Universidad de Málaga

Escuela de Ingenierías Industriales

Departamento de Ingeniería de Sistemas y Automática

Trabajo Fin de Grado

Un *Framework* de Aprendizaje por Refuerzo para Tareas de Manipulación con el Manipulador Robótico Franka

Grado en Ingeniería Electrónica, Robótica y Mecatrónica

Autor: Diego Caruana Montes

Tutor: Juan Manuel Gandarias Palacios
Cotutor: Juan Antonio Fernández Madrigal

22 de junio de 2025

Declaración de Originalidad del Trabajo Fin de Grado

D./Dña. Diego Caruana Montes
DNI/Pasaporte: XXXXXXXX. Correo electrónico: xxxxxxxx@uma.es
Titulación: Grado en Ingeniería Electrónica, Robótica y Mecatrónica
Título del Proyecto/Trabajo: Un *Framework* de Aprendizaje por Refuerzo para Tareas de Manipulación con el Manipulador Robótico Franka

DECLARA BAJO SU RESPONSABILIDAD

Ser autor/a del texto entregado y que no ha sido presentado con anterioridad, ni total ni parcialmente, para superar materias previamente cursadas en esta u otras titulaciones de la Universidad de Málaga o cualquier otra institución de educación superior u otro tipo de fin.

Asimismo, declara no haber trasgredido ninguna norma universitaria con respecto al plagio ni a las leyes establecidas que protegen la propiedad intelectual, así como que las fuentes utilizadas han sido citadas adecuadamente.

En Málaga, a 22 de junio de 2025

Fdo.: Diego Caruana Montes

Resumen

Un *Framework* de Aprendizaje por Refuerzo para Tareas de Manipulación con el Manipulador Robótico Franka

Autor: Diego Caruana Montes

Tutor: Juan Manuel Gandarias Palacios

Cotutor: Juan Antonio Fernández Madrigal

Departamento: Departamento de Ingeniería de Sistemas y Automática

Titulación: Grado en Ingeniería Electrónica, Robótica y Mecatrónica

Palabras clave Robótica; Manipulador; Inteligencia Artificial; Aprendizaje por refuerzo

El presente proyecto consiste en el desarrollo de un *framework* de aprendizaje por refuerzo para un robot manipulador. El manipulador es un Franka Emika Panda, un robot colaborativo de 7 grados de libertad. Este TFG se enmarca dentro del proyecto de investigación TYRELL (Time-Ready Reinforcement Learning for Robotic Skills and Tasks). El *framework* permite crear tareas de aprendizaje por refuerzo y entrenar en entornos simulados modelos que pueden ser desplegados directamente en el robot real. El código se ha desarrollado en el lenguaje de programación Python, y se han utilizado la librería *dm_robotics_panda*, diseñada para controlar al manipulador y realizar simulaciones, las librerías de aprendizaje por refuerzo *Gymnasium* y *Stable-Baselines3*, y la librería `rl_spin_decoupler`, que actúa como puente entre las anteriores. Para demostrar la utilidad del *framework*, se realiza una serie de experimentos. En ellos se entrenan modelos para realizar tareas sencillas de manipulación en entornos simulados, y posteriormente se demuestra su funcionamiento en el robot real.

Este trabajo de fin de estudios ha sido financiado por el proyecto de investigación “Tyrell: Time-Ready Reinforcement Learning for Robotic Skills and Tasks”, código PID2023-147392NB-I00, por MICI-U/AEI/10.13039/501100011033 y los fondos FEDER de la Unión Europea, así como por los proyectos “CONCERTO” (PID2021-127221OB-I00) y “RollGrip” (AT21_00051).

Abstract

A Reinforcement Learning Framework for Manipulation Tasks with the Franka Robotic Manipulator

Author: Diego Caruana Montes
Supervisor: Juan Manuel Gandarias Palacios
Cosupervisor: Juan Antonio Fernández Madrigal
Departamemt: Departamento de Ingeniería de Sistemas y Automática
Degree: Grado en Ingeniería Electrónica, Robótica y Mecatrónica
Keywords: Robótica; Manipulador; Inteligencia Artificial; Aprendizaje por refuerzo

The present project involves the development of a reinforcement learning framework for a robotic manipulator. The manipulator is a Franka Emika Panda, a collaborative robot with seven degrees of freedom. This undergraduate thesis is part of the TYRELL research project (Time-Ready Reinforcement Learning for Robotic Skills and Tasks). The framework enables the creation of reinforcement learning tasks and the training of models in simulated environments that can be directly deployed on the real robot. The code has been developed in the Python programming language, using the *dm_robotics_panda* library, designed for controlling the manipulator and running simulations, along with the reinforcement learning libraries *Gymnasium* and *Stable-Baselines3*, and the `rl_spin_decoupler` library, which serves as a bridge between them. To demonstrate the usefulness of the framework, a series of experiments is carried out. In these experiments, models are trained to perform simple manipulation tasks in simulated environments, and their performance is then demonstrated on the real robot.

This final degree project has been funded by the research project “Tyrell: Time-Ready Reinforcement Learning for Robotic Skills and Tasks”, code PID2023-147392NB-I00, by MICI-U/AEI/10.13039/501100011033 and the European Union’s FEDER funds, as well as by the projects “CONCERTO” (PID2021-127221OB-I00) and “RollGrip” (AT21_00051).

A mi familia, que me ha ayudado a llegar a donde estoy.

Agradecimientos

En primer lugar, quiero agradecer a mis tutores, Juanma y Juan Antonio, por haberme guiado en este Trabajo de Fin de Grado. Esta enriquecedora experiencia, además de contribuir a mi desarrollo profesional, ha despertado en mí un interés por el mundo de la investigación académica. Gracias por todo lo que me habéis enseñado. Gracias a mis padres y mi hermana, que han confiado en mí, me han apoyado, y me han dado energía para seguir esforzándome en esta carrera de fondo. Por último gracias a todos los compañeros que he conocido y han trabajado a mi lado. Os deseo un futuro brillante y que encontréis trabajo de vuestros sueños ¡Gracias a todos!

Índice

Índice de figuras

Figura **Página**

Índice de tablas

Tabla **Página**

CAPÍTULO 1

Introducción

Contenido

En este capítulo introductorio se expone el problema que se aborda en el presente trabajo, y por qué se ha elegido tal problema. Además, se fijan los objetivos y se describe la estructura de la memoria.

1.1. Motivación

En las últimas décadas, el uso de robots manipuladores de todo tipo está experimentando un crecimiento considerable. Entre ellos destacan los robots colaborativos (*cobots*), que están pensados para poder interactuar de manera segura con las personas. Sus propiedades difieren de las de los robots industriales clásicos. Estos últimos son capaces de levantar grandes cargas, más allá del alcance de la fuerza humana; sin embargo, poseen un grado muy limitado de inteligencia, autonomía y sensores. Además, pueden suponer un peligro para los humanos, por lo que es necesario delimitar su espacio de trabajo con una jaula. Los *cobots* no presentan tales capacidades mecánicas y están dotados de sensores que aportan grandes cantidades de información, así como de algoritmos de control inteligente que permiten una colaboración directa con las personas en un mismo espacio [1]. Este tipo de robots tiene múltiples aplicaciones: pueden ser utilizados en procesos industriales y sistemas de seguridad e inspección, así como para asistencia personal y medicina, incluyendo cirugía, prótesis y masajes [2].

Para poder llevar a cabo estas tareas con un alto grado de complejidad, es necesaria la integración de métodos de toma de decisiones, planificación de movimientos y controladores complejos, lo que supone múltiples desafíos tanto a nivel de desarrollo como de implementación. Aquí entra en juego la inteligencia artificial (IA). Existe una gran variedad de métodos basados en IA que son ampliamente utilizados a día de hoy en robótica. Por ejemplo, redes neuronales para visión por computador o algoritmos de planificación para navegación autónoma. Este trabajo se centra en un campo de la inteligencia artificial muy utilizado en investigación en robótica, conocido como aprendizaje por refuerzo [3].

Aplicando técnicas de aprendizaje por refuerzo, un robot puede aprender a realizar una determinada tarea a base de entrenamiento. El robot ejecuta acciones y recibe recompensas en función de su rendimiento. Así, puede utilizar la información de las recompensas para aprender qué acciones son buenas y cuáles no. Una limitación de esta técnica es que entrenar un robot implica realizar una gran cantidad de pruebas en las que el robot interactúa con el entorno y comete errores; por este motivo, es una práctica común, por no decir universal, entrenar los modelos en entornos virtuales. Más información sobre aprendizaje por refuerzo en el capítulo 2.

Entrenar en simulación es más barato, rápido, y seguro que obtener muestras de datos en el mundo real, sin embargo, los simuladores no proporcionan una representación perfecta de la realidad; la simulación física basada en elementos finitos comete errores, especialmente cuando suceden fenómenos relacionados con la fricción, impactos y deformaciones. Las consecuencias de utilizar este tipo de entornos pueden ser modelos incapaces de generalizar correctamente o que requieren grandes ajustes manuales a la hora de implantarlos en el sistema real. Este fenómeno es conocido como *Sim2Real Gap* [4]. Aun así, es común encontrar investigación enfocada únicamente en entrenamiento de modelos en simulación, o *Sim2Null*, que difícilmente se puede aplicar en la realidad. Incluso si los modelos son aptos para su ejecución en un robot, rara vez los desarrolladores aportan una interfaz que permita su despliegue de manera inmediata.

Este Trabajo Fin de Grado (TFG), que se enmarca dentro del proyecto de investigación *TYRELL*[1] (*Time-Ready Reinforcement Learning for Robotic Skills and Tasks*), aborda este último problema, y se enfoca en desarrollar un *framework* de aprendizaje por refuerzo para el manipulador robótico Franka[2], un robot colaborativo de siete grados de libertad.

1.2. Objetivos y contribución

Los objetivos de este trabajo son los siguientes:

1. Desarrollar un *framework* de programación que permita entrenar modelos a través de aprendizaje por refuerzo para tareas de manipulación con el manipulador Franka Emika Panda.
2. El *framework* debe ser compatible con alguna librería ampliamente utilizada que implemente los algoritmos de aprendizaje por refuerzo.
3. Verificar que con el *framework* se pueden entrenar modelos capaces de aprender a completar tareas en un entorno virtual. Para ello, se deben realizar experimentos de entrenamiento en simulación.
4. El *framework* debe permitir entrenar modelos en simulación y también utilizarlos en un robot real sin esfuerzo. Se debe verificar tal funcionalidad realizando experimentos en los que se prueben en un robot real los modelos entrenados en simulación. Este objetivo es de vital importancia de cara a llevar a cabo tareas en el mundo real.

1.3. Estructura de la memoria

El capítulo 2 es una introducción a la teoría del aprendizaje por refuerzo. En él se explican los conceptos fundamentales para entender cómo interaccionan el agente y el entorno, y cómo el agente puede aprender a realizar una tarea. El capítulo 3 describe detalladamente el trabajo realizado, justificando las decisiones que se toman en todo momento. El capítulo 4 recoge los resultados de los experimentos que se han llevado a cabo junto con una minuciosa descripción de los mismos, con el fin de permitir su reproducibilidad. El capítulo 5 consiste en una valoración del proyecto basada en los objetivos planteados al principio del mismo y los resultados obtenidos. Quedan expuestas aquí las aportaciones que ha realizado el presente trabajo, así como posibles caminos para ampliar el mismo.

[1] Página web del proyecto: https://babel.isa.uma.es/research/projects/tyrell/, fecha 19/05/2025.
[2] https://franka.de/

CAPÍTULO 2

Contexto y marco teórico

Contenido

En este capítulo se explica brevemente qué es el aprendizaje por refuerzo. En primer lugar, se describe la estructura de un sistema de aprendizaje por refuerzo y su formalización matemática. En segundo lugar, se esclarece cómo funciona el mecanismo de realimentación gracias al cual un agente es capaz de aprender qué acciones debe tomar para cumplir un objetivo. También se explican los diferentes tipos de algoritmos que existen y sus características. En último lugar, se presenta el estado del arte.

2.1. Modelado de un sistema de aprendizaje por refuerzo

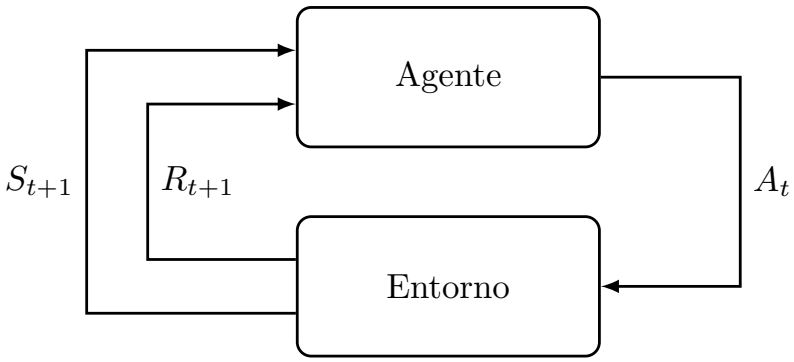


Figura 2.1: Sistema de aprendizaje por refuerzo compuesto de un agente y un entorno. El agente realiza acciones (A_t) sobre el entorno, y este le devuelve una recompensa (R_{t+1}) y el estado (S_{t+1}) en el que se encuentra.

El aprendizaje por refuerzo es un área del *Machine Learning* en la que una máquina aprende cómo debe actuar a base de prueba y error. Cuando se modela un sistema para aplicar técnicas de aprendizaje por refuerzo, aparecen dos elementos principales: el agente y el entorno. El agente es una entidad cuya tarea es actuar con el objetivo de obtener la mayor recompensa posible. Para tomar decisiones tiene en cuenta el estado (S_t) en el que se encuentra el entorno. El entorno se compone de todo lo que no sea el proceso de toma de decisiones y pueda interactuar con el mismo. Dicha interacción queda ilustrada en la figura 2.1. Al comienzo de cada período de tiempo discreto t, el agente observa S_t y decide ejecutar la acción (A_t), por la cual recibirá una recompensa ($R_{t+1} \in \mathbb{R}$) antes del siguiente período de tiempo. La secuencia sigue el orden $(S_0, A_0, R_0, S_1, A_1, R_1, ...)$. A diferencia de la recompensa, que es siempre un valor escalar, el espacio de acciones y el espacio de estados dependen de las características del agente y el entorno: pueden tomarse de espacios multidimensionales, y cada dimensión puede ser discreta o continua.

Formalmente, se asume que la interacción entre el agente y el entorno es un proceso de decisión de Markov (MDP, por sus siglas en inglés, *Markov Decission Process*). Un MDP consiste en una tupla (S, A, R_a, P_a), donde S es el espacio de estados, siendo cada estado único y contiene toda la información del entorno necesaria para tomar decisiones, A es el espacio de acciones, $R_a(s, s')$ es la recompensa por cambiar de un estado s al estado s' tomando la acción a, $P_a(s, s')$ es la probabilidad de cambiar del estado s al estado s' cuando se toma la acción a. Este tipo de procesos puede ser ilustrado mediante grafos como los de la figura 2.2 (tomada de [3]). Es útil asumir que la interacción entre el agente y el entorno viene dada por un proceso de Markov debido a la propiedad de Markov, que establece que la evolución del sistema depende únicamente de su estado actual y no del pasado. Gracias a esta propiedad, si se tiene un agente capaz de acceder al estado del entorno, no necesita tener memoria de lo acontecido anteriormente.

s	a	s'	$p(s' \mid s, a)$	$r(s, a, s')$
high	search	high	α	$r_{\texttt{search}}$
high	search	low	$1-\alpha$	$r_{\texttt{search}}$
low	search	high	$1-\beta$	-3
low	search	low	β	$r_{\texttt{search}}$
high	wait	high	1	$r_{\texttt{wait}}$
high	wait	low	0	-
low	wait	high	0	-
low	wait	low	1	$r_{\texttt{wait}}$
low	recharge	high	1	0
low	recharge	low	0	-

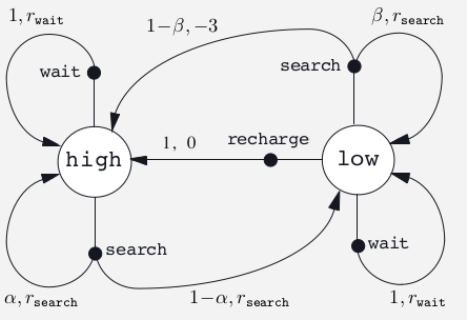


Figura 2.2: Ejemplo de grafo que representa un MDP para el mantenimiento de la batería de un dispositivo, con estados *high* y *low*. Las acciones que se pueden tomar son *recharge*, *search* y *wait*.

Véase el ejemplo de sistema de la figura 2.3. Se tiene una cuadrícula de tamaño 5×5, varios trozos de queso esparcidos aleatoriamente y un ratón. El agente es el ratón, y el resto de elementos forman parte del entorno. El ratón, usando sus ojos, puede ver todas las casillas del entorno, en cuál de ellas se encuentra y cuáles de ellas tienen queso. Ese es el estado del entorno S. Después de observar, el ratón lleva a cabo una acción A, que puede ser permanecer en la misma casilla o desplazarse hacia arriba, hacia abajo, hacia la derecha o hacia la izquierda. Tras cada acción, el ratón recibe una recompensa $R = 10$ si encuentra queso en su nueva posición, y $R = -1$ si la casilla está vacía.

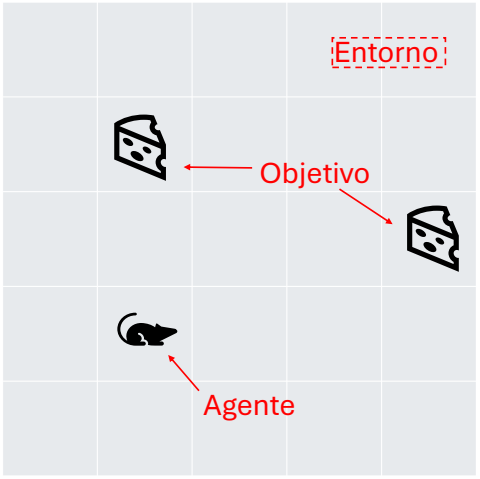


Figura 2.3: Ejemplo de sistema compuesto de un agente y un entorno. El agente es un ratón cuyo objetivo es buscar trozos de queso. El entorno se compone del espacio cuadriculado y el queso.

El modelo ideal de la interacción entre el agente y el entorno consiste en un MDP. Sin embargo, este modelo asume que el agente es capaz de inferir el estado del entorno, un estado que contenga toda la información del mismo necesaria para tomar decisiones y fiable. En el mundo real esto no sucede. Los agentes pueden captar una información reducida, y posiblemente ruidosa, una observación O. Por ejemplo, si se tiene una cámara, esta recibe luz de los objetos a su alrededor, pero la imagen que devuelve no contiene toda la información del entorno. Si hay una pared, no se puede discernir si hay un objeto o no detrás. Este tipo de procesos es conocido como POMDP (*Partially Observable Markov*

Decision Process, en inglés). En caso de que se empleen redes neuronales para la representación de las funciones de estado, el ruido de las observaciones puede ser manejado gracias a la capacidad de generalización de las mismas, por lo que no sería necesario utilizar los formalismos costosos y muy complicados de los POMDPs.

2.2. Aprendizaje del agente

Una vez aclarado cómo se modela un sistema de aprendizaje por refuerzo, es necesario explicar cómo puede aprender el agente. En primer lugar, este debe tener un objetivo bien definido. La ecuación 2.1 define el retorno (G_t) como la acumulación de todas las recompensas obtenidas en un periodo de tiempo. El objetivo del agente, en ese caso, sería tomar decisiones que maximizaran el retorno. Sin embargo, para darle una importancia mayor a las recompensas cercanas en el tiempo que a las esperadas en el largo plazo, y para evitar divergencia en la suma en casos no episódicos, es práctica común utilizar un factor de descuento γ tal que $0 < \gamma < 1$ (véase la ecuación 2.2). Nótese que el retorno puede calcularse de manera recursiva.

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + ... + R_T \tag{2.1}$$

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} = R_{t+1} + \gamma G_{t+1} \tag{2.2}$$

Las acciones tomadas deben constituir una estrategia. Se le llama π a esa función de decisión, también conocida como política. Esta función decide qué acción tomar según el estado en el que se encuentra el entorno, y puede ser determinista, como en la ecuación 2.3, o estocástica, como en la ecuación 2.4. Por ejemplo, una política que maximiza la recompensa del siguiente paso es la denominada "greedy". El objetivo que se ha definido para el agente, que es maximizar el retorno, puede ser reinterpretado como aprender una política π que maximice el retorno total G. Para este fin existen tres enfoques principales: métodos basados en función de valor, métodos basados en políticas, y métodos *Actor-Critic*.

$$\pi(s) = a \tag{2.3}$$

$$\pi(a|s) = p(a|s) \tag{2.4}$$

2.2.1. Métodos basados en función de valor

En los métodos basados en función de valor se calcula una función que indica el «valor» de un estado. Esta función se suele hallar minimizando una función de error o pérdida. Una vez se tiene la

función de valor, se implementa una política basada en dicha función. Existen dos tipos de funciones de valor: funciones de valor del estado y funciones de valor de estado-acción. Las funciones de valor del estado (*state-value function*, en inglés) asignan a cada estado s un valor escalar $v_\pi(s)$ correspondiente al retorno esperado dada una política π (véase la ecuación 2.5). Las funciones de valor de estado-acción (*state-action value function*, en inglés) asignan a cada par estado-acción (s, a) un valor escalar $q_\pi(s, a)$ correspondiente al retorno esperado dada una política π que se siga una vez ejecutada la acción a en el estado s (véase la ecuación 2.6). Un ejemplo muy común es implementar una función de estado-acción y sobre ella utilizar una política *Greedy*, que elige la acción que ofrece una mayor recompensa.

