Reinforcement y Deep Learning en Aplicaciones de Robótica de Enjambre

Eduardo Andrés Santizo Olivet

UVG UNIVERSIDAD DEL VALLE DE GUATEMALA

UNIVERSIDAD DEL VALLE DE GUATEMALA
Facultad de Ingeniería

Reinforcement y Deep Learning en Aplicaciones de Robótica de Enjambre

Trabajo de graduación presentado por Eduardo Andrés Santizo Olivet
para optar al grado académico de Licenciado en Ingeniería Mecatrónica

Guatemala,

2020

Vo.Bo.:

(f) _____

Dr. Luis Alberto Rivera Estrada

Tribunal Examinador:

(f) _____

Dr. Luis Alberto Rivera Estrada

(f) _____

(f) _____

Fecha de aprobación: Guatemala, de diciembre de 2020.

Prefacio

La idea del presente trabajo en una sola oración, como el título sugiere, se podría establecer como: La aplicación de técnicas de aprendizaje reforzado y aprendizaje profundo en el área de inteligencia de enjambre y robótica. Como se puede denotar, esto engloba una variedad de destrezas, todas muy diferentes y a la vez íntimamente relacionadas en el actual ambiente académico, donde la inteligencia artificial gana cada vez mayor prominencia.

Debido a esto, debo agradecer a todos los miembros de la comunidad universitaria que me han traído hasta este punto en mi educación, cada uno aportando una pequeña parte de su conocimiento y experiencia, y ayudándome a forjar buena parte de mis actuales intereses. En particular debo agradecer a MSc. Miguel Zea por introducirme al mundo de la robótica y la teoría de control. A a mi asesor de tesis, el Dr. Luis Alberto Rivera, quien, gracias a su curso de *Machine Learning*, reforzó mi interés por el área de aprendizaje automático y fue una gran influencia para la elección del actual tema de este trabajo. A mi compañera de trabajo y amiga, Gabriela Iriarte, por enseñarme a luchar por lo que deseo, permitirme discutir mis ideas y siempre motivarme a dar más de mi. Sin su ayuda, quizá nunca hubiera llegado a pertenecer a la rama de investigación de inteligencia de enjambre.

Finalmente, debo agradecer a mi familia, sin la cual nada de esto hubiera sido posible; no solo por el simple hecho de darme el privilegio de recibir una educación superior de calidad, sino también por todo el apoyo que me han brindado tanto antes como durante el proceso de creación del presente trabajo. No puedo agradecerles lo suficiente por todo lo que me han dado.

Índice

Lista de figuras

Lista de cuadros

Resumen

El área de inteligencia de enjambre busca emular el comportamiento exhibido por diferentes animales que actúan en conjunto, como parvadas de aves, colonias de hormigas o bancos de peces. Muchas son las áreas académicas que han tomado como inspiración este comportamiento, pero dos muy importantes e íntimamente relacionadas son el área de la informática y la robótica.

De aquí que la Universidad del Valle de Guatemala, como parte de la iniciativa del megaproyecto *Robotat*, decidiera emplear el movimiento de las partículas del algoritmo de *Particle Swarm Optimization Algorithm* (PSO)[1] como una guía para el movimiento suave de robots diferenciales [1] alrededor de un ambiente previamente modelado [2].

En el presente trabajo, se tomaron estos avances y se buscó realizar mejoras a los mismos, haciendo uso de técnicas propias de aprendizaje reforzado y profundo. Específicamente, se presentan dos propuestas puntuales: Una mejora al algoritmo PSO utilizando redes neuronales recurrentes y una alternativa al algoritmo de navegación alrededor de un ambiente conocido por medio de programación dinámica (parte de aprendizaje reforzado).

El método empleado para mejorar el desempeño del algoritmo PSO, se denominó *PSO Tuner* y consiste de una red neuronal recurrente que toma diferentes métricas propias de las partículas PSO y las torna, a través de su procesamiento por medio de una red LSTM, GRU o BiLSTM, en una predicción de los hiper parámetros que debería emplear el algoritmo (ω, ϕ_1 y ϕ_2). Dicha predicción es de carácter dinámico, por lo que en cada iteración se generan las métricas que describen al enjambre (dispersión, coherencia, etc.), se alimentan a la red y esta produce los parámetros a utilizar en la siguiente iteración. Las tres arquitecturas propuestas se entrenaron con un total de 7,700 simulaciones del algoritmo estándar PSO. Luego de ajustar debidamente los hiper parámetros de las redes, el *PSO Tuner* fue capaz de reducir el tiempo de convergencia y susceptibilidad a mínimos locales del PSO original, con la arquitectura basada en BiLSTM presentándose como la mejor de las tres alternativas.

Para la alternativa al algoritmo de navegación alrededor de un ambiente conocido, se utilizó como base el ejemplo de programación dinámica *Gridworld* [3]. En este, un agente se mueve a través de un espacio de estados representado en la forma de una cuadrícula.

[1] Basado en un algoritmo de simulación de parvadas de aves

Para movilizarse de estado a estado, el agente puede hacer uso de cuatro acciones: Moverse hacia arriba, abajo, izquierda o derecha. Según su estado actual y la acción tomada, este transiciona a un nuevo estado y recibe una recompensa. El agente buscará maximizar las recompensas obtenidas generando una ruta óptima desde cada estado hasta la meta.

Para ajustar estas ideas al problema de navegación con robots, se proponen algunas modificaciones. En primer lugar, el agente es capaz de moverse diagonalmente a 45 grados. Esto incrementó el número de acciones disponibles de 4 a 8. Luego, el espacio de trabajo se divide en celdas y se escanea secuencialmente para determinar si estas consisten de una celda obstáculo o meta. Finalmente, haciendo uso de *policy iteration* se genera una acción óptima por estado. Estas sugerencias de acción óptimas son luego utilizadas para generar una trayectoria a seguir por los controladores punto a punto de [1]. Este método probó ser una alternativa válida al método de navegación actual basado en *Artificial Potential Fields* y si se optimiza de mejor manera el algoritmo, podría incluso llegar a proponerse como una alternativa válida a métodos de navegación como el algoritmo $A*$ y el algoritmo de *Probabilistic Road Maps* (PRM).

Finalmente, para auxiliar en el proceso de diseño de estas propuestas, se creó un conjunto de funciones, clases y scripts. Este grupo de herramientas (ahora llamadas *Swarm Robotics Toolbox*) proveen al usuario con la capacidad de visualizar pruebas, guardar figuras, generar vídeos, realizar pruebas estadísticas, entre otros. Debido a que estas herramientas están diseñadas para su futuro uso dentro del ámbito educativo, cada parte del Toolbox está debidamente documentada y presenta una descripción más detallada de su funcionamiento y opciones en el repositorio donde el código de este proyecto se encuentra contenido.

Abstract

Swarm intelligence consists of the area of study that tries to artificially emulate the behavior observed in natural groupings of living organisms like schools of fish, ant colonies or bird flocks. Many academic studies have taken inspiration from this type of behavior, but two very important and intimately connected fields are the computer science and robotics fields.

This is why the Universidad del Valle de Guatemala, as part of the *Robotat* mega-project initiative, decided to use the *Particle Swarm Optimization* algorithm as a guide for the smooth movement of differential robots [1] across a previously known environment [2].

In the following work, this idea was improved upon by making use of reinforcement and deep learning techniques. Specifically, two proposals are presented: An improvement to the PSO algorithm using recurrent neural networks and an alternative to the navigation algorithm making use of dynamic programming (part of reinforcement learning).

The method used to improve the performance of the PSO algorithm was named *PSO Tuner* and it consists of a recurrent neural network that takes different metrics from the PSO particles and turns them, through processing by means of an LSTM, GRU or BiLSTM network, into a prediction of the hyper parameters that the PSO algorithm should use (ω, ϕ_1 and ϕ_2). Said prediction is dynamic, so the metrics that describe the swarm (dispersion, coherence, etc.) are generated in each iteration, and then fed to the network to produce the parameters used in the following iteration. The 3 tested architectures were trained with a total of 170,000 simulations of the standard PSO algorithm. After proper tuning, the *PSO Tuner* was able to reduce the convergence time and susceptibility to local minimums of the original algorithm, with the architecture based on BiLSTM cells being presented as the best of the three proposals.

For the alternative navigation algorithm, the classic reinforcement learning toy example *Gridworld* [3] was used as a basis. In this example, an agent moves through a state space represented in the form of a grid. To move from state to state, the agent can make use of one of four actions: Move up, down, left or right. According to its present state and action, the agent transitions to a new state and receives a reward. The agent will seek to maximize the reward received by generating an optimal route from its current position to the goal.

To fit the problem of robot navigation, different modifications are proposed. First, the agent is able to move diagonally at 45 degrees. This bumps the number of available actions to 8. Then, the work space is divided into cells and sequentially scanned to classify each cell (or state) as an *obstacle* or *goal*. Finally, making use of *policy iteration* an optimal action per state is selected.

The optimal actions are then mapped into a trajectory and then followed by the differential robot using the point-point controllers proposed by [1]. This method proved to be a valid alternative to the current navigation method based on Artificial Potential Fields, and, if the code is optimized properly, it could even be proposed as an alternative to traditional planning based navigation methods like the $A*$ algorithm and the *Probabilistic Road Maps* algorithm (PRM).

Finally, to facilitate the design process of these proposals, a set of functions, classes and scripts were created. This group of tools (now called the *Swarm Robotics Toolbox*) provide the user with the ability to visualize tests, save figures, generate videos, perform statistical analyses, among others. These tools are intended for future educational use, so each element in the toolbox is properly documented, with a more detailed description of its operation and options present in the repository where all the code of this project is contained.

CAPÍTULO 1

Introducción

El algoritmo de Particle Swarm Optimization (PSO) consiste de un algoritmo de optimización estocástico, nacido a partir de la modificación de un algoritmo de simulación de parvadas. Cuando sus creadores [4] tomaron dicho algoritmo y le retiraron las restricciones de proximidad de las "aves", se percataron que las entidades resultantes se comportaban como un optimizador.

Luego de la publicación de esta investigación, una gran cantidad de académicos notaron el potencial del algoritmo. Este es el caso del mega proyecto *Robotat* de la Universidad del Valle de Guatemala, donde el algoritmo PSO se propuso como la base para el sistema de navegación de un conjunto de robots diferenciales. Específicamente, se consiguió implementar una modificación del algoritmo que no solo permitía respetar las limitaciones físicas de los robots [1], sino que también era capaz de esquivar obstáculos [2]. No obstante, los resultados obtenidos eran altamente dependientes de múltiples hiper parámetros que fueron elegidos por [1] luego de realizar diferentes pruebas.

¿Existirá una forma de automatizar el proceso de selección de estos parámetros? ¿Existen alternativas al sistema de navegación propuesto? El presente trabajo de investigación se enfoca en responder ambas preguntas, explorando diferentes alternativas que hacen uso de inteligencia computacional. Específicamente, se propone utilizar redes neuronales recurrentes para la selección de parámetros (aprendizaje profundo), y la utilización de los principios de programación dinámica (aprendizaje reforzado) para proponer un sistema de navegación alternativo al utilizado actualmente por los robots (basado en el uso de campos potenciales artificiales [2]).

Para auxiliar en la realización de pruebas también se presenta un conjunto de herramientas auxiliares programadas en Matlab (ahora denominadas *Swarm Robotics Toolbox*), que permiten visualizar y agilizar el proceso de realización de pruebas, toma de datos y generación de estadísticas; todo desde un mismo script.

CAPÍTULO 2

Antecedentes

El departamento de Ingeniería Electrónica, Mecatrónica y Biomédica de la Universidad del Valle de Guatemala inició su introducción en el mundo de la inteligencia de enjambre con la fase 1 del "Megaproyecto Robotat". En este, diversos estudiantes se enfocaron en el diseño de todo el equipo que sería utilizado en años posteriores: Desde el diseño mecánico y electrónico de los "Bitbots"[1], hasta la construcción física de la mesa donde se colocarían los mismos [6, pág. 19].

Aunque este proyecto finalizó con gran parte de su estructura finalizada, muchos aspectos aún requerían de más trabajo. Debido a esto, en 2019 se comenzaron a refinar múltiples aspectos del "Robotat", como el protocolo de comunicación empleado por los "Bitbots" [6] y el algoritmo de visión computacional que se encargaría de detectarlos sobre la mesa en la que se desplazarían [7]. Otra área de gran enfoque dentro de todo este proceso, consistió del algoritmo encargado de controlar el comportamiento de enjambre de los robots. En esta área se desarrollaron tres tesis distintas.

2.1. Formaciones en Sistemas de Robots Multi-agente

La primera de las mismas, desarrollada por Andrea Peña [8], se enfocó en la utilización de teoría de grafos y control moderno para la creación y modificación de formaciones en conjuntos de múltiples agentes capaces de evadir obstáculos. El algoritmo resultante fue implementado tanto en Matlab como Webots, y aunque exitoso, los actuadores de los diferentes robots diferenciales tendían a emplear una gran cantidad de esfuerzo para alcanzar las posiciones requeridas. Por otro lado, el algoritmo de evasión de obstáculos implementado fue altamente exitoso y marcó un precedente para futuras investigaciones.

[1] Versiones más económicas del robot empleado con propósitos educativos: E-Puck [5]

2.2. Implementación de PSO con Robots Diferenciales Reales

En la segunda tesis, desarrollada por Aldo Aguilar [9], se tomó como base la versión estándar del algoritmo de *Particle Swarm Optimization* (PSO) y se procedió a modificarla para que fuera capaz de ser implementada en robots diferenciales reales. El problema principal con acoplar directamente el movimiento de las partículas del PSO con la locomoción de los robots es que el movimiento de las partículas es sumamente irregular, por lo que puede causar la saturación de los actuadores de los robots.

Para combatir este problema se propuso una modificación: Cada uno de los robots no seguirían el movimiento exacto de una partícula del PSO, sino que cada uno tomaría la posición de la misma como una *sugerencia* de hacia donde desplazarse. Esta sugerencia luego sería alimentada a un controlador de seguimiento punto a punto que se encargaría de calcular la velocidad angular de las dos ruedas del robot diferencial. Se experimentó con ocho diferentes metodologías de control para el movimiento de los robots de forma acorde a sus especificaciones y capacidades.

La efectividad de cada método de control se cuantificó haciendo una interpolación de las trayectorias seguidas por el robot y luego calculando *la energía de flexión* de las mismas. Mientras más grande fuera la energía de la trayectoria, mayor sería el esfuerzo realizado por el robot en términos de sus velocidades, por lo que se consideraban como más efectivos aquellos métodos que contaran con los valores de energía más bajos. Aplicando este criterio a las diferentes pruebas realizadas en un entorno de simulación, se llegó a determinar que los dos mejores controladores para los robots consistían de los controladores LQR y LQI.

2.3. PSO y Artificial Potential Fields

El movimiento de las partículas en la versión *bidimensional* del algoritmo de PSO proviene del movimiento de las mismas sobre una superficie tridimensional denominada "función de costo". El objetivo de las partículas, es encontrar el mínimo global de esta función o las coordenadas (X,Y) correspondientes a la altura más baja de la superficie [10].

En la tercera tesis, desarrollada por Juan Cahueque [2], se explora la idea de diseñar y utilizar funciones de costo "personalizadas" como herramientas de navegación. Llamadas *Artificial Potential Fields* (APF), estas funciones están diseñadas para atraer a las partículas del algoritmo PSO hacia un punto específico del plano mientras esquivan cualquier obstáculo presente en el camino. Para esto, se coloca un valle de gran profundidad en el punto objetivo y colinas de gran magnitud donde existen obstáculos. El resultado: Las partículas tienden a esquivar las grandes alturas, favoreciendo el movimiento coordinado hacia los valles o la meta.

En total se modelaron tres entornos, cada uno con una meta y múltiples obstáculos intermedios de diferentes dimensiones. A estos escenarios se les denominó caso A, B y C. La navegación alrededor de estos entornos se simuló tanto en Matlab como en Webots. Para las simulaciones en Webots, se emplearon versiones modificadas de los controladores propuestos por Aldo Aguilar [9], a manera de hacerlos compatibles con un entorno en el que se presentan obstáculos. Al finalizar se llegó a concluir que el controlador PID con filtros *hard-stops* era

el más útil al momento de evadir obstáculos pequeños y dispersos en el escenario, mientras que el controlador LQR presentaba mayor versatilidad al momento de esquivar obstáculos de mayor tamaño.

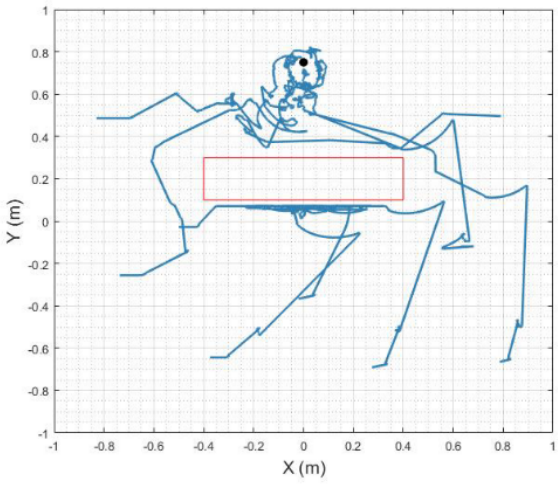

Figura 1: Trayectorias seguidas por E-Pucks en caso A alrededor del obstáculo colocado [2]

2.4. Aprendizaje Reforzado Profundo y Robótica

La robótica está íntimamente relacionada con la teoría de control. Debido a esto, resulta lógico que el aprendizaje reforzado (un área cuyas ideas base están fundamentadas sobre la teoría de control) actualmente se presente como una alternativa válida a algunas técnicas de control. No obstante, uno de los problemas más grandes de emplear aprendizaje reforzado en su forma tradicional es que este requiere de una alta cantidad de conocimiento a priori sobre la situación a enfrentar.

Dos elementos tienden a generar problemas: La definición del espacio de estados y la obtención de la dinámica del sistema[2]. Para problemas con un alto número de estados o cuya dinámica es sumamente compleja, puede llegar a ser casi imposible derivar modelos que describan precisamente a ambos elementos. Para combatir este problema, se pueden utilizar redes neuronales de diferentes tipos como herramientas de modelado no lineal. Al acoplamiento de estas dos destrezas (aprendizaje profundo y aprendizaje reforzado) se le llama aprendizaje reforzado profundo o *deep reinforcement learning*.

En robótica, y más específicamente robótica móvil, existen múltiples estudios que hacen uso de esta nueva destreza, todos empleando un enfoque diferente para su modelado. En algunos casos, como el de [11], se tiene conocimiento previo sobre el ambiente a recorrer (en la forma de una imagen) por lo que se puede emplear una red neuronal convolucional (comúnmente empleada para extraer información sobre una imagen), para extraer el estado actual del sistema y así elegir las velocidades a utilizar por el robot diferencial que navegará

[2]En aprendizaje reforzado, la dinámica de un sistema incluye los mismos elementos que en el caso del control, la única diferencia, es que los términos tienden a renombrarse. La salida de la planta es ahora la acción tomada por el agente, la retroalimentación es ahora una señal de recompensa, etc. [3]

dicho ambiente.

En otros casos, como el de [12], ya se cuenta con información previa sobre la pose, velocidad y distancia del robot hacia los diferentes obstáculos del ambiente, por lo que se puede utilizar toda esta información para definir el estado actual del sistema. Entonces, en este caso, las redes neuronales se utilizan para derivar las ecuaciones necesarias para ejecutar el problema de aprendizaje reforzado. Se emplean dos redes: Una red convolucional capaz de modelar la función de valor de acción óptima ($Q^*(s, a)$) y otra red neuronal convencional que permite modelar la función de valor de estado ($V^*(s)$). Entrenando a dichas redes utilizando otros algoritmos de esquivado de obstáculos como ejemplo, es posible entrenar a un robot diferencial capaz de esquivar obstáculos en situaciones nunca antes experimentadas, únicamente basándose en su actual experiencia.

Todas estas ideas pueden llegar a ser extendidas a sistemas multi-agente, donde las funciones de valor del MDP (*markov decision process*) pueden ser construidas en base a las observaciones conjuntas de múltiples robots explorando de manera simultánea el ambiente [13]. Esto permite generar estrategias de control descentralizadas en grandes espacios de estados, de manera más rápida y efectiva.

CAPÍTULO 3

Justificación

Uno de los algoritmos más populares dentro del área de inteligencia de enjambre, es el algoritmo de *Particle Swarm Optimization* (PSO). Este algoritmo consiste de un método de optimización que hace uso partículas con posiciones y velocidades para la exploración de una "función de costo".

Dado que este algoritmo fue originalmente propuesto en 1995, actualmente existe una gran cantidad de modificaciones y variaciones del mismo. Debido a su eficiencia computacional, no se tiende a variar en gran medida su estructura, colocando mayor énfasis en la modificación de los parámetros asociados al mismo. Según la aplicación, se pueden llegar a favorecer diferentes aspectos como la rapidez de convergencia, exploración de la superficie de costo o la precisión de las partículas en términos de su capacidad para encontrar el mínimo global de la función de costo.

No obstante, en todos estos casos algo permanece constante: no existe un conjunto de parámetros universales que permitan que el algoritmo se ajuste de manera flexible a cualquier aplicación. Existen variaciones "dinámicas" que modifican el valor de los parámetros conforme este se ejecuta, pero su efecto es limitado, comúnmente afectando únicamente propiedades como la exploración y convergencia.

