UNIVERSIDAD DE MÁLAGA

Escuela de Ingenierías Industriales

TRABAJO DE FIN DE GRADO

Departamento de Lenguajes y Ciencias de la Computación

Grado en Ingeniería en Tecnologías Industriales

Mención en Itinerario de Organización

Aprendizaje por refuerzo profundo aplicado a datos obtenidos de sensores de IA desarrollados para la industria de defensa

---

Realizado por:

Teodoro Robles de Molina

Tutorizado por:

María del Carmen Aranda Garrido

MÁLAGA, JUNIO DE 2025

A mis abuelos.

Resumen

El aprendizaje por refuerzo se ha consolidado como una técnica fundamental en el desarrollo de modelos de inteligencia artificial actuales. Su capacidad para optimizar decisiones en entornos dinámicos lo convierte en una herramienta especialmente valiosa en aplicaciones industriales, donde la interacción continua con el medio y la maximización del rendimiento son elementos clave.

Se basa en un proceso de prueba y error mediante el cual un agente interactúa con su entorno para lograr un objetivo. A partir de la observación del estado de lo que lo rodea, toma decisiones ejecutando acciones y recibe una recompensa que indica el nivel de acierto. Con el tiempo, este proceso permite ajustar su comportamiento para maximizar la recompensa obtenida. Si a este análisis le incluimos redes neuronales obtenemos aprendizaje por refuerzo profundo.

En este Trabajo de Fin de Grado se desarrolla un modelo de aprendizaje por refuerzo profundo aplicado al tratamiento de señales obtenidas mediante sensores utilizados en la industria de la defensa. El conjunto de datos que se utiliza para la realización del modelo ha sido extraído de la plataforma Kaggle. A partir de esta base, se realiza un análisis exhaustivo del comportamiento del sistema, con especial atención a la definición de la recompensa y su impacto en el rendimiento del modelo.

Este trabajo se centra en varios aspectos clave: en primer lugar, se realiza un análisis exploratorio de los datos extraídos del repositorio; posteriormente, se analiza el rendimiento de los mismo con redes neuronales, se diseña el modelo de aprendizaje profundo y se configura el agente de aprendizaje por refuerzo. Todo ello con el objetivo de determinar qué configuración de recompensa resulta más eficaz para optimizar el rendimiento del agente.

Abstract

Reinforcement learning has become a fundamental technique in the development of current artificial intelligence models. Its ability to optimize decision-making in dynamic environments makes it a particularly valuable tool in industrial applications, where continuous interaction with the environment and performance maximization are key elements.

It is based on a trial-and-error process in which an agent interacts with its environment to achieve a goal. Based on the observation of the surrounding state, it makes decisions by executing actions and receives a reward that indicates the level of success. Over time, this process allows the agent to adjust its behavior to maximize the reward obtained. When neural networks are incorporated into this analysis, it becomes deep reinforcement learning.

This Bachelor's Thesis develops a deep reinforcement learning model applied to the processing of signals obtained from sensors used in the defense industry. The dataset used for building the model was obtained from the Kaggle platform. Based on this dataset, a thorough analysis of the system's behavior is carried out, with special attention given to the definition of the reward and its impact on the model's performance.

This work focuses on several key aspects: first, an exploratory data analysis is conducted using the data extracted from the repository; then, the performance of this data is analyzed through neural networks, a deep learning model is designed, and a reinforcement learning agent is configured. All of this is done with the aim of determining which reward configuration is most effective for optimizing the agent's performance.

ÍNDICE

Índice

Índice de figuras

INTRODUCCIÓN

CAPÍTULO 1

1. Introducción

1.1. Motivación y contexto

Los conflictos recientes han puesto de manifiesto los riesgos crecientes a los que se enfrentan las fuerzas armadas, obligándolas a adaptarse con rapidez a nuevas amenazas y al avance constante de la tecnología. En este contexto, los sensores se han consolidado como herramientas fundamentales en operaciones militares, al proporcionar información en tiempo real que mejora significativamente la capacidad de decisión. La incorporación de la inteligencia artificial (IA) en estos sistemas representa un paso más en esta evolución, al permitir optimizar los procesos de análisis y apoyar decisiones más precisas y eficientes.

La inteligencia artificial se ha consolidado como una herramienta fundamental para afrontar los desafíos actuales en múltiples sectores. Su aplicación permite automatizar procesos, optimizar el uso de recursos, mejorar la calidad y la productividad, personalizar productos y servicios, y tomar decisiones basadas en datos de forma más eficiente. Dentro del ámbito de la IA y el aprendizaje automático, destacan el aprendizaje por refuerzo (Reinforcement Learning) y el aprendizaje profundo (Deep Learning) como dos de sus subcampos con mayor proyección.

Estas técnicas han comenzado a integrarse en sistemas sensoriales avanzados, especialmente en entornos complejos como el sector de la defensa, donde los sensores con IA permiten analizar información en tiempo real y adaptarse de forma autónoma a situaciones cambiantes. En particular, el uso del aprendizaje por refuerzo permite que estos sensores optimicen su comportamiento a través de la experiencia, mejorando progresivamente su capacidad de respuesta y toma de decisiones.

En este trabajo se propone el uso del aprendizaje por refuerzo para maximizar la recompensa generada por sensores militares dotados de inteligencia artificial. Para ello, se utilizará un conjunto de datos que contiene distintos tipos de información, el cual será analizado y procesado adecuadamente. El objetivo principal es aplicar las técnicas mencionadas anteriormente para desarrollar agentes inteligentes capaces de aprender estrategias óptimas en contextos propios de la industria de defensa, mejorando así el rendimiento y la autonomía de los sistemas sensoriales.

1.2. Objetivos y alcance

El objetivo principal de este trabajo es desarrollar un agente inteligente que, mediante el uso de aprendizaje por refuerzo profundo, sea capaz de aprender estrategias óptimas para maximizar la recompensa generada por un sensor con inteligencia artificial en un entorno simulado. Además, se explorarán distintas técnicas y variaciones para optimizar el rendimiento del agente y se llevará a cabo una evaluación de los resultados obtenidos.

Para la implementación de este proyecto se han utilizado los siguientes recursos:

- Kaggle: plataforma desde la cual se ha obtenido el conjunto de datos y de donde se ha tomado como referencia el código base utilizado para el desarrollo del proyecto.
- Overleaf (LaTeX): editor en línea utilizado para la redacción y compilación de la memoria del Trabajo de Fin de Grado, empleando el lenguaje de marcado LaTeX.
- Anaconda Navigator y Google Colab – Jupyter Notebook: entornos de desarrollo interactivo utilizado para la implementación del código en Python.
  - pandas: biblioteca utilizada para la manipulación y análisis de datos estructurados, especialmente útil para trabajar con estructuras tipo tabla (DataFrames).
  - numpy: proporciona soporte para arreglos multidimensionales y funciones matemáticas de alto nivel.
  - matplotlib y seaborn: bibliotecas para la generación de gráficos estáticos, ideal para visualizar datos en forma de líneas, barras, dispersión, entre otros.
  - scikit-learn y scikit-multilearn: conjunto de herramientas para aprendizaje automático en Python, utilizado para clasificación, regresión, clustering y reducción de dimensionalidad.
  - tensorflow: biblioteca desarrollada por Google para la construcción y entrenamiento de modelos de aprendizaje profundo mediante grafos computacionales.
  - gym: entorno de simulación desarrollado por OpenAI para entrenar y evaluar algoritmos de aprendizaje por refuerzo.
  - xgboost: algoritmo de aprendizaje automático basado en árboles de decisión, muy eficiente y utilizado en tareas de clasificación y regresión.
  - shap: herramienta que permite interpretar modelos de aprendizaje automático mediante el análisis de la contribución de cada variable a las predicciones.
  - kagglehub: biblioteca utilizada para descargar y cargar datasets o modelos directamente desde la plataforma Kaggle dentro de un entorno de ejecución.

1.3. Metodología general del trabajo

Para el desarrollo de este Trabajo de Fin de Grado se ha seguido una metodología de trabajo iterativa incremental, basada en la mejora progresiva de prototipos a lo largo del proceso. Esta estrategia ha permitido evaluar el rendimiento del modelo en distintas fases del proyecto, incorporando cambios y ajustes según los resultados obtenidos y las necesidades detectadas.

El proyecto se ha desarrollado en varias etapas diferenciadas:

1. Elección del conjunto de datos: se seleccionó un dataset realista relacionado con sensores de IA aplicados a la industria y defensa.

2. Estudio previo: se realizó una revisión teórica del aprendizaje por refuerzo, redes neuronales profundas y del lenguaje Python, así como de las librerías necesarias para el desarrollo.

3. Análisis exploratorio de datos: se exploraron y depuraron los datos, identificando patrones y variables influyentes en la recompensa.

4. Diseño del modelo inicial: se implementó un primer prototipo del sistema basado en aprendizaje por refuerzo profundo.

5. Iteraciones de ajuste: el modelo fue modificado en varias ocasiones cambiando hiperparámetros, funciones de activación, optimizadores y variables de entrada.

6. Evaluación y obtención de resultados: se analizaron gráficamente los resultados de cada configuración para comparar el rendimiento del agente y extraer conclusiones, con el posterior desarrollo de la memoria.

Gracias a este enfoque iterativo, ha sido posible mejorar progresivamente la calidad del modelo hasta obtener una versión final funcional y coherente con los objetivos planteados.

1.4. Estructura del documento

En este apartado se detalla brevemente cómo está organizado este documento y cuáles son las secciones que lo estructuran, con el objetivo de facilitar su comprensión y ofrecer una visión general del desarrollo del proyecto.

1. Introducción: Se presenta el objetivo general del proyecto, el contexto de los sensores analizados y la aplicación de la inteligencia artificial en dicho ámbito, con especial atención a las técnicas utilizadas como el aprendizaje por refuerzo.

2. Sensores IA: Se detalla el tipo de sensores que encontramos en el conjunto de datos y su integración con la inteligencia artificial

3. Aprendizaje por refuerzo: Se explica la técnica que utilizan estos sensores, aprendizaje por refuerzo profundo, y todas las características de este área del aprendizaje automático, desde su contexto histórico hasta los diferentes algoritmos de aprendizaje.

4. Herramientas y entorno de desarrollo: Los módulos principales del código utilizado para la implementación del modelo.

5. Conjunto de datos (dataset): Se expone el conjunto de datos utilizado en el análisis.

6. Desarrollo del modelo: Se explicará detalladamente, el procedimiento y toda la arquitectura de la red neuronal, así como el entorno y de los componentes del agente de aprendizaje por refuerzo y el proceso de entrenamiento llevado a cabo.

7. Análisis de resultados: Se comentan los resultados obtenidos a partir del análisis del modelo entrenado.

8. Conclusiones y líneas futuras: Se presenta una reflexión crítica y las conclusiones derivadas del estudio y se exponen nuevos enfoques y mejoras del modelo, así como aplicaciones industriales y militares futuras.

9. Anexo y Bibliografía: Se referencia como material adicional que complementa el contenido de la memoria además el código utilizado.

10. Bibliografía: Se incluyen tanto las referencias bibliográficas utilizadas a lo largo del trabajo

SENSORES IA

CAPÍTULO 2

2. Sensores IA

2.1. Introducción

Antes de abordar el análisis de los sensores utilizados en este trabajo, resulta necesario definir qué se entiende por sensor militar y por sensor basado en inteligencia artificial.

Un sensor militar es un dispositivo o sistema especializado en la detección, seguimiento y recopilación de información sobre distintos aspectos del entorno en operaciones de combate. Estos sensores pueden ser de diversos tipos, como detectores de radar, infrarrojos, acústicos o químicos, y desempeñan un papel fundamental en tareas de vigilancia, identificación de amenazas y mejora de la conciencia situacional. Gracias a su capacidad para proporcionar datos precisos y en tiempo real, permiten a las fuerzas militares adoptar decisiones informadas y aumentar la eficacia de sus operaciones.

2.2. Tipos de sensores utilizados

En muchos sistemas de defensa modernos se emplean sensores capaces de registrar información del entorno utilizando tecnologías como el radar, el sonar o la detección por infrarrojos. A continuación, se presenta una breve descripción de cada uno de estos tipos de sensores, destacando sus principales características y su función dentro de sistemas de vigilancia, seguimiento y toma de decisiones.

2.2.1. Sensor de radar

Un sensor radar militar, Figura 1, es un sistema electromagnético activo que emite ondas de radio y analiza su reflejo en objetos para detectar, localizar, identificar y rastrear amenazas o blancos en distintos entornos operativos. Su uso es fundamental en defensa y vigilancia, ya que permite obtener información precisa sobre la posición, velocidad y trayectoria de aeronaves, misiles, vehículos, buques y otros objetivos, incluso en condiciones adversas como mal tiempo o baja visibilidad.

El funcionamiento básico es el siguiente: primero, el radar emite pulsos de radiofrecuencia, es decir, ondas electromagnéticas, en una dirección específica. Estas ondas alcanzan un objeto y parte de la señal es reflejada de vuelta. Un procesador analiza el tiempo de retorno, la intensidad y el efecto Doppler [1] para determinar la distancia, velocidad y dirección del objeto.

[1]Efecto Doppler: es un fenómeno físico en el que la frecuencia de una onda cambia debido al movimiento relativo entre la fuente emisora y el receptor. Si el objeto se acerca al emisor, las ondas se comprimen, aumentando la frecuencia percibida (desplazamiento al azul en ondas electromagnéticas). Si el objeto se aleja, las ondas se expanden, disminuyendo la frecuencia percibida (desplazamiento al rojo en ondas electromagnéticas).

Figura 1: Sensor de radar

2.2.2. Sensor de sonar

El sonar, acrónimo de *Sound Navigation and Ranging*, es una tecnología que utiliza ondas sonoras para detectar, localizar y analizar objetos en medios como el agua, Figura 2. En el ámbito militar, el sonar es esencial para operaciones de vigilancia, guerra antisubmarina y navegación segura.

El funcionamiento del sonar se basa en la emisión y recepción de ondas sonoras. Un transmisor genera pulsos acústicos que viajan a través del agua. Cuando estas ondas encuentran un objeto, parte de la señal se refleja y regresa al receptor, que analiza la información para determinar la ubicación, tamaño y naturaleza del objeto detectado. El tiempo que tarda la señal en viajar hasta el objeto y regresar permite calcular la distancia, mientras que la intensidad y frecuencia de la onda reflejada proporcionan detalles sobre el material y la forma del objeto.

Figura 2: Sensor de sonar

2.2.3. Sensor infrarrojo

Los sensores infrarrojos militares son dispositivos que detectan la radiación infrarroja emitida por objetos y seres vivos, Figura 3, permitiendo identificar y rastrear objetivos en condiciones de baja visibilidad o en completa oscuridad. Estos sensores son fundamentales en aplicaciones de defensa y seguridad, proporcionando una ventaja táctica en operaciones nocturnas y en entornos con visibilidad limitada.

Todos los cuerpos emiten energía en forma de radiación infrarroja debido a su temperatura. Los sensores infrarrojos detectan esta radiación y la convierten en señales eléctricas. Estas señales son posteriormente procesadas para crear imágenes térmicas o para activar sistemas de alerta. Esto permite visualizar diferencias de temperatura y detectar la presencia de objetos o personas.

Figura 3: Sensor infrarrojo

2.3. Integración de IA en sensores

Los sensores constituyen la piedra angular de muchos sistemas de inteligencia artificial, actuando como los ojos y oídos de las máquinas al recopilar los datos que posteriormente serán procesados y analizados. No se limitan únicamente a la captación de señales, sino que también están diseñados para procesar información en bruto, identificar patrones y dotar de significado a los datos que recogen. Por ejemplo, un sensor integrado en una cámara inteligente puede no solo detectar un rostro en una multitud, sino también reconocerlo e incluso estimar características como la edad, el género o el estado emocional de la persona observada.

Los sensores inteligentes son una técnica desarrollada en la década de 1970, cuando las capacidades de procesamiento —basadas en sistemas de lectura integrados con procesamiento de señal— aún estaban lejos de alcanzar la complejidad necesaria en los sistemas avanzados de vigilancia y alerta por infrarrojos, debido a la enorme cantidad de ruido y señales no deseadas emitidas por el entorno operativo, especialmente en aplicaciones militares. Esta tecnología permaneció restringida a un entorno militar cerrado, hasta que

en la década de 1990 experimentó un auge en aplicaciones y prestaciones gracias a los avances significativos en la lectura y procesamiento integrado de señales, posibilitados por las tecnologías CCD-CMOS en matrices de detectores de plano focal (FPA). De hecho, el rápido desarrollo de la tecnología de procesamiento de "integración a muy gran escala" (VLSI) y de matrices de detectores electroópticos permitió crear nuevas generaciones de sensores inteligentes con procesamiento de señal altamente mejorado, mediante la integración de microcomputadores y otros procesadores de señal VLSI dentro de la propia estructura del sensor. Esto permitió lograr funciones básicas propias del ojo humano, como la mirada dinámica, la compensación de no uniformidad, y el filtrado espacial y temporal [5].

En general, un sensor inteligente tiene tres componentes principales: un sensor que captura datos del entorno o del equipo; un microprocesador, que calcula la salida del sensor mediante programación; y recursos de comunicación que informan a la salida del microprocesador de la acción a tomar. Al capturar datos relevantes, los sensores convierten la información analógica en señales digitales para facilitar el procesamiento por parte de dispositivos electrónicos. La comunicación inalámbrica, utilizando tecnologías como redes celulares, Wi-Fi, Bluetooth, LoRa, permite la transmisión efectiva de estos datos, contribuyendo a la conectividad en la red IoT.

La inteligencia incorporada en los sensores va más allá de la mera recopilación de datos, incorporando algoritmos avanzados e incluso técnicas de aprendizaje automático para adaptarse a patrones y mejorar su precisión con el tiempo.

Un ejemplo notable de esta evolución lo constituyen los sensores de imagen con inteligencia artificial incorporada. En estos dispositivos, la señal capturada por el chip de píxeles se procesa directamente mediante algoritmos de IA dentro del propio sensor, eliminando la necesidad de procesadores externos de alto rendimiento o memorias auxiliares. Este enfoque, conocido como Edge AI, permite el desarrollo de sistemas inteligentes más rápidos, autónomos y con menor consumo energético, al mismo tiempo que reduce la emisión térmica del dispositivo.

Estos sensores generan metadatos, información semántica derivada de la imagen, como la que podemos observar en la Figura 4 en lugar de almacenar la imagen completa, lo que no solo acelera el procesamiento de seguimiento de objetos, sino que también disminuye el volumen de datos manejados y mejora la seguridad al minimizar los riesgos de privacidad asociados a la captura de imágenes.

Figura 4: Sensor con IA incorporada

En este contexto, los sensores de imagen de nueva generación, integran un procesador digital de señales (DSP) dedicado al procesamiento IA y memoria específica para los modelos de aprendizaje automático. La información extraída se puede exportar en diversos formatos, ya sea en imágenes completas (YUV/RGB) o en áreas específicas (Regiones de Interés, ROI), optimizando el uso de recursos según las necesidades de la aplicación. Además, gracias a la realización de tareas de procesamiento y análisis directamente en el sensor, se logra un seguimiento de objetos de alta precisión en tiempo real durante la grabación de vídeo, superando las limitaciones tradicionales asociadas a la transmisión masiva de datos entre dispositivos [7].

2.3.1. Sensores IA en ámbito de la defensa

En la actualidad, las aplicaciones de inteligencia artificial en el ámbito militar están revolucionando la forma en que se llevan a cabo las operaciones y estrategias de defensa. Desde el uso de algoritmos avanzados para predecir movimientos enemigos hasta la implementación de drones autónomos, Figura 5, la IA está potenciando la eficacia y precisión de las fuerzas armadas.

El ejército utiliza la inteligencia artificial de manera estratégica en el campo de batalla, aprovechando los datos en tiempo real recolectados por diversos sensores para monitorear la posición, estado físico, emocional y mental de los soldados, así como sus capacidades para enfrentar diferentes situaciones. Esta tecnología permite tomar decisiones más eficientes y precisas en tiempo real, mejorando la efectividad y seguridad de las operaciones militares [8].

La IA puede jugar un papel crítico en la automatización de tareas subordinadas, como la coordinación de enjambres de drones y otras plataformas no tripuladas. Esto libera recursos humanos para centrarse en operaciones más complejas y permitiendo así una mayor atención a la toma de decisiones.

Sin embargo, hay que tener en cuenta las limitaciones que pueden llegar a tener estos sistemas, cuanto a precisión, debido a la calidad de los datos a interpretar. Por eso, hoy en día, en este ámbito la IA está más del lado de un complemento al ser humano que un sustituto. La interacción entre ser humano e IA será un factor clave para el éxito de las misiones.

Figura 5: Enjambre de drones militares controlados con IA

APRENDIZAJE POR REFUERZO

CAPÍTULO 3

3. Aprendizaje por refuerzo

3.1. Contexto histórico

La historia del aprendizaje por refuerzo se compone de dos hilos principales que se desarrollaron de manera independiente antes de converger en el enfoque moderno del aprendizaje por refuerzo. Uno de estos hilos se origina en el concepto de "aprendizaje por prueba y error", que tuvo sus raíces en la psicología del aprendizaje animal. El segundo hilo se relaciona con el "control óptimo" y su solución mediante el uso de funciones de valor y programación dinámica. Estos acontecimientos históricos se entrelazan con los primeros avances de la IA[2].

El término "control óptimo" comenzó a utilizarse en la década de 1950 para describir el desafío de diseñar un controlador que hiciera que un sistema se comportara de la mejor manera posible, buscando siempre minimizar algún tipo de medida (como el error, el gasto de energía o el tiempo) a lo largo de su funcionamiento. Una de las aproximaciones clave para abordar este problema se desarrolló a mediados de la década de 1950, gracias a Richard Bellman y otros, que expandieron la teoría del siglo XIX de Hamilton y Jacobi. Este enfoque introdujo conceptos fundamentales como el "estado" de un sistema dinámico y la "función de valor" o "función de retorno óptimo", lo que condujo a la formulación de la conocida ecuación de Bellman, ecuación (1).

$$V(s) = \mathbb{E}\left[R(s,a) + \gamma \sum_{t=1}^{\infty} \gamma^{t-1} R(s_t)\right] \tag{1}$$

- $V(s)$: Valor del estado s. determinada política.
- $\mathbb{E}[\cdot]$: Esperanza matemática.
- $R(s,a)$: Recompensa inmediata obtenida al ejecutar la acción a en el estado s.
- γ: Factor de descuento. Es un parámetro entre 0 y 1 que determina la importancia de las recompensas futuras.
- $\sum_{t=1}^{\infty} \gamma^{t-1} R(s_t)$: Suma de las recompensas futuras descontadas desde el tiempo $t = 1$ en adelante.
- $R(s_t)$: Recompensa recibida al estar en el estado s_t en el instante t.

Los métodos que surgieron para resolver problemas de control óptimo a través de esta ecuación se denominaron "programación dinámica." Estos conceptos, y otros como: "procesos de decisión markovianos" (MDP) o "iteración de políticas", son fundamentales en la teoría y los algoritmos modernos del aprendizaje por refuerzo.

[2]El apartado 3.1. se basa íntegramente en la obra de Sutton y Barto [13].

La conexión entre el control óptimo, la programación dinámica y el aprendizaje por refuerzo no fue inmediata y tardó en reconocerse. Uno de los primeros intentos de conectar el control óptimo y la programación dinámica con el aprendizaje fue realizado por Paul Werbos en 1987, quien propuso un enfoque aproximado llamado "programación dinámica heurística". Aunque, la verdadera integración de los métodos de programación dinámica con el aprendizaje en línea se materializó gracias al trabajo de Chris Watkins en 1989, gracias a su enfoque del aprendizaje por refuerzo utilizando el formalismo MDP.

En cuanto al concepto de "aprendizaje por prueba y error", el psicólogo estadounidense R. S. Woodworth fue uno de los primeros en explorar esta idea, que se remonta a la década de 1850. Edward Thorndike, por su parte, fue uno de los pioneros en expresar la esencia del aprendizaje por prueba y error como un principio de aprendizaje.

> *"De varias respuestas dadas ante una misma situación, aquellas que van acompañadas, o son seguidas de cerca, por una sensación de satisfacción para el animal estarán, en igualdad de condiciones, más firmemente asociadas a dicha situación, de modo que, cuando esta se repita, será más probable que esas respuestas vuelvan a producirse. Por el contrario, aquellas respuestas que vayan acompañadas, o seguidas de cerca, por una sensación de incomodidad tendrán, en igualdad de condiciones, una asociación más débil con esa situación, por lo que será menos probable que se repitan. Cuanto mayor sea la satisfacción o la incomodidad, mayor será el fortalecimiento o debilitamiento del vínculo.".*

– Edward Thorndike, 1911.

Estos y otros experimentos más como los realizados por B.F. Skinner, contribuyeron significativamente a una comprensión más profunda de cómo los organismos aprenden a través de la experimentación y la aplicación de refuerzos.

A principios del Siglo XX, el filósofo ruso Iván Pavlov, realizó otro experimento, denominado el perro de Pavlov, el cuál consistía en presentar a un perro inmediatamente, antes de la aparición del alimento, un estímulo condicionado, constituido por el sonido de un timbre. Esto hacía que el perro salivase cada vez que oía el timbre, es decir, comenzaba a manifestar un comportamiento que normalmente se emitía en respuesta a la aparición de la comida, aunque en esa prueba particular la comida no se le administraba. Los estudios de Iván Pavlov muestran que un estímulo que al principio no provoca ninguna reacción puede llegar a provocar una respuesta si se presenta varias veces junto con otro estímulo que sí genera esa reacción de forma natural. Siempre que ambos estímulos ocurran muy cerca en el tiempo, el primero acaba asociándose al segundo y es capaz, por sí solo, de generar la misma respuesta.

El "refuerzo" consiste en hacer más fuerte una conducta cuando un animal recibe un estímulo agradable (llamado reforzador) justo después de haber hecho algo o de haber ocurrido otro estímulo. Si ese refuerzo se da en el momento adecuado, el animal aprende que esa conducta tiene consecuencias positivas y es más probable que la repita.

La idea de implementar el aprendizaje por prueba y error en una computadora apareció entre los primeros pensamientos sobre la posibilidad de la inteligencia artificial. En un informe de 1948, Alan Turing describió un diseño para un "sistema de placer-dolor" que funcionaba de acuerdo con la Ley del efecto:

> *"Cuando se alcanza una configuración para la cual la acción es indeterminada, se hace una selección aleatoria de los datos que faltan y se hace la entrada apropiada en la descripción, tentativamente, y se aplica. Cuando ocurre un estímulo de dolor, todas las entradas tentativas se cancelan, y cuando ocurre un estímulo de placer, todas se vuelven permanentes."*

– Alan Turing, 1948.

Alan Turing, aunque planteó la premisa inicial, no desarrolló en profundidad este concepto. No fue hasta el año 1954, año de su muerte, que Wesley Clark y Belmont Farley realizaron simulaciones de aprendizaje por refuerzo en una red neuronal dentro de una computadora digital. Al mismo tiempo, Marvin Minsky presentó una red neuronal de aprendizaje por refuerzo analógica en su tesis doctoral en Princeton.

Sin embargo, es importante mencionar que incluso antes de 1954, se habían concebido ingeniosos dispositivos de aprendizaje por refuerzo, aunque estos eran de naturaleza electromecánica en lugar de informática. Un ejemplo de ello es el laberinto diseñado por Claude Shannon, Figura 6.

Un experimento que involucraba un ratón que navegaba por un laberinto denominado "Theseus", Figura 7. Este ratón empleaba un enfoque de prueba y error para encontrar su camino hacia una ubicación objetivo en el laberinto. La fascinante particularidad de este experimento reside en que, una vez que el ratón ha aprendido la ruta correcta hacia su destino, puede ser colocado en cualquier punto que haya visitado previamente durante sus exploraciones.

Figura 6: Laberinto de Shannon

Figura 7: Ratón Theseus

Su funcionamiento se basa en una serie de componentes y algoritmos que le permiten explorar el laberinto de manera metódica y resolver problemas complejos.

En su recorrido, el ratón utiliza sus sensores de cobre, en forma de bigotes, para detectar barreras y callejones sin salida. Si se encuentra con una barrera, el ratón Theseus retrocede, cambia de dirección y continúa su búsqueda hasta encontrar un camino abierto hacia la meta. Este proceso se repite de manera persistente hasta que se alcanza el objetivo del laberinto.

La verdadera innovación de este ratón y su laberinto reside en sus cuatro habilidades operativas clave. Theseus puede resolver problemas mediante el método de prueba y error, recordar soluciones previas y aplicarlas en momentos posteriores, incorporar nueva información a soluciones previamente recordadas y, por último, olvidar una solución para aprender una nueva cuando se presenta un problema diferente. Esta capacidad de adaptación y aprendizaje lo convierte en una herramienta valiosa para la resolución de problemas complejos.

En ese tiempo, Minsky exploró modelos computacionales de aprendizaje por refuerzo y dio origen a lo que denominó la "Calculadora analógica de refuerzo neuronal estocástico" (SNARC), la primera máquina de red neuronal artificial concebida. Con el uso de componentes analógicos y electromecánicos, construyeron una red neuronal de 40 neuronas. Cada neurona fue diseñada con un condensador para la memoria a corto plazo y un potenciómetro para la memoria a largo plazo.

En la década de 1960, los términos "refuerzo" y "aprendizaje por refuerzo" hicieron su debut en la literatura de ingeniería. Un artículo influyente en este contexto fue el de Minsky, titulado "Pasos hacia la inteligencia artificial", en el cual se abordaron diversos temas cruciales para el aprendizaje por refuerzo.

En los años 1961 y 1963, Donald Michie presentó un sistema de aprendizaje basado en la prueba y error que revolucionó la comprensión de cómo las máquinas pueden aprender y mejorar su desempeño en un juego tan clásico como el tres en raya (conocido también como TaTeTi o Tic Tac Toe), Figura 8.

Figura 8: MENACE (Matchbox Educable Noughts and Crosses Engine)

Este sistema, denominado MENACE (Matchbox Educable Noughts and Crosses Engine), consistía en un enfoque ingenioso que utilizaba cajas de fósforos para representar cada posible posición en el juego. Dentro de cada caja de fósforos se almacenaban fósforos de colores distintos, cada uno representando una opción de movimiento a partir de la posición correspondiente. Al seleccionar aleatoriamente un fósforo de la caja asociada a la posición actual del juego, MENACE tomaba su decisión de movimiento. Después de cada partida, se ajustaban las cantidades de fósforos en las cajas utilizadas durante el juego. Esta acción servía para reforzar las decisiones acertadas y castigar las decisiones equivocadas.

Este enfoque de aprendizaje basado en la retroalimentación y la adaptación continua sienta las bases para comprender cómo las máquinas pueden mejorar sus habilidades a través de la experiencia y la toma de decisiones autónoma.

Un hito temprano se produjo en 1977, cuando Ian H. Witten propuso la primera regla de aprendizaje basada en diferencia temporal (TD), conocida actualmente como TD(0), marcando el inicio formal de este subcampo. Más adelante, en 1988, Richard S. Sutton dio un paso clave al separar el aprendizaje de diferencia temporal del control, proponiéndolo como un método general de predicción. También introdujo el algoritmo TD(λ), investigando sus propiedades de convergencia y utilidad práctica.

La integración definitiva entre los enfoques de control óptimo y aprendizaje por diferencia temporal llegó en 1989 con la creación del algoritmo Q-learning, desarrollado por Christopher J.C.H. Watkins. Este avance permitió que un agente pudiera aprender una política óptima sin necesidad de conocer el modelo del entorno.

El creciente interés por las redes neuronales en la década de 1980 impulsó el desarrollo del aprendizaje por refuerzo profundo. Un ejemplo destacado fue TD-Gammon, desarrollado por Gerald Tesauro en 1992, Figura 9. Este programa, sin conocimiento previo, aprendió a jugar al backgammon a nivel intermedio enfrentándose a sí mismo mediante el algoritmo TD(λ) y redes neuronales.

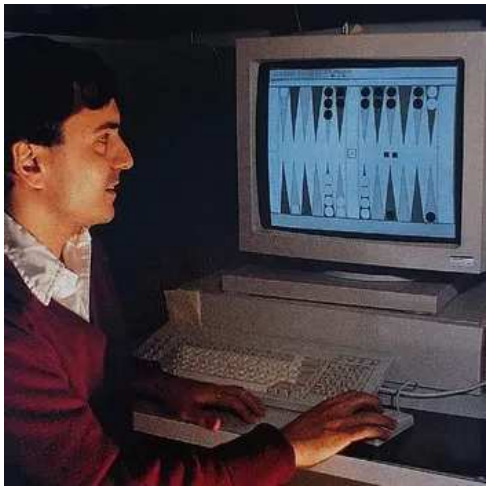

Figura 9: Gerry Tesauro utilizando su programa para jugar al backgammon

En 1996, la publicación del libro *Reinforcement Learning: An Introduction* por Richard S. Sutton y Andrew G. Barto consolidó la disciplina, convirtiéndose en una referencia fundamental tanto en la informática como en la neurociencia computacional. Ese mismo año, la supercomputadora Deep Blue derrotó al campeón mundial Garry Kasparov, un hito simbólico en la relación entre inteligencia artificial y capacidades humanas, aunque no directamente vinculado al aprendizaje por refuerzo.

A partir de 2012, con el auge del aprendizaje profundo, se revitalizó el interés por el uso de redes neuronales profundas como aproximadores de funciones dentro del aprendizaje por refuerzo. En 2013, DeepMind presentó avances notables en videojuegos de Atari mediante aprendizaje por refuerzo profundo (Deep RL), y en 2015 alcanzó un nuevo hito con AlphaGo, el primer programa en vencer a un jugador profesional de Go en un tablero completo, gracias a la combinación de redes neuronales profundas y aprendizaje por refuerzo.

El progreso continuó en años posteriores con desarrollos como Pluribus, un sistema capaz de derrotar a múltiples jugadores profesionales en póker, y OpenAI Five, que logró vencer a los campeones mundiales de Dota 2 en 2019.

