UNIVERSIDAD DE GRANADA

TRABAJO FIN DE GRADO

INGENIERÍA EN INGENIERÍA INFORMATICA

Desarrollo de un robot autónomo con un sistema de detección de anomalías

Desarrollo de sistema de detección de anomalías destinado a un uso real en entornos cambiantes

Autor
Fernando González Domenech (alumno)

Directores
Javier Medina Quero (tutor1)
Aurora (tutor2)

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

—

Granada, mes de 201

Desarrollo de un robot autónomo con un sistema de detección de anomalías

Desarrollo de sistema de detección de anomalías destinado a un uso real en entornos cambiantes.

Autor
Fernando González Domenech (alumno)

Directores
Javier Medina Quero (tutor1)
Aurora (tutor2)

Desarrollo de un robot autónomo con un sistema de detección de anomalías: Desarrollo de sistema de detección de anomalías destinado a un uso real en entornos cambiantes

Fernando González Domenech (alumno)

Palabras clave: Inteligencia Artificial, Robótica, Anomalías, Automatización, Python

Resumen

Este Trabajo de Fin de Grado presenta el desarrollo de un sistema autónomo virtual capaz de detectar anomalías visuales en un entorno 3D simulado. El objetivo principal del proyecto es la integración de técnicas de inteligencia artificial y aprendizaje por refuerzo para lograr una exploración inteligente del entorno y la detección activa de elementos visualmente inusuales. El proyecto comenzó inicialmente con la intención de implementar el sistema sobre un robot físico hexápodo (Freenove Big Hexapod Kit), utilizando hardware como Raspberry Pi, cámara y sensores de ultrasonidos. Sin embargo, debido a limitaciones técnicas y de tiempo (latencias, control de motores, falta de fiabilidad en tiempo real), el enfoque fue redirigido hacia un entorno totalmente virtual desarrollado con PyBullet. El sistema final consiste en un entorno simulado tridimensional en el que un agente autónomo explora su entorno utilizando una red neuronal convolucional ResNet50 para extraer características visuales (embeddings) y un modelo LSTM para predecir y detectar anomalías en la escena. El comportamiento del agente se entrena mediante Deep Q-Learning (DQN) para maximizar la detección activa de anomalías visuales. Además, el modelo LSTM se reentrena en tiempo real con las nuevas observaciones detectadas, permitiendo una adaptación continua del sistema.

Development of an autonomous robot with an anomaly detection system: Development of an animal detection system intended for real-world use in changing environments

Fernando González Domenech (student)

Keywords: Artificial Intelligence, Robotics, Anomalies, Automation, Python

Abstract

This Thesis presents the development of a virtual autonomous system capable of detecting visual anomalies in a simulated 3D environment. The main objective of the project is the integration of artificial intelligence and reinforcement learning techniques to achieve intelligent exploration of the environment and the active detection of visually unusual elements.

The project initially began with the intention of implementing the system on a physical hexapod robot (Freenove Big Hexapod Kit), using hardware such as the Raspberry Pi, a camera, and ultrasonic sensors. However, due to technical and time constraints (latencies, motor control, lack of real-time reliability), the focus was redirected toward a fully virtual environment developed with PyBullet.

The final system consists of a simulated three-dimensional environment in which an autonomous agent explores its environment using a ResNet50 convolutional neural network to extract visual features (embeddings) and an LSTM model to predict and detect anomalies in the scene. The agent's behavior is trained using Deep Q-Learning (DQN) to maximize active detection of visual anomalies. Furthermore, the LSTM model is retrained in real time with newly detected observations, allowing for continuous adaptation of the system.

Yo, **Fernando González Domenech**, alumno de la titulación TITULACIÓN de la **Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación de la Universidad de Granada**, con DNI 77139130M, autorizo la ubicación de la siguiente copia de mi Trabajo Fin de Grado en la biblioteca del centro para que pueda ser consultada por las personas que lo deseen.

Fdo: Fernando González Domenech

Granada a 18 de Agosto de 2025.

D. **Javier Medina Quero (tutor1)**, Profesor del Área de XXXX del Departamento YYYY de la Universidad de Granada.

D. **Aurora (tutor2)**, Profesor del Área de XXXX del Departamento YYYY de la Universidad de Granada.

Informan:

Que el presente trabajo, titulado ***Desarrollo de un robot autónomo con un sistema de detección de anomalías: Desarrollo de sistema de detección de anomalías destinado a un uso real en entornos cambiantes***, ha sido realizado bajo su supervisión por **Fernando González Domenech (alumno)**, y autorizamos la defensa de dicho trabajo ante el tribunal que corresponda.

Y para que conste, expiden y firman el presente informe en Granada a X de mes de 201 .

Los directores:

Nombre Apellido1 Apellido2 (tutor1) **Nombre Apellido1 Apellido2 (tutor2)**

Agradecimientos

A mi familia y en especial a mis padres que siempre han estado hasta el último momento motivándome a seguir haciendo lo que me gusta y apasiona.

Índice

Capítulo 1

Introducción

1.1. Introducción del tema

La detección de anomalías se ha convertido en una herramienta esencial de la inteligencia artificial, ya que se encarga de identificar datos u observaciones que se apartan de un comportamiento considerado normal. Esta disciplina permite detectar fraudes financieros, intrusiones de red, irregularidades médicas y defectos de fabricación, ámbitos donde la reconocimiento de anomalías puede salvar costes, proteger la seguridad y preservar la confianza de los usuarios (Cloudera,2020) [8]. En el ámbito de las finanzas, por ejemplo, la detección de transacciones atípicas ayuda a prevenir fraudes y mejorar la calidad de los datos (MindBridge,2025) [7]. En medicina, analizar registros de pacientes y señales biomédicas permite identificar patologías incipientes, mientras que en fabricación la detección de defectos en productos industriales mejora el control de calidad (Cloudera,2020) [8].

Desde un punto de vista conceptual, las anomalías pueden clasificarse en *puntuales*, *contextuales* y *colectivas*. Las anomalías **puntuales** corresponden a valores aislados que divergen significativamente de los demás, como una transferencia bancaria desproporcionada. Las **contextuales** son aquellas que sólo resultan extrañas en un determinado contexto (por ejemplo, un gasto elevado en un periodo de gastos usualmente bajos), mientras que las **colectivas** involucran grupos de observaciones que, de forma conjunta, muestran un comportamiento inusual, como una serie de fallos consecutivos en un sistema (MindBridge,2025) [7]. Identificar estos patrones implica comprender no solo las desviaciones individuales sino también el entorno temporal y espacial en el que ocurren.

El avance de la inteligencia artificial ha permitido pasar de técnicas estadísticas simples a métodos basados en *aprendizaje profundo*. En entornos complejos con grandes volúmenes de datos e interdependencias no lineales, los métodos tradicionales presentan limitaciones al depender de supuestos sobre la distribución de los datos y ser sensibles a las atípicos[**?**]. Las re-

des neuronales profundas, en cambio, pueden modelar relaciones complejas y detectar tanto anomalías conocidas como desconocidas (Cloudera,2020) [8]. Entre las estrategias más utilizadas se encuentran los *autoencoders*, que aprenden a reconstruir datos normales y identifican anomalías a partir del error de reconstrucción; las *redes generativas adversarias (GAN)*, capaces de modelar distribuciones de datos y generar ejemplos ficticios; y las *redes recurrentes* como LSTM, que capturan las dependencias en series temporales (MindBridge,2025) [7].

La robótica móvil plantea retos específicos: los robots operan en entornos cambiantes y deben reaccionar ante sucesos imprevistos para garantizar su seguridad y la del entorno. Un sistema incapaz de identificar un objeto desconocido o una modificación del escenario se expondrá a errores y accidentes. Investigaciones recientes destacan que la robustez de un sistema automatizado se relaciona directamente con su capacidad para detectar y rectificar anomalías (Frontiers,2025) [1]. Las soluciones manuales se han vuelto impracticables ante la cantidad de información generada por sensores y cámaras, por lo que se recurre a métodos automáticos capaces de procesar datos en tiempo real.

Para detectar anomalías visuales, este proyecto utiliza **redes convolucionales** (CNN). Estas redes están diseñadas para extraer características jerárquicas de imágenes, aprendiendo automáticamente bordes, texturas y objetos mediante bloques de convolución, agrupamiento y capas densas (Shen *et al.*,2018) [5]. Su capacidad para aprender representaciones discriminativas las convierte en la base de muchos sistemas de visión por computador. Sobre estas representaciones se aplica un **modelo secuencial de tipo LSTM**, especializado en analizar datos ordenados temporalmente. Las redes *LSTM* se introdujeron para resolver el problema del desvanecimiento del gradiente y son capaces de retener información durante períodos prolongados mediante puertas de entrada, olvido y salida (Hochreiter y Schmidhuber,1997; Olah,2015) [2]. Esto las hace adecuadas para detectar cambios sutiles en secuencias de características visualmente complejas.

Finalmente, la toma de decisiones del robot se aborda mediante **aprendizaje por refuerzo profundo**. El algoritmo *Deep Q-Network* (DQN) combina un agente que interacciona con el entorno recibiendo recompensas o penalizaciones con una red neuronal que estima los valores de acción. Para estabilizar el entrenamiento se emplea una red objetivo independiente que se actualiza periódicamente (Mnih *et al.*, 2015; Inoxoft, 2023) [6]. Gracias a esta estrategia, el agente aprende políticas que maximizan la recompensa acumulada en escenarios donde la dinámica del entorno es desconocida. La integración de visión, modelado secuencial y aprendizaje por refuerzo constituye un enfoque potente para robots autónomos capaces de reconocer anomalías y actuar en consecuencia.

Además, el presente proyecto se apoya en **PyBullet**, un simulador de física de código abierto ampliamente adoptado por la comunidad investigadora.

Este entorno permite simular dinámicas de robots y entrenar algoritmos de aprendizaje por refuerzo de manera flexible y reproducible (Sunder, 2021) [3]. A diferencia de los robots físicos, la simulación elimina riesgos de seguridad, facilita la creación de distintos escenarios y acelera el proceso de entrenamiento.

1.2. Motivación

La motivación de este trabajo surge de la necesidad de desarrollar robots que sean *conscientes de su entorno* y reaccionen ante situaciones imprevistas. En sistemas automatizados, la capacidad para detectar anomalías se correlaciona con la fiabilidad y la robustez, ya que permitiría anticipar fallos y tomar medidas correctivas antes de que ocurran accidentes o interrupciones de servicio (Frontiers, 2025) [1]. La creciente digitalización de los procesos industriales y el incremento de la autonomía en vehículos, drones y robots de servicio han generado entornos de alta variabilidad donde los comportamientos raros son inevitables. Realizar la detección manual en tiempo real es inviable; por tanto, la automatización de estas tareas representa un requisito fundamental para garantizar la seguridad y la eficiencia operativa.

Otro componente motivador es la transición desde hardware real hacia **entornos simulados**. El uso de un robot físico hexápodo se encontró limitado por la latencia en el control remoto, la inestabilidad de los sensores y la dificultad de replicar condiciones consistentes de entrenamiento. En contraste, las simulaciones permiten diseñar y probar comportamientos en un entorno controlado, eliminar el desgaste físico del robot y repetir experimentos bajo las mismas condiciones. Diversos estudios subrayan que la evaluación en simulación puede ser un *sustituto escalable y reproducible* de la evaluación real, siempre que se contemplen las diferencias entre la realidad y el mundo virtual (Li *et al.*, 2024) [4]. La simulación es una herramienta imprescindible en ámbitos como la conducción autónoma, donde las políticas se prueban exhaustivamente en entornos virtuales antes de llevarse a la práctica para garantizar la seguridad (Li *et al.*, 2024) [4]. PyBullet proporciona esta flexibilidad y ha demostrado ser un entorno eficaz para entrenar y evaluar algoritmos de aprendizaje por refuerzo (Sunder, 2021) [3].

La motivación científica también reside en la integración de diferentes paradigmas de inteligencia artificial. Combinar **visión por computador**, **modelado temporal** y **aprendizaje por refuerzo** permite estudiar cómo estas disciplinas se complementan. La detección de anomalías utilizando representaciones aprendidas de imágenes se alinea con las técnicas no supervisadas, ideales cuando no se dispone de ejemplos etiquetados de todas las anomalías posibles (MindBridge, 2025) [7]. Las redes LSTM añaden la dimensión temporal, fundamental para distinguir entre variaciones transitorias y cambios persistentes. El aprendizaje por refuerzo introduce la toma de de-

cisiones adaptativa, enseñando al agente a explorar su entorno, minimizar el riesgo y maximizar la recompensa. Esta sinergia abre una línea de investigación que puede aplicarse en multitud de dominios, desde la inspección industrial hasta la exploración planetaria.

1.3. Objetivos

Objetivo general

El objetivo principal de este proyecto es **diseñar e implementar un sistema autónomo** en un entorno simulado que **explore su entorno**, detecte **anomalías visuales** en tiempo real e interactúe con él mediante **aprendizaje por refuerzo**. Para lograrlo se integrarán modelos de visión por computador, redes recurrentes y algoritmos de control, evaluando el rendimiento de cada componente y del sistema en su conjunto.

Objetivos específicos

1. **Definir el marco teórico y la taxonomía de anomalías.** Se revisará la literatura sobre detección de anomalías para establecer una clasificación clara de los diferentes tipos (puntuales, contextuales y colectivas) y se discutirán sus aplicaciones en robótica y otros campos (MindBridge, 2025) [7]. Esta revisión servirá para delimitar el alcance del problema y orientar la selección de técnicas.

2. **Diseñar y configurar el entorno de simulación.** Utilizando PyBullet, se desarrollará un escenario con obstáculos, texturas y elementos aleatorios que puedan introducir anomalías visuales. Se calibrarán los parámetros físicos y la resolución de la cámara virtual para obtener datos de alta calidad, y se incorporará un sensor de distancia que ayude al agente a evitar colisiones. Este entorno se utilizará para el entrenamiento y la evaluación del sistema (Sunder, 2021) [3].

3. **Implementar un extractor de características visuales.** Se entrenará o adaptará una red CNN para procesar las imágenes captadas por el robot. Esta red generará *embeddings* que resumen la información visual y servirán de base para la detección de anomalías. Para maximizar la generalización se explorarán diferentes arquitecturas (por ejemplo, ResNet, EfficientNet) y técnicas de transferencia de aprendizaje. Se evaluará la capacidad del extractor para diferenciar entre patrones normales y anómalos (Shen *et al.*, 2018) [5].

4. **Desarrollar un modelo secuencial para la detección.** Se diseñará un modelo LSTM que reciba como entrada las secuencias de embeddings producidas por la CNN. El objetivo será detectar cambios significativos en el tiempo que indiquen la aparición de una anomalía.

Para ello se evaluarán diferentes configuraciones (número de capas, funciones de activación, regularización) y se estudiará la influencia de la longitud de la ventana temporal sobre la sensibilidad del modelo (Olah, 2015) [2].

5. **Integrar aprendizaje por refuerzo para la exploración.** El agente aprenderá a moverse en el entorno utilizando una política basada en DQN. Se definirá una función de recompensa que incentive la exploración eficaz del entorno, penalice las colisiones y premie la detección temprana de anomalías. Se analizarán parámetros como la tasa de descuento, la estrategia de exploración (ϵ-greedy) y la actualización de la red objetivo para lograr un aprendizaje estable (Mnih *et al.*, 2015; Inoxoft, 2023) [6].

6. **Establecer métricas de evaluación y protocolos de prueba.** Se definirán métricas cuantitativas para medir la precisión en la detección de anomalías (tasa de verdaderos positivos, tasa de falsos positivos), la eficiencia de la navegación (distancia recorrida, número de colisiones) y la robustez frente a perturbaciones del entorno. La evaluación se realizará principalmente en simulación, lo que permitirá repetir experimentos de manera reproducible y comparar distintas variantes del sistema (Li *et al.*, 2024) [4].

7. **Explorar métodos de aprendizaje no supervisado y adaptación continua.** Se investigarán técnicas como autoencoders y modelos basados en densidades para complementar el enfoque LSTM. Además, se implementará un mecanismo de reentrenamiento en línea que permita incorporar nuevas anomalías detectadas al modelo, mejorando la precisión del sistema en tiempo real (MindBridge, 2025) [7].

8. **Analizar la posibilidad de transferencia a hardware real.** Aunque este proyecto se desarrolla en simulación, se evaluarán las condiciones necesarias para trasladar el sistema a un robot físico. Esto incluye estudiar técnicas de *sim-to-real* para reducir la brecha entre la simulación y la realidad, calibrar sensores reales y adaptar las estrategias de control. Este objetivo abrirá la vía para futuras investigaciones y aplicaciones comerciales (Li *et al.*, 2024) [4].

9. **Documentar y difundir los resultados.** Se elaborará una documentación detallada que describa el entorno, los algoritmos y experimentos. Siguiendo los principios de ciencia abierta, se publicarán el código y los datos para facilitar la reproducibilidad y permitir que otros investigadores extiendan el trabajo. La difusión incluirá la elaboración de un informe técnico y la preparación de posibles artículos científicos.

Estos objetivos buscan no sólo resolver un caso concreto de detección de anomalías, sino también contribuir al avance de la investigación en robots autónomos capaces de adaptarse a entornos dinámicos y, eventualmente, transferir sus capacidades a sistemas reales.

Capítulo 2

Estado del arte

2.1. Robótica móvil en entorno simulado

El desarrollo y la validación de algoritmos para robots móviles exigen entornos controlados y repetibles en los que evaluar las respuestas del robot ante distintas situaciones. La simulación cubre este papel al proporcionar espacios virtuales donde se pueden modelar los aspectos mecánicos, electrónicos y de software de un robot sin arriesgar equipos físicos ni incurrir en elevados costes de prototipado. Según el sitio oficial de MathWorks, las simulaciones permiten diseñar y analizar el comportamiento de un robot en un espacio virtual y detectar fallos de diseño, optimizar el rendimiento y validar algoritmos antes de pasar a la fase de prototipado físico [9]. Entre los beneficios más destacados figuran la seguridad y la reducción de riesgos (no hay peligro para las personas ni para los equipos), el ahorro económico al evitar la construcción de múltiples prototipos, la posibilidad de iterar con rapidez sobre diferentes configuraciones y la evaluación de algoritmos en entornos complejos que sería difícil o peligroso recrear en el mundo real [9].

Para apoyar la investigación, han surgido numerosas plataformas de simulación. A continuación se presentan las más relevantes para el aprendizaje por refuerzo y la robótica móvil, clasificadas por entornos de programación y características.

2.1.1. Entornos Python

El lenguaje Python domina gran parte del desarrollo en inteligencia artificial debido a su ecosistema de bibliotecas científicas y a la sencillez de su sintaxis. En el ámbito de la simulación, existen entornos que integran motores de física y una interfaz Python que los hace muy atractivos para la investigación y la docencia.

PyBullet. Esta biblioteca se ha consolidado como uno de los simuladores de referencia para aprendizaje por refuerzo y robótica. PyBullet es

un motor de física de código abierto que permite simular dentro de Python las dinámicas de cuerpos rígidos en tiempo real. De acuerdo con la guía de simuladores del *Robotics Knowledgebase*, PyBullet ofrece simulación de física precisa y eficiente, admite la creación de modelos y escenas personalizadas, es gratuito y multiplataforma [10]. Entre sus ventajas se encuentran la facilidad para definir robots articulados mediante URDF (Universal Robot Description Format), la posibilidad de acceder a sensores simulados y la integración con bibliotecas de aprendizaje por refuerzo como OpenAI Gym. Sin embargo, su entorno de visualización es básico y el gran abanico de opciones puede resultar abrumador para usuarios noveles [10].

Isaac Lab. Otro ejemplo de entorno Python es Isaac Lab, construido sobre el motor *Isaac Sim* de NVIDIA. Este framework es de código abierto y está acelerado por GPU; ofrece simulaciones de física de alta fidelidad mediante el motor PhysX y renderizado fotorrealista, lo que lo hace adecuado para entrenar políticas de robots antes de transferirlas al mundo real [11]. Sus principales pros son la simulación realista de sensores, la aceleración por GPU para grandes cargas de trabajo y una arquitectura modular que permite definir diversos modelos de robots [11]. Como contrapartida, requiere hardware gráfico de altas prestaciones y tiene una curva de aprendizaje más pronunciada debido a su complejidad interna [11]. Isaac Lab resulta ideal cuando se necesita entrenar políticas con información visual realista o cuando se requiere velocidad al lanzar miles de episodios en paralelo.

