Universidad Politécnica
de Madrid

Escuela Técnica Superior de Ingenieros Informáticos

Máster Universitario en Software y Sistemas

Trabajo Fin de Máster

Desarrollo de un Modelo de Aprendizaje por Refuerzo para el Control de Movimiento de un Robot

Autor(a): Jose Luis Carrillo Onairan

Tutor(a): Santiago Tapia

Madrid, 07/2025

Este Trabajo Fin de Máster se ha depositado en la ETSI Informáticos de la Universidad Politécnica de Madrid para su defensa.

Trabajo Fin de Máster
Máster Universitario en Software y Sistemas
Título: Entrenamiento de un Modelo de Aprendizaje por Refuerzo para el Control de Movimiento de un Robot
Mes Año

Autor(a): Jose Luis Carrillo Onairan

Tutor(a):
Santiago Tapia
Dep. de Lenguajes y Sistemas Informáticos e Ingeniería de Software
ETSI Informáticos
Universidad Politécnica de Madrid

Agradecimientos

A mi familia, por su apoyo incondicional y por ser mi impulso constante para seguir adelante, incluso en los momentos más desafiantes de mi estancia en Madrid.

Al Dr. Santiago Tapia, por creer en mis capacidades y por su dedicación y acompañamiento durante cada etapa de este Trabajo de Fin de Máster.

A la Universidad Politécnica de Madrid, por ofrecerme una formación de excelencia y por el valioso acompañamiento de sus docentes a lo largo de esta etapa.

Y, de manera muy especial, a mi novia, por ser mi refugio y mi fortaleza en este proceso.

Resumen

El aprendizaje por refuerzo (RL) ha impulsado la robótica móvil, donde los robots aprenden a navegar por terrenos complejos sin reglas predeterminadas o mapas previos altamente detallados. Este trabajo de fin de Master sigue esta tendencia al enfocarse en el diseño, entrenamiento y validación de un modelo utilizando el algoritmo Proximal Policy Optimization (PPO) para el control autónomo de un ROSbot XL simulado en Gazebo Ignition y ROS 2 Humble. El objetivo es entrenar a un agente para lograr metas establecidas y evitar choques mientras se ajustan a las condiciones cambiantes de manera segura y efectiva; todo ello basado únicamente en los datos sensoriales del LIDAR y la odometría del robot.

El enfoque del proyecto radica en crear un entorno de observación que equilibre la riqueza sensorial y la eficiencia computacional al reducir las medidas LIDAR de 282 dimensiones a solo 37. Esto garantiza conservar información crucial para la navegación sin afectar la rapidez del proceso de entrenamiento. Esta estrategia de reducción junto con la definición de la función de recompensa, permitirá que el agente priorice la alineación al objetivo y evitar los obstáculos de manera proactiva, premiando los comportamientos positivos y penalizando aquellas acciones que pongan en riesgo la integridad del robot o su capacidad para completar la tarea asignada.

Para el entrenamiento, se realizó la integración con ROS 2, Gazebo Ignition y Gymnasium, estableciendo un entorno modular que permitió al ROSbot aprender en 200,000 pasos de entrenamiento en la simulación, con supervisión constante de métricas mediante TensorBoard. Se empleó una arquitectura MLP para procesar observaciones vectoriales de forma eficiente, logrando un aprendizaje progresivo en el que el agente desarrolló maniobras de avance alineado, evasión de colisiones y desbloqueo ante estancamientos, consolidando una política robusta y segura.

En la etapa de validación del modelo, los resultados demostraron que el agente logró alcanzar exitosamente diversos objetivos tanto en escenarios familiares (ya vistos durante el entrenamiento) como en ambientes nuevos con obstáculos. El robot no dependió en ningún momento de la memorización de rutas concretas, sino que tomó decisiones en tiempo real basadas únicamente en sus percepciones sensoriales. Estos resultados también apoyan la usabilidad y aplicabilidad del aprendizaje por refuerzo en escenarios de robótica en simulación, para la posterior transferencia de políticas aprendidas a entornos real.

Finalmente, este Trabajo de Fin de Master presenta una contribución práctica y académica en el área de la robótica móvil, mostrando cómo el aprendizaje por refuerzo puede ser aplicado para crear agentes que pueden navegar por espacios complejos de manera progresiva, segura y autónoma, presentando un proceso de aprendizaje modular y reproducible utilizando herramientas de código abierto como ROS 2, Gazebo y Stable-Baselines 3, y estableciendo bases sólidas para futuras investigaciones y aplicaciones en exploración, logística, soporte autónomo, entre otros.

Abstract

Reinforcement learning (RL) has driven mobile robotics, where robots learn to navigate complex terrains without predetermined rules or highly detailed prior maps. This master's thesis follows this trend by focusing on the design, training, and validation of a model using the Proximal Policy Optimization (PPO) algorithm for the autonomous control of a ROSbot XL simulated in Gazebo Ignition and ROS 2 Humble. The objective is to train an agent to achieve established goals and avoid collisions while adapting safely and effectively to changing conditions; all of this based solely on the robot's LIDAR and odometry sensor data.

The project's approach lies in creating an observation environment that balances sensory richness and computational efficiency by reducing the LIDAR measurements from 282 dimensions to only 37. This ensures that crucial navigation information is preserved without affecting the speed of the training process. This reduction strategy, along with the definition of the reward function, will allow the agent to prioritize goal alignment and proactively avoid obstacles, rewarding positive behaviors and penalizing actions that put the robot's integrity or its ability to complete the assigned task at risk.

For the training, integration with ROS 2, Gazebo Ignition, and Gymnasium was carried out, establishing a modular environment that allowed the ROSbot to learn in 200,000 training steps in the simulation, with constant metric monitoring through TensorBoard. An MLP architecture was used to efficiently process vector observations, achieving progressive learning in which the agent developed aligned forward maneuvers, collision avoidance, and recovery from getting stuck, consolidating a robust and safe policy.

In the model validation stage, the results showed that the agent successfully achieved various objectives both in familiar scenarios (already seen during training) and in new environments with obstacles. The robot did not rely at any time on memorizing specific routes but instead made real-time decisions based solely on its sensory perceptions. These results also support the usability and applicability of reinforcement learning in robotics scenarios in simulation for the subsequent transfer of learned policies to real environments.

Finally, this master's thesis presents a practical and academic contribution in the field of mobile robotics, showing how reinforcement learning can be applied to create agents capable of progressively, safely, and autonomously navigating complex spaces, presenting a modular and reproducible learning process using open-source tools such as ROS 2, Gazebo, and Stable-Baselines 3, and establishing solid foundations for future research and applications in exploration, logistics, autonomous support, among others.

Tabla de contenidos

1 Introducción 1
1.1 Objetivo general 1
1.2 Objetivos específicos 2
1.3 Justificación 2
1.4 Estructura del documento 3
2 Estado del Arte 4
2.1 Fundamentos del Aprendizaje por refuerzo 4
2.1.1 Agente 4
2.1.2 Entorno 5
2.1.3 Estado 5
2.1.4 Acción 5
2.1.5 Recompensa 5
2.1.6 Política 6
2.2 Algoritmo PPO 6
2.3 Control de trayectorias en la robótica móvil 8
2.4 Percepción sensorial LIDAR y odometría 9
2.5 Aplicaciones recientes de RL en robótica móvil 10
2.5.1 Navegación reactiva sin mapa (TurtleBot3 – DDPG) 11
2.5.2 Navegación reactiva en entornos dinámicos (Jackal – PPO) 11
2.5.3 Navegación multipropósito en entornos 3D con SAC en Unity 11
2.6 Herramientas y frameworks para simulación y control en la robótica 11
2.6.1 Robot Operating System (ROS) 11
2.6.2 Gazebo 12
2.6.3 Integración de ROS y Gazebo en el aprendizaje por refuerzo 13
2.7 Plataforma robótica: ROSbot XL 14
2.7.1 Visión general de la plataforma 14
2.7.2 Capacidades del ROSbot para aplicaciones de RL 15
2.8 Arquitecturas neuronales en RL para robótica móvil 16
2.8.1 Redes neuronales de perceptrón multicapa (MLP) 16
2.8.2 Redes convolucionales (CNN) 17
2.8.3 Redes recurrentes (RNN y LSTM) 18
2.8.4 Arquitectura hibrida 19
2.9 Métricas de rendimiento en RL 20
2.9.1 Recompensa acumulada 20
2.9.2 Tasa de éxito 20
2.9.3 Números de pasos por episodio 20
2.9.4 Distancia mínima al objetivo 21
2.9.5 Entropía de la política 21

2.9.6 Pérdida de valor y política 21
2.10 Transferencia sim-to-real en aprendizaje por refuerzo 21
2.10.1 El reality gap y sus implicaciones 22
2.10.2 Aplicaciones exitosas de sim-to-real en robótica móvil 22
2.10.3 Relevancia del sim-to-real en RL para robótica 23
3 Metodología y Diseño 24
3.1 Objetivo del entrenamiento y comportamiento 24
3.2 Definición del espacio de observación y acción 25
3.2.1 Variables del entorno y su función en el aprendizaje 28
3.3 Diseño de la función de recompensa 29
3.3.1 Propósito y comportamiento deseado 30
3.3.2 Recompensas 31
3.3.3 Penalizaciones 32
3.3.4 Fórmula 33
3.4 Configuración del modelo PPO 35
3.4.1 Arquitectura de la política 35
3.4.2 Hiperparámetros 35
3.5 Plan de entrenamiento y validación 38
4 Implementación del modelo 39
4.1 Diseño del entorno simulado 39
4.2 Flujo de simulación e integración 39
4.2.1 Configuración del entorno de simulación con Gazebo Ignition 40
4.2.2 Creación de un nodo ROS 2 como interfaz con el entorno 40
4.2.3 Definición del entorno Gym compatible con Stable-Baselines3 40
4.2.4 Entrenamiento del robot con PPO 40
4.2.5 Evaluación del modelo entrenado en simulación 41
4.3 Arquitectura del entorno y sincronización sensorial 41
4.4 Consideraciones técnicas y limitaciones 43
4.4.1 Consideraciones técnicas 43
4.4.2 Limitaciones identificadas 43
4.4.3 Entorno de desarrollo y configuración técnica 43
4.4.4 Características de hardware utilizado 45
5 Resultados y análisis 46
5.1 Evolución del aprendizaje (entrenamiento) 46
5.1.1 Proceso de aprendizaje progresivo 46
5.1.2 Indicadores cuantitativos del progreso 48
5.1.3 Observaciones cualitativas en el simulador 53
5.2 Comportamiento 54
5.2.1 Comportamiento observado en espacios abiertos 54

5.2.2 Comportamiento en entorno con obstáculos 55
5.2.3 Mecanismo de escape en atascos 56
5.3 Análisis de episodios exitosos y fallidos 57
5.3.1 Análisis de episodios exitosos 57
5.3.2 Análisis de episodios fallidos 61
5.3.3 Lecciones aprendidas 67
5.4 Discusión critica 68
6 Conclusiones 69
7 Bibliografía 70
8 Anexos 73
8.1 Configuración del modelo PPO y entrenamiento 73
8.2 Estructura del entorno de entrenamiento 73
8.3 Ejecución y validación del modelo entrenado 75

1 Introducción

La robótica móvil se ha visto muy beneficiada de las técnicas novedosas de inteligencia artificial, donde destaca el aprendizaje por refuerzo (Reinforcement Learning, RL). En comparación con los métodos convencionales de control, donde se definen explícitamente reglas para cada situación, el RL da la oportunidad a un agente para poder entender acciones de comportamiento a partir de su experiencia en el entorno, tomando decisiones en función de las recompensas obtenidas.

El TFM es precisamente en este sentido al plantear el diseño, implementación y evaluación de un modelo de aprendizaje por refuerzo a partir del algoritmo Proximal Policy Optimization (PPO) para el control del movimiento autónomo de un robot móvil simulado en un entorno ROS 2 Humble y Gazebo Ignition. El robot que se utiliza, que cuenta con sensores como LIDAR y odometría, debe aprender a moverse en direcciones de objetivos definidos en el entorno evitando cualquier tipo de obstáculo, sin disponer de un mapa previo ni de reglas codificadas.

La principal motivación del trabajo es comprobar hasta qué punto es posible aplicar las técnicas modernas de aprendizaje por refuerzo a tareas de navegación autónoma usando simuladores de alta fidelidad y sensores realistas que imitan condiciones del mundo real. Además, se ha puesto un gran interés en observar cómo se comporta el agente entrenado ante situaciones críticas como desvíos, bloqueos, colisiones, etc.

Basado en experimentos de entrenamiento y validación, demostramos que el modelo puede aprender políticas de navegación potentes que resultan en un comportamiento intuitivo y coherente en entornos dinámicos. Esta contribución no solo está orientado a la investigación, sino también sentar las bases sólidas para aplicaciones en logística, robótica de servicios, exploración o asistencia autónoma.

1.1 Objetivo general

Desarrollar y entrenar en Proximal Policy Optimization (PPO) un modelo de aprendizaje por refuerzo para controlar un robot móvil simulado que se mueve de manera autónoma y aprende a alcanzar ciertos objetivos mientras evita obstáculos a través de sensores LIDAR y odometría, con ROS 2 y Gazebo.

1.2 Objetivos específicos

Los objetivos específicos son:

i. Crear un espacio de observación utilizando datos del LIDAR y la orientación relativa al objetivo.
ii. Implementar un entorno de entrenamiento en ROS 2 y Gazebo Ignition que sea compatible con PPO.
iii. Definir una función de recompensa que incentive comportamientos deseables como la alineación, el avance y la evasión de colisiones.
iv. Entrenar y evaluar el modelo PPO a través de episodios simulados en diferentes escenarios.

1.3 Justificación

En un mundo cada vez más poblado por máquinas, la exploración de entornos por un robot que pueda moverse y desempeñarse de manera segura por sí mismo es quizás uno de los desafíos y oportunidades más omnipresentes de la IA aplicada. Este tipo de tareas han requerido tradicionalmente una programación manual complicada, mapas finamente ajustados o estrategias de control fijas, todo lo cual ha limitado la flexibilidad y escalabilidad de los sistemas robóticos.

La aplicación del aprendizaje por refuerzo, en este caso el algoritmo PPO, permite soluciones más versátiles y genéricas, ya que el robot puede aprender de experiencias pasadas y no necesita ser monitoreado de cerca ni tener un modelo sofisticado de su entorno. Este aspecto de adaptación autónoma de las políticas está en línea con las necesidades industriales y de investigación actuales, por ejemplo, la asistencia de robots para trabajar en entornos no estructurados que cambian con el tiempo

Además, el uso de los entornos de simulación como ROS 2 y Gazebo Ignition, permiten simular entornos como el mundo real, lo que facilita la transferencia de la política aprendida en la simulación al entorno físico. La naturaleza modular de este trabajo, combinada con un diseño centrado en la observación, la valoración de recompensas y el comportamiento, lo convierte en un punto de partida sólido para futuras investigaciones.

Este TFM no solo cubre un alcance de vanguardia en robótica, sino que también proporciona un caso de uso práctico, reproducible y bien fundamentado sobre cómo integrar RL con herramientas modernas de ROS para abordar problemas de navegación desafiantes.

1.4 Estructura del documento

El Trabajo de Fin de Máster está organizado en ocho capítulos, organizados de manera progresiva para facilitar la solución propuesta y los resultados obtenidos.

Capítulo 1 – Introducción: Presenta el contexto general del trabajo, los objetivos, así como la justificación.

Capítulo 2 – Estado del arte: Define los fundamentos teóricos del aprendizaje por refuerzo, el algoritmo PPO, y componentes claves en la navegación de robots móviles como LIDAR y odometría. Además, se analizan los entornos de simulación utilizados y se incluyen aplicaciones recientes de RL en la robótica móvil.

Capítulo 3 – Metodología: Describe el diseño del entorno de simulación, la construcción del espacio de observación y acción, la función de recompensa, la arquitectura del modelo PPO y el plan de entrenamiento y validación.

Capítulo 4 – Implementación del modelo: Detalla la integración entre ROS 2, Gazebo Ignition y Gymnasium, así como la arquitectura modular del entorno, la publicación de comandos de movimiento y la sincronización sensorial necesaria para el entrenamiento y prueba del agente.

Capítulo 5 – Resultados y análisis: Presenta los resultados obtenidos durante las fases de entrenamiento y validación, incluyendo gráficas, métricas de rendimiento y observaciones del comportamiento del robot. También se discuten los episodios exitosos y fallidos, y se reflexiona sobre la eficacia del modelo entrenado.

Capítulo 6 – Conclusiones: Consolida las contribuciones más relevantes obtenidas del trabajo, sus limitaciones y sugiere direcciones para trabajos futuros, utilizando los resultados obtenidos y la experiencia adquirida a lo largo del trabajo realizado durante el proyecto.

Capítulo 7 – Bibliografía: Incluye todo el material académico, técnico y científico utilizado y referido en el informe con el estilo de citación APA adecuado.

Capítulo 8 – Anexos: Incluye material complementario relevante como fragmentos de código, configuraciones específicas, resultados adicionales y capturas de simulación que respaldan el desarrollo del trabajo y facilitan su reproducción.

2 Estado del Arte

La robótica móvil ha presenciado muchos desarrollos en los últimos años en el contexto de los algoritmos de inteligencia artificial, donde la técnica principal ha sido el aprendizaje por refuerzo. Un agente aprende políticas de comportamiento interactuando con el entorno, por lo que no necesita reglas explicitas ni codificación complicada. Los fundamentos del aprendizaje por refuerzo, el algoritmo PPO, los sensores y los simuladores se describen en este capítulo, así como el uso de estos componentes para el control de trayectorias en la robótica móvil.

2.1 Fundamentos del Aprendizaje por refuerzo

El Aprendizaje por refuerzo (RL), es un paradigma del aprendizaje automático en la que un agente (para nuestro caso será un robot) aprende a tomar decisiones interactuando con un entorno. El agente aprende por prueba y error a maximizar una señal de recompensa que guía su comportamiento hacia un conjunto de objetivos.

Según Sutton y Barto (2018), el RL se basa en los procesos de decisión de Markov (MDPs), que representan el entorno como un conjunto de estados, acciones y recompensas probabilísticas. A diferencia del aprendizaje supervisado, el RL no ofrece supervisión explícita sino una señal de recompensa dispersa y diferida.

En este contexto, el agente busca aprender una política que maximice la suma esperada de recompensas futuras. Este enfoque se ha aplicado con éxito a la navegación de robots, control de brazos manipuladores y el aprendizaje de estrategias en juegos complejos (Russell & Norvig, 2020).

Varios autores destacan que la estructura modular del RL se utiliza para construir sistemas complejos sin reglas explícitas. Por ejemplo, Kaelbling et al. (1996) y Mnih et al. (2015) han demostrado cómo el RL puede ser escalado para controlar entornos continuos y de alta dimensionalidad, lo cual es especialmente relevante en robótica, donde las observaciones suelen provenir de múltiples sensores.

El RL no solo aprende comportamientos óptimos, sino también adaptarse a entornos cambiantes. Esta naturaleza adaptativa es esencial en robótica porque la dinámica del mundo real no siempre es predecible. Según Kober et al. (2013), la flexibilidad del RL lo convierte en una herramienta idónea para la robótica autónoma, al permitir que los robots desarrollen habilidades de forma incremental, mediante prueba y error.

Los componentes principales del RL son:

2.1.1 Agente

El agente es el componente central que aprende, toma decisiones y ejecuta acciones en el entorno, su tarea principal es aprender a seleccionar acciones óptimas que maximicen la recompensa esperada.

Sutton y Barto (2018) describen al agente como una entidad autónoma capaz de percibir su estado actual, interactuar con el entorno y aprender de las consecuencias.

Además, según Kober et al. (2013), el agente puede ser entrenado en simulación y transferido posteriormente al mundo real, siempre que las observaciones y acciones estén correctamente mapeadas. Esto permite experimentar de forma segura y eficiente, minimizando los riesgos que implicaría entrenar directamente en hardware físico.

2.1.2 Entorno

El entorno representa todo aquello con lo que interactúa el agente. En el contexto del RL, el entorno define las dinámicas del sistema: cómo cambian los estados en respuesta a las acciones del agente y qué recompensas se generan. En simulación robótica, el entorno suele estar compuesto por el mapa, los obstáculos, las dinámicas físicas y los sensores simulados.

Según Koenig y Howard (2004), entornos como Gazebo permiten simular con alta fidelidad la interacción de un robot con el mundo físico, incluyendo colisiones, fricción y sensores. Esto resulta esencial para que el agente aprenda comportamientos realistas que puedan generalizar a entornos reales.

2.1.3 Estado

El estado es una representación del conocimiento que tiene el agente sobre la situación actual del entorno. En sistemas robóticos, los estados suelen incluir datos de sensores como LIDAR, cámaras o información de posición.

Según Kaelbling et al. (1996), la elección del estado es crítica, debe ser informativa, estable y computacionalmente manejable. Un buen diseño del estado permite al agente discriminar entre situaciones relevantes y tomar decisiones adecuadas, mientras que un estado mal formulado puede dificultar el aprendizaje o inducir comportamientos erráticos.

2.1.4 Acción

Las acciones son las decisiones que el agente puede tomar en un estado determinado. En entornos de control robótico, las acciones suelen corresponder a comandos motores, como velocidades lineales y angulares. Si un modelo, tiene acciones continuas y bidimensionales, permite una navegación más suave y realista que los entornos con acciones discretas.

Según Lillicrap et al. (2016), el espacio de acción continuo es preferible en tareas de control fino, como la navegación en entornos cerrados, ya que permite al agente optimizar pequeños ajustes en movimiento.

2.1.5 Recompensa

La recompensa es el principal mecanismo de retroalimentación en RL. Después de cada acción, el agente recibe una señal escalar que indica cuán buena fue

esa decisión. La función de recompensa debe ser cuidadosamente diseñada para guiar al agente hacia el comportamiento deseado.

Mnih et al. (2015) destaca que las funciones de recompensa densas (que otorgan feedback frecuente) suelen ser más efectivas que aquellas escasas, ya que permiten al agente aprender más rápidamente.

2.1.6 Política

La política es la estrategia que sigue el agente para seleccionar acciones. Es una función que mapea estados a probabilidades de acciones. En PPO, esta política se representa mediante una red neuronal, entrenada para maximizar la recompensa esperada acumulada.

Sutton y Barto (2018) clasifican las políticas en deterministas (siempre devuelven la misma acción para un estado) y estocásticas (devuelven una distribución). PPO utiliza políticas estocásticas, lo que facilita la exploración y evita caer en mínimos locales.

Además, la política puede ser entrenada de manera directa (como en PPO) o indirecta, mediante estimaciones de valor (como en DQN). Es importante un entrenamiento directo para mantener la arquitectura simple y eficiente.

En tareas multi-agente, como demuestra Berner et al. (2019), el diseño de políticas coordinadas entre agentes es fundamental para lograr objetivos globales. En entornos altamente competitivos como el videojuego Dota 2, es posible escalar políticas mediante entrenamiento distribuido y compartir parámetros entre agentes. Esta estrategia inspira aplicaciones en robótica multi-agente, como la navegación cooperativa o la cobertura de áreas amplias.

2.2 Algoritmo PPO

Uno de los avances más relevantes en el campo del aprendizaje por refuerzo ha sido la introducción del algoritmo Proximal Policy Optimization (PPO), diseñado por OpenAI como una alternativa eficiente y estable a otros métodos de optimización de políticas como TRPO (Trust Region Policy Optimization) (Schulman et al., 2017). PPO se ha convertido en un estándar en la comunidad de aprendizaje por refuerzo, ya que ofrece un buen compromiso entre simplicidad, rendimiento y estabilidad, y es particularmente adecuado para tareas basadas en control continuo, como la navegación robótica.

PPO sigue la política de aprendizaje por política directa (policy gradient), en la cual el objetivo es aprender una política con parámetros de redes neuronales para obtener la recompensa acumulativa esperada. Emplea la función de pérdida con clip para evitar que la política aprendida cambie demasiado en la actualización. Esta restricción evita que el nuevo comportamiento del agente se desvíe excesivamente del comportamiento anterior, lo que podría causar inestabilidad o hacer olvidar políticas efectivas previas.

El algoritmo ha demostrado empíricamente tener éxito en entornos de simulación como MuJoCo cuando se aplica frente a otros algoritmos en el aprendizaje por refuerzo profundo como TRPO, A2C o CEM. En la Figura 1, se muestra que PPO supera al resto de algoritmos en todas las tareas de locomoción continua, destacando que PPO puede aprender políticas estables con menos pasos de entrenamiento.

Figura 1

Comparación de algoritmos en varios entornos MuJoCo

Nota. Adaptado de Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). Proximal Policy Optimization Algorithms. arXiv preprint arXiv:1707.06347.

El término *proximal* hace referencia precisamente a la idea de actualizar la política sin alejarse mucho de la versión anterior. Es decir, esto se implementa mediante una penalización que actúa sobre la ratio de probabilidad entre la política nueva y la antigua. Este enfoque ha demostrado ser altamente efectivo en escenarios de alta dimensionalidad, con entornos ruidosos y acciones continuas, como en el caso del robot ROSbot XL que navega en un entorno con obstáculos dinámicos.

En aplicaciones reales, PPO ha sido utilizado para entrenar agentes en juegos complejos como Dota 2 (Berner et al., 2019), como se ha demostrado en trabajos como los de OpenAI, PPO ha logrado controlar brazos robóticos, locomoción bípeda, y navegación móvil sin necesidad de modelos cinemáticos explícitos.

