Universidad Zaragoza

Trabajo Fin de Máster

Control de robots mediante algoritmos de aprendizaje por refuerzo profundo

Robot control by Deep Reinforcement Learning techniques

Autor

Alberto Simón Tena

Director

Rubén Martínez Cantín

ESCUELA DE INGENIERÍA Y ARQUITECTURA

2021

TRABAJOS DE FIN DE GRADO / FIN DE MÁSTER

Escuela de Ingeniería y Arquitectura Universidad Zaragoza

DECLARACIÓN DE AUTORÍA Y ORIGINALIDAD

(Este documento debe remitirse a seceina@unizar.es dentro del plazo de depósito)

D./Dª. Alberto Simón Tena ,

en aplicación de lo dispuesto en el art. 14 (Derechos de autor) del Acuerdo de 11 de septiembre de 2014, del Consejo de Gobierno, por el que se aprueba el Reglamento de los TFG y TFM de la Universidad de Zaragoza,

Declaro que el presente Trabajo de Fin de Estudios de la titulación de Máster Universitario en Ingeniería Electrónica (Título del Trabajo)

Control de robots mediante algoritmos de aprendizaje por refuerzo profundo.

es de mi autoría y es original, no habiéndose utilizado fuente sin ser citada debidamente.

Zaragoza, a día 24 de noviembre de 2021

SIMON TENA ALBERTO FERNANDO - 18044369H

Firmado digitalmente por SIMON TENA ALBERTO FERNANDO - 18044369H
Fecha: 2021.11.24 20:13:35 +01'00'

Fdo: Alberto Simón Tena

A Tomàs Pallejà

AGRADECIMIENTOS

Agradezco a mi tutor, Rubén Martínez-Cantín, por guiarme con paciencia en una disciplina y un entorno nuevos para mi; a Stephen James, programador de las librerías **PyRep** y **RLBench**, tributarias de este trabajo, y a la legión de personas que contribuyen con su talento a la creación de herramientas de código abierto.

Robot control by Deep Reinforcement Learning techniques.

Abstract

The aim of this project is to explore the state of the art in Deep Reinforcement Learning Techniques in order control a realistic 7-joints robotic arm. Following the study of classic Reinforcement Learning (from now on, *RL*) techniques, and due the complexity of the environment, we junp to cutting-edge algorithms *Soft Actor Critic*, *Proximal Policy Optimization* or *Twin Delayed DDPG*, among others, using techniques from *Deep Machine Learning* to solve Bellman functions and implement their policy. There is a practical approach in order to apply those algorithms to a continous-state environment of actions and observations where a virtualised robot has to learn how to reach a target position with no supervision at all. Step by step, best algorithm is selected and tested in order to find optimal dimensions of both **actor** and **critic** Neural Networks.

Control de robots mediante algoritmos de aprendizaje por refuerzo profundo.

Resumen

El objetivo de este proyecto consiste en investigar el estado del arte en técnicas de aprendizaje por refuerzo profundo con el objetivo de controlar un brazo robótico virtual de 7 juntas modelado con un sistema real. A partir del estudio de los postulados del aprendizaje por refuerzo o *Reinforcement Learning*, se abordan los algoritmos más recientes, como *Soft Actor Critic*, *Proximal Policy Optimization* o *Twin Delayed DDPG*, entre otros, que aproximan las funciones Bellman que determinan la política del robot mediante el uso de redes neuronales y técnicas de aprendizaje automático profundo o *Deep Machine Learning*. Se establece una aproximación práctica de estos algoritmos en su versión aplicable a entornos continuos, que constituyen las entradas y salidas del brazo robótico, encargado de aprender a alcanzar una posición determinada sin ningún tipo de supervisión humana. Paso a paso, se escoge el algoritmo que mejor resuelve este tipo de problemas, antes de abordar los parámetros óptimos y el tamaño de sendas redes neuronal para el **actor** y el **crítico** que mejor se adaptan a este tipo de problemas.

Índice

Capítulo 1

Introducción y objetivos

El aprendizaje por refuerzo representa un paradigma que va más allá de la inteligencia artificial y la programación de sistemas. En la primera edición de su obra, Richard S. Sutton y Andrew G. Barto, los investigadores que desarrollaron la teoría detrás del *Reinforcement Learning*, motivan sus planteamientos detrás de campos como la neurociencia o el comportamiento animal, a los que que intentan dar un formalismo matemático. Además, reconocen la estrecha vinculación entre sus postulados y otras areas de la ingeniería como la teoría del control óptimo en **ingeniería de control** o la programación dinámica en **matemáticas** o **informática**. [1]

1.1. ¿En qué consiste el Aprendizaje por Refuerzo?

El aprendizaje por refuerzo o *Reinforcement Learning* es un aprendizaje basado en la experiencia, en la interacción con el entorno y en el clásico «prueba y error». Es la forma en que intuimos que una criatura aprende un comportamiento. La psicología evolutiva ha impregnado su visión en esta rama de la inteligencia artificial.

Su objetivo como ciencia fundamental abarca el problema de la toma de decisiones óptimas e intersecta conocimientos de otras disciplinas. El diagrama de Venn de la figura 1.1 ilustra que el problema del aprendizaje por refuerzo se nutre de (y puede aportar a) múltiples campos.

Todos tenemos la experiencia de haber aprendido comportamientos inconscientes. Por ejemplo, cuando estamos aprendiendo una nueva conducta como conducir un automóvil, enfocamos nuestra atención a ver lo que sucede alrededor como resultado de nuestras acciones. Esta es la base del aprendizaje por refuerzo.

[1] "Particularly important have been the contributions establishing and developing the relationships to the theory of optimal control and dynamic programming"[1]p.vi

Hay evidencias circunstanciales (estadísticas) de la similitud entre los algoritmos de aprendizaje no supervisado a través de redes neuronales profundas con mecanismos de aprendizaje en animales [2].

A la entidad que aprende no se le programa qué acciones realizar en cada momento. En su lugar, se diseña una estrategia basada en recompensas[2], según las acciones que deseemos. De esta forma es la propia máquina, mediante «prueba y error», quien acaba estableciendo un patrón de comportamiento que produce la mayor de las recompensas posibles.

Las acciones que gobiernan un brazo robótico pueden estimarse a través de cálculo iterativo o geométrico, la llamada *cinemática inversa*, que aumentan exponencialmente con la complejidad del robot. En el mejor de los casos, podemos gobernar la posición (estática) el movimiento (cinemática), las fuerzas (dinámica) y la posición del comportamiento de ese robot, pero a todas luces es un método vetusto si pretendemos emular un comportamiento inteligente.

Sin embargo, mediante técnicas de RL, no es complicado diseñar un escenario donde el brazo robótico se gobierne a sí mismo, y sólo nos tengamos que preocupar de diseñar una estrategia de recompensas en función de cuán similar es su comportamiento respecto al de un humano, al menos en determinadas tareas como abrir una puerta o mover una palanca. ¡Cuidado! Es relativamente sencillo plantear la estrategia, pero implementar un algoritmo que realmente sea capaz de maximizar esa recompensa satisfaciendo los objetivos planteados ya es otro cantar.

Ántes de continuar, cabe plantearse si un algoritmo de esa misma categoría es capaz de solventar problemas sencillos que actualmente resuelve la *cinemática inversa*. Para ello, empezamos tareas más sencillas o meta-tareas, como gobernar el brazo robótico para alcanzar una posición determinada expresada en coordenadas cartesianas. El objetivo es que la pinza del robot alcance la posición en el menor número de iteraciones posibles, pudiéndose diseñar la recompensa de forma que optimice su comportamiento no sólo en el tiempo que tarda; sino, por ejemplo, en la energía que consume (sin necesidad de calcular el momento de inercia de cada articulación respecto a su ángulo de junta en cada momento... únicamente con una medida de los ángulos de junta y de la intensidad de corriente consumida en cada momento[3]).

Si el algoritmo es capaz de controlar un conjunto de acciones tan complejo como los

[2]Nótese la influencia de la psicología conductual al analizar la similitud entre el concepto de recompensa y el concepto de refuerzo de Paulov

[3]La matemática que sustenta los algoritmos de RL puede ser áspera, pero el planteamiento de los problemas es (y debe ser) sencillo. Una ventaja para un programador poco dado a escribir código.

de un brazo robótico humanoide, la resolución de tareas más complejas será probablemente un problema de planteamiento, pero no de capacidad.

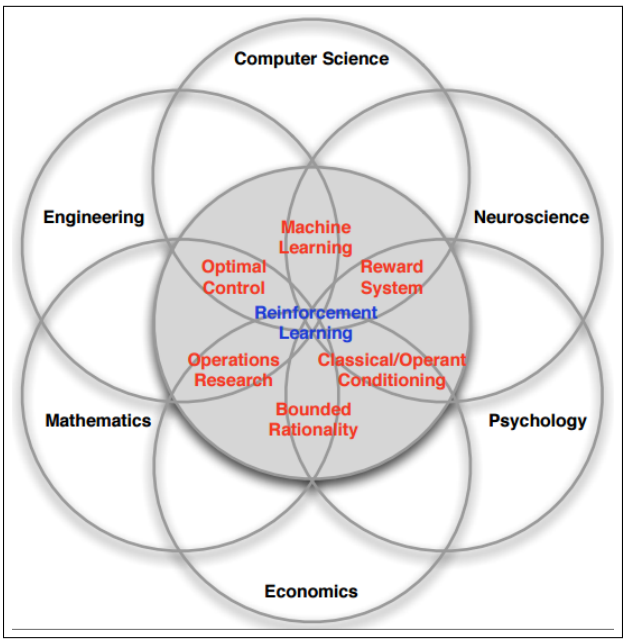


Figura 1.1: Diagrama de Venn que relaciona el aprendizaje por refuerzo con otras disciplinas[3].

1.1.1. Relación con otras ramas de la inteligencia artificial

El aprendizaje automático o *Machine Learning* formaliza problemas de forma que se puedan solventar aplicando un algoritmo iterativo. Estos algoritmos algoritmos aprenden patrones ocultos o descubren tendencias en conjuntos de datos. Estas características extraidas de los datos permiten construir un modelo simplificado de la realidad del cual se extrae la solución al problema planteado. Existen dos grandes enfoques bien diferenciados: el aprendizaje supervisado y el aprendizaje no supervisado.

En el aprendizaje supervisado, distinguimos entre datos de entrenamiento y datos problema, estando los primeros ya correctamente etiquetados o clasificados, se espera que el modelo o algoritmo aprendido sea capaz de etiquetar correctamente los datos del problema mediante un algoritmo que genera predicciones de forma iterativa.

En cambio, el aprendizaje no supervisado puede o no dividir sus datos de entrada entre datos de los datos de entrenamiento y datos problema, pero en ningún caso se incluyen las etiquetas, y es el algoritmo quien encuentra patrones que permitan agrupar posteriormente esos datos.

El aprendizaje por refuerzo constituye un tercer paradigma de aprendizaje. Se parece a un aprendizaje no supervisado porque ambas técnicas basan la resolución de sus

problemas en la forma de plantear el entrenamiento. Pero en problemas de RL se tiene en cuenta la secuencia de acontecimientos, como veremos en el siguiente capítulo al hablar de las **cadenas de Markov**.

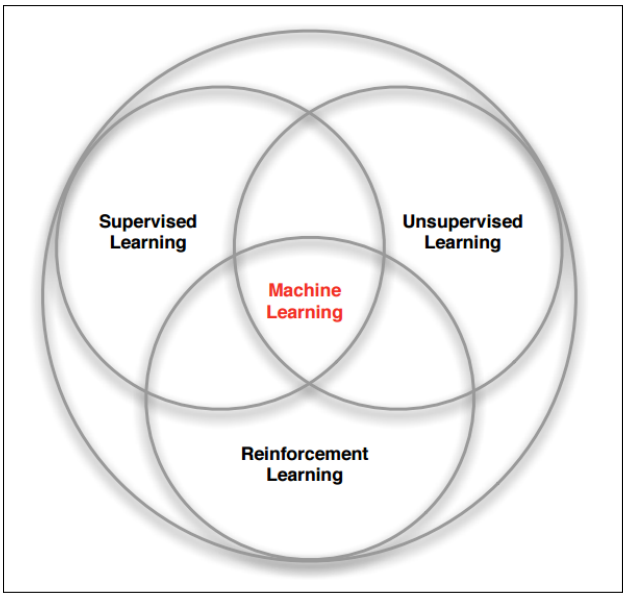


Figura 1.2: Diagrama de Venn mostrando la relación entre ML, RL y otros campos de la Inteligencia Artificial [3].

Los algoritmos de *Reinforcement learning* guardan similitudes con estos dos grandes grupos de *Machine Learning*, pero no se superponen totalmente 1.2, constituyendo una categoría aparte. Algunas de las estas técnicas, como el uso de redes neuronales profundas en lo que se conoce como *Deep Machine Learning*, han revolucionado los algoritmos de RL propiciando su popularización en la última década [4].

1.1.2. El agente en su entorno

En un problema de RL, el conjunto de nuestro robot con su algoritmo se conoce como **Agente**. El agente interactúa consigo mismo y con su **entorno**, que es la representación de la tarea asignada, mediante un conjunto finito de **acciones**. El entorno es reactivo: responde a las acciones del agente, surgiendo así la **interacción** agente-entorno. Como consecuencias de estas acciones, el entorno modifica su **estado**. y el agente recibe una **recompensa**.

El entorno está parametrizado por un conjunto de variables relacionadas con el problema, que serán aquellas susceptibles de ser modificadas por la acción del agente. En nuestro brazo robótico virtual, el color de la mesa sobre el que se monta puede ser importante, y constituir una variable de entorno; sin embargo, el precio de la misma es,

probablemente, una variable totalmente irrelevante para el problema y debe quedar fuera de la construcción del entorno.

Este conjunto de variables relevantes, y el rango de todos los valores necesarios para parametrizar el entorno, constituye un espacio vectorial conocido como **espacio de estados**. Cada estado particular representa una instancia del entorno en un momento t determinado.

ebido a que a menudo consideramos que el agente no tiene acceso al estado completo del entorno, de manera estricta se le llama **espacio de observaciones** a la parte del estado. Si una acción no tiene efecto alguno sobre una variable del entorno, esta no constituye una variable de estado; sin embargo, pueden existir variables de estado no observables susceptibles de ser modificadas mediante la acción del agente. En este trabajo el robot virtual ofrce un entorno totalmente observable, por lo que se solapa el significado de **estado** con el de **observación**.

Hablamos de que el agente recibe una **recompensa** por cada acción, pero se trata de un valor escalar numérico que debe aproximar cuán buena ha sido la acción del agente de cara a solucionar la tarea total. También podría apoximarse este concepto al de «castigo». En este caso, no se maltrata físicamente al brazo robótico, ni se le insulta; sencillamente, el valor de la recompensa es un escalar negativo. Si podemos establecer una **función de recompensa** o *reward function* que relacione la secuencia estado-acción-estado con la recompensa esperada final, podremos escoger la acción que maximice el valor de esta función.

La suma de las recompensas recolectadas al realizar una tarea se llama **retorno**, aunque en la bibliografía se confunde en ocasiones con la recompensa.

En problemas como una competición no será posible obtener ninguna recompensa en función de los pasos realizados, sino únicamente al final de la partida, en función de si se ha ganado o se ha perdido, sin importar el camino necesario para llegar a vencer al oponente. Sería una recompensa positiva si el agente ganara el juego (porque el agente habría logrado el resultado deseado) o una recompensa negativa (penalización) si el agente perdiera el juego.

1.1.3. Secuencia de aprendizaje

La figura 1.3 ilustra la secuencia de control de un brazo robótico como un problema de RL. El aprendizaje se produce en bucle hasta completar la tarea deseada (un fin por agotamiento también se considera un episodio, aunque termine en fracaso; recordemos que el objetivo es aprender por aciertos, pero también por errores

Las interacciones entre el agente y el entorno se prolongan durante varios pasos. A cada ciclo lo llamaremos iteración t o *step*, y marca una interacción entre el agente y el entorno en el tiempo. También se usa iteración como sinónimo de time step cuando estamos programando, puesto que los *time step* son los índices de los bucles.

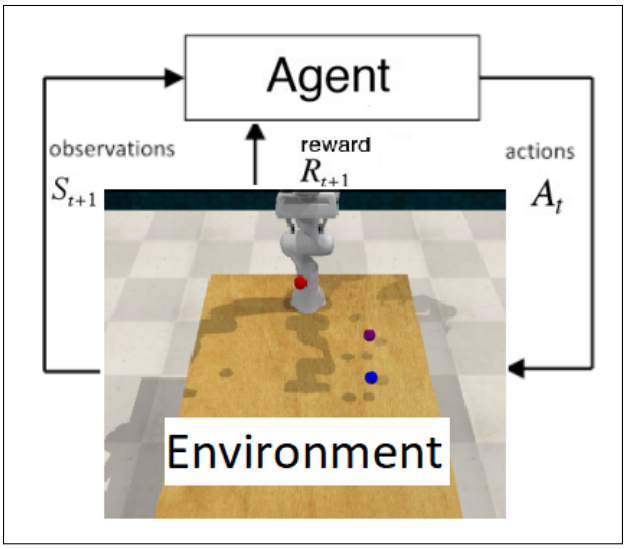


Figura 1.3: Brazo robótico como un problema de RL.

En cadapaso, el agente observa el entorno, realiza una acción y recibe un nuevo estado y recompensa, consideramos entrada en el nuevo step $t+1$. Con estos, el agente decide cuál es la siguiente acción que debe realizar. Esta acción acaba teniendo un efecto en el entorno que cambia su estado, recibiendo una nueva observación y una nueva recompensa que tendrá en cuenta en $t+2$.

1.1.4. Política

Parafraseando al personaje «Hannibal», de la serie de televisión «El equipo A» (*"The A-Team"* en versión original), al agente le encanta que «los planes salgan bien». El comportamiento del agente, que ejecuta sus tareas mientras aprende, se basa en un plan preestablecido. Este plan es el que, a su vez, en sus entrenamientos en la idea de maximizar el retorno obtenido en cada episodio.

Este plan o **política** define el comportamiento del agente en un entorno. Considera el estado y la observación obtenida por la última acción, y determina la siguiente acción a realizar. Una política es una función que determina la siguiente acción a realizar cuando el agente se encuentra en un estado del entorno.

En un agente no entrenado, se puede partir de una política uniforme, donde la probabilidad de todas las acciones es idéntica, o una política elegida al azar. A medida

que va entrenándose puede aprender a optimizar su política para alcanzar una política mejor. En el primer episodio, este agente realiza una acción aleatoria en cada estado y trata de saber si la acción es buena o mala en base a la recompensa obtenida. A través de una serie de episodios, el agente aprenderá a realizar buenas acciones en cada estado.

En general las políticas también pueden clasificarse como deterministas o estocásticas. Para entornos sencillos y discretos, toda la política puede escribirse con una tabla donde todos los posibles estados se representen por una tabla, y en las casillas de esa tabla detalle qué acciones ejecutar. Hablamos entonces de una **política determinista**.

Pero en nuestro problema nos encontraremos con políticas más generales con comportamiento estocástico. Una **política estocástica** tiene una distribución de probabilidad sobre las acciones que un agente puede realizar en un estado dado. Así que en lugar de realizar la misma acción cada vez que el agente visita el estado, el agente realiza diferentes acciones basado cada vez en una distribución de probabilidad[5]. Cuando el espacio de acciones es discreto, decimos que el agente sigue una **política categórica**, mientras que una **política gaussiana** se sigue cuando el espacio de acciones es contínuo, definido por una media μ_a y una desviación estándar σ.

1.2. Elección del algoritmo óptimo

Para decidir si un algoritmo es mejor que otro, debemos contar con un criterio objetivo.

¿Cuántas muestras son necesarias para obtener una buena política? La figura 1.4 muestra las familias de algoritmos de RL clasificadas en función de su **eficiencia de muestro**. Los algoritmos *off policy* pueden mejorar la política sin generar un nuevo muestreo al cambiar la política, una ventaja respecto a los algoritmos *on policy*, que deben generar un nuevo muestreo cada vez que se actualiza la política.

En general, una mayor eficiencia está asociada a un menor **tiempo de computación**.

Estabilidad El algoritmo converge a una secuencia de estados. Y en caso de que converja, ¿a dónde? ¿siempre y en todo caso? La **estabilidad** del algoritmo será otro factor clave para evaluar su rendimiento.

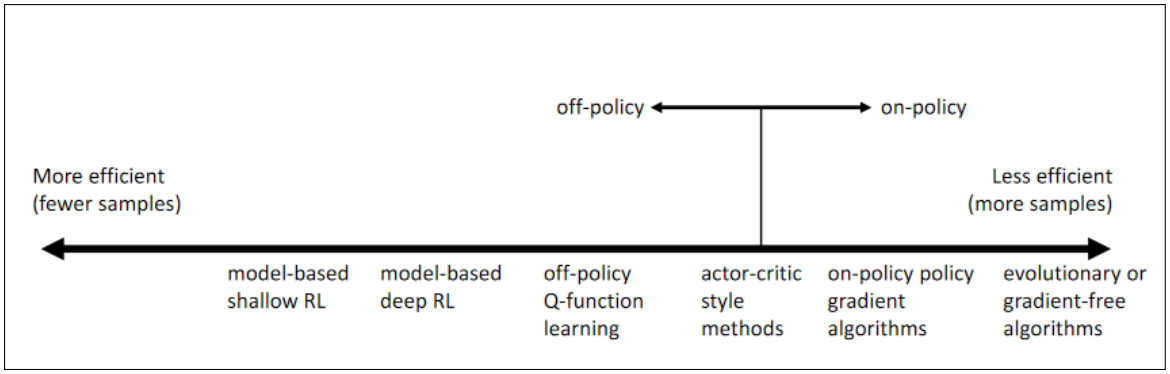


Figura 1.4: Clasificación de los métodos obtenidos en función de la eficiencia en términos de muestreo[1].

1.3. Objetivos

En este trabajo se van a investigar y comparar diversos algoritmos recientes de *Deep Reinforcement Learning*, tales como **Deep Q-Network**, **Soft Actor-Critic** o **Proximal Policy Optimization** para el control flexible de robots.

La evaluación se va a realizar en un entorno de simulación virtual realista **CoppeliaSim**, entrenando una tarea cuya complejidad radica en la ausencia de supervisión y la cantidad de datos a obtener del entorno (más de 40 observaciones para un robot con 7 juntas).

Para conseguir un entorno funcional, aprovechan las herramientas que permiten integrar en **Python** tanto el simulador como librerías avanzadas de algoritmos de *Reinforcement Learning* con la idea de evaluar su idoneidad al problema planteado.

Todo el trabajo de programación se llevará a cabo en lenguaje **Python**, bien mediante programas o *scripts*, bien mediante entornos de ejecución tipo **Jupyter** o **Colab**. Tanto el código disponible en los anexos como las imágenes, animaciones y vídeos utilizados en esta memoria están disponibles en el repositorio `https://github.com/albertost85/RoboticArm` de Github.

1.4. Metodología

Dada la novedad que representa para el autor de este proyecto el campo de la inteligencia artificial y las técnicas de *Deep Reinforcement Learning*, las bases del trabajo se fundamentan sobre un estudio teórico de la propuesta planteada por [1] que ha evolucionado hasta eclosionar en los algoritmos que resuelven satisfactoriamente esos mismos problemas planteados en los años 80 utilizando redes neuronales profundas y

la potencia de cómputo de los ordenadores actuales[4] [6] [7].

El núcleo del trabajo descansa sobre el brazo robótico *Franka Emika Panda* virtualizado en la herramienta de simulación robótica **CoppeliaSim**. Sobre la herramienta de simulación robótica, se instalan las librerías **PyRep**[4] **RLBench**[5], que permiten modelar el entorno robótico como un problema de *Reinforced Learning* siguiendo el paradigma **Gym** de **OpenAI**[6].

Se aborda la complejidad que representa utilizar algoritmos de que permiten solventar problemas de *Reinforcement Learning* con técnicas *Deep Learning* mediante el uso de un conjunto de herramientas o *framework* incluidas en la librería **Stable Baselines 3**. Esta librería incluye las últimas versiones de los algoritmos derivados de *Actor-Critic* que han eclosionado desde 2015.

La elección del algoritmo óptimo para un problema robótico es el primer paso en el capítulo dedicado a procedimientos y metodología. Pero ha sido necesario simplificar el problema antes de abordar el problema, construyendo una función de recompensa adecuada, comparando dos estrategias distintas: una que penaliza el tiempo que tarda cada episodio en ser resuelto, y otra que añade un premio por finalizar el episodio en un tiempo inferior al límite.

Se evalúan parámetros como el tamaño de la red neuronal, el tamaño del búfer de experiencias pasadas o el peso de la exploración al comienzo del algoritmo, creando en cada caso modelos entrenados que permiten comparar la eficacia de cada estrategia.

1.5. Motivación

La importancia de los brazos robóticos en la industria no ha dejado de crecer desde que *General Electric* presentase su robot **Unimate** en 1961 [8]. Desde entornos manufactureros a la ejecución de tareas en entornos peligrosos, su uso como herramienta redunda en la mejora de la eficiencia y la productividad de la industria.

Paralelamente a su evolución en la industria, han ido apareciendo modelos más potentes o más precisos dependiendo de las tareas para las que hayan sido concebidos, haciéndoles aptos como la cirugía[9]. Sin embargo, una característica definitoria es su flexibilidad. Como ejemplo de máquina herramienta casi perfecta, una misma cadena cinemática puede adaptar varios cabezales, uniendo a la agilidad para alcanzar cual-

[4] https://github.com/stepjam/PyRep

[5] https://github.com/stepjam/RLBench

[6] OpenAI es una compañía de investigación de inteligencia artificial sin fines de lucro que tiene como objetivo promover y desarrollar inteligencia artificial: https://gym.openai.com/

quier espacio del espacio cartesiano en el espacio una gran variedad de funciones a realizar.

El control de estos brazos robótico es complejo. Entornos de programación específicos para cada familia aumentan el coste que de por sí tienen los sistemas mecánicos que componen el robot. Los algoritmos de control numérico tradicionalmente asociados a este tipo de máquinas son precisos, pero requieren una programación exhaustiva de las tareas a realizar. Este tipo de programación no aprovecha la flexibilidad que ofrece un brazo robótico. Al aumentar el número de juntas que componen el brazo, es posible alcanzar trayectorias con una combinación de trayectorias, las cuales no siempre es posible estimar por métodos algebraicos. Es un problema conocido como cinemática inversa del robot, que en este trabajo se omitirá deliberadamente.

Utilizando como entrada de datos al sistema todas las variables que ofrece un entorno robótico virtual, 3 tal como se expone en el capítulo, no se programa, al inicio del experimento, una estrategia de control sistemática[7]. Las acciones del brazo robótico sin entrenar serán completamente aleatorias; su movimiento es caótico y una reproducción en vídeo del entorno deja bien a las claras que no existe una estrategia digna de tal nombre encaminada a resolver el problema. Definiendo únicamente una función de recompensa (unas pocas líneas de código) y contando con un tiempo de simulación, la máquina «aprende» por sí misma a llegar al objetivo.

A pesar del coste computacional que representa explotar un algoritmo de *Reinforcement Learning* en un entorno de estados «continuos» y acciones también «continuas» necesarias para gobernar 7 juntas robóticas, la utilidad de estas técnicas radica en la simplificación del problema y el éxito en el funcionamiento de un robot ya entrenado.

Para comprender la importancia de esta simplificación, es posible extender estas mismas técnicas a un brazo robótico completamente diferente. O al mismo brazo robótico, pero en esta ocasión con una de sus juntas «averiada». Y todo ello sin necesidad de reprogramar el entorno, a costa del coste computacional que requiere entrenar de nuevo al robot (en comparación entrenamiento *ad-hoc*).

1.6. Organizaión

En el capítulo 2 se expone todo el marco teórico detrás del *Reinforcement Learning*, explicando en de detalle las dos familias clásicas de algoritmos de aprendizaje por refuerzo profundo s a partir de las cuales se construyen los algoritmos *Actor-Critic* y

[7] A excepción de una función auxiliar encargada de calcular la distancia al objetivo, con objeto de evaluar el comportamiento del robot

evoluciones posteriores como A2C, PPO, DDPG, SAC y TD3. En el capítulo 3 se analiza la virtualización de un brazo robótico real (el modelo *Franka Emika Panda*, hasta quedar totalmente asimilado a un entorno compatible con la herramienta *Gym* junto al *framework Stable Baselines 3* que permite la implementación sencilla de todas estas técnicas, para ensayar su aplicación en en problema robótico. En el capítulo 4 se describen los procedimientos seguidos con el robot virtual para determinar el mejor algoritmo de control para el problema inicial, así como los parámetros óptimos, y su extensión a otros problemas relacionados con la robótica, agrupándose junto a los resultados obtenidos de todos estos experimentos. Finalmente, en el capítulo 5 de conclusiones se sistematiza el conocimiento adquirido en este trabajo exponiendo las técnicas aprendidas para controlar un brazo robótico por *Reinforcement Learning*.

Capítulo 2

Reinforcement Learning

Casi todos los problemas de RL se pueden formalizar como un proceso de toma de decisiones de Markov (MDP en adelante). Incluso en los problemas donde el entorno es parcialmente observable.

Sobre este proceso de decisiones se construye la idea dentral detrás del libro «Reinforcement Learning: An Introduction»[1], de Richard S. Sutton y Andrew G. Barto[1] , que supuso el nacimiento del aprendizaje por refuerzo. En la elaboración de este capítulo se han utilizado materiales del curso ofrecido en **DeepMind** por David Silver [2], un discípulo de Richard Sutton conocido por desarrollar el primer algoritmo capaz de vencer jugando al juego *Go* [10]. Para una lectura introductoria, la mejor recomendación en español es el libro de Jordi Torres, Introducción al Aprendizaje por Refuerzo Profundo, publicado este mismo año [5].

La notación matemática de los anexos sigue la notación de [1], pero en este apartado se ha seguido una notación más relajada que simplifica algunos detalles de notación del libro de Jordi Torres «Introducción al aprendizaje por refuerzo profundo»[5], también basada en el libro de Sutton, con algunas licencias ya que no incluye demostraciones matemáticas de la mayoría de fórmulas.

[1] El Dr. Richard S. Sutton es actualmente un investigador de **DeepMind** y un reconocido profesor de ciencias de la computación en la Universidad de Alberta. El Dr. Sutton es considerado uno de los padres fundadores del aprendizaje por refuerzo computacional moderno. El Dr. Andrew G. Barto es un profesor emérito en la Universidad de Massachusetts Amherst y fue el director de la tesis doctoral del Dr. Sutton.

[2] https://deepmind.com/learning-resources/-introduction-reinforcement-learning-david-silver

2.1. Problema de aprendizaje por refuerzo como un MDP

Se construye sobre un proceso de recompensa de Markov[3], al que se añade un conjunto finito de acciones $\mathcal{A}$[4] que extiende la tupla hasta $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$. Para más información, ver el anexo C.

La recompensa por una acción $r(\boldsymbol{s}, \boldsymbol{a})$, y las probabilidades de transición de un estado actual a un estado siguiente $p(\boldsymbol{s}'|\boldsymbol{s}, \boldsymbol{a})$ conforman un proceso de Markov de toma de decisiones.

La meta del aprendizaje por refuerzo es encontrar una política π_θ que determine, para cada estado del espacio de estados, una acción del espacio de acciones que maximice la recompensa.

En un entorno donde la observabilidad (en cuanto a los parámetros que incluyen en el problema) es total, las observaciones o_t representan el estado s_t del sistema en ese instante t. Para cada estado, la **política** $\pi_t heta(a_t|s_t)$ asocia una distribución de probabilidad al conjunto de acciones $a_t \in \mathcal{A}$ asociado a ese estado s_t.

Un problema de *Reinforcement Learning* tiende hacia una política óptima, que maximice el **retorno** en el problema planteado. Para ello, el problema aplica una determinada política π_θ en un momento determinado. θ representa una determinada elección de política; pueden ser los parámetros de un algoritmo de decisión lineal, o los pesos de una red neuronal.

La probabilidad de una determinada cadena de acciones o **trayectoria** $s_1, a_1, s_2, a_2, ..., s_T, a_t$ puede representarse como:

$$p_\theta(\tau) = p_\theta(\boldsymbol{s}_1, \boldsymbol{a}_1, ..., \boldsymbol{s}_T, \boldsymbol{a}_T) = p(\vec{s}_1) \prod_{t=1}^{T} \pi_\theta(\boldsymbol{a}_t|\boldsymbol{s}_t) p(\boldsymbol{s}_{t+1}|\boldsymbol{s}, \boldsymbol{a}_t) \tag{2.1}$$

[3] La recompensa inmediata ahora puede estar asociada a una acción, o a un estado.

[4] La matriz de probabilidades no sólo tiene en cuenta el estado precedente y el estado final, sino que está asociada a una acción.

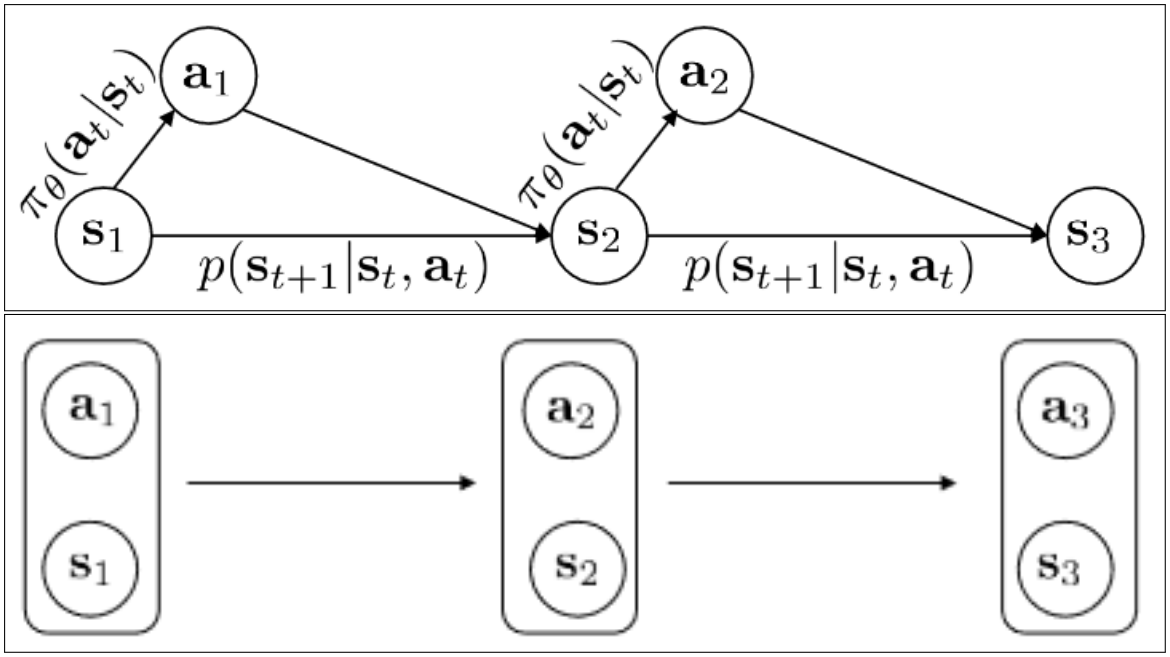


Figura 2.1: MDP considerando estados y acciones como eslabones de una cadena de Markov[1].

La cadena de acciones de la figura 2.1 es una cadena de Markov donde se redefine la probabilidad conjunta de la siguiente acción y el siguiente estado como el producto entre la probabilidad del estado siguiente condicionado a una acción $a_t = k$, y la política asociada a la acción siguiente condicionada al estado siguiente:

$$p((\boldsymbol{s}_{t+1}, \boldsymbol{a}_{t+1}|(\boldsymbol{s}_t, \boldsymbol{a}_t)) = p((\boldsymbol{s}_{t+1}|\boldsymbol{s}_t)\pi_\theta(\boldsymbol{a}_{t+1}|\boldsymbol{s}_{t+1}) \tag{2.2}$$

2.2. Funciones de valor

2.2.1. Función valor de estado V

«¿Qué esperar desde aquí?». Más formalmente, la función de valor del estado mide la bondad de cada estado según el retorno con descuento al seguir una política π determinada.

La función valor es la que realmente evalúa una secuencia de estados, y se expresa por $v(s)$. En un momento determinado t, devuelve qué retorno se espera G_t siendo s el estado actual $S_t = s$:

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] = \mathbb{E}_\pi[\sum_{j=0}^{T} \gamma^j r_{t+j+i}|S_t = s] \tag{2.3}$$

Esta expresión describe el valor esperado del retorno con descuento G_t, comenzando desde el estado s_t en la iteración t y considerando la política π presente en ese momento, hasta el final de la trayectoria en T.

2.2.2. Esperanza recompensa esperada.

Hablamos del concepto matemático **esperanza** $\mathbb{E}$ porque se evalúa en un entorno estocástico. En RL, no se focaliza una determinada trayectoria, sino que se computa la recompensa esperada por todo un conjunto de trayectorias.

Una forma de construir una esperanza o valor razonable consiste en sumar la recompensa inmediata de todo el conjunto de estados y acciones seguidas.

$$E_{\tau \sim p_\theta(\tau)} \left[\sum_{t=1}^{T} r(\boldsymbol{s}_t, \boldsymbol{a}_t) \right] \tag{2.4}$$

Otra forma, consiste en expresar la esperanza, como esperanzas anidadas, de forma recursiva p. De forma que la expectativa es igual a la expectativa obtenida en el primer estado, de la expectativa de la primera acción, de la recompensa obtenida por esa primera acción en ese primer estado.:

$$E_{\boldsymbol{s}_1 \sim p(\boldsymbol{s}_1)} \left[E_{\boldsymbol{a}_1 \sim \pi(\boldsymbol{a}_1|\boldsymbol{s}_1)} \left[r(\boldsymbol{s}_1, \boldsymbol{a}_1) + ... | \boldsymbol{s}_1 \right] \right] \tag{2.5}$$

El segundo eslabón se construye de forma similar, pero condicionado al estado $\boldsymbol{s}_1$:

$$E_{\boldsymbol{s}_1 \sim p(\boldsymbol{s}_1)} \left[E_{\boldsymbol{a}_1 \sim \pi(\boldsymbol{a}_1|\boldsymbol{s}_1)} \left[r(\boldsymbol{s}_1, \boldsymbol{a}_1) + E_{\boldsymbol{s}_2 \sim p(\boldsymbol{s}_2|\boldsymbol{s}_1, \boldsymbol{a}_1)} \left[E_{\boldsymbol{a}_2 \sim \pi(\boldsymbol{a}_2|\boldsymbol{s}_2)} \left[r(\boldsymbol{s}_2, \boldsymbol{a}_2) + ... | \boldsymbol{s}_2 \right] | \boldsymbol{s}_1, \boldsymbol{a}_1 \right] | \boldsymbol{s}_1 \right] \right] \tag{2.6}$$

Hay una serie de esperanzas que están condicionadas sólo a $\boldsymbol{s}_1$, $\boldsymbol{s}_2$....

2.2.3. Función de valor de acción Q

La función de valor de la acción o función Q define el valor de realizar una acción determinada, determinada por una cierta política π, en un estado concreto s. Se puede expresar como:

$$Q_\pi(\boldsymbol{s}_t, \boldsymbol{a}_t) = \mathbb{E}_\pi[G_t | S_t = s, A_t = a] \tag{2.7}$$

$$= \mathbb{E}_\pi \left[\sum_{j=0}^{T} \gamma^j r_{t+j+1} | S_t = s, A_t = a \right] \tag{2.8}$$

En Sutton:

$$\sum_{t'=t}^{T} \mathbb{E}_{\pi_\theta} \left[r(\boldsymbol{s}_{t'}, \boldsymbol{a}_{t'}) | \boldsymbol{s}_t, \boldsymbol{a}_t \right] \tag{2.9}$$

Es una función importante; de alguna manera, maximizar la función valor es el objetivo todo algoritmo de RL.

Determinación de Q^π Aunque no se pueda conocer con exactitud, hay formas de aproximar el valor de Q^π.

2.2.4. Función V vía función Q

Las funciones V y Q están relacionadas[5].

En un entorno estocástico, la suma de las probabilidades de todas las acciones salientes de un estado es siempre igual a 1. Por lo tanto, la función del valor de estado es equivalente a la suma de las funciones de valor de acción de todas las acciones salientes de ese mismo estado s, multiplicado por la probabilidad de cada acción, es decir, la esperanza.

$$V_\pi(s) = \sum_a \pi(a|s) Q_\pi(s,a) \tag{2.10}$$

2.2.5. Valor práctico de las funciones Q y V

Si tenemos una política π, y conocemos la función $Q^\pi(\boldsymbol{s}, \boldsymbol{a}$, ¿Cómo podemos mejorar π?

Determinación total Una aproximación consiste en encontrar la política $\pi'(\boldsymbol{a}|\boldsymbol{s})$ que maximice el valor de $Q^\pi(\boldsymbol{s}, \boldsymbol{a}$, y asignarle el valor 1.

Gradiente Se puede utilizar un gradiente para incrementar la *bondad* de las acciones $\boldsymbol{a}$. Si el valor $Q^\pi(\boldsymbol{s}, \boldsymbol{a}$ es mayor que $V^\pi(\boldsymbol{s})$, entonces $\boldsymbol{a}$ es mejor que la suma ponderada de todas las acciones posibles. Puede a´si modificarse la política π para incrementar el valor de $\boldsymbol{a}$ y disminuir el resto.

2.3. Redes neuronales

El concepto de red neuronal se ha popularizado con la difusión de las técnicas de Machine Learning, pero es un concepto nacido en los años 60 a partir de la publicación de Marvin Minsky y Seymour Papert sobre la idea previa del perceptrón. El perceptrón es un elemento teórico que imita el comportamiento de una neurona. Cuando una

[5] La función valor $V^\pi(\boldsymbol{s})$ representa el valor medio del valor esperado $Q(\boldsymbol{s}, \boldsymbol{a})$ de todas las acciones, tomando una cierta política $\pi(\boldsymbol{a}|\boldsymbol{s})$: $V^\pi(\boldsymbol{s}) = E\left[Q^\pi(\boldsymbol{s}, \boldsymbol{a}\right]$

neurona es estimulada, a través de alguna de sus dendritas, por un pico de tensión superior a los $35(mV)$, ésta reproduce un pulso de tensión entre los $60(mV)$ y los $85(mV)$[11] que recorre el axón hasta conectar con la dendrita de otra neurona.

El perceptrón reproduce este comportamiento mediante una función de activación. Una de las más conocidas es la función sigmoide, que aproxima una función todo/nada:

$$sigm(x) = \frac{1}{1+e^{-x}} \tag{2.11}$$

Esta función toma valores próximos a 0 en el subdominio de los números reales negativos, y próximos a 1 en el subdominio de los números reales positivos. Es una función fácilmente derivable $f'(x) = f(x)(1-f(x))$.

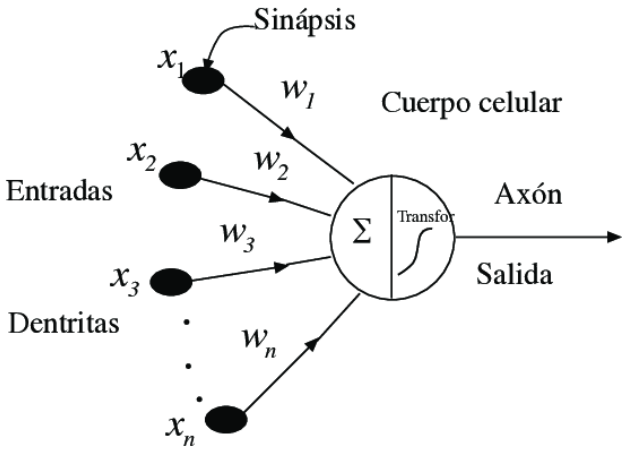


Figura 2.2: Representación funcional de un perceptrón y semejanza con una neurona artificial[12].

Los perceptrones o neuronas virtuales, como la representación de la figura 2.2 se agrupan por capas, donde todas las neuronas se interconectan a todas las neuronas de la siguiente capa. A este tipo de conexión se les conoce como capas densas o **MLP** *Multi Layer Perceptron*. Cada una de estas conexiones tiene asociado un valor numérico, un «peso», de tal forma que la entrada a la función de activación de un perceptrón particular es la suma de los valores de los perceptrones de la capa inmediatamente anterior multiplicados por sus respectivos pesos.

