Diseño y simulación de un controlador inteligente utilizando aprendizaje por refuerzo Q-learning para la navegación autónoma de dos robots móviles

Sergio Manuel Carreño Puentes

10441711897

Universidad Antonio Nariño

Programa Ingeniería Electrónica

Facultad de Ingeniería Mecánica, Electrónica y Biomédica

Bogotá D.C, Colombia

2022

Diseño y simulación de un controlador inteligente utilizando aprendizaje por refuerzo Q-learning para la navegación autónoma de dos robots móviles

Sergio Manuel Carreño Puentes

Proyecto de grado presentado como requisito parcial para optar al título de:
Ingeniero Electrónico

Director:
Ph.D. Christian Camilo Erazo Ordoñez

Línea de Investigación:
Control, automatización y robótica

Universidad Antonio Nariño
Programa Ingeniería Electrónica
Facultad de Ingeniería Mecánica, Electrónica y Biomédica
Bogotá D.C, Colombia
2022

NOTA DE ACEPTACIÓN

El trabajo de grado titulado

_____,

Cumple con los requisitos para optar

Al título de _____.

_____

Firma del Tutor

_____

Firma Jurado

_____

Firma Jurado

Bogotá, 07 de junio de 2022

Contenido

Lista de Figuras

Pág.

Lista de Tablas

Dedicatoria,

A Dios, por estar siempre presente en mi vida y guiarme en los momentos difíciles.

A mis padres, porque siempre han estado a mi lado y me han apoyado en todos mis sueños. Sin su apoyo no hubiera podido llegar hasta aquí, así que este trabajo es para ustedes.

Agradecimientos

Primeramente y, ante todo, quiero agradecer a Dios por todas las bendiciones que me ha brindado, especialmente por el don de la salud y el intelecto.

Agradezco también a la Universidad Antonio Nariño, mi alma mater, que me ha brindado la oportunidad de estudiar y formarme como un profesional calificado y forjado en los valores.

A los docentes, les agradezco por compartir su conocimiento profesional y personal.

A mi director, el Ingeniero Christian Erazo, le agradezco por la paciencia, la comprensión y por expresarme sus palabras de aliento, sabiendo él que mi proceso, desde el inicio, resultó más complejo de lo común.

A mis amigos, algunos ya ingenieros y otros que están a puertas de serlo, les agradezco por el apoyo, las palabras de aliento, las risas, los momentos especiales y porque sé que están ahí para cuando los necesite y de igual manera, ahí estaré para ellos.

A quien me ha ayudado durante todo mi proceso como estudiante en la UAN.

Agradezco especialmente a las personas que de una u otra forma me ayudaron activamente durante el desarrollo de este trabajo de grado, mil gracias.

A mi familia, les agradezco por creer siempre en mí y ser mi soporte.

Resumen

La planeación de trayectorias en los robots móviles autónomos es un problema abierto debido a que, al trabajar en ambientes dinámicos, resulta muy costoso programar todo el sistema de navegación para una aplicación particular o en su defecto resultaría muy complicado, para el programador, lograr predecir de manera acertada los cambios en el entorno. A fin de contribuir a este campo, en el presente documento se muestra el proceso de diseño de un controlador inteligente basado en aprendizaje reforzado y más concretamente utilizando el algoritmo Q-learning para lograr conducir dos robots móviles a través de un entorno simulado, y que de manera autónoma logren aprender la trayectoria que los llevará a una posición objetivo sin poseer conocimiento previo sobre el ambiente de trabajo.

Palabras clave: Aprendizaje por refuerzo, aprendizaje automático, aprendizaje profundo, neurona artificial, redes neuronales, robots autónomos.

Abstract

Trajectory planning in autonomous mobile robots is an open problem because, when working in dynamic environments, it is very expensive to program the entire navigation system for a particular application or, failing that, it would be very difficult for the programmer to correctly predict the changes in the environment. In order to contribute to this field, this document shows the process of designing an intelligent controller based on reinforced learning and more specifically using the Q-learning algorithm to drive two mobile robots through a simulated environment, and that autonomously manage to learn the trajectory that will take them to a target position without having prior knowledge about the work environment.

Keywords: Reinforcement learning, machine learning, deep learning, artificial neuron, neural networks, autonomous robots.

1. Introducción

El término robótica se introdujo por primera vez en la obra literaria de ficción Frankenstein de Mary Shelley a finales del año 1817, y además es donde se dio lugar al término robot, pero fue Isaac Asimov quien hizo aparecer por primera vez en alguna de sus obras una máquina mecánica bien diseñada (Contreras, 2003). Desde aquel entonces se ha visto la gran acogida y el inmenso impacto que los robots han tenido en el crecimiento industrial convirtiéndose en parte fundamental del desarrollo tecnológico, teniendo además una gran participación en acciones de rescate o siendo hasta el momento los únicos residentes en marte (Díaz Pinzón, 2018).

Especialmente en los últimos años la robótica se ha incorporado en la vida cotidiana de muchas personas (Ambhore, 2020), inclusive sin tener el conocimiento de las diversas áreas que componen dicho campo, tales como la física, la matemática o el cálculo (Díaz Pinzón, 2018). Por lo cual, se podrían presentar dificultades, si no se cuenta con la ayuda de un experto, al momento de configurar el robot para realizar un trabajo diferente al que se estableció en un principio (Peña Solórzano, 2015). O, en otro contexto más específico, si es requerido que el robot se desplace por ambientes dinámicos donde, para el programador, resulta de gran dificultad predecir los cambios del entorno (de Lope, 2008). Por ello, resulta necesario buscar la manera de equiparlos con herramientas que ofrezcan al robot la opción de tomar decisiones según sea necesario y partiendo de los conocimientos obtenidos de una base de datos, o de manera más práctica y eficiente que él aprenda por sí solo y en base a su propia experiencia (sin tener conocimientos previos) (CSV, 2017). En una proyección a futuro los objetivos para la robótica son numerosos, como caminatas y carreras semejantes a las humanas o algunos animales, la correcta implementación de la navegación móvil en entornos peatonales, la automatización colaborativa, inspección y mantenimiento automatizado de aeronaves (Pierson & Gashler, 2017), entre otros. En el camino hacia el cumplimiento de estos objetivos se presentan desafíos importantes y para los cuales las tecnologías de inteligencia artificial (IA) se encuentran entre las más solicitadas.

El problema de la navegación autónoma de robots móviles es un campo ampliamente estudiado y desde hace años se buscan diferentes métodos para resolverlo (Ramírez Serrano, 1996), sin embargo, es en los años más recientes se han empezado a implementar sistemas

basados en inteligencia artificial debido a su gran crecimiento y variedad de aplicaciones. En especial, el aprendizaje por refuerzo o aprendizaje reforzado se presenta como una alternativa altamente eficiente para el desarrollo de robots móviles puesto que se no se requiere una base de datos para entrenar el modelo (como las otras técnicas de IA) lo cual reduce tiempos de entrenamiento y la necesidad de potencia computacional (CSV, 2017; Lobos Tsunekawa, 2018). Además, según Mónica Rivera los resultados obtenidos al comparar el modelo simulado con un modelo físico son aceptables aun cuando el entorno de simulación y entrenamiento no es idéntico al entorno de pruebas (Arias Rivera, 2018).

Y es que el aprendizaje por refuerzo no se limita únicamente a los robots móviles autónomos, sino que se extiende a una amplia variedad de aplicaciones como, por ejemplo, en la navegación autónoma de vehículos de superficie no tripulados (USV) y su potencial uso para construir sistemas no tripulados cuyo entorno de trabajo es el océano y que sean capaces de liderar misiones marítimas complejas (Yan et al., 2021). En la navegación autónoma de globos estratosféricos donde se debe tener en cuenta la evaluación de múltiples señales, como la velocidad del viento y la elevación solar, donde no resulta confiable el uso de técnicas de control convencionales y se debe optar por algoritmos de aprendizaje reforzado capaces de adaptarse al entorno dinámico (Bellemare et al., 2020; Semnani et al., 2020). O en la navegación autónoma de vehículos aéreos no tripulados (UAV) en entornos no estructurados, donde resulta necesario utilizar algoritmos de control suficientemente robustos y de aprendizaje que contribuyan al proceso de adaptación de los vehículos (Ruiz Barreto & Bravo Navarro, 2019).

Un algoritmo de aprendizaje por refuerzo frecuentemente utilizado en la planificación de rutas y la navegación autónoma de robots móviles es el Q-learning (Huang et al., 2005; Li et al., 2006; Strauss & Sahin, 2008), introducido por Watkins en 1989 (Watkins & Peter, 1992), y cuyo motor de aprendizaje se basa en la interacción de estado-acción, en la cual se busca maximizar una función de recompensa estimulando la toma de decisiones que llevarán al agente a realizar una acción en busca de obtener una recompensa cada vez mayor.

Actualmente el concepto de inteligencia artificial abarca diferentes disciplinas y campos de estudio que se pueden jerarquizar. Como se observa a continuación:

- **Inteligencia artificial:** Programas con la habilidad de aprender y razonar como humanos.
    - **Machine Learning (aprendizaje de máquina):** Algoritmos con la habilidad de aprender sin necesidad de ser explícitamente programados.

        - **Aprendizaje supervisado:** Algoritmos de carácter predictivo e impulsado por tareas en el cual se requiere un set de datos previamente etiquetados según la categoría.
        - **Aprendizaje no supervisado:** Algoritmos que no requiere una base de datos previamente etiquetados para detectar los patrones y realizar una predicción.
        - **Aprendizaje reforzado:** Algoritmos que definen modelos enfocados a maximizar una función de recompensa, basados en acciones-recompensas y la respuesta a su interacción con el ambiente. Estos modelos aprenden de sus errores y por ello se asemejan a la psicología conductista de los humanos.

"La IA trata de abarcar y estudiar muchas de las capacidades del hombre para poder ofrecérselas a la máquina y también al propio hombre en el entendimiento de los principios de su inteligencia" (Contreras, 2003), de aquí se puede entender por qué que se busca imitar el proceso por el cual los humanos aprenden ciertas tareas, en base a ensayos, errores y recompensas, siendo este un camino bastante prometedor para lograr cumplir los objetivos a futuro de la robótica, y siendo el aprendizaje por refuerzo el eje central en el desarrollo del presente proyecto, haciendo uso específicamente del algoritmo Q-learning para conseguir que un sistema de control aprenda la trayectoria que deben seguir dos robots móviles a fin de llegar a un punto objetivo dentro de un entrono simulado.

2. Antecedentes

2.1 Planteamiento Del Problema

Uno de los problemas fundamentales de los robots móviles autónomos es la planificación de las rutas. Dado que existen ambientes donde los robots móviles pueden encontrar obstáculos o cambios abruptos en las trayectorias, con lo cual, al momento de realizar tareas de navegación, necesitan apoyo de un experto (Li et al., 2006). Una herramienta útil y prometedora para obtener de manera satisfactoria la autonomía de robots móviles en ambientes dinámicos es el aprendizaje por refuerzo (RL, en inglés Reinforcement Learning), ya que les ofrece la capacidad de comprender y adaptarse a los cambios en el entorno de trabajo (Yan et al., 2021), (Xu, 2021). En este sentido, el robot móvil es considerado un agente (el robot móvil es controlado por un agente) que aprende trayectorias a partir del entrenamiento basado en políticas de recompensa y penalización.

Si embargo, en este enfoque, no es suficiente entrenar el sistema con un solo agente ya que, al desplazarse inicialmente de manera aleatoria por el entorno, puede encontrar y aprender una ruta que lo lleve a la posición deseada en la cual su función de recompensa es máxima, pero si el espacio de trabajo tiene diferentes rutas, no se garantiza que la primera trayectoria aprendida sea la óptima o la más corta entre los dos puntos, lo cual resulta problemático puesto que, si el robot consigue maximizar su función de recompensa, descartará inmediatamente la opción de explorar otras alternativas quedándose con la primera aprendida (Baker et al., 2019; Florensa et al., 2017). Para solucionar este tipo de problemas, generalmente se opta por utilizar sistemas multi agentes, es decir, entornos de simulación en los que se entrenan múltiples agentes para cumplir un mismo objetivo o compitiendo entre ellos por un objetivo individual, obteniendo así un sistema en el cual, cada mínima estrategia aprendida por un agente será suficiente para forzar a los demás agentes a mejorar las suyas, obligándolos a adaptarse y así encontrar estrategias cada vez más avanzadas, en el presente caso, la trayectoria más corta.

En la literatura se encuentra que el aprendizaje por refuerzo utilizando múltiples agentes se aplica a un amplio número de escenarios obteniendo resultados satisfactorios; no obstante, para el aprendizaje de trayectorias óptimas en robots móviles utilizando múltiples

agentes se encontró que es un campo poco estudiado y, generalmente, se hace uso de múltiples algoritmos para solucionar los problemas de navegación. Sin embargo, no se enfocan en encontrar la ruta más corta que llevará a los agentes desde el punto inicial hasta el punto objetivo y, por ello, se pretende utilizar un sistema multi agente para dar solución al problema en cuestión.

Para la adecuación del entorno y el modelamiento de los robots móviles se deben tener en cuenta diferentes aspectos matemáticos que describan al sistema para obtener un modelo suficientemente robusto en el entorno de simulación y reducir al mínimo los errores al momento de llevarlo a la práctica. Partiendo de ello se plantea la pregunta problema; ¿Cómo implementar el algoritmo de aprendizaje por refuerzo Q-learning en un ambiente simulado para lograr que dos robots móviles se desplacen en un entorno, aprendiendo de manera autónoma la trayectoria más corta que los llevará a través de este sin chocar con obstáculos?

2.2 Objetivos

2.2.1 Objetivo General

Diseñar un controlador inteligente basado en el algoritmo de aprendizaje por refuerzo Q-learning para lograr la navegación autónoma de dos robots móviles, en un entorno simulado.

2.2.2 Objetivos específicos

- Definir los estados y acciones para construir la red neuronal del crítico de valor Q que permita el aprendizaje de trayectorias en los robots móviles.
- Establecer la función de recompensa basada en premios y castigos para permitir al agente escoger entre realizar una acción ya conocida (explotación) o realizar una acción por primera vez (exploración).
- Simular el modelo del entorno y los agentes utilizando la herramienta Simulink de Matlab.

- Evaluar el rendimiento del algoritmo observando si los robots móviles aprenden de manera autónoma una trayectoria tal que los lleve desde un punto aleatorio del entorno hacia un punto deseado.

2.3 Metodología

Para cumplir de manera satisfactoria cada uno de los objetivos y teniendo en cuenta que es un proyecto multidisciplinar se pretenden realizar las siguientes actividades:

• En primera instancia se definirá la red neuronal para el crítico de valor Q que será implementada en los dos robots donde se tendrán en cuenta los siguientes estados observables: posición, orientación y velocidad, asimismo se establecerán las dos posibles acciones (traslación y rotación) dependiendo de cada estado. Posteriormente dicha red neuronal se utilizará para definir las representaciones políticas que definirán al agente para ello, se utilizará la función `rlQValueRepresentation` de Matlab.

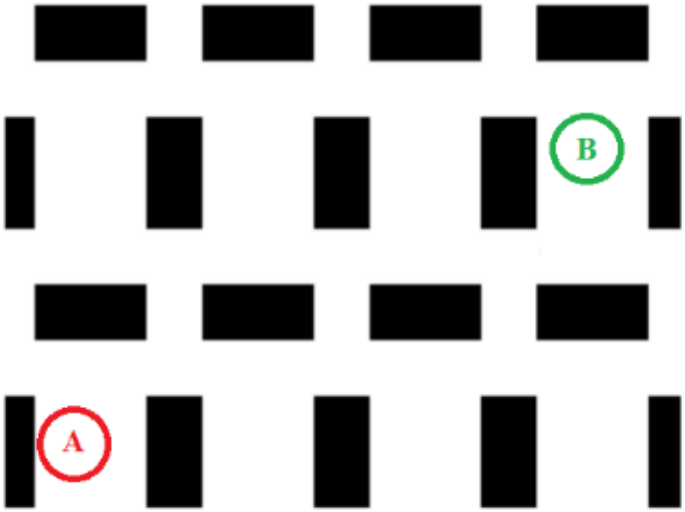
• Se definirá una función de recompensa que anime al robot a moverse hacia la posición objetivo, para ello se buscarán dos funciones que logren hacer esto de manera independiente en cada dimensión. Puesto que la recompensa depende del estado, éste se escogerá teniendo en cuenta que el agente cuenta con la siguiente información: Las observaciones, la recompensa y si el sistema alcanzó una fase terminal, para la presente propuesta, esta fase terminal se considerará como tal si los robots llegan a su ubicación objetivo, si chocan con los bloques negros (ver **Figura *2-1***) que representan un obstáculo o si chocan entre ellos.
Dicha recompensa se traduce en una suma de expresiones matemáticas p. ej. exponenciales, en la que por cada eje (x, y) se busca una función que tenga un valor máximo en la posición objetivo respecto a un plano cartesiano dispuesto sobre el entorno.

• En la **Figura *2-1*** se puede observar una aproximación del entorno que se implementará para validar el algoritmo. En este se encuentra el punto de partida de

los robots (A) y la ubicación final deseada (B), como se aprecia en la figura cada uno de los bloques constituye un obstáculo que los robots deben evitar, asimismo se aprecia que existen diferentes trayectorias que se pueden seguir para lograr llegar al punto B, esto con el fin de verificar si es posible encontrar la trayectoria más corta.

Figura 2-1

Ejemplo del entorno donde se entrenará el agente



Nota. Este es un ejemplo del entorno que se utilizará para entrenar al agente. Fuente: (Tearle, n.d.)

El modelo final podría diferir del modelo presentado en la **Figura *2-1***, sin embargo, cambiaría únicamente el tamaño y la distribución.

Puesto que el modelo no se implementará de manera física se simulará la dinámica del robot utilizando la herramienta Simulink. En este el agente observará seis valores continuos: posición en x, posición en y, ubicación respecto a y ($\sin(\theta)$), ubicación con respecto a x ($\cos(\theta)$), velocidad traslacional (v) y velocidad angular (ω).

Puesto que los agentes, el entorno, los estados observados y la acciones se representan como variables se hará uso la función `rlDDPGAgent` de Matlab para definir al agente.

- Finalmente se analizará el desempeño de la técnica utilizada teniendo en cuenta que el aprendizaje de los agentes lo dicta la función de recompensa y si esta se maximiza. Al realizar el entrenamiento con la función *train* de Matlab por defecto se muestra el progreso del entrenamiento en una pestaña llamada "administrador de episodios" en la cual se puede observar gráficamente la recompensa obtenida por cada episodio de entrenamiento y la gráfica de la recompensa promedio. Si no se abre el administrador de episodios se puede utilizar la función *inspectTrainingResult* de Matlab para visualizar la información del entrenamiento siempre y cuando se guarde la estructura que contenga la información del entrenamiento.

  Por último, se presentarán los resultados obtenidos y las conclusiones teniendo en cuenta el cumplimiento del objetivo general.

3. Marco Teórico

3.1 Aprendizaje por refuerzo

A continuación, se presentan algunos conceptos importantes para tener en cuenta y comprender el aprendizaje por refuerzo. Esta información se basa en las referencias (MathWorks, 2019, 2020; Tearle, n.d.; Torres, n.d.).

El aprendizaje por refuerzo es una técnica de aprendizaje de máquina donde no se necesita una base de datos para entrenar al modelo, este aprende a realizar tareas al interactuar con un entorno dinámico utilizando una política de ensayo y error. En este enfoque de aprendizaje se busca que el modelo maximice una función de recompensa acumulada tomando una serie de decisiones que aprende por sí solo y sin que haya sido programado explícitamente para cumplir la tarea. El modelo de aprendizaje por refuerzo que se utilizó en el presente trabajo está conformado por:

3.1.1 Observaciones y estados

Los estados representan la respuesta del entorno a un estímulo ocasionado por el agente, las observaciones son dichos estados visualizados en un momento específico, para el presente caso, las observaciones representan: la posición del robot (x, y) con respecto a un eje de coordenadas, la orientación $(\sin(\theta), \cos(\theta))$, la velocidad traslacional (v) y la velocidad angular (ω).

Haciendo la analogía con un sistema de control las observaciones se pueden entender como la señal del error.

3.1.2 Agente

El aprendizaje por refuerzo consta de dos pilares generales: el agente y el entorno.

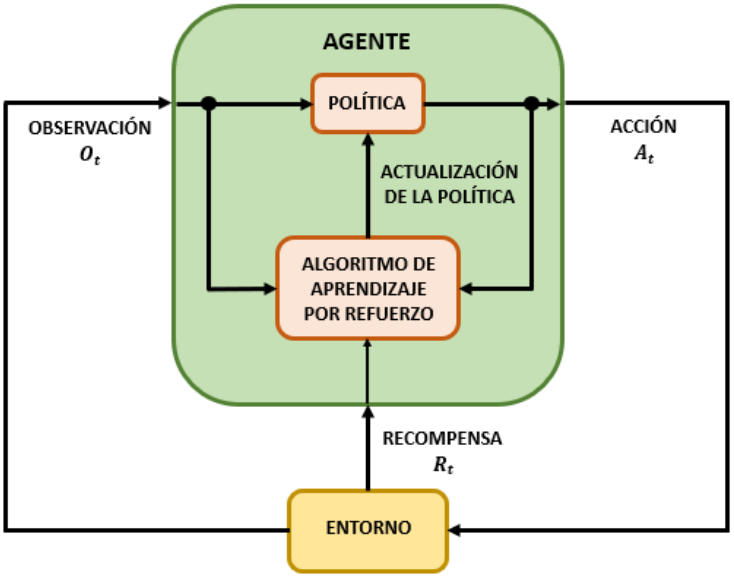
Un agente es el órgano encargado de tomar decisiones que producen consecuencias (genera acciones) en el entorno. El agente está conformado por una política y un algoritmo de aprendizaje (ver **Figura 3-1**) que actualiza constantemente la política a fin de encontrar la más adecuada. Un agente requiere una forma de representar dicha política, para ello, otra

parte importante del agente son las representaciones de actores y críticos (definidos más adelante).

En el siguiente esquema (**Figura 3-1**) se presenta la vista general de un escenario de aprendizaje por refuerzo.

Figura 3-1

Escenario de aprendizaje por refuerzo



Nota. En este diagrama se ilustra de manera general la estructura de un algoritmo de aprendizaje por refuerzo. Fuente. (MathWorks, 2019)

En la caja de herramientas de aprendizaje por refuerzo de MATLAB se dispones de 8 tipos de agentes. Cada uno de ellos se puede entrenar en un entorno con observaciones discretas o continuas, y de manera similar para los espacios de acción (discretos o continuos). Los agentes se pueden recopilar como en la **Tabla 3-1**, sin embargo, más adelante se explicará más detalladamente el agente DDPG pues fue el que se utilizó en el presente trabajo.

Tabla 3-1.

Tipos de agentes incorporado en la caja de herramientas de aprendizaje por refuerzo de Matlab

Agente	Espacio de acción	Espacio de observación
Agente de aprendizaje Q	Discreto	Continuo o discreto
Agente SARSA	Discreto	Continuo o discreto
Agentes de red Q profunda (DQN)	Discreto	Continuo o discreto
Agentes de gradiente de políticas (PG)	Discreto o continuo	Continuo o discreto
Agentes de gradiente de políticas determinísticas profundos (DDPG)	Continuo	Continuo o discreto
Agentes de gradiente de políticas deterministas profundos con retraso doble (TD3)	Continuo	Continuo o discreto
Agentes de Actor-Crítico (AC)	Discreto o continuo	Continuo o discreto
Agentes de optimización de políticas proximales (PPO)	Discreto o continuo	Continuo o discreto

Nota: En la caja de herramientas de aprendizaje por refuerzo se incluyen 8 agentes, cada uno orientado a solucionar cierto tipo de problemas. Fuente. (MathWorks, 2019)

3.1.3 Entorno

El entorno es todo lo que está fuera del agente, es decir, el mundo. Donde el agente envía acciones y de quien recibe recompensas. También es donde se entrena el agente. En aplicaciones de sistemas de control a dicho sistema externo se le conoce mayormente como el conjunto de la planta, los sensores y la señal de referencia.

En el aprendizaje por refuerzo el entorno conforma la mayor parte del sistema, incluyendo la dinámica de este, en realidad, el entorno es todo menos el agente. El agente es una parte del software que genera las acciones y actualiza constantemente la política a través del aprendizaje (como se mencionó anteriormente).

3.1.4 Función de recompensa

El objetivo del aprendizaje por refuerzo es entrenar a un agente para que logre desarrollar una tarea en un entorno dinámicamente desconocido. Este agente recibe observaciones de dicho entorno y una recompensa que numéricamente mide cuán acertada es una acción a fin de cumplir un objetivo general.

Cuando el agente interactúa con el entorno recibe una recompensa numérica que se utiliza para medir el desempeño de cada acción realizada por el agente con respecto a los objetivos de la tarea a realizar. En general la función de recompensa se utiliza para alentar o penalizar las acciones que se realizan. Si la función de recompensa tiene más de una señal de entrada es importante establecer un peso a cada señal individual, de manera similar a cuando se establecen los pesos entre conexiones de las redes neuronales, para determinar en qué medida se afecta el resultado acumulado según la señal.

3.1.5 Acción

En cada uno de los posibles Estados el entorno presenta una posibilidad de acciones a realizar, de las cuales el agente es quien se encarga de elegir una única acción que modificará el entorno cambiándolo de estado, y con ello, obteniendo una recompensa (positiva o negativa). En el presente caso las acciones corresponden a la fuerza que se aplica a cada par de llantas del robot móvil diferencial, esta fuerza corresponde a girar las rueda en un sentido o en otro utilizando la máxima potencia del motor, para fines del programa, estas acciones están acotadas en el rango [-1, 1].

Por ser un robot diferencial dicha fuerza se aplica a cada par de llantas. Utilizando la siguiente notación se definirán las acciones posibles de los robots:

F_d = Fuerza aplicada a las ruedas del lado derecho
F_i = Fuerza aplicada a las ruedas del lado izquierdo

$$F_{traslación} = \frac{1}{2}(F_d + F_i) \quad (1)$$

$$F_{rotación} = \frac{1}{2}(F_d - F_i) \quad (2)$$

Las acciones posibles para cada robot son de traslación y rotación, definidas como sigue:

$\text{Traslación} = +1$: Hacia adelante

$\text{Traslación} = -1$: Hacia atrás

$\text{Rotación} = +1$: sentido contrario a las manecillas del reloj

$\text{Rotación} = -1$: Sentido horario

3.1.6 Política

La política se puede entender como una función que comprende el valor objetivo que tiene escoger una opción. Matemáticamente se podría expresar como:

$$acciones = \text{política}(observaciones\ de\ estado)$$

Esta función es una regla que determina cuál es la acción que se debe realizar según el estado particular en el que se encuentre en entorno.

El objetivo del aprendizaje por refuerzo es encontrar la política adecuada que satisfaga el objetivo particular, pero ¿qué significa una política adecuada? Una política adecuada es aquella que produce una recompensa total mayor, lo cual significa que se están realizando de manera satisfactoria las tareas. Para encontrar la política adecuada se hace uso de un algoritmo de aprendizaje que actualiza constantemente los parámetros de dicha función para encontrar la óptima a fin de maximizar la recompensa recibida a largo plazo. Para conseguirlo, se debe realizar una gran cantidad de simulaciones para entender y "aprender" cómo se afecta la recompensa total si se realiza cierta acción partiendo de un estado particular.

Al ser una función, por lo general se puede representar la política como un aproximador de funciones que, de manera similar a una red neuronal, tiene parámetros ajustables.

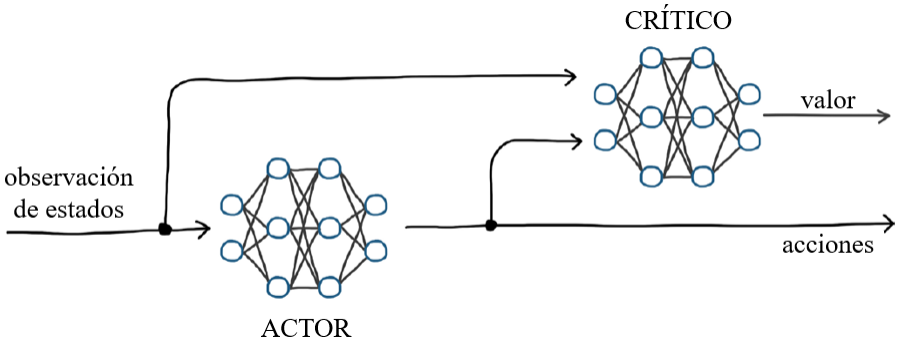
3.1.7 Actor y crítico

En el aprendizaje por refuerzo un agente puede tener uno o más aproximadores de función para entrenar la política, en el presente caso se utilizaron dos aproximadores de funciones (actor y crítico) representados con redes neuronales. El actor es quien toma las

decisiones de qué acción se debe realizar según el estado en que se encuentre el entorno. El crítico es quien valora en qué medida se ve afectada la recompensa acumulada al escoger tal o cual acción partiendo del presente estado, a diferencia del actor, el crítico no selecciona una acción, únicamente evalúa la conveniencia y el coste de escoger la acción estando en una situación concreta.

Figura 3-2

Algoritmo de aprendizaje de actor-crítico



Nota. El actor y el crítico están representados por redes neuronales, el actor toma la decisión de la acción que se debe realizar y utiliza como métrica el valor que le haya dado el crítico al par estado-acción. Fuente. (MathWorks, 2020)

En la **Figura 3-2** se puede observar cómo interactúan el actor y el crítico, allí se muestra que cada uno está representado por una red neuronal de manera independiente. El actor está definido con una red neurona que tiene como parámetros de entrada las observaciones de estados y como salida se tiene la acción que, a largo plazo, maximiza la recompensa acumulada. El crítico se define con una estructura de red neuronal diferente a la del actor, pues esta debe tener una secuencia de capas por cada una de las rutas de entrada (observaciones y acciones) que después se combinan. La capa de salida debe estar totalmente conectada y conformada por una sola neurona que representa el valor esperado para la recompensa a largo plazo (el valor Q).

3.1.8 Ejemplo 1: tabla de valor Q

En la **Tabla *3-2*** se representan 4 estados y las posibles acciones que se pueden escoger partiendo de cada estado. Los valores numéricos representan el coste de escoger cada acción, donde un valor cercano a cero representa una respuesta negativa y un valor cercano a cien representa una respuesta ideal. El agente es el encargado de escoger la acción que se debe realizar, para el Estado 1 el actor puede escoger una acción con coste de 60, 25 o 15, si el actor se entrenó correctamente deberá escoger la Acción 1 (de menor coste), si se encuentra en el Estado 2 debería escoger la Acción 2 y de manera análoga con los demás estados. Cada uno de estos valores o costes fueron asignados por el crítico, quien después de un alto número de pruebas determinó cómo se afecta una recompensa futura al escoger entre las tres acciones partiendo de cada estado.

