT se actualiza al valor correspondiente de

B_guaranteed_change[i]. Si se ha seleccionado la opción de SLA fijo y también se ha alcanzado el instante de cambio, el ancho de banda garantizado se fija en 400 Mbps (este valor es configurable). Finalmente, se actualiza *self.B_max* multiplicando el *array B_guaranteed* por *self.temp_ciclo*, lo que permite ajustar los valores de ancho de banda en función del tiempo o del ciclo actual.

```
for i in range(self.num_ont):
    if self.instantes ⩾ self.instantes_cambio[i] and self.SLA_seleccionado == "2":
        self.B_guaranteed[i] = self.B_guaranteed_change[i] * 1e6
    elif self.instantes ⩾ self.instantes_cambio[i] and self.SLA_seleccionado == "1":
        self.B_guaranteed[i] = 400e6
    else:
        self.B_guaranteed[i] = 600e6
    self.B_max = np.array(self.B_guaranteed) * self.temp_ciclo
```

Figura 56. Modificaciones en la función *step* del entorno 4.

4.2.4 Escenario 4: ONTs con SLAs garantizados y no garantizados

En este último escenario se distinguen dos tipos de ONTs: aquellas asociadas a operadores tradicionales, con SLAs garantizados, y las pertenecientes a operadores virtuales o competidores con precios más bajos, que disponen de SLAs flexibles o no garantizados. De este modo, algunas ONTs cuentan con recursos mínimos asegurados, mientras que otras dependen exclusivamente de la disponibilidad de la red. Este contexto refleja una situación mucho más compleja y cercana a la realidad del mercado, en la que el sistema debe aprender a priorizar a las ONTs con SLAs garantizados, sin descuidar al mismo tiempo la demanda proveniente de las ONTs con condiciones más flexibles.

En la Figura 57 se puede observar cómo se determinan los anchos de banda máximos y mínimos para cada tipo de ONT. Para las ONTs garantizadas se establece un máximo de 600 Mbps, mientras que para las flexibles se asigna un máximo de 800 Mbps (aunque estos parámetros son configurables). Además, se definen dos variables específicas para los ONT flexibles (de tipo 2): *B_max_ONT2* indica su ancho de banda máximo garantizado (800 Mbps) y *B_min_ONT2* representa un ancho de banda mínimo, en este caso definido como el 60% de su máximo, qué si deberá cumplirse. Todos estos valores podrán ser configurados.

```
        for i in range(self.num_ont):
            if i < self.num_guaranteed_ont:
                self.B_max[i] = 600e6 * self.temp_ciclo
            else:
                self.B_max[i] = 800e6 * self.temp_ciclo

        self.B_guaranteed_max = 800e6
        self.B_max_ONT2 = self.B_guaranteed_max * self.temp_ciclo
        self.B_min_ONT2 = self.B_guaranteed_max * self.temp_ciclo * 0.6
```

Figura 57. Modificaciones en la función *step* del entorno 4.

Un cambio importante en el escenario 4 fue discretizar el espacio de acciones. El espacio de acción (*action space*) es el conjunto de todas las acciones posibles que un agente puede realizar en un entorno[2]. En los tres primeros escenarios se empleó un espacio de acciones continuo mediante la función *spaces.Box* (Figura 58). De esta forma el agente puede seleccionar directamente valores continuos dentro de un intervalo.

```
self.action_space = spaces.Box(low=0, high=1, shape=(self.num_ont,), dtype=np.float64)
```

Figura 58. Definición del espacio de acciones de los primeros 3 escenarios.

Sin embargo, la complejidad del escenario 4 llevó a definir un espacio de acciones discretas, tal y como se meustra en la Figura 59. En concreto, estas acciones definen el ancho de banda asignado a las ONTs por el agente (*B_alloc*). Al emplear acciones discretas, es más facil para el modelo converger en el valor óptimo de su *reward.* El espacio de acciones discreto se definió mediante la función *spaces.MultiDiscrete*.

En este caso el agente elige entre un conjunto finito de opciones enteras, en concreto 181 posibles entre 0 y 900 en saltos de 5 (0,5,10,15,...,895,900). Estos valores representan el ancho de banda que el agente puede asignar (*B_alloc*). Así, acotamos la decisión del agente, que puede elegir entre menos cantidad de valores a la hora de determinar este ancho de banda. Así, pues primero definimos el espacio de forma que las acciones puedan tomar valores de 0 a 180, tal y como se observa en la Figura 59.

```
self.action_space = spaces.MultiDiscrete([181] * self.num_ont)
```

Figura 59. Definición del espacio de acciones discreto del escenario 4.

Después normalizamos este valor (Figura 60) de forma que, multiplicamos el valor por 5 (180 * 5 = 900) y después lo dividimos entre 900 para que esté normalizada entre 0 y 1 (al igual que la acción continua de los anteriores escenarios). Esto nos asegura que las acciones que pueda tomar el agente estén dentro de un rango de valores razonable y no demasiado variable, lo que facilita el entrenamiento. Además, se han realizado los cambios pertinentes en la función *calculate_reward*.

```
action = np.array(action)
action_real = action * 5 / 900
self.B_alloc = np.clip(action_real, self.epsilon, 1) * self.B_max
```

Figura 60. Normalización del espacio de acciones discreto del escenario 4.

4.3 Configuración de Optuna para nuestros escenarios de trabajo

Como se ha mencionado anteriormente, *Optuna* es una librería de *Python* cuyo uso fundamental es optimizar automáticamente los hiperparámetros, de forma que mejore el rendimiento de los modelos de *Machine Learning* sobre los que se aplica[4]. A continuación, se presentan las partes más importantes del código de configuración de la herramienta en nuestro sistema, junto con una breve explicación de cada una.

Tal y como se observa en el código de la Figura 61, la función *make_env* inicializa el entorno, en este caso, *RedesOpticasEnv*, el cual modela la red de acceso óptica 10G-GPON y con parámetros ajustables, tales como: número de ONTs (*num_ont*), la tasa de transmisión (*TxRate*), el ancho de banda garantizado por ONT (*B_guaranteed*) y el número de ciclos de simulación (*n_ciclos*). Esto garantiza la reproducibilidad y la correcta instanciación del entorno durante las múltiples ejecuciones requeridas por *Optuna*.

```
    def make_env (num_ont, TxRate=10e9, B_guaranteed=[], n_ciclos=200, rank=0, seed=0):
        def _init():
            env = RedesOpticasEnv(render_mode=None, seed=seed, num_ont=num_ont, TxRate=TxRate,
B_guaranteed=B_guaranteed,n_ciclos=n_ciclos)
            return env
        return _init
```

Figura 61. Inicialización del entorno en el *script* de *Optuna*.

La función *objective* es una función fundamental en el proceso de optimización. En ella se define un diccionario en el que se establecen los valores de los hiperparámetros que se irán probando a lo largo de la optimización. En nuestro caso hemos empleado los valores que se muestran en la Figura 62. La Tabla 5 muestra dichos hiperparámetros, su efecto en las simulaciones, los calores que toman y la forma en la que se muestrean [39]. Estos se corresponden con los hiperparámetros configurables del algoritmo PPO de nuestro trabajo.

		Efecto sobre el entrenamiento al…	
Hiperparámetros	**Valores**	**Aumentar**	**Disminuir**
learning_rate	1e-5 – 1e-3	Entrenamiento más rápido pero inestable.	Entrenamiento más estable pero lento.
n_steps	256, 512, 1024, 2048	Mejores estimaciones, pero mayor costo computacional	Entrenamiento más rápido con peores estimaciones
batch_size	32, 64, 128, 256	Menos varianza y gradientes más estables.	Más exploración y ruido.
n_epochs	3 – 30	Se repasan más los datos (riesgo de que se produzca overfitting)	Entrenamiento más rápido pero peor aprovechamiento de los datos.
gamma	0.9 – 0.9999	Se valoran más las recompensas futuras.	Se valoran más las recompensas inmediatas.
gae_lambda	0.8 – 0.99	Menor varianza y más sesgo.	Mayor varianza y menos sesgo.
clip_range	0.1 – 0.4	Cambios más grandes en la política.	Cambios más pequeños en la política.
clip_range_vf	0.1 – 0.4	Función de valor con menos restricciones. Menos estabilidad.	Más restricciones en la función de valor, más estabilidad.
max_grad_norm	0.3 – 5.0	Gradientes más grandes.	Gradientes más pequeños.

Tabla 5. Hiperparámetros que se modifican a lo largo de la optimización

```
def objective(trial):
    hyperparams = {
     'learning_rate':trial.suggest_float('learning_rate',1e-5,1e-3),
     'n_steps':trial.suggest_categorical('n_steps', [256, 512, 1024]),
     'batch_size':trial.suggest_categorical('batch_size', [128, 256]),
     'n_epochs':trial.suggest_int('n_epochs', 3, 20),
     'gamma':trial.suggest_float('gamma', 0.9, 0.9999),
     'gae_lambda':trial.suggest_float('gae_lambda', 0.8, 0.99),
     'clip_range':trial.suggest_float('clip_range', 0.1, 0.4),
     'clip_range_vf':trial.suggest_float('clip_range_vf', 0.1, 0.4),
     'max_grad_norm':trial.suggest_float('max_grad_norm', 0.3, 5.0)  }
```

Figura 62. Hiperparámetros descritos en la función *objective* de *Optuna*.

Lo siguiente que se define en la función *objective* es una condición esencial entre los hiperparámetros del algoritmo PPO: si el tamaño del parámetro *batch_size* es mayor que el número de *steps, n_steps,* se igualan al mismo valor. Esto es porque el algoritmo PPO no puede procesar más muestras de las que genera en cada vez que se itera.

Una vez que se cumple esta relación, se fijan los parámetros que definen el entorno 10G-EPON para cada escenario. Por ejemplo, para este caso de la Figura 63: *num_ont* de 16 ONTs, *TxRate* de 10 Gbps, 400 ciclos y se especifica que las primeros 6 ONTs tendrán un ancho de banda garantizado de 600 Mbps, mientras que los demás serán flexibles. Estos datos se emplean para crear el entorno de evaluación (*eval_env*) utilizando la función *make_env* antes mencionada (Figura 63). Finalmente, se construye el modelo de PPO empleando todos los parámetros mencionados en esta función *objective*.

```
    if hyperparams['batch_size'] > hyperparams['n_steps']:
            hyperparams['batch_size'] = hyperparams['n_steps']


        num_ont = 16
        TxRate = 10e9
        n_ciclos = 400
        B_guaranteed = [600e6 if i < 6 else 0 for i in range(num_ont)]


        eval_env= make_env(num_ont, TxRate, B_guaranteed, n_ciclos, seed=100)()
        model = PPO('MlpPolicy', eval_env, verbose=0, **hyperparams)
```

Figura 63. Construcción del modelo PPO en el *script* de *Optuna*.

Posteriormente, se emplea un *callback* de evaluación *EvalCallback* (Figura 64) para monitorizar la recompensa media que se va alcanzando durante el entrenamiento. Gracias a esta función, recibiremos actualizaciones mientras se lleve a cabo el estudio. Para ello se definen una serie de parámetros:

- *best_model_save_path:* guarda el mejor modelo en *./best_model/*.
- *logs_path:* guarda logs en *./logs/*.
- *eval_freq:* evalúa cada 2048 *steps*.
- *deterministic=True:* esta opción no añade ruido e incluye menos exploración.
- *n_eval_episodes:* número de episodios con los que se va evaluando el rendimiento.

```
eval_callback = EvalCallback(
    eval_env,
    best_model_save_path="./best_model/",
    log_path="./logs/",
    eval_freq=2048,  # Evaluate every 2048 steps
    deterministic=True,
    render=False,
    n_eval_episodes=10,
)
```

Figura 64. Declaración de función *EvalCallBack* de *Optuna*.

Como se observa en el código siguiente de la Figura 65, se usa un total de 100000 *timesteps* en cada ensayo (*trial*); y en caso de fallo de entrenamiento, la función devuelve el peor resultado posible, garantizando la estabilidad del estudio.

```
    try:
        model.learn(total_timesteps=100000, callback=eval_callback)
        if eval_callback.best_mean_reward is not None:
            mean_reward = eval_callback.best_mean_reward
        else:
            mean_reward, _ = evaluate_policy(model, eval_env, n_eval_episodes=10)
    except Exception as e:
        print(f"Trial failed: {str(e)}")
        mean_reward = -float('inf')
    trial.report(mean_reward, step=1)
    if trial.should_prune():
        raise optuna.TrialPruned()
    return mean_reward
```

Figura 65. Caso de fallo en la ejecución del *script* de *Optuna*.

A continuación, la función *optimize_ppo_hyperparams crea* un estudio de *Optuna* (*study*) (Figura 68) con las siguientes características [4]:

- Muestreo mediante *TPESampler* (*Tree-structured Parzen Estimator Sampler*) (Figura 66), empleado en espacios de búsqueda continuos y categóricos. El *sampler* es el encargado de decidir que valores de hiperparámetros se usan en cada ensayo.

```
def optimize_ppo_hyperparams(env, n_trials=50, timeout=None, n_jobs=25):
        sampler = TPESampler(seed=42)
```

Figura 66. *Sampler* dentro de la función *optimize_ppo_hyperparams*.

- *Pruner* (poda) (Figura 67) mediante *Median Pruner,* que elimina ensayos poco prometedores a partir de la mediana de resultados obtenidos en ensayos previos. El *pruner* se emplea para detener ensayos que no tienen buenos resultados.

```
        pruner = MedianPruner(n_startup_trials=10, n_warmup_steps=5)
```

Figura 67. *Pruner* dentro de la función *optimize_ppo_hyperparams*.

- Dirección de optimización destinada maximizar la recompensa media. Así *Optuna* buscará la combinación de hiperparámetros que de la mayor recompensa posible.
- Almacenamiento en base de datos *SQLite*, lo cual permite pausar y reanudar el estudio sin pérdida de información.

```
    study = optuna.create_study(
            direction='maximize',
            sampler=sampler,
            pruner=pruner,
            storage='sqlite:///ppo_optuna_study.db.sqlite'
     )
```

Figura 68. Creación del estudio de *Optuna* con los parámetros correspondientes.

Al final del código se ejecuta la optimización de hiperparámetros (Figura 69). La función *objective* se va evaluando con configuraciones diferentes del PPO. Para controlar la ejecución, se definen estos parámetros: *n_trials* indica cuántos ensayos realizar, *timeout* fija el tiempo máximo del estudio*, n_jobs* establece el número de procesos que se pueden ejecutar en paralelo y *show_progress_bar=True* que activa una barra de progreso.

```
    study.optimize(
                objective,
                n_trials=n_trials,
                timeout=timeout,
                n_jobs=n_jobs,
                show_progress_bar=True
            )
        if __name__ == "__main__":
            study = optimize_ppo_hyperparams (
                make_env,
                n_trials=100,
                n_jobs = 10,
                timeout = 432000000000000000000000
            )
```

Figura 69. Optimización de los hiperparámetros del estudio para cada *trial.*

Finalmente, los resultados del estudio (incluyendo la configuración óptima encontrada y el historial de pruebas) se almacenan mediante la librería *joblib*, lo que permite su posterior análisis y visualización, tal y como se observa en la Figura 70.

```
import joblib
joblib.dump(study, 'ppo_optuna_study.pkl')
```

Figura 70. Almacenamiento de los resultados en *joblib*.

Por otro lado, *Optuna* ofrece la herramienta *Optuna dash-board*, una interfaz que nos permite obtener información en tiempo real sobre el estudio que se está llevando a cabo [43]. Es de gran utilidad a la hora de monitorizar la optimización ya que los estudios de *Optuna* suelen ser muy largos y tardar días. Para acceder basta con escribir el comando de la Figura 71 en el terminal y acceder al enlace mediante nuestro navegador.

```
(.venv) PS C:\Users\Usuario\Desktop\teleco\TFG\Redes_opticas_escenarios> optuna-dashboard
sqlite:///ppo_optuna_study.db.sqlite
Listening on http://127.0.0.1:8080/
Hit Ctrl-C to quit.
```

Figura 71. Acceso a *Optuna-dashboard*.

En la imagen de la Figura 72 se muestra un ejemplo de la evolución de las recompensas obtenidas en cada uno de los *trials*, es decir, las pruebas con distintos hiperparámetros. Se puede observar que el valor más alto del *reward* se obtiene bastante pronto en el estudio.