De forma estática, siempre que no tengan un tamaño excesivo, estas redes neuronales se pueden implementar en microcontroladores sin grandes quebraderos de cabeza en cuanto al tiempo de procesamiento. Para que la red neuronal sea adaptativa y «aprenda», es necesario adaptar los pesos de las interconexiones. En Machine Learning, esta actualización se hace por gradientes mediante una técnica conocida como *backpropagation*. Esta técnica implica calcular las derivadas de cada función de activación que

constituye la red neuronal, aplicar la regla de la cadena para obtener derivadas sucesivas, y actualizar los pesos de la red, en cada paso de ejecución. Afortunadamente, herramientas como **PyTorch** o **Tensorboard** calculan los gradientes y aplican la técnica de propagación inversa de forma automática, dejando al programador la tarea de elegir el tipo de red, dimensionar su arquitectura, y elegir el tipo de funciones de activación y de salida de la red.

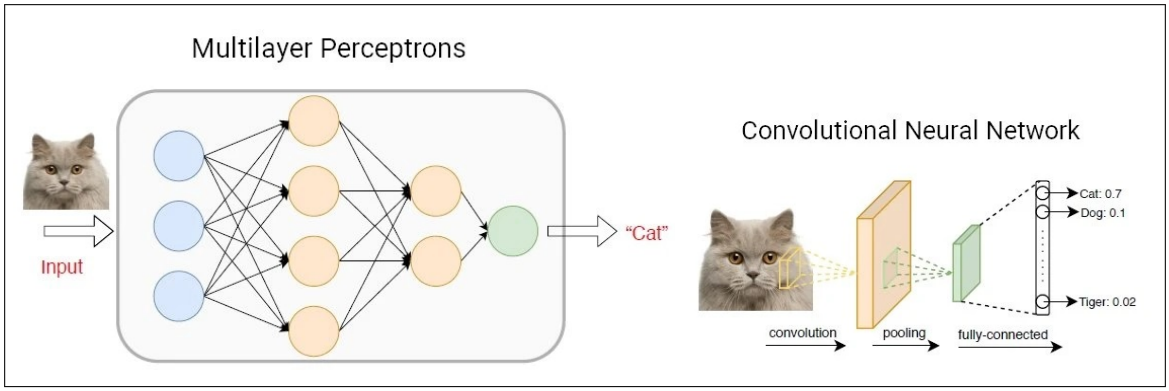


Figura 2.3: Diferencias entre MLP y CNN[13].

Existen dos grandes familias: las redes neuronales compuestas por multicapas o *MultiLayer Perceptron* **MLP**, especialmente indicadas para observaciones compuestas por vectores de estado, las redes neuronales convolucionales o *Convolutional Neuron Network* **CNN**, indicadas para el tratamiento de observaciones compuestas por imágenes. Las redes convolucionales suelen tener un mayor número de capas, pues añaden una capa extra al tratamiento de conjuntos de datos de entrada. En la figura 2.3[6] se ilustra esta diferencia.

Las redes neuronales pueden utilizarse para representar una política (la entrada puede ser la distribución de probabilidad del estado actual), y ofrecer una salida que represente la distribución de probabilidad de las acciones que puede tomar el agente. O pueden utilizarse para calcular la función valor del estado. Si ajustamos los pesos de la red neuronal, también cambiará la distribución de probabilidad de las acciones (o la función valor), modificando el comportamiento del agente. Aunque no tengamos una definición formal para construir una política óptima, podemos alcanzarla, o al menos un óptimo local, actualizando por la técnica del gradiente los pesos de la red neuronal de forma iterativa.

[6] https://viso.ai/deep-learning/deep-neural-network-three-popular-types

2.4. Métodos *Value-based*

Aunque no exista un método general que permita optener una política óptima o sistematizar una función valor para cada problema, existen algoritmos que aproximan estas funciones mediante iteraciones. Los métodos value-based optimizan la función Q para obtener las preferencias de elección de las acciones por parte del agente.

En un entorno compuesto por estados y acciones discretas, podemos asignar valores en una tabla o **función Q** a cada par estado acción(s, a). El algoritmo **Q-Learning** precisamente obtimiza una tabla de este tipo.

En un entorno de acciones y observaciones discretas, o entornos continuos más complejos, esta aproximación resulta inviable. En estos casos, el algoritmo DQN o **Deep-Q-Network** expande las técnicas usadas en **Q-Learning** mediante el uso de redes neuronales profundas[14] . Este algoritmo reveló su utilidad en 2015, cuando un ordenador aprendió a jugar a 49 videojuegos Atari diferentes, previamente implementados en **Gym**[14], mejor que los humanos[7]

La red neuronal de este algoritmo está diseñada para producir en una sola pasada un valor Q para cada una de las acciones posibles, evitando así tener que ejecutar la red individualmente para cada acción y ayuda a aumentar la velocidad de aprendizaje del agente, lo cual permite elegir más rápidamente la acción que tenga el valor Q máximo[5].

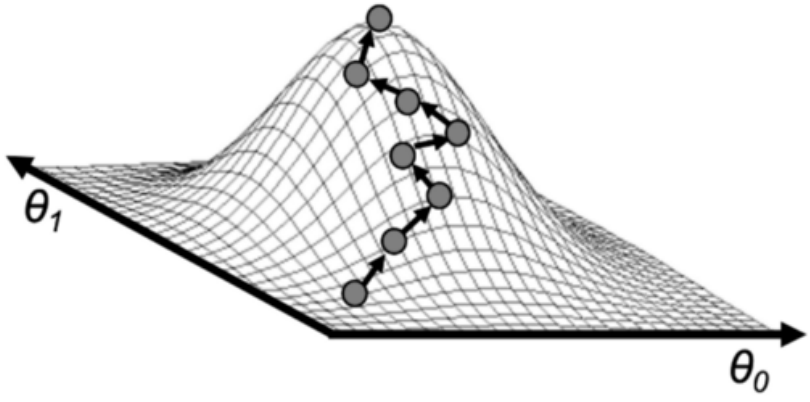


Figura 2.4: Representación visual del algoritmo por descenso de gradiente para una red neuronal de 2 parámetros θ_1 y θ_2[5]

.

Para hacer esto, debemos calcular los valores estimados usando la ecuación de Bellman y, luego, considerar que tenemos un problema de aprendizaje supervisado. Para conocer el en profundidad el algoritmo DQN, véase anexo G.

2.5. Métodos *Policy-search*

La familia de algoritmos *policy-search* nos permiten buscar directamente una función de política en lugar de una función de valor. Es decir, en lugar de entrenar una red neuronal para que genere los valores de las acciones o estados, entrenaremos una red neuronal para que indique cuál puede ser la siguiente acción a realizar.

Los métodos *policy-search* ofrecen ciertas ventajas sobre los métodos *value-based*. Ahora, la red neuronal de un algoritmo *policy-based* retornará las probabilidades de cada acción posible, y la exploración se realiza automáticamente al elegir una acción siguiendo la distribución de probabilidades (no hace falta incluirla explícitamente como paso en el algoritmo). Además, obteniendo la acción de esta forma, tampoco hace falta añadir al algoritmo «trucos» como experience replay o target network para mejorar la estabilidad del entrenamiento de la red neuronal.

Los métodos *policy-gradient* son una subclase de los métodos que ajustan los parámetros de la red neuronal para una política óptima a través del ascenso del gradiente, que nos indica en cada time *step* la dirección en la cual debemos cambiar los parámetros de la red neuronal para mejorar la política (subir la hacia la cima).Para conocer más sobre un algoritmo de esta familia, el **REINFORCE**, véase el anexo G. En cambio, los métodos *value-based* pueden beneficiarse de datos antiguos obtenidos de aplicar otras políticas anteriores. Esto implica que los métodos *policy-based* suelan ser menos eficientes en cuanto a muestras y que requieran una mayor interacción con el entorno. Recuerde que los métodos *value-based* pueden beneficiarse de grandes búferes, como la técnica *experience replay*.

2.6. Métodos *Actor-Critic*

Los métodos *actor-critic* mezclan propiedades de *value-based* y *policy-search*.

Uno de los problemas que presentan los métodos basados en *policy-gradient* radica en la estabilidad; es decir, una alta variabilidad inherente en las probabilidades y los valores de recompensa esperados.

La varianza es un término estadístico que mide la dispersión de la distribución de probabilidad. Por ejemplo, en una distribución normal, una varianza baja representa una montaña picuda en la media, mientras que una distribución con una varianza alta la distribución dibuja una colina con laderas más aplanadas y una cima más baja.

La forma de calcular el gradiente de la función valor en el método REINFORCE era:

$$\nabla_\theta U(\theta) \leftarrow \sum_{t=0}^{H} \nabla \log \pi_\theta(a_t|s_t) G_t \tag{2.12}$$

Donde la recompensa descontada G_t en cada experiencia t actúa como un factor de escala. Imaginesmos 3 episodios consecutivos con valores dispares, siendo G_1 y G_2 idénticos, positivos, y G_3 un valor negativo mucho mayor que los anteriores. Las acciones tomadas en $t = 1$ y $t = 2$ reciben una pequeña recompensa, mientras que la tercera acción en q_3 recibe un valor negativov como recompensa. EL gradiende resultante al considerar los 3 episodios aleja la política enormemente de la acción tomada en el paso 3, y ligeramente la acerca a las acciones tomadas en los pasos 1 y 2. Sin embargo, imaginemos ahora que todos los valores de recompensa son positivos, siendo G_1 y G_2 valores positivos mucho mayoes a G_3, también positivo pero cecano a 0. En este caso, el gradiente dirigirá la política enormemente hacia las acciones tomadas en t_1 y $t = 2$, y prácticamente ignorará la acción tomada en $t = 3$.

La dependencia del gradiente de la política respecto al valor medio de las recompensas esperadas G_t es una desventaja en esete tipo de algoritmo. Sólo si consideramos en un muestreo cuasi infinito esta disparidad se hace insignificante. Una forma de atenuar este efecto consiste en sustraer a cada recompensa esperada el valor medio de las recompensas esperadas obtenidas en cada episodio. A este valor medio se le conoce como línea base o, en terminología de RL, **baseline**.

Parece una buena idea considerar, en vez de una secuencia de recompensas esperadas obtenidas, que este valor **baseline** dependa del estado. Para imaginarlo, volvemos al algoritmo DQN, donde cada tupla estado-acción tenía un valor $Q(a, s)$. Si el estado s tiene un valor intrínseco, podemos redefinir el valor acción-estado $Q(a, s)$ como:

$$Q(a, s) = V(s) + A(s, a) \tag{2.13}$$

Donde $V(s)$ es ese valor de **baseline** y $A(a, s)$ es un valor esperado en la transición acción-estado. No conocemos el valor $V(s)$ pero, al igual que en el algoritmo **DQN**, podemos utilizar una red neuronal para estimar el valor $V(s)$ en base a observaciones. Esta red neuronal se entrena de la misma manera que DQN utiliaba el valor óptimo de la ecuación valor Bellman para optimizar, por el método de los gradientes del error cuadrático obtenido, los pesos de la red neuronal. A esta red neuronal se la conoce como **Crítico**

Cuando tenemos, aunque sea por aproximación, un valor intrínseco para cada estado, es inmediato pensar usar este valor para optimizar el valor de la red neuronal que

estima el valor de la tupla acción-estado $V(s,a)$. Sin embargo, también puede utilizarse para optimizar los pesos λ que conforman una política π, optimizando directamente la red neuronal que dicta la política del agente. A esta red neuronal se la conoce como **Agente**.

En la práctica, estas redes se superponen, al menos en ciertos valores de entrada. Es posible diseñar la arquitectura de una red conjunta, donde la entrada sea el estado de observaciones, y sólo cambien en la capa externa, definiendo múltiples salidas para el **actor**, y una salída para el **crítico**, dos redes independientes, o una red con unas primeras capas compartidas y capas intermedias independientes. En cualquier caso, la entrada a ambas redes neuronales es la misma: el espacio de observaciones [15].

Algoritmo 2.1: Actor-Critic

```
1  Initialize random network parameters θ with random values
2  while not done
3      Play N steps in the environment with π_θ saving epochs of (s_t, a_t, r_t).
4      if (done)
5          R = 0
6      else
7          R = V_theta(s_t)
8      end
9      Update backwards:
10     for i = t − 1...t_start
11         R← r_i + γR
12         Accumulate policy gradients: Δθ_π ← ∇_θ θ_π + ∇_θ log π_θ(a_i|s_i)(R − V_θ(s_i))
13         Accumulate value gradients: Δθ_v ← ∇_θ θ_v + ∂(R−V_θ(s_i))/∂θ_v
14         Update networks with gradients.
15     end
16 end
17 return π
```

El control del brazo robótico es tan complejo en términos de RL, que no basta ninguno de las familias anteriormente descritas. Hay que recurrir directamente a métodos *actor-critic* tales como A2C/A3C[16], DDPG[17], TD3[18], SAC[19] o PPO[20].

2.7. Deterministic Policy Gradients

Podemos aplicar de la misma idea que constituyen los métodos *Actor-Critic* a un método *off-policy*.

La función del actor es convertir un estado en una acción. De forma intuitiva, es posible sustituir esta función en el crítico, de forma que obtengamos ese valor de estado Q como una función $Q(s,\mu(s))$. Las redes neuronales son, en aprendizaje profundo, una implementación práctica de las funciones matemáticas.

Ahora, el valor de recompensa descontada se estima únicamente en función del estado y de los parámetros θ_μ de la red neuronal del actor y θ_Q de la red neuronal del crítico. El objetivo es optimizar estos pesos para obtener el mayor valor posible en la recompensa

obtenida. Lo que, de nuevo en términos matemáticos, significa que queremos obtener el gradiente de nuestra política, para actualizar en función de este valor.

En su artículo[4], David Silver prueba que un gradiente de política estocástico es squivalente al gradiente de una política determinista. Para calcular la política, únicamente es necesario calcular el gradiente de la función $Q(s, \mu(s))$. Aplicando la regla de la cadena, este gradiente se equipara a:

$$\nabla_s Q(s, \mu(s)) = \nabla_a Q(s, a) \nabla_{\theta_\mu} \mu(s) \tag{2.14}$$

Aunque tanto A2C como DDPG pertenecen a una misma familia de métodos, la forma en que se utiliza la red neuronal del crítico es diferente. En A2C el crítico constituye un valor *baseline* que se suma a la reompensa obtenida; es una pieza «opcional» del algoritmo que se emplea para mejorar la estabilidad. En DDPG, con una política determinista, los gradientes de la función $Q(s, \mu(s))$ se calculan a partir del crítico, que utiliza las acciones producidas por la red neuronal del actor. De esta forma, todo el sistema es diferenciable y susceptible a optimización por gradiente.

En resumen, el crítico se actualiza igual que en A2C, pero la red neuronal del actor se actualiza para optimizar los la salida del crítico.

Además, al ser un método *off-policy*, éste se mejora conforme el búfer de repetición de épocas aumenta.

Al ser un método determinístico, hay que introducir de forma arbitraria un componente de exploración. En un entorno estocástico, se hace añadiendo ruido aleatorio a las acciones en función de un parámetro ϵ.

2.8. Proximal Policy Optimization

El origen histórico del método PPO se encuentra en la librería creada por el OpenAI Team. Es una simplificación del algoritmo TRPO hecha por John Schulman y otros autores en 2017 [20].

De nuevo, parte del clásico **A2C** pero cambia la fórmula utilizada para estimar los gradientes de la política. En vez de utilizar el gradiente del logaritmo sobre la distribución de probabilidad $\nabla_\theta \log_{\pi_\theta}(a_t|s_t)$, utiliza un ratio entre la vieja y la nueva política:

$$\nabla_\theta U(\theta) \leftarrow \sum_{t=0}^{H} \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_o ld}(a_t|s_t)} A_t \tag{2.15}$$

Otra diferencia es que en **PPO** necesita contar con un búfer de experiencias completas antes de poder realizar ciclos de entrenamiento.

2.9. Soft Actor Critic

EL algoritmo SAC, propuesto por un grupo de investigadores de la Universidad de Berkeley en 2018[21], es una de las últimas y exitosas incorporaciones a los algoritmos de RL.

A pesar de su nombre, la idea detrás de esta técnica es más cercana al método **DDPG** que al método **A2C**.

La idea principal detrás de esta modificación reside en el concepto **Regularización de la entropía**. Se añade una recompensa supernumeraria (es una recompensa propia del algoritmo, y no del problema RL, en [21] le llaman «temperatura») en cada tiempo de ejecución, proporcional a la entropía de la política en ese instante determinado.[7].

Con esta definición en mente, una política óptima π^* en un entorno estocástico $\mathbb{E}_{\tau\sim\pi}$ $\mathbb{E}_\pi$ o es aquella que maximiza, no sólo la recompensa, sino también esta medida de la entropía:

$$\pi^* = \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t (R(s_t, a_t, s_{t+1}) + \alpha H(\pi(s_t))) \right] \tag{2.16}$$

De esta forma, se maximiza la exploración dando al agente un extra por acabar en estados donde la entropía es máximo, lo que fuerza una exploración del entorno.

Es común que, además, los algoritmos SAC incluyan una doble red neuronal para estimar los valores Q. Se escoge el menor de estos valores como aproximación de la función de la función Q.

La red neuronal del agente se actualiza igual que en el algoritmo DDPG. La red neuronal del crítico se optimiza teniendo en cuenta el gradiente no sólo del valor Q(a,s) del siguiente estado, sino también la entropía del siguiente estado ($\log \pi_\theta(s)$, donde el valor $\pi_\theta(s)$ se toma ponderando todo el espacio de acciones presente en el estado s.

2.10. Twin Delayed Deep Deterministic

El algoritmo TD3 es una evolución de DDPG presentada en 2018 [18], la diferencia más significativa consiste en retrasar la actualización de la red neuronal del actor respecto a la red neuronal del crítico, consiguiendo así, según los autores, una mayor inmunidad frente a perturbaciones

[7] El concepto de entropía es usado en termodinámica, mecánica estadística, teoría de la información... En todos los casos la entropía se concibe como una «medida del desorden» o la «peculiaridad de ciertas combinaciones», y se denota como $H(P) = \mathbb{E}_{x\sim P}[-\log(P)]$

Figura 2.5: El mal menor causará siempre menos daño al actualizar una política.

En este sentido, es una aproximación similar al algoritmo SARSA de entornos discretos; donde, para cada episodio, el agente recibe un primer estado s_t, escoge una acción a_t, recibe una recompensa r_{t+1} y evoluciona al estado s_{t+1}, utilizando de nuevo la misma política en t para seleccionar qué acción a_{t+1} ejecuta en $t+1$[8].

Una vez se tiene creada esta secuencia, se actualiza la tabla de valores $Q(s_t, a_t)$,la recompensa inmediate r_{t+1} y el valor descontado del siguiente par estado-acción $\gamma Q(s_{t+1}, a_{t+1})$

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[R_{t+1} + \gamma Q(s_{t+1}, a_{t+1} - Q(s_t, a_t) \right] \tag{2.17}$$

Además, el crítico descansa sobre dos redes neuronales distintas, tal como hace el método SAC, utilizándose el menor valor de ellos para estimar la función Q. Una posible infraestimación no es un problema, ya que estos valores no se propagan en el algoritmo, únicamente la influyen en la actualización del gradiente. Como expresa la imagen de la figura 2.5[9], una sobreestimación es más peligrosa que una infraestimación.

Por último, se fuerza la exploración, reduciendo la dependencia de una política determinista, añadiendo una pequeña cantidad de ruido a las acciones y promediando valores obtenidos en episodios consecutivos o *batches*.

[8] De ahí el acrónimo SARSA: State, Action, Reward, State, Action

[9] Referencia tomada de https://towardsdatascience.com/td3-learning-to-run-with-ai-40dfc512f93, originalmente publicada en https://funnytimes.com/wp-content/uploads/2011/10/131986994517768.png

Capítulo 3

Robótica como problema de *Reinforcement Learning*

Un brazo robótico es una cadena cinemática capaz de llevar a cabo tareas de forma flexible. El control de un brazo robótico con *Reinforcement Learning* puede abrir el campo a explotar esa flexibilidad sin necesidad de recurrir a complejos algoritmos de control numérico. El brazo robótico elegido es el *Franka Emika Panda*, ya programado en la librería **RLBench**.

Mecánicamente, el brazo consiste en una cadena cinemática abierta con 7 juntas de revolución (1 grado de libertad) y una pinza prensil compuesta por dos articulaciones prismáticas (1 grado de libertad) simétricas, que a efectos prácticos no se consideran en las ecuaciones cinemáticas. De hecho, el control del brazo robótico consiste en encontrar la trayectoria para que el centro de la pinza vaya desde un punto y rotación[1] $A \in \mathrm{R}^4$ hasta el punto y rotación $B \in \mathrm{R}^4$, por lo que el control de la pinza del robot queda fuera del ámbito del control de la trayectoria.

3.1. Cinemática directa

La cinemática directa relaciona la posición de una cadena robótica a partir de valores específicos llamados parámetros. En el caso de nuestro brazo robótico, estos parámetros son los ángulos de junta. A partir de este morfismo, pueden construirse las ecuaciones de la cinemática del robot.

Aunque siempre es posible obtener la cinemática directa de un brazo robótico, queda fuera del objetivo de este trabajo hacerlo, ya que el simulador robótico se encarga da proporcionar la posición de los elementos del robot junto a sus parámetros, en

[1] La posición de un punto o vector en el espacio y su rotación puede representarse a través de un cuaternión, que es un espacio de dimensión R^4

tiempo real. Para más información sobre cómo implementar una función auxiliar que determine estos parámetros a partir de los ángulos de junta, véase anexo F

3.2. Cinemática inversa

La cinemática inversa resulta útil para controlar un brazo robótico de cadena abierta [22], ya que permite relacionar los puntos finales de cada eslabón de la cadena en el espacio cartesiano con los ángulos de junta del robot[23]. De existir, es una aplicación del tipo: $F_{ik}(\theta_1, \theta_2, \theta_3, \theta_4, \theta_5, \theta_6, \theta_7) \rightarrow \{p_x, p_y, p_z, \alpha, \beta, \gamma\}$.

Dicha aplicación algebraica no dice nada acerca de cómo planear las trayectorias para ir de un punto A a un punto B, que son múltiples. Existen métodos algebraicos [24], geométricos [25] e iterativos [26] que dan respuesta al problema, cuyo coste computacional aumenta en una relación exponencial respecto al número de juntas del robot y sus grados de libertad. Están surgiendo algoritmos basados en redes neuronales [27], algoritmos genéticos[19] [2] o algoritmos de optimización por enjambre de partículas [28].

Para parametrizar la dinámica completa de un brazo robótico es necesario conocer los llamados *parámetros de base*, que son combinaciones lineales de los parámetros dinámicos de cada enlace o junta[29]. Excluida la fricción, cada enlace tiene asociados 10 parámetros dinámicos; a saber: la masa, la posición del centro de masas (Un vector con las 3 coordenadas en espacio euclídeo R^3), y un tensor de inercia compuesto de 6 elementos.

Como puede apreciarse, no es un problema trivial determinar la cinemática inversa del brazo robótico que puede tener múltiples soluciones, o incluso no tener solución por métodos algebraicos. De la misma forma que los parámetros D-H permiten obtener la cinemática directa, existen métodos iterativos que, en el mejor de los casos, convergen a una posible solución.

3.3. CoppeliaSim

La simulación robótica es en sí misma una alternativa que permite obtener la información de un ssitema robótico virual. No es necesario desarrollar métodos que permitan

[2] Un algoritmo genético es una búsqueda heurística inspirada en la teoría de la evolución natural de las especies de Charles Darwin. Refleja el proceso de selección natural en el sentido de seleccionar, entre un conjunto de soluciones finitas, aquellas que satisfacen un criterio y son capaces asimismo de generar nuevas soluciones basadas en una combinación lineal de las anteriores

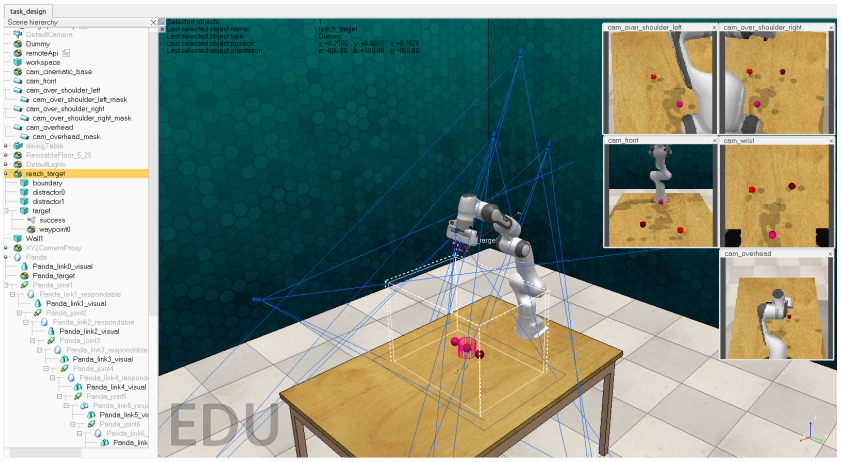


Figura 3.1: Franka Emika Panda en CoppeliaSim.

conocer la cinemática inversa, ya que los entornos de simulación ofrecen toda la información en tiempo real. Existen dos entornos de simulación robótica muy presentes en la literatura: **Bullet**[30] y **MuJoCo**[31]. Sin embargo, están basadas en entornos de simulación física, no específicamente robótica. Puede ser frustrante simular un brazo robótico completo utilizando estas plataformas y parametrizar la cinemática directa e inversa completamente.

CoppeliaSim, anteriormente conocido como **V-ReP** es una plataforma de experimentación puramente robótica. Con una interfaz más amigable y un conjunto de robots preseleccionados, presenta una interfaz más sencilla al usuario y todas las variables asociadas a un robot y su entorno accesibles en forma de esquema. Cabe mencionar que el brazo robótico **Franka Emika Panda** ya se encuentra modelado en la herramienta, por lo que es posible añadirlo al entorno virtual y empezar a trabajar con él directamente.

En la figura 3.1 podemos ver al robot **Panda**. Por defecto, la escena consiste en una mesa de madera donde el brazo robótico está engastado, rodeada por 3 luces direccionales. Además del rénder, es posible acceder a las observaciones RGB que simula la cámara de la pinza, o cámaras RGB estéreo situadas a ámbos lados de la pinza robótico. En cuanto a otras variables, junto a las variables utilizadas para construir y simular el robot, es posible acceder a los ángulos de junta, velocidades, momentos de inercia y la posición de la pinza, tanto para **Panda** como para otros brazos robóticos incluidos con el programa.

Estas son algunas de sus característiacs:

- Multiplataforma (Linux, Mac, Windows).
- Varias interfaces de comunicación con la herramienta (incluyendo LUA, plug-

gins en C++, y APIs para 6 lenguajes de programación).

- Soporte para herramientas de simulación de entornos físicos. Bullet, ODE, Newton y Vortex, siendo éstas intercambiables en el entorno.
- Cinemática directa e inversa del robot en tiempo de simulación.
- Planificación de trayectorias.
- Arquitectura distribuida de control basada en scripts Lua.

3.4. PyRep

Para facilitar su integración con otros sistemas y herramientras de programación, **CoppeliaSim** es accesible mediante una interfaz **Python** sobre la que interactúa **PyRep** de forma diferente a la *API* nativa. Es precisamente esta herramienta la que permitirá el resto del trabajo, permitiendo aplicar los algoritmos de *Reinforcement Learning* sobre un robot virtual, como si fuese un entorno real, a la vez que facilita la abstracción del sistema al ofrcer todas las variables en tiempo real.

Pyrep es una herramienta creada específicamente para la investigación en aprendizaje robótico. Necesita una versión compatible de **CoppeliaSim** o **V-REP** para funcionar y permite lanzar la simulación directamente desde **Python**. Así, permite ejecutar código de forma sincronizada con el robot virtual. Es un entorno especialmente interesante para la investigación en *Reinforcement Learning* o *Imitation Learning*. **Pyrep** ha sido creada y se mantiene por cortesía de Stephen James, investiagdor postdoctoral en el laboratorio de robótica de la Universidad de Berkeley[3].

Además de la integración, integra una herramienta de renderizado que acelera la simulación respecto al entorno original. En una prueba[32], los autores del artículo asegura obtener velocidad 10 veces superior a la interfaz original de **CoppeliaSim** para **Python**.

Pyrep se organiza en torno a un hilo de ejecución, del cual sólo se permite una instancia, un entorno o escena, y un robot asociado al entorno [33].

3.4.1. Instancia

Ejecución de la instancia de simulación.

```
from pyrep import PyRep
```

[3] https://github.com/stepjam/PyRep

```
pr = PyRep()
# Launch the application with a scene file in headless mode
pr.launch('scene.ttt', headless=True)
pr.start() # Start the simulation

# Do some stuff

pr.stop() # Stop the simulation
pr.shutdown() # Close the application
```

3.4.2. Entorno

Es posible cargar entornos predefinidos, o un entorno por defecto. En ambos casos, se puede modificar el entorno por programación.

```
from pyrep.objects.shape import Shape
from pyrep.const import PrimitiveShape

object = Shape.create(type=PrimitiveShape.CYLINDER,
                      color=[r,g,b], size=[w, h, d],
                      position=[x, y, z])
object.set_color([r, g, b])
object.set_position([x, y, z])
```

3.4.3. Uso del robot

El robot se controla como un objeto. Se establece el valor consigna de variables como la velociadd de los ángulos de junta, y se ejecuta el código de forma síncrona con instrucciones tipo **step()**.

```
from pyrep import PyRep
from pyrep.robots.arms.panda import Panda
from pyrep.robots.end_effectors.panda_gripper import PandaGripper

pr = PyRep()
# Launch the application with a scene file that contains a robot
pr.launch('scene_with_panda.ttt')
pr.start() # Start the simulation

arm = Panda() # Get the panda from the scene
gripper = PandaGripper() # Get the panda gripper from the scene

velocities = [.1, .2, .3, .4, .5, .6, .7]
arm.set_joint_target_velocities(velocities)
pr.step() # Step physics simulation

done = False
# Open the gripper halfway at a velocity of 0.04.
while not done:
    done = gripper.actuate(0.5, velocity=0.04)
    pr.step()

pr.stop() # Stop the simulation
pr.shutdown() # Close the application
```

Los robots de **CoppeliaSim** siguen un diseño modular; es decir, el brazo robótico y el extremo (una pinza u otro accesorio) se tratan como objetos distintos. Para utilizar en **Pyrep** un robot, éste tiene que haber sido definido previamente en la carpeta *robots/ttm*s. Una forma ordenada de trabajar consiste en construir los robots como un diccionario de brazo y pinza.

```
class MyRobot(object):
  def __init__(self, arm, gripper):
    self.arm = arm
    self.gripper = gripper

arm = Panda() # Get the panda from the scene
gripper = PandaGripper() # Get the panda gripper from the scene

# Create robot structure
my_robot_1 = MyRobot(arm, gripper)
# OR
my_robot_2 = {
  'arm': arm,
  'gripper': gripper
}
```

3.5. RLBench

Hasta ahora he hablado del robot (concretamente, el modelo *Franka Emika Panda* o *Panda*, por abreviar. Desde su naturaleza física a su equivalente virtual. Sobre el equivalente virtual se ha construido, gracias a la herramienta *PyRep*, un objeto en **Python** que representa al **robot** en su *entorno*. Ya tenemos una terna casi lista para plantear un problema de *Reinforcement earning* (no es directo, pero a través del entorno podríamos extraer las **observaciones**). Sin embargo, todavía falta la clave para plantear el problema, y es el modo de asignar una **recompensa** en función de una **tarea** asignada.

La última capa que falta añadir a estas herramientas, agrupadas cual muñeca rusa o *matrioska*[4], es la librería **RLBench**[5]. también escrita por Stephen James y diseñada para ejecutarse sobre **PyRep**[34].

Ha sido programado con la idea de tener una librería de código abierta que permita:

- Ofrecer un banco de desarrollo y entorno de aprendizaje mixto para *Reinforcement Learning* y métodos de control tradicionales.
- Permitr ensayos de larga duración, multitarea y con variaciones de cada tarea que puedan ser solventadas, cada una, de forma distinta.

[4] **RLBench** se ejecuta sobre **PyRep**, y a su vez esta librería necesita **CoppeliaSim**. Pido perdón si algún ruso se ofende por hacer analogía entre estas 3 herramientas y una muñeca rusa, cuyo número de elementos es al menos 5.

[5] https://github.com/stepjam/RLBench

– Permitir la creación de tareas personalizadas.

La libreriá **RLBench** se construye en torno a 3 conceptos: **tareas variaciones**, **episodios**. Cada tarea contiene una serie de variaciones, y por cada variación, la tarea puede ser resuelta en un conjunto de infinitos episodios. Un episodio τ perteneciente a una variación v de la tarea T se define como un conjunto de n observaciones o_n y acciones a_n

$$\tau = \{(o_1, a_1), ..., (o_n, a_n)\} \in v_i \tag{3.1}$$

El concepto de **variaciones** de una tarea no es un concepto propio de RL. Sin embargo, tal como indica el autor, se podría pensar que «agarrar un cubo rojo» y «agarrar un cilindro verde» son variaciones de una tarea genérica «agarrar un objeto X». Utilizando esta generalización, es posible reducir la complejidad en el número de tareas a plantear.

En su faceta como banco de trabajo para *Reinforcement Learning*, **RLBench** ofrece un acceso simplificado al entorno a través de observaciones. Siguiendo el modelo síncrono de **PyRep**, la simulación se desarrolla en pasos y es este paso de simulación el que toma un vector con las **acciones** y devuelve un vector de **observaciones**.

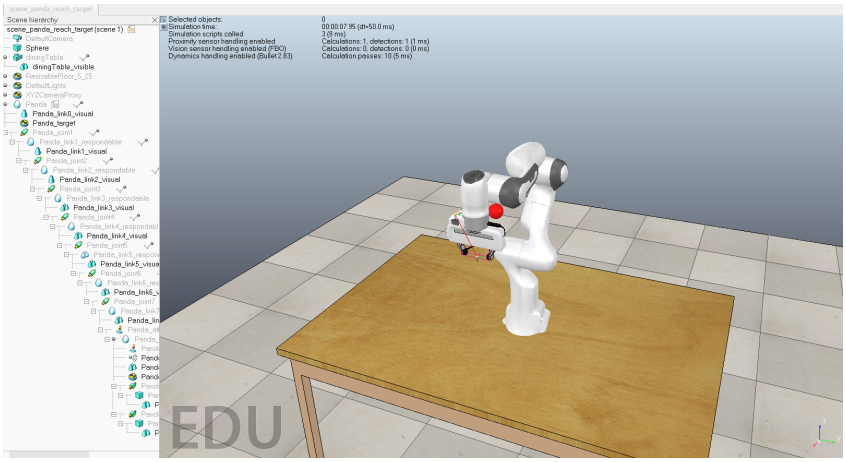


Figura 3.2: RLBench en CoppeliaSim.

La tarea *ReachTarget* hace aparecer una esfera roja en la posición objetivo que debe alcanzar la pinza, como se ve en la figura 3.2. Por defecto, la recompensa es siempre 0 mientras el robot no alcance la posición. Una vez alcanzada la posición, el entorno devuelve el valor 1 a la recompensa, y da por finalizada la tarea.

```
from rlbench.environment import Environment
from rlbench.action_modes import ArmActionMode, ActionMode
from rlbench.observation_config import ObservationConfig
from rlbench.tasks import ReachTarget
```

```
import numpy as np

class Agent(object):

    def __init__(self, action_size):
        self.action_size = action_size

    def act(self, obs):
        arm = np.random.normal(0.0, 0.1, size=(self.action_size -1,))
        gripper = [1.0] # Always open
        return np.concatenate([arm, gripper], axis=-1)


obs_config = ObservationConfig()
obs_config.set_all(True)

action_mode = ActionMode(ArmActionMode.ABS_JOINT_VELOCITY)
env = Environment(
    action_mode, obs_config=obs_config, headless=False)
env.launch()

task = env.get_task(ReachTarget)

agent = Agent(env.action_size)

training_steps = 120
episode_length = 40
obs = None
for i in range(training_steps):
    if i % episode_length == 0:
        print('Reset␣Episode')
        descriptions, obs = task.reset()
        print(descriptions)
    action = agent.act(obs)
    print(action)
    obs, reward, terminate = task.step(action)

print('Done')
env.shutdown()
```

Con un abundante conjunto tareas pre-programadas (la mayoría para el brazo robótico *Panda*), y un acceso directo a todas las variables necesarias para convertir la tarea en un problema de *Reinforcement Learning*, queda a criterio del programador cómo elegir el algoritmo que devuelva un vector de acción en función de las observaciones obtenidas en el entorno (en el ejemplo anterior, la acción era escogida siempre como un vector de números aleatorios). A ello se dedica el siguiente capítulo.

3.6. Herramienta *Gym*

Suelen explicarse los fundamentos de RL utilizando ejemplos discretos y limitados. Uno de los ejemplos más utilizados en la literatura, por su sencillez, es el entorno *Frozen-Lake*, incluido en la librería *Gym*. Un problema real, sin embargo, suele ser un problema de valores continuos; es decir, un valor del mundo real (ya sea una observación o una acción a tomar) se codifica en una variable que puede tomar cualquier valor numérico entre un máximo y un mínimo. Los ángulos de junta de un robot, la fuerza o velociad a aplicar, o la cantidad de color rojo que hay en un píxel codificado

por *RGB* son casos que se ajusta más a este último paradigma. Para poder trabajar con variables de esta naturaleza en un problema de RL, se extiende el concepto de espacio discreto a un espacio probabilístico, donde todas las posibles acciones, desde un valor $-\infty$ a un valor ∞ tienen asignada una probabilidad de acuerdo a una distribución de probabilidad determinada (por ejemplo, piensese en una distribución normal de probabilidad). La librería Gym está preparada para trabjar con este tipo de variables: cada una de las acciones u observaciones se representa por una variable de tipo *Box*.

3.6.1. Librerías RL

El primer paso es importar las librerías, tanto gym como la propia de RLBench:

[1]:
```
import gym
import rlbench.gym
```

3.6.2. Entorno

Luego, se debe crear el entorno especificando una tarea, que debe estar disponible. RLBench expande los entornos de Gym con entornos robóticos de **Coppelia**. Utilizaremos el entorno " 'reach_target-state-v0' ", que incluye al robot Panda. Ejecuta la escena definida en el fichero *reach_target.py*, variación 0, con una bola haciendo de objetivo y dos bolas haciendo de distractores.

```
from typing import List, Tuple
import numpy as np
from pyrep.objects.shape import Shape
from pyrep.objects.proximity_sensor import ProximitySensor
from rlbench.const import colors
from rlbench.backend.task import Task
from rlbench.backend.spawn_boundary import SpawnBoundary
from rlbench.backend.conditions import DetectedCondition


class ReachTarget(Task):

    def init_task(self) -> None:
        self.target = Shape('target')
        self.distractor0 = Shape('distractor0')
        self.distractor1 = Shape('distractor1')
        self.boundaries = Shape('boundary')
        success_sensor = ProximitySensor('success')
        self.register_success_conditions(
            [DetectedCondition(self.robot.arm.get_tip(), success_sensor)])

    def init_episode(self, index: int) -> List[str]:
        color_name, color_rgb = colors[index]
        self.target.set_color(color_rgb)
        color_choices = np.random.choice(
            list(range(index)) + list(range(index + 1, len(colors))),
            size=2, replace=False)
        for ob, i in zip([self.distractor0, self.distractor1], color_choices):
            name, rgb = colors[i]
            ob.set_color(rgb)
        b = SpawnBoundary([self.boundaries])
```

```
        for ob in [self.target, self.distractor0, self.distractor1]:
            b.sample(ob, min_distance=0.2,
                     min_rotation=(0, 0, 0), max_rotation=(0, 0, 0))

        return ['reach␣the␣%s␣target' % color_name,
                'touch␣the␣%s␣ball␣with␣the␣panda␣gripper' % color_name,
                'reach␣the␣%s␣sphere' %color_name]

    def variation_count(self) -> int:
        return len(colors)

    def base_rotation_bounds(self) -> Tuple[List[float], List[float]]:
        return [0.0, 0.0, 0.0], [0.0, 0.0, 0.0]

    def get_low_dim_state(self) -> np.ndarray:
        # One of the few tasks that have a custom low_dim_state function.
        return np.array(self.target.get_position())

    def is_static_workspace(self) -> bool:
        return True
```

El entorno se crea con el método *make()*, indicando el nombre con el que está registrado el la librería en el argumento. Otros argumentos: *render_mode*: Nos devuelve un rénder del robot o un *array RGB* con una imagen del entorno al hacer un rénder. *observation_mode*: Tiene dos formas de observación, como *array RGB* de las cámaras virtuales, o como estados del robot. En la sección **Espacio de observaciones** es explica en qué consiste cada modo y cómo deben invocarse.

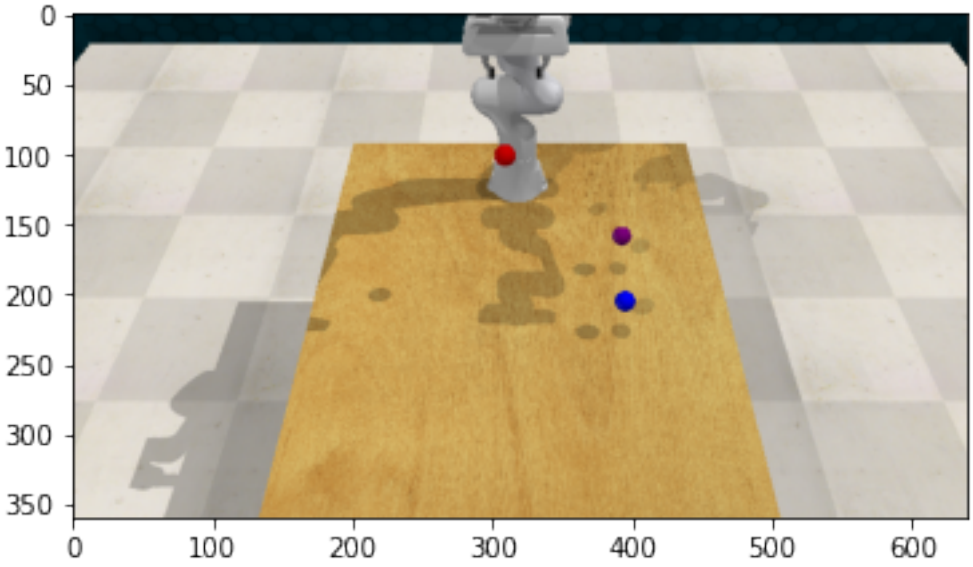
[2]:
```
env = gym.make('reach_target-state-v0', render_mode='rgb_array',␣
↪observation_mode='state')
```

El entorno establece un estado inicial, que sitúa el objetivo y los distractores en posiciones aletaorias, y al robot en una posición inicial por defecto. La función *reset()* devuelve una observación inicial del entorno.

[10]:
```
obs = env.reset()
```

Una vez creado e inicializado el entorno, podemos obtener un rénder del mismo. En el argumento de la función *render()*, debe indicarse como argumento el mismo *render_mode* indicado en el constructor del entorno. Al ser un *array RGB*, hay que importar el paquete **pyplot** de **matplotlib** para crear una figura donde visualizar la imagen:

[11]:
```
from matplotlib import pyplot as plt
render = env.render(mode='rgb_array')
fig = plt.figure()
im = plt.imshow(render)
plt.show()
```



3.6.3. Espacio de observaciones

Existen dos modos de observación: un espacio (*box*) RGB que representa la imagen obtenida por una cámara virtual, o un espacio continuo con un conjunto de variables que representan el estado del robot.

Los estados representan los ángulos de junta del brazo robótico y su dinámica. Además de la apertura, por cada ángulo de junta se recibe su velocidad y la fuerza aplicada. En la figura 3.3 podemos ver el brazo robótico Panda utilizado por defecto en este trabajo, con la posición de sus enlaces y juntas. La posición de la pinza de agarre se expresa como un todo o nada, 0 ó 1, en función de si está cerrada o abierta.

Para que la observación devuelva los estados, hay que pasar el parámetro *state* en el método **observation_mode**:

[3]:
```
env = gym.make('reach_target-state-v0', render_mode='rgb_array',␣
↪observation_mode='state')
```

De la pinza se recibe su posición en forma matricial, su grado de apertura y la fuerza que ejerce. También la posición de la cámara en la cintura del brazo, *'wrist camera'*, y la posición de el objetivo en el entorno, expresada en forma matricial como un conjunto de coordenadas y cuaterniones.