Debido a esto, en este proyecto se desea diseñar un sistema basado en redes neuronales recurrentes que lleve al desarrollo de un selector de parámetros dinámico e inteligente, capaz de ajustar los mismos de forma automática. Esto aportará hacia el actual método de navegación propuesto por [1] y [2], el cual hace uso del algoritmo PSO para generar las trayectorias seguidas por un conjunto de robots diferenciales a través de un ambiente con obstáculos.

Además, también se explora una potencial alternativa para resolver dicho problema de navegación, la cual hace uso de programación dinámica (propio de aprendizaje reforzado) para generar trayectorias desde un punto de inicio arbitrario hasta una meta dispuesta por el usuario. La idea de esta propuesta, consiste de no solo aportar una nueva herramienta de navegación a la iniciativa del "Mega-proyecto Robotat", sino también marcar un precedente

para el futuro uso de aprendizaje reforzado en esta área, por parte de futuros estudiantes de la Universidad del Valle de Guatemala.

Para visualizar, ejecutar y facilitar la implementación de todos los elementos previamente descritos, se construyó un conjunto de funciones, clases y scripts dentro de Matlab (ahora nombradas conjuntamente como la *Swarm Robotics Toolbox*). Cada parte de este conjunto de herramientas está debidamente documentado y cuenta con suficiente flexibilidad como para utilizarse para probar una variedad de metodologías de navegación para robots diferenciales en futuras investigaciones.

CAPÍTULO 4

Objetivos

4.1. Objetivo General

Optimizar la selección de parámetros en algoritmos de inteligencia de enjambre mediante el uso de Reinforcement Learning y Deep Learning.

4.2. Objetivos Específicos

- Combinar los trabajos sobre Artificial Potential Fields (APF) y Particle Swarm Optimization (PSO) desarrollados en fases previas del proyecto Robotat.
- Construir una colección de datos de entrenamiento y validación para usar en métodos de Reinforcement y Deep Learning, a partir de múltiples corridas de algoritmos de robótica de enjambre.
- Desarrollar e implementar un algoritmo que determine automáticamente los mejores parámetros para los algoritmos de robótica de enjambre.

CAPÍTULO 5

Alcance

En esta investigación se buscaba tomar los avances de los dos antecesores directos de este proyecto ([1], [2]) y combinarlos para generar un método de navegación para robots diferenciales basado en el *Particle Swarm Optimization Algorithm* o PSO. En su forma canónica o estándar, este algoritmo depende de diferentes hiper-parámetros que deben ser elegidos por el usuario. Según el valor de los mismos, se pueden alterar propiedades como la dispersión, precisión y velocidad de convergencia de las partículas.

Para auxiliar en el proceso de selección de estos parámetros, en este trabajo se implementó la estrategia de selección de parámetros automatizada nombrada *Deep PSO Tuner*. Esta permite la selección dinámica y automática de los hiper-parámetros del PSO, utilizando como guía el comportamiento de simulaciones previas del algoritmo. En total se experimentó con tres tipos de red neuronal recurrente: LSTM, BiLSTM y GRU, entrenando a cada arquitectura con un total de 7,700 simulaciones previas del algoritmo PSO. Luego de ajustar los hiper parámetros propios de las redes (número de neuronas, capas, *batch size*, *learning rate*, entre otros), se determinó que en la mayor parte de los casos, la red BiLSTM y LSTM proporcionaban una mejora sustancial en la velocidad de convergencia y susceptibilidad a mínimos locales de los algoritmo PSO estándar.

Además de esto, también se presenta una alternativa al método de navegación basado en el PSO, que emplea las ideas del ejemplo clásico de aprendizaje reforzado y programación dinámica *Gridworld* para generar trayectorias a través de un entorno con obstáculos. Ambas propuestas, fueron construidas haciendo uso del *Swarm Robotics Toolbox*, un conjunto de herramientas que permite la simulación de una variedad de elementos: Desde las partículas propias del algoritmo PSO, hasta robots diferenciales, incluyendo métodos de seguimiento de trayectorias y una variedad de funcionalidades adicionales.

Todas estas ideas y propuestas se presentan como una potencial solución al problema de navegación de robots diferenciales a través de un entorno conocido. Por lo tanto, para trabajos posteriores, se podrían tomar diferentes enfoques. Desde el lado de aprendizaje profundo, se podría dar seguimiento al *PSO Tuner*, generando un *dataset* de mayor tamaño, alterando la estructura de inputs y outputs, modificando los hiper parámetros de la red,

diseñando nuevas y mejores métricas para cuantificar el estado actual de las partículas del *swarm*, entre otros. Para el lado de aprendizaje reforzado, se podrían continuar explorando nuevas alternativas para el control de los robots diferenciales, colocando particular énfasis en el área de aprendizaje reforzado, la cual parece haber generado ya una gran cantidad de resultados positivos en el área.

Finalmente, se podría realizar todo lo anterior, tomando la infraestructura del *Swarm Robotics Toolbox* como base y expandiendo sus capacidades para incluir funcionalidades adicionales como: Localización y mapeo, acoplamiento de sensores a cada robot (*lidars* y ultrasónicos), un sistema de colisiones mucho más robusto, exportación de los mapas diseñados en Matlab directamente hacia Webots, etc.

CAPÍTULO 6

Marco teórico

6.1. Particle Swarm Optimization (PSO)

6.1.1. Orígenes e Implementación Original

El algorítmo de *Particle Swarm Optimization* (PSO) consiste de un método de optimización estocástica[1], basado en la emulación de los comportamientos de animales que se movilizan en conjunto. Sus creadores [4] tomaron el algoritmo de simulación de parvada de aves de [14] y experimentaron con el mismo. Luego de múltiples pruebas, se percataron que el algoritmo presentaba las cualidades de un método de optimización. En base a esto, modificaron las reglas del algoritmo de parvada y propusieron la primera iteración del PSO en 1995.

El algoritmo original propone la creación de un conjunto de "m" partículas, cada una con una posición y velocidad correspondientes. Estas partículas se desplazan sobre la superficie de una función objetivo cuyos parámetros (variables independientes) son las "n" coordenadas de cada partícula. A dicha función objetivo se le denomina "función de costo" y al escalar que genera como resultado se le denomina "costo". El objetivo de las partículas es encontrar un conjunto de coordenadas que generen el valor de costo más pequeño posible dentro de una región dada. Para esto, las partículas se ubican en posiciones iniciales aleatorias y proceden a calcular el valor de costo correspondiente a su posición actual.

Si el costo actual es inferior al de su posición previa, se dice que la partícula ha encontrado un nuevo *personal best* ($\vec{p}(t)$). Este proceso se repite para cada partícula en el enjambre, por lo que al finalizar cada iteración del algoritmo se contará con "m" estimaciones de $\vec{p}(t)$. El valor mínimo de todas estas estimaciones se le conoce como mínimo global. Si el mínimo global actual es inferior al de la iteración previa, se dice que se ha encontrado un nuevo *global best* ($\vec{g}(t)$).

[1] Estocástico: Cuyo funcionamiento depende de factores tanto predecibles dado el estado previo del sistema, así como en factores aleatorios

Para la actualización de su posición y velocidad actual, las partículas utilizan el siguiente conjunto de ecuaciones:

$$\begin{aligned}
\vec{V}(t+1) = \quad & \vec{V}(t) & \\
& +C_1(\vec{p_{pos}}(t) - \vec{x}(t)) & \textit{Componente Cognitivo} \\
& +C_2(\vec{g_{pos}}(t) - \vec{x}(t)) & \textit{Componente Social} \\
\vec{X}(t+1) = \quad & \vec{X}(t) + \vec{V}(t+1) &
\end{aligned}$$

Debido a que el *personal best* proviene de la memoria individual de cada partícula sobre su mejor posición hasta el momento, a la sección de la ecuación de la velocidad que utiliza $\vec{p_{pos}}(t)$ se le denomina el "componente cognitivo". Por otro lado, debido a que el *global best* proviene de la memoria colectiva sobre la mejor posición alcanzada hasta el momento, a la sección de la ecuación de la velocidad que utiliza $\vec{g_{pos}}(t)$ se le denomina "componente social".

6.1.2. Mejoras Posteriores

El algoritmo PSO propuesto por [15], presentaba un comportamiento oscilatorio o divergente en ciertas situaciones, por lo que en 2002, [16] se dio a la tarea de diseñar múltiples métodos para restringir y asegurar la convergencia del algoritmo. Uno de los más utilizados hasta la actualidad consiste de una modificación a la regla de actualización de la velocidad conocida como "modelo tipo 1"":

$$\begin{aligned}
\vec{V}(t+1) = \quad & \omega\vec{V}(t) & \textit{Término Inercial} \\
& +R_1C_1(\vec{p_{pos}}(t) - \vec{x}(t)) & \textit{Componente Cognitivo} \\
& +R_2C_2(\vec{g_{pos}}(t) - \vec{x}(t)) & \textit{Componente Social}
\end{aligned}$$

En este nuevo conjunto de ecuaciones, las variables de restricción agregadas (C_1, C_2 y ω) están dadas por las siguientes expresiones:

$$\begin{aligned}
\omega &= \chi & \chi &= \frac{2\kappa}{|2 - \phi - \sqrt{\phi^2 - 4\phi}|} \\
C_1 &= \chi\phi_1 & \phi &= \phi_1 + \phi_2 \\
C_2 &= \chi\phi_2 &&
\end{aligned}$$

Como se puede observar, bajo estas modificaciones, la velocidad es ahora dependiente de tres variables nuevas: ϕ_1, ϕ_2 y κ. Los autores de la modificación sugieren que $\phi_1 = \phi_2 = 2.05$ y $\kappa = 1$, aunque como regla general, para asegurar la convergencia del algoritmo se debe cumplir con que $\kappa > (1 + \phi - 2\sqrt{\phi})|C_2|$.

6.2. *Deep Learning*

En la actualidad, los términos *machine learning* y *deep learning* se han convertido en sinónimos de inteligencia artificial, pero en muchas ocasiones se desconoce la diferencia entre

ambos. De acuerdo con [17], el aprendizaje automático o *machine learning* es un sistema que produce reglas a partir de datos y respuestas, en lugar de producir respuestas a partir de reglas y datos, como es el caso de la programación tradicional.

Más formalmente, *machine learning* se puede definir como la búsqueda de representaciones útiles de un conjunto de datos de entrada dentro de un espacio de posibilidades, utilizando una señal de retroalimentación (datos de entrenamiento) como guía. El *deep learning* entonces, consiste de una sub-área del aprendizaje automático donde se obtienen nuevamente representaciones útiles de datos, pero colocando particular énfasis en el aprendizaje por medio de "capas" apiladas de representaciones o modelos cada vez más complejos [17].

Los modelos utilizados para crear estas capas apiladas se les denomina redes neuronales y similar al cerebro humano, la unidad fundamental de una red es la neurona. Si el número de capas y neuronas del modelo es muy numeroso (ejemplos modernos comunes utilizan cientos de capas sucesivas para sus modelos) el modelo puede ser considerado parte del aprendizaje profundo. De lo contrario, el modelo se clasifica como un "perceptrón multicapa" propio del área de *shallow learning*.

En el contexto de aprendizaje profundo, una neurona consiste de una regresión lineal modificada que toma los outputs de todas las neuronas de la capa previa ($\boldsymbol{x}$), los multiplica por una matriz de pesos (W^T) y luego les suma un vector de constantes denominados *biases* (B). Para acotar la salida de esta regresión lineal (z), dicha salida se introduce en una "función de activación" (σ) que limita el rango de la salida.

$$\boldsymbol{Z} = \boldsymbol{W}^T\boldsymbol{x} + \boldsymbol{B}$$
$$\boldsymbol{A} = \sigma(\boldsymbol{Z})$$

Las reglas que rigen a una neurona, como es posible observar, son sumamente simples. La complejidad de un modelo de aprendizaje profundo, proviene de colocar múltiples neuronas en cada capa, y múltiples capas de las mismas dentro de la red.

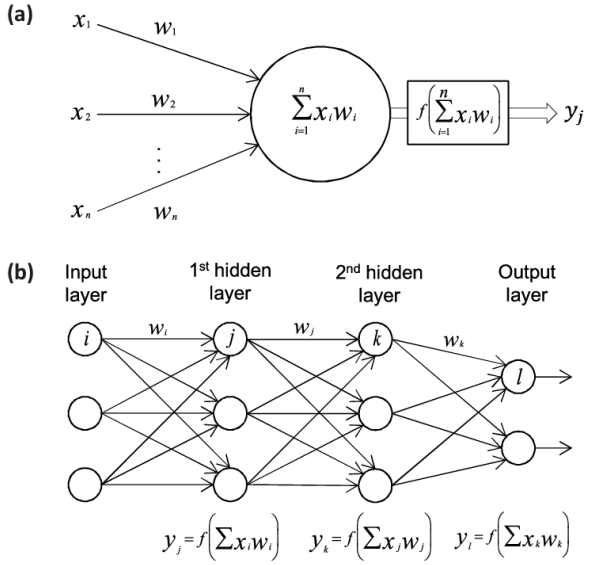

Figura 2: (a) Estructura de neurona. (b) Ejemplo de una red neuronal [18]

Las neuronas de la red "aprenden" alimentando una serie de datos en las neuronas de

entrada o la *input layer* y propagando los datos a través de toda la red hasta finalmente obtener una salida "y". Ahí, la salida es introducida en una función de costo, en conjunto con las salidas deseadas dados los datos de entrada, produciendo un escalar que indica el error inherente a la salida o "costo". El objetivo es minimizar el valor del costo, por lo que este se utiliza como guía para propagar cambios a los vectores $\boldsymbol{W}$ y $\boldsymbol{B}$ de las neuronas individuales. Se continúa este proceso de forma iterativa hasta encontrar el conjunto de vectores $\boldsymbol{W}$ y $\boldsymbol{B}$ que producen el costo más pequeño [17].

6.2.1. Redes Neuronales Recurrentes

Las redes neuronales consistentes de múltiples neuronas y capas interconectadas son altamente poderosas para estimar datos de carácter estático. Para elementos variantes en el tiempo o que traen consigo una estructura secuencial, las redes estándar tienden a generar resultados pobres. Esto se debe a dos aspectos. En primer lugar, las muestras de entrenamiento que forman parte de datos secuenciales tienden a contar con *features* de longitud variable. Por ejemplo, para una red encargada de traducir texto de un lenguaje a otro, una oración en español puede llegar a contener más o menos palabras que su correspondiente traducción en inglés. En segundo lugar, una red tradicional es incapaz de generar relaciones temporales entre diferentes *time-steps* del *dataset* [19].

Una red neuronal recurrente, da solución a ambos problemas. La razón para esto, es que en cada iteración del entrenamiento, la red neuronal calcula su salida como la suma ponderada de su entrada actual y un conjunto de parámetros calculados durante la iteración previa. Esto le brinda la capacidad de utilizar la dimensión temporal para realizar sus estimaciones. Por lo tanto, una red neuronal recurrente de 100 "celdas", no consiste de una capa de 100 neuronas, sino de una única neurona que procesa de manera recurrente los datos de 100 iteraciones consecutivas. A pesar de esto, en diagramas, esta única neurona tiende a "desplegarse" con fines informativos.

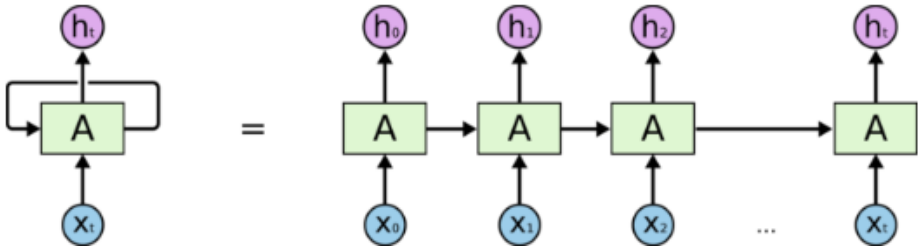

Figura 3: Representación desplegada de una red neuronal recurrente [20]

Matemáticamente, se puede establecer que una red neuronal recurrente calcula el estimado de su salida ($\hat{y}^{<t>}$) en un *time-step* t haciendo uso de su entrada actual ($x^{<t>}$) y una constante $a^{<t-1>}$ que representa la información proveniente de *time steps* anteriores.

$$\begin{aligned} a^{<t>} &= g\left(W_{aa}a^{<t-1>} + W_{ax}x^{<t>} + b_a\right) \\ \hat{y}^{<t>} &= g\left(W_{ya}a^{<t>} + b_y\right) \end{aligned} \tag{1}$$

En este conjunto de ecuaciones, la función g representa una función de activación. Comúnmente para el cálculo de $a^{<t>}$ se emplea la función $ReLU$ o tangente hiperbólica, mien-

tras que para $\hat{y}^{<t>}$ se puede variar la función según los requerimientos de la tarea. Para una tarea de clasificación, por ejemplo, se puede emplear una función sigmoide que limite los valores de salida (probabilidades) entre 0 y 1 [19]. Esta notación es comúnmente simplificada para obviar el uso de múltiples constantes W en la ecuación para $a^{<t>}$

$$\begin{aligned} a^{<t>} &= g\left(W_a\left[a^{<t-1>}, x^{<t>}\right] + b_a\right) \\ \hat{y}^{<t>} &= g\left(W_{ya}a^{<t>} + b_y\right) \end{aligned} \tag{2}$$

En este caso, las constantes W_{aa} y W_{ax} son concatenadas horizontalmente, mientras que la sección $\left[a^{<t-1>}, x^{<t>}\right]$ de la ecuación 2, representa una concatenación vertical de los vectores $a^{<t-1>}$ y $x^{<t>}$ (x se coloca abajo de a).

6.2.2. Tipos de Capa en Redes Neuronales Recurrentes

Las ecuaciones previamente especificadas definen el comportamiento de una capa estándar recurrente. Esta es útil para ciertos problemas, pero para problemas de mayor complejidad existen 2 alternativas de capa muy comunes en el área de redes neuronales recurrentes: La *gated recurrent unit* (GRU) y la *long short term memory* (LSTM).

***Gated Recurrent Unit* (GRU)**

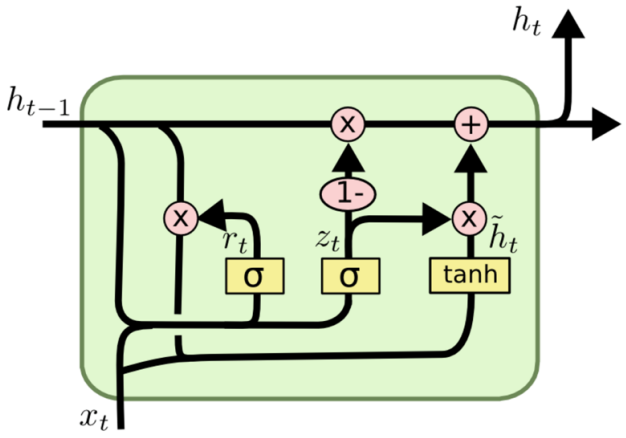

Figura 4: Estructura interna de una neurona GRU [21]

Simplificación de la neurona LSTM. Debido a su estructura y complejidad reducida, estas comúnmente tienden a utilizarse en mayores números para capturar dependencias entre muestras temporales (*time steps*) que se encuentran muy lejanas entre si temporalmente. Las ecuaciones que rigen dicha neurona, son una versión modificada de las ecuaciones 1 y 2.

$$
\begin{aligned}
z_t &= \sigma\left(W_z \cdot [h_{t-1}, x_t] + b_z\right) \\
r_t &= \sigma\left(W_r \cdot [h_{t-1}, x_t] + b_r\right) \\
\tilde{h}_t &= \tanh\left(W_h \cdot [r_t \odot h_{t-1}, x_t] + b_h\right) \\
h_t &= (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t
\end{aligned}
\tag{3}
$$

Donde h_t consiste del valor de memoria interna de la neurona, $\tilde{h}_t$ consiste del estimado para el siguiente valor de la memoria interna, z_t representa la *gate* de actualización de la memoria (la probabilidad de sustituir el valor previo de memoria por su estimado) y r_t representa la *gate* de relevancia (que tan relevante es el *time step* pasado para el cálculo del estimado actual de la memoria interna).

Cabe mencionar que en estas ecuaciones se utiliza la función de activación "sigmoide" ya que esta función causa que los valores de salida sean, o muy cercanos a 0 o muy cercanos a 1. De aquí la razón que se le llame a los términos que emplean estas funciones *gates*, ya que consisten de virtualmente una compuerta lógica binaria que tiende a dejar pasar toda la información (cuando su valor es 1) o simplemente impide el paso de la información (cuando su valor es 0) [19].