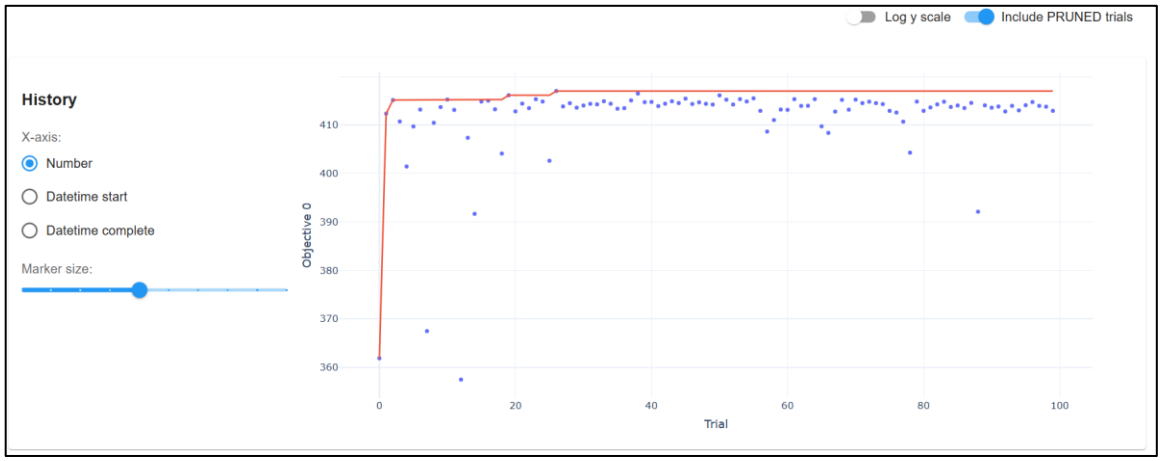


Figura 72. Gráfica que muestra los valores del *reward* obtenidos en cada *trial* de *Optuna*.

Optuna-dashboard también nos ofrece información sobre la importancia de los hiperparámetros a la hora de influir en el valor de la recompensa. Así como del tiempo que tarda en realizarse el estudio y cada uno de los *trials*, como podemos ver en la Figura 73.

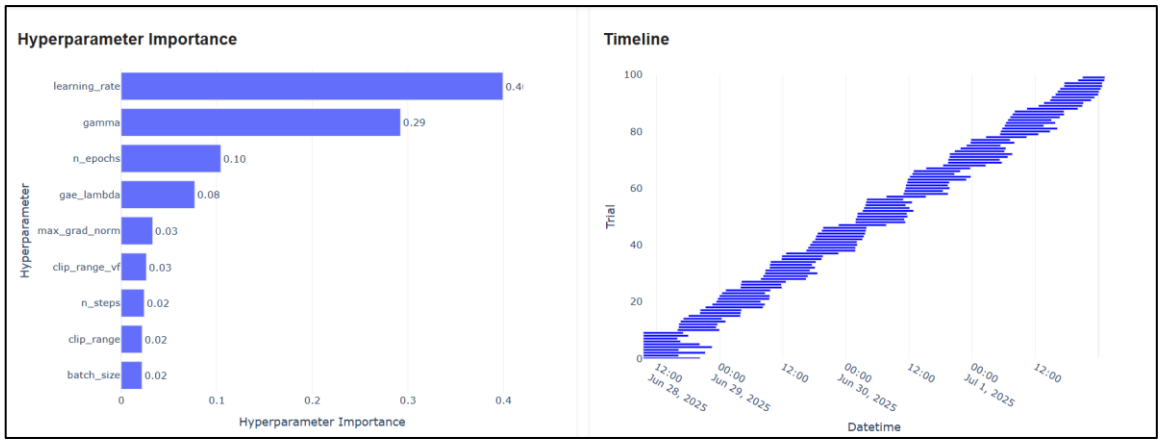


Figura 73. Gráficas que muestran la importancia de cada hiperparámetro y el tiempo que tarda en completarse cada simulación.

El dato más importante que nos da esta interfaz es el *trial* con mejor *reward* que se ha obtenido hasta el momento. Como podemos comprobar en la Figura 74, en este caso el mejor trial fue el 26, el *reward* obtenido es el señalado en rojo, y los hiperparámetros que han dado este resultado son los que se muestran debajo en gris. Esto nos permitirá realizar pequeñas pruebas a lo largo del estudio.

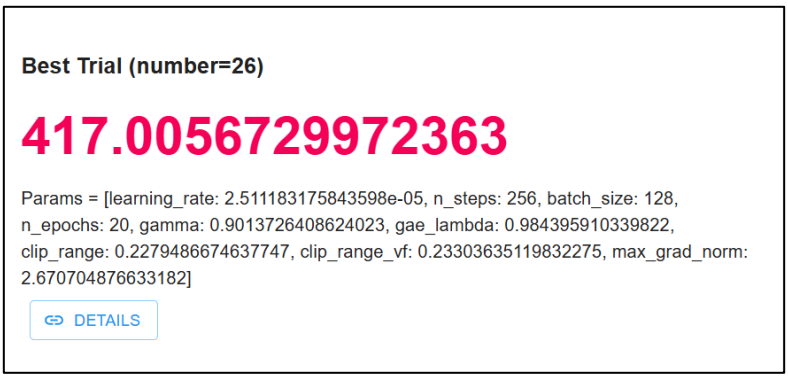


Figura 74. Gráficas que muestran la importancia de cada hiperparámetro y el tiempo que tarda en completarse cada simulación

4.4 Análisis de resultados del Escenario 1: Evaluación con Tráfico Simétrico

En este escenario, se considera que todas las ONTs tienen el mismo SLA y tráfico simétrico, con un ancho de banda garantizado de 600 Mbps. Para este escenario los valores óptimos de los hiperparámetros obtenidos con *Optuna* han sido los que se muestran en la Figura 75. Estos hiperparámetros también serán usados para la optimización de los escenarios 2 y 3, mientas que para el escenario 4 se realizó un nuevo estudio de *Optuna*.

```
n_steps = 256
batch_size = 128
model = PPO(
        "MlpPolicy",
        vec_env,
        n_steps = n_steps,
        batch_size = batch_size,
        learning_rate =2.511183175843598e-05,
        n_epochs= 20,
        gamma= 0.9013726408624023,
        gae_lambda= 0.984395910339822,
        clip_range= 0.2279486674637747,
        clip_range_vf= 0.23303635119832275,
        max_grad_norm= 2.670704876633182,
        tensorboard_log= ". /ppo_tensorboard_logs/",
        device="cuda"
    )
```

Figura 75. Hiperparámetros óptimos obtenidos por *Optuna* para los escenarios 1,2 y3.

Para el escenario 1 se procede a mostrar el resultado de simulaciones tanto con carga media (0.4) como con carga alta (0.8), lo que implica que las ONTs están transmitiendo a una media de 400 Mbps y a 800 Mbps, respectivamente, ya que su tasa máxima es 1 Gpbs. Cabe destacar que debido a la naturaleza de este tipo de tráfico auto-similar (con fuentes de Pareto) la carga al inicio de las simulaciones es muy irregular y tarda en estabilizarse en un valor concreto. Es decir, los valores de la carga siempre serán aproximados y ligeramente distintos para cada ONT en tiempo real.

Para el primer caso, con una carga media de 0.4 obtenemos las gráficas de la Figura 76 y 77 para la ONT 1. Se ha escogido la representación de una única ONT porque todas tienen el mismo comportamiento y para no integrar demasiadas gráficas en la memoria. La Figura 76 representa en tiempo real la evolución del ancho de banda demandado y asignado una ONT en Mbps, ONT 1 en este caso; y la Figura 77 muestra la evolución del tamaño de su cola también a lo largo del tiempo (el tamaño máximo de la cola es 800 Mbits).

Como se puede observar en estas gráficas el tráfico asignado a las ONT está limitado en el valor de 600 Mbps. Este valor corresponde al ancho de banda contratado por el usuario, es decir *B_guaranteed*. Debido a esto cuando el usuario demanda más de 600Mbps el ancho de banda que se le ofrece se trunca en 600Mbps, lo que repercute en la acumulación de paquetes en la cola, mientras que si pide menos de ese valor se le asigna sin problema, contribuyendo a que la cola se vacíe. Esto se observa también en las gráficas, ya que la gráfica que representa la cola sigue el comportamiento de la del tráfico. Se observa que el agente DRL funciona adecuadamente, ya que en la Figura 76 el ancho de banda asignado sigue en tiempo real la evolución del ancho de banda demandado, limitándose al máximo garantizado por el SLA correspondiente (600 Mbps) cuando la demanda lo supera.

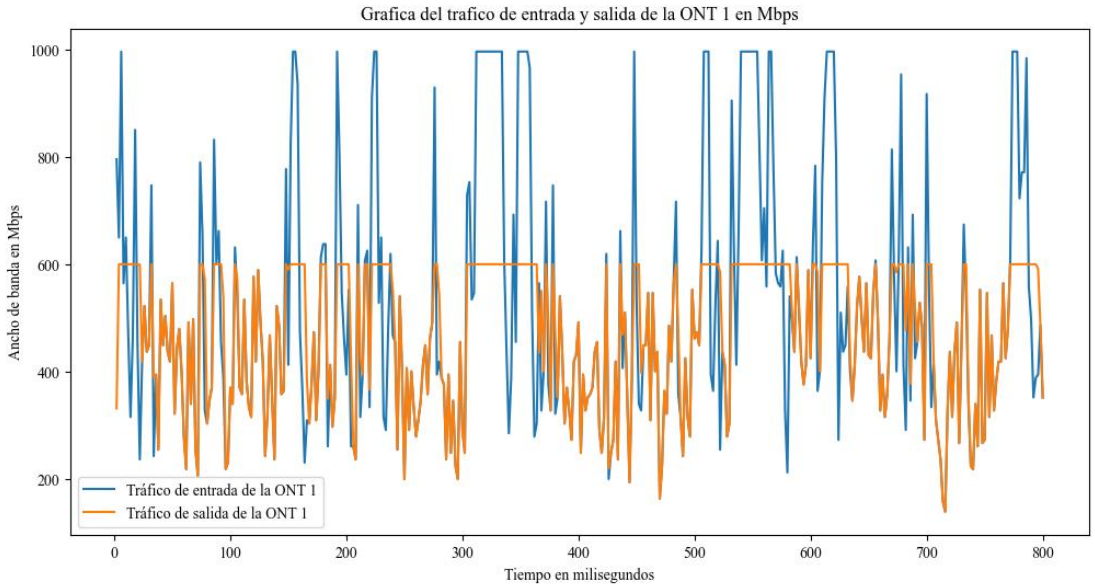


Figura 76. Gráfica que representa el tráfico asignado y de entrada de la ONT 1 con carga 0.4

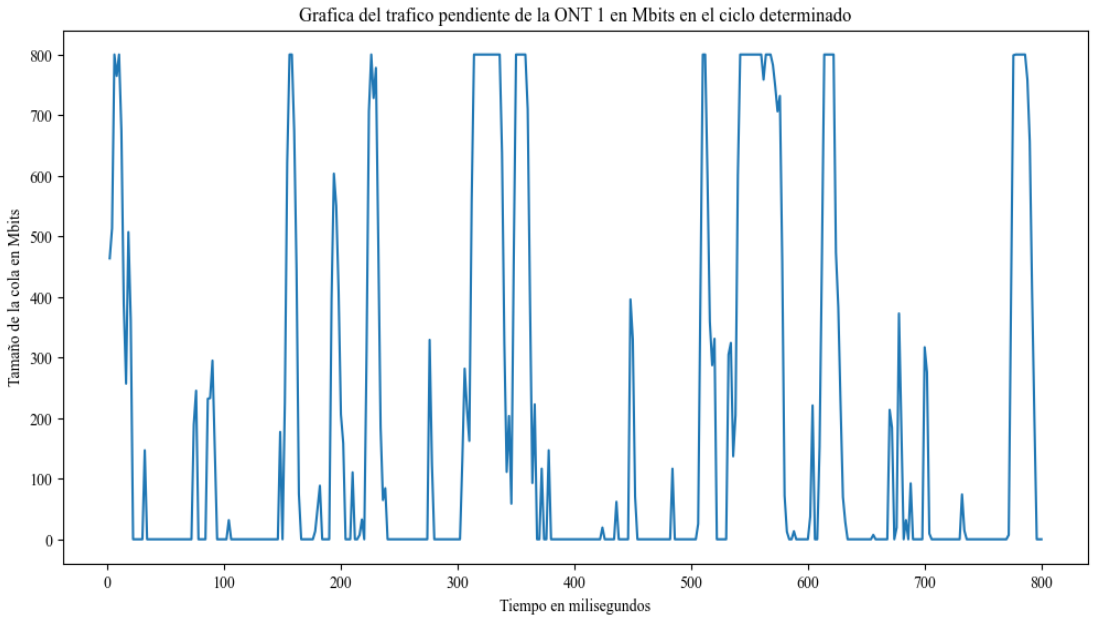


Figura 77. Gráfica que representa la cola de la ONT 1 con carga 0.4.

Para estudiar el comportamiento de los distintos escenarios se empleó la curva de *reward* que permite observar el valor de la recompensa acumulada durante el entrenamiento para dicho escenario. En la Figuras 78 (sin optimizar con *Optuna*) y 79 (optimizado) se muestra la mejora de la curva de *reward* una vez aplicada la optimización con *Optuna*.

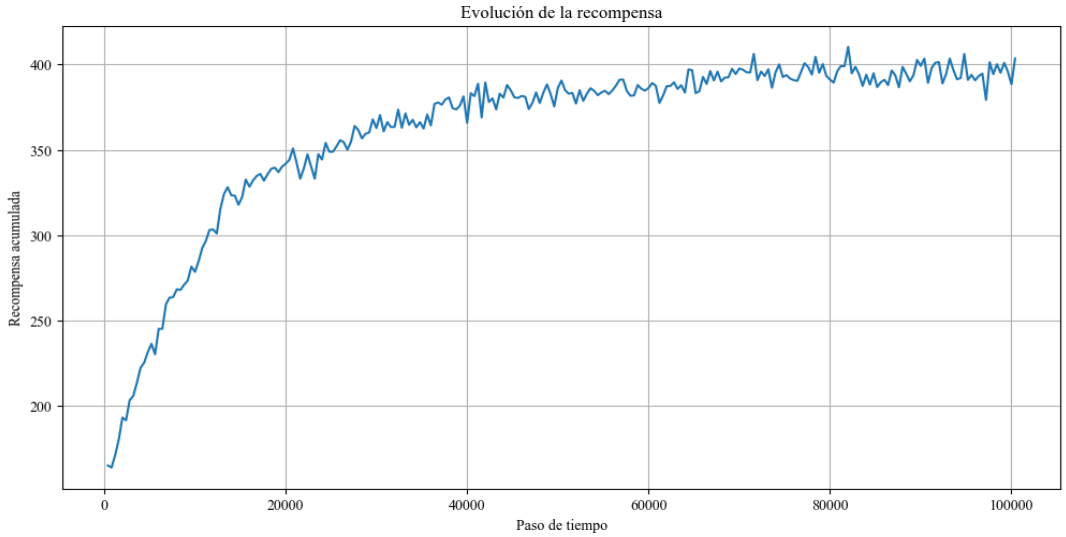


Figura 78. Gráfica de evolución de la recompensa en el Escenario 1 sin optimizar con carga 0.4.

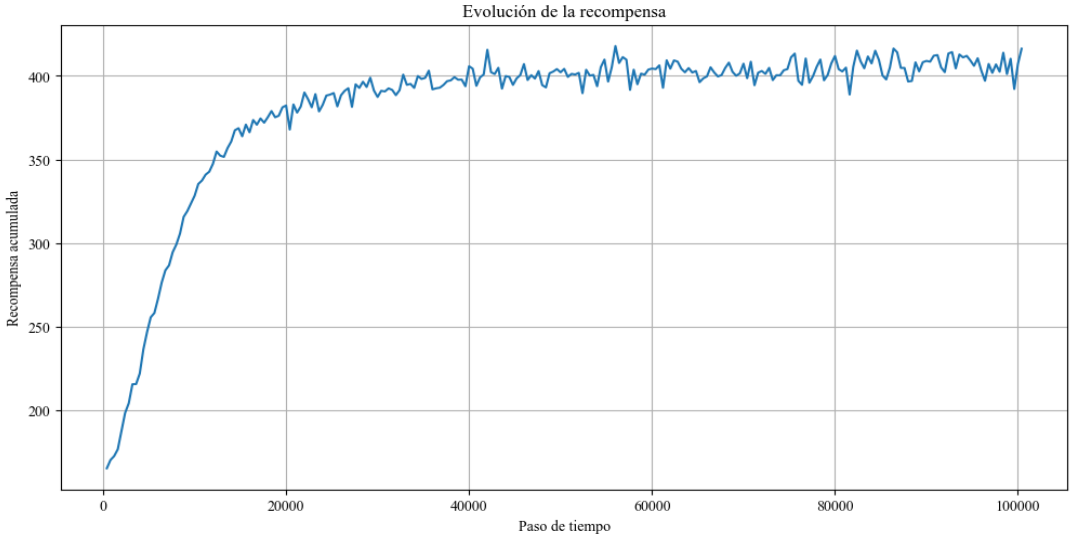


Figura 79. Gráfica de evolución de la recompensa en el Escenario 1 optimizada con carga 0.4.