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s, \pi] \tag{2.5}$$

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a, \pi] \tag{2.6}$$

Los métodos basados en función de valor son útiles para espacios discretos de pocas dimensiones, e incluso es posible hallar la política óptima π^* en algunos casos. Sin embargo, requieren de bastabte memoria para almacenar el valor de los estados o los pares estado-acción, por lo que no escalan bien y no son apropiados para espacios continuos. Algunos ejemplos de algoritmos: Q-learning [5], SARSA [6] y DQN [7].

2.2.2. Métodos basados en política

Los métodos basados en política aprenden una función parametrizada descrita en la ecuación 2.7, donde el vector de parámetros es $\theta \in \mathbb{R}^n$. La variante más común de los métodos basados en políticas es el grupo de los que usan gradiente de política (*policy gradient methods*, en inglés), caracterizados por la ecuación 2.8. Este tipo de métodos busca maximizar localmente una función escalar de rendimiento $J(\theta)$ calculando su gradiente $\nabla J(\theta)$ respecto del vector de parámetros θ y actualizando el vector en la dirección ascendente del vector gradiente; es decir, aplicando un ascenso de gradiente. La función de rendimiento suele ser el retorno esperado G. El parámetro α es la tasa de aprendizaje (*learning rate*, en inglés), un escalar que regula el tamaño de la corrección. Es un parámetro muy importante para el aprendizaje, ya que su valor puede tener efectos en la velocidad de éste y también evitar atascarse en mínimos locales.

$$\pi(a|s, \theta) = Pr\{A_t = a|S_t = s, \theta_t = \theta\} \tag{2.7}$$

$$\theta_{t+1} = \theta_t + \alpha \nabla J(\theta_t) \tag{2.8}$$

Esta clase de métodos funciona mejor que los métodos basados en función de valor cuando se tienen espacios continuos o altamente dimensionales, aunque a veces presentan inestabilidad en el proceso de aprendizaje debido a cambios grandes en el cálculo del gradiente. Algunos ejemplos de algoritmos de este tipo son: REINFORCE [8], TRPO (*Trust Region Policy Optimization*) [9], PPO (*Proximal Policy Optimization*) [10].

2.2.3. Métodos *Actor-Critic*

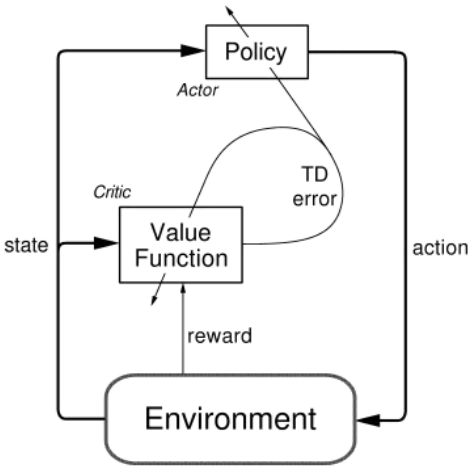


Figura 2.4: Estructura de un agente de tipo *Actor-Critic*. Consta de dos componentes. El actor decide las acciones que se aplican al entorno. El crítico evalúa las acciones del actor, y sustituye la señal de recompensa por una señal filtrada.

Los métodos *Actor-Critic* mezclan los métodos basados en valor con los métodos basados en políticas. Como muestra la figura 2.4 (tomada de [3]), se tienen dos componentes: el actor y el crítico. El actor es el encargado de decidir qué acciones se toman, mientras que el crítico evalúa dichas acciones. El crítico implementa un método basado en valor que utiliza la observación del estado y la recompensa. El actor implementa un método basado en política, pero en lugar de recibir el estado y la recompensa, (como se haría en un método basado en política puro), recibe el estado y una señal de evaluación emitida por el crítico.

Esta clase de métodos funciona muy bien cuando se tienen espacios continuos, y presentan un aprendizaje más estable que los métodos basados en políticas, ya que no se realimenta el actor directamente con la recompensa, que puede presentar una gran varianza. El nivel de complejidad es mayor que en otros métodos, sin embargo, requieren un buen ajuste de los parámetros de entrenamiento. Algunos ejemplos de estos algoritmos son: A3C [11], DDPG [12], TD3 [13] y SAC (*Soft Actor-Critic*) [14].

El algoritmo que se utiliza en los experimentos de este proyecto es SAC. Se caracteriza por ser un algoritmo *Off-Policy*, es decir, que puede utilizar durante el aprendizaje cualquier política aunque no

sea la óptima, por aprender una política estocástica y por tener como objetivo maximizar tanto la recompensa como la entropía de la política, lo que le lleva a mantener su capacidad de exploración. Véase la ecuación 2.9, donde $J(\pi)$ es la función objetivo a maximizar, $r(s_t, a_t)$ es la recompensa por realizar la acción a_t en el estado s_t, $\mathcal{H}(\pi(\cdot|s_t))$ representa la entropía de la política π en el estado s y α es el coeficiente de temperatura.

$$J(\pi) = \sum_{t=0}^{\infty} \mathbb{E}_{(s_t,a_t)\sim\rho_\pi} [r(s_t, a_t) + \alpha\mathcal{H}(\pi(\cdot|s_t))] \tag{2.9}$$

2.3. Estado del arte: Frameworks de aprendizaje por refuerzo

En esta sección se comentan los diferentes *frameworks* que están relacionados con este trabajo. Dichos proyectos proveen entornos simulados para trabajar con robots manipuladores en tareas de aprendizaje por refuerzo. Se analizan las características de cada uno, señalando sus puntos fuertes y débiles.

2.3.1. dm_robotics_panda

dm_robotics_panda [15] es una librería basada en el software de Google *DeepMind* y el simulador de físicas *MuJoCo* [16]. Véase la figura 2.5. Implementa HIL (*Hardware in the loop*), permitiendo controlar al mismo tiempo el robot real y la simulación, que se realimenta con la información proveniente de los sensores reales. Esto satisface la necesidad de desarrollar código para adaptar el agente entrenado en simulación al control del robot físico. Un problema que presenta es que la clase software que representa el entorno no pertenece a ningún estándar, por lo que no es directamente compatible con librerías de aprendizaje por refuerzo como SB3 (*Stable-Baselines3*). Esto implica que el usuario tiene que crear un programa que haga de puente entre dm_robotics_panda y la librería, o implementar los algoritmos de aprendizaje por refuerzo desde cero. Además, no incluye modelos del entorno predefinidos, por lo que el usuario debe diseñar los suyos.

2.3.2. panda-gym

El framework panda-gym [17], por otra parte, incluye entornos diseñados con la librería *Gymnasium* [18] y el simulador utilizado es *PyBullet*. Véase la figura 2.6. El código es muy sencillo de usar e incluye entornos con tareas variadas (*Reach*, *Push*, *Slide*, *PickAndPlace*, *Stack* y *Flip*). Es un entorno muy popular, y existen multitud de modelos accesibles en *Hugging Face*[1], una página web donde cualquier usuario puede compartir y descargar modelos de inteligencia artificial de código abierto. Sin embargo, la última versión (v3) todavía no es compatible con SB3 y existen diferentes errores que dificultan su uso.

[1] https://huggingface.co/

2.3.3. panda_mujoco_gym

panda_mujoco_gym [19] está basado en el simulador *MuJoCo* y contiene varios entornos de *Gymnasium* con tareas predeterminadas (véase la figura 2.7). Es fácil de utilizar con SB3. Sin embargo, cuenta solamente con tres entornos (*Push*, *Slide* y *PickAndPlace*) que buscan posicionar un cubo en un lugar de diferentes maneras, y no cuenta con un entorno de tipo *Reach*, que es la tarea más sencilla para empezar a trabajar creando modelos para manipulación robótica.

2.3.4. gymnasium-robotics

gymnasium-robotics [20] también utiliza el simulador *MuJoCo*, y destacan los entornos *Fetch-v4*. En lugar del Franka Emika Panda, cuentan con otro robot que también tiene 7 grados de libertad (véase la figura 2.8). Esto podría dificultar la transferencia de la política aprendida de un manipulador a otro. Al ser entornos de *Gymnasium*, son fáciles de utilizar con SB3. Las tareas disponibles son *Fetch*, *Push*, *Reach* y *Slide*.

2.3.5. robosuite

robosuite [21] ha sido desarrollado por la Universidad de Stanford y está basado en *MuJoCo* (véase la figura 2.9). Incluye una amplia variedad de entornos preparados para tareas de manipulación con una amplia gama de robots, entre los que se encuentra el Franka Emika Panda. Los entornos no son de *Gymnasium*, pero su API es muy parecida y existe un *wrapper* que les permite comportase de tal manera, haciendo que sea compatible con librerías como SB3.

De todos los *frameworks* vistos anteriormente, en este trabajo se utiliza *dm_robotics_panda*. El motivo principal es que es el único *framework* que incluye la funcionalidad de controlar un manipulador real, lo que supone una amplia ventaja respecto del resto de opciones. Además, presenta una arquitectura modular que permite configurar en detalle el entorno, así como una simulación de alta fidelidad basada en *MuJoCo*.

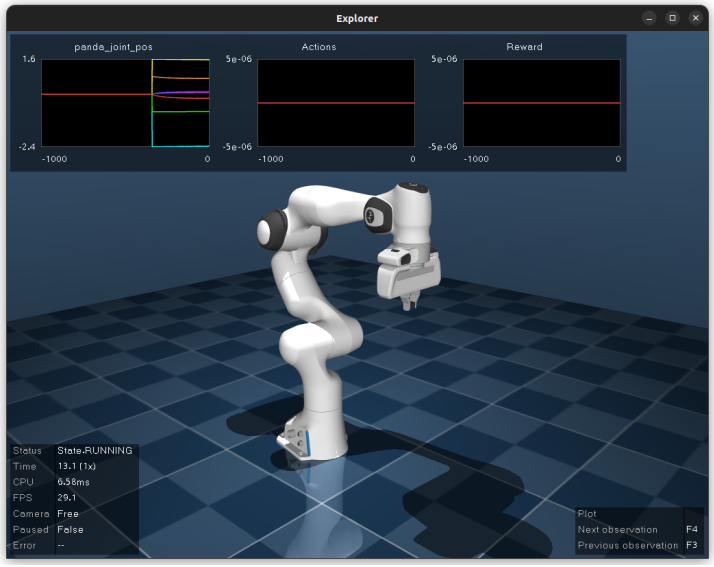


Figura 2.5: Imagen del entorno de simulación del software dm_robotics_panda. El entorno incluye un robot manipulador Franka Emika Panda situado en el centro de un mapa con una superficie plana.

(Fuente: https://jeanelsner.github.io/dm_robotics_panda/_images/gui.png, 19/05/2025)

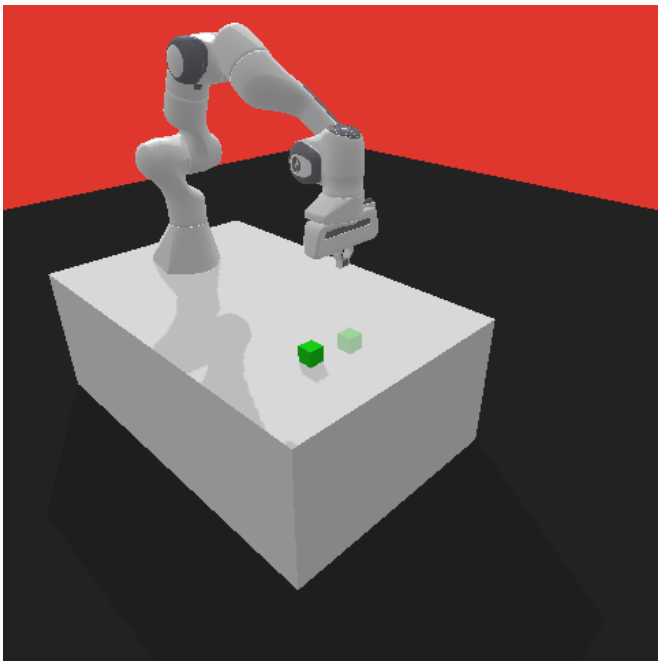


Figura 2.6: Imagen del entorno de simulación del software panda-gym. Un manipulador Franka Emika está montado en una plataforma, en la que se encuentra un objeto cúbico de color verde que tiene que llevar a un punto objetivo.

(Fuente: https://panda-gym.readthedocs.io/en/latest/_images/opengl.png, 19/05/2025)

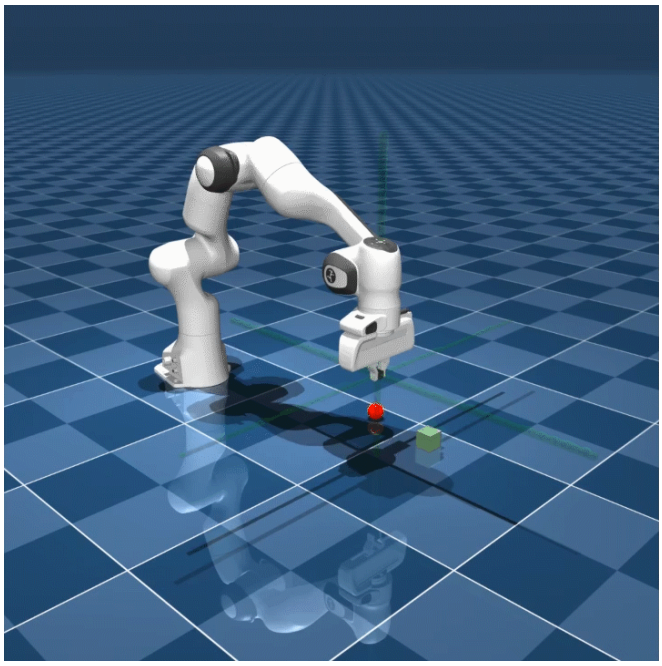


Figura 2.7: Imagen del entorno de simulación del software panda_mujoco_gym. Un robot Franka Emika Panda está situado en el centro de una superficie plana, y hau un objeto cúbico de color verde.

(Fuente https://github.com/zichunxx/panda_mujoco_gym/blob/master/docs/push.gif, 19/05/2025)

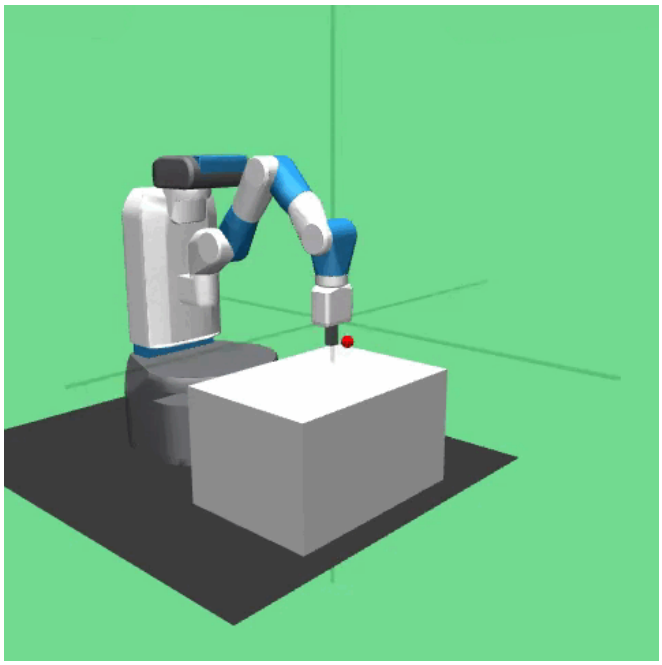


Figura 2.8: Imagen del entorno de simulación del software gymnasium-robotics. Incluye un manipulador genérico con 7 grados de libertad.

(Fuente: https://robotics.farama.org/_images/reach.gif, 19/05/2025)

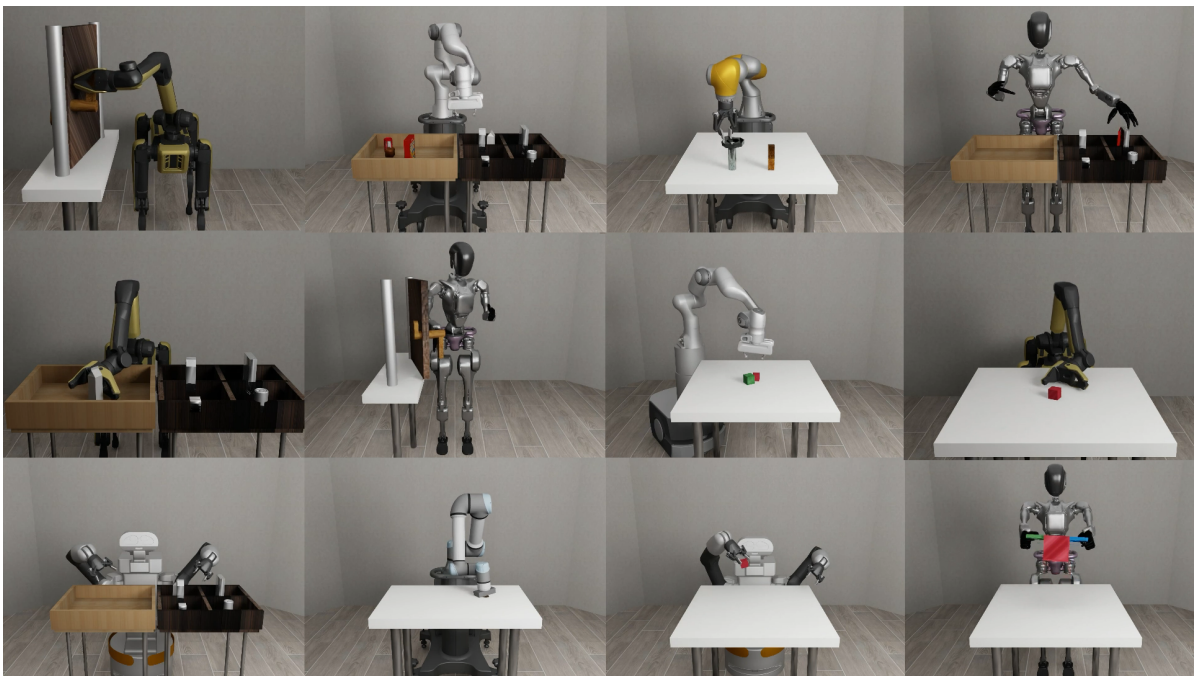


Figura 2.9: Imagen de varios entornos de simulación del software *robosuite*. Incluye robots de tipo manipulador, cuadrúpedos, humanoides, además de mesas y objetos varios.

(Fuente: https://github.com/ARISE-Initiative/robosuite/blob/master/docs/images/gallery.png, 19/05/2025)

CAPÍTULO 3

Metodología

Contenido

En este capítulo se explican los detalles técnicos del funcionamiento del *framework*. En primer lugar, se presentan las dependencias de software del proyecto. En segundo lugar, se describe la arquitectura del *framework*, y se detalla en profundidad la funcionalidad de cada componente. Finalmente, se presenta una guía para el usuario.

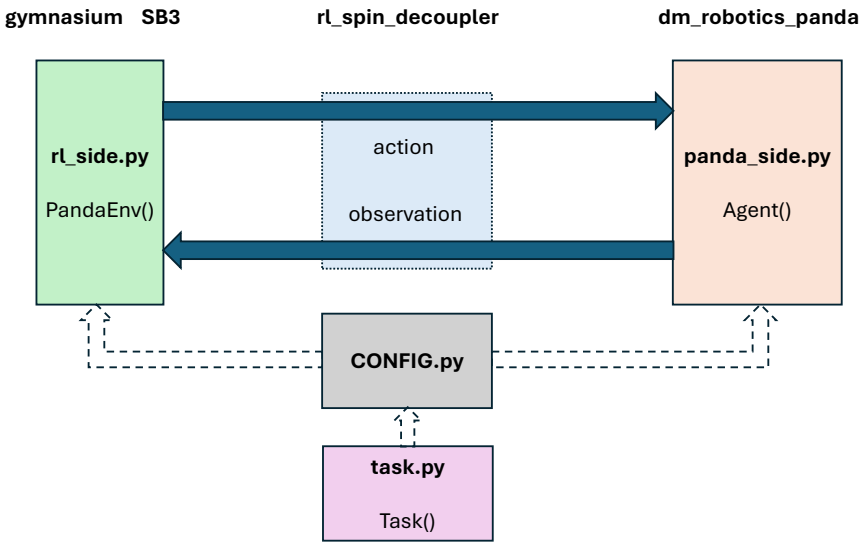


Figura 3.1: Arquitectura del *framework* franka_rl. Principalmente, consta de dos programas, *rl_side.py* y *panda_side.py*, que se comunican enviando acciones y observaciones. Los archivos *CONFIG.py* y *task.py* se encargan de la configuración.

