



UNIVERSIDAD
POLITÉCNICA
DE MADRID

POLITÉCNICA



Universidad Politécnica de Madrid

Escuela Técnica Superior de Ingeniería y Diseño Industrial

Trabajo de Fin de Grado

Estudio del aprendizaje por refuerzo en
robótica y aplicación práctica en el proyecto
Metatool

Autor: Enrique de Antonio

Tutor: Pfr. Miguel Hernando Gutierrez

Cotutor: Dr. Virgilio Gómez Lambo

Departamento: Ingeniería Eléctrica, Automática y Física Aplicada

Madrid, febrero 2026

Índice general

1. Introducción	5
1.1. Contexto y motivación	5
1.2. Objetivos del trabajo	6
1.3. Alcance y limitaciones	6
1.4. Metodología	8
1.5. Estructura del documento	10
2. Estado del arte del Aprendizaje por Refuerzo (RL) en robótica	11
2.1. Introducción al RL aplicado a la robótica	11
2.2. Aplicaciones en manipulación	12
2.3. Aplicaciones en locomoción	13
2.4. Otras disciplinas	13
2.5. Conclusiones del estado del arte	14
3. Fundamentos teóricos del aprendizaje por refuerzo	16
3.1. El aprendizaje por refuerzo dentro del aprendizaje automático	16
3.2. Estructura del aprendizaje por refuerzo	17
3.3. Proceso de decisión Markov o MDP	18
3.3.1. Formulación de los MDP	19
3.3.2. Ejemplo de un MDP	21
3.3.3. Funciones de Valor y Ecuación de Bellman	23
3.3.4. Política óptima y Valor óptimo	27
3.4. Algoritmos clásicos del aprendizaje por refuerzo	28
3.4.1. Programación Dinámica	29

3.4.2. Método Monte Carlo	31
3.4.3. Temporal Difference o TD	34
3.4.4. On-Policy vs Off-Policy	35
3.5. Consideraciones para estados continuos	36
3.5.1. Parametrización de los estados, w	37
3.5.2. Vector de caracterización de los estados, x	37
3.6. Algoritmos modernos (Aprendizaje profundo)	38
3.7. Observaciones para los ejercicios prácticos	39
4. Análisis de la herramienta IsaacLab	41
4.1. ¿Qué es IsaacLab?	41
4.2. Estructura de la herramienta	42
4.3. Arquitectura de entornos	44
4.3.1. Arquitectura directa	45
4.3.2. Manager Based	45
4.4. Estructuras de datos	47
4.4.1. Clases	48
4.4.2. Tensores: PyTorch	48
4.5. Entrenamiento de agentes	49
4.6. Evaluación de agentes	50
4.7. Análisis Global	51
5. Estudio caso locomoción	52
5.1. Descripción caso práctico	52
5.2. Diagrama de Clases	53
5.3. Análisis de clases	56
5.3.1. DirectRLEnv	56
5.3.2. DirectRLEnvCfg	58
5.3.3. AntEnvCfg	59
5.3.4. ArticulationCfg	61
5.3.5. LocomotionEnv	63

5.4. Registro del Entorno	75
5.5. Aprendizaje y Evaluación	77
5.6. Posibles mejoras	79
5.6.1. Terreno irregular	80
5.6.2. Cámaras	80
6. Reach	83
6.1. Descripción del caso práctico	83
6.2. Diagrama de clases	84
6.3. Manejadores	86
6.4. Configuración y registro del entorno específico	92
6.5. Entrenamiento y evaluación	93
6.6. Mejoras y correcciones	94
6.7. Conclusiones: Directo contra Manejadores	96
7. Trabajo dentro del proyecto MetaTool	97
7.1. Misión y objetivos	98
7.2. Objetivo de la aportación	98
7.3. Levantamiento herramienta	99
7.4. Depuración del arraste con herramienta	101
7.4.1. Conclusiones	105
8. El problema Sim2Real	106
8.1. Enfoques	106
8.2. Aleatorización del dominio.	107
8.3. Implementación en IsaacSim	108
8.4. Implementación en robot real	110
8.4.1. Cliente Python	110
8.4.2. Servidor robot	112
8.5. Ensayos	114
8.5.1. Prueba simulada	114

8.5.2. Prueba real	117
9. Conclusiones Finales	119
9.1. Resultados	119
10. Aportaciones del trabajo	121
10.1. Limitaciones	122
10.2. Trabajos futuros	122
Anexo A. Repositorios de código	130
Anexo B. Vídeos de resultados experimentales	130
Anexo C. Resultados completos de los ensayos	130

Capítulo 1

Introducción

1.1. Contexto y motivación

Según la página oficial de *Nvidia* [1], empresa considerada una de las mayores impulsoras de esta disciplina, el aprendizaje por refuerzo es una técnica de aprendizaje automático que permite a los robots tomar decisiones basadas en la experiencia. En este trabajo de final de grado se estudia esta doctrina; con el objetivo de entender sus conceptos fundamentales y desarrollar (mediante la herramienta de *Nvidia*, *IsaacLab*) distintos entornos capaces de implementar este procedimiento.

La inteligencia artificial es actualmente una tecnología puntera con una gran cantidad de aplicaciones. Concretamente, el aprendizaje automático se ha aplicado en disciplinas como la medicina, en la generación de reportes de imágenes médicas; las finanzas, en la gestión de órdenes de compra; o la energía, en sistemas de refrigeración de servidores [2]. En la robótica concretamente, ha tomado un gran protagonismo. En esta disciplina, grandes empresas del sector, como *Boston Dynamics* han empezado a utilizar aprendizaje automático en múltiples tareas [3].

El interés en este proyecto surge de la posibilidad de aplicar esta herramienta dentro del proyecto *ROMERIN*, un robot modular escalador destinado a la inspección de infraestructuras [4]. Debido a la complejidad del aprendizaje por refuerzo, se identificó la necesidad de realizar un estudio completo. Esto, añadido a la cesión de recursos del proyecto *MetaTool* en la formación, llevó a la colaboración dentro de este proyecto, centrada en el análisis y la revisión de código.

Antes de comenzar el trabajo, en este capítulo, se presentarán los objetivos y contenidos del mismo. De esta forma, se obtendrá una visión clara de las ideas principales y la estructura del documento. En el siguiente apartado, se enumerarán los principales objetivos de este trabajo.

1.2. Objetivos del trabajo

Este proyecto tiene como objetivo realizar un estudio del aprendizaje por refuerzo y la herramienta IsaacLab, para lo cual se fijan dos objetivos principales. En primer lugar, asentar una base teórica sólida tanto del aprendizaje por refuerzo como de la herramienta IsaacLab. Se espera que futuros estudiantes puedan basarse en ella para realizar trabajos en esta disciplina. En segundo lugar, realizar labores dentro del proyecto europeo MetaTool, con el fin de adquirir experiencia práctica.

Para alcanzar estos objetivos, se proponen una serie de objetivos específicos:

1. Obtener una visión general del impacto del aprendizaje por refuerzo en el campo de la robótica.
2. Estudiar las bases del aprendizaje por refuerzo, centrándose en su estructura, fundamentos matemáticos y sus principales algoritmos.
3. Explicar el funcionamiento de la herramienta *IsaacLab* para su aplicación en el ámbito del aprendizaje por refuerzo.
4. Analizar ejemplos de dicha herramienta, para así profundizar en ella y proporcionar una guía práctica para trabajos futuros.
5. Realizar un estudio del proyecto *MetaTool*: misión y visión.
6. Realizar trabajos prácticos en el proyecto con la herramienta *IsaacLab*
7. Estudiar el problema *Sim2Real* y sus posibles soluciones
8. Desarrollar un código para la implementación de políticas.

Para el cumplimiento de estos objetivos, se deberá tener en cuenta todo lo que se abarcará y cómo dicho alcance se adapta a los objetivos propuestos. En el siguiente apartado, e desarrollará este análisis..

1.3. Alcance y limitaciones

En este TFG se cubre el proceso para realizar entornos de aprendizaje por refuerzo. Al tener este objetivo en mente, en este TFG se abordarán distintos aspectos de esta disciplina. Este trabajo contempla desde los fundamentos del aprendizaje, hasta las distintas estructuras de datos, clases y ficheros empleados para la ejecución y simulación de entornos.

En primer lugar, para contextualizar el trabajo dentro del aprendizaje por refuerzo en robótica, se revisará el estado actual del arte. Se presentarán los avances más importantes en distintos campos de la robótica, entre ellos la manipulación, la locomoción y otras aplicaciones como drones, navegación o dispositivos de visión.

A continuación, se explicará la teoría fundamental del aprendizaje por refuerzo. En primer lugar, se presentará la estructura principal que se utiliza en esta disciplina y sus partes, entre las que se encuentran los agentes, los entornos y sus interacciones (acciones, observaciones y recompensas). Dentro de este marco teórico se presentarán los procesos de decisión de Markov (MDP), en los cuales se fundamentan los algoritmos que se utilizarán. Una vez estudiados estos conceptos, se presentarán diversos algoritmos, desde los más simples (*Monte Carlo*, *TD*) hasta aquellos que se utilizarán en las simulaciones (*PPO*, *SAC*, *A2C*).

Una vez desarrollado el marco teórico, se procederá a introducir la herramienta *IsaacLab*. Después de una primera introducción a la herramienta, se comenzará a explicar sus distintas funcionalidades. Primero, se explicará qué es la herramienta y cómo se estructuran las simulaciones dentro de la aplicación. Después, se analizarán las dos principales arquitecturas de los entornos: la forma directa y la forma basada en manejadores. Definidas las arquitecturas, se estudiarán las principales estructuras de datos: las clases y los tensores. Por último, se estudiará cómo, una vez definidos los entornos, se realiza el aprendizaje y se evalúa el resultado final.

A continuación, se analizarán dos casos prácticos de la herramienta *IsaacLab*, cada uno construido a partir de una arquitectura diferente. Para ambos ejemplos, primero se presentará la tarea escogida y la motivación que justifica esta elección. Seguidamente, se presentará el diagrama de clases que describe el entorno a entrenar. Este diagrama de clases se diseccionará, analizando cada una de ellas, con especial interés en sus atributos y métodos. Una vez analizado el entorno, se estudiará la ejecución del aprendizaje, valorando después el resultado final del proceso. Por último, se presentarán algunas mejoras posibles dentro del ejercicio.

Habiendo estudiado las distintas características del aprendizaje por refuerzo (RL) y la herramienta, se desarrollará la participación dentro del proyecto MetaTool. En este capítulo, se definirá el contexto del proyecto *MetaTool* y cuál es su principal objetivo. Seguidamente, se concretarán las tareas en las cuales se intervendrá y el objetivo de la participación. Para cada caso abordado, se expondrán los problemas afrontados y las soluciones tomadas.

Otro punto importante del aprendizaje por refuerzo que se estudiará será el problema del Sim2Real, que consiste en la aplicación de las políticas entrenadas en simulación para el control en el entorno real. Primero, se presentará el concepto de Sim2Real y sus

principales desafíos. Seguidamente, se enumerarán y analizarán las distintas técnicas para realizar este trasvase a la realidad. Finalmente, se desarrollara código para la implementación de las políticas en robots reales.

En la última parte del trabajo, se expondrán las conclusiones del proyecto en su conjunto, valorando las aportaciones realizadas, las dificultades encontradas y las oportunidades para trabajos futuros.

Este trabajo, debido a la gran extensión de esta disciplina, se dejan fuera algunos paradigmas. Por un lado, únicamente se estudia, dentro del aprendizaje automático, el aprendizaje por refuerzo, dejando fuera el aprendizaje supervisado y no supervisado. Además, aunque se emplee aprendizaje por refuerzo profundo, no se profundizará en la base matemática de sus algoritmos, así como en las bibliotecas que los implementan. Se estudiarán las características de las redes neuronales, pero no se profundizará en su formulación matemática.

Este será el alcance completo del trabajo. Sin embargo, se debe tener en cuenta un punto más antes de comenzar con las tareas prácticas: la metodología. En el siguiente apartado se cubrirá este tema.

1.4. Metodología

En este trabajo de final de grado se distinguen principalmente dos líneas: una parte teórica acerca del aprendizaje por refuerzo y una parte práctica mediante programación en Python y URscript. Por otro lado, la preparación de este documento se ha realizado después de ocho meses de trabajo en tareas de programación e investigación de forma independiente o en conjunto con el equipo de investigación del proyecto MetaTool. A continuación, se expondrá la metodología característica de cada apartado.

La introducción, en primer lugar, se ha preparado tras la realización de la mayoría de las labores teóricas y prácticas del trabajo. De este modo, al disponer de una visión general del conjunto del trabajo, se han expuesto las distintas características del mismo y su enfoque general.

Para el estado del arte, con el objetivo de mostrar una visión general del estado del aprendizaje por refuerzo en la robótica, se presentan distintos artículos de investigación sobre esta disciplina y sus aplicaciones en la práctica. Al realizar este ejercicio después de un periodo de aprendizaje y práctica, se han podido determinar los factores más importantes de cada artículo, así como seleccionar los artículos más relevantes.

En el apartado tres, en el cual se exponen los fundamentos teóricos del aprendizaje por refuerzo, se siguió la siguiente metodología. En primer lugar, se estudió un curso de

aprendizaje por refuerzo impartido por David Silver [5]. Mediante este curso se obtuvo una visión general de esta disciplina, entendiendo su estructura general y la fundamentos de sus algoritmos. Una vez obtenida una visión general, y tras aplicar estos conocimientos en labores prácticas, se estudió de forma más específica cada elemento, consultando distintas fuentes de información. Cabe resaltar, dentro de estas fuentes, el libro sobre aprendizaje por refuerzo de Sutton y Barto [6]. Sobre este se fundamenta la gran mayoría de definiciones formales.

El cuarto apartado, acerca de la herramienta *IsaacLab*, se abordó de manera distinta. Al ser *IsaacLab* una herramienta concreta y propiedad de una entidad privada, NVIDIA, existe menos diversidad de información acerca de ella. Por tanto, para su estudio, se utilizó principalmente la información contenida en sus fuentes oficiales. En este apartado concreto, se utilizaron principalmente los tutoriales proporcionados por la plataforma para el aprendizaje de su estructura y aplicación, así como los distintos glosarios de las funciones y clases, así como la información detallada acerca de la propia plataforma y sus bibliotecas. A esto se le sumó el conocimiento adquirido en el trabajo en conjunto con el equipo MetaTool, en especial con Virgilio Gómez, especialista de la plataforma y líder de la división en la que se trabajó.

Los apartado quinto y sexto, al constar de un análisis concreto de un ejemplo proporcionado por la plataforma, se utilizaron los conocimientos adquiridos en el apartado anterior. Para realizar este análisis se ha seguido el siguiente proceso. En primer lugar, se analizó todo el ejercicio en su conjunto, estudiando los distintos ficheros que intervienen en la ejecución. Con esta estructura identificada, se realizó un diagrama de clases, detallando los distintos atributos y las funciones utilizadas. Con este diagrama en mente, se estudió cada clase por sí sola, explicando la función de cada apartado del código y su aportación global al ejercicio.

A su vez, en el apartado séptimo se debe tratar de manera distinta, pues se trata de tareas dentro de un proyecto externo. Por ello, antes de analizar el ejercicio se realizó un pequeño estudio del proyecto general. Con el enfoque general en mente, se llevó a cabo un estudio de las tareas realizadas. A diferencia del apartado anterior, este estudio se basó en la depuración realizada del código, en el cual se llevaron a cabo una serie de correcciones para su correcto funcionamiento.

En el apartado octavo, enfocado a la implementación en el robot de políticas, se estudió un caso real de esta implementación [7], aunque aplicado a un robot distinto. Estudiado este caso, se diseñó un programa para su uso en el *UR3 Robohabilis* [8] [9], utilizando el lenguaje *URScript* [10] para el control del robot y algunas clases derivadas del caso estudiado.

Por último, una vez terminado el trabajo, se expondrán las conclusiones obtenidas

del mismo, basadas en la experiencia obtenida en el transcurso del proyecto.

En resumen, este trabajo sigue distintas metodologías, dependiendo de si se trata de un enfoque práctico o teórico. Considerando el conjunto del trabajo, se podría definir una metodología general. Primero, se realiza el estudio teórico, tanto del aprendizaje por refuerzo en general como de la herramienta específica. Después, se utilizan los conocimientos obtenidos para realizar estudios prácticos y analíticos.

Toda la metodología y alcance descritos se concretan en este documento. A continuación, se presenta cómo se encuentra estructurado.

1.5. Estructura del documento

El documento se estructura en capítulos y una serie de anexos. Los capítulos se agrupan en dos partes diferenciadas. En primer lugar, del capítulo 2 al 4, se introducen las bases teóricas relevantes para la ejecución de ejercicios de entrenamiento: el estado del arte, el marco teórico del aprendizaje por refuerzo y un estudio en profundidad de la herramienta IsaacLab. En segundo lugar, los capítulos 5, 6, 7 y 8 recogen las prácticas realizadas en el proyecto MetaTool y la implementación de una política. Estos capítulos vienen precedidos de una introducción, ya desarrollada, y cerrados por un capítulo de conclusiones finales. Aparte de la documentación, se incluyen una serie de anexos con los materiales relevantes para el trabajo: repositorios, material audiovisual y datos de ensayos.

Presentados el objetivo, la metodología, el alcance y la estructura del documento, se continúa con el desarrollo del trabajo, dando comienzo con el estado del arte.

Capítulo 2

Estado del arte del Aprendizaje por Refuerzo (RL) en robótica

2.1. Introducción al RL aplicado a la robótica

El aprendizaje por refuerzo es una forma de aprendizaje en la que, a través de interactuar con el entorno, se trata de maximizar una recompensa numérica [6, Pág. 1]. Este paradigma se caracteriza por no tener instrucciones definidas sobre cómo actuar y por una realimentación retrasada en el tiempo.

El primer ejemplo del uso de esta disciplina en la robótica se remonta a 1992, donde métodos de aprendizaje por refuerzo se aplicaron en un robot basado en comportamientos, *Obélix* [11]. En este experimento se utilizaba un algoritmo basado en un entorno de aprendizaje por refuerzo para que dicho robot empujase una caja.

En la actualidad, el aprendizaje por refuerzo está afianzado en la robótica como una disciplina de rápido desarrollo. Especialmente, el aprendizaje por refuerzo profundo, basado en la implementación del aprendizaje por refuerzo para crear redes neuronales profundas [12], ha obtenido resultados muy relevantes en estados con un gran número de dimensiones o altamente no lineales, donde otros métodos de control prueban ser muy ineficientes. Estos resultados se han mostrado en multitud de disciplinas dentro de este campo, como la locomoción, la navegación o la manipulación. Además, se ha mostrado también su efectividad tanto en robots individuales como colaborativos [13].

A continuación, se presentan distintos casos de éxitos para distintas disciplinas.

2.2. Aplicaciones en manipulación

La manipulación se da cuando un robot altera su entorno a través de contacto selectivo [14]. La manipulación presenta un gran desafío para cualquier método de aprendizaje, debido a la gran cantidad de observaciones y acciones necesarias para llevar a cabo distintas tareas, las cuales pueden llegar a ser bastante elaboradas. Todo esto lleva a un gran coste computacional y a una elevada complejidad a la hora de simular la física y los espacios de trabajo. Además, la transferencia del aprendizaje al mundo real se vuelve lento e inseguro. A pesar de esto, los métodos de aprendizaje por refuerzo profundo han obtenido un notable éxito dentro de esta disciplina [13].

Un ejemplo de esta aplicación se da en el artículo “*QT-Opt: Scalable Deep Reinforcement Learning for Vision-Based Robotic Manipulation*” [15]. En este trabajo se utilizan métodos de aprendizaje por refuerzo para generalizar el agarre de objetos desconocidos. Para ello, se emplea como entrada una cámara RGB para poder obtener datos acerca de la forma del objeto. A partir de esta entrada, se desarrolla un algoritmo denominado QT-opt para elegir una acción de agarre, conformando una función de valor acción-estado Q y resolviendo esta para obtener el máximo valor de éxito. En el apartado 3, se entrará en detalle sobre cómo se conforma esta función y los distintos algoritmos que se pueden usar para resolverla.

En la actualidad, se encuentran casos como “*DORA: Object Affordance-Guided Reinforcement Learning for Dexterous Robotic Manipulation*” [16]. En él, se propone una nueva aplicación de manipulación para el agarre de objetos siguiendo mapas de *affordances*. Los mapas de *affordances* codifican la superficie de un objeto según las regiones funcionales de este. Este mapa se incluye como información adicional al estado del MDP (concepto que se explica en el apartado 3.2). Combinando el RL con estos mapas se obtiene un agarre más funcional, permitiendo agarrar más veces un martillo por su mango.

Además de estos ejemplos de investigación, esta tecnología se ha comenzado a aplicarse en el entorno industrial. Covariant, una empresa dedicada a la implementación de la inteligencia artificial en la robótica [17], ha desarrollado un robot basado en modelos de aprendizaje por refuerzo. Este robot ha sido entrenado mediante datos multimodales e interacciones físicas reales con el objetivo de ejecutar diversas tareas de manipulación. Covariant sostiene que su robot RFM-1 es capaz de realizar tareas de segmentación e identificación a través de imágenes, así como ejecutar agarres a través de instrucciones de texto y observaciones [18].

2.3. Aplicaciones en locomoción

La locomoción en robótica tiene como objetivo utilizar los motores integrados del robot para transportarse por su entorno. Antes del desarrollo del aprendizaje profundo, la locomoción ya estaba estrechamente ligada a esta disciplina dando grandes avances en el desarrollo de cuadrúpedos. Ya entrada en la era del aprendizaje profundo, se llevó su implementación a otros problemas de locomoción, como robots bípedos [13].

Pese a que los primeros ejemplos de RL en locomoción se aplicaron a estos cuadrúpedos, el verdadero avance de estos llegó con la implementación del DRL [13]. En *RMA: Rapid Motor Adaptation for Legged Robots* [19] se propone un método para el control de cuadrúpedos en entornos rocosos. En este ejemplo se implementa el aprendizaje por refuerzo sobre una política adquirida mediante aprendizaje supervisado. De esta manera, mediante el aprendizaje supervisado se busca estimar un vector intrínseco del entorno que detalla sus propiedades. Posteriormente, en la fase de implementación, se aplica un algoritmo de aprendizaje por refuerzo, que recibe este vector como entrada, para adaptarse así al entorno actual. De este modo, se obtiene una gran eficiencia en nuevos entornos.

La locomoción bípeda, comparada a la locomoción de cuadrúpedos, consta de un problema más complejo. Debido a un menor número de apoyos, se produce una falta de redundancia y una reducción de la estabilidad, haciendo necesario un control más preciso y complejo. Sin embargo, gracias al aprendizaje por refuerzo profundo (DRL) han aparecido casos de éxito en este campo, logrando superar al control clásico en ciertos aspectos. En *Reinforcement Learning for Versatile, Dynamic, and Robust Bipedal Locomotion Control* [20], se presenta un modelo de aprendizaje para el control de bípedos. En él, proponen un doble representación del estado, combinando un registro a corto plazo con otro a largo. Gracias a esto, se obtiene un control robusto, pudiendo adaptarse a las distintas formas de contacto y los cambios en la estabilidad, manteniendo un proceso de adaptación continuo.

2.4. Otras disciplinas

Las disciplinas de locomoción y manipulación serán las que principalmente se tratan en este trabajo. Sin embargo, no son las únicas para las que esta tecnología ha sido utilizada. En este apartado, se estudiará su aplicación en algunas de estas disciplinas.

La navegación es una de estas disciplinas influenciadas por el aprendizaje por refuerzo. Según el estándar IEEE 172-1983, se define como el proceso de dirigir un vehículo a un destino [21]. No debe confundirse con la locomoción, que se centra en buscar cómo se debe dar este desplazamiento. En *Socially aware navigation for mobile robots: a survey on deep reinforcement learning approaches* [22] se muestran distintos enfoques de la

navegación donde se implementan algoritmos de DRL. En él se describe como estos algoritmos permiten integrar una capa social a esta navegación, buscando el confort humano, la percepción social, predicción, etc.

Otra disciplina que se ha visto beneficiada por el aprendizaje por refuerzo es la de los robots aéreos, especialmente los UAVs y sus enjambres. Un UAV, también conocido como *dron*, se define por sus propias siglas *Unmanned Aerial Vehicles*; es decir, son vehículos aéreos controlados que realizan tareas sin operación humana [23]. Un enjambre se puede definir como un conjunto de robots moviéndose conjuntamente y con un control compartido, mostrándose una cooperación entre los distintos integrantes del grupo [24]. Tanto la complejidad del control de los UAV como de sus grupos se ha abordado mediante herramientas de aprendizaje por refuerzo. Por ejemplo, en *Application of Deep Reinforcement Learning to UAV Swarming for Ground Surveillance* [25], se aplican algoritmos PPO para controlar los UAV, donde cada uno de ellos tiene un sub-agente entrenado con este algoritmo. Este enjambre se diseñó para tareas de vigilancia de áreas, pudiendo buscar y fijar objetivos terrestres.

Por último, pese a no ser una disciplina propia de la robótica, la visión artificial también ha sido influenciada por el RL. Esta suele integrarse dentro de modelos VLA (*Vision-Language-Action*). Estos son sistemas que toman observaciones visuales y instrucciones en lenguaje natural para generar órdenes de control [26]. Recientemente se han comenzado a introducir enfoques para integrar el RL dentro de estos modelos. Esto mismo se propone en *A Survey on Reinforcement Learning of Vision-Language-Action Models for Robotic Manipulation* [27], así como una evaluación del problema Sim2Real y la exploración segura.

Este último problema mencionado, el Sim2Real, es algo constante en todas estas disciplinas. Hace referencia a la implementación de las políticas en robots reales. Sin embargo, este problema se manifiesta en la parte final del aprendizaje por refuerzo y por ende, se abordará al final de este, en el capítulo 8.

2.5. Conclusiones del estado del arte

Como se puede observar, el RL es una herramienta ampliamente utilizada en la robótica. El desarrollo del aprendizaje profundo, junto con su capacidad de manejar sistemas no lineales y espacios de estados-acción complejos han permitido la implementación del RL. Cabe resaltar cómo el RL se aplica a disciplinas de diferentes necesidades, adaptándose a la alta dimensionalidad de la manipulación o el estudio dinámico de la locomoción; así como la coordinación multi-agente de los enjambres, la interacción social de la navegación o la combinación de informaciones en el VLA. Todos estos factores lo hacen óptimo

para los distintos temas que se tratan en este trabajo. No obstante, antes de comenzar con ejercicios de aprendizaje, se profundiza extensamente en el marco teórico del RL.

Capítulo 3

Fundamentos teóricos del aprendizaje por refuerzo

3.1. El aprendizaje por refuerzo dentro del aprendizaje automático

El aprendizaje por refuerzo pertenece a una disciplina más grande: el aprendizaje automático. Esta disciplina agrupa todos los ejercicios en los que una máquina aprende acerca de un entorno. Se dice que un programa aprende, si mediante una experiencia asociada a una tarea y una medida de éxito, su rendimiento en dicha tarea mejora en función de la medida seleccionada [28, Pág. 1].

Esta disciplina tiene tres grandes ramas: el aprendizaje supervisado, el aprendizaje no supervisado y el aprendizaje por refuerzo.

El aprendizaje supervisado se encarga de agrupar pares de entradas y salidas de información, mediante ejemplos catalogados [29, Pág. 137]. Estos ejemplos constan de pares de entrada y salida conocidos, los cuales sirven para clasificar futuras entradas. Un problema de aprendizaje supervisado puede ser identificar tipos de animales mediante una base de datos previa. En este caso, se alimenta al modelo con imágenes de animales (entrada) y el nombre (salida). El modelo deberá crear relaciones entre ambos. Se evalúa finalmente al programa por su habilidad para asociar imágenes de animales a su nombre.

El aprendizaje no supervisado, por otro lado, utiliza directamente las entradas sin una salida asociada [30, Pág. 740]. Esto hace que el programa deba buscar patrones lógicos inherentes a la clasificación. Estos patrones se basan en características a estudiar [29, Pág. 142]. Un problema de aprendizaje no supervisado podría ser agrupar imágenes de animales en función de su especie. En este caso, se alimenta al programa solo con las imágenes. Siguiendo únicamente la composición de los animales mostrados, deberá agruparse.

La frontera entre ambas disciplinas puede resultar difusa. No existe una diferencia formal entre ambas, pues la distinción entre una característica a estudiar y una salida asociada no es absoluta [29, Pág. 142]. El aprendizaje por refuerzo se diferencia de ambas a través de una única señal de realimentación (acorde a la definición presentada en el apartado 2.1.). De este modo, esta disciplina combina la supervisión del aprendizaje supervisado con la ventaja de no requerir una gran base de datos catalogada. Gracias a esto, se puede realizar un aprendizaje secuencial sin disponer de un modelo del entorno, lo que lo hace especialmente útil para el estudio de la robótica.

3.2. Estructura del aprendizaje por refuerzo

El aprendizaje por refuerzo está definido por una estructura básica. Esta estructura viene de la formalización del problema como un Proceso de Decisión Markov (MDP) [6, Pág. 47]. Esto proviene de la propia naturaleza del problema, por lo que está ligada al aprendizaje por refuerzo. En el próximo apartado, se estudiarán a fondo los MDPs. Sin embargo, al ser la estructura la base de esta disciplina, se presenta primero.

La estructura del aprendizaje por refuerzo define las interacciones entre un agente y un entorno. El agente ejerce acciones sobre el entorno, influyendo en él activamente. El entorno aporta al agente observaciones y recompensas, obteniendo así información sobre él. Además, el entorno tiene asociado un estado. 3.1

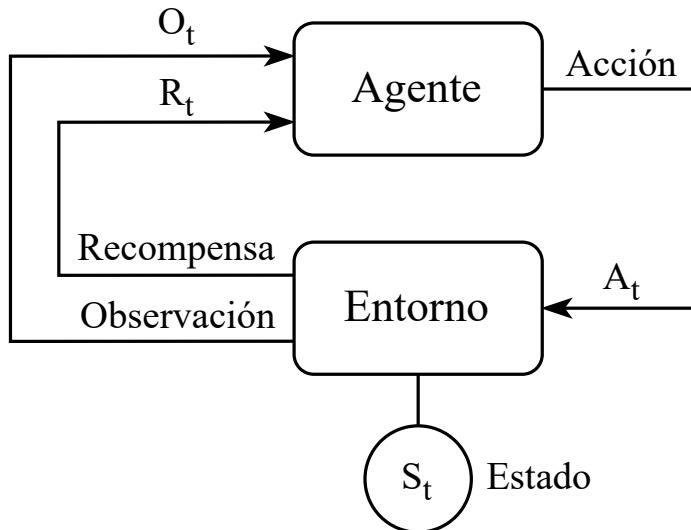


Figura 3.1: Diagrama de interacción agente-entorno.

Un ejemplo de esta estructura, fuera del aprendizaje por refuerzo, estaría en los estudios de condicionamiento operante de Skinner [31]. Estos estudios fueron muy influyentes en los inicios del aprendizaje por refuerzo [6, Pág. 16]. Para poder estudiarlo, se simplificará el ejercicio de estudio. Se supone un ratón en una caja; dentro se colocan dos botones. Uno de ellos emite una descarga eléctrica al animal, mientras que el otro le proporciona un estímulo positivo. Este ejemplo, aunque conceptual, ayuda a comprender mejor esta estructura.

El agente es el sujeto que aprende de la experiencia [6, Pág. 48]. Es responsable de las decisiones tomadas en el ejercicio, es decir, realiza todas las acciones definidas sobre el entorno. En el ejemplo propuesto, el agente sería el ratón. Por otro lado, el entorno comprende todo aquello que no es el agente [6, Pág. 48]. En nuestro ejemplo, comprendería el resto de elementos de estudio, la caja, los botones, etc., así como cualquier otro estímulo externo (los investigadores, el laboratorio). Lo anterior es importante para comprender la diferencia entre entorno, estado y observaciones.

El estado es la representación del entorno [6, Pág. 47]. Describe todos los aspectos relevantes del entorno. Las observaciones, por otro lado, representan toda la información que recibe el agente del entorno [5]. En casos donde el entorno es completamente observable, ambos pueden coincidir. Sin embargo, muchas veces los entornos no son completamente observables, por lo que las observaciones no comprenden todo el entorno; o algunas veces elegimos no observar parte del estado. En el caso de Skinner, el estado y las observaciones coinciden, siendo únicamente la posición de los botones (a la derecha o a la izquierda). En la robótica, la mayoría de las veces tendremos espacios parcialmente observables [32].

La recompensa es una señal numérica única que el agente recibe del entorno [6, Pág. 6]. Esta señal define el objetivo de la tarea sobre la cual el agente aprende. En el ejemplo propuesto, la recompensa sería negativa al recibir una descarga eléctrica y positiva al recibir el estímulo positivo. Cabe resaltar que, en el aprendizaje por refuerzo, la recompensa siempre es numérica, a diferencia del estudio de Skinner.

Por último, las acciones son todos los efectos producidos por decisiones del agente que dirigen al entorno a su siguiente estado [6, Pág. 48]. En el ejemplo del estudio de Skinner, las acciones serían pulsar el botón derecho y pulsar el botón izquierdo.

3.3. Proceso de decisión Markov o MDP

Un proceso de decisión Markov o MDP es una formalización de un proceso de toma de decisiones secuencial. En estos procesos, las acciones influyen tanto en la recompensa inmediata como en la transición de estados. Estos estados tienen a su vez futuras recom-

pensas asociadas, de ahí la realimentación retrasada [6, Pág. 47]. Al definir de manera ideal la toma de decisiones, los MDP sirven para formular el problema de aprendizaje por refuerzo de manera idealizada. Esta ideal permite considerar el entorno como completamente observable, incluso cuando se deriva desde uno parcialmente observable [5, Lección 2].

Otro punto clave de los MDP es la propiedad Markov. Esta propiedad se da cuando el estado actual contiene todos los aspectos que determinan el siguiente estado [6, Pág. 49]. Esto permite ignorar los estados pasados, ya que el siguiente estado depende enteramente del estado actual.

Existen otros tipos de procesos que mantienen esta propiedad y describen transiciones de estados, como las cadenas de Markov o los procesos de recompensa Markov [5]. Estos se pueden considerar como casos particulares de MDP, por lo que no se estudiarán en este trabajo. Sin embargo, si se prefiere comprender gradualmente los conceptos de estados y sus transiciones, es recomendable abordarlos.

3.3.1. Formulación de los MDP

Los MDP tienen asociada una formulación matemática. Para facilitar el estudio de esta, se muestra un diagrama en la figura 3.2. Este diagrama representa parte de un MDP de estados discretos y define la transición de un estado a otro. Un MDP completo conecta varios de estos estados y transiciones, hasta formar cadenas complejas. Esto se puede ver en la figura 3.3, en el siguiente apartado.

En este diagrama, se indican los principales elementos de los MDP:

- S, S' : **Los estados.** A cada paso de tiempo t , se recibe un estado S_t [6, Pág. 48]. En este caso, S_t será S y S_{t+1} será S'.
- π : **La política.** La política define la forma de comportarse del agente [6, Pág. 6]. Esta política es la encargada de seleccionar las acciones. En la figura 3.2, selecciona una de las dos acciones posibles.
- a : **Las acciones.** Las acciones son los efectos intencionados del agente sobre el entorno [6, Pág. 48]. Como se ve en el diagrama, la política selecciona una acción y, de esta acción, se transiciona al siguiente estado.
- p : **La probabilidad.** Una vez tomada una acción, existe una probabilidad de caer en un estado u otro [6, Pág. 48].
- r : **Las recompensas.** Al transicionar a un nuevo estado, se recibe una recompensa numérica [6, Pág. 48].

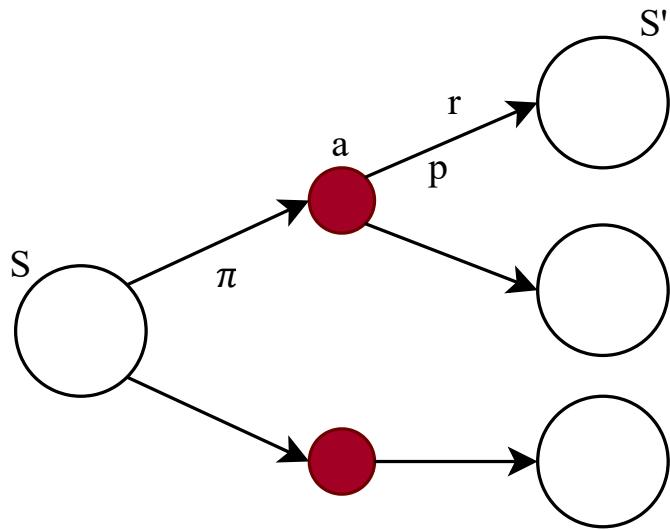


Figura 3.2: Estructura base y elementos de unos MDP.

Es interesante notar que la recompensa no va asociada a un estado. La recompensa se entrega al transicionar de un estado a otro. Es decir, la recompensa entregada puede no ser la misma al entrar en un estado S' desde un estado S , que desde un estado S'' .