Long Short Term Memory **(LSTM)**

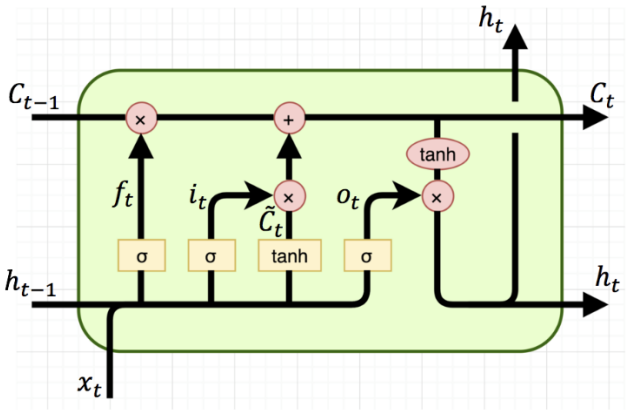

Figura 5: Estructura interna de una neurona LSTM [21]

Versión generalizada de la neurona GRU. Esta neurona presenta una mayor capacidad de predicción debido a un simple cambio: Contrario a la neurona GRU, que controla la actualización de la memoria interna a través de una única *gate* (la *gate* de actualización z_t), una neurona LSTM cuenta con una *gate* para cada término en la ecuación de actualización del valor interno de la memoria C_t (f_t y i_t). Esto permite que, para el nuevo valor de la memoria interna, no se tenga que decidir entre mantener su valor y actualizarlo (como en una neurona GRU). En una neurona LSTM, el nuevo valor de la memoria interna consiste de una combinación de su valor predicho y su valor previo [19]. Las ecuaciones que rigen el comportamiento de esta neurona son las siguientes:

$$
\begin{aligned}
f_t &= \sigma\left(W_f \cdot [h_{t-1}, x_t] + b_f\right) \\
i_t &= \sigma\left(W_i \cdot [h_{t-1}, x_t] + b_i\right) \\
o_t &= \sigma\left(W_o \cdot [h_{t-1}, x_t] + b_o\right) \\
\tilde{C}_t &= \tanh\left(W_c \cdot [h_{t-1}, x_t] + b_c\right) \\
C_t &= f_t \odot C_{t-1} + i_t \odot \tilde{C}_t \\
h_t &= o_t \odot \tanh\left(C_t\right)
\end{aligned}
\tag{4}
$$

Donde C_t consiste del valor de memoria interna de la neurona, $\tilde{C}_t$ consiste del estimado para el siguiente valor de la memoria interna, f_t representa la *gate* de olvidado (que cantidad de la información previa debe mantener en la nueva estimación), i_t representa la *gate* de actualización y o_t consiste de la *gate* de salida (la influencia del estimado actual de la memoria interna, sobre el estimado de salida) [19].

6.2.3. Redes Neuronales Recurrentes Bidireccionales

Uno de los problemas de las redes neuronales recurrentes previamente observadas, es que estas son capaces generar relaciones entre el valores presentes y pasados, pero no entre valores presentes y futuros. Para solucionar este problema, se realiza una modificación a la red recurrente tradicional en la que esta no solo transmite información de iteraciones pasadas al cálculo actual, sino que también transmite información de valores futuros al presente.

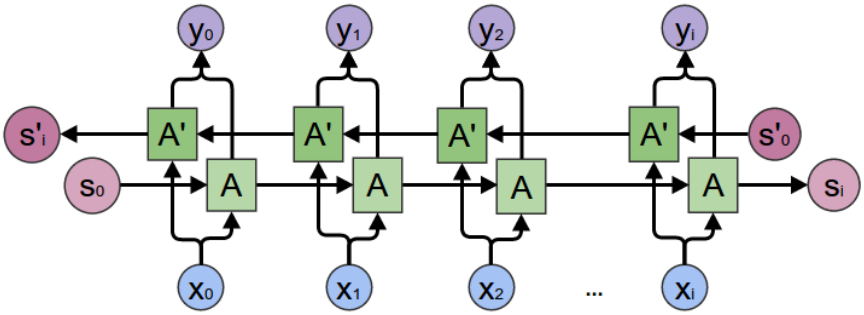

Figura 6: Representación desplegada de una red neuronal recurrente bidireccional [22]

Esta modificación presenta únicamente alteraciones pequeñas a la estructura interna de una neurona recurrente, por lo que, en teoría, se pueden continuar utilizando las mismas neuronas previamente descritas, pero ahora en en su versión bidireccional. De aquí que existan variaciones como la BiLSTM, las cuales permiten generar relaciones temporales tanto hacia el pasado como hacia el futuro.

6.3. *Reinforcement Learning*

Al igual que el aprendizaje profundo, el aprendizaje reforzado o *reinforcement learning* también consiste de una sub-rama del aprendizaje automático, ya que este tipo de aprendizaje es capaz de generar representaciones útiles de datos. La diferencia con el aprendizaje

profundo radica en la forma en la que genera los modelos para estas representaciones. En el caso del aprendizaje reforzado, al sistema no se le dice como operar, sino que este debe descubrir cuales son las acciones que producen la mejor recompensa probando las diferentes opciones disponibles [3].

6.3.1. Multi-armed Bandits

K-Armed Bandit

Contamos con una *bandit machine* (máquinas de *slots* encontradas en un casino) con k brazos. Cada vez que se hala uno de los brazos, se obtiene una recompensa. El objetivo, es maximizar la recompensa obtenida luego de cierta cantidad de juegos o rondas, por ejemplo, luego de 1000 juegos. La recompensa entregada por cada brazo sigue una distribución probabilística distinta para la entrega de su recompensa. A este problema se le denomina el *K-armed bandit problem*

Figura 7: Representación gráfica de un *multi-armed bandit*. Cada palanca observada retorna una recompensa según una distribución probabilística diferente.

Cada una de las k acciones a tomar tienen una recompensa promedio dada por la acción tomada. A esta cantidad se le llamará el "valor" de la acción y está dada por la siguiente expresión

$$\begin{aligned} q_*(a) &\doteq \mathbb{E}\left[R_t \mid A_t = a\right] \quad \forall a \in \{1, \ldots, k\} \\ &= \sum_r p(r \mid a) r \end{aligned} \tag{5}$$

Donde:

$q_*(a)$: Valor de la acción
a : Acción arbitraria
A_t : Acción en el tiempo t
R_t : Recompensa en tiempo t

En otras palabras, el valor se puede re-expresar como la suma de todas las posibles recompensas según su probabilidad de ver dicha recomensa[2]. Si se supiera el valor de cada

[2]Si se trata un caso continuo, la suma se puede sustituir por una integral.

acción, el problema sería trivial porque siempre tomaríamos la acción a con el mayor valor. Comúnmente no conocemos los valores de las acciones con total seguridad, pero podemos estimarlos. A este estimado le llamamos: $Q_t(a)$.

Idealmente el valor de $Q_t(a)$ debería de ser lo más cercano posible a $q_*(a)$. Durante cada *time step* (t), existe al menos una acción cuyo valor es el más alto. Estas acciones se denominan *greedy actions*. Si se toman estas acciones, se dice que se está explotando el conocimiento actual del valor de las acciones. Si se toman las acciones restantes, se dice que se está explorando, porque esto ayuda a mejorar el estimado de las *non-greedy actions*. En explotación, podríamos decir que se toman las acciones inmediatas con el mejor valor, mientras que en la exploración se descubren nuevas opciones que, a largo plazo, crean un mejor panorama de todas las opciones disponibles.

No es posible explorar y explotar de manera simultánea tomando una única acción. Debido a esto, comúnmente se habla de un "conflicto" entre exploración y explotación. Existen métodos complejos para balancear ambas etapas, pero están basadas en formulaciones extensas y conocimiento a priori sobre el problema.

Métodos de Valor-Acción (Método de Promedio de Muestreo)

A los métodos para estimar el valor de una acción $(Q_t(a))$ y luego utilizar este estimado para decidir entre un grupo de acciones se les denomina *métodos de valor-acción*. El valor real de una acción consiste del promedio de la recompensa obtenida cuando esa acción es seleccionada. Una forma de estimar esto es obtener el promedio de las recompensas obtenidas hasta el *time step* $t-1$.

$$Q_t(a) \doteq \frac{\text{suma de recompensas cuando } a \text{ es tomada previo a } t}{\text{veces que se ha tomado } a \text{ previo a } t} \tag{6}$$

Si el denominador es 0 (nunca se ha tomado la acción), entonces asignamos un valor por defecto a $Q_t(a)$, como 0. A medida que el denominador tiende a infinito (se ha tomado muchas veces una acción) $Q_t(a)$ converge a $q_*(a)$. A este método para estimar valores se le denomina promedio muestral o *sample average*, ya que toma el promedio de un conjunto de muestras de las recompensas disponibles.

Métodos para Elegir Acciones

Greedy Action Selection: Se selecciona la acción con el valor estimado más alto (Con el $Q_t(a)$ más alto).

$$A_t \doteq \arg\max_a Q_t(a) \tag{7}$$

Epsilon Greedy: Alternativa simple para actuar avariciosamente buena parte del tiempo, pero cada cierto tiempo (con probabilidad ϵ) se selecciona aleatoriamente una acción de todas las disponibles, independientemente del valor de sus estimados. La ventaja de este método es que a medida que el número de *time steps* incrementa, cada acción será muestreada un número infinito de veces, asegurando que $Q_t(a)$ converja a $q_*(a)$

$$A_t \leftarrow \begin{cases} \text{argmax}_a\, Q_t(a) & \text{con probabilidad } 1-\epsilon \\ a \sim \text{ Uniforme } (\{a_1 \dots a_k\}) & \text{con probabilidad } \epsilon \end{cases} \tag{8}$$

El tipo de método a utilizar cambia según la aplicación. Para recompensas que tengan una alta varianza, por ejemplo, el método ϵ *greedy* obtendrá mejores resultados que la *greedy action selection*. Si la varianza es 0, sería ineficiente implementar un ϵ *greedy*, ya que tomando acciones avariciosas se estimaría el valor de la decisión en el primer intento. A pesar de esto, encontrar un escenario sin varianza es muy extraño. Muy comúnmente, la recompensa obtenida por tomar una acción cambia según el tiempo. Esto causa que el valor real de una acción cambie de manera constante, por lo que es necesario un método ϵ *greedy* que permita explorar las opciones disponibles cada cierto tiempo.

Estimación Incremental

Sabemos que podemos estimar el valor de una acción utilizando el promedio de las recompensas obtenidas. ¿Cómo implementamos esto con memoria y un tiempo de computación por *time step* constantes? Iniciamos analizando el estimado para el valor de la acción n (Q_n). R_i consiste de la recompensa obtenida al seleccionar la recompensa la "i-ésima" vez. Es claro que para la acción n tomaremos en cuenta todas las acciones previas, que en total serían $n-1$ acciones. Por lo tanto, el estimado será igual a:

$$Q_n \doteq \frac{R_1 + R_2 + \cdots + R_{n-1}}{n-1} \tag{9}$$

Para obtener esta implementación, podríamos guardar el valor de todas las recompensas obtenidas y luego estimar el valor. No obstante, si se hace esto, los requerimientos computacionales crecerían con el tiempo[3]. Para solucionar esto, se tomó la fórmula para estimar el valor de la acción y se manipuló para que la actualización de la estimación requiera del menor tiempo de computación posible.

$$\begin{aligned} Q_{n+1} =& \frac{1}{n}\sum_{i=1}^{n} R_i \\ =& \frac{1}{n}\left(R_n + \sum_{i=1}^{n-1} R_i\right) \\ =& \frac{1}{n}\left(R_n + (n-1)\frac{1}{n-1}\sum_{i=1}^{n-1} R_i\right) \\ =& \frac{1}{n}\left(R_n + (n-1)Q_n\right) \\ =& \frac{1}{n}\left(R_n + nQ_n - Q_n\right) \\ =& Q_n + \frac{1}{n}\left[R_n - Q_n\right] \end{aligned} \tag{10}$$

[3]En caso de implementarse en Matlab, las dimensiones del array de recompensas crecería en cada iteración, por lo que el compilador retornaría la advertencia "*you should pre-allocate for speed*"

Donde:

Q_n : Estimado previo

n : Número de estimado

R_n : Recompensa recibida previamente

Otra forma de colocar esta última fórmula escrita es la siguiente:

$$\text{Nuevo Estimado} \leftarrow \text{Viejo Estimado} + \text{Step Size}(\underbrace{\text{Objetivo} - \text{Viejo Estimado}}_{\text{Medida de Error}}) \tag{11}$$

Usando esta regla de actualización incremental y una selección de acción ϵ *greedy* se puede escribir un algoritmo *bandit* completo de la siguiente forma.

Algorithm 1: Algoritmo de Bandit Simple

Inicializar: para $a = 1$ hasta k

$Q(a) \leftarrow 0$

$N(a) \leftarrow 0$

Loop:

$$A \leftarrow \begin{cases} \arg\max_a Q(a) & \text{con probabilidad } 1-\varepsilon \quad \text{(breaking ties randomly)} \\ \text{Una acción aleatoria} & \text{con probabilidad } \varepsilon \end{cases}$$

$R \leftarrow \text{bandit}(A)$

$N(A) \leftarrow N(A) + 1$

$Q(A) \leftarrow Q(A) + \frac{1}{N(A)}[R - Q(A)]$

Nota: La función *bandit* consiste de una función que toma una acción y retorna una recompensa según lo dicten las reglas del sistema[4].

Tracking en un Problema No Estacionario

El planteamiento previo es útil para problemas estacionarios, donde la probabilidad de recompensa no cambia con el tiempo. No obstante, gran parte de los problemas encontrados en el mundo real consisten de problemas no estacionarios. En estos casos, tiene más sentido ponerle mayor prioridad a recompensas recientes, que a recompensas recibidas en el pasado lejano. Una forma popular de conseguir esto, es utilizar un *time-step* constante .

$$Q_{n+1} \doteq Q_n + \alpha\,[R_n - Q_n] \tag{12}$$

Al realizar esta modificación la actualización del estimado actual se puede definir como el promedio ponderado de las recompensas actuales y el estimado inicial Q_1.

[4]Para este algoritmo existe un ejemplo programado en Matlab dentro de la carpeta de *Reinforcement Learning Coursera - Ejercicios*. El archivo en cuestión es: *Capitulo2_ TenArmTestbed.mlx*.

$$
\begin{aligned}
Q_{n+1} =& Q_n + \alpha \left[R_n - Q_n \right] \\
=& \alpha R_n + (1 - \alpha) \, Q_n \\
=& \alpha R_n + (1 - \alpha) \left[\alpha R_{n-1} + (1 - \alpha) Q_{n-1} \right] \\
=& \alpha R_n + (1 - \alpha) \alpha R_{n-1} + (1 - \alpha)^2 Q_{n-1} \\
=& \alpha R_n + (1 - \alpha) \alpha R_{n-1} + (1 - \alpha)^2 \alpha R_{n-2} + \\
& \cdots + (1 - \alpha)^{n-1} \alpha R_1 + (1 - \alpha)^n Q_1 \\
& = (1 - \alpha)^n Q_1 + \sum_{i=1}^{n} \alpha (1 - \alpha)^{n-i} R_)
\end{aligned}
\tag{13}
$$

Como se puede observar, el peso $\alpha(1-\alpha)^(n-i)$ asignado a la recompensa R_i depende de hace cuantas recompensas atrás $(n-i)$ esta recompensa fue experimentada. $(1-\alpha) < 1$, por lo que el peso asignado a cada recompensa disminuye de forma exponencial a manera que incrementa el número de recompensas recibidas (a medida que el exponente de $(1-\alpha)$ aumenta). Esto es comúnmente llamado: *Exponential recency-weighted average* o promedio exponencial ponderado por su carácter reciente.

En algunas situaciones es conveniente variar el *step-size* α a lo largo de la ejecución del programa. A este α variante se le denomina α_n. En el caso del método incremental, por ejemplo, $\alpha_n = 1/n$. Algo interesante es que la convergencia de Q no está asegurada para toda α_n. Para asegurar la convergencia (con probabilidad 1 o total seguridad) se debe cumplir que

$\sum_{n=1}^{\infty} \alpha_n(a) = \infty$	Requerida para garantizar que los pasos son lo suficientemente grandes para eventualmente sobrepasar cualquier transiente inicial
$\sum_{n=1}^{\infty} \alpha_n^2(a) < \infty$	Garantiza que los pasos o steps se tornan lo suficientemente pequeños como para asegurar convergencia.

El $\alpha_n = 1/n$ del método incremental cumple con estas condiciones, pero el α_n con parámetro constante, no. Esto implica que los estimados este último método nunca convergen completamente ya que cambian según las recientemente obtenidas recompensas. Esto no es siempre malo, ya que como ya fue mencionado previamente, este tipo de adaptación dinámica es necesaria para poder adaptarse a ambientes no estacionarios. Existen funciones α_n que cumplen con las condiciones, pero muy raras veces se utilizan fuera de entornos académicos.

Valores Iniciales Optimistas

Como podemos observar, todos los métodos previamente explicados dependen de la estimación inicial del valor de las acciones Q_1.

$$
\begin{aligned}
Q_{n+1} =& Q_n + \alpha\left[R_n - Q_n\right] \\
=& \alpha R_n + (1-\alpha)Q_n \\
=& \alpha R_n + (1-\alpha)\left[\alpha R_{n-1} + (1-\alpha)Q_{n-1}\right] \\
=& \alpha R_n + (1-\alpha)\alpha R_{n-1} + (1-\alpha)^2 Q_{n-1} \\
=& \alpha R_n + (1-\alpha)\alpha R_{n-1} + (1-\alpha)^2 \alpha R_{n-2} + \\
& \cdots + (1-\alpha)^{n-1}\alpha R_1 + (1-\alpha)^n Q_1 \\
=& (1-\alpha)^n Q_1 + \sum_{i=1}^{n} \alpha(1-\alpha)^{n-i} R_i
\end{aligned}
$$

$$
\begin{aligned}
Q_{n+1} &= \frac{1}{n}\sum_{i=1}^{n} R_i \\
&= \frac{1}{n}\left(R_n + \sum_{i=1}^{n-1} R_i\right) \\
&= \frac{1}{n}\left(R_n + (n-1)\frac{1}{n-1}\sum_{i=1}^{n-1} R_i\right) \\
&= \frac{1}{n}\left(R_n + (n-1)Q_n\right) \\
&= \frac{1}{n}\left(R_n + nQ_n - Q_n\right) \\
&= Q_n - \frac{1}{n}\left[R_n - Q_n\right]
\end{aligned}
$$

Para $Q_{n+1} = Q_2$ se emplea el estimado previo $Q_n = Q_1$

En estadística, se puede decir que estas fórmulas tienen un sesgo dado por sus valores iniciales. Para el método de *sample-average* (con un *step size* que decrece), el efecto del sesgo eventualmente desaparece, pero para $\alpha_n = \alpha$ (constante) el sesgo no desaparece. Esto no es un problema, de hecho puede llegar a ser utilizado para informar al agente sobre el tipo de recompensas que puede esperar. El único problema, es que este estimado inicial consiste de un nuevo parámetro a elegir.

Por ejemplo, si le colocamos un valor alto para la recompensa inicial (mucho más grande que lo que eventualmente va a conseguir), los métodos de acción-valor (estimar el valor de una acción y luego elegir una acción en base a esto) se ven motivados a explorar más. No importando cuales sean las acciones tomadas después, la recompensa siempre será más pequeña, entonces "decepcionado", el agente optará por cambiar de acción, probando todas las opciones disponibles varias veces antes de converger. El resultado es una mayor exploración

A este método que promueve la exploración se le denomina: Valor iniciales optimistas. Esta es una técnica muy útil para problemas estacionarios, pero para no estacionarios, se torna inútil ya que el estimado inicial únicamente aplica para la primera instancia del entorno. Cuando el entorno cambie (por su carácter dinámico), el valor inicial ya no aplica.

Unbiased Constant Step Size Trick

Previamente se explicó que el método de *sample-average* no es susceptible al valor inicial, pero no es muy útil en entornos no-estacionarios. Una forma de obtener "lo mejor de dos mundos" es utilizar un *step size* igual a

$$\beta_n \doteq \alpha/\bar{o}_n \tag{14}$$

Donde α es una constante y o_n es un valor acumulativo que inicia en 0 y se actualiza de la siguiente manera.

$$\bar{o}_n \doteq \bar{o}_{n-1} + \alpha\,(1 - \bar{o}_{n-1}) \tag{15}$$

Este tipo de *step-size* es inmune al efecto del sesgo inicial, además de causar que el estimado del valor de la acción se torne en un *recency weighted-average* como en el método *sample-average*.

Selección de Acción de Límite Superior

Recordar que en el método de ϵ-*greedy* existe la probabilidad de que el agente pruebe opciones "no avariciosas" (*non greedy*), no obstante, no existe preferencia sobre la selección realizada. Claramente sería mejor si se eligieran las acciones "no avariciosas" según su potencial de ser óptima o una buena opción. Para lograr esto podemos elegir una acción tomando en cuenta que tan cercano está un estimado Q_n a un valor máximo y cual es la incertidumbre del propio estimado

$$A_t \doteq \underset{a}{\text{argmax}} \left[\underbrace{Q_t(a)}_{\text{Explotación}} + c \underbrace{\sqrt{\frac{\ln t}{N_t(a)}}}_{\text{Exploración}} \right] \tag{16}$$

Donde:

t : Tiempo actual

$N_t(a)$: Número de veces que una acción a ha sido seleccionada antes del tiempo t[5].

$c > 0$: Parámetro que controla el grado de exploración.

A este tipo de selección de acción se le denomina *upper confidence bound* o UCB. La idea del UCB consiste en calcular los intervalos de incertidumbre de cada uno de los estimados. Estos intervalos nos dicen: "Calculo que el valor real de una acción está entre este límite inferior y este límite superior". Lo que hacemos es que actuamos de forma optimista y establecemos: "Si el valor puede estar entre estos intervalos y queremos la mayor recompensa posible entonces se tomará la acción con el límite superior más alto". Es optimista porque suponemos que en el mejor caso posible, el valor real de la acción se encontrará exactamente en el límite superior, maximizando la recompensa.