Una vez creado el entorno, podemos invocar **env.observation_space** para ver el formato de los datos que conforman las observaciones

```
[40]: print("Observation space: ", env.observation_space)
```

```
Observation space:  Box(-inf, inf, (40,), float32)
```

El estado que este entorno devuelve en cada time observación es una tupla de 40 valores continuos, tipo *BOX* que contienen su posición (*position*), su velocidad (*velocity*), el ángulo del poste (*angle*) y la velocidad angular del poste en la punta (*angular velocity*). En este caso, los valores que pueden tomar estos 4 parámetros que identifican el estado son continuos y su rango se muestra en la tabla 3.1

n	Variable de estado	Valor mínimo	Valor máximo
0	Pinza cerrada	0.0	1.0
1...7	Velocidad angular juntas	-inf	inf
8...14	Posición angular juntas	-inf	inf
15...21	Momento angular juntas	-inf	inf
22...24	Posición extremo pinza	-inf	inf
25..28	Rotación extremo pinza	-inf	inf
29,30	Actuadores pinza	-inf	inf
31...36	Sensores fuerza pinza	-inf	inf
37...39	Posición objetivo	-inf	inf

Tabla 3.1: El estado del brazo robótico y el objetivo se representan por variables continuas.

Pinza cerrada La primera variables es la posición de la pinza. Aunque sea un valor tipo *box*, sólo toma dos valores: 0,0 cuando está cerrada, y 1,0 en cualquier otra posición.

Dinámica brazo Las variables 1 a 7 representan la velocidad angular de cada junta en el momento de la observación. Llamando a la función *arm.get_joint_upper_velocity_limits()* , obtenemos [2,17499, 2,1749, 2,1749, 2,1749, 2,6099, 2,6099, 2,6099][6] , que es el valor máximo de las velocidades. La posición del brazo se expresa mediante el ángulo de cada junta del robot, y el momento angular es la fuerza que ejerce cada junta en un momento particular.

Posición extremo La posición del extremo del robot es un parámetro importante. Las variables 22, 23 y 24 representan la posición en coordenadas cartesianas, y las variables 25 a 26 expresan la rotación del último miembro (la pinza) mediante un cuaternión.

[6] Los decimales se han truncado.

Actuadores pinza La posición de los actuadores que abren la pinza se expresa en sendos valores iguales de signo puesto. La fuerza que ejerce la pinza se expresa por sendas matrices, correspondiente a cada actuador, y representa un vector de fuerza en coordenadas cartesianas.

Objetivo Las coordenadas del objetivo se representan en las últimas 3 variables de la observación como coordenadas cartesianas.

3.6.4. Espacio de acciones

El robot Panda acepta una tupla de 8 acciones, correspondientes a la velocidad a aplicar en cada ángulo de junta (7), y la orden de posición de la pinza, considerada abierta para un valor igual a 0, y cerrada para valores mayores a 0. Es equivalente a aplicar una tensión entre V_{min} y V_{max} a los motores del robot durante un periodo τ de tiempo.

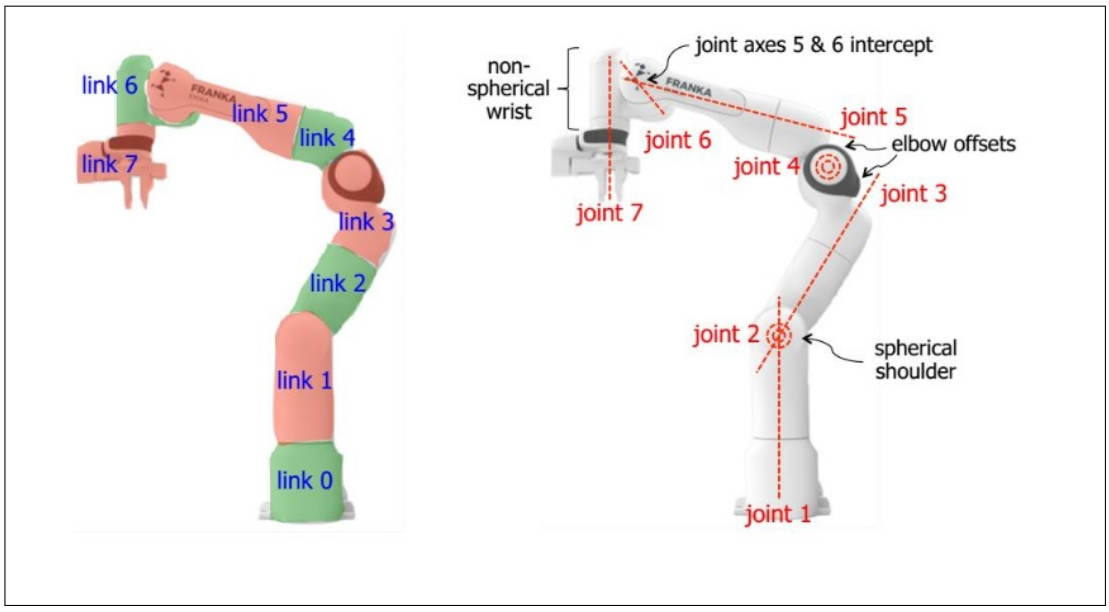


Figura 3.3: Ángulos de junta del robot Panda[35].

Para ver el formato de las acciones, invocamos el método **env.action_space()**, que nos devuelve una tupla de 8 valores continuos en formato **BOX**.

[41]:
```
print("Action space: ", env.action_space)
```

```
Action space:  Box(-1.0, 1.0, (8,), float32)
```

Además del modo **ABS_JOINT_VELOCITY**, es posible configurar en otro tipo de funcionamiento en la librería *RLBench*:

- ABS_JOINT_VELOCITY, velocidad absoluta de los ángulos de junta.

- DELTA_JOINT_VELOCITY, velocidad incremental de los ángulos de junta.
- ABS_JOINT_POSITION, posición absoluta de los ángulos de junta, en rad.
- DELTA_JOINT_POSITION, posición incremental de los ángulos de junta, en rads.
- ABS_JOINT_TORQUE, fuerza/torsión absoluta a aplicar en los ángulos de junta.
- DELTA_JOINT_TORQUE, fuerza/torsión incremental a aplicar en los ángulos de junta.

Aunque estos sean los modos de funcionamiento del robot Panda, la librería define otros formatos de instrucción a la hora de definir vectores de acciones:

- ABS_EE_POSE_WORLD_FRAME, posición de la junta expresada en coordenadas cartesianas y cuateriones.
- DELTA_EE_POSE_WORLD_FRAME, posición incremental de la junta expresada en coordenadas cartesianas y cuateriones.
- ABS_EE_POSE_PLAN_WORLD_FRAME, ídem al anterior pero con el robot planificando la trayectoria.
- ABS_EE_POSE_PLAN_WORLD_FRAME_WITH_COLLISION_CHECK, ídem al anterior, pero evitando colisiones en la planificación de la trayectoria.
- DELTA_EE_POSE_PLAN_WORLD_FRAME, ídem a la opción DELTA_EE_POSE_WORLD_FRAME, pero con el robot planificando la trayectoria.
- EE_POSE_EE_FRAME, igual que ABS_EE_POSE_WORLD_FRAME, pero considerando las coordenadas relativas a la posición anterior. Por definición, es un incremental.
- EE_POSE_PLAN_EE_FRAME, como EE_POSE_EE_FRAME, pero con el robot planificando la trayectoria.

La tabla 3.2 el espacio de acciones que admite el brazo robótico Panda. Todos los valores, incluso la posición de la pinza del robot, son valores discretos tipo *Box*. En la figura 3.3 podemos ver una representación de los ángulos de junta del robot.

3.6.5. Agente

En el ejemplo de la librería RLBench, existe un agente bastante inútil que selecciona, de entre el espacio de acciones posibles, una acción al azar y la ejecuta, sin tener en

Acción	Descripción	Tipo	Mín	Max
0	Junta 1 (rotación base)	*Box*	−1,0	+1,0
1	Junta 2 (inclinación)	*Box*	−1,0	+1,0
2	Junta 3 (rotación antes del codo)	*Box*	−1,0	+1,0
3	Junta 4 (flexión codo)	*Box*	−1,0	+1,0
4	Junta 5 (rotación después del codo)	*Box*	−1,0	+1,0
5	Junta 6 (inclinación cabezal)	*Box*	−1,0	+1,0
6	Junta 7 (rotación cabezal)	*Box*	−1,0	+1,0
7	Pinza (apertura todo/nada)	*Box*	0,0	+1,0

Tabla 3.2: El espacio de acciones en CoppeliaSim es un espacio de variables continuas.

cuenta las observaciones.

```
import gym
import rlbench.gym

env = gym.make('reach_target-state-v0', render_mode='human')

training_steps = 120
episode_length = 40
for i in range(training_steps):
    if i % episode_length == 0:
        print('Reset␣Episode')
        obs = env.reset()
    obs, reward, terminate, _ = env.step(env.action_space.sample())
    env.render() # Note: rendering increases step time.

print('Done')
env.close()
```

El entorno proporciona el método *action_space.sample()* que devuelve una acción aleatoria dentro del espacio de acciones posibles: Es posible resolver el episodio de esta forma, totalmente aleatoria si se deja al agente ejecutar la simulación en bucle[7].

[12]:
```
action = env.action_space.sample()
```

En un entorno Gym, el método step() recibe la acción como argumento y devuelve una tupla con cuatro elementos: una nueva observación del entorno, la recompensa por esa acción, un valor booleano que indica si el escenario ha finalizado (por éxito… o por fracaso, eso debemos deducirlo a partir del valor de la observación) y una cuarta variable con información auxiliar, que depende de cada entorno:

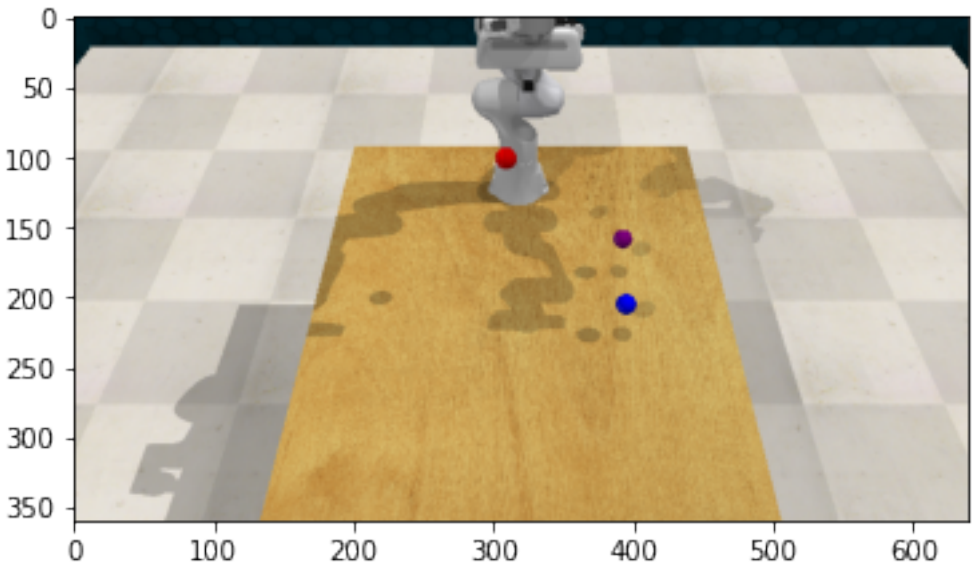
[13]:
```
obs, reward, done, info = env.step(action)
```

Ejecutando de nuevo un rénder, vemos que el entorno ha variado (levemente) respecto

[7] Un mono podría escribir el Quijote si tuviese una cantidad de tiempo infinita y una política estocástica. En nuestro caso, suelen ser necesarios entre 10000 y 20000 pasos para solucionar el entorno de esta forma tan poco eficiente.

a la posición inicial:

[14]:
```
render = env.render(mode='rgb_array')
fig = plt.figure()
im = plt.imshow(render)
plt.show()
```



Podemos definir así un agente y ejecutar el código, o bien hasta que la tarea se cumpla, o bien hasta que haya ejecutado 100 steps y, por lo tanto, de la tarea por finalizada (en un rotundo fracaso):

[39]:
```
import imageio
import numpy as np

class Agent:
    def __init__(self):
        print("Clumsy agent on duty")
    def select_action(self, environment):
        return env.action_space.sample()
agent = Agent()
is_done = False
MAX_STEPS = 100
step = 0
obs = env.reset()
frames = []
frame = env.render(mode='rgb_array')
img_uint8 = (frame*255).astype(np.uint8)
while not is_done and step<MAX_STEPS:
```

```
    frames.append(img_uint8)
    action = agent.select_action(env)
    obs, reward, is_done, info = env.step(action)
    frame = env.render(mode='rgb_array')
    img_uint8 = (frame*255).astype(np.uint8)
    step+=1
frames.append(img_uint8)
imageio.mimsave('Clumsy_Agent.gif', [np.array(frame) for i, frame in␣
 ↪enumerate(frames) if i%2 == 0], fps=29)
if(is_done):
    print(f"Environment solved in {step} steps.")
else:
    print(f"Environment not solved in {MAX_STEPS} steps.")
```

```
Clumsy agent on duty
Environment not solved in 100 steps.
```

La línea *img_uint8 = (frame*255).astype(np.uint8)* transforma la observación obtenida mediante rénder en un *array* de enteros tipo unsigned int de 8 bits. La instrucción *frames.append(img_uint8)* crea un *array* de imágenes que, posteriormente, con la instrucción serán convertidos en una animación de tipo .gif. Únicamente útil para visualizar posteriormente el comportamiento del agente.

3.7. Stable Baselines 3

El entorno **Gym** proporciona los ingredientes para modelizar un problema de RL pero, como hemos visto, no aporta una solución al mismo. Hasta ahora hemos visto que el agente «torpe» recibe un estado $\vec{s}$, y selecciona una acción $\vec{a}$ siguiendo una **política**» que no tiene en cuenta ninguna **experiencia** o **episodio**.

La implementación práctica de cualquier algoritmo de esta sección es matemáticamente más compleja que la de un agente **REINFORCE** o **DQN**. Al aumentar su complejidad, los algoritmos de Rl han propiciado la aparición de *frameworks* que faciliten su implementación. Si **Gym** Facilitaba el planteamiento de un problema de RL, estos *frameworks* facilitan la ejecución de algoritmos más complejos destinados a solucionar el problema.

Estos *frameworks* suelen ser compatibles con **Tensorboard**, **PyTorch** o ambos. Estas librerías ofrecen una colección de herramientaso para desarrollar y entrenar modelos de redes neuronales mediante **Python**, propiciando así su uso en problemas de inteligencia artificial.

Las **Stable Baselines** son un conjunto de implementaciones mejoradas de algoritmos de aprendizaje por refuerzo, todas ellas basadas en las **Baselines** de **OpenAI**. Tanto éxito han tenido las Stable Baselines, que sus autores han ido mejorando ese código hasta lanzado una nueva versión: las Stable Baselines3. Presentan algunos cambios con respecto a sus predecesoras[8]. La más significativa es que las Stable Baselines utilizan *Tensorflow* y las Stable Baseline3 se han pasado a *PyTorch*, lo que se traduce en un código más fácil de leer y de depurar, a costa de una ligera pérdida de velocidad[9].

Name	Box	Discrete	MultiDiscrete	MultiBinary	Multi Processing
A2C	✔	✔	✔	✔	✔
DDPG	✔	✘	✘	✘	✘
DQN	✘	✔	✘	✘	✘
HER	✔	✔	✘	✘	✘
PPO	✔	✔	✔	✔	✔
SAC	✔	✘	✘	✘	✘
TD3	✔	✘	✘	✘	✘
QR-DQN [1]	✘	✔	✘	✘	✘
TQC [1]	✔	✘	✘	✘	✘
Maskable PPO [1]	✘	✔	✔	✔	✔

Figura 3.4: Algoritmos Stable Baselines 3 y su aplicación a problemas de RL.

La tabla fig:tablestablebaselines3algorithms[10] muestra el listado completo de algoritmos disponibles en *Stable Baselines 3*. Estos son los algoritmos de **Stable Baselines3** que podemos utilizar en un entorno robótico de observaciones y acciones continuas (tipo **Box**)[11]:

- A2C
- DDPG
- PPC
- SAC
- TD3

[8] https://github.com/DLR-RM/stable-baselines3
[9] https://www.discoder.tech/stable-baselines3-las-hermanas-mayores-de-stable-baselines/
[10] https://stable-baselines3.readthedocs.io
[11] A partir de la versión 3 v1.1.0, el algoritmo **HER** es una clase de búfer de repetición

Con la librería **Stable baselines 3** podemos construir fácilmente un agente donde las acciones se escojan aleatoriamente.

[1]:
```
import gym
import rlbench.gym
```

[2]:
```
render_mode = None
observation_mode = 'state'
policy_mode = 'MlpPolicy'
env_name = 'reach_target-state-v0'
```

[3]:
```
env = gym.make(env_name, render_mode=render_mode,␣
 ↪observation_mode=observation_mode)
```

Una vez está correctamente instalada, tal como se indica en el anexo A, importamos el algoritmo desde la librería **Stable_Baselines3**:

[4]:
```
from stable_baselines3 import A2C
```

La política de estos algoritmos está construida con redes neuronales. Tenemos que importar la política independientemente escogida. Es importante importarla correctamente, pues aunque sean compatibles, no será lo mismo importar el modelo **MlpPolicy** desde el módulo **a2c** que desde el módulo **sac**:

[5]:
```
from stable_baselines3.a2c.policies import MlpPolicy, CnnPolicy
```

Ahora la creación del modelo de interacción agente-entorno por A2C se reduce a una simple línea, donde se pasa la política y el entorno como argumento:

[6]:
```
policy_mode = 'MlpPolicy'
model = A2C(policy_mode, env, n_steps = 5, verbose=1)
```

```
Using cuda device
Wrapping the env with a `Monitor` wrapper
Wrapping the env in a DummyVecEnv.
```

El entrenamiento del modelo se hace con el método *learn()*, donde el parámetro obligatorio es el número de *steps* que durará el entrenamiento. Además del entrenamiento, se ejecuta una evaluación del entorno cada *eval_freq steps*, y sus resultados se almacenan en *eval_log_path*. Entre sucesivas evaluaciones, se guarda en disco el modelo que haya obtenido mayor recompensa. El *mini-batch* de *n_val_episodes* es el que se ejecuta antes de actualizar la política, y con *tb_log_name* podemos indicar la carpeta donde guardar los *logs* que mostrará **tensorboard**.

[7]:
```
LEARNING_STEPS = 1000
model.learn(LEARNING_STEPS, eval_freq=10000, n_eval_episodes=5,␣
 ↪eval_env = env, eval_log_path="./eval_logs_",␣
 ↪tb_log_name="test_A2C_jupyter", reset_num_timesteps=True)
```

```
Wrapping the env with a `Monitor` wrapper
Wrapping the env in a DummyVecEnv.
-------------------------------------
| time/                 |           |
|    fps                | 20        |
|    iterations         | 100       |
|    time_elapsed       | 24        |
|    total_timesteps    | 500       |
| train/                |           |
|    entropy_loss       | -11.4     |
|    explained_variance | -4.07e+05 |
|    learning_rate      | 0.0007    |
|    n_updates          | 99        |
|    policy_loss        | 4.71      |
|    std                | 1         |
|    value_loss         | 0.331     |
-------------------------------------
-------------------------------------
| time/                 |          |
|    fps                | 20       |
|    iterations         | 200      |
|    time_elapsed       | 48       |
|    total_timesteps    | 1000     |
| train/                |          |
|    entropy_loss       | -11.4    |
|    explained_variance | -529     |
|    learning_rate      | 0.0007   |
|    n_updates          | 199      |
|    policy_loss        | 5.88     |
|    std                | 1        |
|    value_loss         | 0.256    |
-------------------------------------
```

[7]: <stable_baselines3.a2c.a2c.A2C at 0x7f0d42dbb610>

Podemos evaluar el entorno entrenado un determinado número de veces (episodios completos) con la función *evaluate_policy*, donde la política del modelo es un parámetro. En una evaluación determinística de la política, las acciones se toman en base a la probabilidad estocástica de cada una de ellas; en una evaluación determinística, se ejecuta en cada momento la acción más probable. La función devuelve la recompensa media y la desviación de las recompensas obtenidas en ese número de ejecuciones:

[ ]:
```
from stable_baselines3.common.evaluation import evaluate_policy
num_evals = 1
mean_reward, std_reward = evaluate_policy(model.policy, env,␣
  ↪n_eval_episodes=num_evals, deterministic=True)
print(f"mean_reward={mean_reward:.2f} +/- {std_reward}")
```

```
/home/asimon/miniconda3/envs/rl4rob/lib/python3.8/site-
packages/stable_baselines3/common/evaluation.py:65: UserWarning:␣
  ↪Evaluation
environment is not wrapped with a ``Monitor`` wrapper. This may result in
reporting modified episode lengths and rewards, if other wrappers happen␣
  ↪to
modify these. Consider wrapping environment first with ``Monitor``␣
  ↪wrapper.
  warnings.warn(
```

Podemos guardar el modelo entrenado en disco, para recuperarlo más adelante y seguir entrenando, o utilizarlo en un entorno si ya está correctamente entrenado.

[ ]:
```
NAME_MODEL = 'test'
model.save(NAME_MODEL)
```

La política puede guardarse independientemente del modelo. Podría aprovecharse así para un entorno similar, o como un valor de inicialización en un nuevo escenario:

[ ]:
```
policy = model.policy
policy.save(name_model + '_policy')
```

No es el caso del algoritmo **A2C**; pero, en algoritmos *off-policy*, es posible almacenar en disco el búfer de experiencias previas.

[ ]:
```
# now save the replay buffer too No con A2C
# model.save_replay_buffer(name_model + '_replay_buffer')
```

Conocido tanto el nombre del fichero como el entorno de evaluación, es posible cargar el modelo guardado en disco con una simple línea de código:

```
[ ]: model = A2C.load(name_model, env)
```

Igualmente, podemos cargar una política guardad en disco a un modelo existente. Sin embargo, hay que conocer si la política es MLP o CNN, pues el método depende del tipo de política:

```
[ ]: policy = MlpPolicy.load(name_model + '_policy')  # ojo. si es CNN es␣
      ↪distinto
     print("loaded mlp policy from " + name_model + '_policy\n')
```

Capítulo 4

Implementación y resolución de tareas

La definición de la tarea a realizar y la construcción en consecuencia de una función de recompensa lo es todo en un problema de aprendizaje por refuerzo. Hablamos de tarea para referirnos al conjunto de episodios generados bajo una misma premisa; en nuestro caso, alcanzar un punto del espacio marcado por una esfera roja cuya posición se expresa en coordenadas cartesianas y cuya rotación se expresa por medio de cuaterniones. Puesto que el objetivo puede aparecer en cualquier punto del espacio accesible por el robot, a cada una de estas situaciones se les conoce como variaciones de la tarea principal.

4.1. Construcción de la recompensa

El algoritmo A2C no consigue resolver la tarea. El problema *reach_target_v0* plantea un escenario en que la recompensa es 1 si y sólo si el brazo robótico alcanza la posición del objetivo.

Además de la ausencia de una recompensa inmediata que guíe la política del robot en cada paso de simulación, el algoritmo no mejora en ningún momento al solventarse el episodio.

La complejidad del espacio de estados-acciones que puede tomar un brazo robótico de 7 juntas hacen que solventar los episodios de esta manera tarde entre 8000 y 30000 pasos realizando movimientos aleatorios. Serían necesarios muchos más pasos de simulación para entrenar correctamente una política siguiendo esta estrategia.

Además, el hecho de que el objetivo aparezca cada vez en unas coordenadas diferentes aumenta la complejidad del problema, añadiendo más grados de libertad al problema.

4.1.1. Callbacks

Puede utilizarse la estrategia de emplear *callbacks* personalizados que modifiquen la recompensa obtenida en cada iteración, o varíen el comportamiento del entorno de forma que limiten el número de episodios. Pueden construirse expandiendo la clase **BaseCallback** de **Stable Baselines 3**

```
from stable_baselines3.common.callbacks import BaseCallback
class CustomCallback(BaseCallback):
    """
    A custom callback that derives from ``BaseCallback``.

    :param verbose: (int) Verbosity level 0: not output 1: info 2: debug
    """
    self.step = 0
    def __init__(self, verbose=0):
        super(CustomCallback, self).__init__(verbose)
        # Those variables will be accessible in the callback
        # (they are defined in the base class)
        # The RL model
        # self.model = None # type: BaseRLModel
        # An alias for self.model.get_env(), the environment used for training
        # self.training_env = None # type: Union[gym.Env, VecEnv, None]
        # Number of time the callback was called
        # self.n_calls = 0 # type: int
        # self.num_timesteps = 0 # type: int
        # local and global variables
        # self.locals = None # type: Dict[str, Any]
        # self.globals = None # type: Dict[str, Any]
        # The logger object, used to report things in the terminal
        # self.logger = None # type: logger.Logger
        # # Sometimes, for event callback, it is useful
        # # to have access to the parent object
        # self.parent = None # type: Optional[BaseCallback]

    def _on_training_start(self) -> None:
        """
        This method is called before the first rollout starts.
        """
        print("Training start")
        # pass

    def _on_rollout_start(self) -> None:
        """
        A rollout is the collection of environment interaction
        using the current policy.
        This event is triggered before collecting new samples.
        """
        print("Rollout start")
        pass

    def _on_step(self) -> bool:
        """
        This method will be called by the model after each call to `env.step()`.

        For child callback (of an `EventCallback`), this will be called
        when the event is triggered.

        :return: (bool) If the callback returns False, training is aborted early.
        """
        return True

    def _on_rollout_end(self) -> None:
        """
        This event is triggered before updating the policy.
        """
        pass
```

```
    def _on_training_end(self) -> None:
        """
        This event is triggered before exiting the `learn()` method.
        """
        pass
```

Se crea el objeto y se pasa como argumento al método *model.learn()*. Habría que modificar en el método *_on_step(self)* la forma de limitar las iteraciones o modificar la recompensa.

```
myCallback = CustomCallback(verbose = 1)

model = A2C('MlpPolicy', 'reach_target-state-v0')
model.learn(2000, callback=myCallback)
```

4.1.2. Wrappers

Podemos utilizar envolventes o *wrappers* en problemas de RL para modificar un entorno existente. Podemos modificar el problema para que devuelva, en cada iteración, una recompensa inmediata, limitar el número de iteraciones por episodio, o ambas.

Esta es la solución elegida. Se construye una envolvente del entorno *env* que expande el código de sus métodos *init*, *step* y *reset*.

Para limitar el número de episodios, se añade una variable a la clase *env* llamda *max_steps_episode*. Con un valor por defecto de 500 steps, no tiene ningún efecto (no hay limitación) si se establece un número negativo en el constructor del entorno para esta variable.

Además de la posible limitación de episodios, se postulan 4 modelos de recompensa para facilitar la resolución del entorno.

Modo0 Sigue siendo una recompensa todo/nada al final, con la salvedad que devuelve 0 en cada paso, 1 si resuelve el episodio, y −1 en el step final si alcanza el límite de iteraciones por episodio sin resolvere el entorno.

Modo1 Devuelve una recompensa inmediata de −1 con cada episodio. Si el episodio no se ha resuelto, el valor total de recompensa acumulada será igual al negativo de *max_steps_episode*.

Modo2 Como el modo anterior, devuelve una recompensa inmediata compuesta por la suma de −1 y el valor negativo de la distancia al objetivo ,de forma que cuanto más

lejos esté, más negativa será la recompensa.

Modo3 Como en el modo anterior, pero esta vez el agente obtiene una inmensa recompensa, el valor positivo de max_steps_episode, si resuelve el problema antes de que se agote el límite de iteraciones. Es una forma de compensar la enorme tarea de exploración que necesita el robot antes de resolver un episodio por primera vez.

Podría haber sido interesante proponer un cuarto modo, donde la recompensa sea el valor inverso de la distancia. Al aproximarse al objetivo, las recompensas serían cada vez mayores. Es una estrategia interesante seguida en [36] para entrenar meta-tareas que desarrollen políticas de inicio para otras más complejas.

```
class myEnv(gym.Wrapper):
    _max_steps_episode: int
    _step_counter: int
    _distance: float
    _delta_distance: float

    def __init__(self, env=None, max_steps_episode=500, mode="mode0"):
        super(myEnv, self).__init__(env)
        self._step_counter = 0
        self._max_steps_episode = max_steps_episode
        self._mode = mode
        self._distance = -1
        print("Wrapping␣the␣environment␣in␣a␣homemade␣step␣to␣do␣cosas␣model␣" + mode + "\n")

    def reset(self):
        obs = self.env.reset()
        try:
            self._distance = distance_cal(obs)
            print("Reset␣environment,␣initial␣distance␣is␣{0}\n".format(self._distance))
        except:
            print("Something␣went␣wrong␣on␣reset")
        return obs

    def step(self, action):
        self._step_counter += 1
        obs, reward, done, info = self.env.step(action)
        try:
            if done:
                print("Episodio␣resuelto␣en␣{0}␣steps\n".format(self._step_counter))
                self._step_counter = 0
            # mode0 returns -1 if goal is not achieved in max_steps_episode, and 1 if is achieved before.
            if self._mode == "mode0":
                if self._max_steps_episode > 0:
                    if self._step_counter >= self._max_steps_episode:
                        done = True
                        reward = -1
                        self._step_counter = 0
                        print("Reset␣environment,␣took␣too␣much\n")
            # mode1 might or might not have an episode step limit, but returns -1 in each step while goal ↙
                  is not
            # accomplished
            if self._mode == "mode1":
                if not done:
                    reward = -1
                if self._max_steps_episode > 0:
                    if self._step_counter >= self._max_steps_episode:
                        done = True
                        self._step_counter = 0
                        print("Reset␣environment,␣took␣too␣much\n")
```

```
            # mode2 might or might not have an episode step limit, but a reward proportional to the delta ↙
                 distance to
            # target. In every step, this reward is decreased. When finished, reward is 1. if doesn't ↙
                 reach the
            # target in the max_step, returns a negative reward tbd.
            if self._mode == "mode2":
                #new_distance = distance_cal(obs)
                #self._delta_distance = new_distance -self._distance
                #self._distance = new_distance
                if not done:
                    # reward = np.tanh(1/(distance+1e-5))
                    #reward = np.exp(-1 * distance_cal(obs))
                    reward = -distance_cal(obs) -1 # Penalización por cada step de más que tarde en ↙
                         resolver el episodio
                if self._max_steps_episode > 0:
                    if self._step_counter >= self._max_steps_episode:
                        done = True
                        self._step_counter = 0
                        print("A␣cascarla,␣ha␣tardado␣mucho\n")
            if self._mode == "mode3": # Devuelve un reward arbitrario e inmenso = 500 si resuelve el ↙
                 episodio
                if not done:
                    reward = -distance_cal(obs) -1 # Penalización por cada step de más que tarde en ↙
                         resolver el episodio
                else:
                    reward = 500
                if self._max_steps_episode > 0:
                    if self._step_counter >= self._max_steps_episode:
                        done = True
                        self._step_counter = 0
                        print("A␣cascarla,␣ha␣tardado␣mucho\n")
        except:
            print("Error␣on␣step")
        return obs, reward, done, info
```

Para los modos 1 y 2 se necesita calcular en cada iteración la distancia entre la pinza robótica y el objetivo. Tanto las coordenadas del objetivo como de la punta del brazo robótico están en las observaciones. Se programa una función auxiliar que simplifique el cálculo.

```
def distance_cal(obs):
    '''calculate distance between end effector and target'''
    ee_pose = np.array(obs[22:25])
    target_pose = np.array(obs[-3:])

    distance = np.sqrt((target_pose[0]-ee_pose[0])**2 +
                    (target_pose[1]-ee_pose[1])**2 + (target_pose[2]-ee_pose[2])**2)

    # reward = np.tanh(1/(distance+1e-5))
    # reward = np.exp(-1*distance)

    return distance
```

Todas las distancias del entorno son inferiores a 1. No aporta (e incluso se ha probad) utilizar la distancia cuadrática en vez de la distancia al objeto.

Este envolvente se aplica al entorno antes de crear el modelo. Los métodos *learn* y *evaluate* aplican también sus propias envolventes al entorno, por eso es importante aplicarlo antes que los algoritmos de RL, como las capas de una cebolla.

```
    env = gym.make(env_name, render_mode=render_mode, observation_mode=observation_mode)
    # Wrapping custom environment to limit episode steps. Wrapper is in external
    # library. Also returns -1 when max limit has been reached.
    env = myEnv(env, mode=reward_model)
```

4.2. Problema de prueba

El algoritmo A2C es incapaz de solventar el escenario modificando únicamente la recompensa. Antes de probar otros algortimos, decido simpliificar la tarea, haciendo que el objetivo aparezca siempre en la misma posición cuando se resetea el entorno.

Para ello, se modifica en el *wrapper* el método *reset()* de forma que fuerce la posición del target a una posición fija:

```
class myEnv(gym.Wrapper):
    ...
    def reset(self):
        obs = self.env.reset()
        #
        self.task._task.target.set_position([0.35000000, 0.22000000, 0.97000000])
        obs, _, _, _ = self.env.step(np.array([0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0]))
        ...
```

4.2.1. A2C

Tras una ejecución de 7 millones de pasos, los resultados de la gráfica 4.1 no son alentadores. En la primera se muestra la recompensa media obtenida por cada episodio, siendo valores próximos a 0 aquellos que resuelven el episodio. En la segunda gráfica se muestra la longitud media por episodio; cuanto menos tarde, mejor habrá sido el resultado y en consecuencia la recompensa será mayor (menos negativa). El límite de pasos por episodio está en 500, pero la recompensa en cada paso es función de la distancia, por lo que no hay un límite tan claro como en la longitud. Este es el más sencillo de los algoritmos, se esperaba una enorme exploración, pero una convergencia visible hacia la solución. Al fin y al cabo, bastaría con que el robot encuentre un movimiento repetitivo una vez encontrada la solución, pues el objetivo aparece siempre en las mismas coordenadas, y el robot inicia la exploración siempre desde la misma posición. el problema puede residir tanto en la naturaleza del algoritmo como en el tamaño de la red neuronal, incapaz de modelar de forma eficiente el comportamiento del brazo robótico y la función valor de cada estado.

En vez de continuar por este camino se recurre a otros algoritmos utilizados en la literatura para resolver problemas relacionados con la robótica.

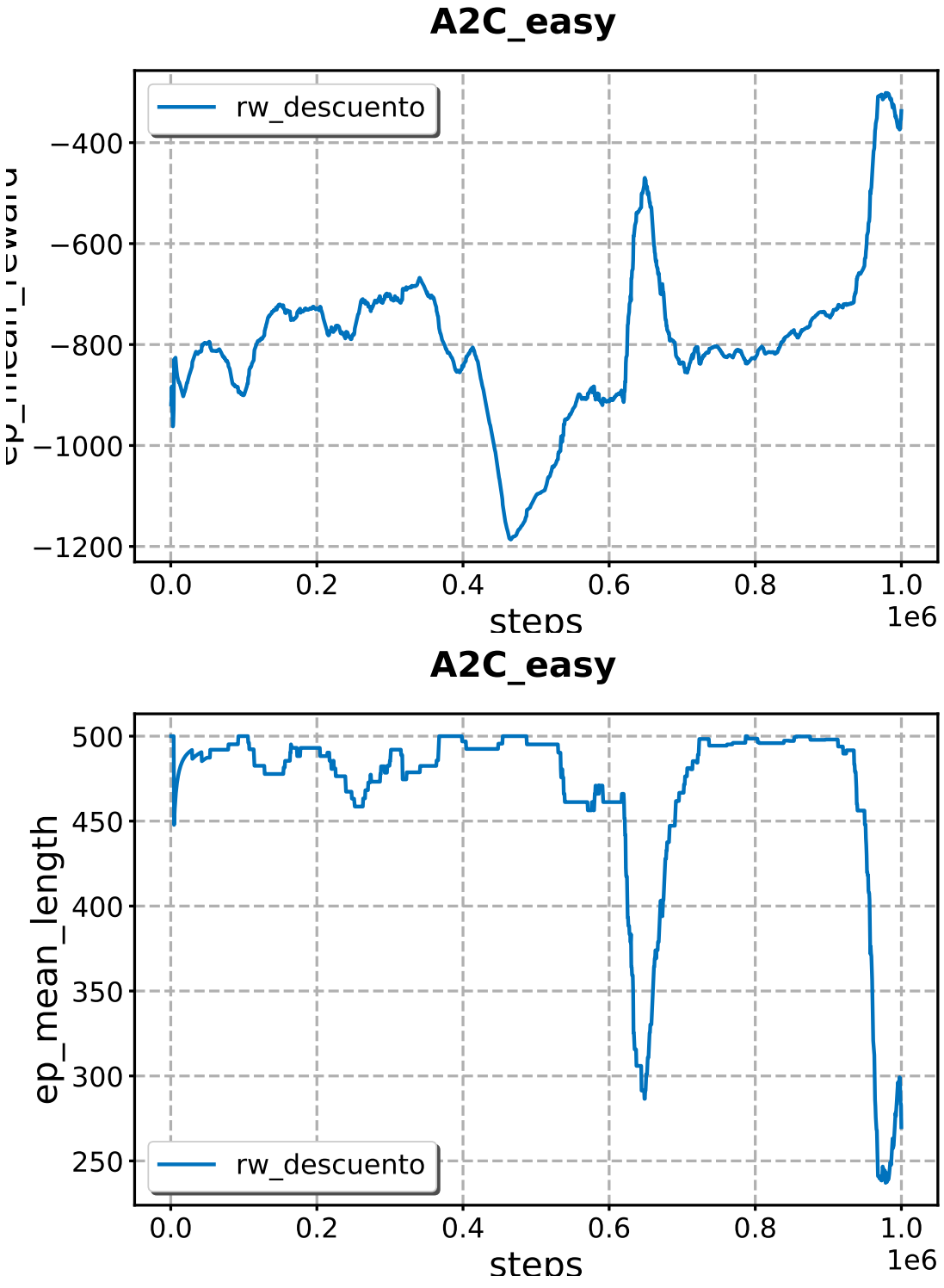


Figura 4.1: Algoritmo A2C en modo fácil usando el modelo de recompensa sin premio final.

4.2.2. DDPG

La figura 4.2 muestra la evolución de estos dos métodos. Con una recompensa en función de la distancia, el algoritmo es incapaz de encontrar una solución. Y con una recompensa al final de cada episodio resuelto satisfactoriamente, el algoritmo necesita alrededor de 300000 pasos hasta encontrar una solución.

El algoritmo DDPG parece capaz de solventar el episodio para el método 3, con una gran recompensa al final; sin embargo, no es capaz de hacer lo mismo con el método 2, sin ese gran final y guiado únicamente por la distancia al objetivo. Es uno de los algoritmos más sencillos, así que hay margen para encontrar una solución mejor.

4.2.3. PPO

En el entorno de los 100000 pasos, el algoritmo PPO tiene una tendencia a encontrar una solución, tanto con el método 2 como con el método 3, como se muestra en la figura 4.3. Sin embargo, esta tendencia no es suficiente y se corta el algoritmo. El algoritmo PPO es el primero que muestra un comportamiento prometedor. Se decide seguir probando métodos en vez de alargar el número de pasos con este ensayo. Pero como el A2C, es un algoritmo demasiado sencillo para este tipo de problemas, y no vale la pena abundar en su ejecución.

4.2.4. TD3

El algoritmo TD3 es uno de los más recientes y avanzados. Se prueba directamente con el método 2, el que ha resultado ser más complicado de solventar con los métodos anteriores. En la figura 4.4 se observa cómo el algoritmo **TD3** es capaz de encontrar una solución que permanece estable tomando 200000 pasos de entrenamiento, siempre que reciba una recompensa grande al final de cada episodio satisfactorio. A pesar de ser el algoritmo más

4.2.5. SAC

Por último, los resultados realmente prometedores vienen de la mano del algoritmo **SAC** o **Soft Actor Critic**. Probado también con el método 2 de recompensa, muestra una convergencia a la solución óptima y se mantiene así en toda la ejecución, sin fluctuar.

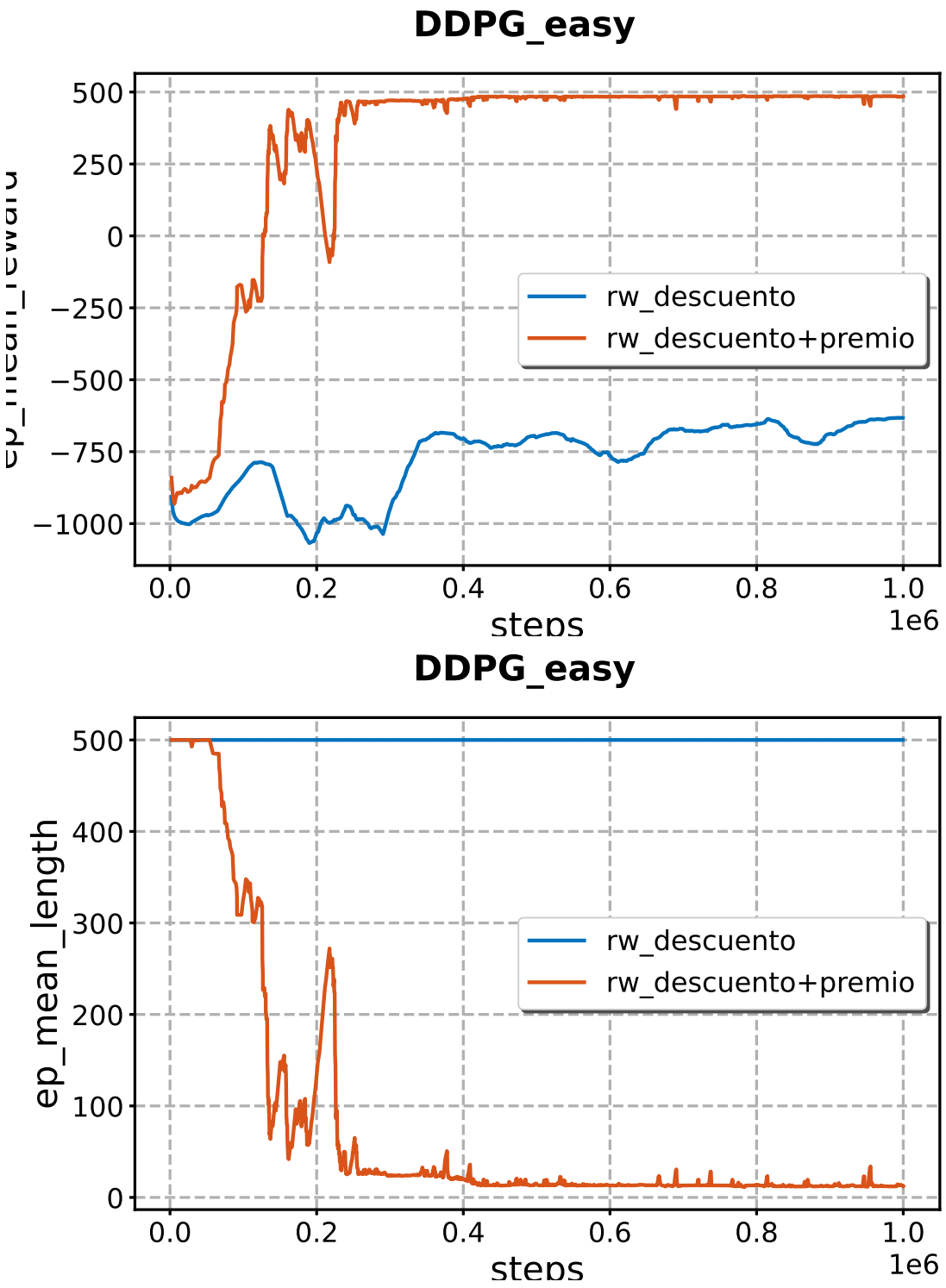


Figura 4.2: Algoritmo DDPG en modo fácil comparando los modelos de recompensa.

