UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE INGENIERÍA

Interacción de un brazo robótico con un entorno de realidad virtual mediante aprendizaje por refuerzo profundo

TESIS

Que para obtener el título de

Ingeniero en Computación

P R E S E N T A

Alexis Fernando Aguilera Valderrama

DIRECTOR DE TESIS

Dr. Víctor Manuel Lomas Barrié

Ciudad Universitaria, Cd. Mx., 2024

健 友 職 知 術 愛

-Vida. Aguilera Valderrama Alexis Fernando

Agradecimientos

A mi asesor Víctor Manuel Lomas Barrie, por acercarme a un tema de tesis que me apasiona, por su confianza y paciencia. Además de darme las herramientas, consejos y conocimientos adecuados para hacer este trabajo y crecer profesionalmente.

A mi hermana por ser mi compañía más valiosa y alégrame siempre con su energía.

A mis padres por siempre ser un apoyo incondicional toda mi vida. A mi madre por ser la mejor psicóloga que pude tener, siempre diciéndome las palabras adecuadas para seguir adelante. A mi padre, por todas las mañanas que se levantó conmigo para salir a la escuela y siempre escuchar todos mis dilemas de la tesis y darme consejos muy útiles.

Al Doctor Ivan Vladimir Meza Ruíz por su apoyo en mi desarrollo académico y profesional. Además, por su fé en el trabajo que realizaba.

A mis amigos Fátima y José por brindarme su valiosa amistad, incluso en los momentos más difíciles para mi.

A mis compañeros de laboratorio del IIMAS Abner, Alan y Aura por su amistad, apoyo y consejos para realizar esta tesis. También al equipo de servicio social del laboratorio remoto por ser sujetos de pruebas.

A mis mascotas Miel y Denisse por ser una compañía al escribir y darme alegría en el día a día.

Y bueno, también a mi computadora, que rindió en los entrenamientos como una campeona.

Trabajo realizado con el apoyo del Programa UNAM-DGAPA-PAPIME PE111223

Índice general

1 Resumen **1**

1.1 Abstract 2

2 Planteamiento del problema **3**

3 Objetivo e hipótesis **10**

3.1 Hipótesis 10

3.2 Objetivo 10

3.3 Objetivos específicos 11

4 Marco teórico **12**

4.1 Caracterización del brazo robótico 12

4.2 Selección de ambiente virtual 17

4.2.1 Beat Saber y HTC Vive 17

4.3 Definición de destreza humana 21

4.4 Ambiente de simulación 22

4.4.1 Gazebo y ROS Noetic 22

4.5 Red neuronal artificial: perceptron multi-capa 24

4.6 Algoritmos de aprendizaje por refuerzo profundo 32

4.6.1 Aprendizaje por refuerzo profundo 32

4.6.2 Memoria de repetición de experiencia 40

4.6.3 Desempeño de algoritmos en robótica 40

4.6.4 Biblioteca de algoritmos de aprendizaje por refuerzo profundo . . . 41

4.6.5 Gradiente Determinista de Políticas Profundas (Deep Deterministic Policy Gradient - DDPG) 42

4.6.6 TD3 (Twin Delayed DDPG) . . . 44
4.6.7 HER (Hindsight Experience Replay) . . . 44
4.6.8 PPO (Proximal Policy Optimization) . . . 45
4.7 Algoritmos para la visión computacional . . . 47
4.7.1 Red neuronal convolucional . . . 47
4.7.2 Algoritmos de detección de objetos y su desempeño . . . 51
4.7.3 Modelo YOLO (You Only Look Once) . . . 53
4.8 Control jerárquico de un robot . . . 55
4.8.1 Control PID . . . 57
4.9 Consideraciones para transferencia de simulación a entorno real . . . 60

5 Metodología 62
5.1 Modelo simulado de MyCobot 280jn . . . 62
5.1.1 Estructura y propiedades físicas . . . 62
5.1.2 Control de robot . . . 65
5.2 Entrenamiento con redes convolucionales . . . 70
5.3 Entrenamiento virtual con detección de objetos . . . 73
5.3.1 Arquitectura de entrenamiento en ROS . . . 73
5.3.2 Arquitectura de agentes en paralelo . . . 81
5.3.3 Descripción de agente y ambiente para detección de objetos . . . 90
5.3.4 Entrenamiento paralelo de agentes . . . 100
5.4 Visión computacional usando YOLO . . . 114
5.5 Evaluación en entorno virtual . . . 120

6 Resultados y análisis 127
6.1 Evaluación en entorno físico . . . 127
6.1.1 Montado de entorno físico . . . 127
6.1.2 Métricas de desempeño físico . . . 132
6.1.3 Comparación Humano - robot . . . 144

7 Conclusiones y trabajo futuro 147

A Código 151

Índice de figuras

2.1 Ejemplo de entrenamiento en simulación y evaluación en la realidad. Adaptadas de *Boling Yang et al. "Competitive physical human-robot game play"* [11]. 6

4.1 Brazo robótico MyCobot 280 Jetson Nano. Obtenida de: https://top3dshop.com/product/elephant-robotics-mycobot-280-jetson-nano 13
4.2 Servo motor encoder STS3215 14
4.3 diagrama cinemático de Mycobot280jn 15
4.4 Imagen del entorno de *Beat Saber*. Obtenida de https://www.youtube.com/watch?v=jR3MnSIynOs 18
4.5 Set de realidad virtual 19
4.6 Acciones de Lighthouse tracking para estimar posiciones. Adaptadas de: https://www.youtube.com/watch?v=gV1sw4lfwFw 21
4.7 Diagrama de paradigma publicador-subscriptor de ROS. 23
4.8 Representación de perceptron simple. Adaptada de *Neural Networks and Learning Machines* [57] 25
4.9 Funciones de activación más frecuentes. Tomada de Medium, *Basic Overview of Convolutional Neural Network (CNN)* U. Udofia: https://n9.cl/es/s/f7lwtc. 26
4.10 Ejemplo de clasificación no lineal. Adaptada de *Neural Networks and Learning Machines.*[57] 27
4.11 Arquitectura de perceptron multi-capa. Adaptada de *Neural Networks and Learning Machines* [57] 27
4.12 Esquema de la dinámica de aprendizaje por refuerzo 33

4.13 Arquitectura de red neuronal para modelos basados en valor Q(s,a) 35
4.14 Arquitectura de red neuronal para modelos basados en valor V(s) 35
4.15 Arquitectura de red neuronal para modelos basados en política. 37
4.16 Arquitectura de redes neuronales de actor-crítico. 39
4.17 Logo de *stable-baselines 3.*. Obtenida de https://stable-baselines3.readthedocs.io/en/master/ 41
4.18 Arquitectura CNN de LetNet-5. Adaptada de *Gradient-Based Learning Applied to Document Recognition* [70] 47
4.19 Efectos de convolución de una imagen con un filtro. Imagen obtenida de https://setosa.io/ev/image-kernels/ 49
4.20 Reducción por agrupación Máxima 50
4.21 Aplanamiento del mapa de características 50
4.22 Métrica Intersection over Union para clasificación. Adaptada de *Learn OpenCV* https://learnopencv.com 52
4.23 Arquitectura de red neuronal convolucional de YOLO. Adaptada de *You Only Look Once: Unified, Real-Time Object Detection* [74] 53
4.24 Diagrama de paradigma YOLO. Adaptada de *You Only Look Once: Unified, Real-Time Object Detection* [74] 55
4.25 Diagrama de control PID. Adaptada de [76] 58
4.26 Posibles respuestas estables con variación de parámetros PID. Adaptada de https://docs.wpilib.org/es/latest/docs/software/advanced-controls/introduction/introduction-to-pid.html 59

5.1 Simulación de robot en Gazebo (derecha) y en rviz (izquierda). 64
5.2 Diagrama de control jerárquico para Mycobot280 jn. 65
5.3 Función no lineal (sigmoide) para control de velocidad de motores. 66
5.4 Diagrama de estados para evitar colisiones. 68
5.5 Respuesta de control PID de motores simulados. 69
5.6 Respuesta de control PID de motores reales. 69
5.7 Arquitectura asíncrona de entrenamiento en ROS para un solo robot. . . . 74

5.8 Sable simulado con dirección. 78
5.9 Posibles valores de C_{dir} dependiendo de la dirección del cubo. 79
5.10 Diferentes estatus de cubo visualizados. 80
5.11 Modelo estable (verde) modelo no deseado (amarillo). 82
5.12 Arquitectura de entrenamiento paralelo en ROS. 83
5.13 Múltiples robots simulados en Gazebo. 84
5.14 Múltiples tópicos para diferentes canales de comunicación. 84
5.15 Gráfica de recompensa de múltiples agentes entrenando al mismo tiempo. . 85
5.16 Generaciones (60K acciones cada una) a lo largo del entrenamiento. 86
5.17 Arquitectura central de administrador y agentes. 86
5.18 Diagrama de comunicación entre el nodo administrador y los nodos de agentes. 87
5.19 Ejemplo de reporte escrito por cada agente y selección del mejor. 88
5.20 Inicialización del ambiente del agente. 90
5.21 Mapeo de acciones continuas a discretas usando redondeo. 94
5.22 Arquitectura de redes neuronales actor y crítico para DDPG y TD3. 96
5.23 Arquitectura de redes neuronales actor y crítico para PPO. 96
5.24 Funciones de recompensa. 99
5.25 Funciones de recompensa para algoritmos con búffer HER. 99
5.26 Diferentes posiciones iniciales del robot. 102
5.27 Información de los agentes entrenando. 103
5.28 Tres robots mycobot entrenando la dinámica de *Beat Saber*. 103
5.29 Gráficas de recompensa y de pérdida de DDPG en entrenamiento. 105
5.30 Gráficas de recompensa y de pérdida de DDPG+HER en entrenamiento. . 106
5.31 Gráficas de recompensa y de pérdida de TD3 en entrenamiento. 107
5.32 Gráficas de recompensa y de pérdida de TD3+HER en entrenamiento. . . 108
5.33 Gráficas de recompensa y de pérdida de PPO en entrenamiento. 109
5.34 Comportamiento por políticas TD3 y DDPG 110
5.35 Comportamiento por política PPO. 111
5.36 Gráficas de pérdidas de modelo actor-crítico inestable (gris) con respecto a otros estables 112

5.37 Etiquetado de cubos para su entrenamiento en YOLO. 114
5.38 Estadísticas de precisión de YOLO. 115
5.39 Diagrama para clasificación de cubos. 117
5.40 Pasos para identificar el ángulo en diferentes tipos de cubo. 117
5.41 Sistema de visión computacional para identificación de cubos. 119
5.42 Arquitectura interacción robot simulado con *Beat Saber*. 120
5.43 Interacción de robot simulado con *Beat Saber*. 121

6.1 Piezas integradas a MyCobot para la interacción física. 128
6.2 Arquitectura para la interacción robot-HTC Vive. 129
6.3 Distancias de estaciones base y casco HTC Vive con respecto al robot. . . . 130
6.4 Altura de las estaciones base HTC Vive. 130
6.5 Altura de la mesa del robot. 131
6.6 Entorno físico montado y MyCobot jugando Beat Saber. 132
6.7 Movimiento simulado y real con modelo TD3+HER en dificultad fácil. . . 136
6.8 Movimiento simulado y real con modelo TD3+HER en dificultad Difícil. . 137
6.9 Movimiento simulado y real con modelo DDPG+HER en dificultad fácil. . 138
6.10 Movimiento simulado y real con modelo DDPG+HER en dificultad difícil. 139
6.11 Movimiento simulado y real con modelo PPO en dificultad fácil. 140
6.12 Movimiento simulado y real con modelo PPO en dificultad difícil. 141
6.13 Agente humano interactuando con el entorno de realidad virtual. 144

Índice de tablas

4.1 Especificaciones generales del robot MyCobot280 Jetson Nano 14
4.2 Especificaciones de Jetson Nano Developer 16

5.1 Características virtuales-físicas de mycobot 64

5.2 Limites de giro de cada motor 67
5.3 Configuración de parámetros PID para el robot 68
5.4 Espacios de acciones soportados para diferentes modelos. 92
5.5 Recompensa por resultado de corte de cubo. 97
5.6 Especificaciones de hardware para entrenamiento de agentes. 100
5.7 Resultados de entrenamiento de modelo DDPG. 105
5.8 Resultados de entrenamiento de modelo DDPG+HER. 106
5.9 Resultados de entrenamiento de modelo TD3. 107
5.10 Resultados de entrenamiento de modelo TD3+HER. 108
5.11 Resultados de entrenamiento de modelo PPO. 109
5.12 Conjunto de datos para el entrenamiento de YOLO. 115
5.13 Canciones seleccionadas para probar al robot. 122
5.14 Desempeño simulado de modelos en dificultad fácil. 124
5.15 Desempeño simulado de modelos en dificultad difícil. 125

6.1 Especificaciones de hardware para PC BEAT SABER / YOLO. 129
6.2 Desempeño real de los modelos en modo fácil. 133
6.3 Desempeño real de los modelos en modo difícil. 133
6.4 Desempeño promedio de agentes humanos. 145
6.5 Desempeño de TD3+HER en Sinister y comparación con humano. 145
6.6 Desempeño de DDPG+HER en Sinister y comparación con humano. . . . 145
6.7 Desempeño de PPO en Sinister y comparación con humano. 146

7.1 Resumen de precisión de cubos a lo largo del trabajo. 148

Capítulo 1

Resumen

En la última década, los avances en inteligencia artificial y en el hardware en robótica han sido prominentes, a tal grado de que se está logrando automatizar por completo cada parte del robot con la ayuda del aprendizaje profundo para darle una inteligencia parecida a la humana. En el campo de aprendizaje por refuerzo profundo (DRL, por sus siglas en inglés) aplicado en brazos robóticos, se ha logrado una fusión concisa, pero con tareas que no requieren tanta destreza y respuesta de reacción, como el levantamiento y arrastre de objetos, aproximación de posiciones o en deportes que requieren muchas pausas. Por ello, en este trabajo se propone un sistema impulsado por aprendizaje por refuerzo profundo que controla el brazo robótico de pequeña escala MyCobot 280jn para interactuar con un videojuego (exergame) de realidad virtual que establece una dinámica que demanda destreza; como lo es Beat Saber, el cual usa la interfaz HTC Vive. El objetivo de esto es probar si el agente es capaz de igualar la destreza humana con el brazo robótico propuesto. Cabe mencionar que también se incorporan técnicas de visión computacional como *You Only Look Once* (YOLO). Los algoritmos de DRL probados son *Deep Deterministic Policy Gradient* (DDPG), *Twin Delayed DDPG* (TD3), *Hindsight Experience Replay* (HER) y *Proximal Policy Optimization* (PPO). En *Beat Saber* se plantean dos dificultades de prueba, fácil (cortes sin dirección) y difícil (cortes con dirección), ambas con 8 canciones que van entre los 2.1 y 2.8 Cubos/segundo. Por lo tanto, los mejores resultados muestran que DDPG+HER tiene una precisión de 74.37 % en fácil y un 33.10 % en difícil; además, logra igualar en un 83.69 % a la destreza humana en fácil y un 47.65 % en difícil.

1.1. Abstract

In the last decade, advances in artificial intelligence and robotics hardware have been prominent, to the point that each part of the robot is being completely automated with the help of deep learning to give it human-like intelligence. In the field of deep reinforcement learning (DLR) applied in robotic arms, a concise fusion has been achieved, but with tasks that do not require as much dexterity and reaction response, such as lifting and dragging objects, approximation of positions or in sports that require many pauses. Therefore, in this work, a system powered by deep reinforcement learning is proposed, which controls the small-scale robotic arm MyCobot 280jn to interact with a virtual reality video game (exergame) that establishes a dynamic that demands dexterity; as is Beat Saber, which uses the HTC Vive interface. The goal of this is to test whether the agent is able to match human dexterity with the proposed robotic arm. It is worth mentioning that computer vision techniques such as *You Only Look Once* (YOLO) are also incorporated. The tested DLR algorithms are *Deep Deterministic Policy Gradient* (DDPG), *Twin Delayed DDPG* (TD3), *Hindsight Experience Replay* (HER) y *Proximal Policy Optimization* (PPO). In *Beat Saber* there are two test difficulties, easy (cuts without direction) and hard (cuts with direction), both with songs that range between 2.1 and 2.8 Cubes/second. Therefore, the best results showed that DDPG+HER has an accuracy of 74.37 % on easy and 33.10 % on hard; furthermore, it manages to match human skill by 83.69 % on easy and 47.65 % on hard.

Capítulo 2

Planteamiento del problema

Aprendizaje por refuerzo profundo, brazos robóticos y Exergaming

El aprendizaje por refuerzo profundo [1] es una técnica perteneciente al campo de aprendizaje por computadora mediante el uso de redes neuronales profundas. El principal objetivo de dicha técnica es dotar a un agente con una inteligencia que sea capaz de transformar un ambiente, ya sea físico o virtual, con acciones bien definidas para llevar a cabo una tarea. El comportamiento del agente está definido por una función probabilística llamada política $\pi(a|S;\theta)$ [1] la cual define la probabilidad π de llevar a cabo la acción (a) teniendo al agente y al ambiente en un estado (S) en cierto instante en el tiempo. El término θ se refiere a los pesos de la red neuronal a utilizar que será la encargada de estimar las probabilidades para cada acción que pueda realizar el agente en un determinado estado. El ajuste de estos parámetros π se hace mediante la definición de condiciones de recompensas para ciertas circunstancias que pueden presentarse en el ambiente, es decir, castigar o premiar al agente dependiendo de si las acciones realizadas por él, ayudan a resolver óptimamente el problema propuesto. Entonces, con esa información se logra entrenar a la red neuronal para poder completar una tarea.

En el campo de la robótica, este proceso de aprendizaje ha sido ampliamente usado debido a que evita la programación explícita del robot para la realización de tareas y permite explorar soluciones que podrían llegar a ser complejas de deducir por el razonamiento humano.

Las tareas más convencionales en las que se han puesto a brazos robots a realizar son aquellas que emulan algún comportamiento humano, por ejemplo: mover un brazo robótico hacia cierto punto en el espacio, apilar y empujar objetos [2] u otras acciones que requieren más destreza como darle vuelta a un pancake con un sartén [3], jugar *ping-pong* [4], tocar el piano [5], lanzar objetos a canastas [6], escribir braille en un teclado [7] y jugar *curling* [8].

Si bien todas las aplicaciones antes mencionadas han demostrado resultados satisfactorios que pueden competir con la capacidad humana, es crucial mencionar que los ambientes que manejan los agentes tienen un cierto grado de simpleza, haciendo que la tarea a resolver por el robot sea . Por ejemplo, en la línea de investigación del aprendizaje del juego *ping-pong*, *Google* [9] y la universidad *Universitat Tubigen* [10] han tenido desarrollos prominentes donde se han logrado tener brazos robóticos capaces de moverse con destreza y rapidez para poder desafiar a un jugador humano avanzado, logrando sesiones de más de 120 golpes. No obstante, los ambientes resultan ser sencillos por dos razones: 1) Cuando el robot devuelve la bola con la raqueta, tiene tiempo suficiente para regresar a una posición inicial y esperar el siguiente golpe. 2) Los brazos tienden a hacer movimientos repetitivos para lograr el golpe.

Otro caso donde se ha puesto a competir a un ser humano con un robot controlado por un agente fue en el trabajo de Yang [11]. En este se entrenó a un brazo robótico para que pueda realizar esgrima y bloquear los ataques de un agente humano. A pesar de que el brazo mostró una buena destreza y velocidad, el ambiente consiste en que el robot solo tenga que defender un punto estático que siempre estará en las mismas coordenadas todos los juegos, por lo que el ambiente no tiene una alta complejidad.

Entonces, en busca de un ambiente que desafíe la complejidad de las investigaciones antes mencionadas, se pueden considerar a los videojuegos como fuente de ambientes que desafíen a la destreza humana. Hay una extensa línea de investigación centrada en la

aplicación de algoritmos de aprendizaje por refuerzo profundo a videojuegos, ya que estos brindan entornos con dinámicas complejas que permiten medir la eficiencia y los límites de cada algoritmo. Algunos ejemplos de estas investigaciones han sido en el videojuego Doom [12], juegos de Atari [13], Dota 2 [14] y últimamente la inteligencia artificial general capaz de jugar una gran variedad de entornos llamada SIMA [15] desarrollada por *Google*; todos con resultados destacables que desafían a los jugadores más experimentados.

A pesar de ello, este tipo de investigaciones se han centrado en desarrollar agentes meramente virtuales, es decir, que no tienen una entidad física. Sin embargo, existen otros tipos de entornos virtuales que desafían aún más a la capacidad y destreza humana, ya que se necesita el movimiento completo o parcial del cuerpo para jugarlos, este es el caso de los llamados *exergames*. Acorde a Yoonsin O. y Stephen Y. [16] los *exergames* se pueden definir como *"videojuegos que requieren esfuerzo físico o movimientos que van más allá de las actividades sedentarias y que también incluyen actividades de fuerza, equilibrio y flexibilidad"*.

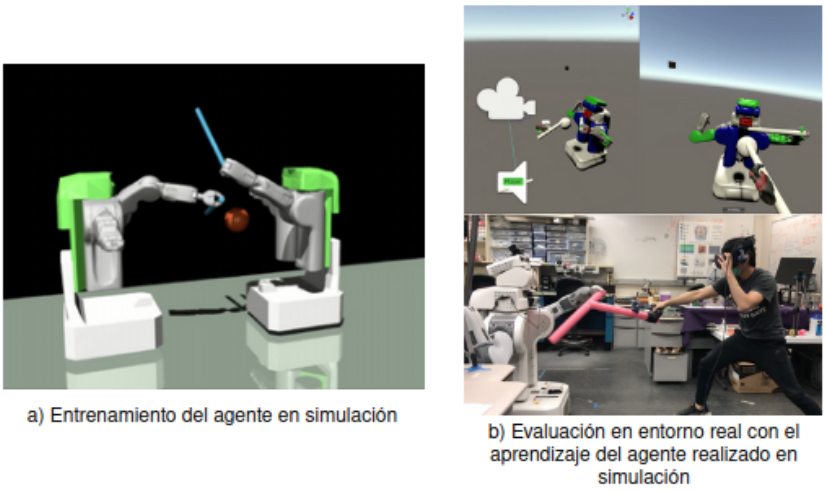
A lo largo de la historia ha habido una gran variedad de interfaces con su vasto repertorio de títulos de *exergames*. Acorde a Finco M. en [17] algunos ejemplos son: *Joyboard* de Atari, juegos de arcade como *Alpine Racer* o *Dance Dance Revolution* ,*Wii* y *Wii Fit* de Nintendo, *Kinect* de Xbox y más recientemente los sistemas de realidad virtual.

Por la popularidad de los sistemas de realidad virtual, tales como *Oculus Rift*, *HTC vive* y *Playstation VR*, ha habido estudios sobre sus efectos en el cuerpo humano al usarlos para *exergaming*. Otro trabajo relevante para este escrito es por el instituto de *Virtual Reality Institute of Health and Excersice* [18] donde compara el esfuerzo físico necesario en algunos *exergames* con ejercicios de la vida real, por ejemplo, títulos como *Beat Saber*, *Until you fall*, *Holopoint* y *Hot Squad* son comparables con jugar tenis por su desgaste físico y destreza rápida requerida; incluso se menciona que pueden llegar a una categoría más elevada en su dificultad máxima. Por lo anterior, estos títulos pueden ser propuestos para desafiar los ambientes ya mencionados de *ping-pong* y *esgrima*.

Entonces, por su nivel de complejidad y la destreza veloz necesaria para interactuar, hacer que un agente de inteligencia artificial pueda controlar un brazo robótico para destacar en un *exergame* desafiante es un paso para tener a un sistema que use el aprendizaje por refuerzo profundo que pueda competir con la destreza de un ser humano.

Retos de velocidad en ambientes complejos

En el aprendizaje de los agentes que controlan robots es común usar una simulación previa para entrenarlos de manera segura; es decir, sin daños a robots físicos. Esto se puede observar en los trabajos anteriormente mencionados [2, 3, 4, 6, 7, 8, 9, 10, 11]. Por ejemplo, en la figura 2.1 se puede observar el trabajo de Yang del robot esgrima, en el cual primero se entrenó al agente en un entorno virtual, para luego transferir el aprendizaje a un robot real.



a) Entrenamiento del agente en simulación

b) Evaluación en entorno real con el aprendizaje del agente realizado en simulación

Figura 2.1: Ejemplo de entrenamiento en simulación y evaluación en la realidad. Adaptadas de *Boling Yang et al. "Competitive physical human-robot game play"* [11].

El autor Kober [19] plantea retos a superar en el aprendizaje aplicado en la robótica considerando el uso de una simulación. A continuación, los retos que más conciernen a un ambiente complejo que cambia velozmente y a la rápida reacción del robot.

- **Muestras en el mundo real**: Con este problema, el autor menciona las diferencias que puede haber entre la simulación y la realidad, tales como desgaste de componentes, variantes en la iluminación, condiciones geográficas, entre otros. De este modo,

se hace énfasis en el tiempo de reacción de los componentes del robot. En simulación, la comunicación entre módulos del sistema y la reacción del robot pueden ser casi inmediatas, no obstante, en el mundo real, estos pueden llevar un tiempo considerable de latencia para llevar a cabo su tarea debido a que están conectados mediante medios físicos, lo que puede llegar a ralentizar la velocidad de reacción.

- **Modelado del robot**: Realizar un buen modelado físico del robot para la simulación siempre es una prioridad, para que así la transferencia a lo real sea lo más directo posible. Entonces, cuando el robot debe presentar movimientos veloces, este modelado puede llegar a la inestabilidad debido a torques repentinos en los motores virtuales. Un efecto de esto puede ser que alguna parte se quedara oscilando indefinidamente. Por ello, es necesario ajustar los parámetros de la simulación para lograr un movimiento lo más fiel a la realidad y que sea ágil.

- **Maldición de la dimensionalidad**: Este problema concierne al tamaño de las dimensiones del estado del ambiente y del espacio de acciones. Si tanto el vector del estado y de las acciones son muy grandes, se requerirá de más tiempo de cómputo para procesarlas en la red neuronal, reduciendo la tasa de acciones por segundo del agente, pero manteniendo un entendimiento más completo de la tarea. En cambio, si ambos vectores son de tamaño pequeño, el tiempo de cómputo se reduce y mejoran la tasa de acciones, pero sacrificando elementos que pueden ayudar a la solución de la tarea.

- **Especificación del objetivo**: El diseño de la recompensa hace referencia a los factores que debemos tomar en cuenta para castigar o premiar al agente. Para ello, se deben identificar los aspectos más relevantes y coherentes para resolver un problema e idear recompensas, ya que de considerar una recompensa que no sea muy importante o que contradiga a las demás, se puede producir ruido que ocasionará que el agente no tenga claro cómo resolver un problema, perdiendo así velocidad y destreza.

Además, el autor Kanervisto [20] describe retos a tratar con ambientes tipo videojuego. A continuación, se mencionan los que conciernen a la velocidad y complejidad del ambiente:

- **Ejecución asíncrona**: El autor menciona que a veces no es posible coordinar el juego con el agente para que así cada acción le corresponda a algún evento, esto por cuestiones de dinámica del juego o técnicas. Este hecho puede ser perjudicial para el aprendizaje, ya que si el ambiente va más rápido que el poder de procesamiento del agente, este no podrá tener el tiempo suficiente para responder adecuadamente a todos los eventos.

- **Observaciones parciales**: En un videojuego, no siempre todos los elementos existentes van a poder ser observables por el agente, por ejemplo, algún objeto detrás de una pared, algo detrás del jugador o las planificaciones de algún otro jugador o inteligencia artificial. Por lo tanto, esto puede acortar más el tiempo que tiene el agente para responder a un evento.

Todos estos puntos se deben considerar para tener a un agente que pueda reaccionar de manera rápida y correcta en un ambiente que presenta complejidad.

Retos de velocidad en visión computacional en robótica

El uso de visión computacional para obtener información del ambiente y así entrenar al modelo ha sido un reto al que se ha enfrentado la robótica. Existe una gran variedad de técnicas para obtener una representación del ambiente basado en imágenes, pero en el campo de manipulación por brazos robóticos, dos de las mas predominantes han sido el uso de redes neuronales convolucionles [21, 22, 23, 24, 25, 26] y la detección de objetos [27, 28, 29, 30] . A continuación una descripción de cada técnica.

- **Redes neuronales convolucionales** El uso de redes neuronales convolucionales ayuda a extraer características de una imagen mediante la aplicación de filtros, además de reducir la dimensionalidad de la misma para que le sea más fácil al agente aprender del ambiente. Esta técnica ha sido ampliamente utilizada en la robótica, esencialmente en la navegación de robots móviles [31] y en tareas de brazos robóticos

como levantamiento y acomodamiento de objetos [21, 22]. No obstante, cuando se usan este tipo de redes, el vector de características de la imagen es muy grande a comparación de otras alternativas para conseguir información del ambiente, cayendo así la maldición de la dimensionalidad antes mencionada y haciendo que al agente le tome más tiempo para procesar una acción o hacer más lento el proceso de aprendizaje.

- **Detección de objetos** El segundo método a considerar para dotar a un agente robótico de visión es mediante la detección de objetos. Este consiste en que mediante algún algoritmo, una computadora sea capaz de identificar un objeto dentro de una imagen y sea debidamente delimitado mediante un contorno, generalmente un rectángulo, el cual está representado por coordenadas X y Y sobre la imagen, que posteriormente es la información que recibe el agente. Asimismo, esta estrategia ha tenido varias aplicaciones con brazos robóticos, como en el arrastre de objetos [27] o en tareas más complejas como jugar fútbol [28]. Sin bien, este ya no presenta el problema de la dimensionalidad, la dificultad reside en la propia velocidad del algoritmo para detectar objetos. En los últimos años, han surgido algoritmos basados en redes neuronales que han logrado la detección de varios objetos en tiempo real, sin embargo, su correcto funcionamiento depende en gran medida de que se posea un hardware especializado para obtener el desempeño deseado.

En este trabajo se buscará determinar cual de los dos métodos conviene para la destreza del robot.

Capítulo 3

Objetivo e hipótesis

3.1. Hipótesis

Un algoritmo de aprendizaje por refuerzo profundo será capaz de manipular virtual y físicamente un brazo robótico que interactúa con un ambiente de realidad virtual de tipo *exergame* mediante interfaces para emular el movimiento humano con el fin de resolver una tarea que requiera una destreza considerable, basándose solamente en la imagen que produce el ambiente de realidad virtual y el estado del brazo robótico.

3.2. Objetivo

Con la hipótesis y lo mencionado en el capítulo Capitulo 2 que concierne a los retos de un agente de aprendizaje por refuerzo profundo de aprender en un ambiente que cambia muy frecuentemente, las dificultades del rendimiento en cuestiones de tiempo de los algoritmos de visión computacional y la falta de investigación al poner un brazo robótico a resolver una tarea que requiere destreza y velocidad continua se tiene el siguiente objetivo:

Entrenar virtualmente a un brazo robotico mediante aprendizaje por refuerzo profundo y visión computacional para que pueda realizar una tarea en un ambiente de realidad virtual que requiera destreza, para así transferirlo a un robot real y compararlo con la habilidad humana.

3.3. Objetivos específicos

- Elaborar una arquitectura de software que permita entrenar un brazo robótico mediante algoritmos de aprendizaje por refuerzo profundo para poder realizar una tarea que requiera destreza.
- Seleccionar algoritmos de aprendizaje por refuerzo profundo que puedan ser útiles en ambientes que cambian con alta frecuencia.
- Crear una simulación computacional que represente físicamente la interacción del brazo robótico con las interfaces de realidad virtual.
- Realizar un controlador del brazo robótico que sea compatible con las acciones que puede realizar un agente de aprendizaje por refuerzo.
- Implementar un algoritmo de visión computacional que permita analizar un ambiente virtual en tiempo real, es decir, que pueda procesar la imagen lo más rápido posible. Además, determinar cual algoritmo es mejor para el trabajo, si uno basado solamente en redes neuronales convolucionales o en detección de objetos, tomando en cuenta el tiempo de entrenamiento y la eficiencia para realizar la tarea.
- Transferir el aprendizaje del agente realizado en el robot simulado a uno en un entorno real y realizar una comparación de rendimiento para cada algoritmo a considerar.
- Realizar una muestra representativa de un grupo humano para caracterizar su destreza en la tarea a resolver, y así poder compararla con la del robot real.

Capítulo 4

Marco teórico

4.1. Caracterización del brazo robótico

En [32] los autores Svemir y Branko brindan una definición de los brazos robóticos de peso ligero. Mencionan que estos, como su nombre lo sugiere, son sistemas de bajo peso que están pensados para interactuar en un ambiente no estructurado y coexistir con el humano de manera segura, esto sin comprometer la capacidad de movimiento y carga efectiva del robot.

En [33] y [34] se hacen compilados sobre las características de brazos de bajo peso. Por ejemplo, el brazo robótico *PhantomX Reactor* pesa 1.36 kg y puede cargar 0.6 kg, *SG6-UT* con 1.06 kg carga 0.4 kg, y *Widowx Mark II* con 1.33 kg carga 0.8 kg. En estos mismos trabajos se menciona que, a pesar de sus capacidades limitadas, tienen una mayor aplicación en la educación y en la investigación debido a su gran facilidad de manipulación, ensamblado, programación, y mantenimiento. Por esta razón, estos sistemas han sido utilizados en varias líneas de investigaciones, tales como la manipulación mediante señales cerebrales [35], robots de servicio [36] y aplicaciones de aprendizaje por refuerzo profundo [37].

Por lo mencionado anteriormente, entre todo el mercado existente de robots de este tipo, se escogió el brazo MyCobot 280 Jetson Nano (Figura 4.2) desarrollado por la empresa Elephant Robotics[1]. Esta selección tomó en cuenta las diversas herramientas que

[1]Robot en la página: https://shop.elephantrobotics.com/en-mx/products/

ofrece para su funcionamiento, tales como: bibliotecas de ROS (Robot Operating System), modelos para su simulación virtual, sistema embebido con GPU integrado, su bajo costo con respecto a la competencia y las APIs de Python[2] y C++ para su manipulación física. Las especificaciones más relevantes del robot se muestran en la tabla 4.1.

Figura 4.1: Brazo robótico MyCobot 280 Jetson Nano. Obtenida de: https://top3dshop.com/product/elephant-robotics-mycobot-280-jetson-nano

mycobot-280-jetson-nano

[2]API pymycobot:https://github.com/elephantrobotics/pymycobot

Tabla 4.1: Especificaciones generales del robot MyCobot280 Jetson Nano

Grados de libertad (*DOF*)	6
Peso	860 g
Carga efectiva	250 g
Voltaje de entrada	DC 8.4-14V
Rango de trabajo	280mm
Precisión de posicionamiento	±0.5mm
Velocidad máxima de articulación	160°/s = 8/9*π
Rangos de libertad de articulaciones	
Articulación 1,2,3,4,5	-165° ∼+165 °
Articulación 6	-175° ∼+175°
Servo motores	
Modelo	STS3215
Torque	19kg.cm
Modo de conexión	*Daisy Chain*
Voltaje de entrada	7.4 V
Limite de giro	Sin limite

Figura 4.2: Servo motor encoder STS3215

Además, este brazo se ha utilizado con éxito en aplicaciones en las áreas de seguridad pública, la industria 4.0 - 5.0 y medicina [38, 39, 40]; sin embargo, este brazo no ha sido utilizado para aplicaciones de aprendizaje por refuerzo profundo.

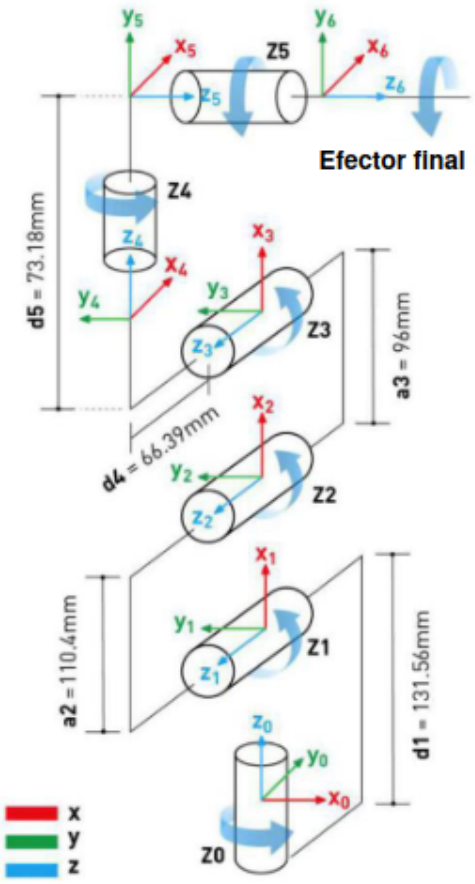


Figura 4.3: diagrama cinemático de Mycobot280jn

En la figura 4.3 se expresa el diagrama cinemático del brazo, el cual está disponible en la documentación del robot[3]. En él se expresan las características de cada articulación revolutiva ZN, tales como su dirección de giro, eje de coordenadas y distancia entre articulaciones. En el modelo se consideran inercias de tipo cilíndrico cuyo eje relativo z_n es el eje de rotación. En la sección 5.1.1 se revisan valores propuestos para la caracterización del robot en la simulación.

[3] https://www.elephantrobotics.com/wp-content/uploads/2021/03/myCobot-User-Mannul-EN-V20210318.pdf

Es de suma importancia mencionar que el brazo robótico, como su nombre sugiere, tiene incorporado un sistema embebido Jetson Nano desarrollado por la empresa Nvidia. Este hardware tiene dos objetivos:

- **Sistema operativo integrado**: Brindarle al robot un sistema operativo con bibliotecas y programas instalados para garantizar el funcionamiento básico del robot, es decir, para poder manejar los motores del brazo.
- **Poder computacional para aprendizaje de máquina**: La tarjeta Jetson Nano tiene una unidad GPU dedicada para operar eficazmente aplicaciones que requieran algoritmos basados en aprendizaje profundo, tal y como la marca *Nvidia* en su presentación[4]. En otras palabras, es para poder realizar inferencias rápidamente con redes neuronales. El desempeño de este componente ha logrado sobresalir con respecto a los dispositivos tradicionales para procesar un modelo de aprendizaje por computadora. Esto se ha demostrado en comparaciones donde se mide el desempeño en modelos de pequeña y mediana escala que involucran tareas de redes neuronales convolucionales, detección de objetos, *big data*, entre otros. [41, 42].

Descritos los objetivos de la tarjeta Jetson Nano, sus aspectos más importantes se pueden observar en la tabla 4.2[4].

Tabla 4.2: Especificaciones de Jetson Nano Developer

CPU	Quad-core ARM Cortex-A57 MPCore processor 1.43 GHz
GPU	NVIDIA Maxwell architecture with 128 NVIDIA CUDA cores
Memoria	4 GB 64-bit LPDDR4, 1600MHz 25.6 GB/s
Almacenamiento	microSD
Conectividad	Gigabit Ethernet, M.2 Key E
Sistema Operativo	Ubuntu 16.04

[4]NVIDIA, https://developer.nvidia.com/embedded/jetson-nano-developer-kit

4.2. Selección de ambiente virtual

4.2.1. Beat Saber y HTC Vive

Para poder escoger un entorno virtual demandante para un ser humano, se consideraron estudios que toman en cuenta la actividad física realizada en videojuegos de realidad virtual [43, 44, 45]. Los *exergames* que han mostrado mayor impacto son aquellos que requieren de un movimiento fundamental de cuerpo completo (piernas, brazos, cabeza y torso) para poder superarlos, tales como *Holopoint*, *Hot Squat* y *Thrill of the Fight*. Por otra parte, otros entornos donde sólo se requiere el movimiento parcial (brazos y/o torso) o lento de cuerpo completo, presentan una exigencia moderada. Estos pueden ser *Fruit Ninja VR*, *Relax Walk VR* y *Beat Saber*.

Revisando las condiciones del ambiente de los *exergames* antes mencionados, se determinó que el mejor para este proyecto es *Beat Saber*[5]. Este título fue lanzado el primero de mayo del año 2018 por la casa productora *Beat Games*. La dinámica del juego, se caracteriza por manejar un ambiente musical donde el jugador posee dos sables de luz con los cuales tiene que cortar cubos que se aproximan a él en una dirección en específico para poder ganar puntos. Estos cubos van apareciendo al ritmo de una pista musical. En la figura 4.4 se puede ver una representación del entorno, donde una jugadora primero corta un cubo rojo y después de aproximadamente 0.4 segundos corta otro cubo gris, ambos en una dirección concreta.

[5]Beat Games, Beat Saber, https://beatsaber.com/

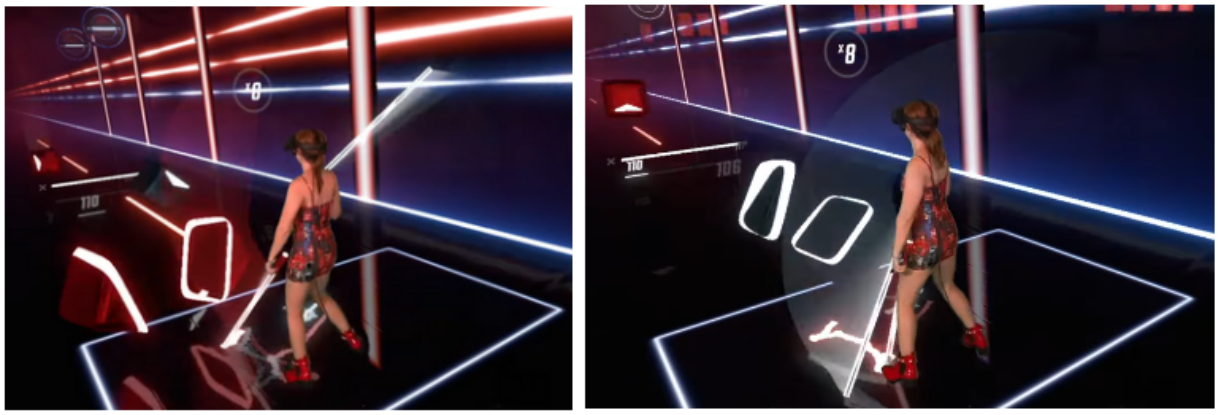


Figura 4.4: Imagen del entorno de *Beat Saber*. Obtenida de https://www.youtube.com/watch?v=jR3MnSIynOs

Con el ambiente descrito, a continuación se mencionan las razones para la selección de *Beat Saber*:

- **Modo de juego de un brazo**: El título tiene varios modos de juego, el más común es con dos sables; sin embargo, existe el modo sable único donde solo es necesario el movimiento de un brazo para jugar, esto es útil tomando en cuenta la posesión de un solo brazo robótico MyCobot280jn.

- **Variedad de dificultades**: El título ofrece varios modificadores que pueden facilitar la dinámica o hacerla más difícil. Con esto en mente, se puede modificar la dificultad del ambiente tal forma que se adapte bien a las capacidades del MyCobot280jn.

- **Destreza considerable**: A pesar de que sea catalogado de exigencia moderada, en los estudios realizados por *Virtual Reality Institute of Health and Excersice* [18] compara la exigencia física y de destreza de *Beat Saber* en una dificultad moderada con la de jugar tennis en la vida real, ejercicio que puede competir con los de esgrima [11] y *ping-pong* [9, 10] explicados en el capítulo 2.

- **Mirada siempre adelante**: Los cubos siempre van a venir de enfrente, por lo que no es necesario crear un mecanismo que tenga voltear a algún lado para poder ver un elemento del ambiente.

- **Ambiente casi totalmente observable**: Como consecuencia del punto anterior, la mayoría del tiempo todos los cubos con su dirección de corte siempre estarán a la vista del jugador, a excepción de que uno esté detrás de otro.

Cabe mencionar que el título se encuentra en la tienda *Steam* y es compatible con el kit de realidad virtual HTC Vive [6].

Con respecto a la interfaz HTC Vive, éste fue lanzado el 5 de abril del 2016 por la empresa *HTC, with technology by Valve*. El kit consiste en un casco de realidad virtual, dos mandos y dos estaciones bases infrarrojas para la detección de los componentes y estimación de las dimensiones de la sala. En la figura 4.5 se pueden observar todos los elementos mencionados. Con estas interfaces, un ser humano puede obtener la inmersión de realidad virtual de cuerpo completo.