Otros motores. Además de PyBullet e Isaac Lab, existen numerosos motores y bibliotecas compatibles con Python como MuJoCo, Brax o Webots. MuJoCo destaca por su rendimiento en la simulación de robots articulados y es ampliamente utilizado en tareas de control robótico de alto rendimiento. Brax emplea *JAX* para realizar simulaciones rápidas en CPU y GPU, lo que permite entrenar agentes de aprendizaje por refuerzo de forma masiva. Webots se centra en la educación y la investigación, permitiendo programar en Python y proporcionando entornos 3D completos con sensores y actuadores. Estas alternativas muestran la diversidad de soluciones disponibles en el ecosistema Python para simular robots móviles.

2.1.2. MATLAB y Simulink

El conjunto formado por MATLAB y su entorno de diseño de sistemas dinámicos, Simulink, se ha convertido en una referencia para el diseño y la simulación de robots móviles. Estas herramientas permiten modelar con exactitud la cinemática y la dinámica de un robot, crear esquemas de control y validar algoritmos antes de llevarlos a la práctica. En la página de MathWorks dedicada a robots móviles se destaca que, mediante MATLAB y Simulink, es posible importar modelos virtuales del robot, refinar requisitos mecánicos y eléctricos, simular modelos de sensores (como sistemas de navegación inercial o GPS), localizar el robot mediante algoritmos de filtro

de partículas o Monte Carlo, construir mapas con técnicas de SLAM, planificar trayectorias con algoritmos A* o RRT, evaluar la suavidad y el margen de seguridad de las rutas y diseñar controladores de seguimiento de trayectorias u evitación de obstáculos [16]. Además, el entorno puede generar código de producción para hardware de destino y conectarse con simuladores externos como Gazebo [16]. Estas capacidades abarcan desde la fase de diseño conceptual hasta la implementación final, pasando por la validación de algoritmos en simulaciones. Incorporar MATLAB/Simulink en un proyecto de robótica móvil aporta rigor matemático, herramientas de análisis y una experiencia consolidada en proyectos de ingeniería.

2.1.3. Unity (ML-Agents)

Los motores de videojuegos se han convertido en plataformas de simulación atractivas por su capacidad para generar entornos 3D complejos y fotorrealistas. Unity, uno de los motores más extendidos, ofrece un kit llamado *ML-Agents Toolkit* que permite transformar escenas de Unity en entornos de entrenamiento para inteligencia artificial. La documentación oficial describe ML-Agents como un proyecto de código abierto que convierte juegos y simulaciones en entornos donde entrenar agentes inteligentes; estos agentes pueden aprender mediante aprendizaje por refuerzo, aprendizaje por imitación o neuroevolución a través de una API de Python [13]. El paquete proporciona implementaciones basadas en PyTorch de algoritmos de última generación que permiten a desarrolladores y aficionados entrenar agentes para juegos en 2D, 3D o realidad virtual/aumentada. Los agentes resultantes pueden emplearse para controlar el comportamiento de personajes no jugadores (NPC), automatizar pruebas de videojuegos o evaluar decisiones de diseño antes del lanzamiento [13]. La comunidad de investigadores de IA y desarrolladores de juegos se beneficia así de un marco común donde probar innovaciones y compartir resultados.

2.1.4. Gym de OpenAI

OpenAI Gym es una de las bibliotecas más populares para el desarrollo de algoritmos de aprendizaje por refuerzo. Consiste en un conjunto de entornos predefinidos con una interfaz homogénea que permite probar agentes en tareas que van desde juegos clásicos y control de sistemas hasta simulaciones robóticas complejas. Una guía de referencia explica que Gym es un kit de herramientas de código abierto diseñado para ayudar a los desarrolladores a construir, probar y ajustar algoritmos de aprendizaje por refuerzo. Ofrece plataformas estandarizadas con entornos predefinidos de control, Atari o robótica donde los agentes aprenden mediante prueba y error [14]. La biblioteca se integra con TensorFlow y PyTorch, lo que facilita su uso en conjunto con redes neuronales profundas. Gym simplifica la comparación de

algoritmos y promueve la reproducibilidad, ya que todos los investigadores utilizan las mismas tareas y métricas [14]. Además, permite crear entornos personalizados siguiendo un patrón común y documentado.

2.1.5. Unión de Gym y Unity

A pesar de sus orígenes distintos, OpenAI Gym y Unity se pueden combinar gracias a un adaptador desarrollado por Unity. La documentación de ML-Agents señala que muchos investigadores interactúan con simulaciones a través de la interfaz Gym; para facilitar este uso, ML-Agents proporciona un envoltorio o *wrapper* que adapta un `UnityEnvironment` a la API de Gym [15]. El *Gym wrapper* forma parte del paquete `mlagents_envs` y se usa mediante la clase `UnityToGymWrapper`. Al crear esta envoltura, el entorno de Unity se comporta como un entorno Gym y puede conectarse con algoritmos de aprendizaje por refuerzo ya implementados en bibliotecas externas [15]. El envoltorio admite opciones como exportar observaciones visuales en formato de 8 bits, aplanar espacios de acción discretos o devolver múltiples observaciones; de esta forma, los parámetros del entorno pueden adaptarse a las necesidades del agente [15]. Estas funcionalidades facilitan la integración de los entornos fotorrealistas de Unity con la infraestructura estándar de experimentos de aprendizaje por refuerzo.

2.2. Redes neuronales

El avance de la inteligencia artificial durante las últimas décadas se apoya fundamentalmente en el éxito de las redes neuronales profundas. Estos modelos están inspirados en el sistema nervioso humano: un conjunto de neuronas artificiales conectadas entre sí que cooperan para representar relaciones no triviales entre las entradas y las salidas. A diferencia de los algoritmos clásicos de aprendizaje automático, las redes neuronales no requieren que un ingeniero defina manualmente qué características son relevantes para resolver la tarea; en su lugar, aprenden de los ejemplos mediante la optimización de sus parámetros. En términos sencillos, una red neuronal está formada por capas de unidades elementales (*perceptrones*) donde cada neurona calcula una combinación lineal z de sus entradas $\mathbf{x} = (x_1, \ldots, x_n)$ y pesos $\mathbf{w} = (w_1, \ldots, w_n)$, a la que se suma un término de sesgo b, es decir

$$z = \sum_{i=1}^{n} w_i x_i + b. \tag{2.1}$$

Este valor se transforma mediante una función de activación no lineal $\phi(z)$ (por ejemplo la función sigmoide $\sigma(x) = 1/(1 + e^{-x})$, la tangente hiperbólica o la unidad lineal rectificada ReLU, definida como $\mathrm{ReLU}(x) = \max(0, x)$). La inclusión de activaciones no lineales es esencial para que la

red sea capaz de aproximar funciones arbitrarias; sin ellas, una red profunda equivale a una única transformación lineal. Investopedia señala que las redes neuronales imitan el modo en que opera el cerebro al transmitir señales a través de conexiones ponderadas y se adaptan a cambios en la entrada sin necesidad de rediseñar criterios de salida [17]. En una arquitectura de múltiples capas se apilan varias transformaciones de este tipo, lo que permite que las primeras capas detecten patrones simples (por ejemplo, bordes en una imagen) y las capas superiores combinen estas características en representaciones de alto nivel.

2.2.1. Componentes de una red neuronal y percepción lineal

Cada neurona artificial contiene varios elementos: un vector de pesos $\mathbf{w}$ que modula la contribución de cada entrada, un sesgo b que permite desplazar la función de activación y una función de activación ϕ que introduce no linealidad. Durante la fase de inferencias, la información fluye desde las entradas hacia las salidas en un proceso denominado *propagación hacia delante*. En la fase de entrenamiento, las redes ajustan sus pesos siguiendo el descenso del gradiente de una función de coste. GeeksforGeeks ilustra el funcionamiento de un perceptrón: primero se calcula la suma ponderada $a_j = \sum_i w_{i,j} x_i$, después se aplica la función de activación $o_j = \phi(a_j)$ para obtener la salida de la neurona[18]. Este esquema se repite capa a capa hasta producir la salida de la red.

Además de las activaciones ya mencionadas, existen funciones como *softmax* para problemas de clasificación multiclase. La función softmax convierte un vector de puntuaciones sin normalizar z_k en una distribución de probabilidad:

$$\text{softmax}(z_k) = \frac{e^{z_k}}{\sum_j e^{z_j}}. \tag{2.2}$$

Para evaluar la discrepancia entre la distribución predicha y la distribución real de etiquetas se utiliza la entropía cruzada, $L = -\sum_k y_k \log \hat{y}_k$. Estas funciones de pérdida permiten que el aprendizaje ajuste los pesos en las direcciones adecuadas.

2.2.2. Retropropagación de errores

El aprendizaje de una red neuronal se basa en ajustar los pesos de modo que la salida predicha se acerque a la salida deseada. Este ajuste se realiza calculando el gradiente de la función de pérdida con respecto a los parámetros mediante el algoritmo de retropropagación. Como explica GeeksforGeeks, la retropropagación (*backpropagation*) propaga el error desde la capa de salida hacia las capas anteriores y utiliza la regla de la cadena para calcular de manera eficiente las derivadas parciales de la pérdida con respecto a cada peso[18]. El algoritmo consta de dos fases:

1. **Propagación hacia delante.** Se calcula la salida de la red aplicando sucesivamente las transformaciones descritas en la ecuación (1). Cada neurona de la capa oculta recibe como entrada la salida de la capa anterior y produce una activación.

2. **Propagación hacia atrás.** Una vez obtenida la salida $\hat{y}$, se evalúa una función de coste (por ejemplo, el error cuadrático medio MSE = $(\hat{y} - y)^2$[18]). El error se propaga hacia atrás calculando los gradientes de la pérdida con respecto a cada peso. Para una neurona j con error δ_j, la actualización de un peso $w_{i,j}$ se obtiene multiplicando la tasa de aprendizaje η, el término de error δ_j y la salida o_i de la neurona anterior[18]:

$$\Delta w_{i,j} = -\eta \, \delta_j \, o_i. \tag{2.3}$$

   La tasa de aprendizaje controla la magnitud del ajuste. Un valor demasiado grande puede provocar oscilaciones en el entrenamiento, mientras que un valor demasiado pequeño ralentiza la convergencia.

Para ilustrar la retropropagación, consideremos una red de ejemplo con dos neuronas ocultas h_1, h_2 y una neurona de salida o. A partir de entradas (x_1, x_2) y pesos iniciales, se calcula la salida de cada neurona mediante la función sigmoide. Después se computa el error $E = 12(\hat{y} - y)^2$ y se derivan las actualizaciones de los pesos usando las fórmulas anteriores[18]. Este procedimiento se repite durante numerosos ciclos (épocas) hasta que la red minimiza la función de pérdida en el conjunto de entrenamiento. Algoritmos como *descenso por gradiente estocástico* (SGD), *momentum* y *Adam* introducen variaciones en esta regla básica para mejorar la eficiencia del aprendizaje.

2.2.3. Redes convolucionales de clasificación

Las redes neuronales convolucionales (*Convolutional Neural Networks*, CNN) están especialmente diseñadas para procesar datos con estructura de rejilla, como imágenes o volúmenes de vídeo. Una CNN se compone de varias capas: capas de convolución, capas de activación, capas de agrupamiento (*pooling*) y capas completamente conectadas. En una capa de convolución se aplican filtros o kernels de tamaño reducido que comparten sus pesos a lo largo de toda la entrada. Cada filtro se desliza sobre la imagen original (definiendo un *stride*) para calcular productos punto entre sus valores y los valores del fragmento de imagen correspondiente, dando lugar a un mapa de características[26]. Matemáticamente, la operación de convolución entre un tensor de entrada X y un filtro K puede expresarse como

$$(K * X)_{i,j} = \sum_{m,n} K_{m,n} \, X_{i+m,j+n}, \tag{2.4}$$

donde el núcleo K tiene dimensiones más reducidas que la entrada y se mueve sobre toda la imagen. La ventaja de esta operación es la reducción del número de parámetros y la capacidad de detectar patrones locales que se repiten en diferentes posiciones.

Tras las capas de convolución se introducen capas de activación (ReLU o tanh) para introducir no linealidad y capas de agrupamiento para reducir la resolución de los mapas de características, lo que aumenta la invarianza a translaciones y reduce la complejidad computacional. A continuación, las capas completamente conectadas combinan las características aprendidas y realizan la clasificación final mediante softmax.

Redes residuales. La profundidad se ha mostrado clave para mejorar la precisión de los modelos de clasificación; sin embargo, las redes muy profundas sufren problemas de degradación y desvanecimiento del gradiente. Las redes residuales (ResNet) introducen conexiones de *salto* que suman la entrada de un bloque a su salida de manera que la función objetivo de cada bloque aprende una función residual $F(x) \approx H(x) - x$. El bloque residual se implementa como $y = F(x) + x$, donde F suele constar de dos capas de convolución con normalización por lotes y activación ReLU[27]. He et al. demostraron que estos atajos facilitan la propagación del gradiente a través de redes muy profundas y permitieron entrenar exitosamente modelos de 50, 101 y 152 capas que vencieron la competición ImageNet en 2015. El artículo de viso.ai profundiza en cómo ResNet-50 construye bloques residuales apilados con conexiones de salto para superar la degradación y conseguir errores de clasificación muy bajos[19]. En la práctica, ResNet ha servido de base para arquitecturas posteriores que integran los atajos en redes más complejas, como DenseNet y Transformers.

Para entrenar una CNN de clasificación se optimiza la función de entropía cruzada entre la distribución de clases predicha y la distribución real. Además, técnicas como la normalización por lotes, el descenso por gradiente estocástico con momentum, el aprendizaje transferido (*transfer learning*) y la regularización (*dropout*) han permitido que las CNN alcancen resultados de vanguardia en conjuntos de datos como ImageNet y CIFAR.

2.2.4. Redes convolucionales de detección

En la detección de objetos no sólo interesa saber qué hay en una imagen, sino también dónde se encuentra cada elemento. Tradicionalmente, los detectores se basaban en un enfoque de dos etapas: un algoritmo genera propuestas de regiones (como los métodos R-CNN) y luego una red convencional clasifica cada región.

Los métodos de una sola etapa, como *YOLO* (*You Only Look Once*), formulan la detección como un único problema de regresión. YOLO divide la imagen de entrada en una cuadrícula $S \times S$. Cada celda predice un conjunto

de cajas delimitadoras y, para cada caja, un vector $(p_{obj}, b_x, b_y, b_w, b_h, c_1, \ldots, c_C)$ donde p_{obj} representa la probabilidad de contener un objeto, (b_x, b_y) son las coordenadas normalizadas del centro de la caja respecto a la celda, (b_w, b_h) son su anchura y altura relativas y $c_1, \ldots, c_C$ son las probabilidades de clase para C clases. Los parámetros (b_x, b_y) se calculan mediante una transformación sigmoide para restringir su rango a $[0, 1]$ y se ajustan a un *ancla* predefinida (p_w, p_h) mediante exponenciales, por ejemplo

$$b_x = \sigma(t_x) + c_x, \quad b_w = p_w e^{t_w}, \tag{2.5}$$

donde t_x, t_w son predicciones lineales de la red y c_x es la coordenada de la celda. Al compararse con el método de dos etapas, YOLO es extremadamente rápido: la primera versión procesaba 45 imágenes por segundo y las versiones recientes superan los 90 FPS[20]. El artículo de DataCamp destaca que YOLO formula la detección como un problema de regresión único y predice simultáneamente las coordenadas de las cajas y las probabilidades de clase[20]. Su popularidad se debe a la velocidad, la precisión y la capacidad de generalizar a distintos tamaños de objeto[20]. Las extensiones (YOLOv2, YOLOv3, YOLOv7) introducen anclas múltiples, capas de detección a diferentes escalas y estrategias de ajuste fino que mejoran la precisión en conjuntos de datos como COCO y PASCAL VOC.

En la práctica, la función de pérdida de YOLO combina términos de localización (error cuadrático sobre b_x, b_y, b_w, b_h), confianza y clasificación, ponderados para equilibrar las contribuciones. Las predicciones se evalúan mediante la intersección sobre unión (IoU) entre las cajas predichas y las cajas reales.

2.2.5. LSTM para secuencias e imágenes

Las redes recurrentes (RNN) son idóneas para procesar secuencias de datos, ya que mantienen un estado interno que se actualiza con cada entrada y permite recordar información previa. Sin embargo, las RNN básicas sufren los problemas de gradiente desvanecido y explotado que impiden aprender dependencias de largo alcance. Para solventarlo, Hochreiter y Schmidhuber propusieron en 1997 las redes de memoria a corto y largo plazo (*Long Short-Term Memory*, LSTM). Estas introducen una estructura de memoria controlada por tres compuertas (*gates*) que gestionan de forma diferenciada la información que se añade, se olvida o se expulsa de la celda. Colah explica que las LSTM aprenden dependencias de largo plazo recordando información durante largo tiempo por defecto[21].

Matemáticamente, en cada instante t la LSTM actualiza su estado de acuerdo con las siguientes ecuaciones. Primero se calculan las activaciones de las compuertas de entrada i_t, olvido f_t y salida o_t aplicando una función sigmoide a combinaciones lineales de la entrada x_t y el estado oculto anterior h_{t-1}: $\mathrm{i}_t = \sigma(W_{xi}x_t + W_{hi}h_{t-1} + b_i)$,

$f_t = \sigma(W_{xf}x_t + W_{hf}h_{t-1} + b_f)$,
$o_t = \sigma(W_{xo}x_t + W_{ho}h_{t-1} + b_o)$.

A continuación se calcula la celda candidata $\tilde{c}_t$ usando una función tangencial y se actualiza el estado de memoria c_t como una mezcla entre la celda anterior y la nueva información[28]: $\tilde{c}_t = \tanh(W_{xc}x_t + W_{hc}h_{t-1} + b_c)$, $c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t$, donde $\odot$ denota el producto elemento a elemento. Finalmente el estado oculto se obtiene filtrando la celda mediante la compuerta de salida:

$$h_t = o_t \odot \tanh(c_t). \tag{2.6}$$

Gracias a estas compuertas, la LSTM decide de manera adaptativa qué información conservar y cuál descartar. Su capacidad para recordar secuencias largas las hace idóneas para tareas como la generación de descripciones de imágenes y vídeos, donde una CNN extrae representaciones visuales cuadro a cuadro y una LSTM genera una frase coherente palabra a palabra.

2.2.6. Transformers y BERT

Aunque las LSTM y las redes recurrentes lograron grandes avances en modelado secuencial, presentan dificultades para paralelizar el procesamiento debido a su naturaleza secuencial. Los *Transformers*, introducidos por Vaswani et al. en 2017, se basan exclusivamente en mecanismos de atención y eliminan la necesidad de recurrencia. La atención auto-regresiva evalúa la relevancia de cada elemento de la secuencia con respecto a los demás mediante el producto escalar entre vectores de consulta Q, claves K y valores V. El módulo de atención escala el producto para evitar variancias explosivas y aplica una normalización suave (*softmax*)[22]:

$$\text{Attention}(Q, K, V) = \text{softmax}\Big(\frac{QK^T}{\sqrt{d_k}}\Big)V. \tag{2.7}$$

El uso de varias cabezas de atención permite que el modelo aprenda relaciones a diferentes escalas dividiendo Q, K y V en múltiples subespacios y procesándolos en paralelo, para luego concatenar y mezclar sus resultados[29]. Cada capa transformer incluye además normalización por capas y conexiones residuales, lo que facilita el entrenamiento de redes muy profundas.

El modelo *BERT* (*Bidirectional Encoder Representations from Transformers*) es uno de los transformers más influyentes. Devlin y colaboradores propusieron entrenar un codificador bidireccional aplicando dos tareas de preentrenamiento sobre grandes corpus de texto: el enmascaramiento de palabras (MLM) y la predicción de la siguiente frase (NSP). El MLM consiste en sustituir aleatoriamente el 15 las palabras por un token especial $[MASK]$ y pedir al modelo que recupere las palabras originales; el NSP entrena al modelo a predecir si dos frases son consecutivas en el corpus[23]. BERT está

disponible en diferentes tamaños (BASE con 110 M de parámetros y 12 capas; LARGE con 340 M de parámetros y 24 capas) y, tras el preentrenamiento, se adapta a tareas específicas añadiendo una capa de salida apropiada. El artículo original destaca que este enfoque de preentrenamiento bidireccional logra resultados de vanguardia en numerosos benchmarks de comprensión de lenguaje[30].