El *framework* que presenta este trabajo ha sido desarrollado en Python y se puede encontrar en el repositorio de GitHub franka_rl[1] de uncore-team[2]. La figura 3.1 es un esquema general de la arquitectura del *framework*. En ella aparecen los archivos de código, las clases que implementan, las librerías que utilizan y se representa mediante flechas cómo se lleva a cabo la comunicación entre procesos. Este *framework* depende de una serie de librerías; las que han tenido un mayor impacto en el proyecto son dm_robotics_panda, Gymnasium, Stable-Baselines3 (SB3) y rl_spin_decoupler.

3.1. Dependencias de software

La herramienta *Gymnasium* permite modelar un entorno y su interacción con un agente siguiendo la teoría de aprendizaje por refuerzo. Proporciona una clase estandarizada *gymnasium.Env* con una serie de métodos que el usuario debe implementar. Los métodos más importantes son *step(action)* y *reset()*. El primero de ellos ejecuta la acción elegida por el agente, actualiza el entorno y devuelve al agente la observación, la recompensa y otro tipo de información auxiliar. El segundo método reinicia el entorno (generalmente de manera aleatoria) y devuelve la primera observación del nuevo episodio.

[1] https://github.com/uncore-team/franka_rl
[2] https://github.com/uncore-team

Código 3.1: Plantilla de una clase de gymnasium preparada para entrenar modelos con SB3.

```
class CustomEnv(gym.Env):
    """Custom Environment that follows gym interface."""

    def __init__(self, arg1, arg2, ...):
        super().__init__()
        # Define action and observation space
        # They must be gym.spaces objects
        self.action_space = ...
        self.observation_space = ...

    def step(self, action):
        ...
        return observation, reward, terminated, truncated, info

    def reset(self, seed=None, options=None):
        ...
        return observation, info

    def render(self):
        ...

    def close(self):
        ...
```

Stable-Baselines3 (SB3) contiene clases que implementan algoritmos de aprendizaje por refuerzo. Es compatible con *Gymnasium*, y permite entrenar un modelo con muy pocas líneas de código. Basta con instanciar el entorno, instanciar el modelo indicándole el entorno en el que va a estar, y llamar al método del modelo `learn(total_timesteps)`, indicando cuánto tiempo dura el entrenamiento (véase el código 3.2).

Código 3.2: Ejemplo de entrenamiento de un modelo con el algoritmo A2C en el entorno *CartPole-v1*.

```
import gymnasium as gym

from stable_baselines3 import A2C

env = gym.make("CartPole-v1", render_mode="rgb_array")

model = A2C("MlpPolicy", env, verbose=1)
model.learn(total_timesteps=10_000)

vec_env = model.get_env()
```

```
obs = vec_env.reset()
for i in range(1000):
    action, _state = model.predict(obs, deterministic=True)
    obs, reward, done, info = vec_env.step(action)
    vec_env.render("human")
    # VecEnv resets automatically
    # if done:
    #   obs = vec_env.reset()
```

Por otra parte, *dm_robotics_panda* proporciona un entorno de simulación en *MuJoCo* para el robot Franka Emika Panda y HIL (*Hardware in the loop*), que permite la ejecución simultánea del código en el robot real y en el virtual. El paquete proporciona una clase `PandaEnvironment` que actúa como el entorno, y es la encargada de manejar la interacción con la simulación y con el robot real. El usuario tiene que implementar por su cuenta una clase para el agente con el método `step(timestep)`, que tome la observación del estado y devuelva la acción seleccionada por el modelo. Esta interfaz es diferente a la de SB3 y *Gymnasium*. Por un lado, SB3 implementa la clase del agente, no es necesario que lo haga el usuario. Por otro lado, la clase del entorno `PandaEnvironment` no sigue el estándar de *Gymnasium*. Este último hecho impide una integración simple similar al ejemplo anterior del código 3.2. Una solución a este problema es utilizar `PandaEnvironment` y dm_robotics_panda únicamente para la simulación y la comunicación con el manipulador, creando un entorno de *Gymnasium* separado de `PandaEnvironment` y que se encargue del aprendizaje del agente. Separando ambas funcionalidades en dos clases diferentes es posible utilizar la interfaz de herramientas estandarizadas. Para implementar esta idea es necesario encontrar alguna manera de comunicar `PandaEnvironment` con el entorno encargado del proceso de aprendizaje.

El módulo *rl_spin_decoupler*[3] establece una comunicación asíncrona basada en *sockets* entre dos programas. Este paquete está pensado para tareas de aprendizaje por refuerzo en las que, por algún motivo, no sea posible integrar en un mismo bucle el entrenamiento o la inferencia de de un modelo, y la actuación del agente. Por ejemplo, cuando la frecuencia de control requerida por el agente sea superior a la del cálculo de la acción por el modelo, el agente deba ejecutar durante varios ciclos la última acción recibida mientras el modelo piensa en la próxima decisión. En el *framework* de este proyecto se utiliza rl_spin_decoupler para comunicar un proceso encargado del control del manipulador con otro proceso que se encarga de la toma de decisiones y el aprendizaje.

3.2. Arquitectura

En este apartado se explica el contenido y la funcionalidad de los principales archivos que conforman el *framework*, y que están relacionados como se indica en la figura 3.1.

[3] https://github.com/uncore-team/rl_spin_decoupler

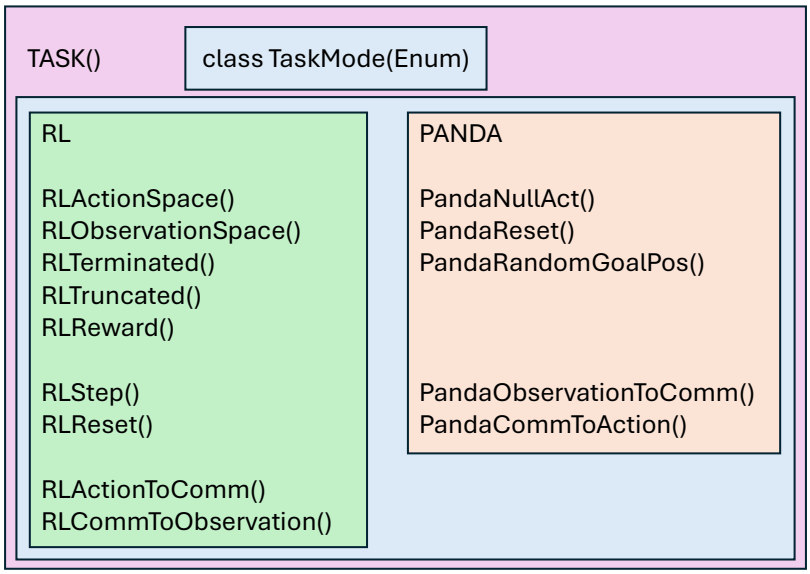


Figura 3.2: Esquema del contenido de task.py. El archivo contiene una clase de Python que incluye métodos para el modelo (RL) y para el control del robot (PANDA).

3.2.1. Definición de la tarea

En el archivo *task.py* se define la tarea de aprendizaje por refuerzo. El usuario puede crear su clase personalizada basada en la clase abstracta `Task`, implementando los métodos correspondientes. Esta clase, ilustrada en la figura 3.2, es importada posteriormente por los archivos *rl_side.py* y *panda_side.py*, que utilizan dichos métodos.

Por una parte, están los métodos correspondientes a *rl_side.py*, que definen el espacio de acciones, el espacio de observaciones, las condiciones de terminación y truncamiento, y la función de recompensa. Además, están el método `RLStep()`, que lleva la cuenta de los pasos de un episodio, `RLReset()`, que reinicia el episodio, y los métodos de comunicación `RLActionToComm()` y `RLCommToObservation()`, que adaptan el formato de las acciones y observaciones que se envían y reciben utilizando la librería *rl_spin_decoupler*.

Por otra parte, están los métodos correspondientes a *panda_side.py*. `PandaNullAct()` devuelve un vector de ceros del tamaño de la acción que requiere el controlador de *dm_robotics_panda*. `PandaReset()` reinicia el episodio. `PandaRandomGoalPos()` genera un punto en el espacio de manera aleatoria, que se utiliza como objetivo a alcanzar en cada episodio. Este método se puede llamar dentro de `PandaReset()`. Por último, están los métodos de comunicación `PandaObservationToComm()` y `PandaCommToAction()` que adaptan el formato de las acciones y observaciones que se envían y reciben utilizando la librería *rl_spin_decoupler*.

Además de los métodos explicados, la clase `Task` contiene una clase interna `TaskMode(Enum)`, donde se declaran los distintos modos de funcionamiento del *framework*. Los modos utilizados en este trabajo se explican más adelante.

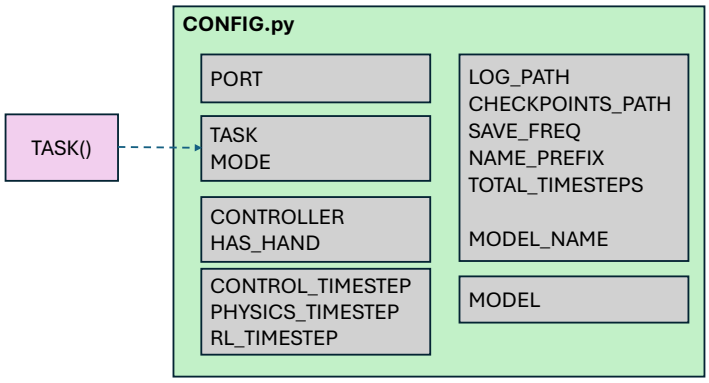


Figura 3.3: Esquema del contenido de CONFIG.py. Incluye parámetros para el control del robot, la tarea, el modo de funcionamiento del *framework* y el entrenamiento.

3.2.2. Configuración

El archivo *CONFIG.py*, cuyo esquema está en la figura 3.3, contiene los parámetros relativos a la ejecución que son susceptibles de ser modificados con frecuencia. De esta manera, no hay necesidad de modificar los archivos principales rl_side.py y panda_side.py, por lo que queda un proyecto de código más organizado.

PORT es el puerto utilizado para las comunicaciones con *rl_spin_decoupler*. TASK es la tarea definida en *task.py*, y MODE el modo de funcionamiento. CONTROLLER es el tipo de controlador de *dm_robotics_panda* utilizado para el manipulador, que puede ser control en velocidad cartesiana o velocidad articular. HAS_HAND indica si el manipulador tiene una herramienta, mano o pinza en el efector final.

Existen tres parámetros de tiempo[4]. PHYSICS_TIMESTEP es el periodo de simulación física discreta que utiliza *MuJoCo*. Debe ser el más pequeño de los parámetros de tiempo. CONTROL_TIMESTEP es el periodo de acción del agente de *dm_robotics_panda* sobre el robot simulado o real, y debe ser superior a PHYSICS_TIMESTEP, ya que que se pueden realizar acciones más rápido de lo que se simula la física. RL_TIMESTEP es el periodo con el que el modelo recibe una nueva observación y toma una nueva decisión. Debe ser superior a CONTROL_TIMESTEP. Esto implica que el agente ejecuta la misma acción durante varios ciclos de reloj, hasta que recibe una acción nueva.

Además, hay parámetros específicos del entrenamiento del modelo. LOG_PATH es el directorio en el que se almacenan los *logs*, es decir, la información del entrenamiento (pasos, recomensa, entropía, etc.). CHECKPOINTS_PATH es el directorio en el que se almacenan copias de seguridad del modelo durante el entrenamiento. SAVE_FREQ es la frecuencia, en pasos de entrenamiento, a la que se guardan los *checkpoints*. NAME_PREFIX es una cadena de caracteres que actúa como prefijo para el nombre de cada *checkpoint*, que además contiene el número de pasos del entrenamiento.

[4]El proyecto TYRELL, comentado en la introducción, está enfocado en la variación del tiempo de toma de decisiones de los modelos.

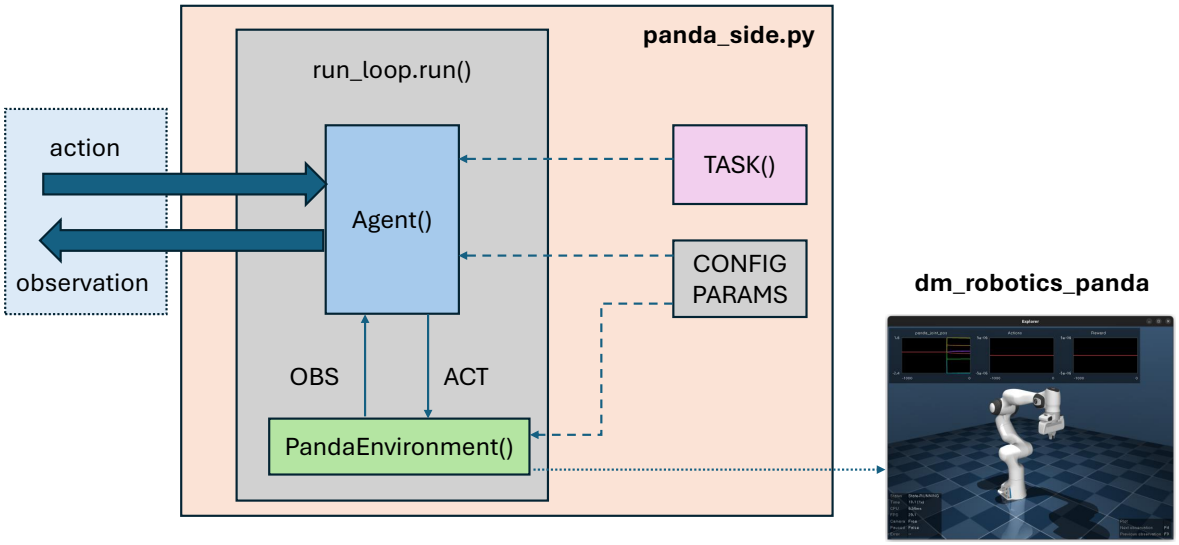


Figura 3.4: Esquema del contenido de panda_side.py. Describe un bucle en el que la clase del agente interactúa con el entorno `PandaEnvironment` y se comunica con rl_side.py.

TOTAL_TIMESTEPS es la cantidad de pasos que debe durar el entrenamiento. MODEL_NAME es el nombre que tendrá el modelo final resultante del entrenamiento.

Por último, cuando el modo de funcionamiento del *framework* no es para entrenar un modelo, sino para explotar un modelo ya entrenado, está el parámetro MODEL. Es el nombre del archivo que contiene el modelo.

3.2.3. Control del robot

El archivo *panda_side.py*, cuyo esquema está en la figura 3.4, se encarga del bucle de control del robot y la simulación. La clase `PandaEnvironment` es el entorno de *dm_robotics_panda*, que recibe acciones y devuelve observaciones. La clase `Agent` se ha implementado en este proyecto, y actúa como el agente, enviando acciones al entorno `PandaEnvironment` y recibiendo sus observaciones.

El interior de la clase del agente está ilustrado en la figura 3.5. El agente contiene una instancia de la clase `AgentSide`, de *rl_spin_decoupler*. Este objeto se encarga de comunicarse mediante *sockets* con el archivo *rl_side.py*, donde se eligen las acciones que realiza el robot. No debe confundirse el control del robot con la toma de decisiones. El agente de este archivo, a través de su método `step()`, actúa sobre el entorno y recibe una observación. Esa observación se le envía a *rl_side.py*, donde se decide ejecutar una acción. El agente recibe esa acción y la ejecuta durante un tiempo determinado, hasta que recibe una nueva orden. Todo esto se realiza dentro del bucle proporcionado por el método `run_loop.run()`, de *dm_robotics_panda*.

La clase interna `StepState(Enum)` tiene varios valores, que indican el estado en que se encuentra el agente. En función de dicho estado, el agente actúa de distintas maneras, como se indica en la figura 3.6. En el estado READYFORRLCOMMAND, el agente espera a recibir una acción a través

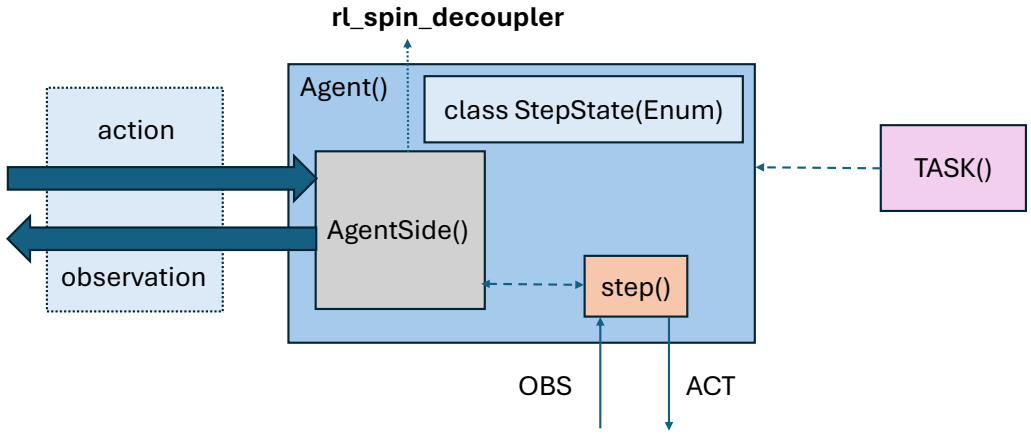


Figura 3.5: Esquema del contenido la clase `Agent` de panda_side.py. Incluye un objeto de tipo `AgentSide` que se encarga de las comunicaciones, y un método `step()` que se ejecuta en el bucle de control.

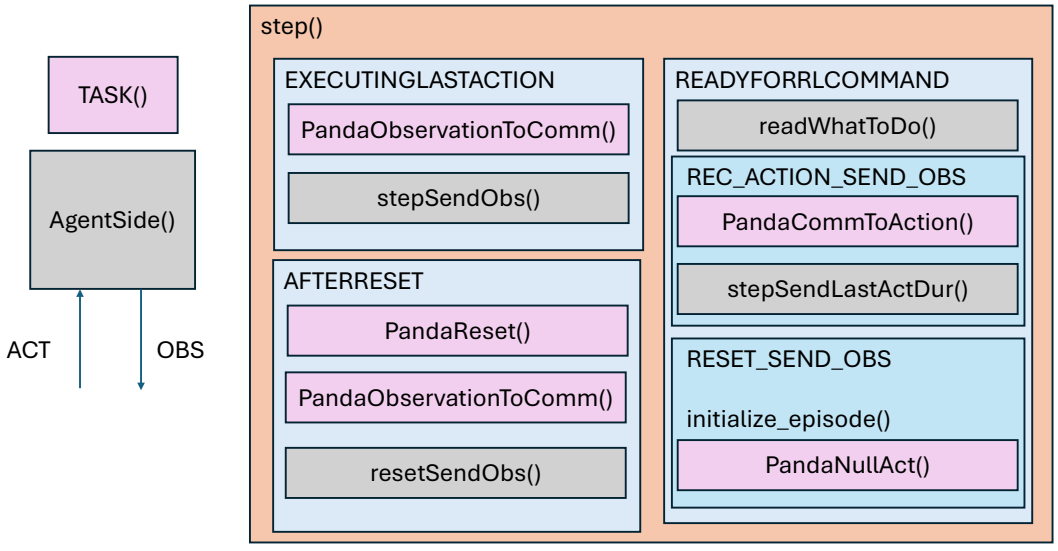


Figura 3.6: Esquema del método `step()` de la clase `Agent` de panda_side.py. Se describen las acciones que toma el agente en distintos estados.