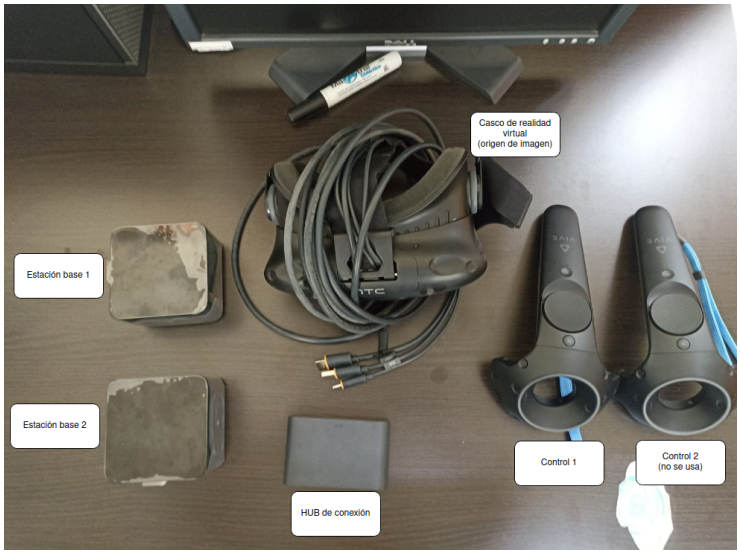


Figura 4.5: Set de realidad virtual

El funcionamiento básico de la inferencia de las posiciones y orientaciones de los componentes del kit de realidad virtual se basa en el método denominado *Lighthouse tracking* o captura de casa luminosa. Esta estrategia fue presentada por primera vez en la conferencia *Hackaday Supercon 2016* por parte de su creador Alan Yates como un método barato computacionalmente, rápido y preciso capaz de capturar el movimiento en 3D enfocado a

[6]Steam, https://store.steampowered.com/app/620980/Beat_Saber/

la realidad virtual. Entonces, acorde a la sesión de la conferencia [46] e información oficial de *SteamVR* [47] el funcionamiento se basa en lo siguiente.

1. El objetivo principal de la tecnología es determinar la posición y orientación de las interfaces, basándose solamente en el tiempo de respuesta que tienen los mismos ante una serie de estímulos. Para lograr esto, primero la estación base lanza un disparo de luz infrarroja que es recibida por los mandos y el casco con la ayuda de múltiples receptores implantados en sus estructuras. Esto inicia un temporizador para cada receptor, el cual mide el tiempo hasta el estímulo del siguiente paso.

2. Inmediatamente después de que el disparo de luz es emitido, la estación base comienza un barrido vertical a lo largo de toda la sala con la ayuda de un láser que forma un plano de luz. Es decir, el barrido se hace de derecha izquierda, pasando por todos los receptores de los componentes, para que así cada uno registre el tiempo en el que el barrido pasa a través de ellos. Cuando el barrido termina, la estación suelta otro disparo de luz para indicarle a todos los componentes que han terminado y reinician el temporizador.

3. Un segundo barrido inicia, pero ahora con orientación horizontal, es decir, de abajo hacia arriba. De la misma manera que en el paso anterior, cada receptor registra el tiempo en el que el barrido lo tocó. Se vuelve a lanzar un disparo de luz para indicar que el barrido ha terminado.

4. Finalmente, cuando ya se tienen los tiempos de respuesta de cada receptor con respecto a los dos barridos, se calcula la orientación y posición de cada uno de los componentes, apoyándose en la triangulación con trigonometría. Hecha la inferencia, el proceso vuelve a iniciar.

El proceso anterior aplica para una sola estación de trabajo, pero cuando se agrega otra, la inferencia mejora y se puede tener un panorama más completo del movimiento del usuario.

En la figura 4.6 se aprecian los barridos horizontales y verticales, así como el disparo de luz infrarroja que hace la estación base.

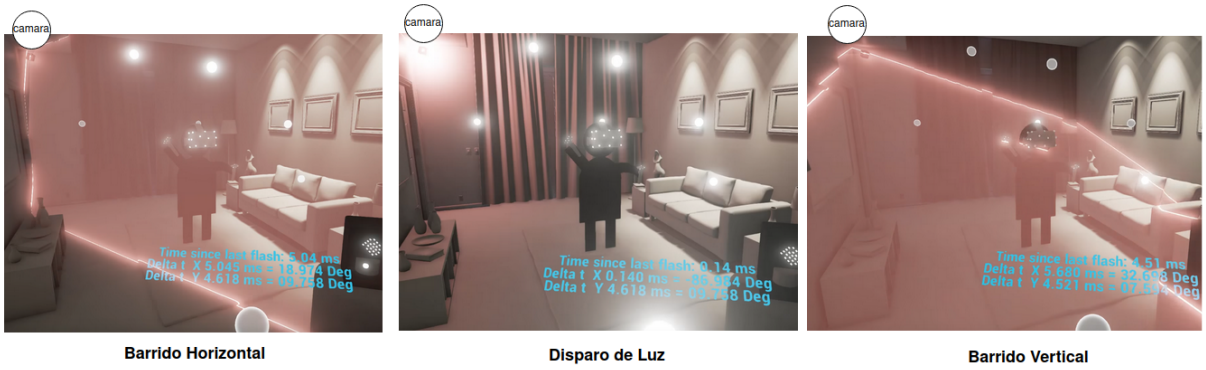


Figura 4.6: Acciones de Lighthouse tracking para estimar posiciones. Adaptadas de: https://www.youtube.com/watch?v=gV1sw4lfwFw

4.3. Definición de destreza humana

Como se muestra en el capítulo anterior, los juegos de realidad virtual han sido objeto de estudio para ver sus efectos en el cuerpo humano. Hablando específicamente de *Beat Saber*, hay investigaciones que apoyan la idea de que el entorno requiere destreza. Por ejemplo, en el trabajo [48] estudiantes de música experimentaron mejoras en su destreza a la hora de tocar sus instrumentos después de sesiones del juego, las cuales ayudaron a desarrollar su coordinación ojo-brazos. Otra investigación que ilustra esta concepción es [49], donde estudiantes de una universidad al jugar experimentaron condiciones de desgaste físicas y mentales similares a las que se tienen cuando se corre en una caminadora.

Entonces, la destreza humana, según el diccionario *Cambridge Dictionary* [50] se define como la *"habilidad de realizar una acción rápida y hábilmente con la manos"*, o como *"la habilidad de pensar rápida y eficientemente o hacer muy bien algo difícil"*.

En el contexto de la robótica, el término ha sido principalmente usado para el uso de manos robóticas para llevar acabo acciones finas [51], sin hacer mayor énfasis en los movimientos del brazo robótico para completar acciones. No obstante, el autor Raymod R. en su trabajo [52] extiende la idea de la destreza al uso de brazos robóticos, ya que determina que a pesar de que es beneficioso el uso de manos robóticas para manipular objetos, el uso de un brazo robótico y un gancho simple como efector final puede realizar muchas tareas que requieren agilidad. Raymod concluye que *"La destreza en general se*

refiere a la variedad de tareas que el sistema puede completar y también a qué tan bien puede realizar esas tareas.", de esta manera se generaliza el término de la destreza en la robótica.

4.4. Ambiente de simulación

4.4.1. Gazebo y ROS Noetic

En [53] se comparó el rendimiento de diferentes herramientas para simular robots bajo el enfoque de aprendizaje por refuerzo, tales como *PyBullet, MuJoCo, Webots y Gazebo*. Como resultado, se demostró que el motor de simulación más eficiente fue Webots; a pesar de ello, el motor seleccionado para este trabajo fue Gazebo. A continuación, se describen las razones de su uso:

- El rendimiento de Gazebo a altas velocidades de simulación mostró ser ineficiente, aproximadamente a un salto de tiempo de $dt \approx 16ms$, no obstante, en este trabajo, debido al poder de cómputo disponible, no se llegará a un salto de tiempo mayor a $dt \approx 5ms$, en el cual Gazebo muestra una estabilidad excelente.
- El simulador dispone de herramientas especiales para conectarse directamente con el middleware ROS, y tomando en cuenta que el agente de aprendizaje por refuerzo profundo será desplegado en el robot MyCobot, que también tiene soporte para ROS, es conveniente desarrollar una solución usando las herramientas que ROS y Gazebo ofrecen.

Como consecuencia de la última razón, se decidió el uso de ROS para permitir la comunicación de Gazebo con cualquier otro programa que ayude a la búsqueda de una solución.

Cabe mencionar que ROS es una biblioteca tipo Middle-ware que permite el paso de mensaje entre programas mediante el uso de *sockets* que están basados en el paradigma publicador-subscriptor, con el fin de comunicar diferentes módulos de robótica. En el modelo publicador-subscriptor, se tiene la concepción de nodos que pueden ser generados

desde cualquier programa. Si un nodo toma el papel de publicador, su objetivo es crear un tópico, el cual es un canal de comunicación donde se envían mensajes de un tipo en específico. Por otro lado, si un nodo toma el papel de subscriptor, este podrá registrarse en algún tópico de un publicador para así poder recibir los mensajes. En la figura 4.7 se muestra un diagrama de este paradigma.

El funcionamiento de todo el medio de comunicación se basa en la existencia de un tercer tipo de nodo denominado como el maestro, que tiene como tarea mantener registro de todos los nodos y tópicos registrados para poder establecer comunicación entre ellos. El nodo maestro es único cada vez que se quiere poner en funcionamiento ROS.

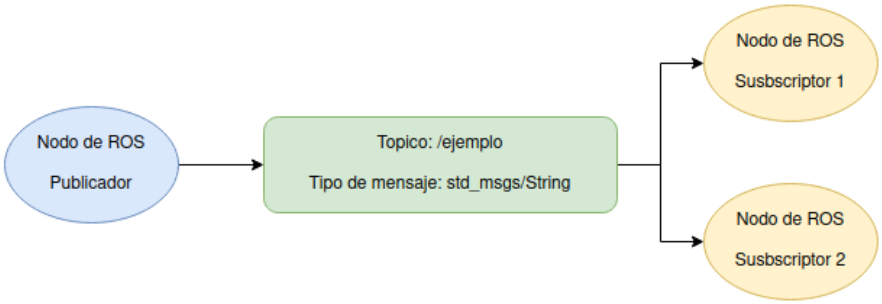


Figura 4.7: Diagrama de paradigma publicador-subscriptor de ROS.

Existen más formas de comunicación dentro de ROS como son los servicios, sin embargo, la publicación de tópicos es la utilizada en el trabajo para transferir información rápidamente, esto a pesar de los problemas de sincronía que puede haber entre los módulos, por ejemplo, el autor Madison menciona en [54] que si el tiempo de respuesta de cada modulo es muy diferente, cada uno estaría procesando versiones diferentes de la información, así creando una inconsistencia en la interpretación del ambiente.

Otra ventaja para usar ROS es la extensa accesibilidad que tiene, para diferentes sistemas operativos y lenguajes de programación. Con respecto a esto último, Python tiene una biblioteca llamada *rospy*, que ofrece una API para comunicarse con las funcionalidades de ROS; entonces, este hecho es conveniente para su incorporación a la API de MyCobot y para los agentes de aprendizaje por refuerzo profundo.

Cabe mencionar que la forma en la que Gazebo interpreta las características de un robot es mediante la conformación de un archivo tipo URDF *(Unified Robotics Description Format)*. En él se especifican todas las características físicas, incluyendo las de los motores.

4.5. Red neuronal artificial: perceptron multi-capa

Como el nombre de aprendizaje por refuerzo profundo sugiere, el objetivo de este es combinar los conceptos planteados por el aprendizaje por refuerzo y las redes neuronales del aprendizaje profundo. Es decir, que una o muchas redes neuronales sean capaces de aproximar una estrategia capaz de resolver un problema.

La red neuronal a utilizar puede ser de muchos tipos, no obstante, la más común es la red neuronal perceptron multicapa. Es la más sencilla de aplicar, ya que simplemente consiste de una capa de entrada, n capas ocultas y una capa de salida; en otras palabras, posee la arquitectura básica de una red neuronal. Por este hecho, en esta sección se brinda una introducción breve a las principales características del perceptron multicapa para poder entender su funcionamiento en el aprendizaje por refuerzo profundo.

Estructura

Desde que se comenzó a estudiar el cerebro para poder dar una explicación del origen de la inteligencia, se notó que las neuronas eran las responsables de dicha capacidad. La interconexión, comunicación y respuesta a estímulos que pose una neurona son factores determinantes que hacen que el cerebro opere de manera lógica. No obstante, al día de hoy no se sabe a ciencia cierta cómo es que estas comunicaciones se llevan acabo para producir la inteligencia, debido a la gran complejidad que lleva analizar un sistema de aproximadamente de 85 a 100 mil millones de neuronas [55]. Lo que sí se conoce es el funcionamiento de cada neurona aisladamente. como efecto de esto, en el año de 1958 en el trabajo [56] el psicólogo Frank Rosenblatt por primera vez acuñó el término de perceptron, el cual es una entidad matemática que modela el funcionamiento de una neurona, ese decir, trata de ejemplificar las señales (estímulos) que llegan a una neurona y cómo ésta las procesa para producir una respuesta adecuada. En la figura 4.8 se aprecian los elementos básicos de un perceptron.

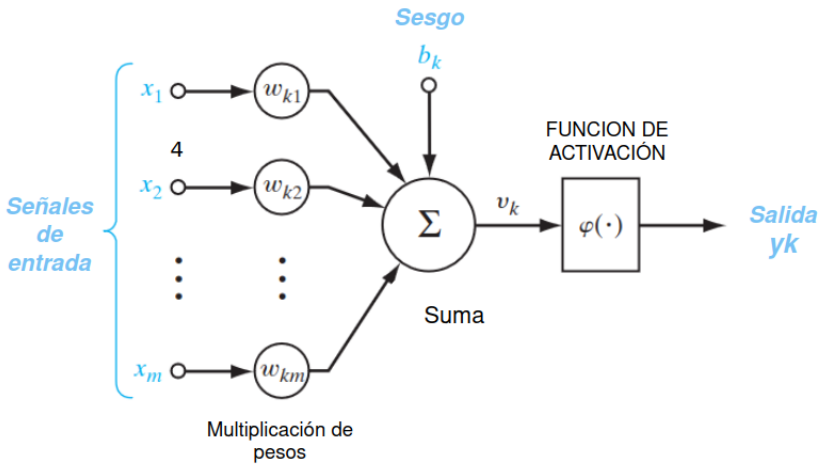


Figura 4.8: Representación de perceptron simple. Adaptada de *Neural Networks and Learning Machines* [57]

El autor Haykin S. en su libro *Neural Networks and Learning Machines* [57] da una explicación detallada del perceptron. En esencia, el modelo describe la suma de varios términos lineales v_k y su mapeo a una función $\phi(v_k)$. Con esto en mente, se tiene que las señales de entrada, o estímulos, están representadas por las variables $(x_1, x_2, ...x_m)$. Los términos w_{km} representan los pesos, los cuales se multiplican por las entradas x_m, tomando en cuenta el respectivo término m; esta multiplicación define la relevancia que tiene cada entrada x_m para el cálculo de la salida. También se considera un término independiente llamado sesgo b_k, donde su principal función es brindarle a la neurona la posibilidad de encontrar patrones en conjunto de datos. La suma de estas multiplicaciones lineales y del sesgo se denomina como v_k. La función de activación $\phi(v_k)$ funciona para agregar no linealidad a la salida y_k de la neurona y para acotar los valores de v_k. Con todo lo anterior descrito, se puede plantear las ecuaciones 4.1 y 4.2 [57].

$$v_k = \sum_{i=0}^{m} w_{ki} x_k \tag{4.1}$$

$$y_k = \phi(v_k + b_k) \tag{4.2}$$

Como ya se mencionó, la función de activación ϕ sirve para mantener la salida de la neurona acotada y agregarle no linealidad. Para ello, se necesita plantear funciones que cumplan con estas tareas. A lo largo del desarrollo de las redes neuronales se han planteado una gran cantidad de funciones, no obstante, las más relevantes son las sigmoide, la tangente hiperbólica y la del Rectificador Lineal Unitario (ReLU)[57]. En la figura 4.9 se puede observar la representación gráfica de estas funciones.

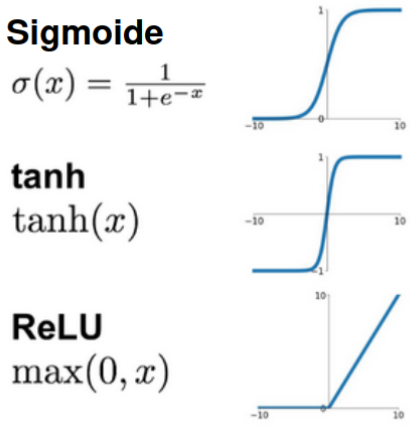


Figura 4.9: Funciones de activación más frecuentes. Tomada de Medium, *Basic Overview of Convolutional Neural Network (CNN)* U. Udofia: https://n9.cl/es/s/f7lwtc.

Con esta arquitectura, Rosenblatt pudo realizar clasificaciones binarias con puertas lógicas con la ayuda de un algoritmo simple para ajustar los pesos $w_k i$. No obstante, esta estructura era muy limitada, ya que no puede resolver una gran mayoría de problemas no lineales, por ejemplo, en la figura 4.10 [57] se muestra un problema de clasificación entre dos clases, en el escenario *a)* es posible separar ambas linealmente, es decir, con un perceptron; no obstante, en el escenario *b)* ya no es posible separarlas linealmente.

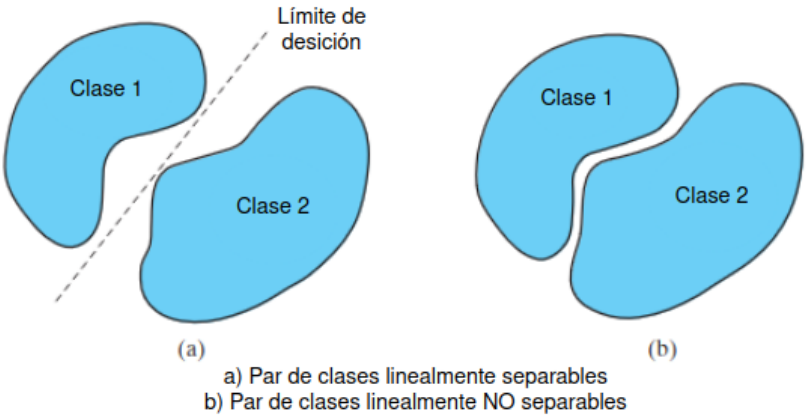


Figura 4.10: Ejemplo de clasificación no lineal. Adaptada de *Neural Networks and Learning Machines.*[57]

Para arreglar lo anterior, en muchos trabajos se comenzaron a usar varias neuronas perceptrones concatenadas para poder resolver problemas más generales y complejos. La investigación que tuvo más relevancia fue [58] en la cual, además de plantear una arquitectura de varias neuronas, brindó un algoritmo llamado retro-propagación para poder ajustar los pesos de todas las neuronas. Entonces, el uso de varias neuronas se conoció como perceptron multicapa, o más recientemente, red neuronal.

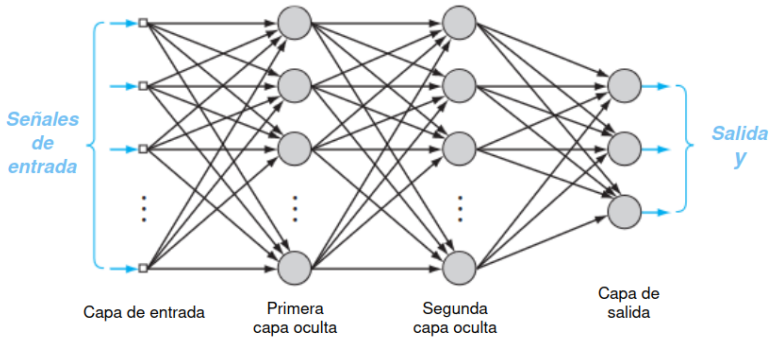


Figura 4.11: Arquitectura de perceptron multi-capa. Adaptada de *Neural Networks and Learning Machines* [57]

De igual manera Haykin [57] da una explicación del perceptron multicapa. En la figura 4.11 se puede observar que el perceptron multicapa consta de arreglos de neuronas denominados capas (l). Hay tres tipos de capas en este modelo: la capa de entrada, capas

ocultas y la capa de salida, cada una con una cantidad determinada de neuronas. Estas capas tienen un orden comenzado por la capa de entrada $l = 0$, siguiendo por las ocultas $l = L - n$ y hasta la capa de salida $l = L$. De este modo, cada neurona de una capa l se conecta con todas las neuronas de la capa inmediata $l + 1$, a excepción de que si la neurona pertenece a la capa de salida; a esto se le llama conexión completa entre capas. Cabe mencionar que cada conexión representa un peso w.

Lo anterior implica que la salida de una neurona en la capa l va a estar influenciada por la salida de todas las neuronas en la capa $l-1$. Por lo que tomando en consideración esto y la conexión máxima, se puede reformular la salida de una neurona $y_j^{(l)}$ como se muestra en la ecuación 4.3 [57] donde el índice l hace referencia a la capa, el índice j indica la neurona dentro de la capa, el índice i hace referencia a los pesos w de la neurona y a las salidas $y^{(l-1)}$ de todas las neuronas de la capa anterior. El término n hace referencia al estado de los pesos w en un tiempo discreto.

$$y_j^{(l)} = \phi_j(\sum_{i=0}^{m} w_{ji}^{(l)}(n) y_i^{(l-1)}(n)) \tag{4.3}$$

De esta manera se obtiene un modelo capaz de aproximar casi cualquier función matemática. Una expresión de esta función general sería la mostrada en la ecuación 4.4.

$$f(X; w) = Y \tag{4.4}$$

Alimentación hacia adelante

El concepto de alimentación hacia adelante es sencillo de entender con lo visto hasta el momento, simplemente es usar la expresión 4.3 por cada capa, comenzando por la primera capa oculta usando al vector X hasta la capa de salida cuando se obtiene a Y.

Función de perdida (loss function)

El principal reto de las redes neuronales es tener datos adecuados para poder ajustar sus parámetros w y obtener una salida Y deseada. Este ajustamiento se llama entrenamiento.

Cuando el modelo no está entrenado, los valores de salida distan de los valores deseados, por lo tanto se puede contabilizar un error e como se muestra en la ecuación 4.5 [57] donde d_j expresa el valor deseado de una neurona en la capa de salida y y_j la salida real de la misma.

$$e_j^{(L)}(n) = d_j(n) - y_j(n) \tag{4.5}$$

Entonces, si se ponderan los errores de todas las neuronas de la capa de salida, se obtiene la función de pérdida ($L(n)$), también llamada función de coste. La definición de la función de pérdida puede ser libre dependiendo del problema a resolver. En el campo de aprendizaje por refuerzo profundo, la definición puede variar dependiendo del algoritmo a utilizar [59, 60, 61]. A pesar de ello, y para ilustrar esta sección, se puede utilizar el error cuadrático medio para ponderar los errores, como se ve en la ecuación 4.6[57].

$$L(n) = \frac{1}{2}\sum_{j=0}^{C}(e_j^{(L)})^2(n) \tag{4.6}$$

Descenso del gradiente y retro-propagación

El objetivo para el entrenamiento es tratar de que la función de pérdida se reduzca, ya que significaría que casi no hay error en la salida. Entonces, el entrenamiento se vuelve en un problema de optimización, donde se busca reducir el valor de $L(n)$ con respecto a todos los parámetros w involucrados en la red. Para ello se plantea el descenso del gradiente que tiene como objetivo modificar los parámetros para reducir el error. Esto se muestra en la ecuación 4.7 [57], en donde $\nabla_w L(n)$ es el gradiente de la función de pérdida con respecto a cada parámetro w y α es la taza de entrenamiento, la cual controla que tanto se modifican los parámetros con cada actualización del descenso del gradiente.

$$w_{ji}^{(l)}(n) = w_{ji}^{(l)}(n-1) + \alpha\nabla_w L(n) \tag{4.7}$$

Entonces, la idea de calcular el gradiente es cuantificar qué tanto tiene que cambiar cada parámetro w para que el error sea 0. Sin embargo, la definición del gradiente fue un reto desde la concepción del perceptrón, pero no fue hasta en el año 1998 que en el trabajo [58] donde se propuso el algoritmo de retro-propagación para calcular los gradientes de todos los parámetros.

Recibe el nombre de retro-propagación ya que los parámetros se van ajustando desde la capa de salida hasta la primera capa oculta, es decir, en dirección contraria a la alimentación hacia adelante. La realización de este algoritmo se basa en el análisis con cálculo vectorial de cómo la salida de una neurona influye en la entrada y salida de otra. Con esto Haykin [57] se plantea la ecuación 4.8 donde el valor de $\delta_j^{(l)}(n)$ depende de qué capa se esté ajustando. Si es la capa de salida, se usa la ecuación 4.9 y si es oculta, se ocupa 4.10.

$$\nabla_w L(n) = \frac{\partial L}{\partial w_{ji}^{(l)}} = \delta_j^{(l)}(n) y_i(n) \tag{4.8}$$

$$\delta_j^{(l)}(n) = e_j^{(L)}(n) \phi_j'(v_j^{(L)}(n)) \tag{4.9}$$

$$\delta_j^{(l)}(n) = \phi_j'(v_j^{(l)}(n)) \sum_{k=0}^{m} \delta_k^{(l+1)}(n) w_{kj}^{(l+1)}(n) \tag{4.10}$$

Optimizador Adam

En este trabajo, se utilizará un algoritmo que mejora el descenso del gradiente. El algoritmo a utilizar se llama ADAM, el cual fue introducido en la investigación [62] como un método para mejorar la velocidad y calidad de aprendizaje, ya que considera cómo va cambiando la función de pérdida a lo largo del tiempo.

Su funcionamiento se basa en el cálculo de los momentos de la función de pérdida, con el fin de obtener una taza de aprendizaje variable. El cálculo de los momentos y la actualización de los parámetros se encuentran expresados en las ecuaciones 4.11, 4.12, 4.13, 4.14 y 4.15. Donde m_n es el primer momento, v_n el segundo momento, $\hat{m}_n$ el primer momento corregido por sesgo, $\hat{v}_n$ el segundo momento corregido por sesgo y β_1 y β_2 son parámetros de sintonización que controlan la decaída exponencial de los momentos.

$$m_n = \beta_1 \cdot m_{n-1} + (1 - \beta_1) \cdot \nabla_w L(n) \tag{4.11}$$

$$v_n = \beta_2 \cdot v_{n-1} + (1 - \beta_2) \cdot \nabla_w L(n)^2 \tag{4.12}$$

$$\hat{m_n} = \frac{m_n}{1 - \beta_1^n} \tag{4.13}$$

$$\hat{v_n} = \frac{v_n}{1 - \beta_2^n} \tag{4.14}$$

$$w_{ji}^{(l)}(n) = w_{ji}^{(l)}(n-1) - \alpha(\frac{\hat{m_n}}{\sqrt{\hat{v_n}} + \epsilon}) \tag{4.15}$$

4.6. Algoritmos de aprendizaje por refuerzo profundo

4.6.1. Aprendizaje por refuerzo profundo

El aprendizaje por refuerzo profundo [1] consiste en que un agente computacional pueda realizar acciones $a \in A$ sobre un ambiente E, ya sea virtual o físico, el cual está definido por estados $s \in S$ que pueden cambiar a medida que el agente ejecuta acciones. La interacción se realiza con el fin de que el agente pueda realizar una tarea en específico; no obstante, este no posee un algoritmo definido que lo guía a la solución deseada, en cambio, toma en cuenta una política π que expresa, en términos de probabilidad, la acción a que se debe realizar dado un estado s para realizar tarea eficientemente. En el contexto de aprendizaje profundo, la política está definida por una red neuronal.

Cabe mencionar que la realización secuencial de acciones que producen estados y recompensas hasta que se completa la tarea se conoce como un episodio o una trayectoria, denominada por τ.

Asimismo, para poder encontrar una política óptima π^*, al agente se le debe otorgar una forma de medir el rendimiento de cada una de sus acciones,o en otras palabras, castigar o recompensar al agente dependiendo de que si sus acciones ayudan a resolver la tarea. A este concepto se le conoce como recompensa R y principalmente está definido por el criterio humano. Todo el proceso descrito anteriormente se puede ver simplificado en la figura 4.12.

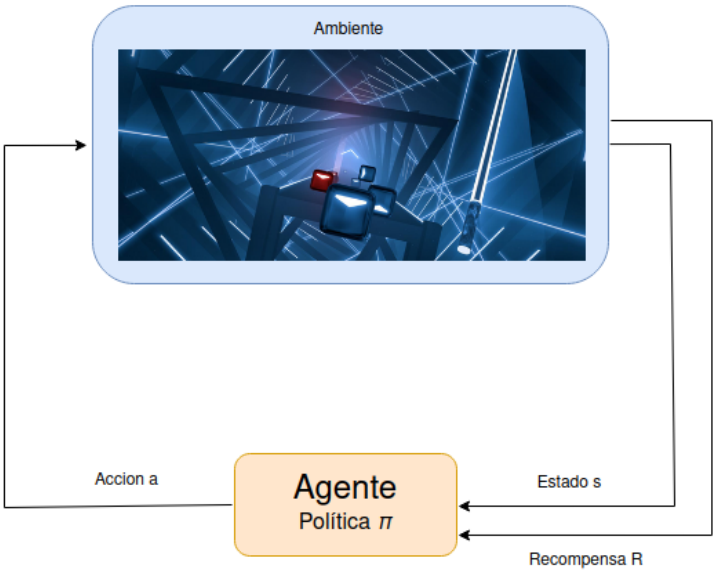


Figura 4.12: Esquema de la dinámica de aprendizaje por refuerzo

De este modo, el objetivo principal del agente es maximizar la obtención de recompensas a medida que ejecuta acciones sobre el ambiente. Esto lo puede realizar si toma en consideración los conceptos de exploración y explotación. La exploración es la capacidad del agente de encontrar nuevas estrategias que le traigan recompensas positivas; generalmente, esto se puede llevar acabo si el agente no siempre selecciona la mejor acción posible que es predicha por el modelo. Por su parte, la explotación es la capacidad del agente para usar las estrategias experimentadas y aprendidas para obtener toda la recompensa positiva que se pueda.

Con todo lo anterior definido, se busca que un agente maximice la ecuación acumulada a lo largo de todos los episodios. Dicha acumulación se presenta en la ecuación 4.16.

$$R_t(\tau) = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \tag{4.16}$$

Donde $R_t(\tau)$ representa la recompensa acumulada con descuento γ obtenida por un agente desde el paso de tiempo t hasta el final del episodio τ. El término γ, acotado en $\{0, 1]$, es el factor de descuento y expresa que las acciones que estén más alejadas del tiempo t, se les darán menor ponderación sobre la recompensa total que el agente llegue a obtener, con el fin de que la recompensa se mantenga acotada y asegurar una buena convergencia

En la línea de investigación de aprendizaje profundo se han desarrollado dos aproximaciones para resolver este problema de maximación: estos son los algoritmos basados en valor y los basados en política [63].

Agentes basados en valor

La aproximación de valor se basa en determinar una política π indirectamente mediante la estimación de posibles valores de recompensa acumulada que se pueden tener en un preciso instante en el tiempo. A la estimación de la recompensa se le conoce como valor.

La definición del valor puede tomar dos sentidos similares. Si se quiere saber, el posible valor que se tiene por el mero hecho de estar en un estado s se considera como v-value y se denomina $V(s)$. En cambio, si se intenta estimar el valor que se conseguirá si se lleva una acción a cuando se está en un estado s se denomina como valor $Q(s, a)$.

Cualquier opción que se decida tomar estará definida por la ecuación de Bellman expresada en la expresión 4.17 y 4.18 [63] para el valor Q(s,a) y V(s), respectivamente.

$$Q(s_t, a_t) = E_\pi[R_t + \gamma \max_{a \in A} Q(s_{t+1}, a_{t+1})] \tag{4.17}$$

$$V(s_t) = E_\pi[R_t + \gamma \max_{a \in A} V(s_{t+1})] \tag{4.18}$$

Donde Q(s,a) define el valor esperado de la recompensa acumulada, si se parte de un estado s y una acción a en el tiempo t, $V(s)$ el valor que tiene un estado s. E_π es el valor esperado de la recompensa siguiendo la política. π, R_t es la recompensa obtenida, y s_{t+1} y a_{t+1} representan el estado y la acción respectivos al tiempo t+1. La expresión max propone que se seleccione el valor Q o V mayor, dado que se supone que se aplican todas las acciones a en el conjunto A.

Por lo tanto, una arquitectura de red neuronal para producir valores Q o V y aproximar una solución a la ecuación de Bellman se muestran en las figuras 4.13 y 4.14 respectivamente.

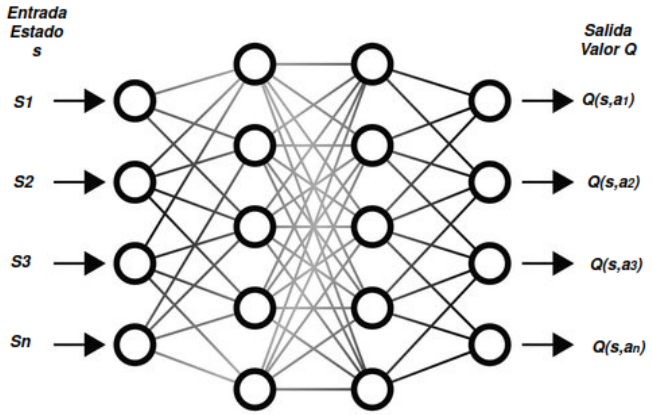


Figura 4.13: Arquitectura de red neuronal para modelos basados en valor Q(s,a)

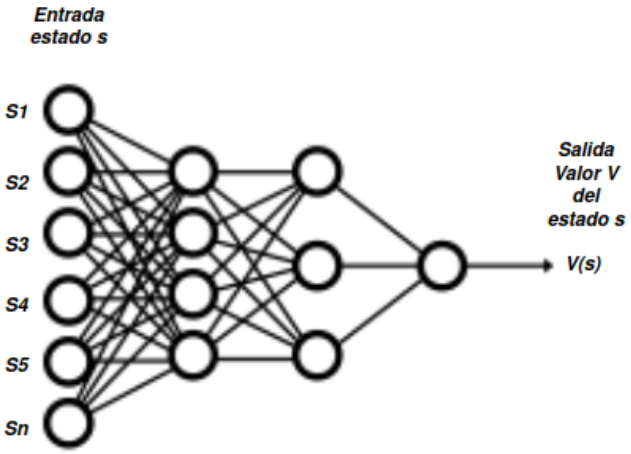


Figura 4.14: Arquitectura de red neuronal para modelos basados en valor V(s)

Cabe destacar que los modelos V(s) y Q(s,a) son deterministas, ya que a una entrada le corresponde un único valor.

Entonces, para poder ajustar los pesos de las redes neuronales, son necesarias funciones de pérdida para poder minimizarlas y así llegar a una buena política. Las expresiones a optimizar se muestran en las ecuaciones 4.19 y 4.20 [64].

$$L(\theta) = E_{\pi}[(r + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}; \theta^{-}) - Q(s, a; \theta))^2] \tag{4.19}$$

$$L(\theta) = E_{\pi}[(r + \gamma \max_{a_{t+1}} V(s_{t+1}; \theta^{-}) - V(s; \theta))^2] \tag{4.20}$$

Los pesos correspondientes a θ es la predicción de la red neuronal entrenada; por otra parte, los pesos θ^{-} corresponden a una copia de la red neuronal original, pero con la diferencia de que la actualización de sus parámetros se hace más lenta. La diferencia de estas dos predicciones se conoce como el error de diferencia temporal, lo que permite poder llevar a cabo un aprendizaje por cada acción realizada.

Agentes basados en política

La principal característica de estos algoritmos es que la política π se optimiza directamente, o sea, la red neuronal ya no toma en cuenta al par (s,a) para hacer una predicción del valor Q o V de cada acción, en cambio, solamente toma como entrada el estado del ambiente y se obtiene como salida una distribución estocástica de las acciones ($\pi(a|s)$). Por lo tanto, una arquitectura de red neuronal para cumplir este mapeo de estados a acciones se muestra en la figura 4.15

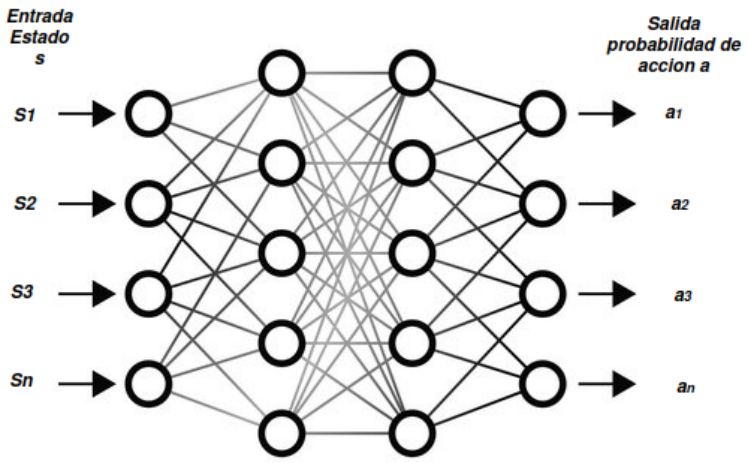


Figura 4.15: Arquitectura de red neuronal para modelos basados en política.

Para poder actualizar los pesos de la red neuronal se utiliza el gradiente de una métrica J que define la recompensa acumulada que puede conseguir un agente basado en los pesos θ propios de la red neuronal. El gradiente de J queda expresado en la ecuación 4.21 [61].

$$\Delta J(\theta) = E[\Delta_\theta log \pi_\theta(a_t|s_t) A_t] \tag{4.21}$$

Donde $\pi(a|s)$ representa la probabilidad de que una acción a_n sea aplicada cuando el estado s se presenta. A_t es un término que representa la ventaja que tiene una acción a en el estado s con respecto a la media de la recompensa de una trayectoria τ.

Paradigma actor-crítico

Los tres modelos mencionados pueden entrenar políticas eficientes, sin embargo, la naturaleza de sus acciones y estados se limitan a ser discretas, disminuyendo el rango de problemas que pueden resolver. Una solución a este problema se presentó en el trabajo [59] donde se presenta una arquitectura híbrida de métodos basados en valor y en política, la cual permite definir acciones continuas y/o estados con dominio infinito ($a \in \mathbb{R}$, $s \in \mathbb{R}$). Con este trabajo se pudo abrir una línea de investigación donde se involucran acciones de naturaleza continua, como lo es el control de velocidad y/o posición de los componentes de un sistema robótico; las aplicaciones de interés en este trabajo se revisan más adelante.

La principal noción de este paradigma es tener a una red neuronal que produzca acciones (actor) y otra red (crítico) que califique mediante valor, ya sea Q o V, la acción realizada por el actor. A continuación, una descripción breve de cada una:

- Red neuronal Actor: Determina la política estocástica $\pi(a|s)$ o determinista $\mu(s)$, es decir, para *pi* se regresa una distribución estocástica de las acciones a realizar y para μ dado un $s \in \mathbb{R}^n$ regresa una única acción $a \in \mathbb{R}^n$. Al ser una red neuronal que no predice un valor esperado de recompensa, puede ser clasificada como basada en política, por lo que su entrenamiento se lleva a cabo con el gradiente de política $\Delta J(\theta)$.

- Red neuronal Crítica: Es la red encargada de evaluar el valor producido por la acción del actor. Este valor puede ser Q(s,a) o V(s), por lo tanto, la salida de la red es un único número. Dicho esto, la optimización de la red se basa en la función de pérdida de la diferencia temporal $\Delta L(\theta)$, tomando en cuenta la acción realizada por el actor.

Una representación de este arreglo de neuronas se ilustra en la figura 4.16. Con esta arquitectura demostró varias ventajas, estas son: posibilidad de trascender al espacio continuo, convergencia más rápida y estable de los modelos, posibilidad de paralelización de agentes y mejora de la exploración de los agentes.

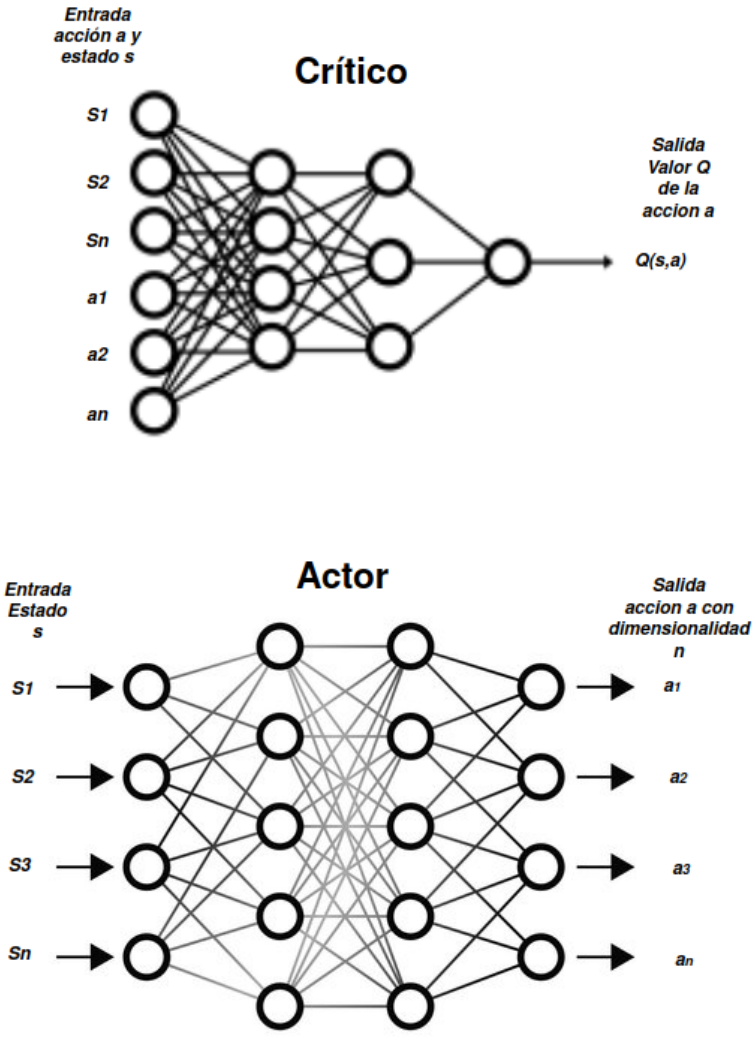


Figura 4.16: Arquitectura de redes neuronales de actor-crítico.

4.6.2. Memoria de repetición de experiencia

Existe otra clasificación de los algoritmos de aprendizaje por refuerzo profundo, que concierne a las experiencias que usa el agente para aprender. Ésta se divide en modelos *on-policy* y *off-policy*. Los modelos *on-policy* tienen como objetivo que el agente entrene únicamente con experiencias generadas por la versión de su política π más reciente. Por otra parte, en las estrategias *off-policy*, el agente puede usar experiencias de versiones pasadas de su política π para aprender. Un método común para lograr el cometido de los modelos *off-policy* es el uso de una memoria de repetición de experiencia.

El concepto fue primeramente desarrollado en [65], donde se propone un búffer R que guardará tuplas de la forma (s, a, r, s'), siendo la acción a, el estado s y la recompensa r pertenecientes al tiempo t, y el estado s' al tiempo $t + 1$. A esta tupla se le conoce como una experiencia.

Su funcionamiento se basa en que cada vez que el agente realiza una acción y se produce un nuevo estado, la tupla se guarda en el búfer R. Cuando el búffer ya tiene una cantidad considerada de experiencias recolectadas y cuando la política se vaya a modificar para aprender, se extrae un subconjunto aleatorio de tamaño N, denominado como lote, y se procede a calcular las funciones de pérdida o de gradientes con un ponderado de todas las muestras. El uso de esta entidad permite que el agente tenga una mayor estabilidad, aprenda más rápido y evita dependencias temporales.

4.6.3. Desempeño de algoritmos en robótica

En el campo de la robótica se han hecho varias investigaciones comparando el rendimiento de algoritmos de aprendizaje profundo por refuerzo en diferentes tareas realizadas principalmente por brazos robóticos [66, 67, 68], las cuales incluyen el control de motores, levantamiento de objetos, posicionamiento de objetos y actividades deportivas de sencilla ejecución. Dadas estas investigaciones, los algoritmos en común que han demostrado mejor desempeño son TRPO, DDPG, TD3 y PPO. En este trabajo se cubrirán solamente los últimos tres mencionados.

4.6.4. Biblioteca de algoritmos de aprendizaje por refuerzo profundo

Para fines de reproducibilidad, es de importancia mencionar la biblioteca de algoritmos de aprendizaje por refuerzo utilizada. La biblioteca designada fue desarrollada en [69] bajo el nombre *stable baselines 3*, su logo puede verse en la figura 4.17. Esta implementa varios algoritmos del estado del arte del aprendizaje por refuerzo profundo, entre ellos, los de interés de este trabajo (DDPG, TD3, PPO y HER). La versión usada es 2.1.0.