2.2.7. Autoencoders y generación de *embeddings*

Los autoencoders son redes neuronales que aprenden a reproducir su propia entrada a través de una representación intermedia de menor dimensión. Un autoencoder consta de dos partes: un codificador f que transforma la entrada x en un vector latente z, y un decodificador g que reconstruye una aproximación $\hat{x}$ de la entrada a partir de z. La red se entrena minimizando una función de pérdida que cuantifica la diferencia entre x y $\hat{x}$, generalmente el error cuadrático medio o la entropía cruzada. La documentación de V7 Labs describe los autoencoders como modelos no supervisados que generan codificaciones compactas de los datos de entrada y constan de un codificador, un cuello de botella donde se almacena la representación comprimida y un decodificador que reconstruye los datos[24]. El cuello de botella obliga al modelo a capturar los patrones esenciales; cuanto más pequeño sea, mayor compresión y mayor generalización.

GeeksforGeeks explica que la arquitectura del autoencoder contiene una capa de entrada, varias capas ocultas, una capa latente y un decodificador espejo; el entrenamiento se realiza mediante retropropagación y optimiza el error de reconstrucción[31]. Por ejemplo, para una entrada x y una reconstrucción $\hat{x} = g(f(x))$, la pérdida de reconstrucción se define como

$$L_{\mathrm{AE}} = x - \hat{x}^2. \tag{2.8}$$

Algunas variantes importantes son los autoencoders *sparsos*, que imponen penalizaciones para que la representación latente sea poco densa; los autoencoders *denoising*, que se entrenan a reconstruir la entrada a partir de una versión ruidosa; y los autoencoders *variacionales* (VAE), donde la representación latente se modela como una distribución gaussiana y se añade un término de divergencia de Kullback–Leibler para regularizar la distribución latente. Estas variaciones permiten generar nuevos ejemplos (por ejemplo, imágenes realistas) y son la base de modelos generativos de gran éxito. En aplicaciones de detección de anomalías, los autoencoders sirven para construir vectores de *embeddings* que representan el estado del entorno; posteriormente se compara la reconstrucción con la entrada y se detectan anomalías cuando el error de reconstrucción excede un umbral[31].

2.3. Detección de anomalías

La detección de anomalías consiste en identificar patrones o puntos de datos que se apartan significativamente de un comportamiento considerado normal. En el contexto de robótica móvil y sistemas autónomos, la capacidad de detectar estas desviaciones es fundamental para garantizar la seguridad y fiabilidad de los robots, prever fallos y actuar de forma preventiva. A continuación se describen los principales enfoques y herramientas de detección de anomalías utilizados en este proyecto, con un énfasis en la detección de anomalías en series temporales, la comparación de representaciones mediante la similitud coseno y el uso de redes neuronales.

2.3.1. Anomalías en series temporales

Una serie temporal es una secuencia de observaciones ordenada en el tiempo. La detección de anomalías en series temporales tiene aplicaciones prácticas en múltiples ámbitos: en la industria, permite identificar fallos en equipos mediante el seguimiento de señales de sensores; en finanzas, ayuda a descubrir fraudes y movimientos atípicos; y en ciberseguridad, sirve para detectar patrones irregulares en el tráfico de red. Según la guía de GeeksforGeeks sobre detección de anomalías en series temporales, la detección de anomalías consiste en encontrar observaciones que se desvían significativamente de los patrones esperados y puede llevarse a cabo mediante métodos no supervisados como el clustering, el análisis de componentes principales (PCA) o los autoencoders[32]. Estas técnicas no requieren que existan ejemplos etiquetados de anomalías y se adaptan bien a entornos donde las irregularidades son infrecuentes o difíciles de etiquetar.

Tipos de anomalías y retos. En las series temporales encontramos distintas clases de anomalías: *anomalías puntuales*, que afectan a un único valor (por ejemplo, un pico abrupto en una señal); *anomalías colectivas*, donde un conjunto de puntos presenta un comportamiento anómalo de forma colectiva; y *anomalías por intervalo*, donde durante un periodo aparece un patrón inusual. Además, existen problemas específicos a la hora de detectar anomalías en series temporales. La presencia de estacionalidad y tendencia en muchos procesos hace difícil distinguir variaciones naturales de anomalías; el ruido y la variabilidad inherentes a datos reales ocultan anomalías; y los patrones pueden evolucionar con el tiempo, lo que requiere métodos adaptativos[32]. Estos retos motivan el empleo de algoritmos que se ajusten dinámicamente y que sean capaces de separar las fluctuaciones propias del sistema de comportamientos verdaderamente anómalos.

Técnicas estadísticas y de aprendizaje automático. Los enfoques tradicionales, como el cálculo del *z-score*, las medias móviles o la descom-

posición en tendencia y componente residual, permiten identificar puntos que se alejan de la distribución esperada. No obstante, estos métodos suelen fallar cuando las anomalías son complejas o cuando los datos son de alta dimensión. Por ello, se emplean enfoques de aprendizaje automático que aprenden las características de la serie. Por ejemplo, los algoritmos de clustering (*k-means*, DBSCAN) detectan puntos que no pertenecen a grupos densos; los bosques de aislamiento (*Isolation Forest*) aíslan los puntos raros mediante árboles aleatorios; y los autoencoders comprenden la estructura de los datos para reconstruirla y estimar errores de reconstrucción[32].

Ejemplo de aplicación. Imaginemos un robot móvil que monitoriza su temperatura interna mediante un sensor. La serie temporal correspondiente presenta un comportamiento casi constante con pequeñas fluctuaciones. Si de repente la temperatura aumenta de forma brusca o disminuye por debajo de un umbral, se produce una anomalía. Esta anomalía puede ser puntual (un pico aislado por un choque térmico) o colectiva (una tendencia creciente durante varios minutos que anticipa un fallo del sistema de refrigeración). Un sistema de detección debe reconocer estas irregularidades para activar mecanismos de protección.

2.3.2. Comparativa de embeddings con similitud coseno

En muchas aplicaciones de detección de anomalías, los datos brutos se transforman en representaciones vectoriales o *embeddings* mediante modelos de aprendizaje profundo. Estas representaciones encapsulan información semántica o estructural de los datos originales y permiten comparar de forma cuantitativa la similitud entre observaciones. La comparación de *embeddings* se lleva a cabo utilizando métricas de similaridad o distancia entre vectores. Entre las métricas más utilizadas se encuentran la distancia euclídea, el producto escalar y la similitud coseno.

Definición de la similitud coseno. La similitud coseno mide el ángulo entre dos vectores en un espacio vectorial normalizado. Se calcula tomando el producto punto de los vectores $\mathbf{a}$ y $\mathbf{b}$ y dividiendo por el producto de sus normas:

$$\mathrm{sim}_{\cos}(\mathbf{a}, \mathbf{b}) = \frac{\mathbf{a} \cdot \mathbf{b}}{\|\mathbf{a}\| \, \|\mathbf{b}\|}. \tag{2.9}$$

Un valor de 1 indica que los vectores están alineados (misma dirección), un valor de 0 que son ortogonales (no comparten información) y un valor de -1 que apuntan en direcciones opuestas. La similaridad coseno es independiente de la magnitud de los vectores y depende únicamente de la orientación. Esta propiedad resulta útil al comparar vectores normalizados,

ya que vectores de distinto tamaño pueden tener la misma dirección y, por tanto, la misma similitud[33].

Ventajas y limitaciones. Las métricas de similitud deben elegirse en función del modelo de *embeddings*. La guía de Pinecone señala que si un modelo ha sido entrenado con la similitud coseno, conviene utilizar la misma métrica para comparar vectores, ya que de lo contrario se pueden obtener resultados inconsistentes[33]. La similitud coseno es apropiada cuando la información está contenida en la dirección del vector (por ejemplo, en representaciones semánticas de palabras o en estados de un robot) y no en su módulo. En contraste, la distancia euclídea considera tanto la magnitud como la dirección y puede ser más adecuada cuando el tamaño del vector tiene significado (por ejemplo, para contar eventos). La elección de la métrica afecta a la detectabilidad de anomalías: dos *embeddings* que difieren en magnitud pero mantienen la misma dirección tendrán similitud coseno alta, mientras que la distancia euclídea puede indicar que son dispares.

Aplicación en detección de anomalías. En el sistema desarrollado en este proyecto, cada observación del robot se codifica en un vector latente mediante un autoencoder o una red convolucional. Para detectar anomalías, se compara el *embedding* actual del entorno con una base de *embeddings* de referencia que representan el estado normal. Si la similitud coseno entre el vector actual y los vectores normales es menor que un umbral predefinido, se considera que el estado es anómalo. Este enfoque tiene la ventaja de ser insensible a escalares y centrarse en la orientación de los vectores, lo que ayuda a detectar cambios sutiles en la estructura de las observaciones.

2.3.3. Redes neuronales para detección de anomalías

Las redes neuronales se han convertido en herramientas clave para la detección de anomalías, debido a su capacidad para aprender representaciones de alto nivel y modelar dependencias temporales o espaciales en los datos. En esta sección se describen tres enfoques principales: redes convolucionales unidimensionales (1D-CNN), autoencoders y LSTM autoencoders.

Redes convolucionales unidimensionales. Las 1D-CNN aplican convoluciones sobre secuencias de valores, lo que las hace idóneas para analizar datos temporales de sensores. Un estudio reciente sobre detección de fallos en máquinas de control numérico concluyó que las 1D-CNN son adecuadas para datos secuenciales y, a diferencia de los métodos tradicionales basados en atributos manuales, pueden aprender automáticamente las dependencias temporales y patrones directamente a partir de los datos brutos[34]. Captar estas dependencias es crucial para detectar anomalías que se manifestan

como variaciones sutiles a lo largo del tiempo. En un robot móvil, un 1D-CNN podría procesar la evolución de la distancia medida por un sensor ultrasónico y detectar comportamientos anómalos, como oscilaciones de frecuencia inusuales, que podrían indicar problemas en el sensor o obstáculos inesperados.

Autoencoders. Los autoencoders son redes neurales auto-supervisadas que aprenden a reproducir su entrada a través de una representación comprimida o latente. La idea básica consiste en entrenar una red con un codificador f y un decodificador g de modo que la salida $\hat{x} = g(f(x))$ se aproxime a la entrada x. Un autoencoder minimiza la diferencia entre la entrada y su reconstrucción mediante una función de pérdida, como el error cuadrático medio $L_{\text{AE}} = \|x - \hat{x}\|^2$. La sección de teoría de un estudio sobre autoencoders para señales de sensores explica que el autoencoder es un modelo no supervisado cuyo objetivo es reconstruir su entrada y consta de un codificador que proyecta los datos al espacio latente y un decodificador que los reconstruye[35]. El espacio latente retiene las características fundamentales de la entrada, por lo que los vectores latentes pueden utilizarse como *embeddings* representativos[35]. Para la detección de anomalías, se entrena el autoencoder únicamente con datos normales, de forma que cuando se procesa un dato anómalo el error de reconstrucción sea alto. Se selecciona un umbral sobre el error de reconstrucción y aquellas observaciones cuyo error supere el umbral se clasifican como anómalas[35]. En el contexto de este proyecto, los autoencoders permiten comprimir imágenes o lecturas de sensores en un vector latente y medir la discrepancia entre la entrada y la reconstrucción para detectar cambios inusuales en el entorno.

Autoencoders LSTM. Para series temporales, los autoencoders pueden combinarse con LSTM en el codificador y/o decodificador para capturar dependencias a largo plazo. El estudio mencionado destaca que los autoencoders LSTM utilizan la memoria a corto y largo plazo de las LSTM, lo que los hace idóneos para analizar datos secuenciales y detectar anomalías en señales de sensores[35]. La principal diferencia entre un autoencoder estándar y uno LSTM radica en la estructura recurrente de este último: las LSTM procesan la secuencia paso a paso y mantienen un estado interno que captura la historia, de modo que pueden detectar patrones temporales complejos. En nuestro sistema, un autoencoder LSTM puede procesar ventanas temporales de imágenes o vectores de sensores para aprender las transiciones normales y señalar como anomalías aquellas secuencias que produzcan un error de reconstrucción elevado.

Ventajas de las redes neuronales. El uso de redes neuronales para la detección de anomalías ofrece varias ventajas frente a los métodos estadísti-

cos: pueden manejar datos de alta dimensión sin necesidad de ingeniería de características, son capaces de captar patrones no lineales complejos y se adaptan a cambios en los datos mediante reentrenamiento. Las 1D-CNN proporcionan rapidez y eficiencia para series temporales, mientras que los autoencoders y autoencoders LSTM permiten definir umbrales automáticos basados en reconstrucción y comparan *embeddings* de manera significativa. En conjunto, estos modelos constituyen la base de nuestro sistema de detección de anomalías en entornos simulados y demuestran que las técnicas de aprendizaje profundo son aptas para supervisar robots en escenarios complejos.

2.4. Aprendizaje por refuerzo

El aprendizaje por refuerzo (RL, del inglés *reinforcement learning*) se ha convertido en una de las disciplinas más activas de la inteligencia artificial moderna. A diferencia de los enfoques supervisados, donde un algoritmo aprende a mapear entradas en etiquetas conocidas, y de los enfoques no supervisados, que extraen patrones de datos sin etiquetar, el RL se centra en la interacción de un agente con un entorno dinámico. El agente observa un estado, ejecuta una acción y recibe una señal de refuerzo (recompensa o penalización); con el tiempo aprende una política que maximiza la recompensa acumulada. Esta forma de aprendizaje no requiere etiquetas previas: el agente aprende mediante prueba y error a partir de la retroalimentación del entorno. GeeksforGeeks destaca que el RL se caracteriza por el aprendizaje mediante interacción, la ausencia de datos etiquetados y la utilización de algoritmos como Q-learning, SARSA o DQN[36]. La ventaja de este enfoque es que los agentes pueden aprender a tomar decisiones óptimas en entornos desconocidos, como en juegos, robótica o logística.

En esta sección se presenta una revisión de los conceptos y algoritmos de RL, organizada de la siguiente manera: una introducción a los fundamentos teóricos, una discusión sobre variantes supervisadas y comportamientos clonados, la presentación de enfoques no supervisados y auto-regulados, un análisis de dos algoritmos emblemáticos (PPO y DQN) y, finalmente, una reflexión sobre métodos híbridos que combinan elementos de aprendizaje supervisado y no supervisado.

2.4.1. Fundamentos del aprendizaje por refuerzo

Un problema de RL se modela formalmente mediante un proceso de decisión de Markov (MDP). Un MDP se define por un conjunto de estados $\mathcal{S}$, un conjunto de acciones $\mathcal{A}$, una dinámica de transición $P(s' \mid s, a)$ que describe la probabilidad de ir al estado s' tras ejecutar la acción a en el estado actual s, y una función de recompensa $r(s, a)$ que asigna un valor al agente por su acción. Un agente elige acciones siguiendo una política $\pi(a \mid s)$ y el

objetivo es maximizar la recompensa acumulada (también llamada *return*) definida como

$$G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}, \tag{2.10}$$

donde $\gamma \in [0, 1]$ es el factor de descuento que pondera las recompensas futuras. Cuando $\gamma = 0$ el agente solo se preocupa por las recompensas inmediatas; cuando γ es cercano a 1, da más importancia a las recompensas a largo plazo. El agente aprende a través del ciclo *percepción–acción*: observa un estado s, selecciona una acción a según su política, recibe una recompensa r y observa un nuevo estado s'. Los componentes clave son el agente, el entorno, los estados, las acciones, la política, la función de valor y la función de recompensa. GeeksforGeeks explica que un agente de RL interactúa con el entorno en este ciclo y usa funciones de valor y políticas para estimar qué acciones deben tomarse[37].

Funciones de valor y ecuaciones de Bellman. Para evaluar la calidad de las acciones, el RL usa dos funciones fundamentales. La función de valor del estado bajo una política π, denotada $V^\pi(s)$, es la expectativa del retorno cuando el agente empieza en el estado s y sigue la política π en adelante:

$$V^\pi(s) = E_\pi[G_t \mid s_t = s]. \tag{2.11}$$

La función de valor de acción (o *función Q*) se define como

$$Q^\pi(s, a) = E_\pi[G_t \mid s_t = s, a_t = a]. \tag{2.12}$$

Ambas funciones satisfacen las ecuaciones de Bellman, que expresan el valor de un estado o acción en términos de las recompensas inmediatas y el valor de estados futuros. Por ejemplo, para la función Q de la política óptima π^* se cumple

$$Q^*(s, a) = E[r(s, a) + \gamma \max_{a'} Q^*(s', a') \mid s, a]. \tag{2.13}$$

Aprender las funciones de valor es uno de los enfoques más comunes para resolver problemas de RL: si se obtiene Q^*, el agente puede actuar de forma óptima seleccionando la acción que maximiza $Q^*(s, a)$ en cada estado.

Algoritmos de control. Los algoritmos de RL pueden clasificarse en *basados en valores* (que estiman Q^* o V^*), *basados en políticas* (que optimizan directamente la política sin estimar funciones de valor) y *actor–crítico* (combinan ambos). Los métodos basados en valores, como Q-learning y SARSA, actualizan las estimaciones de Q o V mediante la ecuación de diferencias temporales (TD). Por ejemplo, la regla de actualización de Q-learning para el par (s, a) es

$$Q(s,a) \leftarrow Q(s,a) + \alpha\Big[r(s,a) + \gamma \max_{a'} Q(s',a') - Q(s,a)\Big], \tag{2.14}$$

donde $\alpha \in (0,1]$ es la tasa de aprendizaje. Este tipo de actualización aproxima la función de valor a través de diferencias temporales y no requiere un modelo del entorno. Otros métodos, como SARSA, utilizan la acción realmente ejecutada en el nuevo estado en lugar de la acción que maximiza Q.

Los métodos basados en políticas buscan maximizar la recompensa esperada siguiendo gradientes de rendimiento respecto a los parámetros de la política. Se basan en la identidad de la derivada del rendimiento para actualizar los parámetros de la política en la dirección que aumenta la recompensa. La familia de métodos actor–crítico combina un *actor* (que actualiza la política) y un *crítico* (que estima la función de valor) para reducir la varianza en las actualizaciones.

2.4.2. Aprendizaje por refuerzo supervisado

Aunque el aprendizaje por refuerzo se distingue de los enfoques supervisados por no requerir etiquetas, existen variantes en las que se incorpora supervisión adicional. En el llamado *aprendizaje por refuerzo supervisado* el agente dispone de información extra —por ejemplo, demostraciones de expertos o comentarios humanos— que guían la optimización de la política. Esta supervisión puede codificarse como un término adicional en la función de pérdida o como un conjunto de muestras etiquetadas que se utilizan para inicializar la política.

Una técnica habitual es el "*reinforcement learning from human feedback*" (RLHF), donde evaluadores humanos puntúan las acciones de un agente; estas puntuaciones se convierten en recompensas de entrenamiento. Otra forma de supervisión es proporcionar un conjunto de trayectorias expertas que se utilizan como demostradoreas para acelerar el aprendizaje. Este escenario, denominado *inicialización con demostraciones*, combina el aprendizaje por refuerzo con técnicas supervisadas, como regresión o clasificación, para que el agente aprenda de forma más eficaz desde el principio.

La principal ventaja de incorporar supervisión es que se reduce la fase de exploración aleatoria y se mejora la seguridad al evitar que el agente realice acciones peligrosas durante el entrenamiento. Sin embargo, demasiada supervisión puede limitar la capacidad del agente para descubrir soluciones novedosas.

2.4.3. Aprendizaje por imitación: *behavior cloning* y más

El aprendizaje por imitación consiste en entrenar agentes a partir de trayectorias de demostración proporcionadas por expertos humanos. Una

de las técnicas más sencillas es el *clonado de comportamiento* ("behavior cloning", BC), que convierte la tarea de RL en un problema de aprendizaje supervisado. Según una descripción general de AI Online Course, el clonado de comportamiento recopila datos de cómo un humano realiza una tarea (por ejemplo, conducir o manipular objetos) y entrena un modelo de IA para imitar esas acciones[38]. Las aplicaciones incluyen coches autónomos, robots industriales y agentes que juegan a videojuegos[38]. El BC es eficiente y puede ser muy preciso si los datos son de alta calidad[38]; sin embargo, presenta limitaciones como sobreajuste a las demostraciones, dificultad para generalizar a situaciones no vistas y falta de creatividad[38]. Para paliar estos problemas se utilizan algoritmos que mezclan BC con exploración, como DAgger (*dataset aggregation*), que intercala demostraciones humanas con retroalimentación del propio agente.

Otras variantes de aprendizaje por imitación. Además del clonado de comportamiento, existen técnicas de imitación basadas en el principio de máximo margen ("learning from demonstration") y métodos basados en adversarios como *Generative Adversarial Imitation Learning* (GAIL), donde un discriminador trata de distinguir entre comportamientos humanos y del agente mientras el generador (la política) intenta engañar al discriminador. Estas técnicas combinan ideas de redes adversarias y aprendizaje por refuerzo para imitar destrezas complejas.