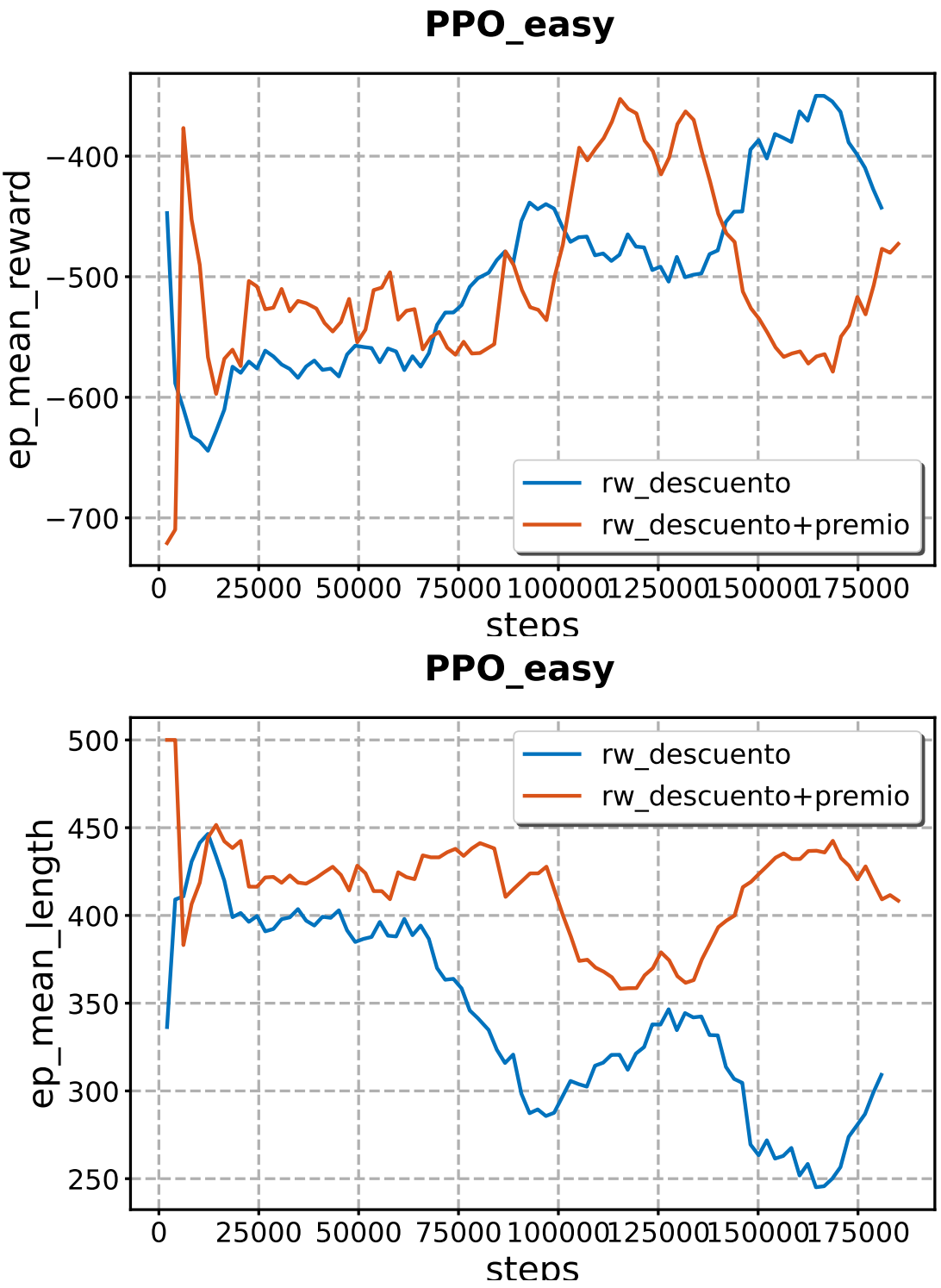


Figura 4.3: Algoritmo PPO en modo fácil comparando los modelos de recompensa.

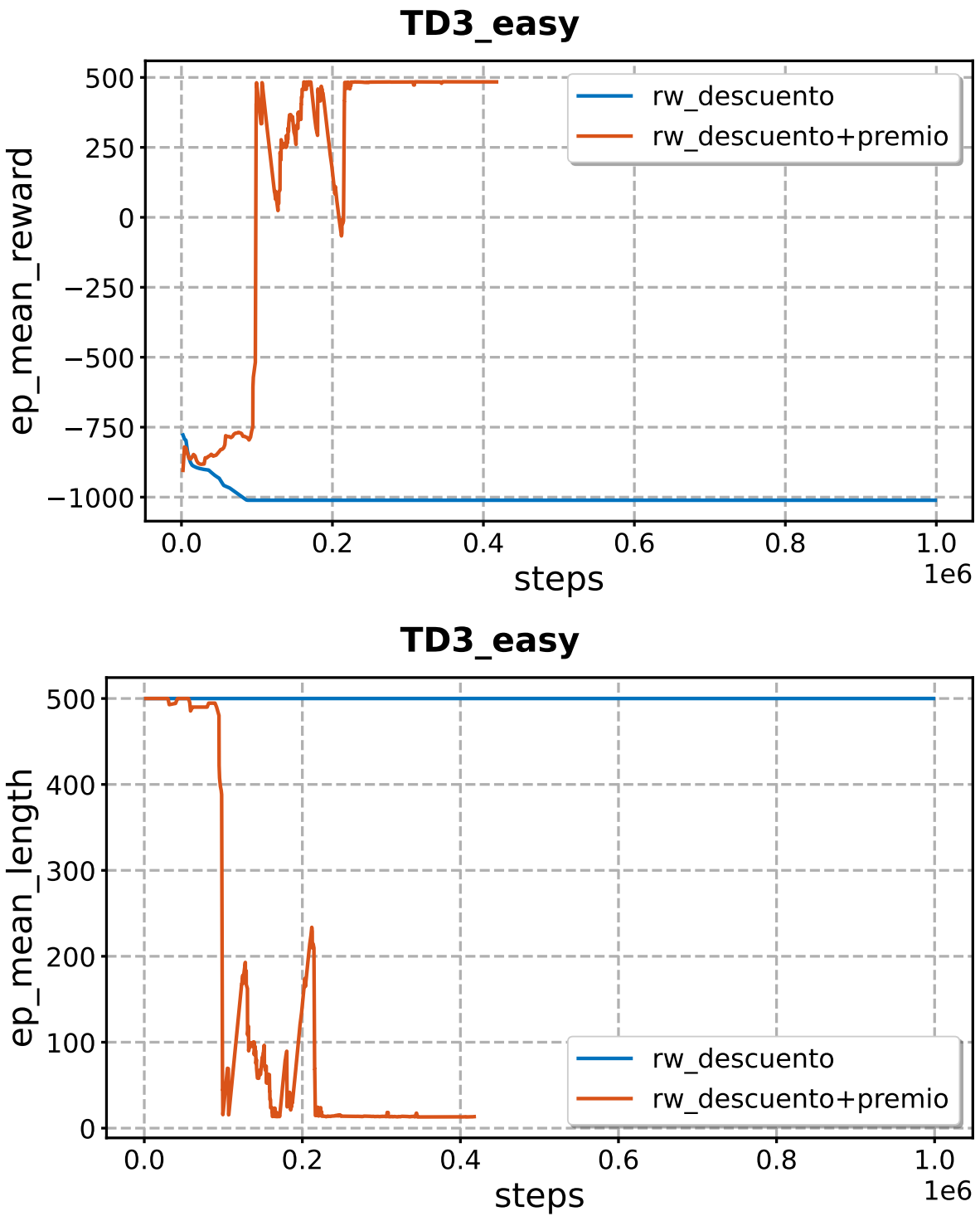


Figura 4.4: Algoritmo TD3 en modo fácil comparando los modelos de recompensa.

4.2.6. Algoritmo SAC

Aunque la gráfica 4.5 sólo muestre el primero de los ensayos éstos se repiten y se obtiene siempre el mismo patrón: una rápida convergencia hacia la solución en el entorno de las 50000 iteraciones. Además, converge a la solución en el mismo número de pasos con ambas estrategias, tanto recibir un premio al final de cada episodio satisfactorio como únicamente con la recompensa en función de la distancia, y de forma más estable. La solución encontrada por el método sin premio final es más óptima que la obtenida con este segundo método, que oscila más. Habría que probar con otras semillas aleatorias para ver si ocurre únicamente en este problema puntualmente. El peso que la exploración del entorno tiene en este algoritmo facilita la resolución de la tarea sin necesidad de manipular la recompensa al final del episodio.

4.3. Problema completo

Con las herramientas adecuadas, el momento de volver a enfrentarse al problema *reach_target_v0*. Tras eliminar la restricción que hacía aparecer el objetivo siempre en la misma posición, se prueba el problema con los valores por defecto, y se obtiene un fracaso.

Los algoritmos RL aceptan múltiples parámetros para adaptarse a las condiciones del problema. Por ejemplo, el búfer de experiencias utilizado para actualizar la política es muy pequeño, así como el número de iteraciones que el algoritmo **SAC** utiliza en explorar el entorno antes de entrenar la política.

Si en esa exploración no ha conseguido resolver el episodio ni una sola vez, podemos considerar que no ha sido una exploración adecuada.

Utilizando la librería *argparse* podemos invocar todos estos parámetros en un fichero **Python** de entrenamieno específico para este algoritmo **SAC**.

Exploración Por defecto, aumenta los pasos de exploración a 20000.

Learning rate Al considerar la longitud de los ensayos, se disminuye el *learning rate* a 0,0003

Tamaño máximo búfer de experiencias El tamaño máximo búfer de iteraciones también multiplica por 10 su valor por defecto, hasta los 1000000

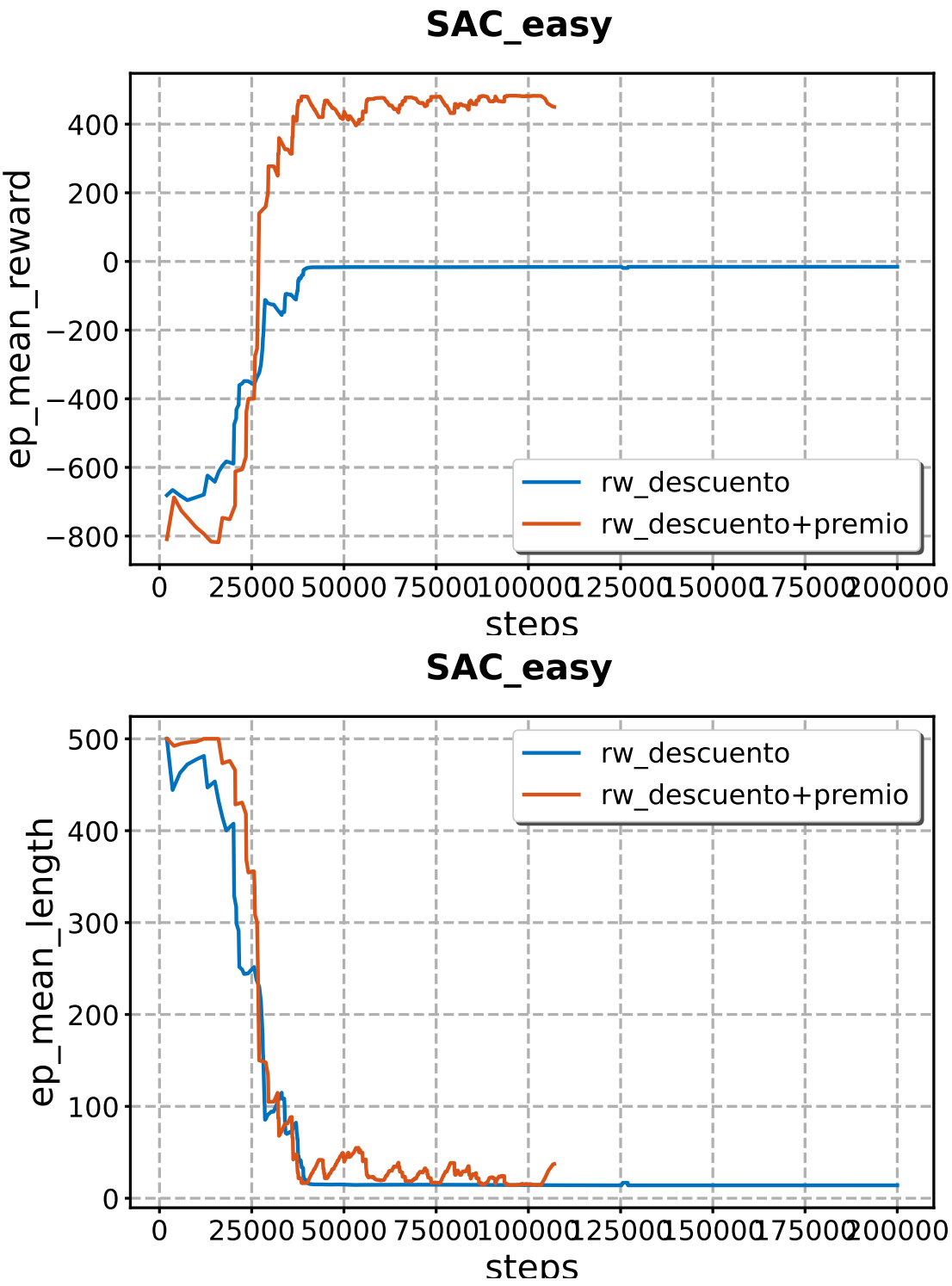


Figura 4.5: Algoritmo SAC en modo fácil comparando los modelos de recompensa.

Semilla aleatoria También es posible variar la semilla Los valores aleatorios utilizan una semilla. Si siempre iniciamos el modelo con los mismos parámetros, no obtenemos una variabilidad suficiente como para probar la validez del algoritmo. Aunque por falta de tiempo no se ejecuten los ensayos con más de un valor de semilla aleatoria, se tiene en cuenta en el código que lanza el entrenamiento.

```
# construct argument parse and parse the arguments
ap = argparse.ArgumentParser()

# Positional arguments
ap.add_argument("workout", help="number␣of␣workout␣steps␣to␣train",
                type=int)

# Optional arguments
ap.add_argument("-t", "--task", required=False, choices=["reach_target"],
                default="reach_target",
                help="Task␣to␣be␣performed.")
ap.add_argument("-i", "--interface", required=False, choices=["human", "rgb", "headless"],
                default="headless",
                help="Enviroment␣interface.")
ap.add_argument("-rm", "--rewardmodel", required=False, choices=["mode0", "mode1", "mode2", "mode3"],
                default="mode2",
                help="choose␣one␣of␣the␣reward␣modes")
ap.add_argument("-nm", "--networkmodel", required=False, type=int, choices=[0, 1, 2, 3, 4, 5],
                default=0,
                help="Choose␣one␣of␣the␣network␣models.␣Different␣models␣for␣CNN␣and␣MLP.␣0␣is␣↙
                      default.␣Check␣file␣for␣more␣info.")
ap.add_argument("-ls", "--learningstarts", required=False, type=int,
                default=20000,
                help="How␣many␣steps␣of␣the␣model␣to␣collect␣transitions␣for␣before␣learning␣starts.")
ap.add_argument("-lr", "--learningrate", required=False, type=float,
                default=0.0003,
                help="Learning␣rate␣for␣optimizer,␣the␣same␣learning␣rate␣will␣be␣used␣for␣all␣↙
                      networks.")
ap.add_argument("-ga", "--gamma", required=False, type=float,
                default=0.99,
                help="The␣discount␣factor")
ap.add_argument("-ta", "--tau", required=False, type=float,
                default=0.005,
                help="The␣soft␣update␣coefficient,␣between␣0␣and␣1")
ap.add_argument("-s", "--seed", required=False, type=int,
                default=0,
                help="Seed␣for␣the␣pseudo␣random␣generators.")
ap.add_argument("-bu", "--buffersize", required=False, type=int,
                default=1000000,
                help="Size␣of␣the␣replay␣buffer.")
ap.add_argument("-ba", "--batchsize", required=False, type=int,
                default=256,
                help="Minibatch␣size␣for␣each␣gradient␣update.")

args = vars(ap.parse_args())
```

4.4. Dimensionando las redes neuronales

Este apartado final está dedicado a dimensionar las redes neuronales del **crítico** y del **actor**. Ha demostrado constituir el problema fundamental a la hora de aplicar el algoritmo **SAC** al problema de robótica.

Las redes neuronales en el algoritmo **Soft Actor Critic** pueden ser comunes para actor

y crítico, o pueden estar diferenciadas. La forma de dimensionaralas es pasar un diccionario con el número de capas al crear la política MLP. Por ejemplo, para rear una red neuronal con 3 capas ocultas de 128 perceptrones por capa:

```
policy_kwargs = dict(net_arch=[128, 128, 128])
model = SAC(policy_mode, env,
            learning_rate=args.get('learningrate'),
            learning_starts=args.get('learningstarts'),
            buffer_size=args.get('buffersize'),
            batch_size=args.get('batchsize'),
            tau=args.get('tau'),
            gamma=args.get('gamma'),
            tensorboard_log="./tb_log_name/",
            policy_kwargs=policy_kwargs,
            verbose=1,
            seed=args.get('seed')
            )
```

4.4.1. Default

Por defecto, la red neuronal MLP del algoritmo **SAC** Una única red, común para crítico y actor, con una salida hacia el espacio de observaciones y una salida hacia el valor del crítico. La arquitectura consta de de 2 capas y 256 perceptrones por capa. Las capas son grandes pero es una red poco profunda para la dinámica de un brazo robótico. El número total de conexiones es del orden de $21e6$. Es una red neuronal común.

Utilizando el argumento *None* en el parámetro *policy_kwargs*, u omitiendo este parámetro, se crea esta red.

```
if args.get('networkmodel') == 0:
       policy_kwargs = None
       policy_name = 'Default'
```

Como se muestra en la figura 4.6, esta red neuronal tiende hacia la solución del problema en el entorno de $1e6$ pasos con el modo de recompensa 3. Sin embargo, no converge hacia ninguna solución con el modo de recompensa 2. La profundidad de la red neuronal puede ser un impedimento a la hora de solucionar el problema, como se ha avanzado en el capítulo anterior.

4.4.2. net=64e3

Podemos disminuir el número de neuronas por capa y sin embargo añadir una nueva capa de profundidad construyendo una única red común para crítico y actor, de 3 capas 64 perceptrones por capa. Es la red más pequeña considerando una profundidad mínima de 3 capas. Mantenemos la arquitectura común entre Actor y Crítico:

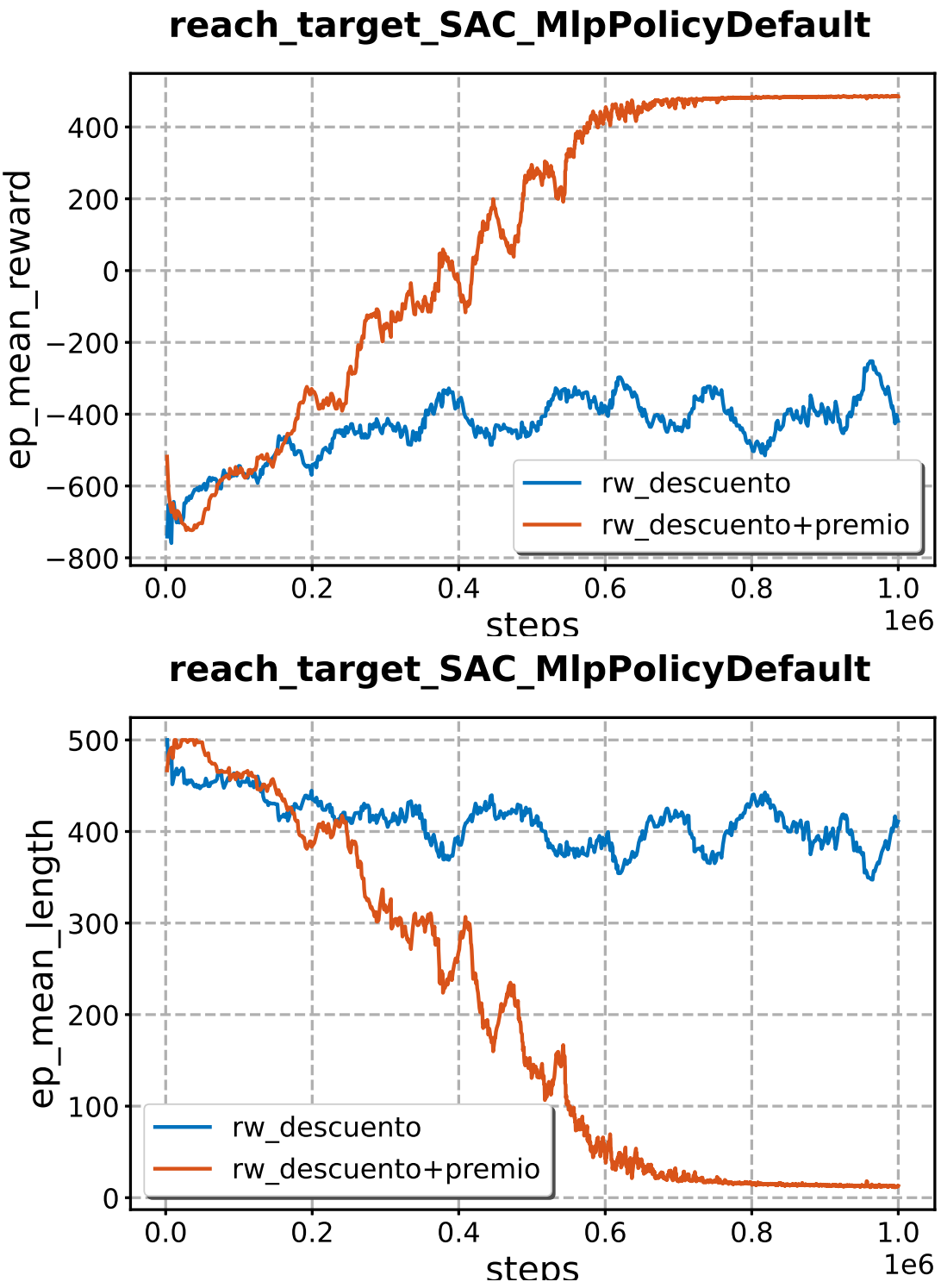


Figura 4.6: Algoritmo SAC con la red por defecto MLP.

```
if args.get('networkmodel') == 2:
    policy_kwargs = dict(net_arch=[64, 64, 64])
    policy_name = 'net=64e3'
```

El número total de conexiones de esta red neuronal es del orden de $84e6$

En la figura 4.7 agente resuelve el entorno con el método de recompensa del premio al final del episodio, pero tarda más tiempo en llegar a una buena solución, en torno al millón y medio de steps. Reducir el número de neuronas por capa aumenta el tiempo que tarda en encontrar la solución. Sin recompensa al final del episodio, no es capaz de resolver el entorno.

4.4.3. net=128e3

Una única red común para crítico y actor, también de 3 capas 128 perceptrones por capa. Podría considerarse un tamaño de red «estándar» para problemas de robótica. Aumentando el número de perceptrones hasta los 128 por capa, el número de conexiones aumenta el orden de $671e6$

```
if args.get('networkmodel') == 1:
    policy_kwargs = dict(net_arch=[128, 128, 128])
    policy_name = 'net=128e3'
```

Nos encontramos en la figura 4.8 con una red neuronal cuyo comportamiento es similar al de la red neuronal por defecto. Sin embargo, converge hacia una solución por ambos métodos, aunque con el premio al final del episodio tarda un poco más que con la red por defecto. Este efecto se observa al disminuir el número de perceptrones por capa.

4.4.4. net=256e3

S Una única red común para crítico y actor, también de 3 capas pero ahora 256 perceptrones por capa. iguiendo el camino emprendido, aumentamos nuevamente el número de perceptrones manteniendo las 3 capas ocultas de la arquitectura común para actor y crítico. Es una red grande, que empieza a pesar computacionalmente.

```
if args.get('networkmodel') == 3:
    policy_kwargs = dict(net_arch=[256, 256, 256])
    policy_name = 'net=256e3'
```

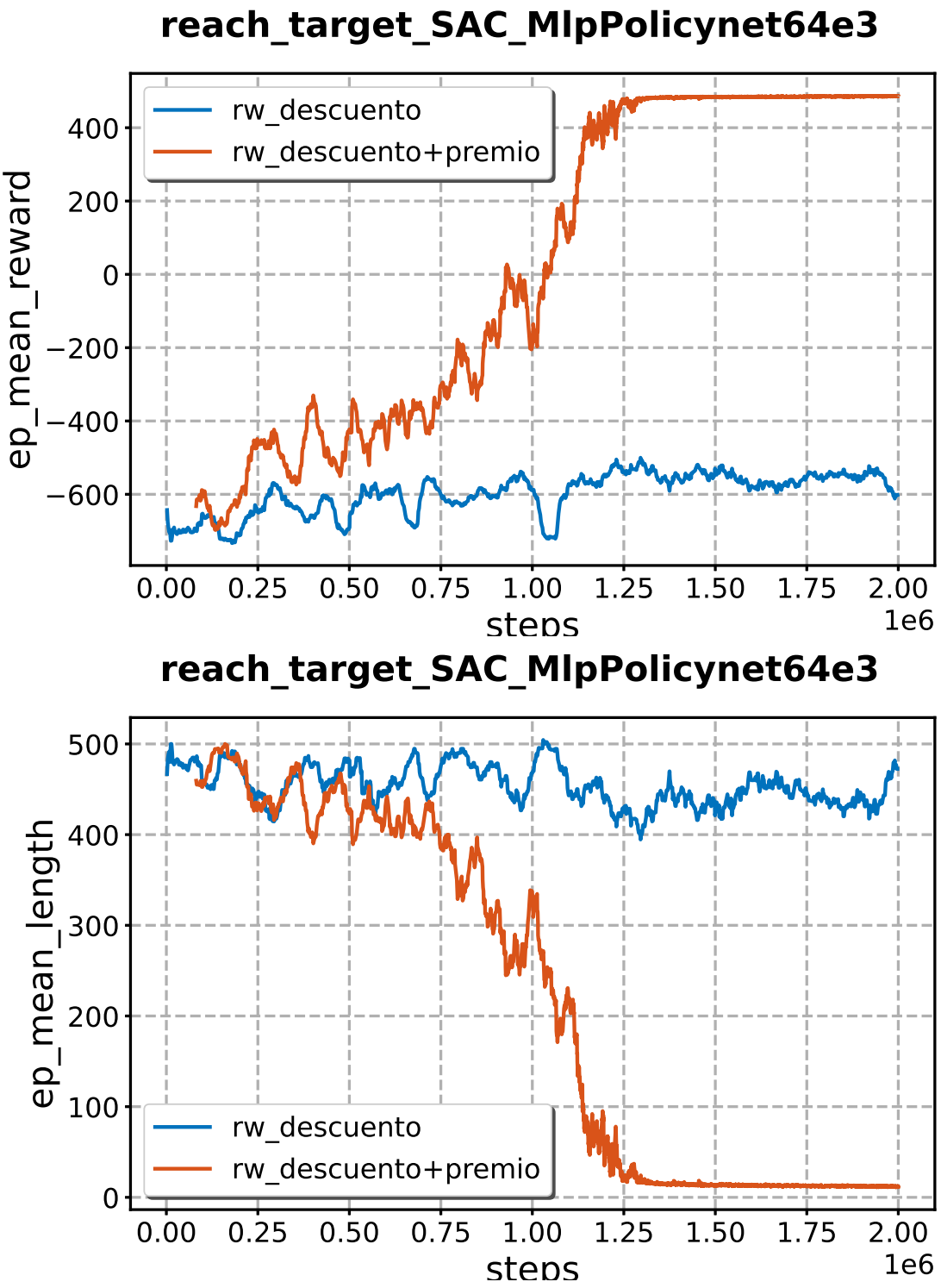


Figura 4.7: Algoritmo SAC con la red [net=64e3] MLP.

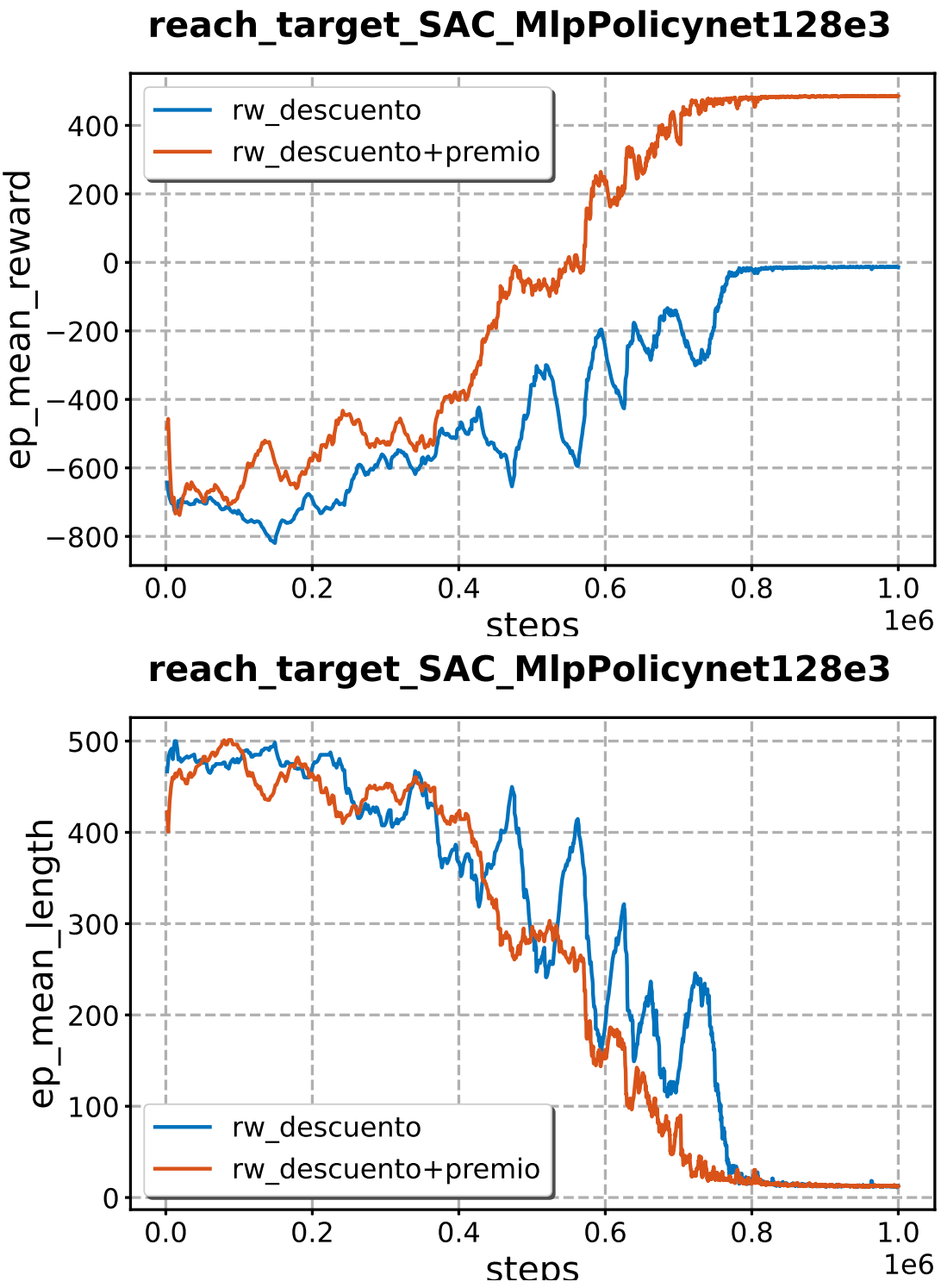


Figura 4.8: Algoritmo SAC con la red [net=128e3] MLP.

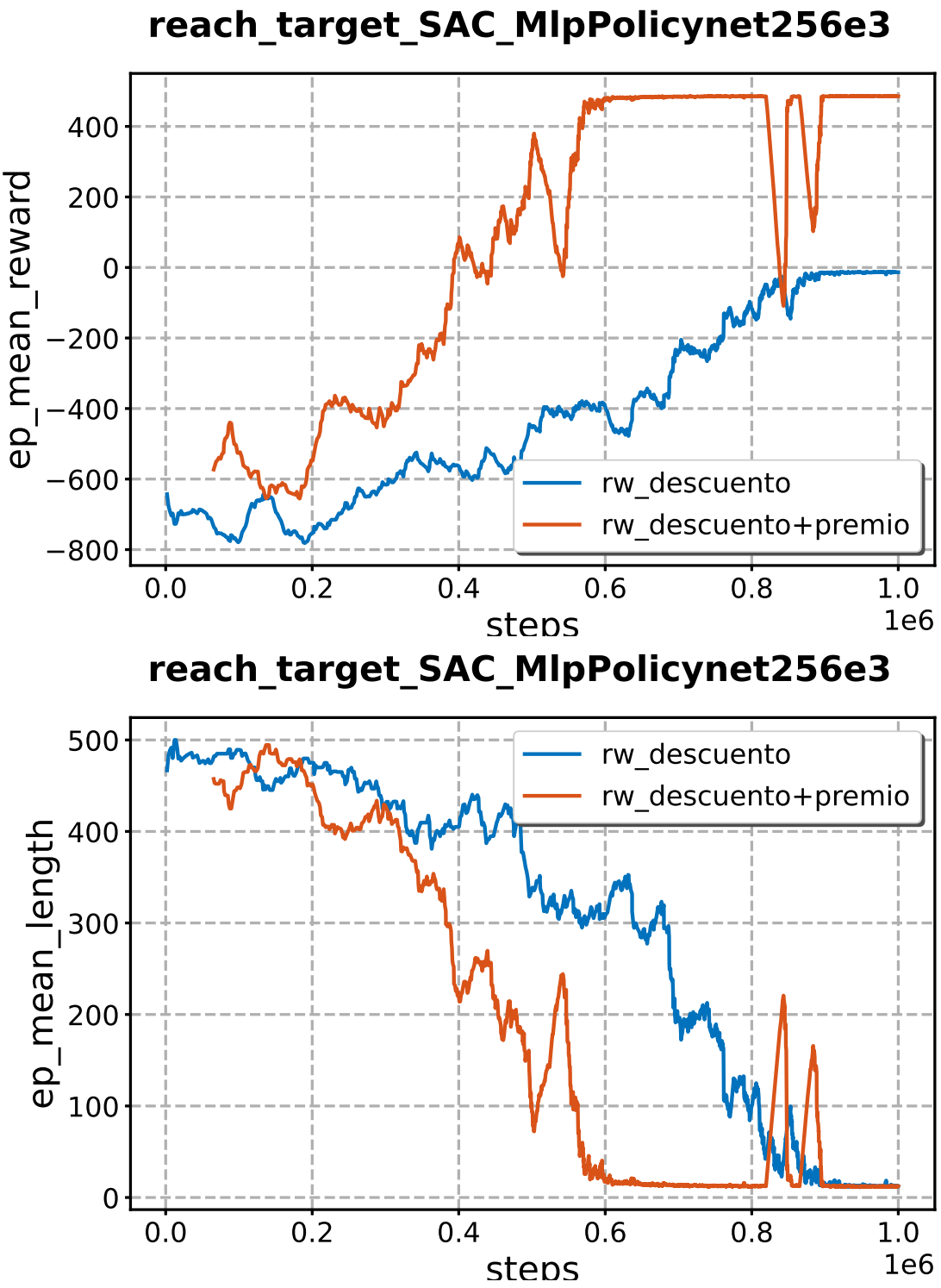


Figura 4.9: Algoritmo SAC con la red [net=256e3] MLP.

Con un número total de conexiones del orden de $5e9$, esta es una de las redes neuronales más extensas, agravando el problema de la inestabilidad y el coste computacional.

La figura 4.9 muestra cómo esta red neuronal es capaz de encontrar una solución ligeramente antes que la capa de 128 perceptrones, aproximadamente a la misma velocidad que la capa por defecto, a costa de encontrar máximos inestables que fluctúan eventualmente hacia peores soluciones durante el entrenamiento. A diferencia de la capa por defecto, también es capaz de encontrar una solución aunque el entorno no devuelva un premio al resolver el episodio completo

4.4.5. pi=64e3q=256e2

Este planteamiento es distinto. Creamos dos arquitecturas separadas: una para el crítico y otra para el actor, manteniendo la estructura del crítico idéntica a la de la política por defecto, y definiendo una red neuronal MLP con 3 capas ocultas de 64 perceptrones para el actor. El actor tiene la profundidad mínima para solventar un problema de robótica, y el crítico tiene un peso «estándar» que coincide con el tamaño por defecto de la red neuronal del algoritmo.

```
if args.get('networkmodel') == 4:
    policy_kwargs = dict(net_arch=dict(pi=[64, 64, 64], qf=[256, 256]))
    policy_name = 'pi=64e3-qf=256e2'
```

El número de conexiones de esta red neuronal está en el orden de $105e6$.

Como se muestra en la figura 4.10, esta aproximación es la mejor de las obtenidas hasta el momento. Converge a una solución en 800000 iteraciones, y lo hace con ámbos métodos, aunque sin premio al final de la recompensa, se observa una inestabilidad al final del episodio. Habría que probar distintas semillas, y mayor número de pasos, para confirmar la inestabilidad de esta red.

4.4.6. pi=64e4q=128e3

Ahora aumentamos un poco más el rizo al aumentar en 1 el número de capas ocultas en las redes neuronales anteriores, y reduciendo el número de perceptrones por capa en la red del crítico a 128. Tenemos una red de 4 capas y 64 perceptrones por capa para el actor, y otra red de 2 capas y 256 perceptrones por capa para el crítico. El actor tiene la red más profunda, pretendiéndose evaluar el efecto que esto supone, y el crítico tiene una red más profunda, 3 capas, pero menos densa, con la mitad de neuronas por capa que la predecesora.

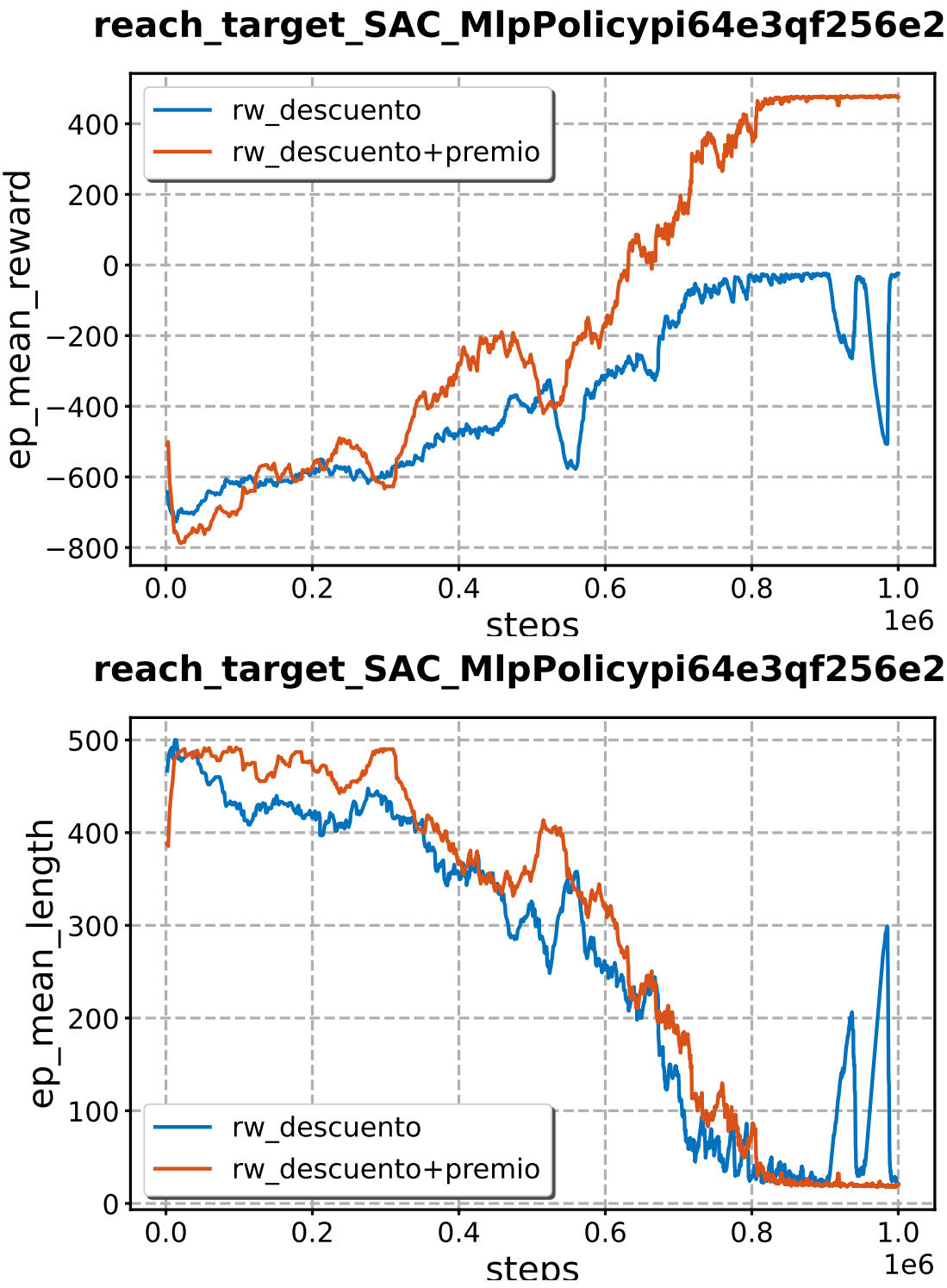


Figura 4.10: Algoritmo SAC con la red [pi=64e3-qf=256e2] MLP.

```
    if args.get('networkmodel') == 5:
        policy_kwargs = dict(net_arch=dict(pi=[64, 64, 64, 64], qf=[128, 128, 128]))
        policy_name = 'pi=64e4-qf=128e3'
```

La red neuronal resultante tiene $5e9$ conexiones, un número similar a la red de 3 capas ocultas de 256 perceptrones por capa.

Los resultados de la figura 4.11 muestran la convergencia más rápida hacia una solución, y mayor estabilidad que en redes neuronales con más perceptrones por capa. Es el mejor resultado y el más estable de todos. Aumentar la profundidad y reducir la densidad de la capa crítica, al contrario de lo esperado, no ha redundado en una convergencia más lenta, y aumentar la profundidad de la red neuronal del actor no comporta un aumento de la inestabilidad, al menos en este ensayo.

4.5. Ensayo con el brazo robótico averiado

El algoritmo **SAC** con resdes neuronales independientes para crítico y actor es capaz de controlar un brazo robótico real de 7 juntas. Pero ¿qué pasa si se estropea una de esas juntas? En teoría, la flexibilidad que ofrece una configuración debería hacer posible que pueda seguir llegando a la mayoría de puntos del espacio modificando el comportamiento del resto de componentes.

En un robot industrial, esto significaría reprogramar completamente el sistema de control. Pero en nuestro caso, el robot se ha programado a sí mismo (el algoritmo aproxima un apolítica óptima π_θ). ¿Qué ocurriría en un caso así?

No es necesario reprogramar el modelo robótico en **CoppeliaSim** para simular una avería. Siguiendo la estrategia ya seguida con la recompensa, podemos construir un envoltorio personalizado que fuerce un mal comportamiento.

```
class disabledRobot(gym.Wrapper):
    _disability: int
    def __init__(self, env=None, disability=0):
        super(disabledRobot, self).__init__(env)
        if disability >127:
            disability = 0
            print(f"Disability␣over␣{disability},␣enter␣a␣7␣bits␣binary␣number")
        if disability > 0:
            print(f"Robot␣disabled␣{disability}␣joints")
        self._disability = disability
    def step(self, action):
        for i, act in enumerate(action):
            if 2**i & self._disability > 0:
                action[i] = 0.0
        #print(f"DEBUG: robot new actions {action}")
        obs, reward, done, info = self.env.step(action)
        return obs, reward, done, info
```

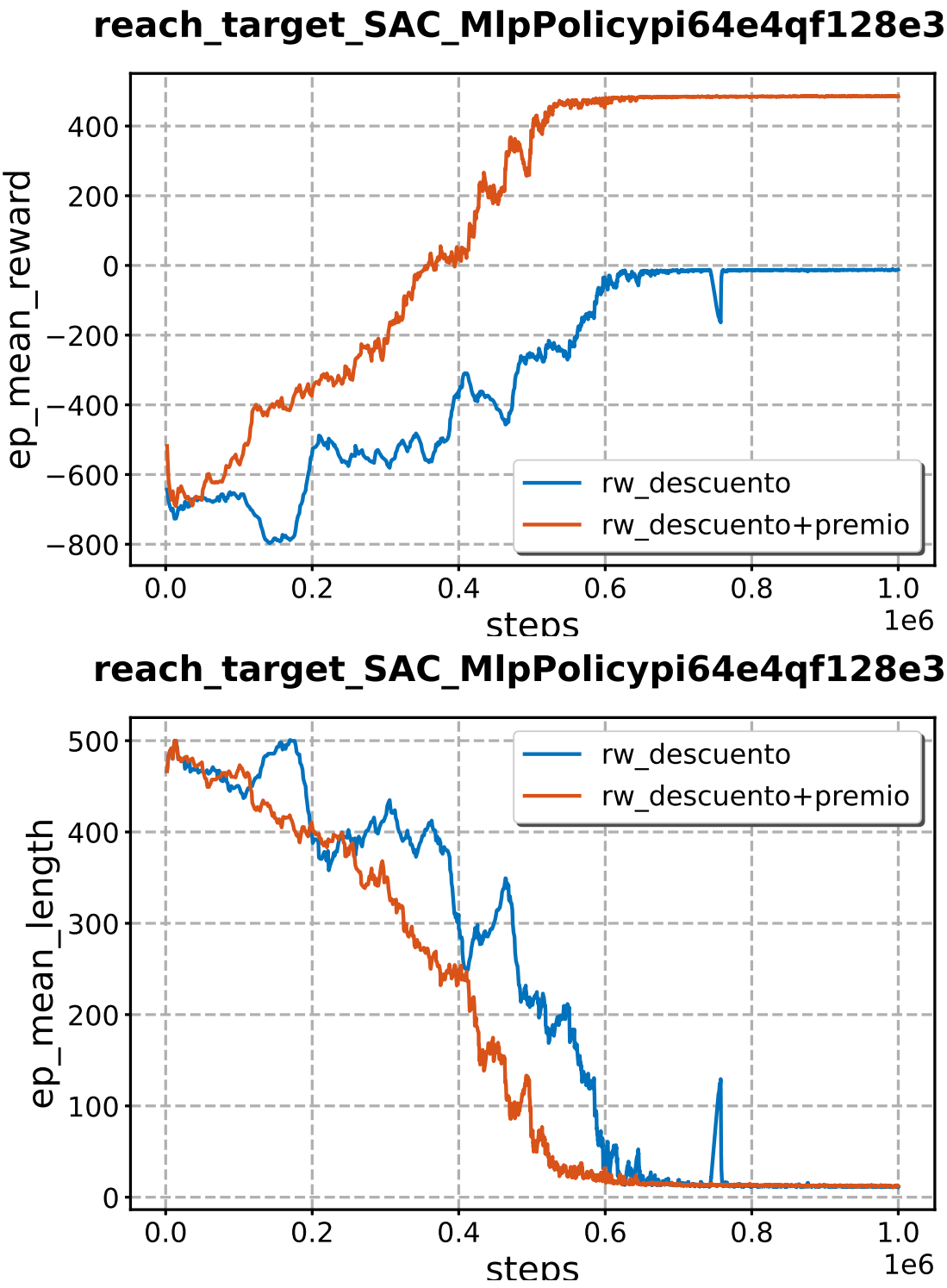


Figura 4.11: Algoritmo SAC con la red [pi=64e4-qf=128e3] MLP.