Actualmente, el aprendizaje por refuerzo profundo se aplica en una amplia variedad de dominios más allá de los juegos. En robótica, ha permitido a robots realizar tareas complejas como manipular un cubo de Rubik con una mano robótica. En sostenibilidad, ha sido clave en la reducción del consumo energético en centros de datos. Y en la conducción autónoma, constituye una de las tecnologías fundamentales tanto en el ámbito académico como en el industrial.

3.2. Fundamento teórico

El aprendizaje por refuerzo (Reinforcement Learning, RL, por sus siglas en inglés) es un enfoque del aprendizaje automático inspirado en la psicología del comportamiento, se basa en la prueba y error. Utilizan un paradigma de recompensa y castigo al procesar los datos. Aprenden de los comentarios de cada acción y descubren por sí mismos las mejores rutas de procesamiento para lograr los resultados finales. Los algoritmos también son capaces de funcionar con gratificación aplazada. La mejor estrategia general puede requerir sacrificios a corto plazo, por lo que el mejor enfoque descubierto puede incluir algunos castigos o dar marcha atrás en el camino. El RL es un potente método que ayuda a los sistemas de inteligencia artificial a lograr resultados óptimos en entornos invisibles.

Funciona de la siguiente manera (Véase 10):

1. Hay que crear un programa al que llamaremos **agente**.
2. Este programa debe comprender las dinámicas del sistema en el que opera, es decir, interpretar correctamente el **entorno** que lo rodea.
3. Debe entender lo que está sucediendo en el entorno, por ejemplo, direcciones o movimientos de algún objeto, esto se llama **estado**.

4. Dependiendo de este estado, el agente deberá moverse de una forma determinada para poder conseguir un objeto, a lo que llamaremos **acción**.

5. El agente tendrá que saber si lo ha hecho bien o mal, si ha cumplido el objetivo o no a lo que llamaremos **recompensa**.

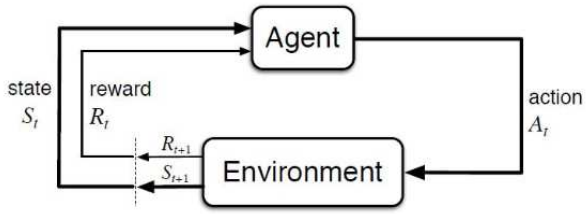

Figura 10: Comportamiento del agente con el entorno [20]

Un agente aprenderá del entorno, mediante la observación de su **estado** y de la interacción de una serie de **acciones**, por las que recibirá una **recompensa**. Estos agentes funcionan en función a unos hiperparámetros, donde podemos encontrar: γ referente a la tasa de descuento, controla cuánto valora el agente las recompensas futuras, cuanto mayor es el valor de gamma más importante son las recompensas futuras. ϵ, tasa de exploración, al principio, el agente explora, elige acciones aleatorias, pero con el tiempo explota su conocimiento aprendido, si el ϵ el del 100 % significa que todas las acciones que toma son aleatorias, mientras que si es del 1 % quiere decir que el agente prácticamente no elige acciones aleatorias, actúa al completo por lo que ha aprendido. η que es la velocidad con la que se actualizan los pesos del modelo. $epsilon - decay$, que indica cómo disminuye la exploración a medida que el agente aprende.

Hay diferentes tipos de agentes, entre los más comunes podemos encontrar:

- Agentes basados en valor: Estos agentes aprenden una función de valor que les indica qué tan buena es una acción o estado, y a partir de eso deciden su política. Podemos encontrar el **Q-learning agent**, que utiliza como algoritmo base [3] el Q-learning y aprende una tabla de valores Q es útil en entornos simples y discretos. O el agente **DQN (Deep Q-Network)**, que utiliza el algoritmo Q-learning junto con las redes neuronales profundas [4], usa una red neuronal para aproximar la Q-table en entornos grandes o continuos.

- Agentes basados en política: Aprenden directamente una política $\pi(a|s)$, en lugar de una función de valor. Podemos destacar el **REINFORCE agent**, que usa el algoritmo REINFORCE, y se basa en el gradiente de política. Este tipo de agente tiene alta varianza.

- Agentes actor-crítico: Donde se usan dos redes neuronales, una para la política (actor) y otra para la función de valor (crítico). Podemos destacar el **A2C**, con algoritmo base Actor-Critic, y usa la ventaja $A(s, a)$ para actualizar el actor.

[3] Todo lo referente a algoritmos será explicado más adelante en el apartado 3.2.2.
[4] Todo lo citado sobre redes neuronales será explicado en más profundidad en el apartado 3.3.1.

También se pueden encontrar distintos tipos de entornos:

- Entornos con espacio de acción **discreto**: Las acciones posibles están definidas en un conjunto finito. Muy comunes en problemas como **CartPole** o **FrozenLake**.
- Entornos con espacio de acción **continuo**: Las acciones pueden tomar valores dentro de un rango continuo. Frecuentes en tareas de control robótico o físicos, como **Pendulum** o **BipedalWalker**.
- Entornos **totalmente observables**: El agente tiene acceso completo al estado del entorno. Son los más usados en entornos simples y simulaciones básicas.
- Entornos **parcialmente observables**: El agente solo recibe información limitada o ruidosa del estado real. Se representan como POMDP y aparecen en entornos más complejos o realistas.
- Entornos **multiagente**: Varios agentes interactúan entre sí, ya sea cooperando o compitiendo. Típicos en juegos, tráfico o simulaciones sociales.
- Entornos **estocásticos**: Las acciones no siempre producen el mismo resultado. Introducen incertidumbre, como ocurre en mercados o sistemas con ruido.

Más allá de la definición del agente y del entorno en sí, se pueden identificar cinco subelementos principales en un sistema de aprendizaje por refuerzo: una política, una señal de recompensa, una función de valor, un modelo del entorno y los episodios.

La **política** define la forma de comportarse el agente en su aprendizaje en un tiempo dado. Se trata de una correspondencia entre los estados percibidos del entorno y las acciones que deben realizarse cuando se está en dichos estados. Es la probabilidad de tomar una acción a en un estado s. En términos psicológicos, equivale a un conjunto de reglas o asociaciones estímulo-respuesta. En algunos casos, la política puede definirse como una función simple o una tabla de consulta; en otros, puede requerir cálculos más complejos, como un proceso de búsqueda. La política constituye el núcleo de un agente de aprendizaje por refuerzo, en el sentido de que, por sí sola, es suficiente para determinar su comportamiento. Puede ser estocástica, si es una distribución de probabilidad o determinista si siempre devuelve la misma acción en cada estado. También hay política como la de $Q-valor$ donde se elige el mayor valor entre las acciones a o la $\epsilon - greedy$, donde empieza eligiendo aleatoriamente a y va reduciendo ϵ mientras avanza y aprende.

La **señal de recompensa** define el objetivo en un problema de aprendizaje por refuerzo. En cada paso de tiempo, el entorno envía al agente un único valor numérico: una recompensa. El único objetivo del agente es maximizar la recompensa total que recibe a largo plazo. Por tanto, la señal de recompensa determina qué eventos son considerados buenos o malos para el agente. La recompensa que recibe el agente en un momento dado depende de la acción que haya realizado y del estado actual del entorno en el que se encuentra, Figura 11. La señal de recompensa es la base principal para modificar la política. Si una acción seleccionada por la política resulta en una recompensa baja, es posible que la política se ajuste para seleccionar una acción diferente en esa misma situación en el futuro.

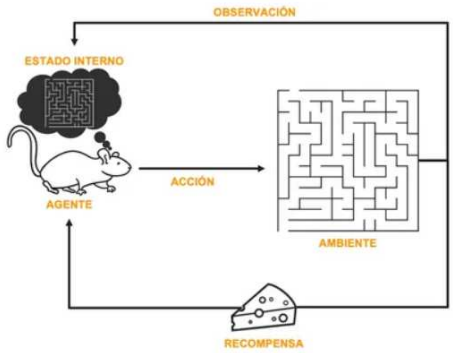

Figura 11: Interacción agente-ambiente [12]

Es fundamental comprender que las recompensas no deben ser una guía detallada sobre cómo llevar a cabo una tarea, sino más bien un incentivo para que el agente resuelva la tarea de la manera que sea más efectiva para alcanzar nuestros objetivos.

La mayoría de los algoritmos de aprendizaje por refuerzo se fundamentan en la estimación de **funciones de valor**. Estas funciones se aplican a estados individuales o pares estado-acción, y su propósito es evaluar cuán beneficioso es para el agente encontrarse en un estado dado o que tan bueno es realizar una acción específica en un estado determinado. Por ejemplo en la política se representa como $\pi(s,a)$, y proporciona información sobre la probabilidad de tomar una acción a cuando el agente se encuentra en el estado s. En términos más informales, el valor de un estado bajo una política particular π, que se denota como $\mathrm{V}\pi(s)$, representa la expectativa de rendimiento al comenzar en dicho estado s y seguir la política π establecida. Podemos definir $\mathrm{V}\pi(s)$ de manera formal como podemos ver en la ecuación (2):

$$V^{\pi}(s) = \mathbb{E}_{\pi}\left[\sum_{t=0}^{\infty} \gamma^t r_t \,\middle|\, s_0 = s\right] = \mathbb{E}_{\pi}\left[G_0 \,|\, s_0 = s\right], \quad s \in \mathcal{S} \tag{2}$$

- $\mathbb{E}_{\pi}$: Representa la **esperanza matemática** bajo la política π, es decir, el valor promedio esperado si el agente sigue dicha política.
- G_0: Denota el **retorno**, es decir, la suma total de recompensas que el agente acumula desde el tiempo t en adelante.
- r_t: Corresponde a la **recompensa** obtenida en el paso temporal t.

El cuarto componente en algunos sistemas de aprendizaje por refuerzo es el **modelo del entorno**. Este permite realizar inferencias sobre el comportamiento futuro del entorno. Por ejemplo, dado un estado y una acción, el modelo puede predecir el siguiente estado y la recompensa correspondiente. Los modelos se utilizan para llevar a cabo tareas de planificación, entendida como cualquier método que permita decidir un curso de acción considerando situaciones futuras antes de que estas ocurran realmente.

Finalmente, el quinto componente corresponde con los **episodios**, que constan de una secuencia de pasos (steps), que también pueden considerarse experiencias completas del agente, ya que cada uno se registra el estado, la acción tomada, la recompensa obtenida, el siguiente estado y si el episodio ha terminado. Cada episodio da tantos pasos como filas tenga la base de datos y hace tantas observaciones como columnas haya en la misma.

En definitiva, el agente a partir de esa observación del entorno, es decir, de vectores (fila de todos los datos de la base de datos), toma una decisión, es decir, una acción y en ese momento, cuando acaba la acción recibe la recompensa, indicando si lo que ha hecho ha estado bien o mal. Este proceso se vuelve a repetir hasta la última fila, marcando el final del episodio, y ya el agente vuelve a reiniciarse y a aprender desde el principio.

3.2.1. Diferencias con otros tipos de aprendizaje

Aprendizaje supervisado

Se trata de un tipo de aprendizaje automático, que como su propio nombre indica, tiene la presencia de un supervisor. Son datos que ya vienen acompañados de las respuestas o clasificaciones correctas como podemos ver en la Figura 12. En este enfoque, primero se entrena al modelo con un conjunto de datos bien etiquetado, y posteriormente se le proporciona un nuevo conjunto de ejemplos. El algoritmo analiza los datos de entrenamiento y aprende a predecir correctamente los resultados para nuevos datos basándose en las etiquetas previamente conocidas.

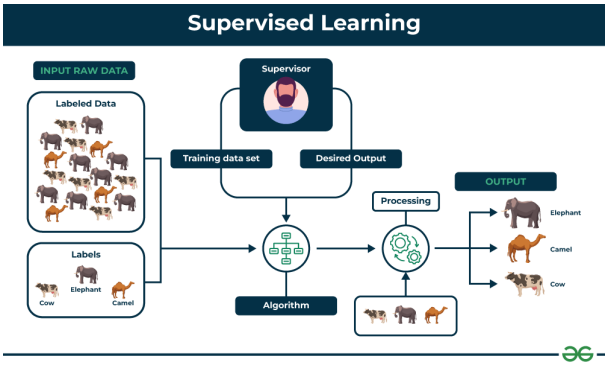

Figura 12: Aprendizaje supervisado [14]

Aprendizaje no supervisado

Al igual que el aprendizaje supervisado, se trata de un tipo de aprendizaje automático. En este caso los datos trabajan sin respuestas o clasificaciones predeterminadas. El objetivo principal está en encontrar patrones y relaciones dentro de los propios datos sin ningún tipo de guía o pista. En este caso, la máquina debe analizar los datos y agruparlos según sus similitudes, patrones o incluso diferencias, como podemos ver en la Figura 13. Es la propia máquina la que debe encontrar esos patrones, ya que a diferencia de el supervisado no tiene un supervisor.

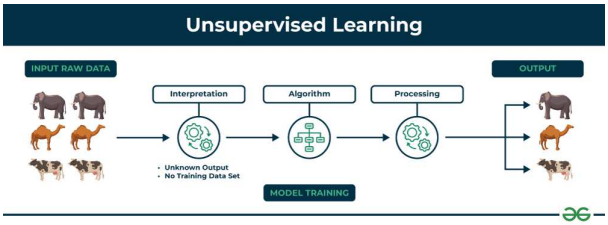

Figura 13: Aprendizaje no supervisado [14]

El aprendizaje por refuerzo se diferencia tanto del aprendizaje supervisado como del no supervisado en la forma en que el modelo aprende a partir de los datos. Mientras que en el aprendizaje supervisado el modelo se entrena con ejemplos etiquetados, y en el no supervisado busca patrones en datos no etiquetados, en el aprendizaje por refuerzo un agente aprende mediante la interacción con un entorno. A través de un proceso de ensayo y error, el agente toma decisiones, recibe recompensas o penalizaciones, y ajusta su comportamiento para maximizar la recompensa acumulada a lo largo del tiempo.

3.2.2. Algoritmos

Un algoritmo es un conjunto de instrucciones definidas, ordenadas y acotadas para resolver un problema, realizar un cálculo o desarrollar una tarea, es un procedimiento paso a paso para conseguir un fin. En la informática, la ciencia de datos o la inteligencia artificial un algoritmo es algo que debemos definir previamente a realizar el código, ya que se debe encontrar la forma de obtener la solución al problema, para luego, a través de este, poder indicarle a la máquina qué acciones queremos que lleve a cabo. Tal y como hemos visto antes se fundamentan en las funciones de valor.

Todo algoritmo está formado por tres partes:

- Entrada: Información que damos al algoritmo con la que va a trabajar para ofrecer la solución esperada.
- Proceso: Conjunto de pasos para que, a partir de los datos de entrada, llegue a la solución de la situación.
- Resultados: a partir de la transformación de los valores de entrada durante el proceso.

Algoritmo Q-learning

Q-learning es un algoritmo de aprendizaje por refuerzo sin modelo utilizado para entrenar agentes capaces de tomar decisiones óptimas a través de la interacción con un entorno. Este método permite al agente explorar distintas acciones y aprender cuáles conducen a mejores resultados. Mediante un proceso de prueba y error, el agente identifica qué acciones generan recompensas, resultados positivos, y cuáles conducen a penalizaciones, resultados negativos.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \right] \tag{3}$$

Con el tiempo, el agente mejora su capacidad de decisión actualizando una tabla Q (Q-table), que almacena los valores Q (Q-values). Estos valores representan la recompensa esperada de realizar una determinada acción en un estado específico.

Este algoritmo se basa en la ecuación de Bellman, (1), donde se evalúan los posibles estados y recompensas futuras. Esta ecuación muestra cómo el valor de estar en un determinado estado depende de las recompensas recibidas y valor de los estados futuros.

Tal y como vemos en la ecuación (3), la ecuación está definida por:

- S_t: Se trata del estado actual.
- A_t: Es la acción realiza por el agente.
- S_{t+1}: Es el siguiente estado al que se mueve el agente.
- a: Es la mejor siguiente acción en el estado S_{t+1}.
- R_{t+1}: Se trata de la recompensa recibida por realizar la acción A_t en el estado S_t.
- γmax_a: Es el factor de descuento que equilibra las recompensas inmediatas con las recompensas futuras. Lo que conseguimos es saber el valor que está teniendo una recompensa futura en el presente.
- α: Es la tasa de aprendizaje que determina en qué medida la nueva información afecta a los antiguos valores Q.

El algoritmo funciona de la siguiente manera: El entorno proporciona al agente un estado S_t que describe la situación o condición actual. En función del estado actual, el agente elige una acción según su política. Esta decisión se basa en la tabla Q que estima las recompensas potenciales para diferentes pares de estado-acción. El agente ejecuta la acción seleccionada y el entorno le proporciona un nuevo estado S_{t+1} y una recompensa R_{t+1}, el agente actualiza la tabla Q utilizando la nueva experiencia. Esto ayuda al agente a estimar mejor qué acciones son más beneficiosas a lo largo del tiempo y por lo tanto el agente mejora su política para tomar mejores decisiones futuras.

Se continúa este ciclo: observa estados, realiza acciones, recibe recompensas y actualiza valores Q a lo largo de muchos episodios. Con el tiempo, el agente aprende la política óptima que produce consistentemente la mayor recompensa posible en el entorno.

Método de gradiente de política

Se trata de un enfoque en el que la política se representa directamente mediante una red neuronal parametrizada por θ, cuyos parámetros se actualizan utilizando el gradiente de la recompensa esperada, con el objetivo de maximizarla.

La fórmula básica es la representada en la ecuación (4).

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta} \left[\nabla_\theta \log \pi_\theta(a|s) \cdot Q^{\pi_\theta}(s,a) \right] \tag{4}$$

- $\nabla_\theta \log \pi_\theta(a|s)$: dirección en la que se deben cambiar los parámetros de la política.
- $Q^{\pi_\theta}(s,a)$: mide lo buena que fue la acción tomada en el estado dado.
- $\mathbb{E}_{\pi_\theta}[...]$: promedio esperado sobre todas las trayectorias posibles siguiendo la política actual.

Sistemas actor-crítico

Este algoritmo es una combinación de los dos anteriores: métodos basados en valores (Q-learning) y los de gradiente de política. Se trata de un enfoque que utiliza dos redes neuronales, una que trabaja como actor, aprendiendo la política y viendo que acción tomar en cada estado y otra que trabaja como crítico, aprendiendo a valorar esa acción en ese estado. Se usa la ventaja $A(s,a)$, que mide lo buena que es la acción comparada con lo normal, com podemos ver la fórmula (5).

$$A(s,a) = Q(s,a) - V(s) \tag{5}$$

- $V(s)$: Estimación del valor del estado.
- $Q(s,a)$: Valor acción-estado

Su objetivo es maximizar la recompensa esperada. Pero en lugar de estimar solo con muestras, el actor usa el feedback del crítico para actualizarse más eficientemente y con menos varianza. Es un sistema de gran complejidad debido a que hay dos redes que entrenar.

3.3. Aprendizaje por refuerzo profundo (Deep reinforcement learning)

El aprendizaje por refuerzo profundo (Deep Reinforcement Learning, DRL, por sus siglas en inglés) representa la combinación de dos áreas clave de la inteligencia artificial: las redes neuronales profundas y el aprendizaje por refuerzo. Esta fusión permite aprovechar, por un lado, la capacidad de las redes neuronales para extraer representaciones complejas a partir de datos (data-driven), y por otro, la capacidad del aprendizaje por refuerzo para optimizar la toma de decisiones a través de la experiencia y la retroalimentación del entorno.

El aprendizaje por refuerzo profundo nace a raíz de las limitaciones del aprendizaje por refuerzo clásico cuando se enfrenta a entornos más complejos. Por ejemplo, en el caso tradicional, no se pueden trabajar con entradas sensoriales como imágenes o señales acústicas, ya que cada situación debe aprenderse desde cero y el sistema solo funciona bien en entornos pequeños y discretos. En cambio, el aprendizaje por refuerzo profundo sí permite tratar con señales complejas, ya que las redes neuronales ayudan al modelo a reconocer patrones comunes entre situaciones parecidas. Esto hace posible que el agente generalice lo aprendido y lo aplique a situaciones nuevas, funcionando mejor en escenarios más realistas.

Los agentes son capaces de aprender directamente reglas a partir de entradas sensoriales, aprovechando la capacidad del aprendizaje profundo para extraer características complejas de datos no estructurados, como imágenes, audio, datos de sensores complejos o correos. El DRL se basa en gran medida en algoritmos como Q-learning, métodos de gradiente de política y sistemas actor-crítico. Conceptos como las redes de valor, las redes de política y el equilibrio entre exploración y explotación son fundamentales en su funcionamiento.

3.3.1. Redes neuronales

Una red neuronal es un modelo computacional inspirado en el funcionamiento del cerebro humano, Figura 14, diseñado para procesar información y aprender a partir de los datos. Estas redes son capaces de identificar patrones sin necesidad de que se definan reglas explícitas previamente. Están compuestas por nodos, también llamados neuronas, interconectados entre sí, los cuales procesan la información, aprenden representaciones internas y son capaces de tomar decisiones en función de lo aprendido.

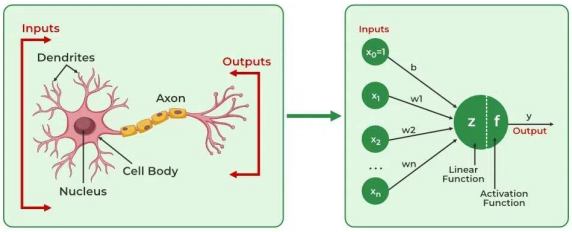

Figura 14: Comparación entre una neuronal biológica y una artificial [21]

Para comprender mejor su funcionamiento, es necesario entender cómo operan las neuronas. Estas en su conjunto forman una red, estando todas interconectadas entre sí. Se puede dividir en tres **capas** (véase Figura 15):

- **Capa de entrada**: Es donde la red recibe los datos de entrada. Cada neurona de entrada corresponde a una característica de los datos de ingreso.
- **Capas ocultas**: Son las capas donde se produce la mayor parte del trabajo computacional. Dependiendo el tamaño de la red puede tener una o varias capas ocultas. La función de esta capa es transformar los datos de entrada en información para que las capas de salidas sean capaces de utilizarlas.
- **Capa de salida**: Se trata de la capa fina y es la que genera una salida del modelo. Puede tener distintos formatos de salida dependiendo de la tarea que estemos realizando.

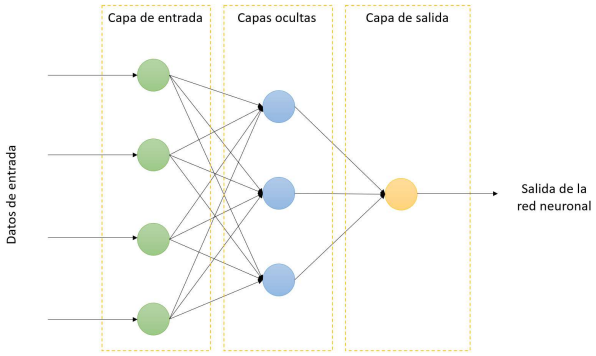

Figura 15: Capas de las redes neuronales [23]

Dependiendo de la forma de los datos (espacial, secuencial, tabular), del objetivo del modelo (clasificar, predecir, generar), y de las restricciones prácticas (tiempo, memoria, potencia de cómputo), se pueden encontrar diferentes tipos de redes neuronales, las más comunes son:

- **Perceptrón Multicapa (MLP) o red neuronal de propagación hacia adelante**: Se trata de un tipo de red neuronal artificial en la que los datos fluyen únicamente en una dirección, desde la capa de entrada hacia la de salida, pasando por las capas ocultas. No existen ciclos ni conexiones recurrentes entre nodos, lo que significa que la información no se retroalimenta ni depende de las entradas anteriores. Se usan en tareas de clasificación y regresión de datos estructurados (tablas, vectores).
- **Redes neuronales convolucionales (CNN)**: se centra en el procesamiento de datos de tipo cuadrícula, como imágenes y vídeos, mediante el uso de capas convolucionales que filtran los patrones y las jerarquías espaciales. Se usa para clasificación de imágenes, detección de objetos, conducción autónoma o visualización en realidad aumentada.
- **Redes neuronales recurrentes (RNN)**: Maneja datos secuenciales donde la salida actual depende no solo de la entrada presente, sino también de las entradas anteriores, ya que la red se retroalimenta para mantener un estado interno o memoria a lo largo del tiempo. Sus aplicaciones son la traducción de idiomas, la clasificación de textos abiertos o la predicción de series de tiempo.

Para que una neurona decida cuánto influye una entrada en su activación, se asignan **pesos** a cada conexión, w_i, de la ecuación (6) correspondiente a la salida en una neurona típica. Estos pesos son valores numéricos que reflejan la importancia relativa de cada entrada, en un principio son aleatorios y se van ajustando poco a poco, como veremos más adelante mediante la retropropagación. El valor final que recibe la neurona se obtiene multiplicando cada entrada por su peso correspondiente, sumando todos los productos, y aplicando una función de activación al resultado. Este proceso se conoce como **propagación directa**.

$$y = \phi\left(\sum_{i=1}^{n} w_i x_i + b\right) \tag{6}$$

1. ϕ : Corresponde a la **función de activación**, cuya finalidad es transformar la salida de la neurona en un valor dentro de un rango determinado, aumentando así el modelado de relaciones no lineales en la red y potenciando a la misma red para modelar datos complejos y matizados.

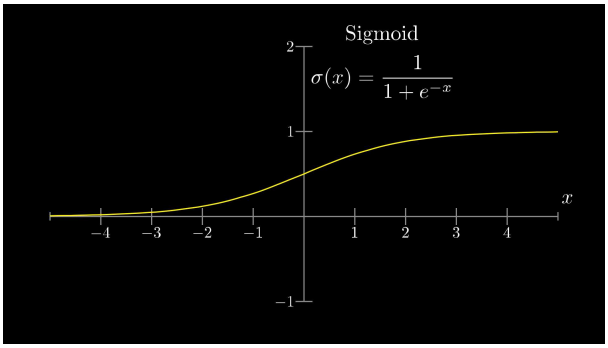

Figura 16: Función sigmoide [15]

Es de suma importancia ya que sin ella, la red neuronal solo se limitaría a modelar reacciones lineales entre entradas y salidas. La mayoría de los datos del mundo real son no lineales. Hay distintos tipos de funciones de activación, se van a definir sigmoid y ReLU:

- **Sigmoid**: como la de la Figura 16. Su función matemática es la de la ecuación (7).

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{7}$$

  La función toma valores cercanos a 0 cuando la entrada es negativa, cercanos a 1 cuando es positiva, y exactamente 0.5 cuando la entrada es 0. Esto permite interpretar la salida como una probabilidad: valores próximos a 1 indican una alta confianza en la clase positiva (clase 1), mientras que valores cercanos a 0 reflejan una alta confianza en la clase negativa (clase 0). En este contexto, la clase 1 representa el caso positivo, por ejemplo, la presencia de una amenaza o un fallo en el sistema. Por el contrario, la clase 0 indica la ausencia de dicha condición. Es importante destacar que ninguna clase implica que el resultado sea bueno o malo. Su interpretación dependerá del problema específico que se esté abordando.

- **ReLU**: De sus siglas en inglés Rectified Linear Unit, es una función de activación que tiene la definición matemática que se presenta en la ecuación (8):

$$\text{ReLU}(x) = \text{máx}(0, x) \tag{8}$$

  Esto significa que, si la entrada x es mayor que 0, la salida es x; y si x es menor o igual a 0, la salida es 0. La función se mantiene plana en 0 a la izquierda del eje, y a partir de $x = 0$ crece de forma lineal, siguiendo la relación $y = x$.

  ReLU tiene algunas alternativas, como ReLU6, Swish o Mish, donde se evita el problema de la neurona muerta. Según el estudio realizado por Dongcheul Lee, profesor en el Departamento de Multimedia de la Universidad de Hannam (Corea del Sur), donde compara distintas funciones de activación para un agente de aprendizaje por refuerzo en un juego de carreras 2D [19], se concluye que ReLU6 fue la que mejor rendimiento obtuvo, superando a otras funciones como ReLU o Swish, y con una diferencia del 35,4 % respecto a CReLU, que fue la que obtuvo la peor recompensa media. Esto demuestra que la elección de la función de activación puede influir significativamente en el rendimiento, incluso usando el mismo entorno y el mismo esquema de aprendizaje por refuerzo.

2. b: Se trata del **sesgo**. Su función es desplazar la salida de la función de activación, permitiendo al modelo ajustar mejor los datos durante el proceso de aprendizaje. Si no existiera el sesgo, todas las funciones de activación de las neuronas pasarían obligatoriamente por el origen (0,0), lo que limitaría enormemente la capacidad del modelo para aprender relaciones complejas. Actúa igual que el termino independiente en una ecuación lineal $y = mx + b$, Permite desplazar la recta verticalmente, igual que en una red neuronal permite ajustar la salida más allá del efecto directo de las entradas.

3. $w_i x_i$: Se tratan de los pesos, w_i y las entradas, x_i. Las entradas son los valores que recibe la neurona desde la capa anterior o directamente desde los datos de

entrada, si se trata de la primera capa. Son las salidas de la neurona anterior, y_i. Los pesos son parámetros ajustables que determinan la importancia de cada entrada. Durante el entrenamiento, la red va ajustando los pesos automáticamente mediante la retropropagación, que se explicará más adelante. En un primer momento se inician con valores aleatorios. Se sabe que los pesos han llegado a un valor óptimo cuando, al combinarlos con las entradas y tras aplicar la función de activación, la salida de la red se aproxima lo máximo posible al valor deseado.

Cálculo del error y retropropagación

Tras la propagación hacia adelante, se completa una **época**, es decir, la red neuronal ha sido recorrida por completo con todos los datos de entrenamiento. A continuación, se evalúa su rendimiento mediante una función de pérdida, que mide la diferencia entre la salida real y la salida predicha; es decir, se realiza el **cálculo del error**.

El objetivo del entrenamiento es minimizar dicha pérdida. En este punto entra en juego la **retropropagación**. Primero, la red calcula la pérdida, que proporciona una medida del error en las predicciones. Esta se basa en la diferencia entre la salida real del modelo y el valor esperado.

Este cálculo se realiza en cada iteración del entrenamiento, y se puede suponer que, a medida que avanzan las épocas, el error tiende a disminuir. Existen varios tipos de error, entre los que se pueden destacar:

- **Error cuadrático medio (MSE) o función de perdida (Loss)**: Este error nos dice cuánto se ha equivocado la red. El objetivo es minimizarlo. Al graficarlo obtenemos cómo aprende el modelo.

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{9}$$

- **Error absoluto medio (MAE)**: Suele ser utilizada como complemento, este error no afecta al entrenamiento, solo se calcula para visualizar. Al ser graficado obtenemos cuánto se desvía de media la predicción del valor real.

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i| \tag{10}$$

  Para ambas ecuaciones:

  - y_i representa el valor real o esperado (target).
  - $\hat{y}_i$ representa el valor predicho por la red neuronal.
  - n es el número total de muestras utilizadas en el cálculo.

A continuación, este error se propaga hacia atrás mediante el proceso de **retropropagación**, que consiste en calcular los gradientes de la función de pérdida con respecto a

cada uno de los pesos modelo, ecuación (11). Una vez obtenidos estos gradientes, entran en juego los **optimizadores**, que se encargan de actualizar dichos pesos y sesgos con el objetivo de minimizar la función de pérdida.

$$\frac{\partial \text{Error}}{\partial w} \tag{11}$$

- $\frac{\partial \text{Error}}{\partial w}$: derivada del error con respecto al peso w, es decir, cuánto cambia el error al modificar ligeramente el valor de w.

Los optimizadores son algoritmos que ajustan los pesos y los sesgos del modelo durante el entrenamiento para minimizar la función de pérdida. Son los culpables de hacer que el red aprenda, ya que enseña a hacer buenas predicciones.

Lo que hacen es aplicar un proceso matemático denominado descenso de gradiente ecuación (12). Consiste en calcular la derivada (gradiente) de la función de pérdida con respecto a cada peso, la misma de la ecuación (11) y mover los pesos en la dirección opuesta al gradiente, ya que con este cálculo lo que obtenemos es ver hacia dónde tiene un mayor aumento el valor del error y lo que nosotros queremos es minimizarlo. Este proceso se repite durante varias épocas hasta encontrar un conjunto de pesos que hagan que la red cometa menos errores.

$$w_{\text{nuevo}} = w_{\text{viejo}} - \eta \cdot \frac{\partial \text{Error}}{\partial w} \tag{12}$$

- w: peso de la red neuronal.
- η: tasa de aprendizaje, que indica cuánto se ajusta el peso en cada iteración.

Hay diferentes tipos de optimizadores. Entre los más comunes podemos encontrar:

- **Gradiente de descenso estocástico (SDG)**: Se trata del más básico, es el que ajusta los pesos en cada muestra o pequeños lotes. Utiliza la ecuación (12). Puede llegar a ser lento especialmente en modelos complejos, debido a estas pequeñas actualizaciones.

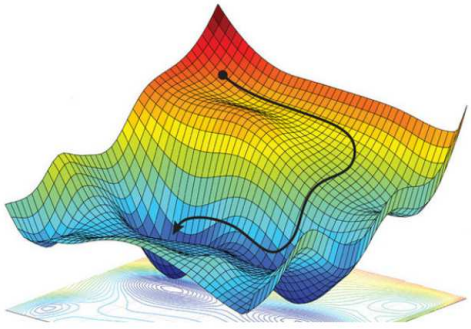

Figura 17: Descenso de gradiente estocástico

- **RMSprop**: Se trata de un método que divide la tasa de aprendizaje entre un promedio de gradientes al cuadrado que decae exponencialmente. Este optimizador es eficaz para gestionar objetivos no estacionarios y se utiliza a menudo para entrenar RNN. Su fórmula matemática es la siguiente, las mostradas en la ecuaciones (13) y (14):

$$s_t = \rho \cdot s_{t-1} + (1-\rho) \cdot \left(\frac{\partial \,\text{Error}}{\partial w_t}\right)^2 \tag{13}$$

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{s_t + \epsilon}} \cdot \frac{\partial \,\text{Error}}{\partial w_t} \tag{14}$$

  Donde:

  - $s_t = \rho \cdot s_{t-1} + (1-\rho) \cdot \left(\frac{\partial \text{Error}}{\partial w_t}\right)^2$
    - s_t: media movil de los cuadrados del gradiente.
    - ρ: factor de decaimiento (tipicamente 0.9).
  - $w_{t+1} = w_t - \frac{\eta}{\sqrt{s_t+\epsilon}} \cdot \frac{\partial \text{Error}}{\partial w_t}$
    - Se adapta el paso de cada peso segun su historial de gradientes.

