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

Capítulo 1

Introducción

1.1. Contexto y motivación

Según la página oficial de *Nvidia* [1], 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 va a estudiar esta doctrina para entender sus conceptos fundamentales y poder crear (mediante la herramienta de *Nvidia*, *IsaacLab*) distintos entornos capaces de ejecutar este procedimiento.

Las inteligencias artificiales son 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; como las finanzas, en la reserva de órdenes de compra; o como la energía, en sistemas de refregamiento de bancos de datos [2]. En la robótica especialmente, ha tomado un gran protagonismo. En esta disciplina, grandes entidades como *Boston Dynamics* han empezado a implementar esta técnica en múltiples tareas [3].

El interés en este proyecto nace de la idea de aplicar esta herramienta dentro del proyecto *ROMERIN*, un robot modular escalador para la inspección de infraestructuras [4]. Debido a la complejidad del aprendizaje por refuerzo, se vio la necesidad de realizar un estudio completo. Esto, añadido a la cesión de recursos del proyecto *MetaTool* en la formación, llevo a colaborar dentro de este proyecto, analizando y revisando código.

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

1.2. Objetivos del trabajo

Este proyecto busca realizar un estudio del aprendizaje por refuerzo y la herramienta IsaacLab, para lo que se fijan dos objetivos principales. En primer lugar, asentar una base teórica fuerte tanto del aprendizaje por refuerzo como la herramienta IsaacLab. Se pretende que futuros estudiantes puedan basarse en ella para realizar trabajos en esta disciplina. En segundo lugar, realizar labores dentro del proyecto europeo MetaTool; utilizando estas para ganar experiencia.

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

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, base matemática y sus principales algoritmos.
3. Explicar el funcionamiento de la herramienta *IsaacLab* para su aplicación en aprendizaje por refuerzo.
4. Analizar ejemplos de dicha herramienta, para así profundizar en ella y proveer de una guía práctica para trabajos futuros.
5. Estudio del proyecto *MetaTool*: Misión y Visión.
6. Realización de trabajos prácticos en el proyecto con la herramienta *IsaacLab*
7. Estudio del problema *Sim2Real* y posibles soluciones
8. Realización de un código para la implementación de políticas.

Para el cumplimiento de estos objetivos, se deberá tener en cuenta todo lo se va abarcar; y cómo este alcance se adapta a los objetivos propuestos. En el siguiente apartado, se realizará esto mismo.

1.3. Alcance y limitaciones

En este TFG cubriremos el proceso para realizar entornos de aprendizaje automático. Al tener este objetivo en mente, en este TFG se podrán encontrar distintos aspectos de esta disciplina. Este trabajo contempla desde los aspectos más fundamentales de la teoría del aprendizaje automático, hasta las distintas estructuras de datos, clases y ficheros que ejecutan y simulan los entornos.

Primero de todo, para situarse dentro del marco 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 una primera instancia, 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 estudiará los procesos de decisión Markov (MDP), en los cuales se asienta la base de los algoritmos que se utilizarán. Una vez estudiado esto, se presentarán algunos de estos algoritmos, desde los más simples (*Monte Carlo*, *TD*) hasta los que utilizaremos 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 y cómo se estructuran las simulaciones dentro de la aplicación. Después, se desarrollará las dos principales arquitecturas de los entornos, la manera directa y la basada en manejadores. Definidas las arquitecturas, se estudiarán las principales estructuras de datos: las clases y los tensores. Por último, se estudiará como, una vez definidos los entornos, se realiza el aprendizaje y cómo se evalúa el resultad final.

A continuación, se analizará dos casos prácticos de la herramienta *IsaacLab*; cada uno construido a partir de una arquitectura diferente. Para ambos ejemplos, primero, se presentará el ejemplo escogido y la motivación detrás de 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 las clases contemplando sus atributos y métodos. Desmenuzado el entorno, se estudiará la ejecución del aprendizaje, valorando después el resultado final de esta. Por último, se presentarán algunas mejoras posibles dentro del ejercicio.

Habiendo estudiado las distintas características del RL y la herramienta, se comenzará a intervenir dentro del proyecto MetaTool. En este capítulo, definiremos el contexto del proyecto *MetaTool* y cuál es su principal objetivo. Seguidamente, se concretará las tareas en las cuales se intervendrá y el objetivo de la participación. Para cada caso realizado, se expondrán los problemas afrontados y las soluciones tomadas.

Otro punto importante del Aprendizaje por Refuerzo que se estudiará es el problema del Sim2Real, que consiste en la aplicación de las redes neuronales entrenadas 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 preparará un código para la implementación de las políticas en robots reales.

En la última parte del trabajo, se expondrán las conclusiones al proyecto en su conjunto, valorando las aportaciones realizadas, las dificultades encontradas y las oportunidades de trabajos futuros. En este trabajo, debido a la gran extensión de esta disciplina, se dejan de cubrir 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, pese a que se realice aprendizaje por refuerzo profundo, no se entrará en detalle en la base matemática de sus algoritmos, así como las bibliotecas implementadas con estos. Se estudiarán las características de las redes neuronales, pero no se profundizará en la matemática detrás de ellas.

Este será el alcance completo del trabajo. Sim 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 existen principalmente dos líneas: una parte teórica acerca del aprendizaje por refuerzo y una parte práctica mediante programación en Python y finalmente URsim. Por otro lado, la preparación de este documento se ha realizado después de 8 meses realizando tareas de programación e investigación por propia cuenta 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 después de haber realizado la mayoría de las labores teóricas y prácticas del trabajo. Teniendo así una visión general del trabajo global, se han expuesto las distintas características del trabajo y su enfoque general.

Para el estado del arte, al querer mostrar una visión general del estado del aprendizaje por refuerzo en la robótica, se han buscado distintos artículos de investigación sobre esta disciplina y sus aplicaciones prácticas. Al haber realizado este ejercicio después de este periodo de aprendizaje y práctica, se han podido identificar los factores más importantes de cada artículo, así como identificar los artículos más relevantes.

En cuanto al apartado 3, en el cual se exponen los fundamentos teóricos del Aprendizaje por Refuerzo, se ha seguido 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 base para sus algoritmos. Una vez obtenida una visión general, y después de aplicar es-

ta visión en labores prácticas, se estudió más específicamente cada elemento, indagando en distintas fuentes de información. Cabe resaltar dentro de estas fuentes, el libro sobre aprendizaje por refuerzo de Sutton y Barto [6]. Sobre este se trabajan la gran mayoría de definiciones formales.

El 4º apartado, acerca de la herramienta *IsaacLab*, se trató 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 utilizó 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; y su información detallada acerca de la propia plataforma y sus bibliotecas. A esto se le sumo el conocimiento aprendido 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.

El apartado 5 y 6, al constar de un análisis concreto de un ejemplo proporcionado por la plataforma, se ha utilizado los conocimientos aprendidos 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 se ponen en ejecución. Con esta estructura identificada, se realizó un diagrama de clases, identificando los distintos atributos y las funciones utilizadas. Con este diagrama en mente, se estudió cada clase por si sola, explicando la función de cada apartado del código y su aportación global al ejercicio.

A su vez el apartado 7, se debe tratar de manera distinta, pues se trata de tareas dentro de un proyecto externo. Por ello, antes de analizarse el ejercicio se realizará un pequeño estudio del proyecto general. Con el enfoque general en mente, se realizará un estudio de las tareas realizadas. A diferencia del apartado anterior, este estudio se basará en la depuración realizada del código, en el cual se realizaron una serie de correcciones para su correcto funcionamiento.

El apartado 8, enfocado a la implementación en el robot de políticas, se estudió un caso real de esta implementación, pero en un robot distinto. Estudiado este caso, se diseño un programa para su uso en el *UR3 Robohabilis*, utilizando el lenguaje *URSim* para el control del Robot y algunas clases implementadas en el caso estudiado.

Por último, una vez terminado el trabajo, se expondrá las conclusiones obtenidas de este, basándose plenamente en la experiencia obtenida en el transcurso del proyecto.

En conclusión, este trabajo sigue distintas metodologías, dependiendo si se trata de un enfoque práctico o teórico. Tomando en conjunto todo, se podría definir una metodología general. Primero, se realiza el estudio teórico, tanto del aprendizaje en refuerzo general

como el de la herramienta específica. Después, se utiliza los conocimientos obtenidos para realizar estudios prácticos y analíticos.

Toda la metodología y alcance descrito se concretan en este documento. A continuación, se explicará como se encuentra estructurado.

1.5. Estructura del documento

PENDIENTE

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 ejercicio 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 comportamiento, *Obélix* [7]. 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 esta afianzado en la robótica como una disciplina de rápido desarrollo. Especialmente, la técnica de aprendizaje profundo, basada en la implementación del aprendizaje por refuerzo para crear redes neuronales profundas [8], ha tenido un gran resultado 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, locomoción, navegación, manipulación, etc. Además, se ha mostrado también su efectividad tanto en robots individuales como colaborativos. [9]

A continuación, presentaremos 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 [10]. 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 físicas y espacios. Añadido a esto, el aprendizaje llevado al mundo real se vuelve lento e inseguro. A pesar de esto, los métodos de aprendizaje por refuerzo profundo han tenido bastante éxito dentro de esta disciplina. [9]

Un ejemplo de esta aplicación se da en el artículo *"QT-Opt: Scalable Deep Reinforcement Learning for Vision-Based Robotic Manipulation"* [11]. En el utilizan métodos de aprendizaje por refuerzo para generalizar el agarre de objetos desconocidos. Para ello, utilizan de entrada una cámara RGB para poder obtener datos acerca de la forma del objeto. Con esta entrada, conforman un algoritmo llamado QT-opt para elegir una acción de agarre, conformando una función acción-estado Q y resolviendo esta para obtener el máximo valor de éxito. En el apartado 3, se entrará en detalle sobre como se conforma esta función y los distintos algoritmos que se pueden usar para resolverla.

En la actualidad, encontramos casos como *"DORA: Object Affordance-Guided Reinforcement Learning for Dexterous Robotic Manipulation"* [12]. En él, se propone un 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, pudiendo coger más veces un martillo por su mango.