2.4.4. Aprendizaje por refuerzo no supervisado

Si bien el RL tradicional se basa en recompensas extrínsecas proporcionadas por el entorno, se ha observado que agentes así entrenados son estrechos y poco generalistas. Una línea de investigación emergente es el *aprendizaje por refuerzo no supervisado*, que busca aprender habilidades útiles sin recompensas externas. El blog del Berkeley AI Research (BAIR) señala que los agentes de RL convencional están supervisados por una recompensa extrínseca, lo que limita su capacidad para generalizar; en cambio, el RL no supervisado define recompensas intrínsecas a partir de tareas auto-supervisadas y persigue el objetivo de aprender comportamientos útiles sin especificar tareas[39]. La idea es que el agente se auto-motiva utilizando medidas de curiosidad, sorpresa o diversidad, lo que le permite explorar el entorno de forma eficaz y adquirir representaciones generales que luego facilitan la adaptación a tareas descendentes. El mismo artículo explica que la diferencia clave entre RL supervisado y no supervisado es que en el primer caso la supervisión proviene de recompensas extrínsecas (diseñadas o proporcionadas por humanos), mientras que en el segundo el agente se retribuye mediante una señal intrínseca derivada de una tarea auto-supervisada[39]. Existen diferentes categorías de métodos: los basados en conocimiento maximizan el error de un modelo predictivo (curiosidad, RND), los basados en

datos maximizan la diversidad de las observaciones (APT, ProtoRL) y los basados en competencia maximizan la información mutua entre estados y un vector latente de habilidades[39].

Objetivos auto-supervisados. Una forma de RL no supervisado consiste en preentrenar al agente mediante un objetivo auto-supervisado (por ejemplo, predecir estados futuros o reducir la incertidumbre de un modelo del mundo) y, más adelante, ajustar la política para optimizar recompensas extrínsecas. Este esquema de preentrenamiento se ha popularizado en visión y lenguaje mediante modelos contrastivos y autoencoders, y se adapta al RL mediante técnicas como *curiosity-driven exploration* o *world models*. La idea subyacente es que un agente que aprende dinámicas básicas del entorno puede adaptarse más rápido a nuevas tareas que uno entrenado desde cero.

2.4.5. Proximal Policy Optimization (PPO)

Proximal Policy Optimization (PPO) es un algoritmo de optimización de políticas que combina la estabilidad de Trust Region Policy Optimization (TRPO) con la simplicidad de métodos basados en gradiente. La documentación de Spinning Up explica que PPO surge del mismo planteamiento que TRPO: encontrar la mayor mejora posible de la política usando los datos disponibles sin provocar un colapso del rendimiento[40]. En lugar de resolver un problema convexo de segundo orden, PPO emplea métodos de primer orden con ajustes que limitan la desviación entre la política nueva y la antigua. Existen dos variantes: *PPO-Penalty*, que penaliza la divergencia de Kullback–Leibler (KL) en la función objetivo, y *PPO-Clip*, que introduce un factor de recorte en la razón de probabilidades para evitar actualizaciones excesivas[40]. La versión usada habitualmente es PPO-Clip.

Los hechos rápidos del algoritmo indican que PPO es de tipo *on-policy* (usa datos recientes), admite espacios de acción discretos o continuos y soporta paralelización[40]. La función objetivo de PPO-Clip para actualizar la política a partir del estado s y la acción a con ventaja estimada $A^{\pi_k}(s,a)$ y razón de probabilidades $r(\theta) = \pi_\theta(a \mid s)/\pi_{\theta_k}(a \mid s)$ es

$$L^{\mathrm{PPO}}(\theta) = E\Big[\mathrm{mín}(r(\theta)A^{\pi_k}(s,a),\ \mathrm{clip}(r(\theta), 1-\epsilon, 1+\epsilon)A^{\pi_k}(s,a))\Big], \quad (2.15)$$

donde ϵ es un hiperparámetro (por ejemplo, 0,2). La operación clip garantiza que los cambios en la política permanezcan dentro de una región fiable. Durante el entrenamiento se realizan varias épocas de descenso estocástico del gradiente sobre este objetivo usando datos de rollouts recientes. PPO ha demostrado gran eficacia en tareas de locomoción y control continuo, y su simplicidad ha popularizado su uso en videojuegos y simuladores.

2.4.6. Deep Q-Network (DQN)

Los algoritmos basados en valores sufren cuando el espacio de estados es grande o continuo, ya que almacenar una tabla de valores se vuelve inviable. Deep Q-Learning (DQN) aborda este problema usando redes neuronales profundas para aproximar la función $Q(s, a; \theta)$. GeeksforGeeks explica que DQN es una extensión de Q-learning que usa aprendizaje profundo para estimar valores en entornos con grandes espacios de estados[41]. En lugar de una tabla, una red neuronal procesa las observaciones (por ejemplo, píxeles de videojuegos) y produce los valores Q de cada acción[41].

La arquitectura típica de un DQN incorpora tres componentes clave[41]:

1. **Red neuronal**: aproxima la función $Q(s, a; \theta)$. En juegos Atari, las entradas son imágenes y las salidas son las estimaciones de valor para cada acción.

2. ***Experience replay***: almacena las transiciones (s, a, r, s') en una memoria y reutiliza muestras aleatorias para romper la correlación temporal entre ejemplos consecutivos[41].

3. **Red objetivo**: una copia de la red Q con parámetros θ^-. Se actualiza periódicamente y se utiliza para calcular los valores objetivo durante el entrenamiento, lo que estabiliza los gradientes[41].

La función de pérdida de DQN mide la diferencia entre el valor predicho y el valor objetivo:

$$L(\theta) = E\Big[\big(r + \gamma \max_{a'} Q(s', a'; \theta^-) - Q(s, a; \theta)\big)^2\Big]. \tag{2.16}$$

Durante el entrenamiento, se inicializa la red principal y la red objetivo, se define una tasa de aprendizaje α, un factor de descuento γ y un índice de exploración ϵ para la política *-greedy*. A cada paso se ejecuta una acción aleatoria con probabilidad ϵ (exploración) o la acción de mayor Q con probabilidad $1 - \epsilon$ (explotación); se almacena la transición en la memoria, se extrae un lote de mini-experiencias y se actualizan los parámetros θ minimizando $L(\theta)$. Periódicamente, los parámetros θ se copian a θ^-. El DQN fue popularizado por el éxito de DeepMind en videojuegos Atari y ha sido extendido mediante variantes como Double DQN, Dueling DQN y Prioritized Experience Replay.

2.4.7. Métodos híbridos: supervisión y no supervisión

En los últimos años han surgido enfoques que combinan aprendizaje supervisado, aprendizaje por refuerzo y técnicas no supervisadas. El objetivo

de estos métodos híbridos es aprovechar los puntos fuertes de cada paradigma para acelerar la exploración, mejorar la generalización y reducir la necesidad de recompensas extrínsecas. A continuación se describen tres líneas de investigación representativas.

Preentrenamiento no supervisado seguido de RL supervisado. El enfoque más sencillo consiste en preentrenar una red neuronal usando objetivos auto-supervisados (por ejemplo, autoencoders, predicción de estados futuros, contrastive learning) para aprender representaciones generales del entorno. Posteriormente, estas representaciones se utilizan como entradas para un algoritmo de RL supervisado con recompensas extrínsecas. Estudios como el de BAIR señalan que un preentrenamiento no supervisado permite que el agente aprenda comportamientos útiles sin recompensa y se adapte rápidamente a tareas descendentes. Este esquema se asemeja al uso de modelos de lenguaje preentrenados en NLP.

Aprendizaje por refuerzo con demostraciones y exploración inherente. Otra línea consiste en combinar demostraciones humanas con RL. Primero se usa clonado de comportamiento para aproximar la política del experto; luego se aplica RL para mejorar la política con recompensas extrínsecas. Métodos como Deep Q-Learning from Demonstrations (DQfD) o Normalized Actor Critic integran los datos de demostración en la función de pérdida para guiar el agente en etapas tempranas. Este enfoque reduce la exploración aleatoria y permite aprender políticas robustas incluso en entornos escasos en recompensas. La fusión de BC y DQN es un ejemplo de este esquema híbrido.

Refuerzo con recompensas intrínsecas y retorno retroalimentado. Finalmente, algunos métodos combinan recompensas intrínsecas con recompensas extrínsecas para equilibrar exploración y explotación. Ejemplos son los agentes de curiosidad, que incentivan la visita de estados inesperados, y los algoritmos de *goal-conditioned RL* (aprendizaje orientado a metas) donde el agente recibe recompensas cuando alcanza sub-objetivos auto-generados. Estas señales intrínsecas pueden considerarse como supervisión interna que guía la exploración, mientras que las recompensas externas aseguran que el agente aprende a resolver la tarea principal.

Capítulo 3

Planificación del proyecto

3.1. Introducción

La fase de planificación es un componente esencial del ciclo de vida de cualquier proyecto. En este trabajo se ha dedicado un esfuerzo considerable a definir las tareas, evaluar las tecnologías y organizar las distintas etapas que permiten alcanzar los objetivos del proyecto de forma eficiente. Para ello, se llevó a cabo un estudio inicial de las tecnologías relacionadas con robótica móvil, visión por computador y aprendizaje por refuerzo. Basándose en ese análisis se elaboró un diseño del sistema que establecía la arquitectura, los módulos y las interacciones. A continuación se implementó el programa, se evaluó mediante pruebas cuantitativas en el entorno simulado y se elaboró la documentación correspondiente. Además, a lo largo de los seis meses que duró el proyecto se mantuvieron reuniones periódicas con el tutor para revisar el avance, ajustar la planificación y resolver dudas. Esta sección describe cómo se definieron los objetivos y el alcance, se realizó el diseño conceptual, se programó un calendario con las fases del proyecto y se analizaron los costes asociados.

3.2. Definición de objetivos y alcance

Antes de iniciar cualquier proyecto es fundamental establecer con claridad qué se quiere lograr y cuáles son las fronteras del trabajo. Los objetivos del proyecto deben responder a la pregunta de qué se pretende conseguir al finalizar el desarrollo. Según la guía de Asana sobre la redacción de objetivos de proyecto, estos objetivos representan aquello que se planea lograr al final del proyecto; pueden incluir entregables tangibles o mejoras intangibles y deben ser alcanzables, específicos, medibles y estar acotados en el tiempo[42]. Además, los objetivos de proyecto se diferencian de los objetivos empresariales o los hitos: mientras que los objetivos corporativos son de largo plazo y de alto nivel, los de proyecto son más detallados y se centran en las entregas

específicas[42]. La misma fuente recomienda que los objetivos se definan al inicio del proyecto y sirvan como brújula para todas las actividades[42].

En este proyecto se establecieron como objetivos principales diseñar un entorno de simulación en el que un agente virtual se desplaza y observa su entorno, desarrollar un sistema de detección de anomalías basado en redes neuronales profundas y aprendizaje por refuerzo, y evaluar su rendimiento en diferentes escenarios. Estos objetivos se concretan en tareas más específicas: (i) seleccionar e integrar bibliotecas de simulación y visión artificial, (ii) implementar arquitecturas de redes neuronales para extraer descriptores de imagen y detectar cambios, (iii) entrenar un agente de aprendizaje por refuerzo que optimice su política de exploración, y (iv) validar la solución mediante métricas de detección y rendimiento computacional. El alcance del proyecto se limita al desarrollo en un entorno simulado con recursos de cómputo disponibles y no contempla la transferencia a hardware real, lo que permite concentrarse en la metodología y el análisis de resultados.

3.3. Diseño conceptual

La fase de diseño conceptual constituye la primera etapa del proceso de diseño, en la que se exploran las ideas generales del sistema sin entrar en detalles de implementación. En el ámbito de la construcción y la ingeniería, esta etapa se caracteriza por centrarse en ideas amplias en lugar de especificaciones concretas, e implica actividades como la lluvia de ideas, el esbozo y la evaluación de posibles soluciones para satisfacer los requisitos del proyecto[43]. Entre las características clave del diseño conceptual se encuentran la resolución creativa de problemas, la planificación preliminar (por ejemplo, mediante bocetos o modelos), el análisis de viabilidad para asegurar que la solución es posible dentro de las limitaciones de presupuesto y normativa, y la colaboración con las partes interesadas[43]. Los beneficios de un diseño conceptual bien estructurado incluyen la reducción de riesgos y costes inesperados, la mejora de la eficiencia y la sostenibilidad del proyecto, la facilitación de la comunicación entre los implicados y la provisión de una hoja de ruta clara para las fases posteriores[43].

En nuestro proyecto, el diseño conceptual consistió en identificar los componentes principales del sistema: un entorno de simulación 3D basado en PyBullet, un módulo de percepción que procesa las imágenes captadas por el agente, un módulo de detección de anomalías compuesto por modelos LSTM, y un agente de aprendizaje por refuerzo que controla el desplazamiento del robot. Durante esta etapa se compararon distintas opciones de simulación (PyBullet, Isaac Lab, Unity), se definieron las estructuras de datos para representar el estado del entorno y se diseñó la arquitectura de la red neuronal. También se evaluó la viabilidad de cada componente considerando el tiempo disponible, la curva de aprendizaje de las herramientas

y los requisitos de hardware. Este trabajo inicial permitió detectar riesgos potenciales y tomar decisiones informadas sobre qué tecnologías emplear.

3.4. Diagrama de fases y calendario

Para organizar el proyecto a lo largo de los seis meses previstos, se elaboró un calendario con las principales fases y la duración aproximada de cada una. La literatura sobre gestión de proyectos destaca que la fase de planificación incluye la identificación de requisitos técnicos, el desarrollo de un calendario detallado, la creación de un plan de comunicación y el establecimiento de metas y entregables[42]. Asimismo, durante la planificación es necesario definir el alcance, crear una línea temporal con cada entrega y establecer mecanismos de gestión del riesgo y del cambio[42]. Con base en estas recomendaciones, se dividió el trabajo en las siguientes etapas:

- **Estudio tecnológico y definición de requisitos (2 semanas).** Se investigaron los motores de simulación, las redes neuronales y los algoritmos de aprendizaje por refuerzo disponibles.

- **Diseño conceptual y arquitectura del sistema (4 semanas).** Se definieron los módulos del programa y sus interacciones, se elaboró la arquitectura de la red neuronal y se diseñó la interfaz entre el agente y el entorno.

- **Implementación (8 semanas).** En esta fase se programaron los distintos componentes: el entorno simulado, el módulo de percepción, el detector de anomalías y el agente de aprendizaje por refuerzo. Se utilizaron metodologías de desarrollo incremental para incorporar las funcionalidades de manera progresiva.

- **Entrenamiento y ajuste (4 semanas).** Se entrenó el agente mediante algoritmos de aprendizaje por refuerzo (DQN) y se ajustaron los hiperparámetros de las redes neuronales para mejorar la precisión y la velocidad de detección.

- **Evaluación y validación (3 semanas).** Se realizaron experimentos controlados para medir la tasa de detección de anomalías, el tiempo de respuesta y la estabilidad del sistema. También se analizaron las limitaciones y se propusieron mejoras.

- **Documentación y cierre (3 semanas).** Se elaboraron informes y documentación técnica, se preparó la memoria del proyecto y se realizaron presentaciones finales. Esta fase incluyó reuniones finales con el tutor para recoger comentarios y ajustar la memoria.

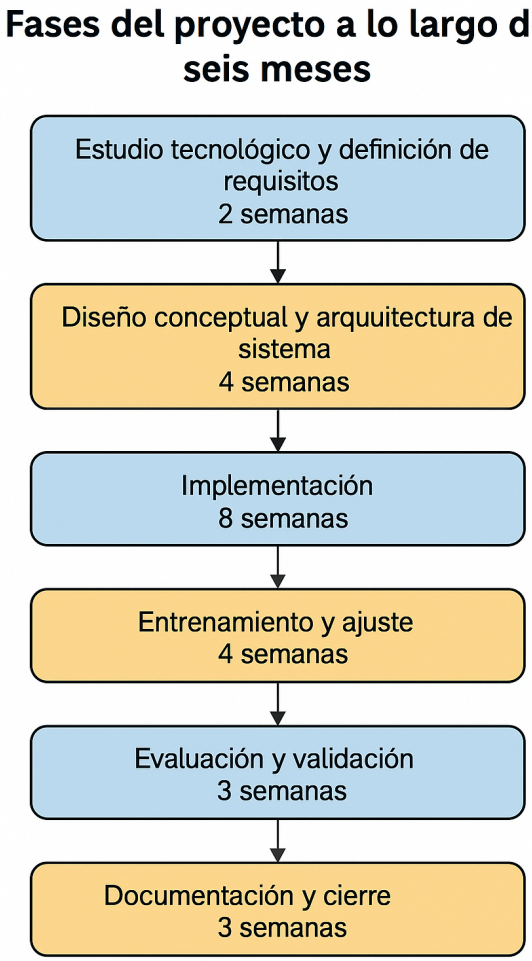

Figura 3.1: Diseño por fases del proyecto.

La distribución anterior suma un total aproximado de veinticuatro semanas, equivalentes a seis meses de trabajo, y permite asegurar que todas las fases reciben la atención adecuada. La planificación se mantuvo flexible, de modo que los tiempos se ajustaron según el progreso real y las dificultades encontradas.

3.5. Costes

Aunque el proyecto se desarrolló íntegramente en un entorno simulado y no requirió hardware especializado ni licencias de software de pago, resulta útil realizar una estimación de los costes para entender el esfuerzo invertido y justificar los recursos empleados. La ingeniería de software describe la

estimación de costes como un proceso sistemático utilizado para prever el esfuerzo (personas–hora o personas–mes), la duración y el coste financiero necesarios para desarrollar, desplegar y mantener un producto software. Esta estimación constituye un paso fundamental de la gestión de proyectos, ya que ayuda a que los interesados tomen decisiones informadas y garantiza que las entregas se realicen a tiempo, dentro del presupuesto y con la calidad deseada[44]. Para generar estimaciones realistas se analizan factores como el tamaño del proyecto, su complejidad, el nivel de experiencia del equipo, las herramientas y tecnologías utilizadas, y los posibles riesgos[44]. Los objetivos de la estimación incluyen optimizar la asignación de recursos, evaluar la viabilidad del proyecto, apoyar la planificación del presupuesto, minimizar riesgos, mejorar la toma de decisiones y establecer calendarios realistas[44].

En nuestro caso, el coste principal corresponde al tiempo de dedicación del desarrollador. Si se considera un único desarrollador trabajando a jornada completa durante seis meses (aproximadamente 960 horas) y se estima una tarifa de 25 eur/hora para un perfil de ingeniero junior, el coste total ascendería a unos 24 000 eur. Este valor incluye las horas de estudio, diseño, implementación, pruebas, documentación y reuniones. Adicionalmente se pueden considerar costes indirectos como el consumo eléctrico del equipo, que es marginal, y el mantenimiento de los servicios en la nube utilizados para el almacenamiento de datos y la ejecución de simulaciones. No se han incluido costes de hardware porque se utilizaron recursos compartidos y un ordenador personal del desarrollador. Tampoco se han contemplado licencias de software, ya que todas las herramientas empleadas son de código abierto.

Para determinar el coste por fase se puede repartir el total en proporción al esfuerzo estimado. Por ejemplo, asignando un 10 % del tiempo al estudio tecnológico, un 20 % al diseño conceptual, un 33 % a la implementación, un 17 % al entrenamiento y ajuste, un 12 % a la evaluación y un 8 % a la documentación, se obtiene el desglose que se muestra en la Tabla 3.1. Estos porcentajes reflejan la prioridad concedida a la implementación y al entrenamiento, que requieren más tiempo y experimentación.

Cuadro 3.1: Estimación de costes por fase del proyecto.

Fase	Porcentaje del tiempo	Coste estimado (eur)
Estudio tecnológico	10 %	2 400
Diseño conceptual	20 %	4 800
Implementación	33 %	7 920
Entrenamiento y ajuste	17 %	4 080
Evaluación y validación	12 %	2 880
Documentación y cierre	8 %	1 920
Total	100 %	**24 000**

Capítulo 4

Metodología

4.1. Objetivo

El objetivo de esta memoria consiste en desarrollar un agente autónomo capaz de desplazarse en un entorno simulado y detectar anomalías visuales. Para alcanzarlo se adopta un enfoque de aprendizaje por refuerzo (*Reinforcement Learning*, RL) combinado con redes neuronales profundas para extraer y modelar características visuales. El sistema se formula como un proceso de decisión de Markov (MDP) en el que el agente observa un estado s_t, toma una acción a_t y recibe una recompensa r_t. El objetivo del agente es maximizar la recompensa acumulada esperada $G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k}$, donde $\gamma \in [0, 1]$ es un factor de descuento que pondera las recompensas futuras [46]. En problemas de control sin modelo como el nuestro se utiliza el aprendizaje Q para aproximar la función de valor de acción $Q^\pi(s, a)$, que estima la recompensa acumulada esperada al ejecutar la acción a en el estado s siguiendo una política π. La regla de actualización de Q–learning, base de nuestro algoritmo, se expresa como

$$Q(s, a) \leftarrow Q(s, a) + \alpha(r + \gamma \max_{a'} Q(s', a') - Q(s, a)),$$

donde α es la tasa de aprendizaje y s' es el siguiente estado [46]. Para entornos con espacios de estado continuos o de alta dimensión, como el nuestro, la tabla Q se sustituye por una aproximación mediante redes neuronales profundas (Deep Q–Networks, DQN) [47]. Además se emplean técnicas como la *experience replay*, que almacena transiciones (s, a, r, s') en una memoria y actualiza la red a partir de mini–lotes aleatorios para romper la correlación temporal [47], y una red objetivo que se actualiza periódicamente para estabilizar el entrenamiento.

