GRADO EN INGENIERÍA MATEMÁTICA E INTELIGENCIA ARTIFICIAL

TRABAJO FIN DE GRADO

Modelado de un entorno virtual para el entrenamiento de algoritmos de aprendizaje por refuerzo profundo con activos industriales

Autor: Jaime Paz Rodriguez

Director: Lucía Güitta López

Madrid

Agosto de 2025

Declaro, bajo mi responsabilidad, que el Proyecto presentado con el título

Modelado de un entorno virtual para el entrenamiento de algoritmos de aprendizaje por refuerzo profundo con activos industriales en la ETS de Ingeniería - ICAI de la Universidad Pontificia Comillas en el

curso académico 2024/25 es de mi autoría, original e inédito y

no ha sido presentado con anterioridad a otros efectos.

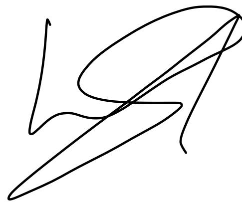
El Proyecto no es plagio de otro, ni total ni parcialmente y la información que ha sido

tomada de otros documentos está debidamente referenciada.

Fdo.: Jaime Paz Rodriguez Fecha: 21 / 08 / 2025

Autorizada la entrega del proyecto

EL DIRECTOR DEL PROYECTO



Fdo.: Lucía Güitta López Fecha: 27 / 08 / 2025

COMILLAS

UNIVERSIDAD PONTIFICIA

ICAI

GRADO EN INGENIERÍA MATEMÁTICA E INTELIGENCIA ARTIFICIAL

TRABAJO FIN DE GRADO

Modelado de un entorno virtual para el entrenamiento de algoritmos de aprendizaje por refuerzo profundo con activos industriales

Autor: Jaime Paz Rodriguez

Director: Lucía Güitta López

Madrid

Modelado de un entorno virtual para el entrenamiento de algoritmos de aprendizaje por refuerzo profundo con activos industriales

Autor: Paz Rodriguez, Jaime.
Director: Güitta López, Lucía.
Entidad Colaboradora: ICAI – Universidad Pontificia Comillas

ABSTRACT

Este trabajo presenta la migración del brazo robótico industrial ABB IRB120 desde MuJoCo a Unity, con el fin de explorar nuevas alternativas en el entrenamiento de agentes de Aprendizaje por Refuerzo Profundo. Se aborda la conversión geométrica y dinámica del modelo, la integración con Python y la definición de un entorno de entrenamiento multimodal. Los resultados muestran que el agente alcanza un 78 % de éxito y reduce la distancia media al objetivo a 0.049 m, validando la viabilidad de Unity como plataforma para el entrenamiento de dicho robot.

Palabras clave: Aprendizaje por refuerzo profundo, Simulación robótica, Proximal Policy Optimization, Motores de simulación, Transferencia sim-to-real, Entrenamiento multimodal.

1. Introducción

El Aprendizaje por Refuerzo Profundo (Deep Reinforcement Learning, DRL) se ha consolidado como una de las metodologías más prometedoras para la robótica, al permitir que los agentes aprendan políticas de control mediante interacción directa con un entorno simulado. Sin embargo, la elección del motor de simulación resulta crítica: debe equilibrar precisión física, eficiencia computacional y flexibilidad en la generación de escenarios.

MuJoCo ha sido tradicionalmente una referencia académica, pero presenta limitaciones en visualización y accesibilidad [1]. Frente a ello, Unity, junto con el paquete ML-Agents, ofrece un motor gráfico avanzado y una integración nativa con Python, lo que lo convierte en una alternativa atractiva para entrenamientos multimodales y escenarios cercanos al mundo real.[2] Este trabajo explora la migración del robot ABB IRB120 desde MuJoCo a Unity, reconstruyendo su geometría, cinemática y dinámica para integrarlo en un escenario más versátil. Se evalúa junto a ello el impacto de esta transición en el proceso de entrenamiento de agentes de DRL.

2. Metodología

El proyecto se desarrolla en dos fases principales. En primer lugar, se realiza la adaptación del robot ABB IRB120 desde el motor de simulación MuJoCo al entorno Unity. Para ello se importan y ajustan las mallas 3D, se reconstruye la jerarquía cinemática del manipulador y se parametrizan sus articulaciones y propiedades dinámicas mediante componentes propios del motor de simulación. Además, se define un escenario mínimo reproducible que incluye plano de trabajo, objetivo y sensores

virtuales.

En la segunda fase se integra el modelo con ML-Agents, estableciendo los espacios de observación y acción, la función de recompensa y las condiciones de finalización de episodios. El entrenamiento se lleva a cabo con el algoritmo Proximal Policy Optimization (PPO) durante $2 \cdot 10^6$ pasos de simulación, monitorizando métricas clave como la recompensa media, la tasa de éxito y la distancia al objetivo.

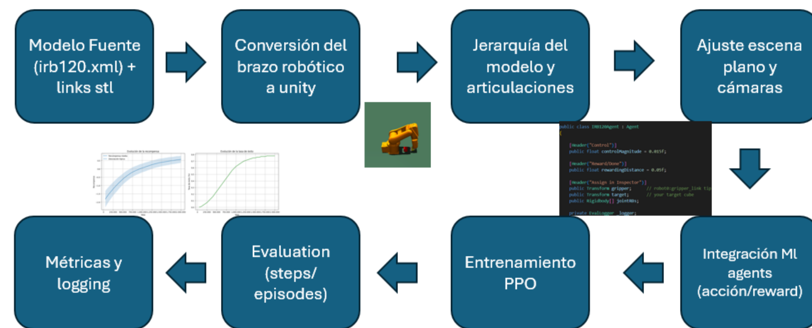


Ilustración 1 – Resumen del flujo experimental del trabajo

3. Descripción del modelo

La adaptación del robot ABB IRB120 desde su definición original en MuJoCo al entorno Unity es un proceso que requiere el uso del *MuJoCo–Unity plug-in* como apoyo inicial. A partir de las mallas y la descripción cinemática del robot, se reconstruye en Unity la jerarquía de eslabones y articulaciones utilizando componentes *ArticulationBody*, ajustando escalas, límites, masas e inercias para garantizar la coherencia dinámica con el modelo de referencia. Este paso permite disponer de una réplica funcional del robot en un entorno más flexible y visualmente avanzado.

Sobre esta base se diseña el modelo de entrenamiento en Unity ML-Agents, que combina observaciones vectoriales (posiciones y velocidades articulares, relación efectora–objetivo) con entradas visuales generadas por cámaras virtuales. El espacio de acción es continuo y de dimensión seis, correspondiente a los grados de libertad del manipulador. La función de recompensa se define como densa, penalizando la distancia al objetivo y las colisiones, e incorporando bonificaciones al éxito. El proceso de entrenamiento se lleva a cabo con el algoritmo Proximal Policy Optimization (PPO), configurado con *batch size* de 1024, *learning rate* de $3 \cdot 10^{-4}$ y un horizonte de $2 \cdot 10^6$ pasos de simulación. ML-Agents proporciona una comunicación directa entre Unity y Python mediante gRPC, además de herramientas de paralelización y monitorización de métricas, lo que permite seguir la evolución de la política en términos de recompensa media, tasa de éxito y distancia media al objetivo.[3]

4. Resultados

Los experimentos de entrenamiento en Unity ML-Agents muestran una evolución consistente en los indicadores clave. Tal como se observa en la Ilustración 3, la recompensa media evoluciona desde valores negativos en las fases iniciales (-1.82 en 50.000 pasos) a estabilizarse en torno a 0.57 tras 2 millones de pasos. Paralelamente, la tasa de éxito incrementa progresivamente hasta alcanzar un **78 %**, mientras que la distancia media entre el efector y el objetivo se reduce desde 0.352 m a menos de 5 cm.

Asimismo, la Ilustración 2 recoge los resultados finales del entrenamiento, que confirman la convergencia del agente hacia políticas estables y eficientes: episodios más cortos (686 pasos de media) y una disminución notable en las colisiones (de 27 a 5 por episodio). Estos resultados evidencian que, pese a la menor precisión física de Unity frente a MuJoCo, la migración del IRB120 y su entrenamiento con PPO permiten alcanzar un desempeño robusto y generalizable en la tarea de aproximación al objetivo.

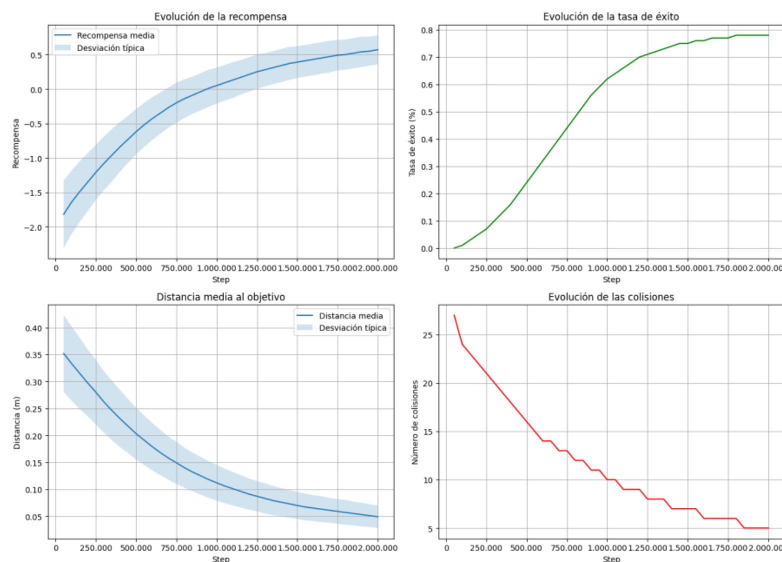


Ilustración 2 – Gráficas de métricas del entrenamiento

Métrica	**50.000 steps**	**1.000.000 steps**	**2.000.000 steps**
Recompensa media	-1.82	0.05	0.57
Tasa de éxito (%)	0.0	62 %	78 %

Ilustración 3– Resultados de entrenamiento en Unity

5. Conclusiones

El proyecto confirma la viabilidad de migrar el robot ABB IRB120 desde MuJoCo a Unity y de entrenarlo con éxito mediante ML-Agents. La adaptación del modelo ha permitido conservar la coherencia geométrica y dinámica del manipulador, mientras que el entrenamiento con PPO demuestra la capacidad del agente para aprender políticas estables y eficientes, alcanzando tasas de éxito del 78 % y reduciendo la distancia media al objetivo a apenas unos centímetros.

Estos resultados validan a Unity como una alternativa sólida para experimentos de Aprendizaje por Refuerzo Profundo en robótica, especialmente cuando se priorizan la flexibilidad del entorno, el realismo visual y la integración directa con librerías de aprendizaje automático.