- **Adagrad**: Adapta la tasa de aprendizaje a los parámetros escalándola inversamente con respecto a la raíz cuadrada de la suma de todos los gradientes cuadrados históricos. Esto ayuda a mejorar el rendimiento con datos dispersos. Su fórmula es la mostrada en las ecuaciones (15) y (16):

$$G_t = G_{t-1} + \left(\frac{\partial \,\text{Error}}{\partial w_t}\right)^2 \tag{15}$$

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \cdot \frac{\partial \,\text{Error}}{\partial w_t} \tag{16}$$

  Donde:

  - $G_t = G_{t-1} + \left(\frac{\partial \text{Error}}{\partial w_t}\right)^2$
    - G_t: acumulado de los cuadrados del gradiente.
  - $w_{t+1} = w_t - \frac{\eta}{\sqrt{G_t+\epsilon}} \cdot \frac{\partial \text{Error}}{\partial w_t}$
    - Disminuye la tasa de aprendizaje a medida que se entrena.

- **Adam**: Es uno de los optimizadores más populares gracias a su eficiente gestión de gradientes dispersos y objetivos no estacionarios. combina las ventajas de otras dos extensiones de SGD: AdaGrad y RMSProp. Calcula las tasas de aprendizaje adaptativo para cada parámetro considerando tanto el primer como el segundo momento de los gradientes. Su ecuación es la mostrada desde la (17) a la (21):

$$m_t = \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot \frac{\partial \text{Error}}{\partial w_t} \tag{17}$$

$$v_t = \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot \left(\frac{\partial \text{Error}}{\partial w_t} \right)^2 \tag{18}$$

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \tag{19}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t} \tag{20}$$

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \cdot \hat{m}_t \tag{21}$$

Donde:

- $m_t = \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot \frac{\partial \text{Error}}{\partial w_t}$
  - m_t: promedio movil del gradiente (momentum).
- $v_t = \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot \left(\frac{\partial \text{Error}}{\partial w_t} \right)^2$
  - v_t: promedio movil del cuadrado del gradiente.
- Correcciones por sesgo:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

- Actualizacion de pesos:

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \cdot \hat{m}_t$$

Problemas dentro de las redes neuronales

Durante el proceso de entrenamiento de las redes neuronales pueden surgir varios problemas que afectan al aprendizaje. Entre ellos destacan el **problema de gradiente**, el **problema de las neuronas muertas** y el **sobreajuste**. Los dos primeros están directamente relacionados con el cálculo de los gradientes durante la retropropagación y el comportamiento de las funciones de activación, mientras que el último con el entrenamiento de la neurona.

Dentro del primero encontramos dos problemas principales: el de gradiente evanescente que ocurre cuando durante la retropropagación, la derivada de la función de activación se va convirtiendo progresivamente pequeña en el movimiento hacia atrás en la red. Es un problema que se asocia principalmente a las funciones de activación de Sigmoid y Tanh, se soluciona con ReLU. El otro problema es de los gradientes explosivos, que ocurren cuando los gradientes de las funciones de pérdidas referentes a los pesos, se convierten extremadamente grandes. Tiene su causa en la retropropagación, donde las derivadas de las capas aumentan progresivamente mientras nos movemos hacia atrás. Es justamente lo opuesto a los gradientes evanescente.

El otro problema, el de las neuronas muertas, ocurre debido a una combinación desafortunada de pesos y entradas, donde la salida de la neurona es siempre 0. Ni genera gradiente, ni puede actualizar sus pesos, queda inútil. Hay dos factores principales que afectan, el primero, es cuando la suma ponderada de $w_i x_i + b$ da siempre valores negativo, haciendo que la salida sea 0 y el segundo factor va relacionado con este, ya que con la salida 0, el gradiente también es 0. Con el tiempo si siempre recibe entradas que la dejan en negativo se queda en 0 estancada para siempre. Algunas de sus soluciones son reducir la tasa de aprendizaje de la red neuronal o cambiar la función de activación como Leaky ReLU.

Finalmente, el sobreajuste que ocurre cuando el modelo aprende demasiado bien los datos de entrenamiento, capturando no solo los patrones generales sino también el ruido y las particularidades específicas del conjunto de entrenamiento. Esto provoca que el modelo tenga un rendimiento excelente sobre esos datos, pero falle al generalizar cuando se enfrenta a datos nuevos o no vistos. Algunas de las técnicas usadas para que no exista sobreajuste son el early stopping (detección anticipada), que detiene el entrenamiento cuando la neurona deja de mejorar o el dropout (descarte aleatorio de neuronas) que desactiva las neuronas aleatoriamente durante el entrenamiento.

Redes neuronales profundas

Las redes neuronales profundas, también conocidas como Deep Neural Networks, son un tipo de red neuronal artificial que contiene múltiples capas ocultas entre la capa de entrada y la capa de salida. Estas redes están diseñadas para aprender representaciones jerárquicas y complejas de los datos, lo que las hace muy efectivas para tareas como reconocimiento de imágenes, procesamiento del lenguaje natural, predicción de series temporales, y muchas otras. La diferencia con una red neuronal tradicional es simplemente esa, su diferencia entre número de capas internas; la tradicional posee pocas capas internas mientras que la profunda tiene muchas, incluso cientos o miles.

3.3.2. Agente de aprendizaje por refuerzo profundo

El agente de aprendizaje por refuerzo profundo (DRL) combina el aprendizaje por refuerzo tradicional con redes neuronales profundas para aproximar la función de valor Q. En este caso, se implementa un agente basado en el algoritmo DQN (Deep Q-Network), que permite al agente aprender qué acción tomar en cada estado para maximizar la recompensa acumulada.

Algoritmo DQN

El algoritmo DQN (de sus siglas en inglés, Deep Q-Network), se obtiene a partir de combinar el algoritmo Q-learning con las redes neuronales profundas. A diferencia del algoritmo Q-learning, utiliza una red neuronal para estimar valores en lugar de una tabla extensa.

Su funcionamiento se basa en lo siguiente:

1. **La red neuronal profunda**: La red neuronal actúa como aproximador de la función de valor Q, que estima la calidad de realizar una acción determinada, a en un estado concreto, s. Esta función se representa como $Q(s, a, \theta)$, donde θ son los pesos y sesgos, los parámetros entrenables de la red.
2. **Memoria de Experiencias (Experience Replay)**: Para evitar que el agente aprenda únicamente de experiencias consecutivas. Se almacena cada transición en una memoria o buffer y durante el entrenamiento, se seleccionan minibatches aleatorios de esta memoria.
3. **Red Objetivo (Target Network)**: Además de la red principal, se emplea una red objetivo. Esta red se utiliza para calcular los valores Q objetivo, proporcionando así una referencia estable.
4. **Función de Pérdida (Loss Function)**: El aprendizaje del agente se guía por una función de pérdida que mide la diferencia entre el valor Q predicho y el valor Q objetivo. Esta pérdida se define como, ecuación (22):

$$L(\theta) = \mathbb{E}\left[\left(r + \gamma \max_{a'} Q(s', a'; \theta^-) - Q(s, a; \theta)\right)^2\right] \tag{22}$$

   - r: recompensa inmediata obtenida.
   - γ: factor de descuento.
   - s': estado siguiente.
   - $\max_{a'} Q(s', a'; \theta^-)$: mejor valor futuro estimado por la red objetivo.
   - $Q(s, a; \theta)$: valor predicho por la red principal.

Proceso de toma de decisiones del agente

El agente de aprendizaje por refuerzo profundo lleva a cabo una serie de acciones durante su entrenamiento, el procedimiento que lleva a cabo es el siguiente:

1. **El agente observa el estado actual**: El entorno le proporciona un estado s. Esto se corresponde con una fila de la base de datos.
2. **Decide qué acción tomar**: En este paso entra en juego la red neuronal principal, que actúa como un aproximador de la función de valor $Q(s, a; \theta)$. Esta red recibe como entrada el estado actual s y devuelve como salida un vector con un valor estimado para cada acción posible. Cada valor del vector representa la recompensa total esperada (a largo plazo) si el agente toma esa acción y continúa actuando de forma óptima. Es decir, no se trata de la recompensa inmediata, sino de una estimación futura que tiene en cuenta las decisiones que vendrán después. El agente utiliza este vector para decidir qué acción tomar, normalmente seleccionando aquella con mayor valor Q, salvo que esté explorando. Cabe destacar que estas redes suelen entrenarse durante pocas épocas por lote para evitar el sobreajuste y permitir que el agente aprenda de forma progresiva.

3. **Ejecución de la acción**: El agente elige la acción con mayor valor o una aleatoria dependiendo de la política que esté usando y se ejecuta. El entorno le devuelve una recompensa inmediata, que ha sido previamente establecida. En general, si acierta la acción que se ha predicho, obtiene una recompensa positiva, de lo contrario la recompensa será negativa.

4. **Obtención del valor que predice la red**: Una vez elegida la acción y obtenida la recompensa inmediata, entra en juego la red objetivo, que es una red distinta a la red principal pero con la misma estructura. En este punto se sincronizan copiando el contenido de la principal a la objetivo. Esta función objetivo se calcula a partir de la ecuación de Bellman, ecuación (1), ahora modificada para el objetivo, ecuación (23):

$$Target = r + \gamma \max_{a'} Q(s', a'; \theta^-) \tag{23}$$

   Se calcula el error, con la función de pérdida, fórmula (22) y este error se retropropaga a través de la red principal para ajustar sus pesos con el objetivo de que, en el futuro, sus predicciones se acerquen más al valor correcto. El procedimiento es el mismo que en el de las redes neuronales clásicas.

5. **Se actualizan los pesos**: Se aplica un algoritmo de optimización (por ejemplo, `Adam` o `SGD`), que ajusta los pesos de la red neuronal en la dirección que reduce el error. Esto permite que la red aprenda y mejore sus predicciones a lo largo de los episodios.

6. El agente **termina este paso** y da el siguiente, volviendo a empezar este proceso.

Para saber si el **agente aprende** con el tiempo, la forma más directa es observar si la recompensa total, que es la suma de todas las recompensas inmediatas que el agente obtiene en cada paso, va aumentando. Si al principio eran bajas o negativas, y al final empieza a sumar valores más altos, es señal de que ha aprendido buenas estrategias.

3.3.3. Diferencia entre época, episodio y experiencia

Dado que se trata de una terminología que se utilizará con frecuencia a lo largo del trabajo y que puede generar confusión, a continuación se aclaran las diferencias entre estos conceptos:

- **Época**: Hace referencia a una pasada completa sobre el conjunto de datos de entrenamiento. Es un término propio del entrenamiento de redes neuronales supervisadas, pero no se utiliza habitualmente en el aprendizaje por refuerzo profundo, ya que este no trabaja directamente con etiquetas.
- **Episodio**: Corresponde a una interacción completa del agente con el entorno, desde el inicio hasta que se alcanza un estado terminal. En el caso de un entorno definido por un conjunto de datos tabular, un episodio puede entenderse como el recorrido completo por todas las filas del entorno.

- **Experiencia**: Se refiere a una única transición en el entorno, formada por una tupla del tipo `(estado, acción, recompensa, nuevo estado, done)`. Por ejemplo, si el entorno contiene tres mil filas, recorrerlas una vez completa generará tres mil experiencias, las cuales conforman un episodio. Igualmente este término también se puede confundir con el **paso**, que se refiere a una única interacción entre el agente y el entorno: el agente observa un estado, toma una acción, y el entorno devuelve una recompensa y un nuevo estado. Es el momento en que ocurre la acción y la transición. Es más una unidad de tiempo dentro del episodio. La **experiencia** es la información que se almacena como resultado del **paso**.

HERRAMIENTAS Y ENTORNO DE DESARROLLO

CAPÍTULO 4

4. Herramientas y entorno de desarrollo

4.1. Python

Para el desarrollo del presente trabajo se ha utilizado el lenguaje de programación Python, ya que es el lenguaje empleado en el entorno original de Kaggle, del cual se ha tomado como base el conjunto de datos y parte del código de referencia. Python es un lenguaje de programación de propósito general, ampliamente reconocido por su versatilidad y facilidad de uso. Una de sus principales características es que se trata de un lenguaje interpretado, lo que significa que ejecuta el código línea por línea, sin necesidad de ser compilado previamente. Esta naturaleza facilita el desarrollo, la depuración y la experimentación en entornos dinámicos. Además, Python cuenta con un ecosistema muy amplio de bibliotecas especializadas, lo que permite abordar tareas complejas de análisis de datos, inteligencia artificial y visualización.

Para la implementación del código, se han utilizado dos entornos: **Anaconda Navigator** y **Google Colab**. La primera es una interfaz gráfica que permite gestionar paquetes, crear y administrar entornos virtuales, así como lanzar diversas aplicaciones de desarrollo. Entre ellas se encuentra **Jupyter Notebook**, entorno interactivo elegido para la ejecución del código en Python, ya que facilita la escritura, prueba y visualización de resultados de manera organizada y eficiente. La segunda es un servicio en la nube que permite escribir y ejecutar código Python directamente desde el navegador web. Se trata de una implementación de Jupyter Notebook alojada en la infraestructura de Google.

El uso combinado de ambos entornos se debe a la elevada carga de procesamiento que requería el proyecto. En las fases iniciales, como el análisis exploratorio de los datos (descrito en el capítulo 4), el rendimiento del ordenador fue suficiente. Sin embargo, en la etapa de entrenamiento del agente, la mayor capacidad de procesamiento ofrecida por Google Colab permitió reducir considerablemente los tiempos de ejecución.

4.1.1. Entorno de ejecución

Para la ejecución del modelo se eligió un entorno con aceleración por GPU (concretamente, una GPU NVIDIA T4) y alta capacidad de memoria RAM. Esta decisión se justifica por la necesidad de reducir los tiempos de entrenamiento y garantizar la fluidez durante la ejecución de los procesos más exigentes computacionalmente, como la retropropagación de errores en redes neuronales profundas y el manejo de grandes volúmenes de datos sensoriales.

La GPU T4 está optimizada para tareas de aprendizaje automático y proporciona una aceleración significativa en comparación con entornos CPU, especialmente al trabajar con bibliotecas como TensorFlow. Además, la alta capacidad de memoria disponible permite cargar y procesar conjuntos de datos extensos sin problemas de rendimiento ni cuellos de botella.

4.2. Bibliotecas del entorno de programación

En cuanto a las bibliotecas utilizadas, estas pueden agruparse en distintas categorías, destacando a continuación las más relevantes por su papel dentro del desarrollo del proyecto:

4.2.1. Manipulación y análisis de datos

`numpy`

Se trata de una biblioteca especializada en el cálculo numérico y en análisis de grandes volúmenes de datos. Incorpora a los arrays. Permite representar colecciones de datos de un mismo tipo en varias dimensiones, y funciones muy eficientes para su manipulación. La gran ventaja de numpy es la velocidad de procesamiento de estos arrays, que lo hace ideal para el procesamiento de vectores y matrices de grandes dimensiones. Es la base de muchas otras bibliotecas como pandas, scikit-learn, tensorflow o matplotlib.

`pandas`

Es una librería especializada en el análisis y manejo de grandes estructuras de datos. Es esencial para el proyecto ya que permite leer archivos .CSV, excel y archivos SQL. Además, pandas sirve como base para muchas tareas de aprendizaje automático, inteligencia artificial y ciencia de datos, y se integra perfectamente con otras bibliotecas como numPy, matplotlib, scikit-learn o tensorFlow.

`os` `pathlib` `shutil`

Son librerías que permiten gestionar rutas o mover archivos de directorios. os proporciona funciones para realizar operaciones del sistema operativo, pathlib ofrece un enfoque orientado a objetos para la manipulación de rutas, y shutil ofrece operaciones de alto nivel para archivos y directorios, como copiar, mover y eliminar.

`json` `joblib`

Ambas son bibliotecas que sirven para el almacenamiento de resultados y configuraciones de un proyecto. json permite trabajar con datos en formato JSON (JavaScript Object Notation), que es muy común para almacenar o intercambiar datos entre aplicaciones. joblib es una librería especializada en guardar y cargar objetos grandes de Python como modelos entrenados o transformadores.

`math`

Es una librería matemática básica, parte del núcleo de Python. Aporta funciones más precisas que numpy para ciertos casos puntuales, como la raíz cuadrada u operaciones matemáticas que no requieren vectores o matrices.

`warnings`

Permite gestionar y controlar los mensajes de advertencia que genera Python. Evita mensajes innecesarios al ejecutar código repetidamente o cuando las advertencias no afectan al resultado final.

4.2.2. Preprocesamiento y visualización de datos

`matplotlib.pyplot` `seaborn`

Se tratan de bibliotecas que perimiten la visualización de datos. Con matplotlib.pyplot podemos crear gráficos comunes, como diagramas de barras, histogramas o diagrmas de dispersión. Seaborn, en cambio crea gráficos estadísticos avanzados, como los mapas de calor.

`sklearn.preprocessing` `sklearn.model_selection`

`skmultilearn.model_selection`

Son bibliotecas que sirven para el preprocesamiento de los datos. sklearn.preprocessing ofrece herramientas para escalar, normalizar, codificar y transformar características de los datos. La idea es que los modelos aprendan mejor si los datos están en una forma adecuada. scikit-learn proporciona utilidades para dividir los datos y validar modelos. Es uno de los pilares del entrenamiento y evaluación de modelos de aprendizaje automático.

`statsmodels`

Se utiliza para realizar análisis estadísticos formales, a diferencia de scikit-learn, que está mas orientada al aprendizaje automático, esta biblioteca se centra más en estimación de parámetros, intervalos de confianza o análisis de regresión detallado, entre otros.

`kagglehub`

Herramienta que facilita el acceso y descarga de modelos o recursos disponibles en Kaggle directamente desde Python.

4.2.3. Modelos de aprendizaje automático, aprendizaje profundo y aprendizaje por refuerzo

`tensorflow`

Plataforma de código abierto para construir y entrenar modelos de aprendizaje automático y redes neuronales, utilizada especialmente en el aprendizaje profundo.

`gym`

Es una biblioteca desarrollada por OpenAI que proporciona una interfaz estándar para crear y entrenar agentes de aprendizaje por refuerzo en entornos simulados.

`collections.deque`

Estructura de datos optimizada para insertar y eliminar elementos rápidamente por ambos extremos, comúnmente utilizada como memoria de experiencia en algoritmos como DQN.

`random`

Módulo de Python que permite generar números aleatorios o seleccionar elementos de forma aleatoria, útil en políticas de exploración como la $\epsilon - greedy$.

CONJUNTO DE DATOS (DATASET)

CAPÍTULO 5

5. Conjunto de datos (Dataset)

5.1. Descripción general del dataset

El conjunto de datos empleado en este Trabajo de Fin de Grado ha sido obtenido del repositorio web Kaggle. Se trata de datos sintéticos generados por sensores que integran técnicas de inteligencia artificial, diseñados para simular escenarios representativos del mundo real en el ámbito de la defensa. El dataset incluye múltiples archivos en formato CSV que contienen información relativa a las lecturas de los sensores, estados del sistema, algoritmos de salida, condiciones ambientales, registro de acciones y señales de recompensa.

Tabla 1: Archivos incluidos en el conjunto de datos

Categoría	**Archivos**
Lecturas de sensores (Sensor Readings)	sensor_readings_train.csv sensor_readings_test.csv sensor_readings_validation.csv
Estados del sistema (System States)	system_states_train.csv system_states_test.csv system_states_validation.csv
Salidas algorítmicas (Algorithmic Outputs)	algorithmic_outputs_train.csv algorithmic_outputs_test.csv algorithmic_outputs_validation.csv
Condiciones ambientales (Environmental Conditions)	environmental_conditions_train.csv environmental_conditions_test.csv environmental_conditions_validation.csv
Registros de acciones (Action Logs)	action_logs_train.csv action_logs_test.csv action_logs_validation.csv
Señales de recompensa (Reward Signals)	reward_signals_train.csv reward_signals_test.csv reward_signals_validation.csv

Cada uno de los archivos del conjunto de datos está dividido en tres subconjuntos: `.train`, `.test` y `.validation`, que contienen 3000, 500 y 250 filas, respectivamente. Esta estructura se repite en todas las categorías mostradas en la Tabla 1.

5.2. División en entrenamiento, validación y prueba

Como se muestra en la Tabla 1, el conjunto de datos está dividido en tres subconjuntos principales: *train*, *validation* y *test*, que corresponden a entrenamiento, validación y prueba, respectivamente, Figura 18. En este apartado se explican sus funciones y características con el fin de facilitar una mejor comprensión del flujo de trabajo del modelo.

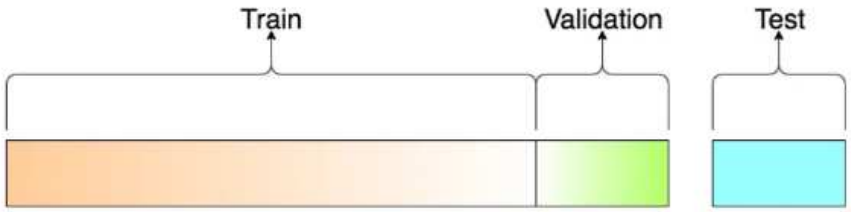

Figura 18: División del conjunto de datos en entrenamiento, validación y prueba [22]

- **Conjunto de entrenamiento:** Es el subconjunto utilizado para ajustar los parámetros internos del modelo, como los pesos y sesgos en una red neuronal. A partir de estos datos, el sistema aprende patrones y estructura su comportamiento, constituyendo la base principal del proceso de aprendizaje.
- **Conjunto de validación:** Se emplea para realizar evaluaciones intermedias durante el desarrollo del modelo. Aunque el sistema accede a estos datos, no los utiliza directamente para aprender, sino que sirven para ajustar los hiperparámetros y optimizar la arquitectura. Su influencia es indirecta, ya que contribuye a mejorar el rendimiento general sin formar parte activa del entrenamiento.
- **Conjunto de prueba:** Reservado exclusivamente para la evaluación final del modelo una vez completado su entrenamiento y validación. No interviene en ningún proceso de ajuste, lo que permite medir de forma objetiva la capacidad de generalización del sistema frente a datos no vistos anteriormente.

5.3. Análisis exploratorio de los datos

Con el fin de orientar el diseño y la evaluación del modelo, se realiza un análisis exploratorio centrado en los datos de recompensa. Para ello, se comparan los distintos archivos del conjunto de datos con el archivo correspondiente a las señales de recompensa. En particular, se analizan los archivos CSV que contienen el identificador `.train`, ya que, como se ha mencionado previamente, son los que presentan un mayor número de filas y, por tanto, ofrecen una base más robusta para el análisis. Para esta fase, los datos fueron descargados localmente para facilitar su manipulación.

Antes de proceder con el análisis, se verificó la existencia de valores nulos en los archivos del conjunto de datos, con el fin de evitar distorsiones en las gráficas y resultados erróneos. Esta comprobación es especialmente relevante en modelos basados en redes neuronales,

ya que estos no pueden procesar valores `NaN` de forma directa. Tras la revisión, se confirmó que no existían valores nulos en ninguno de los archivos.

De igual forma, se evaluó la presencia de filas duplicadas, ya que podrían inducir al modelo a interpretar ciertas situaciones como más frecuentes de lo que realmente son, lo cual afectaría negativamente al aprendizaje del agente. Al igual que en el caso anterior, no se detectaron duplicados en los datos analizados.

Por último, se verificó que todas las columnas de los distintos archivos contuvieran valores coherentes y del mismo tipo, asegurando así la uniformidad de los datos en cada conjunto. Esta comprobación permite garantizar que no existan inconsistencias que puedan afectar al análisis o al entrenamiento del modelo.

Todos los resultados detallados de estas comprobaciones preliminares se recogen en el Anexo A.3, correspondiente a los resultados intermedios y alternativos. En dicho anexo se presentan en distintas tablas clasificadas por archivo, concretamente en la Tabla 14 (valores nulos), la Tabla 15 (filas duplicadas) y la Tabla 16 (coherencia en los tipos de datos por columna). Debido al tamaño considerable de las tablas generadas, se ha optado por mostrar únicamente una parte representativa de ellas. Esta selección permite ilustrar de forma adecuada las comprobaciones descritas anteriormente, sin sobrecargar el contenido del anexo.

Empezando con el análisis exploratorio, para cada uno de los archivos, se han obtenido el número de filas y columnas mediante la función `.shape`, una vista preliminar de las primeras entradas con `.head(5)`, información general sobre las columnas, sus tipos de datos y posibles valores nulos a través de `.info()`, así como estadísticas descriptivas de las variables numéricas utilizando `.describe().T`. En las tablas 2 y 3, se presentan los resultados correspondientes al archivo `sensor_readings_train.csv` como ejemplo representativo.

Tabla 2: Fragmento Representativo del conjunto de datos

Columna	**Tipo de dato**	**No nulos**	**Media (mean)**	**Desviación típica (std)**
timestamp	object	3000	–	–
radar_signal_strength	float64	3000	85.333	7.360
sonar_distance	float64	3000	100.131	14.221
infrared_temperature	float64	3000	34.992	1.658

Tabla 3: Fragmento Representativo de las estadísticas descriptivas de los archivos

Columna	**Mínimo (min)**	**25 % Percentil**	**Mediana (50 %)**	**75 % Percentil**	**Máximo (max)**
radar_signal_strength	68.903	78.594	85.662	92.069	100.502
sonar_distance	77.326	85.880	100.277	114.150	122.355
infrared_temperature	31.016	33.722	35.021	36.237	39.042

A partir del análisis realizado, se observa que ninguno de los archivos presenta valores nulos. Esto se debe a que, en cada caso, el número total de filas coincide exactamente con el número de valores no nulos en las correspondientes columnas. Asimismo, se identifica el tipo de dato presente en cada archivo, perteneciendo todos ellos a estructuras de tipo `DataFrame` de la biblioteca `pandas`. Además, se han obtenido las principales estadísticas descriptivas de las variables numéricas, incluyendo valores máximos, mínimos, medias, desviaciones estándar y cuartiles, lo que permite tener una visión general del comportamiento de los datos antes de su utilización en el modelo.

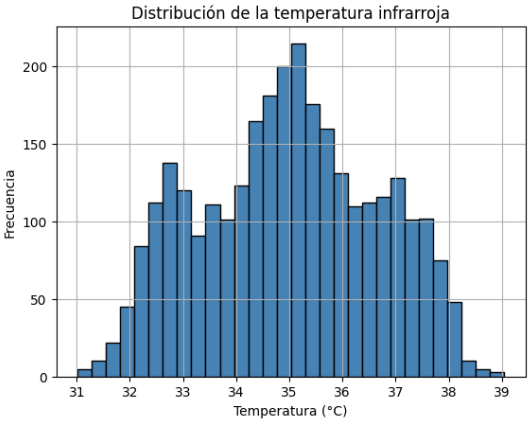

Figura 19: Distribución temperatura infrarroja sensor_readings_train

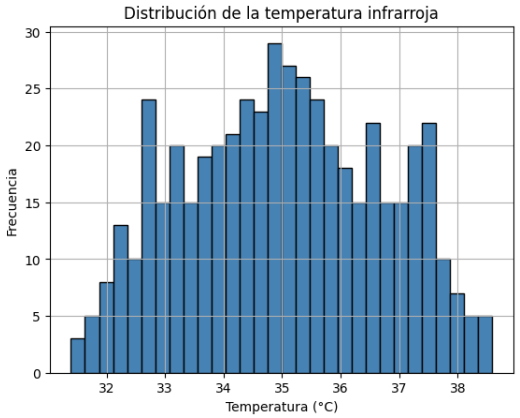

Figura 20: Distribución temperatura infrarroja sensor_readings_test

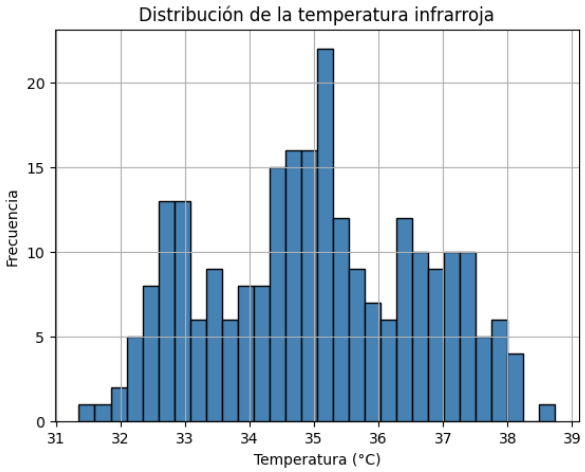

Figura 21: Distribución temperatura infrarroja sensor_readings_validation

Vemos que se trata de un histograma bimodal y que, en las tres gráficas, Figuras 19, 20 y 21, los datos varían de la misma manera. Es decir, tanto en los conjuntos de entrenamiento, prueba y validación, el comportamiento de las variables es prácticamente idéntico. Esto nos permite reafirmarnos en que la mejor opción es utilizar los datos del conjunto de entrenamiento, ya que, además de disponer de una mayor cantidad de muestras, el comportamiento observado es el mismo que en los demás casos.

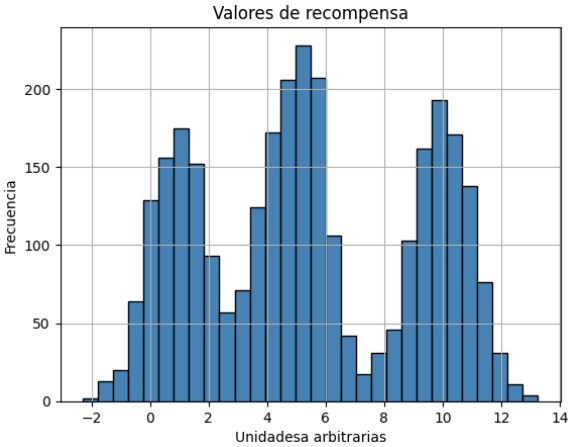

Figura 22: Valores de recompensa

Se observa en la Figura 22 que la mayoría de los valores obtenidos son positivos, lo cual indica que el sistema tiende a funcionar de manera correcta. Además, se aprecia que el entorno es estructurado, ya que se identifican tres picos principales en la distribución de las recompensas. Esto sugiere que el modelo puede realizar tres acciones diferenciadas, correspondiéndose cada una de ellas con recompensas en torno a los valores 1, 5 y 10. Esta estructura clara y fija facilita el proceso de aprendizaje del agente, permitiéndole identificar con facilidad las acciones óptimas a partir de la señal de recompensa recibida.

Tras hacer varios análisis de distintas columnas con respecto a la de recompensa, las gráficas de las Figuras 23 y 24 fueron las únicas que nos información concluyente de cierta influencia sobre el valor de recompensa.

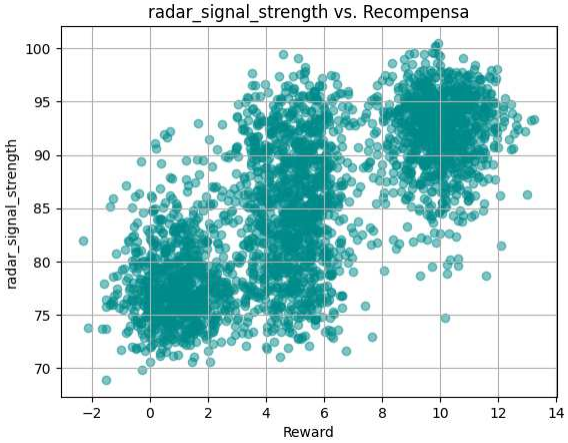

Figura 23: Valores de señal del radar con respecto a los valores de recompensa

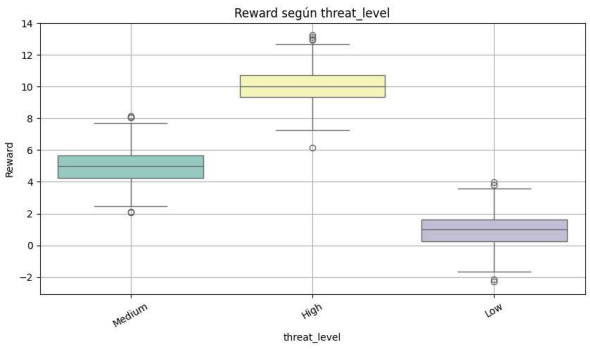

Figura 24: Valores de amenaza con respecto a los valores de recompensa

En la Figura 23, que representa un diagrama de dispersión, se observa que la recompensa varía en función de la señal de radar. Se aprecia una tendencia ascendente, es decir, vemos que a medida que aumenta la intensidad de la señal del radar, los valores de recompensa también van incrementándose.

Por otro lado, en la Figura 24, dado que se trata de datos cualitativos, se ha empleado un diagrama de caja para su representación. En este gráfico se puede observar que a mayor nivel de amenaza, mayor es la recompensa asociada, y viceversa. Las cajas correspondientes a los distintos niveles de amenaza no se solapan, lo que evidencia diferencias reales entre categorías. Esto implica que las recompensas varían de manera clara y consistente en función del nivel de amenaza. Además, se identifican pocos valores atípicos, lo que refuerza la estabilidad de las observaciones.