La motivación final es obtener una política π^* que permita desplazar el robot virtual de forma segura y eficiente, dirigiéndolo hacia zonas del entorno donde la red de detección de anomalías experimente altos errores de

reconstrucción (indicando la presencia de objetos o configuraciones inesperadas). Este objetivo se persigue de manera supervisada en la fase de diseño conceptual (definición de tareas y entornos) y mediante aprendizaje no supervisado en la detección de anomalías, integrando ambos enfoques en un marco de RL.

4.2. Diseño del sistema

El sistema se compone de varios módulos que interactúan en tiempo real. La Figura 4.2 resume la arquitectura general: el entorno simulado genera imágenes RGB de la escena, que son procesadas por un módulo de percepción basado en ResNet50 para extraer *embeddings* de 2 048 dimensiones; a continuación, un módulo de detección de anomalías formado por un modelo LSTM evalúa el error de reconstrucción y produce una recompensa; finalmente, un agente de RL (DQN) decide la acción que debe ejecutar el robot basándose en el estado actual y la recompensa recibida. A lo largo de esta sección se describen cada uno de estos componentes en detalle.

4.2.1. Entorno de simulación

Para reproducir el comportamiento del robot sin las dificultades asociadas a un prototipo físico se optó por un entorno de simulación 3D basado en la librería PyBullet. PyBullet permite simular dinámica de cuerpos rígidos y colisiones en tiempo real, así como cargar modelos de robots en formato URDF. El entorno desarrollado en el fichero `anomaly_env_premium.py` genera dos escenarios distintos: uno con banderas en distintas posiciones y otro con objetos desconocidos dispersos. En ambos casos la escena contiene un suelo, paredes y obstáculos que limitan el desplazamiento del robot. Al iniciar un episodio se llama al método `reset`, que limpia la escena, carga el modelo del robot y posiciona las banderas u objetos dentro del mapa. El pseudocódigo simplificado del método `reset` se muestra a continuación:

Listing 4.1: Inicializacióon del entorno en el método `reset`.

```
def reset(self):
    self._setup_world(self.cap)
    self.embeddings_buffer.clear()
    self.anom_buffer.clear()
    self.current_step = 0

    img = self._render_image()
    emb = self._get_embedding(img)
    dist = self.get_ultrasonic_distance()   # <- sin argumentos
    obs = np.concatenate([emb, [dist]])
    return obs.astype(np.float32)
```

La función `_render_image` emplea la cámara virtual de PyBullet para capturar una imagen RGB de la escena. La cámara se sitúa por encima y

detrás del robot, apuntando hacia su dirección de avance. La imagen de 224×224 píxeles se normaliza y se pasa al módulo de percepción. Las acciones que puede ejecutar el agente son discretas: avanzar, girar a la izquierda y girar a la derecha. Cada llamada a `step(action)` hace avanzar la simulación un paso, actualiza la posición del robot según la acción seleccionada y devuelve la nueva observación, la recompensa y un indicador `done` que señala el final del episodio.

4.2.2. Control del robot en simulación

El robot simulado es un modelo móvil dotado de dos motores que impulsan sus ruedas laterales. Para simplificar el control, se define un espacio de acción discreto: {0: avanzar, 1: girar a la izquierda, 2: girar a la derecha}. El método interno `_move_robot` traduce estas acciones en velocidades lineales y angulares que se aplican a las articulaciones del robot. Durante cada paso de simulación, PyBullet integra las ecuaciones de movimiento y actualiza la posición y orientación del robot en el entorno. El siguiente fragmento de código muestra la implementacion de la funcion move_robot:

Listing 4.2: Implementación del método move_robot para el `control` del robot.

```
def _move_robot(self, action):
        pos, ori = p.getBasePositionAndOrientation(self.robot)
        euler = p.getEulerFromQuaternion(ori)
        yaw = euler[2]
        R = p.getMatrixFromQuaternion(ori)
        fwd = np.array([R[1], R[4], R[7]], dtype=float)

        step_len = 0.3
        min_clearance = 0.7
        safety = 0.1

        if action == 0:  # AVANZAR
            # Obtenemos la distancia del sensor
            dist = self.get_ultrasonic_distance(collision_extra_rays=
                True, max_distance=10.0)
            # Si hay espacio suficiente, avanza; si no, te quedas en
                sitio
            if dist > (min_clearance + safety):
                new_pos = np.array(pos) + fwd * step_len
            else:
                new_pos = np.array(pos)  # bloqueado: no avances
        elif action == 1:  # GIRAR IZQ
            yaw += math.radians(30)
            new_pos = np.array(pos)
        else:              # GIRAR DER
            yaw -= math.radians(30)
            new_pos = np.array(pos)

        # Mantenerse siempre dentro de la sala
        room_size = 10
        wall_thickness = 0.1
        margin = 0.2
```

```
        xmin = -room_size/2 + wall_thickness/2 + margin
        xmax =  room_size/2 - wall_thickness/2 - margin
        ymin = -room_size/2 + wall_thickness/2 + margin
        ymax =  room_size/2 - wall_thickness/2 - margin
        new_pos[0] = float(np.clip(new_pos[0], xmin, xmax))
        new_pos[1] = float(np.clip(new_pos[1], ymin, ymax))

        new_ori = p.getQuaternionFromEuler([0, 0, yaw])
        p.resetBasePositionAndOrientation(self.robot, new_pos.tolist()
            , new_ori)
```

4.2.3. Sistema reactivo para evitar colisiones

Una parte esencial del control es evitar colisiones con las paredes y otros objetos. Para ello se emplea un sensor ultrasónico virtual implementado mediante `rayTest`. Este método lanza múltiples rayos desde la parte frontal del robot y calcula la distancia al primer obstáculo. El método `get_ultrasonic_distance` retorna la mínima distancia medida; si ésta cae por debajo de un umbral se considera que hay una colisión inminente y el episodio se termina con una recompensa negativa. El siguiente fragmento de código muestra la implementación del sensor:

Listing 4.3: Sensor ultrasónico virtual basado en rayos.

```
def get_ultrasonic_distance(self, collision_extra_rays=False,
    max_distance=10.0):
        pos, ori = p.getBasePositionAndOrientation(self.robot)
        rot = p.getMatrixFromQuaternion(ori)
        fwd = [rot[1], rot[4], rot[7]]
        right = [rot[0], rot[3], rot[6]]

        start_center = [pos[0] + fwd[0]*0.2, pos[1] + fwd[1]*0.2, pos
            [2] + 0.5]
        end_center   = [start_center[0] + fwd[0]*max_distance,
                        start_center[1] + fwd[1]*max_distance,
                        start_center[2] + fwd[2]*max_distance]
        result_center = p.rayTest(start_center, end_center)[0]
        distances = [result_center[2]*max_distance if result_center[0]
             != -1 else max_distance]

        if collision_extra_rays:
            shoulder = 0.3
            # izquierdo
            sL = [start_center[0]-right[0]*shoulder, start_center[1]-
                right[1]*shoulder, start_center[2]]
            eL = [sL[0]+fwd[0]*max_distance, sL[1]+fwd[1]*max_distance
                , sL[2]+fwd[2]*max_distance]
            rL = p.rayTest(sL, eL)[0]
            distances.append(rL[2]*max_distance if rL[0] != -1 else
                max_distance)
            # derecho
            sR = [start_center[0]+right[0]*shoulder, start_center[1]+
                right[1]*shoulder, start_center[2]]
            eR = [sR[0]+fwd[0]*max_distance, sR[1]+fwd[1]*max_distance
                , sR[2]+fwd[2]*max_distance]
            rR = p.rayTest(sR, eR)[0]
```

```
25              distances.append(rR[2]*max_distance if rR[0] != -1 else
                    max_distance)
26
27          return min(distances)
```

La figura 4.2 muestra el funcionamiento del sensor ultrasonico.

Además del sensor ultrasónico, el controlador almacena un historial de acciones recientes. Si el agente repite demasiados giros en la misma dirección, se aplica una penalización para desalentar comportamientos de giro en bucle. Este mecanismo ayuda al agente a explorar el espacio de manera más eficiente.

El subsistema de evitación de colisiones actúa como un controlador reactivo. Cada vez que se ejecuta una acción, se comprueba la distancia devuelta por el sensor ultrasónico. Si la distancia es inferior a 0,7 metros, se interpreta que el robot va a chocar con un obstáculo. En ese momento el entorno devuelve un gran castigo negativo (por ejemplo, -0.03).

4.2.4. Componente de planificación y aprendizaje por refuerzo

El núcleo de la planificación reside en un agente de aprendizaje por refuerzo que decide en cada instante la acción que debe ejecutar el robot. El agente se entrena con el algoritmo Deep Q – Learning (DQN) para maximizar la recompensa basada en el error de detección de anomalías.

Estructura del agente. El agente utiliza una red neuronal de política (`MlpPolicy`) con varias capas totalmente conectadas para aproximar la función Q. Esta red toma como entrada el vector de observación de 2 049 dimensiones (2 048 del *embedding* visual) y devuelve un valor Q por cada acción. La pérdida del agente se calcula como el error cuadrático entre la predicción actual y el objetivo TD:

$$L(\theta) = (r + \gamma \max_{a'} Q_{\theta^-}(s', a') - Q_\theta(s, a))^2,$$

donde θ son los parámetros de la red principal y θ^- son los parámetros de la red objetivo [47]. Cada cierto número de pasos se actualizan θ^- con los valores de θ para estabilizar el entrenamiento.

Entrenamiento. En el fichero `evaluate_dqn.py` se define el bucle de entrenamiento y evaluación del agente. Se crea una instancia de `DQN` a partir de la librería `stable_baselines3` con hiperparámetros específicos (tamaño del buffer de *replay*, tasa de aprendizaje, tamaño de lote, etc.). El agente se entrena durante un número determinado de pasos recolectando transiciones en el entorno y actualizando la red cada vez que se dispone de un lote de experiencias. El fragmento siguiente muestra la llamada principal al entrenamiento:

Listing 4.4: Entrenamiento del agente DQN.

```
# Crear entorno y agente
while not done:
    if step_count % 20 == 0:
        model.learn(total_timesteps=16, reset_num_timesteps=False,
            log_interval=10)
    action, _ = model.predict(obs, deterministic=True)
    obs, reward, done, _ = env.step(action)
    total_reward += reward
    step_count += 1
    time.sleep(1/100)
    print(f"Paso {step_count}: Accion {action} | Recompensa {reward:.4
        f}")
```

Estrategias de exploración y explotación. Para equilibrar exploración y explotación se emplea una política $\epsilon-$*greedy*, en la que con probabilidad ϵ se selecciona una acción aleatoria y con probabilidad $1-\epsilon$ se elige la acción con mayor valor Q. Durante el entrenamiento, ϵ disminuye linealmente desde 1 hasta 0,05 para favorecer la exploración inicialmente y la explotación de la política aprendida al final [47].

Aprendizaje por refuerzo no supervisado. Se investigó el aprendizaje por refuerzo no supervisado (URL) que prescinde de recompensas externas y utiliza señales intrínsecas derivadas de la curiosidad del agente [49]. En nuestro problema el error de detección de anomalías actúa como recompensa intrínseca, de manera que el agente explora zonas con alto error sin necesidad de una señal de recompensa manual.

4.2.5. Arquitectura de la red LSTM para detección de anomalías

El módulo de detección de anomalías utiliza una red LSTM para modelar la dinámica temporal de las secuencias de *embeddings* visuales. La idea es entrenar una LSTM a partir de secuencias de cuatro *embeddings* consecutivos $\{e_{t-2}, e_{t-1}, e_t, e_{t+1}\}$ de manera que aprenda a predecir el *embedding* central e_t. Si el error de reconstrucción al comparar la predicción $\hat{e}_t$ con el valor real es pequeño, la secuencia se considera normal; un error grande indica la presencia de un patrón no visto (anomalía). Para ello se diseña un entorno especifico de entrenamiento de forma manual en el que el agente se mueve a traves de las teclas “WAD”, donde la “w” crea el movimiento hacia delante mientras que la “A” y la “D” lo hacen hacia la izquierda y derecha consecutivamente. Este entorno contiene las mismas especificaciones que el mundo por el que el robot final se moverá de forma autonoma por lo que podemos realizar un entrenamiento realista. En este entorno nuestro robot captura un frame por movimiento y los almacena en una carpeta “frame”.

Listing 4.5: Preentreno manual de la red LSTM para el conocimiento del mundo

```
while True:
    p.stepSimulation()
    move_robot_with_keys(robot)

    dist = get_ultrasonic_distance(robot)
    pos, _ = p.getBasePositionAndOrientation(robot)
    text_pos = [pos[0], pos[1], pos[2] + 1.5]
    p.addUserDebugText(f"Distancia: {dist:.2f} m", text_pos,
        replaceItemUniqueId=text_id, textColorRGB=[1,1,0], textSize
        =1.5)

    frame_img = render_first_person_view(robot)
    cv2.imshow("Vista Robot", frame_img)
    cv2.waitKey(1)

    # Guardar frame para entrenamiento
    if is_key_pressed() != None:
        cv2.imwrite(f"frames{cap_str}/frame_{frame_count:05d}.png",
            frame_img)
        frame_count += 1

    time.sleep(1/60)
```

La cantidad de frames capturados debe ser la necesaria para conocer el mundo en el que el agente se encuentra, en nuestro caso para un mundo de 10x10 necesitamos alrededor de 300/400 imágenes.

Una vez guardados los frames se procede a calcular los *embeddings* de cada uno de ellos a través del script `calcular_embeddings_entrenamiento.py`

Listing 4.6: Calculo de embeddings para cada frame

```
# Cargar modelo ResNet50 preentrenado y quitar la  ltima  capa
modelo = models.resnet50(pretrained=True)
modelo.eval()
modelo = torch.nn.Sequential(*list(modelo.children())[:-1])  # Elimina
     la capa final (fc)

# Transformaciones de imagen requeridas por ResNet50
transformaciones = transforms.Compose([
    transforms.Resize(256),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406],
                         std=[0.229, 0.224, 0.225]),
])

# Obtener imagenes
imagenes = [f for f in os.listdir(carpeta_imagenes) if f.lower().
    endswith(('.jpg', '.jpeg', '.png'))]

# Procesar cada imagen
for nombre_archivo in imagenes:
    ruta_imagen = os.path.join(carpeta_imagenes, nombre_archivo)
    imagen = Image.open(ruta_imagen).convert('RGB')
    imagen_tensor = transformaciones(imagen).unsqueeze(0)

```

```
    with torch.no_grad():
        vector = modelo(imagen_tensor)
        vector = vector.squeeze().numpy()

    nombre_vector = os.path.splitext(nombre_archivo)[0] + ".npy"
    ruta_salida = os.path.join(carpeta_salida, nombre_vector)
    np.save(ruta_salida, vector)

    print(f" Vector guardado: {ruta_salida}")
```

El script `entrenar_lstm.py` carga los vectores de entrenamiento previamente generados, construye una red secuencial con dos capas LSTM y una capa densa final de 2 048 neuronas y utiliza el error cuadrático medio (*MSE*) como función de pérdida. La red se entrena durante 50 épocas y se guarda el modelo con mejor resultado. El código simplificado es el siguiente:

Listing 4.7: Entrenamiento de la red LSTM para detección de anomalías.

```
# Cargar secuencias de embeddings (entrada) y embeddings centrales (
    salida)
for i in range(len(archivos) - 4):
    ventana = archivos[i:i+5]

    vectores = [np.load(os.path.join(input_dir, archivo)) for archivo
        in ventana]

    input_vectores = [vectores[0], vectores[1], vectores[3], vectores
        [4]]
    X.append(np.stack(input_vectores))  # (4, 2048)
    y.append(vectores[2])  # (2048,)

X = np.array(X)  # (num_muestras, 4, 2048)
y = np.array(y)  # (num_muestras, 2048)

print(f"Datos preparados: X={X.shape}, y={y.shape}")

# Definir el modelo
model = Sequential([
    Masking(mask_value=0.0, input_shape=(4, 2048)),
    LSTM(512, return_sequences=True),
    LSTM(512, return_sequences=False),
    Dense(2048)
])

# Compilar y entrenar
model.compile(optimizer='adam', loss='mse')
model.fit(x, y, epochs=50, batch_size=32,
          validation_split=0.1,
          callbacks=[ModelCheckpoint('modelo_lstm.h5',
                     save_best_only=True, monitor='val_loss')])
```

A partir de aquí ya tenemos el modelo LSTM entrenado para detectar anomalías dentro de la simulación (siempre y cuando no se altere el mundo). Durante la simulación, cada vez que se obtiene un nuevo *embedding* se actualiza un *buffer* de los últimos cinco vectores. El método `_eval_anomaly` extrae una secuencia de cuatro vectores omitiendo el central y calcula la predicción de la LSTM. La recompensa se deriva del error de reconstrucción

medido como la media del cuadrado de las diferencias entre $\hat{e}_t$ y e_t. Si el error supera un umbral predefinido se considera que se ha detectado una anomalía y se almacena la secuencia para refinar la LSTM con reentrenamiento en línea. Este mecanismo de *online retraining* permite adaptar el modelo a nuevas anomalías detectadas y, por tanto, mejorar su sensibilidad con el tiempo. La siguiente función implementa este mecanismo:

Listing 4.8: Evaluación de anomalías y reentrenamiento en línea.

```
def _eval_anomaly(self, embedding,img):
        self.embeddings_buffer.append(embedding)
        self.img_buffer.append(img)
        if len(self.embeddings_buffer) > 5:
            self.embeddings_buffer.pop(0)
            self.img_buffer.pop(0)
        if len(self.embeddings_buffer) < 5:
            return None

        x_seq = np.array([
            self.embeddings_buffer[0],
            self.embeddings_buffer[1],
            self.embeddings_buffer[3],
            self.embeddings_buffer[4]
        ])
        x_seq = x_seq.reshape(1, 4, -1)
        y_true = self.embeddings_buffer[2].reshape(1, -1)
        y_pred = self.lstm_model.predict(x_seq, verbose=0)

        error = np.mean((y_true - y_pred) ** 2)
        print(error)

        if error > 0.1:
            self._reentrenar_lstm_online(x_seq, y_true)

        return error
```

4.2.6. Flujo de operación

El funcionamiento completo del sistema puede describirse mediante la siguiente secuencia de pasos:

1. **Inicialización**. Se capturan frames de forma manual del mundo que necesitamos simular; se calculan los embeddings de cada frame; se entrena la red LSTM para que conozca dicho mundo; se instancian el entorno, el agente DQN y los buffers de memoria.

2. **Percepción**. En cada iteración el entorno devuelve una imagen RGB de 224×224 píxeles. La imagen se preprocesa (normalización, redimensionado) y se calcula su *embedding* e_t utilizando ResNet50 sin la capa fully connected final.

3. **Detección de anomalías**. El *embedding* e_t se añade al buffer de

la LSTM y se calcula el error de reconstrucción. Si se detecta una anomalía, se almacena la secuencia para reentrenar el modelo.

4. **Cálculo de la recompensa**. La recompensa r_t se compone de una parte positiva proporcional al error de anomalía (a mayor error, mayor recompensa) y penalizaciones por distancia al obstáculo y por repeticiones de acción. De esta forma el agente se ve incentivado a dirigirse hacia zonas con alto error y a evitar colisiones.

5. **Selección de acción**. El agente DQN recibe como estado el vector $[e_t, d_t]$ y elige una acción a_t mediante una política $\epsilon - greedy$. La acción se ejecuta en el entorno y actualiza la posición del robot.

6. **Reentrenamiento de la LSTM**. Cuando se acumulan suficientes anomalías consecutivas, se reentrena el modelo LSTM con los nuevos ejemplos para mejorar su sensibilidad. El reentrenamiento se realiza en segundo plano para no interrumpir la simulación.