Tabla 3-2.

Tabla de costes Q para el Ejemplo 1

	Acción 1	Acción 2	Acción 3
Estado 1	60	25	15
Estado 2	20	50	30
Estado 3	25	25	50
Estado 4	5	85	10

Nota: Esta es una tabla que se utiliza para ejemplificar de manera grafica cómo interactúan el actor y el crítico. Fuente. Elaboración propia.

La **Tabla *3-2*** se conoce como una tabla de valores o cantidades (tabla Q en inglés, de quantity). Al empezar el entrenamiento esta tabla puede estar definida con ceros o con valores aleatorios que se irán actualizando (por el crítico) con cada episodio simulado, de esta forma es como el agente aprende los valores del par estado-acción que generan la mayor recompensa acumulada. El algoritmo Q-learning (explicado más adelante) recibe su nombre debido a que utiliza esta tabla como criterio para tomar decisiones.

3.1.9 Agente de aprendizaje Q (Q-learning)

Un agente de Q-learning cuenta con un crítico de valor que predice la recompensa futura, el agente es quien selecciona la acción que produce una recompensa estimada cuyo valor a largo plazo es mayor (MathWorks, n.d.-a).

Este agente se puede entrenar para actuar en entornos con espacios de observación discretos o continuos, sin embargo, solamente pueden entrenarse en espacios de acción discretos, pues representar un número continuo de acciones con una tabla sería poco práctico y costoso computacionalmente (Tearle, n.d.).

3.1.9.1 Aproximador de función crítica

Para valorar cada estado-acción el agente Q cuenta con un crítico $\boldsymbol{Q}(\boldsymbol{S}, \boldsymbol{A}; \boldsymbol{\phi})$, este aproximador de funciones tiene como entrada las observaciones (S) y las acciones (A) y como salida ofrece el valor esperado para la recompensa futura.

Este es un agente cuya función de valor se representa en una tabla, como en el Ejemplo 1. El parámetro (ϕ) corresponde a los valores reales del crítico ($Q(S, A)$) en la tabla.

En el proceso de entrenamiento es el agente quien ajusta los valores de (ϕ) para luego almacenarlos en el crítico ($Q(S, A)$) y utilizarlos como métrica llegado el momento de elegir las mejores acciones.

3.1.9.2 Algoritmo de entrenamiento

El algoritmo que utilizan los agentes de Q-learning se presenta a continuación:

Algoritmo 1 Algoritmo Q-learning (MathWorks, 2019)

Inicializar el crítico $\boldsymbol{Q}(\boldsymbol{S}, \boldsymbol{A}; \boldsymbol{\phi})$ con valores del parámetro $\boldsymbol{\phi}$ aleatorios

Para cada episodio del entrenamiento hacer:

1. Obtenga la observación inicial $\boldsymbol{S}$ del entorno.
2. Repita lo siguiente para cada paso del episodio hasta que $\boldsymbol{S}$ sea un estado terminal
    a. Para la observación actual $\boldsymbol{S}$, seleccione una acción $\boldsymbol{A}$ aleatoria con probabilidad $\boldsymbol{\epsilon}$. De lo contrario, seleccione la acción para la cual la función

de valor crítico sea mayor

$$\boldsymbol{A} = \text{AmaxQ}(\text{S}, \text{A})$$

b. Ejecutar la acción $\boldsymbol{A}$. Observar la recompensa $\boldsymbol{R}$ y la siguiente observación $\boldsymbol{S'}$.

c. Si $\boldsymbol{S'}$ es un estado terminal, establezca la función de valor $\boldsymbol{y}$ en $\boldsymbol{R}$. De lo contrario, configúrelo en

$$\boldsymbol{y} = \boldsymbol{R} + \boldsymbol{\gamma} \boldsymbol{^{max}_{A}} \boldsymbol{Q}(\boldsymbol{S'}, \boldsymbol{A})$$

d. Calcule la actualización del parámetro crítico.

$$\Delta \boldsymbol{Q} = \boldsymbol{y} - \boldsymbol{Q}(\boldsymbol{S}, \boldsymbol{A})$$

e. Actualice el crítico utilizando la tasa de aprendizaje $\boldsymbol{\alpha}$

$$\boldsymbol{Q}(\boldsymbol{S}, \boldsymbol{A}) = \boldsymbol{Q}(\boldsymbol{S}, \boldsymbol{A}) + \boldsymbol{\alpha} \Delta \boldsymbol{Q}$$

f. Establecer la observación $\boldsymbol{S}$ a $\boldsymbol{S'}$

Donde ϵ es el factor de probabilidad que utiliza el agente para seleccionar una acción aleatoria (incentivar la exploración) o una acción conocida (explotación). γ es un factor de descuento, representa en qué medida se afectan a largo plazo las acciones según sea la recompensa futura estimada, se pondera como la recompensa futura descontada.

3.1.10 Agentes de gradiente de políticas deterministas profundos (DDPG)

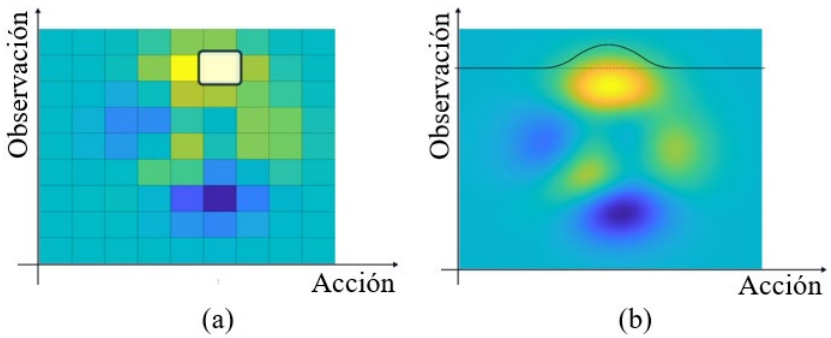
Son un tipo de agente de actor-crítico que se conforman de un actor determinístico y un crítico de valor Q. De forma similar a la de los agentes Q, los agentes DDPG se entrenan en espacios de observación continuos o discretos, sin embargo, se entrenan en espacios de acción continuos.

Cuando se tienen acciones finitas (uso de agente Q), el crítico puede establecer una representación de valor para cada par estado-acción y será más fácil para el actor determinar la acción que producirá una mayor recompensa partiendo de un estado específico, como en la (**Figura 3-3(a)**). No obstante, si los valores no son discretos, resultaría computacionalmente costoso representar los valores en una tabla. Por ello, los agentes DDPG utilizan redes neuronales para representar la función de valor, y para este caso,

escoger la acción que genere la mayor recompensa futura significa encontrar el valor máximo dentro de la superficie de valor (**Figura 3-3(b)**).

Figura 3-3

Tabla de valor Q (a), función de valor Q (b)



Nota. Representación de una tabla de valor Q para acciones discretas (a). Representación de una función de valor Q para un agente DDPG (b). Fuente. (Tearle, n.d.)

3.1.10.1 Aproximador de función de política y valor

Un agente DDPG cuenta con dos aproximadores de función; el actor y el crítico, el actor se representa con una red neuronal que tiene como entrada las observaciones y como salida las acciones, el crítico cuenta dos datos diferentes de entrada, las observaciones y las acciones, por ello se debe representar con una red compuesta por dos capas independientes que posteriormente se unen para generar solamente un valor de salida (el valor Q). Los aproximadores se representan como sigue:

- Actor $\mu(S;\theta)$: El actor recibe como entrada la observación S y como salida entrega la acción que generará la máxima recompensa futura (θ).
- Crítico $Q(S,A;\phi)$: El crítico recibe las observaciones y las acciones como parámetros de entrada y devuelve el valor estimado para la recompensa a largo plazo (ϕ).

En el proceso de entrenamiento el agente va ajustando los valores de (θ) para luego almacenarlos en el actor ($\mu(S)$).

3.1.10.2 Algoritmo de entrenamiento

El algoritmo que utilizan los agentes DDPG se presenta a continuación:

Algoritmo 2 Algoritmo DDPG (MathWorks, 2019)

Inicializar el crítico $\boldsymbol{Q(S, A; \phi)}$ con valores del parámetro $\boldsymbol{\phi}$ aleatorios

Inicializar el actor $\boldsymbol{\mu(S; \theta)}$ con valores del parámetro $\boldsymbol{\theta}$ aleatorios

Para cada episodio del entrenamiento hacer:

1. Para la observación actual $\boldsymbol{S}$, seleccione una acción $\boldsymbol{A = \mu(S) + N}$.
2. Ejecutar la acción $\boldsymbol{A}$. Observar la recompensa $\boldsymbol{R}$ y la siguiente observación $\boldsymbol{S'}$.
3. Almacene la experiencia $\boldsymbol{(S, A, R, S')}$ en el búfer de experiencia.
4. Tomar un lote pequeño de muestras aleatorio de $\boldsymbol{M}$ experiencias $\boldsymbol{(S_i, A_i, R_i, S_i')}$ del búfer de experiencia.
5. Si $\boldsymbol{S_i'}$ es un estado terminal, establezca la función de valor $\boldsymbol{y_i}$ en $\boldsymbol{R_i}$. De lo contrario, configúrelo en

$$\boldsymbol{y_i = R_i + \gamma Q'(S_i', \mu'(S_i'; \theta); \phi)}$$

6. Actualice los parámetros del crítico minimizando la pérdida $\boldsymbol{L}$ en cada experiencia muestreada

$$\boldsymbol{L_k = \frac{1}{M} \sum_{i=1}^{M} (y_i - Q(S_i, A_i; \phi))^2}$$

7. Actualice los parámetros del actor haciendo uso del siguiente gradiente de política de muestra para maximizar la recompensa esperada con descuento

$$\boldsymbol{\nabla_\theta J \approx \frac{1}{M} \sum_{i=1}^{M} G_{ai}\, G_{\mu i}}$$

$\boldsymbol{G_{ai} = \nabla_A Q(S_i, A; \phi)}$ Donde $\boldsymbol{A = \mu(S_i; \theta)}$

$\boldsymbol{G_{\mu i} = \nabla_\theta \mu(S_i; \theta)}$

8. Actualiza los parámetros del actor y del crítico según el método de actualización de destino.

Donde N es el ruido estocástico que introduce el modelo de ruido. γ es un factor de descuento, representa en qué medida se afectan a largo plazo las acciones según sea la recompensa futura estimada, se pondera como la recompensa futura descontada. G_{ai} es el

gradiente de la salida del crítico con respecto a la acción calculada por la red del actor, y $G_{\mu i}$ es el gradiente de la salida del actor con respecto a los parámetros del actor.

Para más información sobre los tipos de agentes disponibles en el software Matlab consultar el **Anexo C**.

3.1.11 *Ejemplo 2: implementación de aprendizaje por refuerzo*

El ejemplo que se expone a continuación está basado en la referencia (Tearle, n.d.)

3.1.11.1 Definir el entorno

Para un robot móvil con tracción diferencial, se establecen 6 variables de estado observables continuas que corresponden a la posición (x, y), la orientación $(\sin(\theta), \cos(\theta))$ y la velocidad (v, ω). Y dos acciones continuas que corresponden a la fuerza traslacional y rotacional.

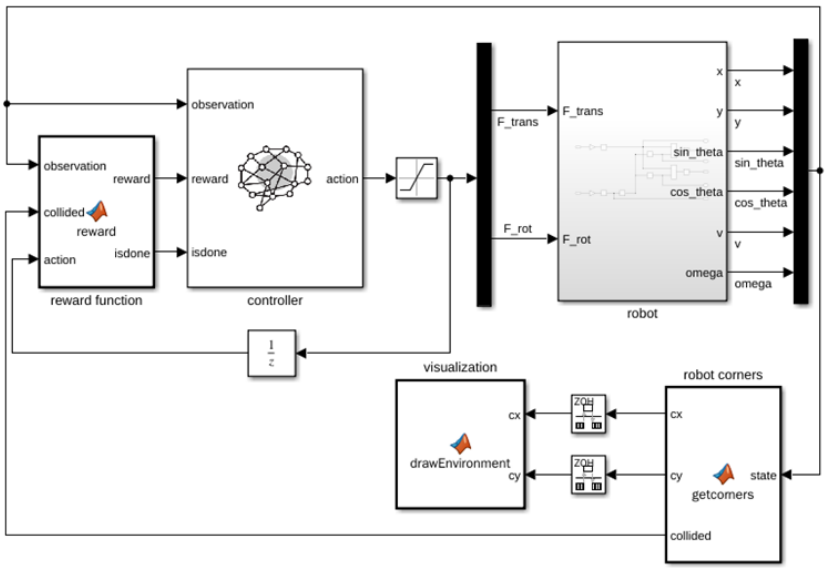
Para definir la variable que representa cada observación de los 6 estados se utiliza la función `rlNumericSpec` cuyo parámetro de entrada es un vector de igual tamaño al número de los estados ([6 1]). La variable definida para representar las acciones se establece con parámetros similares a los utilizados en la variable de observaciones con la misma función, en este caso, el tamaño del vector corresponde a las acciones ([2 1]), adicionalmente se pueden utilizar los parámetros "UpperLimit" y "LowerLimit" con el fin de limitar las acciones en un rango específico.

Para definir el entorno de aprendizaje se utiliza la función `rlSimulinkEnv`, como parámetros se le entrega la ruta de acceso al bloque de aprendizaje por refuerzo direccionándola a partir del nombre del modelo (nombre del archivo de Simulink), la variable de las observaciones y la variable de las acciones.

En la **Figura *3-4*** se observa el modelo del entorno de aprendizaje, compuesto por todos los bloques excepto el del agente (con nombre "controller").

Figura 3-4

Dinámica del robot diferencial implementada en Simulink



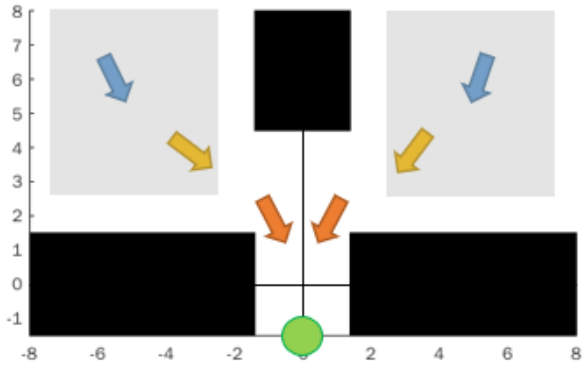
Nota. Modelo del entorno y el agente para un robot implementado en Simulink. Fuente. (Tearle, n.d.)

3.1.11.2 Definir la recompensa

En la **Figura *3-5*** se puede observar un ejemplo del entorno donde se entrena el agente y la posición objetivo en color verde. Las zonas en color gris delimitan la zona donde el robot puede aparecer de manera aleatoria.

Figura 3-5

Entorno donde se entrena el agente con un robot



Nota. El objetivo del robot es llegar a la posición (0,-1.5). Fuente. Elaboración propia con base en (Tearle, n.d.)

La recompensa se puede definir partiendo de la posición objetivo para el robot, en primera instancia se espera que el robot se dirija hacia la posición 0 en el eje x, para ello, se utiliza una función que tenga como salida su valor máximo si x es igual a 0, esto se puede conseguir utilizando una exponencial gaussiana con el exponente negativo, en esta configuración el valor máximo de la función es 1. La forma básica de la función es la presentada en la ecuación (3). Gráficamente la función respecto a x se puede observar la **Figura *3-6***.

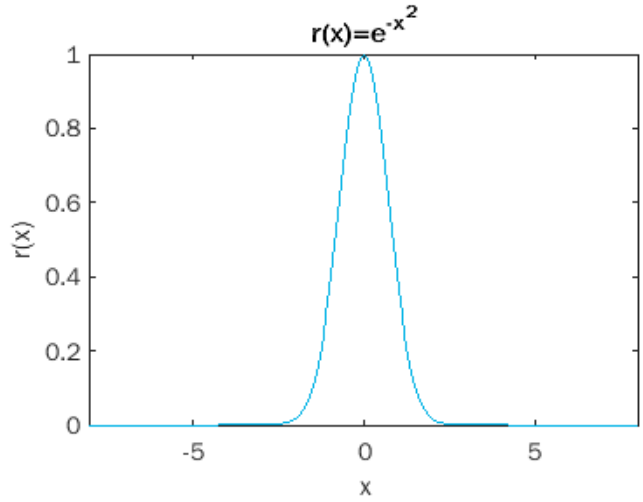
$$r(x) = e^{-x^2} \tag{3}$$

Para lograr que en el eje y se incentive a ir hacia una posición negativa se puede utilizar una función exponencial decreciente, esta se presenta en la ecuación (4). Gráficamente se puede observar en la **Figura *3-7***.

$$r(y) = e^{-y} \tag{4}$$

Figura 3-6

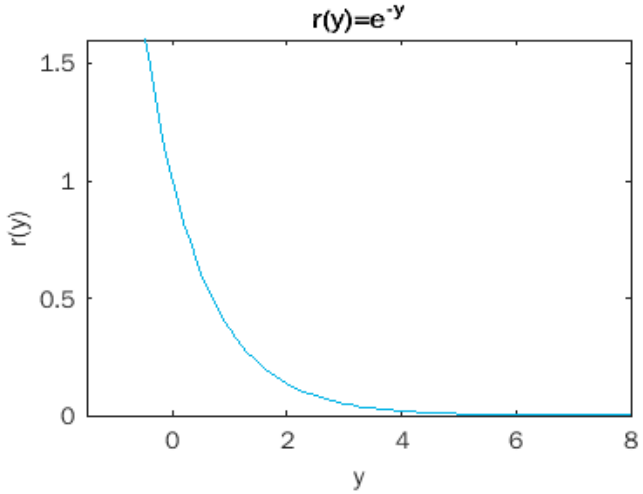
Recompensa del robot para el eje x



Nota. La gráfica se realizó con valores desde -8 hasta 8 con pasos de 0.1. Fuente. Elaboración propia con base en (Tearle, n.d.)

Figura 3-7

Recompensa del robot para el eje y



Nota. La gráfica se realizó con valores desde -1.5 hasta 8 con pasos de 0.1. Fuente. Elaboración propia con base en (Tearle, n.d.)

Para incitar al robot a que se dirija hacia la posición ((x, y) = (0, 0)) y luego hacia la posición (y < 0) se pueden sumar las dos funciones, que logran hacerlo en cada eje de forma independiente, como se observa en la ecuación (5).

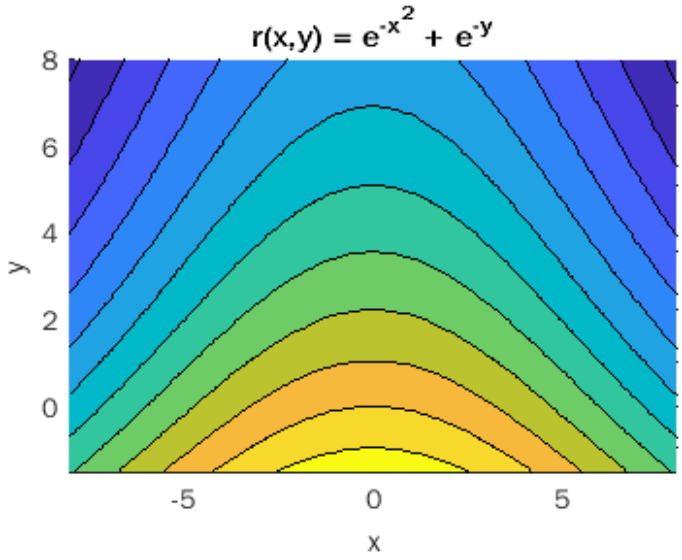
$$r = c_1 e^{-ax^2} + c_2 e^{-by} + c_3 \tag{5}$$

Los coeficientes c_1, c_2, a y b se pueden utilizar para dar forma a la función. El coeficiente c_3 se establece para evitar la "recompense acumulativa positiva" (Tearle, n.d.), esto sucede si el robot recibe recompensas positivas antes de llegar a la posición objetivo, el agente podría aprender a dirigir al robot hacia dicha posición y permanece allí, por ello, este último coeficiente se debe ajustar para que la recompensa sea negativa en todo momento excepto al estar cerca del objetivo donde debería ser un valor aproximado a cero.

En la **Figura *3-8*** se aprecia la forma de la función de recompensa, los colores cálidos representan valores de recompensa altos y los fríos representan recompensas bajas.

Figura 3-8

Forma de la función de recompensa



Nota. La forma de la función muestra que tiene valores más altos cerca a la posición 0 en x y valores negativos en y. Fuente. Elaboración propia

3.1.11.3 Definir al agente

3.1.11.3.1. Crear el actor

Utilizando un agente (DDPG) que está conformado por un actor determinístico y un crítico de valor Q. Cada uno de ellos definido con redes neuronales cuya estructura se presenta en la **Figura 3-9**.

Para definir la red neuronal del actor se utiliza una capa de entrada con el número de neuronas correspondientes al número de estados observables, posteriormente utilizando una capa totalmente conectada con 100 neuronas y una capa ReLu, se pueden repetir las capas según sea necesario debido a la complejidad del problema. Finalmente se utiliza una capa totalmente conectada con 2 neuronas que corresponde al número de acciones (Fuerza de traslación y Rotación) y una capa tanh (tangente hiperbólica) para establecer la salida dentro de los límites -1 y 1.

La taza de aprendizaje se estableció en $5 * 10^{-5}$. Para actualizar los valores de la ganancia (w) y el bias (b) se utiliza el método del descenso del gradiente (Ng, n.d.), para el presente ejemplo se establece el umbral en 10. Estos parámetros se especifican en la función

rlRepresentationOptions que posteriormente se utiliza como opciones al crear el actor.

Finalmente, para crear el actor se utiliza la función *rlDeterministicActorRepresentation*, en sus parámetros de entrada se especifica la matriz donde se definieron las capas de la red neuronal, la variable que almacena las observaciones, la variable donde se almacenan las acciones, el nombre de la capa de entrada de las observaciones, el nombre de la capa de salida y las opciones (establecidas anteriormente y al almacenadas en una variable).

3.1.11.3.2. Crear el crítico

Para crear el crítico se realiza un procedimiento similar al utilizado para crear al actor; Inicialmente se define una red neuronal con una capa de entrada de 6 neuronas, correspondientes al número de estados observables, posteriormente se utilizó una capa totalmente conectada con 100 neuronas y una capa ReLu, Se añadieron varios duplicados de estas últimas capas. Finalmente se utilizó una capa totalmente conectada con 1 neurona que corresponde al valor Q.

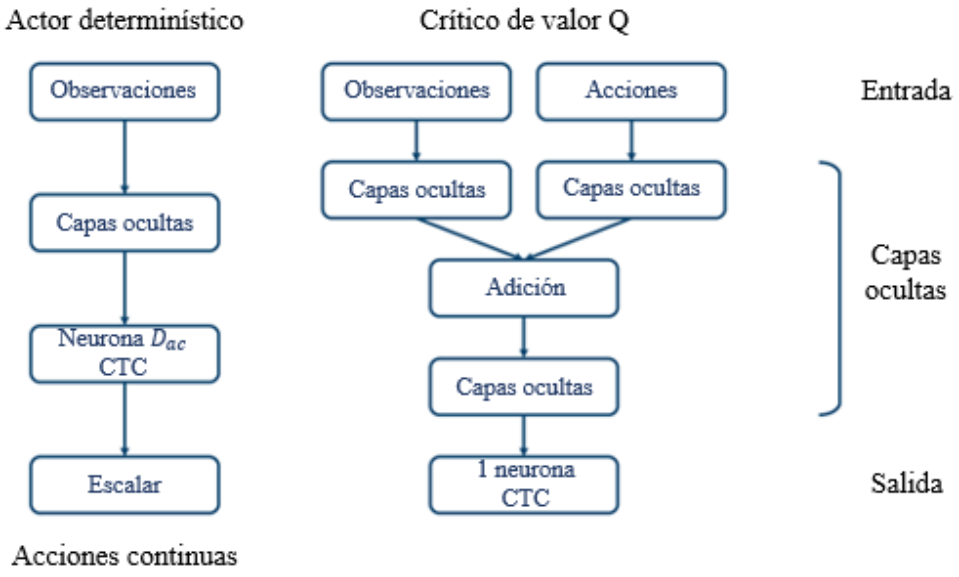
La otra red neuronal independiente se define con una capa de entrada con 2 neuronas que corresponden a las 2 acciones (Fuerza de traslación y Rotación) y una capa totalmente conectada de 100 neuronas. Esta última capa se conecta a la otra entrada de la red neuronal de adición.

Por último, para crear el crítico se utiliza la función *rlQValueRepresentation*, en sus parámetros de entrada se especifica la matriz donde se definieron las capas de la red neuronal, la variable que almacena las observaciones, la variable donde se almacenan las acciones, el nombre de la capa de entrada de las observaciones, el nombre de la capa de entrada de las acciones y las opciones (se utilizaron las mismas que en el actor).

3.1.11.3.3. Crear el agente

Figura 3-9

Estructura de las redes neuronales para el actor y el crítico



Nota. La red neuronal del actor cuenta con una capa totalmente conectada (CTC) y una salida escalar que corresponde a las acciones. El crítico cuenta con una estructura de red (DAG, del inglés directed acyclic graph), puesto que se conforma por dos redes neuronales independientes pero unidas en una neurona (Adición). Fuente. (Tearle, n.d.)

Para crear un agente DDPG se utiliza la función rlDDPGAgente que recibe como parámetro el actor, el crítico y las opciones del agente, estas últimas definidas a continuación:

- Tiempo de muestreo: 0.25
- Longitud del búfer de experiencia: $1 * 10^6$
- Tamaño del mini lote: 128

3.1.11.4 Entrenar el agente

Para realizar el entrenamiento se utiliza la función `train`. Como parámetros de

entrada recibe el agente, el entorno y las opciones de entrenamiento, estas últimas se implementaron con los siguientes parámetros en el ejemplo:

- Pasos máximos por episodio: 120
- Número máximo de episodios: 50000
- Duración media de la ventana de tiempo: 50
- Criterios de detención de entrenamiento: Recompensa promedio
- Valor para detener el entrenamiento: 2

3.1.11.5 Evaluar el agente

Para evaluar el agente se simularon 20 episodios con el agente entrenado, esto utilizando la función *sim*. La **Figura 3-10** muestra el proceso de entrenamiento, este se detuvo en el episodio 28520 puesto que se cumplió el criterio para detener el entrenamiento por la recompensa promedio cuyo valor final fue de 2,1435.

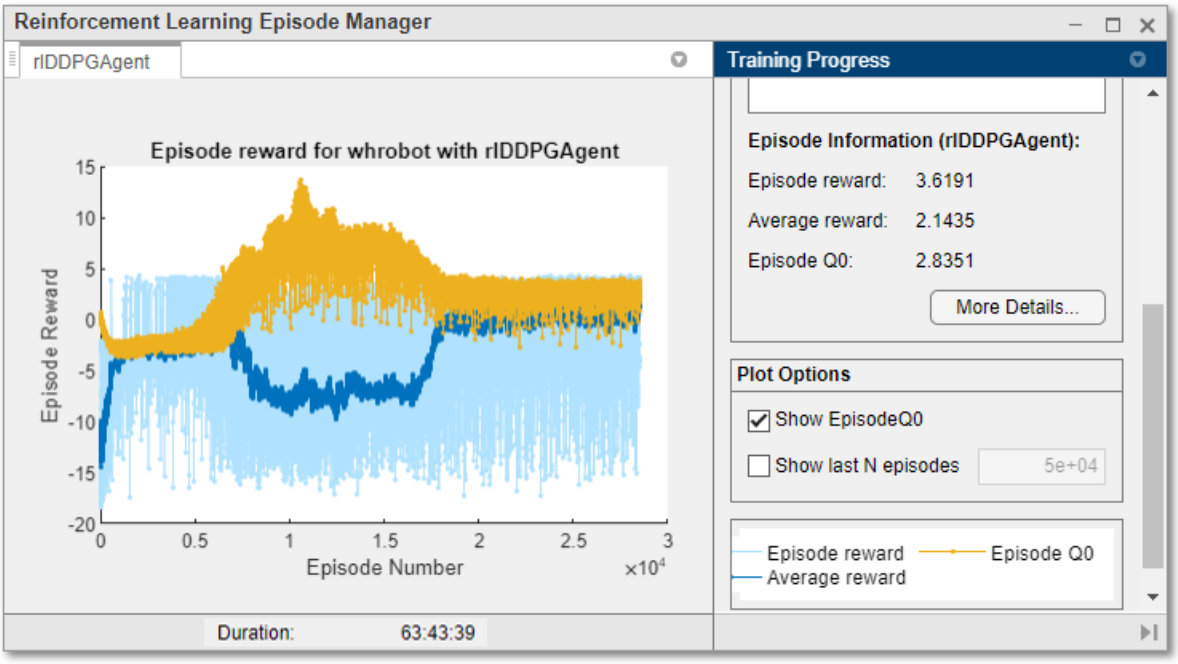
La función *sim* entrega como salida una matriz de estructuras donde cada fila corresponde al episodio. Cada episodio tiene la información de las observaciones de cada paso, el valor de las acciones, la recompensa y si el episodio se terminó por colisión, se cumplió el objetivo o por que se alcanzó el número máximo de pasos.

Después de determinar cuáles episodios terminaron, comparando con el número de episodios ejecutados, se determinó cuáles fueron exitosos obteniendo los siguientes resultados:

- Número de episodios completados: 20
- Número de colisiones: 0
- Episodios que no se completaron: 0

Figura 3-10

Resultado del entrenamiento del agente para un robot

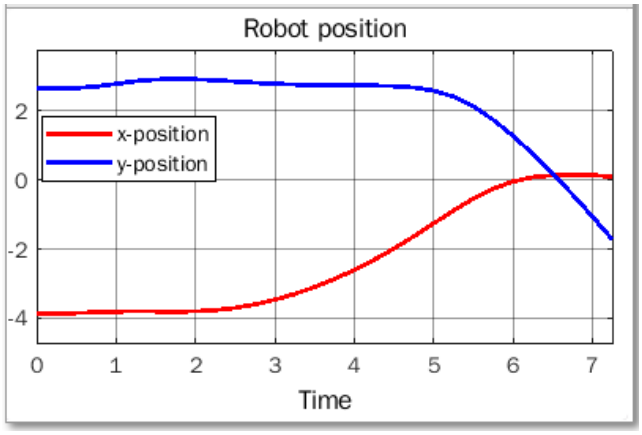


Nota. El resultado del entrenamiento se almacena como una estructura y se puede visualizar gráficamente la recompensa promedio (naranja), la recompensa por episodio (azul) y el valor estimado para la recompensa futura (amarillo), esto con la función `inspectTrainingResult`. Fuente. Elaboración propia.