6. Referencias

[1] **MuJoCo Documentation**. *MuJoCo: Programming Interface*. DeepMind, 2022. Disponible en: https://mujoco.readthedocs.io/en/stable/programming/index.html

[2] **Unity Technologies**. *Unity ML-Agents Toolkit (GitHub repository)*. Unity Technologies, 2023. Disponible en: https://github.com/Unity-Technologies/ml-agents

[3] OpenAI. *Proximal Policy Optimization (PPO)*. OpenAI Spinning Up in Deep RL, 2018. Disponible en: https://spinningup.openai.com/en/latest/algorithms/ppo.html

MODELING OF A VIRTUAL ENVIRONMENT FOR TRAINING DEEP REINFORCEMENT LEARNING ALGORITHMS WITH INDUSTRIAL ASSETS

Author: Paz Rodriguez, Jaime.
Director: Güitta López, Lucía.
Collaborating entity: ICAI – Universidad Pontificia Comillas

ABSTRACT

This work presents the migration of the industrial robotic arm ABB IRB120 from MuJoCo to Unity, with the aim of exploring new alternatives for training Deep Reinforcement Learning agents. The study addresses the geometric and dynamic conversion of the model, its integration with Python, and the definition of a multimodal training environment. The results show that the agent achieves a 78% success rate and reduces the average distance to the target to 0.049 m, validating the feasibility of Unity as a platform for the training of this robot.

Key words: Deep Reinforcement Learning, Robotic simulation, Proximal Policy Optimization, Simulation engines, Tranfer sim-to-real, Multimodal training.

1. Introduction

Deep Reinforcement Learning (DRL) has emerged as one of the most promising methodologies in robotics, as it enables agents to learn control policies through direct interaction with a simulated environment. However, the choice of simulation engine is critical: it must balance physical accuracy, computational efficiency, and flexibility in scenario generation.

MuJoCo has traditionally been an academic reference, but it presents limitations in visualization and accessibility [1]. In contrast, Unity, together with the ML-Agents package, provides an advanced graphics engine and native integration with Python, making it an attractive alternative for multimodal training and scenarios closer to the real world [2]. This work explores the migration of the ABB IRB120 robot from MuJoCo to Unity, reconstructing its geometry, kinematics, and dynamics to integrate it into a more versatile environment. The study also evaluates the impact of this transition on the training process of DRL agents.

2. Methodology

The project is developed in two main phases. First, the ABB IRB120 robot is adapted from the MuJoCo simulation engine to the Unity environment. To achieve this, the 3D meshes are imported and adjusted, the kinematic hierarchy of the manipulator is reconstructed, and its joints and dynamic properties are parameterized using the simulation engine's native components. In addition, a minimal reproducible scenario is defined, which includes a workspace, a target, and virtual sensors.

In the second phase, the model is integrated with ML-Agents, defining the observation and action spaces, the reward function, and the episode termination conditions. Training is carried out using the Proximal Policy Optimization (PPO) algorithm over $2 \cdot 10^6$ simulation steps, monitoring key metrics such as average reward, success rate, and distance to the target.

Source model (irb120.xml) + links stl

Conversion of the robot into unity

Model structure and articulations

Scene adjustments plane and cameras

Integration Ml agents (action/reward)

PPO Training

Evaluation (steps/ episodes)

Performance & logging

Ilustration 1 – Workflow summary

3. **Model configuration**

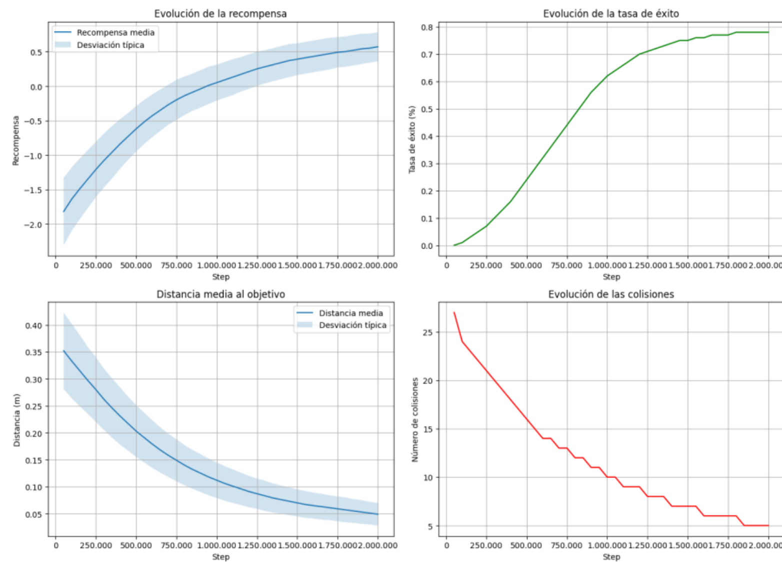
The adaptation of the ABB IRB120 robot from its original definition in MuJoCo to the Unity environment requires the use of the MuJoCo–Unity plug-in as an initial support. Based on the meshes and the robot's kinematic description, the hierarchy of links and joints is reconstructed in Unity using ArticulationBody components, adjusting scales, limits, masses, and inertias to ensure dynamic consistency with the reference model. This step makes it possible to obtain a functional replica of the robot in a more flexible and visually advanced environment.

On this basis, the training model is designed in Unity ML-Agents, combining vector observations (joint positions and velocities, end-effector–target relation) with visual inputs generated by virtual cameras. The action space is continuous and six-dimensional, corresponding to the manipulator's degrees of freedom. The reward function is defined as dense, penalizing distance to the target and collisions, while incorporating success bonuses. The training process is carried out with the Proximal Policy Optimization (PPO) algorithm, configured with a batch size of 1024, a learning rate of $3 \cdot 10^{-4}$, and a horizon of $2 \cdot 10^6$ simulation steps. ML-Agents provides direct communication between Unity and Python via gRPC, as well as tools for parallelization and metric monitoring, which allow tracking the evolution of the policy in terms of average reward, success rate, and average distance to the target.[3]

4. **Results**

The training experiments in Unity ML-Agents show a consistent evolution in the key indicators. As illustrated in Figure 3, the average reward increases from negative values in the initial phases (-1.82 at 50,000 steps) to stabilizing around 0.57 after 2 million steps. In parallel, the success rate progressively increased to 78%, while the average distance between the end-effector and the target decreased from 0.352 m to less than 5 cm.

Similarly, Figure 2 presents the final training results, which confirm the convergence of the agent toward stable and efficient policies: shorter episodes (686 steps on average) and a notable reduction in collisions (from 27 to 5 per episode). These results demonstrate that, despite Unity's lower physical accuracy compared to MuJoCo, the migration of the IRB120 and its training with PPO enable the achievement of robust and generalizable performance in the target-approach task.



Ilustration 2 – Training metrics

Measure	50.000 steps	1.000.000 steps	2.000.000 steps
Mean reward	-1.82	0.05	0.57
Success rate (%)	0.0	62 %	78 %

Ilustration 3– Training results (Unity)

5. Conclusions

The project confirms the feasibility of migrating the ABB IRB120 robot from MuJoCo to Unity and successfully training it through ML-Agents. The model adaptation

preserved the manipulator's geometric and dynamic consistency, while training with PPO demonstrated the agent's ability to learn stable and efficient policies, achieving success rates of 78% and reducing the average distance to the target to just a few centimeters.

These results validate Unity as a solid alternative for Deep Reinforcement Learning experiments in robotics, particularly when prioritizing environment flexibility, visual realism, and direct integration with machine learning libraries.

6. References

[1] **MuJoCo Documentation**. *MuJoCo: Programming Interface*. DeepMind, 2022. Disponible en: https://mujoco.readthedocs.io/en/stable/programming/index.html

[2] **Unity Technologies**. *Unity ML-Agents Toolkit (GitHub repository)*. Unity Technologies, 2023. Disponible en: https://github.com/Unity-Technologies/ml-agents

[3] OpenAI. *Proximal Policy Optimization (PPO)*. OpenAI Spinning Up in Deep RL, 2018. Disponible en: https://spinningup.openai.com/en/latest/algorithms/ppo.html

Índice de la memoria

Capítulo 1. Introducción ... 5

1.1 Contexto ... 6

1.2 Motivación ... 7

1.3 Objetivos ... 7

1.4 Estructura ... 8

Capítulo 2. Descripción de los Motores de Simulación ... 10

2.1 Mujoco ... 11

2.2 Unity ... 13

2.2.1 Ml-Agents ... 15

2.3 Gazebo ... 16

2.4 Comparativa ... 17

Capítulo 3. Adaptación del modelo IRB120 a Unity ... 20

3.1 Preparación del entorno Unity ... 20

3.2 Conversión de activos y geometría ... 21

3.3 Articulaciones y dinámica ... 22

3.4 Escena, colisiones y entorno ... 23

3.5 Integración con Ml-Agents ... 24

Capítulo 4. Entrenamiento y validación del agente ... 26

4.1 Algoritmo de aprendizaje por refuerzo ... 26

4.1.1 Diseño del Agente ... 27

4.1.2 Proceso de entrenamiento ... 28

4.2 Definición del entorno de entrenamiento ... 29

4.2.1 Espacio de observación ... 30

4.2.2 Espacio de acción ... 30

4.2.3 Función de recompensa ... 31

4.2.4 Condiciones de terminación ... 31

Capítulo 5. Análisis de Resultados ... 33

5.1 Métricas Importantes ... 34

Capítulo 6. Conclusiones y trabajos futuros.. 36

Capítulo 7. Bibliografía.. 38

ANEXO A 41

Índice de figuras

Figura 1. Diagrama del flujo experimental del trabajo 9
Figura 2. Representación del robot ABB IRB120 en el entorno MuJoCo. 13
Figura 3. Arquitectura general del entrenamiento de un agente 15
Figura 4. Representación del robot ABB IRB120 20
Figura 5. Panel Behavior Parameters del agente 24
Figura 6. Estructura del archivo de entrenamiento (YAML) 27
Figura 7. Evolución de las principales métricas de entrenamiento 34

Índice de tablas

Tabla 1. Tabla de Comparativa de motores de simulación.. 19

Tabla 2. Tabla de Resultados de entrenamiento 33

Capítulo 1. INTRODUCCIÓN

El aprendizaje por refuerzo profundo (DRL) se emplea en el entrenamiento de agentes que deben resolver tareas complejas a partir de la interacción con un entorno simulado. Es especialmente útil en el ámbito de la robótica industrial ya que permite generar políticas de control sin la necesidad de programar cada movimiento de forma manual. Al mismo tiempo, reduce costes y elimina riesgos asociados al uso de robots físicos.