Este flujo se repite durante miles de episodios hasta que el agente converge a una política que maximiza la recompensa, es decir, que dirige al robot hacia recibir una recompensa (error) relativamente buena.

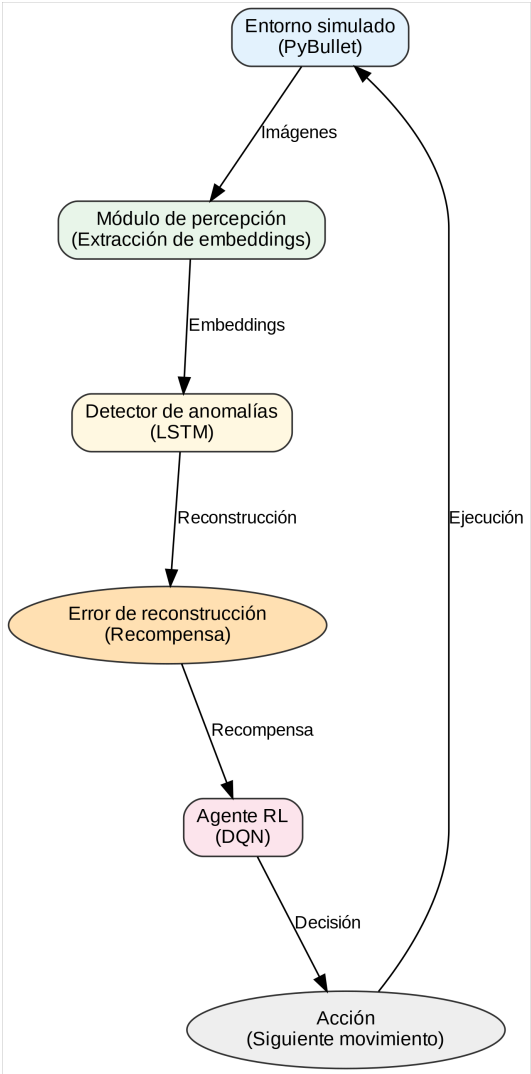

Figura 4.1: Arquitectura del programa.

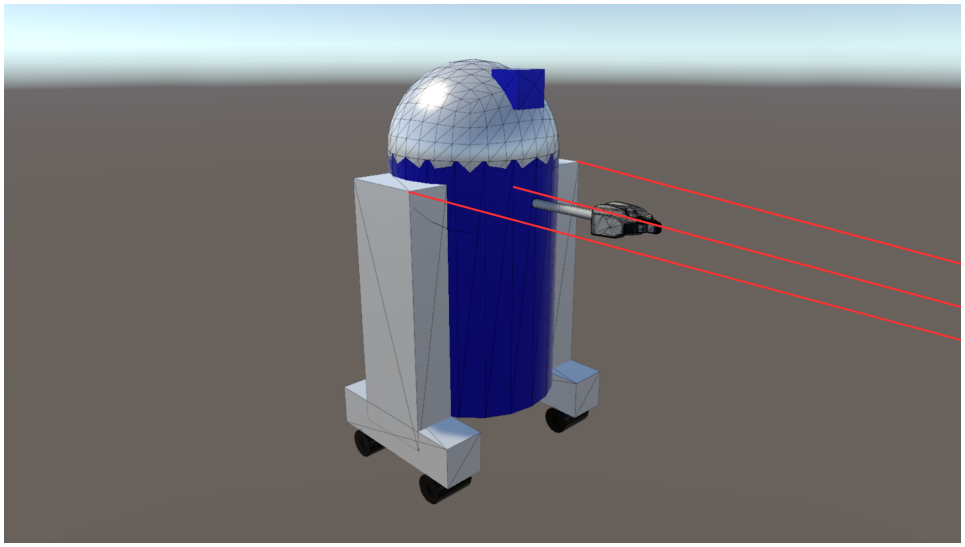

Figura 4.2: Descripcion del funcionamiento real del sensor ultrasonico simulado basado en rayos.

Capítulo 5

Evaluación

En este capítulo se describen las dos formas de evaluación implementadas en el proyecto para validar tanto el comportamiento del agente de aprendizaje por refuerzo como la precisión del módulo de detección de anomalías. La primera sección expone la *evaluación automática* realizada tras cada episodio de entrenamiento de *Deep Q–Learning* (DQN), donde se analiza el nivel de atención calculado por la red LSTM y se generan estadísticas y gráficos automáticos. La segunda sección presenta la *evaluación manual* del sistema de detección de anomalías, en la que un operador controla al robot mediante un teclado y observa en tiempo real el error entre las imágenes reales y las predicciones del modelo.

5.1. Evaluación del agente DQN

La evaluación automática se lleva a cabo al finalizar cada episodio ejecutado con el agente DQN. Para ello se registran dos secuencias: la trayectoria del robot (x_t, y_t) y el nivel de atención e_t que devuelve el módulo de detección de anomalías. El nivel de atención se obtiene a partir del error de reconstrucción producido por el modelo LSTM: si las características visuales generadas por el autoencoder difieren de la predicción, se incrementa el valor de e_t. El código de evaluación calcula la media μ y la desviación estándar σ de la secuencia e, y a partir de ellas el *z–score* $z_t = (e_t - \mu)/\sigma$. Los puntos con $|z_t| > 3$ se consideran anómalos según la regla de las tres sigmas, es decir, están más allá de tres desviaciones estándar de la media[51]. La proporción de valores con $|z_t| > 3$ proporciona una estimación del porcentaje de pasos en los que el agente percibe una anomalía[51].

Además de esta métrica, se calcula la recompensa acumulada $R = \sum_t r_t$ y la recompensa media, entendidas como medidas clásicas de rendimiento en aprendizaje por refuerzo. Estas recompensas permiten comparar la calidad de diferentes políticas y observar si el agente mejora su comportamiento a lo largo del entrenamiento. Para analizar la estabilidad del aprendizaje tam-

bién se considera la eficiencia muestral, es decir, el número de interacciones necesarias para alcanzar un umbral de rendimiento.

5.1.1. Curva de atención y recorrido coloreado

Una vez calculados los valores anteriores, el programa genera dos gráficos. La *curva de atención* muestra la evolución de e_t frente al número de pasos; picos en esta curva indican momentos en los que el módulo LSTM detecta desviaciones significativas del patrón de entrenamiento. El segundo gráfico es una vista cenital del recorrido (x_t, y_t) en la que cada punto se colorea según el valor de e_t. Este tipo de representación se denomina *mapa de calor*: se visualiza una variable numérica mediante una gradación de colores que facilita la identificación de patrones, tendencias y anomalías en datos bidimensionales. Para generar el mapa se utiliza una escala de colores que va de tonos fríos (errores bajos) a tonos cálidos (errores altos), lo que permite localizar de forma intuitiva las zonas del entorno donde el robot percibe mayor desviación[52]. La Figura 5.1 muestra un ejemplo del mapa generado.

5.2. Evaluación del modelo de detección de anomalías LSTM

Además de la evaluación automática con el agente entrenado, se ha desarrollado un procedimiento manual para inspeccionar el comportamiento del módulo LSTM de manera interactiva. Este procedimiento se implementa en el script `simulated_extended_eval.py`, que permite mover al robot con las teclas `WAD` (adelante, izquierda, derecha). Por cada desplazamiento se captura un fotograma de la cámara y se genera su embedding mediante la red ResNet50. A continuación el modelo LSTM predice el embedding esperado para esa posición y se calcula el error de reconstrucción entre el embedding real y el predicho. Este error se muestra en tiempo real en la interfaz gráfica, lo que permite al usuario comprobar visualmente la sensibilidad del modelo a los cambios del entorno. Cuando el robot se acerca a un objeto no visto durante el entrenamiento, el error aumenta de manera notable; por el contrario, cuando se mueve por zonas conocidas, el error se mantiene bajo.

La evaluación manual complementa a la evaluación automática porque proporciona una validación cualitativa del sistema de detección de anomalías. Permite observar comportamientos específicos, ajustar umbrales y validar que la señal de atención se corresponde con percepciones humanas de anomalía. De este modo se verifican hipótesis sobre la robustez del modelo ante variaciones del entorno y se identifican posibles fallos que podrían pasar inadvertidos en la evaluación estadística.

Aquí se muestra una implementación simplificada del sistema de evaluación manual del modelo de detección de anomalías:

Listing 5.1: Evaluación de detección de anomalias manual

```
if is_key_pressed() is not None:
        image_route = f"frames_eval/FRAME-EVAL{frame_count:05d}.png"
        image_name = f"FRAME-EVAL{frame_count:05d}.png"
        cv2.imwrite(image_route, frame_img)
        frame_count += 1

        if frame_count > 4:
            embedding_real = calcular_embedding(image_route,
                image_name)
            error = evaluar_si_hay_anomalia(embedding_real)

            if error is not None and frame_imgs[2] is not None:
                errores_mse.append(error)
                linea_error.set_data(range(len(errores_mse)),
                    errores_mse)
                ax.set_xlim(0, max(50, len(errores_mse)))
                ax.set_ylim(0, max(0.1, max(errores_mse) * 1.2))
                fig.canvas.draw()
                fig.canvas.flush_events()
```

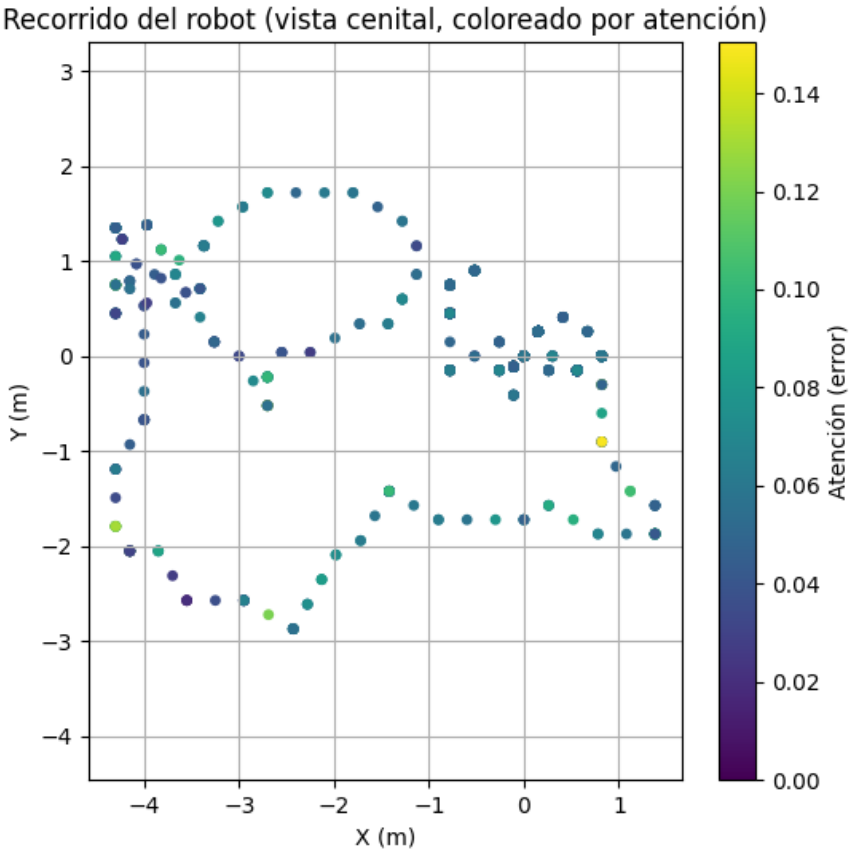

Figura 5.1: Visualizaciones generadas en la evaluación automática: se representan la evolución temporal de la atención y el mapa de calor del recorrido.

Capítulo 6

Resultados

En este capítulo se presentan y discuten los resultados obtenidos tras aplicar las dos metodologías de evaluación descritas en el capítulo anterior. En primer lugar se analiza el comportamiento del agente de aprendizaje por refuerzo (DQN) cuando se evalúa de forma automática en el entorno simulado, tomando como recompensa el error de reconstrucción devuelto por el detector de anomalías. A continuación se describe la evaluación manual del modelo de detección de anomalías, en la que un operador humano recorre el entorno con un teclado y visualiza en tiempo real la respuesta del modelo LSTM ante cambios en el entorno. Para cada caso se comentan los resultados cuantitativos y cualitativos obtenidos, y se ofrecen recomendaciones para mejorar el sistema.

6.1. Evaluación del agente DQN

El agente DQN se entrenó en el entorno simulado descrito en capítulos anteriores, empleando como señal de recompensa el error de reconstrucción del detector de anomalías. Durante la fase de entrenamiento se aplicó un esquema de aprendizaje en línea: cuando el error de reconstrucción superaba un umbral durante varios pasos consecutivos se añadían las secuencias anómalas a un conjunto de entrenamiento y se reentrenaba el modelo LSTM de manera incremental. Este enfoque está motivado por los trabajos recientes en detección de anomalías que señalan que los modelos entrenados en lotes de datos pueden quedar obsoletos cuando cambian las propiedades estadísticas del entorno. El fenómeno de *concept drift* produce que los modelos históricos dejen de ser válidos, por lo que es preferible emplear aprendizaje en línea para adaptar el modelo a las nuevas distribuciones de datos[53]. La incorporación de un reentrenamiento incremental permitió que nuestro detector se ajustara a las anomalías detectadas durante la exploración, reduciendo progresivamente el error medio.

Para evaluar objetivamente la política aprendida se registraron varias

métricas: la recompensa acumulada por episodio, la evolución del error de reconstrucción, el porcentaje de pasos marcados como anómalos y la distribución de trayectorias. La Figura 6.1 muestra un ejemplo de la evolución del error durante un episodio: se observa que, después de un periodo inicial de exploración, el error disminuye y se estabiliza a medida que el agente incrementa la frecuencia de reentrenamiento y aprende a evitar configuraciones que producen alto error. Esta tendencia es coherente con la literatura sobre aprendizaje en línea, en la que los algoritmos actualizan sus parámetros tras cada muestra para adaptarse a los cambios en la distribución y mantener la precisión[53].

No obstante, el análisis de las trayectorias revela que el agente realiza una exploración limitada del entorno. A pesar de reducir el error, el DQN tiende a explotar las políticas conocidas y se desplaza repetidamente por las mismas zonas, evitando explorar nuevas áreas. Este comportamiento es un ejemplo del dilema exploración–explotación descrito en la literatura de aprendizaje por refuerzo: cualquier agente debe equilibrar el beneficio inmediato de explotar las acciones conocidas con la necesidad de explorar opciones nuevas que puedan conducir a recompensas mayores[54]. Diversos estudios han mostrado que los organismos y algoritmos resuelven este dilema mediante dos estrategias complementarias: la búsqueda dirigida de información y la aleatorización de decisiones[54]. Nuestro agente utiliza una política *epsilon–greedy* que introduce un porcentaje fijo de acciones aleatorias, pero ese valor podría ser demasiado bajo para incentivar una exploración más amplia. La Figura 6.2 representa el recorrido del robot durante un episodio utilizando un mapa de calor: los colores más cálidos indican zonas donde el error de reconstrucción (nivel de atención) es alto. Se aprecia que el robot recorre repetidamente un subconjunto del escenario, lo que sugiere que un ajuste de los parámetros de exploración (por ejemplo, un valor de ε adaptativo o técnicas como Thompson sampling) podría mejorar la cobertura.

Los resultados cuantitativos muestran que la recompensa acumulada aumenta con el número de episodios y que el porcentaje de falsos positivos (anomalías detectadas en ausencia de cambios reales) se mantiene bajo. Sin embargo, la ausencia de exploración extensa limita la capacidad del agente para descubrir todas las anomalías posibles. De acuerdo con estudios recientes, combinar exploración dirigida y aleatoria puede mejorar la eficiencia de aprendizaje y evitar quedar atrapado en políticas subóptimas[54].

6.2. Evaluación del modelo de detección de anomalías LSTM

Además de la evaluación automática integrada en el agente DQN, se evaluó el modelo de detección de anomalías de forma independiente mediante el script `simulated_extended_eval.py`. En este procedimiento, un usuario

controla manualmente al robot utilizando teclas `WASD` para moverse por el entorno. En cada paso se captura un fotograma de la cámara y se calcula el error de reconstrucción entre la imagen real y la imagen predicha por el modelo LSTM. Este error se muestra en tiempo real en la interfaz gráfica, permitiendo observar la respuesta del detector frente a distintos estímulos. La evaluación manual permite valorar el detector en ausencia de aprendizaje por refuerzo y verificar su capacidad para generalizar a nuevas escenas.

Los experimentos se llevaron a cabo en dos tipos de escenarios. En el primer escenario se colocaron objetos de distintas formas y colores; el modelo LSTM se preentrenó con un conjunto de imágenes que incluía ciertas configuraciones de estos objetos y, durante la evaluación, se modificaron la forma o el color de algunos elementos. El error de reconstrucción se disparó al aproximarse a un objeto no conocido, lo que indica que el modelo detecta correctamente la anomalía 6.3. Con objetos de formas diferenciadas el error vuelve a valores bajos cuando el robot regresa a zonas conocidas, demostrando una buena capacidad de generalización. En el segundo escenario se utilizaron muros decorados con banderas de distintos países. El modelo se entrenó con un conjunto de banderas específicas y, en la evaluación, se sustituyó alguna bandera por la de otro país. En este caso la detección resultó más difícil: algunos cambios de color fueron demasiado sutiles para generar un error elevado, y en ocasiones el sistema produjo falsos positivos, aumentando el error sin que existiera una anomalía real 6.4. Estas observaciones concuerdan con los trabajos que señalan que los modelos de detección basados en reconstrucción son más sensibles a cambios estructurales o de textura que a cambios cromáticos leves[53].

Una ventaja del enfoque de detección basado en LSTM es su capacidad de adaptación al cambio mediante aprendizaje en línea. Como se explicó anteriormente, los modelos entrenados en lotes pueden perder precisión cuando cambian las propiedades estadísticas de los datos; el aprendizaje incremental permite ajustarse a estas variaciones[53]. No obstante, en escenarios donde las anomalías son difíciles de distinguir, como las banderas, es necesario ampliar el conjunto de entrenamiento para abarcar una mayor variedad de patrones. En nuestros experimentos, un preentrenamiento con 300–400 imágenes fue suficiente para obtener una detección estable; sin embargo, aumentar el número de imágenes a varios miles mejoraría la representación de la distribución normal y reduciría los falsos positivos.

6.3. Discusión de resultados

Los dos métodos de evaluación proporcionan una visión complementaria del comportamiento del sistema. La evaluación automática demuestra que el agente DQN es capaz de aprender políticas que reducen el error de reconstrucción gracias al reentrenamiento en línea, aunque la falta de exploración

limita su capacidad para identificar todas las anomalías. Este resultado destaca la importancia de diseñar mecanismos de exploración adecuados y de considerar estrategias de aprendizaje activo que combinen exploración dirigida y aleatoria[54].

La evaluación manual, por su parte, confirma la capacidad del modelo LSTM para detectar anomalías en tiempo real, incluso cuando el robot es controlado por un humano. La respuesta del modelo es robusta ante cambios de forma y textura, pero presenta más dificultades ante alteraciones puramente cromáticas. A pesar de la presencia de falsos positivos, el sistema muestra un comportamiento estable y puede servir como base para un sistema de monitorización autónomo. Ampliar el preentrenamiento con un mayor número de imágenes y aplicar técnicas de regularización podrían mejorar la precisión y reducir los falsos positivos.

En general, los resultados sugieren que combinar aprendizaje por refuerzo con detección de anomalías basada en predicción es una estrategia viable para entornos simulados. El uso de aprendizaje en línea permite adaptar el sistema a cambios imprevistos, mientras que la evaluación manual ofrece un mecanismo para validar el detector de manera independiente.

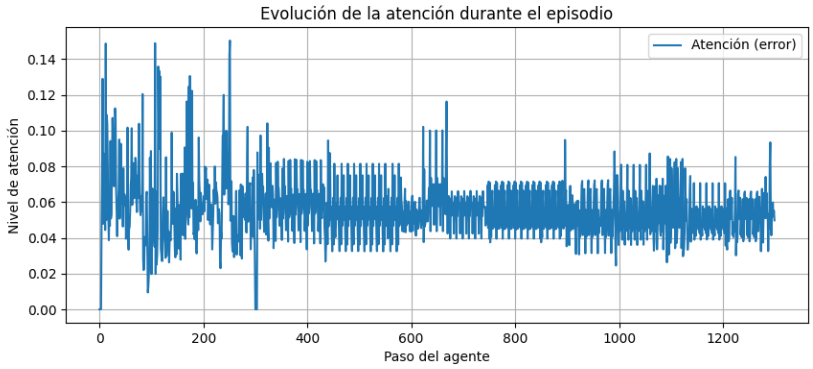

Figura 6.1: Evolución del nivel de atención (error de reconstrucción) durante un episodio de evaluación automática del agente DQN. Se aprecia una reducción gradual del error gracias al reentrenamiento en línea.