En la **Figura 3-11** se observa la posición del robot para un solo episodio con el agente entrenado, en esta se observa que el valor final está alrededor de las coordenadas (0,-1.5).

Figura 3-11

Posición del robot para un episodio



Nota. Trayectoria del robot partiendo de una posición aleatoria y yendo hacia la posición (0,-1.5) con el agente entrenado. Fuente. Elaboración propia

3.2 Robot Diferencial

El término robótica apareció por primera vez en una obra literaria importante escrita en el año 1817 por Mary Shelley y que se titula "Frankenstein". El término robot, del checo "robota" – que significa servidumbre – se introdujo por primera vez en la obra "Rossum's Universal Robots" del escritor checoslovaco Karel Capek publicada en 1917, sin embargo, fue Isaac Asimov Asimov quien hizo aparecer por primera vez en alguna de sus obras una máquina mecánica bien diseñada y que garantizaba cumplir con las tres leyes de la robótica que él mismo escribió (Contreras, 2003; Kubánková, 2020; Salazar, n.d.), y son:

1. "Un robot no puede actuar contra un ser humano o, mediante la inanición, permitir que un ser humano sufra daños"
2. "Un robot debe obedecer las órdenes dadas por los seres humanos, salvo que estén en conflicto con la primera ley"
3. "Un robot debe proteger su propia existencia, a no ser que este en conflicto con las dos primeras leyes"

En los años recientes la robótica se ha constituido por un amplio conjunto de disciplinas y, por ende, bastantes campos de estudio y aplicación, sin embargo, sus bases están fundamentadas en el trabajo conjunto de la mecánica, electrónica y computación. En especial esta última ha tenido mayor crecimiento, pues el término de inteligencia artificial se ha introducido como una parte muy importante en aplicaciones de robótica (IAT, n.d.).

Debido a la gran variedad de campos de aplicación existen diferentes tipos de robot que se pueden clasificar según sus grados de libertad, su lenguaje de programación o su arquitectura. Según esta última, se define el robot móvil como un sistema con la presencia de una base móvil (con ruedas, orugas o patas) que otorga libertar al robot para desplazarse libremente a través del entorno (Siciliano et al., 2009).

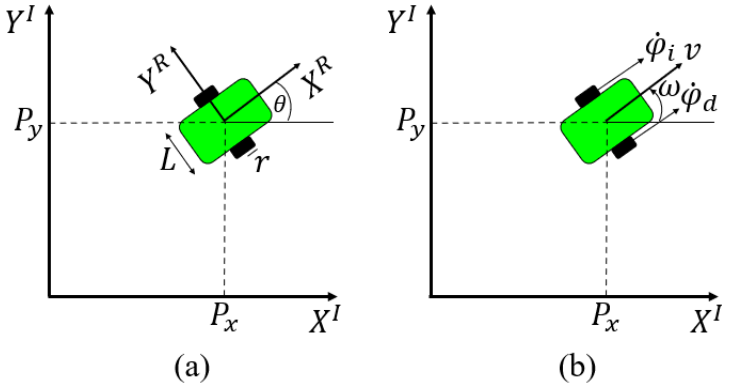
3.2.1 Cinemática del robot móvil diferencial

La información que se presenta a continuación se basa en las referencias (Bañó, 2003; Mezeaa, n.d.; Solaque et al., 2014)

Para el modelamiento cinemático del robot móvil es necesario conocer su desplazamiento y dirección ocasionado por la acción de las ruedas, para ello se utilizarán las dos representaciones de la **Figura *3-12***.

Figura 3-12

Diagrama de configuración para el robot móvil con tracción diferencial



Nota. El plano (a) muestra las variables físicas del robot móvil con tracción diferencial. El plano (b) muestra las variables que definen el moviemiento del robot. Fuente. Elaboración propia

En la **Figura *3-12*(a)** se observa el robot móvil diferencial con su marco de referencia de coordenadas locales (X^R, Y^R) ubicado sobre el plano inercial de coordenadas globales (X^I, Y^I), también se muestra la ubicación del punto medio del robot (P_x, P_y) y el ángulo de giro (θ), ello con respecto al plano inercial, también se muestra la longitud entre los centros de las ruedas (L) y el radio de las ruedas (r). En el plano de la **Figura *3-12*(b)** se representa la velocidad traslacional del robot (v), la velocidad angular de la rueda izquierda y derecha ($\dot{\varphi}_i$ y $\dot{\varphi}_d$ respectivamente) y la velocidad angular (ω) del robot.

La velocidad traslacional del robot (v) está definida por el promedio de las velocidades individuales de cada rueda y es proporcional al radio (r) de las ruedas, esto se define matemáticamente en la ecuación (6).

$$v = \left(\frac{\dot{\varphi}_i + \dot{\varphi}_d}{2}\right) r \tag{6}$$

Para lograr una rotación del robot sin conseguir que se desplace, las velocidades de cada rueda deben tener la misma magnitud, pero en dirección opuesta, por ello, la velocidad angular se define como la velocidad de las ruedas sobre la distancia que hay entre ellas (L). Al igual que la velocidad traslacional, este movimiento es proporcional al radio de las ruedas, matemáticamente se define en la ecuación (7).

$$\omega = \left(\frac{\dot{\varphi}_i - \dot{\varphi}_d}{L}\right) r \tag{7}$$

La configuración que describe la pose del robot tiene dos parámetros: velocidad, con sus componentes en (x, y), y la velocidad angular, esto se define en (8) respecto al plano del robot. La componente y de la velocidad es cero ya que el desplazamiento del robot está orientado sobre el eje x.

$$\dot{q}^R = \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} v \\ 0 \\ \omega \end{bmatrix} \tag{8}$$

Aplicando la matriz de rotación respecto al eje global (Z^I) que se expresa en la referencia (Siciliano et al., 2009), se obtiene la configuración de la pose respecto al plano inercial (X^I, Y^I), expresado en la ecuación (9).

$$\dot{q}^I = \begin{bmatrix} \dot{x} \\ \dot{y} \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \sin(\theta) \end{bmatrix} \begin{bmatrix} v \\ 0 \end{bmatrix} \tag{9}$$

En el caso de 2D la velocidad angular representada respecto al plano local es la misma respecto al plano global. Partiendo de ello y expandiendo la ecuación (9) obtiene la ecuación (10).

$$\begin{aligned} \dot{x} &= v * \cos(\theta) \\ \dot{y} &= v * sen(\theta) \\ \dot{\theta} &= \omega \end{aligned} \tag{10}$$

Reemplazando las ecuaciones (6) y (7) en (10) y, aplicando algebra, se obtiene la representación matricial de las ecuaciones que describen el movimiento del robot, en (11).

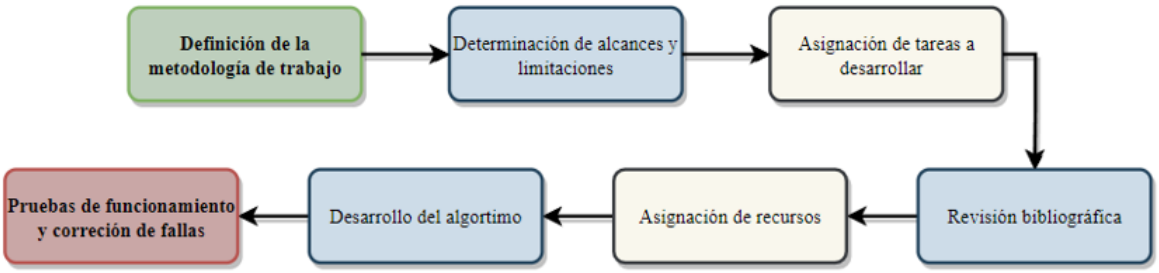
$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} \frac{r}{2} cos(\theta) & \frac{r}{2} cos(\theta) \\ \frac{r}{2} sen(\theta) & \frac{r}{2} sen(\theta) \\ \frac{r}{L} & -\frac{r}{L} \end{bmatrix} \begin{bmatrix} \dot{\varphi}_i \\ \dot{\varphi}_d \end{bmatrix} \tag{11}$$

4. Metodología de Desarrollo

El proceso metodológico que se llevó a cabo para lograr cumplir los objetivos del presente proyecto se presenta en la **Figura 4-1.**

Figura 4-1

Proceso metodológico para la implementación del algoritmo



Nota. La imagen muestra la secuencia lógica de las etapas llevas a cabo para el desarrollo del proyecto. Fuente. Elaboración propia

4.1 Determinación del alcance y limitaciones

El proyecto tenía como fin el diseño de un controlador basado en el algoritmo de aprendizaje por refuerzo Q-learning para conseguir que dos robots móviles navegaran de forma autónoma a través de un entorno simulado compuesto por bloques, cuya estructura representa obstáculos que deben ser evitados por los robots en busca de aprender la trayectoria que los condujera hacia una ubicación deseada y que estuviese definida a través de una función de recompensa.

La culminación exitosa de este proyecto abre grandes posibilidades para continuar investigando en la implementación de algoritmos de inteligencia artificial junto con procesos de control a fin de automatizar los algoritmos implementados en la actualidad, logrando aprender trayectorias cada vez más complejas, haciendo uso de controladores más robustos o logrando vincular diferentes técnicas de IA con otros tipos de robot (Díaz Pinzón, 2018) .Además, los resultados obtenidos en el proyecto pueden ser utilizados para realizar una

comparación sobre la eficacia y confiabilidad de los modelos simulados vs los modelos implementados de manera física si se llegase a construir un prototipo.

4.2 Asignación de tareas a desarrollar

Habiendo definido los objetivos y el alcance del proyecto, se definió un cronograma acorde a la resolución de cada objetivo general, sin embargo, debido a la capacidad computacional con la que se contaba, se dilataron los tiempos en gran medida.

Esta etapa resultó de gran importancia pues, aunque los tiempos definidos para del desarrollo se movieron, la asignación de tareas y sus tareas fue adecuada pues permitió lograr los objetivos del proyecto.

4.3 Revisión bibliográfica

La experiencia que se tenía previamente sobre inteligencia artificial se aprendió en algunas materias cursadas durante la carrera, específicamente, reconocimiento de patrones y en la materia de anteproyecto. Posteriormente, la revisión bibliográfica inició por realizar un curso sobre aprendizaje por refuerzo ofreció por el software Matlab en su página oficial de MathWorks, seguidamente se realizó una búsqueda de artículos y trabajos de grado, así como libros donde se habla principalmente de aprendizaje por refuerzo y, en una proporción más pequeña pero no menos importante, libros sobre robótica.

4.4 Asignación de recursos

Esta fue una de las etapas más importantes y críticas durante el desarrollo del proyecto, pues inicialmente se contaba con un equipo computacional con especificaciones promedio pero que no eran suficientes para obtener los resultados en el plazo establecido inicialmente. Por ello, se buscó la ayuda de otra persona con un computador más potente en cuanto a procesamiento y memoria RAM, Y fue en este donde se desarrolló la mayor parte del proyecto, sin embargo, los resultados finales se obtuvieron en un computador con especificaciones significativamente mayores a este último, pues el modelo final de los agentes entrenados se obtuvieron en un computador de escritorio con 16 gigas de memoria

RAM, una tarjeta gráfica dedicada de 12 gigas y un procesador AMD Ryzen 5 de 12 CPUs a 4.2GHz.

4.5 Desarrollo del algoritmo

En eta etapa se tomó como referencia el algoritmo desarrollado durante el curso de aprendizaje por refuerzo ofrecido por la plataforma de MathWorks. Se tomó este modelo y se modificó para implementarlo en un entorno multiagente con el fin de cumplir los objetivos del proyecto.

4.6 Pruebas de funcionamiento y corrección de fallas

Como etapa final, se probó el algoritmo de aprendizaje por refuerzo y se verificó que los agentes aprendieron a conducir a los robots hacia la posición deseada partiendo de una posición aleatoria, sin embargo, hubo que hacer un alto número de pruebas haciendo pequeñas modificaciones y observando cómo ello afectaba el tiempo de entrenamiento y la capacidad de aprendizaje de los agentes.

5. Estados y Acciones Para el Sistema Multiagente

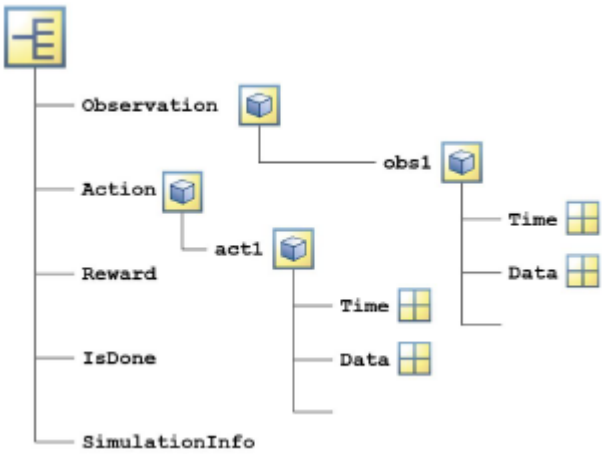
En Matlab se pueden crear redes neuronales utilizando la caja de herramientas de aprendizaje profundo (Deep Learning Toolbox). Para definir las capas de la red se puede hacer en una matriz donde cada elemento es una variable que representa una capa de la red. Las capas cuentan con ciertos parámetros como el tipo de capa, el número de neuronas, el nombre de la capa, entre otros.

Si bien existe una gran variedad de capas para construir redes neuronales y que están disponibles en la caja de herramientas de aprendizaje profundo, para aplicar aprendizaje por refuerzo, hay un número de capas limitado y que se exponen en la **Tabla 11-1** del **Anexo A**.

En aprendizaje por refuerzo desarrollado con el software Matlab una gran parte de los datos se almacena en estructuras, como se observa en la **Figura 5-1**. Esta estructura corresponde a la salida de la función `sim` para un episodio de prueba y con un solo agente, pues al utilizar múltiples agentes se obtendrá una estructura por cada uno de ellos y se almacenarán como una matriz de estructuras. Cada estructura con el valor de las observaciones, las acciones realizadas, la recompensa obtenida por cada paso de tiempo y si el objetivo de ese agente en específico se cumplió o no.

Figura 5-1

Ejemplo de una estructura de datos

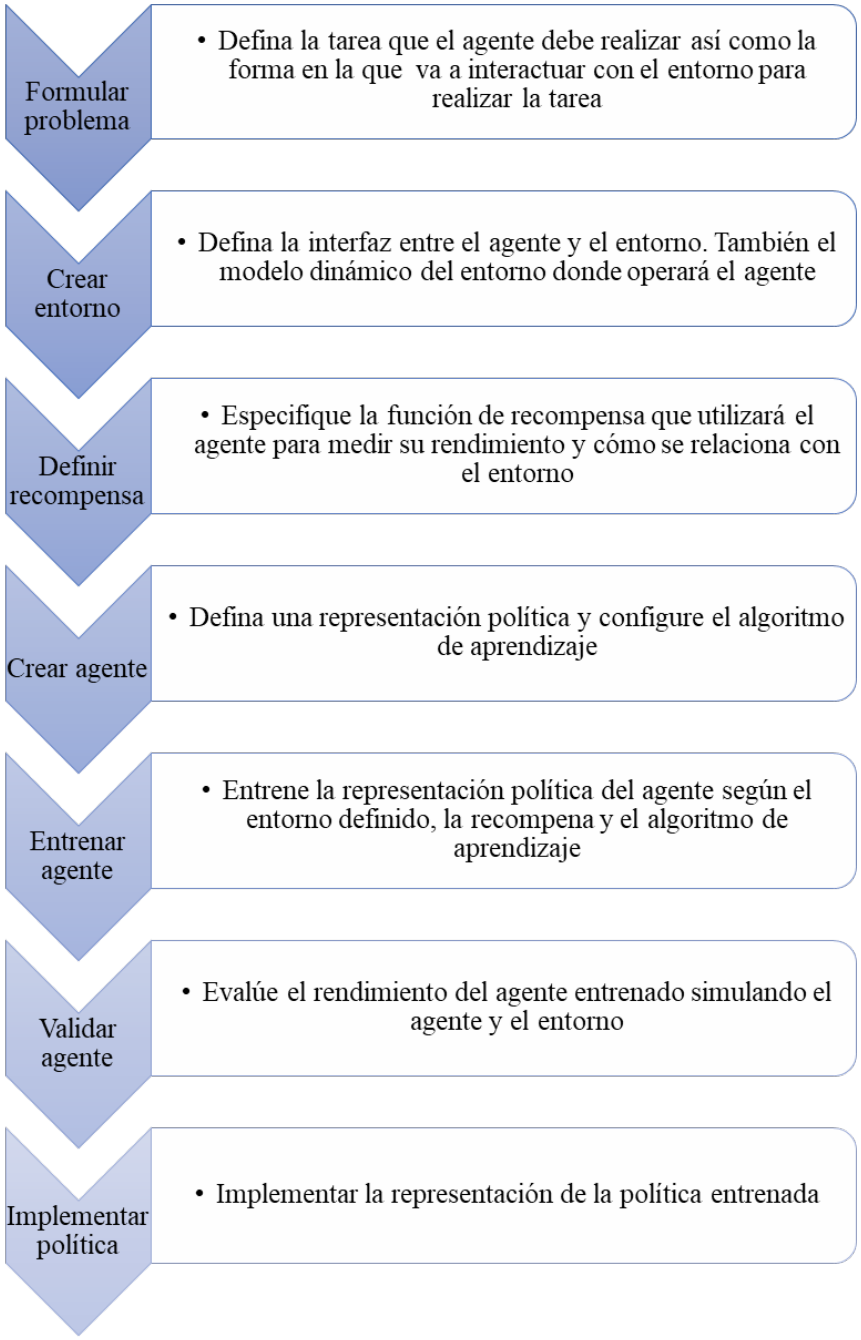


Nota. Este es un ejemplo la estructura de observaciones y acciones que se obtiene al utilizar la función `sim` de Matlab. Fuente. (Tearle, n.d.)

En la **Figura *5-2*** se muestra el diagrama de flujo general que se utilizó como guía para implementar el aprendizaje por refuerzo utilizando el software Matlab y Simulink:

Figura 5-2

Diagrama de flujo para aplicar aprendizaje por refuerzo



Nota. Fuente. Elaboración propia basada en (MathWorks, n.d.-b; Tearle, n.d.)

Al igual que para el modelo con un solo robot, los dos robots móviles con tracción diferencial tendrán 6 variables de estado observables continuas que corresponden a la posición (x, y), la orientación $(\sin(\theta), \cos(\theta))$ y la velocidad (v, ω). Y dos acciones continuas (para cada robot) que corresponden a la fuerza traslacional y rotacional, como en el Ejemplo 2.

Para definir las variables que representa cada observación de los 6 estados se utilizó la función *rlNumericSpec* cuyo parámetro de entrada es un vector de igual tamaño al número de los estados ([6 1]), las dos variables cuentan con un identificador que las diferencia para cada uno de los agentes. Las variables definidas para representar las acciones se establecieron con parámetros similares a los utilizados en las variables de observación utilizando la misma función, en este caso, el tamaño de cada vector corresponde a las acciones ([2 1]) disponibles para cada agente, adicionalmente se utilizó el parámetro "UpperLimit" y "LowerLimit" establecidos en 1 y -1 respectivamente.

Para definir el entorno de aprendizaje se utilizó el modelo dinámico del robot que se presenta en la **Figura 7-2**. También se utilizó la función rlSimulinkEnv, como parámetros se le entregó el nombre del modelo (nombre del archivo de Simulink) junto con la ruta de acceso a los bloques de aprendizaje por refuerzo, junto a ello, un vector que contenía las observaciones (concatenadas) de cada robot, al igual que las acciones.

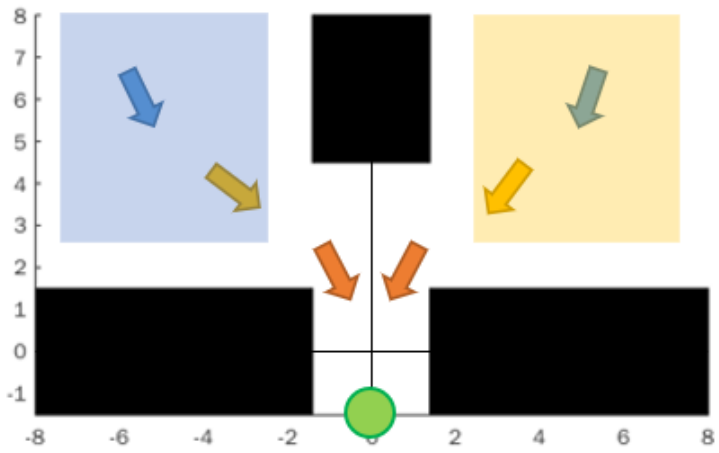
Finalmente, se utilizó una función de inició aleatoria para que, al empezar cada episodio, de entrenamiento o de verificación, los robots inicien en una posición aleatoria (pero limitada dentro de un área para cada uno) y una orientación aleatoria.

6. Función de Recompensa

En la **Figura 6-1** se puede observar el entorno donde se entrenaron los agentes y la posición objetivo para los dos robots en color verde. Debido a problemas de rendimiento del pc donde se entrenó este modelo, se estableció como restricción a cada robot que apareciera en una posición aleatoria (al iniciar un nuevo episodio) en una de las dos zonas delimitadas por el color azul y amarillo, en este sentido, el robot móvil nombrado como "robot" aparecía en una ubicación aleatoria únicamente dentro la zona delimitada en color azul, el robot móvil nombrado como "robot1" aparecía en una posición aleatoria dentro de la zona delimitada en color amarillo, sin embargo, la posición objetivo era la misma para los dos.

Figura 6-1

Entorno donde se entrenaron los dos agentes



Nota. El objetivo para los dos robots es llegar a la posición (0,-1.5). Fuente. (Tearle, n.d.)

La recompensa se definió partiendo de la posición objetivo para el robot, en primera instancia se deseaba que el robot se dirigiese hacia la posición 0 en el eje x, para ello, se utilizó una función que tuviese como salida su valor máximo si x es igual a 0, esto se consiguió utilizando una exponencial gaussiana con el exponente negativo. En esta configuración el valor máximo de la función es 1, la función implementada es la presentada

en la ecuación (12) y se tomó como base la función expuesta en el ejemplo 2. Gráficamente la función respecto a x se puede observar en la **Figura 6-2**.

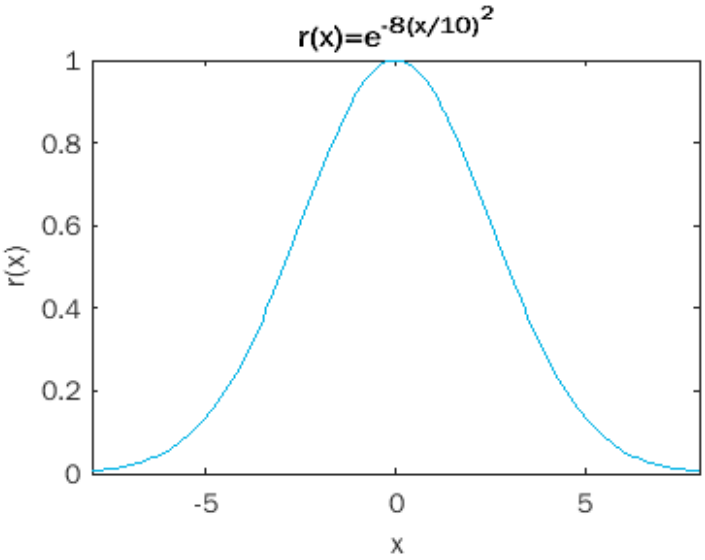
$$r(x) = e^{-8*(x/10)^2} \quad (12)$$

Para lograr que en el eje y se incentivara ir hacia la ubicación -1.5 se utilizó una función exponencial decreciente (al igual que en el ejemplo 2), esta se presenta en la ecuación (13). Gráficamente se puede observar en la **Figura 6-3**.

$$r(y) = e^{-3*(y/10)} \quad (13)$$

Figura 6-2

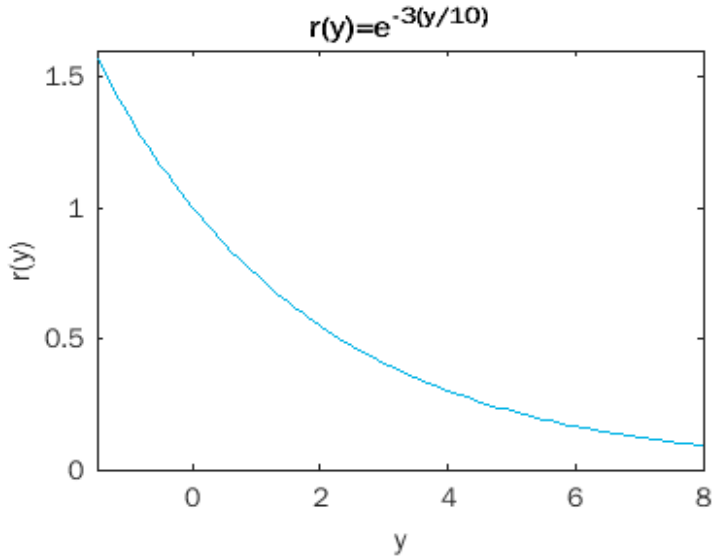
Recompensa del robot para el eje x



Nota. La gráfica se realizó con valores desde -8 hasta 8 con pasos de 0.1. Fuente. Elaboración propia con referencia en (Tearle, n.d.)

Figura 6-3

Recompensa del robot para el eje y



Nota. La gráfica se realizó con valores desde -1.5 hasta 8 con pasos de 0.1. Fuente. Elaboración propia con referencia en (Tearle, n.d.)

Para incitar al robot a que se desplazara hacia la posición ((x, y) = (0, 0)) y luego hacia la posición (y = -1.5) se sumaron las dos funciones que lograban hacerlo en cada eje de forma independiente como se observa en la ecuación (14). En la **Figura 6-4(a)** se aprecia la forma de la función de recompensa, los colores cálidos representan valores de recompensa altos y los fríos representan recompensas negativas, la **Figura 6-4(b)** muestra la forma de la función de recompensa sobre el entorno de entrenamiento.

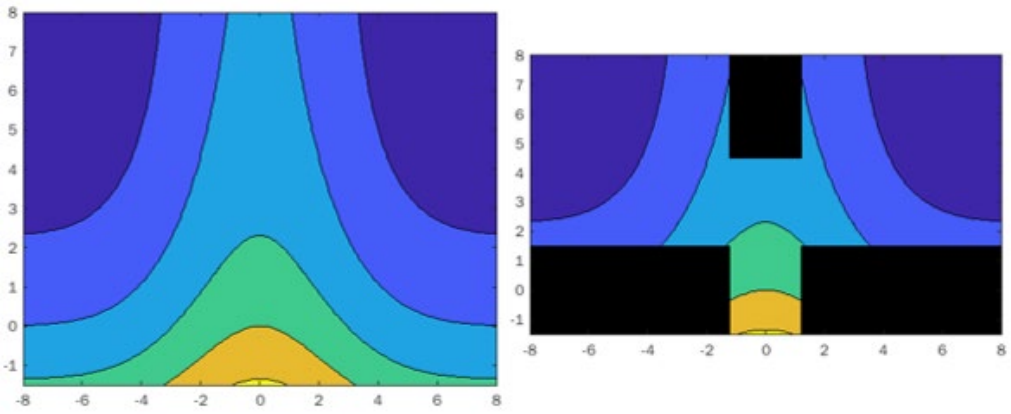
$$r = c_1 e^{-8(x/10)^2} + c_2 e^{-3(y/10)} + c_3 \tag{14}$$

Modificando los coeficientes c_1, c_2 y c_3 para dar forma a la función de recompensa se obtuvo la función presentada en la ecuación (15).

$$r = 0.05 e^{-8(x/10)^2} + 0.06 e^{-3(y/10)} - 0.14 \tag{15}$$

Figura 6-4

Forma de la función de recompensa



Nota. La figura (a) muestra la forma de la función de recompensa. La figura (b) muestra la forma de la fusión de recompensa sobre el entorno de entrenamiento. Fuente. Elaboración propia basada en (Tearle, n.d.)

Teniendo la forma de la función se agregaron recompensas y penalizaciones en base a las acciones que realice o el estado en el que se encuentra, esto para aportar más información al agente. Por ejemplo, cada episodio de entrenamiento se cuenta como terminado si algún robot choca con uno de los obstáculos (se debe penalizar con recompensas más bajas), o si llega a la posición objetivo (se recompensa mayormente). También se puede reducir el esfuerzo de los motores, puesto que las acciones del robot son fuerzas de rotación y traslación cuya energía de los motores proviene de las baterías, se puede desalentar la acciones que generan esfuerzo innecesario que representa una mayor carga para las baterías. Para lograr alcanzar la meta utilizando acciones mínimas se puede agregar un factor que limite las acciones con relación al cumplimiento del objetivo. De manera similar se puede agregar un factor que desaliente comportamientos no deseados, como girar demasiado rápido. La ecuación (16) recoge todos los factores de la función de recompensa que se utilizó (Tearle, n.d.).

$$r = 0.05e^{-8\left(\frac{x}{10}\right)^2} + 0.06e^{-3\left(\frac{y}{10}\right)} - 0.14 - 0.01\omega^2 + 5Finalizado - 2Colisión + 0.01\sum acción^2 \quad (16)$$

7. Actor, Crítico y Desarrollo en Simulink

7.1 Crear el actor

En el presente caso se utilizaron agentes de gradiente de políticas determinísticas profundos (DDPG) que están conformados por un actor determinístico y un crítico de valor Q.

Para definir la red neuronal de cada actor se utilizó una capa de entrada con 6 neuronas correspondientes al número de estados observables, posteriormente se utilizó una capa totalmente conectada con 100 neuronas y una capa ReLu, estas capas se copiaron dos veces más como sigue; capa totalmente conectada con 100 neuronas → capa ReLu → capa totalmente conectada con 100 neuronas → capa ReLu. Finalmente se utilizó una capa totalmente conectada con 2 neuronas que corresponde al número de acciones (Fuerza de traslación y Rotación) y una capa tanh (tangente hiperbólica) para establecer la salida dentro de los límites -1 y 1.