La elección del simulador resulta crítica. El entorno debe reproducir la dinámica del sistema con suficiente fidelidad, integrar de forma estable librerías de aprendizaje automático y ejecutar un gran número de interacciones en tiempos razonables. En este ámbito, MuJoCo es uno de los motores más utilizados por su precisión en dinámicas de contacto y por la facilidad que ofrece para modelar sistemas articulados. No obstante, sus limitaciones en accesibilidad, capacidades gráficas y escalabilidad hacen que no siempre sea el más adecuado en todos los contextos.

Por este motivo, el trabajo se centra en la migración de un modelo del robot ABB IRB120, desarrollado inicialmente en MuJoCo, al entorno Unity. Esta plataforma, junto con el paquete ML-Agents, incorpora un motor gráfico avanzado, flexibilidad en el diseño de escenarios y comunicación directa con Python. Todo ello la convierte en una alternativa interesante, de manera especial en este caso con aplicaciones industriales.

El objetivo principal es comprobar la viabilidad de la migración y analizar de qué manera influye Unity en el entrenamiento de un agente con DRL. Para lograrlo, se adapta el modelo IRB120 en formato XML a un formato compatible con Unity, estableciendo la comunicación con Python y realizando entrenamientos con distintos algoritmos de DRL.

El estudio aborda tanto la parte técnica de la conversión del robot como su integración con ML-Agents. Además, examina las implicaciones del cambio de motor en aspectos clave: la

estabilidad de las políticas aprendidas, la eficiencia del proceso de entrenamiento y la capacidad de generalización de los agentes.

CONTEXTO ACTUAL

Se dispone de un entorno de simulación completo del robot industrial ABB IRB120 desarrollado en MuJoCo e integrado en Python. Este entorno está concebido como una extensión de la librería OpenAI Gym, lo que permite definir un marco estandarizado de observaciones, acciones y recompensas, además de garantizar la compatibilidad con librerías de aprendizaje por refuerzo profundo como PyTorch.[1]

El modelo del robot se describe mediante un archivo que especifica la geometría de cada uno de sus enlaces, el gripper y los actuadores, junto con las restricciones de movimiento y los elementos adicionales del escenario, como el plano de suelo y el objetivo a alcanzar. A partir de esta descripción física, el módulo irb120.py implementa la clase encargada de gestionar la simulación, inicializar el motor de MuJoCo y proporcionar las funciones necesarias para el entrenamiento del agente.[2] Entre ellas se incluyen la generación de observaciones multimodales —que combinan imágenes RGB renderizadas desde la cámara virtual con los estados articulares del robot—, la aplicación de acciones discretizadas en las articulaciones y el cálculo de la recompensa asociada a la distancia al objetivo.

Sobre esta base se construye la infraestructura de entrenamiento definida, que permite ejecutar procesos de aprendizaje en paralelo mediante el algoritmo A3C (Asynchronous Advantage Actor-Critic). Se coordina la interacción entre el agente y el entorno, y se gestiona la red neuronal con arquitectura actor-crítico y memoria LSTM, estableiendo los parámetros globales de la sesión de entrenamiento, como la tasa de aprendizaje, la frecuencia de evaluación o la duración máxima de los episodios. El repositorio se completa con módulos auxiliares dedicados a la optimización, la definición de arquitecturas y las funciones de soporte, lo que dota al proyecto de una estructura modular y reproducible.

Gracias a esta configuración, el sistema ya permite entrenar políticas de control capaces de guiar el efector final del robot hacia un objetivo tridimensional, evaluando tanto la precisión del movimiento como la estabilidad del aprendizaje. Este marco constituye el contexto sobre el que se desarrolla el presente trabajo, centrado en la migración del modelo del IRB120 al entorno Unity ML-Agents con el fin de analizar las diferencias entre motores de simulación y su impacto en la eficacia del entrenamiento.

1.1 MOTIVACIÓN

La investigación en aprendizaje por refuerzo profundo aplicado a la robótica requiere entornos de simulación que combinen realismo físico, eficiencia computacional y flexibilidad en el diseño de escenarios. Aunque MuJoCo ha sido una herramienta de referencia por su precisión en la modelización de sistemas dinámicos, presenta limitaciones en visualización y accesibilidad que reducen su aplicabilidad en contextos industriales y de validación visual [3].

Unity, mediante el paquete ML-Agents, ofrece una alternativa atractiva al incorporar un motor gráfico avanzado y un marco de comunicación fluido con Python. La migración del entorno del IRB120 a esta plataforma responde al interés por aprovechar estas ventajas y estudiar cómo afectan al rendimiento del entrenamiento, a la estabilidad de las políticas aprendidas y a la transferencia potencial hacia entornos más realistas.

.

1.2 OBJETIVOS

El objetivo principal de este trabajo es migrar el modelo del robot ABB IRB120, previamente desarrollado en MuJoCo, al entorno Unity ML-Agents. Esto no se limita únicamente a la conversión del modelo, sino que busca asegurar que la nueva simulación preserve las

propiedades dinámicas y cinemáticas del robot, para poder utilizarse como base para el entrenamiento de algoritmos de aprendizaje por refuerzo profundo.

En este contexto, se plantean como metas complementarias la adaptación del modelo virtual al formato de Unity, la integración de un flujo de comunicación fluido con Python a través de ML-Agents y la validación del sistema mediante sesiones de entrenamiento de agentes. El análisis del rendimiento obtenido en Unity permite, además, establecer una comparación con la versión original en MuJoCo.

1.3 ESTRUCTURA DEL TRABAJO

La memoria se divide en seis capítulos estructurados de la siguiente manera:

El capítulo 2 presenta un análisis de los principales motores de simulación usados en la investigación con aprendizaje por refuerzo. Se revisan MuJoCo, Unity ML-Agents y, de forma más breve, Gazebo. Se señalan sus ventajas y limitaciones en realismo físico, flexibilidad, accesibilidad e integración con librerías de aprendizaje automático.

El capítulo 3 explica la adaptación del modelo del robot ABB IRB120 de MuJoCo a Unity. Se detallan los pasos para trasladar la descripción física y geométrica, configurar articulaciones y actuadores, y definir observaciones y recompensas. También se comentan los problemas técnicos encontrados y las soluciones aplicadas.

El capítulo 4 describe el entrenamiento de agentes en Unity con ML-Agents. Se explica la integración con Python, los algoritmos usados y las configuraciones de entrenamiento. Se presta especial atención a la definición de episodios, recompensas y métricas de evaluación.

El capítulo 5 muestra los resultados obtenidos tras entrenar los agentes en Unity y los compara con los generados previamente en MuJoCo. Se revisan métricas como la estabilidad de las sesiones de entrenamiento, el tiempo necesario para alcanzar un buen rendimiento y la capacidad de los modelos para adaptarse a situaciones distintas a las vistas en el

entrenamiento. Esto permite observar de manera clara qué efectos tiene cambiar de motor de simulación.

El capítulo 6 recoge las conclusiones generales del proyecto y plantea posibles mejoras a futuro. Entre ellas se considera probar escenarios más exigentes, evaluar distintos algoritmos de forma comparativa y llevar las políticas entrenadas a robots físicos para comprobar su desempeño más allá del entorno de simulación.

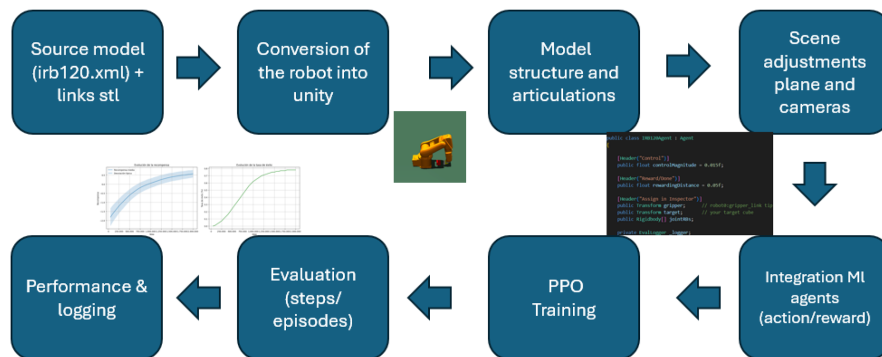


Figura 1. Diagrama del flujo experimental del trabajo

Capítulo 2. DESCRIPCIÓN DE LOS MOTORES DE SIMULACIÓN

Entrenar directamente sobre un robot real no resulta viable en la gran mayoría de casos por el coste de tiempo y recursos, el desgaste de los componentes y los riesgos que aparecen al ejecutar interacciones no controladas.

Por esta razón, el uso de un motor de simulación es fundamental. Para que sea útil y eficiente debe cumplir varios requisitos. El primero es la fidelidad física, es decir, que pueda reproducir de forma precisa la dinámica del sistema, incluyendo colisiones, restricciones, fricción y gravedad. El segundo es la eficiencia computacional, ya que la velocidad con la que el simulador genera transiciones estado–acción–recompensa influye directamente en el tiempo de entrenamiento. En tercer lugar, está la integración con frameworks de aprendizaje automático como PyTorch o TensorFlow, lo que exige interfaces estables y bien documentadas. Además, aspectos como la calidad visual, la facilidad para diseñar escenarios y el apoyo de comunidades activas también marcan la diferencia en la práctica y ayudan al desarrollo del agente.

En investigación con DRL destacan tres motores. MuJoCo (Multi-Joint dynamics with Contact), el cual se ha convertido en un estándar en el ámbito académico por su solver físico optimizado y su integración con librerías como OpenAI Gym, lo que lo ha hecho base de muchos benchmarks en control continuo [3]. Unity ML-Agents ofrece un motor gráfico avanzado y herramientas específicas para entrenar agentes, con una API en Python que permite implementar algoritmos de DRL de forma directa, además de funciones como domain randomization y entrenamiento multiagente.[4] Gazebo, en cambio, resalta por su integración con ROS, lo que lo hace especialmente adecuado para llevar políticas entrenadas a robots reales (sim2real), espacialmente en tareas de robótica móvil y manipulación [5],[6].

Este capítulo revisa las características de cada motor y analiza sus ventajas y limitaciones en el entrenamiento de agentes de DRL. Esta comparación permite situar la migración del modelo ABB IRB120 a Unity dentro de un contexto más amplio y entender cómo la elección del simulador afecta a la calidad, eficiencia y transferibilidad de los resultados de entrenamiento.