En entornos de control robótico continuo, PPO resulta particularmente efectivo al trabajar con espacios de acción multidimensionales y continuos (velocidad lineal y angular). Además, su compatibilidad con marcos como Stable-Baselines3 facilita su integración con simuladores como Gazebo, permitiendo experimentar con diferentes configuraciones sin requerir arquitecturas complejas.

2.3 Control de trayectorias en la robótica móvil

El control de trayectorias en robótica móvil es una disciplina que busca diseñar algoritmos capaces de guiar a un robot desde un punto inicial hasta un destino, evitando colisiones y optimizando variables como el tiempo, la distancia y el consumo energético. Tradicionalmente, esta tarea se ha abordado mediante planificadores cinemáticos o geométricos, como A*, Dijkstra, o Dynamic Window Approach (DWA), los cuales requieren modelos explícitos del entorno y de la dinámica del robot (LaValle, 2006).

Sin embargo, en entornos dinámicos, los enfoques tradicionales de replaneamiento pueden resultar ineficientes, generando trayectorias erráticas o incluso bloqueos del robot frente a obstáculos impredecibles. Wang et al. (2020) describen este fenómeno como el "problema del robot congelado" y proponen integrar guías globales con planeadores locales basados en RL, lo que permite un comportamiento más robusto y fluido en escenarios cambiantes.

El control de trayectoria mediante RL se basa en la formulación del problema como una secuencia de decisiones: el robot observa su entorno, ejecuta una acción (como avanzar o girar), y recibe una recompensa en función de su progreso hacia el objetivo y su seguridad. Así, el agente aprende no solo a llegar al destino, sino a hacerlo de manera eficiente y segura, penalizando colisiones, estancamientos o desalineaciones.

En investigaciones recientes, autores como Tai et al. (2017) han demostrado que es posible entrenar agentes RL capaces de navegar en entornos desconocidos sin necesidad de un mapa global, utilizando únicamente datos sensoriales como el LIDAR. Este enfoque, denominado navegación "mapless", ha permitido simplificar el pipeline de navegación robótica y facilitar la transferencia a entornos reales.

Además, el control mediante RL puede adaptarse fácilmente a objetivos múltiples, entornos cambiantes o incluso tareas de seguimiento de trayectorias dinámicas. En trabajos como los de Kiran et al. (2022), se han entrenado agentes para seguir trayectorias predefinidas, evitando obstáculos que aparecen durante el movimiento, lo que representa una evolución respecto a los métodos estáticos.

2.4 Percepción sensorial LIDAR y odometría

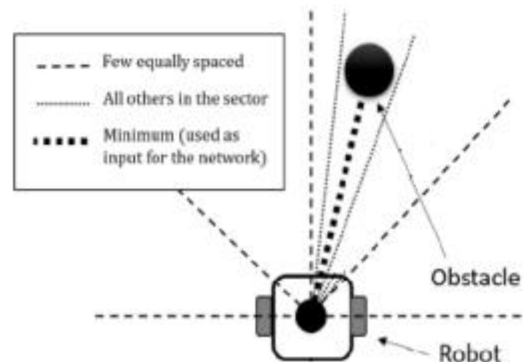
En el campo de la robótica móvil, la capacidad de un agente para percibir su entorno es un factor importante para su autonomía y rendimiento. Las dos fuentes más comunes de información son el LIDAR (Light Detection and Ranging) y la odometría. Estas herramientas permiten construir una representación del mundo circundante y del propio estado del robot, importantes para tomar decisiones.

El sensor LIDAR emite pulsos láser y mide el tiempo que tarda en recibir la señal reflejada, permitiendo estimar con gran precisión la distancia a objetos en todas las direcciones. Este tipo de sensor proporciona datos espaciales en tiempo real, incluso en ausencia de luz o con condiciones visuales adversas, siendo especialmente valioso para navegación y detección de obstáculos (Zhang & Singh, 2014).

En su diseño de observación, los autores clasifican los 30 valores de LIDAR en 10 grupos de 3 medidas, de las cuales seleccionan el mínimo de cada uno. Este método reduce la dimensión del espacio y maneja los obstáculos más cercanos (Taheri et al., 2024). Este enfoque se aprecia en la Figura 2, donde se describe el proceso de simplificación de los datos LIDAR para crear los bloques frontales, empleados como entrada directa a la red neuronal del agente.

Figura 2

Agrupación de medidas LIDAR en bloques frontales



Nota. Adaptado de Taheri, H., Hosseini, S. R., & Nekoui, M. A. (2024). Deep Reinforcement Learning with Enhanced PPO for Safe Mobile Robot Navigation. arXiv preprint arXiv:2405.16266.

En el contexto de aprendizaje por refuerzo, el LIDAR se ha convertido en una herramienta clave para definir el estado del agente. Autores como Tai et al. (2017) han demostrado que un agente puede aprender a navegar sin mapas, utilizando únicamente las medidas LIDAR como observaciones. Esta capacidad ha permitido reducir la dependencia de módulos complejos de mapeo y localización.

Por otro lado, la odometría estima el desplazamiento del robot a partir de datos de los encoders de las ruedas, y habitualmente se complementa con una IMU

(Inertial Measurement Unit) para mejorar la estimación de orientación y mitigar errores acumulativos. Aunque la odometría puede verse afectada por errores acumulativos, su combinación de sensores como el LIDAR permite mejorar la calidad de la estimación posicional (Fox et al., 1999). En el caso de aprendizaje por refuerzo, la odometría es una herramienta fundamental para estimar la posición del robot, siendo utilizada como parte del estado del modelo, así como para definir las recompensas y condiciones de éxito.

2.5 Aplicaciones recientes de RL en robótica móvil

En los últimos años, el RL ha demostrado ser una poderosa herramienta para dotar a los robots móviles de habilidades autónomas. En lugar de usar mapas predefinidos o caminos programados, estos robots aprenden ejecutando acciones en su entorno y experimentando retroalimentación sensorial.

Uno de los ejemplos más representativos es el trabajo basado en Chen et al. (2017), en el cual un robot móvil aprende a moverse de manera socialmente consciente en presencia de personas. A diferencia del enfoque reactivo tradicional, esta política aprendida por DRL permite al robot idear soluciones que aseguran el espacio personal y el comportamiento humano.

La figura 3 demuestra que el robot puede maniobrar con flexibilidad en la multitud y ajustar su trayectoria para ser seguro y colaborativo. Esta metodología es particularmente adecuada para tareas de interacción cercana o asistencia en entornos urbanos donde la interacción humano-robot es constante.

Figura 3

Ejemplo de navegación socialmente consciente mediante DRL en un entorno compartido con humanos

Nota. Adaptado de Chen, Y. F., Liu, M., Everett, M., & How, J. P. (2017). Socially Aware Motion Planning with Deep Reinforcement Learning. arXiv preprint arXiv:1703.08862.

2.5.1 Navegación reactiva sin mapa (TurtleBot3 – DDPG)

Uno de los primeros enfoques en aplicar RL a la navegación fue presentado por Tai et al., quienes utilizaron DDPG para entrenar un TurtleBot3 en entornos simulados en Gazebo, sin necesidad de mapas ni localización global. El agente aprendía directamente desde 10 medidas LIDAR seleccionadas y un vector hacia el objetivo. La política aprendida fue capaz de transferirse a la realidad con mínimos ajustes.

Este estudio fue pionero en demostrar la transferencia sim-real, y sentó las bases para usar observaciones sensoriales mínimas como base para una política eficaz.

2.5.2 Navegación reactiva en entornos dinámicos (Jackal – PPO)

En 2024, Taheri, Hosseini y Nekoui presentaron un enfoque con PPO mejorado para navegación segura de un robot móvil en entornos complejos simulados en Gazebo. El agente combinaba percepciones LIDAR con una política neuronal ampliada y una función de recompensa cuidadosamente diseñada: “Collision-free motion is essential for mobile robots … We employ PPO … accompanied by a well-designed reward function” (Taheri et al., 2024, resumen).

2.5.3 Navegación multipropósito en entornos 3D con SAC en Unity

Li y colaboradores (2022) adoptaron Soft Actor–Critic (SAC) para enseñar a un agente virtual a moverse entre múltiples objetivos en un entorno tridimensional generado por Unity ML-Agents. Esta estrategia permite aprender una política que decide no solo cómo navegar, sino también qué destino priorizar.

2.6 Herramientas y frameworks para simulación y control en la robótica

El avance en el desarrollo de algoritmos de aprendizaje por refuerzo aplicados a la robótica móvil ha ido de la mano con el progreso de herramientas y frameworks que permiten simular, controlar y monitorizar el comportamiento de los agentes en entornos virtuales de alta fidelidad antes de transferirlos a plataformas físicas. Estas herramientas no solo facilitan la implementación modular de arquitecturas de control y percepción, sino que también promueven la reproducibilidad y escalabilidad de las investigaciones en robótica (Quigley et al., 2009; Macenski et al., 2022).

2.6.1 Robot Operating System (ROS)

Robot Operating System (ROS) es un framework de código abierto que actúa como un sistema operativo flexible para robots, proporcionando abstracción de hardware, manejo de procesos en paralelo, y un ecosistema de herramientas para visualización y depuración (Quigley et al., 2009). Su arquitectura se basa

en un sistema de publicación-suscripción mediante tópicos, servicios y acciones, que facilita la construcción de sistemas distribuidos en robótica de forma modular y escalable (Macenski et al., 2022).

Con la llegada de ROS 2, se integró DDS (Data Distribution Service) como capa de comunicación (RMW), permitiendo la personalización del canal de datos según las necesidades de latencia, fiabilidad o rendimiento en sistemas robóticos en tiempo real (Maruyama et al., 2016). Entre las principales características que aporta ROS al ecosistema de robótica y aprendizaje por refuerzo destacan:

i. Abstracción y normalización de mensajes de sensores y actuadores.
ii. Integración con herramientas de visualización como RViz y rqt para depuración y supervisión en tiempo real.
iii. Gestión de calidad de servicio (QoS) para garantizar sincronización sensorial, fundamental en el entrenamiento de agentes con RL.
iv. Facilidad de integración con simuladores y frameworks de aprendizaje profundo.

2.6.2 Gazebo

Gazebo es un simulador de código abierto que permite modelar de forma realista entornos tridimensionales, simulando dinámicas físicas, sensores y robots con alta fidelidad. Desde su creación, se ha convertido en un estándar en investigación robótica por su integración directa con ROS, permitiendo a los agentes recibir información sensorial y ejecutar comandos de control de forma idéntica a la de un entorno físico (Koenig & Howard, 2004).

Gazebo incluye:

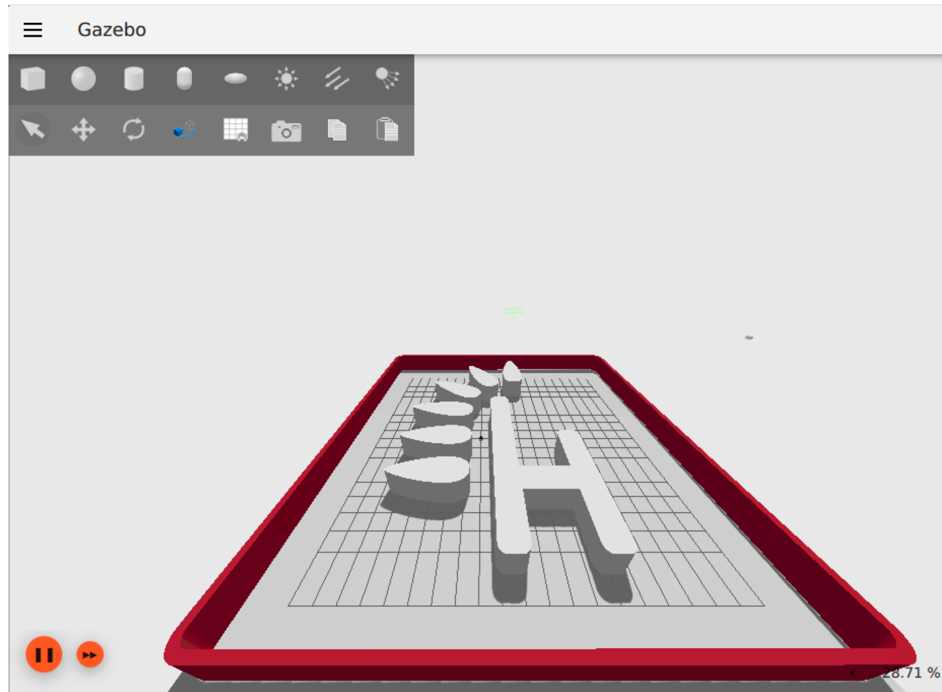
- Simulación de dinámicas físicas (colisiones, fricción, fuerzas).
- Soporte de sensores como LIDAR, cámaras, GPS e IMU.
- Capacidad de definir entornos con obstáculos, iluminación variable y elementos dinámicos.

Gracias a estas capacidades, Gazebo ha sido utilizado para entrenar y validar políticas de aprendizaje por refuerzo antes de su implementación en robots físicos, permitiendo realizar iteraciones de entrenamiento seguras y controladas (Zhu et al., 2020). En la figura 4, se muestra el entorno del simulador.

Un ejemplo relevante es el trabajo de Lin et al. (2020), donde se emplearon ROS 2 y Gazebo para evaluar políticas de RL en mapas con obstáculos y configuraciones diversas, demostrando que la integración de ambos frameworks facilita la experimentación sistemática y la evaluación cuantitativa del desempeño de los agentes entrenados.

Figura 4

Interfaz gráfica de Gazebo



Nota. Interfaz del simulador Gazebo Ignition, con un robot ROSbot XL y el mapa virtual por defecto de Husarion.

2.6.3 Integración de ROS y Gazebo en el aprendizaje por refuerzo

La combinación de ROS y Gazebo se ha convertido en un estándar para el entrenamiento y validación de agentes basados en aprendizaje por refuerzo, particularmente en robótica móvil. Esta integración permite:

i. Definir entornos de entrenamiento realistas y personalizables.
ii. Obtener datos sensoriales con ruido controlado y variabilidad de escenarios.
iii. Probar políticas de control en tiempo real sin riesgo físico.
iv. Facilitar la transferencia de políticas entrenadas en simulación al mundo real (sim-to-real transfer), reduciendo la brecha entre el entrenamiento y la implementación práctica (Sadeghi & Levine, 2017).

Como se muestra en la Figura 5, la integración de ROS y Gazebo permite estructurar un flujo de simulación y control eficiente para el entrenamiento de agentes de aprendizaje por refuerzo

Figura 5

Flujo de integración de ROS y Gazebo en el entrenamiento de agentes de aprendizaje por refuerzo en robótica móvil.

Nota: Elaboración propia basada en el esquema conceptual de integración descrito por Macenski et al. (2022) y Koenig & Howard (2004).

2.7 Plataforma robótica: ROSbot XL

El ROSbot XL es una plataforma robótica de código abierto diseñada por la compañía Husarion, orientada a investigación y desarrollo en robótica móvil autónoma y aplicaciones avanzadas de inteligencia artificial. Su arquitectura, basada en ROS 2 Humble y Ubuntu 22.04, lo convierte en una opción sólida para experimentación en aprendizaje por refuerzo, permitiendo el desarrollo de sistemas de navegación, percepción y planificación de trayectorias en entornos reales y simulados de manera eficiente (Husarion, 2024).

El ROSbot XL combina robustez física con capacidades de computación en borde, facilitando el despliegue de modelos entrenados en simulación, integrando sensores de calidad, y asegurando compatibilidad con herramientas estándar en investigación robótica como Gazebo, RViz, ROS 2, y frameworks de RL.

2.7.1 Visión general de la plataforma

El ROSbot XL es un robot de cuatro ruedas con tracción diferencial, diseñado para navegar de forma autónoma en entornos interiores y exteriores. Su chasis resistente de aluminio permite soportar operaciones prolongadas, mientras que su sistema de suspensión y neumáticos de goma permiten un desplazamiento

estable en superficies irregulares. Además, cuenta con opciones de configuración que incluyen Raspberry Pi 4, Intel NUC o NVIDIA Jetson Orin Nano, adaptándose a distintas necesidades de procesamiento y aplicaciones específica, en la Figura 6 se muestra la versión del ROSbot XL

Figura 6

ROSbot XL con LIDAR frontal y plataforma modulable

Nota: Imagen adaptada de Husarion. (s.f.). Manual ROSbot XL – Overview. Recuperado de https://husarion.com/manuals/rosbot-xl/overview/

Entre sus principales componentes, incluye:

i. Sistema operativo Ubuntu 22.04 con ROS 2 Humble, garantizando compatibilidad con nodos de control, navegación y percepción.
ii. LIDAR 2D (RPLIDAR A3 o Slamtec S2) con un rango de hasta 25 m y cobertura de 360°.
iii. IMU (unidad de medición inercial) para estimación de orientación y aceleración.
iv. Odómetro basado en encoders para estimación de desplazamiento lineal y angular.
v. Batería Li-Ion de larga duración (6S, 10 Ah) que proporciona entre 2 y 4 horas de autonomía según carga y condiciones de operación.

2.7.2 Capacidades del ROSbot para aplicaciones de RL

El diseño técnico del ROSbot XL encaja con las necesidades prácticas de los algoritmos de aprendizaje por refuerzo, especialmente el PPO. Su configuración física, sensorial y de control lo convierte en una plataforma versátil tanto para simulación como para despliegue real.

Entre las capacidades más relevantes para el aprendizaje por refuerzo destacan:

i. Compatibilidad con entornos de simulación: Su integración con ROS 2 y Gazebo Ignition facilita la creación de entornos simulados donde entrenar políticas de RL de forma segura y reproducible antes de transferirlas al robot físico, implementando estrategias de sim-to-real.
ii. Percepción de alta calidad: El LIDAR 2D de 360°, combinado con odometría e IMU, permite al ROSbot XL generar observaciones ricas que facilitan el entrenamiento de agentes RL para tareas de navegación autónoma, evitando colisiones y optimizando rutas en tiempo real.
iii. Ecosistema abierto y mantenible: La plataforma utiliza Ubuntu y ROS 2, permitiendo la integración con frameworks como Gymnasium y Stable-Baselines3 para entornos RL, así como herramientas de visualización como RViz y ROSbag para el análisis de datos de entrenamiento y test.
iv. Relevancia en investigación: El ROSbot XL ha sido utilizado en diversos proyectos académicos y de investigación para evaluar modelos de RL en navegación autónoma, exploración de entornos desconocidos y seguimiento de trayectorias, permitiendo comparar resultados entre simulación y mundo real.

2.8 Arquitecturas neuronales en RL para robótica móvil

El aprendizaje por refuerzo profundo (Deep Reinforcement Learning, DRL) surge de la integración del aprendizaje por refuerzo clásico (RL) con redes neuronales profundas, permitiendo abordar tareas de control en entornos complejos y con espacios de estados de alta dimensionalidad, como sucede en la robótica móvil (Arulkumaran et al., 2017).

A diferencia del RL tradicional, que depende de tablas de valores o aproximadores lineales, el DRL permite utilizar redes neuronales como aproximadores de funciones de valor, políticas o ambas, facilitando que los agentes aprendan directamente a partir de observaciones sensoriales complejas como imágenes, datos LIDAR o señales multivariadas (Mnih et al., 2015).

2.8.1 Redes neuronales de perceptrón multicapa (MLP)

Las redes neuronales de perceptrón multicapa (MLP) son modelos de aprendizaje profundo ampliamente utilizados en aprendizaje por refuerzo, especialmente cuando las observaciones se representan como vectores numéricos (como distancias, ángulos o velocidades). Un MLP está compuesto por una capa de entrada, una o más capas ocultas con funciones de activación no lineales, y una capa de salida. Gracias a esta estructura, pueden aproximar funciones complejas y modelar de forma efectiva la relación entre estados y acciones dentro de un entorno (Goodfellow et al., 2016; Bishop, 2006).

En RL, estas redes se usan principalmente para representar funciones de valor o políticas, siendo una arquitectura robusta y de bajo coste computacional cuando el espacio de observación no requiere procesamientos espaciales complejos como en imágenes (Arulkumaran et al., 2017). Por ejemplo, en algoritmos como Deep Q-Network (DQN) (Mnih et al., 2015) y Deep Deterministic Policy Gradient (DDPG) (Lillicrap et al., 2016), los MLP permiten al agente

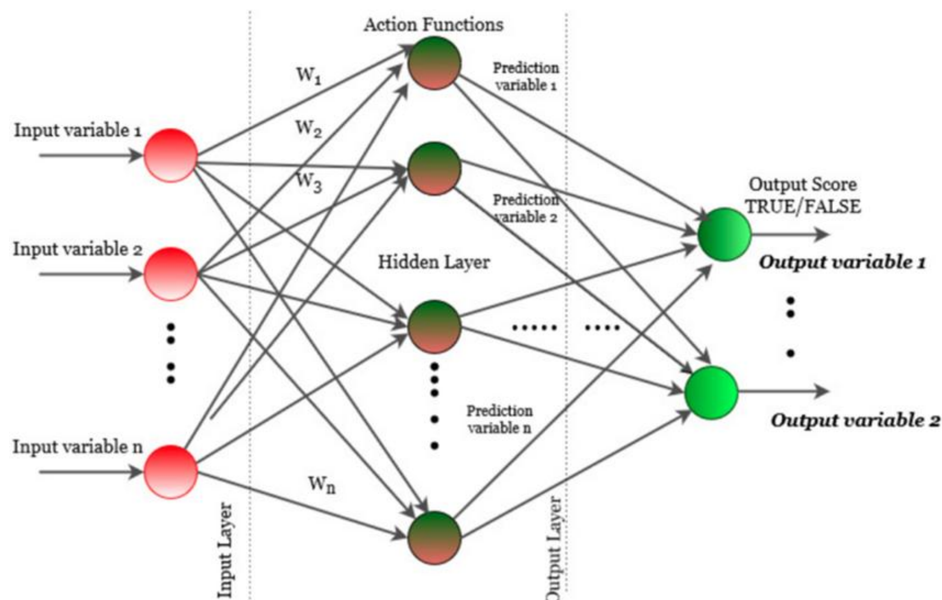
aprender a partir de observaciones vectoriales de forma eficiente, contribuyendo de manera esencial a la toma de decisiones en entornos de control continuo y discreto (Mousavi et al., 2018).

En la Figura 7 se muestra de forma clara cómo funciona un perceptrón multicapa (MLP). Se muestra las capas de entrada, varias capas ocultas conectadas entre sí y la capa de salida, representando cómo los datos van pasando de una capa a otra mientras se transforman para aprender patrones complejos y generar predicciones de manera progresiva.

En resumen, las redes MLP son una herramienta fundamental en RL, ya que permiten procesar datos numéricos estructurados y aprender representaciones que optimizan la interacción del agente con su entorno.

Figura 7

Arquitectura de un perceptrón multicapa (MLP), mostrando las conexiones entre la capa de entrada, las capas ocultas y la capa de salida



Nota: Adaptado de Jahangeer, N., Ganesan, S., & Begum, A. A. S. (2022). A study on different deep learning algorithms used in deep neural nets: MLP SOM and DBN. Wireless Personal Communications. Recuperado de [ResearchGate]

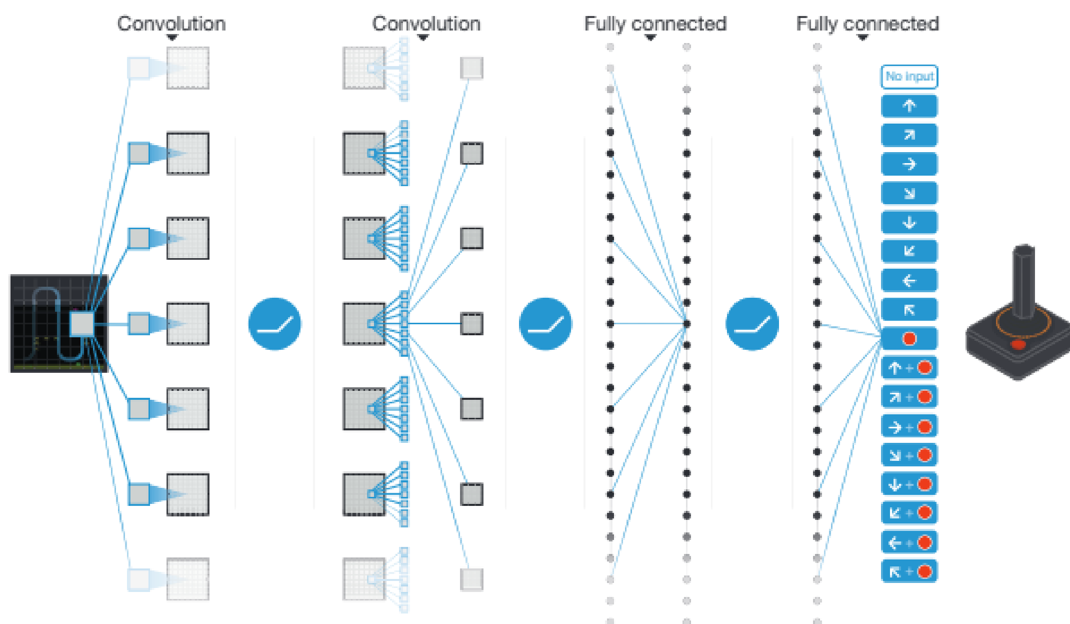
2.8.2 Redes convolucionales (CNN)

Cuando las entradas del agente incluyen imágenes de cámara o mapas de ocupación, las redes convolucionales (CNN) se utilizan en DRL para extraer características espaciales relevantes, permitiendo reducir la dimensionalidad de la información sensorial y manteniendo las relaciones espaciales (Mnih et al., 2015).