Figura 4.17: Logo de *stable-baselines 3*.. Obtenida de https://stable-baselines3.readthedocs.io/en/master/

Se escogió esta opción por las siguientes razones:

- Como esta elaborada sobre la biblioteca de inteligencia artificial *PyTorch*, la cual está en constante actualización, stable-baselines 3 también tiene un alto soporte que lo mantiene eficiente.
- Su uso en Python es mucho más sencillo y compacto en comparación de otras opciones (Ej. Dopamine, OpenIA-baselines, keras-rl). Así, se puede implementar fácilmente con ROS.
- Tiene soporte para diferentes aplicaciones, como lo es *Tensorboard*, la cual se utiliza en el trabajo para tener una estadística de la recompensa de los agentes.

4.6.5. Gradiente Determinista de Políticas Profundas (Deep Deterministic Policy Gradient - DDPG)

El algoritmo DDPG fue presentado en el mismo trabajo en que se presentó la implementación del paradigma actor-crítico [59], por lo que su funcionamiento es una aplicación directa de la arquitectura. No obstante, la característica más relevante de este algoritmo es que la política es determinista $\mu(s)$, es decir, que cada estado s va a producir una acción $a \in \mathbb{R}^n$ que es aplicada directamente al ambiente, de allí su nombre de política determinista. Además, su crítico calcula valores de tipo $Q(s, a)$.

Acorde a la documentación de OpenAI, stable baselines 3 utiliza el algoritmo 1[7] para la implementación de DDPG.

[7] https://spinningup.openai.com/en/latest/algorithms/ddpg.html

Algoritmo 1: Algoritmo de DDPG

Inicializar actor μ_θ y crítico Q_ϕ;
Inicializar las redes objetivo $\theta_{obj} \leftarrow \theta$, $\phi_{obj} \leftarrow \phi$ y un buffer de repetición D; **do**
    Observar s_t y seleccionar una acción $a_t = \mu_\theta$;
    Ejecutar a_t sobre el ambiente;
    Observar el siguiente estado s_t, la recompensa r_t;
    Guardar (s, a, r, s_{t+1}) en el búffer de repetición D;
    si *si* s_{t+1} *es terminal* **entonces**
        reinicia todo el ambiente ;
    si *si se deben hacer actualizaciones* **entonces**
        para *cada actualizacion necesaria* **hacer**
            Aleatoriamente tomar un lote de transiciones, $B = \{(s, a, r, s_{t+1})\}$ desde D;
            Determinar objetivos: $y(r, s_{t+1}) = r_t + \gamma Q_{\phi_{obj}}(s_{t+1}, \mu_{\theta_{obj}}(s_{t+1}))$;
            Actualizar el crítico con el gradiente descendiente con: $\Delta_\phi \frac{1}{|B|} \sum_{(s,a,r,s_{t+1})\in B}(Q_\phi(s, a) - y(r, s_{t+1}))^2$;
            Actualizar actor con gradiente ascendente con: $\Delta_\theta \frac{1}{|B|} \sum_{s\in B} Q_\phi(s, \mu_\theta(s))$
            Actualizar las redes objetivo con:
            $\phi_{obj} \leftarrow \rho\phi_{obj} + (1 - \rho)\phi$
            $\theta_{obj} \leftarrow \rho\theta_{obj} + (1 - \rho)\theta$
        fin
while *Hasta convergencia*;

4.6.6. TD3 (Twin Delayed DDPG)

El algoritmo fue introducido en [60] como una mejora de DDPG, el cual aplica estrategias para evitar problemas inherentes al paradigma actor-crítico, tales como:

- Para resolver el sesgo de la sobreestimación de la recompensa acumulada por parte del crítico, se hace uso de dos redes neuronales críticas, de las cuales se toma como válido el valor Q con menor magnitud para efectuar el aprendizaje, dando como resultado un política más precisa.
- Para aumentar la exploración del algoritmo, se le agrega ruido a las acciones de salida de la neurona actor objetivo θ^-. La estrategia recibe el nombre de suavizado de la política objetivo.
- Con el objetivo de tener un entrenamiento más estable, la actualización de los pesos de la neurona actor se hace con menor frecuencia. A esto se lo denomina como actualizaciones retardadas de política.

4.6.7. HER (Hindsight Experience Replay)

Planteado en el trabajo [2] propone un tipo de búffer alternativo de repetición de experiencia. Su funcionamiento se basa en la existencia de un objetivo g, al cual el agente tiene que llegar, de esta manera, la tupla tiene la forma $(s_t||g, a_t, r_t, s_{t+1}||g)$. Además, existe un término g', que define el objetivo al que pudo llegar el agente, ya sea exitoso o no.

Con esto dicho, el objetivo de HER es crear experiencias falsas, pero cambiando el objetivo original g por el alcanzado por el agente g'. Por esta razón, cada vez que se extraiga un lote para el entrenamiento, se crearán tuplas falsas a partir de las originales pero con la forma $(s_t||g', a_t, r'_t, s_{t+1}||g')$. Nótese que la recompensa en el tiempo t es recalculada con el objetivo g'.

Este proceso trae ventajas como un aprendizaje más rápido en procesos que tenga objetivos concretos y uso de un sistema de recompensa escasa, es decir, que la recompensa solo se proporcione cuando el objetivo es logrado o no.

El búffer es compatible con los algoritmos *off-policy* por lo que en este trabajo, también se medirá su rendimiento usando esta técnica.

4.6.8. PPO (Proximal Policy Optimization)

Propuesto en el trabajo [61], el algoritmo de optimización de política proximal también es un algoritmo basado en el paradigma actor-crítico, pero con las características de que su actor es de política estocástica $\pi(a|s)$ y el valor que calcula su crítico es $V(s)$. El modelo propone la función objetivo expresada en 4.22 y 4.23.

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} \tag{4.22}$$

$$L^{CLIP}(\theta) = E_t[min(r_t(\theta)A_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)A_t)] \tag{4.23}$$

Se tiene en cuenta una política $\pi_{\theta old}$ que es una versión anterior a la política más reciente. De esta manera, el termino $r_t(\theta)$ expresa la probabilidad que hay de tomar la acción predicha por la política nueva con respecto a la antigua, lo que garantiza que los pesos de la política nueva no se alejen demasiado de su versión anterior.

Por otra parte, también se implementa un método de recorte *clip* sobre el término $r_t(\theta)$ que limita su valor entre $1-\epsilon$ y $1+\epsilon$ con el fin de no haya actualizaciones muy grandes en los pesos θ, evitando así caer en inestabilidades.

Estos aspectos brindan ventajas, tales como: un entrenamiento más estable con respecto a otros modelos basado en política, y una implementación más simple, ya que la sintonización de sus parámetros es más sencilla.

Acorde a la documentación de *OpenIA*, *stable baselines 3* utiliza el algoritmo 2[8] para poder implementar PPO.

[8] https://spinningup.openai.com/en/latest/algorithms/ppo.html

Algoritmo 2: Algoritmo de PPO

Inicializar la política π_θ y la función de V-valor V_ϕ;

para *k = 0,1,2,...* **hacer**

  Coleccionar un conjunto de trayectorias $D_k = \tau_i$ resultantes de aplicar la política π;

  Calcular las recompensas R_t;

  Calcular las ventajas $A(s_t, a_t)$ usando la red de valor $V_\phi(s)$;

  Actualizar la política maximizando con gradiente ascendente con: $\theta_{k+1} = arg\,\text{máx}_\theta \frac{1}{|D_k|T} \sum_{\tau \in D_k} \sum_{t=0}^{T} min(r_t(\theta)A(s_t, a_t), clip(r_t(\theta), 1-\epsilon, 1+\epsilon)A(s_t, a_t))$

  Actualizar la función $V_\phi(s)$ maximizando con gradiente descendiente con:

  $\phi_{k+1} = arg\,\text{mín}_\theta \frac{1}{|D_k|T} \sum_{\tau \in D_k} (V_\phi(s_t) - R_t)^2$

fin

4.7. Algoritmos para la visión computacional

En este trabajo se proponen dos estrategias para la visión computacional del robot, una es mediante el uso de redes neuronales convolucionales y la otra en la detección de objetos; por lo tanto, de estas dos se selecciona la que mejor pueda adaptarse al objetivo.

En esta sección se describe el funcionamiento de las redes neuronales convolucionales y la selección de un algoritmo de detección de objetos junto a su funcionamiento.

4.7.1. Red neuronal convolucional

Desde el inicio en la investigación de la visión computacional ha estado el reto del procesamiento de imagen, ya que estas estructuras de datos bidimensionales tienen un gran número de datos. Por ejemplo, una imagen a color de 128 x 128 tiene 49125 datos a analizar, lo cual se hace un reto identificar patrones dentro de la misma. Una de las aproximaciones que ha logrado superar este problema de dimensionalidad es el uso de las redes neuronales convolucionales, también llamadas CNN. Esta técnica de procesado de imagen e identificación de patrones fue presentada por primera vez en el trabajo [70] donde se puso a este método a identificar letras escritas a mano con la arquitectura LetNet-5 mostrada en la figura 4.18. Los resultados presentados sobre el error de clasificación son destacables en comparación a otros algoritmos de extracción de características, como lo poden ser regresiones lineales, análisis PCA, máquinas de soporte vectorial y redes neuronales normales.

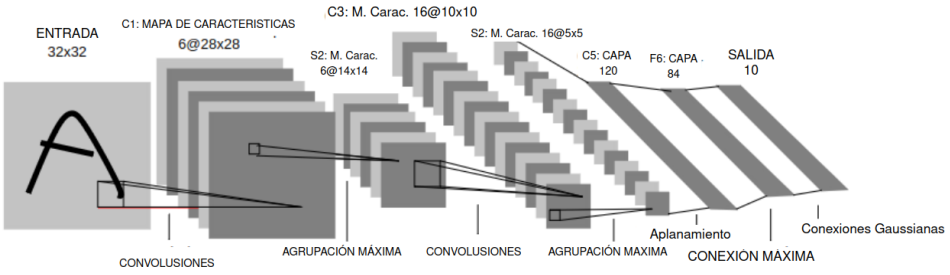


Figura 4.18: Arquitectura CNN de LetNet-5. Adaptada de *Gradient-Based Learning Applied to Document Recognition* [70]

Las redes neuronales convolucionales resuelven el problema de la dimensionalidad, ya que tienen la misión de reducir el tamaño de la imagen, pero sin sacrificar información valiosa que ayude a la identificación de patrones.

El funcionamiento de esta estrategia está influenciado fuertemente por la forma en la que el ojo humano puede identificar patrones en la vida real. La forma en la que se plantea matemáticamente la visión humana es mediante el uso de las convoluciones. Recordando que la operación de convolución se representa por el símbolo $*$ y se expresa como lo indica la ecuación 4.24. Donde x es una imagen de dimensiones (m_x, n_x) y h un filtro con dimensiones (m_h, n_h) las cuales tienen que ser mucho menores a las de la imagen x.

$$x[m,n] * h_[m,n] = \sum_{j=-\infty}^{\infty} \sum_{i=-\infty}^{\infty} x[m,n] \cdot h[m-i, n-j] \tag{4.24}$$

Hay que entender que un filtro h es una matriz cuyos valores están diseñados para resaltar o atenuar alguna característica dentro de la imagen; por ejemplo, resaltar bordes en alguna dirección, difuminar la imagen, detectar figuras geométricas, entre otros. Con esto dicho, la idea de la convolución en 2 dimensiones es que el filtro h vaya recorriendo cada píxel de la imagen x, para que así todos los píxeles que queden dentro del área del filtro h en una posición [m,n] se multipliquen elemento a elemento con los del filtro y se sumen para tener el valor de ese píxel en la imagen modificada. En la figura 4.19 se observa este proceso, donde la matriz roja en la imagen izquierda representa el filtro recorriendo la imagen y el cuadrado rojo en la imagen derecha es el resultado de la suma de la multiplicación elemento a elemento de los píxeles traslapados con el filtro. El efecto de la convolución en este ejemplo es el resaltado de bordes sobre una imagen.

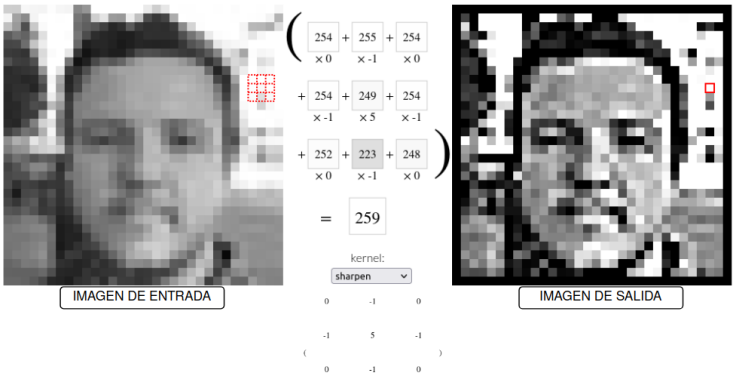


Figura 4.19: Efectos de convolución de una imagen con un filtro. Imagen obtenida de https://setosa.io/ev/image-kernels/

La red neuronal convolucional implementa cuatro tipos de capas que sirven para implementar el proceso antes descrito y hacerlo eficiente en aspectos temporales, computacionales y de calidad de extracción de características. A continuación se brinda una explicación breve de estas capas.

Capas convolucionales

Estas capas son las encargadas de realizar la operación de convolución. Para tener un mapa de características más representativo de la imagen, el proceso de convolución se hace con diferentes filtros. De esta manera, cada capa de convoluciones está representada por un vector de 3 dimensiones (W, H, F) donde W y H representan el ancho y largo de las imágenes resultantes de las convoluciones y F el número de filtros aplicados.

Se puede determinar la dimensión de salida con la ecuación 4.25, considerando una imagen cuadrada con dimensión m_x y un filtro cuadrado con m_h.

$$W, H = \frac{m_x - m_h + 2P}{s} + 1 \tag{4.25}$$

Donde P se refiere al relleno (*padding*) y son los píxeles de contorno que se añaden a la imagen para asegurar que el filtro pase por todos los píxeles. Además, S se refiere al incremento de paso de píxeles del filtro (*stride*).

Agrupación Máxima

La agrupación máxima consiste en reducir el tamaño de una matriz, pero extrayendo la características más relevantes de las convoluciones. Recorre una matriz de la misma manera que un filtro con una imagen, pero en vez de llevar acabo una multiplicación con los elementos traslapados, extrae el elemento con mayor valor para considerarlo como píxel en una nueva imagen/matriz. Esto se hace para cada elemento en una capa (W, H, F) En la figura 4.20 se aprecia el funcionamiento de esta estrategia.

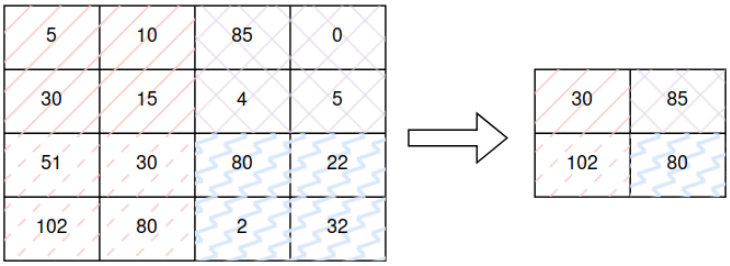


Figura 4.20: Reducción por agrupación Máxima

Cabe mencionar que las capas de agrupación máxima y convolucion se pueden poner en el orden que sea y cuántas veces se requiera, esto depende de la arquitectura deseada.

Aplanamiento

El aplanamiento simplemente consiste en concatenar todas las columnas o filas de todas las matrices presentes en una capa (W, H, F), de tal manera que quede un vector unidimensional con todos los valores del mapa de características. Esto con el fin de que pueda servir como entrada de una red neuronal normal. En la figura 4.21 se puede observar la redimensión para una matriz.

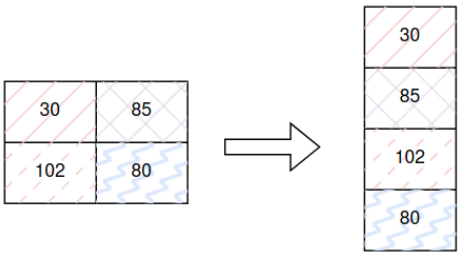


Figura 4.21: Aplanamiento del mapa de características

Conexión completa

La conexión completa solo hace referencia a la conexión entre el vector unidemnsional de mapas de características con una red neuronal convencional para comenzar el proceso de inferencia de la imagen.

Las aplicaciones que han tenido las redes neuronales convolucionales son muy numerosas en prácticamente cualquier campo, ya que permiten obtener información certera de una imagen si se entrena la red neuronal con los datos adecuados. En el ámbito del aprendizaje por refuerzo profundo, es común someter la imagen producida por un ambiente a un procesamiento con capas convolucionales y de agrupación máxima con el fin de obtener un mapa de características que simplifique el estado del ambiente, para que así el agente de inteligencia artificial pueda considerarlo y determine qué acción realizar.

Por lo tanto, en este trabajo se analizará si es viable seguir dicha estrategia en contraste con la detección de objetos.

4.7.2. Algoritmos de detección de objetos y su desempeño

La detección de objetos consiste en encontrar entidades dentro de una imagen para ser localizadas y clasificadas. La localización del objeto se puede llevar a cabo de distintas maneras, no obstante, la estrategia más común es mediante el uso de cajas delimitadoras que encierran al objeto a clasificar.

En esta línea de investigación se han desarrollado una gran variedad de algoritmos, principalmente apoyados en las redes neuronales convolucionales, ya que han demostrado ser excelentes para detección de características. Algunos ejemplos de estos algoritmos son Fast R-CNN (Redes neuronales convolucionales basadas en regiones rápidas), SSD (Detector multi caja de disparo único), ResNet (Redes residuales) y YOLO (Solo miras una vez).

Con la aparición de estos algoritmos surgieron investigaciones para determinar el desempeño de cada uno para encontrar y clasificar objetos [71, 72, 73]. Se usaron las métricas como FPS (Frames per Second), mAP (Mean Average precision) y IoU (Intersection Over Union) para poder definir la eficiencia de cada uno.

La métrica de IoU hace referencia a la precisión del modelo de definir una caja que defina perfectamente al objeto. En la figura 4.22 se puede ver una representación de esta medición. Entonces, teniendo en cuenta a la caja objetivo (roja) y la caja pronosticada (azul), el cálculo IoU se lleva acabo dividendo la intersección de estas dos cajas entre la unión de las mismas.

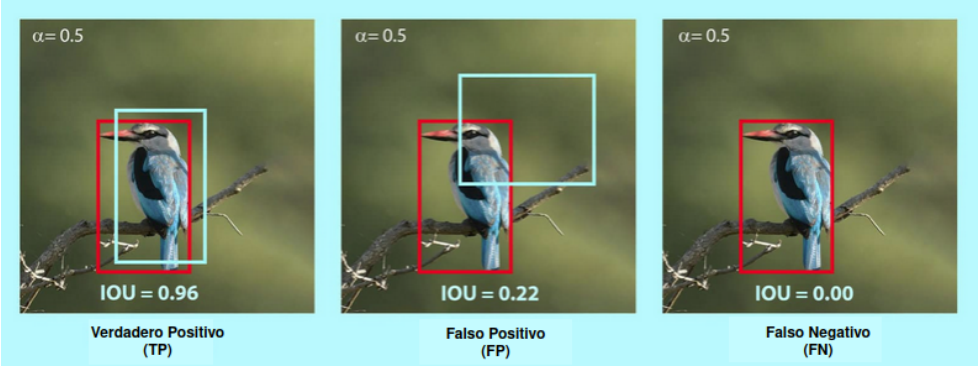


Figura 4.22: Métrica Intersection over Union para clasificación. Adaptada de *Learn OpenCV* https://learnopencv.com

Con esta métrica y tomando en cuenta un valor umbral α para clasificar las predicciones como verdaderos positivos (TP), falsos positivos (FP) y falsos negativos (FN), se puede elaborar métricas de precisión y *recall*. Recordando que la precisión se define como la cantidad de verdaderos positivos clasificados como correctos $TP/(TP + FP)$ y el recall es la relación entre las predicciones positivas verdaderas y el número total de positivos reales $TP/(TP + FN)$.

La forma en la que se calculan estas dos métricas es iterativa, es decir, que se van actualizando con cada imagen que se procesa. Por lo tanto, se puede hacer una curva de precisión-recall que muestra la evolución de los valores de estos criterios conforme se van clasificando imágenes de un conjunto de datos. De esta manera, si se obtiene el área bajo la curva de dicho proceso, se define la métrica AP (Avarage Precision) y para obtener mAP se hace un promedio tomando en cuenta los valores AP de todas las clases a clasificar.

Entonces, considerando los resultados de métricas de las investigaciones mencionadas y el requisito en este trabajo de tener una detección de objetos lo más rápido posible sin importar mucho la precisión, se seleccionó al algoritmo YOLO ya que tiene los mejores resultados en la velocidad de procesamiento y métricas mAP y IoU decentes en una gran variedad de conjunto de datos.

4.7.3. Modelo YOLO (You Only Look Once)

El algoritmo YOLO fue acuñado en el trabajo [74], y fue presentado como una solución para lograr la detección de múltiples objetos en tiempo real, ya que pudo lograr valores mAP comparables a los algoritmos del momento, pero haciéndolo a una velocidad mucho mayor.

El funcionamiento del paradigma YOLO para hacer una predicción lo conforman principalmente tres etapas:

1. Re-escalamiento: La imagen es redimensionada a medidas específicas para que sea compatible con la entrada de la red neuronal convolucional que especifica YOLO.

2. Alimentación de la red neuronal convolucional: Una vez con las medidas requeridas, la imagen es alimentada a la red neuronal convolucional para formar un vector de características que luego es alimentada a una red neuronal normal. En la figura 4.23 se puede observar la arquitectura de la red neuronal de YOLO.

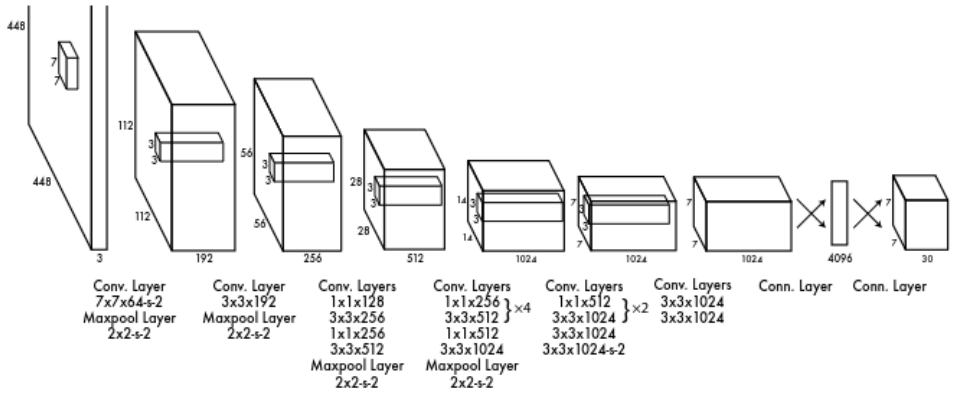


Figura 4.23: Arquitectura de red neuronal convolucional de YOLO. Adaptada de *You Only Look Once: Unified, Real-Time Object Detection* [74]

Para entender la salida de la red neuronal, se debe tener la concepción de que la imagen está dividida en una dimensión S x S formando así un mapa de cuadrados,. Cada cuadro del mapa se le asigna una cantidad B de cajas delimitadoras que tratarán de encasillar al objeto y un vector de probabilidad p_c que indica las probabilidades de que un objeto pertenezca a alguna clase C. Cada caja B está representada por su centro (x,y) su ancho y largo (w,h) y un indicador de confianza C_o que especifica que tan probable es que esté en esa caja B el objeto clasificado en p_c. Por lo tanto, teniendo en cuenta todo lo anterior, la salida de la red neuronal sería un vector de dimensiones $(S, S, B * 5 + C)$. Siguiendo esta idea y considerando valores propuestos en el mismo artículo de YOLO [74] (B = 2, C = 20 y S = 7), se tendría un vector como el mostrado en la expresión 4.26 para cada cuadrado del mapa. Entonces, la tarea de la red neuronal de YOLO es aproximar esos valores para detectar el objeto.

$$[p_1, p_2, p_3, ..., p_{20}, x_1, y_1, w_1, h_1, C_1, x_2, y_2, w_2, h_2, C_2] \tag{4.26}$$

3. Selección de cajas delimitadoras: Finalmente, se seleccionan las cajas que tengan un nivel de confianza mayor a un umbral y son las mostradas como resultado final. Esto solo es válido en la predicción, en el entrenamiento se debe utilizar la supresión no máxima para poder determinar una caja B óptima para ajustar la red neuronal.

Estos tres procesos se pueden ver simplificados en la figura 4.24

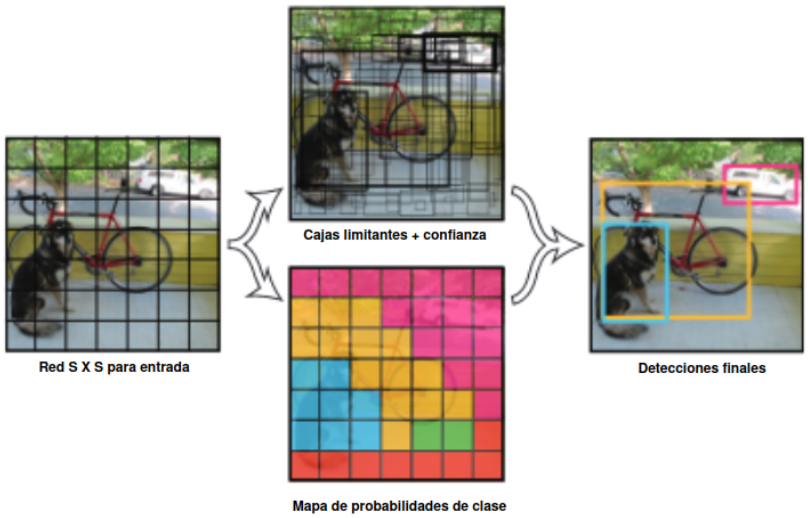


Figura 4.24: Diagrama de paradigma YOLO. Adaptada de *You Only Look Once: Unified, Real-Time Object Detection* [74]

YOLO posee varias versiones que pueden variar en varios aspectos, sobre todo en la estructura de la red neuronal convolucional, para poder mejorar el rendimiento mAP del modelo. De esta manera, la versión del algoritmo que mejor ha mostrado resultados es YOLO v7. Esto se muestra en el trabajo [75] del 2022, donde se presume que el algoritmo fue capaz de obtener un AP del 56 % a una velocidad de que va desde 5 FPS hasta 150 FPS, cifras que sobrepasan a versiones anteriores de YOLO o a otros algoritmos, debido a que estos llegaban con dificultad a un valor aproximado de AP del 53 % a solo 10 FPS.

YOLO V7 tiene diferentes distribuciones que se adaptan a diversas necesidades. En este trabajo se considera el modelo Yolov7-tiny[9], ya que maneja un número de parámetros menor (6.2 M) haciendo que sea un modelo más rápido.

4.8. Control jerárquico de un robot

Acorde a [76] el control de un robot tiene como objetivo bríndale una cinemática apropiada al robot para que pueda realizar tareas de manera precisa e inteligente. Para lograr esto, se desarrolla algún modelo matemático que indica cómo deben comportarse

[9] https://github.com/WongKinYiu/yolov7/tree/main

los componentes del robot a lo largo del tiempo. En el caso de los motores de un brazo robótico, un modelo de control puede indicar qué tan rápido y hacia qué sentido debe rotar para poder alcanzar una posición objetivo, o el voltaje necesario para que un motor pueda alcanzar cierta velocidad.

Siguiendo el ejemplo de los motores, un modelo de control está enfocado a reducir el error θ_e que hay entre alguna posición θ_d objetivo y la posición real del motor θ para así poder tener la cinemática adecuada.

Dentro de la teoría de control existe el control jerárquico de un sistema que consiste en la delegación de funciones en diferentes niveles para que el robot pueda moverse de forma adecuada y pueda cumplir con la tarea solicitada. En el artículo [77] se presenta una arquitectura muy común de este tipo de control, la cual consta de tres niveles:

- **Control de alto nivel**: Define la parte inteligente y lógica del movimiento del robot, es decir, elabora un plan de movimiento que cumple con la tarea deseada. Para cumplir lo anterior, al nivel se le debe otorgar información necesaria para tener una noción del estado del robot y poder determinar la acción a realizar.

- **Control de medio nivel**: Toma en consideración el plan realizado por el alto nivel para crear comandos básicos que pueda ejecutar algún componente del robot. Por ejemplo, en el caso de un brazo robótico, a qué velocidad o posición se tiene que mover cada articulación para realizar la tarea.

- **Control de bajo nivel**: Se caracteriza por brindar un movimiento físico estable a cada parte del robot, tomando en cuenta el comando brindado por el control de nivel medio. Para esto, se debe contar con un modelo físico que describa por completo la cinemática de cada una de sus partes.

Este tipo de arquitectura ha sido empleado principalmente en aplicaciones donde un agente de aprendizaje por refuerzo profundo es el responsable de darle una lógica a cada componente de un robot para que se mueva eficientemente; por ello, es natural que el agente se ocupe del control de alto nivel. Ejemplos de esta noción pueden encontrarse en trabajos como la navegación de un robot salamandra [78] o de un robot bipedal [79].

Una buena solución para el control de bajo nivel en el modelo jerárquico es el conocido control PID. Un ejemplo de su uso está en el trabajo del robot bipedal [79].

4.8.1. Control PID

El control PID (Proporcional - Integral - Diferencial) tiene como objetivo mover los componentes del robot de tal manera que reduce la diferencia entre el estado que tienen en un determinado tiempo con respecto a uno deseado, pero siempre manteniendo una noción de los cambios de velocidad o posición que debe tener una entidad real para alcanzar una meta. Este tipo de control es retroalimentado, es decir, que toma en cuenta salidas anteriores para determinar una salida futura que es necesaria para cumplir con la meta.

En el contexto de motores o actuadores, el control PID puede ser aplicado a la posición de giro θ que se tiene en un tiempo determinado. Por lo tanto, el error entre una posición objetivo θ_d es expresado en la ecuación 4.27.

$$\theta_e(t) = \theta_d(t) - \theta(t) \tag{4.27}$$

Entonces, la expresión general que ofrece el control PID es la dada en la ecuación 4.28 [76]. Asimismo, su diagrama de control queda ilustrado en la figura 4.25.

$$\tau = K_p\theta_e + K_i \int_0^t \theta_e(t)\,dt \; + K_d\dot{\theta}_e \tag{4.28}$$

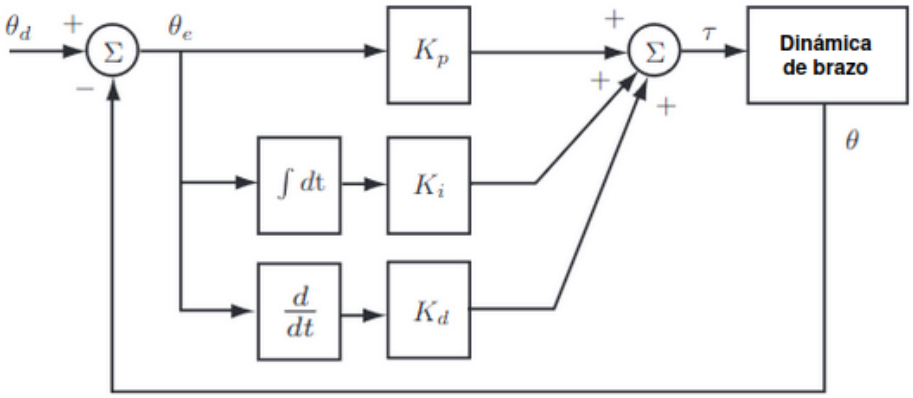


Figura 4.25: Diagrama de control PID. Adaptada de [76]

La salida de la ecuación 4.28 es un torque τ, el cual es el que tiene que ser aplicado sobre el robot para lograr que el motor se aproxime a la posición deseada θ_d. Teniendo en cuenta lo anterior, en la misma ecuación hay tres términos que ayudan al cálculo de la aproximación. El primero se denomina como proporcional (P) y es directamente proporcional al error de la medición, dando como efecto a que el movimiento sea mayor si el error es grande. El segundo término es el integrativo (I) que toma en cuenta el error acumulado de tiempo pasados ocasionando una reducción en los errores estacionarios, es decir, si la señal de salida queda en un valor constante cercano al valor deseado, la parte integrativa tratará de reducir esa diferencia. Por último, el término derivativo (D) se encarga de calcular la tasa de cambio que tiene el error con respecto al tiempo, ocasionando que si la tasa de cambio es grande, el sistema tienda a detenerse.

De esta manera, los parámetros K_p, K_i y K_d son críticos para determinar el comportamiento del sistema. ya que establecen la influencia que tiene cada uno de los tres términos en el sistema. Se deben sintonizar de tal manera que representen con fidelidad la dinámica de un componente; por ello, encontrar valores adecuados no es trivial, ya que pequeñas variaciones en estos pueden producir salidas diferentes. En la figura 4.26 se pueden observar algunas posibles salidas con la variación de los tres parámetros.

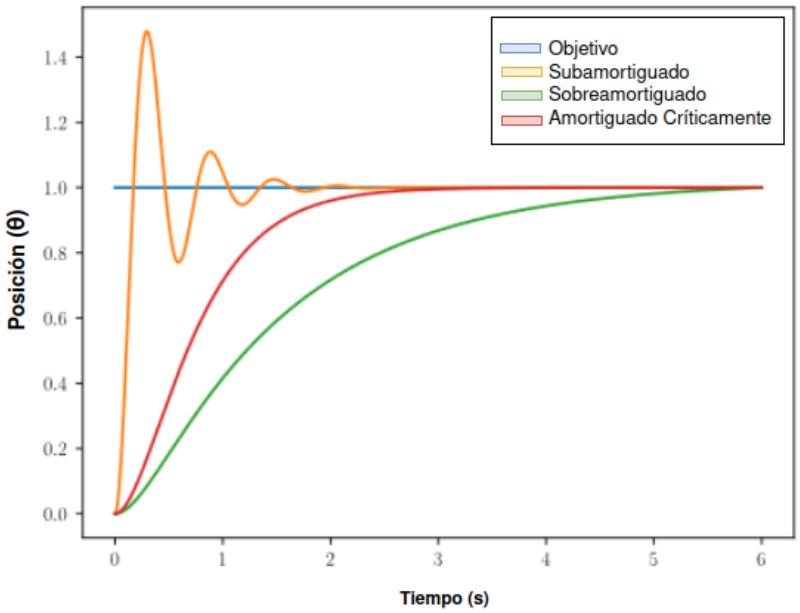


Figura 4.26: Posibles respuestas estables con variación de parámetros PID. Adaptada de https://docs.wpilib.org/es/latest/docs/software/advanced-controls/introduction/introduction-to-pid.html

También se pueden obtener salidas inestables que tienden a oscilar infinitamente o nunca llegar al objetivo. En este trabajo se siguió una estrategia de prueba y error para determinar valores que se ajusten a todas las condiciones físicas propuestas del brazo robótico.

4.9. Consideraciones para transferencia de simulación a entorno real

En el artículo [80] se mencionan algunos métodos que se han desarrollado para transferir políticas aprendidas en entornos simulados a un robot en el mundo físico, teniendo un enfoque sobre el aprendizaje por refuerzo profundo. A continuación se explica brevemente cada uno de estos métodos:

- **Identificación de sistema**: Esta técnica se basa en realizar una simulación lo más cercana a la realidad, es decir, poseer mecanismos que modelen con precisión las dinámicas del robot y características del entorno. Sin embargo, realizar dichos modelos pueden ser demandantes computacionalmente y no ser viables para el aprendizaje del agente.

- **Aleatoriedad de dominio**: Parecido a la transferencia de identificación del sistema, tiene como propósito tener una representación más fidedigna de la realidad, pero introduciendo factores de ruido y variación en todos los factores involucrados en la simulación. Por ejemplo, incertidumbre en la lectura de sensores, variación en las fuerzas de actuadores, diferentes tipos de iluminación, cambios en las características físicas del ambiente, entre otros. Con la variación de condiciones del entorno, se logra tener un agente más generalizado y capaz de adaptarse a cambios.

- **Adaptación de Dominio**: En esta aproximación se busca unificar los espacios de características tanto de la simulación y del mundo real. Esto se logra primero entrenando al agente con todas las ventajas que ofrece un entorno simulado, para después realizar un entrenamiento de afinamiento sobre el robot real para que pueda acostumbrarse a las perturbaciones e incertidumbres que se pueden presentar.

En el mismo trabajo mencionado [80] se utiliza la estrategia de identificación del sistema para tener una representación del mundo real, es decir, se elabora un modelo que describa el movimiento del robot sin considerar perturbaciones mayores. Con esto en mente la simulación realizada en Gazebo para el brazo robótico debe considerar los siguientes aspectos:

- **Dinámicas inerciales y de control**: Consiste en determinar las características físicas y dinámicas del robot para tener una representación real. Esto incluye tomar en cuenta factores como medidas del robot, características de los motores, pesos de cada componente para calcular inercias y coeficientes de fricción. Además de brindar un control que se adapte de manera estable a todas las características antes mencionadas.

- **Métodos de seguridad**: Cuando se trabaja con entornos simulados, el desgaste de los componentes a medida que el robot de mueve casi no se toma en cuenta. Por ejemplo, un agente de aprendizaje por refuerzo puede realizar una acción que requiera un esfuerzo considerable en los motores, haciendo que a la larga los motores vayan perdiendo potencia o den lecturas sesgadas de posición; por ello, es relevante considerar limitaciones de movimiento para evitar movimientos que pueden ser dañinos para el sistema en un entorno real. Además, poner límites de movimiento también ayuda a evitar que haya autocolisiones en el brazo robótico.

- **Aceleración de simulación**: Como muchos de los componentes físicos simulados se basan en el paso de tiempo dt, es fundamental que a la hora de aumentar el paso tiempo para acelerar la simulación se verifique que todas las entidades presenten las mismas condiciones que tienen en el tiempo real, ya que cabe la posibilidad de que muestren inestabilidad y se tenga que hacer un reajuste de parámetros.

- **Tiempo de respuesta de componentes**: En una simulación, el tiempo con el que los componentes pueden procesar comandos puede ser muy corto, en la resolución de fracciones de milisegundos. No obstante, esta respuesta no puede ser tan rápida en el mundo real, lo que puede ocasionar inestabilidad o bajo desempeño del modelo. Para contrarrestar este efecto, en la simulación se deben considerar tiempos de respuesta cercanos a la realidad, limitando los comandos enviados por segundo. En el contexto de aprendizaje por refuerzo profundo, también se debe limitar la frecuencia con la que el agente envía comandos al robot para que éste tenga tiempo de procesar la mayoría de ellos.

Capítulo 5

Metodología

5.1. Modelo simulado de MyCobot 280jn

5.1.1. Estructura y propiedades físicas

El robot MyCobot tiene soporte en ROS, como consecuencia, el modelo del robot se puede conseguir fácilmente en el repositorio GitHub de Elephant Robotics. No obstante, la versión oficial no cuenta con una configuración de las variables físicas del robot, por ello se utilizó una variación del modelo presente en otro repositorio [1] que implementa propiedades físicas como la inercia y pesos de cada parte. En la figura 5.1 se puede ver el robot simulado en Gazebo y visto en rviz.

Es de importancia revisar las características físicas en la tabla 5.1, ya que son las que ponen las condiciones inerciales de cada pieza involucrada, para que así se pueda sintonizar un control adecuado para el robot. Todas las características están expresadas en un documento URDF que usa Gazebo para simular el robot; estas consisten en:

- **Variables inerciales**: Las partes del robot se representan como inercias de tipo cilíndrico, por lo que es necesario especificar la masa, radio y altura. Para cada cilindro se usa la matriz inercial mostrada en la ecuación 5.1.

[1] Tiryoh, https://github.com/Tiryoh/mycobot_ros

$$I_{cilindro} = \begin{bmatrix} \frac{m*(3r^2+h^2)}{12} & 0 & 0 \\ 0 & \frac{m*(3r^2+h^2)}{12} & 0 \\ 0 & 0 & \frac{mr^2}{2} \end{bmatrix} \tag{5.1}$$

- **Características de los motores**: Se incluyen variantes físicas para la caracterización de los motores. Como en el robot real, todos los grados de libertad se manejan con el mismo tipo de motores, todos comparten las mismas características. La constante de torque determina la capacidad de carga de cada motor. La constante de amortiguamiento se refiere a la pérdida de energía a lo largo de un sistema oscilante y el factor de fricción es la resistencia al movimiento del motor.

- **Características del control**: El control HTC Vive está representado por un prisma rectangular que es relativo a la posición del efector final. Por ello, la matriz que modela su inercia está representada en la ecuación 5.2.

$$I_{prisma} = \begin{bmatrix} \frac{m*(y^2+z^2)}{12} & 0 & 0 \\ 0 & \frac{m*(z^2+x^2)}{12} & 0 \\ 0 & 0 & \frac{m*(x^2+y^2)}{12} \end{bmatrix} \tag{5.2}$$

Teniendo en cuenta los valores originales propuestos en el repositorio del modelo, se modificaron ligeramente para que pudieran tener una mejor consistencia con el control PID que se desarrolla más adelante.

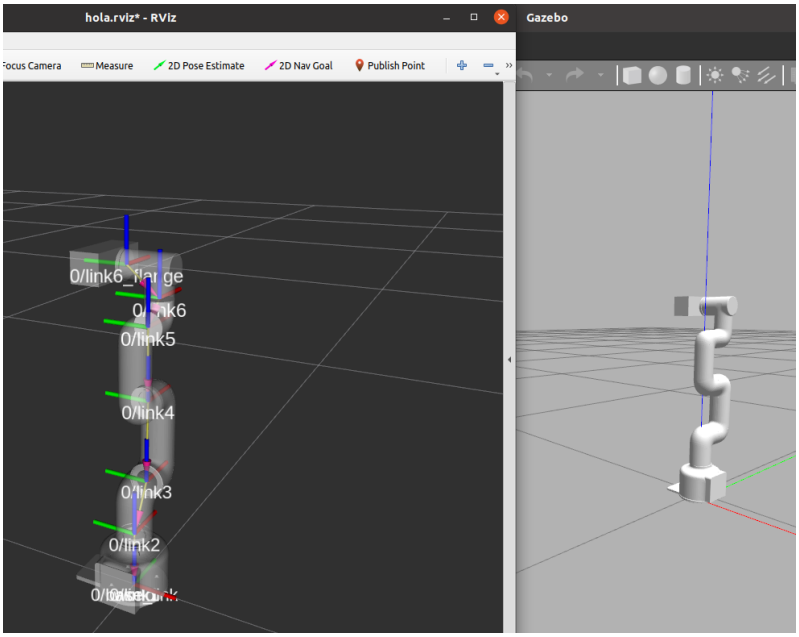


Figura 5.1: Simulación de robot en Gazebo (derecha) y en rviz (izquierda).

Tabla 5.1: Características virtuales-físicas de mycobot

No. Motor	Masa (Kg)	Radio (m)	Altura (m)	Constante de torque (t)	Coeficiente Amort. (Ns/m)	Coeficiente de fricción (N*m)
1	1.0	0.05	0.07	0.418	1.0e-6	2.0e-1
2	0.2	0.05	0.08	0.418	1.0e-6	2.0e-1
3	0.4	0.05	0.155	0.418	1.0e-6	2.0e-1
4	0.4	0.05	0.155	0.418	1.0e-6	2.0e-1
5	0.2	0.05	0.05	0.418	1.0e-6	2.0e-1
6	0.1	0.05	0.06	0.418	1.0e-6	2.0e-1
Control HTC Vive						
	Masa (Kg)	Ancho y (m)	Alto z (m)	Largo x (m)		
	0.22	0.11	0.08	0.21		

5.1.2. Control de robot

Para este trabajo se modeló una arquitectura de control que permitiera mover al robot aproximadamente a 20 Hz, es decir, que el agente pueda mandar 20 comandos cada segundo, esto con el fin de tener un movimiento ágil y rápido del robot. Para cumplir este requisito, se idealizó un control jerárquico que posee cuatro entidades de control que condicionan el movimiento del robot, los cuales se encuentran estructurados en tres niveles que se muestran en la figura 5.2. A continuación se ofrece una explicación del funcionamiento de cada nivel.

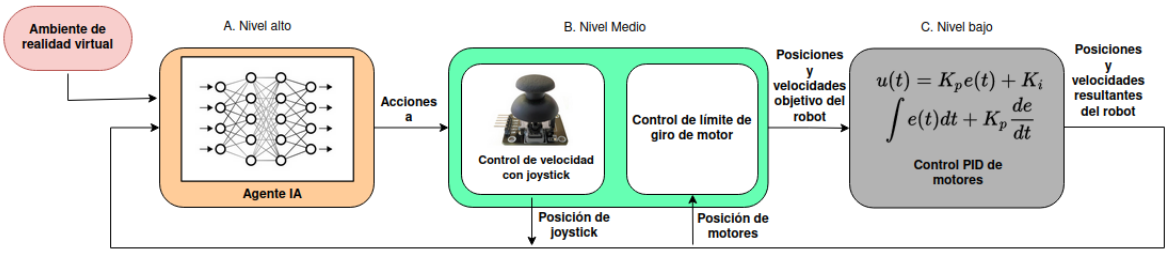


Figura 5.2: Diagrama de control jerárquico para Mycobot280 jn.