Referente a las probabilidades, cabe definir correctamente cómo funcionan estas. Estas definen la dinámica del MDP [6, Pág. 48], por lo que son claves para el desarrollo del aprendizaje. Para ello, se va a utilizar la formula postulada en el libro de Sutton y Barto [6, Pág. 48], sobre el cual se han desarrollado las definiciones de este apartado. Se define la probabilidad como una función determinista de 4 argumentos: $S \times R \times S \times A \rightarrow [0, 1]$. Esta matriz nos permite obtener en función del estado anterior y la acción tomada, el estado actual y la recompensa recibida.

$$p(s', r | s, a) = \Pr(S_{t+1} = s', R_{t+1} = r | S_t = s, A_t = a) \quad (3.1)$$

Esta relación es importante para calcular la función de valor, concepto que veremos en el apartado 3.3.4. Primero, sin embargo, se estudiará un ejemplo más complejo de un MDP.

3.3.2. Ejemplo de un MDP

En este apartado, se presenta un ejemplo de MDP (figura 3.3) y se comenta sobre sus puntos más interesantes.

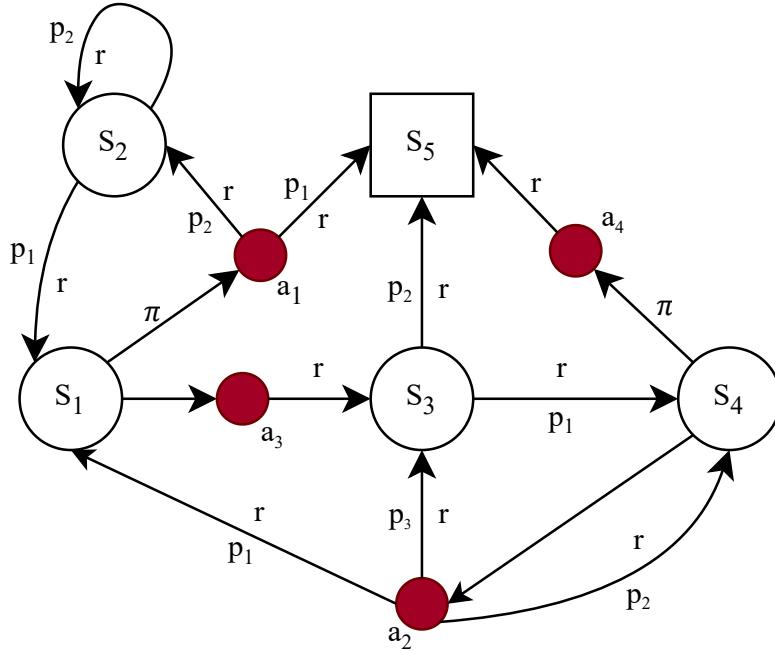


Figura 3.3: Ejemplo de un MDP completo

Este diagrama representa un MDP completo. Sobre este ejemplo se pueden observar la naturaleza de las transiciones entre estados. Cabe notar que el estado S_5 simboliza el final del proceso; de ahí que venga representado con un cuadrado [5]. Un estado termina un proceso cuando transiciona únicamente hacia si mismo, generando recompensas igual a 0. A este estado se le conoce como *estado absorbente* [6, Pág. 57].

Para ilustrar mejor aspectos relevantes de los MDP, se estudiarán distintos estados de este diagrama.

En la figura 3.4, se ven los estados a los que se puede transicionar. Al existir una probabilidad de mantenerse en el mismo estado, esto puede derivar en bucles. Por ello, es importante considerar que, aunque se termine en el mismo estado, puede haber una recompensa en dicho instante. Esto puede utilizarse para penalizar o recompensar la movilidad del sistema.

En el estado tres (figura 3.5), podemos ver un ejemplo de falta de acciones. Si se considerara como un proceso individual, se debería definir como un proceso de recompensa

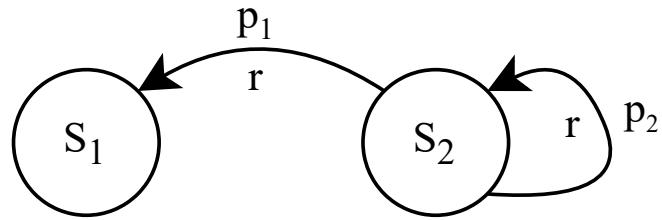


Figura 3.4: Subdiagrama del estado 2.

Markov. Por otro lado, si no tuviésemos en cuenta las recompensas, se podría definir como una cadena de Markov [5].

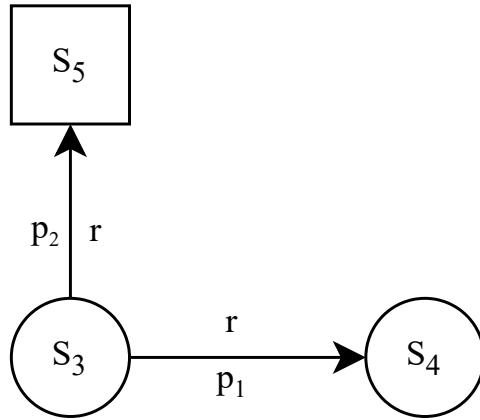


Figura 3.5: Subdiagrama del estado 3.

Por último, en el estado 4 (figura 3.6), vemos un ejemplo de la estructura definida, integrado dentro de un MDP. La política debe elegir entre dos acciones, a_2 y a_4 . Si se toma la acción 4, se entra directamente en el estado final. Si se toma, por otra parte, la acción a_2 , se encuentran distintas transiciones asociadas a probabilidades. En este caso, se puede observar la naturaleza de las probabilidades descritas. La probabilidad de transicionar al estado 3, y obtener su recompensa, desde el estado S_4 tomando la acción a_2 es p_3 . Cabe resaltar que la suma de todas las probabilidades asociadas al par estado-acción S_4 y a_2 debe ser 1, siguiendo la siguiente ecuación 3.2 [6, Pág. 48]:

$$\sum_{s \in S} \sum_{r \in R} p(s', r | s, a) = 1, \text{ para todo } s \in S, a \in A \quad (3.2)$$

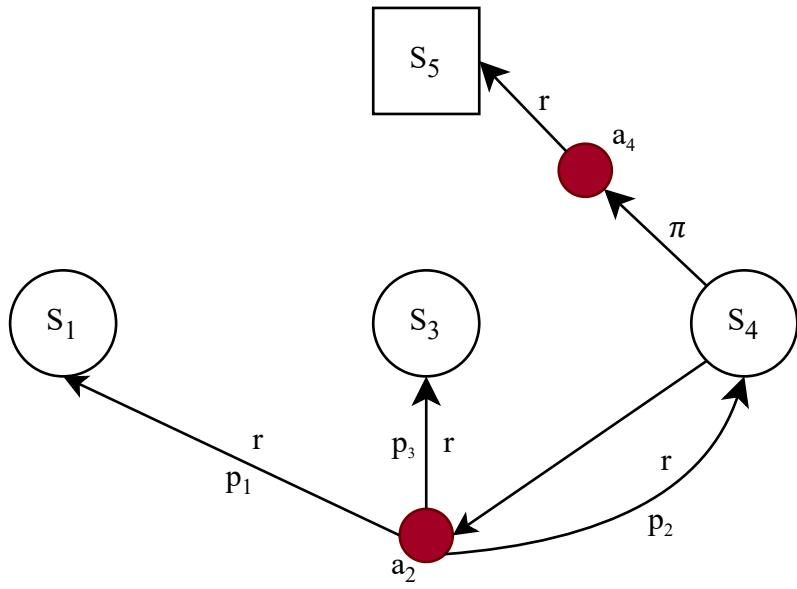


Figura 3.6: Subdiagrama del estado 4.

3.3.3. Funciones de Valor y Ecuación de Bellman

Las funciones de valor son una parte elemental del aprendizaje por refuerzo. Permiten analizar la recompensa esperada (retorno) en un estado, siguiendo una política concreta.

Antes de poder definir formalmente las funciones de valor, se deben definir dos conceptos básicos. Por un lado, el *retorno*, que se asocia, en su caso más simple, a la suma de una secuencia de recompensas:

$$G_t = R_{t+1} + R_{t+2} + \dots + R_T \quad (3.3)$$

donde T simboliza el final del episodio estudiado [6, Pág. 54]. Sobre este concepto, se define también el *retorno descontado*. Este concepto incluye un *factor de descuento*, γ , el cual permite graduar la importancia que se le da a las recompensas futuras sobre la actual. El *retorno descontado* se define mediante la siguiente ecuación:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \quad (3.4)$$

donde el *factor de retorno*, γ es un valor entre 0 y 1 [6, Pág. 54]. Es importante notar que

se puede reorganizar la ecuación como:

$$G_t = R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3} + \dots) = R_{t+1} + G_{t+1} \quad (3.5)$$

asociando así el valor de retorno con el propio del siguiente estado t . Esta asociación será importante para las funciones de valor, como veremos más adelante.

Para ejemplificar este valor de retorno, se va a estudiar un caso de la figura 3.3. Para ello, se toma una secuencia de estados S_1, S_3, S_4 y S_5 . A su vez se da valor a las recompensas, como se puede ver en la figura 3.7. Para dicha secuencia, donde $S_t = S_1$, se obtiene el siguiente valor de retorno:

$$G_t = r_{t+1} + r_{t+2} + r_{t+3} = -1 + 2 + 10 = 11 \quad (3.6)$$

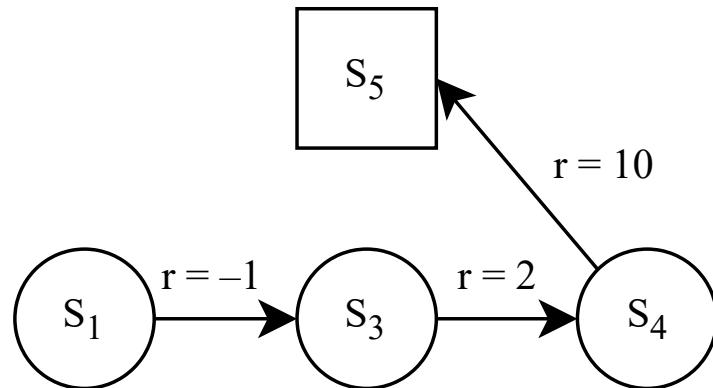


Figura 3.7: Ejemplo para el cálculo del valor de retorno

Otro concepto importante a tener en cuenta antes de estudiar es el de la *política*. Este punto ya se definió en el apartado 3.2, pero se incluye ahora una definición formal mediante la siguiente ecuación:

$$\pi(a | s) = Pr\{A_t = a | S_t = s\} \quad (3.7)$$

es decir, es la probabilidad de tomar una acción A_t dependiendo del estado actual S_t [6, Pág. 58]. Sobre esta política existe una *política óptima*, π^* . Esta *política óptima* es el objetivo final del aprendizaje. Después de definir las funciones de valor, definiremos formalmente este concepto.

Una vez definidos el *retorno*, el *factor de descuento* y la *política*, se pasa a definir las funciones de valor. Las funciones de valor describen el retorno esperado en un instante siguiendo una política concreta. Existen dos tipos de funciones de valor:

- 1. Función de valor estado:** La función de valor de un estado s es el retorno esperado desde s siguiendo una política π [6, Pág. 58]:

$$v_\pi(s) = \mathbb{E}[G_t | S_t = s], \text{ para todo } s \in S \quad (3.8)$$

- 2. Función de valor estado-acción:** La función de valor de un par estado-acción $((s, a))$, es el retorno esperado tomado la acción a desde el estado s [6, Pág. 58]:

$$q_\pi(s, a) = \mathbb{E}[G_t | S_t = s, A_t = a] \quad (3.9)$$

Cómo antes se definió en la ecuación 3.5, existe una continuidad de estas relaciones sobre el retorno y las recompensas [6, Pág. 59]. Es decir, si conocemos el retorno esperado del siguiente estado (o par estado-acción) y conocemos la recompensa inmediata, podemos obtener la función valor del estado actual. A esta relación se le llama *Ecuación de Bellman*, la cual será la base de un gran número de algoritmos, como se verá en el apartado 3.4. La *Ecuación de Bellman* se obtiene de modo que, para función valor estado (teniendo en cuenta la figura 3.2) [6, Pág. 59]:

$$\begin{aligned} v_\pi(s) &= \mathbb{E}[G_t | S_t = s] = \\ &= \sum_a \pi(a | s) \sum_{s'} \sum_r p(s', r | s, a) [r + \gamma \mathbb{E}[G_{t+1} | S_{t+1}]] = \text{ por 3.5} \\ &= \sum_a \pi(a | s) \sum_{s'} \sum_r p(s', r | s, a) [r + \gamma v_\pi(s')], \text{ para todo } s \in S, \end{aligned} \quad (3.10)$$

o de manera simplificada [5]:

$$v_\pi = R^\pi + \gamma P^\pi v_\pi$$

y para función valor estado-acción (según la figura 3.8):

$$\begin{aligned} q_\pi(s, a) &= \mathbb{E}[G_t | S_t = s, A_t = a] = \\ &= \sum_{s', r} p(s', r | s, a) (r + \gamma \sum_{a'} \pi(a' | s') \mathbb{E}[G_{t+1} | S_t = s', A_t = a']) \text{ por 3.5} \\ &= \sum_{s', r} p(s', r | s, a) [r + \gamma \sum_{a'} \pi(a' | s') q_\pi(s', a')] \end{aligned} \quad (3.11)$$

Ahora, se va a estudiar un ejemplo de aplicación sobre la figura 3.9. En este ejemplo se retoma un ejemplo anterior (figura 3.6), relativo al análisis del MDP completo (figura 3.3). En este ejemplo, se supone que se conoce la función de valor estado de los estados S_5 , S_3 y S_1 . Se quiere conocer x , que sería la función de valor estado de S_4 , $v_\pi(S_4)$. Se supone que la política π tiene una probabilidad del 80 % de elegir la acción a_4 , y que las

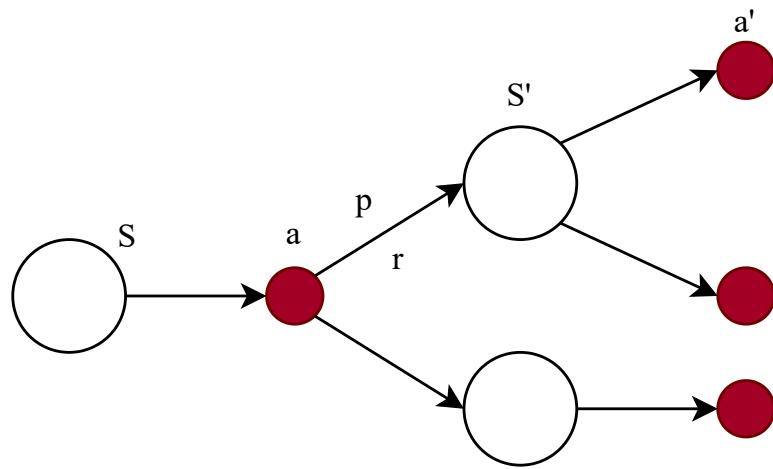


Figura 3.8: Estructura de un MDP enfocada al par estado-acción

probabilidades p_1 , p_2 y p_3 son, respectivamente, 0.5, 0.3 y 0.2; y que el *factor de descuento*, γ , es 0.6.

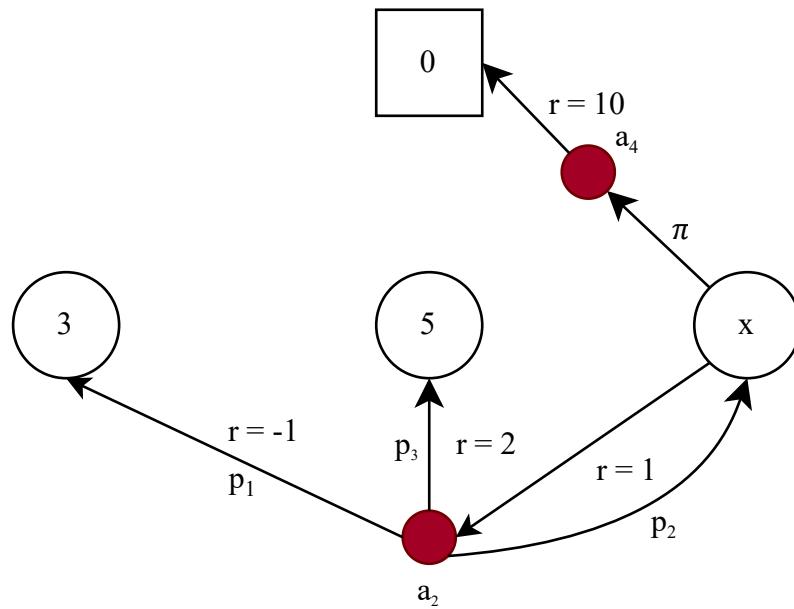


Figura 3.9: Ejemplo para el cálculo de la función de valor.

Para fragmentar la resolución de este problema, tendremos en cuenta ambas ecuaciones de Bellman, la de estado 3.10 y la de estado-acción 3.11:

$$q_\pi(s, a) = \sum_{s', r} p(s', r | s, a)[r + \gamma v_\pi(s')] \text{ por 3.10 y 3.11} \quad (3.12)$$

$$v_\pi(s) = \sum_a \pi(a | s) q_\pi(s, a) \text{ por 3.10 y 3.9} \quad (3.13)$$

Con estas ecuaciones definidas se puede proceder con el problema. Primero se calculan las funciones de valor de estado-acción:

$$\begin{aligned} q_\pi(S_4, a_4) &= r + \gamma v_\pi(S_5) = 10 + 0,6 * 0 = \\ &= 10 \\ q_\pi(S_4, a_2) &= p_1 * (r + \gamma v_\pi(S_1)) + p_2 * (r + \gamma v_\pi(S_4)) + p_3 * (r + \gamma v_\pi(S_5)) = \\ &= 0,5(-1 + 0,6 * 3) + 0,3(1 + 0,6 * x) + 0,2 * (2 + 0,6 * 5) = \\ &= 1,7 + 0,18x \end{aligned}$$

Una vez se tiene las funciones de valor estado-acción, se puede calcular la función de valor estado mediante la ecuación 3.8:

$$\begin{aligned} v_\pi(S_4) &= \pi * q(S_4, a_4) + (1 - \pi) * q(S_4, a_2) = 0,8 * 10 + 0,2 * (1,7 + 0,18 * v_\pi(S_4)) \\ v_\pi(S_4) &= \frac{2085}{241} \approx 8,7 \end{aligned}$$

Como se puede observar, cuando se conocen las dinámicas del MDP (las probabilidades) y el resto de funciones de valor, es fácil obtener la función valor. Sin embargo, si se quiere conocer la función de valor de cada estado y par estado-acción, se debe calcular para cada caso. En la mayoría de casos, no se dispone de las dinámicas del MDP o, aun conociéndolas, es demasiado complicado derivar de ellas los valores [6, Pág. 65-66]. Para esto, dentro del aprendizaje por refuerzo se han desarrollado algoritmos capaces de resolver estos problemas. Se estudiarán en la sección 3.4. Antes, se debe comprender por qué es importante conocer estos valores, lo cual se verá en el próximo apartado.

3.3.4. Política óptima y Valor óptimo

Resolver un problema de aprendizaje por refuerzo es encontrar una política que obtenga una gran cantidad de recompensa en el tiempo [6, Pág. 62]. Para poder entonces escoger la mejor política se debe tener un criterio. Para ello, se van a utilizar tres conceptos, desarrollados en el libro de Sutton y Barto [6]:

1. Una política es mejor o igual que otra si y solo si las funciones de valor de estado para dicha política son iguales o mayores para todos los estados:

$$\pi \geq \pi' \Leftrightarrow v_\pi(s) \geq v_{\pi'}(s), \text{ para todo } s \in S \quad (3.14)$$

2. Existe al menos una política mejor que el resto, la *política óptima*. A pesar de que puede haber más de una, se denotan a todas como π_* .
3. Todas las políticas óptimas comparten la misma función valor estado, la llamada *función de valor estado óptima*, así como la misma función de valor estado-acción, la *función de valor estado-acción óptima*:

$$v_*(s) = \max_{\pi} v_{\pi}(s), \text{ para todo } s \in S \quad (3.15)$$

$$q_*(s, a) = \max_{\pi} q_{\pi}(s, a), \text{ para todo } s \in S, a \in A \quad (3.16)$$

Por esto es tan importante conocer las funciones valor. Nos permiten por un lado, diferenciar si una política es mejor que otra; o en otras palabras, si una decisión es mejor que otra. Además, como se verá en la siguiente sección 3.4, son la clave para obtener o aproximarnos a la política óptima.

3.4. Algoritmos clásicos del aprendizaje por refuerzo

Existen múltiples algoritmos para resolver el aprendizaje por refuerzo. En esta sección se estudiarán algunos de ellos. A lo largo del desarrollo de la disciplina, estos algoritmos han aumentado progresivamente su complejidad con el objetivo de abordar problemas más amplios. Los primeros algoritmos se centran en MDP discretos e ideales, por lo que su aplicación práctica es limitada y, en el contexto de este trabajo, inexistente. No obstante, su estudio resulta fundamental para comprender los principios teóricos que rigen el aprendizaje de los agentes. La base conceptual de estos algoritmos es común.

Los algoritmos usados en este trabajo son proporcionados por bibliotecas, por lo que no se trabajan directamente. El enfoque práctico de este trabajo está en la elaboración de la estructura sobre la cual aprende el algoritmo. Esto incluye la construcción de entornos, la obtención de observaciones, el cálculo de las recompensas y la gestión de acciones. A pesar de esto, se van a exponer aspectos relevantes en esta sección. En el apartado 3.5, se estudiará cómo se aplican los algoritmos a MDP continuos mediante el uso de aproximaciones y pesos; así como sus objetivos, ventajas y desventajas en el apartado 3.6.

3.4.1. Programación Dinámica

La programación dinámica o DP se refiere al conjunto de algoritmos usados para obtener la política óptima de un modelo MDP perfecto. La clave de estos algoritmos es el uso de las funciones de valor para organizar y estructurar la búsqueda de buenas políticas. La política óptima se obtiene a partir de la función de valor estado óptima (ecuación 3.15) o la función de valor acción-estado óptima (ecuación 3.16). A su vez, estas satisfacen la ecuación de Bellman [6, Pág. 73]:

$$v_*(s) = \max_a \sum_{s',r'} p(s',r | s,a) [r + \gamma v_*(s')] \text{ por 3.10 y 3.15} \quad (3.17)$$

$$q_*(s,a) = \sum_{s',r'} p(s',r | s,a) [r + \gamma \max_{a'} q_*(s',a')] \text{ por 3.11 y 3.16} \quad (3.18)$$

para todo $s \in S$, y $a \in A(s)$, y $s' \in S^+$.

Los algoritmos de DP utilizan las ecuaciones de Bellman con una serie de reglas de actualización, buscando simplemente el valor máximo de recompensa, es decir, de funciones de valor [6, Pág. 74]. Cabe resaltar que para poder realizar el cálculo de la política óptima se deben conocer las dinámicas del sistema, es decir, se conoce $p(s',r | s,a)$ para todo $s \in S$, $a \in A(s)$, $r \in R$ y $s' \in S^+$ [6, Pág. 74].

Los algoritmos de DP se construyen con distintos procesos a realizar [6, Pág. 70-79]:

- Evaluación de la política. Se calculan nuevos valores para las funciones de valor, $v_{k+1}(s)$, en función de la política escogida y los valores actuales de la función de valor, v_k , usando la ecuación de Bellman:

$$v_{k+1} = \sum_a \pi_k(a | s) \sum_{s',r} p(s',r | s,a) [r + \gamma v_k(s')] \text{ para todo } s \in S \text{ por 3.10} \quad (3.19)$$

- Mejora de política. En este paso se escoge una nueva política, valorando las distintas acciones mediante la función de valor estado-acción. Al tener los valores de la función valor estado, se puede utilizar la ecuación 3.12. Para confeccionar la política, al buscar la obtención de la mayor cantidad de recompensa, se postula el término de 3.20, la cual responde a:

$$\pi'(a | s) = \arg \max_a q_\pi(s,a), \text{ para todo } s \in S \quad (3.20)$$

A parte, existen varias maneras de iterar estos procesos para obtener la política óptima [6, Pág. 80-87]:

- Iteración de política. En este procedimiento se evalúa una política, para después escoger una política mejor mediante 3.20. De esta forma, obtenemos la siguiente

secuencia:

$$\pi_0 \rightarrow v_0 \rightarrow \pi_1 \rightarrow v_1 \rightarrow \dots \rightarrow \pi_* \rightarrow v_* \quad (3.21)$$

- Iteración de valor. En este procedimiento se busca obtener directamente la función de valor óptima. En este caso, se introduce un paso distinto a los anteriores. Se lleva la selección codiciosa junto con la evaluación de la política, de modo que:

$$v_{k+1}(s) = \max_a \sum_{s',r|s,a} p(s',r | s, a)[r + \gamma v_k(s')] \text{ para todo } s \in S \text{ por 3.17 y 3.19} \quad (3.22)$$

Una vez obtenida la política óptima (cuando $v_{k+1} - v_k = 0$), se configura la política en función de 3.20.

- DP asíncrona. Estos algoritmos, en vez de realizar análisis completos, trabajan actualizando estados concretos, sin necesidad de mantener una estructura concreta. Estos métodos siguen requiriendo la misma necesidad de computación, pero aceleran el desarrollo de políticas. Se centran en estados clave y actualizan la política en función de estos.

En estos algoritmos, los dos procesos estudiados se encadenan en dos fases: una fase de evaluación de la política (cálculo de funciones valor) y otra de mejora de la política. En la iteración de política, se alterna continuamente una fase de evaluación con otra de mejora. Por otro lado, en iteración de valor, se mantiene una fase de evaluación, donde una vez obtenida la función de valor óptima se aplica una fase de mejora.

Existe un cuarto tipo de algoritmos que permiten las interacciones entre ambas fases, los basados en la *iteración de política generalizada* o GPI. Estas fases compiten y cooperan entre sí, alejando a la otra de su objetivo a la vez que convergen hacia un mismo punto. Se observa que cada fase tiene un objetivo: en evaluación se busca obtener la función de valor óptima y en mejora la política óptima. Esto se puede observar en el siguiente diagrama 3.10 [6, Pág. 87]:

DP es inefectivo para grandes problemas, pero son bastante eficientes en la resolución de MDP. Estos métodos son mejores que cualquier búsqueda directa, ya que aseguran la convergencia en la política óptima. Existen también métodos lineales, pero se vuelven ineficientes al escalar el número de entornos [6, Pág. 87].

Por otro lado, DP no es viable en el campo de la robótica. En primer lugar, siguen siendo demasiado ineficientes. Este método necesita un estudio completo de todos los estados, incluso en DP asíncronos. Es inviable cuando se trabaja un gran número de estados. Además, en robótica, los estados son continuos, por lo que habría que introducir

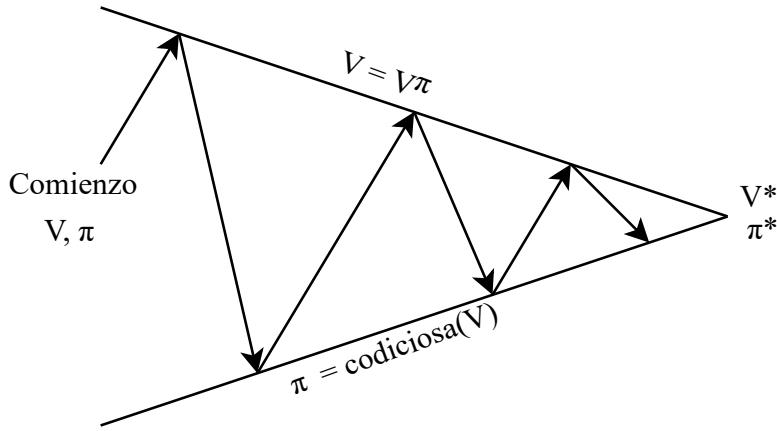


Figura 3.10: Diagrama de fases en DP [6, Pág. 87]

una aproximación. Por último, pese a la posibilidad de calcular su modelo dinámico, esto no resulta práctico debido a su complejidad y no linealidad. Por esto, se vuelve más interesante utilizar otros algoritmos que no necesitan este modelo dinámico. En los siguientes apartados, se analizarán algunos de estos algoritmos.

3.4.2. Método Monte Carlo

El primero de estos algoritmos que se estudiará es el método de Monte Carlo o MC. Este método se formalizó en 1949, en el artículo de Metropolis y Ulam [33]. En él se presenta un método para calcular mediante un enfoque estadístico y resolver problemas matemáticos complejos de abordar analíticamente. En este artículo se pone de ejemplo las interacciones entre neutrones y átomos en una reacción nuclear. Este método hace uso la ley de números grandes y los teoremas fundamentales de la teoría de la probabilidad para converger a un resultado aproximado al real.

En el caso de aprendizaje por refuerzo, este método permite realizar el aprendizaje sin tener un contexto previo del entorno, aprendiendo únicamente de la experiencia y no de un estudio previo del modelo [6, Pág. 91]. Esto resuelve los principales problemas de DP, la necesidad de conocer el modelo completo del entorno. Esto se hace estimando la media de los retornos obtenidos, dividiendo la experiencia en episodios. Estos episodios deberán terminar en un retorno, independientemente de las acciones tomadas [6, Pág. 91]. En otras palabras, deberán terminar en un estado absorbente (concepto introducido en el apartado 3.3.2).

Como en DP y, como se verá en el resto de algoritmos, el ejercicio de MC se divide en dos partes: predicción y control. La predicción podría asociarse a la evaluación de la política, salvo que en MC se trata de una estimación; por esto se usa el término

predicción. El control se asociaría a la mejora de la política. A continuación, se describen la implementación de la predicción y del control como problemas separados.

Predicción

La predicción en Monte Carlo busca estimar las funciones de valor para una política determinada [6, Pág. 92]. Teniendo en cuenta que las funciones de valor son el retorno esperado de un estado o un par estado acción (ecuación 3.8), aplicando el método de Monte Carlo a esta tarea, simplemente se debe realizar la media de los retornos obtenidos para calcular las funciones de valor. Se puede actualizar entonces la función de valor estado de modo que [5, Lección 4]:

$$\begin{aligned} N(s) &\leftarrow N(s) + 1 \\ S(s) &\leftarrow S(s) + G_t \\ V(s) &\leftarrow \frac{V(s)}{N(s)} \\ V(s) &\rightarrow v_\pi(s) \text{ cuando } N(s) \rightarrow \infty \end{aligned}$$

teniendo un registro de los retornos obtenidos ($S(s)$) y el número de veces que se ha entrado en el estado ($N(s)$); haciendo la media en cada uno de ellos ($V(s)$). Esta media converge al valor real de la función valor estado cuando el numero de retornos obtenidos, en un estado, tiende a infinito.

Utilizando este método de actualización existen dos principales algoritmos: MC de *primera visita* y MC de *todas las visitas* [6, Pág. 93]. En el algoritmo de primera visita, se actualizan los valores únicamente la primera vez que se alcanza dicho estado; mientras que en el de todas las visitas, se actualiza cada vez que se alcanza.

En Monte Carlo, siempre se debe tener en consideración la falta de un modelo del entorno; por ello, es más interesante estimar el valor de las acciones (función de valor estado-acción) que del estado (función de valor) [6, Pág. 96]. Al no tener este modelo, no se puede saber cuál será el siguiente estado, mientras que siempre se tiene conocimiento de las acciones a realizar, ya que dependen directamente del agente en estudio. Se debe adaptar entonces el método de Monte Carlo al valor de las acciones:

$$\begin{aligned} N(s, a) &\leftarrow N(s, a) + 1 \\ SA(s, a) &\leftarrow SA(s, a) + G_t \\ Q(s, a) &\leftarrow \frac{SA(s, a)}{N(s, a)} \\ Q(s, a) &\rightarrow q_\pi(s, a) \text{ cuando } N(s, a) \rightarrow \infty \end{aligned}$$

teniendo en cuenta un algoritmo de primera visita o de todas las visitas.

A pesar de la ventaja de valorar acciones en vez de estados, al centrarse en las acciones, se crea un problema. El objetivo de la predicción es determinar qué acciones son mejores; sin embargo, si se tiene una política determinista, algunos pares acción-estado nunca se darán [6, Pág. 96]. Este problema se tiene en cuenta siempre que se desconozca el modelo: una vez encontrada una buena política, se debe seguir al pie de la letra o variarla para continuar buscando nuevas políticas. Este problema se denomina *explotación contra exploración* [5]. En el siguiente apartado, se verán formas de mejorar las políticas, manteniendo en todo tiempo la exploración.

Control

El control en el método de Monte Carlo, así como en el resto de algoritmos, se centra en la aproximación de la política óptima. Para ello, se puede tomar como referencia los algoritmos GPI de DP. Se tendrán dos procesos distintos: uno en el que se estima el valor de las acciones y otro donde se mejorará la política en función de estas [6, Pág. 97].

En el caso de MC, como se viene repitiendo, no se conoce el modelo del entorno, por lo que para usar una política codiciosa (ecuación 3.20), de igual modo que en DP, se debe introducir el concepto de *inicios explorativos* [6, Pág. 92]. Para que este se dé, debe existir una probabilidad de empezar en cualquier de los pares estado-acción; de modo que todos se puedan valorar.

En los casos aplicados a la robótica que se estudiarán, no se da esta condición. En estos casos, al ser estados continuos es difícil que esta situación se de; existen infinitos estados. Además, es interesante tener un único punto de inicio de referencia. Por ello, se deben encontrar otras formas de gestionar la exploración.

Una de estas formas se denomina ϵ – *codiciosa*. Esta política regula el grado de exploración a través de un coeficiente ϵ . A través de este método, se puede escoger la política de la siguiente manera [6, Pág. 101]:

$$\pi(a | S_t) = \begin{cases} 1 - \epsilon + \frac{\epsilon}{N_a(S_t)} & \text{si } a = \arg \max_a q(a, S_t) \\ \frac{\epsilon}{N_a(S_t)} & \text{si } a \neq \arg \max_a q(a, S_t) \end{cases} \quad \text{para todo } S_t \in S \quad (3.23)$$

de modo que N_a sea el número de acciones y ϵ un número entre 0 y 1. De este modo, existe una posibilidad de tomar cada una de las acciones, regulando a su vez esta probabilidad mediante el factor ϵ . El valor de este último parámetro variará en función del objetivo del aprendizaje. Este objetivo, a su vez, cambiará, dependiendo de la forma en la que se trabaje sobre la política, lo cual se analiza en el apartado 3.4.4.

Problema del método Monte Carlo

El principal problema con el método Monte Carlo es la toma de valores y su actualización. Monte Carlo (figura 3.11) necesita esperar hasta el final del episodio para obtener el retorno y ajustar los valores función. Esto alarga los procesos de computación, dependiendo a su vez de la duración de los episodios. Sin embargo, se tiene a disposición una

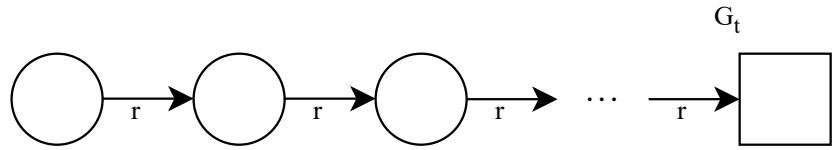


Figura 3.11: Figura de obtención del retorno en MC.

herramienta que permite actualizar inmediatamente: el retorno esperado, es decir, la función de valor. A continuación, se presenta un algoritmo que hace uso de esta herramienta para adelantar la actualización.

3.4.3. Temporal Difference o TD

Temporal Difference o TD es una de las ideas que ha revolucionado el aprendizaje por refuerzo. Mezcla dos ideas principales: aprendizaje de experiencia directa (aportado desde Monte Carlo) y utilización de los valores estimados (aportado desde DP) [6, Pág. 119]. Por un lado, no estudia todos los casos particularmente ni necesita información previa sobre ellos, sino que actualiza los estados en los que va entrando y recibiendo recompensa mediante ello. Por otro lado, esta actualización se hace utilizando el valor del siguiente estado (predicción) o acción (control). De este modo, los algoritmos TD se basan en la siguiente actualización [6, Pág. 120 y 129]:

$$V(S_t) \leftarrow V(S_t) + \alpha[R_{t+1} + V(S_{t+1}) - V(S_t)] \quad \text{Predicción} \quad (3.24)$$

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)] \quad \text{Control (SARSA)} \quad (3.25)$$

De este modo, se puede actualizar inmediatamente después de pasar el estado, trabajando siempre con la información más reciente.

Esta rápida actualización es una gran ventaja, pero en la gran mayoría de casos se deberá buscar un compromiso, de modo que la mayor parte de la capacidad computacional no se consuma en las actualizaciones. Para ello, existen los algoritmos *n-pasos TD* [6, Pág. 141]. Estos permiten regular en qué momento se actualizan las funciones valor. Un TD de *1-paso*, sería el algoritmo que se acaba de estudiar, donde se actualiza en el siguiente instante; es decir, siguiente estado. Por otro lado, un TD de infinitos pasos, *∞ -pasos*,

correspondería a un algoritmo Monte Carlo. Se puede configurar entonces la actualización de los estados en función de n , como se observa en la figura 3.12.