Estas redes han demostrado ser efectivas para navegación visual, interpretación de mapas de ocupación y para la integración de percepción con control de movimiento en robots móviles (Mirowski et al., 2017). En la Figura 8 se muestra cómo las imágenes crudas obtenidas del entorno se procesan mediante múltiples capas convolucionales y totalmente conectadas, permitiendo al agente aprender políticas directamente desde entradas visuales sin necesidad de preprocesamiento manual, demostrando la aplicabilidad de las CNN en DRL para navegación y control en entornos complejos

Figura 8

Arquitectura de red convolucional (CNN)



Nota: Adaptado de Human-level control through deep reinforcement learning (Mnih et al., 2015).

2.8.3 Redes recurrentes (RNN y LSTM)

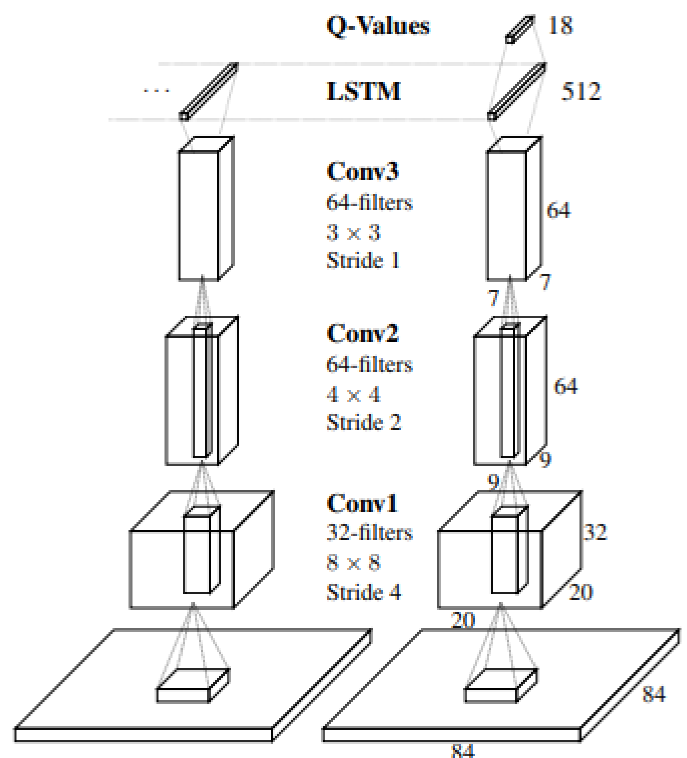
Los entornos robóticos suelen ser parcialmente observables debido a limitaciones sensoriales o la presencia de obstáculos que ocultan partes del entorno. Para abordar este desafío, se utilizan redes neuronales recurrentes (RNN) y, más específicamente, Long Short-Term Memory (LSTM), que permiten al agente mantener una memoria de observaciones pasadas, generando una representación interna del estado actual (Hausknecht & Stone, 2015).

El uso de redes recurrentes mejora la capacidad del agente para actuar en entornos dinámicos, aprendiendo a inferir la posición de objetivos u obstáculos incluso cuando estos no son visibles de manera constante. Esto es relevante en la navegación móvil donde las decisiones dependen no solo de la observación actual, sino también del historial reciente (Zhu et al., 2017).

En la Figura 9 se muestra la arquitectura de redes neuronales recurrentes utilizada para el aprendizaje por refuerzo profundo en entornos parcialmente observables. La RNN integra capas convolucionales para la extracción de características espaciales y una capa LSTM que permite al agente mantener memoria de observaciones pasadas, facilitando el aprendizaje de políticas que requieren integración temporal en escenarios con información parcial.

Figura 9

Arquitectura de RNN con capas convolucionales y LSTM para DRL



Nota: Adaptado de Deep Recurrent Q-Learning for Partially Observable MDPs (Hausknecht & Stone, 2015)

2.8.4 Arquitectura hibrida

En aplicaciones avanzadas, se han explorado arquitecturas híbridas que combinan CNN + LSTM, permitiendo al agente extraer características espaciales de imágenes y mantener un historial de observaciones en entornos complejos. Estas combinaciones se aplican en tareas como exploración, mapeo simultáneo (SLAM) asistido por RL o navegación visual guiada por objetivos (Gupta et al., 2017).

2.9 Métricas de rendimiento en RL

Medir el rendimiento de un agente en aprendizaje por refuerzo (RL) es esencial no solo para validar la efectividad de la política aprendida, sino también para garantizar que el comportamiento sea eficiente, seguro y generalizable. Las métricas de rendimiento en RL permiten identificar tendencias de aprendizaje, cuellos de botella, posibles problemas de sobreajuste y comparaciones objetivas entre algoritmos y configuraciones de entrenamiento (Henderson et al., 2018).

El uso de métricas de rendimiento bien definidas no solo facilita la comparación entre distintos algoritmos y configuraciones, sino que también permite identificar posibles problemas durante el entrenamiento, como estancamientos, sobreajuste, o falta de exploración (Henderson et al., 2018).

2.9.1 Recompensa acumulada

La métrica más utilizada en RL es la recompensa acumulada por episodio, que refleja el desempeño del agente al sumar todas las recompensas recibidas durante la ejecución de un episodio. Un aumento sostenido de esta métrica suele indicar que el agente está aprendiendo a completar la tarea de manera más eficiente (Mnih et al., 2015). Matemáticamente, se define como:

$$R_{total} = \sum_{t=0}^{T} r_t$$

Donde r_t es la recompensa obtenida en el paso t y T es el número total de pasos en el episodio. Un aumento consistente R_{total} a lo largo de los episodios generalmente indica progreso en el aprendizaje de la política.

2.9.2 Tasa de éxito

La tasa de éxito mide el porcentaje de episodios en los que el agente logra completar la tarea definida, por ejemplo, alcanzar un objetivo sin colisionar. Esta métrica es especialmente relevante en entornos de navegación, donde no solo importa avanzar, sino hacerlo de forma segura y eficiente (Tai et al., 2017).

2.9.3 Números de pasos por episodio

El número de pasos por episodio mide la eficiencia temporal del agente, indicando cuántas acciones toma antes de completar una tarea con éxito. Una política óptima debería minimizar esta métrica mientras mantiene una alta tasa de éxito.

En investigaciones como la de Lillicrap et al. (2015), se ha demostrado que un descenso en esta métrica, acompañado de una tasa de éxito estable, indica un aprendizaje de trayectorias más eficientes.

2.9.4 Distancia mínima al objetivo

Cuando el agente no logra alcanzar el objetivo, se utiliza la distancia mínima al objetivo durante el episodio como métrica de evaluación. Esta se define como:

$$d_{min} = min_t \quad || p_t - p_{goal} ||$$

Donde p_t es la posición del agente en el tiempo t, p_{goal} es la posición del objetivo. Esta métrica se ha utilizado, por ejemplo, en trabajos como el de Gupta et al. (2017) para evaluar el grado de aproximación del agente en escenarios complejos de navegación.

2.9.5 Entropía de la política

La entropía de la política mide el nivel de exploración que mantiene el agente durante el entrenamiento. Una entropía alta indica que el agente sigue explorando acciones, mientras que una entropía baja puede reflejar que el agente ha convergido a una política más determinista, aunque un descenso prematuro de la entropía puede ser un indicativo de sobreajuste (Schulman et al., 2017).

2.9.6 Pérdida de valor y política

Durante el entrenamiento con algoritmos como PPO o DDPG, se monitorean las curvas de pérdida del valor estimado y de la política, permitiendo identificar si existen divergencias, oscilaciones o estancamientos en el proceso de aprendizaje (Lillicrap et al., 2015).

En síntesis, el uso de métricas de rendimiento en aprendizaje por refuerzo no se limita al seguimiento de la recompensa acumulada, sino que incluye un conjunto de indicadores que permiten evaluar la calidad, estabilidad y eficiencia del comportamiento del agente durante y después del entrenamiento. Estas métricas forman parte esencial del proceso de evaluación y análisis en trabajos de investigación y desarrollo que emplean RL, proporcionando evidencia objetiva sobre la efectividad del modelo.

2.10 Transferencia sim-to-real en aprendizaje por refuerzo

Uno de los desafíos más relevantes en el campo del aprendizaje por refuerzo aplicado a la robótica móvil es la transferencia de políticas entrenadas en simulación hacia entornos reales, un proceso conocido como sim-to-real transfer. Entrenar agentes en simuladores como Gazebo es seguro, reproducible y eficiente en términos de tiempo, pero las diferencias entre el mundo simulado y el mundo real, denominadas reality gap, pueden limitar la efectividad de las políticas cuando se despliegan en hardware físico (Tobin et al., 2017).

2.10.1 El reality gap y sus implicaciones

El reality gap surge debido a discrepancias en la dinámica física (fricción, colisiones, latencias de sensores), diferencias en la percepción (ruido en LIDAR, iluminación en cámaras) y variaciones en la interacción entre el robot y el entorno. Estas discrepancias pueden provocar que una política que funcione correctamente en simulación falle o tenga un rendimiento subóptimo en la realidad.

Un factor crítico dentro del reality gap es el modelado imperfecto de sensores y actuadores. Por ejemplo, los sensores en simulación generan datos con ruido controlado y predecible, mientras que, en el mundo real, las lecturas de LIDAR o cámaras pueden verse afectadas por condiciones de iluminación, superficies reflectantes y oclusiones dinámicas. Del mismo modo, las simulaciones de motores y controladores de velocidad no siempre replican de forma precisa las fluctuaciones de potencia o los micro desplazamientos que ocurren en entornos reales, impactando en tareas de navegación que requieren precisión en las trayectorias.

El reality gap también se ve amplificado por la diferencia en las condiciones ambientales y de interacción con el entorno. Mientras que en simulación las condiciones como la iluminación, la textura de los suelos y las propiedades de fricción suelen ser constantes o ideales, en el mundo real estos factores pueden cambiar dinámicamente, introduciendo incertidumbres que afectan directamente la ejecución de la política aprendida (Sadeghi & Levine, 2017).

2.10.2 Aplicaciones exitosas de sim-to-real en robótica móvil

Uno de los ejemplos más emblemáticos del uso de domain randomization para la transferencia sim-to-real lo presentó Tobin et al. (2017), quienes entrenaron un brazo robótico en un entorno simulado con variaciones aleatorias en color, iluminación, posición de cámara y textura de fondo. Gracias a estas variaciones, la política aprendida en simulación fue capaz de detectar y localizar objetos reales con una precisión de hasta el 97% sin requerir reentrenamiento en el mundo físico, demostrando la efectividad de este enfoque en tareas de percepción visual complejas.

De manera similar, Sadeghi y Levine (2017) aplicaron el entrenamiento de un dron para navegación en interiores utilizando imágenes sintéticas generadas en simulación. Para ello, emplearon un pipeline de entrenamiento con imágenes fotorrealistas renderizadas con variaciones de textura, iluminación y geometría del entorno, permitiendo que la política aprendida se transfiriera directamente al hardware real. El dron logró navegar en interiores evitando colisiones y manteniendo estabilidad en vuelos prolongados, todo sin utilizar una sola imagen real en la fase de entrenamiento (CAD2RL).

Hwangbo et al. (2019) utilizaron un simulador de alta fidelidad para entrenar un robot cuadrúpedo a ejecutar caminatas ágiles y saltos dinámicos. Para reducir el reality gap, modelaron con precisión las dinámicas de contacto, fricción y flexibilidad de las patas, logrando transferir políticas que permitieron al robot saltar obstáculos y desplazarse sobre terrenos irregulares de forma estable. Los resultados mostraron que políticas entrenadas en simulación

podían transferirse con un rendimiento robusto, reduciendo el coste de pruebas físicas y acelerando el desarrollo de locomoción avanzada.

2.10.3 Relevancia del sim-to-real en RL para robótica

La transferencia sim-to-real ha demostrado ser un componente esencial para la aplicabilidad del aprendizaje por refuerzo en robótica, permitiendo que las políticas aprendidas en entornos controlados se utilicen de manera segura y eficiente en sistemas físicos. Gracias a esta técnica, es posible reducir drásticamente los costos asociados al entrenamiento en hardware real, que incluye no solo el tiempo y desgaste del robot, sino también la necesidad de entornos de prueba seguros y controlados que pueden no estar disponibles para todas las instituciones de investigación.

La transferencia sim-to-real permite validar de forma efectiva el rendimiento de algoritmos de RL en escenarios prácticos, asegurando que el comportamiento del agente no solo sea eficiente en términos de recompensa acumulada, sino también seguro y confiable en condiciones no ideales (Tobin et al., 2017). En el contexto de la robótica móvil, esto es vital para garantizar que los robots puedan tomar decisiones autónomas mientras evitan colisiones, se adaptan a obstáculos dinámicos y operan en espacios desconocidos.

Finalmente, la integración de pipelines de sim-to-real en sistemas de desarrollo de RL en robótica ha impulsado avances en la implementación de comportamientos autónomos complejos, como la navegación en entornos interiores y exteriores, la exploración de espacios desconocidos y la colaboración en entornos multi-robot (Sadeghi & Levine, 2017). Gracias a estas técnicas, el aprendizaje por refuerzo se consolida como una herramienta poderosa y práctica para el desarrollo de sistemas robóticos autónomos con capacidades de adaptación y aprendizaje continuo.

3 Metodología y Diseño

En este capítulo se describe la metodología empleada para el diseño, configuración y planificación del entrenamiento del modelo. Se detalla la definición del espacio de observación y acción, la formulación de la función de recompensa y la configuración de los hiperparámetros del modelo, fundamentando cada decisión con criterios técnicos. Finalmente, se presenta un plan de entrenamiento y validación para la versión final del modelo (v4).

3.1 Objetivo del entrenamiento y comportamiento

El objetivo del entrenamiento es que el agente (ROSbot XL), representado por un modelo PPO, aprenda una política de control que le permita desplazarse hacia un objetivo predefinido evitando colisiones. Esta tarea se desarrolla sin mapas, reglas programadas ni planificación global, únicamente a partir de la percepción sensorial (LIDAR) y la odometría local.

El comportamiento que se busca entrenar se puede desglosar en los siguientes aspectos:

1. Orientación inteligente hacia el objetivo: el robot debe detectar si el objetivo está a su izquierda o derecha y girar progresivamente hasta alinearse.
2. Avance seguro: una vez orientado, debe avanzar en línea recta, pero deteniéndose si detecta un obstáculo.
3. Evasión de obstáculos: en presencia de un obstáculo frontal, el robot debe detenerse, retroceder si es necesario y buscar un nuevo ángulo de aproximación.
4. Autonomía en decisiones locales: debe poder desbloquearse si queda atascado, evitando repeticiones de acciones sin resultado.
5. Comprobación de éxito: debe reconocer que ha alcanzado el objetivo al reducir la distancia cuadrada por debajo de un umbral definido en la parametrización (GOAL_REACHED_THRESHOLD).

Mientras que en el apartado 2.5 se revisaron aplicaciones recientes de RL en robótica móvil utilizando algoritmos como DDPG, SAC y PPO en plataformas TurtleBot3, Jackal y entornos Unity, el presente trabajo difiere en el uso de:

i. Un ROSbot XL con un LIDAR de 360° y odometría.
ii. Un espacio de observación comprimido para eficiencia computacional.
iii. El uso de PPO en entornos simulados con ROS 2 y Gazebo Ignition específicamente optimizados para tareas de navegación, priorizando la evasión de colisiones y la orientación hacia el objetivo.
iv. Un entorno realista con obstáculos y ruidos sensoriales para la validación de políticas.

Estas diferencias refuerzan la aplicabilidad del modelo para su futura transferencia sim-to-real, consolidando su relevancia para navegación autónoma en entornos de investigación.

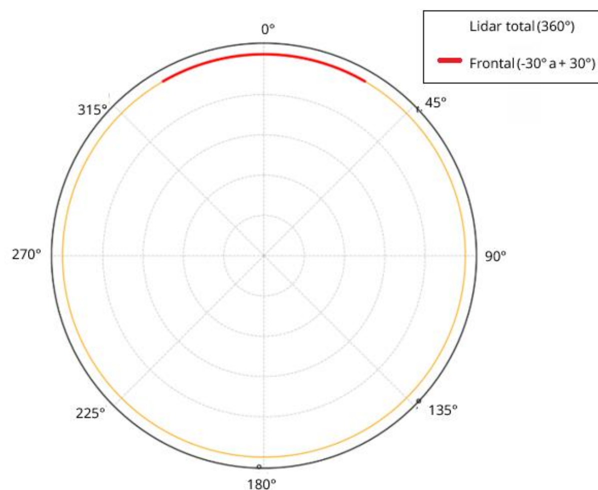
3.2 Definición del espacio de observación y acción

Uno de los desafíos en el aprendizaje por refuerzo aplicado a la robótica es la correcta definición del espacio de observación. Una representación del estado demasiado compleja puede generar altos costes computacionales y dificultar la convergencia, una demasiado simplificada puede inducir pérdida de información crítica. En este trabajo, se optó por un diseño equilibrado entre expresividad, eficiencia y sensibilidad a obstáculos.

En versiones iniciales del modelo, el vector de observación incluía 240 medidas LIDAR frontales (desde –30° a +30°) y 40 medidas laterales distribuidas cada 9°, cubriendo el resto del entorno. A esto se sumaban dos variables adicionales: la orientación angular (ángulo relativo al objetivo) y la distancia al mismo. Este diseño generaba un vector total de 282 dimensiones, como se muestra en la Figura 10.

Figura 10

Campo de visión LIDAR del ROSbot XL



Nota. El gráfico muestra la cobertura completa de 360° del sensor LIDAR. Se resalta el sector frontal (de -30° a +30°), que es el más crítico para evitar colisiones y guiar el movimiento del robot.

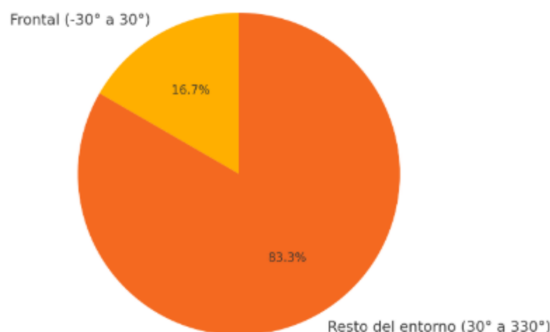
Aunque este espacio de observación era rico en información, su alta dimensionalidad impactaba negativamente en el rendimiento: el tiempo de entrenamiento superaba las 4 horas por cada 10.000 pasos, y el comportamiento del agente mostraba lentitud y sobreajuste.

Como solución, se implementó un proceso de reducción basado en agrupamiento por bloques. Las 240 medidas frontales se dividieron en 30 grupos de 8 valores, seleccionando el mínimo de cada bloque como representación. De igual forma, las 40 medidas laterales se resumieron en 5 bloques. Esta transformación redujo el vector sensorial LIDAR de 280 a tan solo 35 valores.

La Figura 11 muestra la proporción visual entre el sector frontal priorizado y el resto del campo de visión.

Figura 11

Distribución del campo visual del LIDAR del ROSbot XL



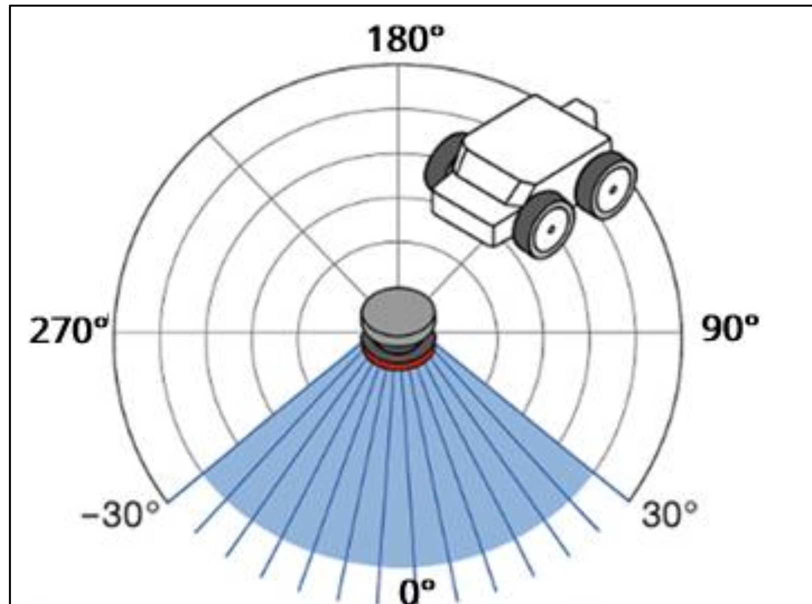
Nota. El sector frontal (–30° a +30°) representa solo el 16.7% del campo visual total, pero contiene el 85% de la resolución sensorial seleccionada tras el agrupamiento. Esto justifica su prioridad en el diseño del entorno.

La reducción a 35 valores no solo mejora la eficiencia, sino que conserva la capacidad de detección de obstáculos críticos, ya que se conservaba el valor mínimo por bloque, actuando como un eficaz detector de proximidad.

Sumando los dos valores adicionales (ángulo y distancia al objetivo), el vector final queda definido con 37 medidas. Este nuevo diseño permitió completar 200.000 pasos de entrenamiento en aproximadamente 7 horas, con mejoras notables en la convergencia y menor sobreajuste. La Figura 12 muestra la representación visual del escaneo frontal desde el punto de vista físico del robot:

Figura 12

Escaneo frontal del ROSbot XL con 240 medidas LIDAR



Nota. El gráfico muestra el escaneo del sensor frontal directamente sobre el robot. Visualmente refuerza la idea del rango -30° a +30° como el sector de entrada crítica para navegación.

Respecto al espacio de acción, en las versiones preliminares del modelo, se configuro rangos más conservadores con límites de –0.2 m/s a 0.3 m/s en velocidad lineal y –0.5 rad/s a +0.5 rad/s en velocidad angular para evitar movimientos bruscos mientras el agente aprendía a desplazarse.

Sin embargo, en la versión final (v4) del modelo, tras el análisis de comportamientos en el simulador y la visualización de métricas en TensorBoard, se ajustó el espacio de acción a valores que permiten maniobras más decididas y efectivas sin sacrificar estabilidad, el espacio de acción finalmente para la velocidad lineal fue un rango entre –0.2 m/s (retroceso controlado) y 0.35 m/s (avance rápido), para la velocidad angular entre –0.8 rad/s y +0.8 rad/s. Estos cambios permiten que al agente:

- Ejecutar giros más amplios para evasión en entornos complejos.
- Aprovechar oportunidades de avance con mayor velocidad en trayectorias despejadas.
- Romper atascos en menos pasos mediante retrocesos y giros decididos.

3.2.1 Variables del entorno y su función en el aprendizaje

Las variables clave que componen el estado del entorno observado por el agente, junto con su propósito funcional y la justificación de su inclusión dentro del espacio de observación:

1) self.robot_x, self.robot_y

   Objetivo: Representan la posición actual del robot en el plano del entorno simulado.

   Justificación: Estas coordenadas son esenciales para determinar la distancia euclidiana al objetivo y calcular el desplazamiento relativo del robot. Aunque no se usan directamente como observación, son necesarias para calcular otras variables observables como distance2 y angle, que sí forman parte del vector de entrada a la política.

2) self.robot_yaw

   Objetivo: Ángulo de orientación del robot respecto al eje horizontal del mundo simulado.

   Justificación: Permite calcular el desfase angular (angle) entre la dirección actual del robot y la dirección al objetivo. Esta diferencia angular determina el tipo de giro que el robot debe aplicar, siendo una guía fundamental para la navegación reactiva.

3) self.angle

   Objetivo: Ángulo relativo entre la orientación del robot y la posición del objetivo, normalizado a $[-\pi, \pi]$.

   Justificación: Informa al agente cuán desviado está del rumbo óptimo hacia el objetivo. Una alineación cercana a cero indica una buena orientación. Se incluye directamente en el vector de observación para favorecer políticas que premien el alineamiento.

4) self.distance2

   Objetivo: Distancia al cuadrado entre el robot y el objetivo.

   Justificación: Elegir la distancia cuadrada en lugar de la euclidiana reduce el coste computacional, evitando el uso de la raíz cuadrada en cada paso. Esta variable mide el progreso y se usa para calcular el delta de avance (delta_d) entre pasos consecutivos.

5) self.laser_ranges

   Objetivo: Conjunto reducido de medidas del sensor LIDAR, representando obstáculos alrededor del robot.

   Justificación: Esta información permite detectar paredes, objetos y zonas libres de paso. Se procesan 30 medidas frontales y 5 laterales para reducir la dimensionalidad. Se usa agrupamiento por bloques para capturar el entorno de forma robusta y eficiente.

6) self.state

Objetivo: Vector de entrada a la red neuronal del agente, compuesto por [angle, distance2] + laser_ranges.

Justificación: Resume en un único array la información necesaria para que el agente tome decisiones. Este diseño equilibra información sensorial y orientación objetivo.

7) self.stuck_counter

Objetivo: Contador de pasos consecutivos sin progreso.

Justificación: Identifica bloqueos o comportamientos repetitivos. Al superar cierto umbral (STUCK_LIMIT), se activa una política de escape que permite al robot retroceder y girar para desbloquearse.

3.3 Diseño de la función de recompensa

La función de recompensa fue diseñada para guiar al agente (ROSbot) hacia objetivos definidos dentro de un entorno simulado, evitando obstáculos y optimizando tanto la orientación como el avance.