A *Agente IA (Nivel alto)*

Tiene como tarea recabar información del robot y del ambiente virtual para producir acciones que permitan mover un joystick virtual que especifica la velocidad angular de un motor del robot; más adelante se explica la entidad del joystick. Cabe mencionar que las acciones son: mover el joystick a la izquierda cierta cantidad (-1), moverlo a la derecha (1) o no moverlo (0). Las características de este módulo son revisadas a profundidad en la sección 5.3.3.

B *Control de joystick y límites (Nivel Medio)*

Este nivel lo constituyen dos módulos que tienen el objetivo de formar una tarea básica para los motores del robot, la cual está basada en la petición del agente; es decir, proporcionar las posiciones y velocidades que tienen que alcanzar los motores en un determinado tiempo para cumplir las acciones requeridas por el control de nivel alto.

Teniendo en cuenta que el agente va a interactuar a una frecuencia relativamente alta con el robot, se necesita de una alternativa que regule los cambios drásticos de velocidad que pueden ser solicitados por el mismo agente. Por ello, en el primer módulo se ideó un mecanismo tipo joystick que especifica la velocidad deseada del motor.

El funcionamiento de este consiste en que, cuando el agente mueva el joystick a la izquierda o a la derecha, el joystick se moverá a la dirección deseada una pequeña cantidad, y dependiendo de la posición en la que esté, es como se determina la velocidad deseada del motor.

El mapeo entre la posición del joystick y la velocidad deseada se calcula con la ecuación 5.3 que expresa una función sigmoidal que está acotada por v_{max} cuyo valor es $\frac{7}{9}\pi$ ya que es un poco menos de la velocidad máxima de cada motor presente en las especificaciones del robot.

$$v(x) = v_{max}(1 - \frac{2}{1 + e^{-5x}}) \tag{5.3}$$

Adicionalmente, la función no lineal 5.3 se discretizó con el fin de no tener cambios de velocidad tan variantes. De esta manera, la función que especifica la velocidad con respecto a la posición del joystick se puede ver en la figura 5.3.

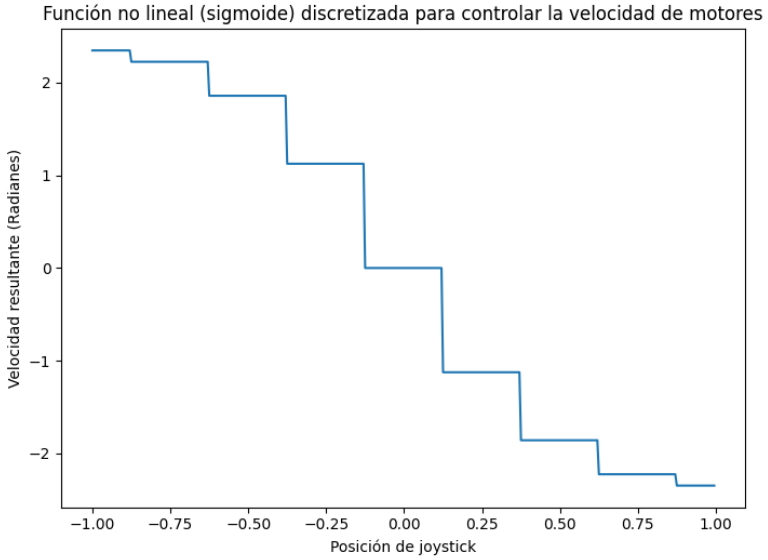


Figura 5.3: Función no lineal (sigmoide) para control de velocidad de motores.

El segundo módulo dentro del nivel tiene la tarea de limitar el alcance de los giros de cada motor para evitar que haya colisiones con el mismo brazo robótico. Los límites propuestos se encuentran expresados en la tabla 5.2.

Tabla 5.2: Limites de giro de cada motor

No. Motor	Límite inferior Radianes	Límite superior Radianes
Motor 1	-0.8	1.27
Motor 2	-0.35	0.25
Motor 3	-0.3	0.3
Motor 4	-1.1	1.3
Motor 5	-1.4	0.2
Motor 6	-2.0	1.6

Asimismo, establecer límites adecuados para el movimiento del robot es de gran ayuda en el entrenamiento, debido a que se reduce el espacio de trabajo y el agente tiene que explorar menos para encontrar una política adecuada.

Sobre el mecanismo de límite de giro, este se basa en la posición deseada de cada motor que es calculada a partir de la velocidad brindada por el módulo del joystick. Si la posición del motor se encuentra dentro de los rangos, la velocidad del motor es normal; por otro lado, si la posición sale del límite, comienza una reducción de la velocidad hasta cero, dando como resultado el frenado del motor y no aceptando velocidades provenientes del joystick en la misma dirección. Una vez que la velocidad cambia a la dirección contraria, se sale de la zona de frenado, haciendo que el motor se mueva con normalidad. Este mecanismo se muestra en el diagrama de estados en la figura 5.4.

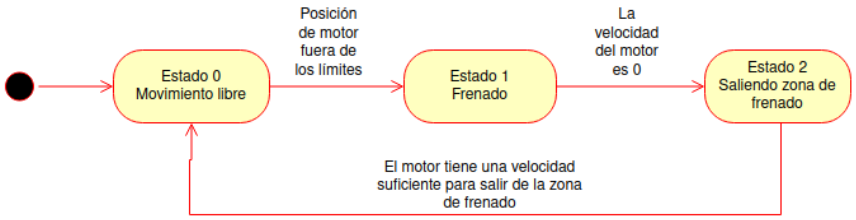


Figura 5.4: Diagrama de estados para evitar colisiones.

C *Control PID (Nivel Bajo)*

En el robot real, la API de MyCobot ofrece varias funciones que permiten mover a los motores a determinadas posiciones. La usada en este trabajo es *send_ angles* que toma la posición de los 6 motores y la velocidad de movimiento. De esta manera, internamente el sistema implementa un control PID para lograr la correcta ejecución de la función. Por esta razón, es necesaria la configuración de un control PID en el robot simulado. Gazebo proporciona las funciones necesarias para configurar los parámetros del control de manera sencilla.

Para fines de la simulación se propusieron dos configuraciones de PID mostradas en la tabla 5.3, la primera es la que representa lo más cercano a las condiciones reales del robot. No obstante, la segunda configuración es para usarse cuando Gazebo es acelerado para tener un aprendizaje más rápido. Esto se debe a que en una simulación en tiempo real el paso de tiempo es de $dt \approx 0.001$, mientras que en una acelerada 3 veces el paso de tiempo es $dt \approx 0.0035$. Entonces, teniendo en cuenta que el control del modelo PID depende del paso de tiempo dt, valores específicos de los parámetros en diferentes dt pueden tener efectos muy diferentes e incluso pueden llegar a desestabilizar los motores. Por ello, se especificaron valores más pequeños de PID para garantizar la estabilidad a pasos grandes de dt.

Tabla 5.3: Configuración de parámetros PID para el robot

Velocidad de simulación	P	I	D
Simulación normal - dt = 0.001	10.0	0	0.5
Simulación acelerada - dt = 0.0035	7.0	0	0.025

Para el salto de tiempo $dt \approx 0.001$ se tomó en cuenta la configuración de PID que especifica la API de MyCobot. Esta información se puede conseguir con la función get_servo_data.

En las figuras 5.5 y 5.6 se pueden apreciar las respuestas PID de los motores en simulación y en físico, respectivamente.

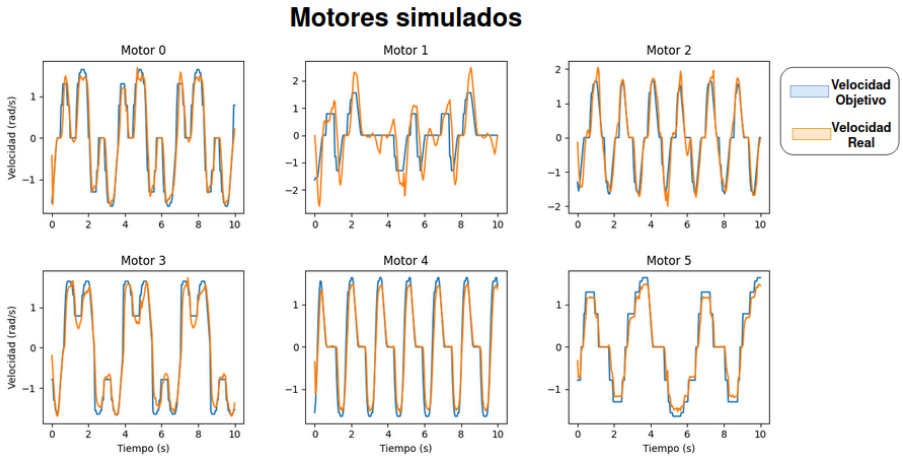


Figura 5.5: Respuesta de control PID de motores simulados.

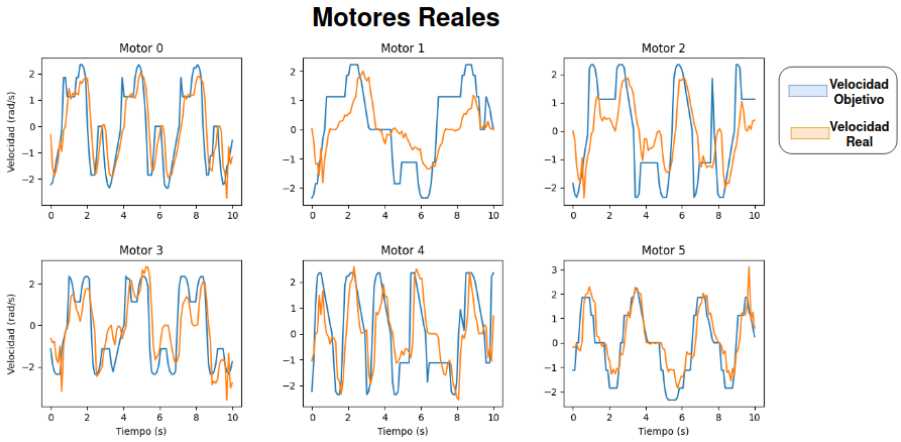


Figura 5.6: Respuesta de control PID de motores reales.

Los motores que más discrepan de la simulación son los motores 1 y 2, debido a que son los que más peso cargan cuando el robot está inclinado, haciendo que por inercia aumente o disminuya su velocidad.

Además, en el control real, cuando se está constantemente leyendo las lecturas de los motores, tiende a haber fallas, es decir, que la API de pymycobot regresa una lectura vacía, haciendo que la actualización del control no sea posible por unos milisegundos, esto principalmente por cuestiones de seguridad. Es por ello que hasta las gráficas de la velocidad objetivo en ambas figuras hay diferencias.

5.2. Entrenamiento con redes convolucionales

En este trabajo se plantean dos posibles estrategias para poder manejar al brazo robótico mediante un agente de aprendizaje por refuerzo profundo, la detección de objetos que tiene como objetivo identificar a los cubos dentro del juego y pasar sus posiciones a la red neuronal. Por otra parte, la red convolucional buscaría solamente obtener la imagen del juego, procesarla mediante capas de convolución y agrupación máxima para extraer características y pasar la información resultante al agente para que produzca una acción. Sin embargo, al momento de hacer la investigación de las oportunidades del método de convoluciones, se observó que había serias desventajas que podrían afectar a la destreza y capacidad de aprendizaje del brazo robótico.

Se toman como punto de comparación trabajos donde el brazo robótico ha tenido que realizar tareas como poner el efector final en una posición o en el levantamiento de objetos, ya que estas acciones involucran aproximarse a un objetivo estático, lo cual es un buen punto de partida si se quiere comparar con objetivos que se mueven. De esta manera, a continuación se mencionan los inconvenientes de la estrategia de convoluciones basados en la investigación y en el análisis de las características necesarias para cumplir los objetivos del trabajo.

- **Alta dimensionalidad**: La ventaja directa de manejar una imagen como entrada de la red neuronal del agente es que éste tendría un aprendizaje más cercano al humano y reduciría la complejidad de idear una observación que represente al ambiente, pero como consecuencia, la dimensionalidad de la entrada de la red neuronal aumenta considerablemente. A pesar de ser procesada por la red neuronal convolucional, se consiguen dimensiones mayores a 1500 neuronas. La consecuencia más directa de

esto es que toma un tiempo muy grande para entrenar al agente. Esto se ve reflejado si se comparan los pasos necesarios para tener resultados aceptables. Por ejemplo, en trabajos donde se considera una detección de objetos [29, 30] usualmente se usan aproximadamente 100,000 pasos para el entrenamiento; por otro lado, trabajos en donde se toma completamente la imagen [23, 24, 25, 26], aproximadamente demora más de 1 millón de pasos para tener resultados óptimos. Esto quiere decir que el método de convoluciones suele demorar de 8 a 15 veces más dependiendo del caso.

- **Incertidumbre en el estado del robot y tasa de éxito**: Con respecto al problema anterior, el estado de los motores del robot no se puede concatenar directamente con la imagen, debido a que el número de neuronas representando a la imagen es mayor al número representando al estado del robot (solamente 6 en el caso de MyCobot), por lo que las neuronas del robot tendrían poca o nula influencia en la decisión.

  Para superar este problema se han propuesto varias soluciones. Una de ellas es que, además de proporcionar una imagen del ambiente, también se proporcionen múltiples imágenes desde diferentes ángulos del brazo robótico para así inferir su posición. Esta técnica no ha mostrado resultados tan alentadores con los algoritmos que nos ofrece stable-baselines-3, ya que en entornos reales no se ha logrado una tasa de éxito mayor al 60 % en tareas como posicionamiento del efector final en un punto [26] o en levantamiento de objetos [23, 24, 25]. En contraste, los modelos que utilizan una estrategia de detección de objetos [29] o directamente YOLO [30] para el levantamiento de objetos, logran una tasa de éxito de más del 95 %.

  Una estrategia que ha mostrado mejores resultados es que aparte de extraer características con las capas convolucionales, se tiene que codificar el estado del robot a una dimensión más grande mediante el uso de una red neuronal aparte. Este método ha logrado ser más eficiente con una tasa de éxito del 91.1 % [81] pero requiriendo múltiples cámaras con diferentes perspectivas del robot.

- **Simulación limitada**: La aproximación más directa para producir la imagen del juego es directamente extrayéndola cuando este se está ejecutando. Pero esto trae varias desventajas: una sería que, al depender directamente de la ejecución del juego,

se reducen las posibilidades de poder acelerar la simulación del robot, ya que el tiempo estaría atado al tiempo marcado por *Beat Saber*. La aceleración del juego podría ser posible si se inyecta código en él; no obstante, para lograr esto se tendría que aplicar ingeniería inversa para descifrar una manera de manipular la velocidad de la canción y hacer el juego más rápido, pero idear un código que permita hacer lo anterior puede ser exhaustivo.

Además, no se podría implementar un aprendizaje con múltiples brazos robóticos, ya que cada agente necesitaría una ejecución única del juego, lo cuál podría utilizar muchos recursos, o en su defecto, el juego no permitiría tener varias instancias de sí mismo debido a la configuración de *Steam*.

- **Recreación de ambiente virtual compleja**: Una alternativa para superar los retos descritos en el punto anterior es la creación de un ambiente propio para poder manejar todos los aspectos de velocidad, y a su vez producir imágenes que representen de manera fidedigna la dinámica de juego. Desgraciadamente, resultaría complejo recrear una simplificación del entorno de *Beat Saber*, ya que este posee muchas animaciones, efectos de cámara y cambios de iluminación, esto inclusive desactivando todos los efectos especiales posibles dentro del juego real. Entonces, si no se hace una representación fiel, puede que el agente no se desempeñe bien cuando esté en el ambiente original. Además, se tendría que programar la dinámica del juego para que el ambiente sea responsivo a los movimientos del robot.

Por las razones antes descritas, se descartó utilizar la estrategia de redes neuronales convolucionales y seguir con la detección de objetos, ya que es la que puede tener mejores resultados en un tiempo mucho menor. No obstante, no se descarta la posibilidad de trabajar con este método en un trabajo futuro con el fin de mejorar los resultados o la generalización del agente.

5.3. Entrenamiento virtual con detección de objetos

En esta sección se explica la estrategia que se siguió para conseguir a un agente de aprendizaje por refuerzo profundo que maneje cada motor del robot para que sea capaz de jugar *Beat Saber* con el set de realidad virtual HTC Vive, esto impulsado por la detección de objetos que identifica los cubos y la dirección de corte dentro del juego.

Primero se explica la arquitectura asíncrona de nodos en ROS, la cual involucra al agente, el control del robot, la simulación en Gazebo y un motor de juego muy básico creado para este trabajo que sirve como una simplificación de *Beat Saber*. También se explican más a detalle los elementos de aprendizaje por refuerzo profundo involucrados y la implementación de YOLO para detectar cubos.

Además, se presentan los resultados del entrenamiento usando la simplificación de *Beat Saber* y pruebas con el robot simulando ya con una política entrenada jugando al título real.

5.3.1. Arquitectura de entrenamiento en ROS

La arquitectura utilizada para entrenar es paralela, es decir, que se inicializan varios robots simulados al mismo tiempo y entrenan sin compartir ningún tipo de información. No obstante, para profundizar más en este método, primero se tiene que entender el funcionamiento del sistema para un solo robot. El diagrama que describe toda la arquitectura se muestra en la figura 5.7.

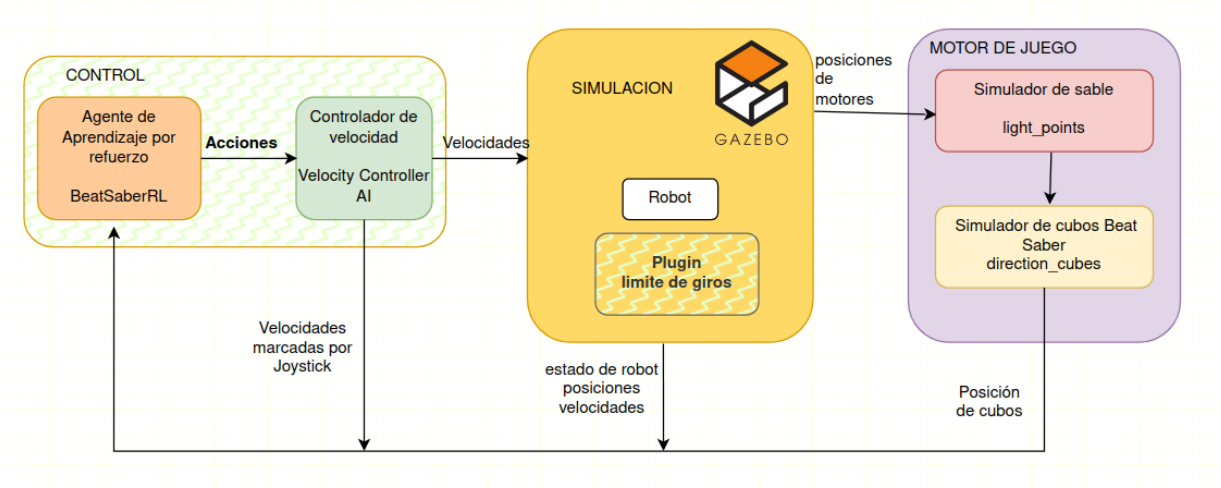


Figura 5.7: Arquitectura asíncrona de entrenamiento en ROS para un solo robot.

Es de vital importancia mencionar que la arquitectura es asíncrona, es decir, que la simulación no está centrada en una sola entidad. Esto trae ventajas y desventajas como las mencionadas a continuación.

Ventajas

- **Alta modularidad**: Como cada nodo tiene una tarea en específico, esto permite ejecutarlos en diferentes máquinas para repartir la carga de trabajo como se crea conveniente.
- **Depuración eficiente**: La simulación está dividida en módulos que pueden ser activados o desactivados sin afectar directamente a los otros. Esto permite hacer cambios en el código de un módulo en tiempo real y ponerlo a prueba sin apagar todo el sistema entero, especialmente si se usa Python como lenguaje de programación.
- **Noción de tiempos de respuesta**: Con las herramientas que proporciona ROS, es posible regular la frecuencia de procesamiento de cada nodo individualmente, de tal manera que la frecuencia se parezca al tiempo de respuesta de un componente real. Por ejemplo, el tiempo que le toma al robot para enviar las posiciones deseadas a cada motor del robot.

Desventajas

- **Limite en el tiempo de aceleración**: Como la comunicación entre nodos es por paso de mensajes por *socket*s, este protocolo de comunicación no es el más rápido para ser usado en la comunicación entre procesos, ocasionando una latencia que puede ser despreciable en tiempo normal. No obstante, si se acelera el tiempo de simulación, el impacto de la latencia puede ser dañino para los módulos, ya que estarían recibiendo información muy tardía o sin coherencia. Por lo tanto, el paso de mensajes es un obstáculo para la aceleración del sistema.

Siguiendo con el funcionamiento de la arquitectura, de la figura 5.7 se pueden identificar cinco nodos principales: *BeatSaberRL, velocity_ controller_ AI, límite de giros, light_ points y direction_ cubes*; cada uno cumple tareas específicas para que el sistema funcione. A continuación se brinda una explicación de cada nodo.

Nodo light points

Su objetivo es crear la identidad del sable para cortar los cubos. Su estructura simplemente se basa en definir dos puntos en el efector final del brazo robótico, los cuales marcan en coordenadas globales del comienzo y el fin del sable, o en otras palabras, se modela como un segmento en el espacio que depende de dos puntos.

Para lograr esto, el nodo realiza matrices de transformación con respecto a las articulaciones del robot. Las matrices de translación y rotación se muestran en las ecuaciones 5.4 y 5.5 respectivamente, donde x,y,z expresan la localización local de la articulación con respecto a su padre (articulación anterior) y qx,qy,qz y qw los cuaterniones de rotación locales.

$$T_A = \begin{pmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{5.4}$$

$$R_A = \begin{pmatrix} 1-2qy^2-2qz^2 & 2(qx)qy-2(qw)qz & 2(qx)qz+2(qw)qy & 0 \\ 2(qx)qy+2(qw)qz & 1-2qx^2-2qz^2 & 2(qy)qz-2(qw)qx & 0 \\ 2(qx)qz-2(qw)qy & 2(qy)qz-2(qw)qx & 1-2qx^2-2qy^2 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{5.5}$$

Entonces, si se multiplica la matriz de translación con la de rotación, se obtiene la matriz de transformación del espacio de coordenadas A, como se muestra en la ecuación 5.6.

$$T_{transA} = T_A R_A \tag{5.6}$$

Por lo tanto, para conseguir la matriz de transformación que mapea de un espacio de coordenadas de una articulación B (Hijo) a otra articulación A (Padre), se tienen que multiplicar las matrices de transformación de ambas, como se puede apreciar en la ecuación 5.7. Con ello se pueden conseguir coordinadas pertenecientes a B expresadas en términos de A.

$$T_{A->B} = T_{transA} T_{transB} \tag{5.7}$$

En la simulación, si la multiplicación de matrices de transformación se hace de manera concatenada desde el origen de coordenadas, después a la primera articulación del robot y hasta el efector final del mismo, se obtiene la matriz $T_{efector}$ que mapea puntos del espacio de co-ordenadas del efector final a puntos globales. Tomando en consideración esto, el cálculo de los puntos globales se expresa en la ecuación 5.8. Donde x,y,z representan valores en el espacio de coordenadas del efector final. Para definir los dos puntos del segmento del sable, en este trabajo se propuso como origen el punto (-0.06,0.2,0) y como punto final (-0.3,0.2,0) con respecto al efector final; esto da un sable de $0.3 - 0.06 = 0.24$ unidades de longitud.

$$P_{origen} = T_{efector} \begin{pmatrix} x_{origen} \\ y_{origen} \\ z_{origen} \\ 1 \end{pmatrix}, P_{fin} = T_{efector} \begin{pmatrix} x_{fin} \\ y_{fin} \\ z_{fin} \\ 1 \end{pmatrix} \tag{5.8}$$

Por otra parte, para simular la dirección de corte, se analiza la forma de moverse de cada punto del sable. Para facilitar esto, se propone un vector que va desde el punto de origen al punto final del sable, dando como resultado el vector $\vec{S}$, como se muestra en la expresión 5.9.

$$\vec{S} = P_{fin} - P_{origen} \tag{5.9}$$

Con el vector $\vec{S}$, para conseguir el vector de velocidad, simplemente se resta el vector $\vec{S}$ en un tiempo t con el del tiempo t-1 para obtener una derivada de la posición. Esto se muestra en la ecuación 5.10.

$$\vec{S_v} = \dot{\vec{S}} = S_t - S_{t-1} \tag{5.10}$$

Con todos los elementos mencionados, el sable puede ser visualizado como se muestra en la figura 5.8. Donde la línea roja es el sable y la flecha azul es el vector de velocidad $\vec{S_v}$.

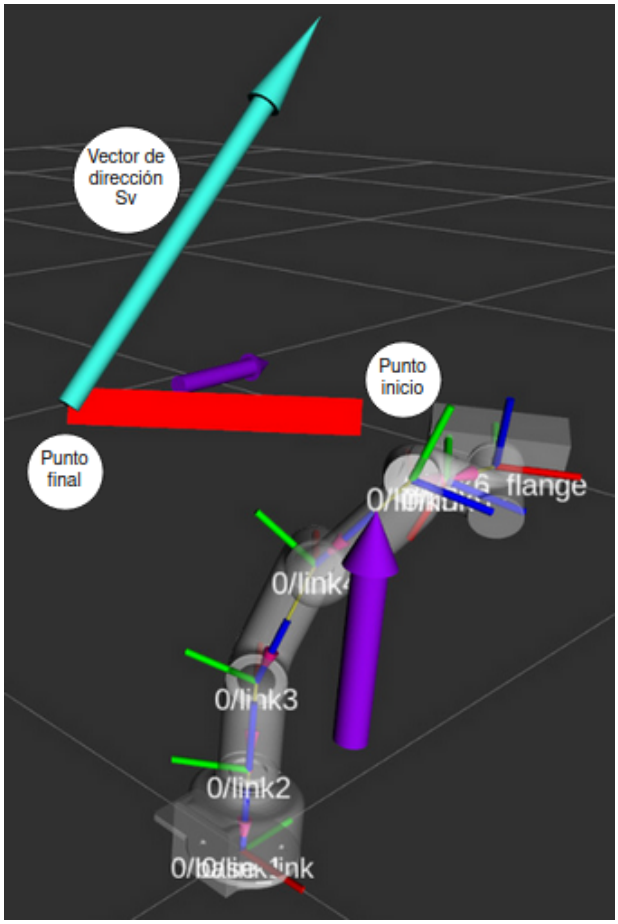


Figura 5.8: Sable simulado con dirección.

Nodo direction cubes

El nodo tiene la finalidad de brindar un ambiente que ejemplifica la dinámica de *Beat Saber*, es decir, es capaz de generar cubos con una dirección de corte en especifico, los cuales se aproximan al robot, para que éste pueda cortarlos con el sable planteado en *light_points*. El objetivo principal de tener este ambiente es tener control total sobre la velocidad del juego, y así poder acelerar mucho más el proceso de aprendizaje.

Cabe mencionar que en el entrenamiento no se hace una detección de objetos para los cubos, en su lugar se utilizan las características de los cubos producidos por el nodo, no obstante, en el proceso de evaluación sí se utiliza la detección de objetos.

Cada cubo del ambiente tiene las siguientes características:

- **Posición**: Vector que caracteriza la posición tridimensional del cubo con respecto al origen de coordenadas. Se denota con C_{pos}.

- **Dirección de corte**: Dado que la dirección de corte de los cubos son paralelos al jugador y todos son múltiplos de 45°, solo basta con un vector bidimensional de la forma (x,y) para especificar la dirección de corte; este vector se denota con C_{dir}. En la figura 5.9 se pueden apreciar los diferentes valores que puede tomar este vector considerando la dirección del cubo en *Beat Saber*.

Figura 5.9: Posibles valores de C_{dir} dependiendo de la dirección del cubo.

- **Tamaño**: Especifica las dimensiones del cubo, entre más grande, será más fácil para el agente cortarlo.

- **Velocidad**: Es la velocidad con la que el cubo se acerca al jugador. Es lineal, por lo que solo depende de un número y se asigna aleatoriamente cuando el cubo es creado. Se utilizan cubos con diferentes velocidades para que el agente no se acostumbre a una sola velocidad.

- **Estatus**: El cubo puede tener 4 estatus diferentes: si todavía no es golpeado (0 amarillo), el cubo pasó un límite y no fue golpeado (-1 rojo), se golpeó pero no en la dirección correcta (1 verde) y se cortó en la dirección correcta (2 azul). En la figura 5.10 se puede ver una representación de estos estatus.

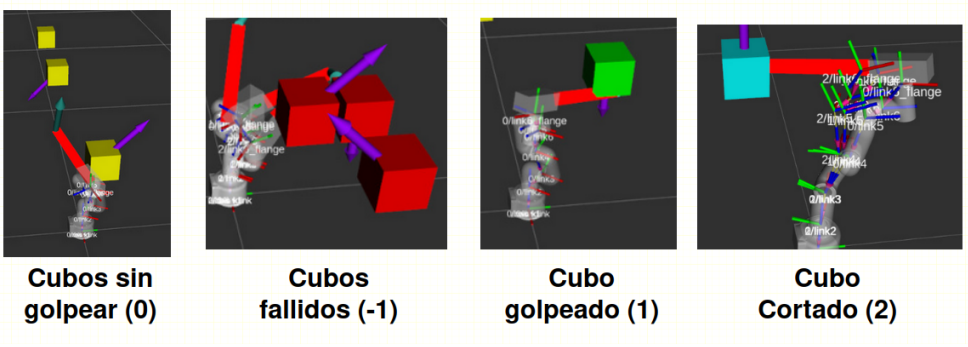


Figura 5.10: Diferentes estatus de cubo visualizados.

Para la detección de colisión entre el sable y los cubos se utilizó una implementación disponible en internet[2], la cual modela la intersección entre un cubo sin rotación con un segmento conformado por dos puntos. Estas características fueron perfectas para adaptarse al ambiente realizado.

Con el fin de detectar si un corte se hizo en la dirección correcta, se usa la expresión 5.11 que usa la operación de ángulo entre vectores para medir que tan similares son el vector de la velocidad del sable $\vec{S_v}$ y el vector de corte del cubo colisionado $\vec{C_{dir}}$.

$$cos(\theta) = \frac{\vec{S_v} \cdot \vec{C_{dir}}}{|\vec{S_v}||\vec{C_{dir}}|} \tag{5.11}$$

Recordando que la operación coseno tiene como rango de valores [-1,1], implica que cuando el valor es -1 los vectores son paralelos pero en dirección contraria, cuando es 0 los vectores son perpendiculares y cuando es 1 los vectores son paralelos con la misma dirección. Por ello, el objetivo para que se detecte un corte correcto es que el coseno sea lo más cercano a 1. Dado esto, se establece un valor umbral que define un valor mínimo del coseno para que se considere un corte correcto.

Nodo agente de aprendizaje profundo

Es el nodo encargado de cargar el agente de aprendizaje por refuerzo profundo para que comience a generar acciones basándose en el estado del motor del juego y el robot.

Este nodo se explica a profundidad en la sección 5.3.3. No obstante, es de importancia mencionar que el agente puede controlar la continuidad del sistema entero, es decir, que es

[2] https://3dkingdoms.com/weekly/weekly.php?a=3

capaz de detener y reanudar el tiempo de todos los módulos. Esto se debe a que cuando se está entrenando con modelos tipo *on-policy* (Ej. DDPG y TD3) cada vez que se termina un episodio (se golpea, corta o falla un cubo), la producción de acciones debe frenarse para que se pueda iniciar el ajuste de parámetros con la memoria de experiencia, lo cual lleva unas cuantas fracciones de segundo. De esta manera, si todos los componentes no se detuvieran, el agente quedaría incomunicado por ese intervalo de tiempo, ocasionando que el entrenamiento del agente sea más difícil o incluso inestable.

Nodo Velocity controller AI

El nodo se encarga de modelar el elemento de control de nivel medio del joystick para que el agente pueda manejar los motores del robot. Esta entidad se explica a detalle en la sección 5.1.2 correspondiente al control de robot.

Nodo plugin limite de giros

El nodo incorporado en un plugin de Gazebo tiene como objetivo traducir las velocidades del joystick a posiciones de los motores que el control PID se encargará de ejecutar. Al mismo tiempo, el nodo implementa el mecanismo para frenar al robot en caso de que algún motor sobre pase el límite propuesto, dicho mecanismo está descrito en la sección 5.1.2.

5.3.2. Arquitectura de agentes en paralelo

La arquitectura antes mencionada funciona eficazmente para entrenar a un único agente, a pesar de ello, tener solo a uno puede traer ciertos inconvenientes que obstaculizan el entrenamiento de una buena política. A continuación se mencionan estos problemas.

- **Inestabilidad de modelos**: Muchas veces los modelos muestran estabilidad a lo largo del entrenamiento, no obstante, debido a un comportamiento inesperado de los factores de exploración y explotación, existe la posibilidad de que el agente siga un comportamiento indeseado, haciendo que quede atrapado en una política que no resuelva el problema, o que al agente se le sea muy difícil retomar un aprendizaje

adecuado. En la figura 5.11 se muestra que, mientras una versión de un modelo aprende apropiadamente, logrando obtener recompensa positiva, otra versión idéntica en las mismas condiciones muestra cómo no logra aprender eficientemente, por lo que recibe recompensas negativas.

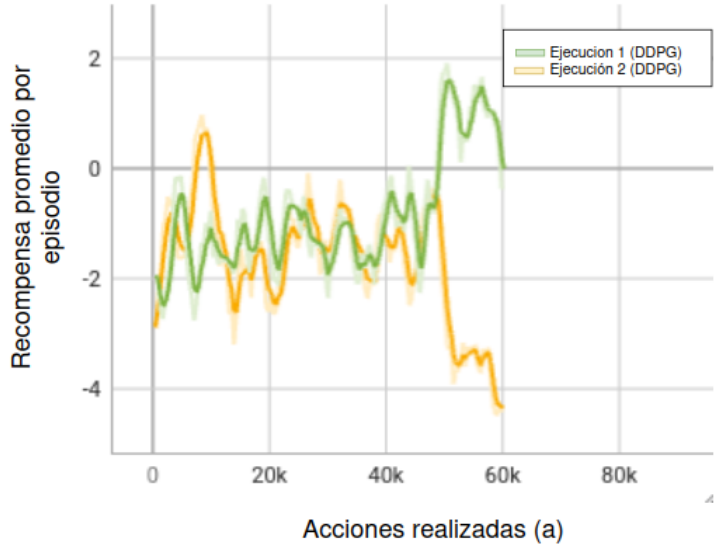


Figura 5.11: Modelo estable (verde) modelo no deseado (amarillo).

- **Exploración limitada**: Como se mencionó en el punto anterior, hay posibilidades de que cuando se ejecute un modelo, se obtengan resultados diferentes, ya que dependiendo de las condiciones del ambiente, el agente puede deducir diversas estrategias mediante la exploración, las cuales pueden llevar a obtener una política eficiente. Sin embargo, estar probando modelos uno por uno para encontrar dicha política puede ser exhaustivo, debido a que generalmente cada modelo lleva un tiempo considerable para ver resultados significativos.

- **Desaprovechamiento de recursos**: Cuando se entrena a un solo agente con un tiempo de aceleración en la simulación, solo una parte de los recursos de una computadora son utilizados. De esta manera, una forma de aprovechar mejor los recursos es acelerando aún más el tiempo de todos los módulos y de la simulación, pero esto pondría en riesgo la cohesión del sistema. Por ejemplo, si un cubo va a una velocidad muy alta, podría atravesar el sable sin ser cortado, ya que los saltos en su posición serían muy grandes.

Por todos los inconvenientes mencionados, se decidió idear una arquitectura que permitiera crear varios agentes que trabajen de manera paralela con el fin de encontrar una buena política en menor tiempo. Al mencionar que se ejecutan de manera paralela, se refiere a que los agentes no comparten ningún tipo de recurso, son solo procesos que se ejecutan aisladamente; sin embargo, sí existe una comunicación entre ellos que se explica más adelante. En la figura 5.12 se muestra un diagrama de la arquitectura propuesta.

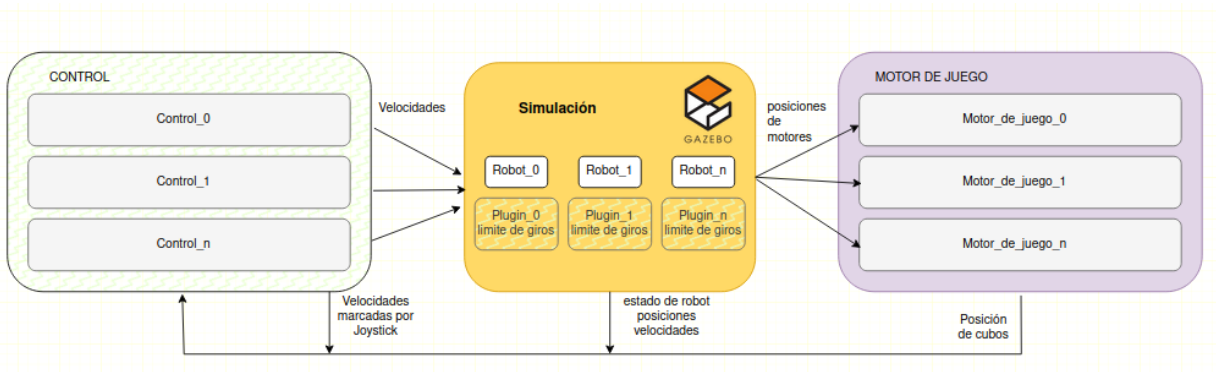


Figura 5.12: Arquitectura de entrenamiento paralelo en ROS.

La arquitectura paralela la conforman los mismos nodos y bloques que en la arquitectura de un solo agente, pero con la diferencia de que se inicializan múltiples nodos del mismo tipo, de los cuales a los tópicos se les asigna un canal de comunicación diferente. Esto se logra añadiendo un sufijo (1,2,3,..,n) a cada tópico de un conjunto de nodos del mismo tipo. Por ejemplo, si el nodo *direction_cubes* originalmente tenía un tópico llamado *game_info* ahora a cada nodo del grupo se le asignará un respectivo tópico del tipo *game_info_0*, *game_info_1* y hasta *game_info_n*, ocasionando que sólo los nodos con

el mismo sufijo de tópico se puedan comunicar entre ellos, y aislándose de los demás. El número n representa la cantidad de agentes que se desean entrenar al mismo tiempo.

De esta manera, cada agente inicializado podrá tener un canal de comunicación único para poder manejar al robot correspondiente.

La entidad que permanece como única es la simulación Gazebo, ya que dentro de ella se simulan los n robots especificados. En la figura 5.13 se pueden ver múltiples robots simulados y en la figura 5.14 se observan los tópicos de cada canal de comunicación.

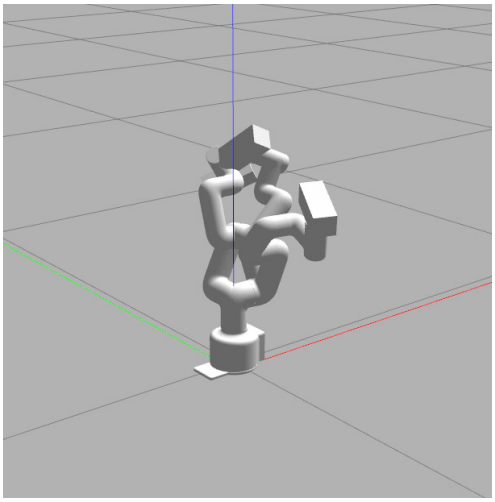


Figura 5.13: Múltiples robots simulados en Gazebo.

```
serapf@serapf-B450-AORUS-M:~/catkin_ws/BeatSaberRL/try3$ rostopic echo /
/agent_manager_status           /light_points_1
/agents_status                  /light_points_2
/clicked_point                  /move_base_simple/goal
/clock                          /mycobot_jointstates_0
/cubes_info_0                   /mycobot_jointstates_1
/cubes_info_1                   /mycobot_jointstates_2
/cubes_info_2                   /rosout
/game_info_0                    /rosout_agg
/game_info_1                    /tf
/game_info_2                    /tf_global_0
/gazebo/link_states             /tf_global_1
/gazebo/model_states            /tf_global_2
/gazebo/parameter_descriptions  /tf_static
/gazebo/parameter_updates       /velocity_controller_0
/gazebo/performance_metrics     /velocity_controller_1
/gazebo/set_link_state          /velocity_controller_2
/gazebo/set_model_state         /visualization_marker_0
/IA_actions_0                   /visualization_marker_0_array
/IA_actions_1                   /visualization_marker_1
/IA_actions_2                   /visualization_marker_1_array
/IA_status_0                    /visualization_marker_2
/IA_status_1                    /visualization_marker_2_array
/IA_status_2                    /visualization_marker_3
```

Figura 5.14: Múltiples tópicos para diferentes canales de comunicación.

Siguiendo la estrategia de paralelismo, la gráfica de la recompensa a través del tiempo tendría el aspecto mostrado en la figura 5.15. En donde cada curva de diferente color representa a un agente. De la misma figura, se puede observar los diferentes comportamientos que puede tomar un mismo agente, principalmente en el comienzo del entrenamiento.

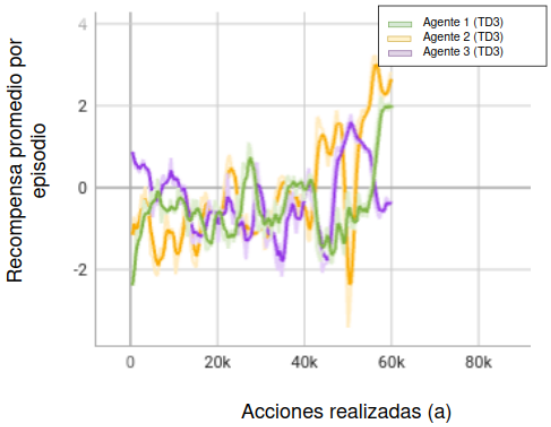


Figura 5.15: Gráfica de recompensa de múltiples agentes entrenando al mismo tiempo.

Este paradigma de comunicación en paralelo de ROS está presente en diferentes investigaciones relacionadas con aprendizaje por refuerzo profundo, tales como *Parallel Gym Gazebo* [82] o *MultiROS* [83] donde se muestran arquitecturas muy similares a la presentada en este trabajo, sin embargo, se decidió por hacer la propia para fines de simplicidad y poder modificar aspectos que se ajusten al objetivo principal.

Otro mecanismo propuesto en la arquitectura en paralelo para tener un entrenamiento más eficiente es la creación de generaciones. En este trabajo, se define como generación a la ejecución de n agentes a lo largo de un periodo de tiempo, delimitado por una cantidad a de acciones realizadas por un agente, para que cuando se alcance ese límite de acciones, se seleccione al agente que mejor entrenamiento tuvo, se descarten los otros y se remplacen con copias del mejor para que así se inicie una nueva generación. Esta estrategia imita a algoritmos evolutivos de selección natural. En la figura 5.16 se muestra el comportamiento de las generaciones en el entrenamiento; el cual expresa la selección del modelo mejor entrenando cada 60 mil acciones.

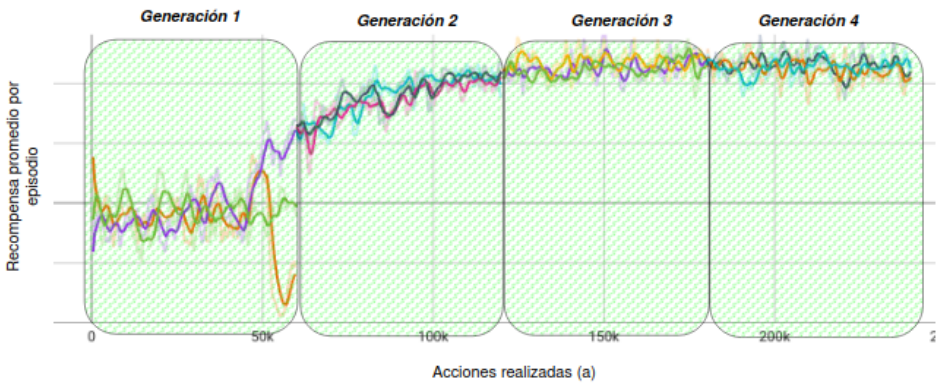


Figura 5.16: Generaciones (60K acciones cada una) a lo largo del entrenamiento.