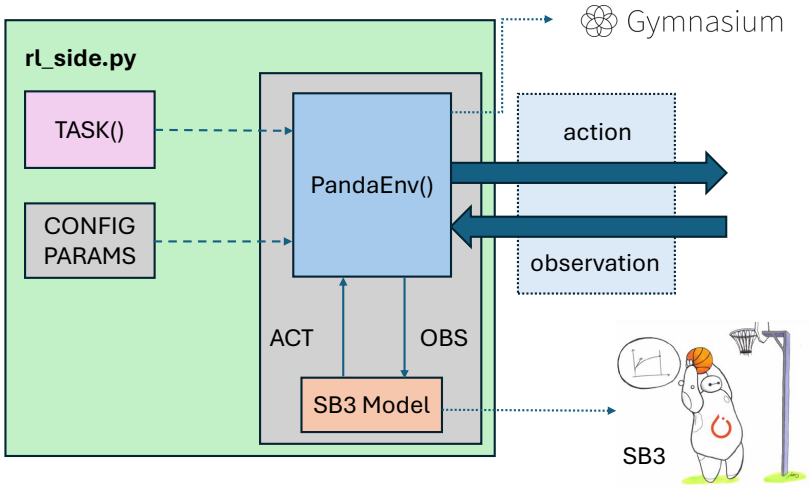


Figura 3.7: Esquema del contenido de rl_side.py. Incluye una clase `PandaEnv` que actúa como entorno de *Gymnasium* para el modelo de SB3.

de `AgentSide`. Cuando recibe esa acción, puede hacer dos cosas. Si está en mitad de un episodio, actualiza la acción a realizar y envía la duración de la última acción. Si comienza un nuevo episodio, reinicia el simulador con una posición aleatoria, y actualiza la acción con un valor nulo. En el estado EXECUTINGLASTACTION, el agente ejecuta la última acción recibida. Una vez pasa el periodo de control CONTROL_TIMESTEP, hay que tomar una nueva decisión, así que envía la última observación y cambia al estado READYFORRLCOMMAND. En el estado AFTERRESET, el agente genera un nuevo objetivo aleatorio con `PandaReset()` y envía la observación de comienzo del episodio.

3.2.4. Modelo

El archivo *rl_side.py*, cuyo esquema está en la figura 3.7, es donde se ejecuta el modelo que decide las acciones que ejecuta el manipulador. El modelo, basado en un algoritmo de la librería *SB3*, interactúa con *PandaEnv()*, una clase basada en un entorno de *gymnasium*. Esta clase[5] toma de la tarea TASK la definición del entorno (espacio de acciones, espacio de observaciones, recompensa, etc.). El interior de la clase *PandaEnv()* se muestra en la figura 3.8. El entorno contiene una instancia de la clase *RLSide()*, un servidor de *rl_spin_decoupler*. Este servidor recibe observaciones de *panda_side.py*, se las da al modelo, y envía la acción elegida.

La interacción entre el entorno y el modelo se realiza a través de los métodos `step()` y `reset()`. La funcionalidad de ambos métodos queda descrita en la figura 3.9. El método `reset()` reinicia el episodio, espera a recibir una observación y se la da al modelo. El método `step()` se ejecuta en cada paso del episodio. Comienza actualizando la cuenta de pasos con el método `RLStep()`. Seguidamente, envía la acción que ha elegido el modelo y espera a recibir una observación. Cuando recibe la

[5] No confundir *PandaEnv()*, de *rl_side.py*, con *PandaEnvironment()*, de *panda_side.py*.

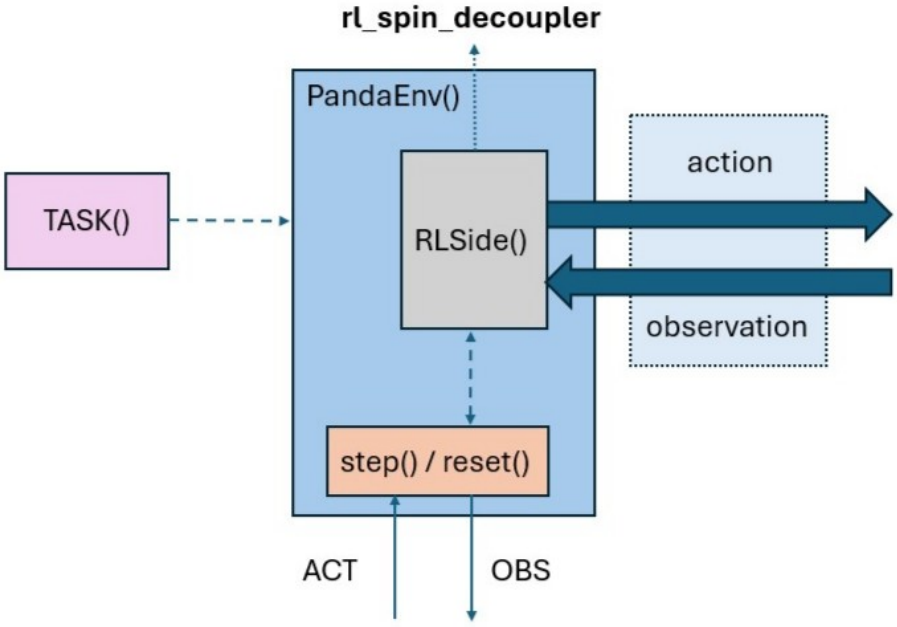


Figura 3.8: Esquema del contenido la clase `PandaEnv` de rl_side.py. Un objeto de tipo `RLSide` se encarga de las comunicaciones con *panda_side.py*, y los métodos `step()` y *reset()* de la interacción con el entorno.

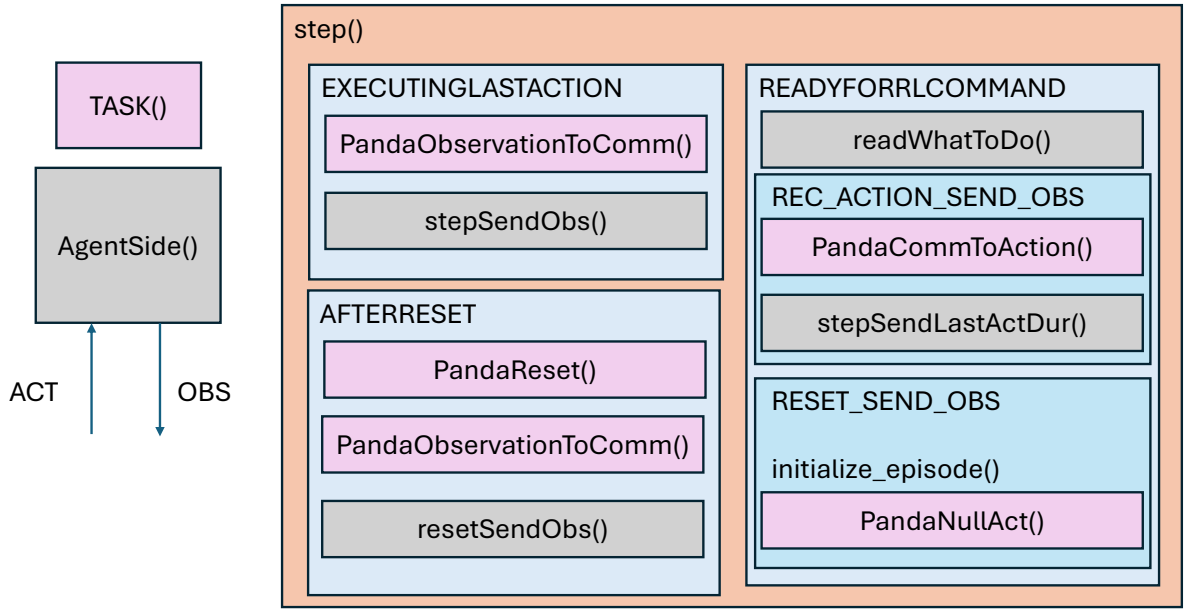


Figura 3.9: Esquema del método `step()` de la clase `PandaEnv` de rl_side.py.

observación, comprueba si el episodio ha terminado y calcula la recompensa. Esta información se la da al modelo, que elige una nueva acción para el siguiente paso.

3.2.5. Modos de funcionamiento

En este proyecto se han implementado tres modos de ejecución diferentes para el *framework*: LEARN, TEST y TEST_GUI. Estos modos se declaran en `TaskMode(Enum)`, la clase interna de `TASK`. La selección del modo de funcionamiento se puede hacer en el parámetro MODE del archivo de configuración.

El primero de ellos es LEARN. Está pensado para el entrenamiento de un nuevo modelo. Los parámetros relativos al aprendizaje (LOG_PATH, CHECKPOINTS_PATH, SAVE_FREQ, NAME_PREFIX, TOTAL_TIMESTEPS y MODEL_NAME) se pueden modificar en el archivo de configuración. Se debe utilizar el archivo rl_side.py.

El segundo modo de funcionamiento es TEST. Su finalidad es explotar el modelo, ejecutando en bucle varios episodios con una posición objetivo aleatoria. Se debe utilizar el archivo test.py, y la ruta del archivo que contiene el modelo se puede definir en el parámetro MODEL del archivo de configuración.

El tercer y último modo de funcionamiento es TEST_GUI. Al igual que en TEST, el objetivo es comprobar el funcionamiento de un modelo, pero solamente hay un episodio que no tiene fin en el que el usuario elige la posición objetivo a través de una interfaz gráfica. Puede verse la interfaz en la figura 3.10. Concretamente, en este proyecto se ha trabajado con tareas de tipo *Reach*, en las que el objetivo es alcanzar una pose con el efector final, por lo que la interfaz ha sido diseñada para introducir coordenadas (X, Y, Z). Además, tiene un gráfico tridimensional en el que se muestra el vector de error, que corresponde a la diferencia entre la posición objetivo y la posición del efector final. Se debe utilizar el archivo test_gui.py, y la ruta del archivo que contiene el modelo se puede definir en el parámetro MODEL del archivo de configuración.

3.3. Guía de uso

Para utilizar este *framework* hay que seguir una serie de pasos. En primer lugar, hay que configurar el entorno de trabajo:

1. Instalar Ubuntu 20.04 en un ordenador.

2. Crear un directorio vacío para trabajar, el *workspace*. Se puede hacer ejecutando el comando `mkdir <workspace-name>`.

3. Dentro del *workspace*, crear un entorno virtual de Python ejeutando el comando `python -m venv .venv` e instalar dependencias indicadas en el apartado 3.1.

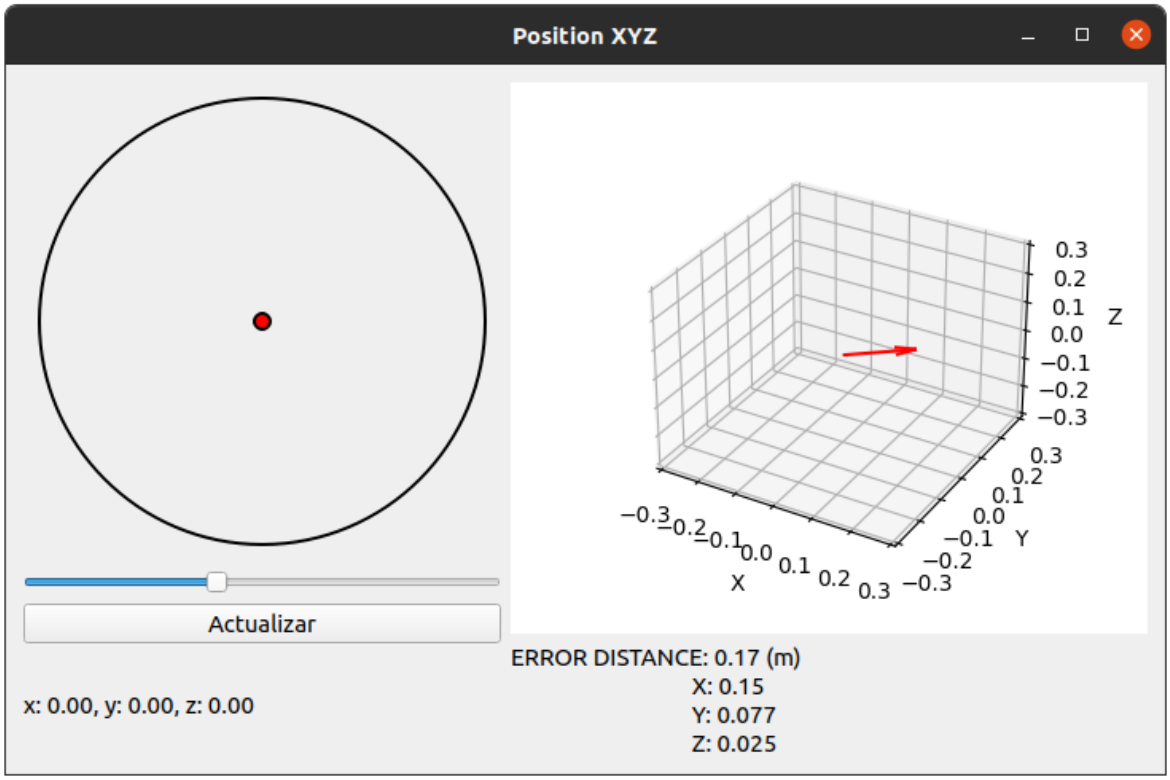


Figura 3.10: Interfaz gráfica para el modo de funcionamiento TEST_GUI. La parte izquierda está dedicada al control de la posición objetivo. La parte derecha es un gráfico del vector de error en posición cartesiana.

4. Clonar el repositorio de GitHub *rl_spin_decoupler* en el interior del *workspace*, ejecutando el comando `git clone <URL>`.

5. Realizar una sutil modificación en *rl_spin_decoupler*: en el archivo *rl_spin_decoupler/spindecoupler.py* hay que añadir un punto para importar de manera relativa la librería de *sockets*. Véase el código 3.3.

6. Copiar los archivos de código en la carpeta *template* del repositorio franka_rl. Al principio de algunos archivos puede haber una línea similar a `sys.path.append(os.path.abspath("../../"))`. Esta línea se utiliza para incluir el paquete rl_spin_decoupler, y habrá que añadir **../** las veces que sea necesario en función de la profundidad del árbol de carpetas con el que se trabaje. También puede clonarse el repositorio en el el directorio raíz del espacio de trabajo.

7. En task.py, crear una clase que herede de `Task` e implementar todas las funciones necesarias para definir completamente la tarea.

Código 3.3: Modificación del archivo *rl_spin_decoupler/spindecoupler.py*.

```
    from enum import Enum
    from .socketcomms.comms import ClientCommPoint,ServerCommPoint
```

Una vez configurado el espacio de trabajo, hay que seguir una serie de pasos para ejecutar el código:

1. Modificar los parámetros correspondientes en CONFIG.py.
2. Abrir dos terminales en el ordenador.
3. Activar el entorno virtual de Python en cada uno de los terminales ejecutando el comando `source .venv/bin/activate`.
4. En el primer terminal, ejecutar el programa con el servidor de rl_spin_decoupler, que será rl_side.py, test.py o test_gui.py dependiendo del modo seleccionado. Por ejemplo, `python test_gui.py`.
5. Esperar a que aparezca un mensaje señalando que el servidor está listo, similar a "Server comm point listening".
6. En el segundo terminal, ejecutar el cliente de rl_spin_decoupler, `python panda_side.py`. El resultado debe ser similar a la figura 3.11.
7. En caso de estar en el modo LEARN, esperar a que el modelo termine de entrenar. Si el modo de ejecución es TEST o TEST_GUI, simplemente observar el comportamiento del robot o interactuar con la interfaz gráfica.

Una vez finalizado el entrenamiento de un modelo, los resultados se pueden visualizar ejecutando el comando `tensorboard -logidr <path>`. Este comando abre un servidor en el puerto 6006 al que se puede acceder a través de un navegador web como Google Chrome. Solamente hay que escribir `localhost:6006` en la barra de búsqueda. Véase el ejemplo de la figura 3.12.

En la carpeta *utils* del repositorio *franka_rl* está el código *calculate_training_time.py*, que permite hallar el tiempo de establecimiento del entrenamiento de un modelo. Dicho tiempo de establecimiento está definido como el número de pasos de entrenamiento hasta que se alcanza el 95 % del valor máximo de la recompensa sin volver a bajar de dicho valor. El programa devuelve el resultado en forma de gráfica, y se ha utilizado en los experimentos que se realizan en la sección 4; por ejemplo, la figura 4.7.

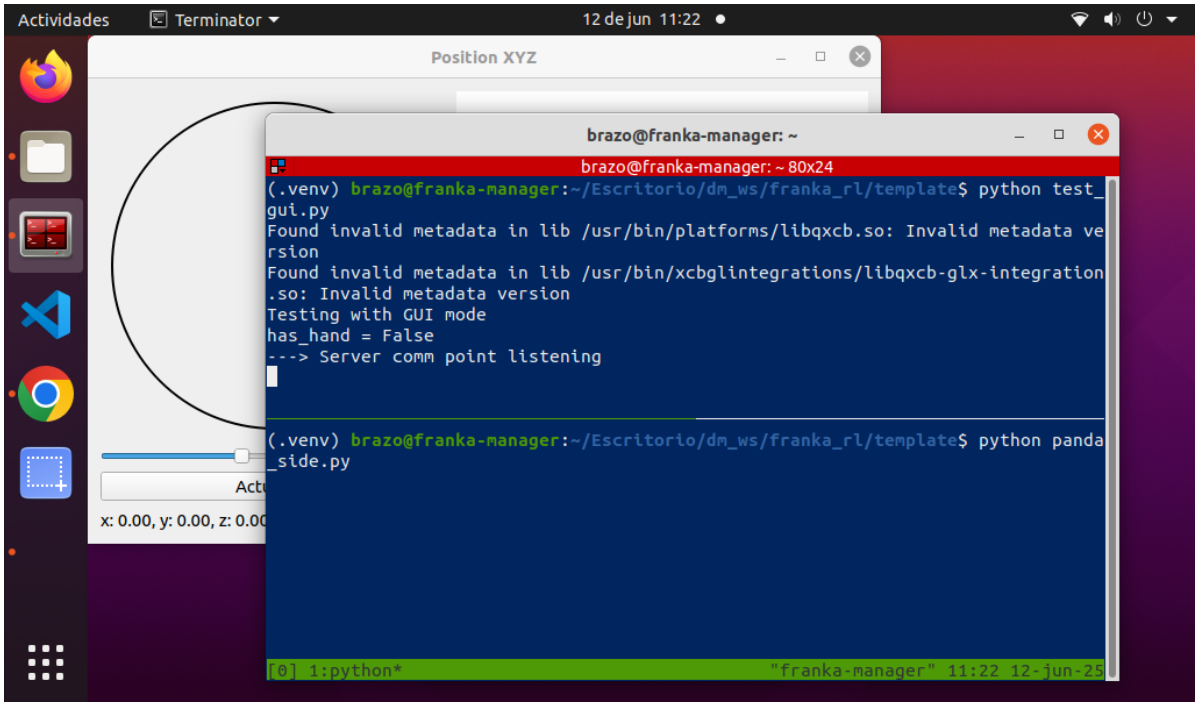


Figura 3.11: Imagen de los dos terminales que se deben ejecutar para utilizar el *framework*.

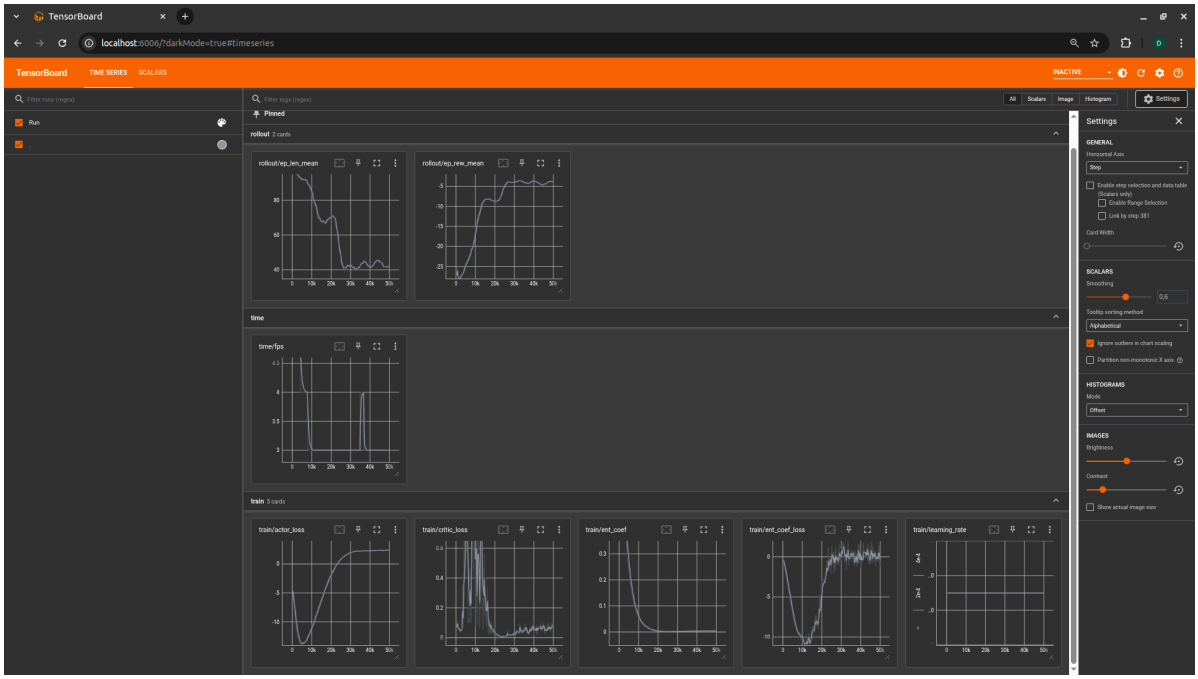


Figura 3.12: Interfaz gráfica que permite visualizar los resultados con *tensorboard*.