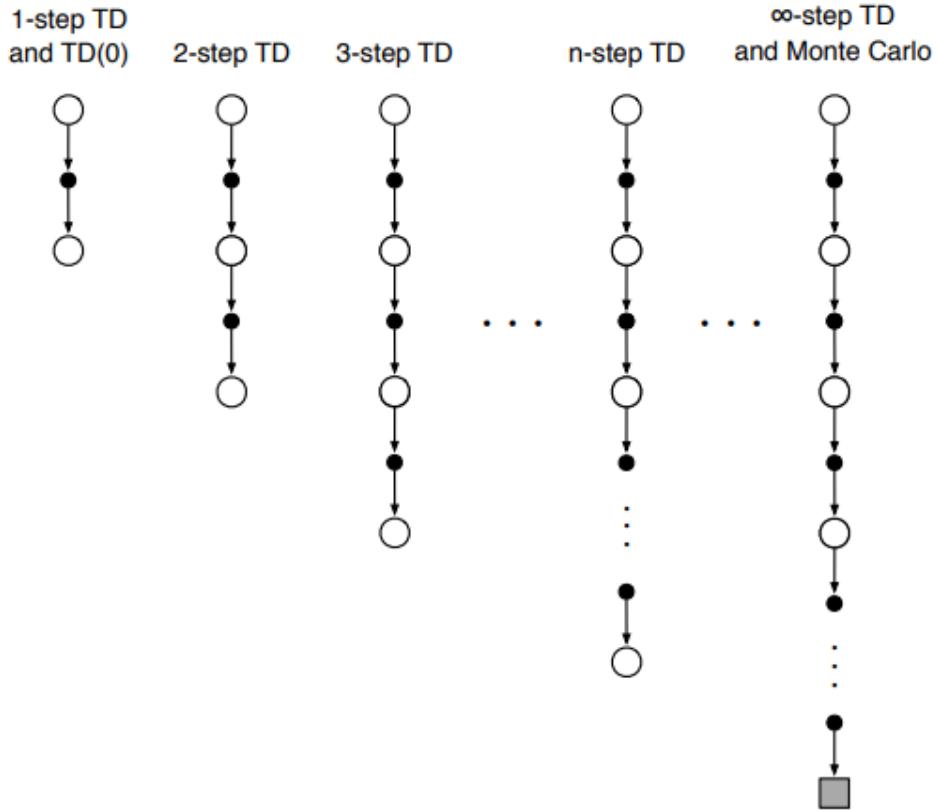


Figura 3.12: Resumen de los tipos de $TD\ n$ -pasos

El control de este método, por otro lado, mantiene el uso de los valores Q , tal como se ha indicado anteriormente. Sin embargo, como se adelantó en el apartado 3.4.2, no siempre se trabajará de la misma forma estos valores. Existen dos formas de trabajar los valores de la política, en *on-policy* y *off-policy*. A continuación, se presentan las diferencias entre ambas formas de trabajar.

3.4.4. On-Policy vs Off-Policy

En el primer caso, *On-Policy*, se aprende acerca de la política π a partir de la experiencia muestreada de π [5, Lección 5]. Es decir, la información obtenida (retornos y funciones de valor), proviene de la política la cual se aspira mejorar. Un ejemplo de este modo de trabajo es el método SARSA (ecuación 3.25). En él, se trabaja sobre una única política π , para la cual se estiman los valores de acción, $q_\pi(s, a)$, y se mejora en función de estos valores. En este caso, se ajusta de la política ϵ -codiciosa de modo que $\epsilon = 1/t$ [6, Pág. 129]. Así, se comenzará con una gran exploración, valorando así todas las acciones posibles; para tender con el tiempo a una política estable (poca exploración

$\epsilon \rightarrow 0$). El problema de esta forma de trabajo es que ϵ tiende a 0. Para que la política converja a la óptima se debe garantizar que todos los pares estado-acción se visiten, lo que puede no darse si ϵ decrece demasiado rápido.

En el segundo caso, *Off-Policy*, se aprende acerca de la política π a partir de la experiencia muestreada desde μ [5, Lección 5]. Es decir, se tienen dos políticas, una política que se quiere mejorar y una política de la cual se aprende. Esto permite mantener una exploración constante, ya que se aprende sobre una política base robusta; asegurando así la convergencia. Un ejemplo de esta forma de trabajo está en el algoritmo *Q-learning*, que actualiza tal que [6, Pág. 131]:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)] \quad (3.26)$$

Ambas formas de trabajar son útiles. *On-policy* mantiene una estabilidad y coherencia entre el aprendizaje y la acción, haciéndola más predecible y segura; pero su convergencia es lenta y esta no está asegurada. *Off-policy*, por otro lado, permite aprender una buena política rápidamente, manteniendo la exploración; sin embargo, sacrifica la estabilidad.

Con todos estos conceptos, se pueden realizar ejercicios completos de aprendizaje. Son, además, la base de los algoritmos modernos, los cuales se utilizan en los ejercicios aplicados a la robótica. Sin embargo, hay un punto que los limita: el número de estados. Como se viene indicando, cuando se trabaja con un gran número de estados (o son infinitos como en el caso de los estados continuos), se deben introducir nuevos conceptos, presentados en los siguientes apartados. Cabe resaltar que estos conceptos se alejan del enfoque de este proyecto, la construcción de entornos, pero se deben comprender y tener en cuenta.

3.5. Consideraciones para estados continuos

Todos los métodos que han sido vistos hasta el momento sirven para estados discretos. Concretamente, en los dos últimos, se observa una actualización de un valor asociado a un estado $v(S_t)$ que debe acercarse a los valores de retorno obtenidos, es decir, que se actualizan los estados en función del retorno $S_t \rightarrow G_t$ (en Monte Carlo) o el retorno esperado del siguiente estado y la recompensa $S_t \rightarrow R_{t+1} + \gamma v(S_{t+1})$ [6, Pág. 198]. Se debe actualizar por tanto uno a uno todos los estados, lo cual es inviable cuando el número de estados tiende a infinito.

Para resolver este problema, se introduce una tarea de aproximación de funciones. Esta tarea conlleva una gran complejidad y no se abordará en detalle. Sin embargo, se van

a introducir dos conceptos que tomarán relevancia en las futuras tareas de aprendizaje: la parametrización y la caracterización.

3.5.1. Parametrización de los estados, w

Para la aproximación de funciones, se introduce una función de parametrización, w . El objetivo cuando se introduce este nuevo concepto se puede entender las actualizaciones de manera que $s \rightarrow u$, donde para cada estado se busca aproximar su valor a un objetivo u [6, Pág. 198]. Se puede entender entonces las actualizaciones como un ejercicio de *aproximación de funciones*, donde se trata de minimizar el error cuadrático (ecuación 3.27) [6, Pág. 199].

$$\overline{VE} = \sum_{s \in S} \mu(s)[v_\pi - \hat{v}(s, w)]^2 \quad (3.27)$$

donde $\mu(s)$ pondera el valor del error para cada estado, siendo $0 \leq \mu \leq 1$.

Existen distintas maneras de configurar esta parametrización. Esta parte se aleja del enfoque del trabajo, ya que la construcción de estos algoritmos es un tema complejo. En nuestras tareas prácticas el enfoque será construir los entornos para utilizar algoritmos preestablecidos. Sin embargo, cabe destacar un concepto más de este tipo de actualización y aproximación: los vectores de caracterización.

3.5.2. Vector de caracterización de los estados, x

Uno de los casos de aproximación de funciones más importante se recogen en los métodos lineales, donde la función aproximada, $\hat{v}(s, w)$, es una función lineal del vector de parametrización, w . Para ello, a cada estado s , se le asocia un vector tal que $x(s) = (x_1(s), x_2(s), \dots, x_d(s))^T$, con el mismo número de componentes que w . De este modo, los estados continuos quedan definidos mediante un vector de caracterización previamente definido:

$$\hat{v}(s, w) = w^T x(s) \quad (3.28)$$

definiendo los estados continuos por un vector de caracterización previamente definido.

Estos dos conceptos definidos permiten, mediante gradientes, aplicar los algoritmos de aprendizaje clásico para ajustar ambos [6, Pág. 202 y 203]. Se define el valor de los estados mediante un vector de caracterización y una parametrización. Los algoritmos modernos, que se utilizarán en este trabajo, trabajan en la base de estos conceptos. Trabajan con métodos no lineales, haciendo uso de redes neuronales para aproximar la función de

valor. Sin embargo, hacen uso del concepto de vector de caracterización, cambiando la parametrización única por múltiples capas de transformaciones. A continuación, se estudiarán brevemente algunos de estos algoritmos, resaltando sus principales características y objetivos.

3.6. Algoritmos modernos (Aprendizaje profundo)

Los algoritmos modernos que se presentan a continuación forman parte de una subdisciplina del aprendizaje por refuerzo: el *aprendizaje profundo*. Esta disciplina se diferencia del aprendizaje por refuerzo clásico en su uso de redes neuronales artificiales (*Artificial Neural Networks* o ANNs) [6, Pág. 475].

Las ANNs están formadas por una cantidad variable de capas. Las ANNs son alimentadas por un vector, x , el cual se va transformando a través de las capas. Cada capa puede entender como una función. El vector x entra a una primera capa, $f^{(1)}(x)$ y el resultado de esta se alimenta a la segunda, $f^{(2)}(f^{(1)}(x))$ así hasta llegar a una última capa de salida, donde se obtiene el resultado final de la red [29, Pág. 164]. Por ejemplo, para la figura 3.13, se tienen 4 capas, de modo que la función de esta ANNs sería $f(x) = f^{(4)}(f^{(3)}(f^{(2)}(f^{(1)}(x))))$. Cabe resaltar que el único valor conocido desde fuera sería el de la última capa; no se conocen las salidas del resto de capas, por lo cual se las denomina como *capas ocultas*.

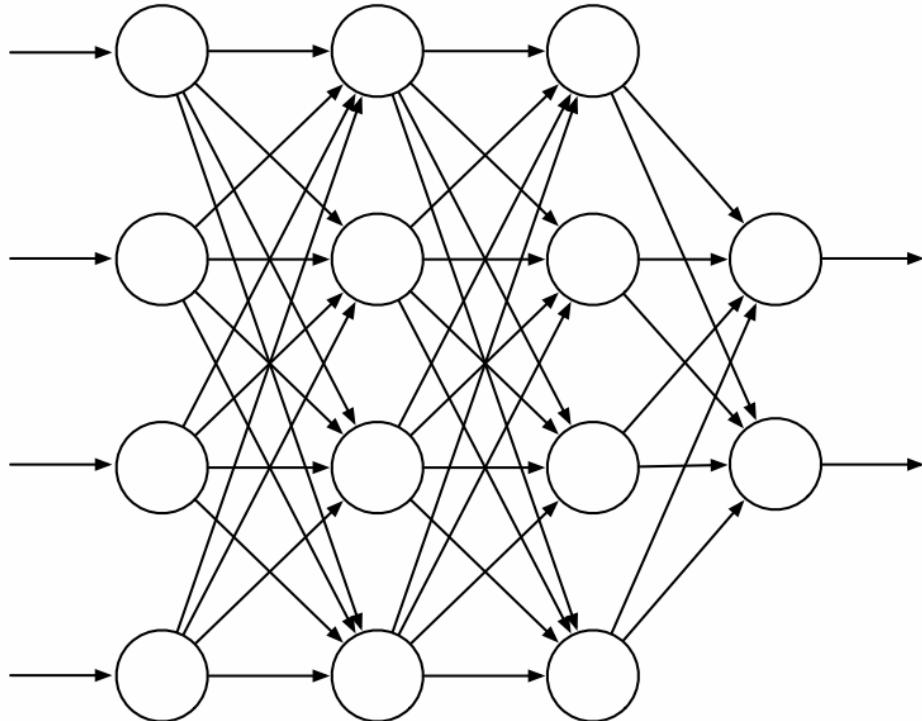


Figura 3.13: Ejemplo de una red neuronal artificial [6, Pág. 224]

Atendiendo a las consideraciones del apartado anterior, pese a la no linealidad de estos métodos, se pueden abstraer ambos conceptos a las ANN. Por un lado, se mantiene una parametrización, w , que en vez de ser lineal, consiste de una multitud de capas funcionales, $f^{(*)}$. Por otro lado, se puede asociar el vector de caracterización, $x(s)$, con el vector de entrada de la red neuronal, el cual también debe ser una representación del estado. Algunos de estos algoritmos son:

- *Deep Q-Networks* o **DQN**: Este algoritmo combina el método *Q-learning* (Off-policy) con el uso de redes neuronales [34]. Consta de uno de los primeros casos de éxito del aprendizaje por refuerzo, en el cual se aprendió a jugar al videojuego Atari a través del estudio de partidas [35].
- *Actor-Critic Algorithms*, **A2C**, **A3C**: Los algoritmos Actor-Crítico, están basados en dos procesos simultáneos: el actor, encargado de mejorar la política, y el crítico, que evalúa las acciones tomadas por el actor, estimando las funciones de valor [6, Pág. 331]. En la práctica moderna, tanto el actor como el crítico se implementan a través de redes neuronales (A3C) [36].
- *Proximal Policy Optimization* o **PPO**: El algoritmo PPO parte de la base de los algoritmos actor crítico, introduciendo el concepto de *función objeto sustituta recortada*. Esta función se encarga de limitar la capacidad de variación de la política, haciéndola más estable en su entrenamiento [37].
- *Soft Actor-Critic* o **SAC**: Este algoritmo busca maximizar tanto la recompensa como la entropía [38]. Es decir, mediante una estructura de actor crítico, busca aumentar la recompensa buscando a su vez continuar explorando distintas acciones.

Estos son solo algunos de los algoritmos utilizados en aprendizaje profundo. Como se verá en los ejemplos prácticos 5 y 6, las bibliotecas encargadas de implementar estos algoritmos usan una mezcla de estos para sacar el máximo rendimiento.

Este último apartado finaliza el desarrollo teórico del aprendizaje por refuerzo. En él, se ha construido una base sólida de conceptos clave para realizar los ejercicios prácticos. A continuación, se dedica un apartado a la conexión de todos estos conceptos dentro del campo de la robótica, facilitando así la comprensión de sus aplicaciones prácticas.

3.7. Observaciones para los ejercicios prácticos

Este trabajo, como se ha indicado anteriormente, está centrado en la aplicación del aprendizaje por refuerzo en la robótica. Para poder llevar estos conceptos teóricos

presentados a este campo, es importante analizar su papel y cómo se relacionan con este campo.

En primer lugar, el aprendizaje por refuerzo se basa en mejorar el rendimiento en una tarea a partir de la experiencia. Por tanto, es fundamental definir claramente la tarea a implementar. En el primer caso que se estudiará (capítulo 5), se busca enseñar a un robot araña a caminar; mientras que en el segundo caso (capítulo 6), dentro del proyecto MetaTool, se enseñará a un robot a empujar un cubo con una herramienta. En ambos casos, contar con una definición precisa de la tarea es esencial para configurar adecuadamente el entorno.

Una vez definida la tarea, es necesario visualizar la estructura general. Inicialmente, se podría asociar al agente con el propio robot; sin embargo, en los casos estudiados, esta afirmación resulta inexacta. El agente corresponde a la red neuronal utilizada para el aprendizaje, mientras que el robot forma parte del entorno. Esto se evidencia al observar que gran parte de las entradas al agente provienen del robot: posición y velocidad de las articulaciones, energía empleada, presión ejercida, entre otros datos. La red neuronal es, por tanto, la que decide las acciones que ejecutará el robot, constituyendo el verdadero objetivo del aprendizaje.

Los algoritmos, si bien no constituyen el enfoque principal del trabajo, deben considerarse al diseñar los entornos. Habiendo estudiado la base matemática, desde los procesos de Markov hasta los vectores de caracterización, se comprende la importancia de seleccionar adecuadamente las observaciones, recompensas y acciones. Las recompensas se traducen directamente en las funciones de valor, que rigen los algoritmos estudiados. Las observaciones, a su vez, definen el vector de caracterización, determinando cómo se identifican los estados y orientando la selección de acciones. Finalmente, las acciones constituyen el objetivo central del estudio; su correcta definición es clave para maximizar el desempeño del robot.

Además del análisis de los algoritmos, el concepto redes neuronales presentado también será importante, especialmente en la implementación del aprendizaje a la realidad (capítulo 8). La política final resultante se obtendrá como una red neuronal, por lo que se debe tener en mente su funcionamiento para implementar en el robot real.

Todos estos conocimientos teóricos serán utilizados en los próximos capítulos. Aunque algunos de ellos no se muestren directamente, el éxito y entendimiento de los ejercicios vendrá determinado por la comprensión de estos conceptos. Una vez entendidos estos, es hora de proceder a la construcción de entornos. Para ello, se utilizará la herramienta IsaacLab, la cual se analizará en detalle antes de comenzar con los ejercicios.

Capítulo 4

Análisis de la herramienta IsaacLab

En este capítulo se presenta la herramienta IsaacLab, haciendo uso de la documentación oficial de IsaacLab [39], el informe técnico [40] y la biblioteca de APIs [41]. El objetivo es obtener una visión global de cómo funciona la herramienta, con el fin de comprender y crear entornos. A lo largo de él se estudiarán los conceptos básicos, los cuales se abarcarán en los próximos capítulos con ejemplos prácticos.

4.1. ¿Qué es IsaacLab?

IsaacLab es una herramienta utilizada en el entrenamiento de robots en Python. Su principal objetivo es simplificar los procesos en este tipo de ejercicios [39]. IsaacLab está optimizado para su uso en GPUs, combinando la renderización de imágenes realistas con el motor de físicas *PhysX* para construir simulaciones realistas [40].

IsaacLab es una herramienta integrada dentro de IsaacSim. IsaacSim, a su vez, es una aplicación construida sobre NVIDIA Omniverse, la cual permite desarrollar, simular y probar robots controlados por IA en entornos virtuales [42]. IsaacLab está formada por un conjunto de herramientas utilizadas dentro del simulador IsaacSim. De este modo, pese a que se trabaja en su mayor parte con IsaacLab, se adquirirán en este capítulo algunos conceptos de IsaacSim.

Los principales incentivos para usar IsaacLab son [39]:

- Modularidad: capacidad de modificar y añadir nuevos entornos, robots y sensores. Estos elementos se utilizan dentro de clases generalizadas, limitando las modificaciones en la estructura general.
- Código abierto: mantenimiento de un código abierto y libre para la comunidad. Esto permite una completa libertad para modificar cualquier código y adaptarlo a

las necesidades del entorno.

- Gran cantidad de ejemplos y recursos: IsaacLab cuenta con un gran número de entornos, sensores y tareas preparadas para el entrenamiento. De este modo se parte de una base sobre la que construir las tareas personalizadas.

Por estos motivos, se ha escogido esta herramienta para realizar los entrenamientos. La principal desventaja de esta herramienta es la necesidad de utilizar un hardware específico, las tarjetas RTX de NVIDIA. Sin ellas no se pueden llevar a cabo los ejercicios de aprendizaje, ya que IsaacLab trabaja directamente sobre ellas. Esto permite maximizar el rendimiento de las tarjetas gráficas, pero limita el uso a la disposición de estos recursos. En este proyecto, se contó de estas tarjetas gracias a la cesión de un ordenador por parte del equipo MetaTool.

En resumen, IsaacLab es una herramienta para realizar ejercicios de aprendizaje por refuerzo en IsaacSim. Con esto en mente, se presenta su estructura externa.

4.2. Estructura de la herramienta

La herramienta IsaacLab se centra en la construcción de los sistemas, los cuales luego se someten al aprendizaje. Estos reciben las acciones del agente (la red neuronal), procesando las recompensas y observaciones correspondientes [39, Walkthrough, Environment Background Design]. Para la construcción de estos, IsaacLab utiliza la misma estructura que IsaacSim. Esta estructura define y gestiona los entornos. Se puede imaginar esta como una muñeca rusa, donde cada nivel contiene al resto (figura 4.1).

El primer nivel consiste en la aplicación. Esta administra los recursos del sistema y es la encargada de lanzar y destruir la simulación [39]. La aplicación se puede gestionar a partir de la API `isaaclab.app` [41]. Esta API se encarga de gestionar el lanzamiento de la aplicación, así como los distintos argumentos pertinentes a esta.

Esta aplicación contiene la simulación, la cual, como ya se ha mencionado, es creada y destruida por ella. Esta es la encargada de definir cómo funcionarán las físicas, el tiempo y la gravedad. Dicha simulación divide el ejercicio en múltiples instantes de tiempo, distribuyendo las tareas de cálculo en subprocesos [39]. La simulación, al igual que la aplicación, tiene su propia API, `isaaclab.sim` [41]. Con esta, se definen parámetros como el tamaño de paso o la fuerza de la gravedad.

La simulación contiene todos los elementos relevantes a esta, los cuales se agrupan en el concepto de *mundo* [39]. Este mundo se define por el origen de coordenadas. Este se toma como referencia para ubicar el resto de objetos. El mundo se estructura en dos componentes más: el escenario y la escena. El escenario, por un lado, proporciona un

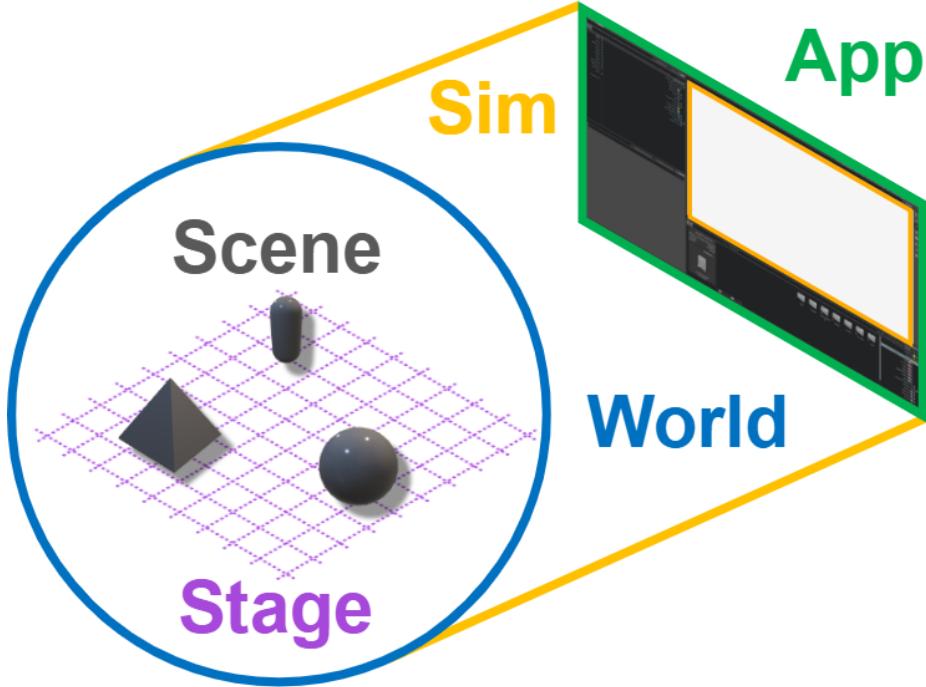


Figura 4.1: Estructura de la herramienta IsaacLab [39].

contexto geográfico dentro de la escena [39]. Es decir, permite utilizar dentro de las escenas un origen de coordenadas propio. La escena, por su parte, será la encargada de administrar los elementos que conforman el entorno.

El entorno como tal está organizado en un escenario (figura 4.2) y será administrado por la escena. Es por esto que la escena tiene su propia API, *isaaclab.scene* [41], la cual permite gestionar y obtener datos de todos los elementos del entorno. Estos se organizan en *primarios*, elementos separados dentro de la organización del escenario que son importados a la escena a través de un archivo USD (figura 4.2) [39]. USD, por su parte, es el lenguaje de descripción de robots y entornos [43]. En este trabajo no se maneja este lenguaje, pero sí se utilizan los ficheros de este tipo para importar los *primarios*.

IsaacLab, por lo tanto, está organizado en 5 conceptos fundamentales. Se asienta una referencia central, el mundo. Este contiene el escenario y la escena, los cuales organizan los entornos. Dicho mundo está contenido por la simulación, que define las propiedades de este. Por último, la simulación es gestionada por la aplicación. Esta jerarquía constituye la estructura externa a los entornos, sobre la cual se asientan. A continuación, se presentarán las distintas maneras de construir los entornos.

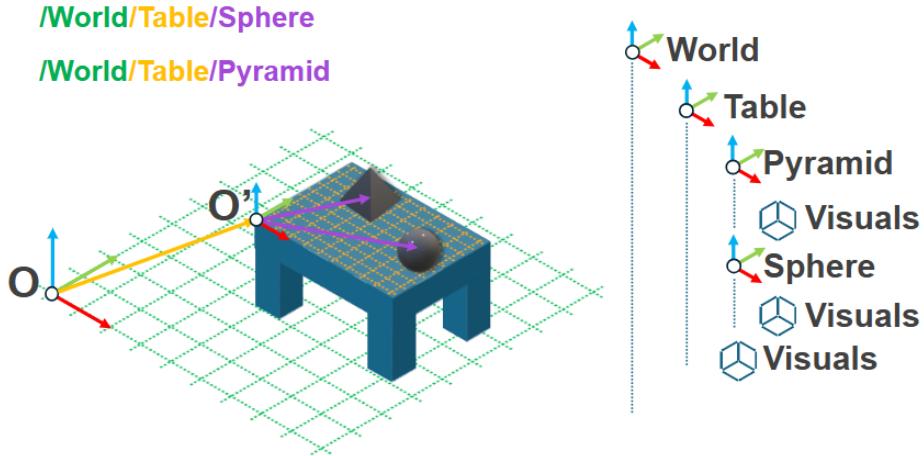


Figura 4.2: Organización de un escenario dentro de IsaacLab [39].

4.3. Arquitectura de entornos

Teniendo en cuenta la estructura anterior, se procede ahora a estudiar la construcción de entornos. Se recuerda que los entornos son los encargados de recoger las acciones del agente y procesar las observaciones y recompensas. Aparte de esta definición, se debe tener en cuenta el campo en el que se trabaja, la robótica. Por esto, el eje central de todos los entornos será el robot. Con todo esto en mente, se pueden definir los objetivos del diseño de entornos [39]:

1. Definir el robot y las acciones.
2. Definir los parámetros de la simulación.
3. Definir la forma de clonado y el número de entornos.
4. Procesar las acciones y calcular las recompensas
5. Definir los estados absorbentes y los reinicios.

Los entornos estarán constituidos por un robot y el resto de su entorno (objetos, obstáculos, efectos visuales, etc.). Sobre este entorno se definirán las acciones (asociadas directamente al robot), las recompensas y las observaciones. También se debe definir dentro de este entorno los estados absorbentes (finales). En algunas ocasiones el robot alcanzará un estado donde no será relevante continuar con el aprendizaje. En ese estado, se cerrará el episodio y se reiniciará el entorno. Por último, IsaacLab permite optimizar el tiempo de entrenamiento clonando los entornos. De este modo, en vez de tener un único entorno, se pueden tener múltiples entornos entrenando simultáneamente.

Para definir todos estos aspectos, existen dos principales maneras de programar los entornos:

4.3.1. Arquitectura directa

La manera directa, como su propio nombre indica, es la más franca de las dos. Esta forma permite implementar todos los puntos antes mencionados en un mismo *script*. Los entornos directos heredan de la clase `DirectRLEnv`, dentro de la API `isaaclab.envs`. Para programar el entorno, en este caso, se definen las funciones abstractas de esta clase. Seguidamente, la clase se encapsula en un *wrapper* y se alimenta a una de las bibliotecas con los algoritmos de aprendizaje por refuerzo. Este proceso se describe en detalle en el apartado 4.5. A su vez, un ejemplo de este tipo de construcción se presenta en el capítulo 5.

En la figura 4.3 se presenta un esquema de las interacciones de la clase `DirectRLEnv`, bajo el nombre *Environment Scripting*. Esta clase se encarga de comunicar el entorno (la escena) y el agente. Por tanto, volviendo a la estructura del aprendizaje por refuerzo (3.2), esta clase representa las interacciones entre ambos elementos.

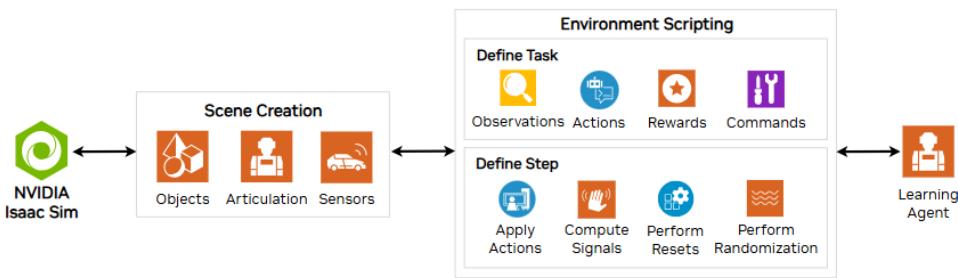


Figura 4.3: Diagrama para el modo directo de IsaacLab [39].

En resumen, la manera directa se centra en una única clase para la organización de las interacciones entre el entorno y el agente.

4.3.2. Manager Based

La manera basada en manejadores (*Manager-based*) descompone las distintas partes de las interacciones en componentes individuales, los manejadores [39]. Estos manejadores aislan distintos aspectos de la comunicación y gestión del agente y entorno, facilitando la organización de estos.

Entre los distintos manejadores están [39]:

- **Manejador de eventos (EventManager):** se encarga de definir y administrar aspectos del entorno. Esta clase es la encargada de ir cambiando parámetros de la escena durante los entrenamientos; de esta manera se pueden obtener entornos versátiles y variados durante un mismo entrenamiento.

- Manejador de observaciones (**ObservationManager**): se encarga de definir y gestionar las observaciones.
- Manejador de acciones (**ActionManager**): se encarga de definir y gestionar las acciones.
- Manejador de recompensas (**RewardManager**): se encarga de definir y gestionar las recompensas.
- Manejador de objetivos (**CommandManager**): se encarga de variar los objetivos del entrenamiento, siendo capaz de generar distintos valores en función de un rango.
- Manejador de finalización (**TerminationManager**): se encarga de definir las condiciones en las cuales se pone fin al episodio.

Estos son los principales manejadores que se utilizarán en los entornos. Existe un manejador más, el manejador del currículum (**CurriculumManager**), encargado de crear una hoja de ruta para el entrenamiento. En este trabajo, al trabajar con objetivos concretos, no es necesario, bastando el manejador de órdenes para darle flexibilidad al ejercicio. Todos estos manejadores se alojan en una clase global, **ManagerBasedRLEnv**, la cual se encarga de coordinarlos. Esta clase, al igual que la de modo directo, se encuentra dentro de la API *isaaclab.envs* [41]. Los manejadores se desarrollan como clases pertenecientes a esta clase global. Estas clases se encuentran dentro de la API *isaaclab.managers* [41].

Más adelante, al entrar a estudiar el proyecto MetaTool, se presentarán algunos ejemplos de esta forma de conformar entornos. El concepto más importante dentro de esta forma es el de término (*Term*). Estos manejadores se conforman de múltiples términos, los cuales descomponen aún más el ejercicio. Por ejemplo, para un manejador de observaciones con tres observaciones distintas (posición de las articulaciones, velocidad de las articulaciones y posición de un objeto) se tendrán tres términos distintos. De este modo, únicamente es necesario definir las observaciones individualmente en estos términos, el manejador después se encarga de entregársela al agente.

En el modo basado en manejadores, las interacciones entre el agente y el entorno, así como su administración, se fragmentan en múltiples manejadores, facilitando la definición y la gestión de los distintos elementos. Esto se muestra en la figura 4.4.

Comparando ambas maneras de programar, el modo basado en manejadores permite tener un *script* más organizado, facilitando las modificaciones. Gracias a su formato de clases, tiene una gran capacidad modular; sin embargo, esta forma pierde parte del control. El modo directo, pese a ser más complicado de visualizar y gestionar, permite mantener un control sobre la definición de los elementos y cómo se estructuran. Ambos modos de programar son útiles; por sus facilidades el modo de manejadores se priorizará.

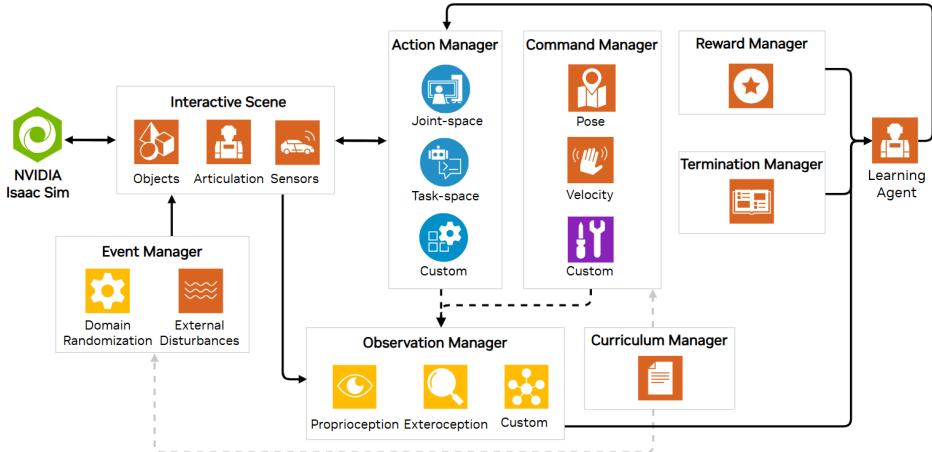


Figura 4.4: Diagrama para el modo basado en manejadores de IsaacLab [39].

En este trabajo, para tener primero una visión global de la construcción de entornos, se estudia a fondo un caso directo. Una vez comprendida esta forma de trabajo, se continuará en MetaTool con el modo de manejadores.

Ambas formas de programar hacen uso de dos conceptos fundamentales, las clases y los tensores. Las clases, tanto en el modo basado en manejadores como en el modo directo, son fundamentales; los entornos en ambas formas se definen a través de clases (`DirectRLEnv` y `ManagerBasedRLEnv`). Dentro de estas clases, se encuentra toda la información relevante del entorno, que en su mayoría se define como tensores *Torch*. A continuación, se profundiza en esta estructuración de la información.

4.4. Estructuras de datos

Pese a que este trabajo se centre en la construcción de entornos, eso no significa que se deba perder referencia de la base matemática y los algoritmos de entrenamiento. Se deben poder definir correctamente todos los datos necesarios para que los algoritmos puedan trabajar correctamente. Es por esto que se realizó un estudio exhaustivo de la teoría del aprendizaje por refuerzo clásico y por lo que ahora se estudiará con detalle cómo se almacenan la información del entorno, para así poder comprender y definir las observaciones, recompensas y acciones correctamente.

Como se ha mencionado en el apartado anterior, existen dos estructuras principales de datos, las clases y los tensores. Referente a esto, cabe hacer una aclaración. Dentro del lenguaje de programación Python hay múltiples estructuras de datos relevantes: variables, métodos, etiquetas, etc. Estos conceptos son propios de Python por lo que no son de interés en este trabajo. No obstante, dentro de IsaacLab, toman especial relevancia las clases (en especial las clases de configuración) y los tensores (trabajados con *PyTorch*). A

continuación, se presenta el motivo de esta relevancia y los aspectos más interesantes.

4.4.1. Clases

Las clases permiten agrupar datos y funcionalidades dentro de un objeto [44, Tutorials, 9. Classes]. Es por tanto un elemento clave dentro de la estructura de datos de los entornos. No solo los manejadores y los entornos globales se estructuran en clases, sino los propios robots, los primitivos o las escenas, entre otros, también se estructuran en clases. Como se analiza al completo en los ejemplos, se accede a todos los datos desde estas clases.

Las clases como tal son un elemento básico de la programación orientada a objetos, por lo que se presupone un conocimiento previo básico sobre estas. Sin embargo, las clases dentro de IsaacLab se definen de una manera estandarizada, a través de otras clases [39].

Las clases establecidas dentro de IsaacLab se definen a través de las clases de configuración. En estas clases se definen los parámetros correspondientes a la clase que se quiere definir. Por ejemplo, si se quiere instanciar una clase `InteractiveScene`, para crear una escena interactiva, se debe definir primero una clase `InteractiveSceneCfg`. Estas clases de configuración, las cuales se analizan en detalle en los ejemplos, se definen a través del decorador `configclass`. Un decorador es una función que, en este caso, toma una clase como argumento y devuelve una clase modificada [44, Glossary, decorator]. Este decorador prepara la clase para trabajarse como un argumento.

Dentro de las clases, se encuentran datos relevantes del entorno. Por ejemplo, en una clase `Articulation`, se encuentra una variable `data`, que es a su vez una clase `ArticulationData`, la cual contiene información relevante del robot, como la posición de las articulaciones, `Articulation.data.jointpos`. Esta variable final puede expresarse como un vector sencillo, con dimensión igual al número de articulaciones. No obstante, al trabajar con múltiples entornos clonados en paralelo, esto no es eficaz, pues entonces se debería tener una clase para cada entorno. Para compactar la información, IsaacLab usa tensores de *PyTorch*. En el siguiente apartado se estudiará qué es un tensor, cómo se utilizan y por qué son importantes.