2.1 MUJOCO

MuJoCo (Multi-Joint dynamics with Contact) es un motor de simulación física de propósito general, diseñado para modelar estructuras articuladas que interactúan con su entorno. Fue creado por Roboti LLC y en 2021 pasó a ser propiedad de DeepMind. Desde 2022 está disponible como software de código abierto y se ha consolidado como una de las principales herramientas en investigación en robótica, biomecánica y aprendizaje por refuerzo profundo.[3]

A diferencia de otros motores, MuJoCo combina dos enfoques que suelen estar separados. Por un lado, utiliza coordenadas generalizadas para describir de manera eficiente sistemas con múltiples grados de libertad. Por otro, incorpora un solver de contactos basado en optimización convexa. Esta última característica es una de sus principales innovaciones, ya que sustituye los modelos de contacto rígido basados en problemas de complementariedad lineal (LCP), que son más costosos e inestables, por un modelo de contactos blandos (*soft contacts*). Este enfoque proporciona una formulación convexa, analíticamente invertible y numéricamente más eficiente. Gracias a ello, el motor puede manejar de forma unificada distintos tipos de fricción (estática, dinámica, torsional y de rodadura), además de límites articulares y restricciones de tendones.

En términos de **pipeline computacional**, MuJoCo separa la descripción del modelo (mjModel) de las variables dinámicas (mjData). El primero contiene información estructural invariante (geometrías, articulaciones, actuadores, parámetros físicos), mientras que el segundo gestiona los estados dinámicos que cambian en cada paso de simulación (posiciones, velocidades, fuerzas, contactos, etc.). Esta separación facilita la paralelización,

el muestreo masivo para DRL y la reutilización de modelos en diferentes condiciones iniciales. El avance de la simulación se realiza mediante la función principal mj_step(m, d), que integra las ecuaciones de movimiento en tiempo continuo con distintos esquemas numéricos. MuJoCo ofrece integradores de tipo **semiimplícito de Euler**, **implícito en velocidad** e incluso **Runge-Kutta de cuarto orden**, lo que permite adaptar el balance entre precisión y coste computacional a las necesidades de cada aplicación.

Otro aspecto relevante es la flexibilidad en la definición de actuadores. El modelo abstracto de actuación permite representar motores eléctricos, sistemas neumáticos e hidráulicos, músculos biológicos o controladores PD, con la posibilidad de introducir dinámicas internas de activación (tercer orden). Esto convierte a MuJoCo en una herramienta muy adecuada para experimentos de control avanzado e identificación de parámetros.

En cuanto a la modelización, el simulador emplea el lenguaje MJCF (MuJoCo XML format) como formato nativo, aunque también admite la importación de ficheros URDF. El compilador traduce estas descripciones a estructuras internas optimizadas para la simulación, pudiendo exportarlas a un formato binario (.mjb) para acelerar la carga [8] . La visualización se gestiona mediante un renderizador OpenGL nativo, que permite depurar y analizar simulaciones mostrando fuerzas de contacto, marcos de referencia o ejes de actuadores en tiempo real.

La combinación de precisión física y eficiencia numérica ha llevado a MuJoCo a convertirse en la base de muchos benchmarks estándar en DRL, como los entornos de **OpenAI Gym** (Hopper, Walker2D, HalfCheetah, Ant, Humanoid, entre otros), y ha sido utilizado ampliamente en investigación sobre sim-to-real, control de robots manipuladores y locomoción de sistemas complejos. Sin embargo, presenta limitaciones en la calidad gráfica, la accesibilidad para usuarios no especializados y la escalabilidad en escenarios multimodales, aspectos que motivan la exploración de alternativas como Unity.

Figura 2. Representación del robot ABB IRB120 en el entorno MuJoCo.

2.2 UNITY

Unity es un motor de desarrollo multiplataforma utilizado de forma generalizada en videojuegos, aplicaciones de simulación y entornos de realidad virtual. A diferencia de motores pensados únicamente para simulación física, Unity nació como una plataforma gráfica orientada al renderizado en tiempo real. Su arquitectura modular integra distintos componentes, como el motor de físicas, gráficos 3D, sonido, animaciones y la posibilidad de ampliar funcionalidades mediante *scripts* en C#.

En lo relativo a simulación física, Unity emplea el motor PhysX de NVIDIA. Este permite trabajar con cuerpos rígidos, colisiones, fricción, articulaciones y restricciones cinemáticas. Aunque su nivel de precisión no alcanza el de motores especializados como MuJoCo en la resolución de contactos complejos, ofrece un buen compromiso entre realismo y eficiencia en tiempo real. [4] Además, el sistema admite la conexión con librerías externas, lo que facilita extender sus capacidades cuando el proyecto lo requiere.

Uno de los elementos que diferencia a Unity es su *graphics pipeline*. El motor es compatible con HDRP (*High Definition Render Pipeline*) y URP (*Universal Render Pipeline*), lo que permite generar entornos con un alto grado de detalle visual. Este aspecto es especialmente útil en investigación en robótica, donde la calidad de las imágenes sintéticas es un factor clave para tareas de visión artificial y entrenamiento basado en datos visuales. Unity también proporciona herramientas para la creación de *datasets* sintéticos con anotaciones automáticas, incluyendo *bounding boxes*, máscaras de segmentación, mapas de profundidad y normales de superficie. Estas funciones lo convierten en una opción práctica para experimentos relacionados con percepción y aprendizaje automático.

La arquitectura de Unity se organiza en torno a un **sistema de entidades jerárquico** compuesto por *GameObjects* y *Components*. Cada objeto de la escena puede tener propiedades físicas (cuerpos rígidos, colisionadores), propiedades gráficas (mallas, materiales, texturas) y comportamientos definidos mediante scripts. Esta modularidad permite construir entornos altamente configurables, desde simples prototipos hasta escenarios industriales complejos [4],[5].

En el contexto de la simulación robótica, Unity ofrece varias ventajas:Escalabilidad para crear escenarios multimodales y realistas.Capacidad de **domain randomization** (variación de texturas, luces, posiciones, ruidos sensoriales) para mejorar la robustez de los agentes y facilitar la transferencia sim2real. Herramientas de depuración visual avanzadas que permiten inspeccionar la interacción entre el agente y su entorno en tiempo real.

Sin embargo, Unity carece por sí mismo de un marco nativo para aprendizaje automático. Para suplir esta limitación se desarrolló **Unity ML-Agents**, un paquete que extiende las capacidades del motor y lo conecta con librerías de aprendizaje profundo en Python.[4]

2.2.1 UNITY ML-AGENTS

Unity ML-Agents (MLA) es un paquete de código abierto creado por Unity Technologies que extiende las capacidades del motor para trabajar con algoritmos de aprendizaje automático. En la práctica, establece un enlace entre Unity y Python. Unity se utiliza como simulador del entorno y Python ejecuta el proceso de entrenamiento del agente..

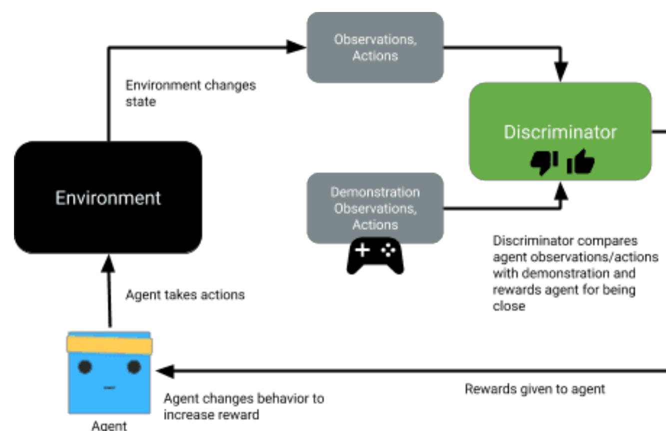


Figura 3 *Arquitectura general del entrenamiento de un agente en Unity Ml-Agents .[Fuente: Unity][11]*

Desde el punto de vista técnico, ML-Agents se compone de tres elementos principales:

1. Unity Environment (side C#): cada escena de Unity se configura como un entorno de entrenamiento, donde los agentes se definen como *GameObjects* con un componente Agent. Este componente controla cómo se recogen las observaciones (vectoriales, imágenes, raycasts, etc.), cómo se aplican las acciones y cómo se calculan las recompensas y condiciones de finalización de episodios [27].

2. Python API: mediante el paquete mlagents en Python, el entorno Unity se comunica a través de un protocolo gRPC. Esto permite lanzar entrenamientos con algoritmos de DRL (como PPO, SAC o DDPG), gestionar múltiples instancias del simulador en paralelo y registrar métricas de aprendizaje [9].

3. El Entrenador (Trainer), implementado en PyTorch, constituye el núcleo encargado de ajustar las políticas de los agentes. ML-Agents incluye implementaciones propias

de algoritmos como Proximal Policy Optimization (PPO), Soft Actor-Critic (SAC) y Behavioral Cloning (BC). [9] Además, permite conectar con otros frameworks externos cuando es necesario.

Una de las ventajas más destacadas es la posibilidad de entrenar en paralelo y de forma distribuida, lo que ayuda a reducir de manera considerable los tiempos de convergencia. También incorpora utilidades de domain randomization y curriculum learning, que permiten pasar de escenarios sencillos a entornos progresivamente más complejos.

En cuanto a las observaciones, ML-Agents admite distintos tipos de entradas: vectores numéricos como estados articulares o posiciones, observaciones visuales en forma de imágenes RGB o de profundidad, e incluso datos de sensores simulados como raycasts o cámaras múltiples [3], [9] . Esta capacidad de combinar señales heterogéneas lo hace especialmente útil en robótica, donde trabajar con información multimodal es un requisito fundamental.

2.3 GAZEBO

Gazebo es un simulador físico orientado a robótica que destaca por su integración nativa con **ROS (Robot Operating System)**, lo que lo convierte en una herramienta de referencia para la validación de algoritmos en sistemas reales. El motor físico principal es **ODE (Open Dynamics Engine)**, aunque también admite Bullet, DART o Simbody, lo que permite ajustar el balance entre precisión y rendimiento según la aplicación [12].

Los modelos de robot se definen en **URDF** o **SDF**, y pueden incorporar sensores simulados como cámaras RGB-D, LiDAR o IMUs. Estos sensores publican datos directamente en tópicos ROS, lo que facilita la reutilización de los mismos algoritmos tanto en simulación como en hardware real [12]. Este acoplamiento Gazebo–ROS constituye su principal fortaleza frente a otros motores.

Sin embargo, Gazebo presenta limitaciones relevantes para el aprendizaje por refuerzo profundo: el coste computacional de sus solvers de contacto lo hace menos eficiente que MuJoCo, y su motor gráfico es menos avanzado que Unity, lo que restringe la generación de datos visuales realistas. Por ello, aunque es excelente para **sim2real y pruebas de integración de sistemas**, no resulta el más adecuado para **entrenamientos masivos de agentes DRL**.