CAPÍTULO 4

Experimentos y resultados

Contenido

En este capítulo se pone a prueba el funcionamiento del *framework* desarrollado. Se llevan a cabo una serie de experimentos de entrenamiento de modelos en simulación. Tras el entrenamiento, se graban vídeos del comportamiento de los modelos y se obtienen métricas para comprender lo acontecido y evaluar su rendimiento. Cada experimento está acompañado de imágenes relevantes de los vídeos.

4.1. Entorno experimental

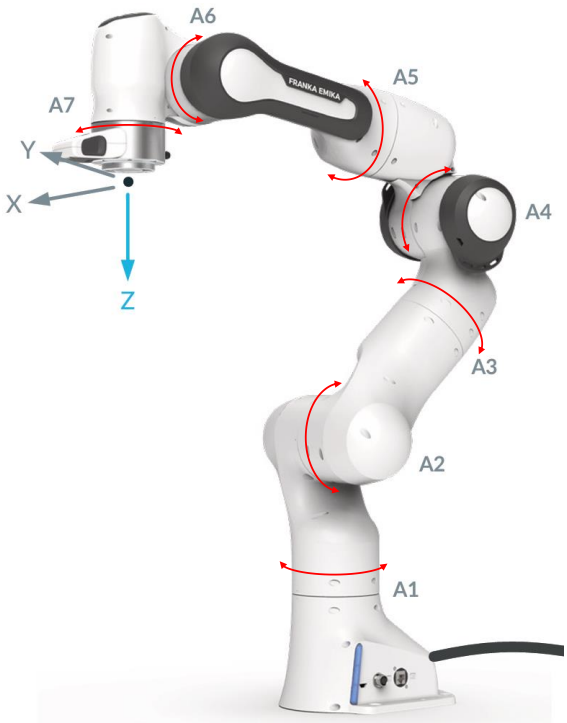


Figura 4.1: Manipulador Franka Emika Panda.

Figura 4.2: En la imagen aparece el robot manipulador Franka Emika Panda. Esta imagen ha sido obtenida de su hoja de datos.

El robot Franka Emika Panda de la figura 4.1 es el cobot que se utiliza en este trabajo. Posee 7 grados de libertad, por lo que es un robot redundante. Esta característica lo hace ideal para proyectos de investigación. Según su hoja de datos, es capaz de levantar $3kg$ en la posición más desfavorable (completamente extendido), tiene un alcance de $855mm$ y puede desplazar su efector final a una velocidad máxima de $2m/s$. Cada articulación cuenta con sensores de par, que permiten realizar control de fuerza. Además, tiene mecanismos de seguridad que detienen el robot si se detectan fuerzas excesivas que puedan dañar el hardware o las personas y objetos cercanos. El manipulador puede ser controlado mediante el sistema FCI (*Franka Control Interface*), que se encarga de recibir comandos y enviar la información de los sensores desde el ordenador propio del manipulador al ordenador de control remoto. Para mayor detalle, véase la figura 4.3. En este proyecto, el software *dm_robotics_panda* se encarga de las comunicaciones con el FCI. El usuario solamente necesita entrar en la página DESK con la dirección IP del robot (https://<ip>/desk/) para desactivar los frenos de las articulaciones, como se muestra en las figuras 4.4a y 4.4b, y activar el FCI, como se muestra en la figura 4.4c.

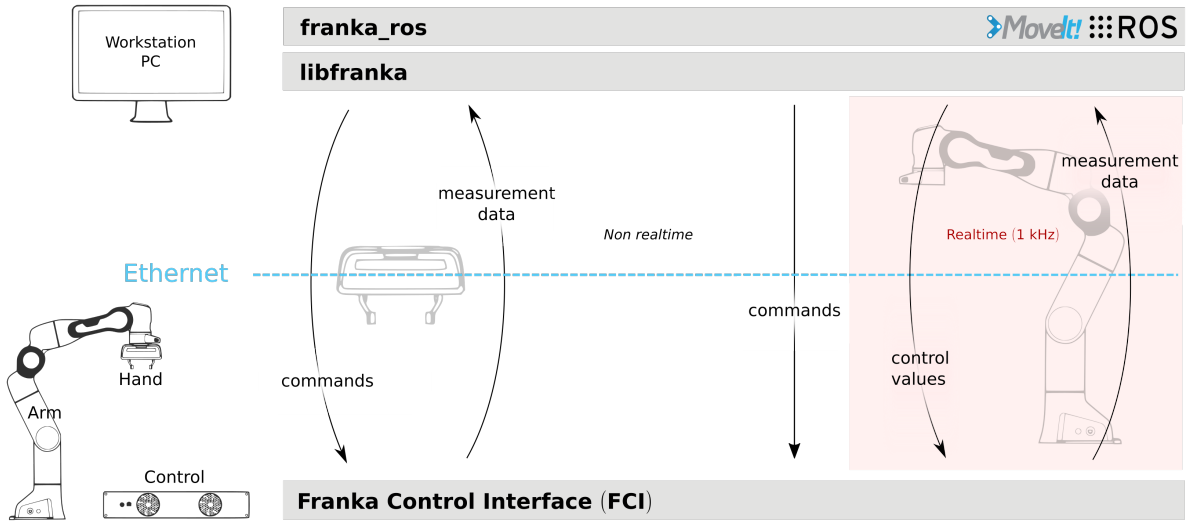
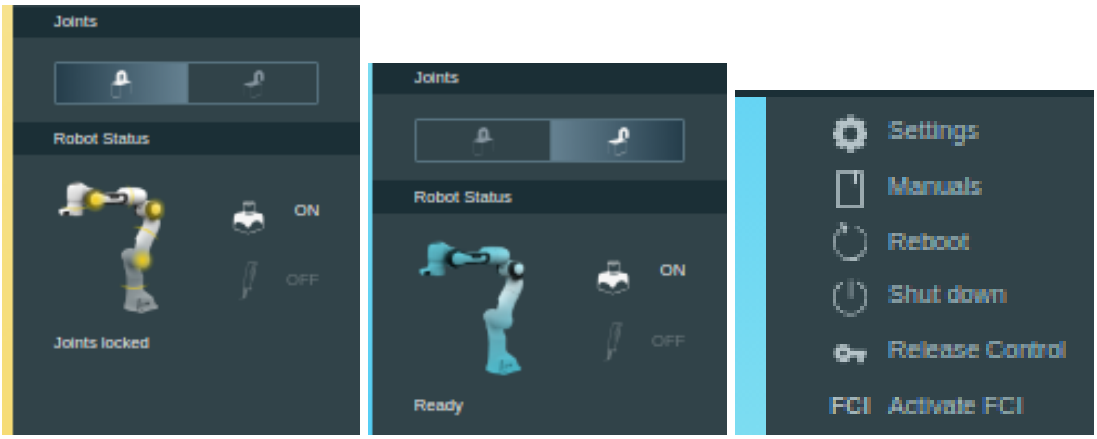


Figura 4.3: Arquitectura del sistema FCI.

(Fuente: https://frankaemika.github.io/docs/overview.html, 24/05/2025)



(a) (b) (c)

Figura 4.4: Acceso mediante la página DESK al control del robot Franka. Desde ella, se pueden controlar los frenos articulares y el FCI. (a) Imagen de los frenos articulares activados. (b) Imagen de los frenos articulares desactivados. (c) Imagen del botón con el que se activa el FCI.

4.2. Protocolo de experimentación

Métrica	modelo
Controlador	-
N	-
$\mathscr{A}$	-
$\mathscr{O}$	-
μ (m)	-
max (m)	-
σ (m)	-
Observaciones	-

Tabla 4.1: Tabla del protocolo de experimentación con las características y métricas a rellenar de un modelo.

En todos los experimentos, la tarea a realizar es de tipo *Reach*. Esto quiere decir que el objetivo es que el robot alcance una posición o *pose* cartesiana con su efector final. Se considera que el origen de coordenadas global $(X, Y, Z) = (0, 0, 0)$ está situado en la base del robot. El algoritmo utilizado para el agente es SAC, de tipo *Actor-Critic*. Podría utilizarse cualquier otro (PPO, A2C, etc.), pero queda fuera del alcance de este proyecto la comparación de diferentes algoritmos. Para obtener un modelo funcional, la técnica que se aplica fundamentalmente es *reward shaping*. Consiste en cambiar la función de recompensa definida en la tarea en busca de aportar la máxima información útil posible al modelo. Todos los experimentos de aprendizaje se llevan a cabo en simulación; entrenar en el robot supondría que podría chocar con algún objeto y sufrir daños. En todos los experimentos realizados se utilizan los siguientes parámetros temporales:

- PHYSICS_TIMESTEP: 0.002 s.
- CONTROL_TIMESTEP: 0.05 s.
- RL_TIMESTEP: 0.1 s.

Se graban vídeos de los modelos en modo TEST o TEST_GUI una vez termine el entrenamiento. Los vídeos de experimentos realizados en simulación se graban con la herramienta *OBS Studio*. Los vídeos de experimentos realizados con un robot real se graban en el laboratorio con la cámara de un teléfono móvil.

En los experimentos de entrenamiento en simulación se deben evaluar los modelos resultantes. Para ello, se debe cumplimentar la tabla 4.1, con la información de cada experimento. Si algún dato no se tiene, debe ser marcado con un guión (-). Las primeras filas son un resumen de las características del modelo, y las últimas evalúan su rendimiento. Las métricas de distancia al objetivo se obtienen a partir de muestras tomadas manualmente en el modo TEST_GUI. Se debe recorrer de manera aleatoria el espacio de trabajo. Como el efector final puede no quedarse completamente quieto,

difícilmente se puede tomar el valor de distancia, que cambia múltiples veces por segundo. Para abordar este problema de oscilaciones, cuando el efector final alcance el objetivo, se debe esperar unos segundos y realizar la toma de datos del error mostrado en la interfaz gráfica de forma manual.

4.3. Experimentos en simulación

4.3.1. Experimento 1: primer modelo funcional de control en velocidad cartesiana, reach2.

El primer experimento en proporcionar un modelo que ha aprendido alcanzar un punto en el espacio ha sido reach2, cuya tarea está definida en la clase TaskReach2. El espacio de observaciones consiste en un diccionario con varias entradas. Véase la ecuación 4.1. La primera entrada es **r**, la posición relativa entre el efector final y el objetivo medida en metros. Es un vector de tres dimensiones (X, Y, Z). La segunda entrada es d, la distancia entre el efector final y el objetivo. Es el módulo del vector de posición relativa, un escalar en metros. La tercera entrada es f, la magnitud de la fuerza resultante en el efector final medida en N. En cuanto a la actuación, se ha utilizado el controlador en velocidad del efector final de dm_robotics_panda. El espacio de acciones de la ecuación 4.2 es un vector **a** de tres dimensiones (X, Y, Z) con valores de velocidad en m/s. La orientación no se controla, siempre se envía una consigna de velocidad angular nula. Por otra parte, el episodio termina si el efector final se encuentra a una distancia inferior a un umbral, o si se supera un límite de fuerza (por motivos de seguridad). Si estas condiciones no se cumplen, pero el episodio se alarga demasiado, existe un límite de pasos, y si este es alcanzado, el episodio se trunca. En cuanto a la función de recompensa (véase la ecuación 4.3), esta tiene en cuenta la distancia al objetivo y la fuerza ejercida en el efector final. En condiciones normales, hay una penalización proporcional a la distancia. Esto actúa como incentivo para acercarse al objetivo. Si la distancia es inferior a un umbral, hay una recompensa positiva. Si se excede el límite de fuerza permitida, hay una penalización y termina el episodio. Por último, si el episodio se prolonga excesivamente y acaba truncándose, hay una gran penalización. De esta manera, se penaliza un modelo que tarda demasiado en cumplir su objetivo. Otros parámetros que se han utilizado en la tarea:

- max_steps: 300. Es el número máximo de pasos que puede durar un episodio.
- d_{min}: 0.2 (m). Es la distancia umbral que determina cuándo el efector final ha llegado al objetivo.
- F_{max}: 40 (N). Es la fuerza máxima permitida. Si se supera, termina el episodio.
- v_{max}: 0.5 (m/s). Es la velocidad máxima permitida del efector final en cada eje del espacio cartesiano. Véase el espacio de acciones en la ecuación 4.1.
- f_{max}: 100 (N). Es la magnitud máxima de fuerza que se puede medir en el espacio de observaciones. Véase el espacio de observaciones en la ecuación 4.1.

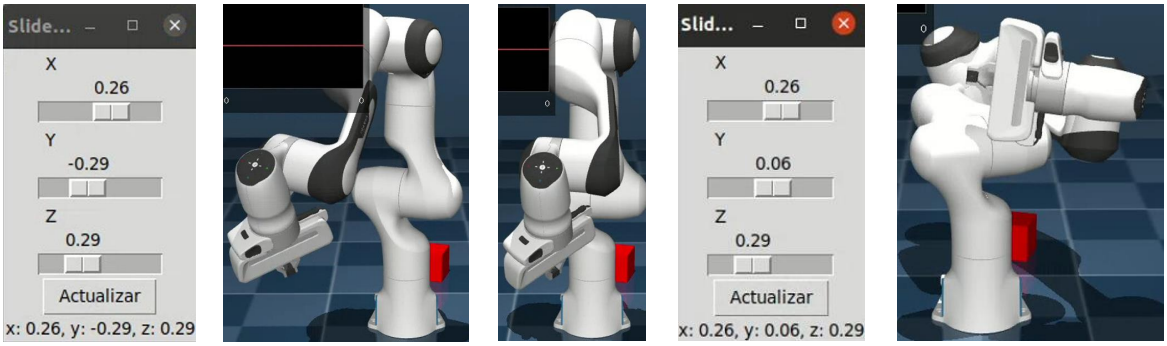


Figura 4.5: Imágenes del vídeo de reach2 en modo TEST_GUI. Las capturas 2 y 3 corresponden a un mismo punto objetivo. La captura 5 muestra la colisión del manipulador consigo mismo.

$$\mathcal{O} = \left\{(\mathbf{r}, d, f) \in \mathbb{R}^3 \times \mathbb{R} \times \mathbb{R} \;\middle|\; \mathbf{r} \in [-r_{\text{máx}}, r_{\text{máx}}]^3,\ d \in [0, d_{\text{máx}}],\ f \in [0, F_{\text{máx}}]\right\} \tag{4.1}$$

$$\mathcal{A} = \left\{\mathbf{a} \in \mathbb{R}^3 \;\middle|\; \mathbf{a} \in [-v_{\text{máx}}, v_{\text{máx}}]^3\right\} \tag{4.2}$$

$$R(o, \text{terminated}, \text{truncated}) = \begin{cases} -100, & \text{si } f > f_{\text{máx}} \\ 100 \cdot (d_{\text{mín}} - d), & \text{si } d < d_{\text{mín}} \\ -500, & \text{si el episodio termina por truncamiento} \\ -10 \cdot d, & \text{en cualquier otro caso} \end{cases} \tag{4.3}$$

El modelo ha sido entrenado varias decenas de miles de pasos. Se desconoce la cifra exacta del tiempo de entrenamiento. El motivo es que, al ser uno de los primeros experimentos, no ha sido realizado de manera tan rigurosa y metodológica como el último. Cuando se hizo este experimento, no se pensó en guardar *logs* del mismo.

Los resultados obtenidos se pueden ver en una serie de vídeos en Youtube. En el primer vídeo de este experimento [1] se prueba el modelo en modo TEST. Se puede comprobar a simple vista que el manipulador tiende a desplazar su efector final en la dirección del cubo rojo, situado en la posición del objetivo. El episodio termina cuando el manipulador se encuentra a una cierta distancia, y comienza un nuevo episodio con un objetivo diferente. En el segundo vídeo de este experimento [2] se prueba el modelo en modo TEST_GUI. En este caso, el cubo rojo no tiene ninguna importancia; la

[1] Prueba del modelo reach2 en modo TEST, https://youtu.be/Cg07MDmWieM?si=9ebygA1svdIBGkXE

[2] Prueba del modelo reach2 en modo TEST_GUI, https://www.youtube.com/watch?v=yN3O39m8asI&list=PLRtbP_50L6KPKD9RDeXNnOCo2GeXu3mjP&index=1

Muestra	Distancia al objetivo (m)
1	0.113
2	0.094
3	0.088
4	0.216
5	0.269
6	0.107
7	0.097
8	0.044
9	0.126
10	0.127
11	0.071
12	0.098
13	0.024
14	0.172
15	0.033
Media	0.112
Desviación Típica	0.066
Máximo	0.269

Tabla 4.2: Estimación de la precisión del modelo reach2 mediante muestreo, cálculo de la media y desviación típica de la distancia al objetivo.

posición objetivo (X, Y, Z) es definida por el usuario a través de una simple interfaz gráfica con tres deslizaderas. Nótese que el eje X apunta hacia adelante del robot, el eje Y hacia su izquierda, y el eje Z hacia arriba. En el segundo 7 se fija el objetivo en $(0{,}26, -0{,}29, 0{,}29)m$, situado delante, a la derecha del robot y a media altura. En los segundos posteriores se observa cómo el robot se dirige a esa zona, pero el comportamiento es altamente inestable y oscilatorio. Aunque se haya conseguido que el manipulador sea capaz de acercarse a un punto, este no ha aprendido a mantenerse quieto en dicho punto. El motivo es que este requisito no se ha tenido en cuenta al construir la tarea y los episodios terminan instantáneamente al acercarse. La figura 4.5 muestra cómo, para un mismo objetivo, la posición del efector final no se mantiene fija. La precisión del modelo se calcula en la tabla 4.2. En estado estacionario, la distancia media al objetivo es de $0{,}112m$, con una desviación típica de $0{,}066m$, y alcanzando un valor máximo de $0{,}269m$. Con errores de posicionamiento del orden de las decenas de centímetros, no se puede realizar de manera apropiada tareas de manipulación; por ejemplo, coger un objeto (*Pick and Place*). Otro problema que puede observarse en ambos vídeos es que el manipulador colisiona consigo mismo. Este hecho, además de suponer un riesgo para la integridad física del manipulador, puede cambiar la orientación del efector final, como se muestra en la última imagen de la figura 4.5. Aunque la orientación todavía no haya sido tenida en cuenta, podría serlo en el futuro, ya que es fundamental en cualquier tarea de manipulación.

4.3.2. Experimento 2: un modelo preciso y estable, reach4.

En el experimento reach4 se entrena un modelo con la tarea TaskReach4, cuyo objetivo principal es conseguir que el efector del final se mantenga en la posición objetivo sin oscilaciones, mejorando los resultados de reach2. El espacio de observaciones de la ecuación 4.4 es similar al de TaskReach2, pero se añade el campo , que incluye un vector con las velocidades en los ejes X, Y y Z del sistema de coordenadas global. El espacio de acciones y el controlador del robot son los mismos que en reach2, realizando un control de velocidad en el espacio cartesiano. El modelo controla solamente la posición, no la orientación. En cuanto a la terminación del episodio, este no acaba de manera inmediata al acercarse al objetivo. Se ha incluido un contador que aumenta conforme el efector final del manipulador pasa tiempo a una distancia del objetivo inferior a un umbral. Si el contador supera un límite, termina el episodio, y si el efector final se aleja más allá de la distancia umbral del objetivo, empieza de nuevo en 0. El episodio también puede terminar, por motivos de seguridad, si se detecta una fuerza excesiva en el efector final. Si estas condiciones no se cumplen, pero el episodio se alarga demasiado, existe un límite de pasos, y si este es alcanzado, el episodio se trunca. Se elimina el cubo rojo que se usaba anteriormente como objetivo, ya que el robot colisionaría con el mismo al encontrarse en la posición del objetivo. Ahora, la función que genera la posición aleatoria del objetivo está definida en la tarea (véase el código 4.1). Esta función genera un punto aleatorio en un cuadrado centrado en la base del robot, excluyendo un cilindro que lo contiene. De esta manera, se evita dar como objetivo un punto dentro de la base del robot, que es inaccesible. Por otra parte, la función de recompensa (véase la ecuación 4.5) se ha diseñado de manera que sea derivable y normalizada, con el objetivo de suavizar todo tipo de oscilación. Es una recompensa normalizada basada en penalizaciones, cuya componente principal ($k_d = 89$) es la distancia al objetivo. También se incluyen pequeñas penalizaciones por fuerza ($k_f = 5$), velocidad ($k_v = 5$) y tiempo ($k_t = 1$), y cuando el efector final está cerca del objetivo, se actualiza el contador *self.steps_near_goal*, encargado de la terminación del episodio. Nótese que $K = \sum k_i = 100$ es un factor de normalización, para que la penalización siempre tenga un valor entre 0 y -1. El modelo ha sido entrenado 50000 pasos. Otros parámetros que se han utilizado en la tarea:

- max_steps: 100. Es el número máximo de pasos que puede durar un episodio. Se ha reducido respecto de reach2, evitando que los episodios se alrguen en exceso.
- steps_near_goal_min: 30. Es el número mínimo de pasos que el efector final debe estar cerca del objetivo para terminar el episodio exitosamente.
- d_{min}: 0.2 (m). Es la distancia umbral que determina cuándo el efector final ha llegado al objetivo.
- d_{max}: 3 (m). Es la máxima distancia al objetivo que se puede medir en el espacio de observaciones.