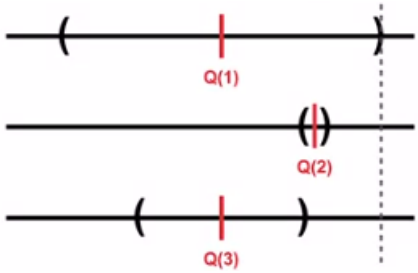

Figura 8: Tres estimados diferentes para el valor de una acción. La línea punteada representa el valor a tomar dado que se trata del límite superior de la incertidumbre

[5]Si $N_t(a) = 0$, entonces a es considerada como una acción "maximizadora".

La raíz de la expresión representa la incertidumbre o varianza del estimado del valor de la acción a. Al maximizar esta sección, entonces se obtiene una especie de "límite superior" para el posible valor real de la acción a. Algunos puntos importantes a mencionar sobre este método son los siguientes:

- Cada vez que se tome una acción a el denominador aumenta, por lo que la incertidumbre baja.
- Si se toma una acción distinta de a entonces t incrementa mientras $N_t(a)$ permanece igual, por lo que el numerador crece en conjunto con la incertidumbre.
- Se usa un logaritmo en el numerador para que el crecimiento del numerador se haga cada vez más pequeño. Debido a esto, todas las acciones se seleccionarán, pero aquellas con estimados bajos o que se han elegido muy seguido se elegirán con cada vez menos frecuencia.

Este método es útil, pero para entornos no-estacionarios y espacios de estados muy grandes, su uso se torna impráctico.

6.3.2. Procesos de Decisión de Markov (MDP's)

Interfaz Agente-Ambiente

En una tarea a solucionar al "aprendiz" y "tomador de decisiones" se le llama "agente". Todo con lo que interactúa el agente se le denomina "medio ambiente"[6]. Ambos elementos interactúan de forma continua

- **El agente** tomando decisiones y **el medio ambiente** respondiendo a las mismas presentando nuevas situaciones al agente.
- **El medio ambiente** crea recompensas (valores numéricos) que el agente busca maximizar.

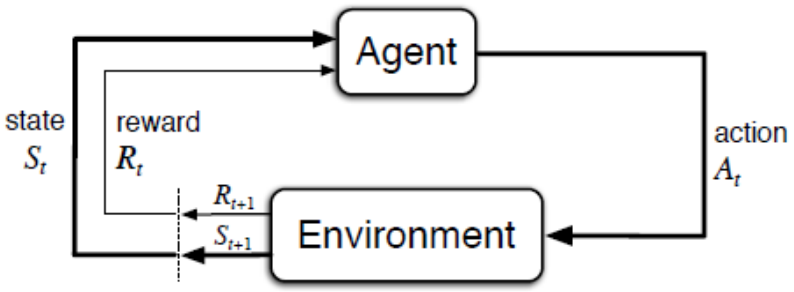

Figura 9: Interacción Agente-Ambiente en un proceso de decisión de Markov [3]

[6]Algunos de los términos de *Reinforcement Learning* tienen un análogo en teoría de control: Agente = Controlador. Ambiente = Planta. Acción = Señal de Control.

Esta interacción ocurre en pasos

1. El agente interactúa luego de intervalos de tiempo discretos (1, 2, 3, 4, 5, ...)
2. En cada *time step* el agente recibe información sobre el estado del ambiente: S_t
3. En base a este estado selecciona una acción: A_t
4. Un *time step* más tarde, el agente recibe una recompensa numérica $R_(t+1)$ como consecuencia de su acción.
5. Vuelve a encontrar un nuevo estado.

Si este proceso se definiera como una secuencia de señales esta consistiría de una secuencia similar a la siguiente: Estado, Acción, Recompensa, Estado, ...

$$S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, \ldots$$

En un *Markov Decision Process* (MDP) finito, los estados, acciones y recompensas tienen un número finito de elementos (los *arrays* que codifican estos tienen dimensiones predeterminadas). Además, las variables S_t y R_t tienen distribuciones probabilísticas bien definidas únicamente dependientes del estado o acción pasados. En otras palabras, la probabilidad de que aparezca un estado y recompensa específicos, únicamente depende del estado y acción previos.

Podemos decir que esta no es una restricción del MDP, pero del estado actual. El estado debe incluir información sobre todos los aspectos de la interacción "agente-ambiente" que pueden llegar a generar un cambio en el futuro. Si esto es cierto, se dice que el estado tiene la "propiedad de Márkov"

$$p\left(s', r \mid s, a\right) \doteq \Pr\left\{S_t = s', R_t = r \mid S_{t-1} = s, A_{t-1} = a\right\} \tag{17}$$

Con esta función p (dinámica de MDP) uno puede calcular otras cosas se deseen saber sobre el ambiente, por ejemplo:

- Probabilidades de transición entre estados

$$p\left(s' \mid s, a\right) \doteq \Pr\left\{S_t = s' \mid S_{t-1} = s, A_{t-1} = a\right\} = \sum_{r \in \mathcal{R}} p\left(s', r \mid s, a\right) \tag{18}$$

- La recompensa esperada para una pareja "estado-acción"

$$r(s, a) \doteq \mathbb{E}\left[R_t \mid S_{t-1} = s, A_{t-1} = a\right] = \sum_{r \in \mathcal{R}} r \sum_{s' \in \mathcal{S}} p\left(s', r \mid s, a\right) \tag{19}$$

- Las recompensas esperadas para el triplete "estado-acción-siguiente estado".

$$r\left(s,a,s'\right) \doteq \mathbb{E}\left[R_t \mid S_{t-1}=s, A_{t-1}=a, S_t=s'\right] = \sum_{r\in\mathcal{R}} r\frac{p\left(s',r \mid s,a\right)}{p\left(s' \mid s,a\right)} \tag{20}$$

Comúnmente se utiliza la notación dependiente de 4 argumentos (la primera antes de estas tres), pero estas otras pueden ser útiles ocasionalmente. El *framework* de un MDP es muy flexible y aplicable a una variedad de problemas. Por ejemplo:

- Los *time steps* no necesariamente corresponden a intervalos uniformes de tiempo, pueden referirse a etapas de toma de decisiones y "actuación".
- Las acciones tomadas pueden de decisiones simples (voltajes a aplicar) o de decisiones complejas (ir o no al colegio). Pueden ser cualquier decisión que deseemos aprender a hacer.
- Los estados pueden estar definidos por medidas de bajo nivel (medidas de sensores) o medidas más abstractas y de alto nivel (descripciones simbólicas de objetos en un cuarto). Un estado puede consistir de cualquier conocimiento útil para tomar una decisión o tomar una acción.

La frontera entre agente y ambiente no es típicamente la misma que la frontera física de un robot o animal. En un robot por ejemplo, los motores, sensores y extremidades deberían considerarse parte del ambiente, en lugar de formar parte del agente como se esperaría. Las recompensas también son calculadas dentro de la frontera física de un robot o animal, pero no se consideran parte del agente como tal. Como regla general, cualquier cosa que no pueda ser arbitrariamente cambiada por el agente, se considera como exterior y por lo tanto, parte de su ambiente.

No todo con lo que el agente interactúa es desconocido. Generalmente el agente está consciente de la forma en la que las recompensas son obtenidas en base a sus acciones y estado actual. Siempre consideramos el cálculo de recompensa como algo externo al agente ya que define la tarea que debe resolver el agente y por lo tanto, está fuera de su control el poder cambiarla de forma arbitraria. Por ejemplo: Un agente sabe todo sobre su ambiente, pero aun así tiene problemas para resolver la tarea de *reinforcement learning* que se le da. Por ejemplo, alguien puede saber cómo funciona un cubo Rubik's, pero aun así le puede costar resolverlo.

La frontera "agente-ambiente" representa el límite del control absoluto del agente, no el límite de su conocimiento. En la práctica, esta frontera se establece una vez se ha elegido el juego de estados, acciones y recompensas que formarán parte del proceso de toma de decisiones que se desea resolver. El *MDP framework*, es una abstracción del problema de "aprendizaje orientado a objetivos basado en interacción". Este problema propone que elementos como sensores, memoria y aparatos de control, además del objetivo a cumplir se pueden reducir como 3 señales:

- Acciones: Decisiones del agente
- Estados: Fundamentos utilizados para tomar decisiones
- Recompensas: Señal que define el objetivo del agente

Metas y Recompensas

Hipótesis de recompensa: Todo aquello que podemos definir como objetivos o propósitos puede llegar a interpretarse como la maximización del valor esperado de la suma acumulativa de una señal escalar recibida denominada "recompensa".

Algunos ejemplos de recompensas son:

- Robot aprendiendo a caminar: Recompensa en cada *time step* proporcional al desplazamiento hacia adelante del robot.
- Robot aprendiendo a escapar de un laberinto: Recompensa de -1 por cada *time step* que esté dentro del laberinto para que aprenda a escapar rápido.
- Robot aprendiendo a navegar un espacio: Recompensa negativa cuando hay un choque.

La recompensa es una forma de comunicarle al agente qué es lo que se desea conseguir, no cómo se desea conseguirlo. Por ejemplo: Jugando ajedrez, no es recomendado que al agente se le recompense por comer piezas, sino solo por ganar el juego. Si se le dan recompensas por completar "sub-objetivos", puede que el agente aprenda a comer muchas piezas pero eventualmente pierda.

Retornos y Episodios

El objetivo de un agente es el siguiente

- Informalmente: Maximizar la recompensa acumulativa que este recibe a largo plazo.
- Formalmente: Si la secuencia de recompensas obtenida se define como $R_(t+1), R_(t+2), R_(t+3), \ldots$, deseamos maximizar el "retorno esperado" G_t o una función de la secuencia de recompensas.

En su forma más simple, el retorno es

$$G_t \doteq R_{t+1} + R_{t+2} + R_{t+3} + \cdots + R_T \tag{21}$$

Donde:
T : *Time step* final

Esta función es aleatoria, debido al comportamiento dinámico de la MDP. Debido a esto maximizamos el "retorno esperado". Esta forma de estructurar el retorno, es útil en problemas donde existe una noción natural de un *time step* final. En otras palabras, es útil para interacciones "agente-ambiente" que se pueden separar naturalmente en sub-secuencias o episodios. Por ejemplo, un episodio podría consistir de las partidas en un juego o intentos para recorrer un laberinto.

Cada episodio termina con un "estado terminal", seguido por un *reset* al estado inicial estándar o a una elección entre una distribución estándar de los diferentes estados iniciales. Un episodio iniciará de la misma manera, independientemente de la forma en la que terminó el último episodio. Entonces se puede considerar que todos los episodios terminan en el mismo "estado terminal", pero con diferentes recompensas en el camino. A tareas con este tipo de episodios se les denomina "tareas episódicas". En estas existen:

S : Estados no terminales
S^+ : Estados no terminales
T : Tiempo de finalización

Tareas que no pueden separarse en episodios definidos se denominan "tareas continuas". Para estas tareas es mucho más difícil definir una función de retorno ya que $T = \infty$ luego de un largo tiempo. Para estas tareas se utilizan "descuentos" porque de lo contrario las sumatorias no serían finitas. Un agente trata de seleccionar acciones a manera de maximizar la suma de las recompensas descontadas que recibe a lo largo del tiempo. En particular, elige A_t para maximizar el "retorno descontado"

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{22}$$

Donde:
$0 \leq \gamma \leq 1$: Tasa de descuento

Los valores de recompensa recientes tienen más peso que los futuros ya que los futuros están multiplicados por potencias cada vez más altas de gamma.

Tasa de descuento: Determina el valor presente de recompensas futuras

- Una recompensa recibida k *time steps* en el futuro valdrá $\gamma^{(k-1)}$ veces lo que valdría si se recibiera inmediatamente.
- Si $\gamma < 1$ la suma infinita del retorno descontado tiene un valor finito, mientras la secuencia de recompensas esté acotada.
- Si $\gamma = 0$ el agente solo se enfoca en maximizar las recompensas inmediatas. El agente se vuelve "miope".
- Cuando $\gamma \leftarrow 1$ el agente toma más en cuenta recompensas futuras.

Un retorno puede calcularse de manera iterativa de la siguiente manera:

$$\begin{aligned} G_t &\doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + \cdots \\ &= R_{t+1} + \gamma \left(R_{t+2} + \gamma R_{t+3} + \gamma^2 R_{t+4} + \cdots \right) \\ &= R_{t+1} + \gamma G_{t+1} \end{aligned} \tag{23}$$

A pesar que la sumatoria tiende al infinito, esta es finita si las recompensas son constantes y distintas de cero (siempre y cuando $\gamma < 1$). Si la recompensa es $+1$ el retorno sería:

$$G_t = \sum_{k=0}^{\infty} \gamma^k = \frac{1}{1-\gamma} \tag{24}$$

Notación Unificada para Tareas Episódicas y Continuas

Como pudimos ver hay dos tipos de tareas de aprendizaje reforzado: Episódicas y continuas. Cada una tiene una notación única, pero a veces se utilizan ambas en el mismo contexto. Para facilitar su manejo matemático se establece una notación que permita hablar de la misma manera para ambos casos. En lugar de considerar una tarea como una larga secuencia de *time steps*, ahora se considera como una serie de episodios, cada uno conformado por un número finito de *time steps*.

Debido a esto, ahora la notación para S_t, R_t, A_t pasa a ser: $S_(t,i)$, $R_(t,i)$, $A_(t,i)$.

Donde:
i : Número de episodio
t : Número de *time step* del episodio actual. Siempre inicia de 0 cada vez que se pasa a un nuevo episodio

A pesar de esta notación, la mayor parte del tiempo se habla de un único episodio en particular o de cosas que son aplicables a todos los episodios. Debido a esto, se tiende a obviar la i que hace referencia al número de episodio y se regresa a la notación previa. Ahora hay otro problema: El retorno G_t se define de manera diferente para tareas episódicas y continuas.

- En tareas episódicas, el retorno es una suma finita.
- En tareas continuas es una suma infinita.

Para unificar ambas expresiones, se considera que, cuando el agente llega al estado terminal, este entra a un "estado de absorción" que solo transiciona a sí mismo y genera recompensas de 0.

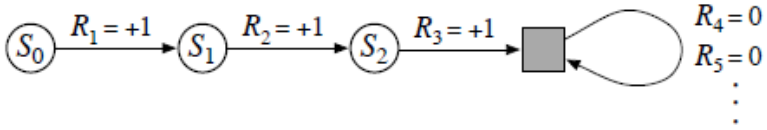

Figura 10: Sumar las recompensas de los primeros 3 estados es lo mismo que sumar la serie infinita, ya que luego se pasa al estado de absorción (cuadro gris).

Esto incluso puede aplicarse en conjunto con "descuentos". Entonces, obviando la i de número de episodio y tomando en cuenta la posibilidad que $\gamma = 1$ si la suma continúa definida, el retorno se define como:

$$G_t \doteq \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k \tag{25}$$

Donde:

t : *Time step* actual

k : Siguiente *time step*

Policies y Funciones de Valor

Casi todos los algoritmos en aprendizaje reforzado involucran la estimación de una "función de valor": Función que toma los estados (o las parejas estado-acción) y estima que tan bueno es para el agente estar en un estado específico según sus posibles estados futuros (o que tan bueno es realizar una acción específica en un estado dado).

La medida de que tan bueno viene de la cantidad de recompensas futuras que puede llegar a esperar el agente (o el retorno esperado G_t). Las recompensas dependen de dos factores: Las acciones que tomará el agente y el estado en el que se encuentra. La función que establece la forma de actuar del agente, se llama *policy*. En otras palabras, una *policy* consiste de la forma en la que un agente selecciona una acción. Existen dos tipos:

- Determinística: Mapea o asigna una acción a cada estado. El agente puede seleccionar la misma acción en dos estados diferentes. También se pueden obviar acciones completamente.
- Estocástica: Asigna probabilidades a cada una de las acciones disponibles en cada estado.

Una *policy* solo depende del estado presente. No puede depender de factores como "tiempo" u otros estados. *Policies* que dependen de otros factores se consideran "inválidas".

CAPÍTULO 7

Deep PSO Tuner

La selección de hiper parámetros en un algoritmo siempre tiende a consistir de un proceso largo y tedioso para el investigador, muchas veces requiriendo de más trabajo que la propia implementación del algoritmo como tal. Para evitar realizar esto de forma manual, muchos investigadores han empleado una variedad de metodologías. Para problemas de baja complejidad, un simple barrido de parámetros puede llegar a ser suficiente, pero para otros casos, una estrategia más "inteligente" es requerida.

Una de las opciones más comunes para problemas de mayor complejidad es la utilización de un optimizador. Cuando el algoritmo que se intenta ajustar por medio de un optimizador, consiste de un optimizador como tal, se dice que se está aplicando "meta-optimización". Dentro de esta área, es muy frecuente el uso de algoritmos genéticos, evolutivos y de enjambre. Uno de los más populares es el algoritmo de *Particle Swarm Optimization* (PSO), que ha sido ampliamente utilizado para la selección de hiper-parámetros en redes neuronales [23] (e incluso en otros algoritmos PSO [24]).

En la propuesta presentada a continuación, se da vuelta a esta idea: "¿Porqué no utilizar redes neuronales para seleccionar los parámetros del algoritmo PSO?". A continuación se presenta todo el proceso de diseño y validación que llevó a la construcción de esta idea.

7.1. Coeficientes de Restricción

De acuerdo con [25], la regla de actualización de posición y velocidad del algoritmo de PSO original tenía la siguiente forma

$$\begin{aligned} v(t+1) &= v(t) + \vec{U}\,(0,\phi_1) \otimes (\overrightarrow{p_{\text{local}}} - \vec{x}_i) + \vec{U}\,(0,\phi_2) \otimes (\overrightarrow{p_{\text{global}}} - \overrightarrow{x_i}) \\ x(t+1) &= x(t) + v(t+1) \end{aligned} \tag{26}$$

Donde:

$\vec{U}\,(0,\phi_1)$: Número uniformemente distribuido entre 0 y ϕ_1

$\otimes$: Element-wise product[1]

Con esta forma, esta versión del PSO presentaba ciertos problemas de inestabilidad, en particular porque el término de la velocidad ($v(t)$) tendía a crecer desmesuradamente. Para limitarlo, la primera solución propuesta por [4] fue truncar los posibles valores que podía llegar a tener la velocidad de una partícula a un valor entre ($-V_{max}$, V_{max}). Esto producía mejores resultados, pero la selección de V_{max} requería de una gran cantidad de pruebas para poder obtener un valor óptimo para la aplicación dada. Poco después [26] propusieron una modificación a la regla de actualización de la velocidad de las partículas: La adición de un coeficiente ω multiplicando a la velocidad actual.

$$v(t+1) = \omega v(t) + \vec{U}\,(0,\phi_1) \otimes (\overrightarrow{p_{\text{local}}} - \overrightarrow{x_i}) + \vec{U}\,(0,\phi_2) \otimes (\overrightarrow{p_{\text{global}}} - \overrightarrow{x_i}) \tag{27}$$

A este término se le denominó coeficiente de inercia, ya que si se hace un análogo entre la regla de actualización y un sistema de fuerzas, el coeficiente representa la aparente "fluidez" del medio en el que las partículas se mueven. La idea de esta modificación era iniciar el algoritmo con una fluidez alta ($\omega = 0.9$) para favorecer la exploración y reducir su valor hasta alcanzar una fluidez baja ($\omega = 0.4$) que favorezca la agrupación y convergencia. En la actualidad, existen múltiples métodos para seleccionar y actualizar ω [27].

Finalmente, el método más reciente para limitar la velocidad consiste de aquel propuesto por [16]. En este, se modeló al algoritmo como un sistema dinámico y se obtuvieron sus eigenvalores. Se pudo llegar a concluir que, mientras dichos eigenvalores cumplieran con ciertas relaciones, el sistema de partículas siempre convergería. Una de las relaciones más fáciles y computacionalmente eficientes de implementar consiste de la siguiente modificación al PSO:

$$\begin{aligned} v(t+1) &= \chi\left(v(t) + \vec{U}\,(0,\phi_1) \otimes (\overrightarrow{p_{\text{local}}} - \vec{x}_i) + \vec{U}\,(0,\phi_2) \otimes (\overrightarrow{p_{\text{global}}} - \vec{x}_i)\right) \\ \chi &= \frac{2\kappa}{\phi - 2 + \sqrt{\phi^2 - 4\phi}} \\ \phi &= \phi_1 + \phi_2 > 4 \end{aligned} \tag{28}$$

Al implementar esta constricción (Generalmente utilizando $\phi_1 = \phi_2 = 2,05$), se asegura la convergencia en el sistema, por lo que teóricamente ya no es necesario truncar la velocidad. Un aspecto importante a mencionar es que comúnmente se tiende a distribuir la constante χ en toda la expresión de la siguiente manera:

$$\begin{aligned} v(t+1) &= \chi v(t) + \chi\vec{U}\,(0,\phi_1) \otimes (\overrightarrow{p_{\text{local}}} - \overrightarrow{x_i}) + \chi\vec{U}\,(0,\phi_2) \otimes (\overrightarrow{p_{\text{global}}} - \overrightarrow{x_i}) \\ v(t+1) &= \chi v(t) + C_1 \otimes (\overrightarrow{p_{\text{local}}} - \overrightarrow{x_i}) + C_2 \otimes (\overrightarrow{p_{\text{global}}} - \overrightarrow{x_i})) \end{aligned} \tag{29}$$

[1] En Matlab esta operación se consigue anteponiendo un punto a la operación a realizar. Por ejemplo: ".*"

Aquí se puede llegar a ver más claramente como χ consiste del nuevo valor para el coeficiente de inercia, por lo que la adición de una constante ω sería redundante o incluso detrimental para el algoritmo (ya que introduce efectos imprevistos en la estabilidad del sistema). No obstante, al observar los resultados obtenidos a través del algoritmo modificado PSO (MPSO) de [1], se hizo evidente que la inclusión de una constante de inercia, en conjunto con los parámetros de constricción, puede llegar a consistir de una adición útil y válida.