El *wrapper* admite un número entero inferior a 127, que es el número máximo que puede codificarse en enteros sin signo de 7 bits. Los bits de este entero representan una avería en la junta correspondiente. Para simular una avería, la acción del robot se sustituye con un 0,0 sea cual sea el valor de la acción de entrada al robot, de tal forma que un 4 representa un fallo en la tercera junta, y un 9 representa un fallo en sendas juntas primera y cuarta.

¿Qué ocurre si la avería se da en un robot previamente entrenado? ¿Necesitará más o menos entreanamiento para ser funcional de nuevo? Para responder a esta pregunta, se plantea un programa capaz de cargar tanto modelos previamente entrenados como de crear nuevos, y entrenar el brazo robótico. Las características de estos modelos son idénticos a la red neuronal tipo 5 del ejemplo anterior; es decir, un actor con 4 capas de 64 perceptrones y un crítico con 3 capas de 128 perceptrones. El resto de parámetros también son idénticos.

El código se encuentra tanto en el anexo B como en github [1]. Dada la complejidad con que se guardan los nombres de los modelos, los argumentos de este programa se escogen mediante una interfaz *CLI*.

Se compara el tiempo que tarada un robot previamente entrenado en aprender con el que tarda un robot «averiado» que empieza a entrenar desde 0. los resultados son sorprendentes, ya que el robot entrenado vuelve a aprender en aproximadamente 20000 pasos, muy poco si lo comparamos con el tiempo que necesita el robot para aprender desde 0, aproximadamente $1e6$ pasos.

Listing 4.1: CLI programa robot minusválido

```
(rl4rob) asimon@i3vea1:~/RoboticArm$ python super_sac.py
    ____ _ __ __ __ ____ __ __
   / __ \(_)________ _/ /_ / /__ ____/ / / __ \____ / /_ ____ / /_
  / / / / / ___/ __ `/ __ \/ / _ \/ __ / / /_/ / __ \/ __ \/ __ \/ __/
 / /_/ / (__ ) /_/ / /_/ / / __/ /_/ / / _, _/ /_/ / /_/ / /_/ / /_
/_____/_/____/\__,_/_.___/_/\___/\__,_/ /_/ |_|\____/_.___/\____/\__/


Wrapping the environment in a homemade step to do cosas model mode2

Robot disabled 4 joints
? Choose the model New one
? Enter custom name? pruebacojo
? How many workout steps? 100000
```

A continuación, se exponen los resultados de un brazo robótico entrenado con una red neuronal [pi=64e4-qf=128e3] (la mejor de las anteriores), y un método de recompensa 2 (sin premio final al final de cada episodio), con el algoritmo SAC, al que se le ha averiado la junta 3 (ver anexo F para ver las implicaciones en la dinámica del robot).

[1] https://github.com/albertost85/RoboticArm

Para explorar la viabilidad del episodio (que el robot siga siendo capaz de alcanzar los puntos del espacio a pesar de las dificultades), primero se entrena al brazo robótico empezando de 0, antes de cargar la política de un brazo robótico ya entrenado y comparar resultados.

La junta que se estropea es la 3. Cabe mencionar que estos resultados necesariamente serán distintos en función de la junta estropeada, o de la combinación de juntas estropeadas. Con esta salvedad, se acepta el resultado obtenido como un comportamiento general y no se extiende la experiencia a más ensayos.

4.5.1. Agente por entrenar

El robot empieza el entrenamiento desde 0. Como se muestra en la figura 4.12, los resultados son algo peores a los obtenidos con el brazo robótico intacto. A pesar de tener inutilizada una de las juntas, el espacio de acciones tiene las mismas dimensiones, por lo que se reduce la flexibilidad del robot pero no se reduce el número de dimensiones del problema. Aproximadamente en el entorno de los $1e6$ steps se converge hacia una solución.

4.5.2. Agente entrenado

Tal como muestra la figura 4.13, los resultados son sorprendentes. Obsérvese que el eje de ordenadas empieza a partir de $1e6$ pasos, justo donde acaba el entrenamiento con el robot sin avería. Tras unos pasos iniciales donde tanto la recompensa media como la longitud por episodio media aumentan, rápidamente vuelve a converger hacia una política óptima, que mantiene estable. Sólo han sido necesarios unos 30000 adicionales para compensar esta avería.

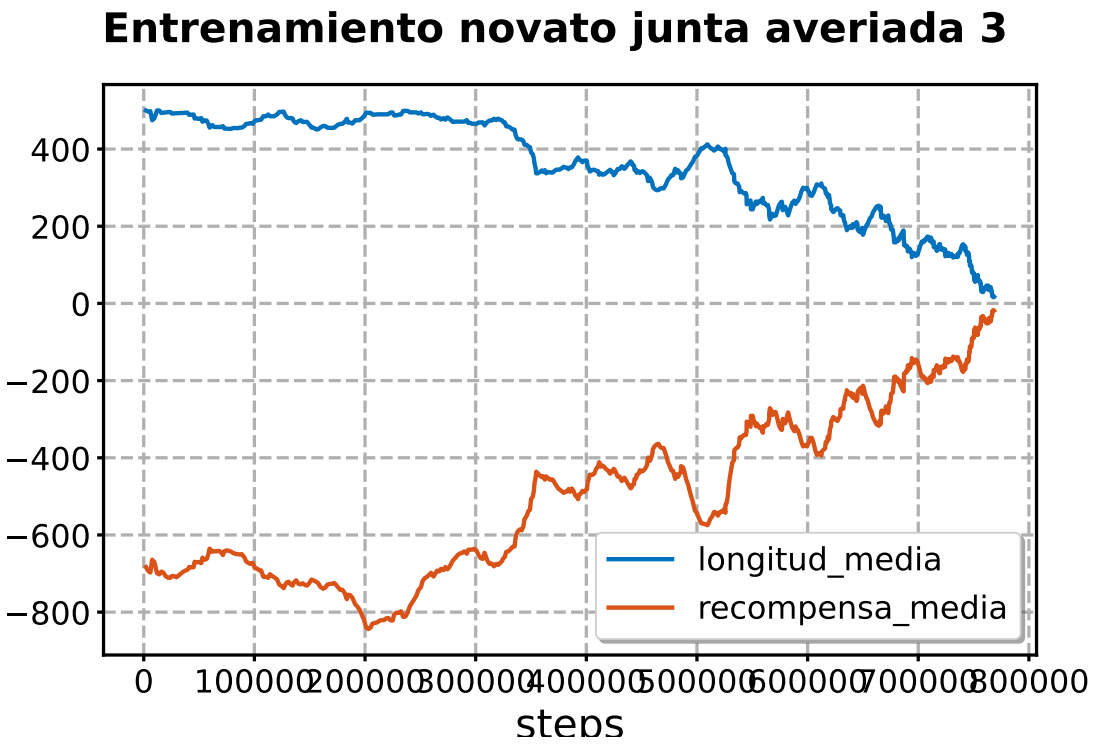


Figura 4.12: Primer entrenamiento de un brazo robótico al que se le estropea la junta 3.

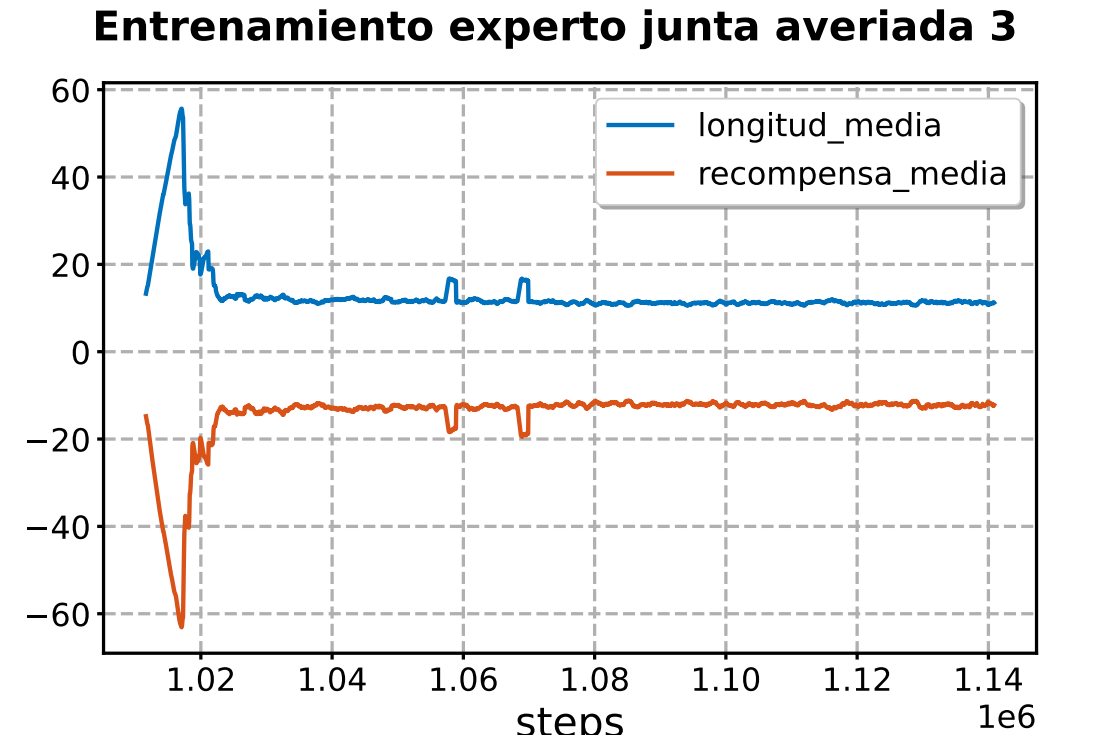


Figura 4.13: Evolución de un brazo robótico entrenado al que se le estropea la junta 3 y vuelve a entrenar.

Capítulo 5

Conclusiones

Es posible controlar un brazo robótico con algoritmos de *Reinforcement Learning*. Las políticas definidas mediante redes neuronales profundas son suficientes para controlar el movimiento del robot optimizando la cinemática. Estudios como [27] o [28] ya alcanzan una conclusión similar, pero programando las redes neuronales con otros procedimientos que nada tienen que ver con nuestro estudio.

El elevado coste en tiempo y recursos computacionales que conlleva entrenar un brazo robótico de esta manera puede ser asumido entrenando la mayor parte del algoritmo en robots virtuales, en nuestro caso **Coppeliasim**, y dejando en manos del robot real bien los últimos pasos del entrenamiento, bien el entrenamiento de una segunda red neuronal adaptativa más pequeña. La ventaja de utilizar redes neuronales para controlar un robot real reside en que, una vez entrenadas, son capaces de ofrecer respuestas suficientemente rápidas como para satisfaces los requisitos de un control en tiempo real [37].

Queda fuera del alcance de este trabajo, pero puede ser una continuación interesante la implementación de una red neuronal con parámetros ya obtenidos en simulación a un brazo robótico real. Un microcontrolador puede emular el comportamiento de una red neuronal ya entrenada; el coste del equipo y la actualización de los pesos de la red neuronal en tiempo real sobre el microcontrolador serían los principales retos de ese nuevo problema.

Cada uno de los ensayos toma varios días de duración. Para dar solidez a los ensayos habría que repetirlos con valores de semilla distintos a 0, considerando el valor promedio y la robustez frente al ruido.

El estado del arte en técnicas de *Reinforcement Learning* es suficiente, a fecha de la presentación de esta memoria, para gobernar un brazo robótico, pero no todos los algo-

ritmos son válidos para este propósito. El método **Soft Actor Critic** o **SAC** ha demostrado ser el mejor de cuantos se han probado, aunque el método **TD3** abra un camino menos prometedor que no se ha explorado. La idoneidad del algoritmo **SAC** radica en su capacidad para explorar aleatoriamente el entorno. La exploración en un enorme entorno de variables discretas que constituye el espacio de estados del robot es clave para solucionar el problema.

Asimismo, en ausencia de un modelo de comportamiento que guíe la construcción de una política inicial para el robot, una de las premisas de entrada de este trabajo, los métodos *off-policy* son los más adecuados para explorar el entorno. La principal desventaja es la capacidad de memoria necesaria para guardar el acumulador o *buffer* de repeticiones.

En cuanto al dimensionamiento de las redes neuronales, en general un mayor número de perceptrones e interconexiones aumenta la variabilidad, y permiten obtener antes una solución, a costa de ser una solución local que por la naturaleza del algoritmo se volverá inestable. Redes neuronales más pequeñas tardan más tiempo en encontrar una solución, pero fluctúan menos. Redes neuronales más profundas son capaces de solucionar problemas imposibles para redes neuronales más pequeñas.

Existe una clara ventaja en utilizar redes separadas para el actor y para el crítico del algoritmo. La red neuronal del crítico no modela una dinámica compleja y necesita menos capas y más neuronas por capa. La red neuronal del actor, modela el comportamiento del brazo robótico, debe tener un número de capas mayor que la del crítico, al menos 3 ó 4 en nuestro caso, y menos neuronas por capa que el crítico.

Bibliografía

[1] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[2] Maneesh Bhand, Ritvik Mudur, Bipin Suresh, Andrew Saxe, and Andrew Ng. Unsupervised learning models of primary cortical receptive fields and receptive field plasticity. In J. Shawe-Taylor, R. Zemel, P. Bartlett, F. Pereira, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 24. Curran Associates, Inc., 2011.

[3] David Silver. Introduction to reinforcement learning with david silver, May 2021.

[4] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic policy gradient algorithms. In *Proceedings of the 31st International Conference on International Conference on Machine Learning - Volume 32*, ICML'14, page I–387–I–395. JMLR.org, 2014.

[5] Jordi Torres. *Introducción al aprendizaje por refuerzo*. Watch This Space, 2021.

[6] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, Jun 2013.

[7] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei Rusu, Joel Veness, Marc Bellemare, Alex Graves, Martin Riedmiller, Andreas Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518:529–33, 02 2015.

[8] Shimon Nof. *Handbook of Industrial Robotics*. John Wiley & Sons, 1999.

[9] Michael E. Moran. Evolution of robotic arms. *Journal of Robotic Surgery*, 2007.

[10] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. Mastering chess and shogi by self-play with a general reinforcement learning algorithm, 2017.

[11] B. Alberts, D. Bray, J. Lewis, M. Raff, K. Roberts, and J.D. Watson. *Molecular Biology of the Cell*. Garland, 4th edition, 2002.

[12] Ulises Castro Peñaloza, Nun Pitalua-Diaz, Jose Ruz-Hernandez, and Ruben Lagunas-Jimenez. *Introducción a los Sistemas Inteligentes.* UABC-UNISON, 12 2009.

[13] Gaudenz Boesch. Deep neural network: The 3 popular types (mlp, cnn and rnn), Oct 2021.

[14] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning, 2013.

[15] Maxim Lapan. *Deep Reinforcement Learning Hands-On*. Packt Publishing, Birmingham, UK, 2018.

[16] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning, 2016.

[17] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning, 2019.

[18] Scott Fujimoto, Herke van Hoof, and David Meger. Addressing function approximation error in actor-critic methods, 2018.

[19] Cornel Secară and Luige Vladareanu. Iterative strategies for obstacle avoidance of a redundant manipulator. *Proceedings of the Romanian Academy - Series A: Mathematics, Physics, Technical Sciences, Information Science*, 9, 01 2010.

[20] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017.

[21] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor, 2018.

[22] Z Bingul, H M Ertunc, and C Oysu. Applying Neural Network to Inverse Kinematic Problem for 6R Robot Manipulator with Offset Wrist. In Bernardete Ribeiro, Rudolf F Albrecht, Andrej Dobnikar, David W Pearson, and Nigel C Steele, editors, *Adaptive and Natural Computing Algorithms*, pages 112–115, Vienna, 2005. Springer Vienna.

[23] H. Zhang and R.P. Paul. A parallel inverse kinematics solution for robot manipulators based on multiprocessing and linear extrapolation. *IEEE Transactions on Robotics and Automation*, 7(5):660–669, 1991.

[24] Richard P. Paul and Bruce Shimano. Kinematic control equations for simple manipulators. In *1978 IEEE Conference on Decision and Control including the 17th Symposium on Adaptive Processes*, pages 1398–1406, 1978.

[25] R Featherstone. Position and Velocity Transformations Between Robot End-Effector Coordinates and Joint Angles. *The International Journal of Robotics Research*, 2(2):35–45, 1983.

[26] James Korein and Norman Badler. Techniques for generating the goal directed motion of articulated structures. *Computer Graphics and Applications, IEEE*, 2:71–81, 12 1982.

[27] Raşit Köker, Cemil Öz, Tarık Çakar, and Hüseyin Ekiz. A study of neural network based inverse kinematics solution for a three-joint robot. *Robotics and Autonomous Systems*, 49(3):227–234, 2004.

[28] P Kalra, P B Mahapatra, and D K Aggarwal. An evolutionary approach for solving the multimodal inverse kinematics problem of industrial robots. *Mechanism and Machine Theory*, 41(10):1213–1229, 2006.

[29] Wisama Khalil. Modeling and Control of Manipulators - Part I: Geometric and Kinematic Models. Lecture, January 2019.

[30] Erwin Coumans. Bullet physics simulation. In *ACM SIGGRAPH 2015 Courses*, SIGGRAPH '15, New York, NY, USA, 2015. Association for Computing Machinery.

[31] Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033, 2012.

[32] Stephen James, Marc Freese, and Andrew J. Davison. Pyrep: Bringing v-rep to deep robot learning. *arxiv*, 6 2019.

[33] Stephen James, Marc Freese, and Andrew J. Davison. Pyrep: Bringing v-rep to deep robot learning. *arXiv preprint arXiv:1906.11176*, 2019.

[34] Stephen James, Zicong Ma, David Rovick Arrojo, and Andrew J. Davison. Rlbench: The robot learning benchmark & learning environment. *IEEE Robotics and Automation Letters*, 2020.

[35] Amir Trabelsi, Juan Sebastián Sandoval Arévalo, Ghiss Moncef, and med amine Laribi. *Development of a Franka Emika Cobot Simulator Platform (CSP) Dedicated to Medical Applications*, pages 95–103. Springer, 05 2021.

[36] Allan Jabri, Kyle Hsu, Ben Eysenbach, Abhishek Gupta, Sergey Levine, and Chelsea Finn. Unsupervised curricula for visual meta-reinforcement learning, 2019.

[37] Yonghua Bai, Minzhou Luo, and Fenglin Pang. An algorithm for solving robot inverse kinematics based on foa optimized bp neural network. *Applied Sciences*, 11:7129, 08 2021.

[38] S. Z. Ramírez, K. Pérez. Self conscious robots in induction heating home appliances. *IEEE transactions on anthropomorphic robots*, 2018.

[39] Brandon S. Allbery, K. Bostic, Drew Eckhardt, Rik Faith, Karl Goetz, Stephen L. Harris, Ian Jackson, A. Jaeger, Jeff Licquia, John A. Martin, Ian McCloghrie, Christopher D. Metcalf, Ian Murdock, David C. Niemi, Lennart Poettering, D. Quinlan, E. Raymond, R. Russell, Mike Sangrey, D. Silber, Thomas Sippel-Dau, Theodore Ts 'o, Stephen Tweedie, Fred N. van Kempen, Bernd Warken, Mats Wichmann, and C. Yeoh. Filesystem hierarchy standard. In *LFS*, 2015.

[40] Lancelot Da Costa, Noor Sajid, Thomas Parr, Karl Friston, and Ryan Smith. The relationship between dynamic programming and active inference: the discrete, finite-horizon case, 2020.

Lista de Figuras

Lista de Tablas

Anexos

Anexos A

Preparación del entorno

El sistema operativo utilizado tanto en el servidor local como en la máquina de la universidad es Ubuntu 18.04.5 LTS en su versión de 64 bits.

En el caso de la máquina local, he utilizado una imagen mini que contiene los componentes imprescindibles del ssitema. Sobre la instalación mínima, se ha añadido el entorno gráfico Ubuntu-Desktop, basado en Gnome.

Una vez configurado el sistema (y hecho limpieza de programas y componentes que no pensaba utilizar, se puede instalar el mismo entorno de desarrollo siguiendo las instrucciones enumeradas a continuación.

A.1. Conda

Conda es un gestor de paquetes y un sistema de gestión de entornos de código abierto, multiplataforma y de lenguaje agnóstico. Está publicado bajo la licencia BSD por Continuum Analytics.[Conda reference].

Conda permite aislar programas, versiones de python y librerías dentro de entornos aislados. Aunque la versión de Python instalada en el sistema operativo sea la 3.9.5 (la última disponible mientras estoy escribiendo este párrafo), es posible crear un entorno Conda con la versión 3.5.2 de Python. Aunque las librerías se installen con pip, siendo éste un paquete más de Conda, se pueden instalar componentes específicos del sistema operativo como la librería GL.

Instalar Curl **Curl** es una herramienta que permite descargar ficheros desde la consola.

Listing A.1: Install curl tool

```
asimon@argonautapc:~$ sudo apt-get install curl
```

Descargar script de instalación Conda se instala con la distribución Anaconda, o mini-conda, una versión con menos características pensada para entornos livianos. Según la documentación, la clave de encriptación debe ser idéntica a `2751ab3d678ff0277ae80f9e8a74f218cfc70fe9a9cdc7bb1c137d7e47e33d53` para la versión 2021.05, Linux x64.

Listing A.2: Download and check Anaconda

```
asimon@argonautapc:~$ curl -O https://repo.anaconda.com/archive/Anaconda3-2021.05-Linux-x86_64.sh
  % Total % Received % Xferd Average Speed Time Time Time Current
                             Dload Upload Total Spent Left Speed
100 544M 100 544M 0 0 11.8M 0 0:00:45 0:00:45 --:--:-- 12.2M
asimon@argonautapc:~$ sha256sum Anaconda3-2021.05-Linux-x86_64.sh
2751ab3d678ff0277ae80f9e8a74f218cfc70fe9a9cdc7bb1c137d7e47e33d53 Anaconda3-2021.05-Linux-x86_64.sh
```

Ejecutar script Al ejecutar el script, hay que pulsar [Enter] hasta leer los términos del acuerdo y aceptar los términos. Una vez aceptados, nos preguntará el directorio de instalación, que he mantenido por defecto en la carpeta de usuario. Por último, nos preguntará si queremos inicializar Anaconda3 con el comando çonda init". Escribimos zes".

Listing A.3: Install Anaconda

```
asimon@argonautapc:~$ bash Anaconda3-2021.05-Linux-x86_64.sh

Welcome to Anaconda3 2021.05

In order to continue the installation process, please review the license
agreement.
Please, press ENTER to continue
...
Please answer 'yes' or 'no':'
>>> yes

Anaconda3 will now be installed into this location:
/home/asimon/anaconda3

  - Press ENTER to confirm the location
  - Press CTRL-C to abort the installation
  - Or specify a different location below
PREFIX=/home/asimon/anaconda3
Unpacking payload ...
...
Preparing transaction: done
Executing transaction: done
installation finished.
Do you wish the installer to initialize Anaconda3
by running conda init? [yes|no]
[no] >>> yes
```

A.1.1. Activar y probar Anaconda

Activar Anaconda Para activar anaconda en el terminal, ejecutamos la instrucción:

Listing A.4: Activates Anaconda

```
asimon@argonautapc:~$ source ~/.bashrc
(base) asimon@argonautapc:~$
```

Ahora, aparece un (base) en la terminal. Aparecerá sin necesidad de ejecutar la instrucción anterior cada vez que abramos una nueva terminal. La palabra *base* indica que estamos en el entorno por defecto.

Enumerar entornos Para ver los entornos diponibles:

Listing A.5: List environments in Conda

```
(base) asimon@argonautapc:~$ conda env list
# conda environments:
#
base * /home/asimon/anaconda3
```

Paquetes disponibles Y para ver los paquetes disponibles para su instalación mediante la instrucción *conda install*:

Listing A.6: List packages in actual environment

```
(base) asimon@argonautapc:~$ conda list
# packages in environment at /home/asimon/anaconda3:
#
# Name Version Build Channel
_ipyw_jlab_nb_ext_conf 0.1.0 py38_0
_libgcc_mutex 0.1 main
alabaster 0.7.12 pyhd3eb1b0_0
...
zope.interface 5.3.0 py38h27cfd23_0
zstd 1.4.5 h9ceee32_0
```

A.1.2. Uso de Conda

Crear entorno Al crear un nuevo entorno de Conda, podemos especificar la versión de Python que queremos usar por defecto. En nuestro caso, la 3.8.8. Hay que aceptar la instalación de una serie de paquetes por defecto.

Listing A.7: Create conda environment

```
(base) asimon@argonautapc:~$ conda create --name rl4rob python=3.8.8
Collecting package metadata (current_repodata.json): done
Solving environment: done


==> WARNING: A newer version of conda exists. <==
  current version: 4.10.1
  latest version: 4.10.3

Please update conda by running

    $ conda update -n base -c defaults conda



## Package Plan ##

  environment location: /home/asimon/anaconda3/envs/rl4rob

  added / updated specs:-
     python=3.8.8
```

```
The following packages will be downloaded:

    package | build---------------------------
    |-----------------
    _openmp_mutex-4.5 | 1_gnu 22 KB
    ca-certificates-2021.7.5 | h06a4308_1 113 KB
    certifi-2021.5.30 | py38h06a4308_0 138 KB
    ld_impl_linux-64-2.35.1 | h7274673_9 586 KB
    libgcc-ng-9.3.0 | h5101ec6_17 4.8 MB
    libgomp-9.3.0 | h5101ec6_17 311 KB
    libstdcxx-ng-9.3.0 | hd4cf53a_17 3.1 MB
    pip-21.1.3 | py38h06a4308_0 1.8 MB
    sqlite-3.36.0 | hc218d9a_0 990 KB------------------------------------------------------------

                                       Total: 11.8 MB

The following NEW packages will be INSTALLED:

  _libgcc_mutex pkgs/main/linux-64::_libgcc_mutex-0.1-main
  _openmp_mutex pkgs/main/linux-64::_openmp_mutex-4.5-1_gnu
  ca-certificates pkgs/main/linux-64::ca-certificates-2021.7.5-h06a4308_1
  certifi pkgs/main/linux-64::certifi-2021.5.30-py38h06a4308_0
  ld_impl_linux-64 pkgs/main/linux-64::ld_impl_linux-64-2.35.1-h7274673_9
  libffi pkgs/main/linux-64::libffi-3.3-he6710b0_2
  libgcc-ng pkgs/main/linux-64::libgcc-ng-9.3.0-h5101ec6_17
  libgomp pkgs/main/linux-64::libgomp-9.3.0-h5101ec6_17
  libstdcxx-ng pkgs/main/linux-64::libstdcxx-ng-9.3.0-hd4cf53a_17
  ncurses pkgs/main/linux-64::ncurses-6.2-he6710b0_1
  openssl pkgs/main/linux-64::openssl-1.1.1k-h27cfd23_0
  pip pkgs/main/linux-64::pip-21.1.3-py38h06a4308_0
  python pkgs/main/linux-64::python-3.8.8-hdb3f193_5
  readline pkgs/main/linux-64::readline-8.1-h27cfd23_0
  setuptools pkgs/main/linux-64::setuptools-52.0.0-py38h06a4308_0
  sqlite pkgs/main/linux-64::sqlite-3.36.0-hc218d9a_0
  tk pkgs/main/linux-64::tk-8.6.10-hbc83047_0
  wheel pkgs/main/noarch::wheel-0.36.2-pyhd3eb1b0_0
  xz pkgs/main/linux-64::xz-5.2.5-h7b6447c_0
  zlib pkgs/main/linux-64::zlib-1.2.11-h7b6447c_3


Proceed ([y]/n)? y
```

Activar entorno Es importante instalar las librerías utilizadas con el entorno de desarrollo activado.

Listing A.8: Activate conda environment

```
(base) asimon@argonautapc:~$ conda activate rl4rob
(rl4rob) asimon@argonautapc:~$
```

Salir del entorno Podemos salir del entorno activando uno nuevo, o desactivando el entorno actual, en cuyo caso volvemos al entorno *base* por defecto.

Listing A.9: Disable conda environment

```
(rl4rob) asimon@argonautapc:~$ conda deactivate
(base) asimon@argonautapc:~$
```

A.1.3. Iconos

Crear icono Anaconda Para crear el icono en el menú de aplicaciones, hay que descargar un icono y crear el fichero *anaconda.desktop*:

Listing A.10: Create Conda Icon

```
(base) asimon@argonautapc:~$ sudo curl -o /opt/CoppeliaSim_Edu_V4_2_0_Ubuntu18_04/helpFiles/logo.png ↙
     https://user-images.githubusercontent.com/8070210/114549467-5793d900-9c61-11eb-88f2-24996a6c03e6.png
  % Total % Received % Xferd Average Speed Time Time Time Current
                              Dload Upload Total Spent Left Speed
100 28161 100 28161 0 0 381k 0 --:--:-- --:--:-- --:--:-- 381k
(base) asimon@argonautapc:~$ cd .local/share/applications/
(base) asimon@argonautapc:~/.local/share/applications$ sudo gedit anaconda.desktop
```

Y rellenar la siguiente información:

```
[Desktop Entry]
Name=Anaconda
Version=2.0
Type=Application
Exec=/home/asimon/anaconda3/bin/anaconda-navigator
Icon=/home/asimon/anaconda3/pkgs/anaconda-navigator-2.0.3-py38_0/lib/pyth
Terminal=false
StartupNotify=true
```

Crear icono Jupyter De forma similar, se puede crear un icono para ejecutar los cuadernos Jupyter.

Listing A.11: Create Jupyter Icon

```
(base) asimon@argonautapc:~$ cd .local/share/applications/
(base) asimon@argonautapc:~/.local/share/applications$ sudo gedit jupyter.desktop
```

```
[Desktop Entry]
Name=Jupyter
Version=6.3
Type=Application
Exec=/home/asimon/anaconda3/bin/jupyter-notebook
Icon=/home/asimon/anaconda3/pkgs/anaconda-navigator-2.0.3-py38_0/lib/pyth
Terminal=false
StartupNotify=true
```

Añadir entorno CONDA a Jupyter Para añadir el entorno CONDA a jupyter, hay que instalar y configurar ipykernel

Listing A.12: Configure conda environment on Jupyter notebook

```
(rl4rob) asimon@argonautapc:~$ conda install -c anaconda ipykernel
Collecting package metadata (current_repodata.json): done
Solving environment: done

...
```

```
(rl4rob) asimon@argonautapc:~$ python -m ipykernel install --user --name=rl4rob
Installed kernelspec rl4rob in /home/asimon/.local/share/jupyter/kernels/rl4rob
```

A.2. Coppeliasim

Coppeliasim es el sucesor de V-Rep. Aunque el motor del proyecto es el entorno *Gym* y la librería enfocada a aprendizaje por refuerzo *Stable Baselines*, el programa Coppeliasim simula la dinámica de un brazo robótico comercial, y constituye el músculo del proyecto.

A.2.1. Descargar e instalar

Descargar Podemos descargar la última versión disponible (V4.2.0) para Ubuntu 18.04 desde la página web del programa (https://www.coppeliarobotics.com/downloads), o utilzando la herramienta Curl:

Listing A.13: Download CoppeliaSim

```
(base) asimon@argonautapc:~$ curl -O ↙
     https://www.coppeliarobotics.com/files/CoppeliaSim_Edu_V4_2_0_Ubuntu18_04.tar.xz
  % Total % Received % Xferd Average Speed Time Time Time Current
                             Dload Upload Total Spent Left Speed
100 147M 100 147M 0 0 11.7M 0 0:00:12 0:00:12 --:--:-- 12.1M
```

Descomprimir ficheros Descomprimimos el contenido del fichero descargado en el directorio del sistema */opt/*

Listing A.14: Extract CoppeliaSim

```
(base) asimon@argonautapc:~$ sudo tar xf CoppeliaSim_Edu_V4_2_0_Ubuntu18_04.tar.xz -C /opt/
```

Instalar librerías adicionales Hay que instalar las siguientes librerías para que coppelia-sim funcione correctamente:

- livavcodec-dev
- libavformat-dev
- libswscale-dev
- gnupg2

Listing A.15: Install libraries for CoppeliaSim

```
(base) asimon@argonautapc:/usr/local/CoppeliaSim_Edu_V4_2_0_Ubuntu18_04$ sudo apt-get install libavcodec-
    dev libavformat-dev libswscale-dev gnupg2
```

Probar Coppeliasim Podemos probar Coppeliasim ejecutando el fichero coppealiasim.sh. Si todo es correcto, debería abrirse la ventana.

Listing A.16: Install libraries for CoppeliaSim

```
\begin{lstlisting}[style=MyBashStyle,caption={Install libraries for CoppeliaSim}]
(base) asimon@argonautapc:/usr/local/CoppeliaSim_Edu_V4_2_0_Ubuntu18_04$ sudo apt-get install libavcodec-
    dev libavformat-dev libswscale-dev gnupg2
```

A.2.2. Añadir iconos

Para añadir un icono de aplicación, creamos el fichero «coppealisim.desktop» en *.localshareapplications*:

Listing A.17: Create Coppeliasim icon

```
(base) asimon@argonautapc:~$ cd .local/share/applications/
(base) asimon@argonautapc:~/.local/share/applications$ sudo gedit coppeliasim.desktop
```

Con el siguiente contenido:

```
[Desktop Entry]
Name=CoppeliaSim
Version=4.2
Type=Application
Exec=/opt/CoppeliaSim_Edu_V4_2_0_Ubuntu18_04/coppeliaSim.sh
Icon=/opt/CoppeliaSim_Edu_V4_2_0_Ubuntu18_04/helpFiles/logo.png
Terminal=false
StartupNotify=true
```

A.2.3. Librerías ROS [opcional]

Asociar el repositorio de los paquetes ROS.

Añadir repositorios Para poder instalar las herramientas utilizando el gestor de paquetes predeterminado en linux.

Listing A.18: Add Ros repositories

```
(base) asimon@argonautapc:~$ curl -s https://raw.githubusercontent.com/ros/rosdistro/master/ros.asc | ↙
     sudo apt-key add -
OK
(base) asimon@argonautapc:~$ sudo sh -c 'echo␣"deb␣[arch=$(dpkg␣--print-architecture)]␣↙
     http://packages.ros.org/ros2/ubuntu␣$(lsb_release␣-cs)␣main"␣>␣/etc/apt/sources.list.d/ros2-
    latest.list'
(base) asimon@argonautapc:~$ sudo apt update
```

Instalar los paquetes ROS *ros-dashing-desktop* y *ros-dashing-base*

Listing A.19: Install ROS packages

```
(base) asimon@argonautapc:~$ sudo apt install ros-dashing-desktop
...
```

```
(base) asimon@argonautapc:~$ sudo apt install ros-dashing-ros-base
...
```

Puesta a punto del entorno Añadiendo el siguiente fichero al souce:

Listing A.20: ROS environment setup

```
(base) asimon@argonautapc:~$ source /opt/ros/dashing/setup.bash
```

Probar la instalación Ejecutando un código de demostración.

Listing A.21: Test ROS environment

```
(base) asimon@argonautapc:~$ ros2 run demo_nodes_cpp talker
[INFO] [talker]: Publishing: 'Hello␣World:␣1'
[INFO] [talker]: Publishing: 'Hello␣World:␣2'
...
```

A.3. Instalar PyRep

PyRep es una librería Python que conecta con el modelo de Coppeliasim. No es una librería enfocada a Aprendizaje por Refuerzo [33], pero es un componente esencial del proyecto.

A.3.1. Configurar variables del sistema de CoppeliaSim

Configurar variables del sistema La librería necesita de las siguientes variables del sistema direccionadas al directorio de instalación de Coppelia:

- COPPELIASIM_ROOT
- LD_LIBRARY_PATH
- QT_QPA_PLATFORM_PLUGIN_PATH

Añadir variables a la terminal Hay que añadir las siguientes líneas para que estén de forma permanente en la terminal. Para no tener que esperar al reinicio, se ejecuta el comando *source* y probamos que efectivamente funcionan.

Listing A.22: Add CoppeliaSim Environment variables

```
(base) asimon@argonautapc:~$ echo '#␣CoppeliaSim␣environment␣variables␣for␣Pyrep>␣export␣↙
     COPPELIASIM_ROOT=/opt/CoppeliaSim_Edu_V4_2_0_Ubuntu18_04/
>␣export␣LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$COPPELIASIM_ROOT
>␣export␣QT_QPA_PLATFORM_PLUGIN_PATH=$COPPELIASIM_ROOT' >> ~/.bashrc
(base) asimon@argonautapc:~$ source ~/.bashrc
(base) asimon@argonautapc:~$ echo $COPPELIASIM_ROOT
/opt/CoppeliaSim_Edu_V4_2_0_Ubuntu18_04/
(base) asimon@argonautapc:~$ cd $COPPELIASIM_ROOT
(base) asimon@argonautapc:/opt/CoppeliaSim_Edu_V4_2_0_Ubuntu18_04$ cd ~
(base) asimon@argonautapc:~$
```

Añadir variables Conda Para que las variables estén disponibles en otras aplicaciones de Conda como Jupyter notebooks, es necesario añadirlas al entorno

Listing A.23: Add CoppeliaSim Environment variables

```
(base) asimon@argonautapc:~$ conda activate rl4rob
(rl4rob) asimon@argonautapc:~$ conda env config vars set COPPELIASIM_ROOT=COPPELIASIM_ROOT
(rl4rob) asimon@argonautapc:~$ conda env config vars set LD_LIBRARY_PATH=$LD_LIBRARY_PATH
(rl4rob) asimon@argonautapc:~$ conda env config vars set ↙
     QT_QPA_PLATFORM_PLUGIN_PATH=$QT_QPA_PLATFORM_PLUGIN_PATH
```

A.3.2. Descargar e instalar librería

No se instala desde el gestor de paquetes y librerías PIP, sino desde el propio github.

Descargar librería Descargamos la librería en la carpeta de usuario. Para descargar la librería desde github, es necesario tener instalada la herramienta *git* en el sistema.

Listing A.24: Download Pyrep library

```
(base) asimon@argonautapc:~$ sudo apt install git

...

(base) asimon@argonautapc:~$ git clone https://github.com/stepjam/PyRep.git
Cloning into 'PyRep'...
remote: Enumerating objects: 1214, done.
remote: Counting objects: 100% (158/158), done.
remote: Compressing objects: 100% (114/114), done.
remote: Total 1214 (delta 94), reused 89 (delta 42), pack-reused 1056
Receiving objects: 100% (1214/1214), 66.70 MiB | 10.50 MiB/s, done.
Resolving deltas: 100% (769/769), done.
(base) asimon@argonautapc:~$ cd PyRep
(base) asimon@argonautapc:~/PyRep$
```

Instalar dependencias Las dependencias que necesita la librería están en el fichero *requirements.txt*. Es importante estar dentro del entorno conda adecuado.

Listing A.25: Install pyrep dependencies

```
(rl4rob) asimon@argonautapc:~/PyRep$ pip install -r requirements.txt
Collecting numpy
  Downloading numpy-1.21.1-cp38-cp38-manylinux_2_12_x86_64.manylinux2010_x86_64.whl (15.8 MB)
     | | 15.8 MB 13.3 MB/s
Collecting cffi==1.14.2
  Downloading cffi-1.14.2-cp38-cp38-manylinux1_x86_64.whl (410 kB)
     | | 410 kB 14.6 MB/s
Collecting pycparser
  Downloading pycparser-2.20-py2.py3-none-any.whl (112 kB)
     | | 112 kB 14.7 MB/s
Installing collected packages: pycparser, numpy, cffi
Successfully installed cffi-1.14.2 numpy-1.21.1 pycparser-2.20
```

Instalar librería Una vez instaladas las dependencias, podemos instar la librería en si.

Listing A.26: Install pyrep library

```
(rl4rob) asimon@argonautapc:~/PyRep$ pip install .
Processing /home/asimon/PyRep
  DEPRECATION: A future pip version will change local packages to be built in-place without first ↙
       copying to a temporary directory. We recommend you use --use-feature=in-tree-build to test your ↙
       packages with this new behavior before it becomes the default.
   pip 21.3 will remove support for this functionality. You can find discussion regarding this at ↙
        https://github.com/pypa/pip/issues/7555.
Building wheels for collected packages: PyRep
  Building wheel for PyRep (setup.py) ... done
  Created wheel for PyRep: filename=PyRep-4.1.0.2-cp38-cp38-linux_x86_64.whl size=632954 ↙
       sha256=c585e82a68820c666f8bb6f09504272a37e44cb81d3a73247692e7cd90865eab
  Stored in directory: /tmp/pip-ephem-wheel-cache-
      oe7f083x/wheels/88/00/a0/ba9cd64798ab76fa657065eb5911a5679194ce325c9fa93d2b
Successfully built PyRep
Installing collected packages: PyRep
Successfully installed PyRep-4.1.0.2
(rl4rob) asimon@argonautapc:~/PyRep$
```

A.3.3. Probar la librería Pyrep

Para probar la librería, podemos ejecutar uno de los ejemplos.

Ejecutar *Panda Reach Target* Si todo ha ido bien, se abrirá una ventana de CoppeliaSim A.1 donde el robot Panda alcanza una bola roja 10 veces.

Listing A.27: Test Pyrep library with example

```
(rl4rob) asimon@argonautapc:~$ python ~/PyRep/examples/example_panda_reach_target.py
Reached target 0!
Reached target 1!
Reached target 2!
Reached target 3!
Reached target 4!
Reached target 5!
Reached target 6!
Reached target 7!
Reached target 8!
Reached target 9!
[CoppeliaSim:loadinfo] done.
```

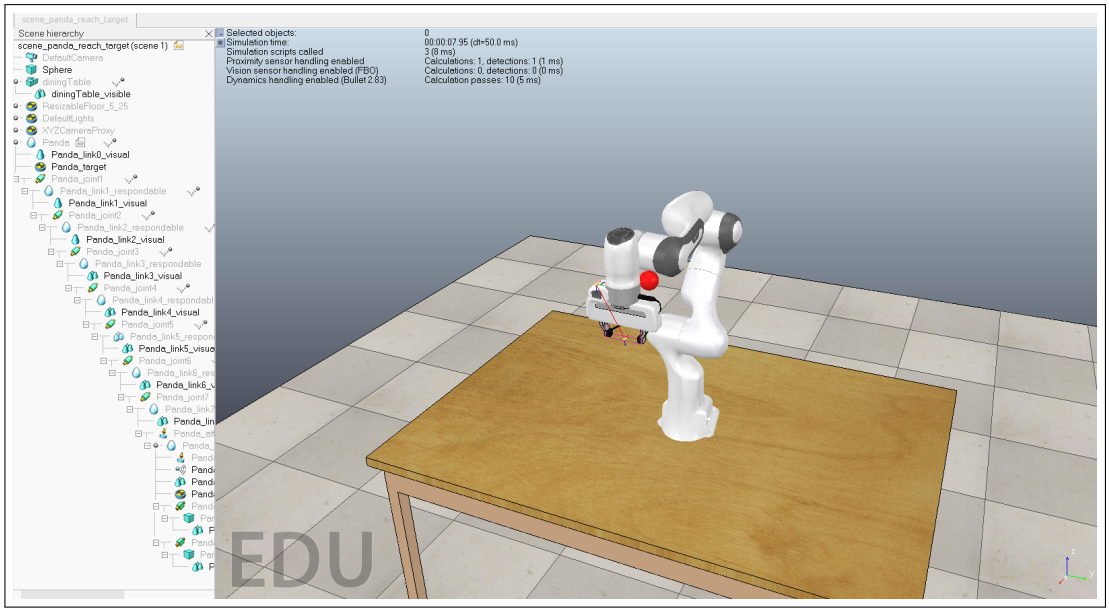


Figura A.1: Ejemplo *Panda-reach-target* de la librería *PyRep*.