Añadido a estos ejemplos de investigación, esta tecnología se ha empezado a aplicar en el entorno industrial. Covariant, una empresa dedicada a la implementación de la inteligencia artificial en la robótica [13], 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 realizar 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 realizar agarres a través de instrucciones de texto y observaciones. [14]

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 venía ya muy ligada a esta disciplina dando grandes avances en el desarrollo de cuadrúpedos. Ya entrada en la era del aprendizaje profundo, se llevo su implementación a otros problemas de locomoción, como por ejemplo robots bípedos. [9]

Pese a que los primeros ejemplo de RL en locomoción se aplicaron a estos cuadrúpedos, el desarrollo real de estos llego con la implementación del DRL [9]. En *"RMA: Rapid Motor Adaptation for Legged Robots"* [15] 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 aprender a estimar un vector intrínseco del entorno que detalla sus propiedades. Luego 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 consta, comparada a la locomoción de cuadrúpedos, de un problema más complejo. Debido a un menor número de apoyos, se obtiene una falta de redundancia y una reducción de la estabilidad, haciendo necesario un control más preciso y complejo. Sin embargo, gracias al 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"* [16], se presenta un modelo de aprendizaje para el control de bípedos. En él, proponen un doble registro de estados, 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 aprendizaje constante.

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 [17, Pág. 1].

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

El aprendizaje supervisado aprende a agrupar pares de entradas y salidas de información, a través de ejemplos catalogados [18, Pág. 137]. Estos ejemplos constan de pares entrada y salida conocidos, los cuales sirven para clasificar futuras entradas. Un problema de aprendizaje supervisado podría ser identificar tipos de animales mediante una base de datos previa. En este caso, se alimenta al modelo con imágenes de animales (entrada) y su nombre (salida). El modelo deberá crear relaciones entre ambos. Se evalúa finalmente al programa por su habilidad de asociar imágenes de animales a su nombre.

El aprendizaje no supervisado, por otro lado, utiliza directamente las entradas sin una salida asociada [19, Pág. 740]. Esto hace que el programa deba buscar patrones lógicos inherentes a su clasificación. Estos patrones se basan en características a estudiar [18, Pág. 142]. Un problema de aprendizaje no supervisado podría ser agrupar imágenes de animales en función de su especie. Es este caso, se alimenta al programa solo con las imágenes. Siguiendo únicamente la composición de los animales mostrados, deberá agruparlos.

La frontera entre ambas disciplinas puede resultar difusa. No existe una diferencia formal entre ambas, pues la diferencia entre una característica a estudiar y una salida asociada no es absoluta [18, 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 la hace especialmente útil para el estudio de la robótica.

3.2. Estructura del Aprendizaje por Refuerzo

El aprendizaje por refuerzo esta 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 existe 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 el activamente. El entorno aporta al agente observaciones y recompensas, obteniendo así información sobre él. Además, el entorno, tiene asociado un estado. 3.1

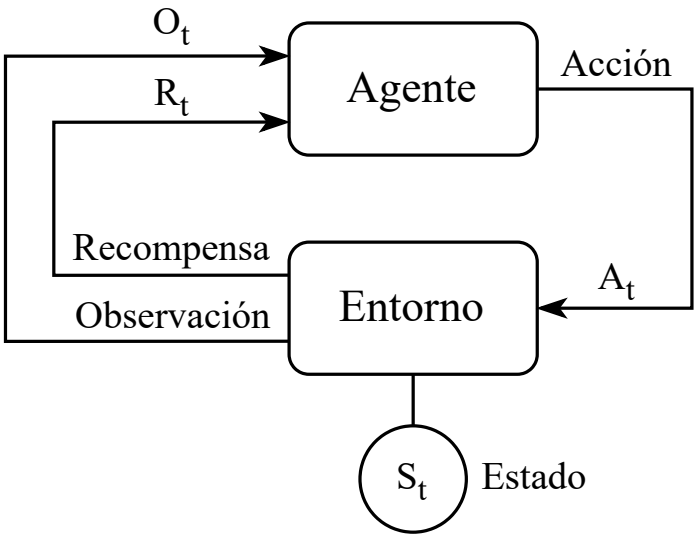

Figura 3.1: Interacción agente–entorno.

Un ejemplo de esta estructura, fuera del aprendizaje por refuerzo, estaría en los estudios de condicionamiento operante de Skinner [20]. 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. Uno de ellos emite una descarga eléctrica al animal, mientras que el otro le proporciona un estímulo positivo. Este ejemplo, a pesar de ser 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). Esto 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, ambas 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 Skinner, el estado y las observaciones coinciden, siendo únicamente la posición de los botones (derecha o izquierda). En la robótica, la mayoría de las veces tendremos espacios parcialmente observables [21].

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 ejemplificado.

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 sería 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 recompensas asociadas, de ahí la realimentación retrasada [6, Pág. 47]. Al definir de manera

ideal la toma de decisiones, sirven para formular el problema de aprendizaje por refuerzo de manera idealizada. Esta idealización 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 olvidar 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 MDPs, por lo que no se estudiarán en este trabajo. Sin embargo, si se prefiere entender gradualmente los conceptos de estados y sus transiciones, es recomendable trabajarlos.

3.3.1. Formulación de los MDP

Los MDP tienen asociada una formulación matemática. Para facilitar el estudio de esta, se facilita 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 varias 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]. Está 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].

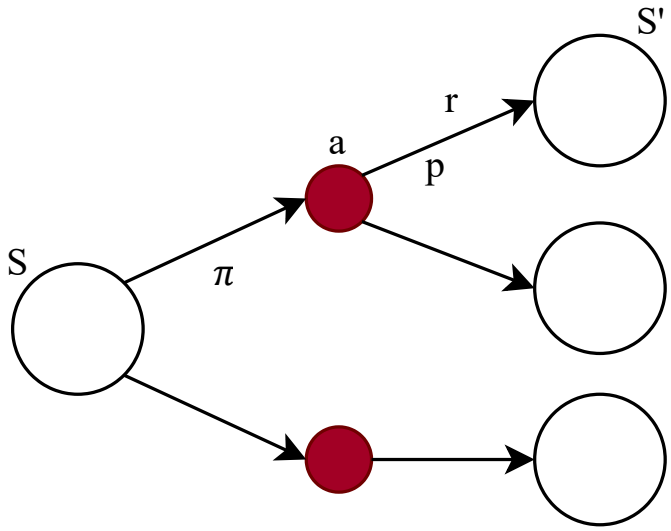

Figura 3.2: Subestructura de un MDP.

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 a un estado S' desde un estado S, que desde un estado S".

Referente a las probabilidades, cabe definir correctamente como 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 trabajado las definiciones de este apartado. Se define la probabilidad como una función determinista de 4 argumentos: $SxRxSxA \rightarrow [0, 1]$. Esta matriz nos permite obtener en función del estado anterior y la acción tomada, el estado actual y la recompensa recibida.

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

Esta relación es importante a la hora de 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.

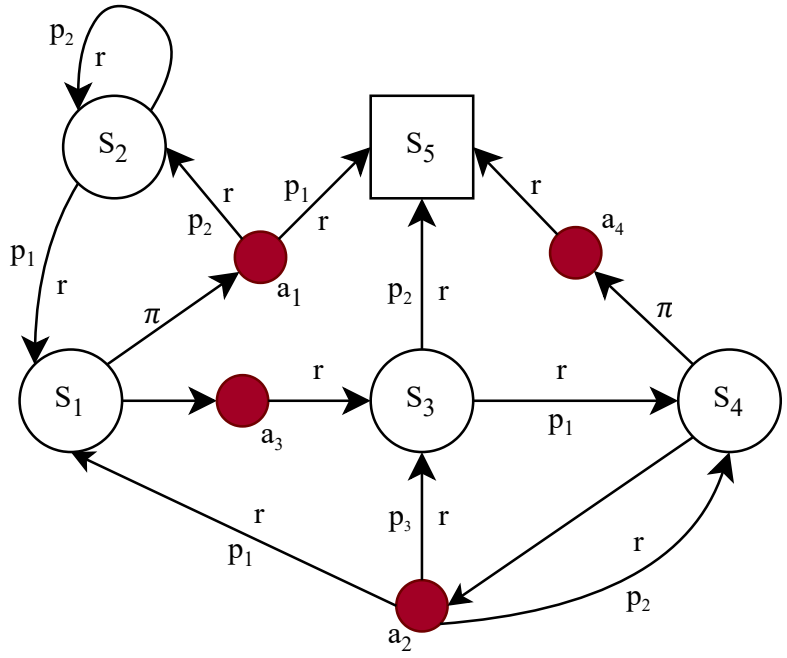

Figura 3.3: Ejemplo de un MDP completo

Este diagrama representa un MDP completo. Sobre este ejemplo se pueden observar la naturaleza de las trasiciones 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 absorvente* [6, Pág. 57].

Para ilustrar mejor aspectos relevantes de los MDP, se irá estudiando 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 esto, es importante tener en cuenta que aunque se acabe en el mismo estado, puede haber una recompensa en dicho instante. Se puede tener en cuenta 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 considerar como un proceso individual, se debería definir como un Proceso de Recompensa

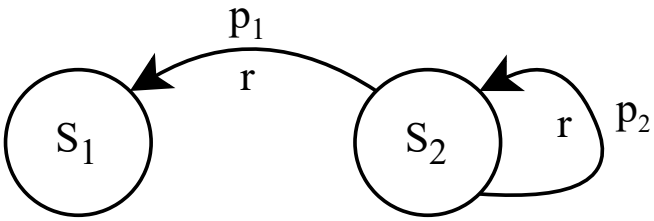

Figura 3.4: Subdiagrama del estado 2.

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

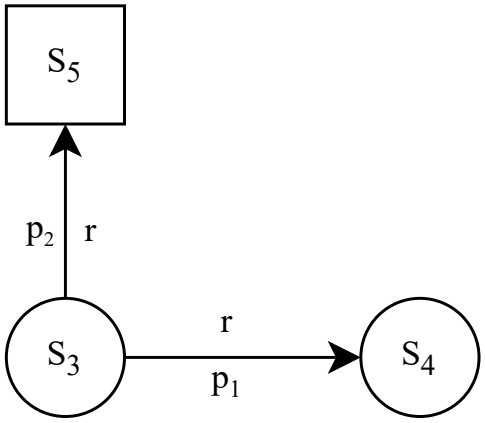

Figura 3.5: Subdiagrama del estado 3.