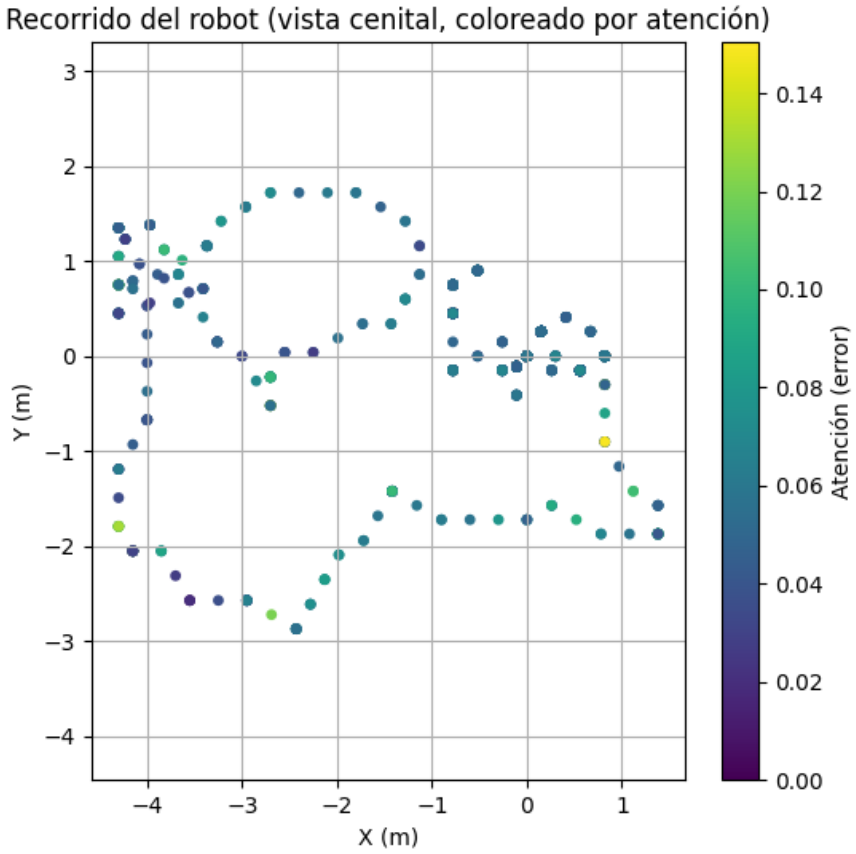

Figura 6.2: Mapa de calor del recorrido del agente DQN en un episodio de evaluación. Los puntos representan las posiciones (x, y) del robot, coloreadas según el error de reconstrucción. La concentración de puntos en ciertas zonas refleja la falta de exploración.

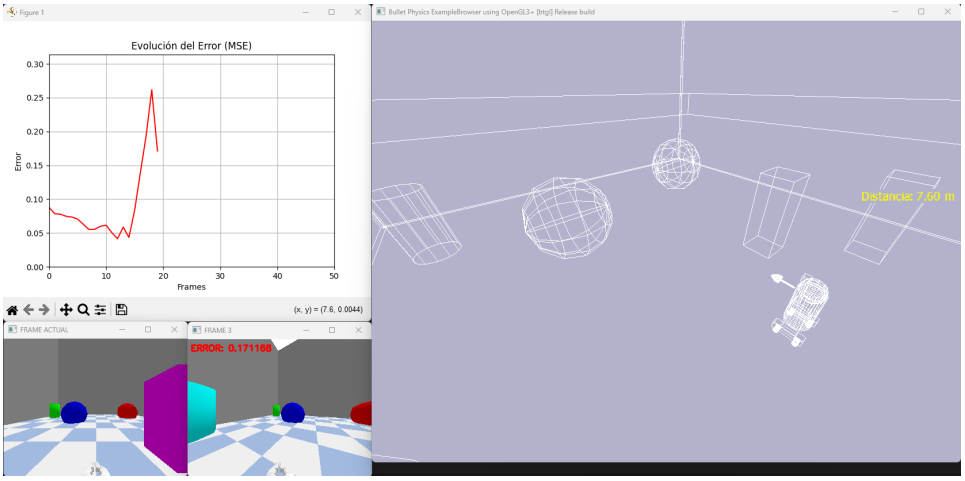

Figura 6.3: Resultados de la evaluación manual del modelo LSTM en un escenario con objetos de distintas formas y colores. En este caso se desconoce el objeto que es una esfera de color azul. Podemos comprobar como el error se dispara.

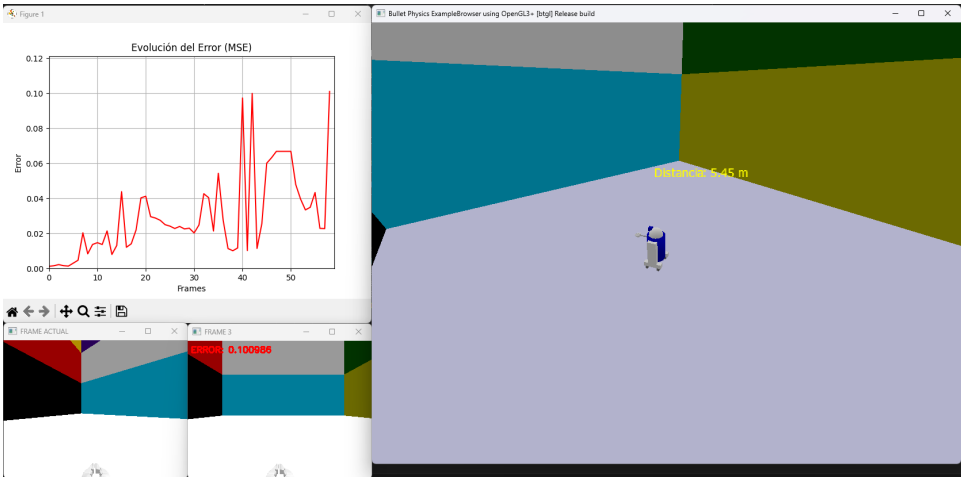

Figura 6.4: Resultados de la evaluación manual del modelo LSTM en un escenario con muros decorados con banderas. En este caso se desconoce la bandera de color azul. Podemos comprobar como el error se dispara cada vez que la ve.

Capítulo 7

Próximos pasos

El trabajo realizado en este proyecto demuestra la viabilidad de utilizar un agente de aprendizaje por refuerzo para aprender políticas de exploración mientras se detectan anomalías visuales en un entorno simulado. Sin embargo, el campo de la detección de anomalías y el control autónomo avanza rápidamente, y existen múltiples direcciones de mejora. En este capítulo se proponen varias líneas de trabajo para enriquecer el sistema, optimizar su rendimiento y trasladar la metodología a entornos más realistas. Cada subsección analiza las ventajas de la aproximación propuesta, su fundamento teórico y los retos técnicos que plantea.

7.1. Modelo de generación de *embeddings* con autoencoder

En el prototipo actual se emplea la red ResNet50 para extraer representaciones de 2 048 dimensiones a partir de las imágenes RGB. Estas representaciones se alimentan al modelo LSTM encargado de predecir la siguiente descripción del entorno. Aunque las redes convolucionales pre-entrenadas son eficaces, consumen memoria y no están optimizadas para capturar las estructuras específicas del entorno simulado. Una alternativa prometedora consiste en entrenar un autoencoder sobre las imágenes del simulador y utilizar su representación latente como *embedding*.

Un autoencoder es una red neuronal no supervisada compuesta por un codificador que comprime la entrada a un espacio latente de baja dimensión y un decodificador que intenta reconstruir la entrada a partir de esa representación. En la literatura se ha demostrado que los autoencoders pueden *comprimir dinámicas complejas en representaciones de baja dimensión y reconstruirlas con gran fidelidad*; además, sus embeddings son tan efectivos o incluso mejores que los datos originales en tareas de clasificación o predicción[55]. Estas redes capturan la información esencial de los datos de entrada con muchas menos variables que los modelos mecanicistas tradicionales[55].

Al entrenar un autoencoder específico para el entorno simulado es posible obtener representaciones más compactas que las de ResNet y optimizadas para las características visuales del escenario.

El entrenamiento podría realizarse de forma no supervisada generando un gran número de imágenes sintéticas de distintos mapas y condiciones de iluminación. El codificador aprendería a extraer características de alto nivel (colores, formas, texturas) y la dimensionalidad de la capa latente se ajustaría para equilibrar la compresión con la capacidad de reconstrucción. Una vez entrenado, el autoencoder serviría como generador de *embeddings* y su salida reemplazaría al vector de 2 048 valores de ResNet. Estos embeddings podrían alimentar tanto al detector de anomalías como al agente RL. Además, el modelo codificador puede ser compacto y eficiente si se utiliza una arquitectura ligera (por ejemplo, capas convolucionales y convoluciones separables) y técnicas de regularización como autoencoders *sparsos* o variacionales. Como los embeddings están ajustados al entorno, se espera que la red LSTM o su sustituto (véase Sección 7.3) necesite menos capacidad para modelar las dinámicas.

Otra ventaja de esta aproximación es que el autoencoder se puede optimizar conjuntamente con el detector de anomalías para minimizar el error de reconstrucción y maximizar la separabilidad entre patrones normales y anómalos. La literatura sobre compresión de datos y microbiología muestra que estas representaciones pueden incluso superar a los datos originales en la detección de patrones sutiles[55]. Como trabajo futuro se propone implementar un autoencoder y comparar la calidad de sus embeddings frente a las características de ResNet, evaluando su impacto en la detección de anomalías y en el aprendizaje por refuerzo.

7.2. Edge computing usando similitud coseno

El sistema actual se ejecuta en un entorno de escritorio y requiere una GPU para procesar las imágenes y calcular las recompensas en tiempo real. Para desplegarlo en robots de bajo consumo o en plataformas de campo sería necesario reducir el consumo de memoria y la latencia. La computación en el borde (edge computing) permite *procesar los datos cerca de donde se generan, reduciendo la latencia y evitando la transferencia de grandes volúmenes de datos a un servidor central*[57]. El procesamiento local es especialmente relevante en aplicaciones de detección de anomalías donde se requiere una respuesta en tiempo real; además, algunas implementaciones colocan autoencoders ligeros en los dispositivos edge y actualizan los modelos localmente con un servidor central que redistribuye versiones comprimidas[57].

Una estrategia para reducir la complejidad del modelo consiste en utilizar autoencoders cuantizados. En trabajos recientes se ha propuesto un *quantized autoencoder* (QAE) para detección de intrusos que permite desplegar

modelos de autoencoder en dispositivos IoT y edge con recursos limitados; esta técnica logra reducir el uso de memoria en un 70 %, el tamaño del modelo en un 92 % y la utilización de CPU en un 27 % con respecto al autoencoder completo, manteniendo su capacidad para detectar anomalías de forma no supervisada[56]. El proceso de cuantización se basa en podar pesos irrelevantes, agrupar parámetros en clústeres y convertir los pesos flotantes a formatos enteros de baja precisión. El resultado es un modelo ligero que puede ejecutarse en microcontroladores y procesadores embebidos.

Además de la cuantización, se puede emplear la *similitud coseno* como métrica de comparación entre embeddings. La similitud coseno mide el ángulo entre dos vectores y es independiente de sus magnitudes; es útil para comparar representaciones normalizadas y detectar desviaciones en la dirección del espacio latente. Comparar el embedding actual con el embedding reconstruido por el autoencoder mediante la similitud coseno proporciona un indicador de normalidad sin necesidad de redes recurrentes. Esta métrica es computacionalmente más sencilla que la distancia euclídea en espacios de alta dimensión y se adapta bien a implementaciones hardware. En combinación con los modelos QAE, la similitud coseno podría ejecutarse en microprocesadores de 32 bits sin necesidad de operaciones de coma flotante, permitiendo la detección de anomalías en robots de bajo consumo.

El uso de *knowledge distillation* y cuantización para desplegar modelos en el borde también se ha explorado en la literatura. Al transferir el conocimiento de un modelo grande (profesor) a un modelo pequeño (estudiante) mediante pérdidas basadas en similitud coseno se obtiene un estudiante compacto que conserva la precisión del modelo original; esta técnica se complementa con la cuantización y permite que redes de detección se ejecuten en dispositivos embebidos con latencia reducida[56]. Una línea de trabajo futura será rediseñar el módulo de detección para que utilice un autoencoder cuantizado y la similitud coseno como métrica de error, evaluando su rendimiento en un ordenador monoplaca (Raspberry Pi) y comparando la detección con la versión original.

7.3. Transformers y arquitecturas *BERT*

El modelo de detección de anomalías actual se basa en una red LSTM que, aunque eficaz, presenta limitaciones a la hora de capturar dependencias muy largas y paralelizar el aprendizaje. Los transformers se han convertido en una arquitectura dominante en campos como el procesamiento del lenguaje natural, la visión por computador y la recomendación, donde han demostrado un rendimiento óptimo y una reducción significativa en el número de parámetros[58]. La clave de estas redes es el mecanismo de autoatención que proyecta las secuencias en un espacio de alta dimensión y calcula las dependencias internas mediante productos escalares y pesos de atención.

Esta operación se puede paralelizar completamente y evita los problemas de desvanecimiento del gradiente de las RNN.

En el contexto de la detección de anomalías, se han desarrollado variantes como el *disentangled dynamic deviation transformer network*, que combina un bloque de características basado en grafos y un bloque temporal basado en transformer para modelar las dependencias complejas entre sensores y a lo largo del tiempo. La parte temporal aprende relaciones de largo y corto plazo mediante autoatención, mejorando la capacidad de predicción y elevando el nivel de detección de anomalías[58]. Este modelo demuestra que los transformers pueden superar a las redes recurrentes al capturar mejor las correlaciones multiescala en series temporales. Otros trabajos han demostrado que el transformer reduce la dimensión de los parámetros manteniendo o mejorando la precisión, e introduce mecanismos de atención explicables, selección de características y puertas que suprimen elementos redundantes[58].

Para nuestro sistema, sustituir la LSTM por un transformer o por una arquitectura tipo BERT (entrenada con pretextos como el enmascaramiento de pasos temporales) permitiría aprender dependencias de largo alcance entre embeddings visuales. La entrada al transformer serían secuencias de embeddings generados por el autoencoder o por ResNet, y el modelo predeciría el siguiente embedding de la secuencia. Las ventajas incluyen el paralelismo durante el entrenamiento, la posibilidad de preentrenamiento autosupervisado en grandes volúmenes de datos sintéticos y la mejora de la interpretación del modelo mediante las matrices de atención. También se podría emplear un transformer ligero (como MobileBERT) para reducir la complejidad y adaptarlo al edge computing. El reto principal será ajustar el tamaño del modelo y la longitud de las ventanas de atención para que el entrenamiento sea manejable.

7.4. Robot físico

El proyecto original contemplaba la utilización de un robot físico para evaluar el sistema de detección de anomalías en un entorno real. El objetivo era observar cómo el robot hexápodo reaccionaba ante cambios inesperados en su entorno, combinando un módulo de percepción visual, un detector de anomalías y un agente de control. Sin embargo, durante el desarrollo surgieron diversos impedimentos relacionados con la estabilidad de los sensores, la latencia en la comunicación remota y la dificultad de generar un entorno controlado reproducible. Estas limitaciones llevaron a adoptar un entorno simulado como solución provisional. A pesar de ello, se avanzó considerablemente en la preparación del robot físico, documentando los materiales, el montaje y la arquitectura de conexión empleada.

7.4.1. Materiales utilizados

Para la fase de robot físico se empleó una combinación de componentes de hardware y software. La plataforma mecánica elegida fue el *Freenove Big Hexapod Robot Kit for Raspberry Pi*, que proporciona una estructura de seis patas equipada con 20 servomotores, un controlador y soporte para la Raspberry Pi. Como ordenador de a bordo se utilizó una *Raspberry Pi 3 Model B+* con una cámara oficial y un sensor de ultrasonidos HC–SR04 para capturar imágenes y medir distancias. El kit se alimentó con baterías recargables y se completó con cables de conexión (incluyendo RJ45 para la configuración inicial), un adaptador microSD–SD y una tarjeta microSD para el sistema operativo. Además, se preparó un conjunto de objetos cotidianos (zapatillas, gorras, macetas, peluches) y un recinto de cartón de 3×3 m para crear escenas de entrenamiento.

En cuanto al software, se instaló *Raspberry Pi OS* en la tarjeta microSD mediante Raspberry Pi Imager y se configuraron servicios de red (SSH, VNC y FTP) para la comunicación remota. El programa de control se desarrolló en Python aprovechando la librería oficial de Freenove para gestionar los servomotores, la cámara y el sensor de ultrasonidos. Como modelo de percepción se utilizó la red *ResNet50*, integrada para extraer características de las imágenes captadas. Esta configuración es coherente con estudios recientes que utilizan la Raspberry Pi como controlador de robots hexápodos y gestionan los servomotores mediante comunicación I2C, delegando la percepción y el envío de datos a un ordenador principal.

7.4.2. Montaje

El montaje del robot físico se realizó siguiendo las instrucciones del kit Freenove. Se distinguen cinco fases principales:

1. **Instalación del sistema operativo.** La Raspberry Pi se configuró con Raspberry Pi OS, habilitando los servicios SSH y VNC para el acceso remoto. Se verificó la conectividad mediante cable HDMI, teclado y ratón, y se instaló el cliente VNC para futuras conexiones.

2. **Ensamblaje de la estructura.** Se montaron las patas y el chasis del hexápodo siguiendo la guía del fabricante, asegurando que cada servomotor quedase en la posición correcta. Se conectaron los servomotores al controlador del kit y se colocó la Raspberry Pi en su soporte.

3. **Instalación de servomotores.** Cada pata dispone de varios grados de libertad; se conectaron los 20 servomotores al controlador y se calibraron mediante el modo de calibración incluido en la librería de Freenove, para que los movimientos fueran precisos.

4. **Instalación de cámara y sensor de ultrasonidos.** La cámara se montó en el frontal del robot y se conectó a la Raspberry Pi; el sensor HC–SR04 se instaló para medir distancias. Se verificó su funcionamiento mediante pruebas sencillas en Python.

5. **Calibración y pruebas.** Tras el montaje, se ejecutaron los modos de calibración y ahorro de la librería de Freenove para asegurar que los servomotores respondían correctamente. Se realizaron pruebas de movimiento y captura de imágenes en el recinto de cartón preparado para el entrenamiento.

7.4.3. Conectividad y arquitectura de comunicación

Debido a las limitaciones de cómputo de la Raspberry Pi 3B+, se decidió emplear una arquitectura cliente–servidor para el control del robot. La Raspberry Pi actúa como servidor: captura imágenes y lecturas del sensor de ultrasonidos y envía estos datos al ordenador portátil a través de *sockets*. El programa servidor se ejecuta en la Raspberry Pi y utiliza funciones de la librería Freenove para obtener la imagen y la distancia:

Listing 7.1: Captura de imagen y distancia en el servidor

```
def capture_image(self):
    # Guarda la imagen en disco y devuelve sus bytes
    self.camera.save_image(IMG_FILENAME)
    with open(IMG_FILENAME, 'rb') as f:
        return f.read()

def capture_distance(self):
    # Obtiene la distancia medida por el sensor de ultrasonidos
    return self.ultrasonic.get_distance()

def send_data(self, img_bytes, distance):
    # Env a longitud de imagen, imagen y distancia al cliente
    img_len = len(img_bytes)
    self.client_socket.sendall(struct.pack('<I', img_len))
    self.client_socket.sendall(img_bytes)
    self.client_socket.sendall(struct.pack('<f', distance))
    print(f"Datos enviados: imagen ({img_len} bytes), distancia: {
        distance} cm")

def receive_action(self):
    # Recibe la acci n seleccionada por el cliente
    data = self.client_socket.recv(1024).decode()
    if not data:
        return None
    return data.strip()

while True:
    img_bytes = capture_image()
    distance = capture_distance()
    send_data(img_bytes, distance)
    action = receive_action()
    # Aplicar la acci n a los servomotores...
```

En el ordenador portátil se ejecuta el cliente, que recibe la imagen y la distancia, calcula los *embeddings* mediante ResNet50, actualiza un buffer de cinco imágenes y utiliza un modelo LSTM para predecir la imagen central. El error entre la predicción y la imagen real se compara con un umbral para determinar si existe una anomalía y se emplea como recompensa para un agente DQN que decide la siguiente acción. El control manual con teclas `WASD` permite recopilar entre 200 y 400 imágenes de entrenamiento antes de lanzar el aprendizaje automático.