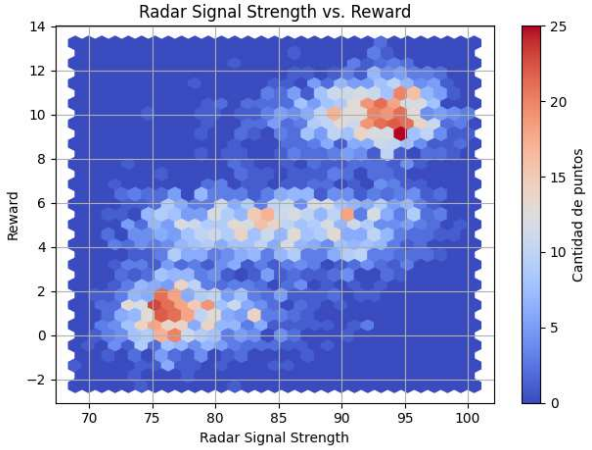

Figura 25: Variación de la señal del radar con respecto a la recompensa en un diagrama de dispersión hexagonal

En el gráfico de la Figura 25 observamos altas concentraciones de puntos para ciertos rangos, concretamente entre 75 y 80, y entre 90 y 100. Estas concentraciones se asocian a valores de recompensa que se sitúan principalmente entre 0 y 2 (zona inferior izquierda) y entre 8 y 10 (zona superior derecha). A diferencia del gráfico anterior vemos que la zona central del gráfico, es decir de los valores de 80 y 90, sale también marcada pero no con tanta intensidad. Por lo que podemos concluir que los valores de recompensa de 4 a 6, no son tan comunes como podíamos intuir.

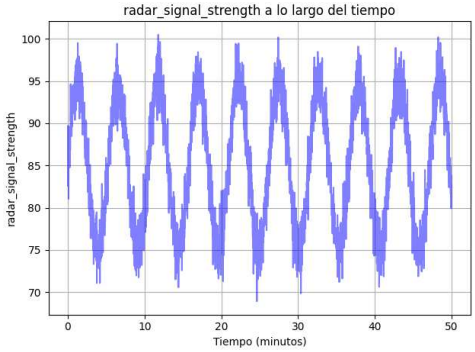

Figura 26: Distribución de la señal del radar con respecto al tiempo

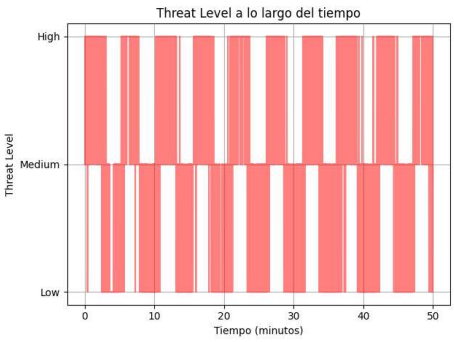

Figura 27: Distribución del rango de amenaza con respecto al tiempo

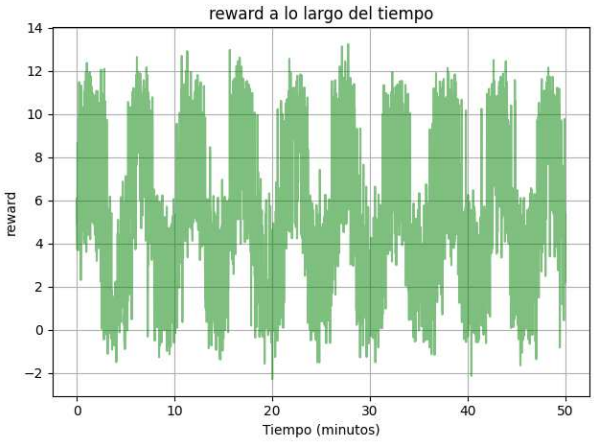

Figura 28: Distribución de la recompensa con respecto al tiempo

A partir del análisis temporal realizado, se observa en las Figuras 26, 27 y 28 que la evolución de la señal de radar y del nivel de amenaza sigue un patrón que guarda una correspondencia clara con la evolución de las recompensas a lo largo del tiempo. La señal de radar presenta una oscilación periódica bien definida, mientras que el nivel de amenaza muestra cambios discretos entre categorías, ambos reflejándose de manera coherente en la evolución de la recompensa. Esta correlación sugiere que el sistema es capaz de interpretar correctamente las señales del entorno, asignando recompensas acordes al comportamiento observado. En consecuencia, se reafirma que las variables seleccionadas proporcionan información suficiente y consistente para guiar eficazmente el proceso de aprendizaje del agente.

5.4. Limitaciones y justificación del uso de los valores de recompensa

Una de las principales limitaciones del conjunto de datos radica en la ausencia de información sobre el origen o la lógica que genera los valores de recompensa. Los valores aparecen ya calculados en los archivos proporcionados, sin documentación adicional que permita comprender su procedencia.

Se han identificado ciertas correlaciones entre variables como la intensidad del radar o el nivel de amenaza y los valores de recompensa, sin embargo, estos patrones no permiten justificar de forma razonada y completa el origen de dicha señal.

No obstante, el uso de estos valores se justifica por una razón fundamental: son **los datos que proporciona directamente el conjunto original**, y constituyen **la única fuente disponible** para entrenar al agente. En el contexto del aprendizaje por refuerzo, el agente no necesita conocer cómo se calcula la recompensa, sino simplemente adaptarse a ella para optimizar su comportamiento. Por tanto, se parte de la premisa de que dichos valores reflejan un entorno cerrado y simulado, diseñado por el autor del dataset, en el que la señal de recompensa representa las consecuencias (positivas o negativas) de las acciones tomadas.

En definitiva, aunque no es posible justificar los valores de recompensa desde un punto de vista analítico o transparente, se asume su validez operativa como parte integral del entorno de entrenamiento proporcionado. Son los valores que el conjunto de datos ha aportado y, como tales, son los que utilizan tanto el agente como las redes neuronales para llevar a cabo su aprendizaje.

DESARROLLO DEL MODELO

CAPÍTULO 6

6. Desarrollo del modelo

6.1. Preparación de los datos

Los aspectos relativos a la descripción, estructura y análisis exploratorio del conjunto de datos se han abordado en profundidad en el Capítulo 4. En esta sección, se resume de forma operativa cómo se ha llevado a cabo la preparación final de los datos para su uso tanto en la red neuronal como en el modelo de aprendizaje por refuerzo profundo.

En este capítulo se llevó a cabo el análisis utilizando directamente los archivos del repositorio de Kaggle, en lugar de descargarlos localmente como se indicó en el capítulo anterior. Esta decisión se debe a que, como se ha explicado anteriormente, durante la etapa de entrenamiento del agente era necesario disponer de una mayor capacidad de procesamiento. Por lo tanto, se consideró que la mejor opción era trabajar directamente con los datos del repositorio de Kaggle mediante el uso de la biblioteca `kagglehub`.

Se comenzó comprobando la calidad de los datos, realizando una nueva revisión para verificar si existía algún valor nulo en las columnas o si alguna de las bases de datos contenía únicamente una columna. Se pudo comprobar que no era así, tal y como se observó también en el capítulo 4. Asimismo, se verificó que todas las bases de datos contasen con la columna *timestamp*, ya que esta es esencial para poder realizar la unión entre ellas.

Utilizando la función `merge` de la biblioteca `pandas`, se realizó una cadena de fusiones entre todas las bases de datos que contenían el identificador `.train`, utilizando la columna `timestamp` como clave de unión. De este modo, fue posible trabajar con los datos en una única base de datos integrada, lo cual resulta esencial para el posterior procesamiento de variables, el entrenamiento del modelo y la evaluación de las predicciones de *reward*.

Se procede al preprocesamiento de los datos con el objetivo de preparar las variables para su uso en la red neuronal. En primer lugar, se convierten las columnas categóricas, es decir, aquellas que contienen texto, en valores numéricos, ya que las redes neuronales no pueden trabajar directamente con texto o categorías nominales. Para ello, se emplea la clase `LabelEncoder()` del módulo `sklearn.preprocessing`, que permite crear una instancia del codificador que asigna un número entero distinto a cada categoría de una columna. Por ejemplo, si los valores son `Low`, `Medium` y `High`, estos podrían codificarse como 0, 1 y 2, respectivamente.

A continuación, se escalan las variables numéricas del conjunto de datos a una escala estándar. Esta práctica es fundamental para que el modelo aprenda de manera eficiente, ya que muchas redes neuronales son sensibles a las diferencias de escala entre características. Por ejemplo, si una variable oscila entre 0 y 1, y otra entre 0 y 1000, esta última podría dominar el cálculo y dificultar la convergencia del modelo. A modo ilustrativo, se muestra a continuación una tabla de valores originales y sus respectivos valores escalados:

Tabla 4: Ejemplo de valores que han sido transformados a una escala con media cero y desviación unitaria.

Valor original	Valor escalado
100	1,5
80	0,7
60	-0,1
40	-0,9

Para llevar a cabo esta transformación, se utiliza un escalador estándar de `scikit-learn` y, mediante el método `fit_transform()`, se calcula la media y la desviación estándar de cada columna numérica, transformando sus valores según la fórmula (24):

$$z = \frac{x - \mu}{\sigma} \tag{24}$$

donde:

- x es el valor original.
- μ es la media de la columna.
- σ es la desviación estándar.

En esta implementación se decidió escalar únicamente las variables de entrada, manteniendo la variable de salida en su escala original. Esta decisión se debe a que la salida representa el valor real de la recompensa, el cual es fundamental para que el agente de aprendizaje por refuerzo pueda aprender correctamente.

Escalar la *reward* podría distorsionar su significado, ya que el agente no estaría recibiendo recompensas en la magnitud real del entorno, lo que dificultaría o incluso impediría un aprendizaje efectivo. Al mantener la recompensa sin escalar, se garantiza que los valores utilizados para calcular la función objetivo del agente reflejen correctamente el comportamiento deseado del sistema.

Por último, se preparan los datos para entrenar un modelo de aprendizaje automático.

Se define la variable `X` como el conjunto de características (variables independientes), seleccionando todas las columnas excepto `timestamp` y `reward`. La columna `timestamp` se excluye porque no aporta un valor predictivo directo, ya que simplemente representa la dimensión temporal. La columna `reward`, por su parte, es la variable objetivo que se desea predecir, y por tanto se asigna a la variable `y`.

A continuación, se realiza una nueva división del conjunto de entrenamiento (los datos `_train`) para obtener un subconjunto destinado a la validación del modelo. Esta

subdivisión se realiza con el objetivo de evaluar el rendimiento del modelo durante el entrenamiento, ajustar hiperparámetros, seleccionar funciones de activación adecuadas y detectar posibles problemas de *overfitting*.

Para ello, se utiliza la función `train_test_split`, que divide los datos en un 80 % para entrenamiento (`X_train`, `y_train`) y un 20 % para validación (`X_val`, `y_val`).

Se emplea el parámetro `random_state=42` para garantizar la reproducibilidad de los resultados. Este parámetro controla la aleatoriedad en la división de los datos, y se establece comúnmente en 42 por convención, aunque cualquier valor fijo serviría con el mismo propósito.

Tras la división, el conjunto de entrenamiento quedó compuesto por 2400 muestras, mientras que el conjunto de validación contiene 600 muestras. El modelo se entrena utilizando un total de 19 variables predictoras.

6.2. Arquitectura de la red neuronal para la predicción de la recompensa

En este apartado se explicará el uso de esta red neuronal que tiene un rol auxiliar y analítico. Aunque esta red no se emplea directamente durante el entrenamiento del agente ni se integra en el entorno de aprendizaje por refuerzo, su desarrollo permite explorar la capacidad predictiva del conjunto de datos y analizar el impacto de distintas configuraciones arquitectónicas sobre la calidad de la predicción de la recompensa. Se realizará principalmente prediciendo el valor de la recompensa y comparándolo con el valor real de `reward`, a través de la función de perdida (MSE) y el error cuadrático medio (MAE).

6.2.1. Características de la red neuronal

En este apartado se describen las principales características estructurales de la red neuronal desarrollada para predecir la recompensa a partir de las variables de entrada.

Tipo de red

Listing 1: Arquitectura del modelo para la predicción de la recompensa

```
model = Sequential([
    Dense(256, input_dim=X_train.shape[1]),
    Lambda(ses_activation),
    Dropout(0.3),
    Dense(512),
    Lambda(ses_activation),
    Dropout(0.3),
    Dense(512),
    Lambda(ses_activation),
    Dense(1, activation='linear')])  # Regression output
```

Se trata de una red Perceptrón Multicapa (MLP), también conocida como red neuronal de propagación hacia adelante, en la que los datos fluyen únicamente en una dirección: desde la capa de entrada hasta la de salida. Esta arquitectura se diferencia de otras como las redes neuronales recurrentes (RNN), que presentan dependencias temporales, o las redes convolucionales (CNN), orientadas al procesamiento espacial.

En este caso, el modelo está compuesto por varias capas densas intermedias, lo que lo convierte en una red neuronal profunda. Estas capas están activadas mediante una función personalizada `ses_activation`, implementada mediante capas `Lambda`. Además, se incorporan capas `Dropout` para reducir el riesgo de sobreajuste, y una capa de salida con activación lineal, adecuada para generar una predicción continua del valor de `reward`.

A continuación, en la Tabla 5, se presenta un resumen de la arquitectura, donde se especifican las capas que lo componen, la forma de salida de cada una de ellas y el número de parámetros entrenables.

Tabla 5: Resumen de la arquitectura del modelo

Capa (tipo)	**Forma de salida**	**Parámetros**
`dense` (Dense)	(None, 256)	5,120
`lambda` (Lambda)	(None, 256)	0
`dropout` (Dropout)	(None, 256)	0
`dense_1` (Dense)	(None, 512)	131,584
`lambda_1` (Lambda)	(None, 512)	0
`dropout_1` (Dropout)	(None, 512)	0
`dense_2` (Dense)	(None, 512)	262,656
`lambda_2` (Lambda)	(None, 512)	0
`dense_3` (Dense)	(None, 1)	513
Total		**399,873**
Parámetros entrenables		**399,873**
Parámetros no entrenables		**0**

Este resumen permite observar la complejidad del modelo, así como la distribución de parámetros entre las distintas capas. En este caso, la red cuenta con un total de 399,873 parámetros entrenables, repartidos en cuatro capas densas y tres capas `Lambda` que aplican la activación personalizada. Las capas `Dropout`, por su parte, no añaden parámetros, pero cumplen una función clave en la regularización.

Función de activación

Como se ha descrito en la definición de la red neuronal del modelo, esta está formada por cuatro capas densas y tres capas `Lambda` que aplican una función de activación personalizada. Las capas densas no tienen especificada ninguna activación, por lo que, por defecto, implementan una transformación lineal. Esta transformación puede expresarse mediante la siguiente ecuación:

$$y = Wx + b \tag{25}$$

donde:

- W es la matriz de pesos.
- x es el vector de entrada.
- b es el vector de sesgos.

La función de activación utilizada es una combinación personalizada de las funciones `sigmoid` y `ReLU`, multiplicadas por un factor de escala $\alpha = 0{,}1$. Esta función, denominada `ses_activation`, fue definida por el autor del conjunto de datos original. Su implementación en código es la siguiente:

Listing 2: Definición de la función de activación personalizada SES

```
def ses_activation(x, alpha=0.1):
    return K.sigmoid(x) * K.relu(x) * alpha
```

Esta función combina los beneficios de ambas activaciones clásicas. Por un lado, la función $ReLU$ anula todos los valores negativos, lo que introduce no linealidad y reduce el riesgo de saturación por valores bajos. Por otro lado, la función *sigmoid* suaviza la salida para los valores positivos, evitando transiciones bruscas. Finalmente, el factor de escala α limita la magnitud de la activación, lo que contribuye a evitar explosiones numéricas durante el entrenamiento.

Intuitivamente, esta función puede interpretarse como: "activa solo los valores positivos, pero modula la salida suavemente y evita que crezca demasiado rápido". En la Figura 29 se muestra su comportamiento gráfico.

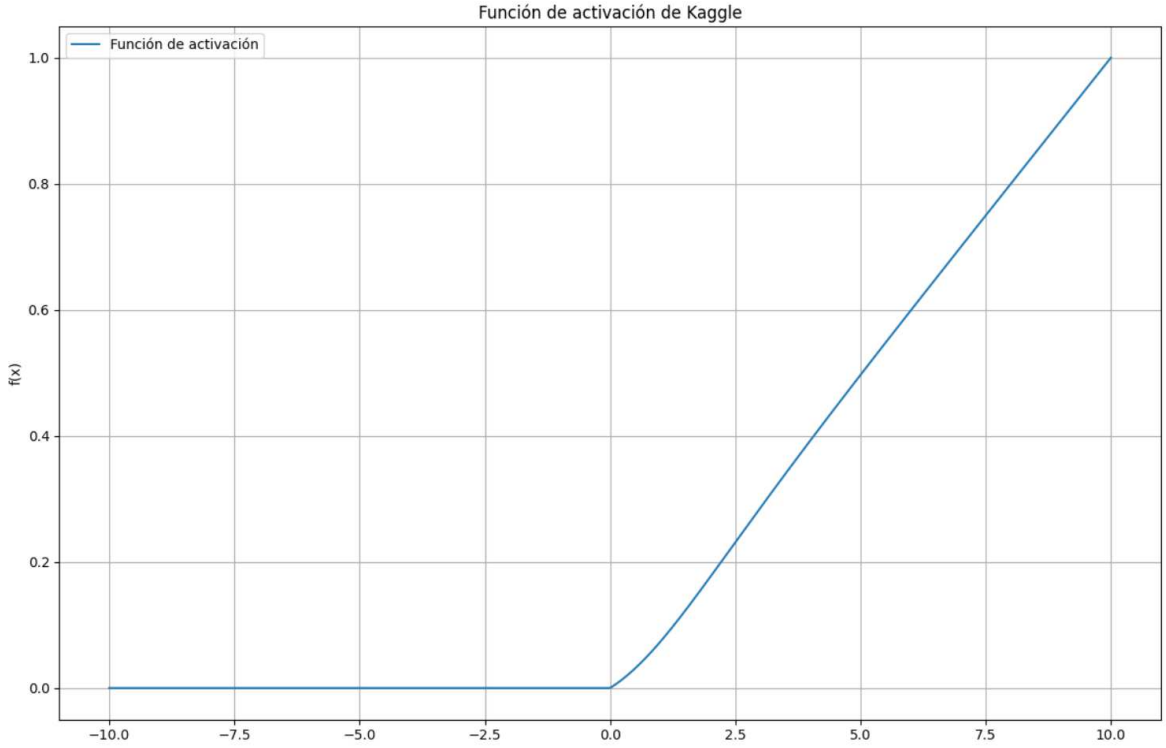

Figura 29: Representación gráfica de la función de activación `ses_activation`

En secciones posteriores se analizan los efectos del uso de las funciones de activación por separado, `sigmoid` y `ReLU`, así como `ReLU6`.

Optimizador

Listing 3: Compilación del modelo con optimizador Adam y función de pérdida MSE

```
model.compile(optimizer='adam', loss='mse', metrics=['mae'])
```

Se utiliza el optimizador `Adam`, que es ampliamente reconocido por su capacidad para adaptarse dinámicamente a la magnitud de los gradientes durante la optimización, lo que suele traducirse en una convergencia más rápida y estable frente a otras alternativas clásicas.

Además, su comportamiento robusto ante datos ruidosos y su eficiencia computacional lo convierten en una elección muy frecuente en contextos de aprendizaje automático con conjuntos de datos reales y complejos, como es el caso de este trabajo. Por tanto, su adopción en el modelo original se considera razonable y apropiada.

Como podemos ver en el código, se utiliza el error cuadrático medio (MSE), ecuación (9) y el error absoluto medio (MAE), ecuación (10).

6.2.2. Modelo entrenado

El modelo fue entrenado mediante el método `fit()` de la API de `Keras`, utilizando los conjuntos `X_train` y `y_train` como datos de entrenamiento, y `X_val` y `y_val` como datos de validación. Se estableció un número máximo de 100 épocas y un tamaño de lote (*batch size*) de 32 muestras por iteración.

Para evitar el sobreajuste y reducir el tiempo de entrenamiento innecesario, se implementó la técnica de parada temprana mediante el callback `EarlyStopping` de Keras. Esta técnica monitoriza la pérdida en el conjunto de validación (`val_loss`) y detiene el entrenamiento si no se observa una mejora tras 4 épocas consecutivas.

El fragmento de código correspondiente al entrenamiento es el siguiente:

Listing 4: Entrenamiento del modelo con early stopping

```
history = model.fit(
    X_train, y_train,
    epochs=100,
    batch_size=32,
    validation_data=(X_val, y_val),
    callbacks=[
        tf.keras.callbacks.EarlyStopping(monitor='val_loss',
           patience=4, restore_best_weights=True)
    ]
)
```

6.3. Entorno personalizado para datos de sensores de IA con OpenAI Gym

En este apartado se describen las características principales del entorno personalizado desarrollado utilizando la biblioteca `OpenAI Gym`, diseñado específicamente para trabajar con el conjunto de datos de sensores de inteligencia artificial. Este entorno actúa como una interfaz entre el agente de aprendizaje por refuerzo y los datos, definiendo los espacios de observación y acción, la lógica de transición de estados, las condiciones de finalización de episodios y, especialmente, la función de recompensa.

Asimismo, se detallan las modificaciones realizadas sobre el entorno original con el objetivo de mejorar el comportamiento del agente, facilitar su aprendizaje y adaptar la dinámica del entorno a las características concretas del problema abordado en este trabajo.

6.3.1. Clase AISensorEnv

Para aplicar aprendizaje por refuerzo profundo sobre el conjunto de datos sensoriales, se ha desarrollado un entorno personalizado que permite simular una interacción continua entre un agente y un sistema dinámico. Este entorno, implementado siguiendo la interfaz estándar de `OpenAI Gym`, se basa en los datos reales del conjunto de sensores y tiene como finalidad ofrecer al agente una representación estructurada del entorno en el que debe aprender a actuar.

A través de este entorno, se convierte una serie de datos históricos en una experiencia secuencial con la que el agente puede interactuar paso a paso. En cada instante, el agente recibe una observación del sistema y debe tomar una decisión, cuya calidad será evaluada mediante una función de recompensa. De esta forma, se simula un proceso de toma de decisiones que permite al modelo aprender comportamientos óptimos a partir de la experiencia acumulada.

El diseño de este entorno ha sido adaptado específicamente a las necesidades del proyecto, permitiendo flexibilidad en el tratamiento de los datos y control sobre las reglas de interacción. Cabe destacar que el entorno original proporcionado en el código base fue modificado, ya que el agente no obtenía resultados satisfactorios en las primeras pruebas de entrenamiento. Por ello, se realizaron ajustes en la lógica del entorno y en la definición de la función de recompensa, con el objetivo de facilitar el aprendizaje del agente y mejorar su capacidad para adaptarse a las decisiones esperadas en el conjunto de datos.

6.3.2. Características del entorno

En esta sección se describen las características principales del entorno personalizado, sin entrar aún en la definición de la recompensa proporcionada al agente. Dado que la función de recompensa constituye uno de los elementos más relevantes del entorno, su explicación detallada se presenta en el apartado 6.3.3.

Espacio de observación

El entorno proporciona al agente una observación basada en las variables sensoriales del conjunto de datos, excluyendo las columnas `timestamp` y `reward`, ya que esta última es la variable objetivo, y la primera no aporta información relevante para la toma de decisiones. En cada paso, el agente recibe un vector numérico que representa el estado actual del entorno.

Listing 5: Definición de las columnas sensoriales del entorno

```
self.feature_columns = self.data.columns.difference(['reward', '
    timestamp']).tolist()
```

Espacio de acción

El agente puede elegir entre cinco acciones posibles. Aunque no se especifica con detalle qué representa cada acción, el espacio de acciones está definido como discreto, lo que implica que las decisiones del agente están limitadas a un conjunto finito de opciones enteras.

Listing 6: Definición del espacio de acción y observación

```
self.action_space = spaces.Discrete(5)
self.observation_space = spaces.Box(
    low=-np.inf, high=np.inf, shape=(len(self.feature_columns),),
        dtype=np.float32
)
```

Dinámica temporal

El entorno avanza paso a paso a través de los datos, comenzando por el primer registro y recorriendo secuencialmente cada instancia sensorial. En cada iteración, se actualiza el paso actual y se proporciona una nueva observación correspondiente a ese instante. Este comportamiento se define principalmente en los métodos `reset`, `_get_observation` y `step`, donde se controla el avance y la obtención del estado actual.

Listing 7: Gestión del paso actual y observaciones

```
def reset(self):
    self.current_step = 0
    self.done = False
    return self._get_observation()

def _get_observation(self):
    observation = self.data.loc[self.current_step, self.
       feature_columns].values.astype(np.float32)
    return observation
```

Condición de finalización del episodio

El episodio concluye cuando el agente ha recorrido todos los registros disponibles en el conjunto de datos. Esta condición se define dentro del método `step`, donde se comprueba si se ha alcanzado el final de la secuencia.

Listing 8: Condición de finalización del episodio

```
if self.current_step >= len(self.data) - 1:
    self.done = True
```

Política de interacción del entorno

Aunque el entorno no define una política explícita, su lógica sugiere que el agente debe aprender una política que le permita seleccionar acciones alineadas con las decisiones reales contenidas en el conjunto de datos. Esta relación se refleja en el método `step`,

donde se compara la acción elegida por el agente con la acción esperada (`action_taken`) para calcular la recompensa.

Listing 9: Referencia a la acción esperada para el cálculo de recompensa

```
expected_action = self.data.loc[self.current_step, 'action_taken']
```

6.3.3. Obtención de la recompensa

El entorno proporciona al agente los siguientes valores de recompensa en función de la acción que selecciona, según se muestra en el siguiente fragmento de código:

Listing 10: Lógica de la función de recompensa

```
if action == expected_action:
    reward = 1.0
elif abs(action - expected_action) == 1:
    reward = -0.05
else:
    reward = -0.4
```

- **1.0**: Se asigna cuando la acción elegida por el agente coincide exactamente con la acción esperada según el conjunto de datos. Se considera la predicción óptima.
- **-0.05**: Se asigna cuando la acción seleccionada difiere en una unidad de la acción correcta. Esta recompensa intermedia penaliza ligeramente el error, pero reconoce una aproximación razonable.
- **-0.4**: Se asigna cuando la diferencia entre la acción del agente y la esperada es superior a una unidad. Se trata de una penalización más severa que refleja un error significativo.

De esta manera, el agente recibe señales que le permiten distinguir entre acciones correctas, parcialmente aceptables y claramente incorrectas. Esta estructura de recompensas facilita un aprendizaje más progresivo y eficiente, ya que refuerza los comportamientos adecuados y penaliza en función del grado de error, favoreciendo una mejor exploración y convergencia durante el entrenamiento.

Ajustes realizados tras el primer entrenamiento

Una vez realizada una primera ejecución del modelo, se pudo comprobar que el agente no aprendía de forma significativa. Una de las principales razones fue la función de recompensa definida inicialmente en el entorno, la cual se muestra a continuación:

Listing 11: Recompensa original basada directamente en los datos

```
def step(self, action):
    reward = self.data.loc[self.current_step, 'reward']
    self.current_step += 1
    if self.current_step >= len(self.data):
        self.done = True
    return self._get_observation(), reward, self.done, {}
```

En esta implementación, la recompensa que recibe el agente proviene directamente de la columna `reward` del conjunto de datos, sin evaluar si la acción seleccionada por el agente fue adecuada o no. Esto implica que, independientemente de la acción que tome el agente, siempre recibe el mismo valor de recompensa que está almacenado en los datos, sin importar si su decisión fue correcta o no. Como consecuencia, el agente no puede aprender qué acciones son mejores, ya que no hay una relación directa entre su comportamiento y la recompensa recibida.

Este enfoque no fomenta el aprendizaje, ya que la señal de recompensa no depende de la acción del agente. Este no tiene forma de diferenciar entre decisiones correctas e incorrectas, ni puede ajustar su política en función de los errores cometidos. Por ello, fue necesario redefinir la función de recompensa para que estuviera vinculada directamente a la calidad de la acción seleccionada, permitiendo así un aprendizaje efectivo por parte del agente.

La función de recompensa expuesta anteriormente soluciona el problema al establecer una relación directa entre la acción seleccionada por el agente y la recompensa obtenida. Así es recompensado positivamente cuando acierta, penalizado ligeramente si se aproxima, y castigado si su acción es claramente incorrecta. Esta retroalimentación diferenciada permite al agente ajustar su comportamiento y aprender una política más eficaz a lo largo del entrenamiento.

6.4. Arquitectura del agente de aprendizaje por refuerzo profundo

En esta sección se describen la composición de agente de aprendizaje por refuerzo profundo. El tipo de agente personalizado, la red neuronal incorporada dentro del mismo así como sus diferencias con la red neuronal anterior utilizada para el análisis de la predicción de la recompensa, la política y los hiperparámetros, junto con los ajustes realizados de la primera ejecución del código.

6.4.1. Clase DQNAgent

La clase `DQNAgent` define el comportamiento del agente de aprendizaje por refuerzo profundo basado en el algoritmo DQN (Deep Q-Network). Esta clase contiene todos los elementos necesarios para la interacción con el entorno, la toma de decisiones, el almace-

namiento de experiencias pasadas y el entrenamiento de la red neuronal encargada de estimar los valores de acción, Q-values. Sus componentes principales incluyen la inicialización de parámetros, la construcción del modelo, la política de exploración, el almacenamiento en memoria y el método de entrenamiento mediante repetición de experiencias.

6.4.2. Red neuronal del agente

El agente utiliza una red neuronal como función de aproximación para estimar los valores `Q` asociados a cada estado y acción. Se trata de una red neuronal profunda ya que consta de dos capas densas con 64 neuronas cada una, activadas mediante la función personalizada `ses_activation`. La capa de salida tiene tantas neuronas como acciones posibles, es decir 5, con activación lineal, ya que se espera un valor continuo para cada acción.

Utiliza el mismo optimizador que el de la red neuronal para la predicción de la recompensa, en este caso `Adam`.

Listing 12: Arquitectura de la red neuronal del agente

```
model.add(Dense(64, input_dim=self.state_size, activation=Lambda(
    ses_activation)))
model.add(Dense(64, activation=Lambda(ses_activation)))
model.add(Dense(self.action_size, activation='linear'))
```

La finalidad principal de esta red es estimar los valores de acción (Q-values) asociados a cada estado posible del entorno. A partir de estas estimaciones, el agente puede tomar decisiones que maximicen la recompensa acumulada a largo plazo, aprendiendo una política óptima mediante la interacción continua con el entorno.

Mientras que la primera red desarrollada previamente para predecir el valor de `reward` tenía un propósito analítico y de exploración de los datos, esta red cumple un papel central en el proceso de toma de decisiones del agente dentro del marco del aprendizaje por refuerzo.

Cabe destacar que, a diferencia de la implementación habitual de DQN, donde se utiliza una red objetivo para estimar el valor futuro esperado y estabilizar el entrenamiento, en esta implementación se prescinde de dicha red. En su lugar, se emplea únicamente la recompensa inmediata generada por el entorno como valor objetivo para el entrenamiento. Esta decisión simplifica la arquitectura y centra el aprendizaje en la corrección de las decisiones individuales, sin estimar recompensas futuras.

Este comportamiento se refleja en el método `replay()` de la clase `DQNAgent`, donde el valor objetivo se define exclusivamente como la recompensa inmediata (`reward`) obtenida por el agente en cada transición:

Listing 13: Cálculo del valor objetivo sin red secundaria

```
target = reward  # solo se considera la recompensa inmediata
if not done:
    target += self.gamma * np.amax(self.model.predict(next_state.
       reshape(1, -1), verbose=0)[0])
```

Aunque el fragmento conserva la estructura típica del DQN clásico, el hecho de no contar con una red objetivo independiente `target network` implica que la estimación del valor futuro (`max Q(s', a')`) se realiza con la misma red que se está entrenando, lo que puede reducir la estabilidad del aprendizaje.

Esta decisión modifica la función de pérdida clásica del algoritmo DQN, que se definía con anterioridad en la ecuación (22) en el Capítulo 3. En la versión implementada, al prescindir de la red objetivo y del valor futuro, la función de pérdida se simplifica a:

$$\mathcal{L}(\theta) = \mathbb{E}\left[(r - Q(s, a; \theta))^2\right] \tag{22'}$$

Del mismo modo, la expresión del valor objetivo cambia respecto a la ecuación (23) clásica, para quedar simplemente como:

$$\text{Target} = r \tag{23'}$$

Con estas simplificaciones no quiere decir que el agente no aprenda. Lo que conseguimos es que sea capaz de actuar adecuadamente en el entorno centrado en la optimización de la recompensa inmediata. Este enfoque simplificado permite que el modelo aprenda a replicar las acciones esperadas según las observaciones sensoriales, aunque no desarrolla una política basada en el retorno acumulado a largo plazo como haría un agente DQN completo.

6.4.3. Política del agente

La política de decisión del agente se basa en una estrategia ϵ-greedy, que combina exploración y explotación. Con probabilidad ϵ, el agente selecciona una acción de forma aleatoria, lo que favorece la exploración del espacio de acciones. Con probabilidad $1 - \epsilon$, elige la acción que maximiza el valor estimado por la red neuronal para el estado actual. Esta combinación permite que el agente descubra nuevas estrategias en las primeras fases del entrenamiento, mientras que, a medida que ϵ disminuye, se favorece un comportamiento más determinista basado en lo aprendido.

Listing 14: Política epsilon-greedy del agente

```
def act(self, state):
    if np.random.rand() <= self.epsilon:
        return random.randrange(self.action_size)  # Acci n
           aleatoria (exploraci n)
```

```
    state = state.reshape(1, -1)
    act_values = self.model.predict(state, verbose=0)
    return np.argmax(act_values[0])  # Mejor acci n seg n el
       modelo (explotaci n)
```

La disminución progresiva del parámetro ϵ se implementa en el método `replay()`, mediante la siguiente instrucción:

Listing 15: Reducción de la tasa de exploración

```
if self.epsilon > self.epsilon_min:
    self.epsilon *= self.epsilon_decay
```

En cada iteración del entrenamiento, el valor de `epsilon` se multiplica por el factor `epsilon_decay`, lo que provoca que su valor disminuya de forma gradual hasta alcanzar un mínimo definido por `epsilon_min`. Los valores concretos de estos hiperparámetros se detallan en el siguiente apartado.