Por último, en el estado 4 (figura 3.6), vemos un ejemplo como 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 a el estado final. Si se toma, por otra parte, la acción 2, encontramos distintas transiciones asociadas a probabilidades. Dentro de este caso, podemos observar la naturaleza de las probabilidades descritas. La probabilidad de transicionar al estado 3, y obtener su recompensa asociada, 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]:

$$\sum s \in S \sum r \in Rp(s', r|s, a) = 1, \text{para todos} \in S, a \in A \tag{3.2}$$

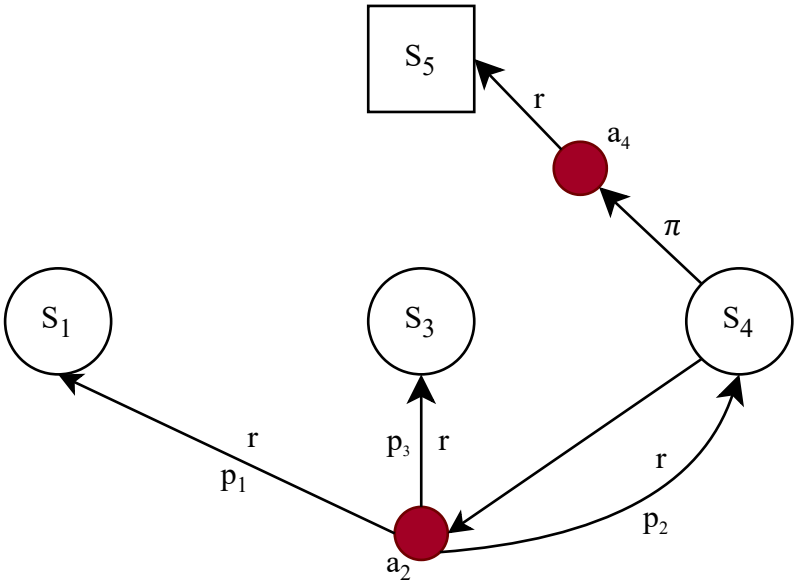

Figura 3.6: Subdiagrama del estado 4.

3.3.3. Funciones de Valor y Ecuación de Bellman

Las funciones de valor son una parte elemental del aprendizaje por refuerzo. Estas nos permiten analizar la recompensa esperada de 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*. Este concepto se asocia, en su caso más simple, a la suma de una secuencia de recompensas:

$$G_t = R_{t+1} + R_{t+2} + \cdots + R_T \tag{3.3}$$

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:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\text{inf}} \gamma^k R_{t+k+1} \tag{3.4}$$

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:

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

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

Para ejemplificar este valor de retorno, vamos 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:

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

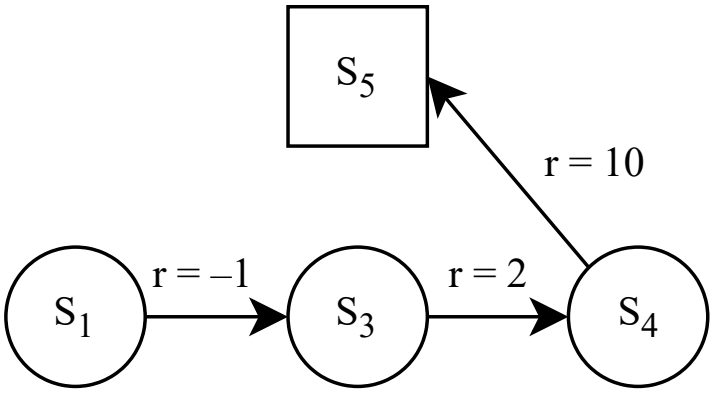

Figura 3.7: Ejemplo para el cálculo del valor de retorno

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:

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

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]:

$$v_\pi(s) = \mathbb{E}[G_t \mid S_t = s], \text{para todos} \in S \tag{3.8}$$

2. **Función de valor estado-acción**: La función de valor de un par acción, a, y estado, s, es el retorno esperado tomado la acción, a como punto de partida [6, Pág. 58]:

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

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, cómo veremos en el apartado 3.4. La *Ecuación de Bellman* se obtiene de modo que [6, Pág. 59] para función valor estado (teniendo en cuenta la figura 3.2):

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

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):

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

Ahora, se va estudiar un ejemplo de aplicación sobre la figura 3.9. En este ejemplo se trabaja en 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, $v_\pi(s)$. Se supone que la política, π, tiene un probabilidad del 80 % de elegir la acción, a_4, que las probabilidades p_1, p_2 y p_3, son respectivamente 0.5, 0.3 y 0.2 y que el *factor de descuento*,

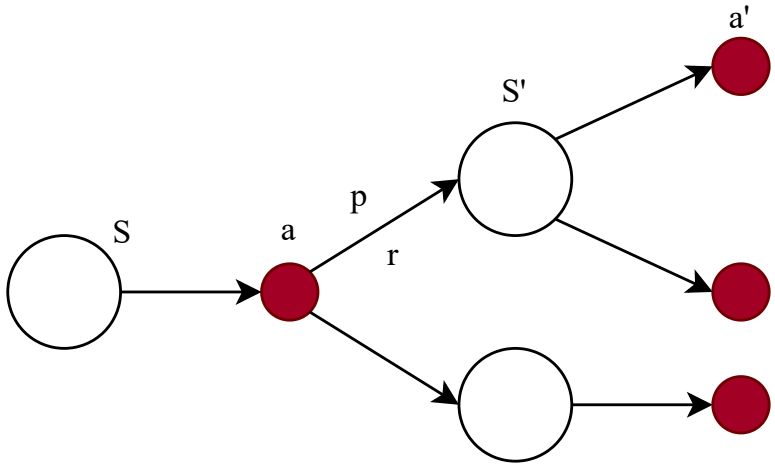

Figura 3.8: Estrcutura de un MDP enfocada al par estado-acción

γ, es 0.6.

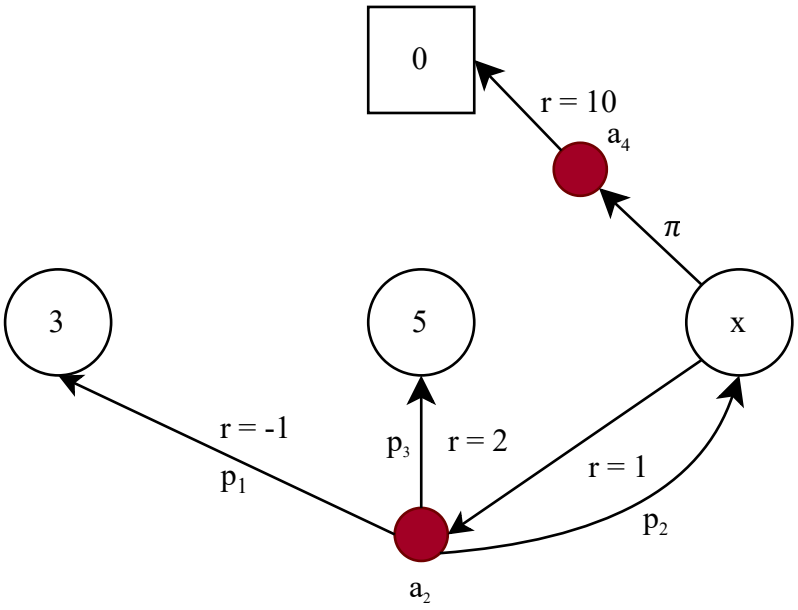

Figura 3.9: Ejemplo para el cálculo de la función de valor.

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:

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

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

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

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

Una se tiene las funciones de valor acción-estado, se puede calcular la función 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, aún cociéndolas, 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 de esto, se debe comprender porqué 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, vamos 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 valor estado para dicha política son iguales o mayores para todos los estados:

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

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*, asi como la misma función de valor estado-acción, la *función de valor estado-acción óptima*:

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

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

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 veremos 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, veremos varios de estos. Dentro de la disciplina de Aprendizaje por Refuerzo, estos algoritmos han aumentado su complejidad para poder cubrir problemas de más amplios. Los primeros algoritmos cubren MDP discretos e ideales, por lo que tienen una aplicación práctica reducida; y en el caso de este trabajo, nula. Sin embargo, es importante comprender su teoría para entender como aprenden los agentes. La base de todos estos algoritmos es compartida.

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 como 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 perfecto MDP. 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]:

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

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

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 simple el valor máximo de recompensa, es decir, de funciones de valor [6, Pág. 74]. Cabe resaltar que para poder realizar el calculo de la política óptima se deben conocer las dinámicas del sistema, es decir, se conoce $p(s', r \mid s, a)$ para todo $s \in S$, $a \in A(s)$, $r \in R$ y $s' \in S^+$ [6, Pág. 74]

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

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

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

- 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 termino de 3.20, la cual responde a:

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

Sumado a esto, existen dos 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:

$$\pi_0 \to v_0 \to \pi_1 \to v_1 \to \cdots \to \pi_* \to v_* \tag{3.21}$$

- 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:

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

  Una vez obtenida la política óptima (cuando $v_{k+1} - v_k = 0$), configuramos 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 generaliza* 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 lineares, pero se vuelven ineficientes al escalar el número de entornos [6, Pág. 87].

Por otro lado, DP no son 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. Por esto, es inviable cuando se trabaja un gran número de estados. Además, en robótica, los estados son continuos; por lo que habría que

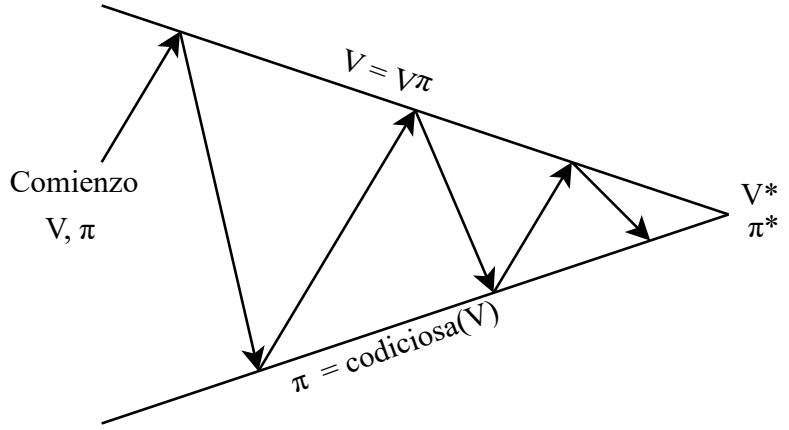

Figura 3.10: Diagrama de fases en DP [6, Pág. 87]

introducir una aproximación. Por último, en robótica, pese a la posibilidad de calcular su modelo dinámico, esto no interesa 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 Montecarlo

El primero de estos algoritmos que se estudiará es el método de Monte Carlo o MC. Este método se formalizo en 1949, en el artículo de Metropolis y Ulam [22]. En el se presenta un método para calcular mediante un enfoque estadístico para resolver problemas matemáticos complejos y difíciles 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 próximo 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 unos principales problemas de DP, la necesidad de conocer el modelo completo del entorno. Esto se hace estimando la media de los retornos obtenidos. Para ello, se deberá dividir 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).