4.4.2. Tensores: PyTorch

Un tensor es una matriz multidimensional de datos de un mismo tipo [45]. Un tensor tiene un número de dimensiones, llamadas ejes. Cada una de estas dimensiones tiene un tamaño, al que llamaremos tamaño del eje. Este tipo de dato permite organizar grandes cantidades de información en un mismo sitio. Por ejemplo, si se tienen 10 entornos, en cada uno de los cuales se tiene un robot con una cámara, donde dicha cámara tiene

una resolución de 800×400 , se podría definir un tensor que guardase los valores RGB de estas cámaras. Dicho tensor tendría la siguiente forma: [10, 800, 400, 3]. Este tensor tendría 4 ejes, con tamaños respectivos de 10, 800, 400 y 3. De este modo, en una matriz de 4 dimensiones, se almacena de forma compactada el valor de todos los bits de las 10 cámaras. Cada eje contiene a su eje inmediatamente inferior. El eje superior de este tensor corresponde al tamaño 10, siendo este el eje 0. Cada elemento de este eje, contiene el eje 1, con su tamaño de 800 elementos. Cada uno de esos elementos del eje 1, contiene un eje 2; y de igual manera con el eje 3 (o -1), siendo contenido por cada uno de los elementos del eje 2. Este tensor, por tanto, tendría 9.600.000 elementos. Cada uno de estos elementos, debe ser de un dato concreto.

IsaacLab hace uso de estos tensores, mediante la herramienta PyTorch, para poder almacenar toda la información de los entornos en una sola clase. PyTorch, por su parte, es una librería centrada en la optimización de operaciones con tensores a través de GPU [45]. IsaacLab utiliza esta librería dentro de sus librerías de algoritmos para el aprendizaje, por lo que todos los datos que se pretendan trabajar deberán manejarse mediante estos tensores. Esto incluye las observaciones, recompensas y acciones. Por esto, manejar correctamente los tensores es prioritario para la construcción de los entornos y sus interacciones con el agente.

Dentro de los ejemplos, estudiaremos detenidamente las distintas operaciones que se utilizan. No obstante, antes de analizarlos, se debe estudiar cómo se entrena finalmente los entornos, lo cual se verá en el próximo apartado.

4.5. Entrenamiento de agentes

Una vez configurados los entornos y las interacciones, se debe entrenar el agente en ellos. El enfoque de este trabajo se centra en esta primera parte, por lo que no se entrará en detalle en el propio aprendizaje y sus algoritmos. IsaacLab provee de una serie de librerías, las cuales aportan los algoritmos de aprendizaje. Existen cuatro librerías principales [39]:

- SKRL [46]: se enfoca en la modularidad, simplicidad y transparencia. Es especialmente útil si se pretende modificar los algoritmos para experimentar o investigar.
- RSL-RL [47]: se crea pensando en robótica, manipulación y locomoción, siendo la más específica de las cuatro. Esto lleva también a una corta documentación y alta complejidad.
- RL-Games [48]: esta biblioteca busca la optimización en entrenamientos intensivos y multi-agentes; especialmente útil para entornos complejos.

- Stable-Baselines [49]: de las 4 librerías, es la más documentada y con más comunidad y soporte, sin embargo, su presencia en ejemplos y sus capacidades son bastante limitadas.

Teniendo en cuenta las características de todas estas librerías, la que mejor se adapta a este trabajo es la librería RSL-RL. Al no modificar ni entrar a estudiar la composición de su algoritmo, no afecta en gran medida la falta de documentación. Por otro lado, su especialización en robótica la hace tener el mejor rendimiento entre todas las bibliotecas [39].

Para poder utilizar las bibliotecas, IsaacLab provee de un proceso para registrar y configurar agentes, *gymnasium* [50]. *Gymnasium* es una biblioteca de Python en código abierto. Proporciona una API para comunicar los algoritmos de aprendizaje con los entornos definidos. Para ello, se registra la configuración de los entornos a través de esta API, creando así una tarea con una configuración de agente asociada. De esta manera, al ejecutar los *scripts* de entrenamiento, `train.py`, se puede seleccionar la tarea con el argumento `-task` y realizar el entrenamiento.

El producto final del entrenamiento es la red neuronal parametrizada, la cual se guarda dentro del proyecto o la herramienta IsaacLab. Antes de poder utilizar esta red, se debe evaluar cómo se comporta; para lo cual tenemos distintos enfoques. En el próximo apartado se estudia cada uno de ellos.

4.6. Evaluación de agentes

Una vez obtenida la red neuronal, que corresponde al agente entrenado, el siguiente objetivo consiste en implementar esta red neuronal en el robot real. Este proceso conforma un problema en su conjunto llamado *sim2real* [51]. Este problema se verá en detalle en el capítulo 8, donde se integrarán alguna de las redes neuronales generadas en robots reales. Sin embargo, es prioritario para la seguridad del robot y de los operarios probar antes esta red neuronal en un entorno seguro. Por este motivo, se sigue el siguiente procedimiento:

- *Sim-in-Sim*: Primero, mediante las herramientas de IsaacLab, se probará la red neuronal en el mismo sistema donde se ha implementado el aprendizaje.
- *Sim2Sim*: Segundo, se realizará una simulación con herramientas externas para probar el comportamiento del robot en un entorno que tenga en cuenta las características reales de este.
- *Sim2Real*: Por último, se implementa la red neuronal en el robot real.

El primer paso es el único que se realice con la herramienta IsaacLab. Esta provee en su biblioteca de algoritmos de un script para poder probar el resultado del aprendizaje, *play.py*. Con este script se puede implementar la red neuronal sobre el mismo entorno y simulación entrenada. De este modo, se pueden estudiar detenidamente las distintas recompensas y a través de ellas valorar el trabajo del robot.

Para el resto de pasos, se utilizan otro tipo de herramientas y códigos, por lo que esta constituye la última utilización de la herramienta IsaacLab. Estudiando todas las utilidades de IsaacLab, se puede observar cómo abarca la gran parte del trabajo a realizar. Por este motivo, antes de entrar a los ejemplos prácticos, se debe abstraer una idea general de la herramienta.

4.7. Análisis Global

IsaacLab es una de las principales herramientas para el aprendizaje por refuerzo en la robótica. Su aplicación base, IsaacSim, está preparada para afrontar las dificultades que presenta la IA. Además, su utilización del potencial de las tarjetas GPU, las RTX de NVIDIA, pese a limitar el uso a este hardware, permite realizar ejercicios complejos de aprendizaje en poco tiempo.

La construcción de entornos dentro de esta herramienta, por su parte, requiere de un conocimiento extenso de las API y librerías asociadas. Sin embargo, las dos formas de programación permiten flexibilidad a la hora de desarrollar. Añadido a esto, una vez entendido el funcionamiento de la herramienta, construir y modificar entornos se vuelve más sencillo y sistemático. Gracias a esto, y a la gran cantidad de ejemplos, se encuentra que para la mayoría de ejercicios se puede trabajar sobre estos ejemplos, adaptándolos al entorno deseado.

En el próximo capítulo, se presenta un ejemplo de la programación directa. Se analizará cuidadosamente y se propondrán algunas mejoras.

Capítulo 5

Estudio caso locomoción

En este capítulo, se estudia un ejemplo de la herramienta IsaacLab. Con este estudio se pretende analizar las distintas partes de la construcción de entornos a través de la forma directa. Primero, se analizará el caso y el objetivo de este. Después, se realizará un diagrama de clases con las más relevantes, analizando sus métodos y atributos. Una vez definido el diagrama de clases, se analizará cada una detenidamente, entrando en detalle sobre sus métodos y atributos, se verá la función y definición de cada uno. A continuación, se estudiará el registro a través de *gymnasium*, repasando la configuración del agente. Una vez registrado el entorno, se procederá al entrenamiento de este y a la evaluación del resultado final. Por último, se propondrán algunas mejoras para futuros estudios de aprendizaje por refuerzo.

5.1. Descripción caso práctico

El primer ejemplo escogido para el estudio es el entorno *Isaac-Ant-v0*. En este entorno se busca enseñar a andar a un robot araña de cuatro patas, denominado en IsaacLab *Ant* (figura 5.1). El objetivo principal es desplazar el robot en una dirección concreta, manteniendo el torso paralelo al suelo. Se considerará que el robot se mantiene paralelo al suelo cuando su plano en los ejes x e y sea paralelo al plano xy del origen de coordenadas.

Analizar este ejercicio es una parte integral del presente trabajo. El objetivo a futuro es crear una guía para realizar futuros ensayos de aprendizaje por refuerzo. Este caso se relaciona directamente con dos proyectos internos de la escuela en la que se desarrolla este trabajo, *Romerín* [4] y *Tarántula* (en fase de desarrollo). Por tanto, este análisis tiene dos objetivos: analizar el problema concreto de locomoción para robots araña y estudiar un caso práctico de programación directa.

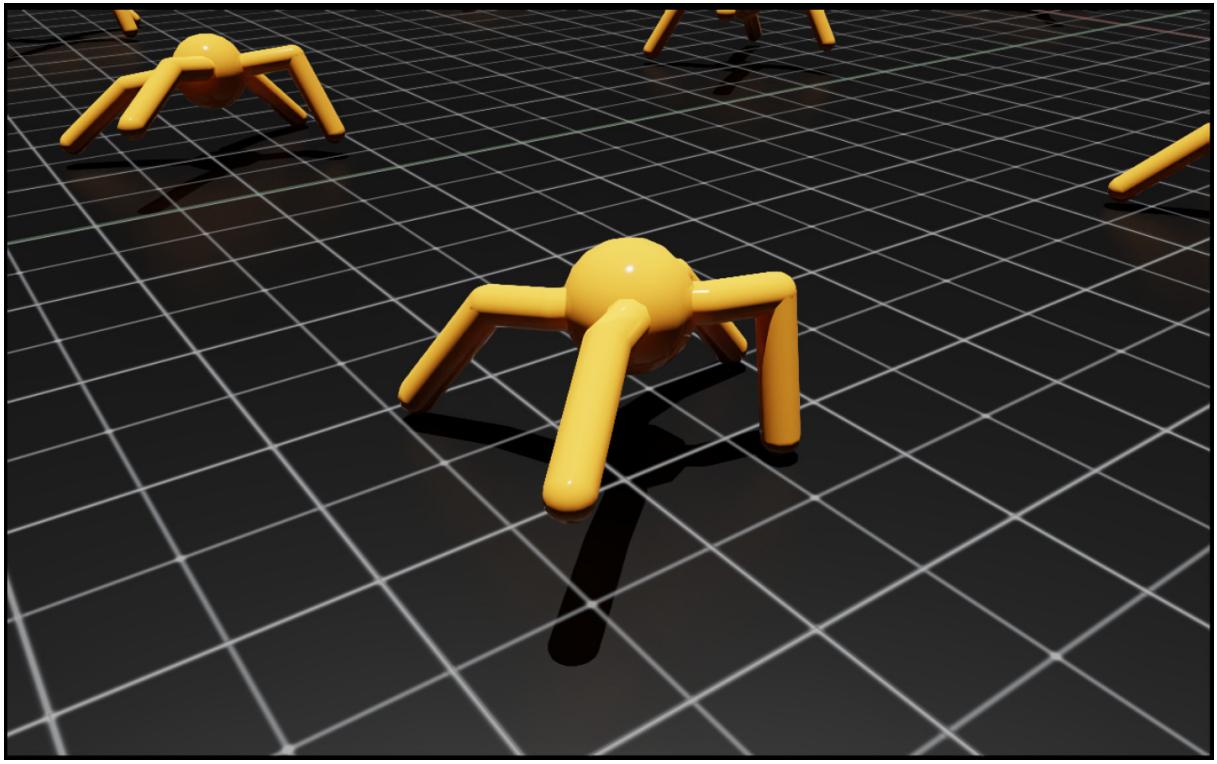


Figura 5.1: Robot araña o *Ant*, objetivo del aprendizaje para el primer caso práctico.

El código de este ejercicio se ha extraído de la herramienta IsaacLab; el código se puede encontrar dentro del repositorio de la herramienta [40], accesible desde la documentación [39]. En este capítulo, se analizará el código desde el diagrama de clases; aportando donde sea necesario los fragmentos de código relevantes. Durante el análisis, también se irá indicando dónde se encuentra la parte del código a la cual se hace referencia. Se ha preparado un proyecto de IsaacLab aislado de la herramienta, incluido en los anexos *ARMetaToolPG*, con todos los códigos utilizados. Se procederá ahora a la definición del diagrama de clases y su estudio.

5.2. Diagrama de Clases

El diagrama de clases del entorno de la araña se muestra en la figura 5.2. El diagrama se utiliza para obtener una visión general del código del entorno y para simplificar su análisis posterior. No se incluyen la totalidad de los métodos y atributos, pues gran parte de ellos no son relevantes para casos generales como los que se estudiarán. El resto de métodos y atributos se emplean únicamente en funcionalidades muy concretas o para procesos internos de IsaacLab.

El diagrama muestra la construcción del entorno, sobre el cual se entrena el entorno en el apartado 5.4 y 5.5. El entorno gira alrededor de dos piezas centrales, la clase

AntEnv y **AntEnvCfg**. Al estar trabajando en el caso directo, ambas clases heredan de sus contrapartes del modo directo: **DirectRLEnv** y **DirectRLEnvCfg**, respectivamente. Estas clases están definidas dentro del código de IsaacLab; cada vez que se construya un entorno en modo directo se hereda de ambas.

En el caso del directo, la clase de configuración, aquella que hereda de **DirectRLEnv Cfg**, se encarga de definir los parámetros físicos y de las interacciones del entorno, las características de la simulación y la escena (con el robot y el resto de elementos incluidos). Esta clase siempre será un atributo de la clase principal del entorno, aquella que hereda de **DirectRLEnv**. Esta segunda clase toma un gran protagonismo en el modo directo. Sobre ella cae la responsabilidad de definir cómo se implementa la configuración del entorno, gestionando las interacciones y parte del proceso de aprendizaje y creando la escena a partir de lo definido.

En este caso particular, se debe analizar de dónde provienen ambas clases. Por un lado, la clase de configuración **AntEnvCfg** hereda directamente de la clase de configuración base. Sin embargo, la clase principal del entorno de la araña hereda en un paso previo de una clase **LocomotionEnv**, que a su vez hereda de **DirectRLEnv**. Esta clase intermedia resulta especialmente útil, ya que generaliza una tarea concreta encargada de resolver el problema de locomoción. De esta manera, se puede heredar de esta clase para cualquier problema de este tipo, ajustando la escena al caso específico dentro de la configuración y ajustando parámetros concretos en la principal.

Este esquema se repite en la gran mayoría de los casos de programación directa. Por ello, es importante comprender cómo se implementa y define cada clase. En el próximo apartado, se estudiará detenidamente cada una de las clases.

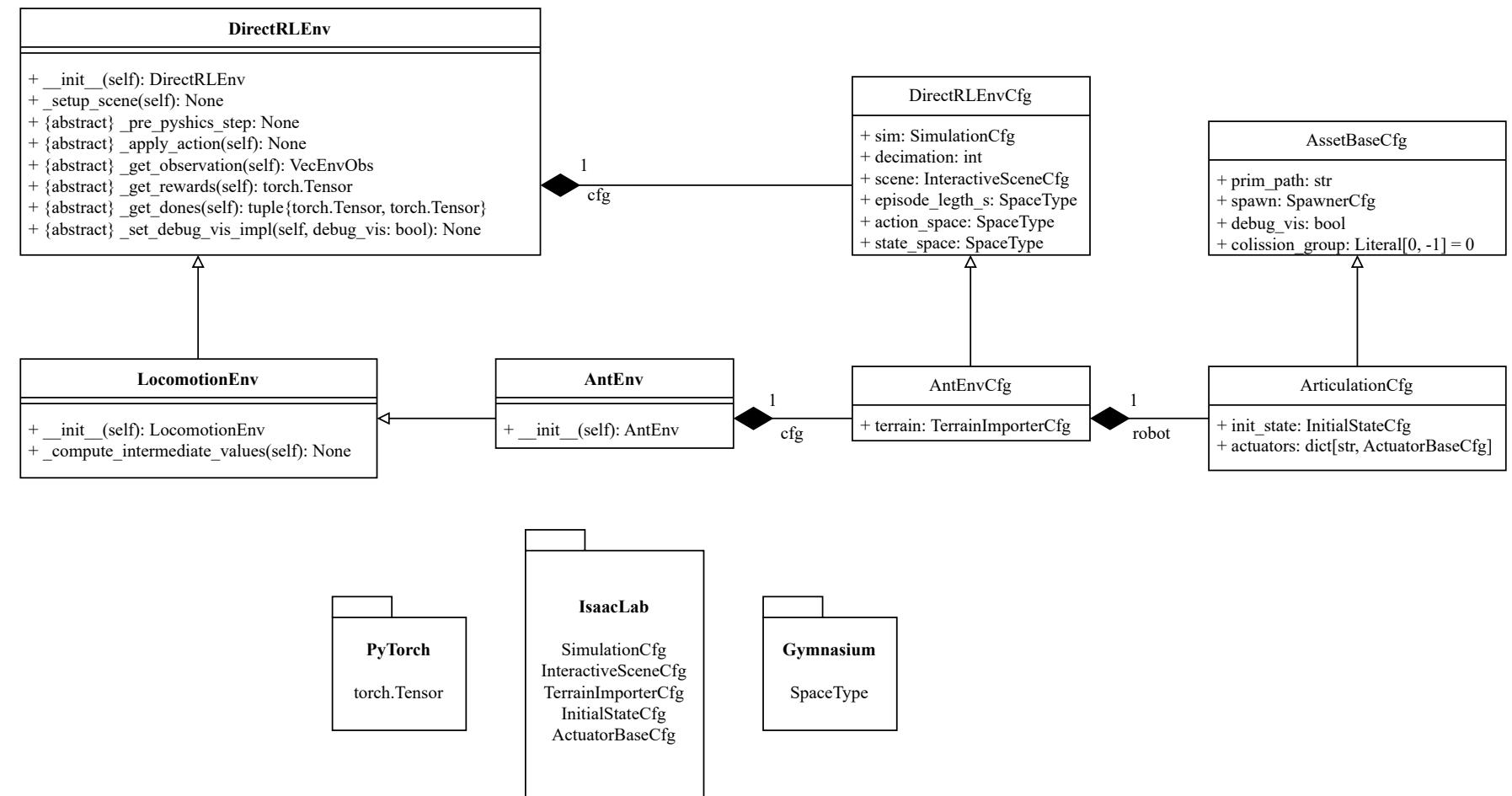


Figura 5.2: Diagrama UML del ejemplo araña, programación directa.

5.3. Análisis de clases

En este apartado se estudiará cada una de las clases mostradas en el diagrama, analizando los métodos y atributos en él definidos. Para cada una de las clases se indica dónde se puede encontrar el código. Después se explica la funcionalidad del método o atributo. Dentro de esta explicación, se muestra algunas partes del código donde exista un interés en la implementación del método, especialmente en aquellos que definan observaciones u recompensas del entorno. Se comienza el estudio en la clase principal padre, después a las distintas clases de configuración y por último a las clases heredadas.

5.3.1. DirectRLEnv

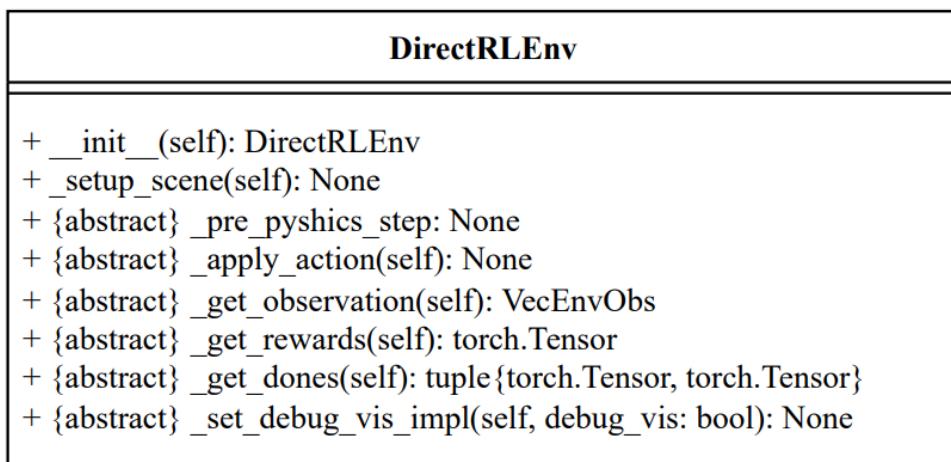


Figura 5.3: Imagen del diagrama referente a la clase DirectRLEnv.

La clase `DirectRLEnv` (figura 5.3.1) se encuentra definida en el código fuente de IsaacLab. Se puede acceder al código a través de una API de IsaacLab [41], concretamente en `isaaclab.envs.DirectRLEnv`. Una vez en dicha página, se debe seguir el enlace asociado al título, en el botón de "[source]", tal y como se indica en la figura 5.3.1.

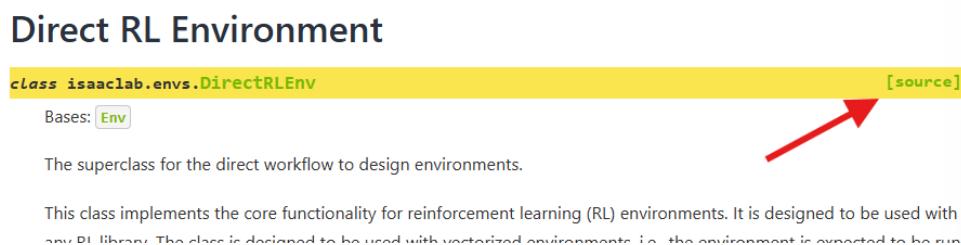


Figura 5.4: Imagen de la documentación oficial de IsaacLab con el link al código fuente [41].

Esta clase es el pilar fundamental del entorno. A través de sus métodos crea el entorno y define sus propiedades e interacciones. El primer elemento relevante de esta clase es del atributo definido como `cfg`. Este atributo almacena una clase `DirectRLEnvCfg`

y define la configuración del entorno que se pretende construir. Es por ello que se debe recoger en el constructor, que es el primer método definido en el diagrama. El constructor de esta clase es complejo y amplio, pero para el enfoque de este trabajo solo se tendrá en cuenta la recepción del atributo `cfg`. El resto del código va enfocado al propio funcionamiento interno de IsaacLab, el cual no se estudiará.

El resto de métodos no son definidos en esta clase, sino que son declarados. Exceptuando el método `_set_up_scene(self)`, el resto son métodos abstractos. Estos métodos se definen en las clases heredadas, con el objetivo de definir el funcionamiento de la clase. Más adelante, en el sub-apartado (figura 5.3.5), se verán ejemplos de sus implementaciones. En este apartado, se estudia únicamente el objetivo principal de cada una:

- `_set_up_scene(self)`: se encarga de configurar la escena, implementando los elementos definidos en la clase de configuración.
- `_pre_physics_step(self)`: define las acciones previas a realizar el cálculo de la física del entorno.
- `_apply_actions(self)`: procesa las acciones y se envían al robot.
- `_get_observations(self)`: se encarga de calcular y definir las observaciones realizadas sobre el entorno.
- `_get_rewards(self)`: calcula y define las recompensas obtenidas del entorno.
- `_get_dones(self)`: define y comprueba las condiciones de reinicio del entorno.
- `_set_debug_vis_impl(self)`: Se encarga de crear o configurar la visualización de los objetos en escena.

Esta clase define todas las funciones que deben utilizarse para crear y gestionar el entorno. Dentro de esta clase, existen otros métodos como `step(self)` o `render(self)`, que utilizan los métodos anteriores para crear el proceso de interacción con el entorno. Esta parte del código no es relevante para este trabajo, pues forma parte del funcionamiento propio de IsaacLab y no se debe modificar a la hora de crear los entornos. No obstante, a pesar de no ser parte del enfoque del trabajo, para tareas de depuración es necesario comprender este proceso.

Tal como se ha mencionado, el elemento que define gran parte de esta implementación sera la clase de configuración. A continuación, se estudia la clase base para posteriormente analizar las respectivas clases heredadas.

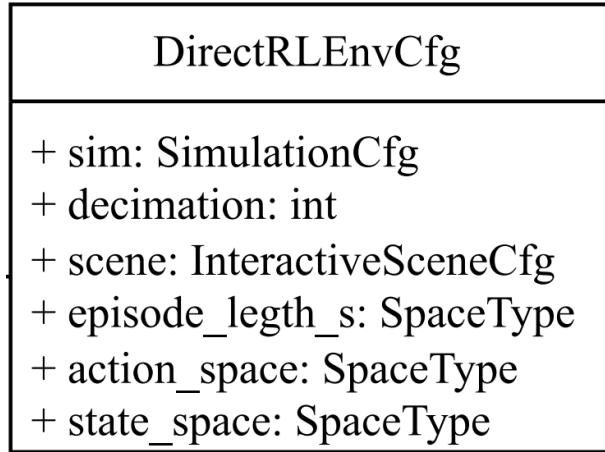


Figura 5.5: Imagen del diagrama referente a la clase DirectRLEnvCfg.

5.3.2. DirectRLEnvCfg

La clase DirectRLEnvCfg (figura 5.3.2), al igual que la anterior, se encuentra definida el código fuente; pudiéndose acceder de la misma manera desde la API `isaaclab.envs.DirectRLEnvCfg`. Esta clase está definida como una `configclass`. Este tipo de clase se introdujo en el apartado 4.4.1. Almacena únicamente atributos, lo que facilita su gestión dentro del funcionamiento de la herramienta. En este apartado, se enumeran y analizan los atributos más relevantes de esta clase y cómo afectan al entorno.

- **sim**: almacena una clase `SimulationCfg`, encargada de configurar los principales parámetros de la simulación.
- **decimation**: almacena un valor numérico entero (de tipo entero) que define el número de acciones realizadas antes de actualizar la política.
- **episode_length_s**: almacena un valor numérico decimal (de tipo real) que define la duración de un episodio.
- **scene**: almacena una clase `InteractiveSceneCfg` que define los elementos incluidos dentro de una escena, así como las propiedades de la misma escena.
- **obs_space**: almacena una clase `SpaceType` que indica el número de observaciones realizadas sobre el entorno.
- **action_space**: almacena una clase `SpaceType` que indica el número de acciones.

Conviene destacar algunos aspectos acerca de estos atributos. Exceptuando el atributo `sim`, el resto tienen asociada una constante `MISSING`. Esta constante se asegura de que estos atributos sean definidos dentro de una clase heredada; es decir, todos los atributos deben ser definidos en una clase específica de configuración. En segundo lugar, es

interesante notar que existen dos variables para el número de acciones y observaciones pero no para las recompensas. Esto es debido a que la recompensa debe definirse como una señal numérica, tal y como dicta el aprendizaje por refuerzo. El tamaño de las observaciones y las acciones por su parte define la dimensión de la red neuronal.

Vista la clase base de la configuración del entorno, se estudia cómo se hereda de ella para comenzar a definir un entorno concreto.

5.3.3. AntEnvCfg

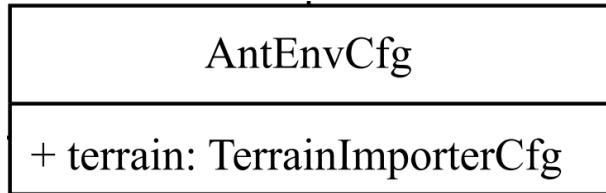


Figura 5.6: Imagen del diagrama referente a la clase `AntEnvCfg`.

La clase a estudiar es la clase `AntEnvCfg`. Esta clase está definida dentro del repositorio IsaacLab, en el directorio `source/isaaclab_tasks/isaaclab_tasks/direct/ant/ant_env.py` [40]. Esta clase hereda directamente de la clase `DirectRLEnvCfg`, incluyendo dos nuevos atributos: `terrain` y `robot`. El atributo `terrain` almacena una clase `TerrainImporterCfg`, encargada de configurar el terreno del entorno. Por otro lado, el atributo `robot` se encarga de definir las características del robot a entrenar, almacenando una clase `ArticulationCfg`; esta clase se implementa en el siguiente apartado.

En este caso, al ser una implementación de una clase, se estudia el código detenidamente. En él, primero se importan las distintas herramientas y bibliotecas que se utilizan. Entre ellas se pueden encontrar las clases de simulación, las clases de configuración, etc. Para poder importar una clase, un método o una constante, primero se debe localizar la API donde está definida y después indicarla. En el código 5.1, se puede ver un ejemplo, donde se importa la clase `TerrainImporterCfg` de la API `isaaclab.terrains`. También se pueden importar clases definidas en archivos aparte, como se hace con la constante `ANT_CFG` (código 5.2), que guarda la configuración del robot.

Listing 5.1: Ejemplo para importar una clase de una API

```
from isaaclab.terrain import TerrainImporterCfg
```

Listing 5.2: Ejemplo para importar una clase de un archivo

```
from isaaclab_assets.robots.ant import ANT_CFG
```

Seguidamente, se comienza a definir la clase. En primer lugar, se definen distintos atributos concretos. Entre ellos se encuentran los ya mencionados `episode_length_s`, `decimation` y `observation_space`. También se definen algunos nuevos atributos, como `action_scale`, que sirve para escalar la acción en el procesado. Seguidamente se configura la simulación (código 5.3). En el constructor, se definen dos atributos principales: `dt`, que define el tiempo entre los pasos del proceso, y `render_interval`, que define cada cuánto se actualiza la visualización. Después se define el atributo `terrain`, con una clase `TerrainImporter`. Este atributo define cómo será el suelo, desde su construcción hasta sus propiedades físicas. En este caso, no se considera relevante detallarlo ya que genere un plano simple, pero en el apartado de mejoras, se estudia detenidamente esta clase para generar otro tipo de terrenos.

Listing 5.3: Definición de la configuración de la simulación

```
sim: SimulationCfg = SimulationCfg(dt=1 / 120, render_interval=
    decimation)
```

Continuando dentro de la clase, se define el atributo `scene`, mediante una clase `InteractiveSceneCfg` (código 5.4). Dentro de esta clase, se definen con el constructor distintos parámetros referentes al número de entornos. En IsaacLab se entrena con múltiples copias de un mismo entorno en paralelo. Esta clase es la encargada de almacenarlos y gestionarlos. Por ello, se deben definir algunos parámetros relevantes como el número de entornos (`num_envs`), el espacio entre estos (`env_spacing`) y la forma de clonado (`replicate_physics` y `clone_fabric`). Justo después, se define el atributo `robot`, encargado de configurar el robot del entorno. Este atributo se asocia a una constante, importada, como antes se ha visto, de un archivo aparte. En el próximo apartado se muestra cómo se configura el robot araña. También se define el atributo `joint_gears`, encargado de ajustar la fuerza aplicada en las acciones. Estas acciones van estrechamente relacionadas con la configuración del robot, configuradas en la constante importada.

Listing 5.4: Definición de la configuración de la escena

```
scene: InteractiveSceneCfg = InteractiveSceneCfg(
    num_envs=4096, env_spacing=4.0, replicate_physics=True,
    clone_in_fabric=True)
```

Por último, para terminar de definir la configuración del entorno, se deben indicar los pesos que se utilizan para cada recompensa. En el apartado 5.3.5 se describen cuales son estas recompensas y cómo se aplica este peso. No obstante, antes de llegar a estas se analiza la configuración del robot.

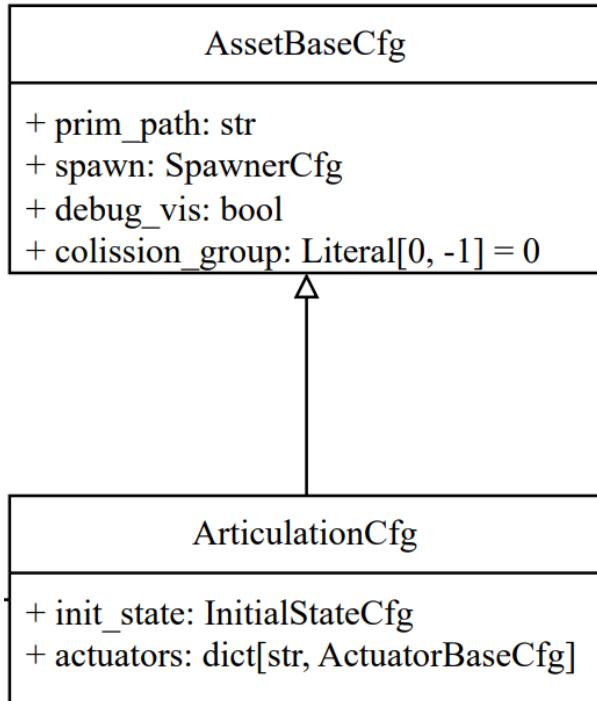


Figura 5.7: Imagen del diagrama referente a la clase `ArticulationCfg`

5.3.4. `ArticulationCfg`

La clase `ArticulationCfg` sirve para configurar la implementación del robot del entorno. Esta configuración se puede definir a través de su constructor. En este apartado, se estudia la implementación de esta clase para el caso concreto de locomoción del robot araña. Esta implementación se realiza en el archivo `source/isaaclab_assets/isaaclab_assets/robots/ant.py` [40], cuyo código se muestra en 5.5.

Esta clase hereda de `AssetBaseCfg`, dirigida a configurar cada prim de la simulación. Esta clase asocia el prim a una ruta dentro del mundo (definido en el apartado 4.2) y define la forma en la que se crea, normalmente a través de un archivo USD. También define si este archivo es visible, a través del atributo `debug_vis` y con que objetos puede colisionar, mediante el atributo `collision_group`. Por esto, se utilizan estos mismos atributos para definir el robot.

Listing 5.5: Implementación de la clase `ArticulationCfg`

```

from __future__ import annotations

import isaaclab.sim as sim_utils
from isaaclab.actuators import ImplicitActuatorCfg
from isaaclab.assets import ArticulationCfg
from isaaclab.utils.assets import ISAAC_NUCLEUS_DIR

ANT_CFG = ArticulationCfg(

```

```

prim_path="{ENV_REGEX_NS}/Robot",
spawn=sim_utils.UsdFileCfg(
    usd_path=f"{ISAAC_NUCLEUS_DIR}/Robots/IsaacSim/Ant/
        ant_instanceable.usd",
    rigid_props=sim_utils.RigidBodyPropertiesCfg(
        disable_gravity=False,
        max_depenetration_velocity=10.0,
        enable_gyroscopic_forces=True,
    ),
    articulation_props=sim_utils.
        ArticulationRootPropertiesCfg(
            enabled_self_collisions=False,
            solver_position_iteration_count=4,
            solver_velocity_iteration_count=0,
            sleep_threshold=0.005,
            stabilization_threshold=0.001,
        ),
    copy_from_source=False,
),
init_state=ArticulationCfg.InitialStateCfg(
    pos=(0.0, 0.0, 0.5),
    joint_pos={
        ".*_leg": 0.0,
        "front_left_foot": 0.785398, # 45 degrees
        "front_right_foot": -0.785398,
        "left_back_foot": -0.785398,
        "right_back_foot": 0.785398,
    },
),
actuators={
    "body": ImplicitActuatorCfg(
        joint_names_expr=[".*"],
        stiffness=0.0,
        damping=0.0,
    ),
},
)

```

Esta implementación se almacena en la constante `ANT_CFG`, que posteriormente se importa dentro de la configuración del entorno, como se ha indicado en el apartado anterior. En el constructor, se definen los dos atributos heredados de la clase `AssetBaseCfg`.

En primer lugar, se define el atributo `prim_path`, el cual define la ruta donde se guarda el elemento primitivo. Este atributo usa una cadena formateada que permite almacenarlo en cada uno de los entornos, manteniendo el mismo esquema. En segundo lugar, se define el atributo `spawn`, que define la creación del primitivo. Este atributo se define a través de una clase `UsdFileCfg`. Esta clase indica el archivo que se utiliza para generar el robot en la escena, mediante el atributo `usd_path`. Este archivo se encuentra guardado dentro de IsaacSim, por lo que se usa la constante `ISAAC_NUCLEUS_DIR`, que apunta a los archivos de esta aplicación. También se definen las propiedades relevantes de la articulación con los atributos `rigid_props` y `articulation_props`. Por último, mediante el atributo `copy_from_source`, se indica si se usa una copia del archivo o el propio archivo. En este caso, al no realizar modificaciones, se indica con un valor `False` el uso del archivo original.