La decisión del agente entre explorar o explotar se implementa en el método `act()`, cuya lógica se es la mostrada en el código referente a la política epsilon-greedy del agente. La función `np.random.rand()` genera un número aleatorio entre 0 y 1. Si dicho número es menor o igual que `epsilon`, el agente toma una acción aleatoria, fomentando así la exploración del entorno. En caso contrario, el agente explota el conocimiento aprendido: utiliza la red neuronal para predecir los valores $Q(s, a; \theta)$ de todas las acciones posibles en el estado actual y selecciona aquella con el valor más alto, empleando la función `np.argmax()`.

6.4.4. Hiperparámetros clave

En este trabajo se han utilizado varios hiperparámetros clave que influyen directamente en el comportamiento y rendimiento del agente. A continuación, se describen los valores seleccionados y las razones que motivan su elección:

- `gamma = 0.95`: Se ha utilizado un valor relativamente alto para dar peso al posible valor futuro de cada decisión, sin llegar a valores cercanos a 1 que podrían generar inestabilidad si el entorno fuera muy largo o incierto.
- `epsilon = 0.8`: Se parte de un valor de exploración alto para fomentar que el agente pruebe diferentes acciones durante las primeras fases del entrenamiento. Esto fue especialmente importante en este caso, ya que al no usar una red objetivo ni una estimación de retorno acumulado, se requiere una exploración más intensa al principio para evitar políticas prematuramente rígidas.
- `epsilon_min = 0.01` y `epsilon_decay = 0.99`: Estos valores controlan cómo se reduce la exploración con el tiempo. El valor mínimo se fijó en 0.01 para mantener algo de aleatoriedad en fases finales, mientras que el de `decay` se ajustó de

forma empírica tras observar que valores menores hacían que el agente explotara demasiado pronto y dejara de explorar.

- `learning_rate = 0.0005`: Se ha optado por una tasa de aprendizaje baja con el fin de evitar oscilaciones bruscas en la actualización de pesos.
- `memory = deque(maxlen=2000)`: El tamaño de la memoria se ha limitado a 2000 experiencias de almacenamiento para asegurar variedad en el entrenamiento sin consumir demasiados recursos.

Estos valores no son universales, sino que han sido elegidos en base al comportamiento observado durante las primeras ejecuciones del modelo, buscando un equilibrio entre estabilidad, capacidad de exploración y velocidad de aprendizaje.

Ajustes realizados tras el primer entrenamiento

Los cambios no afectan a la estructura de la red neuronal ni a la lógica interna de la clase DQNAgent, sino que se centran exclusivamente en el ajuste de los hiperparámetros, tras detectar en las primeras ejecuciones que el agente presentaba un aprendizaje deficiente.

- `epsilon = 1.0` → `epsilon = 0.8`: Se reduce la tasa inicial de exploración. En pruebas previas se observó que un valor de 1.0, que implica explorar completamente al inicio, generaba un comportamiento demasiado aleatorio en las primeras fases, impidiendo que el agente comenzase a consolidar patrones de aprendizaje útiles.
- `epsilon_decay = 0.995` → `epsilon_decay = 0.99`: Se incrementa ligeramente la velocidad de transición desde la exploración hacia la explotación. Con el valor original, el agente tardaba demasiado en reducir su comportamiento aleatorio, lo que dificultaba la convergencia hacia una política más estable.

Estas modificaciones fueron necesarias para asegurar que el agente pudiera comenzar a aprender patrones de comportamiento de manera más efectiva desde las primeras fases del entrenamiento, reduciendo el tiempo necesario para estabilizar la política aprendida.

6.5. Entrenamiento y evaluación del agente

En este apartado se detalla el proceso completo seguido para entrenar al agente así como los criterios utilizados para evaluar su rendimiento. Primero se analizan las condiciones del entrenamiento, el número de episodios y los pasos realizados en cada uno.

6.5.1. Dinámica de entrenamiento y de episodios

En esta sección se describen los parámetros definidos para entrenar al agente y el motivo de su elección.

Listing 16: Parámetros de entrenamiento

```
state_size = X_train.shape[1]  # Debe ser 19
action_size = env.action_space.n
batch_size = 64
n_episodes = 300
output_dir = 'model_output/ai_sensor_dqn/'

if not os.path.exists(output_dir):
    os.makedirs(output_dir)

agent = DQNAgent(state_size, action_size)
```

En un primer momento se realizó el entrenamiento ejecutando 100 episodios, pero finalmente se pasó a una ejecución de 300. Este cambio se decidió porque con tan pocos episodios no era posible comprobar correctamente si el agente estaba aprendiendo o no.

El tamaño del `batch` se mantiene en 64, ya que permitió un entrenamiento estable sin requerir demasiados recursos. El número de acciones y el tamaño del estado se obtienen directamente del entorno y de los datos preprocesados.

Durante cada episodio, el agente se inicializa con `env.reset()`, interactúa con el entorno hasta que termina, y al final se entrena con la función `replay()`.

A continuación se muestra el bucle principal de entrenamiento del agente. En cada uno de los 300 episodios, el agente interactúa con el entorno hasta que el episodio termina:

Listing 17: Entrenamiento del agente por episodios

```
rewards = []
step_rewards_matrix = []

for e in range(n_episodes):
    state = env.reset()
    total_reward = 0
    done = False
    episode_step_rewards = []

    while not done:
        action = agent.act(state)
        next_state, reward, done, _ = env.step(action)
        agent.remember(state, action, reward, next_state, done)
        state = next_state
        total_reward += reward
        episode_step_rewards.append(reward)
```

```
    step_rewards_matrix.append(episode_step_rewards)
    rewards.append(total_reward)

    agent.replay(batch_size)
```

En cada episodio, el entorno se reinicia y el agente comienza desde el paso cero. A medida que avanza, toma decisiones, recibe recompensas y almacena cada experiencia. Al finalizar el episodio, se guarda la recompensa total y se entrenan los pesos de la red una única vez mediante la función `replay()`. Esto permite medir la progresión del aprendizaje a lo largo de los episodios y registrar tanto la recompensa total como la obtenida en cada paso.

Ajustes realizados tras el primer entrenamiento

En la primera ejecución del código, el agente entrenaba su red neuronal en cada paso del episodio. Sin embargo, se terminó optando por entrenar únicamente una vez al finalizar cada episodio. Este cambio se ha realizado con el objetivo de reducir el coste computacional y aumentar la estabilidad del aprendizaje, evitando actualizaciones excesivamente frecuentes que puedan introducir ruido o sobreajuste.

A continuación, se muestra la comparación entre ambos enfoques:

Listing 18: Primer entrenamiento (entrena en cada paso)

```
for time in range(1):
    ...
    agent.replay(batch_size)  # Entrena tras cada paso
```

Listing 19: Entrenamiento modificado (entrena una vez por episodio)

```
while not done:
    ...
agent.replay(batch_size)  # Entrena solo al final del episodio
```

ANÁLISIS DE RESULTADOS

CAPÍTULO 7

7. Análisis de resultados

7.1. Resultados obtenidos de la red neuronal para la predicción de la recompensa

En este capítulo se presentan y analizan los resultados obtenidos tras el entrenamiento del modelo de red neuronal definido para la predicción de la recompensa.

Durante el entrenamiento, el modelo mostró una mejora progresiva en la función de pérdida (MSE) y en la métrica de evaluación (MAE) tanto en los datos de entrenamiento como en los de validación. En las primeras épocas, la pérdida era elevada, pero disminuyó de forma constante hasta estabilizarse alrededor de la época 25.

A continuación, se muestra un fragmento representativo del registro de entrenamiento:

Listing 20: Resumen del entrenamiento

```
Epoch 1/100
loss: 41.08 - mae: 5.31 - val_loss: 7.30 - val_mae: 2.25
Epoch 2/100
loss: 6.80 - mae: 2.13 - val_loss: 5.08 - val_mae: 1.89
Epoch 3/100
loss: 4.88 - mae: 1.83 - val_loss: 4.91 - val_mae: 1.85
...
Epoch 27/100
loss: 1.48 - mae: 0.97 - val_loss: 1.05 - val_mae: 0.83
Epoch 28/100
loss: 1.49 - mae: 0.96 - val_loss: 1.07 - val_mae: 0.83
Epoch 29/100
loss: 1.40 - mae: 0.94 - val_loss: 1.23 - val_mae: 0.89
```

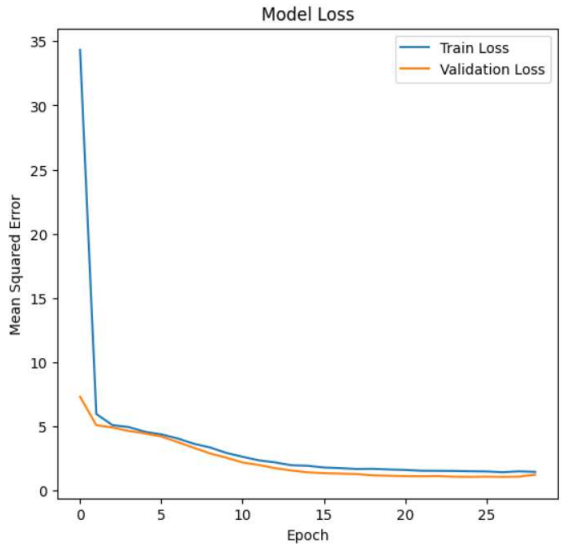

Figura 30: Evolución de la función de pérdida (MSE)

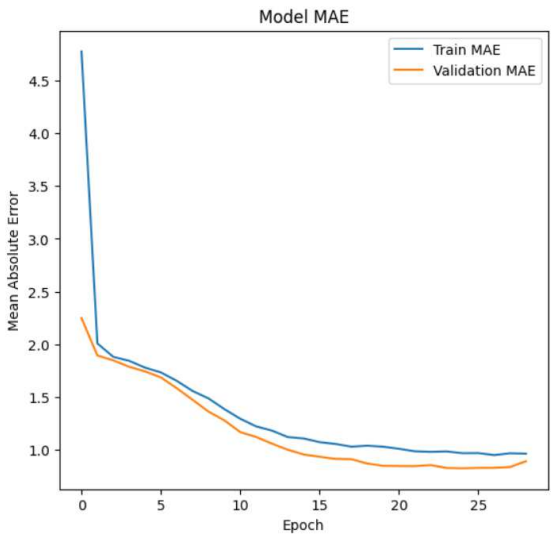

Figura 31: Evolución del error medio absoluto (MAE)

Como se observa en las gráficas de las Figuras 30 y 31, la línea azul representa el error en el conjunto de entrenamiento, es decir, el error que comete el modelo sobre los datos que ha utilizado para aprender. Por su parte, la línea naranja muestra el error en el conjunto de validación, correspondiente a datos que el modelo no ha visto durante el entrenamiento.

Ambas curvas descienden de forma similar y se mantienen próximas a lo largo de las épocas, lo que indica que el modelo no solo ha aprendido correctamente, sino que también generaliza bien a datos no vistos.

7.2. Resultados obtenidos del aprendizaje del agente original

Tras el entrenamiento del agente, se obtuvieron una serie de gráficas con las que podemos evaluar el proceso de aprendizaje y los cambios de su comportamiento en cada episodio. Para el análisis de las gráficas se ha optado por ver principalmente la evolución de la recompensa por episodios.

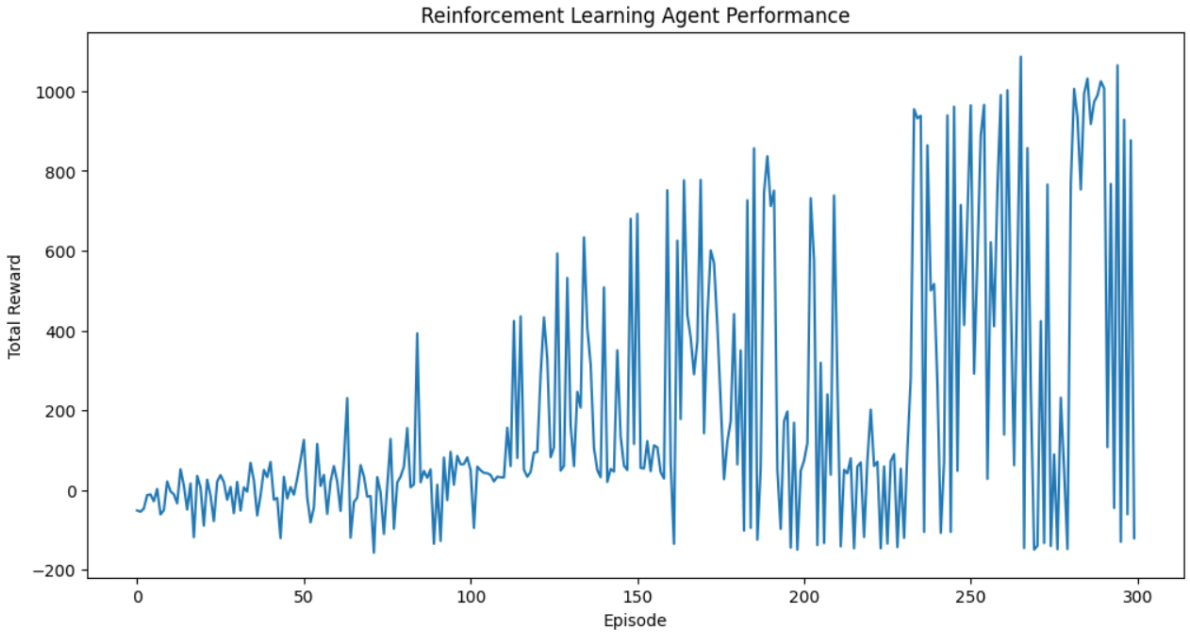

Figura 32: Evolución de la recompensa que recibe el agente original durante los 300 episodios

En la gráfica, mostrada en la Figura 32, se puede ver claramente el proceso de aprendizaje del agente. En los primeros 100 episodios, las recompensas son en su mayoría bajas, situándose entre 100 y 200, con una proporción elevada de valores negativos. A medida que avanzan los episodios, el rendimiento mejora notablemente, llegando incluso a alcanzarse una recompensa de 1086,25 en el episodio 266. Finalmente, el agente obtiene una recompensa inmediata media de 200,98 a lo largo de los 300 episodios. El valor final registrado de ϵ es de 0,04, lo que indica que el agente solo toma decisiones aleatorias en un 4 % de los casos, mientras que el 96 % de sus acciones se basan en lo aprendido a través del entrenamiento.

Este comportamiento también puede observarse de forma cuantitativa en la Tabla 6, donde se presenta el porcentaje de recompensas positivas y negativas acumuladas por tramos de 10 episodios. Se observa una clara tendencia ascendente en la proporción de recompensas positivas.

Tabla 6: Porcentaje de recompensas positivas y negativas por bloques

Episodios	**Recompensas Positivas (%)**	**Recompensas Negativas (%)**
0–10	20.00	80.00
0–20	35.00	65.00
0–30	40.00	60.00
0–40	45.00	55.00
0–50	46.00	54.00
0–60	48.33	51.67
0–70	48.57	51.43
0–80	48.75	51.25
0–90	53.33	46.67
0–100	56.00	44.00
0–110	59.09	40.91
0–120	62.50	37.50
0–130	65.38	34.62
0–140	67.86	32.14
0–150	70.00	30.00
0–160	71.88	28.12
0–170	72.94	27.06
0–180	74.44	25.56
0–190	74.21	25.79
0–200	74.00	26.00
0–210	74.29	25.71
0–220	74.09	25.91
0–230	73.91	26.09
0–240	74.17	25.83
0–250	74.40	25.60
0–260	75.38	24.62
0–270	75.56	24.44
0–280	74.64	25.36
0–290	75.52	24.48
0–300	75.00	25.00

Cabe destacar también que, a pesar de la mejora general, el agente sigue presentando algunas recaídas significativas en episodios avanzados. Estas caídas puntuales reflejan

cierta irregularidad en el comportamiento, por lo que se debe llevar a cuestión si se tratan de fallos significativos o es común en agentes que aprenden mediante exploración, especialmente en entornos complejos.

Sin embargo, se puede llegar a una primera conclusión: los cambios realizados tanto en el entorno como en el diseño del agente y en la estrategia de entrenamiento han permitido que el agente evolucione adecuadamente. Aunque su rendimiento no es completamente estable, se observa una mejora sostenida en su comportamiento, lo que indica que ha aprendido a actuar de forma más eficaz a lo largo de los episodios.

7.2.1. Análisis de los resultados

Además de las métricas utilizadas anteriormente para analizar el comportamiento del agente, se han empleado otras herramientas complementarias que permiten observar con mayor detalle su evolución durante el proceso de aprendizaje.

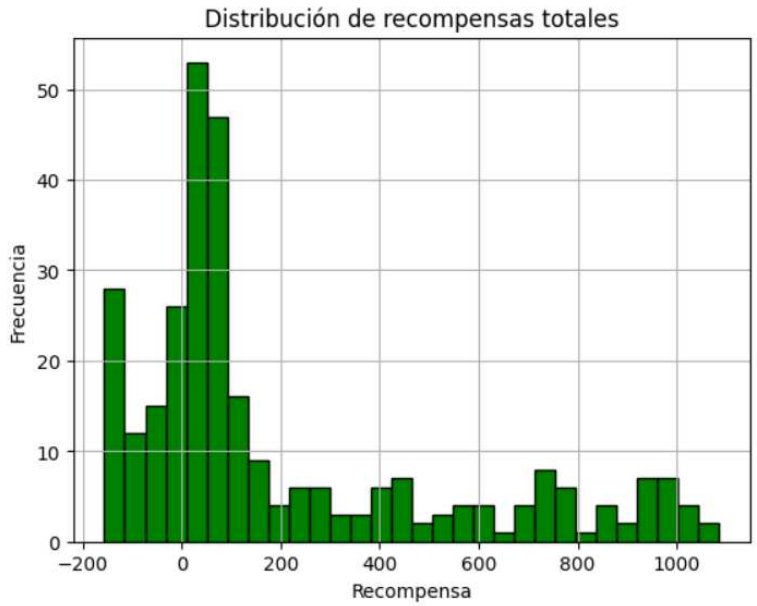

Figura 33: Distribución de recompensas totales obtenidas por el agente

La Figura 33 muestra un histograma con la distribución de recompensas totales obtenidas por el agente a lo largo de los episodios. Se pueden extraer varias conclusiones: en primer lugar, existe una clara concentración de recompensas en el rango entre 0 y 200, con un pico especialmente pronunciado entre 50 y 100, lo cual justifica que la media final de recompensas sea de 200,98.

También se observa que, aunque predominan las recompensas positivas alcanzando valores más altos, hay una proporción considerable de episodios con recompensas negativas, distribuidas principalmente entre -200 y 0. Esta acumulación de resultados negativos indica que, a pesar del aprendizaje progresivo, el comportamiento del agente todavía presenta cierta inestabilidad en determinadas fases del entrenamiento.

Figura 34: Media móvil de las recompensas

En la Figura 34 se muestra la media móvil de las recompensas, calculada tomando el promedio de los últimos 10 episodios, desplazándose un episodio en cada paso. Se observa una tendencia general creciente desde el episodio 0 hasta aproximadamente el 260, lo que indica que el agente mejora progresivamente su rendimiento.

Sin embargo, también se aprecian picos muy altos seguidos de caídas pronunciadas, especialmente en la parte final. Aunque la recompensa media llega a alcanzar casi 1000 en uno de los episodios, estos valores no se mantienen estables. Por tanto, se puede concluir que el agente ha logrado aprender, pero aún presenta ciertos puntos de inestabilidad en su comportamiento.

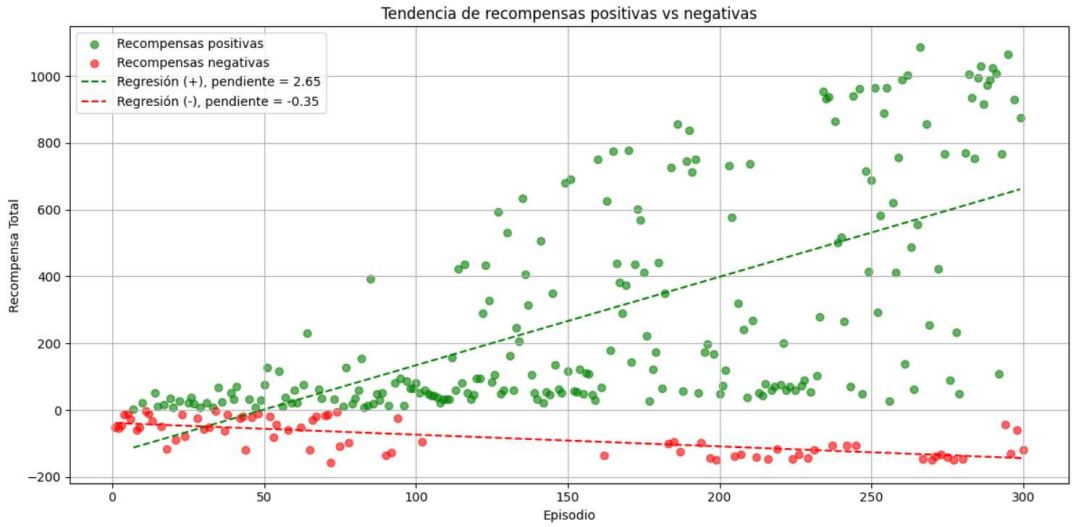

Figura 35: Tendencia de recompensas positivas y negativas

La Figura 35 muestra una comparativa entre la tendencia de las recompensas positivas

y negativas a lo largo del entrenamiento. Se observa que la regresión de las recompensas positivas tiene una pendiente claramente creciente (2.65), mientras que la correspondiente a las recompensas negativas presenta una pendiente mucho menor y negativa (-0.35). Esto indica que el número y magnitud de las recompensas positivas tiende a aumentar, mientras que las negativas disminuyen, lo que refuerza la conclusión de que el agente está aprendiendo de forma progresiva.

Este resultado se ve aún más reforzado en la Figura 36, donde se representan las pendientes calculadas en bloques de 50 episodios. Esta segmentación permite observar cómo evoluciona el comportamiento del agente en distintas fases del entrenamiento.

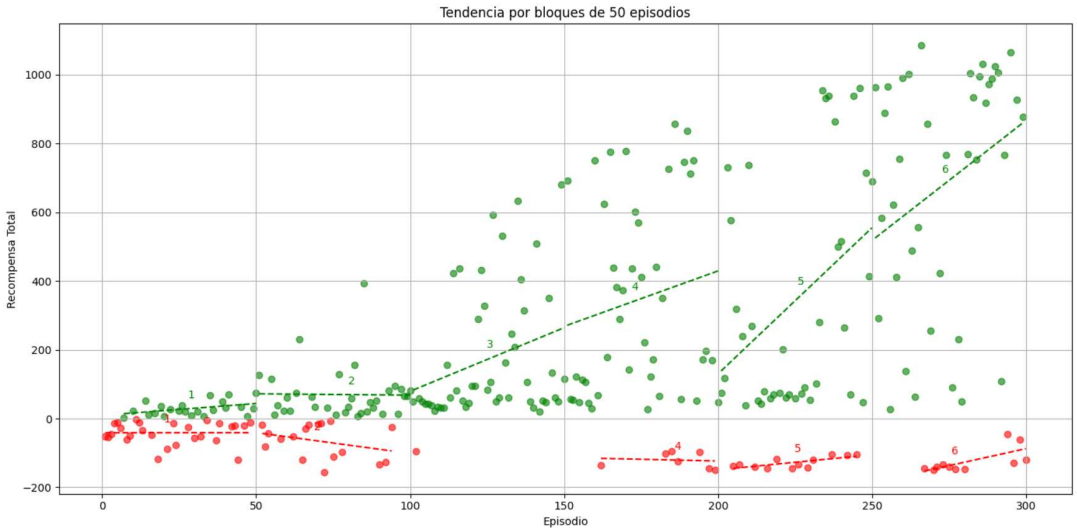

Figura 36: Tendencia de la recompensa por bloques de 50 episodios

En la Figura 36 se aprecia cómo, a partir del bloque 3, las pendientes de las recompensas positivas aumentan notablemente, alcanzando valores como 3.70 en el bloque 3 y hasta 8.50 en el bloque 5. Este crecimiento refleja un salto cualitativo en el rendimiento del agente. En contraste, las pendientes de las recompensas negativas son mucho más bajas, como -0.21 en el bloque 4, o incluso positivas, como 1.98 en el bloque 6, lo que indica que las penalizaciones se reducen o estabilizan en fases más avanzadas.

Tabla 7: Pendientes por bloque de 50 episodios

Bloque	**Pendiente (+)**	**Pendiente (−)**
1	0.69	0.00
2	-0.08	-1.21
3	3.70	–
4	3.23	-0.21
5	8.50	0.88
6	6.98	1.98

En conjunto, estos análisis muestran que el comportamiento del agente mejora progresi-

vamente, especialmente a partir del segundo tercio del entrenamiento. Aunque persisten algunas recompensas negativas, su frecuencia e intensidad disminuyen o se estabilizan, mientras que el número de episodios exitosos y de alta recompensa crece de forma sostenida.

7.3. Experimentación del modelo

A continuación, tal y como se definió en un principio como uno de los objetivos del Trabajo de Fin de Grado, se procede al análisis del modelo modificando distintos parámetros y comparando los resultados obtenidos con los del modelo original. Entre dichos parámetros se encuentran las funciones de activación, los optimizadores y el conjunto de variables utilizadas como entrada, centrando especialmente la atención en aquellas que mostraron mayor influencia sobre la variable de recompensa, tal y como se analizó en el Capítulo 4.

La elección de experimentar con estas variables se debe a su impacto directo en el rendimiento del agente, así como a su amplia utilización y reconocimiento en el ámbito del aprendizaje profundo. Tanto las funciones de activación como los optimizadores son considerados parámetros clave en el diseño de redes neuronales, y son objeto habitual de análisis y ajuste en numerosos trabajos, debido a la influencia que ejercen sobre la capacidad de aprendizaje, la velocidad de convergencia y la estabilidad del entrenamiento. Las variables de entrada determinan la calidad de la información disponible para la toma de decisiones.

Para la comparativa se ha graficado únicamente 150 episodios del modelo original como podemos ver en la Figura 37, debido al elevado coste computacional y al tiempo requerido para simular los 300 episodios completos. Esta decisión no compromete el análisis del rendimiento del agente, ya que, tal y como se observó en el modelo original, a partir del episodio 100 comienzan a producirse variaciones significativas en el comportamiento, lo que permite evaluar si el agente está aprendiendo de manera efectiva.

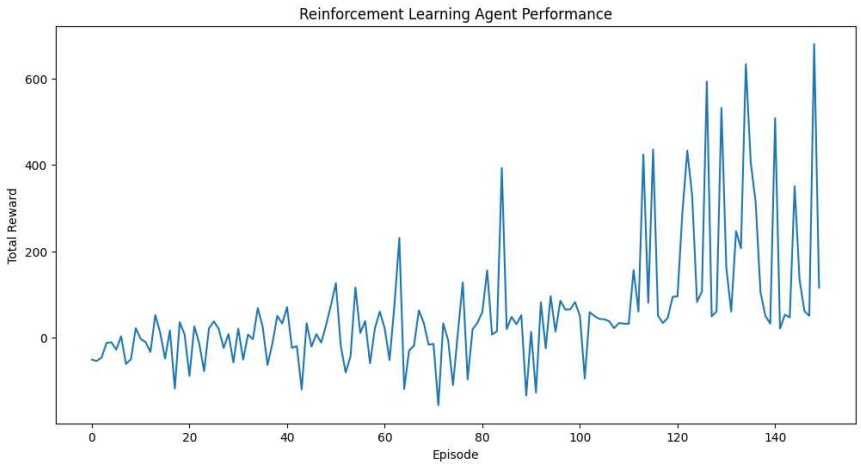

Figura 37: Evolución de la recompensa que recibe el agente original durante 150 episodios

Se realizaron modificaciones tanto en la red neuronal utilizada para evaluar el rendimiento

de los datos en función de la recompensa, como en la red empleada por el agente durante el proceso de aprendizaje por refuerzo.

7.3.1. Variación de las funciones de activación

Puesto que la función de activación del modelo `ses_activation` está compuesta por dos funciones conocidas, `sigmoid` y `ReLU`, se decidió experimentar con ambas por separado para evaluar el rendimiento del agente en cada caso y compararlo con el del modelo original. Además, se incluye una comparación adicional con `ReLU6`, con el objetivo de analizar su impacto, tal y como se expone en el apartado 3.3.1 del Capítulo 3.

Sigmoid

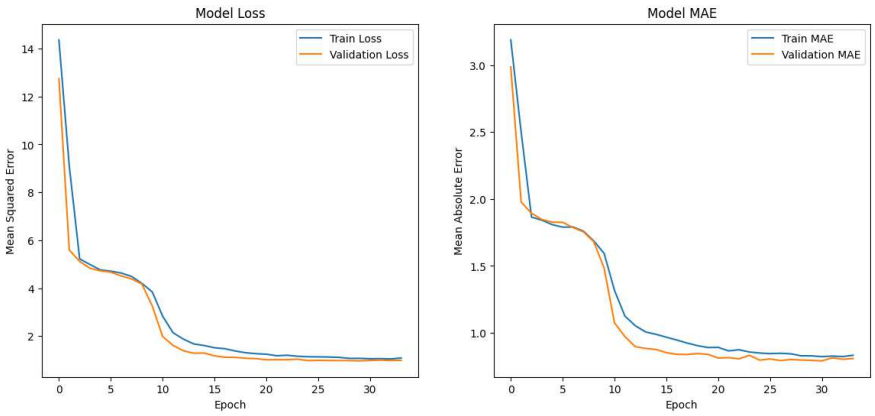

Figura 38: Evolución del error con la función de activación Sigmoid

Tal y como se observa en la Figura 38, donde se presentan dos gráficas una para la función de pérdida (MSE) y otra para el error medio absoluto (MAE). Ambas curvas descienden de forma similar y se mantienen próximas a lo largo de las épocas. Los errores finales son muy bajos, lo que indica que, incluso utilizando únicamente la función `sigmoid`, el modelo es capaz de aprender correctamente.

En cuanto al entrenamiento del agente, tal y como se muestra en la Figura 39, se mantiene un comportamiento sostenido similar al del modelo original, aunque con picos menos pronunciados. A partir del episodio 100 se observa un crecimiento considerable de la recompensa acumulada.

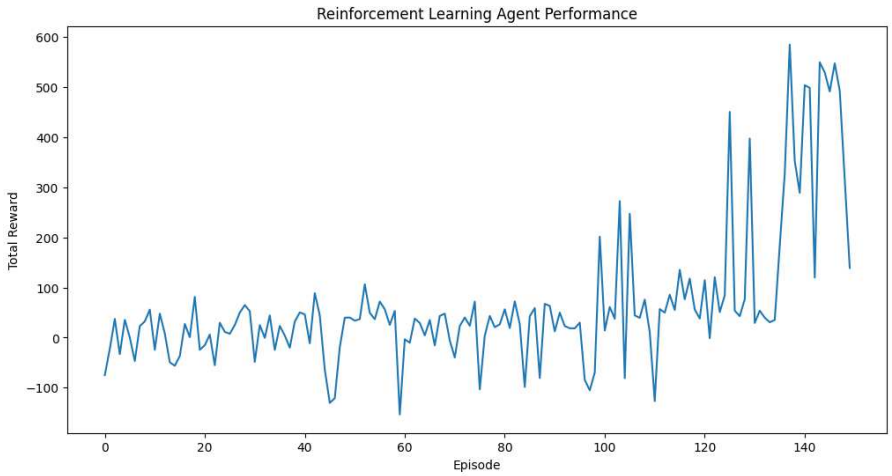

Figura 39: Evolución de la recompensa que recibe el agente con sigmoid

Tabla 8: Porcentaje de recompensas positivas y negativas con sigmoid por bloques

Bloque de episodios	**Recompensas Positivas (%)**	**Recompensas Negativas (%)**
0–10	60.00	40.00
0–20	55.00	45.00
0–30	63.33	36.67
...	...	...
0–130	73.08	26.92
0–140	75.00	25.00
0–150	76.67	23.33

En la Tabla 8 [5], donde se muestran los tres primeros bloques de episodios y los tres últimos, se observa un cambio claro en la distribución de recompensas: al principio predominan las negativas, mientras que hacia el final la mayoría son positivas.

Este mismo patrón se ve reforzado en la Figura 40, donde se analiza la pendiente de la tendencia de recompensas positivas y negativas. Ambas pendientes son similares a las del modelo original, lo que permite concluir que el agente también aprende progresivamente con esta configuración de activación.

[5]Se ha optado por acortar estas tablas en adelante para ahorrar espacio, mostrando únicamente una selección representativa de los tramos.