La taza de aprendizaje se estableció en $5 * 10^{-5}$. Para actualizar los valores de la ganancia (w) y el bias (b) se utiliza el método del descenso del gradiente (Ng, n.d.), para el presente caso se estableció el umbral en 10. Estos parámetros se especificaron en la función `rlRepresentationOptions` que posteriormente se utilizó como opciones al crear cada actor.

Finalmente, para crear los actores se utilizó la función `rlDeterministicActorRepresentation`, en sus parámetros de entrada se especificó la matriz donde se definieron las capas de la red neuronal, la variable que almacena las observaciones, la variable donde se almacenan las acciones, el nombre de la capa de entrada de las observaciones, el nombre de la capa de salida y las opciones (establecidas anteriormente), esto se hizo de igual forma para los dos agentes.

7.2 Crear el crítico

Para crear el crítico se realizó un procedimiento similar al utilizado para crear al actor; Inicialmente se definió una red neuronal con una capa de entrada de 6 neuronas, correspondientes al número de estados observables, seguidamente se utilizó una capa

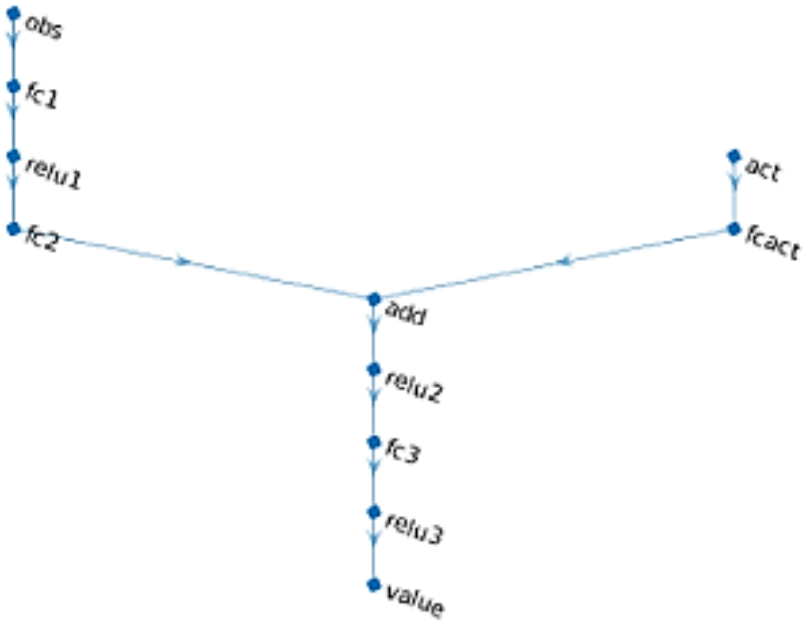
totalmente conectada con 100 neuronas y una capa ReLu, posteriormente se utilizó otra capa totalmente conectada con 100 neuronas, se utilizó una capa de adición con dos entradas y en una de ellas se conectó la red creada hasta el momento, se utilizó otra capa ReLu → capa totalmente conectada con 100 neuronas → capa ReLu. Finalmente se utilizó una capa totalmente conectada con 1 neurona que corresponde al valor Q.

La otra red neuronal independiente se definió con una capa de entrada con 2 neuronas que corresponden a las 2 acciones (Fuerza de traslación y Rotación) y una capa totalmente conectada de 100 neuronas. Esta última capa se conectó a la otra entrada de la red neuronal de adición, quedando finalmente como se observa en la **Figura 7-1**.

Finalmente, para crear el crítico se utilizó la función `rlQValueRepresentation`, en sus parámetros de entrada se especificó la matriz donde se definieron las capas de la red neuronal, la variable que almacena las observaciones, la variable donde se almacenan las acciones, el nombre de la capa de entrada de las observaciones, el nombre de la capa de entrada de las acciones y las opciones (se utilizaron las mismas que en el actor). Este proceso se realizó para los dos agentes de la misma forma.

Figura 7-1

Estructura de la red neuronal del crítico



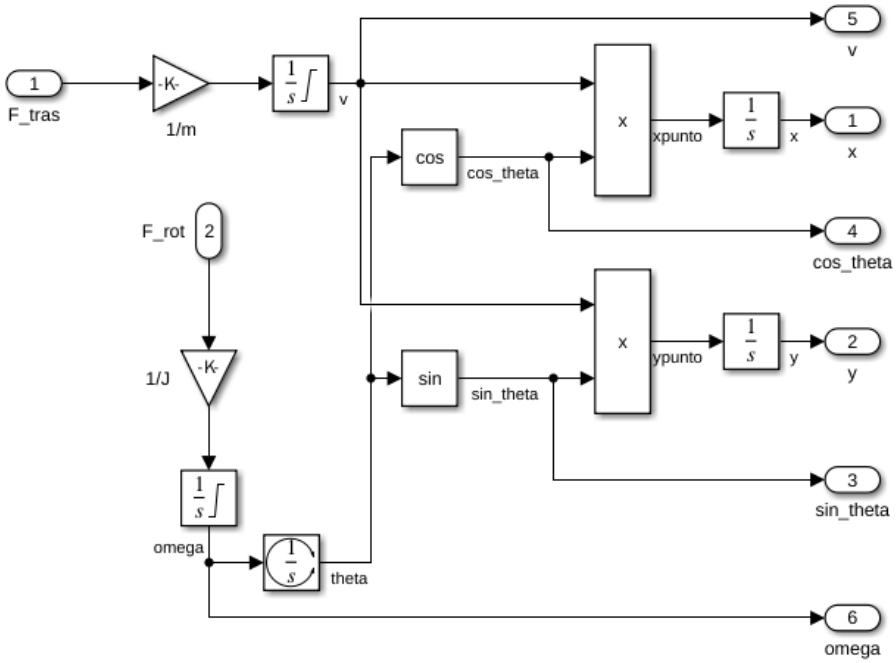
Nota. Estructura de la red neuronal del crítico. Fuente. (Tearle, n.d.)

7.3 Desarrollo en simulink

El modelo de la **Figura 7-2** presenta la dinámica del robot móvil con tracción diferencial, esta se almacenó dentro de un bloque como se observa en la **Figura 7-3**.

Figura 7-2

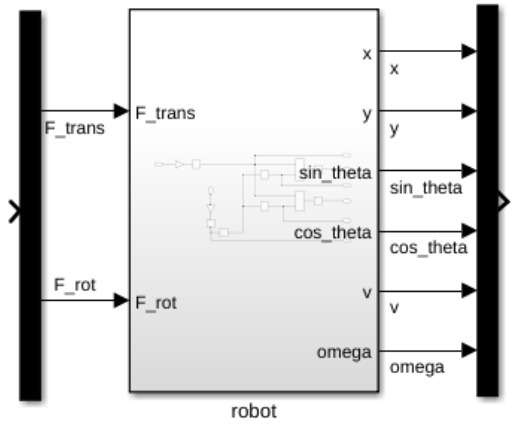
Dinámica del robot diferencial implementada en Simulink



Nota. Dinámica del robot implementada en la herramienta Simulink. Fuente. (Tearle, n.d.)

Figura 7-3

Bloque del robot móvil con tracción diferencial

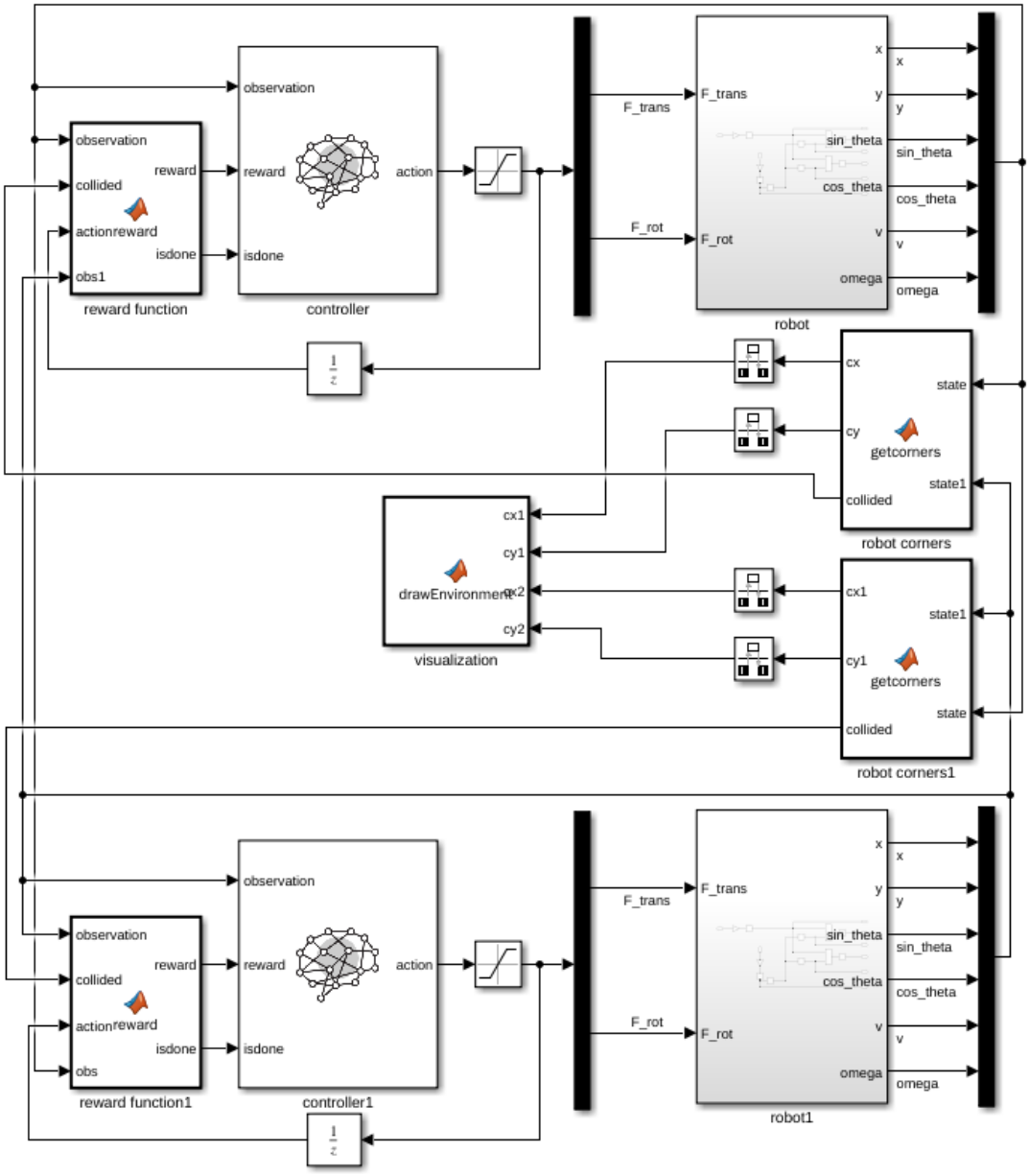


Nota. Dinámica del robot móvil con tracción diferencial almacenada dentro de un bloque para simplificar el modelo del entorno de simulación. Fuente. (Tearle, n.d.)

El modelo de cada robot es el mismo implementado en la prueba para un solo robot (como en el ejemplo 1) y que se puede observar en la **Figura 7-3**. Este se duplicó para conformar el entorno multiagente como se observa en la **Figura 7-4**.

Figura 7-4

Modelo del entorno y los agentes (controller y contoller1) implementado en Simulink



Nota. Modelo del entorno con dos robots (robot y robot1) y dos agentes (controller y controller1) implementado en Simulink. Fuente. Elaboración propia con base en (MathWorks, n.d.-b; Tearle, n.d.)

8. Evaluación del Algoritmo

8.1 Entrenamiento de los agentes

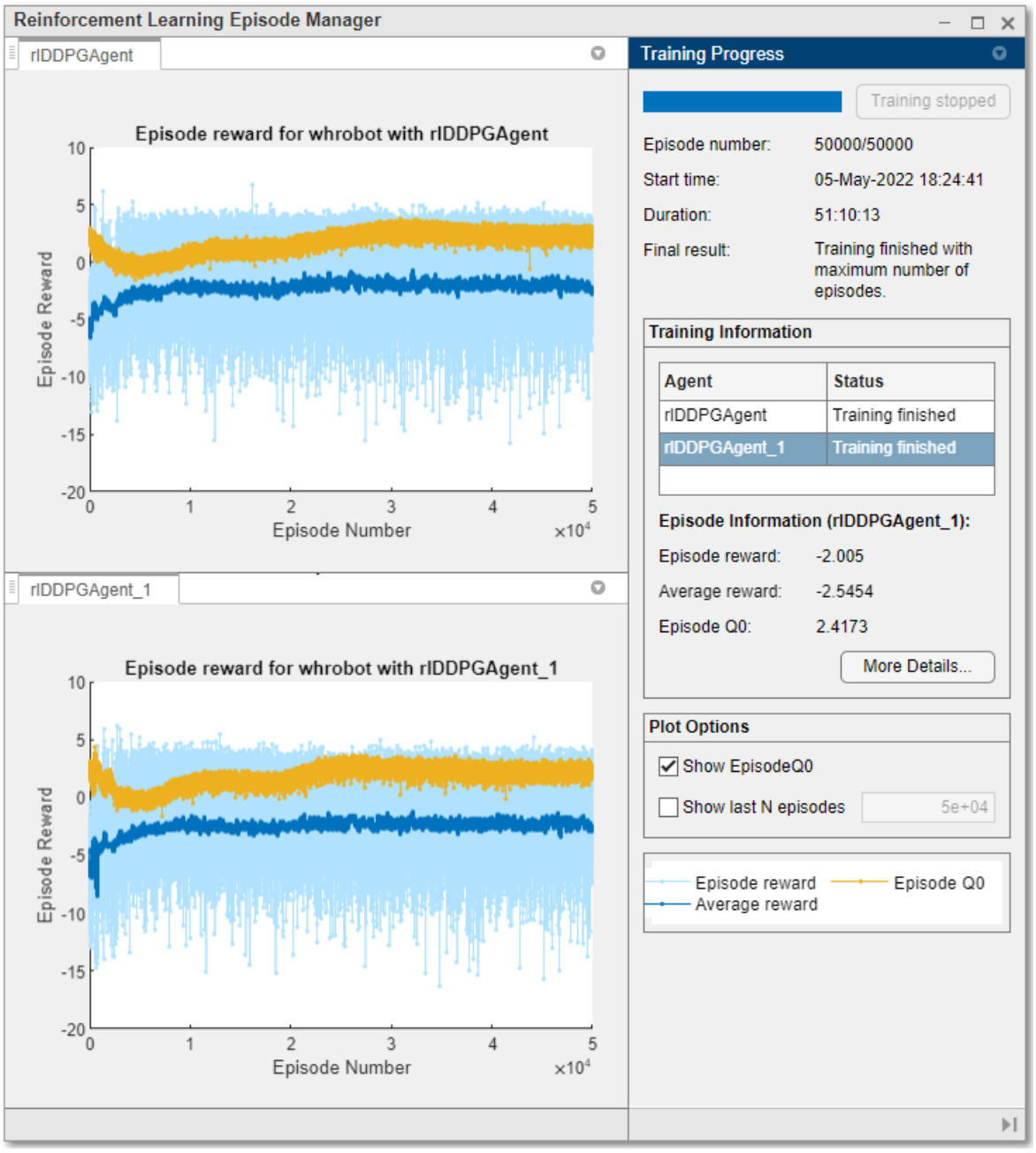
Para realizar el entrenamiento de los dos agentes se utilizó la función *train*. Como parámetros de entrada esta función recibe los dos agentes (en forma de vector), el entorno y las opciones de entrenamiento, estas últimas se implementaron con los siguientes parámetros:

- Pasos máximos por episodio: 120
- Número máximo de episodios: 50000
- Duración media de la ventana de tiempo: 50
- Criterios de detención de entrenamiento: Recompensa promedio
- Valor para detener el entrenamiento: 2
- Tamaño de la ventana para calcular la recompensa promedio: 100 episodios
- Este entrenamiento se inició partiendo del agente preentrenado de la sección **3.1.11**.

La salida de la función *train* es una estructura que contiene la información de la recompensa por cada uno de los episodios, la recompensa promedio, y la predicción de la recompensa futura a largo plazo realizada por el crítico. En la **Figura 8-1** se puede observar el proceso del entrenamiento, se evidencia que se detuvo al cumplir el número máximo de episodios (5000) sin embargo, dicho modelo partió de un agente preentrenado con alrededor de 28000 episodios. Si bien no se consiguió que la recompensa promedio llegase al valor de 2, fue suficiente para que los dos agentes aprendieran a controlar los robots y direccionarlos hacia la posición deseada.

Figura 8-1

Resultado del entrenamiento de sistema multiagente



Nota. Si bien la recompensa promedio no llegó al valor esperado, se cumplió el criterio de episodios máximos y se detuvo el entrenamiento. Fuente. Elaboración propia

8.2 Evaluación de los agentes

Para evaluar los dos agentes se simularon 20 episodios, esto utilizando la función *sim*. Al ser un sistema multiagente, dicha función entrega como salida una matriz de estructuras donde cada fila corresponde al episodio y cada columna a los resultados de cada agente. Las filas de la matriz tienen la información de las observaciones por cada paso, el valor de las acciones, la recompensa y si el episodio se terminó por colisión, se cumplió el objetivo o por que se alcanzó el número máximo de pasos. Cabe resaltar que para cada episodio de prueba la posición inicial de los dos robots se establece de forma aleatoria.

Después de determinar cuáles episodios terminaron para cada agente, comparando con el número de episodios ejecutados, se determinó cuáles fueron exitosos, en cuántos ocurrió alguna colisión y cuantos episodios no se completaron. Los resultados obtenidos se presentan en la **Tabla 8-1**.

Tabla 8-1.

Resultados de la validación de los agentes

	Agente 1 (controller)	Agente 2 (controller1)
Número de episodios completados	20	20
Número de colisiones	0	0
Episodios que no se completaron	0	0

Nota: Los resultados se extrajeron utilizando la función *sim* y la notación de puntos. Fuente. Elaboración propia

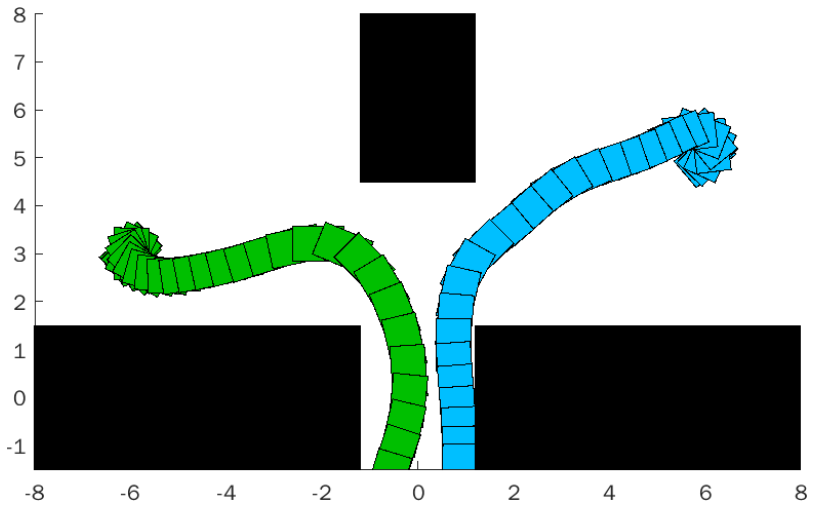
Para mostrar gráficamente los resultados obtenidos se simularon 5 episodios más para obtener la trayectoria de los dos robots y verificar la señal del error para cada agente. En la **Figura 8-12** se muestran las posiciones que se tomaron para cada episodio. Los resultados se presentan a continuación.

En la **Figura 8-2** se muestra gráficamente la trayectoria de los agentes a través del entorno, evidenciando que llegan a la posición objetivo en el episodio 1 de prueba.

En la **Figura 8-3** se observa la señal del error para la posición respecto x y para la posición respecto a y de los dos robots, esto para el episodio número 1 de validación.

Figura 8-2

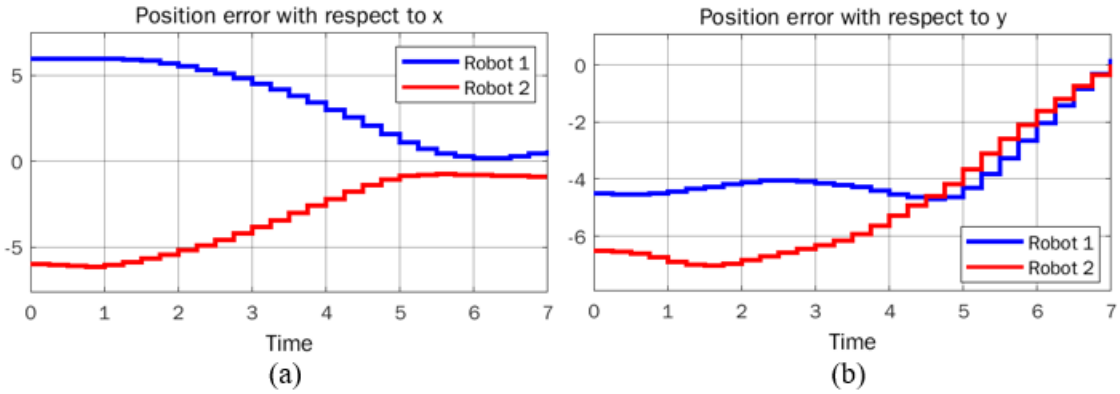
Trayectoria de los robots en el entorno (episodio 1)



Nota. Posición de los dos robots para el episodio 1 de validación con los agentes entrenados. Fuente. Elaboración propia

Figura 8-3

Señales del error (episodio 1)



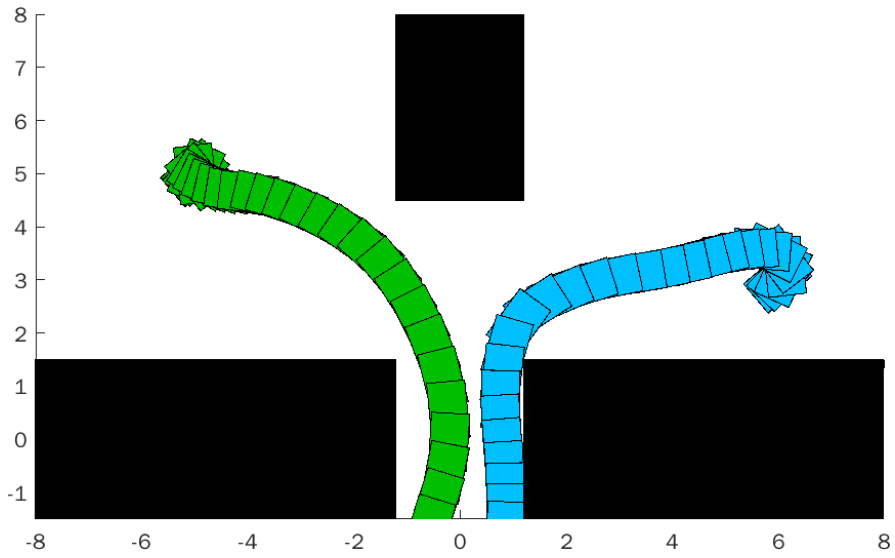
Nota. La figura (a) muestra el error de posición respecto a x de los dos robots. La figura (b) muestra el error de posición respecto a y. Esto para el episodio 1. Fuente. Elaboración propia

En la **Figura 8-4** se muestra gráficamente la trayectoria de los agentes a través del entorno, evidenciando que llegan a la posición objetivo en el episodio 2 de prueba.

En la **Figura 8-5** se observa la señal del error para la posición respecto x y para la posición respecto a y de los dos robots, esto para el episodio número 2 de validación.

Figura 8-4

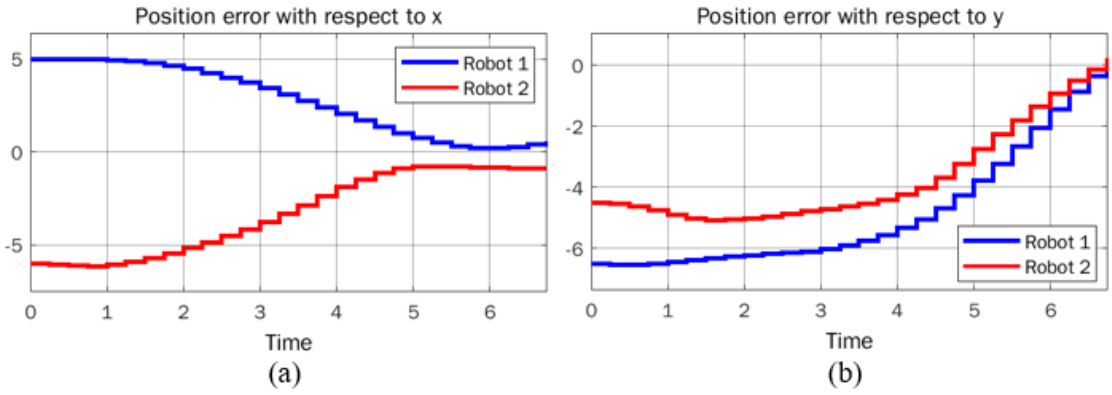
Trayectoria de los robots en el entorno (episodio 2)



Nota. Posición de los dos robots para el episodio 2 de validación con los agentes entrenados. Fuente. Elaboración propia

Figura 8-5

Señales del error (episodio 2)



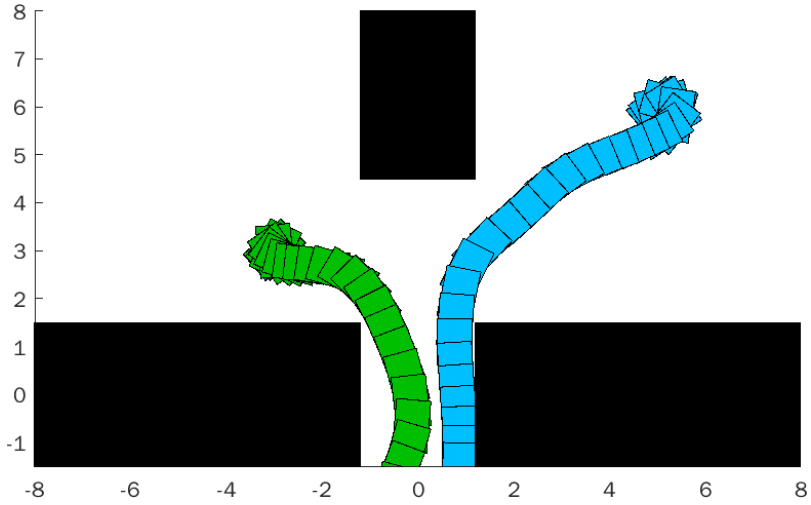
Nota. La figura (a) muestra el error de posición respecto a x de los dos robots. La figura (b) muestra el error de posición respecto a y. Esto para el episodio 2. Fuente. Elaboración propia

En la **Figura *8-6*** se muestra gráficamente la trayectoria de los agentes a través del entorno, evidenciando que llegan a la posición objetivo en el episodio 3 de prueba.

En la **Figura 8-7** se observa la señal del error para la posición respecto x y para la posición respecto a y de los dos robots, esto para el episodio número 3 de validación.

Figura 8-6

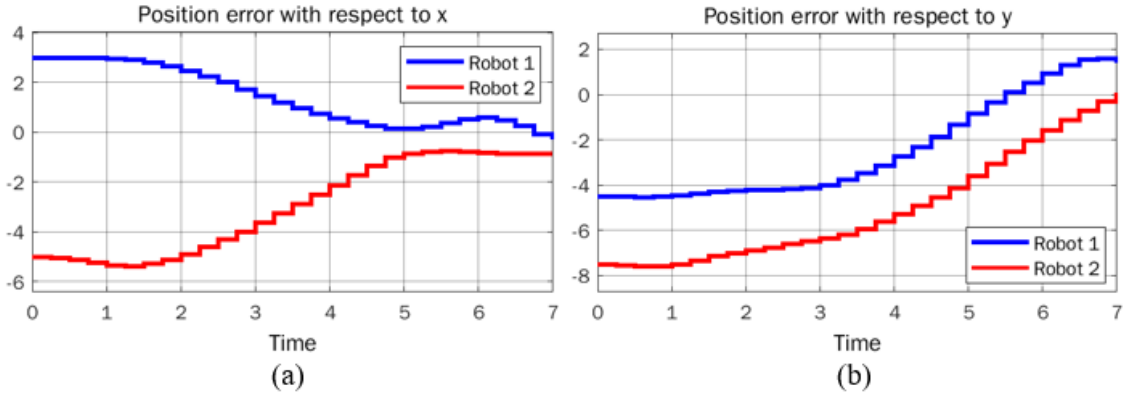
Trayectoria de los robots en el entorno (episodio 3)



Nota. Posición de los dos robots para el episodio 3 de validación con los agentes entrenados. Fuente. Elaboración propia

Figura 8-7

Señales del error (episodio 3)



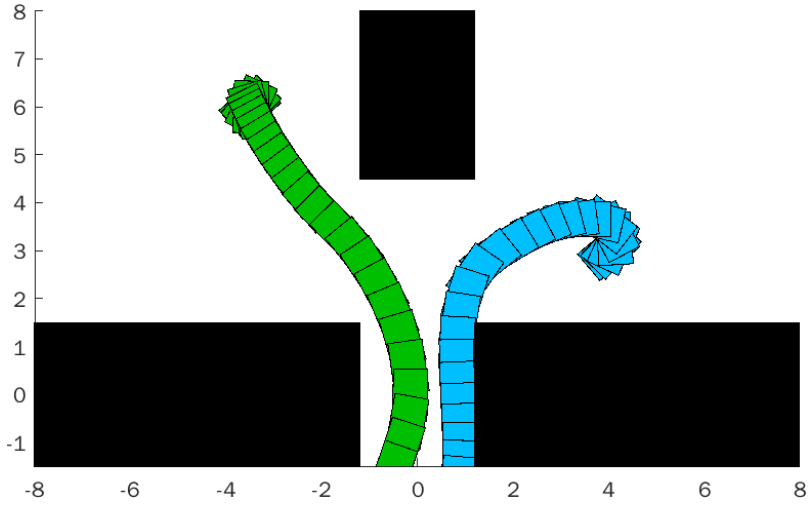
Nota. La figura (a) muestra el error de posición respecto a x de los dos robots. La figura (b) muestra el error de posición respecto a y. Esto para el episodio 3. Fuente. Elaboración propia

En la **Figura *8-8*** se muestra gráficamente la trayectoria de los agentes a través del entorno, evidenciando que llegan a la posición objetivo en el episodio 4 de prueba.