Para el caso de carga de 0.8 (ONTs transmitiendo a 800 Mbps), se muestran las mismas gráficas en las Figura 80 y 81. Se observa que al aumentar la carga a un valor alto observamos que el tráfico de entrada se sitúa casi siempre por encima de 600 Mbps, lo que hace que a la ONT se le asigne un ancho de banda de 600Mbps la mayoría del tiempo, ya que es su ancho de banda garantizado, esto es, su máximo. El mismo comportamiento se ve en la Figura 81 en la evolución de las colas, donde se observa que la cola se mantiene prácticamente llena en toda la simulación. Una vez más se demuestra que el agente funciona de manera óptima para este escenario con carga muy alta.

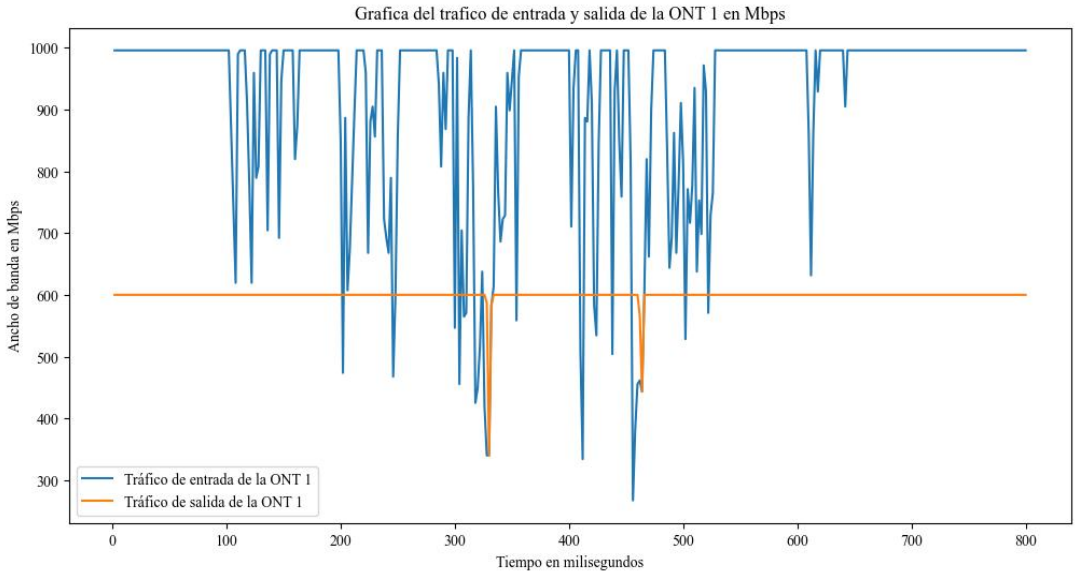


Figura 80. Gráfica que representa el tráfico asignado y de entrada de la ONT 1 con carga 0.8.

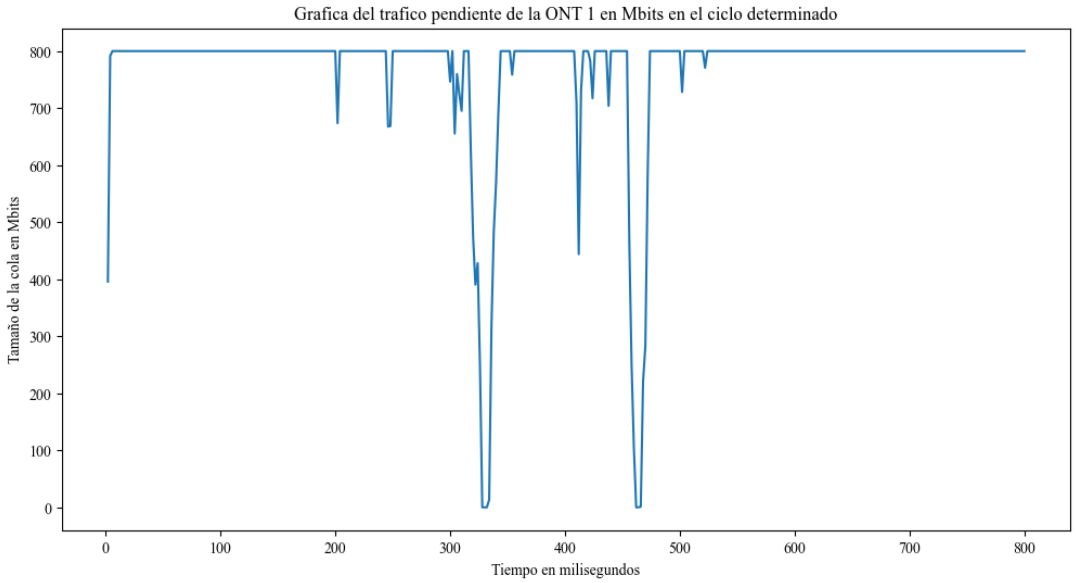


Figura 81. Gráfica que representa la cola de la ONT 1 con carga 0.8.

Al aumentar la carga, la curva del *reward* se vuelve mucho más estable y no tiene nada de ruido (Figuras 82 y 83). Esto se debe a que el sistema permanece casi constantemente saturado y, por tanto, la distribución de ancho de banda es casi siempre constante, ofreciendo 600Mbps casi siempre, lo que hace que este caso sea muy poco variable y bastante sencillo de comprender para el agente.

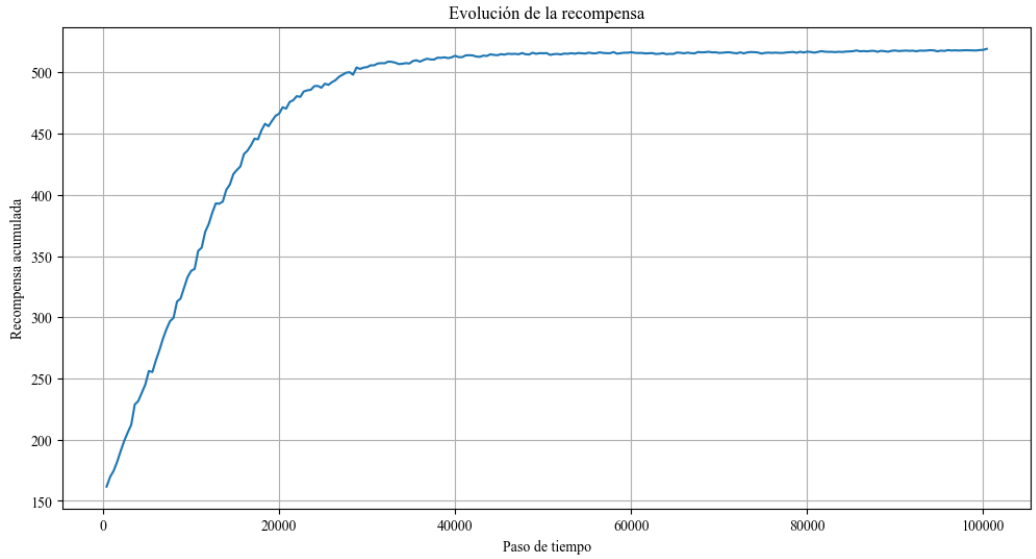


Figura 82. Gráfica de evolución de la recompensa en el Escenario 1 sin optimizar con carga 0.8.

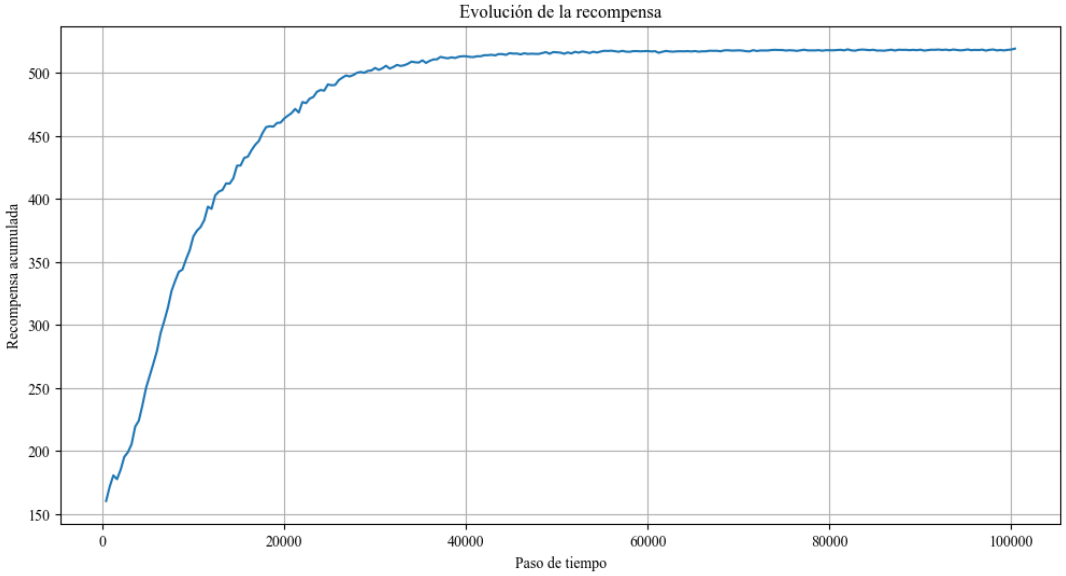


Figura 83. Gráfica de evolución de la recompensa en el Escenario 1 optimizada con carga 0.8.

4.5 Análisis de resultados del Escenario 2: Evaluación con Tráfico Asimétrico

En el escenario 2, cada una de las ONTs presenta una carga distinta, lo que hace que el comportamiento del agente deba adaptarse a la carga de cada una, teniendo en cuenta que algunas ONTs demandarán más y otras menos.

Para el escenario 2 vamos a observar las gráficas que representan el tráfico de dos ONTs distintas dentro de la misma simulación. Estas ONTs han sido elegidas en concreto porque son representativas del funcionamiento general del escenario. En concreto, se ha escogido

la ONT 2 que tiene una carga de 0.4 (400 Mbps) y la ONT 6 que tiene una carga media de 0.2 (200 Mbps).

En la Figura 84 podemos comprobar que hay más carga que en la Figura 86, donde el ancho de banda asignado casi nunca llega a los 600 Mbps. En concreto la Figura 84 (ONT 2) muestra la evolución del ancho de banda asignado en tiempo real y el ancho de banda demandado para la ONT 2, y se observa que al igual que en el escenario 1, cuando esta demanda un valor superior a 600 Mbps, se le asigna un ancho de banda de 600 Mbps, mientras que si demanda menos se le asigna lo que pide. La Figura 85 muestra la evolución del tamaño de la cola a lo largo del tiempo y se observa que el tamaño tiene el mismo perfil del tráfico de entrada de la Figura 84, esto es lógico ya que cuando hay mucha demanda y se supera el SLA la demanda restante que no se satisface se almacena en las colas. Una vez que la demanda va cayendo, las colas se van vaciando. Este comportamiento nos indica que el agente DRL se comporta de forma adecuada.

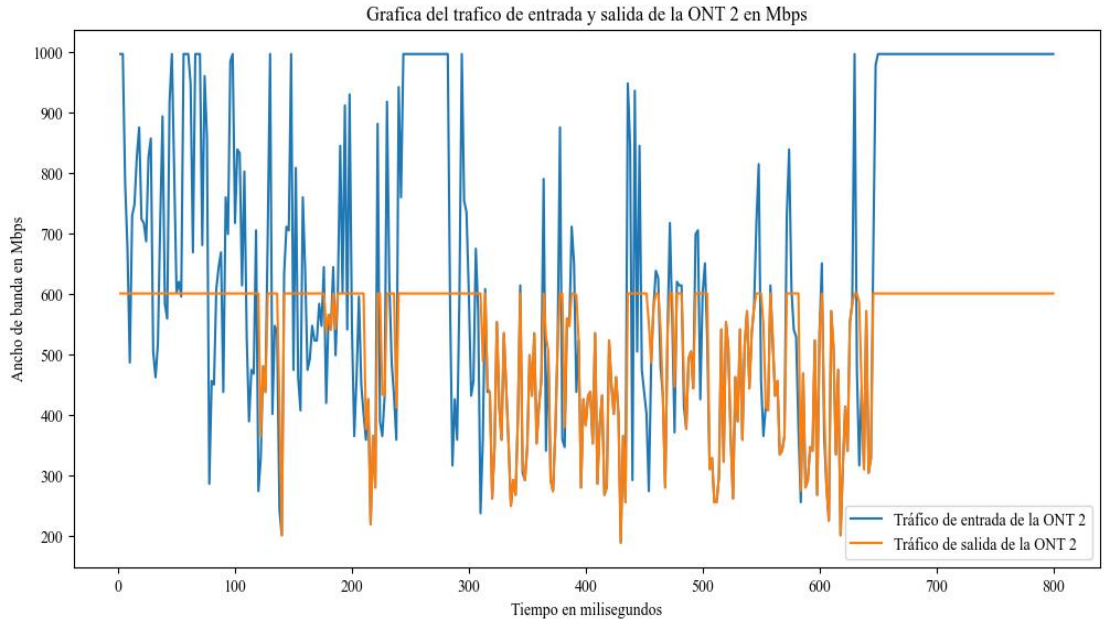


Figura 84. Gráfica que representa el tráfico asignado y de entrada de la ONT 2 en el escenario 2 con carga 0.4.

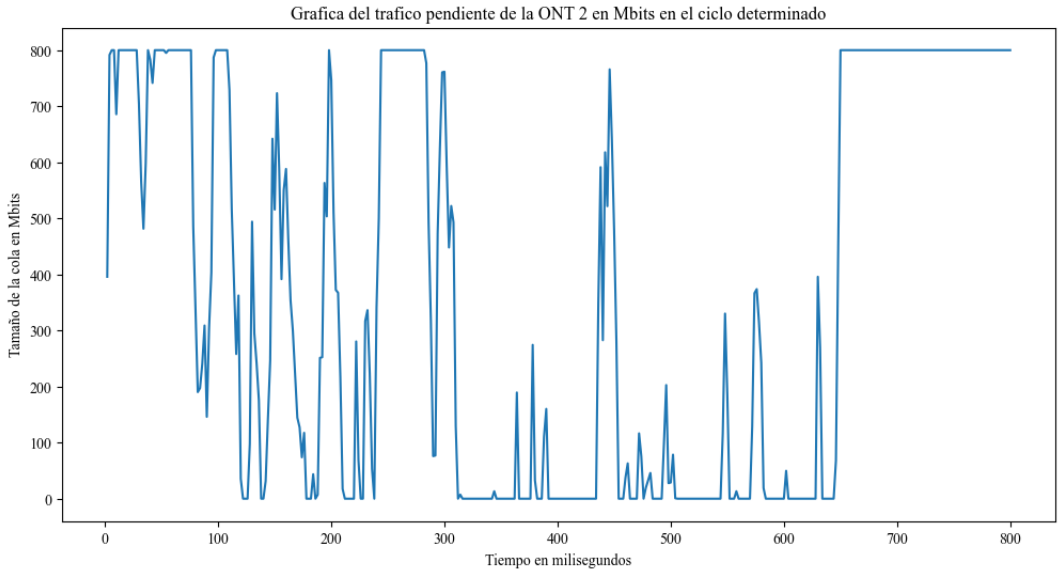


Figura 85. Gráfica que representa la cola de la ONT 2 en el escenario 2 con carga 0.4.

Por otro lado, en la Figura 86 (ONT 6) se observa que, al ser la carga muy baja, en general tendremos poca demanda por parte de la ONT salvo en algunos puntos concretos y por lo tanto le da casi siempre el ancho de banda demandado. La Figura 87 muestra la evolución del tamaño de la cola a lo largo del tiempo y se observa que las colas están casi siempre vacías salvo en algunos puntos donde el tráfico de entrada es alto. Por lo tanto, el agente DRL muestra un comportamiento adecuado: asigna a la ONT el ancho de banda solicitado siempre que la demanda sea inferior a 600 Mbps (*B_guaranteed*), lo que permite vaciar las colas; mientras que, cuando la demanda supera este valor, asigna el máximo garantizado y la parte restante se acumula en la cola.