La selección del mejor agente implica una coordinación de todos los agentes para que terminen una generación al mismo tiempo y puedan comunicar sus resultados. No obstante, la comunicación agente-agente puede ser algo complicada de desarrollar. Por lo mismo, se ideó un nodo administrador que tiene como tarea coordinar a todos los agentes y hacer la selección del mejor. En el diagrama de la figura 5.17 se aprecia la arquitectura central que se maneja para comunicar al administrador con los agentes.

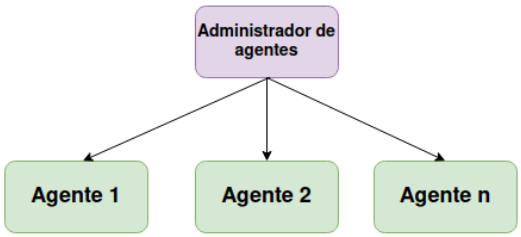


Figura 5.17: Arquitectura central de administrador y agentes.

De esta manera, la comunicación detallada entre los agentes y el administrador se muestra en el diagrama de la figura 5.18.

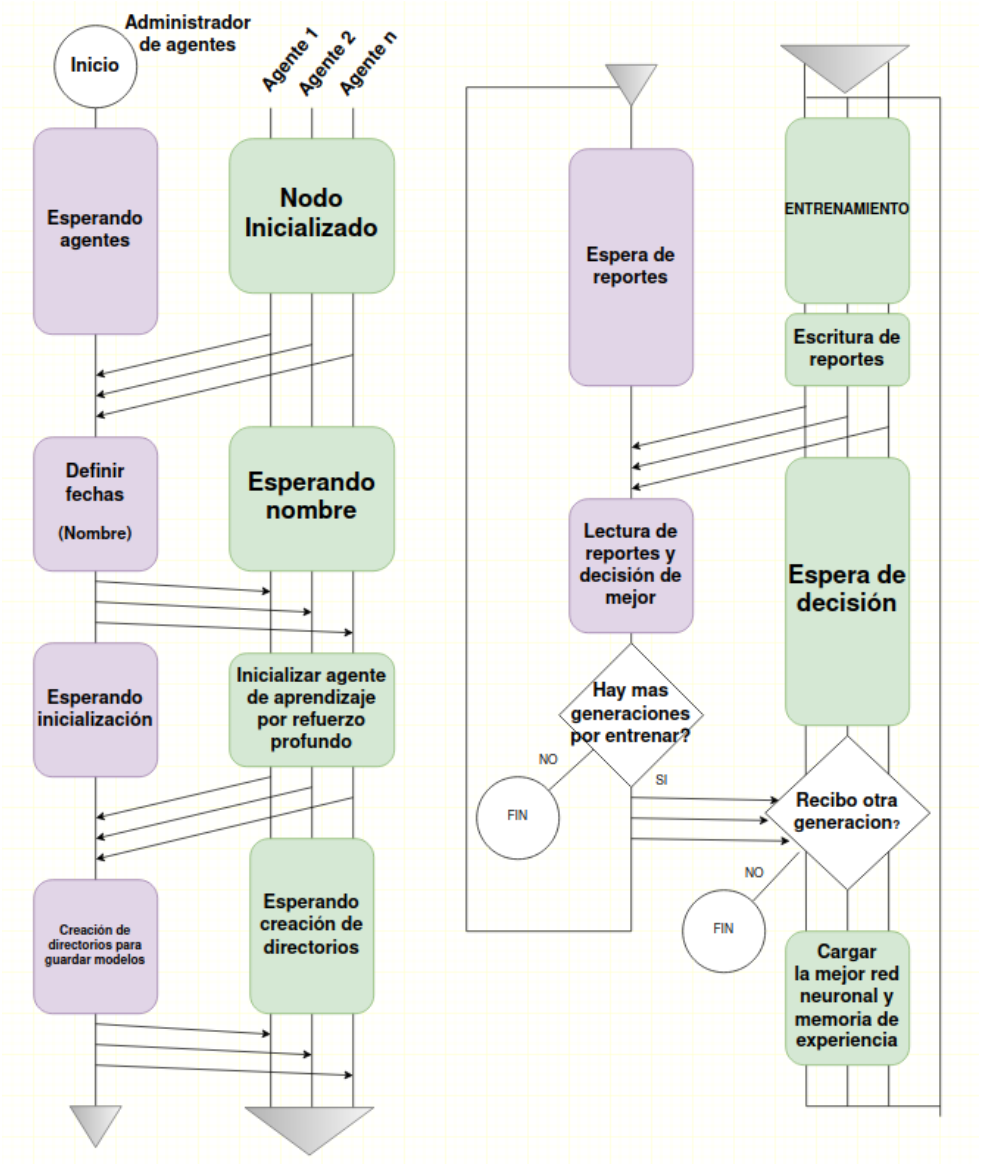


Figura 5.18: Diagrama de comunicación entre el nodo administrador y los nodos de agentes.

La comunicación para la sincronización de agentes se describe en los siguientes pasos:

1. El administrador de agentes se inicializa y se especifica el número n de agentes a entrenar. Con ello, el nodo espera a que los n agentes inicien una comunicación.

2. Una vez inicializados los n agentes, el administrador define la misma fecha de creación para todos los agentes involucrados en el entrenamiento. Dicha fecha sirve como nombre para diferenciar diferentes ejecuciones del entrenamiento por generaciones.

3. Cuando ya se tienen los nombres definidos, cada agente inicializa el ambiente, la red neuronal y la memoria de repetición de experiencia. Además, el administrador crea directorios donde se guardan todos los agentes entrenados en una generación para su previo uso.

4. Cada agente inicia individualmente el entrenamiento de una generación, entonces cuando esta se termina, cada agente escribe un reporte donde viene un resumen de su desempeño a lo largo de toda la generación, incluyendo información como recompensa acumulada, cubos fallidos, golpeados, cortados y el radio de precisión, el cual se explicará más adelante.

5. El administrador lee los reportes, y selecciona al mejor con base en la recompensa acumulada obtenida. En la figura 5.19 se puede observar el contenido de cada reporte y la selección del mejor.

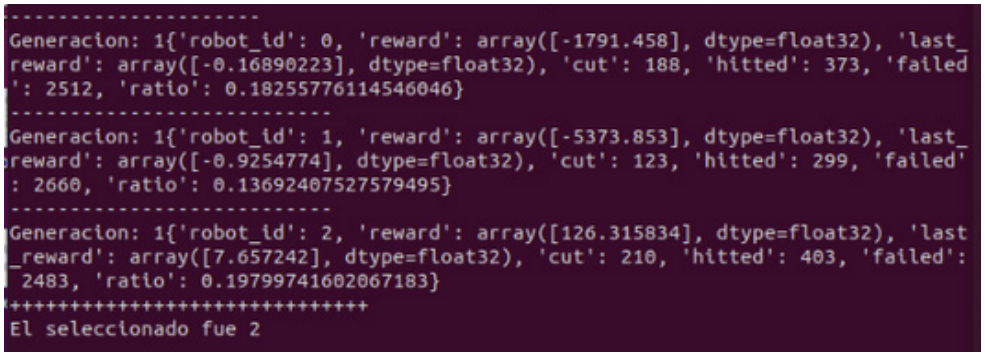


Figura 5.19: Ejemplo de reporte escrito por cada agente y selección del mejor.

6. Si todavía hay más generaciones G por entrenar, el administrador comunica a los agentes cuál fue el mejor para que se reemplacen por el seleccionado y vuelvan a iniciar el entrenamiento del paso 4. Por otra parte, si ya se entrenaron todas las generaciones, el administrador antes de terminar indica a los agentes que ya se acabó el entrenamiento, para que así también finalicen su ejecución.

Recapitulando lo visto en esta sección, se presentó una arquitectura en ROS que permite el entrenamiento paralelo de agentes de aprendizaje por refuerzo profundo, esto combinando la idea de generaciones G que consiste en la selección del mejor modelo cada cierto número de acciones a. Todo esto con el fin de hacer más eficiente la búsqueda de una política π óptima.

5.3.3. Descripción de agente y ambiente para detección de objetos

En esta sección se describe a detalle todas las características del ambiente y del agente necesarias para llevar a cabo un aprendizaje por refuerzo profundo, basándose en el entorno virtual que simplifica la dinámica de *Beat Saber*. La incorporación de YOLO para la detección de objetos se implementa para el proceso de evaluación descrita en la sección 5.5.

Ambiente

Recordando que el ambiente es el que define la información disponible que el agente puede usar para su aprendizaje, se tiene que esta entidad se encarga de suscribirse a los tópicos necesarios provenientes de los nodos de simulación de Gazebo, de control y del motor de juego, para que así pueda filtrar y ordenar los datos para que el agente los pueda manejar. De esta manera, el agente tendrá la tarea de cortar con el sable a los cubos que vayan apareciendo y se aproximen hacia él.

En la figura 5.20, se puede observar la inicialización del ambiente, lo cual consiste en que cada suscriptor del ambiente reciba datos del nodo correspondiente.

```
Getting light points nodes
Waiting game controller info
Waiting for robot joint state node
Waiting for cubes info node
Nuevo modelo creado: DDPG GENE - cu 2 - Arw brain 0
Using cuda device
Wrapping the env with a `Monitor` wrapper
Wrapping the env in a DummyVecEnv.
```

Figura 5.20: Inicialización del ambiente del agente.

Cabe mencionar que cada vez que un cubo es cortado, golpeado o fallado termina un episodio para poder aprender de la experiencia. Lo anterior implica que no hay un límite exacto de pasos para truncar el episodio. Aun así, se puede decir que la duración aproximada de un episodio es de 40 pasos, puede ser menos o más, dependiendo de la frecuencia de aparición y velocidad de los cubos.

Todo el código del ambiente se encuentra en el apéndice A.

Estados (observaciones)

Definir una observación que de una buena concepción de todo el ambiente es crucial para el buen aprendizaje del agente. Por ello, se consideraron los siguientes aspectos para formar una representación del estado del sistema:

- **Estado del robot**: De la simulación de Gazebo se extraen los datos sobre la posición y velocidad que tiene cada motor. Como son 6 motores involucrados, esto nos daría 12 neuronas.

- **Estado del joystick**: Para que el agente tenga un apoyo de la noción de la velocidad de cada motor, la salida de velocidades del joystick también es tomada en cuenta, dando como resultado 6 neuronas a considerar.

- **Características de cubos**: Con respecto a los cubos, de cada uno se toman en cuenta las coordenadas tridimensionales C_{pos} y el vector de corte C_{dir}. Este último se menciona que es de 2 dimensiones solamente, pero para fines de simplicidad para la operación de la recompensa, se añadió una dimensión más. Con esto dicho se tendrían 6 neuronas por cubo.

- **Cantidad de cubos**: Lo deseable es que el agente considere a todos los cubos presentes en el ambiente virtual, sin embargo, la dimensión fija de la entrada de una red neuronal limita cuántos cubos se pueden tomar en cuenta. Entonces, para enfrentar este inconveniente, se establece un número $n_{cu} \in \mathbb{N} \geq 1$ que indica cuantos cubos se van a considerar para la observación, donde el primer cubo a ser considerado es aquel que esté más cercano al robot y, a partir de este hacia atrás, hasta el cubo n_{cu}. Por lo tanto, las neuronas a considerar serían $6 * n_{cu}$.

  De esta mecánica, se tiene que tener en cuenta casos espaciales. Uno sería cuando la cantidad de cubos en el ambiente es menor a los cubos a considerar n_{cu}, en este caso se rellenan las neuronas correspondientes a los cubos faltantes con un vector de ceros $(0, 0, 0, 0, 0, 0)$. Por otra parte, si no hay un cubo presente en el ambiente, se da un cubo imaginario estático $(-0.4, 0, 0.3, 0, 0, 0)$ el cual esta posicionado con la intención de que el robot mantenga una buena posición para cubos futuros.

Para este trabajo se considera un valor de n_{cu} de 2; por lo tanto, si sumamos todas las neuronas involucradas, tendríamos un total de $12 + 6 + (6 * 2) = 30$ para describir una observación del ambiente.

Cabe mencionar que en el caso de usar el búffer de repetición de experiencia HER, además de especificar dicha observación, también se tiene que indicar el elemento objetivo g y el objetivo logrado por el robot g' al final del episodio. En este caso, el objetivo g sería la posición C_{pos} y dirección de corte C_{dir} del primer cubo n_{cu}, y el objetivo logrado g' sería algún punto del sable y la dirección $\vec{S_v}$ del mismo. Para tener un corte más cercano al centro del sable, se tomó en cuenta el punto ubicado a $\frac{3}{4}$ de la longitud del sable. Esta noción se explica mejor en la recompensa del agente.

Acciones

Como ya se ha mencionado en este trabajo, las acciones del agente son multidimensionales discretas acotadas entre -1 a 1, es decir, dado un conjunto de acciones A se tiene que $a \in A; a \in \mathbb{Z}^6; -1 \leq a_1, a_2, ..., a_6 \leq 1$; por ejemplo, una acción a tendría la forma $[-1, 0, 1, 0, 0, 1]$ donde -1 es mover el joystick a la izquierda, 0 es no moverlo y 1 es moverlo a la derecha cierta cantidad. Cabe mencionar que cada elemento a_n le corresponde a cada motor del robot.

Esta forma de interpretar el funcionamiento del agente tiene un inconveniente con respecto a los modelos a utilizar (DDPG, TD3 y HER), ya que no aceptan acciones discretas. En la tabla obtenida de la documentación de *stable-baselines 3* [3], se pueden observar los espacios de acciones que puede manejar cada modelo de interés en este trabajo.

Tabla 5.4: Espacios de acciones soportados para diferentes modelos.

Modelo	Continua ($\mathbb{R}^n$)	Discreta ($\mathbb{N}^1$)	Multi-discreta ($\mathbb{N}^n$)
DQN	X	✓	X
DDPG	✓	X	X
TD3	✓	X	X
PPO	✓	✓	✓

[3] https://stable-baselines3.readthedocs.io/en/master/guide/algos.html

Una primera aproximación para poder controlar los 6 motores fue mediante el algoritmo DQN, pero dado el hecho de que solamente maneja acciones discretas $\mathbb{N}^1$, se tenía que codificar todas las posibles combinaciones en las que se pueden mover los motores, las cuales son $3^6 = 729$, esto resulta impráctico de manejar en código. Por ello, la segunda aproximación fue con el uso de un espacio multi-discreto con dimensionalidad 6, pero el único algoritmo propuesto en este trabajo que soporta este espacio de acciones es PPO, lo que limita mucho los algoritmos que pueden ser utilizados. Entonces, para superar esta dificultad, se propuso una extrapolación al espacio continuo.

La extrapolación mencionada consiste en calcular una acción en el dominio de los reales para luego discretizarla a los valores deseados, o en otras palabras, mapear del espacio continuo al discreto con alguna regla. Entonces, teniendo en cuenta un espacio de acciones continuo donde $a \in A; a \in \mathbb{R}^6; -1 \leq a_1, a_2, ..., a_6 \leq 1$, la regla de mapeo a discreto es la expresada en la ecuación 5.12, donde simplemente se realiza una operación de redondeo con criterio de 0.5.

$$A_{\mathbb{Z}}(a_{\mathbb{R}}) = \lfloor a + 0.5 \rfloor \tag{5.12}$$

Así mismo, se puede apreciar este mapeo en la figura 5.21.

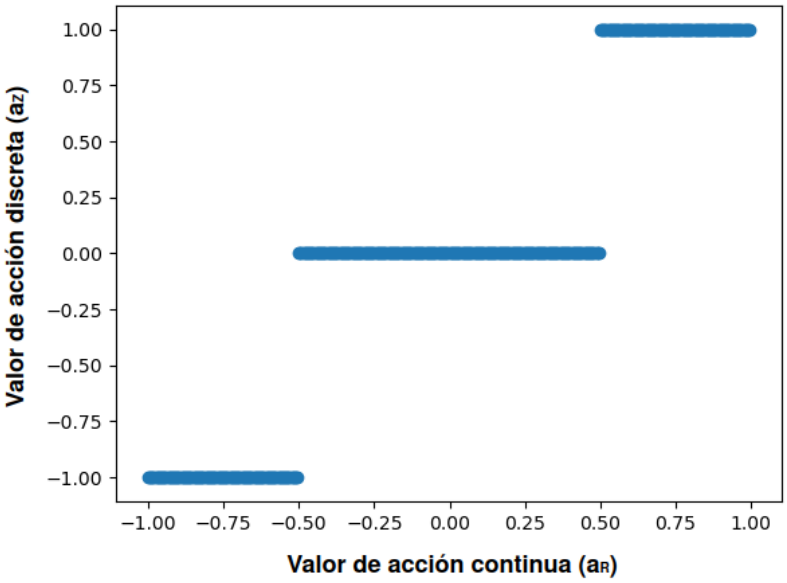


Figura 5.21: Mapeo de acciones continuas a discretas usando redondeo.

Con esto, ya es posible producir las 3 acciones propuestas utilizando algoritmos que producen una salida continua.

Arquitectura de redes neuronales

Ya mencionada la estructura de las observaciones y de las acciones, se puede plantear las arquitecturas de las redes neuronales.

Para los algoritmos TD3 Y DDPG que son basados puramente en el paradigma actor-crítico, se propone la arquitectura de redes neuronales mostradas en la figura 5.22 en donde se plantean las capas de entrada, ocultas y de salida. Nótese que las capas ocultas tienen funciones de activación tipo ReLu.

En el caso de PPO se manejan exactamente las mismas arquitecturas, pero bajo el paradigma de las redes de política $\pi(a|s)$ y de valor $V(s)$. La modificación se muestra en la figura 5.23 donde se muestra la arquitectura para PPO.

Conocer el número de parámetros a ajustar en cada modelo es fundamental para determinar una arquitectura adecuada que pueda ejecutarse correctamente sobre el hardware de la tarjeta JetsonNano. Con esto en mente, un conteo de los parámetros a ajustar por modelo se lleva a cabo:

- DDPG: En el caso de DDPG se manejan $(30*150)+(150*100)+(100*140)+(140*6) = 34340$ de parámetros para el actor y $(36*140)+(140*100)+(100*120)+(120*1) = 31160$ para el crítico. Lo que da un total de 65500 parámetros para ajustar.

- TD3: Para TD3 es casi el mismo número que DDPG, solamente que sumando otros 31160 parámetros debido al doble crítico que se maneja. Por lo tanto, se tendrían 96660 parámetros.

- PPO: El número de parámetros para PPO sería de $(30*150)+(150*100)+(100*140)+(140*6) = 34340$ de parámetros para la política y $(30*140)+(140*100)+(100*120)+(120*1) = 30320$ para la red neuronal de valor. Esto da un resultado de 64660 parámetros.

Con este número de parámetros se garantiza que la tarjeta JetsonNano pueda alcanzar una velocidad de 20 Hz de respuesta para el agente sin saturar la memoria RAM.

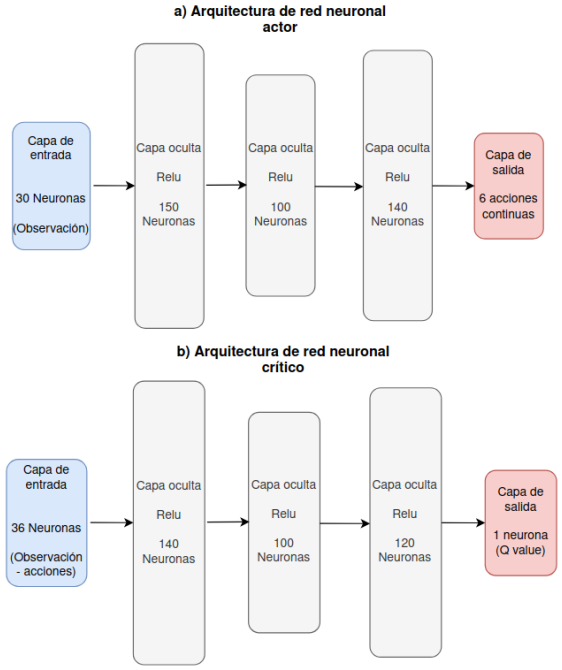


Figura 5.22: Arquitectura de redes neuronales actor y crítico para DDPG y TD3.

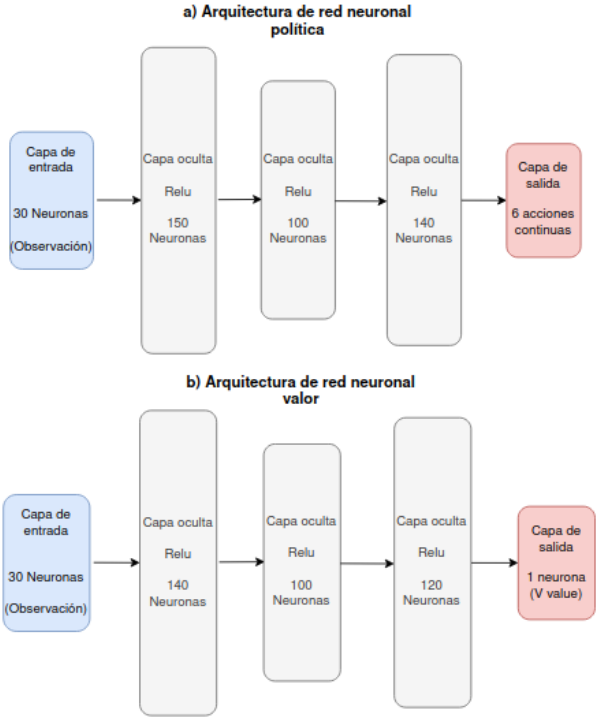


Figura 5.23: Arquitectura de redes neuronales actor y crítico para PPO.

Recompensas

La recompensa del agente se basa en 3 criterios simples, los cuales se describen a continuación:

- **Resultado de corte de cubo**: Se considera el resultado de corte de un cubo como recompensa. Como se ha mencionado, el cubo tiene 3 resultados diferentes: fallido, golpeado y cortado; por ello, en la tabla 5.5 se muestra la ponderación de cada resultado en la recompensa.

Tabla 5.5: Recompensa por resultado de corte de cubo.

Estatus	Recompensa
Fallido (-1 - Rojo)	-0.4
Golpeado (1 - Verde)	4
Cortado (2 - Azul)	7

- **Distancia euclidiana tridimensional**: La técnica más usada para que brazos robóticos puedan alcanzar objetivos específicos es el cálculo de la distancia euclidiana entre el efector final del brazo y el objetivo. Entre menor sea la distancia, mayor es la recompensa dada. En el caso de este trabajo, la distancia se mide con respecto al punto ubicado a $\frac{3}{4}$ de la longitud del sable $P_{\frac{3}{4}sable}$ y el cubo más cercano al brazo robótico $Cpos$. En las ecuaciones 5.13 y 5.14 se muestra el cálculo de esta recompensa.

$$d = \sqrt{(P_{\frac{3}{4}sable_x} - C_{pos_x})^2 + (P_{\frac{3}{4}sable_y} - C_{pos_y})^2 + (P_{\frac{3}{4}sable_z} - C_{pos_z})^2} \tag{5.13}$$

$$r(d) = \begin{cases} 0.55 - d, & \text{si } d < 1 \\ -0.45, & \text{otro caso} \end{cases} \tag{5.14}$$

- **Distancia euclidiana bidimensional**: Muy similar a la distancia tridimensional, pero esta se mide solamente con respecto a un plano. En el entorno creado de cubos, estos se mueven a lo largo del eje x; por lo tanto, la distancia se mide con respecto al plano ZY. Las ecuaciones 5.15 y 5.16 definen la recompensa.

$$d = \sqrt{(P_{\frac{3}{4}sable_y} - C_{pos_y})^2 + (P_{\frac{3}{4}sable_z} - C_{pos_z})^2} \tag{5.15}$$

$$r(d) = \begin{cases} 0.2 - d, & \text{si } d < 1 \\ -0.8, & \text{otro caso} \end{cases} \tag{5.16}$$

  La razón de la creación de este criterio de recompensa es incitar al agente a que posicione rápidamente el sable enfrente del cubo que será cortado. Con esta recompensa, el robot mostró más precisión de corte.

- **Similitud del coseno**: Muy similar a la expresión usada para detectar el corte de un cubo mostrada en la ecuación 5.11, este criterio de recompensa utiliza la similitud del coseno para incitar al agente a que corte el cubo en la dirección correcta. Entonces, considerando a $cos(\theta)$ como la similitud entre vectores, la recompensa queda expresada en la ecuación 5.17.

$$r(d) = \begin{cases} 0.5 * cos(\theta), & \text{si } cos(\theta) > 0 \\ 0, & \text{si } cos(\theta) \leq 0 \end{cases} \tag{5.17}$$

  Cabe mencionar que la recompensa de similitud solo es activada cuando la distancia tridimensional es menor a un valor umbral. Esto evita que el robot quiera tener el sentido de corte del cubo cuando ni siquiera está cerca de él.

Las recompensas planteadas anteriormente se pueden ver graficadas en la figura 5.24.

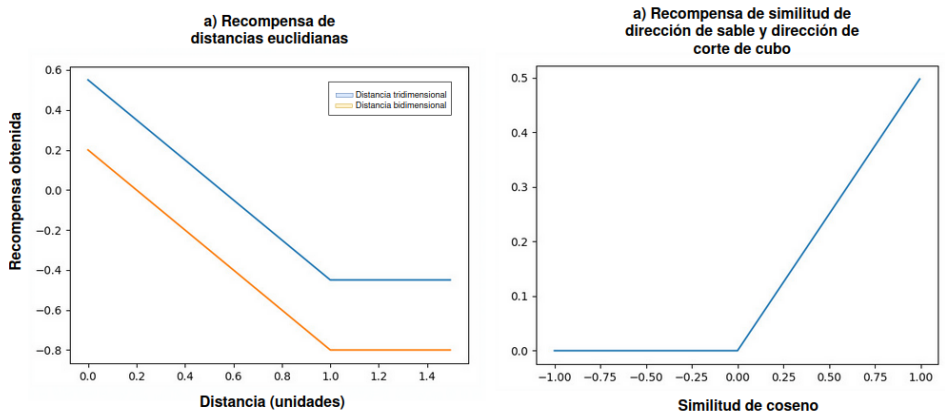


Figura 5.24: Funciones de recompensa.

Para el caso del uso del búffer HER, en el trabajo de presentación del algoritmo [2], se menciona que usando recompensas escasas se tiene un mejor rendimiento. Por ello, en este trabajo cuando se entrenan modelos implementando HER, se utiliza una versión discretizada de las recompensas para acercarse más a la concepción de recompensas escasas. En la figura 5.25 se puede ver la discretización de las recompensas.

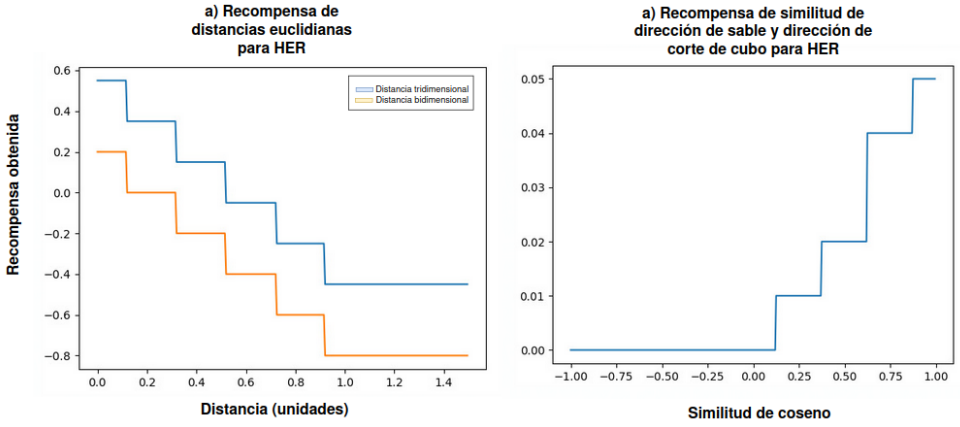


Figura 5.25: Funciones de recompensa para algoritmos con búffer HER.

5.3.4. Entrenamiento paralelo de agentes

Con la arquitectura y el agente de aprendizaje por refuerzo profundo explicados, se puede comprender el proceso de entrenamiento del brazo robótico para que aprenda del ambiente simplificado de *Beat Saber*.

El hardware utilizado para llevar a cabo el entrenamiento se describe en la tabla 5.6. Los aspectos de hardware son considerados como de rendimiento medio, lo que brinda una potencia computacional suficiente para entrenar múltiples agentes.

Tabla 5.6: Especificaciones de hardware para entrenamiento de agentes.

Componente	Descripción
GPU	GTX 1660 Super 6GB 192-bit GDDR6
CPU	AMD Ryzen 5 3400G 3.70GHz, Quad-Core, 4MB L3
RAM	2 x 8GB DDR4, 3200MHz
Tarjeta Madre	micro B450 AORUS

Para llevar a cabo el entrenamiento se tuvieron que tomar en cuenta las siguientes consideraciones:

- **Cantidad de agentes**: El número de agentes que se pueden entrenar paralelamente está restringido en mayor medida por la memoria de los componentes de hardware; es decir, tanto la GPU como la RAM deben satisfacer la demanda de los n agentes. Como consecuencia, y tomando en cuenta los recursos disponibles, se pudieron inicializar 3 agentes en total. Si se iniciaba un cuarto agente, la computadora corría el riesgo de reiniciarse.

- **Aceleración de simulación**: En Gazebo, cuando se quiere acelerar la simulación, se deben sintonizar los parámetros de paso de tiempo dt y el tiempo objetivo de aceleración t_{acc}. La sintonización es necesaria debido a que el tiempo de aceleración resultante no siempre es exacto, ya que este suele disminuir cuando el equipo está sobre un uso demandante; de hecho, entre más agentes se simulan, el tiempo de aceleración tiende a disminuir. Entonces, experimentando con los 3 agentes, se logró llegar a una configuración de $dt = 0.038$ y $t_{acc} = 4$ para lograr una aceleración de x3 de manera estable.

- **Cantidad de generaciones y longitud**: El tiempo deseado para cada generación se decidió que sea aproximadamente de 50 minutos, ya que en la arquitectura de un solo agente, se observó que este era el tiempo aproximado para que un agente se volviera estable o inestable si iniciaba su entrenamiento desde cero. Entonces, tomando en cuenta la frecuencia del agente de 20 Hz, se tendría un aproximado de 60,000 pasos por generación. Además, se determinó que el total de generaciones fueran 5, ya que en la mayoría de los modelos generados después de la generación 4 ya no había una mejora significativa en el rendimiento.

- **Hiperparámetros de modelos**: Los modelos utilizados en este trabajo se dividen en dos tipos *on-policy* (PPO) y *off-policy* (TD3 y DDPG). Estos manejan parámetros en común, pero sí presentan algunas diferencias.

  Para el algoritmo PPO se maneja tasa de aprendizaje $\alpha = 0.0003$, factor de descuento $\gamma = 0.99$, factor lambda $\lambda = 0.95$, pasos para actualización $n_{steps} = 2048$, tamaño de lote $batch\_size = 128$, rango de recorte $clip = 0.2$, valor máximo de recorte de 0.5 y coeficiente de funciones de valor $vf = 0.5$.

  Los algoritmos DDPG y TD3 manejan parámetros muy similares. Entonces, para estos dos se tiene: tasa de aprendizaje $\alpha = 0.001$, factor de descuento $\gamma = 0.99$, coeficiente de actualización suavizada $\tau = 0.005$, pasos para iniciar aprendizaje $learning\_start = 45,000$, tamaño de búffer de repetición de experiencia $buffer\_size = 70,000$, tamaño de lote $batch\_size = 150$, ruido de política objetivo de 0.2 y límite de ruido de política objetivo de 0.5.

- **Posición inicial de brazo robótico**: Se desea que el brazo robótico siempre inicie una canción en *Beat Saber* en la posición (0,0,0,0,0,0), no obstante, si el robot sostiene el mando en dicha configuración, el sable quedaría señalando a la izquierda y lo ideal es que el sable esté apuntando hacia enfrente, ya que así es más fácil cortar cubos. Por lo que se optó a girar el robot 90° a la derecha como se muestra en la figura 5.26 para no sacrificar la configuración de 0's. Con esta posición se logró que el robot encontrara una política óptima más rápido.

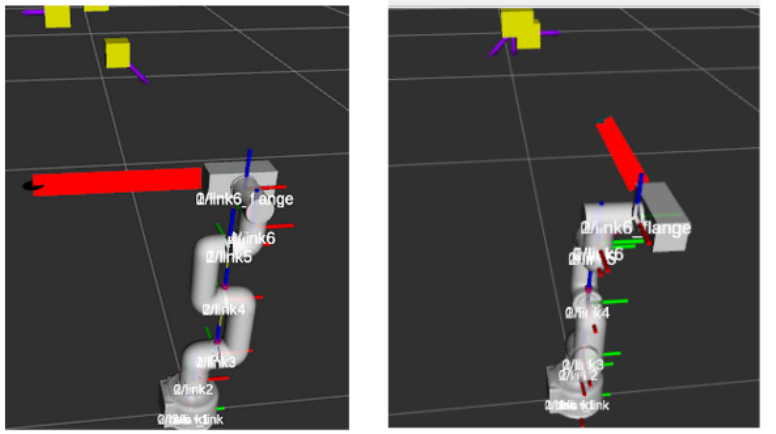


Figura 5.26: Diferentes posiciones iniciales del robot.

Con todo lo anterior establecido, se procedió a realizar el entrenamiento de los modelos. En la figura 5.27 se muestran datos de cada agente en tiempo de entrenamiento y en la figura 5.28 se aprecia el entrenamiento de los 3 agentes propuestos.

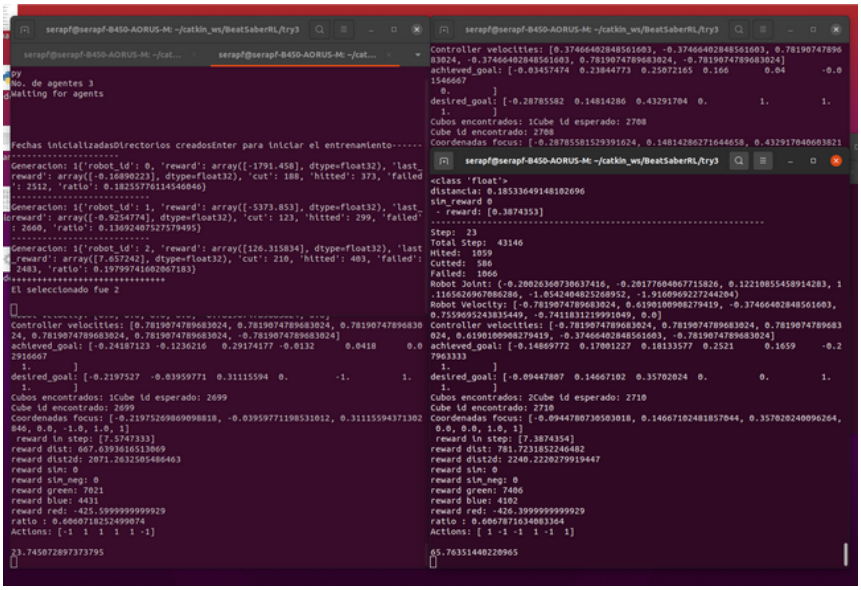


Figura 5.27: Información de los agentes entrenando.

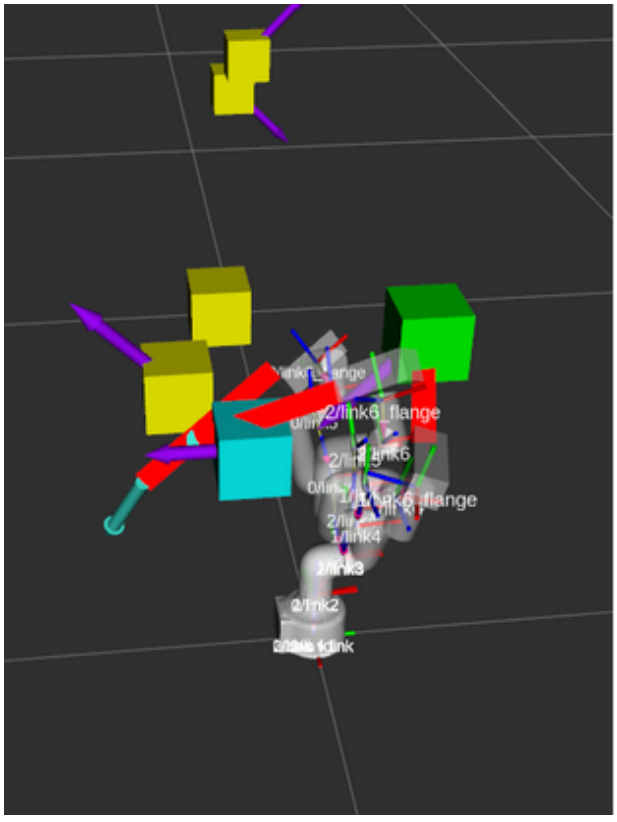


Figura 5.28: Tres robots mycobot entrenando la dinámica de *Beat Saber*.

Caracterización de modelos entrenados

A continuación se muestra el resultado de entrenamiento de cada modelo. Para cada uno se presenta información como: número de modelo seleccionado en cada generación (selec), la recompensa acumulada total, número de cubos fallidos, golpeados, cortados, el porcentaje de cubos golpeados y cortados y el porcentaje solamente tomando en cuenta los cubos cortados. Además, se proporcionan visiones gráficas de la evolución de la recompensa a través de las generaciones y de las funciones de pérdida de cada modelo. Para fines de simplicidad de visualización, solo se muestra el mejor modelo de cada generación.

DDPG

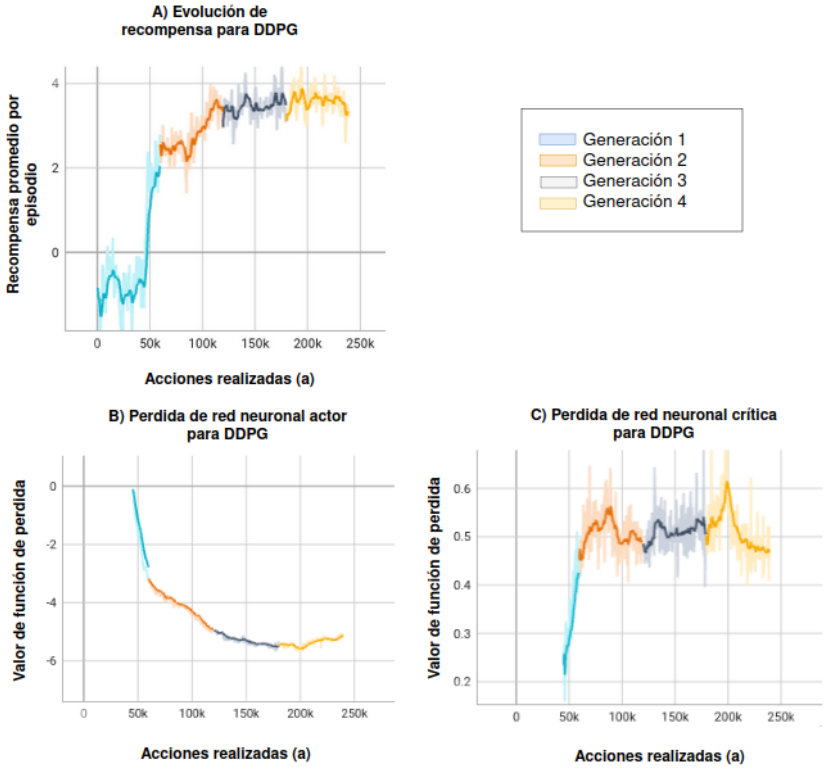


Figura 5.29: Gráficas de recompensa y de pérdida de DDPG en entrenamiento.

Tabla 5.7: Resultados de entrenamiento de modelo DDPG.

Gen	Selec	Recompensa Acumulada	Cubos Fallados (F)	Cubos Golpeados (G)	Cubos Cortados (C)	% G+C	% C
1	0	-145.688	2376	415	243	21.68	8.00
2	1	10324.68	1850	1027	770	49.27	21.11
3	1	12561.78	1430	1225	969	60.50	26.73
4	1	13052.77	1362	1225	1068	62.73	29.22

DDPG+HER

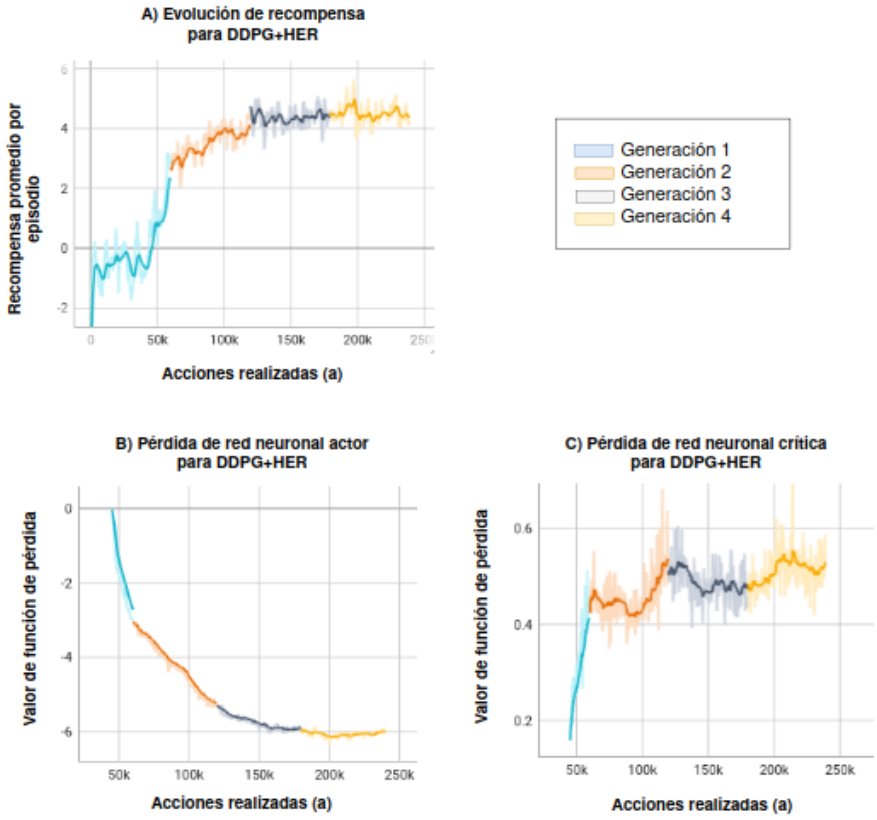


Figura 5.30: Gráficas de recompensa y de pérdida de DDPG+HER en entrenamiento.

Tabla 5.8: Resultados de entrenamiento de modelo DDPG+HER.

Gen	Selec	Recompensa Acumulada	Cubos Fallados (F)	Cubos Golpeados (G)	Cubos Cortados (C)	% G+C	% C
1	1	424.34	2456	386	206	19.42	6.75
2	0	12788.46	1778	1033	775	50.41	21.61
3	2	15709.60	1378	1188	1042	61.80	28.88
4	0	16149.34	1310	1247	1038	63.56	28.88

TD3

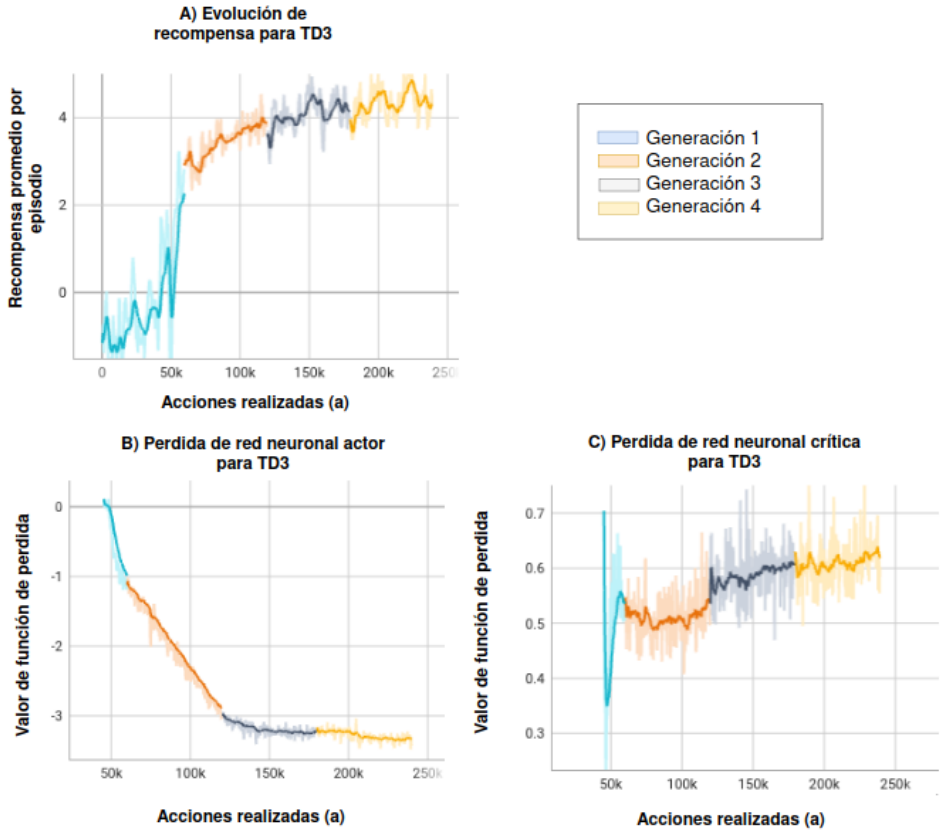


Figura 5.31: Gráficas de recompensa y de pérdida de TD3 en entrenamiento.