A.4. Instalar RLBench

RLBench es una librería pensada para facilitar la investigación en Aprendizaje por Refuerzo, Aprendizaje por imitación, visión por computador, y áreas similares de investigación en robótica. Necesita a la librería Pyrep instalada anteriormente para conectarse con Coppeliasim.

A.4.1. Descargar e instalar librería

También se descarga desde el propio github.

Descargar librería Descargamos la librería en la carpeta de usuario.

Listing A.28: Download Pyrep library

```
(rl4rob) asimon@argonautapc:~/PyRep$ cd ~
(rl4rob) asimon@argonautapc:~$ git clone https://github.com/stepjam/RLBench.git
Cloning into 'RLBench'...
remote: Enumerating objects: 17050, done.
remote: Counting objects: 100% (28/28), done.
remote: Compressing objects: 100% (28/28), done.
remote: Total 17050 (delta 4), reused 14 (delta 0), pack-reused 17022
Receiving objects: 100% (17050/17050), 481.67 MiB | 12.09 MiB/s, done.
Resolving deltas: 100% (3200/3200), done.
Checking out files: 100% (10105/10105), done.
```

Instalar dependencias Las dependencias que necesita la librería están en el fichero *requirements.txt*. Es importante estar dentro del entorno conda adecuado.

Listing A.29: Install pyrep dependencies

```
(rl4rob) asimon@argonautapc:~$ cd RLBench/
(rl4rob) asimon@argonautapc:~/RLBench$ pip install -r requirements.txt
Requirement already satisfied: numpy in /home/asimon/anaconda3/envs/rl4rob/lib/python3.8/site-packages ↙
     (from -r requirements.txt (line 1)) (1.21.1)
Collecting Pillow
  Downloading Pillow-8.3.1-cp38-cp38-manylinux_2_5_x86_64.manylinux1_x86_64.whl (3.0 MB)
     | | 3.0 MB 4.5 MB/s
Collecting pyquaternion
  Downloading pyquaternion-0.9.9-py3-none-any.whl (14 kB)
Collecting html-testRunner
  Downloading html_testRunner-1.2.1-py2.py3-none-any.whl (11 kB)
Requirement already satisfied: setuptools in /home/asimon/anaconda3/envs/rl4rob/lib/python3.8/site-
    packages (from -r requirements.txt (line 5)) (52.0.0.post20210125)
Collecting natsort
  Downloading natsort-7.1.1-py3-none-any.whl (35 kB)
Collecting Jinja2>=2.10.1
  Downloading Jinja2-3.0.1-py3-none-any.whl (133 kB)
     | | 133 kB 14.5 MB/s
Collecting MarkupSafe>=2.0
  Downloading MarkupSafe-2.0.1-cp38-cp38-manylinux2010_x86_64.whl (30 kB)
Installing collected packages: MarkupSafe, Jinja2, pyquaternion, Pillow, natsort, html-testRunner
Successfully installed Jinja2-3.0.1 MarkupSafe-2.0.1 Pillow-8.3.1 html-testRunner-1.2.1 natsort-7.1.1 ↙
     pyquaternion-0.9.9
```

Instalar librería Una vez instaladas las dependencias, podemos instar la librería en si.

Listing A.30: Install pyrep library

```
(rl4rob) asimon@argonautapc:~/RLBench$ pip install .
Processing /home/asimon/RLBench
  DEPRECATION: A future pip version will change local packages to be built in-place without first ↙
       copying to a temporary directory. We recommend you use --use-feature=in-tree-build to test your ↙
       packages with this new behavior before it becomes the default.
   pip 21.3 will remove support for this functionality. You can find discussion regarding this at ↙
        https://github.com/pypa/pip/issues/7555.
Building wheels for collected packages: rlbench
  Building wheel for rlbench (setup.py) ... done
  Created wheel for rlbench: filename=rlbench-1.1.0-py3-none-any.whl size=228813974 ↙
       sha256=0222adbb62e2106795721bec3b26ebd4b78d732c182700fa05fc5c016c6993cd
  Stored in directory: /tmp/pip-ephem-wheel-cache-
      2cz9le3q/wheels/66/25/b2/0acca50e82ffd44c5c057934ba30c490e377abb3be610b0c46
Successfully built rlbench
Installing collected packages: rlbench
Successfully installed rlbench-1.1.0
```

A.4.2. Probar la librería RLBench

Para probar la librería, podemos ejecutar uno de los ejemplos. Existe un ejemplo que utiliza la librería Gym para modelizar un problema de Aprendizaje por Refuerzo.

Ejecutar *Panda Single Task RL* Si todo ha ido bien, se abrirá una ventana de CoppeliaSim A.2 donde el robot Panda ejecuta acciones aleatorias un número finito de veces antes de resetear el episodio.

Listing A.31: Test RLBench library with example

```
(rl4rob) asimon@argonautapc:~/RLBench$ cd ..
(rl4rob) asimon@argonautapc:~$ python RLBench/examples/single_task_rl.py
```

```
Reset Episode
['reach␣the␣red␣target', 'touch␣the␣red␣ball␣with␣the␣panda␣gripper', 'reach␣the␣red␣sphere']
[-0.28168189 0.06569289 -0.0153507 0.17863778 -0.06433866 0.13922382
  0.05112903 1. ]
...
[CoppeliaSim:loadinfo] done.
```

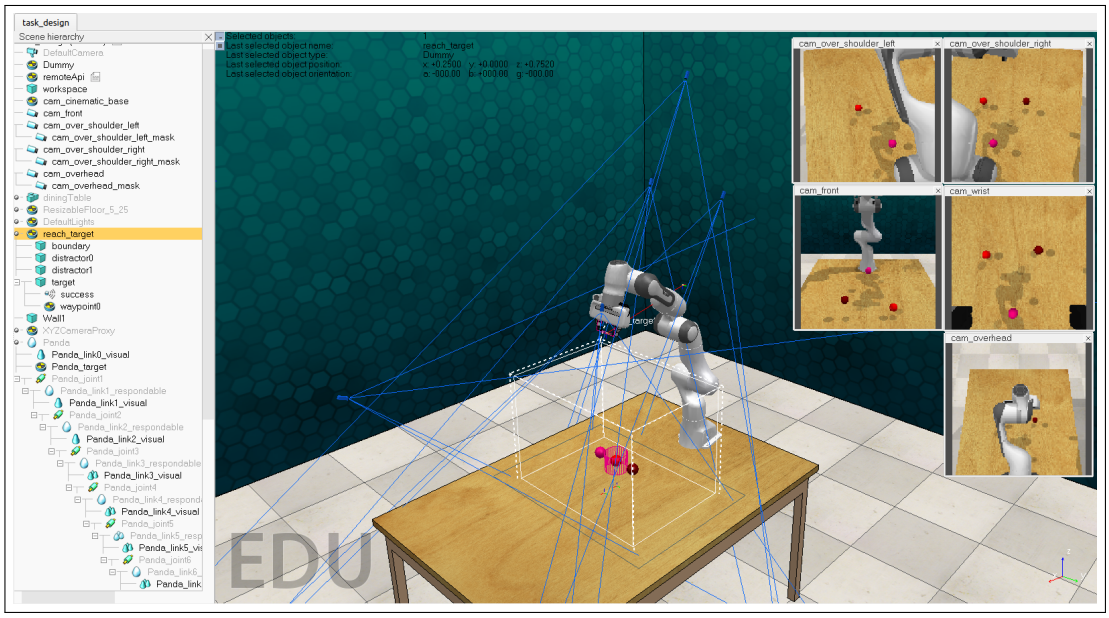


Figura A.2: Ejemplo de una tarea sencilla en *RLBench*.

Problema de compatibilidad con QT Existe un problema de compatibilidad con la librería QT que utiliza coppeliasim. Una forma de solucionarlo es instalar en el entorno conda la versión 4.3.0.36 de librería opencv-python.

Listing A.32: Pip install opencv-python

```
(rl4rob) asimon@argonautapc:~$ pip install opencv-python==4.3.0.36
Collecting opencv-python==4.3.0.36
```

A.5. Librería StableBaselines3

La librería StableBaselines3 utiliza Pytorch en vez de Tensorflow. Se instala con la instrucción *PIP* dentro del entorno de conda.

pyTorch Instalar pytorch y torchvision como paquetes conda.

Listing A.33: Install StableBaselines3 library

```
(rl4rob) asimon@argonautapc:~$ pconda install pytorch torchvision -c pytorch
```

Instalar librería El parámetro *[extra]* indica que se instalarán características adicionales como Tensorboard, OpenCV o atary-py.

Listing A.34: Install StableBaselines3 library

```
(rl4rob) asimon@argonautapc:~$ pip install 'stable-baselines3[extra]'
Collecting stable-baselines3[extra]

...
```

A.6. Instalar Pycharm Pro

Pycharm pro es un IDE o entorno de desarrollo para Python. Aunque al ser un lenguaje interpretado puedan escribirse los scripts en un blog de notas, un entorno de desarrollo ofrece innegables ventajas como la posibilidad de autocompletar expresiones o navegar fácilmente entre el código y las librerías importadas.

Es posible crear una suscripción educativa a Pycharm pro utilizando la cuenta de correo de estudiante de la Universidad de Zaragoza.

A.6.1. Instalar

Descargar instalador Se puede descargar la última versión del instalador desde la propia página web, o utilizando la instrucción curl. Tras la descarga, lo descomprimimos en el directorio que ocupará el programa; en nuestro caso, *opt*

Ejecutar script de instalación dentro del directorio /bin/ existe un script *pycharm.sh* que al ejecutarse por primera vez copia los ficheros de configuración en */.config/JetBrains/PyCharm2021.1.* y los ejecuta.

Listing A.35: Configure Pycharm

```
(base) asimon@argonautapc:~$ curl -O 'https://download-cdn.jetbrains.com/python/pycharm-professional-
    2021.1.3.tar.gz'
  % Total % Received % Xferd Average Speed Time Time Time Current
                                Dload Upload Total Spent Left Speed
100 556M 100 556M 0 0 11.9M 0 0:00:46 0:00:46 --:--:-- 11.5M
(base) asimon@argonautapc:~$ sudo tar -xf pycharm-professional-2021.1.3.tar.gz -C /opt/
(base) asimon@argonautapc:~$ rm pycharm-professional-2021.1.3.tar.gz
```

Create desktop entry It is possible to create desktop icon from settings menu on pycharm, as figure A.3 shows.

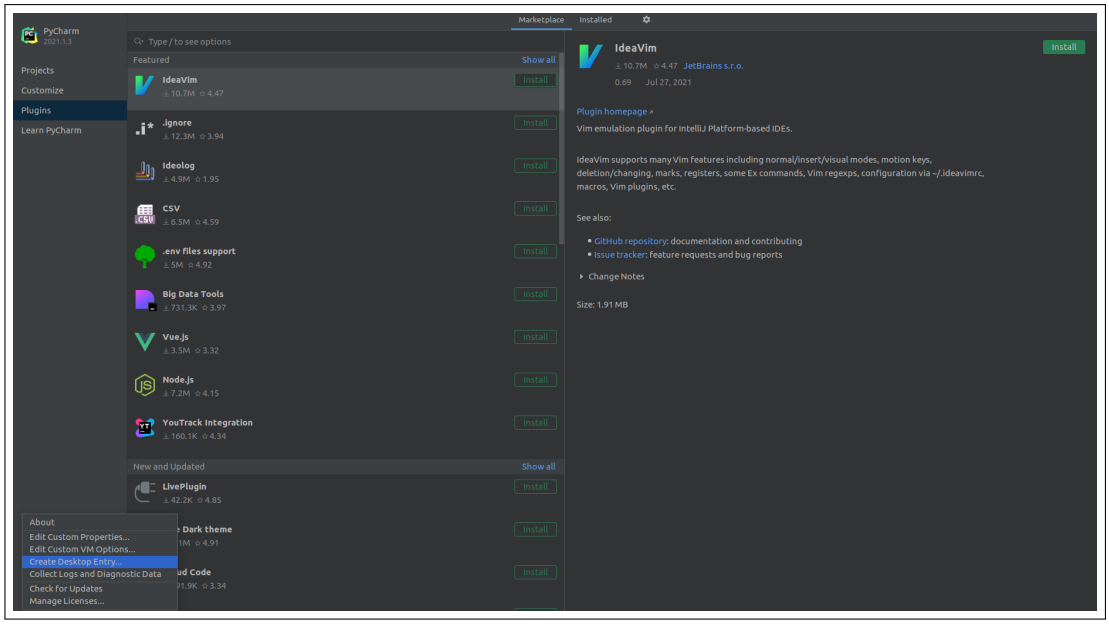


Figura A.3: Asociar un ejecutable desde el programa.

Añadir pycharm al PATH Para ejecutar Pycharm desde cualquier lugar, hay que añadir la ruta del script de ejecución de pycharm al PATH. Una forma de hacerlo es crear un enlace simbólico en el directorio */usr/local/bin*:

Listing A.36: Add Pycharm to PATH

```
(base) asimon@argonautapc:~$ sudo ln -s /opt/pycharm-2021.1.3/bin/pycharm.sh /usr/local/bin/pycharm
```

Configure Anaconda Hay que especificar la ruta de pycharm en Anaconda, tal como se indica en la figura A.4.

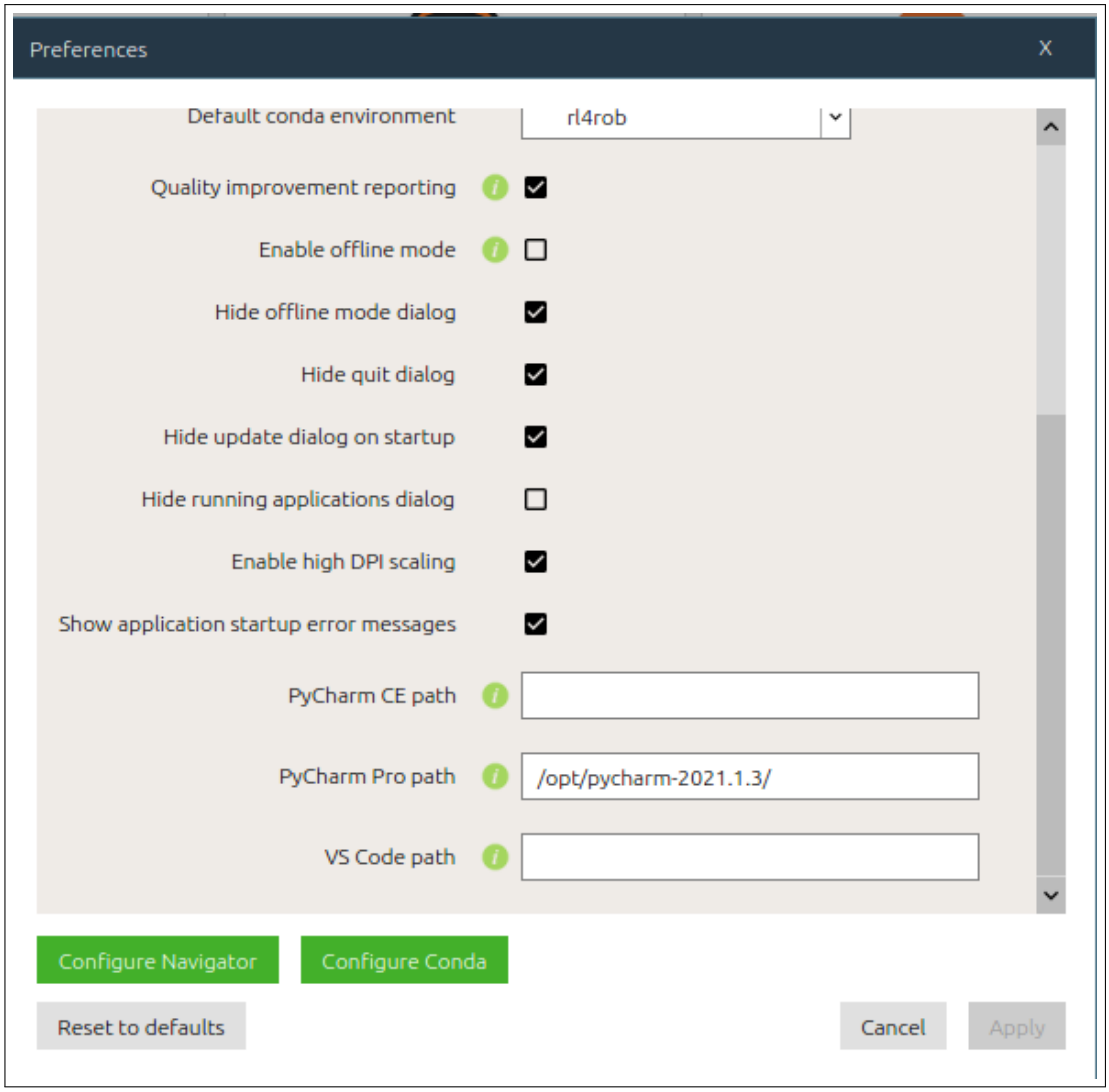


Figura A.4: Pycharm *root config* desde Anaconda.

Crear proyecto en Pycharm Al ejecutar pycharm y crear un proyecto, es necesario escoger el intérprete ya existente que corresponde al intérprete python del entorno *rl4rob* creado. En la figura A.5 se muestra el ejemplo en nuestra instalación.

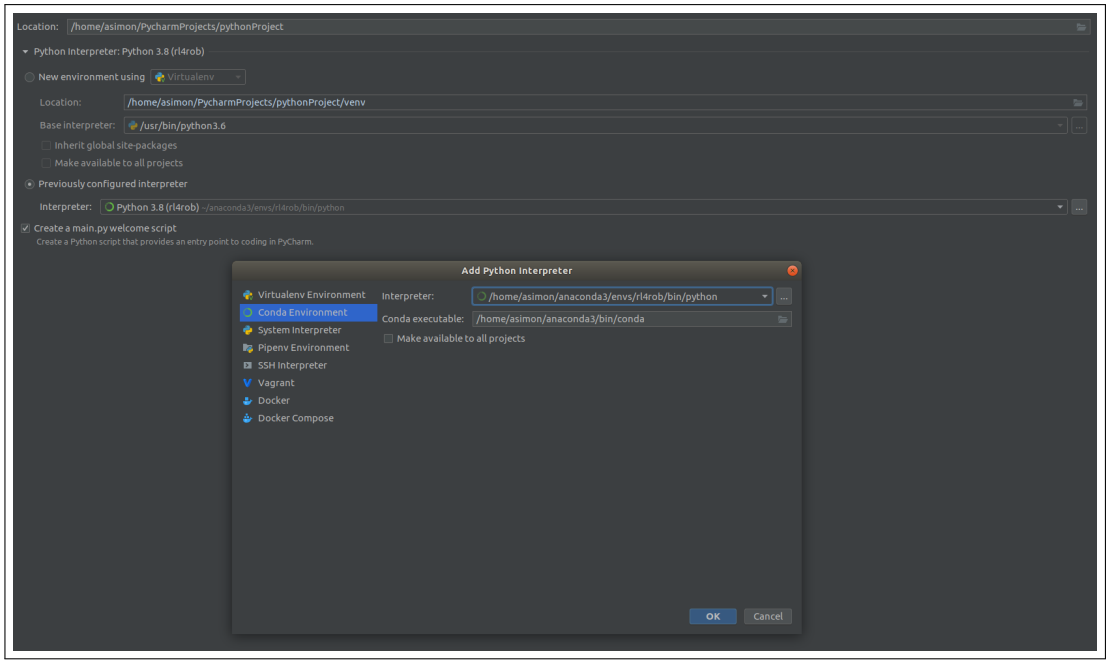


Figura A.5: Selección del intérprete Conda al crear un nuevo proyecto.

A.7. Instalar Google Cloud Suite SDK [opcional]

Google tiene un servicio de computación en la nube, donde se puede configurar un servidor virutal al que conectarse remotamente. A comienzos del proyecto se utilizó esta opción para tener un equipo funcionando con ubuntu 18.04 y el resto de software necesario.

Es posible administrar las instancias en la nube desde la ventana de comandos instalando un *SDK* de google.

A.7.1. Instalar SDK

Añadir la fuente del paquete Para su instalación en Ubuntu, hay que añadir la url de distribución de paquetes de google al sistema, y comprobar que la herramienta *apt-transport-https* está instalada.

Listing A.37: Add Google Coud packages source

```
(base) asimon@argonautapc:~$ echo "deb␣[signed-by=/usr/share/keyrings/cloud.google.gpg]␣↙
     https://packages.cloud.google.com/apt␣cloud-sdk␣main" | sudo tee -a /etc/apt/sources.list.d/google-
    cloud-sdk.list
(base) asimon@argonautapc:~$ sudo apt-get install apt-transport-https ca-certificates gnupg
```

Clave pública de Google Cloud Importamos el fichero que contiene la clave pública de Google Cloud:

Listing A.38: import Google Cloud public key

```
(base) asimon@argonautapc:~$ curl https://packages.cloud.google.com/apt/doc/apt-key.gpg | sudo apt-key --
    keyring /usr/share/keyrings/cloud.google.gpg add -
```

Instalar los paquetes Hay que actualizar los paquetes del sistema antes de instalar el paquete google-cloud-sdk.

Listing A.39: Install Google Cloud

```
(base) asimon@argonautapc:~$ sudo apt-get update && sudo apt-get install google-cloud-sdk
```

A.7.2. Configurar SDK

Inicialización Se inicializa el SDK con el proyecto. Cada instancia está asociada a una zona, que habrá que seleccionar correctamente.

Listing A.40: Init Google Cloud SDK

```
(base) asimon@argonautapc:~$ gcloud init
```

Autenticación Se abre una ventana en el navegador por defecto para la autenticación. Después, la inicialización sigue su curso.

Seleccionar proyecto y zona Es necesario seleccionar el proyecto y la zona donde se ha creado la instancia.

Listing A.41: Configure Google Coud project and zone

```
You are logged in as: [albertost85@gmail.com].

Pick cloud project to use:
 [1] involuted-ratio-312008
 [2] Create a new project
Please enter numeric choice or text value (must exactly match list
item): 1

Your current project has been set to: [involuted-ratio-312008].

Do you want to configure a default Compute Region and Zone? (Y/n)? Y
...
Your project default Compute Engine zone has been set to [europe-north1-a].
You can change it by running [gcloud config set compute/zone NAME].
...
Your project default Compute Engine region has been set to [europe-north1].
```

Ver instancias disponibles Se pueden listar las instancias, su estado, y sus IPs.

Listing A.42: List Google Coud Instances

```
(base) asimon@argonautapc:~$ gcloud compute instances list
NAME ZONE MACHINE_TYPE PREEMPTIBLE INTERNAL_IP EXTERNAL_IP STATUS
rl4robots europe-north1-a e2-standard-2 10.166.0.2 34.88.12.169 RUNNING
```

Arrancar y parar instancia Es posible arrancar y detener instancias

Listing A.43: Manage Google Coud Instances

```
(base) asimon@argonautapc:~$ gcloud compute instances start rl4robots
Starting instance(s) rl4robots...done.
Updated [https://compute.googleapis.com/compute/v1/projects/involuted-ratio-312008/zones/europe-north1-
    a/instances/rl4robots].
Instance internal IP is 10.166.0.2
Instance external IP is 34.88.184.229
(base) asimon@argonautapc:~$ gcloud compute instances stop rl4robots
Stopping instance(s) rl4robots...done.
Updated [https://compute.googleapis.com/compute/v1/projects/involuted-ratio-312008/zones/europe-north1-
    a/instances/rl4robots].
```

Anexos B

Código

Este anexo recoge las partes esenciales del código. Ha sido publicado en el repositorio Github https://github.com/albertost85/RoboticArm, donde se podrá encotnrar la versión más reciente del mismo. Además, se incluyen ficheros complementarios no incluidos en el anexo, así como archivos multimedia, anumaciones, y cuadernos *Colab* o *Jupyter*

B.1. Programa principal

Code/main.py

```
import os
import argparse
import gym
import torch
import rlbench.gym
from myLib.myCosas import execfile
from myLib.myCosas import myEnv
from stable_baselines3 import A2C, PPO, SAC, DDPG, TD3
from torch.utils.tensorboard import SummaryWriter
from stable_baselines3.common.evaluation import evaluate_policy
from stable_baselines3.sac.policies import MlpPolicy, CnnPolicy
from stable_baselines3.common.noise import NormalActionNoise, OrnsteinUhlenbeckActionNoise

writer = SummaryWriter()

# Directories
# Create log dir
log_dir = "logs/"
os.makedirs(log_dir, exist_ok=True)

# construct argument parse and parse the arguments
ap = argparse.ArgumentParser()
ap.add_argument("-e", "--example", required=False, help="Run␣one␣of␣the␣↙
      examples:\n\t'panda_grab'\n\t'panda_gym'"
                                                "\n\t'baselines_lunar'")
ap.add_argument("-t", "--target", required=False, help="Environment␣targets\n\t'reach_target-state-v0'")
ap.add_argument("-i", "--interface", required=False, help="Enviroment␣↙
      interface\n\t'human'\n\t'rgb'\n\t'headless'")
ap.add_argument("-m", "--model", required=False, help="Stable␣baselines3␣↙
      model\n\t'A2C'\n\t'DQN'\n\t'DDPG'"
                                              "\n\t'HER'\n\t'PPO'\n\t'SAC'\n\t'TD3'")
ap.add_argument("-n", "--name", required=False, help="Enter␣name␣of␣the␣model,␣if␣not␣existing,␣will␣be␣↙
      created")
ap.add_argument("-w", "--workout", required=False, help="Number␣of␣iterations␣to␣train␣the␣model")
```

```
ap.add_argument("-v", "--evaluate", required=False, help="Evaluate␣the␣policy␣unless␣is␣zero␣or␣none")
ap.add_argument("-rm", "--rewardmodel", required=False, help="choose␣one␣of␣the␣reward␣modes")
args = vars(ap.parse_args())
args.setdefault("target", "reach_target")
args.setdefault("interface", "rgb")
args.setdefault("model", "A2C")
args.setdefault("rewardmodel", "mode0")

reset_timesteps = False

if args.get('example') is not None:
    if args.get('example') == 'panda_grab':
        execfile("examples/pyrep_panda-reach-target.py")
    elif args.get('example') == 'panda_gym':
        execfile("examples/rlbench_gym.py")
    elif args.get('example') == 'baselines_lunar':
        execfile("examples/testlunar.py")
    else:
        print("Argument␣{}␣is␣not␣a␣valid␣example".format(args.get('example')))
else:
    if args.get('target') == 'reach_target':
        env_name = 'reach_target-state-v0'
    else:
        env_name = 'reach_target-state-v0'
        print("Argument␣{}␣is␣not␣a␣valid␣environment␣target,␣setting␣'reach_target-state-v0'"
              .format(args.get('target')))

    if args.get('interface') == 'human':
        render_mode = 'human'
        observation_mode = 'state'
        policy_mode = 'MlpPolicy'
    elif args.get('interface') == 'rgb':
        render_mode = 'rgb_array'
        observation_mode = 'vision'
        policy_mode = 'CnnPolicy'
    elif args.get('interface') == 'headless':
        render_mode = None
        observation_mode = 'state'
        policy_mode = 'MlpPolicy'
    else:
        render_mode = 'rgb_array'
        observation_mode = 'vision'
        policy_mode = 'CnnPolicy'
        print("Argument␣{}␣is␣not␣a␣valid␣environment␣mode,␣setting␣to␣'array'"
              .format(args.get('interface')))
    if args.get('rewardmodel') is None:
        reward_model = "mode0"
    elif args.get('rewardmodel') == 'mode0':
        reward_model = "mode1"
    elif args.get('rewardmodel') == 'mode1':
        reward_model = "mode1"
    elif args.get('rewardmodel') == 'mode2':
        reward_model = "mode2"
    elif args.get('rewardmodel') == 'mode3':
        reward_model = "mode3"
    else:
        reward_model = "mode0"
    if args.get('model') is None:
        model_name = 'A2C'
    else:
        model_name = args.get('model')

    # All ready to create environment
    print(env_name)
    print(render_mode)
    print(observation_mode)
    env = gym.make(env_name, render_mode=render_mode, observation_mode=observation_mode)
    # Wrapping custom environment to limit episode steps. Wrapper is in external
    # library. Also returns -1 when max limit has been reached.
    env = myEnv(env, mode=reward_model)

    name_model = env_name + '_' + '_' + model_name + '_' + reward_model
    if model_name == 'A2C':
```

```
    try:
        # the saved model does not contain the replay buffer

        model = A2C.load(name_model, env)

        # load it into the loaded_model #NO en A2C, SI en SAC
        # model.load_replay_buffer(name_model + '_replay_buffer')
        # tb_log_name = model.num_timesteps + '_run'
        # now the loaded replay is not empty anymore
        print(f"The loaded_model has {model.num_timesteps} transitions in its buffer")

        # Load the policy independently from the model
        try:
            if policy_mode == 'CnnPolicy':
                policy = CnnPolicy.load(name_model + '_policy') # ojo. si es CNN es distinto
                print("loaded cnn policy from " + name_model + '_policy\n')
            else:
                policy = MlpPolicy.load(name_model + '_policy') # ojo. si es CNN es distinto
                print("loaded mlp policy from " + name_model + '_policy\n')
            model.policy = policy
        except:
            print("no nada")
        # Evaluate the loaded policy
        # mean_reward, std_reward = evaluate_policy(saved_policy, env, n_eval_episodes=1, ↙
              deterministic=True)

        # print(f"mean_reward={mean_reward:.2f} +/- {std_reward}")

        # Retrieve the environment
        # env = model.get_env()

    except FileNotFoundError:
        print(f"File model not found: {name_model} creating new one")
        model = A2C(policy_mode, env, n_steps = 5, verbose=1, tensorboard_log="./tb_log_name/")
        reset_timesteps = True
elif model_name == 'PPO':
    try:
        model = PPO.load(name_model, env)
        print(f"The loaded_model has {model.num_timesteps} transitions in its buffer")
        # Load the policy independently from the model
        try:
            if policy_mode == 'CnnPolicy':
                policy = CnnPolicy.load(name_model + '_policy') # ojo. si es CNN es distinto
                print("loaded cnn policy from " + name_model + '_policy\n')
            else:
                policy = MlpPolicy.load(name_model + '_policy')
                print("loaded mlp policy from " + name_model + '_policy\n')
            model.policy = policy
        except:
            print("no nada")
    except FileNotFoundError:
        print(f"File model not found: {name_model} creating new one")
        model = PPO(policy_mode, env, verbose=1, tensorboard_log="./tb_log_name/")
        reset_timesteps = True
elif model_name == 'SAC':
    try:
        model = SAC.load(name_model, env)
        print(f"The loaded_model has {model.num_timesteps} transitions in its buffer")

        # Load the policy independently from the model
        try:
            if policy_mode == 'CnnPolicy':
                policy = CnnPolicy.load(name_model + '_policy') # ojo. si es CNN es distinto
                print("loaded cnn policy from " + name_model + '_policy\n')
            else:
                policy = MlpPolicy.load(name_model + '_policy') # ojo. si es CNN es distinto
                print("loaded mlp policy from " + name_model + '_policy\n')
            model.policy = policy
        except:
            print("no nada")
    except FileNotFoundError:
        print(f"File model not found: {name_model} creating new one")
        policy_kwargs = dict(net_arch=dict(pi=[64, 64, 64], qf=[256, 256]))
```

```
        #policy_kwargs = dict(net_arch=[128, 128, 128])
        #policy_kwargs = None
        model = SAC(policy_mode, env, verbose=1, tensorboard_log="./tb_log_name/", ↙
              policy_kwargs=policy_kwargs) #, buffer_size = 1000000, learning_starts=20000), ↙
              batch_size=1000, train_freq = (1, "episode"))
        reset_timesteps = True
elif model_name == 'TD3':
    try:
        model = TD3.load(name_model, env)
        print(f"The␣loaded_model␣has␣{model.num_timesteps}␣transitions␣in␣its␣buffer")

        # Load the policy independently from the model
        try:
            if policy_mode == 'CnnPolicy':
                policy = CnnPolicy.load(name_model + '_policy') # ojo. si es CNN es distinto
                print("loaded␣cnn␣policy␣from␣" + name_model + '_policy\n')
            else:
                policy = MlpPolicy.load(name_model + '_policy') # ojo. si es CNN es distinto
                print("loaded␣mlp␣policy␣from␣" + name_model + '_policy\n')
            model.policy = policy
        except:
            print("no␣nada")
    except FileNotFoundError:
        print(f"File␣model␣not␣found:␣{name_model}␣creating␣new␣one")
        model = TD3(policy_mode, env, verbose=1, tensorboard_log="./tb_log_name/")
        reset_timesteps = True
elif model_name == 'DDPG':
    try:
        model = DDPG.load(name_model, env)

        print(f"The␣loaded_model␣has␣{model.num_timesteps}␣transitions␣in␣its␣buffer")

        # Load the policy independently from the model
        try:
            if policy_mode == 'CnnPolicy':
                policy = CnnPolicy.load(name_model + '_policy') # ojo. si es CNN es distinto
                print("loaded␣cnn␣policy␣from␣" + name_model + '_policy\n')
            else:
                policy = MlpPolicy.load(name_model + '_policy')
                print("loaded␣mlp␣policy␣from␣" + name_model + '_policy\n')
            model.policy = policy
        except:
            print("no␣nada")
    except FileNotFoundError:
        print(f"File␣model␣not␣found:␣{name_model}␣creating␣new␣one")
        model = DDPG(policy_mode, env, verbose=1, tensorboard_log="./tb_log_name/")
        reset_timesteps = True
else:
    print("Argument␣{}␣is␣not␣a␣valid␣model,␣setting␣to␣'A2C'"
          .format(args.get('model')))
    try:
        model = A2C.load(name_model, env)
    except FileNotFoundError:
        model = A2C(policy_mode, env, verbose=1)

if args.get('workout') is not None:
    # Evaluate the model every n steps
    # and save the evaluation to the "logs/" folder
    # model.set_env(env)
    #my_eval_env = gym.make(env_name, render_mode=render_mode, observation_mode=observation_mode)
    #my_eval_env = myEnv(env, mode=reward_model)
    final_steps = int(args.get('workout')) + model.num_timesteps
    # Si se interrumpe el entrenamiento por runtimeerror de coppelia, probar si funciona.
    while model.num_timesteps < final_steps:
        if reset_timesteps:
            print("No␣need␣to␣reset␣environment\n")
        else:
            model.env.reset()
        try:
            model.learn(int(args.get('workout')), eval_freq=10000, n_eval_episodes=5, eval_env = env, ↙
                  eval_log_path="./eval_logs_" + name_model + '_' + reward_model + '/', ↙
                  tb_log_name=model_name + '_' + reward_model, reset_num_timesteps=reset_timesteps)
        except RuntimeError as e:
```

```
            print("\n---------------------------\nRuntime␣error:␣{0}\n---------------------------
                \n".format(e))
            print("Saving␣model␣with␣{0}␣timesteps\n".format(model.num_timesteps))
            reset_timesteps = False
        # save the model
        model.save(name_model)


    # now save the replay buffer too No con A2C
    # model.save_replay_buffer(name_model + '_replay_buffer')

    # Save the policy independently from the model
    # Note: if you don't save the complete model with `model.save()`
    # you cannot continue training afterward
    policy = model.policy
    policy.save(name_model + '_policy')

if args.get('evaluate') is not None:
    # Evaluate the policy evaluate_policy function not existing
    num_evals = int(args.get('evaluate'))
    if num_evals > 0:
        mean_reward, std_reward = evaluate_policy(model.policy, env, n_eval_episodes=num_evals, ↙
            deterministic=True)
        print(f"mean_reward={mean_reward:.2f}␣+/-␣{std_reward}")

"""
# Pruebas
import matplotlib.pyplot as plt
env.render(mode=render_mode)
obs = env.reset()
plt.imshow(obs['wrist_rgb'])
plt.show()
img = obs['front_rgb'] # state, left_shoulder_rgb, right_shoulder_rgb
plt.imshow(img)
plt.show()"""

env.close()
```

B.2. Programa entrenamiento SAC

Code/train_sac.py

```
import os
import argparse
import gym
import rlbench.gym
from myLib.myCosas import myEnv
from torch.utils.tensorboard import SummaryWriter
from stable_baselines3.common.evaluation import evaluate_policy
from stable_baselines3.common.noise import NormalActionNoise, OrnsteinUhlenbeckActionNoise
from stable_baselines3 import SAC
from stable_baselines3.sac.policies import MlpPolicy, CnnPolicy


# construct argument parse and parse the arguments
ap = argparse.ArgumentParser()

# Positional arguments
ap.add_argument("workout", help="number␣of␣workout␣steps␣to␣train",
                type=int)

# Optional arguments
ap.add_argument("-t", "--task", required=False, choices=["reach_target"],
                default="reach_target",
                help="Task␣to␣be␣performed.")
ap.add_argument("-i", "--interface", required=False, choices=["human", "rgb", "headless"],
                default="headless",
                help="Enviroment␣interface.")
ap.add_argument("-rm", "--rewardmodel", required=False, choices=["mode0", "mode1", "mode2", "mode3"],
```

```
                default="mode2",
                help="choose␣one␣of␣the␣reward␣modes")
ap.add_argument("-nm", "--networkmodel", required=False, type=int, choices=[0, 1, 2, 3, 4, 5],
                default=0,
                help="Choose␣one␣of␣the␣network␣models.␣Different␣models␣for␣CNN␣and␣MLP.␣0␣is␣default.␣↙
                     Check␣file␣for␣more␣info.")
ap.add_argument("-ls", "--learningstarts", required=False, type=int,
                default=20000,
                help="How␣many␣steps␣of␣the␣model␣to␣collect␣transitions␣for␣before␣learning␣starts.")
ap.add_argument("-lr", "--learningrate", required=False, type=float,
                default=0.0003,
                help="Learning␣rate␣for␣optimizer,␣the␣same␣learning␣rate␣will␣be␣used␣for␣all␣networks.")
ap.add_argument("-ga", "--gamma", required=False, type=float,
                default=0.99,
                help="The␣discount␣factor")
ap.add_argument("-ta", "--tau", required=False, type=float,
                default=0.005,
                help="The␣soft␣update␣coefficient,␣between␣0␣and␣1")
ap.add_argument("-s", "--seed", required=False, type=int,
                default=0,
                help="Seed␣for␣the␣pseudo␣random␣generators.")
ap.add_argument("-bu", "--buffersize", required=False, type=int,
                default=1000000,
                help="Size␣of␣the␣replay␣buffer.")
ap.add_argument("-ba", "--batchsize", required=False, type=int,
                default=256,
                help="Minibatch␣size␣for␣each␣gradient␣update.")

args = vars(ap.parse_args())

if args.get('task') == 'reach_target':
    env_name = 'reach_target-state-v0'

if args.get('interface') == 'human':
    render_mode = 'human'
    observation_mode = 'state'
    policy_mode = 'MlpPolicy'
elif args.get('interface') == 'rgb':
    render_mode = 'rgb_array'
    observation_mode = 'vision'
    policy_mode = 'CnnPolicy'
elif args.get('interface') == 'headless':
    render_mode = None
    observation_mode = 'state'
    policy_mode = 'MlpPolicy'

# Network choices based on policy mode are defined here
if policy_mode == 'CnnPolicy':
        policy_kwargs = None
        policy_name = 'Default'
elif policy_mode == 'MlpPolicy':
    if args.get('networkmodel') == 0:
        policy_kwargs = None
        policy_name = 'Default'
    elif args.get('networkmodel') == 1:
        policy_kwargs = dict(net_arch=[128, 128, 128])
        policy_name = 'net=128e3'
    elif args.get('networkmodel') == 2:
        policy_kwargs = dict(net_arch=[64, 64, 64])
        policy_name = 'net=64e3'
    elif args.get('networkmodel') == 3:
        policy_kwargs = dict(net_arch=[256, 256, 256])
        policy_name = 'net=256e3'
    elif args.get('networkmodel') == 4:
        policy_kwargs = dict(net_arch=dict(pi=[64, 64, 64], qf=[256, 256]))
        policy_name = 'pi=64e3-qf=256e2'
    elif args.get('networkmodel') == 5:
        policy_kwargs = dict(net_arch=dict(pi=[64, 64, 64, 64], qf=[128, 128, 128]))
        policy_name = 'pi=64e4-qf=128e3'
```

```
reward_model = args.get('rewardmodel')

name_experiment = f"{args.get('task')}_SAC_{policy_mode}[{policy_name}]_rw={reward_model}_" \
                  f"lr={args.get('learningrate')}_starts={args.get('learningstarts')}_" \
                  f"buffer={args.get('buffersize')}_batch={args.get('batchsize')}_" \
                  f"tau={args.get('tau')}_gamma={args.get('gamma')}_seed={args.get('seed')}"
name_short = f"{args.get('task')}_SAC_{policy_mode}[{policy_name}]_{reward_model}_seed={args.get('seed')}"
# Summary writer for tensorboard
writer = SummaryWriter()

# Directories
# Create log dir
log_dir = "SAC_eval_logs"
model_dir = "SAC_models"
os.makedirs(log_dir, exist_ok=True)
os.makedirs(model_dir, exist_ok=True)

# All ready to create environment
env = gym.make(env_name, render_mode=render_mode, observation_mode=observation_mode)

# Wrapping custom environment to limit episode steps. Wrapper is in external
# library. Also returns -1 when max limit has been reached.
env = myEnv(env, mode=reward_model)

print(name_short)



# Create Reinforcement Learning model
reset_timesteps = False
try:
    model = SAC.load(f"{model_dir}/{name_experiment}", env)
    print(f"The␣loaded_model␣has␣{model.num_timesteps}␣transitions␣in␣its␣buffer")
    # Load replay buffer
    try:
        model.load_replay_buffer(f"{model_dir}/{name_experiment}_replay_buffer")
        print(f"Loaded␣buffer␣from␣{model_dir}/{name_experiment}_replay_buffer")
    except:
        print(f"Something␣went␣wrong␣while␣loading␣buffer␣{model_dir}/{name_experiment}_replay_buffer␣↙
              from␣disk")
    # Load the policy independently from the model
    try:
        if policy_mode == 'CnnPolicy':
            policy = CnnPolicy.load(f"{model_dir}/{name_experiment}_policy") # ojo. si es CNN es distinto
            print(f"Loaded␣CNN␣policy␣from␣{model_dir}/{name_experiment}_policy")
        else:
            policy = MlpPolicy.load(f"{model_dir}/{name_experiment}_policy") # ojo. si es MLP es distinto
            print(f"Loaded␣MLP␣policy␣from␣{model_dir}/{name_experiment}_policy")
        model.policy = policy
    except:
        print(f"Something␣went␣wrong␣while␣loading␣policy␣{policy_mode}␣from␣disk")
except FileNotFoundError:
    print(f"File␣model␣not␣found:␣{name_experiment}␣creating␣new␣one")
    reset_timesteps = True
    model = SAC(policy_mode, env,
                learning_rate=args.get('learningrate'),
                learning_starts=args.get('learningstarts'),
                buffer_size=args.get('buffersize'),
                batch_size=args.get('batchsize'),
                tau=args.get('tau'),
                gamma=args.get('gamma'),
                tensorboard_log="./tb_log_name/",
                policy_kwargs=policy_kwargs,
                verbose=1,
                seed=args.get('seed')
                ) # train_freq = (1, "episode"))

# Estimación de episodios que debería tener
final_steps = int(args.get('workout')) + model.num_timesteps

# Si se interrumpe el entrenamiento por runtimeerror de coppelia, probar si funciona.
while model.num_timesteps < final_steps:
    if reset_timesteps:
```

```
        print("No␣need␣to␣reset␣environment\n")
    else:
        model.env.reset()
    try:
        model.learn(args.get('workout'), eval_freq=10000, n_eval_episodes=5, eval_env = env, ↙
             eval_log_path=f"./{log_dir}/{name_short}/", tb_log_name=name_short, ↙
             reset_num_timesteps=reset_timesteps)
    except RuntimeError as e:
        print("\n----------------------------\nRuntime␣error:␣{0}\n----------------------------\n".format(e))
        print("Saving␣model␣with␣{0}␣timesteps\n".format(model.num_timesteps))
        reset_timesteps = False
    # save the model
    model.save(f"{model_dir}/{name_experiment}")


    # now save the replay buffer too
    model.save_replay_buffer(f"{model_dir}/{name_experiment}_replay_buffer")

    # Save the policy independently from the model
    # Note: if you don't save the complete model with `model.save()`
    # you cannot continue training afterward
    policy = model.policy
    policy.save(f"{model_dir}/{name_experiment}_policy")

# close environment
env.close()
print(f"C'est␣fini.␣Total␣timesteps={model.num_timesteps}")
```