Así 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 termino

predicción. El control se asociaría a la mejora de la política. A continuación, se estudiará como se implementa cada parte. En este caso, se estudiará la predicción y el 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 la 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 la 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 en los cuales 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 cosideración 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 cual será el siguiente estado; mientras que siempre tendremos 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 que acciones son mejores; sin embargo, si se tiene un 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, veremos 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 exploratorios* [6, Pág. 92]. Para que este se de, debe existir una probabilidad de empezar en cualquier de los pares estado acción; de modo que todos se puedan valorar.

Esta condición no es fácil o eficiente que se de. En los casos aplicados a la robótica que se estudiarán, no se dá esta condición. En estos casos, al ser estados continuos es difícil que esta situación se de; tenemos 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 $\epsilon - codiciosa$. Esta política regula el grado de exploración a través de un coeficiente ϵ. A través de este método, podemos escoger la

política de la siguiente manera [6, Pág. 101]:

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

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 estudiará 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

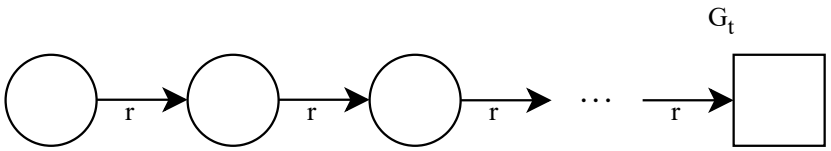

Figura 3.11: Figura de obtención del retorno en MC.

herramienta que permite actualizar inmediatamente: el retorno esperado, es decir, la función de valor. A continuación, se estudiará 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 revolucionó 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]:

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

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

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 no se ocupe la gran parte de la capacidad computacional en las actualizaciones. Para ello, existen los algoritmos *n-pasos TD* [6, Pág. 141]. Estos permiten regular en que momento se actualizan las funciones valor. Un TD de *1-paso*, sería el algoritmo que se acaba de estudiar, donde actualizamos 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 puede observar en la figura 3.12.

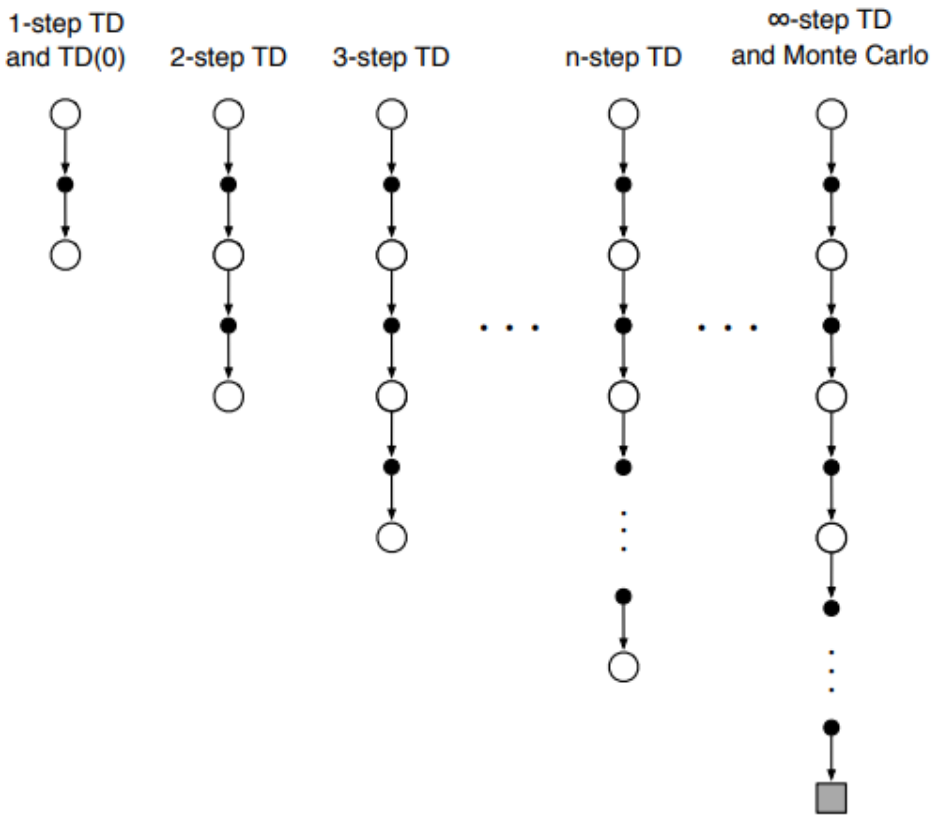

Figura 3.12: Resumen de los tipos de *TD n-pasos*

El control de este 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 trabajaremos 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 verán 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 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 el parámetro de $\epsilon - codiciosa$ 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 única política estable (poca exploración $\epsilon \to 0$). El problema de esta forma de trabajo es que al tender ϵ a 0, se dejan de visitar todos los pares estado-acción, por lo que no converge a la política óptima.

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 tiene 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 esta en el algoritmo *Q-learning*, que actualiza tal que [6, Pág. 131]:

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

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 no es segura. *Off-policy*, por otro lado, permite aprender una buena política rápidamente y manteniendo la exploración; sacrificando la estabilidad.

Con todos estos conceptos, se pueden realizar ejercicios completos de aprendizaje. Son, además la base de los algoritmos modernos, los cuales utilizaremos en nuestros 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; los cuales veremos a continuación. 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 obtenido, es decir, que actualizamos los estados en función del retorno $S_t \rightarrowtail G_t$ (en Monte Carlo) o el retorno esperado del siguiente estado y la recompensa $S_t \rightarrowtail R_{t+1} + \gamma v(S_{t+1})$ [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 con lleva una gran complejidad y no se va a trabajar 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 la actualización de manera que $s \rightarrowtail u$, 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].

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

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 linear del vector de parametrización, w. Para ello, a cada estado s, se le asocia un vector tal que

$x(s) = (x_1(s), x_2(s), \cdots, x_d(s))^T$, con el mismo número de componentes que w. De este modo, se define la función aproximada del valor estado como:

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

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 definen entonces el valor de los estados mediante un vector de caracterización y una parametrización. Los algoritmos modernos, que utilizaremos en este trabajo, trabajan en la base de estos conceptos. Estos 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 por encima algunos de estos algoritmos, resaltando sus principales características y objetivos.

3.6. Algoritmos modernos (Aprendizaje profundo)

Los algoritmos modernos a los que se estudiaran 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 se pude 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 obtendríamos el resultado final de la red [18, Pág. 164]. Esta última capa devuelve el resultado final. Por ejemplo, para la figura 3.13, tendríamos 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 resultado final; no se conocerían las salidas del resto de capas, por lo cual se las denomina como *capas ocultas*.

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, mantenemos una parametrización, w, que en vez de ser lineal, consiste de una multitud de capas funcionales, $f^{(*)}$. Por otro lado, podemos 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:

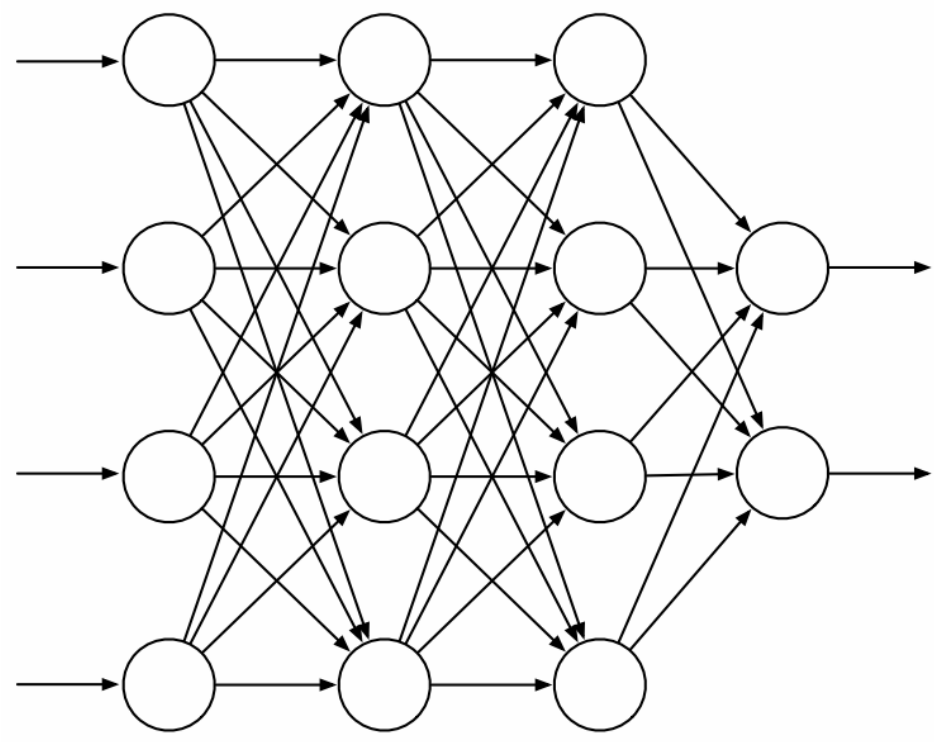

Figura 3.13: Ejemplo de una red neuronal artificial [6, Pág. 224]

- *Deep Q-Networks* o **DQN**: Este algoritmo combina el método *Q-learning* (Off-policy) con el uso de redes neuronales [23]. 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 [24].

- *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) [25].

- *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 [26].

- *Soft Actor-Critic* o **SAC**: Este algoritmo busca maximizar tanto la recompensa como la entropía [27]. Es decir, mediante una estructura de actor crítico, busca aumentar la recompensa buscando a su vez continuar explorando distintas acciones.

Estos son meramente algunos de los algoritmos utilizados en aprendizaje profundo. Como veremos en los ejemplos prácticos **??** y **??**, 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. Para ello, antes de nada, creo importante dedicar un apartado a como se conectan todos estos conceptos dentro del campo de la robótica; así se entenderán mejor las aplicaciones prácticas.

3.7. Observaciones para los ejercicios prácticos

Este trabajo, como se viene indicando, esta centrado en la aplicación del aprendizaje por refuerzo en la robótica. Para poder llevar estos conceptos teóricos presentados a este campo, es importante diseminar cada uno y ver como se conectan con el campo.

Primero de todo, el aprendizaje por refuerzo se basa en la mejora del rendimiento en una tarea con la experiencia. Es por tanto clave tener clara la tarea a implementar. En el primero de los casos que veremos (capítulo **??**) se busca enseñar a un robot araña a caminar, mientras que en el segundo caso (capítulo **??**), ya dentro del proyecto MetaTool, se enseñará a un robot a empujar un cubo con una herramienta. En ambos casos, se deberá tener clara esta tarea para configurar adecuadamente el entorno.