2.4 COMPARATIVA ENTRE MOTORES DE SIMULACIÓN

Gazebo ofrece una integración nativa con **ROS**, lo que lo convierte en una herramienta de referencia en la validación de sistemas robóticos reales. La posibilidad de simular sensores completos (cámaras, LiDAR, IMUs, GPS) y publicar su información en tópicos ROS permite trasladar fácilmente los algoritmos a robots físicos, lo que lo hace idóneo para proyectos orientados al sim2real. [7] Sin embargo, esta ventaja viene acompañada de ciertas limitaciones: la eficiencia computacional es reducida en comparación con MuJoCo y Unity, lo que penaliza entrenamientos que requieren millones de interacciones, y su motor gráfico resulta poco adecuado para generar entornos visualmente ricos o datasets sintéticos de visión artificial.

Unity, por el contrario, se apoya en un motor gráfico avanzado capaz de producir entornos altamente realistas y configurables, con soporte directo para **domain randomization** y generación de observaciones visuales (RGB, profundidad, segmentación). Estas características lo hacen más atractivo para experimentos de aprendizaje profundo en escenarios donde la percepción juega un papel clave. La integración con **ML-Agents** proporciona además una API directa en Python para el entrenamiento con algoritmos de DRL, con soporte para paralelización y aprendizaje curricular, simplificando el flujo de

trabajo respecto a Gazebo. Aunque su motor físico (PhysX) es menos preciso que los solvers especializados, ofrece un rendimiento suficiente para las tareas planteadas en este proyecto.

En consecuencia, **se ha optado por Unity como motor de simulación complementario a MuJoCo**, priorizando la flexibilidad para diseñar entornos, el realismo visual y la integración nativa con herramientas de aprendizaje profundo. Esta elección permite no solo comparar los resultados obtenidos con MuJoCo, sino también abrir la puerta a aplicaciones futuras que involucren visión por computador y escenarios multimodales.

La siguiente tabla resume las características más relevantes de los tres motores de simulación analizados, en el contexto del aprendizaje por refuerzo profundo en robots:

Característica	***MuJoCo***	***Unity***	***Gazebo***
Precisión física	Muy alta, solver en coordenadas generalizadas con contactos convexos optimizados.	Media (PhysX), suficiente para muchas tareas pero menos precisa en contactos complejos.	Media- baja, depende del motor físico usado (ODE, Bullet, DART).
Eficiencia computacional	Muy alta, optimizado para millones de pasos en DRL.	Alta, soporta instancias paralelas, aunque con sobrecarga gráfica.	Baja-media, penalizado en DRL por overhead de ROS y solvers menos eficientes.
Gráficos y visualización	Básicos (OpenGL nativo).	Muy avanzados (HDRP/URP, datasets sintéticos con anotaciones).	Limitados, gráficos funcionales pero sin realismo.
Lenguaje de modelado	MJCF (XML propio), compatible con URDF.	Escenas en GameObjects/Components (C#), extensible con ML-Agents.	URDF/SDF (XML), estándar en ROS.
Actuadores y control	Modelo abstracto muy flexible (motores, músculos, PD, hidráulicos).	Estándar PhysX + actuadores programables vía scripting.	Amplia variedad vía ROS plugins, orientado a robots reales.
Sensores virtuales	Limitados (sites, cámaras básicas).	Muy avanzados (RGB, depth, segmentación, raycasts, LiDAR simulado).	Muy completos (cámaras, LiDAR, GPS, IMU, etc.).
Integración con ML	Directa con OpenAI Gym, PyTorch, TensorFlow.	API nativa Python vía ML-Agents (gRPC).	ROS + frameworks externos, no optimizado para DRL.
Uso típico	Benchmarks de DRL, locomoción, manipulación.	Simulación multimodal, visión por computador, DRL industrial.	Validación sim2real, robótica móvil, integración con ROS.
Limitaciones principales	Gráficos básicos, curva de aprendizaje en MJCF.	Precisión física menor que MuJoCo, curva de aprendizaje en diseño de entornos.	Baja eficiencia para DRL, render limitado.

Tabla 1: Comparativa Motores de Simulación

Capítulo 3. ADAPTACIÓN DEL IRB120 A UNITY

El robot ABB IRB120 se encuentra definido en **MuJoCo** mediante un fichero en formato MJCF (irb120.xml) que describe la geometría, los grados de libertad y las propiedades dinámicas del sistema, acompañado de los ficheros de malla en formato STL de cada eslabón. En Python se implementa la lógica del entorno, con los espacios de observación y acción, la evolución temporal del sistema y la función de recompensa correspondiente.

El objetivo es trasladar dicho modelo al motor **Unity**, con el fin de reproducir fielmente su cinemática y dinámica al tiempo que se aprovechan las capacidades gráficas avanzadas y la integración directa con **ML-Agents**. La adaptación no se reduce a importar geometría, sino que requiere reconstruir la jerarquía del robot, configurar las articulaciones con sus limitaciones y parámetros físicos, definir un entorno de simulación adecuado y establecer la conexión con ML-Agents para el entrenamiento de agentes. El resultado es un modelo coherente con la versión de MuJoCo pero integrado en una plataforma más versátil y escalable para entrenamientos con algoritmos de aprendizaje por refuerzo profundo.

3.1 PREPARACIÓN DEL ENTORNO UNITY Y ML-AGENTS

La adaptación del modelo comienza con la configuración del entorno de trabajo en Unity, para lo cual se emplea la versión Unity 2022.3 LTS, recomendada por su estabilidad y plena compatibilidad con ML-Agents. El proyecto está configurado en modo 3D y se establece el sistema de unidades en metros para mantener la coherencia con el modelo original desarrollado en MuJoCo.

Desde el Package Manager, propio de Unity, se añaden los paquetes básicos para el trabajo. En primer lugar, ML-Agents 2.3.0, encargado de la comunicación entre la simulación y los algoritmos de aprendizaje por refuerzo. Después, Barracuda 2.1.0, que permite ejecutar en Unity modelos de redes neuronales exportados en formato ONNX [27] , [15]. Finalmente,

de manera opcional, se encuentra incorporado el Universal Render Pipeline (URP), con el objetivo de mejorar la eficiencia del renderizado en la generación de entornos [21].

A nivel de configuración física, se modifican los parámetros globales de la simulación en Unity [12]. El Fixed Timestep se establece en 0,02 s, equivalente a 50 Hz, lo que asegura una integración numérica suficientemente precisa sin penalizar en exceso el rendimiento. Se incrementa además el número de iteraciones del solver de contactos para mejorar la estabilidad de las articulaciones durante el entrenamiento.

En paralelo, se prepara el entorno de ejecución en Python 3.10 mediante un entorno virtual dedicado. En este se instalan las dependencias oficiales de ML-Agents (mlagents==0.28.0 y mlagents-envs==0.28.0), junto con PyTorch 2.0 como backend de aprendizaje y TensorBoard para la monitorización de métricas de entrenamiento. La comunicación entre Unity y Python se realiza a través del protocolo gRPC, lo que permite mantener procesos de simulación y entrenamiento desacoplados pero sincronizados en tiempo real.

3.2 CONVERSIÓN DE ACTIVOS Y GEOMETRÍA

La migración del modelo comienza con la conversión de los activos geométricos definidos originalmente en MuJoCo. El fichero irb120.xml describe la estructura jerárquica del robot ABB IRB120 y enlaza las mallas en formato **STL** correspondientes a cada uno de sus eslabones. Estas mallas se extraen y se convierten en **activos de Unity** importándolas al proyecto mediante el *Asset Importer*. Durante este proceso se configuran las propiedades de importación para garantizar consistencia entre simuladores: la escala se ajusta a **0,001**, dado que los ficheros STL suelen estar definidos en milímetros y Unity utiliza metros como unidad base; las normales se recalculan para asegurar una correcta iluminación y las colisiones de malla se sustituyen, siempre que es posible, por **colliders primitivos** (cajas, cápsulas o cilindros) con el fin de mejorar el rendimiento y la estabilidad numérica [14].

Una vez importadas las mallas, se construye en el panel **Hierarchy** de Unity la jerarquía padre–hijo que reproduce la cinemática serial del IRB120 [8]. La base del robot se

establece como objeto raíz y, de forma anidada, se añaden sucesivamente los seis eslabones, cada uno alineado con el eje de rotación correspondiente. Para ello se utilizan referencias de los *anchors* definidos en el XML original, que se verifican manualmente comparando posiciones y orientaciones de cada STL. Esta organización jerárquica es crítica para garantizar que las transformaciones locales de cada eslabón se transmitan correctamente a los subsiguientes, respetando la cadena cinemática.

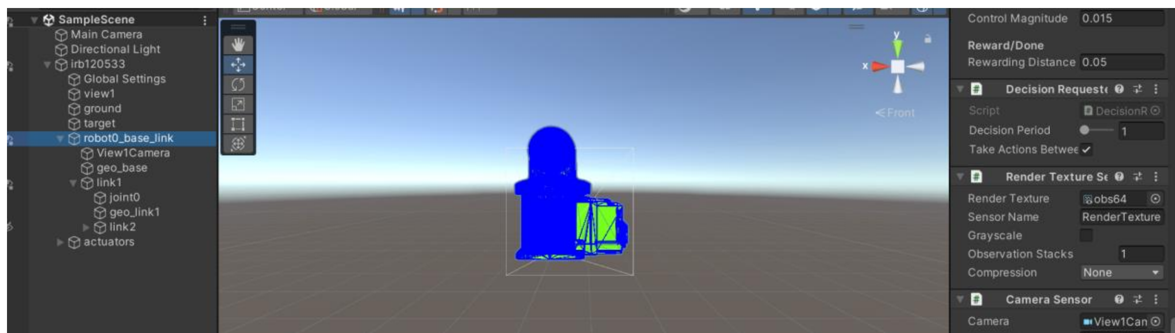


Figura 4. Representación del robot ABB IRB120 con la jerarquía propia de Unity

En esta fase también se depura la geometría, eliminando vértices redundantes y optimizando las mallas para tiempo real, reduciendo la complejidad de renderizado sin comprometer la precisión de colisiones. El resultado, que se puede observar en la figura superior es un conjunto de objetos 3D listos para asociarse a las articulaciones y parámetros físicos en Unity, manteniendo la coherencia geométrica y de escala con respecto al modelo original de MuJoCo.