La recompensa considera múltiples factores que influyen en la toma de decisiones del agente. Cada acción ejecutada genera una recompensa inmediata compuesta por varias sub-recompensas y penalizaciones, según las siguientes dimensiones:

1. Progreso hacia el objetivo: Se premia el avance si la distancia al objetivo disminuye, en lugar de utilizar la distancia euclidiana, se utiliza el cálculo de la diferencia de la distancia al cuadrado (distance2) entre pasos consecutivos. Esta decisión reduce el coste computacional (al evitar la raíz cuadrada) y mantiene la proporcionalidad con el progreso real del robot. Además, este enfoque permite una señal de recompensa clara y estable que refleja de manera precisa cuán efectivo ha sido el paso dado en acercarse al objetivo.
2. Orientación al objetivo: Se recompensa al agente por reducir el ángulo relativo entre la dirección del robot y la posición del objetivo. Se otorgan bonificaciones adicionales si el robot se encuentra bien alineado.
3. Avance alineado: Si el agente avanza mientras está correctamente orientado, se proporciona una recompensa adicional que incentiva el avance únicamente cuando el ángulo relativo al objetivo es menor a 0.2 radianes.
4. Evasión de obstáculos: Se penaliza fuertemente la colisión directa y se aplican penalizaciones proporcionales si el robot está muy cerca de un obstáculo frontal.
5. Comportamiento repetitivo o atascado: Se detecta si el robot repite las mismas acciones o permanece en la misma posición sin avanzar. En estos casos, se activa un contador de bloqueo y se aplican penalizaciones progresivas, fomentando la exploración.
6. Escape inteligente: Si el robot logra desbloquearse y cambia de estrategia, se le recompensa por salir del estado de atasco. Además, si intenta nuevas direcciones con giros, se premia el intento de escape.

7. Comportamientos indeseables: Se penaliza retroceder cuando el robot está correctamente orientado, girar innecesariamente si está perfectamente alineado, o no corregir la orientación cuando el ángulo hacia el objetivo es elevado.

3.3.1 Propósito y comportamiento deseado

a) Comportamiento que se busca entrenar

El objetivo principal es que el robot aprenda a navegar de forma autónoma hacia objetivos específicos, desplazándose en un entorno simulado con obstáculos y sin conocimiento previo del mapa. Este comportamiento implica mucho más que simplemente avanzar, ya que el agente debe ser capaz de tomar decisiones coherentes basadas únicamente en datos sensoriales (LIDAR y odometría), sin instrucciones explícitas.

El comportamiento esperado incluye:

1. Orientarse hacia el objetivo de forma progresiva, girando hacia la dirección más corta.
2. Avanzar de manera constante y segura cuando el camino está despejado.
3. Detenerse o maniobrar al detectar un obstáculo en el campo visual frontal.
4. Detectar situaciones de atasco (cuando repite acciones sin avanzar) y ejecutar maniobras de escape.
5. Aprovechar las oportunidades de avance alineado cuando su orientación está correctamente ajustada.
6. Finalizar exitosamente el episodio alcanzando el objetivo dentro de un umbral de proximidad.

b) Como se promueve el comportamiento

La función de recompensa está diseñada para reforzar de manera positiva cada una de estas conductas deseadas y penalizar aquellas que resultan ineficientes, inseguras o improductivas.

1. Se premia al robot cuando se acerca al objetivo (delta_d > 0) y cuando reduce su ángulo con respecto al destino.
2. Si el robot está bien alineado y avanza, recibe recompensas adicionales que refuerzan ese patrón.
3. Cuando el entorno está libre de obstáculos y el agente mantiene la dirección, se le otorga un bonus extra como reconocimiento de una navegación fluida y eficaz.
4. Si el robot detecta un obstáculo cercano o colisiona, se le aplica una penalización fuerte, enseñándole a evitar zonas peligrosas.
5. La función de recompensa también castiga la repetición de acciones y la inactividad, reforzando la necesidad de explorar nuevas decisiones cuando se encuentra en una situación ineficaz.
6. Además, si el robot intenta maniobras de escape (por ejemplo, retroceder y girar), aunque no resulten inmediatas, son recompensadas como señales de comportamiento adaptativo.

c) Estructura de la recompensa

Cada valor dentro de la función de recompensa fue calibrado en base a múltiples pruebas y versiones preliminares del modelo. Las decisiones de diseño no fueron arbitrarias, sino que se tomaron con el propósito de equilibrar la señal de aprendizaje, evitando tanto recompensas excesivas como castigos injustificados que pudieran bloquear el aprendizaje.

3.3.2 Recompensas

Las recompensas durante la evolución del modelo en las distintas versiones fueron ajustándose, luego de exhaustivas pruebas y simulaciones, las recompensas finales se encuentran en la Tabla 1.

Tabla 1

Recompensas del modelo

Condición	Recompensa	Variable
Al alcanzar objetivo	1000	REWARD_GOAL_REACHED
Si el ángulo hacia el objetivo es menor a 0.1 rad	200	REWARD_WELL_ALIGNED
Avance alineado con buena orientación (ángulo < 0.2 y avance)	400	REWARD_ADVANCE_ALIGNED
Si el robot logra salir de un atasco prolongado	300	REWARD_ESCAPE_SUCCESS
Si el robot intenta girar para escapar	3	REWARD_ROTATION_TRY_ESCAPE
Si la distancia al objetivo ha disminuido significativamente	200	REWARD_PROGRESS_STEP
Mantenimiento de alineación estable	50	REWARD_STEADY_ALIGNMENT

Como condiciones más relevantes, establecer una recompensa destacada de +1000 al alcanzar el objetivo, deja claro al agente que este es el comportamiento más valioso que puede aprender, priorizando de forma efectiva la finalización exitosa de episodios.

Por otro lado, la definición de recompensas intermedias consistentes (+200 a +400) por mantener alineación y avanzar alineado, ya que estas acciones son críticas para un desplazamiento seguro y eficiente en entornos con obstáculos. Al reforzar estas conductas intermedias, el agente desarrolla políticas que favorecen la alineación progresiva antes de avanzar, garantizando trayectorias más rectas y una reducción de colisiones, lo que ha demostrado ser clave en la mejora del rendimiento con respecto a las versiones anteriores del modelo.

3.3.3 Penalizaciones

Las penalizaciones fueron ajustándose en base al resultado del entrenamiento y test, con una total atención en la visualización del simulador y en los logs, las penalizaciones para la versión del modelo se encuentran en la Tabla 2.

Tabla 2

Penalizaciones del modelo

Condición	Penalización	Variable
Al detectar colisión directa	-1000	PENALTY_COLLISION
Penalización acumulativa por atasco	-0.5 * stuck_counter	PENALTY_STUCK_INCREMENT
Repetición de acciones sin cambio de comportamiento	-1.0	PENALTY_REPEATED_ACTION
Mal orientado y no gira	-150.0	PENALTY_BAD_ORIENTATION_NO_ROTATION
Retroceso estando bien orientado	-250.0	PENALTY_REVERSE_WHEN_ALIGNED
Atasco prolongado sin salida	-800.0	PENALTY_LONG_STUCK

Las penalizaciones han sido configuradas para equilibrar el refuerzo positivo, garantizando que el agente no solo avance hacia el objetivo, sino que lo haga de forma segura y eficiente.

Una colisión se penaliza con -1000, reflejando su gravedad y reforzando la necesidad de evitarlas a toda costa. Esta penalización elevada actúa como un disuasivo claro, previniendo que el agente adopte políticas de avance agresivo sin considerar la proximidad de obstáculos.

Asimismo, la incorporación de penalizaciones proporcionales por atasco, repetición de acciones y falta de movimiento, con el objetivo de empujar al agente a explorar estrategias alternativas y desbloquearse de situaciones sin progreso. Este enfoque promueve comportamientos de escape y fomenta la exploración, reduciendo el riesgo de que el agente quede atrapado en ciclos ineficientes de acción-reacción.

Estas penalizaciones, al ser implementadas de forma progresiva y contextual, aseguran que el aprendizaje sea robusto, seguro y alineado con los objetivos de navegación autónoma, demostrando su efectividad en la reducción de colisiones y la mejora de la tasa de éxito en las pruebas realizadas con el modelo actualizado.

3.3.4 Fórmula

La definición de la función de recompensa se formuló con un esquema ponderado, combinando incentivos por avance hacia el objetivo y penalizaciones por colisiones y desviaciones angulares, dado que no existe una manera directa de derivar teóricamente pesos óptimos en RL para navegación reactiva en entornos dinámicos (Sutton & Barto, 2018). Las acciones realizadas para la definición son:

i. Observación visual en Gazebo Ignition, evaluando la calidad de los comportamientos aprendidos, como el alineamiento al objetivo, la capacidad de evasión de obstáculos y la eficiencia de la trayectoria.
ii. Análisis de logs de entrenamiento en TensorBoard, observando la evolución de la recompensa media por episodio, las tasas de colisión y los pasos por episodio para cada configuración de pesos.
iii. Ajuste de un parámetro por iteración, manteniendo fijos los demás, para aislar y entender el impacto individual de cada factor en el comportamiento del agente.
iv. Comparación con métricas de éxito, priorizando configuraciones que maximizaban la tasa de éxito y minimizaban colisiones y pasos innecesarios.

La función de recompensa ponderada continua en el paso de tiempo *t:*

$$r_t = \alpha \cdot \Delta d_t + \beta \cdot |\theta_t| + \gamma \cdot v_t^{lineal} + \delta \cdot v_t^{angular}$$

Donde:

- $\Delta d_t = d_{t-1} - d_t$: cambio en la distancia objetivo (avance).
- $|\theta_t|$: valor absoluto del ángulo entre la orientación del robot y la dirección al objetivo.
- v_t^{lineal}: velocidad lineal (avance frontal) del robot en el paso actual.
- $v_t^{angular}$: velocidad angular (rotación) del robot en el paso actual, donde se penaliza la rotación excesiva

Los pesos definidos se muestran en la Tabla 3.

Tabla 3

Pesos de la función recompensa

Variable	Peso	Nombre	Descripción
α	150.0	WEIGHT_ADVANCE_DELTA_D	Recompensa por acercarse al objetivo
β	-150.0	WEIGHT_ANGLE_DEVIATION	Penalización por desviación angular
γ	200.0	WEIGHT_FORWARD_MOTION	Recompensa por avance frontal
δ	-80.0	WEIGHT_ROTATION	Penalización por rotación excesiva

Los pesos finales seleccionados (por ejemplo, +150 por alcanzar el objetivo, -150 para penalización de desviación angular) demostraron ser adecuados porque:

i. Permitieron que el agente aprenda a priorizar el avance hacia el objetivo con alineación adecuada.
ii. Evitaron bloqueos por repeticiones innecesarias de acciones y favorecieron el desbloqueo en atascos.
iii. Resultaron en un aprendizaje estable, sin caídas abruptas en las recompensas, ni oscilaciones extremas en la política.
iv. El agente priorizo avanzar alineado, evitar colisiones y desbloquearse ante atascos.

Así, se logró un balance práctico entre alineación, avance, evasión de obstáculos y penalización por colisiones, garantizando un aprendizaje seguro y eficiente.

3.4 Configuración del modelo PPO

El entrenamiento del modelo se ha llevado a cabo utilizando el algoritmo PPO que es ampliamente adoptada en robótica por su equilibrio entre estabilidad, eficiencia computacional y facilidad de implementación. Para este trabajo, se utilizó la librería Stable-Baselines3, que integra herramientas modernas como PyTorch y Gymnasium, permitiendo un entrenamiento transparente y reproducible.

El entrenamiento fue ejecutado en una máquina con Ubuntu 22.04 LTS, Python 3.10, PyTorch 2.6 y WSL 2 como entorno de ejecución.

3.4.1 Arquitectura de la política

La arquitectura seleccionada en este trabajo es la red neuronal perceptrón multicapa (MLP) coherente con los lineamientos presentados en el apartado 2.8 del Estado del Arte.

En dicho apartado se explicó que, para tareas de navegación robótica con entradas vectoriales, las MLP son adecuadas por las siguientes razones (Lillicrap et al., 2015; Mnih et al., 2015)

i. Son idóneas para procesar vectores de observación con información sensorial comprimida, como las 37 dimensiones utilizadas en este TFM (35 valores LIDAR, ángulo y distancia),
ii. Capacidad de aproximación de funciones no lineales, permitiendo mapear de forma precisa las observaciones (distancia al objetivo, ángulo, LIDAR) a acciones continuas (velocidad lineal y angular).
iii. Estabilidad y simplicidad en el ajuste de hiperparámetros, reduciendo la complejidad y evitando problemas de sobreajuste.
iv. Requieren menor tiempo de cómputo que arquitecturas CNN o RNN, facilitando el entrenamiento y la inferencia en tiempo real incluso en hardware modesto.

Esta arquitectura es utilizada de forma consistente en trabajos previos de RL en robótica móvil por su capacidad para generalizar políticas efectivas en entornos de navegación sin necesidad de recurrir a CNN o RNN. Por tal motivo, la decisión se seleccionar MLP garantiza estabilidad, eficiencia computacional y coherencia con los principios expuestos en el Estado del Arte.

3.4.2 Hiperparámetros

El ajuste de los hiperparámetros fue una parte crítica del proceso. En versiones preliminares del modelo, el entrenamiento con valores por defecto generaba comportamientos inconsistentes o atascos frecuentes. Por ello, en la versión final (v4) se realizó un ajuste fino, priorizando la estabilidad del aprendizaje, la diversidad de exploración y la velocidad de convergencia.

En la Tabla 4, se presenta los hiperparámetros utilizados, su propósito, y los valores recomendados típicos para tareas similares.

Tabla 4

Hiperparámetros del modelo

Parámetro	Valor	Objetivo	Rangos	Comentario
learning_rate	0.0001	Controla la velocidad de aprendizaje del modelo	1e-5 – 1e-3	Valor bajo para evitar oscilaciones en la política
n_steps	2048	Tamaño del batch de experiencia antes de cada actualización	512 – 8192	Mayor estabilidad con valores altos
batch_size	64	Tamaño de cada minibatch durante la actualización	32 – 256	Compromiso entre estabilidad y velocidad
gamma	0.99	Factor de descuento para recompensas futuras	0.9 – 0.999	Valor común para tareas con secuencias largas
gae_lambda	0.95	Suaviza la ventaja generalizada (GAE)	0.9 – 0.97	Ayuda a balancear sesgo y varianza
clip_range	0.2	Limita cuánto puede cambiar la política por actualización	0.1 – 0.3	Evita saltos inestables en políticas
ent_coef	0.02	Estimula la exploración de nuevas acciones	0.01 – 0.1	Ayuda a evitar políticas demasiado deterministas
Policy	MlpPolicy	Tipo de arquitectura utilizada para la red neuronal	Mlp	Es adecuado para observaciones vectoriales

Justificación

1. learning_rate = 0.0001

   Se eligió un valor bajo para permitir actualizaciones suaves y evitar inestabilidad. En versiones preliminares con valores más altos (por ejemplo: 0.001), el robot oscilaba en su política y no convergía de forma estable.

2. n_steps = 2048

   En las primeras versiones, se aplicó el valor de 512, se notó que el agente actualizaba la política con muy poca experiencia acumulada, lo que generaba sobreajuste a situaciones puntuales. Con un valor más alto como 2048, se logró una ventana de experiencia más rica, permitiendo actualizaciones de la política basadas en episodios más completos y variados, esto llevo a estimaciones de ventaja y decisiones más robustas.

3. batch_size = 64

   En versiones preliminares del modelo, se probaron tamaños más pequeños como 32, que aceleraban la actualización, pero introducían mucho ruido en el gradiente, afectando negativamente la estabilidad. Por otro lado, valores superiores como 128 o 256 ralentizaban las actualizaciones y requerían más memoria sin mejoras visibles. Finalmente, se adoptó 64 como valor intermedio óptimo, proporcionando un equilibrio entre estabilidad, tiempo de cómputo y capacidad de generalización.

4. gamma = 0.99

   En las pruebas de las versiones preliminares del modelo, se aplicaron valores menores (gamma = 0.95), el agente aprendía a priorizar recompensas inmediatas, lo que conducía a comportamientos impacientes: giraba o retrocedía en exceso ante obstáculos sin buscar una trayectoria eficiente. Al aumentar gamma a 0.99, se reforzó la planificación a largo plazo, permitiendo que el agente mantuviera una orientación estratégica hacia el objetivo.

5. gae_lambda = 0.95

   Se inicio el modelo con un valor de 0.9, el modelo mostraba una alta sensibilidad a cambios en la función de valor, generando fluctuaciones y decisiones poco coherentes. Al aumentar el parámetro a 0.95, se suavizaron las estimaciones de ventaja generalizada, lo que resultó en una política más consistente y un entrenamiento más estable, especialmente en episodios largos donde los efectos de decisiones pasadas influyen acumulativamente.

6. clip_range = 0.2

   Con valores más bajos como 0.1, la política se volvía excesivamente conservadora, limitando las mejoras por paso y ralentizando el aprendizaje. Por el contrario, valores más altos como 0.3 daban lugar a saltos demasiado amplios en la política, degradando el comportamiento en episodios consecutivos. El valor final de 0.2 permitió mantener un buen equilibrio entre estabilidad y capacidad de adaptación.

7. ent_coef = 0.02

   En las versiones iniciales del modelo, se aplicó el valor de 0.01, y se notó que el robot convergía rápidamente a una política determinista que funcionaba en entornos simples, pero fallaba al enfrentarse a obstáculos inesperados. Subir el valor a 0.02 incrementó la diversidad de acciones exploradas durante el entrenamiento, mejorando la capacidad del agente para descubrir rutas alternativas y desbloquearse por sí mismo.

El modelo final (v4) fue entrenado con 200,000 pasos de simulación, lo que equivale aproximadamente a 7 horas de experiencia simulada.

Una de las mejoras más relevantes respecto a versiones preliminares del modelo, fue la optimización del espacio de observación, que pasó de 282 dimensiones a solo 37, sin pérdida de información esencial. Esto se logró mediante una estrategia de agrupamiento inteligente de las 1440 medidas del LIDAR, seleccionando únicamente las distancias más representativas de cada bloque frontal y lateral.

Esta reducción no solo disminuyó el coste computacional del modelo, sino que también aceleró el tiempo de entrenamiento y favoreció la convergencia estable de la política, al evitar ruido sensorial redundante. Con ello, el agente pudo enfocar su atención en señales clave del entorno para tomar decisiones más ágiles y eficaces.

3.5 Plan de entrenamiento y validación

El plan de entrenamiento y validación tiene como objetivo evaluar la eficacia del modelo PPO aplicado al ROSbot XL para tareas de navegación autónoma. El entrenamiento se estructura en fases de 40,000 pasos hasta alcanzar un total de 200,000 pasos con 7 horas aproximadamente de experiencia simulada, utilizando el entorno RosbotEnv integrado con Gazebo Ignition y ROS 2 Humble.

Durante la fase de entrenamiento, se monitorean métricas clave como la recompensa media por episodio, la entropía de la política y las pérdidas de política y valor mediante TensorBoard, permitiendo evaluar la progresión del aprendizaje y la estabilidad de la política aprendida.

La política se entrena con un espacio de acción continuo, utilizando velocidades lineales y angulares con límites de seguridad definidos en la configuración del entorno, y un espacio de observación de 37 dimensiones para optimizar el cómputo sin sacrificar información crítica. La función de recompensa está diseñada para incentivar el avance seguro hacia el objetivo, mantener la alineación y evitar colisiones, fomentando comportamientos de evasión de obstáculos de forma proactiva.

En la fase de validación, se ejecutan pruebas con objetivos aleatorios en entornos con obstáculos variados, registrando métricas de rendimiento como la tasa de éxito, pasos por episodio, duración y distancia final al objetivo. Estas pruebas permiten evaluar el comportamiento del agente tras el entrenamiento, identificando fortalezas y limitaciones de la política aprendida, cuya discusión y análisis de resultados se presentan en el Capítulo 5.

4 Implementación del modelo

Para el desarrollo del TFM se ha creado un entorno de entrenamiento robusto y totalmente integrado con ROS 2 Humble y el simulador Gazebo Ignition. Esta sección describe en detalle cómo se implementó la arquitectura que permite la interacción entre el agente entrenado, el entorno simulado, y el marco de aprendizaje por refuerzo, desde el flujo general de simulación hasta las consideraciones técnicas derivadas de la experiencia en este trabajo.

4.1 Diseño del entorno simulado

Se ha utilizado el simulador Gazebo Ignition Fortress, integrado con ROS 2 Humble mediante puentes ros_gz_bridge, y la interfaz rclpy en Python. Esta combinación permite modelar dinámicas físicas realistas, comunicaciones asíncronas entre sensores y actuadores, y un ciclo de percepción-acción compatible con el agente entrenado.

El entorno de simulación fue diseñado para representar un espacio con obstáculos estáticos, en que se incluyó el ROSbot XL de Husarion, equipado con un escáner LIDAR 2D de 1440 mediciones por giro (4 por grado) y odometría proporcionada por el simulador, que emula la información de posición y orientación en base al modelo físico del robot y sus parámetros de movimiento. Este modelo se lanza mediante los paquetes rosbot_xl_gazebo y rosbot_xl_description, ofreciendo parámetros físicos ajustados a condiciones reales.

Se definieron varios objetivos de navegación, cada uno representado por coordenadas fijas. A partir de posiciones iniciales aleatorias, el agente debía aprender a aproximarse a estos puntos evitando colisiones y superando situaciones de estancamiento. Durante el entrenamiento, se introdujeron perturbaciones en los sensores (ruido LIDAR) y variaciones en la dinámica para fomentar la generalización de la política aprendida.

Esta arquitectura permitió simular 200,000 pasos de entrenamiento en un entorno seguro, replicable y sin riesgos para hardware físico, cumpliendo así los principios de aprendizaje basado en simulación.

4.2 Flujo de simulación e integración

El flujo de integración del sistema permite al robot aprender comportamientos útiles de forma autónoma a partir de observaciones realistas, se basa en una arquitectura en la que convergen cuatro elementos clave:

a. Gazebo Ignition como simulador 3D del entorno físico.
b. ROS 2 Humble como infraestructura de comunicación robótica.
c. Gymnasium como API de entrenamiento compatible con RL.
d. Stable-Baselines3 como librería de entrenamiento PPO.

4.2.1 Configuración del entorno de simulación con Gazebo Ignition

La simulación se ejecuta con Gazebo Ignition, cargando el modelo del ROSbot XL dentro de un mundo virtual. El entorno incluye:

i. Controladores de movilidad (diferencial).
ii. Sensor LIDAR de 360° con 1440 medidas por escaneo, de los cuales en la versión final del modelo se incluyen solo 35 medidas.
iii. Publicadores de odometría (/odom).
iv. Soporte para ros_gz_bridge que conecta datos entre Gazebo y ROS 2.

4.2.2 Creación de un nodo ROS 2 como interfaz con el entorno

En cada episodio, se lanza un nodo ROS 2 personalizado que actúa como puente entre el entorno Gym y el sistema ROS. Este nodo se encarga de:

i. Suscribirse a /scan (LIDAR) y /odom (posición).
ii. Publicar velocidades de movimiento en /cmd_vel.
iii. Sincronizar la llegada de datos mediante un objeto Future, asegurando que cada paso del entorno (step) se base en información actualizada.

Este diseño es fundamental para mantener el control temporal y garantizar que el agente responda a situaciones actuales, no a datos atrasados.

4.2.3 Definición del entorno Gym compatible con Stable-Baselines3

El entorno RosbotEnv, desarrollado como una subclase de gym.Env, implementa los métodos step(), reset() y observation_space, entre otros.

La observación que recibe el agente en cada paso está formada por:

- Un ángulo relativo al objetivo (self.angle, en radianes).
- Una distancia al cuadrado hacia el objetivo (self.distance).
- 35 medidas del LIDAR procesadas (30 frontales, 5 laterales).

Esto conforma un vector de estado de dimensión 37, diseñado para equilibrar riqueza sensorial y eficiencia computacional.

4.2.4 Entrenamiento del robot con PPO

El entrenamiento se realiza con PPO mediante Stable-Baselines3. En el script train_ppo.py, se inicializa la política con una red neuronal MlpPolicy, utilizando los siguientes hiperparámetros:

i. learning_rate = 0.0001
ii. gamma = 0.99 (factor de descuento)
iii. gae_lambda = 0.95 (ventajas generalizadas)
iv. clip_range = 0.2 (estabilidad de actualizaciones)
v. ent_coef = 0.02 (exploración)

Se entrena en 5 fases de 40.000 pasos cada una, generando checkpoints intermedios y logs para visualización con TensorBoard.

4.2.5 Evaluación del modelo entrenado en simulación

Finalizado el entrenamiento, se utiliza test_rosbot.py para cargar el modelo PPO y probarlo en el mismo entorno simulado. En esta fase:

i. Se publica un objetivo dinámico en el tópico /goal.
ii. El entorno espera a recibir datos de /scan y /odom.
iii. El modelo genera acciones (forward, rotation) que se publican en /cmd_vel.
iv. El comportamiento del robot se registra en logs del terminal, permitiendo analizar si alcanza el objetivo, cuántos pasos emplea y qué estrategia sigue.

4.3 Arquitectura del entorno y sincronización sensorial

El entorno personalizado RosbotEnv implementa la interfaz estándar de gym. Env, lo que permite su uso directo con cualquier algoritmo de la biblioteca stable-baselines3. Internamente, RosbotEnv actúa como un nodo ROS 2, utilizando publicaciones y suscripciones para interactuar con los tópicos del robot simulado.