B.3. Programa entrenamiento modelos entrenados SAC

Code/super_sac.py

```
import os
import sys
import gym
import torch
import rlbench.gym
from os.path import isfile, join
from pyfiglet import Figlet
from PyInquirer import style_from_dict, Token, prompt
from examples import custom_style_2
from prompt_toolkit.validation import Validator, ValidationError
from myLib.myCosas import myEnv, disabledRobot
from torch.utils.tensorboard import SummaryWriter
from stable_baselines3.common.evaluation import evaluate_policy
from stable_baselines3.common.noise import NormalActionNoise, OrnsteinUhlenbeckActionNoise
from stable_baselines3 import SAC
from stable_baselines3.sac.policies import MlpPolicy, CnnPolicy

class NumberValidator(Validator):

    def validate(self, document):
        try:
            int(document.text)
        except ValueError:
            raise ValidationError(message="Please␣enter␣a␣number",
                                  cursor_position=len(document.text))

class GymModelValidator(Validator):
    def validate(self, document):
        pass
mypath = os.path.realpath('.')
mypath = os.path.join(mypath, 'SAC_models')

def return_files():
    return [f for f in os.listdir(mypath) if (not f.endswith("buffer")) and (not f.endswith("policy") ↙
          and isfile(join(mypath, f)))]
```

```
class NumberValidator(Validator):

    def validate(self, document):
        try:
            int(document.text)
        except ValueError:
            raise ValidationError(message="Please␣enter␣a␣number",
                                  cursor_position=len(document.text))

mystyle = style_from_dict({
    Token.Separator: '#cc5454',
    Token.QuestionMark: '#673ab7␣bold',
    Token.Selected: '#cc5454', # default
    Token.Pointer: '#673ab7␣bold',
    Token.Instruction: '', # default
    Token.Answer: '#f44336␣bold',
    Token.Question: '',
})

questions = [
    {
        'type': 'list',
        'name': 'model_open',
        'message': 'Choose␣the␣model',
        'choices': ["New␣one"] + return_files()
    },
    {
        'type': 'input',
        'name': 'custom_name',
        'message': 'Enter␣custom␣name?',
        'validate': GymModelValidator
    },
    {
        'type': 'input',
        'name': 'model_workout',
        'message': 'How␣many␣workout␣steps?',
        'validate': NumberValidator,
        'filter': lambda val: int(val)
    },
]

def main():
    f = Figlet(font='slant')
    print(f.renderText('Disabled␣Robot'))
    env_name = 'reach_target-state-v0'
    render_mode = None
    observation_mode = 'state'
    policy_mode = 'MlpPolicy'
    env = gym.make('reach_target-state-v0', render_mode=None, observation_mode='state')
    env = myEnv(env, mode='mode2')
    env = disabledRobot(env, 4)
    answers = prompt(questions, style=mystyle)
    my_name = answers.get("custom_name")
    route_to_save = os.path.join(mypath, answers.get("custom_name"))
    if answers.get("model_open") == "New␣one":
        route_to_model = os.path.join(mypath, answers.get("custom_name"))
        policy_kwargs = dict(net_arch=dict(pi=[64, 64, 64,64], qf=[128, 128, 128]))
        reset_timesteps = True
        model = SAC(policy_mode, env,
                learning_rate=0.0003,
                learning_starts=20000,
                buffer_size=1000000,
                batch_size=256,
                tau=0.005,
                gamma=0.99,
                tensorboard_log="./tb_log_name/",
                policy_kwargs=policy_kwargs,
                verbose=1,
                seed=0
                ) # train_freq = (1, "episode"))
        print(f"Creating␣fresh␣model␣{route_to_model}")
    else:
```

```
        route_to_model = os.path.join(mypath, answers.get("model_open"))
        reset_timesteps = False
        try:
            model = SAC.load(route_to_model, env)
            print(f"Opened␣model␣in␣with␣{model.num_timesteps}␣transitions␣in␣its␣buffer")
        except:
            print(f"File␣model␣not␣found:␣{route_to_model}")
    # Estimación de episodios que debería tener
    final_steps = int(answers.get("model_workout")) + model.num_timesteps
    while model.num_timesteps < final_steps:
        if reset_timesteps:
            print("No␣need␣to␣reset␣environment\n")
        else:
            model.env.reset()
        try:
            model.learn(int(answers.get("model_workout")), eval_freq=10000, n_eval_episodes=5, eval_env = ↙
                  env, eval_log_path=f"./SAC_eval_logs/{my_name}/", ↙
                  tb_log_name=answers.get("custom_name"), reset_num_timesteps=reset_timesteps)
        except RuntimeError as e:
            print("\n----------------------------\nRuntime␣error:␣{0}\n----------------------------
                 \n".format(e))
            print("Saving␣model␣with␣{0}␣timesteps\n".format(model.num_timesteps))
            reset_timesteps = False
        # save the model
        print(f"Saving␣model␣in␣{route_to_save}")
        model.save(route_to_save)

    # close environment
    env.close()
    print(f"C'est␣fini.␣Total␣timesteps={model.num_timesteps}")


if __name__ == "__main__":
    main()
```

B.4. Librería personalizada

Code/myCosas.py

```
import numpy as np
import gym
import os
from stable_baselines3.common.results_plotter import load_results, ts2xy, plot_results
from stable_baselines3.common.callbacks import BaseCallback


class myEnv(gym.Wrapper):
    _max_steps_episode: int
    _step_counter: int
    _distance: float
    _delta_distance: float

    def __init__(self, env=None, max_steps_episode=500, mode="mode0"):
        super(myEnv, self).__init__(env)
        self._step_counter = 0
        self._max_steps_episode = max_steps_episode
        self._mode = mode
        self._distance = -1
        print("Wrapping␣the␣environment␣in␣a␣homemade␣step␣to␣do␣cosas␣model␣" + mode + "\n")

    def reset(self):
        obs = self.env.reset()
        # OJOAhora el objetivo estará siempre en la misma posición. Para facilitar (aún más) las cosas ↙
              al robot.
        #self.task._task.target.set_position([0.35000000, 0.22000000, 0.97000000])
        obs, _, _, _ = self.env.step(np.array([0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0]))
        try:
            self._distance = distance_cal(obs)
            print("Reset␣environment,␣initial␣distance␣is␣{0}\n".format(self._distance))
```

```
        except:
            print("Something␣went␣wrong␣on␣reset")
        return obs

    def step(self, action):
        self._step_counter += 1
        obs, reward, done, info = self.env.step(action)
        try:
            if done:
                print("Episodio␣resuelto␣en␣{0}␣steps\n".format(self._step_counter))
                self._step_counter = 0
            # mode0 returns -1 if goal is not achieved in max_steps_episode, and 1 if is achieved before.
            if self._mode == "mode0":
                if self._max_steps_episode > 0:
                    if self._step_counter >= self._max_steps_episode:
                        done = True
                        reward = -1
                        self._step_counter = 0
                        print("Reset␣environment,␣took␣too␣much\n")
            # mode1 might or might not have an episode step limit, but returns -1 in each step while goal ↙
                  is not
            # accomplished
            if self._mode == "mode1":
                if not done:
                    reward = -1
                if self._max_steps_episode > 0:
                    if self._step_counter >= self._max_steps_episode:
                        done = True
                        self._step_counter = 0
                        print("Reset␣environment,␣took␣too␣much\n")

            # mode2 might or might not have an episode step limit, but a reward proportional to the delta ↙
                  distance to
            # target. In every step, this reward is decreased. When finished, reward is 1. if doesn't ↙
                  reach the
            # target in the max_step, returns a negative reward tbd.
            if self._mode == "mode2":
                #new_distance = distance_cal(obs)
                #self._delta_distance = new_distance -self._distance
                #self._distance = new_distance
                if not done:
                    # reward = np.tanh(1/(distance+1e-5))
                    #reward = np.exp(-1 * distance_cal(obs))
                    reward = -distance_cal(obs) -1 # Penalización por cada step de más que tarde en ↙
                          resolver el episodio
                if self._max_steps_episode > 0:
                    if self._step_counter >= self._max_steps_episode:
                        done = True
                        self._step_counter = 0
                        print("A␣cascarla,␣ha␣tardado␣mucho\n")
            if self._mode == "mode3": # Devuelve un reward arbitrario e inmenso = 500 si resuelve el ↙
                  episodio
                if not done:
                    reward = -distance_cal(obs) -1 # Penalización por cada step de más que tarde en ↙
                          resolver el episodio
                else:
                    reward = 500
                if self._max_steps_episode > 0:
                    if self._step_counter >= self._max_steps_episode:
                        done = True
                        self._step_counter = 0
                        print("A␣cascarla,␣ha␣tardado␣mucho\n")
        except:
            print("Error␣on␣step")
        return obs, reward, done, info


def averaverqueyolovea(env, model, max_steps=1000):
    episode_rewards = [0.0]
    steps = 0

    obs = env.reset()
    done = False
```

```
    while (not done) and (steps < max_steps):
        steps = steps + 1
        action, _states = model.predict(obs)
        obs, reward, done, info = env.step(action)
        episode_rewards[-1] += reward
        env.render()
    return done, episode_rewards


def evaluate_1(env, model, num_steps=1000):
    """
    Evaluate a RL agent
    :param model: (BaseRLModel object) the RL Agent
    :param num_steps: (int) number of timesteps to evaluate it
    :return: (float) Mean reward for the last 100 episodes
    """
    episode_rewards = [0.0]

    obs = env.reset()
    for i in range(num_steps):
        # _states are only useful when using LSTM policies
        action, _states = model.predict(obs)

        obs, reward, done, info = env.step(action)

        # Stats
        episode_rewards[-1] += reward
        if done:
            obs = env.reset()
            episode_rewards.append(0.0)
    # Compute mean reward for the last 100 episodes
    mean_100ep_reward = round(np.mean(episode_rewards[-100:]), 1)
    print("Mean␣reward:", mean_100ep_reward, "Num␣episodes:", len(episode_rewards))

    return mean_100ep_reward


def execfile(filepath, my_globals=None, my_locals=None):
    if my_globals is None:
        my_globals = {}
    my_globals.update({
        "__file__": filepath,
        "__name__": "__main__",
    })
    with open(filepath, 'rb') as file:
        exec(compile(file.read(), filepath, 'exec'), my_globals, my_locals)

def distance_sqrt_cal(obs):
    '''calculate distance between end effector and target'''
    ee_pose = np.array(obs[22:25])
    target_pose = np.array(obs[-3:])

    distance_sqrt = (target_pose[0]-ee_pose[0])**2 + (target_pose[1]-ee_pose[1])**2 + (target_pose[2]-
        ee_pose[2])**2

    # reward = np.tanh(1/(distance+1e-5))
    # reward = np.exp(-1*distance)

    return distance_sqrt

def distance_cal(obs):
    '''calculate distance between end effector and target'''
    ee_pose = np.array(obs[22:25])
    target_pose = np.array(obs[-3:])

    distance = np.sqrt((target_pose[0]-ee_pose[0])**2 +
                    (target_pose[1]-ee_pose[1])**2 + (target_pose[2]-ee_pose[2])**2)

    # reward = np.tanh(1/(distance+1e-5))
    # reward = np.exp(-1*distance)

    return distance
```

B.5. Librería callbacks

Code/myCallbacks.py

```
import os
import numpy as np
from stable_baselines3.common.results_plotter import load_results, ts2xy
from stable_baselines3.common.callbacks import BaseCallback


class SaveOnBestTrainingRewardCallback(BaseCallback):
    """
    Callback for saving a model (the check is done every ``check_freq`` steps)
    based on the training reward (in practice, we recommend using ``EvalCallback``).

    :param check_freq: (int)
    :param log_dir: (str) Path to the folder where the model will be saved.
      It must contains the file created by the ``Monitor`` wrapper.
    :param verbose: (int)
    """
    def __init__(self, check_freq: int, log_dir: str, verbose=1):
        super(SaveOnBestTrainingRewardCallback, self).__init__(verbose)
        self.check_freq = check_freq
        self.log_dir = log_dir
        self.save_path = os.path.join(log_dir, 'best_model')
        self.best_mean_reward = -np.inf

    def _init_callback(self) -> None:
        # Create folder if needed
        if self.save_path is not None:
            os.makedirs(self.save_path, exist_ok=True)

    def _on_step(self) -> bool:
        if self.n_calls % self.check_freq == 0:

            # Retrieve training reward
            x, y = ts2xy(load_results(self.log_dir), 'timesteps')
            if len(x) > 0:
                # Mean training reward over the last 100 episodes
                mean_reward = np.mean(y[-100:])
                if self.verbose > 0:
                    print("Num␣timesteps:␣{}".format(self.num_timesteps))
                    print("Best␣mean␣reward:␣{:.2f}␣-␣Last␣mean␣reward␣per␣episode:␣{:.2f}"
                          .format(self.best_mean_reward, mean_reward))

                # New best model, you could save the agent here
                if mean_reward > self.best_mean_reward:
                    self.best_mean_reward = mean_reward
                    # Example for saving best model
                    if self.verbose > 0:
                        print("Saving␣new␣best␣model␣to␣{}".format(self.save_path))
                        self.model.save(self.save_path)

        return True


class CustomCallback(BaseCallback):
    """
    A custom callback that derives from ``BaseCallback``.

    :param verbose: (int) Verbosity level 0: not output 1: info 2: debug
    """
    self.step = 0
    def __init__(self, verbose=0):
        super(CustomCallback, self).__init__(verbose)
        # Those variables will be accessible in the callback
        # (they are defined in the base class)
        # The RL model
        # self.model = None # type: BaseRLModel
        # An alias for self.model.get_env(), the environment used for training
        # self.training_env = None # type: Union[gym.Env, VecEnv, None]
        # Number of time the callback was called
```

```
        # self.n_calls = 0 # type: int
        # self.num_timesteps = 0 # type: int
        # local and global variables
        # self.locals = None # type: Dict[str, Any]
        # self.globals = None # type: Dict[str, Any]
        # The logger object, used to report things in the terminal
        # self.logger = None # type: logger.Logger
        # # Sometimes, for event callback, it is useful
        # # to have access to the parent object
        # self.parent = None # type: Optional[BaseCallback]

    def _on_training_start(self) -> None:
        """
        This method is called before the first rollout starts.
        """
        print("Training␣start")
        # pass

    def _on_rollout_start(self) -> None:
        """
        A rollout is the collection of environment interaction
        using the current policy.
        This event is triggered before collecting new samples.
        """
        print("Rollout␣start")
        pass

    def _on_step(self) -> bool:
        """
        This method will be called by the model after each call to `env.step()`.

        For child callback (of an `EventCallback`), this will be called
        when the event is triggered.

        :return: (bool) If the callback returns False, training is aborted early.
        """
        return True

    def _on_rollout_end(self) -> None:
        """
        This event is triggered before updating the policy.
        """
        pass

    def _on_training_end(self) -> None:
        """
        This event is triggered before exiting the `learn()` method.
        """
        pass
```

Anexos C

Fundamentos RL

C.1. Proceso de Markov MP o cadena

C.1.1. Propiedad de Markov

Idea El futuro es independiente del pasado, únicamente depende del presente, porque el pasado ya se ha condensado en el presente.

Definición matemática De form amatemática, la probabilidad de un estado futuro depende únicamente del estado presente, siendo irrelevante la cadena de estados que han llevado al estado presente:

$$\mathbb{P}[S_{t+1}|S_t] = \mathbb{P}[S_{t+1}|S_1, ..., S_t] \tag{C.1}$$

Eso significa que el estado captura toda la información relevante que explica su transición al siguiente estado. Puede descartarse por innecesaria la secuencia de estados hasta el estado presente.

Transición entre dos estados Para un estado de markov definido así, la transición entre un estado particular s y un estado s' se expresa como una probabilidad:

$$\mathcal{P}_{s \to s'} = \mathbb{P}[S_{t+1} = s'|S_t = s] \tag{C.2}$$

Matriz de transición de estados Generalizando para n estados, podemos definir una matriz que contenga las probabilidades de transición entre entre cualesquiera de esos n estados:

$$\mathcal{P} = \begin{bmatrix} \mathcal{P}_{1,1} & \cdots & \mathcal{P}_{1,n} \\ \vdots & \ddots & \\ \mathcal{P}_{n,1} & & \mathcal{P}_{n,n} \end{bmatrix} \tag{C.3}$$

C.1.2. Cadena de Markov

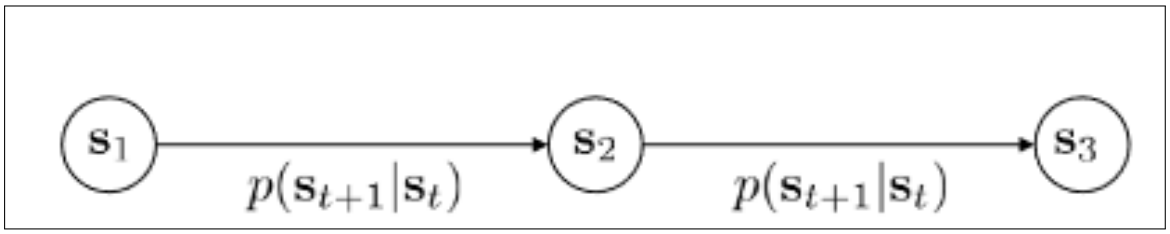


Figura C.1: La probabilidad de transición entre eslabones de una cadena de Markov depende únicamente del estado presente y del estado futuro; nunca de los anteriores[1].

Un proceso de markov consiste en una sucesión de estados $S_1, S_2, ...$ cumplen con la propiedad de Markov, donde la transición entre estados tiene asociada una probabilidad (figura C.1. Así pues, todo proceso de markov se compone de una tupla de estados y probabilidades $\langle \mathcal{S}, \mathcal{P} \rangle$ finitos.

Estacionario En una cadena de Markov estacionaria, los valores de la matriz de probabilidades permanecen fijas a lo largo del problema.

Muestreo Un muestreo es una secuencia concreta de estados que cumplen con la propiedad de Markov. No tiene por qué (de hecho sería un caso extremo) contener todos los estados que componen la candena de Markov, y puede contener un estado en varias posiciones de la cadena.

C.1.3. Elementos de una cadena de Markov

$\mathcal{S}$ Conjunto de estados, pueden ser continuos o discretos

$\mathcal{T}$ Es un operador de transición. Define la distribución de probabilidades de los estados siguientes dado el estado actual $p(s_{t+1}|s_t)$. Se trata de un operador que define a cada transición posible entre estados, condicionada al estado actual, una probabilidad.

Vector de probabilidades Sea $\mu_{t,i}$ La probabilidad de estar en un estado $s_t = i$ en un tiempo t determinado. Así pues, en cada instante t $\boldsymbol{mu}_t$ es un vector de probabilidades de transición a cada uno de los i-ésimos estados.

Relación entre el operador $\mathcal{T}$ y el vector de probabilidades De forma similar, definimos $\mathcal{T}_{t.i} = p(s_{t+1} = i|s_t = j)$ como la probabilidad de pasar de un estado $s_{t+1} = i$

desde un estado actual $s_t = j$. Conocido $\boldsymbol{\mu_t}$, podemos entonces conocer $\boldsymbol{\mu_{t+1}}$ a través del operador $\mathcal{T}$: $\boldsymbol{\mu_{t+1}} = \mathcal{T}\boldsymbol{\mu_t}$

C.2. Cadena de decisiones de Markov

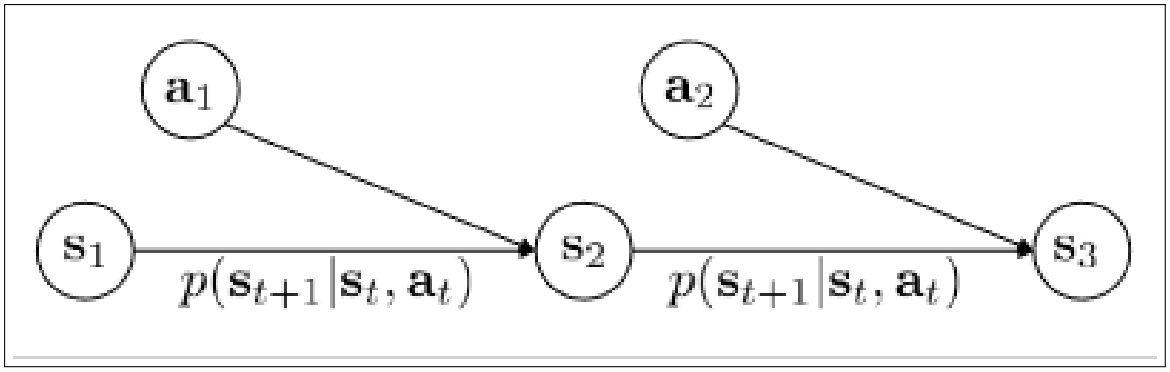


Figura C.2: MDP[1].

La cadena de Markov no entiende de acciones, sino de probabilidades asociadas a cada cambio de estado. Ampliando el concepto a una tupla de estados y transiciones que conforme una cadena de Markov, construimos una cadena de decisiones de Markov como la de la figura C.2:

$$\mathcal{M} = \{\mathcal{S}, \mathcal{A}, \mathcal{T}, \} \tag{C.4}$$

$\mathcal{A}$ Acciones. Como los estados pueden ser continuas o discretas. Ahora, las probabilidades entre estados deben tener en cuenta el vector de acciones $\boldsymbol{a_t}$: $p(s_{t+1}|s_t, a_t$.

Vector probabilidad acciones Ahora, definimos $\xi_{t,k}$ como la probabilidad de tomar una determinada acción k en un instante t, siendo $\boldsymbol{\xi_t}$ el vector de probabilidades de cada posible acción en un momento determinado t.

Tensor de transiciones Como consecuencial, la matriz de transiciones tiene ahora 3 dimensiones: para el i-ésimo estado presente, para la j-sima probabilidad de estado siguiente y para la k-sima probabilidad de acción: $\mathcal{T}_{i,j,k} = p(s_{t+1} = i|s_t = j, a_t = k)$.

En el caso de que el tiempo tienda a infinito. El operador de transiciones $\mathcal{T}$ devuelve

la tupla estad-acción siguiente:

$$\begin{pmatrix} \boldsymbol{s}_{t+1} \\ \boldsymbol{a}_{t+1} \end{pmatrix} = \mathcal{T} \begin{pmatrix} \boldsymbol{s}_t \\ \boldsymbol{a}_t \end{pmatrix} \tag{C.5}$$

$$\begin{pmatrix} \boldsymbol{s}_{t+2} \\ \boldsymbol{a}_{t+2} \end{pmatrix} = \mathcal{T} \begin{pmatrix} \boldsymbol{s}_{t+1} \\ \boldsymbol{a}_{t+1} \end{pmatrix} \tag{C.6}$$

$$\tag{C.7}$$

Extendiendo el concepto:

$$\begin{pmatrix} \boldsymbol{s}_{t+k} \\ \boldsymbol{a}_{t+k} \end{pmatrix} = \mathcal{T}^k \begin{pmatrix} \boldsymbol{s}_t \\ \boldsymbol{a}_t \end{pmatrix} \tag{C.8}$$

Sin en algún momento la cadena de probabilidades converge a una distribución estacionaria, la probabilidad de un de estados permanecerá igual:

$$\mu = \mathcal{T}\mu \tag{C.9}$$

$$(\mathcal{T} - \boldsymbol{I})\mu = 0; \tag{C.10}$$

La solución a la expresión anterior existe bajo una serie de condiciones, y la distribución estacionaria $\mu = p_\theta(\boldsymbol{s}, \boldsymbol{a})$.

Función de recompensa La función de recompensa a largo plazo es un escalar producto de la multiplicación del espacio de estados por el espacio de acciones.

C.3. Proceso de recompensas de Markov MRP

Sin recompensa, no se puede hablar de RL.

Un proceso de recompensas de Markov consiste en una cadena de Markov a la que se le añade una función de recompensa $\mathcal{R}$, y un factor de descuento comprendido entre 0 y 1 $\gamma \in [0, 1]$: $\langle \mathcal{S}, \mathcal{P}, \mathcal{R}, \gamma \rangle$.

C.3.1. Función de recompensa

La función de recompensa R_s, devuelve un determinado valor de recompensa futuro para un estado particular presente $S_t = s$:

$$R_s = \mathbb{E}[R_{t+1} | S_t = s] \tag{C.11}$$

C.3.2. Función de retorno

La función de retorno G_t es una fucnión que acumula las recompensas obtenidas desde el instante t hasta el final de la secuencia de estados.

Es esta función la que devolverá un valor realmente útil para evaluar un muestreo de estados de Markov. Podría darse el caso de que un subconjunto de estados devuelvan una recompensa pésima, pero conduzan a un estado que devuelva una gran recompoensa. Esta es la función que tendrá en cuenta el balance final.

Tasa de descuento γ es una tasa de descuento que prioriza la inmediatez de la recompensa.

$$G_t = R_{t+1} + \gamma R_{t+2} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{C.12}$$

myopic Un valor de γ cercano a 0 prioriza la recompensa inmediata, despreciando la recompoensa a largo plazo.

far-sighted Un valor de γ cercano a 1 pondera las recompensas futuras tanto (= 1) o casi tanto como la recompensa inmediata.

Redundancia Aplicar una tasa de descuento es una forma práctica de evitar una recompensa infinita en un proceso cíclico de Markov.

Incertidumbre De forma similar a la tasa de interés sobre el valor futuro, la incertidumbre aumenta cuando las recompensas están asociadas a estados alejados del estado actual, por ello tiene sentido aplicar una cierta tasa de descuento que compense esta incertidumbre.

Comportamiento biológico Priorizar las recompensas inmediatas es un comportamiento que también sigue la conducta de animales y humanos.

Proceso finito Si sabemos con seguridad que todo muestreo de la cadena de markov acaba en un número finito de estados, puede omitirse la tasa de descuento $\gamma = 1$.

C.3.3. Conjunto de acciones

Dado un estado determinado s, se define como política a la distribución sobre las acciones que pueden tomarse en ese estado. La formulación de la probabilidad asociada

a una acción a en un estado s se expresa de forma estocástica como:

$$\pi(a|s) = \mathbb{P}[A_t = a, S_t = s] \tag{C.13}$$

Esta distribución ofrece un mapa para cada estado, asociando una probabilidad a cada una de las posibles acciones.

Al tratarse de estados de una cadena de Markov, dichas probabilidades dependen sólo del estado, y no del tiempo t.

Una cadena de estados de Markov $S1, S2, ...$ es el resultado de una política determinada.

Como las recompensas están asociadas a los estados en una cadena de Markov, la política también determina un proceso de recompensa de Markov.

C.4. Proceso de Markov parcialmente observable

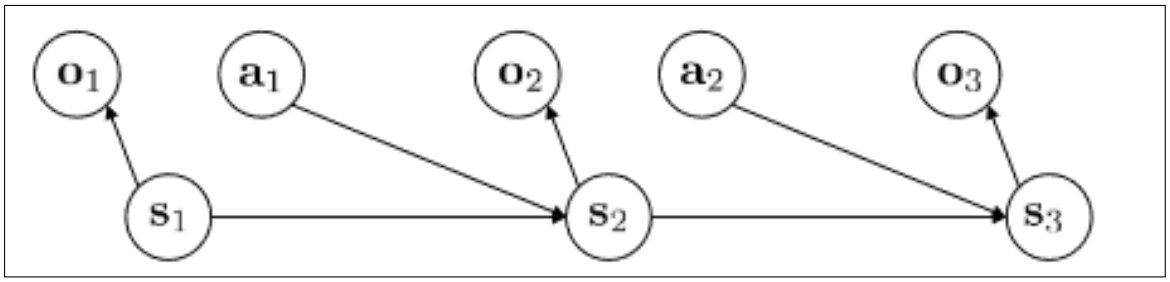


Figura C.3: MDP parcialmente observable[1].

Cuando el espacio de estados no es totlamente observable, podemos construir un proceso de markov parcialmente observable añadiendo algunos componentes, como en la figura C.3.

$\mathcal{O}$ Espacio de boservaciones. Puede ser discreto o continuo, es una abstracción del espacio de estados.

Probabilidad de emisión Relaciona cada observación o_t con su estado s_t a través de una probabilidad.

Anexos D

Policy Gradient Algorithm

El **Policy Gradient Algorithm** es uno de los primeros algoritmo de descenso o ascenso para RL. Se describe su funcionamiento porque se enfrenta a problemas reales a la hora de plantear un problema de RL.

D.1. Planteamiento

Una política puede formularse expresamente, o a través de un modelo de funcionamiento. Cuando hablamos de una política π_θ, θ representa los parámetros de esa política, que pueden ser los pesos de una red neuronal, o los coeficientes que multiplican un problema de regresión lineal.

Trayectoria Secuencia de acciones. La distribución estadística de una trayectoria asociada a la política π_theta se expresa por conveniencia como $p_\theta(\tau)$:

$$p_\theta(\boldsymbol{s}_1, \boldsymbol{a}_1, ..., \boldsymbol{s}_t, \boldsymbol{a}_t) = p_\theta(\tau) \tag{D.1}$$

La distribución de transiciones de una trayectoria descrita por la política π_theta se expresa de forma matemática como el producto del estado inicial, que asumismos desconocido por la la distribución de la transición ¿qué cojones? que también es desconocida por y la política, que es conocida, de todos los instantes desde t=1 a T.

D.1.1. Meta

La meta consiste maximizar la recompensa acumulada en todas las acciones escogiendo π_θ.

¿Cómo evaluar este objetivo? Si no conocemos el estado inicial o la distribución de transiciones, sólo tenemos una política y un entorno. Una forma es general muestras

de la distribución $pi_\theta(\tau)$ bajo la cual se toma la esperanza (y promediar los valores de la cantidad dentro de la esperanza de esas muestras.

D.1.2. Muestreo

Muestreo de $\pi_\theta(\tau)$ La forma de obtener muestras de $\pi_\theta(\tau)$ es utilizar una política inicial para actura sobre el entorno

Función de recompensa A efectos de notación, es conveniente escribir la esperanza de recompensa bajo una distribución de trayectorias $p_\theta(\tau)$ como:

$$J(\theta) = E_{\tau \sim p_\theta(\tau)[\sum_t r(\boldsymbol{s}_t, \boldsymbol{a}_t)]} \tag{D.2}$$

Estimación función de recompensa Cuando se estima evaluando esa política N veces, se puede aproximar la función $J(\theta)$ por:

$$J(\theta) \simeq \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} r(\boldsymbol{s}_{i,t}, \boldsymbol{a}_{i,t}) \tag{D.3}$$

Este muestreo se hace bajo una misma política, y es una estimación. ¿Cómo maximizar este valor? El objetivo es maximizar la función valor modificando la composición θ de la política π_θ.

Función J como integral Hacemos un cambio de notación para expresar $J(\theta)$ en función de la política π_θ, y como la integral de la probabilidad de una trayectoria por la recompensa obtenida por esa trayectoria:

$$J(\theta) = E_{\tau \sim \pi_\theta(\tau)}[r(\tau)] = \int \pi_\theta(\tau) r(\tau) d\tau \tag{D.4}$$

Gradiente de J A partir de la definición anterior, la definición de gradiente es inmediata:

$$\nabla_\theta J(\theta) = \int \nabla_\theta \pi_\theta(\tau) r(\tau) d\tau \tag{D.5}$$

La dificultad en la expresión anterior consiste en obtener una expresión par $\pi_t heta(\tau)$, ya que la distribución inicial de trayectorias es desconocida.

Derivada de un logaritmo Aplicando la derivada del logaritmo, obtenemos la siguiente identidad:

$$\pi_\theta(\tau) \nabla_\theta \log \pi_\theta(\tau) = \pi_\theta(\tau) \frac{\nabla_\theta \pi_\theta(\tau)}{\pi_\theta(\tau)} = \nabla_\theta \pi_\theta(\tau) \tag{D.6}$$

Ahora podemos expresar el gradiente de la función J de forma más conveniente, como una esperanza:

$$\nabla_\theta J(\theta) = \int \pi_\theta(\tau) \nabla_\theta \log \pi_\theta(\tau) r(\tau) d\tau \tag{D.7}$$

$$\nabla_\theta J(\theta) = E_{\tau \sim \pi_\theta(\tau)} \left[\nabla_\theta \log \pi_\theta(\tau) r(\tau) \right] \tag{D.8}$$

Volviendo a la definición original de una distribución de trayectorias bajo una política π_θ, tomando el algoritmo de ambos lados convertimos el proudcto en una suma de logaritmos::

$$\pi_\theta(\tau) = \pi_\theta(\boldsymbol{s}_1, \boldsymbol{a}_1, ..., \boldsymbol{s}_t, \boldsymbol{a}_t) = p(\boldsymbol{s}_1) \prod_{t=1}^{T} \pi_\theta(\boldsymbol{a}_t|\boldsymbol{s}_t) p(\boldsymbol{s}_{t+1}|\boldsymbol{s}_t, \boldsymbol{a}_t) \tag{D.9}$$

$$\log \pi_\theta(\tau) = \log p(\boldsymbol{s}_1) + \sum_{t=1}^{T} (\log \pi_\theta(\boldsymbol{a}_t|\boldsymbol{s}_t) + \log p(\boldsymbol{s}_{t+1}|\boldsymbol{s}_t, \boldsymbol{a}_t)) \tag{D.10}$$

Dentro de la expresión D.10, existen 3 términos: el logaritmo de la distribución inicial del primer estado, el logaritmo de las probabilidades de transición, y el logaritmo de las probabilidades de la política... literalmente. te has quedado a gusto, hijo de puta.

Hay que tomar el gradiente de la expresión D.10. Como la distribución de probabilidades del estado inicial no varía, su gradiente es 0. La distribución de probabilidades de transición entre estados, condicionada a una acción, no está determinada por la política (sino por la acción). La acción cambia al cambiar la política, pero la probabilidad de cabio de estado condicionado a la acción es fija y depende del modelo. Por lo tanto:

$$\nabla_\theta \left[\log p(\boldsymbol{s}_1) + \sum_{t=1}^{T} (\log \pi_\theta(\boldsymbol{a}_t|\boldsymbol{s}_t) + \log p(\boldsymbol{s}_{t+1}|\boldsymbol{s}_t, \boldsymbol{a}_t)) \right] = \sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(\boldsymbol{a}_t|\boldsymbol{s}_t) \tag{D.11}$$

Sustituyendo obtenemos una expresión del gradiente muy conveniente

$$\nabla_\theta J(\theta) = E_{\tau \sim \pi_\theta(\tau)} \left[\left(\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(\boldsymbol{a}_t|\boldsymbol{s}_t) \right) \left(\sum_{t=1}^{T} r(\boldsymbol{s}_t, \boldsymbol{a}_t) \right) \right] \tag{D.12}$$

Para obtener $\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(\boldsymbol{a}_t|\boldsymbol{s}_t)$, utilizamos el proceso de muestreo. A la vez que la recompensa, se suma el valor de $\nabla_t heta \log \pi_\theta$ y se promedia según el número de iteraciones:

$$\nabla_\theta J(\theta) \simeq \frac{1}{N} \sum_{i=1}^{N} \left(\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(\boldsymbol{a}_{i,t}|\boldsymbol{s}_{i,t}) \right) \left(\sum_{t=1}^{T} r(\boldsymbol{s}_{i,t}, \boldsymbol{a}_{i,t}) \right) \tag{D.13}$$

Una vez estimado este valor $\nabla_\theta J(\tau)$, actualizamos los pesos de θ:

$$\theta \leftarrow \theta + \alpha \nabla_\theta J(\tau) \tag{D.14}$$

D.1.3. Observabilidad parcial

En un entorno de observabilidad parcial, donde no podemos obtener una representación completa del estado a partir de una observación, la aproximación del gradiente de la función J debe formularse como:

$$\nabla_\theta J(\theta) \simeq \frac{1}{N}\sum_{i=1}^{N}\left(\sum_{t=1}^{T}\nabla_\theta \log \pi_\theta(\boldsymbol{a}_{i,t}|\boldsymbol{o}_{i,t})\right)\left(\sum_{t=1}^{T} r(\boldsymbol{s}_{i,t},\boldsymbol{a}_{i,t})\right) \tag{D.15}$$

Además, cualquier sistema parcialmente observable puede formar parte de un proceso de toma de decisiones de markov si asimilamos la el estado a la observación parcial recibida.

Ojo, la función de recompensa está asociada a un estado. Podemos asumir que el estado determinado es inferencialbe por una observación, y utilizar la misma función de recompensa.

D.2. Algoritmo

El algoritmo de RL recorre la siguiente secuencia, al menos de forma teórica.

- Muestrear una trayectoria utilizando una política $\pi(\boldsymbol{a}_t|\boldsymbol{s}_t$ cualquiera.
- Estimar el valor de $\nabla_\theta J(\theta)$ utilizando los resultados del muestreo en la expresión D.13.
- Actualizar la política utilizando la expresión D.14

D.2.1. Problemas en el algoritmo

En realidad, el algoritmo de las narices no funciona.

Imaginando un coche autónomo, $\pi_\theta(\boldsymbol{a}_t|\boldsymbol{s}_t)$ podría ser una red neuronal que toma como entrada las observaciones del entorno $\boldsymbol{s}_t$, y devuelve una acción $\boldsymbol{a}_t$.

El objetivo de un aprendizaje supervisado sería que $\pi_\theta(\boldsymbol{a}_t|\boldsymbol{s}_t)$ imitase el comportamiento de un conductor para obtener la mayor función de coste.

D.2.2. Ensayo mediante prueba y error

Formaliza de forma matemática la técnica de "prueba y error".

Sea τ_i la trayectoria obtenida en el muestreo i-ésimo. Lo que garantiza este algoritmo es que esta trayectoria tenga más probabilidad de ocurrir, cuanto más alto sea el gradiente de la función J asociado a este muestreo.

D.2.3. Causalidad

Dependencia de r Este algoritmo depende fuertemente de $r(\tau)$. Es susceptible a un cambio de escala o nivel en la forma en que se calcula r.

D.3. Reducir la varianza de r

La varianza de una variable disminuye en la misma proporción que la variable si se escala por un determinado factor. Aplicado a una escala temporal, donde la recompensa tiene más valor al principio que al final, formaliza que la política en un tiempo $t' > t$ no afecte la recompensa en t Una forma de reducir la varianza de r

Reformulando la expresión D.13, garantizamos que las recompensas asociadas a la política *pi* existente en un momento determinado sólo afectan a la probabilidad de esa política:

$$\nabla_\theta J(\theta) \simeq \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(\boldsymbol{a}_{i,t}|\boldsymbol{s}_{i,t}) \left(\sum_{t'=t}^{T} r(\boldsymbol{s}_{i,t}, \boldsymbol{a}_{i,t}) \right) \tag{D.16}$$

Sigue siendo un estimador válido del gradiente $\nabla_\theta J(\tau)$.

De forma pragmática, se está multiplicando las distribuciones $\nabla_\theta \log \pi_\theta(\boldsymbol{a}_{i,t}|\boldsymbol{s}_{i,t})$ por valores más pequeños conforme avanza t, reduciendo así la varianza de r.

Notación Q Utilizando notación Q para la recompensa o función valor, esta forma de disminuir su peso con el paso del tiempo se denota por $\hat{Q}_{i,t}$:

$$\nabla_\theta J(\theta) \simeq \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(\boldsymbol{a}_{i,t}|\boldsymbol{s}_{i,t}) \hat{Q}_{i,t} \tag{D.17}$$

D.3.1. Ajuste de la recompensa

El ajuste de las función de recompensa también afecta enormemente al algoritmo. Imaginemos que la función de recompensa devuelve valores del orden de mangitud de $10^6 \pm 1$.

La solución es ajustar las magnitudes de estas funciones respecto a la media:

$$\nabla_\theta J(\theta) \simeq \sum_{i=1}^{N} \nabla_\theta \log \pi_\theta(\tau) \left[r(\tau) - b \right] \tag{D.18}$$

Invariabilidad Puede demostrarse la invariabilidad de la esperanza acudiendo a la formulación original. Supongamos la expresión D.8, referida únicamente al gradiente de la distribución π_θ de una trayectoria τ multiplicado por un escalar b

$$E\left[\nabla_\theta \log \pi_\theta(\tau) b\right] = \int \pi_\theta(\tau) \nabla_\theta \log \pi_\theta(\tau) b d\tau \tag{D.19}$$

Utilizando la expresión de la derivada del logaritmo, y conmutando la integral por el operador gradiente:

$$\pi_\theta(\tau) \nabla_\theta \log \pi_\theta(\tau) = \nabla_\theta \pi_\theta(\tau) \tag{D.20}$$

$$E\left[\nabla_\theta \log \pi_\theta(\tau) b\right] = \int \nabla_\theta \pi_\theta(\tau) b d\tau \tag{D.21}$$

$$= b \nabla_\theta \int \pi_\theta(\tau) d\tau \tag{D.22}$$

La integral de una distribución de probabilidad es 1, y el gradiente de una constate es 0:

$$E\left[\nabla_\theta \log \pi_\theta(\tau) b\right] = 0 \tag{D.23}$$

D.3.2. Solución no óptima

Ajustar en base a la media no es el mejor valor, pero es siempre una opción buena (mejor que no ajustar) y razonable.

D.3.3. Solución óptima

Varianza Aunque no afecte a la esperanza, sí afecta a la varianza. La ecuación de la varianza puede escribirse como:

$$Var[x] = E[x^2] - E[x]^2 \tag{D.24}$$

$$= E\left[\left(\nabla_\theta \log \pi_\theta(\tau) \left[r(\tau) - b\right]\right)^2\right] - E\left[\nabla_\theta \log \pi_\theta(\tau) \left[r(\tau) - b\right]\right]^2 \tag{D.25}$$

La esperanza no varía por añadir sumar un factor b:

$$Var[x] = E\left[\left(\nabla_\theta \log \pi_\theta(\tau) \left[r(\tau) - b\right]\right)^2\right] - E\left[\nabla_\theta \log \pi_\theta(\tau) r(\tau)\right]^2 \tag{D.26}$$

Derivando respecto a b:

$$\frac{\partial Var}{\partial b} = \frac{\partial}{\partial b} E\left[\left(\nabla_\theta \log \pi_\theta(\tau) \left[r(\tau) - b\right]\right)^2\right] \tag{D.27}$$

$$= \frac{\partial}{\partial b} E\left[\left(\nabla_\theta \log \pi_\theta(\tau)\right)^2 r(\tau)^2 - 2b \left(\nabla_\theta \log \pi_\theta(\tau)\right)^2 r(\tau) + \left(\nabla_\theta \log \pi_\theta(\tau)\right)^2 b^2\right] \tag{D.28}$$

$$= \frac{\partial}{\partial b} E\left[-2 \left(\nabla_\theta \log \pi_\theta(\tau)\right)^2 r(\tau) + 2 \left(\nabla_\theta \log \pi_\theta(\tau)\right)^2 b\right] \tag{D.29}$$

Existe un valor b para el cual la derivada de la varianza es 0:

$$b = \frac{E\left[(\nabla_\theta \log \pi_\theta(\tau))^2 r(\tau)\right]}{E\left[(\nabla_\theta \log \pi_\theta(\tau))^2\right]} \tag{D.30}$$

La expresión anterior no pondera las recomensas entre sí, sino que las pondera por un gradiente. Si el gradiente es un vector, el parámetro b también es multidimensional, y ofrece un ajuste en cada dimensión del gradiente que conforma la esperanza.

D.4. Policy gradient is on-policy

Cada vez que se cambian los parámetros θ del gradiente, hay que volver a muestrear por completo, ya que la esperanza está condicionada a la política anterior $E_{\tau\sim\pi_\theta(\tau)}$:

$$\nabla_\theta J(\theta) = E_{\tau\sim\pi_\theta(\tau)}\left[\nabla_\theta \log \pi_\theta(\tau) r(\tau)\right] \tag{D.31}$$

Ineficiencia Normalmente las redes neuronales sólo cambian ligeramente con cada cambio, y volver a muestrear es caro en términos de continuidad. Por eso este tipo de algoritmos son ineficientes.