3.3 ARTICULACIONES Y DINÁMICA

La reconstrucción cinemática del IRB120 en Unity se realiza con **ArticulationBody**, de modo que cada eslabón queda representado por un cuerpo articulado con un único grado de libertad rotacional, heredando transformaciones locales de su antecesor. El eje de giro se alinea con el definido en el MJCF original y el punto de anclaje se establece en el pivote geométrico del STL, garantizando que el par cinemático reproduzca el movimiento prescrito. La conversión de límites articulares de MuJoCo (en radianes) a Unity (en grados) mantiene

la asimetría cuando existe, y se complementa con la parametrización del **ArticulationDrive** en modo de posición o velocidad según el tipo de acción que más conviene al entrenamiento posterior [21] . Para estabilizar la dinámica durante simulación intensiva, se ajustan *stiffness*, *damping*, *forceLimit* y el número de iteraciones del solver de contactos, priorizando la no penetración y el seguimiento suave del *setpoint* antes que el tiempo de asentamiento mínimo [3].

Las **masas** y **tensores de inercia** se definen por eslabón. Cuando no se dispone de inercia fiable en el MJCF o el STL carece de densidad realista, se estima a partir de volumen y material y se valida empíricamente comprobando la respuesta transitoria ante impulsos pequeños y la ausencia de oscilaciones numéricas. El rozamiento articular y la amortiguación viscosa se ajustan para aproximar el *bias* dinámico del modelo original sin introducir sobreamortiguamiento que degrade la capacidad de seguimiento. El resultado es una cadena seriada con seis articulaciones, coherente en rangos, pares admisibles y rigidez efectiva, apta para control por consignas articulares o para control cinemático del efector con lazos externos. Este esquema es consistente con el flujo de conversión modular planteado en la memoria técnica del proyecto (conversión de XML, ajuste de parámetros físicos y restricciones), donde se prioriza reproducibilidad y separación clara entre geometría, cinemática y parámetros de simulación.

3.4 ESCENA, COLISIONES Y ENTORNO

La escena de Unity incorpora un plano de trabajo con material físico definido (fricción estática/dinámica y *bounciness* controlados) y un **objeto objetivo** que actúa como referencia de tarea. Las **colisiones** del robot se modelan, siempre que es posible, con *colliders* primitivos estrechamente ajustados a la malla visible, reservando *MeshCollider* para geometrías inevitables y marcándolo en modo *convex* para preservar estabilidad con PhysX. El filtrado de contacto mediante **Layers** evita autocolisiones no deseadas y reduce el coste del *broadphase* [20] ,[21]. La **cámara** principal se configura con FOV y *clipping planes*

adecuados a la escala del robot; si la tarea requiere observación visual, se habilitan cámaras adicionales o *RenderTextures* dedicados, manteniendo resoluciones moderadas para no saturar el *frame budget*. Cuando procede, se introduce **domain randomization** controlada (iluminación, materiales del plano, posición inicial del objetivo) para robustecer la política sin alterar la especificación cinemática. Este ensamblaje reproduce el "entorno mínimo reproducible" descrito en la planificación metodológica del proyecto, separando activos, física y mecanismos de observación para facilitar pruebas de renderizado y evaluación.

3.5 INTEGRACIÓN CON ML-AGENTS

La integración con **ML-Agents** se articula mediante un componente Agent en C#. El agente implementa la inicialización del episodio, la recolección de **observaciones** y la aplicación de **acciones** sobre los *ArticulationBody*. Las observaciones abarcan el estado articular (ángulos y velocidades normalizados), la pose relativa del objetivo y, si aplica, vectores de contacto o *raycasts*; en tareas con percepción, se añaden *stacks* de imágenes procedentes de cámaras dedicadas [22] , [27]. Las acciones se codifican como incrementos de consigna por articulación o referencias directas de posición/velocidad, garantizando acotación y *rate limiting* para evitar choques numéricos con el *drive*.

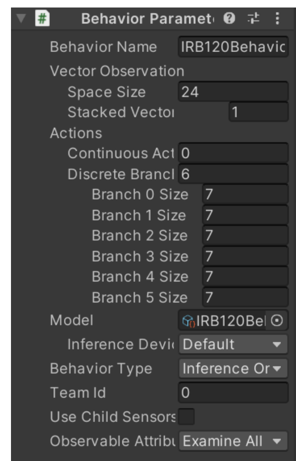


Figura 5. Panel Behavior Parameters del agente con definición de observaciones y acciones

La **recompensa** densa combina reducción de distancia euclídea del efector al objetivo, penalización por consumo de par o exceso de velocidad y bonificaciones por éxito; los criterios de terminación incluyen alcanzar tolerancia espacial, agotar el horizonte o incurrir en colisión crítica. La configuración del *Behavior Parameters,* visible en la figura 5, especifica tamaños de espacios, periodo de decisión y dispositivo de inferencia, asegurando compatibilidad con el *trainer* en Python y manteniendo la trazabilidad de experimentos. Este acoplamiento Unity↔Python responde a la arquitectura modular definida para conversión y entrenamiento, donde la escena proporciona señales bien definidas al lazo de optimización externo.

Capítulo 4. ENTRENAMIENTO Y VALIDACIÓN DEL AGENTE

4.1 *ALGORITMO DE APRENDIZAJE POR REFUERZO*

El entrenamiento del manipulador ABB IRB120 se realiza mediante **aprendizaje por refuerzo profundo (DRL)** con el algoritmo **Proximal Policy Optimization (PPO)**. Este algoritmo forma parte del toolkit ML-Agents de Unity y se utiliza de manera habitual en tareas de control continuo.

El algoritmo **PPO** pertenece a la familia de los métodos *actor-crítico*. El actor se encarga de generar la política estocástica $\pi(a|s)$, esto quiere decir que asigna probabilidades a las acciones continuas en función del estado observado. Por su parte, el crítico estima el valor esperado del estado $V(s)$, lo que permite evaluar la bondad de las acciones propuestas y ajustar la política en consecuencia [16], [18], [23]. Esta interacción entre actor y crítico garantiza un aprendizaje más eficiente y estable en entornos complejos.

La elección de PPO responde a tres ventajas principales. En primer lugar, la **estabilidad**, ya que introduce una restricción mediante un *clipped objective* que evita actualizaciones demasiado bruscas en la política. En segundo lugar, la **eficiencia de muestreo**, pues permite reutilizar lotes de experiencias durante varias épocas de optimización, reduciendo así el coste computacional. Finalmente, la **escalabilidad**, dado que es capaz de manejar tanto observaciones vectoriales como visuales, lo cual resulta fundamental en este trabajo, donde se utilizan ángulos articulares junto con entradas de cámara.

La implementación de PPO en ML-Agents se parametriza cuidadosamente para garantizar una convergencia estable y eficiente. El *batch size* se fija en 1024, controlando el número de experiencias usadas en cada actualización. El *buffer size* se marca en 40 960, lo que define el tamaño total de experiencias acumuladas antes de cada optimización. La tasa de

aprendizaje (*learning rate*) se mantiene constante en $3 \cdot 10^{-4}$ tras verificar que proporcionaba buenos resultados. El número de épocas (*num_epoch*) se configura en 3, lo que permite realizar varias pasadas de optimización sobre cada *buffer*. Por último, el valor de gamma se fija en 0.99, actuando como factor de descuento para ponderar la importancia de las recompensas futuras.

```yaml
behaviors:
  IRB120Behavior:
    trainer_type: ppo
    hyperparameters:
      batch_size: 1024
      buffer_size: 40960
      learning_rate: 3.0e-4
      beta: 5.0e-3
      epsilon: 0.2
      lambd: 0.95
      num_epoch: 3
      learning_rate_schedule: constant
    network_settings:
      normalize: true
      hidden_units: 128
      num_layers: 2
      # (no vis_encode_type)
    reward_signals:
      extrinsic:
        gamma: 0.99
        strength: 1.0
    max_steps: 2000000
    time_horizon: 128
    summary_freq: 10000
    checkpoint_interval: 50000
    keep_checkpoints: 10
engine_settings:
  no_graphics: false
```

Figura 6. Estructura del archivo de entrenamiento (YAML)

4.1.1 DISEÑO DEL AGENTE

El agente se define como un componente de ML-Agents asociado al prefab del IRB120. Su arquitectura de red neuronal sigue la configuración establecida en el archivo YAML de entrenamiento (Figura 6):

- Capa de entrada: recibe como entradas concatenadas (a) los estados articulares (posiciones y velocidades), (b) la posición relativa efector–objetivo y (c) la imagen 64×64×3 de la cámara virtual.

- Codificación visual: la observación de la cámara se procesa mediante un codificador convolucional de tipo *simple* (CNN con capas de 3×3 filtros).

- Capas ocultas: dos capas totalmente conectadas de 128 neuronas cada una, con activación ReLU.

- Memoria recurrente: se incluye una LSTM con tamaño 128 y secuencia de 32 pasos, lo que permite al agente explotar información temporal y suavizar la toma de decisiones.

- Capa de salida: genera 6 valores de acción continua (uno por cada articulación), normalizados en el rango [-1, 1].

Este diseño refleja un equilibrio entre complejidad y eficiencia: suficiente capacidad para procesar información multimodal (vectorial + visual), pero sin llegar a un sobreajuste que comprometa la generalización.

4.1.2 Proceso de entrenamiento

El proceso de entrenamiento en Unity ML-Agents sigue una interacción iterativa entre el entorno y el algoritmo PPO:

1. Inicialización: se lanza la escena en Unity con el agente IRB120, configurado para recibir decisiones a intervalos de un paso de simulación (0,02 s).

2. Generación de episodios: cada episodio comienza con el robot en una configuración inicial aleatoria y con el objetivo (cubo) situado en distintas posiciones dentro de un área prefijada.

3. Recogida de experiencias: en cada paso, el agente observa el estado, genera una acción continua y recibe la recompensa correspondiente. Estas transiciones (s, a, r, s’) se almacenan en el buffer.

4. Optimización: una vez alcanzado el tamaño del buffer (40960 experiencias), se ejecuta la fase de entrenamiento. PPO actualiza la política mediante gradiente descendente sobre lotes de tamaño 1024, durante 3 épocas [18] , [19].

5. Reinicio: al finalizar un episodio por éxito, colisión o agotamiento de pasos, el entorno se reinicia y se vuelve al paso 2.

El entrenamiento se ejecutó durante un total de $2{\cdot}10^6$ pasos de simulación, lo que corresponde a unas 4 horas de cómputo en el sistema CPU empleado para el entrenamiento.

Durante este proceso, ML-Agents genera automáticamente los ficheros de resumen en TensorBoard, donde se monitorizan métricas como la recompensa media, la entropía de la política o la magnitud de los gradientes. Asimismo, cada cierto número de pasos (50.000 en este caso) se almacenan checkpoints en formato ONNX, lo que permite reanudar o evaluar el modelo entrenado [16].