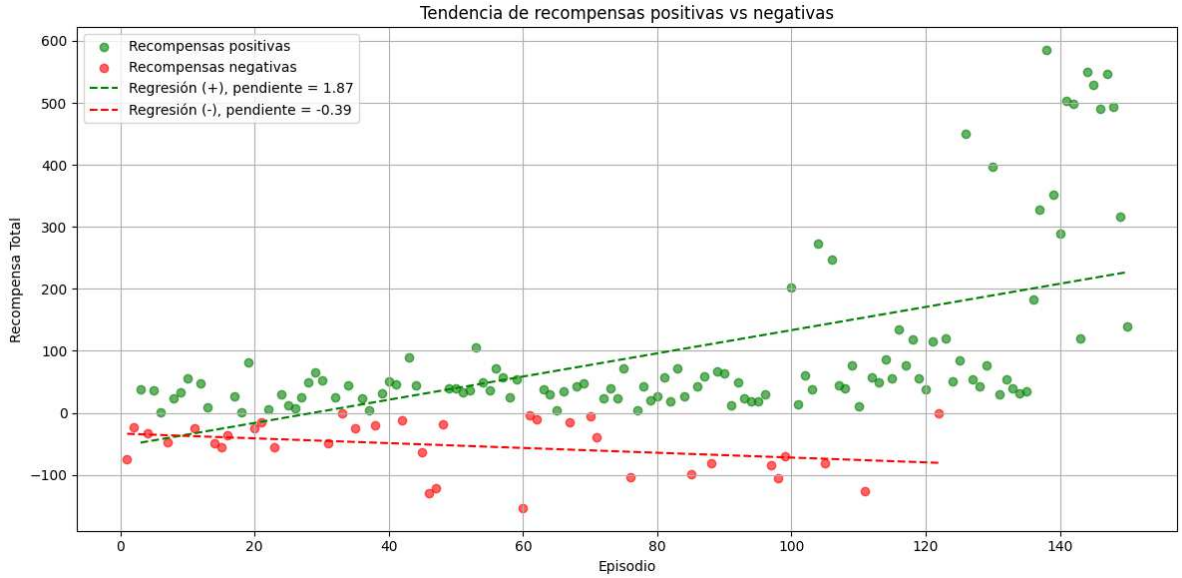

Figura 40: Tendencia de recompensas positivas y negativas con sigmoid

ReLU

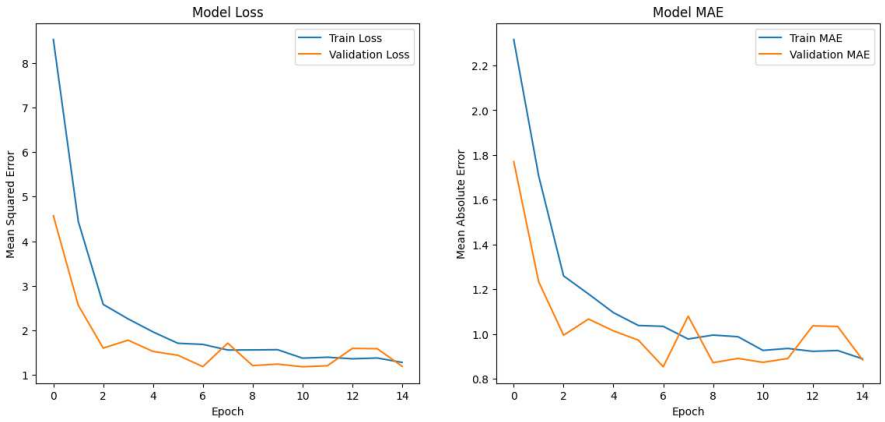

Figura 41: Evolución del error con la función de activación ReLU

En la Figura 41 se muestran dos gráficas correspondientes a la evolución de la función de pérdida (MSE) y el error medio absoluto (MAE) durante el entrenamiento del modelo con la función de activación `ReLU`. A diferencia de los modelos anteriores, se observa una mayor divergencia entre ambas métricas, lo que sugiere una mayor inestabilidad en el aprendizaje. Aunque ambas curvas tienden a disminuir, la diferencia de comportamiento indica un ajuste menos fino del modelo.

A pesar de ello, el agente logra aprender, como se puede observar en la Figura 42, donde se representa la evolución de la recompensa total por episodio. En este caso, el aprendizaje es más irregular, con picos de alto rendimiento seguidos de caídas pronunciadas. En comparación con los modelos entrenados con `sigmoid` o con la función personalizada original, el rendimiento general con `ReLU` es inferior.

Figura 42: Evolución de la recompensa que recibe el agente con ReLU

Este comportamiento se ve reflejado también en los porcentajes de recompensas positivas y negativas por bloques de 10 episodios. Como se muestra en la Tabla 9, el porcentaje de recompensas positivas se mantiene por debajo de los obtenidos con otras funciones de activación, aunque presenta una ligera mejora progresiva.

Tabla 9: Porcentaje de recompensas positivas y negativas con ReLU por bloques

Bloque de episodios	**Recompensas Positivas (%)**	**Recompensas Negativas (%)**
0–10	20.00	80.00
0–20	20.00	80.00
0–30	23.33	76.67
...	...	...
0–130	56.15	43.85
0–140	59.29	40.71
0–150	61.33	38.67

Por último, en la Figura 43, se analiza la tendencia de las recompensas positivas y negativas. Aunque el comportamiento del agente sigue siendo menos estable, se observa una pendiente positiva de 1,99 y una pendiente negativa casi nula de -0,25. Esto sugiere que, a pesar de su menor rendimiento general, el agente logra mejorar con el tiempo.

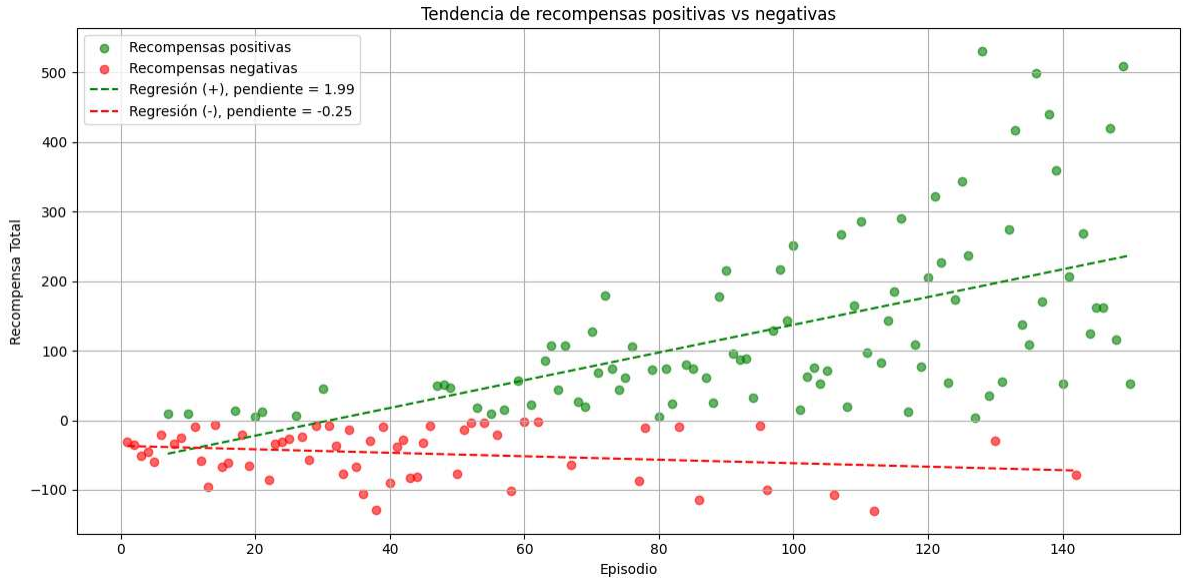

Figura 43: Tendencia de recompensas positivas y negativas con ReLU

ReLU6

Como análisis adicional, se incorporó una variante de la función `ReLU`, conocida como `ReLU6`, que en la literatura se considera una posible mejora respecto a la original al limitar su salida máxima a 6, lo que puede ayudar a prevenir explosiones de activación y mejorar la estabilidad del entrenamiento [19].

Sin embargo, tal y como se observa en la Figura 44, el agente no llega a mostrar un comportamiento de aprendizaje claro. A lo largo de los 150 episodios no se aprecia una tendencia ascendente sostenida en las recompensas, ni una disminución consistente en los valores negativos. El rendimiento se mantiene altamente oscilante y con valores en su mayoría bajos, lo que indica que esta función de activación no ha favorecido el aprendizaje en este caso concreto.

Debido a los resultados obtenidos, no se han considerado necesarias más gráficas para este experimento, ya que la conclusión principal es que el agente no logra aprender una política eficaz con `ReLU6`.

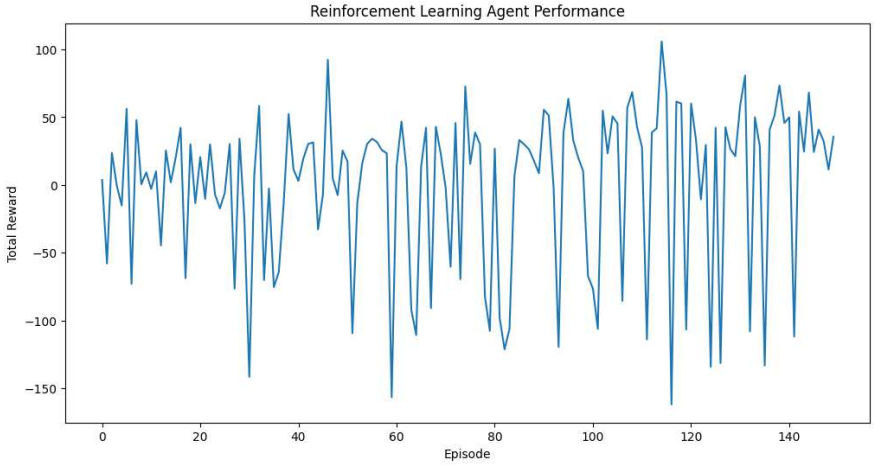

Figura 44: Evolución de la recompensa que recibe el agente con ReLU6

7.3.2. Comparación de los optimizadores

Como en el modelo original se decidió utilizar el optimizador `Adam`, uno de los más usados, se realizó un estudios de que optimzadores podían ser interesantes y se llegó a la conslusión que se podrían comparar con `RMSprop` y `SDG`.

RMSprop

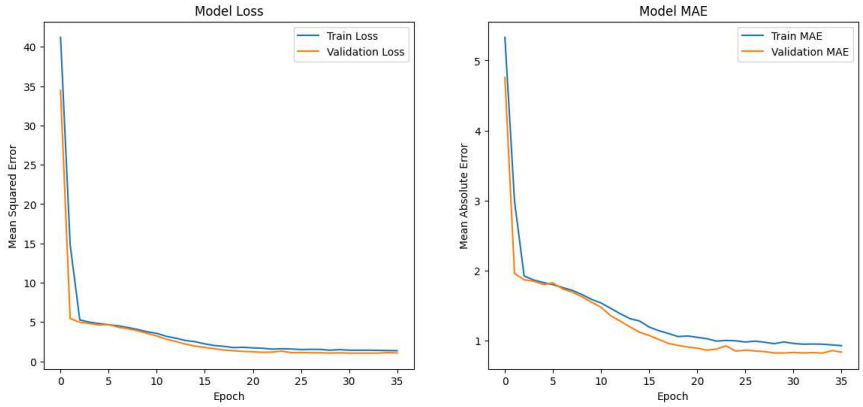

Figura 45: Evolución del error con el optimizador RMSprop

Al igual que en los apartados anteriores, en la Figura 45 se muestran dos gráficas correspondientes a la evolución de la función de pérdida (MSE) y del error medio absoluto (MAE) durante el entrenamiento del modelo, en este caso utilizando el optimizador `RMSprop`. Se puede comprobar que ambos errores evolucionan de forma muy similar a lo largo de las épocas, lo que indica que el modelo está aprendiendo correctamente desde una perspectiva supervisada.

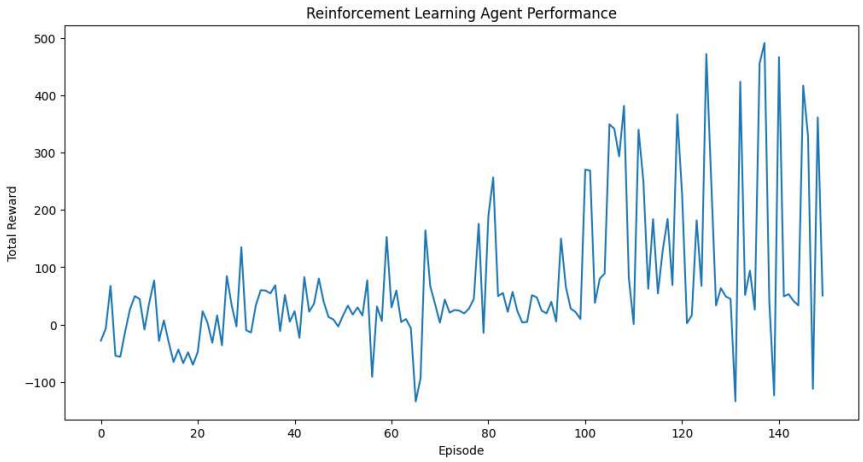

Figura 46: Evolución de la recompensa que recibe el agente con RMSprop

En cuanto al entrenamiento del agente, la Figura 46 muestra que el comportamiento aprendido es eficaz, aunque presenta ciertas recaídas pronunciadas en distintos momentos. Esto sugiere un proceso de aprendizaje algo inestable, aunque con resultados finales positivos.

Tabla 10: Porcentaje de recompensas positivas y negativas con RMSprop por bloques

Bloque de episodios	**Recompensas Positivas (%)**	**Recompensas Negativas (%)**
0–10	40.00	60.00
0–20	35.00	65.00
0–30	43.33	56.67
...	...	...
0–130	79.23	20.77
0–140	79.29	20.71
0–150	80.00	20.00

Tal y como se observa en la Tabla 10, el porcentaje de recompensas positivas aumenta de forma progresiva durante el entrenamiento, alcanzando cifras cercanas al 80 % al final de los 150 episodios. Este resultado se refuerza con la Figura 47, en la que se representa la tendencia de recompensas positivas y negativas. La pendiente positiva confirma que el agente mejora su rendimiento global, a pesar de la inestabilidad puntual observada en algunos episodios.

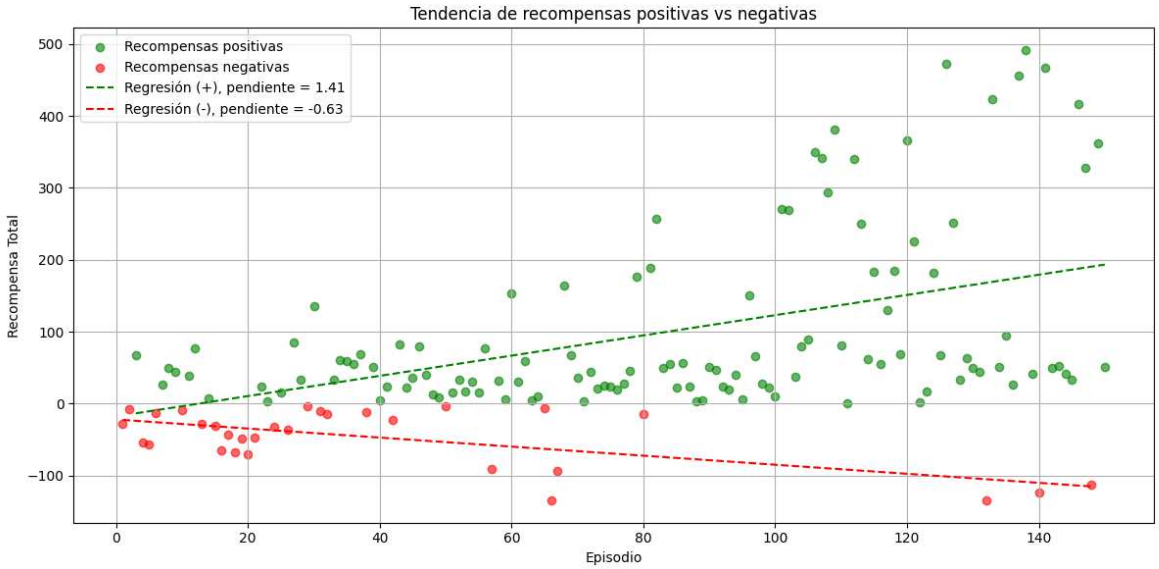

Figura 47: Tendencia de recompensas positivas y negativas con RMSprop

SDG

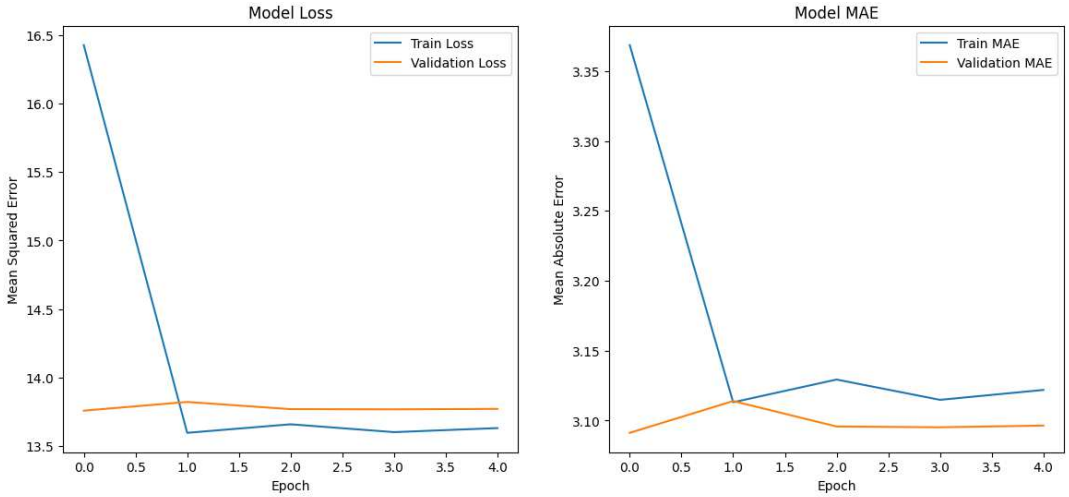

Figura 48: Evolución del error con el optimizador SGD

En la Figura 48 se muestran las gráficas de la función de pérdida (MSE) y del error medio absoluto (MAE) durante el entrenamiento supervisado utilizando el optimizador `SGD`. El modelo llegó a 5 cuando dejó de mejorar. Esto es poco para tareas con datos sensoriales y aprendizaje por refuerzo, limitando así su rendimiento.

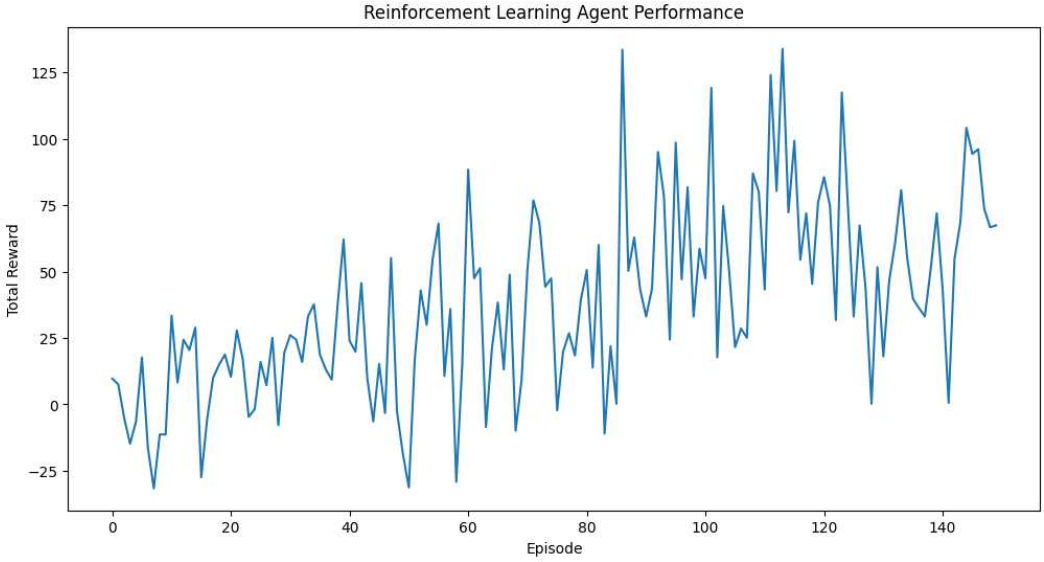

Figura 49: Evolución de la recompensa que recibe el agente con SGD

En cuanto al comportamiento del agente, la Figura 49 muestra una progresión positiva en la recompensa total obtenida por episodio. Aunque el agente parte de una situación muy inestable, con recompensas bajas y dispersas, a partir de la segunda mitad del entrenamiento se empieza a observar una mejora sostenida en el rendimiento, con picos más altos y una menor frecuencia de caídas extremas, pero siendo todavía muy inestable.

Tabla 11: Porcentaje de recompensas positivas y negativas con SGD por bloques

Bloque de episodios	**Recompensas Positivas (%)**	**Recompensas Negativas (%)**
0–10	30.00	70.00
0–20	55.00	45.00
0–30	60.00	40.00
...	...	...
0–130	83.08	16.92
0–140	84.29	15.71
0–150	85.33	14.67

Tal y como se muestra en la Tabla 11, el porcentaje de recompensas positivas crece de manera progresiva y consistente a lo largo del entrenamiento. En los últimos bloques, supera el 85 %, lo cual indica un aprendizaje efectivo.

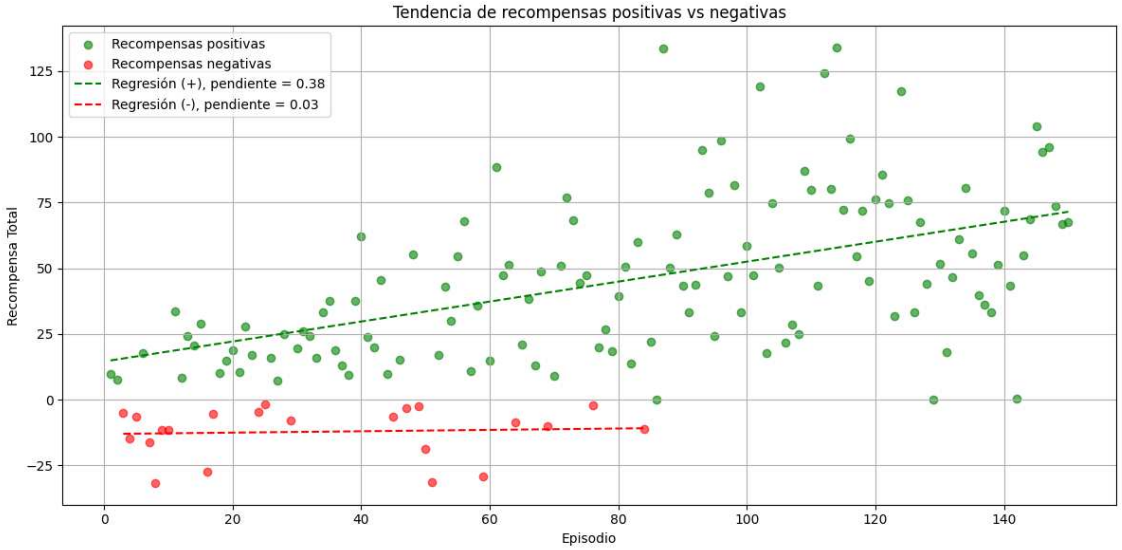

Figura 50: Tendencia de recompensas positivas y negativas con SGD

Esta mejora se ve confirmada en la Figura 50, donde se observa una tendencia ascendente clara en las recompensas positivas, con una pendiente de 0,38. La pendiente de las recompensas negativas es prácticamente nula de 0,03, lo que refleja una disminución en su impacto a lo largo del tiempo. En conjunto, estos resultados indican que el optimizador `SGD` permite al agente aprender de forma progresiva, aunque con un ritmo más lento y menos estable.

7.3.3. Impacto de las variables de entrada sobre la recompensa

En esta parte se realizará un análisis que continúa con el descrito en el Capítulo 4, donde se podía observar que existían algunas columnas más influyentes que otras con respecto a la recompensa. Estas eran, en concreto: `radar_signal_strength` y `threat_level`, correspondientes con la amplitud de la señal y el nivel de amenaza.

A diferencia de los apartados anteriores, el cambio en este caso se produjo en la unión de las columnas mediante la función `merge`, donde se indicaron explícitamente las columnas que se querían incluir para el análisis.

El estudio se ha dividido en dos partes: un primer análisis con las columnas que afectan significativamente a la recompensa, y un segundo análisis con aquellas que no presentan correlación con dicha variable.

Para estos casos se han analizado un mayor número de episodios, en concreto 240, con el objetivo de observar con mayor claridad el comportamiento del agente y reforzar el análisis previo del Capítulo 4. No obstante, no se ha llegado a los 300 episodios utilizados en otros experimentos debido al elevado coste computacional que suponía realizar el entrenamiento.

Análisis de las columnas con influencia sobre reward

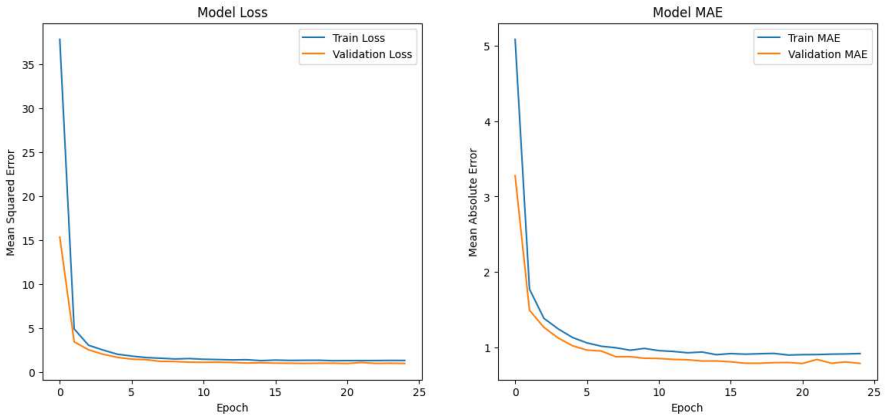

Figura 51: Evolución del error con las columnas influyentes

En la Figura 51 se muestra la evolución del error al entrenar el modelo exclusivamente con las columnas que previamente se identificaron como influyentes en la variable de recompensa. Tanto la pérdida (MSE) como el error medio absoluto (MAE) descienden de manera pronunciada durante las primeras épocas y se estabilizan rápidamente, mostrando una trayectoria decreciente suave y sin signos de sobreajuste. Este comportamiento es indicativo de que las variables seleccionadas aportan información significativa para predecir la recompensa.

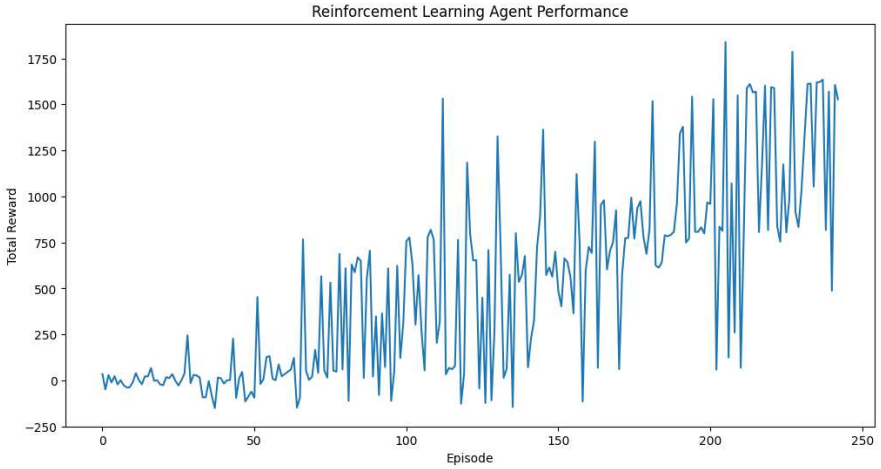

Figura 52: Evolución de la recompensa que recibe el agente con las columnas influyentes

En cuanto a la recompensa que recibe el agente al trabajar únicamente con las columnas influyentes, lo más destacable son los altos valores alcanzados, llegando incluso a superar los 1750. Esto contrasta claramente con los experimentos anteriores, donde las recompensas rara vez superaban los 1000, incluso en el modelo original. Además, se observa un comportamiento general estable por parte del agente, con caídas menos pronunciadas y un patrón que indica que está aprendiendo de forma sólida.

Tabla 12: Porcentaje de recompensas positivas y negativas de las columnas influyentes por bloques

Episodios	Recompensas Positivas (%)	Recompensas Negativas (%)
0–10	40.00	60.00
0–20	50.00	50.00
0–30	53.33	46.67
0–40	52.50	47.50
...	...	...
0–210	82.38	17.62
0–220	83.18	16.82
0–230	83.91	16.09
0–240	84.58	15.42

Estos datos los de la Tabla 12, junto con la gráfica de la media móvil de recompensas mostrada en la Figura 53, refuerzan la idea de que el agente ha aprendido de forma clara y efectiva. Se puede observar una mejora progresiva sin grandes retrocesos, lo que implica estabilidad y consolidación del comportamiento aprendido.

Figura 53: Media móvil de las recompensas con columnas influyentes

Por último, la Figura 54 muestra las pendientes de evolución de recompensas positivas y negativas. La pendiente positiva es de 5,68, considerablemente superior a la observada en el resto de experimentos, mientras que la pendiente negativa se mantiene prácticamente nula (-0,72). Esto representa una señal clara del buen rendimiento del agente y de la relevancia de estas variables seleccionadas.

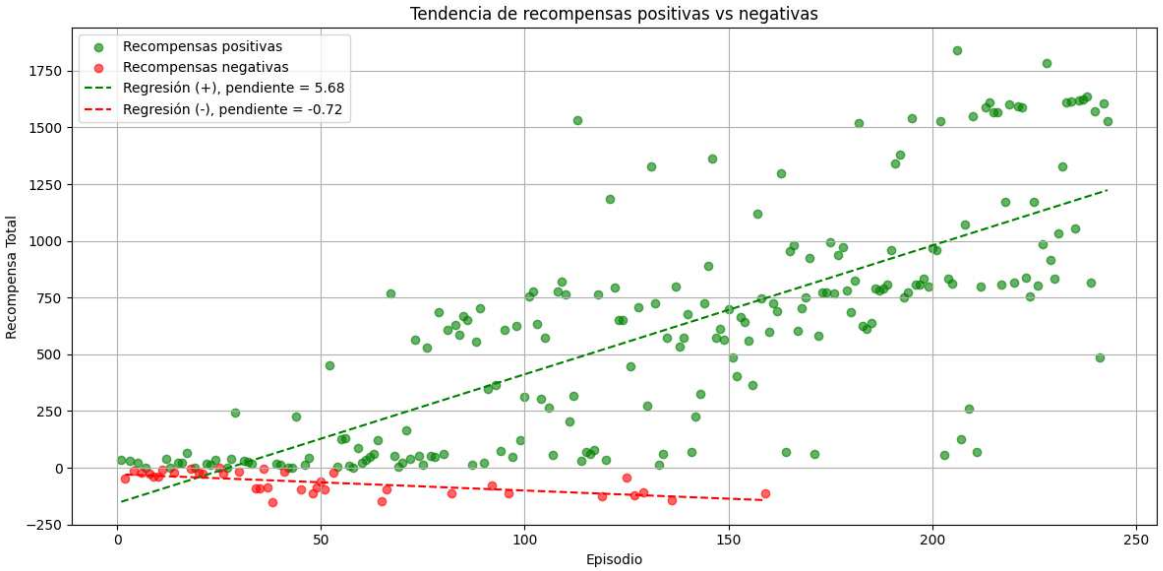

Figura 54: Tendencias de recompensas positivas y negativas con columnas influyentes

Análisis de las columnas sin influencia sobre reward

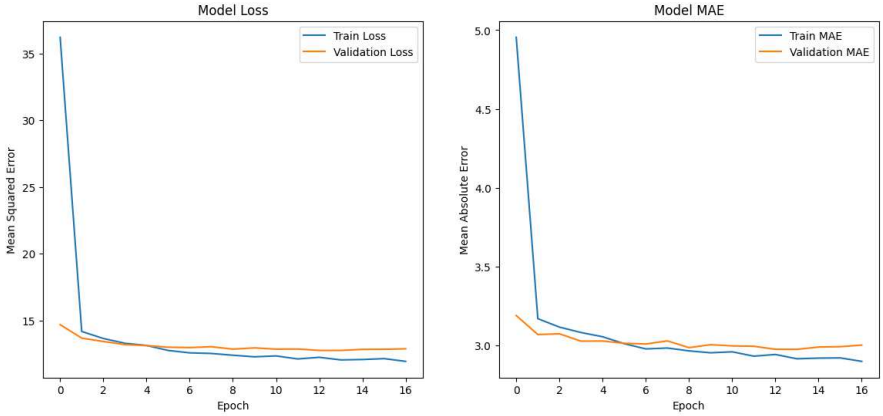

Figura 55: Evolución del error con las columnas no influyentes

La Figura 55 muestra el entrenamiento del modelo utilizando únicamente aquellas columnas que no presentaban correlación significativa con la variable de recompensa. Aunque las curvas de MSE y MAE también muestran un descenso inicial, su evolución es mucho más lenta y sus valores se estabilizan en un punto más alto que en el caso anterior. Además, se observa que la diferencia entre el conjunto de entrenamiento y el de validación es más amplia y presenta ligeras oscilaciones, lo cual puede ser un indicio de menor capacidad de generalización.

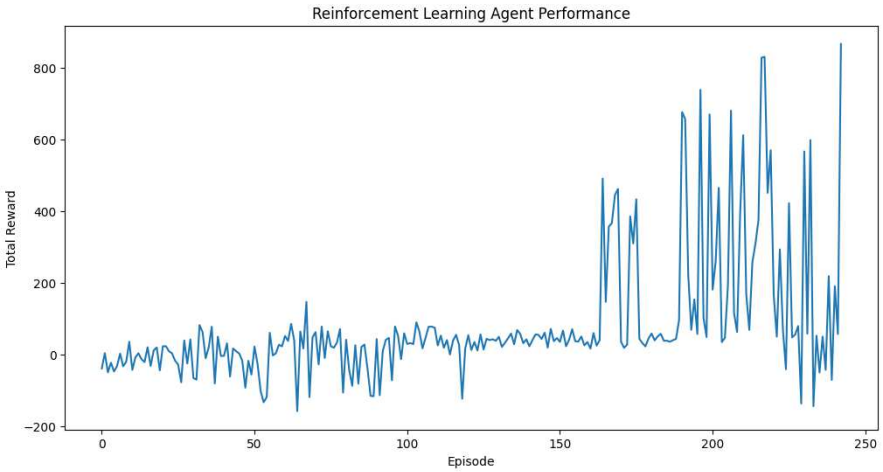

Figura 56: Evolución de la recompensa que recibe el agente con las columnas no influyentes

A diferencia de los resultados obtenidos con las columnas influyentes, en este caso, tal y como se observa en la Figura 56, al agente le cuesta más empezar a aprender. Se mantiene prácticamente estable en recompensas cercanas a 0 hasta pasados los 150 episodios, donde se produce un cambio significativo y comienza a alcanzar valores de recompensa que superan incluso los 800 puntos.