La arquitectura del entorno incluye:

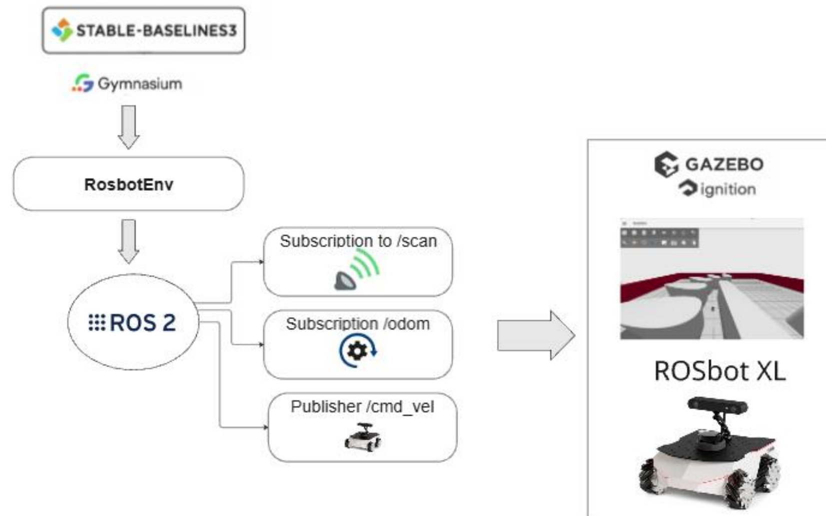
a) Suscripción a /scan: Se reciben datos del sensor LIDAR, que se procesan 35 medidas representativas mediante agrupamiento por bloques.
b) Suscripción a /odom: Se utiliza para calcular la posición y orientación del robot, así como el ángulo y la distancia al objetivo.
c) Publicador en /cmd_vel: Envía comandos de movimiento que son ejecutados en la simulación.
d) Gestión de episodios: La función reset() reinicia el entorno, selecciona un nuevo objetivo si corresponde y espera sincronización completa de sensores antes de comenzar.
e) Evaluación de recompensa: El entorno calcula dinámicamente la recompensa en función de la orientación, colisiones, distancia, estancamiento, y otros indicadores del comportamiento.
f) Inicialización (__init__): Define los parámetros del entorno, crea el nodo ROS 2, configura los espacios Gymnasium y establece los valores iniciales.
g) Funciones de Callbacks sensoriales (laser_callback, odom_callback): Definidos para capturar los datos de LIDAR y odometría, y los almacenan en variables internas. En la versión final del modelo, el laser_callback() aplica una compresión por bloques (agrupación de 8 índices), retornando el valor mínimo de cada bloque. Esto reduce la dimensionalidad de 280 entradas a solo 35 sin perder sensibilidad ante obstáculos.
h) Método step(): Es el núcleo del entorno. Publica una acción, espera a recibir nuevos datos, calcula la nueva observación, recompensa y si el episodio terminó. Incluye lógica para detectar colisiones, estancamientos, progreso y orientación.
i) El uso de constantes en el archivo rosbot_config.py permite modificar umbrales, recompensas y penalizaciones sin alterar el núcleo de Rosbot_Env.

Para comprender la integración de los componentes utilizados en este trabajo, se presenta el diagrama de la arquitectura del entorno de simulación y

entrenamiento. La Figura 13, muestra cómo el entorno personalizado RosbotEnv interactúa con la biblioteca Stable-Baselines3, ROS 2 y el simulador Gazebo Ignition, permitiendo el entrenamiento del modelo de aprendizaje por refuerzo sobre el robot ROSbot XL en un entorno simulado.

Figura 13

Integración de RosbotEnv con Stable-Baselines3, ROS 2 y Gazebo Ignition para el entrenamiento del ROSbot XL



Nota. La figura muestra el flujo de datos entre Stable-Baselines3, Gymnasium y el entorno RosbotEnv, que se comunica mediante ROS 2 con los tópicos /scan, /odom y /cmd_vel. Gazebo Ignition simula el entorno físico.

La sincronización entre las acciones del agente y las observaciones sensoriales del entorno es esencial para garantizar consistencia durante el entrenamiento. En esta implementación, el envío de comandos de acción se realiza mediante la publicación en el tópico /cmd_vel usando Twist, un mensaje estándar en ROS 2 que define la velocidad lineal y angular del robot.

Tras la publicación de cada acción, el entorno bloquea la ejecución mediante el uso de rclpy.spin_until_future_complete(self.node, self.future), que espera a que las dos callbacks sensoriales hayan recibido nuevas observaciones. Cada callback establece self.future.set_result(True) solo una vez se ha recibido y procesado la nueva muestra.

Este sistema garantiza que cada paso de entrenamiento se base únicamente en datos reales y síncronos, evitando inconsistencias o acciones basadas en observaciones antiguas. Además, permite una frecuencia de control adaptable, aunque se ha mantenido en 20 Hz para emular el tiempo real.

La arquitectura también incluye filtros simples para evitar que valores extremos o erróneos de sensores interfieran en el aprendizaje, como la asignación de un valor máximo (10.0 m) cuando el LIDAR no detecta ningún obstáculo.

4.4 Consideraciones técnicas y limitaciones

Durante la implementación y ejecución del modelo, se identificaron varias consideraciones técnicas que fueron fundamentales para alcanzar resultados satisfactorios:

4.4.1 Consideraciones técnicas

i. Se empleó el paquete rosbot_xl_ros para simular correctamente la cinemática del robot.
ii. Se utilizó ros_gz_bridge para asegurar la conexión entre los tópicos de ROS 2 y Gazebo Ignition.
iii. Se configuró el entorno con una frecuencia de control de 20 Hz, lo que permitió un equilibrio entre fluidez y estabilidad en la simulación.
iv. La selección de objetivos se automatizó a través del vector GOALS, con cambio dinámico al completar un episodio.

4.4.2 Limitaciones identificadas

El simulador Gazebo intenta emular las características del LIDAR de forma realista, incluyendo ruido sensorial, rangos de lectura y latencias, aunque siempre existen limitaciones comparadas con condiciones reales, como la falta de interferencias de luz o superficies altamente reflectantes.

El entrenamiento presento lentitud en versiones preliminares del modelo, ya que durante los primeros 40.000 pasos, tuvo dificultades para aprender comportamientos útiles. Esto se mitigó afinando la función de recompensa y los hiperparámetros.

A pesar de trabajar en simulación, los datos del LIDAR a veces presentaban lecturas nulas o inconsistentes. Por ello, se aplicó el filtrado por np.nanmin() para reducir errores.

4.4.3 Entorno de desarrollo y configuración técnica

Con la finalidad de garantizar la estabilidad y compatibilidad del sistema, se configuró un entorno que integra múltiples componentes de software. Cada una de las herramientas y librerías utilizadas cumple un rol esencial en la simulación, entrenamiento y ejecución del modelo PPO sobre el robot ROSbot XL.

En la Tabla 5, se describen los principales elementos del entorno de desarrollo, junto con su versión, función dentro del sistema y consideraciones técnicas relevantes.

Tabla 5

Tabla de componentes y librerías

Componente/ Librería	Versión	Función en el modelo	Detalles técnicos
WSL 2	Windows Subsystem for Linux 2	Proporciona un entorno Linux completo dentro de Windows	Permite ejecutar ROS 2 y Gazebo Ignition nativamente sin necesidad de máquina virtual externa. Soporte de red y acceso al sistema de archivos de Windows.
Ubuntu 22.04 LTS (Jammy)	Kernel: 5.15.x	Sistema operativo base	Elegido por su compatibilidad estable con ROS 2 Humble y paquetes de simulación robótica. Ejecutado dentro de WSL
ROS 2 Humble	2022 (LTS)	Middleware robótico	Maneja la comunicación entre nodos, tópicos como /cmd_vel, /scan, /odom, y proporciona la infraestructura para el puente ROS–Gazebo.
Gazebo Ignition Fortress	v6	Simulador	Simula el entorno 3D y la dinámica del ROSbot XL. Se emplea ros_gz_bridge para conectar datos de simulación con ROS 2.
Python	3.10.12	Lenguaje de programación	Versión recomendada para ROS 2 y compatible con las versiones actuales de Gymnasium, SB3 y PyTorch. Se usó para scripts de entrenamiento y test.
Stable-Baselines3	2.5.0	Entrenamiento con PPO	Librería de RL en PyTorch. Se utilizó con MlpPolicy para entrenar una política continua. Incluye logging con TensorBoard.
Gymnasium	1.0.0	API para entornos RL	Define métodos como step(), reset() y los espacios de acción/observación.
Pytorch	2.6.0 + cu124	Backend de aprendizaje	Ejecuta el modelo de red neuronal (policy + critic). Incluye soporte CUDA para entrenamiento acelerado.
VS Code	1.101	IDE de desarrollo	Entorno de codificación, ejecución de scripts, y edición de nodos ROS 2

La elección de estas librerías y componentes responde a tres objetivos clave:

- Compatibilidad con ROS 2 y simuladores de nueva generación (Ignition Fortress).
- Entrenamiento robusto con algoritmos de RL modernos (PPO sobre Stable-Baselines3).
- Facilidad de desarrollo y reproducibilidad mediante un entorno basado en WSL 2 y Ubuntu LTS, sin complejidad adicional en la configuración.

4.4.4 Características de hardware utilizado

Para la implementación del modelo se utilizaron entornos simulados que requieren una combinación considerable de potencia de cálculo, almacenamiento y compatibilidad gráfica. En la Tabla 6, se presentan los recursos de hardware del equipo utilizado para el entrenamiento y test, comparados con los requisitos mínimos y recomendados para ejecutar Ignition Gazebo y entornos de aprendizaje por refuerzo con PPO.

Tabla 6

Requerimientos de hardware

Recurso	Equipo utilizado	Requisito mínimo	Requisito recomendado
CPU	Intel Core i5-8250U @ 1.6 GHz (4C/8T)	Intel i5 / Ryzen 3 (2.5 GHz+)	Intel i7 / Ryzen 5 (3.0 GHz+)
RAM	16 GB DDR4	8 GB	16 GB a más
GPU (Intel)	Intel UHD 620 (128 MB VRAM)	-	NVIDIA GTX 1050 o superior
GPU (NVIDIA)	GeForce MX110 (2 GB VRAM)	-	GTX 1050 o superior
Sistema Operativo	Windows 11 Pro + WSL 2 (Ubuntu 22.04)	Ubuntu 20.04 / Windows 10	Ubuntu 22.04 / Windows 10

La implementación del modelo, fue desarrollado en un equipo que, aunque cuenta con RAM suficiente y un procesador con múltiples hilos, dispone de una GPU dedicada de nivel básico (MX110) y una gráfica integrada Intel que no es compatible con Ignition Gazebo de forma nativa.

Esto obligó a implementar múltiples estrategias de optimización computacional:

- Reducción del espacio de observación LIDAR de 282 a 37 dimensiones.
- Entrenamiento en fases para evitar cargas excesivas.
- Control de frecuencia a 20 Hz para equilibrar precisión y rendimiento.
- Uso de una única instancia de simulación (sin paralelización ni entornos vectorizados reales).

5 Resultados y análisis

En este capítulo, se muestra los resultados obtenidos tras del entrenamiento del modelo v4, simulado con Gazebo Ignition, ROS 2 Humble y Gymnasium, completando 200,000 pasos y 7 horas aproximadamente de experiencia simulada. Es importante resaltar que la versión final del modelo paso por un proceso iterativo de tres versiones previas (v1, v2 y v3), mejorando lentamente el espacio de observabilidad, la función de recompensa y los hiperparámetros, proporcionando la oportunidad de identificar y abordar problemas relacionados con la estabilidad, la velocidad de aprendizaje y las limitaciones de comportamiento en entornos densos en obstáculos antes de establecerse en la versión final del modelo.

Asimismo, se analizan la evolución del aprendizaje, el comportamiento del agente en distintas pruebas, métricas de desempeño y resultados obtenidos de cara a la futura transferencia sim-to-real y la robustez del modelo en tareas de navegación autónoma con evasión de obstáculos

5.1 Evolución del aprendizaje (entrenamiento)

El entrenamiento del modelo se realizó en 5 fases de 40,000 pasos, totalizando 200,000 pasos; además, se registraron métricas en TensorBoard para permitir un análisis detallado desde el principio hasta el final del proceso de entrenamiento. Cada fase produjo un checkpoint del modelo entrenado (rosbot_model_v4_checkpoint_N) para facilitar la comparación de la iteración de políticas a través de las fases. Esta estrategia permite guardar avances progresivos y evitar la pérdida de entrenamiento ante cualquier interrupción.

El proceso se llevó a cabo en un entorno con ROS 2 Humble, Gazebo Ignition y Gymnasium integrados a través de ros_gz_bridge; y el entrenamiento se realizó con Stable-Baselines3, con una frecuencia de control de 20 Hz (50 ms por ciclo), lo que significa que, después de publicar una acción en /cmd_vel, se espera la nueva información de LIDAR y odometría antes de pasar al siguiente paso de entrenamiento. Este tiempo incluye latencias de simulación, publicación y callbacks sensoriales, asegurando que las decisiones del agente se basen en datos recientes y precisos

Durante el entrenamiento, el agente interactúa con el entorno simulando acciones a través del publicador /cmd_vel y recibiendo observaciones desde los topicos /scan y /odom.

5.1.1 Proceso de aprendizaje progresivo

Durante las primeras fases de entrenamiento, se observó que el agente ejecutaba acciones de forma aleatoria, reflejando la necesidad de exploración inicial con alta entropía de política, como es característico en algoritmos de RL. A partir de la segunda fase, comenzaron a consolidarse patrones de comportamiento relevantes, donde el agente:

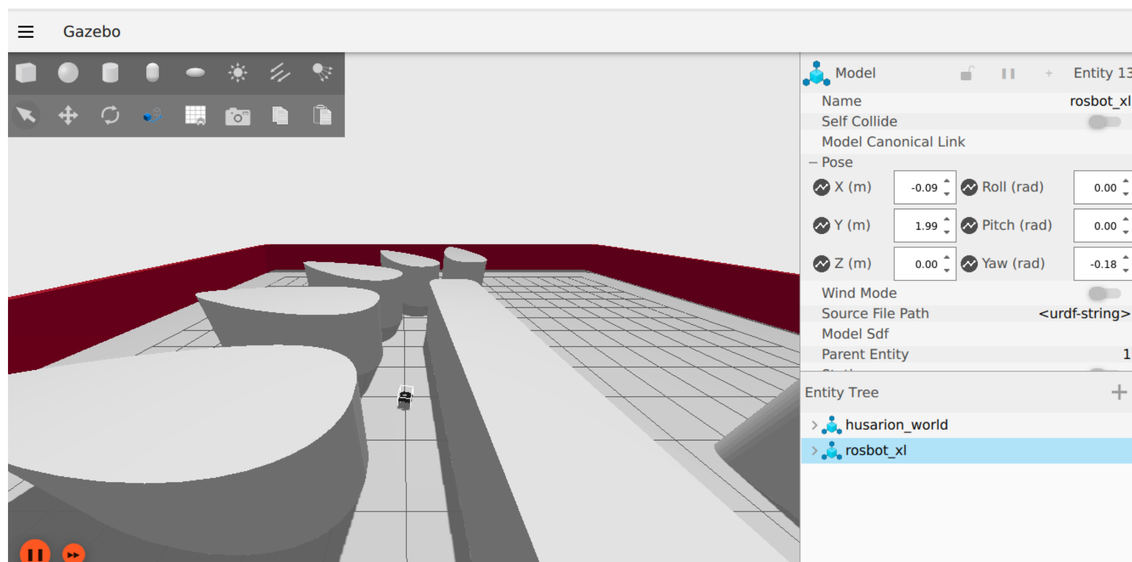
i. Aprendía a mantener el rumbo hacia el objetivo.
ii. Evitaba colisiones anticipando la proximidad de obstáculos detectados por el LIDAR.
iii. Corregía la orientación antes de avanzar, favoreciendo trayectorias más directas.
iv. Activaba estrategias de desbloqueo tras detectar pasos sin progreso, empleando retrocesos y giros controlados.

Este aprendizaje progresivo permitió al modelo refinar su política de navegación, ajustando sus decisiones en función de las recompensas obtenidas, con una evolución visible en la reducción de comportamientos erráticos y en la mejora de la estabilidad en las trayectorias.

Para ilustrar este proceso de aprendizaje, en la Figura 14 se muestra al ROSbot XL desplazándose en el entorno simulado con obstáculos durante el entrenamiento, reflejando cómo el agente avanza y corrige su orientación en espacios reducidos mientras ejecuta su política de navegación aprendida. Asimismo, en la Figura 15 se presenta un fragmento del registro del entorno durante el entrenamiento, donde se visualizan en tiempo real las métricas de distancia al objetivo, ángulo de orientación, recompensas obtenidas y el estado de avance del episodio, mostrando de forma clara cómo el agente evalúa sus decisiones paso a paso mientras refina su comportamiento.

Figura 14

El ROSbot avanzando en el entorno de entrenamiento en el simulador Gazebo Ignition.



Nota. Captura del simulador mostrando el desplazamiento del ROSbot XL en un escenario con obstáculos, mientras aplica la política de navegación durante el entrenamiento.

Figura 15

Registro de métricas de navegación y recompensas durante el entrenamiento.

```
[1750956680.597583009] [rosbot_env]: 📍>>> Distancia al objetivo: Distancia²=7.8757
[1750956680.599406709] [rosbot_env]: >>>> Reducción de ángulo hacia objetivo. Δángulo = 0.01
[1750956680.602290609] [rosbot_env]: >>> Progreso notable: Δd = 0.060
[1750956680.603860709] [rosbot_env]: >>> Recompensa: 3.10, Done: False
[1750956680.712520211] [rosbot_env]: 📍>>> Distancia al objetivo: Distancia²=7.8757
[1750956680.714189311] [rosbot_env]: >>> Progreso notable: Δd = 0.060
[1750956680.715152311] [rosbot_env]: >>> Recompensa: -2.66, Done: False
[1750956680.761377312] [rosbot_env]: 📌INICIO: Robot en (0.20, 1.91) | Offset x, y: (0.00, 2.00) | Ángul
-0.21 | Pose x,y: 0.20, -0.09
[1750956680.763157112] [rosbot_env]: 📍>>> Distancia al objetivo: Distancia²=7.8491
[1750956680.764906412] [rosbot_env]: >>>> Reducción de ángulo hacia objetivo. Δángulo = 0.01
[1750956680.767140512] [rosbot_env]: >>> Recompensa: 1.04, Done: False
[1750956680.879867715] [rosbot_env]: 📍>>> Distancia al objetivo: Distancia²=7.8491
[1750956680.881490315] [rosbot_env]: >>> Recompensa: -4.80, Done: False
[1750956680.912143515] [rosbot_env]: 📌INICIO: Robot en (0.20, 1.91) | Offset x, y: (0.00, 2.00) | Ángul
-0.19 | Pose x,y: 0.20, -0.09
[1750956680.956253016] [rosbot_env]: 📍>>> Distancia al objetivo: Distancia²=7.8248
```

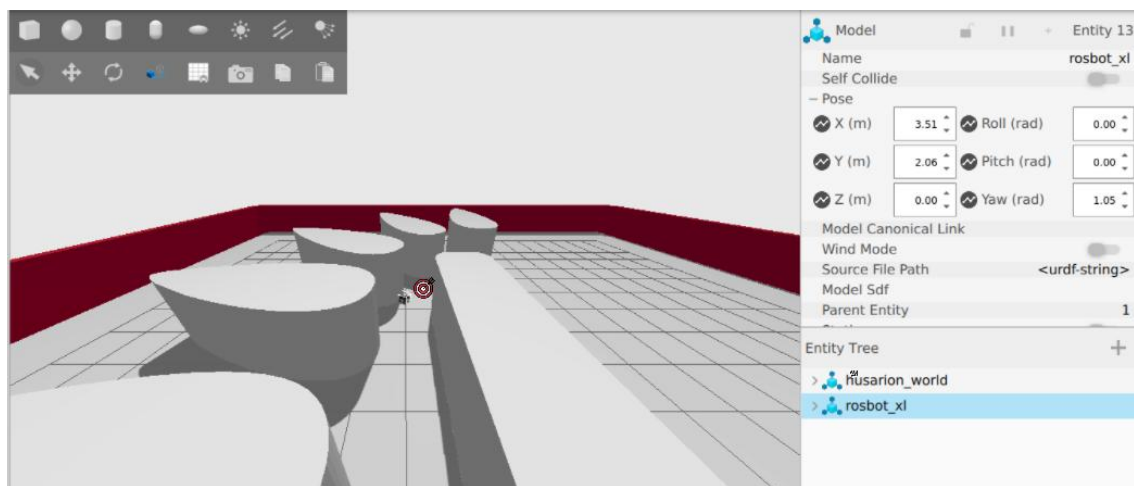
Nota. Fragmento del log durante el entrenamiento, mostrando en tiempo real las métricas utilizadas para calcular las recompensas y tomar decisiones de avance y orientación.

5.1.2 Indicadores cuantitativos del progreso

La supervisión del entrenamiento se realizó mediante la constante visualización del simulador y la monitorización con TensorBoard, lo que permitió visualizar en tiempo real la evolución de las métricas clave, identificar tendencias de aprendizaje y detectar posibles inestabilidades en la política. En la Figura 16 se muestra al ROSbot XL durante uno de los episodios de entrenamiento, alcanzando el objetivo mientras sortea obstáculos, ejemplificando el contexto de navegación utilizado para recolectar datos sensoriales y evaluar el aprendizaje en cada iteración.

Figura 16

El ROSbot XL alcanzando el objetivo durante el entrenamiento.



Nota. Captura de Gazebo Ignition generada durante el entrenamiento del modelo PPO.

Entre las métricas supervisadas, se destacan cinco indicadores principales que permitieron analizar en detalle el progreso del entrenamiento:

1. Evolución de la pérdida de entrenamiento (train/loss)

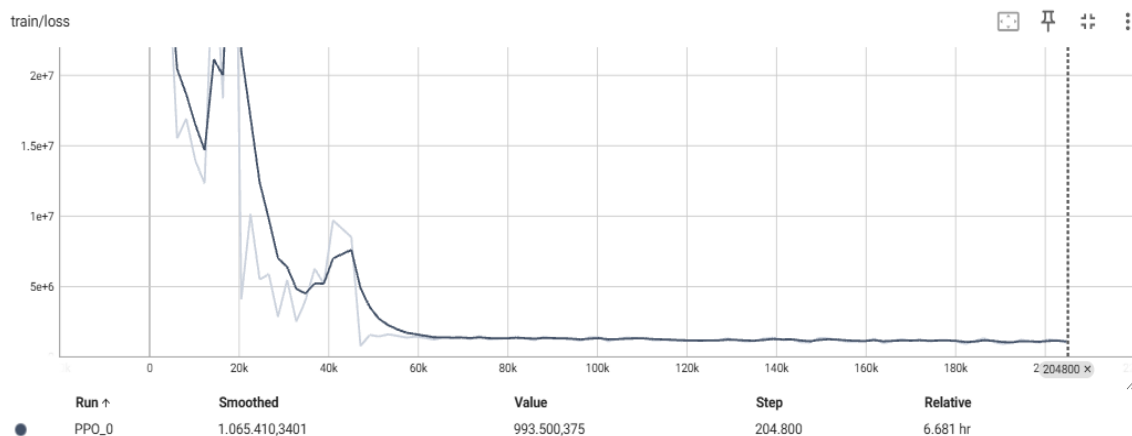
   Durante el entrenamiento la métrica mostró un comportamiento estable y progresivo, reflejando la correcta sintonización de los hiperparámetros utilizados. Al inicio, la pérdida se encontraba en valores elevados superiores a 20 millones, característicos de la fase de exploración donde el agente aún no posee experiencia. A medida que avanzaron las iteraciones, se observó una reducción sostenida, descendiendo a valores en el rango de 1 a 2 millones alrededor de los 50,000 pasos y estabilizándose en torno a 993,500 al finalizar los 204,800 pasos en un tiempo total de 6.68 horas.

   Este comportamiento indica que el agente fue consolidando progresivamente patrones de navegación y predicciones de valor, sin oscilaciones bruscas, gracias a la configuración de los hiperparámetros: learning_rate (0.0001), que permitió actualizaciones suaves; clip_range (0.2), que evitó saltos inestables; y n_steps (2048), que ofreció ventanas de experiencia ricas y variadas. La estabilidad alcanzada en la métrica confirma que el modelo logró generar una política robusta, eficiente y lista para validar en pruebas de navegación autónoma en entornos con obstáculos.

   En la siguiente Figura 17 se muestra la evolución de la pérdida de entrenamiento registrada durante todo el proceso.

Figura 17

Evolución de la pérdida de entrenamiento (train/loss)



Nota. Análisis de la métrica train/loss generado por TensorBoard durante el entrenamiento del modelo.

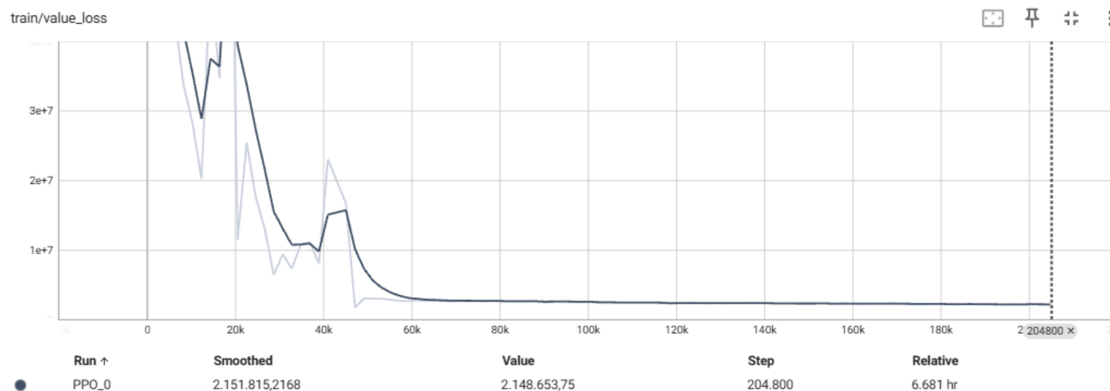
2. Evaluación de la estabilidad del crítico durante el entrenamiento (train/value loss)

   Durante el entrenamiento, la métrica inició en valores elevados superiores a 30 millones, reflejando la falta de referencias iniciales del agente al interactuar con el entorno. A medida que avanzaron las iteraciones, se observó un descenso sostenido y estable, reduciéndose a un rango entre 7 y 10 millones alrededor de los 50,000 pasos, hasta consolidarse en 2,150,000 al finalizar los 204,800 pasos. Este comportamiento evidencia que el crítico aprendió de manera progresiva a estimar de forma precisa las recompensas acumuladas y el valor de cada estado en el proceso de toma de decisiones del agente. La estabilidad alcanzada en las últimas fases refleja que las decisiones tomadas por el agente se basan en predicciones consistentes, reduciendo errores en la estimación de las ventajas y evitando comportamientos inestables.