4.2 DEFINICIÓN DEL ENTORNO DE ENTRENAMIENTO

El entorno de entrenamiento se implementa en **Unity** a partir de la adaptación del modelo cinemático-dinámico del robot ABB IRB120. Este modelo se describe originalmente en el fichero *irb120.xml* de MuJoCo y posteriormente se transfiere al motor físico de Unity mediante componentes *ArticulationBody*. Este proceso garantiza la preservación de las propiedades dinámicas esenciales, como masas, inercias, límites articulares y fricciones, dentro de un marco de simulación plenamente compatible con ML-Agents [23].

La escena de Unity incorpora varios elementos principales que permiten la interacción del agente con su entorno. En primer lugar, el **plano de trabajo** actúa como superficie estática de referencia para el movimiento del robot y como punto de apoyo para el posicionamiento del objetivo [24]. El **robot IRB120** se ensambla de manera jerárquica mediante enlaces articulados, donde cada junta se controla a través de consignas continuas aplicadas sobre los *drives* de *ArticulationBody*. El **objetivo** o *target* está representado por un cubo que se coloca

en posiciones aleatorias dentro de un área predefinida, lo que permite aumentar la robustez del aprendizaje y la capacidad de generalización del agente.

Además, se incorporan **sensores visuales** en forma de cámaras virtuales, acopladas al efector final o en puntos estratégicos de la escena. Estas cámaras están configuradas con una resolución de *RenderTexture* de 64×64 en RGB, proporcionando observaciones visuales directas al agente. Finalmente, se definen **colisionadores** en todos los enlaces del robot y en los objetos del entorno, lo que permite detectar colisiones [27] . Este aspecto resulta fundamental tanto para la definición de la señal de recompensa como para la correcta terminación de los episodios de entrenamiento.

4.2.1 ESPACIO DE OBSERVACIÓN

El vector de observación se construye a partir de la concatenación de diferentes modalidades de información: **Observaciones articulares**: posiciones y velocidades de las seis articulaciones del IRB120, normalizadas entre [-1, 1] para favorecer la estabilidad del aprendizaje. **Relación efector–objetivo**: vector que representa la posición relativa del efector respecto al cubo objetivo en coordenadas cartesianas. **Entradas visuales**: imágenes RGB de 64×64 píxeles capturadas por la cámara virtual, procesadas mediante un encoder convolucional.

La unión de estas observaciones vectoriales y visuales dota al agente de una percepción multimodal, lo que incrementa su capacidad para resolver tareas complejas de manipulación en el entorno.

4.2.2 ESPACIO DE ACCIÓN

El espacio de acción es **continuo y de dimensión 6**, correspondiente al número de grados de libertad del robot. Cada acción generada por la red neuronal representa un incremento de consigna para la posición angular de cada articulación, dentro de un rango limitado por los topes mecánicos de la estructura.

Esta formulación tiene dos ventajas principales: Permite un **control suave y progresivo**, evitando saturaciones abruptas en el movimiento y asegura la **consistencia física** con las restricciones del robot real, manteniendo fidelidad al modelo ABB IRB120.

4.2.3 FUNCIÓN DE RECOMPENSA

La función de recompensa es el componente que marca la dirección del entrenamiento y define qué comportamientos aprende el agente. Su diseño es crítico porque de él dependen la precisión de las trayectorias, la eficiencia del aprendizaje y la seguridad de los movimientos.

El criterio principal es la distancia euclídea entre el efector y el objetivo. El agente recibe una penalización proporcional a esa distancia: cuanto más cerca esté, menor es el castigo. Esto empuja al sistema a reducir el error de posición. Cuando el efector alcanza el objetivo dentro de un margen de 2 cm, se concede una recompensa positiva adicional, que refuerza el comportamiento correcto.

Se contemplan también varios castigos. Cualquier colisión con el plano de trabajo o con el propio robot genera una penalización inmediata. Con ello se busca que los movimientos sean más seguros. Además, se aplica un coste energético, de forma que acciones innecesarias o demasiado rápidas resultan penalizadas. El objetivo es promover trayectorias suaves y con menor consumo. Por último, si un episodio llega al número máximo de pasos sin éxito, se añade una penalización por tiempo, lo que evita que el agente explore trayectorias demasiado largas e ineficaces.

4.2.4 CONDICIONES DE TERMINACIÓN

Un episodio finaliza en cualquiera de los siguientes casos:

Éxito: el efector final se posiciona dentro de la zona de tolerancia definida alrededor del cubo. Colisión: el robot entra en contacto con objetos o con su propio cuerpo en configuraciones no deseadas. Límite temporal: se alcanza el máximo de pasos por episodio sin haber cumplido los objetivos.

De esta manera, cada episodio proporciona una señal binaria de éxito o fallo, acompañada de un perfil denso de recompensas que permite a PPO ajustar la política con granularidad.

Capítulo 5. ANÁLISIS DE RESULTADOS

La Tabla 2 resume la evolución de las métricas de rendimiento desde los 50.000 hasta los 2.000.000 pasos de entrenamiento. A medida que avanza el entrenamiento, todas las variables clave mejoran: la recompensa media aumenta, la tasa de éxito crece de forma sostenida, la distancia al objetivo se reduce y las colisiones son cada vez menos frecuentes.

Métrica	**50.000 steps**	**1.000.000 steps**	**2.000.000 steps**
Recompensa media	-1.82	0.05	0.57
Tasa de éxito (%)	0.0	62 %	78 %
Distancia media (m)	0.352	0.112	0.049
Steps / episodio	986	766	686
Colisiones/ episodio	27	10	5

Tabla 2. Resultados del entrenamiento

En la primera etapa, alrededor de los 50.000 pasos, el agente muestra un desempeño muy limitado. La recompensa media es negativa (-1.82) y la tasa de éxito es del 0 %, lo que indica que no completa episodios con éxito. Los intentos son largos, con una duración media de 986 pasos, y se registran aproximadamente 27 colisiones por episodio. La distancia media entre el efector y el objetivo se mantiene en 0.352 m, señal de que el agente todavía no consigue acercarse de manera efectiva al punto de destino.

A los 1.000.000 de pasos los resultados mejoran notablemente. La recompensa media pasa a valores positivos (0.05), la tasa de éxito alcanza el 62 %, y la distancia media al objetivo baja a 0.112 m. El número de colisiones se reduce a unas 10 por episodio y la duración de

los episodios cae a 766 pasos, señal de que el agente empieza a generar trayectorias más eficientes.

En el estado final del entrenamiento (2.000.000 pasos), las métricas muestran una clara convergencia. La **recompensa media se sitúa en 0.57**, con una tasa **de éxito del 78 %,** distancia media al objetivo de tan solo 0.049 m y una reducción de colisiones hasta 5 por episodio. Los episodios, además, se completan con 686 pasos de media, lo que supone una mejora del 30 % respecto al inicio.

Estos resultados confirman que el agente, entrenado en Unity con ML-Agents, logra aprender una política efectiva que maximiza la recompensa y reduce tanto la distancia al objetivo como las colisiones, garantizando un desempeño eficiente y seguro.

5.1 MÉTRICAS IMPORTANTES

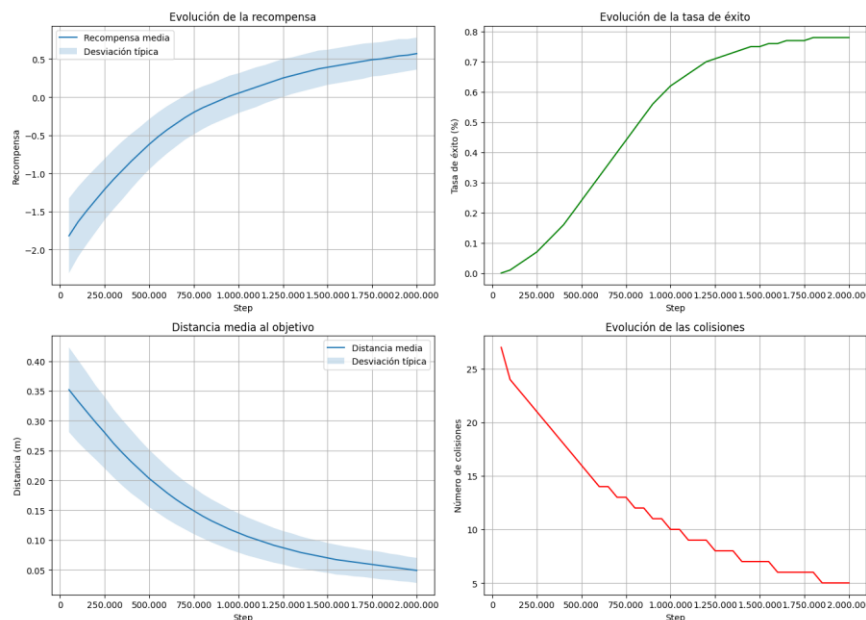


Figura 7. Evolución de métricas importantes durante el entrenamiento

La Figura superior muestra la evolución de la recompensa media a lo largo de los 2 millones de pasos de entrenamiento. Se observa una transición desde valores negativos iniciales de -

1.82 en 50.000 pasos hasta estabilizarse en torno a 0.57 en el estado final. El intervalo de desviación típica, que alcanza 0.49 en el inicio, se reduce hasta valores cercanos a 0.21, lo que confirma la convergencia de la política de entrenamiento y una menor dispersión de resultados entre episodios.

También se observa como la tasa de éxito experimenta un crecimiento sostenido desde valores próximos al 0 % en las primeras fases hasta alcanzar el 78 % en torno a 2.000.000 de pasos. La curva presenta una pendiente más pronunciada entre 400.000 y 1.000.000 de pasos, indicando que en ese intervalo se concentra la mayor ganancia de rendimiento del agente.

De la misma manera, se representa la distancia media entre el efector y el objetivo. La magnitud desciende de 0.352 m al inicio del entrenamiento a 0.049 m en el estado final, con un comportamiento claramente decreciente y estable. Este resultado implica que, en promedio, el agente es capaz de posicionar el efector a menos de cinco centímetros del objetivo, lo que constituye un nivel de precisión adecuado para la tarea planteada.

También se puede observar la evolución de las colisiones por episodio. Aunque no constituye la métrica principal de optimización, sí resulta un indicador de seguridad del comportamiento aprendido. El número de colisiones pasa de 27 en la fase inicial a únicamente 5 en el estado final, lo que refleja una exploración más controlada y una política menos propensa a trayectorias erráticas.

En conjunto, las curvas confirman que el agente entrenado en Unity con ML-Agents presenta una progresión estable hacia políticas cada vez más eficientes, con un incremento de la recompensa, un aumento de la tasa de éxito y una reducción tanto en la distancia media al objetivo como en la frecuencia de colisiones.