Por lo tanto, previo a iniciar con el proceso de diseño, se decidió que el algoritmo PSO que se optimizaría, tendría la forma del MPSO propuesto por [1]. Esto implica que la ecuación para la actualización de la velocidad de las partículas PSO, tomará la siguiente forma:

$$\begin{aligned} v(t+1) &= \chi\left(\omega v(t) + \vec{U}\left(0,\phi_1\right) \otimes (\overrightarrow{p_{\text{local}}} - \vec{x}_i) + \vec{U}\left(0,\phi_2\right) \otimes (\overrightarrow{p_{\text{global}}} - \vec{x}_i)\right) \\ \chi &= \frac{2\kappa}{\phi - 2 + \sqrt{\phi^2 - 4\phi}} \\ \phi &= \phi_1 + \phi_2 \end{aligned} \tag{30}$$

La ventaja de esta ecuación, es que permite englobar tres métodos distintos de restricción en 1: Inercia, constricción y mixto. Si se desea utilizar el método de restricción por inercia, solo basta hacer que $\chi = \phi_1 = \phi_2 = 1$. Si se desea utilizar el método por constricción de [16] solo se hace que $\omega = 1$. Finalmente, si se desea utilizar el método mixto propio del MPSO de [1], se coloca $\phi_1 = 2$, $\phi_2 = 10$ y ω como una inercia de "exponente natural" (*Exponent1* en la *SR Toolbox*).

7.2. Pruebas Preliminares

Las redes neuronales tienden a presentarse como una herramienta mágica capaz de hacerlo todo. Dada una estructura de inputs y outputs adecuada, así como una arquitectura minuciosamente ensamblada, los resultados capaces de ser obtenidos con las mismas parecen ilimitados. Fue debido a esta ingenuidad e ilusión inicial, que las primeras pruebas realizadas en la construcción del *PSO Tuner* finalizaron en fracasos.

7.2.1. Primera Red: Time-stepper y Back Propagation

La primera prueba realizada estuvo inspirada por el capítulo 6.6 del libro "*Data-driven Science and Engineering: Machine Learning, Dynamical systems, and Control*" escrito por [28]. En este capítulo, nombrado "Neural Networks for Dynamical Systems", se expone la idea de un *time stepper*, o una red neuronal capaz de emular el comportamiento interno de la planta en un sistema dinámico. La idea es que el usuario tome muestras de las entradas y salidas del sistema, y en base a estas entrene a una *shallow neural network* para que intente replicar los patrones observados.

Como ejemplo, se propone modelar el conjunto de 3 ecuaciones diferenciales conocido como Lorenz 63.

$$\begin{aligned} \dot{x} &= \sigma(y - x) \\ \dot{y} &= x(\rho - z) - y \\ \dot{z} &= xy - \beta z \end{aligned} \tag{31}$$

Se definen las tres ecuaciones, se coloca el valor de las constantes β, ρ y σ, y luego se procede a utilizar un *solver* de ecuaciones diferenciales para simular el sistema. Se realiza un total de 100 corridas del *solver*, alterando la condición inicial utilizada en cada una. El resultado final son 2 matrices de input y output con 80000 filas (100 corridas de 800 time steps) y 3 columnas (una por cada variable: x, y y z).

Estas muestras son luego alimentadas a una *feed forward net* de 3 capas con 10 neuronas por capa. Dado que la matriz de output es igual a la matriz de input, pero con un desfase de 1 *time step*, la red tomará el estado del sistema en un *time step* t e intentará estimar su estado en $t + 1$. De aquí el nombre *time stepper* descrito previamente.

La red neuronal resultante, es luego comparada contra una resolución analítica de estas ecuaciones por medio de un solver. Generando la trayectoria del sistema dentro del espacio de estados, se hace evidente que la red es capaz de replicar la trayectoria perfectamente al inicio, pero comienza a perder precisión luego de cierto punto [28].

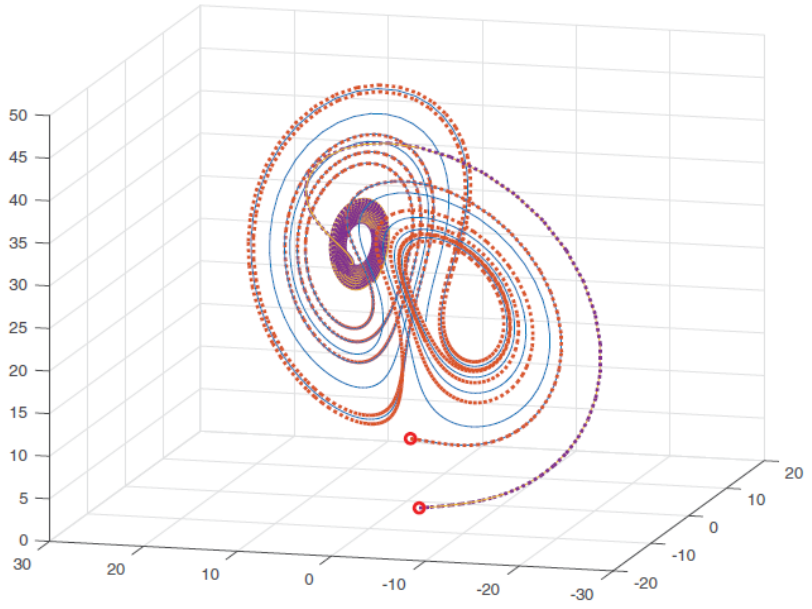

Figura 11: Comparación entre la evolución natural del sistema de Lorenz y la predicción de la red neuronal (Línea punteada) para dos condiciones iniciales aleatorias [28]

.

Los resultados parecían prometedores, por lo que se decidió aplicar las mismas ideas para el algoritmo PSO. La idea era ensamblar un *time stepper* para el algoritmo PSO, que fuera capaz de modelar de alguna manera su funcionamiento. Para esto, el estado del enjambre iba a consistir de una concatenación de todas las coordenadas X de las partículas, seguido de una concatenación de todas las coordenadas Y. Dado que se eligió un total de 1000 partículas a simular, la matriz resultante contaba con 2000 columnas y una fila por *time*

step del algoritmo simulado. El número de *time steps* del algoritmo se limitó a 1000 y se realizaron 100 corridas. Por lo tanto, las matrices de input y output consistían de matrices de 100000 x 2000.

Al intentar entrenar la misma arquitectura de red propuesta por [28] (3 capas y 10 neuronas por capa) utilizando estos datos, Matlab retornó un error casi inmediatamente, declarando que el sistema no contaba con la memoria suficiente como para calcular el jacobiano. El sistema requería de 52 gigas de RAM para realizar el cálculo, una cantidad exorbitante para una red tan pequeña.

Listing 7.1: Código de creación y entrenamiento red *time-stepper*

```
% 3 capas intermedias. 10 neuronas por capa.
net = feedforwardnet([10 10 10]);

% Funciones de activacion
net.layers{1}.transferFcn = 'logsig';
net.layers{2}.transferFcn = 'radbas';
net.layers{3}.transferFcn = 'purelin';

% Entrenamiento
net = train(net,input.',output.');
```

En una red neuronal, existen dos etapas distintivas de entrenamiento: *forward propagation* y *back propagation*. El paso de *forward propagation* implica el cálculo de las salidas de la red, en función de sus entradas, pesos y *biases*. Es un proceso relativamente simple que únicamente involucra sumas, multiplicaciones y la aplicación de funciones de activación. *Back propagation*, por otro lado, consiste de una operación mucho más compleja. En este paso, la red calcula el error de su salida al introducir la misma, en conjunto con el vector de salida esperado, en una función de costo. En función del costo generado, la red establece en que grado debe alterar cada uno de sus parámetros para generar un mejor estimado. Para realizar esto, emplea las derivadas parciales del costo. Calcular dichas derivadas de forma tradicional es computacionalmente ineficiente, por lo que una alternativa común es la utilización del jacobiano numérico [19].

De aquí proviene el error retornado por Matlab. El jacobiano numérico cuenta con un tamaño proporcional a los datos de entrada, por lo que para una matriz de 100000 x 2000, el tamaño del jacobiano se disparará rápidamente a dimensiones incapaces de ser manejadas por hardware tradicional. La forma de solucionar esto, consiste en dividir la totalidad de los datos en *batches* o pequeños sub-grupos de datos que pueden ser procesados de forma más sencilla. Desafortunadamente, la herramienta de entrenamiento de *shallow neural nets* proveída por Matlab, únicamente es capaz de alimentar los datos en su totalidad a la red.

7.2.2. Segunda Red: 2004 Entradas, 4 Salidas y Adam

Luego de investigar un poco, se hizo evidente, que la tarea en cuestión no iba a poder ser realizada empleando las herramientas de *shallow neural networks* de Matlab. Entonces se comenzaron a buscar alternativas. Entre estas, se encontró un ejemplo proveído por Matlab,

el cual enseñaba a estimar el número de contagiados de varicela empleando datos históricos de años previos [2].

En dicho ejemplo, se hacía uso de neuronas conocidas como LSTM's. De acuerdo a la documentación proporcionada por Matlab, estas eran capaces de tomar secuencias de datos y producir, luego de su entrenamiento, relaciones temporales entre los *time steps* de las secuencias de entrenamiento. Una vez se corroboraron los resultados obtenidos en el ejemplo, se decidió comenzar a adaptar el ejemplo para su uso en el *PSO Tuner*.

En el caso de redes neuronales profundas, Matlab toma cada columna como una muestra correspondiente a un *time step*, con cada fila consistiendo de una *feature*. Por lo tanto, se decidió construir un *feature vector* consistente de:

- 1000 filas de las coordenadas X de las 1000 partículas a simular.
- 1000 filas de las coordenadas Y de las 1000 partículas a simular.
- 2 filas para media de las coordenadas X y Y. Una para X y otra para Y.
- 2 filas para la desviación estándar de las coordenadas X y Y. Una para X y otra para Y.

El vector columna resultante contaba con 2004 *features*. Cabe mencionar que se decidió incluir la desviación estándar y la media de las partículas sobre cada eje, ya que se pensó que estos parámetros podían ser después manipulados por el usuario para intencionalmente generar diferentes patrones en el movimiento de las partículas. Por ejemplo, si la red recibía una señal "falsa" indicando que existe una mayor dispersión en el enjambre, se podía observar si la red generaba como consecuencia, un cambio acorde para poder generar dicha dispersión.

Para su salida, se decidió que la red generaría 4 estimados: ω, ϕ_1, ϕ_2 y el número de iteraciones para converger. Nuevamente, se decidió incluir esta última métrica adicional, ya que se consideró conveniente que la red fuera capaz de estimar el número de iteraciones que le tomaría converger al algoritmo PSO. Finalmente, para la arquitectura y opciones de entrenamiento se decidió emplear la configuración ya proporcionada por el ejemplo. La única opción que se alteró fue el tipo de optimizador utilizado. En lugar de utilizar el optimizador "adam" sugerido, se decidió emplear el optimizador de "descenso gradiente estocástico" o "SGDM".

	Opciones de Entrenamiento					
Arquitectura	Optimizador	Max Epochs	Gradient Threshold	Initial Learn Rate	Learn Rate Drop Period	Learn Rate Drop Factor
Inputs (2004) Neuronas LSTM (200) Ouputs (4) Capa de Regresión	SGDM	250	1	0.005	125	0.2

Cuadro 1: Arquitectura y opciones de entrenamiento para la primera red de prueba con LSTM's

[2]Este ejemplo puede ser encontrado como `Train_Shallow&DeepNN_DatosSecuenciales.mlx` de la carpeta de "Ejemplos y Scripts Auxiliares" del SR Toolbox.

Al igual que en la primera prueba, se inició el proceso de entrenamiento y, aunque Matlab no retornó un error, la gráfica de RMSE y *Loss* no presentó avance alguno. Luego de algunos segundos, Matlab dio fin al proceso de entrenamiento y retornó un modelo "entrenado". Al investigar un poco, se descubrió que el descenso de gradiente estocástico, es un método mayormente utilizado en redes neuronales profundas consistentes de simples perceptrones o neuronas clásicas (*fully connected layers* en Matlab). Para neuronas de mayor complejidad como una LSTM, es casi una necesidad el uso de un optimizador más robusto, entre los cuales se encuentra *Adam* [29].

Cambiando el tipo de optimizador de regreso a "Adam", el sistema entrenó exitosamente. A la red le tomó un aproximado de 10 minutos entrenar con las 7000 columnas de datos que le fueron proporcionadas. Al finalizar, se guardó el modelo generado y se acopló a una simulación de prueba de un algoritmo PSO en la cual la red tenía control sobre los parámetros ω, ϕ_1 y ϕ_2 de las partículas. Al iniciar la simulación, las partículas parecían dispararse repentinamente hacia las 4 esquinas de la mesa de trabajo. No importando la función de costo en la que se desplegaran, las partículas siempre presentaban el mismo comportamiento: Apenas se iniciaba el algoritmo, las partículas eran disparadas hacia las esquinas de la mesa de trabajo, incrementando drásticamente su dispersión.

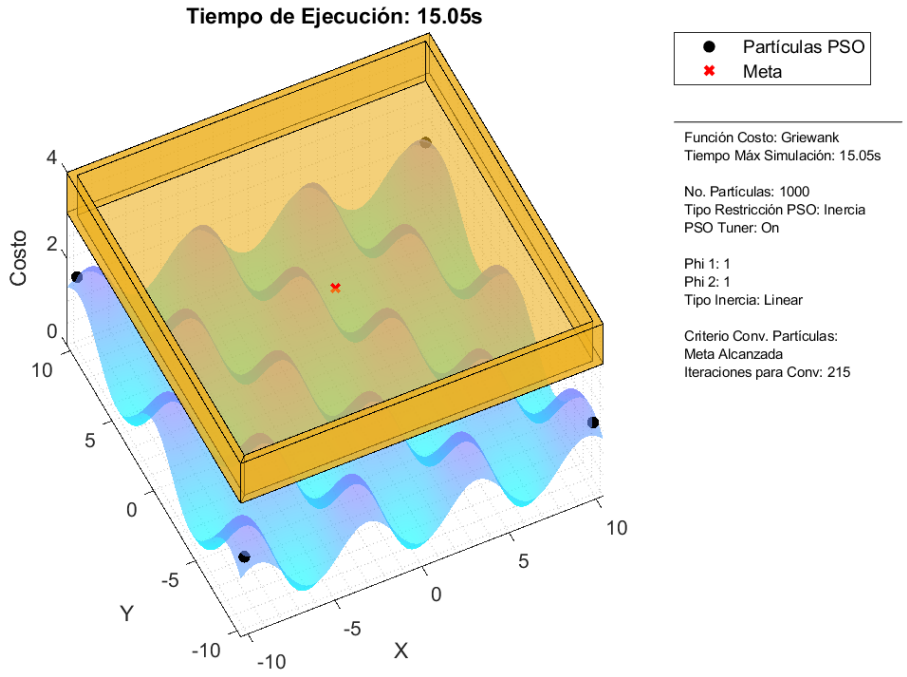

Figura 12: Partículas en las esquinas de la región de búsqueda luego de la ejecución de la primera prueba del PSO Tuner con neuronas LSTM.

.

Se intentó alterar la arquitectura utilizada para la red, sustituyendo las neuronas de tipo LSTM por neuronas tradicionales o *fully connected layers*. El resultado fue similar al anterior. Se continuaron alterando los diferentes hiper parámetros de las redes para observar si se generaban cambios significativos en el resultado, pero ninguna permutación de los parámetros parecía surtir efecto.

Nuevamente, luego de investigar, se descubrió que bases de datos que contienen una gran cantidad de ruido o aleatoriedad en sus entradas y salidas requieren de cierta preparación previa y filtrado para generar resultados efectivos [19]. En este caso, debido al carácter estocástico que caracteriza al movimiento de las partículas, lo más probable es que la red haya sido incapaz de derivar patrón alguno a partir del simple movimiento de cada uno de los integrantes del enjambre.

7.3. *Feature Vector* y Métricas de PSO

Luego de las dos pruebas fallidas previamente descritas, era claro que se requería de una re-estructuración del *feature vector* o vector de entrada a la red. La estructura previamente propuesta, consistente de todas las coordenadas, medias y desviaciones estándar, no solo tenía la desventaja de presentar demasiado ruido para poder generar conclusiones significativas sobre el estado de las partículas, sino que también carecía de escalabilidad.

Es evidente que, si la dimensionalidad del *feature vector* se hace dependiente del número de partículas simuladas, para cada cambio en el número de partículas se deberá entrenar un nuevo modelo que tome en cuenta las nuevas dimensiones del vector de entrada. Una alternativa para combatir este problema de escalabilidad es utilizar un método que "comprima" el número de *features* a un número menor. Para esto existen diferentes estrategias: *Principal component analysis*, *auto-encoders*, redes neuronales con una gradual reducción en el número de neuronas por capa, entre otros.

No obstante, esta reducción posterior resulta excesiva y puede llegar a incrementar innecesariamente el número de cálculos a realizar en cada iteración del algoritmo PSO. Entonces, se decidió optar por una nueva idea: Modelar el estado actual del enjambre a partir de diferentes métricas que cuantifiquen aspectos como su precisión, dispersión, rapidez de convergencia, etc. Esta idea no solo soluciona el problema de la escalabilidad y los datos ruidosos, sino que simultáneamente reduce significativamente la dimensionalidad del *feature vector*. A continuación se listan las diferentes métricas consideradas.

7.3.1. Diámetro y Radio de Enjambre

Específicamente, se decidió emplear 4 métricas: Cohesión, desviación estándar promedio, distancia normalizada a la meta y promedio de la distancia promedio entre partículas

CAPÍTULO 8

Swarm Robotics Toolbox: Herramienta de Simulación para Realización de Pruebas de Algoritmos de Enjambre en Matlab

Una de las primeras tareas que se realizó como parte de esta tesis, fue comprender y unir el contenido de las tesis de Aldo Nadalini y Juan Pablo Cahueque para su utilización conjunta. Conforme se comenzaron a agregar más y más funcionalidades a este script, se observó el potencial del mismo como un set de herramientas de simulación para diferentes aplicaciones de robótica de enjambre. De aquí nace la *Swarm Robotics Toolbox*[1]

El SR Toolbox consiste de un conjunto de funciones internas y externas que interactúan de manera conjunta a través de un script principal denominado `SR_Toolbox.mlx`. Todas la funciones que que componen el Toolbox están diseñadas para agilizar el proceso de debugging, realización de pruebas, validación de resultados, etc. Por lo mismo, este no cuenta con una interfaz o GUI asociada, ya que su inclusión solo alargaría el proceso de integración de nuevas características.

Casi todas las líneas de código en el script principal y funciones asociadas están comentadas, pero a continuación se presenta una explicación de alto nivel de todas las funcionalidades incluidas.

8.1. Livescripts

La extensión `.mlx` del script principal corresponde a un *livescript*. Si alguna vez se ha programado *python* a través de un *Jupyter Notebook*, el usuario estará familiarizado con la

[1]Inicialmente se le había nombrado *PSO Toolbox*, pero debido a que posteriormente se le agregaron funcionalidades que no hacían uso del algoritmo de PSO, esta fue renombrada para evitar confusiones sobre sus capacidades.

idea de un *livescript*. Un livescript permite el uso de código, imágenes, texto, ecuaciones, índices, secciones y otras características útiles dentro del mismo archivo.

A pesar de todo esto, una desventaja de los livescripts, es que estos iniciaron como una herramienta muy mal optimizada para Matlab, por lo que no se recomendaba su uso como un sustituto para un script tradicional. Estos pueden ser abiertos desde Matlab 2014a[2], pero mientras más antigua sea la versión, menor será el rendimiento observado en el script. En versiones más recientes (2020a), el rendimiento es casi idéntico al de un script tradicional.

La razón de emplear este formato para la SR Toolbox (En lugar de un archivo `.m` tradicional), es que los scripts de este tipo permiten realizar explicaciones mucho más claras sobre las ecuaciones, y planteamientos empleados en la Toolbox. La idea es tratar de contener la información dentro del mismo script, para así evitar tener que acudir a una fuente externa para comprender las formulaciones empleadas.

8.2. Matlab y Hardware

El SR Toolbox se probó en Matlab 2018b y 2020a. En ambas versiones funciona correctamente, pero como es de esperar, en la versión más reciente el rendimiento es mejor. El rendimiento (Específicamente la animación del movimiento de los robots) también es altamente dependiente del hardware empleado. La SR Toolbox se probó en una computadora con un CPU Intel i7-4790k de 4.4 GHz, 16 GBs de RAM DDR3 y una tarjeta gráfica NVIDIA GTX 780.

8.3. Setup Path y Limpieza de Workspace

Al apenas abrir el script, se encontrará con el índice y su primera sección: *Setup de Path*. Es necesario ejecutar esta sección para que el Toolbox se ubique en el directorio correcto (En caso el repositorio haya sido instalado en una nueva ruta, únicamente hace falta cambiar la ruta establecida por el script), incluya las carpetas de las diferentes funciones que utiliza y valide la calidad del archivo JSON que utiliza para sus features de *autocomplete*[3].