- r_{max}: 1.5 (m). Es la máxima distancia al objetivo en cada uno de los ejes que se puede medir en el espacio de observaciones.
- f_{max}: 40 (N). Es la fuerza máxima permitida. Si se supera, termina el episodio.
- v_{max}: 0.5 (m/s). Es la velocidad máxima permitida del efector final en cada eje del espacio cartesiano.
- F_{max}: 100 (N). Es la magnitud máxima de fuerza que se puede medir en el espacio de observaciones.

$$\begin{aligned}\mathcal{O} = \{ (\mathbf{v}, \mathbf{r}, d, f) \in \mathbb{R}^3 \times \mathbb{R}^3 \times \mathbb{R} \times \mathbb{R} \mid\ & \mathbf{v} \in [-v_{\max}, v_{\max}]^3, \\ & \mathbf{r} \in [-r_{\max}, r_{\max}]^3, \\ & d \in [0, d_{\max}], \\ & f \in [0, f_{\max}] \}\end{aligned} \tag{4.4}$$

Código 4.1: Generación de un objetivo aleatorio en TaskReach4.

```
def PandaRandomGoalPos(self):
    limit = 0.7
    x_bounds = (-limit,limit)
    y_bounds = (-limit,limit)
    z_bounds = (0.2,limit)
    R = 0.3 # Safety radius for the robot
    while True:
        x = random.uniform(*x_bounds)
        y = random.uniform(*y_bounds)
        z = random.uniform(*z_bounds)
        pos = [x,y,z]
        if x**2 + y**2 >= R**2: # Check the point is not too close to
            the robot
            return pos
```

$$R(o) = -\frac{1}{K}\left(k_d \cdot \frac{d}{d_{max}} + k_f \cdot \frac{f}{f_{max}} + k_v \cdot \frac{\|\mathbf{v}_{t-1} - \mathbf{v}_t\|}{v_{max}\sqrt{3}} + k_t\right) \tag{4.5}$$

Tras el entrenamiento, las pruebas con el modelo han sido grabadas. En el primer vídeo[3] se prueba el modelo en el modo de funcionamiento TEST_GUI. En el segundo 10 se fija como objetivo el punto $(0{,}26, -0{,}35, 0{,}34)\,m$, situado delante, a la derecha del robot y a media altura. En comparación con

[3]Prueba del modelo reach4 en modo TEST_GUI, https://youtu.be/XpvOwASS4hQ?si=5NOirruKOytVxFQQ

Figura 4.6: Imágenes del vídeo de reach4 en modo TEST_GUI. La segunda captura es el efector final estático en el punto de la primera captura. La tercera muestra la colisión del manipulador consigo mismo.

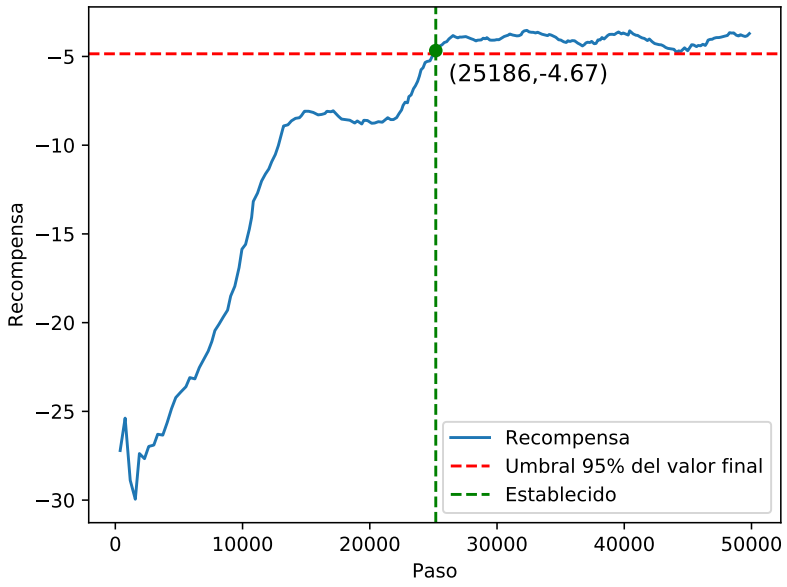


Figura 4.7: Evolución de la recompensa obtenida durante el entrenamiento, y tiempo de establecimiento al 95%.

Muestra	Distancia al objetivo (m)
1	0.037
2	0.118
3	0.149
4	0.074
5	0.059
6	0.081
7	0.047
8	0.053
9	0.091
10	0.087
11	0.036
12	0.087
13	0.079
14	0.043
15	0.160
Media	0.080
Desviación Típica	0.038
Máximo	0.160

Tabla 4.3: Estimación de la precisión del modelo reach4 mediante muestreo, cálculo de la media y desviación típica de la distancia al objetivo.

reach2, el modelo es mucho más estable, las oscilaciones en torno al punto objetivo son visiblemente más pequeñas. Véase la segunda imagen de la figura 4.6. La precisión del modelo se calcula en la tabla 4.3. En estado estacionario, la distancia media al objetivo es de 0,080m, con una desviación típica de 0,038m, y alcanzando un valor máximo de 0,160m. El error cometido es menor que con el modelo reach2. Por otra parte, en el minuto 1:00 se selecciona mediante la interfaz gráfica un punto más allá del límite del espacio de trabajo. Esto es algo que la interfaz gráfica no debería permitir, y se debe ajustar en experimentos futuros. Al final del vídeo, en el minuto 1:40, se selecciona un punto cercano al origen de coordenadas, en el que se encuentra la base del manipulador. El resultado es una colisión no deseada entre la base y el efector final, como muestra la tercera imagen de la figura 4.6.En cuanto al tiempo de aprendizaje, la figura 4.7 muestra que han pasado 25186 pasos hasta que el modelo ha terminado de aprender, alcanzando la máxima recompensa. Otro detalle relativo a la implementación es que la función de generación de puntos aleatorios tiene forma prismática cuadrada, mientras que el alcance del robot en el plano paralelo al suelo es circular. Sería conveniente sustituir dicha función por una generación de puntos homogénea en un espacio cilíndrico o esférico, permitiendo al manipulador explorar por igual su espacio de trabajo.

4.3.3. Experimento 3: un modelo que aprende mal, reach5_1.

En el experimento reach5_1 se entrena un modelo con la tarea TaskReach5. Su objetivo es evitar que el manipulador colisione consigo mismo y que no tome una pose articular extraña que impida

el correcto desplazamiento del efector final, mejorando así los resultados de reach4. El espacio de observaciones (véase la ecuación 4.6) es similar al de reach4, pero se añade el campo $\boldsymbol{\theta}$, que contiene la pose articular del manipulador. La pose articular consiste en un vector de siete dimensiones con el ángulo en radianes de cada articulación. El espacio de acciones y el controlador del robot son los mismos que en reach2 y reach4, realizando un control de velocidad en el espacio cartesiano. El modelo controla solamente la posición, no la orientación. La función de recompensa no varía respecto de reach4. Tampoco cambian las condiciones de terminación del episodio, pudiendo terminar cuando se supera un límite de fuerza, o se está el suficiente tiempo cerca del objetivo de manera ininterrumpida; y si se supera un límite de pasos, el episodio se trunca. En cuanto a la función que genera el objetivo de manera aleatoria, ha sido modificada para generar los puntos en un cilindro que se extiende a lo largo del eje Z (véase el código 4.2), con el objetivo de que el manipulador puede explorar mejor su espacio de trabajo. Los parámetros de la tarea son los mismos que en reach4, excepto por v_{max}, que se ha reducido a 0,4 con el fin de evitar velocidades excesivamente altas.

$$\begin{aligned}\mathcal{O} = \Big\{(\boldsymbol{\theta}, \mathbf{v}, \mathbf{r}, d, f) \in \mathbb{R}^7 \times \mathbb{R}^3 \times \mathbb{R}^3 \times \mathbb{R} \times \mathbb{R} \;\Big|\; & \\ \boldsymbol{\theta} \in [-2\pi, 2\pi]^7, & \\ \mathbf{v} \in [-v_{max}, v_{max}]^3, & \\ \mathbf{r} \in [-r_{max}, r_{max}]^3, & \\ d \in [0, d_{max}], & \\ f \in [0, f_{max}]\Big\} &\end{aligned} \tag{4.6}$$

Código 4.2: "Generación de un objetivo aleatorio en TaskReach5."

```
def PandaRandomGoalPos(self):
    # BOUNDS
    r_bounds = (0.3,0.7)
    z_bounds = (0.2,0.7)

    # GET RANDOM POINT
    r = random.uniform(*r_bounds)
    z = random.uniform(*z_bounds)
    theta = random.uniform(0,2*np.pi)

    # COORDINATES TF
    x = r*np.cos(theta)
    y = r*np.sin(theta)

    return [x,y,z]
```

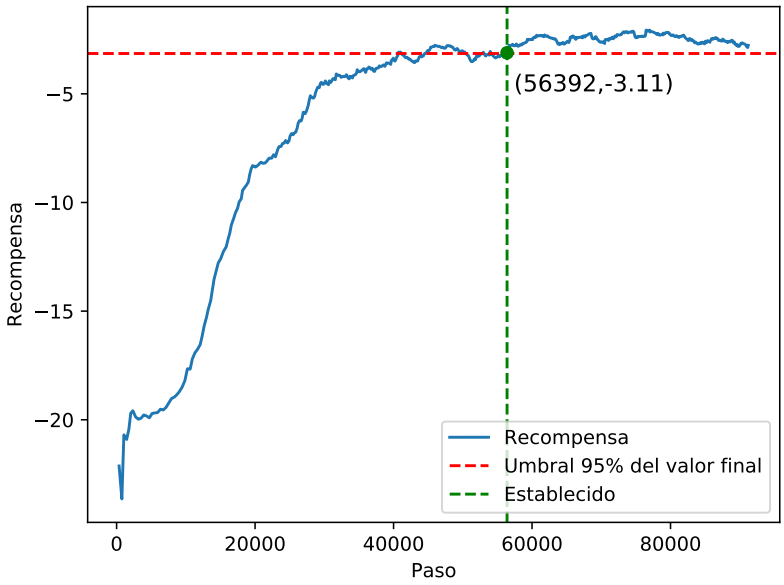


Figura 4.8: Evolución de la recompensa obtenida durante el entrenamiento de reach5_1, y tiempo de establecimiento al 95 %.

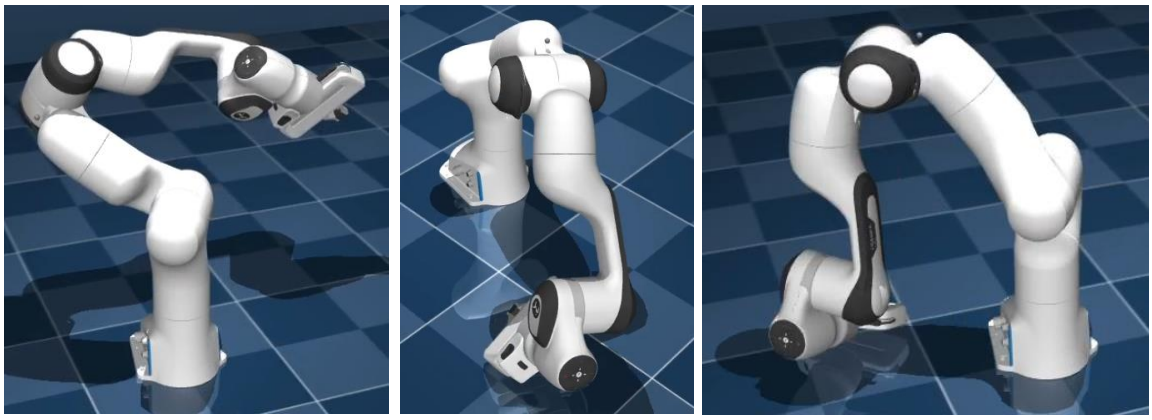


Figura 4.9: Imágenes del vídeo de reach5_1 en modo TEST_GUI. La captura de la izquierda muestra el efector final estable en un punto. Las capturas del centro y la derecha muestran colisiones del efector final con el suelo.

En la figura 4.8 se muestra que han pasado 56392 pasos hasta que se ha estabilizado el aprendizaje del modelo. En el video[4] se ha probado el modelo resultante del entrenamiento. Aunque en el segundo 54 del vídeo el manipulador llega al objetivo y se mantiene estable en él (véase la imagen izquierda de la figura 4.9), se observa que todos los episodios posteriores hasta el segundo 50 del vídeo acaban por colisión; es un comportamiento extraño. Por una parte, el robot a veces es capaz de completar la tarea con éxito, alcanzando el objetivo con una estabilidad comparable a la de reach4. Sin embargo, también ha aprendido a chocar contra el suelo, terminando el episodio prematuramente. Uno de los posibles motivos es la función de recompensa. Es una función densa, con una baja penalización por fuerza, y que no tiene una penalización dispersa al final del episodio en caso de colisión, como sí tenía reach2. Sin embargo, reach4 utilizaba la misma función de recompensa, pero no se ha observado en ningún momento ese tipo de comportamiento. Otra posible explicación sería la introducción de la pose articular en el espacio de observaciones, que es la única diferencia relevante entre reach5_1 y reach4. Al incrementar el espacio de observaciones en siete dimensiones, es posible que el manipulador necesite más episodios para explorarlo por completo. Puede ser que simplemente, el modelo no haya terminado de aprender. En el caso de que el motivo sea el primero, que el robot ha aprendido a chocar porque la penalización a largo plazo es menor, una solución sería no terminar los episodios por un exceso de fuerza, sino únicamente cuando se cumple el objetivo, o se excede el tiempo límite.

Este modelo no se puede considerar un modelo exitoso debido a su propensión a colisionar. Por lo tanto, se omite la toma de medidas para calcular la precisión del mismo.

4.3.4. Experimento 4: un modelo que recibe demasiada información irrelevante, reach5_2.

En el experimento reach5_2 se entrena un modelo con la tarea TaskReach5_2. Esta tarea es semejante a TaskReach5, con la única diferencia de que los episodios no terminan cuando se detecta una fuerza excesiva, con el objetivo de evitar que el robot aprenda a provocar una colisión para terminar el episodio. Se mantienen los objetivos del experimento anterior: evitar que el manipulador colisione consigo mismo y que no tome una pose articular extraña que impida el correcto desplazamiento del efector final, mejorando así los resultados de reach4.

Se ha entrenado el modelo durante 80000 pasos, y la figura 4.13 muestra que han sido necesarios 38715 para aprender la política. La duración ha sido inferior a reach5_1, que aprendió de manera incorrecta, y superior a reach4, cuyo espacio de observaciones tenía siete dimensiones menos. Tras el entrenamiento, las pruebas con el modelo han sido grabadas. En el primer vídeo[5] se observa un comportamiento estable, semejante a reach4. Véase, por ejemplo, la figura 4.10. No se puede discernir cuál de los dos modelos tiene un mejor rendimiento. La única diferencia es que gracias a la nueva función que genera un objetivo dentro de una región cilíndrica, en lugar de prismática, ya no

[4] Prueba del modelo reach5_1 en modo TEST, https://youtu.be/M3rT08je_Dg
[5] Prueba del modelo reach5_2 en modo TEST, https://youtu.be/EQL4kRvVtDg

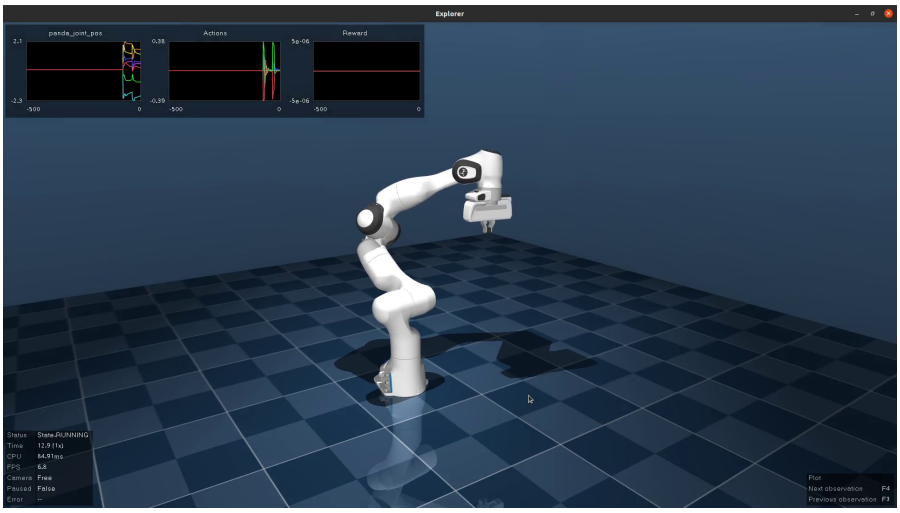


Figura 4.10: Captura del vídeo de reach5_2 en modo TEST, donde el efector final alcanza de manera efectiva el objetivo.

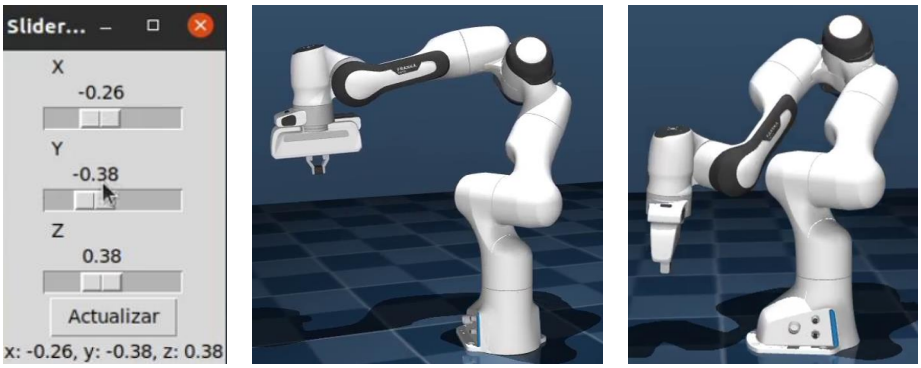


Figura 4.11: Imágenes del vídeo de reach5_2 en modo TEST_GUI. La imagen del centro muestra el efector final estable en un punto a cierta altura Z, indicada en la imagen de la izquierda. La imagen de la derecha muestra el efector final en un punto a una altura visiblemente inferior.

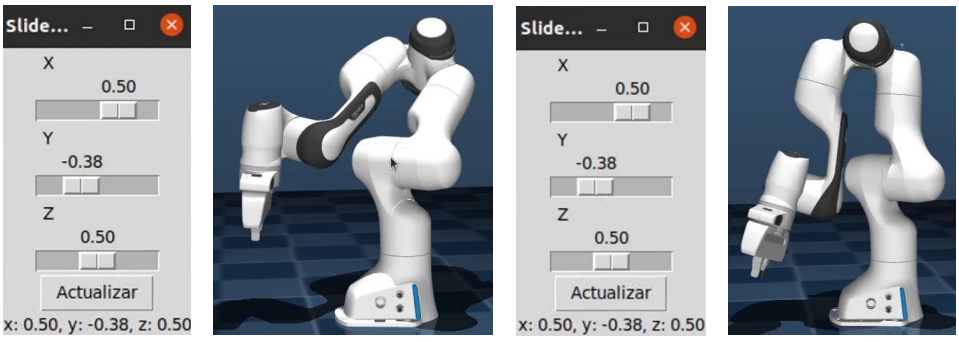


Figura 4.12: Imágenes del vídeo de reach5_2 en modo TEST_GUI. La segunda imagen muestra el efector final estable en un punto en la parte trasera del robot, justo cuando el objetivo cambia a la parte delantera (X positiva). La imagen de la derecha muestra la colisión del efector final con la base del manipulador.

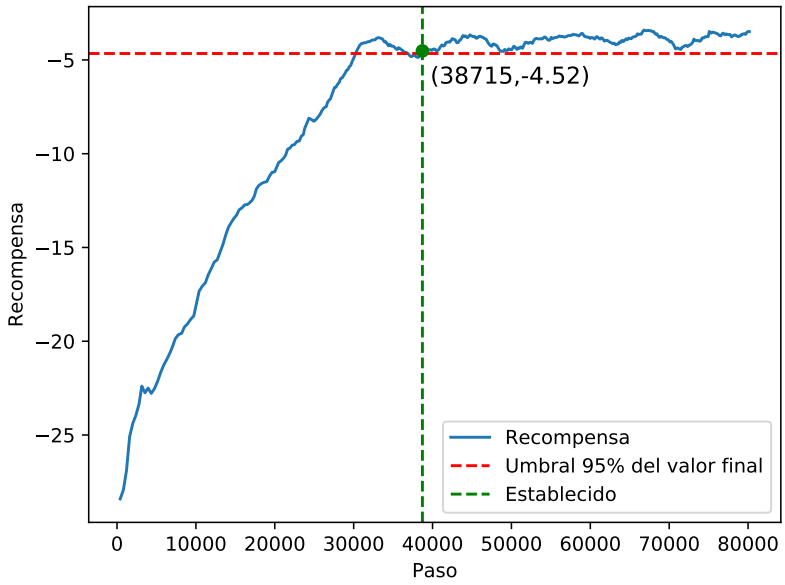


Figura 4.13: Evolución de la recompensa obtenida durante el entrenamiento de reach5_2, y tiempo de establecimiento al 95%.