Tabla 5.9: Resultados de entrenamiento de modelo TD3.

Gen	Selec	Recompensa Acumulada	Cubos Fallados (F)	Cubos Golpeados (G)	Cubos Cortados (C)	% G+C	% C
1	1	53.22	2395	386	237	20.64	7.85
2	2	12625.74	1539	1197	879	57.42	24.31
3	1	14603.539	1232	1274	1092	65.75	30.32
4	2	15773.74	1150	1228	1244	69.38	34.34

TD3+HER

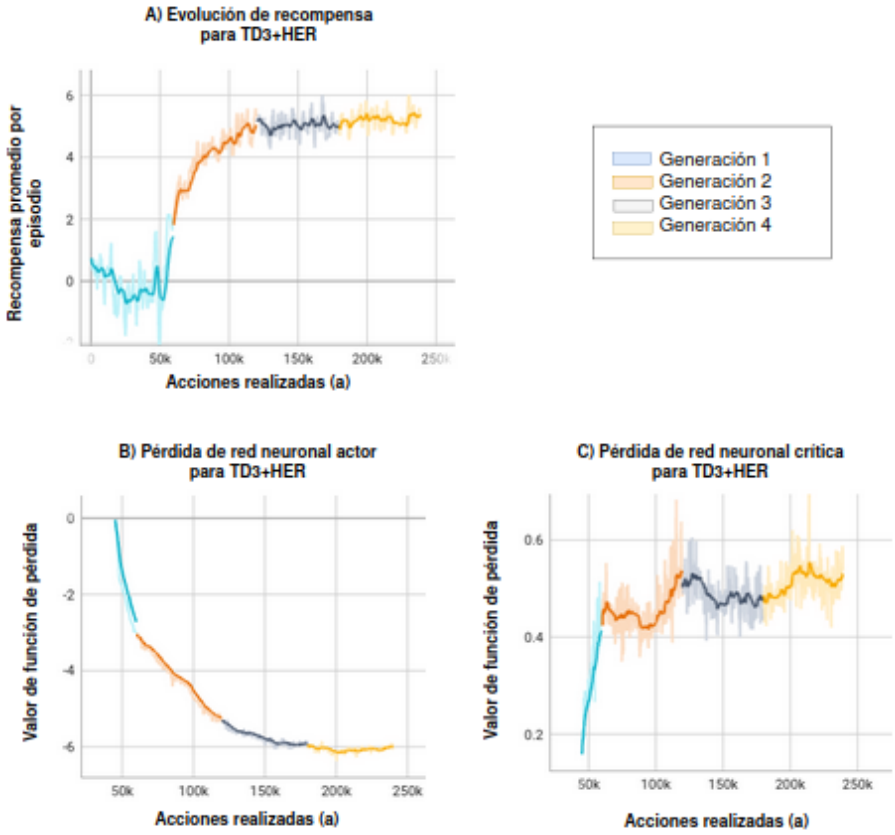


Figura 5.32: Gráficas de recompensa y de pérdida de TD3+HER en entrenamiento.

Tabla 5.10: Resultados de entrenamiento de modelo TD3+HER.

Gen	Selec	Recompensa Acumulada	Cubos Fallados (F)	Cubos Golpeados (G)	Cubos Cortados (C)	% G+C	% C
1	1	144.38	2332	356	186	18.85	6.4
2	0	14593.20	1442	1277	752	58.45	21.61
3	2	17415.66	1052	1376	1011	69.67	29.39
4	0	18123.15	982	1275	1208	71.65	34.86

PPO

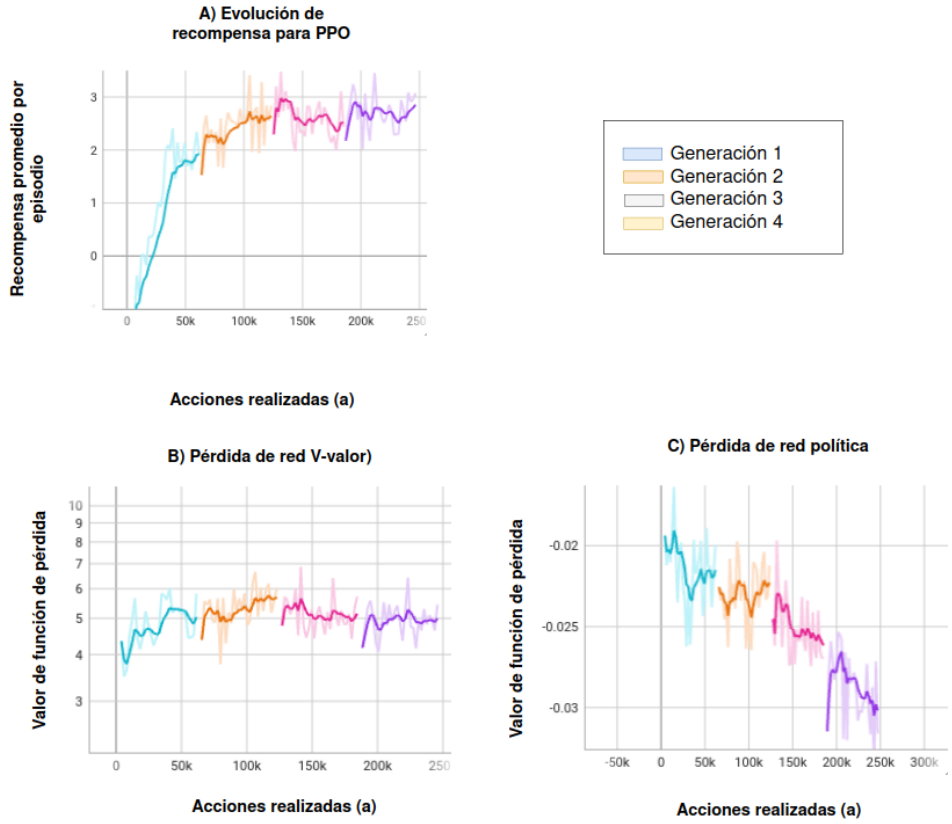


Figura 5.33: Gráficas de recompensa y de pérdida de PPO en entrenamiento.

Tabla 5.11: Resultados de entrenamiento de modelo PPO.

Gen	Selec	Recompensa Acumulada	Cubos Fallados (F)	Cubos Golpeados (G)	Cubos Cortados (C)	% G+C	% C
1	1	5755.23	2446	1217	670	43.54	15.46
2	1	13074.693	1383	1871	1081	68.09	24.94
3	1	14624.409	1346	1865	1195	70.40	27.12
4	2	15223.203	1250	1901	1220	71.40	27.91

Después del entrenamiento, los modelos TD3, DDPG y PPO mostraron resultados satisfactorios. A continuación se mencionan brevemente aspectos para dar una mejor caracterización de los modelos entrenados.

Rendimiento

El rendimiento de cada modelo se determina tomando en cuenta el porcentaje de cubos cortados (% C) y el porcentaje de cubos golpeados y cortados (% G+C). Con ello, se observa que el que mejor resultado obtuvo fue TD3 en especial su variante TD3+HER, teniendo un % G+C de 71 % y 34.4 % de % C. Además de tener una recompensa acumulada de 18123, la cual es mayor a su análogo DDPG+HER con 16149.

Los modelos basados en TD3 y DDPG, presentaron un comportamiento similar, el cual consistía en mantener el sable verticalmente apuntando hacia abajo y moverlo a la derecha e izquierda cada vez que se aproximaba un cubo. En la figura 5.34 se aprecia este comportamiento.

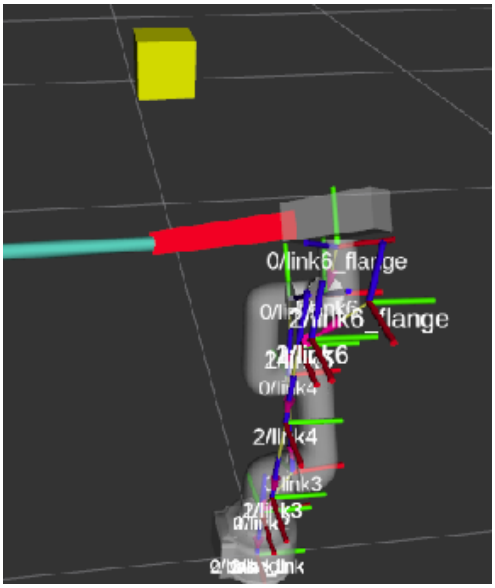


Figura 5.34: Comportamiento por políticas TD3 y DDPG

En el caso del modelo basado en política PPO obtuvo un desempeño de % G+C de 71.40 %, un % C de 27.91 % y una recompensa acumulada de 15223.203. La política presentó un comportamiento sin mucho movimiento, que se centraba en mantener el sable en una posición horizontal y moverlo arriba y abajo cada vez que un cubo se acercaba. Esto se ve representado en la figura 5.35.

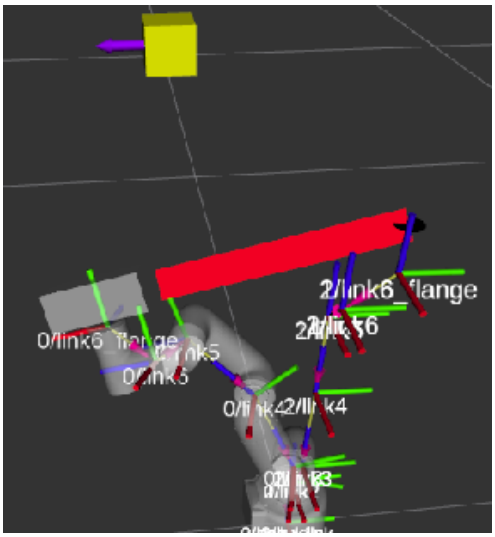


Figura 5.35: Comportamiento por política PPO.

Convergencia

Todos los modelos muestran una convergencia eficiente con el entrenamiento propuesto. La ganancia de la recompensa se mantiene relativamente constante y los valores de las funciones de pérdida no se disparan.

Además, el tiempo para que cada modelo convergiera también fue eficiente. Para los modelos *off-policy* TD3 y DDPG fueron necesarias aproximadamente 3 horas para llegar a la generación 4, donde se muestra una convergencia estable. Por su parte, PPO logra la convergencia más rápida de todos los modelos, dado que llega a la generación 4 en solo 2 horas de entrenamiento. Esto se debe a que los algoritmos off-policy necesitan de una pausa para extraer elementos de la memoria de repetición de experiencia y aprender de ellos, lo que implica una pérdida de aproximadamente medio segundo por cada episodio, mismos en los que la simulación se detiene; de manera contraria, los algoritmos *on-policy* aprenden conforme se van ejecutando las acciones sin interrupciones entre episodios.

Lo anterior demuestra que los algoritmos pueden aprender del ambiente propuesto resultando en una convergencia; cada uno con su propia capacidad de aprendizaje. Dicho esto, el algoritmo que demostró una mayor estabilidad en la recompensa y tiempo de convergencia con respecto a la cantidad de acciones realizadas fue el algoritmo TD3. Su variante TD3+HER alentó el proceso de convergencia pero logró un rendimiento mejor.

Funciones de pérdida

En el análisis de la evolución del entrenamiento fue de vital importancia ir analizando las gráficas de valores de las funciones de pérdida de las redes neuronales involucradas en cada algoritmo, ya que nos puede expresar la estabilidad del modelo. De esta manera, todos los modelos logran tener una convergencia en sus funciones de pérdida.

Para contrastar lo anterior, en la imagen 5.36 se muestra un modelo que no logra tener una convergencia en sus funciones de pérdida. Cabe mencionar que esta inestabilidad no se ve reflejada directamente en la recompensa, ya que esta puede aumentar, disminuir o estancarse intermitentemente, pero nunca alcanzando resultados satisfactorios de rendimiento.

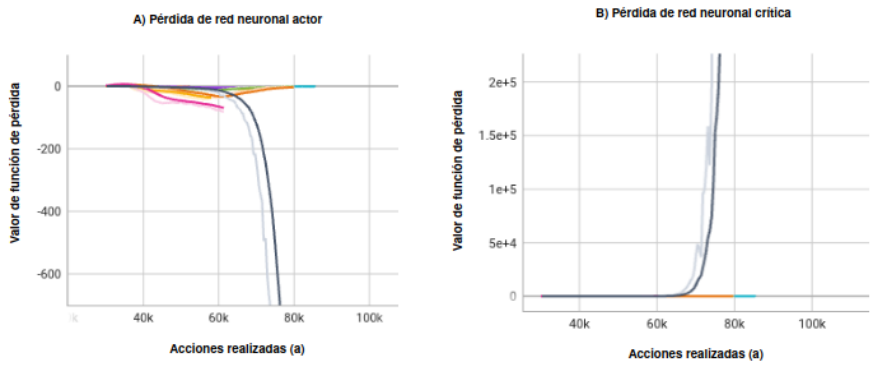


Figura 5.36: Gráficas de pérdidas de modelo actor-crítico inestable (gris) con respecto a otros estables

Con esto en mente, cuando el modelo funciona adecuadamente, hay una gran correlación entre el movimiento de la recompensa con las funciones de pérdida. En el caso de los modelos actor-crítico, en la gráfica de pérdida del actor se puede observar que la convergencia de la función de pérdida se da al mismo tiempo que el de la recompensa. Además,

si el valor de la pérdida del actor se acerca a 0, el modelo tendrá un mejor rendimiento. Esto se ve mostrado si se comparan los valores de la gráfica de TD3 5.31B) con los de DDPG 5.29B); se observa que el valor convergente de TD3 es -3 mientras que en DDPG es -5.8 y como consecuencia, TD3 tiene un rendimiento mayor. Esto se puede explicar si se recuerda que TD3 mejora el problema de sobre-estimamiento de la recompensa en DDPG.

El modelo PPO también alcanzó una buena convergencia en su función de recompensa y su función de pérdida de V-valor.

Tomando en consideración todo lo anterior, se puede decir que el modelo con mejores resultados fue TD3 en la fase de entrenamiento, tanto en su versión normal como en su variante con HER, seguido de PPO y al último DDPG.

5.4. Visión computacional usando YOLO

Con los modelos entrenados, se puede hacer una evaluación de los mismos ya interactuando con el entorno *Beat Saber* virtualmente; para ello, antes se describirá el entrenamiento de YOLO v7 Tiny para identificar los cubos y un proceso para identificar la dirección de corte del cubo.

La forma más convencional de entrenar un modelo YOLO, es conseguir imágenes de algún medio para luego manualmente delimitar los objetos de interés, marcándolos con un rectángulo y especificando la clase a la que pertenecen. Para este trabajo simplemente se recurrió a tomar capturas de pantalla de vídeos presentes en internet que muestran la jugabilidad del entorno; de esta manera, se tomaron en cuenta vídeos donde la iluminación, efectos especiales, colores de cubos y de sables varían. En la figura 5.37 se puede observar el proceso de etiquetado con el programa labelImg [4]; como se puede observar, sólo hay una clase existente que es el cubo. Así mismo, en al tabla 5.12 se puede ver un resumen de las imágenes y cubos clasificados para el entrenamiento de YOLO.

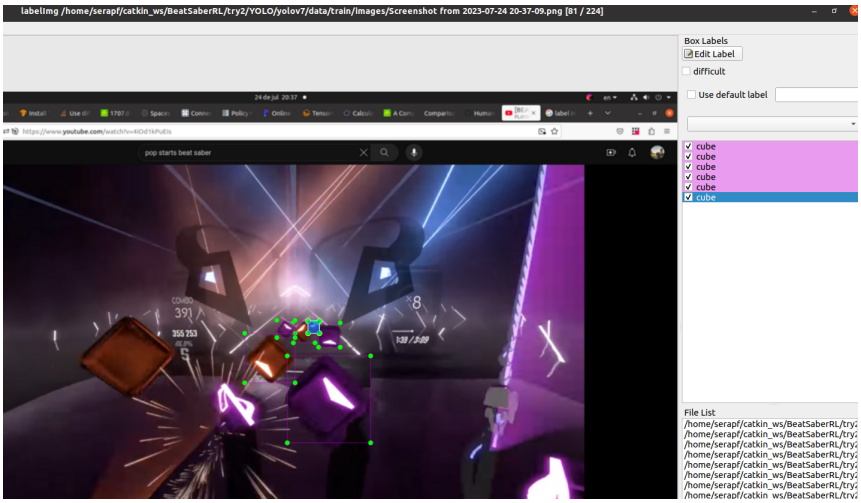


Figura 5.37: Etiquetado de cubos para su entrenamiento en YOLO.

[4] https://github.com/HumanSignal/labelImg

Tabla 5.12: Conjunto de datos para el entrenamiento de YOLO.

Conjunto de datos	Imágenes	Cubos
Entrenamiento	225	936
Validación	35	172

Después de un entrenamiento de 300 épocas, se logró tener una precisión del modelo muy buena, ya que se obtuvo un valor de mAP mayor al 85 %, lo que indica que el modelo encierra muy bien a los cubos en la imagen. En la figura 5.38 se pueden ver los resultados de entrenamiento de YOLO.

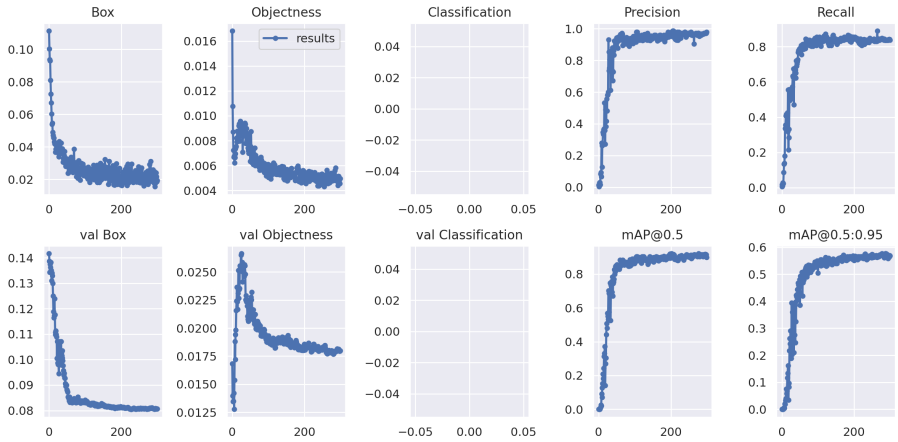


Figura 5.38: Estadísticas de precisión de YOLO.

Con respecto a la detección del ángulo de corte, se intentó que YOLO pudiera identificar la orientación del cubo especificando más clases tales como: clase cubo 0, 45, 90, 135, 180, 225, 270, 315 y sin dirección; de esta manera serían 9 clases. No obstante, a la hora de entrenar el modelo no podía converger ya que no podía identificar cubos y ni asignar un ángulo, por lo que se concluyó que YOLO no puede ser utilizado para clasificar un objeto dependiendo de su orientación.

Dando el inconveniente, se recurrió a otros algoritmos de visión computacional para identificar la orientación. El proceso que se siguió para la identificación del ángulo de corte se describe a continuación:

1. El cubo es debidamente delimitado por YOLO, se extrae de la imagen, se convierte a una escala de grises y se lleva a cabo una binarización de la imagen, es decir, convertirla a un conjunto de 1 y 0 tomando como criterio el valor de cada píxel y un valor umbral; si el valor del píxel es mayor al valor de umbral mencionado, se convierte en 1, si no, en 0.

2. La estructura del cubo es sumamente ventajosa, debido a que la dirección es marcada con una área en blanco y es la más grande del cubo; por lo tanto, se utiliza una detección de contorno para identificar dicha área grande. Esto es un paso intermedio que sirve para descartar cualquier otra impureza pequeña que puede encontrarse en el cubo, ya que solamente se toma en cuenta el contorno con mayor área.

3. Con el área identificada, se calcula el centro de la misma con respecto a toda la imagen del cubo. El cálculo de este punto es simplemente la suma de las coordenadas X y Y por separado y su división entre la cantidad de píxeles dentro del contorno. Esto se puede ver expresado en la ecuación 5.18, donde X_c y Y_c son el centro del contorno, x_n y y_n son las coordenadas del n-ésimo píxel pertenecientes al contorno y N es la cantidad de píxeles totales en el contorno.

$$X_c = \frac{1}{N}\sum_{n=0}^{N} x_n, Y_c = \frac{1}{N}\sum_{n=0}^{N} y_n \tag{5.18}$$

4. Con el punto central de la flecha del cubo identificado, se utilizó el ángulo entre dos vectores $cos(\theta)$. Para identificar el ángulo se trazan dos vectores, el primero es un vector estático $(1, 0)$ y el segundo es un vector que va desde el centro de la imagen hasta el punto centro del contorno; entonces, dependiendo del ángulo θ expresado en grados, se mapea a una clase en específico. En la ecuación 5.19 se aprecia una expresión que mapea el ángulo a un índice que indica la clase de cubo identificado. Esta clasificación se puede observar en la figura 5.39.

   Adicionalmente, si la magnitud del vector que va desde el centro de la imagen al punto centro del contorno es muy pequeña, quiere decir que se trata de un cubo punto, es decir, sin dirección.

$$i(\theta) = \lfloor \frac{1}{45}(\theta + 180) + \frac{1}{2} \rfloor \%8 \tag{5.19}$$

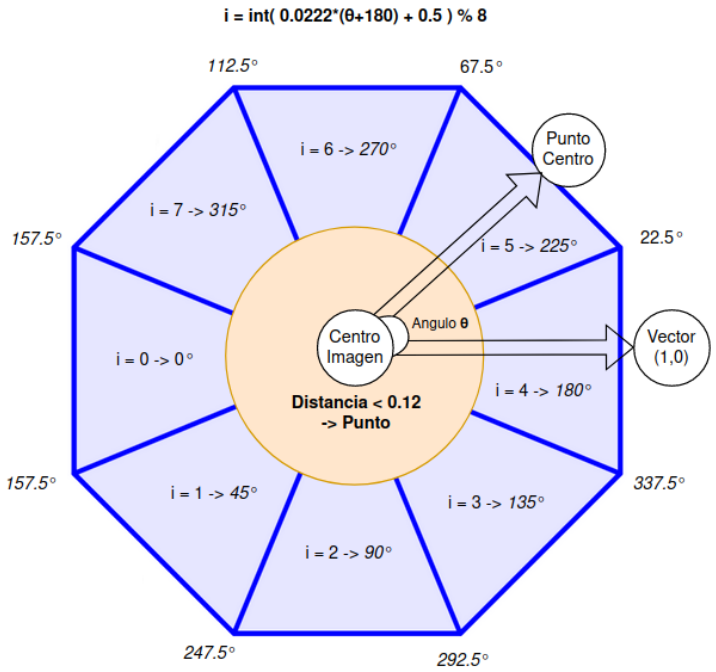


Figura 5.39: Diagrama para clasificación de cubos.

Todo el proceso descrito con anterioridad se puede ver ilustrado en la figura 5.40.

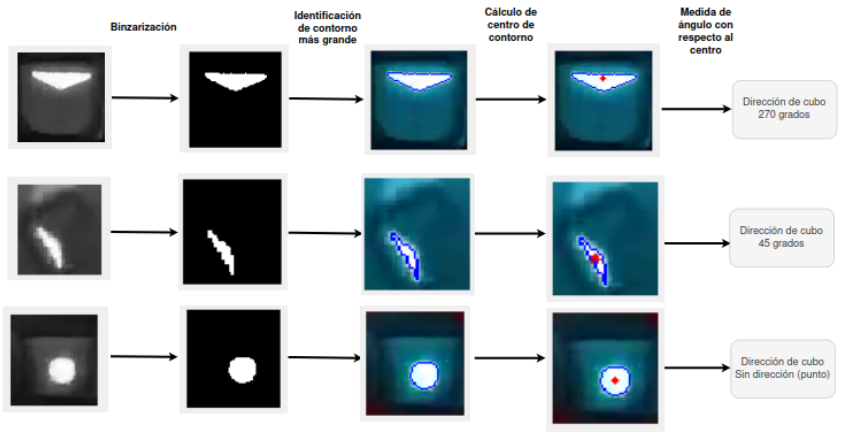


Figura 5.40: Pasos para identificar el ángulo en diferentes tipos de cubo.

Proyección tridimensional

Con esto, ya se tiene un sistema de visión computacional capaz de identificar cubos y de detectar la dirección de corte, no obstante, para terminar de implementar la visión del agente se tiene que idear una forma de mapear los cubos detectados en la imagen bidimensional a un espacio tridimensional. Para lograr lo anterior, se tomó en cuenta el modelo de cámara estenopeica (*pinhole camera*).

En la computación gráfica se utiliza el modelo de cámara estenopeica para proyectar puntos de un objeto tridimensional a una imagen bidimensional para poder mostrarla en pantalla. En el trabajo [84] se muestra una expresión que modela dicha proyección, la cual está estableciendo que la cámara no muestra rotación en ningún eje. Se puede obtener la expresión denotada en la ecuación 5.21. Donde (u,v) son los puntos de la imagen, (X,Y,Z) son los puntos tridimensionales y S es un factor de escala. La matriz A mostrada en la ecuación 5.20 se conoce como intrínseca, donde α y β son los factores de escala en la imagen en el eje u y v respectivamente, (u_0, v_0) es el origen de coordenadas en la imagen y γ es la asimetría de los ejes en la imagen, por lo general este último siempre es 0.

$$A = \begin{pmatrix} \alpha & \gamma & u_0 \\ 0 & \beta & v_o \\ 0 & 0 & 1 \end{pmatrix} \tag{5.20}$$

$$s \begin{pmatrix} u \\ v \\ 1 \end{pmatrix} = A \begin{pmatrix} X \\ Y \\ Z \end{pmatrix} \tag{5.21}$$

De este modo, si se obtiene la matriz inversa intrínseca (A^{-1}) podemos obtener un modelo que exprese en términos de un punto bidimensional otro tridimensional, como se muestra en la ecuación 5.22 ,dado que $\gamma = 0$.

$$\begin{pmatrix} X \\ Y \\ Z \end{pmatrix} = \begin{pmatrix} \frac{s}{\alpha}(u - u_0) \\ \frac{s}{\beta}(v - v_0) \\ s \end{pmatrix} \tag{5.22}$$

En este trabajo se utilizó como valores de parámetros los siguientes: $\alpha = 0.9$,$\beta = 0.7$, $u_0 = 0.5$ y $v_0 = 0.5$.

Como se puede ver en la ecuación 5.22 la profundidad, que es representada por el eje Z, es igual al factor de escalamiento S; como consecuencia, el factor S representa la profundidad que tiene el objeto en la imagen, por lo que se vuelve necesaria una forma de calcular la profundidad de los cubos en la imagen del entorno.

Para hacer esta estimación se aprovecha el hecho de que la posición de la cámara dentro del juego es estática y los cubos se aproximan a ella; de este modo, se toma en cuenta la caja delimitadora del cubo detectado por YOLO, de ella se determina el lado más largo y, en función de ella, se hace un aproximado de qué tan lejos está el cubo de la cámara.

Con esta forma de identificación de profundidad, con el modelo de cámara estenopeica y el modelo YOLO, se tuvo como resultado un sistema que permitiera proyectar los cubos de la imagen producida por *Beat Saber* a cubos con posición tridimensional y dirección de corte. El resultado se puede observar en la figura 5.41.

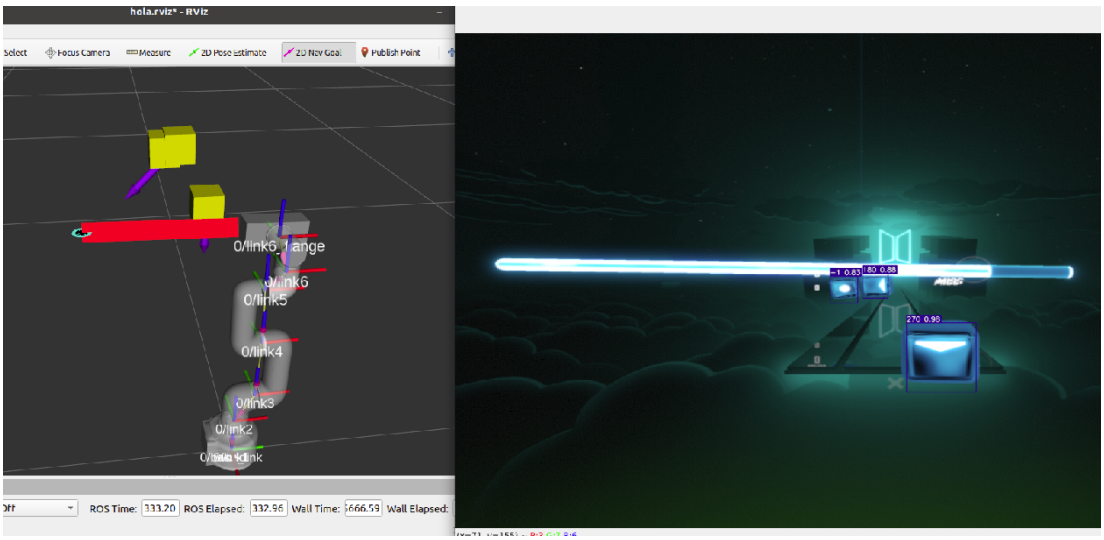


Figura 5.41: Sistema de visión computacional para identificación de cubos.

5.5. Evaluación en entorno virtual

Con los agentes entrenados y una visión computacional desarrollada, es posible hacer pruebas donde el robot virtual pueda desempeñarse en una canción de *Beat Saber*. Para ello, se necesitó hacer una conexión con el motor de juego real. En la arquitectura mostrada en la figura 5.42 se aprecian los nuevos nodos para lograr dicha conexión.

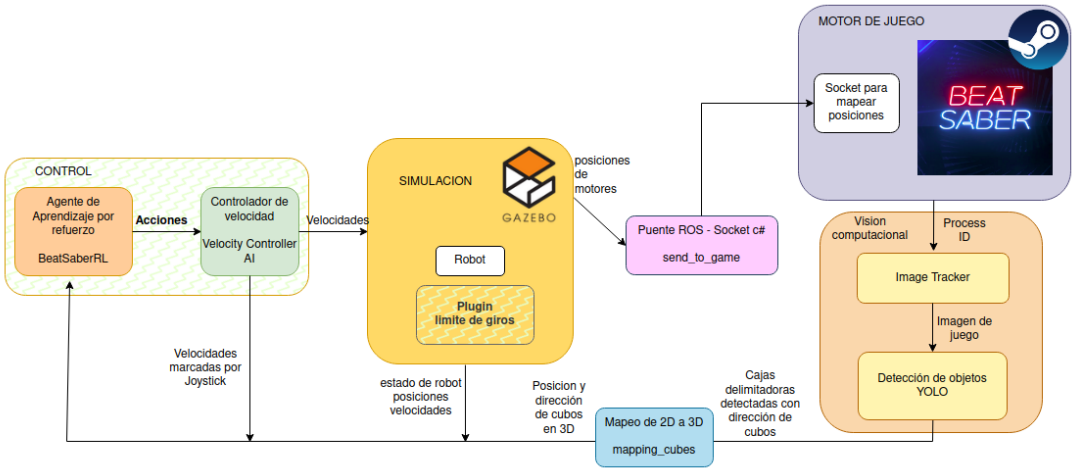


Figura 5.42: Arquitectura interacción robot simulado con *Beat Saber*.

A continuación se brinda un breve resumen de cada nodo en la arquitectura.

send_to_game

Para entender el fin del nodo, se debe tener en cuenta que el lenguaje en el que está escrito *Beat Saber* es C#, el cual no tiene un soporte directo en ROS, por lo que se necesita una comunicación intermediaria.

Establecer esta comunicación es la tarea principal de este nodo. Para lograrlo, primero se inyectó código en el juego que permitiera conectarse a él mediante un *socket* convencional de C# para poder manipular la posición del sable. Con esto establecido, el nodo *send_to_game* también activa un *socket* convencional de Python para conectarse al *socket* de C#. Con la conexión realizada, el mismo nodo se suscribe al tópico de las posiciones de las articulaciones del robot y, con base en eso, se envía la posición y rotación del sable del robot al juego.

image Tracker

La función del nodo es simple, solamente extrae la imagen directamente del juego basándose en el ID del proceso que tiene la ventana que lo renderiza.

YOLO

Recibe la imagen extraída por *image Tracker* y es procesada para obtener la posición de los cubos en la imagen y la dirección de corte. Este nodo no lleva a cabo la proyección tridimensional de los cubos.

mapping cubes

Recibe la información de los cubos detectados por YOLO e implementa el modelo de cámara estenopeica para poder proyectarlos tridimensionalmente.

Con las modificaciones a la arquitectura, ya es posible realizar pruebas virtuales con cada modelo entrenado. En la figura 5.43 se puede observar cómo el robot simulado interactúa con el ambiente virtual.

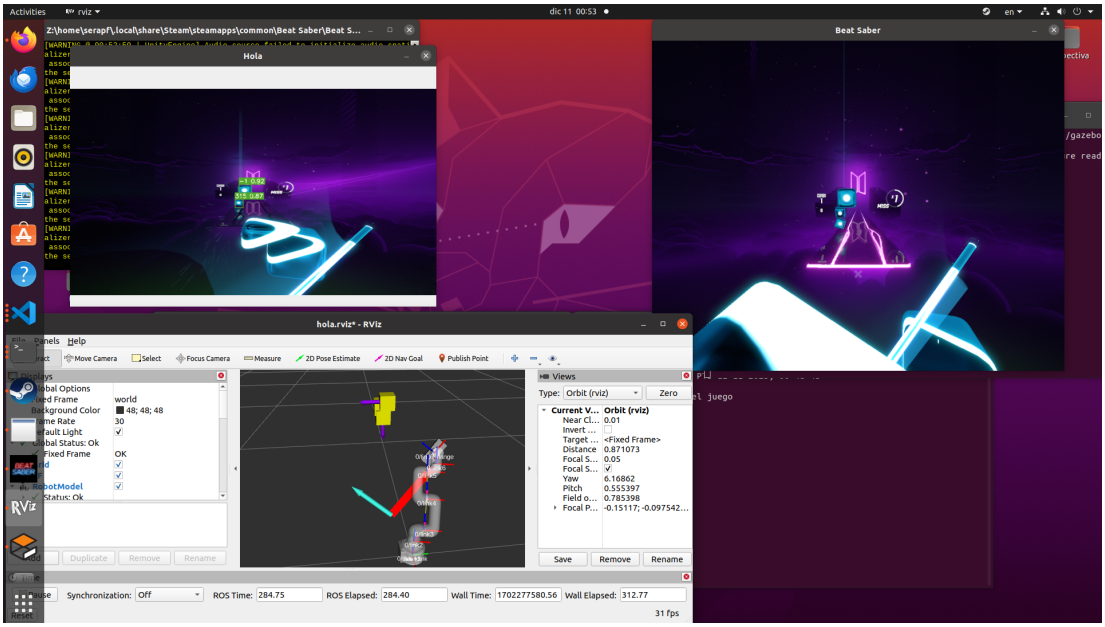


Figura 5.43: Interacción de robot simulado con *Beat Saber*.

Entonces, las pruebas a realizar tienen en consideración los siguientes aspectos:

- **Dificultades**: Cada modelo será puesto a prueba para que juegue una canción en una dificultad fácil (sin dirección de corte de cubos) y también en una dificultad difícil (con dirección de corte de cubos).
- **Canciones y velocidad**: Para las pruebas se seleccionaron ocho pistas musicales diferentes. Cabe señalar que la selección de las canciones no fue aleatoria; de hecho, esta se basó en la estadística de notas por segundo de cada canción, para que de esta manera la dificultad de las canciones sea similar. Por lo tanto, el rango de valores de cubos por segundo está entre 2 y 3 cubos/s. En la tabla 5.13 se aprecian las canciones a probar con sus debidas estadísticas. Además, la velocidad de todas las canciones es un 30 % más lenta con el fin de no estresar tanto al robot y que se le sea más fácil jugar.

Tabla 5.13: Canciones seleccionadas para probar al robot.

Canción	Autor	Duración	Notas	Notas/s	Tempo (BMP)
Beat Saber	Jaroslav Beck	1 min 50 seg	228	2.06	166
$100 Bills	Jaroslav Beck	2 min 21 seg	377	2.66	210
Sinister	Ako	3 min 40 seg	521	2.37	120
Bug	Kairiki Bear ft. Hatsune Miku	2 min 51 seg	454	2.65	196
Failure Girl	Kairiki Bear, Maretu ft. Hatsune Miku	3 min 3 seg	437	2.38	118
Escape	Jaroslav Beck Summer Haze	2 min 41 seg	368	2.28	175
Firestarter	Tanger	3 min 2 seg	365	2.00	135
Into the Dream	Jaroslav Beck	3 min 14 seg	395	2.03	183
	Total	22 min 42 seg	3145		
	Total (30 % lento)	29 min 30 seg			

- **Condiciones de juego**: Para hacer el juego lo más simple posible, se activaron modificadores que eliminan otros aspectos de la dinámica, tales como minas y paredes, que son obstáculos que se deben esquivar. Además, se activa el modo inmortal, el cual consiste en que el jugador no va a perder a pesar de que comenta muchos errores en la canción.

Con lo anterior propuesto, se podría llegar a interpretar que la cantidad de pruebas no es suficiente para caracterizar a los modelos, ya que solo se usan 8 canciones; no obstante, se debe tener en cuenta el tipo de muestra para caracterizar al robot.

Sobre el contexto de *Beat Saber*, puede haber dos perspectivas para interpretar la muestra que determina la precisión del robot. La primera consiste en tratar la cantidad de cubos cortados en una sola canción como una muestra, como consecuencia solo se tendrían 8 muestras por modelo y dificultad, lo cual se puede concluir que no es representativo.

Por otro lado, la segunda consiste en tomar en cuenta la cantidad de cubos cortados a lo largo de todas las canciones, por lo tanto se tendrían 3145 muestras, lo cual se puede considerar como representativo para la caracterización.

Ambas aproximaciones siguen tipos de muestreo diferentes. Si se toma como muestra una canción, se estaría aplicando un muestreo de tipo Poisson, la cual expresa la probabilidad de ocurrencia de un evento n veces en un intervalo fijo de tiempo λ. Por su parte, si se toma como muestra la mera acción de cortar los cubos sin considerar la concepción de la canción, se estaría siguiendo un muestreo de tipo Bernoulli, el cual tiene dos posibles resultados: se cortó o no se cortó. Bajo esta lógica se tendrían las 3145 muestras para determinar la probabilidad de corte.

En el ámbito de aprendizaje por refuerzo profundo aplicado en brazos robóticos, es común seguir la aproximación de Bernoulli, ya que cada muestra consiste en contestar si el robot tuvo éxito o no al resolver una tarea, y con la iteración de varias veces de la tarea se llega a un porcentaje de precisión confiable. Por ello, se siguió la estrategia de tratar a cada corte de cubo como una muestra individual de tipo Bernoulli.

Dicho esto, en esta sección solamente se hará énfasis en el porcentaje de cubos acertados. El análisis del movimiento de cada motor y del sable se hace en el capítulo de resultados, ya que se compara con los movimientos del robot real.

Con todo lo anterior mencionado, a continuación se muestra en la tabla 5.14 el desempeño del robot simulado a lo largo de las canciones en la dificultad fácil y en la tabla 5.15 los resultados de la dificultad difícil.

Tabla 5.14: Desempeño simulado de modelos en dificultad fácil.

Canción	Cubos	TD3	TD3+ HER	DDPG	DDPG+ HER	PPO
Beat Saber	228	166	163	132	170	52
$100 Bills	377	321	314	250	322	91
Sinister	521	412	423	321	397	228
Bug	454	363	394	318	362	212
Failure Girl	437	359	381	303	337	194
Escape	368	311	282	207	298	97
Firestarter	365	314	327	298	316	149
Into the Dream	395	308	318	281	276	167
Total	3145	2554	2602	2110	2478	1190
%	100	81.20	82.73	67.09	78.79	37.83

Tabla 5.15: Desempeño simulado de modelos en dificultad difícil.

Canción	Cubos	TD3	TD3+ HER	DDPG	DDPG+ HER	PPO
Beat Saber	228	55	59	47	52	15
$100 Bills	377	123	112	118	112	21
Sinister	521	249	220	197	202	112
Bug	454	205	186	175	182	101
Failure Girl	437	159	176	149	168	86
Escape	368	110	103	103	110	38
Firestarter	365	200	185	168	175	69
Into the Dream	395	171	173	215	170	84
Total	3145	1272	1214	1172	1171	526
%	100	40.44	38.60	37.26	37.23	16.72

En la experimentación virtual se observó que los modelos con mejores resultados son los de tipo *off-policy*, en especial los modelos TD3+HER y DDPG+HER. En la dificultad fácil, estos se caracterizaron por mantener un movimiento inteligente para cortar los cubos, es decir, mantenían la punta del sable en el centro a la espera de los cubos, para que así pudieran alcanzarlos con movimientos ágiles.

Por otra parte, el algoritmo *on-policy* PPO no pudo lograr un corte eficiente de cubos en la dificultad fácil, principalmente porque casi no movía al robot o lo hacía de manera lenta. Esto puede deberse a que el algoritmo está planeado para que la política no tenga actualizaciones tan grandes, por lo que dificulta el proceso de exploración y se conforma con recompensas obtenidas por la distancia de los cubos más que por golpearlos. En otras palabras, el algoritmo PPO no muestra mucha generalización, ya que se le dificulta adaptarse a un ambiente que se comporta ligeramente diferente al que fue entrenado.

Con la experimentación de la dificultad difícil no se alcanzaron resultados eficientes. En todos los modelos se observó una baja en la destreza del movimiento del robot; en otras palabras, el robot mostraba más duda para alcanzar los cubos y no golpeaba en la dirección correcta. Esto es un comportamiento esperado, ya que en el entrenamiento

los modelos tampoco pudieron lograr cortar en la dirección correcta de manera eficiente. Además, el algoritmo para detectar el ángulo de corte no era perfecto, podía tener ruido que ocasionaba que marcara un ángulo errado, haciendo que el agente tuviera problemas para saber a qué dirección cortar.

Con lo analizado, se decidió que los modelos que serán probados en un ambiente real son DDPG+HER, TD3+HER y PPO. Se descartan a DDPG y TD3, ya que mostraron menos desempeño que sus variantes con HER, en especial en la dificultad fácil. Por su parte, PPO se pone en práctica para observar cómo un algoritmo *on-policy* se comporta en un ambiente real.

Es importante señalar que los porcentajes mostrados en el entrenamiento paralelo son similares a los obtenidos en esta experimentación, a excepción de PPO, lo que demuestra que el entrenamiento representa bien a la dinámica de *Beat Saber*. Esto, a pesar de que hay una diferencia en la naturaleza del movimiento de los cubos, es decir, en la simulación se tiene que cuando los cubos aparecen enfrente del brazo robótico, estos se aproximan a él a una velocidad lineal y moviéndose a lo largo del eje x en línea recta, o en otras palabras, se mueven siguiendo una perspectiva ortogonal; de manera contraria en la evaluación, debido a la naturaleza del sistema de visión computacional propuesto en este trabajo, los cubos se mueven siguiendo una perspectiva radial; o sea, los cubos aparecen con respecto a un punto foco y de allí se van aproximando a lo largo de una recta con componentes x, y, z y a una velocidad no lineal.

Lo anterior descrito expresa que el robot no solo aprendió a poner el sable en un punto específico para que este sea cortado cuando pase, sino que también aprendió a reducir la distancia del sable con el cubo, despreciando que este no solo se esté moviendo a través del eje x. Esto demuestra una buena generalización del agente a diversos tipos de movimientos de los cubos.

Capítulo 6

Resultados y análisis

En esta sección se pondrán a prueba los modelos resultantes del entrenamiento virtual con la detección de objetos, teniendo al robot real y al set de realidad virtual HTC. Para fines de simplificación de pruebas, se pondrán a prueba los algoritmos de DDPG+HER, TD3+HER y PPO. Además, se hará una comparación con el rendimiento humano, bajo condiciones similares al robot.