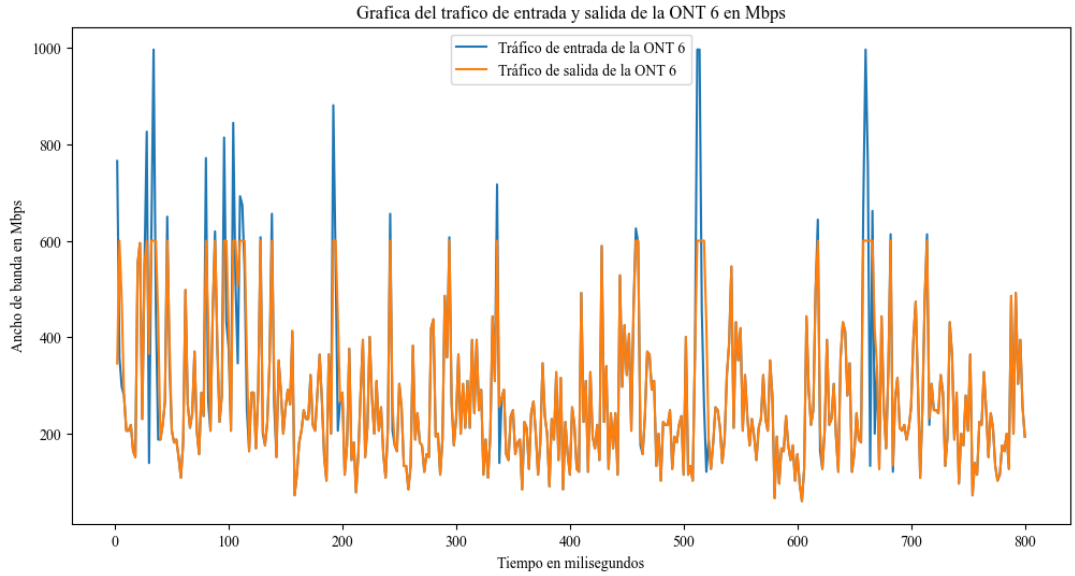


Figura 86. Gráfica que representa el tráfico asignado y de entrada de la ONT 6 en el escenario 2 con carga 0.2.

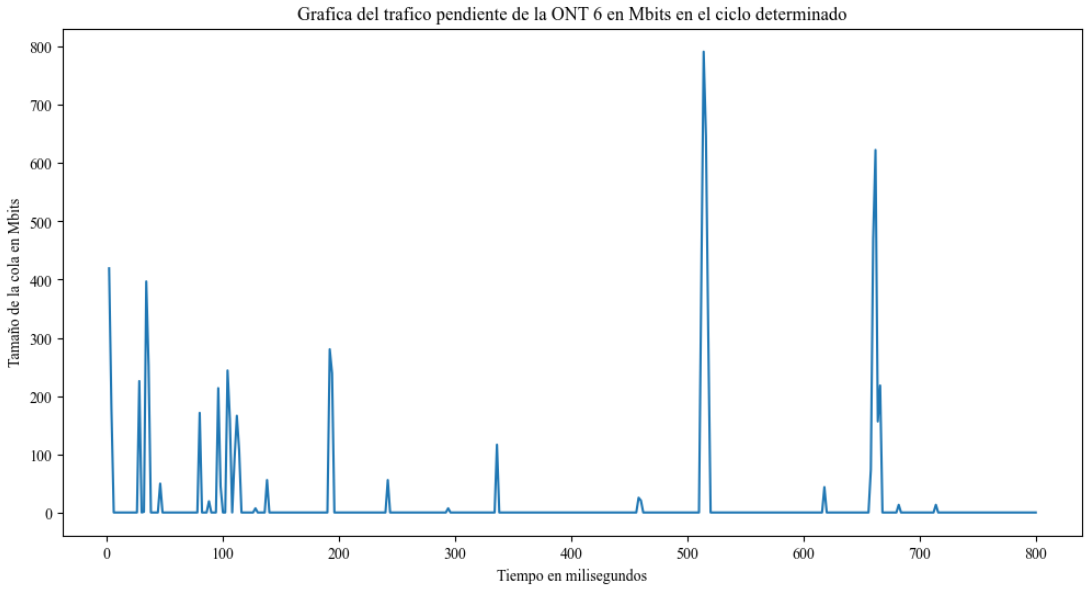


Figura 87. Gráfica que representa la cola de la ONT 6 en el escenario 2 con carga 0.2.

Si analizamos ahora la función de recompensa, se observa en la Figura 88, al aumentar la complejidad del escenario, la curva de *reward* parece menos estable. Igualmente, la optimización de hiperparámetros demuestra una grandísima mejora (Figura 89), dando un resultado estable y con poco ruido.

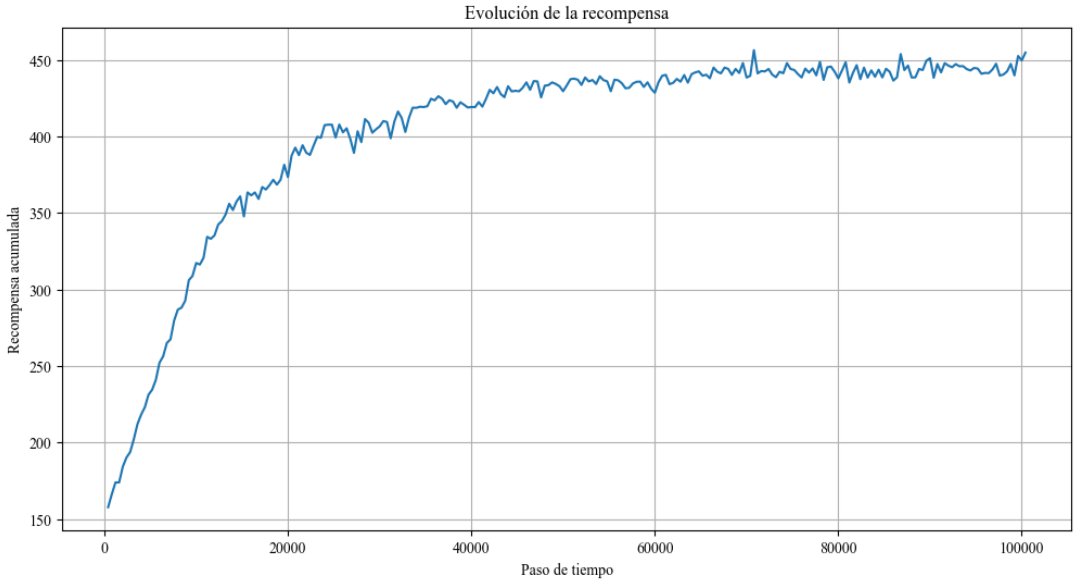


Figura 88. Gráfica de evolución de la recompensa en el Escenario 2 sin optimizar.

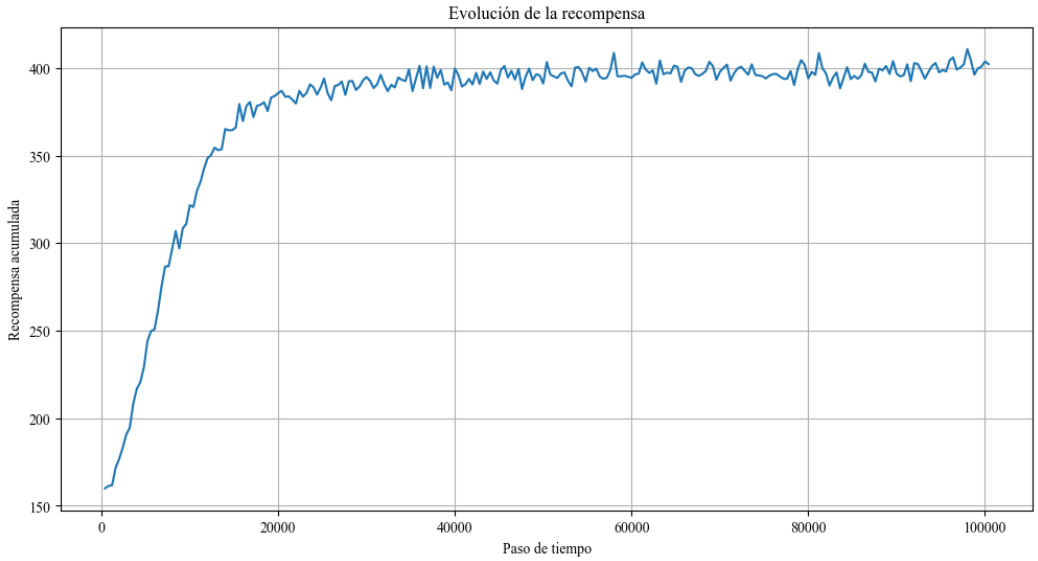


Figura 89. Gráfica de evolución de la recompensa en el Escenario 2 optimizada.

A modo de resumen, se puede concluir que el agente DRL funciona adecuadamente, ya que en las figuras de tráfico asimétrico el ancho de banda asignado sigue en tiempo real la evolución del ancho de banda demandado, limitándose al máximo garantizado por el SLA correspondiente (600 Mbps) cuando la demanda lo supera. Además, el modelo muestra un rendimiento igualmente consistente con ONTs que generan tráfico simétrico y asimétrico sin necesidad de optimización adicional, lo que refuerza su robustez en entornos 10G-EPON con características de tráfico no homogéneas.

4.6 Análisis de resultados del Escenario 3: Evaluación con condiciones dinámicas

En el escenario 3 se distinguen dos casos de ancho de banda garantizado con comportamiento dinámico. En el primer caso, el SLA parte de 600 Mbps y en un instante aleatorio se reduce a 400 Mbps (estos parámetros son configurables). El segundo caso representa una situación más extrema: en un instante aleatorio, el SLA de cada ONT cambia a un valor diferente elegido aleatoriamente entre 200 Mbps y 800 Mbps (estos parámetros son configurables). A continuación, se pasa a describir el análisis de resultados para cada uno de los casos.

Análisis de resultados y evaluación de prestaciones para el Caso 1.

Para el primer caso observamos el comportamiento que se muestra en la Figura 90 y Figura 92 para la ONT 2 y la ONT 3, respectivamente (ambas para una carga de 0.4). Como se

puede ver en las imágenes, el SLA cambia en instantes distintos, pero al mismo valor, 400 Mbps. Se observa además que el modelo DRL actúa de forma adecuada, asignando el ancho de banda en función de la demanda, siguiendo de manera óptima sus variaciones en tiempo real y limitándolo al valor máximo permitido cuando esta lo supera. En las gráficas que representan el comportamiento de las colas (Figura 91 y 93), al reducirse el SLA a 400 Mbps la ocupación de estas aumenta en ambos casos. Esto se debe a que, al reducir el SLA, el ancho de banda asignable a las ONTs se limita a 400 Mbps en lugar de 600 Mbps, por lo que se satisface menos demanda y el excedente se acumula en las colas.

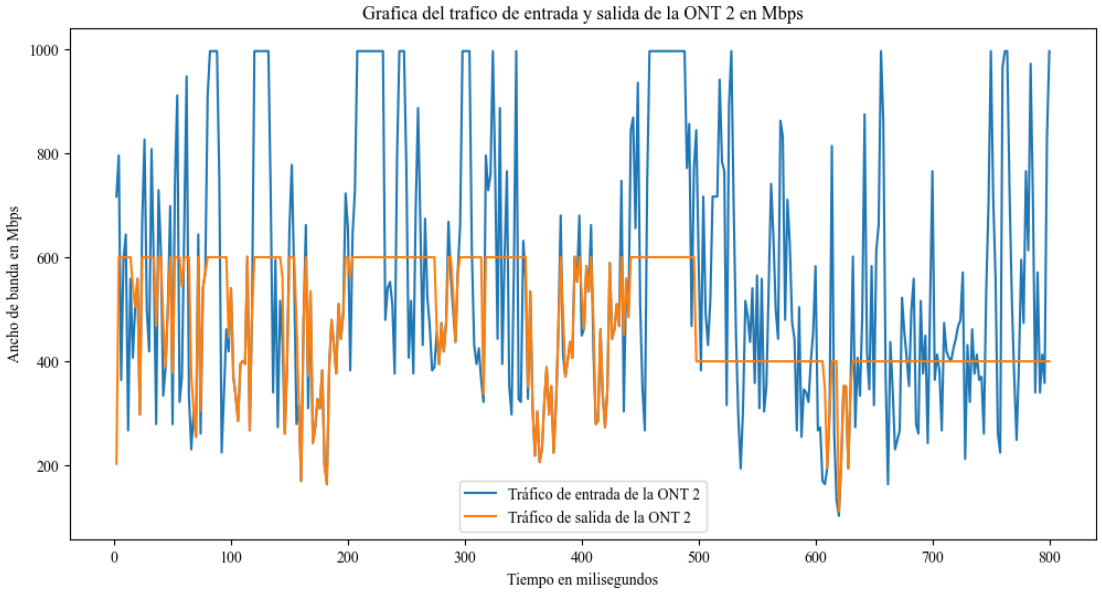


Figura 90. Gráfica que representa el tráfico asignado y de entrada de la ONT 2 en el escenario 3.

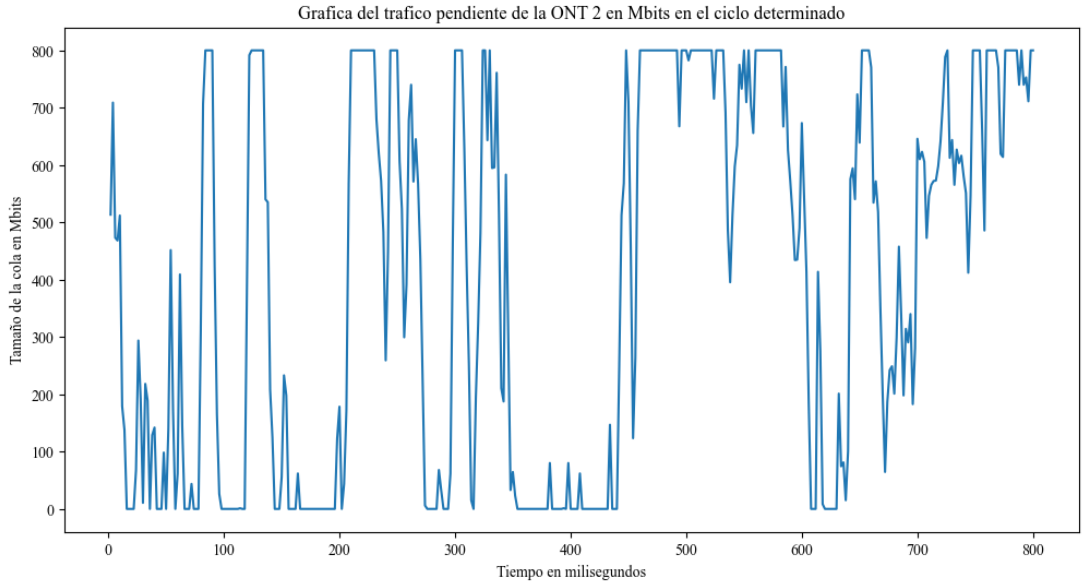


Figura 91. Gráfica que representa la cola de la ONT 2 en el escenario 3.

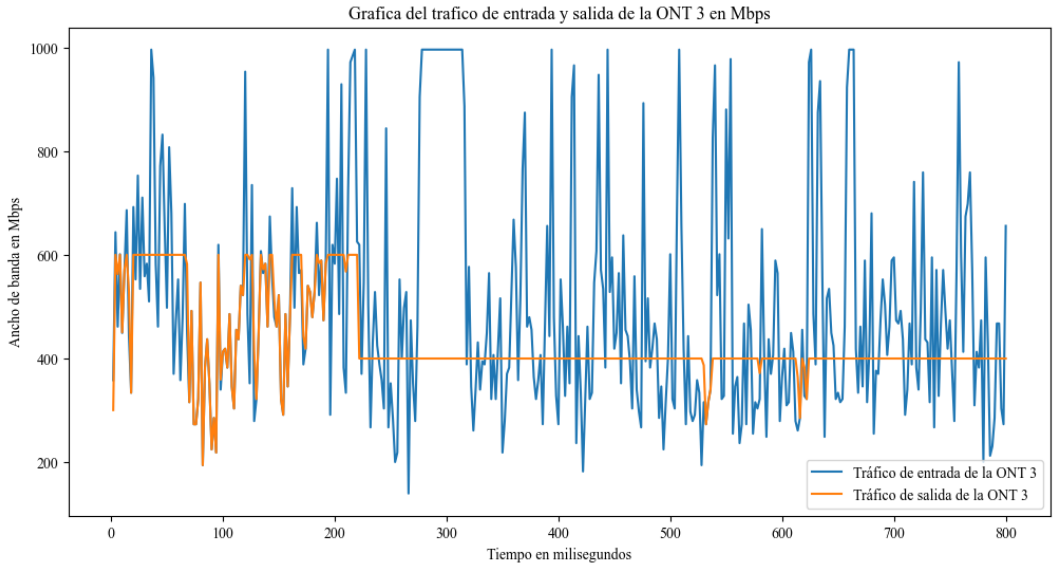


Figura 92. Gráfica que representa el tráfico asignado y de entrada de la ONT 3 en el escenario 3.