Esta arquitectura se alinea con estudios que utilizan la Raspberry Pi para controlar robots móviles y transmitir datos sensoriales a un ordenador principal mediante Wi-Fi; en estos sistemas la Raspberry Pi gobierna el movimiento y comunica los datos a través de ROS o *topics* mientras los servomotores se controlan mediante I2C. La separación de tareas facilita el procesamiento intensivo en el portátil y la captura en tiempo real en el robot, a la vez que simplifica la depuración y el desarrollo.

7.4.4. Conclusiones y problemas encontrados

Aunque el montaje y la conectividad se completaron con éxito, surgieron problemas que motivaron la transición a la simulación. La Raspberry Pi tiene recursos limitados, por lo que el procesamiento de visión y el control en tiempo real provocaban retardos perceptibles. La latencia en la comunicación Wi-Fi añadía retraso en la respuesta del robot, y la sensibilidad del sensor de ultrasonidos generaba lecturas ruidosas. Además, crear un entorno físico controlado con variedad de anomalías resultó laborioso. Estas dificultades, junto con la mayor flexibilidad de un entorno simulado, justificaron la decisión de finalizar el proyecto en un mundo virtual. No obstante, la experiencia adquirida en el montaje y la comunicación servirá como base para una implementación futura en hardware real, donde se evaluarán las políticas aprendidas y se aprovechará la arquitectura cliente–servidor descrita.

7.5. Redes convolucionales y odometría

La simulación empleada hasta ahora considera un entorno discreto en el que el agente aprende a moverse con tres acciones (avanzar, girar a la izquierda y girar a la derecha). Aunque esta aproximación es válida para explorar el concepto de detección de anomalías, carece de una percepción semántica detallada y de la capacidad de estimar la pose del robot con precisión. La comunidad de SLAM y navegación ha evolucionado hacia sistemas que combinan la detección de objetos con algoritmos de odometría para operar en entornos dinámicos. Una línea de trabajo futura consiste en integrar redes convolucionales de detección (por ejemplo, YOLO) y técnicas de posicionamiento mediante odometría para mejorar el conocimiento del entorno y la localización del robot.

Los sistemas de SLAM basados en YOLO utilizan redes de detección ligeras para identificar objetos dinámicos y estáticos y, mediante la segmentación, eliminan los puntos de características asociados a objetos en movimiento. Por ejemplo, el sistema SEG-SLAM construye un módulo de fusión de detección y segmentación con YOLOv5 sobre el framework ORB-SLAM3 para obtener información semántica de los objetos dinámicos en tiempo real[59]. Este sistema utiliza la profundidad y la geometría epipolar para juzgar los puntos de características dinámicos y potencialmente dinámicos, eliminando aquellos que perturban el mapeado; al reconstruir el fondo estático, mejora la precisión de localización y la construcción de mapas[59]. De este modo, el robot puede navegar en escenas con personas u objetos en movimiento sin que su estimación de pose se degrade.

Integrar una red YOLO con odometría supondría dotar al agente de un modelo de localización visual en tiempo real. El detector YOLO proporcionaría cuadros delimitadores y etiquetas de clase de los objetos vistos; estos servirían para identificar obstáculos y regiones de interés. Paralelamente, un módulo de odometría (por ejemplo, ORB-SLAM o una variante de odometría visual inercial) estimaría la trayectoria del robot a partir de características visuales y sensores inerciales. Los objetos detectados podrían usarse para enriquecer el mapa con información semántica o para ajustar la detección de anomalías: encontrar objetos inesperados o cambios en la apariencia de los muros (como banderas alteradas) sería más fácil si se dispone de sus ubicaciones. Este enfoque también abre la puerta a algoritmos de planificación que combinen la detección de anomalías con la navegación hacia zonas desconocidas.

Adicionalmente, se podrían explorar redes de detección más recientes (YOLOv8 u otras variantes eficientes) junto con técnicas de rechazo dinámico basadas en máscaras y profundidad. El objetivo es construir un sistema que no sólo detecte anomalías visuales sino que también sepa dónde están en el mapa, posibilitando intervenciones automáticas o asistencia a operadores humanos.

Capítulo 8

Conclusiones

El trabajo presentado en esta memoria demuestra que la combinación de técnicas de visión artificial, modelos de detección de anomalías y aprendizaje por refuerzo es una vía prometedora para dotar a robots móviles de una capacidad de exploración inteligente en entornos desconocidos. A lo largo del proyecto se ha diseñado un sistema completo —desde la generación de datos sintéticos hasta la evaluación cuantitativa— que integra un entorno de simulación 3D basado en PyBullet, un módulo de percepción que extrae descriptores visuales mediante ResNet50, un modelo de detección de anomalías basado en redes LSTM y un agente de aprendizaje por refuerzo (DQN) que utiliza el error de reconstrucción como señal de recompensa. Esta cadena de procesamiento permite que el robot aprenda a desplazarse evitando zonas con alto error visual, es decir, regiones que difieren de su experiencia previa.

La fase de planificación y diseño supuso descomponer el problema en módulos autocontenidos, definir interfaces claras entre ellos y establecer un calendario realista. El diseño conceptual permitió identificar los componentes clave y fijar las restricciones de tiempo y recursos. En la metodología se detalló la implementación del entorno simulado, el control reactivo basado en sensores ultrasónicos, el detector de anomalías y la política de exploración. Las evaluaciones automáticas mostraron que el agente DQN es capaz de reducir progresivamente el error de reconstrucción gracias al reentrenamiento en línea, aunque tiende a explotar trayectorias conocidas y no explora suficientemente el entorno, lo que ejemplifica el clásico dilema exploración–explotación. La evaluación manual del detector LSTM demostró que el sistema responde con sensibilidad a cambios en la forma y textura de los objetos, pero arroja algunos falsos positivos en escenarios con variaciones cromáticas menores.

Los resultados indican que la adaptación en línea es esencial para mantener el rendimiento en presencia de *concept drift*. Los modelos entrenados en lotes pueden quedarse obsoletos a medida que cambian las propiedades del entorno, mientras que el aprendizaje incremental actualiza los parámetros

de la red en función de los nuevos datos y mantiene su precisión. Asimismo, la política de exploración debería enriquecerse con métodos que equilibren la búsqueda de nuevas zonas y la explotación de caminos ya conocidos, como técnicas de curiosidad intrínseca o políticas adaptativas basadas en incertidumbre. La comparación entre los escenarios con objetos y con banderas sugiere que el sistema es más eficaz cuando las anomalías implican cambios estructurales bien definidos; un mayor número de ejemplos de entrenamiento y técnicas de regularización podrían reducir los falsos positivos en casos sutiles.

En conjunto, este proyecto sienta las bases para un sistema autónomo capaz de detectar y aprender patrones visuales inusuales mientras navega de forma inteligente. La modularidad de la arquitectura facilita la incorporación de mejoras: por ejemplo, emplear autoencoders para generar *embeddings* más compactos, trasladar los cálculos a dispositivos *edge* con modelos cuantizados, sustituir la LSTM por transformers o BERT para capturar dependencias de largo alcance, o integrar redes de detección como YOLO junto con odometría visual para dotar al agente de percepción semántica y localización precisa. La extensión a un robot físico será el siguiente hito lógico, permitiendo validar las políticas aprendidas en escenarios reales y cerrar el ciclo de simulación–realidad. Esta línea de investigación abre la puerta a aplicaciones en vigilancia, inspección industrial y entornos poco accesibles, donde la detección de anomalías y la autonomía del robot son críticas.

Capítulo 9

Referencias

[1] Frontiers. (2025). *A systematic survey: role of deep learning-based image anomaly detection in industrial inspection contexts*. Frontiers in Robotics and AI. https://www.frontiersin.org/articles/10.3389/frobt.2025.1554196/full

[2] Olah, C. (2015). *Understanding LSTM Networks*. Colah's Blog. https://colah.github.io/posts/2015-08-Understanding-LSTMs/

[3] Sunder, R. (2021). *Spinning Up in Deep Reinforcement Learning (With PyBullet)*. Medium. https://medium.com/sorta-sota/spinning-up-in-deep-reinforcement-l

[4] Li, X., Hsu, K., Gu, J., *et al.* (2024). *Evaluating Real-World Robot Manipulation Policies in Simulation*. arXiv:2405.05941. https://arxiv.org/abs/2405.05941

[5] Shen, D., Wu, G., & Suk, H. I. (2018). *Deep Learning in Medical Image Analysis*. Springer. https://doi.org/10.1007/s13244-018-0639-9

[6] Inoxoft. (2023). *Deep Q-Learning Explained: Step-by-Step Guide*. Inoxoft. https://inoxoft.com/blog/deep-q-learning-explained-a-comprehensive-guide/

[7] MindBridge. (2025). *Anomaly Detection Techniques: How to Uncover Risks, Identify Patterns, and Strengthen Data Integrity*. MindBridge Blog. https://www.mindbridge.ai/blog/anomaly-detection-techniques-how-to-uncover-risks-identi

[8] Cloudera. (2020). *Deep Learning for Anomaly Detection*. Cloudera Fast Forward Labs. https://ff12.fastforwardlabs.com/

[9] MathWorks. (2025). *What Is Robot Simulation?*. Disponible en: https://www.mathworks.com/discovery/robot-simulation.html

[10] Robotics Knowledgebase. (2025). *Choose a Simulator – PyBullet*. Recuperado de https://roboticsknowledgebase.com/wiki/robotics-project-guide/choose-a-sim/

[11] Robotics Knowledgebase. (2025). *Choose a Simulator – Isaac Lab*. Recuperado de https://roboticsknowledgebase.com/wiki/robotics-project-guide/choose-a-sim/

[12] Open Source Robotics Foundation. (2023). *Gazebo*. Disponible en: https://openrobotics.org/

[13] Unity Technologies. (2025). *ML-Agents Toolkit Overview*. Consultado en https://unity-technologies.github.io/ml-agents/ML-Agents-Overview/

[14] Zilliz. (2025). *What is the OpenAI Gym?*. Disponible en: https://milvus.io/ai-quick-reference/what-is-the-openai-gym

[15] Unity Technologies. (2025). *Unity ML-Agents Gym Wrapper*. Disponible en: https://unity-technologies.github.io/ml-agents/Python-Gym-API/

[16] MathWorks. (2025). *MATLAB and Simulink for Mobile Robots*. Recuperado de https://www.mathworks.com/solutions/robotics/mobile-robots.html

[17] Investopedia. (2024). *What Is a Neural Network?*. Recuperado de https://www.investopedia.com/terms/n/neuralnetwork.asp

[18] Wikipedia. (2024). *Backpropagation*. Recuperado de https://en.wikipedia.org/wiki/Backpropagation

[19] Boesch, G. (2023). *Deep Residual Networks (ResNet, ResNet-50) – A Complete Guide*. viso.ai. Recuperado de https://viso.ai/deep-learning/resnet-residual-neural-network/

[20] DataCamp. (2024). *YOLO Object Detection Explained*. Recuperado de https://www.datacamp.com/blog/yolo-object-detection-explained

[21] Olah, C. (2015). *Understanding LSTM Networks*. Blog de Colah. Recuperado de https://colah.github.io/posts/2015-08-Understanding-LSTMs/

[22] DataCamp. (2024). *How Transformers Work: A Detailed Exploration of Transformer Architecture*. Recuperado de https://www.datacamp.com/tutorial/how-transformers-work

[23] Devlin, J., Chang, M.-W., Lee, K., & Toutanova, K. (2019). *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. arXiv preprint arXiv:1810.04805.

[24] Bandyopadhyay, H. (2021). *Autoencoders in Deep Learning: Tutorial & Use Cases*. V7 Labs. Recuperado de https://www.v7labs.com/blog/autoencoders-guide

[25] GeeksforGeeks. (2025). *Backpropagation in Neural Network*. Recuperado de https://www.geeksforgeeks.org/machine-learning/backpropagation-in-neural-

[26] GeeksforGeeks. (2025). *Introduction to Convolution Neural Network*. Recuperado de https://www.geeksforgeeks.org/machine-learning/introduction-convolu

[27] Dive into Deep Learning. (2025). *7.6. Residual Networks (ResNet)*. En *Convolutional Modern Neural Networks*. Recuperado de https://classic.d2l.ai/chapter_convolutional-modern/resnet.html

[28] Dive into Deep Learning. (2025). *9.2. Long Short-Term Memory (LSTM)*. Recuperado de https://classic.d2l.ai/chapter_recurrent-modern/lstm.html

[29] UvA Deep Learning Course. (2025). *Transformers and Multi-Head Attention*. Recuperado de https://uvadlc-notebooks.readthedocs.io/en/latest/tutorial_notebooks/transformers/transformer.html

[30] Columbia University. (2023). *The Basics of Language Modeling with Transformers: BERT*. Recuperado de https://columbia.nycai.org/bert

[31] GeeksforGeeks. (2024). *Autoencoders in Machine Learning*. Recuperado de https://www.geeksforgeeks.org/autoencoders-in-machine-learning/

[32] GeeksforGeeks. (2025). *Anomaly Detection in Time Series Data*. Recuperado de https://www.geeksforgeeks.org/machine-learning/anomaly-detection-in-time-series-data/

[33] Pinecone. (2023). *Vector Similarity Explained*. Recuperado de https://www.pinecone.io/learn/vecto similarity/

[34] Athar, A., Mozumder, M. A. I., & Ali, S. *et al.* (2024). Deep learning-based anomaly detection using one-dimensional convolutional neural networks (1D CNN) in machine centers (MCT) and CNC machines. *PeerJ Computer Science*. Recuperado de https://pmc.ncbi.nlm.nih.gov/articles/PMC11623112/

[35] Esmaeili, F., Cassie, E., Nguyen, H. P. T., *et al.* (2023). Anomaly Detection for Sensor Signals Utilizing Deep Learning Autoencoder-Based Neural Networks. *Bioengineering*, 10(4), 405. Recuperado de https://pmc.ncbi.nlm.nih.gov/articles/PMC10136

[36] GeeksforGeeks. (2025). *Supervised vs Unsupervised vs Reinforcement Learning*. Disponible en https://www.geeksforgeeks.org/machine-learning/supervised-vs-reinforcement-vs-unsupervised/

[37] GeeksforGeeks. (2025). *Reinforcement Learning*. Recuperado de https://www.geeksforgeeks.org/reinforcement-learning/

[38] AI Online Course. (2024). *What is Behavior Cloning*. Disponible en https://www.aionlinecourse.com/ai-basics/behavior-cloning

[39] Laskin, M., Yarats, D., & al. (2021). *The Unsupervised Reinforcement Learning Benchmark*. Blog del Berkeley Artificial Intelligence Research (BAIR). Recuperado de https://bair.berkeley.edu/blog/2021/12/15/unsupervised-rl/

[40] Spinning Up Documentation. (2023). *Proximal Policy Optimization*. Recuperado de https://spinningup.openai.com/en/latest/algorithms/ppo.html

[41] GeeksforGeeks. (2025). *Deep Q-Learning in Reinforcement Learning*. Recuperado de https://www.geeksforgeeks.org/deep-learning/deep-q-learning/

[42] Kissflow. (2025). *5 Phases of Project Management – A Complete Breakdown*. Recuperado de https://kissflow.com/project/five-phases-of-project-management/

[43] Boom & Bucket. (2025). *Conceptual Design in Construction and Its Relation to Concept Development*. Recuperado de https://www.boomandbucket.com/blog/conceptual-design

[44] GeeksforGeeks. (2025). *Software Cost Estimation*. Recuperado de https://www.geeksforgeeks.org/software-engineering/software-cost-estimation/

[45] Asana. (2025). *How to Write an Effective Project Objective, With Examples*. Recuperado de https://asana.com/resources/how-project-objectives

[46] GeeksforGeeks. (2024). *Q-Learning in Reinforcement Learning*. Recuperado de https://www.geeksforgeeks.org/q-learning-in-reinforcement-learning/

[47] GeeksforGeeks. (2024). *Deep Q-Learning in Reinforcement Learning*. Recuperado de https://www.geeksforgeeks.org/deep-q-learning-in-reinforcement-learning/

[48] OpenAI Spinning Up. (2023). *Proximal Policy Optimization*. Recuperado de https://spinningup.openai.com/en/latest/algorithms/ppo.html

[49] Berkeley Artificial Intelligence Research (BAIR). (2021). *The Unsupervised Reinforcement Learning Benchmark*. Recuperado de https://bair.berkeley.edu/blog/202
rl/

[50] AI Online Course. (2024). *What is Behavior Cloning*. Recuperado de https://www.aionlinecourse.com/ai-basics/behavior-cloning

[51] Z score for outlier detection - GeeksforGeeks. (2025). *Z score for Outlier Detection - Python*. Consultado el 19 de agosto de 2025, de https://www.geeksforgeeks.org
learning/z-score-for-outlier-detection-python/

[52] GeeksforGeeks. (2024). *What is Heatmap Data Visualization and How to Use It?*. Recuperado de https://www.geeksforgeeks.org/data-visualization/what-is-heatmap-data-visualization-and-how-to-use-it/

[53] D. Hirtenstein *et al.*, "Online Machine Learning for Anomaly Detection in Time Series Data," 2024. El artículo explica que los modelos entrenados en lotes pueden quedarse obsoletos debido al *concept drift* y que el aprendizaje en línea adapta continuamente el modelo a los cambios, manteniendo la precisión.

[54] R. C. Wilson, E. Bonawitz, V. D. Costa y R. B. Ebitz, "Balancing exploration and exploitation with information and randomization," *Current Opinion in Behavioral Sciences*, vol. 38, pp. 49–56, 2021. Este trabajo revisa el dilema exploración–explotación y argumenta que los organismos combinan una exploración dirigida (sesgo hacia la información) y una exploración aleatoria para equilibrar el aprendizaje de nuevas recompensas y la obtención de recompensas inmediatas.

[55] Y. Baig, H. R. Moore, H. Xu, y col. *Autoencoder neural networks enable low-dimensional structure analyses of microbial growth dynamics*. Nature Communications, 14:7937, 2023. Esta obra demuestra que los autoencoders pueden comprimir dinámicas complejas en representaciones de baja dimensión y reconstruirlas con gran fidelidad, obteniendo embeddings tan efectivos o mejores que los datos originales para tareas de clasificación y predicción.

[56] F. Esmaeili, E. Cassie, H. P. T. Nguyen, *et al. Anomaly Detection for Sensor Signals Utilizing Deep Learning Autoencoder-Based Neural Networks*. Bioengineering, 10(4):405, 2023. El artículo propone modelos de autoencoder cuantizados (QAE-u8 y QAE-f16) para detectar anomalías de forma no supervisada en dispositivos de IoT con recursos limitados. El modelo QAE-u8 reduce el uso de memoria en un 70 %, el tamaño del modelo en un 92 % y la utilización de CPU en un 27 % en comparación con un autoencoder completo, lo que permite su implementación en dispositivos edge.

[57] D. R. Patrikar y M. R. Parate. *Anomaly detection using edge com-*

puting in video surveillance system: review. International Journal of Multimedia Information Retrieval, 11(2):85–110, 2022. Este estudio explica que la computación en el borde procesa los datos donde se generan, reduciendo la latencia en aplicaciones de detección de anomalías y permitiendo que autoencoders ligeros se desplieguen en dispositivos terminales; también señala que las tareas de detección pueden repartirse entre dispositivos para reducir los tiempos de respuesta.

[58] H. Y. Zhang, J. F. Xu, F. Wu, *et al. Disentangled Dynamic Deviation Transformer Networks for Multivariate Time Series Anomaly Detection*. IEEE Transactions on Neural Networks and Learning Systems, 2023. El trabajo introduce un bloque temporal basado en transformer que aprende las dependencias de largo y corto plazo mediante autoatención; esta estrategia mejora la capacidad de predicción y la detección de anomalías al considerar correlaciones multiescala en distintos pasos de tiempo739068755947407†L352-L356. Además, los autores destacan que los transformers se utilizan ampliamente en diversos campos y reducen el número de parámetros al tiempo que capturan relaciones de largo alcance.

[59] P. Cong, J. Li, J. Liu, *et al. SEG-SLAM: Dynamic Indoor RGB-D Visual SLAM Integrating Geometric and YOLOv5-Based Semantic Information*. Sensors, 24(7):2102, 2024. El sistema SEG-SLAM construye un módulo de fusión de detección y segmentación con YOLOv5 sobre ORB-SLAM3; combina detección de objetos y segmentación semántica con información de profundidad y geometría epipolar para juzgar puntos de características dinámicos y eliminar aquellos que perturban el mapeado. Esto reduce la interferencia de objetos dinámicos, mejora la reconstrucción del fondo estático y aumenta la precisión de localización y de generación de mapas.