Los otros dos atributos que se deben indicar en el constructor son `init_state` y `actuators`. Por un lado, `init_state` define la posición inicial del robot mediante la clase `InitialStateCfg`. En el constructor de esta clase, se debe indicar la posición del robot referente al mundo, mediante el atributo `pos`; y la posición de las articulaciones. La posición de las articulaciones se indica mediante un diccionario. En él, a todas las patas se les asocia el mismo valor, utilizando una cadena con el carácter `"*"`. Esto hace que todas las articulaciones terminadas en `"_leg"` se asocie el mismo valor a todas las articulaciones. Por otro lado, el atributo `actuators` define el movimiento de las articulaciones, definiéndose a través de un diccionario. En este caso, se define un único tipo de movimiento mediante `ImplicitActuatorCfg`, en la cual se asocia el movimiento a todas las articulaciones y se dan los valores de rigidez (`stiffness`) y amortiguación (`damping`).

Una vez definido el robot mediante esta clase, se tienen todos los elementos necesarios para construir el entorno. En el siguiente apartado, se estudia la clase `LocomotionEnv`, que hereda de `DirectRLEnv` y define los entornos e interacciones de las tareas de locomoción.

5.3.5. LocomotionEnv

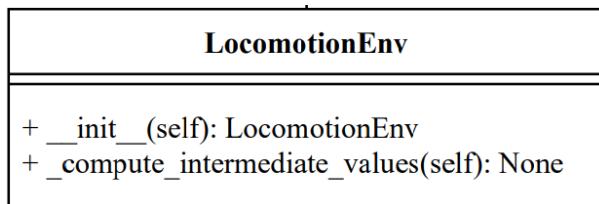


Figura 5.8: Imagen del diagrama referente a la clase `LocomotionEnv`.

En este apartado se estudia la definición de la clase `LocomotionEnv`. Al ser el ele-

mento principal que describe la tarea se analiza cada uno de sus métodos, viendo tanto su objetivo como el desarrollo del código. Para ello, se expone el método y su objetivo, después se muestra el código y se explica el contenido. Todo el código de la clase se encuentra en el repositorio de la herramienta IsaacLab, en el directorio `source/isaaclab_tasks/isaaclab_tasks/direct/locomotion/locomotion_env.py`.

El primer método implementado en el archivo es `normalize_angle(x)` (código 5.6). Este método se encarga de utilizar herramientas de PyTorch para normalizar el ángulo. Con esta función, los ángulos se transforman a un rango $[-\pi, \pi]$. Esto convierte a los ángulos en números más fáciles de tratar, pues se evita el uso de números mayores con el incremento por vuelta.

Listing 5.6: Definición del método `normalize_angle(x)`

```
def normalize_angle(x):
    return torch.atan2(torch.sin(x), torch.cos(x))
```

Este método se implementa utilizando la función de PyTorch `torch.atan2(y, x)` [45], la cual devuelve un valor en el rango estipulado. Esta función es alimentada con otras dos funciones de esta biblioteca `torch.sin(x)` y `torch.cos(x)` [45]. Estas hacen que se preserve la dirección angular y se elimine el número de vueltas. Como se muestra en el resto de métodos, se implementan múltiples funciones de PyTorch, ya que en todo momento se trabaja con tensores, que, como ya se ha comentado, permiten almacenar la información de todos los entornos en un único lugar. Estos métodos se almacenan en el código mediante el objeto módulo `torch`.

Definida esta función, que resulta de utilidad en próximos métodos, se declara la clase `LocomotionEnv`. El primer método definido en esta es su constructor. El constructor encarga almacenar todos los datos relevantes que se conocen en el momento de la inicialización, así como declarar tensores relevantes para el cálculo de las recompensas y observaciones. A continuación, se expone el código de este método en el listado 5.7.

Listing 5.7: Definición del constructor de la clase `LocomotionEnv`

```
def __init__(self, cfg: DirectRLEnvCfg, render_mode: str | None = None, **kwargs):
    super().__init__(cfg, render_mode, **kwargs)

    self.action_scale = self.cfg.action_scale
    self.joint_gears = torch.tensor(self.cfg.joint_gears,
                                    dtype=torch.float32, device=self.sim.device)
    self.motor_effort_ratio = torch.ones_like(self.
                                              joint_gears, device=self.sim.device)
    self._joint_dof_idx, _ = self.robot.find_joints(".*")
```

```

        self.potentials = torch.zeros(self.num_envs, dtype=torch.
            float32, device=self.sim.device)
        self.prev_potentials = torch.zeros_like(self.potentials)
        self.targets = torch.tensor([1000, 0, 0], dtype=torch.
            float32, device=self.sim.device).repeat(
            (self.num_envs, 1))
    )
    self.targets += self.scene.env_origins
    self.start_rotation = torch.tensor([1, 0, 0, 0], device=
        self.sim.device, dtype=torch.float32)
    self.up_vec = torch.tensor([0, 0, 1], dtype=torch.float32
        , device=self.sim.device).repeat((self.num_envs, 1))
    self.heading_vec = torch.tensor([1, 0, 0], dtype=torch.
        float32, device=self.sim.device).repeat(
        (self.num_envs, 1))
    )
    self.inv_start_rot = quat_conjugate(self.start_rotation).
        repeat((self.num_envs, 1))
    self.basis_vec0 = self.heading_vec.clone()
    self.basis_vec1 = self.up_vec.clone()

```

En primer lugar, se utiliza el constructor de `DirectRLEnv`. Este permite instanciar todos aquellos valores que se necesitan por defecto, como la configuración o el número de entornos. Después se definen distintos atributos relevantes al procesado de las acciones. Se define la escala de las acciones, se transforma el vector de la ponderación de la fuerza, se declara un tensor completo a uno con la dimensión del vector de la ponderación y se recoge el nombre de las distintas articulaciones. En estos atributos destaca el uso del método `torch.tensor` [45], que permite crear un tensor, y del método `torch.ones_like` [45], que permite crear un tensor inicializado completamente a uno con la dimensión del referenciado. Seguidamente, se definen todos los atributos relevantes al cálculo de las observaciones y las recompensas. Algunos atributos, como `potentials` o `prev_potentials`, al tratarse más adelante, se inicializan a cero, mediante el método `torch.zeros` [45], en el cual se indica directamente la dimensión; o el método `torch.zeros_like`, donde la dimensión se da indirectamente a través de un sensor. Otros, como `start_rotation`, que indica la rotación inicial de la araña, o `heading_vec`, que indica un vector de referencia para el avance, se definen directamente con `torch.tensor`. En algunos de ellos, se utiliza el método `torch.Tensor.repeat()` [45], que permite duplicar el tensor. En el caso de `inv_start_rot`, donde se calcula el inverso de la rotación inicial, se repite dicho valor por el número de entornos en el eje 0 y por 1 en el eje 1, quedando un tensor de forma

[num_envs, 4].

El siguiente método que se define es `_set_up_scene`. Este es el encargado de, con los elementos definidos en la configuración, crear la escena. El código de la definición de este método se muestra a continuación (código 5.8).

Listing 5.8: Definición del método `_set_up_scene(self)` de la clase `LocomotionEnv`

```
def _setup_scene(self):
    self.robot = Articulation(self.cfg.robot)
    # inclusion del plano del entorno
    self.cfg.terrain.num_envs = self.scene.cfg.num_envs
    self.cfg.terrain.env_spacing = self.scene.cfg.env_spacing
    self.terrain = self.cfg.terrain.class_type(self.cfg.
        terrain)
    # clonar y replicar
    self.scene.clone_environments(copy_from_source=False)
    # incluir la articulacion a la escena
    self.scene.articulations["robot"] = self.robot
    # add lights
    light_cfg = sim_utils.DomeLightCfg(intensity=2000.0,
        color=(0.75, 0.75, 0.75))
    light_cfg.func("/World/Light", light_cfg)
```

Lo primero que realiza este método es instanciar una clase `Articulation` mediante la configuración del robot, almacenada en la configuración del entorno. Después, se completa la configuración del terreno con el número de entornos y el espacio entre ellos definido, y se instancia de la misma manera. Seguidamente, se define la forma de clonado de los entornos, en este caso, al negar `copy_from_source`, los entornos clonados no heredan los estados del original, siendo así independientes de este. El robot, al ser la pieza central, tiene un hueco asignado dentro de la escena, por lo que se debe asociar a esta, a pesar de tenerlo declarado también en otra variable. Por último, se configuran las luces para la visualización de la escena.

Dos métodos, que en este caso van estrechamente relacionados, son `_pre_physics_step()` y `_apply_action()`. Pese a que tienen objetivos distintos (como se describe en el apartado 5.3.1), en este caso ambos métodos tratan las acciones; esto se puede observar en el siguiente código (código 5.9)

Listing 5.9: Definición del método `_pre_physics_step(self)` y `_apply_action` de la clase `LocomotionEnv`

```
def _pre_physics_step(self, actions: torch.Tensor):
    self.actions = actions.clone()
```

```

def _apply_action(self):
    forces = self.action_scale * self.joint_gears * self.
        actions
    self.robot.set_joint_effort_target(forces, joint_ids=self.
        ._joint_dof_idx)

```

En el método `_pre_physics_step` se guarda la acción a realizar, indicada dentro del método. Por otro lado, en el método `_apply_action()` se calculan las fuerzas, mediante los distintos parámetros de escala y proporción, y luego se aplican a las articulaciones del robot.

El siguiente método definido es `_compute_intermediate_values(self)`. Este se declara directamente en esta clase, no es heredado de su clase base. Su objetivo es realizar una serie de cálculos para calcular distintos valores intermedios. Estos valores luego se utilizan en distintas evaluaciones del proceso de aprendizaje. Este método utiliza una función `compute_intermediate_rewards(...)`, definida fuera de la clase; esta es la que contiene estos cálculos. Esta definición se muestra en el siguiente código (código 5.10).

Listing 5.10: Definición del método `compute_intermediate_rewards(...)`

```

def compute_intermediate_values(
    targets: torch.Tensor,
    torso_position: torch.Tensor,
    torso_rotation: torch.Tensor,
    velocity: torch.Tensor,
    ang_velocity: torch.Tensor,
    dof_pos: torch.Tensor,
    dof_lower_limits: torch.Tensor,
    dof_upper_limits: torch.Tensor,
    inv_start_rot: torch.Tensor,
    basis_vec0: torch.Tensor,
    basis_vec1: torch.Tensor,
    potentials: torch.Tensor,
    prev_potentials: torch.Tensor,
    dt: float,
):
    to_target = targets - torso_position
    to_target[:, 2] = 0.0

    torso_quat, up_proj, heading_proj, up_vec, heading_vec =
        compute_heading_and_up(
            torso_rotation, inv_start_rot, to_target, basis_vec0,
            basis_vec1, 2

```

```

        )

    vel_loc, angvel_loc, roll, pitch, yaw, angle_to_target =
        compute_rot(
            torso_quat, velocity, ang_velocity, targets,
            torso_position
        )

    dof_pos_scaled = torch_utils.maths.unscale(dof_pos,
                                                dof_lower_limits, dof_upper_limits)

    to_target = targets - torso_position
    to_target[:, 2] = 0.0
    prev_potentials[:] = potentials
    potentials = -torch.norm(to_target, p=2, dim=-1) / dt

    return (
        up_proj,
        heading_proj,
        up_vec,
        heading_vec,
        vel_loc,
        angvel_loc,
        roll,
        pitch,
        yaw,
        angle_to_target,
        dof_pos_scaled,
        prev_potentials,
        potentials,
    )

```

A continuación, se van a analizar cada uno de los parámetros calculados:

- **up_proj**: corresponde a la proyección escalar del vector que indica la dirección superior del torso de la araña. Es decir, el vector normal al torso. Se calcula en referencia a la rotación del torso, su rotación inicial y la referencia antes dada de la dirección superior objetivo, [0, 0, 1].
- **heading_proj**: este atributo, de manera análoga al anterior, indica el vector dirección de la araña, tomando de referencia el vector de dirección definido [1, 0, 0].

- **up_vec**: vector que representa la dirección superior del torso. La proyección de este sobre el vector de dirección superior, [0, 0, 1], resultaría en el atributo **up_proj**
- **heading_vec**: de manera análoga al anterior, este atributo representa el vector de la dirección. La proyección de este sobre el vector de dirección superior, [0, 0, 1], resulta en el atributo **heading_proj**.
- **vel_loc**: indica la velocidad con la cual el robot se acerca al objetivo establecido en **targets**.
- **roll, pitch, yaw**: indica la rotación del torso de la araña en esa convención. Lo hace a partir del cuaternión almacenado en **torso_quat**.
- **angle_to_target**: indica el ángulo entre el vector de dirección y el vector hacia el objetivo.
- **dof_pos_scaled**: almacena la posición de las articulaciones, desnormalizadas respecto a sus límites.
- **prev_potentials**: almacena el valor anterior de los potenciales.
- **potentials**: calcula si el robot se acerca o se aleja del objetivo.

Cada uno de estos elementos son relevantes en la valoración y observación de los entornos, por lo que este método se debe ejecutar en cada paso de la simulación.

A continuación, se analiza el método `_get_observations(self)`. Este es un método de abstracto de la clase base y debe utilizarse para recoger las observaciones del entorno. Para ello, se conforma un tensor de un único eje donde vienen todos los valores de las observaciones. La dimensión de este único eje es el número de entradas que tendrá la red neuronal. Este método se define de la siguiente manera (código 5.11):

Listing 5.11: Definición del método `_get_observations(self)`

```
def _get_observations(self) -> dict:
    obs = torch.cat(
        (
            self.torso_position[:, 2].view(-1, 1),
            self.vel_loc,
            self.angvel_loc * self.cfg.angular_velocity_scale,
            normalize_angle(self.yaw).unsqueeze(-1),
            normalize_angle(self.roll).unsqueeze(-1),
            normalize_angle(self.angle_to_target).unsqueeze(-1),
            self.up_proj.unsqueeze(-1),
            self.heading_proj.unsqueeze(-1),
```

```

        self.dof_pos_scaled,
        self.dof_vel * self.cfg.dof_vel_scale,
        self.actions,
    ),
    dim=-1,
)
observations = {"policy": obs}
return observations

```

Esta método utiliza la función `torch.cat` para concatenar una serie de vectores. Estos vectores conforman las distintas observaciones que se realizarán sobre el entorno. Entre ellas se encuentran las siguientes:

- `torso_position[:, 2]`: La altura del torso, es decir, su posición en el eje Z.
- `vel_loc`: La velocidad de aproximación al objetivo, calculada en `_compute_intermediate_rewards(self)`.
- `angvel_loc`: La velocidad angular con la que se aproxima al objetivo, calculada en `_compute_intermidiate_rewards(self)`.
- `yaw, roll`: Ángulos de rotación sobre el eje X y el eje Z. Se normalizan ambos.
- `angle_to_target`: Ángulo respecto al objetivo, calculada en `_compute_intermidiate_rewards(self)`.
- `up_proj`: Proyección del normal al torso, calculada en `_compute_intermidiate_rewards(self)`.
- `heading_proj`: Proyección del vector de dirección del torso, calculada en `_compute_intermidiate_rewards(self)`.
- `dof_pos_scaled`: La posición de las articulaciones escalada, calculada en `_compute_intermidiate_rewards(self)`.
- `dof_vel`: La velocidad de las articulaciones, la cual se escala multiplicando por el parámetro correspondiente.
- `actions`: La última acción registrada.

Conviene destacar que todas las variables deben darse con dos ejes, uno de dimensión igual al número de entornos y otro con dimensión igual al tamaño de dicha observación. Para ello, en algunos casos se usa la función `torch.unsqueeze(dim)` [45]. Esta función añade un eje en la dimensión indicada. Por ejemplo, la variable `yaw`, es del tipo

[num_envs]; con esta función se convierte en [num_envs, 1]. También se usa el método `torch.Tensor.view(*shape)` [45], que cambia la forma, indicando el tamaño. El valor -1 calcula automáticamente la dimensión del eje. En este caso, se obtendría un vector de la forma [num_envs, 1]. Completando el método `torch.cat`, se indica el eje sobre el que se hace la concatenación; al indicar -1, sería el último eje. La observación luego se almacena en un diccionario con la cadena `policy` se devuelve.

Una vez obtenidas las observaciones, se continúa utilizando los parámetros de nuestro entorno para calcular las recompensas. Esto se hace con el método heredado `_get_rewards(self)`. Este método, al igual que `_compute_intermediate_values(self)`, utiliza una función definida fuera de la clase: `compute_rewards(...)`. El retorno de este método será el resultado de la clase principal, por lo que se analizará esta más detenidamente. A continuación, se muestra el código (código 5.12):

Listing 5.12: Definición del método `compute_rewards(self)`

```
def compute_rewards(
    actions: torch.Tensor,
    reset_terminated: torch.Tensor,
    up_weight: float,
    heading_weight: float,
    heading_proj: torch.Tensor,
    up_proj: torch.Tensor,
    dof_vel: torch.Tensor,
    dof_pos_scaled: torch.Tensor,
    potentials: torch.Tensor,
    prev_potentials: torch.Tensor,
    actions_cost_scale: float,
    energy_cost_scale: float,
    dof_vel_scale: float,
    death_cost: float,
    alive_reward_scale: float,
    motor_effort_ratio: torch.Tensor,
):
    heading_weight_tensor = torch.ones_like(heading_proj) *
        heading_weight
    heading_reward = torch.where(heading_proj > 0.8,
        heading_weight_tensor, heading_weight * heading_proj / 0.8)

    # aligning up axis of robot and environment
    up_reward = torch.zeros_like(heading_reward)
    up_reward = torch.where(up_proj > 0.93, up_reward + up_weight
        , up_reward)
```

```

# energy penalty for movement
actions_cost = torch.sum(actions**2, dim=-1)
electricity_cost = torch.sum(
    torch.abs(actions * dof_vel * dof_vel_scale) *
    motor_effort_ratio.unsqueeze(0),
    dim=-1,
)

# dof at limit cost
dof_at_limit_cost = torch.sum(dof_pos_scaled > 0.98, dim=-1)

# reward for duration of staying alive
alive_reward = torch.ones_like(potentials) *
    alive_reward_scale
progress_reward = potentials - prev_potentials

total_reward = (
    progress_reward
    + alive_reward
    + up_reward
    + heading_reward
    - actions_cost_scale * actions_cost
    - energy_cost_scale * electricity_cost
    - dof_at_limit_cost
)
# adjust reward for fallen agents
total_reward = torch.where(reset_terminated, torch.ones_like(
    total_reward) * death_cost, total_reward)
return total_reward

```

En este método se calculan una serie de recompensas que luego se suman al final para obtener un único valor numérico. El valor final, almacenado en `total_reward`, puede tener dos valores: un valor fijo derivado del coste de terminación o un valor variable dependiendo de las recompensas obtenidas. La selección de este valor se da mediante el método `torch.where` [45]. En él, se introduce un tensor de valores booleanos que indican cuáles de los entornos han finalizado. En aquellos que el entorno haya finalizado, se guarda el coste de terminación como recompensa, y en el resto, la recompensa variable. Esta recompensa variable está conformada por las siguientes recompensas:

- `progress_reward`: recompensa por tender a acercarse o tender a alejarse. Se calcula

a partir de los potenciales, siendo esta recompensa la variación de este.

- `alive_reward`: recompensa por continuar el ejercicio.
- `up_reward`: recompensa por mantener el torso paralelo al suelo.
- `heading_reward`: recompensa por mantener la dirección del torso hacia el objetivo.
- `action_cost`: penalización por el uso de acciones.
- `electricity_cost`: penalización por el gasto energético.
- `dof_at_limit_cost`: penalización por forzar las articulaciones a su límite.

Todas las recompensas son tensores con forma [num_envs, 1]. Estas recompensas se pueden calcular de distintas maneras. Por ejemplo, `up_reward` se calcula de manera absoluta. Si la proyección sobrepasa un valor fijo, obtiene la totalidad de la recompensa, de lo contrario, obtiene un valor nulo. `heading_reward`, por otro lado, se calcula de manera que se toma un valor estándar, y se obtiene una recompensa progresiva hasta llegar a este, donde se obtiene la recompensa total. Estas distintas formas de calcular las recompensas dependen de lo que se quiera conseguir. En este caso, se es más flexible con la dirección, pero se busca mantener el torso paralelo al suelo dentro del rango indicado.

Dentro de estas recompensas cobra relevancia el atributo `reset_terminated`, el cual indica los entornos que han finalizado. Este atributo se calcula con el método `_get_dones(self)`, que es el siguiente método definido. Este método también es heredado de la clase base. Se define mediante el siguiente código (código 5.14):

Listing 5.13: Definición del método `_get_dones(self)`

```
def _get_dones(self) -> tuple[torch.Tensor, torch.Tensor]:  
    self._compute_intermediate_values()  
    time_out = self.episode_length_buf >= self.  
        max_episode_length - 1  
    died = self.torso_position[:, 2] < self.cfg.  
        termination_height  
    return died, time_out
```

En esta clase, antes de calcular las terminaciones, se llama al método `_compute_intermediate_rewards(self)`. Esto se debe a que el método `_get_dones(self)` se declara al comienzo de cada paso de simulación, por lo que de este modo, se calculan en cada momento los valores necesarios. Después de esto, se comprueban los dos casos de finalización. El primero de ellos se da cuando se supera el tiempo máximo de episodio; el segundo cuando la posición del torso baja de un límite. En ambos casos, se almacena un tensor de forma [num_envs, 1], donde el último eje guarda un valor booleano. Los dos tensores tienen

funciones distintas. Ambos indicarán una terminación del episodio del entorno, pero solo los indicados en `died` tendrán penalización [41, isaaclab.envs, DirectRLEnv, `step(self, action)`].

Estos tensores indican por lo tanto qué entornos deben reiniciarse, pues han llegado al final de su episodio. Este reinicio se define dentro del método `_reset_idx(self)`, también heredado de la clase base. El código del método es el siguiente:

Listing 5.14: Definición del método `_reset_idx(self)`

```
def _reset_idx(self, env_ids: torch.Tensor | None):
    if env_ids is None or len(env_ids) == self.num_envs:
        env_ids = self.robot._ALL_INDICES
    self.robot.reset(env_ids)
    super().__reset_idx(env_ids)

    joint_pos = self.robot.data.default_joint_pos[env_ids]
    joint_vel = self.robot.data.default_joint_vel[env_ids]
    default_root_state = self.robot.data.default_root_state[
        env_ids]
    default_root_state[:, :3] += self.scene.env_origins[
        env_ids]

    self.robot.write_root_pose_to_sim(default_root_state[:, :7], env_ids)
    self.robot.write_root_velocity_to_sim(default_root_state[:, 7:], env_ids)
    self.robot.write_joint_state_to_sim(joint_pos, joint_vel,
                                        None, env_ids)

    to_target = self.targets[env_ids] - default_root_state[:, :3]
    to_target[:, 2] = 0.0
    self.potentials[env_ids] = -torch.norm(to_target, p=2,
                                           dim=-1) / self.cfg.sim.dt

    self._compute_intermediate_values()
```

Este método recibe un tensor con los entornos a reiniciar indicados en `env_ids`. En este entorno, al tener como único elemento el robot, primero se identifica cada uno de los robots a reiniciar. Después, se hace el reinicio generalizado, mediante el método de la clase padre. Una vez reiniciados los entornos, se lleva el robot al estado y posición originales, usando las variables almacenadas dentro de la clase. También se vuelve a calcular el objetivo,

ahora desde la posición original; así como los potenciales. Por último, al haber variado las posiciones y los estados del robot, se deben volver a calcular los valores intermedios.

En esta clase, se definen todos los aspectos relevantes del entorno. De esta manera, se definen los pasos a seguir dentro de la simulación, indicando cada una de las interacciones entre el entorno y el agente. Más adelante, en el capítulo siguiente, se aprecia la diferencia con la arquitectura por manejadores. En esta clase, se ha podido estudiar profundamente cómo se estructuran y calculan las observaciones y las recompensas, así como la realización de reinicios y la creación de los elementos de la escena; en el modo por manejadores, el enfoque estará situado en las clases y no tanto en la definición de los métodos.

Antes del estudio acerca del registro de este entorno se analiza la clase final, `AntEnv`, que hereda de esta clase y modifica los distintos atributos de esta para adaptarla a su caso específico. Esta clase no se puede instanciar todavía, pues falta por definir atributos como los pesos de las recompensas o el robot del entorno. Todos estos atributos ya han sido definidos dentro de la configuración de la clase. La clase `AntEnv` queda definida en el mismo archivo que la clase `AntEnvCfg`. Se muestra su definición en el código 5.15. Esta clase solo indica el tipo de configuración que recibe, `AntEnvCfg`, y utiliza el constructor de la clase padre para instanciar el entorno. Al tener todos los parámetros definidos dentro de la configuración y heredar de la clase `LocomotionEnv`, obteniendo la definición de las interacciones y construcciones, no hace falta definir ningún elemento más, pudiendo pasar directamente a su registro.

Listing 5.15: Definición de la clase `AntEnv`

```
class AntEnv(LocomotionEnv):
    cfg: AntEnvCfg

    def __init__(self, cfg: AntEnvCfg, render_mode: str | None =
        None, **kwargs):
        super().__init__(cfg, render_mode, **kwargs)
```

5.4. Registro del Entorno

El registro del entorno sirve para registrar una tarea para su entrenamiento. Este proceso se hace en dos pasos: configurar el algoritmo de entrenamiento y registrarla dentro de *gymnasium*.

La primera parte de este proceso se realiza en el archivo `source/isaaclab_tasks/isaaclab_tasks/direct/ant/agents/rsl_rl_ppo_cfg.py`. En este archivo se definen los principales parámetros del entrenamiento. Primero, se definen parámetros para la pro-

pia ejecución de la simulación junto con el entrenamiento. Entre ellos, el número máximo de iteraciones, el guardado de la política o el nombre del ejercicio. Después se define el formato de la política, indicando el ruido y los parámetros propios del actor y el crítico que forman esta. Por último, se define el formato del algoritmo, indicando parámetros propios del PPO a implementar. Como el enfoque de este trabajo se centra en los entornos, no se entrará en gran detalle sobre esta configuración.

La segunda parte de este proceso registra el entorno y la configuración del algoritmo dentro de *gymnasium* [50]. Para ello se define el siguiente código (código 5.16):

Listing 5.16: Registro del entorno *Ant*

```
gym.register(
    id="Isaac-Ant-Direct-v0",
    entry_point=f"__name__.ant_env:AntEnv",
    disable_env_checker=True,
    kwargs={
        "env_cfg_entry_point": f"__name__.ant_env:AntEnvCfg",
        "rl_games_cfg_entry_point": f"{agents.__name__}:
            rl_games_ppo_cfg.yaml",
        "rsl_rl_cfg_entry_point": f"{agents.__name__}.
            rsl_rl_ppo_cfg:AntPPORunnerCfg",
        "skrl_cfg_entry_point": f"{agents.__name__}:skrl_ppo_cfg.
            yaml",
    },
)
```

Se utiliza el método `gymnasium.register`, el cual registra el entorno junto a la configuración del agente. En el entrenamiento, se extraen ambas partes y se implementan para realizar el entrenamiento. Este proceso se realiza a través de los *scripts* propios de la librería de aprendizaje y es el mismo para cualquier ejercicio. Por tanto, el ejercicio de entrenamiento dependerá de dos factores. Por un lado, el entorno; este se define mediante el atributo `entry_point`, que indica la clase del entorno principal, y, dentro del atributo `kwargs`, la cadena asociada a `env_cfg_entry_point`, que indica la clase configuración del entorno. Dentro de `kwargs` se define también la otra parte del ejercicio, la configuración del agente. En el párrafo anterior, se ha centrado en la biblioteca `rsl_rl`, la que se usará en este proyecto; sin embargo, en este ejercicio se definen también las demás bibliotecas. Como se observa, se puede llamar a un archivo o, en este caso, a una clase `AntPPORunnerCfg`.

Con la clase registrada, se pueden realizar tanto los ejercicios de entrenamiento como de evaluación. En el siguiente apartado, se muestra cómo ejecutar ambos procesos a través de la terminal.

5.5. Aprendizaje y Evaluación

Una vez estudiado tanto el registro del entorno como su construcción, se procede a realizar el ejercicio de entrenamiento. Para ello, se utilizan los archivos contenidos en `IsaacLab/scripts/reinforcement_learning/rsl_rl`. Dentro de esta carpeta se encuentran los códigos para realizar el aprendizaje, en `train.py`, y la evaluación, `play.py`. Ambos códigos son propios de la biblioteca `rsl_rl` y la herramienta IsaacLab, por lo que no se van a analizar individualmente. En este apartado se estudia cómo ejecutar ambos códigos a través de la terminal, indicando con las etiquetas todos los datos necesarios.

El primer paso es realizar un entrenamiento de prueba, para probar qué el entorno se genere adecuadamente y el entrenamiento se pueda realizar de la forma indicada. Para ello, se utiliza la siguiente sentencia por terminal (código 5.17):

Listing 5.17: Entrenamiento de prueba para la locomoción de *Ant*

```
python scripts/reinforcement_learning/rsl_rl/train.py --task  
Isaac-Ant-Direct-v0 --num_envs 4
```

Con esta sentencia ejecutamos el entrenamiento, indicando la tarea que se quiere entrenar y el número de entornos a utilizar. En un primer momento se utilizan un número pequeño de entornos para valorar la construcción del entorno dentro del simulador. Al ejecutarlo, se abre la aplicación IsaacSim y se generan cuatro de los robots a entrenar (figura 5.5). Se observa cómo los cuatro robots mueven sus articulaciones de forma errática y al caer su torso sobre el suelo se reinicia el entorno.

Una vez se ha realizado una prueba satisfactoria, se procede a realizar el entrenamiento completo. Para ello, se debe generar una mayor cantidad de entornos, con el objetivo de acelerar el proceso. Para ello, se utiliza la siguiente sentencia (código 5.18):

Listing 5.18: Entrenamiento completo para el entorno *Ant*

```
python scripts/reinforcement_learning/rsl_rl/train.py --task  
Isaac-Ant-Direct-v0 --num_envs 512 --headless
```

Con esta sentencia, se ejecuta el mismo código de entrenamiento, pero aumentando el número de entornos a 512 y utilizando la nueva etiqueta `--headless`. Esta última etiqueta permite ejecutar el entrenamiento sin visualizarlo en el simulador, lo que permite optimizar el proceso en realizarlo en menos tiempo. Para poder evaluar el proceso de entrenamiento se obtiene una serie de información en la terminal. Esta información se muestra en la figura 5.5. Dentro del cajetín se obtiene información acerca del número de iteraciones o la longitud del episodio. El valor más relevante para evaluar el proceso de entrenamiento es la recompensa media. Esta no solo ayuda al robot al aprendizaje si no que da información sobre como se desenvuelve el agente dentro en la tarea. En su última iteración (figura



Figura 5.9: Prueba para el entrenamiento del robot araña

5.5), la recompensa media asciende a 12055.38, por lo que se puede intuir que la araña realiza correctamente el ejercicio de locomoción.

```

Learning iteration 20/1000

    Computation: 41704 steps/s (collection: 0.273s, learning 0.120s)
    Mean action noise std: 0.85
    Mean value function loss: 118.4774
    Mean surrogate loss: -0.0079
    Mean entropy loss: 10.1072
    Mean reward: -75.92
    Mean episode length: 449.42

    Total timesteps: 344064
    Iteration time: 0.39s
    Time elapsed: 00:00:09
    ETA: 00:07:20

```

Figura 5.10: Primeras iteraciones del ejercicio de aprendizaje

El resultado de este aprendizaje consiste de una serie de modelos, almacenados en la carpeta `IsaacLab/logs/rsl_r1`. Dentro de esta carpeta se almacena un modelo cada 50 iteraciones (este parámetro se puede variar dentro de la configuración del agente). Estos modelos contienen la red neuronal con la política implementada en cada iteración. Sin embargo, estos modelos no se pueden utilizar; para ello, primero se debe seleccionar uno de los modelos y evaluarlo con `play.py`. Este fichero se ejecuta mediante la siguiente sentencia (código 5.19):

```

Learning iteration 999/1000

    Computation: 40951 steps/s (collection: 0.272s, learning 0.128s)
    Mean action noise std: 0.30
Mean value function loss: 4095.8117
    Mean surrogate loss: -0.0049
    Mean entropy loss: 1.6020
    Mean reward: 12055.38
    Mean episode length: 832.07
-----
Total timesteps: 16384000
Iteration time: 0.40s
Time elapsed: 00:06:06
ETA: 00:00:00

```

Figura 5.11: Última iteración del ejercicio de aprendizaje

Listing 5.19: Entrenamiento para la evaluación de la política generada para el entorno *Ant*

```
python scripts/reinforcement_learning/rsl_rl/play.py --task Isaac-Ant-Direct-v0 --num_envs 4 --checkpoint logs/rsl_rl/ant_direct/2025-12-29_10-04-20/model_900.pt
```

Al ejecutar esta sentencia se abre el simulador, pudiendo estudiar el movimiento de la araña. A través de una inspección visual se confirma que el movimiento de la araña es correcto, por lo que el valor alto de recompensa se adecua a las expectativas. Además de esta evaluación, al ejecutar este fichero se obtiene un archivo `policy.pt`, almacenado en el directorio `logs/rsl_rl/ant_direct/load_run/exported`. Este archivo sí puede ser exportado y utilizado, permitiendo implementarla en el robot real. Sin embargo, este ejercicio no está diseñado para dicha implementación al tratarse de un ejemplo teórico. En el siguiente apartado se proponen ciertas mejoras para posibles futuras implementaciones.

5.6. Posibles mejoras

Con vistas a los futuros proyectos dentro de la ETSIDI, se van a proponer e integrar dos principales mejoras. Estas se introducirán en este trabajo, pero su expansión e implementación final se darán en futuros proyectos. La primera de ellas consta de cambiar el terreno donde se entrena la locomoción. En el ejercicio realizado, el terreno es completamente plano, algo poco usual en el mundo real. La segunda mejora a integrar es el uso de cámaras. Estas podrán ayudar a evitar obstáculos o mejorar la locomoción en terrenos irregulares.

5.6.1. Terreno irregular

Para introducir el terreno irregular se va utilizar una nueva clase de configuración `TerrainGeneratorCfg`. Esta clase permite definir un nuevo tipo de terreno, que luego es seleccionado dentro de `TerrainImporterCfg`. En primer lugar, se define la generación del terreno y, después, mediante los atributos `terran_type` y `terrain_generator`, se genera el terreno definido.

La definición de la generación del terreno es la siguiente (código 5.20):

Listing 5.20: Definición del terreno con relieve a generar.

```
RANDOM_ROUGH_CFG = terrain_gen.TerrainGeneratorCfg(
    size = (100.0, 100.0),
    num_rows = 1,
    num_cols = 1,
    horizontal_scale=0.1,
    vertical_scale=0.005,
    slope_threshold=0.75,
    use_cache=False,
    sub_terrains = {
        "random_rough": terrain_gen.HfRandomUniformTerrainCfg(
            proportion= 1.0, noise_range=(0.02, 0.10), noise_step
            =0.002
        )
    }
)
```

Mediante este código, se define, de forma externa a la clase de configuración, una constante con la clase pertinente. Esta clase define un terreno irregular cuadrado de 100 metros de lado. Para generar el propio terreno, se indica como único sub-terreno una clase `HfRandomUniformTerrainCfg` [41], característica de un terreno rugoso. Con su constructor, se indica la proporción de este terrero (al ser el único sub-terreno, 1), la altura máxima y mínima del terreno (indicado mediante `noise_range`) y la resolución de dicho terreno, es decir, la distancia mínima entre dos puntos adyacentes. En otros casos, se puede hacer uso del concepto de sub-terreno para crear distintas zonas con distintos tipos de terreno. En este caso, al ser meramente una prueba, basta con generar un único tipo de terreno.

5.6.2. Cámaras

La otra mejora a implementar es el uso de cámaras. Existen dos tipos de cámaras para incluir en la simulación, cámaras normales, generadas por la clase `CameraCfg`, y

cámaras de mosaico, definidas por la clase `TiledCameraCfg`. Ambas clases se definen de forma análoga, su diferencia radica en el procesamiento de las imágenes, estando optimizado en las cámaras de mosaico. En este caso se usará directamente la cámara de mosaico. El código para su configuración es el siguiente, (código 5.21):

Listing 5.21: Definición de la configuración de la cámara a generar.

```
camera: TiledCameraCfg = TiledCameraCfg(
    prim_path="/World/envs/env_*/Robot/torso/FrontCamera",
    update_period= 0.1,
    height = 64,
    width= 64,
    data_types= ["rgb", "depth"],
    spawn= PinholeCameraCfg(
        focal_length = 24.0, focus_distance = 400, clipping_range
        =(0.1, 20)
    ),
    offset= TiledCameraCfg.OffsetCfg(pos = (0.3, 0, 0), rot =
        (0.9239,0,-0.3827,0), convention = "world")
)
```

En este código incluido dentro de la clase de configuración, `RoughAntEnvCfg`, se define la configuración de la cámara. La cámara se acopla al lateral del torso de la araña, con una rotación sobre el eje y de -45 grados, de modo que apunte a la nueva superficie rugosa generada. Se permiten recopilar dos tipos de datos: los colores rgb y la profundidad. Se ajusta también los parámetros de la cámara, como el tipo de cámara a usar, mediante la clase `PinholeCameraCfg`, el tiempo de actualización y la resolución. Esta configuración debe ser instanciada manualmente. Para ello, al definir la nueva clase principal para el entrenamiento, `RoughAntEnv`, se debe re-definir la función `_set_up_scene(self)` para generar la cámara (código 5.22).