En la **Figura 8-9** se observa la señal del error para la posición respecto x y para la posición respecto a y de los dos robots, esto para el episodio número 4 de validación.

Figura 8-8

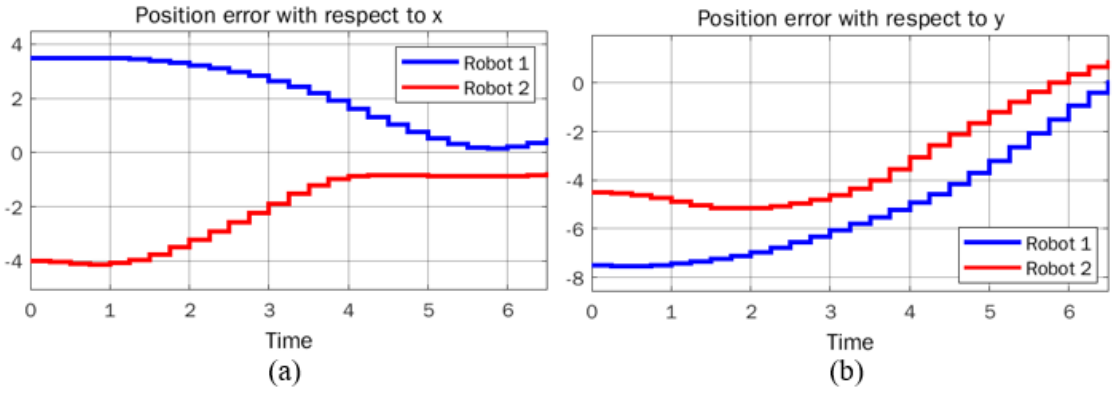
Trayectoria de los robots en el entorno (episodio 4)



Nota. Posición de los dos robots para el episodio 4 de validación con los agentes entrenados. Fuente. Elaboración propia

Figura 8-9

Señales del error (episodio 4)



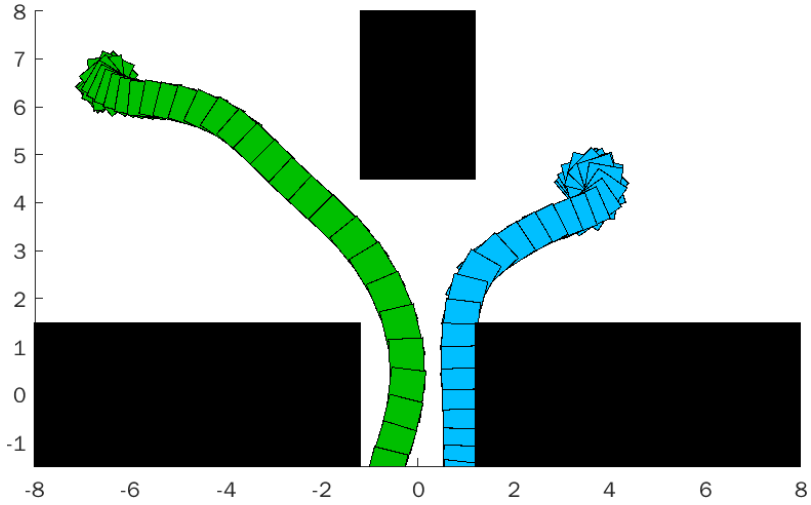
Nota. La figura (a) muestra el error de posición respecto a x de los dos robots. La figura (b) muestra el error de posición respecto a y. Esto para el episodio 4. Fuente. Elaboración propia

En la **Figura *8-10*** se muestra gráficamente la trayectoria de los agentes a través del entorno, evidenciando que llegan a la posición objetivo en el episodio 5 de prueba.

En la **Figura 8-11** se observa la señal del error para la posición respecto x y para la posición respecto a y de los dos robots, esto para el episodio número 5 de validación.

Figura 8-10

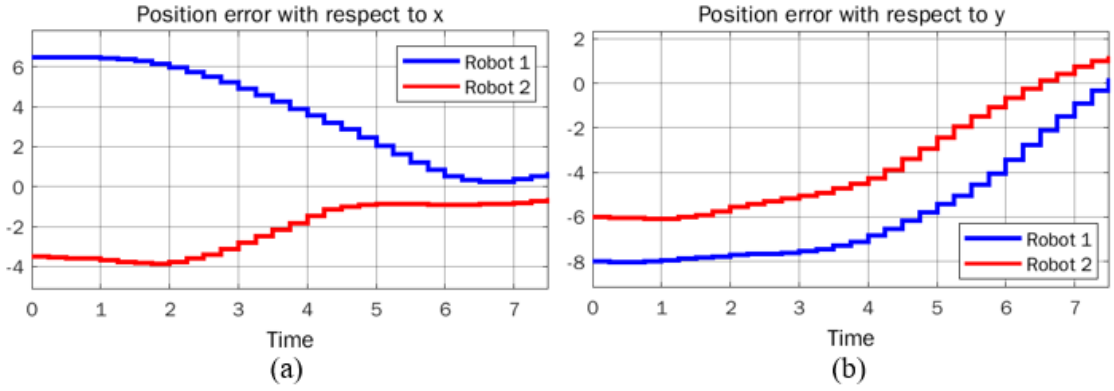
Trayectoria de los robots en el entorno (episodio 5)



Nota. Posición de los dos robots para el episodio 5 de validación con los agentes entrenados. Fuente. Elaboración propia

Figura 8-11

Señales del error (episodio 5)



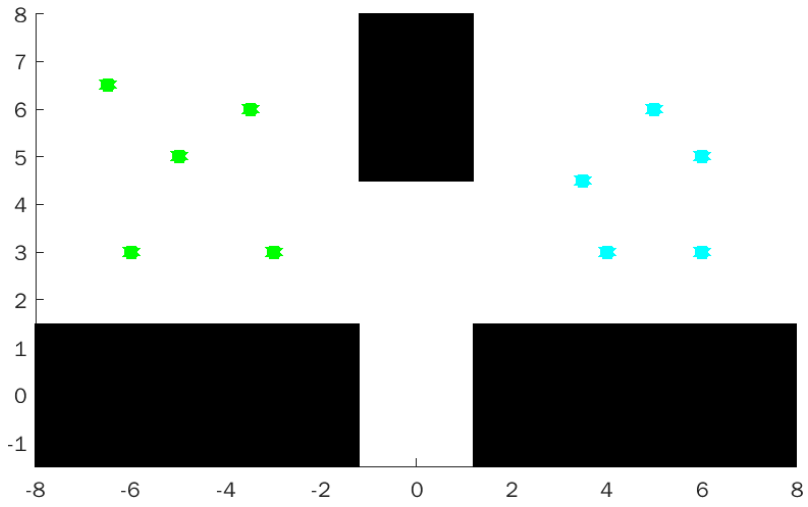
Nota. La figura (a) muestra el error de posición respecto a x de los dos robots. La figura (b) muestra el error de posición respecto a y. Esto para el episodio 5. Fuente. Elaboración propia

De los 5 últimos episodios de prueba que se realizaron se evidencia que los dos robots se dirigen hacia la posición $(0, -1.5)$, aunque debido al espacio y la restricción de colisión entre ellos mismos, en el eje x el robot1 se dirige levemente hacia la derecha y el otro hacia la izquierda. También se evidencia que el error respecto a y no es cero debido a que los robots llegan a destiempo a la posición objetivo y, el primero que llega, se desplaza levemente más que el otro sobre el eje y negativo.

En la **Figura 8-12** se observa un mapa de posiciones sobre el entorno, estas fueron las ubicaciones iniciales de los robots que se tomaron para la evaluación de las 5 pruebas.

Figura 8-12

Posiciones iniciales considerada para la segunda prueba de evaluación de los agentes

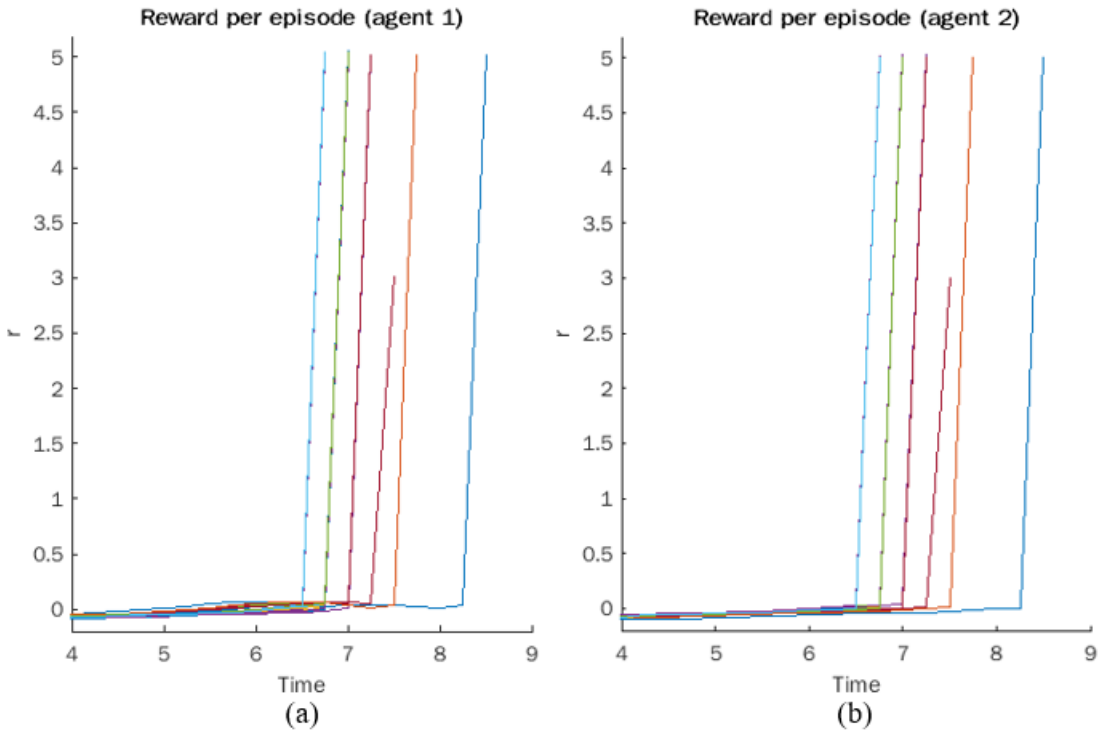


Nota. Las posiciones en color verde fueron las iniciales para cada episodio de entrenamiento del robot 1 controlado por el agente "controller", las posiciones en color azul fueron las iniciales para cada episodio de entrenamiento del robot 2 controlado por el agente "controller1".

La **Figura 8-13** muestra la recompensa que obtuvo cada agente en los 20 episodios de prueba, se puede apreciar que siempre fue positiva para todos los episodios.

Figura 8-13

Recompensa por episodio para el sistema multiagente



Nota. La recompensa de algunos episodios es la misma obtenida en otros, por ello no se evidencia de manera clara la totalidad de los 20 episodios. Fuente. Elaboración propia

8.3 Prueba en nuevo entorno

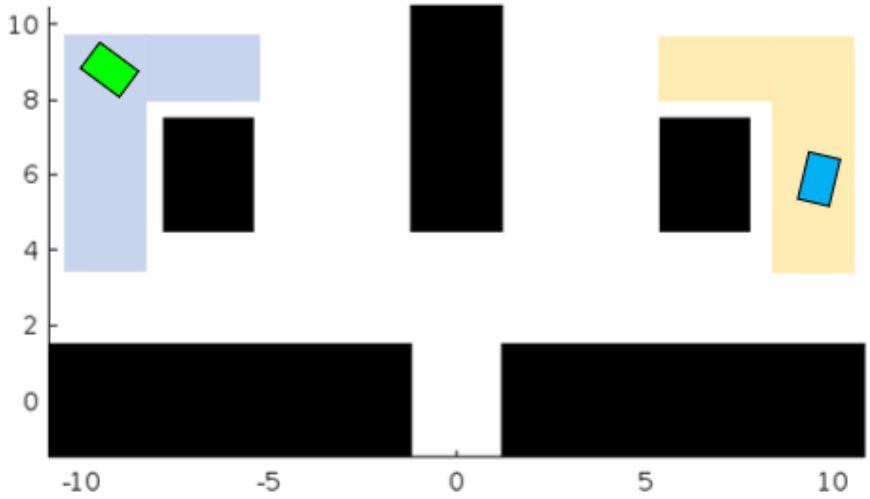
8.3.1 Entrenamiento de los agentes

Para el entrenamiento de los agentes sobre el entorno B (ver **Figura 8-14**) se utilizaron los mismos parámetros de entrenamiento presentados en la sección **8.1**. También se utilizó dicho modelo como agente preentrenado para el entrenamiento en el entorno B. En la **Figura 8-15** se presenta el proceso de entrenamiento, allí se observa que se detuvo de manera forzada oprimiendo el botón de Stop Training al llevar alrededor de 13000 episodios, pues al realizar pruebas previamente donde se entrenaron los agentes en un tiempo

aproximado de 80 horas, se observó que a partir de los 10000 episodios la gráfica tenía la misma tendencia y no se cumplía el criterio de recompensa promedio para detenerse automáticamente. Aunque se detuvo el entrenamiento de manera forzada, en dicho modelo los agentes aprendieron a controlar los robots y direccionarlos hacia la posición deseada.

Figura 8-14

Nuevo entorno de entrenamiento (entorno B)



Nota. Nuevo entorno que incluye obstáculos para los robots. Fuente. Elaboración propia

8.3.2 Evaluación de los agentes

Para evaluar los agentes se simularon 20 episodios como también se realizó en la sección **8.2**. Los resultados obtenidos se presentan en la **Tabla 8-2**.

Tabla 8-2.

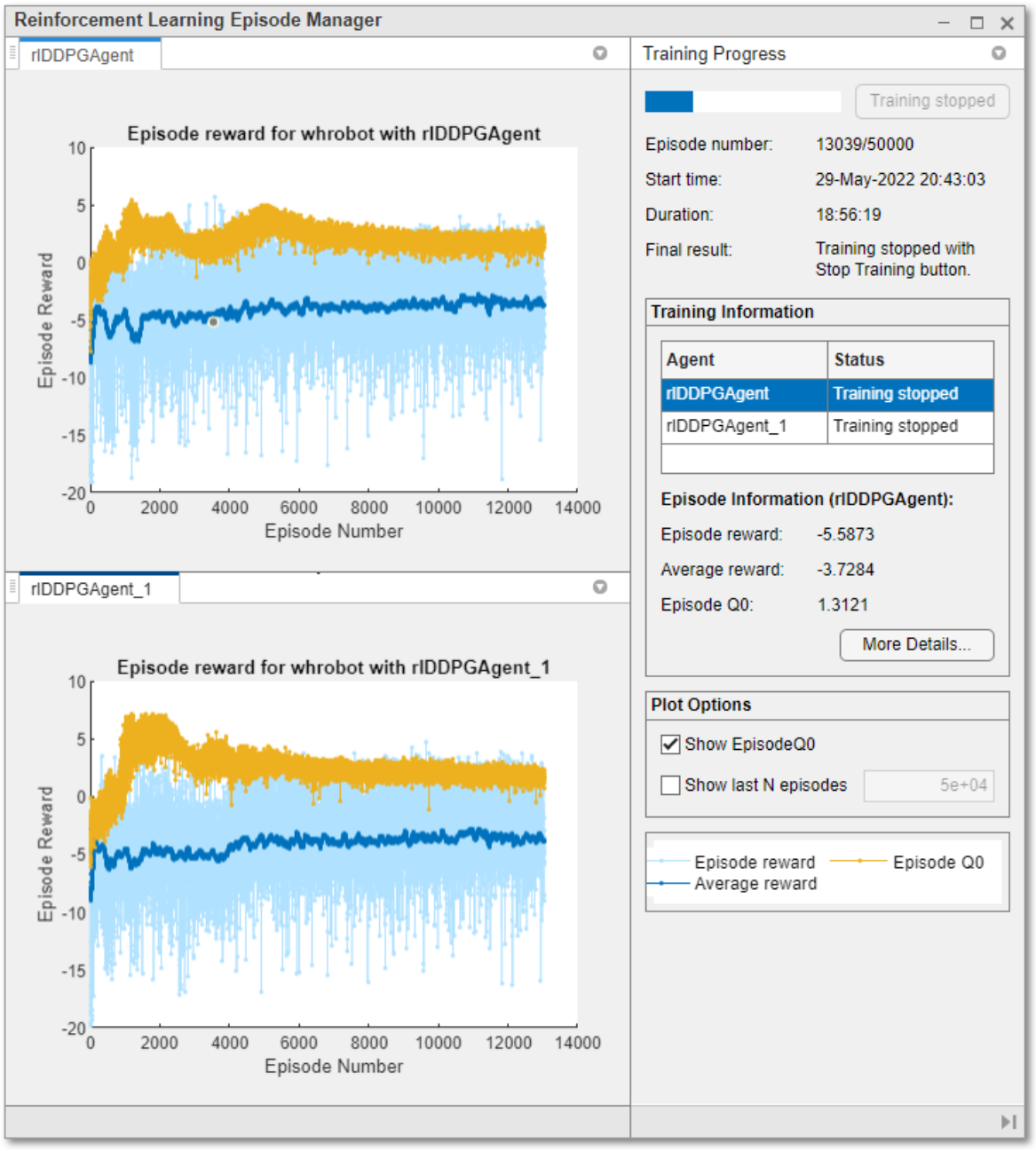
Resultado de la validación de los agentes en nuevo entorno (entorno B)

	Agente 1 (controller)	Agente 2 (controller1)
Número de episodios completados	20	20
Número de colisiones	0	0
Episodios que no se completaron	0	0

Nota: Los resultados se extrajeron utilizando la función sim y la notación de puntos. Fuente. Elaboración propia

Figura 8-15

Resultado del entrenamiento de sistema multiagente en nuevo entorno (entorno B)

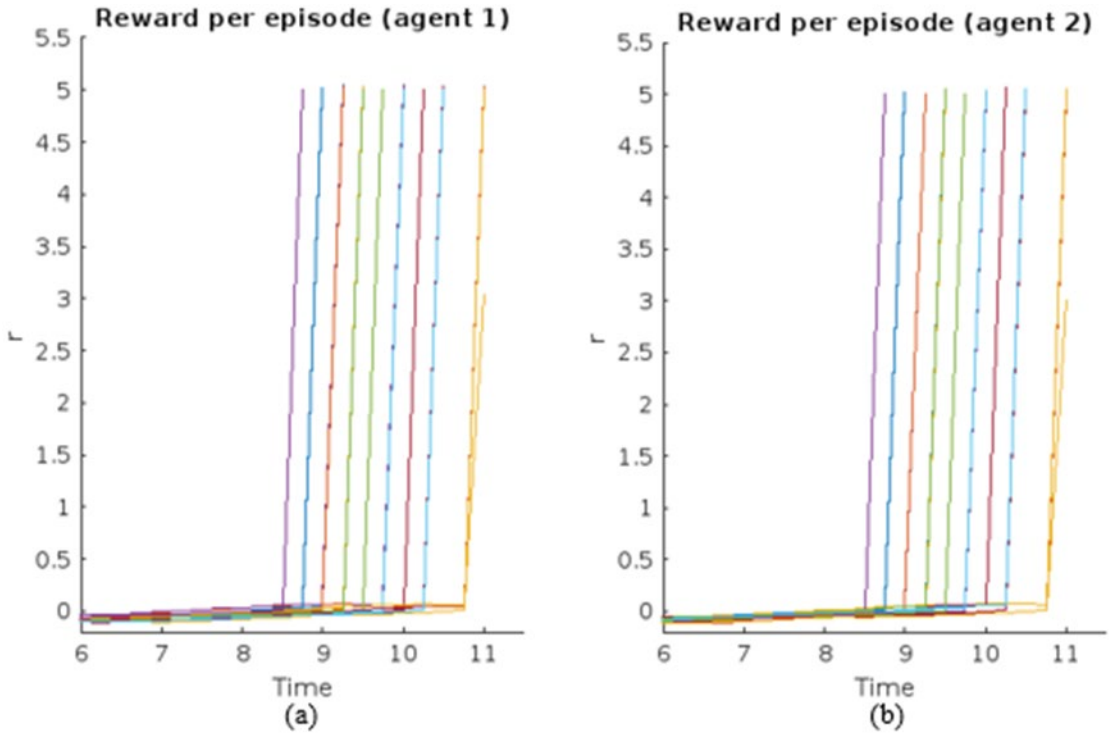


Nota. Aunque se detuvo el entrenamiento de manera forzada, los agentes lograron aprender y cumplir el objetivo. Fuente. Elaboración propia

La **Figura 8-16** muestra la recompensa que obtuvo cada agente en los 20 episodios de prueba, allí se puede apreciar que siempre fue positiva al finalizar el episodio. También se aprecia que el tiempo en el cual se obtiene la recompensa es mayo en comparación con la recompensa obtenida en la sección **8.2** y que se observa en la **Figura 8-13**, esto debido a que los robots deben recorrer una mayor distancia.

Figura 8-16

Recompensa por episodio para el sistema multiagente en nuevo entorno (entorno B)



Nota. La recompensa de algunos episodios es la misma obtenida en instantes de tiempo muy similares al obtenido en otros, por ello no se evidencia de manera clara la totalidad de los 20 episodios. Fuente. Elaboración propia

9. Recomendaciones

Como se menciona en la introducción del presente documento, el aprendizaje y uso de la inteligencia artificial es una herramienta que permite al ingeniero tener una alta competitividad en el mercado laboral debido al amplio uso de la IA en un extenso número de áreas, por ello, primeramente, se sugiere habilitar espacios de investigación donde se aprendan las bases matemáticas y de programación requeridas para fortalecer su proceso de formación e investigación en el uso de las múltiples herramientas de las que se disponen para practicar inteligencia artificial.

Ahora bien, en el marco del presente proyecto la recomendación primordial está enfocada en el equipo donde se entrenarán los agentes, ya que, como se menciona en la sección **10** y que se apoya en el **Anexo B**, resulta demasiado costoso computacionalmente utilizar aprendizaje por refuerzo con un sistema multiagentes que interactúan de manera individual para conseguir un objetivo general, por ejemplo, en el presente caso, puesto que los robots se pueden identificar como barreras dinámicas en el entorno de entrenamiento, se deben utilizar demasiadas restricciones para que el entrenamiento converja a lo esperado, esto puede traer grandes dificultades al implementarlo de manera práctica ya que en el mundo real no se pueden ignorar dichas restricciones. Por ello, para poder realizar aplicaciones reales se recomienda un pc con memoria RAM superior a 12GB y con una tarjeta gráfica dedicada, pues Matlab permite simular en paralelo haciendo uso de la GPU.

10. Discusión y Conclusiones

En este trabajo de grado se logró desarrollar y simular un sistema multiagente basado en el algoritmo de aprendizaje por refuerzo Q-learning que permitió la navegación autónoma de dos robots móviles a través de un entorno donde aparecían de forma aleatoria y eran conducidos hacia una posición objetivo, logrando obtener una efectividad del 100%.

- Se definieron correctamente los estados y acciones para cada uno de los agentes, permitiendo construir una red neuronal para cada crítico de valor Q, que permitió el aprendizaje de trayectorias en los dos robots móviles.
- Se estableció de manera adecuada la función de recompensa que durante el entrenamiento permitió a los agentes escoger entre realizar una acción ya conocida (explotación) o realizar una acción por primera vez (exploración).
- Se evaluó el rendimiento de los agentes simulando 20 episodios en los cuales la ubicación inicial de cada robot era aleatoria y a partir de allí el agente se encargaba de orientar al robot hacia los Estados que largo plazo le generaban una mayor recompensa acumulada, esto es, logrando el llegar a la posiciónl objetivo. de esos 20 episodios en el 100% de los casos los robots llegaron a la posición objetivo sin chocar con ningún obstáculo ni con ellos mismos.

Si bien se logró cumplir con el objetivo general del proyecto, genera cierta incertidumbre sobre la eficiencia y eficacia del uso de este tipo de técnicas para solucionar problemas en la práctica, pues, por ejemplo, se generan dudas sobre el costo computacional necesario para trabajos robustos comparado con el tiempo que podría invertir un ingeniero experto en el tema, o si aún no lo fuera, sumar el tiempo que le tomaría aprender a desarrollar dicha tarea.

Si se quisiera entrenar en un ambiente más grande resultaría altamente costoso computacionalmente debido a que debe hacer un alto número de pruebas para aprender qué acción realizar en cada uno de los posibles estados de todo el entorno y, si es muy grande, va a tener una cantidad inmensa de estados y tendría que haber estado al menos una vez en cada uno de ellos y probado las posibles acciones para saber qué acción le genera una recompensa mayor en el largo plazo, puesto que el agente únicamente puedes seleccionar

las acciones que generan mejor desempeño a largo plazo si durante el entrenamiento se encontró en los mismos estados específicamente.

Se observó que la inteligencia artificial puede aprovecharse de problemas o vacíos que no se contemplen durante la programación, pues en el presente caso, hubo una prueba de entrenamiento en la que la inteligencia artificial explotó una falla respecto a los límites del área de entrenamiento y la condición establecida para un episodio ejecutado satisfactoriamente. En primera instancia no se había establecido un límite en el entorno de tal forma que la inteligencia artificial lo entendiera, por ejemplo, como una colisión. En segundo lugar el número de pasos por cada simulación era demasiado alto logrando así maximizar mayormente su recompensa si uno los robots se dirigía hacia la parte superior del entorno recibiendo recompensas negativas pequeñas y el otro se desplazaba hacia la posición 0 en el eje x y rápidamente se manera seguida haciabel eje y negativo hasta completar el número máximo de pasos por episodio, consiguiendo así obtener una recompensa promedio mayor en un número de episodios significativamente más pequeño que en otras pruebas, pero que al momento de evaluar al agente no se obtuvo ningun episodio satisfactorio.

11. Anexos

11.1 Anexo A: Redes Neuronales

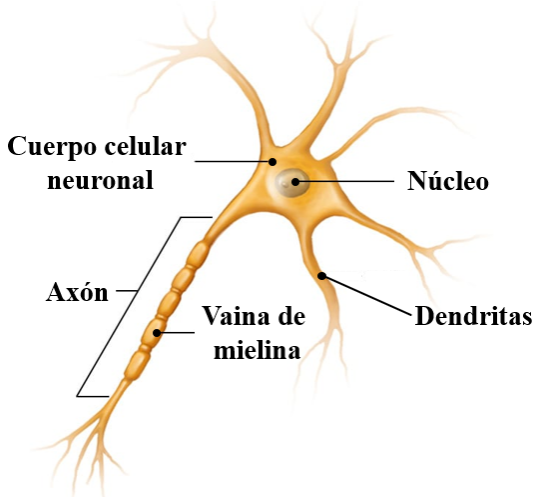
11.1.1 *Inspiración Biológica*

Anatómicamente una neurona recibe constantes pulsos eléctricos a través de un conjunto de delicadas estructuras llamadas dendritas, esta información es tratada por la neurona y a partir de esta toma las decisiones. Las neuronas emiten pulsos de actividad eléctrica hacia otras neuronas a través de una fibra llamada axón, que se escinde en millones de ramificaciones.

El cuerpo o soma de la neurona es el encargado de realizar el tratamiento a los pulsos eléctricos que se reciben, es el "órgano de cómputo". En la **Figura 11-1** se puede apreciar la estructura de una neurona biológica, esta se debe tener en cuenta para entender el pensamiento que estuvo detrás de la concepción de una neurona artificial.

Figura 11-1

Estructura de una neurona biológica



Nota. La neurona biológica está conformada por el soma o cuerpo celular neuronal, en el cual se encuentra el núcleo de la neurona. El axón, la ramificación principal a través de la cual se envía información (en forma de pulsos eléctricos), las dendritas, son ramificaciones

más pequeñas que el axón y a través de ella se recibe la información. La vaina de mielina es un compuesto graso que recubre los nervios del cerebro. Fuente. (Clinic, 2021)

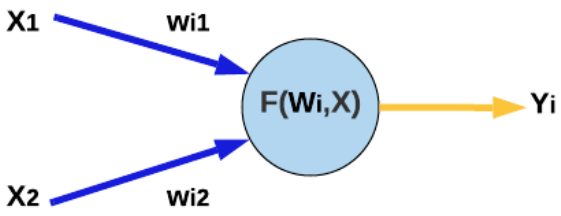
Aunque las neuronas generalmente son cinco o seis órdenes de magnitud más lentas que las puertas lógicas de silicio, cuyos eventos ocurren cerca de los 10^{-3} y 10^{-9} segundos respectivamente, el cerebro compensa esta diferencia gracias a la cantidad de neuronas interconectadas entre ellas. A pesar de que una neurona es capaz de realizar numerosas tareas a una velocidad relativamente lenta, lo más interesante surge de la interconexión entre ellas (Artola Moreno, 2019).

11.1.2 *Neurona artificial*

La neurona artificial se puede entender como una operación algebraica con la cual se pretende imitar algunas características de la neurona biológica. En general, tiene entradas a través de las cuales recibe la información (de manera similar a las dendritas), posteriormente, en el núcleo de la neurona, se procesa la información para enviarla a la salida (a través del axón, en una neurona biológica) (Palmer & Montaño, n.d.). El modelo simplificado de una neurona artificial se puede representar como se observa en la **Figura 11-2**.

Figura 11-2

Ilustración de una neurona artificial simple



Nota. Representación de una neurona artificial. Se observan las dos entradas (X_1, X_2) o dendritas (haciendo analogía con la neurona biológica) junto con los pesos correspondientes a cada entrada (Wi_1, Wi_2), en el círculo azul se encuentra la función de activación ($F(W, X)$)

aplicada a cada entrada multiplicada por su peso (proceso del que se encarga el soma), finalmente se observa la salida Y_i (axón). Esta neurona simple se conoce como perceptrón. Fuente. Elaboración propia.

En el modelo se puede apreciar:

La(s) entrada(s) de la red neuronal (X_n): Cada entrada es un valor numérico (donde $X \in \mathbb{R}$) que representa una característica de la información que va a ser procesada por las neuronas de entrada y, posteriormente, pasara a otras neuronas que conforman la capa oculta en las que se aplicarán procesos matemáticos para finalmente, enviar la información a la capa de salida.

Un conjunto de sinapsis o conectores, cada uno de los cuales tiene un peso sináptico (W_{in}) que define la "fuerza" o "peso" que ejerce dicha conexión sobre la salida, para una entrada específica.