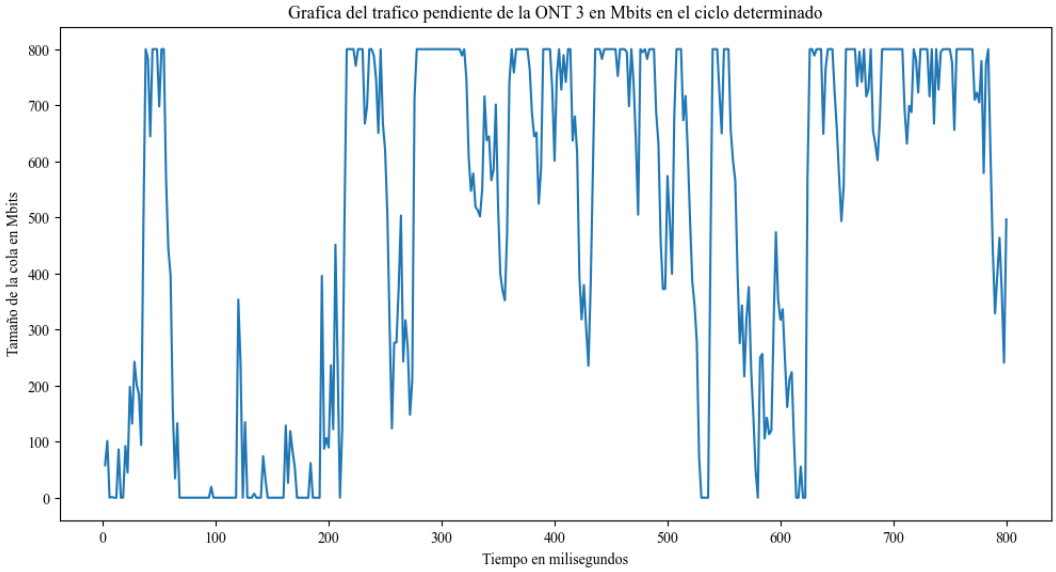


Figura 93. Gráfica que representa la cola de la ONT 3 en el escenario 3.

Si se observa ahora el comportamiento de la recompensa, la mejora en este caso de la curva de *reward* es sustancial. De hecho, en el caso sin optimizar (Figura 94) el *reward* no parece estabilizarse del todo, aumentando ligeramente. Mientras tanto, el caso optimizado, la Figura 95, el *reward* se estabiliza antes.

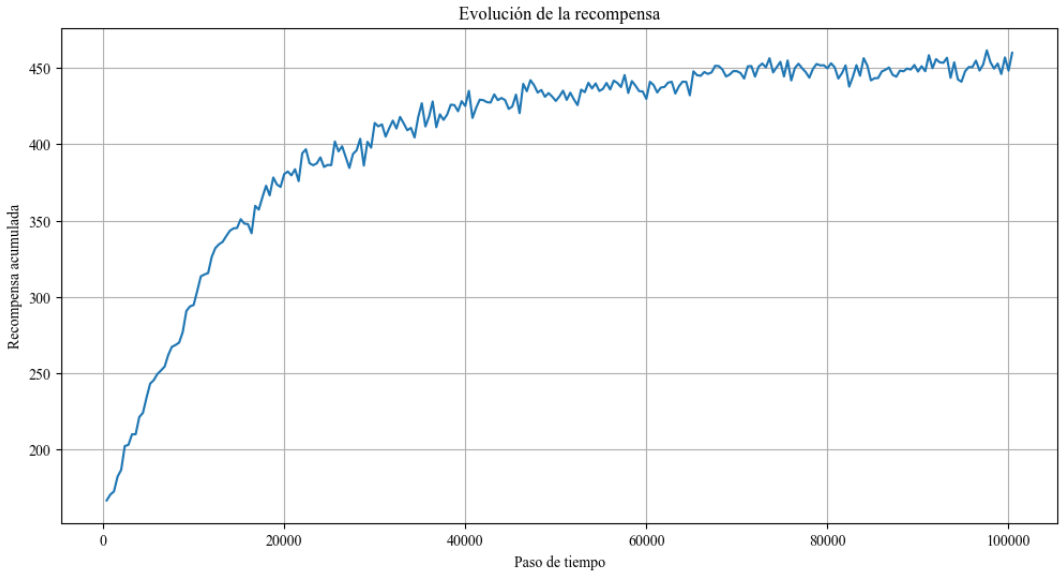


Figura 94. Gráfica de evolución de la recompensa en el Escenario 3 con SLA fijo sin optimizar.

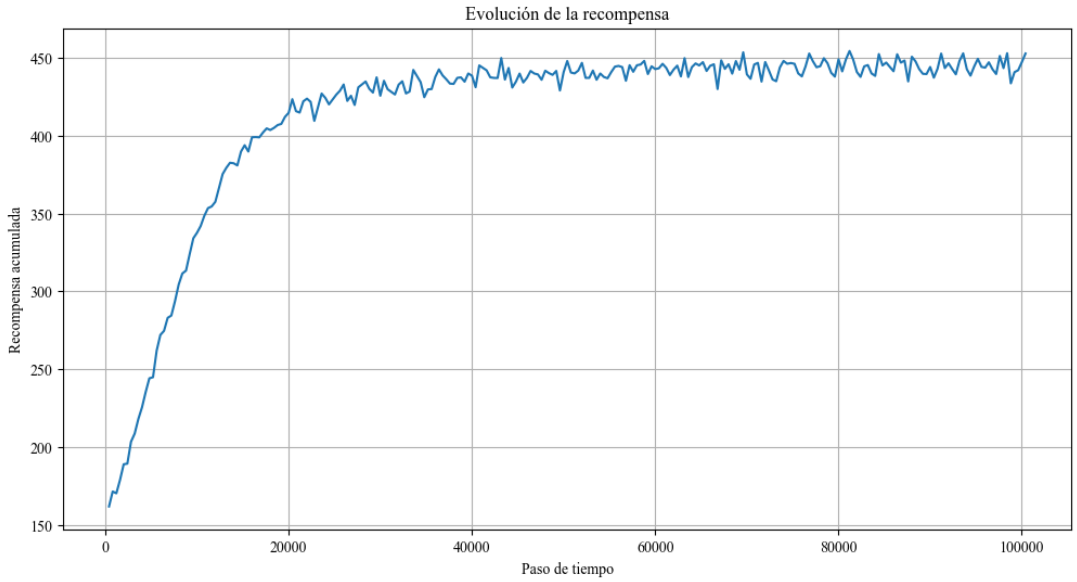


Figura 95. Gráfica de evolución de la recompensa en el Escenario 3 con SLA fijo optimizado

Análisis de resultados y evaluación de prestaciones para el Caso 2

En este segundo caso observamos que el SLA cambia también en momentos distintos, pero, además, cambia a un valor distinto para cada ONT. Este caso es muy extremo que nos aporta información sobre la capacidad de los modelos DRL de aprender en casuísticas más complejos. La Figura 96 y Figura 98, muestran la evolución en tiempo real del ancho banda asignado versus el ancho de banda demandado para la ONT 1 y ONT 3, respectivamente, para una carga media de 0.4 (400 Mbps). Se observa que el modelo DRL a pesar de enfrentarse a casos muy distintos es capaz de asignar correctamente el ancho de

banda, nunca sobrepasándose del SLA en el que se encuentra y ajustando el ancho de banda asignado a la demanda cuando este se encuentra por debajo del SLA siempre que la cola no esté demasiado saturada.

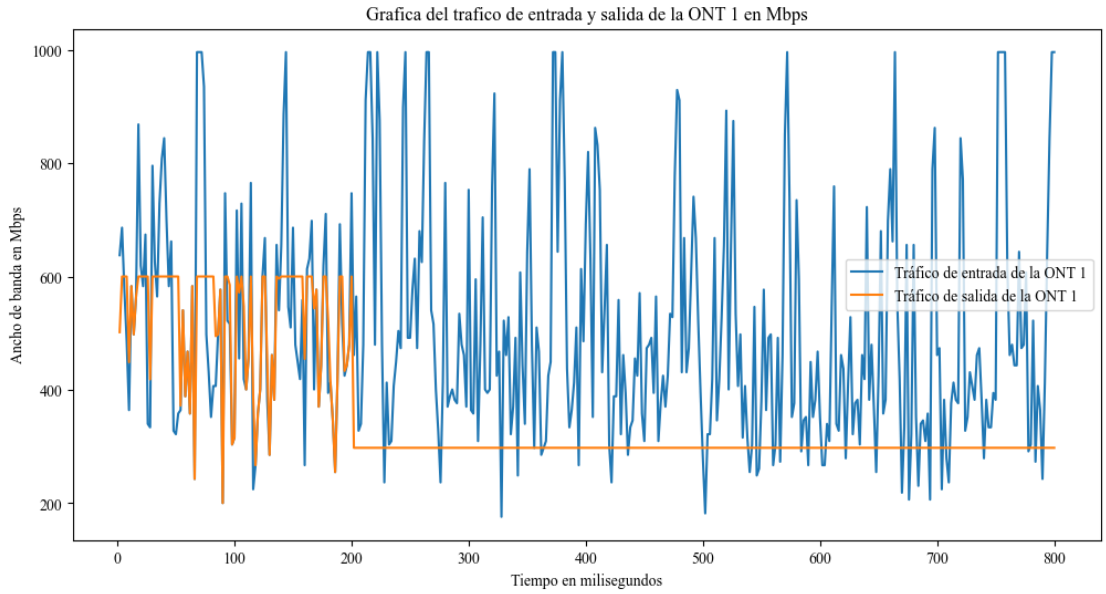


Figura 96. Gráfica que representa el tráfico asignado y de entrada de la ONT 1 en el escenario 3.

Por otro lado, si observamos la Figura 97 el efecto del cambio del SLA sobre la cola es muy claro. Esto se debe a que el SLA ha bajado a un valor muy bajo, de unos 300 Mbps y, por tanto, es muy complicado satisfacer toda la demanda. En la Figura 99 no se ve tanto cambio, ya que el SLA sube hasta 700 Mbps y la carga (400 Mbps) no es muy alta.

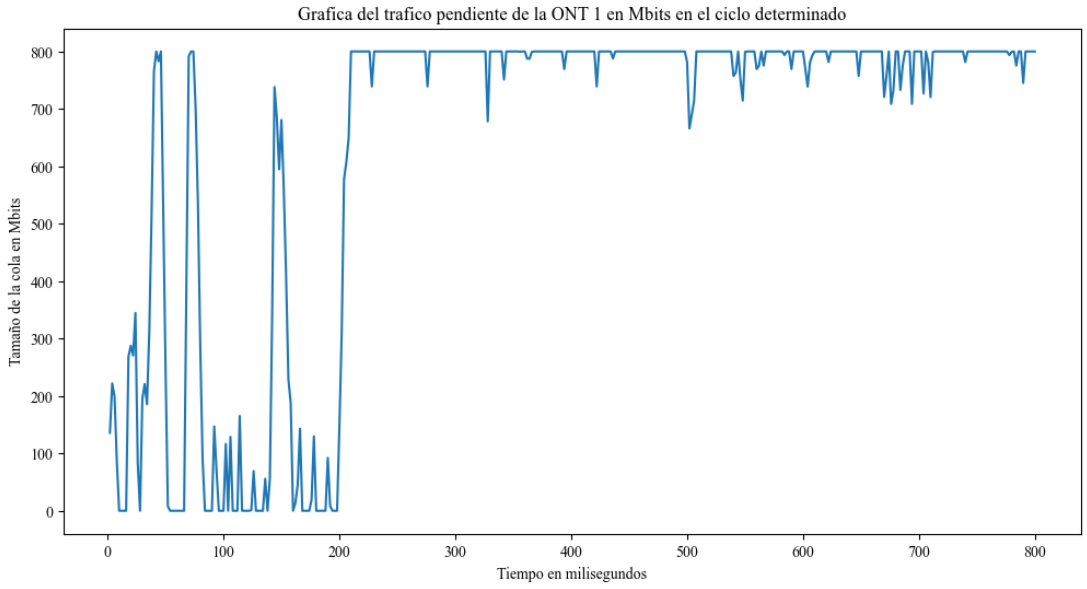


Figura 97. Gráfica que representa la cola de la ONT 1 en el escenario 3.

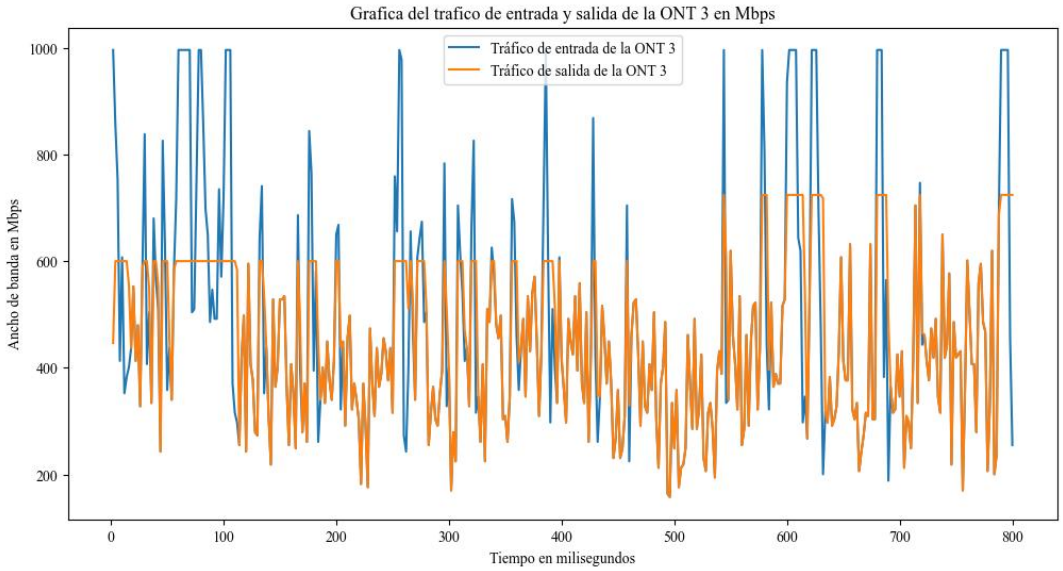


Figura 98. Gráfica que representa el tráfico asignado y de entrada de la ONT 3 en el escenario 3.

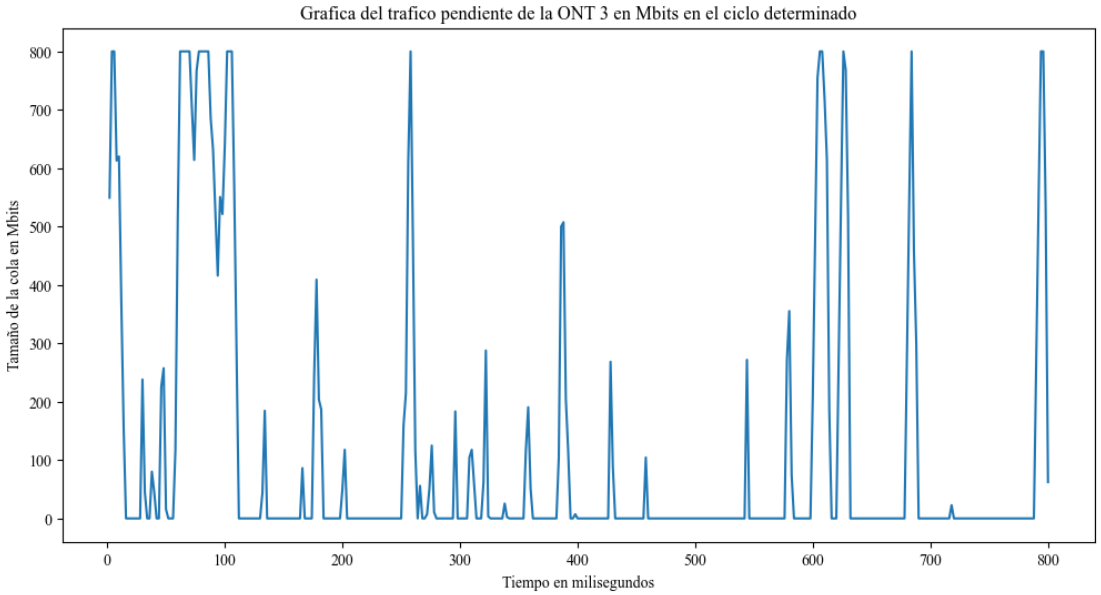


Figura 99. Gráfica que representa la cola de la ONT 3 en el escenario 3.