Listing 5.22: Implementación de la cámara en la escena.

```
def _setup_scene(self):
    super().___setup_scene()
    self.camera = TiledCamera(self.cfg.camera)
```

El mayor inconveniente de utilizar las cámaras es el tamaño de su información. Para cualquier cámara, si se quiere usar la información de la profundidad, se deberá tratar con un tensor de forma [num_envs, height, width, 1]. Esto quiere decir que para cada entorno, tomando el ejemplo propuesto, si se quisiese usar la imagen al completo como observación, se tendría un vector de observaciones con una dimensión de 4096 elementos; usando de ejemplo una resolución pobre como la propuesta (64×64). Esto hace su uso in-

viable en la mayoría de casos. Sin embargo, se pueden sortear estas dificultades realizando el procesamiento de imágenes externamente o pre-procesando la información para reducir el tamaño de la observación.

Este ejercicio se deja para futuros trabajos. En el enfoque actual se pretende cimentar las bases para que otros alumnos o interesados puedan trabajar en este campo. El uso de visión artificial junto con la inteligencia artificial sobresale como un tema interesante para su estudio.

Todas estas mejoras se pueden encontrar en el proyecto preparado para el trabajo, incluido en los anexos. El archivo donde se implementan dichas mejoras es el `source/ARMetaToolPG/ARMetaToolPG/tasks/direct/ant/rough_ant.py`

Analizado este problema, se analiza un nuevo ejemplo en el siguiente capítulo. A diferencia de este, en él se utiliza el modo por manejadores.

Capítulo 6

Reach

En este capítulo se va estudiar un nuevo ejercicio de aprendizaje por refuerzo. Para este caso, se estudiará la construcción de entornos por manejadores. Debido a la constitución del código en esta forma de trabajar, el análisis será menos profundo. Esta forma permite estructurar el código de manera más superficial, organizándose en dos elementos: manejadores y términos.

Para el estudio se desarrolla y analiza el diagrama de clases. Dentro de este diagrama de clases se analiza su pieza central, `UR3EnvCfg`, la cual define la estructura de manejadores. Después se estudia, al igual que en el capítulo anterior, el registro, entrenamiento y evaluación del ejercicio de entrenamiento. Por último, se proponen e implementan algunas mejoras para este caso. Antes de todo esto, se va a presentar el caso de estudio.

6.1. Descripción del caso práctico

Para este segundo caso de estudio se transfiere el enfoque al ámbito de la manipulación. Se realiza un ejercicio *reach*, un paso previo a cualquier problema de manipulación. En él, se requiere llevar el efecto final del robot a una posición y orientación concreta. El robot elegido es el UR3, de *Universal Robots* [9]. Se ha escogido este robot por dos motivos: su disponibilidad en el laboratorio de la universidad y el uso de este robot en el proyecto MetaTool. El caso, por otro lado, ha sido escogido para introducir la forma de programación por manejadores y rotar hacia el enfoque de trabajo de MetaTool. No obstante, es un caso seguro de implementar, que a su vez permite la recolección de datos de posición y orientación objetivos. Por ello, también se selecciona para realizar la implementación final.

Este código parte de una base, un ejemplo de las mismas características dentro de la herramienta IsaacLab. Gracias al gran volumen de ejemplos, lo común es par-

tir de uno de estos ejemplos y realizar modificaciones al código. Esta forma de trabajar es la que se ha implementado, partiendo del código de IsaacLab contenido en `source/isaaclab_tasks/isaaclab_tasks/manager_based/manipulation/reach/reach_env_cfg.py` dentro de la herramienta IsaacLab.

A continuación, se presenta el estudio del código, comenzando por el análisis del diagrama de clases desarrollado.

6.2. Diagrama de clases

En la figura 6.2 se muestra el diagrama de clases preparado para este ejemplo. A primera vista, se puede observar que los elementos del diagrama orbitan alrededor de la clase `ReachEnvCfg`. Esta clase hereda de `ManagerBasedCfg` y sirve de base para las clases del entorno específico, `UR3ReachEnvCfg` y `UR3ReachEnvCfg_PLAY`. De manera análoga al ejercicio anterior, existe una clase para una tarea común, `ReachEnvCfg`.

La clase `ReachEnvCfg` define la tarea general a implementar. Para cada aspecto general tiene un manejador distinto, los cuales se analizan detenidamente en futuros apartados. Cada uno de estos manejadores tiene como atributos distintos términos, que conforman la configuración de cada elemento. Por ejemplo, el manejador `RewardsCfg` está constituido por una serie de términos `RewTerm`.

A partir de esta clase heredan `UR3ReachEnvCfg` y `UR3ReachEnvCfg_PLAY`; la primera de estas pensada para el entrenamiento y la segunda para la evaluación. Cabe notar que no se hereda a partir de la clase principal `ManagerBasedRLEnv`, sino únicamente de las configuraciones. Esto ocurre ya que, a diferencia de la manera directa, en esta forma de programar se definen únicamente las piezas y elementos de las interacciones, los procesos propios de esta vienen ya definidos en la clase principal base.

En el siguiente apartado, se analiza cada uno de los manejadores individualmente, inspeccionando el código utilizado para cada uno de ellos.

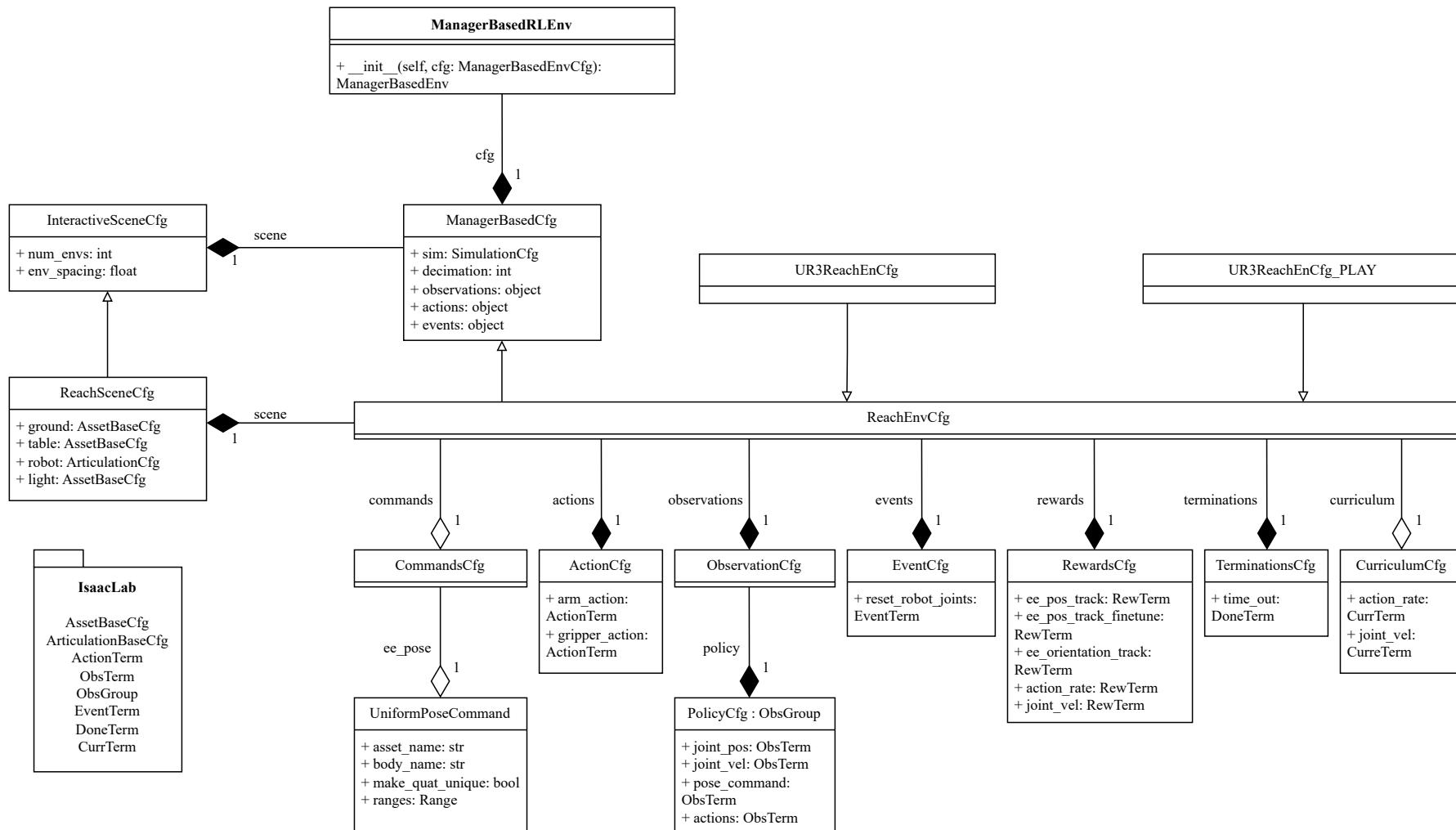


Figura 6.1: Diagrama UML del ejemplo *reach*, programación por manejadores.

6.3. Manejadores

En este apartado se estudia cada uno de los manejadores detenidamente. Todos los manejadores y clases se definen en el archivo `source/isaaclab_tasks/isaaclab_tasks/manager_base/ach/reach_env_cfg.py`

ReachSceneCfg La primera clase definida dentro del código no es un manejador como tal, sino la clase con la que se definen los elementos de la escena. No obstante, pese a no encontrarse entre los manejadores, se trata dentro del código como uno, definiendo los distintos elementos que lo componen y dejando su creación final a la clase base. Al estar tratando con la clase de configuración de la tarea base, no se define ningún elemento específico. Todos ellos se definen a partir de la clase `AssetBaseCfg`, la clase básica para la definición de *primitivos* [41].

El código utilizado para la definición de esta escena es el siguiente (código 6.1):

Listing 6.1: Definición de la escena mediante la clase `ReachSceneCfg`

```
@configclass
class ReachSceneCfg(InteractiveSceneCfg):
    # plano
    ground = AssetBaseCfg(
        prim_path="/World/ground",
        spawn=sim_utils.GroundPlaneCfg(),
        init_state=AssetBaseCfgInitialStateCfg(pos=(0.0, 0.0,
                                                    -1.05)),
    )
    # mesa
    table = AssetBaseCfg(
        prim_path="{ENV_REGEX_NS}/Table",
        spawn=sim_utils.UsdFileCfg(
            usd_path=f"{ISAAC_NUCLEUS_DIR}/Props/Mounts/
                    SeattleLabTable/table_instanceable.usd",
        ),
        init_state=AssetBaseCfgInitialStateCfg(pos=(0.55, 0.0,
                                                    0.0), rot=(0.70711, 0.0, 0.0, 0.70711)),
    )
    # robots
    robot: ArticulationCfg = MISSING
    # luces
    light = AssetBaseCfg(
        prim_path="/World/light",
```

```

        spawn=sim_utils.DomeLightCfg(color=(0.75, 0.75, 0.75),
                                       intensity=2500.0),
    )

```

Se declaran 4 elementos a crear en la escena. El primero de ellos es el plano general, almacenado en la variable `ground`. Este plano se define utilizando el plano por defecto, situándolo aproximadamente a un metro del origen. Después, se define la mesa de la escena, utilizando una mesa propia de IsaacLab y se sitúa aproximadamente a medio metro del origen de la escena. Seguidamente, se declara el robot. Al ser este un elemento específico de la aplicación se utiliza la constante `MISSING`, para poder reemplazarlo más adelante. Por último, se define el tipo de iluminación. Estos elementos se almacenan dentro de cada entorno como *primitivos* (con la constante `ENV_REGEX_NS`) o en el mundo (con el directorio `/World/`). De esta manera, la escena queda estructurada a través de sus distintos elementos para que la clase `ManagerBasedRLEnv` pueda crearla directamente.

CommandsCfg La siguiente clase definida es el primer manejador definido, `CommandsCfg`. Este manejador se encarga de generar objetivos para el ejercicio de entrenamiento. En el caso en cuestión, el objetivo es llevar el efecto final a una posición y orientación concreta. Para poder llevar a distintos puntos el efecto se debe entrenar para un rango de puntos; si se entrena para un punto concreto, solo podría llegar a este.

Con esta clase se pretende generar una serie de puntos objetivo para entrenar el movimiento hacia estos. El código sería el siguiente (código 6.2):

Listing 6.2: Definición de los objetivos de entrenamiento mediante la clase `CommandsCfg`

```

@configclass
class CommandsCfg:
    """Command terms for the MDP."""

    ee_pose = mdp.UniformPoseCommandCfg(
        asset_name="robot",
        body_name=MISSING,
        resampling_time_range=(4.0, 4.0),
        debug_vis=True,
        ranges=mdp.UniformPoseCommandCfg.Ranges(
            pos_x=(0.35, 0.65),
            pos_y=(-0.2, 0.2),
            pos_z=(0.15, 0.5),
            roll=(0.0, 0.0),
            pitch=MISSING, # depends on end-effector axis
            yaw=(-3.14, 3.14),

```

```
) ,  
)
```

Dentro de la clase, se define una única variable donde se almacenarán las posiciones objetivo. Todas las posiciones van referenciadas al origen de coordenadas del entorno. Estos puntos deben ir referidos además al efecto final, indicando el robot en `asset_name` y la articulación o enlace al que va referenciado, `body_name`. Se indica también el intervalo de tiempo en el que se actualiza esta posición, en este caso 4 segundos, y si el punto es visible en la simulación. Por último, mediante la variable `ranges`, se define el rango de posiciones y rotaciones a generar.

ActionCfg Una vez definidos los objetivos del aprendizaje, se definen las acciones a través de su manejador `ActionsCfg`. Este manejador define mediante cada término los distintos bloques de acciones. En este caso, (código 6.3), se definen dos tipos de movimientos que tiene cualquier brazo robótico, el movimiento del brazo y el del *gripper*.

Listing 6.3: Declaración de las acciones dentro de su manejador, `ActionsCfg`

```
class ActionsCfg:  
    arm_action: ActionTerm = MISSING  
    gripper_action: ActionTerm | None = None
```

ObservationsCfg El siguiente manejador a tratar es el que define las observaciones. Dentro de este manejador se definen las distintas observaciones que se realizan sobre el entorno y se entregan al agente. Esta información viene estructurada dentro de una clase propia que hereda de `ObsGroup`. Esto es necesario para que la clase principal, `ManagerBasedRLEnv`, pueda concatenar las observaciones manteniendo su orden. Este orden cobra relevancia al implementar la red neuronal, pues se debe procesar un vector de observaciones con la misma forma. El código utilizado para definir este manejador es el siguiente (código 6.4):

Listing 6.4: Definición de las observaciones dentro de su manejador, `ObservationCfg`

```
class ObservationsCfg:  
    @configclass  
    class PolicyCfg(ObsGroup):  
        # observation terms (order preserved)  
        joint_pos = ObsTerm(func=mdp.joint_pos_rel, noise=Unoise(  
            n_min=-0.01, n_max=0.01))  
        joint_vel = ObsTerm(func=mdp.joint_vel_rel, noise=Unoise(  
            n_min=-0.01, n_max=0.01))
```

```

pose_command = ObsTerm(func=mdp.generated_commands,
    params={"command_name": "ee_pose"})
actions = ObsTerm(func=mdp.last_action)

def __post_init__(self):
    self.enable_corruption = True
    self.concatenate_terms = True

# observation groups
policy: PolicyCfg = PolicyCfg()

```

De este modo, a través de términos, se definen 4 observaciones:

- `joint_pos`: posición relativa a la posición inicial.
- `joint_vel`: velocidad relativa a la velocidad inicial.
- `pose_command`: objetivo para el alcance.
- `actions`: las acciones anteriores.

A parte de definir estas 4 observaciones, dentro de la clase `PolicyCfg` se habilita el ruido (utilizado en las dos primeras observaciones), mediante el atributo `enable_corruption`; así como indicar la concatenación de las observaciones. Por último, ya dentro del manejador, se almacena este grupo en una variable `policy`.

EventCfg Otro de los manejadores que se debe definir es el de eventos. Este manejador se encarga de realizar procesos en momentos puntuales del proceso. Para este caso se define un único evento que ocurre al reiniciar el entorno: el reinicio de la posición de las articulaciones. Este reinicio además es escalado, lo que significa que la posición inicial variará dentro del rango estipulado (código 6.5).

Listing 6.5: Definición de los eventos dentro de su manejador, `EventsCfg`

```

class EventCfg:
    """Configuration for events."""

    reset_robot_joints = EventTerm(
        func=mdp.reset_joints_by_scale,
        mode="reset",
        params={
            "position_range": (0.5, 1.5),
            "velocity_range": (0.0, 0.0),

```

```

} ,
)

```

RewardsCfg Las recompensas también se definen a través de un manejador. En el su caso, al no ser necesario organizarlas de una manera concreta, pues se combinarán en un solo valor numérico, se pueden declarar a través de términos. Cada término tiene asociado una función con la que calcular la recompensa, junto con los argumentos de dicha función, y el peso de dicha recompensa. En el caso de estudio, este manejador se define de la siguiente forma (código 6.6):

Listing 6.6: Definición de las recompensas dentro de su manejador, RewardsCfg

```

class RewardsCfg:
    """Reward terms for the MDP."""

    # task terms
    end_effector_position_tracking = RewTerm(
        func=mdp.position_command_error,
        weight=-0.2,
        params={"asset_cfg": SceneEntityCfg("robot", body_names=
            MISSING), "command_name": "ee_pose"},

    )
    end_effector_position_tracking_fine_grained = RewTerm(
        func=mdp.position_command_error_tanh,
        weight=0.1,
        params={"asset_cfg": SceneEntityCfg("robot", body_names=
            MISSING), "std": 0.1, "command_name": "ee_pose"},

    )
    end_effector_orientation_tracking = RewTerm(
        func=mdp.orientation_command_error,
        weight=-0.1,
        params={"asset_cfg": SceneEntityCfg("robot", body_names=
            MISSING), "command_name": "ee_pose"},

    )

    # action penalty
    action_rate = RewTerm(func=mdp.action_rate_12, weight
        =-0.0001)
    joint_vel = RewTerm(
        func=mdp.joint_vel_12,
        weight=-0.0001,
        params={"asset_cfg": SceneEntityCfg("robot")},


```

```
)
```

Dentro del manejador se definen 5 recompensas:

- `end_effector_position_tracking`: penalización por el error de posición en la posición.
- `end_effector_position_tracking_fine_grained`: recompensas por encontrarse cerca de la posición objetivo.
- `end_effector_orientation_tracking`: penalización por el error de la orientación.
- `action_rate`: penalización por el uso de las acciones.
- `joint_vel`: penalización por la velocidad de las articulaciones.

Todas estas recompensas se suman para obtener la recompensa global. Cabe resaltar de nuevo la presencia de la constante `MISSING`, que deberá sustituirse al implementar la configuración específica del entorno.

TerminationsCfg Otro de los manejadores que deben definirse es el manejador para las terminaciones. Este manejador se encarga de identificar cuando los episodios de los entornos debe terminarse. En el caso de estudio, solo existe una condición, que se agote el tiempo de simulación (código 6.7)

Listing 6.7: Definición de las terminaciones dentro de su manejador, `TerminationsCfg`

```
@configclass
class TerminationsCfg:
    """Termination terms for the MDP."""

    time_out = DoneTerm(func=mdp.time_out, time_out=True)
```

CurriculumCfg El último manejador definido es el del currículum. Este manejador es el encargado de alterar parámetros de la simulación para que se adapten a nuevas condiciones de entrenamiento. En este caso, se utiliza para modificar el peso de las penalizaciones por acciones y velocidades, buscando, después de un tiempo de entrenamiento, mejorar la eficiencia del movimiento (código 6.8).

Listing 6.8: Definición del currículum dentro de su manejador, `CurriculumCfg`

```
@configclass
class CurriculumCfg:
    """Curriculum terms for the MDP."""
```

```

action_rate = CurrTerm(
    func=mdp.modify_reward_weight, params={"term_name": "action_rate", "weight": -0.005, "num_steps": 4500}
)

joint_vel = CurrTerm(
    func=mdp.modify_reward_weight, params={"term_name": "joint_vel", "weight": -0.001, "num_steps": 4500}
)

```

Una vez definidos los manejadores, estos se almacenan dentro del configurador general, `ReachEnvCfg`, tal y como se muestra en el diagrama de clases 6.2. En esta clase también se definen algunos parámetros de interés dentro de la función `__pos_init__(self)`, como por ejemplo el atributo `decimation` o el `episode_length_s`. Sin embargo, donde se concentran las definiciones más relevantes es en los manejadores y en la clase de configuración específica, `UR3ReachEnvCfg`.

6.4. Configuración y registro del entorno específico

La configuración específica del entorno, `UR3ReachEnvCfg`, usando el robot UR3e con gripper, viene definida dentro del archivo `source/ARMetaToolPG/ARMetaToolPG/tasks/manager.py` alojado dentro del proyecto general de este TFG. Para ajustar esta configuración al entorno y poder instanciar la clase, se definirán una serie de atributos dentro del método `__pos_init__(self)`.

El primero de ellos que se definirá será el atributo `robot`. Para ello, en un archivo aparte, `source/ARMetaToolPG/ARMetaToolPG/assets/robots/ur3_configuration.py`, se ha preparado la configuración de este robot, almacenándola en la constante `UR3e`. Dentro de esta configuración se utiliza un archivo `usd`, que es representativo del UR3e con gripper, y fue cedido por el equipo de MetaTool [8].

Seguidamente se indica la referencia del robot para las recompensas. En el apartado anterior se ha visto como la referencia del robot quedaba indicada como `MISSING`. Ahora, esta referencia se ajusta a una parte del robot que tiene como centro de coordenadas el centro del gripper `gripper_center`. Continuando con aspectos del robot, se definen también las acciones del brazo, las cuales también habían quedado pendiente. Por último, se reajusta la generación de coordenadas, donde se indica el rango del entrenamiento.

Además de esta clase de configuración, específica para el entrenamiento, se define otra para su posterior evaluación. Esta hereda de la clase anterior, sobre la cual se hacen

dos únicas modificaciones: el numero de entornos y su espaciado, y la inhabilitación del ruido en las observaciones.

Teniendo ambas configuraciones definidas y la configuración del agente previamente definida, ahora en `source/ARMetaToolPG/ARMetaToolPG/tasks/manager_based/reach/config/ur_3/agents/rsl_rl_ppo_cfg.py`, se puede registrar, por separado, las tareas de entrenamiento y evaluación. Este registro se da en el archivo: `source/ARMetaToolPG/ARMetaToolPG/tasks/manager_based/reach/config/ur_3/agents/rsl_rl_ppo_cfg.py`. Para ambos casos, se vuelve a realizar el registro como en el capítulo anterior, esta vez utilizando como punto de entrada para la clase general `ManagerBasedRLEnv`.

Teniendo ambas tareas registradas, se puede pasar a realizar el entrenamiento del agente. Esto se verá en el próximo apartado.

6.5. Entrenamiento y evaluación

De una misma manera que en el capítulo anterior, se va realizar el aprendizaje y evaluación de este ejercicio. En primer lugar, se ejecuta un entrenamiento de prueba con pocos entornos para comprobar que estos se generan correctamente. Después de comprobar que esto ocurre (figura 6.5), se puede proceder al entrenamiento con un mayor número de entornos, sin necesidad de ejecutar el simulador.

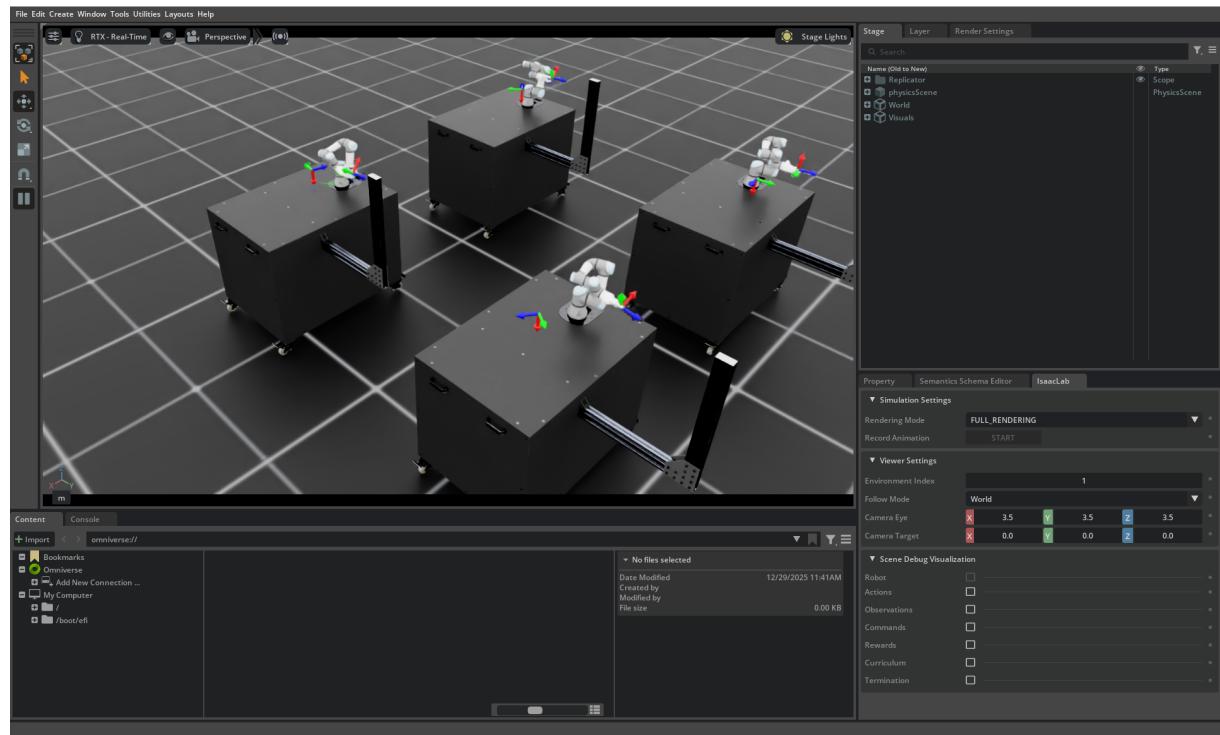


Figura 6.2: Prueba para el entrenamiento del robot araña

Para este caso se utilizarán 512 entornos. Es interesante notar que al utilizar la forma de manejadores, al descomponer las recompensas en términos individuales, el entre-

namiento entrega la información de las recompensas desglosada; a diferencia de la forma directa, donde solo se obtenía el valor de la recompensa global media. Esto nos permite tener más información y poder detectar fácilmente posibles fallos en el entrenamiento. Una vez terminado el entrenamiento, se observa que la recompensa es negativa. Esto, sin embargo, no es relevante, pues lo importante es que la recompensa se maximice, no que esta sea alta. En este caso, al penalizar en la mayoría de recompensas, lo óptimo es que la recompensa sea próxima a 0, lo cual parece cumplirse. El siguiente paso, será evaluar con `play.py`, de modo que podamos estudiar si cumple los objetivos y si esta recompensa puede minimizarse de alguna forma.

Al evaluar la política de esta manera se observan un problema principal. Por un lado, existen posiciones que requieren que el robot contacte consigo mismo o con el suelo. Esto es un problema, pues puede inhabilitar el movimiento o dañar el robot. Como se espera poder implementar esta política en el robot real, en el próximo apartado se corregirá y mejorará el ejercicio para hacerlo apto para su implementación final.

6.6. Mejoras y correcciones

La implementación de este ejercicio se realizará en un robot UR3, con un gripper incluido. Por tanto, es importante obtener un modelo fiel a este robot; es decir, se debe tener en cuenta el gripper en el entrenamiento. Para ello, se ha utilizado un fichero .usd que contiene este gripper. En el proyecto general de este trabajo **ARMetaToolPG**, se puede encontrar este archivo en la carpeta `source/ARMetaToolPG/data/ur3e`. No obstante, para este ejercicio de alcance, no se tiene en cuenta las acciones del gripper; no se estudia agarre. Por ello, para facilitar el ejercicio y que la posición de la pinza no afecte a la política, se elimina los accionamientos de cierre. De este modo, se obtiene un modelo que tiene en cuenta el gripper, sin que este afecte en las observaciones al posicionamiento. Para este modelo se crea una nueva configuración, mostrada en el siguiente código 6.9:

Listing 6.9: Definición del modelo para el UR3 con gripper, `UR3_CFG_NP`

```
UR3_CFG_NP = ArticulationCfg(
    spawn=sim_utils.UsdFileCfg(
        usd_path=f'{ARMT_ASSETS_DATA_DIR}/ur3e/ur3e_gripper_np.
            usd',
        rigid_props=sim_utils.RigidBodyPropertiesCfg(
            disable_gravity=False,
            max_depenetration_velocity=5.0,
        ),
        articulation_props=sim_utils.
            ArticulationRootPropertiesCfg(
```

```

        enabled_self_collisions=True,
    )
),
init_state=ArticulationCfg.InitialStateCfg(
    joint_pos={
        "shoulder_pan_joint": -1.57,
        "shoulder_lift_joint": -1.744,
        "elbow_joint": 1.57,
        "wrist_1_joint": (4.71-6.283),
        "wrist_2_joint": (4.71-6.283),
        "wrist_3_joint": (4.36-6.283),
    },
    pos = [0.0, 0.0, 0.0],
    rot = [0.70710678, 0, 0, -0.70710678]
),
actuators={
    "arm": ImplicitActuatorCfg(
        joint_names_expr=[".*"],
        velocity_limit=100.0,
        effort_limit=87.0,
        stiffness=800.0,
        damping=40.0,
    ),
},
)

```

An su vez, en esta nueva configuración, se ajusta la rotación y la posición inicial del robot para que coincida con la del robot de pruebas.

Por otro lado, también con el objetivo de realizar pruebas, se va a ajustar los rangos de coordenadas. Por un lado, se va a definir un nuevo rango de operación, teniendo en cuenta el espacio donde se va a trabajar. Por otro lado, para analizar el ajuste de la orientación, se va a buscar mantener el gripper paralelo al suelo, de un mismo modo que el torso de la araña. Para ello, se van a mantener la rotación en *roll* y *pitch* constante, para variar únicamente la rotación en *yaw* (código 6.10).

Listing 6.10: Cambio en los rangos de entrenamiento, dentro de la clase UR3ReachEnvCfg

```

self.commands.ee_pose.ranges.pos_y = (-0.40, -0.20)
self.commands.ee_pose.ranges.pos_z = (0.15, 0.35)
self.commands.ee_pose.ranges.pos_x = (-0.20, 0.30)
self.commands.ee_pose.ranges.roll = (math.pi, math.pi)
self.commands.ee_pose.ranges.pitch = (0, 0)

```

```

self.commands.ee_pose.ranges.yaw = (-math.pi/2, math.pi
/2)

```

También se va cambiar la referencia del marco. Para ello, se utilizará el miembro virtual del modelo situado en el centro del gripper, `gripper_center` (código 6.11):

Listing 6.11: Cambio en las referencias del entrenamiento, dentro de la clase `UR3ReachEnvCfg`

```

self.rewards.end_effector_position_tracking.params["  

    asset_cfg"].body_names = ["gripper_center"]  

self.rewards.end_effector_position_tracking_fine_grained.  

    params["asset_cfg"].body_names = ["gripper_center"]  

self.rewards.end_effector_orientation_tracking.params["  

    asset_cfg"].body_names = ["gripper_center"]  

self.commands.ee_pose.body_name = "gripper_center"

```

Por último, se incluye una nueva penalización, en el caso de que se entre contacto con la mesa. Esta penalización se incluye después de haberse trabajado en el proyecto MetaTool, siendo una de las recompensas más utilizadas durante este trabajo. Esta última penalización, `touch_desk`, se verá en detalle en el próximo capítulo. A su vez, la evaluación de estas correcciones se dará en el último capítulo, junto al estudio del problema *sim2real*.

6.7. Conclusiones: Directo contra Manejadores

Una vez analizadas ambas formas de programar poseemos evaluar ambas conjuntamente. Como se ha comentado en el apartado anterior, la forma directa permite un mayor control del entorno, definiendo explícitamente la implementaciones de las acciones, la recogida de observaciones y la generación de recompensas. Al realizar el ejercicio por manejadores, esta definición viene predefinida, recayendo sobre el desarrollador únicamente la definición de las propias observaciones, recompensas y acciones. En tareas sencillas, o en casos donde el peso del entrenamiento no recaiga principalmente en estos elementos, sino en como se apliquen, el modo directo es el más útil. Sin embargo, para casos más complejos, con múltiples recompensas y observaciones, la manera más eficiente es por manejadores. Los ejemplos del proyecto MetaTool, que se verán en el próximo capítulo, utilizan esta segunda forma.

Capítulo 7

Trabajo dentro del proyecto MetaTool

Durante este trabajo, la contribución del equipo del proyecto MetaTool ha sido imprescindible. Por un lado, la gran parte del entrenamiento se ha realizado en ordenadores provistos por este equipo, trabajando dentro de su laboratorio en el CAR (Centro de Automática y Robótica, CSIC) [52]. Por otro lado, el investigador principal del laboratorio, Virgilio Gómez Lambo, aporto tanto visión como los distintos objetivos a cumplir dentro del laboratorio. Los objetivos propuestos fueron:

- Realización de un ejercicio lifting para una herramienta.
- Depuración de un ejercicio para el arrastre con herramienta.
- Implementación de un código para el problema de Sim2Sim.
- Implementación de un módulo para el problema de Sim2Real.

En este primer capítulo, se estudiará en primer lugar el proyecto, tratando de entender su misión y objetivos. Después se estudiará el objetivo de la contribución y la misión propia dentro del proyecto. Por último, se analizarán los dos primeros objetivos, viendo los ejercicios realizados. En estos casos, no se analizará el código completo, sino las partes de el que sean de interés.

Los dos otros objetivos conformaran el siguiente capítulo, cerrando los contenidos del trabajo para pasar a las conclusiones. Ahora, se va a estudiar la misión y propósito del proyecto global.

7.1. Misión y objetivos

Este proyecto se asienta en una idea central: el uso de la autoevaluación para la creación de herramientas. Se presupone que la creación de herramientas viene derivada de la capacidad humana de comprender su inhabilidad para la realización de ciertas tareas; primero utilizando herramientas naturales (como palos o piedras), para después crear nuevas mejor adaptadas a estas. Esta transición requiere el uso de una serie de herramientas, como por ejemplo la predicción, la meta-cognición, la abstracción y la creatividad; todas ellas asociadas al ser humano. La intención de este proyecto radica en utilizar herramientas de inteligencia artificial para adquirir estas habilidades. Para ello, se tienen tres objetivos principales:

- Estudiar las habilidades de meta-cognición y la capacidad de percepción como factores para el desarrollo de la fabricación de herramientas. Cabe especificar que la meta-cognición es la habilidad de auto-evaluar, regular y ser consciente de los procesos cognitivos internos [53].
- Desarrollar un modelo computacional para la creación de percepción sintética basada en herramientas de meta-cognición y predicción, desarrollando a su vez la fabricación de herramientas.
- Validar el modelo anterior mediante herramientas de inteligencia artificial.

El proyecto en si abarca una gran cantidad de terreno, sin embargo, el aporte de este trabajo será bastante limitado. En el siguiente apartado, se comentará el enfoque de este trabajo.

7.2. Objetivo de la aportación

Este trabajo de final de grado se centra principalmente en la construcción de entornos y el uso de la herramienta IsaacLab. Por esto, la aportación será limitada a estos conceptos. En vez de trabajar en elementos de percepción, se trabajará sobre el uso de herramientas a través de la inteligencia artificial.

En primer lugar, se trabajarán sobre los códigos ejemplo de IsaacLab [40] y propios códigos de MetaTool [8], para adaptar los distintos problemas a los objetivos del proyecto. Después, en el siguiente apartado, se diseñará una implementación sim2sim y sim2real para la implementación en robots reales, aplicables al proyecto MetaTool.

Se comenzará en el siguiente apartado con el primer objetivo, el levantamiento de una herramienta.

7.3. Levantamiento herramienta

El primer objetivo de esta parte del trabajo es realizar el levantamiento y agarre de una herramienta. Este ejercicio fue recomendado por los investigadores del proyecto para comenzar a realizar tareas de aprendizaje. Fue uno de los primeros ejercicios realizados. En él, se busca utilizar el robot *Franka*, para levantar un martillo de juguete.

Para la implementación de este ejercicio se utilizaron una serie de recursos. En primer lugar, se utilizó los códigos de la herramienta IsaacLab contenidos en `source/isaaclab_tasks/isaac`. Estos códigos formulan el problema para un cubo básico. La idea, en este como en gran parte de los ejercicios, es tomar un código para una tarea base; después analizarlo y entenderlo para finalmente modificarlo, adaptando el entorno al objetivo e incluyendo las observaciones y recompensas extras para obtener un resultado exitoso. En segundo lugar, se utilizaron los archivos usd provistos en el *Franka* por IsaacLab, y en el martillo, por MetaTool.