Donde:

i representa el número de la capa

n representa la neurona de la capa i

La función de red $(\sum)$ junto con la función de activación (F), encargada de realizar la suma de las señales de entrada ponderado por las respectivas sinapsis de la neurona. La función de activación se utiliza para aplicar la no linealidad a las operaciones. Las más comunes se presentan a continuación (Calvo, 2018):

- Sigmoide: Con rango de salida entre 0 y 1.

$$y(x) = \frac{1}{(1 + e^{-x})} \tag{17}$$

- Tangente hiperbólica: Con rango de salida entre -1 y 1.

$$y(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \tag{18}$$

- ReLu: Con rango entre 0 e ∞.

$$y(x) = \max(0, x) = \begin{cases} 0 \; para \; x < 0 \\ x \; para \; x \geq 0 \end{cases} \tag{19}$$

- Softmax: utilizada generalmente en la salida para acotar la salida en el rango de 0 a 1, donde se puede interpretar como una probabilidad.

$$y(x)_i = \frac{e^{x_i}}{\sum_{j=1}^{N} e^{x_j}} \quad (20)$$

Donde i representa una neurona (de N neuronas) de la capa anterior a la softmax y N es el total de las neuronas de la capa anterior a la softmax.

La salida ($Y(x)_i$) en función de la activación (generalmente función identidad).

Cada neurona representa una operación algebraica donde multiplica el valor de la entrada por una constante de la rama (llamada peso) y agrega otro valor constante (sesgo o bias). El valor resultante se puede pasar a una función de activación y luego a la capa siguiente para que sea la entrada de otra neurona y seguir un proceso muy similar. A continuación, se presenta matemáticamente el valor de la salida de la neurona artificial simple (ver **Figura 11-2**):

Sin la función de activación:

$$Y(x)_1 = X_1 * W1_1 + X_2 * W1_2 + b \quad (21)$$

Con la función de activación sigmoide (ecuación (1)):

$$\sigma(x) = \frac{1}{1+e^{-(X_1 * W1_1 + X_2 * W1_2 + b)}} \quad (22)$$

11.1.3 *Clasificación de redes neuronales*

11.1.3.1 Clasificación según su topología

Las redes neuronales se pueden clasificar según su topología y según el método de aprendizaje (Calvo, 2017a). Según su topología:

Red neuronal monocapa: Conocida también como perceptrón simple. Se compone por una sola neurona con múltiples entradas que se operan con el peso de cada una y se suman, para finalmente enviar el valor a la función de activación y posteriormente a la salida.

Red neuronal multicapa: También conocida como perceptrón multicapa, es una composición de diferentes redes monocapa. Cada capa de la red multicapa está conformada por múltiples neuronas, la primera capa (capa de entrada) generalmente tiene una neurona por cada dato de entrada, posteriormente la información se pasa a las neuronas que

conforman la ‘capa oculta’ en la que puede haber múltiples capas con múltiples neuronas, finalmente hay una capa de salida que puede estar conformada por una sola neurona o por múltiples neuronas. Según la conexión entre las capas puede ser una red parcialmente conectada o completamente conectada.

Red neuronal convolucional (CNN): Las CNN son un tipo de red neuronal multicapa en la que se combinan capas de convolución con capas de reducción de manera alternada. La ventaja de este tipo de redes es que se entrena por partes para realizar una tarea (detectar una característica) lo que disminuye en gran medida el número de capas ocultas (Calvo, 2017b). Este tipo de rede suele utilizarse más comúnmente con imágenes para aplicaciones de visión computacional.

Red neuronal recurrente (RNN): Las RNN permiten una conexión arbitraria entre las neuronas, por lo cual no tienen una estructura definida en capas, incluso se pueden crear ciclos conectando la salida de una neurona a la entrada de la misma, permitiendo que la red tenga memoria.

11.1.3.2 Clasificación según el método de aprendizaje

Dependiendo del problema se pueden distinguir los siguientes esquemas de aprendizaje (Calvo, 2017a; Isasi & Galván, 2004):

Aprendizaje supervisado: En este modelo de aprendizaje la base de datos que se utiliza para el entrenamiento debe tener dos tipos de atributos: los datos en sí mismos y alguna información relativa a la resolución del problema (una etiqueta), esta se utiliza para comparar la salida de la red neuronal con la salida que debería haber generado (información que se dispone en el conjunto de entrenamiento), y a partir de esta diferencia se modificarán los pesos de la red.

Aprendizaje no supervisado: Para este modelo los datos del conjunto de entrenamiento tienen únicamente la información de los ejemplos, y no se tiene nada que permita orientar el proceso de aprendizaje. En el entrenamiento, la red modifica el peso de cada conexión a partir de la información interna.

Aprendizaje por refuerzo: El aprendizaje por refuerzo se puede interpretar como una variante del aprendizaje supervisado, en esta no se dispone de una etiqueta que permita

evaluar el error que cometa la red neuronal por cada ejemplo de entrenamiento, en su lugar se determina si la salida que generó la red para dicho modelo es o no adecuada.

Como se mencionó anteriormente las redes neuronales con topología de perceptrón multicapa se construyen utilizando múltiples neuronas en una capa y posteriormente utilizando diferentes capas para procesar los datos. Algunas capas comúnmente utilizadas en aprendizaje por refuerzo se presentan en la **Tabla 11-1**.

Tabla 11-1.

Capas de aprendizaje profundo que se utilizan en RL

Capa	Descripción
`imageInputLayer`	Capa que recibe vectores en 2-D y datos normalizados.
`tanhLayer`	Capa de activación con la función tangente hiperbólica.
`reluLayer`	Capa de activación con la función ReLu.
`fullyConnectedLayer`	Capa completamente conectada, todas las salidas de la capa anterior se conectan a cada neurona de la capa completamente conectada.
`Convolution2Layer`	Capa en la que se aplican filtros de convolución a los datos de entrada.
`additionLayer`	Capa donde se unen salidas de múltiples capas juntas.
`concatenationLayer`	Esta capa concatena las entradas en una dimensión establecida.

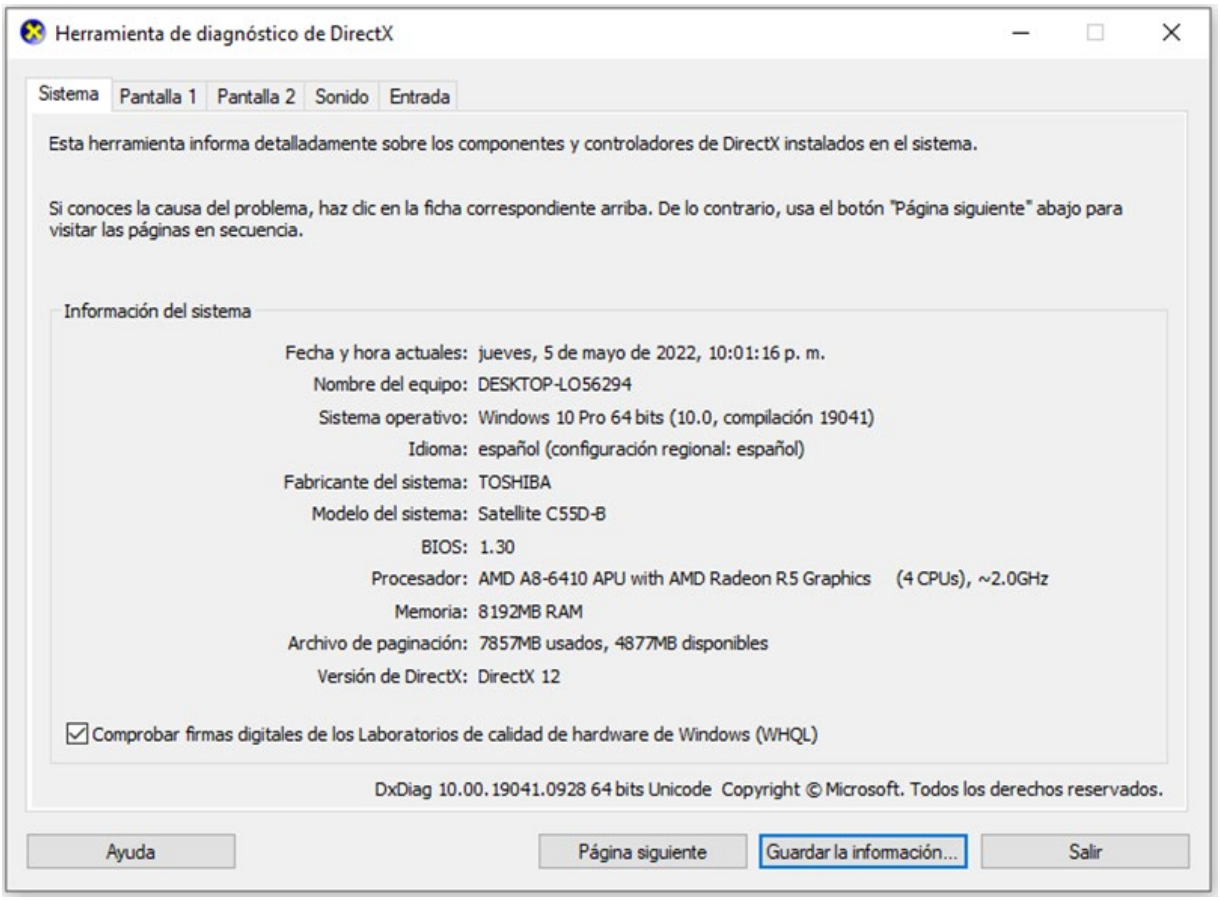
Nota: Estas son algunas de las capas de aprendizaje profundo que se encuentran en la caja de herramientas de aprendizaje profundo (Depp Learning en inglés) en Matlab y que son compatibles con la caja de herramientas de aprendizaje por refuerzo (Reinforcement Learning en inglés). Fuente. (MathWorks, 2019).

11.2 Anexo B: Dificultades presentadas durante el desarrollo

Para las primeras pruebas de entrenamiento se utilizó un computador con las especificaciones mostradas en la **Figura *11-3*** y **Figura *11-4***. De estas pruebas se concluyó que no se contaba con las especificaciones computacionales necesarias para cumplir con los objetivos del proyecto, pues en una prueba que se realizó para entrenar el modelo con un solo robot, llegó a 25000 episodios en un tiempo superior a 150 horas y de manera similar a como se observa en la **Figura *11-5*** el sistema no evolucionó según lo esperado. Por esto se decidió no realizar más pruebas en dicho computador.

Figura 11-3

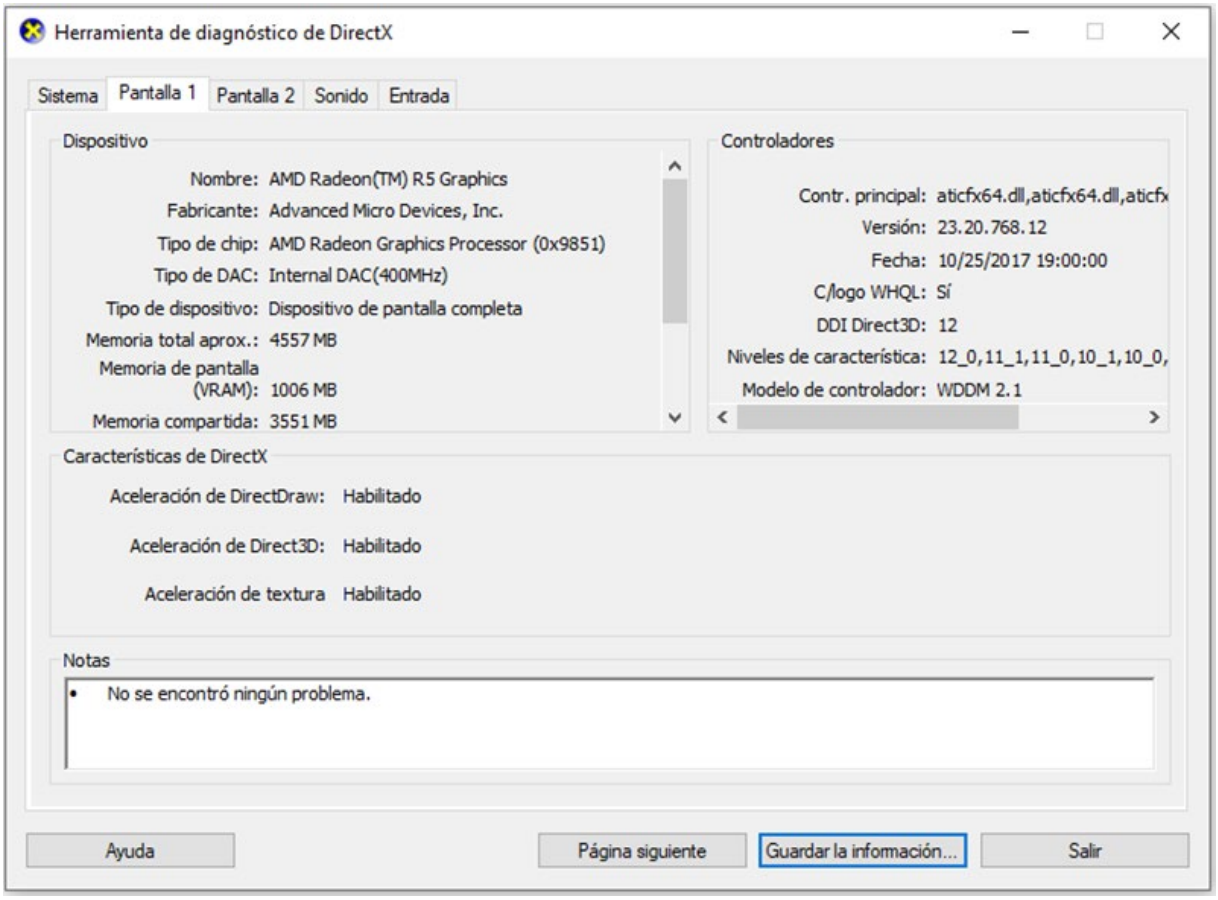
Especificaciones (sistema) del pc utilizado en las primeras pruebas del entrenamiento (para un robot)



Nota. Entre las especificaciones más destacadas se encuentra que tiene un procesador AMD A8 con 4CPUs de 2GHz y 8GB de memoria RAM. Fuente. Propia

Figura 11-4

Especificaciones (pantalla 1) del pc utilizado en las primeras pruebas del entrenamiento (para un robot)



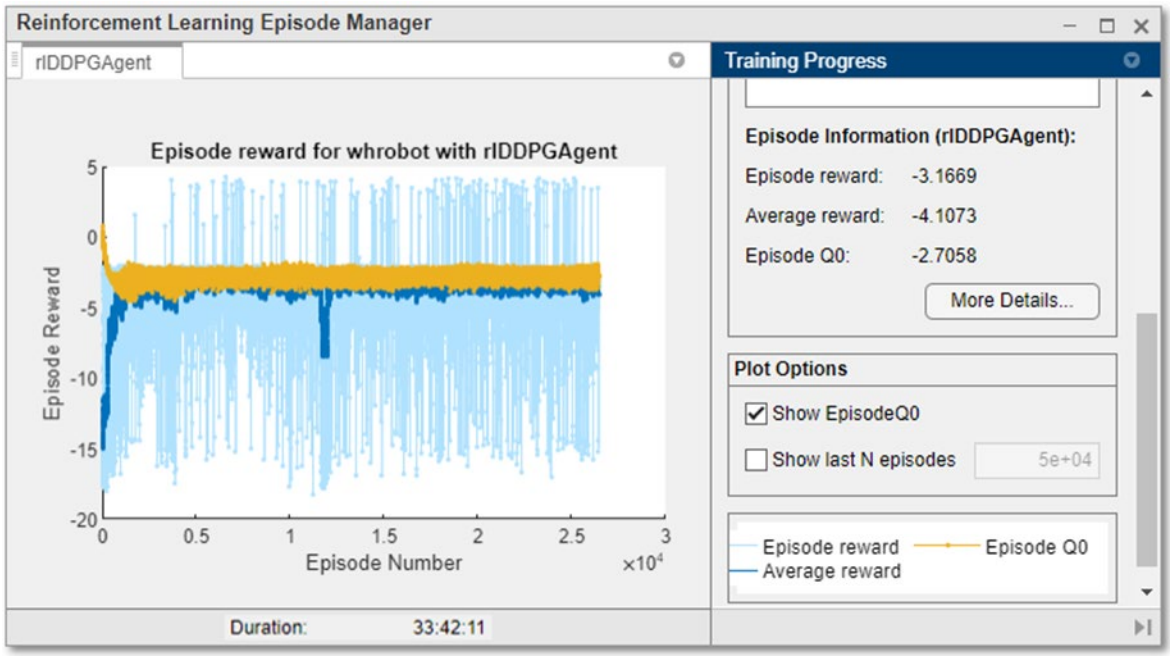
Nota. En especificaciones de pantalla se observa que cuenta con tarjeta gráfica interna de referencia AMD Radeon (TM) R5. Fuente. Propia

Durante la primera prueba que se realizó para un solo robot se decidió detener el entrenamiento cuando el número de episodios estaba alrededor de los 27000, pues no se evidenció que el sistema evolucionara según lo esperado. En la **Figura *11-5*** se observa que el sistema se mantiene sin presentar variaciones significativas a lo largo de los episodios, salvo algunas ocasiones que se obtuvieron recompensas positivas, sin embargo, no eran suficientes para que el crítico predijera valores positivos para la recompensa promedio en el

futuro y, si esto no sucedía, el actor no actualizaba los parámetros de su red neuronal para escoger acciones que lograran cumplir el objetivo, pues no se incita a ello.

Figura 11-5

Resultado del entrenamiento del agente para un robot (fallido)



Nota. Primero entrenamiento fallido para un agente con un robot, se tomó la decisión de detener el entrenamiento de manera forzada, pues no se observó una evolución favorable en las 33 horas de entrenamiento. Fuente. Elaboración propia

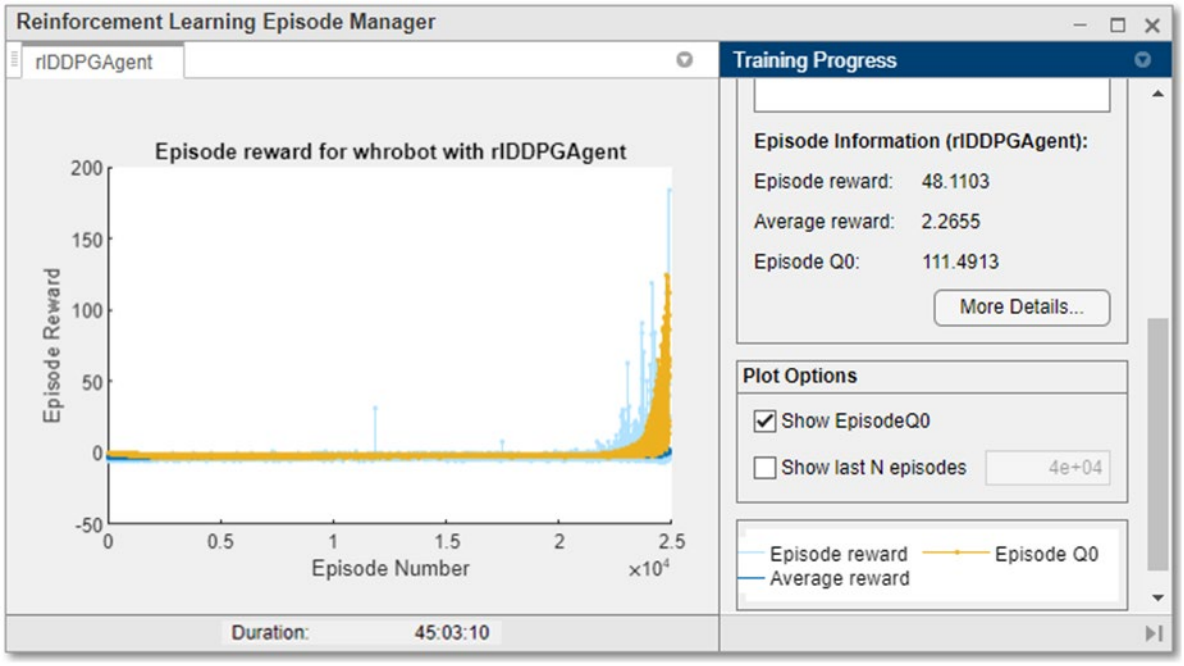
Inicialmente se estaba utilizando la función rlOptimizerOptions para definir la tasa de aprendizaje y el umbral del gradiente de las redes neuronales, tanto para el crítico como para el actor, pues según la página oficial de MathWorks se recomienda utilizar esta nueva función debido a ciertos problemas de compatibilidad, sin embargo, se realizó una nueva prueba con la función rlRepresentationOptions que está definida para versiones anteriores a la 2021b de Matlab, con esta se obtuvieron los resultados esperados y que se evidencian en la sección 3.1.11, dicho modelo se entrenó en la versión 2022a de Matlab.

Durante la etapa inicial del proyecto se había tomado la decisión de entrenar a un solo agente para que controlada dos robots y obtener una base de datos que se utilizaría

como agente preentrenado al momento de entrenar los dos agentes, sin embargo, en este modelo se encontraron múltiples dificultades debido a que el agente no era "consciente" de los dos robots, los procesaba como todo, es decir, únicamente tenía doce entradas y cuatro acciones, pero no identificaba a qué correspondía cada una, por ello, en el primer entrenamiento el agente aprendió que, al dejar un robot girando en círculos en una posición sin que chocara con ningún obstáculo, y llevando al otro robot hacia una posición negativa sobre el eje y, lograba obtener una recompensa muy alta. esto se evidencia en la **Figura *11-6***. Al observar estos resultados se pusieron algunas restricciones en cuanto a los límites del mapa, contando como colisión sí salían de estos, adicionalmente se colocó un límite en la parte inferior el valor en $y = -4$, encontrando que al tener tantas restricciones el crítico no incentivaba al actor a explorar nuevas acciones como se observa en la **Figura *11-7***.

Figura 11-6

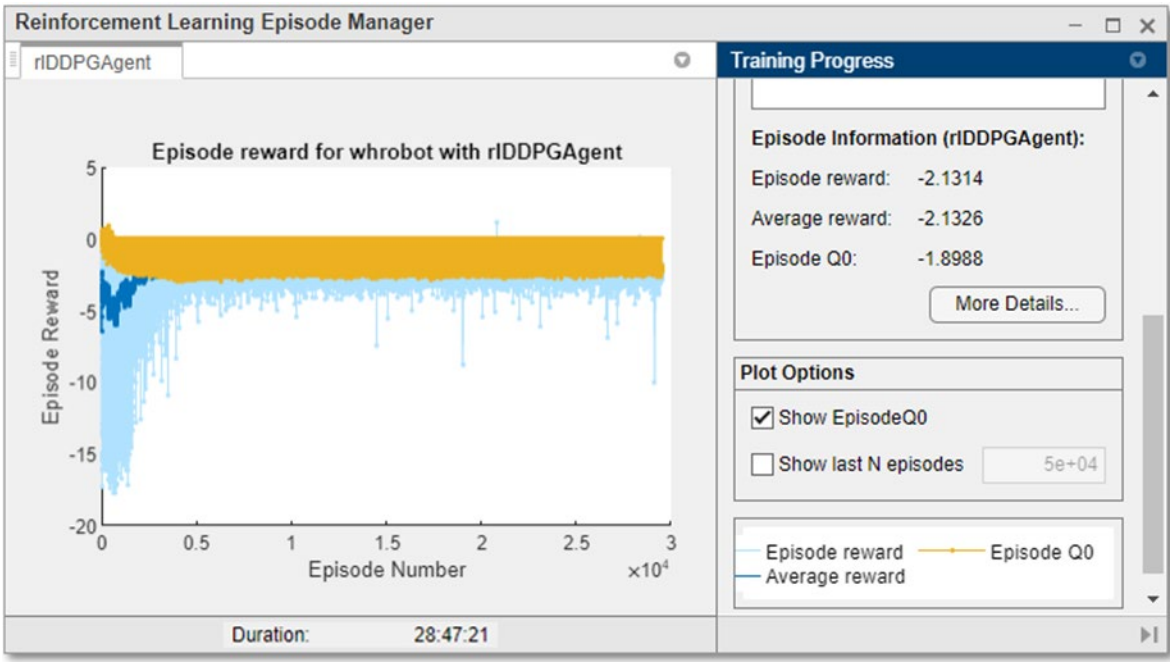
Resultado del entrenamiento de un agente con 2 robots (prueba 1)



Nota. En esta primera prueba las restricciones se limitaban al choque con los obstáculos en color negro. Fuente. Elaboración propia

Figura 11-7

Resultado del entrenamiento de un agente con 2 robots (prueba 2)

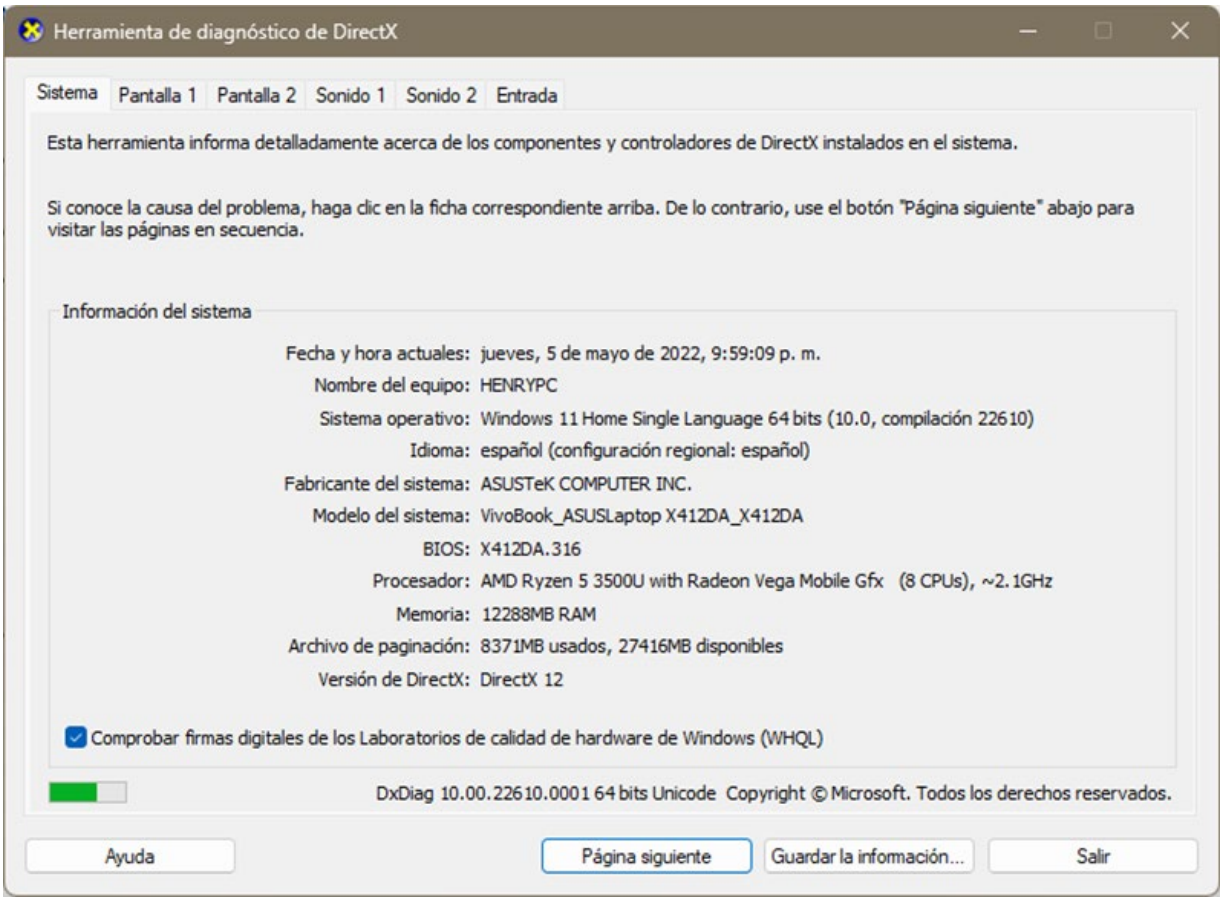


Nota. En esta prueba se incluyeron las restricciones que determinaban los límites de entorno como una colisión. Fuente. Elaboración propia

En las **Figura *11-8*** y **Figura *11-9*** se observan las especificaciones del computador en el cual se obtuvo el resultado del entrenamiento presentado en sección **3.1.11**. También fue el dispositivo en el cual se realizó una gran cantidad de pruebas, pues al no tener suficiente conocimiento sobre este campo de estudio se realizó un alto número de simulaciones en las que se modificaba levemente un parámetro y se ponía nuevamente a entrenar el agente, esto para examinar cómo se afectaba el comportamiento con dicho cambio. Sin embargo, para obtener el resultado final presentado en el presente proyecto se utilizó el computador con las especificaciones mostradas en las **Figura *11-10*** y **Figura *11-11***.