Tabla 13: Porcentaje de recompensas positivas y negativas de las columnas no influyentes por bloques

Episodios	**Recompensas Positivas (%)**	**Recompensas Negativas (%)**
0–10	30.00	70.00
0–20	35.00	65.00
0–30	43.33	56.67
0–40	45.00	55.00
...	...	...
0–210	77.14	22.86
0–220	78.18	21.82
0–230	78.26	21.74
0–240	77.50	22.50

En la Tabla 13 se puede observar que al comienzo del entrenamiento predominan las recompensas negativas, siendo las positivas únicamente el 30 % en el primer bloque de episodios. Sin embargo, hacia el final del entrenamiento, estas cifras se invierten, alcanzando porcentajes de recompensas positivas cercanos al 80 %.

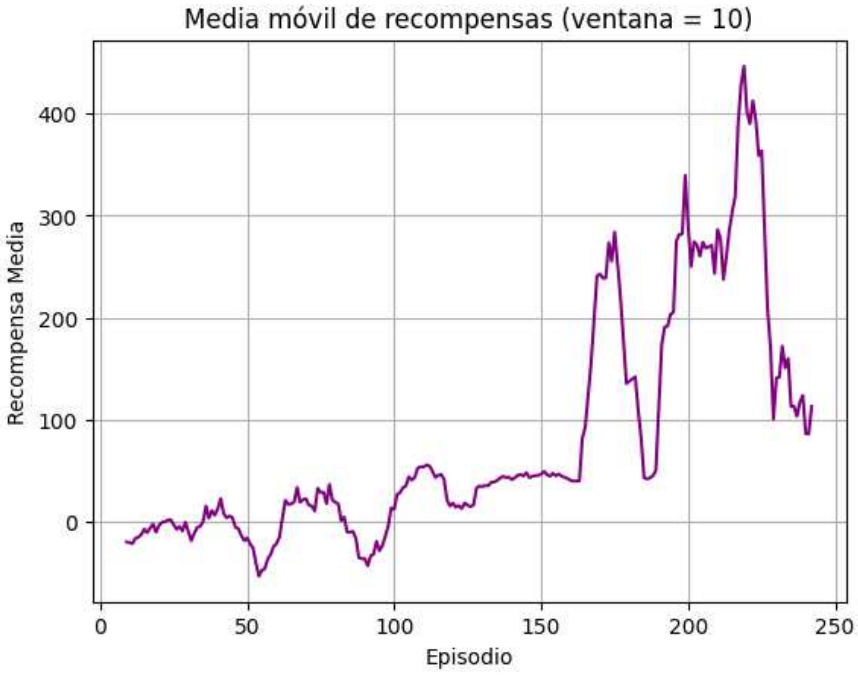

Figura 57: Media móvil de las recompensas con columnas no influyentes

La Figura 57, correspondiente a la media móvil de las recompensas, refuerza esta conclusión. Aunque al principio la evolución es prácticamente plana, a partir del episodio 150 se produce un crecimiento más claro. Aun así, el agente sufre algunas recaídas a lo largo del proceso, aunque sin llegar a perder completamente el progreso adquirido.

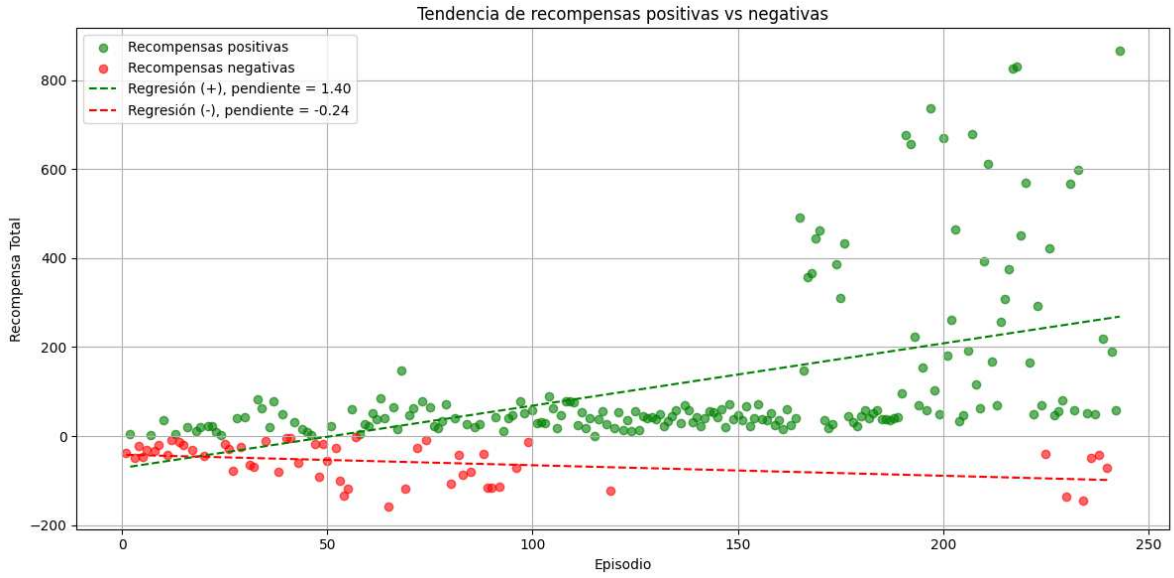

Figura 58: Tendencia de recompensas positivas y negativas con columnas no influyentes

Por último, en la Figura 58 se muestra la tendencia de las recompensas positivas y negativas. La pendiente positiva es de 1,40 lo que es bastante menor que la obtenida en el análisis con columnas influyentes, mientras que la negativa es casi nula, con un valor de

-0,24. Además, durante gran parte del entrenamiento se observan valores muy cercanos a cero. A pesar de todo, gracias a la ligera tendencia positiva que se manifiesta especialmente en los últimos episodios, se puede confirmar que el agente logra aprender, aunque con menor eficiencia que cuando se le proporciona información más relevante.

CONCLUSIONES Y LÍNEAS FUTURAS

CAPÍTULO 8

8. Conclusiones y líneas futuras

8.1. Conclusiones generales del trabajo

En este Trabajo de Fin de Grado se ha trabajado a partir de un conjunto de datos extraído del repositorio Kaggle. A partir de él, se ha realizado un análisis exploratorio donde se han identificado varias correlaciones entre distintas variables y la variable de salida. Una vez completado este proceso inicial, se analizó el rendimiento de las variables respecto a la salida mediante una red neuronal profunda. Posteriormente, se implementó un agente de aprendizaje por refuerzo profundo y se analizó su comportamiento durante el entrenamiento.

Como parte del análisis adicional, se modificaron distintos parámetros, como las funciones de activación y los optimizadores, y se comparó el rendimiento del agente al usando las variables influyentes y no influyentes frente al uso de todas las variables del conjunto. Todo este proceso ha permitido comprender en profundidad las fases necesarias para construir y evaluar el rendimiento de un agente de aprendizaje por refuerzo, desde los datos iniciales hasta el análisis final de los resultados.

En conclusión, los resultados obtenidos han sido positivos, y se ha logrado demostrar el funcionamiento completo de un agente de aprendizaje por refuerzo desde cero: desde la creación del entorno y el diseño del agente, hasta la definición de una política y una estructura de recompensas coherente con el objetivo del sistema.

Considero que ha sido un trabajo muy completo, ya que se han abordado en profundidad temas clave de gran relevancia actual. Por un lado, se ha trabajado con un conjunto de datos con potencial de aplicación tanto en entornos industriales como en contextos estratégicos como la defensa, lo que amplía el alcance de los resultados obtenidos. Por otro, se han aplicado técnicas avanzadas de inteligencia artificial, como el análisis de correlaciones, redes neuronales profundas y aprendizaje por refuerzo, lo que permite tener una visión amplia del potencial de estas herramientas cuando se combinan en un proyecto completo.

Desde un punto de vista personal, este trabajo me ha aportado una comprensión global del funcionamiento de los sistemas de inteligencia artificial, permitiéndome recorrer todo el proceso: desde la preparación y análisis de los datos hasta la implementación de modelos complejos que aprenden a tomar decisiones.

8.2. Valoración del modelo y de los resultados

Para valorar el modelo original es esencial comprobar su rendimiento a lo largo de los episodios, tal y como se comentó en el Capítulo 7, en los apartados 7.2 y 7.2.1, donde se observa que el agente aprende progresivamente. No obstante, cabe destacar las fuertes caídas que sufre durante todo el procedimiento, llegando incluso a registrar recompensas

negativas en episodios avanzados.

Se puede llegar a la conclusión de que esto ocurre por diversas razones. En primer lugar, debido a la política de exploración seguida, donde el valor de ϵ comienza en un 80 % y va disminuyendo gradualmente, se realizan muchas acciones aleatorias incluso en fases avanzadas del entrenamiento. De hecho, el agente termina el episodio 300 con un valor de $\epsilon = 0{,}04$, por lo que aún existe un 4 % de probabilidad de que tome decisiones aleatorias, lo que puede explicar algunas de las recaídas bruscas observadas.

En segundo lugar, como se explicó en el apartado 6.4.2, este modelo no implementa una red objetivo (target network), lo cual incrementa la sensibilidad a recompensas puntuales mal distribuidas y puede generar picos de error más acusados.

Por último, también es importante señalar que estas recaídas forman parte del comportamiento habitual en agentes de aprendizaje por refuerzo. Durante el entrenamiento, el agente reentrena su red con nuevas muestras en cada episodio. Esto puede dar lugar a sobreajustes temporales a patrones recientes que no representan adecuadamente el conjunto de datos completo, generando así fases de inestabilidad en las que parece que el agente olvida lo aprendido anteriormente.

8.2.1. Columnas con y sin influencia sobre la salida

Siguiendo en la línea del modelo original, se pudo comprobar que, al separar las columnas en influyentes y no influyentes, el agente seguía siendo capaz de aprender en ambos casos. Este resultado es especialmente relevante en el caso de las columnas no influyentes, ya que, a priori, podría presuponerse que el agente no lograría aprender adecuadamente al estar privadas de una correlación directa con la variable de recompensa. Sin embargo, los resultados muestran que, aunque tarde más en aprender, el agente alcanza niveles de rendimiento comparables a los del modelo original.

Además, se observa que el comportamiento del agente con las columnas no influyentes es incluso menos irregular: experimenta menos caídas pronunciadas que en el caso del agente original o del que utiliza únicamente las columnas influyentes, siendo este último el que alcanza las recompensas más altas.

De este análisis pueden extraerse varias conclusiones de peso. La primera es la importancia de realizar un análisis previo, el cual proporcionó información muy relevante en cuanto al funcionamiento de los datos con respecto a la variable de salida encontrando patrones claros de su comportamiento. La segunda es que aunque algunas columnas no presentaban correlación aparente con la variable de recompensa en el análisis previo, podrían contener relaciones internas o patrones relevantes que contribuyen al aprendizaje del agente. Por último, con un carácter más general, es que el agente se beneficia del uso del conjunto completo de variables, incluso aquellas con baja correlación directa.

Esto se debe a que dichas variables parecen aportar estabilidad al entrenamiento, reduciendo la magnitud de las recaídas. De hecho, cuando el agente se entrena únicamente con las columnas influyentes, se alcanzan valores de recompensa tan altos como 1750, pero también se registran caídas pronunciadas hasta -250, lo que evidencia una mayor inestabilidad.

8.2.2. Funciones de activación

En cuanto a los cambios de hiperparámetros de la red neuronal en el modelo original, se comenzó por el análisis de las funciones de activación, comparándolas entre sí. Los resultados muestran que la función `sigmoid` proporciona un comportamiento mucho más estable que `ReLU`, llegando incluso a alcanzar recompensas cercanas a los 600, mientras que `ReLU` se queda alrededor de los 500. Además, `sigmoid` presenta menos caídas pronunciadas, lo que indica una mayor regularidad en el aprendizaje. En definitiva, el rendimiento con `sigmoid` resulta claramente superior al obtenido con `ReLU`.

Una posible explicación de esta diferencia radica en la propia naturaleza de estas funciones. Mientras que `sigmoid` permite valores negativos, `ReLU` los anula completamente, los transforma en cero, lo que podría limitar la capacidad del modelo para captar ciertos patrones presentes en los datos.

Por otro lado, la función de activación `ReLU6`, que fue incorporada como experimento adicional, muestra claramente que el agente no logra aprender de forma efectiva. Este comportamiento puede deberse a que, al limitar el valor máximo de salida a 6, la función `ReLU6` ignora muchos de los valores altos presentes en el conjunto de datos, que podrían ser esenciales para el aprendizaje del agente. Es posible que en el artículo en el que se presentó esta función, los datos empleados no tuvieran una escala tan elevada, y por tanto la limitación no resultara perjudicial.

Finalmente, al comparar estos resultados con los del modelo original en el que se utilizaba la función personalizada `ses_activation` (producto de `sigmoid`, `ReLU` y 0,1), se observa que la función `sigmoid` en solitario presenta un comportamiento muy similar. Ambos modelos muestran caídas y picos similares, alcanzan cotas similares de recompensa, y es a partir del episodio 120 cuando comienzan a obtener sus mejores resultados. Se puede concluir, por tanto, que la función `sigmoid` tiene una influencia clara en el rendimiento de la función compuesta `ses_activation`, aunque no se puede determinar con certeza si su comportamiento seguiría siendo estable en fases más avanzadas del entrenamiento. Y que el uso de `ReLU` hace que la función sea más estable, debido a que la función `sigmoid` tiene el problema del desvanecimiento del gradiente, como vimos en el apartado 3.3.1.

8.2.3. Optimizadores

En cuanto a los optimizadores, el uso de `RMSprop` ofrece un aprendizaje caracterizado por fuertes recaídas. Aunque el agente alcanza recompensas elevadas cercanas a 400, en pasos siguientes puede descender bruscamente hasta valores negativos, como -100, y viceversa. Este patrón evidencia que el agente es capaz de aprender, pero lo hace de forma menos estable que en el modelo original. Este comportamiento podría deberse a que `RMSprop` está especialmente diseñado para entrenar redes neuronales recurrentes (RNN), y su aplicación en este contexto puede no resultar tan eficaz.

Con el optimizador `SGD` (Stochastic Gradient Descent), el comportamiento del agente resulta aún más irregular. A lo largo de los episodios se observan grandes oscilaciones, con recompensas positivas relativamente bajas, alrededor de 125 y recompensas negativas no tan pronunciadas, alrededor de -25. Sin embargo, destaca un patrón clave: conforme avanza el entrenamiento, las recompensas negativas desaparecen progresivamente, incluso más que en cualquiera de los otros modelos evaluados. Aunque a simple vista la gráfica podría hacer pensar que el agente no aprende, un análisis más profundo permite concluir que sí lo hace, y de forma correcta, aunque necesitaría un mayor número de episodios para alcanzar un rendimiento comparable. Este comportamiento puede explicarse por la simplicidad del algoritmo de actualización de pesos que utiliza `SGD`.

En conclusión, ninguno de los modelos con optimizadores alternativos mejora el rendimiento del modelo original, ya sea por la inestabilidad que presentan durante el entrenamiento o por la lentitud en el proceso de aprendizaje.

8.3. Limitaciones encontradas

Las principales limitaciones encontradas durante la realización del Trabajo de Fin de Grado estuvieron relacionadas principalmente con el entrenamiento del agente de aprendizaje por refuerzo profundo. Uno de los mayores desafíos fue el tiempo de ejecución necesario para obtener las recompensas, ya que completar los 300 episodios requería aproximadamente 12 horas. Esta fue la principal razón por la que se decidió reducir el número de episodios y acortar los tiempos de ejecución en los análisis posteriores.

8.4. Líneas futuras

A pesar de que los resultados obtenidos han sido satisfactorios, se pueden proponer diversas mejoras al modelo, con el objetivo de optimizar el rendimiento del agente, solucionar algunos de los problemas detectados durante el trabajo o profundizar en el análisis de su comportamiento. Entre las posibles líneas de mejora destacan las siguientes:

- **Análisis de datos con nuevas métricas**: Realizar un análisis más profundo de los datos utilizando nuevas métricas que permitan detectar correlaciones no evidentes entre las columnas inicialmente clasificadas como no influyentes y la variable de recompensa. Esto permitiría comprender con mayor claridad por qué el agente también es capaz de aprender a partir de estas variables.
- **Incluir una red neuronal objetivo**: Incorporar una red neuronal objetivo podría contribuir a estabilizar el entrenamiento y mejorar la convergencia del agente DQN, especialmente en entornos más complejos o con mayor variabilidad. Esta red es común en entornos donde la señal de recompensa es más ruidosa o ambigua, y donde se requiere mayor robustez en el proceso de aprendizaje.
- **Reducción del tiempo de entrenamiento**: Explorar técnicas que permitan acelerar el proceso de entrenamiento, como el uso de técnicas de experiencia priorizada, entrenamientos paralelos o arquitecturas más eficientes, con el fin de facilitar análisis más amplios sin comprometer los recursos computacionales.
- **Aumento de episodios y ajuste de hiperparámetros**: Aumentar el número total de episodios podría permitir observar la evolución del agente a más largo plazo y comprobar si finalmente alcanza una política totalmente estable. Además, experimentar con el parámetro ϵ en la estrategia ϵ-greedy permitiría evaluar el impacto de reducir o acelerar la transición entre exploración y explotación.
- **Aplicación práctica del modelo**: Investigar posibles aplicaciones del modelo en contextos reales de la industria o defensa, donde el uso de sensores IA y aprendizaje por refuerzo podría resultar útil para la toma de decisiones o mantenimiento predictivo.

ANEXOS

A. Anexos

A.1. Código fuente principal

Listing 21: Carga de librerias y configuracion del entorno

```
!pip install pandas numpy matplotlib seaborn scikit-learn
   tensorflow gym statsmodels xgboost optuna shap kagglehub
   scikit-multilearn

# Importacion de librerias
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import os
import math
import json
from pathlib import Path
import shutil
import joblib
import warnings
warnings.filterwarnings('ignore')

# Mostrar graficos dentro del notebook
%matplotlib inline

# Preprocesamiento y transformacion de datos
from sklearn.preprocessing import StandardScaler, MinMaxScaler,
   OneHotEncoder, LabelEncoder
from sklearn.model_selection import train_test_split
from sklearn.pipeline import Pipeline
from sklearn.base import BaseEstimator, TransformerMixin

from statsmodels.stats.outliers_influence import
   variance_inflation_factor
from statsmodels.tools.tools import add_constant

# Modelos de Machine Learning
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import (
    RandomForestClassifier,
    GradientBoostingClassifier,
    AdaBoostClassifier
)
from xgboost import XGBClassifier
#from catboost import CatBoostClassifier
from sklearn.multioutput import MultiOutputClassifier
```

```
# Deep Learning (TensorFlow / Keras)
#import tensorflow as tf
#from tensorflow.keras.models import Sequential
#from tensorflow.keras.layers import Dense, Dropout, Lambda,
    BatchNormalization, Activation
#from tensorflow.keras import backend as K
#from tensorflow.keras.utils import get_custom_objects
#from tensorflow.keras.optimizers import Adam

# Aprendizaje por Refuerzo
import gym
from gym import spaces
from collections import deque
import random

# Evaluacion de modelos
from sklearn.metrics import (
    accuracy_score,
    precision_score,
    recall_score,
    f1_score,
    classification_report,
    make_scorer,
    jaccard_score,
    multilabel_confusion_matrix
)
from sklearn.model_selection import cross_val_score
from skmultilearn.model_selection import IterativeStratification

# Analisis dimensional
from sklearn.decomposition import PCA
from sklearn.neighbors import NearestNeighbors

# Optimizacion y explicabilidad
import optuna
import shap

# Kaggle (solo si usas KaggleHub)
import kagglehub
```

Listing 22: Carga de datos y analisis exploratorio

```
# Define constants
dataset_identifier = "emirhanai/advanced-signal-processing-
    dataset-from-ai-sensors"
input_data_folder = "input_data"  # Folder to store final files
extract_dir = os.path.join(os.getcwd(), input_data_folder)  #
    Final extraction path

# Download the dataset from Kaggle
path = kagglehub.dataset_download(dataset_identifier)
```

```
# Create the destination directory if it doesn't exist
if not os.path.exists(extract_dir):
    os.makedirs(extract_dir)

# Copy files from the Kaggle download path to your input folder
for file_name in os.listdir(path):
    full_file_name = os.path.join(path, file_name)
    if os.path.isfile(full_file_name):
        shutil.copy(full_file_name, extract_dir)

# Print the names of files copied to input_data
print(f"The extracted files are:")
for file_name in os.listdir(extract_dir):
    print(f"- {file_name}")

# Cargar todos los CSVs al diccionario 'data'
data = {}
for filename in os.listdir(extract_dir):
    if filename.endswith(".csv"):
        key = filename.replace(".csv", "")
        data[key] = pd.read_csv(os.path.join(extract_dir,
           filename))

# Mostrar las primeras filas de sensor_readings_train como
   comprobacion
data['sensor_readings_train'].head()

for name, df in data.items():
    print(f"\\n {name} | Shape: {df.shape}")
    display(df.head(5))              # Muestra 5 primeras filas
       del archivo
    print(df.info())                 # Muestra tipos de columnas y
        nulos
    display(df.describe().T)         # Estadisticas basicas de
       columnas numericas

for name, df in data.items():
    print(f"\\n Archivo: {name}")
    print(df.isna().sum())

for name, df in data.items():
    print(f"\\n Archivo: {name}")
    print(f"Numero de filas duplicadas: {df.duplicated().sum()}")

for name, df in data.items():
    print(f"\\n===== {name} =====")

    # Check for Mixed Data Types
    print("-> Tipos de datos mezclados por columna:")
    for col in df.columns:
```

```
        num_types = df[col].apply(type).nunique()
        print(f"  {col}: {num_types} tipo(s) distinto(s)")

    # Identify Columns with Constant Values
    print("\\n-> Valores unicos por columna (numericas):")
    for col in df.select_dtypes(exclude=['object']):
        unique_vals = df[col].nunique()
        if unique_vals == 1:
            print(f"  '{col}' tiene un unico valor constante.")
        else:
            print(f"  '{col}' tiene {unique_vals} valores unicos.
               ")
```

Listing 23: Graficos de variables respecto a la recompensa

```
# Histograma de la temperatura infrarroja en distintos conjuntos
df = data['sensor_readings_train']
plt.hist(df['infrared_temperature'], bins=30, color='steelblue',
   edgecolor='black')
plt.title("Distribucion de la temperatura infrarroja")
plt.xlabel("Temperatura (C)")
plt.ylabel("Frecuencia")
plt.grid(True)
plt.show()

df = data['sensor_readings_test']
plt.hist(df['infrared_temperature'], bins=30, color='steelblue',
   edgecolor='black')
plt.title("Distribucion de la temperatura infrarroja")
plt.xlabel("Temperatura (C)")
plt.ylabel("Frecuencia")
plt.grid(True)
plt.show()

df = data['sensor_readings_validation']
plt.hist(df['infrared_temperature'], bins=30, color='steelblue',
   edgecolor='black')
plt.title("Distribucion de la temperatura infrarroja")
plt.xlabel("Temperatura (C)")
plt.ylabel("Frecuencia")
plt.grid(True)
plt.show()

# Comparar reward con columnas del archivo sensor_readings_train
columnas_objetivo = ['radar_signal_strength', 'sonar_distance', '
   infrared_temperature']
x = data['reward_signals_train']['reward']
df_sensores = data['sensor_readings_train']

for col in columnas_objetivo:
    if col in df_sensores.columns:
        plt.scatter(x, df_sensores[col], alpha=0.5, color='
```

```
            darkcyan')
        plt.title(f"{col} vs. Recompensa")
        plt.xlabel("Reward")
        plt.ylabel(col)
        plt.grid(True)
        plt.show()

# Comparar reward con condiciones medioambientales
columnas_ambientales = ['temperature', 'humidity']
x = data['reward_signals_train']['reward']
df_ambiente = data['environmental_conditions_train']

for col in columnas_ambientales:
    if col in df_ambiente.columns:
        plt.scatter(x, df_ambiente[col], alpha=0.5, color='
            darkorange')
        plt.title(f"{col} vs. Recompensa")
        plt.xlabel("Reward")
        plt.ylabel(col)
        plt.grid(True)
        plt.show()

# Comparar reward con columna duration_seconds
columna_objetivo = ['duration_seconds']
x = data['reward_signals_train']['reward']
df_sensores = data['action_logs_train']

for col in columna_objetivo:
    if col in df_sensores.columns:
        plt.scatter(x, df_sensores[col], alpha=0.5, color='
            darkcyan')
        plt.title(f"{col} vs. Recompensa")
        plt.xlabel("Reward")
        plt.ylabel(col)
        plt.grid(True)
        plt.show()

# Hexbin plot entre radar_signal_strength y reward
x = data['sensor_readings_train']['radar_signal_strength']
y = data['reward_signals_train']['reward']

plt.hexbin(x, y, gridsize=30, cmap='coolwarm')
plt.colorbar(label='Cantidad de puntos')
plt.title("Radar Signal Strength vs. Reward")
plt.xlabel("Radar Signal Strength")
plt.ylabel("Reward")
plt.grid(True)
plt.tight_layout()
plt.show()
```

Listing 24: Boxplots, graficas temporales y merge de datasets

```
# Cargamos el DataFrame correspondiente
df_env = data['environmental_conditions_train']
df_reward = data['reward_signals_train']

# Concatenamos ambos DataFrames por fila (indice)
df_combined = pd.concat([df_env, df_reward], axis=1)

# Columnas categoricas a analizar
categorical_columns = ['weather_condition', 'terrain_type', '
   signal_interference_level']

# Crear boxplot para cada columna categorica vs reward
for col in categorical_columns:
    if col in df_combined.columns:
        plt.figure(figsize=(8, 5))
        sns.boxplot(x=df_combined[col], y=df_combined['reward'],
           palette='pastel')
        plt.title(f"Reward segun {col}")
        plt.xlabel(col.replace("_", " ").capitalize())
        plt.ylabel("Reward")
        plt.xticks(rotation=45)
        plt.grid(True)
        plt.tight_layout()
        plt.show()

# Logs de acciones
df_actions = data['action_logs_train']
df_rewards = data['reward_signals_train']
df_merged = df_actions.join(df_rewards['reward'])

columnas_categoricas = ['action_taken', 'action_result']

for col in columnas_categoricas:
    if col in df_merged.columns:
        plt.figure(figsize=(10, 5))
        sns.boxplot(x=df_merged[col], y=df_merged['reward'],
           palette='Set2')
        plt.title(f"Reward segun {col}")
        plt.xlabel(col)
        plt.ylabel("Reward")
        plt.xticks(rotation=30)
        plt.grid(True)
        plt.show()

# Salidas algoritmicas
df_outputs = data['algorithmic_outputs_train']
df_rewards = data['reward_signals_train']
df_merged_outputs = df_outputs.join(df_rewards['reward'])

columnas_output = ['target_identification', 'threat_level', '
```

```
   navigation_instruction']

for col in columnas_output:
    if col in df_merged_outputs.columns:
        plt.figure(figsize=(10, 5))
        sns.boxplot(x=df_merged_outputs[col], y=df_merged_outputs
           ['reward'], palette='Set3')
        plt.title(f"Reward segun {col}")
        plt.xlabel(col)
        plt.ylabel("Reward")
        plt.xticks(rotation=30)
        plt.grid(True)
        plt.show()

# Graficas de linea por variable en el tiempo
graficas = [
    ("sensor_readings_train", ["radar_signal_strength"], 'blue'),
    ("algorithmic_outputs_train", ["threat_level"], 'red'),
    ("reward_signals_train", ["reward"], 'green')
]

for archivo, columnas, color in graficas:
    df = data[archivo]
    if 'timestamp' in df.columns:
        df['timestamp'] = pd.to_datetime(df['timestamp'])
        df['minutes'] = (df['timestamp'] - df['timestamp'].iloc
           [0]).dt.total_seconds() / 60

        for col in columnas:
            if col in df.columns:
                if col == "threat_level":
                    df[col + '_num'] = df[col].map({"Low": 1, "
                       Medium": 2, "High": 3})
                    plt.plot(df['minutes'], df[col + '_num'],
                       color=color, alpha=0.5)
                    plt.title("Threat Level a lo largo del tiempo
                       ")
                    plt.xlabel("Tiempo (minutos)")
                    plt.ylabel("Threat Level")
                    plt.yticks([1, 2, 3], labels=["Low", "Medium"
                       , "High"])
                    plt.grid(True)
                    plt.tight_layout()
                    plt.show()
                else:
                    plt.plot(df['minutes'], df[col], color=color,
                        alpha=0.5)
                    plt.title(f"{col} a lo largo del tiempo")
                    plt.xlabel("Tiempo (minutos)")
                    plt.ylabel(col)
                    plt.grid(True)
```

```
                    plt.tight_layout()
                    plt.show()

# Procesamiento de timestamps
for name in data:
    if 'timestamp' in data[name].columns:
        data[name]['timestamp'] = pd.to_datetime(data[name]['
           timestamp'], errors='coerce')
data[name] = data[name].dropna(subset=['timestamp'])

# Merge final del conjunto de entrenamiento
train_data = data['sensor_readings_train'].merge(
    data['system_states_train'], on='timestamp'
).merge(
    data['algorithmic_outputs_train'], on='timestamp'
).merge(
    data['environmental_conditions_train'], on='timestamp'
).merge(
    data['action_logs_train'], on='timestamp'
).merge(
    data['reward_signals_train'], on='timestamp'
)
```

Listing 25: Preprocesamiento, modelo con funcion SES y entrenamiento

```
# Limpieza de columnas y tratamiento de nulos
train_data = train_data.drop(['minutes_x', 'minutes_y', '
   threat_level_num', 'minutes'], axis=1)
train_data.head()

numerical_cols = train_data.select_dtypes(include=['float64', '
   int64']).columns
train_data[numerical_cols] = train_data[numerical_cols].fillna(
   train_data[numerical_cols].mean())

categorical_cols = train_data.select_dtypes(include=['object']).
   columns
train_data[categorical_cols] = train_data[categorical_cols].
   fillna(train_data[categorical_cols].mode().iloc[0])

# Codificacion de variables categoricas
le = LabelEncoder()
for col in categorical_cols:
    train_data[col] = le.fit_transform(train_data[col])

# Separacion y escalado
X = train_data.drop(['timestamp', 'reward'], axis=1)
y = train_data['reward']
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Division train/val
```

```
from sklearn.model_selection import train_test_split
X_train, X_val, y_train, y_val = train_test_split(X_scaled, y,
    test_size=0.2, random_state=42)

print(f"Training samples: {X_train.shape[0]}")
print(f"Validation samples: {X_val.shape[0]}")
print(train_data.columns)

# Modelo de red con funcion de activacion personalizada SES
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout, Activation,
    Lambda
from tensorflow.keras import backend as K
from tensorflow.keras.utils import get_custom_objects
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.activations import swish

def ses_activation(x, alpha=0.1):
    return K.sigmoid(x) * K.relu(x) * alpha

get_custom_objects().update({'SES': Activation(ses_activation)})

# Visualizacion de la funcion personalizada
x = np.linspace(-10, 10, 500)

def relu6(x): return np.minimum(np.maximum(0, x), 6)
def mish(x): return x * np.tanh(np.log1p(np.exp(x)))
def ses_activacion(x): return relu6(x)*mish(x)*0.1

plt.figure(figsize=(12, 8))
plt.plot(x, ses_activacion(x), label='Funcion de activacion')
plt.title("Funcion de activacion de prueba")
plt.xlabel("x")
plt.ylabel("f(x)")
plt.grid(True)
plt.legend()
plt.tight_layout()
plt.show()

# Definicion del modelo
model = Sequential([
    Dense(256, input_dim=X_train.shape[1]),
    Lambda(ses_activation),
    Dropout(0.3),
    Dense(512),
    Lambda(ses_activation),
    Dropout(0.3),
    Dense(512),
    Lambda(ses_activation),
    Dense(1, activation='linear')
```

```
])

model.compile(optimizer='adam', loss='mse', metrics=['mae'])
model.summary()

# Entrenamiento del modelo
history = model.fit(
    X_train, y_train,
    epochs=100,
    batch_size=32,
    validation_data=(X_val, y_val),
    callbacks=[
        tf.keras.callbacks.EarlyStopping(monitor='val_loss',
           patience=4, restore_best_weights=True)
    ]
)

# Graficas de perdida y MAE
plt.figure(figsize=(14, 6))

plt.subplot(1, 2, 1)
plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Model Loss')
plt.ylabel('Mean Squared Error')
plt.xlabel('Epoch')
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(history.history['mae'], label='Train MAE')
plt.plot(history.history['val_mae'], label='Validation MAE')
plt.title('Model MAE')
plt.ylabel('Mean Absolute Error')
plt.xlabel('Epoch')
plt.legend()

plt.show()
```

Listing 26: Definicion del entorno personalizado AISensorEnv

```
class AISensorEnv(gym.Env):
    """
    Custom Environment for AI Sensor Data using OpenAI Gym
    """
    def __init__(self, data):
        super(AISensorEnv, self).__init__()
        self.data = data.reset_index(drop=True)
        self.current_step = 0
        self.done = False

        # Excluir columnas 'reward' y 'timestamp'
        self.feature_columns = self.data.columns.difference(['
```

```
            reward', 'timestamp']).tolist()

        # Definir espacio de acciones y observaciones
        self.action_space = spaces.Discrete(5)  # Ejemplo: 5
            acciones posibles
        self.observation_space = spaces.Box(
            low=-np.inf, high=np.inf, shape=(len(self.
                feature_columns),), dtype=np.float32
        )

    def reset(self):
        self.current_step = 0
        self.done = False
        return self._get_observation()

    def _get_observation(self):
        observation = self.data.loc[self.current_step, self.
            feature_columns].values.astype(np.float32)
        return observation

    def step(self, action):
        expected_action = self.data.loc[self.current_step, '
            action_taken']

        if action == expected_action:
            reward = 1.0
        elif abs(action - expected_action) == 1:
            reward = -0.05
        else:
            reward = -0.4

        # Imprimir cada 300 pasos
        if self.current_step % 300 == 0:
            print(f"Step: {self.current_step}, Action: {action},
                Expected: {expected_action}, Reward: {reward}")

        self.current_step += 1
        if self.current_step >= len(self.data) - 1:
            self.done = True

        return self._get_observation(), reward, self.done, {}

    def render(self, mode='human'):
        pass  # No implementado
```