La primera modificación que se realizó fue la sustitución del cubo por la herramienta. Para ello, se alojó el archivo USD en la carpeta de datos del proyecto y se sustituyó el atributo `usd_path`, dentro del atributo `spawn` asociado al elemento de la escena `object`, para la clase específica `FrankaCubeLiftEnv`, renombrada en el proyecto `FrankaToolLiftEnv`. Una vez sustituido el cubo, se realizó un primer entrenamiento para observar los posibles problemas que produce esta sustitución. Se encontraron principalmente dos: el agarre no se daba en el punto correcto y el robot tocaba el suelo.

Para el primer problema se tomó la ruta más sencilla y eficiente, mover el eje de coordenadas. Para ello, se abrió el archivo de usd del martillo dentro de IsaacSim, seleccionando la malla y desplazando el eje hasta el mango. Al utilizar esta solución se deben tener en cuenta una serie de condicionantes. Pese a que este ejemplo no se integrará en el sim2real, el tomar el mango como punto central de la herramienta sobre el que tomar las recompensas es algo común en otros ejercicios. Esto quiere decir, que al tomar la posición de la herramienta se debe precisar exactamente el punto del cual se quiere agarrar esta. Esto, por otro lado, es un condicionante nuestro, y no derivado del entrenamiento.

Este punto fue una de las grandes lecciones derivadas de la colaboración con el proyecto MetaTool. Dentro del aprendizaje por refuerzo hay cierta información que se dan al robot y cierta información que se quiere que aprenda por su cuenta. La ideas que se den al robot limitaran a su vez el rango del aprendizaje. Por ejemplo, al darle el punto de agarre, no prueba distintas posiciones de agarre, las cuales podrían serle beneficiosas en este ejercicio. Sin embargo, si no le damos dicho punto, no se tiene una referencia clara para recompensarle por acercarse a la herramienta; también puede cogerlo desde un punto que no es interesante para el ejercicio. Se debe encontrar el punto exacto donde se

maximiza la eficiencia del entrenamiento, intentando limitar su entrenamiento lo mínimo posible.

Para el segundo problema, se decidió crear una nueva recompensa, o en este caso más bien penalización. Esta penalización se da cuando se entra en contacto con la mesa. Para detectar este contacto, se incluyó un sensor de contacto entre la mesa y el gripper. Esta sensor de contacto se debe definir dentro de la clase específica de configuración, `FrankaCubeLiftEnvCfg`. Se define de la siguiente manera (código 7.1):

Listing 7.1: Definición del sensor de contacto, dentro de la clase `FrankaCubeLiftEnvCfg`

```
self.scene.robot_desk_contact_sensor = ContactSensorCfg(
    prim_path="{ENV_REGEX_NS}/Table",
    filter_prim_paths_expr = ["{ENV_REGEX_NS}/Robot/*.*"],
    track_air_time=True,
    track_pose=True,
    update_period=0.0,
    debug_vis=False,
    force_threshold = 1.0
)
```

Este objeto calcula la fuerza ejercida entre la mesa y las distintas partes del robot. Para que se pueda instanciar este objeto, se necesita también definir como verdadera el atributo de la configuración de la mesa `contact_sensor_active`. Además, dentro de este sensor, se establece límite de 1.0, mediante la variable `force_threshold`. Esto hace que el sensor no detecte fuerzas menores de la indicada. Tomando en consideración este sensor, se definió la siguiente función para la penalización (código 7.2):

Listing 7.2: Definición del sensor de contacto,

```
def touch_desk(env: ManagerBasedRLEnv,
    robot_desk_contact_sensor_cfg: SceneEntityCfg =
        SceneEntityCfg("robot_desk_contact_sensor"),
):
    tool_contact_sensor: ContactSensor = env.scene[
        robot_desk_contact_sensor_cfg.name]
    contact_active = (torch.norm(tool_contact_sensor.data.
        net_forces_w, -1) > 0.0).bool()
    return torch.where(contact_active, 1, 0)
```

Con esta penalización se devuelve una señal 1 en el momento en que se detecta una fuerza en cualquiera de los ejes. Al tener un límite establecido, el sensor no salta con ruido inherente a un contacto real. De este modo, se penaliza que el robot toque la mesa, pero no lo prohíbe, pues contactos puntuales son inevitables en el agarre. También es importante

por esto ajustar bien la penalización. Al ser un valor binario, la penalización dependerá enteramente en el peso. En este caso, al querer evitar lo máximo posible el contacto, se escoge una ponderación de -3. Siendo premiado el alcance a 1 y el levantamiento a 15, este rango permite conseguir el objetivo.

7.4. Depuración del arraste con herramienta

En el segundo problema, se indicó desde el proyecto MetaTool que una serie de las recompensas del ejercicio de arrastre no funcionaba correctamente. Se decidió por tanto analizar el ejercicio al completo para poder entenderlo, puesto que se buscaba integrar este ejercicio en el sim2real, y resolver el problema de la recompensa.

El objetivo del entrenamiento es enseñar a un robot a agarrar una herramienta y con ella arrastrar un objeto, en este caso un cubo. Para ello, se utiliza el robot *Robohabilis* [8], mostrado en la figura 7.4. Este robot consiste de una base central a la que se conectan dos robots *UR3e* [9] y una cámara de visión, la cual no se contemplará en este trabajo.

Para esta tarea desde el proyecto MetaTool se diseñaron una serie de recompensas:

- `reaching_tool`: alcanzar la herramienta con el efecto final
- `reaching_object`: alcanzar la herramienta con la herramienta.
- `lifting_tool`: levantar la herramienta.
- `grasping_tool`: agarrar la herramienta.
- `pulling_object`: arrastrar el objeto.
- `object_goal_tracking`: llevar el objeto al objetivo.
- `object_goal_tracking_fine_grained`: llevar el objeto al objetivo, teniendo en cuenta un mayor grado de precisión. La anterior sería la recompensa por proximidad, esta sería una recompensa extra por alcanzarlo.
- `action_rate`: penalización por el uso de acciones.
- `joint_vel`: penalización por la cantidad de movimiento.

Las funciones que determinan el cálculo de estas recompensas, `func`, vienen definidas en el archivo `source/MT_ext/MT_ext/tasks/manipulation/pull_object/mdp/rewards.py`

De todas estas recompensas, donde se encontró un problema fue en la recompensa para el arrastre de la herramienta, `object_is_pulled`. Se observa dentro del entrenamiento, que a pesar de no estar realizándose el arrastre, la recompensa se da en todo

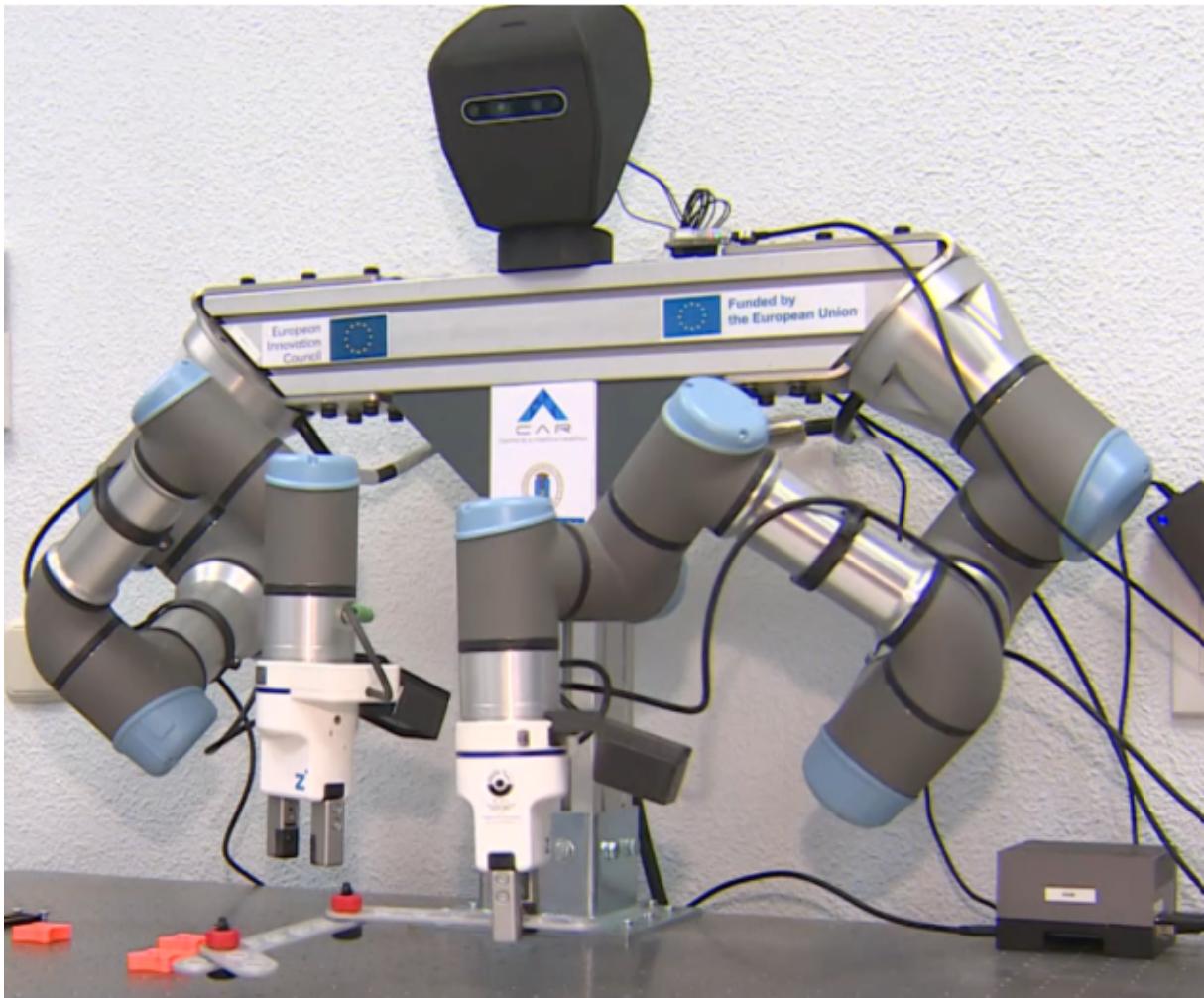


Figura 7.1: Robot RoboHabilis [54]

momento, obteniendo un promedio de alrededor de 9'7 sobre 10. Esto es un problema, ya que no permite evaluar el arrastre correctamente.

Esta recompensa se obtiene mediante la función `object_is_pulled(...)`. Esta función se muestra a continuación:

Listing 7.3: Definición de la función para el cálculo de la recompensa por arrastre.

```
def object_is_pulled(
    env: ManagerBasedRLEnv,
    std: float,
    tool_cfg: SceneEntityCfg = SceneEntityCfg("tool"),
    object_cfg: SceneEntityCfg = SceneEntityCfg("object"),
    object_contact_sensor_cfg: SceneEntityCfg = SceneEntityCfg("object_contact_sensor"),
    tool_contact_sensor_cfg: SceneEntityCfg = SceneEntityCfg("tool_contact_sensor"),
) -> torch.Tensor:
```

```

tool: RigidObject = env.scene[tool_cfg.name]
object: RigidObject = env.scene[object_cfg.name]
object_contact_sensor: ContactSensor = env.scene[
    object_contact_sensor_cfg.name]
tool_contact_sensor: ContactSensor = env.scene[
    tool_contact_sensor_cfg.name]
tool_contact_active = (tool_contact_sensor.data.
    current_contact_time.squeeze(-1) > 0).float()
object_contact_active = (object_contact_sensor.data.
    current_contact_time.squeeze(-1) > 0).float()
# Get positions of tool and object
tool_pos_w = tool.data.root_pos_w
object_pos_w = object.data.root_pos_w # End-effector position
: (num_envs, 3)
object_tool_distance = torch.norm(tool_pos_w - object_pos_w,
    dim=1) # Distance of the end-effector to the object: (
    num_envs,)
valid_contact = (tool_contact_active * object_contact_active).
    bool()
object_tool_distance = torch.where(valid_contact, 0.0,
    object_tool_distance)
# Compute distance-based reward component
distance_reward = 1 - torch.tanh(object_tool_distance / std)
#Combine contact presence and proximity reward
total_reward = object_contact_active * distance_reward *
    tool_contact_active
return total_reward

```

Antes de evaluar el problema, se va a analizar como se calcula la recompensa. Este análisis será también importante para ver un ejemplo de como se calculan las recompensas normalizadas.

En primer lugar, se extraen los distintos elementos del entorno. Entre ellos se encuentran la herramienta, el objeto y sus respectivos sensores. Seguidamente se comprueba el contacto en ambos sensores, utilizando el atributo `current_contact_time` de la clase `ContactSensor` [41]. El sensor de la herramienta se activa cuando hay un contacto entre la herramienta y el robot, mientras que para el objeto se activa con un contacto entre la herramienta y el objeto. Después se procede a calcular, la distancia entre el objeto y la herramienta. Añadido a este calculo, se supone que cuando hay un contacto activo, es decir, el robot toca la herramienta y la herramienta el objeto, la distancia es 0. Esta distancia después se normaliza con `torch.tanh()` [45], obteniendo valores entre -1 y 1; y

en el caso de 0, 1 como recompensa de la distancia. Esta recompensa se multiplica después por los contactos activos y se entrega finalmente.

A primera se puede ver que el cálculo de la posición no afecta en sí al cálculo de la recompensa, pues para obtener recompensa debe haber contacto activo y siempre que haya contacto activo se obtendrá una distancia de 0 y recompensa de 1. Por tanto, este cálculo se puede omitir. Sin embargo, esta no es la raíz del problema.

Habiendo realizado una depuración en tiempo de compilación, se observó que los sensores de contacto tienen un ruido. Este ruido hace que el sensor se active a pesar de no encontrar un contacto real. Para resolver este problema se introdujo una nueva variable dentro de los sensores de contacto, `force_threshold`. Esta modificación se realiza dentro de la clase específica de configuración, `RobohabilisCubePullEnvCfg`, definida dentro del archivo `source/MT_ext/MT_ext/tasks/manipulation/pull_object/config/robohabilis/joint_p...`. De este modo, se limita la fuerza con la cual el sensor se activa, evitando que salte con el ruido. Sin embargo, al volver a depurar el código, se observó como seguía activándose. Esto era debido a que la variable `contact_alive` no tiene en cuenta este límite. Por esto, se utilizó la variable `net_forces_w`, la cual si se ve afectada por esta variable. De este modo se solucionó al completo el problema, resultando en el siguiente código:

Listing 7.4: Corrección de la función para el cálculo de la recompensa por arrastre.

```
def object_is_pulled(
    env: ManagerBasedRLEnv,
    object_contact_sensor_cfg: SceneEntityCfg = SceneEntityCfg("object_contact_sensor"),
    tool_contact_sensor_cfg: SceneEntityCfg = SceneEntityCfg("tool_contact_sensor"),
) -> torch.Tensor:
    object_contact_sensor: ContactSensor = env.scene[
        object_contact_sensor_cfg.name]
    tool_contact_sensor: ContactSensor = env.scene[
        tool_contact_sensor_cfg.name]

    tool_contact_active = (torch.norm(tool_contact_sensor.data.net_forces_w, -1) > 0).float()
    object_contact_active = (torch.norm(object_contact_sensor.data.net_forces_w, -1) > 0).float()

    valid_contact = tool_contact_active * object_contact_active

    return valid_contact
```

De este modo, corrigiendo el problema principal, se consigue el objetivo fijado. La mejora en el nuevo ejercicio es que, al entrar en contacto con el objeto, este contacto se mantiene, resultando en un mayor control sobre la posición.

7.4.1. Conclusiones

Estos ejercicios fueron los primeros tratados dentro de este trabajo. Pese a que las correcciones necesarias fueron menores, el análisis completo para encontrar dichos errores, utilizando las distintas herramientas de depuración, permitió comprender el flujo de ejecución de los códigos a la estructuración de la información. Los conceptos anteriormente expuestos provienen en gran parte del análisis y corrección de estos ejercicios expuestos. A su vez, también fueron claves para el siguiente capítulo, donde se desarrollará un programa para la implementación de políticas.

Capítulo 8

El problema Sim2Real

El problema Sim2Real hace referencia a la diferencia entre el rendimiento de una política en el simulador y en el mundo real. Estas diferencias pueden deberse a multiples factores. Puede presentarse por la mala representación de características del entorno; también por la percepción del robot, que puede hacer variar las observaciones [55]. Estos y otros factores hacen que sea un problema común para todos los ejercicios de aprendizaje por refuerzo.

En este capítulo se presentarán algunas soluciones para el problema. Después, se seleccionará aquella que se adapte mejor al trabajo, implementando la solución dentro del ejercicio de entrenamiento. A continuación, se implementará una nueva simulación dentro de IsaacSim. En esta nueva simulación, no se usarán las herramientas de IsaacLab. El objetivo es probar la política en un entorno de simulación externo al probado. El siguiente paso será la implementación en el robot real. Para ello, se ha diseñado un modulo en python usando herramientas de UR-RTDE [56]. Por último, se realizará un ensayo, implementando una política en un robot.

8.1. Enfoques

El primer punto que se debe estudiar del problema Sim2Real son los distintos enfoques con los que se puede trabajar para minimizar su efecto. La manera más eficaz de poder adaptar a la implementación real es aleatorizar ciertos parámetros que puedan diferir en el mundo real. De este modo, se puede entrenar la política para afrontarse a variaciones en dichos parámetros. Teniendo en cuenta esto, surgen distintos tipos de aprendizaje [51]:

- Simulación ideal: no se incluye ningún parámetro aleatorizado.
- *Fine-tunning*: primero se entrena en una simulación ideal. Después, se identifican

los parámetros que necesitan ser aleatorizados y se entrena para cada uno de ellos por separado.

- Curriculum: de nuevo, entrena primero en una simulación ideal e identifica los parámetros a aleatorizar. En este caso, los parámetros se van incluyendo uno a uno, manteniendo los anteriores; finalizando por tanto con un entrenamiento en el que se tienen en cuenta todos los parámetros aleatorizados.
- Ideal a aleatorio: primero se entrena con una simulación ideal y después se vuelve a entrenar con todos los parámetros aleatorizados.
- Aleatorización del dominio: se entrena desde una simulación que contempla desde el principio todas las variables aleatorizadas.

En *"Analysis of Randomization Effects on Sim2Real Transfer in Reinforcement Learning for Robotic Manipulation Tasks"* [51] se realizó un estudio de estos distintos tipos de entrenamiento. De este mismo artículo se obtuvo los enfoques anteriores. El que obtuvo el mejor resultado fue el aprendizaje con aleatorización del dominio, seguido por *fine-tunning*. Debido a ser el mejor modo, y tener tiempo limitado dentro del entrenamiento, se escoge esta opción a seguir.

En el próximo apartado se estudiará como incluir esta herramienta en el aprendizaje, concretamente en el ejercicio de empuje (pues en el de alcance viene integrado en el ejemplo).

8.2. Aleatorización del dominio.

La aleatorización del dominio consiste en proveer a la simulación de una serie de variaciones en el entrenamiento para generalizar la política en el mundo real [57]. Para poner un ejemplo de esta implementación se introducirán para el ejemplo de empuje una variabilidad en algunas de sus observaciones y eventos.

Dentro de las observaciones, la clase `ObsTerm` permite introducir ruido a la observación mediante el parámetro del constructor `noise`. Esta ruido funcionará como la variabilidad del sistema. Para este, se utilizará la clase `GaussianNoiseCfg`. Esta clase recibe la siguiente serie de parámetros en su constructor [41]:

- `mean`: la media del ruido. Esto nos permite incluir un desfase al parámetro. En este caso, la media será 0, pues no contamos con ninguna desviación en los medidores.
- `std`: desviación estándar del ruido.
- `operation`: operación para incluir el ruido

Este tipo de ruido se incluirá a las observaciones de la velocidad, la posición del objeto y la posición de la herramienta.

Por otro lado, dentro del manejador de eventos tenemos distintas formas de introducir variabilidad al sistema. Esto se realiza mediante el parámetro `func`. En este ejemplo se han incluido dos funciones capaces de variar aspectos relevantes:

- `randomize_joint_parameters`: varía la posición de las articulaciones desde las cuales se empieza.
- `randomize_rigid_body_material`: varía los parámetros de los objetos físicos.
- `randomize_rigid_body_mass`: varía la masa de los objetos rígidos en escena.

Con esta variabilidad incluida se ha analizado como incluir aleatorización del dominio y preparado el sistema para una posible implementación. A continuación se estudiará como realizar una simulación externa a la herramienta IsaacLab.

8.3. Implementación en IsaacSim

A continuación se va estudiar la implementación del sistema en IsaacSim. Esta implementación es de desarrollo propio, empleando la clase `PolicyController` y `ConfigLoader` entre otras herramientas de IsaacSim [58]. El código se encuentra dentro del proyecto general, incluido en los anexos. A su vez, el código viene inspirado en los ejemplos de IsaacSim, implementando una estructura distinta ya que estos vienen en una extensión [58, `isaacsim.examples.interactive`].

El primer archivo relevante a esta implementación se encuentra en `source/ARMetaToolPG/ARMeta`. Este archivo declara una clase `RobohabilisPullObjectPolicy` que hereda de la clase `PolicyController`. Esta clase encarga de cargar y manejar la política. A continuación, se van a estudiar el objetivo de sus funciones, que serán a su vez sus responsabilidades:

- `__init__(...)`: se encarga de inicializar los principales atributos de la clase, como los objetos de la escena (entregados por el constructor) y el robot (creado y definido en la clase base); así como cargar la política, almacenada en este caso en `source/ARMetaToolPG/ARMetaToolPG/assets/policys/policy_pull_object_rh`. Este método se define en la clase específica
- `load_policy(...)`: se encarga de cargar la política. Es usado en el constructor para este fin. Este método queda definido en la clase base.
- `_compute_observations(...)`: se encarga de calcular las observaciones, almacenándolas en el parámetro `obs`. Este método se define en la clase específica.

- `_compute_action(...)`: se encarga de calcular las acciones utilizando la política. Este método queda definido dentro de la clase base.
- `forward(...)`: se encarga de aplicar las acciones al robot. Para ello se debe extraer la información de las acciones y preparar la indicación de la posición. Se debe recordar que en este caso se trabaja con dos tipos de acciones, binarias y de posición.
- `initialize(...)`: se encarga de inicializar el robot y los objetos en la escena.

Esta clase después se almacena como atributo de otra clase, `RoboHabilisTask`, la cual se define para agrupar los procesos de la tarea a ejecutar en IsaacSim. Esta clase se define en `scripts/sim2sim/load_robohabilis.py`, el archivo que se ejecutará dentro de IsaacSim mediante *Window → ScriptEditor*. Esta clase tiene las siguientes funciones:

- `__init__(...)`: se encarga de limpiar el mundo existente y crear uno nuevo. Caber resaltar que IsaacSim sigue la misma estructura que IsaacLab en los elementos de la simulación; estructura vista en el apartado 4.2.
- `set_up_scene(...)`: se encarga de definir los elementos de la simulación. El robot se define dentro de `RobohabilisPullObjectPolicy`, mientras que los objetos mediante la clase `RigidObject` [58, `isaacsim.core.experimental.prims`].
- `load_world_async(...)`: se encarga de cargar el mundo e inicializar las físicas de simulación. Por último, llama a la función `setup_post_load(...)`, que veremos a continuación.
- `setup_post_load(...)`: se encarga de cargar la llamada recurrente al método `on_physics_step()` así como inicializar los distintos elementos. Por último, llama al método del mundo `play_async(...)`, que mantiene el bucle a la llamada recurrente.
- `on_physics_steps`: se encarga de llamar a la función `forward()`, la cual avanza la simulación.

Dentro de este archivo también se puede encontrar la función que define el bucle asíncrono. Para ello, se usa la biblioteca `asyncio` de python [44]. La llamada a esta función permite ejecutar el programa dentro de IsaacSim sin bloquear sus procesos internos. Para la ejecución del código, se utiliza la función `load_robohabilis()`. Este método instancia la clase anterior, crea la escena con el método `set_up_scene()` y termina llamando a la función `load_world_async()`.

Con esto, se puede ejecutar una simulación externa al aprendizaje. En este trabajo, este script ha sido de gran utilidad. No solo para la evaluación de la política, dónde es un

paso clave en su implementación, sino también para explorar posiciones de robot, utilizando la clase `SingleArticulation` dentro del controlador. Por otro lado, la evaluación de la posible implementación de la política de empuje, pues permite tener un fácil acceso a las posiciones de la herramienta y el objeto. En el siguiente apartado, se estudiará la implementación en el robot real, utilizando un cliente python [56] y un robot UR3 como servidor, controlado desde el cliente python.

8.4. Implementación en robot real

Para la implementación en el robot real se van a utilizar dos lenguajes de programación. En primer lugar, utilizando python y la herramientas de Universal Robots de RTDE (Real-Time Data Exchange). Esta parte de la aplicación se encargará de cargar la política, recibir el estado del robot, calcular la acción siguiente y enviar dicha acción al robot. El robot se encarga de gestionar los movimientos del robot, regulando y sincronizando el proceso, preparar el robot para la ejecución del movimiento y aplicar las acciones que lo conforman.

En este apartado, se estudiará cada código por encima. En el cliente python se analizará el código en su conjunto mediante el diagrama de clases y se explicará como se maneja la comunicación con el robot real. Para el programa del robot se analizará el código al completo.

8.4.1. Cliente Python

El cliente python es el encargado de gestionar la política y enviar la información de las acciones al robot. Este cliente se conforma a través de un módulo de python desarrollado en este trabajo, bajo el nombre `sim2real`. Este módulo contiene varios ejercicios de implementación. En este apartado se estudiará uno de ellos, enfocado al ejercicio *Reach*. Para ello, se analizará el diagrama de clases creado y mostrado en la figura 8.1.

La pieza central de este diagrama es la clase `EnvironmentAdapter`. Esta clase se encarga de almacenar y gestionar el robot, la política y sus interacciones. Esta clase no es instanciable, sirve simplemente como base para los casos específicos. Esta clase generaliza parte de la construcción de las clases específicas, almacenando la política y la interfaz del robot; así como la gestión de las interacciones entre ambas, definidas en el método `step`. De esta clase hereda el adaptador específico para el caso del *Reach*, la clase `ReachUR3`.

La política queda almacenada en el atributo `controlador`, a través de una clase `PolicyController`. Esta clase se ha extraído de un ejemplo de referencia recomendado por el proyecto MetaTool, un ejemplo para el alcance de un robot *Kinova* [7]. A su vez,

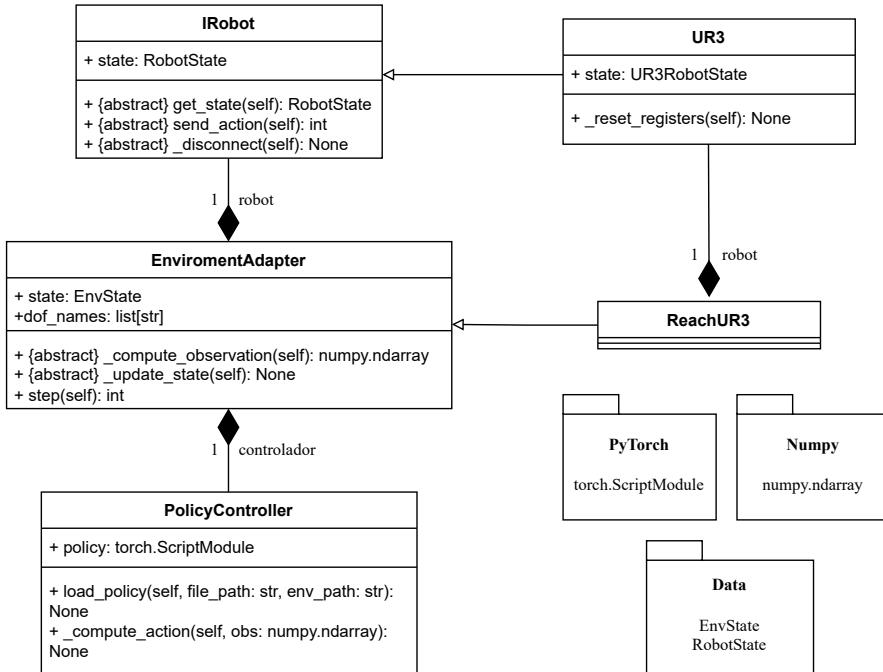


Figura 8.1: Diagrama de clases del módulo sim2real aplicado al ejercicio *Reach*.

este es un código adaptado de la clase `PolicyController` antes utilizada [58], extrayendo la creación del robot. En este módulo, la clase se vuelve a modificar, extrayendo el cálculo de observaciones que se realizará en la clase `EnvironmentAdapter`. El objetivo de esto es que esta clase funcione directamente como una red neuronal, a la que entregamos observaciones y devuelve acciones.

El robot, por otro lado, queda almacenado en el atributo `robot`, a través de la clase `IRobot`. Esta clase define una serie de métodos para el control de este. Por un lado, su constructor inherente, que se encarga de crear la conexión. Por otro lado, el método `get_state(self)`, encargado de recibir la información y devolver una clase de datos con esta. Otro método que se debe definir es `send_action(self)`, encargado de gestionar el envío de la acción al robot. Por último, se debe definir el método `_disconnect`, encargado de gestionar la desconexión del robot. De esta clase, hereda la clase específica para el UR3, la cual define cada uno de estos métodos, utilizando herramientas de UR-RTDE [56]. Estas herramientas permiten realizar una conexión con el robot e intercambiar información. La información intercambiada viene especificada en un fichero .xml. El cliente python recibe del robot la posición de las articulaciones (`actual_q`), la velocidad de las articulaciones (`actual_qd`) y un registro de salida (`output_int_register_0`). Por otro lado, el robot recibe información dentro de sus registros de entrada, la acción a realizar mediante las posiciones de las articulaciones objetivo y un `watchdog`, que junto con el registro de salida permiten sincronizar los dos procesos.

De este modo, este módulo cliente de python permite intercambiar información con el robot y gestionar las interacciones de este con la política. A continuación, se estudiará como se utiliza esta información para crear un bucle de control en el robot, utilizando URscript.

8.4.2. Servidor robot

El cliente python, como se ha visto, es el encargado de enviar las posiciones al robot. Esto lo hace escribiendo en sus registros. El robot por su parte, actúa como un servidor, recibiendo las posiciones en sus registros y gestionándolas, llevando al robot hacia estas. Esto se hace a través de un lenguaje de programación concreto, URscript [10]. Con este se crea un código que luego se incluye al programa del robot, ejecutándolo en un bucle continuo. A continuación se va a analizar por completo el código, que se muestra en la lista 8.1.

Listing 8.1: Código implementado en el programa del robot UR3e

```
v = 0.5
a = 0.5
pose_init = [1.57, -1.744, 1.57, 4.7102, 4.7102, 4.3602]
global ready = False
write_output_integer_register(0, 0)

def inicio():
    global joints = pose_init
    movej(joints, a, v)
    ready = True
    global flag = False
    write_output_integer_register(0, 1)
end

def updateActionRTDE():
    global action = [1.57, -1.744, 1.57, 4.7102, 4.7102, 4.3602]
    i=0
    q= get_actual_joint_positions()
    while i<6:
        action = read_input_float_register(i)
        i = i + 1
    end
    return action
end
```

```

# Definicion del programa principal.

def main_program():
    while ready == False or get_digital_out(2) == False or
        read_input_integer_register(0) == 0:
        if(ready == False):
            inicio()
        else:
            sync()
    end
end

while get_digital_out(2) == True and
    read_input_integer_register(0) == 1:
    sync()
    global flag = True
    joints = updateActionRTDE()
    movej(joints, v, a)
    pose = get_actual_tcp_pose()
    rpy=rotvec2rpy([pose[3], pose[4], pose[5]])
    global roll = rpy[0]
    global pitch = rpy[1]
    global yaw = rpy[2]
end
write_output_integer_register(0, 0)
sync()

main_program()

```

En primer lugar, se definen las distintas variables del programa: la velocidad y la aceleración, la posición inicial, un indicador del estado del robot; a su vez que se desactiva el registro de salida, para cerrar cualquier cliente activo. A continuación, se definen dos funciones para la ejecución del programa. Por un lado, la función de inicio, que lleva al robot a la posición inicial y activa el para que el registro de salida para que se pueda conectar el cliente. Por el otro, la función para la lectura de los registros, la cual devuelve un array con las posiciones indicadas en ellos. Definidas las variables y las funciones principales, se define la función principal que se ejecutará en bucle.

La función principal tiene dos bucles, cada uno de los cuales responde a una fase del funcionamiento. El primero de los bucles llama a la función de inicio, siempre que el robot no se encuentre listo. Una vez se encuentre listo, se sincroniza continuamente con el cliente hasta que se den las condiciones necesarias para continuar: robot listo, señal

digital del robot activada y *watchdog* activado. Cuando se cumplen estas condiciones, se sale del bucle y se entra en el segundo. Este segundo realiza la acción de control, llamando a la función de lectura de registros. Concretamente, se extrae la acción, se mueve el robot hacia la posición requerida por la red neuronal y se calcula el RPY del efecto final (*roll pitch yaw*). Este bucle se mantiene mientras el *watchdog* y la señal digital se mantengan activos.

De este modo, se ejerce un control sobre el despliegue de la política en el robot, pudiendo continuar con la realización de pruebas. En el siguiente apartado se estudiará como se realizan estas y los resultados obtenidos.

8.5. Ensayos

Los ensayos se realizarán sobre el ejercicio presentado reach. Se realizará en primer lugar una prueba simulada y después una prueba en el robot real. La prueba simulada se realizará dentro de un simulador proporcionado por Universal Robots, URSim [59]. Este simulador utiliza la misma interfaz de control que el robot real, PolyScope [60], y simula el movimiento de este. Esta misma interfaz es la que se utiliza para controlar el robot real, por lo que el código y la metodología son los mismos en ambos ensayos.

8.5.1. Prueba simulada

La prueba simulada es útil para verificar el comportamiento correcto del robot de forma segura antes de su implementación. El simulador utilizado es provisto por Universal Robots, adecuándose a las características reales del robot, teniendo en cuenta limitaciones en la fuerza, al no estar conectado a un brazo robótico real [59].

El ensayo se realiza en una serie de pasos. En primer lugar, se carga el script dentro del programa del robot, para después ejecutarlo. Este programa lleva al robot a la posición inicial y espera la referencia de las posiciones. En segundo lugar, se ejecuta el cliente python desde el mismo u otro ordenador en la misma red. Este cliente se conecta a la IP del robot y envía las referencias de las posiciones. Una vez el cliente está conectado y enviando posiciones, se activa la señal digital 2 para dar comienzo al movimiento. Por último, una vez el robot se detiene, se anotan los valores de la posición del efecto final. Estos valores se indican en la pestaña *Move*, en la cual hay que seleccionar que estén referenciados a la base. También se anotan los valores de *roll*, *pitch* y *yaw*, indicados en la sub-pestaña de variables, dentro de la pestaña *Program*. Finalizada la prueba, se desactiva la señal digital 2, desconectando el cliente y llevando el robot a su posición inicial.