Una vez definida la tarea, se deberá visualizar la estructura general. Mi primera intuición al entender esta estructura dentro de las tareas a estudiar, fue asociar el agente al propio robot. Sin embargo, después de haberlos implementado, veo que es una afirmación desacertada. El agente en los casos trabajados siempre será la red neuronal sobre la que se trabaja. El robot, en realidad, forma parte del entorno. Esto queda claro cuando gran parte de las observaciones se toman del robot: la posición y velocidad de las articulaciones, la energía empleada, presión sobre este, etc. En los casos estudiados, esta red neuronal es la que decide las acciones que tomará el robot, siendo el verdadero objetivo del aprendizaje.

Los algoritmos, por otro lado, no serán el enfoque principal del trabajo, pero se deberán tener en cuenta a la hora de construir los entornos. Habiendo estudiado la base matemática, desde los procesos Markov hasta los vectores de caracterización, se comprende la importancia de elegir unas buenas observaciones, recompensas y acciones. Las recompensas, por un lado, se traducen directamente en las funciones de valor, las cuales rigen todos los algoritmos vistos. Por otro lado, las observaciones, definirán el vector de caracterización, definiendo como se identificarán los estados y decidiendo las acciones a través de ellas. Por último, las acciones, serán el principal objetivo del estudio; definirlas correctamente es clave para sacar el máximo partido a las capacidades de los robots.

Añadido a este análisis de los algoritmos, las redes neuronales estudiadas también serán importantes, especialmente en la implementación del aprendizaje a la realidad (capítulo **??**). La política final resultante se obtendrá como una red neuronal, por lo que

habrá que tener en mente su funcionamiento para implementar en el robot real.

Como se puede observar, 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, utilizaremos la herramienta IsaacLab, la cual analizaremos en detalle antes de comenzar con los ejercicios.

Capítulo 4

Análisis de la herramienta IsaacLab

En esta capítulo se va a introducir la herramienta IsaacLab. Haciendo uso de la documentación oficial de IsaacLab [28], el informe técnico [29] y la biblioteca de APIs [30]. El objetivo es obtener una visión global de cómo funciona la herramienta y cómo generar episodios, para luego después tener la capacidad de entender y crear entornos. En este capítulo se estudiarán los conceptos básicos, los cuales se estudiaran a fondo en los próximos capítulos con ejemplos prácticos.

4.1. ¿Qué es IsaacLab?

IsaacLab es un módulo de trabajo para el entrenamiento de robots en python. Su principal objetivo es simplificar las rutas de trabajo en este tipo de ejercicios [28]. IsaacLab esta enfocado en el trabajo sobre GPUs, combinando la renderización de imágenes realistas con el motor de físicas *PhysX* para construir simulaciones fieles a la realidad [29].

IsaacLab está construido sobre IsaacSim. IsaacSim es una aplicación construida sobre NVIDIA Omniverse, la cual permite desarrollar, simular y probar robots controlados por IA en entornos virtuales [31]. IsaacLab se puede entender como un conjunto de herramientas para usar dentro del simulador IsaacSim. De este modo, pese a que se trabajará enteramente con IsaacLab, se adquirirá en este capítulo algunos conceptos de IsaacSim.

Los principales incentivos para usar IsaacLab son [28]:

- Modularidad: capacidad de modificar y añadir nuevos entornos, robots y sensores; pudiendo utilizar todos estos en bibliotecas comunes, limitando las modificaciones.
- Código abierto: mantenimiento de un código abierto y libre para la comunidad. Esto permite 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. Esto permite partir 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 puede utilizar esta herramienta, ya que IsaacLab esta preparado para utilizarlas directamente. Esto saca el máximo partido a las tarjetas gráficas, pero limita el uso a la disposición de estos recursos. Por la parte de este proyecto, se dispuso de estas tarjetas gracias a la cesión de un ordenador por parte del equipo MetaTool. Aprovecho este momento para dar las gracias a Virgilio Gómez Lambo, tanto por los recursos prestados como por su ayuda en el entendimiento de la herramienta y el RL.

En conclusión, IsaacLab es una herramienta para realizar ejercicios de aprendizaje por refuerzo en IsaacSim. Con esto en mente, se va estudiar cual es la estructura externa de la herramienta.

4.2. Estrutura de la herramienta

La herramienta IsaacLab se centra en la construcción de los entornos, los cuales luego se someten al aprendizaje. Estos entornos, reciben las acciones del agente (la red neuronal) y procesando las recompensas y observaciones correspondientes [28, Walkthrough, Enviroment Background Design]. Para la construcción de estos entornos, IsaacLab utiliza la misma estructura que IsaacSim. Esta estructura define y gestiona los entornos. Se puede imaginar esta estructura como una muñeca rusa, donde cada nivel contiene al resto (figura 4.1).

El primer nivel consiste de la aplicación. La aplicación gestiona los recursos del sistema y es la encargada de lanzar y destruir la simulación [28]. La aplicación se puede gestionar a partir de la API *isaaclab.app* [30]. 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 esta. La simulación es la encargada de definir como funcionará las físicas, el tiempo y la gravedad. La simulación divide el ejercicio en múltiples instantes de tiempo, dividiendo las tareas de cálculo en subprocesos [28]. La simulación, al igual que la aplicación, tiene su propia API, *isaaclab.sim* [30]. Con esta, se definen parámetros como el tamaño de paso o la fuerza de la gravedad.

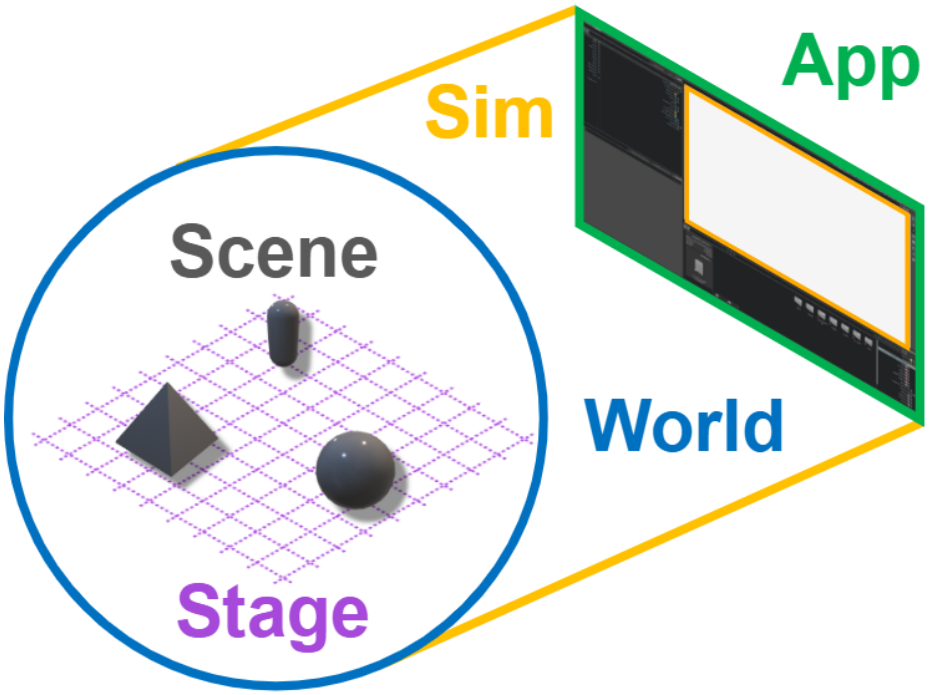

Figura 4.1: Estructura de la herramienta IsaacLab [28].

La simulación a la vez contiene todos los elementos relevantes a esta, los cuales agrupamos en el concepto de *mundo* [28]. Este mundo se define por el origen de coordenadas, el cual se toma como referencia para ubicar el resto de elementos. El mundo se estructura en dos elementos más: el escenario y la escena. El escenario, por un lado, provee de un contexto geográfico dentro de la escena [28]. 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, estará organizado por tanto en un escenario (figura 4.2) y administrado por la escena. Es por esto que la escena tiene su propia API, *isaaclab.scene* [30], la cual nos permite gestionar y obtener datos de todos los elementos del entorno. Estos elementos 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 **??**) [28]. USD, por su propia parte, es el lenguaje de descripción robots y entornos [32]. En este trabajo no manejaremos este lenguaje, pero si utilizaremos los ficheros de este tipo para importar los primarios.

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

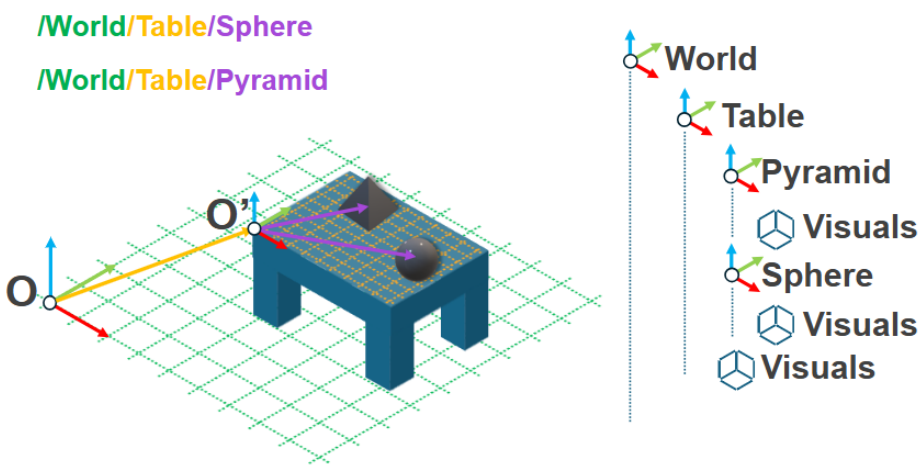

Figura 4.2: Organización de un escenario dentro de IsaacLab [28].

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 cliente y procesar las observaciones y recompensas. A parte 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 [28]:

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. Calcular y entregar las acciones y 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 deberemos definir dentro de este entorno los estados absorbentes (finales). En algunas ocasiones el robot alcanzará un estado donde no será relevant continuar con el aprendizaje. En ese estado, se cerrará el episodio y se reiniciará el entorno. Por último, IsaacLab nos permite optimizar el tiempo de entrenamiento clonado 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. Direct Based

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 una clase `DirectRLEnv`, dentro de la API *isaaclab.envs*. Para programar el entorno entonces, se definen las funciones abstractas de esta clase. Seguidamente, la clase se envuelve en un *wrapper* y se alimenta a una de las bibliotecas con los algoritmos de aprendizaje por refuerzo. Este proceso se verá en detalle en el apartado 4.5. A su vez, un ejemplo de este tipo de construcción se verá en el capítulo **??**.