De nuevo, la mejora de la gráfica de *reward* tras la optimización de hiperparámetros es considerable. Como se puede observar en las Figuras 100 y 101 en la primera la gráfica converge hacia un valor óptimo más o menos en torno a las 40000 *timesteps*, mientras que, en el caso optimización se observa la mejora en torno a los 20000, esto es, mucho antes.

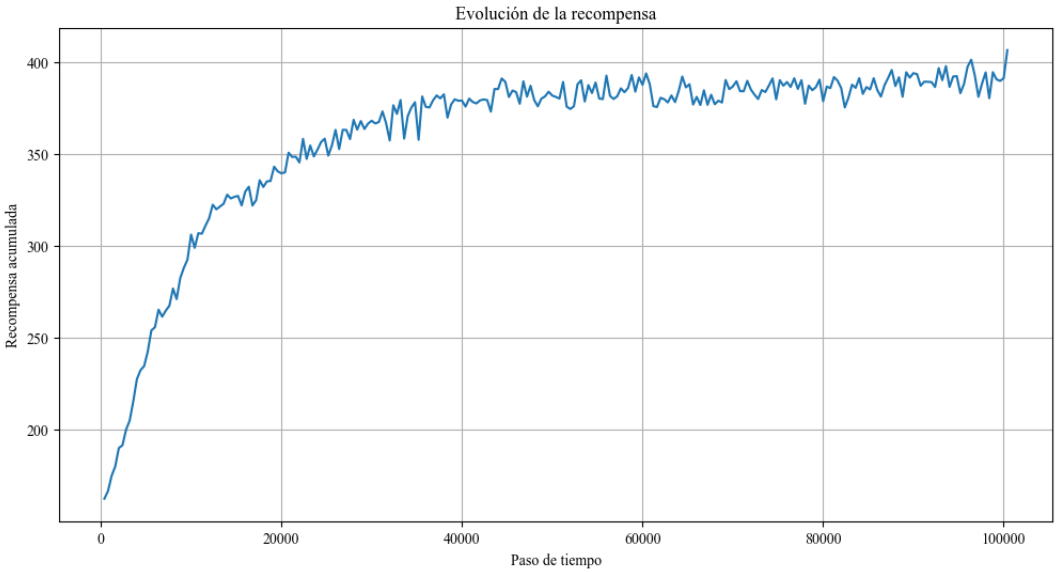


Figura 100. Evolución de la recompensa en el Escenario 3 con SLA aleatorio sin optimizar.

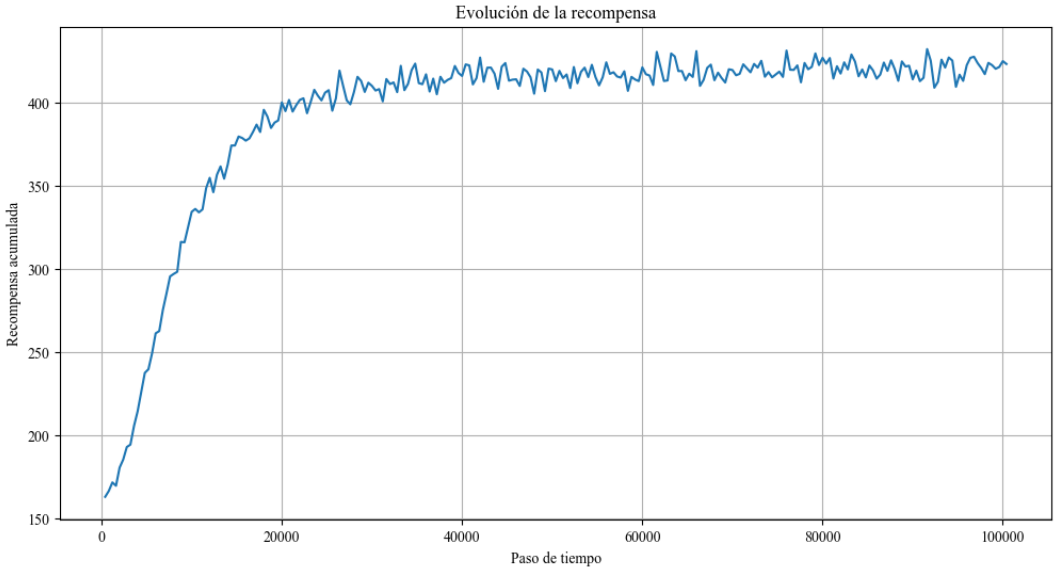


Figura 101. Evolución de la recompensa en el Escenario 3 con SLA aleatorio optimizado.

4.7 Análisis de resultados del Escenario 4: Evaluación con ONTs garantizadas y no garantizadas

El escenario 4 representa un caso más complejo y realista que los escenarios previos. En él se hace la distinción entre dos tipos de ONTs. Las primeras seis son garantizadas, es decir, se comportan como las ONTs del escenario 1 (6 ONTs) y representan clientes de operadores tradicionales a los que se les ofrece un ancho de banda garantizado de 600Mbps. Por otro lado, el segundo tipo son ONTs flexibles (10 ONTs), que representarían

usuarios que han accedido a un plan de una operadora virtual que les ofrece 800 Mbps de máximo garantizado y 480 Mbps de mínimo garantizado, esto es 60% del máximo. Los hiperparámetros obtenidos al optimizar este escenario con *Optuna* fueron los que se muestran en la Figura 102. El proceso para obtenerlos fue análogo a los anteriores casos.

```
model = PPO(
        "MlpPolicy",
        vec_env,
        learning_rate= 0.001,
        n_steps= 256,
        batch_size= 32,
        n_epochs= 30,
        gamma= 0.94,
        gae_lambda= 0.8878686405859539,
        clip_range= 0.3492601502681344,
        clip_range_vf= 0.14642359654938747,
        max_grad_norm= 3.460846959513738,
        tensorboard_log="./ppo_tensorboard_logs/",
        device="cpu"
    )
```

Figura 102. Hiperparámetros óptimos obtenidos por *Optuna* para el escenario 4

Para hacer las pruebas de rendimiento en este escenario, se han empleado casos de carga alta (0.8), carga se puede observar en las siguientes figuras como una línea discontinua en azul claro. Observamos que la ONT 2, mostrada en la Figura 103 presenta un comportamiento idéntico al de las ONT del escenario 1 que recibían una alta carga. Como era necesario que la carga fuese lo más alta posible, se necesitaba que las simulaciones fuesen lo más largas posibles, ya que el generador de tráfico auto-similar con fuentes de Pareto que empleamos es algo inestable al principio (debido a la naturaleza del propio tráfico). Para garantizar la mayor estabilidad posible, se hicieron simulaciones de 8000 ms en lugar de 800 ms como en los anteriores escenarios. Esto hace que las gráficas mostradas a continuación se vean de forma algo menos definida, pero fue necesario para la correcta interpretación del escenario.

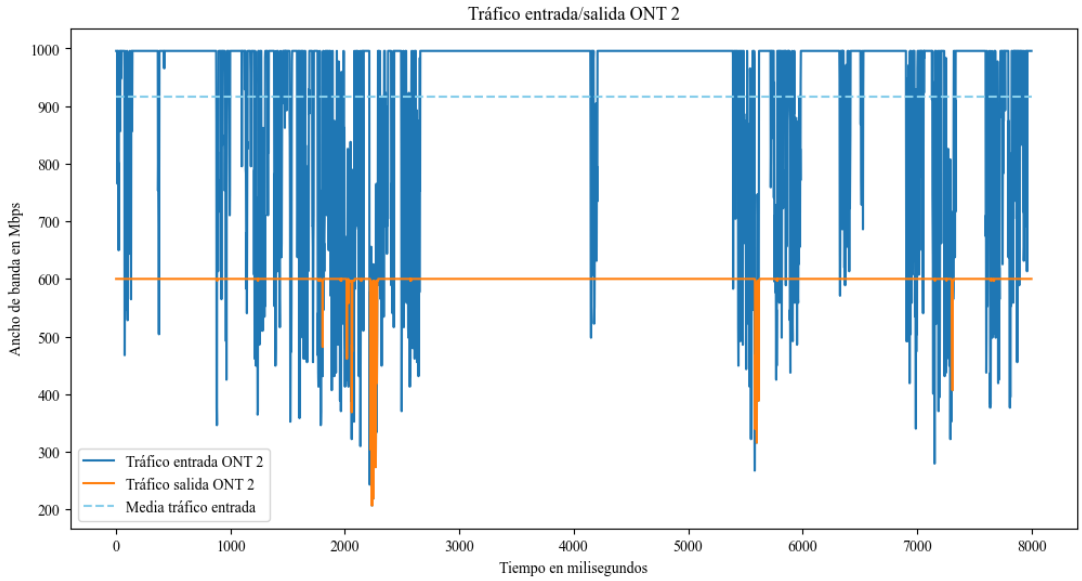


Figura 103. Gráfica que representa el tráfico asignado y de entrada de la ONT 2 (garantizada) en el escenario 4.

A continuación, en la Figura 104 se observa como la cola es muy similar a la gráfica del tráfico demandado, lo que es coherente según lo que hemos definido en el agente DRL e indica que la asignación se está haciendo de forma correcta.

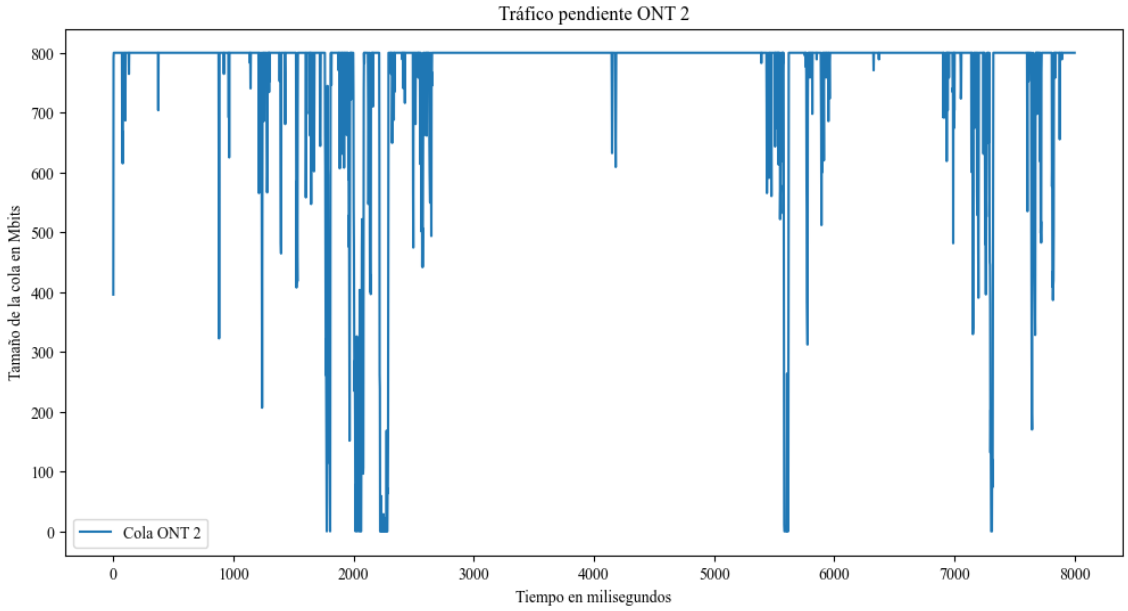


Figura 104. Gráfica que representa la cola de la ONT 2 (garantizada) en el escenario 4.

A continuación, se muestra el comportamiento de las ONTs flexibles. Como podemos ver en la Figura 105, que muestra el comportamiento de la ONT 7, su tráfico asignado puede encontrarse por debajo del de las ONTs garantizadas o ligeramente por encima. Esto se debe a que, en esta clase de servicios ofrecidos por las operadoras virtuales, los usuarios

de ONTs garantizadas tienen prioridad a la hora de recibir su asignación de ancho de banda. Por tanto, en casos donde la demanda de los usuarios es muy alta, los usuarios con tarifas flexibles no recibirán el ancho de banda máximo que han contratado (en este caso 800 Mpbs). Cabe destacar que en esta figura concreta observamos una carga media cercana a 0.9, de unos 0.87. Esto se explica debido a la inestabilidad del generador de tráfico de Pareto.

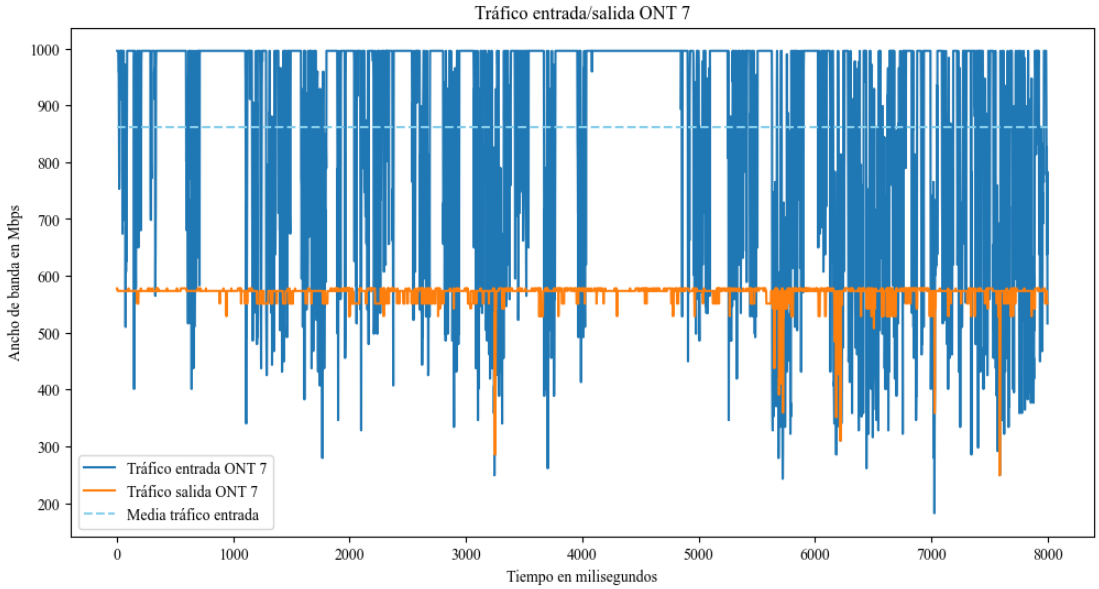


Figura 105. Gráfica que representa el tráfico asignado y de entrada de la ONT 7 (flexible) en el escenario 4.

De nuevo, en la Figura 106, la cola sigue aproximadamente la gráfica del tráfico demandado, lo que indica que para las ONT flexibles la asignación es adecuada.

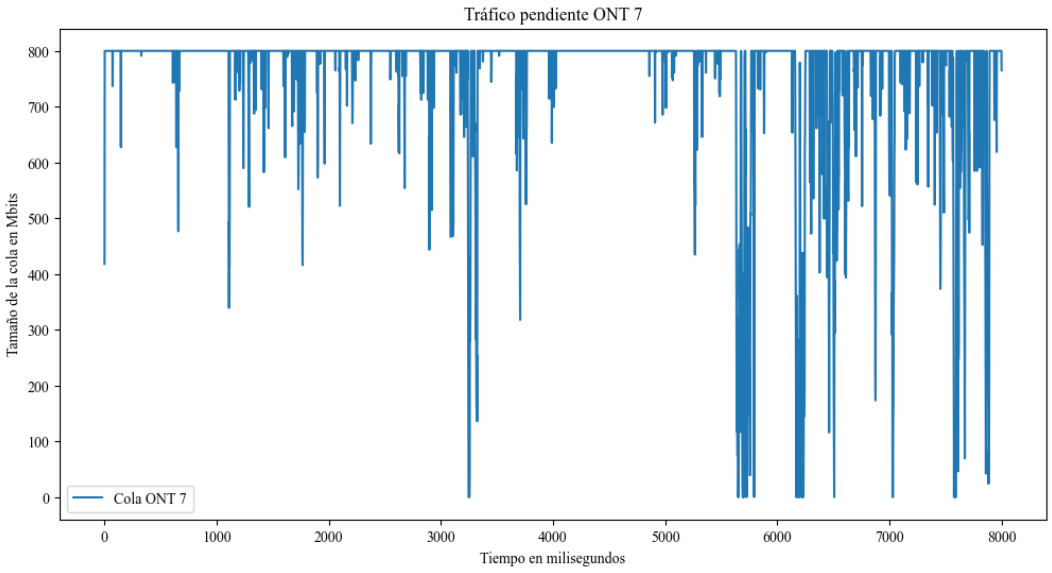


Figura 106. Gráfica que representa la cola de la ONT 7 (flexible) en el escenario 4.