Las pruebas realizadas en las simulación permitieron confirmar el correcto funcio-

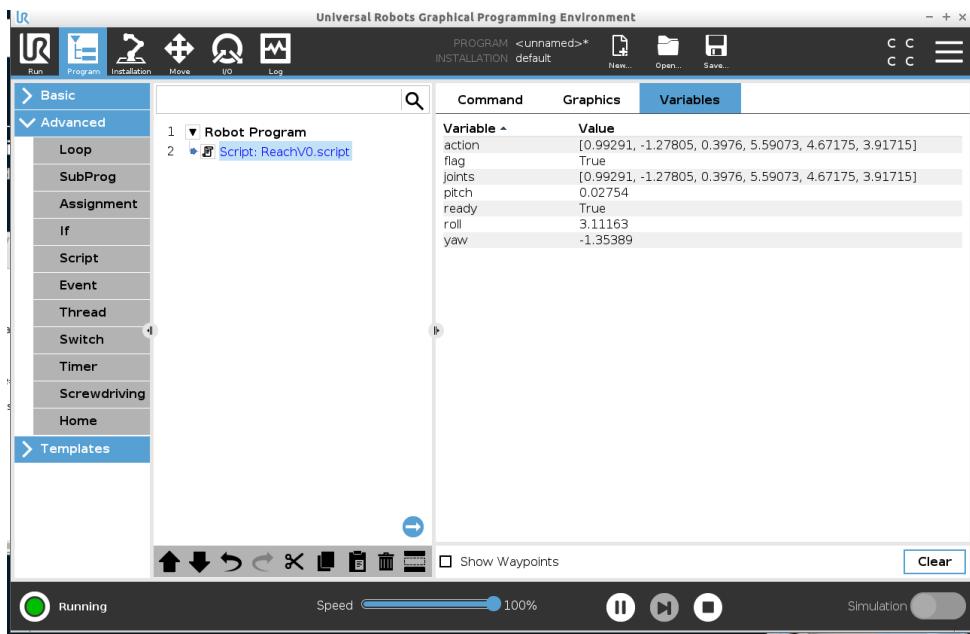


Figura 8.2: Pestaña *Program*, con la señal del

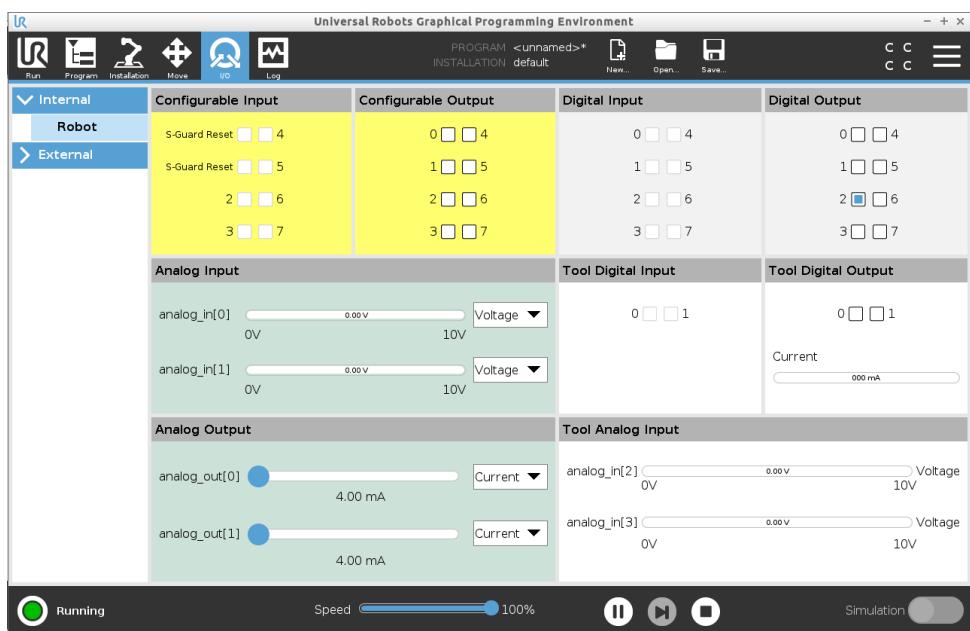


Figura 8.3: Pestaña *I/O* con señal digital 2 activa.

namiento de la implementación del Sim2Real. El robot mantenía un movimiento constante y sin variaciones, verificando la funcionalidad del programa cliente y del robot. No obstante, el error obtenido fue considerable.

Por un lado, al trabajar sin la herramienta definida dentro de la simulación se obtuvo un desplazamiento cercano a los 10 cm, aproximadamente el tamaño del gripper. Al mantener una rotación fija, esta diferencia solo afectó al eje Z, obteniendo en los otros ejes una desviación cercana a los 3 mm en el caso de X y 5 mm en el caso de Y. Por otro lado, la desviación obtenida en los ejes fue de 2,7 cm para el eje X, 3,2 cm para el Y y 1'7

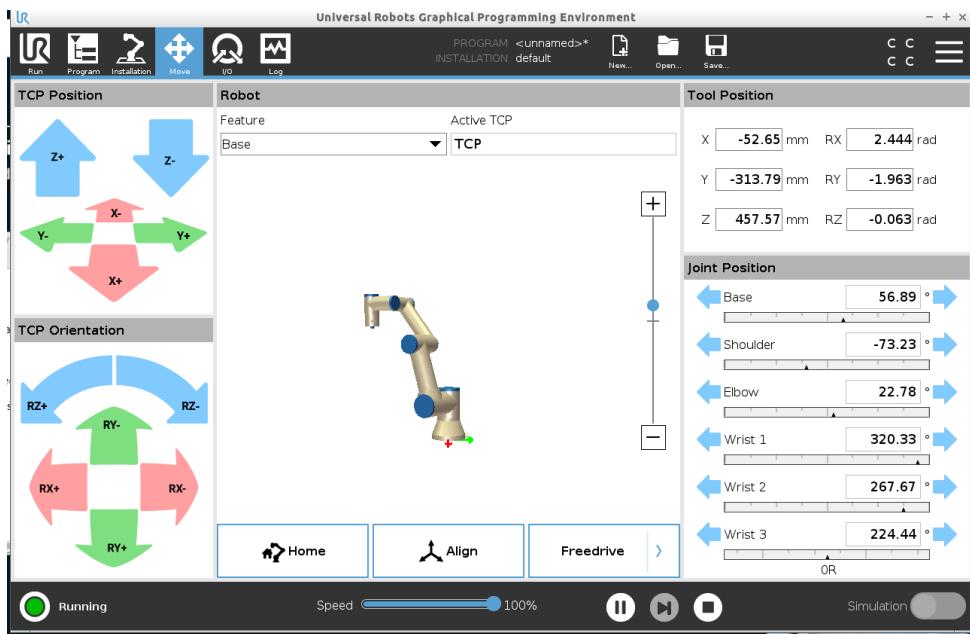


Figura 8.4: Pestaña Move en un ensayo implementado.

cm para el Z. Estos valores no indican una precisión suficiente para su implementación en tareas reales, pero demuestran el funcionamiento de las redes neuronales para el control del robot. Mediante una revisión de las recompensas, aumentando el valor de la precisión, así como la realización de entrenamientos más largos, se podría obtener una mayor precisión.

En relación con la orientación, las desviaciones obtenidas en los ángulos de rotación se sitúan en torno a 6° para el roll y 5° para el pitch y el yaw, lo que permite considerar la estimación como una aproximación inicial válida. En el caso del yaw se observa un desplazamiento constante cercano a los 135° , tal y como se muestra en la *****INCLUSIÓN DE FIGURA PENDIENTE*****, que no corresponde a un error del modelo sino a un offset sistemático. Este desfase se debe, por un lado, a las diferencias de convenio empleadas en la definición de los sistemas de referencia y, por otro, a un desfase adicional de aproximadamente 45° asociado a la orientación fija del eje de la herramienta. Una vez considerado este offset, la dispersión del error en yaw es comparable a la obtenida en los otros ejes de rotación.

La tabla con todos los datos recogidos se presenta en el anexo C, para facilitar la lectura y presentación de este documento.

Demostrado el correcto funcionamiento de la herramienta y la seguridad de su funcionamiento se trata en el siguiente capítulo con la implementación real en el robot.

8.5.2. Prueba real

Las pruebas realizadas en el entorno real presentaron resultados coherentes con los obtenidos en simulación. No obstante, durante su ejecución surgieron diversas dificultades técnicas y operativas que limitaron el alcance y la repetibilidad de los ensayos. En este apartado se describen los principales problemas encontrados durante la realización de las pruebas reales, las soluciones implementadas para mitigarlos y aquellos aspectos que no pudieron resolverse dentro del marco temporal y técnico del presente trabajo.

El primer problema en la implementación fue la normalización de ángulos. Las redes neuronales trabajadas utilizan rangos de ángulos normalizados, algo que no sucede en los robots reales. Para solventar el problema se introduce una normalización a la entrada y una función para calcular el ángulo más próximo. Por ejemplo, al recibir un angulo de 4.7124 radianes convierte este dato en -1.5708. Si la red neuronal calcula ir a la posición -1.9199 radianes, se envía la posición equivalente más cercana, en este caso 4.3633 radianes.

Listing 8.2: Código para la normalización del ángulo dentro del cliente python, dentro de la clase UR3Sim, en el método `get_state`

```
for i in range(len(robotstate.joint_position)):
    robotstate.joint_position[i] =(robotstate.
        joint_position_real[i]+math.pi)%(2*math.pi)-
        math.pi
```

Listing 8.3: Código para la búsqueda del ángulo próximo dentro del cliente python, dentro de la clase UR3Sim, en el método `send_action`

```
for i in range(len(joint_pos)):
    diff= (joint_pos[i] - robotstate.
        joint_position_real[i] + math.pi) % (2*math.pi)
        - math.pi
    joint_pos[i]=robotstate.joint_position_real[i]+
        diff
```

Con esta corrección de los ángulos implementada se realizaron primeras pruebas, en los que aparecieron errores referentes a las articulaciones del robot. Con el objetivo de identificar la posible causa de los errores se redujo la velocidad del robot. La función implementada para el movimiento, `servoj`, permite enviar una posición objetivo al robot hacia la que debe moverse durante cierto tiempo [10]. Los parámetros de esta función no permiten realizar un control preciso de la velocidad sin afectar a la calidad del movimiento. Por esto, se implementó una interpolación para acortar la distancia indicada al robot a través de una interpolación.

Listing 8.4: Código para la reducción de la distancia mediante interpolación, dentro de la clase `UR3Sim`, en el método `send_action`

```

        step=np.zeros_like(joint_pos)
        for i in range(len(joint_pos)):
            delta_q=joint_pos[i]- robotstate.
                joint_position_real[i]
            max_step= self.max_step_d*self.dt
            step[i]=np.clip(delta_q, -max_step, max_step)
            joint_pos[i]=robotstate.joint_position_real[i]+
                step[i]
    
```

Gracias a esta función se pudieron realizar pruebas con el robot real. Mediante esta función, con un salto de máximo de 1 rad/s aplicando una frecuencia de 125Hz, se realizaron varias 5 pruebas exitosas. Sin embargo, la velocidad se vuelve demasiado baja para que la implementación sea funcional. No obstante, en dos de estas pruebas se obtuvieron valores aproximados a la simulación, con un margen de error en el orden de los milímetros, verificando de este modo los resultados obtenidos en la simulación.

Los resultados obtenidos presentan un margen de mejora significativo. Por un lado, un análisis más detallado del sistema de control del UR3e permitiría esclarecer la necesidad de las limitaciones de velocidad impuestas durante los ensayos. Asimismo, en pruebas preliminares con otros métodos de control se detectó la presencia de ruido en las posiciones entregadas al robot, provocando vibraciones durante su movimiento. Este fenómeno podría estar relacionado con los fallos observados en las articulaciones, por lo que resulta de interés un estudio futuro sobre técnicas de filtrado, incluyendo su integración en redes neuronales.

No obstante, los resultados obtenidos demuestran la funcionalidad del enfoque de aprendizaje por refuerzo, así como la correcta implementación de la política de control. Una vez abordados los problemas asociados al control del robot, sería posible ampliar el alcance del sistema hacia ejercicios más complejos. Cabe destacar que el uso de redes neuronales resulta especialmente adecuado para tareas de mayor complejidad, manteniéndose inalterado el procedimiento de implementación de la política.

En este contexto, tareas como el empuje con herramienta se presentan como una extensión natural del trabajo realizado. De hecho, dentro del módulo cliente Sim2Real se ha preparado una variante orientada a este caso; sin embargo, debido a dificultades surgidas durante el entrenamiento y a las limitaciones observadas en el sistema de control, no fue posible su implementación en el marco del presente trabajo.

Realizadas estas pruebas, se presentan en el siguiente capítulo las conclusiones finales del proyecto.

Capítulo 9

Conclusiones Finales

El trabajo realizado presenta el marco teórico del aprendizaje por refuerzo y su implementación mediante la herramienta IsaacLab, todo ello enfocado a la robótica. El aprendizaje por refuerzo es una de las tecnologías punteras en el control de robots. Este trabajo es una base para la implementación de esta disciplina dentro de la escuela, siendo relevante para futuros trabajos. En este capítulo se analizará el aporte de este trabajo y los resultados obtenidos.

9.1. Resultados

Al comienzo del trabajo se presentaron una serie de objetivos: dos objetivos principales y ocho secundarios. En esta sección se analizará el cumplimiento de estos objetivos, comenzando por los objetivos secundarios, para después presentar los principales.

- Objetivo 1: Se realizó un estudio del arte. En este, se presentan la relación del aprendizaje por refuerzo con la robótica y distintos artículos que muestran aplicaciones prácticas en distintas disciplinas.
- Objetivo 2: Se realizó una extensa base teórica del aprendizaje por refuerzo. En ella, se presentó la definición y la estructura del aprendizaje, su base matemática junto con los MDPs y los distintos algoritmos aplicables. Estos se presentan de una forma didáctica, con el objetivo de usar la documentación como material académico.
- Objetivo 3: Se realizó un análisis profundo de la herramienta IsaacLab. Se presentó la base de la herramienta, abarcando su estructura, las distintas formas de programación, las estructuras de datos y los principales procesos de entrenamiento y evaluación.

- Objetivo 4: Se realizaron dos análisis sobre ejercicios prácticos, abarcando ambas formas de programación e implementando mejoras en ambos.
- Objetivo 5: Se trabajó dentro del proyecto MetaTool y se presento su misión y objetivos.
- Objetivo 6: Se presentaron algunos de los ejercicios prácticos trabajados y las correcciones implementadas.
- Objetivo 7: Se estudió el problema Sim2Real y se presentaron distintas soluciones, implementando finalmente la aleatorización del dominio.
- Objetivo 8: Se diseño un código para la implementación de políticas, obteniendo resultados aptos en simulación y limitados en la realidad.

Teniendo en cuenta estos objetivos secundarios, se presenta una evaluación de los objetivos principales. Por un lado, la creación de una base teórica a quedado cubierta en este trabajo, mediante los objetivos 1, 2, 3 y 4. El resultado en su conjunto es bueno, habiendo construido una base teórica sólida para futuros trabajos. Por otro lado, la implicación dentro del proyecto MetaTool ha sido limitada, debido a su alta complejidad. Sin embargo, tomando los objetivos 5, 6, 7 y 8, se ha dado cierta involucración que, aunque no haya tenido un efecto en el proyecto general, ha sido beneficiosa para los conocimientos y contenidos aportados. Por tanto, se han cumplido los objetivos con éxitos, asumiendo ciertas limitaciones en la segunda parte del trabajo.

Capítulo 10

Aportaciones del trabajo

Añadido a los objetivos de este trabajo, este trabajo contiene más aportaciones dignas de resaltar y que complementan estos objetivos. En esta sección se presentarán, dividiendo de nuevo el trabajo en dos partes marcadas: teoría (objetivos 1 a 4) y práctica (objetivos 5 a 8).

La parte teórica, por un lado, no es únicamente una presentación de conceptos. El capítulo 3 sigue una progresión propia de este trabajo. Esta permite seguir los conceptos progresivamente y manteniendo una cohesión. Si bien esta estructura está influenciada por los principales materiales utilizados [5] [6], no responde a ninguna de estos dos, sino se vuelve una combinación de ambos y de la propia experiencia de aprendizaje. Además, la mayor parte de este material se encuentra en lengua inglesa, por lo que la traducción al español se vuelve valiosa. El capítulo 4, por otro lado, realiza una labor parecida, agrupando toda la información contenida dentro de la documentación oficial [39]. La información contenida en este capítulo se reparte en distintos apartados. En este capítulo se agrupa toda esta información, teniendo en cuenta una aplicación didáctica, al igual que en el apartado 3. Por último, los análisis profundos de cada ejercicio práctico son completamente propios de este trabajo, derivados de multiples horas de análisis e investigación dentro de las APIs de la herramienta [41]. Estos análisis permiten facilitar la compresión del conjunto global de entrenamiento, que supone la gran parte del trabajo realizado y permite acortar el tiempo dedicado a esto en futuros proyectos.

La parte práctica, por otro lado, se encuentra más limitada, pero no carece en absoluto de aportaciones. La principal de estas son los distintos módulos implementados para la implementación de políticas. Los módulos existentes actualmente son extremadamente limitados a aplicaciones concretas. El módulo implementado permite una gran flexibilidad a la hora de diseñar entornos, adaptándose a la modularidad de IsaacLab. A su vez, los resultados obtenidos en la práctica, a pesar de ser deficientes, son una de las grandes aportaciones en este trabajo. En la mayoría de artículos expuestos se indican

únicamente casos de éxito. Siendo el primer trabajo referente a esta disciplina dentro de la universidad estos resultados sirven de referencia para futuras aplicaciones.

10.1. Limitaciones

Pese a las aportaciones expuestas, se han encontrado ciertas limitaciones. Por un lado, en la parte teórica, la principal limitación de este trabajo es en el construcción de algoritmos propios. Esta limitación no es derivada del trabajo sino del enfoque. La construcción de algoritmos parte de la base teórica expuesta, pero requiere de una mayor investigación acerca de redes neuronales y algoritmos modernos de aprendizaje profundo. En este trabajo se han expuesto las bases de estos, pero realizar un estudio extenso requiere de sentido gracias a las bibliotecas de algoritmos contenidas dentro IsaacLab. La limitación en la parte práctica, sin embargo, es mayor. Los problemas encontrados en el control del robot limitaron la implementación del ejercicio *Reach*, así como la implementación de ejercicios más complicados dentro de MetaTool, implementados sobre el *Robohabilis*.

Las limitaciones del trabajo no suponen un fracaso para la creación de la base teórica realizada. Esta base permite planear futuros proyectos, que expandan sobre esta base.

10.2. Trabajos futuros

La base expuesta y las limitaciones encontradas abren camino para futuros trabajos que expandan sobre el desarrollo de este. En esta sección se proponen posibles enfoques para nuevos trabajos:

- **Implementación de políticas en UR3.** La disponibilidad de estos robots en el laboratorio de la ETSIDI permiten realizar gran número de pruebas en ellos. Sin embargo, los distintos problemas encontrados en el control de estos mediante políticas ha sido el principal limitador de este trabajo. Se propone un trabajo en el que se utilicen directamente las políticas generadas en este u otro trabajo pare enfocarse en su implementación; y no en la generación como en este caso.
- **Generación de políticas con otras herramientas.** IsaacLab no es la única herramienta de aprendizaje automático, existen otras aplicaciones y herramientas que disponen de estas funcionalidades. IsaacLab es la más potente, pero su uso de las tarjetas gráficas es muy limitante. La base teórica del aprendizaje por refuerzo, el módulo de implementación de políticas y las exemplificaciones en IsaacLab son extrapolables a cualquier caso de aprendizaje.

- **Aplicación de políticas en robots.** Este trabajo, combinado a los anteriores expuestos, permitirían tener la base suficiente para llevar a cabo proyectos enfocados únicamente a la aplicación de las políticas en robos reales. Este tipo de proyectos debería dividirse en dos trabajos separados. Por un lado, la generación de los elementos a simular y sus políticas; y por el otro, la implementación Sim2Real en el robot real.
- **Diseño de nuevos algoritmos.** Este tipo de proyectos divergen del terreno de la escuela ETSIDI, estando más relacionados con programación que con implementaciones ingenieriles. No obstante, la base teórica presentada supone un gran punto de inicio para cualquier trabajo de estas características.

El objetivo de próximos trabajos debería ser la consolidación de resultados reales, algo en lo que en este trabajo no se ha podido alcanzar. Para ello se realiza una recomendación: utilizar este trabajo como base teórica y enfocar los esfuerzos en un único problema. Este trabajo, debido a su amplitud teórica, se enfoca en múltiples problemas en un solo trabajo. Esto requiere gran cantidad de tiempo, pues cada ejemplo práctico o ejercicio práctico tienen una serie de problemas individuales. Esta multitud de problemas individuales consumen gran cantidad de horas, por lo que es prioritario consolidar los resultados una vez superados.

Bibliografía

- [1] What is Reinforcement Learning?
- [2] Yuxi Li. Reinforcement Learning Applications, August 2019. arXiv:1908.06973 [cs].
- [3] Starting on the Right Foot with Reinforcement Learning.
- [4] About Us – ROMERIN.
- [5] David Silver. Lectures on Reinforcement Learning.
- [6] Richard S. Sutton and Andrew Barto. *Reinforcement learning: an introduction*. Adaptive computation and machine learning. The MIT Press, Cambridge, Massachusetts London, England, second edition edition, 2020.
- [7] Louis Le Lay. Kinova gen3 rl & sim2real toolkit, July 2025.
- [8] MetaTool Consortium. Metatool. Software framework, 2024.
- [9] Robot A/Ss. UR3e.
- [10] Universal Robots. *URScript Programming Language*. Universal Robots A/S, 2023. Lenguaje de programación para robots colaborativos UR.
- [11] Sridhar Mahadevan and Jonathan Connell. Automatic programming of behavior-based robots using reinforcement learning. *Artificial Intelligence*, 55(2-3):311–365, June 1992.
- [12] Vincent François-Lavet, Peter Henderson, Riashat Islam, Marc G. Bellemare, and Joelle Pineau. An Introduction to Deep Reinforcement Learning. *Foundations and Trends® in Machine Learning*, 11(3-4):219–354, December 2018. Publisher: Now Publishers, Inc.
- [13] Chen Tang, Ben Abbatematteo, Jiaheng Hu, Rohan Chandra, Roberto Martín-Martín, and Peter Stone. Deep Reinforcement Learning for Robotics: A Survey of Real-World Successes, September 2024. arXiv:2408.03539 [cs].

- [14] Matthew T. Mason. Toward Robotic Manipulation. *Annual Review of Control, Robotics, and Autonomous Systems*, 1(1):1–28, May 2018.
- [15] Dmitry Kalashnikov, Alex Irpan, Peter Pastor, Julian Ibarz, Alexander Herzog, Eric Jang, Deirdre Quillen, Ethan Holly, Mrinal Kalakrishnan, Vincent Vanhoucke, and Sergey Levine. QT-Opt: Scalable Deep Reinforcement Learning for Vision-Based Robotic Manipulation, November 2018. arXiv:1806.10293 [cs].
- [16] Lei Zhang, Soumya Mondal, Zhenshan Bing, Kaixin Bai, Diwen Zheng, Zhaopeng Chen, Alois Christian Knoll, and Jianwei Zhang. DORA: Object Affordance-Guided Reinforcement Learning for Dexterous Robotic Manipulation, May 2025. arXiv:2505.14819 [cs].
- [17] Covariant | About.
- [18] Introducing RFM-1: Giving robots human-like reasoning capabilities.
- [19] Ashish Kumar, Zipeng Fu, Deepak Pathak, and Jitendra Malik. RMA: Rapid Motor Adaptation for Legged Robots, July 2021. arXiv:2107.04034 [cs].
- [20] Zhongyu Li, Xue Bin Peng, Pieter Abbeel, Sergey Levine, Glen Berseth, and Koushil Sreenath. Reinforcement Learning for Versatile, Dynamic, and Robust Bipedal Locomotion Control, August 2024. arXiv:2401.16889 [cs].
- [21] IEEE Standard Definitions of Navigation Aid Terms. *IEEE Std 172-1983*, pages 1–34, June 1983.
- [22] Ibrahim Khalil Kabir and Muhammad Faizan Mysorewala. Socially aware navigation for mobile robots: a survey on deep reinforcement learning approaches, November 2025. arXiv:2512.00049 [cs].
- [23] Syed Agha Hassnain Mohsan, Nawaf Qasem Hamood Othman, Yanlong Li, Mohammed H. Alsharif, and Muhammad Asghar Khan. Unmanned aerial vehicles (UAVs): practical aspects, applications, open challenges, security issues, and future trends. *Intelligent Service Robotics*, 16(1):109–137, 2023.
- [24] Ahmad Reza Cheraghi, Sahdia Shahzad, and Kalman Graffi. Past, Present, and Future of Swarm Robotics, January 2021. arXiv:2101.00671 [cs].
- [25] Raúl Arranz, David Carramiñana, Gonzalo de Miguel, Juan A. Besada, and Ana M. Bernardos. Application of Deep Reinforcement Learning to UAV Swarming for Ground Surveillance. *Sensors*, 23(21):8766, October 2023. arXiv:2501.08655 [cs].

- [26] Kento Kawaharazuka, Jihoon Oh, Jun Yamada, Ingmar Posner, and Yuke Zhu. Vision-Language-Action Models for Robotics: A Review Towards Real-World Applications. *IEEE Access*, 13:162467–162504, 2025. arXiv:2510.07077 [cs].
- [27] Haoyuan Deng, Zhenyu Wu, Haichao Liu, Wenkai Guo, Yuquan Xue, Ziyu Shan, Chuanrui Zhang, Bofang Jia, Yuan Ling, Guanxing Lu, and Ziwei Wang. A Survey on Reinforcement Learning of Vision-Language-Action Models for Robotic Manipulation.
- [28] Tom Mitchell and Hill McGraw. *Machine Learning textbook*. McGraw-Hill Science/Engineering/Math, 1997.
- [29] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.
- [30] Stuart J. Russell and Peter Norvig. *Inteligencia artificial: un enfoque moderno*. Pearson Educación, 2004. Traducción al español de *Artificial Intelligence: A Modern Approach*.
- [31] B. F. Skinner. *The behavior of organisms: an experimental analysis*. The behavior of organisms: an experimental analysis. Appleton-Century, Oxford, England, 1938. Pages: 457.
- [32] Leslie Pack Kaelbling, Michael L. Littman, and Anthony R. Cassandra. Planning and acting in partially observable stochastic domains. *Artificial Intelligence*, 101(1–2):99–134, 1998.
- [33] Nicholas Metropolis and S. Ulam. The Monte Carlo Method. *Journal of the American Statistical Association*, 44(247):335–341, September 1949. Publisher: ASA Website _eprint: <https://www.tandfonline.com/doi/pdf/10.1080/01621459.1949.10483310>.
- [34] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, February 2015. Publisher: Nature Publishing Group.
- [35] Marc G. Bellemare, Will Dabney, and Mark Rowland. *Distributional Reinforcement Learning*. The MIT Press, May 2023.
- [36] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous Methods for Deep Reinforcement Learning, June 2016. arXiv:1602.01783 [cs].

- [37] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal Policy Optimization Algorithms, August 2017. arXiv:1707.06347 [cs].
- [38] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor, August 2018. arXiv:1801.01290 [cs].
- [39] NVIDIA Corporation. Isaac Lab Documentation.
- [40] Mayank Mittal, Pascal Roth, James Tigue, Antoine Richard, Octi Zhang, Peter Du, Antonio Serrano-Muñoz, Xinjie Yao, René Zurbrügg, Nikita Rudin, Lukasz Wawrzyniak, Milad Rakhsha, Alain Denzler, Eric Heiden, Ales Borovicka, Ossama Ahmed, Iretiayo Akinola, Abrar Anwar, Mark T. Carlson, Ji Yuan Feng, Animesh Garg, Renato Gasoto, Lionel Gulich, Yijie Guo, M. Gussert, Alex Hansen, Mihir Kulkarni, Chenran Li, Wei Liu, Viktor Makovychuk, Grzegorz Malczyk, Hammad Mazhar, Masoud Moghani, Adithyavairavan Murali, Michael Noseworthy, Alexander Poddubny, Nathan Ratliff, Welf Rehberg, Clemens Schwarke, Ritvik Singh, James Latham Smith, Bingjie Tang, Ruchik Thaker, Matthew Trepte, Karl Van Wyk, Fangzhou Yu, Alex Millane, Vikram Ramasamy, Remo Steiner, Sangeeta Subramanian, Clemens Volk, CY Chen, Neel Jawale, Ashwin Varghese Kuruttukulam, Michael A. Lin, Ajay Mandlekar, Karsten Patzwaldt, John Welsh, Huihua Zhao, Fatima Anes, Jean-Francois Lafleche, Nicolas Moënne-Loccoz, Soowan Park, Rob Stepinski, Dirk Van Gelder, Chris Amevor, Jan Carius, Jumyung Chang, Anka He Chen, Pablo de Heras Ciechomski, Gilles Daviet, Mohammad Mohajerani, Julia von Murralt, Viktor Reutskyy, Michael Sauter, Simon Schirm, Eric L. Shi, Pierre Terdiman, Kenny Vilella, Tobias Widmer, Gordon Yeoman, Tiffany Chen, Sergey Grizan, Cathy Li, Lotus Li, Connor Smith, Rafael Wiltz, Kostas Alexis, Yan Chang, David Chu, Linxi "Jim" Fan, Farbod Farshidian, Ankur Handa, Spencer Huang, Marco Hutter, Yashraj Narang, Soha Pouya, Shiwei Sheng, Yuke Zhu, Miles Macklin, Adam Moraevanszky, Philipp Reist, Yunrong Guo, David Hoeller, and Gavriel State. Isaac lab: A gpu-accelerated simulation framework for multi-modal robot learning. *arXiv preprint arXiv:2511.04831*, 2025.
- [41] NVIDIA Corporation. API Reference — Isaac Lab Documentation.
- [42] NVIDIA Corporation. Isaac Sim Documentation.
- [43] NVIDIA Corporation. OpenUSD Fundamentals — Isaac Sim Documentation.
- [44] Python Software Foundation. Python documentation, 2024.
- [45] PyTorch Team. Pytorch documentation, 2025.

- [46] Antonio Serrano-Muñoz, Dimitrios Chrysostomou, Simon Bøgh, and Nestor Arana-Arexolaleiba. skrl: Modular and flexible library for reinforcement learning. *Journal of Machine Learning Research*, 24(254):1–9, 2023.
- [47] Clemens Schwarke, Mayank Mittal, Nikita Rudin, David Hoeller, and Marco Hutter. Rsl-rl: A learning library for robotics research. *arXiv preprint arXiv:2509.10771*, 2025.
- [48] Denys Makoviichuk and Viktor Makoviychuk. rl-games: A high-performance framework for reinforcement learning, May 2021.
- [49] Antonin Raffin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus, and Noah Dormann. Stable-baselines3: Reliable reinforcement learning implementations. *Journal of Machine Learning Research*, 22(268):1–8, 2021.
- [50] Mark Towers, Ariel Kwiatkowski, Jordan Terry, John U Balis, Gianluca De Cola, Tristan Deleu, Manuel Goulão, Andreas Kallinteris, Markus Krimmel, Arjun KG, et al. Gymnasium: A standard interface for reinforcement learning environments. *arXiv preprint arXiv:2407.17032*, 2024.
- [51] Josip Josifovski, Mohammadhossein Malmir, Noah Klarmann, Bare Luka Žagar, Nicolás Navarro-Guerrero, and Alois Knoll. Analysis of Randomization Effects on Sim2Real Transfer in Reinforcement Learning for Robotic Manipulation Tasks, October 2022. *arXiv:2206.06282 [cs]*.
- [52] UPM CAR, CSIC. CAR.
- [53] Damien S. Fleur, Bert Bredeweg, and Wouter van den Bos. Metacognition: ideas and insights from neuro- and educational sciences. *NPJ Science of Learning*, 6:13, June 2021.
- [54] RTVE es (Texto) / MARC CAMPDELACREU y MARTA ANTÓN (Vídeo). La inteligencia artificial da un salto evolutivo con robots que crean sus propias herramientas, April 2025. Section: NOTICIAS.
- [55] Longchao Da, Justin Turnau, Thirulogasankar Pranav Kutralingam, Alvaro Velasquez, Paulo Shakarian, and Hua Wei. A Survey of Sim-to-Real Methods in RL: Progress, Prospects and Challenges with Foundation Models, March 2025. *arXiv:2502.13187 [cs]*.
- [56] UniversalRobots/RTDE_python_client_library, December 2025. original-date: 2022-01-27T14:25:21Z.

- [57] Josh Tobin, Rachel Fong, Alex Ray, Jonas Schneider, Wojciech Zaremba, and Pieter Abbeel. Domain Randomization for Transferring Deep Neural Networks from Simulation to the Real World, March 2017. arXiv:1703.06907 [cs].
- [58] NVIDIA Corporation. NVIDIA Isaac Sim, 2024. Robotics simulation framework based on Omniverse.
- [59] Universal Robots A/S. URSim: Offline Simulator for Universal Robots, 2025. Simulador offline del entorno PolyScope.
- [60] Universal Robots A/S. PolyScope Simulator, 2025. Software de simulación para robots colaborativos UR.

Anexo A. Repositorios de código

El código desarrollado en el presente trabajo se encuentra disponible en los siguientes repositorios de GitHub:

- **Repositorio 1 - Proyecto General de IsaacLab:** Proyecto general con todos los códigos del trabajo relevantes a la herramienta IsaacLab y la aplicación IsaacSim.
<https://github.com/EdeAntonio/ARMetaToolPG>
- **Repositorio 2 - Proyecto Sim2Real:** Proyecto con el módulo para la implementación de las políticas generadas.
<https://github.com/EdeAntonio/Sim2RealEdAS>

Estos repositorios contienen el código empleado para el entrenamiento de la red neuronal, así como las herramientas utilizadas para la evaluación experimental.

Anexo B. Vídeos de resultados experimentales

Debido a las limitaciones del formato escrito, los resultados experimentales se complementan con una serie de vídeos que muestran el comportamiento del sistema durante los ensayos de alcance.

Los vídeos se encuentran disponibles en la siguiente carpeta de Google Drive:

https://drive.google.com/drive/folders/19nxncHIfXZ1wsK8j33GQqheYxs2wF4jY?usp=drive_link

En dichos vídeos se pueden observar tanto los casos representativos como situaciones límite del método propuesto.

Anexo C. Resultados completos de los ensayos

La figura 10.2 recoge el conjunto completo de resultados obtenidos en los ensayos de alcance realizados. Con el objetivo de preservar la legibilidad y el formato visual, la tabla se incluye como imagen en orientación horizontal.

Nº Prueba	Valores Objetivo						Prueba simulada						Corrección Convenio		Corrección TCP	
	X (m)	Y (m)	Z (m)	Roll (rad)	Pitch (rad)	Yaw (rad)	X (rad)	Y (rad)	Z (rad)	Roll (rad)	Pitch (rad)	Yaw (rad)	x_corr (m)	y_corr (m)	z_off (m)	
1,000	0,120	0,320	0,180	3,142	0,000	-1,200	-0,088	-0,308	0,267	-3,050	-0,118	2,689	0,088	0,308	0,161	
2,000	-0,050	0,250	0,300	3,142	0,000	0,850	0,063	-0,276	0,423	3,060	0,110	-1,514	-0,063	0,276	0,317	
3,000	-0,220	0,380	0,220	3,142	0,000	-0,450	0,197	-0,364	0,319	-2,900	0,113	-2,800	-0,197	0,364	0,213	
4,000	0,050	0,290	0,340	3,142	0,000	1,100	-0,052	-0,313	0,457	3,110	0,027	-1,350	0,052	0,313	0,351	
5,000	-0,180	0,210	0,160	3,142	0,000	-0,900	0,133	-0,253	0,296	2,990	0,021	-3,060	-0,133	0,253	0,190	
6,000	0,190	0,350	0,280	3,142	0,000	0,300	-0,232	-0,321	0,384	-3,120	0,000	-1,950	0,232	0,321	0,278	
7,000	-0,100	0,270	0,200	3,142	0,000	-1,450	0,076	-0,297	0,320	3,130	0,029	2,640	-0,076	0,297	0,214	
8,000	-0,270	0,230	0,330	3,142	0,000	0,600	0,235	-0,221	0,415	-3,010	0,070	-1,760	-0,235	0,221	0,309	
9,000	0,080	0,400	0,250	3,142	0,000	-0,150	-0,056	-0,370	0,335	-3,130	-0,130	-2,550	0,056	0,370	0,229	
10,000	0,000	0,310	0,190	3,142	0,000	1,450	0,033	-0,394	0,322	-3,120	0,147	-0,990	-0,033	0,394	0,216	
11,000	-0,140	0,340	0,270	3,142	0,000	-0,700	0,134	-0,328	0,373	-2,960	0,078	-3,084	-0,134	0,328	0,267	
12,000	-0,160	0,220	0,310	3,142	0,000	1,200	0,178	-0,216	0,407	3,050	0,068	-1,190	-0,178	0,216	0,301	
13,000	0,030	0,370	0,230	3,142	0,000	0,400	-0,028	-0,372	0,336	3,080	0,026	-1,910	0,028	0,372	0,230	
14,000	-0,080	0,260	0,350	3,142	0,000	-0,250	0,092	-0,234	0,437	-3,070	-0,082	-2,630	-0,092	0,234	0,331	

Errores							
x_err (m)	y_err (m)	z_err (m)	z_err_2 (m)	r_err (rad)	p_err (rad)	ya_err (rad)	
-0,032	-0,012	0,087	-0,019	0,092	-0,118	-2,394	
-0,013	0,026	0,123	0,017	-0,082	0,110	-2,364	
0,023	-0,016	0,099	-0,007	0,242	0,113	-2,350	
0,002	0,023	0,117	0,011	-0,032	0,027	-2,450	
0,047	0,043	0,136	0,030	-0,152	0,021	-2,160	
0,042	-0,029	0,104	-0,002	0,022	0,000	-2,250	
0,024	0,027	0,120	0,014	-0,012	0,029	-2,193	
0,035	-0,009	0,085	-0,021	0,132	0,070	-2,360	
-0,024	-0,030	0,085	-0,021	0,012	-0,130	-2,400	
-0,033	0,084	0,132	0,026	0,022	0,147	-2,440	
0,006	-0,012	0,103	-0,003	0,182	0,078	-2,384	
-0,018	-0,004	0,097	-0,009	-0,092	0,068	-2,390	
-0,002	0,002	0,106	0,000	-0,062	0,026	-2,310	
-0,012	-0,026	0,087	-0,019	0,072	-0,082	-2,380	

	X (m)	Y (m)	Z (m)	Roll (rad)	Pitch (rad)	Yaw (rad)
Error Medio	0,003	0,005	0,106	0,025	0,026	-2,345
Des. Típica	0,027	0,032	0,017	0,110	0,085	0,087

Figura 1: Resultados completos de los ensayos de alcance.