D.4.1. off-policy policy gradient

Si no se muestrea $\pi_\theta(\tau)$, sino otra distribución $\bar{\pi}(\tau)$. Para ello, se hace depender la esperanza de una nueva distribución de probabilidades. De forma analítica:

$$E_{x\sim p(x)}[f(x)] = \int p(x) f(x) dx \tag{D.32}$$

$$= \int \frac{q(x)}{q(x)} p(x) f(x) dx \tag{D.33}$$

$$= \int q(x) \frac{p(x)}{q(x)} f(x) dx \tag{D.34}$$

$$= E_{x\sim q(x)}\left[\frac{p(x)}{q(x)} f(x)\right] \tag{D.35}$$

$$J(\theta) = E_{\tau\sim\bar{\pi}(\tau)}\left[\frac{\pi_\theta(\tau)}{\bar{\pi}_\theta(tau)} r(\tau)\right] \tag{D.36}$$

Ahora hay que escoger $\bar{\pi}(\tau)$ para que el cociente $\frac{\pi_\theta(\tau)}{\bar{\pi}(\tau)}$ pueda cancelar los siguientes términos:

$$\frac{\pi_\theta(\tau)}{\bar{\pi}(\tau)} = \frac{\cancel{p(\boldsymbol{s}_1)} \prod_{t=1}^{T} \pi_\theta(\boldsymbol{a}_t|\boldsymbol{s}_t) \cancel{p(\boldsymbol{s}_{t+1}|\boldsymbol{s}_t, \boldsymbol{a}_t)}}{\cancel{p(\boldsymbol{s}_1)} \prod_{t=1}^{T} \bar{\pi}(\boldsymbol{a}_t|\boldsymbol{s}_t) \cancel{p(\boldsymbol{s}_{t+1}|\boldsymbol{s}_t, \boldsymbol{a}_t)}} \tag{D.37}$$

No sé qué dice, ahora dice que $\bar{\pi}(\tau)$ es la política anterior, mientras que $\pi_\theta(\tau)$ es la nueva política. Creo que ha empezado justo al revés

Nuevo salto de rana, es posible derivar el gradiente de política (ya me he perdido) con importance sampling, que es lo anterior.

¿Cómo podemos estimar unos nuevos parámeotros θ' sacados de la manga? Tomamos nuestro .ºbjective value"(la función J, función valor, recompensa acumulada... el hideputa nombra las cosas como le sale de los cojones) en función de θ':

$$J(\theta') = E_{\tau\sim\pi_\theta(\tau)}\left[\frac{\pi_{\theta'}(\tau)}{\pi_\theta(\tau)} r(\tau)\right] \tag{D.38}$$

La ventaja de la expresión anterior es que θ' sólo aparece de forma explícita en el cociente, y no en la esperanza. De forma similar, en la expresión del gradiente:

$$\nabla_{\theta'} J(\theta') = E_{\tau\sim\pi_\theta(\tau)}\left[\frac{\nabla_{\theta'}\pi_{\theta'}(\tau)}{\pi_\theta(\tau)} r(\tau)\right] \tag{D.39}$$

$$= E_{\tau\sim\pi_\theta(\tau)}\left[\frac{\pi_{\theta'}(\tau)}{\pi_\theta(\tau)} \nabla_{\theta'}\log\pi_{\theta'}(\tau) r(\tau)\right] \tag{D.40}$$

$$= E_{\tau\sim\pi_\theta(\tau)}\left[\left(\prod_{t=1}^{T}\frac{\pi_{\theta'}(\boldsymbol{s}_t|\boldsymbol{a}_t)}{\pi_\theta(\boldsymbol{s}_t|\boldsymbol{a}_t)}\right)\left(\sum_{t=1}^{T}\nabla_{\theta'}\log\pi_{\theta'}(\boldsymbol{s}_t|\boldsymbol{a}_t)\right)\left(\sum_{t=1}^{T} r(\boldsymbol{s}_t,\boldsymbol{a}_t)\right)\right] \tag{D.41}$$

Explotando de nuevo la causalidad, de forma que las recompensas pasadas no influyan en las políticas pasadas:

$$\nabla_{\theta'} J(\theta') = E_{\tau\sim\pi_\theta(\tau)}\left[\left(\sum_{t=1}^{T}\nabla_{\theta'}\log\pi_{\theta'}(\boldsymbol{s}_t|\boldsymbol{a}_t)\right)\left(\prod_{t'=1}^{t}\frac{\pi_{\theta'}(\boldsymbol{s}_{t'}|\boldsymbol{a}_{t'})}{\pi_\theta(\boldsymbol{s}_{t'}|\boldsymbol{a}_{t'})}\right)\left(\sum_{t'=t}^{T} r(\boldsymbol{s}_{t'},\boldsymbol{a}_{t'})\left(\prod_{t''=t}^{t'}\frac{\pi_{\theta'}(\boldsymbol{s}_{t''}|\boldsymbol{a}_{t''})}{\pi_\theta(\boldsymbol{s}_{t''}|\boldsymbol{a}_{t''})}\right)\right)\right] \tag{D.42}$$

Para simplificar el algoritmo, elimimos (igualando a 1el último productorio $\prod_{t''=t}^{t'}\frac{\pi_{\theta'}(\boldsymbol{s}_{t''}|\boldsymbol{a}_{t''})}{\pi_\theta(\boldsymbol{s}_{t''}|\boldsymbol{a}_{t''})}$). El algoritmo sigue convergiendo a una mejora de la política:

$$\nabla_{\theta'} J(\theta') = E_{\tau\sim\pi_\theta(\tau)}\left[\left(\sum_{t=1}^{T}\nabla_{\theta'}\log\pi_{\theta'}(\boldsymbol{s}_t|\boldsymbol{a}_t)\right)\left(\prod_{t'=1}^{t}\frac{\pi_{\theta'}(\boldsymbol{s}_{t'}|\boldsymbol{a}_{t'})}{\pi_\theta(\boldsymbol{s}_{t'}|\boldsymbol{a}_{t'})}\right)\left(\sum_{t'=t}^{T} r(\boldsymbol{s}_{t'},\boldsymbol{a}_{t'})\right)\right] \tag{D.43}$$

Si T es muy grande, tanto numerador como denominador en el gradiente anterior tienden a 0, aumentando la varianza y la incertidumbre.

El muestreo original *on-policy* aproximaba la política $\pi_\theta(\boldsymbol{s}_t,\boldsymbol{a}_t)$ mediante una tupla $(\boldsymbol{s}_{i,t},\boldsymbol{a}_{i,t})$ (en realidad no tiene nada que ver para llegar a la expresión siguiente, pero está en la diapositiva):

$$\nabla_\theta J(\theta) \simeq \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T}\nabla_\theta\log\pi_\theta(\boldsymbol{a}_{i,t}|\boldsymbol{s}_{i,t})\hat{Q}_{i,t} \tag{D.44}$$

De forma similar, en el muestreo *off-policy*, y utilizando al regla de la cadena [1]:

$$\nabla'_\theta J(\theta') \simeq \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T}\frac{\pi_{\theta'}(\boldsymbol{s}_{i,t},\boldsymbol{a}_{i,t})}{\pi_\theta(\boldsymbol{s}_{i,t},\boldsymbol{a}_{i,t})}\nabla_{\theta'}\log\pi_{\theta'}(\boldsymbol{a}_{i,t}|\boldsymbol{s}_{i,t})\hat{Q}_{i,t} \tag{D.45}$$

$$\simeq \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T}\frac{\pi_{\theta'}(\boldsymbol{s}_{i,t})}{\pi_\theta(\boldsymbol{s}_{i,t})}\frac{\pi_{\theta'}(\boldsymbol{s}_{i,t}|\boldsymbol{a}_{i,t})}{\pi_\theta(\boldsymbol{s}_{i,t|\boldsymbol{a}_{i,t}})}\nabla_{\theta'}\log\pi_{\theta'}(\boldsymbol{a}_{i,t}|\boldsymbol{s}_{i,t})\hat{Q}_{i,t} \tag{D.46}$$

Haciendo un acto de fe (así lo ha dicho), ignoramos el siguiente término sabiendo que el error cometido será pequeño.

$$\nabla'_\theta J(\theta') \simeq \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T}\frac{\pi_{\theta'}(\boldsymbol{s}_{i,t})}{\pi_\theta(\boldsymbol{s}_{i,t})}\frac{\pi_{\theta'}(\boldsymbol{s}_{i,t}|\boldsymbol{a}_{i,t})}{\pi_\theta(\boldsymbol{s}_{i,t|\boldsymbol{a}_{i,t}})}\nabla_{\theta'}\log\pi_{\theta'}(\boldsymbol{a}_{i,t}|\boldsymbol{s}_{i,t})\hat{Q}_{i,t} \tag{D.47}$$

$$\nabla_{\theta'} J(\theta') \simeq \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T}\frac{\pi_{\theta'}(\boldsymbol{s}_{i,t}|\boldsymbol{a}_{i,t})}{\pi_\theta(\boldsymbol{s}_{i,t|\boldsymbol{a}_{i,t}})}\nabla_{\theta'}\log\pi_{\theta'}(\boldsymbol{a}_{i,t}|\boldsymbol{s}_{i,t})\hat{Q}_{i,t} \tag{D.48}$$

D.5. Policy gradient with automatic differentiation

Es ineficiente calcular de forma explícita el gradiente del logaritmo de un espacio probabilístico π de acciones $a_{i,t}$ condicionadas a un estado $s_{i,t}$

Se implementa "pseudo loss"de likelihood de funcion de coste... vete a la mierda un rato

Algo así como:

$$\nabla_\theta J_{ML}(\theta = \simeq \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T}\nabla_\theta\log\pi_\theta(\boldsymbol{a}_{i,t}|\boldsymbol{s}_{i,t}) \tag{D.49}$$

y entonces:

$$J_{ML}(\theta) \simeq \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T}\log\pi_\theta(\boldsymbol{a}_{i,t}|\boldsymbol{s}_{i,t}) \tag{D.50}$$

¿Para qué? Calcula el gradiente por ti:

$$J(\theta) \simeq \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T}\log\pi_\theta(\boldsymbol{a}_{i,t}|\boldsymbol{s}_{i,t})\hat{Q}_{i,t} \tag{D.51}$$

A $\log\pi_\theta(\boldsymbol{a}_{i,t}|\boldsymbol{s}_{i,t})$ le llama cross entropy (discrete) or squared error (Gaussian)

[1] Regla de la cadena de la probabilidad condicional: $\mathbb{P}(x^{(1)},...,x^{(n)}) = \mathbb{P}(x^{(1)})\prod_{i=2}^{n}\mathbb{P}(x^{(i)}|x^{(1)},...,x^{(i-1)})$

D.5.1. Pseudocódigo

Dada una tupla de NxT Da acciones y NxT Ds estados, y una política cualquiera que devuelve acciones en función de estados.

Algoritmo D.1: A2C

```
Se utiliza la cross entropy mencionada a salto de mata para evaluar el coste, utilizando
un tensor de acciones predichas, y un tensor de acciones muestreadas.
lofits \gets policy.predictions(states) # This should return (NxT) x Da tensor of actions
negative_likelihoods ← tf.nn.softmax_cross_entropy_with_logits(labels = actions
, logits = logits)
loss ← tf.reduce\_mean(negative\_likelihoods)
Se utiliza el resultado anterior para obtener
una serie de escalares necesarios para el paso siguiente. Ademas se normaliza el resultado
respecto a la media de la funcion de coste (creo)
Se calculan las variables en funcion del resultado anterior y una variable.
gradients ← loss.gradients(loss, variables)

Se introduce ahora un tensor de NxT valores. Es un tensor de recompensas que tiene en cuenta
unicamente la recompensa acumulada desde el final de la trayectoria hasta el momento actual
es el componente Q̂_{i,t} de la funcion de coste.

weighted\_negative\_likelihoods ← tf.multiply(negative\_likelihoods, q\_values)
loss ← tf.reduce_mean(weighted\_negative\_likelihoods)
```

A la hora de calcular el tensor de $q - values$, hay que tener en cuenta el ajuste de los datos.

Anexos E

Términos *Reinforcement Learning*

Experiencia El conjunto del estado, la acción, la recompensa y el nuevo estado en cada time step se llama experiencia y, como veremos, estos procesos forman una tupla que será usada para el aprendizaje del agente. Cada una de las interacciones del agente con el entorno se llama **experiencia**, y se representa por: (s_t, a_t, r_t, s_{t+1}).

Episodio La tarea que el agente está tratando de resolver puede tener o no tener un final natural. Las tareas que tienen un final natural, como una competición contra otro agente, se llaman tareas episódicas. Por el contrario, las tareas que no lo tienen se denominan tareas continuas; por ejemplo, aprender el movimiento hacia adelante de un robot. Todas las interacciones desde el episodio inicial $t = 0$ hasta el episodio final se expresan como: $t = n$: $(s_0, a_0, r_0, s_1, a_1, r_1, ..., s_n$.

Trayectoria Cualquier secuencia consecutiva de episodios. Es un concepto más flexible que el de **episodio**.

Política determinista La política que determina el comportamiento del agente con el entorno puede ser determinista o estocástica. Una política determinista ofrece al agente una única acción posible en un determinado estado s. Es decir, se define mediante una función que mapea cada uno de los estados a una acción en particular.

Política estocástica Sin embargo, el tipo más común de política se define de manera estocástica; es decir, a cada estado s le corresponde una distribución de probabilidad sobre las acciones que el agente puede realizar.

Política óptima Salvo para problemas muy sencillos, no tiene sentido plantear una política inamovible. El agente modifica la política al interaccionar con el entorno; este

es el objetivo de un problema RL: encontrar una **política óptima**. A grandes rasgos, la literatura del tema divide los métodos de optimización de políticas en dos categorías: **value-based** y **policy-based**.

Función de transición Cada estado tiene asociada una **función de transición** que relaciona el conjunto de acciones con el siguiente estado posible. Es una función del entorno que puede ser conocida *model-based* o no *model-free* por el agente.

model-based Si el agente conoce el modelo del entorno lo suficientemente bien como para avanzar el resultado de sus acciones, o al menos de las variables del entorno susceptibles de ser modificadas por sus aciones, decimos que es un aprendizaje por refuerzo basado en modelo. En estos casos, puede encontrarse la solución al problema mediante programación dinámica[40].

model-free Cuando el agente no conoce el modelo del entorno, o este conocimeinto el incompleto, debe tomar decisiones a ciegas. El agente puede intentar aprender el modelo explícitamente como parte del algoritmo, y asimilarse a un problema *model-based*, o puede aprender únicamente de su experiencia sin construir un modelo den entorno. En este caso lo llamamos aprendizaje por refuerzo sin modelo.

Anexos F

Cinemática directa brazo robótico Panda

Un método práctico para determinar la cinemática directa consiste en seguir la metodología *Denavit-Hartenberg*[1], se asignan los sistemas de coordenadas al brazo robótico siguiendo las convenciones D-H, tal como se muestra en la figura F.1, extraida de [35].

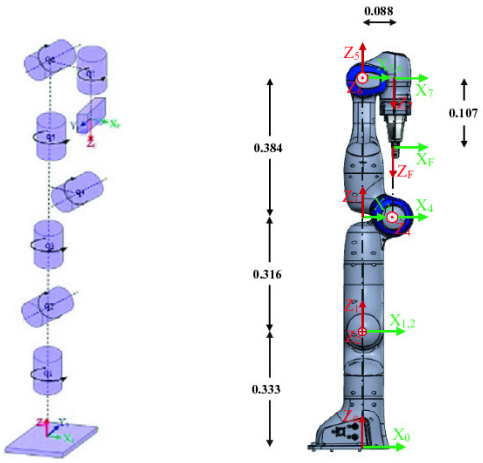


Figura F.1: Parámetros D-H del robot Franka Emika Panda D-H[35].

El sistema descrito tiene cuatro siete juntas y 8 sistemas de coordenadas. El ángulo de torsión α_i, la longitud de cada miembro a_i, la desviación prixmática d_i y el ángulo de junta θ_i de cada junta se describen en la tabla F.1

Una vez definido el sistema de coordenadas de las uniones, así como sus parámetros, se puede obtener la ecuación cinemática. La matriz que transforma entre sistemas de coordenadas adjuntas ${}_{i-1}{}^{i}T$ viene dada por una matriz F.1, resultado del producto

[1]Se trata de un procedimieto sistemático para describir la estructura cinemática de una cadena articulada constituida por articulaciones con un solo grado de libertad. A cada articulación se le asigna un Sistema de Referencia Local con origen en un punto q_i y ejes ortonormales $\{x_i, y_i, z_i\}$, siendo q_0 el punto de anclaje fijo e inmóvil y $\{x_0, y_0, z_0\}$ los ejes ortonormales de la Base sobre la que está montada toda la estructura de la cadena.

i	α_{i-1} (rad)	a_{i-1} (mm)	d_i (mm)	θ_i (rad)
1	0	0,0	330,0	θ_1
2	$-\pi/2$	0,0	0,0	θ_2
3	$\pi/2$	0,0	316,0	θ_3
4	$\pi/2$	82,5	0,0	θ_4
5	$-\pi/2$	−82,5	384,0	θ_5
6	$\pi/2$	0,0	0,0	θ_6
7	$\pi/2$	88,0	0,0	θ_7

Tabla F.1: Parámetros D-H del robot Franka Emika Panda.

$R_x(\alpha_{i-1})$, que es la matriz de rotación que representa la rotación del ángulo α_{i-1} alrededor del eje x, $R_z(\theta_i)$, que de la misma manera representa la rotación de los ángulos de junta θ_i alrededor del eje z, $D_x(a_{i-1})$, que es la matriz de traslación sobre el eje x, y $D_z(d_i)$, que representa la traslación sobre el eje z.

$$ {}_{i-1}{}^{i}T = R_x(\alpha_{i-1})D_x(a_{i-1})R_z(\theta_i)D_z(d_i) \tag{F.1} $$

$$ = \begin{pmatrix} \cos(\theta_i) & -\sin(\theta_i) & 0 & \alpha_{i-1} \\ \sin(\theta_i)\cos(\alpha_{i-1}) & \cos(\theta_i)\cos(\alpha_{i-1}) & -\sin(\alpha_{i-1}) & -\sin(\alpha_{i-1})d_i \\ \sin(\theta_i)\sin(\alpha_{i-1}) & \cos(\theta_i)\sin(\alpha_{i-1}) & \cos(\alpha_{i-1}) & \cos(\alpha_{i-1})d_i \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{F.2} $$

Desarrollando F.1 para los sistemas de coordenadas de las 7 juntas:

$$ {}_{0}{}^{1}T = \begin{pmatrix} \cos(\theta_1) & -\sin(\theta_1) & 0 & 0 \\ \sin(\theta_1) & \cos(\theta_1) & 0 & 0 \\ 0 & 0 & 1 & d_i \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{F.3} $$

$$ {}_{1}{}^{2}T = \begin{pmatrix} \cos(\theta_2) & -\sin(\theta_2) & 0 & -\pi/2 \\ 0 & 0 & 1 & d_2 \\ -\sin(\theta_2) & -\cos(\theta_2) & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{F.4} $$

$$ {}_{2}{}^{3}T = \begin{pmatrix} \cos(\theta_3) & -\sin(\theta_3) & 0 & \pi/2 \\ 0 & 0 & -1 & -d_3 \\ \sin(\theta_3) & \cos(\theta_3) & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{F.5} $$

$$ {}_{3}{}^{4}T = \begin{pmatrix} \cos(\theta_4) & -\sin(\theta_4) & 0 & \pi/2 \\ 0 & 0 & -1 & -d_4 \\ \sin(\theta_4) & \cos(\theta_4) & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{F.6} $$

$$ {}_{4}{}^{5}T = \begin{pmatrix} \cos(\theta_5) & -\sin(\theta_5) & 0 & -\pi/2 \\ 0 & 0 & 1 & d_5 \\ -\sin(\theta_5) & -\cos(\theta_5) & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{F.7} $$

$$ {}_5^6T = \begin{pmatrix} \cos(\theta_6) & -\sin(\theta_6) & 0 & \pi/2 \\ 0 & 0 & -1 & -d_6 \\ \sin(\theta_6) & \cos(\theta_6) & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{F.8} $$

$$ {}_6^7T = \begin{pmatrix} \cos(\theta_7) & -\sin(\theta_7) & 0 & \pi/2 \\ 0 & 0 & -1 & -d_7 \\ \sin(\theta_7) & \cos(\theta_7) & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{F.9} $$

Las transformadas asociadas a cada junta se pueden multiplicar para obtener la transformación de las coordenadas entre eslabones conjuntos de la cadena. Para obtener la transformación entre el sistema de coordenadas en el origen y el final de la cadena, ${}_0^7T$:

$$ {}_0^7T = {}_0^1T\,{}_1^2T\,{}_2^3T\,{}_3^4T\,{}_4^5T\,{}_5^6T\,{}_6^7T = \begin{pmatrix} {}_0^7R_{3x3} & {}_0^7P_{3x1} \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} {}_0^7r_{11} & {}_0^7r_{12} & {}_0^7r_{13} & {}_0^7p_x \\ {}_0^7r_{21} & {}_0^7r_{22} & {}_0^7r_{23} & {}_0^7p_y \\ {}_0^7r_{31} & {}_0^7r_{32} & {}_0^7r_{33} & {}_0^7p_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{F.10} $$

El vector de traslación ${}_0^7P_{3x1}$ representa las coordenadas del punto final de la cadena en el sistema de referencia de coordenadas inicial. Sus 3 componentes p_x, p_y y p_z representan la traslación en los ejes x_0, y_0 y z_0.

$$ \begin{aligned} {}_0^7p_x = {} & -384\,(-\sin\theta_1\sin\theta_3 + \cos\theta_1\cos\theta_2\cos\theta_3)\sin\theta_4 + 384\sin\theta_2\cos\theta_1\cos\theta_4 \\ & + 316\sin\theta_2\cos\theta_1 \end{aligned} \tag{F.11} $$

$$ \begin{aligned} {}_0^7p_y = {} & -384\,(\sin\theta_1\cos\theta_2\cos\theta_3 + \sin\theta_3\cos\theta_1)\sin\theta_4 + 384\sin\theta_1\sin\theta_2\cos\theta_4 \\ & + 316\sin\theta_1\sin\theta_2 \end{aligned} \tag{F.12} $$

$$ {}_0^7p_z = 384\sin\theta_2\sin\theta_4\cos\theta_3 + 384\cos\theta_2\cos\theta_4 + 316\cos\theta_2 + 330 \tag{F.13} $$

La matriz de rotación ${}_0^7R_{3x3}$ desde el sistema de coordenadas original hasta el sistema de coordenadas de la última junta es una matriz $3x3$. Cada uno de sus componentes es una combinación sinusoidal de los respectivos parámetros y ángulos de junta. En nuestro caso, con ayuda del paquete simbólico *PYText* podemos calcular cada componente de la matriz, lo que da idea de la complejidad con la que aumentan la cinemática de un robot al aumentar el número de eslabones:

$$
\begin{aligned}
{}_0^7r_{11} = {} & ((((-\sin\theta_1\sin\theta_3+\cos\theta_1\cos\theta_2\cos\theta_3)\cos\theta_4+\sin\theta_2\sin\theta_4\cos\theta_1)\cos\theta_5 \\
& -(\sin\theta_1\cos\theta_3+\sin\theta_3\cos\theta_1\cos\theta_2)\sin\theta_5)\cos\theta_6 \\
& +(-(-\sin\theta_1\sin\theta_3+\cos\theta_1\cos\theta_2\cos\theta_3)\sin\theta_4+\sin\theta_2\cos\theta_1\cos\theta_4)\sin\theta_6)\cos\theta_7 \\
& +(((-\sin\theta_1\sin\theta_3+\cos\theta_1\cos\theta_2\cos\theta_3)\cos\theta_4+\sin\theta_2\sin\theta_4\cos\theta_1)\sin\theta_5 \\
& +(\sin\theta_1\cos\theta_3+\sin\theta_3\cos\theta_1\cos\theta_2)\cos\theta_5)\sin\theta_7
\end{aligned}
\tag{F.14}
$$

$$
\begin{aligned}
{}_0^7r_{12} = {} & -((((-\sin\theta_1\sin\theta_3+\cos\theta_1\cos\theta_2\cos\theta_3)\cos\theta_4+\sin\theta_2\sin\theta_4\cos\theta_1)\cos\theta_5 \\
& -(\sin\theta_1\cos\theta_3+\sin\theta_3\cos\theta_1\cos\theta_2)\sin\theta_5)\cos\theta_6+(-(-\sin\theta_1\sin\theta_3 \\
& +\cos\theta_1\cos\theta_2\cos\theta_3)\sin\theta_4+\sin\theta_2\cos\theta_1\cos\theta_4)\sin\theta_6)\sin\theta_7 \\
& +(((-\sin\theta_1\sin\theta_3+\cos\theta_1\cos\theta_2\cos\theta_3)\cos\theta_4+\sin\theta_2\sin\theta_4\cos\theta_1)\sin\theta_5 \\
& +(\sin\theta_1\cos\theta_3+\sin\theta_3\cos\theta_1\cos\theta_2)\cos\theta_5)\cos\theta_7
\end{aligned}
\tag{F.15}
$$

$$
\begin{aligned}
{}_0^7r_{13} = {} & (((-\sin\theta_1\sin\theta_3+\cos\theta_1\cos\theta_2\cos\theta_3)\cos\theta_4+\sin\theta_2\sin\theta_4\cos\theta_1)\cos\theta_5 \\
& -(\sin\theta_1\cos\theta_3+\sin\theta_3\cos\theta_1\cos\theta_2)\sin\theta_5)\sin\theta_6-(-(-\sin\theta_1\sin\theta_3 \\
& +\cos\theta_1\cos\theta_2\cos\theta_3)\sin\theta_4+\sin\theta_2\cos\theta_1\cos\theta_4)\cos\theta_6
\end{aligned}
\tag{F.16}
$$

$$
\begin{aligned}
{}_0^7r_{21} = {} & ((((\sin\theta_1\cos\theta_2\cos\theta_3+\sin\theta_3\cos\theta_1)\cos\theta_4+\sin\theta_1\sin\theta_2\sin\theta_4)\cos\theta_5 \\
& -(\sin\theta_1\sin\theta_3\cos\theta_2-\cos\theta_1\cos\theta_3)\sin\theta_5)\cos\theta_6+(-(\sin\theta_1\cos\theta_2\cos\theta_3 \\
& +\sin\theta_3\cos\theta_1)\sin\theta_4+\sin\theta_1\sin\theta_2\cos\theta_4)\sin\theta_6)\cos\theta_7+(((\sin\theta_1\cos\theta_2\cos\theta_3 \\
& +\sin\theta_3\cos\theta_1)\cos\theta_4+\sin\theta_1\sin\theta_2\sin\theta_4)\sin\theta_5+(\sin\theta_1\sin\theta_3\cos\theta_2 \\
& -\cos\theta_1\cos\theta_3)\cos\theta_5)\sin\theta_7
\end{aligned}
\tag{F.17}
$$

$$
\begin{aligned}
{}_0^7r_{22} = {} & -((((\sin\theta_1\cos\theta_2\cos\theta_3+\sin\theta_3\cos\theta_1)\cos\theta_4+\sin\theta_1\sin\theta_2\sin\theta_4)\cos\theta_5 \\
& -(\sin\theta_1\sin\theta_3\cos\theta_2-\cos\theta_1\cos\theta_3)\sin\theta_5)\cos\theta_6+(-(\sin\theta_1\cos\theta_2\cos\theta_3 \\
& +\sin\theta_3\cos\theta_1)\sin\theta_4+\sin\theta_1\sin\theta_2\cos\theta_4)\sin\theta_6)\sin\theta_7 \\
& +(((\sin\theta_1\cos\theta_2\cos\theta_3+\sin\theta_3\cos\theta_1)\cos\theta_4+\sin\theta_1\sin\theta_2\sin\theta_4)\sin\theta_5 \\
& +(\sin\theta_1\sin\theta_3\cos\theta_2-\cos\theta_1\cos\theta_3)\cos\theta_5)\cos\theta_7
\end{aligned}
\tag{F.18}
$$

$$
\begin{aligned}
{}_0^7r_{23} = {} & (((\sin\theta_1\cos\theta_2\cos\theta_3+\sin\theta_3\cos\theta_1)\cos\theta_4 \\
& +\sin\theta_1\sin\theta_2\sin\theta_4)\cos\theta_5-(\sin\theta_1\sin\theta_3\cos\theta_2-\cos\theta_1\cos\theta_3)\sin\theta_5)\sin\theta_6 \\
& -(-(\sin\theta_1\cos\theta_2\cos\theta_3+\sin\theta_3\cos\theta_1)\sin\theta_4+\sin\theta_1\sin\theta_2\cos\theta_4)\cos\theta_6
\end{aligned}
\tag{F.19}
$$

$$
\begin{aligned}
{}_0^7r_{31} = &(((-\sin\theta_2\cos\theta_3\cos\theta_4 + \sin\theta_4\cos\theta_2)\cos\theta_5\sin\theta_2\sin\theta_3\sin\theta_5)\cos\theta_6 + \\
&+ (\sin\theta_2\sin\theta_4\cos\theta_3 + \cos\theta_2\cos\theta_4)\sin\theta_6)\cos\theta_7 + ((-\sin\theta_2\cos\theta_3\cos\theta_4 \\
&+ \sin\theta_4\cos\theta_2)\sin\theta_5 - \sin\theta_2\sin\theta_3\cos\theta_5)\sin\theta_7
\end{aligned} \tag{F.20}
$$

$$
\begin{aligned}
{}_0^7r_{32} = &-(((-\sin\theta_2\cos\theta_3\cos\theta_4 + \sin\theta_4\cos\theta_2)\cos\theta_5 + \sin\theta_2\sin\theta_3\sin\theta_5)\cos\theta_6 + \\
&(\sin\theta_2\sin\theta_4\cos\theta_3 + \cos\theta_2\cos\theta_4)\sin\theta_6)\sin\theta_7 + ((-\sin\theta_2\cos\theta_3\cos\theta_4 \\
&+ \sin\theta_4\cos\theta_2)\sin\theta_5 - \sin\theta_2\sin\theta_3\cos\theta_5)\cos\theta_7
\end{aligned} \tag{F.21}
$$

$$
\begin{aligned}
{}_0^7r_{33} = &((-\sin\theta_2\cos\theta_3\cos\theta_4 + \sin\theta_4\cos\theta_2)\cos\theta_5 + \sin\theta_2\sin\theta_3\sin\theta_5)\sin\theta_6 \\
&- (\sin\theta_2\sin\theta_4\cos\theta_3 + \cos\theta_2\cos\theta_4)\cos\theta_6
\end{aligned} \tag{F.22}
$$

Pueden describirse los componentes de la matriz de rotación en función de los ángulos α, β y γ, que representan los ángulos de rotación respecto a los ejes fijo de coordenadas x_0, y_0 y z_0 respectivamente:

$$
{}_0^7R_{XYZ}(\alpha,\beta,\gamma) = \tag{F.23}
$$

$$
\begin{pmatrix}
\cos\alpha\cos\beta & \cos\alpha\sin\beta\sin\gamma - \sin\alpha\cos\gamma & \cos\alpha\sin\beta\cos\gamma + \sin\alpha\sin\gamma \\
\sin\alpha\cos\beta & \sin\alpha\sin\beta\sin\gamma - \cos\alpha\cos\gamma & \sin\alpha\sin\beta\cos\gamma - \cos\alpha\sin\gamma \\
-\sin\beta & \cos\beta\sin\gamma & \cos\beta\cos\gamma
\end{pmatrix} \tag{F.24}
$$

La relación entre los componentes de la matriz de rotación ${}_0^7R_{3x3}$ y los ángulos α, β y γ es directa:

$$
\alpha = \arctan\left(\frac{r_{21}}{r_{11}}\right) \tag{F.25}
$$

$$
\beta = \arctan\left(\frac{-r_{31}}{\sqrt{r_{11}^2 + r_{21}^2}}\right) \tag{F.26}
$$

$$
\gamma = \arctan\left(\frac{r_{32}}{r_{33}}\right) \tag{F.27}
$$

Aunque laborioso y complejo [2], es posible obtener la cinemática directa a partir de las expresiones algebraicas enumeradas. La posición y la rotación de cada eslabón viene determinada por una expresión matricial del tipo $F_{fk}(\theta_1,\theta_2,\theta_3,\theta_4,\theta_5,\theta_6,\theta_7) \rightarrow \{p_x, p_y, p_z, \alpha, \beta, \gamma\}$ que depende únicamente de los ángulos de junta del robot. La expresión, formada por productos y sumas de senos y cosenos, es derivable y permite obtener la dinámica del sistema.

[2] En https://github.com/albertost85/RoboticArm/blob/main/notebooks/Symbolic_Matrix.ipynb está el código que ha permitido obtener las expresiones anteriores utilizando el paquete SymPy para Python.

Anexos G

Algoritmos prácticos

G.1. Ecuación de Bellman

Retorno con descuento El retorno con descuento prioriza las recompensas próximas multiplicando cada una de ellas por un factor de descuento $\gamma \in [0, 1]$, que puede escribirse de forma recursiva:

$$G_t = \sum_{j=0}^{T} \gamma^j r_{t+j+1} = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+1} + ... + \gamma T - 1 r_T G_t \qquad = r_{t+1} + \gamma(G_{t+1}) \tag{G.1}$$

G.1.1. Ecuación de Bellman para la función V

Podemos definir de forma recursiva la función de Bellman para la función V, que expresa que el valor de un estado se puede obtener como una suma de la recompensa inmediata y el valor descontado del siguiente estado.

$$V(s_t) = r_{t+1} + \gamma V(s_{t+1}) \tag{G.2}$$

Para un entorno estocástico de políticas estocásticas, y utilizando la notación π para especificar que esos valorees es obtienen bajo la premisa de una política π:

$$V_\pi(s) = \sum_a \pi(a|s) \sum_s^{\prime} P(s'|s, a)[R(s, a, s') + \gamma V_\pi(s')] \tag{G.3}$$

$P(s'|s, a)$ indica la probabilidad de alcanzar el estado s'l realizar una acción a en el estado s. En esta expresión se está calculando la esperanza matemática (el promedio ponderado), es decir, una suma de la estimación a partir de cada potencial valor del siguiente estado multiplicado por la probabilidad de transición a este estado.

Una notación para expresar la misma fórmula que podemos expresar en el libro de Sutton, y para enfatizar que las acciones siguen una distribución dada por la política,

y que los estados siguen la distribución de la dinámica del entorno es la siguiente [5]

$$V_\pi(s) = \mathbb{E}_{a\simeq\pi, s'\simeq p} R(s,a,s') + \gamma V_\pi(s') \tag{G.4}$$

Esta ecuación de Bellman simplifica el cálculo de la función de valor del estado, de modo que en lugar de tener que calcular el valor en múltiples time steps, podemos encontrar la solución de un problema complejo dividiéndolo en subproblemas recursivos más simples y encontrando sus soluciones con, por ejemplo, el algoritmo de Value Iteration.

G.1.2. Ecuación de Bellman para la función Q

De forma análoga, podemos expresar el valor de la función Q para un entorno estocástico que sigue una política π también estocástica:

$$Q_\pi(s,a) = \sum P(s'|s,a)[R(s,a,s') + \gamma \sum_a \pi(a'|s') Q_\pi(s',a')] \tag{G.5}$$

En [1], esta ecuación se conoce directamente como *Bellman expectation equation* o función Q.

G.2. Métodos value-based

Uno de los requisitos fundamentales para el proceso de optimización es que los datos de entrenamiento sean independientes y que estén distribuidos de manera idéntica. Pero, en general, cuando el agente interactúa con el entorno, la secuencia de tuplas de experiencia que se obtienen puede estar altamente correlacionada y el algoritmo de Q-Learning que aprende de cada una de estas tuplas en orden secuencial corre el riesgo de dejarse influir por los efectos de esta correlación[5].

La propuesta de los autores del método DQN para evitar esto fue utilizar un gran búfer de memoria con un histórico de experiencias pasadas y extraer los datos para entrenar de este búfer, en lugar de usar las experiencias más recientes del agente. Esta técnica se denomina experience replay y consiste en mantener un búfer de memoria que contiene una colección de tuplas de experiencia (s_t, a_t, r_t, s_{t+1}) anteriores del agente. Se almacenan experiencias pasadas y luego usar un subconjunto aleatorio de estas experiencias para actualizar la red neuronal, en lugar de usar solo la experiencia más reciente. Y esto se consigue muestreando un pequeño lote de tuplas de este búfer experience replay.

Las tuplas se agregan gradualmente al búfer a medida que el agente interactúa con el entorno. La implementación básica se realiza con un búfer de tamaño fijo, en el que se

agregan nuevas tuplas con los datos que va obteniendo el agente mientras se eliminan las tuplas de la experiencia más antigua.

Además de romper correlaciones «dañinas» para el aprendizaje, esta técnica nos permite aprender de tuplas individuales varias veces, recordar ocurrencias raras y, en general, hacer un mejor uso de nuestra experiencia.

Hay dos fases principales que están intercaladas en el algoritmo DQN. Una es cuando se obtienen las muestras de la interacción del agente con el entorno al realizar acciones y estas se almacenan en forma de tuplas en el búfer de memoria experience repaypresentado anteriormente. La otra fase es aquella en la que se selecciona aleatoriamente un pequeño lote de tuplas de esta experience repay, y se entrena la red neuronal con ese lote de datos usando técnicas de *Machine Learning*, como el gradiente de la figura 2.4.

off-policy En el aprendizaje off-policy, la tabla o función Q se aprende seleccionando acciones con estrategias diferentes a las marcadas por la política que guía al agente.

Estas dos fases no dependen directamente una de la otra y podríamos realizar varios pasos de muestreo y luego un paso de aprendizaje, o incluso varios pasos de aprendizaje con diferentes lotes aleatorios. La única restricción es que no se podrá ejecutar la fase de aprendizaje de inmediato y se deberá esperar hasta tener suficientes tuplas en el búfer de memoria experience replay.

El algoritmo se pueden encontrar en el repositorio [1], y una explicación más detallada en el libro [5].

Algoritmo G.1: DQN

```
Initialize main network Q
Initialize target network Q̂
Initialize experience replay memory D
Initialize the agent to interact with the environment
while not converged do
        ε ← setting new epsilon with ε − decay
        Choose an action a from state s using policy ε − greedy(Q)
        Agent takes action a, observe reward r, and next state s'
        Store transition (s,a,r,s',done) in D
        if enough experiences in D then
                Sample a random batch of N transitions from D
                for every transition (s_i, a_i, r_i, s'_i) in batch do
                        Compute Q(s_i, a_i)
                        Compute max Q̂(s'_i, a_i')
                        if done then
                                y_i ← r_i
                        else
                                y_i ← r_i + γ max Q̂(s'_i, a_i')
                        end
                        Calculate the loss Λ
                        Update Q using the SGD algorithm by minimizing the loss Λ
                        Every C steps, copy parameters from Q to Q̂
```

[1] https://github.com/jorditorresBCN/aprendizaje-por-refuerzo

```
                end
            end
        end
    return Q
```

G.3. Métodos policy-based

Intuitivamente, el algoritmo comienza con una estimación inicial de los parámetros de la red neuronal (puede ser aleatoria) y, luego, el algoritmo evalúa el gradiente de la función del retorno esperado en ese punto que indica la dirección del aumento más pronunciado. De esta manera, podemos saber cómo modificar el valor de los parámetros (coordenadas en el espacio). Con ello se espera que el nuevo valor de los parámetros defina un punto en el espacio para el cual el retorno esperado sea mayor. Luego, el algoritmo repite este proceso iterativamente hasta que alcanza los parámetros que consiguen el retorno máximo.

La aplicación de estos métodos es independiente de la naturaleza estocástica o determinística de la política. Con una política estocástica, la salida de la red neuronal es un vector que representa una distribución de probabilidad de las acciones (en lugar de devolver una única acción determinista) con la que el agente seleccionará una acción. Esto significa que, si el agente se encuentra en el mismo estado diversas veces, es posible que no siempre seleccione la misma acción.

El método REINFORCE se basa en trayectorias en lugar de episodios, ya que asi´se permite actualizar la política óptima tanto en tareas episódicas (devuelven una recomensa únicamente al final del episodio) como continuas. **Retorno de trayectoria** τ se conforma por la secuencia de ganancias (recompensa por el factor de descuento)[5].

Algoritmo G.2: REINFORCE

```
Initialize the parameters θ
while not converged do
    Collect trajectory τ = (s_0, a_0, r_1, s_1, a_1, ..., r_{H+q}, s_{H+1})
    Estimate R(τ) = (G_0, G_1, ..., G_H)
    Update expected gain U(θ)
    ∇_θ U(θ) ← Σ_{t=0}^{H} ∇ log π_θ(a_t|s_t) G_t
    θ ← θ + α ∇_θ U(θ)
end
return π
```

Podemos adaptar un ejemplo del entorno gym *Cart-and-pole* para entrenar un agente REINFORCE que interactúa con nuestro brazo robótico. Incluye la creación de una red neuronal con una sola capa oculta de 256 neuronas. Este ejemplo está basado en el libro[5]. Puede encontrarse el algoritmo original en el repositorio [2], y este mismo algoritmo en el repositorio https://github.com/albertost85/RoboticArm.

[2] https://github.com/jorditorresBCN/aprendizaje-por-refuerzo

```
HORIZON = 500
MAX_TRAJECTORIES = 500
GAMMA = 0.99
HIDDEN_SIZE = 256

score = []
class Agent:
    def __init__(self, env):
        self.score =[]
        self.transitions = []
        self.current_state = env.reset()
        self.done = False
        obs_size = env.observation_space.shape[0]
        n_actions = env.action_space.shape[0]


        self.model = torch.nn.Sequential(
            torch.nn.Linear(obs_size, HIDDEN_SIZE),
            torch.nn.ReLU(),
            torch.nn.Linear(HIDDEN_SIZE, n_actions),
            torch.nn.Softmax(dim=0)
        )
        print (model)
        print("REINFORCE␣agent␣on␣duty")

    def select_action(self, env):
        self.pred = model(torch.from_numpy(state).float())
        self.action = np.random.choice(np.array([0,1]), p=self.pred.data.numpy())
        return self.action
    def train(self,env):
        for t in range(HORIZON):
            self.actions_prob = model(torch.from_numpy(current_state).float())
            self.action = np.random.choice(np.array([0,1]), p=self.actions_prob.data.numpy())
            self.previous_state = self.current_state
            self.current_state, _, self.done, _ = env.step(self.action)
            self.transitions.append((self.previous_state, self.action, t+1))
            if self.done:
                break
        self.score.append(len(self.transitions))
        self.reward_batch = torch.Tensor([r for (s,a,r) in self.transitions]).flip(dims=(0,))
    batch_Gvals =[]
    for i in range(len(self.transitions)):
        new_Gval=0
        power=0
        for j in range(i,len(transitions)):
            new_Gval=new_Gval+((GAMMA**power)*reward_batch[j]).numpy()
            power+=1
        self.batch_Gvals.append(new_Gval)
    self.expected_returns_batch=torch.FloatTensor(self.batch_Gvals)
    self.expected_returns_batch /= self.expected_returns_batch.max()

    self.state_batch = torch.Tensor([s for (s,a,r) in self.transitions])
    self.action_batch = torch.Tensor([a for (s,a,r) in self.transitions])

    self.predicted_batch = model(self.state_batch)
    self.prob_batch = self.predicted_batch.gather(dim=1,index=self.action_batch.long().view(-
        1,1)).squeeze()

    self.loss = -torch.sum(torch.log(self.prob_batch) * self.expected_returns_batch)

    self.loss.backward()

    if self.trajectory % 50 == 0 and self.trajectory>0:
        print('Trajectory␣{}\tAverage␣Score:␣{:.2f}'.format(self.trajectory, np.mean(self.score[-50:-1])))
```

Varianza de los gradientes Con el algoritmo REINFORCE, el método actualiza los parámetros de la política tomando muestras aleatorias. Esto introduce una alta varia-

bilidad inherente en las probabilidades y los valores de recompensa esperados, porque durante el entrenamiento cada trayectoria puede desviarse entre sí en gran medida. En consecuencia, esto generará gradientes ruidosos que provocarán un aprendizaje inestable con una lenta convergencia y pueden desviar el aprendizaje en una dirección no óptima[5].

Ejecución pesada La ejecución es pesada en términos computacionales, especialmetne para un entorno robótico como el nuestro. Para ser eficiente, este algoritmo nevesita explorar el entorno.

La elección entre los métodos *policy-based* y *value-based* depende del problema. Si el acceso al entorno es eficiente y rápido en términos computacionales, un método algoritmo DQN que permita parametrizar ese entorno en función de unos valores Q será una opción interesante. Sin embargo, en entornos más donde la observación del entorno no sea completa, los métodos *value-based* ofrecerán una ventaja.