En las siguientes figuras (Figura 107 y 108) observamos el comportamiento de otra de las ONTs flexibles del sistema, en concreto la ONT 10. Podemos observar como esta recibe un ancho de banda ligeramente superior a la anterior, de unos 650 Mbps. En un caso totalmente ideal, estas ONTs deberían recibir exactamente lo mismo, pero el índice IJF no es tan preciso.

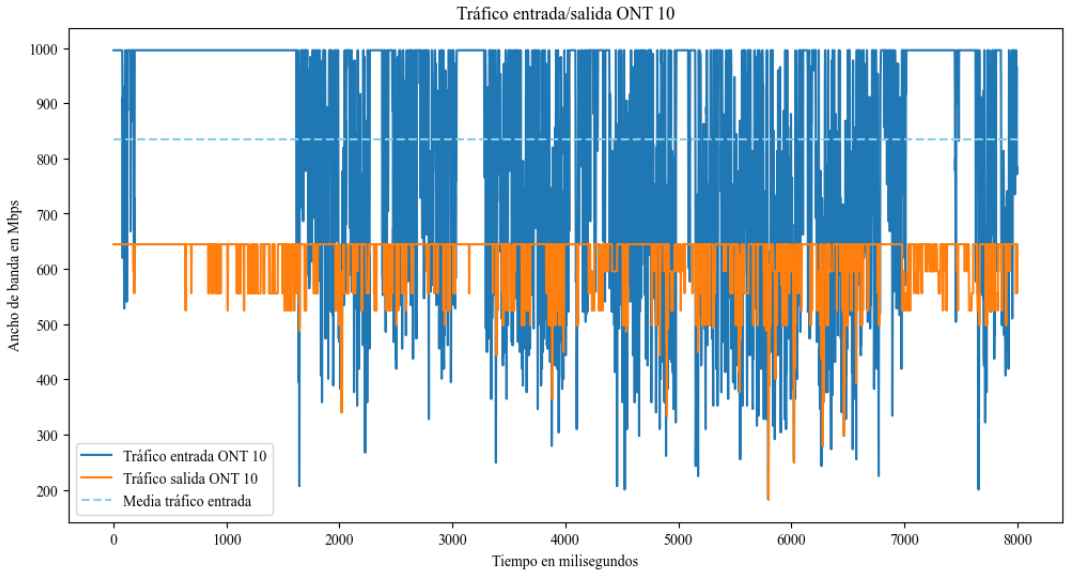


Figura 107. Gráfica que representa el tráfico asignado y de entrada de la ONT 10 (flexible) en el escenario 4.

Observamos que en el caso de la ONT 10 la cola (Figura 108) está más desocupada que en el de la ONT 7. Esto se explica si tenemos en cuenta que, por un lado, su carga media es ligeramente menor, de aproximadamente 820 Mbps y que además se le asigna un ancho de banda garantizado más alto que a la ONT 7.

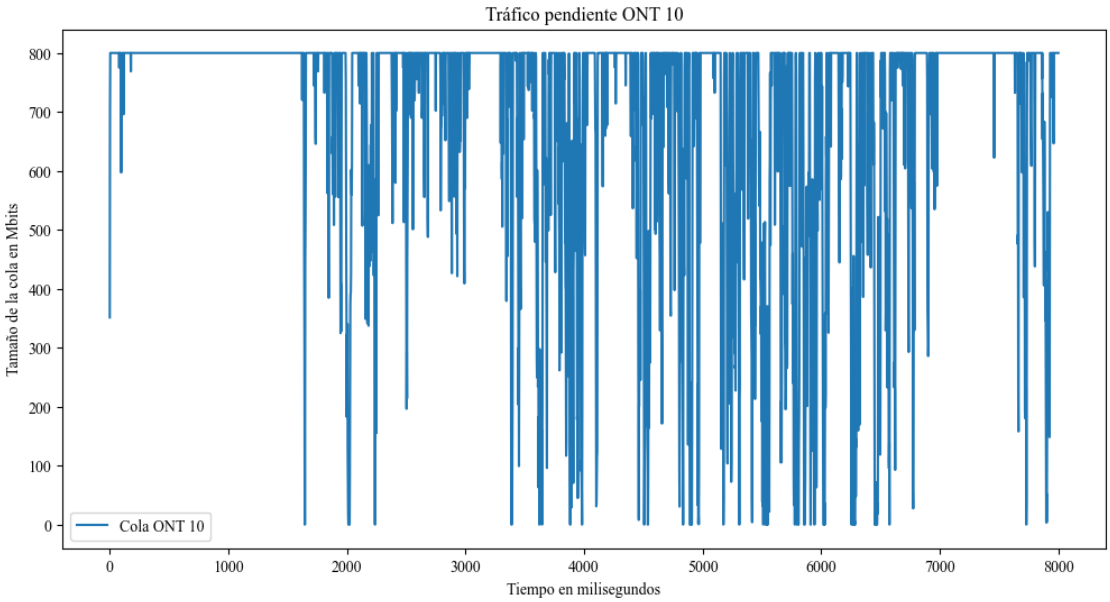


Figura 108. Gráfica que representa la cola de la ONT 10 (flexible) en el escenario 4.

Además, se incluyó una gráfica que reflejase el valor del índice IJF a lo largo de la simulación para las ONTs flexibles. Como se observa en la Figura 109, el IJF se mantiene en torno a 1, lo que implica que la equidad entre las ONTs flexibles es muy buena.

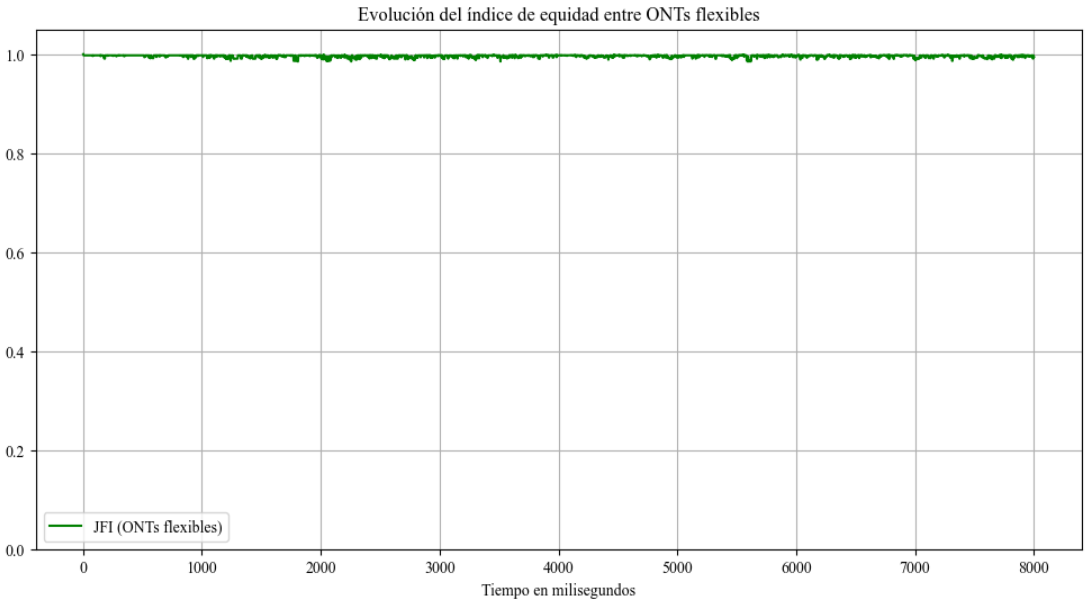


Figura 109. Gráfica que representa el valor que toma el índice IFJ en el escenario 4.

Un resultado destacable es la curva de *reward*, que muestra gran estabilidad y una rápida convergencia hacia su valor óptimo, tal como se observa en la Figura 110, incluso en el caso sin optimización. Esto se debe probablemente a la discretización del espacio de acciones, que simplifica la búsqueda del agente de acciones óptimas, reduciendo

considerablemente las acciones posibles entre las que puede elegir. Esta opción de discretizar el espacio limita en cierta medida la exploración del modelo; sin embargo, se abarca un rango amplio de situaciones, lo que permite que el modelo DRL esté bien representado y muestre un comportamiento óptimo. Por otro lado, la optimización en el escenario 4 fue análoga a la de los tres escenarios anteriores, empleando la herramienta *Optuna.* Si bien la curva de *reward* optimizada (Figura 111) parece levemente más inestable, se observa como converge más rápido, antes de los *20000 timesteps,* a diferencia del caso sin optimizar. Además, consigue un valor de *reward* acumulado algo más alto.

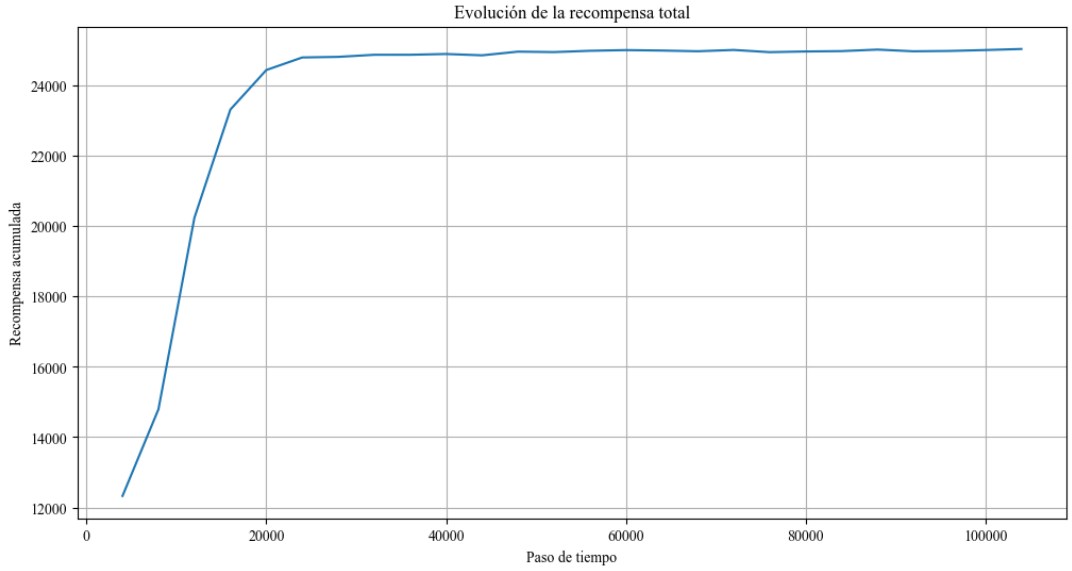


Figura 110. Gráfica de evolución de la recompensa en el Escenario 4 sin optimizar.

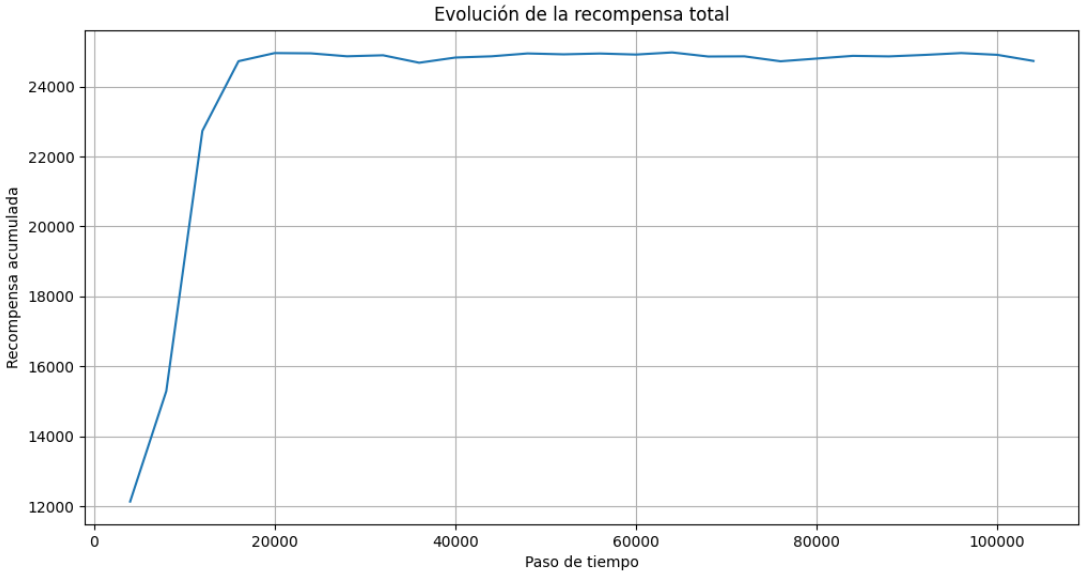


Figura 111. Gráfica de evolución de la recompensa en el Escenario 4 optimizada.

Uno de los controles adicionales que se hizo en el escenario 4 fue representar el ancho de banda total asignado a las ONTs, como se muestra en la Figura 112, aunque en este caso se muestra sin la optimización de hiperparámetros. Esto permite asegurarse de que no nos sobrepasamos de la capacidad máxima de la OLT (10Gbps). Como podemos observar la mayoría del tiempo el ancho de banda total asignado se encuentra en torno a 9,6 Gbps y solo en puntos despreciables se supera la capacidad. Esto se puede justificar si tenemos en cuenta que la capacidad total no es lo mismo que la capacidad efectiva del sistema. En concreto este tipo de redes tienen una codificación 64b/66b, es decir 64 bits de datos y los encapsula en palabras de 66 bits, de esta forma los dos bits extras funcionan como una suerte de cabeceras. Así, si consideramos que por cada 64 bits netos se añaden dos extras, tendremos una eficiencia de un 97% en decir 9,7 Gbps de datos útiles [44].

Finalmente, una mejora clave del proceso de optimización de hiperparámetros con Optuna se evidenció en la gráfica de utilización máxima de la capacidad de la OLT (Figura 113). En comparación con la Figura 112, sin optimización, la señal presenta una reducción significativa del ruido y, además, no supera los 10 Gbps, lo cual constituye una característica fundamental. Debido a ello, la optimización mejora de forma considerable el modelo, al dar lugar a un sistema mucho más cuidadoso en la gestión de la capacidad de la OLT. Es importante destacar que las fluctuaciones en la asignación de ancho de banda se deben al carácter ráfagoso del tráfico, de modo que en determinados instantes algunas ONTs apenas generan demanda y en otros bastante.

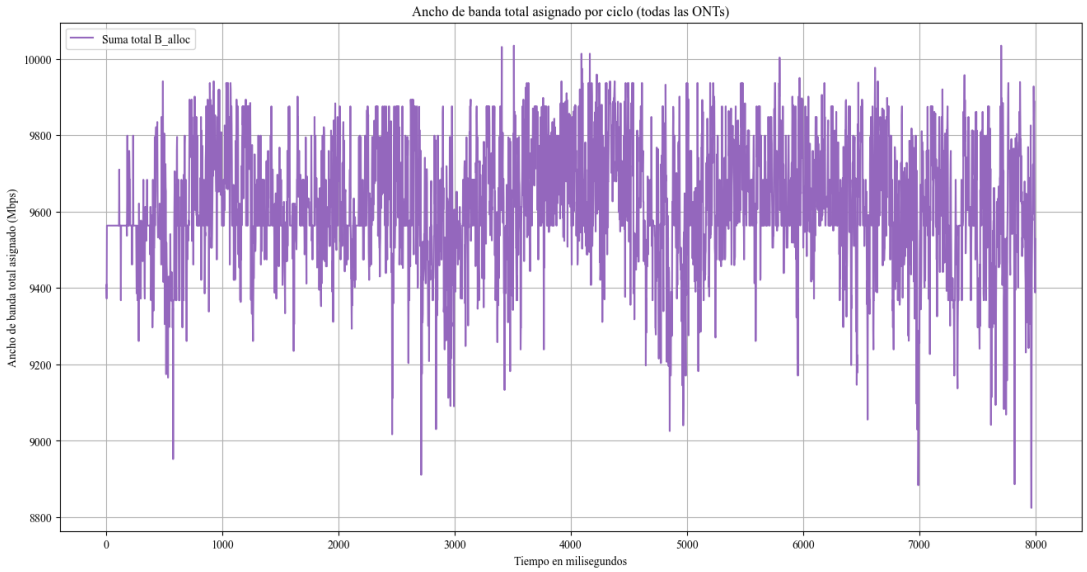


Figura 112. Gráfica de ancho de banda total asignado sin optimización.