Capítulo 6. CONCLUSIONES Y TRABAJOS FUTUROS

El trabajo demuestra la viabilidad de migrar un modelo robótico originalmente implementado en MuJoCo al entorno Unity con el paquete ML-Agents, validando su uso para entrenar agentes mediante algoritmos de aprendizaje por refuerzo profundo. La conversión del ABB IRB120 ha requerido la reconstrucción completa de la geometría y la jerarquía cinemática, la parametrización de articulaciones y la adaptación de dinámicas en Unity, así como la definición de un entorno de entrenamiento con recompensas y observaciones coherentes con la tarea planteada.

Los experimentos de entrenamiento con el algoritmo Proximal Policy Optimization (PPO) evidencian que el sistema en Unity es capaz de aprender políticas estables y eficientes. A lo largo de $2 \cdot 10^6$ pasos de simulación, la recompensa media evoluciona desde valores negativos (-1.82) hasta estabilizarse en 0.57, mientras que la tasa de éxito alcanza un 78 % y la distancia media entre efector y objetivo se reduce a menos de 5 cm. Estos resultados confirman que Unity, pese a no disponer de un motor físico tan preciso como MuJoCo, proporciona un marco adecuado para la experimentación en aprendizaje por refuerzo.

En términos de aportaciones, el trabajo no solo pone de manifiesto el procedimiento técnico para trasladar un modelo robótico desde MuJoCo a Unity, sino que también establece una metodología reproducible que combina fidelidad geométrica, coherencia dinámica y compatibilidad con librerías de entrenamiento en Python. Asimismo, los resultados obtenidos demuestran la capacidad de Unity para entrenar manipuladores articulados en tareas de posicionamiento, lo que abre la puerta a escenarios más complejos en los que intervengan visión artificial o interacción multimodal.

De cara a trabajos futuros, se identifican varias líneas de desarrollo. Una primera mejora consiste en extender la función de recompensa para incluir criterios adicionales como suavidad de trayectorias o eficiencia energética, lo que permitiría políticas más realistas en un contexto industrial. Asimismo, sería interesante evaluar otros algoritmos de DRL

disponibles en ML-Agents, como SAC o DDPG, y analizar su impacto en estabilidad y convergencia. Otra línea relevante es la integración de técnicas de domain randomization, que facilitarían la transferencia de políticas entrenadas en Unity hacia un robot físico, reduciendo la brecha sim-to-real. Finalmente, la combinación con ROS mediante conectores intermedios abriría la posibilidad de validar las políticas aprendidas directamente sobre hardware real, consolidando Unity como una herramienta intermedia entre la simulación puramente académica y la robótica aplicada.

En conclusión, este proyecto cumple con los objetivos planteados: la adaptación exitosa del ABB IRB120 a Unity, la validación de su entrenamiento mediante ML-Agents y el análisis de los resultados alcanzados. Se establece así una base sólida sobre la que construir futuros trabajos orientados a la transferencia sim-to-real y al diseño de entornos de aprendizaje cada vez más realistas y exigentes.

Capítulo 7. BIBLIOGRAFÍA

[1] Robotis. “Unity Simulation”. ROBOTIS e-Manual. https://roboti.us/book/unity.html.

[2] Dong, Lixiang. *Memoria del Máster Universitario en Ingeniería de Control Industrial*. Universidad Pontificia Comillas, 2020. https://repositorio.comillas.edu/xmlui/bitstream/handle/11531/46852/MemoriaMIC_Lixiang_Dong.pdf?sequence=2

[3] *MuJoCo Documentation.* “Overview.” *MuJoCo Read the Docs*. https://mujoco.readthedocs.io/en/stable/overview.html

[4] *Ye, Linqi.* Rankun Li, Xiaowen Hu, Jiayi Li, Boyang Xing, Yan Peng, Bin Liang, IROS 2025 Workshop Paper (PDF). Unity RL Playground: A Versatile Reinforcement Learning Framework for Mobile Robots*IROS 2025*, 2025. https://linqi-ye.github.io/docs/iros25v2.pdf.

[5] *Ilosvay, Viktória Brigitta.* “Unity ML Agents: Wall Jump and SoccerTwos environment using Reinforcement Learning (RL) technique.” *Final Project Thesis*, enero 2024. https://www.researchgate.net/publication/377273320_Unity_ML_Agents_Wall_Jump_and_SoccerTwos_environment_using_Reinforcement_Learning_RL_tecnique

[6] *Bertsekas, Dimitri P. A Course in Reinforcement Learning (2nd Edition)*. MIT, 21 de junio de 2025. https://www.mit.edu/~dimitrib/RLCOURSECOMPLETE%202ndEDITION.pdf

[7] *Murphy, Kevin P.* “Reinforcement Learning: An Overview.” *CoRR (arXiv)*, diciembre 2024. https://arxiv.org/abs/2412.05265

[8] *Rahul, Vaddadi Sai & Chakraborty, Debajyoti.* “Exploring reinforcement learning techniques for discrete and continuous control tasks in the MuJoCo environment.” *arXiv*, julio 2023. https://arxiv.org/abs/2307.11166

[9] Balint-H. *MuJoCo + Unity ML-Agents Tutorials.* GitHub repository, 2025. https://github.com/Balint-H/mj-unity-tutorial

[10] Kaup, Michael; Wolff, Cornelius; Hwang, Hyerim; Mayer, Julius; Bruni, Elia. *A Review of Nine Physics Engines for Reinforcement Learning Research.* arXiv preprint, julio 2024. https://arxiv.org/html/2407.08590v1

[11] Unity Technologies. “Entrena a tus agentes 7 veces más rápido con ML-Agents.” *Unity Blog (español)*, noviembre 2019. https://unity.com/es/blog/engine-platform/training-your-agents-7-times-faster-with-ml-agents

[12] Peake, Ian; La Delfa, Joseph; Bejarano, Ronal; Blech, Jan Olaf. “Simulation Components in Gazebo.” *ResearchGate*, octubre 2024. https://www.researchgate.net/publication/352525101_Simulation_Components_in_Gazebo

[13] Igras-Cybulska, Magdalena; Kolber-Bugajska, Barbara; Salamon, Rafał; et al. “Supporting Unity Developers with an AI-Powered Asset: Insights from an Exploratory User Study on Multiplayer Game Development.” *Proceedings of the 1st International Workshop on Designing and Building Hybrid Human-AI Systems* (co-located with AVI 2024), junio 2024. https://www.researchgate.net/publication/381292625_Supporting_Unity_Developers_with_an_AI-Powered_Asset_Insights_from_an_Exploratory_User_Study_on_Multiplayer_Game_Development .

[14] *Unity Asset Creation. ResearchGate*. — El enlace no permite obtener fecha ni autores claros. https://www.researchgate.net/publication/347422363_Unity_Asset_Creation ResearchGate

[15] Kritz, Joshua; de Roos, Mart; Ferreira Pires, Luís; Moreira, João; Guizzardi, Giancarlo. “UgameFeature: Automatic Code Generation for Unity Game Projects.” *10th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2022)*, enero 2022. https://www.researchgate.net/publication/358598450_UgameFeature_Automatic_Code_Generation_for_Unity_Game_Projects

[16] *[0910.2405] Generating Concise and Readable Summaries of XML Documents. arXiv* preprint, fecha no disponible. https://arxiv.org/abs/0910.2405

[17] “Features of preparing 3D objects modeled in Blender for import into Unity 3D.” *ResearchGate*, agosto 2025. https://www.researchgate.net/publication/353694254_FEATURES_OF_PREPARING_3D_OBJECTS_MODELED_IN_BLENDER_FOR_IMPORT_INTO_UNITY_3D ResearchGate

[18] Schulman, John; Wolski, Filip; Dhariwal, Prafulla; Radford, Alec; Klimov, Oleg. “Proximal Policy Optimization Algorithms.” *arXiv preprint*, julio 2017. https://arxiv.org/abs/1707.06347

[19] Amirzadeh, Rasoul; Thiruvady, Dhananjay R.; Nazari, Asef; Ee, Mong Shan; y otros. "A Framework for Empowering Reinforcement Learning Agents with Causal Analysis: Enhancing Automated Cryptocurrency Trading." *arXiv preprint*, octubre 2023. https://arxiv.org/abs/2310.09462

[20] Unity Technologies. *ML-Agents API documentation*. Unity Documentation (packages), versión 2.0. https://docs.unity3d.com/Packages/com.unity.ml-agents@2.0/api/index.html

[21] *Adaptive Training of Value Functions via a Generative Model*. Artículo en actas de congreso de MIT (2017). https://dspace.mit.edu/bitstream/handle/1721.1/146162/3512290.3528844.pdf?sequence=1&isAllowed=y

[22] Tobias Huber, Maximilian Demmler. "GANterfactual-RL: Understanding Reinforcement Learning Agents' Strategies through Visual Counterfactual Explanations. Universidad de Southampton. 2023. https://www.southampton.ac.uk/~eg/AAMAS2023/pdfs/p1097.pdf

[23] *Lucas A.E Pineda Metz, An Evaluation of Unity ML-Agents Toolkit for Learning Boss – MSc Thesis*. 2020. https://skemman.is/bitstream/1946/37111/1/An%20Evaluation%20of%20Unity%20ML-Agents%20Toolkit%20for%20Learning%20Boss%20-%20MSc%20Thesis.pdf

[24] Yusef Savid, Simulated Autonomous Driving Using Reinforcement Learning: A Comparative Study on Unity's ML-Agents Framework." *ResearchGate Mayo 2023*. https://www.researchgate.net/publication/370795300_Simulated_Autonomous_Driving_Using_Reinforcement_Learning_A_Comparative_Study_on_Unity's_ML-Agents_Framework

[25] González Petit, Fernando. *Unity HEX BattleGame: Reinforcement Learning con Unity ML-Agents para entornos multi-agente y toma de decisiones bajo demanda*. Trabajo de Fin de Grado, Universidad de La Laguna, septiembre 2020. https://riull.ull.es/xmlui/handle/915/21354

[26] Unity Technologies. *Training ML-Agents*. Unity ML-Agents Toolkit Documentation (GitHub Pages). https://unity-technologies.github.io/ml-agents/Training-ML-Agents/

[27] Andersson, Pontus. *Future-proofing Video Game Agents with Reinforced Learning and Unity ML-Agents*. Trabajo de Fin de Grado, Luleå University of Technology, Departamento de Computer Science, Electrical and Space Engineering, 2021. https://www.diva-portal.org/smash/get/diva2:1605238/FULLTEXT01.pdf

ANEXO I

[1] Enlace al repositorio de GitHub con todo el código usado para el proyecto:

https://github.com/jaimepri/IRB120_PPO_Unity