Muestra	**Distancia al objetivo (m)**
1	0.112
2	0.134
3	0.081
4	0.136
5	0.205
6	0.244
7	0.225
8	0.301
9	0.210
10	0.139
11	0.226
12	0.053
13	0.187
14	0.112
15	0.146
Media	0.167
Desviación Típica	0.068
Máximo	0.301

Tabla 4.4: Estimación de la precisión del modelo reach5_2 mediante muestreo, cálculo de la media y desviación típica de la distancia al objetivo.

se seleccionan puntos fuera del espacio de trabajo, que provocan resultados altamente oscilantes. La precisión del modelo se calcula en la tabla 4.4. En estado estacionario, la distancia media al objetivo es de $0{,}167m$, con una desviación típica de $0{,}068m$, y alcanzando un valor máximo de $0{,}301m$. Los tres valores son peores que los de reach4 e incluso reach2. En el segundo vídeo[6], en el minuto 1:10 se elige un objetivo en la parte trasera del robot a una altura $Z = 0{,}35m$, la misma altura que el punto anterior del minuto 0:57. Sin embargo, el efector final se encuentra a una altura claramente inferior, como se ve en la figura 4.11. Este hecho indica que el modelo no ha aprendido por igual en todo el espacio de observaciones. En el minuto 1:27 se elige cambia el punto objetivo, y el manipulador sigue una trayectoria en la que colisiona con su base. Véase la figura 4.12. Se puede deducir que no es suficiente con dar información de la pose articular al modelo mientras el control sea en velocidad cartesiana, porque no se aprovechan todos los grados de libertad. Es más, la introducción de esas siete dimensiones en el espacio de observaciones es contraproducente, ya que la precisión del modelo ha reducido. El próximo paso podría ser un modelo basado en control articular, que además controle la orientación del efector final.

4.3.5. Experimento 5: primer modelo funcional de control en velocidad articular.

En el experimento reach6 se ha entrenado un modelo con la tarea TaskReach6_3. El objetivo del modelo es controlar tanto la posición como la orientación utilizando un controlador en velocidad articular, y evitar colisiones y poses articulares antinaturales, que restrinjan el correcto desplazamiento del efector final. Para empezar con el control de orientación, el objetivo siempre será apuntar hacia abajo. Si este objetivo se consigue correctamente, se podrá generalizar más adelante para cualquier orientación, pero es mejor incrementar progresivamente la dificultad de la tarea. El espacio de observaciones es el mismo que en reach5_2, incluyendo la pose articular del manipulador, velocidad en el espacio cartesiano, posición relativa al objetivo, orientación relativa al objetivo y distancia al mismo. La orientación relativa al objetivo está representada mediante cuaternios, con el convenio (x, y, z, w), que es el utilizado por dm_robotics_panda. El nuevo espacio de acciones (véase la ecuación **??**) consiste en un vector de siete dimensiones, donde cada componente es la velocidad angular de una articulación del manipulador. Dicha velocidad angular se ha limitado a $\omega_{max} = 1 rad/s$. Al igual que en TaskReach5_2, el episodio no termina en caso de detectarse un exceso de fuerza; acaba solamente cuando el efector final se mantiene durante un tiempo cerca del objetivo. Si se excede un tiempo límite, el episodio se trunca. La función de recompensa (véase la ecuación 4.8), como en los últimos experimentos, es una función densa, normalizada y basada en una serie de penalizaciones. La penalización principal ($k_d = 69$) corresponde a la distancia al punto objetivo. En segundo lugar, se ha introducido una nueva penalización ($k_o = 25$) por error en la orientación del efector final θ. También, se incluyen unas pequeñas penalizaciones por velocidad ($k_v = 5$) y tiempo excesivos ($k_t = 1$). En la figura 4.14 se muestra la nueva interfaz gráfica para el modo TEST_GUI. Ha sido actualizada utilizando la librería *PySide6* para crear una entrada cilíndrica

[6] Prueba del modelo reach5_2 en modo TEST_GUI, https://youtu.be/KG4g0hAcPzQ

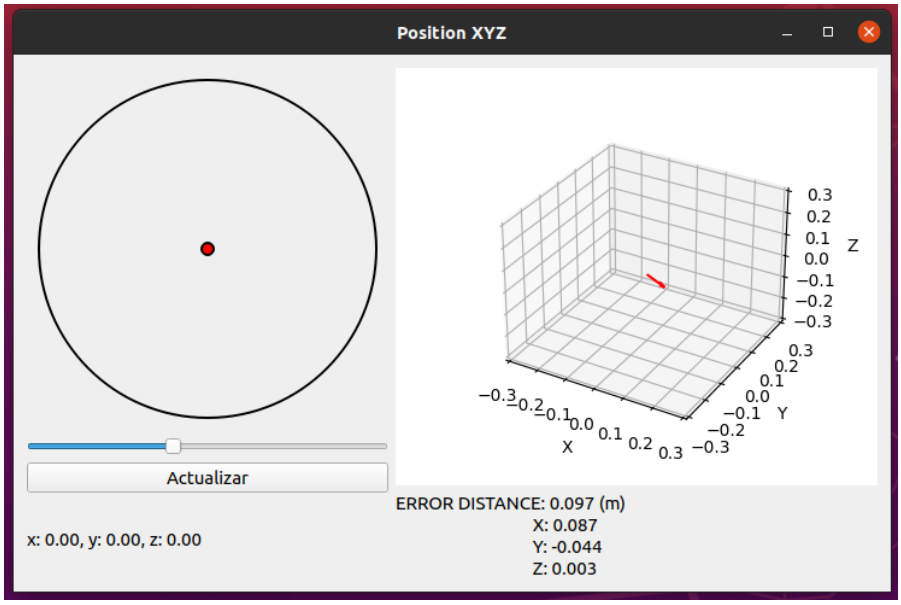


Figura 4.14: Interfaz gráfica para probar el modelo reach6 en el modo de funcionamiento TEST_GUI. Muestra cómo el robot tiene dificultades para alcanzar un punto situadao en su parte trasera izquierda.

de la posición objetivo. Además, se puede visualizar el vector de error de la posición. Por último, se ha añadido un archivo CONFIG.py que centraliza la elección de los parámetros. De esta manera, el usuario solamente tiene que modificar este archivo en lugar de rl_side.py y panda_side.py, dejando un framework más limpio, organizado y fácil de usar.

$$\mathscr{A} = \{\mathbf{a} \in \mathbb{R}^7 \mid \mathbf{a} \in [-\omega_{max}, \omega_{max}]^7\} \tag{4.7}$$

$$R(o) = -\frac{1}{K}\left(k_d \cdot \frac{d}{d_{\text{máx}}} + k_o \cdot \frac{\theta}{\pi} + k_v \cdot \frac{\|\mathbf{v}_{t-1} - \mathbf{v}_t\|}{v_{\text{máx}}\sqrt{3}} + k_t\right) \tag{4.8}$$

Se ha entrenado el modelo durante 100000 pasos, y la figura 4.13 muestra que han sido necesarios 78314 para aprender la política. La duración ha sido superior a todos los modelos anteriores. Esto se debe a que el controlador articular tiene siete grados de libertad, en lugar de 3. El incremento dimensional implica un modelo más complejo, que necesita más tiempo para aprender. Además, la relación entre el espacio cartesiano de la observación y el articular de la actuación no es lineal. En el primer video[7] se prueba el modelo en modo TEST. Se puede observar que el manipulador busca en cada episodio una posición objetivo, y que controla la orientación, buscando apuntar hacia abajo con el efector final. El comportamiento es altamente oscilante, a diferencia de otros experimentos anteriores como reach4. Este hecho puede deberse a que es espacio de acciones ha pasado de tener

[7] Prueba del modelo reach6 en modo TEST, https://youtu.be/7BT5CKZGxQE

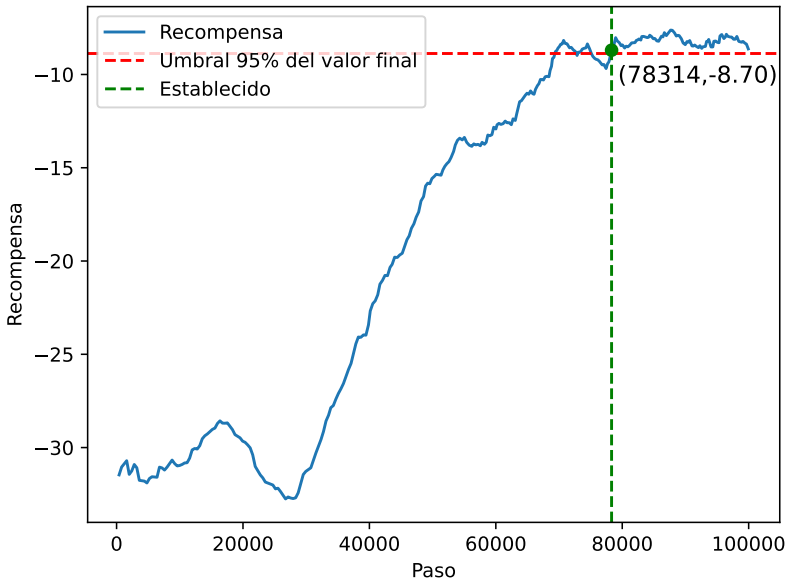


Figura 4.15: Evolución de la recompensa obtenida durante el entrenamiento de reach6, y tiempo de establecimiento al 95 %.

Muestra	**Distancia al objetivo (m)**
1	0.065
2	0.13
3	0.209
4	0.069
5	0.099
6	0.066
7	0.291
8	0.145
9	0.153
10	0.150
11	0.153
12	0.252
13	0.140
14	0.088
15	0.145
Media	0.144
Desviación Típica	0.066
Máximo	0.291

Tabla 4.5: Estimación de la precisión del modelo reach6_3 mediante muestreo, cálculo de la media y desviación típica de la distancia al objetivo.

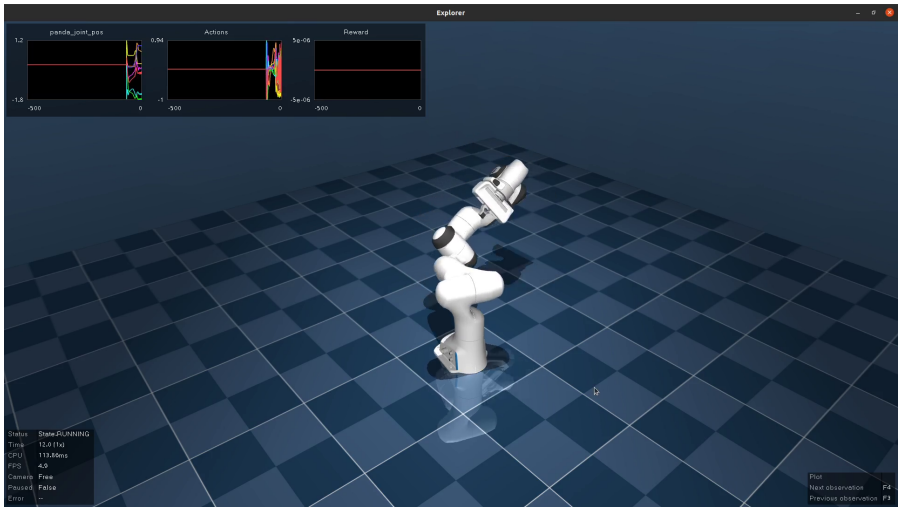


Figura 4.16: Video reach6 en modo TEST, minuto 0:18.

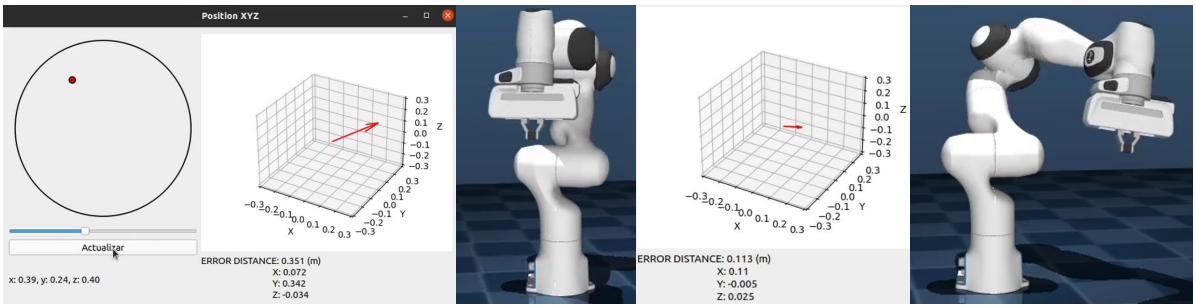


Figura 4.17: Imágenes del vídeo de reach6 en modo TEST_GUI. La imagen de la izquierda muestra el momento exacto en el que se fija una nueva posición objetivo, y el vector de error crece hasta $0{,}351m$. La imagen de la derecha muestra cuando el efector final alcanza el objetivo, y el error disminuye a $0{,}113m$.

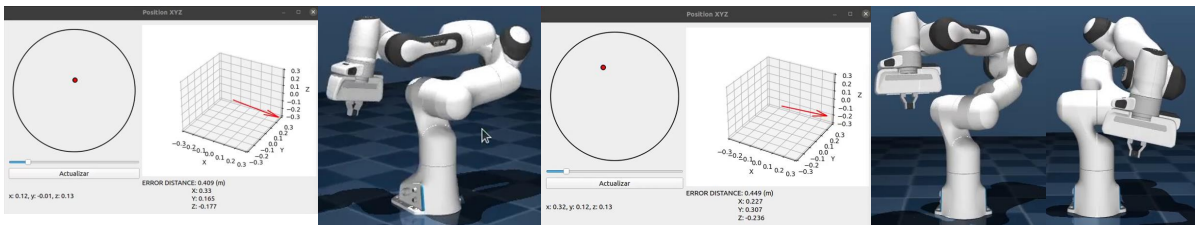


Figura 4.18: Imágenes del vídeo de reach6 en modo TEST_GUI, evitando chocar. Las imágenes de la izquierda muestran el efector final atascado, incapaz de avanzar y con un gran error. En la tercera imagen, tras cambiar ligeramente el objetivo, el efector final da la vuelta a la base. En la imagen de la derecha, el efector final llega al objetivo.

tres dimensiones a siete, lo cual supone un gran aumento de la complejidad del control. En los minutos 0:18 (figura 4.16) y 1:30 se muestra que el modelo tiene aún más dificultades cuando el objetivo está en la parte trasera del robot. Un posible motivo es que la inicialización de la pose no es homogénea en todo el espacio de trabajo, y el efector final comienza el episodio siempre en la parte delantera del manipulador, con una orientación variable, pero cercana a la vertical. Si se quiere un modelo capaz de generalizar la tarea a cualquier pose, es necesario cambiar esto.

En el segundo vídeo[8] se prueba el modelo en modo TEST_GUI. El minuto 0:30 es un claro ejemplo de que, al cambiar el punto objetivo al lado izquierdo del manipulador, este se desplaza en su busca. En la interfaz gráfica a la izquierda de la figura 4.17 se muestra el vector de error, junto con sus componentes y magnitud, justo en el momento en el que se cambia el punto. En la misma figura a la derecha, el efector final alcanza el objetivo, y el error mostrado es menor. La precisión del modelo se calcula en la tabla 4.5. En estado estacionario, la distancia media al objetivo es de $0{,}144m$, con una desviación típica de $0{,}066m$, y alcanzando un valor máximo de $0{,}291m$. Los tres valores son peores que los de reach4 y reach2, pero mejores que reach5_2. Una posible manera de mejorar la precisión de la posición sería eliminar el contador que provoca el fin del episodio cuando el efector final pasa un tiempo cerca del objetivo. También sería conveniente añadir a la interfaz gráfica alguna manera de visualizar el error cometido en la orientación. Para terminar, en el minuto 2:00 el efector final se encuentra en la parte trasera derecha del robot, y se le asigna ir a un punto de la parte delantera izquierda. Véase la figura 4.18. Ir en línea recta supondría una colisión con la base del manipulador. Sin embargo, se ve claramente cómo el efector final, tras unos segundos atascado, acaba bordeando el obstáculo sin chocar. De todos los experimentos realizados hasta el momento, es la primera vez que hay señales de que el modelo ha aprendido, en algunos casos, a evitar colisiones.

4.4. Experimentos en el robot real

Una vez conseguido un modelo relativamente estable, se puede probar la implementación en el robot real con un riesgo reducido de dañar el hardware. El objetivo de estos experimentos es comprobar si el aprendizaje realizado en simulación con el framework puede ser transferido al mundo real de manera efectiva.

4.4.1. Experimento 6: prueba en el robot real de reach4.

En este experimento se prueba el modelo reach4 en el robot real. Es el modelo de control en velocidad cartesiana más prometedor de los obtenidos hasta el momento. En el vídeo[9] el efector final comienza en la posición de inicio. En el segundo 9 recibe una orden de desplazarse a un punto a la izquierda de la imagen, al cual se aproxima correctamente en el segundo 14. A partir del segundo 20

[8] Prueba del modelo reach6 en modo TEST_GUI, https://youtu.be/KG4gOhAcPzQ

[9] Prueba del modelo reach4 en modo TEST_GUI en el robot real, https://youtu.be/NeyScID3Ab8?si=ymjQXUhM2iToQ1Vp

Figura 4.19: Manipulador Franka Emika Panda en su posición inicial al comienzo del experimento 6.

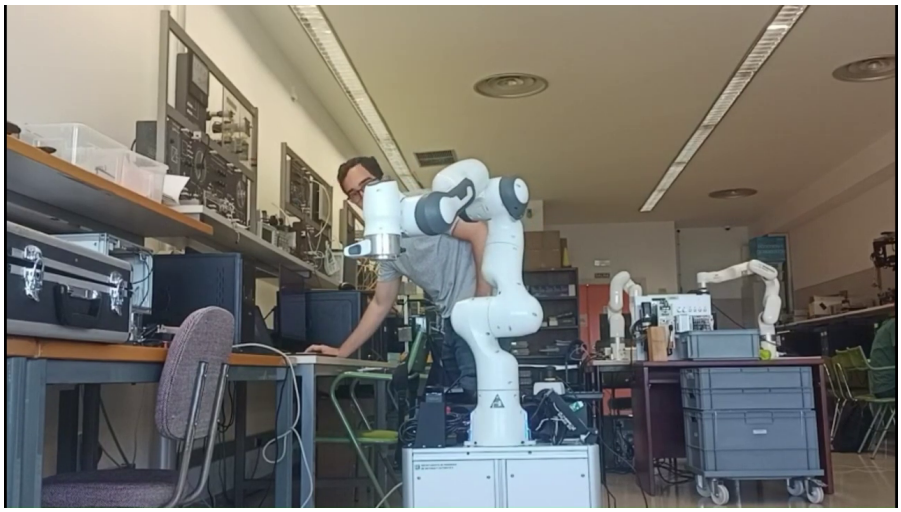


Figura 4.20: Manipulador Franka Emika Panda con el efector final en el punto objetivo, situado a la izquierda.

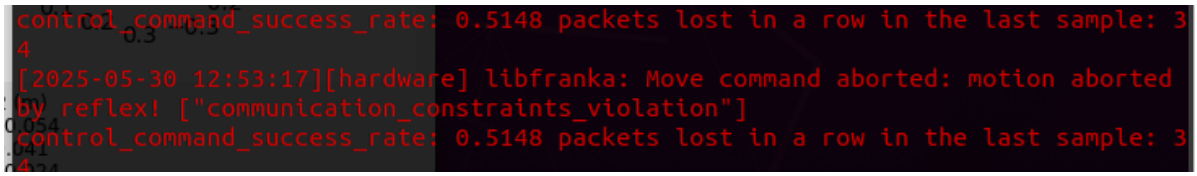


Figura 4.21: Mensaje que aparece en el terminal durante el experimento en el robot real. Es un mensaje de error de la librería libfranka, que indica que se han perdido paquetes en las comunicaciones con el robot.