Listing 27: Definicion del agente DQN personalizado

```
class DQNAgent:
    def __init__(self, state_size, action_size):
        self.state_size = state_size
        self.action_size = action_size
```

```
        # Hiperparametros
        self.memory = deque(maxlen=2000)
        self.gamma = 0.95    # Tasa de descuento
        self.epsilon = 0.8   # Tasa de exploracion
        self.epsilon_min = 0.01
        self.epsilon_decay = 0.99
        self.learning_rate = 0.0005

        # Construccion del modelo
        self.model = self._build_model()

    def _build_model(self):
        model = Sequential()
        model.add(Dense(64, input_dim=self.state_size, activation
            =Lambda(ses_activation)))
        model.add(Dense(64, activation=Lambda(ses_activation)))
        model.add(Dense(self.action_size, activation='linear'))
        optimizer = Adam(learning_rate=self.learning_rate,
            clipnorm=1.0)
        model.compile(loss='mse', optimizer=optimizer)
        return model

    def remember(self, state, action, reward, next_state, done):
        self.memory.append((state, action, reward, next_state,
            done))

    def act(self, state):
        if np.random.rand() <= self.epsilon:
            return random.randrange(self.action_size)
        state = state.reshape(1, -1)
        act_values = self.model.predict(state, verbose=0)
        return np.argmax(act_values[0])

    def replay(self, batch_size):
        if len(self.memory) < batch_size:
            return
        minibatch = random.sample(self.memory, batch_size)
        for state, action, reward, next_state, done in minibatch:
            state = state.reshape(1, -1)
            next_state = next_state.reshape(1, -1)
            target = reward
            if not done:
                target += self.gamma * np.amax(self.model.predict
                    (next_state.reshape(1, -1), verbose=0)[0])
            target_f = self.model.predict(state.reshape(1, -1),
                verbose=0)
            target_f[0][action] = target
            self.model.fit(state.reshape(1, -1), target_f, epochs
                =1, verbose=0)
        if self.epsilon > self.epsilon_min:
            self.epsilon *= self.epsilon_decay
```

```
# Parametros para entrenamiento DQN
state_size = X_train.shape[1]
action_size = env.action_space.n
batch_size = 64
n_episodes = 300
output_dir = 'model_output/ai_sensor_dqn/'

# Crear directorio de salida si no existe
if not os.path.exists(output_dir):
    os.makedirs(output_dir)

# Inicializar agente
agent = DQNAgent(state_size, action_size)
```

Listing 28: Entrenamiento del agente DQN y evaluacion global

```
rewards = []
step_rewards_matrix = []  # Para almacenar las recompensas por
   step de cada episodio

for e in range(n_episodes):
    state = env.reset()
    total_reward = 0
    done = False
    episode_step_rewards = []

    while not done:
        action = agent.act(state)
        next_state, reward, done, _ = env.step(action)
        agent.remember(state, action, reward, next_state, done)
        state = next_state
        total_reward += reward
        episode_step_rewards.append(reward)

    step_rewards_matrix.append(episode_step_rewards)
    rewards.append(total_reward)

    # Reentrenamiento por episodio
    agent.replay(batch_size)

    print(f"\\n====== EPISODE {e+1}/{n_episodes} ======  Total
       Reward: {total_reward:.2f} | Epsilon: {agent.epsilon:.2f
       }\\n")

    # Guardar pesos cada 50 episodios o al final
    if (e + 1) % 50 == 0 or e == n_episodes - 1:
        agent.model.save_weights(output_dir + f"model_ep{e+1}.
           weights.h5")

# Fin del entrenamiento
print("\\n========== ENTRENAMIENTO COMPLETADO ==========\\n")
```

```
print(f"Episodios totales: {n_episodes}")
print(f"Recompensa media final: {np.mean(rewards):.2f}")
print(f"Recompensa maxima alcanzada: {np.max(rewards):.2f}")
print(f"Ultima epsilon: {agent.epsilon:.2f}")

# Evaluacion final
if np.mean(rewards[-30:]) > np.mean(rewards[:30]):
    print(" El agente ha mejorado su rendimiento a lo largo del
       entrenamiento.")
else:
    print(" El agente no muestra una mejora clara. Considera
       revisar el entorno, los hiperparametros o la funcion de
       recompensa.")
```

Listing 29: Graficos y analisis de recompensas por episodio

```
# Recompensa total por episodio
plt.figure(figsize=(12,6))
plt.plot(rewards)
plt.xlabel('Episode')
plt.ylabel('Total Reward')
plt.title('Reinforcement Learning Agent Performance')
plt.show()

# Funcion para calcular porcentajes por tramo
def calculate_reward_percentages(rewards):
    total = len(rewards)
    positive = sum(1 for r in rewards if r > 0)
    negative = sum(1 for r in rewards if r <= 0)
    return {
        'Episodios Analizados': total,
        'Recompensas Positivas (%)': round(100 * positive / total
           , 2),
        'Recompensas Negativas (%)': round(100 * negative / total
           , 2)
    }

# Segmentos
raw_segments = list(range(10, 310, 10))
segments = [s for s in raw_segments if s <= len(rewards)]
percentages_data = [calculate_reward_percentages(rewards[:seg])
   for seg in segments]
df_percentages = pd.DataFrame(percentages_data)

print("Porcentaje de recompensas positivas/negativas por tramo:")
display(df_percentages)

# Recompensa total graficada
plt.figure(figsize=(12, 6))
plt.plot(range(1, len(rewards) + 1), rewards, marker='o',
   linestyle='-', color='orange')
plt.title('Recompensa total por episodio')
```

```
plt.xlabel('Episodio')
plt.ylabel('Recompensa Total')
plt.grid(True)
plt.tight_layout()
plt.show()

# Histograma de recompensas
plt.hist(rewards, bins=30, color='green', edgecolor='black')
plt.title('Distribucion de recompensas totales')
plt.xlabel('Recompensa')
plt.ylabel('Frecuencia')
plt.grid(True)
plt.show()

# Media movil
window = 10
rolling_avg = pd.Series(rewards).rolling(window).mean()
plt.plot(rolling_avg, color='purple')
plt.title(f'Media movil de recompensas (ventana = {window})')
plt.xlabel('Episodio')
plt.ylabel('Recompensa Media')
plt.grid(True)
plt.show()

# Regresion positiva y negativa
episodes = np.arange(1, len(rewards) + 1)
rewards = np.array(rewards)

pos_episodes = episodes[rewards > 0]
pos_rewards = rewards[rewards > 0]
neg_episodes = episodes[rewards <= 0]
neg_rewards = rewards[rewards <= 0]

pos_slope, pos_intercept = np.polyfit(pos_episodes, pos_rewards,
    1)
neg_slope, neg_intercept = np.polyfit(neg_episodes, neg_rewards,
    1)

plt.figure(figsize=(12, 6))
plt.scatter(pos_episodes, pos_rewards, color='green', label='
    Recompensas positivas', alpha=0.6)
plt.scatter(neg_episodes, neg_rewards, color='red', label='
    Recompensas negativas', alpha=0.6)

plt.plot(pos_episodes, pos_slope * pos_episodes + pos_intercept,
    color='green', linestyle='--',
          label=f'Regresion (+), pendiente = {pos_slope:.2f}')
plt.plot(neg_episodes, neg_slope * neg_episodes + neg_intercept,
    color='red', linestyle='--',
          label=f'Regresion (-), pendiente = {neg_slope:.2f}')
```

```
plt.title('Tendencia de recompensas positivas vs negativas')
plt.xlabel('Episodio')
plt.ylabel('Recompensa Total')
plt.grid(True)
plt.legend()
plt.tight_layout()
plt.show()

# Analisis por bloques
episodes = np.arange(1, len(rewards) + 1)
rewards = np.array(rewards)
block_size = 50
num_blocks = len(rewards) // block_size

plt.figure(figsize=(14, 7))
print("Pendientes por bloque:")
for i in range(num_blocks):
    start = i * block_size
    end = (i + 1) * block_size
    block_episodes = episodes[start:end]
    block_rewards = rewards[start:end]

    pos_eps = block_episodes[block_rewards > 0]
    pos_vals = block_rewards[block_rewards > 0]
    neg_eps = block_episodes[block_rewards <= 0]
    neg_vals = block_rewards[block_rewards <= 0]

    plt.scatter(pos_eps, pos_vals, color='green', alpha=0.6)
    plt.scatter(neg_eps, neg_vals, color='red', alpha=0.6)

    pos_text = "No hay suficientes puntos"
    neg_text = "No hay suficientes puntos"

    if len(pos_eps) > 1:
        pos_slope, pos_intercept = np.polyfit(pos_eps, pos_vals,
           1)
        y_pos = pos_slope * pos_eps + pos_intercept
        plt.plot(pos_eps, y_pos, linestyle='--', color='green')
        mid_idx = len(pos_eps) // 2
        if mid_idx > 0:
            plt.text(pos_eps[mid_idx], y_pos[mid_idx] + 30, f'{i
               +1}', color='green', fontsize=10, ha='center')
        pos_text = f'pendiente = {pos_slope:.2f}'

    if len(neg_eps) > 1:
        neg_slope, neg_intercept = np.polyfit(neg_eps, neg_vals,
           1)
        y_neg = neg_slope * neg_eps + neg_intercept
        plt.plot(neg_eps, y_neg, linestyle='--', color='red')
        mid_idx = len(neg_eps) // 2
        if mid_idx > 0:
```

```
            plt.text(neg_eps[mid_idx], y_neg[mid_idx] + 30, f'{i
               +1}', color='red', fontsize=10, ha='center')
        neg_text = f'pendiente = {neg_slope:.2f}'

    print(f'Bloque {i+1} (+): {pos_text} - Bloque {i+1} (-): {
       neg_text}')

plt.title('Tendencia por bloques de 50 episodios')
plt.xlabel('Episodio')
plt.ylabel('Recompensa Total')
plt.grid(True)
plt.tight_layout()
plt.show()
```

A.1.1. Cambios realizados para las distintas variables

Listing 30: Reduccion del conjunto de entrenamiento a columnas influyentes

```
# Fusion de todos los conjuntos de entrenamiento por timestamp
train_data = data['sensor_readings_train'].merge(
    data['system_states_train'], on='timestamp'
).merge(
    data['algorithmic_outputs_train'], on='timestamp'
).merge(
    data['environmental_conditions_train'], on='timestamp'
).merge(
    data['action_logs_train'], on='timestamp'
).merge(
    data['reward_signals_train'], on='timestamp'
)

# Eliminar columnas innecesarias generadas en el proceso
train_data = train_data.drop(['minutes_x', 'minutes_y', '
   threat_level_num', 'minutes'], axis=1)

# Mostrar primeras filas del dataset combinado
train_data.head()

# Filtrar solo columnas con influencia real sobre la recompensa
train_data = train_data[['timestamp', 'threat_level', '
   radar_signal_strength', 'action_taken', 'reward']]

...

# Hyperparameters for DQN
state_size = X_train.shape[1]  # Should be 19
action_size = env.action_space.n
batch_size = 64
n_episodes = 240
output_dir = 'model_output/ai_sensor_dqn/'
```

```
# Ensure output directory exists
if not os.path.exists(output_dir):
    os.makedirs(output_dir)

# Initialize the agent
agent = DQNAgent(state_size, action_size)
```

Listing 31: Reduccion del dataset a columnas no influyentes

```
# Fusion de todos los conjuntos de entrenamiento por timestamp
train_data = data['sensor_readings_train'].merge(
    data['system_states_train'], on='timestamp'
).merge(
    data['algorithmic_outputs_train'], on='timestamp'
).merge(
    data['environmental_conditions_train'], on='timestamp'
).merge(
    data['action_logs_train'], on='timestamp'
).merge(
    data['reward_signals_train'], on='timestamp'
)

# Eliminacion de columnas innecesarias tras el merge
train_data = train_data.drop(['minutes_x', 'minutes_y', '
   threat_level_num', 'minutes'], axis=1)

# Mostrar resultado del dataset combinado
train_data.head()

# Filtrar para quedarnos unicamente con columnas NO influyentes
columnas_no_influyentes = train_data.columns.difference(['
   threat_level', 'radar_signal_strength'])
train_data = train_data[columnas_no_influyentes]

...

# Hyperparameters for DQN
state_size = X_train.shape[1]  # Should be 19
action_size = env.action_space.n
batch_size = 64
n_episodes = 240
output_dir = 'model_output/ai_sensor_dqn/'

# Ensure output directory exists
if not os.path.exists(output_dir):
    os.makedirs(output_dir)

# Initialize the agent
agent = DQNAgent(state_size, action_size)
```

Listing 32: Uso de la funcion de activacion sigmoid en el modelo y el agente

```
from tensorflow.keras.layers import Lambda

# Modelo de red neuronal con activacion sigmoid
model = Sequential([
    Dense(256, input_dim=X_train.shape[1], activation='sigmoid'),
    Dropout(0.3),
    Dense(512, activation='sigmoid'),
    Dropout(0.3),
    Dense(512, activation='sigmoid'),
    Dense(1, activation='linear')  # Salida de regresion
])

# Compilacion del modelo
model.compile(optimizer='adam', loss='mse', metrics=['mae'])

# Resumen de la arquitectura
model.summary()

# Redefinicion del agente con activacion sigmoid
class DQNAgent:
    def __init__(self, state_size, action_size):
        self.state_size = state_size
        self.action_size = action_size

        # Hiperparametros
        self.memory = deque(maxlen=2000)
        self.gamma = 0.95
        self.epsilon = 0.8
        self.epsilon_min = 0.01
        self.epsilon_decay = 0.99
        self.learning_rate = 0.0005

        self.model = self._build_model()

    def _build_model(self):
        model = Sequential()
        model.add(Dense(64, input_dim=self.state_size, activation
           ='sigmoid'))
        model.add(Dense(64, activation='sigmoid'))
        model.add(Dense(self.action_size, activation='linear'))
        optimizer = Adam(learning_rate=self.learning_rate,
           clipnorm=1.0)
        model.compile(loss='mse', optimizer=optimizer)
        return model

    def remember(self, state, action, reward, next_state, done):
        self.memory.append((state, action, reward, next_state,
           done))

    def act(self, state):
```

```
        if np.random.rand() <= self.epsilon:
            return random.randrange(self.action_size)
        state = state.reshape(1, -1)
        act_values = self.model.predict(state, verbose=0)
        return np.argmax(act_values[0])

    def replay(self, batch_size):
        if len(self.memory) < batch_size:
            return
        minibatch = random.sample(self.memory, batch_size)
        for state, action, reward, next_state, done in minibatch:
            state = state.reshape(1, -1)
            next_state = next_state.reshape(1, -1)
            target = reward
            if not done:
                target += self.gamma * np.amax(self.model.predict
                    (next_state.reshape(1, -1), verbose=0)[0])
            target_f = self.model.predict(state.reshape(1, -1),
                verbose=0)
            target_f[0][action] = target
            self.model.fit(state.reshape(1, -1), target_f, epochs
                =1, verbose=0)
        if self.epsilon > self.epsilon_min:
            self.epsilon *= self.epsilon_decay

# Parametros del entrenamiento
state_size = X_train.shape[1]
action_size = env.action_space.n
batch_size = 64
n_episodes = 150
output_dir = 'model_output/ai_sensor_dqn/'

# Asegurar directorio de salida
if not os.path.exists(output_dir):
    os.makedirs(output_dir)

# Inicializar agente con sigmoid
agent = DQNAgent(state_size, action_size)
```

Listing 33: Uso de la funcion de activacion ReLU en el modelo y el agente

```
from tensorflow.keras.layers import Lambda

# Modelo de red neuronal con activacion ReLU
model = Sequential([
    Dense(256, input_dim=X_train.shape[1], activation='relu'),
    Dropout(0.3),
    Dense(512, activation='relu'),
    Dropout(0.3),
    Dense(512, activation='relu'),
    Dense(1, activation='linear')  # Salida de regresion
])
```

```
# Compilar el modelo
model.compile(optimizer='adam', loss='mse', metrics=['mae'])
model.summary()

# Definicion del agente con ReLU
class DQNAgent:
    def __init__(self, state_size, action_size):
        self.state_size = state_size
        self.action_size = action_size

        # Hiperparametros
        self.memory = deque(maxlen=2000)
        self.gamma = 0.95
        self.epsilon = 0.8
        self.epsilon_min = 0.01
        self.epsilon_decay = 0.99
        self.learning_rate = 0.0005

        self.model = self._build_model()

    def _build_model(self):
        model = Sequential()
        model.add(Dense(64, input_dim=self.state_size, activation
           ='relu'))
        model.add(Dense(64, activation='relu'))
        model.add(Dense(self.action_size, activation='linear'))
        optimizer = Adam(learning_rate=self.learning_rate,
           clipnorm=1.0)
        model.compile(loss='mse', optimizer=optimizer)
        return model

    def remember(self, state, action, reward, next_state, done):
        self.memory.append((state, action, reward, next_state,
           done))

    def act(self, state):
        if np.random.rand() <= self.epsilon:
            return random.randrange(self.action_size)
        state = state.reshape(1, -1)
        act_values = self.model.predict(state, verbose=0)
        return np.argmax(act_values[0])

    def replay(self, batch_size):
        if len(self.memory) < batch_size:
            return
        minibatch = random.sample(self.memory, batch_size)
        for state, action, reward, next_state, done in minibatch:
            state = state.reshape(1, -1)
            next_state = next_state.reshape(1, -1)
            target = reward
```

```
            if not done:
                target += self.gamma * np.amax(self.model.predict
                    (next_state.reshape(1, -1), verbose=0)[0])
            target_f = self.model.predict(state.reshape(1, -1),
                verbose=0)
            target_f[0][action] = target
            self.model.fit(state.reshape(1, -1), target_f, epochs
                =1, verbose=0)
        if self.epsilon > self.epsilon_min:
            self.epsilon *= self.epsilon_decay

# Parametros del entrenamiento
state_size = X_train.shape[1]
action_size = env.action_space.n
batch_size = 64
n_episodes = 150
output_dir = 'model_output/ai_sensor_dqn/'

# Crear directorio si no existe
if not os.path.exists(output_dir):
    os.makedirs(output_dir)

# Inicializar agente con ReLU
agent = DQNAgent(state_size, action_size)
```

Listing 34: Uso de la funcion de activacion ReLU6 en el modelo y el agente

```
from tensorflow.keras.layers import Lambda, ReLU

# Modelo de red neuronal con activacion ReLU6
model = Sequential([
    Dense(256, input_dim=X_train.shape[1]),
    ReLU(max_value=6),
    Dropout(0.3),
    Dense(512),
    ReLU(max_value=6),
    Dropout(0.3),
    Dense(512),
    ReLU(max_value=6),
    Dense(1, activation='linear')
])

# Compilacion del modelo
model.compile(optimizer='adam', loss='mse', metrics=['mae'])
model.summary()

# Definicion del agente DQN con ReLU6
class DQNAgent:
    def __init__(self, state_size, action_size):
        self.state_size = state_size
        self.action_size = action_size
```

```
        # Hiperparametros
        self.memory = deque(maxlen=2000)
        self.gamma = 0.95
        self.epsilon = 0.8
        self.epsilon_min = 0.01
        self.epsilon_decay = 0.99
        self.learning_rate = 0.0005

        self.model = self._build_model()

    def _build_model(self):
        model = Sequential()
        model.add(Dense(64, input_dim=self.state_size))
        model.add(ReLU(max_value=6))
        model.add(Dense(64))
        model.add(ReLU(max_value=6))
        model.add(Dense(self.action_size, activation='linear'))
        optimizer = Adam(learning_rate=self.learning_rate,
           clipnorm=1.0)
        model.compile(loss='mse', optimizer=optimizer)
        return model

    def remember(self, state, action, reward, next_state, done):
        self.memory.append((state, action, reward, next_state,
           done))

    def act(self, state):
        if np.random.rand() <= self.epsilon:
            return random.randrange(self.action_size)
        state = state.reshape(1, -1)
        act_values = self.model.predict(state, verbose=0)
        return np.argmax(act_values[0])

    def replay(self, batch_size):
        if len(self.memory) < batch_size:
            return
        minibatch = random.sample(self.memory, batch_size)
        for state, action, reward, next_state, done in minibatch:
            state = state.reshape(1, -1)
            next_state = next_state.reshape(1, -1)
            target = reward
            if not done:
                target += self.gamma * np.amax(self.model.predict
                   (next_state.reshape(1, -1), verbose=0)[0])
            target_f = self.model.predict(state.reshape(1, -1),
               verbose=0)
            target_f[0][action] = target
            self.model.fit(state.reshape(1, -1), target_f, epochs
               =1, verbose=0)
        if self.epsilon > self.epsilon_min:
            self.epsilon *= self.epsilon_decay
```

```
# Parametros de entrenamiento
state_size = X_train.shape[1]
action_size = env.action_space.n
batch_size = 64
n_episodes = 150
output_dir = 'model_output/ai_sensor_dqn/'

# Crear carpeta si no existe
if not os.path.exists(output_dir):
    os.makedirs(output_dir)

# Inicializar agente con ReLU6
agent = DQNAgent(state_size, action_size)
```

Listing 35: Uso del optimizador RMSprop en el modelo y el agente

```
from tensorflow.keras.layers import Lambda
from tensorflow.keras.optimizers import RMSprop

# Modelo de red neuronal con SES y RMSprop
model = Sequential([
    Dense(256, input_dim=X_train.shape[1]),
    Lambda(ses_activation),
    Dropout(0.3),
    Dense(512),
    Lambda(ses_activation),
    Dropout(0.3),
    Dense(512),
    Lambda(ses_activation),
    Dense(1, activation='linear')
])

model.compile(optimizer=RMSprop(learning_rate=0.001), loss='mse',
    metrics=['mae'])
model.summary()

# Definicion del agente con SES y RMSprop
class DQNAgent:
    def __init__(self, state_size, action_size):
        self.state_size = state_size
        self.action_size = action_size

        # Hiperparametros
        self.memory = deque(maxlen=2000)
        self.gamma = 0.95
        self.epsilon = 0.8
        self.epsilon_min = 0.01
        self.epsilon_decay = 0.99
        self.learning_rate = 0.0005

        self.model = self._build_model()
```

```
    def _build_model(self):
        model = Sequential()
        model.add(Dense(64, input_dim=self.state_size, activation
            =Lambda(ses_activation)))
        model.add(Dense(64, activation=Lambda(ses_activation)))
        model.add(Dense(self.action_size, activation='linear'))
        optimizer = RMSprop(learning_rate=self.learning_rate,
            clipnorm=1.0)
        model.compile(loss='mse', optimizer=optimizer)
        return model

    def remember(self, state, action, reward, next_state, done):
        self.memory.append((state, action, reward, next_state,
            done))

    def act(self, state):
        if np.random.rand() <= self.epsilon:
            return random.randrange(self.action_size)
        state = state.reshape(1, -1)
        act_values = self.model.predict(state, verbose=0)
        return np.argmax(act_values[0])

    def replay(self, batch_size):
        if len(self.memory) < batch_size:
            return
        minibatch = random.sample(self.memory, batch_size)
        for state, action, reward, next_state, done in minibatch:
            state = state.reshape(1, -1)
            next_state = next_state.reshape(1, -1)
            target = reward
            if not done:
                target += self.gamma * np.amax(self.model.predict
                    (next_state.reshape(1, -1), verbose=0)[0])
            target_f = self.model.predict(state.reshape(1, -1),
                verbose=0)
            target_f[0][action] = target
            self.model.fit(state.reshape(1, -1), target_f, epochs
                =1, verbose=0)
        if self.epsilon > self.epsilon_min:
            self.epsilon *= self.epsilon_decay

# Parametros del entrenamiento
state_size = X_train.shape[1]
action_size = env.action_space.n
batch_size = 64
n_episodes = 150
output_dir = 'model_output/ai_sensor_dqn/'

# Crear directorio si no existe
if not os.path.exists(output_dir):
```

```
    os.makedirs(output_dir)

# Inicializar agente con optimizador RMSprop
agent = DQNAgent(state_size, action_size)
```

Listing 36: Uso del optimizador SGD en el modelo y el agente

```
from tensorflow.keras.layers import Lambda
from tensorflow.keras.optimizers import SGD

# Modelo de red neuronal con SES y optimizador SGD
model = Sequential([
    Dense(256, input_dim=X_train.shape[1]),
    Lambda(ses_activation),
    Dropout(0.3),
    Dense(512),
    Lambda(ses_activation),
    Dropout(0.3),
    Dense(512),
    Lambda(ses_activation),
    Dense(1, activation='linear')
])

model.compile(optimizer=SGD(learning_rate=0.01, momentum=0.9),
   loss='mse', metrics=['mae'])
model.summary()

# Definicion del agente con SES y SGD
class DQNAgent:
    def __init__(self, state_size, action_size):
        self.state_size = state_size
        self.action_size = action_size

        # Hiperparametros
        self.memory = deque(maxlen=2000)
        self.gamma = 0.95
        self.epsilon = 0.8
        self.epsilon_min = 0.01
        self.epsilon_decay = 0.99
        self.learning_rate = 0.0005

        self.model = self._build_model()

    def _build_model(self):
        model = Sequential()
        model.add(Dense(64, input_dim=self.state_size, activation
           =Lambda(ses_activation)))
        model.add(Dense(64, activation=Lambda(ses_activation)))
        model.add(Dense(self.action_size, activation='linear'))
        optimizer = SGD(learning_rate=self.learning_rate,
           momentum=0.9, clipnorm=1.0)
        model.compile(loss='mse', optimizer=optimizer)
```

```
        return model

    def remember(self, state, action, reward, next_state, done):
        self.memory.append((state, action, reward, next_state,
           done))

    def act(self, state):
        if np.random.rand() <= self.epsilon:
            return random.randrange(self.action_size)
        state = state.reshape(1, -1)
        act_values = self.model.predict(state, verbose=0)
        return np.argmax(act_values[0])

    def replay(self, batch_size):
        if len(self.memory) < batch_size:
            return
        minibatch = random.sample(self.memory, batch_size)
        for state, action, reward, next_state, done in minibatch:
            state = state.reshape(1, -1)
            next_state = next_state.reshape(1, -1)
            target = reward
            if not done:
                target += self.gamma * np.amax(self.model.predict
                   (next_state.reshape(1, -1), verbose=0)[0])
            target_f = self.model.predict(state.reshape(1, -1),
               verbose=0)
            target_f[0][action] = target
            self.model.fit(state.reshape(1, -1), target_f, epochs
               =1, verbose=0)
        if self.epsilon > self.epsilon_min:
            self.epsilon *= self.epsilon_decay

# Parametros del entrenamiento
state_size = X_train.shape[1]
action_size = env.action_space.n
batch_size = 64
n_episodes = 150
output_dir = 'model_output/ai_sensor_dqn/'

# Crear directorio si no existe
if not os.path.exists(output_dir):
    os.makedirs(output_dir)

# Inicializar agente con optimizador SGD
agent = DQNAgent(state_size, action_size)
```

A.2. Resultados alternativos en el análisis exploratorio

Este anexo recoge las comprobaciones preliminares realizadas sobre los archivos del conjunto de datos, con el objetivo de asegurar su integridad antes del análisis exploratorio.

Tabla 14: Fragmento Representativo de la comprobación de valores nulos en los archivos

Archivo	**Columnas con valores nulos**
sensor_readings_train	timestamp: 0, radar_signal_strength: 0, sonar_distance: 0, infrared_temperature: 0
sensor_readings_test	timestamp: 0, radar_signal_strength: 0, sonar_distance: 0, infrared_temperature: 0
sensor_readings_validation	timestamp: 0, radar_signal_strength: 0, sonar_distance: 0, infrared_temperature: 0

Tabla 15: Fragmento Representativo de la comprobación de filas duplicadas en los archivos

Archivo	**Filas duplicadas**
sensor_readings_train	0
sensor_readings_test	0
sensor_readings_validation	0

Tabla 16: Fragmento Representativo de los tipos de datos y número de valores únicos por archivo

Archivo	**Columnas con tipos mixtos**	**Columnas con valores únicos (numéricas)**
sensor_readings_train	Todas con 1 tipo	radar_signal_strength (3000), sonar_distance (3000), infrared_temperature (3000)
sensor_readings_test	Todas con 1 tipo	radar_signal_strength (500), sonar_distance (500), infrared_temperature (500)
sensor_readings_validation	Todas con 1 tipo	radar_signal_strength (250), sonar_distance (250), infrared_temperature (250)

Bibliografía

[1] Revista Ejércitos. *Defensa C-UAS (II): Sensores*. Disponible en: https://www.revistaejercitos.com/articulos/defensa-c-uas-ii-sensores/. Consultado en abril de 2025.

[2] Genevo. *¿Cómo funciona un detector de radar?*. Disponible en: https://www.genevo.com/es/como-funciona-detector-de-radar/. Consultado en abril de 2025.

[3] ICEEbook. *SONAR: qué es, cómo funciona, tipos, alcance y aplicaciones*. Disponible en: https://iceebook.com/sonar-que-es-como-funciona-tipos-alcance-aplicaciones. Consultado en abril de 2025.

[4] Infrarrojo.es. *Sensores de radiación infrarroja: cómo funcionan y para qué se usan*. Disponible en: https://infrarrojo.es/blog/sensores-de-radiacion-infrarroja-como-funcionan-y-para-que-se-usan. Consultado en abril de 2025.

[5] Carlo Corsi. *Smart Sensors: Why and when the origin was and why and where the future will be*. Vol. 8993, Dec. 2013, p. 899302. doi: 10.1117/12.2030025.

[6] V2COM. *IoT: Sensores Inteligentes*. Disponible en: https://v2com.com/es/2023/12/23/iot-sensores-inteligentes/#:~:text=En%20general%2C%20un%20sensor%20inteligente,de%20la%20acci%C3%B3n%20a%20tomar.

[7] Magnet Reviews. *Así funciona el primer sensor con inteligencia artificial del mundo*. Disponible en: https://magnetreviews.tech/asi-funciona-el-primer-sensor-con-inteligencia-artificial-del-mundo/.

[8] Dudas y Textos. *Aplicaciones de la inteligencia artificial en el ámbito militar*. Disponible en: https://dudasytextos.com/militar/militar/aplicaciones-de-inteligencia-artificial-en-el-ambito-militar/?utm_source=chatgpt.com.

[9] Joan Cerretani. *Aprendizaje por refuerzo (RL) – Capítulo 1: Historia del aprendizaje por refuerzo*. Medium, 2023. Disponible en: https://medium.com/@joancerretanids/aprendizaje-por-refuerzo-rl-cap%C3%ADtulo-1-historia-del-aprendizaje-por-refuerzo-parte-3-fc4d17197680.

[10] Joan Cerretani. *Aprendizaje por refuerzo (RL) – Capítulo 2: Introducción, parte 1 – ¿Qué es el aprendizaje por refuerzo?*. Medium, 2023. Disponible en: https://medium.com/@joancerretanids/aprendizaje-por-refuerzo-rl-cap%C3%ADtulo-2-introducci%C3%B3n-parte-1-qu%C3%A9-es-el-aprendizaje-por-bde1cf97af43.

[11] Joan Cerretani. *Aprendizaje por refuerzo (RL) – Capítulo 2: Introducción, parte 2 – Recompensas, retornos y Markov*. Medium, 2023. Disponible en: https://medium.com/@joancerretanids/aprendizaje-por-refuerzo-rl-cap%C3%ADtulo-2-introducci%C3%B3n-parte-2-recompensas-retornos-y-markov-36ee763ea9bf.

[12] Joan Cerretani. *Aprendizaje por refuerzo (RL) – Capítulo 2: Introducción, parte 3 – Funciones de valor*. Medium, 2023. Disponible en: https://medium.com/@joancerretanids/aprendizaje-por-refuerzo-rl-cap%C3%ADtulo-2-introducci%C3%B3n-parte-3-funciones-de-valor-555a92c78515.

[13] Richard S. Sutton y Andrew G. Barto. *Reinforcement Learning: An Introduction*. Segunda edición (en progreso). A Bradford Book, The MIT Press, Cambridge, Massachusetts – London, England, 2014–2015.

[14] GeeksforGeeks. *GeeksforGeeks – A computer science portal for geeks*. Disponible en: https://www.geeksforgeeks.org/.

[15] 3Blue1Brown. *Neural Networks – A visual introduction*. Serie de 8 vídeos en YouTube. Disponible en: https://www.youtube.com/watch?v=aircAruvnKk&list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi.

[16] Programación en Python. *Aprende con Alf: Curso de programación en Python*. Disponible en: https://aprendeconalf.es/.

[17] W3Schools. *Tutoriales de Python*. Disponible en: https://www.w3schools.com/.

[18] Python Software Foundation. *Documentación oficial de la biblioteca estándar de Python*. Disponible en: https://docs.python.org/3/library/.

[19] Dongcheul Lee. *Comparison of Reinforcement Learning Activation Functions to Improve the Performance of the Racing Game Learning Agent*. Journal of Information Processing Systems, Vol. 16, No. 5, pp. 1074–1082, 2020. Disponible en: https://doi.org/10.3745/JIPS.02.0141.

[20] Decide Soluciones. *Algoritmos Multi-Armed Bandits en sistemas de recomendación*. Disponible en: https://decidesoluciones.es/algoritmos-multi-armed-bandits-en-sistemas-de-recomendacion/. Consultado en junio de 2025.

[21] Michael Nayna. *Gemini is a Committee*. 2024. Disponible en: https://www.michaelnayna.com/p/gemini-is-a-committee. Consultado en junio de 2025.

[22] eDreams ODIGEO Tech. *Computer Vision Techniques with Python*. Medium, 2019. Disponible en: https://medium.com/edreams-odigeo-tech/computer-vision-techniques-with-python-f24a2b62a456. Consultado en junio de 2025.

[23] Interactivechaos. *Estructura de una red neuronal*. Disponible en: https://interactivechaos.com/es/manual/tutorial-de-machine-learning/estructura-de-una-red-neuronal. Consultado en junio de 2025.