Luego se llega a la segunda sección: *Limpieza de Workspace*. Como lo menciona su nombre, esta sección se encarga de limpiar todas las variables del Workspace en caso existieran variables pre-existentes propias de otros scripts o de ejecuciones previas del Toolbox. Además de esto, también se limpian las variables persistentes empleadas dentro de diferentes funciones del Toolbox.

En Matlab, los valores de las variables dentro de una función desaparecen luego de que la misma finaliza su ejecución. Para poder mantener el valor de una variable entre diferentes

[2]Livescripts que incluyen elementos embebidos pueden ser abiertos a partir de Matlab 2016a. Si se abren con las versiones 2014 y 2015, Matlab ignorará todo excepto el código.

[3]Cuando el usuario utiliza una función en un livescript, Matlab le ofrece sugerencias sobre los diferentes parámetros que puede cambiar y las opciones disponibles. Estas sugerencias se pueden escribir manualmente para funciones realizadas por el usuario y es la razón de la inclusión de un archivo JSON. El nombre del archivo JSON no se puede cambiar.

llamadas a la función, se declara a la variable como `persistent`. La desventaja de declarar variables de este tipo, es que su valor se restablece hasta que el usuario reinicia Matlab. Para limpiar estas variables de forma programática, el usuario debe escribir `clear` seguido del nombre de la función que contiene variables persistentes.

8.4. Parámetros Generales

Si se continúa, se alcanza la sección: *Parámetros y Settings*. Esta sección permite controlar una gran variedad de elementos propios de la simulación. Desde parámetros dimensionales y visuales, hasta el generador de números aleatorios a emplear por el programa. A continuación se presenta una breve explicación de cada uno de los parámetros que pueden llegar a ser cambiados:

8.4.1. Método

- `Metodo`: Tipo de método que se simulará. Se incluye un *dropdown menu* que permite elegir una de las opciones disponibles. El usuario puede elegir tres tipos de método: Métodos dependientes de PSO, métodos basados en el seguimiento de una trayectoria y métodos mixtos (Mezcla PSO y seguimiento de trayectorias). En el caso de los métodos PSO, escribir en consola `help CostFunction`, para más información.

8.4.2. Dimensiones de Mesa de Trabajo

- `AnchoMesa`: Ancho de la mesa del Robotat. Unidades en metros.
- `AltoMesa`: Ancho de la mesa del Robotat. Unidades en metros.

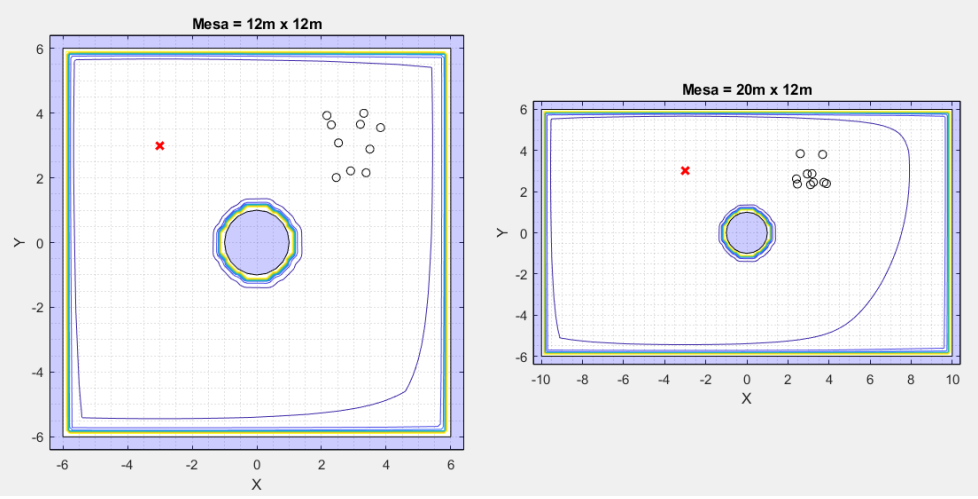

Figura 13: Efectos de alterar el ancho y alto de la mesa de trabajo.

- `Margen`: Ancho del margen uniforme que existirá alrededor de los bordes de la mesa de trabajo. Unidades en metros.

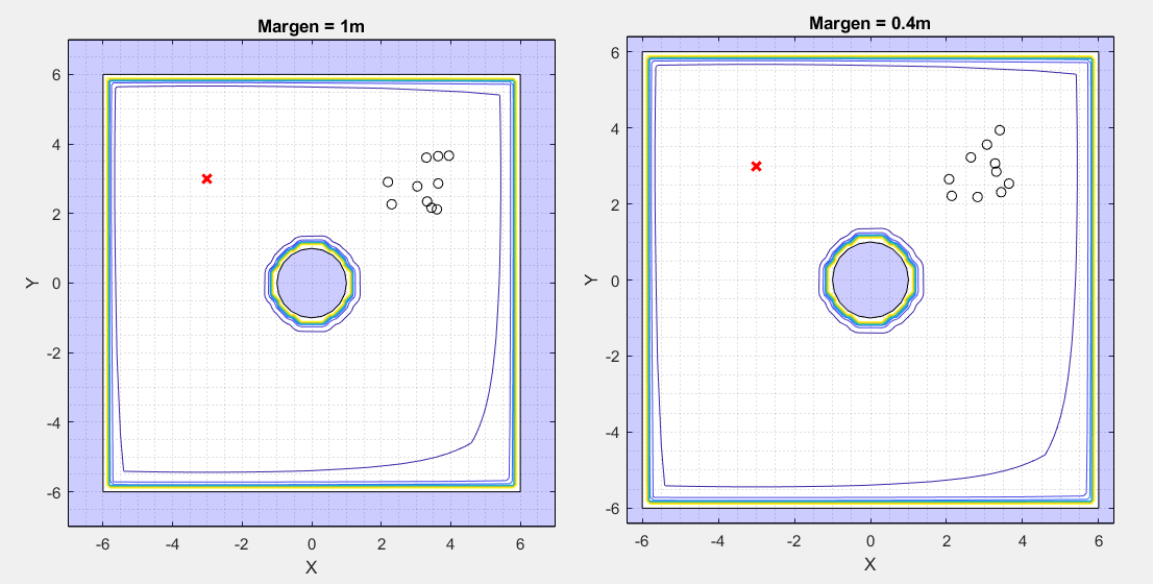

Figura 14: Efectos de alterar el tamaño del margen de la mesa de trabajo.

8.4.3. Settings de Simulación

- `EndTime`: Duración total de la simulación en segundos.
- `dt`: Delta de tiempo, tiempo de muestreo o cantidad de segundos que habrán pasado entre cada una de las iteraciones del *loop* principal del algoritmo.

8.4.4. Settings de Partículas PSO

- `NoParticulas`: Cantidad de partículas a utilizar dentro del algoritmo de PSO. En los métodos dependientes de PSO, el número de partículas tiende a sobre-escribir el número de E-Pucks o robots también.
- `PartPosDims`: El algoritmo de PSO consiste de un algoritmo de optimización por sobre todo. Debido a esto, el algoritmo es capaz de ser utilizado en problemas de casi cualquier dimensionalidad. Este parámetro permite cambiar el número de dimensiones que contiene el vector de posición de cada una de las partículas PSO.
- `CriterioPart`: Criterio de convergencia que utilizará el algoritmo PSO para establecer que debe finalizar. Para más información escribir `help getCriteriosConvergencia`.

8.4.5. Settings de Seguimiento de Trayectorias

- `TrayectoriaCiclica`: En métodos de seguimiento de trayectorias, el robot está activamente siguiendo un conjunto de puntos en orden secuencial. Si se establece que se

desea una trayectoria cíclica, cuando el robot alcance el último punto de su trayectoria, este tomará como siguiente punto a seguir el primer punto en la trayectoria. Si la trayectoria no es cíclica, la trayectoria no cambia al llegar al último punto.

8.4.6. Settings de E-Pucks

- `NoPucks`: Cantidad de robots diferenciales a simular.
- `EnablePucks`: Si únicamente se desea visualizar el movimiento de las partículas en un método dependiente de PSO, se permite que el usuario desactive los robots E-Puck al colocar `EnablePucks = 0`.

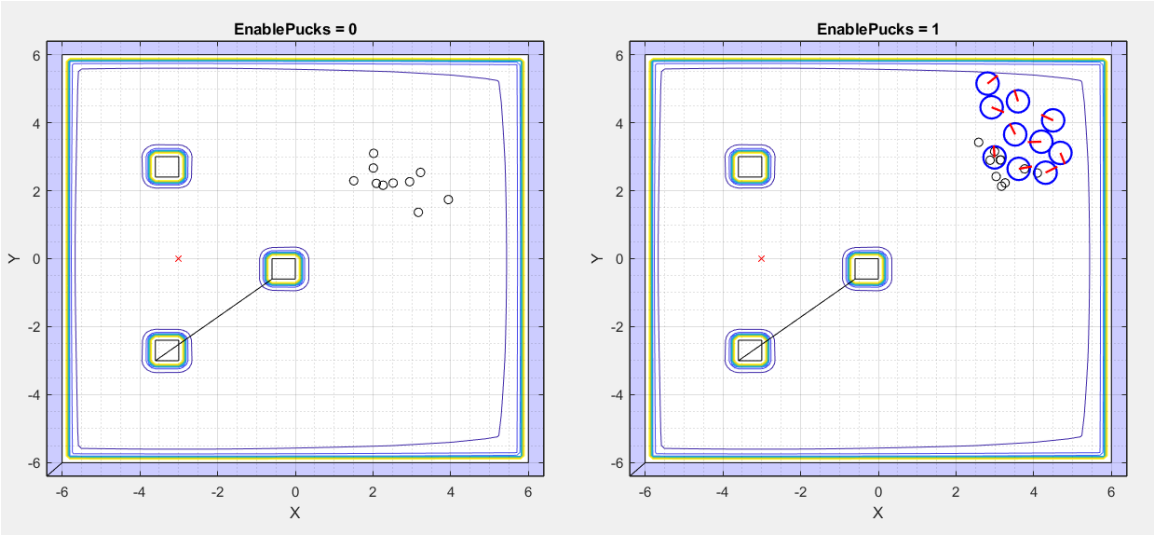

Figura 15: Efectos de alterar el parámetro `EnablePucks`.

- `RadioLlantasPuck`: Radio de las ruedas que emplea el robot diferencial. Unidades en metros.
- `RadioCuerpoPuck`: Distancia del centro del robot a sus ruedas. Unidades en metros
- `RadioDifeomorfismo`: Al momento de derivar la cinemática directa asociada con un robot diferencial, se hace evidente que el modelo derivado es altamente no lineal [1]. Para poder aplicar control a dicho robot, entonces se supone que no se controlará la posición y velocidad del centro del robot como tal, sino de un punto delante de él (Comúnmente, ubicado en los extremos de su radio en caso se trate de un robot circular). La distancia que existe entre el centro del robot y este punto a controlar se le denomina radio de difeomorfismo. Unidades en metros.
- `PuckVelMax`: Velocidad angular máxima que pueden alcanzar las ruedas del robot. Unidades en rad/s.
- `ControladorPucks`: Tipo de controlador a utilizar para el movimiento punto a punto de los E-Pucks. Existen 5 opciones: LQR, LQI, Lyapunov, Pose Simple y Closed-Loop

Steering. Entre estos, los dos mejores se consideran el LQI y LQR, con el peor siendo el de Closed-Loop Steering. Este último funciona, pero presenta dificultades que incluso se manifestaron y no pudieron ser resueltas dentro de la tesis de [1].

- `CriterioPuck`: Similar al parámetro de `CriterioPart`. Determina el criterio de convergencia que utilizará el *main loop* para determinar el momento en el que debe finalizar su ejecución.

8.4.7. Modo de Visualización de Animación

- `ModoVisualización`: `2D`, `3D` o `None`. El modo 3D se recomienda para observar más fácilmente la forma de la función de costo en métodos dependientes de PSO. El 2D es más útil para observar el movimiento de las partículas y/o robots.

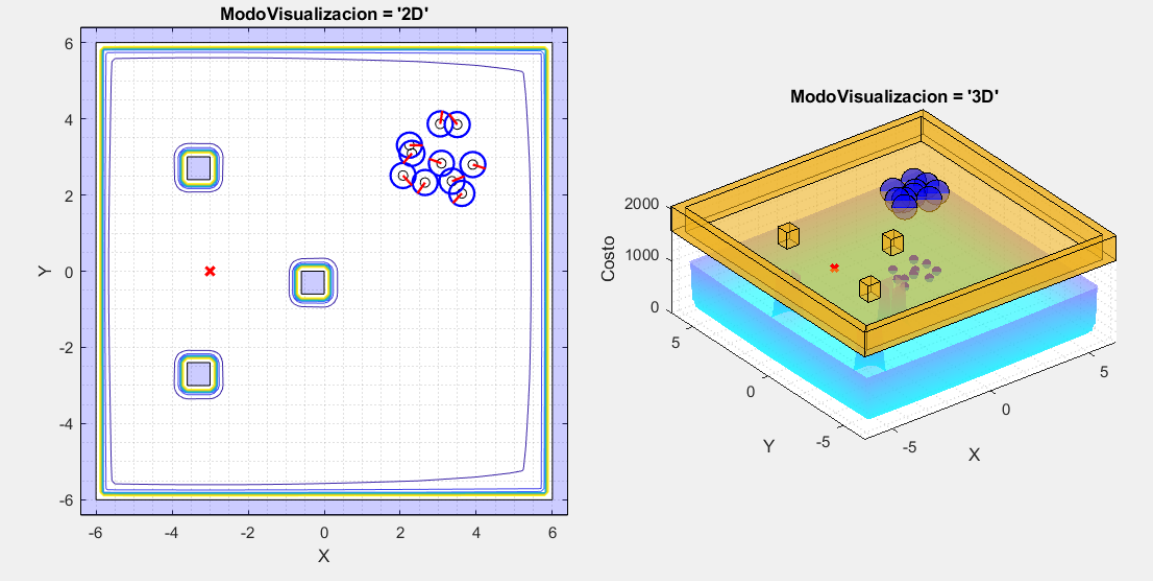

Figura 16: Efectos de alterar el parámetro `ModoVisualizacion`.

- `EnableRotacionCamara`: Parámetro binario únicamente válido para el modo de visualización 3D. Cuando Matlab grafica en 3D, este elige un ángulo óptimo para posicionar la *cámara* que enfoca el plot. Al habilitar esta opción, Matlab gira la cámara alrededor del plot a una velocidad constante.

- `VelocidadRotacion`: Velocidad o cantidad de grados que se mueve la cámara alrededor del plot por iteración del *main loop*. Mientras más bajo el valor absoluto de esta cantidad más lenta será la rotación. Por defecto, la cámara rota a favor de las manecillas del reloj. Si se desea que rote en contra de las manecillas, la velocidad debe consistir de un valor negativo.

8.4.8. Guardado para Animación

- `SaveFrames`: Parámetro binario. Si está habilitado, permite guardar la animación actual como una secuencia de imágenes independientes tipo PNG. Todas las imágenes son colocadas dentro del folder *Output Media\Frames*Nombre de simulación actual\. Útil para crear GIFS dentro de Latex utilizando el paquete *animate*.
- `FrameSkip`: La funcionalidad de `Save Frames` guarda una gran cantidad de imágenes por simulación. Para aliviar un poco la cantidad de imágenes guardadas, se puede elegir cuantas iteraciones deben pasar desde la última frame capturada, para capturar el siguiente frame. Valor recomendado: 3 frames.
- `SaveVideo`: Parámetro binario. Si está habilitado, permite guardar la animación actual como un video de 30 FPS en formato *.mp4*. El video resultante es colocado dentro del folder *Output Media\Video*.
- `SaveGIF`: Parámetro binario. Permite guardar la animación actual como una imagen animada tipo GIF. El GIF resultante es colocado dentro del folder *Output Media\GIFs*.

8.4.9. Seed Settings

En programación, una seed consiste de un número utilizado para inicializar un generador pseudo-aleatorio de números. En el caso de Matlab, la seed es el valor que utiliza Matlab para generar los números aleatorios al momento de llamar funciones como `randn()` o `randi()`. Si al inicio del programa se le provee una seed fija a Matlab, entonces cada vez que se llame a una función que genere valores aleatorios, se generarán los mismos resultados. En otras palabras, el usuario estará asegurando la replicabilidad de sus resultados. Si no se elige una seed explícitamente, Matlab utiliza una variedad de parámetros para generar una seed aleatoria. A continuación se listan algunos parámetros relacionados con la seed a utilizar:

- `SeedManual`: Parámetro binario. Si está habilitado, el usuario sobrescribe la seed seleccionada automáticamente por Matlab.
- `Seed`: Valor para la seed a utilizar en caso el usuario decida sobrescribir la seed utilizada por defecto por Matlab. Si `SeedManual` está deshabilitado, entonces `Seed` guarda el valor de la seed aleatoria elegida por Matlab.

8.5. Reglas de Método a Usar

Como se mencionó previamente, en el Toolbox, los robots pueden recorrer la mesa de trabajo utilizando 3 tipos de metodologías:

- Seguimiento de Trayectoria: Un algoritmo toma la forma del ambiente a recorrer y luego genera una trayectoria desde el punto de partida hasta la meta. Un controlador

de seguimiento de puntos luego se encarga de controlar el robot móvil hasta que llegue a la meta.

- Exploración con PSO: Los robots son liberados en el ambiente a explorar y estos lo recorren hasta finalmente alcanzar la meta. No se requiere conocimiento previo sobre el ambiente. Hace uso de PSO.
- Exploración Dinámica: Muy similar al PSO, pero obviamente sin la utilización de dicho algoritmo. No se requiere de conocimiento previo sobre el ambiente.

Cada método cuenta con sus particularidades y reglas, entonces en esta sección, el código revisa las listas de métodos disponibles y luego procede a aplicar las diferentes condiciones y reglas propias a cada caso. Por ejemplo, si el método utilizado es dependiente de PSO, se toma nota del tipo de método actual[4], activa la bandera `isPSO` y luego se establece si el método consiste de una función de costo *Benchmark* por medio de la bandera `isBenchmark`.

En caso el usuario desee agregar nuevos métodos, todas las decisiones de alto nivel sobre su ejecución deben de colocarse en esta sección. En particular, es muy importante que el usuario agregue el nombre de su método a una de las listas de métodos al inicio de esta sección.

8.6. Región de Partida y Meta

La partículas y robots son colocados por primera vez en la mesa de trabajo dentro de una *región de partida*. Su posición dentro de dicha región es aleatoria y uniformemente distribuida. El usuario puede modificar el centro de la región de partida (`Centro_RegionPartida`), así como la dispersión de la región (`Dispersión_RegionPartida`). Con dispersión, se hace referencia a que tan a la derecha/izquierda y arriba/abajo, pueden extenderse las partículas si se toma como origen el centro de la región.

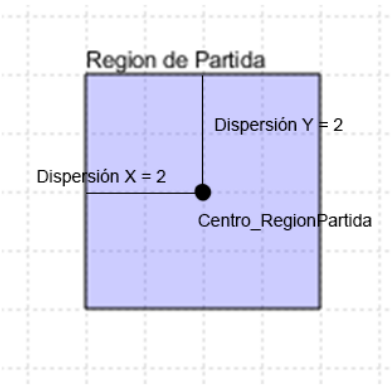

Figura 17: Explicación visual de cómo funciona la dispersión para la región de partida.

[4]Se guarda un string asociado al tipo de método, ya sea *PSO*, *Trayectoria* o *Dinamico*. Si el método es de carácter mixto, entonces se guarda un array que contiene cada uno de los métodos involucrados en su ejecución

Para la meta, el usuario tiene una variedad de opciones. Si el método actual consiste de un método de tipo *Trayectoria*, el usuario puede elegir si desea que las trayectorias sean *Multi-meta* o de meta única. Si se elige *Multi-meta*, cada E-Puck sigue una trayectoria diferente y va cambiando a su siguiente meta según vaya alcanzando su meta actual. En este caso, el array que contiene las trayectorias consiste de un array tridimensional, donde cada fila corresponde a las coordenadas (X,Y) de la meta de un E-Puck (Deben existir tantas filas como E-Pucks, ya que debe existir una meta para cada E-Puck) y cada *capa* a lo largo de la tercera dimensión, consiste del siguiente punto a seguir en la trayectoria.

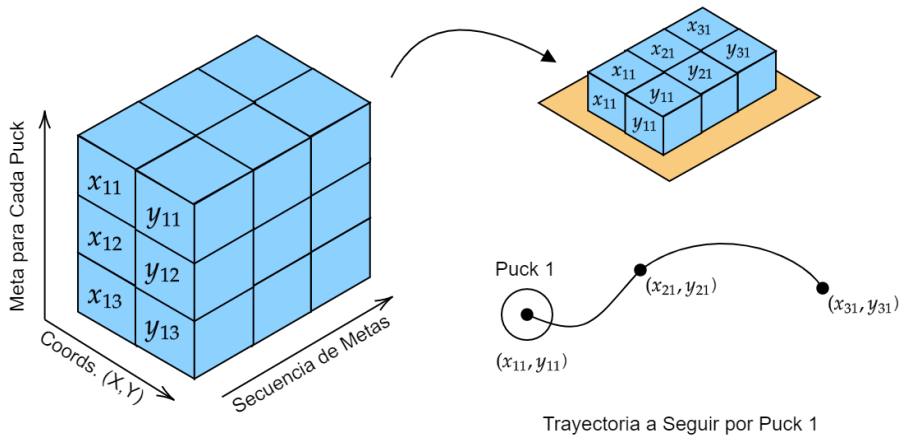

Figura 18: Estructura del vector de trayectorias para el caso *Multi-meta*

Si se elige una meta única, todos los E-Pucks siguen a un mismo punto y una vez su distancia promedio alcanza un cierto threshold, la meta cambia al siguiente punto en la trayectoria. En este caso el array que contiene las trayectorias consiste de un array bidimensional, con cada fila representando una nueva meta en la secuencia.