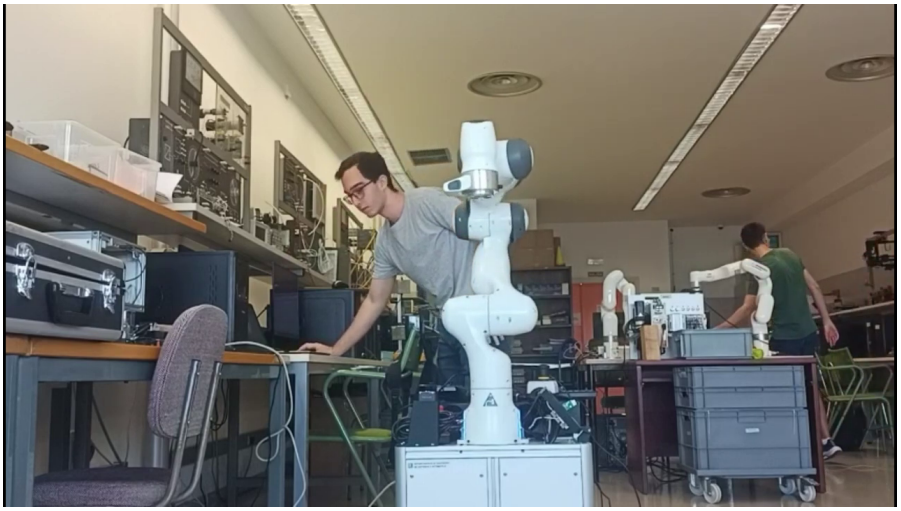


Figura 4.22: Manipulador Franka Emika Panda en su posición inicial al comienzo del experimento 7.

se observan algunos problemas de fluidez en el movimiento del robot. Desde el terminal se deduce que es debido a pérdida de paquetes en las comunicaciones, como se muestra en la figura 4.21. Este fenómeno sucede frecuentemente a lo largo del vídeo. Es probable que el motivo sea la arquitectura de control del robot. Esta arquitectura consta de un bucle en el que se ejecutan varias veces las acciones seleccionadas por el modelo, y cuando pasa un tiempo, espera a recibir una nueva acción. El problema es que esta función bloquea la ejecución del bucle. Este aspecto permite entrenar en simulación con la seguridad de que el tiempo de control es correcto, pero parece provocar problemas en la implementación con el robot físico. Por lo demás, el comportamiento es similar al reflejado en simulación; el robot tiende a ir al punto objetivo. Se puede afirmar que se transfiere el conocimiento adquirido en simulación a la realidad de manera exitosa.

4.4.2. Experimento 7: prueba en el robot real de reach6.

En este experimento se prueba el modelo reach6 en el robot real. Este modelo controla la velocidad de las articulaciones del manipulador. Del vídeo[10] se pueden extraer conclusiones similares al experimento anterior. El efector final comienza en la posición de inicio, y se cambia continuamente su pose objetivo, comenzando por algunos puntos en la parte delantera del manipulador. En el principio del vídeo se ve claramente que hay pérdida de paquetes en las comunicaciones, lo cual provoca paradas cortas en la trayectoria seguida por el manipulador. En el segundo 25 del vídeo, por ejemplo, se fija como objetivo un punto en la parte izquierda de la imagen, y el efector final se dirige al mismo manteniendo la orientación vertical. Este experimento confirma que el conocimiento del modelo entrenado en simulación es transferible al robot real.

[10]Prueba del modelo reach6 en modo TEST_GUI en el robot real, https://youtu.be/1fdbq5b4yNo?si=dRNNUzx5vZZZUa0d

Figura 4.23: Manipulador Franka Emika Panda con el efector final en el punto objetivo, situado a la izquierda. La orientación, aunque es aproximadamente vertical, tiene un error visiblemente mayor a la posición.

4.5. Discusión de los resultados

En los experimentos realizados en simulación (1, 2, 3, 4 y 5) se han entrenado una serie de modelos que realizan una tarea de tipo *Reach*. La tabla 4.6 es un resumen de los resultados obtenidos. De los modelos de control en velocidad cartesiana reach2 y reach4 se deduce que es necesario enseñar al modelo no solo a alcanzar el punto, sino también a quedarse en el mismo. No tener en cuesta este requisito de diseño provoca inestabilidad y oscilaciones en el periodo estacionario. El modelo reach5_1 muestra que los entornos deben ser diseñados con cuidado, especialmente la función de recompensa, que puede incentivar comportamientos imprevisibles e indeseados. Este modelo aprendió a terminar rápidamente el episodio para no acumular penalizaciones en el largo plazo. Si existe una manera de maximizar la función de recompensa que no está relacionada con el objetivo original de la tarea, el modelo puede aprender una tarea completamente diferente. El modelo reach5_2 prueba que un modelo con más información no necesariamente es mejor. La introducción de 7 dimensiones en el espacio de observaciones no ha surtido ningún efecto positivo. Es importante incluir información relevante que el modelo pueda aprovechar para deducir el comportamiento que debe aprender. De lo contrario, este aumento de la complejidad puede suponer una pérdida de precisión y un aumento de los recursos invertidos, principalmente tiempo. El último modelo de control en velocidad articular, reach6, tiene un grado de complejidad superior a los demás, teniendo 7 dimensiones en el espacio de actuación y 18 en el espacio de observaciones. La duración de su entrenamiento ha sido superior a la de modelos más simples, y la precisión tampoco es mejor. Este modelo, a diferencia de los anteriores, es capaz de controlar la orientación y reajustarla si se desvía del objetivo. Además, ha sido capaz de aprender, a partir de la observación de la pose articular, a evitar que el efector final colisione con el manipulador.

Métrica	**reach2**	**reach4**	**reach5_1**	**reach5_2**	**reach6**
Controlador	V. cartesiana	V. cartesiana	V. cartesiana	V. cartesiana	V. articular
N	-	25186	56392	38715	78314
$\mathscr{A}$	3	3	3	3	7
$\mathscr{O}$	5	8	15	15	18
μ (m)	0.112	0.080	-	0.167	0.144
max (m)	0.269	0.160	-	0.301	0.291
σ (m)	0.066	0.038	-	0.068	0.066
Observaciones	Oscila	Estable	Colisiona	Estable	Oscila

Tabla 4.6: Recopilación de las métricas correspondientes a los modelos entrenados en los experimentos. Comparación del tipo de controlador, duración del entrenamiento, espacio de acciones, espacio de observaciones y distancia al objetivo (μ, máximo y desviación estándar).

En cuanto a los experimentos en el robot real (6 y 7), han demostrado que algunos de los modelos entrenados en simulación (reach4 y reach6) pueden ser utilizados para controlar el manipulador Franka Emika Panda. El efector final del manipulador siempre ha ido a los objetivos que el usuario ha seleccionado. El comportamiento observado es visiblemente similar al de los experimentos en simulación. La única diferencia es un problema de implementación relacionado con las comunicaciones. La pérdida de paquetes en la comunicación entre ordenadores provoca que la trayectoria seguida por el efector final sufra paradas momentáneas. Este fenómeno está claramente no relacionado con el aprendizaje del modelo, y el motivo más probable es el bloqueo del bucle de control del robot.

Se puede afirmar que el *framework* permite entrenar y probar en simulación modelos que aprenden a realizar tareas básicas de manipulación mediante aprendizaje por refuerzo, y la política aprendida en simulación puede ser transferida a un manipulador real. La única limitación relevante del *framework* es el fallo de las comunicaciones cuando se despliega el modelo en un manipulador real, que deberá resolverse en trabajos futuros.

Capítulo 5

Conclusiones

En este proyecto se ha desarrollado con éxito un *framework* de aprendizaje por refuerzo para el robot manipulador Franka Emika Panda. El *framework* es compatible con la librería de algoritmos *Stable-Baselines3*, una de las librerías más comúnmente utilizadas. Se han documentado cinco experimentos de entrenamiento de modelos en simulación. En cuatro de ellos se han obtenido modelos que han aprendido a realizar la tarea básica de manipulación, alcanzar un punto en el espacio. Se han documentado dos experimentos en los que se han probado en un robot real modelos entrenados en simulación. En estos experimentos, se ha demostrado los modelos funcionan en la realidad, y que se pueden utilizar de manera inmediata, sin ningún tipo de adaptación especial.

Más allá de los objetivos propuestos, existen múltiples maneras de maneras de mejorar el trabajo realizado. En primer lugar, hay que arreglar el bucle de control del robot. Se ha comprobado que el movimiento del manipulador real no siempre es fluido, y se producen pérdidas de paquetes en las comunicaciones, posiblemente porque el bucle se detiene mientras se recibe una nueva acción del modelo. Este hecho es conveniente para el entrenamiento en simulación porque se controla cuánto tiempo se ejecuta cada acción. Sin embargo, sería ideal utilizar en el robot real procesamiento paralelo (*threads* o *multiprocessing* de Python) para evitar el problema del bloqueo, que es uno de los principales puntos débiles del *framework*. Otro ejemplo de las limitaciones de este proyecto es que el entrenamiento solo puede ser realizado por un robot virtual a la vez, no se pueden tener varios entornos tomando datos en paralelo (entornos vectorizados). Además, el entrenamiento de modelos para este trabajo ha sido realizado en tiempo real. Implementar funcionalidades que aceleren el proceso de entrenamiento resultaría de gran utilidad, ya que permitiría aprovechar la capacidad computacional de los equipos modernos.

Si bien es importante considerar sus limitaciones, también existen futuras líneas de trabajo

orientadas a la explotación del *framework*. Por ejemplo, entrenar modelos más precisos para tareas de manipulación de tipo *Reach*, o para tareas más complejas, como *Pick-and-Place* o abrir una puerta. Respecto del proyecto de investigación *TYRELL*, se pueden hacer experimentos con distintos tiempos de toma de decisiones utilizando el *framework* presentado en este proyecto. Para ello, sería de gran utilidad realizar una ligera modificación, haciendo que el tiempo de toma de decisiones, que actualmente es constante, pueda ser variable.

Bibliografía

[1] M. Peshkin and J. E. Colgate, "Cobots," *Industrial Robot: An International Journal*, vol. 26, no. 5, pp. 335–341, 1999.

[2] C. Taesi, F. Aggogeri, and N. Pellegrini, "Cobot applications—recent advances and challenges," *Robotics*, vol. 12, no. 3, 2023. [Online]. Available: https://www.mdpi.com/2218-6581/12/3/79

[3] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, 2nd ed. The MIT Press, 2018.

[4] S. Höfer, K. Bekris, A. Handa, J. C. Gamboa, M. Mozifian, F. Golemo, C. Atkeson, D. Fox, K. Goldberg, J. Leonard *et al.*, "Sim2real in robotics and automation: Applications and challenges," *IEEE transactions on automation science and engineering*, vol. 18, no. 2, pp. 398–400, 2021.

[5] C. J. Watkins and P. Dayan, "Q-learning," *Machine learning*, vol. 8, pp. 279–292, 1992.

[6] G. A. Rummery and M. Niranjan, *On-line Q-learning using connectionist systems*. University of Cambridge, Department of Engineering Cambridge, UK, 1994, vol. 37.

[7] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, "Human-level control through deep reinforcement learning," *nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[8] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," *Machine learning*, vol. 8, pp. 229–256, 1992.

[9] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz, "Trust region policy optimization," in *International conference on machine learning*. PMLR, 2015, pp. 1889–1897.

[10] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

[11] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," in *International conference on machine learning*. PmLR, 2016, pp. 1928–1937.

[12] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," *arXiv preprint arXiv:1509.02971*, 2015.

[13] S. Fujimoto, H. Hoof, and D. Meger, "Addressing function approximation error in actor-critic methods," in *International conference on machine learning*. PMLR, 2018, pp. 1587–1596.

[14] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, "Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor," in *International conference on machine learning*. Pmlr, 2018, pp. 1861–1870.

[15] J. Elsner, "Taming the panda with python: A powerful duo for seamless robotics programming and integration," *SoftwareX*, vol. 24, p. 101532, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2352711023002285

[16] E. Todorov, T. Erez, and Y. Tassa, "Mujoco: A physics engine for model-based control," in *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, 2012, pp. 5026–5033.

[17] Q. Gallouédec, N. Cazin, E. Dellandréa, and L. Chen, "panda-gym: Open-Source Goal-Conditioned Environments for Robotic Learning," *4th Robot Learning Workshop: Self-Supervised and Lifelong Learning at NeurIPS*, 2021.

[18] M. Towers, A. Kwiatkowski, J. Terry, J. U. Balis, G. D. Cola, T. Deleu, M. Goulão, A. Kallinteris, M. Krimmel, A. KG, R. Perez-Vicente, A. Pierré, S. Schulhoff, J. J. Tai, H. Tan, and O. G. Younis, "Gymnasium: A standard interface for reinforcement learning environments," 2024. [Online]. Available: https://arxiv.org/abs/2407.17032

[19] Z. Xu, Y. Li, X. Yang, Z. Zhao, L. Zhuang, and J. Zhao, "Open-source reinforcement learning environments implemented in mujoco with franka manipulator," 2023.

[20] R. de Lazcano, K. Andreas, J. J. Tai, S. R. Lee, and J. Terry, "Gymnasium robotics," 2024. [Online]. Available: http://github.com/Farama-Foundation/Gymnasium-Robotics

[21] Y. Zhu, J. Wong, A. Mandlekar, R. Martín-Martín, A. Joshi, S. Nasiriany, Y. Zhu, and K. Lin, "robosuite: A modular simulation framework and benchmark for robot learning," in *arXiv preprint arXiv:2009.12293*, 2020.

Apéndice A

En este apéndice se describe el hardware del robot RAFI, que se ha utilizado en el laboratorio para los experimentos de la sección 4.4.

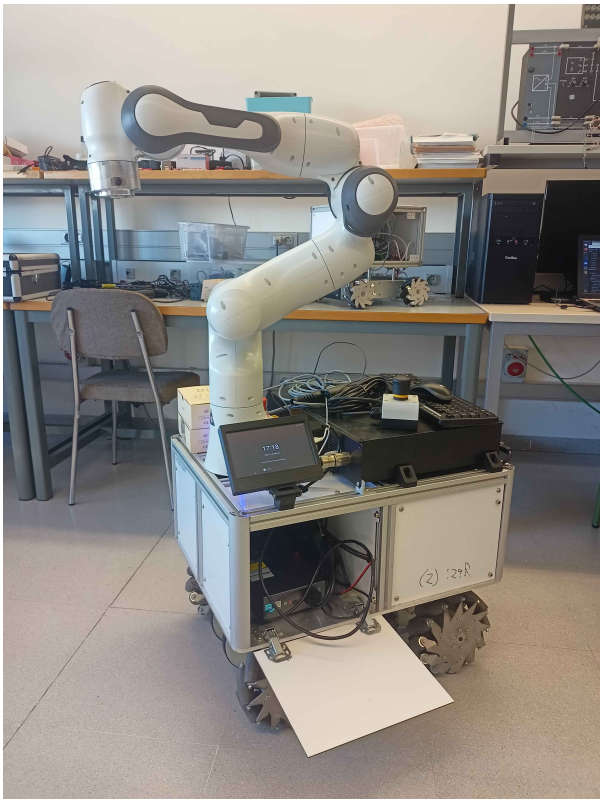


Figura 1: Robot de Asistencia Física Inteligente (RAFI). Está compuesto de un manipulador Franka Emika Panda y una base móvil con ruedas mecanum.

En el laboratorio de la Universidad de Málaga está RAFI, un robot compuesto de una base móvil con ruedas suecas y un brazo manipulador Franka Emika Panda de 7 grados de libertad. En este trabajo se utiliza únicamente el brazo manipulador. Para encender el robot y conectarse al mismo, hay que seguir una serie de pasos. En primer lugar, conectar el cable gris de alimentación a la corriente y poner en la posición del círculo el interruptor que hay justo debajo 2a. Después, levantar la seta roja de emergencia que hay en la parte trasera del robot 2b, abrir la compuerta en la que se encuentra el ordenador introduciendo una herramienta larga por el hueco que hay detrás de la pantalla y empujando la chapa hacia afuera 2c, y pulsar el botón de encendido de ese ordenador 2d. Posteriormente, se deben levantar las setas roja y negra 2e, pertenecientes a la alimentación del ordenador del manipulador y al mismo manipulador, y poner en la posición de la raya el interruptor 2f.

El ordenador del robot tiene como sistema operativo Ubuntu 20.04 y se puede acceder a él mediante *Secure Shell* a través de una red local. Para ello, hay que ejecutar el comando *ssh -XC <user>@<ip>*, donde C significa que se compriman los datos, para una mayor velocidad, y X significa que puedan abrirse ventanas gráficas interactivas. También se puede controlar directamente conectando mediante USB un teclado y un ratón. Para que el código que utiliza dm_robotics_panda funcione correctamente, es necesario entrar en DESK y asegurarse de que los frenos están desactivados y el FCI

activo. Se puede acceder a esta página introduciendo en el buscador Google Chrome la ip del ordenador del manipulador, que es 172.16.0.2 por defecto. Se recomienda utilizar Google Chrome porque Mozilla Firefox a veces devuelve un error relacionado con la seguridad de la página, impidiendo el acceso a la misma. Una vez está todo listo y se ha accedido mediante SSH, hay que ir al directorio en el que se tenga un entorno virtual de Python con dm_robotics_panda instalado y activarlo, tras lo cual ya se puede ejecutar el código. Por ejemplo, *python3 example.py –gui –robot-ip 172.16.0.2*. Para más información sobre trabajar con el framework, véase la guía de usuario en la sección 3.3.

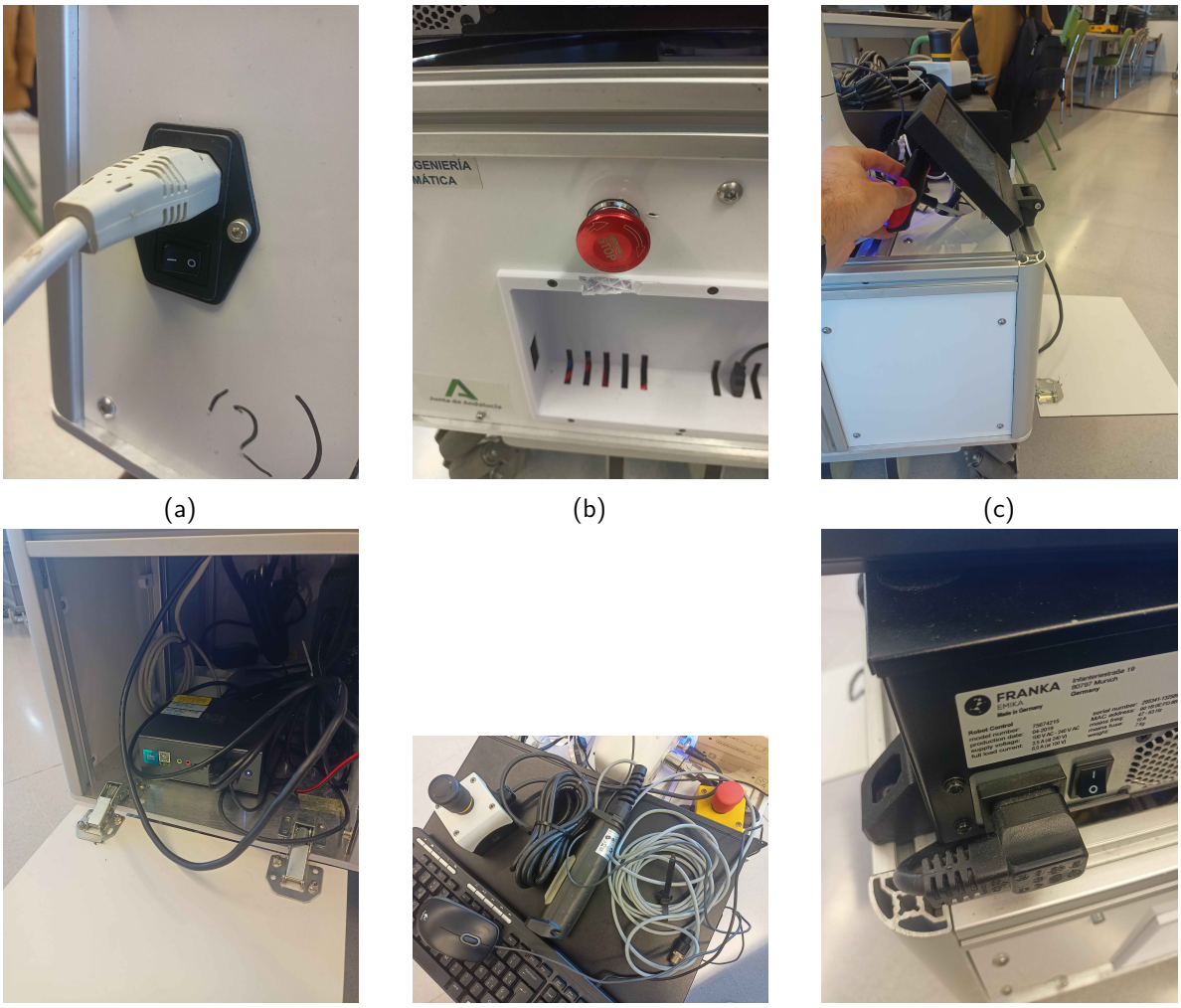


Figura 2: Interfaz hardware de RAFI. (a) Alimentación. (b) Seta de emergencia. (c) Compuerta. (d) Ordenador central de RAFI. (e) Setas del manipulador. (f) Botón de encendido del ordenador del manipulador.