   En la Figura 18 se muestra la evolución de la pérdida del crítico, permitiendo visualizar de forma clara este proceso de consolidación.

Figura 18

Evolución de la pérdida del crítico (train/value_loss) durante el entrenamiento del modelo



Nota. La imagen muestra la reducción sostenida de la pérdida del crítico a medida que el agente consolida su capacidad de estimar el valor de las acciones y estados.

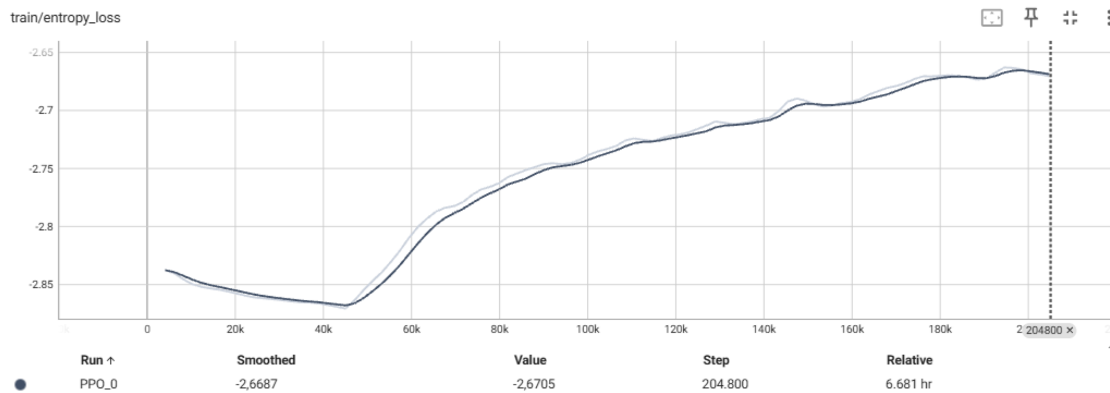
3. Supervisión de la entropía de la política durante el entrenamiento

   En la fase inicial la métrica train/entropy_loss, comenzó en valores alrededor de -2.85, reflejando un alto nivel de exploración mientras el agente interactuaba de forma aleatoria con el entorno y recolectaba experiencias variadas. A medida que avanzaron las iteraciones, se observó una ligera disminución hasta los 40,000 pasos, seguida de un incremento progresivo de la entropía hasta alcanzar valores de -2.67 al finalizar los pasos. Este incremento indica que la política logró preservar una diversidad controlada en la generación de acciones, evitando la

convergencia prematura hacia una política que pudiera limitar la capacidad de adaptación del agente en entornos con obstáculos o escenarios no vistos previamente. En la Figura 19 se muestra la evolución de la entropía de la política durante todo el entrenamiento.

Figura 19

Evolución de la entropía de la política (train/entropy_loss) durante el entrenamiento del modelo



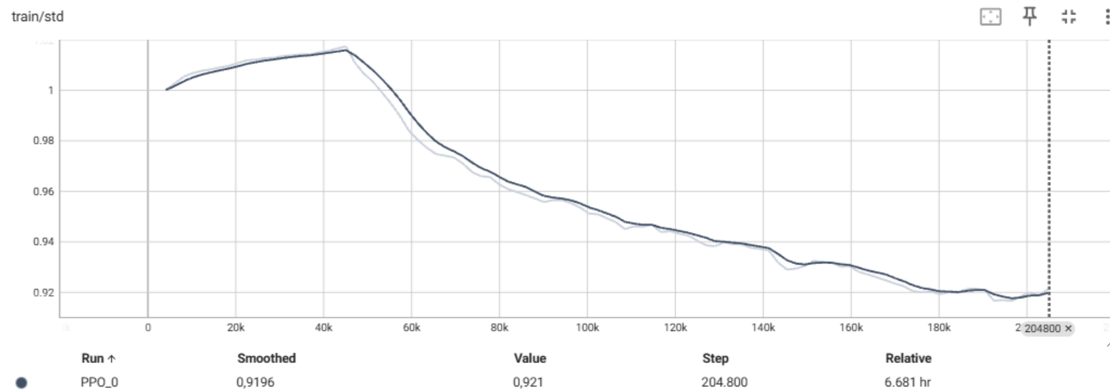
Nota. La figura muestra la evolución de la entropía de la política durante todo el entrenamiento.

4. Evolución de la variabilidad de la política durante el entrenamiento

La métrica train/std permite analizar la variabilidad en las acciones generadas por la política a lo largo del entrenamiento. Durante las primeras fases, la desviación estándar comenzó en valores de 0.92 y presentó un leve aumento inicial, seguido de un descenso sostenido hasta estabilizarse nuevamente en 0.92 al finalizar los 204,800 pasos. Esta reducción progresiva indica que el modelo fue especializando las acciones generadas, priorizando movimientos eficientes y consistentes en las decisiones de navegación, sin eliminar completamente la exploración necesaria para la adaptabilidad del agente. En la Figura 20 se muestra la evolución de la desviación estándar de la política durante todo el entrenamiento.

Figura 20

Evolución de la desviación estándar de la política (train/std) durante el entrenamiento del modelo



Nota. La figura de la desviación estándar, evidencia la reducción controlada de la variabilidad en las acciones generadas por la política.

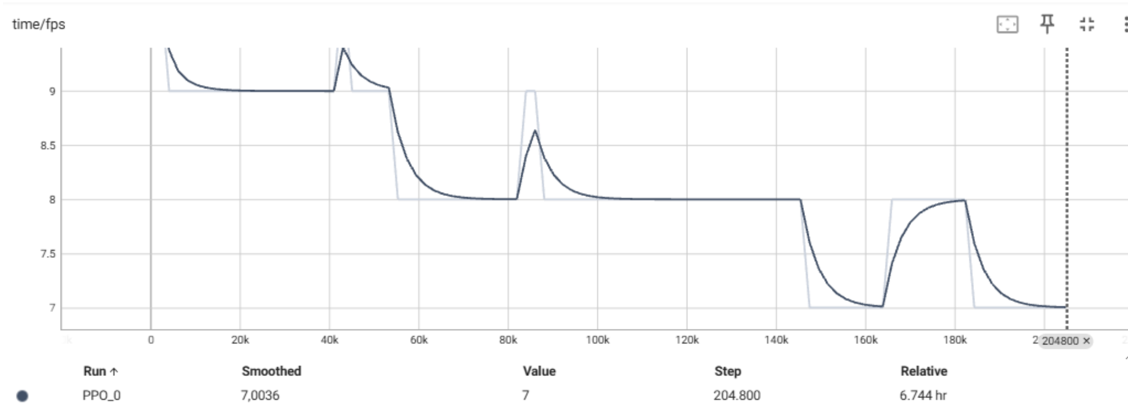
5. Rendimiento computacional durante el entrenamiento

   La tasa de frames por segundo (time/fps) permite evaluar la eficiencia de procesamiento durante el entrenamiento, reflejando cuántos pasos por segundo logra procesar el sistema mientras ejecuta la simulación.

   Al inicio del entrenamiento, se observa una tasa de 9 fps, adecuada para asegurar fluidez en la simulación y consistencia en la recolección de datos sensoriales y la ejecución de acciones. A medida que avanza el entrenamiento, se identifica una reducción progresiva en la tasa de fps, llegando a valores de 7 fps al finalizar los pasos del entrenamiento. El mantenimiento estable de 7 fps durante las últimas fases confirma que el hardware utilizado puede soportar entrenamientos largos y complejos, garantizando la consistencia temporal en la sincronización de acciones y observaciones. La Figura 21, muestra la evolución de los frames por segundo.

Figura 21

Evolución de los frames por segundo (fps) durante el entrenamiento



Nota. Gráfico en TensorBoard donde se observa el rendimiento de procesamiento durante las fases de entrenamiento con Gazebo Ignition y ROS 2.

En conjunto, estos indicadores confirman que el modelo implementado pudo aprender de forma progresiva una política de navegación robusta, segura y eficiente, utilizando los datos del LIDAR y la odometría para tomar decisiones autónomas en tiempo real. El uso de TensorBoard resultó esencial para validar cada fase del entrenamiento, facilitando la supervisión y el ajuste oportuno de parámetros durante las fases críticas, garantizando la estabilidad del aprendizaje y la calidad de las políticas generadas.

5.1.3 Observaciones cualitativas en el simulador

Se realizó un seguimiento cualitativo del comportamiento del ROSbot durante el entrenamiento, lo que permitió analizar el progreso del agente desde una perspectiva práctica, interpretando de manera visual y contextual las decisiones que la política aprendida ejecutaba en cada fase.

Al inicio del entrenamiento, el ROSbot exhibía movimientos erráticos, giros aleatorios y frecuentes colisiones, reflejando la fase de exploración necesaria en RL para recopilar experiencias diversas. Sin embargo, conforme comenzó a asociar acciones con recompensas y penalizaciones, se observaron cambios progresivos en su comportamiento, visibles durante las sesiones de supervisión en el simulador.

Una de las acciones más relevantes fue alinear su orientación hacia el objetivo antes de avanzar con velocidad lineal significativa. Este comportamiento emergió a medida que el agente internalizó que avanzar sin una orientación adecuada resultaba en trayectorias ineficientes y penalizaciones indirectas debido a desvíos y bloqueos. La observación en el simulador mostraba cómo el agente realizaba pequeños ajustes angulares, corrigiendo su rumbo de forma iterativa hasta lograr un ángulo de aproximación óptimo.

En entornos con obstáculos, el agente fue capaz de realizar maniobras de evasión con giros suaves, reduciendo temporalmente la velocidad lineal para evitar colisiones mientras conservaba una trayectoria que, en la medida de lo posible, mantenía la alineación con el objetivo. Se evidenció que, tras detectar proximidad con un obstáculo, el agente prefería desviarse ligeramente, rodear el obstáculo y retomar la orientación, evitando maniobras abruptas que pudieran comprometer la estabilidad de la trayectoria.

Un comportamiento relevante fue la activación del mecanismo de desbloqueo en situaciones de atasco, identificado cuando el agente permanecía durante un umbral de pasos sin reducción significativa de la distancia al objetivo. En estos casos, ejecutaba retrocesos controlados y giros de mayor amplitud, buscando liberar el espacio alrededor y explorar nuevas rutas de aproximación. Este comportamiento refleja la capacidad de adaptación del modelo, permitiéndole romper ciclos de bloqueo en entornos con configuraciones de obstáculos complejas.

Estos comportamientos cualitativos emergieron como resultado directo de la estructura de la función de recompensa diseñada en el modelo, que fomentaba el progreso hacia el objetivo, la alineación angular y penalizaba de forma significativa las colisiones y los bloqueos prolongados.

5.2 Comportamiento

Tras finalizar el entrenamiento, la validación del modelo demostró un comportamiento consistente, eficiente y seguro al realizar tareas de navegación autónoma hacia objetivos aleatorios evitando obstáculos en entornos complejos. Se observaron patrones de comportamiento que reflejan que el agente no solo aprendió a moverse en línea recta, sino a tomar decisiones reactivas ante su entorno, incluyendo la corrección de ángulos, la evasión de obstáculos y la priorización de alineación antes del avance.

5.2.1 Comportamiento observado en espacios abiertos

Durante las pruebas en espacios abiertos, el ROSbot demostró un comportamiento fluido y eficiente, aplicando las políticas aprendidas para avanzar de forma estable hacia los objetivos sin colisiones y con correcciones de alineación precisas. En estos escenarios, el robot mantuvo trayectorias rectas y seguras, aprovechando al máximo el espacio disponible y reduciendo progresivamente la distancia al objetivo, con un ángulo de orientación que se mantuvo bajo en la mayoría de los pasos.

Sin embargo, en algunos casos, el ROSbot mostró una tendencia a mantener trayectorias rectas incluso si un leve ajuste de dirección podría haber optimizado la distancia al objetivo. Este comportamiento no contradice el modelo, sino que refleja la política entrenada, que prioriza la seguridad y la estabilidad del avance antes que la reducción agresiva del tiempo de llegada, conservando un margen de seguridad ante posibles obstáculos en el entorno.

En resumen, el robot pudo navegar de forma autónoma y confiable en espacios abiertos, alcanzando objetivos de forma consistente y segura, aunque con

margen de mejora en la optimización de trayectorias para escenarios donde un ajuste de rumbo podría acortar distancias de forma más eficiente.

5.2.2 Comportamiento en entorno con obstáculos

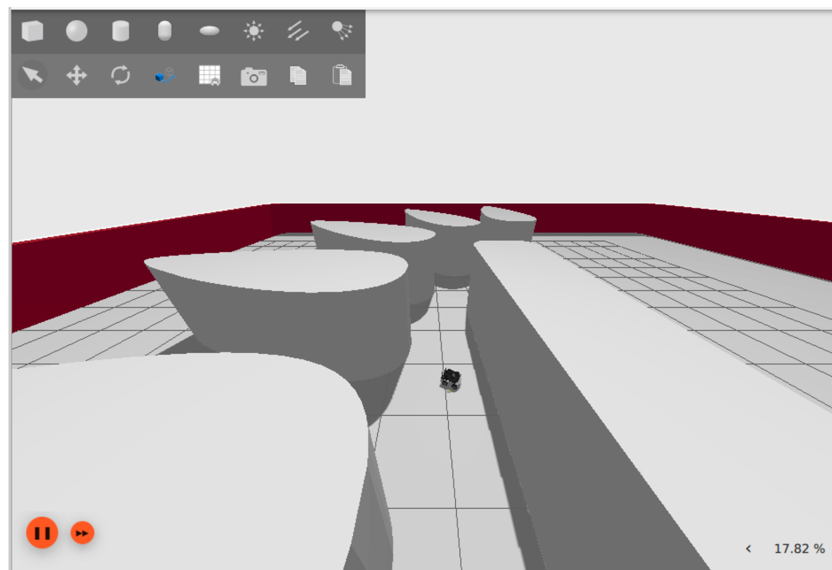
Se observó que, al acercarse a obstáculos en su camino, el ROSbot disminuía la velocidad y ejecutaba giros controlados para bordearlos, manteniendo el rumbo general hacia el objetivo siempre que fuera posible. Este comportamiento evidencia que el robot interpretó de forma efectiva las penalizaciones por colisiones y las recompensas por avances alineados, permitiéndole navegar en entornos con obstáculos distribuidos de manera heterogénea, incluso en pasillos estrechos, sin detenerse o quedar atascado en la mayoría de los casos.

Sin embargo, en algunos escenarios con geometrías complejas (como la letra "H"), el robot mostró limitaciones al priorizar trayectorias rectas en lugar de ejecutar giros amplios o retrocesos estratégicos que podrían haber optimizado su aproximación al objetivo. Esto refleja que la política PPO aprendida, si bien es robusta para evitar colisiones, no incorporó de forma suficiente estrategias de cambio de dirección pronunciado cuando la situación lo requería, lo que puede derivar en fallos al intentar alcanzar ciertos objetivos en entornos con obstáculos complejos.

En resumen, el ROSbot pudo navegar de forma autónoma en entornos con obstáculos, evitando colisiones, manteniendo el rumbo y mostrando un desplazamiento seguro y estable, validando la efectividad de las políticas y penalizaciones utilizadas durante el entrenamiento.

Figura 22

Comportamiento del ROSbot XL en entorno con obstáculos durante la validación.



Nota. Captura del simulador mostrando al ROSbot sorteando obstáculos mientras se desplaza hacia un objetivo, evidenciando la aplicación de la política de navegación aprendida.

5.2.3 Mecanismo de escape en atascos

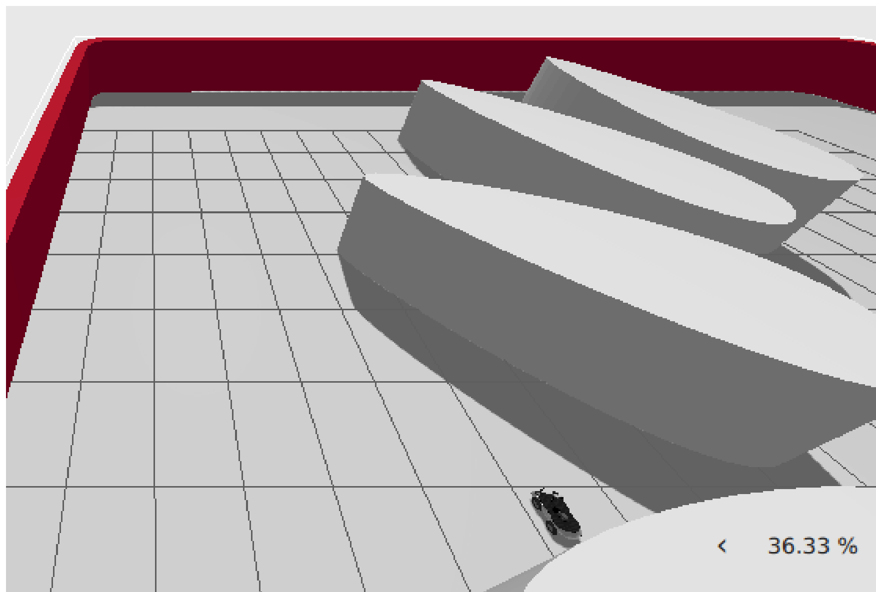
Durante las pruebas, se observó que el ROSbot aplicaba de forma efectiva el mecanismo de escape aprendido cuando detectaba falta de progreso en su trayectoria, activándose tras varios pasos sin reducción significativa de la distancia al objetivo. En estos casos, el robot ejecutaba pequeñas maniobras de retroceso y giros controlados para reajustar su orientación y liberar posibles bloqueos generados por la proximidad a obstáculos.

Este comportamiento fue consistente tanto en espacios abiertos con obstáculos aislados como en pasillos estrechos, donde el ROSbot detectaba que mantener la trayectoria recta no resultaba en progreso. Las acciones de escape le permitieron reanudar el avance de forma autónoma sin intervención externa, evitando bloqueos prolongados, validando la efectividad de la penalización incremental por atasco y de la recompensa por avances progresivos implementadas durante el entrenamiento.

En resumen, el mecanismo de escape en atascos demostró ser funcional y útil para mantener la autonomía del ROSbot XL durante la navegación, aunque se identifican oportunidades de mejora para reforzar su eficacia en entornos complejos donde se requiera mayor flexibilidad en la estrategia de liberación. En la Figura 23, se muestra el retroceso y escape inteligente del atasco.

Figura 23

Mecanismo de escape en atascos del ROSbot durante la validación.



Nota. Captura del simulador mostrando al ROSbot XL ejecutando maniobras de retroceso y giro para salir de un atasco, aplicando el mecanismo de escape aprendido durante el entrenamiento.

5.3 Análisis de episodios exitosos y fallidos

En este apartado se analizan los episodios ejecutados en la fase de validación (test) tras el entrenamiento del modelo, distinguiendo entre episodios exitosos y fallidos, con el objetivo de evaluar el comportamiento de la política aprendida, identificar patrones de éxito y reconocer limitaciones en entornos complejos.

5.3.1 Análisis de episodios exitosos

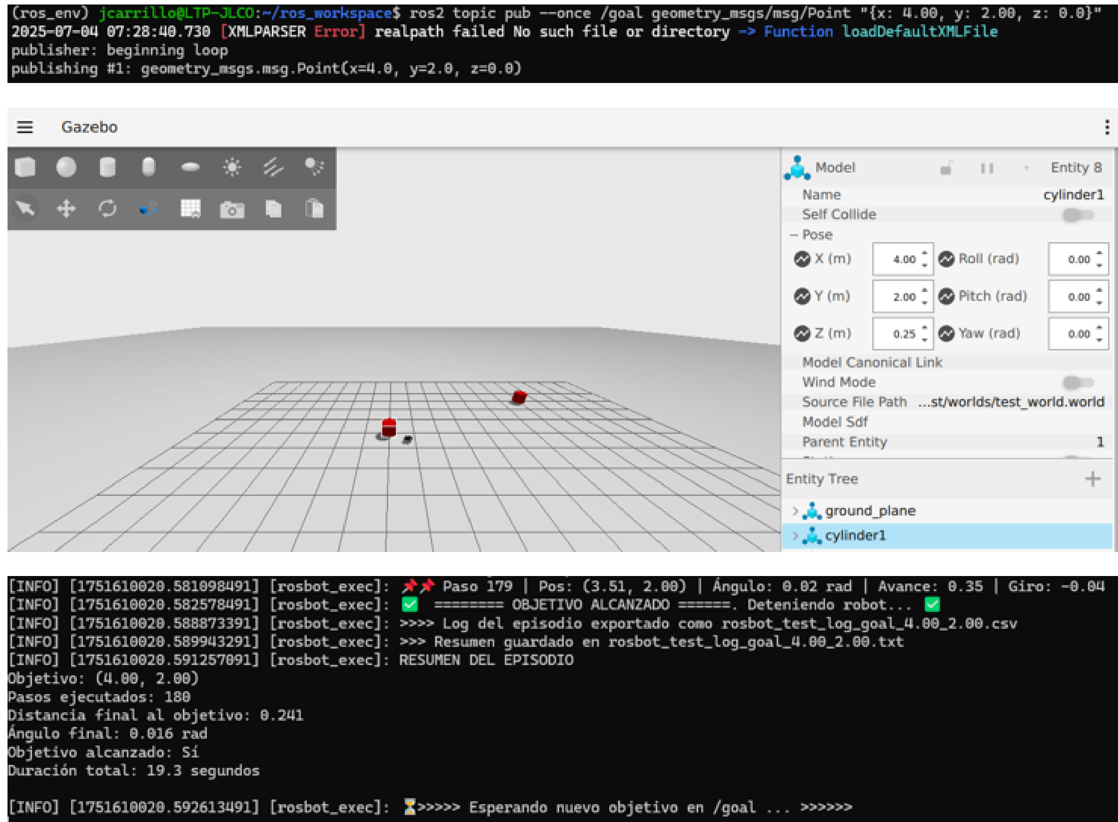
Durante la validación, se observó que el ROSbot logro alcanzar los objetivos, incluso en un nuevo escenario con cuatro cilindros como obstáculos distribuidos de forma heterogénea, diferente al mapa de entrenamiento de Husarion. Este aspecto es clave, ya que el robot no memorizó el mapa ni la disposición de los obstáculos, sino que actuó en base a las políticas aprendidas durante el entrenamiento, evaluando en tiempo real las observaciones del LIDAR y de la odometría para tomar decisiones autónomas.

Estos resultados reflejan que las penalizaciones por colisiones y el refuerzo positivo por avances alineados y progresivos aplicados durante el entrenamiento lograron consolidar una política de navegación robusta. Asimismo, se utilizó el mismo sistema de coordenadas basado en odometría empleado durante el entrenamiento, asegurando coherencia en la interpretación de la posición y el desplazamiento del robot durante las pruebas.

En las pruebas del modelo, se lanzaron 2 episodios de validación, previamente se obtuvieron las coordenadas de los cilindros para establecerlos como objetivos. En la Figura 24, el ROSbot logra alcanzar el objetivo ubicado en las coordenadas (4.00, 2.00), completando el episodio en 180 pasos con una distancia final de 0.241 m al objetivo y un ángulo final de 0.016 rad. Durante este episodio, el robot muestra avances progresivos hacia el objetivo, manteniendo alineaciones precisas y evitando colisiones con los cilindros, lo que evidencia un control efectivo de la orientación y de la trayectoria.