Ahora bien, si el método seleccionado consiste de una función de costo de tipo *Benchmark*, se ignoran las metas declaradas explícitamente por el usuario y el script realiza un sweep sobre toda la superficie de costo, revisando cuales son los puntos de menor costo. Estos puntos de costo mínimo (Los mínimos de la función) se toman como las metas de la simulación. Los agentes pueden elegir libremente a cual meta dirigirse.

8.7. Obstáculos en Mesa de Trabajo

Para probar las capacidades de navegación de los robots, el Toolbox posee múltiples herramientas para diseñar y posicionar obstáculos en la mesa de trabajo. A continuación se presentan las diferentes opciones disponibles.

8.7.1. Polígono

El usuario puede dibujar un polígono que desee posicionar en la mesa de trabajo. La interfaz en la que se dibuja el obstáculo también incluye la región de partida y el/los puntos meta para que el usuario evite colocar el obstáculo sobre estos (Aunque aún puede hacerlo). Para cerrar el polígono y finalizar la creación del obstáculo, se puede dar doble click en cualquier parte del plot o se puede hacer click sobre el primer vértice colocado.

Con esta herramienta únicamente se puede crear un único obstáculo para la mesa de trabajo[5] (No importando su complejidad). Debido a esto, el usuario debe ser cuidadoso al momento de crear el polígono. Si se desean crear múltiples polígonos o figuras personalizadas en pantalla, se recomienda utilizar la herramienta de *Imagen*.

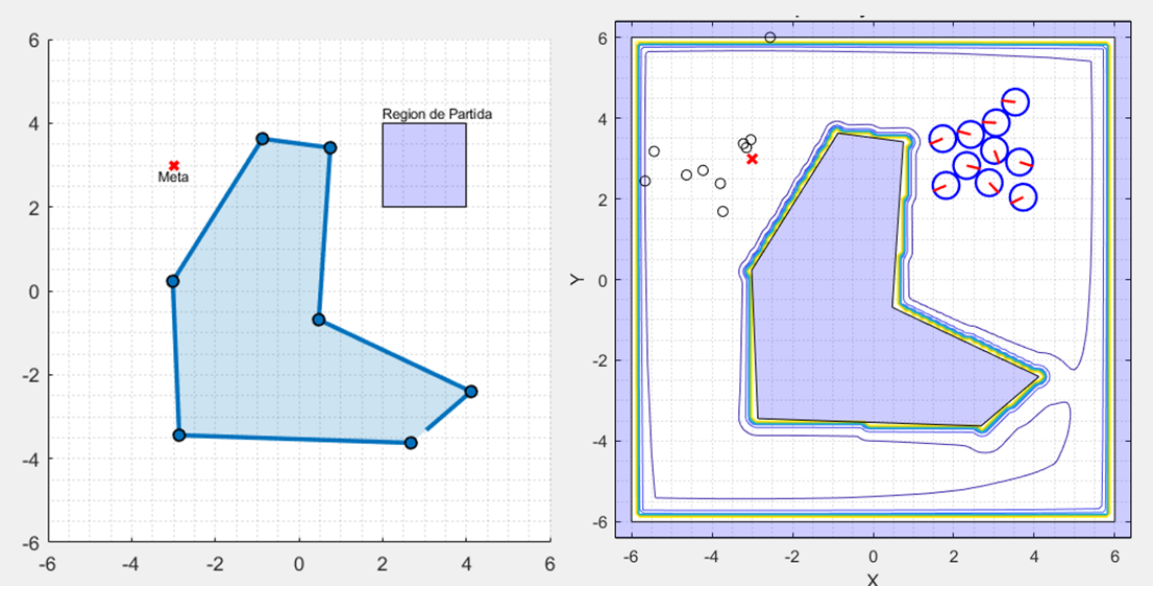

Figura 19: Creación de obstáculo poligonal.

8.7.2. Cilindro

Cilindro: Coloca un cilindro en el centro de la mesa de trabajo. El radio puede cambiarse manualmente alterando el parámetro `RadioObstaculo`. La altura del obstáculo en su vista 3D puede ser alterada usando el parámetro `AlturaObstaculo`.

8.7.3. Imagen

El usuario puede tomar una imagen en blanco y negro de un mapa (Con los obstáculos en negro y el espacio vacío en blanco), colocarla en el directorio base del script principal (O

[5]Se intentó implementar una funcionalidad que permitiera crear múltiples polígonos. La idea era que el usuario presionara un botón y se rehabilitara la opción para dibujar un polígono. Desafortunadamente, el manejo de elementos GUI dentro de un script tradicional es relativamente complicada, por lo que se optó por abandonar la idea.

dentro de la carpeta *Mapas\Imágenes*) y luego procesarla para convertirla en un obstáculo utilizable dentro del Toolbox.

Para su funcionamiento, esta herramienta hace uso de la función `ImportarMapa.m`. Dicha función toma como entrada una imagen y a través de una variedad de operaciones, extrae los vértices de los obstáculos presentes en la imagen. Estos pueden ser luego utilizados en el script principal para graficar los obstáculos y calcular otros aspectos adicionales como la función de costo asociada al mapa (En el caso del método `APF` o `Jabandzic`).

Dado que este proceso puede llegar a ser altamente intensivo para el sistema dependiendo de la complejidad del obstáculo, la función cuenta con medidas adicionales para poder revisar si ya existen datos previamente procesados sobre la imagen proporcionada por el usuario. Si ya existen vértices asociados con la imagen proporcionada, el usuario puede elegir reutilizar los datos guardados para así evitar la carga computacional asociada. También se incluyen medidas para revisar el nivel de similitud de la imagen suministrada, con el de las imágenes guardadas. Si es lo suficientemente parecido, el programa nuevamente pregunta si el usuario desea reutilizar datos previos.

Si se desea comprender más a profundidad la forma en la que funciona dicha función (O refinar el sinnúmero de parámetros de los que depende la función), se provee una versión alternativa en formato `.mlx` que presenta figuras y métodos alternativos para realizar el mismo proceso de extracción de vértices.

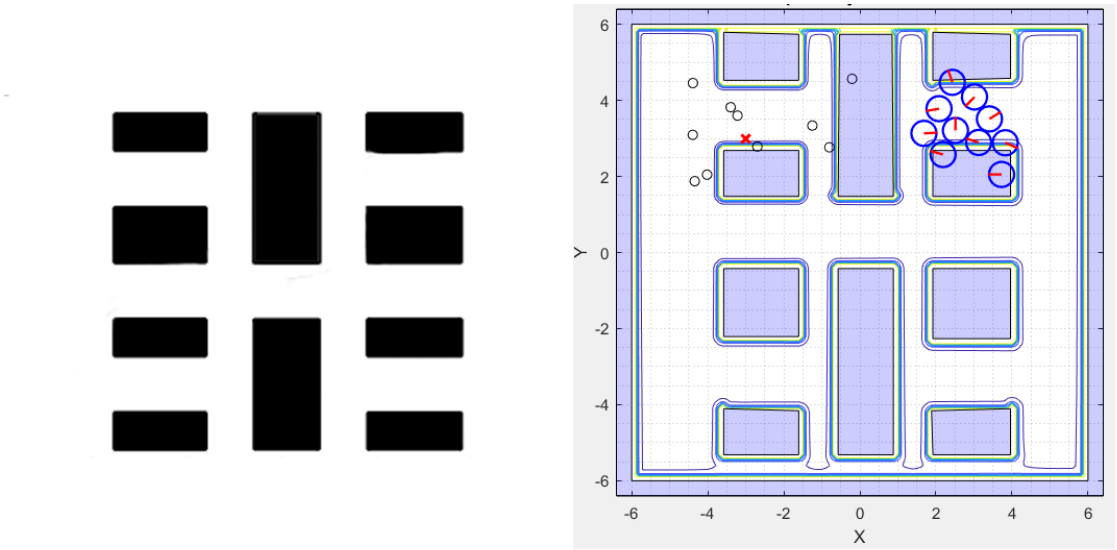

Figura 20: Creación de obstáculos basados en una imagen en blanco y negro.

8.7.4. Caso A

Réplica del escenario A utilizado en la tesis de Juan Pablo Cahueque [2].

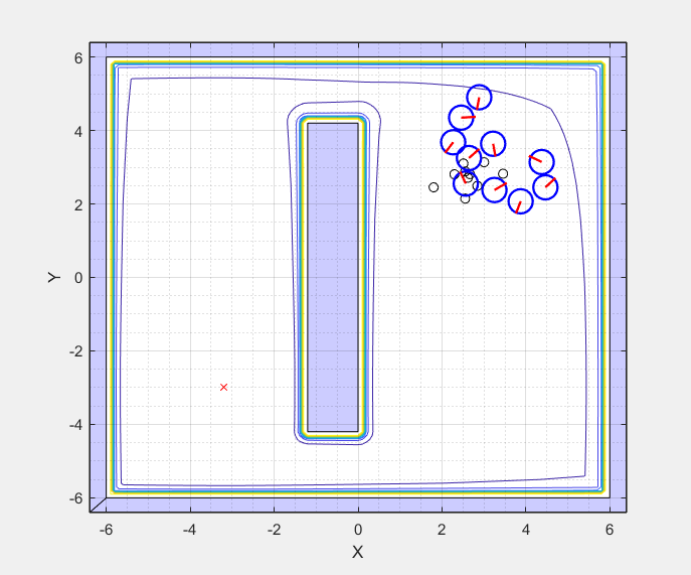

Figura 21: Caso A en tesis de Juan Pablo Cahueque

8.7.5. Caso B

Réplica del escenario B utilizado en la tesis de Juan Pablo Cahueque [2].

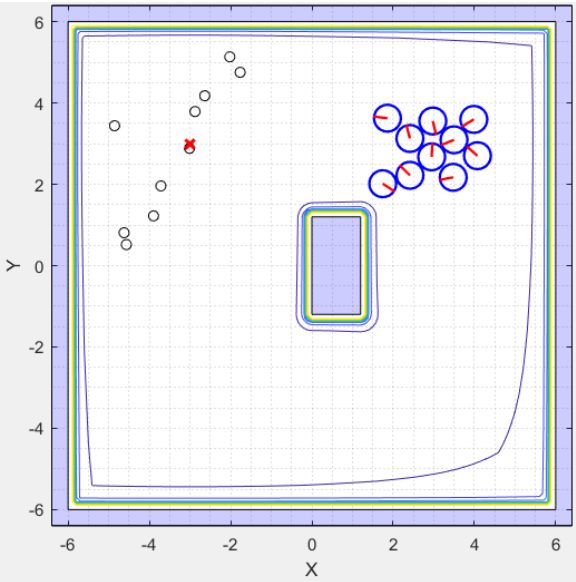

Figura 22: Caso B en tesis de Juan Pablo Cahueque

8.7.6. Caso C

Réplica del escenario C utilizado en la tesis de Juan Pablo Cahueque [2].

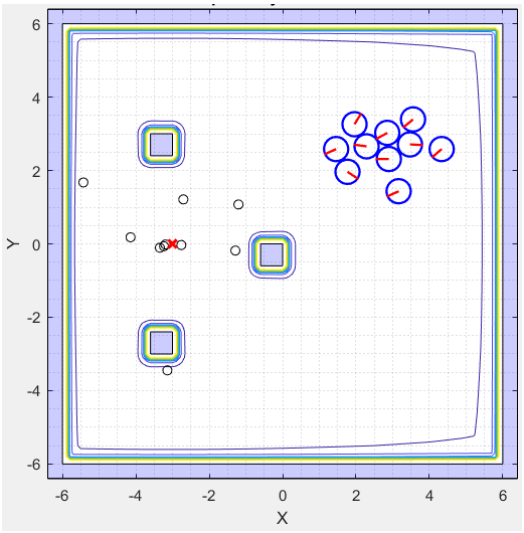

Figura 23: Caso C en tesis de Juan Pablo Cahueque

8.8. Setup - Métodos PSO

Todos los métodos PSO requieren de una inicialización previa para elementos como su función de costo, posición inicial de sus partículas, entre otros.

8.8.1. Posición Inicial de Partículas

Las partículas del algoritmo PSO se distribuyen de manera aleatoria y uniformemente distribuida dentro de la región de partida. A pesar de esto, el usuario puede suministrarle cualquier posición inicial a la clase `PSO.m` y esta lo procesará de manera acorde. En el caso bidimensional tratado en los métodos dependientes de PSO, se asume que las partículas se mueven sobre el plano (X,Y).

8.8.2. Parámetros Ambientales

Según el tipo método elegido, la evaluación de la función de costo asociada puede llegar a requerir de más o menos parámetros. Funciones *Benchmark* no requieren parámetros adicionales, pero métodos como APF (Artificial Potential Fields) o Jabandzic requieren de parámetros propios del ambiente como los vértices de los obstáculos (`XObs` / `YObs`), el tamaño de la mesa de trabajo (`LimsX` / `LimsY`), el punto de meta, la posición actual del E-Puck y la posición actual de cualquier obstáculo dinámico presente en la simulación (Obstáculos móviles).

Para darle mayor flexibilidad al usuario al momento de implementar nuevos métodos, este puede definir la cantidad de parámetros adicionales que le desea pasar a la función de costo a utilizar. Los cambios en el input de la función se pueden tomar en cuenta por medio del *input parser* de la función `CostFunction.m`

8.8.3. Búsqueda Numérica del Mínimo de la Función de Costo

Se extrae una muestra de todas las parejas coordenadas presentes en el espacio de trabajo y se evalúan dentro de la función de costo elegida. Esto genera la superficie de costo correspondiente a la función. En el caso de funciones *Benchmark*, esto se utiliza únicamente para determinar las metas de la función.

Para el método de Artificial Potential Fields (APF), esta evaluación preliminar permite almacenar en memoria la superficie de costo completa, para que en llamadas posteriores a la función `CostFunction.m`, esta únicamente se encargue de extraer los datos de la superficie de costo previamente procesada. En el caso del método de Jabandzic, este debe generar de manera dinámica su superficie de costo, por lo que luego de este barrido, se ignoran los valores adquiridos por diferentes variables persistentes internas. Para ambos casos (APF y Jabandzic), las metas declaradas en la sección de *Región de Partida y Meta* permanecen inalteradas.

8.8.4. Inicialización de PSO

Una vez calculada la superficie de costo, e inicializadas las posiciones de las partículas, se crea un objeto de la clase *PSO*. Este objeto corresponde a una simulación del algoritmo, por lo que al crearlo por primera vez, la clase realiza los ajustes respectivos de forma interna. Para más información sobre las propiedades y métodos de esta clase escribir en consola `help PSO`. Un aspecto importante a tomar en cuenta es que, previo a llamar al método `PSO.RunStandardPSO()`, el usuario debe primero configurar las restricciones a utilizar. De aquí la sección siguiente.

8.8.5. Coeficientes de Constricción e Inercia

En el *toolbox* se ofrecen tres opciones de restricción para las constantes de la ecuación de actualización de la velocidad del algoritmo PSO:

- Inercia: Si se desea abandonar el esquema que asegura la convergencia propuesto por Clerc [15], el usuario puede obviar la ecuación y utilizar el valor de , o inercia, que desee. Se ofrecen 5 tipos diferentes de inercia. Para más información escribir en consola: `help ComputeInertia`
- Inercia: Si se desea abandonar el esquema que asegura la convergencia propuesto por Clerc [15], el usuario puede obviar la ecuación y utilizar el valor de , o inercia, que desee. Se ofrecen 5 tipos diferentes de inercia. Para más información escribir en consola: `help ComputeInertia`

$$\begin{aligned} \omega &= \chi \qquad \chi = \frac{2\kappa}{\mid 2 - \phi - \sqrt{\phi^2 - 4\phi} \mid} \\ C_1 &= \chi\phi_1 \qquad\qquad \phi = \phi_1 + \phi_2 \\ C_2 &= \chi\phi_2 \end{aligned} \tag{32}$$

- Mixto: Uso simultáneo de un tipo de inercia (Por defecto exponencialmente decreciente), en conjunto con los parámetros de constricción propuestos por Clerc. Utilizado por Aldo en su tesis.

8.9. Setup - Gráficas

Las figuras en la Toolbox, son todas manejadas por medio de `handlers`. Esto implica que todas las gráficas se generan por primera vez previo a iniciar la ejecución del algoritmo y una vez se ingresa al *main loop*, únicamente se actualizan las propiedades asociadas a cada una de las gráficas. Esto permite la inclusión de un mayor número de elementos en la animación, sin afectar de manera significativa el rendimiento de las animaciones.

El tamaño de la figura puede configurarse y su posición se ajusta para siempre coincidir con el centro de la pantalla del equipo al ejecutar el script. Esta figura tiene 3 partes: La leyenda, la descripción y la región de simulación.

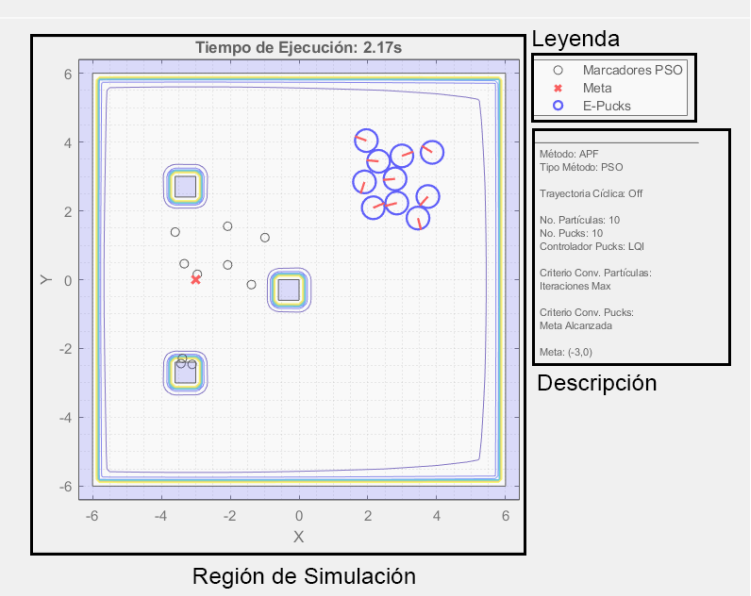

Figura 24: Partes de la figura de simulación

La descripción lista las características actuales de la simulación y debe ser redactada manualmente por el usuario bajo su conveniencia. Por defecto, esta incluye información como el método utilizado, el tipo de método, el número de partículas y robots, los criterios de convergencia, entre otros[6].

La región de simulación, es la parte más importante y es la que despliega la representación gráfica de todo el procesamiento realizado. Aquí se presentan los obstáculos presentes en la mesa (En color azul claro), los robots E-Puck (Círculos azules, en caso se encuentre habilitado el parámetro `EnablePucks`), las funciones de costo asociadas en caso el método

[6]Una función bastante útil para el usuario al momento de redactar esta descripción es `bin2OnOff()`. Esta permite que el usuario incluya variables binarias en su descripción y la función lo traduce en un string correspondiente al estado (1 para `On` y 0 para `Off`.

sea dependiente de PSO (Graficada como un `surf` en el modo 2D y como un `surf3` para el modo 3D), las partículas, entre otros. Además, el título cambia de manera dinámica para presentar el tiempo en segundos que ha transcurrido desde el inicio de la simulación.

Finalmente, se cuenta con la leyenda. Esta indica que representa cada elemento en la región de simulación y puede ser alterada de forma sencilla por el usuario. Previo a generar todos los plots asociados con la región de simulación, se definen dos arrays vacíos: `LeyendaTexto` y `LeyendaHandles`. En caso el usuario desee que uno de los elementos graficados se incluya en la leyenda, este debe realizar un *append* del handle (`NombrePlot(1)`) de dicho elemento a `LeyendaHandles` y un append del nombre que se desea desplegar para dicho elemento en `LeyendaTexto`.

```
% Meta a alcanzar
PlotPuntoMeta = scatter(Meta(:,1), Meta(:,2), 60, 'red', 'x','LineWidth',2);
LeyendaTexto = [LeyendaTexto "Meta"];
LeyendaHandles = [LeyendaHandles PlotPuntoMeta(1)];
```

Figura 25: Ejemplo: Inclusión de la meta en la leyenda de la gráfica

Al finalizar, la leyenda incluirá todos los elementos incluidos. Este sistema se diseñó específicamente con el propósito de brindar mayor modularidad al sistema de leyendas. Además, también se aseguró que la descripción de la gráfica se ajustara para siempre presentarse a cierta distancia por debajo de la leyenda no importando el tamaño de la misma.

8.10. Main Loop

El *main loop* del simulador se separa en diferentes secciones de ejecución secuenciales:

1. En caso el método seleccionado sea dependiente de PSO, se simulan todos los elementos relacionados al algoritmo. Esto incluye actualizar los parámetros ambientales requeridos por las funciones de costo, correr el propio algoritmo como tal (Por medio del método `RunStandardPSO()`), y actualizar la nueva meta para los robots.

2. En caso el método seleccionado haga uso de seguimiento de trayectorias, se toman las posiciones actuales de los Pucks, y se analiza (Según sea el caso) la cercanía de estos hasta la meta. Si están lo suficientemente cerca a su meta respectiva, entonces el programa cambia a una nueva meta y se actualiza la meta actual.