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

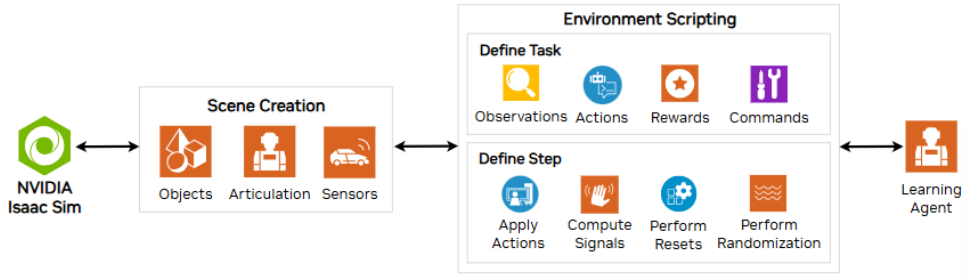

Figura 4.3: Diagrama para el modo directo de IsaacLab [28].

En conclusión, 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 [28]. Estos manejadores aíslan distintos aspectos de la comunicación y gestión del agente y entorno, facilitando la organización de estos.

Entre los distintos manejadores están [28]:

- 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 (`Observation Manager`): se encarga de definir y gestionar las observaciones.
- Manejador de acciones (`Action Manager`): se encarga de definir y gestionar las acciones.
- Manejador de recompensas(`Reward Manager`): se encarga de definir y gestionar las recompensas.
- Manejador de objetivos (`Command Manager`): se encarga de variar los objetivos del entrenamiento, siendo capaz de generar distintos valores en función de un rango.
- Manejador de finalización (`Termination Manager`): 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 (`Curriculum Manager`), encargado de crear una hoja de ruta para el entrenamiento. En este trabajo, al trabajar con objetivos concretos, no será 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* [30]. Los manejadores, se desarrollarán como clases pertenecientes a esta clase global. Estas clases se encuentran dentro de la API *isaaclab.managers* [30].

Más adelante, al entrar a estudiar el proyecto MetaTool, se estudiarán 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, solo hace falta definir las observaciones individualmente en estos términos, el manejador después se encarga de entregársela al agente.

En el modo basado en manejadores entonces, las interacciones entre el agente y el entorno, así como su administración, se fragmenta en múltiples manejadores, facilitando la definición y la gestión de los distintos elementos. Esto lo podemos ver 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 como se estructuran. Ambos modos de programar son útiles; por sus facilidades el modo de manejadores se priorizará.

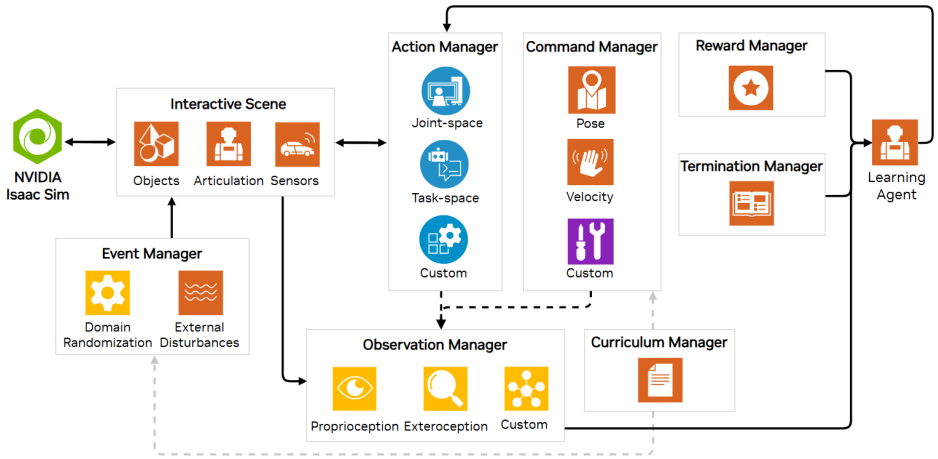

Figura 4.4: Diagrama para el modo basado en manejadores de IsaacLab [28].

En este trabajo, para tener primero una visión global de la construcción de entornos, se estudiará 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 encontrará toda la información relevante del entorno, que en su mayoría estará definida como tensores *Torch*. A continuación, se entrará un poco más en detalle sobre 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 debe 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 como 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 multiples estructuras de datos relevantes: variables, métodos, etiquetas, etc. Sin embargo, estos conceptos son propios de python por lo que no son de interés en este trabajo. Sin embargo, dentro de IsaacLab, toman especial relevancia

las clases (en especial las clases de configuración) y los tensores (trabajados con *PyTorch*). A continuación, veremos 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 [33, 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 veremos profundamente en los ejemplos, se accederá 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 básico sobre estas. Sin embargo, las clases dentro de IsaacLab se defines de una manera estandarizada, a través de otras clases [28].

Las clases establecidas dentro de IsaacLab se definen a través de otra clase, 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 verán 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 [33, Glossary, decorator]. Este decorador prepara la clase para trabajarse como un argumento.

Dentro de las clases, se encontrarán datos relevantes al entorno. Por ejemplo, en una clase *Articulation*, se encuentra una variable *data*, que es a su vez es una clase *ArticulationData*, la cual contiene información relevante al robot, como la posición de las articulaciones, *Articulation.data.jointpos*. Esta variable final, podría 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 deberíamos entonces 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 porqué son importantes.

4.4.2. Tensores: PyTorch

Un tensor es un matriz multi-dimensional de datos de un mismo tipo [34, torch.Tensor]. Un tensor tiene un número de dimensiones, a los que llamaremos ejes, y cada una de estas dimensiones tiene un tamaño, al que llamaremos tamaño de eje. Este tipo de dato permite organizar grandes cantidades de información en un mismo sitio. Por ejemplo, si se tienen

10 entornos, donde en cada uno se tiene un robot con una cámara, donde dicha cámara tiene una resolución de 800x400, 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 comprensiva el valor de todos los bits de las 10 cámaras. Cada eje, contiene a su eje inmediatamente inferior. El eje superior de este tensor sería el 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 [34]. 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 vayan utilizando. No obstante, antes de empezar a estudiarlos, se debe estudiar como se entrenan 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 él. El enfoque de este trabajo 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 4 librerías principales [28]:

- SKRL [35]: se enfoca en la modularidad, simplicidad y transparencia. Es especialmente útil si se pretende modificar los algoritmos para experimentar o investigar.
- RSL-RL [36]: se crea pensando en robótica, manipulación y locomoción, siendo la más específica de las tres. Esto lleva también a una corta documentación y alta complejidad.
- RL-Games [37]: esta biblioteca busca la optimización en entrenamientos intensivos y multi-agentes; especialmente útil para entornos complejos.

- Stable-Baselines [38]: 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 nos afecta en gran medida la falta de documentación. Por otro lado, su especialización en robótica la hace tener el mejor rendimiento ente todas las bibliotecas [28].

Para poder utilizar las bibliotecas, IsaacLab provee de un proceso para registrar y configurar agentes, *gymnasium* [39]. *Gymnasium*, es una biblioteca de python en código abierto. Provee de una API para comunicar los algoritmos de aprendizaje con los entornos definidos. Para ello, registraremos la configuración de los entornos a través de esta API, creando asi 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 como se comporta; para lo cual tenemos distintos enfoques. En el próximo apartado se estudiará cada uno de ellos.

4.6. Evaluación de agentes

Una vez obtenida la red neuronal, que vendría ser el 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* [40]. Este problema se verá en detalle en el capítulo **??**, donde integraremos 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 esto, se seguirá el siguiente procedimiento:

- *Sim-in-Sim*: Primero, mediante las herramientas de IsaacLab, se probará la red neuronal en la 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 implementará la red neuronal en el robot real.

El primer paso será 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 puede estudiar detenidamente las distintas recompensas y a través de ellas valorar el trabajo del robot.

Para el resto de pasos, se utilizarán otro tipo de herramientas y códigos, por lo que este también es la última utilización de la herramienta IsaacLab. Estudiando todas las utilidades de IsaacLab, se puede observar como abarca la gran parte del trabajo a realizar. Es por esto que 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 afronta la IA. Además, su utilización del potencial de las tarjetas GPU, las RTX, pese a limitar el uso a este hardware, permiten realizar ejercicios complejos de aprendizaje en poco tiempo.

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

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

Capítulo 5

Estudio caso locomoción

En este capítulo, se va a estudiar 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 principales clases y sus métodos y atributos más relevantes. 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 a su vez la configuración del agente. 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.

5.1. Descripción caso práctivo

El primer ejemplo escogido para el estudio es el entorno Ïsaac-Ant-v0". En este entorno se busca enseñar a andar a un robot araña de cuatro patas, en IsaacLab llamado *Ant* (figura 5.1). El objetivo principal será desplazar el robot en un dirección concreta.

Analizar este ejercicio es una parte integral de este trabajo. El objetivo a futuro de este trabajo es crear una guía para realizar futuros ensayos de aprendizaje por refuerzo. Este caso, se relaciona directamente con dos proyectos internos de la universidad, *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 la programación directa.