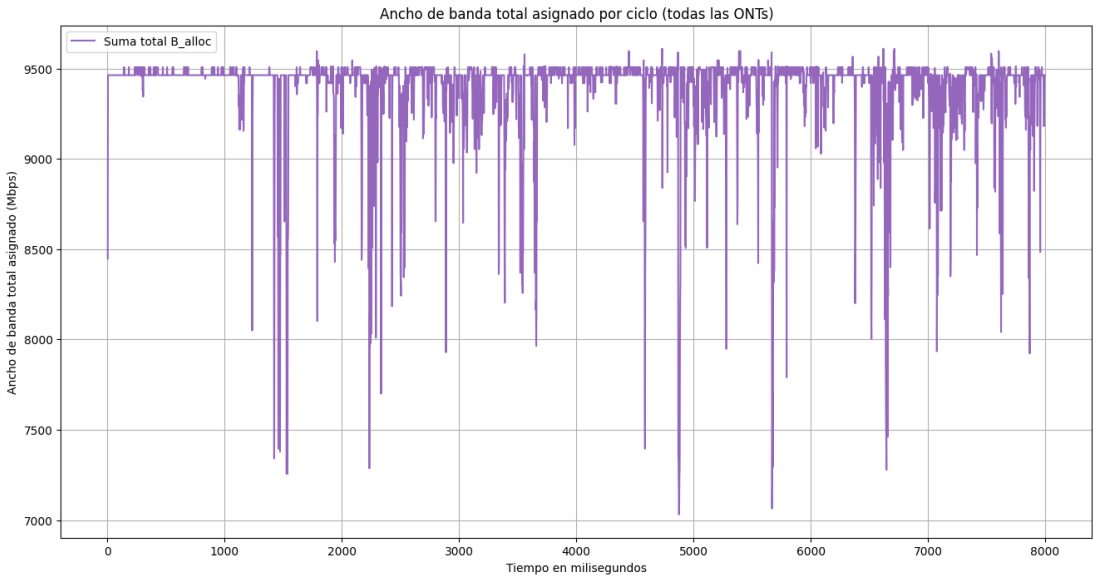


Figura 113. Gráfica de ancho de banda total asignado optimizada.

4.8 Conclusiones

En el cuarto capítulo se ha estudiado el resultado de las simulaciones para los cuatro escenarios y la optimización de hiperparámetros con *Optuna*, Los resultados obtenidos en los escenarios 1 y 2 demuestran que el agente asigna de correctamente el ancho de banda tanto con tráfico simétrico como asimétrico, mostrando una mejora en la curva de recompensa cuando se aplica la optimización de hiperparámetros.

En el escenario 3, en el caso en el que el SLA va variando de forma dinámica en diferentes instantes, el agente DRL demostró la capacidad de adaptarse, manteniendo la estabilidad del *reward* incluso en condiciones extremas como la mostrada en el caso 2.

Finalmente, en el caso del escenario 4, que incorporó ONTs con SLA garantizado y ONTs flexibles, era el caso más complejo ya que requería priorizar usuarios y equilibrar el reparto del ancho de banda sobrante. Incluso en este caso el agente desarrollado presenta un comportamiento muy bueno.

En general, los resultados son muy positivos y ponen de manifiesto el potencial de combinar técnicas de inteligencia artificial en 10G-EPON con un agente DRL entrenado mediante PPO, demostrando su efectividad en distintos escenarios de red con patrones dinámicos de tráfico y condiciones variables de calidad de servicio.

5

Conclusiones y Líneas futuras

5.1 Conclusiones

Las conclusiones principales obtenidas una vez llevado a cabo este Trabajo Fin de Grado se enfocan en la utilidad de emplear herramientas de aprendizaje profundo en problemas relacionados con la gestión de datos en sistemas de telecomunicaciones. En este caso concreto en redes de acceso ópticas 10G-EPON.

Después de haber estudiado el comportamiento de los modelos de aprendizaje por refuerzo profundo, así como de las redes 10G-EPON, parece claro que su uso conjunto puede ser de gran utilidad a la hora de simular escenarios dinámicos y realistas.

En este trabajo se integró en el simulador previamente desarrollado un patrón de tráfico autosimilar, generado a partir de fuentes de Pareto, con el fin de representar un comportamiento más realista.

También se ha procedido a modificar los entornos previamente desarrollados para que se adaptasen al nuevo modelo de simulación, desarrollando y optimizando las funciones de *reward* para los distintos escenarios, así como adaptando el resto de funciones al nuevo comportamiento del simulador. Estos entornos incluían modelos de tráfico simétrico, asimétrico y dinámico. Además, se ha desarrollado un nuevo escenario 4 más complejo y realista, que simula una red con ONTs garantizadas y no garantizadas pertenecientes a usuarios de operadores tradicionales y virtuales. El estudio de este escenario nos ha llevado a la aplicación de métodos más complejos para garantizar el correcto aprendizaje del modelo.

Por otro lado, se ha empleado la herramienta de optimización *Optuna*, que ha permitido obtener mejores resultados en el aprendizaje del agente DRL, optimizando los hiperparámetros que marcan el comportamiento de este. Además, su uso nos ha permitido

ver como la modificación de estos parámetros influye sobre el aprendizaje, dándonos una visión más profunda del funcionamiento del agente.

Los resultados obtenidos confirman la eficacia de integrar técnicas de IA en 10G-EPON mediante un agente DRL entrenado con PPO, evidenciando su capacidad para adaptarse con éxito a diversos escenarios de red, incluso bajo patrones dinámicos de tráfico y condiciones de calidad de servicio. Estos hallazgos ponen de relieve el gran potencial de la propuesta y abren la puerta a futuras mejoras y aplicaciones en entornos más complejos

5.2 Líneas futuras

Ente trabajo abre la puerta a muchas vías de investigación enfocadas en la mejora del simulador. Una de las mejoras consiste en realizar una optimización mucho más profunda de los hiperparámetros mediante *Optuna*. Para ello será necesario ejecutarlo en servidores con mayor capacidad de cómputo, ya que en este trabajo se ha trabajado bajo restricciones de recursos computacionales.

Por otro lado, también relacionado con la optimización, se ha realizado un estudio preliminar, aunque limitado, para buscar los pesos óptimos de la función *reward* en los diferentes casos de estudio. Este es un problema complejo que se basaría en, una vez conocido el resultado deseado, analizar entre cientos de combinaciones posibles para observar cual de estas permite alcanzar este resultado de la forma más eficiente.

Otra línea futura interesante, sería estudiar cómo opera el simulador empleando algoritmos distintos al PPO. Además, sería conveniente analizar en profundidad el escenario 4, enfocándonos en porqué en casos complejos como este, es necesario discretizar el espacio de acciones para su correcto funcionamiento en lugar de emplear un espacio continuo.

Asimismo, es posible desarrollar escenarios aún más realistas que los estudiados en este trabajo, incorporando nuevas características, como la consideración en el sistema DRL de parámetros adicionales —por ejemplo, el retardo medio de perfiles o de servicios prioritarios—, y ampliando el análisis a redes EPON de mayor capacidad, como 25G o 50G PON.

6 Bibliografía

[1] "IEEE Standard for Information technology-- Local and metropolitan area networks-- Specific requirements-- Part 3: CSMA/CD Access Method and Physical Layer Specifications Amendment 1: Physical Layer Specifications and Management Parameters for 10 Gb/s Passive Optical Networks," Sep. 11, 2009, *IEEE, Piscataway, NJ, USA*. doi: 10.1109/IEEESTD.2009.5294950.

[2] R. S. . Sutton and A. G. . Barto, *Reinforcement learning : an introduction*. The MIT Press, 2020.

[3] A. Plaat, *Deep Reinforcement Learning*. Singapore: Springer Nature Singapore, 2022. doi: 10.1007/978-981-19-0638-1.

[4] T. Akiba, S. Sano, T. Yanase, T. Ohta, and M. Koyama, "Optuna: A next-generation hyperparameter optimization framework." [Online]. Available: https://optuna.org/

[5] Python Software Foundation, "Python." [Online]. Available: https://www.python.org/

[6] Microsoft, "Visual Studio: IDE y editor de código para desarrolladores de software y equipos." [Online]. Available: https://visualstudio.microsoft.com/es/

[7] Farama Foundation, "Gymnasium Documentation." [Online]. Available: https://gymnasium.farama.org/introduction/basic_usage/

[8] David Pérez Moreno, "Aprendizaje por refuerzo en redes ópticas pasivas (PON)," Trabajo Fin de Grado, Universidad de Burgos Escuela Politécnica Superior, Burgos, 2024.

[9] Microsoft, "Microsoft Teams: software de chat en grupo." [Online]. Available: https://www.microsoft.com/es-es/microsoft-teams/group-chat-software

[10] C. F. Lam, *Passive optical networks : principles and practice*. Amsterdam ; Elsevier/Academic Press, 2007.

[11] S. VIAVI Solutions, "VIAVI Solutions, '¿Qué es una red óptica pasiva (PON)?'"

[12] Jose María Robledo Sáez, "Implementación de un simulador de redes de acceso pasivas en OMNeT++ ," Proyecto Fin de Carrera, E.T.S.I. de Telecomunicación, Universidad de Valladolid., Valladolid, 2012.

[13] "IEEE Standard for Information technology-- Local and metropolitan area networks-- Part 3: CSMA/CD Access Method and Physical Layer Specifications Amendment: Media Access Control Parameters, Physical Layers, and Management Parameters for Subscriber Access Networks," Jun. 24, 2004, *IEEE, Piscataway, NJ, USA*. doi: 10.1109/IEEESTD.2004.94617.

[14] G. Kramer and G. Pesavento, "Ethernet passive optical network (EPON): building a next-generation optical access network," *IEEE Communications Magazine*, vol. 40, no. 2, pp. 66–73, 2002, doi: 10.1109/35.983910.

[15] Dexiang John Xu, Wei Yen, and E. Ho, "Proposal of a new protection mechanism for ATM PON interface," in *ICC 2001. IEEE International Conference on Communications. Conference Record (Cat. No.01CH37240)*, IEEE, pp. 2160–2165. doi: 10.1109/ICC.2001.937039.

[16] F. Selmanovic and E. Skaljo, "GPON in Telecommunication Network," in *International Congress on Ultra Modern Telecommunications and Control Systems*, IEEE, Oct. 2010, pp. 1012–1016. doi: 10.1109/ICUMT.2010.5676500.

[17] "IEEE Standard for EthernetAmendment 1: Physical Layer Specifications and Management Parameters for Extended Ethernet Passive Optical Networks," Aug. 23, 2013, *IEEE, Piscataway, NJ, USA*. doi: 10.1109/IEEESTD.2013.6587259.

[18] G. Kramer, B. Mukherjee, and G. Pesavento, "IPACT a dynamic protocol for an Ethernet PON (EPON)," *IEEE Communications Magazine*, vol. 40, no. 2, pp. 74–80, 2002, doi: 10.1109/35.983911.

[19] C. Albarrán, "Qué es un Operador Móvil Virtual (OMV), cómo funcionan y cuál elegir en España," May 2024, *Redes & Telecom, Digital360 Iberia*. [Online]. Available: https://www.redestelecom.es/especiales/que-es-un-omv-como-funcionan-y-cual-elegir-en-espana/

[20] Y. Fernández, "OMV: qué es y en qué se diferencia de los operadores tradicionales," Jan. 2021, *Xataka, Webedia*. [Online]. Available: https://www.xataka.com/basics/omv-que-que-se-diferencia-operadores-tradicionales

[21] R. Ozalp, A. Ucar, and C. Guzelis, "Advancements in Deep Reinforcement Learning and Inverse Reinforcement Learning for Robotic Manipulation: Toward Trustworthy, Interpretable, and Explainable Artificial Intelligence," *IEEE Access*, vol. 12, pp. 51840–51858, 2024, doi: 10.1109/ACCESS.2024.3385426.

[22] G. de la Cruz, Y. Du, and M. Taylor, "Pre-training Neural Networks with Human Demonstrations for Deep Reinforcement Learning. ," 2018.

[23] K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Bharath, "Deep Reinforcement Learning: A Brief Survey," *IEEE Signal Process Mag*, vol. 34, no. 6, pp. 26–38, Nov. 2017, doi: 10.1109/MSP.2017.2743240.

[24] S. Gu, T. Lillicrap, Z. Ghahramani, R. E. Turner, B. Schölkopf, and S. Levine, "Interpolated policy gradient: merging on-policy and off-policy gradient estimation for deep reinforcement learning," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, in NIPS'17. Red Hook, NY, USA: Curran Associates Inc., 2017, pp. 3849–3858.

[25] R. Devidze, "Reward Design for Reinforcement Learning Agents," Mar. 2025.

[26] C. J. C. H. Watkins and P. Dayan, "Q-learning," *Mach Learn*, vol. 8, no. 3–4, pp. 279–292, May 1992, doi: 10.1007/BF00992698.

[27] Q. Huang, "Model-Based or Model-Free, a Review of Approaches in Reinforcement Learning," in *2020 International Conference on Computing and Data Science (CDS)*, IEEE, Aug. 2020, pp. 219–221. doi: 10.1109/CDS49703.2020.00051.

[28] J. Clifton and E. Laber, “Q-Learning: Theory and Applications,” *Annu Rev Stat Appl*, vol. 7, no. 1, pp. 279–301, Mar. 2020, doi: 10.1146/annurev-statistics-031219-041220.

[29] M. Roderick, J. MacGlashan, and S. Tellex, “Implementing the Deep Q-Network,” Nov. 2017.

[30] R. G. . McClarren, *Machine learning for engineers : using data to solve problems for physical systems*. Springer, 2021.

[31] J. Peters and S. Schaal, “Policy Gradient Methods for Robotics,” in *2006 IEEE/RSJ International Conference on Intelligent Robots and Systems*, IEEE, Oct. 2006, pp. 2219–2225. doi: 10.1109/IROS.2006.282564.

[32] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, “Proximal Policy Optimization Algorithms,” Aug. 2017.

[33] Milvus, “What are the advantages of deep reinforcement learning over traditional methods?,” 2025. [Online]. Available: https://milvus.io/ai-quick-reference/what-are-the-advantages-of-deep-reinforcement-learning-over-traditional-methods

[34] H. Mao, M. Alizadeh, I. Menache, and S. Kandula, “Resource Management with Deep Reinforcement Learning,” in *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, New York, NY, USA: ACM, Nov. 2016, pp. 50–56. doi: 10.1145/3005745.3005750.

[35] Clara Ruiz de las Heras, “Repositorio de GitHub.” Accessed: Sep. 21, 2025. [Online]. Available: https://github.com/claritaruizheras/Redes_opticas_escenarios

[36] Chongtao Guo, Min Sheng, Yan Zhang, and Xijun Wang, “A Jain’s Index Perspective on α-Fairness Resource Allocation over Slow Fading Channels,” *IEEE Communications Letters*, vol. 17, no. 4, pp. 705–708, Apr. 2013, doi: 10.1109/LCOMM.2013.021913.130025.

[37] C. Kim, T.-W. Yoo, and B.-T. Kim, “An Improved Cyclic Water-Filling EPON DBA Providing Priority Scheduling and Faster Allocation Time,” in *The 9th*

International Conference on Advanced Communication Technology, IEEE, Feb. 2007, pp. 1391–1396. doi: 10.1109/ICACT.2007.358616.

[38] J. Zhang and N. Ansari, "Dynamic Time Allocation and Wavelength Assignment in Next Generation Multi-Rate Multi-Wavelength Passive Optical Networks," in *2010 IEEE International Conference on Communications*, IEEE, May 2010, pp. 1–5. doi: 10.1109/ICC.2010.5502243.

[39] A. Raffin, A. Hill, A. Gleave, A. Kanervisto, M. Ernestus, and N. Dormann, "PPO — Stable Baselines3." [Online]. Available: https://stable-baselines3.readthedocs.io/en/master/modules/ppo.html#parameters

[40] B. C. Arnold, "Pareto Distribution," in *Wiley StatsRef: Statistics Reference Online*, Wiley, 2015, pp. 1–10. doi: 10.1002/9781118445112.stat01100.pub2.

[41] G. ODE, "Réquiem por la Evaluación del Desempeño: un análisis post-mortem," 2016, *ODE, Barcelona, España*. [Online]. Available: http://www.ode.es/Art_Requiem_Eva_Desempe/requiem.htm

[42] V. Alarcón-Aquino, L. G. Guerrero-Ojeda, J. Rodríguez-Asomoza, and R. Rosas-Romero, "Análisis de Tráfico Auto-similar en Redes de Comunicaciones Usando Onditas (Wavelets)," *Información tecnológica*, vol. 16, no. 2, 2005, doi: 10.4067/S0718-07642005000200010.

[43] Optuna, "Getting Started — Optuna Dashboard documentation." Accessed: Sep. 09, 2025. [Online]. Available: https://optuna-dashboard.readthedocs.io/en/latest/getting-started.html

[44] Ethernet Alliance, "Overview of 10G-EPON," 2011, [Online]. Available: https://www.ethernetalliance.org/wp-content/uploads/2011/10/10GEPON_WP_EA_from-FC_Final_updated_V2d4.pdf