Figura 24

ROSbot alcanzando primer objetivo durante validación en un mundo distinto al entrenamiento

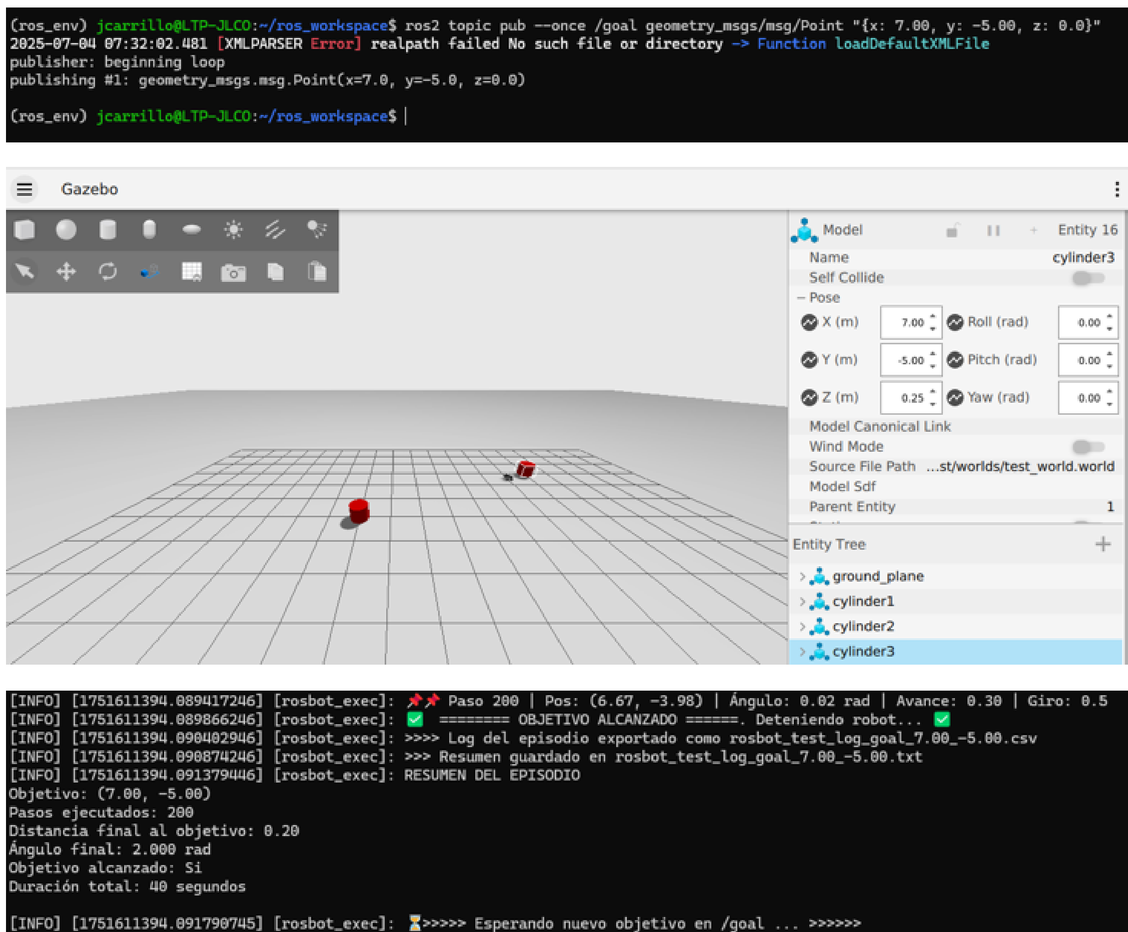


Nota. Captura del simulador mostrando al ROSbot XL en un episodio de validación alcanzando un objetivo tras avanzar de forma alineada y evitar obstáculos, aplicando las políticas aprendidas durante el entrenamiento.

Luego, en la Figura 25, se muestra otro episodio exitoso en el que el ROSbot alcanza un objetivo ubicado en las coordenadas (7.00, -5.00). En este caso, el episodio se completó en 200 pasos, con una distancia final al objetivo de 0.20 m y un ángulo final de 2.000 rad, alcanzando el objetivo tras 40 segundos de ejecución. Durante esta prueba, el robot ejecutó maniobras de avance y giro, realizando correcciones de orientación de forma autónoma en un entorno que no conocía previamente.

Figura 25

ROSbot XL alcanzando otro objetivo durante la validación del modelo.



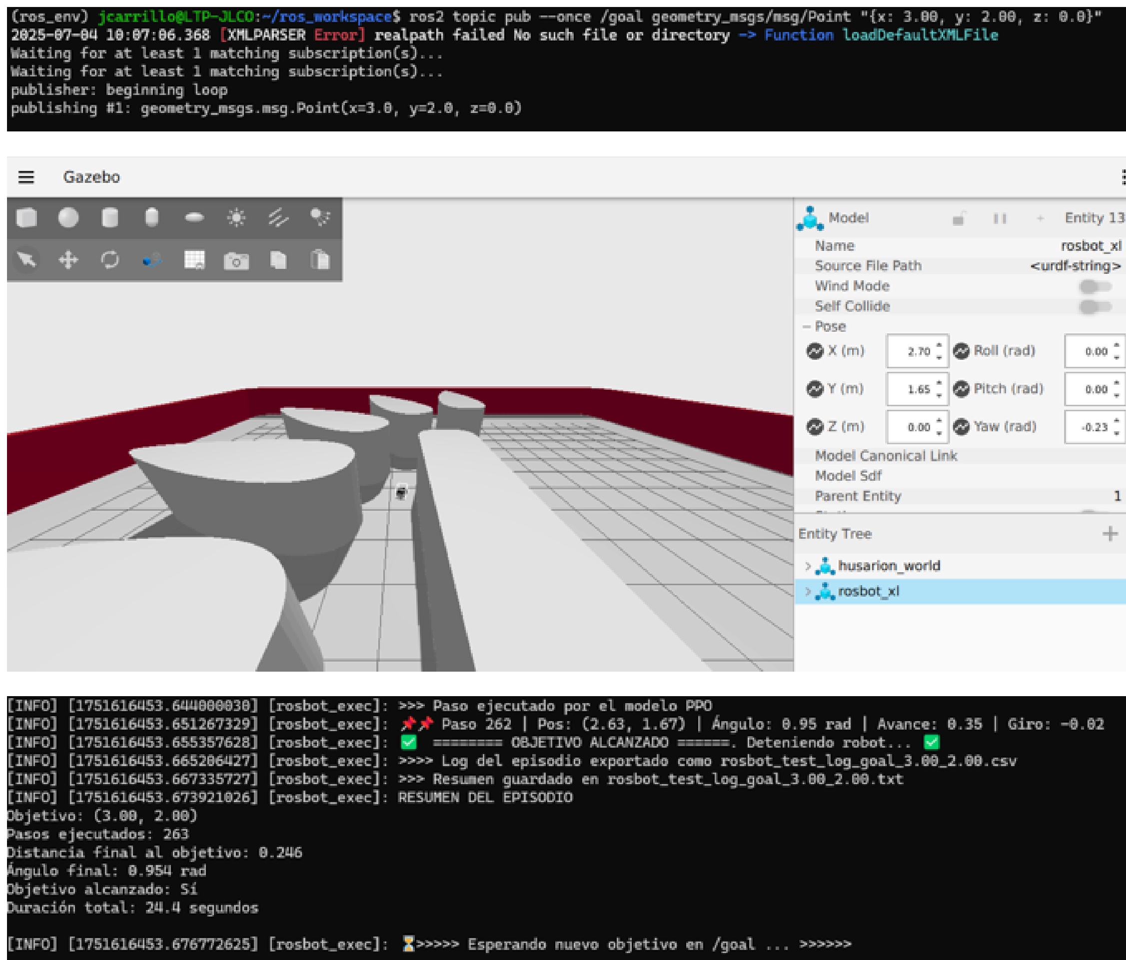
Nota. Captura del simulador mostrando al ROSbot XL alcanzando un objetivo en un episodio de validación, demostrando la capacidad de navegación autónoma en un entorno de prueba distinto al de entrenamiento.

Finalmente, se lanzó otro test en el mismo escenario de Husarion utilizado durante el entrenamiento para evaluar la capacidad del ROSbot de aplicar las políticas aprendidas en un entorno previamente explorado, sin que ello implicara una memorización de trayectorias exactas, sino confirmando el uso de estrategias de navegación basadas en la política PPO aprendida y el procesamiento en tiempo real de las observaciones del LIDAR y la odometría.

El objetivo se ubicó en las coordenadas (3.00, 2.00). El ROSbot alcanzó el objetivo tras 263 pasos ejecutados en 24.4 segundos, con una distancia final al objetivo de 0.246 m y un ángulo final de 0.954 rad, demostrando un control efectivo de la orientación y de la trayectoria, con maniobras de avance alineado, giros suaves y uso de la evasión de obstáculos en pasillos angostos. La Figura 26, muestra la evidencia de que el ROSbot ha alcanzado el objetivo en un escenario con obstáculos en pasillos, tras ejecutar un desplazamiento progresivo y controlado.

Figura 26

ROSbot XL alcanzando el objetivo en el escenario de entrenamiento.



Nota. El escenario de Husarion muestra al ROSbot alcanzando un objetivo, aplicando en tiempo real la política aprendida durante el entrenamiento, manteniendo coherencia en el sistema de referencia y demostrando desplazamiento autónomo con evasión de obstáculos.

5.3.2 Análisis de episodios fallidos

Durante las pruebas de validación, se realizaron 3 test en el mismo escenario de entrenamiento del ROSbot XL, con el objetivo de evaluar la capacidad del agente de alcanzar objetivos colocados en distintas posiciones dentro del entorno. A pesar de que el escenario era conocido en términos de geometría, el robot no memoriza rutas específicas debido a la naturaleza del aprendizaje por refuerzo, sino que toma decisiones en tiempo real basadas en las observaciones del LIDAR, la odometría y la política aprendida.

En estos tres episodios, el robot no logró alcanzar los objetivos establecidos, mostrando limitaciones que no contradicen el modelo ni la red configurada, sino que reflejan las características inherentes de la política entrenada, el rango de acciones y las recompensas aprendidas durante el entrenamiento, en conjunto con las dificultades que supone navegar en un entorno con pasillos estrechos y geometrías complejas.

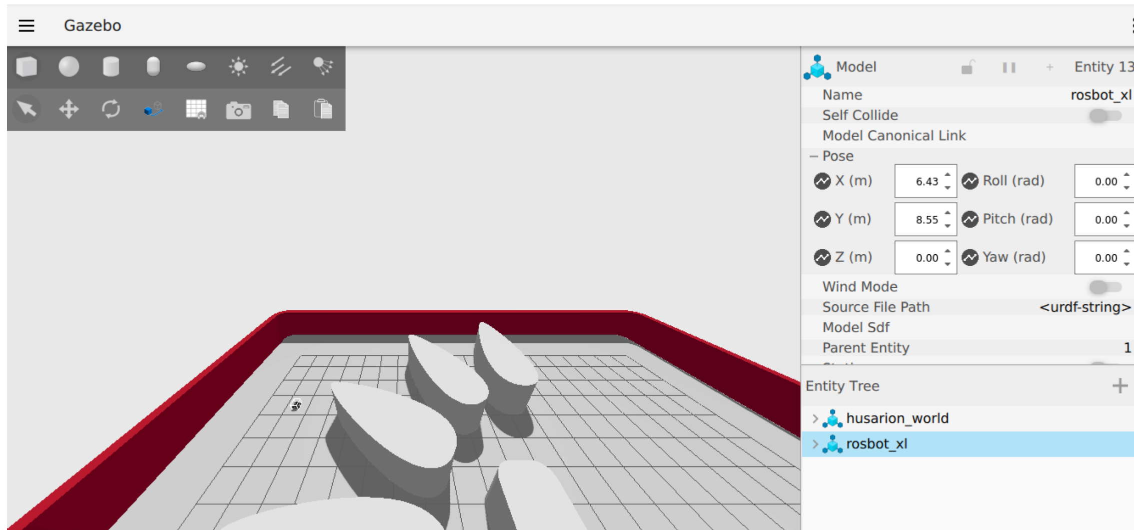
i. Primer objetivo con coordenadas en (6.43, 8.55)

En este primer caso, se estableció un objetivo en las coordenadas (como se aprecia en la Figura 27), situado en la parte superior izquierda del mapa. El ROSbot XL, al iniciar el episodio en las coordenadas (0,2), priorizó el avance en línea recta bordeando por la derecha del obstáculo frontal, en lugar de realizar un giro amplio hacia la izquierda para mejorar el ángulo al objetivo. El episodio, configurado con un límite de 1000 pasos, concluyó sin alcanzar el objetivo, con el robot detenido en la posición (10.54, -0.62), lejos de la meta.

Este comportamiento evidencia que la política prioriza la reducción de la distancia al objetivo de manera directa siempre que detecta trayectorias libres, evitando giros bruscos que podrían ser penalizados o interpretados como atascos por la política aprendida. No es un fallo del modelo, sino una manifestación de las prioridades aprendidas: avance progresivo, evitar colisiones y conservar alineación, incluso si se requiere un desvío amplio no previsto por la política en espacios amplios con rutas indirectas. En la Figura 28 se muestra la escena del incumplimiento del objetivo, y en la Figura 29 se muestra el log del episodio que evidencia el fallo.

Figura 27

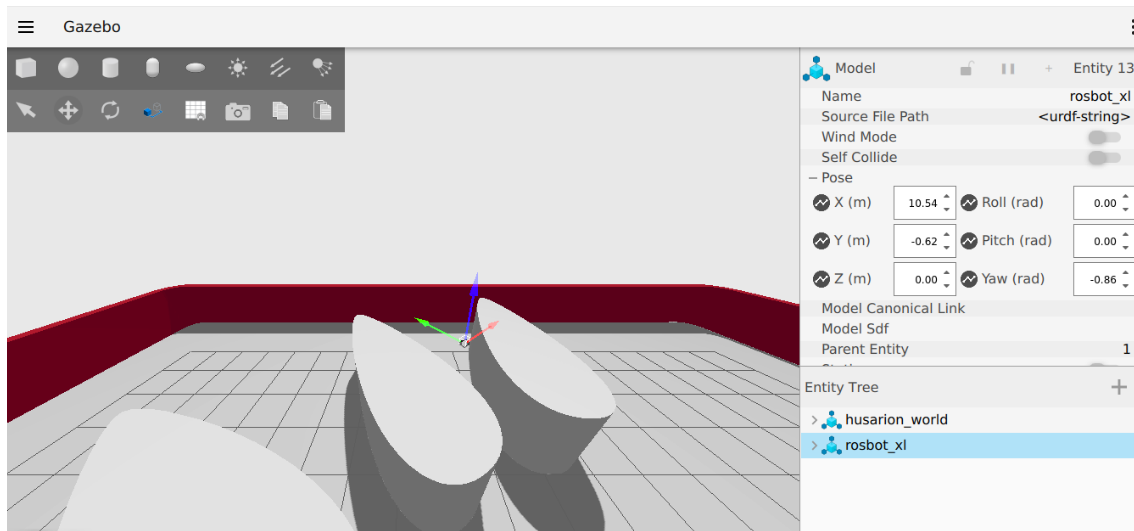
Coordenadas del objetivo (6.43, 8.55) que el ROSbot deberá llegar



Nota. La imagen muestra la posición del objetivo en el cual el robot deberá llegar, demostrando el aprendizaje obtenido.

Figura 28

Resultado del objetivo sin éxito en coordenadas (6.43, 8.55)



Nota. El ROSbot prioriza el avance en línea recta bordeando por la derecha, sin alcanzar el objetivo en el episodio de validación.

Figura 29

Log del episodio fallido en coordenadas (6.43, 8.55)

```
[INFO] [1751616730.075712477] [rosbot_exec]: >>> Paso ejecutado por el modelo PPO
[INFO] [1751616730.079708577] [rosbot_exec]: 📌📌 Paso 999 | Pos: (11.17, -2.75) | Ángulo: 2.82 rad | Avance: 0.35 | Giro: -0.04
[INFO] [1751616730.094915877] [rosbot_exec]: >>>> Log del episodio exportado como rosbot_test_log_goal_6.43_8.55.csv
[INFO] [1751616730.096104977] [rosbot_exec]: >>> Resumen guardado en rosbot_test_log_goal_6.43_8.55.txt
[INFO] [1751616730.097228477] [rosbot_exec]: RESUMEN DEL EPISODIO
Objetivo: (6.43, 8.55)
Pasos ejecutados: 1000
Distancia final al objetivo: 150.160
Ángulo final: 2.825 rad
Objetivo alcanzado: No
Duración total: 94.3 segundos

[INFO] [1751616730.099854277] [rosbot_exec]: ⏳>>>>> Esperando nuevo objetivo en /goal ... >>>>>>
```

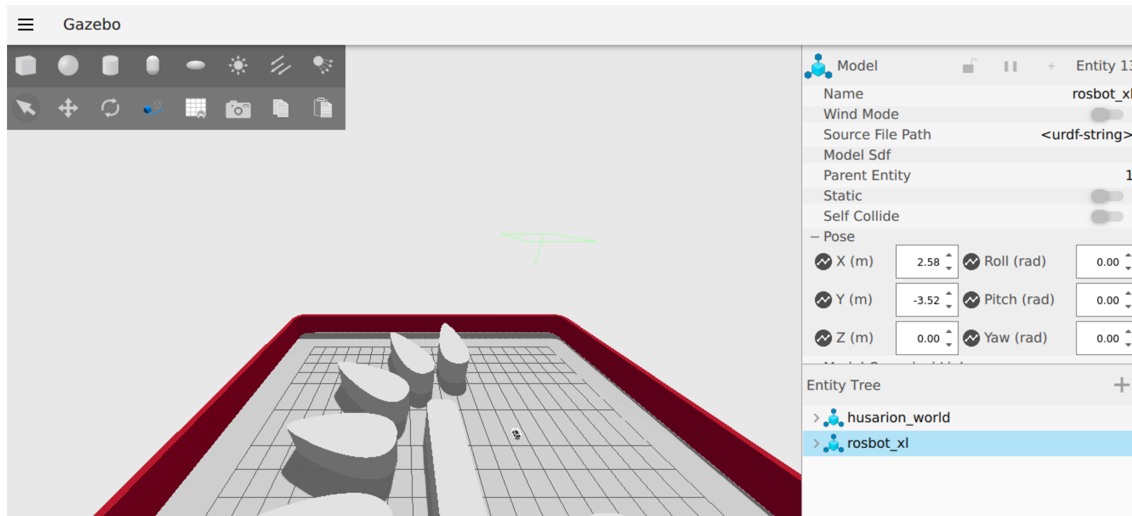
Nota. Registro del episodio mostrando los pasos ejecutados, la distancia final y el ángulo, confirmando que el robot no alcanzó el objetivo tras completar los pasos.

ii. Segundo objetivo con coordenadas en (2.58, -3.52)

El segundo el objetivo se encontraba localizado al sur del punto de inicio (como se aprecia en la Figura 30). El ROSbot XL partió correctamente alineado, pero al aproximarse a la intersección con la geometría de la letra “H” en el escenario, en lugar de girar para ajustar su trayectoria y dirigirse hacia el objetivo, continuó hacia la derecha, priorizando una trayectoria recta que le impidió rodear el obstáculo y llegar a la meta. Este resultado muestra que la política aprendida tiende a mantener trayectorias rectas en presencia de obstáculos, en lugar de ejecutar maniobras de giro agresivo si no se percibe una penalización inmediata, y resalta la necesidad de ajustar la política o las recompensas si se requiere priorizar maniobras de cambio de dirección más pronunciadas en pasillos complejos. En la Figura 31 se muestra la escena donde el ROSbot no llega al objetivo, y en la Figura 32 se presenta el log del episodio fallido.

Figura 30

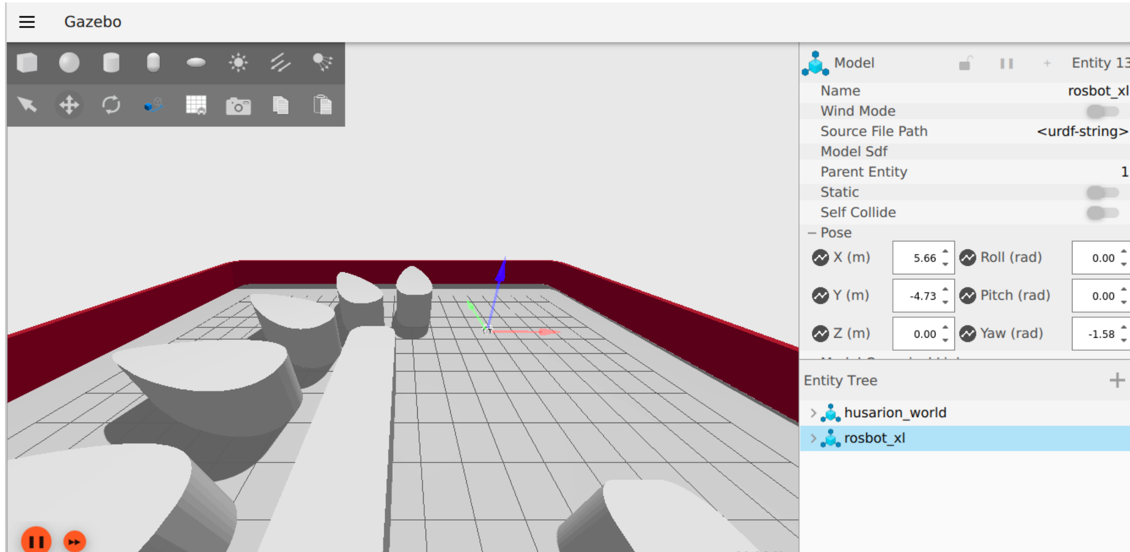
Coordenadas del objetivo (2.58, -3.52) que el ROSbot deberá llegar



Nota. La imagen muestra la posición del objetivo en el cual el robot deberá llegar, demostrando el aprendizaje obtenido.

Figura 31

Resultado del objetivo sin éxito en coordenadas (2.58, -3.52)



Nota. El ROSbot XL se dirige de forma recta desviándose de la trayectoria necesaria para alcanzar el objetivo al sur del escenario.

Figura 32

Log del episodio fallido en (2.58, -3.52).

```
[INFO] [1751618281.533611620] [rosbot_exec]: 📌📌 Paso 998 | Pos: (5.73, -4.78) | Ángulo: -1.96 rad | Avance: 0.35 | Giro: -0.05
[INFO] [1751618281.646792637] [rosbot_exec]: >>> Paso ejecutado por el modelo PPO
[INFO] [1751618281.660553239] [rosbot_exec]: 📌📌 Paso 999 | Pos: (5.73, -4.78) | Ángulo: -1.96 rad | Avance: 0.35 | Giro: -0.04
[INFO] [1751618281.712989847] [rosbot_exec]: >>>> Log del episodio exportado como rosbot_test_log_goal_2.58_-3.52.csv
[INFO] [1751618281.714766547] [rosbot_exec]: >>> Resumen guardado en rosbot_test_log_goal_2.58_-3.52.txt
[INFO] [1751618281.718675348] [rosbot_exec]: RESUMEN DEL EPISODIO
Objetivo: (2.58, -3.52)
Pasos ejecutados: 1000
Distancia final al objetivo: 11.532
Ángulo final: -1.959 rad
Objetivo alcanzado: No
Duración total: 97.2 segundos
```

Nota. Registro del episodio mostrando los pasos ejecutados y la trayectoria seguida, sin éxito en alcanzar el objetivo.

iii. Tercer objetivo con coordenadas en (-6.13, 2.77)

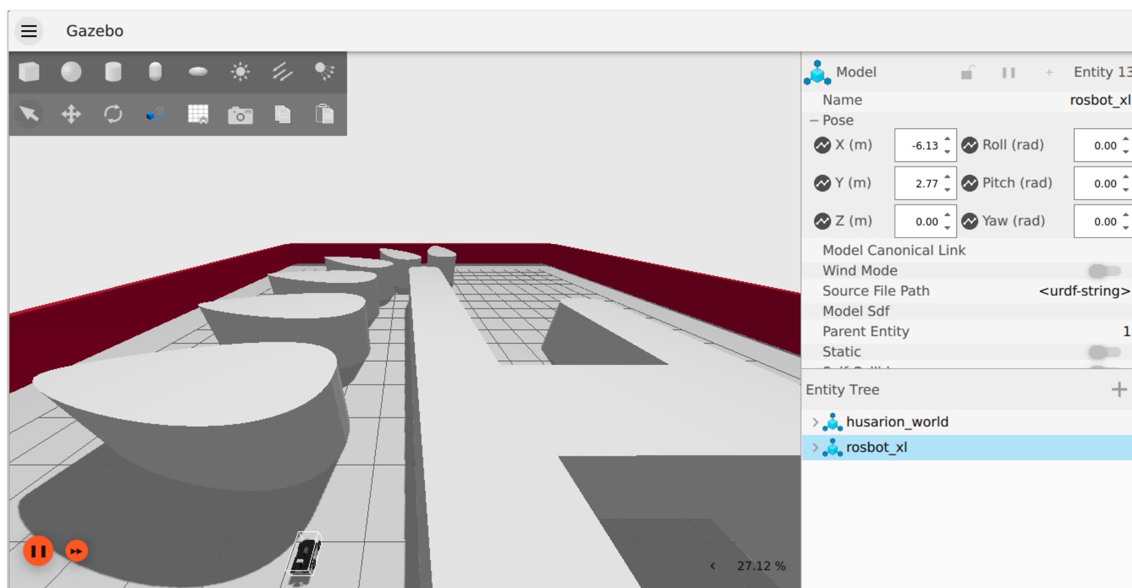
Se seleccionó un objetivo detrás del punto de inicio del robot (como se aprecia en Figura 33), en las coordenadas (-6.13, 2.77). Este test se ejecutó con un límite de 3000 pasos para analizar la capacidad del robot de priorizar el retroceso o el giro completo cuando el objetivo se encuentra detrás.

El ROSbot priorizó el avance en línea recta hacia adelante, sin importar que el objetivo se encontraba detrás, avanzando de forma indefinida y quedando desalineado respecto al objetivo, resultando en un episodio fallido.

Este comportamiento es consistente con la política aprendida, que no incluyó retrocesos prolongados como estrategia de navegación ni priorizó objetivos detrás del robot en fases de entrenamiento, dado que las recompensas configuradas y las acciones permitidas favorecían el avance progresivo con alineación hacia adelante. En la Figura 34 se muestra la escena donde el ROSbot no llegó al objetivo, y en la Figura 35 el log del episodio.