6.1. Evaluación en entorno físico

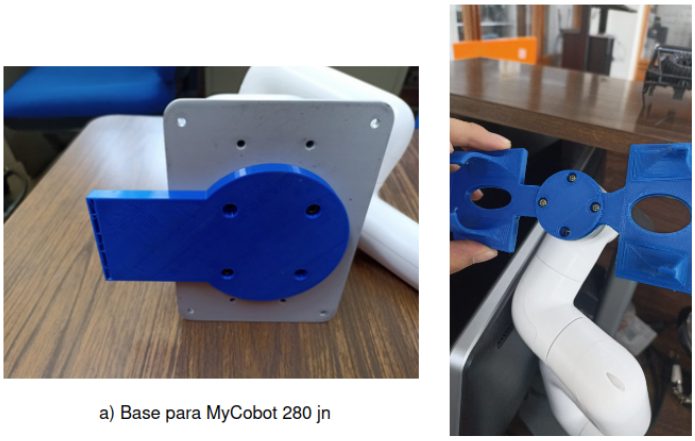
6.1.1. Montado de entorno físico

Antes de comenzar el análisis del desempeño de los modelos, es fundamental describir las características y preparación del entorno físico para que el robot sea capaz de sostener el mando del HTC vive e interactuar con el ambiente virtual de Beat Saber.

Para que MyCobot pueda moverse con seguridad y levantar el control de HTC Vive, se imprimieron y ensamblaron las piezas mostradas en la figura 6.1. La pieza en la ilustración 6.1a se descargó de un modelo en internet[1], la cual es una base que tiene la tarea de sostener a MyCobot con la ayuda de una pinza de presión que aprieta la parte sobresaliente con alguna superficie, en este caso una mesa. La segunda pieza mostrada en 6.1b también fue conseguida en internet[2] y modificada con el objetivo de adaptarla a la estructura de

[1] https://www.thingiverse.com/thing:4725036
[2] https://cults3d.com/en/3d-model/game/vive-wand-full-body-tracking

MyCobot y a su vez de que pese lo menos posible; de esta manera se logró que la pieza pesara un aproximado de 30 gramos. Que teniendo en cuenta el peso del control HTC Vive, se tendría un total de 230 gramos aproximadamente, lo cual todavía no sobrepasa la carga máxima del brazo.



a) Base para MyCobot 280 jn

a) Soporte MyCobot 280Jn para control HTC Vive

Figura 6.1: Piezas integradas a MyCobot para la interacción física.

La tarjeta JetsonNano puede ejecutar los modelos entrenados llevando su capacidad casi al 70 % de RAM y 80 % de CPU y logrando la velocidad de respuesta de aproximadamente 20 Hz. A pesar de ello, el sistema embebido ya no tendría capacidad para ejecutar YOLO para alcanzar la detección en tiempo real. Por ello, se decidió dividir la carga de trabajo con otra computadora. De esta manera, MyCobot 280jn estaría ejecutando todo el control del robot y la inteligencia artificial, y por su parte, la segunda computadora se encargaría del funcionamiento de HTC Vive, la ejecución de Beat Saber y la detección de cubos con YOLO. La comunicación entre la computadora y MyCobot se lleva acabo mediante un cable Ethernet UTC 5, logrando un tiempo de respuesta de 1.5 ms. En este medio se transmite la información detectada por YOLO y el estado del juego. En la figura 6.2 se muestra toda la arquitectura lógica para la interacción física.

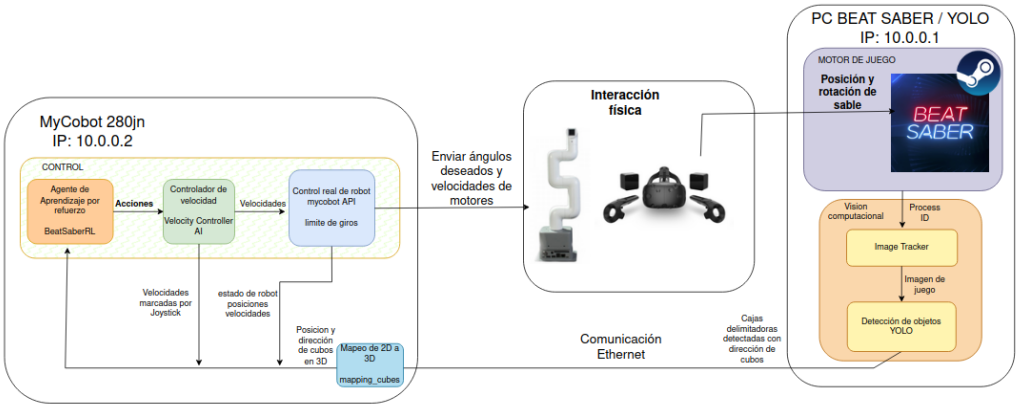


Figura 6.2: Arquitectura para la interacción robot-HTC Vive.

En la tabla 6.1 se muestran las características de hardware de la segunda computadora. Con ello, Beat Saber se ejecuta adecuadamente y YOLO da un aproximado de 80 FPS.

Tabla 6.1: Especificaciones de hardware para PC BEAT SABER / YOLO.

Componente	Descripción
GPU	NVIDIA Quadro RTX 4000, 16GB GDDR6
CPU	Intel Core i7-9700 4.70GHz, Quad-Core
RAM	2 x 8GB DDR4, 3200MHz
Tarjeta Madre	Dell Precision 3630

Para tener una mejor caracterización de la posición de cada componente del HTC Vive, en la figura 6.3 se muestran las distancias de los componentes con respecto al robot; es decir, la distancia de las estaciones base con su grado de rotación y la distancia del caso de realidad virtual. También en la figura 6.4 se muestra la altura de cada una de las estaciones y la altura de las mesas donde estaban montadas. Por último, en la imagen 6.5 se muestra la altura de la mesa donde estaba instalado MyCobot, o en otras palabras, la altura del brazo robótico con respecto al suelo.

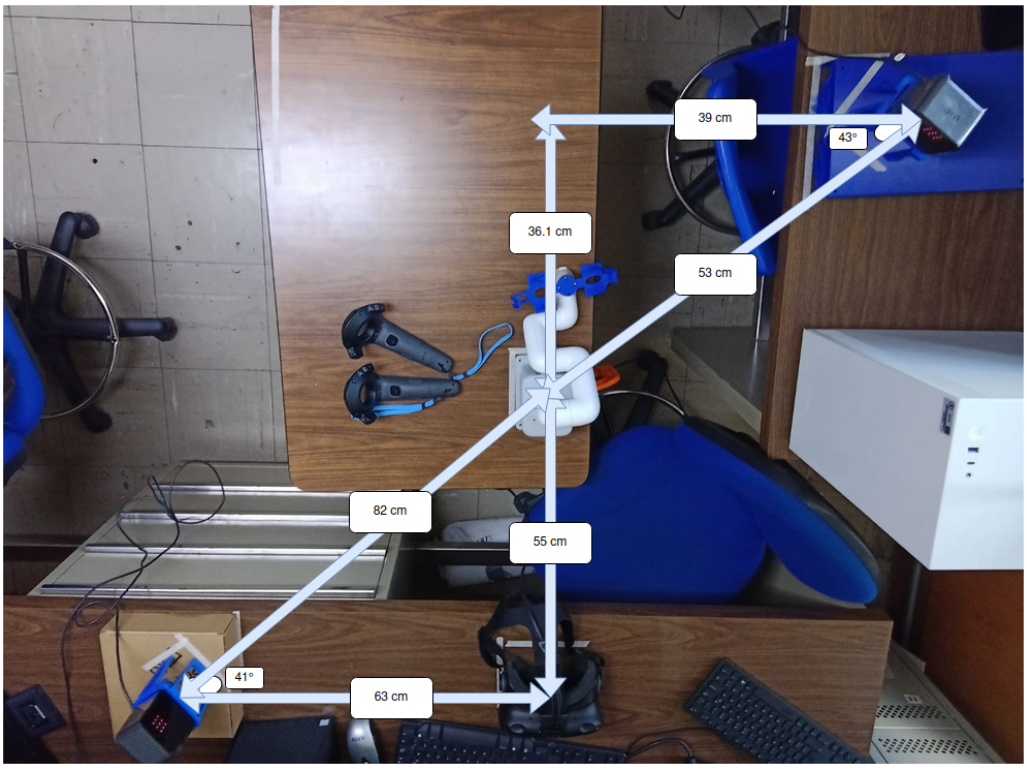


Figura 6.3: Distancias de estaciones base y casco HTC Vive con respecto al robot.

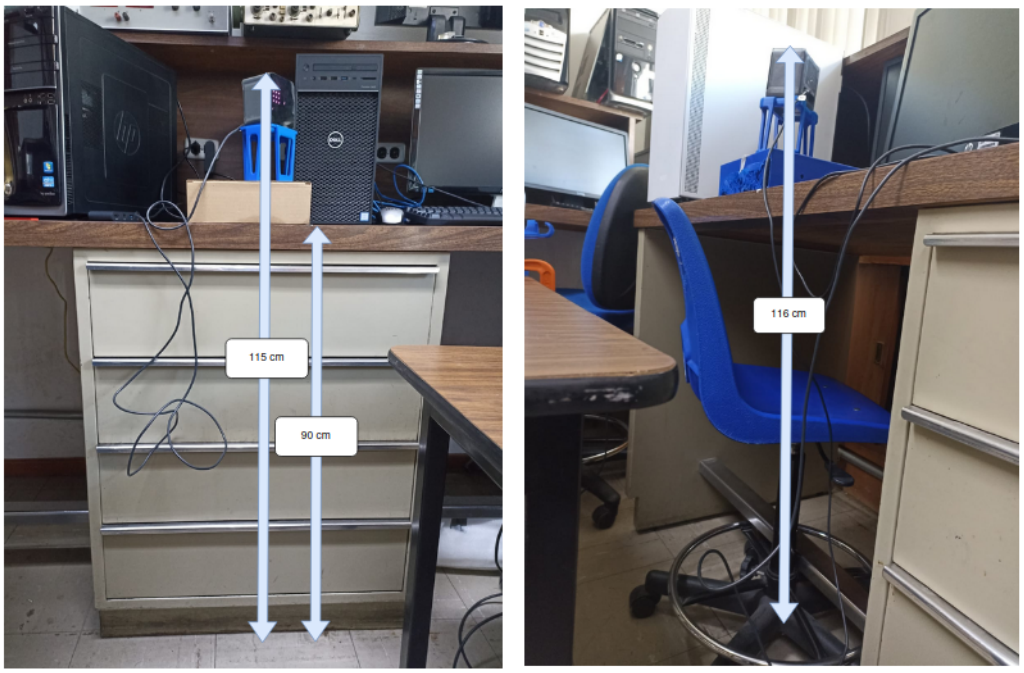


Figura 6.4: Altura de las estaciones base HTC Vive.

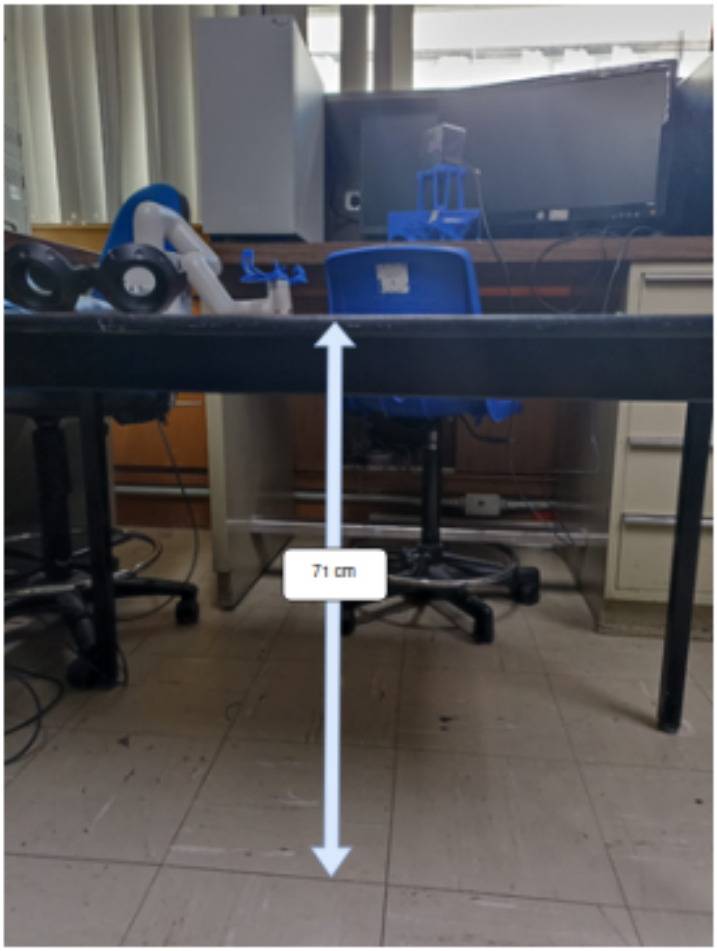


Figura 6.5: Altura de la mesa del robot.

En la figura 6.6 se aprecian todo los elementos montados y al robot ya jugando un nivel del entorno virtual *Beat Saber*. Con toda la caracterización del entorno, las pruebas realizadas para medir el rendimiento del robot se presentan en la siguiente sección.

Figura 6.6: Entorno físico montado y MyCobot jugando Beat Saber.

6.1.2. Métricas de desempeño físico

Las pruebas a realizar en el entorno físico son las mismas que se hicieron en el entorno simulado, es decir, se van a poner a prueba las 8 canciones propuestas que dan un total de 3145 cubos a cortar, esto con una realentización del 30 %. No obstante, se excluyeron los modelos TD3 y DDPG, ya que sus variantes con HER mostraron mejor rendimiento en simulación. Los resultados de las pruebas se encuentran expresados en las tablas 6.2 y 6.3 para las dificultades fácil y difícil respectivamente.

Tabla 6.2: Desempeño real de los modelos en modo fácil.

Canción	Cubos	TD3+ HER	DDPG+ HER	PPO
Beat Saber	228	130	141	97
$100 Bills	377	248	296	181
Sinister	521	372	387	284
Bug	454	331	346	277
Failure Girl	437	325	319	253
Escape	368	265	282	162
Firestarter	365	281	293	186
Into the Dream	395	279	275	197
Total	3145	2231	2339	1637
%	100	70.93	74.37	52.05

Tabla 6.3: Desempeño real de los modelos en modo difícil.

Canción	Cubos	TD3+ HER	DDPG+ HER	PPO
Beat Saber	228	33	56	10
$100 Bills	377	88	101	15
Sinister	521	195	180	97
Bug	454	153	160	84
Failure Girl	437	120	141	70
Escape	368	103	120	31
Firestarter	365	130	123	62
Into the Dream	395	153	160	52
Total	3145	975	1041	421
%	100	31.00	33.10	13.38

Con respecto a la dificultad fácil y los algoritmos *on-policy* TD3 y DDPG se obtuvieron resultados destacables, mostrando un resultado de 70.93 % y 74.37 % de precisión respectivamente. Si se comparan los porcentajes con los de la simulación, se puede notar que bajaron, en especial el algoritmo TD3, ya que bajo casi un 12 % de precisión.

Estas diferencias en la precisión pueden darse por diferentes factores, ya sea por la posición de la cámara dentro del juego (determinada por la posición del casco de HTC Vive), la calibración del centro del ambiente virtual, la posición real del robot con respecto a las estaciones HTC Vive, la calibración de la cámara estenopeica. No obstante, la razón más relevante es por el ruido que puede haber en el movimiento del robot real, haciendo que el robot tenga comportamientos no esperados, tales como el moverse más lento, o más rápido de lo debido, haciendo que la precisión baje. A todos los factores mencionados con anterioridad se pueden denominar como brecha de la realidad, ya que son variables que aparecen por la misma complejidad de tratar con componentes reales.

Sobre la dificultad difícil, ambos modelos presentaron los mismos problemas de movimientos presentes en la simulación y la reducción de porcentaje provocada por la brecha de la realidad; estos dos factores ocasionaron que se obtuviera un porcentaje de 31 % para TD3 y 33.10 % para DDPG, no obstante, siguen siendo coherentes con las capacidades del agente mostradas en el entrenamiento y simulación.

Con estos dos experimentos se puede decir que el modelo DDPG se comportó mejor que TD3 en el entorno real. Mientras que en el simulado TD3 se comporta mejor que DDPG.

En el caso del modelo PPO, mostró un comportamiento no esperado. Se observó que se obtuvo un mejor rendimiento en ambas dificultades con respecto a la simulación, pasando de tener un 37.53 % a un 52.05 %. La mejora se debe a que el agente movía al robot con más frecuencia y velocidad que en la simulación. Esto refuerza la idea de que el algoritmo PPO al no hacer actualizaciones grandes en su política, tiene un movimiento muy limitado, pero cuando se agrega el ruido de la realidad, el modelo tiende a explorar más para conseguir mejor recompensa.

Ya con los resultados analizados de cada modelo, se puede hacer una comparación de los movimientos ejecutados por los motores tanto en simulación como en el entorno físico, esto con el fin de determinar si la transferencia de la política a la realidad fue eficiente y coherente con lo dado en la simulación. Para lograr lo anterior, cuando se estaban reproduciendo las canciones, se grabó la posición en radianes de cada motor y la posición en coordenadas en (x, y, z) del punto final del sable generado en ROS a lo largo del tiempo. Los resultados se pueden observar en las 5 figuras que se muestran a continuación.

En cada figura se caracteriza el movimiento del robot; entonces, para poder tener un análisis, se tomaron fragmentos de canciones de aproximadamente 15 segundos para observar el movimiento de los modelos. Por cada figura, las dos primeras gráficas muestran el valor de los ángulos de cada motor tanto simulados como en real. Las tres siguientes gráficas muestran las componentes x (rojo), y (verde) y z (azul) del punto final del sable; la línea sólida representa el valor de la ejecución real y la punteada en la simulación.

TD3+HER (Fácil)

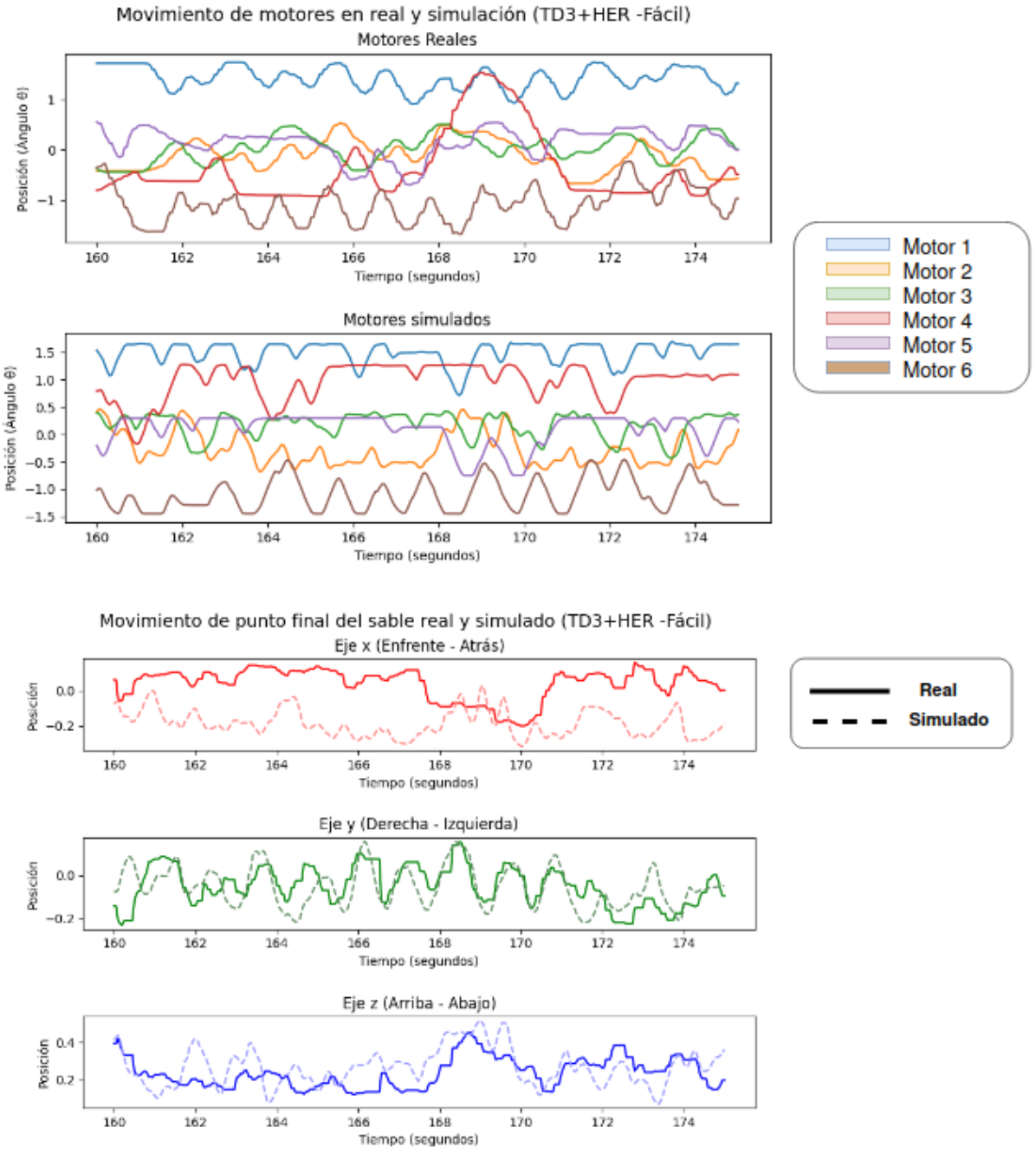


Figura 6.7: Movimiento simulado y real con modelo TD3+HER en dificultad fácil.

TD3+HER (Difícil)

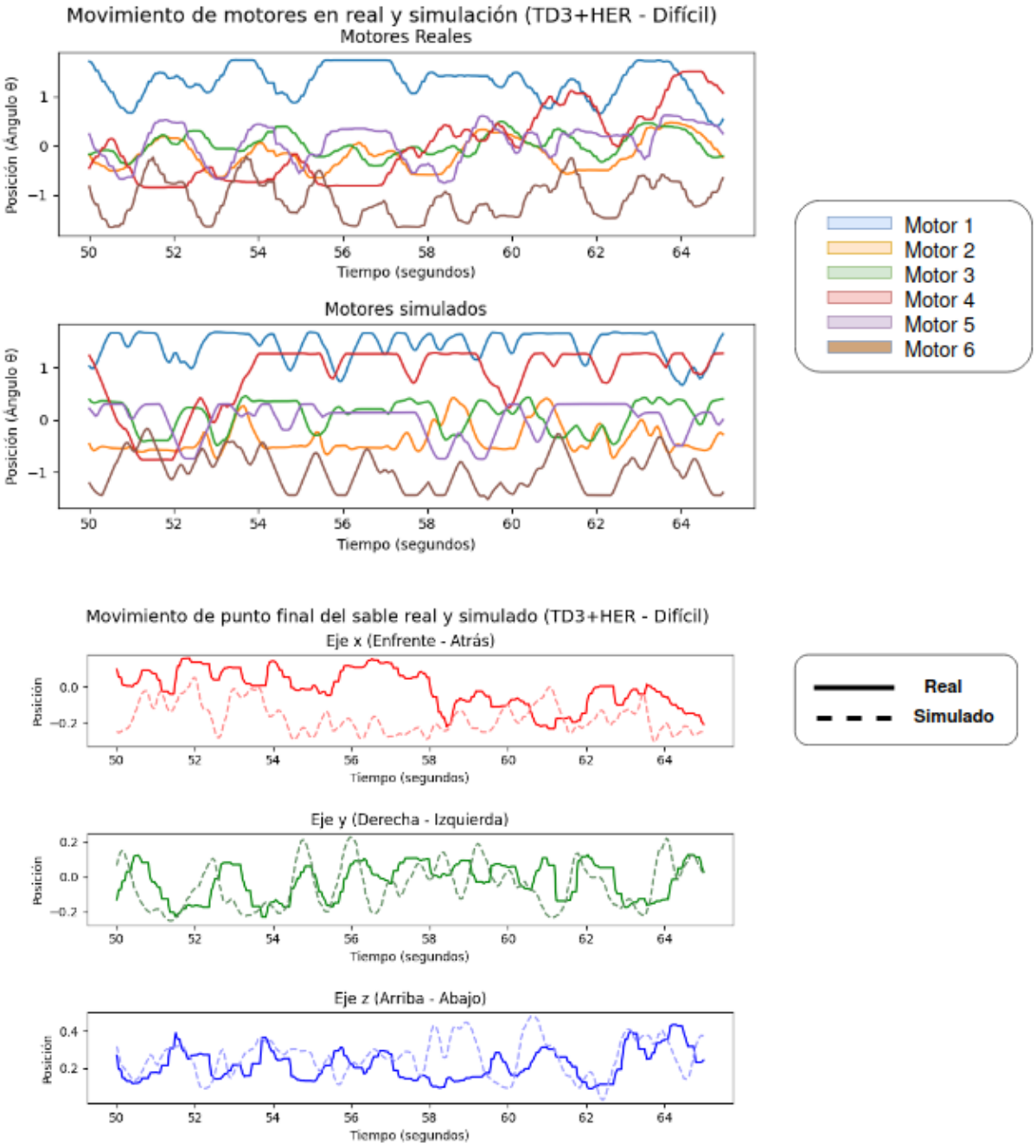


Figura 6.8: Movimiento simulado y real con modelo TD3+HER en dificultad Difícil.

DDPG+HER (Fácil)

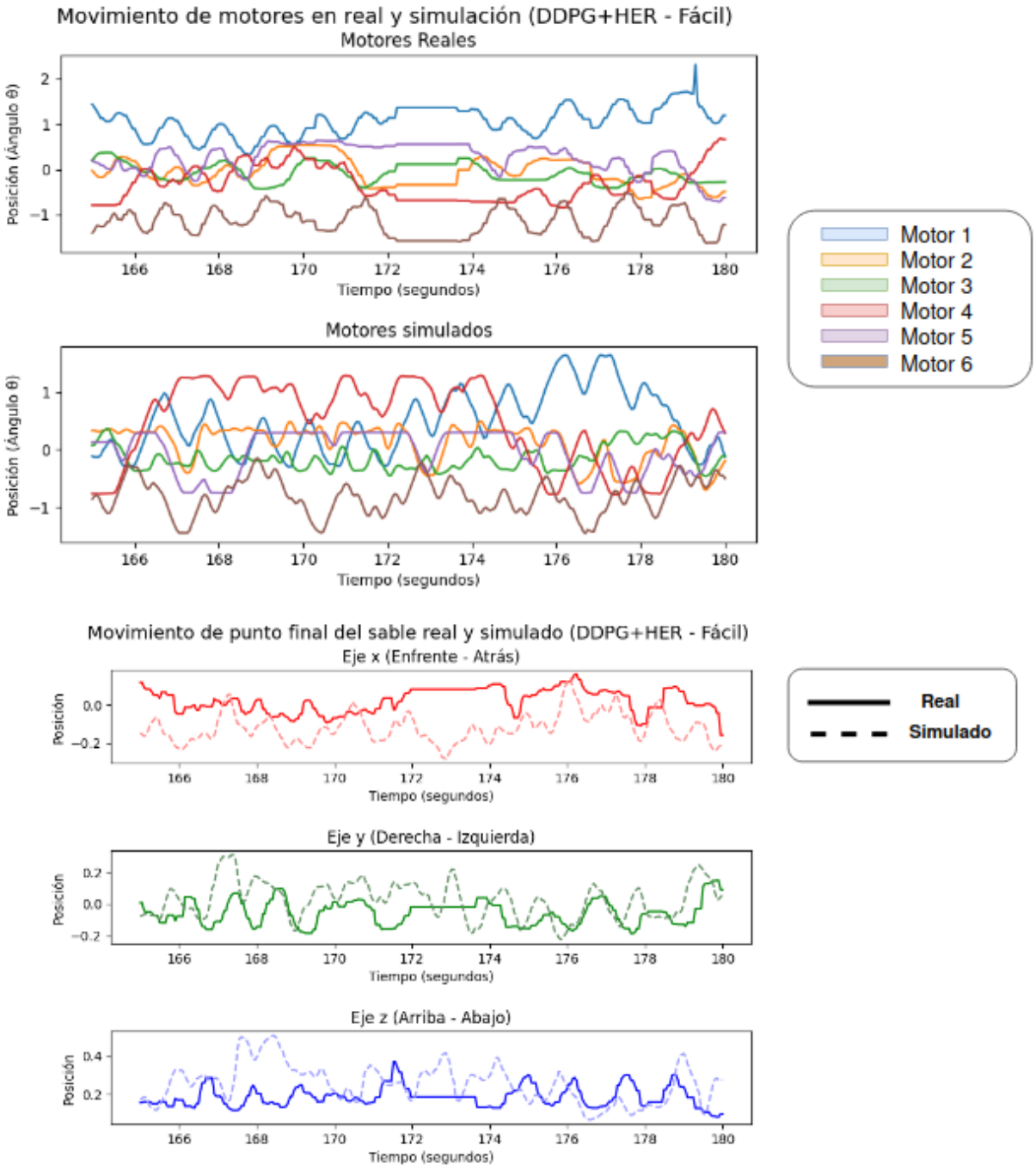


Figura 6.9: Movimiento simulado y real con modelo DDPG+HER en dificultad fácil.

DDPG+HER (Difícil)

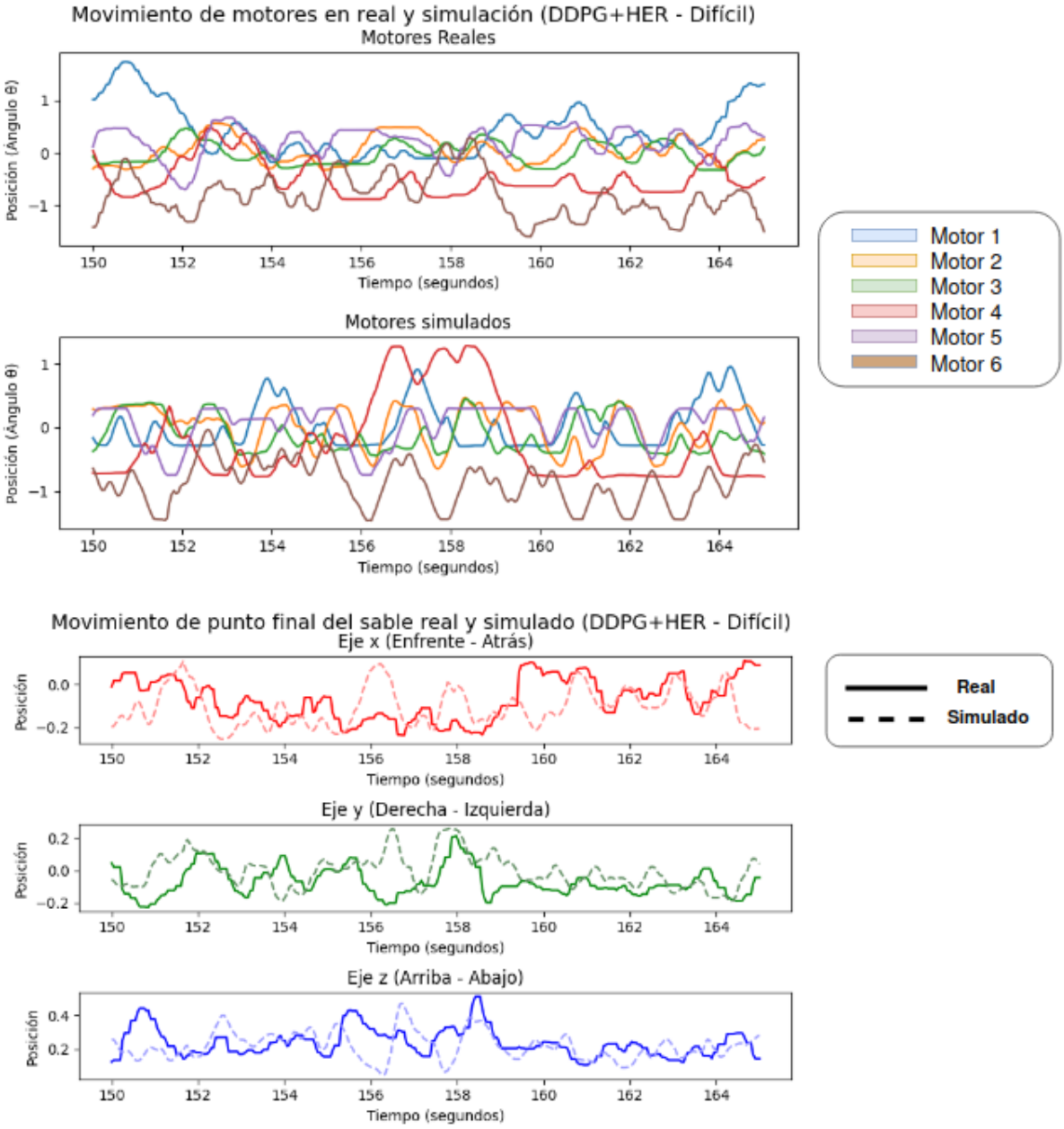


Figura 6.10: Movimiento simulado y real con modelo DDPG+HER en dificultad difícil.

PPO (Fácil)

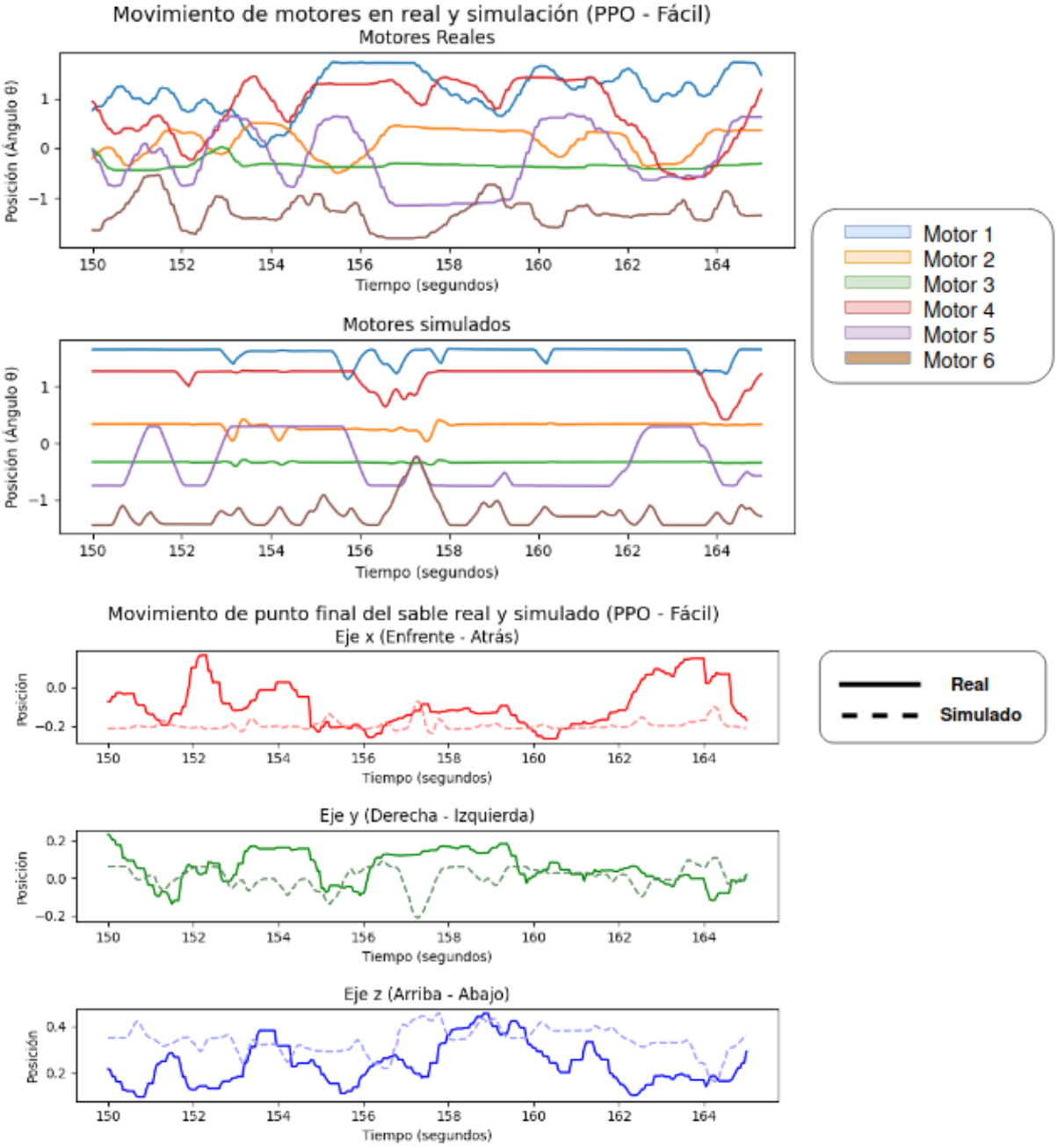


Figura 6.11: Movimiento simulado y real con modelo PPO en dificultad fácil.

PPO (Difícil)

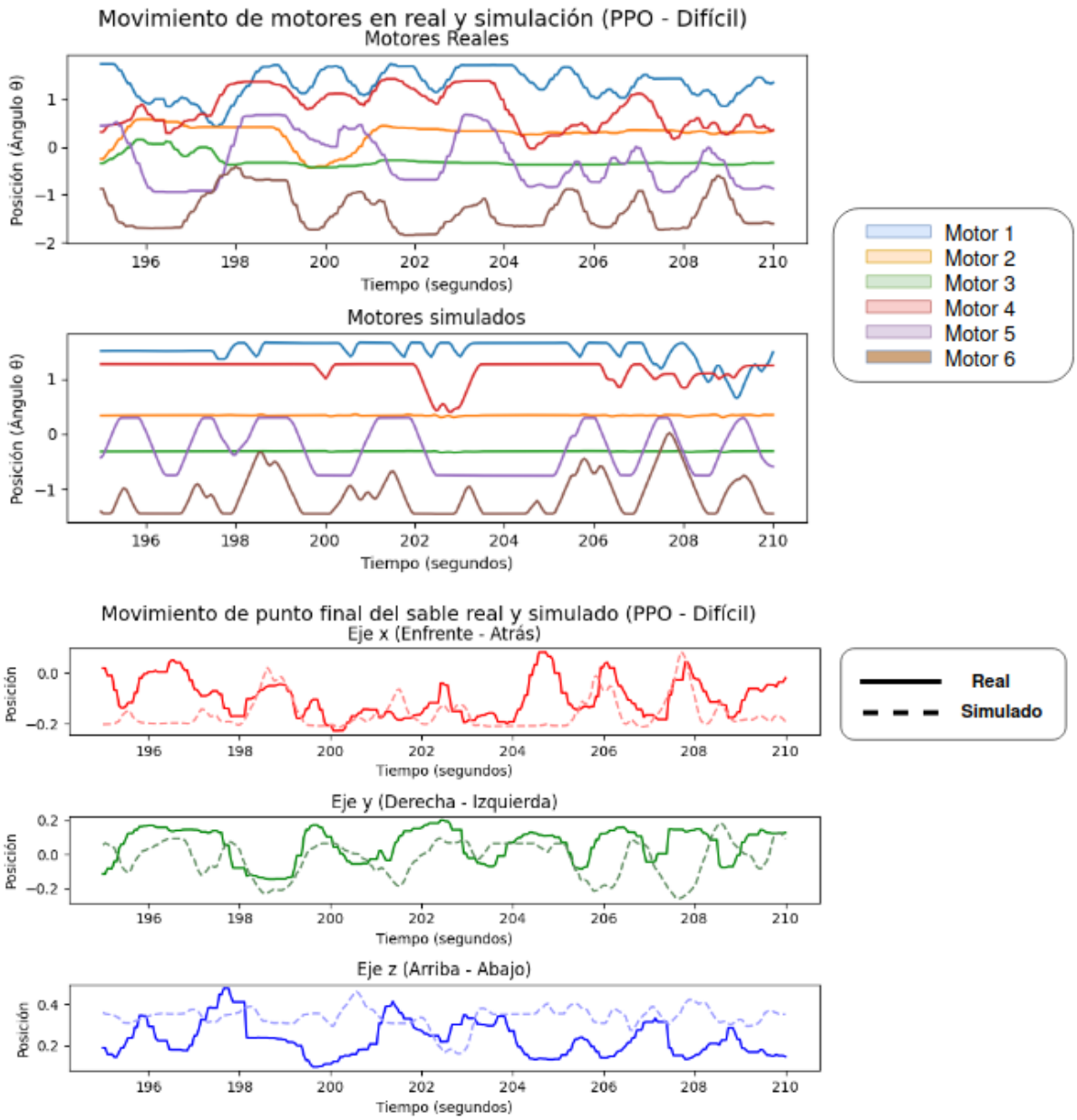


Figura 6.12: Movimiento simulado y real con modelo PPO en dificultad difícil.

El contenido de las gráficas complementa los resultados vistos en el porcentaje de precisión, esto al observar comportamientos interesantes que sirven para caracterizar mejor a los modelos. A continuación se brinda un análisis de los aspectos más destacables.

- **Generalización del problema**: Si se compara el movimiento de los motores en cada caso, se aprecia que no siguen una cinemática similar, con lo que se podría deducir que el agente no está cumpliendo con la tarea. A pesar de ello, si se toma en cuenta la posición del sable, se puede ver una gran correlación simulación-realidad en la mayor parte del tiempo, en especial en los ejes y y z, correspondientes al ancho y altura respectivamente. Esto refleja lo aprendido por el robot, lo cual consiste en que el agente es capaz de cortar los mismos cubos siguiendo estrategias diferentes, o sea, diferentes configuraciones de los motores. Se podría decir que esta es una ventaja de haber utilizado cinemática directa.

  Entonces, lo anterior expresa una buena generalización de los modelos cuando se encuentran en diferentes posiciones; no obstante, PPO es la excepción que se analiza más adelante.

- **Movimiento de la punta del sable**: De las tres dimensiones de movimiento, el eje x es el que menos guarda correlación simulación-realidad. Esto es lógico, ya que el eje x corresponde a la profundidad, ya que no importa mucho si el cubo se corta un poco antes o después cuando ya está en el alcance del robot. En cambio, la correlación en los ejes y y z son fundamentales, ya que definen por dónde va a pasar el cubo, es por eso que el robot aprendió a aproximarlos bien.

- **Movimiento en la dificultad difícil**: A pesar de que el robot no logró destacar en la dificultad difícil, la correlación simulación-realidad sigue siendo alta, indicando que aprendió algo en concreto pero no lo óptimo para cortar con precisión. Esto refuerza la idea de que el problema puede estar en el planteamiento de las funciones de recompensa que indican cómo es que aprende el robot.

- **Soporte a perturbaciones**: El robot, cuando ya tenía un tiempo considerable jugando, comenzaba a presentar una perturbación que hacía que se detuviera por aproximadamente 2 segundos y después volviera a la normalidad. Esto quiere decir que el robot podía recuperarse de un suceso inesperado. Este comportamiento se ve ilustrado en la figura 6.9 donde se aprecia cómo entre los segundos 172 y 174 el robot dejó de moverse en la realidad, aun así después de la interrupción pudo seguir jugando con normalidad. Esto agrega más valor a la generalización del agente.

- **Comportamiento de PPO**: Como ya se ha mencionado, el comportamiento de PPO fue irregular en la comparación simulación-realidad. Si se analizan las figuras 6.12 y 6.12 se puede ver que los motores simuladores casi no presentan movimiento, mientras que en los reales se presenta una cinemática más activa. Esto da como resultado que la correlación de la posición del sable no sea tan alta como en DDPG y TD3.

6.1.3. Comparación Humano - robot

Para tener un punto de comparación de un factor humano, se midió el desempeño de personas al jugar *Beat Saber*. Lo ideal es que las personas realizaran todas las canciones en ambas dificultades como lo realizó cada modelo, no obstante, por cuestiones de tiempo de disponibilidad de cada persona, no podía lograr hacer la prueba, ya que significaba que cada persona jugara casi una hora sin parar. Por ello, la prueba se redujo a una canción que sea representativa de todas las canciones consideradas.

La canción seleccionada para la prueba fue *Sinister*, ya que es la canción que más cubos pose, su duración es la mayor, su precisión es cercana a la media de cada modelo y sus cubos por segundo (2.37) también se acerca a la media de todas las canciones (2.30).