El código de este ejercicio se ha extraído de la herramienta IsaacLab; este se puede encontrar dentro del repositorio de la herramienta [29], accesible desde la documentación [28]. En este capítulo, se analizará el código desde el diagrama de clases; aportando

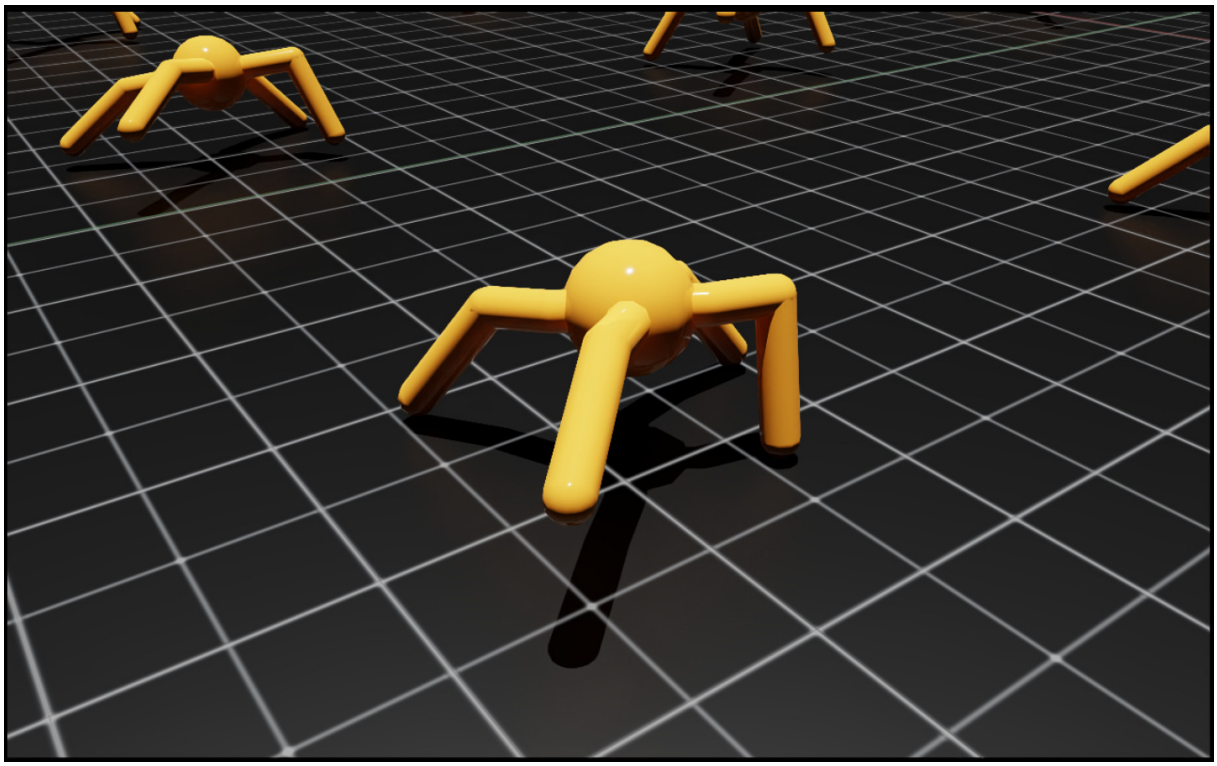

Figura 5.1: Robot araña o *Ant*, objetivo del aprendizaje para el primer caso práctico.

donde sea necesario los fragmentos de código relevante. Durante el análisis, también se irá indicando donde se encuentra la parte del código a la cual se hace referencia. Se ha preparado un proyecto de IsaacLab con todos los códigos utilizados; por lo cual, se indicará la referencia del código de IsaacLab y el proyecto. 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 diagram se utiliza para obtener una visión general de el código del entorno y para simplificar el futuro análisis de este. No se incluyen la totalidad de métodos y atributos, pues gran parte de estos no son relevantes para casos generales como los que se estudiarán. El resto de métodos y atributos son menos relevantes, usándose para funcionalidades muy concretas o para procesos internos de IsaacLab.

El diagrama muestra la construcción del entorno, sobre el cual se entrenará en el apartado 5.4 y 5.5. El entorno gira alrededor de dos piezas centrales, la clase `AntEnv` y la clase `AntEnvCfg`. Al estar trabajando en el caso directo, ambas clases heredan de sus contrapartes del modo directo: `DirectRLEnv` y `DirectRLCfg`, respectivamente. Estas clases están definidas dentro del código de IsaacLab; cada vez que se construya en un

entorno en modo directo se heredara de ambas.

En el caso del directo, la clase de configuración, aquella que hereda de `DirectRLEnvCfg`, 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 como se implementa la configuración del entorno, definiendo las interacciones y parte del proceso de aprendizaje y creando la escena a partir de lo definido.

Por otro lado, en este caso particular, se debe analizar de donde provienen ambas clases. Por un lado, la clase de configuración `AntEnvCfg`, hereda directamente de la clase de configuración original. Sin embargo, la clase principal del entorno de la araña hereda en un paso previo de una clase `LocomotionEnv`; esta hereda, esta vez sí, de `DirectRLEnv`. Esta clase intermedia es de gran utilidad, 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 locomoción, ajustando la escena al caso concreto 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 esto, es importante comprender como se implementa y define cada clase. En el próximo apartado, se estudiará detenidamente cada una de las clases.

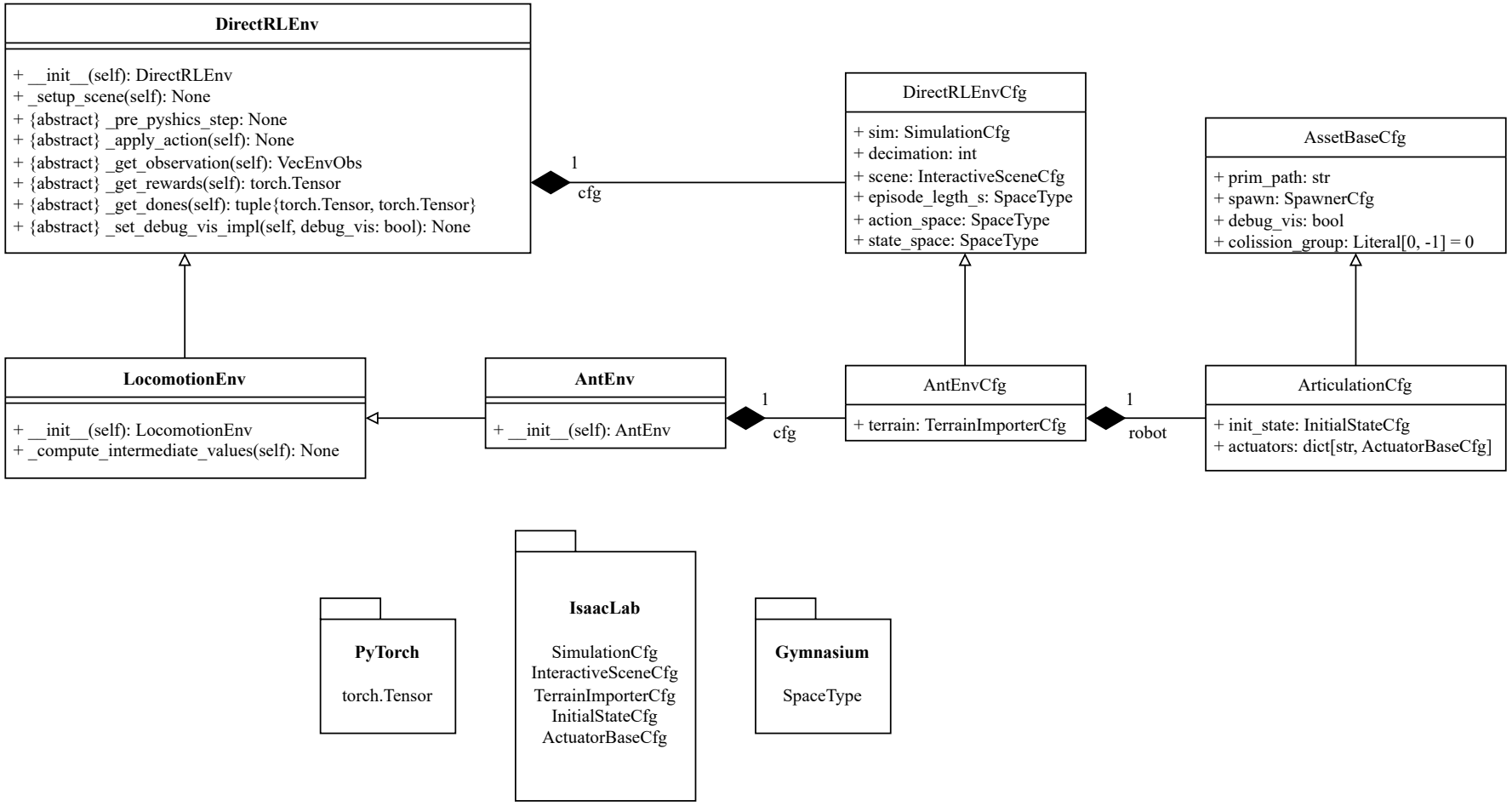

Figura 5.2: Diagrama UML del ejemplo araña, programación directa.

5.3. Análisis de clases

En este apartado se estudiará cada una de las clases mostradas en el diagrama, analizando los métodos y atributos definidos en el diagrama. Para cada una de las clases se indicará donde se puede encontrar el código. Después se explicarán la funcionalidad del método o atributo. Dentro de esta explicación, se mostrarán algunas partes del código donde exista un interés en la implementación del método; especialmente en aquellos que definan observaciones u recompensas del entorno. Se comenzará estudiando la clase principal padre, para pasar después a las distintas clases de configuración y se terminará con las clases heredadas de la primera.

5.3.1. DirectRLEnv

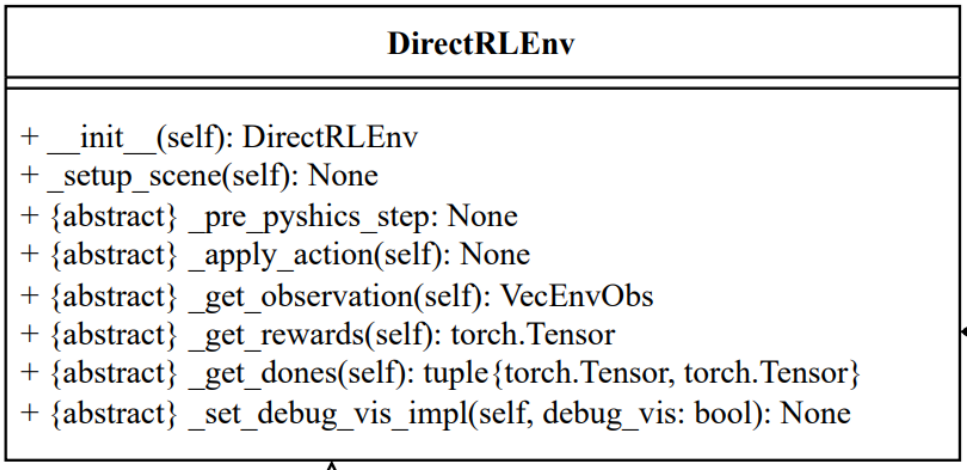

Figura 5.3: Imagen del diagrama referente a la clase `DirectRLEnv`.

La clase `DirectRLEnc` (figura 5.3.1) se encuentra definida en el código fuente de IsaacLab. Se puede acceder al código a través de la biblioteca de API de IsaacLab [30], concretamente en *isaaclab.envs.DirectRLEnv*. Una vez ahí, se debe seguir el enlace asociado al título, en el botón de "[source]"; tal y como se indica en la figura 5.3.1.