Figura 33

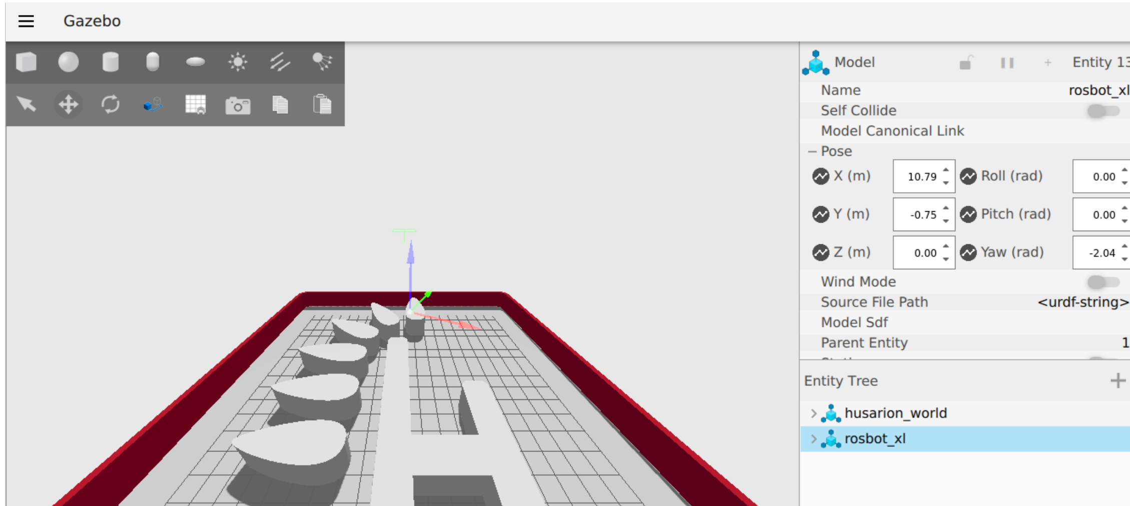
Coordenadas del objetivo (-6.13, 2.77) *que el ROSbot deberá llegar*



Nota. La imagen muestra la posición del objetivo en el cual el robot deberá llegar, demostrando el aprendizaje obtenido.

Figura 34

Resultado del objetivo sin éxito en coordenadas (-6.13, 2.77)



Nota. El ROSbot continúa avanzando de forma recta sin girar hacia atrás para alcanzar el objetivo, reflejando las limitaciones de la política aprendida.

Figura 35

Log del episodio fallido en (-6.13, 2.77)

```
[INFO] [1751620525.699893003] [rosbot_exec]: 📌📌 Paso 2999 | Pos: (4.72, -25.50) | Ángulo: -2.30 rad | Avance: 0.35 | Giro: -0.03
[INFO] [1751620525.755769195] [rosbot_exec]: >>>> Log del episodio exportado como rosbot_test_log_goal_-6.13_2.77.csv
[INFO] [1751620525.758025494] [rosbot_exec]: >>> Resumen guardado en rosbot_test_log_goal_-6.13_2.77.txt
[INFO] [1751620525.761378994] [rosbot_exec]: RESUMEN DEL EPISODIO
Objetivo: (-6.13, 2.77)
Pasos ejecutados: 3000
Distancia final al objetivo: 916.833
Ángulo final: -2.296 rad
Objetivo alcanzado: No
Duración total: 272.1 segundos
```

Nota. Registro del episodio mostrando la falta de alineación con el objetivo ubicado detrás del punto de inicio, resultando en un fallo al alcanzar la meta.

Análisis global de resultados

Estos tres episodios fallidos no contradicen el modelo ni la red configurada, sino que exponen los límites naturales de la política aprendida bajo la configuración de recompensas, penalizaciones y el rango de acciones establecido en el entrenamiento. Los resultados reflejan que el robot prioriza la seguridad (evitando colisiones), el avance progresivo y la alineación frontal al objetivo.

Asimismo, que no se entrenó específicamente para objetivos detrás del punto de inicio ni para giros agresivos en entornos con geometrías complejas. Finalmente, que el sistema funciona de manera coherente con los principios del aprendizaje por refuerzo, donde el comportamiento emerge de las recompensas y penalizaciones configuradas.

Estos hallazgos ofrecen oportunidades de mejora en iteraciones futuras, como el ajuste de recompensas por realineación en escenarios complejos, ampliación del rango de acciones para maniobras de retroceso estratégicas o el entrenamiento adicional en entornos con objetivos traseros para robustecer la política.

5.3.3 Lecciones aprendidas

La etapa de validación del modelo permitió la caracterización concreta de los puntos fuertes y débiles de la política de navegación aprendida por el ROSbot, aportando valiosas lecciones para las futuras iteraciones, así como una comprensión sólida del comportamiento del agente bajo escenarios reales de prueba.

Por un lado, los episodios exitosos demostraron que el agente pudo ejecutar eficazmente la política aprendida durante el entrenamiento, avanzando suavemente sin colisionar y buscando una dirección específica mientras se dirigía hacia el objetivo en entornos con obstáculos. Este comportamiento se observó en el escenario de entrenamiento, pero también en uno diferente con cilindros colocados en posiciones fijas dentro del mundo, esto con el fin de validar que el robot no simplemente memoriza rutas o mapas, sino que actúa siguiendo las observaciones de LIDAR y la odometría, aplicando así en tiempo real la regla aprendida de que el avance seguro y la alineación efectiva son esenciales.

Por el contrario, los episodios fallidos arrojaron información valiosa sobre las capacidades y limitaciones de la política entrenada. Se identificó que:

i. El agente tiende a priorizar trayectorias rectas incluso si un giro amplio o un cambio de dirección podría acercarlo al objetivo.
ii. No se cuenta con estrategias eficientes para manejar objetivos ubicados detrás del punto de inicio, ya que la política aprendida no prioriza retrocesos prolongados como mecanismo de aproximación a la meta.
iii. La geometría compleja del entorno (como la letra “H”) puede generar situaciones donde el robot sigue rutas que mantienen la alineación general, pero lo alejan de la meta al no realizar maniobras agresivas de giro, debido a la configuración de recompensas y penalizaciones durante el entrenamiento.

Estos resultados no contradicen el modelo ni la red utilizada, sino que reflejan que el agente actúa de manera coherente con la política aprendida, priorizando el avance progresivo, la alineación hacia el objetivo y la seguridad frente a colisiones. Sin embargo, evidencian áreas específicas donde la política puede ser refinada para enfrentar de forma más eficaz casos de objetivos en posiciones complejas o escenarios que requieren cambios de estrategia.

5.4 Discusión critica

Los resultados obtenidos tras el entrenamiento y la validación del modelo confirman que el enfoque de aprendizaje por refuerzo profundo, junto con un diseño cuidadoso del espacio de observación, la función de recompensa y la arquitectura de la política, permite entrenar agentes capaces de aprender comportamientos de navegación autónoma seguros y eficientes en entornos con obstáculos complejos. El agente logró alcanzar los objetivos en escenarios conocidos y nuevos, aplicando políticas de avance progresivo, corrección de orientación y evasión de obstáculos, emergentes a partir de las recompensas definidas y sin necesidad de rutas preprogramadas.

Durante las pruebas, el ROSbot demostró comportamientos no triviales como mantener la alineación antes de avanzar, reducir velocidad y desviarse suavemente al detectar obstáculos, y activar mecanismos de desbloqueo tras detectar falta de progreso, reflejando que la política aprendida prioriza tanto la seguridad como la eficiencia de desplazamiento.

No obstante, las validaciones también permitieron identificar limitaciones que abren oportunidades de mejora:

i. El agente tiende a detenerse o mantener trayectorias rectas en corredores angostos si percibe riesgo de colisión, evitando maniobras de giro agresivas para conservar la seguridad.
ii. Se observaron dificultades para aproximarse a objetivos ubicados muy cerca de paredes o detrás del punto de inicio, debido a que la política aprendida favorece el avance frontal y carece de retrocesos prolongados como estrategia de navegación.
iii. En ciertos escenarios, se detectaron oscilaciones en la orientación antes de avanzar, producto de la priorización de la alineación y del uso exclusivo de percepciones locales sin planificación global.

En conjunto, los resultados confirman que el modelo entrenado logró dotar al ROSbot XL de capacidades de navegación autónoma, demostrando que es posible emplear aprendizaje por refuerzo en robótica móvil bajo ROS 2 y Gazebo para resolver tareas de desplazamiento en entornos no estructurados, utilizando únicamente percepciones sensoriales en tiempo real.

6 Conclusiones

La realización de este TFM pone de manifiesto que el aprendizaje por refuerzo es una opción viable y eficaz para instruir sistemas de navegación autónoma en robots móviles. Esto se ha demostrado utilizando el ROSbot XL en simulaciones bajo ROS 2 y Gazebo Ignition. Gracias al uso de PPO (Proximal Policy Optimization) y una estructura MLP, junto con una definición meticulosa de la observación, la recompensa y las sanciones, se pudo entrenar un modelo que permite al robot moverse con seguridad y eficiencia en entornos con obstáculos.

Los datos obtenidos muestran que el modelo desarrolló efectivamente habilidades de navegación autónoma, ya que logró:

i. Adquirir comportamientos de navegación complejos de forma espontánea, priorizando la alineación antes de avanzar, realizando maniobras para esquivar obstáculos con soltura y activando estrategias para salir de situaciones de bloqueo, manteniendo así el movimiento sin ayuda externa.
ii. Lograr objetivos de forma autónoma tanto en entornos ya conocidos como en escenarios nuevos con obstáculos, operando sin necesidad de mapas guardados y basándose solo en lo que percibe el LIDAR y la odometría, lo que confirma que el agente puede tomar decisiones en tiempo real de manera consistente y segura durante la navegación.

Además, se ha verificado que el uso de la odometría como referencia de posición, que fue constante durante el entrenamiento y la validación, favoreció la estabilidad del aprendizaje y la posibilidad de replicar los resultados.

Sin embargo, el proyecto también permitió detectar ciertas limitaciones relacionadas con el enfoque adoptado:

1. Problemas para alcanzar objetivos situados detrás del punto de partida, debido a que la política entrenada carece de estrategias para retroceder de forma prolongada.
2. Inclinación a mantener trayectorias rectas, incluso cuando un giro amplio habría sido más eficiente en ciertas situaciones.
3. Oscilaciones en la orientación antes de avanzar en escenarios complicados, lo que se debe a la prioridad dada a la alineación y a la percepción local sin una planificación global.

Estas limitaciones no invalidan la efectividad del modelo, sino que simplemente reflejan el carácter cauto de la política aprendida y el enfoque local de la estructura utilizada, que prima la seguridad y la estabilidad en el avance.

En general, el proyecto ha cumplido su propósito de demostrar el potencial del aprendizaje por refuerzo en tareas de navegación autónoma con robótica móvil, estableciendo una base firme para su desarrollo futuro en entornos reales.

7 Bibliografía

Arulkumaran, K., Deisenroth, M. P., Brundage, M., & Bharath, A. A. (2017). A brief survey of deep reinforcement learning. IEEE Signal Processing Magazine, 34(6), 26-38. https://doi.org/10.1109/MSP.2017.2743240

Berner, C., Brockman, G., Chan, B., Cheung, V., Debiak, P., Dennison, C., ... & Zhang, S. (2019). Dota 2 with large scale deep reinforcement learning. *arXiv preprint arXiv:1912.06680*.

Chen, Y. F., Liu, M., Everett, M., & How, J. P. (2019). Socially aware motion planning with deep reinforcement learning. In *IROS 2017*. https://arxiv.org/abs/1703.08862

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep learning*. MIT Press.

Gupta, S., Davidson, J., Levine, S., Sukthankar, R., & Malik, J. (2017). Cognitive mapping and planning for visual navigation. Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2616-2625. https://doi.org/10.1109/CVPR.2017.280

Hausknecht, M., & Stone, P. (2015). Deep recurrent Q-learning for partially observable MDPs. arXiv preprint arXiv:1507.06527. https://arxiv.org/abs/1507.06527

Henderson, P., Islam, R., Bachman, P., Pineau, J., Precup, D., & Meger, D. (2018). Deep reinforcement learning that matters. *Proceedings of the AAAI Conference on Artificial Intelligence*, 32(1).

Husarion. (2024). ROSbot XL – autonomous mobile robot platform. Husarion. https://husarion.com/manuals/rosbot-xl/

Hwangbo, J., Lee, J., Dosovitskiy, A., Bellicoso, C. D., Tsounis, V., Koltun, V., & Hutter, M. (2019). Learning agile and dynamic motor skills for legged robots. Science Robotics, 4(26), eaau5872. https://doi.org/10.1126/scirobotics.aau5872

Kiran, B. R., Sobh, I., Talpaert, V., Mannion, P., et al. (2022). *Deep Reinforcement Learning for Autonomous Driving: A Survey*. IEEE Transactions on Intelligent Transportation Systems, 23(6), 4909–4926.

Koenig, N., & Howard, A. (2004). Design and use paradigms for Gazebo, an open-source multi-robot simulator. In *2004 IEEE/RSJ IROS* (Vol. 3, pp. 2149–2154). IEEE.

Maruyama, Y., Kato, S., & Azumi, T. (2016). Exploring the performance of ROS2. Proceedings of the 13th International Conference on Embedded Software (EMSOFT), 1–10. https://doi.org/10.1145/2968478.2968502

Macenski, S., White, R., Gin, D., & Christ, S. (2022). Robot Operating System 2: Design, architecture, and uses in the wild. Science Robotics, 7(66), eabm6074. https://doi.org/10.1126/scirobotics.abm6074

Mirowski, P., et al. (2017). Learning to navigate in complex environments. arXiv preprint arXiv:1611.03673. https://arxiv.org/abs/1611.03673

Mousavi, S. S., Schukat, M., & Howley, E. (2018). Deep reinforcement learning: An overview. arXiv preprint arXiv:1806.08894. https://arxiv.org/abs/1806.08894

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., et al. (2015). *Human-level control through deep reinforcement learning*. Nature, 518(7540), 529–533.

Lillicrap, T. P., Hunt, J. J., Pritzel, A., et al. (2016). *Continuous control with deep reinforcement learning*. arXiv preprint arXiv:1509.02971.

Li, Y., Zheng, J., & Ma, Y. (2022). *Multi-goal mobile robot navigation using SAC in Unity3D environments*. *Simulation Modelling Practice and Theory*, 114, 102–118.

Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., ... & Ng, A. Y. (2009). ROS: an open-source Robot Operating System. ICRA Workshop on Open Source Software, 3(3.2), 5.

Russell, S., & Norvig, P. (2020). *Artificial Intelligence: A Modern Approach* (4th ed.). Pearson.

Sadeghi, F., & Levine, S. (2017). CAD2RL: Real single-image flight without a single real image. Robotics: Science and Systems (RSS). https://doi.org/10.15607/RSS.2017.XIII.034

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). *Proximal Policy Optimization Algorithms*. arXiv preprint arXiv:1707.06347.

Sutton, R. S., & Barto, A. G. (2018). *Reinforcement Learning: An Introduction* (2nd ed.). MIT Press. https://www.andrew.cmu.edu/course/10-703/textbook/BartoSutton.pdf

Taheri, H., Hosseini, S. R., & Nekoui, M. A. (2024). *Deep Reinforcement Learning with Enhanced PPO for Safe Mobile Robot Navigation*. arXiv. https://arxiv.org/abs/2405.16266

Tai, L., Paolo, G., & Liu, M. (2017). Virtual-to-real deep reinforcement learning: Continuous control of mobile robots for mapless navigation. In *IROS 2017*. https://doi.org/10.1109/IROS.2017.8202133

Tobin, J., Fong, R., Ray, A., Schneider, J., Zaremba, W., & Abbeel, P. (2017). Domain randomization for transferring deep neural networks from simulation to the real world. 2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), 23–30. https://doi.org/10.1109/IROS.2017.8202133

Wang, B., Liu, Z., & Li, Q. (2020). *Mobile Robot Path Planning in Dynamic Environments through Globally Guided Reinforcement Learning*. arXiv. https://arxiv.org/abs/2005.05420

Zhang, J., & Singh, S. (2014). LOAM: Lidar Odometry and Mapping in Real-time. In *Robotics: Science and Systems Conference*.

Zhu, Y., et al. (2017). Target-driven visual navigation in indoor scenes using deep reinforcement learning. 2017 IEEE International Conference on Robotics and Automation (ICRA), 3357-3364. https://doi.org/10.1109/ICRA.2017.7989381

8 Anexos

8.1 Configuración del modelo PPO y entrenamiento

El archivo “train_ppo.py”, contiene la configuración de la política PPO, los hiperparámetros seleccionados y la estructura de entrenamiento progresivo con checkpoints, alineado con la explicación del Capítulo 4.

```
# Instanciar el modelo PPO
model = PPO(
    policy="MlpPolicy",
    env=env,
    verbose=1,
    learning_rate=0.0001,
    n_steps=2048, # Cada 2048 pasos , el modelo revisa lo aprendido, calcula ventajas, y ajusta la red neuronal.
    batch_size=64,
    gamma=0.99,
    gae_lambda=0.95,
    clip_range=0.2,
    ent_coef=0.02 ,  # Estimula explorar acciones nuevas
    tensorboard_log="./rosbot_tensorboard_v4/"
)
```

```
# Entrenamiento en 5 fases de 40 000 pasos con checkpoints
total_phases = 5
steps_per_phase = 40_000

for i in range(total_phases):
    training_node.get_logger().info(f">>>> Fase {i+1}/{total_phases} - Entrenando {steps_per_phase} pasos...")
    model.learn(total_timesteps=steps_per_phase, reset_num_timesteps=False)
    checkpoint_name = f"rosbot_model_v4_checkpoint_{i+1}"
    model.save(checkpoint_name)
    training_node.get_logger().info(f"✅ Checkpoint guardado: {checkpoint_name}.zip")

# Guardar modelo completo
model.save("rosbot_model_v4")
training_node.get_logger().info("✅ >>> Entrenamiento completado, modelo guardado como 'rosbot_model_v4.zip'")

rclpy.shutdown()
```

8.2 Estructura del entorno de entrenamiento

El archivo “rosbot_env.py”, contiene la reducción del LIDAR a 37 medidas y cómo se define el espacio de acción de velocidad lineal y angular. Asimismo, la asignación de recompensas y penalizaciones.

```
# LIDAR : 240 medidas de frente (-30° a +30°) + resto cada 30 indices = 280 medidas
self.laser_ranges = np.ones(TOTAL_LIDAR_MEASUREMENTS) * LIDAR_MAX_RANGE
self.state = np.append([self.angle, self.distance2], self.laser_ranges)
self.observation_space = spaces.Box(low=0.0, high=LIDAR_MAX_RANGE, shape=(OBSERVATION_DIM,), dtype=np.float64) # LI

# Acción: velocidad lineal y angular , Retroceder lentamente , combinar retroceso con giro
self.action_space = spaces.Box(low=np.array(ACTION_LOW), high=np.array(ACTION_HIGH), dtype=np.float64)

self.node.get_logger().info("✅  Entorno Rosbot inicializado...")
```

```python
def laser_callback(self, msg):
    #self.node.get_logger().info("!!!!!!!!!!! LASER callback invocado !!!!!!!!!!")
    self.laser_received = True

    # Índices de LIDAR frontal y lateral , usando índices circulares
    front_indices = list(range(1320, 1440)) + list(range(0, 120)) # -30° a +30° → 240 indices (1 grado = 4 medidas) , R
    side_indices = list(range(0, 1440, 36))  # Resto del entorno 40 indices , =>> cada 36 índices =  9°
    side_indices = [i for i in side_indices if i < 1320 or i >= 120] # excluir duplicados
    indices = front_indices + side_indices
    selected_ranges = np.array(msg.ranges)[indices]
    selected_ranges[selected_ranges == 0.0] = np.nan # Filtrado del LIDAR para falsos positivos

    # 240 índices de front, agrupados de 8 en 8 → 30 bloques → índice con el menor valor LIDAR por bloque.
    # 40 índices de laterales, agrupados de 8 en 8 → 5 bloques → índice con el menor valor LIDAR por bloque.
    reduced_ranges = [
        np.nanmin(selected_ranges[i:i + 8])
        for i in range(0, len(selected_ranges), 8)
    ]

    self.laser_ranges = np.nan_to_num(reduced_ranges, nan=LIDAR_MAX_RANGE, posinf=LIDAR_MAX_RANGE, neginf=0.0)
    self.laser_ranges = np.clip(self.laser_ranges, 0.0, LIDAR_MAX_RANGE)
```

```python
    # *************************
    # RECOMPENSAS:
    # *************************

    # Recompensa por llegar al  objetivo
    if self.distance2 < GOAL_REACHED_THRESHOLD **2:
        reward += REWARD_GOAL_REACHED
        done = True
        self.change_goal_on_reset = True #Cambiar objetivo
        self.node.get_logger().info(f" ✅ ✅ ✅ Objetivo alcanzado. episodio finalizado Distancia²={self.distance2:.4f}

    # Recompensa por reducir el ángulo hacia el objetivo (orientación inteligente)
    if not hasattr(self, "prev_angle"):
        self.prev_angle = abs(self.angle)

    angle_diff = self.prev_angle - abs(self.angle)
    if angle_diff > 0.01:
        reward += 100.0 * angle_diff
        self.node.get_logger().info(f"➡ Reducción de ángulo hacia objetivo. Δángulo = {angle_diff:.2f}")
    self.prev_angle = abs(self.angle)

    # Bonus si está bien alineado
    if abs(self.angle) < 0.1:
        reward += REWARD_WELL_ALIGNED
    if abs(self.angle) < 0.1 and forward > 0.05:
        reward += REWARD_STEADY_ALIGNMENT
```

```python
# *************************
# PENALIZACIONES:
# *************************

if dist_front < COLLISION_THRESHOLD:  #Si hay un obstáculo muy cerca (< 20 cm)
    reward -= PENALTY_COLLISION
    done = True
    self.change_goal_on_reset = False
    self.node.get_logger().warn(f"❌  Colisión detectada. dist_front={dist_front:.2f}")
elif dist_front < NEAR_OBSTACLE_THRESHOLD:
    reward -= (0.3 - dist_front) * 1200.0   # Muy cerca
    self.node.get_logger().warn(f"⚠ Muy cerca de obstáculo. dist_front={dist_front:.2f}")
```

```python
# Penalización por inactividad
if dist_front < 0.25 and delta_d <= 0.002:
    self.stuck_counter += 1
    reward -= PENALTY_STUCK_INCREMENT * self.stuck_counter  # penalización progresiva
    if repeated_actions:
        reward -= PENALTY_REPEATED_ACTION  # penalizacion adicional por no intentar nada nuevo
        self.node.get_logger().warn("⚠️ Repeticion de acciones detectada durante atasco.")
else:
    if self.stuck_counter >= self.stuck_limit:
        reward += REWARD_ESCAPE_SUCCESS  # recompensa por salir del atasco
        self.node.get_logger().info("➡️ Salida inteligente del atasco. Recompensa otorgada.")
    self.stuck_counter = 0

# Penalización si está mal orientado pero no gira
if abs(self.angle) > 1.0 and abs(rotation) < 0.1:
    reward -= PENALTY_BAD_ORIENTATION_NO_ROTATION
    self.node.get_logger().warn("⚠️ Mal orientado y sin corrección.")

# Penalizar giros si ya está perfectamente alineado
if abs(self.angle) < 0.03 and rotation != 0:
    reward -= 100.0

# Penalización por retroceder cuando está bien alineado
if forward < 0.0 and abs(self.angle) < 0.2:
    reward -= PENALTY_REVERSE_WHEN_ALIGNED
```

8.3 Ejecución y validación del modelo entrenado

El archivo "test_rosbot.py", define cómo se evalúa el modelo entrenado sobre el ROSbot XL, aplicando la política aprendida en simulación en tiempo real.

```python
def main(args=None):
    rclpy.init(args=args)
    executor_node  = RosbotExec()
    model = PPO.load("rosbot_model_v4")
    executor_node.node.get_logger().info(">>>>> Modelo v4 PPO cargado exitosamente. >>>>")
```

```python
def ejecutar_episodio():
    done = False
    step_count = 0
    MAX_STEPS = 3000
    log_data = []
    start_time = time.time()

    while rclpy.ok() and not done and step_count < MAX_STEPS:
        if step_count == 0:
            executor_node.node.get_logger().info(f" 🔍 Estado inicial >> pos=({executor_node.robot_x:.2f}, {executor

        executor_node.laser_received = False
        executor_node.odom_received = False
        executor_node.future = Future()
        rclpy.spin_until_future_complete(executor_node.node, executor_node.future, timeout_sec=0.1)
        executor_node.node.get_logger().info(">>> Paso ejecutado por el modelo PPO")

        obs = executor_node.state.reshape(1, -1)
        action, _ = model.predict(obs, deterministic=True)
        forward, rotation = action[0]

        twist = Twist()
        twist.linear.x = forward
        twist.angular.z = rotation
        executor_node.publisher.publish(twist)
        executor_node.node.get_logger().info(
        f"📌📌 Paso {step_count:03d} | Pos: ({executor_node.robot_x:.2f}, {executor_node.robot_y:.2f}) | "
        f"Ángulo: {executor_node.angle:.2f} rad | Avance: {forward:.2f} | Giro: {rotation:.2f}"
        )
```

```python
while rclpy.ok():
    executor_node.goal_received = False
    executor_node.node.get_logger().info("⏳ >>>>> Esperando nuevo objetivo en /goal ... >>>>>>")

    while not executor_node.goal_received:
        rclpy.spin_once(executor_node.node, timeout_sec=0.1)

    executor_node.node.get_logger().info(f"✅ !!! Objetivo recibido: ({executor_node.x_goal:.2f}, {executor_node.y_goal:.2f})")

    executor_node.laser_received = False
    executor_node.odom_received = False
    executor_node.future = Future()
    rclpy.spin_until_future_complete(executor_node.node, executor_node.future, timeout_sec=1.0)

    executor_node.node.get_logger().info(f">>>>> Estado inicial: pos=({executor_node.robot_x:.2f}, {executor_node.robot_y:.2f}), "
        f"distancia2={executor_node.distance2:.4f}")

    executor_node.update_state()
    ejecutar_episodio()
```