La muestra para hacer la estadística consistió en 12 individuos entre los 19 y 29 años, los cuales jugaron la canción propuesta, tanto en sus dificultades fácil y difícil. Además de jugar en condiciones parecidas a las que fue sometido el robot, en la figura 6.13 se aprecia un agente humano jugando la canción. Cabe mencionar que los sujetos de prueba no tenían experiencia previa con Beat Saber, solamente lo conocían. Con lo anterior mencionado, en la tabla 6.4 se encuentran expresados los resultados de este experimento que caracterizan al desempeño humano.

Figura 6.13: Agente humano interactuando con el entorno de realidad virtual.

Tabla 6.4: Desempeño promedio de agentes humanos.

Dificultad	Total de cubos	Promedio de Cubos golpeados $\bar{x}$	Varianza de muestra s	Precisión %
Fácil	521	461	18.31	88.29
Difícil		362	27.23	70.05

Para poder caracterizar los modelos con esta canción, cada uno jugó la canción cuatro veces en ambas dificultades para tener una precisión más representativa en esta canción. Los resultados se pueden ver en las tablas 6.5, 6.6 y 6.7. Además, se proporciona una comparación con el porcentaje humano, de tal manera que se expresa en porcentaje qué tanto se acercaron los modelos al desempeño humano.

Tabla 6.5: Desempeño de TD3+HER en Sinister y comparación con humano.

Dificultad	Total de cubos	Promedio de Cubos golpeados $\bar{x}$	Varianza de muestra s	Precisión %	Relación Robot/Humano
Fácil	521	370.6	5.98	71.01	80.04
Difícil		173.4	8.01	33.28	47.50

Tabla 6.6: Desempeño de DDPG+HER en Sinister y comparación con humano.

Dificultad	Total de cubos	Promedio de Cubos golpeados $\bar{x}$	Varianza de muestra s	Precisión %	Relación Robot/Humano
Fácil	521	385.8	6.14	73.89	83.69
Difícil		176.6	6.69	33.38	47.65

Tabla 6.7: Desempeño de PPO en Sinister y comparación con humano.

Dificultad	Total de cubos	Promedio de Cubos golpeados $\bar{x}$	Varianza de muestra s	Precisión %	Relación Robot/Humano
Fácil	521	279.4	7.73	53.62	60.73
Difícil		102.3	9.65	19.57	27.79

Con los resultados se pudo observar que DDPG+HER fue el modelo que más se aproximó al desempeño humano, con una relación robot/humano de 83.69 % en la dificultad fácil y un 47.65 % en difícil.

Por otra parte, el que peor se aproximó fue PPO con una relación de 60.73 % en fácil y 27.79 % en difícil.

Capítulo 7

Conclusiones y trabajo futuro

En este trabajo se desarrolló con éxito un agente de aprendizaje por refuerzo profundo con visión computacional capaz de controlar a un brazo robótico de pequeña escala como lo es MyCobot 280jn para que interactuara con el entorno virtual Beat Saber por medio de la interfaz HTC Vive y pudiera alcanzar hasta cierto punto la destreza humana. Cabe resaltar que la visión computacional se llevó a cabo con el modelo YOLO V7.

Se pusieron a prueba los algoritmos PPO, DDPG, TD3 estos dos últimos con una variante de búffer de repetición de experiencia tipo HER. Con estos algoritmos se observó que DDPG+HER tuvo mejores resultados al ejecutar 8 canciones a 70 % de velocidad en Beat Saber, obteniendo una precisión del 74.37 % en una dificultad fácil (cubos sin dirección) y en difícil (cubos con dirección) un 33.10 %, más adelante se ahonda en este último porcentaje. Además, fue capaz de alcanzar un 83.69 % de la capacidad humana en una de las 8 canciones antes mencionadas en la dificultad fácil, mientras que en la difícil se pudo alcanzar un 47.65 % de la capacidad en esa misma canción.

Los resultados de los modelos desde el entrenamiento, la evaluación en simulación, la evaluación en entorno físico y la comparación con el desempeño humano se muestran en la tabla 7.1. Con estos datos se puede observar que el segundo mejor fue TD3 y el de peor desempeño fue PPO. Con los resultados, también se puede determinar que TD3 no se adaptó bien al salto de la simulación a lo físico.

Tabla 7.1: Resumen de precisión de cubos a lo largo del trabajo.

Fácil				
Modelo	Precisión Entrenamiento %	Precisión Evaluación Virtual %	Precisión Evaluación Física %	Comparación Robot/Humano % (1 canción)
DDPG+HER	63.56	78.79	74.37	83.69
TD3+HER	71.65	82.73	70.93	80.04
PPO	71.83	37.83	52.05	60.73
Difícil				
DDPG+HER	28.88	37.23	33.10	47.65
TD3+HER	34.86	38.60	31.00	47.50
PPO	27.91	16.72	13.38	27.79

Con los resultados, se puede determinar que TD3 no se adaptó del todo bien al salto de la simulación a lo físico. A pesar de mostrar una velocidad considerable, la precisión al cortar los cubos disminuyó. En el caso de PPO, a pesar de haber tenido un buen entrenamiento, no presentó un desempeño destacable en las siguientes fases debido a que no se adaptó a las diferencias que habían con respecto al entrenamiento, lo que resultaba que el modelo no moviera casi al robot. En resumen, no tuvo una buena generalización del ambiente y no posee una buena velocidad de respuesta rápida.

Sobre el uso del algoritmo HER, se observó que no disminuye considerablemente el tiempo de entrenamiento, pero sí mejora un poco la precisión y estabilidad de los modelos bajo el contexto del trabajo.

También se logró desarrollar un sistema de entrenamiento paralelo de robots sobre la plataforma ROS, que permite desempeñar entrenamientos más rápidos y estables. Este funciona gracias a la creación de canales de tópicos, nodos, suscriptores y publicadores que aíslan el funcionamiento de un robot con otro.

Con respecto a la dificultad difícil, se observa que desde el entrenamiento los modelos tenían problemas con cortar los cubos en la dirección correcta, lo que quiere decir que en el entrenamiento faltó algún factor que incitara más al agente a cortar los cubos correctamente. Por ello, el trabajo futuro estaría centrado en encontrar más estrategias para mejorar el porcentaje de precisión de los modelos tanto en fácil como en difícil. Por lo tanto, a continuación se brindan algunos aspectos que se pueden llegar a mejorar.

- **Sistema de recompensas**: Una mejora directa es la creación de criterios de recompensas adecuados. En este trabajo se propusieron 4 sistemas de recompensas que lograron que el agente pudiera golpear los cubos con el sable, pero no cortarlos en la dirección correcta. Una estrategia que pudiera funcionar para mejorar la precisión es que se recompense al agente si corta al cubo de lado, para así evitar que solamente ponga el sable enfrente del cubo y esperar a que llegue. De esta manera se podría mejorar la sincronía sable-cubo y como consecuencia la precisión de corte.

- **Sintonía fina (*fine-tunning*)**: Una estrategia para superar los efectos negativos de la transferencia de simulación o físico, es la sintonía fina, la cual consiste en someter al agente a un breve entrenamiento con el robot real, para que así pueda acostumbrarse a los cambios y ruidos generados por la brecha de realidad.

- **El uso del API**: La velocidad de reacción también puede mejorar el rendimiento de los modelos. La respuesta de 20 Hz de la inteligencia artificial dio lo necesario para los resultados conseguidos. No obstante, si se intenta aumentar la frecuencia, pueden llegar a surgir inconvenientes debido al uso de la API. Con respecto a esto, pueden surgir dos problemas principales: 1) El primero consiste en que el envío de comandos a los motores del robot con la API de pymycobot es muy lento en comparación de otros métodos. Por ejemplo, para enviar un comando a los motores con la API puede llegar a tardar hasta 0.025 segundos, mientras que otras formas de comunicación consiguen 0.01 segundos fácilmente. 2) El segundo problema es que, a frecuencias altas, la lectura de los encoders de los motores puede llegar a fallar, haciendo imposible la retroalimentación del estado del robot. Por estos dilemas es que se debe proponer una forma más rápida de comunicación con los motores.

Como se mencionó en el planteamiento del problema del trabajo, se propusieron dos paradigmas de visión computacional para procesar los estados del ambiente. En el trabajo se desarrolló la solución por detección de objetos con YOLO v7 y usando una política de perceptrón multi-capa; no obstante, la solución de utilizar solamente la red neuronal convolucional como política sigue disponible para su desarrollo. Para lograr el uso de la red, se deben considerar tres aspectos fundamentales: el primero es crear o conseguir un ambiente que simule lo más cercano la imagen producida por Beat Saber para poder implementar la simulación acelerada. El segundo aspecto es disponer de un hardware potente para poder llevar a cabo el largo entrenamiento que requieren este tipo de redes. El tercero es crear una representación que exprese bien el estado del robot que pueda ser concatenada a la red convolucional.

De crear una política exitosa con esta estrategia, se podrían optimizar recursos. Por ejemplo, ya no se necesitaría la segunda computadora que procesa YOLO, ya que la imagen pasaría directamente a la JetsonNano para ser procesada por la red neuronal convolucional.

Todo el código del proyecto se encuentra subido a GitHub[1]. Aun así, en este trabajo se incluye en el apéndice A, el código del ambiente con el fin de demostrar los componentes del aprendizaje por refuerzo profundo.

[1] https://github.com/AlexisAguileraValderrama/BeatSaber_robot/tree/master

Apéndice A

Código

```
from collections import OrderedDict
from typing import Any, Dict, Optional, Tuple, Union

import numpy as np
from gymnasium import Env, spaces
from gymnasium.envs.registration import EnvSpec

from stable_baselines3.common.type_aliases import GymStepReturn

import rospy
from std_msgs.msg import Int16MultiArray, Float32MultiArray

from sensor_msgs.msg import  JointState
from geometry_msgs.msg import PoseArray

from image_tracker.msg import CubeArray_msg
import time

import math

from threading import Thread
import datetime
from numpy.linalg import norm

import pickle

class BeatSaberEnvMulti(Env):
    spec = EnvSpec("BeatSaberEnvMulti-v0", "no-entry-point")
```

```
    state: np.ndarray

    def status_publisher_function(self):
        while not self.detener:
            msg_ia_status = Int16MultiArray()
            msg_ia_status.data = [self.ia_status]
            self.pub_ia_status.publish(msg_ia_status)
            time.sleep(0.005)

    def recieve_info(self,data):
      if self.in_step == False:
        self.cube_info = data

    def recieve_controller_info(self,data):
      if self.in_step == False:
        self.controller_info = data

    def recieve_light_points(self,data):
      if self.in_step == False:
        self.origin_point = data.poses[0]
        self.end_point = data.poses[1]
        self.delta_vector = data.poses[2]

    def recieve_robot_joint_state(self,data):
      if self.in_step == False:
        self.robot_joint_state = data
        self.joint_seq = self.joint_seq + 1

    def __init__(
            self,
            num_cubes = 4,
            multi_num = None,
            continous_space = True,
            arrows = True
    ):
        super().__init__()
        self.num_cubes = num_cubes
        self.multi_num = multi_num
        self.continous_space = continous_space

        self.detener = False

        self.arrows = arrows

```

```
        if arrows:
            self.num_caract = 6
            self.hit_reward = 4
            self.cutted_reward = 7
            self.failed_reward = -0.1
        else:
            self.num_caract = 3
            self.hit_reward = 7
            self.cutted_reward = 7
            self.failed_reward = -0.4

        self.accel_time = 1

        self.step_count = 0
        self.step_count_total = 0

        self.correct_count = 1
        self.correct_cut_count = 0
        self.failed_count = 0

        self.in_step = False
        self.total_reward = 0
        self.last_reward = 0

        self.cube_info = None

        self.origin_point = None
        self.end_point = None
        self.delta_vector = None

        self.robot_joint_state = None
        self.joint_seq = 0

        self.focused_cube = None
        self.neuron_cubes = []
        self.discarded_cubes = []
        self.discard_capacity = 30

        self.controller_info = None

        self.ia_status = 0

        self.last_time = 0

```

```
117         self.reward_dist = 0
118         self.reward_sim = 0
119         self.reward_dist2d = 0
120         self.reward_green = 0
121         self.reward_red = 0
122         self.reward_blue = 0
123         self.reward_sim_neg = 0
124 
125         self.ratio = 1
126 
127         self.val_range_dist = (0,1)
128         self.function_dist = lambda x : 0.55 - x
129 
130         self.val_range_dist_2d = (0,1)
131         self.function_dist_2d = lambda x : 0.2 - x
132 
133         self.val_range_angle = (0,1)
134         self.function_angle = lambda x : 1.5*x
135 
136         self.pub_action = rospy.Publisher('IA_actions_'
137                                           +str(multi_num),Int16MultiArray,
138                                             queue_size=10)
139         self.pub_ia_status = rospy.Publisher("IA_status_"+
140                                              str(multi_num),
141                                              Int16MultiArray, queue_size=10)
142 
143         rospy.Subscriber("/cubes_info_"+str(multi_num),
144                          CubeArray_msg, self.recieve_info)
145         rospy.Subscriber("/mycobot_jointstates_"+str(multi_num),
146                          JointState, self.recieve_robot_joint_state)
147         rospy.Subscriber("/light_points_"+str(multi_num),
148                          PoseArray, self.recieve_light_points)
149         rospy.Subscriber("/velocity_controller_"+str(multi_num),
150                          Float32MultiArray, self.recieve_controller_info)
151 
152         self.status_thread = Thread(target=self.status_publisher_function)
153         self.status_thread.start()
154 
155         print("Getting light points nodes")
156         while self.origin_point is None:
157           time.sleep(0.2)
158 
159         print("Waiting game controller info")
160         while self.controller_info is None:
```

```
          print("ESperando")
          time.sleep(0.2)

        print("Waiting for robot joint state node")
        while self.robot_joint_state is None:
          time.sleep(0.2)

        print("Waiting for cubes info node")
        while self.cube_info is None:
          time.sleep(0.2)

        #Actions
        if continous_space:
            self.action_space = spaces.Box(low=-1, high=1,
                                           shape=(6,), dtype=np.float32)
        else:
           self.action_space = spaces.MultiDiscrete([3,3,3,3,3,3])

        self.observation_space = spaces.Dict(
                    {
                        "observation": spaces.Box(
                            low=-5,
                            high=5,
                            shape=(18 + (self.num_cubes) * self.num_caract,), # x y z
                            dtype=np.float32,
                        ),
                        "achieved_goal": spaces.Box(
                            low=-5,
                            high=5,
                            shape=(7,),
                            dtype=np.float32,
                        ),
                        "desired_goal": spaces.Box(
                            low=-5,
                            high=5,
                            shape=(7,),
                            dtype=np.float32,
                        ),
                    }
                )

    def seed(self, seed: int) -> None:
        self.obs_space.seed(seed)

```

```
205
206     def get_value_reward(self,x,function, val_range):
207
208         if x <= val_range[0]:
209            return function(val_range[0])
210         if x >= val_range[1]:
211            return function(val_range[1])
212
213         return function(x)
214
215
216     def _get_obs(self) -> Dict[str, Union[float, np.ndarray]]:
217
218         self.desired_goal = [-0.4,0.0,0.3, 0,0,0,  0]
219
220         median_x = 3*(self.end_point.position.x - self.origin_point.position.x)/4.0 \
221                     + self.origin_point.position.x
222         median_y = 3*(self.end_point.position.y - self.origin_point.position.y)/4.0 \
223                     + self.origin_point.position.y
224         median_z = 3*(self.end_point.position.z - self.origin_point.position.z)/4.0 \
225                     + self.origin_point.position.z
226
227         self.achieved_goal = [median_x,
228                               median_y,
229                               median_z,
230                               self.delta_vector.position.x,
231                               self.delta_vector.position.y,
232                               self.delta_vector.position.z,
233                               0]
234
235         if self.focused_cube is not None:
236             self.desired_goal = [self.focused_cube.pose.position.x,
237                                  self.focused_cube.pose.position.y,
238                                  self.focused_cube.pose.position.z,
239                                  self.focused_cube.cut_vector.x,
240                                  self.focused_cube.cut_vector.y,
241                                  self.focused_cube.cut_vector.z,
242                                  1]
243
244             self.achieved_goal = [median_x,
245                                  median_y,
246                                  median_z,
247                                  self.delta_vector.position.x,
248                                  self.delta_vector.position.y,
249                                  self.delta_vector.position.z,
```

```
                                self.focused_cube.status]

        info_cubes = [[cu.pose.position.x,
                       cu.pose.position.y,
                       cu.pose.position.z,
                       cu.cut_vector.x,
                       cu.cut_vector.y,
                       cu.cut_vector.z,
                       cu.status] for cu in self.neuron_cubes]

        concat=[]
        for i in range(self.num_cubes):
            if i < len(info_cubes):
                concat.extend(info_cubes[i][:self.num_caract]) ##### quitar arrow
            else:
                concat.extend([0,0,0 ,0,0,0 ,0][:self.num_caract]) #####quitar arrow

        observation = np.concatenate((np.array([x for x in \
                                    self.robot_joint_state.velocity]),
                                    np.array(self.robot_joint_state.position),
                                    np.array([x for x in self.controller_info.data]),
                                    np.array(self.desired_goal[:self.num_caract]),
                                    np.array(concat[self.num_caract:])), axis=None)
        return OrderedDict(
            [
                ("observation", observation),
                ("achieved_goal",np.array(self.achieved_goal)),
                ("desired_goal", np.array(self.desired_goal)),
            ]
        )

    def reset(
        self, *, seed: Optional[int] = None, options: Optional[Dict] = None
    ) -> Tuple[Dict[str, Union[float, np.ndarray]], Dict]:

        self.in_step = True

        if seed is not None:
            self.obs_space.seed(seed)

        self.step_count = 0

        self.focused_cube_id = -1
        self.focused_cube = None
```

```
294
295         obs = self._get_obs()
296
297         self.in_step = False
298
299         if self.accel_time > 1.5:
300             return obs, {}
301         else:
302            return obs
303
304     def step(self, action) -> GymStepReturn:
305
306         current_time = datetime.datetime.now()
307         if self.last_time == 0:
308             self.last_time = current_time
309
310         delta_time = (current_time - self.last_time).microseconds/1000000
311
312         self.ia_status = 1
313         self.in_step = True
314
315         if self.continous_space:
316            discrete_actions = np.round(action,0).astype(int)
317         else:
318            discrete_actions = action - [1,1,1,1,1,1]
319
320         message = Int16MultiArray()
321         message.data = discrete_actions
322         self.pub_action.publish(message)
323
324         self.focused_cube = None
325         self.neuron_cubes = []
326         anomaly = 0
327
328         for cube in self.cube_info.cubes:
329             if cube.id not in self.discarded_cubes:
330                 self.neuron_cubes.append(cube)
331
332                 if len(self.neuron_cubes) == 1:
333                     self.focused_cube = cube
334                     self.focused_cube_id = cube.id
335
336                 if cube.status != 0:
337                    anomaly = cube.status
338                    self.discarded_cubes.append(cube.id)
```

```
339
340             if len(self.neuron_cubes) == self.num_cubes:
341                 break
342
343         if len(self.discarded_cubes) >= self.discard_capacity:
344             self.discarded_cubes.pop(0)
345
346         observation = self._get_obs()
347
348         reward = self.compute_reward(self.achieved_goal,self.desired_goal,{})
349
350         terminated = False
351
352         if self.achieved_goal[6] == 1 or anomaly == 1: #Le dio
353             reward += self.hit_reward
354             self.reward_green += self.hit_reward
355             self.correct_count += 1
356             terminated = True
357         elif self.achieved_goal[6] == 2 or anomaly == 2: #LO corto
358             reward += self.cutted_reward
359             self.reward_blue += self.cutted_reward
360             self.correct_cut_count += 1
361             terminated = True
362         elif self.achieved_goal[6] == -1 or anomaly == -1: #No le dio
363             reward += self.failed_reward
364             self.reward_red += self.failed_reward
365             self.failed_count += 1
366             terminated = True
367
368         truncated = False
369
370         self.step_count += 1
371         self.step_count_total += 1
372
373         info = {"is_success": terminated}
374
375         self.ratio = (self.correct_count + \
376                       self.correct_cut_count)/ \
377                       (self.correct_count + \
378                        self.correct_cut_count + \
379                        self.failed_count)
380
381         print(f'---------------------------------------------------------- \n' + \
382               f'Step:  {self.step_count} \n' + \
383               f'Total Step:  {self.step_count_total} \n' + \
```

```
            f'Hited:  {self.correct_count} \n' + \
            f'Cutted:  {self.correct_cut_count} \n' + \
            f'Failed:  {self.failed_count} \n' + \
            f'Robot Joint: {self.robot_joint_state.position} \n' + \
            f'Robot Velocity: {[ x/self.accel_time
                                for x in self.robot_joint_state.velocity]} \n' + \
            f'Controller velocities: {[ x/self.accel_time
                                       for x in self.controller_info.data]} \n' + \
            #f'Observation: {observation["observation"]} \n' + \
            f'achieved_goal: {observation["achieved_goal"]} \n' + \
            f'desired_goal: {observation["desired_goal"]} \n' + \
            f'Cubos encontrados: {len(self.neuron_cubes)}'
            f'Cube id esperado: {self.focused_cube_id} \n' + \
            f'Cube id encontrado: {self.focused_cube.id \
                                 if self.focused_cube is not None \
                                 else "No encontrado"} \n' + \
            f'Coordenadas focus: {self.desired_goal} \n '+ \
            f'reward in step: {reward} \n' + \
            f'reward dist: {self.reward_dist} \n' + \
            f'reward dist2d: {self.reward_dist2d} \n' + \
            f'reward sim: {self.reward_sim} \n' + \
            f'reward sim_neg: {self.reward_sim_neg} \n' + \
            f'reward green: {self.reward_green} \n' + \
            f'reward blue: {self.reward_blue} \n' + \
            f'reward red: {self.reward_red} \n' + \
            f'ratio : {self.ratio} \n' + \
            f'Actions: {discrete_actions} \n' )

        self.in_step = False

        self.ia_status = 0 if terminated else 1

        self.total_reward += reward
        self.last_reward = reward

        if delta_time == 0:
            print("esperando comparacion")
        else:
            print(1.0/delta_time)

        self.last_time = current_time

        time.sleep(0.04/self.accel_time)

```

```
428         return observation, reward, terminated, truncated, info
429
430     def compute_reward(
431         self, achieved_goal: Union[float, np.ndarray],
432         desired_goal: Union[float, np.ndarray],
433         _info: Optional[Dict[str, Any]]
434     ) -> np.float32:
435
436         arr_achieved = np.array(achieved_goal,ndmin=1)
437         arr_desired = np.array(desired_goal,ndmin=1)
438
439         print(f'++++++++++++++++++++++++++++++++++++++++++++ \n' + \
440               f'achieved_goal: {arr_achieved} \n' + \
441               f'desired_goal: {arr_desired} \n' )
442
443         in_step = True
444         print(arr_achieved.shape)
445         if arr_achieved.shape != (7,):
446            arr_achieved = arr_achieved[0]
447            arr_desired = arr_desired[0]
448            in_step = False
449
450         x1 = arr_achieved[0]
451         y1 = arr_achieved[1]
452         z1 = arr_achieved[2]
453
454         x2 = arr_desired[0]
455         y2 = arr_desired[1]
456         z2 = arr_desired[2]
457
458         reward_sim = 0
459         reward_dist = 0
460         reward_dist2d = 0
461
462         dist = math.sqrt((x2-x1)**2 + (y2-y1)**2 + (z2-z1)**2)
463         dist2d = math.sqrt((y2-y1)**2 + (z2-z1)**2)
464
465         reward_dist = self.get_value_reward(dist,
466                                     self.function_dist,
467                                     self.val_range_dist)
468
469
470         print(f"dist: {dist}")
471         print(f"reward_dist: {reward_dist}")
```

```
472
473         reward_dist2d = self.get_value_reward(dist2d,
474                                   self.function_dist_2d,
475                                   self.val_range_dist_2d)
476         sim = 0
477
478         if dist < 0.1 and self.arrows:
479             delta_array = arr_achieved[3:6]
480             cube_cut_array = arr_desired[3:6]
481
482             norm_delta = norm(delta_array)
483             norm_cube_cut = norm(cube_cut_array)
484
485             if norm_delta == 0:
486                 if norm_cube_cut == 0:
487                     pass
488                 else:
489                     pass
490             else:
491                 if norm_cube_cut == 0:
492                     pass
493                 else:
494                     sim = np.dot(delta_array,cube_cut_array) / \
495                         (norm_delta*norm_cube_cut)
496                     reward_sim = self.get_value_reward(sim,
497                                       self.function_angle,
498                                       self.val_range_angle)
499
500         reward = reward_dist + reward_dist2d + reward_sim
501
502         if in_step:
503            self.reward_dist += reward_dist
504            self.reward_dist2d += reward_dist2d
505            self.reward_sim += reward_sim
506            self.reward_sim_neg += reward_sim if reward_sim < 0 else 0
507            print("In step")
508
509         print(type(reward))
510
511         reward = np.array([reward], dtype=np.float32)
512
513         print(f"distancia: {dist}")
514         print(f"sim_reward {reward_sim}")
515         print(f" - reward: {reward}")
516
```

```
517         return reward
518
519     def render(self) -> Optional[np.ndarray]:  # type: ignore[override]
520         if self.render_mode == "rgb_array":
521             return self.state.copy()
522         print(self.state)
523         return None
524
525     def write_report(self, path_name):
526
527         report = {"robot_id" : self.multi_num,
528                   "reward":self.total_reward,
529                   "last_reward":self.last_reward,
530                   "cut":self.correct_cut_count,
531                   "hitted":self.correct_count,
532                   "failed" : self.failed_count,
533                   "ratio":self.ratio}
534
535         file = open(path_name,'wb')
536         pickle.dump(report,file)
537         file.close()
538
539     def full_reset(self, kill_status_thread):
540         self.reset()
541         self.ia_status = -1
542         time.sleep(8)
543
544         if kill_status_thread:
545             self.detener = True
```

Referencias

[1] Seyed Sajad Mousavi, Michael Schukat y Enda Howley. "Deep reinforcement learning: An overview". En: *Proceedings of SAI Intelligent Systems Conference (IntelliSys) 2016*. Lecture notes in networks and systems. Cham: Springer International Publishing, 2018, págs. 426-440. DOI: https://doi.org/10.1007/978-3-319-56991-8_32.

[2] Marcin Andrychowicz et al. "Hindsight Experience Replay". En: *Advances in Neural Information Processing Systems*. Ed. por I. Guyon et al. Vol. 30. Curran Associates, Inc., 2017. URL: https://proceedings.neurips.cc/paper_files/paper/2017/file/453fadbd8a1a3af50a9df4df899537b5-Paper.pdf.

[3] Petar Kormushev, Sylvain Calinon y Darwin Caldwell. "Reinforcement learning in robotics: Applications and real-world challenges". En: *Robotics* 2.3 (jul. de 2013), págs. 122-148. DOI: https://doi.org/10.3390/robotics2030122.

[4] Saminda Wishwajith Abeyruwan et al. "i-sim2real: Reinforcement learning of robotic policies in tight human-robot interaction loops". En: *Conference on Robot Learning*. PMLR. 2023, págs. 212-224.

[5] Kevin Zakka et al. "Robopianist: Dexterous piano playing with deep reinforcement learning". En: *7th Annual Conference on Robot Learning*. 2023.

[6] Andy Zeng et al. "Tossingbot: Learning to throw arbitrary objects with residual physics". En: *IEEE Transactions on Robotics* 36.4 (2020), págs. 1307-1319.

[7] Alex Church et al. "Deep reinforcement learning for tactile robotics: Learning to type on a braille keyboard". En: *IEEE Robotics and Automation Letters* 5.4 (2020), págs. 6145-6152.

[8] Dong-Ok Won, Klaus-Robert Müller y Seong-Whan Lee. "An adaptive deep reinforcement learning framework enables curling robots with human-like performance in real-world conditions". En: *Science Robotics* 5.46 (2020), eabb9764.

[9] Saminda Wishwajith Abeyruwan et al. "i-sim2real: Reinforcement learning of robotic policies in tight human-robot interaction loops". En: *Conference on Robot Learning*. PMLR. 2023, págs. 212-224.

[10] Jonas Tebbe et al. "Sample-efficient reinforcement learning in robotic table tennis". En: *2021 IEEE international conference on robotics and automation (ICRA)*. IEEE. 2021, págs. 4171-4178.

[11] Boling Yang et al. "Competitive physical human-robot game play". En: *Companion of the 2021 ACM/IEEE International Conference on Human-Robot Interaction*. 2021, págs. 242-246.

[12] Guillaume Lample y Devendra Singh Chaplot. "Playing FPS games with deep reinforcement learning". En: *Proc. Conf. AAAI Artif. Intell.* 31.1 (feb. de 2017). DOI: https://doi.org/10.1609/aaai.v31i1.10827.

[13] Volodymyr Mnih et al. "Playing Atari with deep reinforcement learning". En: (dic. de 2013). DOI: https://doi.org/10.48550/arXiv.1312.5602. arXiv: 1312.5602 [cs.LG].

[14] OpenAI et al. "Dota 2 with large scale deep reinforcement learning". En: (2019). DOI: https://doi.org/10.48550/arXiv.1912.06680.

[15] Google DeepMind. *A generalist AI agent for 3D virtual environments*. en. https://deepmind.google/discover/blog/sima-generalist-ai-agent-for-3d-virtual-environments/. Accessed: 2024-4-3.

[16] Yoonsin Oh y Stephen Yang. "Defining exergames & exergaming". En: *Proceedings of meaningful play* 2010 (2010), págs. 21-23.

[17] Mateus David Finco y Richard Wilhelm Maass. "The history of exergames: promotion of exercise and active living through body interaction". En: *2014 IEEE 3nd*

International Conference on Serious Games and Applications for Health (SeGAH). IEEE. 2014, págs. 1-6.

[18] Virtual Reality Institute of Health y Exercise. "VR exercise ratings". En: (2017). URL: https://vrhealth.institute/vr-ratings/.

[19] Jens Kober, J Andrew Bagnell y Jan Peters. "Reinforcement learning in robotics: A survey". En: *The International Journal of Robotics Research* 32.11 (2013), págs. 1238-1274.

[20] Anssi Kanervisto. "Advances in deep learning for playing video games". Tesis doct. Itä-Suomen yliopisto, 2022.

[21] Jianhao Fang et al. "Deep reinforcement learning enhanced convolutional neural networks for robotic grasping". En: *Volume 3B: 47th Design Automation Conference (DAC)*. Virtual, Online: American Society of Mechanical Engineers, ago. de 2021. DOI: https://doi.org/10.1115/DETC2021-67225.

[22] Natanael Magno Gomes et al. "Deep reinforcement learning applied to a robotic pick-and-place application". En: *Communications in Computer and Information Science*. Communications in computer and information science. Cham: Springer International Publishing, 2021, págs. 251-265. DOI: https://doi.org/10.1007/978-3-030-91885-9_18.

[23] Stephen James y Edward Johns. "3D simulation for robot arm control with deep Q-learning". En: (2016). DOI: https://doi.org/10.48550/arXiv.1609.03759.

[24] Minoru Sasaki et al. "Sim–real mapping of an image-based robot arm controller using Deep Reinforcement learning". en. En: *Appl. Sci. (Basel)* 12.20 (oct. de 2022), pág. 10277. DOI: https://doi.org/10.3390/app122010277.

[25] Deirdre Quillen et al. "Deep reinforcement learning for vision-based robotic grasping: A simulated comparative evaluation of off-policy methods". En: *2018 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 2018, págs. 6284-6291. DOI: https://doi.org/10.48550/arXiv.1802.10264.

[26] Fangyi Zhang et al. "Towards vision-based deep reinforcement learning for robotic motion control". En: (2015). DOI: https://doi.org/10.48550/arXiv.1511.03791.

[27] Hiba Sekkat et al. "Vision-based robotic arm control algorithm using deep reinforcement learning for autonomous objects grasping". en. En: *Appl. Sci. (Basel)* 11.17 (ago. de 2021), pág. 7917. DOI: https://doi.org/10.3390/app11177917.

[28] Márton Szemenyei y Vladimir Estivill-Castro. "Fully neural object detection solutions for robot soccer". en. En: *Neural Comput. Appl.* 34.24 (dic. de 2022), págs. 21419-21432. DOI: https://doi.org/10.1007/s00521-021-05972-1.

[29] Xiaowei Xing y Dong Eui Chang. "Deep reinforcement learning based robot arm manipulation with efficient training data through simulation". En: *2019 19th International Conference on Control, Automation and Systems (ICCAS)*. IEEE. 2019, págs. 112-116. DOI: https://doi.org/10.48550/arXiv.1907.06884.

[30] Hiba Sekkat et al. "Vision-based robotic arm control algorithm using deep reinforcement learning for autonomous objects grasping". En: *Applied Sciences* 11.17 (2021), pág. 7917. DOI: https://doi.org/10.3390/app11177917.

[31] Lei Tai y Ming Liu. "Towards cognitive exploration through deep reinforcement learning for mobile robots". En: (oct. de 2016). DOI: http://export.arxiv.org/pdf/1610.01733. arXiv: 1610.01733 [cs.RO].

[32] Svemir Popi y Branko Miloradovi. "Light weight robot arms-an overview". En: *INFOTEH-JAHORINA* 14 (2015).

[33] Zhenli Lu et al. "A brief survey of commercial robotic arms for research on manipulation". En: *2012 IEEE Symposium on Robotics and Applications (ISRA)*. IEEE. 2012, págs. 986-991.

[34] Eric Eaton et al. "Design of a low-cost platform for autonomous mobile service robots". En: *IJCAI workshop on autonomous mobile service robots*. 2016.

[35] Jehangir Arshad et al. "Intelligent Control of Robotic Arm Using Brain Computer Interface and Artificial Intelligence". En: *Applied Sciences* 12.21 (2022), pág. 10813.

[36] Shang Gao et al. "A Hybrid Navigation and Image Processing Model with Dynamic Mobile Manipulation". En: *2019 IEEE International Conference on Robotics and Biomimetics (ROBIO)*. IEEE. 2019, págs. 2175-2180.

[37] Kirk Duran. "Deep Reinforcement Learning for a Four Degree of Freedom Robot Arm Control Simulation accelerated by Human Demonstrations". Tesis doct. Sonoma State University, 2021.

[38] Andini Pratiwi, Erna Budhiarti Nababan y Amalia. "Detection of the use of mask to prevent the spread of COVID-19 using SVM, Haar Cascade Classifier, and robot arm". En: *Data Science: J. of Computing and Appl. Informatics* 6.2 (jul. de 2022), págs. 125-137. DOI: https://doi.org/10.32734/jocai.v6.i2-9289.

[39] Wagner Augusto Aranda Cotta, Sérgio Ivan Lopes y Raquel Frizera Vassallo. "Towards the cognitive factory in industry 5.0: From concept to implementation". en. En: *Smart Cities* 6.4 (ago. de 2023), págs. 1901-1921. DOI: https://doi.org/10.3390/smartcities6040088.

[40] Laijun Yang et al. "Vision-based robot arm control interface for retrieving objects from the floor". en. En: *J. Robot. Mechatron.* 35.2 (abr. de 2023), págs. 501-509. DOI: https://doi.org/10.20965/jrm.2023.p0501.

[41] Pilsung Kang y Jongmin Jo. "Benchmarking modern edge devices for AI applications". en. En: *IEICE Trans. Inf. Syst.* E104.D.3 (mar. de 2021), págs. 394-403. DOI: https://doi.org/10.1587/transinf.2020edp7160.

[42] Stephan Patrick Baller et al. "DeepEdgeBench: Benchmarking Deep Neural Networks on Edge Devices". En: *2021 IEEE International Conference on Cloud Engineering (IC2E)*. 2021, págs. 20-30. DOI: 10.1109/IC2E52221.2021.00016.

[43] Trenton H Stewart et al. "Actual vs. perceived exertion during active virtual reality game exercise". en. En: *Front. Rehabil. Sci.* 3 (ago. de 2022), pág. 887740. DOI: https://doi.org/10.3389/fresc.2022.887740.

[44] Eric Evans et al. "Physical activity intensity, perceived exertion, and enjoyment during head-mounted display virtual reality games". en. En: *Games Health* 10.5 (oct. de 2021), págs. 314-320. DOI: https://doi.org/10.1089/g4h.2021.0036.

[45] Ja Steeves et al. "EXERCISE INTENSITY CLASSIFICATION OF ACTIVE VIRTUAL REALITY GAMES IN YOUTH (8-12)". En: *International Journal of Exercise Science* 16.2 (2023).

[46] Alan Yates y HACKADAY. *Alan Yates on the Impossible Task of Making Valve's VR Work*. Youtube. 2016. URL: https://www.youtube.com/watch?v=75ZytcYANTA&t.

[47] *Use the World's Best Virtual Reality Technology, Royalty Free*. SteamVR. URL: https://partner.steamgames.com/vrlicensing.

[48] Sebastian Rutkowski et al. "Training using a commercial immersive virtual reality system on hand–eye coordination and reaction time in young musicians: A pilot study". En: *International journal of environmental research and public health* 18.3 (2021), pág. 1297.

[49] Tuong Thai. "The Influence Of Exergaming On Heart Rate, Perceived Exertion, Motivation To Exercise". En: *And Time Spent Exercising, Honor Thesis* (2019).

[50] *Dexterity*. En: *Cambridge Dictionary*. Cambridge University. URL: https://dictionary.cambridge.org/us/dictionary/english/dexterity.

[51] Aude Billard y Danica Kragic. "Trends and challenges in robot manipulation". En: *Science* 364.6446 (2019), eaat8414.

[52] Raymond R Ma y Aaron M Dollar. "On dexterity and dexterous manipulation". En: *2011 15th International Conference on Advanced Robotics (ICAR)*. IEEE. 2011, págs. 1-7.

[53] Marian Körber et al. "Comparing popular simulation environments in the scope of robotics and reinforcement learning". En: (2021). DOI: https://doi.org/10.48550/arXiv.2103.04616.

[54] Madison Elias et al. "Performance Optimization Analysis of Robotic Operating System ROS Communication". En: *International Journal of Youth Science and Technology Innovation* 1.1 (2023).

[55] Suzana Herculano-Houzel. "The human brain in numbers: a linearly scaled-up primate brain". En: *Frontiers in human neuroscience* 3 (2009), pág. 857.

[56] "The perceptron: a probabilistic model for information storage and organization in the brain." En: *Psychological review* 65.6 (1958), pág. 386.

[57] Simon O Haykin. *Neural Networks and Learning Machines*. 3.ª ed. Upper Saddle River, NJ: Pearson, nov. de 2008.

[58] Yann LeCun et al. "Gradient-based learning applied to document recognition". En: *Proceedings of the IEEE* 86.11 (1998), págs. 2278-2324.

[59] Timothy P Lillicrap et al. "Continuous control with deep reinforcement learning". En: (2015). DOI: https://doi.org/10.48550/arXiv.1509.02971.

[60] Scott Fujimoto, Herke van Hoof y David Meger. "Addressing function approximation error in actor-critic methods". En: (2018). DOI: https://doi.org/10.48550/arXiv.1509.02971.

[61] John Schulman et al. "Proximal Policy Optimization Algorithms". En: (2017). DOI: https://doi.org/10.48550/arXiv.1707.06347.

[62] Diederik P Kingma y Jimmy Ba. "Adam: A method for stochastic optimization". En: *arXiv preprint arXiv:1412.6980* (2014). DOI: arXiv:1412.6980.

[63] Richard S Sutton y Andrew G Barto. *Reinforcement Learning*. 2.ª ed. Adaptive Computation and Machine Learning series. Cambridge, MA: Bradford Books, nov. de 2018.

[64] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". en. En: *Nature* 518.7540 (feb. de 2015), págs. 529-533. DOI: https://doi.org/10.1038/nature14236.

[65] Long-Ji Lin. *Reinforcement learning for robots using neural networks*. Carnegie Mellon University, 1992.

[66] Pierre Aumjaud et al. "Reinforcement learning experiments and benchmark for solving robotic reaching tasks". En: *Workshop of Physical Agents*. Springer. 2020, págs. 318-331.

[67] A. Rupam Mahmood et al. "Benchmarking Reinforcement Learning Algorithms on Real-World Robots". En: *Proceedings of The 2nd Conference on Robot Learning*. Ed. por Aude Billard et al. Vol. 87. Proceedings of Machine Learning Research. PMLR, 29–31 Oct de 2018, págs. 561-591. URL: https://proceedings.mlr.press/v87/mahmood18a.html.

[68] Yan Duan et al. "Benchmarking Deep Reinforcement Learning for Continuous Control". En: *Proceedings of The 33rd International Conference on Machine Learning*. Ed. por Maria Florina Balcan y Kilian Q. Weinberger. Vol. 48. Proceedings of Machine Learning Research. New York, New York, USA: PMLR, 20–22 Jun de 2016, págs. 1329-1338. URL: https://proceedings.mlr.press/v48/duan16.html.

[69] Antonin Raffin et al. "Stable-Baselines3: Reliable Reinforcement Learning Implementations". En: *Journal of Machine Learning Research* 22.268 (2021), págs. 1-8. URL: http://jmlr.org/papers/v22/20-1364.html.

[70] YANN LECUN et al. "Gradient-Based Learning Applied to Document Recognition". En: *PROCEEDINGS OF THE IEEE* 86.11 (1998). DOI: 10.1109/5.726791.

[71] Debani Prasad Mishra et al. "Various object detection algorithms and their comparison". En: 29.1 (ene. de 2022), pág. 330. DOI: 10.11591/ijeecs.v29.i1.pp330-338.

[72] Martinus Grady Naftali, Jason Sebastian Sulistyawan y Kelvin Julian. "Comparison of object detection algorithms for street-level objects". En: (2022). DOI: https://doi.org/10.48550/arXiv.2208.11315.

[73] Shrey Srivastava et al. "Comparative analysis of deep learning image detection algorithms". en. En: *J. Big Data* 8.1 (dic. de 2021). DOI: https://doi.org/10.1186/s40537-021-00434-w.

[74] Joseph Redmon et al. "You Only Look Once: Unified, Real-Time Object Detection". En: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. Jun. de 2016.

[75] Chien-Yao Wang, Alexey Bochkovskiy y Hong-Yuan Mark Liao. "YOLOv7: Trainable bag-of-freebies sets new state-of-the-art for real-time object detectors". En: (2022). DOI: https://doi.org/10.48550/arXiv.2207.02696.

[76] Kevin M Lynch y Frank C Park. *Modern robotics: Mechanics, planning and control*. en. Cambridge, England: Cambridge University Press, mayo de 2017.

[77] H Bruyninckx et al. "A roadmap for autonomous robotic assembly". En: *Proceedings of the 2001 IEEE International Symposium on Assembly and Task Planning (ISATP2001). Assembly and Disassembly in the Twenty-first Century. (Cat. No.01TH8560)*. Fukuoka, Japan: IEEE, 2002. DOI: 10.1109/ISATP.2001.928965.

[78] Xueyou Zhang et al. "Reinforcement learning-based hierarchical control for path following of a salamander-like robot". En: *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. Las Vegas, NV, USA: IEEE, oct. de 2020. DOI: 10.1109/IROS45743.2020.9341656.

[79] Wei Zhu, Fahad Raza y Mitsuhiro Hayashibe. "Reinforcement learning based hierarchical control for path tracking of a wheeled bipedal robot with Sim-to-real framework". En: *2022 IEEE/SICE International Symposium on System Integration (SII)*. Narvik, Norway: IEEE, ene. de 2022. DOI: 10.1109/SII52469.2022.9708882.

[80] Wenshuai Zhao, Jorge Pena Queralta y Tomi Westerlund. "Sim-to-real transfer in deep reinforcement learning for robotics: A survey". En: *2020 IEEE Symposium Series on Computational Intelligence (SSCI)*. Canberra, Australia: IEEE, dic. de 2020. DOI: https://doi.org/10.1109/SSCI47803.2020.9308468.

[81] Shirin Joshi, Sulabh Kumra y Ferat Sahin. "Robotic Grasping using Deep Reinforcement Learning". En: (2020). DOI: https://doi.org/10.48550/arXiv.2007.04499.

[82] Zhen Liang et al. "Parallel gym gazebo: a scalable parallel robot deep reinforcement learning platform". En: *2019 IEEE 31st International Conference on Tools with Artificial Intelligence (ICTAI)*. IEEE. 2019, págs. 206-213. DOI: 10.1109/ICTAI.2019.00037.

[83] Jayasekara Kapukotuwa et al. "MultiROS: ROS-Based Robot Simulation Environment for Concurrent Deep Reinforcement Learning". En: *2022 IEEE 18th International Conference on Automation Science and Engineering (CASE)*. IEEE. 2022, págs. 1098-1103. DOI: 10.1109/CASE49997.2022.9926475.

[84] Zhengyou Zhang. "A flexible new technique for camera calibration". En: *IEEE Transactions on pattern analysis and machine intelligence* 22.11 (2000), págs. 1330-1334. DOI: 10.1109/34.888718.