3. Se actualiza la dinámica de los E-Pucks: Se resuelve cualquier colisión que haya ocurrido en la iteración previa, se obtiene el output de los controladores para cada robot y se actualiza la posición y orientación de los mismos. También se toma nota de las posiciones, velocidades y orientaciones de cada robot y se guarda cada uno en un historial diferente.

4. Se actualizan los handles de todos los elementos gráficos de la simulación (En caso el modo de visualización no sea *None*), así como el título para reflejar el avance de tiempo.

5. Se evalúan los criterios de convergencia correspondientes a los E-Pucks. Se determina si no hay que detener la ejecución y si no se detiene, se guarda la frame actual en el medio de salida actual elegido (Video, GIF o secuencia de imágenes).

En el momento en el que el *main loop* finaliza, el algoritmo grafica las trayectorias seguidas por cada robot o partícula (Se grafican las trayectorias de partículas si `EnablePucks = 0`) y se actualiza la leyenda. También se finaliza la creación de la grabación actual y se ajusta la posición de la descripción para tomar en cuenta cualquier incremento en el tamaño de la leyenda.

8.11. Análisis de Resultados

Al finalizar la simulación, se pueden generar 4 gráficas informativas

8.11.1. Evolución del Global Best

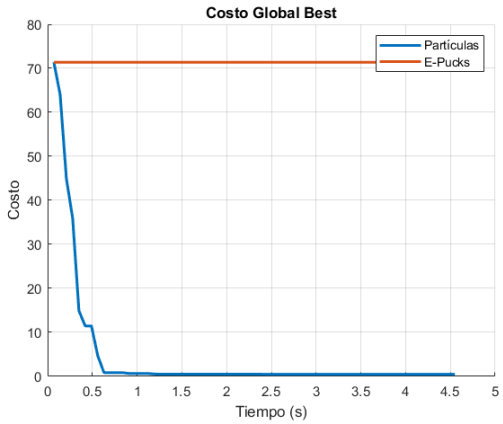

Figura 26: Evolución de la minimización hacia el global best de la función

Utilizada para determinar si las partículas efectivamente minimizan la función de costo que se eligió. Si se conoce cual es el costo mínimo de la función, esta gráfica puede utilizarse para determinar si las partículas alcanzaron el mínimo global o un mínimo local.

8.11.2. Análisis de Dispersión de Partículas

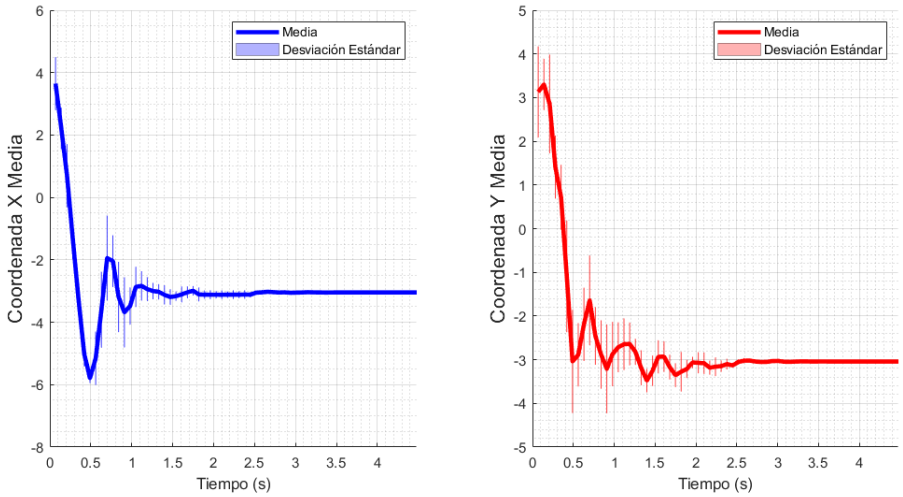

Figura 27: Dispersión de las partículas sobre el eje X y Y.

Dos cualidades importantes de las partículas del PSO es su capacidad de exploración y la precisión de su minimización. Con estas gráficas, la precisión se puede evaluar viendo la línea gruesa coloreada y la exploración utilizando las líneas correspondientes a la desviación estándar. Si las líneas gruesas se estabilizan en las coordenadas de la meta, las partículas son precisas. Si la desviación estándar es muy pronunciada, las partículas exploran minuciosamente el área de trabajo antes de converger. En el caso presentado, por ejemplo, las partículas son precisas y convergen con rapidez, aunque exploran poco.

8.11.3. Velocidad de Motores

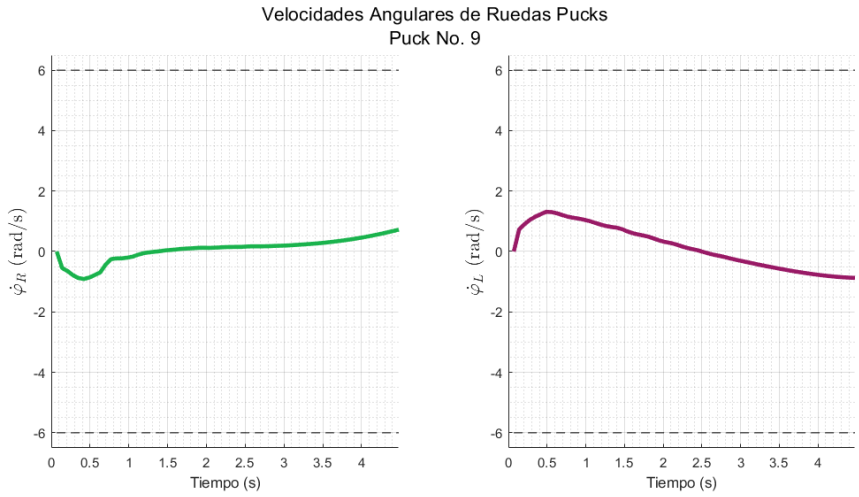

Figura 28: Velocidad angular observada en los motores del puck con los picos más altos de velocidad en dicha corrida

Utilizando la cinemática inversa de un robot diferencial se calculan las velocidades an-

gulares de las ruedas de todos los robots (Ecuación 33). El Toolbox obtiene las velocidades angulares medias de todas las ruedas y determina cual fue el robot con las velocidades más altas. Toma este robot como selección y grafica la evolución de las velocidades angulares de sus dos ruedas. Útil para analizar si los actuadores del robot crítico presentan saturación. Como ayuda se incluyen líneas punteadas, las cuales consisten de los límites de velocidad con los que cuenta el robot (Basado en `PuckVelMax`).

8.11.4. Suavidad de Velocidades

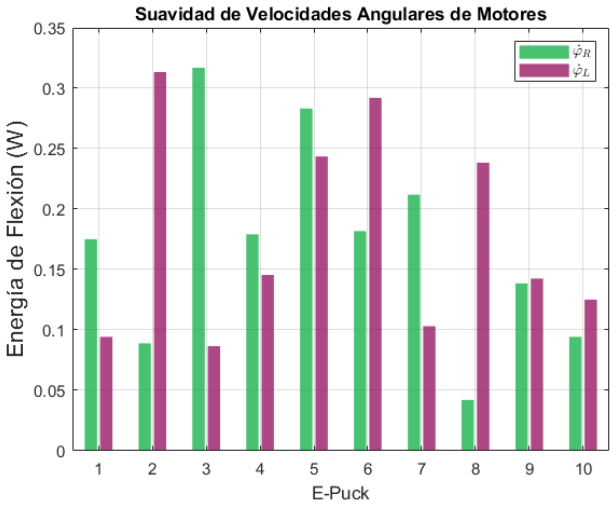

Figura 29: Energía de flexión observada en las velocidades angulares de las ruedas de cada puck.

Basado en el criterio de evaluación empleado por Aldo en su tesis. Se realiza una interpolación de los puntos que conforman la curva de velocidades angulares de las ruedas, y luego se calcula la energía de flexión de la curva. Si la energía de flexión es baja, la suavidad de operación es mucho mayor. Prueba ideal para diagnosticar cuantitativamente la suavidad de operación.

8.12. Colisiones

Un entorno de simulación realista es necesario para obtener resultados útiles al momento de realizar pruebas. Debido a esto, se implementó «Collision Detection» entre los robots. Durante cada iteración, los robots revisan la distancia entre ellos (Para más información escribir en consola: help getDistsBetweenParticles) y si esta es menor a 2 radios de E-Puck, los robots se clasifican como «en colisión». Seguido de esto se procede resolver las colisiones, alejando a los robots el uno del otro hasta eventualmente resolver todas las colisiones existentes.

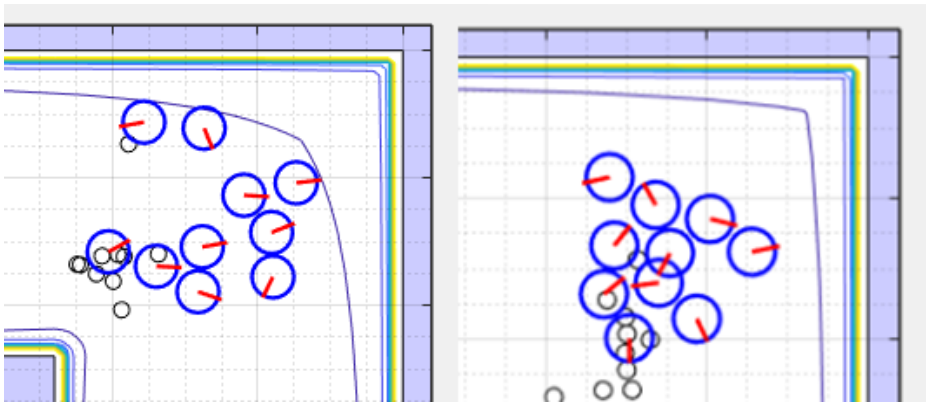

Figura 30: Con solución de colisiones (Izquierda) y sin solución de colisiones (Derecha)

Desgraciadamente, debido a que al alejar un robot del otro se pueden llegar a crear más colisiones, en algunas ocasiones el algoritmo puede no converger en una solución. Por lo tanto, **el algoritmo implementado es inestable y si no se restringe puede llegar a trabar Matlab**. Para controlar esto se le colocó un número máximo de iteraciones en las que puede llegar a producir una solución válida. Con esta «solución», el algoritmo funciona relativamente bien aunque puede producir errores frecuentemente.

Si se desea, el usuario puede acceder a la función SolveCollisions.m y cambiar el parámetro IteracionesMax. Los errores disminuyen al incrementar el número de iteraciones, pero el tiempo computacional requerido incrementa. En futuras versiones del Toolbox se desea implementar un algoritmo de «Collision Detection» mucho más robusto como «Speculative Collisions» que también incluya elementos como las paredes o los obstáculos como tal.

8.13. Controladores

Una de las formas más simples de movilidad para un robot diferencial consiste en el seguimiento punto a punto. En este, el robot diferencial se acerca a una meta puntual presente en el plano de trabajo y una vez se encuentra lo suficientemente cercano a la misma, el robot simplemente se detiene o busca una nueva meta. Para alcanzar dicha meta puntual, el robot hace uso de una variedad de estrategias de control poder generar la velocidad lineal y angular del robot.

En la simulación de Matlab, estas cantidades pueden ser utilizadas directamente para dictar la posición y orientación de los robots. No obstante, esto es posible debido a que se trata de una abstracción de la realidad. En el caso de un robot real, el usuario comúnmente tiene control sobre las velocidades angulares de cada una de las ruedas del robot. Para obtener estas velocidades a partir de la velocidad lineal y angular del robot, se emplean las siguientes ecuaciones

$$\dot{\varphi}_R = \frac{2v + 2\omega l}{2r}, \quad \dot{\varphi}_L = \frac{2v - 2\omega l}{2r} \tag{33}$$

En el Toolbox se ofrecen 5 controladores diferentes, cada uno basado en una estrategia de control distinta: LQR, LQI, controlador de pose simple, controlador de pose con criterio de estabilidad de Lyapunov y controlador de direccionamiento por lazo cerrado. Estos fueron basados en los controladores diseñados por [1]. Debido a esto, los mismos se comportan de manera muy similar a los resultados reportados en su tesis. Esto implica que los mejores controladores (En términos de su suavidad en velocidades) son el LQR y el LQI, y el peor (El cuál incluso presentó problemas de implementación) consistió del controlador de direccionamiento por lazo cerrado. Para más información escribir en consola `help getControllerOutput`.

8.14. Criterios de Convergencia

La función `EvalCriteriosConvergencia.m` permite que el usuario genere una señal binaria de parada según el estado actual de la simulación. Específicamente, el usuario puede evaluar uno de tres criterios disponibles:

1. `Meta Alcanzada`: Cierto porcentaje de entidades ha alcanzado la o las metas colocadas. El usuario puede alterar el threshold de cercanía a meta utilizado y el porcentaje de entidades que deben alcanzar su meta respectiva para enviar una señal binaria positiva.
2. `Entidades Detenidas`: Cierto porcentaje de entidades se han quedado «quietas» o se han movido poco desde la última iteración. Se puede modificar el threshold de diferencial de distancia que debe de existir entre iteraciones para que la entidad se considere quieta y el porcentaje de entidades que deben estar quietas para finalizar el algoritmo.
3. `Iter Máx Alcanzadas`: Se ha alcanzado el número máximo de iteraciones.

El uso de la palabra *entidades* en lugar de robots o partículas es intencional, y se debe a que esta función permite la evaluación de criterios de convergencia tanto para las partículas del algoritmo PSO, como para los robots móviles que se desplazan en el espacio de trabajo. La única diferencia radica en las posiciones suministradas a la función. Si se van a evaluar los criterios de una partícula, se usan las posiciones de una partícula, mientras que si se van a evaluar los criterios de un E-Puck, se usan las posiciones los E-Pucks. Para más información sobre entradas, salidas y parámetros de la función escribir en consola `help EvalCriteriosConvergencia`

CAPÍTULO 9

Conclusiones

CAPÍTULO 10

Recomendaciones

- Implementar un modo manual en el PSO toolbox, el cual permita que el usuario controle como un robot a control remoto a los E-Pucks. Este modo puede ser utilizado luego para realizar «Inverse Reinforcement Learning», donde a partir de un ejemplo dado por los diseñadores, el algoritmo intenta deducir porque ese ejemplo dado consiste de la policy óptima a seguir.
- El objetivo de un agente con Reinforcement Learning es encontrar la policy óptima que genere la mayor recompensa a largo plazo. Existen métodos que permiten encontrar de manera estocástica una policy óptima, dotando a diferentes agentes con diferentes policies y luego utilizando algoritmos genéticos para poder seleccionar entre las mejores policies hasta eventualmente encontrar la óptima.

CAPÍTULO 11

Bibliografía

[1] A. Nadalini, «Investigación de Parámetros Utilizados en Algoritmos de Optimización de Enjambre», págs. 1-5, 2019.

[2] J. Cahueque, «Implementación de Enjambre de Robots en Operaciones de Búsqueda y Rescate», Tesis doct., Universidad del Valle de Guatemala, 2019.

[3] R. S. Sutton y A. G. Barto, *Reinforcement Learning, An Introduction*, Second, F. Bach, ed. Cambridge, Massachusetts: MIT Press, 2018, pág. 400, ISBN: 9780262039246.

[4] R. Eberhart y J. Kennedy, «New optimizer using particle swarm theory», en *Proceedings of the International Symposium on Micro Machine and Human Science*, 1995, págs. 39-43, ISBN: 0780326768. DOI: 10.1109/mhs.1995.494215.

[5] F. Mondada, M. Bonani, X. Raemy, J. Pugh, C. Cianci, A. Klaptocz, S. Magnenat, J.-c. Zufferey, D. Floreano y A. Martinoli, «The e-puck, a robot designed for education in engineering», en *Proceedings of the 9th conference on autonomous robot systems and competitions*, vol. 1, 2009, págs. 59-65.

[6] J. Castillo, «Diseñar e implementar una red de comunicación inalámbrica para la experimentación en robótica de enjambre», Tesis doct., 2019, pág. 54.

[7] A. Hernández, «Desarrollo e implementación de algoritmo de visión por computador en una mesa de pruebas para la experimentación con micro-robots móviles en robótica de enjambre», Tesis doct., 2019.

[8] A. Maybell y P. Echeverría, «Algoritmo de sincronización y control de sistemas de robots multi-agente para misiones de búsqueda», Tesis doct., Universidad del Valle de Guatemala, 2019.

[9] A. Nadalini, «Algoritmo Modificado de Optimización de Enjambre de Partículas (MPSO)», Tesis doct., Universidad del Valle de Guatemala, 2019.

[10] K. F. Uyanik, «A study on Artificial Potential Fields», vol. 2, n.º 1, págs. 1-6, 2011.

[11] M. Gromniak y J. Stenzel, «Deep Reinforcement Learning for Mobile Robot Navigation», *2019 4th Asia-Pacific Conference on Intelligent Robot Systems, ACIRS 2019*, págs. 68-73, 2019. DOI: 10.1109/ACIRS.2019.8935944.

[12] X. Lu, Y. Cao, Z. Zhao e Y. Yan, *Deep Reinforcement Learning Based Collision Avoidance Algorithm for Differential Drive Robot*. Springer International Publishing, 2018, págs. 186-198, ISBN: 9783319975863. DOI: 10.1007/978-3-319-97586-3. dirección: http://dx.doi.org/10.1007/978-3-319-97586-3%7B%5C_%7D1.

[13] M. Hüttenrauch, A. Oic y G. Neumann, «Deep reinforcement learning for swarm systems», *Journal of Machine Learning Research*, vol. 20, págs. 1-31, 2019, ISSN: 15337928. arXiv: 1807.06613.

[14] C. W. Reynolds, «Flocks, herds, and schools: A distributed behavioral model», *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 1987*, vol. 21, n.º July, págs. 25-34, 1987. DOI: 10.1145/37401.37406.

[15] M. Clerc, «The swarm and the queen: Towards a deterministic and adaptive particle swarm optimization», en *Proceedings of the 1999 Congress on Evolutionary Computation, CEC 1999*, vol. 3, 1999, págs. 1951-1957, ISBN: 0780355369. DOI: 10.1109/CEC.1999.785513.

[16] M. Clerc y J. Kennedy, «The Particle Swarm — Explosion , Stability , and Convergence in a Multidimensional Complex Space», *IEEE Transactions on Evolutionary Computation*, vol. 6, n.º 1, págs. 58-73, 2002.

[17] F. Chollet, *Deep Learning With Python*. Manning, 2018, pág. 373, ISBN: 9781617294433.

[18] S. Vieira, W. Pinaya y A. Mechelli, «Using deep learning to investigate the neuroimaging correlates of psychiatric and neurological disorders: Methods and applications», *Neuroscience Biobehavioral Reviews*, vol. 74, ene. de 2017. DOI: 10.1016/j.neubiorev.2017.01.002.

[19] A. Ng, *Sequence Models*.

[20] A. Mittal, *Understanding RNN and LSTM*, oct. de 2019. dirección: https://towardsdatascience.com/understanding-rnn-and-lstm-f7cdf6dfc14e.

[21] A. dprogrammer, *RNN, LSTM amp; GRU*, jun. de 2020. dirección: http://dprogrammer.org/rnn-lstm-gru.

[22] C. Lee, *Understanding Bidirectional RNN in PyTorch*, mar. de 2018. dirección: https://towardsdatascience.com/understanding-bidirectional-rnn-in-pytorch-5bd25a5dd66.

[23] K. Parsopoulos y M. Vrahatis, «Recent approaches to global optimization problems through Particle Swarm Optimization», *Natural Computing*, vol. 1, n.º 2/3, págs. 235-306, 2002, ISSN: 1567-7818. DOI: 10.1023/A:1016568309421.

[24] K. Mason, J. Duggan y E. Howley, «A meta optimisation analysis of particle swarm optimisation velocity update equations for watershed management learning», *Applied Soft Computing Journal*, vol. 62, págs. 148-161, 2018, ISSN: 15684946. DOI: 10.1016/j.asoc.2017.10.018. dirección: http://dx.doi.org/10.1016/j.asoc.2017.10.018.

[25] J. Kennedy, «Chapter 6 SWARM INTELLIGENCE», en *HANDBOOK OF NATURE-INSPIRED AND INNOVATIVE COMPUTING Integrating Classical Models with Emerging Technologies*, 2006, págs. 187-220, ISBN: 9780387405322.

[26] C. Eberhart e Y. Shi, «Comparing Inertia Weights and Constriction Factors in Particle Swarm Optimization», *IEEE*, n.º 7, págs. 84-88, 2000.

[27] J. C. Bansal, P. K. Singh, M. Saraswat, A. Verma, S. S. Jadon y A. Abraham, «Inertia weight strategies in particle swarm optimization», en *Proceedings of the 2011 3rd World Congress on Nature and Biologically Inspired Computing, NaBIC 2011*, 2011, págs. 633-640, ISBN: 9781457711237. DOI: 10.1109/NaBIC.2011.6089659.

[28] S. L. Brunton y J. N. Kutz, *Data-driven science and engineering: machine learning, dynamical systems, and control*. 2019, ISBN: 9781108422093. DOI: 10.1080/00107514.2019.1665103.

[29] A. B. Basnet, *LSTM Optimizer Choice ?*, ago. de 2017. dirección: https://deepdatascience.wordpress.com/2016/11/18/which-lstm-optimizer-to-use/.

CAPÍTULO 12

Anexos

12.1. Cosos cosos cosos