Figura 11-8

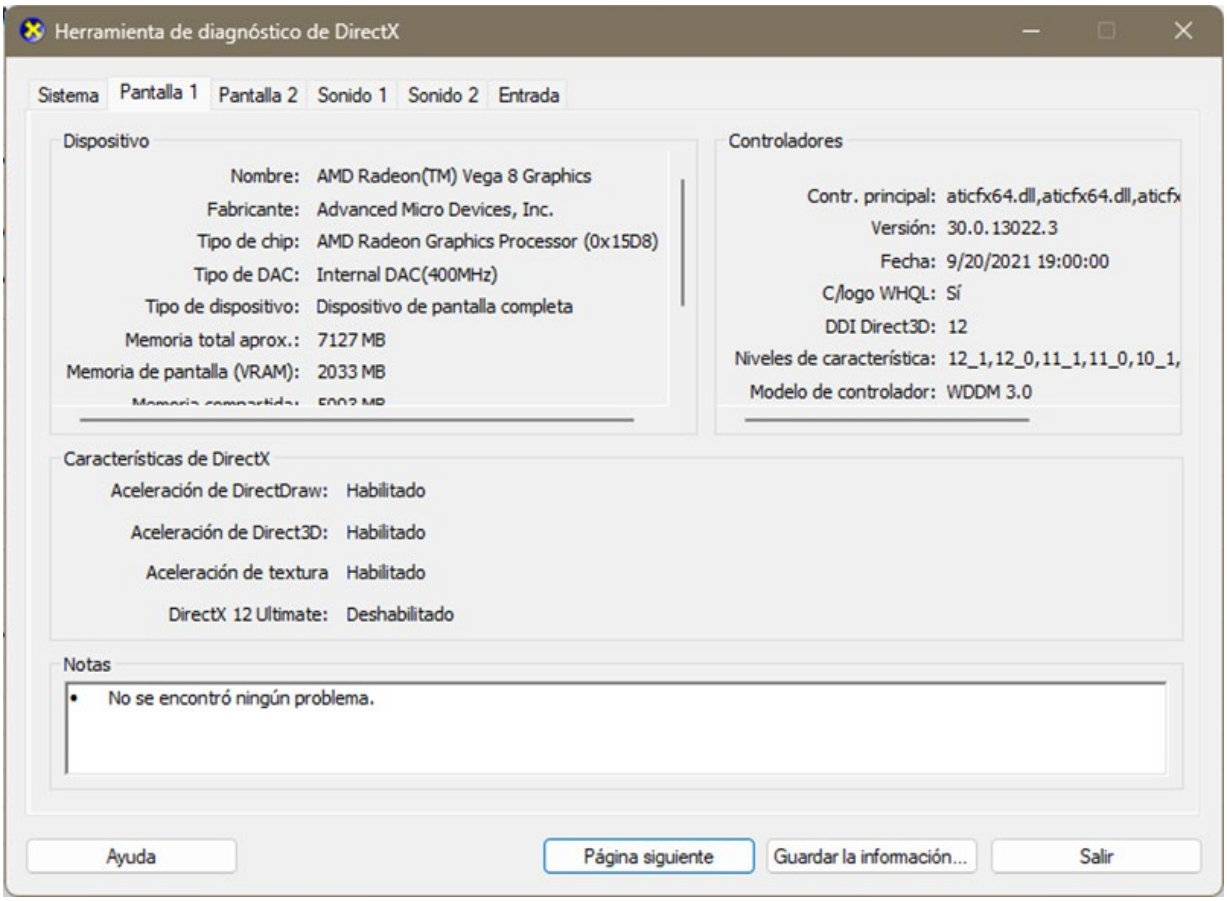
Especificaciones (sistema) del computador portátil



Nota. Entre las especificaciones más destacadas se encuentra que tiene un procesador AMD Ryzen 5 con 8CPUs de 2.1GHz y 12GB de memoria RAM. Fuente. Propia

Figura 11-9

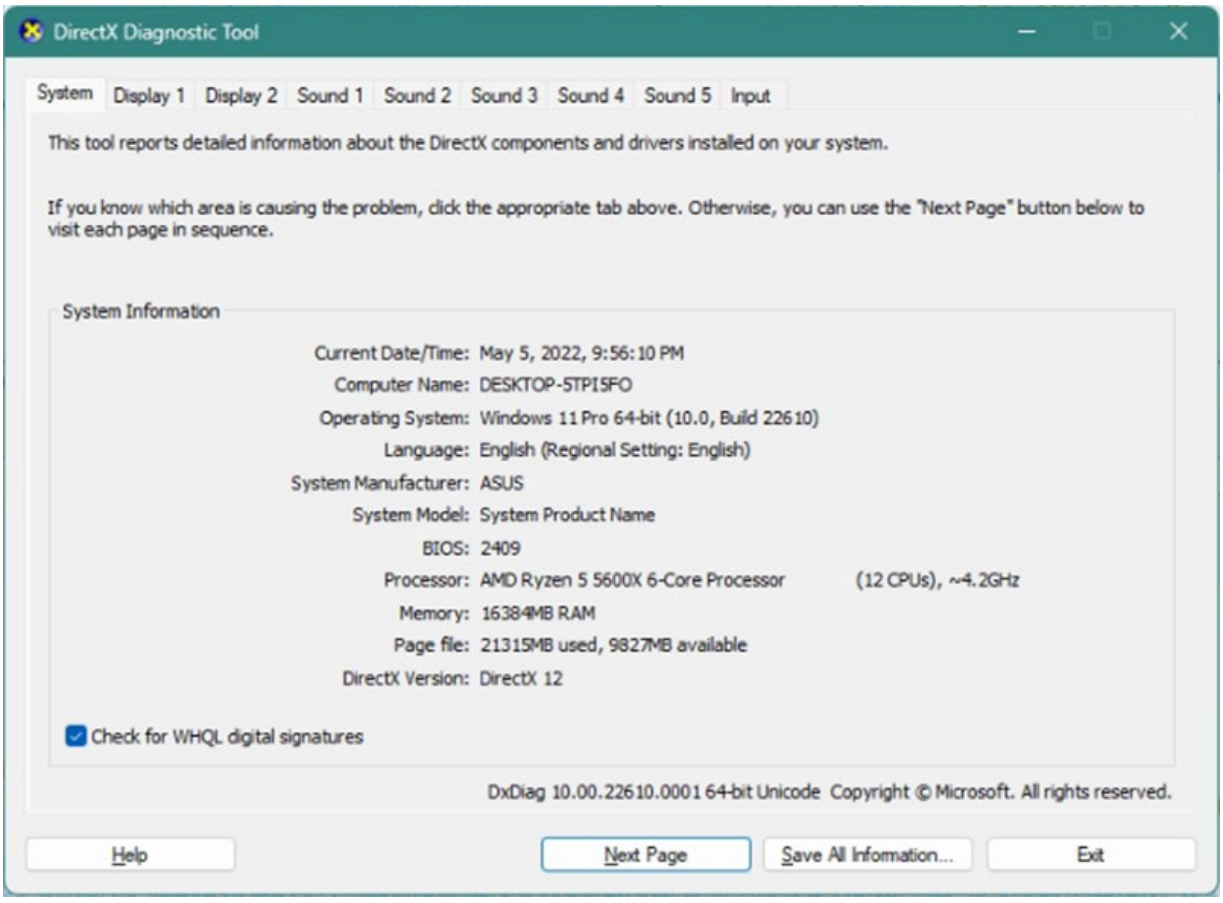
Especificaciones (pantalla 1) del computador portátil



Nota. En especificaciones de pantalla se observa que cuenta con tarjeta gráfica interna de referencia AMD Radeon (TM) Vega 8. Fuente. Propia

Figura 11-10

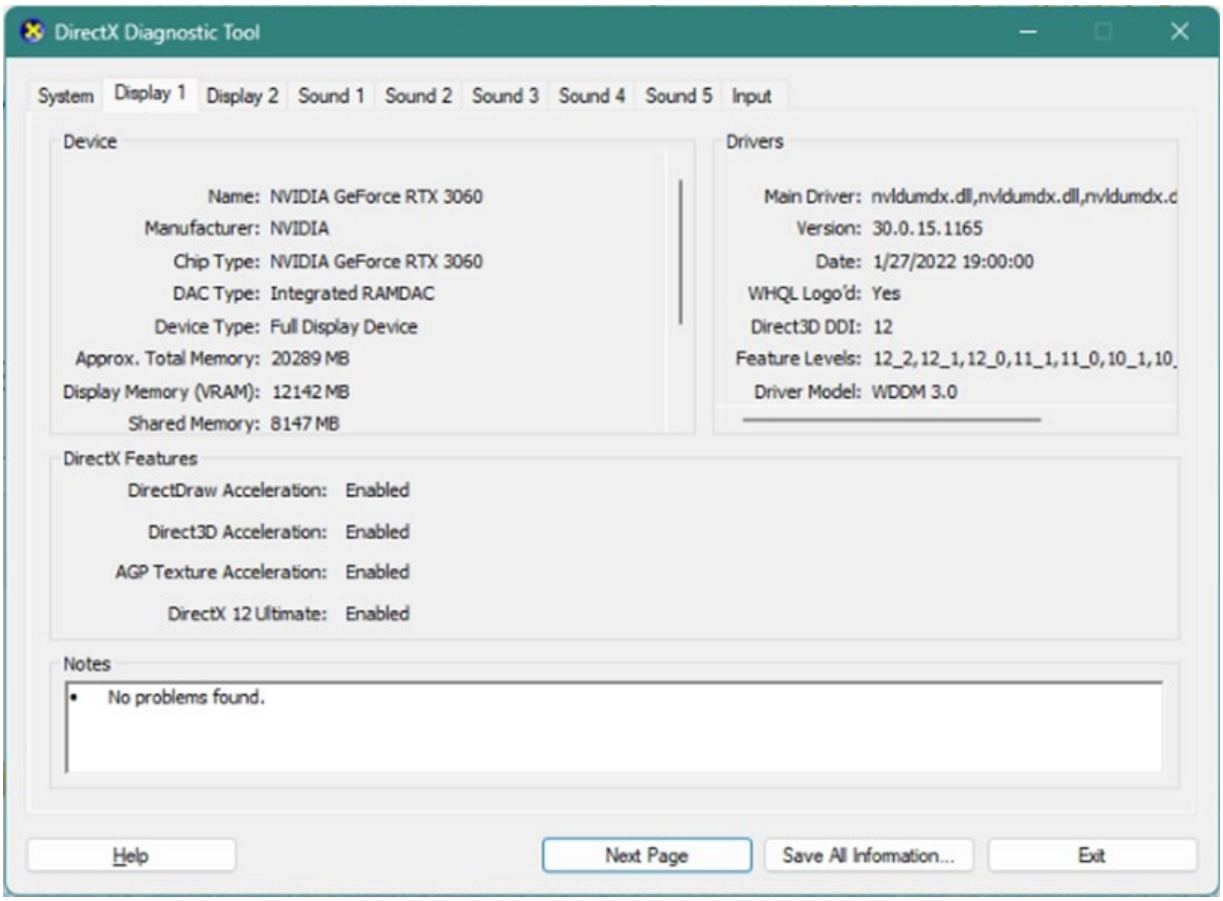
Especificaciones (sistema) del computador de escritorio



Nota. Entre las especificaciones más destacadas se encuentra que tiene un procesador AMD Ryzen 5 con 12CPUs de 4.2GHz y 16GB de memoria RAM. Fuente. Propia

Figura 11-11

Especificaciones (pantalla 1) del computador de escritorio



Nota. En especificaciones de pantalla se observa que cuenta con tarjeta gráfica integrada de referencia NVIDIA Geforce RTX 3060. Fuente. Propia

11.3 Anexo C: Agentes disponibles en Matlab

A continuación, se presenta una sección tomada del libro *Reinforcement Learning Toolbox: User's Guide* disponible en: https://www.mathworks.com/help/reinforcement-learning/ en la sección PDF Documentation (disponible con cuenta MathWorks).

Built-In Agents

Reinforcement Learning Toolbox software provides the following built-in agents. Each agent can be trained in environments with continuous or discrete observation spaces and the following action spaces.

Agent	Actions
"Q-Learning Agents" on page 4-4	Discrete
"SARSA Agents" on page 4-6	Discrete
"Deep Q-Network Agents" on page 4-8	Discrete
"Policy Gradient Agents" on page 4-11	Discrete or continuous
"Deep Deterministic Policy Gradient Agents" on page 4-14	Continuous
"Twin-Delayed Deep Deterministic Policy Gradient Agents" on page 4-17	Continuous
"Actor-Critic Agents" on page 4-21	Discrete or continuous
"Proximal Policy Optimization Agents" on page 4-24	Discrete or continuous

Custom Agents

You can also train policies using other learning algorithms by creating a custom agent. To do so, you create a subclass of a custom agent class, defining the agent behavior using a set of required and optional methods. For more information, see "Custom Agents" on page 4-28.

See Also

`rlACAgent` | `rlDDPGAgent` | `rlDQNAgent` | `rlPGAgent` | `rlPPOAgent` | `rlQAgent` | `rlSARSAAgent`

More About

- "What Is Reinforcement Learning?" on page 1-3
- "Train Reinforcement Learning Agents" on page 5-2

Q-Learning Agents

The Q-learning algorithm is a model-free, online, off-policy reinforcement learning method. A Q-learning agent is a value-based reinforcement learning agent which trains a critic to estimate the return or future rewards.

For more information on the different types of reinforcement learning agents, see “Reinforcement Learning Agents” on page 4-2.

Q-learning agents can be trained in environments with the following observation and action spaces.

Observation Space	Action Space
Continuous or discrete	Discrete

During training, the agent explores the action space using epsilon-greedy exploration. During each control interval the agent selects a random action with probability ϵ, otherwise it selects an action greedily with respect to the value function with probability 1-ϵ. This greedy action is the action for which the value function is greatest.

Critic Function

To estimate the value function, a Q-learning agent maintains a critic $Q(S,A)$, which is a table or function approximator. The critic takes observation S and action A as inputs and outputs the corresponding expectation of the long-term reward.

For more information on creating critics for value function approximation, see “Create Policy and Value Function Representations” on page 3-2.

When training is complete, the trained value function approximator is stored in critic $Q(S,A)$.

Agent Creation

To create a Q-learning agent:

1. Create a critic using an `rlQValueRepresentation` object.
2. Specify agent options using an `rlQAgentOptions` object.
3. Create the agent using an `rlQAgent` object.

Training Algorithm

Q-learning agents use the following training algorithm. To configure the training algorithm, specify options using `rlQAgentOptions`.

- Initialize the critic $Q(S,A)$ with random values.
- For each training episode:
  1. Set the initial observation S.
  2. Repeat the following for each step of the episode until S is a terminal state:
     a. For the current observation S, select a random action A with probability ϵ. Otherwise, select the action for which the critic value function is greatest.

$A = \max_A Q(S, A)$

To specify ϵ and its decay rate, use the `EpsilonGreedyExploration` option.

b Execute action A. Observe the reward R and next observation S'.

c If S' is a terminal state, set the value function target y to R. Otherwise set it to:

$y = R + \gamma \max_A Q(S', A)$

To set the discount factor γ, use the `DiscountFactor` option.

d Compute the critic parameter update.

$\Delta Q = y - Q(S, A)$

e Update the critic using the learning rate α.

$Q(S, A) = Q(S, A) + \alpha * \Delta Q$

Specify the learning rate when you create the critic representation by setting the `LearnRate` option in the `rlRepresentationOptions` object.

f Set the observation S to S'.

See Also

`rlQAgent` | `rlQAgentOptions`

More About

- "Reinforcement Learning Agents" on page 4-2
- "Create Policy and Value Function Representations" on page 3-2
- "Train Reinforcement Learning Agents" on page 5-2
- "Train Reinforcement Learning Agent in Basic Grid World" on page 1-8

SARSA Agents

The SARSA algorithm is a model-free, online, on-policy reinforcement learning method. A SARSA agent is a value-based reinforcement learning agent which trains a critic to estimate the return or future rewards.

For more information on the different types of reinforcement learning agents, see “Reinforcement Learning Agents” on page 4-2.

SARSA agents can be trained in environments with the following observation and action spaces.

Observation Space	Action Space
Continuous or discrete	Discrete

During training, the agent explores the action space using epsilon-greedy exploration. During each control interval the agent selects a random action with probability ϵ, otherwise it selects an action greedily with respect to the value function with probability 1-ϵ. This greedy action is the action for which the value function is greatest.

Critic Function

To estimate the value function, a SARSA agent maintains a critic $Q(S,A)$, which is a table or function approximator. The critic takes observation S and action A as inputs and outputs the corresponding expectation of the long-term reward.

For more information on creating critics for value function approximation, see “Create Policy and Value Function Representations” on page 3-2.

When training is complete, the trained value function approximator is stored in critic $Q(S,A)$.

Agent Creation

To create a SARSA agent:

1. Create a critic using an `rlQValueRepresentation` object.
2. Specify agent options using an `rlSARSAAgentOptions` object.
3. Create the agent using an `rlSARSAAgent` object.

Training Algorithm

SARSA agents use the following training algorithm. To configure the training algorithm, specify options using `rlSARSAAgentOptions`.

- Initialize the critic $Q(S,A)$ with random values.
- For each training episode:
  1. Set the initial observation S.
  2. For the current observation S, select a random action A with probability ϵ. Otherwise, select the action for which the critic value function is greatest.

$$A = \max_{A} Q(S, A)$$

To specify ϵ and its decay rate, use the `EpsilonGreedyExploration` option.

3 Repeat the following for each step of the episode until S is a terminal state:

a Execute action A. Observe the reward R and next observation S'.

b Select an action A' by following the policy from state `S'`.

$$A' = \max_{A'} Q(S', A')$$

c If S' is a terminal state, set the value function target y to R. Otherwise set it to:

$$y = R + \gamma Q(S', A')$$

To set the discount factor γ, use the `DiscountFactor` option.

d Compute the critic parameter update.

$$\Delta Q = y - Q(S, A)$$

e Update the critic using the learning rate α.

$$Q(S, A) = Q(S, A) + \alpha * \Delta Q$$

Specify the learning rate when you create the critic representation by setting the `LearnRate` option in the `rlRepresentationOptions` object.

f Set the observation S to S'.

g Set the action A to A'.

See Also

`rlSARSAAgent` | `rlSARSAAgentOptions`

More About

- “Reinforcement Learning Agents” on page 4-2
- “Create Policy and Value Function Representations” on page 3-2
- “Train Reinforcement Learning Agents” on page 5-2
- “Train Reinforcement Learning Agent in Basic Grid World” on page 1-8

Deep Q-Network Agents

The deep Q-network (DQN) algorithm is a model-free, online, off-policy reinforcement learning method. A DQN agent is a value-based reinforcement learning agent that trains a critic to estimate the return or future rewards. DQN is a variant of Q-learning. For more information on Q-learning, see “Q-Learning Agents” on page 4-4.

For more information on the different types of reinforcement learning agents, see “Reinforcement Learning Agents” on page 4-2.

DQN agents can be trained in environments with the following observation and action spaces.

Observation Space	Action Space
Continuous or discrete	Discrete

During training, the agent:

- Updates the critic properties at each time step during learning.
- Explores the action space using epsilon-greedy exploration. During each control interval the agent selects a random action with probability ϵ, otherwise it selects an action greedily with respect to the value function with probability 1-ϵ. This greedy action is the action for which the value function is greatest.
- Stores past experience using a circular experience buffer. The agent updates the critic based on a mini-batch of experiences randomly sampled from the buffer.

Critic Function

To estimate the value function, a DQN agent maintains two function approximators:

- Critic $Q(S,A)$ — The critic takes observation S and action A as inputs and outputs the corresponding expectation of the long-term reward.
- Target critic $Q'(S,A)$ — To improve the stability of the optimization, the agent periodically updates the target critic based on the latest critic parameter values.

Both $Q(S,A)$ and $Q'(S,A)$ have the same structure and parameterization.

For more information on creating critics for value function approximation, see “Create Policy and Value Function Representations” on page 3-2.

When training is complete, the trained value function approximator is stored in critic $Q(S,A)$.

Agent Creation

To create a DQN agent:

1. Create a critic using an `rlQValueRepresentation` object.
2. Specify agent options using an `rlDQNAgentOptions` object.
3. Create the agent using an `rlDQNAgent` object.

DQN agents support critics that use recurrent deep neural networks as functions approximators.

Training Algorithm

DQN agents use the following training algorithm, in which they update their critic model at each time step. To configure the training algorithm, specify options using `rlDQNAgentOptions`.

- Initialize the critic $Q(s,a)$ with random parameter values θ_Q, and initialize the target critic with the same values: $\theta_{Q'} = \theta_Q$.
- For each training time step:

  1. For the current observation S, select a random action A with probability ϵ. Otherwise, select the action for which the critic value function is greatest.

     $$A = \arg\max_A Q(S, A|\theta_Q)$$

     To specify ϵ and its decay rate, use the `EpsilonGreedyExploration` option.
  2. Execute action A. Observe the reward R and next observation S'.
  3. Store the experience (S,A,R,S') in the experience buffer.
  4. Sample a random mini-batch of M experiences (S_i,A_i,R_i,S'_i) from the experience buffer. To specify M, use the `MiniBatchSize` option.
  5. If S'_i is a terminal state, set the value function target y_i to R_i. Otherwise set it to:

     $$A_{\max} = \arg\max_{A'} Q(S_i', A'|\theta_Q)$$
     $$y_i = R_i + \gamma Q'(S_i', A_{\max}|\theta_{Q'}) \quad \text{(double DQN)}$$

     $$y_i = R_i + \gamma \max_{A'} Q'(S_i', A'|\theta_{Q'}) \quad \text{(DQN)}$$

     To set the discount factor γ, use the `DiscountFactor` option. To use double DQN, set the `UseDoubleDQN` option to `true`.
  6. Update the critic parameters by one-step minimization of the loss L across all sampled experiences.

     $$L_k = \frac{1}{M}\sum_{i=1}^{M}\left(y_i - Q(S_i, A_i|\theta_Q)\right)^2$$
  7. Update the target critic parameters depending on the target update method For more information, see “Target Update Methods” on page 4-9.
  8. Update the probability threshold ϵ for selecting a random action based on the decay rate specified in the `EpsilonGreedyExploration` option.

Target Update Methods

DQN agents update their target critic parameters using one of the following target update methods.

- **Smoothing** — Update the target parameters at every time step using smoothing factor τ. To specify the smoothing factor, use the `TargetSmoothFactor` option.

  $$\theta_{Q'} = \tau\theta_Q + (1-\tau)\theta_{Q'}$$
- **Periodic** — Update the target parameters periodically without smoothing (`TargetSmoothFactor = 1`). To specify the update period, use the `TargetUpdateFrequency` parameter.

- **Periodic Smoothing** — Update the target parameters periodically with smoothing.

To configure the target update method, create a `rlDQNAgentOptions` object, and set the `TargetUpdateFrequency` and `TargetSmoothFactor` parameters as shown in the following table.

Update Method	**`TargetUpdateFrequency`**	**`TargetSmoothFactor`**
Smoothing (default)	`1`	Less than `1`
Periodic	Greater than `1`	`1`
Periodic smoothing	Greater than `1`	Less than `1`

References

[1] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing Atari With Deep Reinforcement Learning," *NIPS Deep Learning Workshop*, 2013.

See Also

`rlDQNAgent` | `rlDQNAgentOptions`

More About

- "Reinforcement Learning Agents" on page 4-2
- "Create Policy and Value Function Representations" on page 3-2
- "Train Reinforcement Learning Agents" on page 5-2

Policy Gradient Agents

The policy gradient (PG) algorithm is a model-free, online, on-policy reinforcement learning method. A PG agent is a policy-based reinforcement learning agent which directly computes an optimal policy that maximizes the long-term reward.

For more information on the different types of reinforcement learning agents, see “Reinforcement Learning Agents” on page 4-2.

PG agents can be trained in environments with the following observation and action spaces.

Observation Space	Action Space
Discrete or continuous	Discrete or continuous

During training, a PG agent:

- Estimates probabilities of taking each action in the action space and randomly selects actions based on the probability distribution.
- Completes a full training episode using the current policy before learning from the experience and updating the policy parameters.

Actor and Critic Functions

PG agents represent the policy using an actor function approximator $\mu(S)$. The actor takes observation S and outputs the probabilities of taking each action in the action space when in state S.

To reduce the variance during gradient estimation, PG agents can use a baseline value function, which is estimated using a critic function approximator, $V(S)$. The critic computes the value function for a given observation state.

For more information on creating actors and critics for function approximation, see “Create Policy and Value Function Representations” on page 3-2.

Agent Creation

To create a PG agent:

1. Create an actor representation using an `rlStochasticActorRepresentation` object.
2. If you are using a baseline function, create a critic using an `rlValueRepresentation` object.
3. Specify agent options using the `rlPGAgentOptions` object.
4. Create the agent using an `rlPGAgent` object.

Training Algorithm

PG agents use the REINFORCE (Monte-Carlo policy gradient) algorithm either with or without a baseline. To configure the training algorithm, specify options using `rlPGAgentOptions`.

REINFORCE Algorithm

1. Initialize the actor $\mu(S)$ with random parameter values θ_μ.

2. For each training episode, generate the episode experience by following actor policy $\mu(S)$. To select an action, the actor generates probabilities for each action in the action space, then the agent randomly selects an action based on the probability distribution. The agent takes actions until it reaches the terminal state, S_T. The episode experience consists of the sequence:

   $$S_0, A_0, R_1, S_1, \ldots, S_{T-1}, A_{T-1}, R_T, S_T$$

   Here, S_t is a state observation, A_{t+1} is an action taken from that state, S_{t+1} is the next state, and R_{t+1} is the reward received for moving from S_t to S_{t+1}.
3. For each state in the episode sequence; that is, for t = 1, 2, ..., T-1, calculate the return G_t, which is the discounted future reward.

   $$G_t = \sum_{k=t}^{T} \gamma^{k-t} R_k$$
4. Accumulate the gradients for the actor network by following the policy gradient to maximize the expected discounted reward. If the `EntropyLossWeight` option is greater than zero, then additional gradients are accumulated to minimize the entropy loss function.

   $$d\theta_\mu = \sum_{t=1}^{T-1} G_t \nabla_{\theta_\mu} \ln \mu(S_t|\theta_\mu)$$
5. Update the actor parameters by applying the gradients.

   $$\theta_\mu = \theta_\mu + \alpha d\theta_\mu$$

   Here, α is the learning rate of the actor. Specify the learning rate when you create the actor representation by setting the `LearnRate` option in the `rlRepresentationOptions` object. For simplicity, this step shows a gradient update using basic stochastic gradient descent. The actual gradient update method depends on the optimizer specified using `rlRepresentationOptions`.
6. Repeat steps 2 through 5 for each training episode until training is complete.

REINFORCE with Baseline Algorithm

1. Initialize the actor $\mu(S)$ with random parameter values θ_μ.
2. Initialize the critic $V(S)$ with random parameter values θ_Q.
3. For each training episode, generate the episode experience by following actor policy $\mu(S)$. The episode experience consists of the sequence:

   $$S_0, A_0, R_1, S_1, \ldots, S_{T-1}, A_{T-1}, R_T, S_T$$
4. For t = 1, 2, ..., T:

   - Calculate the return G_t, which is the discounted future reward.

     $$G_t = \sum_{k=t}^{T} \gamma^{k-t} R_k$$
   - Compute the advantage function δ_t using the baseline value function estimate from the critic.

     $$\delta_t = G_t - V(S_t|\theta_V)$$
5. Accumulate the gradients for the critic network.

   $$d\theta_V = \sum_{t=1}^{T-1} \delta_t \nabla_{\theta_V} V(S_t|\theta_V)$$

6 Accumulate the gradients for the actor network. If the `EntropyLossWeight` option is greater than zero, then additional gradients are accumulated to minimize the entropy loss function.

$$d\theta_\mu = \sum_{t=1}^{T-1} \delta_t \nabla_{\theta_\mu} \ln \mu(S_t|\theta_\mu)$$

7 Update the critic parameters θ_V.

$$\theta_V = \theta_V + \beta d\theta_V$$

Here, β is the learning rate of the critic. Specify the learning rate when you create the critic representation by setting the `LearnRate` option in the `rlRepresentationOptions` object.

8 Update the actor parameters θ_μ.

$$\theta_\mu = \theta_\mu + \alpha d\theta_\mu$$

9 Repeat steps 3 through 8 for each training episode until training is complete.

For simplicity, this actor and critic updates in this algorithm show a gradient update using basic stochastic gradient descent. The actual gradient update method depends on the optimizer specified using `rlRepresentationOptions`.

References

[1] R. J. Williams, “Simple statistical gradient-following algorithms for connectionist reinforcement learning," *Machine Learning*, vol. 8, issue 3-4, pp. 229-256, 1992.

See Also

`rlPGAgent` | `rlPGAgentOptions`

More About

- “Reinforcement Learning Agents” on page 4-2
- “Create Policy and Value Function Representations” on page 3-2
- “Train Reinforcement Learning Agents” on page 5-2

Deep Deterministic Policy Gradient Agents

The deep deterministic policy gradient (DDPG) algorithm is a model-free, online, off-policy reinforcement learning method. A DDPG agent is an actor-critic reinforcement learning agent that computes an optimal policy that maximizes the long-term reward.

For more information on the different types of reinforcement learning agents, see “Reinforcement Learning Agents” on page 4-2.

DDPG agents can be trained in environments with the following observation and action spaces.

Observation Space	Action Space
Continuous or discrete	Continuous

During training, a DDPG agent:

- Updates the actor and critic properties at each time step during learning.
- Stores past experience using a circular experience buffer. The agent updates the actor and critic using a mini-batch of experiences randomly sampled from the buffer.
- Perturbs the action chosen by the policy using a stochastic noise model at each training step.

Actor and Critic Function

To estimate the policy and value function, a DDPG agent maintains four function approximators:

- Actor $\mu(S)$ — The actor takes observation S and outputs the corresponding action that maximizes the long-term reward.
- Target actor $\mu'(S)$ — To improve the stability of the optimization, the agent periodically updates the target actor based on the latest actor parameter values.
- Critic $Q(S,A)$ — The critic takes observation S and action A as inputs and outputs the corresponding expectation of the long-term reward.
- Target critic $Q'(S,A)$ — To improve the stability of the optimization, the agent periodically updates the target critic based on the latest critic parameter values.

Both $Q(S,A)$ and $Q'(S,A)$ have the same structure and parameterization, and both $\mu(S)$ and $\mu'(S)$ have the same structure and parameterization.

When training is complete, the trained optimal policy is stored in actor $\mu(S)$.

For more information on creating actors and critics for function approximation, see “Create Policy and Value Function Representations” on page 3-2.

Agent Creation

To create a DDPG agent:

1. Create an actor using an `rlDeterministicActorRepresentation` object.
2. Create a critic using an `rlQValueRepresentation` object.
3. Specify agent options using an `rlDDPGAgentOptions` object.

4 Create the agent using an `rlDDPGAgent` object.

Training Algorithm

DDPG agents use the following training algorithm, in which they update their actor and critic models at each time step. To configure the training algorithm, specify options using `rlDDPGAgentOptions`.

- Initialize the critic $Q(S,A)$ with random parameter values θ_Q, and initialize the target critic with the same random parameter values: $\theta_{Q'} = \theta_Q$.
- Initialize the actor $\mu(S)$ with random parameter values θ_μ, and initialize the target actor with the same parameter values: $\theta_{\mu'} = \theta_\mu$.
- For each training time step:

  1 For the current observation S, select action $A = \mu(S) + N$, where N is stochastic noise from the noise model. To configure the noise model, use the `NoiseOptions` option.

  2 Execute action A. Observe the reward R and next observation S'.

  3 Store the experience (S,A,R,S') in the experience buffer.

  4 Sample a random mini-batch of M experiences (S_i,A_i,R_i,S'_i) from the experience buffer. To specify M, use the `MiniBatchSize` option.

  5 If S'_i is a terminal state, set the value function target y_i to R_i. Otherwise set it to:

  $$y_i = R_i + \gamma Q'\left(S_i', \mu'\left(S_i'|\theta_\mu\right)|\theta_{Q'}\right)$$

  The value function target is the sum of the experience reward R_i and the discounted future reward. To specify the discount factor γ, use the `DiscountFactor` option.

  To compute the cumulative reward, the agent first computes a next action by passing the next observation S_i' from the sampled experience to the target actor. The agent finds the cumulative reward by passing the next action to the target critic.