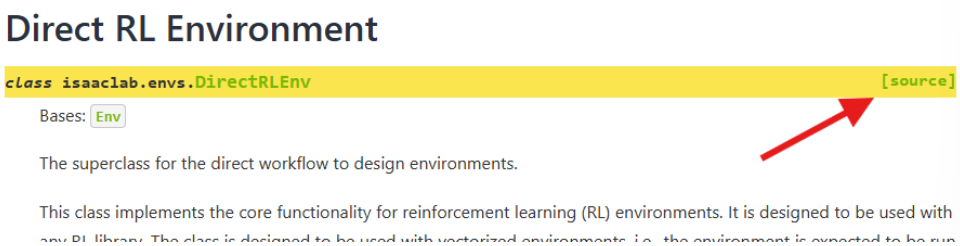

Figura 5.4: Imagen de la documentación oficial de IsaacLab con el link al código fuente [30].

Esta clase, cómo se viene comentando, es el pilar fundamental del entorno. Esta clase, a través de sus métodos crea el entorno y define sus propiedades e interacciones.

El primer elemento relevante de esta clase se trata del atributo definido como `cfg`. Este atributo almacena una clase `DirectRLEnvCfg`. Este atributo se utiliza constantemente en el resto de la clase, ya que es la configuración del entorno que se pretende construir. Es por esto, que se debe recoger en el constructor, 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 de código va enfocado al propio funcionamiento de IsaacLab, el cual no se estudiará.

El resto de métodos no son definidos en esta clase, sino que son meramente declarados. Exceptuando el método `_set_up_scene(self)`, el resto serán 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 estudiará únicamente el objetivo principal de cada una:

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

Esta clase, por tanto, define todas las funciones que deben utilizarse para crear y administrar el entorno. Dentro de esta clase, existen otros métodos como `step(self)` o `render(self)`, los cuales utilizan estos métodos para crear el proceso de comunicación con el entorno. Esta parte del código, no es relevante para este trabajo, pues forma parte del funcionamiento propio IsaacLab y no se deberá modificar a la hora de crear los entornos. Cabe resaltar que, a pesar de no ser parte del enfoque del trabajo, para tareas de depuración se ha necesitado comprender este proceso.

Como ya se ha mencionado, el elemento que definirá gran parte de esta implementación sera la clase de configuración. A continuación, se estudiará la clase base para luego analizar las respectivas clases heredadas.

5.3.2. DirectRLEnvCfg

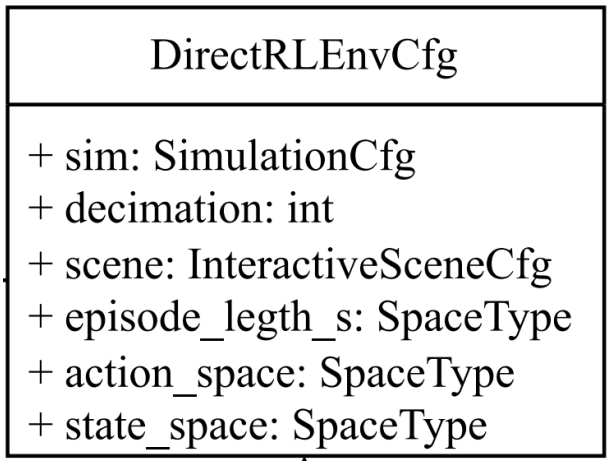

Figura 5.5: Imagen del diagrama referente a la clase `DirectRLEnvCfg`.

La clase `DirectRLEnvCfg` (figura 5.3.3), al igual que la anterior, se encuentra definida el código fuente; pudiéndose acceder de la misma manera desde la API *isaaclab.evns.DirectRLEnvCfg*. Esta clase esta definida como una *config_class*. Este tipo de clase se introdujo en el apartado 4.4.1. Esta tipo de clase almacena únicamente atributos, haciéndola más fácil de gestionar dentro del funcionamiento de la herramienta. En este apartado, se van a enumerar y analizar 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`: Amacena un valor numérico entero (int) que define el número de acciones realizadas antes de actualizar la política.
- `episode_length_s`: Almacena un valor numérico decimal (float) 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 esta.
- `obs_space`: Almacena una clase `SpaceType` que indica el número de observaciones realizadas sobre el entorno.

- `action_space`: De igual manera que el anterior, almacena una clase `SpaceType` que indica el número de acciones.

Cabe resaltar un par de cosas 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 posible clase heredada; es decir, todos los atributos deberán ser definidos en una clase específica de configuración. En segundo lugar, es interesante notar que existen dos variables para el número de las acciones y las observaciones pero no para las recompensas. Esto es debido a que la recompensa deberá definirse como una señal numérica, tal y como dicta el aprendizaje por refuerzo. El tamaño de las observaciones y las acciones por su parte definirán la dimensión de la red neuronal.

Vista la clase base de la configuración de entorno, se va estudiar como se hereda de ella para comenzar a definir un entorno concreto.

5.3.3. AntEnvCfg

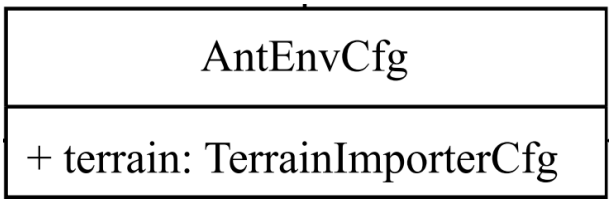

Figura 5.6: Imagen del diagrama referente a la clase `AntEnvCfg`.

La clase a estudiar, es la clase `AntEnvCfg`. Esta clase esta definida dentro del repositorio IsaacLab, en el directorio *source/isaaclab_tasks/isaaclab_tasks/direct/ant/ant_env.py* [29]. Esta clase hereda directamente de la clase `DirectRLEnvCfg`, incluyendo dos nuevos atributos: `terrain` y `robot`. Por su parte, `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 implementará en el siguiente apartado.

En este caso, al ser una implementación de una clase, se va estudiar el código detenidamente.

Al comienzo del código, se importan las distintas herramientas y bibliotecas que vamos a utilizar. 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 esta definida y después indicarla. En el código 5.1, se puede ver un ejemplo, donde se importa la clase `TerraimImporterCfg` de la API `isaaclab.terrains`. También se pueden importar clases definidas en archivos aparte, como se hace con la constante *ANT_CFG* (código **??**), 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`, `action_scale`, `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 cuanto 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 cabe resaltarlo pues se genera un plano simple, pero en el apartado de mejoras, se estudiará 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. Como ya se ha mencionado, 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 a parte. En el próximo apartado se verá como se configura el robot araña. También se define el atributo `joint_gears`, encargado de ajustar la fuerza aplicada en las acciones. Estas acciones también van estrechamente relacionadas con la configuración del robot, configuradas también 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 van a utilizar para cada recompensa. En el apartado 5.3.5 se verá cuales son estas recompensas y como se aplica este peso. No obstante, antes de llegar a estas se va estudiar la configuración del robot.

5.3.4. ArticulationCfg

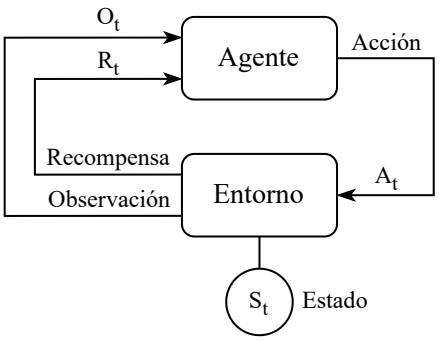

Figura 5.7: Imagen del diagrama referente a la clase `AntEnvCfg`.

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 estudiará la implementación de esta clase para el caso concreto de locomoción para el robot araña. Esta implementación se realiza en el archivo *source/isaaclab_assets/isaaclab_assets/robots/* [29], cuyo código se muestra en 5.5.

Esta clase hereda de la llamada `AssetBaseCfg`, dirigida a configurar cada prim de la simulación. Esta clase, asocia el prim a una dirección 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_bis` y con que objetos puede colisionar, mediante el atributo `colission_group`. Por esto, usaremos 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 luego se importa, como ya se ha visto en el apartado anterior, dentro de la configuración del entorno. En el

constructor, primero se definen los dos atributos heredados de la clase `AssetBaseCfg`.

En primer lugar, 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, 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 a la articulación con los atributos `rigid_props` y `articulation_props`. Por último, mediante el atributo `copy_from_source`, se indica si se usará 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 caracter "*". Esto hace que todas las articulaciones terminadas en "_leg"se les asocie el mismo valor. 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`).

5.3.5. LocomotionEnv

5.4. Registro del Entorno

5.5. Aprendizaje y Evaluación

5.6. Posibles mejoras

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] Sridhar Mahadevan and Jonathan Connell. Automatic programming of behavior-based robots using reinforcement learning. *Artificial Intelligence*, 55(2-3):311–365, June 1992.

[8] 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.

[9] 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].

[10] Matthew T. Mason. Toward Robotic Manipulation. *Annual Review of Control, Robotics, and Autonomous Systems*, 1(1):1–28, May 2018.

[11] 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].

[12] 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].

[13] Covariant | About.

[14] Introducing RFM-1: Giving robots human-like reasoning capabilities.

[15] Ashish Kumar, Zipeng Fu, Deepak Pathak, and Jitendra Malik. RMA: Rapid Motor Adaptation for Legged Robots, July 2021. arXiv:2107.04034 [cs].

[16] 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].

[17] Tom Mitchell and Hill McGraw. *Machine Learning textbook*. McGraw-Hill Science/Engineering/Math, 1997.

[18] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[19] Stuart J. Russell and Peter Norvig. *Inteligencia artificial: un enfoque moderno*. Pearson Educación, 2004. Traducción al español de *Artificial Intelligence: A Modern Approach*.

[20] B. F. Skinner. *The behavior of organisms: an experimental analysis*. The behavior of organisms: an experimental analysis. Appleton-Century, Oxford, England, 1938. Pages: 457.

[21] 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.

[22] 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.

[23] 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.

[24] Marc G. Bellemare, Will Dabney, and Mark Rowland. *Distributional Reinforcement Learning*. The MIT Press, May 2023.

[25] 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].

[26] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal Policy Optimization Algorithms, August 2017. arXiv:1707.06347 [cs].

[27] 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].

[28] NVIDIA Corporation. Isaac Lab Documentation.

[29] 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 Makoviychuk, 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 Muralt, 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 Moravanszky, 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.

[30] NVIDIA Corporation. API Reference — Isaac Lab Documentation.

[31] NVIDIA Corporation. Isaac Sim Documentation.

[32] NVIDIA Corporation. OpenUSD Fundamentals — Isaac Sim Documentation.

[33] Python Software Foundation. Python documentation, 2024.

[34] PyTorch Team. Pytorch documentation, 2025.

[35] 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.

[36] 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.

[37] Denys Makoviichuk and Viktor Makoviychuk. rl-games: A high-performance framework for reinforcement learning, May 2021.

[38] 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.

[39] 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.

[40] 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].