  6 Update the critic parameters by minimizing the loss L across all sampled experiences.

  $$L_k = \frac{1}{M}\sum_{i=1}^{M}\left(y_i - Q(S_i, A_i|\theta_Q)\right)^2$$

  7 Update the actor parameters using the following sampled policy gradient to maximize the expected discounted reward.

  $$\nabla_{\theta_\mu} J \approx \frac{1}{M}\sum_{i=1}^{M} G_{ai} G_{\mu i}$$

  $$G_{ai} = \nabla_A Q(S_i, A|\theta_Q) \quad \text{where } A = \mu\left(S_i|\theta_\mu\right)$$

  $$G_{\mu i} = \nabla_{\theta_\mu}\mu\left(S_i|\theta_\mu\right)$$

  Here, G_{ai} is the gradient of the critic output with respect to the action computed by the actor network, and $G_{\mu i}$ is the gradient of the actor output with respect to the actor parameters. Both gradients are evaluated for observation S_i.

  8 Update the target actor and critic parameters depending on the target update method. For more information see “Target Update Methods” on page 4-16.

For simplicity, the actor and critic updates in this algorithm show a gradient update using basic stochastic gradient descent. The actual gradient update method depends on the optimizer specified using `rlRepresentationOptions`.

Target Update Methods

DDPG agents update their target actor and critic parameters using one of the following target update methods.

- **Smoothing** — Update the target parameters at every time step using smoothing factor τ. To specify the smoothing factor, use the `TargetSmoothFactor` option.

  $\theta_{Q'} = \tau\theta_Q + (1-\tau)\theta_{Q'}$ (critic parameters)

  $\theta_{\mu'} = \tau\theta_\mu + (1-\tau)\theta_{\mu'}$ (actor parameters)
- **Periodic** — Update the target parameters periodically without smoothing (`TargetSmoothFactor = 1`). To specify the update period, use the `TargetUpdateFrequency` parameter.
- **Periodic Smoothing** — Update the target parameters periodically with smoothing.

To configure the target update method, create a `rlDDPGAgentOptions` object, and set the `TargetUpdateFrequency` and `TargetSmoothFactor` parameters as shown in the following table.

Update Method	`TargetUpdateFrequency`	`TargetSmoothFactor`
Smoothing (default)	`1`	Less than `1`
Periodic	Greater than `1`	`1`
Periodic smoothing	Greater than `1`	Less than `1`

References

[1] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra. "Continuous control with deep reinforcement learning," *International Conference on Learning Representations*, 2016.

See Also

`rlDDPGAgent` | `rlDDPGAgentOptions`

More About

- "Reinforcement Learning Agents" on page 4-2
- "Create Policy and Value Function Representations" on page 3-2
- "Train Reinforcement Learning Agents" on page 5-2

Twin-Delayed Deep Deterministic Policy Gradient Agents

The twin-delayed deep deterministic policy gradient (TD3) algorithm is a model-free, online, off-policy reinforcement learning method. A TD3 agent is an actor-critic reinforcement learning agent that computes an optimal policy that maximizes the long-term reward.

For more information on the different types of reinforcement learning agents, see “Reinforcement Learning Agents” on page 4-2.

The TD3 algorithm is an extension of the DDPG algorithm. DDPG agents can overestimate value functions, which can produce suboptimal policies. To reduce value function overestimation includes the following modifications of the DDPG algorithm.

1. A TD3 agent learns two Q-value functions and uses the minimum value function estimate during policy updates.
2. A TD3 agent updates the policy and targets less frequently than the Q functions.
3. When updating the policy, a TD3 agent adds noise to the target action, which makes the policy less likely to exploit actions with high Q-value estimates.

You can use a TD3 agent to implement one of the following training algorithms, depending on the number of critics you specify.

- TD3 — Train the agent with two Q-value functions. This algorithm implements all three of the preceding modifications.
- Delayed DDPG — Train the agent with a single Q-value function. This algorithm trains a DDPG agent with target policy smoothing and delayed policy and target updates.

TD3 agents can be trained in environments with the following observation and action spaces.

Observation Space	Action Space
Continuous or discrete	Continuous

During training, a TD3 agent:

- Updates the actor and critic properties at each time step during learning.
- Stores past experience using a circular experience buffer. The agent updates the actor and critic using a mini-batch of experiences randomly sampled from the buffer.
- Perturbs the action chosen by the policy using a stochastic noise model at each training step.

Actor and Critic Function

To estimate the policy and value function, a TD3 agent maintains the following function approximators:

- Deterministic actor $\mu(S)$ — The actor takes observation S and outputs the corresponding action that maximizes the long-term reward.
- Target actor $\mu'(S)$ — To improve the stability of the optimization, the agent periodically updates the target actor based on the latest actor parameter values.
- One or two Q-value critics $Q_k(S,A)$ — The critics take observation S and action A as inputs and output the corresponding expectation of the long-term reward.

- One or two target critics $Q'_k(S,A)$ — To improve the stability of the optimization, the agent periodically updates the target critics based on the latest parameter values of the critics. The number of target critics matches the number of critics.

Both $\mu(S)$ and $\mu'(S)$ have the same structure and parameterization.

For each critic, $Q_k(S,A)$ and $Q'_k(S,A)$ have the same structure and parameterization.

When using two critics, $Q_1(S,A)$ and $Q_2(S,A)$, each critic can have a different structure, though TD3 works best when the critics have the same structure. When the critics have the same structure, they must have different initial parameter values.

When training is complete, the trained optimal policy is stored in actor $\mu(S)$.

For more information on creating actors and critics for function approximation, see “Create Policy and Value Function Representations” on page 3-2.

Agent Creation

To create a TD3 agent:

1. Create an actor using an `rlDeterministicActorRepresentation` object.
2. Create one or two critics using `rlQValueRepresentation` objects.
3. Specify agent options using an `rlTD3AgentOptions` object.
4. Create the agent using an `rlTD3Agent` object.

Training Algorithm

TD3 agents use the following training algorithm, in which they update their actor and critic models at each time step. To configure the training algorithm, specify options using `rlDDPGAgentOptions`. Here, $K = 2$ is the number of critics and k is the critic index.

- Initialize each critic $Q_k(S,A)$ with random parameter values θ_{Qk}, and initialize each target critic with the same random parameter values: $\theta_{Qk'} = \theta_{Qk}$.
- Initialize the actor $\mu(S)$ with random parameter values θ_μ, and initialize the target actor with the same parameter values: $\theta_{\mu'} = \theta_\mu$.
- For each training time step:
  1. For the current observation S, select action $A = \mu(S) + N$, where N is stochastic noise from the noise model. To configure the noise model, use the `ExplorationModel` option.
  2. Execute action A. Observe the reward R and next observation S'.
  3. Store the experience (S,A,R,S') in the experience buffer.
  4. Sample a random mini-batch of M experiences (S_i,A_i,R_i,S'_i) from the experience buffer. To specify M, use the `MiniBatchSize` option.
  5. If S'_i is a terminal state, set the value function target y_i to R_i. Otherwise set it to:

$$y_i = R_i + \gamma * \min_k\left(Q_{k'}\left(S_i', \text{clip}\left(\mu'\left(S_i'\middle|\theta_\mu\right) + \varepsilon\right)\middle|\theta_{Qk'}\right)\right)$$

The value function target is the sum of the experience reward R_i and the minimum discounted future reward from the critics. To specify the discount factor γ, use the `DiscountFactor` option.

To compute the cumulative reward, the agent first computes a next action by passing the next observation S'_i from the sampled experience to the target actor. Then, the agent adds noise ε to the computed action using the `TargetPolicySmoothModel`, and clips the action based on the upper and lower noise limits. The agent finds the cumulative rewards by passing the next action to the target critics.

6 At every time training step, update the parameters of each critic by minimizing the loss L_k across all sampled experiences.

$$L_k = \frac{1}{M}\sum_{i=1}^{M}\left(y_i - Q(S_i, A_i|\theta_Q)\right)^2$$

7 Every D_1 steps, update the actor parameters using the following sampled policy gradient to maximize the expected discounted reward. To set D_1, use the `PolicyUpdateFrequency` option.

$$\nabla_{\theta_\mu} J \approx \frac{1}{M}\sum_{i=1}^{M} G_{ai} G_{\mu i}$$

$$G_{ai} = \nabla_A \min_k\left(Q_k(S_i, A|\theta_Q)\right) \quad \text{where } A = \mu\left(S_i|\theta_\mu\right)$$

$$G_{\mu i} = \nabla_{\theta_\mu}\mu\left(S_i|\theta_\mu\right)$$

Here, G_{ai} is the gradient of the minimum critic output with respect to the action computed by the actor network, and $G_{\mu i}$ is the gradient of the actor output with respect to the actor parameters. Both gradients are evaluated for observation S_i.

8 Every D_2 steps, update the target actor and critics depending on the target update method. To specify D_2, use the `TargetUpdateFrequency` option. For more information, see “Target Update Methods” on page 4-19.

For simplicity, the actor and critic updates in this algorithm show a gradient update using basic stochastic gradient descent. The actual gradient update method depends on the optimizer specified using `rlRepresentationOptions`.

Target Update Methods

TD3 agents update their target actor and critic parameters using one of the following target update methods.

- **Smoothing** — Update the target parameters at every time step using smoothing factor τ. To specify the smoothing factor, use the `TargetSmoothFactor` option.

  $\theta_{Qk'} = \tau\theta_{Qk} + (1-\tau)\theta_{Qk'}$ (critic parameters)

  $\theta_{\mu'} = \tau\theta_\mu + (1-\tau)\theta_{\mu'}$ (actor parameters)

- **Periodic** — Update the target parameters periodically without smoothing (`TargetSmoothFactor = 1`). To specify the update period, use the `TargetUpdateFrequency` parameter.

  $\theta_{Qk'} = \theta_{Qk}$

  $\theta_{\mu'} = \theta_\mu$

- **Periodic Smoothing** — Update the target parameters periodically with smoothing.

To configure the target update method, create a `rlTD3AgentOptions` object, and set the `TargetUpdateFrequency` and `TargetSmoothFactor` parameters as shown in the following table.

Update Method	`TargetUpdateFrequency`	`TargetSmoothFactor`
Smoothing (default)	`1`	Less than `1`
Periodic	Greater than `1`	`1`
Periodic smoothing	Greater than `1`	Less than `1`

References

[1] Fujimoto, Scott, Herke van Hoof, and David Meger. 'Addressing Function Approximation Error in Actor-Critic Methods'. *ArXiv:1802.09477 [Cs, Stat]*, 22 October 2018. https://arxiv.org/abs/1802.09477.

See Also

`rlTD3Agent` | `rlTD3AgentOptions`

More About

- “Reinforcement Learning Agents” on page 4-2
- “Create Policy and Value Function Representations” on page 3-2
- “Train Reinforcement Learning Agents” on page 5-2
- “Train Biped Robot to Walk Using Reinforcement Learning Agents” on page 5-134

Actor-Critic Agents

You can use the actor-critic (AC) agent, which uses a model-free, online, on-policy reinforcement learning method, to implement actor-critic algorithms, such as A2C and A3C. The goal of this agent is to optimize the policy (actor) directly and train a critic to estimate the return or future rewards. [1]

For more information on the different types of reinforcement learning agents, see “Reinforcement Learning Agents” on page 4-2.

AC agents can be trained in environments with the following observation and action spaces.

Observation Space	Action Space
Discrete or continuous	Discrete or continuous

During training, an AC agent:

- Estimates probabilities of taking each action in the action space and randomly selects actions based on the probability distribution.
- Interacts with the environment for multiple steps using the current policy before updating the actor and critic properties.

Actor and Critic Function

To estimate the policy and value function, an AC agent maintains two function approximators:

- Actor $\mu(S)$ — The actor takes observation S and outputs the probabilities of taking each action in the action space when in state S.
- Critic $V(S)$ — The critic takes observation S and outputs the corresponding expectation of the discounted long-term reward.

When training is complete, the trained optimal policy is stored in actor $\mu(S)$.

For more information on creating actors and critics for function approximation, see “Create Policy and Value Function Representations” on page 3-2.

Agent Creation

To create an AC agent:

1. Create an actor using an `rlStochasticActorRepresentation` object.
2. Create a critic using an `rlValueRepresentation` object.
3. Specify agent options using an `rlACAgentOptions` object.
4. Create the agent using an `rlACAgent` object.

Training Algorithm

AC agents use the following training algorithm. To configure the training algorithm, specify options using an `rlACAgentOptions` object.

1. Initialize the actor $\mu(S)$ with random parameter values θ_{μ}.

2 Initialize the critic $V(S)$ with random parameter values θ_V.

3 Generate N experiences by following the current policy. The episode experience sequence is:

$$S_{ts}, A_{ts}, R_{ts+1}, S_{ts+1}, \ldots, S_{ts+N-1}, A_{ts+N-1}, R_{ts+N}, S_{ts+N}$$

Here, S_t is a state observation, A_t is an action taken from that state, S_{t+1} is the next state, and R_{t+1} is the reward received for moving from S_t to S_{t+1}.

When in state S_t, the agent computes the probability of taking each action in the action space using $\mu(S_t)$ and randomly selects action A_t based on the probability distribution.

ts is the starting time step of the current set of N experiences. At the beginning of the training episode, $ts = 1$. For each subsequent set of N experiences in the same training episode, $ts = ts + N$.

For each training episode that does not contain a terminal state, N is equal to the `NumStepsToLookAhead` option value. Otherwise, N is less than `NumStepsToLookAhead` and S_N is the terminal state.

4 For each episode step $t = ts+1, ts+2, \ldots, ts+N$, compute the return G_t, which is the sum of the reward for that step and the discounted future reward. If S_{ts+N} is not a terminal state, the discounted future reward includes the discounted state value function, computed using the critic network V.

$$G_t = \sum_{k=t}^{ts+N} \left(\gamma^{k-t} R_k\right) + b\gamma^{N-t+1} V(S_{ts+N}|\theta_V)$$

Here, b is `0` if S_{ts+N} is a terminal state and `1` otherwise.

To specify the discount factor γ, use the `DiscountFactor` option.

5 Compute the advantage function D_t.

$$D_t = G_t - V(S_t|\theta_V)$$

6 Accumulate the gradients for the actor network by following the policy gradient to maximize the expected discounted reward.

$$d\theta_\mu = \sum_{t=1}^{N} \nabla_{\theta_\mu} \ln \mu\left(S_t|\theta_\mu\right) * D_t$$

7 Accumulate the gradients for the critic network by minimizing the mean square error loss between the estimated value function $V(_t)$ and the computed target return G_t across all N experiences. If the `EntropyLossWeight` option is greater than zero, then additional gradients are accumulated to minimize the entropy loss function.

$$d\theta_V = \sum_{t=1}^{N} \nabla_{\theta_V} \left(G_t - V(S_t|\theta_V)\right)^2$$

8 Update the actor parameters by applying the gradients.

$$\theta_\mu = \theta_\mu + \alpha d\theta_\mu$$

Here, α is the learning rate of the actor. Specify the learning rate when you create the actor representation by setting the `LearnRate` option in the `rlRepresentationOptions` object.

9 Update the critic parameters by applying the gradients.

$$\theta_V = \theta_V + \beta d\theta_V$$

Here, β is the learning rate of the critic. Specify the learning rate when you create the critic representation by setting the `LearnRate` option in the `rlRepresentationOptions` object.

10 Repeat steps 3 through 9 for each training episode until training is complete.

For simplicity, the actor and critic updates in this algorithm show a gradient update using basic stochastic gradient descent. The actual gradient update method depends on the optimizer specified using `rlRepresentationOptions`.

References

[1] Mnih, V, et al. "Asynchronous methods for deep reinforcement learning," *International Conference on Machine Learning*, 2016.

See Also

`rlACAgent` | `rlACAgentOptions`

More About

- “Reinforcement Learning Agents” on page 4-2
- “Create Policy and Value Function Representations” on page 3-2
- “Train Reinforcement Learning Agents” on page 5-2

Proximal Policy Optimization Agents

The proximal policy optimization (PPO) is a model-free, online, on-policy, policy gradient reinforcement learning method. This algorithm is a type of policy gradient training that alternates between sampling data through environmental interaction and optimizing a clipped surrogate objective function using stochastic gradient descent. The clipped surrogate objective function improves training stability by limiting the size of the policy change at each step. [1]

For more information on the different types of reinforcement learning agents, see “Reinforcement Learning Agents” on page 4-2.

PPO agents can be trained in environments with the following observation and action spaces.

Observation Space	Action Space
Discrete or continuous	Discrete or continuous

During training, a PPO agent:

- Estimates probabilities of taking each action in the action space and randomly selects actions based on the probability distribution.
- Interacts with the environment for multiple steps using the current policy before using mini-batches to update the actor and critic properties over multiple epochs.

Actor and Critic Function

To estimate the policy and value function, a PPO agent maintains two function approximators:

- Actor $\mu(S)$ — The actor takes observation S and outputs the probabilities of taking each action in the action space when in state S.
- Critic $V(S)$ — The critic takes observation S and outputs the corresponding expectation of the discounted long-term reward.

When training is complete, the trained optimal policy is stored in actor $\mu(S)$.

For more information on creating actors and critics for function approximation, see “Create Policy and Value Function Representations” on page 3-2.

Agent Creation

To create a PPO agent:

1. Create an actor using an `rlStochasticActorRepresentation` object.
2. Create a critic using an `rlValueRepresentation` object.
3. Specify agent options using an `rlPPOAgentOptions` object.
4. Create the agent using the `rlPPOAgent` function.

PPO agents support actors and critics that use recurrent deep neural networks as functions approximators.

Training Algorithm

PPO agents use the following training algorithm. To configure the training algorithm, specify options using an `rlPPOAgentOptions`.

1. Initialize the actor $\mu(S)$ with random parameter values θ_μ.
2. Initialize the critic $V(S)$ with random parameter values θ_V.
3. Generate N experiences by following the current policy. The experience sequence is:

   $$S_{ts}, A_{ts}, R_{ts+1}, S_{ts+1}, \ldots, S_{ts+N-1}, A_{ts+N-1}, R_{ts+N}, S_{ts+N}$$

   Here, S_t is a state observation, A_t is an action taken from that state, S_{t+1} is the next state, and R_{t+1} is the reward received for moving from S_t to S_{t+1}.

   When in state S_t, the agent computes the probability of taking each action in the action space using $\mu(S_t)$ and randomly selects action A_t based on the probability distribution.

   ts is the starting time step of the current set of N experiences. At the beginning of the training episode, $ts = 1$. For each subsequent set of N experiences in the same training episode, $ts \leftarrow ts + N$.

   For each experience sequence that does not contain a terminal state, N is equal to the `ExperienceHorizon` option value. Otherwise, N is less than `ExperienceHorizon` and S_N is the terminal state.
4. For each episode step $t = ts+1, ts+2, \ldots, ts+N$, compute the return and advantage function using the method specified by the `AdvantageEstimateMethod` option.

   - **Finite Horizon** (`AdvantageEstimateMethod = "finite-horizon"`) — Compute the return G_t, which is the sum of the reward for that step and the discounted future reward. [2]

     $$G_t = \sum_{k=t}^{ts+N} \left(\gamma^{k-t} R_k\right) + b\gamma^{N-t+1} V(S_{ts+N}|\theta_V)$$

     Here, b is `0` if S_{ts+N} is a terminal state and `1` otherwise. That is, if S_{ts+N} is not a terminal state, the discounted future reward includes the discounted state value function, computed using the critic network V.

     Compute the advantage function D_t.

     $$D_t = G_t - V(S_t|\theta_V)$$

   - **Generalized Advantage Estimator** (`AdvantageEstimateMethod = "gae"`) — Compute the advantage function D_t, which is the discounted sum of temporal difference errors. [3]

     $$D_t = \sum_{k=t}^{ts+N-1} (\gamma\lambda)^{k-t} \delta_k$$

     $$\delta_k = R_t + b\gamma V(S_t|\theta_V)$$

     Here, b is `0` if S_{ts+N} is a terminal state and `1` otherwise. λ is a smoothing factor specified using the `GAEFactor` option.

     Compute the return G_t.

$$G_t = D_t - V(S_t|\theta_V)$$

To specify the discount factor γ for either method, use the `DiscountFactor` option.

5 Learn from experience mini-batches over K epochs. To specify K, use the `NumEpoch` option. For each learning epoch:

a Sample a random mini-batch data set of size M from the current set of experience. To specify M, use the `MiniBatchSize` option. Each element of the mini-batch data set contains a current experience and the corresponding return and advantage function values.

b Update the critic parameters by minimizing the loss L_{critic} across all sampled mini-batch data.

$$L_{critic}(\theta_V) = \frac{1}{M}\sum_{i=1}^{M}(G_i - V(S_i|\theta_V))^2$$

c Update the actor parameters by minimizing the loss L_{actor} across all sampled mini-batch data. If the `EntropyLossWeight` option is greater than zero, then additional entropy loss is added to L_{actor}, which encourages policy exploration.

$$L_{actor}(\theta_\mu) = -\frac{1}{M}\sum_{i=1}^{M}\min\left(r_i(\theta_\mu) * D_i, c_i(\theta_\mu) * D_i\right)$$

$$r_i(\theta_\mu) = \frac{\mu_{Ai}(S_i|\theta_\mu)}{\mu_{Ai}(S_i|\theta_{\mu,\,old})}$$

$$c_i(\theta_\mu) = \max\left(\min\left(r_i(\theta_\mu), 1+\varepsilon\right), 1-\varepsilon\right)$$

Here:

- D_i, G_i are the advantage function and return value for the ith element of the mini-batch, respectively.
- $\mu_i(S_i|\theta_\mu)$ is the probability of taking action A_i when in state S_i, given the updated policy parameters θ_μ.
- $\mu_i(S_i|\theta_{\mu,old})$ is the probability of taking action A_i when in state S_i, given the previous policy parameters ($\theta_{\mu,old}$) from before the current learning epoch.
- ε is the clip factor specified using the `ClipFactor` option.

6 Repeat steps 3 through 5 until the training episode reaches a terminal state.

References

[1] Schulman, J., et al. "Proximal Policy Optimization Algorithms," Technical Report, *ArXiv*, 2017.

[2] Mnih, V., et al. "Asynchronous methods for deep reinforcement learning," *International Conference on Machine Learning*, 2016.

[3] Schulman, J., et al. "High-Dimensional Continuous Control Using Generalized Advantage Estimation," Technical Report, *ArXiv*, 2018.

See Also

`rlPPOAgent` | `rlPPOAgentOptions`

12. Referencias Bibliográficas

Ambhore, S. (2020). A Comprehensive Study on Robot Learning from Demonstration. *2nd International Conference on Innovative Mechanisms for Industry Applications, ICIMIA 2020 - Conference Proceedings*, *Icimia*, 291–299. https://doi.org/10.1109/ICIMIA48430.2020.9074946

Arias Rivera, M. (2018). *Autonomous Navigation by Reinforcement Learning*.

Artola Moreno, Á. (2019). *Clasificación de imágenes usando redes neuronales convolucionales en Python*. Universidad de Sevilla.

Baker, B., Kanitscheider, I., Mrkov, T., Wu, Y., Powell, G., McGrew, B., & Mordatch, I. (2019). *Emergent Tool Use From Multi-Agent Autocurricula*.

Bañó, A. (2003). *Análisis y diseño del control de posición de un robot móvil con tracción diferencial*. http://deeea.urv.cat/public/PROPOSTES/pub/pdf/333pub.pdf

Bellemare, M., Candido, S., Castro, P. S., Gong, J., Machado, M., Subhodeep, M., Ponda, S., & Eang, Z. (2020). Autonomous navigation of stratospheric balloons using reinforcement learning. *Nature 588*, 77–82.

Calvo, D. (2017a). *Clasificación de redes neuronales artificiales*. https://www.diegocalvo.es/clasificacion-de-redes-neuronales-artificiales/

Calvo, D. (2017b). *Red Neuronal Convolucional CNN*. https://www.diegocalvo.es/red-neuronal-convolucional/

Calvo, D. (2018). *Función de activación – Redes neuronales*. https://www.diegocalvo.es/funcion-de-activacion-redes-neuronales/

Clinic, M. (2021). *Célula nerviosa (neurona)*. https://www.mayoclinic.org/es-es/nerve-cell-neuron/img-20007830

Contreras, L. (2003). *Estudio e Implementación de algunos comportamientos básicos de un animal en un robot de tipo genérico* [Tesis profesional, Universidad de las Américas Puebla]. http://catarina.udlap.mx/u_dl_a/tales/documentos/lis/contreras_o_l/

CSV, D. (2017). *AlphaGo Zero, el nuevo gran hito de DeepMind! | DATA COFFEE #1*. https://www.youtube.com/watch?v=yth1hXDfJ-g&ab_channel=DotCSV

de Lope, J. (2008). *Aprendizaje por Refuerzo en Robótica Autónoma*. Universidad Politécnica de Madrid.

Díaz Pinzón, C. A. (2018). Robótica aplicada a la elaboración de un brazo robótico solicionador de SUDOKUS por medio de redes neuronales. In *Politécnico Grancolombiano*.

Florensa, C., Held, D., Geng, X., & Abbeel, P. (2017). *Automatic Goal Generation for Reinforcement Learning Agents*.

Huang, B.-Q., Cao, G.-Y., & Guo, M. (2005). Reinforcement Learning Neural Network to the Problem of Autonomous Mobile Robot Obstacle Avoidance. *International Conference on Machine Learning and Cybernetics*, 85–89. https://doi.org/10.1109/ICMLC.2005.1526924

IAT. (n.d.). *INTELIGENCIA ARTIFICIAL Y ROBÓTICA: EL BINOMIO DEL FUTURO*. Retrieved April 25, 2022, from https://iat.es/tecnologias/inteligencia-artificial/robotica/

Isasi, P., & Galván, I. (2004). *Redes de Neuronas Artificiales: Un Enfoque Práctico* (Pearson Ed). Universidad Carlos II de Madrid.

Kubánková, E. (2020). *Karel Čapek, popularizador de la palabra 'robot', nació hace 130 años - Czech Radio*. https://espanol.radio.cz/karel-capek-popularizador-de-la-palabra-robot-nacio-hace-130-anos-8111099

Li, C., Zhang, J., & Li, Y. (2006). Application of Artificial Neural Network Based on Q-learning for Mobile Robot Path Planning. *2006 IEEE International Conference On Information Acquisition*, 978–982. https://doi.org/10.1109/ICIA.2006.305870

Lobos Tsunekawa, K. I. (2018). *APLICACIONES DEL APRENDIZAJE REFORZADO EN ROBÓTICA MÓVIL*. Universidad de Chile.

MathWorks. (n.d.-a). *Agentes de Q-Learning*. Retrieved March 29, 2022, from https://la.mathworks.com/help/reinforcement-learning/ug/q-agents.html

MathWorks. (n.d.-b). *Train Multiple Agents for Path Following Control*. Retrieved May 5, 2022, from https://la.mathworks.com/help/reinforcement-learning/ug/train-agents-for-path-following.html

MathWorks. (2019). *Reinforcement Learning Toolbox: User's Guide* (Inc. T. MathWorks, Ed.; 1.2).

MathWorks. (2020). *Reinforcement Learning with MATLAB*.

Mezeaa. (n.d.). *MT3005 Lección 12 - modelado de robots móviles - YouTube*. Retrieved April 26, 2022, from https://www.youtube.com/watch?v=5AqXC8ivZ8U&list=PLWIV1mZv4WUbgPWFhWgEucu7tTKollf8K&index=11&ab_channel=Mezeaa

Ng, A. (n.d.). *Redes neuronales y aprendizaje profundo - coursera*. Retrieved April 17, 2022, from https://www.coursera.org/learn/neural-networks-deep-learning

Palmer, A., & Montaño, J. (n.d.). *¿Qué son las redes neuronales artificiales? Aplicaciones realizadas en el ámbito de las adicciones*. https://disi.unal.edu.co/~lctorress/RedNeu/LiRna001.pdf

Peña Solórzano, C. A. (2015). *Aprendizaje robótico por imitación utilizando imágenes 2D y 3D*. Universidad Nacional de Colombia.

Pierson, H. A., & Gashler, M. S. (2017). Deep learning in robotics: a review of recent research. In *Advanced Robotics* (Vol. 31, Issue 16, pp. 821–835). Robotics Society of Japan. https://doi.org/10.1080/01691864.2017.1365009

Ramírez Serrano, A. (1996). *NAVEGACIÓN AUTÓNOMA DE ROBOTS MÓVILES MEDIANTE LÓGICA DIFUSA*. Instituto Tecnológico y de Estudios Superiores de Monterrey.

Ruiz Barreto, D. F., & Bravo Navarro, M. C. (2019). *Navegación Autónoma y Evasión de Obstáculos en UAV usando Aprendizaje por Refuerzo*. Universidad Santo Tomás.

Salazar, M. (n.d.). *La robótica*. Retrieved April 25, 2022, from https://la-rovotica.webnode.com.co/campos-de-aplicacion/ciencia-ficcion/

Semnani, S., Liu, H., Everett, M., Ruite, A., & How, J. (2020). Multi-Agent Motion Planning for Dense and Dynamic Environments via Deep Reinforcement Learning. *IEEE Robotics and Automation Letters*, *5*(2), 3221–3226. https://doi.org/10.1109/LRA.2020.2974695

Siciliano, B., Sciavicco, L., Villani, L., & Oriolo, G. (2009). *Robotics: Modelling, Planning and Control* (pp. 10–15). Springer.

Solaque, L., Molina, M., & Rodríguez, E. (2014). *Seguimiento de trayectorias con un robot móvil de configuración diferencial* (No. 1).

Strauss, C., & Sahin, F. (2008). Autonomous navigation based on a Q-learning algorithm for a robot in a real environment. *2008 IEEE International Conference on System of Systems Engineering*, 1–5. https://doi.org/10.1109/SYSOSE.2008.4724191

Tearle, M. (n.d.). *Reinforcement Learning Onramp*. MathWorks. https://matlabacademy.mathworks.com/es/details/reinforcement-learning-onramp/reinforcementlearning

Torres, J. (n.d.). *Introducción al aprendizaje por refuerzo profundo*. https://torres.ai/aprendizaje-por-refuerzo/

Watkins, C., & Peter, D. (1992). Q-learning. *Machine Learning*, *8*, 279–292.

Xu, N. (2021). Active Object Searching on Mobile Robot Using Reinforcement Learning. *2nd International Conference on Computing and Data Science (CDS)*, 296–300. https://doi.org/10.1109/CDS52072.2021.00058

Yan, N., Huang, S., & Kong, C. (2021). Reinforcement Learning-Based Autonomous Navigation and Obstacle Avoidance for USVs under Partially Observable Conditions. *Academi Editor: Hegazy Rezk*.